NASA/TP–2002–211640

Multi-dimensional Upwind Fluctuation Splitting Scheme with Mesh Adaption for Hypersonic Viscous Flow

William A. Wood
Langley Research Center, Hampton, Virginia

April 2002

The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621–0134

- Phone the NASA STI Help Desk at (301) 621–0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076–1320

NASA/TP–2002–211640

Multi-dimensional Upwind Fluctuation Splitting Scheme with Mesh Adaption for Hypersonic Viscous Flow

William A. Wood
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681–2199

April 2002

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076–1320
(301) 621–0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161–2171
(703) 605–6000

Abstract

Multi-dimensional Upwind Fluctuation Splitting Scheme with Mesh Adaption for Hypersonic Viscous Flow

By
William Alfred Wood III

Committee Chairman: Bernard Grossman
Aerospace Engineering

(ABSTRACT)

A multi-dimensional upwind fluctuation splitting scheme is developed and implemented for two-dimensional and axisymmetric formulations of the Navier-Stokes equations on unstructured meshes. Key features of the scheme are the compact stencil, full upwinding, and non-linear discretization which allow for second-order accuracy with enforced positivity. Throughout, the fluctuation splitting scheme is compared to a current state-of-the-art finite volume approach, a second-order, dual mesh upwind flux difference splitting scheme (DMFDSFV), and is shown to produce more accurate results using fewer computer resources for a wide range of test cases. The scalar test cases include advected shear, circular advection, non-linear advection with coalescing shock and expansion fans, and advection-diffusion. For all scalar cases the fluctuation splitting scheme is more accurate, and the primary mechanism for the improved fluctuation splitting performance is shown to be the reduced production of artificial dissipation relative to DMFDSFV. The most significant scalar result is for

combined advection-diffusion, where the present fluctuation splitting scheme is able to resolve the physical dissipation from the artificial dissipation on a much coarser mesh than DMFDSFV is able to, allowing order-of-magnitude reductions in solution time. Among the inviscid test cases the converging supersonic streams problem is notable in that the fluctuation splitting scheme exhibits superconvergent third-order spatial accuracy. For the inviscid cases of a supersonic diamond airfoil, supersonic slender cone, and incompressible circular bump the fluctuation splitting drag coefficient errors are typically half the DMFDSFV drag errors. However, for the incompressible inviscid sphere the fluctuation splitting drag error is larger than for DMFDSFV. A Blasius flat plate viscous validation case reveals a more accurate v-velocity profile for fluctuation splitting, and the reduced artificial dissipation production is shown relative to DMFDSFV. Remarkably the fluctuation splitting scheme shows grid converged skin friction coefficients with only five points in the boundary layer for this case. A viscous Mach 17.6 (perfect gas) cylinder case demonstrates solution monotonicity and heat transfer capability with the fluctuation splitting scheme. While fluctuation splitting is recommended over DMFDSFV, the difference in performance between the schemes is not so great as to obsolete DMFDSFV. The second half of the dissertation develops a local, compact, anisotropic unstructured mesh adaption scheme in conjunction with the multi-dimensional upwind solver, exhibiting a characteristic alignment behavior for scalar problems. This alignment behavior stands in contrast to the curvature clustering nature of the local, anisotropic unstructured adaption strategy based upon *a posteriori* error estimation that is used for comparison. The characteristic alignment is most pronounced for linear advection, with reduced improvement seen for the more complex non-linear advection and advection-diffusion cases. The adaption strategy is extended to the two-dimensional and axisymmetric Navier-Stokes equations of motion through the concept of fluctuation minimization. The system test case for the adaption strategy is a sting mounted capsule at Mach-10 wind tunnel conditions, considered in both two-dimensional and axisymmetric configurations. For this complex flowfield the adaption results are disappointing since feature alignment does not emerge from the local operations. Aggressive adaption is shown to result in a loss of robustness for the solver, particularly in the bow shock/stagnation point

interaction region. Reducing the adaption strength maintains solution robustness but fails to produce significant improvement in the surface heat transfer predictions.

Contents

List of Tables

List of Figures

Nomenclature

Geometric and independent variables

i, j	Computational indices
$\hat{\imath}, \hat{\jmath}, \hat{k}$	Cartesian unit vectors
J^{-1}	Inverse Jacobian of the coordinate transformation
L_{ref}	Non-dimensionalizing reference length
ℓ	Edge or element length
n	Outward normal vector
r	Distance vector
S	Generalized volume
t	Time
x, y, z	Cartesian coordinates, normalized by L_{ref}
Γ	Perimeter of generalized control volume
ξ, η	Curvilinear coordinates, normalized by L_{ref}
ϖ	Axisymmetric logical flag, $\varpi = 0, 1$
ϖ_a	Axisymmetric flag function, $\varpi_a = 1 - \varpi + \varpi y$
Ω	Generalized integration volume

Dependent variables

$\mathbf{A}$	Flux Jacobian in conservative variables
$\mathcal{A}$	Flux Jacobian in auxiliary variables
a	Sound speed
C_P	Pressure coefficient

c	Nodal update coefficients
c_p	Specific heat at constant pressure
c_v	Specific heat at constant volume
E	Total energy
E_r	Error estimate
e	Internal energy
$\mathbf{F}$	Flux function
$\mathbf{f}$	Numerical flux
H	Total enthalpy
h	Specific enthalpy
$\mathbf{M}$	Subsonic or supersonic matrix dissipation
M	Mach number
P	Pressure
q	Heat flow
$\mathbf{R}$	Area-weighted residual
s	First Riemann variable: $ds = d\rho - \frac{1}{a^2}dP$
T	Temperature
$\mathbf{U}$	Conserved variables
u, v, w	Cartesian velocities
V	Velocity
$\mathbf{V}$	Primitive variables
$\mathcal{V}$	Projected velocity
$\mathbf{W}$	Auxiliary variables
$\mathbf{X}$	Matrix of right eigenvectors
$\mathcal{X}$	Right eigenvectors in auxiliary variables
$\mathbf{Z}$	Parameter vector
α, β	Curvilinear advection speeds
κ	Thermal conductivity
λ	Wavespeed
Λ	Eigenvalue matrix
μ	Coefficient of viscosity

ρ	Density
$\boldsymbol{\tau}$	Stress tensor
Υ	Objective functional related to RMS of fluctuations
Φ	Artificial dissipation function
ϕ	Elemental fluctuation
$\breve{\phi}$	Fluctuation of auxiliary variables formulation
ϕ'	Elemental artificial dissipation
$\breve{\phi}'$	Artificial dissipation in auxiliary variables formulation
ϕ^*	Limited fluctuation

Auxiliary symbols

$\mathbf{D}$	Diagonal matrix
I	Identity matrix
M_ψ	Symmetric averaging function
P_r	Prandtl number, $P_r(\text{air}) = 0.72$
p, q	Arguments of limiter
Q	Fluctuation ratio
R_e	Reynolds number
$\Re$	Gas constant, $\Re(\text{air}) = 287$ J/(kg-K)
γ	Ratio of specific heats, $\gamma(\text{diatomic}) = 1.4$
ϵ	Eigenvalue limiting parameter
ε	Limiter parameter
Ξ	Weight-factor matrix for fluctuation minimization
υ	Finite element shape function
ψ	Limiter function

Operators

$\vec{\nabla}$	Gradient
∇^2	Laplacian, $\nabla^2 = \vec{\nabla} \cdot \vec{\nabla}$
Δ	Forward difference, $\Delta_i x = x_{i+1} - x_i$

∇	Backward difference, $\nabla_i x = x_i - x_{i-1}$
δ	Central difference, $\delta_i x = \frac{1}{2}(x_{i+1} - x_{i-1})$
δ^2	Second central difference, $\delta_i^2 x = \Delta\nabla_i x = \nabla\Delta_i x = x_{i+1} - 2x_i + x_{i-1}$
ϵ_{ijk}	Cyclic-permutation summation

Acronyms

COE	Contributions from other elements
DMFDSFV	Dual mesh flux difference splitting finite volume
LHS	Left-hand side
RHS	Right-hand side
RMS	Root mean square

Subscripts

E	Element
L	Left-hand state
R	Right-hand state
T	Triangle
U	Upwind
2U	Second-order upwind
∞	Reference values (typically equal to freestream values)
0	Stagnation value

Superscripts

i	Inviscid
n	Iteration level
sub	Subsonic
sup	Supersonic
v	Viscous
x,y,z	Cartesian component of a vector or tensor
ξ, η	Component associated with a curvilinear edge

Over-bars are used to represent average values. Vector symbols indicate vectors spanning multiple spatial dimensions. Bold face is used for vectors and tensors of systems. Subscripts of independent variables are short-hand for differentiation. Hats denote unit vectors. Tildes denote Roe-averaged quantities.

Chapter 1

Introduction

1.1 Objective

The present dissertation seeks to develop a next-generation numerical aerothermodynamic predictive capability. High-fidelity flowfield solutions are sought with increased accuracy and reduced analysis time, especially with respect to configuration changes. This is an effort to push the state of the art for aerothermodynamic analysis capability closer to being a usable tool for vehicle designers.

Specifically, the multi-dimensional upwinding concepts of Sidilkover[1, 2, 3] as applied to the Euler equations of inviscid, perfect gas flows are incorporated into a consistent treatment of viscous and heating terms to solve the Navier-Stokes equations of gas dynamics. A detailed analysis of the base algorithm is performed and applications to several validation cases are conducted. Also, aggressive unstructured mesh-adaption strategies are investigated in conjunction with the fluctuation splitting scheme.

1.2 Motivation

Current proposals for future United States access-to-space systems incorporate reusable trans-atmospheric flight vehicles, performing a role similar to the space shuttle orbiter, seeking to minimize recurring costs. For efficient design and operation of these

vehicles aerothermodynamic performance predictions are required to high accuracy and in a timely fashion.

The current computational capability to provide these aerothermodynamic performance predictions for complex vehicles is severely limited by total solution times, which include both domain discretization and flowfield evolution, measured in months. In order for high-fidelity aerothermodynamic predictive tools to play an active role in the design phase a leap in responsiveness must be made over the current state of the art methods, represented by dimensionally-split approximate Riemann solvers.

An effective numerical aerothermodynamic predictive tool must be able to provide rapid analysis of vehicle aerodynamics and control surface effectiveness. To do so flowfield features, such as embedded shocks, shear layers, and boundary layers, must be accurately modeled and solution domains must be easily generated for complex shapes, allowing rapid re-calculation with respect to geometry changes.

The emphasis in aerothermodynamics on heat transfer and high-speed flows provides additional challenges with regard to the accurate resolution of boundary layers, loss of accuracy with highly-stretched meshes, and convergence slowdowns associated with a disparity in information speeds between nearly stagnated boundary layer flow and hypersonic shock layers.

To make the next generation of computational aerothermodynamic predictive tools responsive to the design phase a truly multi-dimensional, robust Navier-Stokes solver based on general unstructured domains is required. Advanced, aggressive convergence acceleration methods and the exploitation of distributed or massively parallel architectures are mandatory.

1.3 Background

1.3.1 Fluid Dynamics Algorithms

The twin goals of improving both accuracy and efficiency have long been objectives of the computational fluid dynamics developers. Accuracy is typically addressed through the spatial discretization, with the Van Leer 'Ultimate' series of papers[4, 5] being a

driving force throughout the 1970's. In the domain of hypersonic applications, efficiency has been improved through the temporal discretization, with MacCormack's work standing out on explicit predictor/corrector[6], implicit line Gauss-Seidel[7, 8], and approximate factorization schemes[9], along with a notable extension by Candler to the LU-SGS scheme[10]. Several industrial solvers, such as GASP[11], have emerged from this finite volume legacy, such as the successful CFL3D code[12].

In his landmark paper, Roe[13], building on the work of Godunov[14], introduced an upwind, approximate-Riemann-problem solution technique for the one-dimensional Euler equations. The Roe scheme has been extended on structured meshes to multiple dimensions by decomposing the domain into locally one-dimensional problems aligned with the computational coordinates. Viscous terms have been incorporated and in a decade's time the structured Roe scheme[1] for Navier-Stokes equations set the standard for continuum aerothermodynamics[15, 16, 17, 18].

Unstructured schemes have since been developed, seeking both much greater flexibility for complex geometries and a reduction in preferential solution directions aligned with computational coordinates. Notable contributions for the unstructured Roe schemes have been made by Barth[19, 20, 21, 22] and Whitaker[23, 24]. These approaches, however, still rely upon a locally one-dimensional, dimensionally-split algorithm at cell faces.

A different tack has been taken by Fey[25, 26], who has advocated the method of transport for a true multi-dimensional treatment of the Euler equations. This method is more an extension of flux vector splitting concepts, an approach known to be more dissipative than flux difference splitting[17].

The advent of multi-dimensional linear advection schemes, termed fluctuation splitting[27, 28, 29, 30, 31, 32], induced Roe to revisit his one-dimensional scheme. The fluctuation splitting formulation is based on an unstructured triangulation and is local to each cell, suitable for application on massively parallel computers. A two-dimensional analog to the Roe scheme for the Euler equations has been developed based on characteristic wave decompositions[33, 34, 35, 36, 37, 38, 39, 40, 41]. Unfortunately, these wave decomposition schemes have been of limited robustness

[1] Includes extensions of the base Roe scheme to higher-order accurate discretizations.

and have not reached their full potential. Applications have also been made to the shallow-water equations[42].

Taking the wave decomposition idea to the extreme, some authors have applied fluctuation splitting to Boltzmann schemes for the Euler equations[43, 44, 45, 46]. These gas kinetic schemes are very expensive to run and show very modest improvements over dimensionally-split finite volume[47].

Diffusion terms have been incorporated into scalar fluctuation splitting schemes[21, 48] and viscous terms have been added to the wave-decomposition models to solve Navier-Stokes problems[49, 50, 51, 52]. While the Galerkin approach to viscous terms appears compatible with fluctuation splitting for convective terms, these schemes suffer from a lack of robustness. Carette[51] states, "However, this scheme is not satisfactory at present in terms of robustness and convergence, and improvements in this respect is still subject [*sic*] of current research." Even for the classical flat plate boundary layer problem Tomaich[50] reports, "...the agreement with the Blasius solution is rather poor."

Sidilkover, who along with Roe established a strong link between fluctuation splitting and upwind flux difference splitting finite volume[3], has proposed an alternative multi-dimensional treatment of the Euler system[1, 2]. Rather than performing a wave decomposition to decouple the Euler equations, Sidilkover employs fluctuation splitting to treat the system of equations as a whole unit. In doing so he has been able to apply efficient multigrid[53, 54, 55] solution strategies to shock reflection and channel flow problems. Advantages of Sidilkover's method for the Euler equations include: arbitrary triangulations, stability of Gauss-Seidel relaxation on high resolution discretization, compact stencil, second-order accuracy, and rotationally invariant artificial dissipation. Among these, Sidilkover[55] claims, "The fundamental advantage of this approach is that it leads to a scheme that combines high-resolution and good stability properties."

Sidilkover's fluctuation splitting scheme for the Euler equations has the promise to satisfy many of the cutting-edge aerodynamic requirements. It provides a truly multi-dimensional treatment of the governing system of equations in an unstructured,

compact, high-order algorithm suitable to distributed and massively parallel computation. Being stable with respect to Gauss-Seidel relaxation, as opposed to Runge-Kutta[56] marching, opens the possibility of greatly improved multigrid convergence rates. Robustness appears to be improved relative to the wave decomposition models.

What is lacking from the scheme to be an aerothermodynamic tool are the extension to the Navier-Stokes equations, rigorous evaluation of robustness and accuracy relative to finite volume schemes on complex geometries, analysis of the effect of grid stretching, and critical determination of the accuracy and efficiency for heat transfer and skin friction calculations.

1.3.2 Mesh Adaption Strategies

Solution adaptive remeshing techniques have been utilized with some success for hypersonic flows on structured domains. A simple, effective approach developed by Gnoffo[57] utilizes a spring analogy energy minimization to align the bow shock and cluster to the boundary layer. This approach works very well for entry forebodies, for which it was developed, but is more difficult to apply to complex vehicle shapes. The method also is unresponsive to embedded shocks or other shock-layer flow features.

Harvey[58, 59, 60] has developed a mesh adaption technique that is sensitive to shock-layer features to obtain parabolized Navier-Stokes solutions over simple configurations, *e.g.* cones, using a spring analogy based on Gnoffo's work. Unfortunately, defining relative clustering strengths for various flow features proved difficult, and a damaging lack of robustness is shown for three-dimensional leeside flowfields[59].

Mesh adaption on unstructured domains offers a significant benefit over structured mesh adaption—the ability to insert and delete nodes. Much of the research in this area has gone into global remeshing using isotropic cells[61, 62, 63, 64, 65, 66, 67]. Grids composed of (nearly) isotropic cells quickly become prohibitively large for hypersonic applications, where the capture of essentially one-dimensional flow features, such as shocks, results in refinement in all three dimensions. These methods typically employ gradient clustering, using a second derivative check on some or all of the dependent variables to define clustering strengths. This sort of clustering is intended

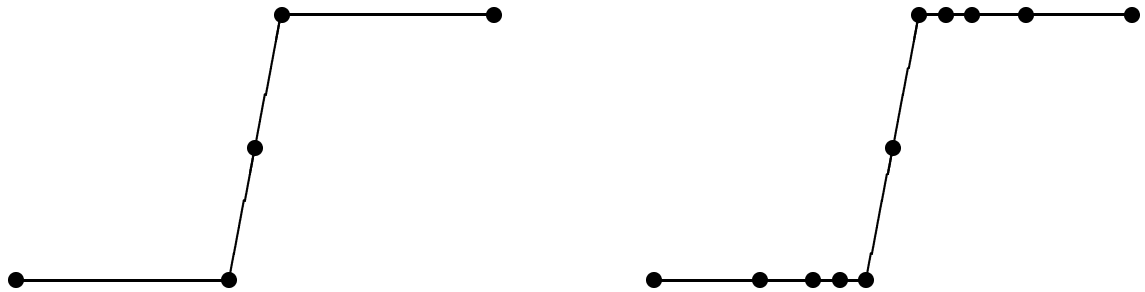


Figure 1.1: Pictorial of "ideal" mesh for shock discontinuity (left) and mesh resulting from gradient based clustering (right).

to reduce the interpolation error in a piecewise-linear data representation[68, 69], but is not necessarily driven by the flow physics, and can lead to excessive clustering or conflicting requirements in certain regions, such as a bow shock or stagnation point. Figure 1.1 presents an illustrative pictorial based on the results of Ait-Ali-Yahia[70] *et al.*, where 18 cells were driven into the bow shock by gradient based clustering. A shock is pictured on the left side of the figure with an ideal mesh for a three-point stencil. On the right side is a mesh typically produced by gradient-based clustering, which gathers more points in the vicinity of the discontinuity without providing sharper resolution.

More recently, impressive results using anisotropic elements have been reported by Habashi[71, 72, 73, 74, 75, 76] *et al.* With that approach, all grid adaptions are local operations, as opposed to global remeshings. Highly-stretched elements are obtained, achieved by equating the interpolation error along each edge, again using a spring analogy minimization. The clustering is still driven by second derivatives in the solution.

Roe[32, 77] has applied the concept of performing global mesh adaption *via* local node movements to a scalar advection problem using fluctuation splitting. His analysis reveals that a characteristic mesh results, with far fewer points required than gradient based clustering would imply. His method is based on the minimization of an objective function formed by taking derivatives of the fluctuation splitting scheme. A differentiable high-resolution linear scheme, which cannot be monotonic[56], is chosen.

For a complex flowfield or for systems of equations it may not be possible to achieve a perfectly-aligned characteristic mesh, in which case the non-monotonic property would most likely be detrimental or even fatal to the solution. It is not clear how to extend the differentiability requirement to a high-resolution, non-linear monotonic scheme.

Another attempt at mesh alignment has been performed by Trépanier[78] *et al.* While their scheme is unresponsive to characteristic lines, a wave decomposition model is used with an inviscid cell-centered finite volume solver to produce shock alignment of unstructured isotropic cells. Less success was demonstrated for shear alignment. One drawback of the method is the need to explicitly detect and classify flow features, which could hinder the extension to three dimensions. Similarly, Parikh[79] *et al.* use a wave decomposition in conjunction with a cell-centered finite volume solver to drive edge alignment, but with automated general feature detection. Unfortunately, their approach was able to provide only extremely modest benefits, in part because there was no mechanism for providing cell stretching.

1.4 Course of Action

A systematic approach to developing new methods is pursued. Current leading-edge technology is embraced and developed into a complete gas dynamics solver, applicable to reentry vehicles across their speed range. Throughout, the scientific method is followed, where new schemes are critically evaluated against the current state of the art.

One-dimensional scalar advection is considered first, where the common ancestry of the upwind finite volume and fluctuation splitting paradigms is presented. Extension to the incorporation of diffusive terms is presented, followed by the treatment of systems, for both the Euler and Navier-Stokes equations. The chapter culminates with the current state of the art in dimensionally-split schemes. Fluctuation splitting provides an identical discretization in one dimension.

Two-dimensional scalar advection on unstructured domains is addressed next. The methodology for applying the dimensionally-split concepts in multiple dimensions is

shown. The different approach of fluctuation splitting is highlighted, providing a truly multi-dimensional treatment. A critical comparison between the upwind finite volume and fluctuation splitting schemes for scalar equations is performed. The extension to diffusive problems is developed, with emphasis placed on consistent incorporation into the fluctuation splitting framework while still maintaining formal second-order accuracy.

Novel concepts for mesh adaption are developed analytically for two-dimensional scalar problems, based on the physics of the solution rather than gradient-based clustering. Emphasis is placed on concepts that can be extended to systems. Demonstrations for simple basic problems and more challenging advection-diffusion problems are performed.

Systems in two dimensions are formulated for the Navier-Stokes equations, representing the crux of the development of the fluctuation splitting concepts. The treatment follows the lead of Sidilkover in treating the system as a cohesive unit, in contrast to the wave-model simplifications of Roe and Deconinck. Specialization to axisymmetric equations of fluid motion provide useful applications and consistent treatment of source terms.

The mesh adaption strategies developed for scalar equations are reformulated for the fluid dynamics systems. The resolution of flowfield features is addressed, along with robustness and convergence of the adaption process. An automatic refinement procedure is sought whereby the solution is converged uniformly with mesh density.

A critical aerothermodynamic evaluation of the fluctuation splitting algorithm and mesh adaption strategies are performed using the primary case of an entry capsule at Mach-10 wind tunnel conditions, for which prior experimental and computational data exists.

Chapter 2

One-Dimensional Analysis

The current state of the art for solving the compressible Navier-Stokes equations, namely upwind flux-difference-split finite volume schemes, is developed on non-uniform meshes in one spatial dimension. Upwind flux difference splitting, in particular the Roe scheme[13, 80], is considered the most accurate scheme for compressible Navier-Stokes applications[17, 18], primarily because of the low levels of artificial dissipation introduced through the matrix dissipation model. The particular finite volume scheme considered in this dissertation can be described as node-based, median-dual mesh upwind Roe flux difference splitting finite volume with limited multi-dimensional reconstruction, and will be abbreviated as DMFDSFV, for dual mesh flux difference splitting finite volume, throughout.

The sections in this chapter progress from scalar advection to diffusion and then to combined advection-diffusion. Treatment of scalar equations is followed by the Euler and Navier-Stokes systems of equations. The fluctuation splitting approach is developed in parallel with the DMFDSFV analysis, and is shown to result in identical discretizations in one dimension.

This chapter has a threefold purpose: to introduce DMFDSFV and fluctuation splitting basics, to develop the state of the art for the locally one-dimensional approximate Riemann solver used in the finite volume algorithms, and to serve as a prelude for the multi-dimensional analyses, where fluctuation splitting offers new capability over the DMFDSFV extensions.

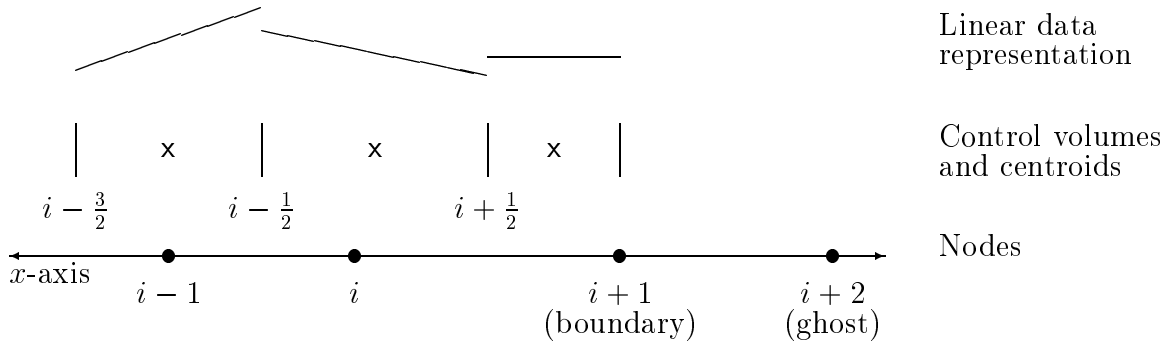


Figure 2.1: One-dimensional finite volume domain.

2.1 Domain

In one dimension the domain considered is a discretization of the x-axis, either with uniform or non-uniform spacing between grid nodes, which are indexed by i. Dependent variables are stored at the nodes.

In the node-based finite volume context a median-dual control volume is constructed about each node by defining a cell face halfway between adjacent nodes. This convention is depicted in Figure 2.1, with the cell faces referred to as the $\pm\frac{1}{2}$ points. In the illustrative case of Figure 2.1, the $i+1$ node is a boundary point, and the corresponding cell extends only from $i+\frac{1}{2}$ to $i+1$. The generalized volume of the median-dual cell is formed as $x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}}$ for interior nodes or $x_{i+1} - x_{i+\frac{1}{2}}$ for the boundary point shown in Figure 2.1. If the solution at the boundary nodes is specified, then there are two fewer interior finite volumes to solve than the number of nodes. If the boundary nodes are updated by the interior scheme, then there is a one-to-one correspondence between the nodes and control volumes, plus two additional numerical boundary conditions, which may be implemented with ghost nodes ($i+2$ in Figure 2.1) or specified boundary fluxes. If the ghost node is used, it can be located co-incident with the physical boundary node, and does not need to have an associated control volume. A ghost node co-located with the physical boundary node,

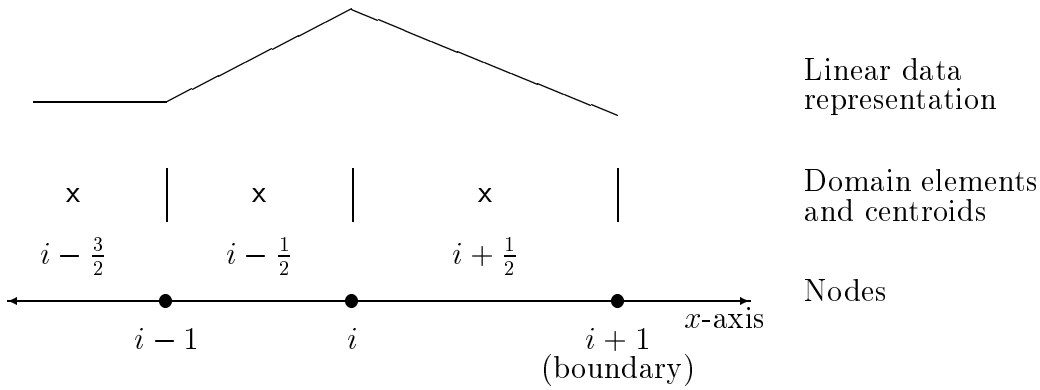


Figure 2.2: One-dimensional fluctuation splitting domain.

without an associated control volume extends more easily to multiple dimensions with unstructured grids than would a formulation based upon physically locating the ghost node.

Notice that for the nodal distribution depicted, which has non-uniform spacing, the cell centroids, denoted by x in Figure 2.1, do not coincide with the nodes. Barth[22] states that nodal storage in this case, referred to as mass lumping in a finite element context, only alters the time accuracy of finite volume schemes, and not the steady state solutions. In the present dissertation unsteady problems will employ uniform grids, though stationary problems are free to use non-uniform node distributions.[1]

Notice also the piecewise-linear representation of data in the finite volume context. Discontinuous jumps in the dependent data are allowed at cell faces.

Figure 2.2 depicts the discretization of the domain for the fluctuation splitting approach. The data is now continuous and piecewise linear over elements defined by the nodes. The centroids of the fluctuation splitting elements are at the same locations as the faces of the finite volume cells. No special definition of a boundary

[1]The use of uniform meshes to avoid degradation of time accuracy due to mass lumping is done for convenience. It is possible to compute cell centroids and use that location in calculations involving cell distances. Conversely, a cell-centered, rather than node based, data storage structure could be adopted for the finite volume discretization.

cell is required. The length of an element is,

$$\ell_{i,i+1} = x_{i+1} - x_i = \Delta_i x \tag{2.1}$$

There are one fewer elements than nodes, and each element is associated with two nodes. Correspondingly, each interior node is associated with two elements, while each boundary node is associated with only one element.

The data structure for fluctuation splitting shown in Figure 2.2 resembles a cell-centered finite volume layout, with a key difference that cell-centered finite volume stores the solution at cell centers, as opposed to at the nodes for fluctuation splitting. Additionally, it is emphasized that the fluctuation splitting data is C-0 continuous whereas cell-centered finite volume would have discontinuous jumps in the data at the cell interfaces.

2.2 Scalar Advection

A hyperbolic conservation law takes the form,

$$\mathbf{U}_t + \vec{\nabla} \cdot \vec{\mathbf{F}} = 0 \tag{2.2}$$

where $\mathbf{U}$ is the vector of conserved variables and $\vec{\mathbf{F}}$ is the flux[2] of these variables. Following Godunov[14], Eqn. 2.2 can be evaluated in an integral sense,

$$\int_\Omega \mathbf{U}_t \, d\Omega = -\int_\Omega \vec{\nabla} \cdot \vec{\mathbf{F}} \, d\Omega \tag{2.3}$$

If the control volume, Ω, is fixed in time, then,

$$\int_\Omega \mathbf{U}_t \, d\Omega = S_\Omega \bar{\mathbf{U}}_t \tag{2.4}$$

where the over-bar indicates a cell-average value. Using the divergence theorem the flux term can be evaluated as,

$$\int_\Omega \vec{\nabla} \cdot \vec{\mathbf{F}} \, d\Omega = \oint_\Gamma \vec{\mathbf{F}} \cdot \hat{n} \, d\Gamma \tag{2.5}$$

[2]The flux function $\vec{\mathbf{F}}$ is a function of the dependent variables $\mathbf{U}$, $\vec{\mathbf{F}} = \vec{\mathbf{F}}(\mathbf{U})$. For some of the scalar cases $\vec{F}$ will also be a function of the independent variables, $\vec{F} = \vec{F}(u, x, y)$. In such cases the flux will be in a variables-separable form, $\vec{F} = \vec{\lambda}(x, y)u$.

where $\hat{n}$ is the outward unit normal to the control volume boundary, Γ.

The one-dimensional scalar advection problem is obtained from Eqn. 2.2 as,

$$u_t + F_x = 0 \tag{2.6}$$

which can be written for a control cell as,

$$S_\Omega \bar{u}_t = -\int_\Omega F_x \, d\Omega = F_{left\ face} - F_{right\ face} \tag{2.7}$$

2.2.1 Linear Advection

Linear advection is obtained from Eqn. 2.6 by choosing $F = \lambda u$. The advection speed, λ, is taken to be constant.

DMFDSFV

Equation 2.7 is expressed for the finite volume about node i with mass lumping to the node as,

$$S_i u_{i_t} = F_{i-\frac{1}{2}} - F_{i+\frac{1}{2}} \cong f_{i-\frac{1}{2}} - f_{i+\frac{1}{2}} = R_i \tag{2.8}$$

where the numerical flux, f, is a difference expression approximating the exact flux function, F. Choosing,

$$f_{i+\frac{1}{2}} = \frac{F_i + F_{i+1}}{2} \tag{2.9}$$

results in a second-order central difference scheme,

$$R_i = -\delta_i F \tag{2.10}$$

The first-order upwind CIR[81] scheme is obtained by the choice,

$$f_{i+\frac{1}{2}} = \frac{\lambda+|\lambda|}{2} u_i + \frac{\lambda-|\lambda|}{2} u_{i+1} \tag{2.11}$$

$$= \frac{F_i + F_{i+1}}{2} - \frac{|\lambda|}{2} \Delta_i u \tag{2.12}$$

giving,

$$R_i = -\delta_i F + \frac{|\lambda|}{2} \delta_i^2 u \tag{2.13}$$

The first-order upwind is then seen to be the same as a central distribution plus an artificial dissipation term,

$$\Phi_{\mathsf{U}} = \frac{|\lambda|}{2}\delta_i^2 u \tag{2.14}$$

Second-order upwind is constructed following the MUSCL concept of Van Leer[5], where a linear reconstruction is performed on each finite volume. The numerical flux of Eqn. 2.11 is modified to be,

$$f_{i+\frac{1}{2}} = \frac{\lambda+|\lambda|}{2}u_{\mathsf{L}} + \frac{\lambda-|\lambda|}{2}u_{\mathsf{R}} \tag{2.15}$$

where u_{L} is the reconstructed conserved variable on the left side of the cell face and u_{R} is the reconstructed variable on the right side of the cell face. Following Barth[22], a limited reconstruction is performed on each cell as,

$$u_{face} = u_i + \psi_i(\vec{\nabla}u)_i \cdot \vec{r}_{face} \tag{2.16}$$

The gradient is evaluated as a central difference,

$$(\vec{\nabla}u)_i = \frac{\delta_i u}{S_i} \tag{2.17}$$

The limiter function, ψ, is employed to provide monotonicity of the solution, based upon positivity arguments. The limiter takes the form,

$$\psi\left(\frac{p}{q}\right)$$

where,

$$p = \frac{u_{i\pm1} - u_i}{2}, \quad q = (\vec{\nabla}u)_i \cdot \vec{r}_{i\pm\frac{1}{2}}$$

The more restrictive of the $\pm$ choices is used for ψ. Some popular limiters are presented in appendix A.

The discrete numerical flux (Eqn. 2.15) expands to,

$$\begin{aligned} f_{i+\frac{1}{2}} &= \frac{\lambda+|\lambda|}{2}\left(u_i + \psi_i\frac{\ell_{i,i+1}}{2S_i}\delta_i u\right) + \frac{\lambda-|\lambda|}{2}\left(u_{i+1} - \psi_{i+1}\frac{\ell_{i,i+1}}{2S_{i+1}}\delta_{i+1}u\right) \\ &= \frac{F_i + F_{i+1}}{2} - \frac{|\lambda|}{2}\Delta_i u + \frac{\ell_{i,i+1}}{4}\left(\frac{\psi_i}{S_i}\delta_i(F+|\lambda|u) - \frac{\psi_{i+1}}{S_{i+1}}\delta_{i+1}(F-|\lambda|u)\right) \end{aligned} \tag{2.18}$$

leading to,

$$R_i = -\delta_i F + \Phi_{\mathsf{U}} + R_{2\mathsf{U}} \tag{2.19}$$

where the second-order correction is,

$$\begin{aligned} R_{2\mathsf{U}} &= \frac{\ell_{i-1,i}}{4}\left(\frac{\psi_{i-1}}{S_{i-1}}\delta_{i-1}(F{+}|\lambda|u) - \frac{\psi_i}{S_i}\delta_i(F{-}|\lambda|u)\right) \\ &- \frac{\ell_{i,i+1}}{4}\left(\frac{\psi_i}{S_i}\delta_i(F{+}|\lambda|u) - \frac{\psi_{i+1}}{S_{i+1}}\delta_{i+1}(F{-}|\lambda|u)\right) \end{aligned} \tag{2.20}$$

The residual (Eqn. 2.19) can be rearranged as,

$$\begin{aligned} R_i &= -\delta_i F - \frac{1}{8}\left[\ell_{i-1,i}\frac{\psi_{i-1}}{S_{i-1}}F_{i-2} - 2\psi_i F_{i-1} + \left(\ell_{i,i+1}\frac{\psi_{i+1}}{S_{i+1}} - \ell_{i-1,i}\frac{\psi_{i-1}}{S_{i-1}}\right)F_i\right. \\ &\left. + 2\psi_i F_{i+1} - \ell_{i,i+1}\frac{\psi_{i+1}}{S_{i+1}}F_{i+2}\right] + \Phi_{2\mathsf{U}} \end{aligned} \tag{2.21}$$

where the artificial dissipation is now,

$$\begin{aligned} \Phi_{2\mathsf{U}} &= \Phi_{\mathsf{U}} + \frac{|\lambda|}{8}\left[-\ell_{i-1,i}\frac{\psi_{i-1}}{S_{i-1}}u_{i-2} + \frac{\psi_i}{S_i}\left(\ell_{i,i+1} - \ell_{i-1,i}\right)u_{i-1}\right. \\ &+ \left(\ell_{i-1,i}\frac{\psi_{i-1}}{S_{i-1}} + \ell_{i,i+1}\frac{\psi_{i+1}}{S_{i+1}}\right)u_i - \frac{\psi_i}{S_i}\left(\ell_{i,i+1} - \ell_{i-1,i}\right)u_{i+1} \\ &- \left.\ell_{i,i+1}\frac{\psi_{i+1}}{S_{i+1}}u_{i+2}\right] \end{aligned} \tag{2.22}$$

On a uniform grid and without limiting, the second-order residual (Eqn. 2.21) reduces to a low-truncation-error central difference minus fourth-order dissipation,

$$R_i = \frac{1}{8}\left(-F_{i-2} + 6F_{i-1} - 6F_{i+1} + F_{i+2}\right) + \frac{|\lambda|}{8}\left(-u_{i-2} + 4u_{i-1} - 6u_i + 4u_{i+1} - u_{i+2}\right) \tag{2.23}$$

Fluctuation splitting

In the fluctuation splitting framework Eqn. 2.7 is evaluated over each domain element, without recourse to the divergence theorem. The element fluctuation is defined as,

$$S_\Omega \bar{u}_t = \phi_{\mathsf{E}} = -\int_\Omega F_x \, d\Omega \tag{2.24}$$

Assuming piecewise linear data, the fluctuation for the cell bounded by x_i and x_{i+1} is evaluated as,

$$\phi_{i,i+1} = -\lambda \int_{x_i}^{x_{i+1}} u_x \, d\ell = -\ell_{i,i+1} \, \lambda \frac{\Delta_i u}{\Delta_i x} = -\Delta_i F \tag{2.25}$$

The elemental update, the LHS of Eqn. 2.24, is formed as,

$$S_\Omega \bar{u}_t = \ell_{i,i+1} \left(\frac{u_i + u_{i+1}}{2} \right)_t = \frac{\ell_{i,i+1}}{2} (u_{i_t} + u_{i+1_t}) = \phi_{i,i+1} \tag{2.26}$$

Partitioning the fluctuation into halves and distributing equally to the nodes yields the basic non-upwind elemental update formula,

$$\frac{\ell_{i,i+1}}{2} u_{i_t} = \frac{\phi_{i,i+1}}{2}, \quad \frac{\ell_{i,i+1}}{2} u_{i+1_t} = \frac{\phi_{i,i+1}}{2} \tag{2.27}$$

Assembling all the elemental contributions to the nodal updates, it is clear each interior node will receive fluctuation signals from the elements adjacent to the left and right. The nodal update is formed as the sum of these fluctuation contributions,

$$\frac{\ell_{i-1,i}}{2} u_{i_t} + \frac{\ell_{i,i+1}}{2} u_{i_t} = \frac{\ell_{i-1,i} + \ell_{i,i+1}}{2} u_{i_t} = S_i u_{i_t} = \frac{\phi_{i-1,i}}{2} + \frac{\phi_{i,i+1}}{2} \tag{2.28}$$

or,

$$S_i u_{i_t} = \frac{\phi_{i-1,i} + \phi_{i,i+1}}{2} \tag{2.29}$$

A popular nomenclature convention for Eqns. 2.27 and 2.29 is to describe the elemental distribution formula as,

$$S_i u_{i_t} \leftarrow \frac{\phi_{i,i+1}}{2} + \text{COE}, \quad S_{i+1} u_{i+1_t} \leftarrow \frac{\phi_{i,i+1}}{2} + \text{COE} \tag{2.30}$$

where COE indicates a sum of similar contributions from other elements joining at that node.

Expanding the nodal update formula (Eqn. 2.29),

$$S_i u_{i_t} = \frac{-\nabla_i F - \Delta_i F}{2} = -\delta_i F \tag{2.31}$$

which is the identical central discretization as for DMFDSFV (Eqn. 2.10).

An upwind scheme can be constructed by introducing artificial dissipation in order to redistribute the fluctuation,

$$\phi'_{\mathsf{E}} = \text{sign}(\lambda)\phi_{\mathsf{E}} \tag{2.32}$$

The upwind distribution formula becomes,

$$\begin{aligned} S_i u_{i_t} &\leftarrow \frac{\phi_{\mathsf{E}} - \phi'_{\mathsf{E}}}{2} + \text{COE} = \frac{\phi_{i,i+1}(1 - \text{sign}(\lambda))}{2} + \text{COE} \\ S_{i+1} u_{i+1_t} &\leftarrow \frac{\phi_{\mathsf{E}} + \phi'_{\mathsf{E}}}{2} + \text{COE} = \frac{\phi_{i,i+1}(1 + \text{sign}(\lambda))}{2} + \text{COE} \end{aligned} \tag{2.33}$$

Using the fluctuation definition (Eqn. 2.25) the nodal update is obtained as,

$$S_i u_{i_t} = -\frac{(\lambda + |\lambda|)\nabla_i u}{2} - \frac{(\lambda - |\lambda|)\Delta_i u}{2} = -\delta_i F + \frac{|\lambda|}{2}\delta_i^2 u \tag{2.34}$$

which is identical to the first-order upwind discretization for DMFDSFV (Eqn. 2.13).

A second-order scheme is easily obtained by adding the exact same DMFDSFV correction, $R_{2\mathsf{U}}$ (Eqn. 2.20), to the nodal update formula (Eqn. 2.34).

2.2.2 Non-linear Advection

Non-linear advection is obtained from Eqn. 2.6 by choosing the flux to be,

$$F = \frac{u^2}{2} \tag{2.35}$$

Define the Jacobian of the flux,

$$A = F_u = u \tag{2.36}$$

so that,

$$F_x = \frac{\partial F}{\partial x} = \frac{\partial F}{\partial u}\frac{\partial u}{\partial x} = F_u u_x = A u_x$$

Equation 2.6 may be rearranged in non-conservation form,

$$u_t + F_x = u_t + A u_x = 0 \tag{2.37}$$

DMFDSFV

Following Roe[13], the analog to the numerical flux of Eqn. 2.15 becomes,

$$\begin{aligned} f_{i+\frac{1}{2}} &= \frac{A_{\mathsf{L}}+|\tilde{A}|_{i+\frac{1}{2}}}{2}u_{\mathsf{L}} + \frac{A_{\mathsf{R}}-|\tilde{A}|_{i+\frac{1}{2}}}{2}u_{\mathsf{R}} \\ &= \frac{F_{\mathsf{L}}+F_{\mathsf{R}}}{2} - \frac{|\tilde{A}|_{i+\frac{1}{2}}}{2}(u_{\mathsf{R}}-u_{\mathsf{L}}) \end{aligned} \tag{2.38}$$

where $\tilde{A}$ is the conservative linearization of the flux Jacobian, which in this case is,

$$\tilde{A}_{i+\frac{1}{2}} = \frac{u_{\mathsf{L}}+u_{\mathsf{R}}}{2} \tag{2.39}$$

A first-order upwind scheme is obtained using piecewise-constant data, $u_{\mathsf{L}} = u_i$, and $u_{\mathsf{R}} = u_{i+1}$. A second-order upwind scheme is constructed using the linear reconstruction of Eqn. 2.16. The first-order residual may be written explicitly as,

$$R_i = -\delta_i F + \frac{|\tilde{A}|_{i+\frac{1}{2}}}{2}\Delta_i u - \frac{|\tilde{A}|_{i-\frac{1}{2}}}{2}\nabla_i u \tag{2.40}$$

Fluctuation splitting

The elemental fluctuation is

$$\phi_{\mathsf{E}} = -\int_{\Omega} F_x \, d\Omega = -\int_{\Omega} A u_x \, d\Omega \tag{2.41}$$

Assuming piecewise-linear data Eqn. 2.41 becomes,

$$\phi_{\mathsf{E}} = -\tilde{A}_{i+\frac{1}{2}}\Delta_i u = -\Delta_i F \tag{2.42}$$

An upwind scheme is created by introducing the artificial dissipation,

$$\phi'_{\mathsf{E}} = \mathrm{sign}(\tilde{A}_{i+\frac{1}{2}})\phi_{\mathsf{E}} = -|\tilde{A}|_{i+\frac{1}{2}}\Delta_i u \tag{2.43}$$

The distribution formula remains,

$$\begin{aligned} S_i u_{i_t} &\leftarrow \frac{\phi_{\mathsf{E}}-\phi'_{\mathsf{E}}}{2} + \mathrm{COE} \\ S_{i+1} u_{i+1_t} &\leftarrow \frac{\phi_{\mathsf{E}}+\phi'_{\mathsf{E}}}{2} + \mathrm{COE} \end{aligned} \tag{2.44}$$

The nodal update is,

$$\begin{aligned} S_i u_{i_t} &= \frac{\phi_{i-1,i} + \phi'_{i-1,i}}{2} + \frac{\phi_{i,i+1} - \phi'_{i,i+1}}{2} \\ &= -\frac{\nabla_i F}{2} - \frac{|\tilde{A}|_{i-\frac{1}{2}}}{2}\nabla_i u - \frac{\Delta_i F}{2} + \frac{|\tilde{A}|_{i+\frac{1}{2}}}{2}\Delta_i u \\ &= -\delta_i F - \frac{|\tilde{A}|_{i-\frac{1}{2}}}{2}\nabla_i u + \frac{|\tilde{A}|_{i+\frac{1}{2}}}{2}\Delta_i u \end{aligned} \tag{2.45}$$

This is the identical update formula as for DMFDSFV (Eqn. 2.40).

Expansion shocks

The discretization of Roe's scheme allows for non-physical expansion shocks that violate the entropy condition. Harten and Hyman[82] proposed a commonly used method for perturbing the wavespeeds such that entropy is satisfied and expansion shocks are prevented. The correction is applied to any wavespeed that can go to zero at a sonic point and takes the form,

$$|\tilde{\lambda}|_{i+\frac{1}{2}} \leftarrow \begin{cases} |\tilde{\lambda}|_{i+\frac{1}{2}} & \text{if } |\tilde{\lambda}|_{i+\frac{1}{2}} \geq \epsilon \\ \frac{1}{2}\left(\frac{\tilde{\lambda}^2_{i+\frac{1}{2}}}{\epsilon} + \epsilon\right) & \text{if } |\tilde{\lambda}|_{i+\frac{1}{2}} < \epsilon \end{cases} \tag{2.46}$$

where the perturbation scale is,

$$\epsilon = \max\left[0, \quad (\tilde{\lambda}_{i+\frac{1}{2}} - \lambda_i), \quad (\lambda_{i+1} - \tilde{\lambda}_{i+\frac{1}{2}})\right] \tag{2.47}$$

2.3 Scalar Advection-Diffusion

The governing equation for scalar advection-diffusion problems in one-dimension is,

$$u_t + F_x = (\mu u_x)_x \tag{2.48}$$

2.3.1 Heat Equation

Modeling of the viscous RHS in Eqn. 2.48 begins with a consideration of the heat equation,

$$u_t = (\mu u_x)_x \tag{2.49}$$

In the finite volume framework one approach to discretizing the viscous term is to construct a viscous flux, so that the nodal update becomes,

$$S_i u_{i_t} = (\bar{\mu} u_x)_{i+\frac{1}{2}} - (\bar{\mu} u_x)_{i-\frac{1}{2}} \tag{2.50}$$

where,

$$(\bar{\mu} u_x)_{i+\frac{1}{2}} = \bar{\mu}_{i+\frac{1}{2}} \left[\frac{(\vec{\nabla} u)_i + (\vec{\nabla} u)_{i+1}}{2} \right] \tag{2.51}$$

with the gradients $\vec{\nabla} u$ defined by Eqn. 2.17. This approach leads to a five-point stencil.

An alternative is to use a finite element discretization, which results in a three-point stencil. This approach is adopted both by Barth[22] and Anderson and Bonhaus[83] in a finite volume context and by Tomaich[50] in a fluctuation splitting context.

A Galerkin finite element discretization, using mass lumping to the nodes, is constructed on the fluctuation splitting domain by integrating with the aid of the finite element linear shape function υ (see Bickford §4.2.2[84] or Bathe §7.2[85]),

$$S_i u_{i_t} = \int_\Omega \upsilon_i (\mu u_x)_x \, d\Omega \tag{2.52}$$

Integrating by parts,

$$S_i u_{i_t} = \upsilon_i (\mu u_x)|_{i-1}^{i+1} - \int_\Omega (\upsilon_i)_x (\mu u_x) \, d\Omega \tag{2.53}$$

The shape function is the linear tent function, and is equal to zero at $x_{i\pm1}$, eliminating the first RHS term of Eqn. 2.53. The remaining term is integrated over each element connecting at node i,

$$S_i u_{i_t} = -\sum_E \int_E \upsilon_x u_x \mu \, d\Omega \tag{2.54}$$

The dependent variable and shape function gradients are constant over the element, and taking the element-average viscosity coefficient the elemental contributions are,

$$\begin{aligned} S_i u_{i_t} &\leftarrow \frac{\Delta_i u}{\ell_{i,i+1}} \bar{\mu}_{i+\frac{1}{2}} + \text{COE} \\ S_{i+1} u_{i+1_t} &\leftarrow -\frac{\Delta_i u}{\ell_{i,i+1}} \bar{\mu}_{i+\frac{1}{2}} + \text{COE} \end{aligned} \tag{2.55}$$

The nodal update is written,

$$S_i u_{i_t} = \frac{\Delta_i u}{\ell_{i,i+1}} \bar{\mu}_{i+\frac{1}{2}} - \frac{\nabla_i u}{\ell_{i-1,i}} \bar{\mu}_{i-\frac{1}{2}} \tag{2.56}$$

2.3.2 Combined Advection and Diffusion

The combined effects of advection and diffusion in the governing equation (Eqn. 2.48) are treated by discretizing the advection terms as discussed in section 2.2 and adding the discretization of the diffusion terms of section 2.3.1. Recall, however, that the upwind advection discretization includes artificial dissipation, which can mask the physical dissipation.

Perhaps the best approach for solving discretized advection-diffusion problems, as suggested by Barth[21], is to include the maximum of either the physical diffusion term, as defined by Eqn. 2.51 or Eqn. 2.55, or the artificial dissipation, the second term of Eqn. 2.38 for DMFDSFV or $\frac{\phi'_{\mathsf{E}}}{2}$ in Eqn. 2.43 for fluctuation splitting.

2.4 Systems

A hyperbolic conservation law for systems (Eqn. 2.2) is written in one dimension as,

$$\mathbf{U}_t + \mathbf{F}_x = 0 \tag{2.57}$$

A decomposition of the flux function is sought such that the system can be expressed as a decoupled set of advection-diffusion equations.

2.4.1 Euler Equations

The one-dimensional Euler equations[86] for perfect gases, suitable for simulating non-reacting, low-Knudson-number shock-tube flows, are written as a conservation law (Eqn. 2.57) with,

$$\mathbf{U} = \left\{ \begin{array}{c} \rho \\ \rho u \\ \rho E \end{array} \right\} \tag{2.58}$$

$$\mathbf{F} = \left\{ \begin{array}{c} \rho u \\ \rho u^2 + P \\ \rho u H \end{array} \right\} \tag{2.59}$$

The Euler equations have a form similar to the non-linear advection problem.

The total energy and enthalpy are obtained from the internal energy and enthalpy,

$$E = e + \frac{u^2}{2} \qquad H = h + \frac{u^2}{2}$$

The energy and enthalpy are related as,

$$h = e + \frac{p}{\rho}$$

The perfect gas equation of state is,

$$P = \rho e(\gamma - 1) \tag{2.60}$$

DMFDSFV

The numerical flux remains as in Eqn. 2.38,

$$\mathbf{f}_{i+\frac{1}{2}} = \frac{\mathbf{F}_{\mathsf{L}} + \mathbf{F}_{\mathsf{R}}}{2} - \frac{|\tilde{\mathbf{A}}|_{i+\frac{1}{2}}}{2}(\mathbf{U}_{\mathsf{R}} - \mathbf{U}_{\mathsf{L}}) \tag{2.61}$$

Roe[13, 80] constructs the conservative linearization for the $|\tilde{\mathbf{A}}|_{i+\frac{1}{2}}$ matrix by introducing the parameter vector,

$$\mathbf{Z} = \sqrt{\rho} \left\{ \begin{array}{c} 1 \\ u \\ H \end{array} \right\} \tag{2.62}$$

The $i + \frac{1}{2}$ state is taken to be a linear average of the parameter vector,

$$\bar{\mathbf{Z}}_{i+\frac{1}{2}} = \frac{\mathbf{Z}_{\mathsf{L}} + \mathbf{Z}_{\mathsf{R}}}{2}$$

Taking the velocity and total enthalpy from the parameter vector,

$$\tilde{u} = \frac{\bar{Z}_2}{\bar{Z}_1} \qquad \tilde{H} = \frac{\bar{Z}_3}{\bar{Z}_1} \tag{2.63}$$

and defining the Roe-density,

$$\tilde{\rho} = \sqrt{\rho_{\mathsf{L}}\rho_{\mathsf{R}}} \tag{2.64}$$

the Jacobian matrix is formed as,

$$|\tilde{\mathbf{A}}| = \tilde{\mathbf{X}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{X}}^{-1} \tag{2.65}$$

The eigenvalues are,

$$\mathbf{\Lambda} = \mathrm{diag}(u,\ u+a,\ u-a) \tag{2.66}$$

The right eigenvectors are,

$$\mathbf{X}^{(1)} = \begin{Bmatrix} 1 \\ u \\ \frac{u^2}{2} \end{Bmatrix} \quad \mathbf{X}^{(2)} = \begin{Bmatrix} 1 \\ u+a \\ H+ua \end{Bmatrix} \quad \mathbf{X}^{(3)} = \begin{Bmatrix} 1 \\ u-a \\ H-ua \end{Bmatrix} \tag{2.67}$$

The product $\tilde{\mathbf{X}}^{-1}(\mathbf{U}_{\mathsf{R}} - \mathbf{U}_{\mathsf{L}})$ results in the characteristic variables,

$$\tilde{\mathbf{X}}^{-1}(\mathbf{U}_{\mathsf{R}} - \mathbf{U}_{\mathsf{L}}) = \tilde{\mathbf{X}}^{-1}d\mathbf{U} = \frac{1}{2\tilde{a}^2}\begin{Bmatrix} 2\tilde{a}^2 d\rho - 2dP \\ dP + \tilde{\rho}\tilde{a}\,du \\ dP - \tilde{\rho}\tilde{a}\,du \end{Bmatrix} \tag{2.68}$$

The sound speed is,

$$a^2 = \frac{\gamma P}{\rho} = \gamma(\gamma-1)e = (\gamma-1)h = (\gamma-1)(H - \frac{u^2}{2}) \tag{2.69}$$

Also note the grouping $\tilde{\rho}\,du$ can be constructed as,

$$\tilde{\rho}\,du = \bar{Z}_1 dZ_2 - \bar{Z}_2 dZ_1 \tag{2.70}$$

As for the scalar case, first-order spatial accuracy is obtained by taking the right state to be $i+1$ and the left state at i. Higher-order accuracy is obtained using gradient reconstruction (Eqn. 2.16) applied either to each of the conserved variables (Eqn.2.58) or each of the primitive variables, which are,

$$\mathbf{V} = \begin{Bmatrix} \rho \\ u \\ P \end{Bmatrix} \tag{2.71}$$

The nodal update is still formed as in Eqn. 2.8. The residual remains as expressed in Eqn. 2.40, but for systems rather than scalar quantities.

Fluctuation splitting

The Euler flux (Eqn. 2.59) can be written in terms of the parameter vector,

$$\mathbf{F} = \left\{ \begin{array}{c} Z_1 Z_2 \\ \frac{\gamma-1}{\gamma} Z_1 Z_3 + \frac{\gamma+1}{2\gamma} Z_2^2 \\ Z_2 Z_3 \end{array} \right\} \tag{2.72}$$

Further, the derivative of the flux is,

$$d\mathbf{F} = \left[\begin{array}{ccc} Z_2 & Z_1 & 0 \\ \frac{\gamma-1}{\gamma} Z_3 & \frac{\gamma+1}{\gamma} Z_2 & \frac{\gamma-1}{\gamma} Z_1 \\ 0 & Z_3 & Z_2 \end{array} \right] d\mathbf{Z} \tag{2.73}$$

By assuming a linear variation of the parameter vector on each element, the fluctuation is obtained from Eqn. 2.41 as,

$$\phi_{\mathsf{E}} = -\int_{\Omega} \mathbf{F}_x \, d\Omega = -\int_{\Omega} \mathbf{F}_Z \mathbf{Z}_x \, d\Omega = -\bar{\mathbf{F}}_Z \Delta_i \mathbf{Z} \tag{2.74}$$

Deconinck[36] *et al.* show,

$$\bar{\mathbf{F}}_Z \Delta_i \mathbf{Z} = \tilde{\mathbf{A}} \Delta_i \mathbf{U} = \Delta_i \mathbf{F} \tag{2.75}$$

when the Roe-averaged forms (Eqns. 2.63 and 2.64) are used to obtain $\tilde{\mathbf{A}}$.

An upwind scheme is constructed by adding the artificial dissipation,

$$\phi'_{\mathsf{E}} = -|\tilde{\mathbf{A}}|_{i+\frac{1}{2}} \Delta_i \mathbf{U} \tag{2.76}$$

where $|\tilde{\mathbf{A}}|$ is defined in Eqn. 2.65. Employing the same distribution formula as for the scalar advection (Eqn. 2.44) leads to an update formula analogous to Eqn. 2.45, showing the equivalence between DMFDSFV and fluctuation splitting for the one-dimensional Euler equations.

Before ending the fluctuation splitting discussion, it is desired to frame the artificial dissipation in the form,

$$\phi'_{\mathsf{E}} = \mathrm{sign}(\tilde{\mathbf{A}}_{i+\frac{1}{2}}) \phi_{\mathsf{E}} \tag{2.77}$$

The difficulty lies in defining the matrix $\text{sign}(\tilde{\mathbf{A}})$. One approach combines Eqns. 2.65, 2.74, 2.75, 2.76, and 2.77 to form,

$$\begin{aligned}\text{sign}(\tilde{\mathbf{A}})\tilde{\mathbf{A}} &= |\tilde{\mathbf{A}}| = \tilde{\mathbf{X}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{X}}^{-1} \\ \text{sign}(\tilde{\mathbf{A}}) &= \tilde{\mathbf{X}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{X}}^{-1}\tilde{\mathbf{A}}^{-1} = \tilde{\mathbf{X}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{\Lambda}}^{-1}\tilde{\mathbf{X}}^{-1}\end{aligned} \tag{2.78}$$

Sidilkover[1] offers an alternative to brute force matrix multiplications for evaluating Eqn. 2.78. Introduce the auxiliary variables, $\mathbf{W}$, defined by the transformation,

$$d\mathbf{U} = \mathbf{U}_W d\mathbf{W} \tag{2.79}$$

where,

$$d\mathbf{W} = \begin{Bmatrix} ds \\ \tilde{\rho} du \\ dP \end{Bmatrix} \tag{2.80}$$

with the first Riemann variable defined as,

$$ds = d\rho - \frac{dP}{a^2} \tag{2.81}$$

The Jacobian of the transformation is,

$$\mathbf{U}_W = \begin{bmatrix} 1 & 0 & \frac{1}{a^2} \\ u & 1 & \frac{u}{a^2} \\ \frac{u^2}{2} & u & \frac{1}{\gamma-1} + \frac{u^2}{2a^2} \end{bmatrix} \tag{2.82}$$

and its inverse is,

$$\mathbf{U}_W^{-1} = \begin{bmatrix} 1-(\gamma-1)\frac{u^2}{2a^2} & (\gamma-1)\frac{u}{a^2} & -(\gamma-1)\frac{1}{a^2} \\ -u & 1 & 0 \\ (\gamma-1)\frac{u^2}{2} & -(\gamma-1)u & \gamma-1 \end{bmatrix} \tag{2.83}$$

The element fluctuation (Eqn. 2.74) can be reworked,

$$\phi_{\mathsf{E}} = -\tilde{\mathbf{A}}\Delta_i\mathbf{U} = -\mathbf{U}_W\mathbf{U}_W^{-1}\tilde{\mathbf{A}}\mathbf{U}_W\mathbf{U}_W^{-1}\Delta_i\mathbf{U} = -\mathbf{U}_W\tilde{\mathcal{A}}\Delta_i\mathbf{W} = \mathbf{U}_W\breve{\phi}_{\mathsf{E}} \tag{2.84}$$

where $\breve{\phi}_{\mathsf{E}}$ is the fluctuation as computed for the auxiliary variables,

$$\breve{\phi}_{\mathsf{E}} = -\tilde{\mathcal{A}}\Delta_i\mathbf{W} \tag{2.85}$$

The flux Jacobian of the auxiliary variable formulation is obtained from the conserved flux Jacobian via the similarity transformation,

$$\tilde{\mathcal{A}} = \mathbf{U}_W^{-1}\tilde{\mathbf{A}}\mathbf{U}_W = \mathbf{U}_W^{-1}\tilde{\mathbf{X}}\tilde{\mathbf{\Lambda}}\tilde{\mathbf{X}}^{-1}\mathbf{U}_W = \tilde{\mathcal{X}}\tilde{\mathbf{\Lambda}}\tilde{\mathcal{X}}^{-1} \tag{2.86}$$

so the eigenvalue matrix, $\mathbf{\Lambda}$ (Eqn. 2.66), remains unaltered. The right eigenvectors are obtained from Eqns. 2.67 and 2.83,

$$\mathcal{X} = \mathbf{U}_W^{-1}\mathbf{X}$$

$$\mathcal{X}^{(1)} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix} \quad \mathcal{X}^{(2)} = \begin{Bmatrix} 0 \\ a \\ a^2 \end{Bmatrix} \quad \mathcal{X}^{(3)} = \begin{Bmatrix} 0 \\ -a \\ a^2 \end{Bmatrix} \tag{2.87}$$

The inverse is easily computed to be,

$$\mathcal{X}^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2a} & \frac{1}{2a^2} \\ 0 & -\frac{1}{2a} & \frac{1}{2a^2} \end{bmatrix} \tag{2.88}$$

The flux Jacobian is evaluated from Eqn. 2.86,

$$\tilde{\mathcal{A}} = \begin{bmatrix} \tilde{u} & 0 & 0 \\ 0 & \tilde{u} & 1 \\ 0 & \tilde{a}^2 & \tilde{u} \end{bmatrix} \tag{2.89}$$

which corresponds to the following non-conservative form of the Euler equations,

$$\begin{aligned} s_t + u s_x &= 0 \\ \rho u_t + u\rho u_x + P_x &= 0 \\ P_t + a^2\rho u_x + u P_x &= 0 \end{aligned} \tag{2.90}$$

Having developed an alternative method for obtaining the elemental fluctuation (Eqn. 2.84), the artificial dissipation can be addressed (Eqn. 2.77).

$$\phi'_{\mathsf{E}} = \mathrm{sign}(\tilde{\mathbf{A}})\phi_{\mathsf{E}} = \mathbf{U}_W\mathbf{U}_W^{-1}\mathrm{sign}(\tilde{\mathbf{A}})\mathbf{U}_W\breve{\phi}_{\mathsf{E}} = \mathbf{U}_W\breve{\phi}'_{\mathsf{E}} \tag{2.91}$$

where,

$$\breve{\phi}'_{\mathsf{E}} = \mathbf{U}_W^{-1}\text{sign}(\tilde{\mathbf{A}})\mathbf{U}_W\breve{\phi}_{\mathsf{E}} = \text{sign}(\tilde{\mathcal{A}})\breve{\phi}_{\mathsf{E}} \tag{2.92}$$

and with the aid of Eqns. 2.78 and 2.86,

$$\text{sign}(\tilde{\mathcal{A}}) = \mathbf{U}_W^{-1}\text{sign}(\tilde{\mathbf{A}})\mathbf{U}_W = \mathbf{U}_W^{-1}\tilde{\mathbf{X}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{\Lambda}}^{-1}\tilde{\mathbf{X}}^{-1}\mathbf{U}_W = \tilde{\mathcal{X}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{\Lambda}}^{-1}\tilde{\mathcal{X}}^{-1} \tag{2.93}$$

Using the eigenvalue and eigenvector definitions (Eqns. 2.66, 2.87, and 2.88) sign($\tilde{\mathcal{A}}$) is evaluated to be,

$$\text{sign}(\tilde{\mathcal{A}}) = \begin{bmatrix} \text{sign}(\tilde{u}) & 0 & 0 \\ 0 & \frac{1}{2}\left[\text{sign}(\tilde{u}+\tilde{a}) + \text{sign}(\tilde{u}-\tilde{a})\right] & \frac{1}{2\tilde{a}}\left[\text{sign}(\tilde{u}+\tilde{a}) - \text{sign}(\tilde{u}-\tilde{a})\right] \\ 0 & \frac{\tilde{a}}{2}\left[\text{sign}(\tilde{u}+\tilde{a}) - \text{sign}(\tilde{u}-\tilde{a})\right] & \frac{1}{2}\left[\text{sign}(\tilde{u}+\tilde{a}) + \text{sign}(\tilde{u}-\tilde{a})\right] \end{bmatrix} \tag{2.94}$$

By considering two cases, for subsonic and supersonic conditions, Eqn. 2.94 takes on simple forms,

$$\text{sign}(\tilde{\mathcal{A}}) = \begin{cases} \mathbf{M}^{sup} & \text{if} \quad |\tilde{u}| > \tilde{a} \\ \mathbf{M}^{sub} & \text{if} \quad |\tilde{u}| < \tilde{a} \end{cases} \tag{2.95}$$

where,

$$\mathbf{M}^{sup} = \text{sign}(\tilde{u})I \tag{2.96}$$

and,

$$\mathbf{M}^{sub} = \begin{bmatrix} \text{sign}(\tilde{u}) & 0 & 0 \\ 0 & 0 & \frac{1}{\tilde{a}} \\ 0 & \tilde{a} & 0 \end{bmatrix} \tag{2.97}$$

2.4.2 Navier-Stokes Equations

The Navier-Stokes equations[87, 88] for the flow of a perfect gas are written in one-dimensional conservation law form (Eqn. 2.57) with $\mathbf{U}$ defined in Eqn. 2.58 and the flux defined as,

$$\mathbf{F} = \mathbf{F}^i - \mathbf{F}^v \tag{2.98}$$

where the inviscid flux, $\mathbf{F}^i$, is the same as the Euler flux (Eqn. 2.59). The viscous flux is,

$$\mathbf{F}^v = \begin{Bmatrix} 0 \\ \tau_{xx} \\ u\tau_{xx} - q_x \end{Bmatrix} \tag{2.99}$$

Using Stokes' hypothesis the stress is,

$$\tau_{xx} = \frac{4}{3}\mu u_x \tag{2.100}$$

Fourier's law for heat flow gives,

$$q_x = -\kappa T_x \tag{2.101}$$

The thermal conductivity is related to the viscosity through the Prandtl number,

$$P_r = \frac{\mu c_p}{\kappa} \tag{2.102}$$

where for air $P_r = 0.72$[89]. The temperature is obtained from the perfect gas equation of state,

$$T = \frac{P}{\rho \Re} \tag{2.103}$$

The inviscid flux is discretized in the manner of section 2.4.1. The contributions from the viscous flux to the nodal update is obtained in a Galerkin sense using the system analog to Eqn. 2.54. No viscous contribution is made to the continuity equation.

Using the linear variation of the parameter vector over an element, the velocity gradient is locally defined on an element[36, 34],

$$(u_x)_{\mathsf{E}} = \overline{\left(\frac{z_2}{z_1}\right)_x} = \overline{\frac{z_{2_x}}{z_1} - \frac{z_2}{z_1^2} z_{1_x}} = \frac{1}{\bar{z}_1}\left(\Delta_i z_2 - \tilde{u}\Delta_i z_1\right) = \frac{\tilde{\rho}}{\bar{z}_1^2}\Delta_i u = \frac{\tilde{\rho}}{\tilde{\tilde{\rho}}}\Delta_i u \tag{2.104}$$

where,

$$\tilde{\tilde{\rho}} = \bar{z}_1^2 = \left(\frac{\sqrt{\rho_i} + \sqrt{\rho_{i+1}}}{2}\right)^2 \tag{2.105}$$

is called the consistent density average. The viscous contribution to the momentum equation can now be expressed,

$$\int_{\mathsf{E}} \frac{4}{3} \upsilon_x \mu u_x \, d\Omega = \frac{4}{3} \upsilon_x \bar{\mu} \frac{\tilde{\rho}}{\tilde{\tilde{\rho}}} \Delta_i u \tag{2.106}$$

The first term of the viscous energy flux is evaluated in a similar manner,

$$\int_{\mathsf{E}} \frac{4}{3} \upsilon_x \mu u u_x \, d\Omega = \frac{4}{3} \upsilon_x \bar{\mu} \tilde{u} \frac{\tilde{\rho}}{\tilde{\tilde{\rho}}} \Delta_i u \tag{2.107}$$

The second term requires some manipulation. Begin by defining the temperature gradient over an element,

$$(T_x)_{\mathsf{E}} = \overline{\left(\frac{P}{\rho\Re}\right)_x} = \overline{\frac{P}{\rho\Re}\left(\frac{P_x}{P} - \frac{\rho_x}{\rho}\right)} = \frac{\tilde{a}^2}{\gamma\Re}\left(\frac{\Delta_i P}{\tilde{P}} - \frac{\Delta_i \rho}{\tilde{\tilde{\rho}}}\right) \tag{2.108}$$

where,

$$\tilde{P} = \frac{\tilde{a}^2\tilde{\tilde{\rho}}}{\gamma} \tag{2.109}$$

The heat flow contribution to the viscous flux is then obtained,

$$\int_{\mathsf{E}} \upsilon_x \kappa T_x \, d\Omega = \upsilon_x \bar{\kappa}\frac{\tilde{a}^2}{\gamma\Re}\left(\frac{\Delta_i P}{\tilde{P}} - \frac{\Delta_i \rho}{\tilde{\tilde{\rho}}}\right) = \upsilon_x \frac{\bar{\mu}c_p}{P_r}\frac{\tilde{a}^2}{\gamma\Re}\left(\frac{\Delta_i P}{\tilde{P}} - \frac{\Delta_i \rho}{\tilde{\tilde{\rho}}}\right) \tag{2.110}$$

The elemental contributions from the viscous terms is similar to Eqn. 2.55,

$$\begin{aligned} S_i\mathbf{U}_{i_t} &\leftarrow \frac{\bar{\mu}_{i+\frac{1}{2}}}{\ell_{i,i+1}}\left\{\begin{array}{c} 0 \\ \frac{4}{3}\frac{\tilde{\rho}}{\tilde{\tilde{\rho}}}\Delta_i u \\ \frac{4}{3}\tilde{u}\frac{\tilde{\rho}}{\tilde{\tilde{\rho}}}\Delta_i u + \frac{c_p}{P_r}\frac{\tilde{a}^2}{\gamma\Re}\left(\frac{\Delta_i P}{\tilde{P}} - \frac{\Delta_i \rho}{\tilde{\tilde{\rho}}}\right) \end{array}\right\} + \text{COE} \\ S_{i+1}\mathbf{U}_{i+1_t} &\leftarrow -\frac{\bar{\mu}_{i+\frac{1}{2}}}{\ell_{i,i+1}}\left\{\begin{array}{c} 0 \\ \frac{4}{3}\frac{\tilde{\rho}}{\tilde{\tilde{\rho}}}\Delta_i u \\ \frac{4}{3}\tilde{u}\frac{\tilde{\rho}}{\tilde{\tilde{\rho}}}\Delta_i u + \frac{c_p}{P_r}\frac{\tilde{a}^2}{\gamma\Re}\left(\frac{\Delta_i P}{\tilde{P}} - \frac{\Delta_i \rho}{\tilde{\tilde{\rho}}}\right) \end{array}\right\} + \text{COE} \end{aligned} \tag{2.111}$$

As discussed for the scalar advection-diffusion equations, when solving the Navier-Stokes equations the maximum of the viscous contribution to the nodal update and the artificial dissipation from the inviscid flux discretization should be utilized. When the physical viscous terms are large enough, no artificial dissipation is needed.

2.5 Finite Volume State of the Art

The one-dimensional analysis of the Navier-Stokes equations represents the state of art for upwind flux difference split finite volume schemes. Extensions of the unstructured finite volume method to multiple spatial dimensions relies upon solving a locally one-dimensional approximate Riemann problem across cell faces, an approach that looses some of the coupling present in a system of equations. Locally one-dimensional

solution techniques also introduce preferential grid-aligned wave directions that may not correspond with physical wave-propagation directions.

The fluctuation splitting framework approaches the governing equations from a different perspective than finite volume, but is seen to result in identical discretizations in one dimension for the DMFDSFV scheme. However, the following chapters show how fluctuation splitting generalizes to multiple dimensions in a more compact and coupled manner than locally one-dimensional finite volume schemes.

Chapter 3

Two-Dimensional Scalar Analysis

Having shown the equivalence of the fluctuation splitting and DMFDSFV schemes for one-dimensional domains, the extensions to two spatial dimensions are considered. The present chapter analyzes the case for a single governing conservation law,

$$U_t + \vec{\nabla} \cdot \vec{F} = \vec{\nabla} \cdot (\mu \vec{\nabla} U) \tag{3.1}$$

to which steady-state solutions are sought. Systems, *e.g.* the Navier-Stokes equations, are deferred to a subsequent chapter.

DMFDSFV is extended in an upwind, edge-based formulation for general unstructured meshes with a multi-dimensional reconstruction. The crucial piece of the solver remains a locally one-dimensional approximate Riemann evaluator, as developed in chapter 2. This locally one-dimensional treatment of the fluxes results in increased production of artificial dissipation, particularly when discontinuities are not aligned with the mesh[40].

The extension of the fluctuation splitting scheme to multiple dimensions takes on the flavor of a node-based upwind residual-distribution algorithm, resulting in a

greater flexibility to propagate multi-dimensional wave phenomenon without dissipation. Fluctuation splitting has a more-compact stencil than DMFDSFV for second-order accuracy and exhibits "zero cross-diffusion"[1] in a grid-aligned condition. Fluctuation splitting is seen to grid-resolve advection-diffusion problems on coarser meshes than DMFDSFV.

This chapter begins by defining the different elemental domain structures for DMFDSFV and fluctuation splitting. Then formulations are developed for the two schemes and applied to two-dimensional linear and non-linear advection model problems. Observations are made about the effects in both schemes of grid orientation on the production of artificial dissipation. Discretizations for diffusion follow, along with results for the heat equation. The chapter concludes with an advection-diffusion test problem, revealing the improved accuracy and decreased solution times using fluctuation splitting *vis a vis* DMFDSFV.

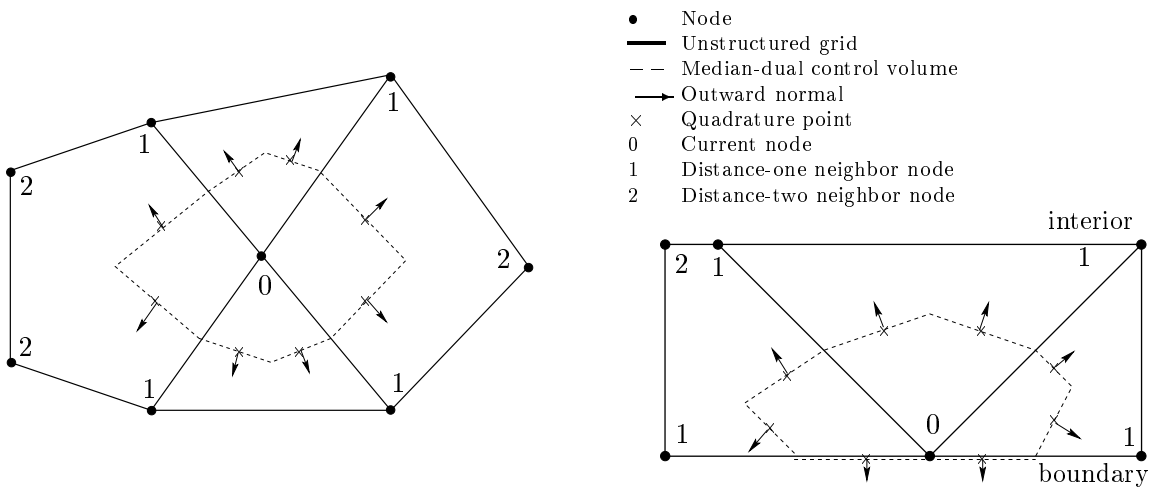
3.1 Domain

The DMFDSFV scheme is implemented for an edge-based data structure. The domain is discretized on an unstructured mesh of arbitrary connectivity. Control volumes are then constructed about each node. One common method for defining the control volumes is to use the median-dual mesh, shown as the dashed lines in Figure 3.1. For a triangulated domain, the generalized median-dual volume about a node equals one-third the sum of the areas of each triangle connected at that node,

$$S_i = \frac{1}{3} \sum_{\forall \mathsf{T} \,|\, i \in \mathsf{T}} S_{\mathsf{T}} \tag{3.2}$$

The fluxes into and out of the control volumes are efficiently computed as a sum of contributions distributed to the nodes from a loop over edges. For each edge, the fluxes through the control faces to the right and to the left of the edge, Figure 3.2, are computed, with the convention that a positive flux is out of the control

[1] "Zero cross-diffusion" refers to the practice of adding artificial dissipation terms in the streamwise direction only, as opposed to adding artificial dissipation in both the streamwise and cross-stream directions.



(a) Interior node.

(b) Boundary node.

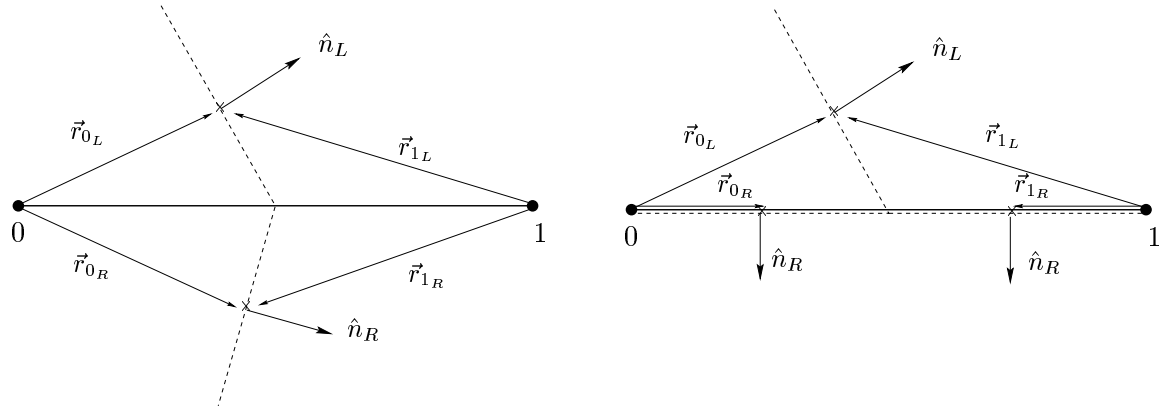
Figure 3.1: Finite volume computational domain for edge-based implementation.

volume surrounding the initiation node and into the control volume surrounding the termination node. The fluxes are evaluated at the quadrature points, and data is reconstructed from the nodes to the quadrature points, denoted by $\times$, along the vectors $\vec{r}$ in Figure 3.2.

For the special case of a boundary edge, shown in Figure 3.2(b), two fluxes are computed to the right-hand side of the edge, one for each associated node.

The DMFDSFV scheme is referred to as a locally one-dimensional scheme because the fundamental Riemann problem is approximately solved normal to a face, with the solution going to either or both of two nodes only, connected by the physical mesh edge. The mesh edge and control volume face can have an arbitrary orientation within multi-dimensional space. The reconstruction step in general can be multi-dimensional.

In contrast to the edge being the fundamental computational element for DMFDSFV, fluctuation splitting is formulated on a multi-dimensional simplex element. In two dimensions the simplex element is the triangle, while in three-dimensions the simplex is a tetrahedron. A pictorial of the domain nomenclature for fluctuation splitting is presented in Figure 3.3. Local curvilinear coordinates, (ξ, η), are defined



(a) Interior edge. (b) Boundary edge.

Figure 3.2: Flux quadrature for edge-based finite volume scheme. Solid bold line is physical mesh, dashed lines are control-volume faces.

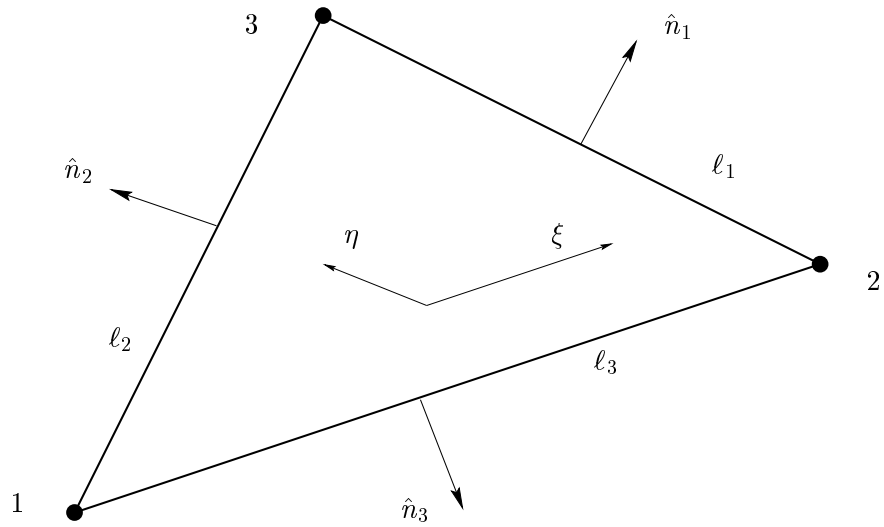


Figure 3.3: Elemental triangular domain for fluctuation splitting.

on each triangle, parallel to two of the sides. A good choice of sides for the curvilinear coordinates to minimize computer round-off error may be the two most orthogonal sides. The fluctuation computed on a triangle may now be sent to one or more of the three vertices (versus two for DMFDSFV), allowing for a truly multi-dimensional distribution scheme.

By definition, all elements are interior to the domain, so no special domain discretization is needed for the boundaries with fluctuation splitting. However, numerical boundary conditions will still need to be applied at the boundary nodes to account for contributions from 'ghost' elements outside the computational domain.

3.2 Advection

Pure advection is obtained from Eqn. 3.1 when $\mu = 0$,

$$U_t + \vec{\nabla} \cdot \vec{F} \ = 0 \tag{3.3}$$

This section extends the DMFDSFV procedure in a straight-forward manner from the one-dimensional analysis in chapter 2. Then the fluctuation splitting method is applied, in a formulation that now differs significantly from DMFDSFV. A temporal pseudo-time marching solution procedure follows, including a positivity analysis yielding timestep restrictions. A statement on the boundary conditions concludes the analytic formulations, leading to results for both linear and non-linear test cases.

3.2.1 Formulations

DMFDSFV

The traditional, locally one-dimensional, approximate Riemann solver finite volume scheme[22] begins by integrating Eqn. 3.3 over the control volumes and applying the divergence theorem,

$$\int_{\Omega} U_t \, d\Omega \ = - \oint_{\Gamma} \vec{F} \cdot \hat{n} \, d\Gamma \tag{3.4}$$

Using mass lumping to the nodes, similar to an explicit finite element treatment[85], the temporal evolution is evaluated on a time-invariant mesh as,

$$\int_{\Omega} U_t \, d\Omega \; = S_i \frac{\partial U_i}{\partial t} \rightarrow \frac{S_i}{\Delta t} \left(U_i^{t+\Delta t} - U_i^t \right) \tag{3.5}$$

The discretization of the convective flux, $\vec{F}$, is performed using Barth's implementation[22] of the upwind, locally one-dimensional, approximate Riemann solver of Roe[13] by constructing the numerical fluxes as a combination of the flux function and artificial dissipation,

$$\oint_{\Gamma} \vec{F} \cdot \hat{n} \, d\Gamma \; \simeq \sum_{faces} f_{face} \Delta\Gamma = -R_i \tag{3.6}$$

The numerical flux at the face is analogous to the one dimensional form (see Eqns. 2.12 and 2.38),

$$f_{face} \; = \frac{1}{2} \left(\vec{F}_{in} + \vec{F}_{out} \right) \cdot \hat{n} - \Phi \tag{3.7}$$

where the artificial dissipation provides the upwinding (see Eqns. 2.14 and 2.38),

$$\Phi \; = \frac{1}{2} |\tilde{\vec{A}} \cdot \hat{n}| (U_{out} - U_{in}) \tag{3.8}$$

Out and *in* refer to states on the outside and inside of Ω at the face. The flux Jacobians are defined,

$$A^{\mathsf{x}} \; = \frac{\partial F^{\mathsf{x}}}{\partial U}, \qquad A^{\mathsf{y}} = \frac{\partial F^{\mathsf{y}}}{\partial U} \tag{3.9}$$

where $\vec{A} = A^{\mathsf{x}}\hat{\imath} + A^{\mathsf{y}}\hat{\jmath}$, $\vec{F} = F^{\mathsf{x}}\hat{\imath} + F^{\mathsf{y}}\hat{\jmath}$, and the tilde indicates the conservative linearizations at the cell face[13].

Piecewise linear reconstruction from the nodal unknowns to the cell faces as in Eqn. 2.16, repeated here,

$$U_{face} \; = U_i + \psi \vec{\nabla} U \cdot \vec{r} \tag{3.10}$$

provides second-order spatial accuracy in smoothly-varying regions of the solution. Median-dual gradients of the dependent variable, $\vec{\nabla} U$, are obtained from the unweighted least squares procedure outlined by Barth. Following Bruner and Walters[90], the limiter is supplied an argument equal to half the argument Barth uses, namely,

$$\psi \; = \psi \left(\frac{U^{min/max} - U_i}{2(\vec{\nabla} U \cdot \vec{r})^{min/max}} \right) \tag{3.11}$$

where $U^{min/max}$ is the minimum (resp. maximum) of U_i and all distance-one neighbors. The most restrictive limiting from choosing the minimum or maximum is used.

In casting the limiter argument in this form, Bruner equates the Barth limiter with Superbee, for a limiter argument less than or equal to one. For the full domain of the argument, the non-symmetric Barth limiter takes the form,

$$\psi\left(\frac{p}{q}\right) = \begin{cases} 0 & \frac{p}{q} \leq 0 \\ 2\frac{p}{q} \quad \text{if} & 0 < \frac{p}{q} < \frac{1}{2} \\ 1 & \frac{p}{q} \geq \frac{1}{2} \end{cases} \tag{3.12}$$

for the limiter cast as Eqn. 3.11. Many limiter functions exist, and several of the more popular versions are detailed in appendix A.

The DMFDSFV flux evaluation needs to be performed three times for each interior triangle, once for each edge of the triangle. For linear advection at a uniform advection speed Eqn. 3.8 requires 4 multiplication/divisions and 2 additions/subtractions. Equation 3.7 requires 4 each multiplications and additions. Ignoring the work required to compute the nodal gradients and limiter, which varies based upon the mesh connectivity, the reconstruction, Eqn. 3.10, requires 6 multiplications and 4 additions per face. Equation 3.6 adds one more multiplication per face, bringing the total operation count for DMFDSFV per triangle to 45 multiplications and 30 additions.

Fluctuation splitting

The Narrow Non-Linear (NNL) fluctuation splitting scheme is presented as a slight re-interpretation of the work of Sidilkover and Roe[3]. The current interpretation is as a volume integral over triangular elements, without recourse to the divergence theorem. The discretized equations, however, are identical to Sidilkover's. This form of fluctuation splitting employs a general limiter function for determining the residual distributions.

Integrating Eqn. 3.3 over an element, where Ω is now the area of the triangular element,

$$\int_\Omega U_t \, d\Omega = -\int_\Omega \vec{\nabla} \cdot \vec{F} \, d\Omega \tag{3.13}$$

For linear variation of the dependent variable over the element, the temporal evolution is (see also Eqn. 2.26),

$$\int_\Omega U_t \, d\Omega \;= S_\mathsf{T} \bar{U}_t = \frac{S_\mathsf{T}}{3}(U_{1_t} + U_{2_t} + U_{3_t}) \tag{3.14}$$

where U_1, U_2, and U_3 correspond to the three nodes defining element Ω.

Using the local curvilinear coordinates (ξ, η), defined in Figure 3.3, the divergence of the convective flux can be transformed,

$$\vec{\nabla} \cdot \vec{F} \;= F_x^\mathsf{x} + F_y^\mathsf{y} = \frac{\partial}{\partial x}(\vec{F} \cdot \hat{\imath}) + \frac{\partial}{\partial y}(\vec{F} \cdot \hat{\jmath}) \tag{3.15}$$

$$\begin{Bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{Bmatrix} = \begin{bmatrix} \xi_x & \eta_x \\ \xi_y & \eta_y \end{bmatrix} \begin{Bmatrix} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{Bmatrix} \tag{3.16}$$

$$\vec{\nabla} \cdot \vec{F} \;= \xi_x(\vec{F} \cdot \hat{\imath})_\xi + \xi_y(\vec{F} \cdot \hat{\jmath})_\xi + \eta_x(\vec{F} \cdot \hat{\imath})_\eta + \eta_y(\vec{F} \cdot \hat{\jmath})_\eta \tag{3.17}$$

Introducing the inverse Jacobian of the coordinate transformation,

$$J^{-1} \;= x_\xi y_\eta - x_\eta y_\xi = \frac{2S_\mathsf{T}}{\ell_1 \ell_3} \tag{3.18}$$

and the invariants of the transformation,

$$\begin{aligned} \xi_x &= \frac{y_\eta}{J^{-1}} = \frac{\hat{n}_1 \cdot \hat{\imath}}{J^{-1}} & \xi_y &= -\frac{x_\eta}{J^{-1}} = \frac{\hat{n}_1 \cdot \hat{\jmath}}{J^{-1}} \\ \eta_x &= -\frac{y_\xi}{J^{-1}} = -\frac{\hat{n}_3 \cdot \hat{\imath}}{J^{-1}} & \eta_y &= \frac{x_\xi}{J^{-1}} = -\frac{\hat{n}_3 \cdot \hat{\jmath}}{J^{-1}} \end{aligned} \tag{3.19}$$

Eqn. 3.17 becomes,

$$\vec{\nabla} \cdot \vec{F} \;= \frac{1}{J^{-1}} \left(\hat{n}_1 \cdot \vec{F}_\xi - \hat{n}_3 \cdot \vec{F}_\eta \right) = \frac{\ell_1 \ell_3}{2S_\mathsf{T}} \left(\hat{n}_1 \cdot \vec{A} U_\xi - \hat{n}_3 \cdot \vec{A} U_\eta \right) \tag{3.20}$$

If $\vec{F}$ is linear or quadratic in U, then for a linear variation of U over the element,

$$\int_\Omega \vec{\nabla} \cdot \vec{F} \, d\Omega \;= \alpha \Delta_\xi U + \beta \Delta_\eta U \tag{3.21}$$

where the difference operators are $\Delta_\xi U = U_2 - U_1$ and $\Delta_\eta U = U_3 - U_2$ and the advection speeds are,

$$\alpha \;= \frac{\ell_1}{2} \hat{n}_1 \cdot \tilde{\vec{A}}, \qquad \beta = -\frac{\ell_3}{2} \hat{n}_3 \cdot \tilde{\vec{A}} \tag{3.22}$$

$\tilde{\vec{A}}$ is now the conservative linearization over the entire triangular element[36].

The advective fluctuation can be defined,

$$\phi = -\int_{\Omega} \vec{\nabla} \cdot \vec{F} \, d\Omega \tag{3.23}$$

The fluctuation can be split,

$$\phi = \phi^{\xi} + \phi^{\eta} \tag{3.24}$$

where,

$$\phi^{\xi} = -\alpha \Delta_{\xi} U, \qquad \phi^{\eta} = -\beta \Delta_{\eta} U \tag{3.25}$$

Following Sidilkover[1] the scheme is extended to second-order spatial accuracy by repartitioning the fluctuation through the use of a symmetric limiter function,

$$\phi^{*\xi} = \phi^{\xi} + \phi^{\eta}\psi(Q) = \phi^{\xi}\left(1 - \frac{\psi(Q)}{Q}\right) \tag{3.26}$$

$$\phi^{*\eta} = \phi^{\eta} - \phi^{\eta}\psi(Q) = \phi^{\eta}\left(1 - \psi(Q)\right) \tag{3.27}$$

with,

$$Q = -\frac{\phi^{\xi}}{\phi^{\eta}} \tag{3.28}$$

In practice, if an averaging function, M_{ψ}, exists for the desired limiter, it is numerically advantageous to compute $M_{\psi}(\phi^{\eta}, -\phi^{\xi}) = \phi^{\eta}\psi(Q)$, avoiding the need to evaluate Q explicitly.

This critical step, allowing the redistribution of the fluctuation, is what principally distinguishes the multi-dimensional fluctuation splitting scheme of Sidilkover from a locally one-dimensional extension of Riemann solvers. There is no analog to this in the formulations of chapter 2.

Upwinding is achieved through the introduction of the artificial dissipation terms,

$$\phi'^{\xi} = \mathrm{sign}(\alpha)\phi^{*\xi}, \qquad \phi'^{\eta} = \mathrm{sign}(\beta)\phi^{*\eta} \tag{3.29}$$

Combining Eqn. 3.14 with a distribution scheme for Eqn. 3.23 and summing over

all elements, the contributions to nodal time derivatives can be written in the form,

$$\begin{aligned} S_1 U_{1_t} &\leftarrow \frac{1}{2}(\phi^{*\xi} - \phi^{\prime\xi}) + COE \\ S_2 U_{2_t} &\leftarrow \frac{1}{2}(\phi^{*\xi} + \phi^{\prime\xi}) + \frac{1}{2}(\phi^{*\eta} - \phi^{\prime\eta}) + COE \\ S_3 U_{3_t} &\leftarrow \frac{1}{2}(\phi^{*\eta} + \phi^{\prime\eta}) + COE \end{aligned} \tag{3.30}$$

or in a more compact form,

$$\begin{aligned} S_i U_{i_t} \leftarrow \frac{1}{4}\Big[& i(3-i)(\phi^{*\xi} + (-1)^i \phi^{\prime\xi}) + (-4 + 5i - i^2)(\phi^{*\eta} - (-1)^i \phi^{\prime\eta})\Big] \\ & + COE \qquad\qquad i = 1, 2, 3 \end{aligned} \tag{3.31}$$

where COE stands for contributions from other elements containing these nodes.

The distribution requires 4 addition/subtractions and 3 multiplication/divisions. Upwinding requires 2 multiplications for each term of Eqn. 3.29. Ignoring the cost of evaluating the limiter function, as was done when counting the DMFDSFV operations, one more multiplication and addition are performed in each of Eqns. 3.26 and 3.27 and for each term of Eqn. 3.25. Finally, α and β, Eqn. 3.22, each require 4 multiplications and one addition for a total operation count of only 19 multiplications, versus 45 for the DMFDSFV scheme. Only 10 additions per triangle are required by fluctuation splitting, versus 30 for DMFDSFV.

3.2.2 Advective Timestep Restriction

Both schemes are formulated as either Gauss-Seidel or Jacobi time-relaxation algorithms.

The nodal updates for the discrete system can be formed as a sum of contributions from all nodes.

$$U_i^{t+\Delta t} = \sum_j c_j U_j = c_i U_i + \sum_{j \neq i} c_j U_j \tag{3.32}$$

For positivity[56] each of the coefficients in Eqn. 3.32 must be non-negative.

In the finite volume context the nodal update (Eqn. 3.32) can be rearranged into

the form of Eqn. 3.5,

$$\frac{S_i}{\Delta t}(U_i^{t+\Delta t} - U_i^t) \;= \frac{S_i}{\Delta t}(c_i - 1)U_i + \frac{S_i}{\Delta t}\sum_{j \neq i} c_j U_j \tag{3.33}$$

For the upwind, edge-based algorithm considered here, each $\frac{S_i}{\Delta t}c_j$ will be related to a positive-definite coefficient equal to zero for outflowing faces and related to the wavespeed for inflowing faces, yielding the restriction $\Delta t \geq 0$ on the timestep. The remaining term can be expressed,

$$\frac{S_i}{\Delta t}(c_i - 1) \;= -\sum_{k \text{ about } i} c_k \tag{3.34}$$

where the c_k coefficients are also positive-definite, either zero for inflowing faces or related to the wavespeed for outflowing faces. Rearranging and imposing the positivity constraint, $c_i \geq 0$, yields the timestep restriction,

$$1 - \frac{\Delta t}{S_i}\sum_{k \text{ about } i} c_k \;= c_i \geq 0 \tag{3.35}$$

$$\Delta t \;\leq \frac{S_i}{\sum_{k \text{ about } i} c_k} \tag{3.36}$$

For fluctuation splitting, the nodal updates are assembled from Eqn. 3.31 as,

$$\frac{S_i}{\Delta t}(U_i^{t+\Delta t} - U_i^t) \;= \sum_{j \neq i} c_j (U_j - U_i) \tag{3.37}$$

In this case the c_j coefficients are formed as contributions from the fluctuations in the triangles to both the left and the right of mesh edge $\overline{ij}$. The positivity restriction on Δt is found to have a similar form as for finite volume (Eqn. 3.36),

$$\Delta t \;\leq \frac{S_i}{\sum_{j \neq i} c_j} \tag{3.38}$$

Local time-stepping based on positivity is shown to yield stable, yet non-converging, solutions in some second-order cases (see section 3.4). Robust convergence is obtained by using the first-order c's in Eqns. 3.36 and 3.38, even for second-order-accurate spatial discretizations. This is similar to the common practice of using a first-order Jacobian discretization in a time-implicit scheme.

An implicit scheme can be constructed for fluctuation splitting by linearizing the flux in time,

$$\vec{F}^{t+\Delta t} = \vec{F}^{t} + \vec{A}^{t}\left(U^{t+\Delta t} - U^{t}\right) \tag{3.39}$$

The spatial discretization remains the same and the coefficients of the LHS of Eqn. 3.37 are changed from the scalar $\frac{S_i}{\Delta t}$ to the operator,

$$\frac{S_i}{\Delta t} + \alpha\Delta_\xi + \beta\Delta_\eta \tag{3.40}$$

A full implicit scheme would require inverting the sparse matrix corresponding to the operator of Eqn. 3.40. Since both Δ_ξ and Δ_η operators use a nearest-neighbor compact stencil, creative re-ordering of the nodes could pack the implicit matrix into a banded diagonal of average bandwidth 6. Another strategy is to use a variant of a red/black coloring, here grouping the nodes into three sets because of the triangular connectivity. The three sets of nodes would then be integrated with a line Gauss-Seidel strategy, where the matrix to invert has been reduced to a simple diagonal form for each set of nodes. The present dissertation does not implement a fully-implicit temporal integration.

3.2.3 Boundary Conditions

Explicit Dirichlet inflow boundary conditions are employed. Advective outflow boundaries are treated for free convection through the boundary nodes, allowing boundary nodes to be handled in the same manner as interior nodes.

3.2.4 Results

Linear

The linear advection equation is obtained from Eqn. 3.3 for a flux function $\vec{F} = \vec{\lambda}U$, yielding,

$$U_t + \vec{\nabla}\cdot(\vec{\lambda}U) = 0 \tag{3.41}$$

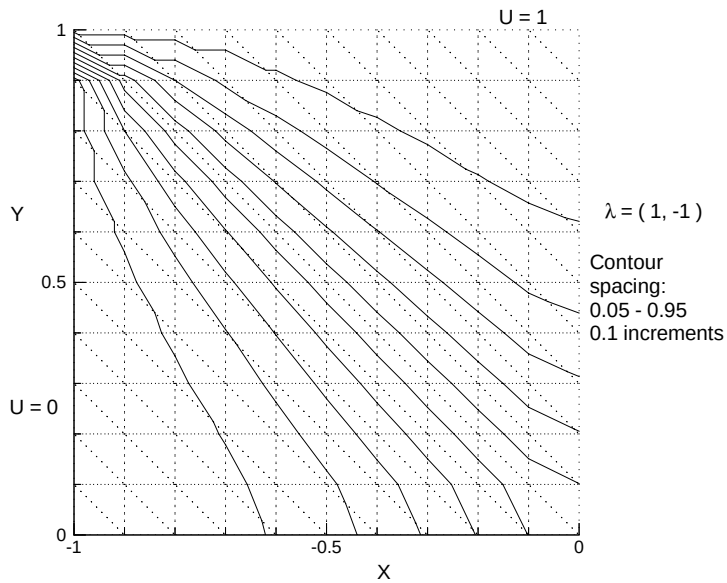


Figure 3.4: First-order fluctuation splitting, uniform advection.

A divergence-free advection velocity is considered, such that $\vec{\nabla}\cdot\vec{\lambda} = 0$. Equation 3.41 can then be written,

$$U_t + \vec{\lambda} \cdot \vec{\nabla} U \ = 0 \tag{3.42}$$

The appropriate averaged flux Jacobian for linear advection is simply $\tilde{\vec{A}} = \vec{\lambda}$, where $\vec{\lambda}$ is evaluated at the quadrature points for DMFDSFV and at the element centroid for fluctuation splitting.

Uniform advection

Uniform advection of the Heavyside function at -45 degrees, $\vec{\lambda} = (1, -1)$, on a cut-cartesian mesh is shown for first-order fluctuation splitting, second-order fluctuation splitting, and second-order DMFDSFV in Figures 3.4–3.6, respectively. The mesh is shown as the dashed background, and equally-spaced contours vary on [0,1], the minimum and maximum solution values. The spread of the contour lines with spatial evolution is indicative of the amount of dissipation introduced into the solution by the discretization of the convective terms.

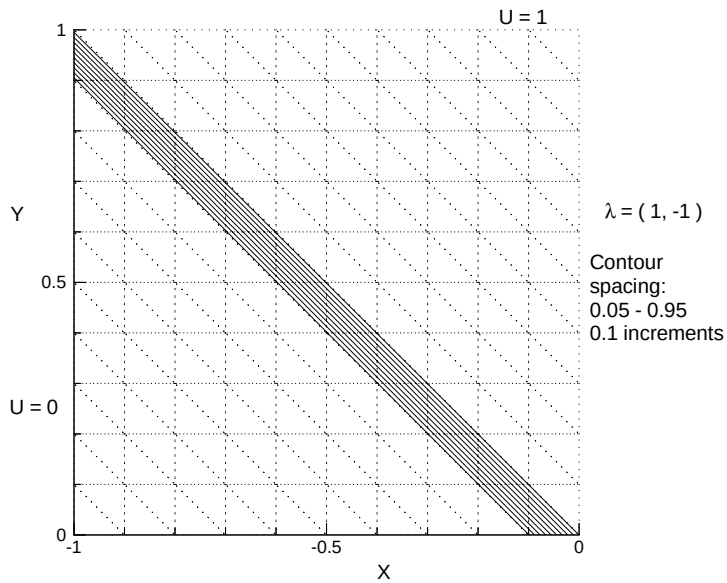


Figure 3.5: Second-order fluctuation splitting, uniform advection.

Second-order fluctuation splitting is seen to be greatly superior to first-order, as expected, reproducing the exact solution in this case with no introduced dissipation. Also, fluctuation splitting represents a significant reduction in numerical diffusion versus the corresponding DMFDSFV scheme, with both results employing the Minmod limiter.

However, the "zero cross-diffusion" results of Figure 3.5 with fluctuation splitting are misleading. In Figure 3.7 the advection velocity has been rotated counter clockwise by 90 degrees on the same grid. Clearly, the artificial dissipation introduced by the fluctuation splitting scheme has been increased.

The corresponding DMFDSFV solution is shown in Figure 3.8. While the change in contour spreading for the DMFDSFV scheme between Figures 3.6 and 3.8 is less dramatic than the change in spreading for the fluctuation splitting scheme in Figures 3.5 and 3.7, the fluctuation splitting results still exhibit less diffusion than the DMFDSFV results, comparing Figures 3.7 and 3.8.

Employing the compressive Superbee limiter with the fluctuation splitting scheme

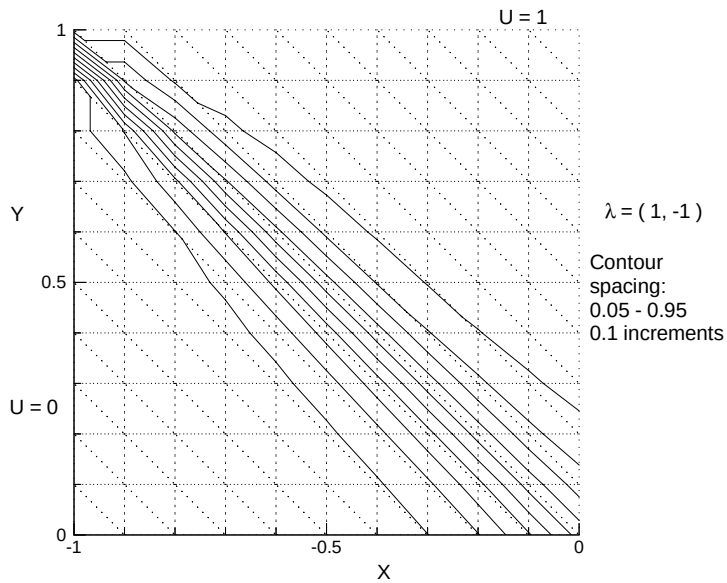


Figure 3.6: Second-order DMFDSFV, uniform advection.

yields the results of Figure 3.9. In this case the discontinuity is confined to a 2–3 cell stencil, and does not grow in space. Applying the Superbee limiter to DMFDSFV cannot eliminate all artificial dissipation for this case, as is possible with fluctuation splitting. The DMFDSFV results (not shown) corresponding to Figure 3.9 spread the discontinuity over four cells by the outflow boundary, with a continually broadening trend.

However, while it is possible to use the Superbee limiter with fluctuation splitting for this case, compressive limiters can be unstable on different grid orientations. For example, no degree of compression is stable for the case of Figure 3.5. This potential for instability is related to global positivity, as discussed by Sidilkover and Roe[3].

The effect of using a general unstructured grid is investigated in Figures 3.10 and 3.11. The unstructured grid in this case was generated using VGRID[91, 92]. The fluctuation splitting solution exhibits less dissipation, but is not as smooth as the DMFDSFV solution. While the fluctuation splitting scheme preserves contact discontinuities over larger spatial ranges than the DMFDSFV scheme, fluctuation

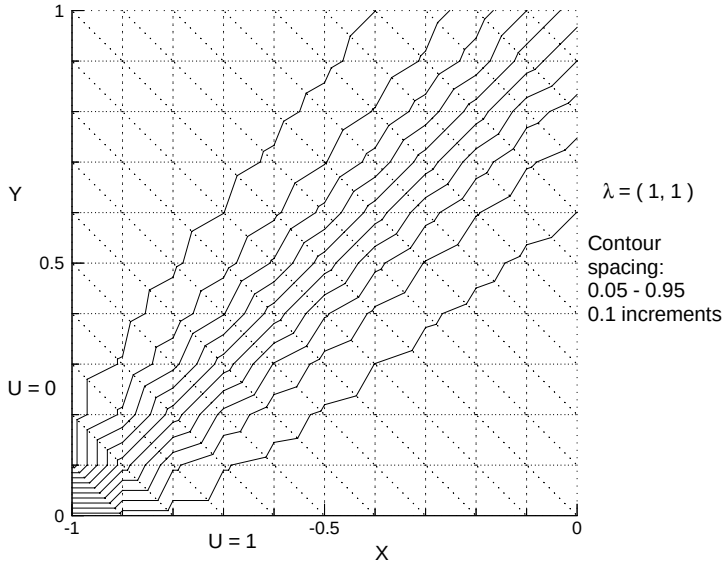


Figure 3.7: Second-order fluctuation splitting, uniform advection.

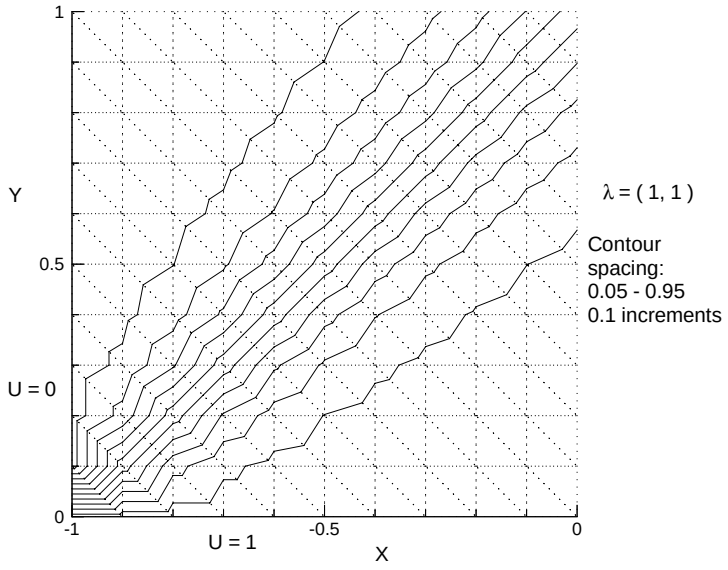


Figure 3.8: Second-order DMFDSFV, uniform advection.

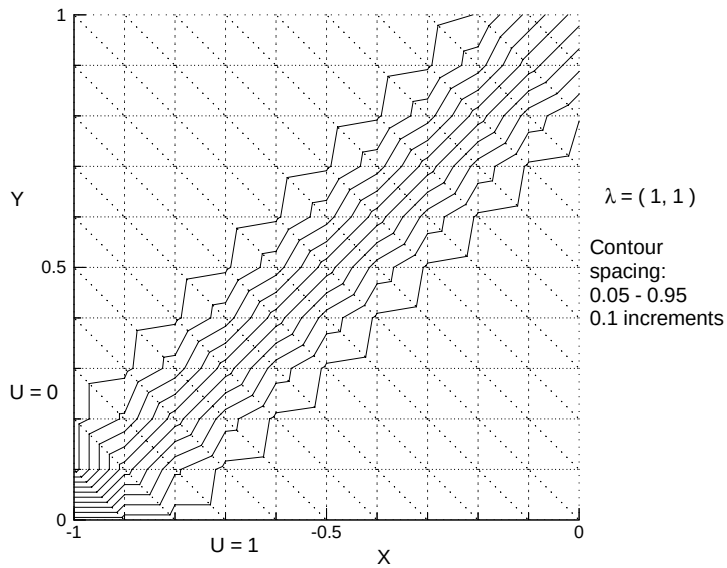


Figure 3.9: Second-order fluctuation splitting with compressive limiter.

splitting does not appear to degenerate gracefully with regard to extreme coarsening of the unstructured mesh for this test case. This behavior could have negative implications for applications employing multigrid convergence acceleration.

Circular advection

Circular advection is achieved by setting $\vec{\lambda} = (y,\ -x)$. A decaying sine-wave input profile is used,

$$U(x,0) \ = (e^x \sin \pi x)^2$$

Results for the two schemes, using the Minmod limiter, are presented on the worse-case cut-cartesian mesh in Figures 3.12 and 3.13. Again, the fluctuation splitting results are considerably less diffusive than the DMFDSFV solution.

The circular-advection problem is also applied on an unstructured mesh. The input profile for this case consists of both a top-hat function and a decaying sine wave, allowing comparisons between the schemes for both sharp discontinuities and

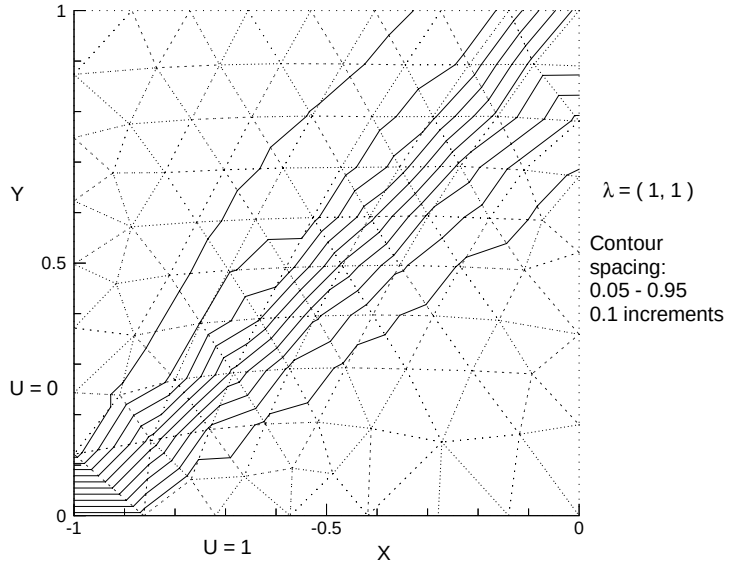


Figure 3.10: Fluctuation splitting on unstructured mesh.

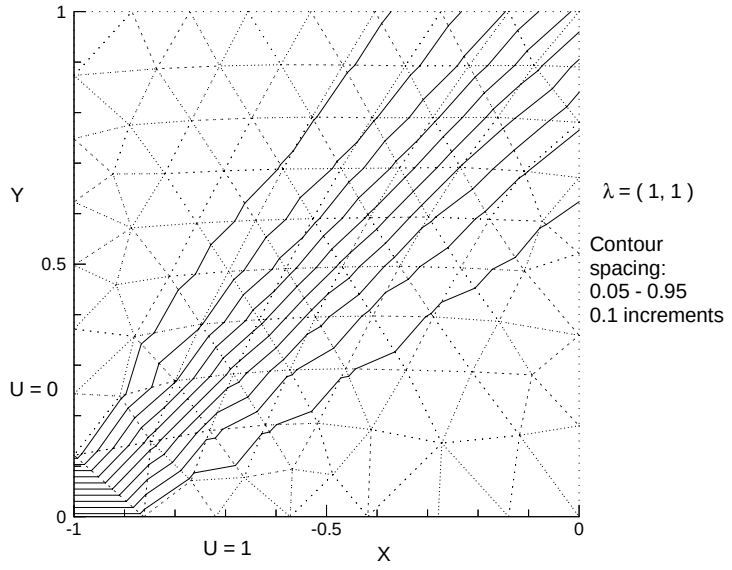


Figure 3.11: DMFDSFV on unstructured mesh.

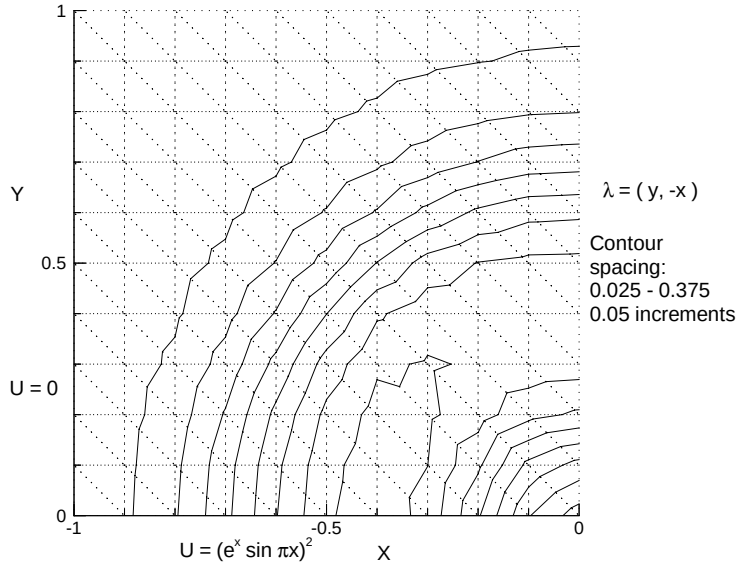


Figure 3.12: Fluctuation splitting, circular advection.

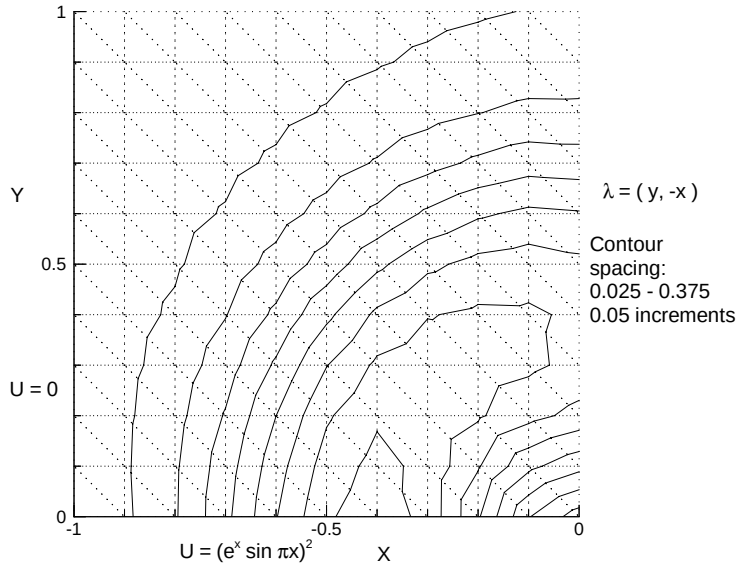


Figure 3.13: DMFDSFV, circular advection.

smooth gradients. The input profile is,

$$U(x,0) \;=\begin{cases} \left(e^{2x}\sin(2\pi x)\right)^2 & -0.5 \le x < 0 \\ 0 & -0.6 \le x < -0.5 \\ 0.4 & -0.8 \le x < -0.6 \\ 0 & -1 \le x < -0.8 \end{cases}$$

Results for this case are displayed in Figure 3.14 for fluctuation splitting and Figure 3.15 for DMFDSFV, both using the Minmod limiter. Fluctuation splitting performs significantly better at preserving the top-hat distribution. Fluctuation splitting also does a better job of maintaining the minimum and maximum values of the sine distribution, though both schemes do well on the smooth gradient portion of the sine wave.

Non-linear

The non-linear advection equation is obtained from Eqn. 3.3 for the flux function $\vec{F} = (\frac{U^2}{2}, U)$. In non-conservative form the equation is written,

$$U_t + UU_x + U_y \;= 0$$

A coalescing shock problem is considered, with an anti-symmetric input profile,

$$\begin{aligned} U(-1,y) \;&= U(0,y) = 0 \\ U(x,0) \;&= -2x - 1 \text{ on } x = (-1,0) \end{aligned}$$

The exact solution to this problem contains symmetric expansion fans on the sides and a compression fan at the inflow that coalesces into a vertical shock at $(x,y) = (-\frac{1}{2}, \frac{1}{2})$.

The first mesh is cut-cartesian containing 26×26 nodes. The fluctuation splitting and DMFDSFV solutions, both using the Minmod limiter, are presented in Figures 3.16 and 3.17. Both algorithms exhibit the same grid dependence on the amount of artificial dissipation as seen before, with the left-half solutions having more diffusion than the right halves, due to the grid orientation. Both methods perform the same in the compression-fan region, coalescing into a shock to within the accuracy of the input-profile discretization.

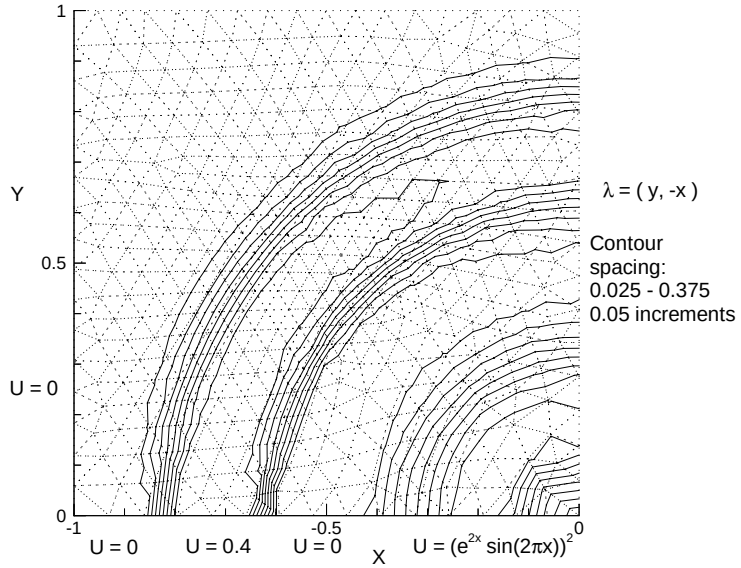


Figure 3.14: Fluctuation splitting on unstructured mesh, circular advection.

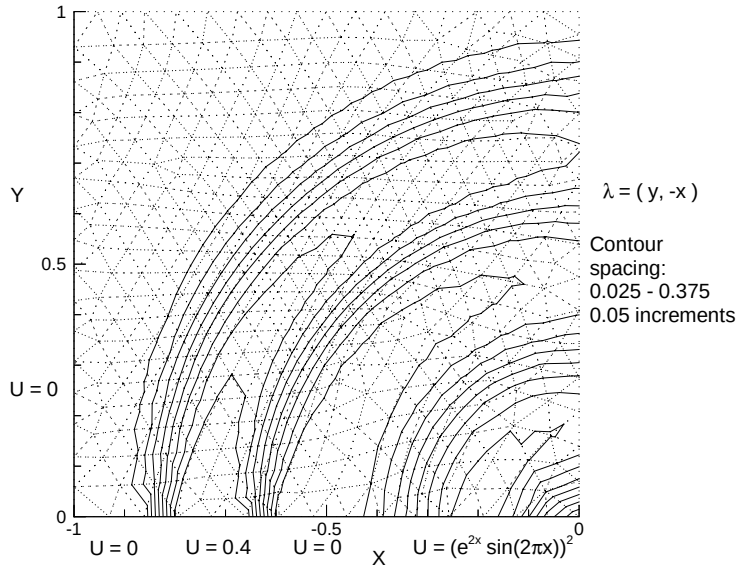


Figure 3.15: DMFDSFV on unstructured mesh, circular advection.

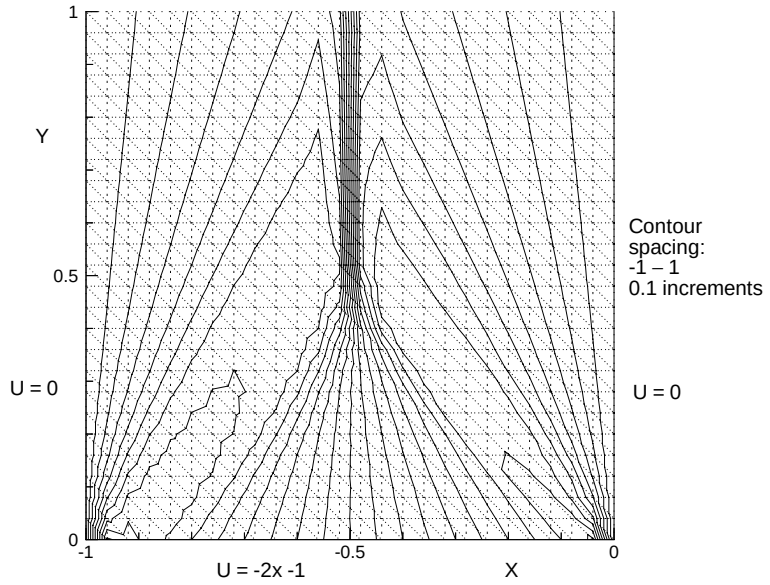


Figure 3.16: Fluctuation splitting, Burgers equation.

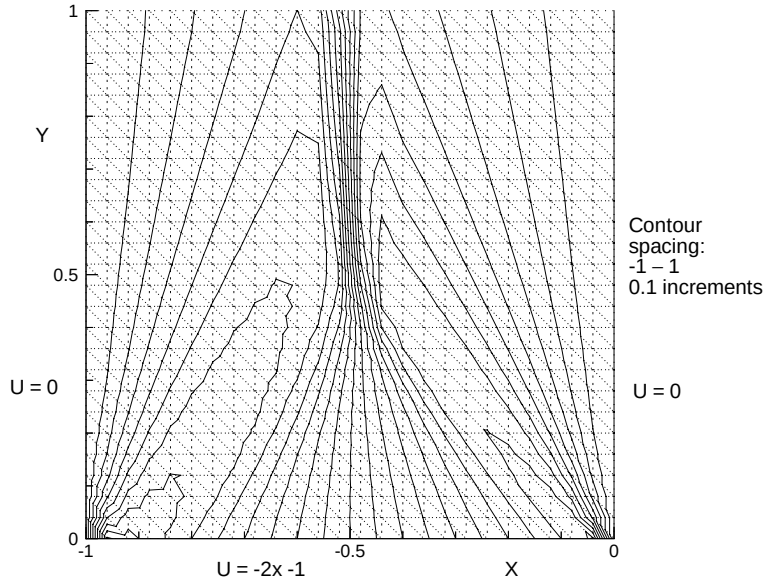


Figure 3.17: DMFDSFV, Burgers equation.

The shock is more sharply defined by fluctuation splitting than by DMFDSFV. Figure 3.16 has the correct shock speed, with nearly the entire gradient captured in one cell thickness. In contrast, Figure 3.17 shows a slightly incorrect shock speed when using DMFDSFV, as the shock progresses to the left beyond the coalescence point, even though the discretization is conservative. The incorrect shock speed results from a non-symmetric distribution of the dependent variable to the left and right of the shock, caused by the excessive artificial diffusion generated on the grid-misaligned (left-hand) side.

Contours of the absolute value of the error are presented in Figures 3.18 and 3.19. Errors from both computed solutions show a lack of symmetry, again reflecting the grid dependence of the artificial diffusion terms. The error levels from fluctuation splitting are less than from DMFDSFV. The shock curvature in the DMFDSFV solution at the coalescing point is clearly visible in Figure 3.19, resulting in significant downstream errors in the shock location as compared with the fluctuation splitting errors.

This problem is repeated on a 25×25 mesh with symmetric diagonal cuts, favorably aligned with the advection directions. The fluctuation splitting and DMFDSFV solutions, Figures 3.20 and 3.21, are in good agreement. Plots of the absolute error contours, Figures 3.22 and 3.23, show fluctuation splitting to be a little more accurate than DMFDSFV for this case.

The final mesh for this case is a truly unstructured triangulation containing 847 nodes and 1617 cells. The nodes are clustered to the outflow boundary, with an arbitrary bias towards the left-hand side, introduced merely as an additional challenge for the schemes. The fluctuation splitting solution is presented in Figure 3.24, showing very accurate and crisp shock resolution and good symmetry in the solution contours despite the mesh-clustering bias. In contrast, the DMFDSFV solution in Figure 3.25 has a more-diffuse shock and again an incorrect shock speed, with the outflow shock offset to the left of $x = -\frac{1}{2}$. The DMFDSFV solution is also somewhat less symmetric than the fluctuation splitting solution.

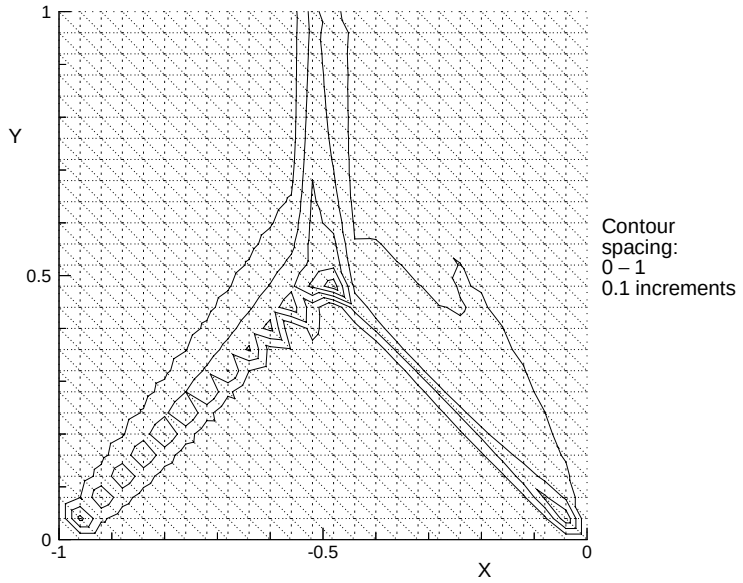


Figure 3.18: Fluctuation splitting, Burgers equation, absolute error.

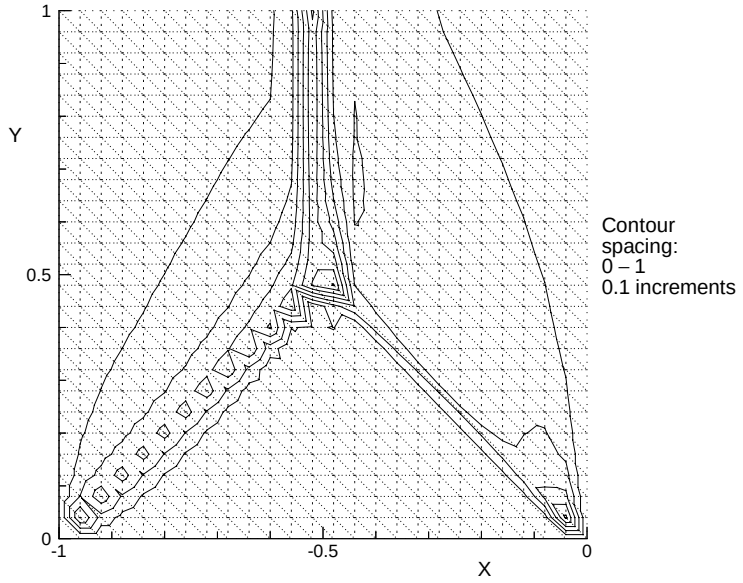


Figure 3.19: DMFDSFV, Burgers equation, absolute error.

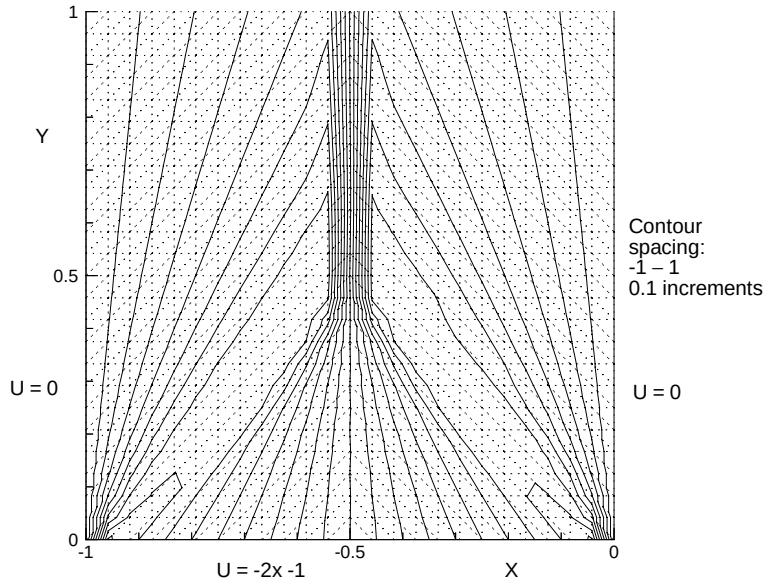


Figure 3.20: Fluctuation splitting, Burgers equation, symmetric mesh.

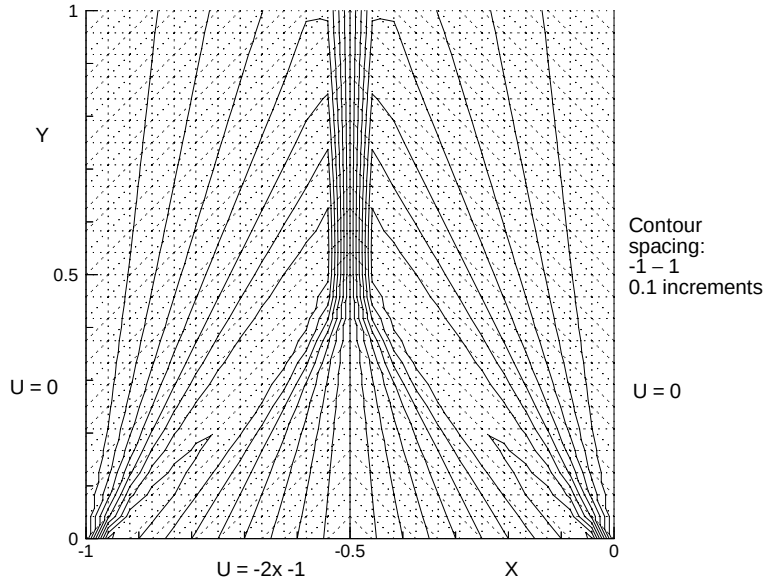


Figure 3.21: DMFDSFV, Burgers equation, symmetric mesh.

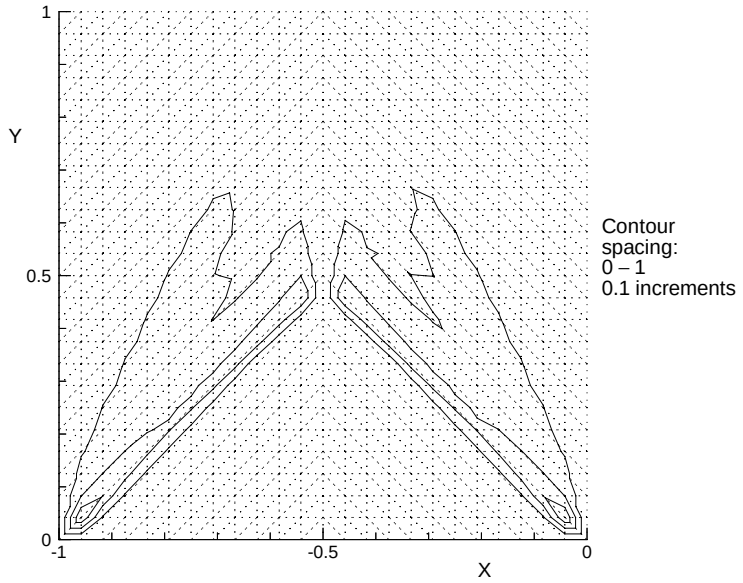


Figure 3.22: Fluctuation splitting, Burgers equation, absolute error.

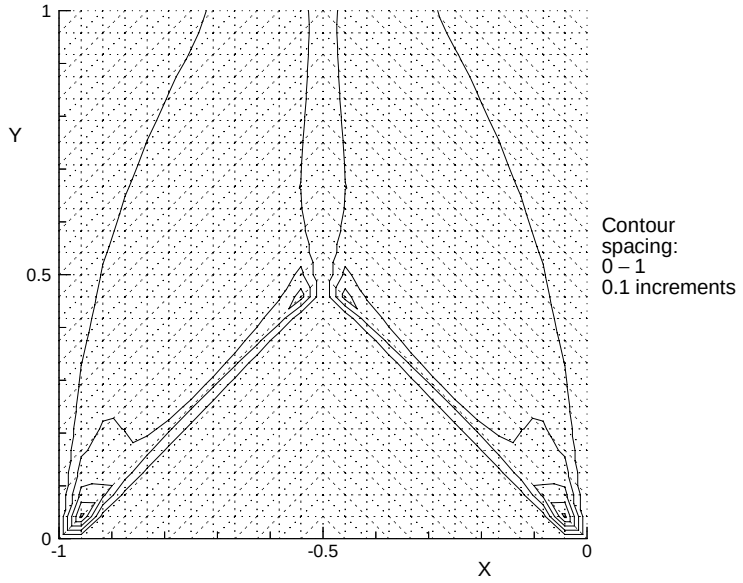


Figure 3.23: DMFDSFV, Burgers equation, absolute error.

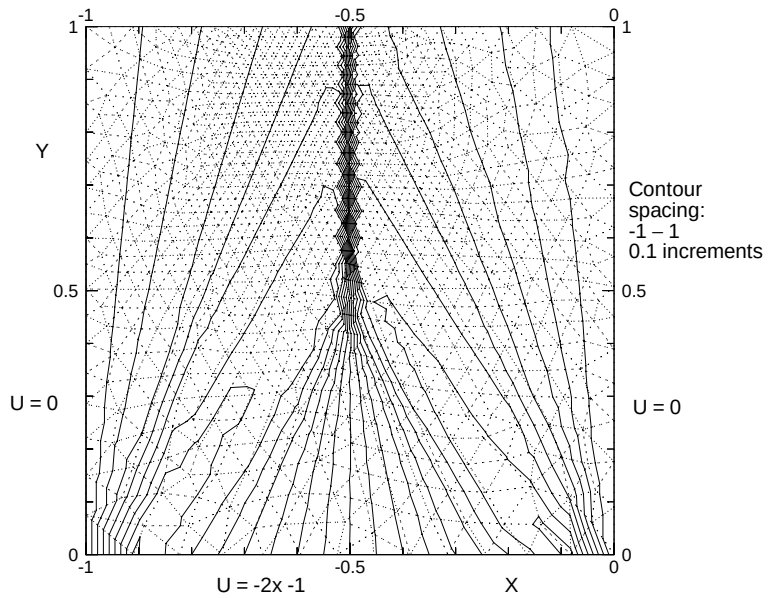


Figure 3.24: Fluctuation splitting, Burgers equation, unstructured mesh.

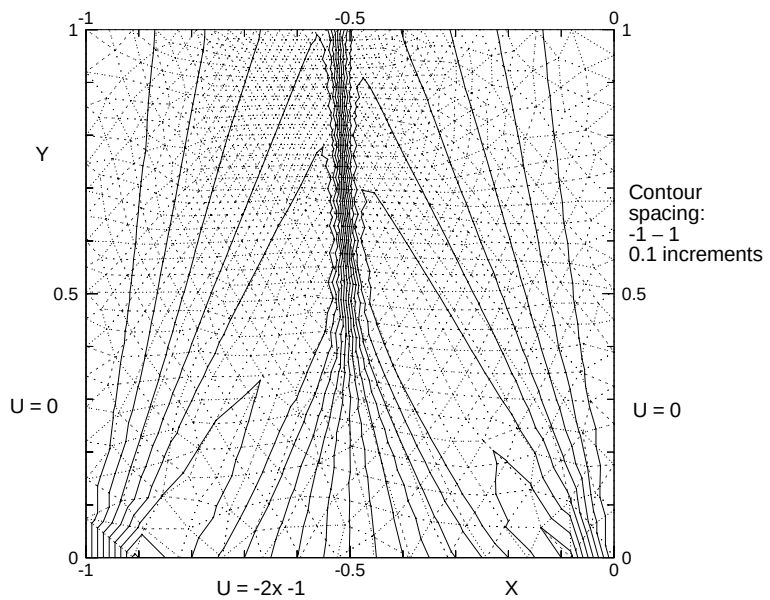


Figure 3.25: DMFDSFV, Burgers equation, unstructured mesh.

3.3 Diffusion

The model diffusion equation, the well known heat equation, is obtained from Eqn. 3.1 when $\vec{F} = 0$,

$$U_t = \vec{\nabla} \cdot (\mu \vec{\nabla} U) \tag{3.43}$$

Lacking cross-derivative terms, the heat equation suffers somewhat as a model for the Navier-Stokes equations.

Two methods for evaluating the diffusion term are detailed. The finite element form has a more compact stencil and will be shown to be more accurate. Either diffusion discretization can be used with either advection discretization, and in practice the finite element diffusion treatment is preferred with both the fluctuation splitting and DMFDSFV advection schemes.

3.3.1 Formulations

Finite volume

In finite volume form, with the aid of the divergence theorem, Eqn. 3.43 becomes,

$$\int_\Omega U_t \, d\Omega = \oint_\Gamma \mu \vec{\nabla} U \cdot \hat{n} \, d\Gamma \tag{3.44}$$

Discretizing the RHS of Eqn. 3.44, in a manner similar to Eqns. 3.6 and 3.7 (see also Eqns. 2.50 and 2.51) yields,

$$\oint_\Gamma \mu \vec{\nabla} U \cdot \hat{n} \, d\Gamma \rightarrow \sum_{faces} \frac{\bar{\mu}}{2} \left(\vec{\nabla} U_{in} + \vec{\nabla} U_{out} \right) \cdot \hat{n} \, \Delta\Gamma \tag{3.45}$$

where in refers to the state inside the control volume and out refers to the state on the other side of the face from the control volume. The diffusion coefficient is averaged over the length of the face. The gradients from Eqn. 3.10 are not limited before averaging at the control-volume faces in Eqn. 3.45, as suggested by Anderson and Bonhaus[83].

Finite element

A finite element treatment, similar to Tomaich[50], is employed. Begin by integrating Eqn. 3.43 over an element, where Ω is now the area of the triangular element,

$$\int_\Omega U_t \, d\Omega \;=\; \int_\Omega \vec{\nabla}\cdot(\mu\vec{\nabla}U)\, d\Omega \tag{3.46}$$

The diffusive fluctuation is defined,

$$\phi^v \;=\; \int_\Omega \vec{\nabla}\cdot(\mu\vec{\nabla}U)\, d\Omega \tag{3.47}$$

Assuming piecewise-linear data and an element-averaged diffusion coefficient leads to a diffusive fluctuation of zero for the triangular element. Introducing the linear nodal shape functions υ_i, such that $\sum_{i=1}^{3}\upsilon_i = 1$, the elemental diffusive fluctuation can be expressed $\phi^v = \sum_{i=1}^{3}\phi_i^v = 0$, where

$$\phi_i^v \;=\; \int_\Omega \upsilon_i\vec{\nabla}\cdot(\mu\vec{\nabla}U)\, d\Omega \tag{3.48}$$

Integrating by parts (analogous to Eqn. 2.53),

$$\phi_i^v \;=\; \oint_\Gamma \upsilon_i\mu\vec{\nabla}U\cdot\hat{n}\, d\Gamma - \int_\Omega \mu\vec{\nabla}U\cdot\vec{\nabla}\upsilon_i\, d\Omega \tag{3.49}$$

The boundary integral in Eqn. 3.49 will cancel for interior nodes on summing contributions from neighboring elements. For linear variation of U over the element the remaining volume integral can be evaluated analytically,

$$\vec{\nabla}U \;=\; \frac{-1}{2S_\mathsf{T}}\sum_{j=1}^{3}U_j\ell_j\hat{n}_j \qquad\qquad \vec{\nabla}\upsilon_i = \frac{-\ell_i\hat{n}_i}{2S_\mathsf{T}} \tag{3.50}$$

$$\phi_i^v \;=\; \frac{-\ell_i}{4S_\mathsf{T}^2}\sum_{j=1}^{3}U_j\ell_j\hat{n}_j\cdot\hat{n}_i\int_\Omega \mu\, d\Omega = -\frac{\bar{\mu}\ell_i}{4S_\mathsf{T}}\sum_{j=1}^{3}U_j\ell_j\hat{n}_j\cdot\hat{n}_i \tag{3.51}$$

where $\bar{\mu}$ is the element-average diffusion coefficient.

The physical dissipation is then distributed to the nodal updates as,

$$S_iU_{i_t} \;\leftarrow \phi_i^v + COE \tag{3.52}$$

3.3.2 Diffusive Timestep Restriction

Continuing the positivity discussion from section 3.2.2, restrictions on the timestep are sought to maintain positive coefficients c_i in Eqn. 3.32.

Unfortunately, the finite element formulation for the diffusive terms (Eqn. 3.51) cannot be guaranteed to preserve local positivity on obtuse triangles (see Barth[22]). Considering only the contributions from the current node, the coefficient for the diffusion term can be written,

$$U_i^{t+\Delta t} = U_i^t \left(1 - \frac{\Delta t}{S_i} \sum_{\forall \mathsf{T} \ni i} \frac{\bar{\mu} \ell_i^2}{4 S_{\mathsf{T}}}\right) \tag{3.53}$$

The resulting timestep restriction is,

$$\Delta t \leq \frac{S_i}{\sum_{\mathsf{T}} \frac{\bar{\mu} \ell_i^2}{4 S_{\mathsf{T}}}} \tag{3.54}$$

In a similar manner the timestep restriction from Eqn. 3.45 is,

$$\Delta t \leq \frac{S_i}{\sum_{\mathsf{T}} \frac{3 \bar{\mu} (\Delta \Gamma)^2}{4 S_{\mathsf{T}}}} \tag{3.55}$$

3.3.3 Boundary Conditions

For the diffusion terms a Neumann outflow boundary is applied with zero gradient normal to the boundary ($\vec{\nabla} U \cdot \hat{n} = 0$), achieved by setting the boundary integral in Eqn. 3.49 to zero.

3.3.4 Heat Equation

The heat equation (Eqn. 3.43) test problem, a steady-state boundary value problem on the unit square in the second quadrant, is taken from Tomaich[50]. The Dirichlet boundary values are,

$$\begin{aligned} U(-1, y) &= 0 & U(0, y) &= \sin(\pi y) \\ U(x, 0) &= 0 & U(x, 1) &= -\sin(\pi x) \end{aligned}$$

The analytical solution on $x = [-1, 0]$, $y = [0, 1]$ is,

$$U(x, y) \ = \frac{1}{\sinh \pi} \left[\sinh(\pi(x+1)) \sin(\pi y) + \ \sinh(\pi y) \sin(\pi(x+1))\right]$$

Both diffusion discretizations, Eqns. 3.45 and 3.51, are compared on a 438-node unstructured mesh. Figures 3.26 and 3.27 plot the absolute value of the error in the converged solutions using Eqns. 3.45 and 3.51, respectively. A carpet plot of the solution, using the finite element formulation, is presented in Figure 3.28.

The finite element treatment is clearly more accurate, and is used to discretize the diffusion terms for both DMFDSFV and fluctuation splitting in the following section. The average-gradient results in Figure 3.26 appear to exhibit a decoupling mode, similar to odd/even decoupling for structured meshes.

3.4 Advection-Diffusion

When considering combined advection and diffusion problems, rather than adding both physical and artificial dissipation terms, it is advantageous to only augment the physical dissipation with as much artificial dissipation as is necessary to ensure monotonicity of the solution. That is, if the physical dissipation term is larger than the computed artificial dissipation, no artificial dissipation is needed.

This rationale can be implemented with fluctuation splitting, for example, by modifying the fluctuation distribution (Eqn. 3.30) to be,

$$\begin{aligned}
S_1 U_{1_t} &\leftarrow \frac{\phi^{*\xi}}{2} + \max\left(-\frac{\phi'^{\xi}}{2}, \phi_1^v\right) + COE \\
S_2 U_{2_t} &\leftarrow \frac{\phi^{*\xi} + \phi^{*\eta}}{2} + \max\left(\frac{(\phi'^{\xi} - \phi'^{\eta})}{2}, \phi_2^v\right) + COE \\
S_3 U_{3_t} &\leftarrow \frac{\phi^{*\eta}}{2} + \max\left(\frac{\phi'^{\eta}}{2}, \phi_3^v\right) + COE
\end{aligned} \tag{3.56}$$

Similarly for DMFDSFV, the artificial dissipation term in Eqn. 3.8 can be compared with the physical diffusion term.

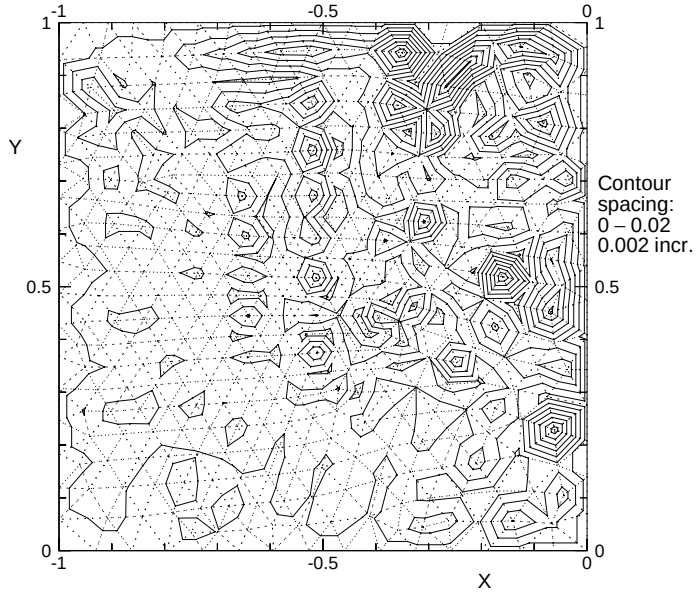


Figure 3.26: Pure-diffusion problem error, diffusion terms from Eqn. 3.45.

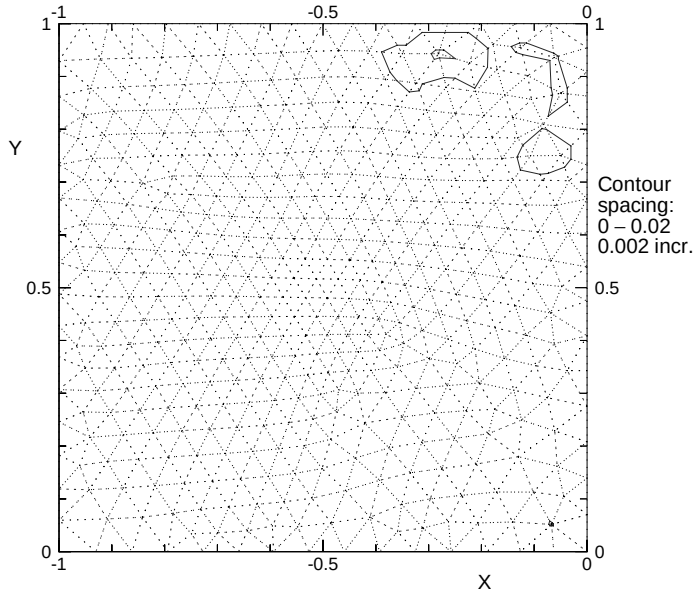


Figure 3.27: Pure-diffusion problem error, diffusion terms from Eqn. 3.51.

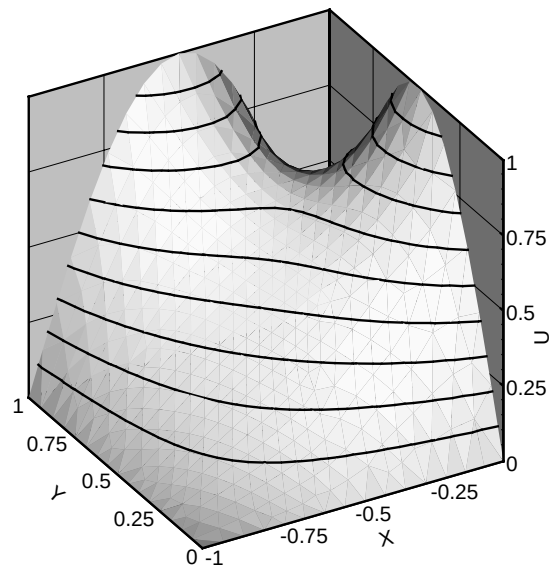


Figure 3.28: Heat equation solution using finite element formulation. Contour increment is 0.1.

Smith & Hutton problem

The final scalar test case is a linear advection-diffusion (Eqn. 3.1) problem due to Smith and Hutton[93]. The flux function is $\vec{F} = \vec{\lambda} U$, with,

$$\vec{\lambda} = \left(2y(1-x^2),\ -2x(1-y^2)\right)$$

The streamlines for this problem, while not truly circular, are similar in orientation to the circular advection problem. The inflow profile is,

$$U(x,0) = 1 + \tanh(20x + 10)$$

The diffusion coefficient is chosen to be a constant, $\mu = 10^{-3}$. The domain is the unit square in the second quadrant. No closed-form solution is known to the author for this problem.

A sequence of five unstructured meshes is considered. The meshes have no preferred clustering or stretching and have nominal node-spacings of 0.1, 0.05, 0.025,

Mesh	Nodes	Fluctuation Splitting	DMFDSFV
		(CPU seconds)	
A	134	< 1	< 1
B	495	1	1
C	1,928	5	8
D	7,529	64	145
E	28,915	760	1880

Table 3.1: Grids and solution times for advection-diffusion problem.

Fluctuation Splitting			DMFDSFV	
$\|\|\phi'\|\|_2$ (art.)	$\|\|\phi^v\|\|_2$ (phys.)	Mesh	$\|\|\Phi\|\|_2$ (art.)	$\|\|\phi^v\|\|_2$ (phys.)
1274	215	A	1918	190
597	265	B	640	176
192	161	C	144	119
54	76	D	46	66
13	36	E	18	36

Table 3.2: L_2-norms ($\times 10^5$) of artificial and physical viscosities for advection-diffusion problem.

0.0125, and 0.00625, labeled as Meshes A–E, respectively. The number of nodes for each mesh, along with the solution times for both fluctuation splitting and DMFDSFV on a 195 MHz SGI R10000 CPU are listed in Table 3.1.

L_2-norms of the artificial and physical viscosities computed using both fluctuation splitting and DMFDSFV are presented for each mesh in Table 3.2. Notice that the norm of the artificial dissipation for both DMFDSFV and fluctuation splitting drops lower than the norm of the physical dissipation on Meshes D and E. However, the norm of the physical dissipation is smaller for DMFDSFV than fluctuation splitting on each mesh A–D. The physical viscosity is driven by the solution curvature, suggesting fluctuation splitting maintains the solution profile sharper than DMFDSFV on the coarser meshes. A comparison of outflow profiles will soon verify this interpretation.

Evidence of a grid-resolved fluctuation splitting solution is seen in Figures 3.29

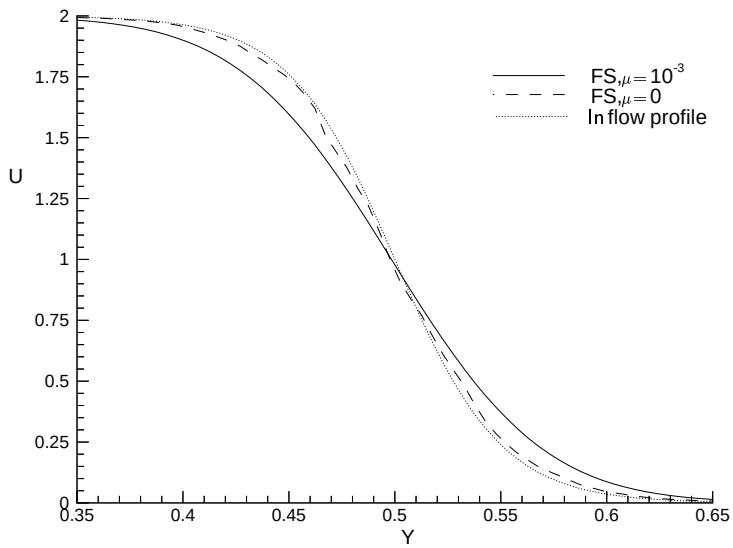


Figure 3.29: Fluctuation splitting profiles on finest mesh, advection-diffusion problem. (FS = fluctuation splitting)

and 3.30. The fluctuation splitting solution on Mesh E at the outflow boundary is presented along with the inflow profile and the corresponding pure-advection ($\mu = 0$) fluctuation splitting solution in Figure 3.29. Because the pure-advection solution is seen to replicate the inflow profile, the fluctuation splitting artificial dissipation is insignificant on this mesh, and hence further grid refinement would not appreciably change the solution. Plotting only the fluctuation splitting results with respect to grid refinement, Figure 3.30 shows a convergence of the outflow profile by Mesh C for fluctuation splitting.

The accuracy of fluctuation splitting and DMFDSFV are compared in Figure 3.31, where the outflow solutions from fluctuation splitting and DMFDSFV are plotted for Meshes C and E. Taking the grid-converged fluctuation splitting Mesh-E solution to be the true solution, it is clear that fluctuation splitting reaches the grid converged solution on a coarser mesh than DMFDSFV.

Computational efficiencies of the two algorithms are compared in Figure 3.32,

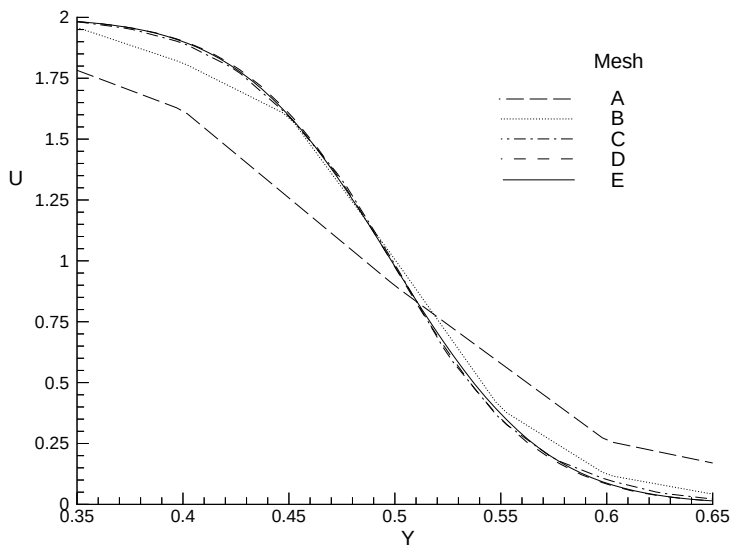


Figure 3.30: Fluctuation splitting grid convergence, advection-diffusion problem.

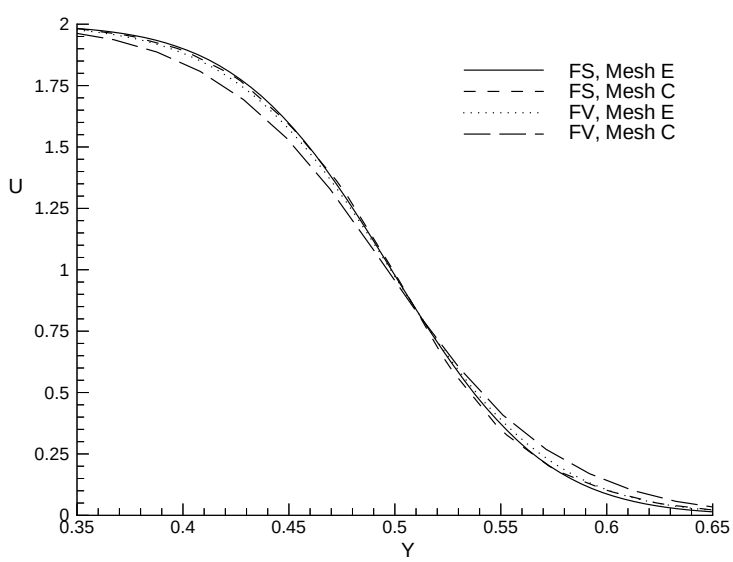


Figure 3.31: Fluctuation splitting and DMFDSFV for advection-diffusion problem. (FS = fluctuation splitting, FV = finite volume)

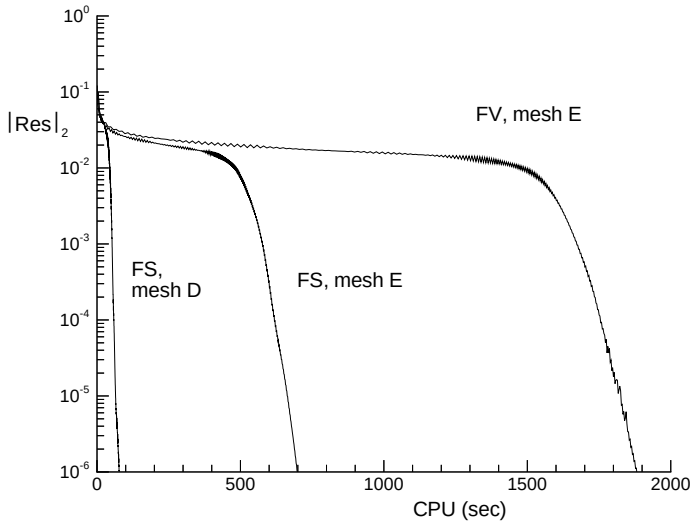


Figure 3.32: Convergence histories for advection-diffusion problem. (FS = fluctuation splitting, FV = finite volume)

where the L_2-norm of the residual is plotted versus CPU time for the fine-mesh fluctuation splitting and DMFDSFV solutions, along with the fluctuation splitting convergence history on Mesh D. The Mesh-E fluctuation splitting solution converges in 760 sec. The corresponding DMFDSFV solution takes 2.5 times longer than fluctuation splitting, due, in part, to the need to reconstruct gradient information at each node with DMFDSFV for second-order spatial accuracy. However, considering the solution time to reach a given accuracy, it is more reasonable to compare the fluctuation splitting solution time on Mesh D to the finest-mesh DMFDSFV solution. The fluctuation splitting Mesh-D solution took only 64 sec, a factor of 29 times less than DMFDSFV on Mesh E, and still shows better accuracy than the fine-mesh DMFDSFV solution.

An even greater speedup is seen with fluctuation splitting in conjunction with the Van Albada limiter, where now the Mesh-B solution over-plots the curve from the finest grid, shown in Figure 3.33. The corresponding DMFDSFV result using the

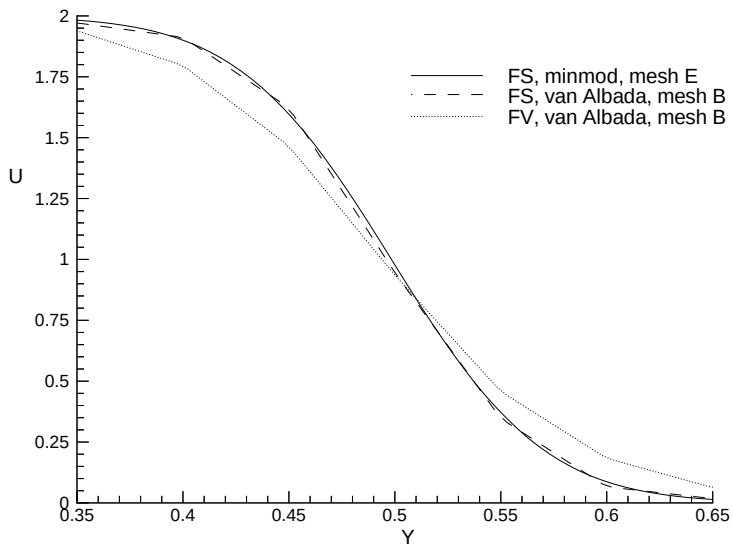


Figure 3.33: Advection-diffusion results using Van Albada limiter. (FS = fluctuation splitting, FV = finite volume)

Van Albada limiter on Mesh B is included, and clearly falls short of the fluctuation splitting accuracy. The DMFDSFV case was repeated with the highly-compressive Superbee limiter with little improvement in accuracy. The solution time for fluctuation splitting on Mesh B is about one second, yielding a speedup factor of 2–3 orders of magnitude over DMFDSFV.

The final set of results addresses convergence issues while pushing the positivity limits. Figure 3.34 compares two convergence histories for the second-order fluctuation splitting on Mesh B. The non-converging, though stable, convergence history is the result of using strict positivity arguments to set the timestep (Eqn. 3.38). The resulting solution is bounded and approximately correct but oscillatory. Limiter "ringing" is the cause of this behavior, with the higher-order discretization for the implicit matrix reducing the diagonal dominance, and hence stability, of the Gauss-Seidel iteration.

Full convergence is achieved by using first-order positivity coefficients, which are

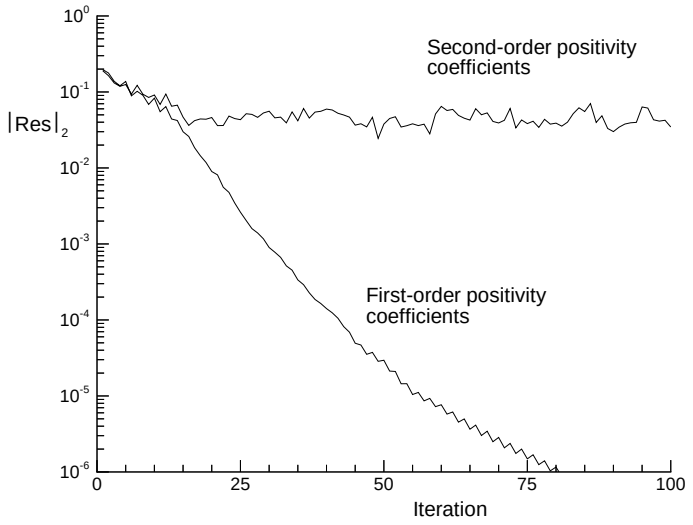


Figure 3.34: Convergence rates using first- and second-order positivity coefficients.

not dependent on the limiters. The resulting local timesteps will not be as large as true second-order positivity would allow, but are more robust.

3.5 Benefits of Fluctuation Splitting

Fluctuation splitting and DMFDSFV schemes have been compared in detail as applied to scalar advection, diffusion, and advection-diffusion problems. The fluctuation splitting scheme is seen to introduce less artificial dissipation while treating advection terms, allowing for more accurate resolution of weakly dissipative advection-diffusion problems. The ability to resolve solutions to these problems on coarser meshes makes the fluctuation splitting scheme the preferred choice over DMFDSFV.

Linear advection test problems were utilized to investigate the dependence of artificial diffusion production on grid orientation. Both fluctuation splitting and DMFDSFV are shown to exhibit grid dependencies, but with fluctuation splitting producing less artificial dissipation on all grids considered.

A non-linear coalescing shock problem further explored grid dependencies as cases were constructed that result in incorrect shock speeds for DMFDSFV. Fluctuation splitting shows correct shock speeds for all grids and provides tighter shock capturing than DMFDSFV.

An advection-diffusion problem with small physical dissipation (diffusion coefficient of 10^{-3}) was considered where the reduction in artificial dissipation with fluctuation splitting results in a significant accuracy improvement over DMFDSFV. Convergence times were compared, showing a speedup of 2.5 for fluctuation splitting over DMFDSFV on identical grids, using a point Gauss-Seidel relaxation. However, a grid convergence study shows fluctuation splitting has better resolution of the solution on a coarser mesh than DMFDSFV does on finer meshes, resulting is a speedup of 29 for fluctuation splitting over DMFDSFV.

Chapter 4

Scalar Mesh Adaption

Mesh adaption strategies, for use in conjunction with the discretization and solution procedures discussed in chapter 3, are sought for two-dimensional scalar problems. Mesh adaption seeks to reduce to a minimum the number of nodes needed to obtain a desired level of solution accuracy. Implicit in this objective is the optimal placement of the retained nodes for a "most-accurate" solution.

Many authors[61, 62, 65, 66, 67, 94] prefer to maintain isotropic (or nearly so) elements while performing unstructured mesh adaption. Their belief is that the accuracy of the solvers degrades on anisotropic elements. However, the use of isotropic elements will lead to excessive mesh densities when resolving phenomenon with disparate length scales in multiple dimensions, *e.g.* plane shocks or boundary layers. A contrasting approach is to employ anisotropically stretched elements, a technique long used for structured meshes[57, 70]. More recently, highly anisotropic elements have been demonstrated with unstructured grids[71, 74, 91, 95].

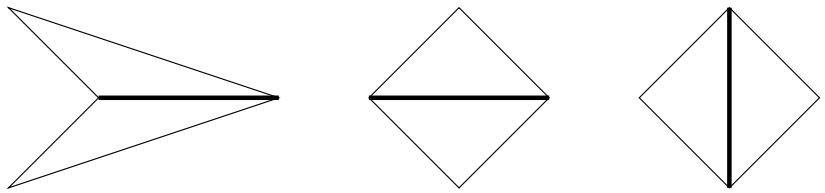
Two primary techniques are commonly used to perform mesh adaptions. One is to perform a global remeshing[96], where the adapted mesh is unrelated to the initial grid. This method works well when the adapted mesh will be significantly different in character from the initial mesh and usually employs an efficient grid-generation code to accomplish the adaption, with the initial solution driving the placement of source terms for the grid generator.

An alternative strategy is to form the adapted mesh from a series of local operations on the initial mesh, based upon the local solution. This approach is very efficient when only a smaller fraction of the total grid requires refinement. Also, local operations can allow for more control over the adapted grid and can be more robust than global grid generators. On the other hand, global remeshings can produce smoother grid distributions than can local operations. The present dissertation will pursue the local adaption strategy, as opposed to global remeshings. As before, unstructured triangulated domains are considered with the solution values stored at the nodes.

This chapter begins by introducing the current state-of-the-art for anisotropic local refinement for finite volume solution methods as advocated by Habashi[71, 74, 75, 76]. Then a consideration of the behavior of fluctuation splitting methods on general meshes is presented. From this, a framework for local, anisotropic mesh refinement is developed in conjunction with the multi-dimensional fluctuation splitting scheme. The behavior of the new fluctuation splitting adaption strategy has a feature-alignment flavor, rather than feature-clustering like current methods, implying a reduction in the required number of nodes, and hence solution times, for non-linear problems.

4.1 Curvature Clustering

Current state-of-the-art for unstructured mesh adaption is based on *a posteriori* error estimation[61, 62, 65, 66, 67, 69, 71, 74, 75, 76, 97, 98, 99]. The error estimates can be derived either by looking at the leading truncation error terms in the spatial discretization or by considering the solution interpolation error. In practice, either approach reduces, for second-order-accurate spatial discretizations, to a check on the curvature of the solution. Nodal positioning is then driven to equate the scaled second derivatives along all edges. If isotropic cells are desired, the magnitude of the local curvature inversely dictates the element sizes, while for anisotropic adaption directional derivatives can be used to stretch elements.



Edge in bold *is not* candidate for swapping.

Edge in bold *is* candidate for swapping.

Edge has been swapped.

Figure 4.1: Edge swapping schematic.

Habashi[76] defines the *a posteriori* error estimate on an edge to vary like,

$$|E_r| \sim \ell^2 \left| \frac{\partial^2 U}{\partial r^2} \right| \tag{4.1}$$

or as "the edge length squared times the second derivative of the solution." In the finite volume context, one simple and efficient way to construct an edge-based error estimate along edge $\overline{01}$ is,

$$|E_r| \sim \left| \frac{\vec{\nabla} U_1 \cdot \frac{\vec{r}_{01}}{\ell_{01}} - \vec{\nabla} U_0 \cdot \frac{\vec{r}_{01}}{\ell_{01}}}{\ell_{01}} \right| \ell_{01}^2 = \left| \left(\vec{\nabla} U_1 - \vec{\nabla} U_0 \right) \cdot \vec{r}_{01} \right| \tag{4.2}$$

The edge-based finite volume scheme detailed in chapter 3 already performs the gradient computations, making the error estimate a trivial computation.

4.1.1 Adaption Mechanics

Edge swapping

One simple method to improve a mesh is to swap edges between nodes, altering the local connectivity. If the triangles to either side of an edge form a convex quadrilateral (Figure 4.1) then that edge is a candidate to be swapped. If the error estimate for the swapped connectivity is smaller than for the current edge connectivity, then the

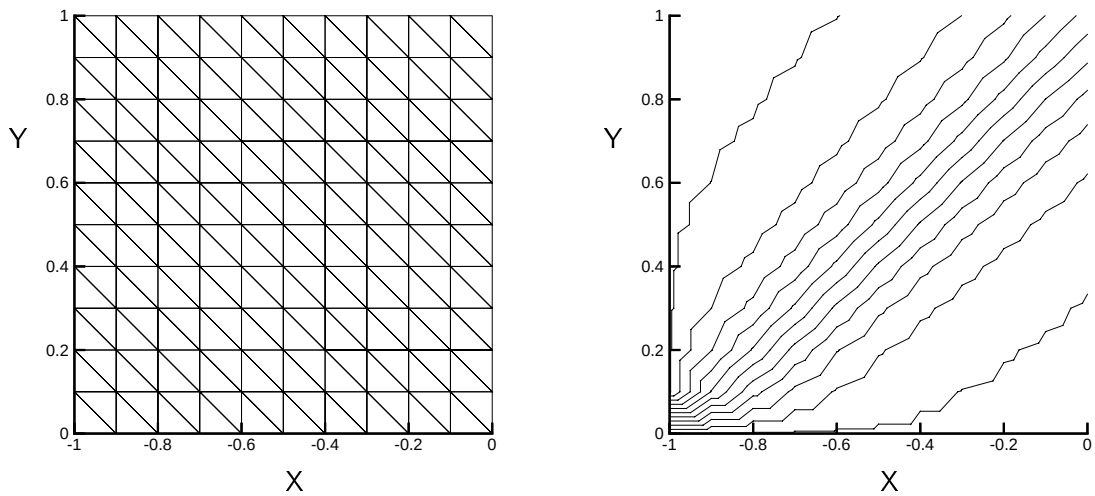


Figure 4.2: Initial, poorly aligned mesh and solution. Solution contours vary on (0,1) with 0.1 increments.

edge is swapped. In practice, an error threshold is employed to avoid limit cycles in smooth regions of the solution.

The edge swapping procedure is demonstrated in Figures 4.2–4.4. The problem statement is linear advection of a shear discontinuity with $\vec{\lambda} = (1, 1)$ and a uniform inflow on the left boundary $U(-1, y) = 0$ and uniform inflow on the bottom boundary $U(x, 0) = 1$. The DMFDSFV scheme is used and reconstruction is performed using the compressive Van Albada limiter. The initial mesh is poorly aligned, Figure 4.2(left), and the resulting solution shows significant contour spreading, Figure 4.2(right). Successive global edge swapping sweeps, Figures 4.3 and 4.4, improve the solution, reducing the contour spread.

Excessive dissipation is created in the bottom left inflow cell for this case, where the diagonal is not swapped with the present routine. Node insertion, to be discussed presently, will be able to handle cells such as these, where the edge swapping logic breaks down.

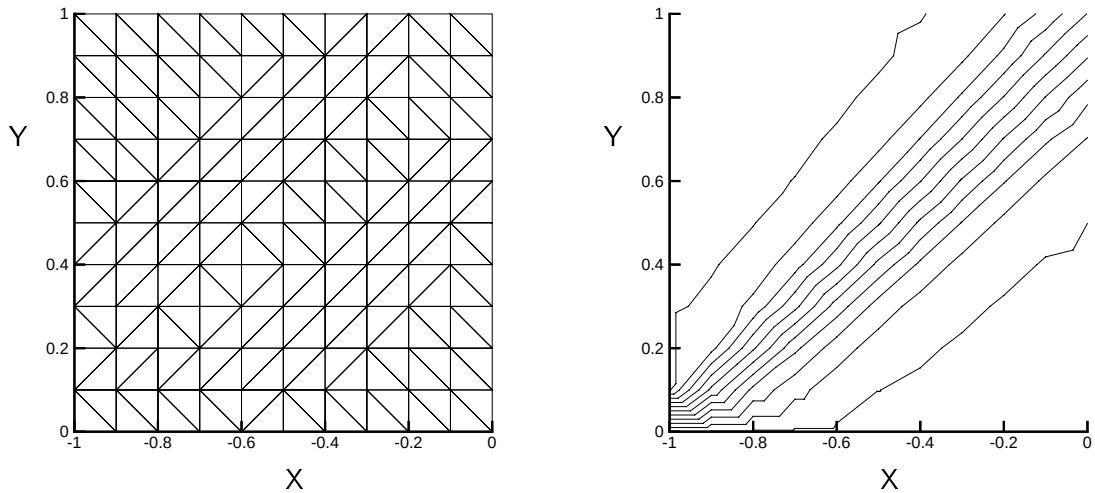


Figure 4.3: First edge swap, with improved solution. Solution contours vary on (0,1) with 0.1 increments.

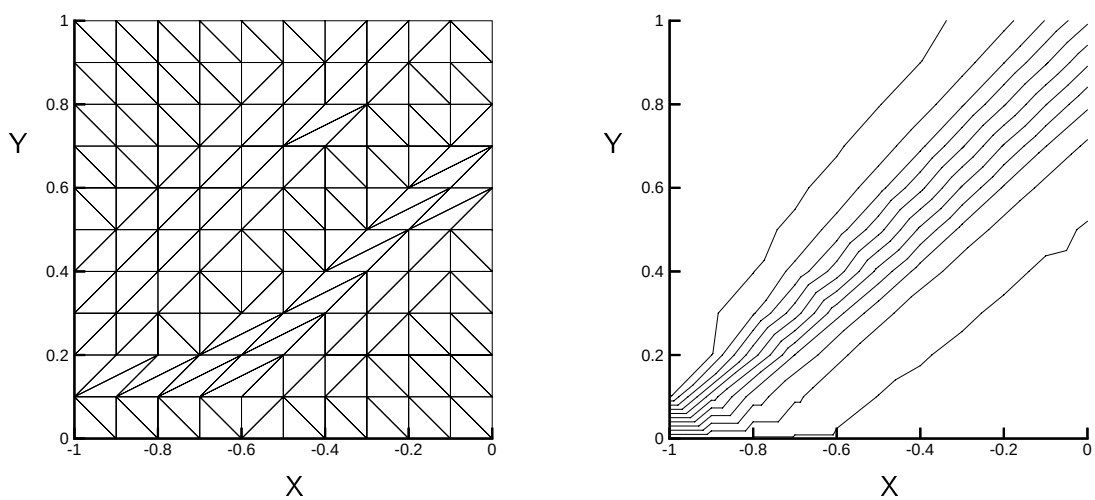


Figure 4.4: Second edge swap, with small solution improvement. Solution contours vary on (0,1) with 0.1 increments.

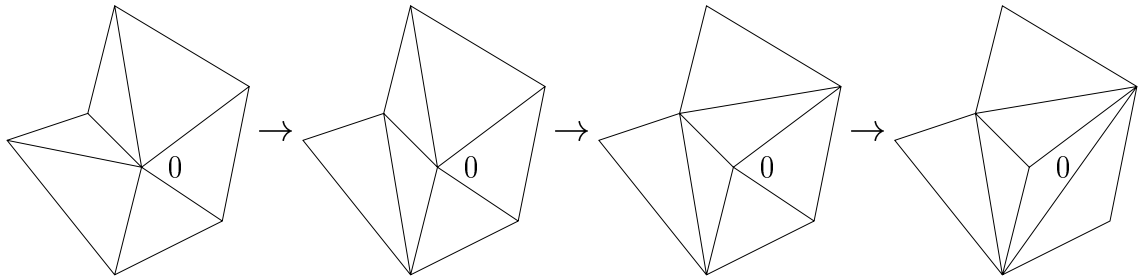


Figure 4.5: Reduction from 6 to 3 edges connected to node 0.

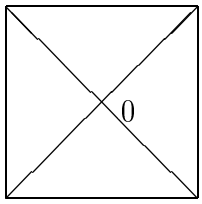


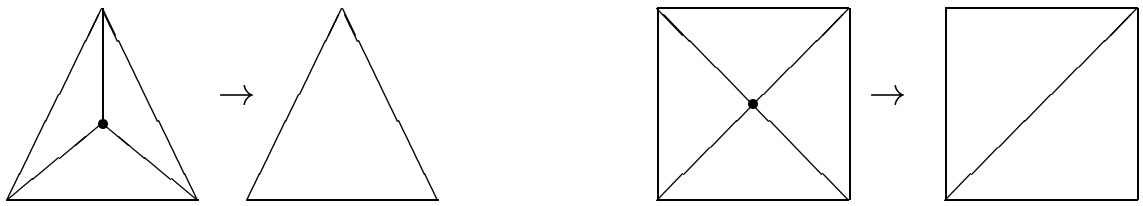
Figure 4.6: Four edges are the minimum that must be connected to node 0.

Point deletion

If all edges connected to a node have an error estimate, Eqn. 4.2, less than a threshold value, then that node can be removed. Once a node is flagged for deletion, the number of edges emanating from that node are reduced *via* edge swapping, as depicted in Figure 4.5. Usually, the number of edges connected at a node can be reduced to three, though Figure 4.6 shows a case where the minimum number of connected edges at the node is four.

The removal of one node also eliminates two triangular elements and three edges. A schematic of the actual removal process of an interior node is depicted in Figure 4.7 for both the three and four edge connectivity cases. The removal of a boundary node with three edges connected, depicted in Figure 4.8, eliminates one triangular element and only two edges.

The objective of point deletion is to minimize the number of grid points while retaining the same accuracy. In practice, bookkeeping of node/cell/edge numbering is the most difficult part of the point deletion process.



(a) Three edges connected. (b) Four edges connected.

Figure 4.7: Schematic of node deletion.

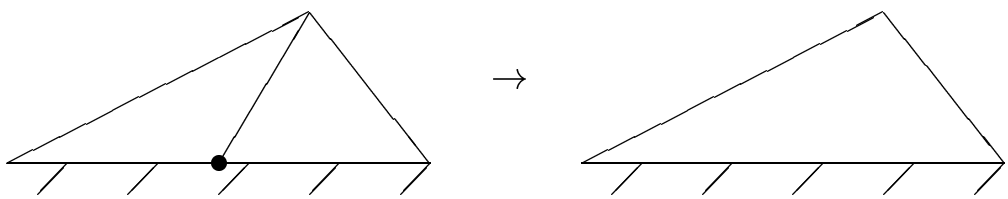


Figure 4.8: Schematic of boundary node deletion.

A numerical example of point deletion is presented for the 45° linear advection example. Figure 4.9 shows a grid and converged DMFDSFV solution to which the edge swapping routine has been applied. The point deletion routine is then applied, resulting in the grid of Figure 4.10. The solution on the reduced mesh remains unchanged from the finer-mesh solution.

Point insertion

Point insertion can be considered to be the opposite procedure of the four-edge connectivity point deletion, shown in Figure 4.7(b). If the error estimate along an edge exceeds a given threshold, then that edge is flagged for refinement. A new node is added at the midpoint of the edge, two new cells are defined, and three new edges are created. The aim of point insertion is to increase solution accuracy, albeit at an increased computational cost due to the additional mesh nodes. Use of the local error estimator allows for selective refinement and results in fewer nodes than would be

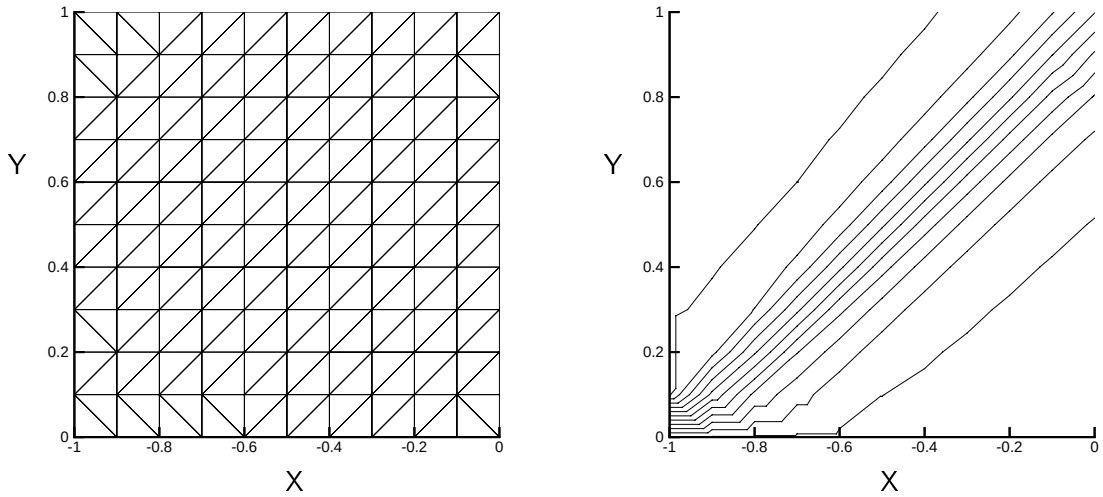


Figure 4.9: Before point deletion. Solution contours vary on (0,1) with 0.1 increments.

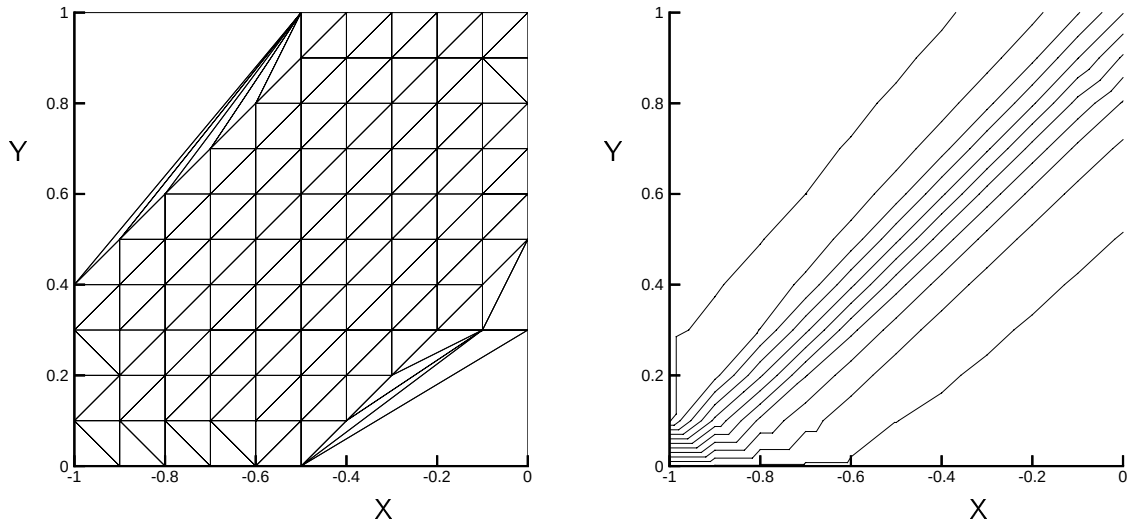


Figure 4.10: After point deletion. Solution contours vary on (0,1) with 0.1 increments.

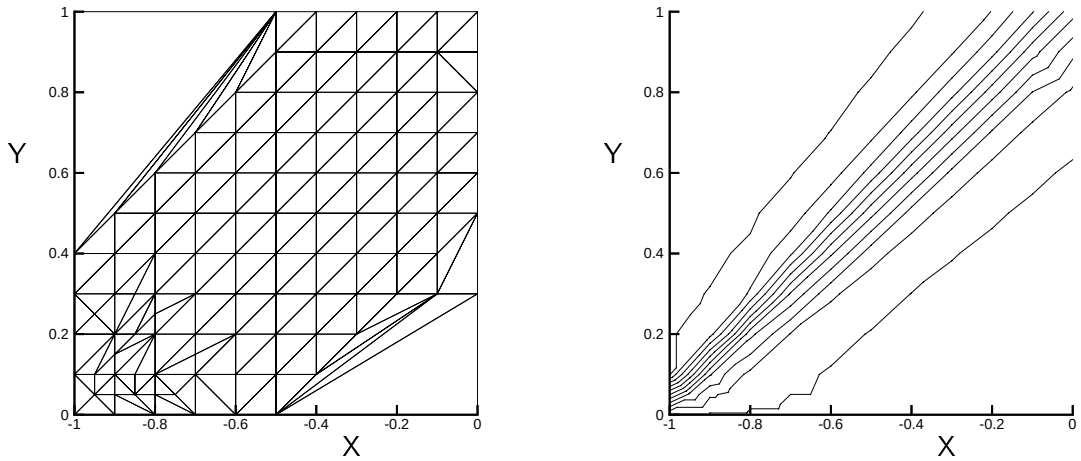


Figure 4.11: After point insertion. Solution contours vary on (0,1) with 0.1 increments.

produced by a global refinement.

The familiar numerical example of this section is continued by applying the point insertion algorithm to the solution and mesh last shown in Figure 4.10. The locally refined mesh, with most of the refinement occurring near the inflow boundary where the greatest artificial diffusion is produced, is shown in Figure 4.11, where 13 new nodes have been introduced. The solution shows improved accuracy, particularly in the reduced spreading in the lower left corner.

Nodal displacement

The fourth local adaption technique considered is nodal displacement. Neither the number of nodes nor their connectivity are altered, but their spatial location is optimized. The nodal displacement concept mimics the Gauss collocation quadrature points familiar in calculus, and hence seeks to improve solution accuracy without increasing computational cost.

The approach taken here uses the spring analogy, similar to the techniques presented by Gnoffo[57] and Ait-Ali-Yahia[70] *et al.* The springs are taken to be the mesh edges. The spring constants are the edge error estimates. A minimization is

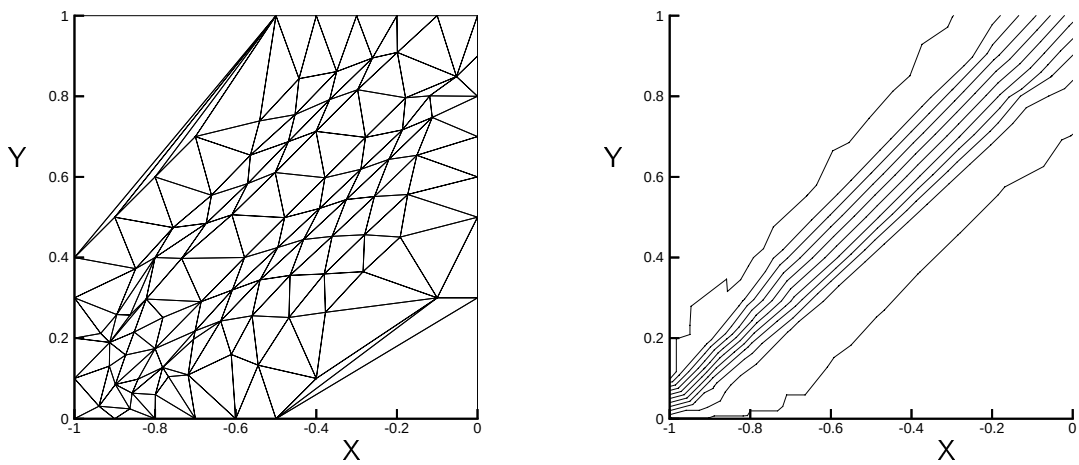


Figure 4.12: After nodal displacements. Solution contours vary on (0,1) with 0.1 increments.

then sought for a potential energy formed as,

$$\sum_j \frac{1}{2} \, |E_r|_j \, \vec{r}_{0j}^{\,2} \qquad j \text{ is all distance 1 neighbors to node 0} \tag{4.3}$$

The minimum is obtained by setting the derivative to zero, giving the requirement,

$$\sum_j |E_r|_j \, \vec{r}_{0'j} \; = 0 \tag{4.4}$$

where $0'$ indicates the new, minimum energy position of node 0. Holding the edge error estimate constant during the displacement, an updated position vector can be obtained,

$$\sum_j |E_r|_j \, (\vec{r}_{0j} + \vec{r}_{0'0}) \; = 0 \tag{4.5}$$

$$\vec{r}_{00'} \; = -\vec{r}_{0'0} = \frac{\sum_j |E_r|_j \, \vec{r}_{0j}}{\sum_j |E_r|_j} \tag{4.6}$$

Equation 4.6 gives the position vector pointing from the current location of node 0 to its adapted location. This adapted location is an attempt to optimally equate the scaled error estimates over all edges connected to the current node. Since the error

estimates are held constant during the local adaption step, an iterative process is required to obtain the optimal nodal positions.

The nodal displacement routine is applied to the results in Figure 4.11, completing one global iteration of the full mesh adaption suite for unstructured meshes, consisting of nodal displacements, insertions, and deletions and edge swapping. Solution improvement, shown in the tighter contour clustering in Figure 4.12, particularly toward the upper right, is demonstrated using the point movement technique.

4.1.2 Adaption Procedure

Following the recommendations of Dompierre[71] *et al.* and Habashi[76] *et al.*, the nodal displacement technique is used as a smoother between applications of the other three adaption operations. Also, close coupling between the solver and the mesh adaptor is to be maintained.

The overall solution strategy therefore follows:

1. Solve on initial mesh.
2. Swap diagonals and iterate solver.
3. Move nodes and iterate solver.
4. Insert nodes and iterate solver.
5. Move nodes and iterate solver.
6. Delete nodes and iterate solver.
7. Swap diagonals and iterate solver.
8. Move nodes and iterate solver.

Steps 4–8 are repeated until convergence of the entire process.

The entire solution-adaptive procedure is applied to the 45° shear problem using the DMFDSFV solver with the Van Albada limiter. The initial grid, containing

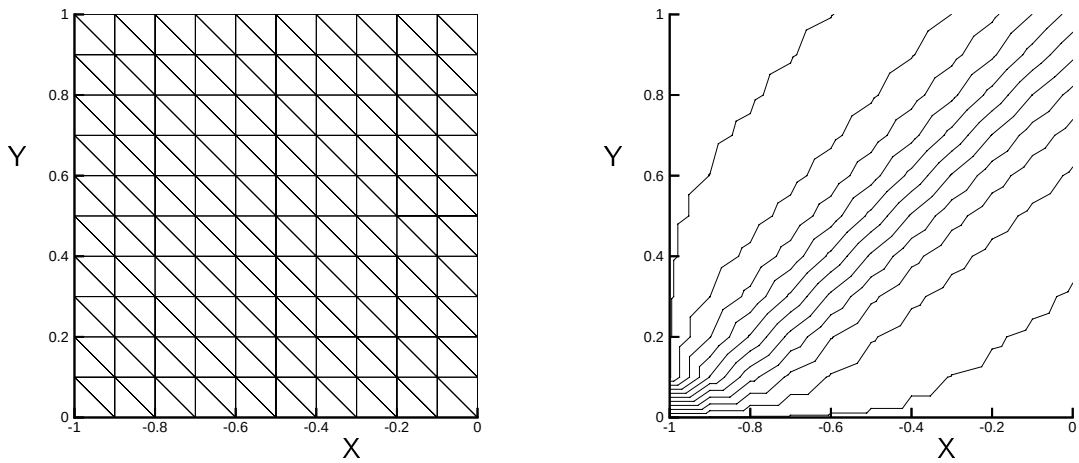


Figure 4.13: Starting mesh and converged solution, $\vec{\lambda} = (1, 1)$. Solution contours vary on (0,1) with 0.1 increments.

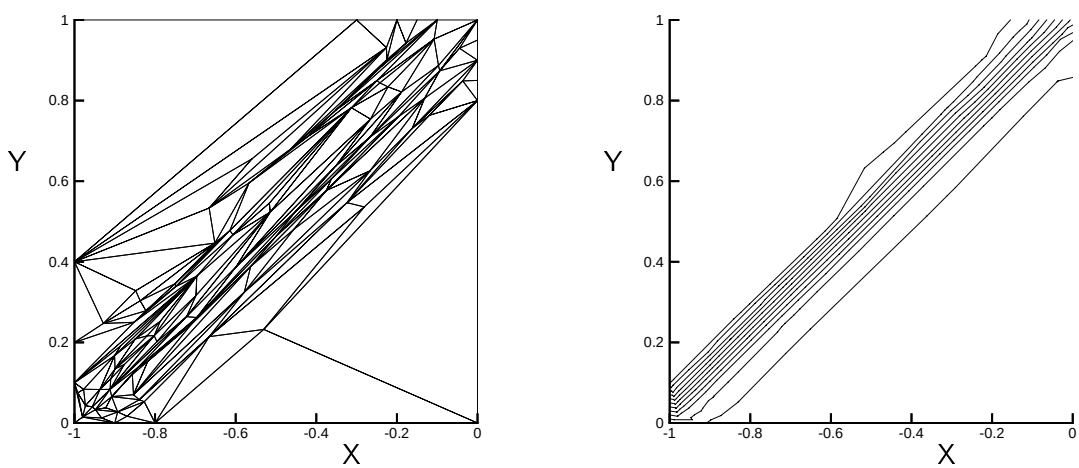


Figure 4.14: Adapted mesh with greatly improved solution. Solution contours vary on (0,1) with 0.1 increments.

121 nodes, and the solution on this mesh are shown in Figure 4.13. After only four complete adaption cycles, the mesh size has been reduced to just 103 nodes while capturing a very accurate solution, as seen in Figure 4.14. While the mesh adaption is highly anisotropic, the solution remains well behaved.

A demonstration of the method is also performed for a case with a variable advection velocity, namely the circular advection problem. The inflow profile contains a linear ramp between $U = 0$ on $x \leq -0.7$ and $U = 1$ on $x \geq -0.6$. The initial mesh is the same as for the shear case, containing 121 nodes. The solution on this mesh is excessively smeared, Figure 4.15. Three mesh adaption cycles were performed, resulting in a final mesh having 146 nodes. A significant improvement is again seen in the resulting solution, Figure 4.16.

4.2 Characteristic Alignment

4.2.1 Exact Advection Solution

While anisotropic adaption to reduce *a posteriori* error estimates represents current state-of-the-art for unstructured-grid finite volume algorithms, it was shown in chapter 3 that there exists a better scheme for multi-dimensional advection than DMFDS-FV, namely fluctuation splitting. The question arises as to whether an edge-based error estimate, which is by the definition of an edge a locally one-dimensional indicator, is an appropriate adaption criteria for a truly multi-dimensional scheme. This section explores the development of candidate multi-dimensional adaption criteria that fit within the framework of the four local anisotropic adaption operations—edge swapping, point insertion, point deletion, and nodal displacement.

As a starting point, posit that an optimal mesh is one with the least number of nodes on which a positive scheme can obtain a solution free from artificial dissipation. The artificial dissipation terms for fluctuation splitting, ϕ'^{ξ} and ϕ'^{η} in Eqn. 3.29, were seen to scale with the limited fluctuations, $\phi^{*\xi}$ and $\phi^{*\eta}$ in Eqns. 3.26 and 3.27, respectively. For a locally positive scheme the limited fluctuations are bounded by the total cell fluctuation. Thus, driving the cell fluctuation to zero will eliminate

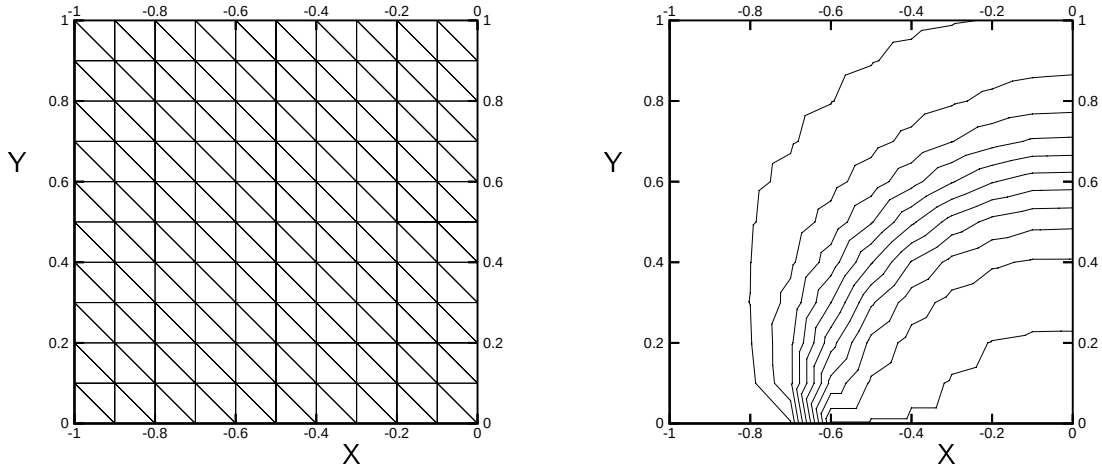


Figure 4.15: Starting mesh and converged solution, $\vec{\lambda} = (y, -x)$. Solution contours vary on (0,1) with 0.1 increments.

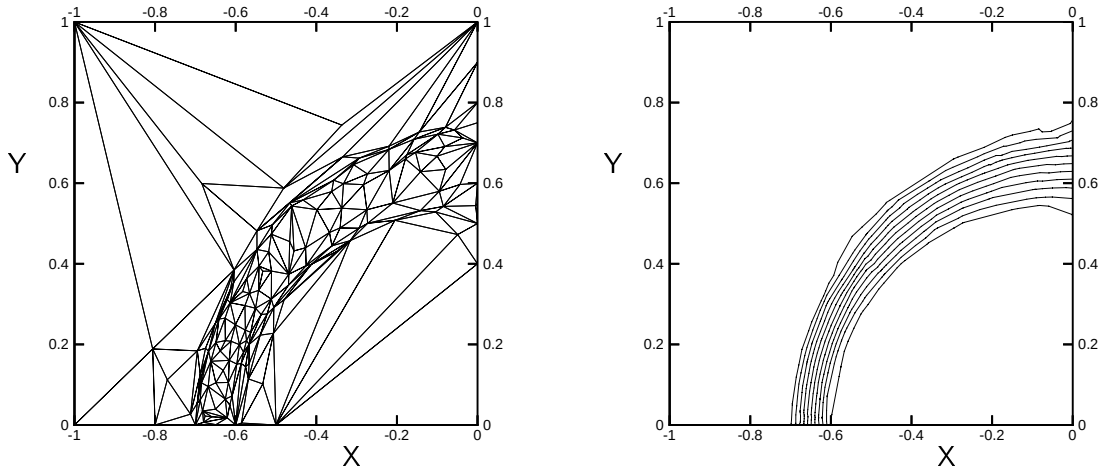


Figure 4.16: Converged DMFDSFV solution and mesh after three mesh adaption cycles, $\vec{\lambda} = (y, -x)$. Solution contours vary on (0,1) with 0.1 increments.

artificial dissipation production, and *vice versa*.

Combining Eqns. 3.22, 3.24, and 3.25, the cell fluctuation is,

$$\phi = -\frac{\ell_1}{2}\hat{n}_1 \cdot \tilde{\vec{A}}(U_2 - U_1) + \frac{\ell_3}{2}\hat{n}_3 \cdot \tilde{\vec{A}}(U_3 - U_2) \tag{4.7}$$

or,

$$\phi = \frac{1}{2}\left[\ell_1\hat{n}_1 U_1 - (\ell_1\hat{n}_1 + \ell_3\hat{n}_3)U_2 + \ell_3\hat{n}_3 U_3\right] \cdot \tilde{\vec{A}} \tag{4.8}$$

Noting the vector identity $\sum_{i=1}^{3} \hat{n}_i \ell_i = 0$, the fluctuation can be expressed in compact notation,

$$\phi = \frac{1}{2}\sum_{i=1}^{3} U_i \ell_i \hat{n}_i \cdot \tilde{\vec{A}} \tag{4.9}$$

Recall from the notation for an element's geometry, Figure 3.3, that edge i is the edge on the opposite side of the triangle from node i. If the cell-averaged flux Jacobian, $\tilde{\vec{A}}$, is parallel to side i, then $\hat{n}_i \cdot \tilde{\vec{A}} = 0$. In the context of mesh adaption, assume that an edge, say without loss of generality edge 3, is aligned to be parallel to $\tilde{\vec{A}}$, then from Eqn. 4.7 it is clear that the fluctuation, and hence the artificial dissipation, goes to zero as $U_2 \to U_1$. Note that this condition for a vanishing fluctuation holds true irrespective of the value U_3.

The known solution to steady advection is that the solution remains invariant along the characteristics. A coupled solution-adaptive strategy is apparent—the adaption aligns one edge of each cell with the discrete counterpart to the characteristic, namely $\tilde{\vec{A}}$, and the fluctuation splitting scheme converges to an exact advection solution by equating each downstream node of an aligned edge with the value of the upstream node. Efforts along this tack follow, though not without some lingering questions. What of the non-linear problem, where characteristics vanish at a shock? How can systems of equations be handled with multiple or imaginary characteristics defined on each cell? What about combined advection-diffusion problems where the exact solution no longer remains invariant along characteristics?

By way of demonstration that in fact fluctuation splitting does compute the exact solution to multi-dimensional linear advection problems when the grid is aligned such that each triangle with non-zero gradient has one edge parallel to the cell-average advection velocity, the two cases employed with the curvature clustering adaption are

considered. The 45° shear case can be captured using a mesh with only 6 nodes, none of which are on the interior. The optimal mesh and fluctuation splitting solution are shown in Figure 4.17. Contrast these results with the mesh-adapted DMFDSFV solution to this problem in Figure 4.14. Fluctuation splitting in this case produces a more accurate, diffusion-free solution on a mesh 17 times smaller. A DMFDSFV solution is performed on this optimal fluctuation splitting grid, yielding the results of Figure 4.18, where 30 percent of the shear discontinuity has been smeared by the outflow boundary.

For the circular advection problem a mesh is constructed containing 10 nodes, an order of magnitude fewer than were in the adapted DMFDSFV results of Figure 4.16. The fluctuation splitting mesh and solution are shown in Figure 4.19, where the inflow profile is exactly mapped to the outflow boundary. The mesh is called 'optimal' for this case because it contains the fewest points required to convect a diffusion-free solution. However, the solution representation on the interior of the domain is exact only in a discrete sense. Note that while the DMFDSFV solutions used the compressive Van Albada limiter, the fluctuation splitting solutions were performed with the non-compressive Minmod limiter. The DMFDSFV solution on the optimal fluctuation splitting grid is shown in Figure 4.20, where it is clear that significantly more nodes will be required to achieve reasonable accuracy.

These test cases suggest a tremendous potential for highly accurate fluctuation splitting solutions on extremely coarse meshes, when properly aligned.

4.2.2 Adaption Strategies for Fluctuation Splitting

The fundamental strategy for optimizing a mesh for fluctuation splitting solutions to linear-advection problems is to align one edge of each triangle with the cell-average characteristic direction. Translating this requirement into a systematic logic for local adaption is not straight-forward, particularly for a variable advection velocity when one edge cannot simultaneously be aligned with the characteristics in the elements to both the right and left of the edge.

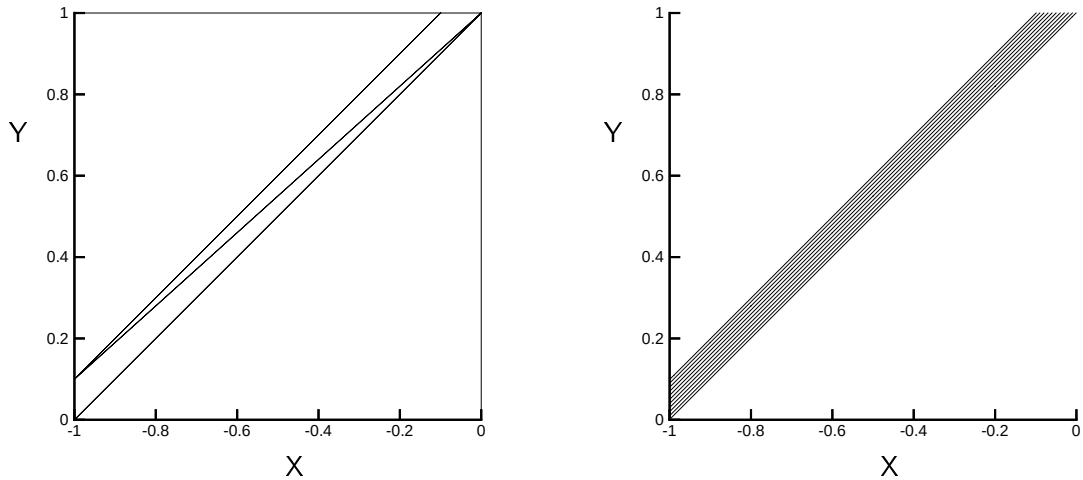


Figure 4.17: Optimal mesh and diffusion-free solution using fluctuation splitting, $\vec{\lambda} = (1, 1)$. Solution contours vary on (0,1) with 0.1 increments.

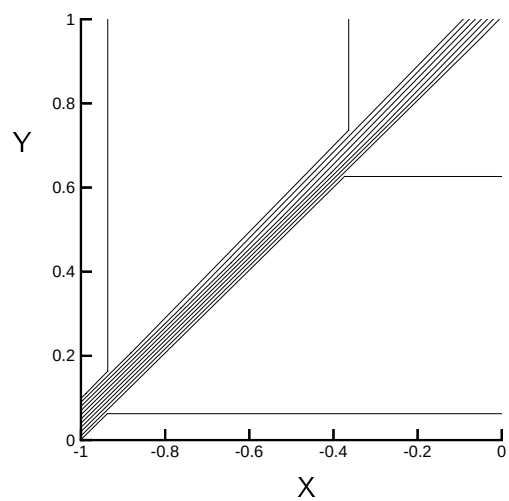


Figure 4.18: DMFDSFV solution on optimal fluctuation splitting mesh, $\vec{\lambda} = (1, 1)$. Solution contours vary on (0,1) with 0.1 increments.

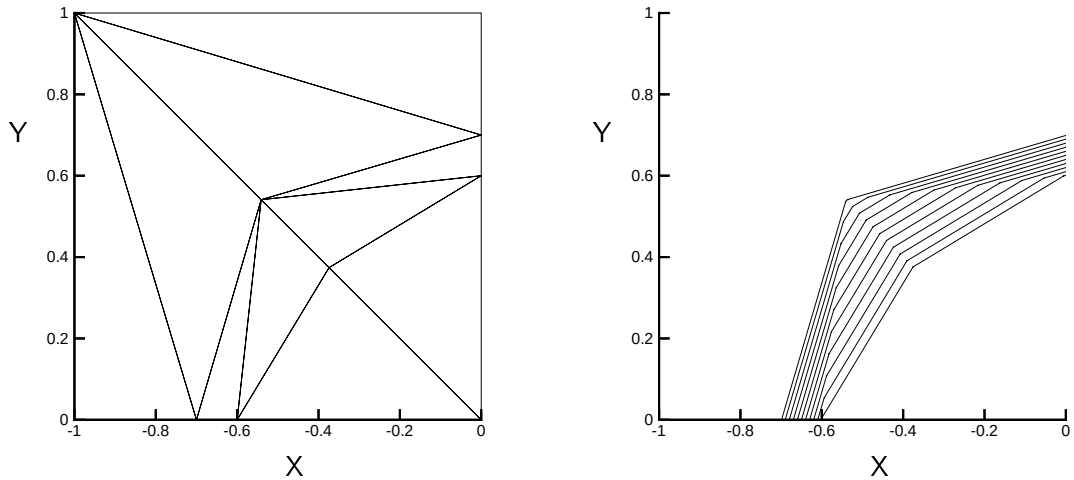


Figure 4.19: Fluctuation splitting optimal grid and diffusion-free solution, $\vec{\lambda} = (y, -x)$. Solution contours vary on (0,1) with 0.1 increments.

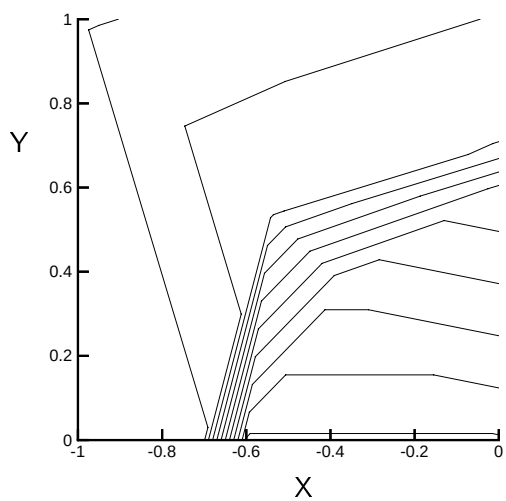
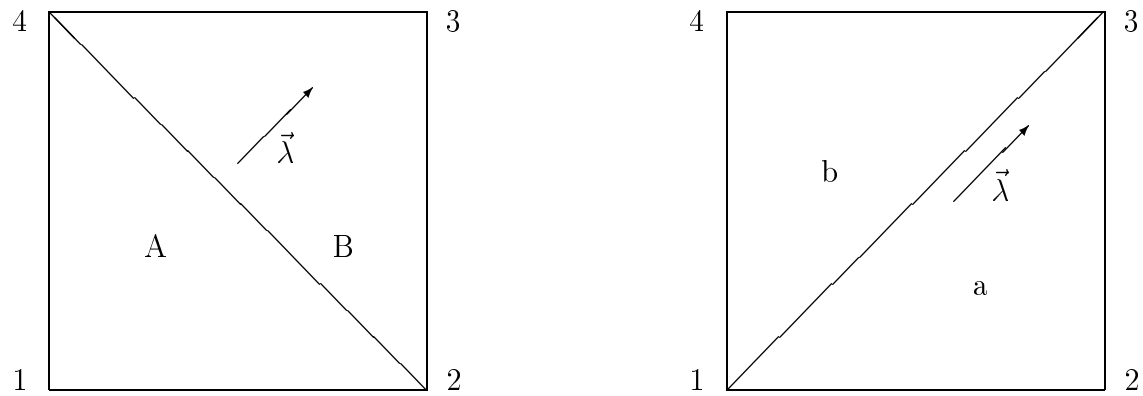


Figure 4.20: DMFDSFV solution on optimal fluctuation splitting mesh, $\vec{\lambda} = (y, -x)$. Solution contours vary on (0,1) with 0.1 increments.



(a) Misaligned mesh.

(b) Aligned mesh.

Figure 4.21: Edge swap options for linear advection with fluctuation splitting.

Edge swapping

Consider the 45° shear advection on a unit square, Figure 4.21. For this problem $\tilde{\vec{A}} = \vec{\lambda} = (1, 1)$ for all cells. The choice of diagonal perpendicular to $\vec{\lambda}$, Figure 4.21(a), constitutes a poorly aligned grid. The other choice of diagonal, Figure 4.21(b), creates a perfectly aligned grid. For the misaligned grid the cell fluctuations can be computed using Eqn. 4.9,

$$\phi_A = \frac{1}{2}(2U_1 - U_2 - U_4) \qquad \phi_B = \frac{1}{2}(U_2 + U_4 - 2U_3) \tag{4.10}$$

and their sum is $\phi_A + \phi_B = U_1 - U_3$.

For the grid-aligned case, the fluctuations are,

$$\phi_a = \frac{1}{2}(U_1 - U_3) = \phi_b = \frac{\phi_A + \phi_B}{2} \tag{4.11}$$

which leads to the observation,

$$\phi_A + \phi_B = \phi_a + \phi_b = 2\phi_a \tag{4.12}$$

Clearly, swapping the diagonal does not in itself reduce the total fluctuation. Yet it is known that fluctuation splitting will obtain the exact solution on the aligned grid and a diffused solution on the misaligned grid (recall the results of section 3.2.4).

Still seeking a link between mesh alignment and fluctuations, consider the RMS of the fluctuations. It is postulated that the RMS of the fluctuations is less on the aligned mesh than on the misaligned mesh. The proof proceeds from Eqn. 4.12,

$$\phi_A + \phi_B = 2\phi_a \tag{4.13}$$

$$\phi_A^2 + 2\phi_A\phi_B + \phi_B^2 = 4\phi_a^2 \tag{4.14}$$

$$\phi_A^2 + \phi_B^2 = 4\phi_a^2 - 2\phi_A\phi_B \tag{4.15}$$

$$= \frac{1}{2}(4\phi_a^2 - 4\phi_A\phi_B) + 2\phi_a^2 \tag{4.16}$$

$$= \frac{1}{2}\left[4\left(\frac{\phi_A + \phi_B}{2}\right)^2 - 4\phi_A\phi_B\right] + 2\phi_a^2 \tag{4.17}$$

$$\phi_A^2 + \phi_B^2 = \frac{1}{2}(\phi_A^2 - 2\phi_A\phi_B + \phi_B^2) + 2\phi_a^2 \tag{4.18}$$

$$= \frac{1}{2}(\phi_A - \phi_B)^2 + 2\phi_a^2 \tag{4.19}$$

$$> 2\phi_a^2 = \phi_a^2 + \phi_b^2 \tag{4.20}$$

The use of the RMS of fluctuations to guide adaption for scalar problems has been proposed by Roe[32, 77]. The fluctuation splitting solver Roe employs is of the non-positive type, and for general solutions suffers from extreme dispersion errors. Also, nodal displacements are the only adaption operation demonstrated.

A demonstration of the present edge swapping is performed for the 45° shear problem. The initial grid and fluctuation splitting solution are shown in Figure 4.22. The fluctuation splitting solution on this mesh is similar to the one obtained using DMFDSFV in Figure 4.13. However, after just a single edge swapping sweep the exact solution is obtained by fluctuation splitting, Figure 4.23. DMFDSFV was not able to match this accuracy from fluctuation splitting even on the highly adapted mesh from Figure 4.14.

Point deletion

Point deletion for a fluctuation splitting grid mimics the procedure previously discussed in the finite volume context. If all elements connected at a given node have

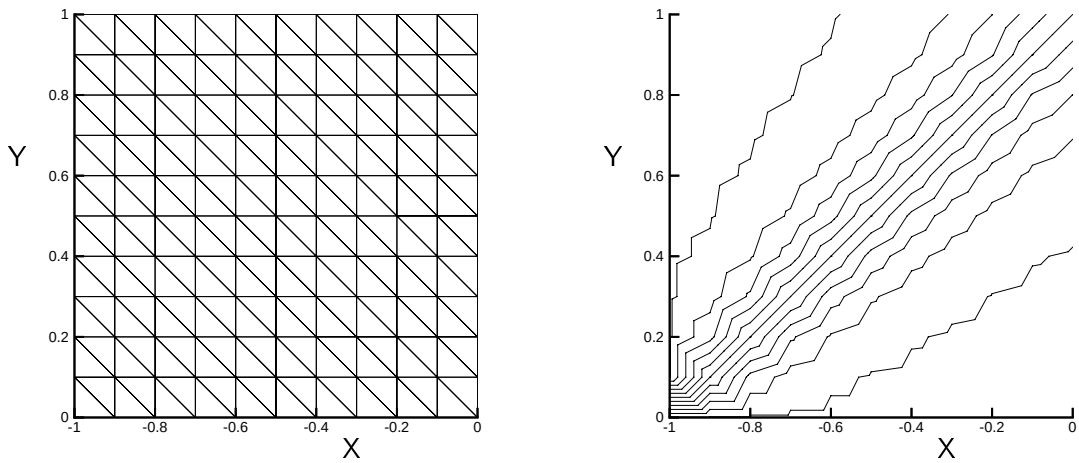


Figure 4.22: Starting mesh and converged fluctuation splitting solution, $\vec{\lambda} = (1, 1)$. Solution contours vary on (0,1) with 0.1 increments.

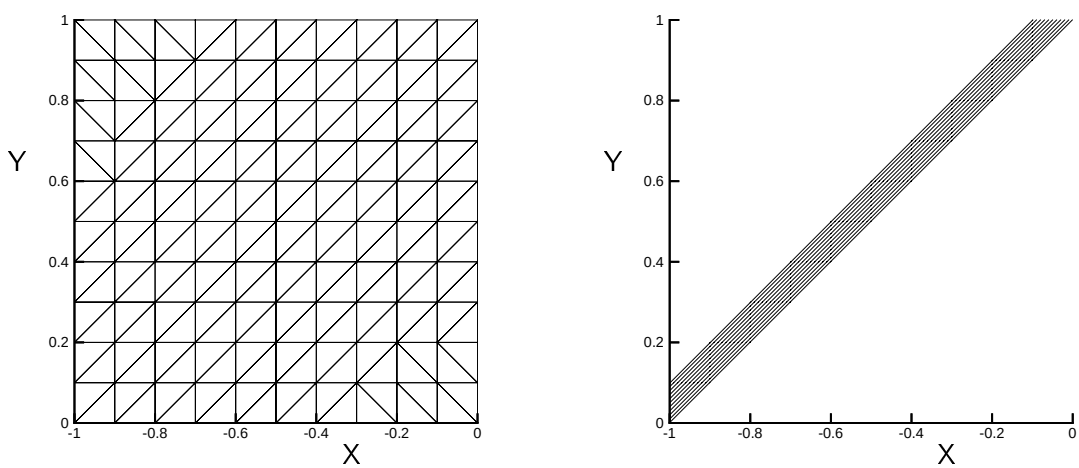


Figure 4.23: Exact solution obtained by fluctuation splitting after one edge swap cycle, $\vec{\lambda} = (1, 1)$. Solution contours vary on (0,1) with 0.1 increments.

small or vanishing fluctuations, then that node is a candidate for deletion. Edge swapping is performed to reduce the number of edges, and hence cells, connected to the node, under the restraint that the RMS fluctuations of the swapped configurations must remain small. If the number of cells connected at the node can be reduced to three, then the node can be deleted if the fluctuation remains small in the triangle formed by the agglomeration of the three connected elements. If four elements remain connected at the node, the point may be deleted if the RMS fluctuations remain small in either of the two possible agglomerations from four triangles to two triangles.

When the point deletion algorithm is applied to the advected shear results seen in the edge swapping demonstration, Figure 4.23, the minimum optimal grid and associated exact solution, presented in Figure 4.17, are obtained after a single loop over the domain.

Nodal displacement

A scheme for displacing nodal coordinates to minimize RMS of fluctuations about the node has been presented by Roe[32, 77]. The development presented by Roe uses the same minimization scheme to evolve the solution as is used to drive the nodal positioning. That type of fluctuation splitting solver has a central difference flavor and is not monotonic. The present procedure incorporates the upwind, non-linear fluctuation splitting algorithm detailed in section 3.2.1 with some of the mesh movement strategies of Roe. The development also bears some resemblance to the adjoint equation adaption schemes of Anderson[98] and Venditti[99]. However, the adjoint schemes seek to evaluate changes in global parameters with respect to an error estimate computed on a coarsened mesh for a finite volume solver, while the present technique is entirely local and does not employ a coarsened sub-mesh and is specifically for a fluctuation splitting discretization.

At a given node, the nodal displacements are computed as a first step and then the solution is updated at the new nodal location *via* a local point-implicit inversion. In this manner a global mesh movement sweep can be accomplished in conjunction with a single Gauss-Seidel iteration of the solver. In this section the current node to be moved is globally numbered node i. Within each triangular element the nodes are

locally numbered 1–3. Derivatives in y are omitted when they exactly follow from the derivatives in x.

The functional to be minimized is defined at the node as a sum of contributions from all cells surrounding the node (equivalent to Eqn. 13, p. 248 in [77]),

$$\Upsilon_i = \frac{1}{2}\sum_{\forall \mathsf{T} \ni i} \phi_{\mathsf{T}} \Xi_{\mathsf{T}} \phi_{\mathsf{T}} = \frac{1}{2}\sum_{\mathsf{T}} \Xi_{\mathsf{T}} \phi_{\mathsf{T}}^2 \tag{4.21}$$

The weighting factor, Ξ_{T}, is a positive scalar, generalizing to a symmetric, positive definite matrix for systems. The functional is thus a sum of positive semi-definite contributions from triangles containing the current node.

The derivative of Υ with respect to a nodal coordinate is,

$$\frac{\partial \Upsilon_i}{\partial x_i} = \sum_{\mathsf{T}} \left(\frac{\phi_{\mathsf{T}}^2}{2} \frac{\partial \Xi_{\mathsf{T}}}{\partial x_i} + \Xi_{\mathsf{T}} \phi_{\mathsf{T}} \frac{\partial \phi_{\mathsf{T}}}{\partial x_i} \right) \tag{4.22}$$

Note that the derivatives in Eqn. 4.22 represent the change in solution values as the discrete mesh is perturbed, and as such are to be interpreted in the context of variational calculus, and not as spatial gradients according to the more-familiar multi-variable calculus.

The minimization of Υ can be performed using a fixed-point iteration to force the derivatives to zero,

$$x_i^{n+1} = x_i^n - \frac{\partial \Upsilon}{\partial x_i} \tag{4.23}$$

$$\Delta^n x_i = -\frac{\partial \Upsilon}{\partial x_i} \tag{4.24}$$

Equation 4.24 can be combined with Eqn. 4.22 in the form of a distribution method of steepest descent,

$$\Delta^n x_i \leftarrow -\frac{\phi_{\mathsf{T}}^2}{2} \frac{\partial \Xi_{\mathsf{T}}}{\partial x_i} - \Xi_{\mathsf{T}} \phi_{\mathsf{T}} \frac{\partial \phi_{\mathsf{T}}}{\partial x_i} \tag{4.25}$$

Convergence can be enhanced over the fixed-point iteration by using a Newton scheme. Expanding the gradient in an approximate Taylor series,

$$0 = \left.\frac{\partial \Upsilon}{\partial x_i}\right|^{n+1} \simeq \left.\frac{\partial \Upsilon}{\partial x_i}\right|^{n} + \Delta^n x_i \frac{\partial}{\partial x_i} \left.\frac{\partial \Upsilon}{\partial x_i}\right|^{n} + \Delta^n y_i \frac{\partial}{\partial y_i} \left.\frac{\partial \Upsilon}{\partial x_i}\right|^{n} \tag{4.26}$$

$$0 = \left.\frac{\partial \Upsilon}{\partial y_i}\right|^{n+1} \simeq \left.\frac{\partial \Upsilon}{\partial y_i}\right|^{n} + \Delta^n x_i \frac{\partial}{\partial x_i} \left.\frac{\partial \Upsilon}{\partial y_i}\right|^{n} + \Delta^n y_i \frac{\partial}{\partial y_i} \left.\frac{\partial \Upsilon}{\partial y_i}\right|^{n} \tag{4.27}$$

leading to the form,

$$\Delta^n \begin{Bmatrix} x_i \\ y_i \end{Bmatrix} = - \begin{bmatrix} \frac{\partial^2 \Upsilon}{\partial x_i^2} & \frac{\partial^2 \Upsilon}{\partial x_i \partial y_i} \\ \frac{\partial^2 \Upsilon}{\partial x_i \partial y_i} & \frac{\partial^2 \Upsilon}{\partial y_i^2} \end{bmatrix}^{-1} \begin{Bmatrix} \frac{\partial \Upsilon}{\partial x_i} \\ \frac{\partial \Upsilon}{\partial y_i} \end{Bmatrix} \tag{4.28}$$

In [32], Roe suggests neglecting the off-diagonal terms in Eqn. 4.28.

Second derivatives of the objective function are,

$$\frac{\partial^2 \Upsilon}{\partial x_i^2} = \sum_{\mathsf{T}} \left[\frac{\phi_{\mathsf{T}}^2}{2} \frac{\partial^2 \Xi_{\mathsf{T}}}{\partial x_i^2} + 2\phi_{\mathsf{T}} \Xi_{\mathsf{T}} \frac{\partial \Xi_{\mathsf{T}}}{\partial x_i} \frac{\partial \phi_{\mathsf{T}}}{\partial x_i} + \Xi_{\mathsf{T}} \left(\frac{\partial \phi_{\mathsf{T}}}{\partial x_i} \right)^2 + \Xi_{\mathsf{T}} \phi_{\mathsf{T}} \frac{\partial^2 \phi_{\mathsf{T}}}{\partial x_i^2} \right] \tag{4.29}$$

$$\begin{aligned} \frac{\partial^2 \Upsilon}{\partial x_i \partial y_i} = \sum_{\mathsf{T}} \Bigg[& \phi_{\mathsf{T}} \left(\frac{\partial \phi_{\mathsf{T}}}{\partial x_i} \frac{\partial \Xi_{\mathsf{T}}}{\partial y_i} + \frac{\partial \Xi_{\mathsf{T}}}{\partial x_i} \frac{\partial \phi_{\mathsf{T}}}{\partial y_i} \right) + \frac{\phi_{\mathsf{T}}^2}{2} \frac{\partial^2 \Xi_{\mathsf{T}}}{\partial x_i \partial y_i} + \Xi_{\mathsf{T}} \frac{\partial \phi_{\mathsf{T}}}{\partial x_i} \frac{\partial \phi_{\mathsf{T}}}{\partial y_i} \\ & + \phi_{\mathsf{T}} \Xi_{\mathsf{T}} \frac{\partial^2 \phi_{\mathsf{T}}}{\partial x_i \partial y_i} \Bigg] \end{aligned} \tag{4.30}$$

The fluctuation can be manipulated from Eqn. 4.9 as,

$$\begin{aligned} \phi &= \frac{1}{2} \left[U_1(y_3 - y_2,\, x_2 - x_3) + U_2(y_1 - y_3,\, x_3 - x_1) + U_3(y_2 - y_1,\, x_1 - x_2) \right] \cdot \tilde{\vec{A}} \\ &= \frac{1}{2} \left[(U_2 - U_3)(y_1,\, -x_1) + (U_3 - U_1)(y_2,\, -x_2) + (U_1 - U_2)(y_3,\, -x_3) \right] \cdot \tilde{\vec{A}} \end{aligned} \tag{4.31}$$

The spatial derivatives take the form,

$$\frac{\partial \phi}{\partial x_2} = \frac{1}{2} \left[(U_1 - U_3) \tilde{A}^{\mathsf{y}} + \sum_{j=1}^{3} U_j \ell_j \hat{n}_j \cdot \frac{\partial \tilde{\vec{A}}}{\partial x_2} + \sum_{j=1}^{3} \ell_j \hat{n}_j \cdot \tilde{\vec{A}} \frac{\partial U_j}{\partial x_2} \right] \tag{4.32}$$

$$\frac{\partial \phi}{\partial y_2} = \frac{1}{2} \left[(U_3 - U_1) \tilde{A}^{\mathsf{x}} + \sum_{j=1}^{3} U_j \ell_j \hat{n}_j \cdot \frac{\partial \tilde{\vec{A}}}{\partial y_2} + \sum_{j=1}^{3} \ell_j \hat{n}_j \cdot \tilde{\vec{A}} \frac{\partial U_j}{\partial y_2} \right] \tag{4.33}$$

The derivatives of the cell-averaged flux Jacobian will depend upon the particular flux function. However, since $\tilde{\vec{A}}$ is a weighted average of three nodal values,

$$\frac{\partial \tilde{\vec{A}}}{\partial x_2} \sim \frac{1}{3} \frac{\partial \vec{A}_2}{\partial x_2} \tag{4.34}$$

For uniform advection, $\frac{\partial \tilde{\vec{A}}}{\partial x_2} = 0$. For circular advection, $\frac{\partial \tilde{\vec{A}}}{\partial x_2} = (0, -\frac{1}{3})$ and $\frac{\partial \tilde{\vec{A}}}{\partial y_2} = (\frac{1}{3}, 0)$. For non-linear problems, in general,

$$\frac{\partial \tilde{\vec{A}}}{\partial x_2} \sim \frac{1}{3} \frac{\partial U_2}{\partial x_2} \tag{4.35}$$

In keeping with the Gauss-Seidel update philosophy, only the derivative of the solution value at the current node is retained in the last term of Eqns. 4.32 and 4.33. That is, if the current node is designated node 2 of the triangle under consideration, then $\frac{\partial U_2}{\partial x_2} = \frac{\partial U_i}{\partial x_i}$ is retained while $\frac{\partial U_1}{\partial x_2} \simeq \frac{\partial U_3}{\partial x_2} \simeq 0$ is assumed.

Evaluating $\frac{\partial U_i}{\partial x_i}$ directly from the high-resolution non-linear fluctuation splitting scheme is impractical. Limiters such as Minmod are not continuously differentiable, and while the Van Albada limiter is differentiable its use does not lead to a convenient explicit form from which to evaluate $\frac{\partial U_i}{\partial x_i}$. As an approximation to $\frac{\partial U_i}{\partial x_i}$ for the non-linear scheme, derivatives are sought using linear distribution schemes. Two linear choices for fluctuation splitting are to use a linearity preserving (second-order spatial accuracy), non-monotonic distribution or an upwind, monotonic first-order distribution. The linearity-preserving, non-monotonic scheme is of the Lax-Wendroff type[100], and tends to produce dispersion waves in response to nodal displacements. This behavior tends to under-predict the change in solution value at the perturbed node relative to the fluctuation splitting scheme employed herein.

The linear upwind scheme is obtained from the present fluctuation splitting scheme by discarding the limiter. This scheme exhibits a dependency on the numbering of nodes within a triangle. To alleviate this dependency, the approximation to $\frac{\partial U_i}{\partial x_i}$ is built by looping over all cells connected to node i and renumbering the nodes within each triangle so that the current node is designated as node 2 of the triangle. It is emphasized that the linear upwind scheme is not used in the calculation of the solution, but only employed to provide an estimate of the solution variation with respect to nodal displacements, providing the forcing functions for the mesh adaption.

The distribution to node 2 of a triangle using the linear upwind scheme is obtained from Eqns. 3.22–3.30, with $\phi^{*\xi} = \phi^{\xi}$ and $\phi^{*\eta} = \phi^{\eta}$, as,

$$\alpha^+(U_1 - U_2) - \beta^-(U_3 - U_2) \ = \alpha^+ U_1 + (\beta^- - \alpha^+)U_2 - \beta^- U_3 \tag{4.36}$$

where,

$$\alpha^{\pm} \ = \frac{\alpha \pm |\alpha|}{2} = \alpha \frac{1 \pm \mathrm{sign}(\alpha)}{2} \qquad \beta^{\pm} = \frac{\beta \pm |\beta|}{2} = \beta \frac{1 \pm \mathrm{sign}(\beta)}{2} \tag{4.37}$$

Assembling all contributions from the surrounding cells of the form in Eqn. 4.36 and

solving for the steady state value of the current node yields, with $U_2 = U_i$,

$$U_i \sum_{\mathsf{T}} (\alpha^+ - \beta^-) \; = \sum_{\mathsf{T}} (\alpha^+ U_1 - \beta^- U_3) \tag{4.38}$$

The variation of the nodal solution with respect to nodal displacement can now be evaluated as,

$$\frac{\partial U_i}{\partial x_i} \sum_{\mathsf{T}} (\alpha^+ - \beta^-) \; = \sum_{\mathsf{T}} \left(U_1 \frac{\partial \alpha^+}{\partial x_i} - U_3 \frac{\partial \beta^-}{\partial x_i} \right) - U_i \sum_{\mathsf{T}} \left(\frac{\partial \alpha^+}{\partial x_i} - \frac{\partial \beta^-}{\partial x_i} \right) \tag{4.39}$$

Recall that the solution at surrounding nodes is held fixed during displacements of the current node.

The derivatives of the functions in Eqn. 4.37 are defined,

$$\frac{\partial \alpha^+}{\partial x} = \begin{cases} 0 & \alpha < 0 \\ \frac{\partial \alpha}{\partial x} & \alpha \geq 0 \end{cases} \qquad \frac{\partial \beta^-}{\partial x} = \begin{cases} \frac{\partial \beta}{\partial x} & \beta \leq 0 \\ 0 & \beta > 0 \end{cases} \tag{4.40}$$

Further, Eqn. 3.22 leads to,

$$\frac{\partial \alpha}{\partial x_2} = \frac{\tilde{A}^{\mathsf{y}}}{2} + \frac{1}{2} \ell_1 \hat{n}_1 \cdot \frac{\partial \tilde{\vec{A}}}{\partial x_2} \qquad \frac{\partial \alpha}{\partial y_2} = -\frac{\tilde{A}^{\mathsf{x}}}{2} + \frac{1}{2} \ell_1 \hat{n}_1 \cdot \frac{\partial \tilde{\vec{A}}}{\partial y_2} \tag{4.41}$$

$$\frac{\partial \beta}{\partial x_2} = \frac{\tilde{A}^{\mathsf{y}}}{2} - \frac{1}{2} \ell_3 \hat{n}_3 \cdot \frac{\partial \tilde{\vec{A}}}{\partial x_2} \qquad \frac{\partial \beta}{\partial y_2} = -\frac{\tilde{A}^{\mathsf{x}}}{2} - \frac{1}{2} \ell_3 \hat{n}_3 \cdot \frac{\partial \tilde{\vec{A}}}{\partial y_2} \tag{4.42}$$

For non-linear problems an implicit relation arises for $\frac{\partial U_i}{\partial x_i}$ and $\frac{\partial \tilde{\vec{A}}}{\partial x_i}$. One option is to neglect the derivatives of $\tilde{\vec{A}}$ in Eqns. 4.41 and 4.42. Another option would be to lag these same terms from the previous iteration level.

Second derivatives of the fluctuation with respect to variation of a nodal location follow from Eqns. 4.32 and 4.33, again incorporating the approximation $\frac{\partial U_1}{\partial x_2} \simeq \frac{\partial U_3}{\partial x_2} \simeq 0$,

$$\frac{\partial^2 \phi}{\partial x_2^2} = (U_1 - U_3) \frac{\partial \tilde{A}^{\mathsf{y}}}{\partial x_2} + \ell_2 \hat{n}_2 \cdot \frac{\partial \tilde{\vec{A}}}{\partial x_2} \frac{\partial U_2}{\partial x_2} + \frac{\ell_2}{2} \hat{n}_2 \cdot \tilde{\vec{A}} \frac{\partial^2 U_2}{\partial x_2^2} + \sum_{j=1}^{3} U_j \frac{\ell_j}{2} \hat{n}_j \cdot \frac{\partial^2 \tilde{\vec{A}}}{\partial x_2^2} \tag{4.43}$$

$$\frac{\partial^2 \phi}{\partial y_2^2} = (U_3 - U_1) \frac{\partial \tilde{A}^{\mathsf{x}}}{\partial y_2} + \ell_2 \hat{n}_2 \cdot \frac{\partial \tilde{\vec{A}}}{\partial y_2} \frac{\partial U_2}{\partial y_2} + \frac{\ell_2}{2} \hat{n}_2 \cdot \tilde{\vec{A}} \frac{\partial^2 U_2}{\partial y_2^2} + \sum_{j=1}^{3} U_j \frac{\ell_j}{2} \hat{n}_j \cdot \frac{\partial^2 \tilde{\vec{A}}}{\partial y_2^2} \tag{4.44}$$

$$\frac{\partial^2\phi}{\partial x_2\partial y_2} = \frac{1}{2}\left[(U_3-U_1)\left(\frac{\partial\tilde{A}^{\mathsf{x}}}{\partial x_2}-\frac{\partial\tilde{A}^{\mathsf{y}}}{\partial y_2}\right)+\ell_2\hat{n}_2\cdot\left(\frac{\partial\tilde{\vec{A}}}{\partial y_2}\frac{\partial U_2}{\partial x_2}+\frac{\partial\tilde{\vec{A}}}{\partial x_2}\frac{\partial U_2}{\partial y_2}\right)\right.$$
$$\left.+\ell_2\hat{n}_2\cdot\tilde{\vec{A}}\frac{\partial^2 U_2}{\partial x_2\partial y_2}+\sum_{j=1}^{3}U_j\ell_j\hat{n}_j\cdot\frac{\partial^2\tilde{\vec{A}}}{\partial x_2\partial y_2}\right] \quad (4.45)$$

Second derivatives of $\tilde{\vec{A}}$ are developed in an analogous manner to its first derivatives.

The second derivatives of the solution at the current node with respect to variations in the position of the current node can be developed from the first derivative expression in Eqn. 4.39, again with the approximation $\frac{\partial U_1}{\partial x_2}\simeq\frac{\partial U_3}{\partial x_2}\simeq 0$,

$$\frac{\partial^2 U_i}{\partial x_i^2}\sum_{\mathsf{T}}(\alpha^+-\beta^-) = \sum_{\mathsf{T}}\left(U_1\frac{\partial^2\alpha^+}{\partial x_i^2}-U_3\frac{\partial^2\beta^-}{\partial x_i^2}\right)$$
$$-2\frac{\partial U_i}{\partial x_i}\sum_{\mathsf{T}}\left(\frac{\partial\alpha^+}{\partial x_i}-\frac{\partial\beta^-}{\partial x_i}\right)-U_i\sum_{\mathsf{T}}\left(\frac{\partial^2\alpha^+}{\partial x_i^2}-\frac{\partial^2\beta^-}{\partial x_i^2}\right) \quad (4.46)$$

$$\frac{\partial^2 U_i}{\partial x_i\partial y_i}\sum_{\mathsf{T}}(\alpha^+-\beta^-) = \sum_{\mathsf{T}}\left(U_1\frac{\partial^2\alpha^+}{\partial x_i\partial y_i}-U_3\frac{\partial^2\beta^-}{\partial x_i\partial y_i}\right)-\frac{\partial U_i}{\partial x_i}\sum_{\mathsf{T}}\left(\frac{\partial\alpha^+}{\partial y_i}-\frac{\partial\beta^-}{\partial y_i}\right)$$
$$-\frac{\partial U_i}{\partial y_i}\sum_{\mathsf{T}}\left(\frac{\partial\alpha^+}{\partial x_i}-\frac{\partial\beta^-}{\partial x_i}\right)-U_i\sum_{\mathsf{T}}\left(\frac{\partial^2\alpha^+}{\partial x_i\partial y_i}-\frac{\partial^2\beta^-}{\partial x_i\partial y_i}\right) \quad (4.47)$$

The remaining derivatives to be specified follow from Eqns. 4.41 and 4.42,

$$\frac{\partial^2\alpha}{\partial x_2^2} = \frac{\partial\tilde{A}^{\mathsf{y}}}{\partial x_2}+\frac{1}{2}\ell_1\hat{n}_1\cdot\frac{\partial^2\tilde{\vec{A}}}{\partial x_2^2} \qquad \frac{\partial^2\alpha}{\partial y_2^2} = -\frac{\partial\tilde{A}^{\mathsf{x}}}{\partial y_2}+\frac{1}{2}\ell_1\hat{n}_1\cdot\frac{\partial^2\tilde{\vec{A}}}{\partial y_2^2}$$
$$\frac{\partial^2\alpha}{\partial x_2\partial y_2} = \frac{1}{2}\frac{\partial\tilde{A}^{\mathsf{y}}}{\partial y_2}-\frac{1}{2}\frac{\partial\tilde{A}^{\mathsf{x}}}{\partial x_2}+\frac{1}{2}\ell_1\hat{n}_1\cdot\frac{\partial^2\tilde{\vec{A}}}{\partial x_2\partial y_2} \quad (4.48)$$

$$\frac{\partial^2\beta}{\partial x_2^2} = \frac{\partial\tilde{A}^{\mathsf{y}}}{\partial x_2}-\frac{1}{2}\ell_3\hat{n}_3\cdot\frac{\partial^2\tilde{\vec{A}}}{\partial x_2^2} \qquad \frac{\partial^2\beta}{\partial y_2^2} = -\frac{\partial\tilde{A}^{\mathsf{x}}}{\partial y_2}-\frac{1}{2}\ell_3\hat{n}_3\cdot\frac{\partial^2\tilde{\vec{A}}}{\partial y_2^2}$$
$$\frac{\partial^2\beta}{\partial x_2\partial y_2} = \frac{1}{2}\frac{\partial\tilde{A}^{\mathsf{y}}}{\partial y_2}-\frac{1}{2}\frac{\partial\tilde{A}^{\mathsf{x}}}{\partial x_2}-\frac{1}{2}\ell_3\hat{n}_3\cdot\frac{\partial^2\tilde{\vec{A}}}{\partial x_2\partial y_2} \quad (4.49)$$

For Ξ, Roe[77] chooses $\Xi_{\mathsf{T}} = \frac{1}{S_{\mathsf{T}}}$, for which the derivatives are,

$$\frac{\partial \Xi_{\mathsf{T}}}{\partial x_i} = -\frac{1}{S_{\mathsf{T}}^2}\frac{\partial S_{\mathsf{T}}}{\partial x_i} \tag{4.50}$$

$$\frac{\partial^2 \Xi_{\mathsf{T}}}{\partial x_i^2} = \frac{2}{S_{\mathsf{T}}^3}\left(\frac{\partial S_{\mathsf{T}}}{\partial x_i}\right)^2 - \frac{1}{S_{\mathsf{T}}^2}\frac{\partial^2 S_{\mathsf{T}}}{\partial x_i^2} \tag{4.51}$$

$$\frac{\partial^2 \Xi_{\mathsf{T}}}{\partial x_i \partial y_i} = \frac{2}{S_{\mathsf{T}}^3}\frac{\partial S_{\mathsf{T}}}{\partial x_i}\frac{\partial S_{\mathsf{T}}}{\partial y_i} - \frac{1}{S_{\mathsf{T}}^2}\frac{\partial^2 S_{\mathsf{T}}}{\partial x_i \partial y_i} \tag{4.52}$$

The area of a triangle is,

$$\begin{aligned} S_{\mathsf{T}} &= \frac{1}{2}\left[x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)\right] \\ &= -\frac{1}{2}\left[y_1(x_2 - x_3) + y_2(x_3 - x_1) + y_3(x_1 - x_2)\right] \end{aligned} \tag{4.53}$$

leading to the derivatives,

$$\frac{\partial S_{\mathsf{T}}}{\partial x_2} = \frac{y_3 - y_1}{2} \qquad \frac{\partial S_{\mathsf{T}}}{\partial y_2} = \frac{x_1 - x_3}{2} \tag{4.54}$$

and,

$$\frac{\partial^2 S_{\mathsf{T}}}{\partial x_2^2} = \frac{\partial^2 S_{\mathsf{T}}}{\partial y_2^2} = \frac{\partial^2 S_{\mathsf{T}}}{\partial x_2 \partial y_2} = 0 \tag{4.55}$$

This choice of weighting emphasizes fluctuations on the smaller cells.

An alternative is to weight all cells equally, with Ξ_{T}=1. Two other obvious choices for weighting are Ξ_{T}=S_{T}, emphasizing fluctuations on the larger cells, and $\Xi_{\mathsf{T}} = \frac{1}{S_{\mathsf{T}}^2}$, for even stronger emphasis on the smaller cells. The derivatives for this latter case are, with Eqn. 4.55,

$$\frac{\partial \Xi_{\mathsf{T}}}{\partial x_i} = -\frac{2}{S_{\mathsf{T}}^3}\frac{\partial S_{\mathsf{T}}}{\partial x_i} \tag{4.56}$$

$$\frac{\partial^2 \Xi_{\mathsf{T}}}{\partial x_i^2} = \frac{6}{S_{\mathsf{T}}^4}\left(\frac{\partial S_{\mathsf{T}}}{\partial x_i}\right)^2 \tag{4.57}$$

$$\frac{\partial^2 \Xi_{\mathsf{T}}}{\partial x_i \partial y_i} = \frac{6}{S_{\mathsf{T}}^4}\frac{\partial S_{\mathsf{T}}}{\partial x_i}\frac{\partial S_{\mathsf{T}}}{\partial y_i} \tag{4.58}$$

The fluctuation splitting node movement techniques are evaluated for the familiar circular advection problem of the present chapter. One cycle of point deletion results in the grid and solution of Figure 4.24. The grid contains 70 nodes.

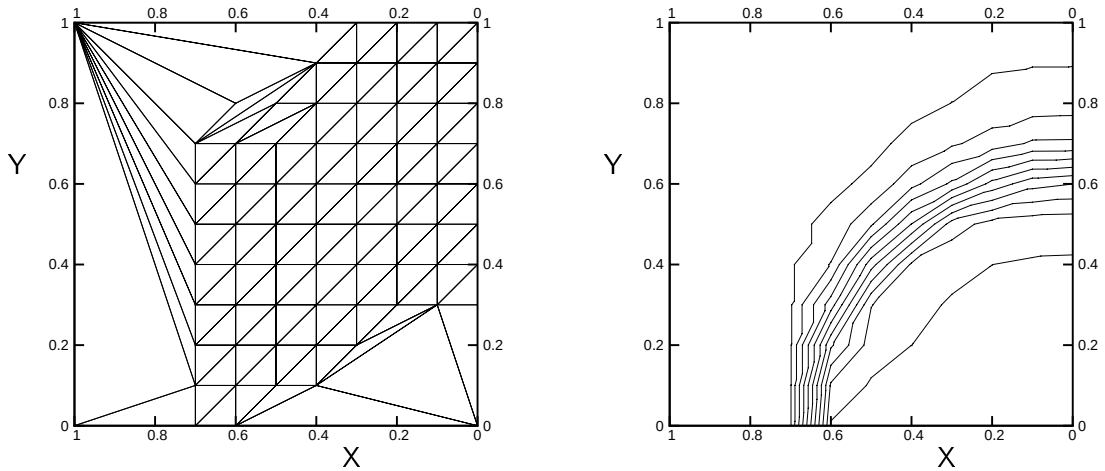


Figure 4.24: Starting mesh and solution for demonstrating fluctuation splitting node movement schemes. Contours vary on (0,1), with 0.1 increment.

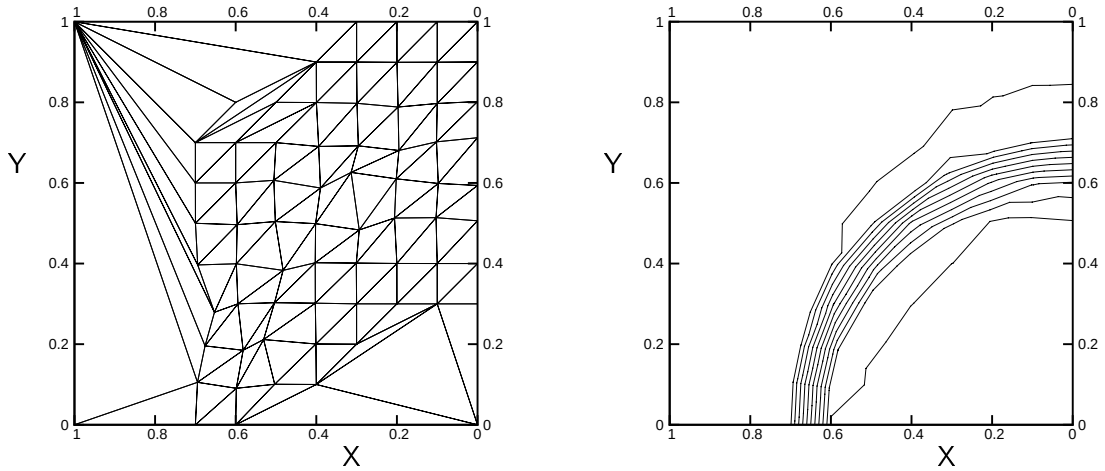


Figure 4.25: Mesh and solution after four fixed-point mesh adaptions with fluctuation splitting. Contours vary on (0,1), with 0.1 increment.

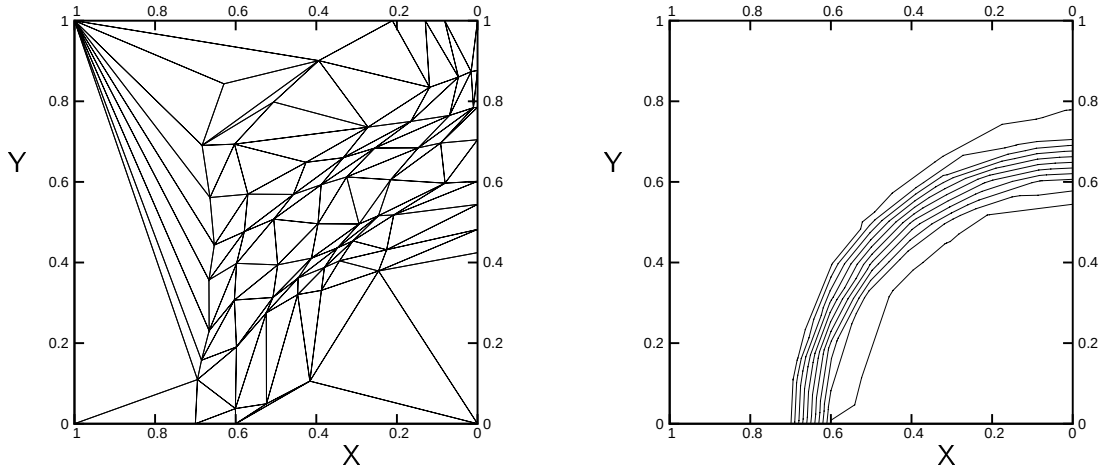


Figure 4.26: Mesh and solution after four diagonal Newton mesh adaptions with fluctuation splitting. Contours vary on (0,1), with 0.1 increment.

Applying the fixed-point iteration strategy for four node movement cycles results in the mesh and solution of Figure 4.25. This fluctuation splitting solution shows a somewhat improved accuracy over the optimally adapted DMFDSFV solution of Figure 4.16, despite a mesh with fewer than half the number of nodes. The fixed-point procedure is found to be a slowly converging technique. To improve the response, local iterations and/or over-relaxation can be applied. Unfortunately, the amount of over/under-relaxation for optimal convergence is problem and grid dependent. For the results of Figure 4.25, an over-relaxation factor of 10 was used with five local iterations per global adaption, with Ξ_T=1.

Employing the full Newton strategy to improve the convergence of the minimization procedure has not been encouraging for this case. Numerical stiffness due to a vanishing determinant in Eqn. 4.28 is the culprit, a condition that is likely to worsen for finer grids.

However, the diagonalized Newton scheme proves robust and fast for this case, without the need for either local iterations or a tunable over/under-relaxation parameter. Four cycles of the diagonal Newton scheme (Ξ_T=1) yield the grid and solution in Figure 4.26. The mesh adaption is clearly more aggressive than for the fixed-

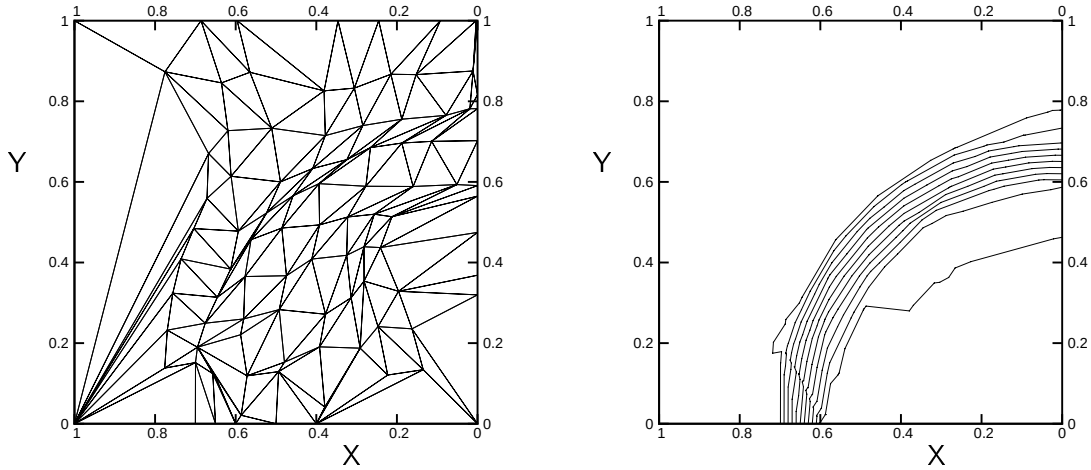
point scheme, achieving slightly improved resolution of the solution at the outflow boundary.

Point insertion

Additional nodes are created by subdividing edges, in a manner similar to that described in section 4.1.1. However, instead of using an edge-based error estimate, the fluctuations in the cells to either side of the edge are used to trigger mesh refinement. If the magnitudes of the fluctuations in both cells, or the only cell for a boundary edge, exceed a threshold, then the edge is split by adding a new node at the edge midpoint.

The various operations for performing mesh adaption in the fluctuation splitting context are combined as a series of sequential steps to form a complete adaption cycle. The steps are arranged in the same order as is enumerated in section 4.1.2. The complete fluctuation splitting mesh adaption is applied to the circular advection test problem for a single cycle. The initial mesh and solution were previously shown in Figure 4.15. The resulting mesh, containing 96 nodes, is shown in Figure 4.27(a). The associated solution, showing very good accuracy, is in Figure 4.27(b). The fluctuation splitting solution-adaptive procedure achieves slightly better solution resolution in a single adaption sweep than the state-of-the-art finite volume solution-adaptive procedure was able to obtain in three adaption cycles on a mesh with 50 percent more nodes. Thus, the present fluctuation splitting adaptive scheme is more than three times as fast as the finite volume scheme which utilizes *a posteriori* error estimates.

Note, however, that the 'optimal' fluctuation splitting mesh from Figure 4.19(a) is not achieved with only one adaption cycle. In fact, it appears that for this variable-velocity advection problem the local adaption strategy falls short of achieving the globally-optimal mesh. A retreat is therefore made from the goal of globally-optimal mesh adaption, and instead the present method seeks a consistently-improved solution/mesh combination with each adaption cycle.



(a) (b)

Figure 4.27: Mesh and solution for circular advection problem after a single mesh adaption cycle with fluctuation splitting. Contours vary on (0,1), with 0.1 increment.

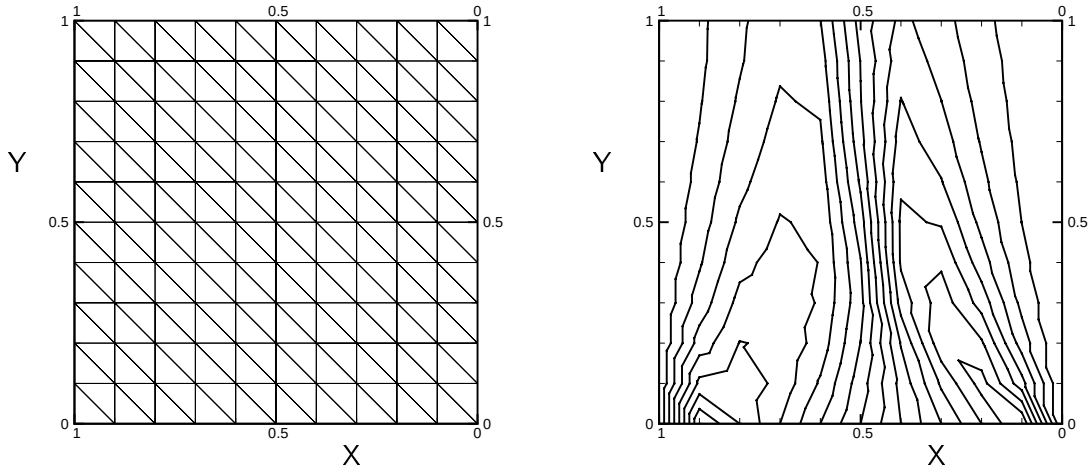
4.3 Non-linear Advection

The two solution-adaptive schemes, a finite volume solver with adaption based on edge error estimates and a fluctuation splitting solver with adaption to minimize fluctuations, are applied to the two-dimensional non-linear advection problem of section 3.2.4, repeated here,

$$U_t + UU_x + U_y = 0$$
$$U(-1, y) = U(0, y) = 0 \qquad U(x, 0) = -2x - 1$$

To avoid the problem of a doubly-defined node at $(-1, 0)$ and $(0, 0)$, the inflow corners are set to $U = 0$, and thus the expansion at the inflow is linearly distributed between the corner points and the first node in from the corner, along the y-axis.

The DMFDSFV scheme with edge error estimates is applied without modification. The initial mesh, Figure 4.28(a), contains 121 nodes. The DMFDSFV solution on this mesh, Figure 4.28(b), is very diffused and exhibits grid-induced asymmetries. Three full cycles of solution adaption are applied, nearly doubling the mesh density to

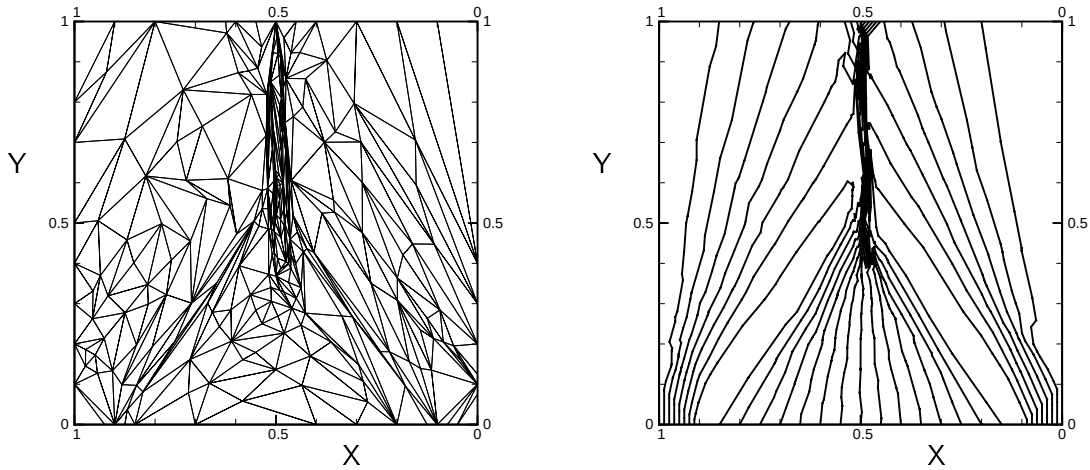


(a) (b)

Figure 4.28: Initial mesh and DMFDSFV solution for non-linear advection case. Contours vary on (-1,1), with 0.1 increment.

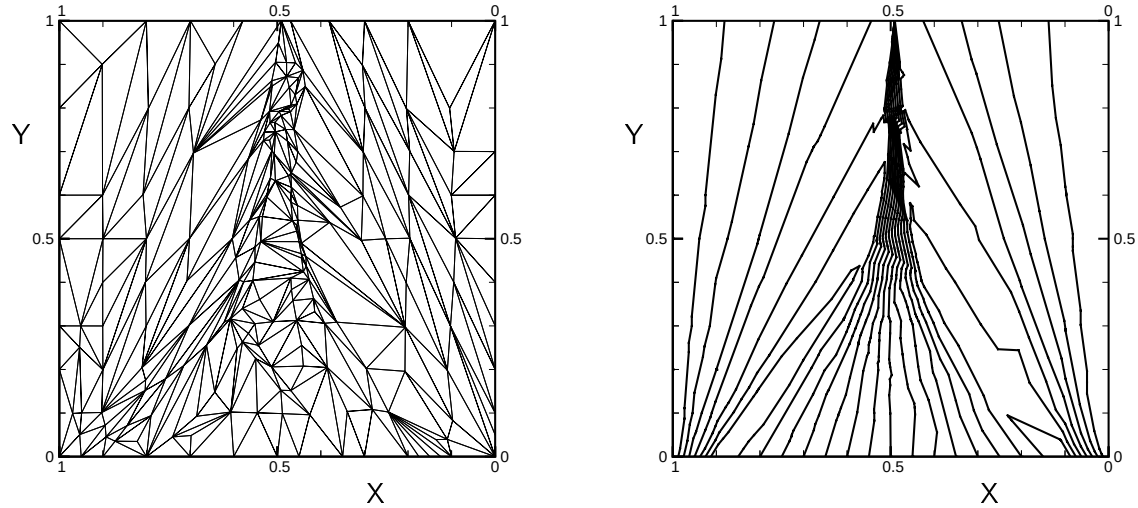
237 nodes, Figure 4.29(a). The solution on this mesh shows a dramatic improvement, Figure 4.29(b), for shock thickness, shock speed, point of coalescence, and preservation of extremum in smooth regions. Note, however, that there is some asymmetry between the expansion fans toward the top of the domain, that the shock is not entirely straight, and that the compression fan begins to coalesce into a shock at $y = 0.4$, instead of at $y = -0.5$, the correct location.

The fluctuation splitting adaption scheme is also applied for three cycles to the same problem and initial grid. The solution on the initial mesh is similar in character to the DMFDSFV solution in Figure 4.28(a). The adapted mesh, containing 206 nodes, and corresponding solution are shown in Figure 4.30. The adapted fluctuation splitting solution, using 14 percent fewer nodes, exhibits slightly greater accuracy than the adapted DMFDSFV solution, particularly in the expansion fan symmetry and shock coalescence point. One feature that is better resolved in the DMFDSFV solution is the extremum between the compression and expansion fans on the lower right-hand side. The fluctuation splitting shock is wider at the coalescence point but then has comparable crispness to the DMFDSFV result and is a little straighter with



(a) (b)

Figure 4.29: Mesh and DMFDSFV solution after three adaption cycles, non-linear advection case. Contours vary on (-1,1), with 0.1 increment.



(a) (b)

Figure 4.30: Mesh and fluctuation splitting solution after three adaption cycles, non-linear advection case. Contours vary on (-1,1), with 0.1 increment.

increasing y. The fluctuation splitting shock at the outflow is ever so slightly offset to the right from $x = -0.5$, the correct location.

The DMFDSFV and fluctuation splitting meshes, Figures 4.29(a) and 4.30(a), have similar character in the expansion and compression fans. The fluctuation splitting adaption does not cluster as many points to the shock, resulting in overall 14 percent fewer nodes. This fact that the fluctuation splitting adaption scheme can resolve crisp shocks without excessive clustering normal to the shock may have favorable implications when considering fluid dynamic problems with extremely strong shocks in the vicinity of more subtle, though still important, features, such as an entropy layer.

While adapting to the non-linear problem, it became necessary to include an eigenvalue limiter in the fluctuation splitting algorithm to prevent expansion shocks. The one-dimensional eigenvalue limiting of Harten and Hymen (see section 2.2.2) is extended to multiple dimensions by searching for expansions in the ξ and η directions separately. Combining Eqns. 3.25–3.29, the artificial dissipation terms can be recast as,

$$\phi'^{\xi} = -|\alpha|\Delta_\xi U + \text{sign}(\alpha) M_\psi, \qquad \phi'^{\eta} = -|\beta|\Delta_\eta U - \text{sign}(\beta) M_\psi \tag{4.59}$$

$|\alpha|$ and $|\beta|$ are then limited as in Eqn. 2.46 with the parameter ϵ (Eqn. 2.47) defined as,

$$\epsilon_{(\alpha)} = \frac{1}{2}\max\left[0,\ \ell_1\hat{n}_1\cdot(\tilde{\vec{A}} - \vec{A}_1),\ \ell_1\hat{n}_1\cdot(\vec{A}_2 - \tilde{\vec{A}})\right] \tag{4.60}$$

$$\epsilon_{(\beta)} = \frac{1}{2}\max\left[0,\ \ell_3\hat{n}_3\cdot(\vec{A}_2 - \tilde{\vec{A}}),\ \ell_3\hat{n}_3\cdot(\tilde{\vec{A}} - \vec{A}_3)\right] \tag{4.61}$$

respectively.

As a final note to this section, the fluctuation splitting solver is applied to the adapted mesh previously used with DMFDSFV, from Figure 4.29(a). The fluctuation splitting solution on this mesh is shown in Figure 4.31, as a demonstration that the fluctuation splitting solver still works on a mesh adapted to solution curvature.

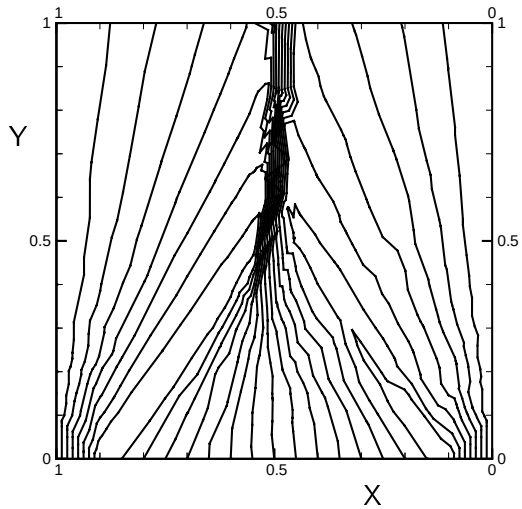


Figure 4.31: Fluctuation splitting solution on curvature-clustering mesh. Contours vary on (-1,1), with 0.1 increment.

4.4 Advection-Diffusion

The two solution-adaptive schemes are applied to the Smith & Hutton advection-diffusion problem of section 3.4, starting on the 1928-node Mesh C. The finite volume adaption with clustering based on edge error estimates is applied without modification, resulting in a 619-node mesh. The RMS difference between the solution profile at the outflow boundary for the adapted DMFDSFV result and the grid-converged solution profile from the previous chapter is listed in Table 4.1 along with the RMS difference for the DMFDSFV solution on the unadapted mesh C. Two cycles of mesh adaption have been performed, reducing the required number of nodes by two-thirds while improving the RMS outflow resolution by 28 percent.

The incorporation of physical diffusion into the adaption strategy utilized with the

	DMFDSFV	Fluctuation splitting
adapted	0.0208	0.0063
unadapted	0.0288	0.0068

Table 4.1: RMS difference of solution outflow profile relative to grid-converged result.

fluctuation splitting scheme requires some modifications. A check on the magnitude of the diffusive fluctuations in the surrounding cells is added to the criteria for deleting a node. For edge swapping, the RMS of the diffusive distributions are added to the RMS advective fluctuations to determine the optimal connectivity. An edge is split, adding a node, if the diffusive distributions sent to the end nodes of the edge exceed a threshold value.

For nodal displacements, the advective fluctuation on a cell, Eqn. 4.31, is modified to include the diffusive distribution sent to the current node, ϕ_i^v defined in Eqn. 3.51, repeated here,

$$\phi_i^v = -\frac{\bar{\mu}\ell_i}{4S_\mathsf{T}} \sum_{j=1}^{3} U_j \ell_j \hat{n}_j \cdot \hat{n}_i \tag{4.62}$$

The additional derivatives to be added to Eqns. 4.32 and 4.33 take the form,

$$\frac{\partial \phi_2^v}{\partial x_2} = -\frac{\bar{\mu}}{4S_\mathsf{T}} \left[\left(\frac{1}{\bar{\mu}} \frac{\partial \bar{\mu}}{\partial x_2} - \frac{1}{S_\mathsf{T}} \frac{\partial S_\mathsf{T}}{\partial x_2} \right) \sum_{j=1}^{3} U_j \ell_j \ell_2 \hat{n}_j \cdot \hat{n}_2 + \ell_2^2 \frac{\partial U_2}{\partial x_2} + (x_3 - x_1)(U_1 - U_3) \right] \tag{4.63}$$

$$\frac{\partial \phi_2^v}{\partial y_2} = -\frac{\bar{\mu}}{4S_\mathsf{T}} \left[\left(\frac{1}{\bar{\mu}} \frac{\partial \bar{\mu}}{\partial y_2} - \frac{1}{S_\mathsf{T}} \frac{\partial S_\mathsf{T}}{\partial y_2} \right) \sum_{j=1}^{3} U_j \ell_j \ell_2 \hat{n}_j \cdot \hat{n}_2 + \ell_2^2 \frac{\partial U_2}{\partial y_2} + (y_3 - y_1)(U_1 - U_3) \right] \tag{4.64}$$

where the assumption $\frac{\partial U_1}{\partial x_2} \simeq \frac{\partial U_3}{\partial x_2} \simeq 0$ has already been applied. The second derivatives of the diffusive distribution follow as,

$$\begin{aligned} \frac{\partial^2 \phi_2^v}{\partial x_2^2} &= -\frac{\bar{\mu}\ell_2^2}{4S_\mathsf{T}} \frac{\partial^2 U_2}{\partial x_2^2} + \left(\frac{\bar{\mu}}{4S_\mathsf{T}^2} \frac{\partial S_\mathsf{T}}{\partial x_2} - \frac{1}{4S_\mathsf{T}} \frac{\partial \bar{\mu}}{\partial x_2} \right) \left[\ell_2^2 \frac{\partial U_2}{\partial x_2} + (x_3 - x_1)(U_1 - U_3) \right] \\ &+ \left[\frac{1}{4\bar{\mu} S_\mathsf{T}} \left(\frac{\partial \bar{\mu}}{\partial x_2} \right)^2 - \frac{1}{4S_\mathsf{T}} \frac{\partial^2 \bar{\mu}}{\partial x_2^2} + \frac{\bar{\mu}}{4S_\mathsf{T}^2} \frac{\partial^2 S_\mathsf{T}}{\partial x_2^2} - \frac{\bar{\mu}}{4S_\mathsf{T}^3} \left(\frac{\partial S_\mathsf{T}}{\partial x_2} \right)^2 \right] \sum_{j=1}^{3} U_j \ell_j \ell_2 \hat{n}_j \cdot \hat{n}_2 \\ &+ \left(\frac{1}{\bar{\mu}} \frac{\partial \bar{\mu}}{\partial x_2} - \frac{1}{S_\mathsf{T}} \frac{\partial S_\mathsf{T}}{\partial x_2} \right) \frac{\partial \phi_2^v}{\partial x_2} \end{aligned} \tag{4.65}$$

$$\begin{aligned}\frac{\partial^2\phi_2^v}{\partial y_2^2} &= -\frac{\bar{\mu}\ell_2^2}{4S_\mathsf{T}}\frac{\partial^2 U_2}{\partial y_2^2} + \left(\frac{\bar{\mu}}{4S_\mathsf{T}^2}\frac{\partial S_\mathsf{T}}{\partial y_2} - \frac{1}{4S_\mathsf{T}}\frac{\partial\bar{\mu}}{\partial y_2}\right)\left[\ell_2^2\frac{\partial U_2}{\partial y_2} + (y_3-y_1)(U_1-U_3)\right] \\ &+ \left[\frac{1}{4\bar{\mu}S_\mathsf{T}}\left(\frac{\partial\bar{\mu}}{\partial y_2}\right)^2 - \frac{1}{4S_\mathsf{T}}\frac{\partial^2\bar{\mu}}{\partial y_2^2} + \frac{\bar{\mu}}{4S_\mathsf{T}^2}\frac{\partial^2 S_\mathsf{T}}{\partial y_2^2} - \frac{\bar{\mu}}{4S_\mathsf{T}^3}\left(\frac{\partial S_\mathsf{T}}{\partial y_2}\right)^2\right]\sum_{j=1}^{3} U_j\ell_j\ell_2\hat{n}_j\cdot\hat{n}_2 \\ &+ \left(\frac{1}{\bar{\mu}}\frac{\partial\bar{\mu}}{\partial y_2} - \frac{1}{S_\mathsf{T}}\frac{\partial S_\mathsf{T}}{\partial y_2}\right)\frac{\partial\phi_2^v}{\partial y_2} \end{aligned} \tag{4.66}$$

$$\begin{aligned}\frac{\partial^2\phi_2^v}{\partial x_2\partial y_2} &= -\frac{\bar{\mu}\ell_2^2}{4S_\mathsf{T}}\frac{\partial^2 U_2}{\partial x_2\partial y_2} + \left(\frac{\bar{\mu}}{4S_\mathsf{T}^2}\frac{\partial S_\mathsf{T}}{\partial x_2} - \frac{1}{4S_\mathsf{T}}\frac{\partial\bar{\mu}}{\partial x_2}\right)\left[\ell_2^2\frac{\partial U_2}{\partial y_2} + (y_3-y_1)(U_1-U_3)\right] \\ &+ \left(\frac{1}{4\bar{\mu}S_\mathsf{T}}\frac{\partial\bar{\mu}}{\partial x_2}\frac{\partial\bar{\mu}}{\partial y_2} - \frac{1}{4S_\mathsf{T}}\frac{\partial^2\bar{\mu}}{\partial x_2\partial y_2} + \frac{\bar{\mu}}{4S_\mathsf{T}^2}\frac{\partial^2 S_\mathsf{T}}{\partial x_2\partial y_2} - \frac{\bar{\mu}}{4S_\mathsf{T}^3}\frac{\partial S_\mathsf{T}}{\partial x_2}\frac{\partial S_\mathsf{T}}{\partial y_2}\right)\sum_{j=1}^{3} U_j\ell_j\ell_2\hat{n}_j\cdot\hat{n}_2 \\ &+ \left(\frac{1}{\bar{\mu}}\frac{\partial\bar{\mu}}{\partial y_2} - \frac{1}{S_\mathsf{T}}\frac{\partial S_\mathsf{T}}{\partial y_2}\right)\frac{\partial\phi_2^v}{\partial x_2} \end{aligned} \tag{4.67}$$

The derivatives of the cell-averaged diffusion coefficient scale like,

$$\frac{\partial\bar{\mu}}{\partial x_2} \sim \frac{1}{3}\frac{\partial\mu_2}{\partial x_2} \qquad\qquad \frac{\partial^2\bar{\mu}}{\partial x_2^2} \sim \frac{1}{3}\frac{\partial^2\mu_2}{\partial x_2^2} \tag{4.68}$$

The variation of the dependent variable at the current node with respect to the nodal position also needs to be modified to account for the diffusion terms. The steady-state value of the current node, Eqn. 4.38, now becomes,

$$U_i\sum_\mathsf{T}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_\mathsf{T}}\right) = \sum_\mathsf{T}\left[\alpha^+U_1 - \beta^-U_3 - \frac{\bar{\mu}\ell_i}{4S_\mathsf{T}}\left(U_1\ell_1\hat{n}_1 + U_3\ell_3\hat{n}_3\right)\cdot\hat{n}_i\right] \tag{4.69}$$

and the variation, Eqn. 4.39, is replaced by,

$$\begin{aligned}\frac{\partial U_i}{\partial x_i}\sum_\mathsf{T}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_2^2}{4S_\mathsf{T}}\right) &= \sum_\mathsf{T}\left\{U_1\frac{\partial\alpha^+}{\partial x_i} - U_3\frac{\partial\beta^-}{\partial x_i}\right. \\ &\left. -\frac{\bar{\mu}}{4S_\mathsf{T}}\left[\ell_i\left(\frac{1}{\bar{\mu}}\frac{\partial\bar{\mu}}{\partial x_i} - \frac{1}{S_\mathsf{T}}\frac{\partial S_\mathsf{T}}{\partial x_i}\right)\left(U_1\ell_1\hat{n}_1\cdot\hat{n}_i + U_3\ell_3\hat{n}_3\cdot\hat{n}_i\right) + (x_3-x_1)(U_1-U_3)\right]\right\} \\ &- U_i\sum_\mathsf{T}\left(\frac{\partial\alpha^+}{\partial x_i} - \frac{\partial\beta^-}{\partial x_i} + \frac{\ell_i^2}{4S_\mathsf{T}}\frac{\partial\bar{\mu}}{\partial x_i} - \frac{\ell_i^2\bar{\mu}}{4S_\mathsf{T}^2}\frac{\partial S_\mathsf{T}}{\partial x_i}\right) \end{aligned} \tag{4.70}$$

The second variations of the dependent variable, Eqns. 4.46 and 4.47, become,

$$\begin{aligned}\frac{\partial^2 U_i}{\partial x_i^2}\sum_{\mathsf{T}}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_{\mathsf{T}}}\right) &= -U_i\sum_{\mathsf{T}}\frac{\partial^2}{\partial x_i^2}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_{\mathsf{T}}}\right)\\ &+ \sum_{\mathsf{T}}\left[U_1\frac{\partial^2\alpha^+}{\partial x_i^2} - U_3\frac{\partial^2\beta^-}{\partial x_i^2} - 2\ell_i(U_1-U_3)(x_3-x_1)\frac{\partial}{\partial x_i}\left(\frac{\bar{\mu}}{4S_{\mathsf{T}}}\right)\right]\\ &- 2\frac{\partial U_i}{\partial x_i}\sum_{\mathsf{T}}\frac{\partial}{\partial x_i}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_{\mathsf{T}}}\right) \qquad (4.71)\end{aligned}$$

$$\begin{aligned}\frac{\partial^2 U_i}{\partial x_i\partial y_i}\sum_{\mathsf{T}}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_{\mathsf{T}}}\right) &= -U_i\sum_{\mathsf{T}}\frac{\partial^2}{\partial x_i\partial y_i}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_{\mathsf{T}}}\right)\\ &- \frac{\partial U_i}{\partial x_i}\sum_{\mathsf{T}}\frac{\partial}{\partial y_i}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_{\mathsf{T}}}\right) + \sum_{\mathsf{T}}\left(U_1\frac{\partial^2\alpha^+}{\partial x_i\partial y_i} - U_3\frac{\partial^2\beta^-}{\partial x_i\partial y_i}\right)\\ &- \sum_{\mathsf{T}}\left\{\ell_i(U_1-U_3)\left[(x_3-x_1)\frac{\partial}{\partial y_i}\left(\frac{\bar{\mu}}{4S_{\mathsf{T}}}\right) + (y_3-y_1)\frac{\partial}{\partial x_i}\left(\frac{\bar{\mu}}{4S_{\mathsf{T}}}\right)\right]\right\}\\ &- \frac{\partial U_i}{\partial y_i}\sum_{\mathsf{T}}\frac{\partial}{\partial x_i}\left(\alpha^+ - \beta^- + \frac{\bar{\mu}\ell_i^2}{4S_{\mathsf{T}}}\right) \qquad (4.72)\end{aligned}$$

The RMS differences of the solution profiles at the outflow boundary relative to the grid-converged result are in Table 4.1. Although the fluctuation splitting solution on the unadapted 1928-node mesh is good, one cycle of mesh adaption reduces the number of nodes by a factor of 2.8 to 695 nodes, while still producing some (7 percent) improvement in accuracy.

4.5 Recapitulation

Current state-of-the-art for anisotropic unstructured mesh adaption based on *a posteriori* error estimates is implemented in an edge-based structure in conjunction with a finite volume solver. This type of adaption results in meshes where the node densities are clustered to regions of high curvature in the solution. Significant improvement in solution accuracy is verified using this technique on scalar model problems.

Recognizing the remarkable property of the discretized fluctuation splitting scheme that multi-dimensional advection can be solved exactly when one edge of each cell

is aligned with the characteristic direction, a different mesh adaption scheme is proposed. Retaining the mechanics for performing only local operations, *i.e.* point insertion/deletion, edge swapping, and nodal displacement, a solution-predictive approach is chosen in favor of *a posteriori* curvature clustering. The concept of aligning cell edges with characteristic directions is generalized as a minimization procedure to allow extension to diffusion problems and systems. Extending this process to non-linear problems leads to a multi-dimensional implementation of eigenvalue limiting.

It is seen that while performing a series of local optimizations does lead to globally improved solution accuracy and reduced grid sizes, in general a truly globally 'optimal' mesh is not achieved in a reasonable number of adaption cycles. However, the solution-predictive adaption in conjunction with the fluctuation splitting scheme does provide moderately more accurate solutions on smaller meshes for comparable number of adaption cycles versus the DMFDSFV solver with adaption driven by error estimates.

Considering extensions to three-dimensional hypersonic flow applications, perhaps the most promising difference between the two adaption strategies lies in their treatment of shocks. For the *a posteriori* adaption, the number of nodes clustered to the shock grows as the shock strength grows, which could lead to a bow shock dominating the adaption for hypersonic problems. In contrast, the minimization of fluctuations tends to merely align the grid with the shock, leaving the points outside the shock largely unaffected.

Chapter 5

Two-Dimensional Systems

5.1 Overview

The fluctuation splitting and DMFDSFV discretization schemes detailed in chapter 3 for scalar advection-diffusion problems are extended in the present chapter to the system of equations governing the compressible fluid dynamics of a dilute, ideal gas for two-dimensional and axisymmetric flows. First, the mass, momentum, and energy equations for an inviscid fluid, termed the Euler[86] equations, are formulated in both the fluctuation splitting and finite volume contexts. Then, the system is extended to include the effects of viscosity and conductivity as the Navier[87]-Stokes[88] equations. The formulation of the upwind fluctuation splitting scheme for the two-dimensional and axisymmetric Navier-Stokes equations is a leading-edge research area.

The chapter concludes with verification and validation of the schemes. A brief discussion of the coding strategy is included, immediately followed by a verification of the code using the test cases and methodology of Shinghal[101] and Roache[102]. The validation cases range from the incompressible flat-plate to a Mach-17 cylinder.

5.2 Inviscid Formulations

Spatial discretizations are developed for the Euler equations using both the DMFDS-FV and fluctuation splitting schemes. The two-dimensional development is extended

to cover axisymmetric problems through the additional appropriate radial derivatives, treated as source terms.

The non-dimensional system of equations are taken from appendix B, Eqn. B.9, with $\vec{\mathbf{F}}^v = \mathbf{B}'^v = 0$ as,

$$\varpi_a \mathbf{U}_t + \vec{\nabla}\cdot(\varpi_a \vec{\mathbf{F}}^i) = \varpi \mathbf{B}'^i \tag{5.1}$$

with,

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{pmatrix} \tag{5.2}$$

$$\vec{\mathbf{F}}^i = \begin{bmatrix} \rho u & \rho v \\ \rho u^2 + P & \rho uv \\ \rho uv & \rho v^2 + P \\ \rho uH & \rho vH \end{bmatrix} \tag{5.3}$$

$$\mathbf{B}'^i = (0, 0, P, 0)^\mathsf{T} \tag{5.4}$$

ϖ is a logical switch between two-dimensional ($\varpi = 0$) and axisymmetric ($\varpi = 1$) equations and $\varpi_a = 1 - \varpi + \varpi y$ is toggled by ϖ between 1 ($\varpi = 0$) and y ($\varpi = 1$).

5.2.1 Finite Volume

Integration of the dependent variables over the control volume about node i is performed as,

$$\int_{\Omega_i} \varpi_a \mathbf{U}_t \, d\Omega = \overline{\varpi}_a S_i \mathbf{U}_{i_t} \tag{5.5}$$

For two-dimensional $\overline{\varpi}_a = 1$, while for axisymmetric $\overline{\varpi}_a$ can be either taken as $\overline{\varpi}_a = y_i$, for mass-lumping to the node, or as the y-value of the centroid of Ω_i.

The axisymmetric source term, $\mathbf{B}'$, is simply evaluated at the node as,

$$\int_{\Omega_i} \mathbf{B}' \, d\Omega = S_i \mathbf{B}'_i \tag{5.6}$$

The discretization of the fluxes follows the development of section 3.2.1, with the only modifications being the inclusion of ϖ_a (equal to 1 for two-dimensional or the y-value of the quadrature point on the face for axisymmetric) and the definition of the artificial dissipation in Eqn. 3.8, which is,

$$\mathbf{\Phi} = \frac{1}{2}|\tilde{\vec{\mathbf{A}}}\cdot\hat{\mathbf{n}}|(\mathbf{U}_\mathsf{R} - \mathbf{U}_\mathsf{L}) \tag{5.7}$$

where by convention the right state is to the outside of the control volume while the left state is to the inside.

In an analogous manner to the conservative one-dimensional linearization of section 2.4.1, the parameter vector, $\mathbf{Z} = \sqrt{\rho}\ [1,\, u,\, v,\, H]^T$, is linearly averaged, $\bar{\mathbf{Z}} = \frac{1}{2}(\mathbf{Z}_\mathsf{L} + \mathbf{Z}_\mathsf{R})$, to provide the quantities,

$$\tilde{u} = \frac{\bar{Z}_2}{\bar{Z}_1}, \qquad \tilde{v} = \frac{\bar{Z}_3}{\bar{Z}_1}, \qquad \tilde{H} = \frac{\bar{Z}_4}{\bar{Z}_1} \tag{5.8}$$

and the Roe-density remains from Eqn. 2.64, $\tilde{\rho} = \sqrt{\rho_\mathsf{L}\rho_\mathsf{R}}$.

The projected flux Jacobian is decomposed as in Eqns. B.37–B.40,

$$|\tilde{\vec{\mathbf{A}}}\cdot\hat{\mathbf{n}}| = \tilde{\mathbf{X}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{X}}^{-1} \tag{5.9}$$

with $\mathbf{\Lambda}$ given in Eqn. B.38 and $\mathbf{X}$ given in Eqn. B.40.

The product $\tilde{\mathbf{X}}^{-1}(\mathbf{U}_\mathsf{R} - \mathbf{U}_\mathsf{L})$ is expressed analogously to Eqn. 2.68[23],

$$\tilde{\mathbf{X}}^{-1}(\mathbf{U}_\mathsf{R} - \mathbf{U}_\mathsf{L}) = \tilde{\mathbf{X}}^{-1}d\mathbf{U} = \frac{1}{2\tilde{a}^2}\begin{pmatrix} 2\tilde{a}^2 d\rho - 2dP \\ 2\tilde{a}^2(n^\mathsf{x}dv - n^\mathsf{y}du) \\ dP + \tilde{\rho}\tilde{a}\, d\mathcal{V} \\ dP - \tilde{\rho}\tilde{a}\, d\mathcal{V} \end{pmatrix} \tag{5.10}$$

where the projected velocity is $\mathcal{V} = \vec{V}\cdot\hat{n}$ and the averaged speed of sound for a perfect gas is,

$$\tilde{a}^2 = (\gamma - 1)\left(\tilde{H} - \frac{\tilde{u}^2 + \tilde{v}^2}{2}\right) \tag{5.11}$$

5.2.2 Fluctuation Splitting

In the fluctuation splitting context, the parameter vector is taken to vary linearly over each element. The assumed linear variation limits the scheme to second-order formal accuracy, but will allow for a positive formulation when combined with a non-linear distribution. Caraeni, Caraeni, and Fuchs[103], basing their work on the published formulations from the present report, use a quadratic variation of the parameter vector to derive a formally third-order scheme. That scheme, however, cannot be positive, a critical requirement for the robust computation of strong shocks.

For a perfect gas, changes to the conserved variables can be related to changes in the parameter vector as,

$$d\mathbf{U} \;= \mathbf{U}_Z \, d\mathbf{Z} \tag{5.12}$$

$$\mathbf{U}_Z \;= \begin{bmatrix} 2Z_1 & 0 & 0 & 0 \\ Z_2 & Z_1 & 0 & 0 \\ Z_3 & 0 & Z_1 & 0 \\ \frac{1}{\gamma}Z_4 & \frac{\gamma-1}{\gamma}Z_2 & \frac{\gamma-1}{\gamma}Z_3 & \frac{1}{\gamma}Z_1 \end{bmatrix} \tag{5.13}$$

Integration of $\varpi_a \mathbf{U}_t$ over an element leads to a mass matrix,

$$\int_\Omega \varpi_a \mathbf{U}_t \, d\Omega \;= \int_\Omega \varpi_a \mathbf{U}_Z \mathbf{Z}_t \, d\Omega \tag{5.14}$$

If mass-lumping to the nodes is employed, introducing temporal, but not spatial, errors, Eqn. 5.14 can be distributed to the nodes as in Eqn. 3.14, so that the sum of all contributions to node i equals $\varpi_{a_i} S_i \mathbf{U}_{i_t}$.

While some authors insist on upwinding source terms[3, 104], the present analysis considers an upwind distribution to be inappropriate for the axisymmetric source terms, which arise from purely geometric manipulations. The axisymmetric source term can be distributed to the node in a mass-lumped analogy as,

$$\varpi_{a_i} S_i \mathbf{U}_{i_t} \;\leftarrow \varpi S_i \mathbf{B}'_i \tag{5.15}$$

which is equivalent to the finite volume treatment of Eqn. 5.6. A modification of this distribution is to send contributions weighted by the averaged values,

$$\varpi_{a_i} S_i \mathbf{U}_{i_t} \;\leftarrow \varpi \frac{S_\mathsf{T}}{3} \tilde{\mathbf{B}}_\mathsf{T} + COE \tag{5.16}$$

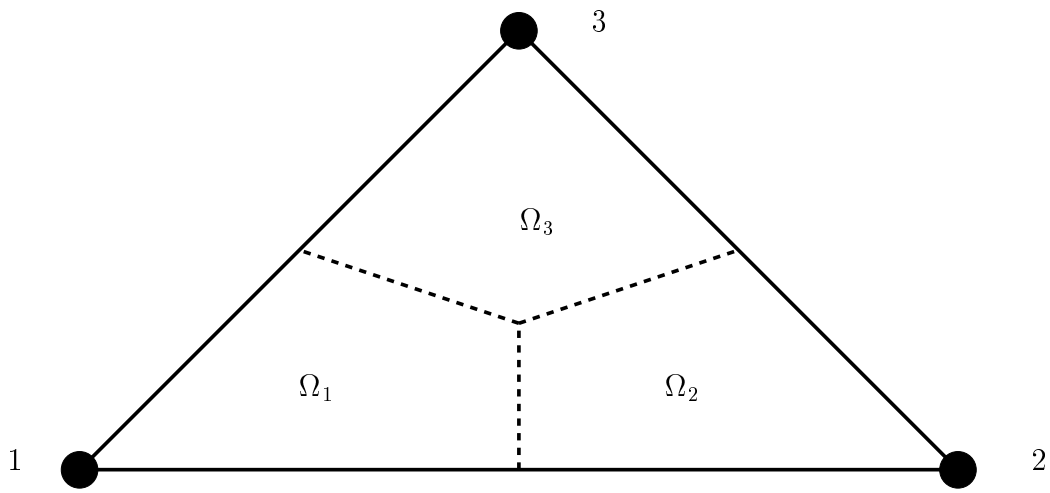


Figure 5.1: Subdivision of triangular element into three quadrilateral integration areas. Dashed lines are the median-dual mesh.

A more rigorous treatment integrates the source term analytically, based on a linear variation of the parameter vector. The only non-zero inviscid source term is,

$$B_3' = P = \frac{\gamma - 1}{\gamma}\left(Z_1 Z_4 - \frac{Z_2^2 + Z_3^2}{2}\right) \tag{5.17}$$

The integration over the triangular element is divided into thirds along the median-dual boundaries, as in Figure 5.1, so that,

$$\Omega = \Omega_1 + \Omega_2 + \Omega_3 \tag{5.18}$$

The subintegrals are then distributed to the nearest node. Notice that the subdivided integration elements, Ω_{1-3}, are quadrilaterals, whereas the original element was a triangle. The distribution formula is thus,

$$\varpi_{a_i} S_i \mathbf{U}_{i_t} \leftarrow \varpi \int_{\Omega_i} \mathbf{B}' \, d\Omega_i + COE \tag{5.19}$$

The integration of the source term over Ω_i is expanded in detail in appendix C.1.1.

Integration of the inviscid flux is performed as,

$$\int_\Omega \vec{\nabla} \cdot \left(\varpi_a \vec{\mathbf{F}}^i\right) \, d\Omega = \int_\Omega \varpi_a \vec{\nabla} \cdot \vec{\mathbf{F}}^i \, d\Omega + \varpi \int_\Omega \mathbf{F}^{i\mathcal{Y}} \, d\Omega \tag{5.20}$$

The y-component of the flux function can be written in terms of the parameter vector as,

$$\mathbf{F}^{iy} = \begin{bmatrix} Z_1 Z_3 \\ Z_2 Z_3 \\ Z_3^2 + \frac{\gamma-1}{\gamma}\left(Z_1 Z_4 - \frac{Z_2^2+Z_3^2}{2}\right) \\ Z_3 Z_4 \end{bmatrix} \tag{5.21}$$

A linear variation of the parameter vector over a triangular element can be represented as,

$$\mathbf{Z}(x,y) = \frac{1}{2S_\mathsf{T}}\epsilon_{ijk}\mathbf{Z}_j\left[(x-x_i)(y_k-y_i)+(y-y_i)(x_i-x_k)\right] \tag{5.22}$$

where ϵ_{ijk} is the cyclic-permutation summation operator. The linear variation can also be written in the element-local (ξ,η) coordinates, referring to Figure 3.3, as,

$$\begin{aligned}\mathbf{Z}(\xi,\eta) &= \mathbf{Z}_1 + \frac{1}{\ell_3}(\mathbf{Z}_2-\mathbf{Z}_1)\xi + \frac{1}{\ell_1}(\mathbf{Z}_3-\mathbf{Z}_2)\eta \\ &= \mathbf{Z}_1 + \frac{1}{\ell_3}\Delta_\xi\mathbf{Z}\,\xi + \frac{1}{\ell_1}\Delta_\eta\mathbf{Z}\,\eta\end{aligned} \tag{5.23}$$

The domain is on $0 \le \eta \le \frac{\ell_1}{\ell_3}\xi$ and $0 \le \xi \le \ell_3$. The Cartesian coordinates map similarly,

$$x(\xi,\eta) = x_1 + \frac{1}{\ell_3}\Delta_\xi x\,\xi + \frac{1}{\ell_1}\Delta_\eta x\,\eta \tag{5.24}$$

$$y(\xi,\eta) = y_1 + \frac{1}{\ell_3}\Delta_\xi y\,\xi + \frac{1}{\ell_1}\Delta_\eta y\,\eta \tag{5.25}$$

Some general integration rules can be developed for linear variations over the triangular elements:

$$\int_\Omega d\Omega = S_\mathsf{T} \tag{5.26}$$

$$\int_\Omega x\,d\Omega = S_\mathsf{T}\bar{x} \tag{5.27}$$

$$\int_\Omega xy\,d\Omega = S_\mathsf{T}\bar{x}\bar{y} - \frac{S_\mathsf{T}}{4}\left(\bar{x}\bar{y} - \frac{1}{3}\sum_{j=1}^{3} x_j y_j\right) \tag{5.28}$$

The cell-averaged value is,

$$\bar{x} = \frac{x_1 + x_2 + x_3}{3} = \frac{1}{3}\sum_{j=1}^{3} x_j \tag{5.29}$$

The last term of Eqn. 5.20 is distributed to the nodes in a manner similar to the source term,

$$\varpi_{a_i} S_i \mathbf{U}_{i_t} \leftarrow -\varpi \int_{\Omega_i} \mathbf{F}^{i^y}\, d\Omega_i + COE \tag{5.30}$$

The integration rules from Eqns. C.16–C.20 are used to evaluate the integral, leading to,

$$\int_{\Omega_i} Z_1 Z_3\, d\Omega_i = \frac{S_\mathsf{T}}{144}\left[14\bar{Z}_1\bar{Z}_3 + 11(Z_{1_i}\bar{Z}_3 + \bar{Z}_1 Z_{3_i}) + 9 Z_{1_i} Z_{3_i} + \sum_{j=1}^{3} Z_{1_j} Z_{3_j}\right] \tag{5.31}$$

for the continuity equation. The integrals for the other governing equations follow directly from Eqn. 5.31.

The remaining term to evaluate in Eqn. 5.20 is the inviscid fluctuation,

$$\begin{aligned}
\boldsymbol{\phi} &= -\int_{\Omega} \varpi_a \vec{\nabla}\cdot\vec{\mathbf{F}}^i\, d\Omega \\
&= -\frac{\ell_1\ell_3}{2S_\mathsf{T}}\int_{\Omega} \varpi_a \left(\hat{n}_1\cdot\vec{\mathbf{F}}^i_\xi - \hat{n}_3\cdot\vec{\mathbf{F}}^i_\eta\right)\, d\Omega \\
&= -\frac{1}{2S_\mathsf{T}}\int_{\Omega} \varpi_a \left(\ell_1\hat{n}_1\cdot\vec{\mathbf{F}}_Z\Delta_\xi\mathbf{Z} - \ell_3\hat{n}_3\cdot\vec{\mathbf{F}}_Z\Delta_\eta\mathbf{Z}\right)\, d\Omega
\end{aligned} \tag{5.32}$$

where $d\vec{\mathbf{F}}^i = \vec{\mathbf{F}}_Z\, d\mathbf{Z}$ and,

$$\vec{\mathbf{F}}^{\mathsf{x}}_Z = \begin{bmatrix} Z_2 & Z_1 & 0 & 0 \\ \frac{\gamma-1}{\gamma}Z_4 & \frac{\gamma+1}{\gamma}Z_2 & -\frac{\gamma-1}{\gamma}Z_3 & \frac{\gamma-1}{\gamma}Z_1 \\ 0 & Z_3 & Z_2 & 0 \\ 0 & Z_4 & 0 & Z_2 \end{bmatrix} \tag{5.33}$$

$$\vec{\mathbf{F}}^{\mathsf{y}}_Z = \begin{bmatrix} Z_3 & 0 & Z_1 & 0 \\ 0 & Z_3 & Z_2 & 0 \\ \frac{\gamma-1}{\gamma}Z_4 & -\frac{\gamma-1}{\gamma}Z_2 & \frac{\gamma+1}{\gamma}Z_3 & \frac{\gamma-1}{\gamma}Z_1 \\ 0 & 0 & Z_4 & Z_3 \end{bmatrix} \tag{5.34}$$

The integration rule Eqn. 5.28 allows for the direct evaluation of Eqn. 5.32 as,

$$\begin{aligned}\boldsymbol{\phi} = &-\frac{1}{2}\overline{\varpi}_a \ell_1 \hat{n}_1 \cdot \left[\overline{\vec{\mathbf{F}}}_Z - \frac{1}{4}\left(\overline{\vec{\mathbf{F}}}_Z - \frac{1}{3}\sum_{j=1}^{3}\frac{\varpi_{a_j}}{\overline{\varpi}_a}\vec{\mathbf{F}}_{Z_j}\right)\right]\Delta_\xi \mathbf{Z} \\ &+ \frac{1}{2}\overline{\varpi}_a \ell_3 \hat{n}_3 \cdot \left[\overline{\vec{\mathbf{F}}}_Z - \frac{1}{4}\left(\overline{\vec{\mathbf{F}}}_Z - \frac{1}{3}\sum_{j=1}^{3}\frac{\varpi_{a_j}}{\overline{\varpi}_a}\vec{\mathbf{F}}_{Z_j}\right)\right]\Delta_\eta \mathbf{Z}\end{aligned} \tag{5.35}$$

Noting that $\tilde{\vec{\mathbf{A}}} = \overline{\tilde{\mathbf{F}}}_{\mathbf{Z}}\,\mathbf{Z}_{\tilde{\mathbf{U}}}$ and,

$$\mathbf{Z}_U = \frac{1}{2\sqrt{\rho}}\begin{bmatrix} 1 & 0 & 0 & 0 \\ -u & 2 & 0 & 0 \\ -v & 0 & 2 & 0 \\ -H + (\gamma-1)\left(u^2+v^2\right) & -2(\gamma-1)u & -2(\gamma-1)v & 2\gamma \end{bmatrix} \tag{5.36}$$

with the tilde-averaged quantities defined as,

$$\tilde{\mathbf{U}} = \mathbf{U}(\bar{\mathbf{Z}}) \qquad \widetilde{\Delta_\xi \mathbf{U}} = \mathbf{U}_{\bar{Z}}\,\Delta_\xi \mathbf{Z} \qquad \widetilde{\Delta_\eta \mathbf{U}} = \mathbf{U}_{\bar{Z}}\,\Delta_\eta \mathbf{Z} \tag{5.37}$$

leads to the fluctuation expressed as,

$$\begin{aligned}\boldsymbol{\phi} = &-\frac{1}{2}\overline{\varpi}_a \ell_1 \hat{n}_1 \cdot \left[\tilde{\vec{\mathbf{A}}} - \frac{\mathbf{1}}{\mathbf{4}}\left(\tilde{\vec{\mathbf{A}}} - \frac{\mathbf{1}}{\mathbf{3}}\sum_{\mathbf{j=1}}^{\mathbf{3}}\frac{\varpi_{\mathbf{a_j}}}{\overline{\varpi}_{\mathbf{a}}}\tilde{\mathbf{A}}_{\mathbf{j}}\right)\right]\widetilde{\Delta_\xi \mathbf{U}} \\ &+ \frac{1}{2}\overline{\varpi}_a \ell_3 \hat{n}_3 \cdot \left[\tilde{\vec{\mathbf{A}}} - \frac{\mathbf{1}}{\mathbf{4}}\left(\tilde{\vec{\mathbf{A}}} - \frac{\mathbf{1}}{\mathbf{3}}\sum_{\mathbf{j=1}}^{\mathbf{3}}\frac{\varpi_{\mathbf{a_j}}}{\overline{\varpi}_{\mathbf{a}}}\tilde{\mathbf{A}}_{\mathbf{j}}\right)\right]\widetilde{\Delta_\eta \mathbf{U}} \\ = &\,\boldsymbol{\phi}^\xi + \boldsymbol{\phi}^\eta\end{aligned} \tag{5.38}$$

Equation 5.38 includes the approximation,

$$\vec{\mathbf{A}}_j \simeq \vec{\mathbf{F}}_{Z_j}\mathbf{Z}_{\tilde{U}} \tag{5.39}$$

A transformation of variables can be made to the auxiliary variables so that,

$$\boldsymbol{\phi} = \tilde{\mathbf{U}}_W \breve{\boldsymbol{\phi}} \tag{5.40}$$

with $\mathbf{U}_W$ given in Eqn. B.61. The similarity transformations developed in section B.3 lead to,

$$\begin{aligned}
\breve{\boldsymbol{\phi}} &= -\frac{1}{2}\overline{\varpi}_a \ell_1 \hat{n}_1 \cdot \left[\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{4}\left(\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{3}\sum_{j=1}^{3}\frac{\varpi_{a_j}}{\overline{\varpi}_a}\vec{\boldsymbol{\mathcal{A}}}_j\right)\right]\widetilde{\Delta_\xi \mathbf{W}} \\
&\quad + \frac{1}{2}\overline{\varpi}_a \ell_3 \hat{n}_3 \cdot \left[\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{4}\left(\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{3}\sum_{j=1}^{3}\frac{\varpi_{a_j}}{\overline{\varpi}_a}\vec{\boldsymbol{\mathcal{A}}}_j\right)\right]\widetilde{\Delta_\eta \mathbf{W}} \\
&= -\overline{\varpi}_a\left(\boldsymbol{\alpha}\,\widetilde{\Delta_\xi \mathbf{W}} + \boldsymbol{\beta}\,\widetilde{\Delta_\eta \mathbf{W}}\right) \\
&= \breve{\boldsymbol{\phi}}^{\xi} + \breve{\boldsymbol{\phi}}^{\eta}
\end{aligned} \tag{5.41}$$

with $\vec{\boldsymbol{\mathcal{A}}} = \mathbf{W}_U \vec{\mathbf{A}} \mathbf{U}_W$ and,

$$\widetilde{\Delta_\xi \mathbf{W}} = \mathbf{W}_{\tilde{U}}\,\widetilde{\Delta_\xi \mathbf{U}} = \mathbf{W}_{\bar{Z}}\,\Delta_\xi \mathbf{Z} \qquad \widetilde{\Delta_\eta \mathbf{W}} = \mathbf{W}_{\tilde{U}}\,\widetilde{\Delta_\eta \mathbf{U}} = \mathbf{W}_{\bar{Z}}\,\Delta_\eta \mathbf{Z} \tag{5.42}$$

where $\mathbf{W}_U$ is given in Eqn. B.62 and,

$$\begin{aligned}
\mathbf{W}_Z &= \sqrt{\rho}\begin{bmatrix} 2-\frac{H}{\gamma h} & \frac{u}{\gamma h} & \frac{v}{\gamma h} & -\frac{1}{\gamma h} \\ -u & 1 & 0 & 0 \\ -v & 0 & 1 & 0 \\ \frac{\gamma-1}{\gamma}H & -\frac{\gamma-1}{\gamma}u & -\frac{\gamma-1}{\gamma}v & \frac{\gamma-1}{\gamma} \end{bmatrix} \\
&= \begin{bmatrix} 2Z_1 - \frac{\gamma-1}{\gamma a^2}Z_4 & \frac{\gamma-1}{\gamma a^2}Z_2 & \frac{\gamma-1}{\gamma a^2}Z_3 & -\frac{\gamma-1}{\gamma a^2}Z_1 \\ -Z_2 & Z_1 & 0 & 0 \\ -Z_3 & 0 & Z_1 & 0 \\ \frac{\gamma-1}{\gamma}Z_4 & -\frac{\gamma-1}{\gamma}Z_2 & -\frac{\gamma-1}{\gamma}Z_3 & \frac{\gamma-1}{\gamma}Z_1 \end{bmatrix}
\end{aligned} \tag{5.43}$$

using the relation,

$$h = c_p T = \frac{a^2}{\gamma - 1} \tag{5.44}$$

The projection of $\vec{\boldsymbol{\mathcal{A}}}$ has the simple form, derived as Eqn. B.54,

$$\hat{n}\cdot\vec{\boldsymbol{\mathcal{A}}} = \begin{bmatrix} \mathcal{V} & 0 & 0 \\ 0 & \mathcal{V}I & \hat{n}^{\mathsf{T}} \\ 0 & a^2\hat{n} & \mathcal{V} \end{bmatrix} = \begin{bmatrix} \mathcal{V} & 0 & 0 & 0 \\ 0 & \mathcal{V} & 0 & n^{\mathsf{x}} \\ 0 & 0 & \mathcal{V} & n^{\mathsf{y}} \\ 0 & a^2 n^{\mathsf{x}} & a^2 n^{\mathsf{y}} & \mathcal{V} \end{bmatrix} \tag{5.45}$$

where the projected velocity is $\mathcal{V} = \hat{n}\cdot\vec{V}$. The generalized advection speeds are,

$$\boldsymbol{\alpha} = \frac{\ell_1}{2}\hat{n}_1\cdot\left[\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{4}\left(\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{3}\sum_{j=1}^{3}\frac{\varpi_{a_j}}{\overline{\varpi}_a}\vec{\boldsymbol{\mathcal{A}}}_j\right)\right] \tag{5.46}$$

$$\boldsymbol{\beta} = -\frac{\ell_3}{2}\hat{n}_3\cdot\left[\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{4}\left(\tilde{\vec{\boldsymbol{\mathcal{A}}}} - \frac{1}{3}\sum_{j=1}^{3}\frac{\varpi_{a_j}}{\overline{\varpi}_a}\vec{\boldsymbol{\mathcal{A}}}_j\right)\right] \tag{5.47}$$

incorporating the approximation,

$$\vec{\boldsymbol{\mathcal{A}}}_j = \mathbf{W}_{\tilde{U}}\vec{\mathbf{A}}_j\tilde{\mathbf{U}}_W \tag{5.48}$$

Linearity preservation for second-order spatial accuracy is obtained by limiting the fluctuations componentwise,

$$\breve{\phi}_j^{*\xi} = \breve{\phi}_j^{\xi} + \breve{\phi}_j^{\eta}\,\psi(Q_j) = \breve{\phi}_j^{\xi}\left(1 - \psi\left(\frac{1}{Q_j}\right)\right) \tag{5.49}$$

$$\breve{\phi}_j^{*\eta} = \breve{\phi}_j^{\eta} - \breve{\phi}_j^{\eta}\,\psi(Q_j) = \breve{\phi}_j^{\eta}\left(1 - \psi\left(Q_j\right)\right) \tag{5.50}$$

where the second equalities hold for symmetric limiters. The limiting ratio is,

$$Q_j = -\frac{\breve{\phi}_j^{\xi}}{\breve{\phi}_j^{\eta}} \tag{5.51}$$

In vector form, Eqns. 5.49 and 5.50 can be written,

$$\breve{\boldsymbol{\phi}}^{*\xi} = \mathbf{D}^{\xi}\breve{\boldsymbol{\phi}}^{\xi} \tag{5.52}$$

$$\breve{\boldsymbol{\phi}}^{*\eta} = \mathbf{D}^{\eta}\breve{\boldsymbol{\phi}}^{\eta} \tag{5.53}$$

with

$$\mathbf{D}^{\xi} = \mathrm{diag}\left(1 - \psi\left(\frac{1}{Q_j}\right)\right) \qquad \mathbf{D}^{\eta} = \mathrm{diag}\left(1 - \psi\left(Q_j\right)\right) \tag{5.54}$$

Upwinding is achieved through the introduction of artificial dissipation

$$\breve{\boldsymbol{\phi}}^{\prime\xi} = \mathrm{sign}(\boldsymbol{\alpha})\breve{\boldsymbol{\phi}}^{*\xi} = -\overline{\varpi}_a\mathbf{M}_{\alpha}\mathbf{D}^{\xi}\mathbf{M}_{\alpha}^{-1}|\boldsymbol{\alpha}|\,\widetilde{\Delta_{\xi}\mathbf{W}} \tag{5.55}$$

$$\breve{\boldsymbol{\phi}}^{\prime\eta} = \mathrm{sign}(\boldsymbol{\beta})\breve{\boldsymbol{\phi}}^{*\eta} = -\overline{\varpi}_a\mathbf{M}_{\beta}\mathbf{D}^{\eta}\mathbf{M}_{\beta}^{-1}|\boldsymbol{\beta}|\,\widetilde{\Delta_{\eta}\mathbf{W}} \tag{5.56}$$

where $\mathbf{M}_\alpha = \text{sign}(\boldsymbol{\alpha})$ and $\mathbf{M}_\beta = \text{sign}(\boldsymbol{\beta})$. The absolute values of the generalized advection speeds are developed using the following decomposition, which is exact for the two-dimensional equations but approximate for the axisymmetric form,

$$\boldsymbol{\alpha} = \frac{\ell_1}{2}\hat{n}_1 \cdot \tilde{\vec{\boldsymbol{\mathcal{A}}}} = \frac{\ell_1}{2}\tilde{\boldsymbol{\mathcal{X}}}\tilde{\boldsymbol{\Lambda}}\tilde{\boldsymbol{\mathcal{X}}}^{-1} \tag{5.57}$$

where $\boldsymbol{\Lambda}$ and $\boldsymbol{\mathcal{X}}$ are defined in Eqns. B.38 and B.57, respectively. The absolute value is then defined as,

$$|\boldsymbol{\alpha}| = \frac{\ell_1}{2}\tilde{\boldsymbol{\mathcal{X}}}|\tilde{\boldsymbol{\Lambda}}|\tilde{\boldsymbol{\mathcal{X}}}^{-1} \tag{5.58}$$

Expressions for the sign of the generalized wavespeeds are developed from,

$$|\boldsymbol{\alpha}| = \mathbf{M}_\alpha \boldsymbol{\alpha} \tag{5.59}$$

$$\mathbf{M}_\alpha = \tilde{\boldsymbol{\mathcal{X}}}|\tilde{\boldsymbol{\Lambda}}|\tilde{\boldsymbol{\Lambda}}^{-1}\tilde{\boldsymbol{\mathcal{X}}}^{-1} \tag{5.60}$$

leading to,

$$\mathbf{M}_\alpha = \text{sign}(\tilde{\mathcal{V}}_\alpha)I \qquad \text{if } |\tilde{\mathcal{V}}_\alpha| > \tilde{a} \tag{5.61}$$

$$\mathbf{M}_\alpha = \begin{bmatrix} \text{sign}(\tilde{\mathcal{V}}_\alpha) & 0 & 0 & 0 \\ 0 & n_1^{\mathsf{y}^2}\text{sign}(\tilde{\mathcal{V}}_\alpha) & -n_1^{\mathsf{x}}n_1^{\mathsf{y}}\text{sign}(\tilde{\mathcal{V}}_\alpha) & \frac{n_1^{\mathsf{x}}}{\tilde{a}} \\ 0 & -n_1^{\mathsf{x}}n_1^{\mathsf{y}}\text{sign}(\tilde{\mathcal{V}}_\alpha) & n_1^{\mathsf{x}^2}\text{sign}(\tilde{\mathcal{V}}_\alpha) & \frac{n_1^{\mathsf{y}}}{\tilde{a}} \\ 0 & \tilde{a}n_1^{\mathsf{x}} & \tilde{a}n_1^{\mathsf{y}} & 0 \end{bmatrix} \qquad \text{if } |\tilde{\mathcal{V}}_\alpha| < \tilde{a} \tag{5.62}$$

where $\mathcal{V}_\alpha = \hat{n}_1 \cdot \vec{V}$. Similarly, defining $|\boldsymbol{\beta}|$ as,

$$|\boldsymbol{\beta}| = -\frac{\ell_3}{2}\tilde{\boldsymbol{\mathcal{X}}}|\tilde{\boldsymbol{\Lambda}}|\tilde{\boldsymbol{\mathcal{X}}}^{-1} \tag{5.63}$$

leads to,

$$\mathbf{M}_\beta = \text{sign}(\tilde{\mathcal{V}}_\beta)I \qquad \text{if } |\tilde{\mathcal{V}}_\beta| > \tilde{a} \tag{5.64}$$

$$\mathbf{M}_\beta = \begin{bmatrix} \text{sign}(\tilde{\mathcal{V}}_\beta) & 0 & 0 & 0 \\ 0 & n_3^{\mathsf{y}^2}\text{sign}(\tilde{\mathcal{V}}_\beta) & -n_3^{\mathsf{x}}n_3^{\mathsf{y}}\text{sign}(\tilde{\mathcal{V}}_\beta) & \frac{n_3^{\mathsf{x}}}{\tilde{a}} \\ 0 & -n_3^{\mathsf{x}}n_3^{\mathsf{y}}\text{sign}(\tilde{\mathcal{V}}_\beta) & n_3^{\mathsf{x}^2}\text{sign}(\tilde{\mathcal{V}}_\beta) & \frac{n_3^{\mathsf{y}}}{\tilde{a}} \\ 0 & \tilde{a}n_3^{\mathsf{x}} & \tilde{a}n_3^{\mathsf{y}} & 0 \end{bmatrix} \qquad \text{if } |\tilde{\mathcal{V}}_\beta| < \tilde{a} \tag{5.65}$$

where $\mathcal{V}_\beta = \hat{n}_3 \cdot \vec{V}$. $\mathbf{M}_\alpha$ and $\mathbf{M}_\beta$ have the property,

$$\mathbf{M}_\alpha^{-1} = \mathbf{M}_\alpha \qquad \mathbf{M}_\beta^{-1} = \mathbf{M}_\beta \tag{5.66}$$

Eigenvalue limiting for the suppression of expansion shocks can be introduced into the expression for $|\boldsymbol{\alpha}|$, Eqn. 5.58, by limiting $|\tilde{\Lambda}_1| = |\tilde{\Lambda}_2| = |\tilde{\mathcal{V}}|$, $|\tilde{\Lambda}_3| = |\tilde{\mathcal{V}} + \tilde{a}|$, and $|\tilde{\Lambda}_4| = |\tilde{\mathcal{V}} - \tilde{a}|$ in the manner of Eqns. 2.46 and 4.60. If the limited eigenvalue is expressed as,

$$|\Lambda|_{lim} = |\Lambda| + \Delta_\Lambda \tag{5.67}$$

then the additional artificial dissipation for eigenvalue limiting in the ξ direction to be added to Eqn. 5.55 is, with $\Delta^+ = \frac{1}{2}(\Delta_{\Lambda_3} + \Delta_{\Lambda_4})$ and $\Delta^- = \frac{1}{2}(\Delta_{\Lambda_3} - \Delta_{\Lambda_4})$,

$$-\bar{\varpi}_a \frac{\ell_1}{2} \begin{bmatrix} \Delta_{\Lambda_1} & 0 & 0 & 0 \\ 0 & n_1^{\mathsf{x}^2}\Delta^+ + n_1^{\mathsf{y}^2}\Delta_{\Lambda_2} & n_1^{\mathsf{x}} n_1^{\mathsf{y}}(\Delta^+ - \Delta_{\Lambda_2}) & \frac{n_1^{\mathsf{x}}}{a}\Delta^- \\ 0 & n_1^{\mathsf{x}} n_1^{\mathsf{y}}(\Delta^+ - \Delta_{\Lambda_2}) & n_1^{\mathsf{x}^2}\Delta_{\Lambda_2} + n_1^{\mathsf{y}^2}\Delta^+ & \frac{n_1^{\mathsf{y}}}{a}\Delta^- \\ 0 & a n_1^{\mathsf{x}}\Delta^- & a n_1^{\mathsf{y}}\Delta^- & \Delta^+ \end{bmatrix} \widetilde{\Delta_\xi W} \tag{5.68}$$

while the eigenvalue limiting in the η direction takes the form,

$$+\bar{\varpi}_a \frac{\ell_3}{2} \begin{bmatrix} \Delta_{\Lambda_1} & 0 & 0 & 0 \\ 0 & n_3^{\mathsf{x}^2}\Delta^+ + n_3^{\mathsf{y}^2}\Delta_{\Lambda_2} & n_3^{\mathsf{x}} n_3^{\mathsf{y}}(\Delta^+ - \Delta_{\Lambda_2}) & \frac{n_3^{\mathsf{x}}}{a}\Delta^- \\ 0 & n_3^{\mathsf{x}} n_3^{\mathsf{y}}(\Delta^+ - \Delta_{\Lambda_2}) & n_3^{\mathsf{x}^2}\Delta_{\Lambda_2} + n_3^{\mathsf{y}^2}\Delta^+ & \frac{n_3^{\mathsf{y}}}{a}\Delta^- \\ 0 & a n_3^{\mathsf{x}}\Delta^- & a n_3^{\mathsf{y}}\Delta^- & \Delta^+ \end{bmatrix} \widetilde{\Delta_\eta W} \tag{5.69}$$

The fluctuation is distributed to the nodes using a simple extension of the scalar distribution, Eqns. 3.30 and 3.31,

$$\begin{aligned} S_1 \mathbf{U}_{1_t} &\leftarrow \frac{1}{2}(\boldsymbol{\phi}^{*^\xi} - \boldsymbol{\phi}'^{\xi}) + COE \\ S_2 \mathbf{U}_{2_t} &\leftarrow \frac{1}{2}(\boldsymbol{\phi}^{*^\xi} + \boldsymbol{\phi}'^{\xi}) + \frac{1}{2}(\boldsymbol{\phi}^{*^\eta} + \boldsymbol{\phi}'^{\eta}) + COE \\ S_3 \mathbf{U}_{3_t} &\leftarrow \frac{1}{2}(\boldsymbol{\phi}^{*^\eta} - \boldsymbol{\phi}'^{\eta}) + COE \end{aligned} \tag{5.70}$$

or in a more compact form,

$$S_i \mathbf{U}_{i_t} \leftarrow \frac{1}{4}\left[i(3-i)(\boldsymbol{\phi}^{*^\xi} + (-1)^i \boldsymbol{\phi}'^{\xi}) + (-4 + 5i - i^2)(\boldsymbol{\phi}^{*^\eta} + (-1)^i \boldsymbol{\phi}'^{\eta}) \right] + COE \qquad i = 1,2,3 \tag{5.71}$$

where Eqn. 5.40 is used to define,

$$\boldsymbol{\phi}^{*^{\xi}} = \tilde{\mathbf{U}}_W \breve{\boldsymbol{\phi}}^{*^{\xi}}, \qquad \boldsymbol{\phi}'^{\xi} = \tilde{\mathbf{U}}_W \breve{\boldsymbol{\phi}}'^{\xi} \tag{5.72}$$

$$\boldsymbol{\phi}^{*^{\eta}} = \tilde{\mathbf{U}}_W \breve{\boldsymbol{\phi}}^{*^{\eta}}, \qquad \boldsymbol{\phi}'^{\eta} = \tilde{\mathbf{U}}_W \breve{\boldsymbol{\phi}}'^{\eta} \tag{5.73}$$

At the 15th AIAA Computational Fluid Dynamics Conference Prof. Philip Roe[105] suggested including the radial distance, ϖ_a, within the definition of the parameter vector. Further reflection reveals that $\sqrt{\varpi_a}$ is the proper term to consider. This approach has the potential to avoid some of the explicit integration by parts in this section and allow for cleaner expressions for many of the combined axisymmetric/two-dimensional relations, such as Eqn. 5.35. However, the flux function $\vec{\mathbf{F}}$ is no longer just a function of state but also of position, so that the gradient can no longer be expanded as $\vec{\nabla}\cdot\vec{\mathbf{F}} = \vec{\mathbf{F}}_Z \cdot \vec{\nabla}\mathbf{Z}$, which could lead to complicated expressions again. Also, the consistent incorporation of this redefinition of the parameter vector into the viscous discretizations is not straightforward due to the difficulty in expressing all terms as explicit quadratic products of the parameter vector.

5.3 Viscous Formulations

The complete set of governing equations for the motion of a two-dimensional or axisymmetric viscous fluid are presented in appendix B as Eqn. B.9,

$$\varpi_a \mathbf{U}_t + \vec{\nabla}\cdot(\varpi_a \vec{\mathbf{F}}^i) = \vec{\nabla}\cdot(\varpi_a \vec{\mathbf{F}}^v) + \varpi \mathbf{B}'^i - \varpi \mathbf{B}'^v \tag{5.74}$$

Integrations of the conserved variables, inviscid flux, and inviscid axisymmetric source proceed as developed in section 5.2.

Integrating the viscous flux over a triangular element leads to the viscous fluctuation (see Eqn. 3.47),

$$\boldsymbol{\phi}^v = \int_\Omega \vec{\nabla}\cdot(\varpi_a \vec{\mathbf{F}}^v)\, d\Omega \tag{5.75}$$

The nodal distributions are developed in a finite element sense by integrating by parts as the extension of Eqn. 3.49,

$$\boldsymbol{\phi}_i^v = \oint_\Gamma \upsilon_i \varpi_a \vec{\mathbf{F}}^v \cdot \hat{n}\, d\Gamma - \int_\Omega \varpi_a \vec{\mathbf{F}}^v \cdot \vec{\nabla}\upsilon_i\, d\Omega \tag{5.76}$$

For interior nodes the boundary integral in Eqn. 5.76 will sum to zero and the volume integral is integrated analytically for a linear variation of the parameter vector,

$$\vec{\nabla}\mathbf{Z} = -\frac{1}{2S_{\mathsf{T}}}\sum_{j=1}^{3}\mathbf{Z}_j\ell_j\hat{n}_j \qquad \vec{\nabla}\upsilon_i = -\frac{\ell_i\hat{n}_i}{2S_{\mathsf{T}}} \tag{5.77}$$

$$\boldsymbol{\phi}_i = \frac{\ell_i}{2S_{\mathsf{T}}}\int_{\Omega}\varpi_a\vec{\mathbf{F}}^v\cdot\hat{n}_i\,d\Omega = \frac{\bar{\varpi}_a\ell_i}{2}\,\tilde{\vec{\mathbf{F}}}^v\cdot\hat{\mathbf{n}}_{\mathbf{i}} \tag{5.78}$$

The viscous flux is,

$$\vec{\mathbf{F}}^v = \frac{1}{R_{e_\infty}}\begin{pmatrix} 0 \\ \boldsymbol{\tau} \\ \kappa\vec{\nabla}T + \vec{V}\boldsymbol{\tau} \end{pmatrix} \tag{5.79}$$

with the shear-stress tensor defined,

$$\boldsymbol{\tau} = \mu\left[\vec{\nabla}^T\vec{V} + \left(\vec{\nabla}^T\vec{V}\right)^T - \frac{2}{3}\vec{\nabla}\cdot\vec{V}I\right] \tag{5.80}$$

Struijs[34] *et al.* have shown that derivatives of primitive variables can be consistently represented in terms of the parameter vector gradients as,

$$\widetilde{\vec{\nabla}\mathbf{V}} = \mathbf{V}_{\bar{Z}}\,\vec{\nabla}\mathbf{Z} \tag{5.81}$$

where,

$$\mathbf{V}_Z = \begin{bmatrix} 2Z_1 & 0 & 0 & 0 \\ -\frac{Z_2}{Z_1^2} & \frac{1}{Z_1} & 0 & 0 \\ -\frac{Z_3}{Z_1^2} & 0 & \frac{1}{Z_1} & 0 \\ \frac{\gamma-1}{\gamma}Z_4 & -\frac{\gamma-1}{\gamma}Z_2 & -\frac{\gamma-1}{\gamma}Z_3 & \frac{\gamma-1}{\gamma}Z_1 \end{bmatrix} \tag{5.82}$$

and,

$$\tilde{\mathbf{V}} = \mathbf{V}(\bar{\mathbf{Z}}) = \begin{pmatrix} \bar{Z}_1^2 \\ \frac{\bar{Z}_2}{\bar{Z}_1} \\ \frac{\bar{Z}_3}{\bar{Z}_1} \\ \frac{\gamma-1}{\gamma}\left[\bar{Z}_1\bar{Z}_4 - \frac{1}{2}\left(\bar{Z}_2^2 + \bar{Z}_3^2\right)\right] \end{pmatrix} \tag{5.83}$$

Further, the consistent temperature gradient is,

$$\widetilde{\vec{\nabla}T} = \frac{\widetilde{\vec{\nabla}P}}{\tilde{\rho}R} - \frac{\tilde{P}\widetilde{\vec{\nabla}\rho}}{\tilde{\rho}^2 R} = \frac{1}{R\tilde{\rho}^2}\left(\tilde{\rho}\widetilde{\vec{\nabla}P} - \tilde{P}\widetilde{\vec{\nabla}\rho}\right) \tag{5.84}$$

The viscous fluctuation is then distributed to the nodes as in Eqn. 3.52,

$$S_i \mathbf{U}_{i_t} \leftarrow \boldsymbol{\phi}_i^v + COE \tag{5.85}$$

An alternate approach to integrating the viscous flux is obtained by using the divergence theorem,

$$\int_{\Omega_i} \vec{\nabla}\cdot(\varpi_a \vec{\mathbf{F}}^v)\, d\Omega = \oint_{\Gamma_i} \varpi_a \vec{\mathbf{F}}^v \cdot \hat{n}\, d\Gamma \tag{5.86}$$

where Ω_i is the generalized control volume containing node i, with two-dimensional area equal to S_i, and Γ_i is the boundary of Ω_i.

Haselbacher[106] *et al.* have recently presented an approximate treatment for integrating Eqn. 5.86 on two-dimensional unstructured grids, which they relate to the thin-layer approximation of the Navier-Stokes equations presented by Gnoffo[16] for structured grids. The method preserves positivity for the Laplacian and is transparent to grid topology.

The development begins with the expression,

$$\vec{\mathbf{F}}^v \cdot \hat{n} = \frac{1}{R_{e_\infty}} \begin{pmatrix} 0 \\ \mu\left[\vec{\nabla}u\cdot\hat{n} + \frac{1}{3}\vec{\nabla}\cdot\vec{V}n^{\mathsf{x}} - \vec{\nabla}v\cdot\hat{t}\right] \\ \mu\left[\vec{\nabla}v\cdot\hat{n} + \frac{1}{3}\vec{\nabla}\cdot\vec{V}n^{\mathsf{y}} + \vec{\nabla}u\cdot\hat{t}\right] \\ \kappa\vec{\nabla}T\cdot\hat{n} + \mu\left[\frac{1}{3}(\vec{\nabla}\cdot\vec{V})(\vec{V}\cdot\hat{n}) + \frac{1}{2}\vec{\nabla}V^2\cdot\hat{n} - u\vec{\nabla}v\cdot\hat{t} + v\vec{\nabla}u\cdot\hat{t}\right] \end{pmatrix} \tag{5.87}$$

where $\hat{t}$ is a tangent vector with components $(-n^{\mathsf{y}}, n^{\mathsf{x}})$. Haselbacher's approximation neglects all tangential terms and approximates $\vec{\nabla}\cdot\vec{V} \simeq n^{\mathsf{x}}\vec{\nabla}u\cdot\hat{n} + n^{\mathsf{y}}\vec{\nabla}v\cdot\hat{n}$. Using the notation $u_n = \vec{\nabla}u\cdot\hat{n}$, etc., and including the axisymmetric terms gives the approximation,

$$\vec{\nabla}\cdot\vec{V} \simeq \vec{V}_n\cdot\hat{n} + \varpi\frac{v}{y} \tag{5.88}$$

leading to,

$$\vec{\mathbf{F}}^v \cdot \hat{n} \simeq \frac{\mu}{R_{e_\infty}} \begin{pmatrix} 0 \\ \vec{V}_n^{\mathsf{T}} + \frac{1}{3}\hat{n}^{\mathsf{T}} \left(\vec{V}_n \cdot \hat{n} + \varpi \frac{v}{y} \right) \\ \frac{\kappa}{\mu} T_n + \vec{V} \cdot \vec{V}_n + \frac{1}{3} \vec{V} \cdot \hat{n} \left(\vec{V}_n \cdot \hat{n} + \varpi \frac{v}{y} \right) \end{pmatrix} \tag{5.89}$$

A further simplification aligns $\hat{n}$ with the nearest mesh edge for faces of Γ on the interior of the domain, so that terms such as u_n reduce simply to the difference in nodal values divided by edge length. Also, ϖ_a is chosen to be the midpoint of the edge.

Including one more approximation, namely replacing the length Γ of the median-dual face with the length of the associated containment-dual face, has the effect of canceling some of the errors for very high-aspect-ratio cells introduced by assuming $\hat{n}$ is edge-aligned. For low-aspect-ratio cells, the containment dual is the same as the median dual and the true $\hat{n}$ is closely aligned with the mesh edge. This implementation is similar to the suggestions of both Barth[22] and Haselbacher[107], yet retains the global median-dual implementation required by a distribution scheme.

The viscous axisymmetric source term has only one non-zero entry,

$$B_3'^v = \frac{2\mu}{3R_{e_\infty}} \left(2\frac{v}{y} - \vec{\nabla} \cdot \vec{V} \right) \tag{5.90}$$

Integrating using the Haselbacher approach leads to,

$$\int_{\Omega_i} B_3'^v \, d\Omega = \frac{4}{3R_{e_\infty}} \int_{\Omega_i} \frac{\mu v}{y} \, d\Omega - \frac{2}{3R_{e_\infty}} \oint_{\Gamma_i} \mu \vec{V} \cdot \hat{n} \, d\Gamma \tag{5.91}$$

Mass lumping to the node for the first term yields,

$$\int_{\Omega_i} \frac{\mu v}{y} \, d\Omega = \mu_i S_i \frac{v_i}{y_i} \tag{5.92}$$

while the second term is evaluated at edge midpoints.

5.4 Boundary Conditions

5.4.1 Weak Formulations

Boundary nodes may be updated either strongly, where the nodal solution values are simply assigned *a priori*, or weakly, where the solution values at the boundary nodes

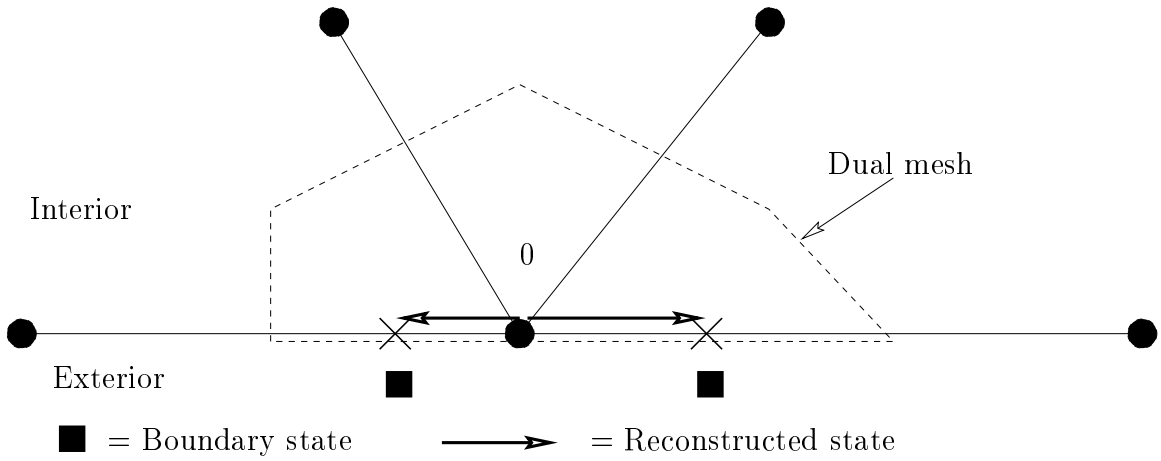


Figure 5.2: Weak implementation of finite volume boundary condition for node 0, imposed by specifying external state. Quadrature points denoted by X's.

are relaxed in time using the same formulations as for interior nodes.

For finite volume, the weak boundary implementation specifies the solution state to the outside of boundary faces, then allows the approximate Riemann solver to construct the appropriate fluxes through the boundary faces. See Figure 5.2 for an illustration of the weak finite volume boundary condition. The solution state to the inside of the boundary face can be set from either a first- or second-order reconstruction from the node. For some cases, second-order reconstruction to boundary faces has led to localized oscillations in the solution convergence at boundary nodes.

Weak formulation of the fluctuation splitting boundary condition is developed using fictitious "ghost" nodes, one for each boundary node, as shown in Figure 5.3. Considering the scalar case, the positioning of a ghost node such that the edge connecting the ghost and boundary nodes is parallel to the advection velocity results in a boundary fluctuation of,

$$\phi_{bc_0} = \frac{1}{2}\ell_{01}\vec{A}\cdot\hat{n}_{01}(U_{f_0} - U_0) \tag{5.93}$$

for node 0 of Figure 5.3. Observe that this formulation is independent of the physical location of the ghost node, so the ghost node can be chosen to be infinitesimally close to the boundary node it supports. The solution state at the ghost node remains to

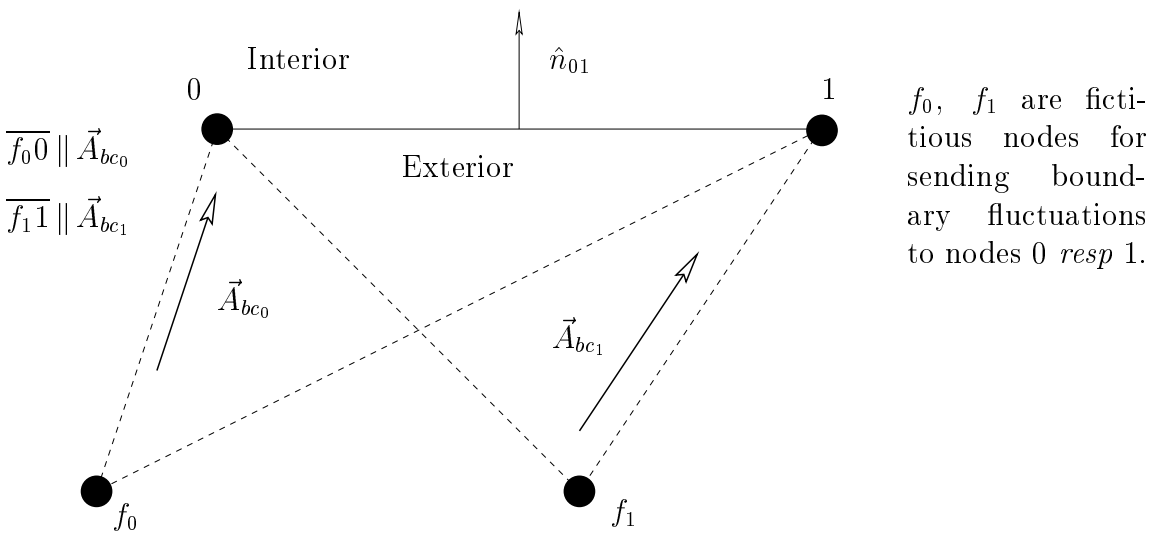


Figure 5.3: Weak implementation of fluctuation splitting boundary condition, imposed by specifying external state at ghost nodes $f_0,\ f_1$.

be specified, and can be varied node-to-node. The associated artificial dissipation is,

$$\phi'_{bc_0} \ = \text{sign}(\vec{A}\cdot\hat{n}_{01})\phi_{bc_0} \tag{5.94}$$

and the resulting distribution is,

$$S_0 U_{0_t} \ \leftarrow \frac{1}{2}(\phi_{bc_0} + \phi'_{bc_0}) + COE \tag{5.95}$$

Since no account of the ghost cell area is made in forming the dual area on the LHS of Eqn. 5.95, a scale factor on $[\frac{1}{2}, 1]$ can be applied to the distribution.

The extension to systems follows by analogy. The boundary fluctuation is defined,

$$\breve{\boldsymbol{\phi}}_{bc_0} \ = \frac{\varpi_{a_0}}{2}\ell_{01}\vec{\boldsymbol{\mathcal{A}}}_{bc_0}\cdot\hat{n}_{01}(\mathbf{W}_{f_0} - \mathbf{W}_0) \tag{5.96}$$

with the artificial dissipation,

$$\breve{\boldsymbol{\phi}}'_{bc_0} \ = \text{sign}(\vec{\boldsymbol{\mathcal{A}}}_{bc_0}\cdot\hat{n}_{01})\breve{\boldsymbol{\phi}}_{bc_0} \tag{5.97}$$

Additional dissipation for the suppression of expansion shocks is added to Eqn. 5.97

following the form of Eqns. 5.68 and 5.69 as,

$$+\varpi_{a_0}\frac{\ell_{01}}{4}\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & n_{01}^{\mathsf{x}^2}(\Delta_{\Lambda_3}+\Delta_{\Lambda_4}) & n_{01}^{\mathsf{x}}n_{01}^{\mathsf{y}}(\Delta_{\Lambda_3}+\Delta_{\Lambda_4}) & \frac{n_{01}^{\mathsf{x}}}{a}(\Delta_{\Lambda_3}-\Delta_{\Lambda_4})\\ 0 & n_{01}^{\mathsf{x}}n_{01}^{\mathsf{y}}(\Delta_{\Lambda_3}+\Delta_{\Lambda_4}) & n_{01}^{\mathsf{y}^2}(\Delta_{\Lambda_3}+\Delta_{\Lambda_4}) & \frac{n_{01}^{\mathsf{y}}}{a}(\Delta_{\Lambda_3}-\Delta_{\Lambda_4})\\ 0 & an_{01}^{\mathsf{x}}(\Delta_{\Lambda_3}-\Delta_{\Lambda_4}) & an_{01}^{\mathsf{y}}(\Delta_{\Lambda_3}-\Delta_{\Lambda_4}) & (\Delta_{\Lambda_3}+\Delta_{\Lambda_4})\end{bmatrix}(\mathbf{W}_{f_0}-\mathbf{W}_0) \tag{5.98}$$

The distribution to the boundary node is then formed as,

$$S_0\mathbf{U}_{0_t} \leftarrow \frac{1}{2}\mathbf{U}_W(\breve{\boldsymbol{\phi}}_{bc_0}+\breve{\boldsymbol{\phi}}'_{bc_0})+COE \tag{5.99}$$

This system treatment is only approximate, as the cross-fluctuation does not vanish when $\vec{V} \parallel \overline{f_0 0}$, as in the scalar case, but reduces to the term,

$$\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & n_{f_0}^{\mathsf{x}}\\ 0 & 0 & 0 & n_{f_0}^{\mathsf{y}}\\ 0 & a^2n_{f_0}^{\mathsf{x}} & a^2n_{f_0}^{\mathsf{y}} & 0\end{bmatrix} \tag{5.100}$$

Strictly, there should be some cross-coupling with the neighboring boundary nodes. However including the term from Eqn. 5.100 requires explicitly locating the ghost nodes, which can be impossible for certain geometries.

The distribution to node 1 is formed analogously, substituting 0 for 1 in Eqns. 5.93–5.100.

5.4.2 Boundary Types

The freestream boundary condition is enacted by specifying a complete, constant thermodynamic state and velocity vector. By using the weak boundary enforcement, this one boundary condition covers the four permutations of subsonic or supersonic, inflow or outflow.

The inviscid wall boundary is implemented by mirroring the primitive variables, either across the face for DMFDSFV or at the node for fluctuation splitting.

Viscous walls define a no-slip velocity and a specified wall temperature. The zero velocity at the wall causes the viscous axisymmetric source to be zero.

Both the full viscous flux and the approximate thin-layer flux of Haselbacher reduce to,

$$\vec{\mathbf{F}}^v \cdot \hat{n} \;= \frac{1}{R_{e_\infty}} \left[0,\, \mu \vec{V}_n,\, \kappa T_n\right] \tag{5.101}$$

at a wall, since $\vec{V}$, $\vec{\nabla}\cdot\vec{V}$, and $\vec{V}_n \cdot \hat{n}$ go to zero. Defining the heat transfer into the wall according to Fourier's law,

$$q_w \;= -\frac{\kappa}{R_{e_\infty}} T_n \tag{5.102}$$

and the wall shear stress,

$$\vec{\tau}_w \;= -\frac{\mu}{R_{e_\infty}} \vec{V}_n \tag{5.103}$$

allows the Eqn. 5.101 to be written as,

$$\vec{\mathbf{F}}^v \cdot \hat{n} \;= -\left[0,\, \vec{\tau}_w,\, q_w\right] \tag{5.104}$$

where the minus sign results from the choice of an outward unit normal, $\hat{n}$, to the control volume, which points into the wall at a boundary.

The solid wall is enforced weakly, by specifying the wall shear that will drive the flow momentum to zero and the heat flux that will drive the solution temperature to the desired wall temperature. An advantage of this weak approach is that wall heat transfer and skin friction are solved for directly, rather than as a post-processed least-squares reconstruction. Using an explicit update, the wall heat flux can be isolated as,

$$q_w \;= \frac{1}{\varpi_a \sum \Gamma_w} \left[\frac{\varpi_a S}{\Delta t}\left(U_4 - U_1\, e(T_w)\right) + RHS_4^{i+v}\right] \tag{5.105}$$

Similarly, the wall shear is,

$$\vec{\tau}_w \;= \frac{1}{\varpi_a \sum \Gamma_w} \left[\frac{\varpi_a S}{\Delta t}(U_2, U_3) + RHS_{2,3}^{i+v}\right] \tag{5.106}$$

On the axisymmetric axis the control-volume centroid for $\overline{\varpi}_a$ in Eqn. 5.5 is used for the temporal evolution, to avoid a singularity in the nodal update. The DMFDSFV axis condition is implemented by setting all fluxes to zero through the axisymmetric axis. For fluctuation splitting the axis is treated in the same manner as an inviscid wall, but with the radial term set to the cell centroid to ensure the numerator in the distribution contribution goes to zero faster than the denominator from the LHS time evolution. These implementations of the axisymmetric axis are observed to work well in practice except at the stagnation point on the axisymmetric axis, which is a singularity in the flowfield as well as a mathematical singularity. The deficiency observed with the weak boundary implementation at the axisymmetric stagnation point is a lack of preservation of massflow, where the velocity vectors sometimes allow a slight leak into or out of the domain, depending upon the particular flow conditions.

5.5 Temporal Evolution

Solutions at the nodes are updated using an explicit forward-Euler LHS. A Jacobi relaxation strategy is followed with local time stepping.

The CFL (Courant[108] *et al.*) criteria for explicit schemes is adapted for use with the node-based unstructured scheme. The inviscid timestep is defined by the most restrictive time for signal propagation, at the eigenvalue speeds, between adjacent nodes,

$$\Delta t_0 \;= \min\left(\frac{\|\vec{r}_{0i}\|}{|\vec{V}_0\cdot\hat{r}_{0i}| + a_0}\right) = \min\left(\frac{\vec{r}_{0i}\cdot\vec{r}_{0i}}{|\vec{V}_0\cdot\vec{r}_{0i}| + a_0\,\|\vec{r}_{0i}\|}\right) \tag{5.107}$$

where the current node is node 0 and i takes on nodal values for each distance-one neighbor of the current node.

The viscous timestep restriction is taken to be an approximation based upon the positivity analysis for the scalar case, Eqn. 3.54, assuming order-1 Prandtl number,

$$\Delta t_0 \;= \frac{4S_0\rho_0 R_{e_\infty}\Re}{\mu_0(\Re+\gamma-1)\sum_{\mathsf{T}}\frac{\ell_0^2}{S_{\mathsf{T}}}} \tag{5.108}$$

The stability and convergence of axisymmetric solutions is found to be enhanced by scaling the timestep for points near the cylindrical axis by the maximum of either the node height or the square root of the median-dual area.

The more restrictive of the inviscid or viscous timestep is used to scale the nodal update.

5.6 Verification and Validation

5.6.1 Coding Strategy

The code is written using a literate programming[109] style that blends source code (written in C) with documentation (LaTeX) in the same file. Maintaining close physical proximity between the code and documentation aids the debugging process. Modularity is emphasized, with individual functions being verified using sample inputs prior to being linked with the main driver routine.

Verification of the complete solver is performed in stages using a methodology derivative of Singhal[101]. A variety of canonical cases are constructed, including grid distortions, that are designed to exercise combinations of the various functions that comprise the complete solver. Validation is the application of the solver to complex flowfields with comparison to benchmark data.

5.6.2 Inviscid Verification

Distorted mesh

The first verification case simply passes a uniform flow through a distorted grid, with success being the preservation of uniformity to at least six significant digits. The domain is initialized to stagnant conditions with freestream flow impulsively applied at the boundaries. A variety of flow angles were tested on $-180° \leq \text{AOA} \leq 180°$ for subsonic, transonic, and supersonic Mach numbers. Regular, high aspect ratio (100), skewed ($2° < \theta < 175°$), and randomly distorted (Figure 5.4) meshes with 121 nodes were used. Initial runs were instrumental in refining the treatment of eigenvalue

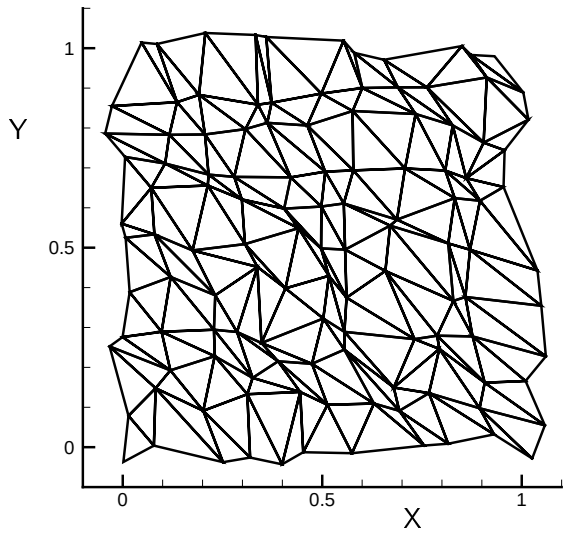


Figure 5.4: Sample randomly distorted mesh used for solver verification cases.

limiting for fluctuation splitting as described in sections 5.2.2 and 5.4. All final runs were successful for both DMFDSFV and fluctuation splitting.

Converging Mach streams

Thermodynamic routines are verified by considering converging Mach streams, inclined at $\pm 10°$. The upper stream is at Mach 2.3 while the lower stream has Mach 1.8. The two streams have matched densities but a temperature ratio of 1.0812, resulting in a horizontal slip line behind the oblique shocks. A complete description of the analytic solution appears in Figure 5.5 and Table 5.1.

A sequence of four meshes, with a refinement ratio of 1.5, is considered. The meshes are triangulated from 16×16, 24×24, 36×36, and 54×54 grids. The triangulated 16×16 grid is shown in Figure 5.6. The finer meshes cover the same domain and are constructed similarly to the shown mesh.

A Mach-number contour plot for fluctuation splitting on the finest mesh is shown

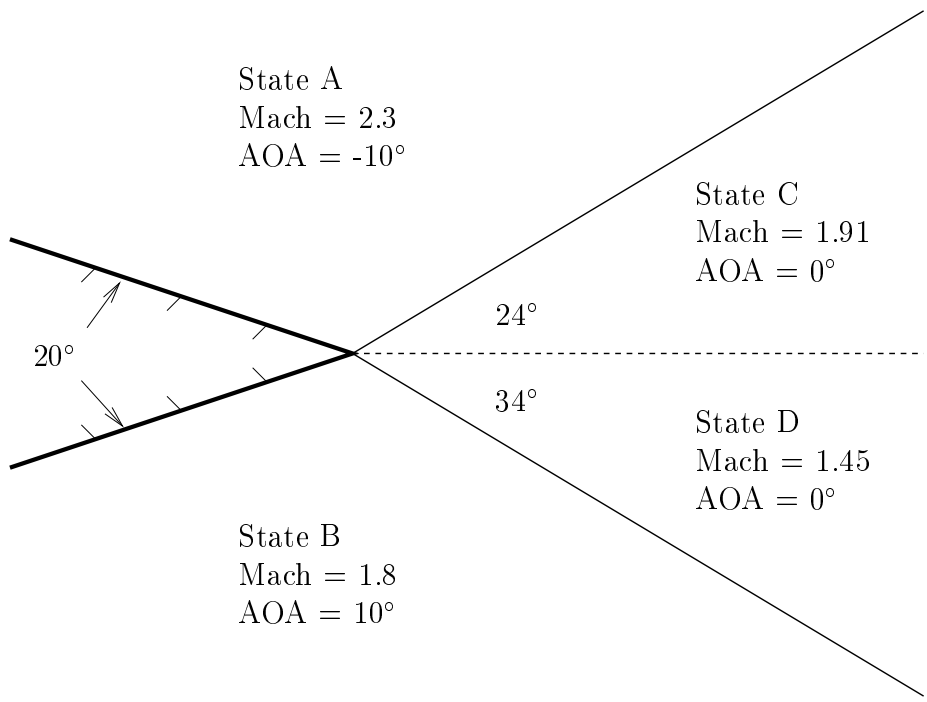


Figure 5.5: Description of converging Mach stream problem. Flow from left to right, with oblique shocks, solid, and slip-line, dashed, emanating from trailing edge of splitter plate.

State	ρ, kg/m^3	T, K	P, kPa	V, m/s
A	1.2	300	103.34	798.6
B	1.2	324.7	111.73	649.9
C	1.813	356.7	185.62	723.7
D	1.718	376.4	185.62	563.7

Table 5.1: Analytic thermodynamic states for converging Mach streams.

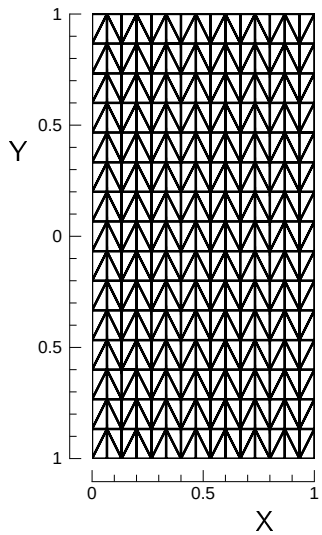


Figure 5.6: 16×16 mesh for converging Mach streams.

in Figure 5.7, showing crisp discontinuity resolution and the correct post-shock Mach numbers. The shock angles for all eight cases, *i.e.* DMFDSFV and fluctuation splitting on each mesh, are measured to be correct within $\pm 1°$. The L_2-norms of the primitive-variables error at states C and D are plotted versus the characteristic mesh size in Figure 5.8. The slopes of the regression lines are indicative of the order of accuracy with respect to grid convergence of the two schemes for this test case. DMFDSFV exhibits second-order convergence, as expected. Unexpectedly, fluctuation splitting shows super-convergence for this particular case. True multi-dimensional upwinding is likely the source of the exceptional fluctuation splitting accuracy for this purely-supersonic flow. Supplementing the graphical determination of the grid-convergence rates, the equations presented by Roache[102], based on a Richardson extrapolation, yield average grid-convergence rates of 3.0 for fluctuation splitting and 2.1 for DMFDSFV.

Temporal convergence rates are plotted in Figure 5.9, with timings performed on an IRIS R10000 platform. All cases were run using the Minmod limiter and a Jacobi

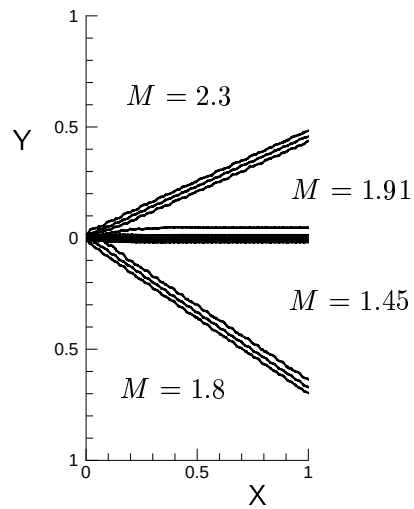


Figure 5.7: Mach contours from fluctuation splitting on finest mesh.

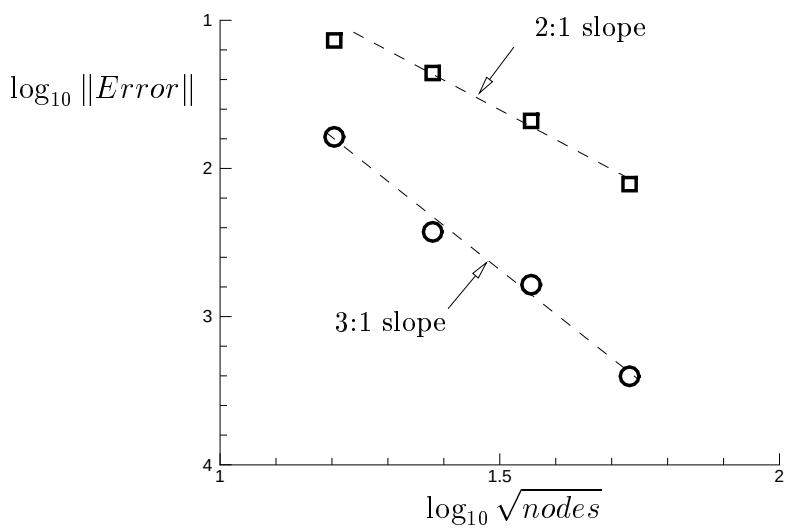


Figure 5.8: Grid convergence rates for converging Mach stream case. Circles = fluctuation splitting, squares = DMFDSFV.

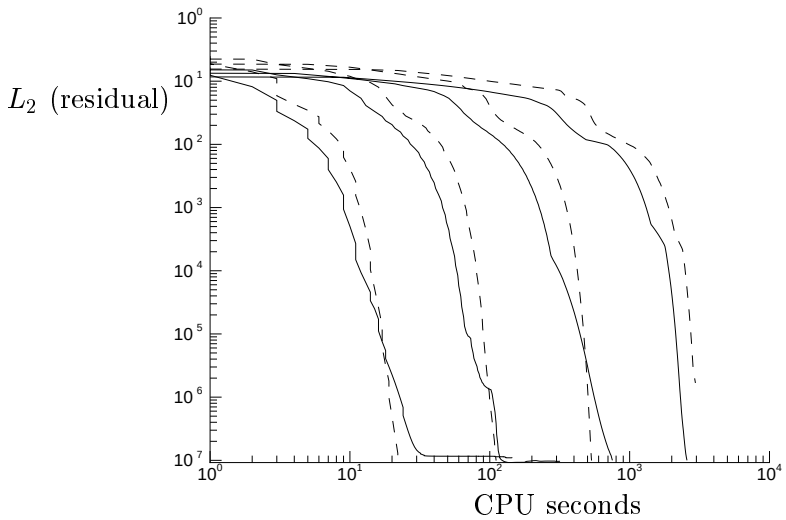


Figure 5.9: Convergence histories for converging Mach stream case. Fluctuation splitting solid, DMFDSFV dashed. Coarsest mesh on left, finest on right.

update strategy with local time steps. Fluctuation splitting was run with a unity CFL number, while best convergence for DMFDSFV was found for CFL=0.7. Fluctuation splitting runs at 145 μs per node per iteration, while DMFDSFV runs at 165 μs per node per iteration.

Diamond airfoil

A verification of the inviscid wall boundary condition is performed on a diamond airfoil at zero angle of attack and Mach 1.5. The flow deflection is five degrees. The grid is shown in Figure 5.10. A Mach-number contour plot using fluctuation splitting is shown in Figure 5.11. The corresponding DMFDSFV solution, not shown, is visually indistinguishable from the fluctuation splitting solution. The analytic drag coefficient, based on chord length, is 0.02760. The fluctuation splitting drag coefficient is 0.02638, for a 4.4 percent error. The DMFDSFV result has an error of 6.6 percent from a drag coefficient of 0.02579.

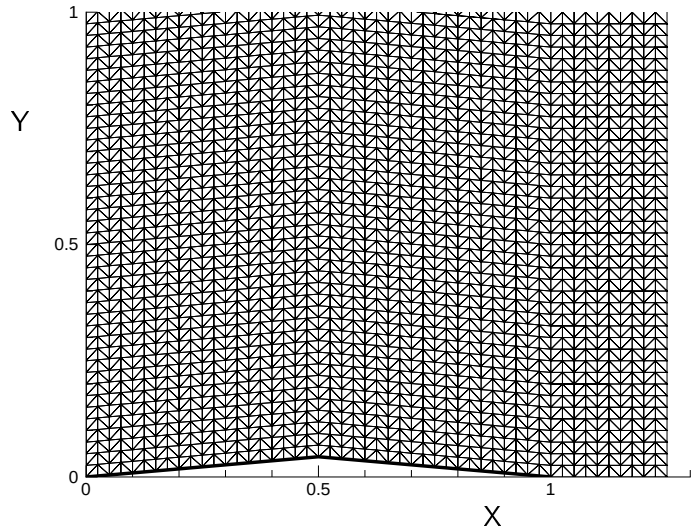


Figure 5.10: Grid for diamond airfoil verification test.

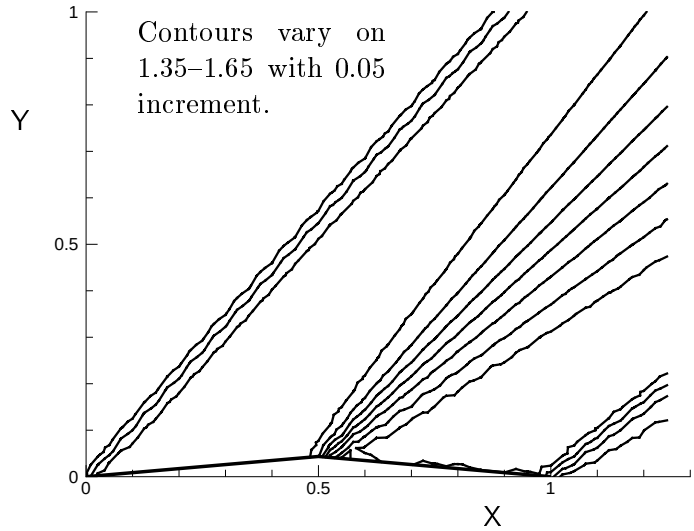


Figure 5.11: Mach contours on diamond airfoil, $M = 1.5$, fluctuation splitting solution.

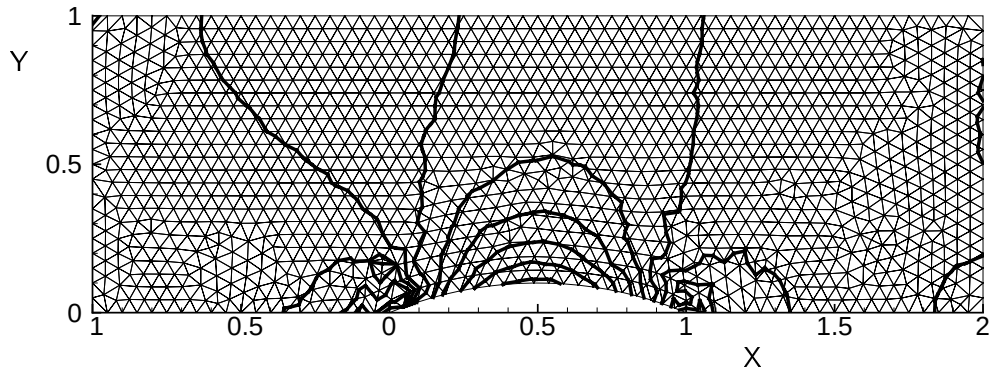


Figure 5.12: Two-dimensional 10 percent circular bump mesh with isobars from fluctuation splitting solution at Mach 0.1.

Circular bump

A subsonic two-dimensional verification is performed on a 10 percent circular bump at Mach 0.1. The 1389-node mesh with isobars from the fluctuation splitting solution is shown in Figure 5.12. A true incompressible inviscid flow would have symmetric isobars fore and aft, and zero drag. The fluctuation splitting drag coefficient, based on cord length is 0.0058. DMFDSFV predicts a drag coefficient more than twice as large, 0.0128. A lower fluctuation splitting drag coefficient is indicative of lower levels of artificial dissipation in the solution for this case.

Sphere

In a similar vein, Mach 0.1 axisymmetric flow over a sphere is tested on a 1369-node mesh. The drag coefficient, based on frontal area, is 0.43 for DMFDSFV but 0.56 for fluctuation splitting. Contrary to expectation, the increased artificial dissipation in the DMFDSFV solution creates enough of a total pressure loss to nearly eliminate separation on the leeside, whereas the leeside increase in pressure toward the centerline in the fluctuation splitting solution does produce a sizable separation region, and in this case a larger drag coefficient. As with subsonic bump case, true incompressible,

inviscid flow should theoretically produce zero drag. Both solvers converged to steady solutions for this case.

Cone

The final inviscid verification is for an 11-degree semi-vertex-angle cone at Mach 1.5. The well-established Taylor-Maccoll[110] method for the conical supersonic Euler equations predicts a drag coefficient, based on base area with no base pressure, of 0.7795. The fluctuation splitting solution, which converged seven orders of magnitude in 38 seconds, predicts a drag coefficient of 0.7785, for only a 0.13 percent error. The DMFDSFV solution, which took 13 percent longer at 43 seconds to reach seven orders of magnitude residual convergence, predicts a 0.7754 coefficient, for an error of 0.53 percent.

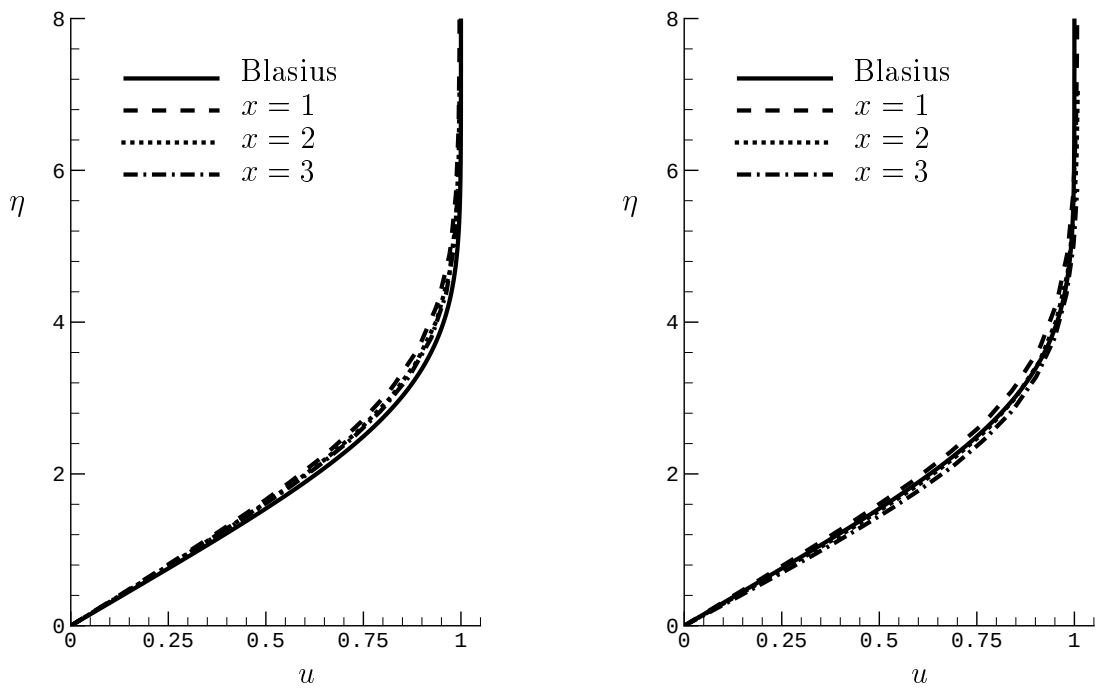
5.6.3 Viscous Validation

Two canonical viscous validation cases are considered: a subsonic flat plate and a hypersonic cylinder. Steady laminar solutions are obtained using the Haselbacher thin-layer viscous treatment with containment-dual modification as described in section 5.3.

Flat plate

The classic Blasius[111] flat-plate boundary layer problem is solved on a rectangular domain. Mach 0.3 flow enters 2 units upstream of the plate leading edge, which is located at the origin. The plate is 4 units long, ending at an extrapolation outflow boundary. The upper boundary is 1.2 units above the plate. The Reynolds number is 10^4.

The meshes are obtained from a structured grid containing 37 equally-spaced points parallel to the plate, 12 points upstream of the plate and 25 points on the plate, and 41 points normal to the plate. The vertical grid spacing at the wall is 0.001 units with an exponential stretching as described in Ref. [112], placing approximately 20 nodes within the boundary layer. The unstructured mesh is formed from



(a) Fluctuation splitting.

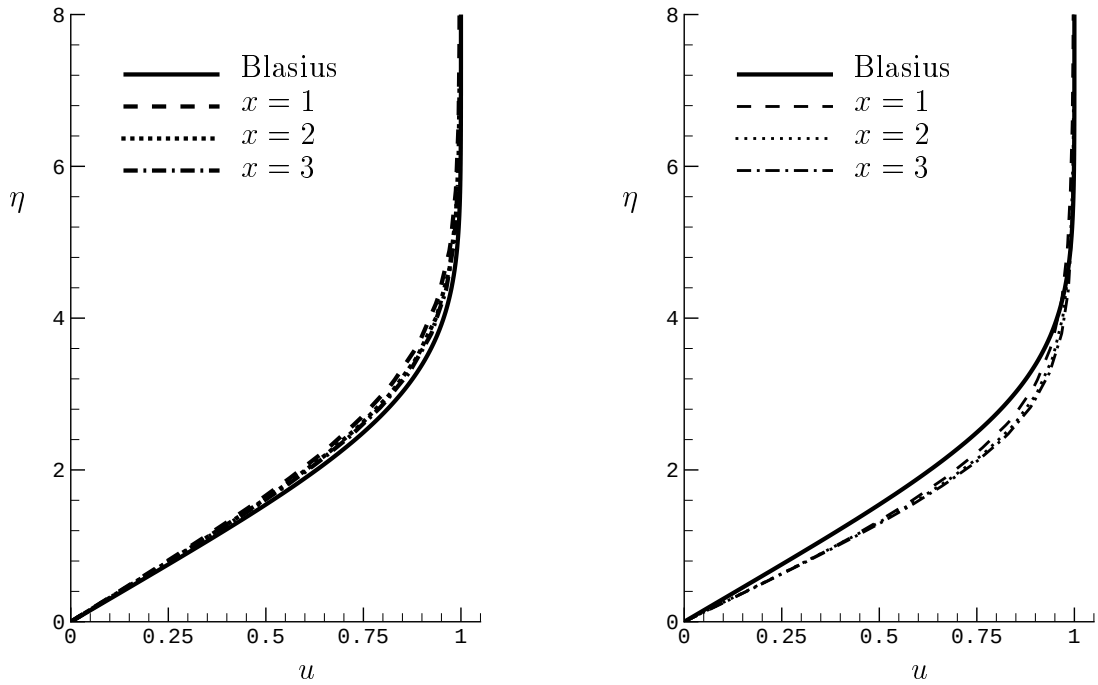
(b) DMFDSFV.

Figure 5.13: Boundary layer profiles of tangential velocity extracted from three stations on flat plate.

the structured grid using diagonal cuts in an alternating pattern. Two coarser meshes are similarly constructed by successively deleting every-other node in the wall-normal direction, leaving 10 and 5 nodes, respectively, in the boundary layer for the medium and coarse grids.

Boundary layer profiles of u are extracted at $x = 1, 2, 3$ from both the fluctuation splitting and DMFDSFV solutions and plotted versus the Blasius solution in Figure 5.13. The boundary layer scaling variable is defined as,

$$\eta = y\sqrt{\frac{R_e}{x}} \tag{5.109}$$



(a) Containment dual approximation.

(b) Strict median dual implementation.

Figure 5.14: Boundary layer profiles computed using two different viscous dual mesh definitions.

Both solution sets match the Blasius profile, indicating well-developed flow with adequate grid resolution on the finest mesh.

Figure 5.14 shows the effect of using the containment-dual approximation in the Haselbacher thin-layer viscous treatment. Boundary layer profiles of u are again extracted at $x = 1, 2, 3$, with both solutions being run with fluctuation splitting. Figure 5.14(a) is the same as Figure 5.13(a), while Figure 5.14(b) uses the strict median-dual definition for the viscous terms. For the highly-stretched grid elements used in this case, it is clear that the containment-dual approximation provides improved boundary-layer resolution, while omitting the approximation leads to a profile that is "too full."

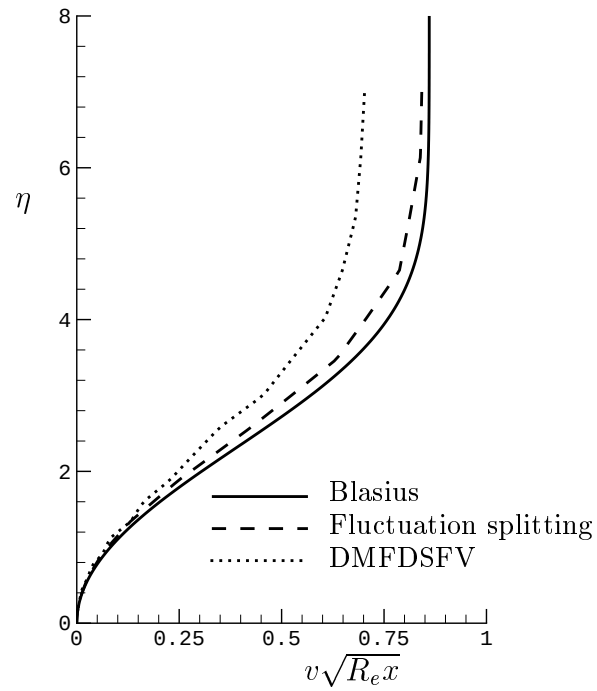
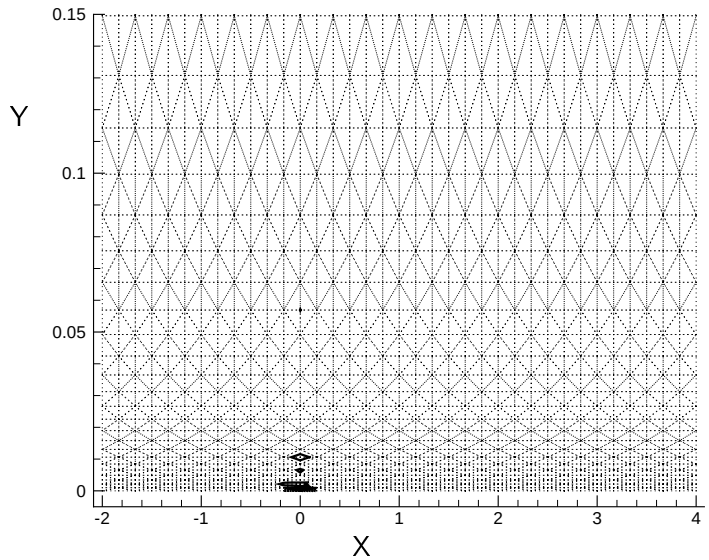


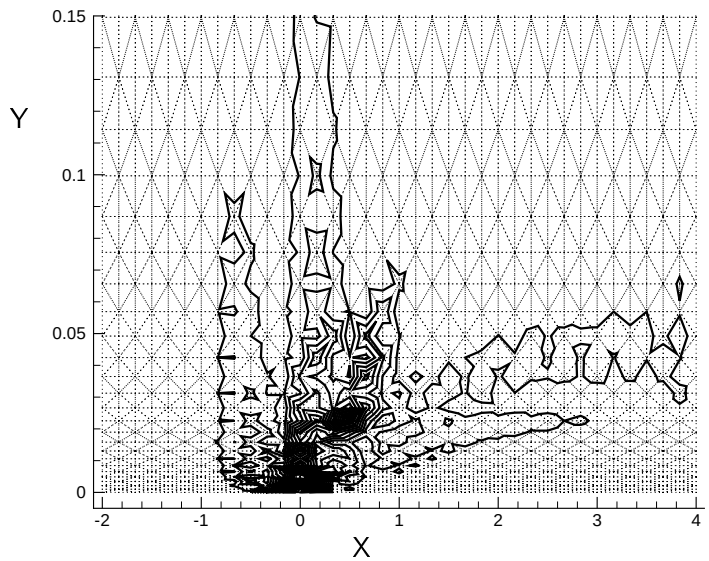
Figure 5.15: Boundary layer profiles of vertical velocity extracted from midpoint of flat plate.

The v-velocity profiles from the fluctuation splitting and DMFDSFV solutions are compared in Figure 5.15, both extracted from the plate at $x = 2$. The fluctuation splitting solution comes much closer to matching the Blasius profile than the DMFDSFV result. Excessive artificial dissipation is produced by the DMFDSFV scheme in the y-momentum equation, which suppresses the v-velocity below the analytic value. The artificial dissipation contributions to the y-momentum equation are plotted for both fluctuation splitting and DMFDSFV in Figure 5.16. The vertical scale has been enlarged by a factor of 30 to zoom in on the boundary layer in Figure 5.16. Clearly, DMFDSFV is producing significantly more artificial dissipation than fluctuation splitting over the length of the boundary layer.

For this essentially incompressible case, the suppression of the vertical velocity



(a) Fluctuation splitting.



(b) DMFDSFV.

Figure 5.16: Artificial dissipation production in the y-momentum equation. Eleven contours spaced equally on 0–0.0005.

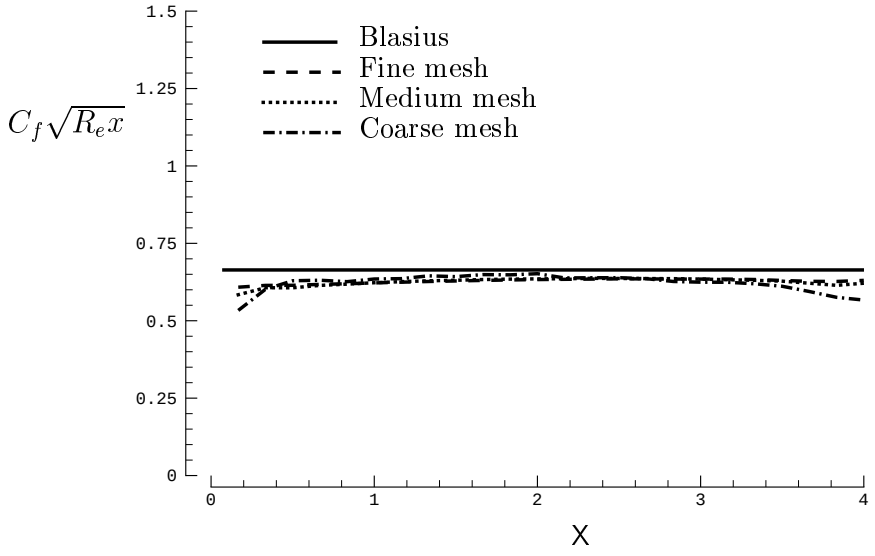
due to excessive artificial dissipation is manifested by an increase in skin friction coefficient, as shown in Figure 5.17, where the friction coefficient increases with running length for DMFDSFV, but not for fluctuation splitting. Recall that DMFDSFV is continuously producing artificial dissipation over the length of the plate while the fluctuation splitting dissipation is restricted to the leading-edge region only. Figure 5.17 presents data from all three grid refinement levels. The DMFDSFV results degrade dramatically with coarsening of the mesh, but the fluctuation splitting results remain relatively invariant with mesh resolution, all the way down to only five nodes in the boundary layer.

The medium-mesh DMFDSFV solution was repeated using the full Navier-Stokes treatment, rather than the thin-layer equations. No change in the skin-friction results are seen over the first half of the plate, Figure 5.18, though there is an 8-percent improvement toward the end of the plate. Solving for the full Navier-Stokes terms requires 11 percent more CPU time per iteration.

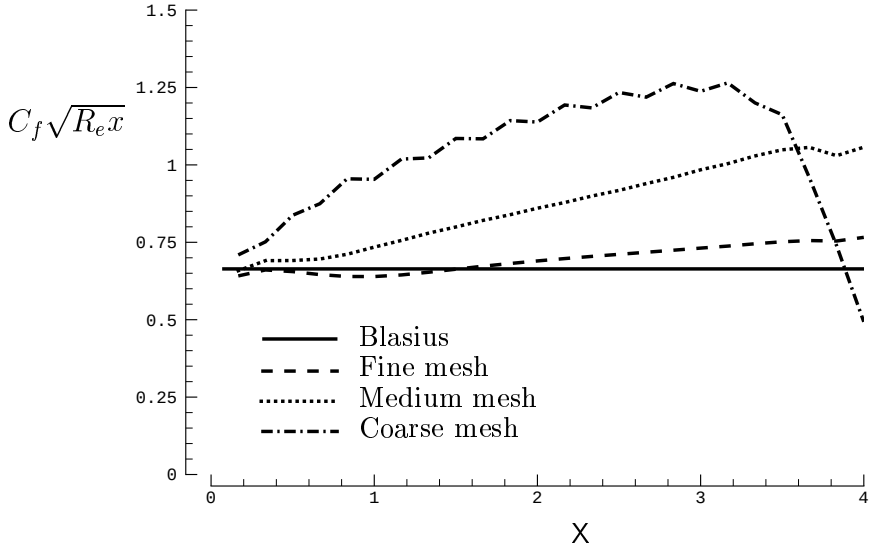
Cylinder

The opposite end of the Mach-number spectrum is used to validate heat-transfer calculations, in this case for a cylinder of 1 m radius in Mach 17.6 flow. The perfect-gas assumption is a poor physical model for these extreme conditions, $V_\infty = 5$ km/s, $\rho_\infty = 0.001$ kg/m^3, $T_\infty = 200$ K, $T_{wall} = 500$ K, but the case provides a severe test of the algorithms under a re-entry scenario. Results are compared against the LAURA[15, 16, 113] benchmarks.[1] The LAURA code is well-established as a structured-grid hypersonic solver. Also included in the LAURA benchmark data is a solution using the unstructured-mesh finite volume solver FUN2D[83]. The FUN2D code employs the same basic inviscid and viscous discretization strategy as the present DMFDSFV scheme. However, differences exist between the two codes in their eigenvalue limiting and their flux limiting, where FUN2D uses the Venkatakrishnan limiter[114] while the present DMFDSFV scheme is using Minmod. Also, FUN2D uses a strong boundary enforcement as opposed to the weak formulation and the FUN2D heating rates are post-processed from the temperature field rather than using the heat flux directly

[1] `http://hefss.larc.nasa.gov/`



(a) Fluctuation splitting.



(b) DMFDSFV.

Figure 5.17: Skin friction coefficients for Blasius flow.

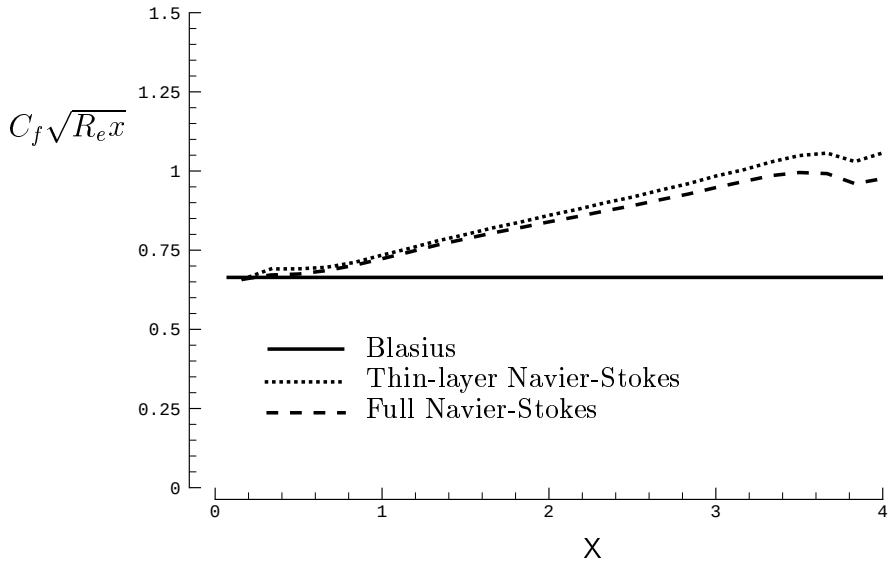
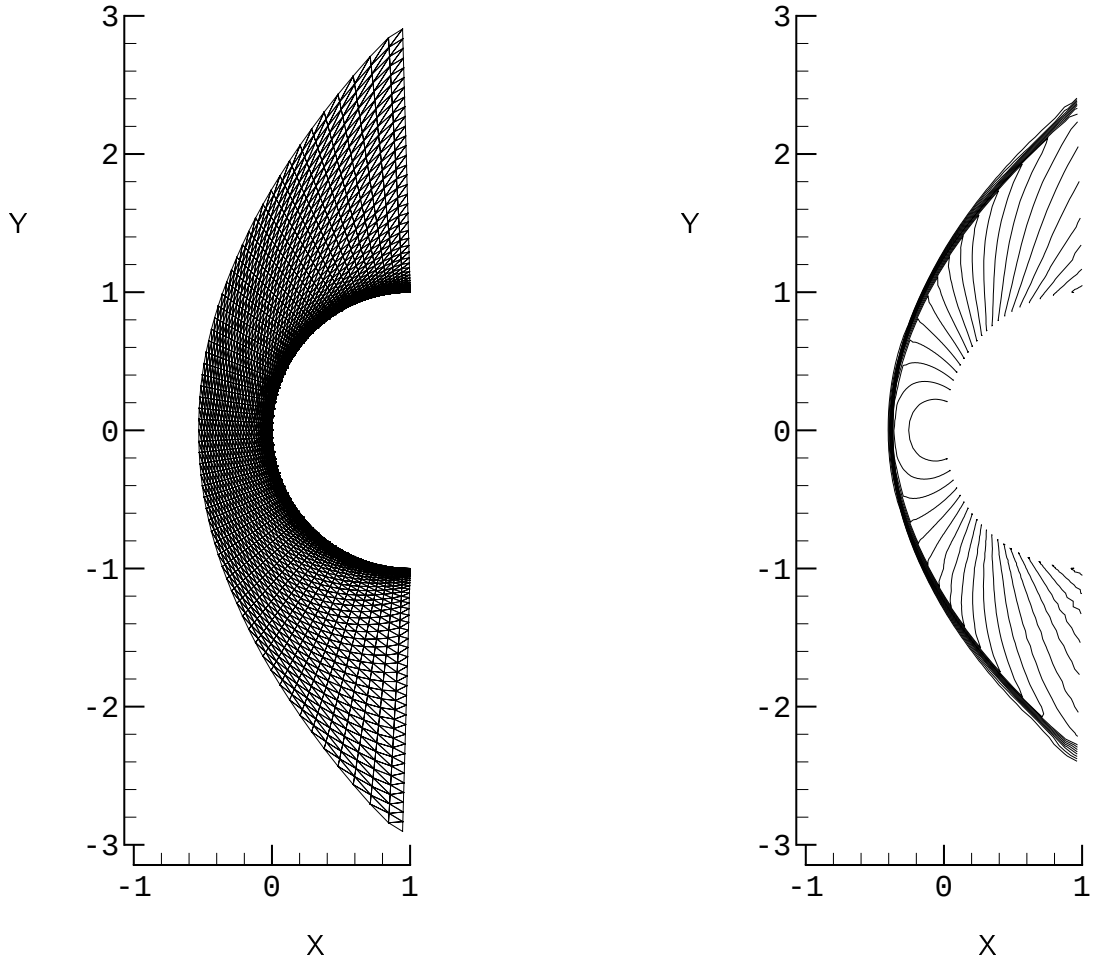


Figure 5.18: Effect of viscous modeling on skin friction.

from the RHS discretization, as is used here.

The unstructured grid for this case was obtained by simple triangulation of the LAURA grid, which has 65 nodes perpendicular to the surface, clustered to the wall, and circumferential nodes spaced every 3 degrees. Only the forward-half of the cylinder is solved, as shown in the mesh and flowfield solution of Figure 5.19.

The surface pressure coefficient is plotted versus rotation angle from the stagnation point for both the fluctuation splitting and DMFDSFV solutions, along with the LAURA and FUN2D results in Figure 5.20. The LAURA, FUN2D, and fluctuation splitting curves all over-plot, and the DMFDSFV solution nearly over-plots, being 1 percent low at the stagnation point and slightly high by a similar amount 90 degrees away. The calculations were repeated on a grid coarsened by a factor of four (skip of two in both structured-grid directions), with surface pressure results plotted in Figure 5.21 along with the fine-mesh LAURA solution. The coarsened fluctuation splitting surface pressures retain good agreement, and the DMFDSFV solution matches over most of the cylinder, with minor exceptions again at the stagnation



(a) Grid.

(b) Pressure contours.

Figure 5.19: Hypersonic cylinder domain with fluctuation splitting solution.

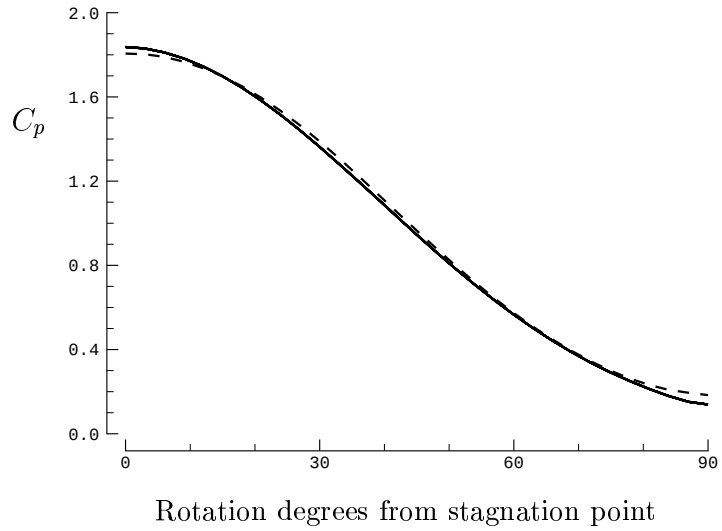


Figure 5.20: Cylinder surface pressures, solid = fluctuation splitting, LAURA, and FUN2D, while dashed = DMFDSFV.

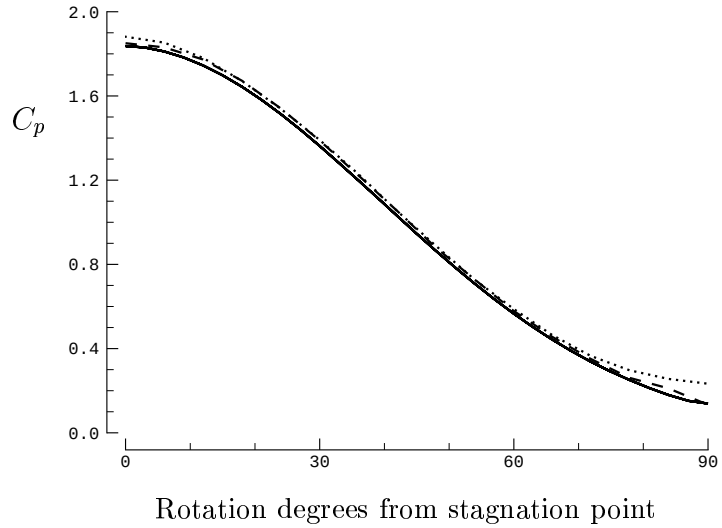


Figure 5.21: Cylinder surface pressures on coarsened mesh, solid = LAURA, dashed = fluctuation splitting, and dotted = DMFDSFV.

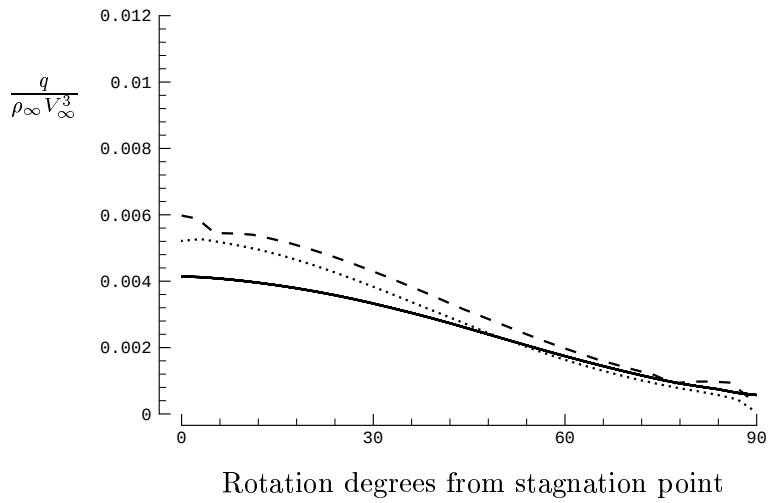


Figure 5.22: Cylinder surface heat-transfer rates, solid = LAURA, dashed = FUN2D, and dotted = fluctuation splitting.

point, 1 percent high on this grid, and at the 90 degree point.

Surface heat-transfer rates for LAURA, FUN2D, and fluctuation splitting are shown in Figure 5.22. Both of the unstructured-mesh solutions show elevated heating at the stagnation region, with fluctuation splitting being 30 percent higher than LAURA while FUN2D is 50 percent higher. The DMFDSFV solution is shown in Figure 5.23, predicting stagnation heating rates more than double the LAURA predictions. Also included in Figure 5.23 is the fluctuation splitting solution on the coarse mesh, which is seen to produce heating rates closer to the LAURA solution than DMFDSFV does on the fine mesh. While the basic DMFDSFV and FUN2D schemes are ostensibly the same on the interior domain, the significant differences in boundary implementation and limiting detailed at the start of this section are the reasons for the differing heat transfer predictions between the codes.

The fine-mesh solutions were repeated using the full Navier-Stokes treatment, and no changes in heating levels were observed.

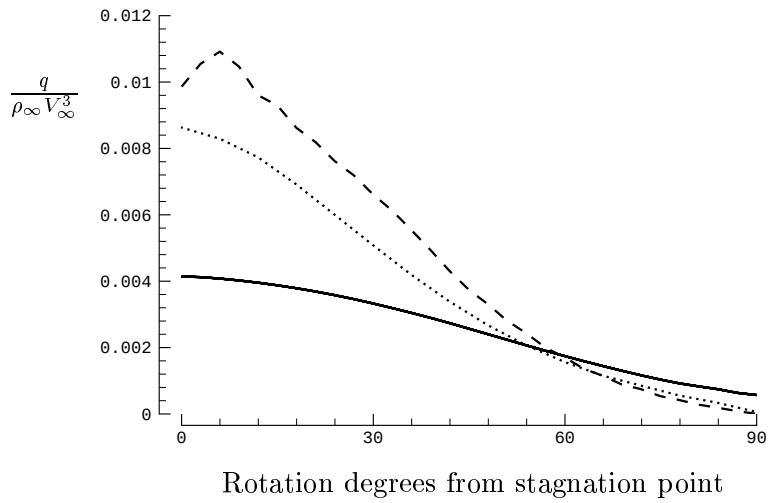


Figure 5.23: Cylinder surface heat-transfer rates, solid = LAURA, dashed = DMFDSFV(fine), and dotted = fluctuation splitting(coarse).

5.6.4 Summary of Results

Six inviscid test cases verify and compare the fluctuation splitting and DMFDSFV implementations. Both schemes are able to maintain uniform flow on a severely distorted mesh. DMFDSFV displays the design order of accuracy, second-order, for converging Mach streams while fluctuation splitting surprisingly displays third-order accuracy for this case. Fluctuation splitting is more accurate than DMFDSFV for the diamond airfoil, circular bump, and supersonic cone, but DMFDSFV is more accurate for the sphere test. Timings reveal fluctuation splitting runs 10–12 percent faster than DMFDSFV per node with the present algorithms.

The two viscous validation cases are an incompressible flat plate and $M = 17.6$ cylinder. On the flat plate fluctuation splitting produces significantly less artificial dissipation than DMFDSFV and provides much better skin friction predictions on coarse meshes, down to only five points in the boundary layer. Both schemes produce excellent surface pressures for the hypersonic cylinder and fluctuation splitting does

better for surface heating, though neither unstructured scheme predicts heating as well as the structured-mesh benchmark result.

Chapter 6

System Mesh Adaption

6.1 Overview

The adaption strategies detailed in chapter 4 for scalar advection-diffusion problems are extended to the Navier-Stokes system of equations. For the first time, the fluctuation minimization strategy for mesh adaption is developed for two-dimensional and axisymmetric systems, and is compared with the curvature clustering adaption that is representative of the current state of the art.

This chapter's application case is a Mach-10 wind tunnel simulation of a sting-mounted Mars capsule model. This case has previously been studied by Hollis[115] using a structured-grid adaption strategy, and the resulting agreement with experimental data for the sting heating was "found to be highly dependent upon grid resolution and grid quality" in the wake. The structured-grid approach to this problem required an extremely fine mesh with extensive adaptation by hand. The challenge for the unstructured-grid techniques will be to obtain comparable accuracy using fewer grid points with fully automated adaption.

6.2 Curvature Clustering

The local, anisotropic adaption strategy based upon *a posteriori* error estimates is extended directly from section 4.1. In extending to the system, Eqn. 4.2 generalizes

to,

$$|\mathbf{E}_r| \sim \left| \frac{\vec{\nabla}\mathbf{U}_1 \cdot \frac{\vec{r}_{01}}{\ell_{01}} - \vec{\nabla}\mathbf{U}_0 \cdot \frac{\vec{r}_{01}}{\ell_{01}}}{\ell_{01}} \right| \ell_{01}^2 = \left| \left(\vec{\nabla}\mathbf{U}_1 - \vec{\nabla}\mathbf{U}_0 \right) \cdot \vec{r}_{01} \right| \tag{6.1}$$

A scalar value for the error estimate is required, with a simple choice being to just take the L_2-norm of the error vector. Also, the error does not necessarily need to be formed from the conserved variables, but could alternatively be formed from the primitive variables, the Mach number as Habashi[76] does, or even a derived quantity such as heat transfer.

The basic operations remain point deletion, edge swapping, point insertion, and nodal displacements. These procedures are applied sequentially in the order listed, with the solver iterating on the solution between each of the four steps, constituting a complete adaption cycle. A tolerance can be set for thresholds to define when the mesh is sufficiently adapted.

While the explicit use of nodal gradients in Eqn. 6.1 makes this style of adaption naturally symbiotic with the DMFDSFV scheme used here, there is no restriction against using curvature clustering with the fluctuation splitting solver. Also note that Eqn. 6.1 is written in vector notation, and so can be applied directly in three dimensions just as it is used here in two spatial dimensions.

6.3 Characteristic Alignment

Chapter 4 demonstrated that fluctuation splitting is an exact solver for linear advection on a characteristically-aligned mesh. Adaption for characteristic alignment was developed as an automated, anisotropic local-operation strategy in sections 4.2 and 4.4. Applications to non-linear and advection-diffusion problems showed that alignment with characteristics was approximated, but not fully achieved, by the automated local scheme. For the Navier-Stokes system, true characteristic alignment is not a physically valid ideal due to the presence of multiple and/or imaginary characteristics. However, the automated local anisotropic adaption strategy is extended here with the objective of achieving solution improvement with fluctuation minimization

as the agent to obtain characteristic-driven alignment in supersonic regions consistent with mesh enrichment in the subsonic regions.

As with the scalar case, the analogy to the edge error estimate is formed from the fluctuations, both inviscid and viscous, in the cells adjoining the edge. For axisymmetric cases the fluctuation is scaled by the inverse of the cell-centroid y value. Additional weighting by the inverse of the square root of the cell area is found to balance the contributions from neighboring cells of disparate sizes.

Point deletion

Nodes are flagged for deletion if the sum of the L_2-norms of the fluctuations in all surrounding cells is below a threshold. Edges are swapped in an attempt to reduce the number or edges connected at the node to three or four to match the canonical point-removal patterns. If the local connectivity is too complicated for the automated pattern matching, the node is simply left in the domain. During a typical cycle, on the order of only 10 percent of the nodes that had been flagged for deletion will be left in the domain because of localized complicated connectivity.

Point insertion

An edge is split, adding a node, if the sum of the L_2-norms of the fluctuations in the cells to either side of the edge exceed a threshold. The cell fluctuations are weighted in the same manner as was described for the point deletions.

Edge swapping

An edge is flagged as a candidate for swapping if the RMS of the fluctuations in the cells to either side of the edge exceed the target threshold. If the swapped configuration maintains a physically valid grid, then the RMS of the fluctuations in the swapped cells are computed. If the swapped configuration has a smaller RMS value then the edge swap is performed.

Nodal displacement

Mesh movement for the system is driven by the minimization of a functional, just as for the scalar case. The system extension of Eqn. 4.21 is,

$$\Upsilon_i = \frac{1}{2}\sum_{\mathsf{T}} \breve{\boldsymbol{\phi}}_{\mathsf{T}}^{\mathsf{T}} \boldsymbol{\Xi}_{\mathsf{T}} \breve{\boldsymbol{\phi}}_{\mathsf{T}} \tag{6.2}$$

The functional Υ can be expressed in terms of either the conserved or auxiliary fluctuations, as the minimization of one implies the minimization of the other. The derivative of Υ (see Eqn. 4.22) is formed using the chain rule as,

$$\frac{\partial \Upsilon_i}{\partial x_i} = \frac{1}{2}\sum_{\mathsf{T}} \left(\frac{\partial \breve{\boldsymbol{\phi}}_{\mathsf{T}}^{\mathsf{T}}}{\partial x_i} \boldsymbol{\Xi}_{\mathsf{T}} \breve{\boldsymbol{\phi}}_{\mathsf{T}} + \breve{\boldsymbol{\phi}}_{\mathsf{T}}^{\mathsf{T}} \frac{\partial \boldsymbol{\Xi}_{\mathsf{T}}}{\partial x_i} \breve{\boldsymbol{\phi}}_{\mathsf{T}} + \breve{\boldsymbol{\phi}}_{\mathsf{T}}^{\mathsf{T}} \boldsymbol{\Xi}_{\mathsf{T}} \frac{\partial \breve{\boldsymbol{\phi}}_{\mathsf{T}}}{\partial x_i} \right) \tag{6.3}$$

Having defined the gradient of the functional, the method of steepest descent from Eqns. 4.24 and 4.25 can be applied directly to drive the nodal displacements.

The weighting factor $\boldsymbol{\Xi}_{\mathsf{T}}$ is a symmetric positive-definite matrix. Weighting each equation equally without regard to cell sizes results in $\boldsymbol{\Xi}_{\mathsf{T}} = I$, while inverse area weighting yields $\boldsymbol{\Xi}_{\mathsf{T}} = \frac{1}{S_{\mathsf{T}}} I$. The derivatives of $\boldsymbol{\Xi}_{\mathsf{T}}$ for these and other area-weighted choices have been covered in Eqns. 4.50–4.56.

The cell fluctuation can be rearranged from Eqn. 5.41 (two-dimensional terms) as,

$$\breve{\boldsymbol{\phi}} = \frac{1}{2}\sum_{j=1}^{3} \ell_j \hat{n}_j \cdot \tilde{\vec{\boldsymbol{\mathcal{A}}}} \mathbf{W}_j \tag{6.4}$$

where $\mathbf{W}_j = \mathbf{W}_{\bar{Z}} \mathbf{Z}_j$. Following Eqns. 4.32 and 4.33, the derivatives of the fluctuation are,

$$\frac{\partial \breve{\boldsymbol{\phi}}}{\partial x_2} = \frac{1}{2}\left[\tilde{\mathcal{A}}^{\mathsf{y}} (\mathbf{W}_1 - \mathbf{W}_3) + \sum_{j=1}^{3} \ell_j \hat{n}_j \cdot \left(\frac{\partial \tilde{\vec{\boldsymbol{\mathcal{A}}}}}{\partial x_2} \mathbf{W}_j + \tilde{\vec{\boldsymbol{\mathcal{A}}}} \frac{\partial \mathbf{W}_j}{\partial x_2} \right) \right] \tag{6.5}$$

$$\frac{\partial \breve{\boldsymbol{\phi}}}{\partial y_2} = \frac{1}{2}\left[\tilde{\mathcal{A}}^{\mathsf{x}} (\mathbf{W}_3 - \mathbf{W}_1) + \sum_{j=1}^{3} \ell_j \hat{n}_j \cdot \left(\frac{\partial \tilde{\vec{\boldsymbol{\mathcal{A}}}}}{\partial y_2} \mathbf{W}_j + \tilde{\vec{\boldsymbol{\mathcal{A}}}} \frac{\partial \mathbf{W}_j}{\partial y_2} \right) \right] \tag{6.6}$$

The flux Jacobian of the auxiliary variables is,

$$\vec{\mathcal{A}} = \vec{V} I + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \hat{\imath} \\ 0 & 0 & 0 & \hat{\jmath} \\ 0 & a^2\hat{\imath} & a^2\hat{\jmath} & 0 \end{bmatrix} \tag{6.7}$$

which leads to the approximation $\frac{\partial \vec{\boldsymbol{\mathcal{A}}}}{\partial x} \simeq \frac{\partial \vec{V}}{\partial x} I$. As was done for the scalar case, it is assumed that moving a node does not change the solution at the other two nodes of the triangle, so that the variation of the cell-average flux Jacobian scales like one-third the variation of the velocity at the node being moved, $\frac{\partial \bar{\vec{\boldsymbol{\mathcal{A}}}}}{\partial x_2} \simeq \frac{1}{3}\frac{\partial \vec{V_2}}{\partial x_2} I$.

Similarly, the variation of the Jacobian of the transformation scales like one-third the nodal variation, $\frac{\partial \mathbf{W}_{\bar{Z}}}{\partial x_2} \sim \frac{1}{3}\frac{\partial \mathbf{Z}_2}{\partial x_2}$, and is neglected as a sub-principle term relative to the change in solution value,

$$\frac{\partial \mathbf{W}_j}{\partial x_2} \simeq \mathbf{W}_{\bar{Z}} \frac{\partial \mathbf{Z}_j}{\partial x_2} \tag{6.8}$$

Since the solution is locally assumed to vary only at the node being moved, i.e., $\frac{\partial \mathbf{Z}_1}{\partial x_2} = \frac{\partial \mathbf{Z}_3}{\partial x_2} = 0$,

$$\sum_{j=1}^{3} \frac{\partial \mathbf{W}_j}{\partial x_2} \simeq \mathbf{W}_{\bar{Z}} \frac{\partial \mathbf{Z}_2}{\partial x_2} \tag{6.9}$$

The remaining term to evaluate is $\frac{\partial \mathbf{Z}_2}{\partial x_2}$. The steady-state distribution can be written,

$$\sum_{\mathsf{T}} \left[\left(I + \mathbf{M}_\alpha\right) \boldsymbol{\alpha} - \left(I + \mathbf{M}_\beta\right) \boldsymbol{\beta} \right] \mathbf{W}_{\bar{Z}} \mathbf{Z}_2 = \sum_{\mathsf{T}} \left[\left(I + \mathbf{M}_\alpha\right) \boldsymbol{\alpha} \mathbf{W}_{\bar{Z}} \mathbf{Z}_1 - \left(I + \mathbf{M}_\beta\right) \boldsymbol{\beta} \mathbf{W}_{\bar{Z}} \mathbf{Z}_3 \right] \tag{6.10}$$

or,

$$\sum_{\mathsf{T}} \left(\boldsymbol{\alpha}^+ - \boldsymbol{\beta}^+\right) \mathbf{W}_{\bar{Z}} \mathbf{Z}_2 = \sum_{\mathsf{T}} \left(\boldsymbol{\alpha}^+ \mathbf{W}_{\bar{Z}} \mathbf{Z}_1 - \boldsymbol{\beta}^+ \mathbf{W}_{\bar{Z}} \mathbf{Z}_3\right) \tag{6.11}$$

where,

$$\boldsymbol{\alpha}^+ = \frac{1}{2}\left(I + \mathbf{M}_\alpha\right)\boldsymbol{\alpha} \tag{6.12}$$

$$\boldsymbol{\beta}^+ = \frac{1}{2}\left(I + \mathbf{M}_\beta\right)\boldsymbol{\beta} \tag{6.13}$$

Differentiating while freezing the Jacobians leads to,

$$\sum_{\mathsf{T}}\left[\left(\frac{\partial\boldsymbol{\alpha}^+}{\partial x_2} - \frac{\partial\boldsymbol{\beta}^+}{\partial x_2}\right)\mathbf{W}_{\bar{Z}}\mathbf{Z}_2 + \left(\boldsymbol{\alpha}^+ - \boldsymbol{\beta}^+\right)\mathbf{W}_{\bar{Z}}\frac{\partial\mathbf{Z}_2}{\partial x_2}\right] = \sum_{\mathsf{T}}\left(\frac{\partial\boldsymbol{\alpha}^+}{\partial x_2}\mathbf{W}_{\bar{Z}}\mathbf{Z}_1 - \frac{\partial\boldsymbol{\beta}^+}{\partial x_2}\mathbf{W}_{\bar{Z}}\mathbf{Z}_3\right) \tag{6.14}$$

$$\sum_{\mathsf{T}}\left(\boldsymbol{\alpha}^+ - \boldsymbol{\beta}^+\right)\mathbf{W}_{\bar{Z}}\frac{\partial\mathbf{Z}_2}{\partial x_2} = \sum_{\mathsf{T}}\left[\frac{\partial\boldsymbol{\alpha}^+}{\partial x_2}\mathbf{W}_{\bar{Z}}\left(\mathbf{Z}_1 - \mathbf{Z}_2\right) - \frac{\partial\boldsymbol{\beta}^+}{\partial x_2}\mathbf{W}_{\bar{Z}}\left(\mathbf{Z}_3 - \mathbf{Z}_2\right)\right] \tag{6.15}$$

$$\frac{\partial\mathbf{Z}_2}{\partial x_2} = -\left[\sum_{\mathsf{T}}\left(\boldsymbol{\alpha}^+ - \boldsymbol{\beta}^+\right)\mathbf{W}_{\bar{Z}}\right]^{-1}\sum_{\mathsf{T}}\left(\frac{\partial\boldsymbol{\alpha}^+}{\partial x_2}\widetilde{\Delta_\xi\mathbf{W}} + \frac{\partial\boldsymbol{\beta}^+}{\partial x_2}\widetilde{\Delta_\eta\mathbf{W}}\right) \tag{6.16}$$

Neglecting the subsonic blending on $\mathbf{M}_\alpha$ and $\mathbf{M}_\beta$ allows $\boldsymbol{\alpha}^+$ and $\boldsymbol{\beta}^+$ to be expressed as,

$$\boldsymbol{\alpha}^+ = \begin{cases}\boldsymbol{\alpha}, & \tilde{\mathcal{V}}_\alpha > 0\\ 0, & \tilde{\mathcal{V}}_\alpha \le 0\end{cases} \qquad \boldsymbol{\beta}^+ = \begin{cases}\boldsymbol{\beta}, & \tilde{\mathcal{V}}_\beta > 0\\ 0, & \tilde{\mathcal{V}}_\beta \le 0\end{cases} \tag{6.17}$$

with derivatives,

$$\frac{\partial\boldsymbol{\alpha}^+}{\partial x_2} = \begin{cases}\frac{\partial\boldsymbol{\alpha}}{\partial x_2}, & \tilde{\mathcal{V}}_\alpha > 0\\ 0, & \tilde{\mathcal{V}}_\alpha \le 0\end{cases} \qquad \frac{\partial\boldsymbol{\beta}^+}{\partial x_2} = \begin{cases}\frac{\partial\boldsymbol{\beta}}{\partial x_2}, & \tilde{\mathcal{V}}_\beta > 0\\ 0, & \tilde{\mathcal{V}}_\beta \le 0\end{cases} \tag{6.18}$$

The derivatives of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ remain as Eqns. 4.41 and 4.42.

Having determined $\frac{\partial\mathbf{Z}_2}{\partial x_2}$, $\frac{\partial\vec{V}_2}{\partial x_2}$ follows from,

$$du = \frac{d(\sqrt{\rho}u) - u\,d(\sqrt{\rho})}{\sqrt{\rho}} \qquad dv = \frac{d(\sqrt{\rho}v) - v\,d(\sqrt{\rho})}{\sqrt{\rho}} \tag{6.19}$$

The concept of fluctuation minimization extends directly to three dimensions for point deletion and insertion. The idea of edge swapping extends as face reconnections amongst tetrahedra in three dimensions which is more complicated due to the

increased degrees of freedom and geometric connection possibilities. The functional for nodal displacements, Eqn. 6.2, is written in vector notation and so can be applied directly to three dimensions. However, not only would the derivatives of the fluctuation splitting scheme need to be reformulated, but the basic fluctuation splitting scheme itself would need to be worked out in detail to move to three dimensions.

A three-dimensional alignment strategy has been employed by Beeman and Powers[116]. That method starts on an attached shock and projects downstream, trying to remain on the shock. This involves iterating an assumed shock shape and an implied body, iterating until the implied body matches the true surface. Open questions remain on how to start on detached shocks and how to handle embedded shocks. A significant drawback mentioned by the authors is that, "[The method] far enough downstream will eventually become unstable."

6.4 Mars Pathfinder

The demonstration case for the system adaption is borrowed from the Mach-10 wind tunnel tests of Hollis[115, 117, 118], which investigated the aerothermodynamic environment experienced by a payload in the wake of an aerobrake. For his dissertation, Hollis specifically looked at the effect of grid adaption on sting heating for his model. Using a structured-grid finite volume solver, a mesh of $125 \times 357 = 44{,}625$ nodes was required to get reasonable pressure and heating-rate agreement on the sting. Additionally, this fine mesh required extensive and time consuming adaption by hand to sufficiently resolve the wake flow.

Using the Hollis data as a benchmark, the present study seeks to achieve comparable results on an automatically-adapted unstructured mesh. Significant time savings can be achieved primarily by automating the adaption procedure, but also reducing the solver iteration and convergence times if the unstructured approach requires fewer grid nodes for the same resolution.

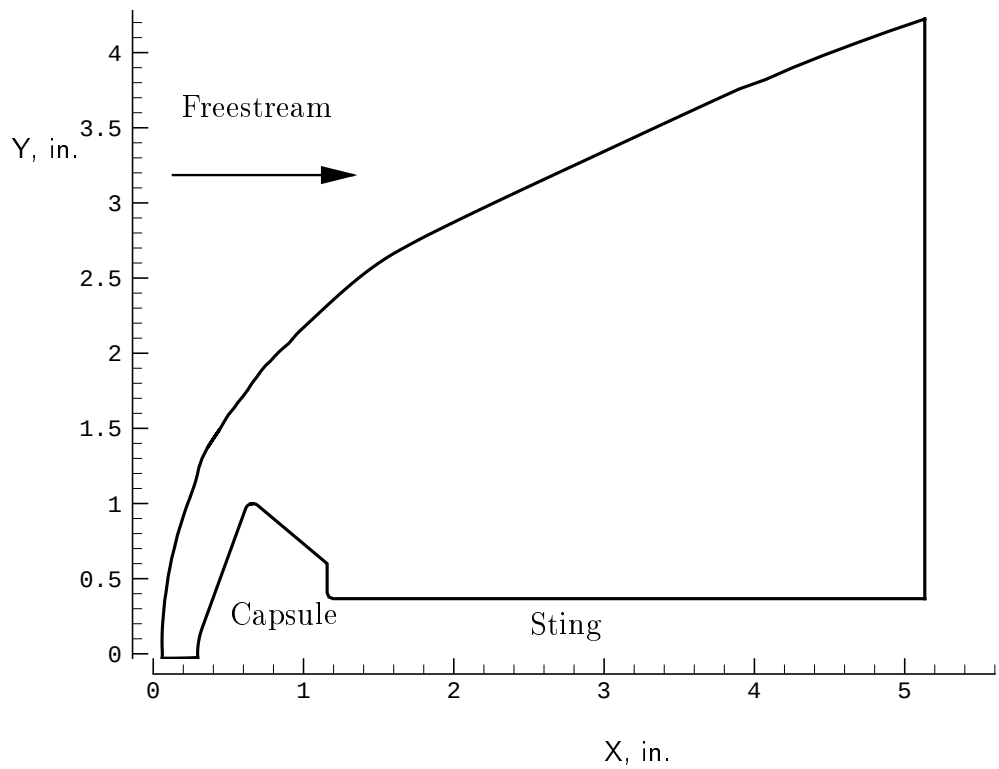


Figure 6.1: Computational domain for axisymmetric Mars Pathfinder capsule.

6.4.1 Configuration and Conditions

The axisymmetric wind tunnel model of the Mars Pathfinder capsule consists of a spherically-blunted 70-degree sphere-cone. The radius is 1 in. and the nose radius is $\frac{1}{2}$ in. The shoulder radius is $\frac{1}{20}$ in. and the aftbody angle is 40 degrees. The base radius is $\frac{3}{5}$ in. The sting is 4 in. long with a radius of $\frac{13}{32}$ in. Figure 6.1 shows the boundary of the computational domain.

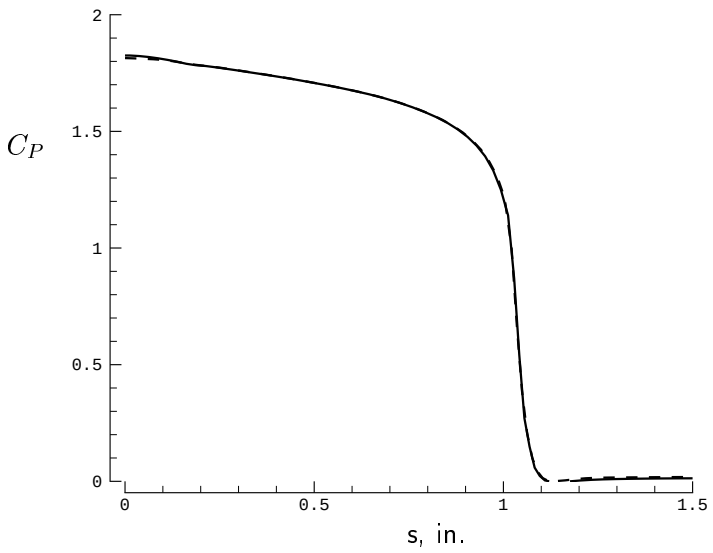
The freestream conditions for the NASA Langley 31-Inch Mach 10 Air Tunnel, corresponding to a nominal Reynolds number per foot of 0.5×10^6 ft^{-1}, are: $P = 69$ Pa, $T = 53$ K, $\rho = 0.0045$ kg/m^3, and $u = 1416$ m/s. The wall temperature is taken to be a uniform 300 K. Historical experience with this tunnel at these conditions indicate that laminar perfect-gas calculations are adequate for comparison with the experimental data.

In addition to the axisymmetric configuration, the present study also considers a two-dimensional version of the problem, using the same geometry and freestream conditions.

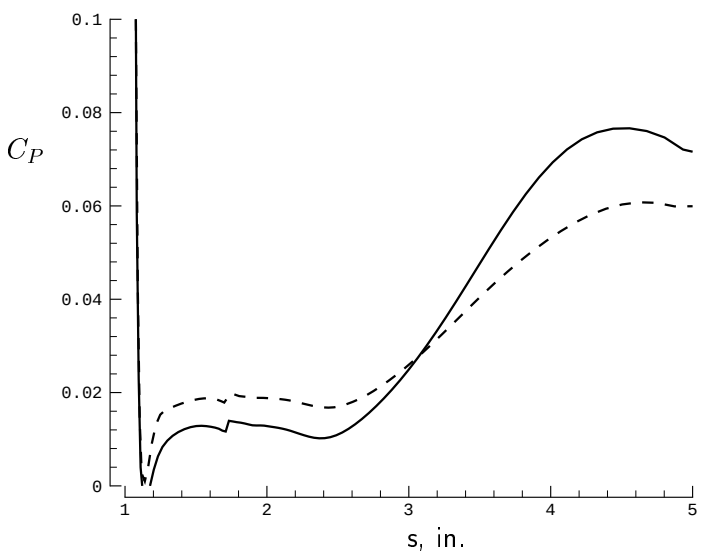
6.4.2 Benchmark Data

For the axisymmetric configuration Hollis has provided the experimental heat-transfer data along with numerical heat-transfer and surface pressure results. The numerical results were generated using the NEQ2D code of Candler[119]. A second set of numerical results were obtained for the present study using the LAURA code of Gnoffo[15]. Both of these codes are structured-grid finite volume solvers. Surface pressure coefficients from the numerical results are shown in Figure 6.2 and heat-transfers for all three datasets are in Figure 6.3, where the size of the symbols is indicative of the uncertainty in the experimental data reported by Hollis, $\pm 7\%$. The forebody pressures are in agreement between NEQ2D and LAURA, but the aftbody pressure agreement is weaker. The computed heating rates match the experimental data, aside from an irregularity in the LAURA result at the stagnation point, along the body to a running length of 3 inches, which is located on the sting. Hollis speculates that the free shear layer in the wake may be transitioning at the point where the experimental and computational heating rates diverge on the sting.

For the two-dimensional configuration, the axisymmetric grid was enlarged to be $125 \times 513 = 64,125$ nodes so as to capture the bow shock due to the greater shock stand-off distance. A LAURA solution was obtained for this case, and the results of grid convergence in the body-normal direction on pressure and heating are shown in Figures 6.4 and 6.5. Surface pressures are largely converged with 129 points in the body-normal direction, and certainly by 257 points. A similar statement applies for the heating, except in the vicinity of the stagnation point. The heating prediction at the nose continually increases with each grid doubling.

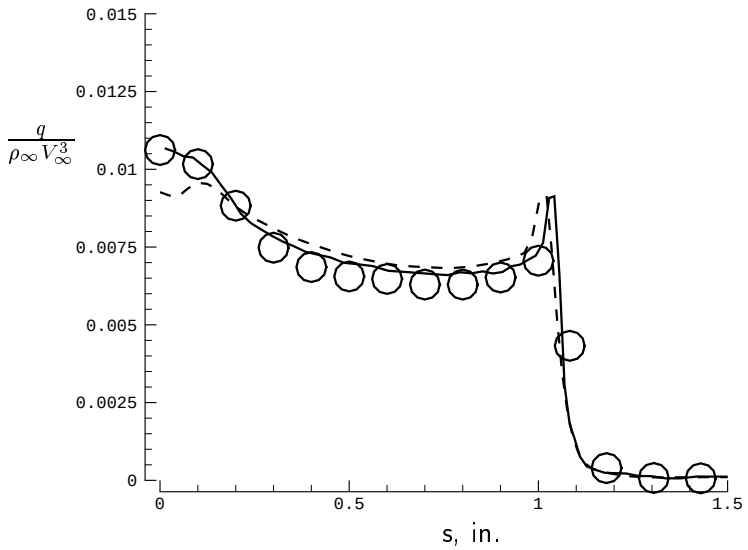


(a) Heatshield.

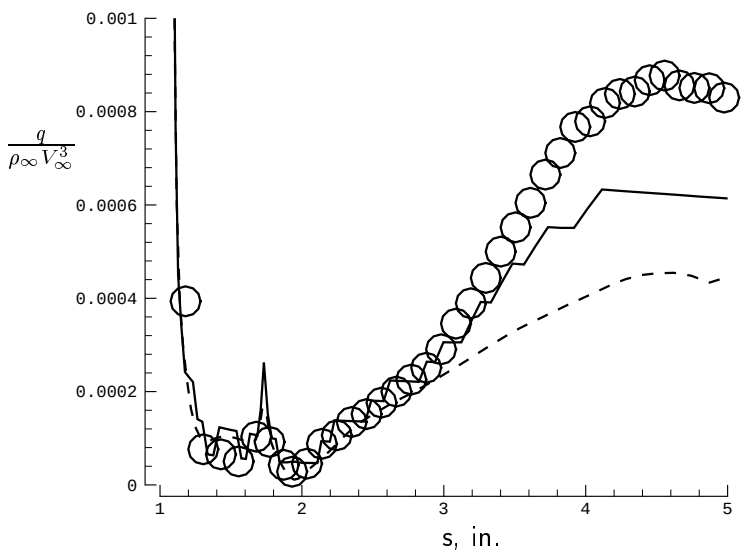


(b) Aftbody and sting.

Figure 6.2: Axisymmetric capsule benchmark surface pressures, solid=NEQ2D, dashed=LAURA.

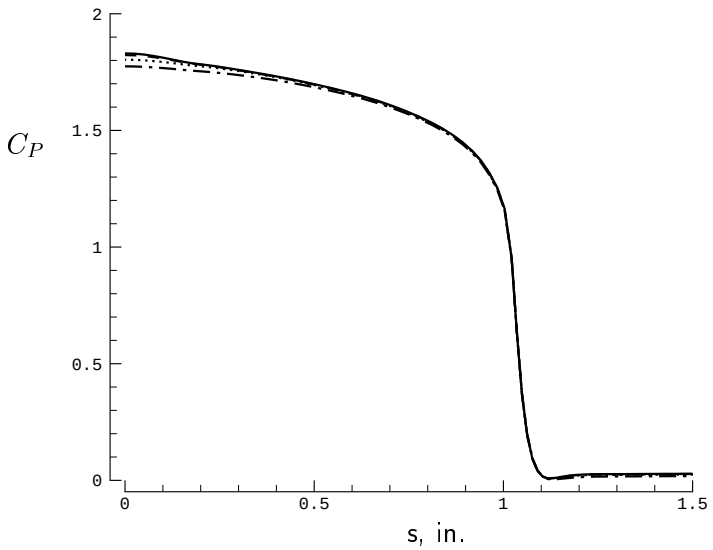


(a) Heatshield.

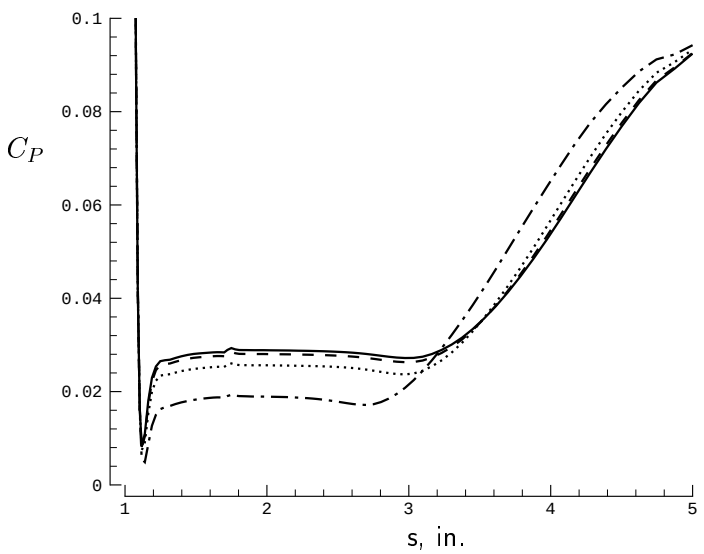


(b) Aftbody and sting.

Figure 6.3: Axisymmetric capsule benchmark surface heating, symbols=experiment, solid=NEQ2D, dashed=LAURA.

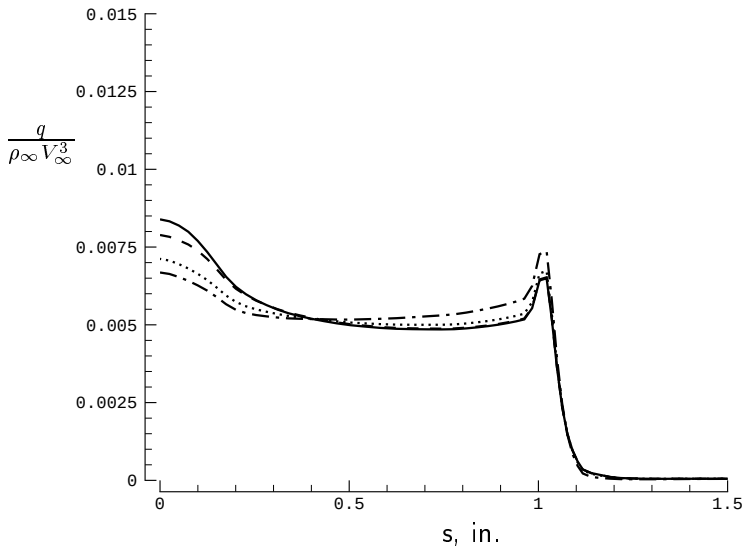


(a) Heatshield.

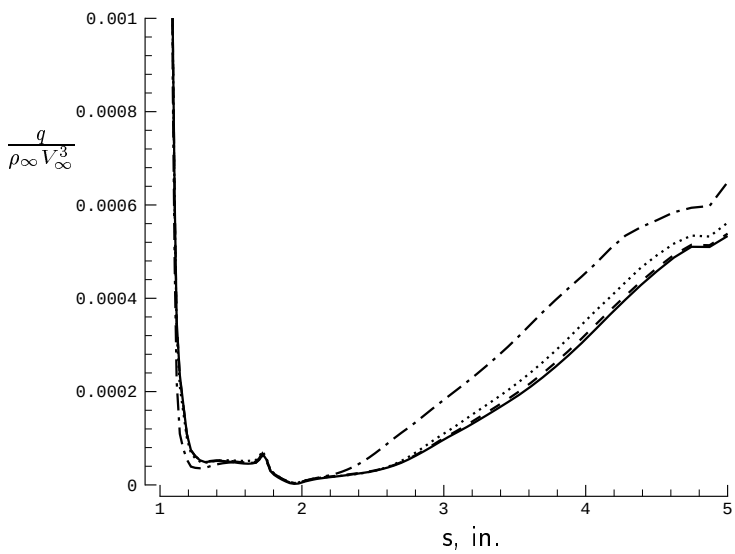


(b) Aftbody and sting.

Figure 6.4: Two-dimensional capsule benchmark surface pressures, solid=513, dashed=257, dotted=129, dash-dot=65 points normal to the surface.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.5: Two-dimensional capsule benchmark surface heating, solid=513, dashed=257, dotted=129, dash-dot=65 points normal to the surface.

6.4.3 Unadapted Baseline

Prior to performing any adaption, both unstructured DMFDSFV and fluctuation splitting schemes were used to obtain solutions on triangulated versions of the structured meshes, as an indication of a target accuracy for the adaption strategies. During this process, a number of problems and sensitivities were encountered.

As a general note the explicit single-stage time integration scheme used here is severely restricted in the stable timestep for high-Reynolds-number viscous problems. A structured-mesh solver, such as the LAURA code, would typically alleviate this stiffness by employing an implicit solver. Another option would be to use a parallelized strategy. However, both the development of an unstructured implicit solver for the present schemes and the parallelization of the schemes were considered beyond the scope of the current work. As a consequence, viscous solutions for the capsule on the fine meshes required weeks of CPU time on a desktop PC.

Since both the DMFDSFV and the fluctuation splitting schemes are implemented at the nodes, the timestep on the axisymmetric axis vanishes, slowing temporal convergence. The radial term in the timestep determination is chosen at the median-dual centroid for nodes on the axisymmetric axis, rather than at $y = 0$, allowing for a more realistic wave propagation on the axis. The solutions are observed to remain temporally stable with this implementation while yielding much faster convergence rates. Yet, the axisymmetric axis is always observed to converge slower than the rest of the flowfield, due to the radial weighting on the control volume. In general, axisymmetric solutions take two to three times longer to converge than an equivalent two-dimensional case because of this axis singularity effect.

The carbuncle effect, Figure 6.6, reared its ugly head for some cases, typically on meshes that are fine in the streamwise direction but very coarse in the body-normal direction. Although the carbuncle effect is a known deficiency[1] in the structured

[1]The carbuncle effect was first reported by Peery and Imlay[120] for a Mach-6 cylinder as an unexplainable "protuberance" in the bow shock at the symmetry plane. Later work by Roberts[121] and Quirk[122] reveal the mathematical basis for the production of spurious solutions for grid-aligned shocks with the Roe flux difference splitting scheme. As Quirk reports, "[P]arallel to the shock, Roe's scheme will not add any dissipation via the contact and shear waves, to counteract perturbations that appear through the acoustic waves; this appears to be a recurring theme whenever Roe's method fails. It is interesting to note that if Harten's entropy fix is applied to the contact and shear waves,

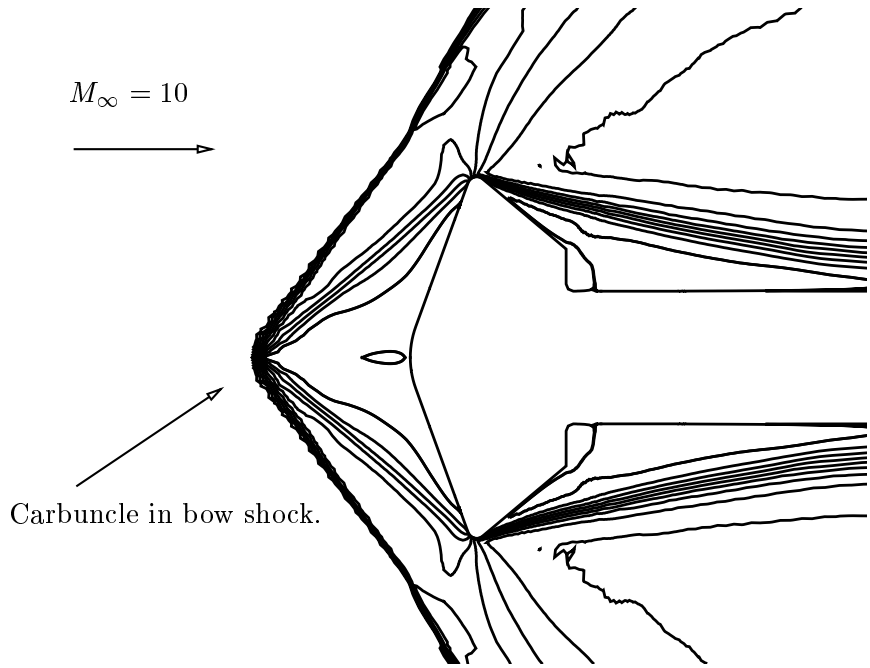


Figure 6.6: Fluctuation splitting u-velocity contours, showing carbuncle formation for two-dimensional capsule case.

Roe solver, its behavior in unstructured schemes has not been as well characterized. Lowering the CFL number by three orders of magnitude did not change the solution, verifying that the phenomenon was not instability induced. Similarly, re-running time-accurately produced the same behavior, strongly suggesting the phenomenon was not transient induced. The traditional fix for the Roe scheme involves adding dissipation through eigenvalue limiting. The present schemes were being run with thin-layer viscous terms and eigenvalue limiting on $V \pm a$ only. Switching to full viscous terms and limiting all eigenvalues both served to reduce the occurrence of the carbuncle effect, Figure 6.7.

The last problem encountered was that the surface heating at the stagnation point had the wrong trend, dropping toward zero heating. Figure 6.8 shows the forebody

any shortcoming of Roe's scheme is invariably cured. However, there is no justification, either physical or mathematical, for applying this fix to these waves, it is just a convenient method for introducing an amount of artificial dissipation into the scheme."

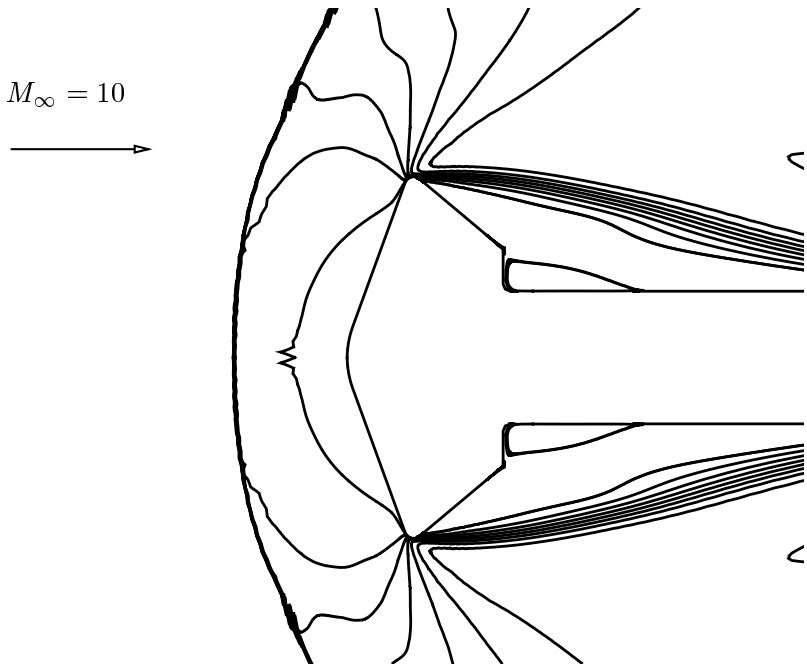


Figure 6.7: Fluctuation splitting u-velocity contours for two-dimensional capsule case.

heating on a two-dimensional 63×129 mesh using fluctuation splitting. Note the dramatic drop in heating at the stagnation point, contrasted with the benchmark data of Figure 6.5(a). Looking into the flowfield for an explanation of the heating trend a recirculation bubble is found at the nose, Figure 6.9, which physically should not be present. The cause of this stagnation recirculation proved extremely time consuming to identify and it was unexpectedly found to be a grid-induced feature. The unstructured meshes for the unadapted cases were initially created by a simple triangulation of the structured grids. All quadrilateral cells were cut in the same direction, and this bias in the diagonals along the axis boundary produced a shift in the stagnation point off the axis. Repeating the calculations on a biased grid covering both the upper and lower half-planes removed the vortex but still shifted the stagnation point. The cure for this problem was to remove the grid bias by alternating the diagonals in the derived unstructured meshes, yielding the vortex-free solution in Figure 6.10 for the same case as is shown in Figure 6.9.

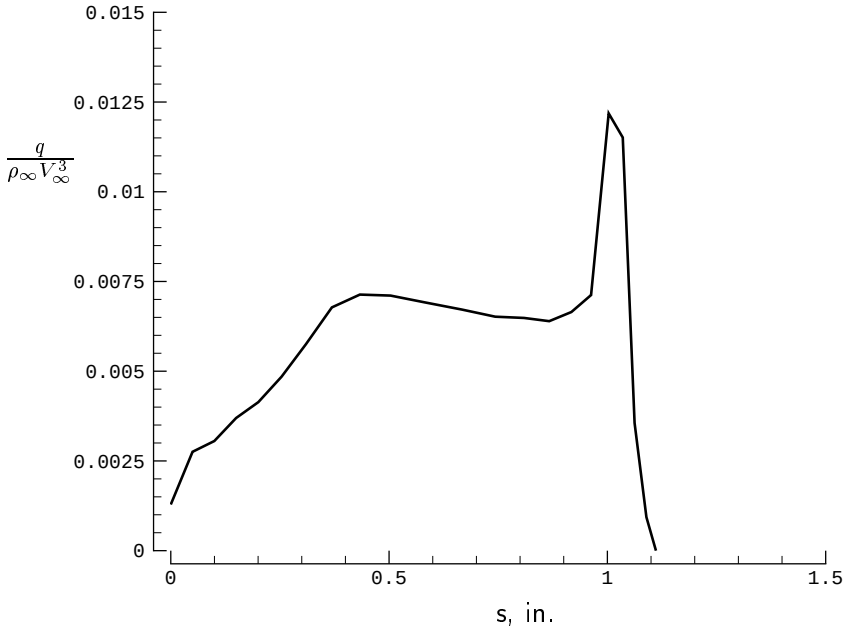


Figure 6.8: Incorrect forebody heating trend, two-dimensional fluctuation splitting.

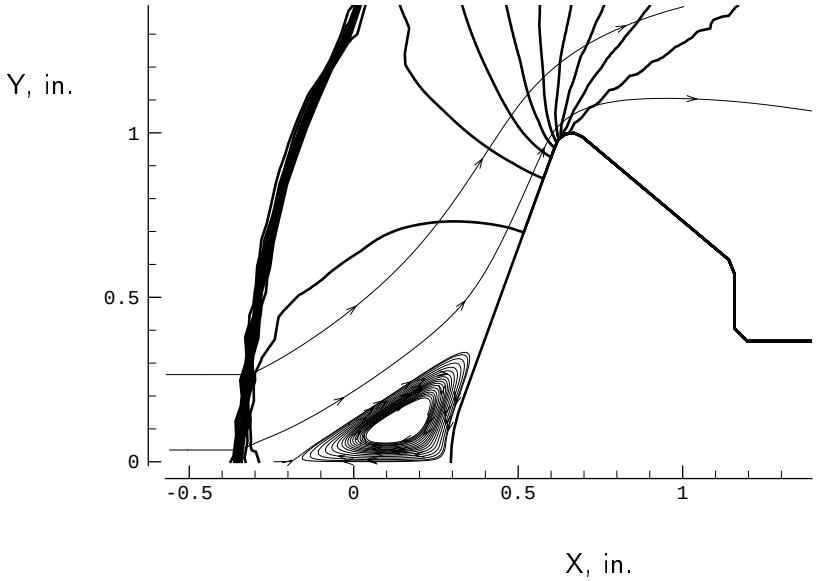


Figure 6.9: Stagnation point vortex.

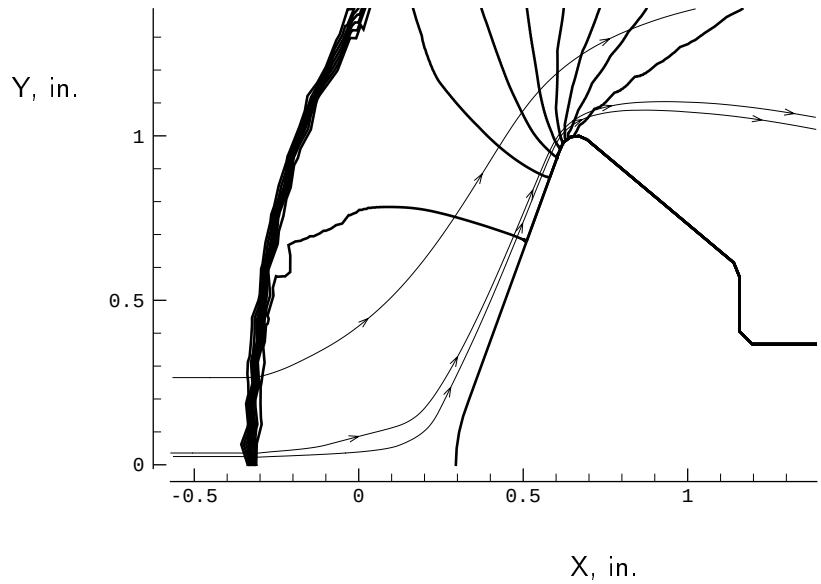


Figure 6.10: Correct streamlines in stagnation region.

32×65	32×129		
63×65	63×129	63×257	
125×65	125×129	125×257	125×513

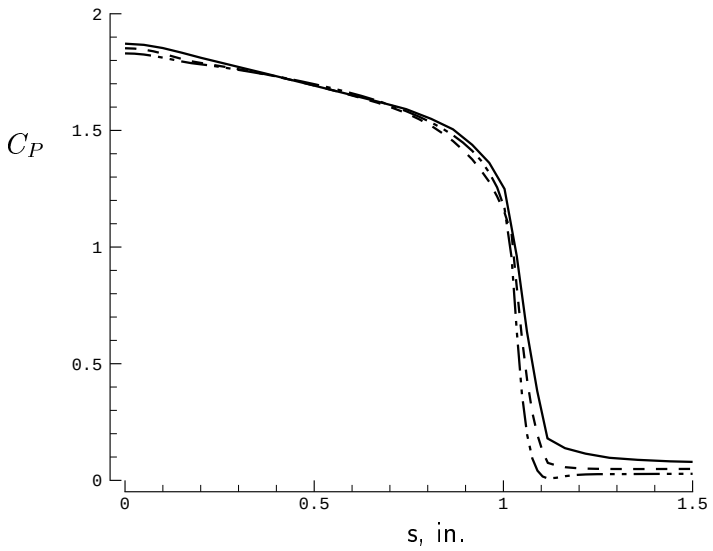
Table 6.1: Matrix of grid dimensions for two-dimensional baseline cases.

Two-dimensional

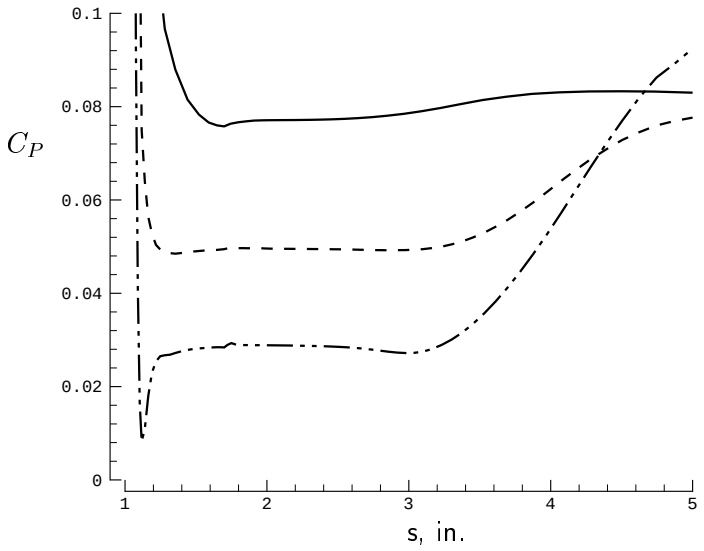
Viscous solutions for the two-dimensional capsule were computed using both fluctuation splitting and DMFDSFV discretizations on a sequence of unadapted grids as listed in Table 6.1.

Results using only 32 points to define the surface were generally under-resolved for both schemes. Results for 63 and 125 surface points are compared in Figures 6.11–6.14 using 257 points normal to the body.

DMFDSFV surface pressures, Figure 6.11, overlap the benchmark results on the forebody for both grids. Aftbody pressure agreement is poor for 63 surface points. The solution improves by going to 125 points, though the pressure coefficient is still over-predicted by a factor of 2. The heating trends for DMFDSFV are reversed

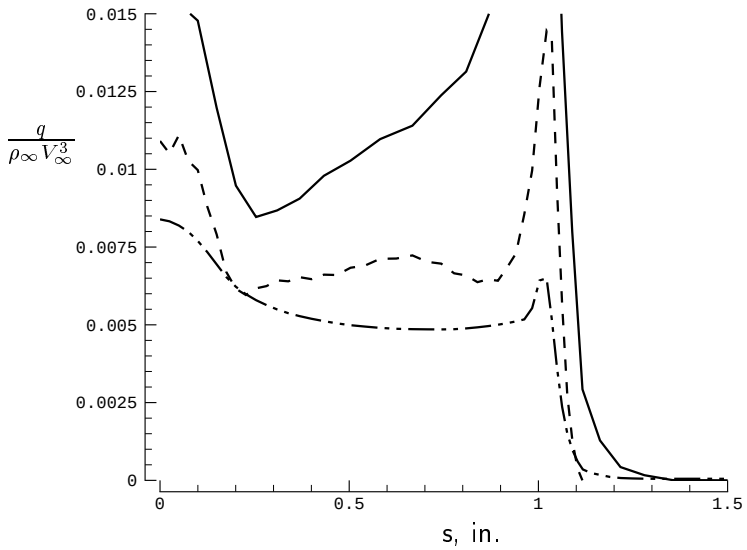


(a) Heatshield.

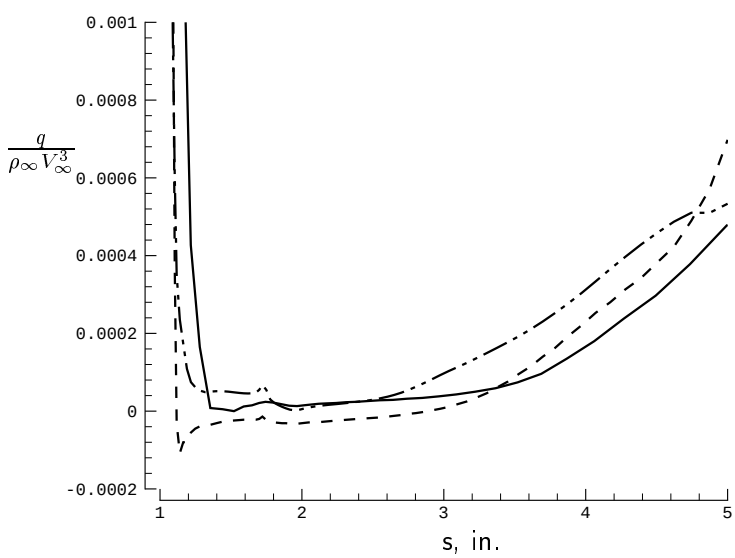


(b) Aftbody and sting.

Figure 6.11: Two-dimensional capsule surface pressures: i-refinement with DMFDS-FV, solid=63, dashed=125, dash-dot-dot=benchmark.

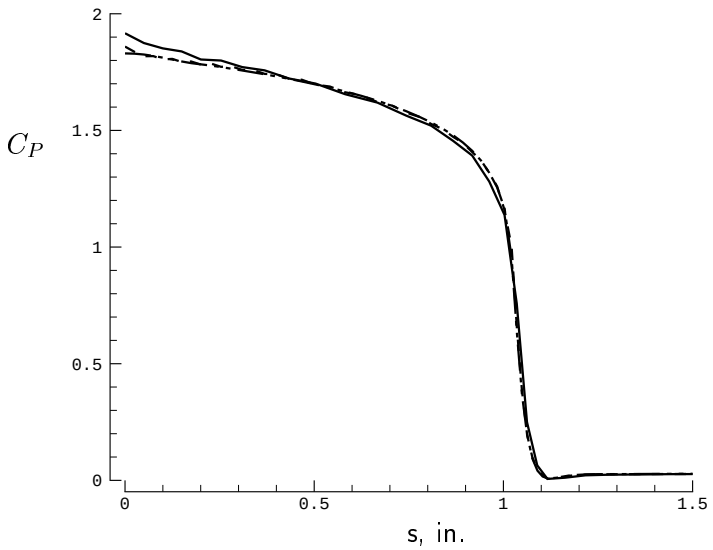


(a) Heatshield.

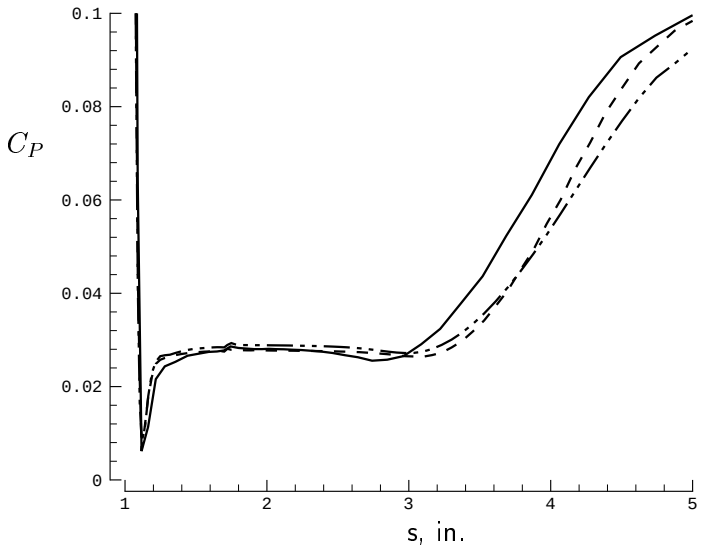


(b) Aftbody and sting.

Figure 6.12: Two-dimensional capsule surface heating: i-refinement with DMFDSFV, solid=63, dashed=125, dash-dot-dot=benchmark.

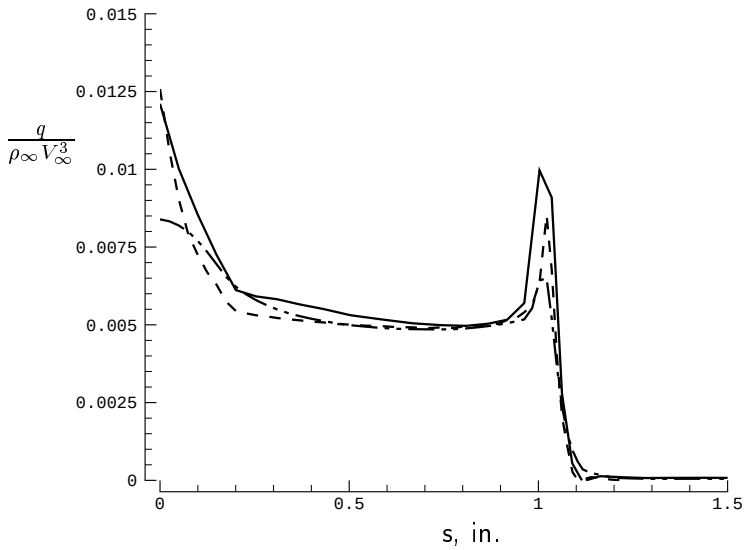


(a) Heatshield.

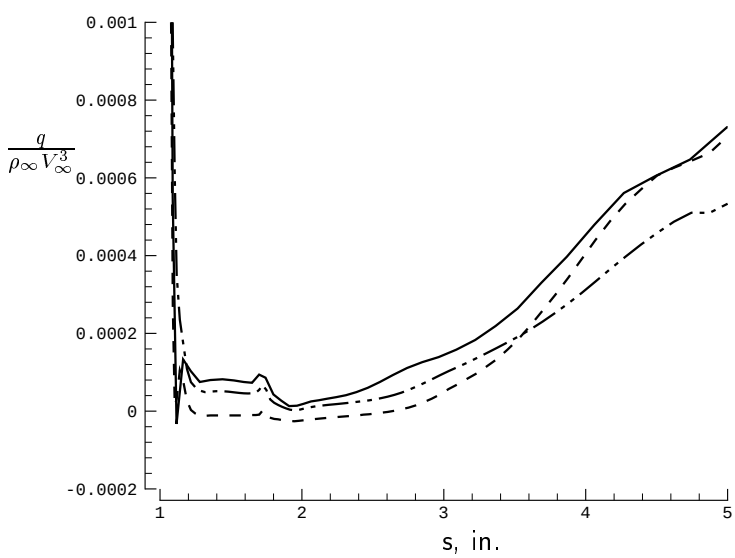


(b) Aftbody and sting.

Figure 6.13: Two-dimensional capsule surface pressures: i-refinement with fluctuation splitting, solid=63, dashed=125, dash-dot-dot=benchmark.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.14: Two-dimensional capsule surface heating: i-refinement with fluctuation splitting, solid=63, dashed=125, dash-dot-dot=benchmark.

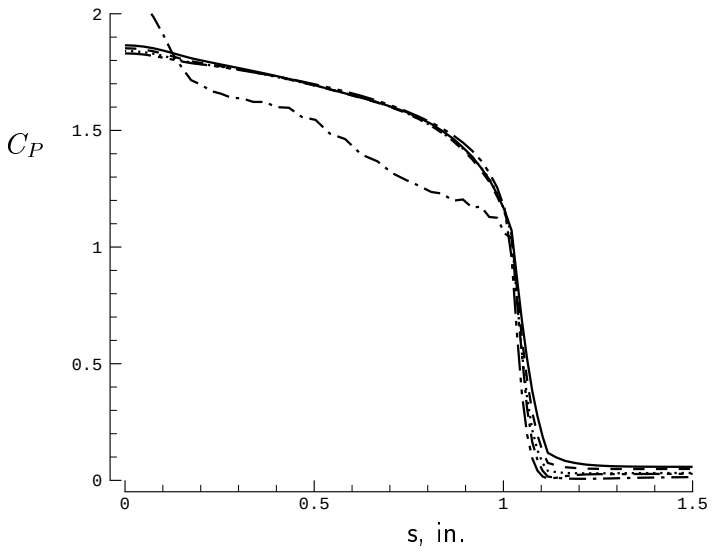
from the pressure trends. Figure 6.12 shows that the coarser-mesh forebody heating is in poor agreement with the benchmark. The finer mesh improves the heating prediction, but is still more than 20 percent higher than the benchmark over much of the heatshield, especially at the shoulder. The aftbody and sting heating rates are reasonable on both meshes.

Fluctuation splitting surface pressures, Figure 6.13, are in very good agreement with the benchmark data, both fore and aft. The 125 surface point mesh is in particularly good agreement, over-plotting the benchmark over all but the tail end of the sting. Fluctuation splitting surface heating, Figure 6.14, is in good agreement for both meshes, with the finer mesh providing better resolution at the shoulder. The glaring weakness in the fluctuation splitting solutions is at the stagnation point, where the heating spikes 40 percent higher than the benchmark. Further, the known correct heating trend at a stagnation point calls for the heating to level off, as the benchmark results do, rather than spike, as with the present results.

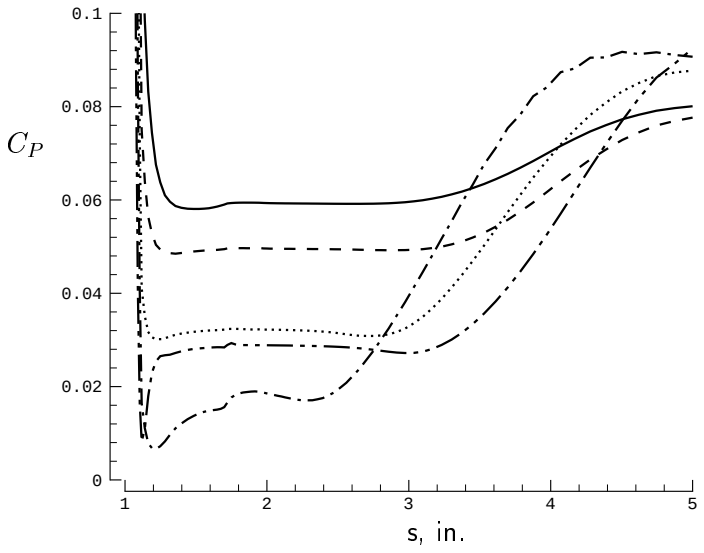
Grid convergence trends in the body-normal direction, for 125 surface points, are shown in Figures 6.15–6.18.

The DMFDSFV surface pressure grid convergence is shown in Figure 6.15. On the forebody excellent agreement with the benchmark is seen for all but the coarsest mesh. On the aftbody and sting, only the solution from the mesh with 129 points normal to the body is in good agreement with the benchmark. The trend with refining the mesh is toward increasing pressures on the aftbody, and the solutions do not appear to be converging toward the benchmark result. The DMFDSFV heating with grid refinement is shown in Figure 6.16. On the heatshield the solution from the j=129 mesh provides the closest match to the benchmark, while the 257-point result is not too much worse. The solution from the coarsest mesh is wildly unresolved and the result from the finest mesh is disappointing. On the aftbody, again the coarsest mesh is under-resolved. The three finer meshes do appear to be converging with grid refinement and have reasonable agreement with the benchmark.

The fluctuation splitting surface pressures, Figure 6.17, show good grid convergence trends, with both of the two finest meshes matching the benchmark over the entire body, and none of the grids produced terrible results. Fluctuation splitting

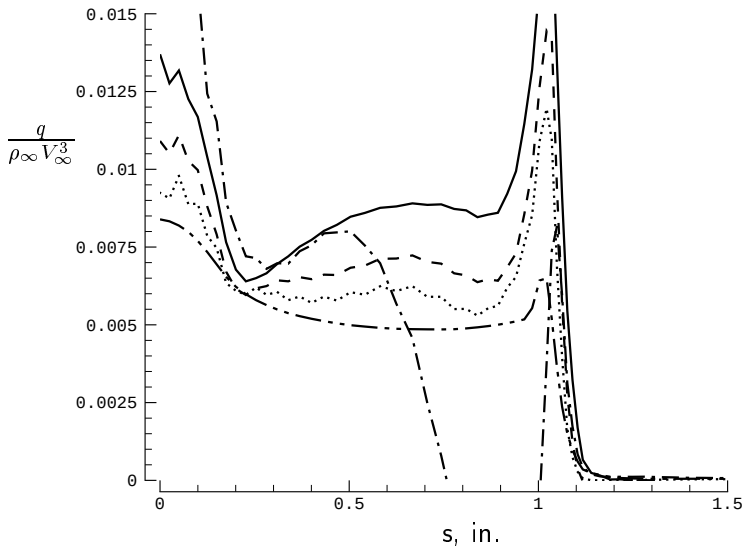


(a) Heatshield.

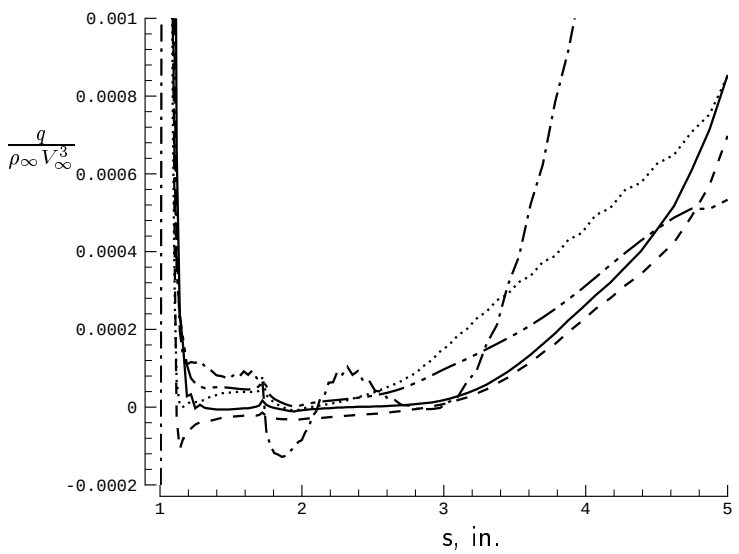


(b) Aftbody and sting.

Figure 6.15: Two-dimensional capsule surface pressures: j-refinement with DMFDS-FV, solid=513, dashed=257, dotted=129, dash-dot=65, dash-dot-dot=benchmark.

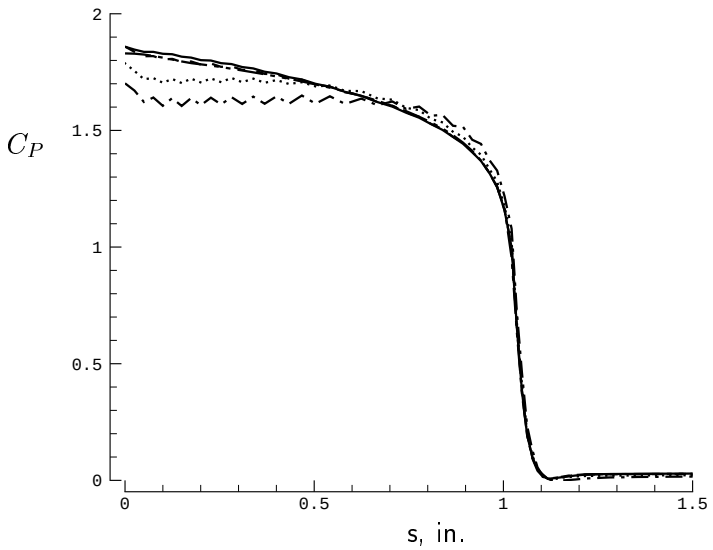


(a) Heatshield.

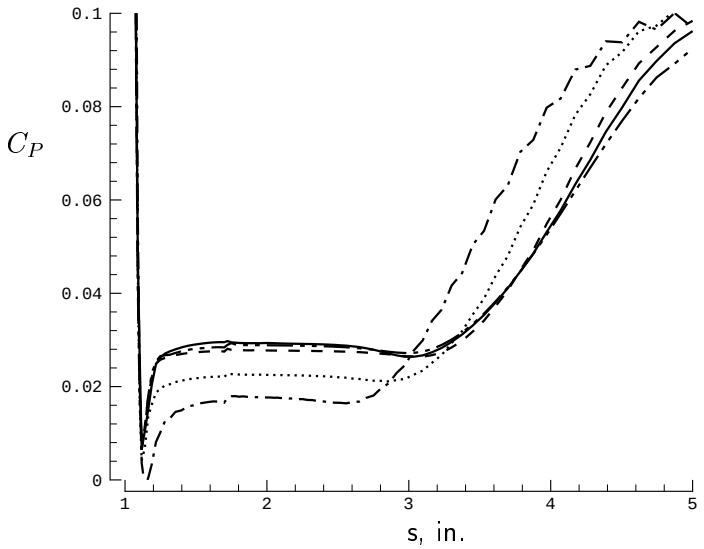


(b) Aftbody and sting.

Figure 6.16: Two-dimensional capsule surface heating: j-refinement with DMFDSFV, solid=513, dashed=257, dotted=129, dash-dot=65, dash-dot-dot=benchmark.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.17: Two-dimensional capsule surface pressures: j-refinement with fluctuation splitting, solid=513, dashed=257, dotted=129, dash-dot=65, dash-dot-dot=benchmark.

32×45	32×89		
63×45	63×89	63×177	
125×45	125×89	125×177	125×353

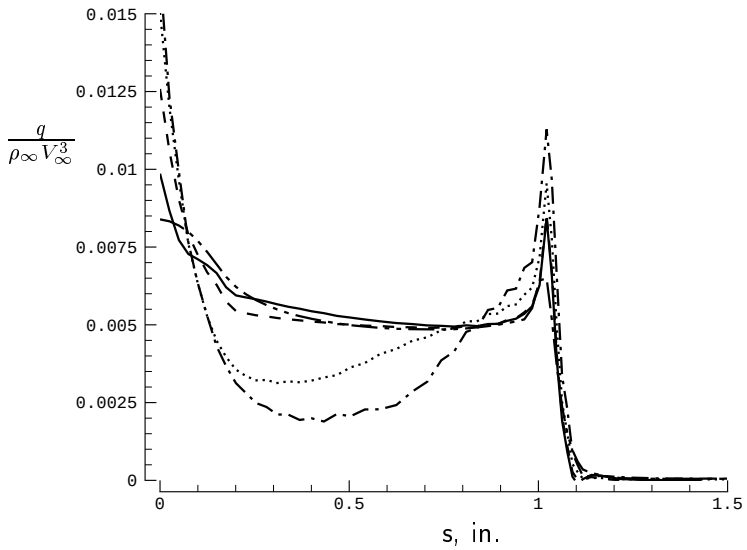
Table 6.2: Matrix of grid dimensions for axisymmetric baseline cases.

heatshield heating, Figure 6.18, also shows good grid convergence trends, with both of the two finest-mesh solutions matching the benchmark well except at the stagnation point, where the 513-point results do better but still have a spike of 15 percent at the symmetry line. Heating results in the wake region also show some grid convergence trends, though the finest mesh resolves an additional wake vortex on the sting, producing an inflection in the data. The coarser meshes actually match the benchmark better on the aftbody, before elevating on the sting where the finer meshes, and the 257-point result in particular, do a better job of matching the benchmark.

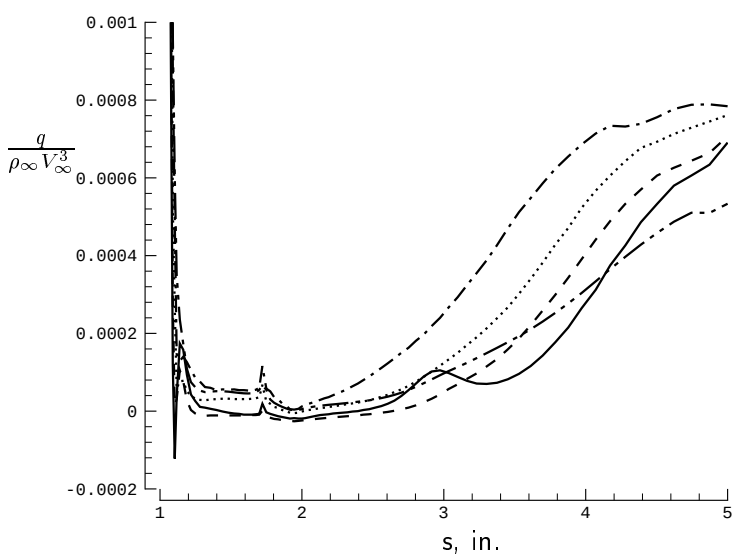
The two-dimensional unadapted baseline results are chosen to be the solutions on the triangulated 125×257 mesh. The fluctuation splitting solutions on this mesh were as good as on the finest mesh but with half the points. For DMFDSFV the 125×129 mesh produced results generally as good as or better than the baseline, though the results are difficult to interpret due to the lack of clear grid-convergence trends. The baseline surface pressures and heating rates are shown in Figures 6.19 and 6.20. Both solvers match forebody pressure with the benchmark. Fluctuation splitting also matches the benchmark pressure on the sting, while the DMFDSFV baseline over-predicts the pressure in this region. The fluctuation splitting forebody heating matches the benchmark well except as previously discussed at the stagnation point. The DMFDSFV forebody agreement is not as good. However, both schemes yield comparable aftbody heating rates.

Axisymmetric

Viscous solutions for the axisymmetric capsule configuration were obtained on a sequence of unadapted meshes formed as triangulations from the structured meshes used for the benchmark computations. The sequence of grids is listed in Table 6.2. All of the axisymmetric solutions to be shown suffer from irregular heat transfer

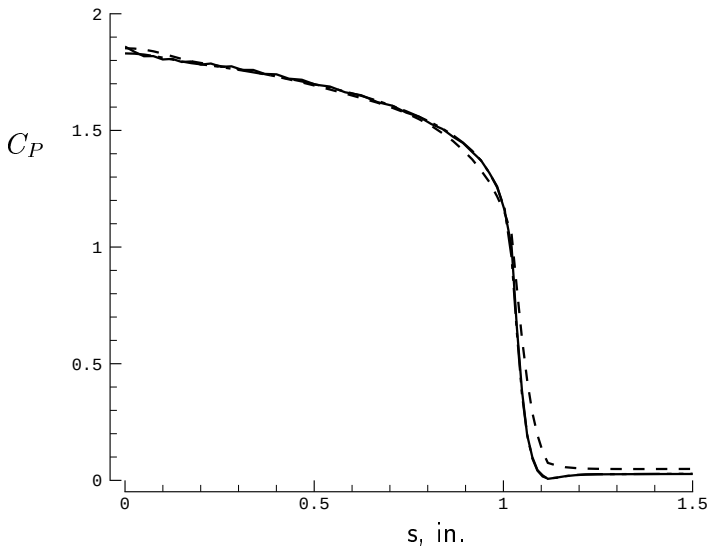


(a) Heatshield.

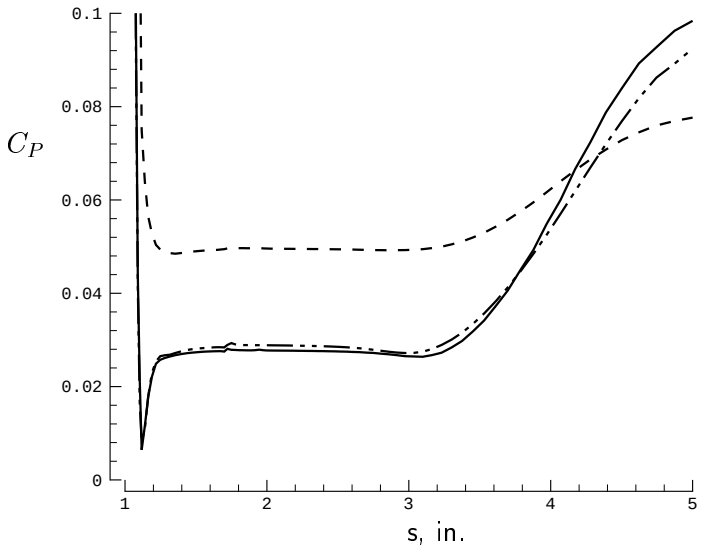


(b) Aftbody and sting.

Figure 6.18: Two-dimensional capsule surface heating: j-refinement with fluctuation splitting, solid=513, dashed=257, dotted=129, dash-dot=65, dash-dot-dot=benchmark.

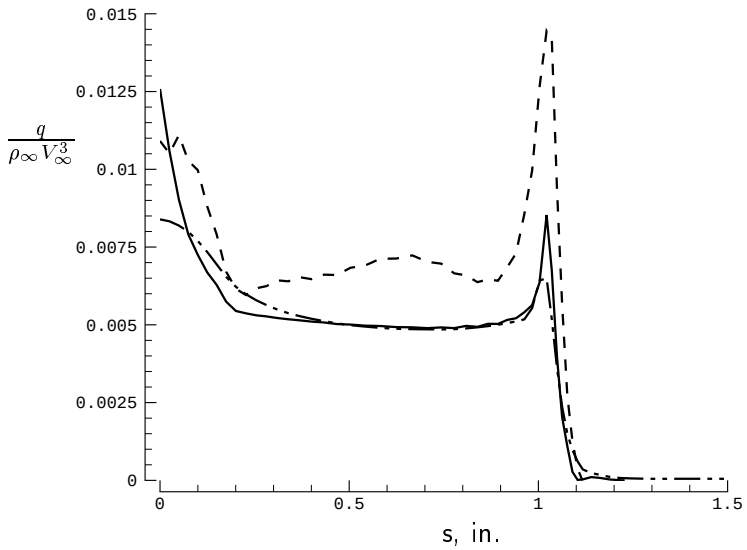


(a) Heatshield.

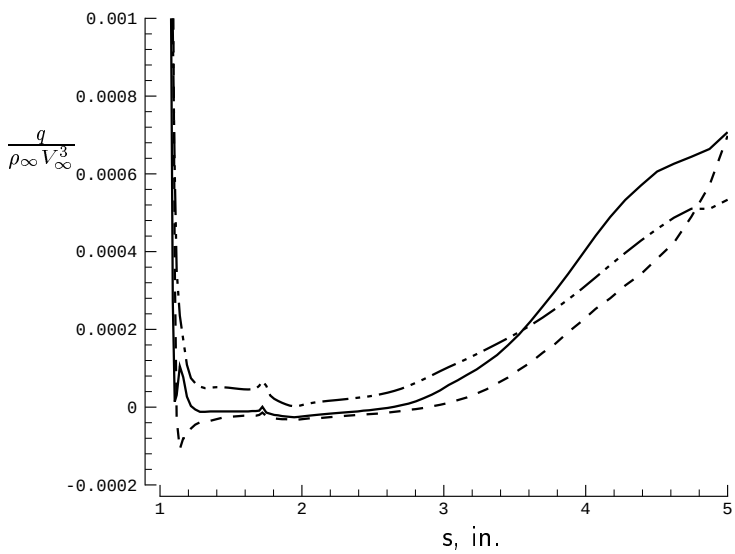


(b) Aftbody and sting.

Figure 6.19: Two-dimensional baseline capsule surface pressures, solid=fluctuation splitting, dashed=DMFDSFV, dash-dot-dot=benchmark.



(a) Heatshield.



(b) Aftbody and sting.

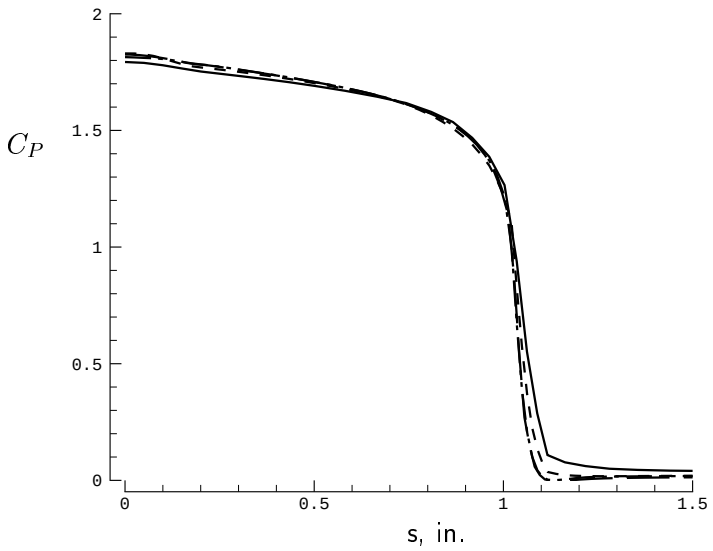
Figure 6.20: Two-dimensional baseline capsule surface heating, solid=fluctuation splitting, dashed=DMFDSFV, dash-dot-dot=benchmark.

patterns at the axis stagnation point, true for both the DMFDSFV and fluctuation splitting formulations as here implemented. The stagnation point surface pressures, however, are well behaved. The difficulty with the heating appears to be a result of applying the weak form of the boundary conditions at both a mathematical singularity, the axisymmetric axis, and a surface flow singularity, the stagnation point, for the node-based schemes. Cell-based, rather than node-based, schemes or strong boundary enforcement formulations might not experience similar stagnation point heating difficulties for the axisymmetric cases. These heating irregularities are confined to the stagnation node and immediate neighbor nodes.

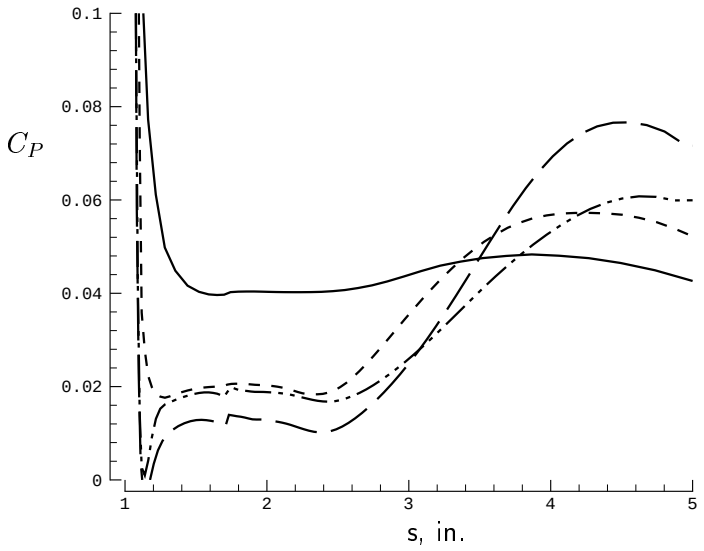
As with the two-dimensional results, the axisymmetric solutions using only 32 points to define the surface were under resolved. Results using 63 and 125 points to define the surface are presented in Figures 6.21–6.24, all with 89 points in the body-normal direction.

DMFDSFV surface pressures, Figure 6.21, overlap the benchmark results on the forebody, especially on the finer mesh. Aftbody pressures are poorly resolved on the coarser mesh. The finer mesh reasonably matches the benchmark except right at the shoulder where the overexpansion is missed. DMFDSFV heating trends with streamwise refinement are shown in Figure 6.22. The forebody heating is much improved on the finer mesh, though still 20 percent higher than the benchmark over most of the heatshield. The stagnation point heating is over-predicted by as much as 40 percent on both meshes and the peak heating on the shoulder is greatly over-predicted. However, on the aftbody and sting the finer mesh results are in very good agreement with the benchmark results. The coarser mesh under-predicts the wake heating.

Fluctuation splitting surface pressures for streamwise grid refinement are shown in Figure 6.23. The forebody pressures are the same for both meshes, but are 3–5 percent lower than the benchmark, except at the stagnation point where the benchmark is matched. Aftbody pressures match the benchmark extremely well for both grids. Forebody heating with fluctuation splitting, Figure 6.24, matches the benchmark fairly well except for over-predicting the shoulder heating spike and the stagnation point maximum. The finer mesh does a better job at the shoulder but results from neither mesh are particularly encouraging at the stagnation point. The fluctuation

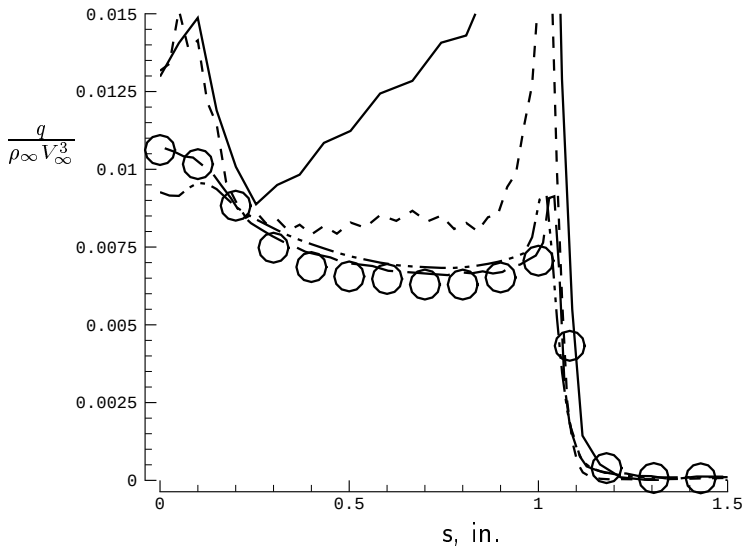


(a) Heatshield.

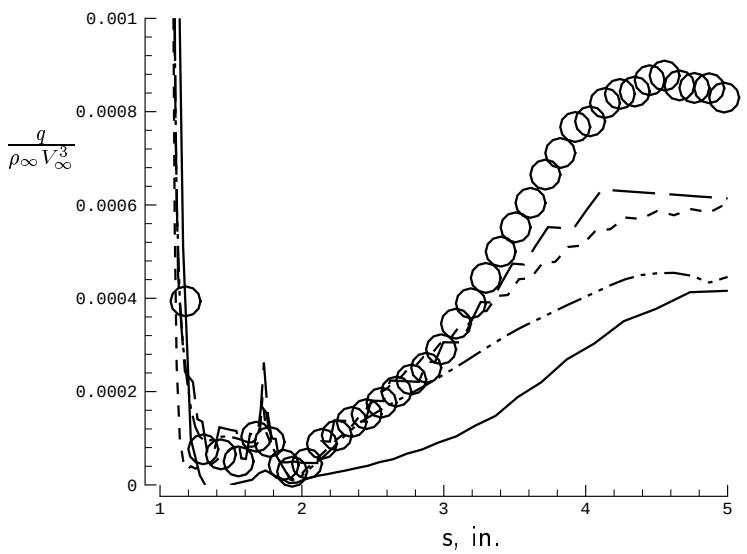


(b) Aftbody and sting.

Figure 6.21: Axisymmetric capsule surface pressures: i-refinement with DMFDSFV, solid=63, dashed=125, long-dash=NEQ2D, dash-dot-dot=LAURA.

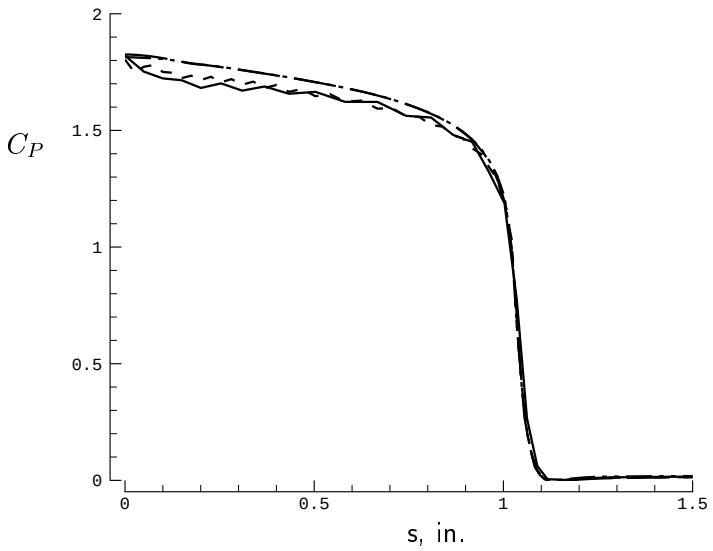


(a) Heatshield.

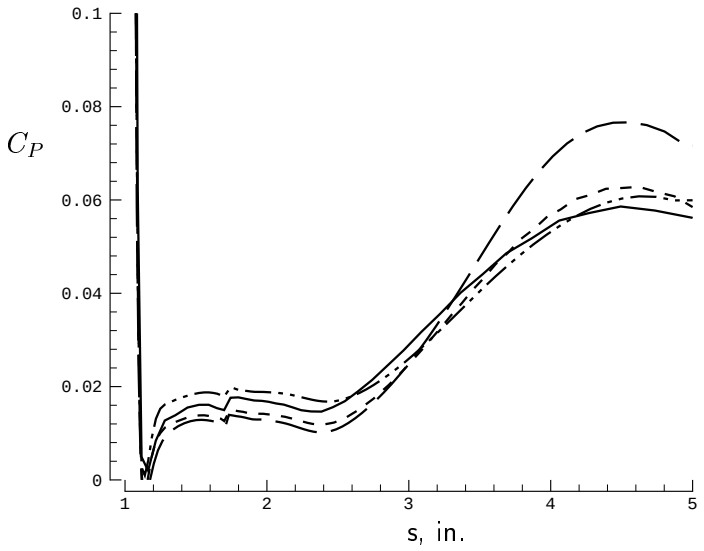


(b) Aftbody and sting.

Figure 6.22: Axisymmetric capsule surface heating: i-refinement with DM-FDSFV, solid=63, dashed=125, long-dash=NEQ2D, dash-dot-dot=LAURA, circles=experiment.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.23: Axisymmetric capsule surface pressures: i-refinement with fluctuation splitting, solid=63, dashed=125, long-dash=NEQ2D, dash-dot-dot=LAURA.

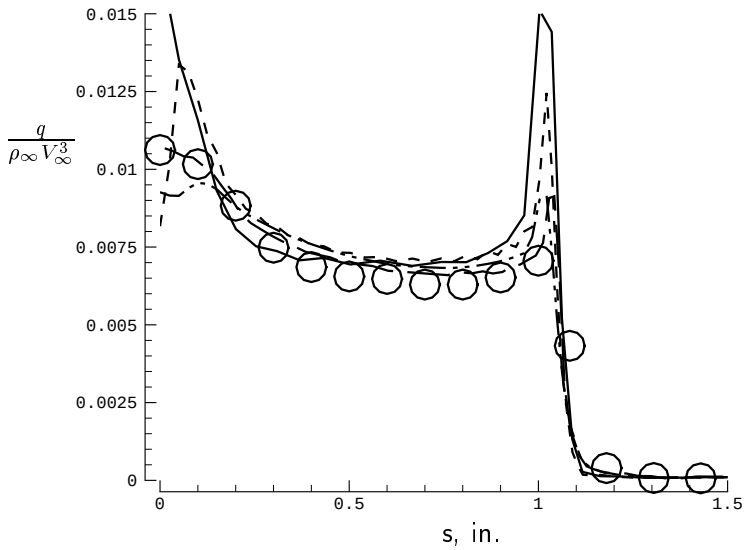
splitting heating results on both meshes provide excellent matches with the benchmark data on the aftbody and sting.

Grid convergence trends in the body-normal direction, for 125 surface points, are shown in Figures 6.25–6.28.

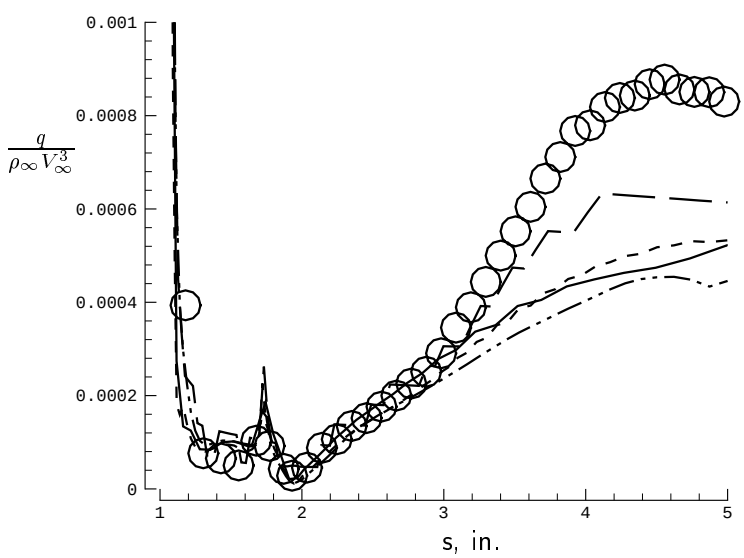
DMFDSFV forebody pressures match the benchmark on all grids, Figure 6.25. However, the aftbody and sting pressures do not show grid convergence. The best match with the benchmark comes on the grid with 89 points normal to the body. It is disappointing that the pressures deteriorate on the finer meshes, with the details being washed out perhaps by a different separation pattern in the wake. The DMFDSFV forebody heating, Figure 6.26, also fails to show a good grid convergence trend, with the results deteriorating on the finer meshes. The two coarser meshes provide the best agreement, with the 89-point results being smoother than the 45-point solution. Still, the forebody heating is over-predicted by 20 percent over most of the heatshield and by more at the stagnation point. On the aftbody and sting only the 89-point solution provides good agreement with the benchmark data.

Surface pressure trends with grid refinement in the body-normal direction using fluctuation splitting are shown in Figure 6.27. The forebody pressure is grid converged on all meshes, but slightly under-predicts the benchmark results. Aftbody and sting pressures on all meshes show excellent agreement with the benchmark solutions. Forebody heating with fluctuation splitting, Figure 6.28, shows good agreement over most of the heatshield for all but the 45-point mesh, which itself is only 20 percent elevated. As usual the stagnation point heating is largely in error. Heating in the wake shows good agreement for all but the finest mesh, where the heating on the sting shows an unexpected inflection point.

The axisymmetric baseline solutions are chosen to be the results on the triangulated 125×89 mesh. The fluctuation splitting solutions are consistently grid converged on this mesh, and the DMFDSFV results on the mesh are generally the best, often better even than on the finer meshes. The axisymmetric baseline pressures are presented in Figure 6.29. On the heatshield the DMFDSFV solution overlaps the benchmark pressures, while the fluctuation splitting result is steadily 3 percent low.

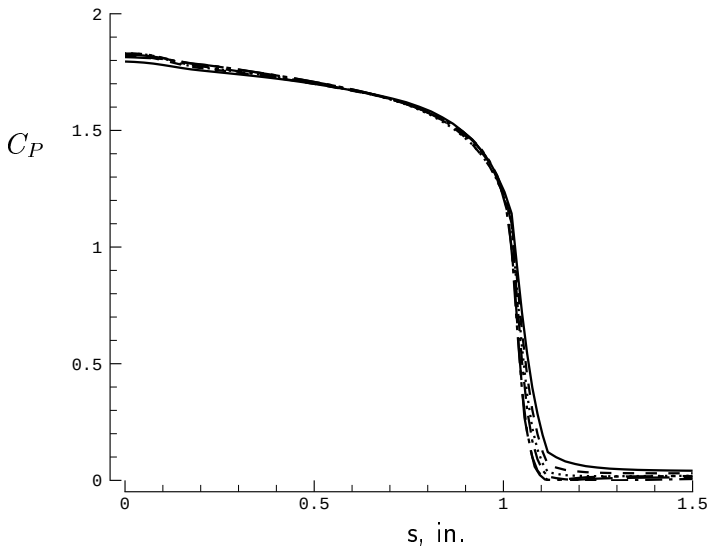


(a) Heatshield.

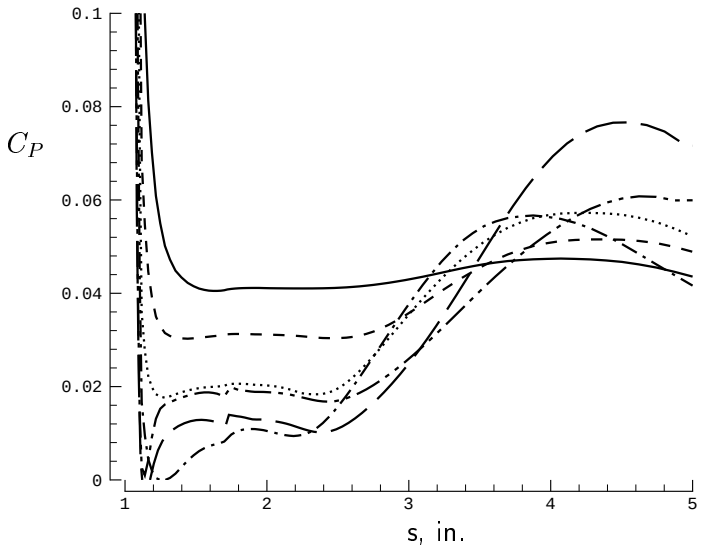


(b) Aftbody and sting.

Figure 6.24: Axisymmetric capsule surface heating: i-refinement with fluctuation splitting, solid=63, dashed=125, long-dash=NEQ2D, dash-dot-dot=LAURA, circles=experiment.

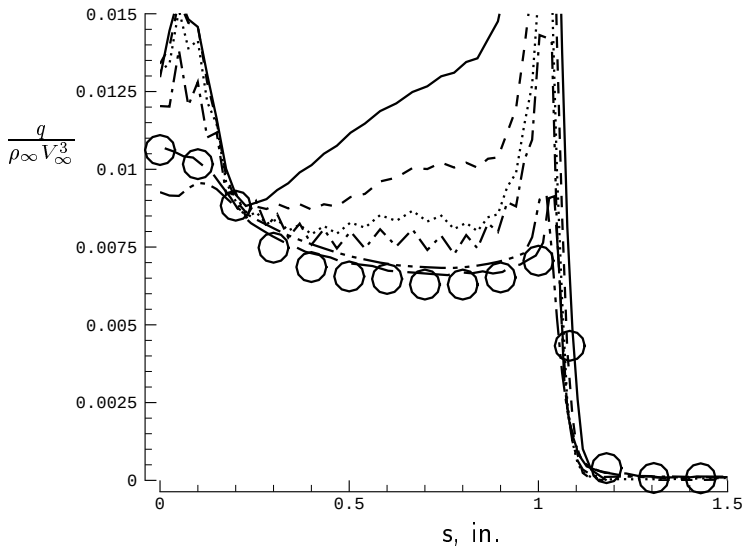


(a) Heatshield.

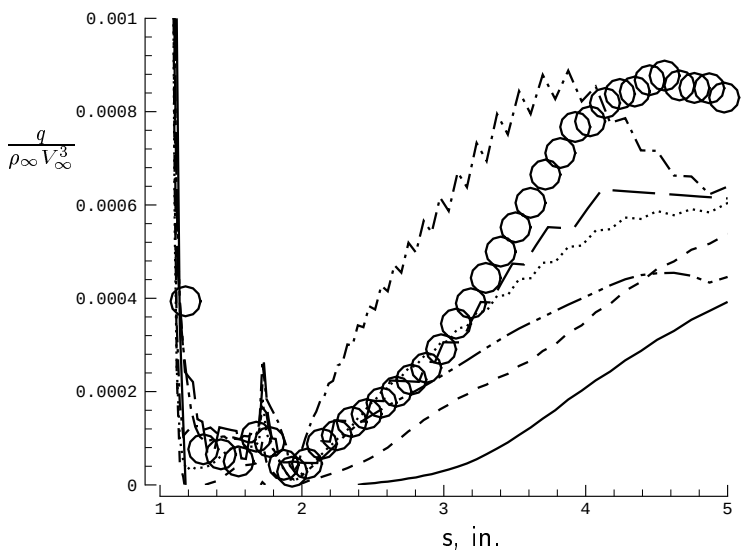


(b) Aftbody and sting.

Figure 6.25: Axisymmetric capsule surface pressures: j-refinement with DMFDS-FV, solid=353, dashed=177, dotted=89, dash-dot=45, long-dash=NEQ2D, dash-dot-dot=LAURA.

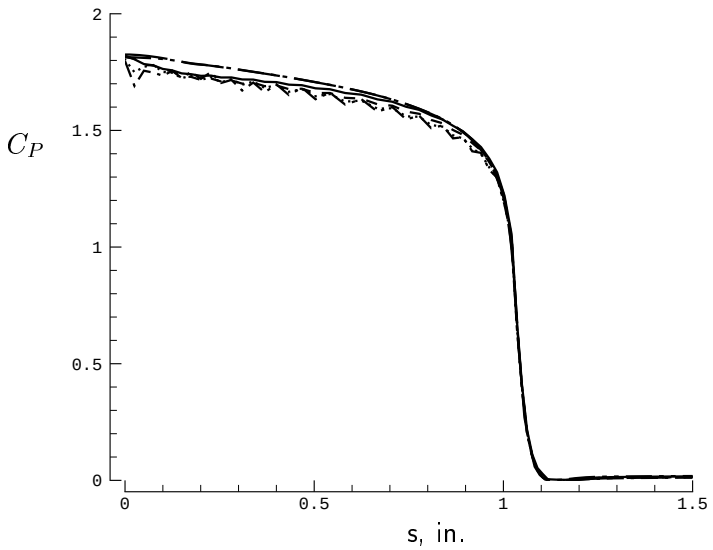


(a) Heatshield.

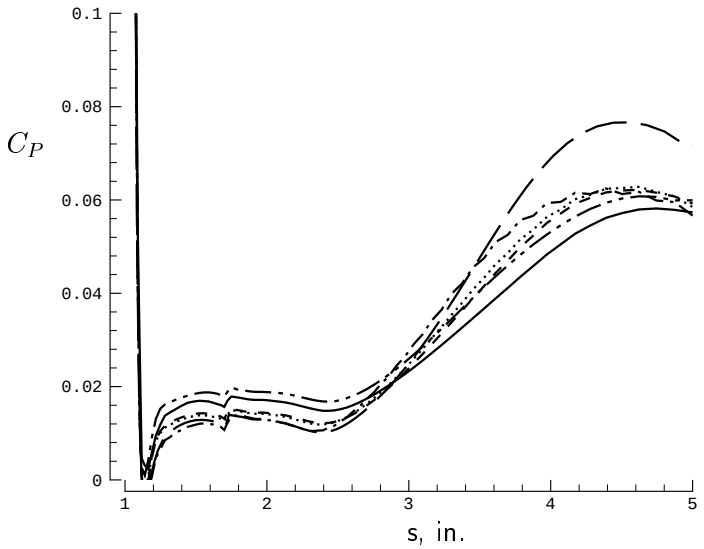


(b) Aftbody and sting.

Figure 6.26: Axisymmetric capsule surface heating: j-refinement with DMFDS-FV, solid=353, dashed=177, dotted=89, dash-dot=45, long-dash=NEQ2D, dash-dot-dot=LAURA, circles=experiment.

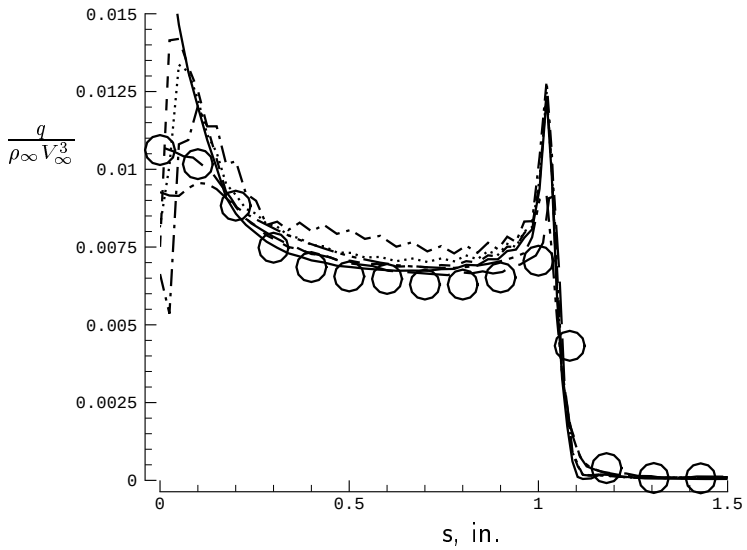


(a) Heatshield.

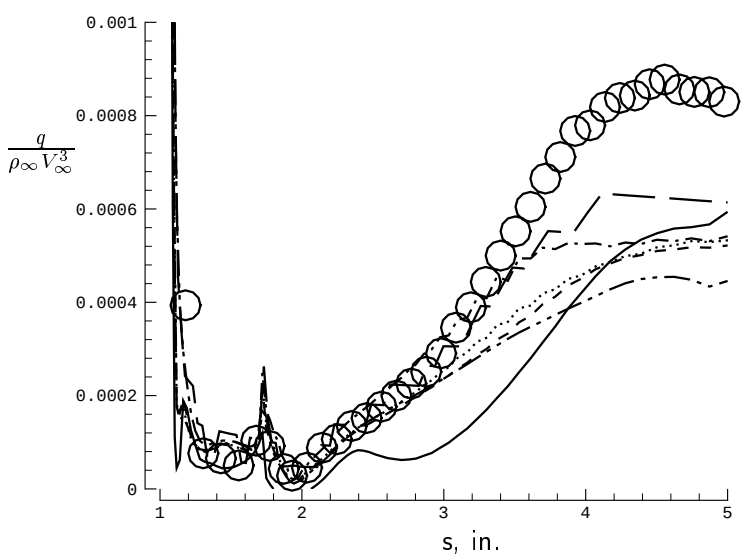


(b) Aftbody and sting.

Figure 6.27: Axisymmetric capsule surface pressures: j-refinement with fluctuation splitting, solid=353, dashed=177, dotted=89, dash-dot=45, long-dash=NEQ2D, dash-dot-dot=LAURA.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.28: Axisymmetric capsule surface heating: j-refinement with fluctuation splitting, solid=353, dashed=177, dotted=89, dash-dot=45, long-dash=NEQ2D, dash-dot-dot=LAURA, circles=experiment.

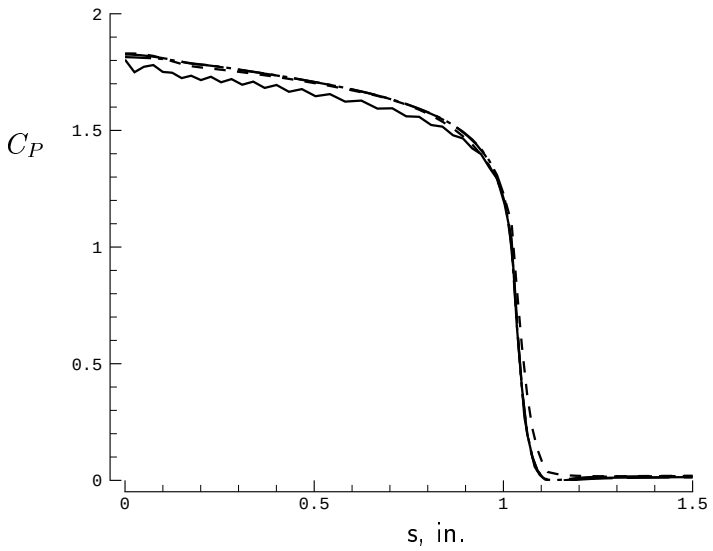
On the aftbody and sting it is the fluctuation splitting solution that provides an excellent match to the benchmark. The DMFDSFV solution is generally good except for missing the overexpansion pressure drop at the shoulder, seen as the pressure minimum at $s = 1.2$. The baseline surface heat transfer rates are shown in Figure 6.30. The fluctuation splitting heating on the forebody is in very good agreement with the benchmark data except at the stagnation point, where the match is poor. The DMFDSFV result generally over-predicts the forebody heating, except for a region between $s = 0.2$–0.4 near the nose, where the agreement with the benchmark data is very good. Heating levels at the stagnation point and shoulder are both greatly over-predicted. Both schemes match the benchmark heating datasets on the aftbody and sting.

6.4.4 Solution-Adapted Results

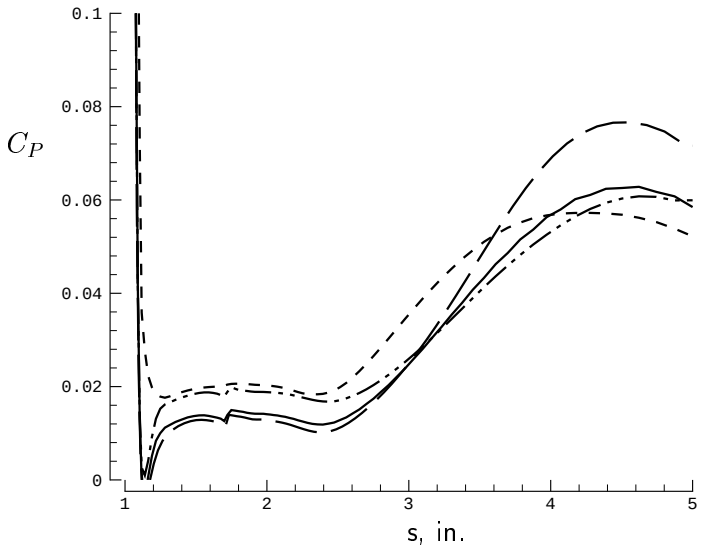
The adaption mechanisms, point deletion, edge swapping, nodal displacements, and point insertion, are applied one at a time, using both the curvature clustering with DMFDSFV and the fluctuation minimization with fluctuation splitting, to assess the effect of each individual component. Then a full adaption cycle is applied using a strategy based upon the results of the individual adaption tests.

Two-dimensional

For point deletion the starting solution is the converged baseline. The baseline mesh has 32,125 nodes, and the objective is to remove points while maintaining the accuracy of the baseline solution. Both schemes were able to remove 10 percent of the nodes (3284 nodes for fluctuation minimization and 3366 for curvature clustering) causing minimal change to the solution. The nodes were predominantly removed from the freestream. A further 10 percent of the nodes were removed (3157 for fluctuation minimization and 3076 for curvature clustering), for a total of 20 percent of the nodes deleted from the initial grid. The heating rates for both schemes are essentially unchanged, Figures 6.31 and 6.32, but both schemes have trouble maintaining the bow shock capture due to the further loss of freestream points, as shown in Figure 6.33 for

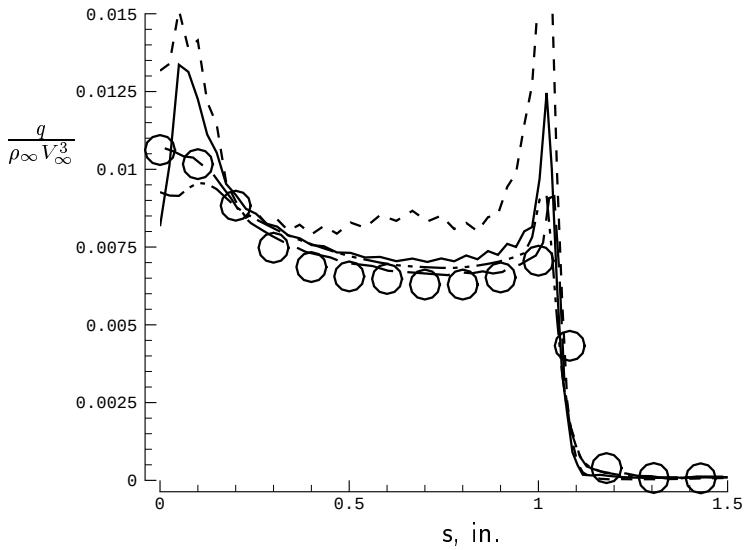


(a) Heatshield.

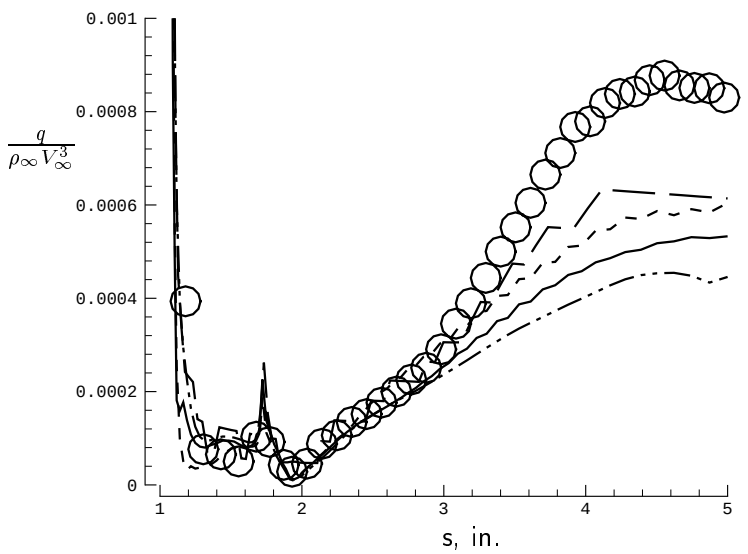


(b) Aftbody and sting.

Figure 6.29: Axisymmetric baseline capsule surface pressures, solid=fluctuation splitting, dashed=DMFDSFV, long-dash=NEQ2D, dash-dot-dot=LAURA.

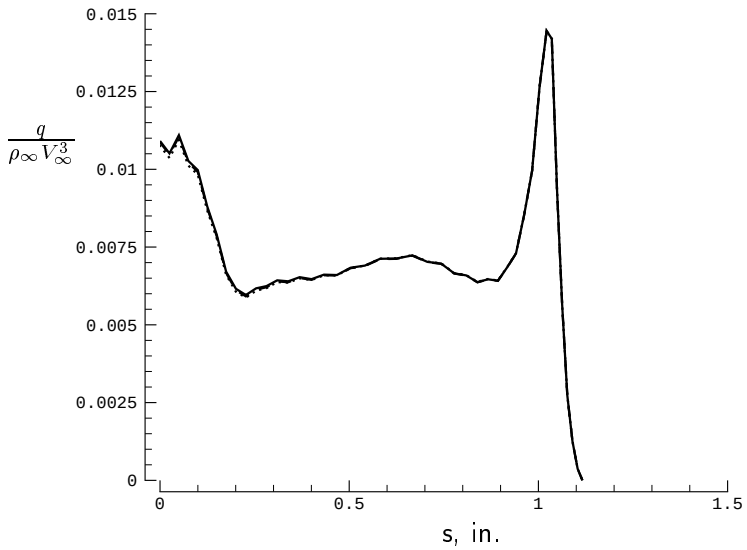


(a) Heatshield.

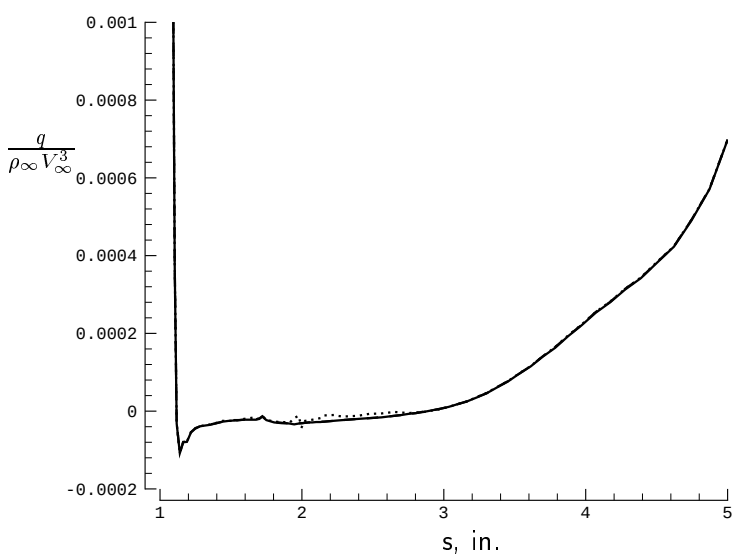


(b) Aftbody and sting.

Figure 6.30: Axisymmetric baseline capsule surface heating, solid=fluctuation splitting, dashed=DMFDSFV, long-dash=NEQ2D, dash-dot-dot=LAURA, circles=experiment.

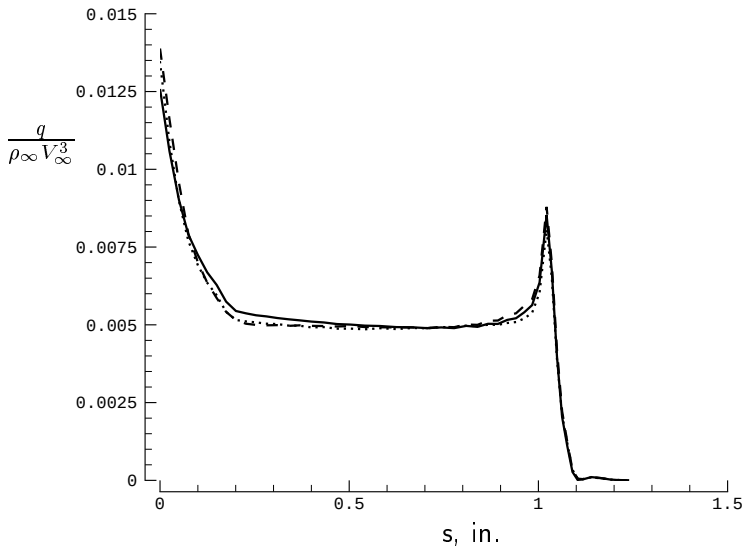


(a) Heatshield.

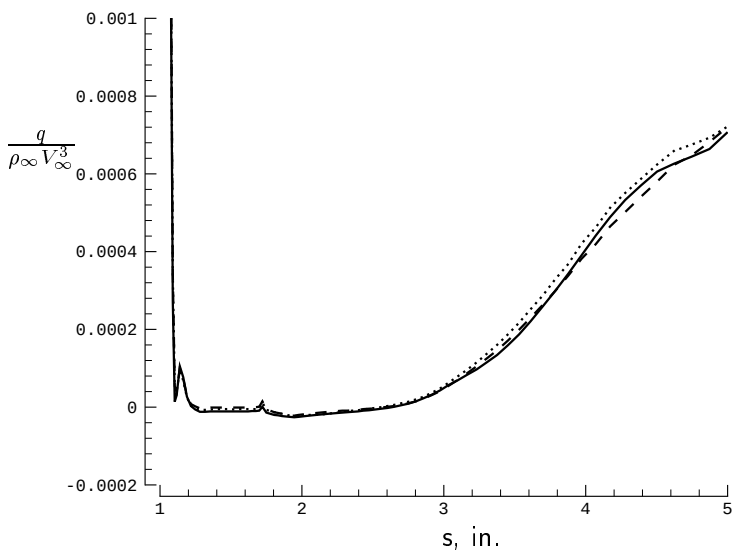


(b) Aftbody and sting.

Figure 6.31: Two-dimensional capsule surface heating after coarsening with curvature clustering, solid=baseline, dashed=10% removed, dotted=20% removed.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.32: Two-dimensional capsule surface heating after coarsening with fluctuation minimization, solid=baseline, dashed=10% removed, dotted=20% removed.

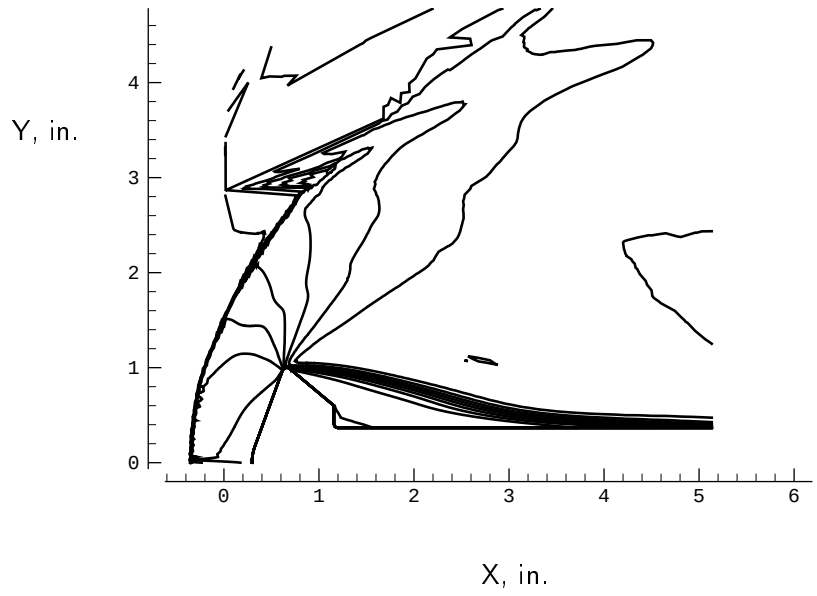
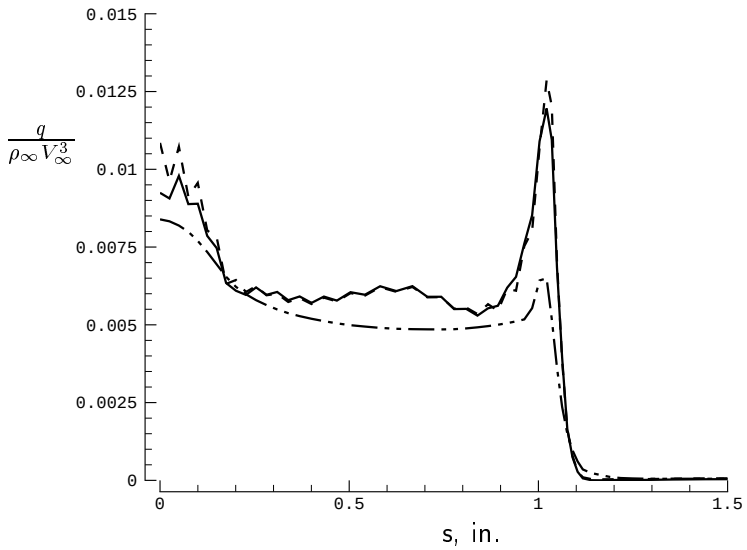


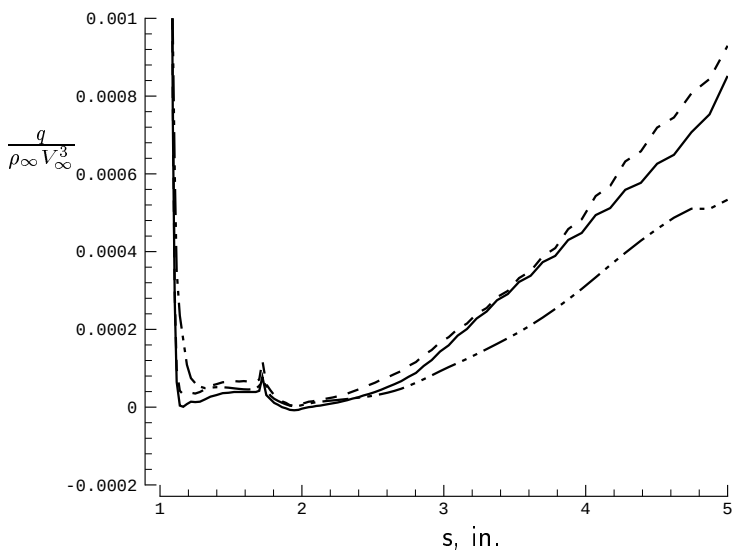
Figure 6.33: Loss of shock capture due to coarsening, fluctuation splitting.

fluctuation splitting with fluctuation minimization. A similar problem also occurred using DMFDSFV with curvature clustering. The removal of more points causes rapid solution deterioration due to the failure of both schemes to properly capture the bow shock on the coarsened meshes.

For evaluating the edge swapping adaption the starting mesh has half as many points as the baseline, about 16,000 nodes and 50,000 edges, with the goal of realizing an improvement in the solution. Curvature clustering was used to swap the 538 edges (approximately 1 percent) with the largest error estimate. These edges were all located in vicinity of the bow shock. The resulting heat transfer rates show only minor changes from the starting solution, Figure 6.34, without showing a clear progression toward the benchmark dataset. Edge swapping with fluctuation minimization is not beneficial for this case. About 1 percent of the edges were swapped but the resulting fluctuation splitting solution does not converge due to ringing of the bow shock near the stagnation point. Figure 6.35 shows pressure contours in the stagnation point region. Notice the irregular contours downstream of the shock. The difficulty in this



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.34: Two-dimensional capsule surface heating after swapping with curvature clustering, solid=start, dashed=1% swapped, dash-dot-dot=benchmark.

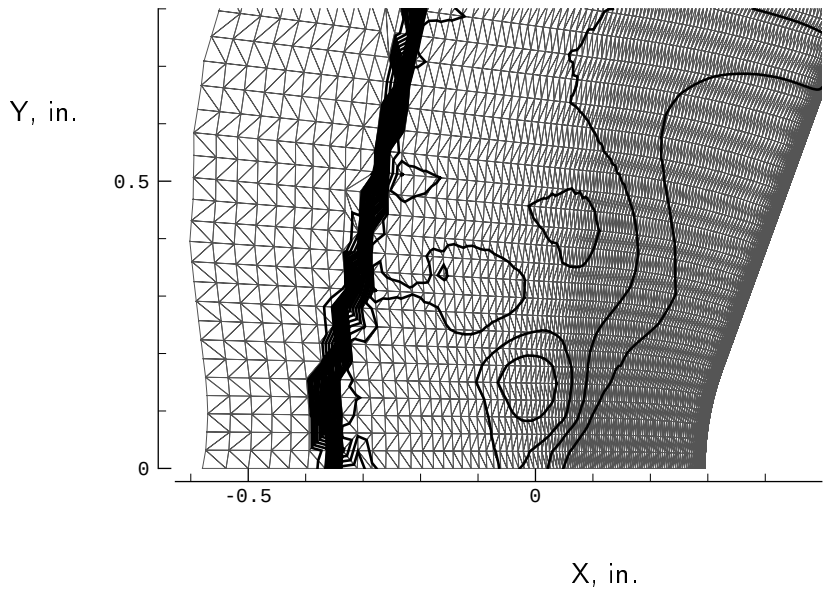


Figure 6.35: Fluctuation splitting shock bulging at stagnation point after edge swaps, pressure contours.

solution is that the bow shock is making discrete jumps across grid lines, induced by swapped edges at the shock. The unadapted bow shock more closely follows the grid lines at the $y = 0$ symmetry plane. Recall that this mesh was originally well-aligned with the bow shock from the structured-grid solvers, and the effect of edge swapping on a more random unstructured mesh may be different. Also, on a finer mesh, such as is used for the baseline solutions, the smaller grid spacing at the shock might lessen the detrimental effects of discrete jumps in the shock location. Edge swapping was tried again for fluctuation minimization, this time only swapping half a percent of the edges, but still produces the same disappointing results.

Nodal displacements are tested using the same 16,000-node mesh as was used to start the edge swapping, also with the goal of improving the solution toward the baseline. Curvature clustering was used to move 1717 nodes with the largest error estimates, about 10 percent of the nodes. The total distance moved was 10.34 in., the average distance moved was 0.006 in., and the RMS distance moved was 0.0095 in. Nodes were moved in the shock, at the shoulder, and in the forebody boundary layer.

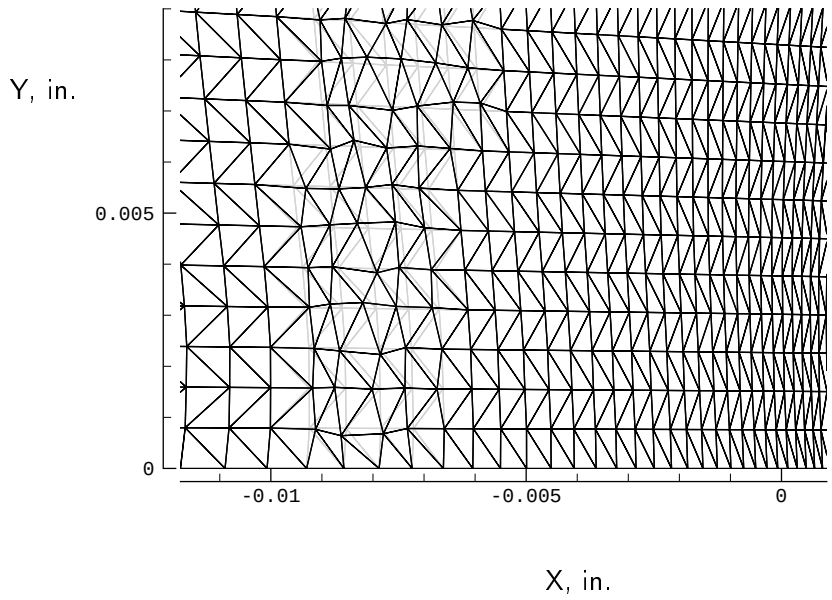
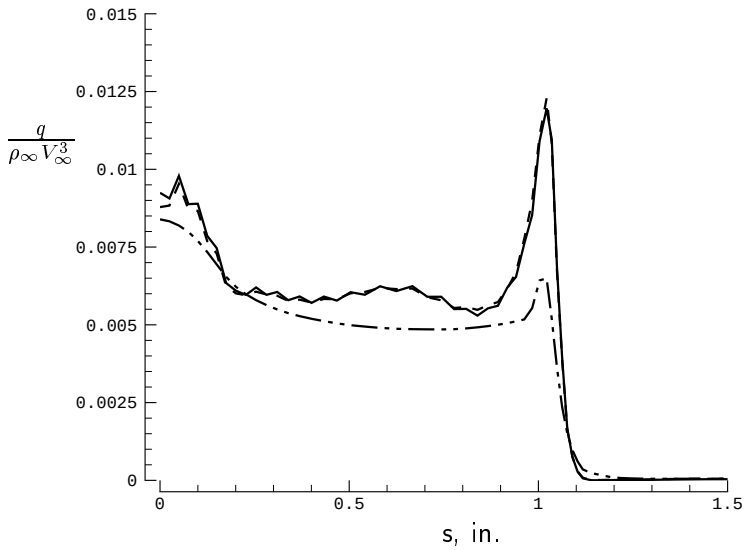


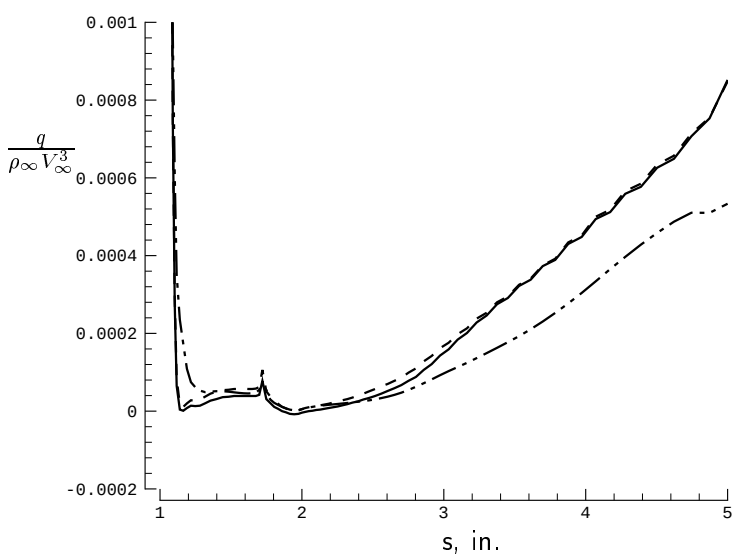
Figure 6.36: Nodal displacements at shock using curvature clustering, original mesh shaded.

A close-up of movement at the bow shock is shown in Figure 6.36, where the original mesh is shaded. Observe that the movement is not aggressive, but rather produces a gentle clustering toward the shock. Surface heating rates for this case are shown in Figure 6.37, where a minimal change in the solution is seen. Displacing nodes with fluctuation minimization moved 1827 nodes a total distance of 0.7488 in., for an average of 0.0004 in. and RMS of 0.0015 in. The fluctuation minimization movement, while moving a few more nodes than the curvature clustering, moved the nodes much smaller distances. Again, the movement occurred at the shock, near the shoulder, and in the forebody boundary layer. Surface heating rates for this case are shown in Figure 6.38. The solution has been made much worse on the forebody, while not much change is seen in the wake. Investigating the solution reveals that a windside vortex pattern has emerged, Figure 6.39, that is causing the lower heat transfer rates on the heatshield. The solution was run a further 800,000 iterations with no change to this windside vortex pattern.

Point insertion is tested on the unadapted 16,125-node mesh, seeking to add

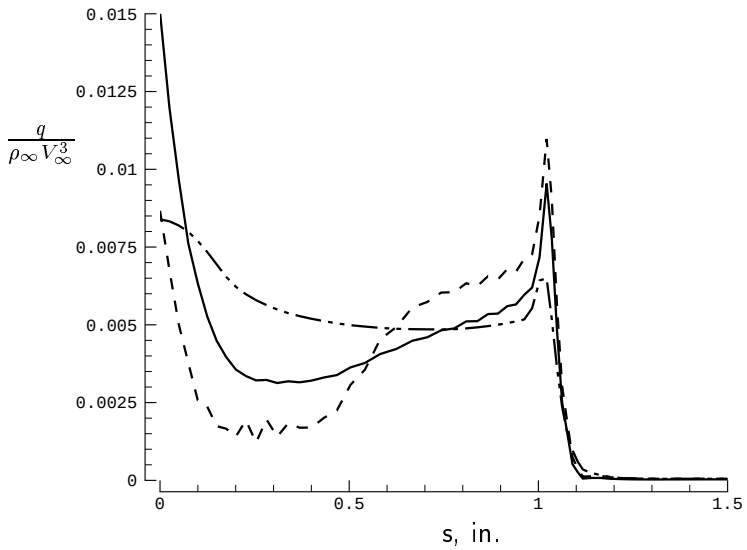


(a) Heatshield.

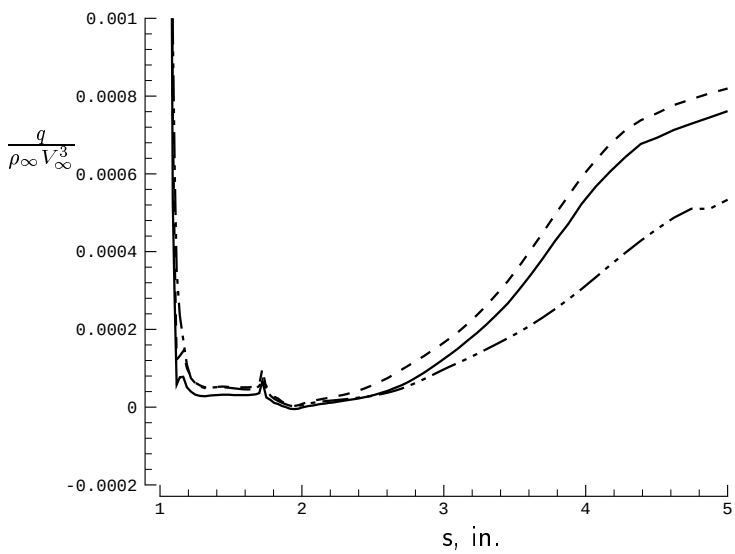


(b) Aftbody and sting.

Figure 6.37: Two-dimensional capsule surface heating after moving with curvature clustering, solid=start, dashed=10% moved, dash-dot-dot=benchmark.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.38: Two-dimensional capsule surface heating after moving with fluctuation minimization, solid=start, dashed=10% moved, dash-dot-dot=benchmark.

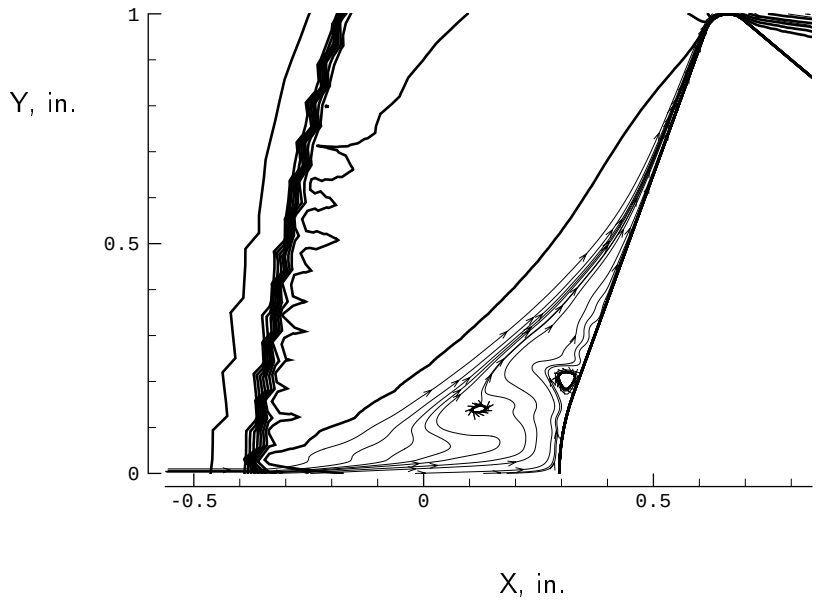
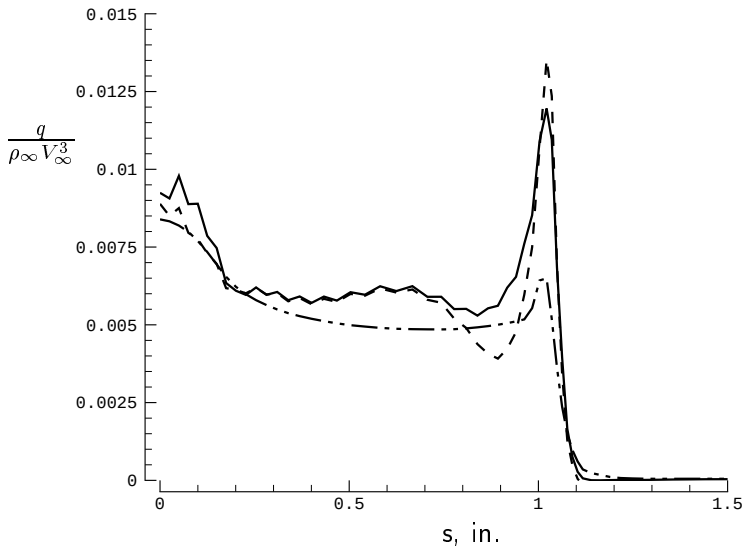


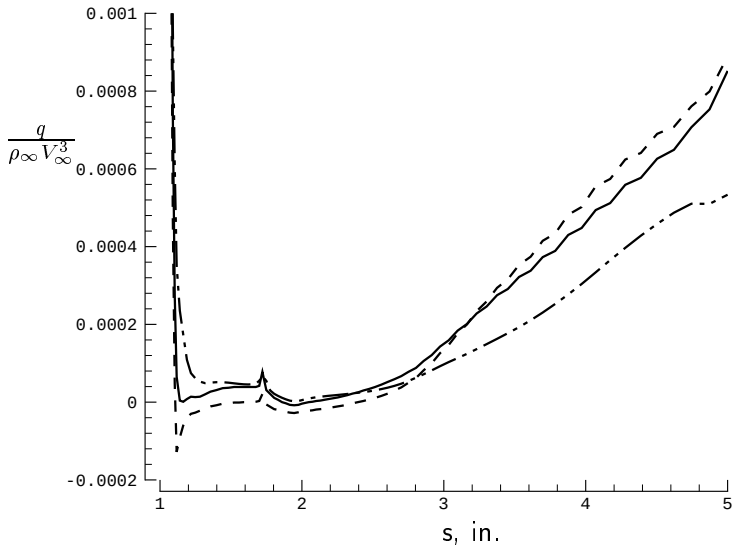
Figure 6.39: Windside vortices produced by fluctuation minimization nodal displacements.

20 percent more nodes. Curvature clustering added 3154 nodes, producing the heating results shown in Figure 6.40. On the forebody, there is improvement in the stagnation point region, but worsening near the shoulder. On the aftbody and sting the heating moves somewhat away from the benchmark result. Fluctuation minimization added 3299 nodes, with heating results shown in Figure 6.41. On the heatshield the heating levels rise markedly toward the benchmark result, but exhibit a high-frequency oscillation of significant amplitude. There is not much change in the heating in the wake region.

For the full adaption the starting solution is taken on the triangulated 125×129 mesh with 16,125 nodes, half the number of nodes used to generate the baseline unadapted solutions. The intention is to look for an improvement in the coarse mesh solution without increasing the number of nodes. Based upon the results of the component adaption tests, the strategy for an adaption cycle is to delete 10 percent of the nodes, swap 1 percent of the edges, move 5 percent of the nodes, and then insert back in 10 percent of the nodes. The solution is re-converged between each

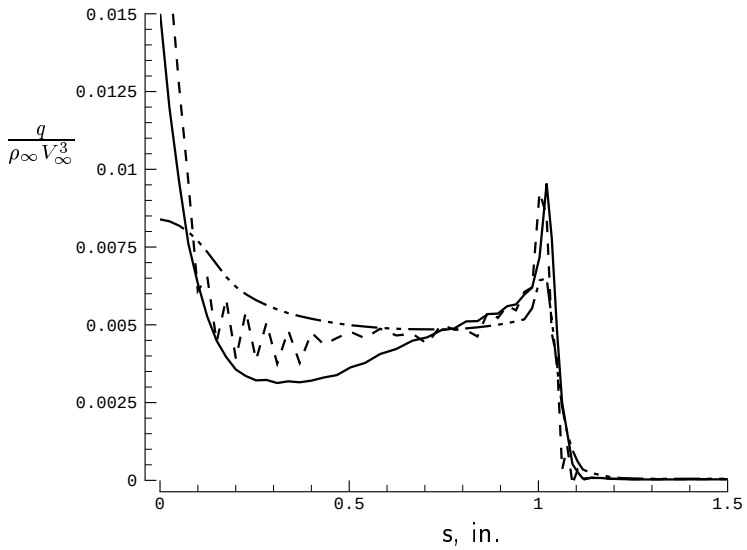


(a) Heatshield.

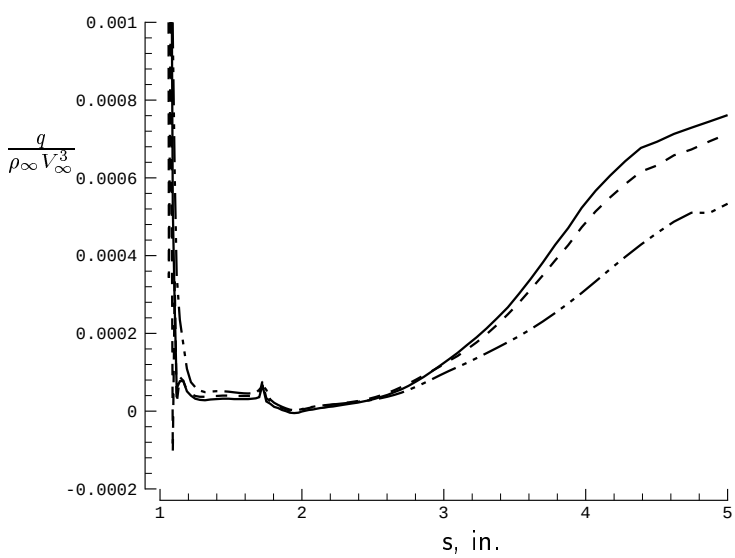


(b) Aftbody and sting.

Figure 6.40: Two-dimensional capsule surface heating after point insertion with curvature clustering, solid=start, dashed=20% inserted, dash-dot-dot=benchmark.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.41: Two-dimensional capsule surface heating after point insertion with fluctuation minimization, solid=start, dashed=20% inserted, dash-dot-dot=benchmark.

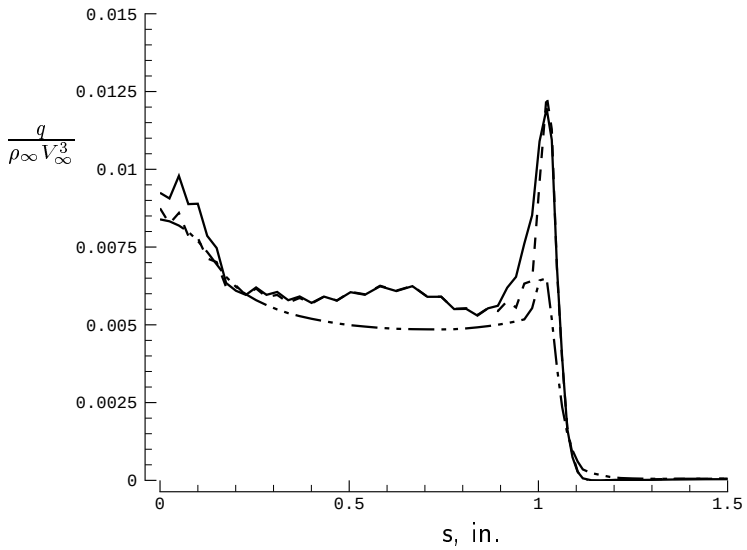
step of the adaption cycle.

The curvature clustering cycle proceeded by deleting 1637 nodes, swapping 444 edges, moving 696 nodes a total distance of 6.23 in., and adding 1599 nodes. The resulting heat transfer plots are shown in Figure 6.42. On the heatshield the stagnation region heating has been improved and is now in excellent agreement with the benchmark. There is no further change in heating over much of the forebody between $s = 0.2$–0.9, but the heating spike at the shoulder is more tightly resolved. In the wake region there is little change in heating on the aftbody, while the sting heating is increased, trending away from the benchmark.

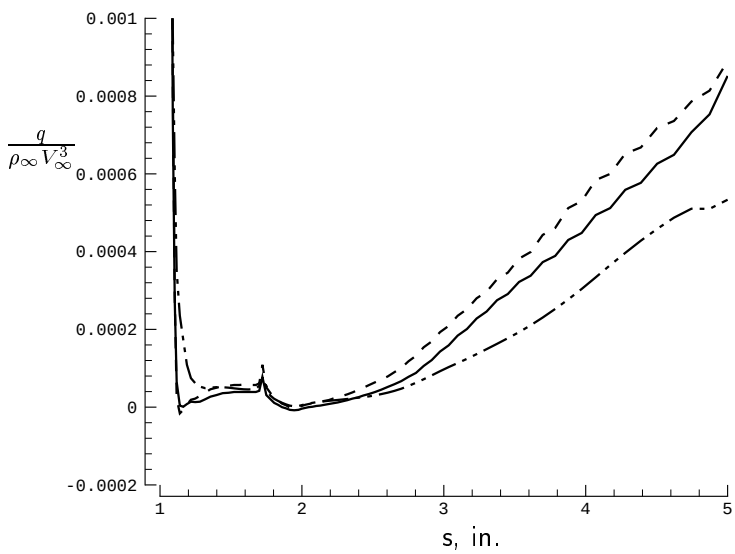
The fluctuation minimization adaption successfully removed 1601 nodes, but ran into trouble again while swapping. Windside vortices were spawned in the stagnation region by an oscillating bow shock. In an effort to damp the solution the CFL number was reduced by an order of magnitude, without producing an improvement in the solution or eliminating the vortices. The number of edges to be swapped was reduced to $\frac{1}{2}$ percent, but the same vortices and oscillating bow shock appeared. For this case edge swapping was omitted and the adaption cycle continued with the nodal displacement step, where 706 nodes were moved a total distance of 0.7 in. Finally, 1638 nodes were added, yielding the results of Figure 6.43. The forebody heating is generally improved toward the benchmark solution, although there is a high-frequency oscillation in the data starting at $s = 0.6$. Heating in the wake is only slightly changed, though the aftbody heating is improved to match the benchmark between $s = 1.2$–1.8.

Axisymmetric

The axisymmetric adaption component testing starts from the the baseline solution and mesh, which contains 11,125 nodes. The intention is to remove nodes without altering the surface heat transfer rates. About 10 percent of the nodes were successfully deleted without impacting the solutions using both curvature clustering, 1029 nodes, and fluctuation minimization, 1079 nodes. A close-up of the fluctuation splitting mesh is shown in Figure 6.44, where it can be seen that most of the nodes are

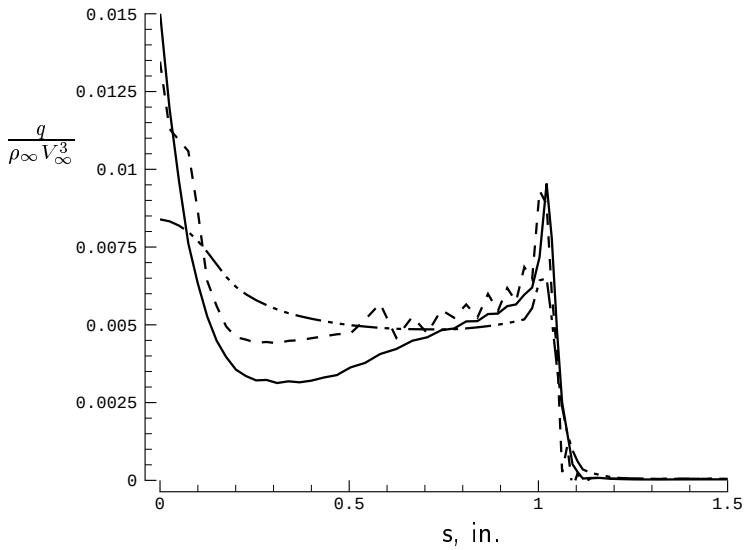


(a) Heatshield.

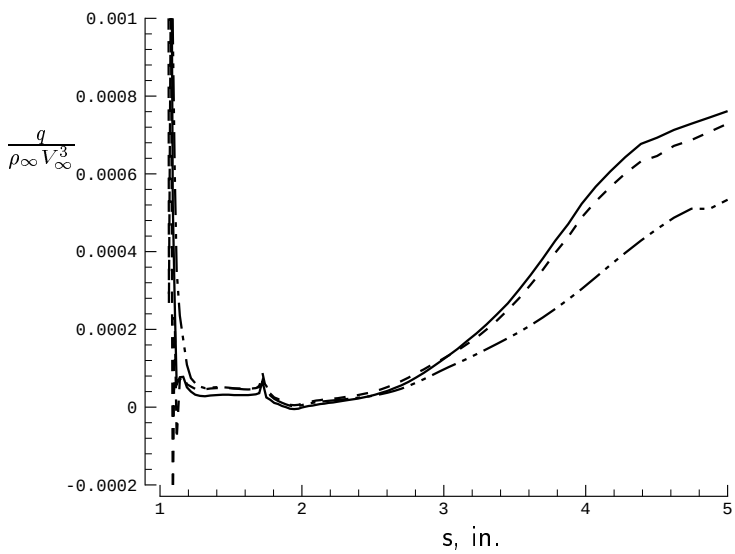


(b) Aftbody and sting.

Figure 6.42: Two-dimensional capsule surface heating after full adaption cycle with curvature clustering, solid=start, dashed=adapted, dash-dot-dot=benchmark.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.43: Two-dimensional capsule surface heating after full adaption cycle with fluctuation minimization, solid=start, dashed=adapted, dash-dot-dot=benchmark.

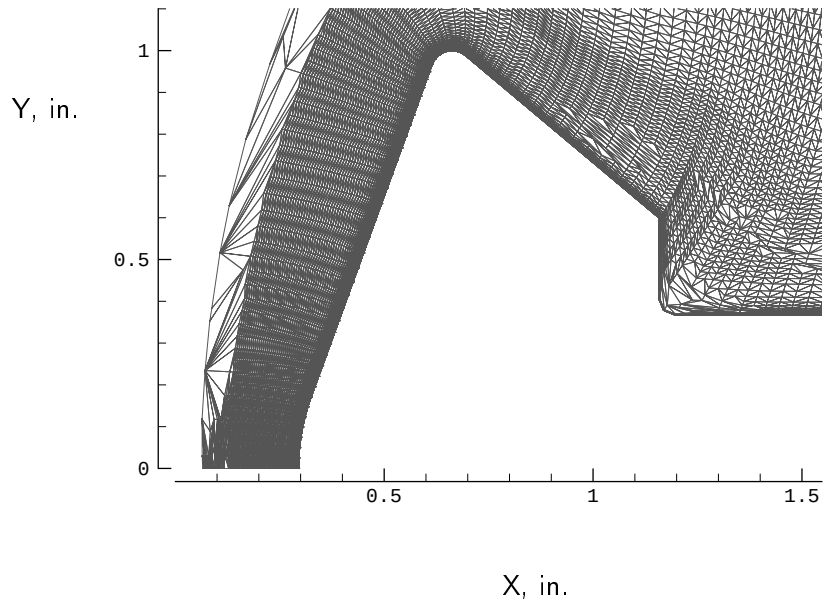
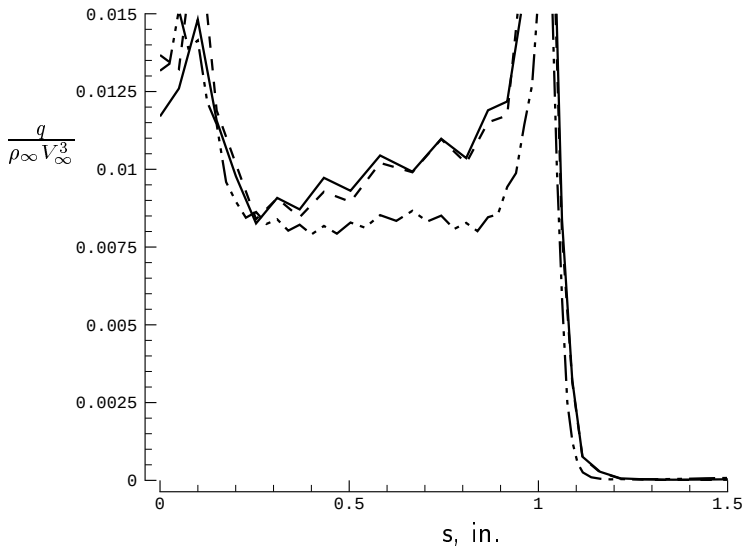


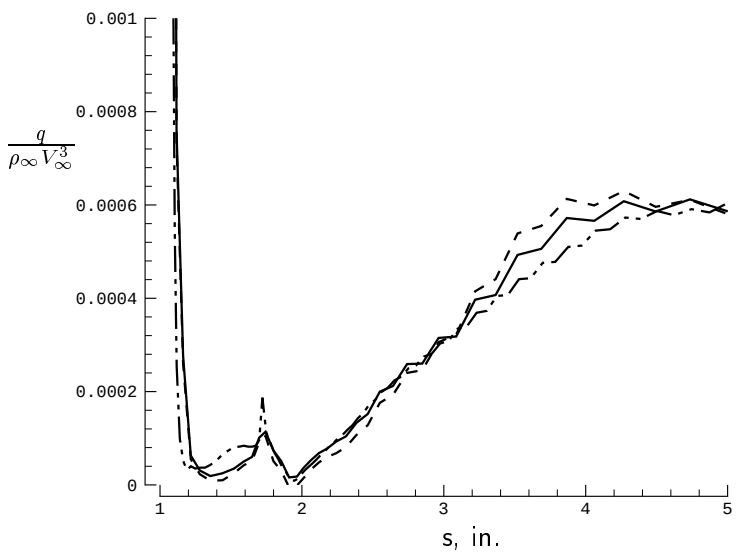
Figure 6.44: Coarsened axisymmetric mesh using fluctuation minimization.

removed from the freestream while a smaller amount are removed from the wake region. Further point reductions were tried, and the curvature clustering successfully removed another 1136 nodes with minimal change in the surface heating. However, removing 1081 more nodes with fluctuation minimization leads to the problem of a loss of shock capture, as was seen in the two-dimensional results of Figure 6.33. In this case, the solution deteriorates to an unacceptable level because of the bow shock blowout.

Edge swapping is started from a mesh four times coarser than the baseline, triangulated from $63 \times 45 = 2835$ nodes, looking for the heating predictions to improve toward the baseline results. The mesh has 8290 edges, and 78, about 1 percent, with the largest error estimates were swapped using curvature clustering. Figure 6.45 shows the initial and adapted heating rates compared with the baseline solution. The edge swapping produces only marginal changes in the heating rates. Fluctuation minimization swapped 88 edges and also shows minor changes in the heating, Figure 6.46, except at the stagnation point where the adapted heating is dramatically

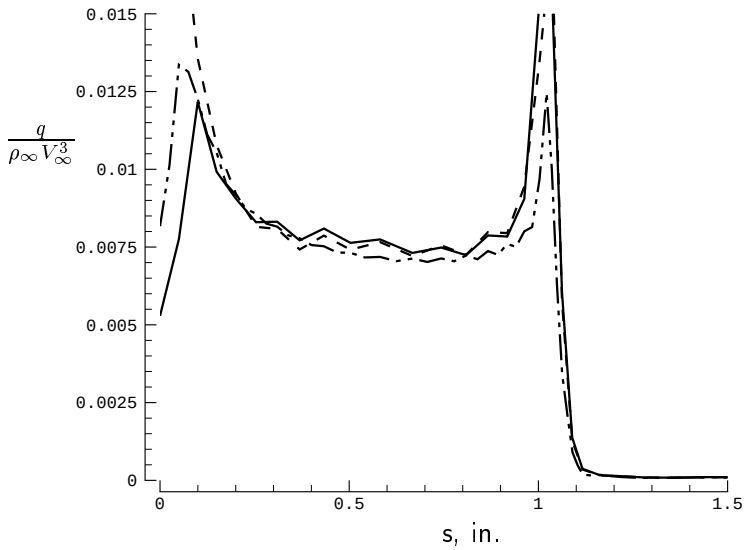


(a) Heatshield.

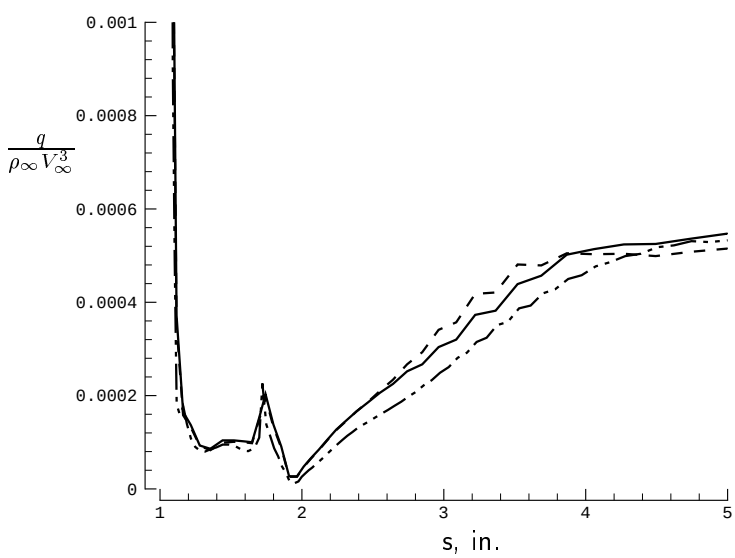


(b) Aftbody and sting.

Figure 6.45: Axisymmetric capsule surface heating after edge swapping with curvature clustering, solid=start, dashed=adapted, dash-dot-dot=DMFDSFV baseline.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.46: Axisymmetric capsule surface heating after edge swapping with fluctuation minimization, solid=start, dashed=adapted, dash-dot-dot=fluctuation splitting baseline.

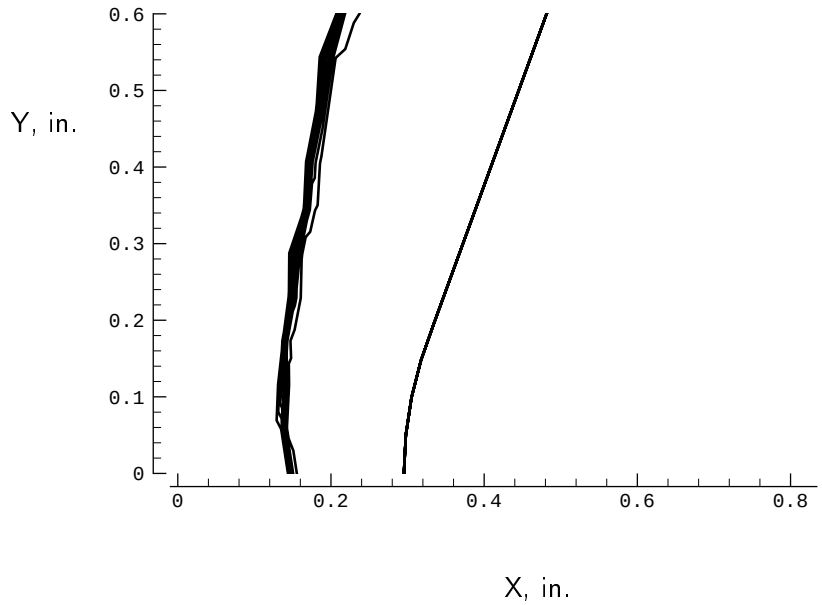
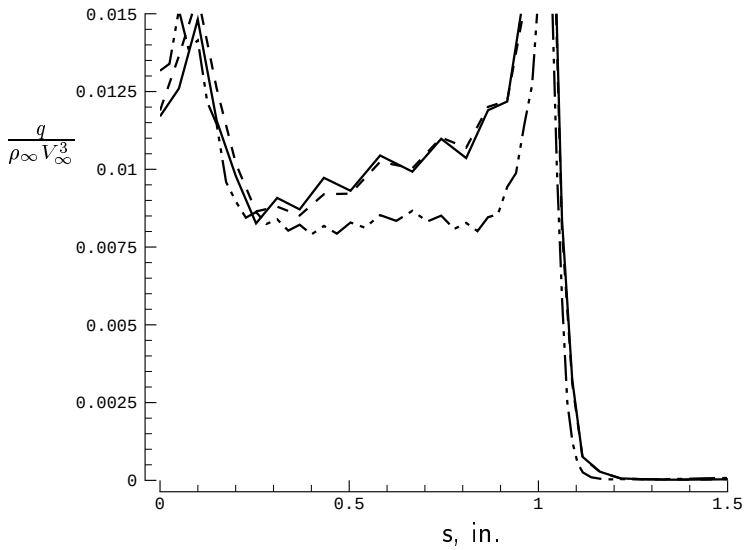


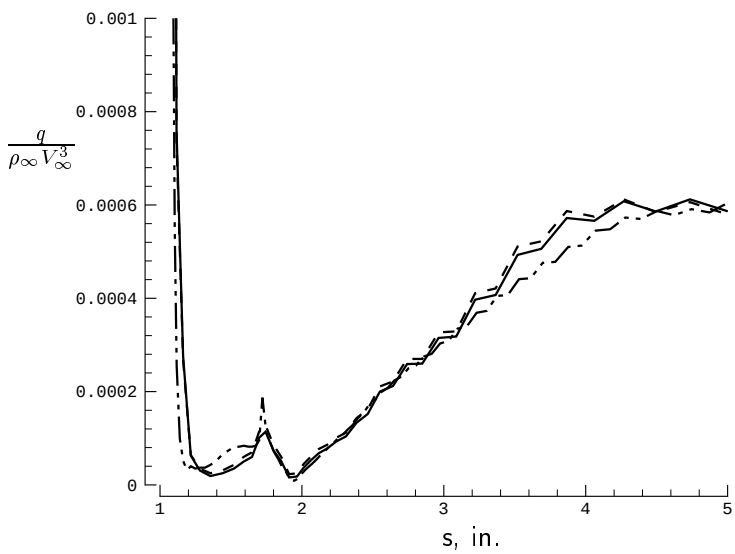
Figure 6.47: Shock kinked in toward stagnation point at the x-axis after axisymmetric edge swapping with fluctuation minimization, velocity contours.

raised well beyond the correct result. Looking at the bow shock in the vicinity of the nose, Figure 6.47, a sharp kink in toward the body is seen along the axisymmetric axis. This kink in the bow shock elevates the pressure along the axis and causes the significant stagnation point heating over-prediction.

The nodal displacement tests start from the same 2835-node meshes and solutions that were used to start the edge swapping check. Five percent of the nodes were moved using each scheme without producing a noticeable change in the solutions. A second pass of node movement was made, this time moving 10 percent of the nodes. Curvature clustering moved the nodes a total distance of 2.88 in. with an RMS of 0.017 in. The movement was performed at the bow shock and in the forebody boundary layer. The results, shown in Figure 6.48, show minimal change with a slight smoothing of the prediction on the heatshield between $s = 0.2$–0.5. With fluctuation minimization the nodes were moved a total distance of 0.18 in. with an RMS of 0.0015 in. Most of this movement was in the shock near the axis. Figure 6.49 shows basically no change to the heating levels due to the nodal displacements.

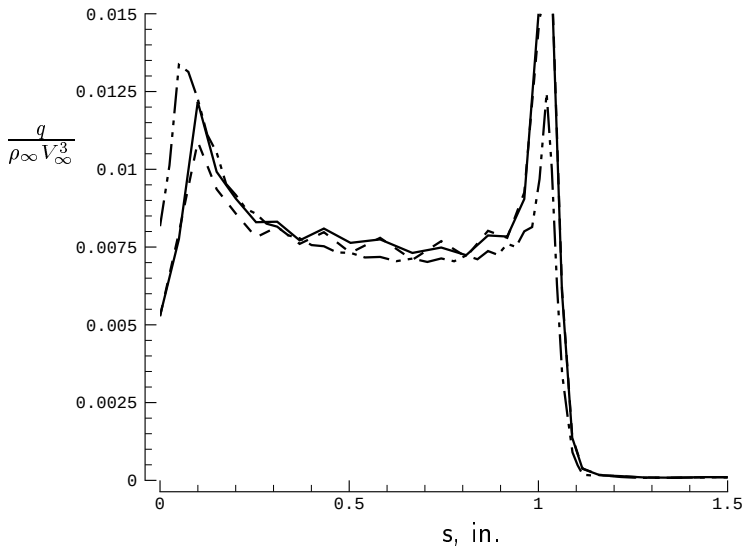


(a) Heatshield.

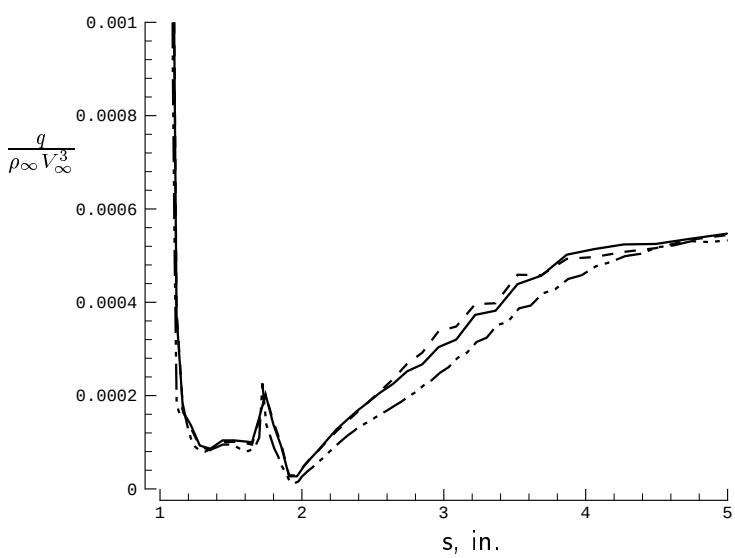


(b) Aftbody and sting.

Figure 6.48: Axisymmetric capsule surface heating after nodal displacements with curvature clustering, solid=start, dashed=10% moved, dash-dot-dot=DMFDSFV baseline.



(a) Heatshield.



(b) Aftbody and sting.

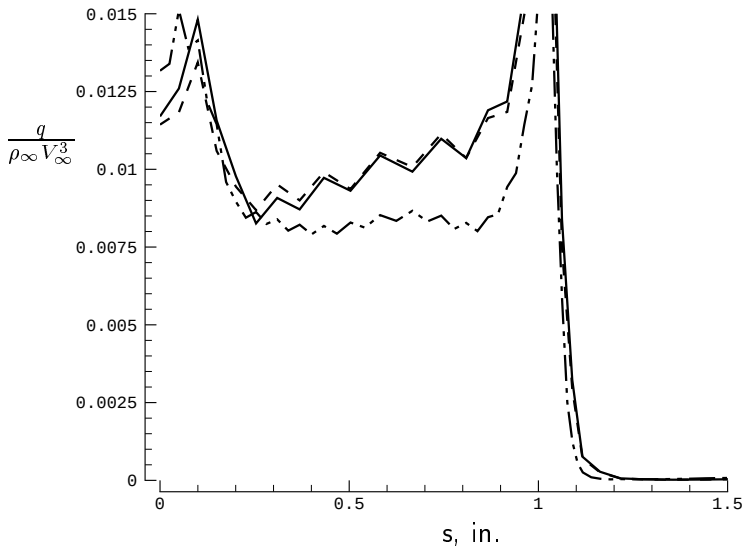
Figure 6.49: Axisymmetric capsule surface heating after nodal displacements with fluctuation minimization, solid=start, dashed=10% moved, dash-dot-dot=fluctuation splitting baseline.

Point insertion is applied to the coarse 2835-node mesh using both schemes to add 10 percent more nodes. Curvature clustering added 283 nodes at the shock and in the windside boundary layer, but does not change the surface heating appreciably, Figure 6.50. Fluctuation minimization added 287 nodes, also in the shock and windside boundary layer. While the average heating levels on the forebody remain roughly the same, Figure 6.51 shows a wild oscillation in the surface heating has been introduced with the additional nodes. The sting heating shows a small reduction, but does not exhibit a clear improvement.

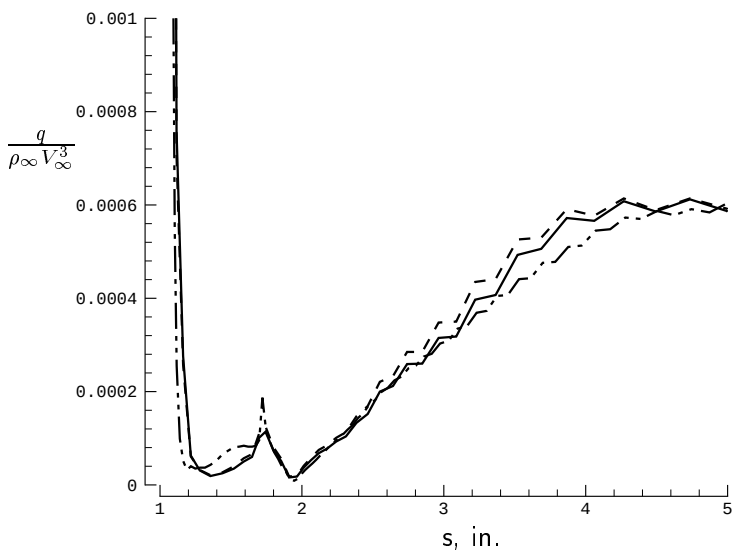
For the full axisymmetric adaption procedure a cycle is defined similar to the cycle for two dimensions. Starting from a converged solution on a mesh with half the number of nodes as the baseline, 125×45 versus 125×89, 10 percent of the nodes are deleted, 1 percent of the edges are swapped, 5 percent of the nodes are moved, and then 10 percent more nodes are added back. The solution is re-converged between each step of adaption.

Curvature clustering deleted 560 of the 5625 nodes, primarily from the freestream but also from the wake and in the sting/vehicle juncture. Of the 14,941 remaining edges, the 150 with the largest error estimates were swapped. These edges were located in the bow shock and in the forebody boundary layer. Then the 253 nodes with the largest error estimates were moved. These nodes were also in the shock and forebody boundary layer. The nodes were moved a total distance of 1.2 in. with an RMS of 0.007 in. Finally, 570 nodes were added, enriching both the shock and the boundary layer. The results of the adaption cycle are shown in Figure 6.52. Essentially no change in surface heating is seen due to the adaption, other than a smoothing of the originally oscillatory data on the sting.

Fluctuation minimization ran into problems with deleting nodes from the starting mesh for the full adaption test. Deleting 10 percent of the nodes proved to be too aggressive, leading to a loss of the bow shock capture. So the adaption cycle was modified to only delete and insert 5 percent of the nodes, rather than the original target of 10 percent. Deletion was successful removing 263 points entirely from the freestream. Edge swapping was viable for this case as 159 edges were swapped. The problems encountered during the two-dimensional edge swapping were not encountered with the

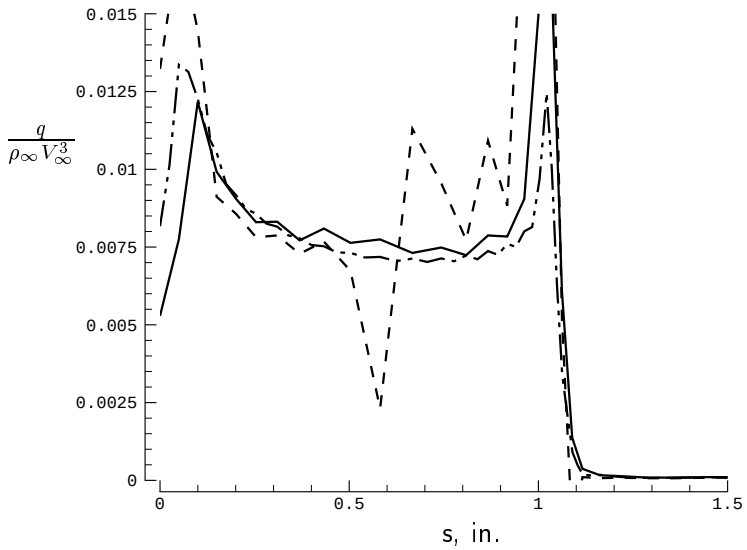


(a) Heatshield.

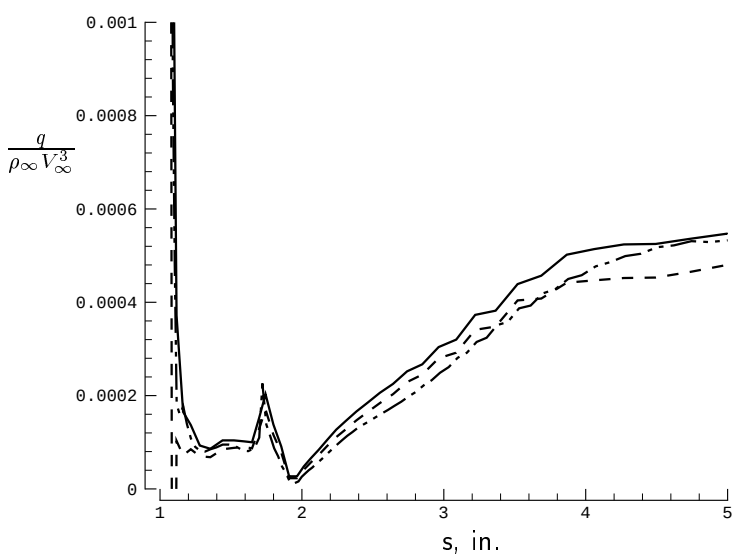


(b) Aftbody and sting.

Figure 6.50: Axisymmetric capsule surface heating after point insertion with curvature clustering, solid=start, dashed=10% inserted, dash-dot-dot=DMFDSFV baseline.

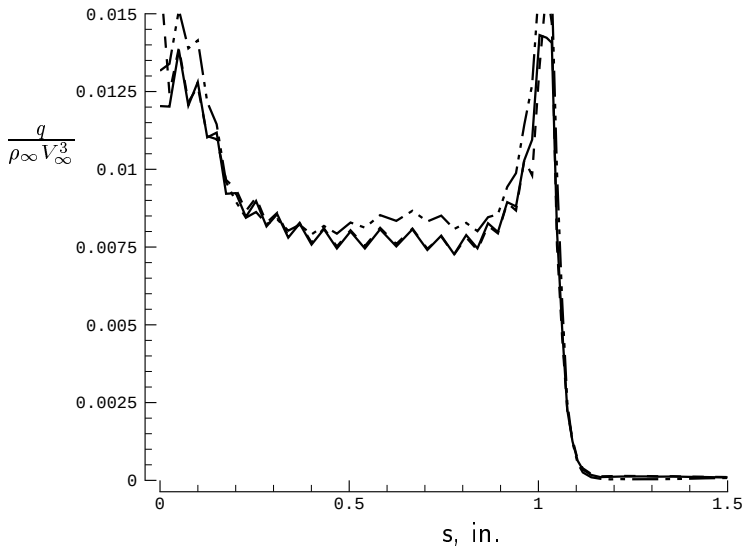


(a) Heatshield.

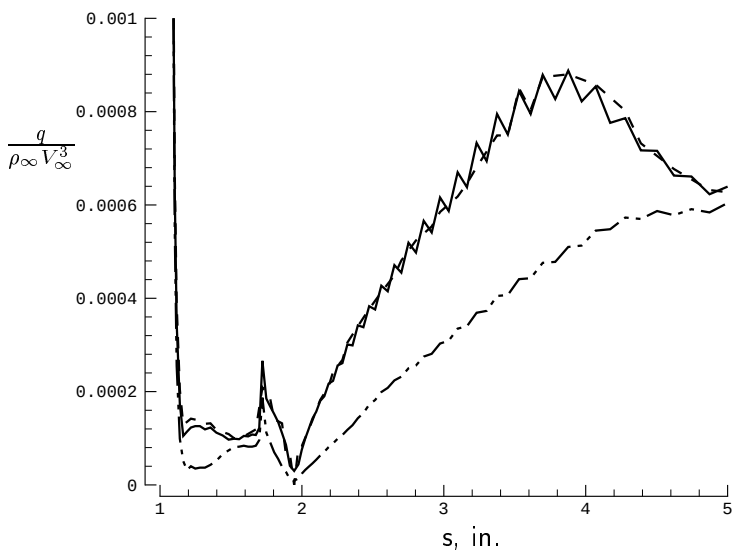


(b) Aftbody and sting.

Figure 6.51: Axisymmetric capsule surface heating after point insertion with fluctuation minimization, solid=start, dashed=10% inserted, dash-dot-dot=fluctuation splitting baseline.



(a) Heatshield.



(b) Aftbody and sting.

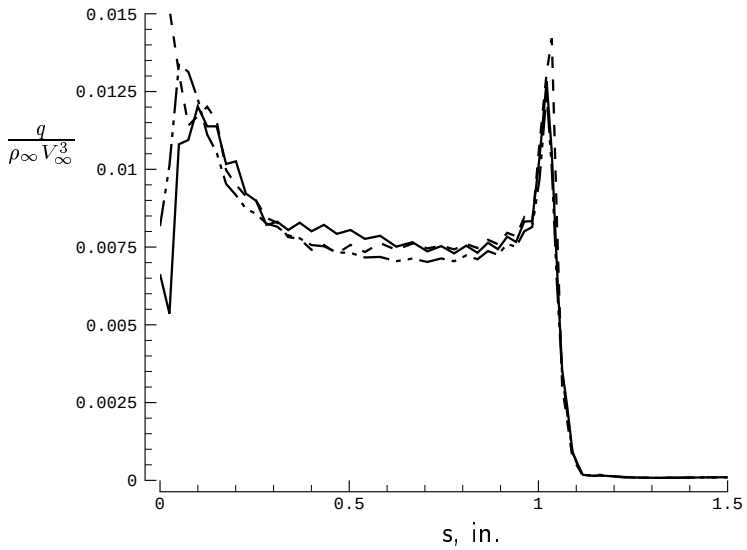
Figure 6.52: Axisymmetric capsule surface heating after full adaption cycle with curvature clustering, solid=start, dashed=adapted, dash-dot-dot=DMFDSFV baseline.

axisymmetric case. The nodal displacements were performed primarily at the shock with a small amount moved in the boundary layer, for a total of 268 nodes moved a distance of 0.22 in., RMS of 0.002 in. Point insertion added 280 nodes, mainly at the shock but also a small amount in the forebody boundary layer. The results of the adaption cycle are shown in Figure 6.53. Most of the heatshield remains unchanged except for an improvement between $s = 0.3$–0.7 and a change in the stagnation point trend from being a large under-prediction to being a large over-prediction. Heating in the wake is largely unchanged other than an increase at the tail end of the sting.

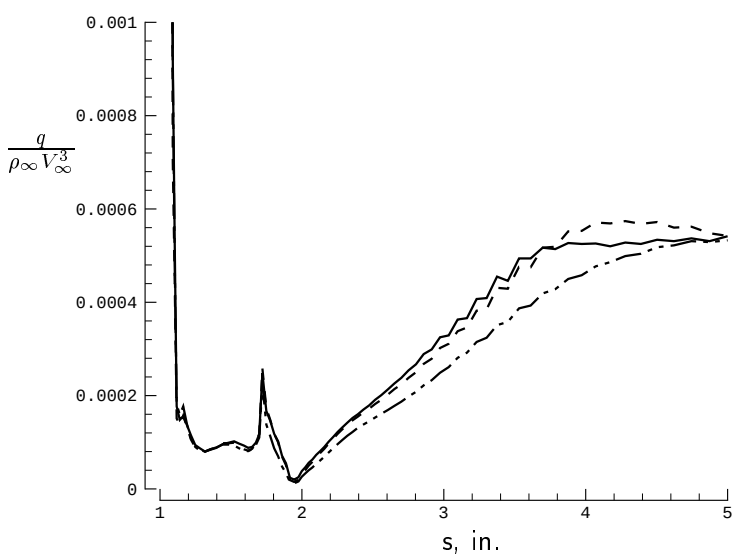
6.5 Discussion

The primary new development of this chapter is the extension of the characteristic alignment mesh adaption strategy to the Navier-Stokes system of equations by a generalization of the fluctuation minimization formulas originally developed for the scalar advection-diffusion equation. Particular emphasis is placed on formulating the nodal displacement forcing functions for the upwind, positive, linearity preserving fluctuation splitting distribution scheme consistently in both axisymmetric and two-dimensional coordinates.

The main test is a hypersonic, perfect gas wind tunnel case for a Martian probe model, configured for both the real axisymmetic geometry and the corresponding two-dimensional geometry. In considering the unadapted results, the two-dimensional baseline solutions for both fluctuation splitting and DMFDSFV are obtained on the 125×257 mesh, the same mesh for which the LAURA benchmark is grid converged, aside from at the stagnation point. The fluctuation splitting surface pressures are in excellent agreement with the benchmark while the DMFDSFV surface pressures match the benchmark on the heatshield but do not match well in the wake. For the forebody heating fluctuation splitting shows excellent agreement with the benchmark aside from at the stagnation point while the DMFDSFV result is consistently high. The aft heating is about the same for both schemes, reasonably agreeing with the benchmark.



(a) Heatshield.



(b) Aftbody and sting.

Figure 6.53: Axisymmetric capsule surface heating after full adaption cycle with fluctuation minimization, solid=start, dashed=adapted, dash-dot-dot=fluctuation splitting baseline.

For the axisymmetric unadapted results, the baseline solutions for both unstructured schemes are obtained on the 125×89 mesh, which is a factor of four coarser than the mesh that Hollis had to use to get good results with the structured finite volume code NEQ2D. DMFDSFV surface pressures are excellent on the forebody, very good on the aftbody, but a little high on the sting between $s = 2.5$–3.5. The fluctuation splitting pressures are a little low on the forebody and excellent in the wake, splitting the difference between the two benchmark datasets. The fluctuation splitting forebody heating is an excellent match with the benchmark, aside from at the stagnation point. DMFDSFV over-predicts the forebody heating. Both schemes produce an excellent match with the benchmark in the wake, splitting the difference between the two computational benchmarks on the sting.

In testing the component adaption techniques point deletion is the most beneficial tool. Coarsening works well removing up to about 20 percent of the nodes, but when taken too far leads to a loss of the bow shock capture, and a rapidly deteriorating solution. The other adaption techniques generally have minimal or negative impacts on the solution, with the exception of point insertion with fluctuation minimization where a little solution improvement occurs. Curvature clustering produces nodal displacements on the order of ten times larger than fluctuation minimization for the same number of points moved, but the fluctuation splitting solutions on the contorted meshes are less robust. The lack of robustness is usually caused by distortions to the bow shock near the symmetry axis. Common failure modes are an oscillating shock shedding windside vortices or a steady shock kink producing unexpected pressure contours and surface heating spikes. The root cause is that the local, compact adaption techniques employed here are mis-aligning the mesh with the bow shock. The typical behavior is that the fluctuation splitting scheme is more sensitive to grid perturbations introduced by the fluctuation minimization adaption in corrupting the solution, but is generally more accurate than DMFDSFV on suitable meshes. The component tests indicate that the mesh adaption for systems can not be nearly as aggressive as was possible for scalar equations.

For the full adaption cycle in two dimensions with curvature clustering the heating in the stagnation region and at the shoulder improves, the sting heating is worse, and

the rest of the vehicle sees no change. With fluctuation minimization the forebody heating improves but becomes oscillatory, the aftbody is the same, and the sting improves. For the axisymmetric case the curvature clustering cycle does not change the magnitude of the heating but does smooth the initially rough heating on the sting. Fluctuation minimization produces some improvement over a quarter of the heatshield but also makes the sting heating a little worse at the tail end, leaving the rest of the heating unchanged. Although heat transfer is a variable of primary interest for aerothermodynamics and may be a first consideration for driving the adaption criteria, the present results emphasize that for blunt-body hypersonic aerothermodynamics the bow shock placement is critical to the forebody heat transfer rates. Due to the strength of the hypersonic bow shock, discrete jumps in shock position from one node to another produce significant pressure disturbance waves which strongly affect the flowfield in the subsonic bubble and can overwhelm more subtle adaption within the boundary layer.

To conclude, the adaption techniques do not significantly improve the solutions, the adaption effectiveness is not consistent, and the solutions display a lack of robustness. Point deletion works the best, but is risky if applied too aggressively. These results indicate that the extension of characteristic alignment based adaption from scalar problems to the Navier-Stokes system is not straightforward due to the non-uniqueness of the characteristics and the complexity of the flowfield features which are not adequately modeled by the minimization of a single functional based upon fluctuations as formulated here.

These local adaption strategies are designed to be synergistic with the compact solvers, and are attractive candidates for parallelization, but the present results may be indicating that the adaption requires a global view to introduce a level of smoothness. Dr. Peter Gnoffo[123] has suggested reworking the adaption strategies to increase the support of the discretization stencil. Such an increased stencil may allow for the introduction of smoothing parameters or other grid-quality controls, but at the expense of requiring more than nearest-neighbor communications, complicating any extension to a parallel computing strategy. The upper limit on an increased stencil would be a fully implicit treatment.

As a post-note to this chapter, Yamaleev and Carpenter[124] have recently presented a very detailed mathematical analysis of the effect of grid adaption for supersonic blunt bodies in the context of second and fourth-order structured finite volume schemes. They specifically consider nodal displacements and point insertion, concluding, "The grid refinement study shows that for the second-order scheme, neither grid adaptation strategy improves the numerical solution accuracy..." For the fourth-order scheme the situation is worse because, "...the design-order error component drastically increases because of the grid nonuniformity." A consensus on adaption guidelines for general supersonic flows with captured multi-dimensional discontinuities remains elusive, for both structured and unstructured meshes.

Chapter 7

Summary and Recommendations

This report addresses the suitability of fluctuation splitting for hypersonic aerothermodynamics. A desirable aerothermodynamic solver must be robust in the presence of strong shocks, meaning that the solution must remain stable and positive without radiating dispersion or phase errors. The desirable scheme must also be of high accuracy to resolve the derivative quantities heat transfer and shear stress, and thus should be non-diffusive. The classic approach to reconciling these two criteria is the construction of a non-linear second-order scheme with reduction to first-order in the vicinity of captured discontinuities. Perhaps the most popular of these schemes for unstructured meshes is finite volume with Roe upwind flux difference splitting and limited linear reconstruction (referred to as DMFDSFV here). A drawback to this scheme is the reliance upon a locally one-dimensional approximate Riemann solver even when applied in two and three dimensions. Fluctuation splitting possesses the benefits of the traditional scheme and adds a truly multi-dimensional upwind distribution. But the discussion of a flow solver independent of the computational domain is only of academic interest, so this report also seeks to develop a mesh adaption scheme in conjunction with the desirable solver to improve the results and reduce the expense of analyzing a vehicle geometry.

The fluctuation splitting scheme considered here had been introduced by Sidilkover as the optimal compact zero-cross-diffusion solver for linear hyperbolic equations, with an extension to the two-dimensional Euler equations on Cartesian grids. The

formulation lacked eigenvalue limiting, an axisymmetric derivation, and a viscous coupling, had not been demonstrated on a general unstructured mesh, was untested for non-linear advection-diffusion, had not been applied to a hypersonic problem, had not been used to calculate heat transfer, and had never been evaluated relative to the current most popular strategy—DMFDSFV. The evaluation of fluctuation splitting begins by proving the equivalence of DMFDSFV and fluctuation splitting in one dimension along with a strategy for viscous coupling to the inviscid flux distribution. Then the extension of both schemes to scalar two-dimensional problems is detailed to underscore the fundamental differences between a multi-dimensional upwinding and the locally one-dimensional treatment, with particular emphasis on the production of artificial dissipation. The superiority of fluctuation splitting over DMFDSFV for both linear and non-linear advection-diffusion is demonstrated and a new mesh adaption strategy for scalar problems is developed to exploit characteristic alignment with fluctuation splitting. This adaption scheme maintains the fully local, compact stencil of fluctuation splitting to allow for future parallelization and use on computer clusters, and is able to produce accuracy and efficiency gains for the scalar problems.

The details of extending multi-dimensional upwind fluctuation splitting to a practical aerothermodynamic solver for the two-dimensional and axisymmetric Navier-Stokes equations are detailed and a suite of verification and validation cases compares the fluctuation splitting and DMFDSFV schemes. Fluctuation splitting is still to be preferred over DMFDSFV for the system of equations, more so for viscous problems than inviscid, but the difference is not as dramatic as is seen for the scalar model problems. While fluctuation splitting should certainly be considered as a viable alternative to DMFDSFV when selecting algorithms for code development, the differences in capability demonstrated here are not significant enough to obsolete existing codes based upon DMFDSFV technology.

The characteristic alignment mesh adaption strategy is extended to two-dimensional and axisymmetric systems as a generalized form of fluctuation minimization. The practical implementation for systems proves to be very difficult, and in the current adaption framework only the point deletion strategy is recommended. None of

the other adaption components produce significant accuracy improvements and often lead to a loss of robustness of the solver. For inviscid or subsonic flows where heating is not an issue, perhaps the current adaption could be of use. For aerothermodynamics though, the local adaption strategy as it is currently presented is not recommended.

Parallelization of the fluctuation splitting scheme appears to be attractive because of the compact stencil. The weak implementation of the boundary conditions developed here should be an enabling mechanism for exchanging boundary data between partitioned sub-domains in a parallelized scheme. Another significant operational improvement could be made by casting the fluctuation splitting in an implicit scheme, with either a point-implicit or colored strategy the most likely format. Two-equation field models for turbulence appear to be the best path for extending the current fluctuation splitting formulation to Reynolds and Farve-averaged flows. One remaining significant unknown for fluctuation splitting with regards to aerothermodynamic capability is the inclusion of thermal and chemical non-equilibrium. In particular, the transformation of the flux Jacobian to auxiliary variables may not yield an easily factorizable system. Appendix B expresses the linearizations and transformations in terms of a general gas, but all the details of such an extension need to be worked out before a fair assessment of implementing a non-equilibrium scheme can be made.

Modifications to the adaption strategy for increased robustness should relax the constraint that the adaption be as compact as the solver, focusing on some form of increased stencil as a means to introduce smoothing. Perhaps, though, a completely different adaption strategy is called for, something with a scaffolding analogy by which the bow shock, embedded shocks, vortices, and other inviscid phenomenon are framed with lines or shell elements while the gaps in the mesh framework are filled in with an unstructured triangulation. Whatever adaption strategy is pursued, the present results underscore the need for excellent bow shock alignment as a prerequisite to boundary layer refinement for the hypersonic blunt-body problem.

Appendix A

Limiters

A limiter is a function used to limit the ratio of two values, satisfying,

$$\psi(0) \ = 0, \quad \psi(1) = 1 \tag{A.1}$$

A symmetric limiter is defined by,

$$\psi\left(\frac{p}{q}\right) \ = \frac{p}{q}\,\psi\left(\frac{q}{p}\right) \tag{A.2}$$

Symmetric limiters can also be expressed in terms of symmetric averaging functions, M_ψ, obeying,

$$q\,\psi\left(\frac{p}{q}\right) \ = M_\psi(p,q) = M_\psi(q,p) = p\,\psi\left(\frac{q}{p}\right) \tag{A.3}$$

A limiter that can achieve a value greater than unity is termed a compressive limiter.

Degenerate limiters

Two degenerate, non-symmetric limiters, satisfying only one of the constraints in Eqn. A.1, are useful. The first order limiter, $\psi = M_\psi = 0$, is employed to limit a scheme to first order spatial accuracy. In contrast, an unlimited scheme results from the choice $\psi = 1$.

Minmod

The Minmod limiter[125] is a non-compressive, symmetric limiter defined as,

$$\psi\left(\frac{p}{q}\right) = \max(0,\ \min(1,\ p/q)) \tag{A.4}$$

or,

$$\psi\left(\frac{p}{q}\right) = \begin{cases} 0 & pq \leq 0 \\ p/q \quad \text{if} & |p| \leq |q| \\ 1 & |p| \geq |q| \end{cases} \tag{A.5}$$

The associated averaging function is,

$$M_\psi(p,q) = \begin{cases} 0 & pq \leq 0 \\ p \quad \text{if} & |p| \leq |q| \\ q & |p| \geq |q| \end{cases} \tag{A.6}$$

The Minmod limiter is the non-compressive limit of a generalized limiter of Sweby[126]. Minmod is achieved by setting the parameter $\varepsilon = 1$. The upper limit on ε is the Superbee limiter[127], $\varepsilon = 2$.

$$\psi\left(\frac{p}{q}\right) = \max[0,\ \min(\varepsilon p/q,\ 1),\ \min(p/q,\ \varepsilon)] \tag{A.7}$$

$$\psi\left(\frac{p}{q}\right) = \begin{cases} 0 & pq \leq 0 \\ \varepsilon p/q & \varepsilon\,|p| \leq |q| \\ 1 \quad \text{if} & |p| \leq |q| \leq \varepsilon\,|p| \\ p/q & |q| \leq |p| \leq \varepsilon\,|q| \\ \varepsilon & \varepsilon\,|q| \leq |p| \end{cases} \tag{A.8}$$

$$M(p,q) = \begin{cases} 0 & pq \leq 0 \\ \varepsilon p & \varepsilon\,|p| \leq |q| \\ q \quad \text{if} & |p| \leq |q| \leq \varepsilon\,|p| \\ p & |q| \leq |p| \leq \varepsilon\,|q| \\ \varepsilon q & \varepsilon\,|q| \leq |p| \end{cases} \tag{A.9}$$

Similar, non-symmetric limiters have been proposed by Chakravarthy[128],

$$\psi\left(\frac{p}{q}\right) = \max[0,\ \min(p/q,\ \varepsilon)] \tag{A.10}$$

and Barth[22],

$$\psi\left(\frac{p}{q}\right) = \max[0,\ \min(\varepsilon p/q,\ 1)] \tag{A.11}$$

The upper bound on these limiters is $1 \leq \varepsilon \leq 2$.

Van Leer

The harmonic Van Leer limiter[4] is a symmetric compressive limiter with an upper bound of 2.

$$\psi\left(\frac{p}{q}\right) = \frac{\frac{p}{q}+|\frac{p}{q}|}{1+|\frac{p}{q}|} = \frac{pq+|pq|}{q^2+|pq|} = \frac{|p|+|q|\frac{p}{q}}{|p|+|q|} \tag{A.12}$$

$$M_\psi = \frac{|p|q+p|q|}{|p|+|q|} \tag{A.13}$$

Van Albada

The Van Albada limiter[129] is a symmetric, differentiable compressive limiter with an upper bound of 1.18.

$$\psi\left(\frac{p}{q}\right) = \frac{\frac{p^2+\varepsilon^2}{q^2+\varepsilon^2}+\frac{p}{q}}{1+\frac{p^2+\varepsilon^2}{q^2+\varepsilon^2}} = \frac{p^2+\varepsilon^2+(q^2+\varepsilon^2)\frac{p}{q}}{p^2+\varepsilon^2+q^2+\varepsilon^2} \tag{A.14}$$

$$M_\psi = \frac{(pq+\varepsilon^2)(p+q)}{p^2+q^2+2\varepsilon^2} \tag{A.15}$$

For this limiter the small parameter, ε, serves to reduce the limiting in smooth regions. Assuming $\mathcal{O}(1)$ variations over a normalized distance, this parameter is scaled as $\varepsilon^2 \sim \ell^3$, where ℓ is the mesh spacing.

Van Albada presents the averaging function as in Eqn. A.15. Limiter-function forms have been presented corresponding to this averaging function in popular sources, such Hirsch[125]. Denoting the ratios p/q by r and ε/q by s, the limiter presented in these sources is,

$$\psi(r) = \frac{(r+1)r}{r^2+1}$$

which, lacking ε, differs from Eqn. A.14,

$$\psi(r) = \frac{(r+1)(r+s^2)}{r^2+1+2s^2}$$

The more popular forms of this limiter do not turn off in smooth regions, thus missing an essential feature of the original Van Albada formulation.

Appendix B

Governing Equations

B.1 Compressible Continuum Gas Dynamics

The Navier[87]-Stokes[88] equations of mass, momentum, and energy conservation for compressible gas dynamics are presented. The formulations are in terms of Cartesian coordinates, but include an axisymmetric source term,[1] $\mathbf{B}$, which is set to zero for two-dimensional and three-dimensional applications.

The derivation of the equations assume a Newtonian stress-strain relationship, Stokes hypothesis on the bulk viscosity, Fourier's law for heat conduction, no body forces, and no external heat addition.

Non-dimensionalization is performed for the following quantities as: length, L_{ref}, velocity, V_∞, time, L_{ref}/V_∞, energy, V_∞^2, density, ρ_∞, pressure, $\rho_\infty V_\infty^2$, viscosity, μ_∞, temperature, T_∞, thermal conductivity, $\mu_\infty V_\infty^2/T_\infty$, and gas constant and specific heats, V_∞^2/T_∞.

The Reynolds number appearing in the non-dimensional equation is defined,

$$R_{e_\infty} = \frac{\rho_\infty V_\infty L_{ref}}{\mu_\infty} \tag{B.1}$$

[1]The axisymmetric source terms were derived from the orthogonal curvilinear formulations of Back[130] and Anderson, Tannehill, and Pletcher[89].

B.1.1 Vector Notation

The system can be expressed in conserved variables as,

$$\mathbf{U}_t + \vec{\nabla}\cdot\vec{\mathbf{F}} = -\varpi\mathbf{B} \tag{B.2}$$

where $\vec{\mathbf{F}} = \vec{\mathbf{F}}^i - \vec{\mathbf{F}}^v$, $\mathbf{B} = \mathbf{B}^i - \mathbf{B}^v$, and,

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho\vec{V}^{\mathsf{T}} \\ \rho E \end{pmatrix} \tag{B.3}$$

$$\vec{\mathbf{F}}^i = \begin{pmatrix} \rho\vec{V} \\ \rho\vec{V}^T\vec{V} + PI \\ \rho\vec{V}H \end{pmatrix} \tag{B.4}$$

$$\vec{\mathbf{F}}^v = \frac{1}{R_{e_\infty}} \begin{pmatrix} 0 \\ \boldsymbol{\tau} \\ \kappa\vec{\nabla}T + \vec{V}\boldsymbol{\tau} \end{pmatrix} \tag{B.5}$$

$$\boldsymbol{\tau} = \mu\left[\vec{\nabla}^T\vec{V} + \left(\vec{\nabla}^T\vec{V}\right)^T - \frac{2}{3}\vec{\nabla}\cdot\vec{V}I\right] \tag{B.6}$$

$$\mathbf{B}^i = \frac{1}{y}\begin{pmatrix} \rho v \\ \rho v\vec{V}^{\mathsf{T}} \\ \rho v H \end{pmatrix} \tag{B.7}$$

$$\mathbf{B}^v = \frac{1}{R_{e_\infty}y}\begin{pmatrix} 0 \\ \left[\mu(\vec{V}_y + \vec{\nabla}v) - \frac{2}{3}\vec{\nabla}(\mu v) - (0, \frac{4}{3})\frac{\mu v}{y}\right]^{\mathsf{T}} \\ \kappa T_y + \mu\vec{V}\cdot(\vec{V}_y + \vec{\nabla}v) - \frac{2}{3}\vec{V}\cdot\vec{\nabla}(\mu v) - \frac{4}{3}\mu v\vec{\nabla}\cdot\vec{V} \end{pmatrix} \tag{B.8}$$

ϖ is the axisymmetric flag, equal to unity for axisymmetric equations and equal to zero for Cartesian equations.

Alternatively, Yu[131] shows the equations can be written,

$$\varpi_a\mathbf{U}_t + \vec{\nabla}\cdot(\varpi_a\vec{\mathbf{F}}) = \varpi\mathbf{B}' \tag{B.9}$$

where,

$$\varpi_a = 1 - \varpi + \varpi y \tag{B.10}$$

and

$$\mathbf{B}' = \mathbf{B}'^{i} - \mathbf{B}'^{v} \tag{B.11}$$

$$\mathbf{B}'^{i} = (0, 0, P, 0)^{\mathsf{T}} \tag{B.12}$$

$$\mathbf{B}'^{v} = \frac{1}{R_{e_\infty}}\left(0, 0, \frac{2\mu v}{y} - \frac{2}{3}\mu\vec{\nabla}\cdot\vec{V}, 0\right)^{\mathsf{T}} \tag{B.13}$$

Also, when using the form Eqn. B.9 the divergence of the velocity in the viscous terms, $\mathbf{F}^v$ and $\mathbf{B}'^v$, is replaced by,

$$\vec{\nabla}\cdot\vec{V} \rightarrow \frac{1}{\varpi_a}\vec{\nabla}\cdot(\varpi_a\vec{V}) = \vec{\nabla}\cdot\vec{V} + \frac{\vec{V}\cdot\vec{\nabla}\varpi_a}{\varpi_a} = \vec{\nabla}\cdot\vec{V} + \frac{\varpi v}{y} \tag{B.14}$$

B.1.2 Two-Dimensional/Axisymmetric Indicial Notation

The axisymmetric source terms are included in braces {}.

Continuity:

$$\rho_t + (\rho u)_x + (\rho v)_y = -\frac{\varpi}{y}\{\rho v\} \tag{B.15}$$

$$\varpi_a\rho_t + \varpi_a(\rho u)_x + \varpi_a(\rho v)_y = -\varpi\{\rho v\} \tag{B.16}$$

$$\varpi_a\rho_t + \varpi_a(\rho u)_x + (\varpi_a\rho v)_y = 0 \tag{B.17}$$

Momentum-x:

$$\begin{aligned}(\rho u)_t + (\rho u^2 + P)_x + (\rho uv)_y = {} & \frac{1}{R_{e_\infty}}\left(\frac{2}{3}[\mu(2u_x - v_y)]_x + [\mu(u_y + v_x)]_y\right) \\ & + \frac{\varpi}{y}\left\{-\rho uv + \frac{1}{R_{e_\infty}}\left[\mu(u_y + v_x) - \frac{2}{3}(\mu v)_x\right]\right\}\end{aligned} \tag{B.18}$$

$$\begin{aligned}\varpi_a(\rho u)_t + \varpi_a(\rho u^2 + P)_x + \varpi_a(\rho uv)_y = {} & \frac{\varpi_a}{R_{e_\infty}}\left(\frac{2}{3}[\mu(2u_x - v_y)]_x + [\mu(u_y + v_x)]_y\right) \\ & + \varpi\left\{-\rho uv + \frac{1}{R_{e_\infty}}\left[\mu(u_y + v_x) - \frac{2}{3}(\mu v)_x\right]\right\}\end{aligned} \tag{B.19}$$

$$\begin{aligned}&\varpi_a(\rho u)_t + \varpi_a(\rho u^2 + P)_x + (\varpi_a\rho uv)_y \\ &\quad = \frac{1}{R_{e_\infty}}\left(\frac{2}{3}\varpi_a[\mu(2u_x - v_y)]_x + [\varpi_a\mu(u_y + v_x)]_y\right) - \frac{2\varpi}{3R_{e_\infty}}\{\mu v\}_x\end{aligned} \tag{B.20}$$

Momentum-y:

$$(\rho v)_t + (\rho uv)_x + (\rho v^2 + P)_y = \frac{1}{R_{e_\infty}}\left([\mu(u_y + v_x)]_x + \frac{2}{3}[\mu(2v_y - u_x)]_y\right) + \frac{\varpi}{y}\left\{-\rho v^2 + \frac{1}{R_{e_\infty}}\left[2\mu v_y - \frac{2}{3}(\mu v)_y - \frac{4}{3}\frac{\mu v}{y}\right]\right\} \tag{B.21}$$

$$\varpi_a(\rho v)_t + \varpi_a(\rho uv)_x + \varpi_a(\rho v^2 + P)_y = \frac{\varpi_a}{R_{e_\infty}}\left([\mu(u_y + v_x)]_x + \frac{2}{3}[\mu(2v_y - u_x)]_y\right) + \varpi\left\{-\rho v^2 + \frac{1}{R_{e_\infty}}\left[2\mu v_y - \frac{2}{3}(\mu v)_y - \frac{4}{3}\frac{\mu v}{y}\right]\right\} \tag{B.22}$$

$$\varpi_a(\rho v)_t + \varpi_a(\rho uv)_x + (\varpi_a \rho v^2)_y + \varpi_a P_y = \frac{\varpi_a}{R_{e_\infty}}\left([\mu(u_y + v_x)]_x + \frac{2}{3}[\mu(2v_y - u_x)]_y\right) + \frac{2\varpi}{3R_{e_\infty}}\left\{2\mu v_y - \mu_y v - 2\frac{\mu v}{y}\right\} \tag{B.23}$$

Energy:

$$(\rho E)_t + (\rho uH)_x + (\rho vH)_y = \frac{1}{R_{e_\infty}}\left[(\kappa T_x)_x + (\kappa T_y)_y + \left[\mu\left(\frac{2}{3}u(2u_x - v_y) + v(u_y + v_x)\right)\right]_x + \left[\mu\left(u(u_y + v_x) + \frac{2}{3}v(2v_y - u_x)\right)\right]_y\right] + \frac{\varpi}{y}\left\{-\rho vH + \frac{1}{R_{e_\infty}}\left[\kappa T_y + \mu(u(u_y + v_x) + 2vv_y) - \frac{2}{3}(u(\mu v)_x + v(\mu v)_y) - \frac{4}{3}\mu v(u_x + v_y)\right]\right\} \tag{B.24}$$

$$\varpi_a(\rho E)_t + \varpi_a(\rho uH)_x + \varpi_a(\rho vH)_y = \frac{\varpi_a}{R_{e_\infty}}\left[(\kappa T_x)_x + (\kappa T_y)_y + \left[\mu\left(\frac{2}{3}u(2u_x - v_y) + v(u_y + v_x)\right)\right]_x + \left[\mu\left(u(u_y + v_x) + \frac{2}{3}v(2v_y - u_x)\right)\right]_y\right] + \varpi\left\{-\rho vH + \frac{1}{R_{e_\infty}}\left[\kappa T_y + \mu(u(u_y + v_x) + 2vv_y) - \frac{2}{3}(u(\mu v)_x + v(\mu v)_y) - \frac{4}{3}\mu v(u_x + v_y)\right]\right\} \tag{B.25}$$

$$\varpi_a(\rho E)_t + \varpi_a(\rho u H)_x + (\varpi_a \rho v H)_y = \frac{1}{R_{e_\infty}} \Bigg[\varpi_a(\kappa T_x)_x + (\varpi_a \kappa T_y)_y + \varpi_a \left[\mu \left(\frac{2}{3} u(2u_x - v_y) + v(u_y + v_x) \right) \right]_x + \left[\varpi_a \mu \left(u(u_y + v_x) + \frac{2}{3} v(2v_y - u_x) \right) \right]_y \Bigg] - \frac{2\varpi}{3R_{e_\infty}} \left\{ (\mu u v)_x + (\mu v^2)_y \right\} \tag{B.26}$$

B.2 Linearizations

The inviscid flux Jacobian is defined as,

$$\vec{\mathbf{A}} = \frac{\partial \vec{\mathbf{F}}^i}{\partial \mathbf{U}} \tag{B.27}$$

For a general gas in two dimensions with,

$$\frac{\partial P}{\partial \rho u} = -u \frac{\partial P}{\partial \rho E}, \qquad \frac{\partial P}{\partial \rho v} = -v \frac{\partial P}{\partial \rho E} \tag{B.28}$$

the Jacobian is,

$$\mathbf{A}^{\mathsf{x}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ P_\rho - u^2 & u(2 - P_{\rho E}) & -v P_{\rho E} & P_{\rho E} \\ -uv & v & u & 0 \\ u(P_\rho - H) & H - u^2 P_{\rho E} & -uv P_{\rho E} & u(1 + P_{\rho E}) \end{bmatrix} \tag{B.29}$$

$$\mathbf{A}^{\mathsf{y}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -uv & v & u & 0 \\ P_\rho - v^2 & -u P_{\rho E} & v(2 - P_{\rho E}) & P_{\rho E} \\ v(P_\rho - H) & -uv P_{\rho E} & H - v^2 P_{\rho E} & v(1 + P_{\rho E}) \end{bmatrix} \tag{B.30}$$

The projection of the Jacobian onto a unit vector can be written in compact notation,

$$\vec{\mathbf{A}} \cdot \hat{n} = \begin{bmatrix} 0 & \hat{n} & 0 \\ P_\rho \hat{n}^T - \mathcal{V}\vec{V}^T & \mathcal{V} I + \vec{V}^T \hat{n} - \hat{n}^T \vec{V} P_{\rho E} & \hat{n}^T P_{\rho E} \\ \mathcal{V}(P_\rho - H) & H\hat{n} - \mathcal{V}\vec{V} P_{\rho E} & \mathcal{V}(1 + P_{\rho E}) \end{bmatrix} \tag{B.31}$$

where the projected velocity is $\mathcal{V} = \vec{V} \cdot \hat{n}$.

Specializing to an ideal gas with an equation of state,

$$P = (\gamma - 1)\rho e = (\gamma - 1)\left(\rho E - \frac{(\rho u)^2 + (\rho v)^2}{2\rho}\right) \tag{B.32}$$

$$P_\rho = \frac{\gamma - 1}{2} V^2, \qquad P_{\rho E} = \gamma - 1 \tag{B.33}$$

where $V^2 = \vec{V} \cdot \vec{V} = u^2 + v^2$, the Jacobian becomes,

$$\mathbf{A}^{\mathsf{x}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{\gamma-1}{2}V^2 - u^2 & -u(\gamma - 3) & -v(\gamma - 1) & \gamma - 1 \\ -uv & v & u & 0 \\ u(\frac{\gamma-1}{2}V^2 - H) & H - u^2(\gamma - 1) & -uv(\gamma - 1) & \gamma u \end{bmatrix} \tag{B.34}$$

$$\mathbf{A}^{\mathsf{y}} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -uv & v & u & 0 \\ \frac{\gamma-1}{2}V^2 - v^2 & -u(\gamma - 1) & -v(\gamma - 3) & \gamma - 1 \\ v(\frac{\gamma-1}{2}V^2 - H) & -uv(\gamma - 1) & H - v^2(\gamma - 1) & \gamma v \end{bmatrix} \tag{B.35}$$

The projected Jacobian is,

$$\vec{\mathbf{A}} \cdot \hat{n} = \begin{bmatrix} 0 & n^{\mathsf{x}} & n^{\mathsf{y}} & 0 \\ -u\mathcal{V} + \frac{\gamma-1}{2}V^2 n^{\mathsf{x}} & \mathcal{V} - (\gamma - 2)un^{\mathsf{x}} & un^{\mathsf{y}} - (\gamma - 1)vn^{\mathsf{x}} & (\gamma - 1)n^{\mathsf{x}} \\ -v\mathcal{V} + \frac{\gamma-1}{2}V^2 n^{\mathsf{y}} & vn^{\mathsf{x}} - (\gamma - 1)un^{\mathsf{y}} & \mathcal{V} - (\gamma - 2)vn^{\mathsf{y}} & (\gamma - 1)n^{\mathsf{y}} \\ \mathcal{V}\left(\frac{\gamma-1}{2}V^2 - H\right) & Hn^{\mathsf{x}} - (\gamma - 1)u\mathcal{V} & Hn^{\mathsf{y}} - (\gamma - 1)v\mathcal{V} & \gamma\mathcal{V} \end{bmatrix} \tag{B.36}$$

The eigensystem for the projected Jacobian is formed as,

$$\vec{\mathbf{A}} \cdot \hat{n} = \mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1} \tag{B.37}$$

The eigenvalues are,

$$\mathbf{\Lambda} = \mathrm{diag}(\mathcal{V}, \mathcal{V}, \mathcal{V} + a, \mathcal{V} - a) \tag{B.38}$$

with the sound speed defined,

$$a^2 = \frac{\gamma P}{\rho} \tag{B.39}$$

The associated eigenvectors are the columns of $\mathbf{X}$,

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ u & -n^{\mathsf{y}} & u + an^{\mathsf{x}} & u - an^{\mathsf{x}} \\ v & n^{\mathsf{x}} & v + an^{\mathsf{y}} & v - an^{\mathsf{y}} \\ \frac{V^2}{2} & vn^{\mathsf{x}} - un^{\mathsf{y}} & H + a\mathcal{V} & H - a\mathcal{V} \end{bmatrix} \tag{B.40}$$

B.3 Variable Transformations

Transformations of the dependent variables can be performed from the conserved variables, Eqn. B.3, to the primitive variables,

$$\mathbf{V} = \begin{pmatrix} \rho \\ \vec{V}^{\mathsf{T}} \\ P \end{pmatrix} \tag{B.41}$$

satisfying $d\mathbf{U} = \mathbf{U}_V \, d\mathbf{V}$, $d\mathbf{V} = \mathbf{V}_U \, d\mathbf{U}$, and $\mathbf{U}_V^{-1} = \mathbf{V}_U$. For a general gas, the transformation matrices are,

$$\mathbf{U}_V = \begin{bmatrix} 1 & 0 & 0 \\ \vec{V}^{\mathsf{T}} & \rho I & 0 \\ V^2 - \frac{P_\rho}{P_{\rho E}} & \rho \vec{V} & \frac{1}{P_{\rho E}} \end{bmatrix} \tag{B.42}$$

$$\mathbf{V}_U = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{\rho}\vec{V}^{\mathsf{T}} & \frac{1}{\rho} I & 0 \\ P_\rho & -\vec{V} P_{\rho E} & P_{\rho E} \end{bmatrix} \tag{B.43}$$

While a perfect gas assumption leads to,

$$\mathbf{U}_V = \begin{bmatrix} 1 & 0 & 0 \\ \vec{V}^{\mathsf{T}} & \rho I & 0 \\ \frac{V^2}{2} & \rho \vec{V} & \frac{1}{\gamma - 1} \end{bmatrix} \tag{B.44}$$

$$\mathbf{V}_U = \begin{bmatrix} 1 & 0 & 0 \\ -\frac{1}{\rho}\vec{V}^{\mathsf{T}} & \frac{1}{\rho} I & 0 \\ \frac{\gamma-1}{2} V^2 & -(\gamma - 1)\vec{V} & \gamma - 1 \end{bmatrix} \tag{B.45}$$

The projected flux Jacobian, Eqn. B.31, can be transformed as,

$$\hat{n}\cdot\vec{\mathbf{A}}_P = \mathbf{V}_U \left(\vec{\mathbf{A}}\cdot\hat{n}\right) \mathbf{U}_V \tag{B.46}$$

For a general gas, Eqn. B.46 leads to,

$$\hat{n}\cdot\vec{\mathbf{A}}_P = \begin{bmatrix} \mathcal{V} & \rho\hat{n} & 0 \\ 0 & \mathcal{V}I & \frac{1}{\rho}\hat{n}^{\mathsf{T}} \\ 0 & \rho[P_\rho - P_{\rho E}(V^2 - H)]\hat{n} & \mathcal{V} \end{bmatrix} \tag{B.47}$$

while the perfect gas version is,

$$\hat{n}\cdot\vec{\mathbf{A}}_P = \begin{bmatrix} \mathcal{V} & \rho\hat{n} & 0 \\ 0 & \mathcal{V}I & \frac{1}{\rho}\hat{n}^{\mathsf{T}} \\ 0 & \gamma P\hat{n} & \mathcal{V} \end{bmatrix} \tag{B.48}$$

Introducing the auxiliary variables such that,

$$d\mathbf{W} = \begin{pmatrix} ds \\ \rho\, d\vec{V}^{\mathsf{T}} \\ dP \end{pmatrix} \tag{B.49}$$

where $ds = d\rho - \frac{1}{a^2}dP$, a transformation from the primitive variables can be introduced satisfying $d\mathbf{V} = \mathbf{V}_W\, d\mathbf{W}$, $d\mathbf{W} = \mathbf{W}_V\, d\mathbf{V}$, and $\mathbf{V}_W^{-1} = \mathbf{W}_V$, with,

$$\mathbf{V}_W = \begin{bmatrix} 1 & 0 & \frac{1}{a^2} \\ 0 & \frac{1}{\rho}I & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{B.50}$$

$$\mathbf{W}_V = \begin{bmatrix} 1 & 0 & -\frac{1}{a^2} \\ 0 & \rho I & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{B.51}$$

The projected flux Jacobian can be transformed accordingly,

$$\hat{n}\cdot\vec{\boldsymbol{\mathcal{A}}} = \mathbf{W}_V(\hat{n}\cdot\vec{\mathbf{A}}_P)\mathbf{V}_W \tag{B.52}$$

$$= \begin{bmatrix} \mathcal{V} & \left\{1-\frac{1}{a^2}[P_\rho - P_{\rho E}(V^2-H)]\right\}\hat{n} & 0 \\ 0 & \mathcal{V}I & \hat{n}^\mathsf{T} \\ 0 & [P_\rho - P_{\rho E}(V^2-H)]\hat{n} & \mathcal{V} \end{bmatrix} \tag{B.53}$$

$$= \begin{bmatrix} \mathcal{V} & 0 & 0 \\ 0 & \mathcal{V}I & \hat{n}^\mathsf{T} \\ 0 & a^2\hat{n} & \mathcal{V} \end{bmatrix} \tag{B.54}$$

The eigensystem of Eqn. B.54 can be expressed as,

$$\hat{n}\cdot\vec{\boldsymbol{\mathcal{A}}} = \boldsymbol{\mathcal{X}}\boldsymbol{\Lambda}\boldsymbol{\mathcal{X}}^{-1} \tag{B.55}$$

where the eigenvalues remain the same as for Eqn. B.38,

$$\boldsymbol{\Lambda} = \mathrm{diag}(\mathcal{V}, \mathcal{V}, \mathcal{V}+a, \mathcal{V}-a) \tag{B.56}$$

and the eigenvectors are,

$$\boldsymbol{\mathcal{X}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \hat{n}^\mathsf{y} & \hat{n}^\mathsf{x} & \hat{n}^\mathsf{x} \\ 0 & -\hat{n}^\mathsf{x} & \hat{n}^\mathsf{y} & \hat{n}^\mathsf{y} \\ 0 & 0 & a & -a \end{bmatrix} \tag{B.57}$$

$$\boldsymbol{\mathcal{X}}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \hat{n}^\mathsf{y} & -\hat{n}^\mathsf{x} & 0 \\ 0 & \frac{1}{2}\hat{n}^\mathsf{x} & \frac{1}{2}\hat{n}^\mathsf{y} & \frac{1}{2a} \\ 0 & \frac{1}{2}\hat{n}^\mathsf{x} & \frac{1}{2}\hat{n}^\mathsf{y} & -\frac{1}{2a} \end{bmatrix} \tag{B.58}$$

The conversion from conserved to auxiliary variables is,

$$d\mathbf{U} = \mathbf{U}_V\, d\mathbf{V} = \mathbf{U}_V\mathbf{V}_W\, d\mathbf{W} = \mathbf{U}_W\, d\mathbf{W} \tag{B.59}$$

$$d\mathbf{W} = \mathbf{W}_V\, d\mathbf{V} = \mathbf{W}_V\mathbf{V}_U\, d\mathbf{U} = \mathbf{W}_U\, d\mathbf{U} \tag{B.60}$$

For a perfect gas, these matrices are,

$$\mathbf{U}_W = \begin{bmatrix} 1 & 0 & \frac{1}{a^2} \\ \vec{V}^\mathsf{T} & I & \frac{1}{a^2}\vec{V}^\mathsf{T} \\ \frac{V^2}{2} & \vec{V} & \frac{1}{\gamma-1}\frac{T_0}{T} \end{bmatrix} \tag{B.61}$$

$$\mathbf{W}_U = \begin{bmatrix} 2-\frac{T_0}{T} & \frac{\vec{V}}{h} & -\frac{1}{h} \\ -\vec{V}^\mathsf{T} & I & 0 \\ \frac{\gamma-1}{2}V^2 & -(\gamma-1)\vec{V} & \gamma-1 \end{bmatrix} \tag{B.62}$$

where the enthalpy is obtained from the total enthalpy,

$$h = H - \frac{V^2}{2} \tag{B.63}$$

and the stagnation temperature is defined,

$$\frac{T_0}{T} = 1 + \frac{\gamma-1}{2}M^2 \tag{B.64}$$

Appendix C

Axisymmetric Source Term Integration

C.1 Inviscid

The perfect-gas inviscid axisymmetric source term can be written either in the form of Eqn. B.7 as,

$$\mathbf{B} = \frac{Z_3}{y}\mathbf{Z} \tag{C.1}$$

or in the alternate form of Eqn. B.12 as,

$$\mathbf{B}' = [0,\, 0,\, P,\, 0]^{\mathsf{T}} = \left[0,\, 0,\, \frac{\gamma - 1}{\gamma}\left(Z_1 Z_4 - \frac{Z_2^2 + Z_3^2}{2}\right),\, 0\right]^{\mathsf{T}} \tag{C.2}$$

The parameter vector, $\mathbf{Z}$, is assumed to vary linearly over a triangular element Ω. The integration of $\mathbf{B}$ over Ω is performed by subdividing Ω into thirds along the median-dual mesh, as depicted in Figure C.1. Notice that each sub-element Ω_i is a quadrilateral over which the parameter vector varies linearly. Also, the area of each Ω_i is one-third of the area of the triangular element,

$$\int_{\Omega_i} d\Omega_i = S_{\Omega_i} = \frac{S_{\mathsf{T}}}{3} \tag{C.3}$$

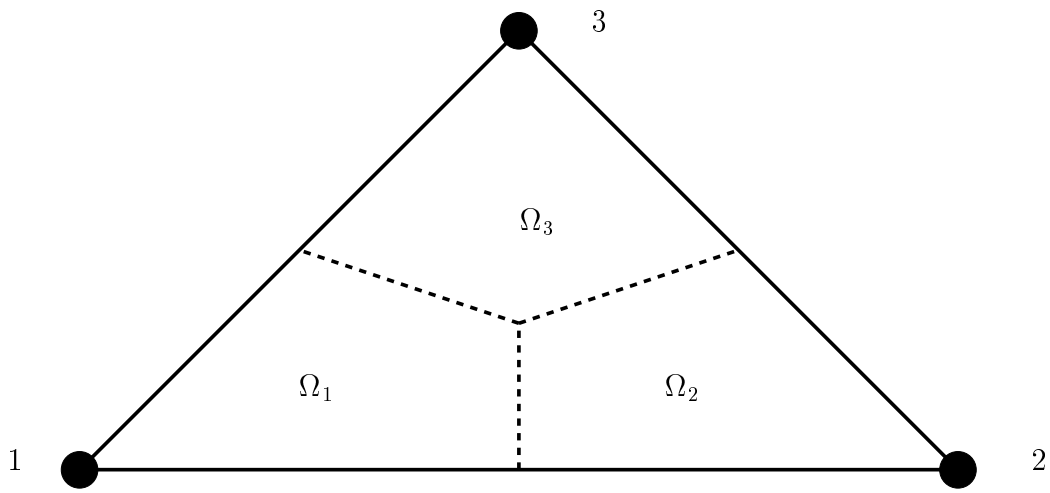


Figure C.1: Subdivision of triangular element into three quadrilateral integration areas. Dashed lines are the median-dual mesh.

A linear variation, say of the mth element of $\mathbf{Z}$, can be represented as,

$$\begin{aligned} Z_m(x,y) &= \frac{1}{2S_{\mathsf{T}}}\left(a_m + b_m x + c_m y\right) \\ &= \frac{1}{2S_{\mathsf{T}}}\epsilon_{ijk} Z_{m_j}\left[(x-x_i)(y_k-y_i)+(y-y_i)(x_i-x_k)\right] \end{aligned} \tag{C.4}$$

where,

$$\begin{aligned} a_m &= Z_{m_1}(x_2y_3 - x_3y_2) + Z_{m_2}(x_3y_1 - x_1y_3) + Z_{m_3}(x_1y_2 - x_2y_1) \\ &= \epsilon_{ijk} Z_{m_j}(x_k y_i - x_i y_k) \end{aligned} \tag{C.5}$$

$$\begin{aligned} b_m &= Z_{m_1}(y_2 - y_3) + Z_{m_2}(y_3 - y_1) + Z_{m_3}(y_1 - y_2) \\ &= \epsilon_{ijk} Z_{m_j}(y_k - y_i) \end{aligned} \tag{C.6}$$

$$\begin{aligned} c_m &= Z_{m_1}(x_3 - x_2) + Z_{m_2}(x_1 - x_3) + Z_{m_3}(x_2 - x_1) \\ &= \epsilon_{ijk} Z_{m_j}(x_i - x_k) \end{aligned} \tag{C.7}$$

In this appendix, ϵ_{ijk} is used to represent the cyclic permutation summation operator.

C.1.1 Integration of Eqn. C.2

The only non-zero term in Eqn. C.2 is $B_3' = P$. The integration over a quadrilateral sub-element can be broken out as,

$$\int_{\Omega_i} P\, d\Omega_i \;= \frac{\gamma-1}{\gamma}\int_{\Omega_i} Z_1 Z_4\, d\Omega_i - \frac{\gamma-1}{2\gamma}\int_{\Omega_i} Z_2^2 + Z_3^2\, d\Omega_i \tag{C.8}$$

For a linear variation of $\mathbf{Z}$, a typical term of Eqn. C.8 can be written,

$$\begin{aligned}\int_{\Omega_i} Z_m Z_n\, d\Omega_i &= \frac{1}{4S_{\mathsf{T}}^2}\int_{\Omega_i}(a_m + b_m x + c_m y)(a_n + b_n x + c_n y)\, d\Omega_i \\ &= \frac{1}{4S_{\mathsf{T}}^2}\int_{\Omega_i}\big[a_m a_n + (a_m b_n + a_n b_m)x + (a_m c_n + a_n c_m)y \\ &\qquad + (b_m c_n + b_n c_m)xy + b_m b_n x^2 + c_m c_n y^2\big]\, d\Omega_i\end{aligned} \tag{C.9}$$

Equation C.3 leads directly to,

$$\int_{\Omega_i} a_m a_n\, d\Omega_i \;= \frac{S_{\mathsf{T}}}{3} a_m a_n \tag{C.10}$$

The remaining integrals in Eqn. C.9 are evaluated using a bilinear mapping to a unit square in (ξ, η) space. Introducing the notation $i-$ and $i+$ to represent the node clockwise/counter-clockwise, respectively, from node i on triangle Ω, *ie,*

$i-$	i	$i+$
3	1	2
1	2	3
2	3	1

the mapping from (x, y) to (ξ, η) is,

$$x(\xi,\eta) \;= r_1 + r_2\xi + r_3\eta + r_4\xi\eta \tag{C.11}$$

$$y(\xi,\eta) \;= t_1 + t_2\xi + t_3\eta + t_4\xi\eta \tag{C.12}$$

where the coefficients are,

	1	2	3	4
r	x_i	$\frac{1}{2}(x_{i+} - x_i)$	$\frac{1}{2}(x_{i-} - x_i)$	$\frac{1}{6}(-x_{i-} + 2x_i - x_{i+})$
t	y_i	$\frac{1}{2}(y_{i+} - y_i)$	$\frac{1}{2}(y_{i-} - y_i)$	$\frac{1}{6}(-y_{i-} + 2y_i - y_{i+})$

ξ runs from node i, $\xi = 0$, to the midpoint of side $\overline{ii+}$, $\xi = 1$. η runs from node i, $\eta = 0$, to the midpoint of side $\overline{ii-}$, $\eta = 1$. $(\xi, \eta) = (1, 1)$ corresponds to the centroid of Ω. The inverse Jacobian of the transformation is,

$$J^{-1} = x_\xi y_\eta - x_\eta y_\xi = \frac{S_\mathsf{T}}{2}\left[1 - \frac{1}{3}(\xi + \eta)\right] \tag{C.13}$$

and a transformed differential area element is,

$$d\Omega_i = J^{-1}\, d\xi\, d\eta \tag{C.14}$$

Using the triangle centroid coordinates,

$$\bar{x} = \frac{x_1 + x_2 + x_3}{3} \qquad \bar{y} = \frac{y_1 + y_2 + y_3}{3} \tag{C.15}$$

the integrations in Eqn. C.9 take the following forms,

$$\int_{\Omega_i} x\, d\Omega_i = \frac{S_\mathsf{T}}{36}(7\bar{x} + 5x_i) \tag{C.16}$$

$$\int_{\Omega_i} y\, d\Omega_i = \frac{S_\mathsf{T}}{36}(7\bar{y} + 5y_i) \tag{C.17}$$

$$\int_{\Omega_i} xy\, d\Omega_i = \frac{S_\mathsf{T}}{144}\left[14\bar{x}\bar{y} + 11(x_i\bar{y} + \bar{x}y_i) + 9x_iy_i + \sum_{j=1}^{3} x_jy_j\right] \tag{C.18}$$

$$\int_{\Omega_i} x^2\, d\Omega_i = \frac{S_\mathsf{T}}{144}\left(14\bar{x}^2 + 22x_i\bar{x} + 9x_i^2 + \sum_{j=1}^{3} x_j^2\right) \tag{C.19}$$

$$\int_{\Omega_i} y^2\, d\Omega_i = \frac{S_\mathsf{T}}{144}\left(14\bar{y}^2 + 22y_i\bar{y} + 9y_i^2 + \sum_{j=1}^{3} y_j^2\right) \tag{C.20}$$

C.1.2 Integration of Eqn. C.1

The integration of the axisymmetric source term, as given in Eqn. C.1, over a quadrilateral sub-element is,

$$\begin{aligned}\int_{\Omega_i} B_m\, d\Omega_i &= \int_{\Omega_i} \frac{1}{y} Z_3 Z_m\, d\Omega_i \\ &= \frac{1}{4S_\mathsf{T}^2}\int_{\Omega_i} \frac{1}{y}\left(a_3 + b_3x + c_3y\right)\left(a_m + b_mx + c_my\right) d\Omega_i \\ &= BT_1 + BT_2 + BT_3 + BT_4\end{aligned} \tag{C.21}$$

where,

$$BT_1 = \frac{a_3 c_m + a_m c_3}{4S_{\mathsf{T}}^2} \int_{\Omega_i} d\Omega_i \tag{C.22}$$

$$BT_2 = \frac{b_3 c_m + b_m c_3}{4S_{\mathsf{T}}^2} \int_{\Omega_i} x \, d\Omega_i \tag{C.23}$$

$$BT_3 = \frac{c_3 c_m}{4S_{\mathsf{T}}^2} \int_{\Omega_i} y \, d\Omega_i \tag{C.24}$$

and,

$$BT_4 = \frac{1}{4S_{\mathsf{T}}^2} \int_{\Omega_i} \frac{1}{y} \left[a_3 a_m + (a_3 b_m + a_m b_3) \, x + b_3 b_m x^2 \right] d\Omega_i \tag{C.25}$$

Equation C.3 leads directly to the first integral,

$$BT_1 = \frac{a_3 c_m + a_m c_3}{12 S_{\mathsf{T}}} \tag{C.26}$$

Using the same bilinear mapping to (ξ, η) space as was described in the previous section, Eqn. C.23 can be written,

$$\begin{aligned} BT_2 &= \frac{b_3 c_m + b_m c_3}{4S_{\mathsf{T}}^2} \int_0^1 \int_0^1 (r_1 + r_2 \xi + r_3 \eta + r_4 \xi \eta) \left(1 - \frac{\xi + \eta}{3} \right) \frac{S_{\mathsf{T}}}{2} \, d\xi \, d\eta \\ &= \frac{b_3 c_m + b_m c_3}{288 S_{\mathsf{T}}} \left[24 r_1 + 11 \left(r_2 + r_3 \right) + 5 r_4 \right] \end{aligned} \tag{C.27}$$

Similarly, Eqn. C.24 becomes,

$$BT_3 = \frac{c_3 c_m}{288 S_{\mathsf{T}}} \left[24 t_1 + 11 \left(t_2 + t_3 \right) + 5 t_4 \right] \tag{C.28}$$

BT_4 in Eqn. C.25 is further simplified,

$$BT_4 = BT_{41} + BT_{42} + BT_{43} \tag{C.29}$$

where,

$$BT_{41} = \frac{a_3 a_m}{4S_{\mathsf{T}}^2} \int_{\Omega_i} \frac{1}{y} \, d\Omega_i \tag{C.30}$$

$$BT_{42} = \frac{a_3 b_m + a_m b_3}{4S_{\mathsf{T}}^2} \int_{\Omega_i} \frac{x}{y} \, d\Omega_i \tag{C.31}$$

$$BT_{43} = \frac{b_3 b_m}{4S_{\mathsf{T}}^2} \int_{\Omega_i} \frac{x^2}{y} \, d\Omega_i \tag{C.32}$$

Using the coordinate mapping in Eqns. C.11–C.12 to transform Eqn. C.30 leads to,

$$\begin{aligned} BT_{41} &= \frac{a_3 a_m}{24 S_{\mathsf{T}}} \int_0^1 \int_0^1 \frac{3-\xi-\eta}{t_1+t_2\xi+t_3\eta+t_4\xi\eta}\, d\xi\, d\eta \\ &= \frac{a_3 a_m}{24 S_{\mathsf{T}}} \left[\int_0^1 \ln\left(\frac{t_1+t_2+(t_3+t_4)\eta}{t_1+t_3\eta}\right)\left(\frac{1}{t_2+t_4\eta}\right)\left(3-\eta+\frac{t_1+t_3\eta}{t_2+t_4\eta}\right) d\eta \right. \\ &\qquad \left. - \frac{1}{t_4}\ln\left|1+\frac{t_4}{t_2}\right| \right] \end{aligned} \tag{C.33}$$

Options for evaluating the remaining integral in Eqn. C.33 are to expand the natural logarithm in a series expansion as,

$$\begin{aligned} \ln\left(\frac{t_1+t_2+(t_3+t_4)\eta}{t_1+t_3\eta}\right) &= \ln\left(\frac{t_1+t_2}{t_1}\right) \\ &+ 2\sum_{n=1}^{\infty} \frac{1}{2n-1}\left\{ \left[\frac{(t_3+t_4)\eta}{2(t_1+t_2)+(t_3+t_4)\eta}\right]^{2n-1} - \left[\frac{t_3\eta}{2t_1+t_3\eta}\right]^{2n-1} \right\} \end{aligned} \tag{C.34}$$

or to resort to a numerical integration scheme, such as Gaussian quadrature.

Equation C.31 can be transformed as,

$$\begin{aligned} BT_{42} &= \frac{a_3 b_m + a_m b_3}{24 S_{\mathsf{T}}} \int_0^1 \int_0^1 \frac{r_1+r_2\xi+r_3\eta+r_4\xi\eta}{t_1+t_2\xi+t_3\eta+t_4\xi\eta}(3-\xi-\eta)\, d\xi\, d\eta \\ &= \frac{a_3 b_m + a_m b_3}{24 S_{\mathsf{T}}} \int_0^1 \int_0^1 \frac{1}{(t_1+t_3\eta)+(t_2+t_4\eta)\xi} \\ &\quad \cdot \left\{ (r_1+r_3\eta)(3-\eta) + \left[(r_2+r_4\eta)(3-\eta)-(r_1+r_3\eta)\right]\xi - (r_2+r_4\eta)\xi^2 \right\} d\xi\, d\eta \end{aligned} \tag{C.35}$$

The components of Eqn. C.35 are evaluated as,

$$\int_0^1 \frac{(r_1+r_3\eta)(3-\eta)}{(t_1+t_3\eta)+(t_2+t_4\eta)\xi}\, d\xi = \frac{(r_1+r_3\eta)(3-\eta)}{t_2+t_4\eta}\ln\left(\frac{t_2+t_4\eta}{t_1+t_3\eta}+1\right) \tag{C.36}$$

$$\begin{aligned} &\int_0^1 \int_0^1 \frac{(r_2+r_4\eta)(3-\eta)-(r_1+r_3\eta)}{(t_1+t_3\eta)+(t_2+t_4\eta)\xi}\xi\, d\xi\, d\eta \\ &\quad = \frac{3r_2-r_1}{t_4}\ln\left|\frac{t_4}{t_2}+1\right| + (3r_4-r_2-r_3)\left[\frac{1}{t_4} - \frac{t_2}{t_4^2}\ln\left|\frac{t_4}{t_2}+1\right|\right] \\ &\quad - \frac{r_4}{2t_4} + \frac{r_4 t_2}{t_4^2}\ln|t_2+t_4| - \frac{r_4 t_2}{t_4^2}\int_0^1 \ln|t_2+t_4\eta|\, d\eta \\ &\quad - \int_0^1 \frac{\left[(r_2+r_4\eta)(3-\eta)-(r_1+r_3\eta)\right](t_1+t_3\eta)}{(t_2+t_4\eta)^2}\ln\left(\frac{t_2+t_4\eta}{t_1+t_3\eta}+1\right) d\eta \end{aligned} \tag{C.37}$$

$$
\begin{aligned}
&\int_0^1 \int_0^1 \frac{r_2 + r_4\eta}{(t_1 + t_3\eta) + (t_2 + t_4\eta)\xi} \xi^2 \, d\xi d\eta \\
&= \int_0^1 \frac{r_2 + r_4\eta}{t_2 + t_4\eta} \left\{ \frac{1}{2} - \frac{t_1 + t_3\eta}{t_2 + t_4\eta} \left[1 - \frac{t_1 + t_3\eta}{t_2 + t_4\eta} \ln \left(\frac{t_1 + t_2 + (t_3 + t_4)\eta}{t_1 + t_3\eta} \right) \right] \right\} d\eta \\
&= \frac{r_2}{2t_4} \ln \left| \frac{t_4}{t_2} + 1 \right| + \frac{r_4}{2t_4} - \frac{r_4 t_2}{2t_4^2} \ln \left| \frac{t_4}{t_2} + 1 \right| + \frac{r_2 t_1}{t_4(t_2 + t_4)} - \frac{r_2 t_1}{t_2 t_4} \\
&\quad - \frac{r_2 t_3 + r_4 t_1}{t_4^2} \left[\ln \left| \frac{t_4}{t_2} + 1 \right| + \frac{t_2}{t_2 + t_4} - 1 \right] - \frac{r_4 t_3}{t_4^3} \left[t_2 + t_4 - 2t_2 \ln \left| \frac{t_4}{t_2} + 1 \right| - \frac{t_2^2}{t_2 + t_4} \right] \\
&\quad + \int_0^1 \frac{(r_2 + r_4\eta)(t_1 + t_3\eta)^2}{(t_2 + t_4\eta)^3} \ln \left(\frac{t_2 + t_4\eta}{t_1 + t_3\eta} + 1 \right) d\eta
\end{aligned}
\tag{C.38}
$$

Finally, Eqn. C.32 is transformed to,

$$
\begin{aligned}
BT_{43} &= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 \int_0^1 \frac{(r_1 + r_2\xi + r_3\eta + r^4\xi\eta)^2}{t_1 + t_2\xi + t_3\eta + t_4\xi\eta} (3 - \xi - \eta) \, d\xi \, d\eta \\
&= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 \int_0^1 \frac{1}{t_1 + t_2\xi + t_3\eta + t_4\xi\eta} \left\{ (r_1 + r_3\eta)^2 (3 - \eta) \right. \\
&\qquad\qquad + \left[2(r_1 + r_3\eta)(r_2 + r_4\eta)(3 - \eta) - (r_1 + r_3\eta)^2 \right] \xi \\
&\qquad + \left. \left[(r_2 + r_4\eta)^2 (3 - \eta) - 2(r_1 + r_3\eta)(r_2 + r_4\eta) \right] \xi^2 - (r_2 + r_4\eta)^2 \xi^3 \right\} d\xi \, d\eta \\
&= BT_{431} + BT_{432} + BT_{433} + BT_{434}
\end{aligned}
\tag{C.39}
$$

where,

$$
BT_{431} = \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 (r_1 + r_3\eta)^2 (3 - \eta) \left(\int_0^1 \frac{d\xi}{t_1 + t_2\xi + t_3\eta + t_4\xi\eta} \right) d\eta \tag{C.40}
$$

$$
\begin{aligned}
BT_{432} &= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 \left[2(r_1 + r_3\eta)(r_2 + r_4\eta)(3 - \eta) - (r_1 + r_3\eta)^2 \right] \\
&\quad \cdot \left(\int_0^1 \frac{\xi \, d\xi}{t_1 + t_2\xi + t_3\eta + t_4\xi\eta} \right) d\eta
\end{aligned}
\tag{C.41}
$$

$$
\begin{aligned}
BT_{433} &= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 \left[(r_2 + r_4\eta)^2 (3 - \eta) - 2(r_1 + r_3\eta)(r_2 + r_4\eta) \right] \\
&\quad \cdot \left(\int_0^1 \frac{\xi^2 \, d\xi}{t_1 + t_2\xi + t_3\eta + t_4\xi\eta} \right) d\eta
\end{aligned}
\tag{C.42}
$$

$$
BT_{434} = -\frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 (r_2 + r_4\eta)^2 \left(\int_0^1 \frac{\xi^3 \, d\xi}{t_1 + t_2\xi + t_3\eta + t_4\xi\eta} \right) d\eta \tag{C.43}
$$

Integrating Eqns. C.40–C.43 with respect to ξ and then η, where possible, yields,

$$BT_{431} = \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 \frac{(r_1 + r_3\eta)^2 (3-\eta)}{t_2 + t_4\eta} \ln\left(\frac{t_2 + t_4\eta}{t_1 + t_3\eta} + 1\right) d\eta \tag{C.44}$$

$$\begin{aligned}
BT_{432} &= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 \frac{[2(r_1 + r_3\eta)(r_2 + r_4\eta)(3-\eta) - (r_1 + r_3\eta)^2]}{t_2 + t_4\eta} \\
&\quad \cdot \left[1 - \frac{t_1 + t_3\eta}{t_2 + t_4\eta} \ln\left(\frac{t_2 + t_4\eta}{t_1 + t_3\eta} + 1\right)\right] d\eta \\
&= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \left\{ \frac{6r_1r_2 - r_1^2}{t_4} \ln\left|\frac{t_4}{t_2} + 1\right| \right. \\
&\qquad + \frac{6r_1r_4 + 6r_2r_3 - 2r_1r_2 - 2r_1r_3}{t_4} \left(1 - \frac{t_2}{t_4} \ln\left|\frac{t_4}{t_2} + 1\right|\right) \\
&\qquad + \frac{6r_3r_4 - 2r_1r_4 - 2r_2r_3 - r_3^2}{t_4} \left[\frac{1}{2} - \frac{t_2}{t_4}\left(1 - \frac{t_2}{t_4} \ln\left|\frac{t_4}{t_2} + 1\right|\right)\right] \\
&\qquad - \frac{r_3r_4}{t_4} \left\{\frac{1}{3} - \frac{t_2}{t_4}\left[\frac{1}{2} - \frac{t_2}{t_4}\left(1 - \frac{t_2}{t_4} \ln\left|\frac{t_4}{t_2} + 1\right|\right)\right]\right\} \\
&\qquad \left. - \int_0^1 \frac{[2(r_1 + r_3\eta)(r_2 + r_4\eta)(3-\eta) - (r_1 + r_3\eta)^2]\,(t_1 + t_3\eta)}{(t_2 + t_4\eta)^2} \right. \\
&\qquad \left. \cdot \ln\left(\frac{t_2 + t_4\eta}{t_1 + t_3\eta} + 1\right) d\eta \right\}
\end{aligned} \tag{C.45}$$

$$\begin{aligned}
BT_{433} &= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \int_0^1 \frac{(r_2 + r_4\eta)^2 (3-\eta) - 2(r_1 + r_3\eta)(r_2 + r_4\eta)}{t_2 + t_4\eta} \\
&\quad \cdot \left\{\frac{1}{2} - \frac{t_1 + t_3\eta}{t_2 + t_4\eta}\left[1 - \frac{t_1 + t_3\eta}{t_2 + t_4\eta} \ln\left(\frac{t_2 + t_4\eta}{t_1 + t_3\eta} + 1\right)\right]\right\} d\eta \\
&= \frac{b_3 b_m}{24 S_{\mathsf{T}}} \left\{ BT_{4331} + BT_{4332} + \int_0^1 \frac{(r_2 + r_4\eta)^2 (3-\eta) - 2(r_1 + r_3\eta)(r_2 + r_4\eta)}{t_2 + t_4\eta} \right. \\
&\qquad \left. \cdot \frac{(t_1 + t_3\eta)^2}{(t_2 + t_4\eta)^2} \ln\left(\frac{t_2 + t_4\eta}{t_1 + t_3\eta} + 1\right) d\eta \right\}
\end{aligned} \tag{C.46}$$

$$\begin{aligned}
BT_{4331} &= \frac{1}{2} \int_0^1 \frac{1}{t_2 + t_4\eta} \left[(3r_2^2 - 2r_1r_2) + (6r_2r_4 - r_2^2 - 2r_1r_4 - 2r_2r_3)\eta \right. \\
&\qquad \left. + (3r_4^2 - 2r_2r_4 - 2r_3r_4)\eta^2 - r_4^2\eta^3\right] d\eta \\
&= \frac{3r_2^2 - 2r_1r_2}{2t_4} \ln\left|\frac{t_4}{t_2} + 1\right| + \frac{6r_2r_4 - r_2^2 - 2r_1r_4 - 2r_2r_3}{2t_4} \left(1 - \frac{t_2}{t_4} \ln\left|\frac{t_4}{t_2} + 1\right|\right) \\
&\quad + \frac{3r_4^2 - 2r_2r_4 - 2r_3r_4}{2t_4} \left[\frac{1}{2} - \frac{t_2}{t_4}\left(1 - \frac{t_2}{t_4} \ln\left|\frac{t_4}{t_2} + 1\right|\right)\right] \\
&\quad - \frac{r_4^2}{2t_4} \left\{\frac{1}{3} - \frac{t_2}{t_4}\left[\frac{1}{2} - \frac{t_2}{t_4}\left(1 - \frac{t_2}{t_4} \ln\left|\frac{t_4}{t_2} + 1\right|\right)\right]\right\}
\end{aligned} \tag{C.47}$$

$$
\begin{aligned}
BT_{4332} = &-\int_0^1 \frac{t_1+t_3\eta}{(t_2+t_4\eta)^2}\left[(3r_2^2-2r_1r_2)+(6r_2r_4-r_2^2-2r_1r_4-2r_2r_3)\eta\right.\\
&\qquad\qquad\left.+(3r_4^2-2r_2r_4-2r_3r_4)\eta^2-r_4^2\eta^3\right]\,d\eta\\
= &\frac{t_1}{t_2}\left(\frac{2r_1r_2-3r_2^2}{t_2+t_4}\right)\\
&+\frac{t_1(r_2^2+2r_1r_4+2r_2r_3-6r_2r_4)+t_3(2r_1r_2-3r_2^2)}{t_4^2}\left(-\frac{t_4}{t_2+t_4}+\ln\left|\frac{t_4}{t_2}+1\right|\right)\\
&+\frac{t_1(2r_2r_4+2r_3r_4-3r_4^2)+t_3(r_2^2+2r_1r_4+2r_2r_3-6r_2r_4)}{t_4^2}\\
&\cdot\left(1+\frac{t_2}{t_2+t_4}-2\frac{t_2}{t_4}\ln\left|\frac{t_4}{t_2}+1\right|\right)\\
&+\frac{t_1r_4^2+t_3(2r_2r_4+2r_3r_4-3r_4^2)}{2t_4^2}\left[\frac{t_4}{t_2+t_4}\right.\\
&\qquad\qquad\left.-3\frac{t_2}{t_4}\left(1+\frac{t_2}{t_2+t_4}-2\frac{t_2}{t_4}\ln\left|\frac{t_4}{t_2}+1\right|\right)\right]\\
&+\frac{t_3r_4^2}{3t_4^2}\left[\frac{t_4-2t_2}{t_2+t_4}+6\frac{t_2^2}{t_4^2}\left(1+\frac{t_2}{t_2+t_4}-2\frac{t_2}{t_4}\ln\left|\frac{t_4}{t_2}+1\right|\right)\right]
\end{aligned}
\tag{C.48}
$$

and,

$$
\begin{aligned}
BT_{434} = &-\frac{b_3b_m}{24S_{\mathsf{T}}}\int_0^1\frac{(r_2+r_4\eta)^2}{t_2+t_4\eta}\\
&\cdot\left\{\frac{1}{3}-\frac{t_1+t_3\eta}{t_2+t_4\eta}\left[\frac{1}{2}-\frac{t_1+t_3\eta}{t_2+t_4\eta}\left(1-\frac{t_1+t_3\eta}{t_2+t_4\eta}\ln\left(\frac{t_2+t_4\eta}{t_1+t_3\eta}+1\right)\right)\right]\right\}\,d\eta\\
= &\frac{b_3b_m}{24S_{\mathsf{T}}}\left[BT_{4341}+BT_{4342}+BT_{4343}\right.\\
&\qquad\qquad\left.+\int_0^1\frac{(r_2+r_4\eta)^2(t_1+t_3\eta)^3}{(t_2+t_4\eta)^4}\ln\left(\frac{t_2+t_4\eta}{t_1+t_3\eta}+1\right)\right]\,d\eta
\end{aligned}
\tag{C.49}
$$

$$
\begin{aligned}
BT_{4341} = &-\frac{1}{3}\int_0^1\frac{(r_2+r_4\eta)^2}{t_2+t_4\eta}\,d\eta\\
= &\frac{-1}{3t_4}\left[r_2r_4+\frac{r_4^2}{2}+\frac{r_2t_4-r_4t_2}{t_4}\left(r_4+\frac{r_2t_4-r_4t_2}{t_4}\ln\left|\frac{t_4}{t_2}+1\right|\right)\right]
\end{aligned}
\tag{C.50}
$$

$$
\begin{aligned}
BT_{4342} &= \frac{1}{2}\int_0^1 \frac{(r_2+r_4\eta)^2(t_1+t_3\eta)}{(t_2+t_4\eta)^2}\,d\eta \\
&= \frac{r_2^2 t_1}{2t_2(t_2+t_4)} + \frac{2r_2r_4t_1+r_2^2t_3}{2t_4^2}\left(\ln\left|\frac{t_4}{t_2}+1\right| - \frac{t_4}{t_2+t_4}\right) \\
&\quad + \frac{r_4^2t_1+2r_2r_4t_3}{2t_4^3}\left(\frac{2t_2t_4+t_4^2}{t_2+t_4} - 2t_2\ln\left|\frac{t_4}{t_2}+1\right|\right) \\
&\quad + \frac{r_4^2t_3}{4t_4}\left[\frac{1}{t_2+t_4}\left(1-\frac{4t_2^2+2t_2t_4}{t_4^2}\right) + \frac{4t_2^2}{t_4^3}\ln\left|\frac{t_4}{t_2}+1\right|\right]
\end{aligned}
\tag{C.51}
$$

$$
\begin{aligned}
BT_{4343} &= -\int_0^1 \frac{(r_2+r_4\eta)^2(t_1+t_3\eta)^2}{(t_2+t_4\eta)^3}\,d\eta \\
&= -\frac{r_2^2t_1^2(2t_2+t_4)}{2t_2^2(t_2+t_4)^2} - \frac{r_2^2t_1t_3+r_2r_4t_1^2}{t_2(t_2+t_4)^2} \\
&\quad + \frac{r_2^2t_3^2+r_4^2t_1^2+4r_2r_4t_1t_3}{2t_4^3}\left(\frac{3}{2} - \frac{t_2(3t_2+4t_4)}{(t_2+t_4)^2} - \ln\left|\frac{t_4}{t_2}+1\right|\right) \\
&\quad - \frac{r_4t_3(r_2t_3+r_4t_1)}{t_4^4}\left[\frac{2t_4^3-9t_2^3-12t_2^2t_4}{(t_2+t_4)^2} + 3t_2\left(\frac{3}{2} - \ln\left|\frac{t_4}{t_2}+1\right|\right)\right] \\
&\quad - \frac{r_4^2t_3^2}{2t_4^5}\left[\frac{t_4^4-4t_2t_4^3+9t_2^4+12t_2^3t_4}{(t_2+t_4)^2} - 6t_2^2\left(\frac{3}{2} - \ln\left|\frac{t_4}{t_2}+1\right|\right)\right]
\end{aligned}
\tag{C.52}
$$

Appendix D

Publications

The following publications have resulted from this dissertation.

Journal articles

- W. A. Wood and W. L. Kleb, Diffusion Characteristics of Finite Volume and Fluctuation Splitting Schemes. *Journal of Computational Physics*, v. 153, n. 2, pp. 353–377, August 1999.

Technical reports

- W. A. Wood, Equivalence of Fluctuation Splitting and Finite Volume for One-Dimensional Gas Dynamics. NASA TM 206271, October 1997.
- W. A. Wood and W. L. Kleb, Comments on the Diffusive Behavior of Two Upwind Schemes. NASA/TM–1998–208738, October 1998.

Conference papers

- W. A. Wood and W. L. Kleb, Diffusion Characteristics of Upwind Schemes on Unstructured Triangulations. AIAA Paper 98–2443, June 1998.
- W. A. Wood and W. L. Kleb, On Multi-dimensional Unstructured Mesh Adaption. AIAA Paper 99–3254, June 1999.

- W. A. Wood and W. L. Kleb, 2-D/Axisymmetric Formulation of Multi-Dimensional Upwind Scheme. AIAA Paper 2001–2630, June 2001.

Future publications are anticipated for the following topics: the placement of eigenvalue limiting within the fluctuation splitting context from chapters 3 and 5; the weak formulation of the boundary condition for the upwind fluctuation splitting distribution scheme from chapters 3 and 5; and the adapted and unadapted capsule results of chapter 6.

Appendix E

Navier-Stokes Solver and Mesh Adaption Code

The source code, written in C, for the two-dimensional and axisymmetric fluctuation splitting and DMFDSFV Navier-Stokes solvers along with the curvature clustering and fluctuation minimization mesh adaption schemes is included. The codes are actually written as the text documents shown here, and are converted to executable code by a PERL script as part of a MAKEFILE compilation.

On the Implementation of New Algorithms for Two-dimensional/Axisymmetric Aerothermodynamics

William A. Wood*
NASA Langley Research Center

October 27, 2001

Abstract

HACK BOY is the developmental wrapper for exercising the various functions used to build the Navier-Stokes solver.

Input and output are in dimensional quantities. SI units are assumed, where necessary.

Nondimensionalizations: length $= L_\infty$, density $= \rho_\infty$, velocity $= V_\infty$, viscosity $= \mu_\infty$, temperature $= T_\infty$, pressure $= \rho_\infty V_\infty^2$, specific heat and gas constant $= V_\infty^2/T_\infty$, energy $= V_\infty^2$, time $= L_\infty/V_\infty$, thermal conductivity $= \mu_\infty V_\infty^2/T_\infty$, heat-transfer rate $= \rho_\infty V_\infty^3$.

*Aerospace Technologist, Aerothermodynamics Branch, Aero- and Gas-Dynamics Division. w.a.wood@larc.nasa.gov (757) 864-8355.

Contents

1 Hack Boy

Begin by defining global variables. Then sequence the input/output, track convergence, supervise global iterations.

```
#include <math.h>        /*  sqrt, acos             */
#include <stdio.h>
#include <errno.h>       /*  error control          */
#include <string.h>      /*  strerror               */
#include <stdlib.h>      /*  malloc, div            */
#include <stddef.h>      /*  NULL                   */
#include <time.h>        /*  time, time_t           */
#include "basics.h"      /*  basic code parameters  */
#include "TriMath.h"     /*  TriaveH                */
#include "rhs.h"         /*  axi_source             */
#include "transform.h"   /*  cons2parm              */
#include "lhs.h"         /*  Bronowski              */
#include "adaption.h"    /*  mesh_adapt             */

/*  Global variable definitions  */

/*  File names  */
char rootf[17]    ;    /*  root file name           */
char inpf[21]     ;    /*  input deck file name     */
char ingridf[21]  ;    /*  input grid file name     */
char insolf[21]   ;    /*  input solution file name  */
char inrstf[21]   ;    /*  restart file name        */
char outplotf[21] ;    /*  plot file name           */
char histplotf[23];    /*  history plot file name   */

/*  Initialize cell, node, and boundary edge lists   */
Node *top_node           = NULL ;    /*  first node in the list      */
Node *bottom_node        = NULL ;    /*  last node in the list       */
unsigned int nodes       = 0    ;    /*  number of nodes             */
Cell *top_cell           = NULL ;    /*  first cell in the list      */
Cell *bottom_cell        = NULL ;    /*  last cell in the list       */
unsigned int cells       = 0    ;    /*  number of cells             */
Bound_edge *top_bedge    = NULL ;    /*  first boundary edge in list */
Bound_edge *bottom_bedge = NULL ;    /*  last boundary edge in list  */
unsigned int bedges      = 0    ;    /*  number of boundary edges    */
Edge *top_edge           = NULL ;    /*  first edge in list          */
Edge *bottom_edge        = NULL ;    /*  last edge in list           */
unsigned int edges       = 0    ;    /*  number of edges             */

/*  control parameters  */
int axisym        = 0    ; /*  axisymmetric flag                      */
int memory        = 0    ; /*  memory reduction level 0,1,2           */
int safety        = 0    ; /*  double-check flag                      */
int scalar        = 0    ; /*  scalar flag                            */
```

```
int solver        = 0    ; /*  FS or FV                                  */
int viscous       = 1    ; /*  inviscid(0), TLNS median dual(1),         */
                           /* TLNS containment dual(2), NSMD(3), NSCD(4)*/
int traceback     = 0    ; /*  traceback level 0,1,2,3                   */
float CFL         = 1    ; /*  timestep factor                           */
float CFL_v       = 0    ; /*  viscous timestep factor                   */
float L2_done     = 1e-6 ; /*  stop criteria for norm                    */
int itr_done      = 1000 ; /*  stop criteria for iterations              */
time_t cpu_done   = 3600 ; /*  maximum run time in seconds               */
int hist_skip_screen = 1 ; /*  history iteration skip factor to screen  */
int hist_skip_file   = 1 ; /*  history iteration skip factor to file    */
int plot_skip        = 1 ; /*  iteration skip factor for plot file      */
double swap_thresh   = 0 ; /*  edge-swapping trigger threshold          */
double delete_thresh = 0 ; /*  point-deletion trigger threshold         */
double add_thresh    = 0 ; /*  point-insertion trigger threshold        */
double move_thresh   = 0 ; /*  node-movement trigger threshold          */
double adapt_target  = 1 ; /*  target adaption error level              */

/*  non-dimensionalization  */
double nondim_L   = 0 ;    /*  length                      */
double nondim_V   = 0 ;    /*  velocity                    */
double nondim_T   = 0 ;    /*  temperature                 */
double nondim_rho = 0 ;    /*  density                     */
double nondim_mu  = 0 ;    /*  viscosity                   */
double T_wall     = 0 ;    /*  cold-wall temperature       */
float  AOA        = 0 ;    /*  angle of attack             */
float  Mach       = 0 ;    /*  freestream Mach number      */
float  Reynolds   = 0 ;    /*  freestream Reynolds number  */


int main( int argc, char *argv[] )
{
  Node *node ;
  Cell *cell ;
  Bound_edge *bedge ;
  int i, nes, nds ;          /* dummy indicies              */
  double distrib[NEQS][3] ;  /* 4 eqns, 3 nodes per cell    */
  float L2 = 1;              /* L2 norm of solution update  */
/*  float L2_last = 1; */
  int iteration = 0 ;        /* iteration counter           */
  time_t cpu_elapsed = 0, start_time = time(NULL) ;
  double t;                  /*  timestep                   */
  int time_accurate = 0;     /*  turn time accuracy off/on  */
  mesh_adapt = none;
```

1.1 Start-Up

Begin the code by getting the global options, a grid to discretize the domain, and a solution vector to begin working with.


```
  Stonehenge( argc, argv );
  Aetna( 0, "Monkeys lit the fire");
  Aetna_node( 0, "Finished Stonehenge");
```

1.2 Mesh Adaption

Perform mesh adaption as desired before evolving solution on the new mesh. If performing adaption to edge error estimates, first build list of edges and evaluate the estimates. In fact, always form the edge errors because we want an edge list.

```
  edge_errors();
  switch( mesh_adapt ){
  case edge_swap_FM: /*  fall through  */
  case edge_swap_H:
    puts("\nPerforming edge-swapping");
    global_error_swap();
    break;
  case delete_FM:    /*  fall through  */
  case delete_H:
    puts("\nPerforming point deletion");
    global_error_delete();
    break;
  case add_FM:       /*  fall through  */
  case add_H:
    puts("\nPerforming point insertion");
    global_error_insert();
    break;
  case move_FM:      /*  fall through  */
  case move_H:
    puts("\nPerforming nodal displacement");
    global_error_move();
    break;
  default:
    puts("Not performing any mesh adaption");
  }
  Aetna_node( 0, "Back from mesh adaption");
```

1.3 Main Loop

Perform loops over the domain to evolve the solution in pseudo-time.

```
  while( iteration < itr_done && L2 > L2_done && cpu_elapsed < cpu_done ){
```

1.3.1 Preliminaries

Begin by zeroing the nodal updates and decoding the parameter and primitive vectors.

```
NODE_LOOP_FORWARD{
  FOR(i,0,NEQS){
    node->update.source[i] = node->update.flux[i]
                           = node->update.viscous[i]
                           = node->update.wall[i]
                           = node->update.art_diss[i] = 0 ;
    if( solver ) node->fv.gradx[i]= node->fv.grady[i]= node->fv.psi[i]=0;
    else node->update.source_fs[i] = 0 ;
  }
  conserved_to_parameter( node->vars.conserved, node->vars.parameter );
  conserved_to_primitive( node->vars.conserved, node->vars.primitive );
}
if( traceback > 1 ) puts("  Zeroed updates");
```

Compute nodal gradients and limiter for finite volume.

```
if( solver ){
  node_gradients( 1 );                    /*  reconstruction: 0=conserved*/
  NODE_LOOP_FORWARD fv_limit_node( 1, node, node->fv.psi ); /* 1=primitive*/
  if( traceback > 1 ) puts("  Computed FV gradients and limiter");
}
Aetna_node( 0, "ready to start cell loop");
```

1.3.2 Assemble RHS

Now loop over all cells. First compute cell-averaged quantities. Then axisymmetric source term distributions, for which FS has one more additional term than FV. Next are inviscid flux distributions. Finally, the viscous terms are distributed.

```
CELL_LOOP_FORWARD{
  Roe_average_cell( cell );

  if( axisym ){                                /*  axisym source term  */
    axi_source( cell, distrib );
    FOR(nes,0,NEQS) FOR(nds,0,3)
      (cell->geom.connect.nodes[nds])->update.source[nes]
      += distrib[nes][nds] ;

    if( ! solver ){                              /*  extra FS source  */
      FS_flux_source( cell, distrib );
      FOR(nes,0,NEQS) FOR(nds,0,3)
        (cell->geom.connect.nodes[nds])->update.source_fs[nes]
        -= distrib[nes][nds] ;
```



```
    }
  }
                                                  /*  inviscid fluxes  */
  ( ! solver ) ? FS_distrib_cell( cell ) : FV_distrib_cell( cell );

  if( viscous ){                                   /*  viscous fluxes  */
    if( viscous == 1 || viscous == 2 )
      Haselbacher( cell, distrib );             /* thin-layer treatment */
    else if( viscous == 3 || viscous == 4 )
      full_viscous( cell, distrib );                      /*  full N-S  */
    else{
      printf("Error, viscous = %d\nExiting\n", viscous );
      exit(1);
    }
    FOR(nes,0,NEQS) FOR(nds,0,3)
      (cell->geom.connect.nodes[nds])->update.viscous[nes]
      += distrib[nes][nds] ;
  }
  Aetna_node( 0, "Finished this cell");
}
if( traceback > 1 ) puts("  Computed inviscid distributions");
Aetna( 0, "Performed interior update");
Aetna_node( 0, "Finished interior update");
```

Impose weak boundary conditions. Finite volume accounts for the flux through boundary edges. Fluctuation splitting distributes a boundary fluctuation, also formed from a loop over boundary edges. Viscous boundary condition only applied for viscous walls, on a second loop.

```
BEDGE_LOOP_FORWARD{
  if( ! solver ) FS_distrib_boundary( bedge );
  else FV_distrib_boundary( bedge );
  Aetna( 0, "Finished this edge");
}
Aetna_node( 0, "finished inviscid boundary update");
BEDGE_LOOP_FORWARD
  if( viscous && bedge->bc == wall_viscous ) Hasel_bc( bedge );
if( traceback > 1 ) puts("  applied boundary conditions");
Aetna( 0, "Performed boundary update");
Aetna_node( 0, "finished viscous boundary update");
```

1.3.3 Temporal Evolution

Evolve the nodal solutions in time. Then query for output of solution history.

```
L2 = 0 ;
t = sync_time();
```

```
    if ( !time_accurate ) t = 0;
    NODE_LOOP_FORWARD L2 += Bronowski( node, t );
    L2 = sqrt( L2 );
    ++iteration ;
    cpu_elapsed = time(NULL) - start_time ;
    if( ! (div(iteration,hist_skip_screen)).rem ){
      printf("Itr=%6d, CPU=%6ds, Log(L-2)= % .2f",
             iteration, (int) cpu_elapsed, log10(L2) );
      puts("");
    }
    if( ! (div(iteration,hist_skip_file)).rem )
      write_hist( iteration, cpu_elapsed, L2, 0 );
    if( ! (div(iteration,plot_skip)).rem ) write_tecplot_0();
    if( traceback > 0 ) printf("Completed %d iterataions\n", iteration );
    Aetna_node( 0, "evolution done");

/*     CFL += ( L2 > L2_last ) ? -.1 : .01 ;  */ /*  adaptive courant number  */
/*     CFL = MIN( 1, MAX( .3, CFL ) ); */
/*     L2_last = L2 ; */

  }                                   /*  That is the end of the main loop  */
  write_hist( iteration, cpu_elapsed, L2, 0 );         /*  write last point  */
```

1.4 Shut-Down

End the program by writing plot files.

```
  WestWorld();
  return 0;
}
```

1.5 Collection of General Purpose Functions

Gather together here some functions, such as for opening files, that might be of general use throughout the program.

1.5.1 Open File

Function to open a file with error checking.

```
FILE *open_file( char *filename, char *mode )
{
  FILE *f ;
  errno = 0 ;
  f = fopen( filename, mode );
  if( f == NULL ){
    fprintf( stderr, "Open of file %s failed: %s\nExiting\n",
```



```
             filename, strerror( errno ));
    exit(1) ;
  }
  return f ;
}
```

1.5.2 Create a New Node

Function to allocate a new node. Assign or null all pointers to linked lists.

```
/*  Allocate a new node, updating linked-list pointers  */
Node *add_node( void )
{
  Node *ptr_new = (Node *) malloc( sizeof( Node ) );
  if( ptr_new ){
    ptr_new->links.previous_node = bottom_node;
    ptr_new->links.next_node     = NULL;
    ptr_new->geom.cellcon        = NULL;
    ptr_new->geom.edgecon        = NULL;
    ptr_new->adapt.error         = 0;
    if( nodes ) bottom_node->links.next_node = ptr_new ;
    else top_node = ptr_new ;
    bottom_node = ptr_new ;
    ptr_new->links.node_number = ++nodes;
  }
  else{
    puts("Ran out of memory requesting a new node");
    printf("Currently have %u nodes and %u cells\n", nodes, cells );
  }
  return ptr_new ;
}
```

1.5.3 Create a New Cell

Function to allocate a new cell. Assign or null all pointers to linked lists.

```
/*  Allocate a new cell, updating linked-list pointers  */
Cell *add_cell( void )
{
  Cell *ptr_new = (Cell *) malloc( sizeof( Cell ) );
  if( ptr_new ){
    ptr_new->links.previous_cell = bottom_cell ;
    ptr_new->links.next_cell = NULL ;
    if( cells )
      bottom_cell->links.next_cell = ptr_new ;
    else
      top_cell = ptr_new ;
```

```
    bottom_cell = ptr_new ;
    ++cells ;
  }
  else{
    puts("Ran out of memory requesting a new cell");
    printf("Currently have %u nodes and %u cells\n", nodes, cells );
  }
  return ptr_new ;
}
```

1.5.4 Delete a cell

Remove a cell from memory.

```
void kill_cell( Cell *cell ){
  if( cell != top_cell )
    cell->links.previous_cell->links.next_cell = cell->links.next_cell;
  else top_cell = cell->links.next_cell;
  if( cell != bottom_cell )
    cell->links.next_cell->links.previous_cell = cell->links.previous_cell;
  else bottom_cell = cell->links.previous_cell;
  free( cell );
  --cells;
}
```

1.5.5 Create an edge

```
Edge *add_edge( void ){
  int i;

  Edge *this_edge = (Edge *) malloc( sizeof( Edge ) );
  if( this_edge ){
    if( !edges++ ){
      top_edge = this_edge;
      this_edge->previous_edge = NULL;
    }
    else{
      bottom_edge->next_edge = this_edge;
      this_edge->previous_edge = bottom_edge;
    }
    bottom_edge = this_edge;

    this_edge->cell_left  = NULL;
    this_edge->cell_right = NULL;
    this_edge->next_edge  = NULL;
    this_edge->error      = 0;
```



```
    FOR(i,0,2){
      this_edge->node[i] = NULL;
      this_edge->r[i] = 0;
    }
  }
  else{
    puts("Not enough memory to allocate another edge");
    printf("nodes=%u, cells=%u, edges=%u, bedges=%u\n",
           nodes, cells, edges, bedges );
    exit(1);
  }
  return this_edge;
}
```

1.5.6 Delete an edge

Remove an edge from memory. If it is a boundary edge, remove from BEDGE list first.

```
void kill_edge( Edge *edge ){
  Bound_edge *bedge;

  if( edge->cell_right == NULL )
    BEDGE_LOOP_FORWARD
      if( bedge->node[1] == edge->node[0] ){
        if( bedge != top_bedge ) bedge->prior->next = bedge->next;
        else top_bedge = bedge->next;
        if( bedge != bottom_bedge ) bedge->next->prior = bedge->prior;
        else bottom_bedge = bedge->prior;
        free( bedge );
        --bedges;
        break;
      }

  if( edge != top_edge )
    edge->previous_edge->next_edge = edge->next_edge;
  else top_edge = edge->next_edge;
  if( edge != bottom_edge )
    edge->next_edge->previous_edge = edge->previous_edge;
  else bottom_edge = edge->previous_edge;
  free( edge );
  --edges;
}
```

1.5.7 Delete a node

Remove a node from memory.

```
void kill_node( Node *node ){
  Node2Cell *pn2c, *pn2c_next;
  Node2Edge *pn2e, *pn2e_next;

  if( node != top_node )
    node->links.previous_node->links.next_node = node->links.next_node;
  else top_node = node->links.next_node;
  if( node != bottom_node )
    node->links.next_node->links.previous_node = node->links.previous_node;
  else bottom_node = node->links.previous_node;

  pn2c = node->geom.cellcon;
  while( pn2c ){
    pn2c_next = pn2c->next;
    free( pn2c );
    pn2c = pn2c_next;
  }

  pn2e = node->geom.edgecon;
  while( pn2e ){
    pn2e_next = pn2e->next;
    free( pn2e );
    pn2e = pn2e_next;
  }

  free( node );
  --nodes;
}
```

1.5.8 Allocate a Node Pointer Array

Allocate memory for an array of pointers to nodes.

```
Node **allocate_node_ptr( unsigned int num )
{
  Node **ptr = (Node **) malloc( num * sizeof( Node *) );
  if( ! ptr ){
    puts("\n** Not enough memory to allocate array of node pointers\n"
         "Stopping");
    exit(1) ;
  }
  return ptr ;
}
```


1.5.9 Number Nodes in a Stack

Collect all the pointers to nodes into a vector so that the nodes can be numbered. The nodes are numbered starting with 1.

```
void stack_nodes( Node *nodal_stack[] )
{
  register Node *node ;
  register unsigned int n = 0 ;
  nodal_stack[0] = NULL ;
  NODE_LOOP_FORWARD{
    node->links.node_number = ++n ;
    nodal_stack[n] = node ;
  }
}
```

1.5.10 Point from Node to Cells

Hook a cell to a given node, to build the linked list of all cells connected at the current node. Allocate memory for a structure pointing from a node to the surrounding cells.

```
void hook_from_node_to_cell( Node *node, Cell *cell )
{
  Node2Cell **n2c = &node->geom.cellcon ;

  while( *n2c ) n2c = &(*n2c)->next ;
  if( (*n2c = (Node2Cell *) malloc( sizeof(Node2Cell) )) ){
    (*n2c)->cell = cell ;
    (*n2c)->next = NULL ;
  }
  else{
    puts("Ran out of memory creating node to cell linked list\nStopping");
    exit(1);
  }
}
```

1.5.11 Point from Node to Edges

Hook an edge to a given node, to build the linked list of all edges connected at the current node. Allocate memory for a structure pointing from a node to the surrounding edges.

```
void hook_from_node_to_edge( Node *node, Edge *edge )
{
  Node2Edge **n2e = &node->geom.edgecon ;

  while( *n2e ) n2e = &(*n2e)->next ;
```

```
  if( (*n2e = (Node2Edge *) malloc( sizeof(Node2Edge) )) ){
    (*n2e)->edge = edge ;
    (*n2e)->next = NULL ;
  }
  else{
    puts("Ran out of memory creating node to edge linked list\nStopping");
    exit(1);
  }
}
```

1.5.12 Disconnect a cell from a node

When a cell has been redefined to no longer include a given node, delete the cell from that node's the node-to-cell pointer list.

```
void break_cell_from_node( Node *node, Cell *cell ){
  Node2Cell **n2c, *dc = NULL ;

  n2c = &(node->geom.cellcon);
  while( *n2c ){
    if( (*n2c)->cell == cell ){
      dc = *n2c;
      *n2c = (*n2c)->next;
      }
    else n2c = &((*n2c)->next);
  }

  if( dc ) free( dc );
  else{
    puts("Error: failed to find a cell formerly connected to this node");
    exit(1);
  }
}
```

1.5.13 Disconnect an edge from a node

When an edge has been redefined to no longer include a given node, delete the edge from that node's the node-to-edge pointer list.

```
void break_edge_from_node( Node *node, Edge *edge ){
  Node2Edge **n2e, *de = NULL ;

  n2e = &(node->geom.edgecon);
  while( *n2e ){
    if( (*n2e)->edge == edge ){
      de = *n2e;
      *n2e = (*n2e)->next;
      }
    else n2e = &((*n2e)->next);
  }

  if( de ) free( de );
  else{
    puts("Error: failed to find an edge formerly connected to this node");
    exit(1);
  }
}
```

1.5.14 Create Boundary Edge Pointer

Allocate memory for a boundary edge definition. Update associated pointers and counters.

```
Bound_edge *add_bedge( void )
{
  Bound_edge *new_bedge = (Bound_edge *) malloc( sizeof( Bound_edge ) );
  if( new_bedge ){
    new_bedge->prior = bottom_bedge ;
    new_bedge->next = NULL ;
    if( bedges ) bottom_bedge->next = new_bedge ;
    else top_bedge = new_bedge ;
    bottom_bedge = new_bedge ;
    ++bedges ;
  }
  else{
    puts("Ran out of memory requesting new boundary edge");
    printf("Currently %u nodes, %u cells, %u bedges\n", nodes, cells, bedges);
  }
  return new_bedge ;
}
```

1.5.15 Build Boundary Edge List

Given two nodes that define an edge, see if this is a boundary edge. The logic is to first assume it is a boundary edge, adding it to the boundary edge linked list (with node order reversed). Then loop over the list to look for a duplicate. If a duplicate is found, remove the current edge (the last entry in the list) and the previously defined duplicate from the linked list.

```
void build_bedges( Node *n0, Node *n1, int safe )
{
  Bound_edge *bedge;

  bedge = add_bedge();                        /*  create the boundary edge  */
  if( bedge ){
```

```
    bedge->node[0] = n1 ;
    bedge->node[1] = n0 ;
```

Assign a boundary condition to the edge. Sequence the conditions in order of precedence.

```
    if( n0->bc == interior || n1->bc == interior ) bedge->bc = interior ;
    else if( n0->bc == vacuum || n1->bc == vacuum )
      bedge->bc = vacuum ;
    else if( n0->bc == extrapolation || n1->bc == extrapolation )
      bedge->bc = extrapolation ;
    else if( n0->bc == freestream_stagnant || n1->bc == freestream_stagnant )
      bedge->bc = freestream_stagnant ;
    else if( n0->bc == freestream_nondim || n1->bc == freestream_nondim )
      bedge->bc = freestream_nondim ;
    else if( n0->bc == freestream_second || n1->bc == freestream_second )
      bedge->bc = freestream_second ;
    else if( n0->bc == wall_inviscid || n1->bc == wall_inviscid )
      bedge->bc = wall_inviscid ;
    else if( n0->bc == wall_viscous && n1->bc == wall_viscous )
      bedge->bc = wall_viscous ;
    else{
      printf("bad boundary condition for and edge\n");
      printf("BCs: %d, %d\nStopping\n", n0->bc, n1->bc );
      exit(1);
    }

    BEDGE_LOOP_FORWARD{                        /*  look for duplicate edges  */
      if( n0 == bedge->node[0] && n1 == bedge->node[1] ){
        --bedges;                                  /*  remove current edge  */
        if( bedge != top_bedge ) bedge->prior->next = bedge->next;
        else top_bedge = bedge->next;
        bedge->next->prior = bedge->prior;
        free( bedge );
        --bedges;                                      /*  remove last edge  */
        bottom_bedge = bottom_bedge->prior;
        free( bottom_bedge->next );
        bottom_bedge->next = NULL;
        break;
      }
    }
  }
  else{
    puts("Not enough memory to allocate another boundary edge");
    printf("nodes=%u, cells=%u, bedges=%u\nStopping\n", nodes, cells, bedges );
    exit(1);
  }
```

Safety check to make sure all boundary edges have valid boundary conditions.


```
  if( safe ) BEDGE_LOOP_FORWARD
    if( bedge->bc <= interior || bedge->bc >= invalid ){
      printf("Bad value for edge boundary condition = %d\n", bedge->bc );
      printf("Node0 x=%f, y=%f\nNode1 x=%f, y=%f\n",
        (bedge->node[0])->geom.coord[0], (bedge->node[0])->geom.coord[1],
        (bedge->node[1])->geom.coord[0], (bedge->node[1])->geom.coord[1]);
    }
}
```

1.5.16 Maximum cell angle

Return the largest angle of a triangular cell, in degrees.

```
double max_cell_angle( Cell *cell ){
  Coord r;
  double l[3], a, b, c, theta, rad2deg;
  Node *n1, *n2;
  int n, i;
  rad2deg = 180. / acos( -1. );

  FOR(n,0,3){
    n1 = cell->geom.connect.nodes[n];
    n2 = cell->geom.connect.nodes[cyclic_plus0(n)];
    FOR(i,0,NDIMS) r[i] = n1->geom.coord[i] - n2->geom.coord[i];
    l[n] = VEL(r);
  }
  if( l[0] > l[1] && l[0] > l[2] ){
    c = l[0];
    a = l[1];
    b = l[2];
  }
  else if( l[1] > l[2] ){
    c = l[1];
    a = l[0];
    b = l[2];
  }
  else{
    c = l[2];
    a = l[0];
    b = l[1];
  }
  theta = acos( ( SQUARE(a) + SQUARE(b) - SQUARE(c) ) / ( 2 * a * b ) );
  return theta * rad2deg;
}
```

1.5.17 Aetna Erupts

Spew out diagnostic information.

```
void Aetna( int choice, char message[80] ){
  enum Aetna_choices{ Skip, Summary, Nodes, Cells, Bedges };
  Node *node;
  int eq ;

  if( choice ){
    printf( "Aetna erupts: %s\n", message );
    printf("%u nodes, %u cells, %u boundary edges\n\n", nodes, cells, bedges );
    switch( choice ){
    case Nodes:
      NODE_LOOP_FORWARD{
        printf("Node %u, bc = %d\n", node->links.node_number, node->bc );
        printf("x = % .4f, y = % .4f\n",
               node->geom.coord[0], node->geom.coord[1]);
        FOR(eq,0,NEQS) printf("U%d= % .4f, ", eq, node->vars.conserved[eq] );
        puts("");
        FOR(eq,0,NEQS) printf("V%d= % .4f, ", eq, node->vars.primitive[eq] );
        puts("");
        FOR(eq,0,NEQS) printf("Z%d= % .4f, ", eq, node->vars.parameter[eq] );
        puts("");
        FOR(eq,0,NEQS) printf("flux%d= % .4f, ", eq, node->update.flux[eq] );
        puts("");
        FOR(eq,0,NEQS) printf("diss%d= % .4f, ", eq, node->update.art_diss[eq]);
        puts("");
      }
      puts("");
      break;
    }
  }
}
```

1.5.18 Aetna Burps out a Node

Print out diagnostic information for a single node. Skip if node 0 is asked.

```
void Aetna_node( unsigned int bad_boy, char message[80] ){
  Node *n = top_node ;
  int i ;

  if( bad_boy ){
    printf("\nAetna_node: %s\n", message );

    while( n->links.node_number != bad_boy ) n = n->links.next_node ;
    printf("Node %u, bc= %d, x=% .4e, y=% .4e, dt=% .4e\n", bad_boy,
           n->bc, n->geom.coord[0], n->geom.coord[1], n->update.dt );
    printf("dual area=% .4e, bound_face=% .4e\n", n->geom.dual_area,
           n->geom.bound_face );
    FOR(i,0,4) printf("U%d=% .5e, ", i, n->vars.conserved[i] ); puts("");
```



```
    FOR(i,0,4) printf("V%d=% .5e, ", i, n->vars.primitive[i] ); puts("");
    FOR(i,0,4) printf("Z%d=% .5e, ", i, n->vars.parameter[i] ); puts("");
    FOR(i,0,4) printf("W%d=% .5e, ", i, n->vars.wall[i]      ); puts("");
    if( axisym ){
      FOR(i,0,4) printf("Axi%d=% .5e, ", i, n->update.source[i] ); puts("");
      if( !solver ) FOR(i,0,4) printf("AxiFS%d=% .5e, ", i,
                                      n->update.source_fs[i] ); puts("");
    }
    FOR(i,0,4) printf("Flux%d=% .5e, ", i, n->update.flux[i]     ); puts("");
    FOR(i,0,4) printf("AD  %d=% .5e, ", i, n->update.art_diss[i] ); puts("");
    FOR(i,0,4) printf("visc%d=% .5e, ", i, n->update.viscous[i]  ); puts("");
    FOR(i,0,4) printf("Wall%d=% .5e, ", i, n->update.wall[i]     ); puts("");
    FOR(i,0,4) printf("dU%d=% .5e, ",   i, n->update.dU[i]       ); puts("");
    if( solver ){
      FOR(i,0,4) printf("GradX%d=% .4e, ", i, n->fv.gradx[i] ); puts("");
      FOR(i,0,4) printf("GradY%d=% .4e, ", i, n->fv.grady[i] ); puts("");
      FOR(i,0,4) printf("Psi%d=% .4e, ",   i, n->fv.psi[i]   ); puts("");
    }
    puts("");
  }
}
```

1.5.19 Aetna burps out a cell

```
void Aetna_cell( Cell *cell ){
  int i, j;

  printf("This cell has nodes at");
  FOR(i,0,3){
    printf(" (");
    FOR(j,0,NDIMS) printf(" %.4f", cell->geom.connect.nodes[i]->geom.coord[j] );
    printf(")");
  }
  puts("");
}
```

1.5.20 Aetna burps out an edge

```
void Aetna_edge( Edge *edge ){
  int i, j;

  printf("This edge has endpoints at");
  FOR(i,0,2){
    printf(" (");
    FOR(j,0,NDIMS) printf(" %.4f", edge->node[i]->geom.coord[j] );
    printf(")");
```

```
  }
  puts("\n and cells to left and right of:");
  Aetna_cell( edge->cell_left );
  if( edge->cell_right ) Aetna_cell( edge->cell_right );
  else puts("no cell right -> boundary edge");
}
```


2 Hand-built BLAS routines

Use these routines when BLAS is not available on a machine.

```
#include "blas.h"

#if BLAS_WORKS==1
double ddot( int length, double *x, int xskip, double *y, int yskip ){
  int n ;
  double z ;
  for( z=0, n=0 ; n<length ; ++n, x+=xskip, y+=yskip )
    z += *x * *y ;
  return z ;
}

double dnrm2( int length, double *x, int xskip ){
  return sqrt( ddot( length, x, xskip, x, xskip ) );
}
#endif
```

3 Ascent of Man

Evolution from chaos. Form the LHS and update the solution at a node. Send pointer to node and return L_2-norm of ΔU at the node.

Limit the update to enforce positive energy, if necessary. This hack hardwired for air and $T_{min} \sim 10\% T_\infty$.

```
#include "blas.h"       /*  dnrm2, ddot                  */
#include <math.h>       /*  fabs, sqrt                   */
#include <float.h>      /*  FLT_EPSILON                  */
#include "basics.h"
#include "lhs.h"
#include "transform.h"  /*  U -> V                       */
#include "therm.h"      /*  a(r,P), mu(T), R_air, GAMM1  */
#include "TriMath.h"    /*  cyclic_plus0, cyclic_minus0  */

float Bronowski( Node *node, double t )
{
  double S = node->geom.dual_area ;
  double pia, LHS ;
  int eq, i ;
  Node2Cell *celln ;
  int cln = 1 ;                /*  number of cells connected to node plus 1  */
  double fun, dfun, funNext ;                        /*  for preventing T<0  */
  double *con, *up ;             /*  pointers to update and conserved vars  */
  const double emin = 0.005 *R_air *nondim_T /(GAMM1 *SQUARE(nondim_V));
  const double rhomin = 0.005;
  SolVec sigm, Unext ;
```

For a point update,

$$\mathrm{LHS}\,\Delta^n U_i = \mathrm{RHS}$$

giving,

$$\Delta^n U = \frac{\mathrm{RHS}}{\mathrm{LHS}}$$

For an explicit scheme,

$$\mathrm{LHS} = \frac{\varpi_{a_i} S_i}{\tau}, \quad \tau = \Delta t$$

where ϖ_a is 1 for 2-D and y for axisymmetric.

```
  /*  For axisymmetric, use mass-lumped integration point, except for */
  /*  on the axis, where the median-dual centroid is used             */
  if( (pia = 1 - axisym + axisym * node->geom.coord[1]) < 10 * FLT_EPSILON ){
    CELL_LOOP_AT_NODE( node ){
      FOR(i,0,3)
        pia += ( ((celln->cell)->geom.connect.nodes[i])->geom.coord[1] ) /
               ( ((celln->cell)->geom.connect.nodes[i])->bc ? 2 : 4 ) ;
```



```
      ++cln ;
    }
    pia /= cln ;
  }
```

Using forward Euler explicit scheme with local time step.

```
  if( !t ) t = node->update.dt;
```

Now form the update.

```
  LHS = pia * S / t ;
  FOR(eq,0,NEQS) node->update.dU[eq] =
          ( node->update.flux[eq] + node->update.art_diss[eq]
          + ( viscous ? node->update.viscous[eq] + node->update.wall[eq] : 0 )
          + ( axisym ? node->update.source[eq]
              + ( !solver ? node->update.source_fs[eq] : 0 ) : 0 ) )
          / LHS ;
```

Hack fix to enforce positive internal energy. Hopefully only occurs during a transient. Allways run with safety enabled when done to verify no problems whose error messages were supressed. Hardwired for NEQS=4.

```
  up = node->update.dU ;
  con = node->vars.conserved ;
  FOR(eq,0,NEQS){
    sigm[eq] = 1 ;
    Unext[eq] = con[eq] + sigm[eq] * up[eq] ;
  }
  if( Unext[0] < rhomin && up[0] < 0 ){
    sigm[0] = MIN( 1, MAX( 0, (rhomin - con[0]) / up[0] ) ) ;
    Unext[0] = con[0] + sigm[0] * up[0] ;
    if( safety ) printf("limited density update node %u, %f\n",
                        node->links.node_number, sigm[0] );
  }
```

This next hack is to suppress stagnation point vortex for axisymmetric fluctuation splitting (hopefully).

```
  if( !solver && up[1] < 0 &&
      1 -axisym +axisym *node->geom.coord[1] < 10 *FLT_EPSILON){
    sigm[1] = MIN( 1, MAX( 0,  -con[1] /up[1]));
    Unext[1] = con[1] +sigm[1] *up[1];
    if( safety ) printf("limited velocity update node %u, %f\n",
```

```
                        node->links.node_number, sigm[1] );
  }

  fun = con[3]/con[0] - .5*(SQUARE(con[1]) + SQUARE(con[2]))/SQUARE(con[0]) ;
  funNext = Unext[3] / Unext[0] -
            .5 * ( SQUARE(Unext[1]) + SQUARE(Unext[2]) ) / SQUARE(Unext[0]) ;
  dfun = funNext - fun ;
  if( funNext < emin && dfun < 0 ){
    if( up[0] < 0 ){
      sigm[0]  = 0 ;
      Unext[0] = con[0] ;
      if( safety ) printf("supressed density update node %u\n",
                          node->links.node_number );
    }
    FOR(eq,1,3){
      if( SQUARE( Unext[eq] ) > SQUARE( con[eq] ) ){
        sigm[eq]  = 0 ;
        Unext[eq] = con[eq] ;
/*      if( safety ) printf("halted mom-%d update node %u\n", */
/*                          eq, node->links.node_number ); */
      }
    }
  }
  funNext = Unext[3] / Unext[0] -
           .5 * (SQUARE(Unext[1]) + SQUARE(Unext[2])) / SQUARE(Unext[0]) ;
  dfun = funNext - fun ;
  if( funNext < emin && dfun < 0 && up[3] < 0 ){
    sigm[3] = Unext[0] * emin - con[3] +
              .5 * (SQUARE(Unext[1]) + SQUARE(Unext[2])) / Unext[0] ;
    sigm[3] /= up[3] ;
    sigm[3] = MIN( 1, MAX( 0, sigm[3] ) );
    Unext[3] = con[3] + sigm[3] * up[3] ;
    if( safety ) printf("limited energy update node %u, %f\n",
                        node->links.node_number, sigm[3] );
  }
```

Finally, update the conserved variables.

```
  FOR(eq,0,NEQS) con[eq] += ( sigm[eq] * CFL * up[eq] );
/*FOR(eq,0,NEQS) node->vars.conserved[eq] += sigm[eq]*CFL*node->update.dU[eq];*/
  return (float) ddot( NEQS, node->update.dU, 1, node->update.dU, 1 );
}
```

3.1 Euler Explicit Time Step

Compute an explicit time step as generalization of 1-D CFL constraint. For inviscid loop over edges connected to this node and find the most restrictive constraint using the edge

length and the projected wavespeed along the edge. Also check the cell midpoint to handle pancake cells. For boundary cells, also check without projecting the velocity for the inviscid timestep, as a hack fix.

$$\tau = \min\left(\frac{\vec{r}_{01}\cdot\vec{r}_{01}}{|\vec{V}_0\cdot\vec{r}_{01}| + a_0\|\vec{r}_{01}\|}\right) \qquad \text{(inviscid)}$$

Viscous timestep:

$$\tau = \frac{4SR_e\rho R}{\mu(R+\gamma-1)\sum_T \frac{\ell_i^2}{S_T}} \qquad \text{(viscous)}$$

```
double dt_Euler_exp( Node *node )
{
  double t, tv=0, a, mu;
  Node2Cell *celln;
  int n, nd;
  Node *cn, *cnp, *cnm;
  Coord r;
  double *Vel;    /*  pointer to the primitive velocity vector  */

  conserved_to_primitive( node->vars.conserved, node->vars.primitive );
  Vel = &node->vars.primitive[1] ;
  a = a_from_rho_P( node->vars.primitive[0], node->vars.primitive[3] );
  t = 100 * sqrt( node->geom.dual_area ) / a ; /* start off with a big value */

  CELL_LOOP_AT_NODE( node ){
    FOR(n,0,3){
      cn = (celln->cell)->geom.connect.nodes[n] ;
      if( cn != node ){
FOR(nd,0,NDIMS) r[nd] = cn->geom.coord[nd] - node->geom.coord[nd] ;
t = MIN( t, VEL2(r) /( fabs( ddot(NDIMS,Vel,1,r,1) ) + a * VEL(r) ) );
      }
    }
    FOR(nd,0,NDIMS) r[nd] = celln->cell->geom.metric.split_point[nd] -
                            node->geom.coord[nd] ;
    t = MIN( t, 2*VEL2(r) / ( fabs( ddot(NDIMS,Vel,1,r,1) ) + a * VEL(r) ) );
    if( node->bc ) t = MIN( t, 2 * VEL(r) / ( VEL(Vel) + a ) );
  }
  if( axisym ) t *= MIN( 2,
                         MAX( node->geom.coord[1], sqrt( node->geom.dual_area ) ) );

  if( !viscous ) tv = t;
  else{
    CELL_LOOP_AT_NODE( node ) FOR(n,0,3)
      if( celln->cell->geom.connect.nodes[n] == node ){
        cnp = celln->cell->geom.connect.nodes[cyclic_plus0(n)];
        cnm = celln->cell->geom.connect.nodes[cyclic_minus0(n)];
        FOR(nd,0,NDIMS) r[nd] = cnp->geom.coord[nd] - cnm->geom.coord[nd];
        tv += VEL2(r) / celln->cell->geom.metric.area;
```

```
    }
    mu=mu_from_T(T_from_rho_P(node->vars.primitive[0],node->vars.primitive[3]));
    tv *= mu * ( GAMM1 + R_air * nondim_T / SQUARE( nondim_V ) );
    tv = 4 * node->geom.dual_area * Reynolds * node->vars.conserved[0] *
      R_air*nondim_T/SQUARE(nondim_V) / tv;
  }

  node->update.dt = MIN(t, CFL_v*tv) /4;/* safety factor that seems to help */
  return node->update.dt;
}
```

3.2 Time Accuracy

Compute local time step restrictions and return the most restrictive value.

```
double sync_time(void){
  double t = 1 / FLT_EPSILON;
  Node *node;

  NODE_LOOP_FORWARD t = MIN( t, dt_Euler_exp(node) );
  return t;
}
```


4 Characteristic Alignment Adaption

Perform local anisotropic adaption, similar to the Habashi style, based on fluctuation minimization to produce characteristic alignment.

```
#include <math.h>       /*  fabs                      */
#include <stdlib.h>     /*  exit                      */
#include "basics.h"
#include "adaption.h"
#include "blas.h"       /*  dnrm2                     */
#include "rhs.h"        /*  cell_fluc, full_viscous   */
#include "TriMath.h"    /*  Triave, cyclic+-          */
#include "transform.h"  /*  dW_dZ                     */
```

4.1 Edge fluctuation sum

Sum the fluctuations from the cells to either side of an edge as an analogy to the edge error estimate.

```
double error_fluc( Cell *cell_left, Cell *cell_right ){
  double edge_fluctuation;
  SolVec fluctuation;
  double pia;

  if(axisym) pia = Triave(cell_left->geom.connect.nodes[0]->geom.coord[1],
                          cell_left->geom.connect.nodes[1]->geom.coord[1],
                          cell_left->geom.connect.nodes[2]->geom.coord[1]);
  else pia = 1;
  pia *= sqrt(cell_left->geom.metric.area);
  cell_fluc(cell_left->geom.connect.nodes, fluctuation);
  edge_fluctuation = dnrm2(NEQS, fluctuation, 1) /pia;
  if(viscous){
    viscous_fluc(cell_left, fluctuation);
    edge_fluctuation += dnrm2(NEQS, fluctuation, 1) /pia;
  }

  if(cell_right){
    if(axisym) pia = Triave(cell_right->geom.connect.nodes[0]->geom.coord[1],
                            cell_right->geom.connect.nodes[1]->geom.coord[1],
                            cell_right->geom.connect.nodes[2]->geom.coord[1]);
    else pia = 1;
    pia *= sqrt(cell_right->geom.metric.area);
    cell_fluc(cell_right->geom.connect.nodes, fluctuation);
    edge_fluctuation += dnrm2(NEQS, fluctuation, 1) /pia;
    if( viscous ){
      viscous_fluc(cell_right, fluctuation);
      edge_fluctuation += dnrm2(NEQS, fluctuation, 1) /pia;
```

```
    }
  }
  else edge_fluctuation *= 2;

  return edge_fluctuation;
}
```

4.2 Viscous cell fluctuation wrapper

Return a sum of the absolute values of the viscous fluctuations distributed to each node of the triangle.

```
void viscous_fluc( Cell *cell, SolVec fluctuation ){
  int eq, n;
  double distrib[NEQS][3];

  full_viscous( cell, distrib );
  FOR(eq,0,NEQS){
    fluctuation[eq] = 0;
    FOR(n,0,3) fluctuation[eq] += fabs( distrib[eq][n] );
  }
}
```

4.3 Move a node for fluctuation minimization

Follow the recipie from the dissertation.

```
void FM_movement( Node *node, Coord R ){
  typedef double Jacobian[NEQS][NEQS];
  Jacobian alphap, dalphadx, dalphady, betap, dbetadx, dbetady;
  Jacobian WZ, AminusB, Lower, Upper;
  int i, j, k;
  Node2Cell *celln;
  Node *node1=NULL, *node2=NULL, *node3=NULL;
  Coord nhat1, nhat2, nhat3;
  double l1, l2, l3, valpha, vbeta, vgamma, sum;
  double u, v, a2;
  SolVec xi_z, eta_z, xi_w, eta_w, AplusBx, AplusBy;
  SolVec dZdx, dZdy, y_is_Ux;
  SolVec phi, dphidx, dphidy;
  SolVec Z1, Z2, Z3, W1, W2, W3, xieta_z, xieta_w;
  SolVec AxW, AyW, Wzdzdx, Wzdzdy;
  Coord dvdx, dvdy;
  Jacobian gammap;

  FOR(i,0,NEQS){
```



```
    FOR(j,0,NEQS) AminusB[i][j] = Lower[i][j] = Upper[i][j] = 0;
    AplusBx[i] = AplusBy[i] = 0;
    Lower[i][i] = 1;
  }

  CELL_LOOP_AT_NODE( node ){
    FOR(i,0,NEQS) FOR(j,0,NEQS){
      alphap[i][j] = dalphadx[i][j] = dalphady[i][j] = 0;
      betap[i][j]  = dbetadx[i][j]  = dbetady[i][j]  = 0;
    }

    FOR(i,0,3) if( celln->cell->geom.connect.nodes[i] == node ){
      node1 = celln->cell->geom.connect.nodes[cyclic_minus0(i)];
      node2 = node;
      node3 = celln->cell->geom.connect.nodes[cyclic_plus0(i)];
    }
    nhat1[0] = node3->geom.coord[1] - node2->geom.coord[1];
    nhat1[1] = node2->geom.coord[0] - node3->geom.coord[0];
    nhat3[0] = node2->geom.coord[1] - node1->geom.coord[1];
    nhat3[1] = node1->geom.coord[0] - node2->geom.coord[0];
    l1 = dnrm2(NDIMS,nhat1,1);
    l3 = dnrm2(NDIMS,nhat3,1);
    FOR(i,0,NDIMS){
      nhat1[i] /= l1;
      nhat3[i] /= l3;
    }

    u = celln->cell->tilde.primitive[1];
    v = celln->cell->tilde.primitive[2];
    a2 = SQUARE(celln->cell->tilde.a);

    if( (valpha = u *nhat1[0] +v *nhat1[1]) > 0 ){
      FOR(i,0,NEQS){
        alphap[i][i] = l1 *valpha /2;
        dalphadx[i][i] =  v /2;
        dalphady[i][i] = -u /2;
      }
      alphap[3][1] = a2 *(alphap[1][3] = l1 *nhat1[0] /2);
      alphap[3][2] = a2 *(alphap[2][3] = l1 *nhat1[1] /2);
      dalphadx[3][2] = a2 *(dalphadx[2][3] =  1 /2);
      dalphady[3][1] = a2 *(dalphady[1][3] = -1 /2);
    }
    if( (vbeta = u *nhat3[0] +v *nhat3[1]) > 0 ){
      FOR(i,0,NEQS){
        betap[i][i] = -l3 *vbeta /2;
        dbetadx[i][i] =  v /2;
        dbetady[i][i] = -u /2;
      }
      betap[3][1] = a2 *(betap[1][3] = -l3 *nhat3[0] /2);
      betap[3][2] = a2 *(betap[2][3] = -l3 *nhat3[1] /2);
```

```
    dbetadx[3][2] = a2 *(dbetadx[2][3] =  1 /2);
    dbetady[3][1] = a2 *(dbetady[1][3] = -1 /2);
  }
  FOR(i,0,NEQS) FOR(j,0,NEQS) alphap[i][j] -= betap[i][j];

  dW_dZ(celln->cell->tilde.parameter, WZ);
  FOR(i,0,NEQS) FOR(j,0,NEQS){
    sum = 0;
    FOR(k,0,NEQS) sum += alphap[i][k] *WZ[k][j];
    AminusB[i][j] += sum;
  }

  FOR(i,0,NEQS){
    xi_z[i]  = node2->vars.parameter[i] - node1->vars.parameter[i];
    eta_z[i] = node3->vars.parameter[i] - node2->vars.parameter[i];
  }
  FOR(i,0,NEQS){
    xi_w[i]  = ddot(NEQS, WZ[i], 1, xi_z, 1);
    eta_w[i] = ddot(NEQS, WZ[i], 1, eta_z, 1);
  }
  FOR(i,0,NEQS){
    AplusBx[i] -= ddot(NEQS, dalphadx[i], 1, xi_w, 1)
               -  ddot(NEQS, dbetadx[i], 1, eta_w, 1);
    AplusBy[i] -= ddot(NEQS, dalphady[i], 1, xi_w, 1)
               -  ddot(NEQS, dbetady[i], 1, eta_w, 1);
  }
}        /*  end of cell loop  */
```

Perform LU decomposition of AminusB matrix.

```
FOR(i,0,NEQS){
  for(sum=0, k=0; k<i; ++k) sum += Lower[i][k] *Upper[k][i];
  if((Upper[i][i] = AminusB[i][i] -sum) == 0 ){
    puts("Zero pivot in LU decomposition, exiting");
    exit(1);
  }
  FOR(j,i+1,NEQS){
    for(sum=0, k=0; k<i; ++k) sum += Lower[i][k] *Upper[k][j];
    Upper[i][j] = AminusB[i][j] -sum;
    for(sum=0, k=0; k<i; ++k) sum += Lower[j][k] *Upper[k][i];
    Lower[j][i] = (AminusB[j][i] -sum) /Upper[i][i];
  }
}
```

Now solve for $\frac{\partial Z}{\partial x}$ and $\frac{\partial Z}{\partial y}$ using forward and backward substitution.

```
FOR(i,0,NEQS){
```



```
  for( sum=0, j=0; j<i-1; ++j ) sum += Lower[i][j] *y_is_Ux[j];
  y_is_Ux[i] = AplusBx[i] - sum;
}
for( i=NEQS-1; i>=0; --i ){
  for( sum=0, j=i+1; j<NEQS; ++j ) sum += Upper[i][j] *dZdx[j];
  dZdx[i] = (y_is_Ux[i] -sum) /Upper[i][i];
}
FOR(i,0,NEQS){
  for( sum=0, j=0; j<i-1; ++j ) sum += Lower[i][j] *y_is_Ux[j];
  y_is_Ux[i] = AplusBy[i] - sum;
}
for( i=NEQS-1; i>=0; --i ){
  for( sum=0, j=i+1; j<NEQS; ++j ) sum += Upper[i][j] *dZdy[j];
  dZdy[i] = (y_is_Ux[i] -sum) /Upper[i][i];
}
```

Perform another cell loop to form ϕ and the derivatives of ϕ.

```
CELL_LOOP_AT_NODE( node ){
  FOR(i,0,NEQS) phi[i] = dphidx[i] = dphidy[i] = 0;
  FOR(i,0,NEQS) FOR(j,0,NEQS) gammap[i][j] = 0;

  FOR(i,0,3) if( celln->cell->geom.connect.nodes[i] == node ){
    node1 = celln->cell->geom.connect.nodes[cyclic_minus0(i)];
    node2 = node;
    node3 = celln->cell->geom.connect.nodes[cyclic_plus0(i)];
  }

  nhat1[0] = node3->geom.coord[1] - node2->geom.coord[1];
  nhat1[1] = node2->geom.coord[0] - node3->geom.coord[0];
  nhat2[0] = node1->geom.coord[1] - node3->geom.coord[1];
  nhat2[1] = node3->geom.coord[0] - node1->geom.coord[0];
  nhat3[0] = node2->geom.coord[1] - node1->geom.coord[1];
  nhat3[1] = node1->geom.coord[0] - node2->geom.coord[0];
  l1 = dnrm2(NDIMS,nhat1,1);
  l2 = dnrm2(NDIMS,nhat2,1);
  l3 = dnrm2(NDIMS,nhat3,1);
  FOR(i,0,NDIMS){
    nhat1[i] /= l1;
    nhat2[i] /= l2;
    nhat3[i] /= l3;
  }

  u = celln->cell->tilde.primitive[1];
  v = celln->cell->tilde.primitive[2];
  a2 = SQUARE(celln->cell->tilde.a);

  dW_dZ(celln->cell->tilde.parameter, WZ);
  FOR(i,0,NEQS){
```

```
  xi_z[i]  = node2->vars.parameter[i] - node1->vars.parameter[i];
  eta_z[i] = node3->vars.parameter[i] - node2->vars.parameter[i];
  xieta_z[i] = node3->vars.parameter[i] - node1->vars.parameter[i];
  Z1[i] = node1->vars.parameter[i];
  Z2[i] = node2->vars.parameter[i];
  Z3[i] = node3->vars.parameter[i];
}
FOR(i,0,NEQS){
  xi_w[i]  = ddot(NEQS, WZ[i], 1,  xi_z, 1);
  eta_w[i] = ddot(NEQS, WZ[i], 1, eta_z, 1);
  xieta_w[i] = ddot(NEQS, WZ[i], 1, xieta_z, 1);
  W1[i] = ddot(NEQS, WZ[i],1, Z1,1);
  W2[i] = ddot(NEQS, WZ[i],1, Z2,1);
  W3[i] = ddot(NEQS, WZ[i],1, Z3,1);
}
valpha = u *nhat1[0] +v *nhat1[1];
vgamma = u *nhat2[0] +v *nhat2[1];
vbeta  = u *nhat3[0] +v *nhat3[1];
FOR(i,0,NEQS){
  alphap[i][i] =  l1 *valpha /2;
  betap[i][i]  = -l3 *vbeta  /2;
  gammap[i][i] =  l2 *vgamma;
}
alphap[3][1] = a2 *(alphap[1][3] =  l1 *nhat1[0] /2);
alphap[3][2] = a2 *(alphap[2][3] =  l1 *nhat1[1] /2);
betap[3][1]  = a2 *(betap[1][3]  = -l3 *nhat3[0] /2);
betap[3][2]  = a2 *(betap[2][3]  = -l3 *nhat3[1] /2);
gammap[3][1] = a2 *(gammap[1][3] =  l2 *nhat2[0]);
gammap[3][2] = a2 *(gammap[2][3] =  l2 *nhat2[1]);

FOR(i,0,NEQS)
  phi[i] = -ddot(NEQS,alphap[i],1,xi_w,1) -ddot(NEQS,betap[i],1,eta_w,1);

FOR(i,0,NEQS) AyW[i] = AxW[i] = 0;
AxW[0] = u *xieta_w[0];
AxW[1] = u *xieta_w[1] + xieta_w[3];
AxW[2] = u *xieta_w[2];
AxW[3] = u *xieta_w[3] + a2 *xieta_w[1];
AyW[0] = v *xieta_w[0];
AyW[1] = v *xieta_w[1];
AyW[2] = v *xieta_w[2] + xieta_w[3];
AyW[3] = v *xieta_w[3] + a2 *xieta_w[2];

dvdx[0] = (dZdx[1] -Z2[1] *dZdx[0] /Z2[0]) /Z2[0];
dvdx[1] = (dZdx[2] -Z2[2] *dZdx[0] /Z2[0]) /Z2[0];
dvdy[0] = (dZdy[1] -Z2[1] *dZdy[0] /Z2[0]) /Z2[0];
dvdy[1] = (dZdy[2] -Z2[2] *dZdy[0] /Z2[0]) /Z2[0];

FOR(i,0,NEQS){
  Wzdzdx[i] = ddot(NEQS, WZ[i],1, dZdx,1);
```



```
  Wzdzdy[i] = ddot(NEQS, WZ[i],1, dZdy,1);
}

FOR(i,0,NEQS){
  dphidx[i] = 0.5 *(-AyW[i] +(l1 *ddot(NDIMS,nhat1,1,dvdx,1) *W1[i] +
                             l2 *ddot(NDIMS,nhat2,1,dvdx,1) *W2[i] +
                             l3 *ddot(NDIMS,nhat3,1,dvdx,1) *W3[i]) /3
                     +ddot(NEQS, gammap[i],1, Wzdzdx,1));
  dphidy[i] = 0.5 *( AxW[i] +(l1 *ddot(NDIMS,nhat1,1,dvdy,1) *W1[i] +
                             l2 *ddot(NDIMS,nhat2,1,dvdy,1) *W2[i] +
                             l3 *ddot(NDIMS,nhat3,1,dvdy,1) *W3[i]) /3
                      +ddot(NEQS, gammap[i],1, Wzdzdy,1));
}
}     /*  end of second cell loop  */
```

Hardwire for identity weighting matrix.

```
  R[0] = -ddot(NEQS, dphidx,1, phi,1);
  R[1] = -ddot(NEQS, dphidy,1, phi,1);
}
```

5 Viscous Evaluation

Viscous interior and boundary routines are included here. Viscous axisymmetric source terms are computed separately in GenSource.

```
#include <float.h>    /*  DBL_EPSILON, FLT_EPSILON              */
#include <math.h>     /*  pow, sqrt, fabs                       */
#include "basics.h"
#include "TriMath.h"  /*  cyclic_plus0                          */
#include "blas.h"     /*  dnrm2                                 */
#include "therm.h"    /*  conductivity, T(rho,P), mu(T), e(T)   */
#include "rhs.h"      /*  prototypes for this file              */
#include "lhs.h"      /*  dt_Euler_exp                          */
#include "transform.h"/*  UtoZ, gradZ_to_gradV                  */
```

5.1 Haselbacher Thin-Layer

Evaluate the viscous terms using Haselbacher[HMP99] edge-based grid-transparent thin-layer method. Basic assumptions are: a) neglect tangential derivatives, b) the velocity divergence is written as the normal derivative dot the unit normal, c) normal derivatives are taken to be the edge derivatives.

$$\int_\Omega \varpi_a \vec{F}^v \cdot \hat{n}\, d\omega \simeq \sum_\Gamma \frac{\varpi_a \mu\, \Delta\Gamma}{R_e} \left\{ \begin{array}{c} 0 \\ \vec{V}_n^T + \frac{1}{3}\hat{n}^T \left(\vec{V}_n \cdot \hat{n} + \varpi \frac{v}{y} \right) \\ \frac{\kappa}{\mu} T_n + \vec{V} \cdot \vec{V}_n + \frac{1}{3}(\vec{V} \cdot \hat{n}) \left(\vec{V}_n \cdot \hat{n} + \varpi \frac{v}{y} \right) \end{array} \right\}$$

$$\Delta\Gamma = \text{length split-point minus edge mid-point}$$

$$\hat{n} = \text{unit normal along edge}$$

$$T_n = \frac{\Delta T}{\ell}, \quad etc.$$

```
void Haselbacher( Cell *cell, double distrib[NEQS][3] ){
  int cnode, i ;              /*  dummy indicies          */
  Coord nhat ;                /*  unit normal along edge  */
  Coord Vvel ;                /*  velocity vector         */
  Coord Vn ;                  /*  velocity derivative     */
  Coord midpoint ;
  double dGam ;               /*  face length             */
  Node *na, *nb ;             /*  edge end nodes          */
  double elen ;               /*  edge length             */
  double mu, pia, f1, fact ;
  double Tn ;                 /*  temperature derivative  */

  FOR(i,0,NEQS) FOR(cnode,0,3) distrib[i][cnode] = 0 ;
```



```
  FOR(cnode,0,3){                        /*  loop on edges of the triangle  */
    na = cell->geom.connect.nodes[cnode] ;
    nb = cell->geom.connect.nodes[cyclic_plus0(cnode)] ;
    FOR(i,0,NDIMS) nhat[i] = nb->geom.coord[i] - na->geom.coord[i] ;
    elen = dnrm2( NDIMS, nhat, 1 );
    FOR(i,0,NDIMS) nhat[i] /= elen ;
    FOR(i,0,NDIMS) midpoint[i] = ( na->geom.coord[i] + nb->geom.coord[i] ) / 2 ;
    pia = (! axisym) ? 1
        : 0.5 *(midpoint[1] +cell->geom.metric.split_containment[1]);
    FOR(i,0,NDIMS) midpoint[i] -= cell->geom.metric.split_containment[i] ;
    dGam = dnrm2( NDIMS, midpoint, 1 );

    mu = ( mu_from_T(T_from_rho_P(na->vars.primitive[0], na->vars.primitive[3]))
         + mu_from_T(T_from_rho_P(nb->vars.primitive[0], nb->vars.primitive[3]))
         ) / 2 ;
    FOR(i,0,NDIMS)
      Vvel[i]= ( na->vars.primitive[i+1] + nb->vars.primitive[i+1] ) / 2 ;
    Tn = ( T_from_rho_P(nb->vars.primitive[0], nb->vars.primitive[3])
         - T_from_rho_P(na->vars.primitive[0], na->vars.primitive[3]) ) / elen ;
    FOR(i,0,NDIMS)
      Vn[i] = ( nb->vars.primitive[i+1] - na->vars.primitive[i+1] ) / elen ;

    f1 = ( ddot( NDIMS, Vn, 1, nhat, 1 ) + axisym * Vvel[1]
         / MAX( pia, 100 * DBL_EPSILON ) ) / 3 ;
    fact = ( Vn[0] + nhat[0] * f1 ) * pia * mu * dGam / Reynolds ;
    distrib[1][cnode] += fact ;
    distrib[1][cyclic_plus0(cnode)] -= fact ;

    fact = ( Vn[1] + nhat[1] * f1 ) * pia * mu * dGam / Reynolds ;
    distrib[2][cnode] += fact ;
    distrib[2][cyclic_plus0(cnode)] -= fact ;

    fact = ( Tn * conductivity( mu ) / mu + ddot( NDIMS, Vvel, 1, Vn, 1 )
           + ddot( NDIMS, Vvel, 1, nhat, 1 ) * f1 ) *pia *mu *dGam /Reynolds ;
    distrib[3][cnode] += fact ;
    distrib[3][cyclic_plus0(cnode)] -= fact ;
  }
}
```

5.2 Haselbacher Boundary

Apply the approximate Haselbacher viscous treatment at a solid-wall boundary. The velocities are zero, as is the projection of the velocity gradient on the outward (to the control volume, inward to the physical wall) unit normal. Weak enforcement is achieved by specifying a heat flux that will drive the temperature to the desired wall temperature and a surface shear that will null the momentum.

```
void Hasel_bc( Bound_edge *bedge ){
```

```
  double pia, dGam, Dt;
  int in, ie, i;
  Node *n;
  Node2Cell *celln;
  int cln;
/* Node *otherNode; */
  const double minimum = 100 * FLT_EPSILON;
  double safetyFactor = 1;

  dGam = sqrt( pow( (bedge->node[0])->geom.coord[0] -
                (bedge->node[1])->geom.coord[0], 2 ) +
                pow( (bedge->node[0])->geom.coord[1] -
                (bedge->node[1])->geom.coord[1], 2 ) ) / 2 ;/*half edge length*/
```

Loop on each end node of the edge.

```
  FOR(in,0,2){
    n = bedge->node[in] ;
    Dt = dt_Euler_exp( n );
    cln = 1;
```

Set the axisymmetric parameter.

If the node is on the axisymmetric axis, use the y-value at the one-eighth point of the edge.

```
/*     if( !axisym ) pia = 1; */
/*     else{ */
/*       pia = n->geom.coord[1]; */
/*       if( pia < minimum ){ */
/*         pia = (bedge->node[0]->geom.coord[1] + bedge->node[1]->geom.coord[1])/8; */
/*         pia = MAX( pia, minimum ); */
/*       } */
/*     } */
```

Copy pia definition from the Bronowski update.

```
  /*  For axisymmetric, use mass-lumped integration point, except for */
  /*  on the axis, where the median-dual centroid is used              */
  if( (pia = 1 - axisym + axisym * n->geom.coord[1]) < minimum ){
    CELL_LOOP_AT_NODE( n ){
      FOR(i,0,3)
        pia += ( ((celln->cell)->geom.connect.nodes[i])->geom.coord[1] ) /
               ( ((celln->cell)->geom.connect.nodes[i])->bc ? 2 : 4 ) ;
      ++cln ;
    }
    pia /= cln ;
```



```
  }
```

Assign the target wall temperature and then restrain how far the current wall temperature is allowed to change this iteration.

```
    n->vars.wall[0] += MIN( .01, MAX( -.01, T_wall - n->vars.wall[0] ) );
```

There is no distribution for the density equation. For the energy equation, impose a heat flux that will drive the wall temperature to the desired point.

$$q_w = \frac{1}{\varpi_a \sum \Delta\Gamma} \left[\frac{\varpi_a S}{\Delta t} (U_3 - U_0 e(T_w)) + allRHS \right]$$

```
/*  try a safety factor at the stagnation point  */
/*      if( axisym && n->geom.coord[1] < minimum ){ */
/*        safetyFactor = 1.0; */
/*        n->update.viscous[3] *= 1.; */
/*        FOR(i,0,NEQS) n->update.source[i] = 0; */
/*        if(!solver) FOR(i,0,NEQS) n->update.source_fs[i] = 0; */
/*      } */
    n->vars.wall[3] =  ( pia * n->geom.dual_area *
                       ( n->vars.conserved[3] - n->vars.conserved[0]
                       * e_from_T( n->vars.wall[0] ) ) / Dt
                       + n->update.flux[3] + n->update.art_diss[3]
                       + n->update.viscous[3]
                       + ( !axisym ? 0 : ( n->update.source[3]
                       + solver ? 0 : n->update.source_fs[3] ) ) )
                       / ( pia * n->geom.bound_face )
                       * safetyFactor;
```

Put in a hack fix for an axisymmetric stagnation point to set the heating equal to the node off the axis.

```
/*     if( n == bedge->node[0] ) otherNode = bedge->node[1]; */
/*     else otherNode = bedge->node[0]; */
/*     if( n->geom.coord[1] < minimum && otherNode->geom.coord[1] > minimum ) */
/*       n->vars.wall[3] = otherNode->vars.wall[3]; */

    n->update.wall[3] -= pia * dGam * n->vars.wall[3] ;
```

Finally apply a surface shear to null the momentum. With any luck, this surface shear vector will be parallel to the surface, though I make no effort to impose this nor provide any check that it is so.

$$\vec{\tau}_w = \frac{1}{\varpi_a \sum \Delta\Gamma} \left[\frac{\varpi_a S}{\Delta t} (U_1, U_2) + allRHS \right]$$

```
    FOR(ie,1,3){
      n->vars.wall[ie] =  ( pia * n->geom.dual_area * n->vars.conserved[ie]
                          / Dt
                          + n->update.flux[ie] + n->update.art_diss[ie]
                          + n->update.viscous[ie]
                          + ( !axisym ? 0 : ( n->update.source[ie]
                          + solver ? 0 : n->update.source_fs[ie] ) ) )
                          / ( pia * n->geom.bound_face );
      n->update.wall[ie] -= pia * dGam * n->vars.wall[ie] ;
    }
  }  /*  loop on edge nodes  */
}
```

5.3 Full Navier-Stokes fluctuation

Compute the viscous fluctuation on a triangular cell for the full Navier-Stokes terms. The net viscous fluctuation is zero, since the viscous terms only redistribute amongst the nodes. The viscous fluctuation to the nodes, for the interior only (boundary contribution sums to zero for internal edges and is treated separately for boundary edges) is,

```
void full_viscous( Cell *cell, double distrib[NEQS][3] ){
  double pia[3], pia_bar;
  double Temp, mu, cond;
  double normal[3][NDIMS];                        /*  scaled outward normal  */
  double F[NEQS][NDIMS];                                   /*  viscous flux  */
  double grad_primitive[NDIMS][NEQS], grad_parameter[NDIMS][NEQS];
  double tau[NDIMS][NDIMS];
  double diverg, sum;
  double params[3][NEQS];
  Coord grad_T;
  double nCoord[3][NDIMS];
  int i, eq;
  Node **cNodes=cell->geom.connect.nodes;

  FOR(i,0,3){
    FOR(eq,0,NDIMS) nCoord[i][eq] = cNodes[i]->geom.coord[eq];
    pia[i] = !axisym ? 1 : nCoord[i][1];
    conserved_to_parameter( cNodes[i]->vars.conserved, params[i] );
    FOR(eq,0,NEQS) params[i][eq] = cNodes[i]->vars.parameter[eq];
  }
  FOR(i,0,3){
    normal[i][0] = nCoord[cyclic_minus0(i)][1] - nCoord[cyclic_plus0(i)][1];
    normal[i][1] = nCoord[cyclic_plus0(i)][0] - nCoord[cyclic_minus0(i)][0];
  }
  pia_bar = TriaveH( pia );
  Roe_average_cell( cell );
  Temp = T_from_rho_P( cell->tilde.primitive[0], cell->tilde.primitive[3] );
```



```
  mu = mu_from_T( Temp );
  cond = conductivity( mu );

  tri_grad( NEQS, params, nCoord, grad_parameter );
  FOR(i,0,NDIMS) FOR(eq,0,NEQS) grad_parameter[i][eq] /= cell->geom.metric.area;
  gradZ_to_gradV( grad_parameter, grad_primitive, cell->tilde.parameter );
```

The temperature gradient,

$$\nabla T = \frac{1}{R\rho^2}(\rho\nabla P - P\nabla\rho)$$

```
  FOR(i,0,NDIMS)
    grad_T[i] = ( cell->tilde.primitive[0] * grad_primitive[i][3] -
                  cell->tilde.primitive[3] * grad_primitive[i][0] ) *
                SQUARE( nondim_V ) / ( R_air * nondim_T *
                                       SQUARE( cell->tilde.primitive[0] ) );
```

The stress tensor,

$$\tau = \mu[\nabla^T\vec{V} + (\nabla^T\vec{V})^T - \frac{2}{3}\nabla\cdot\vec{V}]$$

```
  diverg = 2 * ( grad_primitive[0][1] + grad_primitive[1][2] ) / 3;
  tau[0][0] = mu * ( 2 * grad_primitive[0][1] - diverg );
  tau[0][1] = mu * ( grad_primitive[0][2] + grad_primitive[1][1] );
  tau[1][0] = mu * ( grad_primitive[0][2] + grad_primitive[1][1] );
  tau[1][1] = mu * ( 2 * grad_primitive[1][2] - diverg );
```

The viscous flux,

$$F^v = \frac{1}{R_e}\begin{pmatrix} 0 \\ \tau \\ \kappa\nabla T + \vec{V}\tau \end{pmatrix}$$

```
  FOR(i,0,NDIMS){
    F[0][i] = 0;
    F[1][i] = tau[0][i] / Reynolds;
    F[2][i] = tau[1][i] / Reynolds;
    F[3][i] = ( cond * grad_T[i] + cell->tilde.primitive[1] * tau[0][i]
              + cell->tilde.primitive[2] * tau[1][i] ) / Reynolds;
  }
```

Form the distributions,

$$\phi_i = \frac{\varpi_a \ell_i}{2}\tilde{F}^v\cdot\hat{n}_i$$

```
  FOR(i,0,3) FOR(eq,0,NEQS)
      distrib[eq][i] = 0.5 * pia_bar * ddot( NDIMS, F[eq], 1, normal[i], 1 );
```

```
  if( safety ) FOR(eq,0,NEQS){
    sum = 0.;
    FOR(i,0,3) sum += distrib[eq][i];
    if( fabs(sum) > 1e5 * FLT_EPSILON ){
      printf("Error, viscous fluctuations do not sum to zero:");
      printf(" Eqn=%d, sum=%e\n", eq, sum );
      Aetna_cell( cell );
    }
  }
}
```


6 Finite Volume Functions

Functions associated with a finite volume flux evaluation are grouped in this file.

```
#include <math.h>     /*  fabs, pow, cos, sin */
#include <float.h>    /*  DBL_EPSILON         */
#include <stdlib.h>   /*  free                */
#include "blas.h"     /*  dnrm2               */
#include <stdio.h>    /*  printf, puts        */
#include "rhs.h"
#include "basics.h"
#include "TriMath.h"  /*  cyclic_plus0        */
#include "limiter.h"  /*  limiter             */
#include "transform.h"/*  U->Z, etc           */
#include "bc.h"       /*  bc types            */
#include "therm.h"    /*  P(r,T), a(r,P)      */
```

6.1 Nodal Gradients

Compute nodal gradients using the formulation of Barth[Bar94]. This is for two dimensions only. The Barth scheme is an edge-based scheme. Here, I do loops over cells and then boundary edges, so that each edge is traversed twice, resulting in a factor of 2 duplication of effort.

Gradients are computed for the conserved variables. If other variables are desired, modify the function to take a pointer offset from the node root to point to the desired solution vector.

`reconstruct=0` performs reconstruction on conserved variables, while 1 reconstructs on primitive variables.

```
void node_gradients( int reconstruct )
{
  Node *node, *j1, *j2, **nodal_stack ;
  double *j1U, *j2U ;
  Cell *cell ;
  Bound_edge *bedge ;
  double det ;
  unsigned int j1num ;
  int in, eqn ;                    /*  dummy index                          */
                                   /*  these are the coefficients Barth uses  */
  double *l11, *l12, *l22;
  SolVec *lf1, *lf2;
  l11 = (double *) calloc( nodes, sizeof(double));
  l12 = (double *) calloc( nodes, sizeof(double));
  l22 = (double *) calloc( nodes, sizeof(double));
  lf1 = (SolVec *) malloc( nodes*sizeof(SolVec));
  lf2 = (SolVec *) malloc( nodes*sizeof(SolVec));

  nodal_stack = allocate_node_ptr( nodes+1 );          /*  number the nodes  */
```

```
stack_nodes( nodal_stack );
free( nodal_stack );

NODE_LOOP_FORWARD{                /*  zero out the summation coefficients  */
  j1num = node->links.node_number - 1 ;
  l11[j1num] = l12[j1num] = l22[j1num] = 0 ;
  FOR(eqn,0,NEQS) lf1[j1num][eqn] = lf2[j1num][eqn] = 0 ;
}

CELL_LOOP_FORWARD{                          /*  first loop over all cells  */
  FOR(in,0,3){
    j1 = cell->geom.connect.nodes[in] ;
    j2 = cell->geom.connect.nodes[cyclic_plus0(in)] ;
    if( !reconstruct ){
      j1U = j1->vars.conserved ;
      j2U = j2->vars.conserved ;
    }
    else{
      j1U = j1->vars.primitive ;
      j2U = j2->vars.primitive ;
    }
    barth_grad( j1, j2, j1U, j2U, l11, l12, l22, lf1, lf2 );
  }
}
BEDGE_LOOP_FORWARD{                     /*  then loop over boundary edges  */
  j1 = bedge->node[0] ;
  j2 = bedge->node[1] ;
  if( !reconstruct ){
    j1U = j1->vars.conserved ;
    j2U = j2->vars.conserved ;
  }
  else{
    j1U = j1->vars.primitive ;
    j2U = j2->vars.primitive ;
  }
  barth_grad( j1, j2, j1U, j2U, l11, l12, l22, lf1, lf2 );
}
```

Now the intermediary arrays have all been filled, so compute the gradients.

```
NODE_LOOP_FORWARD{
  j1num = node->links.node_number - 1 ;
  det = l11[j1num] * l22[j1num] - SQUARE(l12[j1num]) ;
  FOR(eqn,0,NEQS){
    node->fv.gradx[eqn] = ( l22[j1num] * lf1[j1num][eqn]
                          - l12[j1num] * lf2[j1num][eqn] ) / det ;
    node->fv.grady[eqn] = ( l11[j1num] * lf2[j1num][eqn]
                          - l12[j1num] * lf1[j1num][eqn] ) / det ;
  }
```



```
  }
 free(l11);
 free(l12);
 free(l22);
 free(lf1);
 free(lf2);
}
```

6.2 Barth Gradient Recipe

This is the recipie for distributing the gradient calculation along an edge. Send pointers to the endpoints of the edge, and the summation arrays are updated.

```
void barth_grad( Node *j1, Node *j2, SolVec j1U, SolVec j2U,
                 double l11[], double l12[], double l22[],
                 double lf1[][NEQS], double lf2[][NEQS])
{
  unsigned int j1num, j2num ;
  double dx, dy, du ;
  int eqn ;

  j1num = j1->links.node_number - 1 ;
  j2num = j2->links.node_number - 1 ;
  dx = j2->geom.coord[0] - j1->geom.coord[0] ;
  dy = j2->geom.coord[1] - j1->geom.coord[1] ;
  l11[j1num] += SQUARE(dx) ;
  l11[j2num] += SQUARE(dx) ;
  l12[j1num] += dx * dy ;
  l12[j2num] += dx * dy ;
  l22[j1num] += SQUARE(dy) ;
  l22[j2num] += SQUARE(dy) ;
  FOR(eqn,0,NEQS){
    du = j2U[eqn] - j1U[eqn] ;
    lf1[j1num][eqn] += dx * du ;
    lf1[j2num][eqn] += dx * du ;
    lf2[j1num][eqn] += dy * du ;
    lf2[j2num][eqn] += dy * du ;
  }
}
```

6.3 Nodal Gradient Limiter

Compute a limiter for the nodal gradient so that reconstruction can be performed as,

$$U = U_0 + \psi_0 \vec{r} \cdot (\nabla U)_0$$

The limiter is defined as,

$$\psi = \psi\left(\frac{\frac{U^{min/max}-U_0}{2}}{\vec{r}\cdot\nabla U}\right) = \frac{1}{\vec{r}\cdot\nabla U} M\left(\frac{U^{min/max}-U_0}{2}, \vec{r}\cdot\nabla U\right)$$

where M is the appropriate averaging function.

```
void fv_limit_node( int reconstruct, Node *node, SolVec psi )
{
  int in, eq, itmp ;    /*  dummy indicie                                  */
  Node2Cell *celln ;
  SolVec Umin, Umax, U0 ;
  Node *other_node ;
  int this_node=0;          /*  this node is node 0,1,2 of the current cell */
  Coord QP ;                /*  quadrature point on dual face               */
  Coord r ;                 /*  position vector to quadrature point         */
  double gr ;               /*  r * grad U                                  */
  double du ;               /*  ( U(max/min) - U0 ) / 2                     */
  double M ;                /*  averaging function associated with limiter  */
  double eps2 ;             /*  parameter for limiter                       */
  double en0, energy, ftemp ;        /*  for preventing negative energies   */
  const double emin = .3 * R_air * nondim_T / GAMM1 / SQUARE( nondim_V );

  FOR(eq,0,NEQS){
    Umin[eq] = Umax[eq] = U0[eq] =
      reconstruct ? node->vars.primitive[eq] : node->vars.conserved[eq] ;
    psi[eq] = 2 ;                                   /*  compressive limit  */
  }
```

Loop on cells connected to this node to find U^{min} and U^{max} from distance-one neighbors.

```
  CELL_LOOP_AT_NODE(node) FOR(in,0,3){
    other_node = (celln->cell)->geom.connect.nodes[in] ;
    FOR(eq,0,NEQS){
      if( reconstruct ){
        Umin[eq] = MIN( Umin[eq], other_node->vars.primitive[eq] );
        Umax[eq] = MAX( Umax[eq], other_node->vars.primitive[eq] );
      }
      else{
        Umin[eq] = MIN( Umin[eq], other_node->vars.conserved[eq] );
        Umax[eq] = MAX( Umax[eq], other_node->vars.conserved[eq] );
      }
    }
  }
```

Now loop on the cells connected to this node to identify the edges connected to the node. Once the edges have been found, then determine the quadrature points and $\vec{r}$ and $\vec{r}\cdot\nabla U$.

At that point we can evaluate the limiter, choosing the appropriate one of (U^{min}, U^{max}). The final limiter will be the most restrictive limiter imposed by each quadrature point.

Note that $|\vec{r}|^3$ is passed to limiter for the van Albada ϵ^2.

```
  CELL_LOOP_AT_NODE(node){
    FOR(in,0,3)                  /*  identify the current node in this cell  */
      if( (celln->cell)->geom.connect.nodes[in] == node ) this_node = in ;

    FOR(itmp,0,2){            /* do only the 2 edges connected to this node */
      other_node = itmp ?
                 (celln->cell)->geom.connect.nodes[cyclic_plus0(this_node)] :
                 (celln->cell)->geom.connect.nodes[cyclic_minus0(this_node)] ;

      quadrature_point( node->geom.coord, other_node->geom.coord,
                        (celln->cell)->geom.metric.split_point, QP );
      FOR(in,0,NDIMS) r[in] = QP[in] - node->geom.coord[in] ;

      eps2 = pow( dnrm2( 2, r, 1 ), 3) ;
      FOR(eq,0,NEQS){                                /*  loop over equations  */
        gr = r[0] * node->fv.gradx[eq] + r[1] * node->fv.grady[eq] ;
        du = ( ( Umax[eq] - U0[eq] ) * gr > 0 )
           ? (Umax[eq] - U0[eq]) / 2 : (Umin[eq] - U0[eq]) / 2 ;
        M = limiter( du, gr, eps2 );
        psi[eq] = MIN( psi[eq], fabs(M) / (fabs(gr) + DBL_EPSILON) );
      }
    }
  }
```

There is the danger of creating non-physical reconstruction states. Reuse the U_{min} variable to hold the reconstructed state. If any states are non-physical, reduce limiters for all equations by same factor at this node.

```
  CELL_LOOP_AT_NODE(node){
    FOR(in,0,3){                  /*  identify the current node in this cell  */
      if( (celln->cell)->geom.connect.nodes[in] == node ) this_node = in ;
    }

    FOR(itmp,0,2){            /* do only the 2 edges connected to this node */
      other_node = itmp ?
                 (celln->cell)->geom.connect.nodes[cyclic_plus0(this_node)] :
                 (celln->cell)->geom.connect.nodes[cyclic_minus0(this_node)] ;

      quadrature_point( node->geom.coord, other_node->geom.coord,
                        (celln->cell)->geom.metric.split_point, QP );
      FOR(in,0,NDIMS) r[in] = QP[in] - node->geom.coord[in] ;

      FOR(eq,0,NEQS){                                /*  loop over equations  */
        gr = r[0] * node->fv.gradx[eq] + r[1] * node->fv.grady[eq] ;
```

```
        Umin[eq] = U0[eq] + psi[eq] * gr ;
      }
      if( !reconstruct ){                                  /*  conserved  */
        en0 = U0[3] / U0[0] - .5*(SQUARE(U0[1]) + SQUARE(U0[2]))/SQUARE(U0[0]);
        energy = Umin[3] / Umin[0] -
                 .5 * ( SQUARE(Umin[1]) + SQUARE(Umin[2]) ) / SQUARE(Umin[0]) ;
      }
      else{                                                /*  primitive  */
        en0 = e_from_rho_P( U0[0], U0[3] );
        energy = e_from_rho_P( Umin[0], Umin[3] );
      }
      if( energy < emin && energy < en0 ){
        ftemp = MAX( 0, (en0 - emin) / (en0 - energy) );\
/*    if( safety ) printf("limiting reconstruction for node %u, ftemp=%f\n", */
/*                        node->links.node_number, ftemp ); */
        FOR(eq,0,NEQS) psi[eq] *= ftemp ;
      }
    }
  }
}
```

6.4 Quadrature Point

Compute the finite volume quadrature point on a dual face, given a physical edge and a split point to the left of the edge. The quadrature point is half-way between the split point and the mid-point of the edge$_{01}$.

Send the coordinates of n_0 and n_1, and the coordinates of the split point in the cell containing the control volume face. Return a pointer to the quadrature point vector.

```
void quadrature_point( Coord x0, Coord x1, Coord SP, Coord QP )
{
  int in ;
  FOR(in,0,NDIMS) QP[in] = ( 2 * SP[in] + x0[in] + x1[in] ) / 4 ;
}
```

6.5 Edge FV Central Difference

Evaluate the central-difference flux contribution from a face to the finite volume update.

$$\varpi_a S_i U_{i_t} \leftarrow -\frac{\vec{F}_{in} + \vec{F}_{out}}{2} \cdot \hat{n}\, \Delta\Gamma \varpi_a$$

where,

$$F_{in} = F(U_{in})$$

$$U_{in} = U_0 + \psi_0 \vec{r}_0 \cdot (\nabla U)_0$$

ϖ_a equals zero for 2D or the quadrature point for axisymmetric.

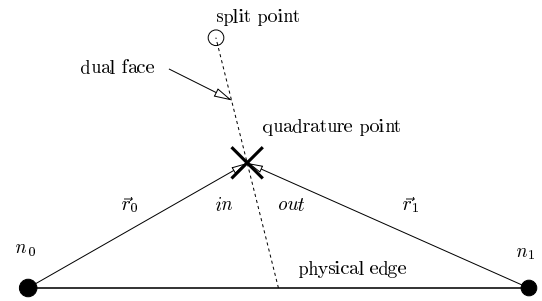


Figure 1: Finite volume face geometry.

Send the nodal values, the split point, a pointer to the flux function, and then fill in the distribution vectors. The face is defined to be from the midpoint of the edge to the split point.

```
void FV_edge_central( Coord x0, Coord x1, Coord SP,
                      SolVec u0, SolVec gx0, SolVec gy0, SolVec psi0,
                      SolVec u1, SolVec gx1, SolVec gy1, SolVec psi1,
                      SolVec dist0, SolVec dist1, void (*flux)() )
{
  Coord r, normal, QP ;
  double pia ;
  int i ;                    /*  dummy indicie  */
  SolVec Utmp, Vtmp, v0 ;
  double F[NEQS][NDIMS] ;    /*  flux           */
  enum { conserved, primitive } reconstruct = primitive ;
```

$$\varpi_a = 1 - \varpi + \varpi QP_y$$

$$\hat{n}\Delta\Gamma = 2(SP_y - QP_y, QP_x - SP_x)$$

```
  quadrature_point( x0, x1, SP, QP );
  pia = 1 - axisym + axisym * QP[1] ;
  normal[0] = 2 * ( SP[1] - QP[1] ) ;
  normal[1] = 2 * ( QP[0] - SP[0] ) ;
```

Compute the flux from the inside(0) and distribute.

```
  FOR(i,0,NDIMS) r[i] = QP[i] - x0[i] ;
  if( reconstruct == conserved ){
    FOR(i,0,NEQS) Utmp[i] = u0[i] + psi0[i] * (r[0] * gx0[i] + r[1] * gy0[i]) ;
  }
  else{
    conserved_to_primitive( u0, v0 );
    FOR(i,0,NEQS) Vtmp[i] = v0[i] + psi0[i] * (r[0] * gx0[i] + r[1] * gy0[i]) ;
    primitive_to_conserved( Vtmp, Utmp );
  }
  (*flux)( Utmp, F );
  FOR(i,0,NEQS) dist1[i] = (F[i][0] * normal[0] + F[i][1] * normal[1]) * pia/2;
```

Now compute the flux from the outside(1) and distribute.

```
  FOR(i,0,NDIMS) r[i] = QP[i] - x1[i] ;
  if( reconstruct == conserved ){
    FOR(i,0,NEQS) Utmp[i] = u1[i] + psi1[i] * (r[0] * gx1[i] + r[1] * gy1[i]) ;
  }
  else{
    conserved_to_primitive( u1, v0 );
    FOR(i,0,NEQS) Vtmp[i] = v0[i] + psi1[i] * (r[0] * gx1[i] + r[1] * gy1[i]) ;
    primitive_to_conserved( Vtmp, Utmp );
  }
  (*flux)( Utmp, F );
  FOR(i,0,NEQS){
    dist1[i] += ( F[i][0] * normal[0] + F[i][1] * normal[1] ) * pia / 2 ;
    dist0[i]  = - dist1[i] ;
  }
}
```

6.6 Finite Volume Cell Distribution

Compute the distributions to the node for a finite volume RHS evaluation. Loop over the three edges of a triangle and call the basic finite volume evaluations for an edge, distributing to the end nodes of the edge.

```
void FV_distrib_cell( Cell *cell )
{
  void (*flux)();        /*  pointer to the flux function  */
  int i, eq ;            /*  dummy indicie                 */
  Node *n0, *n1 ;        /*  node pointers                 */
  SolVec dist0, dist1 ;  /*  temp distribution vectors     */
/*    unsigned int ncheck ; */

  FOR(i,0,3){                            /*  loop on edges of the triangle  */
    n0 = cell->geom.connect.nodes[i] ;
    n1 = cell->geom.connect.nodes[cyclic_plus0(i)] ;
```


Central difference portion.

```
    if( ! scalar ) flux = &Euler_flux_conserved ;
    else flux = &scalar_flux ;

    FV_edge_central( n0->geom.coord, n1->geom.coord,
                     cell->geom.metric.split_point,
                     n0->vars.conserved,
                     n0->fv.gradx, n0->fv.grady, n0->fv.psi,
                     n1->vars.conserved,
                     n1->fv.gradx, n1->fv.grady, n1->fv.psi,
                     dist0, dist1, flux ) ;
    FOR(eq,0,NEQS){
      n0->update.flux[eq] += dist0[eq] ;
      n1->update.flux[eq] += dist1[eq] ;
    }
```

Artificial dissipation portion.

```
    FV_art_diss( n0->geom.coord, n1->geom.coord, cell->geom.metric.split_point,
                 n0->vars.conserved, n0->fv.gradx, n0->fv.grady, n0->fv.psi,
                 n1->vars.conserved, n1->fv.gradx, n1->fv.grady, n1->fv.psi,
                 dist0, dist1 ) ;
    FOR(eq,0,NEQS){
      n0->update.art_diss[eq] += dist0[eq] ;
      n1->update.art_diss[eq] += dist1[eq] ;
    }

/*    ncheck = n0->links.node_number ; */
/*    if( ncheck == 878 ){ */
/*      printf("edge with nodes %u, %u\n", n0->links.node_number, */
/*          n1->links.node_number ); */
/*      Aetna_node( ncheck, "interior update on a cell"); */
/*    } */
/*    ncheck = n1->links.node_number ; */
/*    if( ncheck == 878 ){ */
/*      printf("edge with nodes %u, %u\n", n0->links.node_number, */
/*          n1->links.node_number ); */
/*      Aetna_node( ncheck, "interior update on a cell"); */
/*    } */

  }                                  /*  end loop on edges of the triangle  */
}
```

6.7 Finite Volume Artificial Dissipation

Evaluate the artificial dissipation distribution along an edge for the upwind Roe scheme. The distribution along an edge is,

$$S_0 U_{0_t} \leftarrow \varpi_a \Phi \Delta\Gamma \rightarrow -S_1 U_{1_t}$$

ϖ_a is 1 for 2-D and the y value of the quadrature point for axisymmetric. The dissipation is,

$$\Phi = \frac{1}{2}|\tilde{A} \cdot \hat{n}|(U_{out} - U_{in})$$

```
void FV_art_diss( Coord x0, Coord x1, Coord SP,
                  SolVec u0, SolVec gx0, SolVec gy0, SolVec psi0,
                  SolVec u1, SolVec gx1, SolVec gy1, SolVec psi1,
                  SolVec dist0, SolVec dist1 )
{
  Coord r, normal, QP ;
  double pia, Htilde, atilde, veltilde, vell, velr, al, ar ;
  SolVec Ul, Vl, Zl, Ur, Vr, Zr, Vtilde, xdu, lambda ;
  double X[NEQS][NEQS] ;                              /*  eigenvectors  */
  double dgamma ;                                 /*  length of the face  */
  double den ;                                     /*  temp. denominator  */
  int i, j ;                                          /*  dummy indicies  */
  SolVec v0, Vtmp ;
  enum { conserved, primitive } reconstruct = primitive ;

  int dbugprint = 0 ;
/*   if( fabs(x0[1]+.719569)<.001 && fabs(x1[1]+.86525)<.001 ){ */
/*     printf("caught the edge\n"); */
/*     dbugprint = 1 ; */
/*   } */
```

$$\varpi_a = 1 - \varpi + \varpi Q P_y$$

$$\hat{n}\Delta\Gamma = 2(SP_y - QP_y, QP_x - SP_x)$$

```
  quadrature_point( x0, x1, SP, QP );
  pia = 1 - axisym + axisym * QP[1] ;
  normal[0] = 2 * ( SP[1] - QP[1] ) ;
  normal[1] = 2 * ( QP[0] - SP[0] ) ;
  dgamma = dnrm2( 2, normal, 1 );
  FOR(i,0,NDIMS) normal[i] /= dgamma ;
```

Reconstruct to the left and right states of the conserved variables. Then decode the parameter, primitive, and Roe-averaged variables at the face.


```
                                  /*  reconstructing conserved vectors  */
  FOR(i,0,NDIMS) r[i] = QP[i] - x0[i] ;
  if( reconstruct == conserved ){
    FOR(i,0,NEQS) Ul[i] = u0[i] + psi0[i] * (r[0] * gx0[i] + r[1] * gy0[i]) ;
  }
  else{
    conserved_to_primitive( u0, v0 );
    FOR(i,0,NEQS) Vtmp[i] = v0[i] + psi0[i] * (r[0] * gx0[i] + r[1] * gy0[i]) ;
    primitive_to_conserved( Vtmp, Ul );
  }
  FOR(i,0,NDIMS) r[i] = QP[i] - x1[i] ;
  if( reconstruct == conserved ){
    FOR(i,0,NEQS) Ur[i] = u1[i] + psi1[i] * (r[0] * gx1[i] + r[1] * gy1[i]) ;
  }
  else{
    conserved_to_primitive( u1, v0 );
    FOR(i,0,NEQS) Vtmp[i] = v0[i] + psi1[i] * (r[0] * gx1[i] + r[1] * gy1[i]) ;
    primitive_to_conserved( Vtmp, Ur );
  }

  conserved_to_parameter( Ul, Zl );          /*  variable transformations  */
  conserved_to_parameter( Ur, Zr );
  parameter_to_primitive( Zl, Vl );
  parameter_to_primitive( Zr, Vr );
  Roe_average_edge( Zl, Zr, Vtilde, &Htilde, &atilde );
  if( dbugprint ){
    printf("Ul:");
    FOR(i,0,NEQS) printf(" %f", Ul[i] ); puts("");
    printf("Ur:");
    FOR(i,0,NEQS) printf(" %f", Ur[i] ); puts("");
    printf("Vl:");
    FOR(i,0,NEQS) printf(" %f", Vl[i] ); puts("");
    printf("Vr:");
    FOR(i,0,NEQS) printf(" %f", Vr[i] ); puts("");
    printf("Zl:");
    FOR(i,0,NEQS) printf(" %f", Zl[i] ); puts("");
    printf("Zr:");
    FOR(i,0,NEQS) printf(" %f", Zr[i] ); puts("");
  }
```

Handle the two cases of systems and scalars separately.

```
  if( ! scalar ){                                          /*  systems  */
```

The projected velocities, $\tilde{\mathcal{V}} = (\tilde{u}, \tilde{v}) \cdot \hat{n}$. Also, get the speed of sound to left and right, used in eigenvalue limiting.

```
    veltilde = Vtilde[1] * normal[0] + Vtilde[2] * normal[1] ;
    vell = Vl[1] * normal[0] + Vl[2] * normal[1] ;
    velr = Vr[1] * normal[0] + Vr[2] * normal[1] ;
    al = a_from_rho_P( Vl[0], Vl[3] );
    ar = a_from_rho_P( Vr[0], Vr[3] );
    if( dbugprint ) printf("al=%f atilde=%f ar=%f\n", al, atilde, ar );
```

The absolute value of the projected Jacobian is written,

$$|A \cdot \hat{n}| = X|\Lambda|X^{-1}$$

Then $X^{-1}(U_R - U_L)$ is formed as,

$$\tilde{X}^{-1}dU = \frac{1}{2\tilde{a}^2}\begin{pmatrix} 2\tilde{a}^2 d\rho - 2dP \\ 2\tilde{a}^2(n_x dv - n_y du) \\ dP + \tilde{\rho}\tilde{a}d\mathcal{V} \\ dP - \tilde{\rho}\tilde{a}d\mathcal{V} \end{pmatrix}$$

```
    den = 0.5 / SQUARE( atilde );
    xdu[0] = Vr[0] - Vl[0] - 2 * den * ( Vr[3] - Vl[3] );
    xdu[1] = normal[0] * (Vr[2] - Vl[2]) - normal[1] * (Vr[1] - Vl[1]) ;
    xdu[2] = den * ( Vr[3] - Vl[3] + Vtilde[0] * atilde * (velr - vell) );
    xdu[3] = den * ( Vr[3] - Vl[3] - Vtilde[0] * atilde * (velr - vell) );
```

The absolute values of the eigenvalues are,

$$|\tilde{\Lambda}| = (|\tilde{\mathcal{V}}|, |\tilde{\mathcal{V}}|, |\tilde{\mathcal{V}} + \tilde{a}|, |\tilde{\mathcal{V}} - \tilde{a}|)$$

Perform eigenvalue limiting on all eigenvalues that can go to zero.

```
    lambda[0] = lambda[1] = eigenvalue_limiter( vell, veltilde, velr );
/*     lambda[0] = lambda[1] = fabs( veltilde ); */
    lambda[2] = eigenvalue_limiter( vell+al, veltilde+atilde, velr+ar );
    lambda[3] = eigenvalue_limiter( vell-al, veltilde-atilde, velr-ar );
```

The eigenvectors are,

$$\tilde{X} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ \tilde{u} & -n_y & \tilde{u} + \tilde{a}n_x & \tilde{u} - \tilde{a}n_x \\ \tilde{v} & n_x & \tilde{v} + \tilde{a}n_y & \tilde{v} - \tilde{a}n_y \\ \frac{\tilde{u}^2+\tilde{v}^2}{2} & \tilde{v}n_x - \tilde{u}n_y & \tilde{H} + \tilde{a}\tilde{\mathcal{V}} & \tilde{H} - \tilde{a}\tilde{\mathcal{V}} \end{bmatrix}$$

```
    X[0][0] = X[0][2] = X[0][3] = 1 ;
    X[0][1] = 0 ;
    X[1][0] = Vtilde[1] ;
    X[1][1] = - normal[1] ;
```



```
    X[1][2] = Vtilde[1] + atilde * normal[0] ;
    X[1][3] = Vtilde[1] - atilde * normal[0] ;
    X[2][0] = Vtilde[2] ;
    X[2][1] = normal[0] ;
    X[2][2] = Vtilde[2] + atilde * normal[1] ;
    X[2][3] = Vtilde[2] - atilde * normal[1] ;
    X[3][0] = VEL2(&Vtilde[1]) / 2 ;
    X[3][1] = Vtilde[2] * normal[0] - Vtilde[1] * normal[1] ;
    X[3][2] = Htilde + atilde * veltilde ;
    X[3][3] = Htilde - atilde * veltilde ;
```

Now form the distributions, completing this routine.

```
    FOR(i,0,NEQS){
      for( dist0[i] = j = 0 ; j < NEQS ; ++j )
        dist0[i] += X[i][j] * lambda[j] * xdu[j] ;
      dist1[i] = -( dist0[i] *= pia * dgamma / 2 );
    }

    if( dbugprint ){
      FOR(i,0,NEQS){
        FOR(j,0,NEQS) printf("X[%d][%d]=%f ", i, j, X[i][j] );
        puts("");
      }
      FOR(i,0,NEQS) printf("lambda[%d]=%f ", i, lambda[i] );
      puts("");
      FOR(i,0,NEQS) printf("xdu[%d]=%f ", i, xdu[i] );
      puts("");
    }
  }
```

For scalar, the flux Jacobian needs to be hardwired.

```
  else{                                                    /*  scalar  */
    X[0][0] = cos(AOA) ;                          /*  uniform advection  */
    X[0][1] = sin(AOA) ;

    dist1[0] = -( dist0[0] = dgamma + pia * (Ur[0] - Ul[0]) *
                  fabs(X[0][0] * normal[0] + X[0][1] * normal[1]) / 2 );
  }
}
```

6.8 Boundary Flux for Finite Volume

Evaluate and distribute the flux through a boundary edge. The interior is to the right of the edge, and the boundary to the left. The quadrature points are at the $\frac{1}{4}$ and $\frac{3}{4}$ points along the edge, with distributions going to the nearest node.

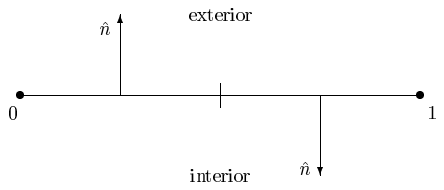


Figure 2: Finite volume boundary edge

Using the convention defined for interior edges, the distribution to node 0 is obtained by sending node 0 as the split-point and replacing the solution at node 1 with the boundary condition. The reverse procedure is sent for obtaining the distribution to node 1.

```
void FV_distrib_boundary( Bound_edge *bedge )
{
  Node *n0 = bedge->node[0], *n1 = bedge->node[1] ;
  SolVec BU, Bgx, Bgy, Bpsi, BV ;                        /*  boundary data  */
  double *Igx, *Igy, *Ipsi ;                    /*  interior reconstruction  */
  SolVec dist0, dist1 ;                              /*  flux distributions  */
  int eq ;
  void (*flux)();                          /*  pointer to the flux function  */
  int reconst = 0 ;                  /*  0=constant, 1=linear reconstruction  */
  double dx, dy, len1, len2, len3 ;  /*  for computing mirror velocities     */

  FOR(eq,0,NEQS){       /* zero gradient and unity limiter for boundary data */
    Bgx[eq] = Bgy[eq] = 0 ;
    Bpsi[eq] = 1 ;
  }
  Igx  = Bgx  ;
  Igy  = Bgy  ;
  Ipsi = Bpsi ;
```

Do node 0 and then node 1. Evaluate central difference portion and then artificial dissipation.

```
  if( ! scalar ) flux = &Euler_flux_conserved ;
  else flux = &scalar_flux ;

  if( reconst ){
    Igx  = n0->fv.gradx ;
    Igy  = n0->fv.grady ;
    Ipsi = n0->fv.psi   ;
```



```
}

/*  get boundary state, assuming no reconstruction to the face  */
switch( bedge->bc ){
case extrapolation:                              /*  zero-order extrapolation  */
  FOR(eq,0,NEQS) BV[eq] = n0->vars.primitive[eq] ;
  break;

case freestream_nondim:                                /*  nondim freestream  */
  BV[0] = 1 ;
  BV[1] = cos( AOA );
  BV[2] = sin( AOA );
  BV[3] = P_from_rho_T( BV[0], 1 );
  break;

case freestream_stagnant:                            /*  stagnant freestream  */
  BV[0] = 1 ;
  BV[1] = BV[2] = 0 ;
  BV[3] = P_from_rho_T( BV[0], 1 );
  break;

case freestream_second:                              /*  second freestream   */
  BV[0] = 1 ;
  BV[1] = cos( -AOA ) * sqrt( BC_pressure_ratio ) / BC_Mach_ratio ;
  BV[2] = sin( -AOA ) * sqrt( BC_pressure_ratio ) / BC_Mach_ratio ;
  BV[3] = BC_pressure_ratio * P_from_rho_T( BV[0], 1 );
  break;

case vacuum:                                                  /*  vacuum  */
  BV[0] = 0.001;
  BV[1] = cos( AOA );
  BV[2] = sin( AOA );
  BV[3] = P_from_rho_T( BV[0], 1 );
  break;

case wall_inviscid:                                  /*  inviscid wall or  */
case wall_viscous:                                   /*  viscous wall      */
  BV[0] = n0->vars.primitive[0] ;    /* match density and pressure  */
  BV[3] = n0->vars.primitive[3] ;
  /*  mirror the velocities  */
  dx = n1->geom.coord[0] - n0->geom.coord[0] ;
  dy = n1->geom.coord[1] - n0->geom.coord[1] ;
  len1 = SQUARE( dx ) + SQUARE( dy ) ;
  len2 = ( SQUARE( dx ) - SQUARE( dy ) ) / len1 ;
  len3 = 2 * dx * dy / len1 ;
  BV[1] = len2 * n0->vars.primitive[1] + len3 * n0->vars.primitive[2] ;
  BV[2] = len3 * n0->vars.primitive[1] - len2 * n0->vars.primitive[2] ;
  break;

default:
```

```
    printf("Bad BC = %d for edge with end points\n( %f, %f ), ( %f, %f )\n",
           bedge->bc, n0->geom.coord[0], n0->geom.coord[1],
           n1->geom.coord[0], n1->geom.coord[1] );
    puts("Stopping");
    exit(1);
  }
  primitive_to_conserved( BV, BU );

  FV_edge_central( n0->geom.coord, n1->geom.coord, n0->geom.coord,
                   n0->vars.conserved, Igx, Igy, Ipsi,
                   BU, Bgx, Bgy, Bpsi, dist0, dist1, flux ) ;
  FOR(eq,0,NEQS) n0->update.flux[eq] += dist0[eq] ;
```

Now artificial dissipation.

```
  FV_art_diss( n0->geom.coord, n1->geom.coord, n0->geom.coord,
               n0->vars.conserved, Igx, Igy, Ipsi,
               BU, Bgx, Bgy, Bpsi, dist0, dist1 );
  FOR(eq,0,NEQS) n0->update.art_diss[eq] += dist0[eq] ;
```

Start second node.

```
  if( reconst ){
    Igx  = n1->fv.gradx ;
    Igy  = n1->fv.grady ;
    Ipsi = n1->fv.psi   ;
  }

  /*  get boundary state, assuming no reconstruction to the face  */
  switch( bedge->bc ){
  case extrapolation:                          /*  zero-order extrapolation  */
    FOR(eq,0,NEQS) BV[eq] = n1->vars.primitive[eq] ;
    break;

  case freestream_nondim:                          /*  nondim freestream  */
    BV[0] = 1 ;
    BV[1] = cos( AOA );
    BV[2] = sin( AOA );
    BV[3] = P_from_rho_T( BV[0], 1 );
    break;

  case freestream_stagnant:                        /*  stagnant freestream  */
    BV[0] = 1 ;
    BV[1] = BV[2] = 0 ;
    BV[3] = P_from_rho_T( BV[0], 1 );
    break;
```



```
  case freestream_second:                          /*  second freestream   */
    BV[0] = 1 ;
    BV[1] = cos( -AOA ) * sqrt( BC_pressure_ratio ) / BC_Mach_ratio ;
    BV[2] = sin( -AOA ) * sqrt( BC_pressure_ratio ) / BC_Mach_ratio ;
    BV[3] = BC_pressure_ratio * P_from_rho_T( BV[0], 1 );
    break;

  case vacuum:                                                  /*  vacuum  */
    BV[0] = 0.001;
    BV[1] = cos( AOA );
    BV[2] = sin( AOA );
    BV[3] = P_from_rho_T( BV[0], 1 );
    break;

  case wall_inviscid:                                /*  inviscid wall or  */
  case wall_viscous:                                 /*  viscous wall      */
    BV[0] = n1->vars.primitive[0] ;          /* match density and pressure  */
    BV[3] = n1->vars.primitive[3] ;
    /*  mirror the velocities  */
    dx = n1->geom.coord[0] - n0->geom.coord[0] ;
    dy = n1->geom.coord[1] - n0->geom.coord[1] ;
    len1 = SQUARE( dx ) + SQUARE( dy ) ;
    len2 = ( SQUARE( dx ) - SQUARE( dy ) ) / len1 ;
    len3 = 2 * dx * dy / len1 ;
    BV[1] = len2 * n1->vars.primitive[1] + len3 * n1->vars.primitive[2] ;
    BV[2] = len3 * n1->vars.primitive[1] - len2 * n1->vars.primitive[2] ;
    break;

  default:
    printf("Bad BC = %d for edge with end points\n( %f, %f ), ( %f, %f )\n",
           bedge->bc, n0->geom.coord[0], n0->geom.coord[1],
           n1->geom.coord[0], n1->geom.coord[1] );
    puts("Stopping");
    exit(1);
  }
  primitive_to_conserved( BV, BU );

  FV_edge_central( n0->geom.coord, n1->geom.coord, n1->geom.coord,
                   BU, Bgx, Bgy, Bpsi, n1->vars.conserved, Igx, Igy, Ipsi,
                   dist0, dist1, flux ) ;
  FOR(eq,0,NEQS) n1->update.flux[eq] += dist1[eq] ;
```

Artificial dissipation.

```
  FV_art_diss( n0->geom.coord, n1->geom.coord, n1->geom.coord,
               BU, Bgx, Bgy, Bpsi,
               n1->vars.conserved, Igx, Igy, Ipsi, dist0, dist1 );
  FOR(eq,0,NEQS) n1->update.art_diss[eq] += dist1[eq] ;
}
```

7 Fluctuation Splitting Functions

```
#include "blas.h"       /*  ddot            */
#include <math.h>       /*  pow, sqrt, fabs */
#include <float.h>      /*  DBL_EPSILON     */
#include <stdlib.h>     /*  exit            */
#include "basics.h"
#include "rhs.h"
#include "TriMath.h"    /*  Triave          */
#include "transform.h"  /*  dW_dZ           */
#include "therm.h"      /*  a(r,P), P(r,T)  */
#include "limiter.h"    /*  limiters        */
#include "bc.h"         /*  Bound_edge      */
```

7.1 FS Cell Distribution

Compute the inviscid distributions to the nodes for a triangular cell using the fluctuation splitting scheme of Sidilkover[Sid94].

$$\phi = -\int_\Omega \varpi_a \nabla \cdot F \; d\Omega$$

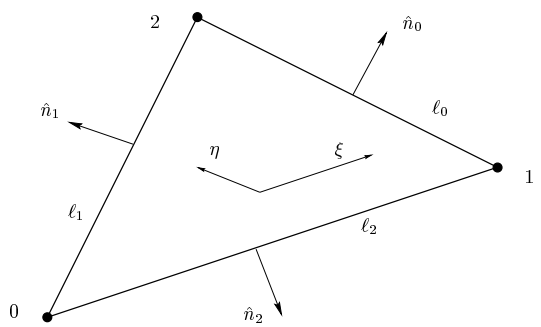


Figure 3: Geometry nomenclature for fluctuation splitting cell.

The axisymmetric parameter ϖ_a is equal to 1 for 2-D or y for axisymmetric.

```
void FS_distrib_cell( Cell *cell )
{
```



```
  Node **node = cell->geom.connect.nodes ;
  double piabar, pia[3] ;                          /*  axi flags            */
  double vn0[4], vn2[4], a[4] ;            /* V * n, a at 3 nodes plus tilde */
  int i, j ;                                       /*  dummy indicie        */
  float c1, c2 ;                                   /*  temporary factors    */

  SolVec xi_z, eta_z, xi_w, eta_w ;                /*  curvilinear derivs   */
  double dw_dz[NEQS][NEQS] ;                       /*  variable transform.  */
  double du_dw[NEQS][NEQS] ;                       /*  variable transform.  */
  double node_AW[3][NEQS] ;                        /*  nodal n * A dW       */
  SolVec avg_AW ;                                  /*  avg n * A dW         */

  Coord n0, n2 ;                                   /*  normal vectors       */
  double l0, l2 ;                                  /*  edge lengths         */

  SolVec xi_f_2, eta_f_2 ;                         /*  2D aux. flucs        */
  SolVec xi_f_a, eta_f_a ;                         /*  axi aux. flucs       */
  SolVec xi_f, eta_f ;                             /*  auxil. fluctuations  */
  SolVec xi_f_art, eta_f_art ;                     /*  aux. art. diss.      */
  SolVec f_xt, f_et ;                              /*  conserved flucs      */
  SolVec f_xa, f_ea ;                              /*  art diss cons flucs  */
  SolVec fb ;                                      /*  cell fluctuation     */
  SolVec artd ;                                    /*  cell art. diss       */

  double M_limit, epsVA ;                          /*  for limiters         */
  double M_alpha[NEQS][NEQS], M_beta[NEQS][NEQS] ; /*  sign of wavespeeds   */
  double eig_limit[NEQS][NEQS] ;                   /*  matrix for e-val lim.*/
```

Assign the axisymmetric weighting factor at the nodes and cell center, if appropriate.

```
/*   puts("\nEntering FS distrib\n");              */
  FOR(i,0,3) pia[i] = ( ! axisym ) ? 1 : node[i]->geom.coord[1] ;
  piabar = TriaveH( pia );
```

Form the gradients of the parameter vector in (ξ, η) space, with constant variations for linear data representation,

$$\Delta_\xi Z = Z_1 - Z_0, \qquad \Delta_\eta Z = Z_2 - Z_1$$

Then assign the transformation from parameter to auxiliary variables, W_Z, defined as $dW = W_Z \, dZ$, at the cell-averaged state. Finally, construct the gradients of the auxiliary variables as,

$$\Delta_\xi W = W_Z \Delta_\xi Z, \qquad \Delta_\eta W = W_Z \Delta_\eta Z$$

```
  FOR(i,0,NEQS){
    xi_z[i] = node[1]->vars.parameter[i] - node[0]->vars.parameter[i] ;
    eta_z[i] = node[2]->vars.parameter[i] - node[1]->vars.parameter[i] ;
  }
```

```
dW_dZ( cell->tilde.parameter, dw_dz );
FOR(i,0,NEQS){
  xi_w[i] = ddot( NEQS, dw_dz[i], 1, xi_z, 1 );
  eta_w[i] = ddot( NEQS, dw_dz[i], 1, eta_z, 1 );
}
```

Normals are formed as,

$$\ell_0 \hat{n}_0 = (\Delta_\eta y, -\Delta_\eta x), \qquad \ell_2 \hat{n}_2 = (\Delta_\xi y, -\Delta_\xi x)$$

```
n0[0] = node[2]->geom.coord[1] - node[1]->geom.coord[1] ;
n0[1] = node[1]->geom.coord[0] - node[2]->geom.coord[0] ;
n2[0] = node[1]->geom.coord[1] - node[0]->geom.coord[1] ;
n2[1] = node[0]->geom.coord[0] - node[1]->geom.coord[0] ;
l0 = dnrm2( NDIMS, n0, 1 );
l2 = dnrm2( NDIMS, n2, 1 );
FOR(i,0,NDIMS){
  n0[i] /= l0 ;
  n2[i] /= l2 ;
}
```

Projected velocities are $\mathcal{V} = \vec{V} \cdot \hat{n}$. Indicies 0–2 are for nodal velocities and speeds of sound, while index 3 is for the cell-averaged values.

```
FOR(i,0,3){
  vn0[i] = ddot( NDIMS, n0, 1, &(node[i]->vars.primitive[1]), 1 );
  vn2[i] = ddot( NDIMS, n2, 1, &(node[i]->vars.primitive[1]), 1 );
  a[i] = a_from_rho_P( node[i]->vars.primitive[0],
                       node[i]->vars.primitive[3] );
}
vn0[3] = ddot( NDIMS, &(cell->tilde.primitive[1]), 1, n0, 1 );
vn2[3] = ddot( NDIMS, &(cell->tilde.primitive[1]), 1, n2, 1 );
a[3] = cell->tilde.a ;
```

The basic 2-D fluctuations in auxiliary variables are formed as,

$$\breve{\phi}_2^\xi = -\bar{\varpi}_a \frac{\ell_0}{2} \hat{n}_0 \cdot \tilde{\mathcal{A}} \, \Delta_\xi W$$

$$\breve{\phi}_2^\eta = \bar{\varpi}_a \frac{\ell_2}{2} \hat{n}_2 \cdot \tilde{\mathcal{A}} \, \Delta_\eta W$$

where,

$$\hat{n} \cdot \mathcal{A} = \begin{bmatrix} \mathcal{V} & 0 & 0 & 0 \\ 0 & \mathcal{V} & 0 & \hat{n}_x \\ 0 & 0 & \mathcal{V} & \hat{n}_y \\ 0 & a^2 \hat{n}_x & a^2 \hat{n}_y & \mathcal{V} \end{bmatrix}$$


```
nA_dW( n0, vn0[3], a[3], xi_w, xi_f_2 );
nA_dW( n2, vn2[3], a[3], eta_w, eta_f_2 );
FOR(i,0,NEQS){
  xi_f_2[i]  *= (-piabar) * l0 * 0.5 ;
  eta_f_2[i] *=   piabar  * l2 * 0.5 ;
}
```

The additional axisymmetric terms are,

$$\breve{\phi}_a^\xi = -\frac{1}{4} \left[\breve{\phi}_2^\xi + \left(\frac{1}{3}\right) \left(\frac{\ell_0}{2}\right) \hat{n}_0 \cdot \sum_{j=0}^{2} \varpi_{a_j} \mathcal{A}_j \, \Delta_\xi W \right]$$

$$\breve{\phi}_a^\eta = -\frac{1}{4} \left[\breve{\phi}_2^\eta - \left(\frac{1}{3}\right) \left(\frac{\ell_2}{2}\right) \hat{n}_2 \cdot \sum_{j=0}^{2} \varpi_{a_j} \mathcal{A}_j \, \Delta_\eta W \right]$$

If this is a 2-D calculation, these terms should be zero.

```
if( ! axisym ) FOR(i,0,NEQS) xi_f_a[i] = eta_f_a[i] = 0 ;
else{                                                  /* skip for 2-D  */
  FOR(i,0,3){
    nA_dW( n0, vn0[i], a[i], xi_w, node_AW[i] );
    FOR(j,0,NEQS) node_AW[i][j] *= l0 / 2 * pia[i] ;
  }
  TriaveV( NEQS, node_AW[0], node_AW[1], node_AW[2], avg_AW );
  FOR(i,0,NEQS) xi_f_a[i] = -( xi_f_2[i] + avg_AW[i] ) / 4 ;

  FOR(i,0,3){
    nA_dW( n2, vn2[i], a[i], eta_w, node_AW[i] );
    FOR(j,0,NEQS) node_AW[i][j] *= l2 / 2 * pia[i] ;
  }
  TriaveV( NEQS, node_AW[0], node_AW[1], node_AW[2], avg_AW );
  FOR(i,0,NEQS) eta_f_a[i] = -( eta_f_2[i] - avg_AW[i] ) / 4 ;

  if( safety && ! axisym )
    FOR(i,0,NEQS)
      if( MAX( fabs(xi_f_a[i]), fabs(eta_f_a[i]) ) /
        ( MAX( fabs(xi_f_2[i]), fabs(eta_f_2[i]) ) + FLT_EPSILON ) > 0.0001 ){
        puts("axi fluctuation not zero for 2D calc");
        printf("xi_f_2[%d]= %f, xi_f_a= %f, eta_f_2= %f, eta_f_a= %f\n",
               i, xi_f_2[i], xi_f_a[i], eta_f_2[i], eta_f_a[i] );
      }
}
```

The full fluctuation is formed as the sum of the 2-D and axi parts,

$$\breve{\phi}^\xi = \breve{\phi}_2^\xi + \breve{\phi}_a^\xi$$

```
FOR(i,0,NEQS){
  xi_f[i] = xi_f_2[i] + xi_f_a[i] ;
  eta_f[i] = eta_f_2[i] + eta_f_a[i] ;
}
```

Now apply limiting to make the scheme linearity preserving (second-order).

$$\breve{\phi}_j^{*\xi} = \breve{\phi}_j^{\xi} + M(-\breve{\phi}_j^{\xi}, \breve{\phi}_j^{\eta})$$

$$\breve{\phi}_j^{*\eta} = \breve{\phi}_j^{\eta} - M(-\breve{\phi}_j^{\xi}, \breve{\phi}_j^{\eta})$$

$$M = \breve{\phi}_j^{\eta}\psi(Q_j), \qquad Q_j = -\frac{\breve{\phi}_j^{\xi}}{\breve{\phi}_j^{\eta}}$$

Send $S\sqrt{S}$ as the van Albada parameter.

```
epsVA = pow( sqrt(cell->geom.metric.area), 3 );
FOR(i,0,NEQS){
  xi_f[i] += ( M_limit = limiter( -xi_f[i], eta_f[i], epsVA ) );
  eta_f[i] -= M_limit ;
}
```

Now enable upwinding through the introduction of artificial dissipation.

$$\breve{\phi}'^{\xi} = \text{sign}(\alpha)\breve{\phi}^{*\xi} = M_\alpha \breve{\phi}^{*\xi}$$

$$\breve{\phi}'^{\eta} = \text{sign}(\beta)\breve{\phi}^{*\eta} = M_\beta \breve{\phi}^{*\eta}$$

Note: currently doing eigenvalue limiting inside the M matrix—may not have to do this.

```
abs_wavespeed( vn0[3], a[3], n0, M_alpha );
abs_wavespeed( vn2[3], a[3], n2, M_beta  );
xi_f_art[0] = M_alpha[0][0] * xi_f[0] ;
eta_f_art[0] = M_beta[0][0] * eta_f[0] ;
FOR(i,1,NEQS){
  xi_f_art[i] = eta_f_art[i] = 0 ;
  FOR(j,1,NEQS){
    xi_f_art[i] += M_alpha[i][j] * xi_f[j] ;
    eta_f_art[i] += M_beta[i][j] * eta_f[j] ;
  }
}
```

Eigenvalue limiting takes the form of an additional artificial dissipation that is selectively added to the velocity and pressure equations only in the presence of a sonic expansion.

$$\breve{\phi}_{EL}^{\xi} = -\bar{\varpi}_a \frac{\ell_0}{2}[\text{matrix}]\Delta_\xi W$$

$$\breve{\phi}_{EL}^{\eta} = +\bar{\varpi}_a \frac{\ell_2}{2}[\text{matrix}]\Delta_\eta W$$

For ξ, the node to the left for eigenvalue limiting is node 0, while the node to the right is node 1. For η, the left node is node 2 while the right node is node 1.

```
if( limited_wavespeed( n0, vn0[0],a[0],vn0[3],a[3],vn0[1],a[1], eig_limit ) )
  FOR(i,0,NEQS)
    xi_f_art[i] -= piabar * l0 / 2 * ddot( NEQS, xi_w, 1, eig_limit[i], 1 );
if( limited_wavespeed( n2, vn2[2],a[2],vn2[3],a[3],vn2[1],a[1], eig_limit ) )
  FOR(i,0,NEQS)
    eta_f_art[i] += piabar * l2 / 2 * ddot( NEQS, eta_w, 1, eig_limit[i], 1 );
```

The fluctuations are converted from the auxiliary variables to the conserved variables as,

$$\phi = \tilde{U}_W \breve{\phi}$$

$$U_W = \begin{bmatrix} 1 & 0 & 0 & \frac{1}{a^2} \\ u & 1 & 0 & \frac{u}{a^2} \\ v & 0 & 1 & \frac{v}{a^2} \\ \frac{V^2}{2} & u & v & \frac{1}{\gamma-1}\frac{T_0}{T} \end{bmatrix}$$

$$\frac{T_0}{T} = 1 + \frac{\gamma-1}{2}M^2$$

$$\frac{1}{\gamma-1}\frac{T_0}{T} = \frac{1}{\gamma-1} + \frac{V^2}{2a^2}$$

```
dU_dW( a[3], &(cell->tilde.primitive[1]), du_dw );
FOR(i,0,NEQS){
  f_xt[i] = f_et[i] = f_xa[i] = f_ea[i] = 0 ;
  FOR(j,0,NEQS){
    f_xt[i] += du_dw[i][j] * xi_f[j] ;
    f_et[i] += du_dw[i][j] * eta_f[j] ;
    f_xa[i] += du_dw[i][j] * xi_f_art[j] ;
    f_ea[i] += du_dw[i][j] * eta_f_art[j] ;
  }
}
```

Finally, the fluctuations are distributed to the nodes.

$$S_0 U_{0_t} \leftarrow \frac{\phi^{*\xi} - \phi'^{\xi}}{2} + COE$$

$$S_1 U_{1_t} \leftarrow \frac{\phi^{*\xi} + \phi'^{\xi}}{2} + \frac{\phi^{*\eta} + \phi'^{\eta}}{2} + COE$$

$$S_2 U_{2_t} \leftarrow \frac{\phi^{*\eta} - \phi'^{\eta}}{2} + COE$$

$$S_i U_{i_t} \leftarrow \frac{1}{4}\left[(2+i-i^2)\left(\phi^{*\xi} - (-1)^i\phi'^{\xi}\right) + i(3-i)\left(\phi^{*\eta} - (-1)^i\phi'^{\eta}\right)\right] + COE$$

```
  FOR(i,0,NEQS) artd[i] = 0 ;
  FOR(i,0,3){
    c1 = ( 2 + i - SQUARE(i) ) / 4. ;
    c2 = ( i * ( 3 - i ) ) / 4. ;
    FOR(j,0,NEQS){
      node[i]->update.flux[j] += c1 * f_xt[j] + c2 * f_et[j] ;
      node[i]->update.art_diss[j] -= pow(-1,i) * ( c1*f_xa[j] + c2*f_ea[j] );
      artd[j] -= pow(-1,i) * ( c1 * f_xa[j] + c2 * f_ea[j] ) ;
    }
  }

  if( safety && ! axisym ){
    FOR(i,0,NEQS) if( fabs( artd[i] ) > FLT_EPSILON )
      printf("Problem, artificial dissipation does not sum to zero\n"
             "for equation %d, = %e\n", i, artd[i] );
    cell_fluc( node, fb );
    FOR(i,0,NEQS) if( fabs( fb[i] - f_xt[i] - f_et[i] ) /
                      ( fabs( fb[i] ) + FLT_EPSILON ) > 0.0001 )
      printf("Problem, upwind fluctuation does not equal cell fluc\n"
             "for equation %d, cell fluc = %e\n"
             "xi_fluc = %e, eta_fluc = %e, sum = %e\n", i, fb[i], f_xt[i],
             f_et[i], f_xt[i] + f_et[i] );
  }
}
```

7.2 $\hat{n}\cdot\mathcal{A}\ \Delta W$

Evaluate the product of the projected flux Jacobian with the jump in auxiliary variables.

$$\hat{n}\cdot\mathcal{A} = \begin{bmatrix} \mathcal{V} & 0 & 0 & 0 \\ 0 & \mathcal{V} & 0 & \hat{n}_x \\ 0 & 0 & \mathcal{V} & \hat{n}_y \\ 0 & a^2\hat{n}_x & a^2\hat{n}_y & \mathcal{V} \end{bmatrix}$$

```
void nA_dW( Coord n, double vn, double a, SolVec dw, SolVec prod )
{
  int i;

  FOR(i,0,NEQS) prod[i] = vn * dw[i] ;
  prod[1] += n[0] * dw[3] ;
  prod[2] += n[1] * dw[3] ;
  prod[3] += SQUARE(a) * ddot(NDIMS, n, 1, &(dw[1]), 1);
}
```


7.3 Sign of Wavespeed Vector

Evaluate the matrices $\mathrm{sign}(\alpha)$ or $\mathrm{sign}(\beta)$. For supersonic flow,

$$M = \mathrm{sign}(V)I$$

while for subsonic flow,

$$M = \begin{bmatrix} \mathrm{sign}(V) & 0 & 0 & 0 \\ 0 & n_y^2\mathrm{sign}(V) & -n_x n_y\mathrm{sign}(V) & \frac{n_x}{a} \\ 0 & -n_x n_y\mathrm{sign}(V) & n_x^2\mathrm{sign}(V) & \frac{n_y}{a} \\ 0 & an_x & an_y & 0 \end{bmatrix}$$

```
void abs_wavespeed( double V, double a, Coord n, double M[NEQS][NEQS] )
{
/*    double f1 = copysign(1,V) ; */
  double f1 = (V>0) ? 1 : -1;

  M[0][0] = f1 ;
  M[0][1] = M[0][2] = M[0][3] = M[1][0] = M[2][0] = M[3][0] = 0 ;

  if( fabs(V) > a ){
    M[1][1] = M[2][2] = M[3][3] = f1 ;
    M[1][2] = M[1][3] = M[2][1] = M[2][3] = M[3][1] = M[3][2] = 0 ;
  }
  else{
    M[1][1] = SQUARE( n[1] ) * f1 ;
    M[2][2] = SQUARE( n[0] ) * f1 ;
    M[1][2] = M[2][1] = - n[0] * n[1] * f1 ;
    M[1][3] = n[0] / a ;
    M[2][3] = n[1] / a ;
    M[3][1] = n[0] * a ;
    M[3][2] = n[1] * a ;
    M[3][3] = 0 ;
  }
}
```

7.4 Basic Inviscid Fluctuation

Evaluate the inviscid fluctuation on a triangle. The fluctuation is defined as,

$$\phi = -\int_\Omega \varpi_a \nabla\cdot F\, d\Omega = -\int_\Omega \varpi_a F_Z \cdot \nabla Z\, d\Omega$$

ϖ_a is 1 for 2-D and y for axisymmetric. Z varies linearly over the triangle. Integration yields,

$$\phi = -S_T\left[\bar{\varpi}_a\bar{F}_Z - \frac{1}{4}\left(\bar{\varpi}_a\bar{F}_Z - \frac{1}{3}\sum_{j=0}^{2}\varpi_{a_j}F_{Z_j}\right)\right]\cdot\nabla Z$$

with,

$$\nabla Z = -\frac{1}{2S_T}\sum_{i=0}^{2} Z_i \ell_i \hat{n}_i$$

$\hat{n}_i$ is an outward unit normal opposite node i.

```
void cell_fluc( Node *n[3], SolVec phi )
{
  double pibar, pin[3] ;                  /*  axisym flags                  */
  double xy[3][NDIMS] ;                   /*  nodal coordinates             */
  double Zn[3][NEQS] ;                    /*  nodal parameter vectors       */
  double Z[NEQS] ;                        /*  cell-avg parameter vector     */
  double gZ[NDIMS][NEQS] ;                /*  gradient of parameter vectors */
  double df_dz[NDIMS][NEQS][NEQS] ;       /*  flux Jacobian                 */
  double FZ[NDIMS][NEQS][NEQS] ;
  int i, j, k, l ;                        /*  dummy indicie                 */

  FOR(i,0,3){
    FOR(j,0,NDIMS) xy[i][j] = n[i]->geom.coord[j] ;
    FOR(j,0,NEQS)  Zn[i][j] = n[i]->vars.parameter[j] ;
  }
  TriaveV( NEQS, Zn[0], Zn[1], Zn[2], Z );
```

Form the axisymmetric flag at nodes and cell center.

```
  FOR(i,0,3) pin[i] = axisym ? n[i]->geom.coord[1] : 1 ;
  pibar = TriaveH( pin );
```

Form gradient of parameter vector multiplied by cell area, $S_T\,\nabla Z$.

```
  tri_grad( NEQS, Zn, xy, gZ );
```

Form flux Jacobian, first at nodes and then cell center, adding the components as we go along. Then build into the fluctuation by multiplying with gradient of parameter vector.

```
  FOR(i,0,NDIMS) FOR(j,0,NEQS) FOR(k,0,NEQS) FZ[i][j][k] = 0 ;

  FOR(i,0,3){
    dF_dZ( Zn[i], df_dz );
    FOR(j,0,NDIMS) FOR(k,0,NEQS) FOR(l,0,NEQS)
      FZ[j][k][l] -= pin[i] * df_dz[j][k][l] / 12 ;
  }

  dF_dZ( Z, df_dz );
  FOR(j,0,NDIMS) FOR(k,0,NEQS) FOR(l,0,NEQS)
```



```
    FZ[j][k][l] -= 0.75 * pibar * df_dz[j][k][l] ;

  FOR(i,0,NEQS)
    phi[i] = ddot( NEQS, FZ[0][i], 1, gZ[0], 1 )
           + ddot( NEQS, FZ[1][i], 1, gZ[1], 1 ) ;
}
```

7.5 Weak FS Boundary Conditions

Implement a weak boundary condition for fluctuation splitting. Given a boundary edge, defined so that the exterior is to the left of the edge, compute a first-order distribution accounting for the fluctuation through the edge. A fictitious ghost node is constructed, and the contribution to the fluctuation from $U_0 - U_1$ is neglected, retaining only the contribution from $U_f - U_{0,1}$. A total of two ghost nodes are constructed, one for each node of the boundary edge.

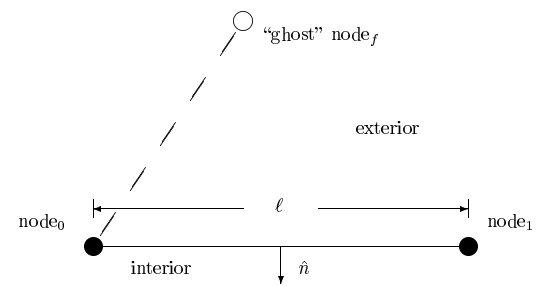


Figure 4: Boundary geometry for fictitious "ghost" node applied for the distribution to node$_0$. A similar ghost node would also be constructed for the distribution to node$_1$.

The basic fluctuation in auxiliary variables is,

$$\breve{\phi} = \frac{\varpi_a}{2}\ell\vec{\mathcal{A}}\cdot\hat{n}\Delta W$$

where ϖ_a is 1 for 2-D or y for axisymmetric. The change in auxiliary variables is formed as,

$$\Delta W = W_U(U_f - U_b) = W_Z(Z_f - Z_b)$$

The artificial dissipation is,

$$\breve{\phi}' = M_f\breve{\phi}$$

where,

$$M_f = \mathrm{sign}(\vec{\mathcal{A}}\cdot\hat{n})$$

Eigenvalue limiting is added to $\breve{\phi}'$ as,

$$+\varpi_a \frac{\ell}{2}[\text{limit\_wavespeed}]\Delta W$$

The distribution, in conserved variables, is,

$$\frac{1}{2}U_W(\breve{\phi}+\breve{\phi}')$$

```
extern void FS_distrib_boundary( Bound_edge *bedge )
{
  Node *node ;                /*  pointer to end nodes of the boundary edge  */
  int in, ie1, ie2 ;                                      /*  dummy index  */
  SolVec Uf, Vf, Zf ;                                /*  ghost node values  */
  SolVec dW, phi, phi_a ;
  double dw_dz[NEQS][NEQS], du_dw[NEQS][NEQS], Mf[NEQS][NEQS] ;
  double eig_limit[NEQS][NEQS] ;                    /*  matrix for e-val lim.*/
  Coord norm ;                      /*  inward unit normal to boundary edge  */
  double length ;                                         /*  edge length  */
  double pia ;                                       /*  axisymmetric flag  */
  double Vnf, Vnn ;                     /*  projected velocity along normal  */
  double af, an ;                                        /*  speed of sound  */
  float sfac =  .5  ;                  /*  factor < 1 to maintain stability  */
  Node2Cell *celln;
  int cln;

  /*  Form edge length and inward normal  */
  norm[0] = (bedge->node[1])->geom.coord[1] - (bedge->node[0])->geom.coord[1] ;
  norm[1] = (bedge->node[0])->geom.coord[0] - (bedge->node[1])->geom.coord[0] ;
  length = dnrm2( NDIMS, norm, 1 );
  FOR(in,0,NDIMS) norm[in] /= length ;

  /*  Loop on the 2 end nodes to form distributions  */
  FOR(in,0,2){
    cln = 1;
    node = bedge->node[in] ;
    pia = ( ! axisym ) ? 1 : node->geom.coord[1] ;
```

Use the same definition of ϖ_a as for the time evolution.

```
    if(pia < 10 *FLT_EPSILON){
      CELL_LOOP_AT_NODE(node){
        FOR(ie1,0,3)
          pia += (((celln->cell)->geom.connect.nodes[ie1])->geom.coord[1])
                 /(((celln->cell)->geom.connect.nodes[ie1])->bc ? 2 : 4);
          ++cln ;
      }
      pia /= cln ;
```



```
    }

    /*  Define ghost node values  */
/*     switch( node->bc ){ */
    switch( bedge->bc ){
    case extrapolation:                          /*  zero-order extrapolation  */
      FOR(ie1,0,NEQS) Vf[ie1] = node->vars.primitive[ie1] ;
      break;

    case freestream_nondim:                            /*  nondim freestream  */
      Vf[0] = 1 ;
      Vf[1] = cos( AOA );
      Vf[2] = sin( AOA );
      Vf[3] = P_from_rho_T( Vf[0], 1 );
      sfac  = MIN( sfac, .6 ) ;
      break;

    case freestream_stagnant:                         /*  stagnant freestream  */
      Vf[0] = 1 ;
      Vf[1] = Vf[2] = 0 ;
      Vf[3] = P_from_rho_T( Vf[0], 1 );
      break;

    case freestream_second:                            /*  second freestream   */
      Vf[0] = 1 ;
      Vf[1] = cos( -AOA ) * sqrt( BC_pressure_ratio ) / BC_Mach_ratio ;
      Vf[2] = sin( -AOA ) * sqrt( BC_pressure_ratio ) / BC_Mach_ratio ;
      Vf[3] = BC_pressure_ratio * P_from_rho_T( Vf[0], 1 );
      break;

    case vacuum:                                                  /*  vacuum  */
      Vf[0] = 0.001;
      Vf[1] = cos( AOA );
      Vf[2] = sin( AOA );
      Vf[3] = P_from_rho_T( Vf[0], 1 );
      break;

    case wall_inviscid:                                /*  inviscid wall or  */
    case wall_viscous:                                 /*  viscous wall      */
      Vf[0] = node->vars.primitive[0] ;      /* match density and pressure  */
      Vf[3] = node->vars.primitive[3] ;
      /*  mirror the velocities  */
      Vf[1] = ( SQUARE(norm[1]) - SQUARE(norm[0]) ) * node->vars.primitive[1]
              - 2 * norm[0] * norm[1] * node->vars.primitive[2] ;
      Vf[2] = ( SQUARE(norm[0]) - SQUARE(norm[1]) ) * node->vars.primitive[2]
              - 2 * norm[0] * norm[1] * node->vars.primitive[1] ;
      break;

    default:
      printf("Bad BC = %d for edge with an end point at\n( %f, %f )\n",
```

```
            bedge->bc, node->geom.coord[0], node->geom.coord[1] );
      puts("Stopping in FS_bound_dist");
      exit(1);
    }
    primitive_to_conserved( Vf, Uf );
    conserved_to_parameter( Uf, Zf );

    /*  Project velocities onto edge inward normal  */
    Vnf = ddot( NDIMS, &(Vf[1]), 1, norm, 1 );
    Vnn = ddot( NDIMS, &(node->vars.primitive[1]), 1, norm, 1 );

    /*  Speed of sound for ghost node and boundary node  */
    af = a_from_rho_P( Vf[0], Vf[3] );
    an = a_from_rho_P( node->vars.primitive[0], node->vars.primitive[3] );

    /*  Form dW = Wz ( Zf - Znode )  */
    dW_dZ( Zf, dw_dz );
    FOR(ie1,0,NEQS){
      dW[ie1] = 0 ;
      FOR(ie2,0,NEQS) dW[ie1] += dw_dz[ie1][ie2] *
                                 ( Zf[ie2] - node->vars.parameter[ie2] );
    }

    /*  Form fluctuation in auxiliary variables  */
    nA_dW( norm, Vnf, af, dW, phi );
    FOR(ie1,0,NEQS) phi[ie1] *= 0.5 * pia * length ;

    /*  Form artificial dissipation for upwinding  */
    abs_wavespeed( Vnf, af, norm, Mf );
    phi_a[0] = Mf[0][0] * phi[0] ;
    FOR(ie1,1,NEQS){
      phi_a[ie1] = 0 ;
      FOR(ie2,1,NEQS) phi_a[ie1] += Mf[ie1][ie2] * phi[ie2] ;
    }

    /*  Eigenvalue limiting  */
    if( limited_wavespeed( norm, Vnf, af, Vnf, af, Vnn, an, eig_limit ) )
      FOR(ie1,0,NEQS)
        phi_a[ie1] += pia * length / 2 * ddot( NEQS, dW, 1, eig_limit[ie1], 1);


    /*  Finally, assign the distribution  */
    dU_dW( af, &(Vf[1]), du_dw );
    FOR(ie1,0,NEQS) FOR(ie2,0,NEQS){
      node->update.flux[ie1] += du_dw[ie1][ie2] * phi[ie2] * 0.5 * sfac ;
      node->update.art_diss[ie1] += du_dw[ie1][ie2] * phi_a[ie2] * 0.5 * sfac ;
    }
  }  /*  End of loop over the 2 nodes for this boundary edge  */
}
```


7.6 FS Eigenvalue Limiting

This function is a wrapper around the eigenvalue limiter that provides the matrix required for additional dissipation for the prevention of expansion shocks with fluctuation splitting.

Send the coordinates of the normal vector along with the projected velocity and speed of sound to the left, center, and right. The function returns null if there is no eigenvalue limiting. The matrix returned is,

$$\begin{bmatrix} |\lambda'_{\ell_0}| & 0 & 0 & 0 \\ 0 & n_x^2 L^+ + n_y^2 |\lambda'_{\ell_1}| & n_x n_y (L^+ - |\lambda'_{\ell_1}|) & \frac{1}{q} n_x L^- \\ 0 & n_x n_y (L^+ - |\lambda'_{\ell_1}|) & n_x^2 |\lambda'_{\ell_1}| + n_y^2 L^+ & \frac{1}{a} n_y L^- \\ 0 & a n_x L^- & a n_y L^- & L^+ \end{bmatrix}$$

where,

$$L^+ = \frac{1}{2}(|\lambda'_{\ell_2}| + |\lambda'_{\ell_3}|), \quad L^- = \frac{1}{2}(|\lambda'_{\ell_2}| - |\lambda'_{\ell_3}|)$$

and,

$$|\lambda'_\ell| = |\lambda_{limited}| - |\lambda|$$

The eigenvalues are: $\lambda_0 = \lambda_1 = v$, $\lambda_2 = v + a$, and $\lambda_3 = v - a$.

```
int limited_wavespeed( Coord n, double vl, double al, double vc, double ac,
                       double vr, double ar, double matrix[NEQS][NEQS] ){
  double lim1 = eigenvalue_limiter(vl, vc, vr) -fabs(vc);
  double lim2 = eigenvalue_limiter(vl +al, vc +ac, vr +ar) -fabs(vc +ac);
  double lim3 = eigenvalue_limiter(vl -al, vc -ac, vr -ar) -fabs(vc -ac);
  double L_plus  = (lim2 +lim3) /2;
  double L_minus = (lim2 -lim3) /2;

  matrix[0][0] = lim1;
  matrix[0][1] = matrix[0][2] = matrix[0][3] = 0;
  matrix[1][0] = matrix[2][0] = matrix[3][0] = 0;
  matrix[1][1] = SQUARE(n[0]) *L_plus +SQUARE(n[1]) *lim1;
  matrix[1][2] = matrix[2][1] = n[0] *n[1] *(L_plus -lim1);
  matrix[2][2] = SQUARE(n[0]) *lim1 +SQUARE(n[1]) *L_plus;
  matrix[1][3] = n[0] *L_minus /ac;
  matrix[2][3] = n[1] *L_minus /ac;
  matrix[3][1] = ac *n[0] *L_minus;
  matrix[3][2] = ac *n[1] *L_minus;
  matrix[3][3] = L_plus;

  return MAX( 0, MAX(lim1,MAX(lim2,lim3)) - FLT_EPSILON ) ? 1 : 0;
}
```

8 Axisymmetric Source Term Evaluation

Functions to evaluate axisymmetric source terms for the Euler equations. The routines compute the distributions to the nodes for a single cell.

```
#include <float.h>     /*  DBL_EPSILON                    */
#include <math.h>      /*  sqrt                           */
#include <stdlib.h>    /*  exit                           */
#include "blas.h"      /*  ddot                           */
#include "rhs.h"       /*  header file for RHS evaluation */
#include "TriMath.h"   /*  TriaveH, cyclic_plus0          */
#include "basics.h"    /*  FOR, Cell, NEQS                */
#include "therm.h"     /*  GAMMA, GAMM1, mu, T            */
```

8.1 Axisymmetric Source Term

Select among various options for forming the axisymmetric source term on a cell, and distribute to the nodes.

```
void axi_source( Cell *cell, double distrib[NEQS][3] )
     /*  distrib is the distribution from the cell to each node  */
     /*  distrib[eqn][node]                                      */
{
  int cnode, tmp1 ;                          /*  temporary indicies         */
  double a[3] = {1,1,1}, b[3] = {1,1,1} ;    /*  vectors for gensource      */
  double distmp[3] ;                         /*  intermediary distribution  */
  double diverg ;                            /*  temp distribution          */
  Node *na, *nb ;                            /*  edge nodes                 */
  Node *anode;
  Coord ell, dGam ;                          /*  edge and face lengths      */
  double ymin, ycount;
  Node2Cell *celln;

  if( traceback > 1 ) printf("In Axi Source, type = %d\n", axi_source_type );
```

Inviscid:

$$B = (0, 0, P, 0)^T$$

Begin by zeroing the distributions.

```
  FOR(tmp1,0,NEQS) FOR(cnode,0,3) distrib[tmp1][cnode] = 0 ;
```

Mass lumped.

$$S_i U_{i_t} \leftarrow B_i \sum \frac{S_T}{3}$$


```
  switch( axi_source_type ){              /*  increasing order of fidelity  */
  case mass_lumped:
    FOR(cnode,0,3){
      FOR(tmp1,0,3)
        a[tmp1] = (cell->geom.connect.nodes[cnode])->vars.primitive[3] ;
      gensource( cell->geom.metric.area, a, b, distmp );
      distrib[2][cnode] = distmp[cnode] ;
    }
    break;
```

Cell-averaged.

$$S_i U_{i_t} \leftarrow \frac{1}{3} \sum S_T \tilde{B}$$

```
  case averaged:
    FOR(tmp1,0,3) a[tmp1] = cell->tilde.primitive[3] ;
    gensource( cell->geom.metric.area, a, b, distrib[2] );
    break;
```

Linear variation of pressure.

```
  case linearP:
    FOR(tmp1,0,3)
      a[tmp1] = (cell->geom.connect.nodes[tmp1])->vars.primitive[3] ;
    gensource( cell->geom.metric.area, a, b, distrib[2] );
    break;
```

Linear variation of parameter vector.

$$P = \frac{\gamma - 1}{\gamma} \left(Z_0 Z_3 - \frac{Z_1^2 + Z_2^2}{2} \right)$$

```
  case linearZ:
    FOR(tmp1,0,3){
      a[tmp1] = (cell->geom.connect.nodes[tmp1])->vars.parameter[0] ;
      b[tmp1] = (cell->geom.connect.nodes[tmp1])->vars.parameter[3] ;
    }
    gensource( cell->geom.metric.area, a, b, distrib[2] );

    FOR(tmp1,0,3){
      a[tmp1] = (cell->geom.connect.nodes[tmp1])->vars.parameter[1] ;
      b[tmp1] = (cell->geom.connect.nodes[tmp1])->vars.parameter[1] ;
    }
    gensource( cell->geom.metric.area, a, b, distmp );
    FOR(cnode,0,3) distrib[2][cnode] -= distmp[cnode] / 2 ;
```

```
    FOR(tmp1,0,3){
      a[tmp1] = (cell->geom.connect.nodes[tmp1])->vars.parameter[2] ;
      b[tmp1] = (cell->geom.connect.nodes[tmp1])->vars.parameter[2] ;
    }
    gensource( cell->geom.metric.area, a, b, distmp );
    FOR(cnode,0,3) distrib[2][cnode] -= distmp[cnode] / 2 ;

    FOR(cnode,0,3) distrib[2][cnode] *= GAMM1 / GAMMA ;
    break;

  default:                                                /*  error trap  */
    printf("Error, bad value in axi_source for axi_source_type =%d\n"
           "Stopping\n", axi_source_type );
    exit(1);
  }
```

Viscous:

$$B = (0,\ 0,\ B_2,\ 0)$$

$$B_2 = \frac{2\mu}{3R_e}\left(\nabla\cdot\vec{V} - 2\frac{v}{y}\right)$$

```
  if( viscous ){
```

Try mass-lumped for the last term,

$$S_i U_{i_t} \leftarrow -\frac{4\mu_i v_i}{3R_e y_i}\sum\frac{S_T}{3}$$

```
    FOR(cnode,0,3){                                   /*  mass lump 1/y term  */
      anode = cell->geom.connect.nodes[cnode];
      ymin = anode->geom.coord[1];
      if( fabs( ymin ) < 10 * FLT_EPSILON ){
ymin = ycount = 0;
CELL_LOOP_AT_NODE( anode ){
  ymin += celln->cell->geom.metric.split_point[1] / 2;
  ++ycount;
}
ymin /= ycount;
      }
      FOR(tmp1,0,3){
        a[tmp1] = mu_from_T( T_from_rho_P(                          /*  mu  */
                anode->vars.primitive[0], anode->vars.primitive[3] ) );
        b[tmp1] = anode->vars.primitive[2] / ymin;                /*  v / y  */
      }
      gensource( cell->geom.metric.area, a, b, distmp );
      distrib[2][cnode] -= 4 * distmp[cnode] / ( 3 * Reynolds ) ;
```



```
    }
```

The remaining term is integrated using the divergence theorem,

$$\frac{2}{3R_e}\int_\Omega \mu\nabla\cdot\vec{V}\,d\omega = \frac{2}{3R_e}\oint_\Gamma \mu\vec{V}\cdot\hat{n}\,d\gamma$$

The integration path goes from edge midpoints through the cell split point, and the normal is approximated as being tanget to the nearest edge, leading to a distribution,

$$\frac{\mu_a+\mu_b}{6R_e}\frac{\Delta\Gamma}{\ell}\left[(u_a+u_b)(x_b-x_a)+(v_a+v_b)(y_b-y_a)\right]$$

$\Delta\Gamma$ is the length from the edge midpoint to the cell split point and ℓ is the length of the edge with endpoints (a,b).

```
    FOR(cnode,0,3){
      na = cell->geom.connect.nodes[cnode] ;
      nb = cell->geom.connect.nodes[cyclic_plus0(cnode)] ;
      ell[0] = ( nb->geom.coord[0] - na->geom.coord[0] ) ;
      ell[1] = ( nb->geom.coord[1] - na->geom.coord[1] ) ;
      dGam[0] = cell->geom.metric.split_containment[0]
              - 0.5 * ( na->geom.coord[0] + nb->geom.coord[0] );
      dGam[1] = cell->geom.metric.split_containment[1]
              - 0.5 * ( na->geom.coord[1] + nb->geom.coord[1] );

      diverg = ( mu_from_T( T_from_rho_P( na->vars.primitive[0],
                                          na->vars.primitive[3] ) ) +
                 mu_from_T( T_from_rho_P( nb->vars.primitive[0],
                                          nb->vars.primitive[3] ) ) )
               / ( 6 * Reynolds );
      diverg *= ( ( na->vars.primitive[1] + nb->vars.primitive[1] ) * ell[0] +
                  ( na->vars.primitive[2] + nb->vars.primitive[2] ) * ell[1] );
      diverg *= dnrm2( NDIMS, dGam, 1 ) / dnrm2( NDIMS, ell, 1 ) ;
      distrib[2][cnode] += diverg ;
      distrib[2][cyclic_plus0(cnode)] -= diverg ;
    }
  }     /*  end viscous  */
}
```

8.2 General Source Term

Compute distribution for a general source term. The integration element is a quadrilateral, obtained as one of the three median dual regions on a triangle. The source term is evaluated as the product of two linear variations. The function returns an array of the distributions to each node, computed as,

$$\int_{\Omega_i} xy\,d\Omega_i = \frac{S_\mathsf{T}}{144}\left[14\bar{x}\bar{y} + 11(x_i\bar{y}+\bar{x}y_i) + 9x_iy_i + \sum_{j=0}^{2} x_jy_j\right]$$

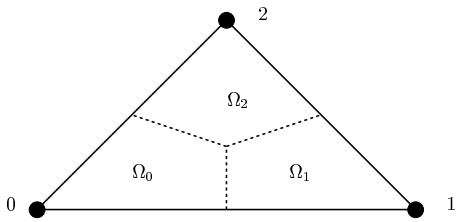


Figure 5: Subdivision of triangular element into three quadrilateral integration areas. Dashed lines are the median-dual mesh. Send the area of the triangle, $S_{\mathsf{T}} = \sum \Omega_i$.

Send the area of the triangle and two arrays of nodal values of the variables for which the moments are desired, and returns a second moment distribution to each node. To get a first moment, send one of the arrays filled with unity. To get the zeroeth moment, send both arrays filled with unity.

```
void gensource( double area,                /*  triangle area            */
                double x[3], double y[3],   /*  nodal values             */
                double distrib[3] )         /*  distributions to return  */
{
  register int i ;
  double distrib_base ;
  double xbar = TriaveH(x), ybar = TriaveH(y) ;

  distrib_base = 14 * xbar * ybar + ddot( 3, x, 1, y, 1 ) ;
  FOR(i,0,3)
    distrib[i] = area / 144 * ( distrib_base
               + ( x[i] * ybar + xbar * y[i] ) * 11 + x[i] * y[i] * 9 ) ;
}
```

8.3 Flux Source for Fluctuation Splitting

Assemble an additional axisymmetric source term for fluctuation splitting, defined on a cell. The source term is,

$$-\int_{\Omega} \mathbf{F}^{\vec{v}} \, d\Omega$$

with,

$$\mathbf{F}^{\vec{v}} = \begin{pmatrix} Z_0 Z_2 \\ Z_1 Z_2 \\ Z_2^2 + \frac{\gamma-1}{\gamma}\left(Z_0 Z_3 - \frac{Z_1^2+Z_2^2}{2}\right) \\ Z_2 Z_3 \end{pmatrix}$$


```
void FS_flux_source( Cell *cell, double distrib[NEQS][3] )
{
  double S = cell->geom.metric.area ;                        /*  cell area  */
  double a[3], b[3] ;                /*  nodal values of parameter vector  */
  Node *n[3] ;                    /*  pointers to nodes defining the cell  */
  int i ;                                                  /*  dummy indicie  */
  double distmp[3] ;                          /*  temporary distributions  */

  FOR(i,0,3) n[i] = cell->geom.connect.nodes[i] ;

  FOR(i,0,3){                                                /*  continuity  */
    a[i] = n[i]->vars.parameter[0] ;
    b[i] = n[i]->vars.parameter[2] ;
  }
  gensource( S, a, b, distrib[0] );

  FOR(i,0,3) a[i] = n[i]->vars.parameter[1] ;               /*  x-momentum  */
  gensource( S, a, b, distrib[1] );

  FOR(i,0,3){                                               /*  y-momentum  */
    a[i] = n[i]->vars.parameter[0] ;
    b[i] = n[i]->vars.parameter[3] ;
  }
  gensource( S, a, b, distrib[2] );
  FOR(i,0,3){
    a[i] = n[i]->vars.parameter[1] ;
    b[i] = n[i]->vars.parameter[1] ;
  }
  gensource( S, a, b, distmp );
  FOR(i,0,3) distrib[2][i] -= distmp[i] / 2 ;
  FOR(i,0,3){
    a[i] = n[i]->vars.parameter[2] ;
    b[i] = n[i]->vars.parameter[2] ;
  }
  gensource( S, a, b, distmp );
  FOR(i,0,3) distrib[2][i] = distmp[i] + GAMM1 / GAMMA *
                          ( distrib[2][i] - distmp[i] / 2 );

  FOR(i,0,3) a[i] = n[i]->vars.parameter[3] ;               /*  energy      */
  gensource( S, a, b, distrib[3] );
}
```

9 Governing Equations

A collection of functions for evaluating terms in the governing equations, such as the flux functions.

```
#include <stdio.h>       /*  printf              */
#include <math.h>        /*  sin, cos            */
#include <stdlib.h>      /*  exit                */
#include "basics.h"      /*  SolVec, NEQS, NDIMS  */
#include "transform.h"   /*  U->Z, Z->V          */
#include "therm.h"       /*  gamma               */
```

9.1 Conserved Euler Flux

Given a vector of conserved variables, return the inviscid flux.

$$F = \begin{bmatrix} \rho u & \rho v \\ \rho u^2 + P & \rho uv \\ \rho uv & \rho v^2 + P \\ \rho uH & \rho vH \end{bmatrix}$$

The conserved, primitive, and parameter variables are all used to simplify the expressions as (base 0),

$$F = \begin{bmatrix} U_1 & U_2 \\ U_1 V_1 + V_3 & U_1 V_2 \\ U_2 V_1 & U_2 V_2 + V_3 \\ Z_1 Z_3 & Z_2 Z_3 \end{bmatrix}$$

```
void Euler_flux_conserved( SolVec U, double F[NEQS][NDIMS] )
{
  SolVec V, Z ;   /*  primitive and parameter vectors  */

  conserved_to_parameter( U, Z );
  parameter_to_primitive( Z, V );

  F[0][0] = U[1]               ;
  F[0][1] = U[2]               ;
  F[1][0] = U[1] * V[1] + V[3] ;
  F[1][1] = U[1] * V[2]        ;
  F[2][0] = U[2] * V[1]        ;
  F[2][1] = U[2] * V[2] + V[3] ;
  F[3][0] = Z[1] * Z[3]        ;
  F[3][1] = Z[2] * Z[3]        ;
}
```


9.2 Scalar Flux

Evaluate the flux function for various scalar equations.

```
void scalar_flux( SolVec U, double F[NEQS][NDIMS] )
{
  int eq, dm ;   /*  dummy indicies  */
```

Choose the scalar flux function here, hardwired.

```
  enum { uniform, circular, smith_hutton, burger } option ;
  option = uniform ;

  FOR(eq,0,NEQS) FOR(dm,0,NDIMS) F[eq][dm] = 0 ;    /* zero the flux to start */
```

Uniform advection,

$$F = \lambda U, \quad \tan(\alpha) = \frac{b}{a}$$

```
  switch( option ){
  case uniform:
    F[0][0] = cos(AOA) * U[0] ;
    F[0][1] = sin(AOA) * U[0] ;
    break;
```

Burgers equation,

$$F = (\frac{u^2}{2}, u)$$

```
  case burger:
    F[0][0] = SQUARE( U[0] ) / 2 ;
    F[0][1] = U[0] ;
    break;

  default:
    printf("Bad value for scalar flux = %d\nStopping\n", option );
    exit(1);
    break;
  }
}
```

9.3 F_Z

Form the two-dimensional Euler flux Jacobian in terms of the parameter vector.

$$F_Z^x = \begin{bmatrix} Z_1 & Z_0 & 0 & 0 \\ \frac{\gamma-1}{\gamma} Z_3 & \frac{\gamma+1}{\gamma} Z_1 & -\frac{\gamma-1}{\gamma} Z_2 & \frac{\gamma-1}{\gamma} Z_0 \\ 0 & Z_2 & Z_1 & 0 \\ 0 & Z_3 & 0 & Z_1 \end{bmatrix}$$

$$F_Z^y = \begin{bmatrix} Z_2 & 0 & Z_0 & 0 \\ 0 & Z_2 & Z_1 & 0 \\ \frac{\gamma-1}{\gamma} Z_3 & -\frac{\gamma-1}{\gamma} Z_1 & \frac{\gamma+1}{\gamma} Z_2 & \frac{\gamma-1}{\gamma} Z_0 \\ 0 & 0 & Z_3 & Z_2 \end{bmatrix}$$

```
void dF_dZ( SolVec z, double df_dz[2][NEQS][NEQS] )
{
  double f1 = GAMM1 / GAMMA ;
  double f2 = GAMP1 / GAMMA ;

  df_dz[0][0][0] = z[1] ;
  df_dz[0][0][1] = z[0] ;
  df_dz[0][0][2] = 0 ;
  df_dz[0][0][3] = 0 ;
  df_dz[0][1][0] = z[3] * f1 ;
  df_dz[0][1][1] = z[1] * f2 ;
  df_dz[0][1][2] =-z[2] * f1 ;
  df_dz[0][1][3] = z[0] * f1 ;
  df_dz[0][2][0] = 0 ;
  df_dz[0][2][1] = z[2] ;
  df_dz[0][2][2] = z[1] ;
  df_dz[0][2][3] = 0 ;
  df_dz[0][3][0] = 0 ;
  df_dz[0][3][1] = z[3] ;
  df_dz[0][3][2] = 0 ;
  df_dz[0][3][3] = z[1] ;

  df_dz[1][0][0] = z[2] ;
  df_dz[1][0][1] = 0 ;
  df_dz[1][0][2] = z[0] ;
  df_dz[1][0][3] = 0 ;
  df_dz[1][1][0] = 0 ;
  df_dz[1][1][1] = z[2] ;
  df_dz[1][1][2] = z[1] ;
  df_dz[1][1][3] = 0 ;
  df_dz[1][2][0] = z[3] * f1 ;
  df_dz[1][2][1] =-z[1] * f1 ;
  df_dz[1][2][2] = z[2] * f2 ;
  df_dz[1][2][3] = z[0] * f1 ;
  df_dz[1][3][0] = 0 ;
```



```
  df_dz[1][3][1] = 0 ;
  df_dz[1][3][2] = z[3] ;
  df_dz[1][3][3] = z[2] ;
}
```

10 Habashi Adaption

Perform anisotropic local adaption based on edge error estimates. The edge error estimates are,

$$|E_r| \sim |(\nabla U_1 - \nabla U_0) \cdot \vec{r}_{01}|$$

The permissible adaptions are: edge swapping, point insertion, point deletion, and node movement.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <math.h>      /*  fabs, sqrt, acos, log10  */
#include <float.h>
#include "basics.h"
#include "adaption.h"
#include "rhs.h"       /*  node_gradients    */
#include "blas.h"      /*  dnrm2             */
#include "TriMath.h"
#include "bc.h"
#include "transform.h" /*  U2V, U2Z          */
```

10.1 Set-up for adaption

Build a list of edges. Then compute nodal gradients. Finally, assign the edge error estimates.

```
void edge_errors( void ){
  int i;
  Cell *cell;
  Edge *edge;
  Node *end_nodes[2];
  double max_error=0;

  CELL_LOOP_FORWARD FOR(i,0,3){
    end_nodes[0] = cell->geom.connect.nodes[i] ;
    end_nodes[1] = cell->geom.connect.nodes[cyclic_plus0(i)] ;
    build_edges( end_nodes, cell );
  }
  printf("Grid has %u edges\n", edges );

  node_gradients( 0 );

  adapt_target = log10( adapt_target );
  EDGE_LOOP_FORWARD{
    if( mesh_adapt==delete_FM || mesh_adapt==move_FM || mesh_adapt==add_FM
        || mesh_adapt==edge_swap_FM )
      edge->error = error_fluc( edge->cell_left, edge->cell_right );
    else edge->error = error_estimate( edge->node, edge->r );
```



```
    FOR(i,0,2) (edge->node[i])->adapt.error += edge->error;
    max_error = MAX( max_error, edge->error );
  }
  max_error     = log10(max_error);
  swap_thresh   = pow(10, MAX(max_error-2, adapt_target));
  add_thresh    = swap_thresh;
  move_thresh   = swap_thresh;
  delete_thresh = pow(10, adapt_target-3);
  if( traceback >= 1 )
    printf("max error=%.2e, swap=%.2e, add=%.2e, move=%.2e, delete=%.2e\n",
   max_error, swap_thresh, add_thresh, move_thresh, delete_thresh);
}
```

10.2 Build the linked list of edges

Given an edge defined by two nodes, search the list of existing edges to see if this is a duplicate edge. If not, then create a new edge.

```
void build_edges( Node *end_nodes[2], Cell *cell ){
  Edge *edge, *this_edge = NULL ;
  int i;

  EDGE_LOOP_FORWARD{
    if( edge->node[0] == end_nodes[1] && edge->node[1] == end_nodes[0] ){
      edge->cell_right = cell ;
      this_edge = edge ;
    }
  }

  if( !this_edge ){
    this_edge = add_edge();
    this_edge->cell_left  = cell ;
    FOR(i,0,2){
      this_edge->node[i] = end_nodes[i] ;
      this_edge->r[i]    = end_nodes[1]->geom.coord[i] -
                           end_nodes[0]->geom.coord[i] ;
      hook_from_node_to_edge( end_nodes[i], this_edge );
    }
  }
}
```

10.3 Swap edges based on error estimates

Loop over entire domain and try to swap desired edges based on an edge error estimate criteria.

```
void global_error_swap( void ){
  Edge *edge ;
```

Here we simply loop over all edges in the domain. If desired, the edges could be sorted or subjected to a tolerance criteria before trying to swap.

```
  EDGE_LOOP_FORWARD if( querry_swap( edge ) ) error_swap( edge );
  swap_edge( NULL );
}
```

10.4 Can this edge be physically swapped?

The edge can be swapped if it is not a boundary edge and if the swapped condition maintains positive cell areas. Disallow swap if one of the new areas would be more than 90 percent smaller than a current area. Prevent the creation of angles greater than 179 degrees.

```
int querry_swap( Edge *edge ){
  double area_left, area_right ;
  double new_area_a, new_area_b ;
  double min_area_old, min_area_new ;
  Node *node[3];
  Node *node_left=NULL, *node_right=NULL;
  Node *node_0=edge->node[0], *node_1=edge->node[1];
  int i;
  double pi = acos( -1. );
  double max_angle = pi * 175. / 180.;
  Coord a, b, c;
  double angle_a, angle_b, new_max_angle;

  if( !edge->cell_right ) return 0;

  area_left  = edge->cell_left->geom.metric.area ;
  area_right = edge->cell_right->geom.metric.area ;
  min_area_old = 0.25 * MIN( area_left, area_right ); /* min allowable area */

  FOR(i,0,3){
    if( edge->node[0] == edge->cell_left->geom.connect.nodes[i] )
      node_left = edge->cell_left->geom.connect.nodes[cyclic_minus0(i)] ;
    if( edge->node[1] == edge->cell_right->geom.connect.nodes[i] )
      node_right = edge->cell_right->geom.connect.nodes[cyclic_minus0(i)] ;
  }
  node[0] = node_left;
  node[1] = edge->node[0];
  node[2] = node_right;
  new_area_a = tri_cell_area( node );
  node[0] = node_right;
  node[1] = edge->node[1];
```



```
  node[2] = node_left;
  new_area_b = tri_cell_area( node );
  min_area_new = MIN( new_area_a, new_area_b );
  if( min_area_new < min_area_old ) return 0;

  FOR(i,0,NDIMS) a[i] = node_right->geom.coord[i] - node_0->geom.coord[i];
  FOR(i,0,NDIMS) b[i] = node_left->geom.coord[i] - node_0->geom.coord[i];
  FOR(i,0,NDIMS) c[i] = node_right->geom.coord[i] - node_left->geom.coord[i];
  angle_a = acos( (VEL2(a) + VEL2(b) - VEL2(c)) / (2 * VEL(a) * VEL(b)) );
  FOR(i,0,NDIMS) a[i] = node_right->geom.coord[i] - node_1->geom.coord[i];
  FOR(i,0,NDIMS) b[i] = node_left->geom.coord[i] - node_1->geom.coord[i];
  angle_b = acos( (VEL2(a) + VEL2(b) - VEL2(c)) / (2 * VEL(a) * VEL(b)) );
  new_max_angle = MAX( angle_a, angle_b );
  if( new_max_angle > max_angle ) return 0;

  return 1;
}
```

10.5 Swap an edge based on error estimate

Decide whether or not to swap an edge based on an edge error estimate. The edge error must exceed the threshold value or do not bother with it.

```
void error_swap( Edge *edge ){
  Node *node_left=NULL, *node_right=NULL, *node[2] ;
  double swapped_error ;
  Coord r01 ;
  int i;
  Cell testCellLeft, testCellRight;

  if( edge->error > swap_thresh ){
    FOR(i,0,3){
      if( edge->node[0] == edge->cell_left->geom.connect.nodes[i] )
        node_left = edge->cell_left->geom.connect.nodes[cyclic_minus0(i)] ;
      if( edge->node[1] == edge->cell_right->geom.connect.nodes[i] )
        node_right = edge->cell_right->geom.connect.nodes[cyclic_minus0(i)] ;
    }
    node[0] = node_left;
    node[1] = node_right;
    testCellLeft.geom.connect.nodes[0] = node_left;
    testCellLeft.geom.connect.nodes[1] = node_right;
    testCellLeft.geom.connect.nodes[2] = edge->node[1];
    testCellRight.geom.connect.nodes[0] = node_right;
    testCellRight.geom.connect.nodes[1] = node_left;
    testCellRight.geom.connect.nodes[2] = edge->node[0];
    TriaveV(NDIMS, testCellLeft.geom.connect.nodes[0]->geom.coord,
                   testCellLeft.geom.connect.nodes[1]->geom.coord,
                   testCellLeft.geom.connect.nodes[2]->geom.coord,
```

```
                  testCellLeft.geom.metric.split_point);
    TriaveV(NDIMS, testCellRight.geom.connect.nodes[0]->geom.coord,
                  testCellRight.geom.connect.nodes[1]->geom.coord,
                  testCellRight.geom.connect.nodes[2]->geom.coord,
                  testCellRight.geom.metric.split_point);
    FOR(i,0,2){
      testCellLeft.geom.metric.split_containment[i] =
        testCellLeft.geom.metric.split_point[i];
      testCellRight.geom.metric.split_containment[i] =
        testCellRight.geom.metric.split_point[i];
    }
    testCellLeft.geom.metric.area =
      tri_cell_area(testCellLeft.geom.connect.nodes);
    testCellRight.geom.metric.area =
      tri_cell_area(testCellRight.geom.connect.nodes);
    testCellLeft.links.previous_cell = NULL;
    testCellRight.links.previous_cell = NULL;
    testCellLeft.links.next_cell = NULL;
    testCellRight.links.next_cell = NULL;
    Roe_average_cell(&testCellLeft);
    Roe_average_cell(&testCellRight);

    FOR(i,0,2)
      r01[i] = edge->node[1]->geom.coord[i] - edge->node[0]->geom.coord[i];
    if( mesh_adapt == delete_FM || mesh_adapt == move_FM ||
        mesh_adapt == add_FM || mesh_adapt == edge_swap_FM )
      swapped_error = error_fluc(&testCellLeft, &testCellRight);
    else
      swapped_error = error_estimate(node, r01);

    if(swapped_error < edge->error) swap_edge(edge);
  }
}
```

10.6 Edge error estimate

Evaluate the edge error estimate as,

$$|E_r| \sim ||(\nabla U_1 - \nabla U_0) \cdot \vec{r}_{01}||$$

```
double error_estimate( Node *node[2], Coord r01 ){
  SolVec Er ;
  int eq ;

  FOR(eq,0,NEQS)
    Er[eq] = (node[1]->fv.gradx[eq] - node[0]->fv.gradx[eq]) * r01[0] +
             (node[1]->fv.grady[eq] - node[0]->fv.grady[eq]) * r01[1] ;
  return dnrm2( NEQS, Er, 1 );
```



```
}
```

10.7 Swap a mesh edge

If the edge is NULL, then write a diagnostic message. Otherwise, rotate the edge clock-wise.

```
void swap_edge( Edge *edge_to_swap ){
  Edge *edge;
  Node *node_left=NULL, *node_right=NULL, *node0, *node1;
  Cell *cell_left, *cell_right;
  int i;
  static unsigned int swapped_edges = 0;

  if( edge_to_swap ){
    ++swapped_edges;
    node0 = edge_to_swap->node[0];
    node1 = edge_to_swap->node[1];
    cell_left = edge_to_swap->cell_left;
    cell_right = edge_to_swap->cell_right;

    FOR(i,0,3){
      if( cell_left->geom.connect.nodes[i] == node0 )
        node_left = cell_left->geom.connect.nodes[cyclic_minus0(i)] ;
      if( cell_right->geom.connect.nodes[i] == node1 )
        node_right = cell_right->geom.connect.nodes[cyclic_minus0(i)] ;
    }
```

Re-define the current edge.

```
    edge_to_swap->node[0] = node_left;
    edge_to_swap->node[1] = node_right;
    FOR(i,0,NDIMS)
      edge_to_swap->r[i] = node_right->geom.coord[i] - node_left->geom.coord[i];
```

Re-attach the edges connected to the redefined cells.

```
    EDGE_LOOP_FORWARD if( edge != edge_to_swap ){
      if( edge->cell_left == cell_left && edge->node[1] == node0 )
        edge->cell_left = cell_right ;
      if( edge->cell_right == cell_left && edge->node[0] == node0 )
        edge->cell_right = cell_right ;
      if( edge->cell_left == cell_right && edge->node[1] == node1 )
        edge->cell_left = cell_left ;
      if( edge->cell_right == cell_right && edge->node[0] == node1 )
        edge->cell_right = cell_left ;
```

```
}
```

Update the cells to left and right of the edge.

```
FOR(i,0,3){
  if( cell_left->geom.connect.nodes[i] == node0 )
    cell_left->geom.connect.nodes[i] = node_right ;
  if( cell_right->geom.connect.nodes[i] == node1 )
    cell_right->geom.connect.nodes[i] = node_left ;
}
cell_left->geom.metric.area = tri_cell_area(cell_left->geom.connect.nodes);
cell_right->geom.metric.area =tri_cell_area(cell_right->geom.connect.nodes);
TriaveV( 2, (cell_left->geom.connect.nodes[0])->geom.coord,
            (cell_left->geom.connect.nodes[1])->geom.coord,
            (cell_left->geom.connect.nodes[2])->geom.coord,
             cell_left->geom.metric.split_point );
TriaveV( 2, (cell_right->geom.connect.nodes[0])->geom.coord,
            (cell_right->geom.connect.nodes[1])->geom.coord,
            (cell_right->geom.connect.nodes[2])->geom.coord,
             cell_right->geom.metric.split_point );
if( viscous==2 || viscous==4 ){
  containment_dual( cell_left );
  containment_dual( cell_right );
}
else FOR(i,0,NDIMS){
  cell_left->geom.metric.split_containment[i] =
    cell_left->geom.metric.split_point[i] ;
  cell_right->geom.metric.split_containment[i] =
    cell_right->geom.metric.split_point[i] ;
}
Roe_average_cell( cell_left );
Roe_average_cell( cell_right );
```

Update all four affected nodes.

```
break_cell_from_node( node0, cell_left );
break_edge_from_node( node0, edge_to_swap );
break_cell_from_node( node1, cell_right );
break_edge_from_node( node1, edge_to_swap );
hook_from_node_to_cell( node_left, cell_right );
hook_from_node_to_edge( node_left, edge_to_swap );
hook_from_node_to_cell( node_right, cell_left );
hook_from_node_to_edge( node_right, edge_to_swap );
node0->geom.dual_area = median_dual_one( node0 );
node1->geom.dual_area = median_dual_one( node1 );
node_left->geom.dual_area  = median_dual_one( node_left );
node_right->geom.dual_area = median_dual_one( node_right );
```


Update the edge and node errors.

```
    node0->adapt.error -= edge_to_swap->error;
    node1->adapt.error -= edge_to_swap->error;
    if( mesh_adapt==delete_FM || mesh_adapt==move_FM || mesh_adapt==add_FM
        || mesh_adapt==edge_swap_FM )
      edge_to_swap->error = error_fluc(cell_left, cell_right);
    else
      edge_to_swap->error = error_estimate(edge_to_swap->node, edge_to_swap->r);
    node_right->adapt.error += edge_to_swap->error;
    node_left->adapt.error += edge_to_swap->error;

/*     if( swapped_edges % 100 == 0 ) write_tecplot_0(); */
  }                                                  /*  if( edge_to_swap )  */

  else printf("%u edges have been swapped\n", swapped_edges );
}
```

10.8 Delete points based on error estimate

```
void global_error_delete( void ){
  Node *node;

  NODE_LOOP_BACKWARD if( node->adapt.error < delete_thresh ) point_delete(node);
  point_delete( NULL );
}
```

10.9 Try to delete a node

Given a node, try to reduce the connectivity of the node so that only three edges are connected to the node. Then delete the node. If NULL is sent, print diagnostic message.

```
void point_delete( Node *node ){
  static unsigned int deleted_nodes = 0;
  static unsigned int tried_but_failed = 0;
  const unsigned int plot_skip = 1000;
  int swaps, hooked_edges;
  Node2Edge *edgen;
  enum { false, true } success = false;

  if( node ){
    if( traceback >=2 )
      printf(" try to delete node with error %.2e:\n", node->adapt.error);
    do{
      swaps = hooked_edges = 0;
```

```
      EDGE_LOOP_AT_NODE( node ){
        ++hooked_edges;
        if( querry_swap( edgen->edge ) ){
          swap_edge( edgen->edge );
          ++swaps;
          break;
        }
      }
    }
    while( swaps );

    if( node->bc != interior ){
      if( bound_point_delete( node ) ) success = true;
      else ++tried_but_failed;
    }
    else if( hooked_edges == 3 ){
      tri_point_delete( node );
      success = true;
    }
    else if( hooked_edges == 4 ){
      if(quad_point_delete(node)==0) success = true;
      else ++tried_but_failed;
    }
    else ++tried_but_failed;
    if( success == true ){
      ++deleted_nodes;
      if( deleted_nodes % plot_skip == 0 ) write_tecplot_0();
    }
    if( traceback >= 2 ) printf(" success=%d, gone=%u\n",success,deleted_nodes);
  }

  else{
    printf("Deleted %u nodes\n", deleted_nodes );
    printf("Tried but failed to delete  %u nodes\n", tried_but_failed );
  }
}
```

10.10 Delete point connected to three cells

Given an interior node connected to three cells, delete the node and associated edges, replacing the three cells with the single outer cell.

```
void tri_point_delete( Node *node ){
  Cell *cA, *cB, *cC;
  Node *nAB, *nAC, *nBC;
  Edge *eAB, *eAC, *eBC, *edge;
  Node2Edge *pn2e;
  int i;
```


Identify the three edges, nodes, and cells hooked to this node.

```
/*                nBC                                          *
 *                 /|\    eAB separates cells A and B          *
 *                / | \   eAC separates cells A and C          *
 *               /  |  \  eBC separates cells B and C          *
 *              / cC| cB\                                      *
 *             /        \                                      *
 *            /    cA     \                                     *
 *       nAC  /____________\ nAB                                */

pn2e = node->geom.edgecon;
eAB = pn2e->edge;
if( eAB->node[0] == node ){
  nAB = eAB->node[1];
  cA  = eAB->cell_right;
  cB  = eAB->cell_left;
}
else{
  nAB = eAB->node[0];
  cA  = eAB->cell_left;
  cB  = eAB->cell_right;
}
pn2e = pn2e->next;
if( pn2e->edge->cell_left == cB ){
  eBC = pn2e->edge;
  nBC = eBC->node[0];
  cC  = eBC->cell_right;
  eAC = pn2e->next->edge;
  if( eAC->node[0] == node ) nAC = eAC->node[1];
  else                       nAC = eAC->node[0];
}
else if( pn2e->edge->cell_right == cB ){
  eBC = pn2e->edge;
  nBC = eBC->node[1];
  cC  = eBC->cell_left;
  eAC = pn2e->next->edge;
  if( eAC->node[0] == node ) nAC = eAC->node[1];
  else                       nAC = eAC->node[0];
}
else{
  eAC = pn2e->edge;
  if( eAC->node[0] == node ){
    nAC = eAC->node[1];
    cC  = eAC->cell_right;
  }
  else{
    nAC = eAC->node[0];
```

```
    cC  = eAC->cell_left;
  }
  eBC = pn2e->next->edge;
  if( eBC->node[0] == node ) nBC = eBC->node[1];
  else                       nBC = eBC->node[0];
}
```

Disconnect the vanishing edges and cells from the nodes. Re-attach the remaining cell to the correct node.

```
break_edge_from_node( nAB, eAB );
break_edge_from_node( nAC, eAC );
break_edge_from_node( nBC, eBC );
break_cell_from_node( nAB, cB );
break_cell_from_node( nAC, cC );
break_cell_from_node( nBC, cB );
break_cell_from_node( nBC, cC );
hook_from_node_to_cell( nBC, cA );
FOR(i,0,3)
  if( cA->geom.connect.nodes[i] == node ) cA->geom.connect.nodes[i] = nBC;
```

Recompute metric terms.

```
cA->geom.metric.area += cB->geom.metric.area + cC->geom.metric.area;
TriaveV( 2, (cA->geom.connect.nodes[0])->geom.coord,
            (cA->geom.connect.nodes[1])->geom.coord,
            (cA->geom.connect.nodes[2])->geom.coord,
             cA->geom.metric.split_point );
if( viscous == 2 || viscous == 4 ) containment_dual( cA );
else FOR(i,0,NDIMS)
  cA->geom.metric.split_containment[i] = cA->geom.metric.split_point[i];

nAB->geom.dual_area = median_dual_one( nAB );
nAC->geom.dual_area = median_dual_one( nAC );
nBC->geom.dual_area = median_dual_one( nBC );
```

Connect the remaining edges connected to cells B and C to cell A.

```
EDGE_LOOP_FORWARD if( edge != eAB && edge != eAC && edge != eBC ){
  if( edge->cell_right==cB || edge->cell_right==cC ) edge->cell_right = cA;
  else if( edge->cell_left==cB || edge->cell_left==cC ) edge->cell_left = cA;
}
```

Kill the node, two cells, and three edges.


```
  kill_node( node );
  kill_cell( cB );
  kill_cell( cC );
  kill_edge( eAB );
  kill_edge( eAC );
}
```

10.11 Delete point connected to four cells

Given an interior node connected to four cells, delete the node, two of the cells, and three edges.

```
int quad_point_delete( Node *node ){
  Node *nAB, *nBC=NULL, *nCD, *nDA=NULL, *ntemp[3];
  Cell *cA, *cB, *cC, *cD;
  Edge *eAB, *eBC=NULL, *eCD, *eDA=NULL, *etemp[4], *edge;
  Node2Edge *pn2e;
  int i;
  typedef unsigned int UINT;
```

Identify the geometry. Edges eAB and eCD have to be the most parallel.

```
  /*            nDA                         nDA         *
   *        cA   eDA   cD                    cA          *
   *  nAB   eAB  node  eCD  nCD  -->  nAB    eAB   nCD   *
   *        cB   eBC   cC                    cB          *
   *             nBC                         nBC         */

  pn2e = node->geom.edgecon;
  FOR(i,0,4){
    etemp[i] = pn2e->edge;
    pn2e = pn2e->next;
  }
  if( etemp[0]->cell_left  != etemp[1]->cell_left  &&
      etemp[0]->cell_left  != etemp[1]->cell_right &&
      etemp[0]->cell_right != etemp[1]->cell_left  &&
      etemp[0]->cell_right != etemp[1]->cell_right )
    if(norm_cross_edges(etemp[0],etemp[1])<norm_cross_edges(etemp[2],etemp[3])){
      eAB = etemp[0];
      eCD = etemp[1];
    }
    else{
      eAB = etemp[2];
      eCD = etemp[3];
    }
  else if( etemp[0]->cell_left  != etemp[2]->cell_left  &&
```

```
 etemp[0]->cell_left  != etemp[2]->cell_right &&
 etemp[0]->cell_right != etemp[2]->cell_left  &&
 etemp[0]->cell_right != etemp[2]->cell_right )
  if(norm_cross_edges(etemp[0],etemp[2])<norm_cross_edges(etemp[1],etemp[3])){
    eAB = etemp[0];
    eCD = etemp[2];
  }
  else{
    eAB = etemp[1];
    eCD = etemp[3];
  }
else{
  if(norm_cross_edges(etemp[0],etemp[3])<norm_cross_edges(etemp[1],etemp[2])){
    eAB = etemp[0];
    eCD = etemp[3];
  }
  else{
    eAB = etemp[1];
    eCD = etemp[2];
  }
}

if( eAB->node[0] == node ){
  nAB = eAB->node[1];
  cA = eAB->cell_right;
  cB = eAB->cell_left;
}
else{
  nAB = eAB->node[0];
  cB = eAB->cell_right;
  cA = eAB->cell_left;
}
if( eCD->node[0] == node ){
  nCD = eCD->node[1];
  cC = eCD->cell_right;
  cD = eCD->cell_left;
}
else{
  nCD = eCD->node[0];
  cD = eCD->cell_right;
  cC = eCD->cell_left;
}

FOR(i,0,4){
  if( etemp[i] != eAB &&
    ( etemp[i]->cell_left == cA || etemp[i]->cell_right == cA ) ){
    eDA = etemp[i];
    if( eDA->node[0] == node ) nDA = eDA->node[1];
    else nDA = eDA->node[0];
  }
```



```
  if( etemp[i] != eAB &&
    ( etemp[i]->cell_left == cB || etemp[i]->cell_right == cB ) ){
    eBC = etemp[i];
    if( eBC->node[0] == node ) nBC = eBC->node[1];
    else nBC = eBC->node[0];
  }
}
```

Before actually deleting, double check that no negative areas will be created. Abort if they will be.

```
ntemp[0] = nAB;
ntemp[1] = nCD;
ntemp[2] = nDA;
if(tri_cell_area(ntemp)<=0) return 1;
ntemp[0] = nCD;
ntemp[1] = nAB;
ntemp[2] = nBC;
if(tri_cell_area(ntemp)<=0) return 1;
```

Disconnect the vanishing edges and cells from the nodes. Re-attach the remaining cells and edges to the correct node.

```
break_edge_from_node( nBC, eBC );
break_edge_from_node( nCD, eCD );
break_edge_from_node( nDA, eDA );
hook_from_node_to_edge( nCD, eAB );

if( eAB->node[0] == node ) eAB->node[0] = nCD;
else eAB->node[1] = nCD;
FOR(i,0,NDIMS)
  eAB->r[i] = eAB->node[1]->geom.coord[i] - eAB->node[0]->geom.coord[i];

break_cell_from_node( nBC, cC );
break_cell_from_node( nCD, cC );
break_cell_from_node( nCD, cD );
break_cell_from_node( nDA, cD );
hook_from_node_to_cell( nCD, cA );
hook_from_node_to_cell( nCD, cB );

FOR(i,0,3){
  if( cA->geom.connect.nodes[i] == node ) cA->geom.connect.nodes[i] = nCD;
  if( cB->geom.connect.nodes[i] == node ) cB->geom.connect.nodes[i] = nCD;
}
```

Recompute metric terms.

```
if((cA->geom.metric.area=tri_cell_area(cA->geom.connect.nodes))<=0)
  printf("Created negative area for cell %.3e\n", cA->geom.metric.area);
if((cB->geom.metric.area=tri_cell_area(cB->geom.connect.nodes))<=0)
  printf("Created negative area for cell %.3e\n", cB->geom.metric.area);
TriaveV( 2, (cA->geom.connect.nodes[0])->geom.coord,
            (cA->geom.connect.nodes[1])->geom.coord,
            (cA->geom.connect.nodes[2])->geom.coord,
             cA->geom.metric.split_point );
TriaveV( 2, (cB->geom.connect.nodes[0])->geom.coord,
            (cB->geom.connect.nodes[1])->geom.coord,
            (cB->geom.connect.nodes[2])->geom.coord,
             cB->geom.metric.split_point );
if( viscous == 2 || viscous == 4 ){
  containment_dual( cA );
  containment_dual( cB );
}
else FOR(i,0,NDIMS){
  cA->geom.metric.split_containment[i] = cA->geom.metric.split_point[i];
  cB->geom.metric.split_containment[i] = cB->geom.metric.split_point[i];
}

nAB->geom.dual_area = median_dual_one( nAB );
nBC->geom.dual_area = median_dual_one( nBC );
nCD->geom.dual_area = median_dual_one( nCD );
nDA->geom.dual_area = median_dual_one( nDA );
```

Correct edges that used to point to cells C and D to point to cells B and A, respectively.

```
EDGE_LOOP_FORWARD{
  if( edge != eBC && edge != eCD && edge != eDA ){
    if( edge->cell_right == cC ) edge->cell_right = cB;
    else if( edge->cell_left == cC ) edge->cell_left = cB;
    if( edge->cell_right == cD ) edge->cell_right = cA;
    else if( edge->cell_left == cD ) edge->cell_left = cA;
  }
}
```

Kill the node, two cells, and three edges.

```
kill_node( node );
kill_cell( cC );
kill_cell( cD );
kill_edge( eBC );
kill_edge( eCD );
kill_edge( eDA );
return 0;
```



```
}
```

10.12 Delete a boundary node

Delete a boundary point that has exactly three edges connected to it, but only if the boundary edges are parallel at this point. Delete the node, one cell, and two edges.

```
int bound_point_delete( Node *node ){
  Node2Edge *edgen;
  Edge *eA=NULL, *eB=NULL, *eAB=NULL, *edge;
  Cell *cA, *cB;
  Node *nA, *nB, *nAB;
  Bound_edge *bedge;
  double crossprod;
  int i, edge_count = 0;
```

Identify the geometry.

```
  /*       nAB ^                   nAB  ^                 *
   *          /|\                     / \                 *
   *         / | \                   /   \                *
   *        / eAB \        -->       /     \               *
   *       /   |   \                /   cA  \              *
   *      / cA | cB \              /         \             *
   * nA  /_____|_____\ nB      nA /___________\ nB         *
   *      eA   n   eB                 eA                  */

  EDGE_LOOP_AT_NODE( node ){
    ++edge_count;
    if( edgen->edge->cell_right == NULL && edgen->edge->node[1] == node )
      eA = edgen->edge;
    else if( edgen->edge->cell_right == NULL && edgen->edge->node[0] == node )
      eB = edgen->edge;
    else eAB = edgen->edge;
  }
  if( edge_count != 3 ) return 0;

  cA = eA->cell_left;
  cB = eB->cell_left;
  nA = eA->node[0];
  nB = eB->node[1];
  if( eAB->node[0] == node ) nAB = eAB->node[1];
  else nAB = eAB->node[0];
```

Edges eA and eB must be (nearly) parallel.

```
  crossprod = cross_edges( eA, eB ) / ( VEL(eA->r) * VEL(eB->r) );
  if( fabs( crossprod ) > .05 ) return 0;
```

Disconnect vanishing cell and edges, reattaching remaing members.

```
  break_edge_from_node( nAB, eAB );
  break_edge_from_node( nB, eB );
  hook_from_node_to_edge( nB, eA );
  nA->geom.bound_face -= VEL(eA->r) / 2;
  nB->geom.bound_face -= VEL(eB->r) / 2;
  eA->node[1] = nB;
  BEDGE_LOOP_FORWARD if( bedge->node[0] == node ) bedge->node[0] = nB;
  FOR(i,0,NDIMS)
    eA->r[i] = eA->node[1]->geom.coord[i] - eA->node[0]->geom.coord[i];
  nA->geom.bound_face += VEL(eA->r) / 2;
  nB->geom.bound_face += VEL(eB->r) / 2;

  break_cell_from_node( nB, cB );
  break_cell_from_node( nAB, cB );
  hook_from_node_to_cell( nB, cA );
  FOR(i,0,3)
    if( cA->geom.connect.nodes[i] == node ) cA->geom.connect.nodes[i] = nB;
```

Recompute metric terms.

```
  cA->geom.metric.area = tri_cell_area( cA->geom.connect.nodes );
  TriaveV( 2, (cA->geom.connect.nodes[0])->geom.coord,
              (cA->geom.connect.nodes[1])->geom.coord,
              (cA->geom.connect.nodes[2])->geom.coord,
               cA->geom.metric.split_point );
  if( viscous == 2 || viscous == 4 ) containment_dual( cA );
  else FOR(i,0,NDIMS)
    cA->geom.metric.split_containment[i] = cA->geom.metric.split_point[i];

  nA->geom.dual_area = median_dual_one( nA );
  nB->geom.dual_area = median_dual_one( nB );
  nAB->geom.dual_area = median_dual_one( nAB );
```

Correct the edge that used to point to cell B to point to cell A.

```
  EDGE_LOOP_FORWARD
    if( edge != eB && edge != eAB ){
      if( edge->cell_right == cB ) edge->cell_right = cA;
      else if( edge->cell_left == cB ) edge->cell_left = cA;
    }
```


Kill the node, a cell, and two edges.

```
  kill_node( node );
  kill_cell( cB );
  kill_edge( eB );
  kill_edge( eAB );

  return 1;
}
```

10.13 Global point insertion, error based

Perform global loop over edges and add a new point at the middle of any large-error edges.

```
void global_error_insert( void ){
  Edge *edge;
  unsigned int split_edges = 0;

  EDGE_LOOP_FORWARD if( edge->error > add_thresh ){
    split_edge( edge );
    ++split_edges;
  }
  printf("Added %u new nodes\n", split_edges );
}
```

10.14 Split an edge

Split an edge in two, adding a node, two cells, and three edges for an interior edge. For a boundary edge, add one node, one cell, and two edges. The process is the reverse of point deletion.

```
void split_edge( Edge *eAB ){
  Node *nAB, *nBC=NULL, *nCD, *nDA=NULL, *node;
  Cell *cA, *cB, *cC=NULL, *cD;
  Edge *eBC=NULL, *eCD, *eDA;
  Bound_edge *bAB=NULL, *bCD, *bedge;
  Node2Edge *edgen;
  int i;
```

Identify the geometry.

```
  /*            nDA                         nDA         *
   *        cA   eDA   cD                    cA          *
   *  nAB   eAB  node  eCD  nCD  <--  nAB   eAB   nCD   *
```

```
 *        cB   eBC   cC                    cB          *
 *             nBC                         nBC         */

nAB = eAB->node[0];
nCD = eAB->node[1];
cA = eAB->cell_left;
FOR(i,0,3) if( cA->geom.connect.nodes[i] == nAB )
  nDA = cA->geom.connect.nodes[cyclic_minus0(i)];

if( (cB = eAB->cell_right) ){
  FOR(i,0,3) if( cB->geom.connect.nodes[i] == nAB )
    nBC = cB->geom.connect.nodes[cyclic_plus0(i)];
}
else BEDGE_LOOP_FORWARD
    if( bedge->node[0] == nCD && bedge->node[1] == nAB ) bAB = bedge;
```

Create the new node.

```
node = add_node();
node->bc = cB ? interior : bAB->bc;
FOR(i,0,NEQS){
  node->vars.conserved[i] = (nAB->vars.conserved[i]+nCD->vars.conserved[i])/2;
  node->vars.wall[i] = ( nAB->vars.wall[i] + nCD->vars.wall[i] ) / 2;
  node->update.art_diss[i] = 0;
}
conserved_to_primitive( node->vars.conserved, node->vars.primitive );
conserved_to_parameter( node->vars.conserved, node->vars.parameter );
FOR(i,0,NDIMS) node->geom.coord[i]=(nAB->geom.coord[i]+nCD->geom.coord[i])/2;
if( cB ){
  node->geom.bound_face = 0;
  node->geom.dual_area = ( cA->geom.metric.area + cB->geom.metric.area ) / 3;
}
else{
  node->geom.bound_face = VEL( eAB->r ) / 2;
  node->geom.dual_area = cA->geom.metric.area / 3;
}

nAB->geom.dual_area -= cA->geom.metric.area / 3;
nCD->geom.dual_area -= cA->geom.metric.area / 3;
if( cB ){
  nAB->geom.dual_area -= cB->geom.metric.area / 3;
  nCD->geom.dual_area -= cB->geom.metric.area / 3;
}
else{
  nAB->geom.bound_face -= VEL( eAB->r ) / 4;
  nCD->geom.bound_face -= VEL( eAB->r ) / 4;
}
```

Add the new cells. Hook to the old nodes. Update nodal values that depend upon the cells.


```
FOR(i,0,3) if(cA->geom.connect.nodes[i]==nCD) cA->geom.connect.nodes[i]=node;
cD = add_cell();
cD->geom.connect.nodes[0] = nDA;
cD->geom.connect.nodes[1] = node;
cD->geom.connect.nodes[2] = nCD;
cD->geom.metric.area = cA->geom.metric.area /= 2;
TriaveV( 2, (cA->geom.connect.nodes[0])->geom.coord,
            (cA->geom.connect.nodes[1])->geom.coord,
            (cA->geom.connect.nodes[2])->geom.coord,
             cA->geom.metric.split_point );
TriaveV( 2, (cD->geom.connect.nodes[0])->geom.coord,
            (cD->geom.connect.nodes[1])->geom.coord,
            (cD->geom.connect.nodes[2])->geom.coord,
             cD->geom.metric.split_point );
if( viscous == 2 || viscous == 4 ){
  containment_dual( cA );
  containment_dual( cD );
}
else FOR(i,0,NDIMS){
  cA->geom.metric.split_containment[i] = cA->geom.metric.split_point[i];
  cD->geom.metric.split_containment[i] = cD->geom.metric.split_point[i];
}

if( cB ){
  FOR(i,0,3)if(cB->geom.connect.nodes[i]==nCD) cB->geom.connect.nodes[i]=node;
  cC = add_cell();
  cC->geom.connect.nodes[0] = nCD;
  cC->geom.connect.nodes[1] = node;
  cC->geom.connect.nodes[2] = nBC;
  cC->geom.metric.area = cB->geom.metric.area /= 2;
  TriaveV( 2, (cB->geom.connect.nodes[0])->geom.coord,
              (cB->geom.connect.nodes[1])->geom.coord,
              (cB->geom.connect.nodes[2])->geom.coord,
               cB->geom.metric.split_point );
  TriaveV( 2, (cC->geom.connect.nodes[0])->geom.coord,
              (cC->geom.connect.nodes[1])->geom.coord,
              (cC->geom.connect.nodes[2])->geom.coord,
               cC->geom.metric.split_point );
  if( viscous == 2 || viscous == 4 ){
    containment_dual( cB );
    containment_dual( cC );
  }
  else FOR(i,0,NDIMS){
    cB->geom.metric.split_containment[i] = cB->geom.metric.split_point[i];
    cC->geom.metric.split_containment[i] = cC->geom.metric.split_point[i];
  }
}

nAB->geom.dual_area += cA->geom.metric.area / 3;
```

```
  nCD->geom.dual_area += cD->geom.metric.area / 3;
  if( cB ){
    nAB->geom.dual_area += cB->geom.metric.area / 3;
    nCD->geom.dual_area += cC->geom.metric.area / 3;
  }
  break_cell_from_node( nCD, cA );
  hook_from_node_to_cell( node, cA );
  hook_from_node_to_cell( nDA, cD );
  hook_from_node_to_cell( nCD, cD );
  hook_from_node_to_cell( node, cD );
  if( cB ){
    break_cell_from_node( nCD, cB );
    hook_from_node_to_cell( node, cB );
    hook_from_node_to_cell( nBC, cC );
    hook_from_node_to_cell( nCD, cC );
    hook_from_node_to_cell( node, cC );
  }
```

Create the new edges and connect to nodes and cells. Modify the existing edge.

```
  eAB->error /= 2;
  eAB->node[1] = node;
  FOR(i,0,NDIMS) eAB->r[i] /= 2;
  eDA = add_edge();
  eDA->cell_left = cA;
  eDA->cell_right = cD;
  eDA->node[0] = node;
  eDA->node[1] = nDA;
  FOR(i,0,NDIMS) eDA->r[i] = nDA->geom.coord[i] - node->geom.coord[i];
  eCD = add_edge();
  eCD->cell_left = cD;
  if( cC ) eCD->cell_right = cC;
  eCD->node[0] = node;
  eCD->node[1] = nCD;
  FOR(i,0,NDIMS) eCD->r[i] = nCD->geom.coord[i] - node->geom.coord[i];
  if( cB ){
    eBC = add_edge();
    eBC->cell_left = cC;
    eBC->cell_right = cB;
    eBC->node[0] = node;
    eBC->node[1] = nBC;
    FOR(i,0,NDIMS) eBC->r[i] = nBC->geom.coord[i] - node->geom.coord[i];
  }

  hook_from_node_to_edge( node, eAB );
  hook_from_node_to_edge( node, eDA );
  hook_from_node_to_edge( nDA, eDA );
  hook_from_node_to_edge( node, eCD );
  break_edge_from_node( nCD, eAB );
```



```
  hook_from_node_to_edge( nCD, eCD );
  if( cB ){
    hook_from_node_to_edge( node, eBC );
    hook_from_node_to_edge( nBC, eBC );
  }

  EDGE_LOOP_AT_NODE( nCD ){
    if( edgen->edge->node[0] == nDA || edgen->edge->node[1] == nDA ){
      if( edgen->edge->cell_left == cA ) edgen->edge->cell_left = cD;
      else edgen->edge->cell_right = cD;
    }
    if( cB && (edgen->edge->node[0] == nBC || edgen->edge->node[1] == nBC) ){
      if( edgen->edge->cell_left == cB ) edgen->edge->cell_left = cC;
      else edgen->edge->cell_right = cC;
    }
  }
```

Handle boundary if this is a boundary edge.

```
  if( bAB ){
    bAB->node[0] = node;
    bCD = add_bedge();
    bCD->bc = bAB->bc;
    bCD->node[0] = nCD;
    bCD->node[1] = node;
  }
}
```

10.15 Global loop for nodal displacement based on error estimates

Perform nodal displacements using the spring analogy based upon edge error estimates. Currently only moving interior nodes.

```
void global_error_move( void ){
  Node *node;
  Cell *cell;
  Edge *longest_edge;
  Node2Edge *edgen;
  int i, try_it;
  Coord R;  /*  vector from current position to new position  */

  NODE_LOOP_FORWARD if(node->bc==interior && node->adapt.error > move_thresh){
    try_it = 0;
    while( !try_it ){
      FOR(i,0,NDIMS) R[i] = 0;

      if( mesh_adapt == move_H ){
```

```
        longest_edge = node->geom.edgecon->edge;
        EDGE_LOOP_AT_NODE( node ){
          if( edgen->edge->node[0] == node )
            FOR(i,0,NDIMS) R[i] += edgen->edge->error * edgen->edge->r[i];
          else FOR(i,0,NDIMS) R[i] -= edgen->edge->error * edgen->edge->r[i];
          if(VEL(edgen->edge->r)>VEL(longest_edge->r)) longest_edge=edgen->edge;
        }
        FOR(i,0,NDIMS) R[i] /= node->adapt.error;
```

Tried to allow more moves by performing edge swaps. Not working as I planned, as too many swaps are badly distorting the grid. Maybe I should only allow an edge swap if the edge error is small.

```
/*       if( move_node( node, R ) || !querry_swap( longest_edge ) ) ++try_it; */
/*       else swap_edge( longest_edge ); */
      }

      else if( mesh_adapt == move_FM ) FM_movement( node, R );
      else{
        printf("Error: bad value for mesh_adapt=%d in global_error_move\n",
               mesh_adapt);
        puts("exiting");
        exit(1);
      }

      move_node( node, R );
      ++try_it;
    }
  }

  move_node( NULL, R );

  CELL_LOOP_FORWARD if( cell->geom.metric.area <= 0 ){
    puts("Error, created negative area during nodal displacements");
    printf("  Cell area = %f for cell:\n", cell->geom.metric.area );
    Aetna_cell( cell );
    exit(1);
  }
}
```

10.16 Move a node

Given a node and a position vector pointing from the current position to the desired new position, move the node there and update all related geometric information. If the new position is invalid, only move the node as far along the directed distance as is legal. For a null node, print diagnostic message.


```
int move_node( Node *node, Coord r ){
  Node2Cell *celln;
  Node2Edge *edgen;
  static unsigned int nodes_moved=0, numFails=0;
  static double dist_moved=0, rms_moved=0;
  int i, itr=0, itr_max=15;
  enum { false=0, true=1 } success=false;
  double area_min1=1, area_max1=0, area_min2, area_max2, area;
  double area_limit = 1.25;
  double angle_max=175, largest_angle;
```

First tentatively move the node and check for negative areas. Also, restrict new min/max cell areas to be within a factor of previous cell min/max areas.

```
  if( node ){
    CELL_LOOP_AT_NODE( node ){
      area_min1 = MIN( area_min1, celln->cell->geom.metric.area );
      area_max1 = MAX( area_max1, celln->cell->geom.metric.area );
    }

    while( ++itr <= itr_max && !success ){
      FOR(i,0,NDIMS) node->geom.coord[i] += r[i];
      area_min2 = area_min1;
      area_max2 = area_max1;
      largest_angle = 90;
      CELL_LOOP_AT_NODE( node ){
        area = tri_cell_area( celln->cell->geom.connect.nodes );
        area_min2 = MIN( area, area_min2 );
        area_max2 = MAX( area, area_max2 );
        largest_angle = MAX( largest_angle, max_cell_angle( celln->cell ) );
      }
      if( largest_angle > angle_max || area_min2 < area_min1 / area_limit ||
          area_max2 > area_max1 * area_limit ){
        FOR(i,0,NDIMS){
          node->geom.coord[i] -= r[i];
          r[i] /= 2;
        }
      }
      else{
        ++nodes_moved;
        dist_moved += VEL(r);
        rms_moved += VEL2(r);
        success = true;

        CELL_LOOP_AT_NODE( node ){
          celln->cell->geom.metric.area =
            tri_cell_area( celln->cell->geom.connect.nodes );
          TriaveV( NDIMS, (celln->cell->geom.connect.nodes[0])->geom.coord,
                          (celln->cell->geom.connect.nodes[1])->geom.coord,
```

```
                        (celln->cell->geom.connect.nodes[2])->geom.coord,
                         celln->cell->geom.metric.split_point );
          if( viscous == 2 || viscous == 4 ) containment_dual( celln->cell );
          else FOR(i,0,NDIMS) celln->cell->geom.metric.split_containment[i] =
                              celln->cell->geom.metric.split_point[i];
        }
        node->geom.dual_area = median_dual_one( node );

        EDGE_LOOP_AT_NODE( node ){
          FOR(i,0,NDIMS) edgen->edge->r[i]=edgen->edge->node[1]->geom.coord[i] -
                                           edgen->edge->node[0]->geom.coord[i];
          edgen->edge->error = error_estimate(edgen->edge->node,edgen->edge->r);
        }
      }
    }
    if( !success ) ++numFails;
  }
  else{
    if(nodes_moved)
      printf("\n%u nodes were moved, total distance = %f\nRMS of moves = %f\n",
             nodes_moved, dist_moved, sqrt(rms_moved/nodes_moved) );
    else puts("\nNo nodes were moved");
    printf("Failed to move %d times\n", numFails );
  }

  return success;
}
```


11 Limiters

11.1 Limiter Functions

Limiter functions, $\psi(r)$, are defined, where $r = P/Q$ is the ratio of the two limiter test values. The limiters are symmetric,

$$\psi\left(\frac{1}{r}\right) = \frac{\psi(r)}{r}$$

or,

$$Q\,\psi(P/Q) = P\,\psi(Q/P)$$

In order to avoid numerical problems with division, the limiters are recast in terms of their equivalent symmetric averaging functions, $M(P,Q)$. These obey,

$$Q\,\psi(P/Q) = M(P,Q) = M(Q,P) = P\,\psi(Q/P)$$

The averaging function M is returned. The two arguments, P and Q, are sent, along with the small parameter, ε^2, used in the van Albada limiter.

```
#include <math.h>     /*  fabs         */
#include <float.h>    /*  DBL_EPSILON  */
#include "limiter.h"
#include "basics.h"   /*  SQUARE       */


double limiter( double P, double Q, double epsVA )
{
  double M ;       /*  averaging function  */
  double comp ;    /*  maximum compression  */
```

Unlimited: This is a non-symmetric limiter, useful for the finite volume formulation. Application to fluctuation splitting still needs to be worked out. It allows a second-order accurate, non-positive scheme, which may still be stable in the absence of shocks.

$$\psi = 1, \quad M = Q$$

```
  switch( limiter_type ){
  case unlimited:
    M = Q ;
    break;
```

van Leer[vL74]

$$\psi\left(\frac{P}{Q}\right) = \frac{\frac{P}{Q} + |\frac{P}{Q}|}{1 + |\frac{P}{Q}|} = \frac{PQ + |PQ|}{Q^2 + |PQ|} = \frac{|P| + |Q|\frac{P}{Q}}{|P| + |Q|}$$

$$M(P,Q) = \frac{|P|Q + P|Q|}{|P| + |Q|}$$

```
case van_leer:
  M = (fabs(P) * Q + P * fabs(Q)) / (fabs(P) + fabs(Q) + DBL_EPSILON) ;
  break;
```

van Albada[vAvLR81]

$$\psi\left(\frac{P}{Q}\right) = \frac{P(P+Q)}{P^2+Q^2} \text{ (Sweby)}$$

$$M(P,Q) = \frac{(P^2+\varepsilon^2)Q + (Q^2+\varepsilon^2)P}{(P^2+\varepsilon^2)+(Q^2+\varepsilon^2)} = \frac{(PQ+\varepsilon^2)(P+Q)}{P^2+Q^2+2\varepsilon^2}$$

where $\varepsilon^2 \sim \Delta x^3$ (assumes typical $\nabla u \sim 1$).

```
case van_albada:
  M = (P * Q + epsVA) * (P + Q) / (SQUARE(P) + SQUARE(Q) + 2 * epsVA) ;
  break;
```

Discontinuous: A generalized limiter[Swe84] based on a given compression limit, ϕ, that contains the first-order, $\phi = 0$, minmod, $\phi = 1$, and superbee, $\phi = 2$, as special cases.

$$\psi_\phi(r) = \max[0,\ \min(\phi r,\ 1),\ \min(r,\ \phi)]$$

$$\psi_\phi\left(\frac{P}{Q}\right) = \begin{cases} 0 & PQ \le 0 \\ \phi P/Q & \phi|P| \le |Q| \\ 1 \quad \text{if} & |P| \le |Q| \le \phi|P| \\ P/Q & |Q| \le |P| \le \phi|Q| \\ \phi & \phi|Q| \le |P| \end{cases}$$

$$M(P,Q) = \begin{cases} 0 & PQ \le 0 \\ \phi P & \phi|P| \le |Q| \\ Q \quad \text{if} & |P| \le |Q| \le \phi|P| \\ P & |Q| \le |P| \le \phi|Q| \\ \phi Q & \phi|Q| \le |P| \end{cases}$$

```
default:
  switch( limiter_type ){
  case compress:
    comp = COMPRESS ;
    break;
  case superbee:
    comp = 2 ;
    break;
  case minmod:
    comp = 1 ;
    break;
  case first_order:    /*  this is the default, so drop through  */
```



```
    default:
      comp = 0 ;
      break;
    }
    M =    P *      Q  <= 0       ?        0 :
        comp * fabs(P) <= fabs(Q) ? comp * P :
        comp * fabs(Q) <= fabs(P) ? comp * Q :
               fabs(P) <= fabs(Q) ?        Q : P ;
  }
  return M ;
}
```

11.2 Eigenvalue Limiting

Perferm eigenvalue limiting using the scheme of Harten and Hyman[HH83]. This is only applied to obtain the absolute value of eigenvalues that pass through zero at a sonic point, and serves to introduce dissipation, leading to entopy positivity and the supression of expansion shocks.

$$|\tilde{\lambda}|_i \leftarrow \begin{cases} |\tilde{\lambda}|_i & \text{if } |\tilde{\lambda}|_i \ge \epsilon \\ \frac{1}{2}\left(\frac{\tilde{\lambda}_i^2}{\epsilon} + \epsilon\right) & \text{if } |\tilde{\lambda}|_i < \epsilon \end{cases}$$

$$\epsilon = \max\left[0, (\tilde{\lambda}_i - \lambda_L), (\lambda_R - \tilde{\lambda}_i)\right]$$

Add the capability to stuff more dissipation into the solution through smearing.

```
double eigenvalue_limiter( double vL, double v, double vR )
{
  const double papsmear = 1 ;         /*  add additional smearing here  */
  double eps = MAX( 0, MAX( v-vL, vR-v ) ) * papsmear ;
  return (fabs(v) >= eps) ? fabs(v) : 0.5 * ( eps + SQUARE(v)/eps );
}
```

12 Roe Averaging

Here we collect functions useful for performing Roe averaging in one and two dimensions.

```
#include "basics.h"     /*  Cell, etc      */
#include "TriMath.h"    /*  TriaveV        */
#include "therm.h"      /*  a(H,V)         */
#include "transform.h"  /*  Z-> V          */
```

12.1 Cell Roe Average

Perform averaging for a triangular cell. It is assumed that the parameter vector has already been defined at the nodes. First average the parameter vector, then decode the primitive variables from the parameter vector.

```
void Roe_average_cell( Cell *cell )
{
  TriaveV( 4, (cell->geom.connect.nodes[0])->vars.parameter,
              (cell->geom.connect.nodes[1])->vars.parameter,
              (cell->geom.connect.nodes[2])->vars.parameter,
               cell->tilde.parameter );
  parameter_to_primitive( cell->tilde.parameter, cell->tilde.primitive );
  cell->tilde.H = cell->tilde.parameter[3] / cell->tilde.parameter[0] ;
  cell->tilde.a = a_from_H_Vel( cell->tilde.H, &cell->tilde.primitive[1] );
}
```

12.2 Edge Roe Average

Perform Roe averaging at a face. Send the parameter vector to the left and right of the face, and return the averaged primitve vector along with the total enthalpy and speed of sound.

```
void Roe_average_edge( SolVec Zl, SolVec Zr, SolVec Vtilde, double *Htilde_p,
                       double *atilde_p )
{
  SolVec Zave ;
  int i ;

  FOR(i,0,NEQS) Zave[i] = ( Zl[i] + Zr[i] ) / 2 ;
  parameter_to_primitive( Zave, Vtilde );
  *Htilde_p = Zave[NEQS-1] / Zave[0] ;
  *atilde_p = a_from_H_Vel( *Htilde_p, &Vtilde[1] );
}
```


13 Stonehenge

Get things going by loading global options, building a grid, and initializing the flowfield.

```
#include <stddef.h>       /*  NULL                            */
#include <string.h>       /*  file name manipulating          */
#include <stdio.h>        /*  read, write utilities, perror   */
#include <time.h>         /*  time, ctime                     */
#include <stdlib.h>       /*  exit, strtod, strtol, free      */
#include <errno.h>        /*  errno                           */
#include <math.h>         /*  cos, sin, acos, pow, fabs       */
#include <float.h>        /*  FLT_EPSILON                     */
#include "basics.h"       /*  basic parameters for entire code  */
#include "therm.h"        /*  P_from_rho_T                    */
#include "TriMath.h"      /*  tri_cell_area, median_dual_all... */
#include "rhs.h"          /*  axi_source_type                 */
#include "limiter.h"      /*  limiter_type                    */
#include "adaption.h"     /*  mesh_adapt                      */
#include "transform.h"    /*  V2U, U2Z                        */

/*  Prototypes only used in this file  */
void read_input_deck( void );
void read_start_grid( void );
void decode_com_opt( const int argc, char *argv[] );
void read_node( FILE* f, const char *format );
void read_cell( FILE* f, const char *format, Node *nodal_stack[] );
void solution_line_up( void );
long S2lint ( char *label, char *string );
double S2doub ( char *label, char *string );

/*  Variables used only in this file  */
enum { form_tec, file_unformatted, file_formatted,
       cold_start=20, create_uniform_nondim, create_uniform_stagnant }
       where_grid = form_tec,
       where_sol  = create_uniform_nondim ;


void Stonehenge( int argc, char *argv[] )
{
  Cell *cell ;
  Node *node ;
  Bound_edge *bedge ;
  int i ;                   /*  dummy indicie           */
  Node *nptmp0, *nptmp1 ;   /*  temp pointers to nodes  */
  Coord DL ;                /*  temporary lengths       */
  double dGam ;             /*  boundary edge length    */
```

Decode command line options.

```
decode_com_opt( argc, argv );
```

Open and read controlling input deck.

```
read_input_deck();
```

At this point there is no grid, so go get a domain.

```
read_start_grid();
```

Select the starting line-up for the dependent variables and assign boundary condition states to the nodes.

```
solution_line_up();
```

13.0.1 Grid Metrics

Perform geometric calculations on the mesh and store them for later use.

```
/*  Cell areas  */
CELL_LOOP_FORWARD
  if((cell->geom.metric.area=tri_cell_area(cell->geom.connect.nodes))<=0){
    printf("\nNegative area = %.3e for cell:\n", cell->geom.metric.area);
    Aetna_cell(cell);
  }

/*  Median-dual areas  */
median_dual_all();
```

For median-dual finite volume mesh, the split point is the cell centroid. Also compute for viscous cases just in case it is needed.

```
/*  Split-points for defining FV median dual mesh  */
if( solver || viscous ) CELL_LOOP_FORWARD
  TriaveV( 2, (cell->geom.connect.nodes[0])->geom.coord,
              (cell->geom.connect.nodes[1])->geom.coord,
              (cell->geom.connect.nodes[2])->geom.coord,
               cell->geom.metric.split_point );
```

If desired, the viscous terms can be computed on a containment-dual mesh instead of the median-dual.


```
if( viscous==2 || viscous==4 ) CELL_LOOP_FORWARD containment_dual( cell );
else if( viscous==1 || viscous==3 ) CELL_LOOP_FORWARD
  FOR(i,0,NDIMS) cell->geom.metric.split_containment[i] =
                 cell->geom.metric.split_point[i] ;
```

Build linked list of boundary edges.

```
/*  Identify boundary edges  */
CELL_LOOP_FORWARD FOR(i,0,3){
  nptmp0 = cell->geom.connect.nodes[i] ;
  nptmp1 = cell->geom.connect.nodes[cyclic_plus0(i)] ;
  build_bedges( nptmp0, nptmp1, 0 );
}
```

Determine the area of viscous-wall faces associated with each node.

```
if( viscous ){
  NODE_LOOP_FORWARD node->geom.bound_face = 0 ;
  BEDGE_LOOP_FORWARD
    if( bedge->bc == wall_viscous ){
      FOR(i,0,NDIMS) DL[i] = (bedge->node[1])->geom.coord[i] -
                             (bedge->node[0])->geom.coord[i] ;
      dGam = VEL(DL) / 2 ;
      FOR(i,0,2) (bedge->node[i])->geom.bound_face += dGam ;
    }
}
```

13.0.2 Plot Starting Point

Write initial grid and solution to plot file, then open convergence history plot file. Zero out any uninitialized variables that are written to the plotfiles.

```
NODE_LOOP_FORWARD FOR(i,0,NEQS){
  node->update.source[i]=node->update.flux[i]=node->update.art_diss[i]=0;
  if( solver ) node->fv.gradx[i] = node->fv.grady[i] = node->fv.psi[i]=0;
}
write_tecplot_0();
write_hist( 0, 0, 1, (where_sol < cold_start) ? 1 : 0 );

if ( traceback > 1 ) puts("No longer on the Salisbury Plain\n");
}
```

14 Command Line Options

Decode the command line options. This consists primarily of assigning file names. If no options are given, the program name is used as the root name for the file names. Root names are limited to 16 characters (plus terminating null character).

```
void decode_com_opt( const int argc, char *argv[] )
{
  register int i ;

  /*  program name and start time  */
  time_t now = time(NULL) ;
  printf("\n   Starting program %s, %s\n\n", argv[0], ctime(&now) );

  /*  default file names  */
  strncpy( rootf, argv[0], 16 );
  rootf[16] = '\0' ;
  strcat( strcpy( inpf,      rootf ), ".inp"   );
  strcat( strcpy( ingridf,   rootf ), ".g"     );
  strcat( strcpy( insolf,    rootf ), ".s"     );
  strcat( strcpy( inrstf,    rootf ), ".rst"   );
  strcat( strcpy( outplotf,  rootf ), ".dat"   );
  strcat( strcpy( histplotf, rootf ), "_h.dat" );

  /*  Now process options to change file names            */
  /*  First parse for a help request or root name change  */
  /*  Then update individual file names                   */
  FOR(i,1,argc){                                        /*  Error and help  */
    if ( argv[i][0] != '-'
         || !strchr( "tirgspc", argv[i][1] )
         || argv[i][2] == '\0' ){
      printf("Error reading command line argument %d, %s\n", i, argv[i] );
      puts("Arguments must start with a '-'");
      puts("  have a single character flag");
      puts("  and no space between the flag and file name");
      puts("Root file can have 16 characters, other files"
           " can have 20 characters, including extension");
      puts("Example:");
      puts("hackboy -tRootName -iInputDeck.inp -rRestart.rst -cHistPlot_h.dat");
      puts("        -gInputGrid.g -sInputSolution.s -pOutPlot.dat");
      puts("Stopping execution");
      exit(1) ;
    }
    else if ( argv[i][1] == 't' ){                /*  Search for root name  */
      strncpy( rootf, &argv[i][2], 16 );
      rootf[16] = '\0' ;
      strcat( strcpy( inpf, rootf ), ".inp" );
      strcat( strcpy( ingridf, rootf ), ".g" );
      strcat( strcpy( insolf, rootf ), ".s" );
```



```
      strcat( strcpy( inrstf, rootf ), ".rst" );
      strcat( strcpy( outplotf, rootf ), ".dat" );
      strcat( strcpy( histplotf, rootf ), "_h.dat" );
    }
  }
  FOR(i,1,argc){                                  /*  Process other options  */
    switch( argv[i][1] ){
    case 'i': strncpy( inpf, &argv[i][2], 20 );
      inpf[20] = '\0'; break;
    case 'r': strncpy( inrstf, &argv[i][2], 20 );
      inrstf[20] = '\0'; break;
    case 'g': strncpy( ingridf, &argv[i][2], 20 );
      ingridf[20] = '\0'; break;
    case 's': strncpy( insolf, &argv[i][2], 20 );
      insolf[20] = '\0'; break;
    case 'p': strncpy( outplotf, &argv[i][2], 20 );
      outplotf[20] = '\0'; break;
    case 'c': strncpy( histplotf, &argv[i][2], 22 );
      histplotf[22] = '\0'; break;
    case 't': break;
    default: printf("Failure for command-line argument: %s\n", argv[i] );
      puts("Exiting");
      exit(1) ;
    }
  }
  printf("Root file name        %s\n", rootf );
  printf("Input deck  name      %s\n", inpf );
  printf("Restart file name     %s\n", inrstf );
  printf("Initial grid name     %s\n", ingridf );
  printf("Initial solution name %s\n", insolf );
  printf("Plot file name        %s\n", outplotf );
  printf("History file name     %s\n\n", histplotf );
}
```

15 Read Input Deck

Here we open and read the main controlling input deck. The input deck name can be entered as a command line option using `-i`.

```
void read_input_deck( void )
{
  char mode[3] = "r";
  FILE *indeck = open_file( inpf, mode );
  char line[80], key[80], value[80] ;
```

Read and parse one line at a time from the input file. The pound symbol, #, indicates a comment.

```
  while( fgets( line, 80, indeck ) ){              /*  While there is input  */
    if( sscanf( line, " %s = %s", key, value ) > 0 ){  /*  If not blank line */

      if( !strcmp( key, "where_grid" ) ){                    /*  Input grid  */
        if( !strcmp( value, "form_tec" ) ) where_grid = form_tec ;
        else if( !strcmp( value, "file_unformatted" ) )
          where_grid = file_unformatted ;
        else{
          printf("Bad value entered for where_grid=%s\nExiting\n", value );
          exit(1) ;
        }
      }

      else if( !strcmp( key, "where_sol" ) ){            /*  Input solution  */
        if( !strcmp( value, "form_tec" ) ) where_sol = form_tec ;
        else if( !strcmp( value, "create_uniform_nondim" ) )
          where_sol = create_uniform_nondim ;
        else if( !strcmp( value, "create_uniform_stagnant" ) )
          where_sol = create_uniform_stagnant ;
        else if( !strcmp( value, "file_unformatted" ) )
          where_sol = file_unformatted ;
        else if( !strcmp( value, "file_formatted" ) )
          where_sol = file_formatted ;
        else{
          printf("Bad value entered for where_sol=%s\nExiting\n", value );
          exit(1) ;
        }
      }

      else if( !strcmp( key, "axisym" ) )             /*  Axisymmetric flag  */
        axisym = (int) S2lint( "axisym", value ) ;

      else if( !strcmp( key, "axi_source_type" ) ){   /*  source option      */
        if     ( !strcmp( value, "mass_lumped")) axi_source_type = mass_lumped ;
```



```
        else if( !strcmp( value, "averaged"   )) axi_source_type = averaged    ;
        else if( !strcmp( value, "linearP"    )) axi_source_type = linearP     ;
        else if( !strcmp( value, "linearZ"    )) axi_source_type = linearZ     ;
        else{
          printf("Bad value entered for axi_source_type=%s\nExiting\n", value );
          exit(1) ;
        }
      }

      else if( !strcmp( key, "memory" ) )             /*  Memory level flag  */
        memory = (int) S2lint( "memory", value ) ;

      else if( !strcmp( key, "safety" ) )             /*  Safety flag        */
        safety = (int) S2lint( "safety", value ) ;

      else if( !strcmp( key, "scalar" ) )             /*  Scalar flag        */
        scalar = (int) S2lint( "scalar", value ) ;

      else if( !strcmp( key, "solver" ) )             /*  Solver flag        */
        solver = (int) S2lint( "solver", value ) ;

      else if( !strcmp( key, "viscous" ) )            /*  Viscous flag       */
        viscous = (int) S2lint( "viscous", value ) ;

      else if( !strcmp( key, "traceback" ) )          /* Traceback level flag */
        traceback = (int) S2lint( "traceback", value ) ;

      else if( !strcmp( key, "limiter_type" ) ){      /*  limiter choice     */
        if     ( !strcmp(value, "first_order") ) limiter_type = first_order ;
        else if( !strcmp(value, "minmod"     ) ) limiter_type = minmod      ;
        else if( !strcmp(value, "superbee"   ) ) limiter_type = superbee    ;
        else if( !strcmp(value, "compress"   ) ) limiter_type = compress    ;
        else if( !strcmp(value, "van_leer"   ) ) limiter_type = van_leer    ;
        else if( !strcmp(value, "van_albada" ) ) limiter_type = van_albada  ;
        else if( !strcmp(value, "unlimited"  ) ) limiter_type = unlimited   ;
      }

      else if( !strcmp( key, "mesh_adapt" ) ){      /*  mesh adaption flag   */
        if     ( !strcmp(value, "none"        ) ) mesh_adapt = none        ;
        else if( !strcmp(value, "edge_swap_H" ) ) mesh_adapt = edge_swap_H ;
        else if( !strcmp(value, "edge_swap_FM") ) mesh_adapt = edge_swap_FM;
        else if( !strcmp(value, "delete_H"    ) ) mesh_adapt = delete_H    ;
        else if( !strcmp(value, "delete_FM"   ) ) mesh_adapt = delete_FM   ;
        else if( !strcmp(value, "add_H"       ) ) mesh_adapt = add_H       ;
        else if( !strcmp(value, "add_FM"      ) ) mesh_adapt = add_FM      ;
        else if( !strcmp(value, "move_H"      ) ) mesh_adapt = move_H      ;
        else if( !strcmp(value, "move_FM"     ) ) mesh_adapt = move_FM     ;
      }

      else if( !strcmp( key, "adapt_target" ) )       /* adaption target     */
```

```
  adapt_target = S2doub( "adapt_target", value ) ;

else if( !strcmp( key, "nondim_L" ) )           /*  nondim length      */
  nondim_L = S2doub( "nondim_L", value ) ;

else if( !strcmp( key, "nondim_V" ) )           /*  nondim velocity    */
  nondim_V = S2doub( "nondim_V", value ) ;

else if( !strcmp( key, "nondim_T" ) )           /*  nondim temperature */
  nondim_T = S2doub( "nondim_T", value ) ;

else if( !strcmp( key, "nondim_rho" ) )         /*  nondim density     */
  nondim_rho = S2doub( "nondim_rho", value ) ;

else if( !strcmp( key, "nondim_mu" ) )          /*  nondim viscosity   */
  nondim_mu = S2doub( "nondim_mu", value ) ;

else if( !strcmp( key, "T_wall" ) )             /*  wall temperature   */
  T_wall = S2doub( "T_wall", value ) ;

else if( !strcmp( key, "AOA" ) )                /*  angle of attack    */
  AOA = (float) ( acos(-1) * S2doub( "AOA", value ) / 180 ) ;

else if( !strcmp( key, "Mach" ) )               /*  freestream Mach    */
  Mach = (float) S2doub( "Mach", value ) ;

else if( !strcmp( key, "Reynolds" ) )           /*  Reynolds number    */
  Reynolds = (float) S2doub( "Reynolds", value ) ;

else if( !strcmp( key, "CFL" ) )                /*  CFL number         */
  CFL = (float) S2doub( "CFL", value ) ;

else if( !strcmp( key, "CFL_v" ) )              /*  viscous CFL number */
  CFL_v = (float) S2doub( "CFL_v", value ) ;

else if( !strcmp( key, "L2_done" ) )            /*  L2 stop limit      */
  L2_done = (float) S2doub( "L2_done", value ) ;

else if( !strcmp( key, "itr_done" ) )           /*  iteration limit    */
  itr_done = (int) S2lint( "itr_done", value ) ;

else if( !strcmp( key, "cpu_done" ) ){          /*  time limit         */
  cpu_done = (time_t) S2lint( "cpu_done", value ) ;
  cpu_done *= 3600;                  /*  convert from hours to seconds  */
  }

else if( !strcmp( key, "hist_skip_screen" ) )   /*  history skip       */
  hist_skip_screen = (int) S2lint( "hist_skip_screen", value ) ;

else if( !strcmp( key, "hist_skip_file" ) )     /*  history skip       */
```



```
      hist_skip_file = (int) S2lint( "hist_skip_file", value ) ;

    else if( !strcmp( key, "plot_skip" ) )          /*  plotfile skip      */
      plot_skip = (int) S2lint( "plot_skip", value ) ;

    else if( strcmp( key, "#" ) ){                                  /*  Error  */
      printf("Problem reading this line from %s, skipping\n", inpf );
      puts( line );
    }
  }
}
```

Now make sure all the non-dimensionalizing and freestream values have been set.

```
if( ! nondim_L ) nondim_L = 1 ;
printf("nondim Length = %.5f\n", nondim_L );

if( ! nondim_T ) nondim_T = 288 ;
printf("nondim Temperature = %.2f\n", nondim_T );

if( ! nondim_mu ) nondim_mu = mu_raw( nondim_T );  /* assume mks units */
printf("nondim viscosity = %e\n", nondim_mu );

if( ! nondim_V ){
  if( Mach ) nondim_V = Mach * sqrt( GAMMA * R_air * nondim_T );
  else nondim_V = 1 ;
}
printf("nondim Velocity = %.2f\n", nondim_V );
if( ! Mach ) Mach = nondim_V / sqrt( GAMMA * R_air * nondim_T );
printf("Mach = %.2f\n", Mach );
if( safety && fabs( Mach - nondim_V / sqrt( GAMMA * R_air * nondim_T ) )
    > 0.001 )
  printf("**Warning, Mach=%f, not=%f\n\n", Mach,
         nondim_V / sqrt( GAMMA * R_air * nondim_T ) );

if( ! nondim_rho ){
  if( Reynolds ) nondim_rho = Reynolds * nondim_mu / ( nondim_V * nondim_L );
  else nondim_rho = 1 ;
}
printf("nondim Density = %.5f\n", nondim_rho );
if( ! Reynolds ) Reynolds = nondim_rho * nondim_V * nondim_L / nondim_mu ;
printf("Reynolds = %.0f\n", Reynolds );
if( safety && fabs( Reynolds - nondim_rho * nondim_V * nondim_L / nondim_mu )
    > 1 )
  printf("**Warning, Reynolds=%.0f, not=%.0f\n\n", Reynolds,
         nondim_rho * nondim_V * nondim_L / nondim_mu );

T_wall = T_wall ? T_wall / nondim_T : 1 ;
```

```
  CFL_v = CFL_v ? CFL_v : CFL ;  /*  if no viscous CFL, set to inviscid  */
}
```

15.1 String to Number Conversions

Convert strings to numbers, with error checking.

```
long S2lint ( char *label, char *string )
{
  long number ;
  char **ptr = NULL;
  errno = 0 ;
  number = strtol( string, ptr, 0 );
  if( errno ) perror( strcat("Failure assigning variable ", label ) );
  return number ;
}

double S2doub ( char *label, char *string )
{
  double number ;
  char **ptr = NULL;
  errno = 0 ;
  number = strtod( string, ptr );
  if( errno ) perror( strcat("Failure assigning variable ", label ) );
  return number ;
}
```


16 Initial Grid

Initialize the grid in linked lists. The grid input file name can be set using the command line option -g.

The only option currently available is to read a formatted TECPLOT file.

```
void read_start_grid( void )
{
  char mode[3] = "r";
  FILE *ingrid = open_file( ingridf, mode );
  char line[81] ;
  unsigned int new_nodes, new_cells ;
  Node **nodal_stack ;

  switch( where_grid ){

  case form_tec:                                /*  formatted Tecplot file  */
    fgets( line, 80, ingrid );                              /*  skip line  */
    fscanf( ingrid, "ZONE N= %u E= %u", &new_nodes, &new_cells );
    fgets( line, 80, ingrid );                        /*  skip rest of line  */
    printf("Initializing grid from %s with %u nodes, %u cells\n\n",
            ingridf, new_nodes, new_cells );

    for( ; new_nodes ; --new_nodes )                            /*  Nodes  */
      read_node( ingrid, " %lf %lf" );
    nodal_stack = allocate_node_ptr( nodes+1 );
    stack_nodes( nodal_stack );                       /*  Number the nodes  */
    for( ; new_cells ; --new_cells )                            /*  Cells  */
      read_cell( ingrid, " %u %u %u", nodal_stack );
    free( nodal_stack );
    break;

  default:                                /*  invalid choice for where_grid */
    printf("Problem reading initial grid from %s\n", ingridf );
    printf("verify value of where_grid = %d\n", where_grid );
    puts("stopping");
    exit(1);
  }
  fclose( ingrid );
  if( traceback > 1 ) puts("Finished reading grid");
}
```

16.1 Read a Node

Allocate memory for a node and then read nodal coordinates from a formatted input file.

```
void read_node( FILE* f, const char *format )
```

```
{
  int success ;
  Node *new_node = add_node();

  if( new_node ){
    success = fscanf( f, format,
                      &bottom_node->geom.coord[0],
                      &bottom_node->geom.coord[1] );
    switch( success ){
    case 2:                      /*  2 is the right number of dims. to read  */
      bottom_node->geom.coord[0] /= nondim_L ;    /*  Non-dimensionalization  */
      bottom_node->geom.coord[1] /= nondim_L ;
      break;
    default:
      printf("Incorrectly read %d coordinates for node %u\nExiting\n",
              success, nodes );
      exit(1);
      break;
    }
  }
  else{
    puts("Initial grid too big to fit in memory, Exiting");
    exit(1);
  }
}
```

16.2 Read a Cell

Allocate memory for a cell, then read defining nodes from a formatted input file. Also attach a hook from each of the three nodes pointing back to the current cell.

```
void read_cell( FILE* f, const char *format, Node *nodal_stack[] )
{
  int success, n ;
  unsigned int n_def[3] ;
  Cell *new_cell = add_cell();

  if( new_cell ){
    success = fscanf( f, format, n_def, (n_def+1), (n_def+2) );
    switch( success ){
    case 3:                          /*  this is the right number to read  */
      FOR(n,0,3){
        bottom_cell->geom.connect.nodes[n] = nodal_stack[n_def[n]] ;
        hook_from_node_to_cell( nodal_stack[n_def[n]], bottom_cell );
      }
      break;
    default:
      printf("Incorrectly read %d connecting nodes for cell %u\nExiting\n",
```



```
              success, cells );
      exit(1);
      break;
    }
  }
  else{
    puts("Initial grid too big to fit in memory, Exiting");
    exit(1);
  }
}
```

17 Starting Line-Up

Pick the starting players for the solution team and assign boundary condition types to the nodes. If this is a restart, then the solution and boundary conditions are read from the file specified by the -s command-line option, insolf.s. If this is a cold-start initialization, then the boundary conditions must still be specified in insolf.s.

```
void solution_line_up( void )
{
  char mode[3] = "r" ;
  unsigned int check_nodes ;
  FILE *insol = NULL ;
  register Node *node ;
  double P_i = P_from_rho_T( 1, 1 ) * nondim_rho * SQUARE(nondim_V) ;
  double u_i = cos( AOA ) * nondim_V ;
  double v_i = sin( AOA ) * nondim_V ;
  int i, boundary_type;
  Cell *cell;

  /*  open solution and/or boundary condition file  */
  insol = open_file( insolf, mode );
  fscanf( insol, " %u", &check_nodes );
  if( check_nodes != nodes ){
    printf("Problem reading initial solution from %s\n", insolf );
    printf("%s has %u nodes, but expected %u nodes from grid file %s\n",
           insolf, check_nodes, nodes, ingridf );
    exit(1);
  }

  NODE_LOOP_FORWARD{
    switch( where_sol ){

    case create_uniform_stagnant:                          /*  Stagnant  */
      node->vars.primitive[0] = nondim_rho ;
      node->vars.primitive[1] = 0 ;
      node->vars.primitive[2] = 0 ;
      node->vars.primitive[3] = P_i ;
      node->vars.wall[0] = nondim_T ;
      node->vars.wall[1] = node->vars.wall[2] = node->vars.wall[3] = 0 ;
      break;

    case create_uniform_nondim:                            /*  Uniform  */
      node->vars.primitive[0] = nondim_rho ;
      node->vars.primitive[1] = u_i ;
      node->vars.primitive[2] = v_i ;
      node->vars.primitive[3] = P_i ;
      node->vars.wall[0] = nondim_T ;
      node->vars.wall[1] = node->vars.wall[2] = node->vars.wall[3] = 0 ;
      break;
```



```
    case file_formatted:                                   /*  Read from file  */
      FOR(i,0,4) fscanf( insol, " %lf", &(node->vars.primitive[i]) );
      FOR(i,0,4) fscanf( insol, " %lf", &(node->vars.wall[i]) );
      break;

    default:
      puts("I don't know how to initialize the solution");
      printf(" where_sol = %d\nStopping\n", where_sol );
      exit(1);
    }

    node->vars.primitive[0] /= nondim_rho ;        /*  Non-dimensionalize  */
    node->vars.primitive[1] /= nondim_V ;
    node->vars.primitive[2] /= nondim_V ;
    node->vars.primitive[3] /= ( nondim_rho * SQUARE(nondim_V) );

    primitive_to_conserved( node->vars.primitive, node->vars.conserved );
    conserved_to_parameter( node->vars.conserved, node->vars.parameter );

    node->vars.wall[0] /= nondim_T ;
    node->vars.wall[1] /= ( nondim_rho * SQUARE(nondim_V) );
    node->vars.wall[2] /= ( nondim_rho * SQUARE(nondim_V) );
    node->vars.wall[3] /= ( nondim_rho * pow( nondim_V, 3 ) );

    fscanf(insol," %d", &boundary_type);        /*  assign BC states here  */
    node->bc = boundary_type;
  }

  fclose( insol );                                          /*  close file  */

  CELL_LOOP_FORWARD Roe_average_cell( cell );  /*  initialize tilde state  */

  if( traceback > 1 ) puts("Finished creating initial solution");
}
```

18 Thermodynamic Routines

All thermodynamic routines are collected in this file.

```
#include <math.h>    /*  sqrt          */
#include <stdio.h>   /*  printf, puts  */
#include "therm.h"
#include "basics.h"

/*  Global variables for this file only  */

enum { ideal } gas ;   /*  ideal = thermally and calorically perfect  */

/*  Thermodynamic Functions  */
```

18.1 $P(\rho, T)$

Use the perfect gas relation,

$$P = \rho R T$$

in non-dimensional form,

$$P^* = \rho^* R^* T^*$$

where,

$$R^* = \frac{R T_\infty}{V_\infty^2}$$

Include a check for high density or low temperature where van der Waals forces become important.

```
double P_from_rho_T( const double rho, const double T )
{
  static int warn = 0 ;                              /*  warn only once  */
  double nondim_R = R_air * nondim_T / SQUARE(nondim_V) ;

  if( safety &&
      ( rho * nondim_rho > RHO_max_perfect || T * nondim_T < T_min_perfect ) )
    if( ! warn++ )
      printf("Warning, real gas effects P(r,T), rho= %.1f, T= %.1f\n",
             rho*nondim_rho, T*nondim_T );

  switch( gas ){
  default: puts("\n***bad value for gas in P_from_rho_T");
    /*     drop through and return perfect gas value  */

  case ideal:
    return rho * nondim_R * T ;
  }
}
```


18.2 $e(\rho, P)$

For a calorically perfect gas, dimensional or nondimensional, we have,

$$e = c_v T = \frac{P c_v}{\rho R} = \frac{P}{(\gamma - 1)\rho}$$

The dimensional temperature is used to check for violation of the calorically perfect assumption, obtained as,

$$T = \frac{P^* V_\infty^2}{\rho^* R}$$

```
double e_from_rho_P( const double rho, const double P )
{
  static int warn = 0 ;                              /*  warn only once  */
  double T ;

  if( safety &&
      ( T = P * SQUARE(nondim_V) / rho / R_air ) > T_max_caloric_perf )
    if( ! warn++ ){
      puts("Warning, gas not calorically perfect");
      printf("Temperature = %f\n", T );
    }

  switch( gas ){
  default: puts("\n***bad value for gas in e_from_rho_P");
    /*     drop through and return calorically perfect gas value  */

  case ideal:
    return P / rho / GAMM1 ;
  }
}
```

18.3 $P(\rho, e)$

For a calorically perfect gas, dimensional or nondimensional, we have,

$$P = \rho R T = (\gamma - 1)\rho c_v T = (\gamma - 1)\rho e$$

The dimensional temperature is used to check for violation of the calorically perfect assumption, obtained as,

$$T = (\gamma - 1)\frac{e^* V_\infty^2}{R}$$

```
double P_from_rho_e( const double rho, const double e )
{
  static int warn = 0 ;                              /*  warn only once  */
  double T ;
```

```
  if( safety &&
      ( T = GAMM1 * e * SQUARE(nondim_V) / R_air ) > T_max_caloric_perf )
    if( ! warn++ ){
      puts("Warning, gas not calorically perfect");
      printf("Temperature = %f\n", T );
    }

  switch( gas ){
  default: puts("\n***bad value for gas in P_from_rho_e");
    /*     drop through and return calorically perfect gas value  */

  case ideal:
    return GAMM1 * rho * e ;
  }
}
```

18.4 $e(h)$

For a calorically perfect gas, dimensional or nondimensional, we have,

$$e = c_v T, \quad h = c_p T, \quad \gamma = \frac{c_p}{c_v}$$

from which,

$$h = \gamma e, \quad e = \frac{h}{\gamma}$$

The dimensional temperature is used to check for violation of the calorically perfect assumption, obtained as,

$$T = \frac{\gamma - 1}{\gamma} \frac{h^* V_\infty^2}{R}$$

```
double e_from_h( const double h )
{
  static int warn = 0 ;                              /*  warn only once  */
  double T ;

  if( safety &&
    (T = GAMM1 * h * SQUARE(nondim_V) / R_air / GAMMA) > T_max_caloric_perf ){
    if( ! warn++ ){
      puts("Warning, gas not calorically perfect in e(h)");
      printf("Temperature = %f\n", T );
    }
  }

  switch( gas ){
  default: puts("\n***bad value for gas in e_from_h");
    /*     drop through and return calorically perfect gas value  */
```



```
  case ideal:
    return h / GAMMA ;
  }
}
```

18.5 $a(H, V)$

Compute the speed of sound, given the total enthalpy and pointer to velocity vector.

For a perfect gas,

$$a^2 = (\gamma - 1)\left(H - \frac{u^2 + v^2 + w^2}{2}\right)$$

```
double a_from_H_Vel( const double H, double V[] )
{
  switch( gas ){
  default: printf("\n** Bad value for gas=%d in a(H,V)\n", gas );
    /*   drop through to return perfect gas value  */
  case ideal:
    return sqrt( GAMM1 * ( H - VEL2(V) / 2 ) );
  }
}
```

18.6 $a(\rho, P)$

Compute speed of sound from density and pressure. For a perfect gas,

$$a = \sqrt{\frac{\gamma P}{\rho}}$$

```
double a_from_rho_P( const double rho, const double P )
{
  switch( gas ){
  default: printf("\n** Bad value for gas=%d in a(r,P)\n", gas );
    /*   drop through to return perfect gas value  */
  case ideal:
    return sqrt( GAMMA * P / rho );
  }
}
```

18.7 $\mu(T)$

Evaluate the viscosity as a function of temperature. Using Sutherland law, in non-dimensional form. Good up to $T \leq 3000$ K.

$$\mu = 1.458 \times 10^{-6} \frac{T^{\frac{3}{2}}}{T + 110.6} \frac{kg}{m \cdot s}$$

$$\mu^* = T^* \sqrt{T^*} \frac{1+a}{T^*+a}$$

$$\mu^* = \frac{\mu}{\mu_\infty}, \quad T^* = \frac{T}{T_\infty}, \quad a = \frac{110.6}{T_\infty}$$

```
double mu_from_T( const double T )
{
  double a = 110.6 / nondim_T ;

  return T * sqrt(T) * ( 1 + a ) / ( T + a ) ;
}
```

18.8 $\mu(T)$, mks

Compute dimensional viscosity in $kg/s \cdot m$ units using Sutherlands law.

```
double mu_raw( const double T ){
  double a = 1.458e-6, b = 110.6 ;
  return a * T * sqrt(T) / ( T + b );
}
```

18.9 $\kappa(\mu)$

Evaluate the thermal conductivity as a function of viscosity. Conductivity is related to viscosity through the Prandtl number, assumed to be constant.

$$P_r = \frac{\mu c_p}{\kappa}$$

In non-dimensional form,

$$\kappa^* = \frac{\mu^* c_p^*}{P_r}$$

$$\kappa^* = \frac{\kappa T_\infty}{\mu_\infty V_\infty^2}, \quad \mu^* = \frac{\mu}{\mu_\infty}, \quad c_p^* = \frac{\gamma R^*}{\gamma - 1}, \quad R^* = \frac{R T_\infty}{V_\infty^2}$$

```
double conductivity( const double mu )
{
  static double Pr = 0.72 ;
  double cp = ( GAMMA / GAMM1 ) * R_air * nondim_T / SQUARE( nondim_V );

  return mu * cp / Pr ;
}
```


18.10 $T(\rho, P)$

Return temperature given density and pressure. Using perfect gas equation of state.

$$T = \frac{P}{\rho R}$$

```
double T_from_rho_P( const double rho, const double P ){
  double R = R_air * nondim_T / SQUARE( nondim_V ) ;

  return P / ( rho * R );
}
```

18.11 $e(T)$

Compute internal energy as a function of temperature for a perfect gas wit constant specific heats.

$$e = c_v T = \frac{RT}{\gamma - 1} \quad \text{(dimensional)}$$

$$e^* = \frac{T^* T_\infty R}{(\gamma - 1) V_\infty^2} \quad \text{(non-dimensional)}$$

```
double e_from_T( const double T ){
  double R = R_air * nondim_T / SQUARE( nondim_V ) ;

  return T * R / GAMM1 ;
}
```

19 Transformations

A collection of functions to perform various variable transformations.

```
#include <math.h>        /*  sqrt, pow             */
#include "transform.h"
#include "basics.h"      /*  SQUARE, VEL2          */
#include "therm.h"       /*  e_from_rho_P, gammas  */
```

19.1 $V \to U$

Given a pointer to a vector of primitive variables, fill a vector of conserved variables.

$$U_0 = V_0 = \rho, \qquad U_1 = V_0 V_1 = \rho u, \qquad U_2 = V_0 V_2 = \rho v$$

$$U_3 = V_0 e(V_0, V_3) + \frac{1}{2} V_0 (V_1^2 + V_2^2) = \rho\, e(\rho, P) + \frac{1}{2}\rho(u^2 + v^2)$$

```
void primitive_to_conserved( double V[], double U[] )
{
  U[0] = V[0] ;                                          /* density      */
  U[1] = V[0] * V[1] ;                                   /* x-momentum   */
  U[2] = V[0] * V[2] ;                                   /* y-momentum   */
  U[3] = V[0]*( e_from_rho_P(V[0],V[3]) + VEL2(&V[1]) / 2 ); /* total energy */
}
```

19.2 $U \to Z$

Given a pointer to a vector of conserved variables, fill a vector of parameter variables.

$$U = (\rho, \rho u, \rho v, \rho E)^T, \quad Z = \sqrt{\rho}(1, u, v, H)^T$$

$$Z_0 = \sqrt{U_0}, \qquad Z_1 = \frac{U_1}{Z_0}, \qquad Z_2 = \frac{U_2}{Z_0}$$

$$Z_3 = \frac{U_3 + P(\rho, e)}{Z_0}$$

```
void conserved_to_parameter( double U[], double Z[] )
{
  double en ;     /*  internal energy  */

  if( safety && U[0] <= 0 ){
    printf("Negative density=%.3f, in U->Z, reset to small number\n", U[0] );
    U[0] = 1e-6 ;
  }
  Z[0] = sqrt( U[0] );
```



```
  Z[1] = U[1] / Z[0] ;
  Z[2] = U[2] / Z[0] ;
  en = ( U[3] - VEL2(&Z[1]) / 2 ) / U[0] ;
  Z[3] = ( U[3] + P_from_rho_e( U[0], en ) ) / Z[0] ;
}
```

19.3 $Z \to V$

Given a pointer to a vector of parameter variables, fill a vector of primitive variables.

$$Z = \sqrt{\rho}(1, u, v, H)^T, \qquad V = (\rho, u, v, P)^T$$

$$V_0 = Z_0^2, \quad V_1 = \frac{Z_1}{Z_0}, \quad V_2 = \frac{Z_2}{Z_0}$$

$$h = H - \frac{1}{2}(u^2 + v^2) = \frac{Z_3}{Z_0} - \frac{1}{2}(V_1^2 + V_2^2)$$

$$V_3 = P = P(\rho, e(h))$$

```
void parameter_to_primitive( double Z[], double V[] )
{
  if( safety && Z[0] <= 0 ){
    printf("Negative parameter0=%f, reset to a small number\n", Z[0] );
    Z[0] = 1e-6 ;
  }
  V[0] = SQUARE(Z[0]) ;
  V[1] = Z[1] / Z[0] ;
  V[2] = Z[2] / Z[0] ;
  V[3] = P_from_rho_e( V[0], e_from_h( Z[3] / Z[0] - VEL2(&V[1]) / 2) );
}
```

19.4 $U \to V$

Given a pointer to conserved variables, fill a vector of primitive variables.

$$U = (\rho, \rho u, \rho v, \rho E)^T, \quad V = (\rho, u, v, P)^T$$

```
void conserved_to_primitive( double U[], double V[] )
{
  V[0] = U[0] ;
  V[1] = U[1] / U[0] ;
  V[2] = U[2] / U[0] ;
  V[3] = P_from_rho_e( U[0], U[3] / U[0] - VEL2(&V[1]) / 2 );
}
```

19.5 $\Delta W/\Delta Z$

Evaluate the Jacobian derivative of the auxiliary variables with respect to the parameter vector. Send the parameter vector and return the Jacobian as,

$$\frac{dW}{dZ} = \sqrt{\rho}\begin{bmatrix} 2-\frac{H}{\gamma h} & \frac{u}{\gamma h} & \frac{v}{\gamma h} & -\frac{1}{\gamma h} \\ -u & 1 & 0 & 0 \\ -v & 0 & 1 & 0 \\ \frac{\gamma-1}{\gamma}H & -\frac{\gamma-1}{\gamma}u & -\frac{\gamma-1}{\gamma}v & \frac{\gamma-1}{\gamma} \end{bmatrix}$$

or,

$$\frac{dW}{dZ} = \begin{bmatrix} 2Z_0 - \frac{\gamma-1}{\gamma a^2}Z_3 & \frac{\gamma-1}{\gamma a^2}Z_1 & \frac{\gamma-1}{\gamma a^2}Z_2 & -\frac{\gamma-1}{\gamma a^2}Z_0 \\ -Z_1 & Z_0 & 0 & 0 \\ -Z_2 & 0 & Z_0 & 0 \\ \frac{\gamma-1}{\gamma}Z_3 & -\frac{\gamma-1}{\gamma}Z_1 & -\frac{\gamma-1}{\gamma}Z_2 & \frac{\gamma-1}{\gamma}Z_0 \end{bmatrix}$$

```
void dW_dZ( SolVec Z, double dw_dz[NEQS][NEQS] )
{
  double a2 ;  /*  sound speed squared  */
  Coord vel ;  /*  velocity vector      */
  double H  ;  /*  total enthalpy       */
  int i ;
  double f1, f2 ;  /*  temp factors     */

  H = Z[3] / Z[0] ;
  FOR(i,0,NDIMS) vel[i] = Z[i+1] / Z[0] ;
  a2 = pow( a_from_H_Vel( H, vel ), 2 );

  f1 = GAMM1 / GAMMA ;
  f2 = f1 / a2 ;

  dw_dz[0][0] = - f2 * Z[3] + 2 * Z[0] ;
  dw_dz[0][1] =   f2 * Z[1] ;
  dw_dz[0][2] =   f2 * Z[2] ;
  dw_dz[0][3] = - f2 * Z[0] ;
  dw_dz[1][0] = - Z[1] ;
  dw_dz[1][1] =   Z[0] ;
  dw_dz[1][2] =   0    ;
  dw_dz[1][3] =   0    ;
  dw_dz[2][0] = - Z[2] ;
  dw_dz[2][1] =   0    ;
  dw_dz[2][2] =   Z[0] ;
  dw_dz[2][3] =   0    ;
  dw_dz[3][0] =   f1 * Z[3] ;
  dw_dz[3][1] = - f1 * Z[1] ;
  dw_dz[3][2] = - f1 * Z[2] ;
  dw_dz[3][3] =   f1 * Z[0] ;
}
```


19.6 $\Delta U/\Delta W$

Evaluate the Jacobian of derivatives to convert from auxiliary variables to conserved variables,

$$U_W = \begin{bmatrix} 1 & 0 & 0 & \frac{1}{a^2} \\ u & 1 & 0 & \frac{u}{a^2} \\ v & 0 & 1 & \frac{v}{a^2} \\ \frac{V^2}{2} & u & v & \frac{1}{\gamma-1}\frac{T_0}{T} \end{bmatrix}$$

$$\frac{T_0}{T} = 1 + \frac{\gamma-1}{2}M^2$$

$$\frac{1}{\gamma-1}\frac{T_0}{T} = \frac{1}{\gamma-1} + \frac{V^2}{2a^2}$$

```
void dU_dW( double a, Coord V, double jac[NEQS][NEQS] )
{
  double a2 = 1 / SQUARE(a) ;

  jac[0][0] = 1 ;
  jac[0][1] = 0 ;
  jac[0][2] = 0 ;
  jac[0][3] = a2 ;
  jac[1][0] = V[0] ;
  jac[1][1] = 1 ;
  jac[1][2] = 0 ;
  jac[1][3] = V[0] * a2 ;
  jac[2][0] = V[1] ;
  jac[2][1] = 0 ;
  jac[2][2] = 1 ;
  jac[2][3] = V[1] * a2 ;
  jac[3][0] = VEL2(V) / 2 ;
  jac[3][1] = V[0] ;
  jac[3][2] = V[1] ;
  jac[3][3] = 1 / GAMM1 + a2 * VEL2(V) / 2 ;
}
```

19.7 $\nabla V = V_Z \nabla Z$

$$V_Z = \begin{bmatrix} 2z_0 & 0 & 0 & 0 \\ -\frac{z_1}{z_0^2} & \frac{1}{z_0} & 0 & 0 \\ -\frac{z_2}{z_0^2} & 0 & \frac{1}{z_0} & 0 \\ \frac{\gamma-1}{\gamma}z_3 & -\frac{\gamma-1}{\gamma}z_1 & -\frac{\gamma-1}{\gamma}z_2 & \frac{\gamma-1}{\gamma}z_0 \end{bmatrix}$$

```
void gradZ_to_gradV( SolVec gradZ[NDIMS], SolVec gradV[NDIMS], SolVec z ){
  double dV_dZ[NEQS][NEQS];
  double fac = GAMM1 / GAMMA;
```

```
  int i, j, n;

  dV_dZ[0][0] = 2 * z[0];
  dV_dZ[0][1] = 0;
  dV_dZ[0][2] = 0;
  dV_dZ[0][3] = 0;
  dV_dZ[1][0] = - z[1] / SQUARE( z[0] );
  dV_dZ[1][1] = 1 / z[0];
  dV_dZ[1][2] = 0;
  dV_dZ[1][3] = 0;
  dV_dZ[2][0] = - z[2] / SQUARE( z[0] );
  dV_dZ[2][1] = 0;
  dV_dZ[2][2] = 1 / z[0];
  dV_dZ[2][3] = 0;
  dV_dZ[3][0] =  fac * z[3];
  dV_dZ[3][1] = -fac * z[1];
  dV_dZ[3][2] = -fac * z[2];
  dV_dZ[3][3] =  fac * z[0];

  FOR(n,0,NDIMS){
    FOR(i,0,NEQS){
      gradV[n][i] = 0;
      FOR(j,0,NEQS) gradV[n][i] += dV_dZ[i][j] * gradZ[n][j];
    }
  }
}
```


20 Triangle math functions

Bundle together some functions that are handy for working with triangles.

```
#include "TriMath.h"
#include "blas.h"


            /*                                  */
            /*  Linear average of three vectors  */
            /*                                  */
void TriaveV( int n, double x1[], double x2[], double x3[], double avg[] )
{
  register int i ;
  FOR(i,0,n) avg[i] = Triave( x1[i], x2[i], x3[i] ) ;
}

            /*                                                  */
            /*  Area of a triangular cell                       */
            /*  send pointer to array of pointers to 3 nodes    */
            /*                                                  */
double tri_cell_area( Node *n[] )
{
  int node ;
  register int coord ;
  double x[3][NDIMS] ;

  FOR(node,0,3){
    FOR(coord,0,NDIMS) x[node][coord] = n[node]->geom.coord[coord] ;
  }
  return ( (x[2][0] - x[1][0]) * (x[0][1] - x[1][1])
           - (x[2][1] - x[1][1]) * (x[0][0] - x[1][0]) ) / 2 ;
}

           /*                                                          */
           /*  Calculate median-dual area about all nodes               */
           /*    first zero out all dual areas                          */
           /*    then distribute 1/3 area of each triangle to its nodes  */
           /*                                                          */
void median_dual_all( void )
{
  Node *node ;
  Cell *cell ;
  register int n ;

  NODE_LOOP_FORWARD node->geom.dual_area = 0 ;
  CELL_LOOP_FORWARD FOR(n,0,3)
    (cell->geom.connect.nodes[n])->geom.dual_area += cell->geom.metric.area/3;
}
```

```
            /*                                          */
            /*  Calculate median-dual area about one node  */
            /*    loop on cells connected to this node,    */
            /*      summing 1/3 cell areas                 */
            /*                                          */
double median_dual_one( const Node *n )
{
  double dual_area = 0 ;
  Node2Cell *ptr = n->geom.cellcon ;

  if( ptr ){
    do dual_area += (ptr->cell)->geom.metric.area / 3 ;
    while( (ptr = ptr->next) );
  }
  else{
    puts("Failure, no cells attached to this node, stopping");
    printf("Nodal x=%f, y=%f\n", n->geom.coord[0], n->geom.coord[1] );
  }
  return dual_area ;
}

/*                                                                  */
/*    Compute cell-area times gradient for linear variation over triangle  */
/*    Send nq dimensions of vector u, that varies linearly over triangle   */
/*         u[3][nq] nodal vectors                                   */
/*         xy[3][2] x,y values of the 3 nodes                       */
/*    Return vector of gradients of u components, grad_u[2][nq]     */
/*    S * grad_u = 0.5 * SUM( u_i * n_i )                           */
/*      n_i is inward scaled normal opposite to node i              */
/*                                                                  */
void tri_grad( int nq, double u[3][NEQS], double xy[3][2],
               double grad_u[2][NEQS] )
{
  double n[3][2] ;     /*  inward scaled normal opposite to each node / 2  */
  int i, j, jp, jm, k ;

  FOR(j,0,3){
    jp = cyclic_plus0(j) ;
    jm = cyclic_minus0(j) ;
    n[j][0] = ( xy[jp][1] - xy[jm][1] ) / 2 ;
    n[j][1] = ( xy[jm][0] - xy[jp][0] ) / 2 ;
  }

  FOR(i,0,nq) FOR(j,0,2)
    for( k=0, grad_u[j][i]=0 ; k<3 ; ++k )
      grad_u[j][i] += u[k][i] * n[k][j] ;
}

/*                                                                  */
```



```
/*  compute the split-point of a triangle for a containment dual         */
/*  the split point is either the centroid of the circle inscribing      */
/*      the triangle for acute triangles or the midpoint of the longest  */
/*      edge for obtuse triangles                                        */
/*                                                                       */
void containment_dual( Cell *cell ){
  double a = .25 / cell->geom.metric.area ;
  double b, L[3], dx, dy ; /*  L are squares of lengths opposite edge i  */
  int i ;
  double *x = cell->geom.metric.split_containment ;

         /*  compute the centroid of the circle inscribing the triangle  */
  for( i=0, x[0]=x[1]=0 ; i<3 ; ++i ){
    b = SQUARE( (cell->geom.connect.nodes[i])->geom.coord[0] ) +
        SQUARE( (cell->geom.connect.nodes[i])->geom.coord[1] ) ;
    dx = (cell->geom.connect.nodes[cyclic_plus0(i)])->geom.coord[0]
       - (cell->geom.connect.nodes[cyclic_minus0(i)])->geom.coord[0] ;
    dy = (cell->geom.connect.nodes[cyclic_plus0(i)])->geom.coord[1]
       - (cell->geom.connect.nodes[cyclic_minus0(i)])->geom.coord[1] ;
    x[0] += b * dy ;
    x[1] -= b * dx ;
    L[i] = SQUARE( dx ) + SQUARE( dy );
  }
  FOR(i,0,NDIMS) x[i] *= a ;

                                          /*  check for obtuse triangle  */
  FOR(i,0,3) if( L[i] >= L[cyclic_plus0(i)] + L[cyclic_minus0(i)] ){
      x[0] = ( (cell->geom.connect.nodes[cyclic_plus0(i)])->geom.coord[0] +
               (cell->geom.connect.nodes[cyclic_minus0(i)])->geom.coord[0] )/2;
      x[1] = ( (cell->geom.connect.nodes[cyclic_plus0(i)])->geom.coord[1] +
               (cell->geom.connect.nodes[cyclic_minus0(i)])->geom.coord[1] )/2;
  }
}

/*                                      */
/*  compute cross product of 2 edges    */
/*                                      */
double cross_edges( Edge *edge1, Edge *edge2 ){
  return edge1->r[0] * edge2->r[1] - edge1->r[1] * edge2->r[0];
}

/*                                      */
/*  normalized cross product of 2 edges  */
/*                                      */
double norm_cross_edges( Edge *edge1, Edge *edge2 ){
  double denom;
  denom = dnrm2(NDIMS,edge1->r,1) *dnrm2(NDIMS,edge2->r,1);
  return fabs(cross_edges(edge1,edge2)) /denom;
}
```

21 West World

The future is upon us, so close out the program.

```
#include <stdio.h>  /*  FILE                    */
#include <stddef.h> /*  NULL                    */
#include <time.h>   /*  time_t, time, ctime     */
#include <stdlib.h> /*  free                    */
#include <math.h>   /*  pow, sin, cos           */
#include <string.h> /*  strcpy, strcat          */
#include "basics.h" /*  Node, Cell, axisym, AOA  */
#include "transform.h"  /*  U2V                 */

void WestWorld( void )
{
  Node *node, *max_node = NULL, **nodal_stack ;
  Cell *cell ;
  char mode[3] = "w";
  FILE *outsol  = open_file( insolf,  mode );
  FILE *outgrid = open_file( ingridf, mode );
  time_t now ;
  double Cdi, Cdv, Cli, Clv, Qtot ;
  float max_norm = 0, next_norm = 0 ;
  int i;
  if( traceback > 1 ) puts("The robots have gone awry");
```

Write final grid and solution to TECPLOT[Tec96] file.

```
  write_tecplot_0();
```

Output solution as restart file.

```
  fprintf( outsol, " %u\n", nodes );
  NODE_LOOP_FORWARD{
    conserved_to_primitive( node->vars.conserved, node->vars.primitive );
    fprintf( outsol, " %.9e %.9e %.9e %.9e %.9e %.9e %.9e %.9e %d\n",
             node->vars.primitive[0] * nondim_rho,
             node->vars.primitive[1] * nondim_V,
             node->vars.primitive[2] * nondim_V,
             node->vars.primitive[3] * nondim_rho * SQUARE(nondim_V),
             node->vars.wall[0] * nondim_T ,
             node->vars.wall[1] * nondim_rho * SQUARE(nondim_V),
             node->vars.wall[2] * nondim_rho * SQUARE(nondim_V),
             node->vars.wall[3] * nondim_rho * pow( nondim_V, 3 ),
             node->bc );
  }
  fclose( outsol );
```


Re-write the grid file, just in case we adapted it.

```
  fprintf( outgrid, "VARIABLES = \"X\" \"Y\"\n" );
  fprintf( outgrid, "ZONE N= %u E= %u F=FEPOINT ET=TRIANGLE\n", nodes, cells );
  NODE_LOOP_FORWARD{
    FOR(i,0,2) fprintf( outgrid, " %.9e", node->geom.coord[i] * nondim_L );
    fprintf( outgrid, "\n");
  }
  nodal_stack = allocate_node_ptr( nodes+1 );
  stack_nodes( nodal_stack );
  CELL_LOOP_FORWARD{
    FOR(i,0,3) fprintf( outgrid, " %u",
                        cell->geom.connect.nodes[i]->links.node_number );
    fprintf( outgrid, "\n" );
  }
  free( nodal_stack );
  fclose( outgrid );
```

Compute aerodynamics.

```
  Euler_Aero( &Cdi, &Cli );
  Stokes_Aero( &Cdv, &Clv, &Qtot );
  printf("\nDrag Coefficient = %f, based on ", Cdi + Cdv );
  puts( axisym ? "pi L^2" : "L * unit depth" );
  printf("Lift Coefficient = %f, based on ", Cli + Clv );
  puts( axisym ? "pi L^2" : "L * unit depth" );
  printf("Total heat transfer rate = %f\n", Qtot );
```

Output the surface quantities, if we have a viscous wall.

```
  if( Cdv || Clv || Qtot ) write_visc_wall();
```

Output the node with the largest residual.

```
  if( safety && itr_done ){
    NODE_LOOP_FORWARD{
      next_norm = (float) ddot( NEQS, node->update.dU, 1, node->update.dU, 1 );
/*  printf("Node %.4d, norm= % .6f\n", node->links.node_number, next_norm ); */
      if( next_norm < 0 )
        Aetna_node( node->links.node_number, "\n** Negative resid!");
      if( next_norm > max_norm ){
        max_norm = next_norm ;
        max_node = node ;
      }
    }
```

```
    Aetna_node( max_node->links.node_number, "\nMax resid at this node");
  }
```

Done with the program.

```
  now = time(NULL) ;                                  /*  program end time  */
  printf("\n   Ending program %s\n\n", ctime(&now) );
}
```

21.1 TECPLOT Solution File

Write grid and solution to TECPLOT[Tec96] file. Open the file if this is the first time through. Set the name of the file using the -p command-line option. Defaults to `root.dat`.

Currently writes non-dimensional values.

```
void write_tecplot_0( void )
{
  static FILE *outplot = NULL ;
  register Node *node ;
  register Cell *cell ;
  Node **nodal_stack ; /* not directly used, but by-product of stack_nodes */
  int i ;
  char mode[3] = "w" ;

  if( ! outplot ){
    outplot = open_file( outplotf, mode );
    fprintf( outplot, "VARIABLES = \"X\" \"Y\" "
             "\"rho\" \"u\" \"v\" \"P\" "
             "\"U1\" \"U2\" \"U3\" \"U4\" "
             "\"Tw\" \"tau-x\" \"tau-y\" \"Qw\" "
             "\"AD1\" \"AD2\" \"AD3\" \"AD4\" "
/*              "\"Dual area\" " */
     "\"Error estimate\" "
     "\"BC type\"" );
    fprintf( outplot, "\n" );
  }

  fprintf( outplot, "ZONE N=%u E=%u F=FEPOINT ET=TRIANGLE\n", nodes, cells );

  /*       Nodes        */
  NODE_LOOP_FORWARD{
    FOR(i,0,2) fprintf( outplot, " %f", node->geom.coord[i] );
    fprintf( outplot, "\n" );
    conserved_to_primitive( node->vars.conserved, node->vars.primitive );
    FOR(i,0,4) fprintf( outplot, " %f", node->vars.primitive[i] );
    fprintf( outplot, "\n" );
    FOR(i,0,4) fprintf( outplot, " %f", node->vars.conserved[i] );
    fprintf( outplot, "\n" );
```



```
    FOR(i,0,4) fprintf( outplot, " %f", node->vars.wall[i] );
    fprintf( outplot, "\n" );
    FOR(i,0,4) fprintf( outplot, " %f", node->update.art_diss[i] );
/*     fprintf(outplot, "\n %f ", node->geom.dual_area); */
    fprintf(outplot, "\n %f ", node->adapt.error);
    fprintf(outplot, "%d\n", node->bc );
  }

  /*       Cells, requires numbering the nodes    */
  nodal_stack = allocate_node_ptr( nodes+1 );
  stack_nodes( nodal_stack );
  CELL_LOOP_FORWARD{
    FOR(i,0,3) fprintf( outplot, " %u",
                        cell->geom.connect.nodes[i]->links.node_number );
    fprintf( outplot, "\n" );
  }
  free( nodal_stack );
  (void) fflush( outplot );
}
```

21.2 Viscous Wall Plot File

Open a TECPLOT data file and write viscous wall nodes as ordered, but unconnected, list. Only want to initialize the TECPLOT header if the file doesn't previously exist. Otherwise, append this solution to the file.

```
void write_visc_wall( void )
{
  char fname[23] ;   /*  output file name  */
  FILE *fout ;
  Node *node ;
  int i, count = 0 ;
  long int end_location ;

  (void) strncpy( fname, outplotf, strlen( outplotf ) - 4 );
  fname[strlen(outplotf)-4] = '\0' ;
  (void) strcat( fname, "_w.dat" );

  NODE_LOOP_FORWARD if( node->bc == wall_viscous ) ++count ;
  if( count ){                      /*  only do this if we have viscous walls  */
    fout = open_file( fname, "a" );

                                 /*  if end = start of file, write header info  */
    end_location = ftell( fout );
    rewind( fout );
    if( end_location == ftell( fout ) )
      fprintf( fout, "VARIABLES = \"X\" \"Y\" "
                     "\"rho\" \"u\" \"v\" \"P\" "
```

```
                    "\"Tw\" \"tau-x\" \"tau-y\" \"Qw\" "
                    "\"BC type\"\n" );

                         /*  now go to end of file and write current zone  */
    (void) fseek( fout, OL, SEEK_END );
    fprintf( fout, "ZONE I= %d F=POINT\n", count );

    NODE_LOOP_FORWARD if( node->bc == wall_viscous ){
      FOR(i,0,2) fprintf( fout, " %f", node->geom.coord[i] );
      fprintf( fout, "\n" );
      conserved_to_primitive( node->vars.conserved, node->vars.primitive );
      FOR(i,0,4) fprintf( fout, " %f", node->vars.primitive[i] );
      fprintf( fout, "\n" );
      FOR(i,0,4) fprintf( fout, " %f", node->vars.wall[i] );
      fprintf( fout, "\n %d\n", node->bc );
    }
  }
}
```

21.3 Convergence History Plot

Send the convergence history to a TECPLOT file. Set the name of the file using the `-c` command-line option. Defaults to `root_h.dat`.

```
void write_hist( int itr, time_t cpu, float L2, int restart )
{
  static FILE *histplot = NULL ;
  char mode[3] = "w", line[121] ;
  double Cdi, Cdv, Cli, Clv, Qtot ;
  static int itr_lastrun = 0, cpu_lastrun = 0 ;

  if( ! histplot ){
    if( restart && ( histplot = fopen( histplotf, "r+" ) ) ){
      while( fgets( line, 121, histplot ) );
      sscanf( line, "%d %d %f", &itr_lastrun, &cpu_lastrun, &L2 );
    }
    else{
      histplot = open_file( histplotf, mode );
      fprintf( histplot,
               "VARIABLES = \"Iteration\" \"CPU sec\" \"L2-resid\"\n"
               "\"Cd_i\" \"Cd_v\" \"Cd\" \"Cl_i\" \"Cl_v\" \"Cl\" \"Q\"\n");
    }
  }

  Euler_Aero( &Cdi, &Cli );
  Stokes_Aero( &Cdv, &Clv, &Qtot );
  fprintf( histplot, "%d %d %.3e %.4e %.4e %.4e %.4e %.4e %.4e\n",
           itr + itr_lastrun, (int) cpu + cpu_lastrun,
```



```
           L2, Cdi, Cdv, Cdi+Cdv, Cli, Clv, Cli+Clv, Qtot );
}
```

21.4 Inviscid Aerodynamics

Compute lift and drag coefficients due to pressure on all wall boundaries.

```
void Euler_Aero( double *cd, double *cl ){
  SolVec nv0, nv1 ;
  int i ;
  register Bound_edge *bedge ;
  double cx = 0, cy = 0 ;

  BEDGE_LOOP_FORWARD
    if( bedge->bc == wall_inviscid || bedge->bc == wall_viscous ){
      FOR(i,0,NEQS) nv0[i] = 0.5 * ( (bedge->node[0])->vars.conserved[i] +
                                     (bedge->node[1])->vars.conserved[i] );
      conserved_to_primitive( nv0, nv1 );

      cx += 2 * nv1[3] * ( pow( (bedge->node[0])->geom.coord[1], 1+axisym ) -
                           pow( (bedge->node[1])->geom.coord[1], 1+axisym ) );
      cy += axisym ? 0 : 2 * nv1[3] *
        ( (bedge->node[1])->geom.coord[0] - (bedge->node[0])->geom.coord[0] );
    }
  if( axisym ){
    *cd = cx ;
    *cl = 0 ;
  }
  else{
    *cd = cos(AOA) * cx + sin(AOA) * cy ;
    *cl = cos(AOA) * cy - sin(AOA) * cx ;
  }
}
```

21.5 Viscous Aerodynamics

Compute skin-friction components to lift and drag coefficients. Compute total heat transfer.

```
void Stokes_Aero( double *cd, double *cl, double *Q ){
  register Bound_edge *bedge ;
  Coord tau, dl ;
  int i ;
  double cx = 0, cy = 0, len, aq ;
  *Q = 0 ;

  BEDGE_LOOP_FORWARD
```

```
    if( bedge->bc == wall_viscous ){
      FOR(i,0,NDIMS) tau[i] = 0.5 * ( (bedge->node[0])->vars.wall[i+1] +
                                      (bedge->node[1])->vars.wall[i+1] );

      cx += 2 * tau[0] * ( (bedge->node[0])->geom.coord[0] -
                           (bedge->node[1])->geom.coord[0] )
                       * ( !axisym ? 1: (bedge->node[0])->geom.coord[1] +
                                        (bedge->node[1])->geom.coord[1] );

      cy += axisym ? 0 : 2 * tau[1] *
        ( (bedge->node[0])->geom.coord[1] - (bedge->node[1])->geom.coord[1] );

      aq = ( (bedge->node[0])->vars.wall[3] + (bedge->node[1])->vars.wall[3] )
           / 2 ;
      FOR(i,0,NDIMS) dl[i] = (bedge->node[1])->geom.coord[i]
                           - (bedge->node[0])->geom.coord[i] ;
      len = VEL(dl) ;
      *Q += aq * len * nondim_rho * pow( nondim_V, 3 );
    }
  if( axisym ){
    *cd = cx ;
    *cl = 0 ;
  }
  else{
    *cd = cos(AOA) * cx + sin(AOA) * cy ;
    *cl = cos(AOA) * cy - sin(AOA) * cx ;
  }
}
```


22 Local Include Files

22.1 adaption.h

```
/*

 */

#ifndef ADAPTION_head
#define ADAPTION_head

enum{ none=0, edge_swap_H, delete_H, add_H, move_H,
edge_swap_FM, delete_FM, add_FM, move_FM } mesh_adapt;

extern int    bound_point_delete( Node *node );
extern void   build_edges( Node *end_nodes[2], Cell *cell );
extern void   edge_errors( void );
extern double error_estimate( Node *node[2], Coord r01 );
extern double error_fluc( Cell *cell_left, Cell *cell_right );
extern void   error_swap( Edge *edge );
extern void   FM_movement( Node *node, Coord R );
extern void   global_error_delete( void );
extern void   global_error_insert( void );
extern void   global_error_move( void );
extern void   global_error_swap( void );
extern int    move_node( Node *node, Coord r );
extern void   point_delete( Node *node );
extern int    quad_point_delete( Node *node );
extern int    querry_swap( Edge *edge );
extern void   split_edge( Edge *edge );
extern void   swap_edge( Edge *edge_to_swap );
extern void   tri_point_delete( Node *node );
extern void   viscous_fluc( Cell *cell, SolVec fluctuation );

#endif  /*  ADAPTION_head  */
/*

 */
```

22.2 basics.h

```
/*

 */
/*  general over-all code parameters  */

#ifndef BASICS_head
#define BASICS_head

#include <stdio.h>  /*  FILE            */
#include "blas.h"   /*  ddot, dnrm2     */
```

```
#include <time.h>   /*  time_t          */
#include "bc.h"     /*  Bc, Bound_edge  */

/*  Macros  */

#define NDIMS 2                 /*  Number of independent dimensions  */
#define NEQS  4                 /*  Number of dependent equations     */

#define MIN(a,b) ((a)<(b)?(a):(b))
#define MAX(a,b) ((a)>(b)?(a):(b))
#define SQUARE(x) ((x)*(x))

#define FOR(i,m,n) for( (i)=(m) ; (i)<(n) ; ++(i) )
#define NODE_LOOP_FORWARD for( node=top_node; node; node=node->links.next_node)
#define NODE_LOOP_BACKWARD \
        for(node=bottom_node; node; node=node->links.previous_node)
#define CELL_LOOP_FORWARD for( cell=top_cell; cell; cell=cell->links.next_cell)
#define BEDGE_LOOP_FORWARD for( bedge=top_bedge; bedge; bedge=bedge->next)
#define EDGE_LOOP_FORWARD for( edge=top_edge; edge; edge=edge->next_edge)
#define CELL_LOOP_AT_NODE(n) \
        for( celln=(n)->geom.cellcon ; celln ; celln=celln->next )
#define EDGE_LOOP_AT_NODE(n) \
        for( edgen=(n)->geom.edgecon ; edgen ; edgen=edgen->next )

#define VEL(v) dnrm2(NDIMS,(v),1)
#define VEL2(v) ddot(NDIMS,(v),1,(v),1)

/*  Data Classes  */

/*  file names  */
extern char rootf[]    ;    /*  root file name           */
extern char inpf[]     ;    /*  input deck file name     */
extern char ingridf[]  ;    /*  input grid file name     */
extern char insolf[]   ;    /*  input solution file name */
extern char inrstf[]   ;    /*  restart file name        */
extern char outplotf[] ;    /*  plot file name           */
extern char histplotf[];    /*  history plot file name   */

/*  node  */
typedef double Coord[NDIMS];
typedef double SolVec[NEQS];
typedef struct N2C { struct scell *cell; struct N2C *next;} Node2Cell;
typedef struct N2E { struct sedge *edge; struct N2E *next;} Node2Edge;
typedef struct { Coord coord; double dual_area, bound_face;
                 Node2Cell *cellcon; Node2Edge *edgecon;} NGeom;
typedef struct { SolVec conserved, primitive, parameter, wall;} Vars;
typedef struct { SolVec source, source_fs, flux, art_diss, viscous, wall, dU;
                 double dt;} Update;
typedef struct { SolVec gradx, grady, psi;} NFVol;
typedef struct snode
```



```
        { NGeom geom; Vars vars; Bc bc; Update update; NFVol fv;
          struct { struct snode *previous_node, *next_node;
          unsigned int node_number;} links;
          struct { double error;} adapt;} Node;
extern Node *top_node    ;        /*  first node in the list  */
extern Node *bottom_node ;        /*  last node in the list   */
extern unsigned int nodes;        /*  number of nodes         */

/*  cell  */
typedef struct { double area; Coord split_point, split_containment;} CMetric;
typedef struct { Node *nodes[3];} CConnect;
typedef struct { CMetric metric; CConnect connect;} CGeom;
typedef struct { SolVec parameter, primitive; double H, a;} CTilde;
typedef struct scell
        { CGeom geom; CTilde tilde;
          struct { struct scell *previous_cell, *next_cell;} links;} Cell;
extern Cell *top_cell    ;        /*  first cell in the list  */
extern Cell *bottom_cell ;        /*  last cell in the list   */
extern unsigned int cells;        /*  number of cells         */

/*  edge  */
typedef struct sedge
        { Cell *cell_left, *cell_right; Node *node[2]; double error;
          Coord r; struct sedge *previous_edge, *next_edge;} Edge;
extern Edge *top_edge    ;        /*  first edge in the list  */
extern Edge *bottom_edge ;        /*  last edge in the list   */
extern unsigned int edges;        /*  number of edges         */

/*  control parameters  */
extern int    axisym          ; /*  axisymmetric flag                      */
extern int    memory          ; /*  memory reduction level 0,1,2           */
extern int    safety          ; /*  double-check flag                      */
extern int    scalar          ; /*  scalar flag                            */
extern int    solver          ; /*  FS(0) or FV(1)                         */
extern int    viscous         ; /*  inviscid(0), TLNS median dual(1),      */
                                /* TLNS containment dual(2), NSMD(3), NSCD(4)*/
extern int    traceback       ; /*  traceback level 0,1,2,3                */
extern float  CFL             ; /*  timestep factor                        */
extern float  CFL_v           ; /*  viscous timestep factor                */
extern float  L2_done         ; /*  stop criteria for norm                 */
extern int    itr_done        ; /*  stop criteria for iterations           */
extern time_t cpu_done        ; /*  maximum run time in seconds            */
extern int    hist_skip_screen; /*  history iteration skip factor to screen  */
extern int    hist_skip_file  ; /*  history iteration skip factor to file    */
extern int    plot_skip       ; /*  iteration skip factor for plot file    */
extern double swap_thresh     ; /*  edge-swapping trigger threshold        */
extern double delete_thresh   ; /*  point-deletion trigger threshold       */
extern double add_thresh      ; /*  point-insertion trigger threshold      */
extern double move_thresh     ; /*  node-movement trigger threshold        */
extern double adapt_target    ; /*  target adaption error level            */
```

```
/*  non-dimensionalizing  */
extern double nondim_L   ;  /*  length                    */
extern double nondim_V   ;  /*  velocity                  */
extern double nondim_T   ;  /*  temperature               */
extern double nondim_rho ;  /*  density                   */
extern double nondim_mu  ;  /*  viscosity                 */
extern double T_wall     ;  /*  cold-wall temperature     */
extern float  AOA        ;  /*  angle of attack           */
extern float  Mach       ;  /*  freestream Mach number    */
extern float  Reynolds   ;  /*  freestream Reynolds number  */

/*  Functions  */
extern void Stonehenge( int argc, char *argv[] );
extern FILE *open_file( char *filename, char *mode );
extern Node *add_node( void );
extern Cell *add_cell( void );
extern Edge *add_edge( void );
extern void stack_nodes( Node *nodal_stack[] );
extern void WestWorld( void );
extern void write_tecplot_0( void );
extern void write_visc_wall( void );
extern void write_hist( int itr, time_t cpu, float L2, int restart );
extern void Euler_Aero( double *cd, double *cl );
extern void Stokes_Aero( double *cd, double *cl, double *Q );
extern void hook_from_node_to_cell( Node *node, Cell *cell );
extern void break_cell_from_node( Node *node, Cell *cell );
extern void hook_from_node_to_edge( Node *node, Edge *edge );
extern void break_edge_from_node( Node *node, Edge *edge );
extern Node **allocate_node_ptr( unsigned int num );
extern Bound_edge *add_bedge( void );
extern void build_bedges( Node *n0, Node *n1, int safe );
extern void Aetna( int choice, char *message );
extern void Aetna_node( unsigned int bad_boy, char message[80] );
extern void Aetna_cell( Cell *cell );
extern void Aetna_edge( Edge *edge );
extern void kill_edge( Edge *edge );
extern void kill_cell( Cell *cell );
extern void kill_node( Node *node );
extern double max_cell_angle( Cell *cell );

#endif   /*  defining BASICS_head  */
/*

 */
```

22.3 bc.h

```
/*

 */
```



```
/*  Header file for Boundary Conditions  */

#ifndef BC_head
#define BC_head

/*  Macros  */

/*  Mach number ratio for second freestream Mach stream  */
#define BC_Mach_ratio ( 2.3 / 1.8 )
/*  Pressure ratio for second freestream Mach stream  */
#define BC_pressure_ratio 1.0812

/*  Data classes  */

/*   boundary conditions  */
/*   change here, and in build_bedges, FV_distrib_boundary  */
typedef enum { interior=0,
               freestream_nondim=10, freestream_stagnant,
                  freestream_second, extrapolation, vacuum,
               wall_inviscid=20, wall_viscous,
               invalid = 30 } Bc ;

/*   boundary edges  */
typedef struct sbound_edge
        { struct sbound_edge *prior, *next ; struct snode *node[2] ; Bc bc ;}
        Bound_edge  ;
extern Bound_edge *top_bedge    ;  /*  first boundary edge in list */
extern Bound_edge *bottom_bedge ;  /*  last boundary edge in list  */
extern unsigned int bedges      ;  /*  number of boundary edges    */

#endif  /*  defining BC_head  */
/*

 */
```

22.4 blas.h

```
/*

 */
/*  definitions for ddot, dnrm2  */

/*  if BLAS does work, set to 0    */
/*  if BLAS not working, set to 1  */

#define BLAS_WORKS 1      /*  0 = cblas libraries exist                */
#define SGI_BLAS 0        /*  are these the screwed-up SGI libraries?  */

#ifndef BLAS_head
#define BLAS_head
```

```
#if BLAS_WORKS==0          /*  use the BLAS library  */
#include <cblas.h>

#if SGI_BLAS               /*  change the BLAS names  */
#define ddot cblas_ddot
#define dnrm2 cblas_dnrm2
#endif

#else                      /*  use my own function definitions  */
#include <math.h>

double ddot( int length, double *x, int xskip, double *y, int yskip );
double dnrm2( int length, double *x, int xskip );

#endif  /*  BLAS_WORKS  */

#endif  /*  BLAS_head  */
/*

 */
```

22.5 lhs.h

```
/*

 */
/*   LHS header file  */

#ifndef LHS_head
#define LHS_head

#include "basics.h"  /*  Node  */

/*  Functions  */
extern float Bronowski( Node *node, double t );
extern double dt_Euler_exp( Node *node );
extern double sync_time( void );

#endif   /*  LHS_head  */
/*

 */
```

22.6 limiter.h

```
/*

 */
/*  Header file for limiters  */
```



```
#ifndef LIMITER_head
#define LIMITER_head

/*  Macros  */
#define COMPRESS 2    /*  maximum value of limiter  */

/*  Data types  */
enum { first_order, minmod, superbee, compress, van_leer, van_albada,
       unlimited } limiter_type ;

/*  Functions  */
extern double limiter( double P, double Q, double epsVA );
extern double eigenvalue_limiter( double vL, double v, double vR );

#endif  /*  LIMITER_head  */
/*

 */
```

22.7 rhs.h

```
/*

 */
/*  Header file for RHS evaluation  */
#ifndef RHS_head
#define RHS_head

#include "basics.h"   /*  Cell  */

/*  Data classes  */
enum { mass_lumped, averaged, linearP, linearZ } axi_source_type ;

/*  Functions  */
extern void gensource( double area, double x[], double y[], double distrib[] );
extern void axi_source( Cell *cell, double distrib[NEQS][3] );
extern void FS_flux_source( Cell *cell, double distrib[NEQS][3] );

extern void Roe_average_cell( Cell *cell );
extern void Roe_average_edge( SolVec Zl, SolVec Zr, SolVec Vtilde,
                              double *Htilde_p, double *atilde_p );

extern void node_gradients( int reconstruct );
extern void barth_grad( Node *j1, Node *j2, SolVec j1U, SolVec j2U,
double l11[], double l12[], double l22[],
double lf1[][4], double lf2[][4] );
extern void fv_limit_node( int reconstruct, Node *node, SolVec psi );
extern void quadrature_point( double x0[], double x1[],
                              double SP[], double QP[] );
```

```
extern void FV_distrib_cell( Cell *cell );
extern void FV_distrib_boundary( Bound_edge *bedge );
extern void FV_art_diss( Coord x0, Coord x1, Coord SP,
                         SolVec u0, SolVec gx0, SolVec gy0, SolVec psi0,
                         SolVec u1, SolVec gx1, SolVec gy1, SolVec psi1,
                         SolVec dist0, SolVec dist1 );
extern void FV_edge_central( Coord x0, Coord x1, Coord SP,
                             SolVec u0, SolVec gx0, SolVec gy0, SolVec psi0,
                             SolVec u1, SolVec gx1, SolVec gy1, SolVec psi1,
                             SolVec dist0, SolVec dist1, void (*flux)() );

extern void FS_distrib_cell( Cell *cell );
extern void nA_dW( Coord n, double vn, double a, SolVec dw, SolVec prod );
extern void abs_wavespeed( double V, double a, Coord n, double M[NEQS][NEQS] );
extern void cell_fluc( Node *n[3], SolVec phi );
extern void FS_distrib_boundary( Bound_edge *bedge );
extern int  limited_wavespeed( Coord n, double vl, double al,
       double vc, double ac, double vr, double ar,
       double matrix[NEQS][NEQS] );

extern void Euler_flux_conserved( SolVec U, double F[NEQS][NDIMS] );
extern void scalar_flux( SolVec U, double F[NEQS][NDIMS] );
extern void dF_dZ( SolVec z, double df_dz[2][NEQS][NEQS] );

extern void Haselbacher( Cell *cell, double distrib[NEQS][3] );
extern void full_viscous( Cell *cell, double distrib[NEQS][3] );
extern void Hasel_bc( Bound_edge *bedge );

#endif  /* defining RHS_head  */
/*

 */
```

22.8 therm.h

```
/*

 */
/*  Header file for thermodynamic functions  */

#ifndef THERMO_head
#define THERMO_head

/*  Macros  */
#define R_air           287.05  /*  air gas constant, J/kg K            */
#define GAMMA           1.4     /*  gamma for diatomic gas              */
#define GAMM1     (GAMMA-1)     /*  gamma minus 1                       */
#define GAMP1     (GAMMA+1)     /*  gamma plus 1                        */
#define T_min_perfect   30      /*  temperature for real gas effects, K  */
#define RHO_max_perfect  9      /*  density for real gas effects, kg/m3  */
```



```
#define T_max_caloric_perf  800 /*  calorically perfect limit, K        */

/*  Functions  */
extern double P_from_rho_T( const double rho, const double T );
extern double P_from_rho_e( const double rho, const double e );
extern double e_from_rho_P( const double rho, const double P );
extern double e_from_h( const double h );
extern double e_from_T( const double T );
extern double T_from_rho_P( const double rho, const double P );
extern double a_from_H_Vel( const double H, double V[] );
extern double a_from_rho_P( const double rho, const double P );
extern double mu_from_T( const double T );
extern double mu_raw( const double T );     /*  dimensional viscosity (mks)  */
extern double conductivity( const double mu );

#endif   /*  defining  THERMO_head  */
/*

 */
```

22.9 transform.h

```
/*

 */
/*  Header file for performing transformations  */

#ifndef TRANSFORM_head
#define TRANSFORM_head

#include "basics.h"  /*  SolVec, NEQS  */

/*  Functions  */
extern void primitive_to_conserved( double V[], double U[] );
extern void conserved_to_parameter( double U[], double Z[] );
extern void conserved_to_primitive( double U[], double V[] );
extern void parameter_to_primitive( double Z[], double V[] );
extern void dW_dZ( SolVec Z, double dw_dz[NEQS][NEQS] );
extern void dU_dW( double a, Coord V, double jac[NEQS][NEQS] );
extern void gradZ_to_gradV( SolVec gradZ[NDIMS], SolVec gradV[NDIMS], SolVec Z);

#endif   /*  defining TRANSFORM_head  */
/*

 */
```

22.10 TriMath.h

```
/*

 */
```

```
/*  Header file for basic triangle-based math manipulations  */
#ifndef Tri_Math_head
#define Tri_Math_head

#include "basics.h"      /*  Node, FOR, etc  */

/*  Macros  */

/*  Perform cyclic permutation counter-clockwise  */
#define cyclic_plus1(i) ((-4+11*(i)-3*(i)*(i))/2)  /*  123->231  */
#define cyclic_plus0(i) ((2+5*(i)-3*(i)*(i))/2)    /*  012->120  */

/*  Perform cyclic permutation clockwise  */
#define cyclic_minus1(i) ((16-13*(i)+3*(i)*(i))/2) /*  123->312  */
#define cyclic_minus0(i) ((4-7*(i)+3*(i)*(i))/2)   /*  012->201  */

/*  Linear average of  three elements  */
#define Triave(x,y,z) (((x)+(y)+(z))/3)
/*  Linear average of  three elements in an array */
#define TriaveH(p) ((*(p) + *((p)+1) + *((p)+2))/3)

/*  Data classes  */

/*  Functions  */
extern void TriaveV( int n, double x1[], double x2[], double x3[],
                     double avg[] );
extern double tri_cell_area( Node *n[] );
extern void median_dual_all( void );
extern double median_dual_one( const Node *n );
extern void tri_grad( int nq, double u[3][NEQS], double xy[3][2],
      double grad_u[2][NEQS] );
extern void containment_dual( Cell *cell );
extern double cross_edges( Edge *edge1, Edge *edge2 );
extern double norm_cross_edges( Edge *edge1, Edge *edge2 );

#endif  /* defining Tri_Math_head  */
/*

 */
```


23 Auxiliary Files

23.1 Makefile

```
#

#  Makefile for converting literate programming into compiled code
#    and/or document

#  Shell type used throughout makefile:
SHELL=/bin/sh

#  Name of the executable code
SRC1=hackboy

#  Subroutines
SUB1=gensource Stonehenge thermo transformations WestWorld roe FiniteVolume \
     limiter GovEqs Bronowski FluctuationSplitting Elmer Habashi Character \
     TriMath blas

#  Local include files
INC1=basics rhs TriMath therm transform limiter bc lhs blas adaption

#  Compiler for source code, compiler flags, linking flags
COMP=cc
#CFLAG1=-n32 -mips4 -O        #  should be using -64 instead of -n32
CFLAG1=-O -ansi -pedantic -Wall # -g
#CFLAG2=-DEBUG:div_check=3:subscript_check=ON:verbose_runtime=ON:fullwarn=ON\
# -DEBUG:woff=15,40,1196,1498
#LFLAG=-lm -lblas
LFLAG=-lm

#  How to handle implicit makes
.SUFFIXES: .lit .tex $(SUFFIXES)

#                     Make procedures

#  Default is to compile the whole code
default: code

#  Compile the code and create the document
both: code latex

#  Code option
code: $(SRC1)

#  Create entire document
latex: $(SRC1).dvi

#  Print the document
```

```
hardcopy: latex
  xdvi $(SRC1)     #  Kill at this point if not right
  cp p2active.tex twoprt.tex
  latex $(SRC1)
  cp p2inact.tex twoprt.tex
  dvips -r -B $(SRC1)
  lp $(SRC1:=.ps)
  dvips -A $(SRC1)
  @echo ' '
  @echo 'Once printing finished, re-insert paper and do:'
  @echo 'lp $(SRC1:=.ps)'
#   cat $(SRC1).ps | rsh ab00 pslpr -m -Plzr2 -SDuplex=DuplexTumble
#   lpq -Plzr2

#  Tars for porting
tar:
tar -cvf hackboy.tar $(SUB1:=.lit) $(INC1:=.h) Makefile figs \
$(SRC1:=.lit) litcon.pl
tarz:
tar -cvzf hackboy.tar.gz $(SUB1:=.lit) $(INC1:=.h) Makefile figs \
$(SRC1:=.lit) litcon.pl

#  Make individual documents for the subroutines
sr: $(SR).dvi

#  Clean-up actions
cleano:
-@rm $(SRC1).o $(SUB1:=.o)
cleanc:
-@for FILE in $(SRC1); do ls | grep -c $$FILE.lit && rm $$FILE.c; done
-@for FILE in $(SUB1); do ls | grep -c $$FILE.lit && rm $$FILE.c; done
cleancode: cleano cleanc
   -@rm $(SRC1)
cleantex:
-@rm *.log *.aux *.dvi *.blg *.bbl *.bak *.fff *.lot *.lof *.ttt \
*.toc *.loa *.lom *.lox *.tex~ *.glo *.gls *.ilg
-@rm -i *.ps
-@for FILE in $(SRC1); do ls | grep -c $$FILE.lit && rm $$FILE.tex; done
-@for FILE in $(SUB1); do ls | grep -c $$FILE.lit && rm $$FILE.tex; done

#  Making individual components

#  How to compile the primary executable
$(SRC1) : $(SRC1:=.o) $(SUB1:=.o) $(INC1:=.h)
@echo Compiling $(SRC1)
$(COMP) -o $(SRC1) $(SRC1:=.o) $(SUB1:=.o) \
$(CFLAG1) $(CFLAG2) $(LFLAG)


#  How to assemble the entire document
```



```
$(SRC1).dvi : $(SRC1:=.tex) $(SUB1:=.tex) $(INC1:=.h)
    latex $(SRC1)
    -@grep -c 'Citation .* undefined.' $(SRC1).log && bibtex $(SRC1)
    -@grep -c 'Rerun to get cross' $(SRC1).log && latex $(SRC1)
    -@grep -c 'may have changed' $(SRC1).log && latex $(SRC1)

#  Individual subroutine documents
$(SR).dvi :
  litcon.pl -latex -sa $(SR).lit
  latex $(SR)
  -@grep -c 'Citation .* undefined.' $(SR).log && bibtex $(SR)
  -@grep -c 'Rerun to get cross' $(SR).log && latex $(SR)
  -@grep -c 'may have changed' $(SR).log && latex $(SR)
  dvips $(SR)
  xdvi $(SR)
  @rm $(SR).{tex,dvi,aux,log,ps}

#  How to get the objects and source code from literate file
.lit.o: $(INC1:=.h)
litcon.pl -c $<
$(COMP) -c $(CFLAG1) $(CFLAG2) $(<:.lit=.c)
.lit.c: $(INC1:=.h)
litcon.pl -c $<
.c.o: $(INC1:=.h)
$(COMP) -c $(CFLAG1) $(CFLAG2) $<

#  How to get the tex files from the literate files
.lit.tex:
litcon.pl -latex $<

#
```

References

[Bar94] Timothy J. Barth. Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations. In *Computational Fluid Dynamics*, number 1994–04 in Lecture Series. von Karman Institute for Fluid Dynamics, 1994.

[HH83] A. Harten and J. M. Hyman. Self adjusting grid methods for one-dimensional hyperbolic conservation laws. *J. Comput. Phys.*, 50:235–269, 1983.

[HMP99] Andreas C. Haselbacher, James J. McGuirk, and Gary J. Page. Finite-volume discretization aspects for viscous flows on mixed unstructured grids. *AIAA J.*, 37(2):177–184, February 1999.

[Sid94] David Sidilkover. A genuinely multidimensional upwind scheme and efficient multigrid solver for the compressible Euler equations. Report 94–84, ICASE, USA, November 1994.

[Swe84] P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM Journal of Numerical Analysis*, 21:995–1011, 1984.

[Tec96] *Tecplot User's Manual: Version 7*. Bellevue, Washington, August 1996.

[vAvLR81] G. D. van Albada, B. van Leer, and W. W. Roberts. A comparative study of computational methods in cosmic gas dynamics. Report 81–24, ICASE, Hampton, August 1981.

[vL74] Bram van Leer. Towards the ultimate conservative scheme. II. Monotonicity and conservation combined in a second order scheme. *J. Comput. Phys.*, 14:361–370, 1974.

Appendix F

Non-linear Conservation Law Solver

The source code, written in FORTRAN, is presented for the fluctuation splitting and DMFDSFV algorithms to solve scalar conservation law problems. Non-linear advection and linear advection-diffusion can both be treated in two dimensions. The grid pre-processor code follows the solver code.

Solver for Nonlinear Conservation-Law Problems $U_t + \nabla \cdot \vec{F}(U) = \nabla \cdot (\nu \nabla U)$

William A. Wood*
NASA Langley Research Center

October 11, 2000

Abstract

A computational solver for nonlinear conservation-law problems, $U_t + \nabla \cdot \vec{F}(U) = \nabla \cdot (\nu \nabla U)$, is presented. Scalar problems in two dimensions are considered. The domain is discretized using unstructured triangular elements. The solution is advanced in time to reach a steady state, either explicitly with local timesteps or with a point-implicit relaxation. Two discretization strategies are employed: the first uses a continuous data representation with a multi-dimensional flux evaluation *via* fluctuation splitting; the second is a dimensionally-split Roe scheme, using the approximate Riemann problem to define the fluxes. Both schemes are node based and $O(\Delta x^2)$ in the steady-state. Viscous terms are discretized in a finite element framework. Parallel algorithm development is adhered to so that quantitative comparisons can be made between the performance of the fluctuation-splitting and dimensionally-split-Roe schemes.

The current code is PECLET, for solving the non-dimensional form of the equations, for constant ν,

$$U_t + \nabla \cdot \vec{F}(U) = \frac{1}{Pe} \nabla^2 U$$

*Aerospace Technologist, Aerothermodynamics Branch, Aero- and Gas-Dynamics Division. `w.a.wood@larc.nasa.gov` (757) 864-8355.

Contents

Parms.inc

Parameter statements for PECLET. Resides in solver directory. Generated by DOMAIN, in the domain directory.

nnmax	Maximum number of nodes
ncmax	Maximum number of cells
nemax	Maximum number of edges
ndim	Number of physical grid-dimensions: 2=(x,y), 3=(x,y,z)
ndim1	Number of dimensions plus one
nbn	Number of boundary nodes
nin	Number of interior nodes
iconc	Maximum number of cells connected at a node
icone	Maximum number of edges connected at a node

```
parameter ( nnmax = 20301, ncmax = 40000, nemax = 60300)
parameter ( ndim =2, ndim1 =3, nbn =   600, nin = 19800)
parameter ( iconc =14, icone =14)
```

Indeck

The controlling input deck to PECLET is indeck. This file resides in the main directory.

```
indeck - controlling input deck to Peclet
Smith and Hutton advection/diffusion problem
 $contr
  igov     = 1,  irelax   = 0, ialterGS = 1,
  ilimiter = 1,  compress = 2.0, igrad = 0,
  iskpout  = 201, itermax  =1600,  itime = 1, ivistype = 0, iminvis = 1,
  residmin = 1.e-6,  sigma =  .9
 $
compress  1-2 - limit of artificial compression for generalized Superbee
ialterGS  0,1 - alternate Gauss-Seidel update sweep
igov       0  - fluctuation-splitting
           1  - edge-based Roe scheme
igrad      0  - Barth gradient reconstruction formulation
           1  - Kleb/Wood gradient reconstruction formulation
ilimiter   0  - first order
           1  - minmod
           2  - generalized superbee
           3  - van Leer harmonic
           4  - van Albada
           5  - unlimited second order
           6  - hardwired Barth
iminvis   0,1 - only use the greater of artificial or true viscosity
irelax    -1  - Jacobi time relaxation
           0  - Gauss-Siedel time relaxation
           1  - Runge-Kutta time integration (RK-1)
iskpout    iteration skip between solution writes
itermax    maximum number of relaxation iterations
itime      1  - use first-order coefficients for timestep positivity
           2  - use second-order coefficients for timestep positivity
ivistype   0  - Finite element viscous formulation
           1  - Finite volume viscous formulation
residmin   residual for convergence check
sigma      CFL-like relaxation number
```


Dims.inc

Dimension and common block statements for PECLET.

Variables

area	Cell areas, S_T, (cell)
areamd	Median-dual areas about each node, S_{MD}, (node)
cfluc	Cell fluctuation, (cell)
du	Update to dependent vector, $\Delta^n u$, (node)
dxyz	Δx, Δy, Δz, and length for each edge, (edge, distance)
edgnorm	Scaled normal vector toward node2 from median dual connection emanating from this edge, (edge, [R,L], Δx_i)
gradu	∇u, (node, xyz)
psi	Reconstruction limiter for edge-based scheme, ψ(node)
r	Vector $\vec{r}_E - \vec{r}_0$ pointing from the node to the quadrature point on the median-dual boundary, (edge, node[1,2], [R,L], xyz)
u	Dependent variable, (node)
visa	Artificial viscosity, (node)
vist	True viscosity, (node)
xyz	Nodal coordinates, (node, x_{1-3})
ic2e	Cell-to-edge pointer, (cell, edge$_{1-4}$, [1=forward, -1=backward])
ic2n	Cell-to-node pointer, ordered CCW, (cell, node$_{1-4}$)
ie2c	Edge-to-cell pointer, (edge, cell to [R,L]) Boundary edge if `ie2c(ie, 1) = -1`
ie2n	Edge-to-node pointer, (edge, node$_{1-2}$)
in2b	Node-to-boundary pointer, identifies boundary nodes and bctype Outflow boundaries need no special treatment (nbn,1) = node, (nbn,2) = bctype[10–19=inflow, 20–29=outflow]
in2c	Node-to-cell pointer, (node, 1–# connecting cells, [cell, node$_{1-4}$])
in2e	Node-to-edge pointer, (node, 1–# connected edges, [edge, (1=first node, -1=second node)])
intnod	Interior node numbering for update sweep, (node $\neq$ boundary)
nc	Number of cells
nodcon	Number of cells and edges emanating from a node, (node, [# cells, # edges])
ne	Number of edges
nn	Number of nodes

```
      include 'parms.inc'

      dimension u(nnmax), xyz(nnmax,ndim), du(nnmax), areamd(nnmax),
     $     gradu(nnmax,ndim), intnod(nin), nodcon(nnmax,2), psi(nnmax),
     $     visa(nnmax), vist(nnmax)
      dimension in2b(nbn,2), in2e(nnmax,icone,2), in2c(nnmax,iconc,2)
      dimension dxyz(nemax,ndim1), edgnorm(nemax,2,ndim),
     $     r(nemax,2,2,ndim)
```

```
 dimension ie2n(nemax,2), ie2c(nemax,2)
 dimension area(ncmax), cfluc(ncmax)
 dimension ic2e(ncmax,4,2), ic2n(ncmax,4)

 common / basics / u, du, dxyz, xyz,
$     intnod, in2b, nodcon, nc, ne, nn
 common / Fsplit / area, cfluc, in2c, ic2n, ic2e
 common / Fvolum / gradu, psi, r, edgnorm, areamd,
$     ie2n, in2e, ie2c
 common / dgnostic / visa, vist
```


Main

Driver routine for PECLET, the 2-D nonlinear advection/diffusion solver for unstructured (triangular and/or quadrilateral) domains.

$$u_t + \nabla \cdot \mathcal{F} = u_t + F_x + G_y = \nabla \cdot (\nu \nabla U)$$

Calls inputs, domain, initial, getgradu, osoln, and spacenode.

To compile, execute `make -f mkpeclet` in solver directory.

To run, execute `rpeclet` script in main dirctory.

To print, execute `make hardcopy` in solver/texfls directory.

```
      program Peclet
      include 'dims.inc'
      common / inpts1 / ilimiter, irelax, igov, itermax, iskpout,
     $     itime, ivistype, ialterGS, iminvis, igrad
      common / inpts2 / compress, residmin, sigma
      real*4 time, etime, dum1, dum2
      integer diagnos
      diagnos = 0          !  0-4
```

Preliminaries

Read the input deck from indeck, the grid and geometric data from domain.b, and the dependent variable from flow.b.

```
      call inputs
      call domain
      call initial ( u, du, nn )
      open (15, file='pec.his', form='unformatted')    ! convergence history
```

Time Evolution

Main loop to relax the dependent variables to their steady-state solution.

```
      do iter = 1, itermax
        if (diagnos .ge. 1) write (6,*) 'Iteration=', iter
        anorm1 = 0.      !  L-1 norm of residual
        anorm2 = 0.      !  L-2 norm of residual
        anormi = 0.      !  L-inf norm of residual
        vanorm = 0.      !  L-2 norm of artificial dissipation
        vtnorm = 0.      !  L-2 norm of physical dissipation
```

If this is a finite volume computation, need to compute ∇u for second-order accuracy. Grad u is lagged at the previous time level in the case of a Gauß-Seidel update.

Grad u could be skipped periodically as the solution nears convergence.

```
        if (igov .eq. 1)
     $       call getgradu (ilimiter, igrad, residmin, compress, iter)
```

Advance one time-step by looping over all nodes and computing the update. If Gauß-Seidel then perform that update on each node. If Jacobi or forward Euler then wait until the end of the timestep to update the nodes. Jacobi and forward Euler done in this manner is twice (FV) or three times (FS) slower than it need be, by either doing the global loop over edges or cells.

At each node compute the semi-discrete RHS,

$$S_i u_{i_t} = RHS = \sum_{j \neq i} c_j (u_j - u_i)$$

`ialterGS` allows for alternating the sweep direction.

```
      if ( ialterGS .eq. 1 .and. mod(iter,2) .eq. 0 ) then
        nodestart = nin
        nodeend = 1
        nodeskip = -1
      else
        nodestart = 1
        nodeend = nin
        nodeskip = 1
      end if
```

Loop on interior nodes

```
      do inodegs = nodestart, nodeend, nodeskip
        nodegs = intnod( inodegs ) !  Current node
        du( nodegs ) = 0.
        if (diagnos .gt. 3) then
          write (6,*) 'Calling SpaceNode from main: anorm1,',
     $          ' anorm2, nodegs, ilimiter, residmin, sigma'
          write (6,*) anorm1, anorm2, nodegs, ilimiter,
     $          residmin, sigma
        end if
        call spacenode ( nodegs, ilimiter, residmin, igov, csum,
     $        compress, itime, ivistype, iminvis )
        anorm1 = anorm1 + abs( du(nodegs) )
        anorm2 = anorm2 + du(nodegs)**2
        anormi = max( anormi, abs( du(nodegs) ) )
        vanorm = vanorm + visa(nodegs)**2
        vtnorm = vtnorm + vist(nodegs)**2
```

If Gauß-Seidel then perform the update now using the local timestep,

$$\Delta t = \frac{S_{MD}}{\sum_{j \neq i} c_j}$$

along with the under-relaxation parameter σ. Jacobi uses the same timestep but performs the update at the end of the global loop over nodes. Forward Euler uses yet another timestep calculation and σ then serves the role of a CFL number. The update is constructed as,

$$\Delta^n u_i = \sigma \frac{\Delta t}{S_i} RHS$$


```
        if ( irelax .lt. 1 ) then     !  GS or Jacobi timestep
          du(nodegs) = du(nodegs) / ( csum + residmin )
          if ( irelax .eq. 0 )        !  The GS update
     $          u(nodegs) = u(nodegs) + sigma * du(nodegs)
        else                          !  Forward Euler timestep
c         call timestep
        end if
        if (diagnos .ge. 3) write (6,201) nodegs, u(nodegs),
     $        anorm2, anorm1, du(nodegs), csum
 201    format ( 'Updated node',i5,' U=', e10.3, ' L2=', e8.1,
     $        ' L1=', e8.1, ' dU=', e10.3, ' csum=', e9.2 )
      end do                          !  Interior nodes
```

Now perform loop over all outflow boundary nodes.

```
        do inodegs = 1, nbn      !   Loop on boundary nodes
          nodegs = in2b( inodegs, 1 )
          if ( in2b( inodegs, 2 ) .eq. 20 ) then
            if (diagnos .gt. 3) then
              write (6,*) 'Calling SpN from main: anorm1, anorm2,',
     $              ' nodegs, ilimiter, residmin, sigma'
              write (6,*) anorm1, anorm2, nodegs, ilimiter,
     $              residmin, sigma
            end if
            call spacenode ( nodegs, ilimiter, residmin, igov, csum,
     $            compress, itime, ivistype, iminvis )
            anorm1 = anorm1 + abs( du(nodegs) )
            anorm2 = anorm2 + du(nodegs)**2
            anormi = max( anormi, abs( du(nodegs) ) )
            vanorm = vanorm + visa(nodegs)**2
            vtnorm = vtnorm + vist(nodegs)**2
            du(nodegs) = du(nodegs) / ( csum + residmin )    ! the Timestep
            if (irelax .eq. 0)                               ! GS update
     $            u(nodegs) = u(nodegs) + sigma * du(nodegs)
            if (diagnos .ge. 3) write (6,201) nodegs, u(nodegs),
     $            anorm2, anorm1, du(nodegs), csum
          else                                  !  Inflow boundary
            du(nodegs) = 0.
            visa(nodegs) = 0.
            vist(nodegs) = 0.
          end if                                !  Outflow boundary?
        end do                                  !  Boundary nodes
        anorm2 = sqrt( anorm2 )
        vanorm = sqrt( vanorm )
        vtnorm = sqrt( vtnorm )
```

Jacobi update.

```
        if (irelax .eq. -1) then
          do in = 1, nn
            u(in) = u(in) + sigma * du(in)
```

```
            if ( diagnos .ge. 2 ) write (6,*) 'Jacobi update, ',
     $            ' node, u, du:', in, u(in), du(in)
          end do
        end if
```

Output convergence history.

```
        time = etime( dum1, dum2 )              !  timer for SGI
        write (15) iter, time, anorm1, anorm2, anormi,
     $        vanorm, vtnorm
```

Are we converged? If so, jump out of loop.

```
      if ( anorm2 .lt. residmin ) then
        write (6,*) ' Convergence achieved'
        go to 100
      end if
```

Do we write the solution now?

```
      if ( mod(iter,iskpout) .eq. 0 ) then
        call osoln ( 2, u, nn )
        write (6,153) iter, anorm2, anorm1, time
 153    format (' iter = ',i6,' L-2 = ',e10.3,' L-1 = ',e10.3,
     $        ' time = ', f8.1 )
        write (14) ( visa(in), vist(in), psi(in), in=1,nn )
      end if
    end do                                  !  Iterations
```

Wrapup

```
    write (6,*) ' Maximum iterations reached without convergence'
    iter = iter - 1
 100 continue

    write (6,*) ' iter = ', iter
    write (6,*) ' anorm2 = ', anorm2
```

Output the ending solution.

```
    call osoln ( 0, u, nn )
    write (14) ( visa(in), vist(in), psi(in), in=1,nn )
    open (144, file = 'fluc.dat', form = 'formatted' )
    write (144,*) 'cell      fluctuation'
    do ic = 1, nc
      write (144,*) ic, cfluc(ic)
    end do

    stop
    end
```


Domain

Read the grid and geometric data from `domain.b`. Called from `main`.

```
    subroutine domain
    include 'dims.inc'
    integer diagnos
    diagnos = 0

    open ( 21, file = 'domain.b', form = 'unformatted' )
    read (21) nn, nc, ne, i4, i5, i6, i7, i8
    if ( nn .gt. nnmax .or. nc .gt. ncmax .or. ne .gt. nemax .or.
   $     i4 .ne. ndim .or. i5 .ne. nbn .or. i6 .ne. nin
   $     .or. i7 .ne. iconc .or. i8 .ne. icone ) then
      write (6,*) 'problem in domain'
      write (6,*) nn, nc, ne, i4, i5, i6, i7, i8
      write (6,*) nnmax, ncmax, nemax, ndim, nbn, nin, iconc, icone
      stop
    end if

    read (21) (intnod(i1), i1=1,nin)
    read (21) ((in2b(i1,i2), i2=1,2), i1=1,nbn)
    read (21) ((nodcon(i1,i2), i2=1,2), i1=1,nn)
    read (21) (((in2c(i1,i2,i3), i3=1,2), i2=1,nodcon(i1,1)),
   $     i1=1,nn)
    read (21) (((in2e(i1,i2,i3), i3=1,2), i2=1,nodcon(i1,2)),
   $     i1=1,nn)
    read (21) ((ic2n(i1,i2), i2=1,4), i1=1,nc)
    read (21) (((ic2e(i1,i2,i3), i3=1,2), i2=1,4), i1=1,nc)
    read (21) ((ie2n(i1,i2), i2=1,2), i1=1,ne)
    read (21) ((ie2c(i1,i2), i2=1,2), i1=1,ne)
    read (21) ((xyz(i1,i2), i2=1,ndim), i1=1,nn)
    read (21) (areamd(i1), i1=1,nn)
    read (21) (area(i1), i1=1,nc)
    read (21) ((dxyz(i1,i2), i2=1,ndim1), i1=1,ne)
    read (21) (((edgnorm(i1,i2,i3), i3=1,ndim), i2=1,2), i1=1,ne)
    read (21) ((((r(i1,i2,i3,i4), i4=1,ndim), i3=1,2), i2=1,2),
   $     i1=1,ne)
    close (21)

    return
    end
```

Fluctuation Splitter

Compute the fluctution and distribution for a triangular cell.
Called from `spacenode`. Calls `limiter`.

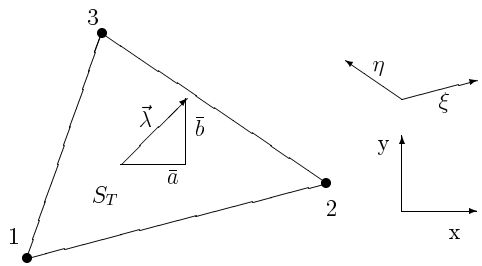


Figure 1: Fluctuation splitting nomenclature for a triangular element.

```
 subroutine flsp( fluc1, fluc2, fluc3,
$     fluc12, fluc21, fluc23, fluc32, alph, beta,
$     rxibar, retbar, abar, bbar,
$     u1, u2, u3, dx1, dy1, dx2, dy2, ilimiter, eps, compress )
 integer diagnos
 diagnos = 0          !  0-3
 if ( diagnos .ne. 0 ) write (6,*) 'In flsp: u1,2,3:',u1,u2,u3
```

$$\alpha = \frac{\bar{a}}{2}(y_3 - y_2) - \frac{\bar{b}}{2}(x_3 - x_2), \quad \beta = -\frac{\bar{a}}{2}(y_2 - y_1) + \frac{\bar{b}}{2}(x_2 - x_1)$$

```
 if (diagnos .eq. 3) write (6,*) 'abar=', abar, ' bbar=', bbar,
$     ' dx1,2=', dx1, dx2, ' dy1,2=', dy1, dy2
 alph = 0.5 * ( abar * dy2 - bbar * dx2 )
 beta = 0.5 * (-abar * dy1 + bbar * dx1 )
 if (diagnos .ge. 2)
$     write (6,*) 'In flsp, alpha, beta', alph, beta
```

$$R^{\xi} = -\alpha(u_2 - u_1), \quad R^{\eta} = -\beta(u_3 - u_2)$$

```
 rxi = alph * (u1 - u2)
 ret = beta * (u2 - u3)
 if (diagnos .ge. 2) write (6,*) 'Rx, Ry', rxi, ret
```

**** note: see Sidilkover about how the limiter makes fluctuation splitting *LP*. How can I change this to be an unlimited scheme? What about *LP* but not *P*?

The higer-order scheme is obtained by limiting the fluctuations.

$$R^{\xi^*} = R^{\xi} + M(-R^{\xi}, R^{\eta}), \quad R^{\eta^*} = R^{\eta} - M(-R^{\xi}, R^{\eta})$$

M is the averaging function for the limiter ψ such that,

$$\psi(Q/P) = \frac{M(Q,P)}{P} = \frac{M(P,Q)}{P}$$

Set small parameter for van Albada limiter,

$$\varepsilon^2 = \left(\frac{\ell_{12} + \ell_{23}}{2}\right)^3$$

```
vAlb = 0.5 * (sqrt(dx1**2 + dy1**2) + sqrt(dx2**2 + dy2**2))
vAlb = vAlb**3
call limiter( ilimiter, -rxi, ret, avM, compress, vAlb )
rxi = rxi + avM
ret = ret - avM
if (diagnos .ge. 2) write (6,*) 'Rxstar, Rystar', rxi, ret
```

Compute artificial viscosity terms,

$$\bar{R}^{\xi} = \mathtt{sign}(\alpha) R^{\xi^*}, \quad \bar{R}^{\eta} = \mathtt{sign}(\beta) R^{\eta^*}$$

```
rxibar = rxi * sign( 1., alph )
retbar = ret * sign( 1., beta )
if (diagnos .ge. 2) write (6,*) 'Rxbar, Rybar', rxibar, retbar
```

Distribute the fluctution to the nodes.

$$\mathtt{fluc1} = \frac{S_T}{3}\frac{\Delta^n u_1}{\tau_T} = \frac{R^{\xi^*} - \bar{R}^{\xi}}{2}$$

$$\mathtt{fluc2} = \frac{S_T}{3}\frac{\Delta^n u_2}{\tau_T} = \frac{R^{\xi^*} + \bar{R}^{\xi}}{2} + \frac{R^{\eta^*} - \bar{R}^{\eta}}{2}$$

$$\mathtt{fluc3} = \frac{S_T}{3}\frac{\Delta^n u_3}{\tau_T} = \frac{R^{\eta^*} + \bar{R}^{\eta}}{2}$$

```
fluc1 = 0.5 * ( rxi - rxibar )
fluc2 = 0.5 * ( rxi + rxibar + ret - retbar )
fluc3 = 0.5 * ( ret + retbar )
if ( diagnos .ge. 1 ) write (6,*) 'Fluc1-3: ',fluc1,fluc2,fluc3
```

Further manipulations can be made by defining,

$$\mathtt{fluc12} = \frac{R^{\xi^*} - \bar{R}^{\xi}}{2}\frac{1}{u_2 - u_1} = \frac{1 - \mathtt{sign}(\alpha)}{2}\left(-\alpha + \frac{M}{u_2 - u_1}\right)$$

$$\mathtt{fluc21} = \frac{R^{\xi^*} + \bar{R}^{\xi}}{2}\frac{1}{u_1 - u_2} = \frac{1 + \mathtt{sign}(\alpha)}{2}\left(\alpha + \frac{M}{u_1 - u_2}\right)$$

$$\texttt{fluc23} = \frac{R^{\eta^*} - \bar{R}^{\eta}}{2}\frac{1}{u_3 - u_2} = \frac{1 - \texttt{sign}(\beta)}{2}\left(-\beta - \frac{M}{u_3 - u_2}\right)$$

$$\texttt{fluc32} = \frac{R^{\eta^*} + \bar{R}^{\eta}}{2}\frac{1}{u_2 - u_3} = \frac{1 + \texttt{sign}(\beta)}{2}\left(\beta - \frac{M}{u_2 - u_3}\right)$$

Leading to the alternate expression, useful for Gauß-Seidel updates,

$$\texttt{fluc1} = \texttt{fluc12}(u_2 - u_1)$$

$$\texttt{fluc2} = \texttt{fluc21}(u_1 - u_2) + \texttt{fluc23}(u_3 - u_2)$$

$$\texttt{fluc3} = \texttt{fluc32}(u_2 - u_3)$$

```
 fluc12 = ( 0.5 - sign(0.5, alph) ) * (-alph +
$     avM / ( u2 - u1 + sign(eps, u2 - u1) ) )
 fluc21 = ( 0.5 + sign(0.5, alph) ) * ( alph +
$     avM / ( u1 - u2 + sign(eps, u1 - u2) ) )
 fluc23 = ( 0.5 - sign(0.5, beta) ) * (-beta -
$     avM / ( u3 - u2 + sign(eps, u3 - u2) ) )
 fluc32 = ( 0.5 + sign(0.5, beta) ) * ( beta -
$     avM / ( u2 - u3 + sign(eps, u2 - u3) ) )

 return
 end
```


Getgradu

Compute ∇u across a median dual, based at the node. This is used for higher-order reconstruction of u with Roe schemes. u can be reconstructed anywhere in the median dual by,

$$u = u_0 + \vec{r} \cdot \nabla u$$

Alternatively, a limited reconstruction can be used such that,

$$u^{recon} = u_0 + \psi\, \vec{r} \cdot \nabla u$$

where the limiter ψ is a function,

$$\psi = \psi\left(\frac{\frac{u^{min/max} - u_0}{2}}{(\vec{r} \cdot \nabla u)^{min/max}}\right) = \frac{1}{(\vec{r} \cdot \nabla u)^{min/max}} M\left(\frac{u^{min/max} - u_0}{2}, (\vec{r} \cdot \nabla u)^{min/max}\right)$$

and M is an appropriate averaging function.

Called by main. Calls limiter and leastsquare.

```
subroutine getgradu( ilimiter, igrad, eps, compress, niter )
include 'dims.inc'
real l11(nnmax), l22(nnmax), l12(nnmax), lf1(nnmax), lf2(nnmax)
dimension amat(icone+1,5), bvec(icone+1), xvec(5)  ! match leastsquare
integer diagnos
diagnos = 0
if (diagnos .gt. 0) write (6,*) 'In Getgradu, diagnos=',diagnos
iLS = 0
```

First order

$\nabla u = 0$

```
if ( ilimiter .eq. 0 ) then
  do in = 1, nn
    do ixyz = 1, ndim
      gradu( in, ixyz ) = 0.
    end do
    psi(in) = 1.
  end do
else
```

Second order

Reconstruct with an unweighted least-squares from Barth[1]. For two dimensions only. First zero the summations.

∇u

```
do in = 1, nn                !   zero initializations
  l11(in) = 0.
  l12(in) = 0.
  l22(in) = 0.
  lf1(in) = 0.
  lf2(in) = 0.
end do
do in1 = 1, icone+1
  bvec(in1) = 0.
  do in2 = 1, 5
    amat(in1,in2) = 0.
    xvec(in2) = 0.
  end do
end do
```

Perform least squares operation to get ∇u. This is the Barth algorithm.

```
if ( iLS .eq. 0 ) then
do k = 1, ne                            !  Loop on edges
  j1 = ie2n( k, 1 )
  j2 = ie2n( k, 2 )
  if (diagnos .ge. 3) then
    write (6,*) 'Edge, n1, n2', k, j1, j2
    write (6,*) 'dx, dy', dxyz(k,1), dxyz(k,2)
    write (6,*) 'l11, l12, l22, lf1, lf2'
    write (6,*) l11(j1), l12(j1), l22(j1), lf1(j1), lf2(j1)
    write (6,*) l11(j2), l12(j2), l22(j2), lf1(j2), lf2(j2)
  end if
  l11( j1 ) = l11( j1 ) + dxyz( k, 1 )**2
  l11( j2 ) = l11( j2 ) + dxyz( k, 1 )**2
  l22( j1 ) = l22( j1 ) + dxyz( k, 2 )**2
  l22( j2 ) = l22( j2 ) + dxyz( k, 2 )**2
  l12( j1 ) = l12( j1 ) + dxyz( k, 1 ) * dxyz( k, 2 )
  l12( j2 ) = l12( j2 ) + dxyz( k, 1 ) * dxyz( k, 2 )
  delu = u( j2 ) - u( j1 )
  lf1( j1 ) = lf1( j1 ) + dxyz( k, 1 ) * delu
  lf1( j2 ) = lf1( j2 ) + dxyz( k, 1 ) * delu
  lf2( j1 ) = lf2( j1 ) + dxyz( k, 2 ) * delu
  lf2( j2 ) = lf2( j2 ) + dxyz( k, 2 ) * delu
  if (diagnos .ge. 3) then
    write (6,*) 'l11, l12, l22, lf1, lf2'
    write (6,*) l11(j1), l12(j1), l22(j1), lf1(j1), lf2(j1)
    write (6,*) l11(j2), l12(j2), l22(j2), lf1(j2), lf2(j2)
  end if
end do
do j = 1, nn                            !  Loop on nodes
  det = l11( j ) * l22( j ) - l12( j ) * l12( j )
  gradu(j,1) = (l22(j) * lf1(j) - l12(j) * lf2(j)) / det
  gradu(j,2) = (l11(j) * lf2(j) - l12(j) * lf1(j)) / det
```



```
  if ( ndim .eq. 3 ) gradu(j,3) = 0.
  if (diagnos .ge. 1) write (6,*) 'Node, gradu:', j,
$       gradu(j,1), gradu(j,2), gradu(j,3)
  if (diagnos .ge. 2) then
    write (6,*) 'l11, l12, l22, lf1, lf2, det'
    write (6,*) l11(j), l12(j), l22(j), lf1(j), lf2(j), det
  end if
end do
```

This is a traditional least squares method. For each edge, $u_i - u_0 = \nabla u_0 \cdot \vec{r}_{0i}$.

```
else
do in = 1, nn         !     loop on nodes
  do ie = 1, nodcon(in,2)    !  loop on edges connected to node
    Nother = ie2n( in2e(in,ie,1), (in2e(in,ie,2) + 3)/2)
    amat(ie,1) = xyz(Nother,1) - xyz(in,1)
    amat(ie,2) = xyz(Nother,2) - xyz(in,2)
    bvec(ie) = u( Nother ) - u( in )
  end do
  call leastsquare ( amat, bvec, xvec, nodcon(in,2), 2 )
  gradu(in,1) = xvec(1)
  gradu(in,2) = xvec(2)
end do
end if
```

ψ_i

Now compute the gradient limiter ψ at this node.

```
      do j = 1, nn
c       if ( j .eq. 5747 .and. niter .eq. 70 ) then
c         diagnos = 4
c       else
c         diagnos = 0
c       end if
```

Barth method
Perform two loops on all edges connecting to this node. The first loop finds u^{min} and u^{max}. The second loop finds the most restrictive limiter to use for the gradient reconstruction over the current medial dual.

```
  if ( igrad .eq. 0 ) then
  Umin = u(j)
  Umax = u(j)
  if ( diagnos .ge. 2 ) write (6,*) 'node ',j,' u ',u(j)
  do nedge = 1, nodcon(j,2)
    ie = in2e(j, nedge, 1)
    Uother = u( ie2n(ie, (in2e(j,nedge,2) + 3)/2) )
    Umin = min( Umin, Uother )
    Umax = max( Umax, Uother )
    if ( diagnos .ge. 4 ) write (6,*) 'edge=',ie,' onode=',
```

```
$           ie2n(ie, (in2e(j,nedge,2) + 3)/2),' Uo=',Uother,
$           'Umin/max=', Umin, Umax
      end do
      psi(j) = 2.
      do nedge = 1, nodcon(j,2)
        ie = in2e(j, nedge, 1)
        do irl = 1, 2
```

This section is hardwired for Barth and Jesperson limiter[2].

```
            if ( ilimiter .eq. 6 ) then
              if ( diagnos .ge. 4 ) then
                write (6,*) 'edge = ',ie,' irl=',irl
                write (6,*) 'r ',
$                   r(ie, (3-in2e(j,nedge,2))/2, irl, 1),
$                   r(ie, (3-in2e(j,nedge,2))/2, irl, 2)
                write (6,*) 'grad u ',gradu(j,1),gradu(j,2)
              end if
            q = gradu(j,1) * r(ie, (3-in2e(j,nedge,2))/2, irl, 1)
$             + gradu(j,2) * r(ie, (3-in2e(j,nedge,2))/2, irl, 2)
            if ( diagnos .ge. 4 ) write (6,*) 'q = ',q
            if ( q .gt. 0. ) then
              psi(j) = min( psi(j), min( 1., (Umax - u(j)) / q ) )
            else if ( q .lt. 0. ) then
              psi(j) = min( psi(j), min( 1., (Umin - u(j)) / q ) )
            else
              psi(j) = min( psi(j), 1. )
            end if
            if ( diagnos .ge. 4 ) write (6,*) 'psi= ', psi(j)
            if ( diagnos .ge. 4 ) write (6,*)
```

General limiter formulation.

```
            else
            p = 0.5 * ( Umax - u(j) )
            q = gradu(j,1) * r(ie, (3-in2e(j,nedge,2))/2, irl, 1)
$             + gradu(j,2) * r(ie, (3-in2e(j,nedge,2))/2, irl, 2)
```

rmag is the length of $\vec{r}$. This is passed for the van Albada limiter as $\varepsilon^2 = \texttt{rmag}^3$.

```
            rmag = sqrt(r(ie, (3-in2e(j,nedge,2))/2, irl, 1)**2
$                + r(ie, (3-in2e(j,nedge,2))/2, irl, 2)**2 )
            rmag = rmag**3
            call limiter ( ilimiter, p, q, aM1, compress, rmag )
            p = 0.5 * ( Umin - u(j) )
            call limiter ( ilimiter, p, q, aM2, compress, rmag )
            psitemp = max( abs(aM1), abs(aM2) ) /
$                ( abs( q ) + eps )
            if ( diagnos .ne. 0 ) then
            if ( psitemp .lt. 0. ) then
              write (6,*) 'Negative limiter in getgradu:', psitemp
              write (6,*) 'Node:', j,' Edge:', ie
```



```
              write (6,*) 'Umax:', Umax,' Umin:', Umin,' U:', u(j)
              write (6,*) 'Mmax:', aM1,' Mmin:', aM2
              write (6,*) 'Grad u:', gradu(j,1), gradu(j,2)
              write (6,*) 'r-vec:',
$                 r(ie, (3-in2e(j,nedge,2))/2, irl, 1),
$                 r(ie, (3-in2e(j,nedge,2))/2, irl, 2)
            end if
            end if
            psi(j) = min ( psi(j), psitemp )
            end if              !   Barth or general limiter
          end do
        end do
        if (diagnos .gt. 0) write (6,*) 'psi = ', psi(j)
        else
```

Kleb/Wood method
aggrad is the magnitude of the gradient. angmax is the maximal directional cosine. angmin is the minimal directional cosine.

```
          aggrad = sqrt( gradu(j,1)**2 + gradu(j,2)**2 )
          angmax = -1.1
          angmin =  1.1
```

Loop over surrounding edges and find the 2 edges most aligned with the gradient direction, one in the positive direction and the other in the negative direction.

$$\cos\phi = \frac{\nabla u \cdot \vec{r}_{ji}}{||\nabla u||\,||\vec{r}||}$$

Compare the slopes along these edges, projected onto the gradient. These slopes are then averaged in limiter,

$$\cos\phi\,\frac{u_i - u_j}{||\vec{r}||}$$

```
          do nedge = 1, nodcon(j,2)      !  loop over connecting edges
            ie = in2e(j, nedge, 1)       !  current edge
            idir = in2e(j, nedge, 2)     !  edge defined forward or backward
            dircos = float(idir) *       !  directional cosine
$                ( gradu(j,1) * dxyz(ie,1) + gradu(j,2) * dxyz(ie,2) )
$                / (aggrad + eps) / dxyz(ie,ndim1)
            if ( abs(dircos) .gt. 1.000001 ) then
              write (6,*) ' bad value for dircos in getgradu',
$                 dircos
              stop
            end if
            if ( dircos .lt. angmin ) then
              angmin = dircos
              slope2 = dircos / dxyz(ie,ndim1) *
$                 ( u( ie2n( ie, (idir+3)/2 ) ) - u(j) )
            end if
            if ( dircos .gt. angmax ) then
```

```
            angmax = dircos
            slope1 = dircos / dxyz(ie,ndim1) *
$                ( u( ie2n( ie, (idir+3)/2 ) ) - u(j) )
          end if
        end do              !   edges about this node

        rmag = ( sqrt( areamd(j) ) )**3
        call limiter ( ilimiter, slope1, slope2, psi(j),
$            compress, rmag )
```

The limiter is now formed as,

$$\psi_j = \frac{M}{||\nabla u||}$$

```
        psi(j) = psi(j) / ( aggrad  + eps )      !   the limiter

      end if          !   igrad
    end do            !   current node
  end if              !   ilimiter - first or second order

  return
  end
```


Initial

Read the starting values for the dependent variable from `flow.b`. Perform other initializations. Called from `main`.

```
      subroutine initial ( u, du, nn )
      include 'parms.inc'
      dimension u(nnmax), du(nnmax)
      integer diagnos
      diagnos = 0
      if ( diagnos .gt. 0 ) write (6,*)' in Initial: diagnos= ',diagnos
```

Read dependent variable from `flow.b`. Loop on the `read` statement to start with the last solution stored in `flow.b`. Leave the file open so future solutions may be appended.

```
      open ( 22, file = 'flow.b', form = 'unformatted' )
 10   read (22, END=11) idum1
      read(22) (u(i1), i1=1, idum1)
      goto 10
 11   if ( idum1 .ne. nn ) then
        write (6,*) 'error reading flow.b in initial'
        write (6,*) 'file has ',idum1,' nodes but nn=', nn
        stop
      end if
```

Zero out the update Δu to start.

```
      do in = 1, nn
        du( in ) = 0.
      end do

      return
      end
```

Inputs

Read controlling input deck, `indeck`. Called from `main`. See `indeck` for a description of the input variables and options.

```
      subroutine inputs
      common / inpts1 / ilimiter, irelax, igov, itermax, iskpout,
     $     itime, ivistype, ialterGS, iminvis, igrad
      common / inpts2 / compress, residmin, sigma

      namelist / contr / ilimiter, itermax, iskpout, igov, irelax,
     $     itime, ivistype, ialterGS, iminvis, igrad,
     $     compress, residmin, sigma

      open (7, file = 'indeck', form = 'formatted' )
      read (7,*)                 ! Free line to identify file name
      read (7,*)                 ! Free line to describe case
      read (7,contr)

      return
      end
```


Limiter

Limiter functions, $\psi(r)$, are defined, where $r = P/Q$ is the ratio of the two limiter test values. The limiters are symmetric,

$$\psi\left(\frac{1}{r}\right) = \frac{\psi(r)}{r}$$

or,

$$Q\,\psi(P/Q) = P\,\psi(Q/P)$$

In order to avoid numerical problems with division, the limiters are recast in terms of their equivalent symmetric averaging functions, $M(P,Q)$. These obey,

$$Q\,\psi(P/Q) = M(P,Q) = M(Q,P) = P\,\psi(Q/P)$$

Called by `flsp` and `getgradu`. ϕ is the artificial compression parameter.

```
      subroutine limiter ( ilimiter, p, q, avM, phi, vAlb )
      integer diagnos
      diagnos = 0
      if ( diagnos .ne. 0 )
     $     write (6,*) 'In Limiter with diagnos=', diagnos
      if ( diagnos .gt. 0 ) write(6,*) 'ilimiter =', ilimiter
```

First order, `ilimiter`=0.

$$\psi = M = 0$$

```
      if ( ilimiter .eq. 0 ) then
        avM = 0.
```

Minmod limiter, `ilimiter`=1.

$$\psi(r) = \max(0,\ \min(1,\ r))$$

$$\psi\left(\frac{P}{Q}\right) = \begin{cases} 0 & PQ \le 0 \\ P/Q \quad \text{if} & |P| \le |Q| \\ 1 & |P| \ge |Q| \end{cases}$$

$$M(P,Q) = \begin{cases} 0 & PQ \le 0 \\ P \quad \text{if} & |P| \le |Q| \\ Q & |P| \ge |Q| \end{cases}$$

```
      else if ( ilimiter .eq. 1 ) then
        avM = sign(1.,q) * max(0., sign(min(abs(p), abs(q)), p*q))
```

Generalized Superbee limiter[3], `ilimiter`=2.

$$\psi_\phi(r) = \max[0,\ \min(\phi r,\ 1),\ \min(r,\ \phi)]$$

$$\psi_\phi\left(\frac{P}{Q}\right) = \begin{cases} 0 & PQ \le 0 \\ \phi P/Q & \phi|P| \le |Q| \\ 1 \quad \text{if} & |P| \le |Q| \le \phi|P| \\ P/Q & |Q| \le |P| \le \phi|Q| \\ \phi & \phi|Q| \le |P| \end{cases}$$

$$M(P,Q) = \begin{cases} 0 & PQ \le 0 \\ \phi P & \phi|P| \le |Q| \\ Q \quad \text{if} & |P| \le |Q| \le \phi|P| \\ P & |Q| \le |P| \le \phi|Q| \\ \phi Q & \phi|Q| \le |P| \end{cases}$$

```
      else if ( ilimiter .eq. 2 ) then
        avM = sign(1.,q) * max( 0., sign( max(
     $       min(abs(q), phi*abs(p)), min(abs(p), phi*abs(q)) ),
     $       p * q ) )
```

van Leer limiter[4], `ilimiter=3`.

$$\psi\left(\frac{P}{Q}\right) = \frac{\frac{P}{Q} + |\frac{P}{Q}|}{1 + |\frac{P}{Q}|} = \frac{PQ + |PQ|}{Q^2 + |PQ|} = \frac{|P| + |Q|\frac{P}{Q}}{|P| + |Q|}$$

$$M(P,Q) = \frac{|P|Q + P|Q|}{|P| + |Q|}$$

```
      else if ( ilimiter .eq. 3 ) then
        avM = ( abs(p) * q + p * abs(q) ) / ( abs(p) + abs(q) + 1.e-6 )
```

van Albada limiter[5], `ilimiter=4`.

$$\psi\left(\frac{P}{Q}\right) = \frac{P(P+Q)}{P^2 + Q^2} \text{ (Sweby)}$$

$$M(P,Q) = \frac{(P^2+\varepsilon^2)Q + (Q^2+\varepsilon^2)P}{(P^2+\varepsilon^2) + (Q^2+\varepsilon^2)} = \frac{(PQ+\varepsilon^2)(P+Q)}{P^2+Q^2+2\varepsilon^2}$$

where $\varepsilon^2 \sim \Delta x^3$ (assumes typical $\nabla u \sim 1$).

```
      else if ( ilimiter .eq. 4 ) then
c       if ( p * q .le. 0. ) then
c         avM = 0.
c       else
c         avM = p * q * ( p + q ) / ( p*p + q*q + 1.e-7 )
c       end if
c       vAlb = 0.000008
        avM = (p * q + vAlb) * ( p + q ) / ( p*p + q*q + 2.*vAlb )
```


Unlimited, `ilimiter=5`.

This is a non-symmetric limiter, useful for the finite volume formulation. Application to fluctuation splitting still needs to be worked out. It allows a second-order accurate, non-positive scheme, which may still be stable in the absence of shocks.

$$\psi = 1, \quad M = Q$$

```
      else if ( ilimiter .eq. 5 ) then
        avM = q
      else
        write (6,*) ' bad value for ilimiter in limiter'
        write (6,*) ' ilimiter = ', ilimiter
      end if

      if ( diagnos .eq. 2 ) write (6,100) p, q, avM
 100  format( 'P = ', e10.3, ', Q = ', e10.3, ' M = ', e10.3 )

      return
      end
```

Osoln

Append the solution to the binary file `flow.b`. Postprocessing must be done by POSTPLT. Called from `main`.

```
      subroutine osoln ( ntimes, u, nn )
      include 'parms.inc'
      dimension u(nnmax)

      write (22) nn
      write (22) (u(i1), i1=1,nn)
```

Close the output file if this is the last time through.

```
      if ( ntimes .eq. 0 ) close (22)

      return
      end
```


SpaceNode

Assemble the RHS spatial discretization at a node. If fluctuation splitting then loop on triangles surronding this node. If finite volume then loop on edges emanating from this node.

$$S_{MD}\frac{\Delta^n u_0}{\Delta t} = RHS = \sum_{edges\ j} (c_{jR} + c_{jL})(u_j - u_0) = \sum_{tri.\ T} \phi_{0_T}$$

where c_{jR} and c_{jL} are the contributing coefficients from the triangles to the right and left of edge j and ϕ_0 is the fluctuation distribution from triangle T to the current node.

Called by `main`. Calls `flsp`, `roe`, `viscos` and `wavespeed`.

```
      subroutine spacenode ( nodegs, ilimiter, residmin, igov, csum,
     $     compress, itime, ivistype, iminvis )
      include 'dims.inc'
      integer diagnos, diagvis, edge1, edge2, edge3
      diagnos = 0          !  0-4  inviscid diagnostics
      diagvis = 0          !  0-1  viscous diagnostics
      csum  = 0.
      cusum = 0.
      ioptim = 0
      visa( nodegs ) = 0.
      vist( nodegs ) = 0.
      if (diagnos .ge. 1) write (6,*) 'In SpaceNode, nodegs=', nodegs
```

Advective contribution to update

Fluctuation Splitting

If fluctuation splitting, then loop over all triangles connected to this node.

```
      if ( igov .eq. 0 ) then
        do ntri = 1, nodcon( nodegs, 1 )
          itri = in2c( nodegs, ntri, 1 ) !  Current triangle
          node1 = ic2n( itri, 1 )
          node2 = ic2n( itri, 2 )
          node3 = ic2n( itri, 3 )
          edge1 = ic2e( itri, 1, 1 )
          edge2 = ic2e( itri, 2, 1 )
          edge3 = ic2e( itri, 3, 1 )
          if (diagnos .ge. 2) then
            write (6,*) 'itri=', itri
            write (6,*) 'nodes1-3', node1, node2, node3
            write (6,*) 'edges1-3', edge1, edge2, edge3
          end if
```

Define wavespeed $\vec{\lambda}$. Form cell-average advection velocity as linear average of nodal velocities.

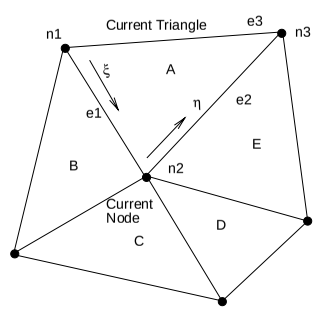


Figure 2: Nomenclature for fluctuation splitting contributions to the current node.

```
      alam1 = 0.
      alam2 = 0.
      do in = 1, 4
        node = ic2n( itri, in )
        if ( node .eq. 0 ) go to 102       !  for triangles
        call wavespeed ( xyz(node, 1), xyz(node, 2),
$            u(node), speedx, speedy, iburg )
        alam1 = alam1 + speedx
        alam2 = alam2 + speedy
      end do
102   continue
      alam1 = alam1 / float(in - 1)
      alam2 = alam2 / float(in - 1)
      if (diagnos .eq. 2) then
        write (6,*) 'cell ', itri,' lambda=', alam1,
$            alam2
      end if
```

Obtain the fluctuation distribution on this triangle.

```
      call flsp( fluc1, fluc2, fluc3,
$     fluc12, fluc21, fluc23, fluc32, alph, beta,
$     rxibar, retbar,
$     alam1, alam2, u(node1), u(node2), u(node3),
$     ic2e(itri,1,2)*dxyz(edge1,1), ic2e(itri,1,2)*dxyz(edge1,2),
```



```
$     ic2e(itri,2,2)*dxyz(edge2,1), ic2e(itri,2,2)*dxyz(edge2,2),
$     ilimiter, residmin*0.1, compress )
      if (diagnos .ge. 3) then
        write (6,*) 'u1-3:', u(node1), u(node2), u(node3)
        write (6,*) 'fluc1-3:', fluc1, fluc2, fluc3
        write (6,*) 'fluc12,21,23,32:',fluc12,fluc21,fluc23,fluc32
      end if
```

Given the fluctuations the running sums for this node can be updated. `cusum` is the nodal update. `csum` is the edge coefficient for computing the timestep. The contributions to the artificial dissipation are assembled.

```
        cfluc(itri) = fluc1 + fluc2 + fluc3
    if ( nodegs .eq. node1 ) then
      if ( itime .eq. 1 ) then             ! use first-order coef.
        csum = csum + max( 0., -alph )
      else
        csum = csum + fluc12
      end if
      cusum = cusum + fluc1
      visa( nodegs ) = visa( nodegs ) - 0.5 * rxibar
    else if ( nodegs .eq. node2 ) then
      if ( itime .eq. 1 ) then             ! use first-order coef.
        csum = csum + max( 0., alph ) + max( 0., -beta )
      else
        csum = csum + fluc21 + fluc23
      end if
      cusum = cusum + fluc2
      visa( nodegs ) = visa( nodegs ) + 0.5 * ( rxibar - retbar )
    else if ( nodegs .eq. node3 ) then
      if ( itime .eq. 1 ) then             ! use first-order coef.
        csum = csum + max( 0., beta )
      else
        csum = csum + fluc32
      end if
      cusum = cusum + fluc3
      visa( nodegs ) = visa( nodegs ) + 0.5 * retbar
    else
      write (6,*) 'Error in GS: nodegs=', nodegs,
$          ' NE node1-3', node1, node2, node3
      stop
    end if

  end do                    !  Triangles surrounding the current node
  else
```

Finite Volume

In order to update the current node, loop over all edges connected to this point.

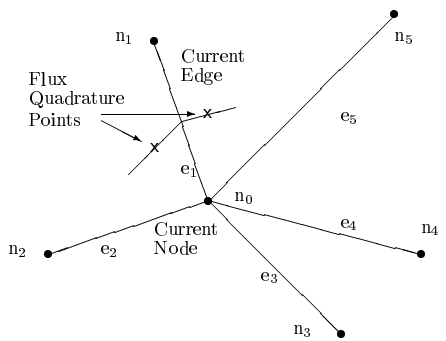


Figure 3: Nomenclature for finite volume contributions to the current node.

```
do nedg = 1, nodcon( nodegs, 2 )
  iedg = in2e( nodegs, nedg, 1 ) !  Current edge
```

For an edge-based Roe scheme, need to consider the median dual cell faces to both the left and right of the current edge.

```
node1 = nodegs
idir = in2e( node1, nedg, 2 )
node2 = ie2n( iedg, (3+idir)/2 )
```

Do the median-dual face to the right and then to the left of this edge. Begin by reconstructing the dependent variables to the median-dual face.

$$u^{recon} = u_0 + \psi\, \vec{r} \cdot \nabla u$$

```
    do irl = 1, 2
      Uout= u(node2) + psi(node2) * ( gradu(node2, 1) *
$        r(iedg, (3+idir)/2, irl, 1) +
$        gradu(node2, 2) * r(iedg, (3+idir)/2, irl, 2) )
      Uin = u(node1) + psi(node1) * ( gradu(node1, 1) *
$        r(iedg, (3-idir)/2, irl, 1) +
$        gradu(node1, 2) * r(iedg, (3-idir)/2, irl, 2) )
      if (diagnos .gt. 3) then
        write (6,*) 'OUT: node, edge, psi, gradu, r', node2, iedg
        write (6,*) psi(node2), gradu(node2, 1), gradu(node2, 2),
$           r(iedg, (3+idir)/2, irl, 1),
$           r(iedg, (3+idir)/2, irl, 2)
        write (6,*) 'IN: node, edge, psi, gradu, r', node1, iedg
        write (6,*) psi(node1), gradu(node1, 1), gradu(node1, 2),
```



```
$           r(iedg, (3-idir)/2, irl, 1),
$           r(iedg, (3-idir)/2, irl, 2)
      end if
      if (diagnos .ge. 3) then
        write (6,*) 'Calling Roe on edge ', iedg
        write (6,*) 'NodeOut, U, Uout:', node2, u(node2), Uout
        write (6,*) 'NodeIn, U, Uin:', node1, u(node1), Uin
      end if
```

Check if edge1 is a boundary edge. If so, then normals definitions change slightly.

```
      if ( ie2c(iedg,irl) .lt. 0 ) then
        call roe ( Uin, Uout,
$           edgnorm(iedg, irl, 1), edgnorm(iedg, irl, 2),
$           xyz(node1, 1) + r(iedg, (3-idir)/2, irl, 1),
$           xyz(node1, 2) + r(iedg, (3-idir)/2, irl, 2),
$           flux, alph, rxibar )
      else
        call roe ( Uin, Uout,
$           idir * edgnorm(iedg, irl, 1),
$           idir * edgnorm(iedg, irl, 2),
$           xyz(node1, 1) + r(iedg, (3-idir)/2, irl, 1),
$           xyz(node1, 2) + r(iedg, (3-idir)/2, irl, 2),
$           flux, alph, rxibar )
      end if
      if (diagnos .ge. 3) write(6,*) 'Flux, alph', flux, alph
```

Given the fluctuations the running sums for this node can be updated.

```
      if ( alph .gt. 0. ) then    ! outflowing
        if ( itime .eq. 1 ) then  ! first order coefficients
          csum = csum + alph
        else
          csum = csum + flux / (u(node1) + residmin*.1)
        end if
      end if
      cusum = cusum - flux
      visa( nodegs ) = visa( nodegs ) + rxibar

    end do                  !  to Right and Left of edge
  end do                    !  Edges surrounding the current node
  end if                    !  Fluctuation splitting or finite volume
```

Diffusive contribution to update–Viscous terms

Finite element

For finite element formulation, loop on surrounding cells. For finite volume formulation, loop on surrounding nodes.

```
  if ( ivistype .eq. 0 ) then    !  finite element form
    do ntri = 1, nodcon( nodegs, 1 )
```

```
      itri = in2c( nodegs, ntri, 1 ) !  Current triangle
      node1 = ic2n( itri, 1 )
      node2 = ic2n( itri, 2 )
      node3 = ic2n( itri, 3 )
      edge1 = ic2e( itri, 1, 1 )
      edge2 = ic2e( itri, 2, 1 )
      edge3 = ic2e( itri, 3, 1 )
```

Obtain the cell-averaged coefficient of viscosity.

```
      call viscos ( vnu )
```

Determine the appropriate nodal distribution via cyclical permutation of indicies to assign the viscous fluctuation.

$$\phi_{v_i} = -\frac{\bar{\nu}}{4\Omega}\sum_{j=1}^{3} u_j \mathbf{n}_{j+1}\cdot\mathbf{n}_{i+1}$$

```
      if ( nodegs .eq. node1 ) then
        vist( nodegs ) = vist(nodegs) - 0.25 * vnu / area(itri) *
$           ( u(node1) * dxyz(edge2,ndim+1)**2 +
$           u(node2) * ic2e(itri,3,2) * ic2e(itri,2,2) * (
$           dxyz(edge3,2) * dxyz(edge2,2) +
$           dxyz(edge3,1) * dxyz(edge2,1) ) +
$           u(node3) * ic2e(itri,1,2) * ic2e(itri,2,2) * (
$           dxyz(edge1,2) * dxyz(edge2,2) +
$           dxyz(edge1,1) * dxyz(edge2,1) ) )
        csum = csum + 0.25 * vnu / area(itri) *
$           dxyz(edge2,ndim+1)**2
      else if ( nodegs .eq. node2 ) then
        vist( nodegs ) = vist(nodegs) - 0.25 * vnu / area(itri) *
$           ( u(node2) * dxyz(edge3,ndim+1)**2 +
$           u(node3) * ic2e(itri,1,2) * ic2e(itri,3,2) * (
$           dxyz(edge1,2) * dxyz(edge3,2) +
$           dxyz(edge1,1) * dxyz(edge3,1) ) +
$           u(node1) * ic2e(itri,2,2) * ic2e(itri,3,2) * (
$           dxyz(edge2,2) * dxyz(edge3,2) +
$           dxyz(edge2,1) * dxyz(edge3,1) ) )
        csum = csum + 0.25 * vnu / area(itri) *
$           dxyz(edge3,ndim+1)**2
      else if ( nodegs .eq. node3 ) then
        vist( nodegs ) = vist(nodegs) - 0.25 * vnu / area(itri) *
$           ( u(node3) * dxyz(edge1,ndim+1)**2 +
$           u(node1) * ic2e(itri,2,2) * ic2e(itri,1,2) * (
$           dxyz(edge2,2) * dxyz(edge1,2) +
$           dxyz(edge2,1) * dxyz(edge1,1) ) +
$           u(node2) * ic2e(itri,3,2) * ic2e(itri,1,2) * (
$           dxyz(edge3,2) * dxyz(edge1,2) +
$           dxyz(edge3,1) * dxyz(edge1,1) ) )
        csum = csum + 0.25 * vnu / area(itri) *
```



```
$           dxyz(edge1,ndim+1)**2
      end if
    end do                        !  looping on triangles
```

Finite volume

Treat the viscous term as a flux at the median dual face, formed as a central difference average of gradients to either side of the face.

```
  else                            !  finite volume form
    do nedg = 1, nodcon( nodegs, 2 )
      iedg = in2e( nodegs, nedg, 1 ) !  Current edge
      node1 = nodegs
      idir = in2e( node1, nedg, 2 )
      node2 = ie2n( iedg, (3+idir)/2 )
      do irl = 1, 2               !  to right and left of edge
```

Determine viscosity coefficient.

```
        call viscos ( vnu )
```

Form the average gradient $\frac{1}{2}(\nabla u_{in} + \nabla u_{out})$. If this is a boundary edge the use $\nabla u = 0$ outside the domain. Then form the contribution to the physical viscosity update as,

$$\texttt{vist} = \nu \nabla u \cdot \hat{n}\, \Delta\Gamma$$

Try to get the timestep contribution in a manner similar to the finite element method. Boundary edges give no contribution to the timestep restriction(?).

```
        if ( ie2c(iedg,irl) .lt. 0 ) then ! boundary edge
          gradavx = 0.5 * gradu(node1,1)
          gradavy = 0.5 * gradu(node1,2)
          vist(node1) = vist(node1) + vnu *
$             ( gradavx * edgnorm(iedg, irl, 1) +
$             gradavy * edgnorm(iedg, irl, 2) )
        else
          gradavx = 0.5 * ( gradu(node1,1) + gradu(node2,1) )
          gradavy = 0.5 * ( gradu(node1,2) + gradu(node2,2) )
          vist(node1) = vist(node1) + vnu *
$             ( gradavx * edgnorm(iedg, irl, 1) * idir +
$             gradavy * edgnorm(iedg, irl, 2) * idir )
          csum = csum + 0.75 * vnu / area(ie2c(iedg,irl)) *
$             (edgnorm(iedg,irl,1)**2 + edgnorm(iedg,irl,2)**2)
        end if
      end do      !  right or left of edge
    end do        !  loop on surrounding edges
  end if          !  viscous computation
  if ( diagvis .gt. 0 ) write (6,*) 'Node=', nodegs, ' vist=',
$     vist(nodegs),' csum=', csum,' cusum=', cusum
```

Nodal update

Assign update to the current node. If `iminvis` is set, then only use the greater of the artificial viscosity and the true viscosity.

```
if ( iminvis .eq. 0 ) then
  du(nodegs) = cusum + vist(nodegs)
else
  if ( abs( visa(nodegs) ) .gt. abs( vist(nodegs) ) ) then
    du(nodegs) = cusum
  else
    du(nodegs) = cusum - visa(nodegs) + vist(nodegs)
  end if
end if
if (diagnos .ge. 1) write (6,*) 'dugs=', du(nodegs)

return
end
```


Roe

Compute the flux for a Roe-scheme through a face. Given u_{in}, u_{out}, the flux function, $\bar{\mathcal{F}}$, and $\bar{n} = (n_x, n_y)$, the outward scaled normal, evaluate the integrated flux as,

$$\text{flux} = \bar{\mathcal{F}} \cdot \bar{n} + \Phi$$

where the upwinding artificial dissipation function, Φ, is,

$$\Phi = -\frac{|\alpha|}{2}(u_{out} - u_{in})$$

with,

$$\alpha = \bar{\mathcal{A}} \cdot \bar{n} = \tilde{A} n_x + \tilde{B} n_y$$

The tensor of the flux Jacobian, $\bar{\mathcal{A}} = (\tilde{A}, \tilde{B})$, must be evaluated as the conservative linearization due to Roe[6].

$\bar{\mathcal{F}} = (F, G)$ is evaluated at the face as,

$$F_{face} = \frac{F_{in} + F_{out}}{2}, \quad G_{face} = \frac{G_{in} + G_{out}}{2}$$

Similarly, the conservative linearizations are,

$$\tilde{A} = \frac{A_{in} + A_{out}}{2}, \quad \tilde{B} = \frac{B_{in} + B_{out}}{2}$$

Called by `spacenode`. Calls `wavespeed`.

```
subroutine roe ( Uin, Uout, anx, any, x, y, flux, alph, Phi )
```

Form the flux Jacobians, $\bar{\mathcal{A}}$.

```
call wavespeed( x, y, Uin, Ain, Bin, iburg )
call wavespeed( x, y, Uout, Aout, Bout, iburg )
Atilde = 0.5 * ( Ain + Aout )
Btilde = 0.5 * ( Bin + Bout )
```

The F and G analytical fluxes.

```
if ( iburg .eq. 1 ) then                    !  Burger's equation
  Fin = 0.5 * Uin**2
  Fout = 0.5 * Uout**2
else                                        !  Linear advection
  Fin = Uin * Ain
  Fout = Uout * Aout
end if
Gin = Uin * Bin
Gout = Uout * Bout
```

The artificial dissipation Φ.

```
alph = ( Atilde * anx + Btilde * any )   !  scaled outward wavespeed
Phi = -0.5 * (Uout - Uin) * abs( alph )
```

The numerical flux.

```
      flux = 0.5*(Fin + Fout)*anx + 0.5*(Gin + Gout)*any + Phi

      return
      end
```


Viscos

Compute the local coefficient of viscosity.
Called by spacenode.

```
      subroutine viscos ( vnu )
      ichoice = 2                !  option for viscosity

      if ( ichoice .eq. 0 ) then
        vnu = 0.
      else if ( ichoice .eq. 1 ) then
        vnu = 1.
      else if ( ichoice .eq. 2 ) then
        vnu = 1.e-1
c       vnu = 1.e-2
c       vnu = 2.e-3
c       vnu = 1.e-3
c       vnu = 1.e-6
      end if

      return
      end
```

Wavespeed

Define the wavespeeds in cartesian components, $\vec{\lambda} = (a,\ b)$, at a point. The wavespeed is the flux Jacobian, $A = \frac{\partial F}{\partial U}$.

Called by `spacenode` and `roe`.

```
      subroutine wavespeed ( x, y, u, speedx, speedy, iburg)
      integer diagnos
      diagnos = 0
      if ( diagnos .ne. 0 ) write (6,*) ' in Wavespeed with ',
     $     'diagnos = ', diagnos

      speedz = 0.        !  2-D only
      idir   = 7         !  Choice of wavespeed
      iburg  = 0         !  Linear advection or Burger's equation flag
```

Uniform advection at 45-deg. $\lambda = (\pm 1, 1)$

```
      if ( idir .eq. 1 ) then
        speedx = 1.
        speedy = 1.
```

Vertical advection. $\lambda = (0, \pm 1)$

```
      else if ( idir .eq. 2 ) then
        speedx = 0.
        speedy =-1.
```

Horizontal advection. $\lambda = (1, 0)$

```
      else if ( idir .eq. 3 ) then
        speedx = 1.
        speedy = 0.
```

Circular advection. $\lambda = (y, -x)$

```
      else if ( idir .eq. 4 ) then
        speedx =  y
        speedy = -x
```

Burger's equation. $\lambda = (u, 1)$

```
      else if ( idir .eq. 5 ) then
        speedx = u
        speedy = 1.
        iburg  = 1
```

Heat equation. $\lambda = (u, 1)$

```
      else if ( idir .eq. 6 ) then
        speedx = 0.
        speedy = 0.
```

Smith and Hutton problem. $\lambda = \big(2y(1-x^2),\ -2x(1-y^2)\big)$


```
      else if ( idir .eq. 7 ) then
        speedx = 2. * y * ( 1 - x**2 )
        speedy = -2. * x * ( 1 - y**2 )
      end if

      if ( diagnos .gt. 0 ) then
        write (6,*) ' speedx,y,z'
        write (6,*) speedx, speedy, speedz
      end if

      return
      end
```

Rburg

The script file to run PECLET is `rpeclet`. Execute in the run directory.

```
# script to run Peclet code

echo ' '
echo 'running Peclet'
cp initial/flow.b .
peclet
echo 'Peclet completed'

echo 'Postprocessing'
cd postplt
postplt
post2
```

Mkpeclet

The script to make PECLET is `mkpeclet`. Execute in the `solver` directory *via* `make -f mkpeclet`.

```
#  Makefile for Peclet code

#  Shell type used throughout makefile:
SHELL=/bin/sh

#  The desired source code
SRC1=peclet

#  Compiler and link flags
#CFLAG=-r8
CFLAG=-trapuv -check_bounds
LFLAG=-lfpe

#  Sets of include files
FLS1=parms
FLS2=dims

#  These files depend on all include files
FLS3=main domain getgradu spacenode

#  These files depend only on the first set of include files
FLS4=initial osoln leastsquare

#  These files do not depend on any include files
FLS5=flsp inputs limiter roe viscos wavespeed
```



```
#  Make procedure
default: dumstart ../$(SRC1) dumend

#  Link object files
dumstart:
-@for FILE in $(FLS3:=.o); do ln -s objs/$$FILE $$FILE; done
-@for FILE in $(FLS4:=.o); do ln -s objs/$$FILE $$FILE; done
-@for FILE in $(FLS5:=.o); do ln -s objs/$$FILE $$FILE; done

#  Make the fortran code
../$(SRC1) : $(FLS1:=.inc) $(FLS2:=.inc) $(FLS3:=.o) $(FLS4:=.o) $(FLS5:=.o)
echo 'Compiling '
f77 -o ../$(SRC1) $(CFLAG) $(LFLAG) $(FLS3:=.o) $(FLS4:=.o) $(FLS5:=.o)

#  Remove object links
dumend:
@for FILE in $(FLS3:=.o); do rm $$FILE; done
@for FILE in $(FLS4:=.o); do rm $$FILE; done
@for FILE in $(FLS5:=.o); do rm $$FILE; done

#  How to obtain the object files from the source files
.f.o:
f77 -c $(CFLAG) $<

#  Dependencies
$(FLS3:=.o) : $(FLS1:=.inc) $(FLS2:=.inc)
$(FLS4:=.o) : $(FLS1:=.inc)
```

Makefile

Execute `make` to generate `.dvi` file for electronic viewing. Execute `make hardcopy` to print a double-sided, 2-up copy.

```
#  Makefile for viewing and printing fortran/latex code

#  Shell type used throughout makefile:
SHELL=/bin/sh

#  Path to search for file dependencies
VPATH=..:../..

#  Root name for .dvi file
PROG=peclet

#  Tex-only files .dvi file depends on
SRC1=$(PROG)
```

```
#  Figures .dvi file depends on
SRC2=gsnode

#  ASCII files for .dvi
SRC3=indeck Makefile mkpeclet rpeclet

#  Include files for .dvi
SRC4=parms dims

#  Fortran files for .dvi
SRC50=domain flsp getgradu initial inputs limiter main
SRC5= $(SRC50) osoln roe spacenode viscos wavespeed

#  Fortran to Latex converter
F2L=f2latex.pl

#  How to handle implicit makes
.SUFFIXES: .tex .inc $(SUFFIXES)

#  make with no arguements--Latex document for viewing
default: viewtex

#  How to view the .dvi file--make viewtex
viewtex: dumstart $(PROG).dvi dumend

#  To make a hardcopy (2up, double-sided)--make hardcopy
hardcopy: viewtex
  xdvi $(PROG)     #  Kill at this point if not right
  cp p2active.tex twoprt.tex
  latex $(PROG)
  cp p2inact.tex twoprt.tex
  dvips $(PROG)
  cat $(PROG).ps | rsh ab00 pslpr -m -Plzr -SDuplex=DuplexTumble
  lpq -Plzr

dumstart:
-@for FILE in $(SRC5:=.f); do cp ../$$FILE .; done
-@for FILE in $(SRC4:=.inc); do cp ../$$FILE .; done


#  Dependencies and how to make .dvi file
$(PROG).dvi:$(SRC1:=.tex) $(SRC2:=.eps) $(SRC3) $(SRC4:=.tex) $(SRC5:=.tex)
    latex $(PROG)
    -@grep -c 'Citation .* undefined.' $(PROG).log && bibtex $(PROG)
    -@grep -c 'Rerun to get cross' $(PROG).log && latex $(PROG)
    -@grep -c 'may have changed' $(PROG).log && latex $(PROG)

#  How to convert .inc files to .tex
.inc.tex:
$(F2L) -s $< > $*.tex
```



```
#  How to convert .f files to .tex
.f.tex:
$(F2L) -s $< > $*.tex

dumend:
@for FILE in $(SRC5:=.f); do rm -f $$FILE; done
@for FILE in $(SRC4:=.inc); do rm -f $$FILE; done
```

References

[1] Barth, T. J., "Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations," *Computational Fluid Dynamics*, No. 1994–04 in Lecture Series, von Karman Institute for Fluid Dynamics, 1994.

[2] Barth, T. J. and Jesperson, D. C., "The Design and Application of Upwind Schemes on Unstructured Meshes," AIAA Paper 89–0366, Jan. 1989.

[3] Sweby, P. K., "High Resolution Schemes Using Flux Limiters for Hyperbolic Conservation Laws," *SIAM Journal of Numerical Analysis*, Vol. 21, 1984, pp. 995–1011.

[4] van Leer, B., "Towards the Ultimate Conservative Scheme. II. Monotonicity and Conservation Combined in a Second Order Scheme," *Journal of Computational Physics*, Vol. 14, 1974, pp. 361–370.

[5] van Albada, G. D., van Leer, B., and Roberts, W. W., "A Comparative Study of Computational Methods in Cosmic Gas Dynamics," Report 81–24, ICASE, Hampton, Aug. 1981.

[6] Roe, P. L., "Discrete Models for the Numerical Analysis of Time-Dependent Multidimensional Gas Dynamics," *Journal of Computational Physics*, Vol. 63, 1986, pp. 458–476.

Grid Pre-Processor for 2-D Non-Linear Advection Solver

William A. Wood*
NASA Langley Research Center

October 10, 2000

Abstract

DOMAIN reads in the grid and boundary pointer as generated by various stand-alone grid generators. The logic is performed to identify all needed edge, element, and node pointers. Calculations of cell areas, face normals, and other grid constants are performed. A binary file is output of all grid related constants. Also output are the necessary dimensions for the non-linear hyperbolic equation solver, so that it may be compiled in a minimum-memory fashion.

Domain

```
program domain
include 'dims.inc'
logical bccheck
integer diagnos
diagnos = 1
```

Inputs

Read the externally generated grid and boundary pointer information.

```
call ingrid
```

Connectivities

Reconnect cells to nodes so that the first 2 edges are the most perpendicular.

*Aerospace Technologist, Aerothermodynamics Branch, Aero- and Gas-Dynamics Division. w.a.wood@larc.nasa.gov (757) 864-8355.


```
do ic = 1, nc
  icn1 = ic2n(ic,1)
  icn2 = ic2n(ic,2)
  icn3 = ic2n(ic,3)
  eleng1 = sqrt( ( xyz(icn2,1) - xyz(icn1,1) )**2 +
$       (xyz(icn2,2) - xyz(icn1,2) )**2 )
  eleng2 = sqrt( ( xyz(icn3,1) - xyz(icn2,1) )**2 +
$       (xyz(icn3,2) - xyz(icn2,2) )**2 )
  eleng3 = sqrt( ( xyz(icn1,1) - xyz(icn3,1) )**2 +
$       (xyz(icn1,2) - xyz(icn3,2) )**2 )
  if ( eleng1 .gt. eleng2 .and. eleng1 .gt. eleng3 ) then
    ic2n(ic,1) = icn2
    ic2n(ic,2) = icn3
    ic2n(ic,3) = icn1
  else if ( eleng2 .gt. eleng1 .and. eleng2 .gt. eleng3 ) then
    ic2n(ic,1) = icn3
    ic2n(ic,2) = icn1
    ic2n(ic,3) = icn2
  end if
end do
```

Given cell-to-node connectivities, determine cell, edge, and node dependencies. Begin by zeroing summations.

```
do i = 1, 2
  do node = 1, nn
    nodcon( node, i ) = 0
  end do
end do
ne = 0        ! number of edges
iconc = 0     ! maximum numer of cells connected to a node
icone = 0     ! maximum numer of edges connected to a node
do i1 = 1, 2
  do i2 = 1, nc
    ic2e(i2, 4, i1) = 0
  end do
end do
```

Since we know cell-to-node, the outer loop is over cells.

```
do icel = 1, nc
```

Then inner loop over the nodes defining this cell.

```
  do i = 1, 4
```

Pick out the current node and the next node, with the current edge between these two nodes. Jump out after three sides if this is a triangle.

```
    inod = ic2n( icel, i )
    if ( inod .eq. 0 ) goto 101
    if ( i .eq. 4 .or. ic2n(icel, i+1) .eq. 0 ) then
```

```
          inod1 = ic2n(icel, 1)
        else
          inod1 = ic2n( icel, i+1 )
        end if
```

For node-to-cell pointer, first increment the counter telling how many cells contain this node. Then attach the current cell to this node.

```
        nodcon(inod, 1) = nodcon(inod, 1) + 1
        iconc = max( iconc, nodcon(inod, 1) )
        if ( nodcon(inod,1) .gt. 40 ) then
          write (6,*) 'Increase dimensions in dims.inc for '
          write (6,*) 'number of cells connected to node ', iconc
          stop
        end if
        in2c(inod, nodcon(inod, 1), 1) = icel
        in2c(inod, nodcon(inod, 1), 2) = i
```

Does the current edge exist? Loop over all previously defined edges connected to this node to find out. First case is an error where edge is traversed twice in same direction. Second case is edge previously defined in opposite direction. Third case is this is a new edge, so define it.

```
        do ie3 = 1, nodcon(inod, 2)
          ie = in2e( inod, ie3, 1 )
          if ( ie2n(ie,1) .eq. inod
     $          .and. ie2n(ie,2) .eq. inod1 ) then
            write (6,*) 'Error: edge traversed twice in same ',
     $            'direction, cell ',icel,' node ', inod
            stop
          else if ( ie2n(ie,1) .eq. inod1
     $          .and. ie2n(ie,2) .eq. inod ) then
            ic2e(icel, i, 1) = ie
            ic2e(icel, i, 2) = -1     !  Current cell to right of edge
            ie2c(ie, 1) = icel
            goto 102
          end if
        end do
```

This is a new edge, so attach to the cell in the forward direction.

```
        ne = ne + 1
        if ( ne .gt. nemax ) then
          write (6,*) 'Too many edges, ne=',ne,', nemax=',nemax
          write (6,*) ' *** stopping ***'
          stop
        end if
        ic2e(icel, i, 1) = ne
        ic2e(icel, i, 2) = 1      !  Current cell to left of edge
        ie2c(ne, 2) = icel        !  Set right cell to boundary
        ie2c(ne, 1) = -1          !  Overwritten later if not a boundary
```

Define the edge endpoints.


```
        ie2n(ne, 1) = inod
        ie2n(ne, 2) = inod1
```

Tell the end nodes that this edge emanates from them.

```
        nodcon(inod, 2) = nodcon(inod, 2) + 1
        nodcon(inod1,2) = nodcon(inod1,2) + 1
        icone = max( icone, nodcon(inod,2), nodcon(inod1,2) )
        if ( nodcon(inod,2) .gt. 40 ) then
          write (6,*) 'Increase dimensions in dims.inc for '
          write (6,*) 'number of edges connected to node ', icone
          stop
        end if
        if ( nodcon(inod1,2) .gt. 40 ) then
          write (6,*) 'Increase dimensions in dims.inc for '
          write (6,*) 'number of edges connected to node ', icone
          stop
        end if
        in2e(inod,  nodcon(inod, 2), 1) = ne
        in2e(inod1, nodcon(inod1,2), 1) = ne
        in2e(inod,  nodcon(inod, 2), 2) =  1
        in2e(inod1, nodcon(inod1,2), 2) = -1
102     continue
      end do                  !   nodes and edges of a cell
101   continue
    end do                    !   next cell
```

Diagnostics

```
    if ( diagnos .eq. 20 .or. diagnos .eq. 21 ) then
      write (6,*)
      write (6,*) ' There are ', nc, ' cells'
      do ic = 1, nc
        write (6,*) 'Cell ', ic
        write (6,*) 'Defining nodes ', (ic2n(ic,in),in=1,4)
        write (6,*) 'Edges ', (ic2e(ic,in,1),in=1,4)
        write (6,*) 'Directions ', (ic2e(ic,in,2),in=1,4)
      end do
    end if
    if ( diagnos .eq. 20 .or. diagnos .eq. 22 ) then
      write (6,*)
      write (6,*) ' There are ', ne, ' edges'
      do ie = 1, ne
        write (6,*) 'Edge ', ie
        write (6,*) 'Defining nodes ', (ie2n(ie,in),in=1,2)
      end do
    end if
    if ( diagnos .eq. 20 .or. diagnos .eq. 23 ) then
      write (6,*)
      write (6,*) ' There are ', nn, ' nodes'
```

```
      do in = 1, nn
        write (6,*) 'Node ', in
        write (6,*) 'Number of cells connected ', nodcon(in,1)
        write (6,*) 'Cells ', (in2c(in,ic,1),ic=1,nodcon(in,1))
        write (6,*) 'Node number ', (in2c(in,ic,2),ic=1,nodcon(in,1))
        write (6,*) 'Number of edges connected ', nodcon(in,2)
        write (6,*) 'Edges ', (in2e(in,ie,1),ie=1,nodcon(in,2))
        write (6,*) 'Direction ', (in2e(in,ie,2),ie=1,nodcon(in,2))
      end do
    end if
```

Order interior nodes

Define the interior nodes and order for the update sweep.

```
    nin = 0     !  Interior node counter
```

Loop over all nodes.

```
    do n1 = 1, nn
```

Reset boundary indicator.

```
      bccheck = .false.
      if ( diagnos .eq. 3 ) write (6,*) 'n1 = ', n1,' bccheck = ',
$       bccheck
```

Check if this node is a boundary node.

```
      do n2 = 1, nbn
        if ( in2b(n2,1) .eq. n1 ) bccheck = .true.
        if ( diagnos .eq. 3 ) write (6,*) 'n2 = ', n2,
$          ' in2b(n2,1) = ', in2b(n2,1), ' bccheck = ', bccheck
      end do
      if ( diagnos .eq. 3 ) write (6,*) 'n1 = ', n1,' bccheck = ',
$       bccheck
```

If not a defined boundary point then treat as interior point.

```
      if ( .not. bccheck ) then
        nin = nin + 1
        intnod(nin) = n1
      end if
    end do
```

Verify nbn + nin = nn.

```
    if ( (nin + nbn) .ne. nn .or. diagnos .eq. 3 ) then
      write (6,*) 'Check interior plus boundary = all nodes'
      write (6,*) 'nn, nbn, nin ', nn, nbn, nin
    end if
```


Geometric computations

```
    call metric
```

Write grid data

Diagnostic check 1: echo grid, boundary information, and areas.

```
    if ( diagnos .ge. 1 ) call ogrid
    call outmesh ( iconc + 26, icone + 26 )
    write (6,*) ' Number of nodes = ', nn
    write (6,*) ' Number of cells = ', nc
    write (6,*) ' Number of edges = ', ne
    write (6,*) ' Normal completion of Domain'
    stop
    end
```

Ingrid

Input grid and boundary information. Grid format is set by `idom1`. Boundary node numbering and bctype are in file `bcpoint.dat`. Called from `domain`. Calls `tecin` and `vgridin`.

idom1 = 10 Read grid in TECPLOT[1] format from `grid.dat`.
11 Read grid and boundary pointer information in VGRID[2] format from `grid.front`.

```
    subroutine ingrid
    include 'dims.inc'
```

Input mesh

```
    write (6,101)
 101 format ( 'Enter idom1 - options for reading external grid',/,
$     'idom1 = 10 - Tecplot finite element format',/,
$     'idom1 = 11 - Vgrid .front format')
    read (5,*) idom1
```

TECPLOT format mesh.

```
    if ( idom1 .eq. 10 ) then
      call tecin
```

VGRID format mesh.

```
    else if ( idom1 .eq. 11) then
      call vgridin
    else
      write (6,*) 'Error in Ingrid: idom1 = ', idom1
    end if
```

Input boundary pointer information.

```
      open ( 12, file = 'bcpoint.dat', form = 'formatted' )
      rewind( 12 )
      read  (12,*)                        ! comment line
      read (12,*) nbn
      do n = 1, nbn
        read  (12,*) (in2b(n,i), i=1,2)
      end do
      close( 12 )

      return
      end
```

Tecin

Read in a finite-element format TECPLOT grid file. Called from `ingrid`.

```
      subroutine tecin
      include 'dims.inc'
      character a1*7, a2*3, a3*14, a4*4, a5*1
      integer diagnos
      diagnos = 0
```

Open input file and read header information.

```
      open ( 8, file = 'grid.dat', form = 'formatted' )
      rewind (8)

      read (8,12) a5
 12   format ( 21x, a1 )
      read  (8,11) a1, nn, a2, nc, a3, a4
c11   format ('ZONE N=',i7,' E=',i7,' F=FEPOINT ET=TRIANGLE')
 11   format ( a7, i7, a3, i7, a14, a4 )
```

Is grid in 2 or 3 dimensional coordinates?

```
      if ( a5 .eq. 'Z' .or. a5 .eq. 'z' ) then
        ndim = 3
        write (6,*) 'Error: 3D - this is a 2D code only'
        write (6,*) 'Stopping in Tecin'
        stop
      else
        ndim = 2
      end if

      if ( diagnos .eq. 1 ) then
        write (6,*)
        write (6,*) 'diagnostic level 1 in ingrid'
        write (6,*) 'a1, a2, a3, a4, a5'
        write (6,*) a1,',',a2,',',a3,',',a4,',',a5
```



```
        write (6,*) ' nn, nc'
        write (6,*) nn, nc
        write (6,*) ' ndim = ', ndim
        write (6,*)
      end if
```

Error checking on dimensions.

```
      if ( nn .gt. nnmax .or. nc .gt. ncmax ) then
        write( 6, 100 ) nn, nnmax, nc, ncmax
 100    format ( /'Dimensions error in Ingrid'/'nn, nnmax ', 2i8/
     $       'nc, ncmax ', 2i8 )
        write (6,*) ' *** stopping  *** '
        stop
      end if
```

Read in nodal coordinates.

```
      do in = 1, nn
        read (8,*) ( xyz(in,jdim), jdim=1,ndim )
      enddo
```

Read in connectivities.

```
      do ic = 1, nc
```

If only triangular elements, read in 3 bounding nodes.

```
        if ( a4 .eq. 'TRIA' .or. a4 .eq. 'tria' ) then
          read  (8,*) ( ic2n(ic,jdim), jdim=1,3 )
```

Otherwise we could have a mixture of triangles and quads for elements. If the fourth node is repeated, the element is a triangle, and we reset the fourth node to null.

```
        else if ( a4 .eq. 'QUAD' .or. a4 .eq. 'quad' ) then
          read (8,*) ( ic2n(ic,jdim), jdim=1,4)
          if ( ic2n(ic,4) .eq. ic2n(ic,3) ) ic2n(ic,4) = 0
        else
          write (6,*) 'Error in Ingrid: a4 = ', a4
          write (6,*) ' *** stopping *** '
          stop
        end if
      end do
      close (8)

      return
      end
```

Vgridin

Read in a triangulated 2-D grid in x,y,z coordinates from the ASCII file `grid.front` in VGRID format. Called from `ingrid`.

```
      subroutine vgridin
      include 'dims.inc'
```

Open the data file and read the header line.

```
      open ( 12, file = 'grid.front', form = 'formatted' )
      rewind (12)
      read (12,*)
```

Extract the number of nodes and elements from the next line. Then do error checking.

```
      read (12,*) idum1, idum2, nn, nc
      if ( nn .gt. nnmax ) then
        write (6,*) ' Error in vgridin  ******'
        write (6,*) ' Not dimensioned for enough nodes.'
        write (6,*) ' nn is ', nn
        write (6,*) ' But only dimensioned for nnmax = ', nnmax
        stop
      end if
      if ( nc .gt. ncmax ) then
        write (6,*) ' Error in vgridin  ******'
        write (6,*) ' Not dimensioned for enough elements.'
        write (6,*) ' nc is ', nc
        write (6,*) ' But only dimensioned for ncmax = ', ncmax
        stop
      end if
```

For each triangular element there are 3 nodes. Now we read into the pointer `ic2n` the node-numbers of the 3 bounding nodes, ordered in a right-hand rule. Since these are triangular elements from VGRID, we set the fourth node to null.

```
      do ic = 1, nc
        read (12,*) idum1, ( ic2n(ic, node), node=1,3)
        ic2n(ic, 4) = 0
      end do
```

For each node we read in the x, y, and z coordinates. If the variation in the z dimension is non-zero, then set $ndim = 3$, otherwise we neglect the z dimension and set $ndim = 2$.

```
      ndim = 2
      zsum = 0.
      do in = 1, nn
        read (12,*) idum1, ( xyz(in, ixyz), ixyz=1, 3)
        zsum = zsum + xyz(in,3)
      end do
      zsum = zsum / float(nn)
      do in = 1, nn
        if ( abs( xyz(in,3) - zsum ) .gt. 1.e-5 ) then
          ndim = 3
          write (6,*) 'Error - using 3D grid in 2D code'
```



```
          write (6,*) 'stopping in Vgridin'
          stop
        end if
      end do
      close (12)

      return
      end
```

Ogrid

Output the grid in TECPLOT[1] finite-element format to `domain.dat`. Boundary conditions written as field variable `u`. Called from `domain`.

```
      subroutine ogrid
      include 'dims.inc'
      dimension u( nnmax )
      logical itriangle, iquad
      itriangle = .false.
      iquad     = .false.
```

Open the output file and write header information.

```
      open ( 8, file = "domain.dat" )
      rewind (8)

      if ( ndim .eq. 2 ) write (8,20)
      if ( ndim .eq. 3 ) write (8,30)
 20   format ('VARIABLES = "X" "Y" "bc" "Amd" "A"')
 30   format ('VARIABLES = "X" "Y" "Z" "bc" "Amd" "A"')
```

Assign the boundary identifications. If an interior point, then $u = 0$. Otherwise, $u = bctype$.

```
      do i = 1, nn
        u(i) = 0.
      end do
      do i = 1, nbn
        u( in2b(i,1) ) = float( in2b(i,2) )
      end do
```

Do we have quads, triangles, or both? Check the fourth element of the cell-to-node pointer. If it is null, then the element is a triangle. If it is non-null, then the element is a quad. If we have mixed triangle and quad elements, we must treat all elements as quads for TECPLOT compatibility.

```
      do ic = 1, nc
        if ( ic2n(ic,4) .eq. 0 ) then
          itriangle = .true.
        else if ( ic2n(ic,4) .gt. 0 ) then
          iquad = .true.
```

```
        else
          write (6,*) 'Error in Ogrid: ic2n(',ic,',4) = ',
     $         ic2n(ic,4), ' < 0'
          stop
        end if
      end do

      if ( iquad ) then
        write (8,11) nn, nc
 11     format ('ZONE N=',i7,' E=',i7,' F=FEPOINT ET=QUADRILATERAL')
      else
        write (8,12) nn, nc
 12     format ('ZONE N=',i7,' E=',i7,' F=FEPOINT ET=TRIANGLE')
      end if
```

Nodal coordinates

```
      do in = 1, nn
        write (8,'(6e11.3)') ( xyz(in,jdim), jdim=1,ndim ), u(in),
     $       areamd(in), areamd(in)/float(nodcon(in,1))
      enddo
```

Connectivities

Triangles only.

```
      if ( itriangle .and. .not. iquad ) then
        do ic = 1, nc
          write (8,*) ( ic2n(ic,jdim), jdim=1,3 )
        enddo
```

Quads only.

```
      else if ( .not. itriangle .and. iquad ) then
        do ic = 1, nc
          write (8,*) ( ic2n(ic,jdim), jdim=1,4 )
        enddo
```

Triangle and quad mixture. Repeat third triangle node to get four nodes for all elements.

```
      else
        do ic = 1, nc
          if ( ic2n(ic,4) .eq. 0 ) then
            write (8,*) ( ic2n(ic,jdim), jdim=1,3 ), ic2n(ic,3)
          else
            write (8,*) ( ic2n(ic,jdim), jdim=1,4 )
          end if
        enddo
      end if
      close (8)
```



```
      return
      end
```

Metric

Compute grid constants including lengths, areas, normals, and position vectors. Called from `domain`.

```
      subroutine metric
      include 'dims.inc'
      integer diagnos
      diagnos = 0
      if (diagnos .ne. 0) write(6,*)' in Metric: diagnos = ',diagnos
```

Compute Δx, Δy, Δz edge lengths

Loop over edges.

```
      do ie = 1, ne
```

Loop on x, y, (and z). $\Delta x = x_2 - x_1$, etc. Edge length stored after the Δ's. Also check for degenerate (zero length) edges.

```
        alength = 0.
        do ixyz = 1, ndim
          dxyz(ie, ixyz) = xyz( ie2n(ie, 2), ixyz ) -
     $         xyz( ie2n(ie, 1), ixyz )
          alength = alength + dxyz(ie, ixyz)**2
          if ( diagnos .eq. 2 )
     $         write (6,*) 'dxyz(', ie, ',', ixyz, ')=',dxyz(ie,ixyz)
        end do
        dxyz(ie, ndim+1) = sqrt( alength )
        if ( diagnos .gt. 0 )
     $       write (6,*) 'alength(', ie, ')= ', sqrt(alength)
        if ( sqrt(alength) .lt. 1.e-6 ) then
          write (6, 101) ie, ie2n(ie,1), ie2n(ie,2),
     $         dxyz(ie,1), dxyz(ie,2), dxyz(ie,3)
          write (6,*) ' *** stopped ***'
          stop
        end if
      end do

 101  format ( 'Degenerate edge computed in Metric, edge ', i5,/
     $     ' between nodes ', i5, ' and ', i5, /
     $     'dx, dy, dz : ', 3e12.5 )
```

Areas

Compute cell areas. Area of a triangle is half the cross-product of two sides. Area of a quad is the sum of two triangular areas.

$$2 \cdot S_T = (x_2 - x_1)(y_3 - y_2) - (x_3 - x_2)(y_2 - y_1)$$

Median-dual areas about each node are formed as distribution sums. Triangles contribute one-third of their area to the node,

$$S_{MD_i} = \frac{1}{3}\sum_T S_T, \quad \text{For } T \text{ joined at } i$$

Quads contribute one-fourth in an analogous manner.

```
      do in = 1, nn
        areamd( in ) = 0.
      end do
      do ic = 1, nc
        area(ic) = 0.5 * ic2e(ic,1,2) * ic2e(ic,2,2) *
$       ( dxyz( ic2e(ic,1,1), 1 ) * dxyz( ic2e(ic,2,1), 2) -
$       dxyz( ic2e(ic,2,1), 1 ) * dxyz( ic2e(ic,1,1), 2 ) )
        if ( ic2n( ic, 4 ) .ne. 0 ) then                  ! then quad
          area(ic) = area(ic) + 0.5 * ic2e(ic,3,2)*ic2e(ic,4,2) *
$         ( dxyz( ic2e(ic,3,1), 1 ) * dxyz( ic2e(ic,4,1), 2) -
$         dxyz( ic2e(ic,4,1), 1 ) * dxyz( ic2e(ic,3,1), 2 ) )
          areamd( ic2n(ic,1) ) = areamd(ic2n(ic,1)) + 0.25*area(ic)
          areamd( ic2n(ic,2) ) = areamd(ic2n(ic,2)) + 0.25*area(ic)
          areamd( ic2n(ic,3) ) = areamd(ic2n(ic,3)) + 0.25*area(ic)
          areamd( ic2n(ic,4) ) = areamd(ic2n(ic,4)) + 0.25*area(ic)
        else
          areamd(ic2n(ic,1)) = areamd(ic2n(ic,1))+0.3333333333*area(ic)
          areamd(ic2n(ic,2)) = areamd(ic2n(ic,2))+0.3333333333*area(ic)
          areamd(ic2n(ic,3)) = areamd(ic2n(ic,3))+0.3333333333*area(ic)
        end if
        if ( diagnos .gt. 0 ) write (6,*) 'area(',ic,')= ', area(ic)
        if ( area(ic) .le. 0. ) then
          write (6,*) 'DANGER - negative area for cell ', ic
          write (6,*) ' area = ', area(ic)
          write (6,*) 'nodes ',ic2n(ic,1),ic2n(ic,2),ic2n(ic,3)
          write (6,*) ' *** stopping *** '
          stop
        else if ( area(ic) .lt. 1.e-6 ) then
          write (6,*) 'WARNING - small area for cell ', ic
          write (6,*) ' area = ', area(ic)
          write (6,*) 'nodes ',ic2n(ic,1),ic2n(ic,2),ic2n(ic,3)
        end if
      end do
```

Median-Dual Normals

Normals to the median-dual areas are computed, scaled by the length of the median-dual edge. Normals are defined pointing toward node2 of the edge on the interior, and outward for boundary edges. Begin by defining all interior sides of edges *via* a loop over all cells. For schematic of geometry see Figure 1.

```
      do ic = 1, nc
```

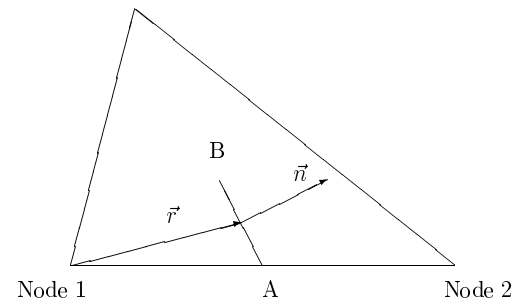



Figure 1: Geometry for median-dual definitions.

Cell centroid, B.

```
        Bx = 0.
        By = 0.
        do in = 1, 4
          node = ic2n(ic, in)
          if ( node .eq. 0 ) goto 100    !  Triangle check
          Bx = Bx + xyz(node, 1)
          By = By + xyz(node, 2)
        end do
 100    continue
        Bx = Bx / float(in - 1)
        By = By / float(in - 1)
```

Loop over the edges of this cell.

```
        do ie = 1, 4
          if (ic2n(ic,ie) .eq. 0) goto 110   ! Triangle check
          iedge = ic2e(ic,ie,1)
          irl = ( 3 + ic2e(ic,ie,2) ) / 2
```

Edge midpoint, A.

```
          Ax = 0.5 * (xyz(ie2n(iedge,1), 1) + xyz(ie2n(iedge,2), 1) )
          Ay = 0.5 * (xyz(ie2n(iedge,1), 2) + xyz(ie2n(iedge,2), 2) )
```

Scaled normal to this section of the median dual boundary.

```
          edgnorm(iedge,irl,1) =   ic2e(ic,ie,2) * ( By - Ay )
          edgnorm(iedge,irl,2) = - ic2e(ic,ie,2) * ( Bx - Ax )
          edgnorm(iedge,irl,3) = 0.
```

Position vector pointing from the nodes to the midpoint of this section of the median dual boundary.

```
        do nodet = 1, 2
          r(iedge,nodet,irl,1) = 0.5 * (Ax + Bx) -
     $          xyz(ie2n(iedge,nodet), 1)
          r(iedge,nodet,irl,2) = 0.5 * (Ay + By) -
     $          xyz(ie2n(iedge,nodet), 2)
          r(iedge,nodet,irl,3) = 0.
        end do
      end do
110   continue
    end do
```

Now handle the boundaries by looping over all edges, checking if a boundary, and defining an outward scaled normal. See Figure 2.

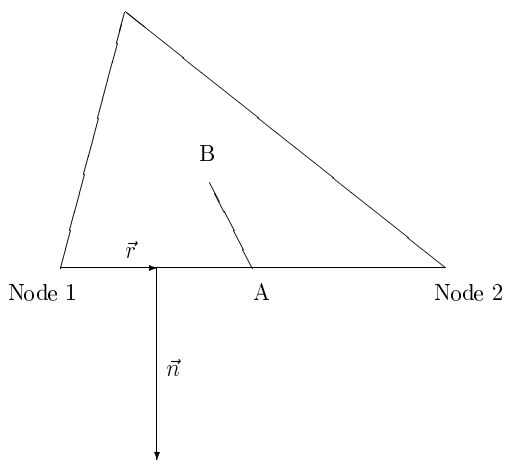


Figure 2: Geometry for boundary edge normals.

```
do ie = 1, ne
  if ( ie2c(ie,1) .eq. -1 ) then
    edgnorm(ie,1,1) = (xyz(ie2n(ie,2),2) - xyz(ie2n(ie,1),2))/2.
    edgnorm(ie,1,2) = (xyz(ie2n(ie,1),1) - xyz(ie2n(ie,2),1))/2.
    edgnorm(ie,1,3) = 0.
```

Vector r for reconstruction is to the quarter points of this edge.

```
    r(ie,1,1,1) = (xyz(ie2n(ie,2),1) - xyz(ie2n(ie,1),1)) / 4.
    r(ie,1,1,2) = (xyz(ie2n(ie,2),2) - xyz(ie2n(ie,1),2)) / 4.
```



```
    r(ie,1,1,3) = 0.
    r(ie,2,1,1) = (xyz(ie2n(ie,1),1) - xyz(ie2n(ie,2),1)) / 4.
    r(ie,2,1,2) = (xyz(ie2n(ie,1),2) - xyz(ie2n(ie,2),2)) / 4.
    r(ie,2,1,3) = 0.
  end if
end do

return
end
```

Outmesh

Write the binary file `domain.b` containing all grid information. Also write `parms.inc` file for dimensioning subsequent codes.

```
subroutine outmesh ( iconc, icone )
include 'dims.inc'
```

domain.b

```
open ( 21, file = 'domain.b', form = 'unformatted' )
write (21) nn, nc, ne, ndim, nbn, nin, iconc, icone
write (21) (intnod(i1), i1=1,nin)
write (21) ((in2b(i1,i2), i2=1,2), i1=1,nbn)
write (21) ((nodcon(i1,i2), i2=1,2), i1=1,nn)
write (21) (((in2c(i1,i2,i3), i3=1,2), i2=1,nodcon(i1,1)),
$     i1=1,nn)
write (21) (((in2e(i1,i2,i3), i3=1,2), i2=1,nodcon(i1,2)),
$     i1=1,nn)
write (21) ((ic2n(i1,i2), i2=1,4), i1=1,nc)
write (21) (((ic2e(i1,i2,i3), i3=1,2), i2=1,4), i1=1,nc)
write (21) ((ie2n(i1,i2), i2=1,2), i1=1,ne)
write (21) ((ie2c(i1,i2), i2=1,2), i1=1,ne)
write (21) ((xyz(i1,i2), i2=1,ndim), i1=1,nn)
write (21) (areamd(i1), i1=1,nn)
write (21) (area(i1), i1=1,nc)
write (21) ((dxyz(i1,i2), i2=1,ndim+1), i1=1,ne)
write (21) (((edgnorm(i1,i2,i3), i3=1,ndim), i2=1,2), i1=1,ne)
write (21) ((((r(i1,i2,i3,i4), i4=1,ndim), i3=1,2), i2=1,2),
$     i1=1,ne)
close (21)
```

parms.inc

```
if ( nn .gt. 999999 .or. nc .gt. 999999 .or. ne .gt. 999999 )
$      then
  write (6,*) 'increase format statement for nn, nc, ne',
$       nn, nc, ne
  stop
end if
open ( 22, file = 'parms.inc', form = 'formatted' )
```

```
      write (22,103)                 !  LaTeX file header stuff
      write (22,104)
      write (22,105)
      write (22,100) nn, nc, ne
      write (22,101) ndim, ndim+1, nbn, nin
      write (22,102) iconc, icone
      close (22)
 100  format (6x, 'parameter ( nnmax =', i6, ', ncmax =', i6,
     $     ', nemax =', i6, ')')
 101  format (6x, 'parameter ( ndim =', i1, ', ndim1 =', i1,
     $     ', nbn =', i6, ', nin =', i6, ')')
 102  format (6x, 'parameter ( iconc =', i2, ', icone =', i2,')')
 103  format ('c \\section{Parms.inc}' / 'c' /
     $     'c Parameter statements for {\\sc Peclet}.' /
     $     'c Resides in {\\sf solver} directory.' /
     $     'c Generated by {\\sc Domain}, in the {\\sf domain} ',
     $     ' directory.' / 'c' /
     $     'c \\begin{tabbing}' / 'c1234567890\\= \\kill')
 104  format ('c nnmax \\> Maximum number of nodes \\\ ' /
     $     'c ncmax \\> Maximum number of cells \\\ ' /
     $     'c nemax \\> Maximum number of edges \\\ ' /
     $     'c ndim  \\> Number of physical grid-dimensions:',
     $     ' 2=(x,y), 3=(x,y,z) \\\ ' /
     $     'c ndim1 \\> Number of dimensions plus one \\\ ' /
     $     'c nbn   \\> Number of boundary nodes \\\ ' /
     $     'c nin   \\> Number of interior nodes \\\ ' /
     $     'c iconc\\>Maximum number of cells connected at a node\\\ '/
     $     'c icone \\> Maximum number of edges connected at a node')
 105  format ( 'c \\end{tabbing}' /)
      return
      end
```

Dims.inc

Dimension and common block statements for DOMAIN. Basic parameters are:

nnmax	Maximum number of nodes expected
ncmax	Maximum number of cells expected
nemax	Maximum number of edges expected

```
      parameter ( nnmax =68720, ncmax =136544, nemax =205263 )
c     parameter ( nnmax =3000, ncmax =6000, nemax =9000 )
c     parameter ( nnmax = 682, ncmax =1304, nemax =2000 )
c     parameter ( nnmax = 121, ncmax = 200, nemax = 350 )
c     parameter ( nnmax = 25, ncmax = 32, nemax = 56 )
```

Variables

area	Cell areas, S_T, (cell)

areamd	Median-dual areas about each node, S_{MD}, (node)
dxyz	Δx, Δy, Δz, and length for each edge, (edge, distance)
edgnorm	Scaled normal vector toward node2 from median dual connection emanating from this edge, (edge, [R,L], Δx_i)
r	Vector $\vec{r}_E - \vec{r}_0$ pointing from the node to the quadrature point on the median-dual boundary, (edge, node[1,2], [R,L], xyz)
xyz	Nodal coordinates, (node, x_{1-3})
ic2e	Cell-to-edge pointer, (cell, edge$_{1-4}$, [1=forward, -1=backward])
ic2n	Cell-to-node pointer, ordered CCW, (cell, node$_{1-4}$)
ie2c	Edge-to-cell pointer, (edge, cell to [R,L]) Boundary edge if `ie2c(ie, 1) = -1`
ie2n	Edge-to-node pointer, (edge, node$_{1-2}$)
in2b	Node-to-boundary pointer, identifies boundary nodes and bctype Outflow boundaries need no special treatment (nbn,1) = node, (nbn,2) = bctype[10–19=inflow, 20–29=outflow]
in2c	Node-to-cell pointer, (node, 1–# connecting cells, [cell, node$_{1-4}$])
in2e	Node-to-edge pointer, (node, 1–# connected edges, [edge, (1=first node, -1=second node)])
intnod	Interior node numbering for update sweep, (node $\neq$ boundary)
nbn	Number of boundary nodes
nc	Number of cells
ndim	Number of physical grid dimensions, (2=x,y), (3=x,y,z)
nin	Number of interior nodes
nodcon	Number of cells and edges emanating from a node, (node, [# cells, # edges])
ne	Number of edges
nn	Number of nodes

```
      dimension xyz(nnmax,3), areamd(nnmax), intnod(nnmax),
     $     nodcon(nnmax,2)
      dimension in2b(nnmax,2), in2e(nnmax,10,2), in2c(nnmax,10,2)
      dimension dxyz(nemax,4), edgnorm(nemax,2,3), r(nemax,2,2,3)
      dimension ie2n(nemax,2), ie2c(nemax,2)
      dimension area(ncmax)
      dimension ic2e(ncmax,4,2), ic2n(ncmax,4)

      common / basics / r, xyz, nc, ndim, ne, nn
      common / pointers / ic2e, ic2n, ie2c, ie2n, in2c, in2e
      common / auxs / area, areamd, dxyz, edgnorm, in2b, intnod,
     $     nbn, nodcon, nin
```

References

[1] *Tecplot User's Manual: Version 7*, Bellevue, Washington, Aug. 1996.

[2] Pirzadeh, S., "Three-Dimensional Unstructured Viscous Grids by the Advancing-Layers Method," *AIAA Journal*, Vol. 34, No. 1, Jan. 1996, pp. 43–49.

Bibliography

[1] David Sidilkover. A genuinely multidimensional upwind scheme and efficient multigrid solver for the compressible Euler equations. Report 94–84, ICASE, USA, November 1994.

[2] David Sidilkover. Multidimensional upwinding and multigrid. AIAA Paper 95–1759, June 1995.

[3] D. Sidilkover and P. L. Roe. Unification of some advection schemes in two dimensions. Report 95–10, ICASE, Hampton, February 1995.

[4] Bram van Leer. Towards the ultimate conservative scheme. II. Monotonicity and conservation combined in a second order scheme. *Journal of Computational Physics*, 14:361–370, 1974.

[5] Bram van Leer. Towards the ultimate conservative scheme. V. A second-order sequel to Godunov's method. *Journal of Computational Physics*, 32:101–136, 1979.

[6] R. W. MacCormack. The effect of viscosity in hypervelocity impact cratering. AIAA Paper 69–354, 1969.

[7] R. W. MacCormack. Current status of numerical solutions of the Navier-Stokes equations. AIAA Paper 85–0032, January 1985.

[8] R. W. MacCormack and G. V. Candler. The solution of the Navier-Stokes equations using Gauss-Seidel line relaxation. *Computers and Fluids*, 17(1):133–150, January 1989.

[9] Robert W. MacCormack and Thomas Pulliam. Assessment of a new numerical procedure for fluid dynamics. AIAA Paper 98–2821, June 1998.

[10] G. V. Candler, M. J. Wright, and J. D. McDonald. A data-parallel LU-SGS method for reacting flows. AIAA Paper 94–0410, January 1994.

[11] A. G. Godfrey. *GASP Version 3 User's Manual*. Aerosoft, Inc., Blacksburg, VA, May 1996.

[12] J. L. Thomas, R. W. Walters, D. H. Rudy, and R. C. Swanson. Upwind relaxation algorithms for Euler/Navier-Stokes equations. In Sharon H. Stack, editor, *Langley Symposium on Aerodynamics*, volume 1, pages 89–107. NASA Langley Research Center, April 1985.

[13] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43(2):357–372, October 1981.

[14] S. K. Godunov. A difference method for the numerical calculation of discontinuous solutions of hydrodynamic equations. *Matematichaskiy Sbornik*, 47(89)(3):271–306, March 1959.

[15] Peter A. Gnoffo, Roop N. Gupta, and Judy L. Shinn. Conservation equations and physical models for hypersonic air flows in thermal and chemical nonequilibrium. NASA TP 2867, February 1989.

[16] P. A. Gnoffo. An upwind-biased, point-implicit relaxation algorithm for viscous, compressible perfect-gas flows. NASA TP 2953, February 1990.

[17] Bram van Leer, James L. Thomas, Philip L. Roe, and Richard W. Newsome. A comparison of numerical flux formulas for the Euler and Navier-Stokes equations. AIAA Paper 87–1104, January 1987.

[18] R. C. Swanson and E. Turkel. Aspects of a high-resolution scheme for the Navier-Stokes equations. AIAA Paper 93–3372, July 1993.

[19] T. J. Barth and D. C. Jesperson. The design and application of upwind schemes on unstructured meshes. AIAA Paper 89–0366, January 1989.

[20] Timothy J. Barth. Numerical aspects of computing viscous high Reynolds number flows on unstructured meshes. AIAA Paper 91–0721, January 1991.

[21] Timothy J. Barth. Recent developments in high order k-exact reconstruction on unstructured meshes. AIAA Paper 93–0668, January 1993.

[22] Timothy J. Barth. Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations. In *Computational Fluid Dynamics*, number 1994–04 in Lecture Series. von Karman Institute for Fluid Dynamics, 1994.

[23] David Lee Whitaker. *Two-Dimensional Euler Computations on a Triangular Mesh Using an Upwind, Finite-Volume Scheme*. PhD thesis, VPI&SU, Blacksburg, VA, December 1988.

[24] D. L. Whitaker and B. Grossman. Two-dimensional Euler computations on a triangular mesh using an upwind, finite-volume scheme. AIAA Paper 89–0470, January 1989.

[25] Michael Fey. Multidimensional upwinding. Part I. The method of transport for solving the Euler equations. *Journal of Computational Physics*, 143(1):159–180, June 1998.

[26] Michael Fey. Multidimensional upwinding. Part II. Decomposition of the Euler equations into advection equations. *Journal of Computational Physics*, 143(1):181–199, June 1998.

[27] H. Deconinck, R. Struijs, and P. L. Roe. Fluctuation splitting for multidimensional convection problems: An alternative to finite volume and finite element methods. In *Computational Fluid Dynamics*, number 1990–03 in Lecture Series. von Karman Institute for Fluid Dynamics, March 1990.

[28] P. L. Roe. "Optimum" upwind advection on a triangular mesh. Report 90–75, ICASE, Hampton, October 1990.

[29] Ch. Hirsch. A general analysis of two-dimensional convection schemes. In *Computational Fluid Dynamics*, Lecture Series 1991-01. von Karman Institute for Fluid Dynamics, February 1991.

[30] H. Deconinck, R. Struijs, G. Bourgois, and P. L. Roe. Compact advection schemes on unstructured grids. In *Computational Fluid Dynamics*, number 1993–04 in Lecture Series. von Karman Institute for Fluid Dynamics, March 1993.

[31] R. Struijs. The fluctuation splitting method. In Cornelis B. Vreugdenhil and Barry Koren, editors, *Numerical Methods for Advection-Diffusion Problems*, volume 45 of *Notes on numerical fluid mechanics*, chapter 11, pages 261–289. Vieweg, Germany, 1993.

[32] Philip L. Roe. Fluctuation splitting schemes on optimal grids. AIAA Paper 97–2034, June 1997.

[33] R. Struijs, H. Deconinck, and P. L. Roe. Fluctuation spliting schemes for the 2D Euler equations. In *Computational Fluid Dynamics*, Lecture Series 1991–01. von Karman Institute for Fluid Dynamics, Belgium, February 1991.

[34] R. Struijs, H. Deconinck, P. de Palma, P. Roe, and K. G. Powell. Progress on multidimensional upwind Euler solvers for unstructured grids. AIAA Paper 91–1550, June 1991.

[35] Michael Rudgyard. Cell vertex methods for steady inviscid flow. In *Computational Fluid Dynamics*, number 1993–04 in Lecture Series. von Karman Institute for Fluid Dynamics, Belgium, March 1993.

[36] H. Deconinck, P. L. Roe, and R. Struijs. A multidimensional generalization of Roe's flux difference splitter for the Euler equations. *Computers and Fluids*, 22(2/3):215–222, 1993.

[37] H. Deconinck, H. Paillère, R. Struijs, and P. L. Roe. Multidimensional upwind schemes based on fluctuation-splitting for systems of conservation laws. *Computational Mechanics*, 11:323–340, 1993.

[38] H. Paillère, H. Deconinck, R. Struijs, P. L. Roe, L. M. Mesaros, and J.-D. Müller. Computations of inviscid compressible flows using fluctuation-splitting on triangular meshes. AIAA Paper 93–3301, July 1993.

[39] L. A. Catalano, P. De Palma, G. Pascazio, and M. Napolitano. A higher-order multidimensional upwind solution-adaptive multigrid solver for compressible flows. In *Lecture Notes in Physics*, number 453, pages 241–245, 1995.

[40] Lisa Marie Mesaros. *Multi-Dimensional Fluctuation Splitting Schemes for the Euler Equations on Unstructured Grids.* PhD thesis, University of Michigan, USA, 1995.

[41] L. A. Catalano, P. de Palma, G. Pascazio, and M. Napolitano. A contribution to multidimensional upwinding and fluctuation splitting: Nonlinear matrix schemes. AIAA Paper 97–2031, June 1997.

[42] M. E. Hubbard and M. J. Baines. Conservative multidimensional upwinding for the steady two-dimensional shallow water equations. *Journal of Computational Physics*, 138:419–448, November 1997.

[43] W. M. Eppard and B. Grossman. A multi-dimensional kinetic-based upwind solver for the Euler equations. AIAA 93–3303, July 1993.

[44] Benoît Perthame and Youchun Qiu. A variant of van Leer's method for multidimensional systems of conservation laws. *Journal of Computational Physics*, 112:370–381, 1994.

[45] B. Perthame, Y. Qiu, and B. Stoufflet. Kinetic discretization of gas dynamics using fluctuation-splitting schemes. Rapport interne 94–11, Centre National de la Recherche Scientifique, Toulouse, 1994.

[46] Sergei V. Pevchin. *New Efficient Contact Discontinuity Capturing Techniques in Supersonic Flow Simulations*. PhD thesis, Virginia Tech, Blacksburg, Virginia, September 1996.

[47] B. Grossman, S. V. Pevchin, and W. M. Eppard. Multi-dimensional, upwind, Euler algorithms from kinetic theory. In D. A. Caughey and M. M. Hafez, editors, *Frontiers of Computational Fluid Dynamics 1994*, chapter 11, pages 169–188. John Wiley & Sons, New York, 1994.

[48] H. Paillère, J. Boxho, G. Degrez, and H. Deconinck. Multidimensional upwind residual distribution schemes for the convection-diffusion equation. *International Journal for Numerical Methods in Fluids*, 23:923–936, 1996.

[49] H. Paillère, E. Van der Weide, and H. Deconinck. Multidimensional upwind methods for inviscid and viscous compressible flows. In *Computational Fluid Dynamics*, number 1995–02 in Lecture Series. von Karman Institute, Belgium, March 1995.

[50] George Timothy Tomaich. *A Genuinely Multi-Dimensional Upwinding Algorithm for the Navier-Stokes Equations on Unstructured Grids Using a Compact, Highly-Parallelizable Spatial Discretization*. PhD thesis, University of Michigan, USA, 1995.

[51] J.-C. Carette and H. Deconinck. Adaptive hybrid remeshing and SUPG/MultiD-upwind solver for compressible high Reynolds number flows. AIAA Paper 97–1857, May 1997.

[52] S. Mitran, D. Caraeni, and D. Livescu. Large eddy simulation of unsteady rotor-stator interaction in a centrifugal compressor. AIAA Paper 97–3006, July 1997.

[53] David Sidilkover and Achi Brandt. Multigrid solution to steady-state two-dimensional conservation laws. *SIAM Journal of Numerical Analysis*, 30(1):249–274, February 1993.

[54] Thomas W. Roberts, David Sidilkover, and R. C. Swanson. Textbook multigrid efficiency for the steady Euler equations. AIAA Paper 97–1949, May 1997.

[55] David Sidilkover. Some approaches towards constructing optimally efficient multigrid solvers for the inviscid flow equations. Report 97–39, ICASE, Hampton, August 1997.

[56] S. Spekreijse. Multigrid solution of monotone second-order discretization of hyperbolic conservation laws. *Mathematics of Computation*, 49:135–155, 1987.

[57] P. A. Gnoffo. A finite-volume, adaptive grid algorithm applied to planetary entry flowfields. *AIAA Journal*, 21(9):1249–1254, September 1983.

[58] A. D. Harvey, S. Acharya, S. L. Lawrence, and S. Cheung. A solution adaptive grid procedure for an upwind parabolized flow solver. AIAA Paper 90–1567, June 1990.

[59] A. D. Harvey, S. Acharya, and S. L. Lawrence. A solution-adaptive grid procedure for the three-dimensional parabolized Navier-Stokes equations. AIAA Paper 91–0104, January 1991.

[60] A. D. Harvey, S. Acharya, and S. L. Lawrence. Prediction of complex three-dimensional flowfields using a solution-adaptive mesh algorithm. AIAA Paper 91-3237, September 1991.

[61] Jean-François Hétu and Dominique H. Pelletier. Adaptive remeshing for viscous incompressible flows. *AIAA Journal*, 30(8):1986–1992, August 1992.

[62] Jean-François Hétu and Dominique H. Pelletier. Fast, adaptive finite element scheme for viscous incompressible flows. *AIAA Journal*, 30(11):2677–2682, November 1992.

[63] V. Parthasarathy and Y. Kallinderis. Adaptive prismatic-tetrahedral grid refinement and redistribution for viscous flows. *AIAA Journal*, 34(4):707–716, April 1996.

[64] David L. Marcum. Adaptive unstructured grid generation for viscous flow applications. *AIAA Journal*, 34(11):2440–2443, November 1996.

[65] F. Ilinca, D. Pelletier, and L. Ignat. Adaptive finite element solution of compressible turbulent flows. AIAA Paper 98–0229, January 1998.

[66] F. Ilinca, D. Pelletier, and L. Ignat. Adaptive finite element solution of compressible turbulent flows. *AIAA Journal*, 36(12):2187–2194, December 1998.

[67] É. Turgeon, D. Pelletier, and L. Ignat. Effects of adaptivity on various finite element schemes for turbulent heat transfer and flow predictions. AIAA Paper 98–0853, January 1998.

[68] O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and *a posteriori* error estimates. Part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, 33:1331–1364, 1992.

[69] O. C. Zienkiewicz and J. Z. Zhu. The superconvergent patch recovery and *a posteriori* error estimates. Part 2: Error estimates and adaptivity. *International Journal for Numerical Methods in Engineering*, 33:1365–1382, 1992.

[70] D. Ait-Ali-Yahia, W. G. Habashi, and A. Tam. A directionally adaptive methodology using and edge-based error estimate on quadrilateral grids. *International Journal for Numerical Methods in Fluids*, 23:673–690, 1996.

[71] J. Dompierre, M.-G. Vallet, M. Fortin, Y. Bourgault, and W. G. Habashi. Anisotropic mesh adaption: Towards a solver and user independent CFD. AIAA Paper 97–0861, January 1997.

[72] F. Taghaddosi, W. G. Habashi, G. Guèvremont, and D. Ait-Ali-Yahia. An adaptive least-squares method for the compressible Euler equations. AIAA Paper 97–2097, June 1997.

[73] Djaffar Ait-Ali-Yahia and Wadgi G. Habashi. Finite element adaptive method for hypersonic thermochemical nonequilibrium flows. *AIAA Journal*, 35(8):1294–1302, August 1997.

[74] W. G. Habashi, M. Fortin, J. Dompierre, M.-G. Vallet, and Y. Bourgault. Anisotropic mesh adaption: A step towards a mesh-independent and user-independent CFD. In V. Venkatakrishnan et al, editor, *Barriers and Challenges in Computational Fluid Dynamics*, pages 99–117. Kluwer Academic Publisher, 1998.

[75] A. Tam, M. P. Robichaud, P. Tremblay, W. G. Habashi, M. Hohmeyer, G. Guèvremont, M. G. Peeters, and P. Germain. A 3-D adaptive anisotropic method for external and internal flows. AIAA Paper 98–0771, January 1998.

[76] W. G. Habashi, J. Dompierre, Y. Bourgault, M. Fortin, and M.-G. Vallet. Certifiable computational fluid dynamics through mesh optimization. *AIAA Journal*, 36(5):703–711, May 1998.

[77] Philip Roe. Compounded of many simples. In V. Venkatakrishnan et al, editor, *Barriers and Challenges in Computational Fluid Dynamics*, pages 241–258. Kluwer Academic Publishers, 1998.

[78] J.-Y. Trépanier, M. Paraschivoiu, M. Reggio, and R. Camarero. A conservative shock fitting method on unstructured grids. *Journal of Computational Physics*, 126(2):421–433, July 1996.

[79] Paresh Parikh, Sudheer N. Nayani, and Ijaz Parpia. Multidimensional wave models for solution-adaptive grid generation. SBIR Final Report Contract No. NAS2–13797, NASA, August 1993.

[80] P. L. Roe. Characteristic-based schemes for the Euler equations. *Annual Review of Fluid Mechanics*, 18:337–365, 1986.

[81] R. Courant, E. Isaacson, and M. Reeves. On the solution of nonlinear hyperbolic differential equations by finite differences. *Pure and Applied Mathematics*, 5:243–255, 1952.

[82] A. Harten and J. M. Hyman. Self adjusting grid methods for one-dimensional hyperbolic conservation laws. *Journal of Computational Physics*, 50:235–269, 1983.

[83] W. Kyle Anderson and Daryl L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23(1):1–21, January 1994.

[84] William B. Bickford. *A First Course in the Finite Element Method*. Richard D. Irwin, Inc., Boston, 1990.

[85] Klaus-Jürgen Bathe. *Finite Element Procedures in Engineering Analysis*. Prentice-Hall, Inc., Englewood Cliffs, USA, 1982.

[86] L. Euler. Principes généraux du mouvement des fluides. *Historical Academy of Berlin, Opera Omnia II*, 12:54–92, 1755.

[87] M. Navier. Mémoire sur les lois du mouvement des fluides. *Mémoire de l'Académie des Sciences*, 6:389, 1827.

[88] G. G. Stokes. On the theories of the internal friction of fluids in motion. *Trans. Cambridge Philosophical Society*, 8:227–319, 1849.

[89] Dale A. Anderson, John C. Tannehill, and Richard H. Pletcher. *Computational Fluid Mechanics and Heat Transfer*. Taylor and Francis, 1984.

[90] Christopher W. S. Bruner and Robert W. Walters. Parallelization of the Euler equations on unstructured grids. AIAA Paper 97–1894, June 1997.

[91] Shahyar Pirzadeh. Three-dimensional unstructured viscous grids by the advancing-layers method. *AIAA Journal*, 34(1):43–49, January 1996.

[92] S. Pirzadeh. Progress toward a user-oriented unstructured viscous grid generator. AIAA Paper 96–0031, January 1996.

[93] R. M. Smith and A. G. Hutton. The numerical treatment of advection: A performance comparison of current methods. *Numerical Heat Transfer*, 5:439–461, 1982.

[94] S. A. Warsi and L. A. Kania. Hybrid grid Navier-Stokes solver with h-refinement adaption. AIAA Paper 98–0545, January 1998.

[95] D. J. Mavriplis. Multigrid strategies for viscous flow solvers on anisotropic unstructured meshes. Report No. 98–6, ICASE, Hampton, January 1998.

[96] E. Turgeon, L. Ignat, and D. Pelletier. Computation of jet impingement heat transfer by adaptive finite element algorithm. AIAA Paper 98–2585, June 1998.

[97] H. Zhang, J. Y. Trépanier, M. Reggio, and R. Camarero. A Navier-Stokes solver for stretched triangular grids. AIAA Paper 92–0183, January 1992.

[98] W. Kyle Anderson and Daryl L. Bonhaus. Airfoil design on unstructured grids for turbulent flows. *AIAA Journal*, 37(2):185–191, February 1999.

[99] David A. Venditti and David L. Darmofal. A multilevel error estimation and grid adaptive strategy for improving the accuracy of integral outputs. AIAA Paper 99–3292, June 1999.

[100] H. Paillère, H. Deconinck, and E. van der Weide. Upwind residual distribution methods for compressible flow: An alternative to finite volume and finite element methods. Part I: Scalar schemes. In *Computational Fluid Dynamics*, number 1997–02 in Lecture Series. von Karman Institute, Belgium, March 1997.

[101] Ashok K. Shinghal. Key elements of verification and validation of CFD software. AIAA Paper 98–2639, June 1998.

[102] Patrick J. Roache. Verification of codes and calculations. *AIAA Journal*, 36(5):696–702, May 1998.

[103] D. Caraeni, M. Caraeni, and L. Fuchs. A parallel multidimensional upwind algorithm for LES. AIAA 2001-2547, June 2001.

[104] Rajendran Mohanraj, Yedidia Neumeier, and Ben T. Zinn. Characteristic-based treatment of source terms in Euler equations for Roe scheme. *AIAA Journal*, 37(4):417–424, April 1999.

[105] Philip L. Roe. Personal communication in Anaheim, California. University of Michigan, Ann Arbor, Michigan, June 2001.

[106] Andreas C. Haselbacher, James J. McGuirk, and Gary J. Page. Finite-volume discretization aspects for viscous flows on mixed unstructured grids. *AIAA Journal*, 37(2):177–184, February 1999.

[107] Andreas C. Haselbacher, James J. McGuirk, and Gary J. Page. Finite-volume discretisation aspects for viscous flows on mixed unstructured grids. AIAA Paper 97–1946, June 1997.

[108] R. Courant, K. O. Friedrichs, and H. Lewy. Über die partiellen differenzengleichurgen der mathematischen physik. *Mathematische Annalen*, 100:32–74, 1928. (see also Ref. [132]).

[109] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford, 1992.

[110] J. W. Maccoll. The conical shock wave formed by a cone moving at high speed. *Proceedings of the Royal Society for Aeronautics*, 159, 1937.

[111] H. Blasius. The boundary layers in fluids with little friction. NACA TM 1256, February 1950.

[112] William L. Kleb, William A. Wood, and Bram van Leer. Efficient multi-stage time marching for viscous flows via local preconditioning. AIAA Paper 99–3267, June 1999.

[113] F. McNeil Cheatwood and Peter A. Gnoffo. User's manual for the Langley aerothermodynamic upwind relaxation algorithm (LAURA). NASA TM 4674, April 1996.

[114] V. Venkatakrishnan. Convergence to steady state solutions of the Euler equations on unstructured grids with limiters. *Journal of Computational Physics*, 118:120–130, 1995.

[115] Brian R. Hollis. Experimental and computational aerothermodynamics of a Mars entry vehicle. NASA CR 201633, December 1996.

[116] E. R. Beeman and S. A. Powers. A method for determining the complete flow field around conical wings at supersonic/hypersonic speeds. AIAA Paper 69–646, June 1969.

[117] B. R. Hollis and J. N. Perkins. High-enthalpy aerothermodynamics of a mars entry vehicle part 1: Experimental results. *Journal of Spacecraft and Rockets*, 34(4):449–456, July 1997.

[118] B. R. Hollis and J. N. Perkins. High-enthalpy aerothermodynamics of a mars entry vehicle part 2: Computational results. *Journal of Spacecraft and Rockets*, 34(4):457–463, July 1997.

[119] G. V. Candler and R. W. MacCormack. Hypersonic flow past 3-D configurations. AIAA Paper 87-0480, January 1987.

[120] K. M. Peery and S. T. Imlay. Blunt-body flow simulations. AIAA Paper 88-2904, July 1998.

[121] T. W. Roberts. The behaviour of flux difference splitting schemes near slowly moving shock waves. *Journal of Computational Physics*, 90:141–160, 1990.

[122] James J. Quirk. A Contribution to the Great Riemann Solver Debate. NASA CR 191409, November 1992.

[123] Peter A. Gnoffo. Personal communication. NASA Langley Research Center, Hampton, Virginia, August 2001.

[124] Nail K. Yamaleev and Mark H. Carpenter. On accuracy of adaptive grid methods for captured shocks. *submitted to Journal of Computational Physics*, 2001.

[125] Charles Hirsch. *Numerical Computation of Internal and External Flows—Volume 2: Computational Methods for Inviscid and Viscous Flows*. John Wiley & Sons Ltd., 1990.

[126] P. K. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM Journal of Numerical Analysis*, 21:995–1011, 1984.

[127] Philip L. Roe. Some contributions to the modelling of discontinuous flows. In B. E. Engquist, S. Osher, and R. C. J. Somerville, editors, *Large-Scale Computations in Fluid Mechanics*, volume 22 of *Lectures in Applied Mathematics*, pages 163–193. American Mathematical Society, Providence, RI, 1985.

[128] S. R. Chakravarthy and S. Osher. High resolution applications of the Osher upwind scheme for the Euler equations. AIAA Paper 83–1943, June 1983.

[129] G. D. van Albada, B. van Leer, and W. W. Roberts. A comparative study of computational methods in cosmic gas dynamics. Report 81–24, ICASE, Hampton, August 1981.

[130] L. H. Back. Conservation equations of a viscous, heat-conducting fluid in curvilinear orthogonal coordinates. NASA TR 32–1332, September 1968.

[131] Sheng-Tao Yu. Convenient method to convert two-dimensional CFD codes into axisymmetric ones. *Journal of Propulsion and Power*, 9(3):493–495, May 1993.

[132] R. Courant, K. O. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal*, 5:215–234, March 1967. (also as AEC Research and Development Report NYO-7689).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
April 2002

3. REPORT TYPE AND DATES COVERED
Technical Publication

4. TITLE AND SUBTITLE
Multi-dimensional Upwind Fluctuation Splitting Scheme with Mesh Adaption for Hypersonic Viscous Flow

5. FUNDING NUMBERS
WU 706-85-41-01

6. AUTHOR(S)
William A. Wood

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681–2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L–18147

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA/TP–2002–211640

11. SUPPLEMENTARY NOTES
The information presented in this report was offered as a dissertation for the degree of Doctor of Philosophy, Virginia Polytechnic Institute and State University, November 9, 2001.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 64 Distribution: Standard
Availability: NASA CASI (301) 621–0390

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
A multi-dimensional upwind fluctuation splitting scheme is developed and implemented for two-dimensional and axisymmetric formulations of the Navier-Stokes equations on unstructured meshes. Key features of the scheme are the compact stencil, full upwinding, and non-linear discretization which allow for second-order accuracy with enforced positivity. Throughout, the fluctuation splitting scheme is compared to a current state-of-the-art finite volume approach, a second-order, dual mesh upwind flux difference splitting scheme (DMFDSFV), and is shown to produce more accurate results using fewer computer resources for a wide range of test cases. A Blasius flat plate viscous validation case reveals a more accurate v-velocity profile for fluctuation splitting, and the reduced artificial dissipation production is shown relative to DMFDSFV. Remarkably the fluctuation splitting scheme shows grid converged skin friction coefficients with only five points in the boundary layer for this case. The second half of the report develops a local, compact, anisotropic unstructured mesh adaption scheme in conjunction with the multi-dimensional upwind solver, exhibiting a characteristic alignment behavior for scalar problems. The adaption strategy is extended to the two-dimensional and axisymmetric Navier-Stokes equations of motion through the concept of fluctuation minimization.

14. SUBJECT TERMS
CFD, Fluctuation Splitting, Mesh Adaption

15. NUMBER OF PAGES
413

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL