Course Notes for MAE 6440 - Advanced Computational Fluid Dynamics

Dr. Aaron Katz
Mechanical and Aerospace Engineering
Utah State University, Logan, UT 84322

Introduction

The book listed for this course is Versteeg and Malalasekera [7]. This book was used for the undergraduate CFD course at USU, and it is listed for this course to avoid the need to purchase another book. However, most of the material for this course comes from other sources, such as these course notes.

The material for this course is based on the idea that students of CFD should understand algorithms at a fundamental level, instead of just running commercial software. The focus on algorithms provides at least three benefits to students: (1) Some students will go on to become CFD code developers, (2) Understanding many of the algorithms underlying commercial CFD codes will make students better users of those codes, and (3) Performing CFD code development can give students an appreciation for the intricacy of CFD codes and a healty dose of skepticism for the results generated by these codes. Furthermore, commercial codes today are quite user friendly with ample tutorials and resources for learning their usage. This course however, is designed to give students an idea of what underlies these codes.

These notes are organized as follows. First, the basic equations of fluid dynamics (laminar Navier-Stokes) are reviewed. Next, a method to discretize these equations in space on triangles is developed. Solution methods are then reviewed to provide strategies to solve the discretized equations. Sample CFD solutions are then given. Finally, two appendices contain information regarding linux usage and practical skills to run the code, as well as an outline of the code structure for `TriSolve2d`, which is the 2D triangle code we will be working with in this course.

1 Equations of Fluid Dynamics

In order to understand the algorithms used to solve the fluid equations, we must first understand the equations themselves. The equations of motion for fluids are in the form of "conservation laws," in which certain quantities are conserved. Specifically, we conserve the following quantities:

1. mass
2. momentum
3. energy

Mass conservation states that matter cannot be created or destroyed. Momentum conservation states that the time rate of change of momentum must equal the summation of external forces (Newton's second law). Energy conservation states that energy cannot be created or destroyed, but may be added to a system through heat and work (1st law of thermodynamics). All of CFD, and everything we will do in this class is built on these simple principles. It is important not to lose sight of these fundamental principles when we begin talking about algorithms.

1.1 Conservation of Mass, Momentum, and Energy

A detailed derivation of the conservation laws for mass, momentum, and energy can be found in many works, such as the classical text of Kundu and Cohen [4]. Here, I will present a short derivation and the important results. For mass conservation, consider a fixed volume in space, V. The amount of mass may change within this volume, and a measure of that change in mass is

$$\frac{d}{dt}\int_V \rho dV = \int_V \frac{\partial \rho}{\partial t} dV, \tag{1}$$

where ρ is the density, and we can bring the derivative inside the integral because the control volume is fixed. Additionally, the mass flow out of the control volume can be represented by

$$\int_A \rho u_j n_j dA, \tag{2}$$

where A is the area enclosing V, u_j is the j^{th} component of the fluid velocity, n_j is the j^{th} component of the outward normal, and summation notation is used.

To conserve mass, the rate of increase in mass in the volume must equal the amount of mass flowing into the volume:

$$\int_V \frac{\partial \rho}{\partial t} dV = -\int_A \rho u_j n_j dA. \tag{3}$$

We can use the divergence theorem to convert the surface integral to a volume integral:

$$\int_V \left[\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_j)}{\partial x_j}\right] dV = 0. \tag{4}$$

Since this expression must hold for any control volume, the integrand must vanish, which leads us to the differential form for mass conservation:

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_j)}{\partial x_j} = 0. \tag{5}$$

This equation is in "strong conservation law" (SCL) form since there are no quantities in front of derivatives. SCL form is necessary to obtain correct shock stregths and speeds for transonic and supersonic flow. There is evidence that SCL is more appropriate for subsonic, nearly incompressible flows as well[5].

I could do a similar derivation for momentum and energy, which would result in the following equations in strong conservation law form:

$$\frac{\partial}{\partial t}(\rho u_i) + \frac{\partial}{\partial x_j}(\rho u_i u_j + p\delta_{ij}) - \frac{\partial}{\partial x_j}(\sigma_{ij}) - \rho g_i = 0 \tag{6}$$

$$\frac{\partial}{\partial t}(\rho e) + \frac{\partial}{\partial x_j}(\rho h u_j) - \frac{\partial}{\partial x_j}(\sigma_{ij} u_i) + \frac{\partial q_j}{\partial x_j} - \rho g_j u_j = 0. \tag{7}$$

Here, p is the pressure, g_i is the i^{th} component of the body force, e is the total energy (internal plus kinetic) per unit mass, $h = e + p/\rho$ is the total enthalpy per unit mass, σ_{ij} is the (deviatoric) stress tensor, and q_j is the j^{th} component of the heat flux vector. Certain CFD algorithms will also require entropy. An entropy-like variable that is suitable for our purposes is $s = p/(\rho^\gamma)$

Up to this point, the only assumptions we have made are that the fluid is a continuum, in which we neglect the true molecular nature of matter. This will be a very good assumption in this class. We will make two other assumptions about the equations of motion for this class. First, we will neglect body forces, such as gravity. This is a good approximation for fluids with low density, such as air. Second, we will assume that the fluid is "Newtonian," which means that the rate of strain of the fluid is linearly proportional to the stress. The classic Newtonian stress tensor has the form

$$\sigma_{ij} = 2\mu e_{ij} - \frac{2}{3}\mu \frac{\partial u_k}{\partial x_k}, \tag{8}$$

where μ is the dynamic viscosity, and e_{ij} is the rate of strain tensor, defined as

$$e_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right). \tag{9}$$

In order to "close" the system of equations, that is, have the same number of equations as unknowns, we must relate p, μ, and q to the other flow quantities. We could assume μ is constant, but experiments show a weak dependence of μ on temperature. This dependence is often accounted for by using Sutherland's Law,

$$\mu = \mu_0 \left(\frac{T}{T_0}\right)^{3/2} \frac{T_0 + S}{T + S}, \tag{10}$$

where μ_0 is the viscosity at a reference temperature T_0, and S is the Sutherland temperature. Quantities for air are taken as

$$\mu_0 = 1.716\text{e-}5 \text{ kg/ms}, \quad T_0 = 273.15 \text{ K}, \quad S = 110.4 \text{ K}. \tag{11}$$

Additionally, we need to relate pressure and temperature to the other flow quantities. Typically, we use the ideal gas assumption,

$$p = \rho RT = (\gamma - 1)\left(\rho e - \frac{1}{2}\rho u_k u_k\right), \tag{12}$$

where R is the specific gas constant, which for air is $R = 287.04$ J/(kg K), and $\gamma = C_p/C_v$ is the ratio of specific heats, which for air is 1.4. Finally, we relate the heat flux vector to the temperature gradient through the Fourier law of heat conduction,

$$q_j = -\kappa \frac{\partial T}{\partial x_j}, \tag{13}$$

where κ is the coefficient of thermal conductivity, defined as

$$\kappa = \frac{C_p \mu}{Pr} = \frac{\gamma R \mu}{(\gamma - 1)Pr}, \tag{14}$$

where Pr is the Prandtl number, which for air is around 0.75.

The description of fluid behavior in terms of conservation of mass, momentum, and energy is now complete. When these equations are solved directly, we call the approach "direct numerical simulation" (DNS). In practice, we find that many problems require excessive grid resolution to resolve all scales of turbulence. In those cases, we desire to use a turbulence model along with the "averaged" flow equations. The turbulence modeling approach allows us to solve for the mean flow, while accounting for the effects of turbulence on the mean flow at reasonable cost. Unfortunately, there does not today exist a single turbulence model which performs well for wide classes of problems. While turbulence is an important topic, we will focus on DNS, which is appropriate for inviscid and laminar viscous flows.

1.2 The System Point of View

When dealing with the Navier-Stokes equations, it is convenient to consider them as a system of equations of the form

$$\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial x_j} = \frac{\partial F_j^v}{\partial x_j}, \tag{15}$$

where the vectors Q, F_j, and F_j^v are defined as

$$Q = \begin{pmatrix} \rho \\ \rho u_i \\ \rho e \end{pmatrix}, \quad F_j = \begin{pmatrix} \rho u_j \\ \rho u_i u_j + p\delta_{ij} \\ \rho h u_j \end{pmatrix}, \quad F_j^v = \begin{pmatrix} 0 \\ \sigma_{ij} \\ \sigma_{ij} u_i - q_j \end{pmatrix}, \tag{16}$$

Here, Q is the vector of conserved variables. These are the dependent variables that we solve for. Also, F_j is known as the inviscid flux vector, and F^v is known as the viscous flux vector.

if we set $F^v = 0$, we obtain the Euler equations, which is a model of "inviscid" flow. Much time in this class will be devoted to understanding inviscid flow, since the inviscid terms require special treatment for stability and accuracy. The viscous terms generally remain well-behaved will little effort on our part.

Using a simple chain rule, we can write the Euler equations in "quasi-linear" form:

$$\frac{\partial Q}{\partial t} + A_k \frac{\partial Q}{\partial x_k} = 0, \tag{17}$$

where A is the flux Jacobian. A is a matrix, whose ij element is defined as

$$A_{ij} = \frac{\partial F_i}{\partial Q_j}. \tag{18}$$

The procedure for computing the flux Jacobian is to express F completely in terms of Q, then differentiate accordingly. The eigenvalues of A represent convective and acoustic wave speeds. Since these eigenvalues are purely real numbers, we say that the Euler equations for a "hyperbolic" of PDE's.

To understand the wave nature of the Euler equations, consider the equations in 1D:

$$\frac{\partial Q}{\partial t} + A \frac{\partial Q}{\partial x} = 0, \tag{19}$$

where $Q = (\rho, \rho u, \rho e)$ is a three component vector, and A is the 3×3 Jacobian matrix. We can diagonalize A with its matrix X of right eigenvectors, such that

$$\Lambda = X^{-1} A X, \tag{20}$$

where Λ is a diagonal matrix containing the eigenvalues of A. If we premultiply our PDE by X^{-1}, we can decouple the system of PDE's into three scalar equations:

$$X^{-1}\frac{\partial Q}{\partial t} + X^{-1} A X X^{-1} \frac{\partial Q}{\partial x} = 0, \tag{21}$$

$$\frac{\partial W}{\partial t} + \Lambda \frac{\partial W}{\partial x} = 0, \tag{22}$$

where we define W such that $\partial W / \partial Q = X^{-1}$, which are known as the "characteristic variables." Since Λ is diagonal, we now have a set of three scalar equations, the i^{th} of which is

$$\frac{\partial w_i}{\partial t} + \lambda_i \frac{\partial w_i}{\partial x} = 0, \tag{23}$$

The solution to Equation 24 is simply

$$w_i(x,t) = w_{i,0}(x - \lambda_i t), \tag{24}$$

where $w_{i,0} = w_i(x,0)$ is the initial condition. In other words, the initial conditions are convected downstream, unchanged, at speed λ_i. Therefore, the originial system in Equation 19 represents a superposition of three wave equations with waves traveling at speeds λ_i. The eigenvalues, λ_i, of are u, $u + c$, and $u - c$, which represent the convective speed, and two acoustic speeds of sound (pressure) wave propagation. Here, $c = \sqrt{\gamma p / \rho}$ is the speed of sound. Our numerical methods need to account for these different wave speeds, which may be positive or negative depending if the flow regime is subsonic, supersonic, or a mix of both (transonic).

Exercises

1. Find the eigenvalues and eigenvectors of the Euler flux Jacobian in one dimension. What are the directions and magnitudes of the various wave components?

2 Unstructured Grid Spatial Discretization

While there are many ways to discretize the equations for conservation of mass, momentum, and energy on unstructured grids, in this class we will base our method on a finite element Galerkin approach. This approach has several advantages over other approaches, such as the finite volume approach, in terms of economy, boundary condition implementation, and storage. As it turns out, after we go through the derivation for this finite element approach, the final result actually looks like a finite volume scheme anyway! But it is important to remember that the roots of this method are actually in the finite element approach. This helps us formulate boundary conditions, source terms, and other aspects correctly to maintain second order accuracy.

2.1 Galerkin Finite Element Approach on Linear Triangles

Before delving into the finite element method, simply consider the task of estimating the gradient of a function f given its value at the three vertices of a triangle. We can use Gauss' Theorem, which states that

$$\int_V \boldsymbol{\nabla} f dV = \int_A f \boldsymbol{n} dA, \tag{25}$$

where $\boldsymbol{n}$ is the outward normal of the area A which bounds some volume in space V. Now, if f is linear, then $\boldsymbol{\nabla} f$ is constant, and we can say

$$\boldsymbol{\nabla} f = \frac{1}{V} \int_A f \boldsymbol{n} dA, \tag{26}$$

For a triangle, we can break the area integral into 3 integrals over each side, using the midpoint rule, which is exact for a linear function:

$$\boldsymbol{\nabla} f = \frac{1}{V} \sum_i \frac{1}{2} (f_i + f_{i+1}) \boldsymbol{n}_i dA_i. \tag{27}$$

Considering the geometry, we note that

$$n_{x,i} dA_i = y_{i+1} - y_i, \tag{28}$$

$$n_{y,i} dA_i = -(x_{i+1} - x_i), \tag{29}$$

where the index i increases counterclockwise around the perimeter of the triangle. We can regroup the summation into contributions from the nodes to obtain

$$\frac{df}{dx} = \frac{1}{2V} \sum_i f_i \Delta \tilde{y}_i, \tag{30}$$

$$\frac{df}{dy} = -\frac{1}{2V} \sum_i f_i \Delta \tilde{x}_i, \tag{31}$$

where $\Delta \tilde{x}_i$ and $\Delta \tilde{y}_i$ are taken in the clockwise sense, opposite node i, as shown in Figure 1.

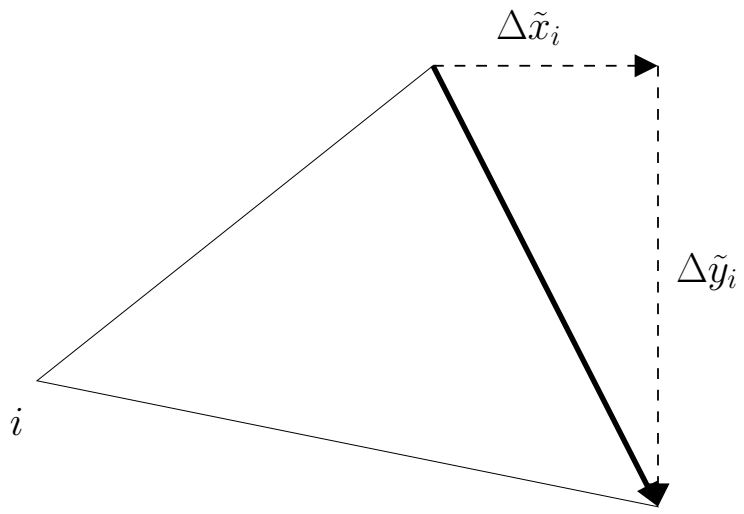

Figure 1: Meaning of $\Delta\tilde{x}_i$ and $\Delta\tilde{y}_i$ used to compute the gradient of a function within a triangle.

2.1.1 Galerkin Treatment of Inviscid Terms

With that in mind, consider the conservation laws expressed in the form of Equation 15. Lets drop the indicial notation and restrict ourselves to 2d inviscid flow, and consider the following form:

$$\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = 0, \tag{32}$$

where

$$Q = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho e \end{pmatrix}, \quad F = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uh \end{pmatrix}, \quad G = \begin{pmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vh \end{pmatrix}. \tag{33}$$

Finite element methods begin by multiplying the governing equation, in this case Equation 58, by an integrable test function $\phi = \phi(x, y)$ and integrating over space:

$$\int_V \phi \left(\frac{\partial Q}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = 0 \right). \tag{34}$$

Integrating by parts, we obtain

$$\int_V \phi \frac{\partial Q}{\partial t} dV - \int_V \left(F \frac{\partial \phi}{\partial x} + G \frac{\partial \phi}{\partial y} \right) dV + \int_\Gamma \phi \left(F n_x + G n_y \right) d\Gamma = 0. \tag{35}$$

(Recall integration by parts in 1d is just

$$\int_1^2 u dv = uv|_1^2 - \int_1^2 v du. \tag{36}$$

We can generalize this to multidimensions with

$$\int_V \frac{\partial f}{\partial x_i} g dV = -\int_V f \frac{\partial g}{\partial x_i} dV + \int_\Gamma f g n_i d\Gamma, \tag{37}$$

where f and g are arbitrary functions, V is the volume, and Γ is the boundary of the volume.)

Different schemes can be obtained by choosing various forms for ϕ. The Galerkin scheme, by definition, means that we choose ϕ to be of the same form that we represent our dependent variables, in this case, the fluxes F and G. For the purposes of this class, we will deal with the set of linear functions to represent F and G, which lead to second order accurate schemes.

To derive the discretization at a node 0, surrounded by nodes in an unstructured mesh, numbered counterclockwise for the sake of notational simplicity, consider ϕ as the linear "tent" function, with a value of $\phi = 1$ at node 0, and $\phi = 0$ everywhere else. The mesh for this configuration is shown in Figure 2. With ϕ defined in this way, it is straightforward to

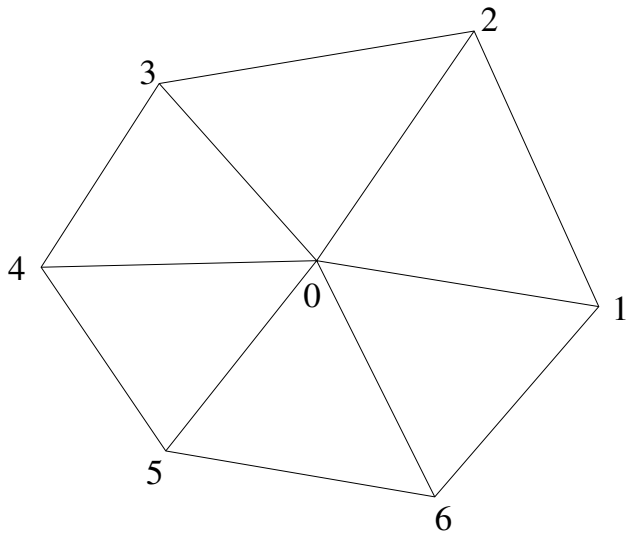

Figure 2: Unstructured stencil needed for the discretization at node 0.

compute $\boldsymbol{\nabla}\phi$ using our triangle formula above. For example in the triangle formed by nodes 0, 1, and 2 (denoted here as Δ_{012}), the gradient of ϕ is

$$\left.\frac{\partial \phi}{\partial x}\right|_{\Delta_{012}} = -\frac{y_2 - y_1}{2V_{012}}, \tag{38}$$

$$\left.\frac{\partial \phi}{\partial y}\right|_{\Delta_{012}} = \frac{x_2 - x_1}{2V_{012}}, \tag{39}$$

Also, recall that the integral of a linear function, f, over a triangle is just the average value of f multiplied by the volume (midpoint rule):

$$\int_{\Delta_{012}} f dV = \frac{1}{3}\left(f_0 + f_1 + f_2\right) V_{012}. \tag{40}$$

Now that we know how to evaluate $\boldsymbol{\nabla}\phi$, which is constant, as well as the integral of F and G over a triangle for linear function, we can evaluate the second integral in 35:

$$- \int_V \left(F\frac{\partial \phi}{\partial x} + G\frac{\partial \phi}{\partial y} \right) dV = -\sum_{\Delta} \int_{\Delta} \left(F\frac{\partial \phi}{\partial x} + G\frac{\partial \phi}{\partial y} \right) dV \tag{41}$$

$$= \sum_{\Delta} \left[\frac{F_0 + F_i + F_{i+1}}{3} \frac{y_{i+1} - y_i}{2V_\Delta} - \frac{G_0 + G_i + G_{i+1}}{3} \frac{x_{i+1} - x_i}{2V_\Delta} \right] V_\Delta \tag{42}$$

$$= \frac{1}{6} \sum_i \left[(F_{i+1} + F_i)(y_{i+1} - y_i) - (G_{i+1} + G_i)(x_{i+1} - x_i) \right]. \tag{43}$$

Note that the contributions from node 0 cancel since

$$\sum_i (y_{i+1} - y_i) = \sum_i (x_{i+1} - x_i) = 0 \tag{44}$$

for a closed boundary.

We need to work on this integral a bit more to get it into a form that is easier to work with. Examining the last line in Equation 41, we can regroup this summation from edges to contributions to nodes:

$$- \int_V \left(F\frac{\partial \phi}{\partial x} + G\frac{\partial \phi}{\partial y} \right) dV = \frac{1}{6} \sum_i \left[F_i(y_{i+1} - y_{i-1}) - G_i(x_{i+1} - x_{i-1}) \right]. \tag{45}$$

Once again, in light of Equation 44, we can add contributions from F_0 and G_0 back in to arrive at

$$- \int_V \left(F\frac{\partial \phi}{\partial x} + G\frac{\partial \phi}{\partial y} \right) dV = \sum_i \left[\frac{F_0 + F_i}{2} \frac{y_{i+1} - y_{i-1}}{3} - \frac{G_0 + G_i}{2} \frac{x_{i+1} - x_{i-1}}{3} \right]. \tag{46}$$

This formula says that we take the average value of F and G between node 0 and each of its surrounding nodes i, multiplied by the proper mesh length in x or y. The mesh lengths have a geometric interpretation, as shown in Figure 3. The terms $(y_{i+1} - y_{i-1})/3$ and $(x_{i+1} - x_{i-1})/3$ represent the area-weighted outward normal of the two bold facets connecting the centroids of the cells to the midpoint of the edge separating the cells. the resulting volume around node 0 formed by connecting cell centroids to edge medians is known as the "median-dual" control volume. While this derivation is really a finite element method, we can begin to see a finite volume-like interpretation.

Denote the median-dual area-weighted normals as

$$A_{x,i} = \frac{y_{i+1} - y_{i-1}}{3}, \tag{47}$$

$$A_{y,i} = -\frac{x_{i+1} - x_{i-1}}{3}, \tag{48}$$

and our formula becomes simply

$$- \int_V \left(F\frac{\partial \phi}{\partial x} + G\frac{\partial \phi}{\partial y} \right) dV = \sum_i \left(\frac{\mathcal{F}_0 + \mathcal{F}_i}{2} \right), \tag{49}$$

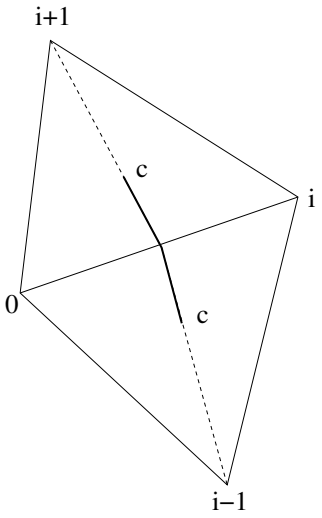

Figure 3: Median-dual control volume face areas.

where $\mathcal{F} = A_x F + A_y G$ is the "directed flux" through median-dual area A.

In addition to this term, we need to concern ourselves with the first and last integral terms in Equation 35. The first integral term represents the time derivative for unsteady physics. If we follow the Galerkin procedure rigorously, we will find that there are contributions from node 0 as well as each surrounding node i to the discretization at node 0. For simplicity, we often "lump" the contributions from the surrounding nodes to the central node 0, resulting in

$$\int_V \phi \frac{\partial Q}{\partial t} dV \approx \sum_{\Delta} \frac{V_\Delta}{3} \frac{dQ_0}{dt} = V_0 \frac{dQ_0}{dt}, \tag{50}$$

where V_0 is one third of the volume of all the cells surrounding node 0. This happens to be the volume enclosed by the median-dual face areas, which makes sense from the finite volume point of view.

The last integral in Equation 35 is an integral over the boundary of the domain. We can apply the same Galerkin method to evaluate this integral. If we do this for the boundary node 0 in Figure 4 for example, the combined contribution from the volume and surface integrals becomes

$$-\int_V \left(F\frac{\partial \phi}{\partial x} + G\frac{\partial \phi}{\partial y} \right) dV \quad + \quad \int_\Gamma \phi \left(F n_x + G n_y \right) d\Gamma = \tag{51}$$

$$\frac{1}{6}\big[(F_1 + F_0)(y_1 - y_0) \quad - \quad (G_1 + G_0)(x_1 - x_0) \tag{52}$$

$$+(F_2 + F_1)(y_2 - y_1) \quad - \quad (G_2 + G_1)(x_2 - x_1) \tag{53}$$

$$+(F_3 + F_2)(y_3 - y_2) \quad - \quad (G_3 + G_2)(x_3 - x_2) \tag{54}$$

$$+(F_4 + F_3)(y_4 - y_3) \quad - \quad (G_4 + G_3)(x_4 - x_3) \tag{55}$$

$$+(F_0 + F_4)(y_0 - y_4) \quad - \quad (G_0 + G_4)(x_0 - x_4)\big], \tag{56}$$

which is basically the same trapezoidal rule obtained at interior nodes. Recasting this into

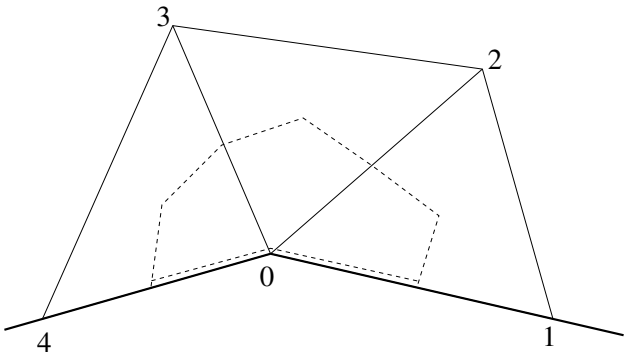

Figure 4: Boundary node stencil.

the form of Equation 49, we have

$$\begin{aligned} -\int_V \left(F\frac{\partial \phi}{\partial x} + G\frac{\partial \phi}{\partial y} \right) dV \quad &+ \quad \int_\Gamma \phi \left(F n_x + G n_y \right) d\Gamma = \\ \big[\frac{1}{6}(5\mathcal{F}_0 + \mathcal{F}_1)_b \quad &+ \quad \frac{1}{6}(5\mathcal{F}_0 + \mathcal{F}_4)_b + \\ \frac{1}{2}(\mathcal{F}_0 + \mathcal{F}_1) + \frac{1}{2}(\mathcal{F}_0 + \mathcal{F}_2) \quad &+ \quad \frac{1}{2}(\mathcal{F}_0 + \mathcal{F}_3) + \frac{1}{2}(\mathcal{F}_0 + \mathcal{F}_4)\big], \end{aligned}$$

where the subscript b represents the contributions from the median-dual edges coinciding with the boundary. These contributions have a $\frac{5}{6}$ on node 0 and a $\frac{1}{6}$ weight on the opposing node on the boundary edge.

In summary, the Galerkin approximation to the Euler equations at an interior node 0 can be written as

$$V_0 \frac{dQ_0}{dt} + \sum_i \frac{\mathcal{F}_0 + \mathcal{F}_i}{2} = 0, \tag{57}$$

where V_0 is one third of the volume of all triangles touching node 0 and $\mathcal{F} = A_x F + A_y G$ is the directed flux through area weighted normal A. For nodes lying on the boundary, different weighting should be used when adding in the contributions from the boundary edges.

2.1.2 Galerkin Treatment of Viscous Terms

While many flows can be modeled as "inviscid," all flows in nature exhibit viscous behavior to some degree. Many flows are dominated by viscosity in certain regions, such as boundary layers, wakes, and shear or mixing layers. In these cases, we cannot neglect the viscous terms in our discretization.

The viscous terms in two dimensions may be expressed as

$$\frac{\partial F^v}{\partial x} + \frac{\partial G^v}{\partial y} \tag{58}$$

where

$$F_v = \begin{pmatrix} 0 \\ \sigma_{xx} \\ \sigma_{xy} \\ u\sigma_{xx} + v\sigma_{xy} - q_x \end{pmatrix}, \quad G^v = \begin{pmatrix} 0 \\ \sigma_{yx} \\ \sigma_{yy} \\ u\sigma_{yx} + v\sigma_{yy} - q_y \end{pmatrix}. \tag{59}$$

The Newtonian stress components may be expressed as

$$\sigma_{xx} = 2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \tag{60}$$

$$\sigma_{yy} = 2\mu\frac{\partial v}{\partial y} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \tag{61}$$

$$\sigma_{xy} = \sigma_{yx} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right). \tag{62}$$

The heat flux terms are expressed as

$$q_x = -\kappa\frac{\partial T}{\partial x}, \quad q_y = -\kappa\frac{\partial T}{\partial y}. \tag{63}$$

The coefficient of heat conduction, κ, is given in Equation 14, and the dependence of viscosity on temperature may be captured with Sutherland's Law, shown in Equation 10.

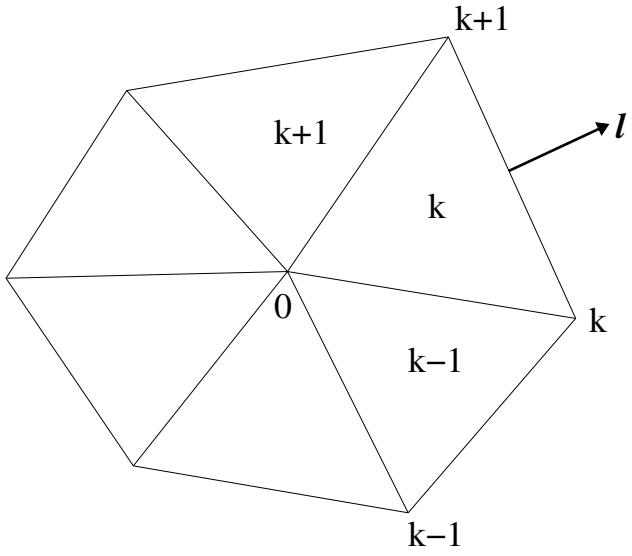

Figure 5: Unstructured stencil needed for the viscous discretization at node 0.

Examining the viscous terms, it is clear that these terms have the form

$$\frac{\partial}{\partial x_i}\left(\alpha\frac{\partial f}{\partial x_j}\right), \tag{64}$$

where α represents the viscosity or heat coefficient, and f represents velocity or temperature. Following the same procedure as for the inviscid terms, we can multiply our "viscous" terms by a test function, ϕ, and integrate over the domain. Considering ϕ to be the unit tent function at a node 0 in the domain, surrounded by nodes and triangles k, as shown in Figure 5, integration by parts yields

$$\int_{V_0} \phi \frac{\partial}{\partial x_i}\left(\alpha \frac{\partial f}{\partial x_j}\right) dV = -\int_{V_0} \frac{\partial \phi}{\partial x_i}\left(\alpha \frac{\partial f}{\partial x_j}\right) dV + \int_{\Omega_0} \phi \left(\alpha \frac{\partial f}{\partial x_j}\right) d\Omega. \tag{65}$$

Consider an interior node, for which the surface integral is not present. We may break up the volume integral over integrals on each triangle, k:

$$-\int_{V_0} \frac{\partial \phi}{\partial x_i}\left(\alpha \frac{\partial f}{\partial x_j}\right) dV = -\sum_k \int_{V_k} \frac{\partial \phi}{\partial x_i}\left(\alpha \frac{\partial f}{\partial x_j}\right) dV. \tag{66}$$

We know how to evaluate the gradient of ϕ, which was given in Equations 38-39. These gradients may be expressed as

$$\frac{\partial \phi}{\partial x_i} = -\frac{l_i}{2V_k}, \quad \boldsymbol{l} = \begin{pmatrix} l_x \\ l_y \end{pmatrix} = \begin{pmatrix} y_{k+1} - y_k \\ x_k - x_{k+1} \end{pmatrix}, \tag{67}$$

where $\boldsymbol{l}$ is the area-weighted normal pointing out of triangle k, shown in Figure 5. Similarly, the gradients of f in each triangle may be computed with Equations 30-31.

Assuming linear distributions, the gradients of ϕ and f are constant and may be taken out of the volume integral. This leaves the α term, which for a linear distribution, by be evaluated as

$$\int_{V_k} \alpha dV = \overline{\alpha}_k V_k, \quad \overline{\alpha}_k = \frac{1}{3}(\alpha_0 + \alpha_k + \alpha_{k+1}). \tag{68}$$

Finally, the Galerkin volume integral becomes

$$-\int_{V_0} \frac{\partial \phi}{\partial x_i}\left(\alpha \frac{\partial f}{\partial x_j}\right) dV = \sum_k \left(\overline{\alpha}_k \frac{l_i}{2} \frac{\partial f}{\partial x_j}\right). \tag{69}$$

The viscous terms may be assembled through a loop over triangles, where the contribution from each triangle is distributed to the three nodes on that triangle.

2.2 Local Extremum Diminishing (LED) Schemes

Unfortunately, application of the scheme in Equation 57 will result in immediate instability. To understand how to stabilize the Galerkin formulation, it is helpful to employ the idea of the local extremum diminishing (LED) property. This leads to a modification of Equation 57 in the form of "artificial dissipation," which can be shown to be equivalent to upwinding in many cases.

To illustrate the idea of LED schemes, consider a semi-discretization (discretized in space, but not time) of a scalar PDE governing some quantity, u, that results in the form

$$\frac{du_j}{d\tau} = \sum_i \alpha_i (u_i - u_j), \tag{70}$$

where the discrete values at nodes i surrounding node j are needed for the spatial discretization. Here, α_i are scalar values resulting from the spatial discretization. If we can get our semi-discrete scheme into this form, then a simple criteria for stability is that all α_i should be positive, a property known as "positivity." This leads to a stable scheme because it prevents local maxima from increasing, and local minima from decreasing. For example, if u_j is a local maximum, then $u_i - u_j$ will always be negative. If α_i is always positive, then $\frac{du_j}{dt}$ will be negative, meaning the value of u_j must decrease. The opposite would be true if u_j was a local minimum. Thus, the LED scheme cannot blow up and u will remain bounded at all nodes.

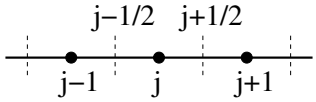

Figure 6: One-dimensional finite-volume stencil.

The 1D scalar wave equation will serve to illustrate the importance of the LED property in designing numerical schemes for PDE's. The 1D wave equation is

$$\frac{\partial u}{\partial \tau} + a\frac{\partial u}{\partial x} = 0, \tag{71}$$

where a is the constant wave speed (taken as positive here with no loss in generality). One may think of semi-discretizing the scheme with the central difference operator on the grid shown in Figure 6, resulting in

$$\frac{\partial u_j}{\partial \tau} + \frac{a}{2\Delta x_j}(u_{j+1} - u_{j-1}) = 0, \tag{72}$$

which may be equivalently expressed as

$$\frac{\partial u_j}{\partial \tau} = \frac{a}{2\Delta x_j}(u_{j-1} - u_j) - \frac{a}{2\Delta x_j}(u_{j+1} - u_j). \tag{73}$$

It is clear that the coefficient $-a/2\Delta x_j$ is negative, violating the LED criteria. In fact, this scheme is unstable. Physically, we know that the wave equation propagates information from left to right for positive a, so using u_{j+1} in the discrete scheme will incorporate information from the wrong direction. Central differencing in this context does not respect the physics, and the result is instability.

On the other hand, if we use a first order upwind semi-discretization,

$$\frac{\partial u_j}{\partial \tau} + \frac{a}{\Delta x_j}(u_j - u_{j-1}) = 0, \tag{74}$$

which may be equivalently expressed as

$$\frac{\partial u_j}{\partial \tau} = \frac{a}{2\Delta x_j}(u_{j-1} - u_j), \tag{75}$$

the LED criteria is satisfied, and a stable scheme results. With the upwind method, we are incorporating information into our discretization in a manner consistent with the physics.

We can generalize these ideas by using "numerical fluxes" containing artificial dissipation. Consider the finite volume discretization of a general 1d conservation law on the grid in Figure 6,

$$\frac{\partial u}{\partial \tau} + \frac{\partial f}{\partial x} = 0. \tag{76}$$

For the linear wave equation, $f = au$. A semi-discretization using numerical fluxes at the cell boundaries results in

$$\Delta x_j \frac{\partial u_j}{\partial \tau} + \hat{f}_{j+\frac{1}{2}} - \hat{f}_{j-\frac{1}{2}} = 0, \tag{77}$$

where the numerical flux is defined as

$$\hat{f}_{j+\frac{1}{2}} = \frac{1}{2}(f_j + f_{j+1}) - d_{j+\frac{1}{2}}, \tag{78}$$

where $d_{j+\frac{1}{2}}$ is the artificial diffusion. For the scalar wave equation, if we set the artificial diffusion to

$$d_{j+\frac{1}{2}} = \frac{|a|}{2}(u_{j+1} - u_j), \tag{79}$$

this will result in a numerical flux,

$$\hat{f}_{j+\frac{1}{2}} = \begin{cases} f_j & a > 0, \\ f_{j+1} & a < 0, \end{cases} \tag{80}$$

which produces an upwind scheme consistent with the direction of a. The addition of the proper amound of artificial diffusion results in an upwind scheme that respects the physics and is LED.

We can extend this idea to a system of non-linear conservation laws, such as the Euler equations in 1d,

$$\frac{\partial Q}{\partial \tau} + \frac{\partial F(Q)}{\partial x} = 0. \tag{81}$$

A stable scheme is constructed by defining a numerical flux of the form

$$\hat{F}_{j+\frac{1}{2}} = \frac{1}{2}(F_j + F_{j+1}) - d_{j+\frac{1}{2}}, \tag{82}$$

where the artificial diffusion is defined as

$$d_{j+\frac{1}{2}} = \frac{|A_{j+\frac{1}{2}}|}{2}(Q_{j+1} - Q_j), \tag{83}$$

and $A = \partial F / \partial Q$ is the flux Jacobian. The absolute value of the Jacobian is defined as

$$|A| = X|\Lambda|X^{-1}, \tag{84}$$

where the columns of X are the eigenvectors of A, and $\Lambda = \mathrm{diag}(u, u + c, u - c)$ contain the eigenvalues. So the absolute value acts only the the eigenvalues of A, which represent wave

speeds. The A with the subscript $j+\frac{1}{2}$ denotes that we must find A at the interface value between j and $j+1$. Let's generalize this to an interface Jacobian separating a left state (L) and a right state (R). It turns out that to be LED, we must define this interface Jacobian such that

$$F_R - F_L = A_{LR}(Q_R - Q_L). \tag{85}$$

Roe showed how to compute the interface Jacobian as the "standard" flux Jacobian, $A = \partial F/\partial Q$, but using "Roe-averaged" variables for the velocities and total enthalpy:

$$u = \frac{\sqrt{\rho_L}u_L + \sqrt{\rho_R}u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, \quad v = \frac{\sqrt{\rho_L}v_L + \sqrt{\rho_R}v_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}, \quad h = \frac{\sqrt{\rho_L}h_L + \sqrt{\rho_R}h_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}. \tag{86}$$

The form of artificial diffusion in Equation 83 is known as "characteristic" diffusion. This method works well for stabilization, but is quite expensive, since it involves eigenvalue decomposition and matrix multiplication. A cheaper scheme with comparable accuracy and stability properties, know as convective upwind split pressure (CUSP) has been developed by Jameson. This scheme works well for compressible flows involving subsonic, transonic, and supersonic regimes.

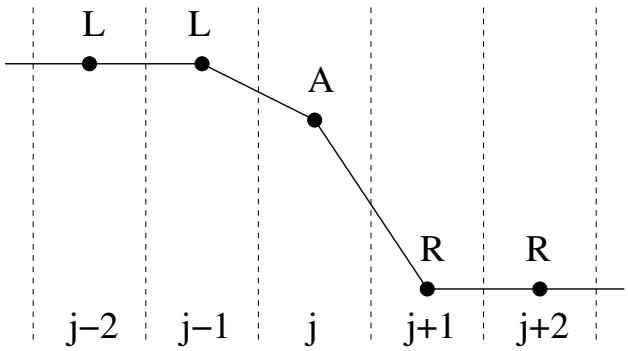

Figure 7: One-dimensional ideal shock between constant left and right values.

To derive this scheme, consider the ideal discrete shock, shown in Figure 7, which transitions between a constant state (L) ahead of the shock to a constant state (R) behind the shock. L is a supersonic state, and R is a subsonic state. The single interior point A is needed to satisfy the initial and boundary conditions. To see this, consider the equation governing the flow, which is of the form in Equation 81. Integrating this equation in space over the domain $(0, L)$ and in time over $(0, T)$ gives

$$\int_0^L Q(T)dx = \int_0^L Q(0)dx - \int_0^T (F_{RB} - F_{LB})dt, \tag{87}$$

where $Q(0)$ is the initial condition, and F_{RB} and F_{LB} are the fluxes and the right and left boundaries. This equation shows that the shock location, x_s, is fixed by the initial and boundary conditions, since

$$\int_0^L Q(T)dx = x_s Q_L + (L - x_s)Q_R. \tag{88}$$

Similarly, the discrete form of Equation 87 is

$$\Delta x \sum_j Q_j(T) = \Delta x \sum_j Q_j(0) - \int_0^T (F_{RB} - F_{LB}) dt. \tag{89}$$

In order for this relation to hold for general initial and boundary conditions, there must be an intermediate state A in the shock.

So the best we can do is capture a shock wave with one interior point. It is desirable to employ schemes that can do this. Smearing the shock over more points can reduce accuracy in smooth regions ahead and behind of the shock. Characteristic diffusion is capable of one interior point, but is expensive. CUSP is also capable of one interior point, but is much cheaper. The form for CUSP (using a general L-R notation) is

$$d_{LR} = \frac{1}{2}\alpha^* c(Q_R - Q_L) + \frac{1}{2}\beta(F_R - F_L), \tag{90}$$

where α^* and β are dimensionless constants. We can determine appropriate values for these constants by considering the flow just ahead and behind the shock. In supersonic flow, perfect upwinding is obtained by setting $\alpha^* = 0$, and $\beta = \text{sign}(M)$, where $M = u/c$ is the Mach number. For subsonic regions, consider cell $j+1$ in Figure 7. The numerical fluxes at the left and right boundaries of this cell are

$$\hat{F}_{RR} = F_R, \quad \hat{F}_{AR} = \frac{1}{2}(F_A + F_R) - \frac{1}{2}\alpha^* c(Q_R - Q_A) + \frac{1}{2}\beta(F_R - F_A). \tag{91}$$

At steady state, these fluxes must equate since the flux into the cell must equal the flux out of the cell. Equating fluxes results in the relation

$$F_R - F_A + \frac{\alpha^* c}{1+\beta}(Q_R - Q_A) = 0. \tag{92}$$

Substituting the Roe matrix in Equation 85 for the fluxes, we arrive at

$$A_{AR}(Q_R - Q_A) = -\frac{\alpha^* c}{1+\beta}(Q_R - Q_A), \tag{93}$$

which says that $-\frac{\alpha^* c}{1+\beta}$ is an Eigenvalue of A_{AR}. We know that the eigenvalues of A_{AR} are $\lambda(A_{AR}) = u, u+c, u-c$ using Roe-averaged values. The only choice for positive diffusion is to set

$$-\frac{\alpha^* c}{1+\beta} = u - c, \quad 0 < u < c. \tag{94}$$

Thus once we know $\alpha^* c$, we can compute β and vice versa, for subsonic flow.

Now we can also express CUSP in terms of a "convective" diffusion and a pressure contribution, as

$$d_{LR} = \frac{1}{2}\alpha c(Q_R - Q_L) + \frac{1}{2}\beta(\bar{Q}(u_R - u_L) + (F_{P,R} - F_{P,L})), \tag{95}$$

where

$$\alpha c = \alpha^* c + \beta \bar{u}, \quad \bar{Q} = \frac{1}{2}(Q_L + Q_R), \quad \bar{u} = \frac{1}{2}(u_L + u_R). \tag{96}$$

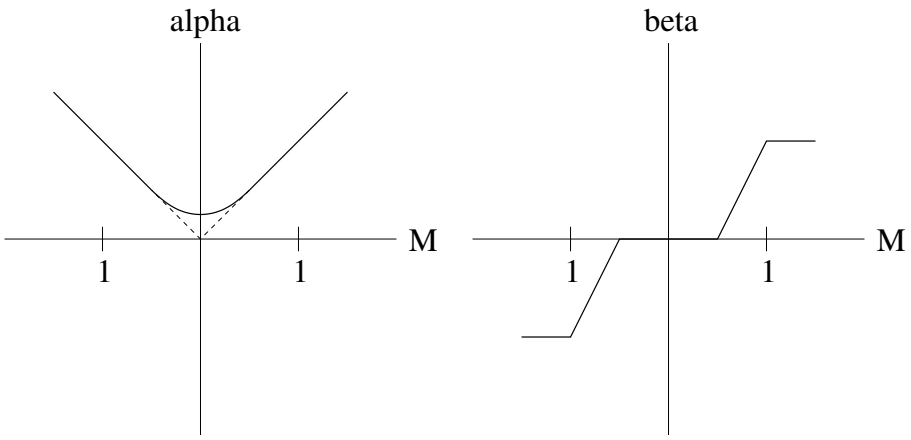

Figure 8: CUSP coefficients α and β as a function of Mach number.

So the total convective diffusion coefficient is αc, which we can set to $\alpha c = |u|$, similar to what we did in the scalar wave equation. Using the relation in Equation 94, and not allowing β to become negative for positive Mach numbers, and treating both positive and negative Mach numbers symmetrically, we arrive at

$$\alpha = |M|, \quad \beta = \begin{cases} \max(0, 2M-1) & 0 \le M \le 1, \\ \min(0, 2M+1) & -1 \le M \le 0, \\ \text{sign}(M) & |M| \ge 1. \end{cases} \tag{97}$$

The coefficient β, can equivalently be expressed as

$$\beta = \begin{cases} +\max(0, \frac{u+\lambda^-}{u-\lambda^-}) & 0 \le M \le 1, \\ -\max(0, \frac{u+\lambda^+}{u-\lambda^+}) & -1 \le M \le 0, \\ \text{sign}(M) & |M| \ge 1, \end{cases} \tag{98}$$

where $\lambda^\pm = u \pm c$. This form may be easier to implement in a computer program.

One final modification regards the behavior of CUSP near a stagnation point where the local Mach number tends to zero. To avoid lowering the artificial dissipation too much, the coefficient α is modified such that

$$\alpha = \frac{1}{2}\left(\alpha_0 + \frac{M^2}{\alpha_0}\right), \quad \text{if } |M| < \alpha_0, \tag{99}$$

where α_0 is typically set to $\alpha_0 = 1/2$. A sketch of α and β as a function of Mach number are shown in Figure 8

Just as in the linear wave equation, the upwinding produced as a result of CUSP leads to first order accuracy. However, we desire to maintain second order accuracy. This can be accomplished through "limited reconstruction," in which the left and right states for the artificial diffusion are reconstructed from surrounding nodal values to the control volume

interface. Linear reconstruction at the interface $j+\frac{1}{2}$ results in

$$Q_L = Q_j + \frac{s}{2}(x_{j+1} - x_j)\left.\frac{dQ}{dx}\right|_j, \quad Q_R = Q_{j+1} - \frac{s}{2}(x_{j+1} - x_j)\left.\frac{dQ}{dx}\right|_{j+1}, \tag{100}$$

where s is a limiter and the gradients of Q are estimates using surrounding nodal values. Using the reconstructed values actually renders the scheme non-LED. However, we really only need the LED property at local extrema. The role of the limiter, s, is to detect these local extrema, at which points we set $s = 0$. Otherwise, $s \approx 1$. This may be accomplished at $j+\frac{1}{2}$ by setting

$$s = 1 - \left|\frac{a-b}{|a|-|b|}\right|^q, \quad a = Q_{j+2} - Q_{j+1}, \quad b = Q_j - Q_{j-1} \tag{101}$$

where q is a positive integer (usally taken to be $q = 3$). If a and b are of opposite signs, which would be the case at a local extremum, then $s = 0$. In smooth regions, the limiter is nearly unity.

CUSP may be extended to the linear Galerkin scheme in Equation 57 by adding artificial diffusion to the central difference type fluxes,

$$V_0\frac{dQ_0}{dt} + \sum_i \hat{\mathcal{F}}_{0i} = 0, \quad \hat{\mathcal{F}}_{0i} = \frac{\mathcal{F}_0 + \mathcal{F}_i}{2} - d_{0i}, \tag{102}$$

$$d_{0i} = \frac{1}{2}\alpha^*(Ac)(Q_R - Q_L) + \frac{1}{2}\beta(\mathcal{F}(Q_R) - \mathcal{F}(Q_L)), \tag{103}$$

$$Q_L = Q_0 + \frac{s}{2}(\boldsymbol{r}_i - \boldsymbol{r}_0)^T\boldsymbol{\nabla}Q_0, \quad Q_R = Q_i - \frac{s}{2}(\boldsymbol{r}_i - \boldsymbol{r}_0)^T\boldsymbol{\nabla}Q_i. \tag{104}$$

Here, $\boldsymbol{r}_i - \boldsymbol{r}_0$ is the position vector connecting nodes 0 and i along an edge. Note the magnitude of the face area, A, is included in the CUSP formulation for triangles. All diffusive contibutions may be assembled in loops over edges and subsequently distributed to the nodes along an edge.

At boundaries, the diffusion should be reduced. For a boundary edge connecting nodes 0 and i, the numerical flux contribution to each node is

$$\hat{\mathcal{F}}_{0i} = \frac{5\mathcal{F}_0 + \mathcal{F}_i}{6} - d_{0i} \text{ (to node 0)}, \quad \hat{\mathcal{F}}_{0i} = \frac{\mathcal{F}_0 + 5\mathcal{F}_i}{6} - d_{0i} \text{ (to node } i\text{)}, \tag{105}$$

where the artificial diffusion contribution, d_{0i}, is the same for both nodes 0 and i, and is defined as

$$d_{0i} = \frac{1}{6}\alpha^*(Ac)(Q_R - Q_L) + \frac{1}{6}\beta(\mathcal{F}(Q_R) - \mathcal{F}(Q_L)), \tag{106}$$

where the 1/2 weighting is replaced with 1/6.

2.3 Boundary Conditions

Along with Equation 15, we also need to satisfy the boundary conditions for nodes lying on domain boundaries. In this class, we will talk about two classes of boundary conditions: Dirichlet and non-Dirichlet. Allmaras [1] provides a nice framework for the implementation of Dirichlet conditions, which includes no-slip walls, slip walls, inflow, and outflow. Non-Dirichlet conditions often arise at far field boundaries, and are often implemented through a modified flux definition.

2.3.1 Dirichlet Boundary Conditions

Oftentimes, PDE's may be derived by taking the variation of some function and setting that variation to zero. We call the PDE a "stationary" solution to the functional. We will not delve into the details of the calculus of variations in this class. Following Allmaras [1], the Navier-Stokes equations may be conceived as satisfying a stationary variation of an unknown functional, I, such that

$$\delta I(Q_s; \delta Q_s) = \int_V (\delta Q_s)^T \left[\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial x_j} - \frac{\partial F_j^v}{\partial x_j} \right] dV = 0, \tag{107}$$

where Q_s are some auxilliary variables used for convenience in the boundary formulation. Allmaras chooses entropy variables:

$$Q_s = \frac{\rho}{p} \begin{pmatrix} \frac{p}{\rho} \frac{\gamma+1-s}{\gamma-1} - e \\ u \\ v \\ -1 \end{pmatrix}, \tag{108}$$

where $s = \log(p/\rho^\gamma)$ For arbitrary δQ_s, this means that our original conservation law must be satisfied.

At the boundaries, we must satisfy l additional equations of the form

$$B(Q_s) = b, \tag{109}$$

where b is constant. This is a general form for Dirichlet boundary conditions. We can incorporate the l extra boundary equations through Lagrange multipliers in the variational statement by defining a new functional, $\tilde{I}$,

$$\tilde{I} = I + \int_{\Gamma_d} \lambda^T (B - b) d\Gamma, \tag{110}$$

where λ is the vector of Lagrange multipliers (length l), and the integral is taken over the portion of the boundary employing Dirichlet conditions. If we take the variation of this new functional and require it to be stationary, we arrive at

$$\delta \tilde{I}(Q_s, \lambda; \delta Q_s, \delta\lambda) = \int_V (\delta Q_s)^T \left[\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial x_j} - \frac{\partial F_j^v}{\partial x_j} \right] dV$$

$$+\int_{\Gamma_d}(\delta Q_s)^T\left[\frac{\partial B^T}{\partial Q_s}\lambda\right]d\Gamma+\int_{\Gamma_d}(\delta\lambda)^T\left[B(Q_s)-b\right]d\Gamma=0. \tag{111}$$

Away from Γ_d, nothing is changed and we have the same interior scheme. On Γ_d we have two new terms to deal with. Given the variational statement, we can derive the Galerkin weak form by substituting the test functions ϕ and θ for the variations δQ_s and $\delta\lambda$. We will keep ϕ the same tent function as before, but define θ to be the Dirac delta function. We will also represent λ with Dirac delta functions. The details of this procedure can be found in [1], but the result is the following set of equations at the boundary nodes:

$$\tilde{S}(Q,\lambda)=S(Q)+\frac{\partial B^T}{\partial Q_s}\lambda=0,$$

$$R_b(Q)=B(Q_s)-b=0, \tag{112}$$

where $S(Q)$ is the residual obtained without the boundary conditions, and $R_b(Q)$ is the "boundary residual." The second equation represents l boundary constraints, for which we added l Lagrange multipliers.

In this class we will deal with five types of Dirichlet conditions: inviscid wall, viscous wall, inflow, outflow, and pure Dirichlet where the entire state is specified and fixed. In the boundary condition definitions below, the *spec* values refer to specified values. For a slip (inviscid) wall, we have one condition ($l=1$), which is

$$B(Q_s)=u_n,\quad b=u_{n,wall}, \tag{113}$$

where $\boldsymbol{n}=(n_x,n_y)$ is the unit vector normal to the boundary, $u_n=\boldsymbol{u}\cdot\boldsymbol{n}$, and $u_{n,wall}=\boldsymbol{u}_{wall}\cdot\boldsymbol{n}$ is the velocity of the wall in a direction normal to the wall. The boundary condition Jacobian for a slip wall is

$$\frac{\partial B}{\partial Q_s}=\frac{p}{\rho}\begin{pmatrix}0 & n_x & n_y & \boldsymbol{u}\cdot\boldsymbol{n}\end{pmatrix}. \tag{114}$$

For a no-slip (viscous) wall, we have three conditions ($l=2$), which are

$$B(Q_s)=\begin{pmatrix}u\\v\end{pmatrix},\quad b=\begin{pmatrix}u_{wall}\\v_{wall}\end{pmatrix}. \tag{115}$$

The boundary condition Jacobian for a no-slip wall is

$$\frac{\partial B}{\partial Q_s}=\frac{p}{\rho}\begin{pmatrix}0 & 1 & 0 & u\\0 & 0 & 1 & v\end{pmatrix}. \tag{116}$$

For subsonic inflow, we have three conditions ($l=3$), which are

$$B(Q_s)=\begin{pmatrix}h\\s\\u_t\end{pmatrix},\quad b=\begin{pmatrix}h_{spec}\\s_{spec}\\u_{t,spec}\end{pmatrix}. \tag{117}$$

Here, $u_t = \boldsymbol{u} \cdot \boldsymbol{t}$ is the velocity tangential to the inflow plane, along the tangential vector, $\boldsymbol{t} = (-n_y, n_x)$ The boundary condition Jacobian for inflow is

$$\frac{\partial B}{\partial Q_s} = \begin{pmatrix} 0 & up/\rho & vp/\rho & (h + \frac{1}{2}q^2)p/\rho \\ -(\gamma-1) & -(\gamma-1)u & -(\gamma-1)v & -(\gamma-1)\frac{1}{2}q^2 \\ 0 & -n_y p/\rho & n_x p/\rho & u_t p/\rho \end{pmatrix}. \tag{118}$$

For subsonic outflow, we have one condition ($l = 1$), which is

$$B(Q_s) = p, \quad b = p_{spec}, \tag{119}$$

The boundary condition Jacobian for outflow is

$$\frac{\partial B}{\partial Q_s} = p \begin{pmatrix} 1 & u & v & h \end{pmatrix}. \tag{120}$$

For pure Dirichlet conditions, we have four conditions ($l = 4$), which are

$$B(Q_s) = \begin{pmatrix} p \\ u \\ v \\ T \end{pmatrix}, \quad b = \begin{pmatrix} p_{spec} \\ u_{spec} \\ v_{spec} \\ T_{spec} \end{pmatrix}. \tag{121}$$

Typically, we specify pressure, velocities, and temperature rather than the conserved variables for convienience. The boundary condition Jacobian for a pure Dirichlet condition is

$$\frac{\partial B}{\partial Q_s} = \frac{p}{\rho} \begin{pmatrix} \rho & \rho u & \rho v & \rho h \\ 0 & 1 & 0 & u \\ 0 & 0 & 1 & v \\ 0 & 0 & 0 & T \end{pmatrix}. \tag{122}$$

2.3.2 Non-Dirichlet Boundary Conditions

Certain boundary conditions are not Dirichlet. One way to handle these conditions is through a flux modification. Actually, many Dirichlet conditions are treated through a flux modification, however, it is not clear how to verify the order of accuracy of boundary conditions that are implemented through flux modification. Therefore, when possible we will use the Dirichlet framework above.

Just as an example of a flux modification boundary implementation, consider a slip wall where we wish to enforce tangency. A Dirichlet implementation of this condition is described in the previous section, however a common implementation of this condition is a flux modification. Recall that the inviscid directed flux may be expressed as

$$\mathcal{F} = A_x F + A_y G = q_n \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho h \end{pmatrix} + \begin{pmatrix} 0 \\ pA_x \\ pA_y \\ 0 \end{pmatrix}, \tag{123}$$

where $q_n = A_x u + A_y v$ is the directed velocity at an edge. The inviscid wall condition states that there should be no component of velocity normal to the surface such that $q_n = 0$. So one way to implement this condition would be to modify the flux definition every time we are on an inviscid boundary edge, such that

$$\mathcal{F} = \begin{pmatrix} 0 \\ pA_x \\ pA_y \\ 0 \end{pmatrix}. \tag{124}$$

Many CFD codes do this. However, as previously stated, it is not clear how to verify the order of accuracy of such an implementation, and methods which cannot be verified are not very useful. Therefore, we will treat inviscid walls with the Dirichlet framework.

The one boundary that cannot be treated with a Dirichlet framework for the purposes of this class is a far field boundary condition for external aerodynamics computations. We will implement this condition using a modified flux based on Riemann invariants. Consider the eigenvalue decomposition performed in Equation 24 in 1D. The characteristic variables, W, are $W = (s, R^+, R^-)$, where s is the entropy, and $R^\pm$ are known as the Riemann invariants:

$$R^\pm = u \pm \frac{2c}{\gamma - 1}. \tag{125}$$

It is clear that R^+ is a quantity that travels to the right at speed $u + c$, while R^- travels to the left at speed $u - c$.

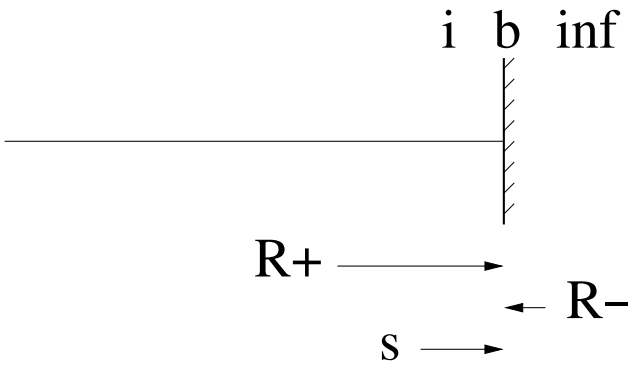

Figure 9: Outflow boundary in one dimension (subsonic).

We can use the Riemann invariants to define the boundary state in subsonic flow. Consider the outflow boundary in 1D, shown in Figure 9. Denote the boundary values as b, the state just to the left of the boundary i (for interior), and the state just to the right of the boundary values ∞, (for freestream conditions). It would be consistent with the wave directions to define the boundary state as

$$R_b^+ = u_i + \frac{2c_i}{\gamma - 1}, \quad R_b^- = u_\infty - \frac{2c_\infty}{\gamma - 1}, \quad s_b = s_i. \tag{126}$$

From this definition, we could obtain the velocity and speed of sound at the boundary as

$$u_b = \frac{1}{2}(R_b^+ + R_b^-), \quad c_b = \frac{\gamma - 1}{4}(R_b^+ - R_b^-), \tag{127}$$

which, with the entropy s, completely defines the state at the boundary node.

Unfortunately, the Riemann invariant method only works exactly in one dimension. However, we can apply the method locally to a far field boundary in multiple dimensions by replacing the derivative in x with a derivative in the direction of the outward normal to the boundary, n. Doing this results in

$$R_b^+ = u_{ni} + \frac{2c_i}{\gamma - 1}, \quad R_b^- = u_{n\infty} - \frac{2c_\infty}{\gamma - 1}, \quad s_b, u_{tb} = \begin{cases} s_i, u_{ti} & \tilde{u}_{ni} > 0, \\ s_\infty, u_{t\infty} & \tilde{u}_{ni} < 0 \end{cases}, \tag{128}$$

where u_n is the component of velocity normal to the boundary, u_t is the component of velocity tangential to the boundary, and $\tilde{u}_{ni}$ is an estimate of the normal velocity at a previous time step. In practice, at a boundary edge connecting two nodes, one may employ the method in Equation 133 to obtain the boundary state at each of the nodes. Once the state is determined, the inviscid fluxes may be computed and weighted using the $5/6, 1/6$ weighting obtained from the Galerkin discretization.

There is one important modification to the method of Riemann invariants for subsonic flow that is important for lifting solutions. The Riemann method requires the use of freestream values at ∞ to determine the state. The circulation computed with freestream values around the far field boundary would be zero which, in light of the Kutta-Joukowski relation would be inconsistent with a lifting solution (eg. an airfoil at angle of attack). To fix this inconsistency, what is often done is to impose a small compressible irrotational vortex by locally modifying the freestream quantities to produce a circulation which matches the lifting solution.

Using the Kutta-Joukowski relation, $l = \rho_\infty q_\infty \Gamma$, the definition of incompressible lift coefficient, $c_{l,0} = l/(1/2\rho_\infty q_\infty^2)a$, and the Prantl-Glauert compressibility correction, $c_l = c_{l,0}/\sqrt{1 - M_\infty^2}$, we may deduce the amount of circulation needed to be consistent with the lift:

$$\Gamma = \frac{1}{2} c_{l,0} M_\infty c_\infty a \sqrt{1 - M_\infty^2}, \tag{129}$$

where M_∞ is the freestream Mach number, and a is the chord length of the lifting body. We can then impose a compressible irrotational vortex at the quarter-chord of the lifting body, which has tangential velocity

$$u_{t,vortex} = \frac{\Gamma}{2\pi R(1 - M_{t\infty}^2)}, \tag{130}$$

where $M_{t\infty} = \boldsymbol{u}_\infty / c_\infty \cdot \boldsymbol{t}$ is the component of the free stream Mach number in the direction of the vortex velocity, and R is the distance from the quarter chord to the point of application of the boundary condition. Here, $M_{t\infty}$ is computed as

$$M_{t\infty} = \frac{1}{c_\infty}(u_\infty \sin\theta - v_\infty \cos\theta), \tag{131}$$

where θ is the polar angle measured from the x-axis at the point of application of the boundary condition. The corrected freestream quantities to be used in the Riemann invariant method are then

$$u_{\infty,corrected} = u_\infty + u_{t,vortex} \sin\theta, \quad v_{\infty,corrected} = v_\infty - u_{t,vortex} \cos\theta \tag{132}$$

The Riemann method and freestream velocity correction needs only to be used for subsonic flow. for supersonic flow, the boundary states may be obtained with

$$Q_b = \begin{cases} Q_i & \tilde{u}_{ni} > 0, \\ Q_\infty & \tilde{u}_{ni} < 0 \end{cases}, \tag{133}$$

which is consistent with the directions of wave propagation. This treatment is critical for stability at far field boundaries.

2.3.3 Boundary Normal Vector Computation

In order to satisfy the boundary conditions, we often need the outward pointing normal vector for the implementation of boundary conditions. The normal at a boundary node can be computed from the normals of the faces touching that node. Consider the boundary face in Figure 10. The nodes are numbered such that if you walk from node 1 to node 2, the

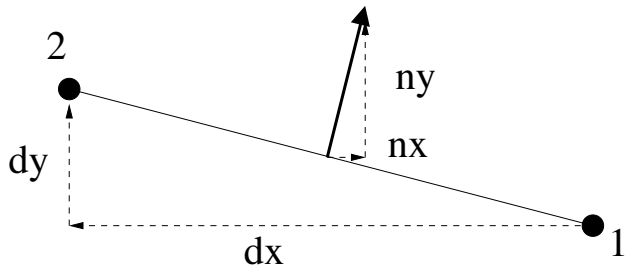

Figure 10: Boundary face normal computation.

boundary is on the right and the interior domain is on the left. Using this convention, the outward normal times the face area (length of the edge) is

$$n_x dA = dy, \quad n_y dA = -dx. \tag{134}$$

By considering the two faces that touch a boundary node, we can compute the node normal with

$$n_x = \frac{\sum_i dy}{\sqrt{\left(\sum_i dy\right)^2 + \left(\sum_i (-dx)\right)^2}}, \quad n_y = \frac{\sum_i (-dx)}{\sqrt{\left(\sum_i dy\right)^2 + \left(\sum_i (-dx)\right)^2}} \tag{135}$$

where the summation is over the i faces touching a given boundary node.

2.4 Least Squares Gradient Computation

In order to achieve a second order accurate scheme we will need the gradients of the conserved variables. One method would be to use Equation 27, but over the entire volume surrounding node 0 shown in Figure 2. Such a procedure is exact for a linear function since it assumes a constant gradient and uses the midpoint rule. Another method which is often used is based on a least squares approach. This approach has the advantage that, in principle, we can compute gradients that are exact for higher order functions, such as quadratics or cubics. In addition, we have more flexibility to choose a different stencil than just the nearest neighbors that are connected by edges.

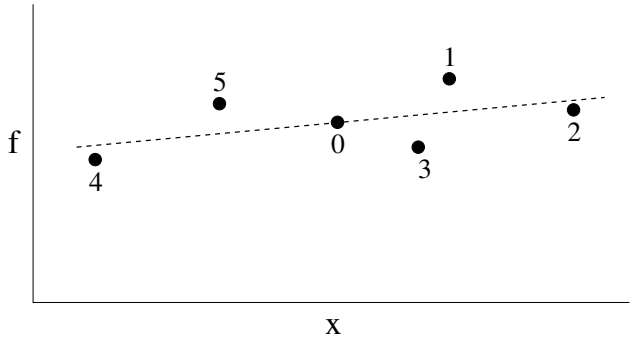

Figure 11: Illustration of linear least squares in one dimension.

The basic idea of using least squares to compute gradients is illustrated in Figure 11 for a function f in one dimension. Suppose we know the value of f at a number of locations near the node 0. We can estimate df/dx at 0 by finding a linear function which is a "best fit" to the data and also passes through node 0. The slope of this line gives us an estimate of the derivative of f. It is easy to see that if f were linear, all the values of f would lie in a straight line, and there would be no approximation error in determining the best fit linear function through the data, and hence, df/dx.

In two dimensions, we fit a plane through the data, which we can express in a form that looks like a Taylor series truncated after the linear terms,

$$\overline{f}(x,y) = f_0 + (x - x_0)\frac{\partial \overline{f}}{\partial x} + (y - y_0)\frac{\partial \overline{f}}{\partial y}, \tag{136}$$

where the bar over the f indicates that this is only a linear approximation. If we expand this Taylor series to each point i in our local least squares stencil, we get

$$f_i - f_0 = (x_i - x_0)\frac{\partial \overline{f}}{\partial x} + (y_i - y_0)\frac{\partial \overline{f}}{\partial y}, \tag{137}$$

$$\Delta f_{0i} = \Delta x_{0i}\frac{\partial \overline{f}}{\partial x} + \Delta y_{0i}\frac{\partial \overline{f}}{\partial y}, \tag{138}$$

where f_i is the discrete value of f at node i, and the Δ notation has been introduced for convenience. If we had only two points in the local least squares stencil, we would have two

equation with two unknowns, $\partial\overline{f}/\partial x$ and $\partial\overline{f}/\partial y$. This is the minimum number of points that exactly defines a plane. Generally, we pick more points in the stencil than the minimum because it helps make the code more robust. In this case, we have $n > 2$ equations, but still only two unknows for the components of the gradients. We can use a least squares approach to find the gradients, in which we find $\partial\overline{f}/\partial x$ and $\partial\overline{f}/\partial y$ such that ϕ,

$$\phi\left(\frac{\partial\overline{f}}{\partial x}, \frac{\partial\overline{f}}{\partial y}\right) = \sum_{i=1}^{n} w_i \left(\Delta x_{0i}\frac{\partial\overline{f}}{\partial x} + \Delta y_{0i}\frac{\partial\overline{f}}{\partial y} - \Delta f_{0i}\right)^2, \tag{139}$$

is a minimum. Here w_i is a weighting term that emphasizes certain nodes over others in the minimization process. Generally, we want nodes that are closer to node 0 to matter more than nodes that are farther away. Typically we set

$$w_i = \frac{1}{\Delta x_{0i}^2 + \Delta y_{0i}^2}, \tag{140}$$

which is just the inverse distance of node 0 to node i, squared.

An equivalent way of stating the least squares problem that is convenient for programming is

$$A(\boldsymbol{\nabla}\overline{f}) = B(\boldsymbol{\Delta} f), \tag{141}$$

where

$$A = \begin{bmatrix} \sum w_i \Delta x_{0i}^2 & \sum w_i \Delta x_{0i}\Delta y_{0i} \\ \sum w_i \Delta x_{0i}\Delta y_{0i} & \sum w_i \Delta y_{0i}^2 \end{bmatrix}, \tag{142}$$

$$B = \begin{bmatrix} w_1\Delta x_{01} & w_2\Delta x_{02} & \cdots & w_n\Delta x_{0n} \\ w_1\Delta y_{01} & w_2\Delta y_{02} & \cdots & w_n\Delta y_{0n} \end{bmatrix}, \tag{143}$$

$$\boldsymbol{\nabla}\overline{f} = \begin{Bmatrix} \partial\overline{f}/\partial x \\ \partial\overline{f}/\partial y \end{Bmatrix}, \tag{144}$$

$$\boldsymbol{\Delta} f = \begin{Bmatrix} \Delta f_{01} \\ \Delta f_{02} \\ \vdots \\ \Delta f_{0n} \end{Bmatrix}. \tag{145}$$

We can solve for the gradients by inverting the symmetric matrix A, which leads to

$$\boldsymbol{\nabla}\overline{f} = A^{-1}B(\boldsymbol{\Delta} f) = C(\boldsymbol{\Delta} f), \tag{146}$$

where C is a $(2 \times n)$ matrix. Since C only depends on the geometry (x and y coordinates) and not on the solution (f), we can compute the entries of C and store them in the grid setup phase once and for all. If we write C as

$$C = \begin{bmatrix} c_{x1} & c_{x2} & \cdots & c_{xn} \\ c_{y1} & c_{y2} & \cdots & c_{yn} \end{bmatrix}, \tag{147}$$

then when we need gradients of any function f at node 0, we can easily compute them as

$$\frac{\partial \overline{f}}{\partial x} = \sum_{i=1}^{n} c_{xi} \Delta f_{0i}, \tag{148}$$

$$\frac{\partial \overline{f}}{\partial y} = \sum_{i=1}^{n} c_{yi} \Delta f_{0i}, \tag{149}$$

or equivalently

$$\frac{\partial \overline{f}}{\partial x} = \sum_{i=0}^{n} c_{xi} f_i, \tag{150}$$

$$\frac{\partial \overline{f}}{\partial y} = \sum_{i=0}^{n} c_{yi} f_i, \tag{151}$$

where

$$c_{x0} = -\sum_{i=1}^{n} c_{xi}, \tag{152}$$

$$c_{y0} = -\sum_{i=1}^{n} c_{yi}. \tag{153}$$

This form provides some convenient checks to see if our computation of c_{xi} and c_{yi} are correct. We know our gradient formulas should compute constant and linear functions exactly. So if $f = K$, where K is a constant then we know that

$$\frac{\partial \overline{f}}{\partial x} = \sum_{i=0}^{n} c_{xi} K = K \sum_{i=0}^{n} c_{xi}, \tag{154}$$

so therefore, $\sum_{i=0}^{n} c_{xi} = 0$. Similarly, $\sum_{i=0}^{n} c_{yi} = 0$. Likewise, our formulas must compute the gradient $f = x$ exactly, which is just $\partial \overline{f}/\partial x = 1$, and $\partial \overline{f}/\partial y = 0$. A similar argument is made for $f = y$. This means that

$$\sum_{i=0}^{n} c_{xi} x_i = 1, \quad \sum_{i=0}^{n} c_{xi} y_i = 0, \tag{155}$$

$$\sum_{i=0}^{n} c_{yi} x_i = 0, \quad \sum_{i=0}^{n} c_{yi} y_i = 1, \tag{156}$$

These checks should be made as soon as the least squares weights are computed.

Exercises

1. Show that the Galerkin finite element method for the Euler equations is exact if F and G are linear functions in space.

3 Solution Methods for Steady and Unsteady Problems

For purposes of obtaining a solution to the temporally and spatially discretized fluid equations, we may express the set of coupled, non-linear equations as

$$S(Q) = 0, \tag{157}$$

where S is termed the "residual." From the previous section, we know that S contains the spatial discretization of the flux using a Galerkin method and the temporal discretization using the backward difference formula. However, the methods used to solve Equation 3.1.1 are general enough to apply to a variety of discretization schemes.

To facilitate a solution, we add a pseudo-time derivative,

$$\frac{\partial Q}{\partial \tau} + S(Q) = 0, \tag{158}$$

and march to a pseudo steady state. The goal of this section is to outline a method for obtaining a steady state in the fastest way possible. Essentially what we have in Equation 158 is an ODE in pseudo-time. Thus, we can use ODE stability theory to design schemes that help achieve steady-state very quickly.

3.1 Analysis of Ordinary Differential Equations

To simplify the analysis, consider a "model" ODE of the form

$$\frac{du}{d\tau} = \lambda u, \tag{159}$$

which has an exact solution

$$u(\tau) = u^0 e^{\lambda \tau}, \tag{160}$$

and we consider $\lambda = \lambda_r + i\lambda_i$ a general complex number. Here, u^0 is the initial value of u at $\tau = 0$. The exact solution in Equation 160 can be written (using Euler's formula) as

$$u(\tau) = u^0 e^{\lambda_r \tau}(\cos \lambda_i \tau + i \sin \lambda_i \tau). \tag{161}$$

In this form it is easy to see that the exact solution is bounded when $\lambda_r < 0$, which makes the oscillatory part decay in time.

Using the model equation, we can investigate the stability of various time integration schemes. Perhaps the simplest is forward Euler:

$$\frac{u^{k+1} - u^k}{\Delta \tau} = \lambda u^k. \tag{162}$$

The discrete solution at the k^{th} pseudo-time level may computed as

$$u^k = \sigma^k u^0, \tag{163}$$

where the amplification factor for the forward Euler scheme is

$$\sigma = 1 + \lambda\Delta\tau. \tag{164}$$

For stability, we need $|\sigma| \leq 1$, which places a time step restriction for the forward Euler scheme. The magnitude of the amplification factor squared is

$$|\sigma|^2 = (1 + \lambda_r\Delta\tau)^2 + (\lambda_i\Delta\tau)^2, \tag{165}$$

which, when set equal to 1 produces a circular stability region of unit radius centered at -1 on the real axis. This is a small stability region compared to the stability region of the exact ODE, which is the entire left-hand plane. Stability restrictions such as this are typical of explicit methods.

Now consider the backward Euler scheme, which is perhaps the simplest implicit method:

$$\frac{u^{k+1} - u^k}{\Delta\tau} = \lambda u^{k+1}. \tag{166}$$

Note that the right-hand side is evaluated at time $k + 1$. This scheme results in an amplification factor,

$$\sigma = \frac{1}{1 - \lambda\Delta\tau}, \tag{167}$$

the magnitude of which is

$$|\sigma|^2 = \frac{1}{(1 - \lambda_r\Delta\tau)^2 + (\lambda_i\Delta\tau)^2}. \tag{168}$$

Since $\lambda_r < 0$, the amplification factor is always less than unity for any $\Delta\tau$, and we have unconditional stability. Unconditional stability is typical of implicit schemes.

Whatever time integration scheme we choose we may extend the analysis to systems of equations, such as the Euler equations. Consider a system of linear ODE's,

$$\frac{du}{d\tau} = Au, \tag{169}$$

where A is a general matrix, independent of u. We can diagonalize A by

$$A = X\Lambda X^{-1}, \tag{170}$$

where X contains the Eigenvectors of A, and Λ is a diagonal matrix containing the Eigenvalues of A. Defining a new variable $v = X^{-1}u$, we may multiply the original system by X^{-1}, resulting in

$$\frac{dv}{d\tau} = \Lambda v, \tag{171}$$

which is a set of decoupled ODE's of the form of our original model equation. All our results for scalar equations hold for the system by considering the magnitude of the largest Eigenvalue of A, known as the spectral radius of A.

Analysis of ODE stability helps us understand time step restrictions arising from explicit schemes. It would seem logical to always choose an implicit scheme to avoid these restrictions,

especially since we are marching to steady state as fast as we can to solve Equation 158. This is not always the case. Consider the forward Euler scheme applied to Equation 158,

$$\frac{Q^{k+1} - Q^k}{\Delta\tau} + S(Q^{k+1}) = 0, \tag{172}$$

where the residual is evaluated at time $k+1$. Since this equation is non-linear, we typically "linearize" the equations by expanding the residual as

$$S(Q^{k+1}) \approx S(Q^k) + \frac{\partial S^k}{\partial Q}(Q^{k+1} - Q^k), \tag{173}$$

where the matrix $\frac{\partial S^k}{\partial Q}$ is known as the "Jacobian of the residual." Substituting the linearized residual into Equation 172 we arrive at a linear system,

$$\left(\frac{I}{\Delta\tau} + \frac{\partial S^k}{\partial Q}\right)\Delta Q = -S(Q^k), \tag{174}$$

where $\Delta Q = Q^{k+1} - Q^k$. When the corrections ΔQ go to zero we have satisfied the residual equation. For infinite $\Delta\tau$, which is possible with a fully implicit scheme, Equation 174 becomes Newton's method, which only requires around 5 iterations to achieve convergence.

There are two problems with using a Newton method for our particular scheme on triangles. First, Newton's method requires a good initial guess to converge. To avoid this problem, we often choose $\Delta\tau$ to be small initially, and ramp up the time step slowly until convergence is reached. The other problem, which is more severe, is that the Jacobian of the residual may require very large storage requirements. The linear system at each pseudo-time step becomes very expensive to solve, outweighing any benefits from the large time steps.

By contrast, forward Euler is a much simpler, low-memory method. Forward Euler applied to Equation 158 is

$$\frac{Q^{k+1} - Q^k}{\Delta\tau} + S(Q^k) = 0, \tag{175}$$

which provides a simple iteration update at each pseudo-step. In practice, a mix of implict and explicit methodology may be optimal. This is an active area of research in CFD.

3.1.1 Explicit Runge-Kutta Schemes

There is another popular class of time integration schemes called Runge-Kutta (RK) methods. These methods provide high-order time accuracy with good stability characteristics, and they are self-starting (do not require information from previous time steps). A general $s-$stage RK method may be written as

$$Q^m = Q^k + \Delta\tau \sum_{i=1}^{s} a_{mi} S(Q^i), \quad m = 1, 2, \cdots, s, \tag{176}$$

$$Q^{k+1} = Q^k + \Delta\tau \sum_{i=1}^{s} b_i S(Q^i), \tag{177}$$

where Q^m is the value of Q at the m^{th} stage. Three RK schemes are often of interest when time accuracy is a concern: RK1 (forward Euler), RK2, and RK4, where the number denotes the number of stages. These schemes are explicit, but implicit RK schemes are also possible. These schemes allow for a larger time step, even extending into the right-hand plane in some cases. This means that the discrete scheme may be stable for an exact solution which is not stable!

We can describe a general explict RK scheme by using a "Butcher table" (sounds gory), which takes the form in Table 1. The a and b coeficients are used in Equations 176 and 177.

Table 1: **Butcher table for a general 4-stage explicit RK scheme.**

0	0			
c_2	a_{21}			
c_3	a_{31}	a_{32}		
c_4	a_{41}	a_{42}	a_{43}	
	b_1	b_2	b_3	b_4

The c coefficients denote the time that each stage is evaluated at within the time step, such that $t^m = t^k + c_m \Delta\tau$. In the fluid equations, time does not appear explicitly, and so the c coefficients are never used.

In terms of the Butcher table, forward Euler is expressed as

0	
	1

Similarly, the popular RK4 scheme is expressed as

0	0			
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

The popularity of RK4 stems from its 4th order accuracy and good stability characteristics. It may seem that this would also be a good scheme for the fluids equations. However, in our context, we only need to integrate in time to reach a steady state. Therefore, we can do better than the classical RK4 scheme for our application since we do not care at all about temporal accuracy. We do, however, care very much about stability, and desire the increased stability bounds that RK methods often provide.

To devise a suitable RK scheme for pseudo-time integration of the fluids equations, consider the model problem

$$\frac{\partial u}{\partial \tau} + a\frac{\partial u}{\partial x} + \mu\Delta x\frac{\partial^2 u}{\partial x^2} = 0, \tag{178}$$

where a is the coefficient of convection, and μ is the coefficient of diffusion. The diffusive term represents some form of artificial diffusion or physical diffusion introduced through the Navier-Stokes viscous terms. Central differencing may be used on the convective term provided the diffusion is of sufficient magnitude to enforce upwinding. Therefore, we may semi-discretize Equation 178 as

$$\frac{\partial u}{\partial \tau} + \frac{a}{\Delta x}(u_{j+1} - u_{j-1})\frac{\mu}{\Delta x}(u_{j+1} - 2u_j + u_{j-1}) = 0. \tag{179}$$

We can represent the solution with pseudo-time dependent parts and spatial parts using a Fourier signal,

$$\begin{aligned} u_j &= \hat{u}(\tau)e^{i\omega x_j}, \\ u_{j+1} &= \hat{u}(\tau)e^{i\omega(x_j+\Delta x)}, \\ u_{j-1} &= \hat{u}(\tau)e^{i\omega(x_j-\Delta x)}. \end{aligned}$$

Substitution of these Fourier-decomposed elements into the discretization results in

$$\frac{\partial \hat{u}}{\partial \tau} = \lambda \hat{u}, \tag{180}$$

where

$$\lambda = \frac{-1}{\Delta x}\left[ia\sin(\omega\Delta x) - 4\mu\sin^2\left(\frac{\omega\Delta x}{2}\right)\right]. \tag{181}$$

From this equation we can see that the stability limit of the convective portion of the discretization is determined by the imaginary axis, while stability of the diffisive terms are determined by the extent of the stability region on the negative real axis. This suggests that by treating convective and diffusive terms independently, we may be able to optimize the overall convection-diffusion system.

Jameson introduced such a strategy by splitting the residual in Equation into convective and diffusion components:

$$S(Q) = R(Q) + D(Q). \tag{182}$$

The Butcher table for the convective terms is simply

0						
α_1	α_1					
α_2	0	α_2				
α_3	0	0	α_3			
α_4	0	0	0	α_4		
α_5	0	0	0	0	α_5	
	0	0	0	0	α_5	0

For the diffusive terms, the Butcher table is

0						
α_1	$\alpha_1\beta_1$					
α_2	$\alpha_2(1-\beta_2)$	$\alpha_2\beta_2$				
α_3	0	$\alpha_3(1-\beta_3)$	$\alpha_3\beta_3$			
α_4	0	0	$\alpha_4(1-\beta_4)$	$\alpha_4\beta_4$		
α_5	0	0	0	$\alpha_5(1-\beta_5)$	$\alpha_5\beta_5$	
	0	0	0	$\alpha_5(1-\beta_5)$	$\alpha_5\beta_5$	0

The combined scheme then becomes

$$\begin{aligned} Q^0 &= Q^k \\ Q^m &= Q^k - \alpha_m \Delta\tau \left[R(Q^{k-1}) + \beta_k D(Q^{k-1}) + (1-\beta_k) D(Q^{k-2}) \right] \\ Q^{k+1} &= Q^5. \end{aligned}$$

The advantage of this formulation is that the α and β coefficients may be opitimized independently to increase the stability region for convective and diffusive contributions, respectively. A good scheme is

$$\begin{aligned} \alpha_1 &= 1/4, & \beta_1 &= 1 \\ \alpha_2 &= 1/6, & \beta_2 &= 0 \\ \alpha_3 &= 3/8, & \beta_3 &= .56 \\ \alpha_4 &= 1/2, & \beta_4 &= 0 \\ \alpha_5 &= 1, & \beta_5 &= .44 \end{aligned}$$

Notice that certain $\beta = 0$, which means that the diffusive contributions, which are often expensive to compute, need not be evaluated at every stage.

3.2 Stability of Hyperbolic Partial Differential Equations

Unique stability aspects are introduced when we consider PDE's instead of just ODE's. Again, we start with model equations. The most general explict scheme for a scalar equation involving u at node i in a mesh is

$$u_i^{k+1} = \sum_j a_{ij} u_j^k, \tag{183}$$

where a_{ij} are scalars independent of u. A way to ensure stability is

$$||u^{k+1}||_\infty \leq ||u^k||_\infty, \tag{184}$$

where the infinity norms are defined as

$$||u^{k+1}||_\infty = \max_i |u_i^{k+1}|, \tag{185}$$

$$||u^k||_\infty = \max_j |u_j^k|, \tag{186}$$

which are the maximum of the absolute values of u over the mesh. The stability condition in Equation 184 is satisfied when

$$\sum_j |a_{ij}| \leq 1, \tag{187}$$

for all nodes i in the mesh. This can be shown by considering at node i,

$$|u_i^{k+1}| \leq \sum_j |a_{ij}||u_j^k| \leq \left(\sum_j |a_{ij}|\right) \max_j |u_j^k| = \left(\sum_j |a_{ij}|\right) ||u^k||_\infty. \tag{188}$$

Thus, the stability condition is satisfied if $\sum_j |a_{ij}| \leq 1$.

With that in mind, consider the "Courant-Friedrichs-Lewy (CFL) condition," which states:

The region of dependence of the numerical scheme must contain the region of dependence of the true PDE.

To illustrate the importance of the CFL condition for stability, consider the linear wave equation,

$$\frac{\partial u}{\partial \tau} + a\frac{\partial u}{\partial x} = 0, \tag{189}$$

which represents propagation of some quantity $u(x,t)$ at constant speed a. The solution to this PDE is $u(x - a\tau, \tau) = u(x, 0)$, which is illustrated in Figure 12. Information propagates

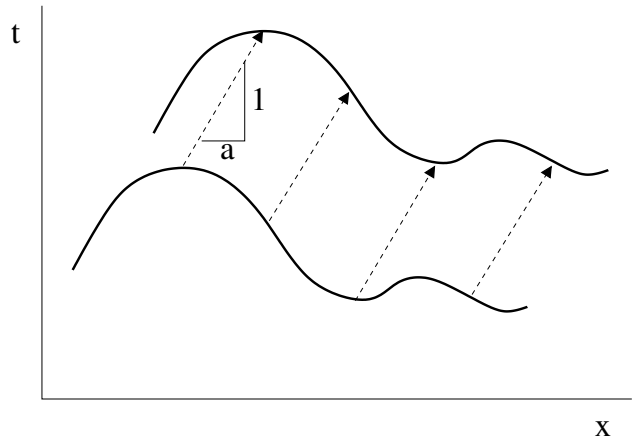

Figure 12: Solution of the linear wave equation in space and time.

along lines of $x - a\tau = const$. These lines along which the solution is constant in space and time are called "characteristics." In the language of the CFL condition, the domain of dependence of the true PDE at any point is the information obtained by marching backward along a characteristic.

Characteristics need not have the same slope, which is an important way to understand shock wave formation. Shock waves are an inherently non-linear phenomenon, so we must consider a non-linear model equation, such as Burger's equation

$$\frac{\partial u}{\partial \tau} + u\frac{\partial u}{\partial x} = 0. \tag{190}$$

Here, the wave speed changes with the local value of u. The solution in space and time to Burger's equation looks something like the illustration in Figure 13. Notice that the slopes of

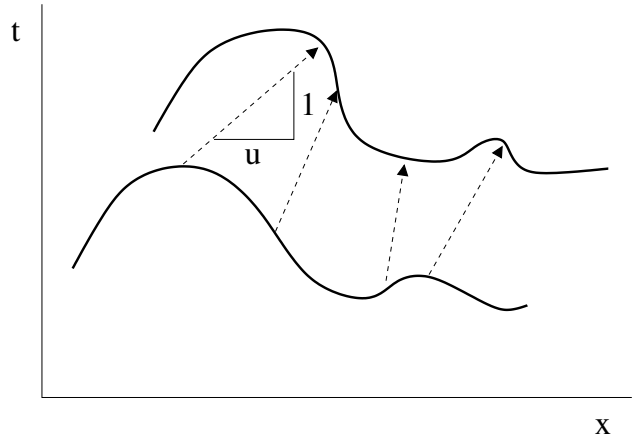

Figure 13: Solution of Burger's equation in space and time.

the characteristics depend locally on the value of u. Information at the peaks moves faster than information in the valleys, causing a steepening of the wave forms and an eventual discontinuity (shock wave).

Application of the forward Euler, first order upwind scheme to the linear wave equation gives

$$u_j^{k+1} = u_j^k - \frac{a\Delta\tau}{\Delta x}(u_j^k - u_{j-1}^k) = (1 - \text{CFL})u_j^k + \text{CFL}u_{j-1}^k, \tag{191}$$

where the CFL number is defined as $\text{CFL} = a\Delta\tau/\Delta x$. Recall from our stability analysis we need $\sum_j |a_{ij}| \leq 1$, which would mean $\text{CFL} \leq 1$ for the forward Euler scheme. This CFL condition is neccessary for stability, but not sufficient.

To illustrate the CFL condition graphically, consider the mesh of points in space and time in Figure 14. The domain of dependence of the numerical scheme at location (j, k) in space and time does not contain the domain of dependence of the governing PDE, which depends on some initial value outside the discrete stencil. Thus, the CFL condition is violated, and we would expect instability. To fix this, we can make the time step smaller, such that the domain of dependence of the exact PDE is included, as shown in Figure 15. In fact, in order to include the domain of dependence of the PDE, we need to pick a time step such that

$$a\frac{\Delta\tau}{\Delta x} < 1, \tag{192}$$

which is precisely the CFL condition found from our stability analysis.

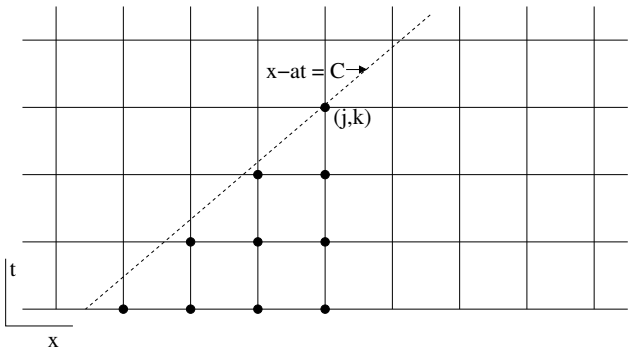

Figure 14: CFL violation for forward Euler upwind scheme.

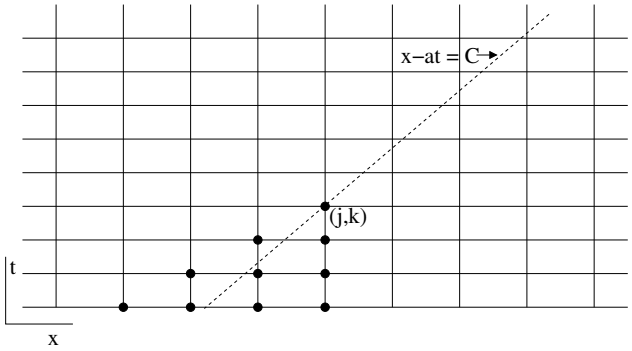

Figure 15: CFL satisfaction for forward Euler upwind scheme.

Figure 16 shows the case of forward Euler with central differencing in space instead of upwind. This case highlights the fact that the CFL condition is merely neccessary, but not sufficient, for stability. Numerical evidence shows clear instability for central differencing, even though the CFL condition may be satisfied.

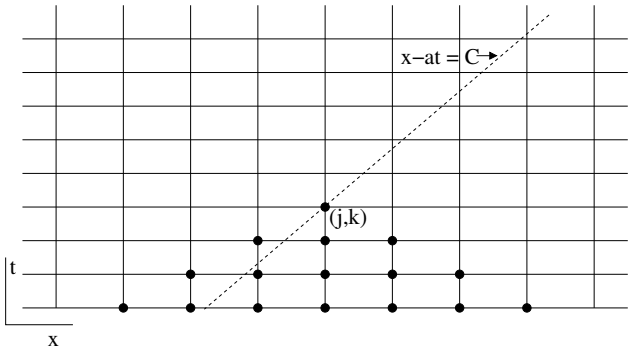

Figure 16: CFL satisfaction for forward Euler central scheme. This scheme is unstable.

Recall that for the Euler equation (in 1D), we do not have a single wave speed a. Instead we have three wave speeds: u, $u + c$, and $u - c$, which represent the convective speed and

two acoustic speeds. In this case we must look at the largest wave speed to satisfy the CFL condition. This point is illustrated in Figure 17, which shows that the CFL condition is satisfied for the two smaller wave speeds, but is violated for the largest wave speed.

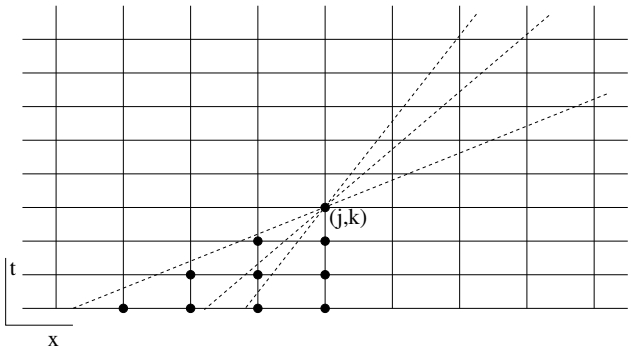

Figure 17: CFL violation for multiple wave speeds.

3.3 CFL Definition for Linear Galerkin Scheme on Triangles

3.3.1 Inviscid Terms

Recall the approximation to the Euler equations using a Galerkin method on linear triangles in Equation 57,

$$V_0 \frac{dQ_0}{dt} + \sum_i \frac{\mathcal{F}_0 + \mathcal{F}_i}{2} = 0, \tag{193}$$

where V_0 is one third of the volume of all triangles touching node 0 and $\mathcal{F} = A_x F + A_y G$ is the directed flux through area weighted normal A. For nodes lying on the boundary, different weighting should be used when adding in the contributions from the boundary edges.

A good CFL definition for this scheme on isotropic grids is

$$\text{CFL} = \frac{\overline{\lambda}\Delta\tau}{V}, \tag{194}$$

where V is the median-dual control volume, $\Delta\tau$ is the pseudo-time step, and $\overline{\lambda}$ is an estimate of the spectral radius (magnitude of the largest Eigenvalue) of the Jacobian of the flux terms. The spectral radius represents the speed of the fastest traveling information represented by the Euler equations. The Eigenvalues of the directed flux Jacobian, $\mathcal{A} = \partial\mathcal{F}/\partial Q$, are

$$\lambda(\mathcal{A}) = q_n, q_n, q_n + Ac, q_n - Ac, \tag{195}$$

where $q_n = A_x u + A_y v$ is the directed edge velocity (multiplied by the face area), and $Ac = A\sqrt{\gamma p/\rho}$ is the speed of sound multiplied by the magnitude of the face area. The magnitude of the largest Eigenvalue is $|q_n| + Ac$, where the absolute value is included to account for both directions of flow. The spectral radius for the inviscid terms may then be estimated by

$$\overline{\lambda} = \sum_i \frac{(|q_n| + Ac)_0 + (|q_n| + Ac)_i}{2}, \tag{196}$$

where different weights are employed at the boundary edges.

3.3.2 Viscous Terms

The viscous discretization is implemented as a sum over triangles, as expressed in Equation 69. It is possible to express this as edge contributions, but this introduces significant complexity. Nonetheless, the viscous spectral radius can be assembled as an edge loop with good results.

Instead of a CFL number which we used for the inviscid terms, we may define a "Von Neumann" number (VNN), in order to estimate our maximum allowable time step with viscosity:

$$\text{VNN} = \frac{\overline{\lambda}^v \Delta\tau}{V}, \tag{197}$$

where

$$\overline{\lambda}^v = \sum_i \frac{\lambda_0^v + \lambda_i^v}{2V_0}, \tag{198}$$

and

$$\lambda^v = \frac{\gamma\mu}{\text{Pr}\rho} A^2, \tag{199}$$

These expressions are derived in Ref. [6]

3.4 Unsteady Flows

For unsteady flows, both physical- and pseudo-time terms will be present in our solution method of Equation 15:

$$\frac{\partial Q}{\partial \tau} + \frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial x_j} = \frac{\partial F_j^v}{\partial x_j}, \tag{200}$$

where τ is pseudo-time and t is physical time. We have discussed at length how to treat the pseudo-time to reach a steady state as fast as possible through stability analysis and ODE theory. Actually, the physical time derivative is much simpler to incorporate into our algorithm and requires only minimal changes over the steady state problem.

To match the second order spatial accuracy we achieve with the Galerkin method, we often choose a second order backward difference formula in physical time:

$$\frac{\partial Q}{\partial t} = \frac{1}{2\Delta t}\left[3(Q^{n+1}) - 4(Q)^n + (Q)^{n-1}\right], \tag{201}$$

where superscript n denotes the location in physical time, and Δt is the physical time step. In the solution method, we will treat all the spatial derivative terms implicitly at time level $n+1$ This allows us to avoid physical time step restrictions.

Recall that after multiplying by a test function and integrating in space, we multiply the equations by units of volume. This is analogous to the finite volume form, in which we integrate the differential form over a volume in space. For a moving mesh, we must account for changing volume within the time derivative as well:

$$\frac{\partial (VQ)}{\partial t} = \frac{1}{2\Delta t}\left[3(VQ)^{n+1} - 4(VQ)^n + (VQ)^{n-1}\right] \tag{202}$$

The physical time term results in the need to solve a pseudo-steady-state problem at each physical time step. Thus, there are two levels of iterating needed for the unsteady Navier-Stokes equations. First, we iterate in physical time by marching forward from the initial conditons. Second, at each of these physcial time locations, we iterate in pseudo-time until we reach a pseudo-steady-state. Actually, each pseudo-time step often requires some form of iterating, making a third level. In our case, we perform several stages of a Runge-Kutta method for this third level. For implicit schemes, often we must perform Gauss-Seidel, Jacobi, or another iterative linear solution method at each pseudo-time step.

3.5 Moving Meshes

Recall that the linear Galerkin scheme on triangles is essentially a finite volume scheme centered at the nodes in the mesh. We can use this observation to account for mesh movement for unsteady flow computations. Mesh movement may consist of rigid motion or mesh deformation. Rigid motion account for single body problems in which the body does not change shape (e.g. pitching airfoil). Mesh deformation is needed if the body changes shape (e.g. flapping wing). The methods developed below work for either rigid motion or deformation.

The finite volume form of Equation 15 is obtained by integrating over a (moving) control volume, $V(t)$,

$$\int_{V(t)} \frac{\partial Q}{\partial t} dV + \int_{A(t)} F_j n_j dA = \int_{A(t)} F_j^v n_j dA. \tag{203}$$

Recall the Liebniz theorem,

$$\frac{d}{dt} \int_{V(t)} Q dV = \int_{V(t)} \frac{\partial Q}{\partial t} dV + \int_{A(t)} Q \dot{x}_j n_j dA, \tag{204}$$

where $A(t)$ is the (moving) control volume boundary, and $\dot{x}_j$ is the velocity of the control volume boundary. Substituting Equation 204 into Equation 203 results in

$$\frac{d}{dt} \int_{V(t)} Q dV + \int_{A(t)} \left(F_j - Q\dot{x}_j\right) n_j dA = \int_{A(t)} F_j^v n_j dA. \tag{205}$$

It is clear from this equation that the moving mesh term only effects the inviscid fluxes while the viscous flux terms remain unchanged. We can therefore drop the viscous terms for our analysis. For piecewise constant solution in control volumes with planar sides, the discretized form of Equation 205 becomes

$$\frac{d}{dt}(V_0 Q_0) + \sum_i \frac{(\mathcal{F} - Q\dot{X}_n)_0 + (\mathcal{F} - Q\dot{X}_n)_i}{2} = 0, \tag{206}$$

where $\mathcal{F} = F_j n_j dA$ is the dot product of the inviscid flux with the directed face area, and $\dot{X}_n = \dot{x}_j n_j dA$ is the dot product of the face velocity with the directed face area evaluated at the quadrature point of the planar control volume face.

Comparing Equation 206 with our previous static mesh linear Galerkin discretization in Equation 57, we note two differences. First, the volume remains inside the time derivative, which makes sense, since for deforming meshes the volume will change in time. Second, the

directed flux is modified by subtracting the mesh velocity term, $Q\dot{X}_n$. The Jacobian of this modified directed flux is

$$\tilde{\mathcal{A}} = \frac{\partial(\mathcal{F} - Q\dot{X}_n)}{\partial Q} = \frac{\partial \mathcal{F}}{\partial Q} - \frac{\partial(Q\dot{X}_n)}{\partial Q} = \mathcal{A} - \dot{X}_n I, \tag{207}$$

which is the original directed flux Jacobian minus the directed face velocity. The Eigenvalues of the modified Jacobian are found by the characteristic equation

$$\det(\mathcal{A} - \dot{X}_n I - \lambda I) = \det(\mathcal{A} - \tilde{\lambda} I) = 0, \tag{208}$$

where $\tilde{\lambda} = \lambda + \dot{X}_n$. This means that the Eigenvalues of $\tilde{\mathcal{A}}$ are the Eigenvalues of $\mathcal{A}$ minus the directed face velocities,

$$\lambda(\tilde{\mathcal{A}}) = (q_n - \dot{X}_n), (q_n - \dot{X}_n), (q_n - \dot{X}_n) + Ac, (q_n - \dot{X}_n) - Ac. \tag{209}$$

Comparing these Eigenvalues to the static mesh Eigenvalues in Equation 195, we have simply subtracted the directed face velocities from q_n. The spectral radius of the moving mesh Eigenvalues is now $|q_n - \dot{X}_n| + Ac$, where the absolute value accounts from propagation in both directions.

3.6 Solution of the Final Unsteady System with Boundary Conditions

Recall at boundary nodes we must solve the extended system of equations:

$$\begin{Bmatrix} R_b(Q) \\ \tilde{S}(Q, \lambda) \end{Bmatrix} = 0, \tag{210}$$

where R_b is the boundary residual of Equation 112, and $\tilde{S}$ is the modified residual containing convective terms (R), diffusive/viscous terms (D), unsteady time terms (T), and boundary terms (P),

$$\tilde{S}(Q, \lambda) = S(Q) + P(Q, \lambda),$$
$$S(Q) = R(Q) + D(Q) + T(Q),$$
$$P(Q) = \frac{\partial B^T}{\partial Q_s}\lambda.$$

Here, B depends on the type of boundary condition used, and λ is a vector of length l of Lagrange multipliers introduced to satisfy l additional boundary conditions. Forms for the vector B and its Jacobian are found in Section 2.3.1. The unsteady time term contains the second order backward difference formula,

$$T(Q) = \frac{1}{2\Delta t}\left[3(VQ) - 4(VQ)^n + (VQ)^{n-1}\right], \tag{211}$$

where superscript n denotes the location in physical time, and Δt is the physical time step. Notice that the $n+1$ term is has no subscript to indicate that it is treated implicitly, at the same time level as all other terms. This allows us to avoid physical time step restrictions.

All our discretization effort boils down to solving the $n+l$ system in Equation 210 at each node. If not at a boundary node, then $l=0$, and R_b and P are omitted. What is the best way to solve this system of equations? Recall, that for the equations of motion, we add a pseudo-time derivative,

$$V\frac{dQ}{d\tau}+\tilde{S}(Q,\lambda)=0, \tag{212}$$

and drive the system to pseudo-steady state. We may use Runge-Kutta methods that are optimized for convection-diffusion systems, where the m^{th} stage solution may be obtained by solving

$$\frac{V}{\alpha_m\Delta\tau}(Q^m-Q^k)+R(Q^{m-1})+D(Q^{m-1})+T(Q^m)+P(Q^m,\lambda^m)=0. \tag{213}$$

Here α_m is a Runge-Kutta coefficient, and $\Delta\tau$ is the pseudo-time step obtained from CFL considerations. The superscript m denotes the Runge-Kutta stage, and the superscript k denotes the pseudo-time iteration. Note that both T and P are treated implicitly in the pseudo-time. Treating T implicitly is important for stability, and treating P implicitly allows us to solve for λ along with Q at each stage.

Let us take a closer look at the implicit treatment of T and P. We can express $T(Q^m)$ as

$$\begin{aligned} T(Q^m) &= \frac{1}{2\Delta t}\left[\left(3(VQ)^m-3(VQ)^k\right)+3(VQ)^k-4(VQ)^n+(VQ)^{n-1}\right] \\ &= \frac{3V}{2\Delta t}\Delta Q+T(Q^{m-1}), \end{aligned}$$

where $\Delta Q=Q^m-Q^k$. Strictly speaking the $m-1$ superscript above should be k, but evaluating it at $m-1$ turns out to be more convenient and doesn't change the accuracy, since it is only a change at the pseudo-time level.

Let us also examine the implicit treatment of P by defining an extended vector of unknowns of length $n+l$,

$$\tilde{Q}=\begin{Bmatrix}\lambda\\ Q\end{Bmatrix}, \tag{214}$$

which contains the Lagrange multipliers as well as the conserved variables. Then we may define $P(\tilde{Q}^m)$ as

$$\begin{aligned} P(\tilde{Q}^m) &= P(\tilde{Q}^{m-1})+\frac{\partial P^{m-1}}{\partial\tilde{Q}}(Q^m-Q^k) \\ &= P(\tilde{Q}^{m-1})+\frac{\partial P^{m-1}}{\partial\tilde{Q}}\Delta Q. \end{aligned}$$

Strictly speaking, Q^k should really be Q^{m-1}, but similar to the physical time term, we can use Q^k since we are not concerned about the accuracy in pseudo time. It turns out to be a choice based on convenience.

Substituting these definitions of T and P into Equation 213 yields

$$\frac{V}{\alpha_m\Delta\tau}\Delta Q+\frac{3V}{2\Delta t}\Delta Q+\frac{\partial P^{m-1}}{\partial\tilde{Q}}\Delta Q=-\tilde{S}(Q^{m-1},\lambda^{m-1}). \tag{215}$$

Defining intermediate variables

$$a = \frac{\alpha_m \Delta\tau}{V}, \quad b = \frac{3V}{2\Delta t}, \tag{216}$$

and multiplying Equation 215 by a yields

$$\left[(1+ab)I + a\frac{\partial P^{m-1}}{\partial \tilde{Q}}\right]\Delta Q = -a\tilde{S}(Q^{m-1}, \lambda^{m-1}). \tag{217}$$

Again, if we are not on a boundary, we can solve for ΔQ directly with

$$\Delta Q = -\frac{a}{1+ab}S(Q^{m-1}). \tag{218}$$

Furthermore, for steady cases, $b = 0$.

At boundaries, we must solve Equation 217 along with the boundary residual equation, $R_b(Q) = 0$. If we treat the boundary residual equation implicitly and linearize in pseudo time, we obtain,

$$\frac{\partial R_b^{m-1}}{\partial Q}\Delta Q = -R_b(Q^{m-1}). \tag{219}$$

We can write Equations 217 and 219 as a single matrix system

$$A\Delta\tilde{Q} = C, \quad A_{(n+l)\times(n+l)} = \left[\begin{array}{c|c} 0_{l\times l} & \left(\frac{\partial R_b}{\partial Q}\right)_{l\times n} \\ \hline \left(a\frac{\partial P}{\partial \lambda}\right)_{n\times l} & \left((1+ab)I + a\frac{\partial P}{\partial Q}\right)_{n\times n} \end{array}\right], \tag{220}$$

$$\Delta\tilde{Q} = \begin{Bmatrix} \Delta\lambda \\ \Delta Q \end{Bmatrix}, \quad C = \begin{Bmatrix} -R_b^{m-1} \\ -a\tilde{S}^{m-1} \end{Bmatrix} \tag{221}$$

This system is solved at each boundary node to obtain the updates ΔQ and $\Delta\lambda$ at each Runge-Kutta stage. Iterations continue until ΔQ and $\Delta\lambda$ become sufficiently small and pseudo-steady state has been obtained. When this has happened, then $C = 0$, and we have solved the discretized equations of motion and boundary conditions simultaneously.

3.7 Preconditioning

The preceeding solution method works reasonably well for transonic flows. However, certain low Mach number flows and supersonic flows present unique difficulties to this method due to disparate wave speed magnitudes. For example, consider one-dimensional flow at $M = 0.1$. We know from an Eigenvalue analysis that entropy waves will travel at $u = .1c$, and acoustic pressure waves will travel at $u + c = 1.1c$ and $u - c = .9c$. In this case the convective speeds are an order of magnitude smaller than the acoustic speeds. Thus, disturbances in our system traveling at the convective speeds will take a long time to propagate out of the domain, leading to slow convergence. Similar behavior is observed for flows around the sonic point ($M = 1.0$).

Preconditioning techniques are aimed in part at equalizing the magnitude of the eigenvalues of the system such that disturbances propagate at the same speed in order to accelerate

convergence. This idea can be illustrated in one dimension. Let us take the Euler equations and multiply the pseudo-time derivative by a "preconditioning matrix," Γ,

$$\Gamma \frac{\partial Q}{\partial \tau} + A \frac{\partial Q}{\partial x} = 0, \tag{222}$$

where A is the flux Jacobian. Note that the steady state of this equation is unchanged from the non-preconditioned form, which we could revert to by setting $\Gamma = I$. Suppose we set $\Gamma = |A|$, where $|A| = X^{-1}|\Lambda|X$. Then we can show that

$$\begin{aligned}
\frac{\partial Q}{\partial \tau} &+ |A|^{-1} A \frac{\partial Q}{\partial x} = 0, \\
\frac{\partial Q}{\partial \tau} &+ X^{-1}|\Lambda|^{-1} X X^{-1} \Lambda X \frac{\partial Q}{\partial x} = 0, \\
\frac{\partial Q}{\partial \tau} &+ X^{-1} \frac{\Lambda}{|\Lambda|} X \frac{\partial Q}{\partial x} = 0, \\
\frac{\partial Q}{\partial \tau} &+ A' \frac{\partial Q}{\partial x} = 0.
\end{aligned}$$

Now $\lambda(A') = \text{sign}\,(\lambda(A))$, such that $\lambda(A') = \pm 1$. The Eigenvalues of the preconditioned have maintained the correct sign, but they have been normalized to all have magnitude of unity.

In two and three dimensions, such perfect preconditioning does not exist. However, we can pick a preconditioning matrix that attempts to equalize the Eigenvalues while reducing storage. In Equation 212, we can add a preconditioning matrix,

$$\Gamma V \frac{dQ}{d\tau} + \tilde{S}(Q, \lambda) = 0. \tag{223}$$

Again, note that the steady-state in unchanged, and the preconditioning in this form only serves to accelerate convergence to this steady-state. Now, in two-dimensions Γ is a 4x4 matrix to be stored at each node in the grid, which greatly increases the storage requirements. Furthermore, in order to compute our ΔQ updates at each Runge-Kutta stage, we will need to invert this matrix at each node, which adds significant computational expense.

Suppose that we could find a preconditioning matrix such that $\Gamma = \mathcal{S}\overline{\Gamma}\mathcal{S}^{-1}$, where $\overline{\Gamma}$ is a symmetric matrix, and $\mathcal{S}$ serves to symmetrize Γ. In that case, our storage of Γ would be cut in half due to symmetry, and we could use Cholesky decomposition to easily compute the inverse of $\overline{\Gamma}$. The Cholesky decomposition may be obtained for symmetric positive definite matrices, leading to the form $\overline{\Gamma} = LL^T$, where L is a lower triangular matrix. Thus, the solution of a linear system $\overline{\Gamma}x = b$ is facilitated by forward and backward sweeps:

$$\begin{aligned}
\overline{\Gamma}x &= b, \\
LL^T x &= b, \\
Ly &= b, \\
y &= L^T x.
\end{aligned}$$

Application of the symmetric preconditioning matrix to Equation 223 leads to

$$\begin{aligned}
\mathcal{S}\overline{\Gamma}\mathcal{S}^{-1}V\frac{dQ}{d\tau} &+ \tilde{S} = 0, \\
V\frac{dQ}{d\tau} &+ \mathcal{S}\overline{\Gamma}^{-1}\mathcal{S}^{-1}\tilde{S} = 0, \\
V\frac{dQ}{d\tau} &+ \mathcal{S}\overline{\Gamma}^{-1}\tilde{S}_1 = 0, \\
V\frac{dQ}{d\tau} &+ \mathcal{S}\tilde{S}_2 = 0, \\
V\frac{dQ}{d\tau} &+ \tilde{S}_3 = 0.
\end{aligned}$$

This allows us to efficiently compute the update at each Runge-Kutta stage.

The incorporation of boundary conditions with preconditioning only requires a small change. Equation 217 becomes

$$\left[\Gamma + (ab)I + a\frac{\partial P^{m-1}}{\partial \tilde{Q}}\right]\Delta Q = -a\tilde{S}(Q^{m-1}, \lambda^{m-1}). \tag{224}$$

If not on a boundary, the update becomes

$$\Delta Q = -(\Gamma + (ab)I)^{-1}aS(Q^{m-1}). \tag{225}$$

Since $\Gamma + (ab)I$ can also be symmetrized as $\Gamma + (ab)I = \mathcal{S}\overline{\Gamma}'\mathcal{S}^{-1}$, where $\overline{\Gamma}' = \overline{\Gamma} + (ab)I$, the Cholesky decomposition technique and reduced storage still hold. For steady flows, $b = 0$, and $\overline{\Gamma}' = \overline{\Gamma}$. For boundary nodes, a full inversion of Equation 220 is still necessary, with the added preconditioning matrix

$$A\Delta\tilde{Q} = C, \quad A_{(n+m)\times(n+m)} = \left[\begin{array}{c|c} 0_{l\times l} & \left(\frac{\partial R_b}{\partial Q}\right)_{l\times n} \\ \hline \left(a\frac{\partial P}{\partial \lambda}\right)_{n\times l} & \left(\Gamma + (ab)I + a\frac{\partial P}{\partial Q}\right)_{n\times n} \end{array}\right]. \tag{226}$$

To improve the effect of the preconditioning, we can use an averaging procedure to compute $\overline{\Gamma}$:

$$\overline{\Gamma}_0 = \sum_i \frac{(\overline{\Gamma}_{node,0} + \overline{\Gamma}_{node,i})}{2}, \tag{227}$$

where the final preconditioning matrix is computed as a sum of the surrounding preconditioning matricies. Here a simple average is shown for interior edges, however 1/6, 5/6 weighting should be used for boundary edges.

The specific forms of $\overline{\Gamma}$, $\mathcal{S}$, and $\mathcal{S}^{-1}$ are

$$\overline{\Gamma} = \begin{bmatrix} R_1 & n_xR_2 & n_yR_2 & 0 \\ & n_y^2Q_1 + n_x^2R_1 & n_xn_y(R_1 - Q_1) & 0 \\ & & n_x^2Q_1 + n_y^2R_1 & 0 \\ & & & Q_1 \end{bmatrix}, \quad \text{symmetric} \tag{228}$$

$$\mathcal{S} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ u & c & 0 & u \\ v & 0 & c & v \\ h & uc & vc & \frac{q^2}{2} \end{bmatrix}, \tag{229}$$

$$\mathcal{S}^{-1} = \begin{bmatrix} \frac{\gamma-1}{2}\frac{q^2}{c^2} & -\frac{\gamma-1}{2}\frac{u}{c^2} & -\frac{\gamma-1}{2}\frac{v}{c^2} & \frac{\gamma-1}{c^2} \\ -\frac{u}{c} & \frac{1}{c} & 0 & 0 \\ -\frac{v}{c} & 0 & \frac{1}{c} & 0 \\ 1-\frac{\gamma-1}{2}\frac{q^2}{c^2} & \frac{\gamma-1}{2}\frac{u}{c^2} & \frac{\gamma-1}{2}\frac{v}{c^2} & -\frac{\gamma-1}{c^2} \end{bmatrix}. \tag{230}$$

In the above matrices,

$$R_1 = \frac{Q_2+Q_3}{2}, \quad R_2 = \frac{Q_2-Q_3}{2}, \quad Q_1 = |q_n|, \quad Q_2 = |q_n + Ac|, \quad Q_3 = |q_n - Ac|,$$

$$q^2 = u^2 + v^2. \tag{231}$$

In addition, the eigenvalues Q_i are modified if they drop below a certain threshold to avoid near-zero elements:

$$\text{if} \quad Q_i < \kappa \quad \text{then} \quad Q_i \to \frac{1}{2}\left(\kappa + \frac{Q_i^2}{\kappa}\right) \tag{232}$$

where $\kappa = Ac$ seems to work well.

4 Verification in CFD

The goal of any discretization scheme, be it first-, second-, or higher-order, is to compute a discrete solution, Q^h, which approaches the exact solution, Q, to the continuous equations as the mesh is refined. The degree to which the discrete and exact solutions do not coincide is referred to here as *solution error*. The rate at which the solution error decreases as the mesh is refined leads to the notion of *order of accuracy*. Solution error is not the same as *truncation error*, which may be defined as the residual present when an exact solution is introduced into the discretization. For unstructured grid finite volume schemes, the two types of error do not neccessarily converge at the same rate. Generally, solution error converges at a faster rate than truncation error for arbitrary meshes [3]. In fact, many finite volume schemes are inconsistent in the truncation error, but still converge to the exact solution [2, 8]. Ultimately, solution error is the primary quantity of interest in assessing order of accuracy.

While convergence rates of truncation error and solution error are not necessarily equal, the two types of error are related. To better understand this, consider the general form of a conservation law of Equation 15,

$$\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial x_j} - \frac{\partial F_j^v}{\partial x_j} = 0, \tag{233}$$

where Q represents conserved variables, F is the inviscid flux, and F^v is the viscous flux. For a moment, consider an inviscid flow ($F^v = 0$), with linear F. The discretization of Equation 15 at steady state becomes

$$DQ^h = BQ_b, \tag{234}$$

where D represents the discretization operating on the discrete solution, and B incorporates the Dirichlet conditions, Q_b. Note that the discrete solution exactly satisfies the system of equations while the exact solution, Q, applied to the discrete equations yields an error such that,

$$DQ = BQ_b + E_t, \tag{235}$$

where E_t is the truncation error.

Using the preceeding equations, we can relate the solution error, E_s, to the truncation error by

$$E_s = Q - Q^h = D^{-1}E_t, \tag{236}$$

or

$$DE_s = E_t. \tag{237}$$

A few observations can be made from the preceeding analysis:

1. The solution error is a solution to the PDE governing Q when driven by the local truncation error as a source term. It is appropriate to think of the truncation error as driving the solution error.

2. The order of convergence of solution error need not be the same as the order convergence of truncation error.

3. We must actually solve the system of equations to determine the solution error, which is of primary interest. Simply computing the residual from the substitution of the exact solution only gives truncation error, which is of secondary importance.

Verification in CFD is all about determining E_s, which decreases as some power of the mesh size,

$$E_s = Kh^p + H.O.T. \tag{238}$$

where K is a constant, p determines the *order of accuracy*, and $H.O.T.$ stands for higher order terms. As we refine the mesh, the $H.O.T.$ become insignificant compared to the leading order term. The exponent p is 0 for and inconsistent scheme, 1 for a first order accurate scheme, 2 for a second order accurate scheme, and so on. From its definition in Equation 236, to determine solution error we need an exact solution against which we can compare our discrete solution. For simple equations, exact solutions are trivial to find. However, for non-linear systems of equations, such as the Euler or Navier-Stokes equations, non-trivial exact solutions are extremely rare or may not exist at all. After all, this is why we are using CFD in the first place. The following sections will detail a method to get around this apparent difficulty in order to determine the solution error.

In contrast to verification, *validation in CFD* refers to comparison of computed results to experimental data. We will treat validation in Section 5

4.1 The Method of Manufactured Solutions

The Method of Manufactured Solutions (MMS) provides a way to verify CFD codes in the absence of exact solutions. MMS allows us to determine the order of accuracy of the interior scheme, boundary scheme, and any effects that mesh quality might have on the solution error. Instead of looking for an exact solution to Equation 233, we pick a "manufactured solution" of our choosing. Since this solution will not satisfy Equation 233 in general, we must add a source term on the right hand side such that our manufactured solution satisfies a modified equation,

$$\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial x_j} - \frac{\partial F_j^v}{\partial x_j} = S(\boldsymbol{x}, t), \tag{239}$$

where S is the source term, which depends on the coordinates and possibly time.

A simple example illustrates the procedure of determining S. Consider the wave equation in 1D,

$$\frac{\partial Q}{\partial t} + a\frac{\partial Q}{\partial x} = 0,$$

where a is the wave speed. Exact solutions to this equation are just $Q = const.$. However, this doesn't help us determine the accuracy of a discretization method because the exact solution is too simple. Almost any discretization of the wave equation could compute a constant solution exactly. Instead, let us pick a manufactured solution arbitrarily, say $Q = \sin \beta x$, where β is a constant. This solution does not satisfy the original governing equation, because only a constant solution is an exact solution. However, we can add a source term to force our manufactured solution to become an exact solution. If we pick $S = a\beta \cos \beta x$, then the

new governing equation becomes

$$\frac{\partial Q}{\partial t} + a\frac{\partial Q}{\partial x} = S(x,t) = a\beta\cos\beta x.$$

We can verify that our manufactured solution, $Q = \sin\beta x$, satisfies this modified equations exactly at steady state. The way we found S was to simply substitute our manufactured solution into the governing equation.

We can apply this method to more complex systems, like the Navier-Stokes equations. For a system of equations, we have to define manufactured solutions for each component. For example a good manufactured solution for density is

$$\rho(x,y) = \rho_0 + \rho_x \sin\left(\frac{\alpha_{\rho x}\pi x}{L}\right) + \rho_y \cos\left(\frac{\alpha_{\rho y}\pi y}{L}\right) + \rho_{xy}\cos\left(\frac{\alpha_{\rho xy}\pi xy}{L^2}\right),$$

where the ρ_q and α_q terms are constants. We could pick similar forms for pressure, and velocity components to complete the solution description. It is important to pick a manufactured solution that exercises all terms in the governing equations. If the solution is too simple, our method may compute the solution exactly, which will not give us the solution error that we are seeking. Additionally, we need to pick values for the constants which are physical. For example, we would not want to pick $\rho_0 = -100$ because negative density is non-physical.

In the 1D wave equation example above, it was trivial to compute S. However, for more complex manufactured solutions and equations, S can become very difficult to compute analytically. Symbolic math packages, such as are available in MATLAB, can help immensely.

An important point in using MMS is that to get accurate results, we must discretize the source term in a manner consistent with the way we discretize other terms in our governing equations. Following our Galerkin method, this means we should multiply S by a test function, $\phi = \phi(x,y)$, and integrate over space:

$$\int_V \phi S dV.$$

If we consider a linear approximation on triangles, we can show that the source term contribution to node 0 surrounded by nodes i is

$$\int_V \phi S dV = \sum_i \frac{S_0 + S_i}{2}\frac{\boldsymbol{A}_{0i}\cdot\boldsymbol{l}_{0i}}{4}, \tag{240}$$

where $\boldsymbol{A}_{0i}$ is the median-dual face area separating nodes 0 and i, and $\boldsymbol{l}_{0i}$ is the vector connecting node 0 and node i.

4.2 Verification of Boundary Conditions

Boundary conditions may also be verified using MMS. In addition to adding a source term to Equation 239, we also add a source term to Equation 112:

$$B(Q_s) - b = S_b(\boldsymbol{x}, t). \tag{241}$$

Here, S_b is found in the same way as S, by substituting the manufactured solution into Equation 112. The boundary residual which we wish to drive to zero becomes

$$R_b(Q) = B(Q_s) - b - S_b(\boldsymbol{x}, t). \tag{242}$$

The same manufactured solution that is used in the interior should also be used on the boundaries. Of course, the source terms will be different since the governing equations are different.

5 Sample CFD Solutions and Validation

A Linux and All That

For this class, you will need access to some sort of Linux platform that you can access easily in class. Linux is free, so if you have your own laptop that you prefer to work on, you could install linux on your laptop and bring that to class. Mac OSX also has the cabability to act like a linux workstation through the command prompt application. If you do not have your own laptop, you can easily obtain an account on the USU HPC cluster. You can log into this cluster from a windows machine (such as those in ENGR 305) using `PuTTY` and (optionally) an X-server, such as `Xming` if you wish to use graphics applications over the network.
The USU HPC website has nice tutorials that will guide you through the process of obtaining an account on the cluster, as well as the login process via PuTTY or through another linux machine. This information can be accessed at:

`https://www.hpc.usu.edu/wiki/GettingStarted`

Follow the instructions for "Getting a DoRC Account," for how to create and log into your account on the cluster. Most students will likely use `PuTTY` to access their accounts on the cluster from the 3rd floor ENGR lab computers during class. See this page for a convenient method of accessing files from the 3rd floor computers:

`https://wiki.rc.usu.edu/wiki/FAQ_AccessingFilesFromWindows7`

An important point when using the cluster is that you should not run code on the "login" nodes. You are automatically placed on one of the login nodes when you access your account on the cluster. To access the compute nodes, the first command you should issue upon logging into the cluster is

```
qsub -I -q interactive -l walltime=1:00:00
```

where `-I` stands for "interactive," and "walltime" specifies the time of your interactive session on the compute nodes. Make sure when you are done with your session, that you log out of your session by typing `exit`. The above command is for a one hour session, which is enough for a class period. More time can be requested for work beyond the class period. You can check the status of your interactive session by typing:

```
qstat -u [A-number]
```

You can delete a specific interactive session by typing:

```
qdel [job-number]
```

If you have never worked in a Linux environment, it might take you a few days to get down some of the basic commands of navigating and managing files and directories. Your fellow classmates and Google can be great resources in this process. I have also included a file on Canvas called `Introduction_to_unix.pdf` that I obtained from the HPC website

that provides a nice introduction to Unix/Linux.

You will also need to be familiar with a text editor such as vi, vim, gedit, or emacs. I prefer to use emacs, but it does require an X-server to run over a network, which can be slow at times. A safe choice is vi or vim if you don't mind learning a new set of commands. Again, your fellow classmates and Google can be great resources here.

To practice compiling and running C++ and Fortran code on the cluster or a similar linux-based system, I have included a test code, `cpp_fortran_test.tgz`, on the course website on Canvas. After downloading the zipped file from Canvas, transfer the file to your account on the cluster:

```
scp cpp_fortran_test.tgz A00308839@login.rc.usu.edu: /
```

If you are using a Windows operating system, follow the instructions for mounting the remote cluster storage locally to your windows machine. Otherwise, you can use WinSCP or a similar tool to copy files to the cluster. Without WinSCP you can use `sftp` and a DOS prompt. Open a DOS prompt and change directories (`cd`) to the location where `cpp_fortran_test.tgz` is located on your local windows machine. Then add to the path variable by typing

```
set path=%path%;c:\software\cygwin\bin
```

Then, start an `sftp` session:

```
sftp [your A#]@login.rc.usu.edu
```

Answer "yes" to the security question, then enter your password on the cluster. Then issue the command

```
put [filename]
```

where `filename` is the name of the file your wish to transfer. The file should then be in your home directory on the cluster.

Then, login to your account on the cluster:

```
ssh A00308839@login.rc.usu.edu
```

Again, if you are using a Windows operating system, here you would use PuTTY to log in. Then request an interactive session as described above with `qsub`. Unzip the test code:

```
tar xvzf cpp_fortran_test.tgz
```

Change to the test code directory:

```
cd cpp_fortran_test
```

Compile the test code using the "Make" utility that comes with every Linux distribution:

```
make
```

Run the code:

```
./test.exec
```

If you can get this far, and you get the "welcome" messages from C++ and Fortran, you should be able to compile the CFD code we will be using this semester.

B `TriSolve2d` Code Structure

In this class your assignment will be to complete certain key subroutines of the CFD code `TriSolve2d`. Upon completion this code will be have the following capabilities:

- 2D flow
- compressible
- subsonic, transonic, supersonic regimes
- viscous flow
- inviscid flow
- low Mach preconditioning
- explicit solution procedure
- agglomeration multigrid

In this section I will describe how the code is organized to perform these capabilites

B.1 Mesh Management versus Mesh Generation

The flow solver, `TriSolve2d`, depends on a corresponding mesh manager, `TriMesh2d`. "Mesh management" is a concept I like to use in place of "mesh generation," shown in Figure 18 Traditionally, mesh generators are a separate code altogether, which separates mesh creation

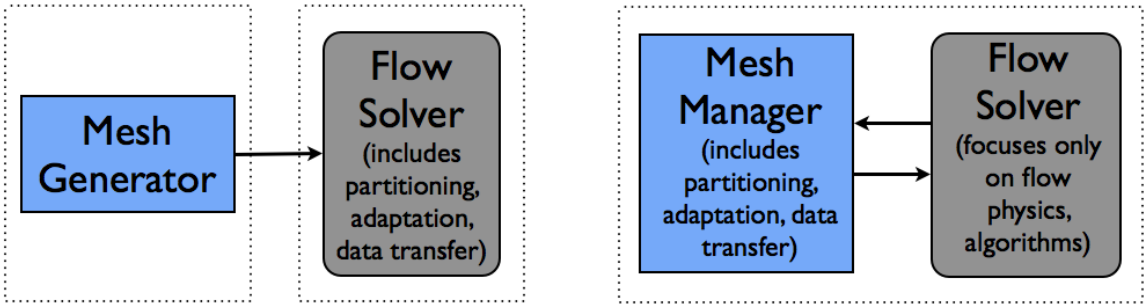

Figure 18: Mesh management versus mesh generation.

and adaptation from the solution process. This places the burden for meshing tasks, such as partitioning, adaptation, and data transfer between partitions within the solver, greatly increasing solver complexity. With mesh management, all the meshing tasks are delegated to a mesh manager, allowing the solver to focus on algorithms and physics. The mesh manager routines are available at run time, allowing for dynamic mesh adaptation and repartitioning. In my research, I am interested in mesh types that can also be generated easily in the mesh

manager and do not require and external mesh generator. However, currently `TriMesh2d` doesn't actually generate the mesh. The meshes I have used in this class are either hand made (for simple cases), or generated by the free software called `Delaundo`, which you can Google and find online.

In this class, you will not deal with `TriMesh2d`, but only with the flow solver itself, `TriSolve2d`. All I will say about `TriMesh2d` is that it performs the following tasks currently:

- reads a mesh file in *.vtk format
- stores the global mesh definition
- partitions the mesh for parallel processing
- performs automatic coarsening for multigrid
- provides communication between parallel partitions
- manages the parallel solution process

If you are interested in the details of `TriMesh2d`, you will have a copy, so feel free to look around.

As the last bullet point above indicates, `TriMesh2d` manages the solution process, since it knows about all partitions. On the other hand, `TriSolve2d` is a bit "naive" since it knows only how to solve for the flow on a single partition, or `TriBlock`. Think of a `TriBlock` as the minimum amount of mesh data needed to describe a single partition of unstructured triangular mesh. Each `TriBlock`, which is a class of `TriMesh2d`, has a corresponding `TriBlockSolver`, which is a class of `TriSolve2d`. This class is located in the file `TriSolve2d/src/interface/TriBlockSolver.C`. This file is the only file in `TriSolve2d` that is written in C++. All other routines are written in Fortran. When `TriMesh2d` manages the parallel solution process, it communicates with the block solvers exclusively through the routines in `TriBlockSolver.C`. The functions of `TriBlockSolver.C` include:

- `input` - reads the solver input file
- `initialize` - initializes the computation
- `shiftTime` - advances in physical time for unsteady runs
- `preparePseudoStep` - prepares to take a pseudo time step
- `cleanUpPseudoStep` - cleans up after taking a pseudo time step
- `output` - generates output for plotting the solution
- `computeRHS` - computes the right hand side residual
- `solve` - updates the solution by taking a psuedo time step
- `restrictQ` - restricts the solution to a coarse multigrid mesh
- `restrictR` - restricts the residual to a coarse multigrid mesh

- `prolong` - prolongs the multigrid corrections to a fine multigrid mesh
- `finalize` - performs final clean up tasks

You should not have to touch anything in `TriBlockSolver.C` for this class, but you should know what it does.

B.2 Solver Layers

If you look at the contents of `TriBlockSolver.C`, you will notice that it calls a number of Fortran subroutines that have the word "Control" somewhere in their name. In fact, "Control" is one of four "layers" into which `TriSolve2d` is organized. The four layers are as follows:

- CONTROL - provides routines called by the block solver
- TIME - responsible for the steady or unsteady solution process
- GEOM - responsible for the spatial discretization
- SYSTEM - responsible for the physical system to be solved

In research, my goal is to make the layers interchangeable. For example if I write a code for 2d triangular meshes, and I would like to write a code for 2d Cartesian meshes that solve the same physical system using the same solution techniqes, I should only have to change the GEOM layer. So organizing the code into modular layers allows me to reuse code for various projects, and among various graduate students. The layer to which a subroutine belongs is always indicated in its name, such as `rhsGEOMResidual`.

B.3 Source Code Directories

In addition to containing its layer, each subroutine name also contains a directory name (the `rhs` in `rhsGEOMResidual`). When designing the code, I divided the subroutines into a number of directories based on their functionality. This directory name is appended to the begining of the subroutine name. All these directories are located in the `src/` directory of `TriSolve2d`. These directories include

- `bc` - boundary conditions
- `gmg` - geometric multigrid
- `grid` - grid setup tasks
- `include` - contains include files
- `init` - initialization routines
- `input` - code input routines
- `interface` - contains the interface file to the meshing layer

- `lhs` - computes the left hand side
- `main` - contains a `main.C` driver program
- `output` - outputs the solution for plotting
- `prep` - prepares the solution process
- `rhs` - computes the right hand side residual
- `shared` - contains shared data in Fortran modules
- `step` - performs pseudo and physical time steps
- `utilites` - contains a number of low level routines

Just by examining a subroutine name, you should be able to quickly locate the routine and know to which layer it belongs.

B.4 Data Structures

Data structures in `TriSolve2d` are divided into TIME, GEOM, and SYSTEM layers, although certain data elements are available to multiple layers. Sometimes, the GEOM layer needs to know how many equations are to be solved (there are four in 2d: mass, x-momentum, y-momentum, and total energy), which is a SYSTEM layer quantity, for example. Certain data in each layer come from the solver input file, and is stored in globally available Fortran modules in the `src\shared` directory. Below is a listing of the important data available in each layer and a short description where needed.

B.4.1 TIME layer data

Input data from `src\shared\sharedTimeInput.F90`:

- `iconvFile` - (int) 1 writes convergence history file, 0 does not
- `restartstep` - (int) step to restart from (restart capability not available yet)
- `nrestart` - (int) interval of unsteady time step to write restart files (restart capability not available yet)
- `noutput` - (int) interval of unsteady time step to write output files
- `nsteps` - (int) number of unsteady time steps
- `npseudosteps` - (int) number of pseudo time steps
- `npseudosteps0` - (int) number of pseudo time steps at the 0th unsteady step or for steady state problems
- `nrkstages` - (int) number of Runge Kutta Stages (choose 5 always)

- `nlevels` - (int) number of multigrid levels (1 for no multigrid)
- `mgcycle` - (int) type of multigrid cycle, 1 for v-cycle, 2 for w-cycle
- `precon` - (int) 1 for preconditioning, 0 for no preconditioning (available in the GEOM layer)
- `dtunsteady` - (real) unsteady time step in seconds (available in the GEOM layer)
- `cfl` - cfl (real) number for pseudo time step (available in the GEOM layer)
- `vnn` - (real) von Neumann number for pseudo time step
- `convlimit` - (real) convergence limit, tells the solver when to consider the equations converged
- `relax` - (real) multigrid relaxation parameter (available in the GEOM layer)
- `smooth` - (real) implicit residual smoothing parameter (available in the GEOM layer)
- `rka(0:nrkstages)` - (real) Runge Kutta fractional time step values (available in the GEOM layer)
- `rkb(0:nrkstages)` - (real) Runge Kutta diffusive term factor (available in the GEOM layer)

Other data:

- `nd` - (int) number of unknowns in the grid
- `ib(nd)` - (int) iblank array that freezes ghost nodes (available in the GEOM layer)
- `rms(nq)` - (real) root mean square of the residual, monitors convergence
- `q(nq,nd)` - (real) vector of conserved variables at current physical time step (available in the GEOM layer)
- `r(nq,nd)` - (real) residual vector (available in the GEOM layer)
- `s(nq,nd)` - (real) MMS source term vector (available in the GEOM layer)
- `d(nq,nd)` - (real) diffusive flux vector (available in the GEOM layer)
- `dn(nq,nd)` - (real) diffusive flux vector from previous RK stage (available in the GEOM layer)
- `v(nd)` - (real) cell volume at current physical time step (available in the GEOM layer)
- `v1(nd)` - (real) cell volume at nth time step
- `v2(nd)` - (real) cell volume at (n-1)th time step

- `q1(nq,nd)` - (real) vector of conserved variables at nth physical time step
- `q2(nq,nd)` - (real) vector of conserved variables at (n-1)th physical time step
- `dt(nd)` - (real) local psuedo time step (available in the GEOM layer)
- `qn(nq,nd)` - (real) vector of conserved variables at beginning of RK pseduo step (available in the GEOM layer)
- `radi(nd)` - (real) inviscid spectral radius (available in the GEOM layer)
- `radv(nd)` - (real) viscous spectral radius (available in the GEOM layer)
- `qa(nqa,nd)` - (real) vector of additional variables (available in the GEOM layer)
- `q0(nq,nd)` - (real) vector of conserved variables at first descent into a coarse multigrid level (available in the GEOM layer)
- `fwc(nq,nd)` - (real) agglomerated fine level residuals (available in the GEOM layer)
- `pc(nqp,nd)` - (real) preconditioning matrix terms (available in the GEOM layer)

B.4.2 GEOM layer data

Input data from `src\shared\sharedGeomInput.F90`:

- `isolnFile` - (int) 1 writes solution file, 0 does not
- `iresdFile` - (int) 1 writes residual file, 0 does not
- `ierrFile` - (int) 1 writes error file, 0 does not (form MMS)
- `isurfFile` - (int) 1 writes surface variables file, 0 does not
- `standAlone` - (int) 1 to run the code stand alone, 0 as part of a larger grid infrastructure (always choose 1)
- `gradient` -(int) 0 for no gradient, 1 for linear least squares, 2 for quadratic least squares, 3 for green-Gauss (always choose 1)
- `flux` - (int) 1 for Galerkin central scheme, 2 for corrected scheme (always choose 1)
- `perturb` - (int) 1 for random nodal perturbation, 0 for no perturbation (always choose 0)
- `coarsedis` - (real) multigrid coarse level dissipation constant

Other data:

- `nNode` - (int) number of nodes
- `nBnode` - (int) number of boundary nodes

- `nGnode` - (int) number of ghost nodes
- `nTri` - (int) number of triangles
- `nGtri` - (int) number of ghost triangles
- `nEdge` - (int) number of edges (including boundary edges)
- `nBedge` - (int) number of boundary edges
- `nPsp1` - (int) dimension of psp1
- `nBcomp` - (int) number of boundary components
- `nNodeP` - (int) number of nodes on the parent multigrid level
- `nGnodeP` - (int) number of ghost nodes on the parent multigrid level
- `nBnodeP` - (int) number of boundary nodes on the parent multigrid level
- `nEdgeP` - (int) number of edges on the parent multigrid level
- `nBedgeP` - (int) number of boundary edges on the parent multigrid level
- `qxStatus` - (int) status of gradient computation, 0 means gradients are not computed, 1 means gradients have already been computed
- `limStatus` - (int) status of limiter computation, 0 means limiters are not computed, 1 means limiters have already been computed
- `tri(3,nTri)` - (int) three node numbers for each triangle inc counter clock-wise order
- `bEdge(2,nBedge)` - (int) on fine levels the two node numbers that make a boundary edge (boundary is on the right when walking from the first node to the second), on coarse levels, the first node is the interior node and the second is a ghost node
- `bTag(nBedge)` -(int) boundary component tag for each boundary edge
- `x(2,nNode)` - (real) x,y coordinates of each node for the current physical time
- `qx(nq,2,nNode)` - (real) gradient of conserved variables
- `qax(nqa,2,nNode)` - (real) gradient of additional variables
- `x0(2,nNode)` - (real) initial x,y coordinates (used for moving meshes)
- `x1(2,nNode)` - (real) x,y coordinates at nth time level
- `x2(2,nNode)` - (real) x,y coordinates at (n-1)th time level
- `psp1(npsp1)` - (int) points surrounding points container array
- `psp2(nNode+1)` - (int) points surrounding points pointer array

- `edge(2,nEdge)` - (int) the two nodes seprated by each edge
- `edgp(2,nEdge)` - (int) on fine levels, the edge "extension" nodes used for limiting
- `edgn(2,nEdge)` - (int) on fine levels, the nodes forming triangles around each edge
- `bnod(2,nBnode)` - (int) on fine levels, a list of boundary nodes and their component tags
- `edgs(2,nEdge)` - (real) area weighted normals for each edge
- `nxb(2,nBnode)` - (real) outward unit normal of each boundary node
- `sb(nq,nBnode)` - (real) vector of MMS source terms for each boundary node
- `lm(nq,nBnode)` - (real) vector of Lagrange Multipliers at each boundary node
- `rb(nq,nBnode)` - (real) vector of boundary condition residuals at each boundary node
- `xv(nEdge+nBedge)` - (real) dot product of face velocities with area weighted normals at each edge (on fine levels there are two entries for each facet of the boundary edge)
- `dv(nEdge)` - (real) change in volume swept by each edge between time levels n-1 and n.
- `bv(2,nBedge)` - (real) velocity components of each boundary edge
- `nv(2,nBnode)` - (real) velocity components of each boundary node
- `ls(2,npsp1)` - (real) x,y least squares gradient weights
- `lim(nEdge)` - (real) limiter for each edge
- `f2cp(nNodeP)` - (int) for each fine level cell on the parent level, the coarse level cell to which it agglomerates
- `f2ce(nEdgeP)` - (int) for each fine level edge on the parent level, the coarse level edge to which it agglomerates (sign indicates the sense of the edge)
- `edgeP(nEdgeP)` - (int) edges on the parent level
- `edgsP(2,nEdgeP)` - (real) face areas on the parent level
- `xvP(nEdgeP)` - (real) dot product of face velocities with area weighted normals on the parent level
- `bvP(2,nBedgeP)` - (real) boundary edge velocities on the parent level
- `qP(nq,nNodeP)` - (real) vector of conserved variables on the parent level
- `rP(nq,nNodeP)` - (real) residual vector on the parent level
- `vP(nNodeP)` - (real) node/cell volumes on the parent level

B.4.3 SYTSEM layer data

Input data from `src\shared\sharedSystemInput.F90`:

- `bcfile` - (char) name of boundary condition file
- `exactsoln` - (int) 0 for freestream initialization, 1 for MMS solution initialization, 2 for perturbed freestream initialization
- `pitch` - (int) for unsteady cases, 0 for no pitching, 1 for 64A010 pitching case (available in the GEOM layer)
- `inviscid` - (int) 1 to add inviscid terms, 0 to not (available in the GEOM and TIME layers)
- `dissipation` - (int) 1 to add dissipation terms, 0 to not (available in the GEOM and TIME layers)
- `viscous` - (int) 1 to add viscous terms, 0 to not (available in the GEOM and TIME layers)
- `source` - (int) 1 to add MMS source terms, 0 to not (available in the GEOM and TIME layers)
- `m0` - (real) freestream Mach number
- `aoa` - (real) angle of attack
- `rho0` - (real) freestream density
- `p0` - (real) freestream pressure
- `Re` - (real) Reynolds number
- `ReRefLength` - (real) length scale used to compute Reynolds number (available in the GEOM layer)
- `rgas` - (real) ideal gas constant
- `gamma` - (real) ratio of specific heats
- `prn` - (real) Prandtl number
- `nq` - (int) number of conservation equations to solve (available in the GEOM and TIME layers)
- `nqa` - (int) number of additional variables to compute (available in the GEOM and TIME layers)
- `nqp` - (int) number of entries in the preconditioning matrix to store (available in the GEOM and TIME layers)

- `nSurf` - (int) number of nodes on solid surface boundaries
- `c0` - (real) freestream speed of sound
- `q0` - (real) freestream velocity magnitude
- `u0` - (real) freestream x-velocity
- `v0` - (real) freestream y-velocity
- `e0` - (real) freestream total energy per unit mass
- `s0` - (real) freestream entropy
- `h0` - (real) freestream total enthalpy per unit mass
- `t0` - (real) freestream temperature
- `mu0` - (real) freestream viscosity
- `gm1` - (real) gamma-1.
- `gmm` - (real) .5*(gamma+1.)/gamma
- `gmg` - (real) gm1/gamma
- `lift` - (real) lift coefficient
- `drag` - (real) drag coefficient
- `entr` - (real) RMS entropy at the `nSurf` surface nodes

Other data:

- `nBpatches` - (int) number of surface patches (available in the GEOM layer)
- `bType(nBpatches)` - (int) boundary type for each surface patch:
  1 = inviscidWall
  2 = viscousWall
  3 = inflow
  4 = outflow
  5 = farField
  6 = dirichlet
  7 = nothing
  (available in the GEOM layer)
- `bValue(nq,nBpatches)` - (real) vector of primitive variables (p,u,v,T) specified for each boundary patch (available in the GEOM layer)

B.5 Edge data structure

The edge-based data structure for fine multigrid levels is shown in Figure 19. The `edge` array contains the two nodes forming the edge. The `edgn` array contains the opposing nodes on the corresponding triangles. The `edgp` array contains the edge extension nodes that are used in the limiting procedure. If an edge impinges on the boundary, the `edgn` entry for that edge is 0, as shown in Figure 20.

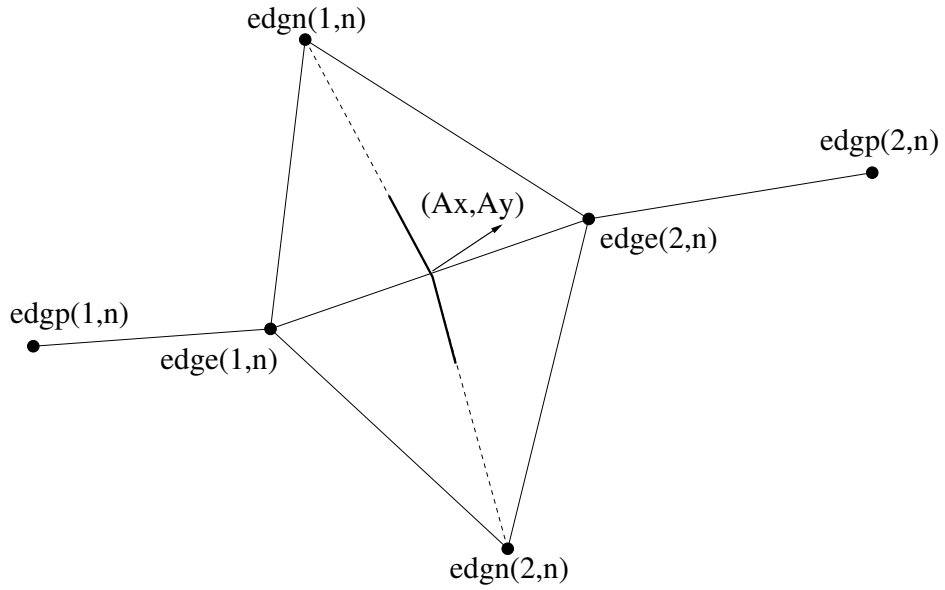

Figure 19: Interior edge-based data structures.

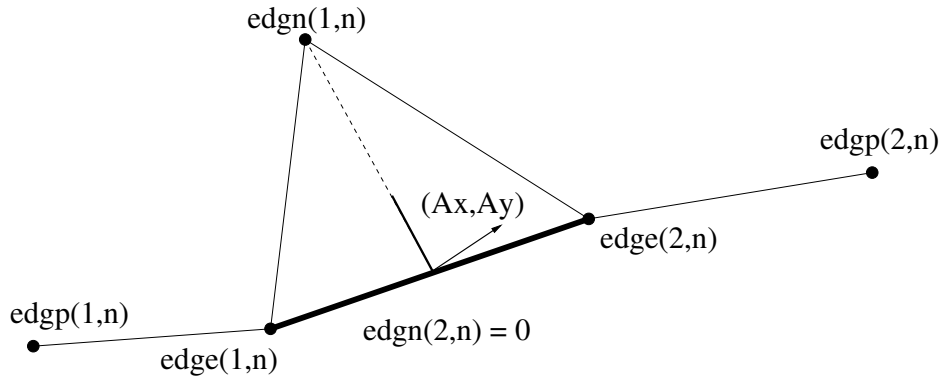

Figure 20: Edge-based data structures impinging on the boundary

The last `nBedge` entries in the edge data structures contain the boundary edges themselves. The boundary edges and their area vectors are shown in Figure 21. Note that both entries of *edgn* are 0, and the area vector lies along the boundary edge itself and must be stored as half its total area, since half will go to each node comprising the edge.

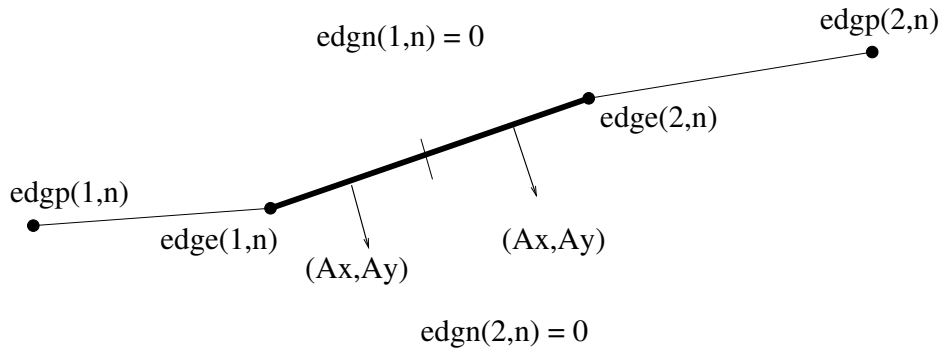

Figure 21: Edge-based data structures along the boundary

B.6 Points Surrounding Points

For certain operations in the code, such as least squares gradient computation, it is helpful to know all the points surrounding a given point. This information is contained in the two arrays `psp1` and `psp2`. To access all the points surrounding node n, for example, one would write

```
DO m=psp2(n)+1,psp2(n+1)
   i = psp1(m)
   WRITE(*,*)'node ',i,' is one of the nodes surrounding node ',n
END DO
```

If you print this output in your code you will notice that the first point surrounding the point n, is n itself. That is,

```
m = psp2(n)+1
WRITE(*,*)n,psp1(m)
```

are the same number. By defining the `psp` arrays in this general way, we can add or take away points easily, and we are not restricted to nearest neighbors.

B.7 Boundary Condition Specification

Both the GEOM layer and the SYSTEM layer play a role in specifying boundary conditions. The GEOM layer is responsible for keeping track of *boundary patches*. A boundary patch is a group of boundary edges for which a boundary condition may be applied. For example, if we run a simulation with two airfoils and an outer far field boundary, airfoil one might be patch 1, airfoil two might be patch 2, and the far field boundary might be patch 3. Each of the `nBedge` boundary edges is assigned a boundary patch tag in the $bTag$ array. It is also useful to tag each of the `nBnode` boundary nodes, which are listed in the first slot of the `bnod` array. Patch tags for the boundary nodes are stored in the second slot of the `bnod` array. So if I print out

```
DO n=1,nBnode
   WRITE(*,*)bnod(1,n),bnod(2,n)
END DO
```

I will end up with a list of nodes which lie on the domain boundaries along with their patch tags. The patch tags for the boundary nodes are the same as for the boundary edges, except for the case in which a boundary node connects two boundary edges of different patches. In those cases, the boundary patch is assigned arbitrarily to one or the other edge patches.

The SYSTEM layer is responsible for assigning each boundary patch a *boundary type* and a set of possible *boundary values*. The boundary type for each of the `nBPatches` boundary patches is storeed in `bType` and follows this numbering convention:

1 = inviscidWall
2 = viscousWall
3 = inflow
4 = outflow
5 = farField
6 = dirichlet
7 = nothing

The "nothing" condition means no boundary treatment is done. This comes in handy at sharp corners, such as the trailing edge of an airfoil, where it is better to do nothing. In addition, each boundary patch may require specified flow values in the case that certain Dirichlet conditions are applied. The `bValue` array stores the vector of pressure, velocities, and temperature for each boundary patch.

Exercises

1. Compile `TriMesh2d` and `TriSolve2d` on your local workstation without errors. To do this, adjust the `Make.rules` for each code according to your system. First, compile `TriMesh2d` by typing "make library". This will create a library to which you can link the solver. Then go to the directory for `TriSolve2d`, and type "make driver". This will compile the solver routines and link them with the meshing layer to the driver file in `src/main/main.C`. An executable will be placed in the `run` directory.

References

[1] S. Allmaras. Lagrange multiplier implementation of Dirichlet boundary conditions in compressible Navier-Stokes finite element methods. *AIAA paper* 2005-4714, AIAA 17th Computatonal Fluid Dynamics Conference, Toronto, June 2005.

[2] B. Despres. Lax theorem and finite volume schemes. *Mathematics of Computation*, 73:1203–1234, 2003.

[3] M. Giles. Accuracy of node-based solutions on irregular meshes. *Lecture Notes in Physics*, 323:273–277, 1989.

[4] P. Kundu and I. Cohen. *Fluid Mechanics, 4th ed.* Burlington, MA, 2008.

[5] R. Martineau, R. Berry, A. Esteve, K. Hamman, D. Knoll, R. Park, and W. Taitano. Comparative analysis of natural convection flows simulated by both the conservation and incompressible forms of hte Navier-Stokes equations in a differentially-heated square cavity. Technical Report INL/EXT-09-15333, Idaho National Laboratory, 2009.

[6] L. Martinelli. Calculations of viscous flow with a multigrid method. Technical report, Princeton Univerisity, 1987.

[7] H. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Essex, England, 2007.

[8] B. Wendroff. Supraconvergence in two dimensions. Technical Report LA-UR-95-3068, Los Alamos, 1995.