Advanced Computational Fluid Dynamics

Aero 623 Course Notes: Winter 2026

Krzysztof Fidkowski
University of Michigan, Ann Arbor, MI, 48109, USA

January 8, 2026

Contents

Chapter 1

Introduction and Review

1.1 Notation

In this text we will use the following notation:

c : (regular font) a **scalar**

$\vec{v}$: (arrow) a **spatial vector**, e.g.

$$\vec{v} = v_1\hat{x}_1 + v_2\hat{x}_2 + v_3\hat{x}_3,$$

where $\hat{x}_i$ is the unit vector in the i^{th} Cartesian direction.

$\mathbf{u}$: (bold) a **state vector**, e.g.

$$\mathbf{u} = [\rho, \rho u, \rho v, \rho E]^T.$$

$\vec{\mathbf{F}}$: (bold and arrow) a **combined spatial and state vector**, e.g.

$$\vec{\mathbf{F}} = \mathbf{F}_1\hat{x}_1 + \mathbf{F}_2\hat{x}_2 + \mathbf{F}_3\hat{x}_3.$$

$\mathbf{A}$: (bold, typically uppercase) a **matrix**, e.g.

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}.$$

We will use index notation, in which v_i denotes the i^{th} component of a vector $\mathbf{v}$ (spatial or state). A repeated index in an expression implies summation, e.g.

$$v_i w_i = \sum_i v_i w_i = \mathbf{v} \cdot \mathbf{w}.$$

∂_i is used to denote differentiation with respect to x_i. So the **gradient** of a scalar field p is the vector

$$\nabla p \;=\; \frac{\partial p}{\partial x_1}\hat{x}_1 + \frac{\partial p}{\partial x_2}\hat{x}_2 + \frac{\partial p}{\partial x_3}\hat{x}_3 = \partial_i p\, \hat{x}_i$$

The **divergence** of a vector field $\vec{v}$ is the scalar $\nabla \cdot \vec{v} = \partial_i v_i$.

When a rank n vector $\mathbf{f}$ is a function of a rank m vector $\mathbf{u}$, the **Jacobian** of $\mathbf{f}$ with respect to $\mathbf{u}$ is an $n \times m$ matrix of partial derivatives,

$$\frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \cdots & \frac{\partial f_1}{\partial u_m} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \cdots & \frac{\partial f_2}{\partial u_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \cdots & \frac{\partial f_n}{\partial u_m} \end{bmatrix}. \tag{1.1.1}$$

1.2 Linear Systems

Consider an $N \times N$ linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1.2.1}$$

where $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a known matrix, $\mathbf{b} \in \mathbb{R}^N$ is a known right-hand side, and $\mathbf{x} \in \mathbb{R}^N$ is the vector of unknowns.

1.2.1 PLU Factorization

For relatively small, dense matrices, we can use Gaussian elimination/PLU factorization. This is a systematic elimination of variables that can be used to solve Equation 1.2.1, or to factor the matrix $\mathbf{A}$ for use in later solutions. We focus on a factorization as it costs the same but is more general: it can be done in place and it allows for quick application of both $\mathbf{A}^{-1}$ and $\mathbf{A}$. The factorization of interest is

$$\mathbf{A} = \mathbf{P}^{-1}\mathbf{L}\mathbf{U}, \tag{1.2.2}$$

where

- $\mathbf{P}$ is a permutation matrix with a single unity entry in every row and column; note, $\mathbf{P}^{-1} = \mathbf{P}^T$,
- $\mathbf{L}$ is a lower-triangular matrix with ones on the main diagonal and zeros above the main diagonal,
- $\mathbf{U}$ is an upper-triangular matrix with zeros below the main diagonal.

For example,

$$\underbrace{\begin{bmatrix} 4 & 5 & 8 & 9 \\ 6 & 9 & 21 & 19 \\ 2 & 1 & 3 & 4 \\ 4 & 11 & 16 & 12 \end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{P}^{-1}} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 2 & 3 & 1 & 0 \\ 3 & 2 & 2 & 1 \end{bmatrix}}_{\mathbf{L}} \underbrace{\begin{bmatrix} 2 & 1 & 3 & 4 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 3 \end{bmatrix}}_{\mathbf{U}}$$

Storing the matrix $\mathbf{P}$ is not necessary, and in practice a vector $\mathbf{p} \in \mathbb{R}^N$ is stored containing the column index of the "1" in each row of $\mathbf{P}$. The algorithm with in-place storage, is:

Algorithm 1 Pivoted Lower-Upper (PLU) factorization

1: Initialize $\mathbf{p} = [1 : N]$
2: **for** $k = 1 : (N-1)$ **do**
3: $\quad k_{\max} =$ row number of maximum magnitude entry at/below $A(k,k)$
4: $\quad$ if $k_{\max} \neq k$, swap$[A(k,:), A(k_{\max},:)]$ and swap$[p(k), p(k_{\max})]$
5: $\quad$ Define $\mathbf{I} = [(k+1) : N]$
6: $\quad A(\mathbf{I},k) = A(\mathbf{I},k)/A(k,k)$
7: $\quad A(\mathbf{I},\mathbf{I}) = A(\mathbf{I},\mathbf{I}) - A(\mathbf{I},k)A(k,\mathbf{I})$
8: **end for**

Following this algorithm, the upper-triangular portion of $\mathbf{A}$, including the diagonal, will be $\mathbf{U}$, while the lower-triangular portion of $\mathbf{A}$, not including the diagonal, will be $\mathbf{L}$ without the trivial 1's on the main diagonal. Due to step 7, this algorithm scales as N^3.

Solving the system Equation 1.2.1 or applying the matrix consists of order N^2 operations, as shown in Algorithms 2 and 3.

Algorithm 2 Solving $\mathbf{x} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P}\mathbf{b}$

1: First, solve $\mathbf{L}\mathbf{y} = \mathbf{P}\mathbf{b}$:
2: **for** $k = 1 : N$ **do**
3: $\quad y(k) = b(p(k)) - \sum_{j=1}^{k-1} L(k,j)y(j)$
4: **end for**
5: Second, solve $\mathbf{U}\mathbf{x} = \mathbf{y}$
6: **for** $k = N : -1 : 1$ **do**
7: $\quad x(k) = \left(y(k) - \sum_{j=k+1}^{N} U(k,j)x(j)\right)/U(k,k)$
8: **end for**

Algorithm 3 Applying $\mathbf{b} = \mathbf{P}^{-1}\mathbf{L}\mathbf{U}\mathbf{x}$

1: First, set $\mathbf{y} = \mathbf{U}\mathbf{x}$:
2: **for** $k = 1 : N$ **do**
3: $\quad y(k) = \sum_{j=k}^{N} U(k,j)x(j)$
4: **end for**
5: Second, set $\mathbf{b} = \mathbf{P}^{-1}\mathbf{L}\mathbf{y}$
6: **for** $k = N : -1 : 1$ **do**
7: $\quad b(p(k)) = y(k) + \sum_{j=1}^{k-1} L(k,j)y(j)$
8: **end for**

Many scripted programming languages come with built-in functions for computing and

applying the PLU factorization. If coding in a compiled language, the Linear Algebra PACKage (LAPACK) is useful http://www.netlib.org/lapack/.

1.2.2 Iterative smoothers

For large, sparse matrices, the PLU factorization becomes prohibitively expensive. Smoothers are methods that require repeated application to drive (often slowly) an initial solution guess, $\mathbf{x}^0$, to a solution with an acceptable level of error via the sequence $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n, \ldots$.

Suppose that our matrix can be separated into $\mathbf{A} = \mathbf{M} - \mathbf{N}$, where $\mathbf{M}$ is easily invertible. In solving Equation 1.2.1 for $\mathbf{x}^{n+1}$, we approximate the left-hand-side as

$$\mathbf{A}\mathbf{x}^{n+1} = (\mathbf{M} - \mathbf{N})\mathbf{x}^{n+1} \approx \mathbf{M}\mathbf{x}^{n+1} - \mathbf{N}\mathbf{x}^n.$$

Setting the above equal to $\mathbf{b}$ and solving for $\mathbf{x}^{n+1}$ yields the following iterative update step,

$$\mathbf{x}^{n+1} = \mathbf{M}^{-1}(\mathbf{b} + \mathbf{N}\mathbf{x}^n). \tag{1.2.3}$$

Sometimes only the action of $\mathbf{M}^{-1}$ is available, so that $\mathbf{N}$ cannot be easily defined. Substituting $\mathbf{N} = \mathbf{M} - \mathbf{A}$ in Equation 1.2.3 yields the more general expression,

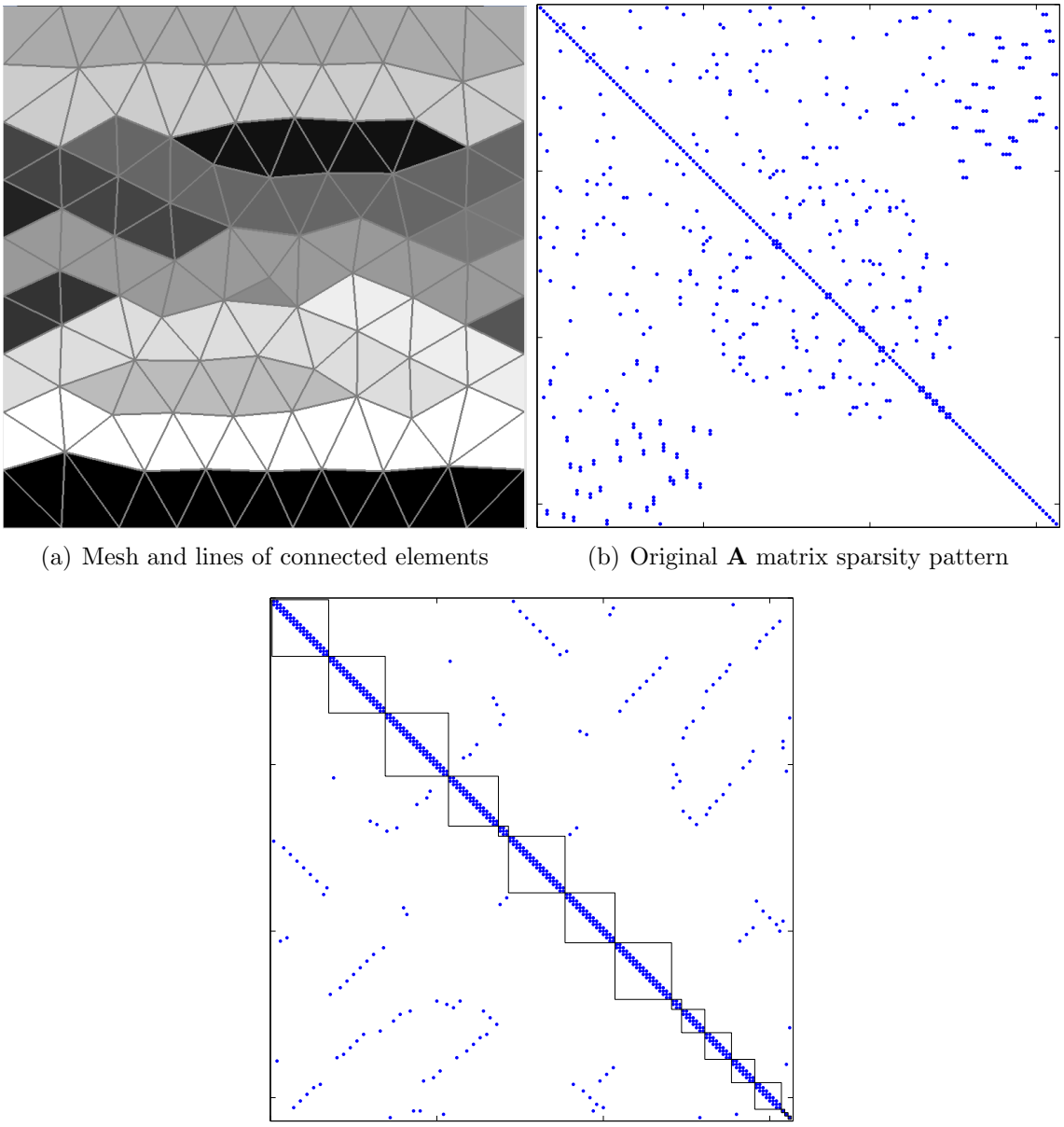
$$\mathbf{x}^{n+1} = \mathbf{x}^n - \mathbf{M}^{-1}\mathbf{r}(\mathbf{x}^n), \qquad \mathbf{r}(\mathbf{x}^n) \equiv \mathbf{A}\mathbf{x}^n - \mathbf{b}. \tag{1.2.4}$$

$\mathbf{M}$ is called a *preconditioner*: it is an approximation to $\mathbf{A}$ that is easy to invert.

A popular choice for $\mathbf{M}$ in CFD discretizations is the block-diagonal of $\mathbf{A}$, where the blocks correspond to all unknowns associated with a single node, cell, or element. The resulting method goes by the names "point", "block", or "element-block" implicit smoothing. It is simple, trivially-parallelizable, but slow to converge.

A more powerful smoother is obtained by extending $\mathbf{M}$ to include *lines* of elements. $\mathbf{M}$ is still easy to invert because it consists of block-tridiagonal systems. Figure 1.2.1 illustrates an example of the $\mathbf{A}$-matrix sparsity pattern after elements are re-ordered according to lines. Line smoothers are used to accelerate convergence of CFD cases with tightly-coupled elements. These occur primarily in the flow-normal direction for highly-anisotropic boundary-layer meshes (e.g. Reynolds-Averaged Navier-Stokes simulations). However line smoothers are also used to reduce stiffness in the flow direction for convection-dominated flows.

Stability and performance of smoothers are analyzed via Fourier methods: an amplification factor is computed for each representable mode on a simple mesh with periodic boundary conditions. We note that for the discontinuous Galerkin finite-element method (to be discussed soon), both of the above smoothers are stable for any order. Finally, we note that smoothers are generally not used in isolation. Rather, they are combined with Krylov or multigrid methods.



(a) Mesh and lines of connected elements

(b) Original **A** matrix sparsity pattern

(c) **A** with elements reordered along lines

Figure 1.2.1: Example of a line-based smoother.

1.2.3 Krylov subspace methods

Preconditioned Krylov subspace methods are among the most powerful techniques for solving large linear systems arising from implicit CFD discretizations. These methods seek the best solution in the vector space

$$\mathcal{K}^n = \text{span}\left\{\mathbf{M}^{-1}\mathbf{b},\, \mathbf{M}^{-1}\mathbf{A}\mathbf{M}^{-1}\mathbf{b},\, (\mathbf{M}^{-1}\mathbf{A})^2\mathbf{M}^{-1}\mathbf{b},\, \ldots,\, (\mathbf{M}^{-1}\mathbf{A})^n\mathbf{M}^{-1}\mathbf{b}\right\}. \tag{1.2.5}$$

$\mathbf{M}$ is the preconditioner matrix, a tractably-invertible approximation to $\mathbf{A}$. $\mathbf{M} = \mathbf{I}$ corresponds to no preconditioner, while $\mathbf{M} = \mathbf{A}$ is the extreme case for which the exact solution lies in $\mathcal{K}^0$.

A popular Krylov subspace method in CFD is the generalized minimal residual method, GMRES, which picks the vector from $\mathcal{K}^n$, call it $\mathbf{u}_*$, that minimizes the L_2 norm of the residual, $||\mathbf{f} - \mathbf{A}\mathbf{u}_*||$. Conceptually, we could just form the $n+1$ vectors in Equation 1.2.5 and solve a least-squares minimization problem. In practice, such an approach would be extremely ill-conditioned, which means sensitive to round-off errors. Instead, we use a variant that employs Gram-Schmidt orthonormalization based on the Arnoldi iteration. A version of this algorithm that uses up to n_{outer} restarts is given in Algorithm 4. In this algorithm,

Algorithm 4 The generalized minimal residual method (GMRES).

$\mathbf{u} = 0$
for $i = 1 : n_{\text{outer}}$ **do**
  $\mathbf{v}_1 = \mathbf{A}\mathbf{u} - \mathbf{b}$, $\beta = ||\mathbf{v}_1||$, $\mathbf{v}_1 = \mathbf{v}_1/\beta$
  $\mathbf{g} = [\beta, 0, \ldots, 0] \in \mathbb{R}^{n+1}$
  for $j = 1 : n$ **do**
    $\mathbf{w} = \mathbf{A}\mathbf{v}_j$
    for $k = 1 : j$ **do**
      $H(k, j) = \mathbf{w}^T\mathbf{v}_k$
      $\mathbf{w} = \mathbf{w} - H(k, j)\mathbf{v}_k$
    end for
    $H(j + 1, j) = ||\mathbf{w}||$
    $\mathbf{v}_{j+1} = \mathbf{w}/||\mathbf{w}||$
    for $k = 1 : j - 1$ **do**,
      $H([k, k + 1], j) = \mathbf{G}_k H([k, k + 1], j)$
    end for
    $\mathbf{G}_j = Givens(H(j, j), H(j + 1, j))$
    $H([j, j + 1], j) = \mathbf{G}_j H([j, j + 1], j)$
    $g([j, j + 1]) = \mathbf{G}_j g([j, j + 1])$
    $g(j + 1)$ contains the L_2 residual norm; check for convergence
  end for
  $\mathbf{y} = \mathbf{H}^{-1}\mathbf{g}$, $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n+1}]$, $\mathbf{u} = \mathbf{u} + \mathbf{V}\mathbf{y}$,
end for

$Givens(a, b)$ is a 2×2 counter-clockwise rotation matrix by $\tan^{-1}(b/a)$. The convergence

rate of GMRES depends on the eigenvalue distribution of $\mathbf{A}$, where clustered eigenvalues are good. The largest storage requirement is the $n+1$ vectors $\mathbf{v}_j$.

GMRES is applicable to the asymmetric linear systems typically found in CFD. Another popular Krylov subspace method is the conjugate gradients (CG) method, which is applicable to symmetric systems. A competitive extension of CG for asymmetric systems is biconjugate gradients (BCG) and an improved variant, Bi-CGSTAB. Unlike GMRES, these methods do not minimize the residual norm, but, with a three-term recurrence relation, they obviate the need to store the entire Krylov subspace basis.

For most problems, Krylov methods like GMRES, CG, BCG, etc. need to be preconditioned to obtain satisfactory performance. A **preconditioner** is a pseudo-inverse to the matrix, one that is generally easy to apply. It accelerates the performance of GMRES because the resulting search space of vectors more quickly hones in on the true solution. The better the preconditioner approximates the inverse of $\mathbf{A}$, the faster the convergence.

1.2.4 Preconditioning

Preconditioners accelerate iterative methods via an approximate inverse. Two popular preconditioners for Krylov methods in CFD are the iterative smoothers introduced above: element-block and element-line Jacobi. The name "Jacobi" signifies that calculations for all elements/lines are performed using iteration n data. "Gauss-Seidel" refers to the practice of over-writing data as it is obtained, resulting in less storage and improved convergence (typically a factor of 2). However, Gauss-Seidel is not easily parallelizable, and its performance depends on the order in which the elements are visited.

Element-line smoothing can show significantly-improved performance compared to element-block smoothing, especially for problems with strong anisotropy (preferred directions) in element coupling. However, the power of element-line smoothing degrades with parallelization, because long lines tend to get cut by mesh partitioning.

Related to the element-line Jacobi preconditioner is incomplete LU factorization (ILU), in which not all terms in the $\mathbf{L}$ and $\mathbf{U}$ matrices are kept in the process of LU factorization. The least-expensive and most popular version of this is ILU(0), where the "0" indicates that no additional "fill" is created in the factorization. That is, entries of $\mathbf{A}$ that are zero remain zero following the in-place factorization. This is accomplished by, during factorization, discarding (setting to zero) any terms that would be placed in these locations. For systems or high-order discretizations, discarding is done in a block fashion, where the blocks contain degrees of freedom associated with an element. The performance of ILU depends on the ordering of the elements, which dictates how many terms are discarded during the factorization. Various heuristic but inexpensive algorithms exist for ordering elements to attain this "minimum discarded fill" property.

Note that any preconditioner can also be used as a stand-alone iterative solver. The result is a division of labor where the preconditioner does not need to address all of the error modes in the problem. The choice of which preconditioner to use for a particular Krylov-subspace method depends on various factors, including the equations solved, the order of the discretization, properties of the mesh, and available memory. The preconditioners

discussed above are popular for CFD, but there are certainly others, e.g. multigrid and defect correction. Scientific packages, such as PETSc [8], support a variety of other preconditioners.

1.3 Numerical Integration

The idea in numerical integration is to approximate integrals by weighted sums of point values. It turns out that on many domains we can integrate polynomial functions up to a certain order, p, exactly, where the order depends on how many points we use.

Presently we restrict our attention to one dimension and work on a fixed domain, $[-1, 1]$. Other intervals can be mapped to this domain. We seek one set of N points, x_q, and weights, w_q, such that for *arbitrary* polynomials $f(x)$ of order p,

$$\int_{-1}^{1} f(x)dx = \sum_{q=1}^{N} w_q f(x_q). \tag{1.3.1}$$

One approach is to sample $f(x)$ at $N = p + 1$ *arbitrary* distinct points x_q, fit an order p polynomial, and integrate this polynomial. This procedure leads to a result that we can express as a weighted linear combination of $f(x_q)$, and the weights are the w_q that we seek. The problem with this approach, however, is that with a careless choice of x_q, some of the weights may turn out to be very large, creating conditioning and numerical precision problems. We can do much better by choosing x_q judiciously.

The key observation behind more powerful numerical integration techniques is that integration turns a function that is a distribution in x into a scalar, the integrand. This loss of information suggests that less information about $f(x)$ is needed to compute the integral than to fully describe the function.

For example, suppose we wanted to integrate an arbitrary linear function on $x \in [-1, 1]$. To do this, we only need one point, since

$$\int_{-1}^{1} f_{\text{linear}}(x)dx = 2f_{\text{linear}}(0) \quad \Rightarrow N = 1, x_1 = 0, w_1 = 2.$$

Figure 1.3.1 shows a graphical proof of this. Note that we would need two points to determine f_{linear}. The key idea that enables accurate integration with few points is that we can tune not only the weights but also the point locations. The derivation proceeds as follows: consider an arbitrary order p polynomial,

$$f(x) = c_0 + c_1 x + \ldots + c_{p-1}x^{p-1} + c_p x^p. \tag{1.3.2}$$

Integrating from -1 to 1,

$$I \equiv \int_{-1}^{1} f(x)dx = 2\left(c_0 + \frac{c_2}{3} + \frac{c_4}{5} + \ldots\right).$$

Note that the odd powers of x integrate to zero, leaving $\lfloor p/2 \rfloor + 1$ terms – $\lfloor a \rfloor$ is the greatest integer less than or equal to a. So to compute I we just need $N = \lfloor p/2 \rfloor + 1$ pieces of

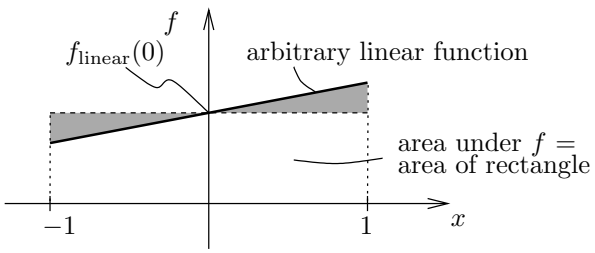


Figure 1.3.1: Integration of a linear function only requires one point.

information (i.e. values of f). Put another way, N points will let us integrate polynomial orders up to $p = 2N - 1$.

For example, suppose $N = 2$. We should be able to integrate polynomials up to $p = 3$ (cubics):

$$\begin{aligned} I = 2\left(c_0 + \frac{c_2}{3}\right) &= w_1 f(x_1) + w_2 f(x_2) \\ &= w_1\left(c_0 + c_1 x_1 + c_2 x_1^2 + c_3 x_1^3\right) + w_2\left(c_0 + c_1 x_2 + c_2 x_2^2 + c_3 x_2^3\right) \end{aligned}$$

Matching coefficients on c_i,

$$\begin{aligned} c_0 : 2 &= w_1 + w_2, \\ c_1 : 0 &= w_1 x_1 + w_2 x_2, \\ c_2 : \frac{2}{3} &= w_1 x_1^2 + w_2 x_2^2, \\ c_3 : 0 &= w_1 x_1^3 + w_2 x_2^3 \end{aligned}$$

Solving this system gives

$$w_1 = w_2 = 1, \quad x_1 = \frac{-1}{\sqrt{3}},\ x_2 = \frac{1}{\sqrt{3}}$$

Note that computing these weights/points required solution of a nonlinear system of equations, and this gets harder for large N.

For general N, and $p = 2N - 1$, the above approach gives

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_N \\ x_1^2 & x_2^2 & x_3^2 & \cdots & x_N^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{2N-1} & x_2^{2N-1} & x_3^{2N-1} & \cdots & x_N^{2N-1} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} \int_{-1}^{1} 1 dx \\ \int_{-1}^{1} x dx \\ \int_{-1}^{1} x^2 dx \\ \vdots \\ \int_{-1}^{1} x^{2N-1} dx \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 2/3 \\ \vdots \\ 0 \end{bmatrix}$$

This non-linear system can be solved for the unknowns, w_q and x_q, via the Newton-Raphson method. However, Newton-Raphson does not always converge, and success depends on the

initial guess, system conditioning, and continuation strategies. In particular, the system becomes increasingly ill-conditioned for high N as columns start to appear linearly dependent, which stems from the fact that monomials are not a good basis.

The solution is to use a well-conditioned basis, $b_i(x)$, instead of monomials in the expansion of $f(x)$ in Equation 1.3.2,

$$f(x) = \sum_{i=0}^{2N-1} c_i b_i(x). \tag{1.3.3}$$

Our quadrature rule should be able to integrate all basis functions exactly,

$$\int_{-1}^{1} b_n(x)dx = \sum_{q=1}^{N} w_q b_n(x_q), \qquad n \in [0, 2N-1] \tag{1.3.4}$$

These $2N$ equations will still generally constitute a nonlinear system for our $2N$ unknowns: N pairs (x_q, w_q). A clever trick to avoid the nonlinear system is to use *orthogonal* polynomials in defining $b_n(x)$, and we separate these into low and high-order basis functions,

"Low-order" basis functions			"High-order" basis functions		
$b_0(x)$	$=$	$l_0(x)$	$b_N(x)$	$=$	$l_N(x)$
$b_1(x)$	$=$	$l_1(x)$	$b_{N+1}(x)$	$=$	$l_N(x)l_1(x)$
	$\vdots$			$\vdots$	
$b_{N-1}(x)$	$=$	$l_{N-1}(x)$	$b_{2N-1}(x)$	$=$	$l_N(x)l_{N-1}(x)$

The $l(x)$ are orthogonal polynomials, and $l_i(x)$ is of order i. These can be obtained by Gram-Schmidt orthogonalization of the monomials. Over our interval $[-1, 1]$ the result is the Legendre polynomials.

Note that we do not use Legendre functions above $l_N(x)$ – i.e. no $l_{N+1}(x)$ etc. Yet we still have a complete basis up to order $2N-1$, as each b_n brings in a new higher-power monomial. Of course, the b_n are not all orthogonal.

The critical observation is that the N "high-order" basis functions each contain $l_N(x)$ in their definition. By choosing x_q to be the roots of $l_N(x)$, we automatically satisfy Equation 1.3.4 for $b_N(x), b_{N+1}(x), \ldots, b_{2N-1}(x)$:

$$\begin{aligned} \int_{-1}^{1} b_{N+j}(x)dx &= \sum_{q=1}^{N} w_q b_{N+j}(x_q), \qquad j \in [0, N-1] \\ \Leftrightarrow \qquad \int_{-1}^{1} l_N(x) l_j(x)dx &= \sum_{q=1}^{N} w_q l_N(x_q) l_j(x_q) \end{aligned}$$

The left-hand side is zero by orthogonality of the $l_i(x)$, and the right-hand side is zero because the x_q are the roots of $l_N(x)$. That just leaves the task of determining N weights, w_q, and

for this we have N equations from Equation 1.3.4 for the "low-order" $b_n(x)$,

$$\int_{-1}^{1} b_n(x)dx = \sum_{q=1}^{N} w_q b_n(x_q), \quad n \in [0, N-1] \quad \Leftrightarrow \quad \int_{-1}^{1} l_n(x)dx = \sum_{q=1}^{N} w_q l_n(x_q).$$

This is just an $N \times N$ linear system,

$$\begin{bmatrix} l_0(x_1) & l_0(x_2) & \cdots & l_0(x_N) \\ l_1(x_1) & l_1(x_2) & \cdots & l_1(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ l_{N-1}(x_1) & l_{N-1}(x_2) & \cdots & l_{N-1}(x_N) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} \int_{-1}^{1} l_0(x)dx \\ \int_{-1}^{1} l_1(x)dx \\ \vdots \\ \int_{-1}^{1} l_{N-1}(x)dx \end{bmatrix}$$

Solving this well-conditioned system gives the N quadrature weights w_n to accompany the N quadrature points x_q.

For higher dimensions, we can develop new rules or extend the existing 1D rules. For integrating on a square, we can use a tensor product of 1D points, $\mathbf{x}_{q,r} = (x_q, x_r)$, and multiply the weights together: $w_{q,r} = w_q w_r$. The result is N^2 points that can integrate tensor-product polynomials up to order p in each dimension. Anisotropic point sets are possible, e.g. order p_1 in one direction and p_2 in the other. Furthermore, we can take out some points to integrate full-order instead of tensor-product space. This is known as sparse-grid or Smolyak quadrature [106].

1.4 Time Marching

We consider a model system of a PDE already discretized in space, leaving the unknowns in a vector $\mathbf{U}$ that must be evolved in time,

$$\frac{d\mathbf{U}}{dt} = \mathbf{F}(\mathbf{U}, t), \qquad \mathbf{U}(0) = \mathbf{U}_0, \qquad \mathbf{U}, \mathbf{F}, \mathbf{H} \in \mathbb{R}^N. \tag{1.4.1}$$

Initially, for analysis, we will assume that the system is linear, so that

$$\frac{d\mathbf{U}}{dt} = \mathbf{F}(\mathbf{U}, t) \equiv \mathbf{A}\mathbf{U} + \mathbf{H}(t), \quad \mathbf{H} \in \mathbb{R}^N, \mathbf{A} \in \mathbb{R}^{N \times N}.$$

Assume that $\mathbf{A}$ is constant and diagonalizable, $\mathbf{A} = \mathbf{L}^{-1}\mathbf{\Lambda}\mathbf{L}$, with $\mathbf{\Lambda} = \text{diag}[\lambda_1, \lambda_2, \ldots, \lambda_N]$. Note, λ_j could be complex even for real-valued $\mathbf{A}$. The system then decouples,

$$\begin{aligned} \frac{d(\mathbf{L}\mathbf{U})}{dt} = \mathbf{\Lambda}\,\mathbf{L}\mathbf{U} + \mathbf{L}\mathbf{H}(t) \quad &\Rightarrow \quad \frac{d\hat{\mathbf{U}}}{dt} = \mathbf{\Lambda}\hat{\mathbf{U}} + \hat{\mathbf{H}}(t) \qquad (\hat{\mathbf{U}} = \mathbf{L}\mathbf{U},\ \hat{\mathbf{H}} = \mathbf{L}\mathbf{H}) \\ &\Rightarrow \quad \frac{d\hat{U}_j}{dt} = \lambda_j \hat{U}_j + \hat{H}_j(t) \qquad (\hat{\mathbf{U}} = \{\hat{U}_j\},\ \hat{\mathbf{H}} = \{\hat{H}_j\}) \end{aligned}$$

Dropping the $\hat{}$ and j subscript, the model *scalar* ODE is

$$\boxed{\frac{dU}{dt} = F(U,t) \equiv \lambda U + H(t)} \tag{1.4.2}$$

The prototypical *explicit* time integration scheme is the **forward Euler** method,

$$\frac{U^{n+1} - U^n}{\Delta t} = \lambda U^n + H(t^n), \qquad U^n = U(t^n),\ t^n = n\Delta t. \tag{1.4.3}$$

An initial perturbation, $\tilde{U}^0 = U^0 + E^0$ will yield the solution $\tilde{U}^n = U^n + E^n$, where E^n satisfies the homogeneous equation,

$$\frac{E^{n+1} - E^n}{\Delta t} = \lambda E^n.$$

Substituting $E^n = g^n E^0$ yields the *error amplification factor* $g = 1 + \lambda\Delta t$, which dictates eigenvalue stability, illustrated in Figure 1.4.1.

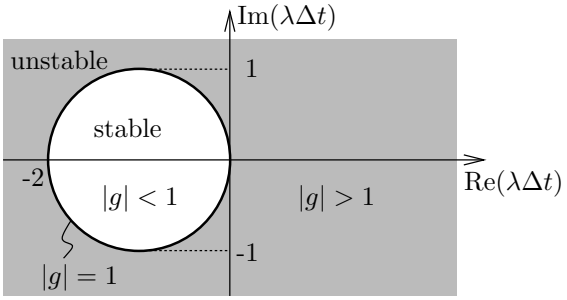


Figure 1.4.1: Forward Euler eigenvalue stability region.

The prototypical *implicit* time integration scheme is the **backward/implicit Euler** method,

$$\frac{U^{n+1} - U^n}{\Delta t} = \lambda U^{n+1} + H(t^{n+1}).$$

The corresponding error amplification equation and amplification factor are

$$\frac{E^{n+1} - E^n}{\Delta t} = \lambda E^{n+1} \quad \Rightarrow \quad g = \frac{1}{1 - \lambda\Delta t}.$$

Figure 1.4.2 shows the eigenvalue stability region, which now includes most of the complex number plane, with the exception of a unit disk in the right-half-plane.

We now make the following definitions:

- A scheme is zero stable if the error remains bounded for $\Delta t \to 0$, assuming $\mathrm{Re}(\lambda) < 0$.
- Eigenvalue stability concerns stability for a fixed $\Delta t > 0$.
- A scheme is A-stable if it is stable ($g \leq 1$) for the entire left-hand plane of $\lambda\Delta t$.
- A scheme is L-stable if it is A-stable and $g(-\infty) \to 0$.

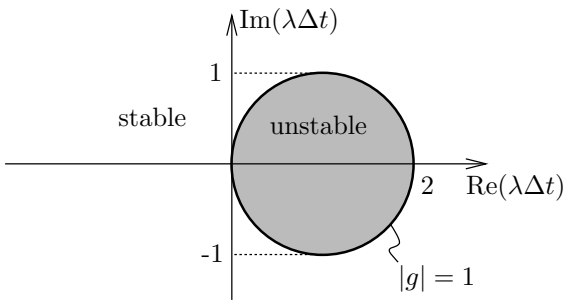


Figure 1.4.2: Backward Euler eigenvalue stability region.

1.4.1 Eigenvalue Analysis

Consider the one-dimensional scalar advection-diffusion equation,

$$\frac{\partial U}{\partial t} - a\frac{\partial U}{\partial x} - \nu\frac{\partial^2 U}{\partial x^2} = 0, \qquad a \geq 0,\ \nu \geq 0. \tag{1.4.4}$$

The analytical solution is $U(x,t) = \sum_{m=-\infty}^{\infty} \hat{U}_m(t) e^{ik_m x}$ where m is an integer and $k_m = 2\pi m$. $\hat{U}_m$ satisfies

$$\frac{d\hat{U}_m}{dt} = \lambda_m \hat{U}_m, \qquad \text{where} \quad \lambda_m = iak_m - \nu k_m^2$$

In the diffusion limit, $\lambda_m = -\nu k_m^2$, which means that the eigenvalues are on the negative real axis, as shown in Figure 1.4.3. Consider the wavelength and wavenumber of mode m: these are related via

$$\underbrace{\text{small wavelength}}_{L_m \sim \Delta x} \leftrightarrow \underbrace{\text{large wavenumber}}_{k_m}: \quad L_m = \frac{2\pi}{k_m}.$$

On a given grid, we can resolve some minimum wavelength, $L_{\min} = 2\pi/k_{\max}$. The corresponding largest magnitude eigenvalue is

$$\lambda_{\min} = -\nu k_{\max}^2 \sim \frac{-1}{\Delta x^2}.$$

e.g. forward Euler stability requires $\Delta t < \frac{2}{\nu k_{\max}^2} \sim \Delta x^2$.

In the inviscid limit, $\lambda_m = iak_m$, which means that the eigenvalues lie on the imaginary axis. As discussed above, $k_{\max}$, maximum wavenumber on a given grid, scales as $1/\Delta x$. Since the eigenvalues are on the imaginary axis, forward Euler is unstable for all Δt. On the other hand, backward Euler is always stable. If ν is slightly greater than zero, viscous effects dominate for small enough Δx (large k_m),

$$\nu k_m^2 > ak_m \ \Leftrightarrow\ k_m > \frac{a}{\nu} \ \Leftrightarrow\ \underbrace{\frac{aL_m}{\nu}}_{\text{cell Reynolds number}} < 2\pi$$

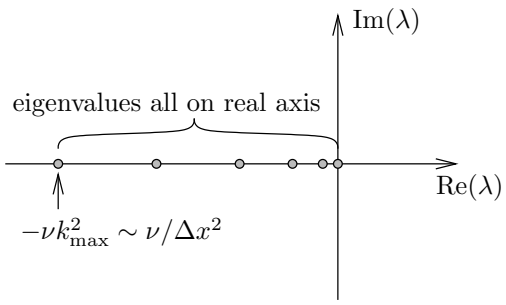


Figure 1.4.3: Diffusion limit eigenvalues.

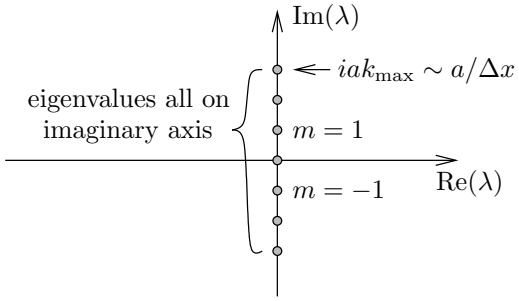


Figure 1.4.4: Inviscid limit eigenvalues.

1.4.2 Multi-step Methods

The key idea of multi-step methods is to use previous time nodes to advance the solution from n to $n+1$. For the prototypical scalar problem,

$$\frac{dU}{dt} = F(U,t), \tag{1.4.5}$$

a general K-step method reads

$$\sum_{k=1-K}^{1} \alpha_k U^{n+k} = \Delta t \sum_{k=1-K}^{1} \beta_k F(U^{n+k}, t^{n+k}) \tag{1.4.6}$$

Consistency, stability, and accuracy dictate choices of K, α_k, β_k. Note, for explicit schemes, $\beta_1 = 0$. The **Dahlquist equivalence theorem** states that

$$\left.\begin{array}{ll} \text{consistency:} & \displaystyle\sum_{k=1-K}^{1} \alpha_k = 0 \;\text{ and }\; \sum_{k=1-K}^{1} k\alpha_k = \sum_{k=1-K}^{1} \beta_k \\ \text{zero-stability:} & \text{roots of } \displaystyle\sum_{k=1-K}^{1} \alpha_k g^k = 0 \text{ satisfy } |g| < 1 \end{array}\right\} \Rightarrow \quad \text{convergence}$$

To analyze eigenvalue stability: we set $U^n = g^n U^0$ and solve for $g(\lambda\Delta t)$. The $|g| = 1$ contour indicates the stability boundary. The order of accuracy of a multi-step method is p when LHS-RHS is $\mathcal{O}(\Delta t^{p+1})$ in Equation 1.4.6.

Sample multi-step methods include:

- Second-order Backward Difference (BDF2): $\frac{3}{2}U^{n+1} - 2U^n + \frac{1}{2}U^{n-1} = \Delta t F(U^{n+1}, t^{n+1})$
- Third-order Backward Difference (BDF3): $\frac{11}{6}U^{n+1} - 3U^n + \frac{3}{2}U^{n-1} - \frac{1}{3}U^{n-2} = \Delta t F(U^{n+1}, t^{n+1})$
- Second-order Adams-Bashforth (AB2): $U^{n+1} = U^n + \Delta t(3F(U^n, t^n) - F(U^{n-1}, t^{n-1}))/2$
- Second-order Adams-Moulton or Trapezoidal (Trap): $U^{n+1} = U^n + \frac{\Delta t}{2}(F(U^{n+1}, t^{n+1}) + F(U^n, t^n))$
- Third-order Adams-Moulton (AM3): $U^{n+1} = U^n + \frac{\Delta t}{12}(5F(U^{n+1}, t^n) + 8F(U^n, t^n) - F(U^{n-1}, t^{n-1}))$

1.4.3 Multi-Stage Methods

The idea in multi-stage methods is to use temporary in-between states in advancing the solution from n to $n+1$. One of the simplest two-stage schemes is the predictor-corrector method,

$$\tilde{U} = U^n + \Delta t\, F(U^n, t^n) \tag{1.4.7}$$
$$U^{n+1} = U^n + \Delta t\, F(\tilde{U}, t^{n+1}) \tag{1.4.8}$$

A more accurate version is the MacCormack scheme,

$$\tilde{U} = U^n + \Delta t\, F(U^n, t^n) \tag{1.4.9}$$
$$U^{n+1} = \frac{1}{2}\left(U^n + \tilde{U} + \Delta t\, F(\tilde{U}, t^{n+1})\right) \tag{1.4.10}$$

Any general second-order accurate predictor-corrector scheme can be written as

$$\begin{aligned} \tilde{U} &= U^n + \alpha\Delta t\, F(U^n, t^n) \\ U^{n+1} &= U^n + \frac{1}{2}\Delta t\left(\frac{1}{\alpha}F(\tilde{U}, t^n + \alpha\Delta t) + \frac{2\alpha - 1}{\alpha}F(U, t^n)\right) \end{aligned}$$

The most popular multi-stage methods are from the Runge-Kutta (RK) family. RK2, a second-order accurate method, is

$$\begin{aligned} F_0 &= F\left(U^n,\, t^n\right) \\ F_1 &= F\left(U^n + \frac{1}{2}\Delta t F_0,\, t^n + \frac{\Delta t}{2}\right) \\ U^{n+1} &= U^n + \Delta t F_1 \end{aligned}$$

RK4, a fourth-order accurate method, is

$$\begin{aligned} F_0 &= F\left(U^n,\, t^n\right) \\ F_1 &= F\left(U^n + \frac{1}{2}\Delta t F_0,\, t^n + \frac{\Delta t}{2}\right) \\ F_2 &= F\left(U^n + \frac{1}{2}\Delta t F_1,\, t^n + \frac{\Delta t}{2}\right) \\ F_3 &= F\left(U^n + \Delta t F_2,\, t^n + \Delta t\right) \\ U^{n+1} &= U^n + \frac{\Delta t}{6}\left(F_0 + 2F_1 + 2F_2 + F_3\right) \end{aligned}$$

Figure 1.4.5 shows stability diagrams for various methods obtained by plotting the $|g| = 1$ contour.

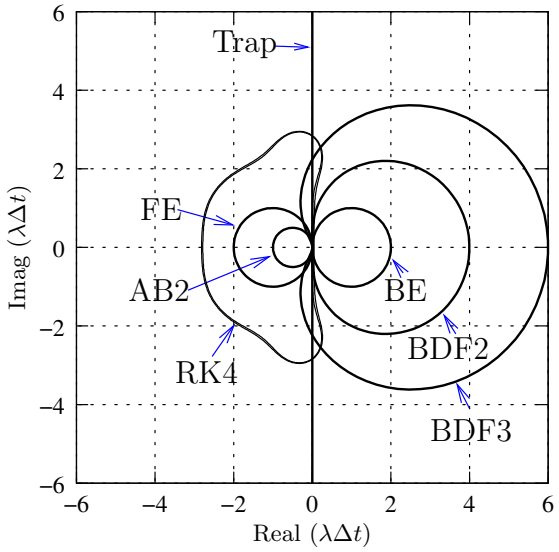


Figure 1.4.5: Eigenvalue stability boundaries for several multi-step and multi-stage time integration methods.

1.4.4 Implicit Methods

We call a system of equations stiff if it exhibits a wide range of time scales, and if the fast scales do not affect accuracy of the outputs of interest. Mathematically, stiff systems have highly-disparate eigenvalues, as shown schematically in Figure 1.4.6. A rule of thumb is that if the ratio of the largest to smallest eigenvalue magnitudes, $|\lambda_{\max}|/|\lambda_{\min}|$ is greater than about 1000, the system is likely stiff. For time-marching stiff systems, implicit methods will usually be more efficient than explicit methods.

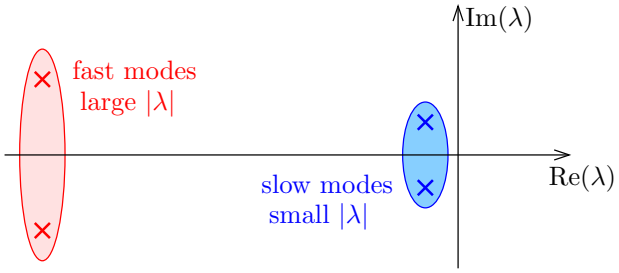


Figure 1.4.6: Stiff systems are characterized by dipsarate eigenvalues.

An example of a stiff system is a highly-damped spring-mass system, shown in Figure 1.4.7. Using x as the displacement from equilibrium, the equations of motion are

$$m\ddot{x} = -kx - c\dot{x}, \quad \mathbf{u} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}, \quad \dot{\mathbf{u}} = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} x \\ \dot{x} \end{bmatrix}}_{\mathbf{u}} = \underbrace{\mathbf{A}\mathbf{u}}_{\mathbf{f}} \tag{1.4.11}$$

The problem becomes stiff when the damping is large. Consider the case $m = 1, k = 1, c =$

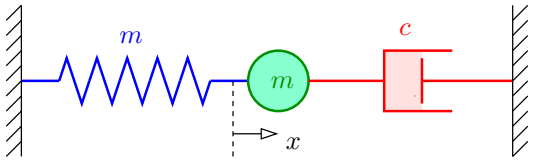


Figure 1.4.7: Damped spring-mass system.

100. The system matrix and eigenvalues are

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -1 & -100 \end{bmatrix}$$

$$\lambda_1 \approx -.01, \quad \lambda_2 \approx -99.99$$

Since $|\lambda|_{\max}/|\lambda|_{\min} \approx 10^4$, the system qualifies as stiff. The eigenvalues are real, so the system does not oscillate. Physically, the behavior is fast damping of large velocities, and

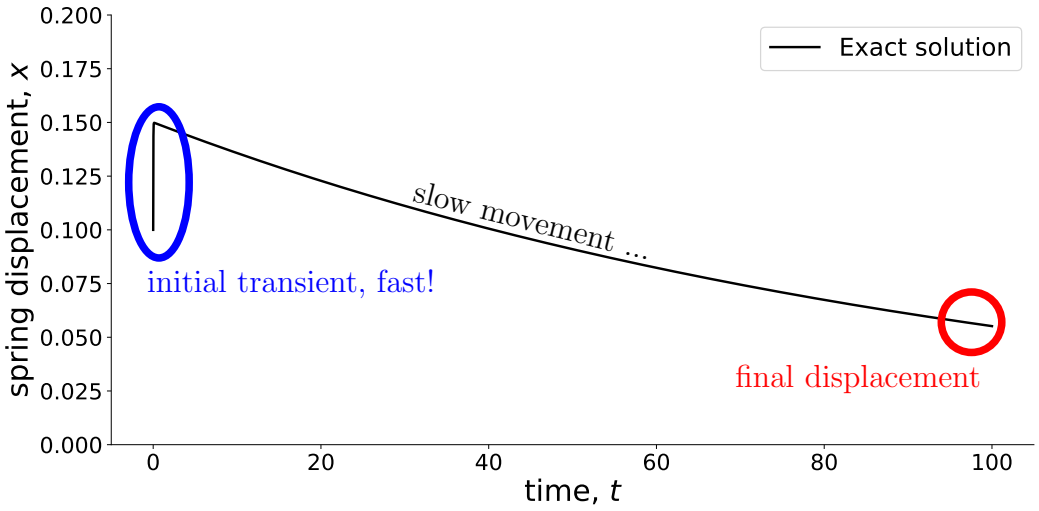


Figure 1.4.8: Solution of a highly-damped spring-mass system.

slow changes in the position. Figure 1.4.8 shows the solution of the spring displacement for an initial state $\mathbf{u}^0 = [0.1, 5]^T$, i.e. a displacement of $x = 0.1$ and velocity of $\dot{x} = 5$.

Let's apply an explicit method, RK2, to this problem. Recall that RK2 for the system $\dot{\mathbf{u}} = \mathbf{f}(\mathbf{u})$ is

$$\begin{aligned} \mathbf{u}^{n+\frac{1}{2}} &= \mathbf{u}^n + \frac{1}{2}\Delta t\, \mathbf{f}(\mathbf{u}^n) \\ \mathbf{u}^{n+1} &= \mathbf{u}^n + \Delta t\, \mathbf{f}(\mathbf{u}^{n+\frac{1}{2}}) \end{aligned}$$

For stability, we need the most negative eigenvalue inside the stability boundary, which extends to -2 in the $\lambda\Delta t$ complex number plane. The eigenvalues of this system are real, and hence we require

$$|\lambda|_{\max}\Delta t \leq 2$$

Since $|\lambda|_{\max} \approx 100$, we have $\Delta t \leq 2/100 = .02$. Figure 1.4.9 shows the results of applying RK2 to this problem, up to $T = 100$, with the initial condition $\mathbf{u}^0 = [0.1, 5]^T$. For $T = 100$ and $\Delta t_{\max} = .02$, we need $N \geq 5000$ time steps to maintain stability, and the results in the figure confirm this limit.

The stability requirement of $\Delta t \leq .02$ for RK2 requires many time steps and $\mathbf{f}$ evaluations for long time horizons. Can we do better with an implicit method? Let's try the BDF2 multi-step method,

$$\frac{3}{2}\mathbf{u}^{n+1} - 2\mathbf{u}^n + \frac{1}{2}\mathbf{u}^{n-1} = \Delta t \mathbf{f}(\mathbf{u}^{n+1})$$

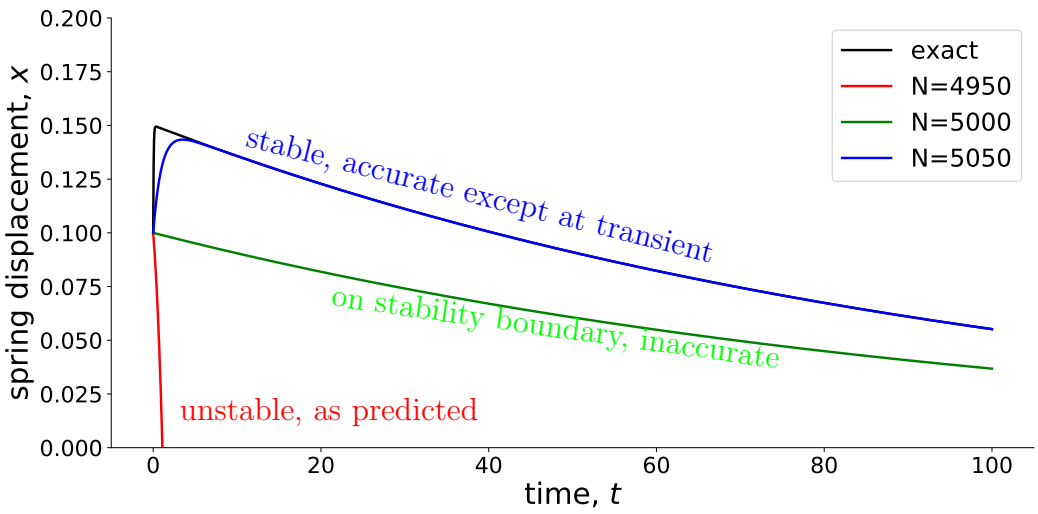


Figure 1.4.9: RK2 applied to the spring-mass system.

For $\mathbf{f} = \mathbf{A}\mathbf{u}$, we can solve for $\mathbf{u}^{n+1}$,

$$\begin{aligned} \frac{3}{2}\mathbf{u}^{n+1} - 2\mathbf{u}^n + \frac{1}{2}\mathbf{u}^{n-1} &= \Delta t \mathbf{A}\mathbf{u}^{n+1} \\ \left(\frac{3}{2}\mathbf{I} - \Delta t \mathbf{A}\right)\mathbf{u}^{n+1} &= 2\mathbf{u}^n - \frac{1}{2}\mathbf{u}^{n-1} \end{aligned}$$

This is a linear system for $\mathbf{u}^{n+1}$, which we can solve at every time step. In fact, as the matrix is small, the inverse can be pre-computed, so that only a matrix-vector multiplication is needed at each time step. Note that as an A-stable method, BDF2 has no time-step restriction, as both eigenvalues in this problem are in the left-half plane. The results in Figure 1.4.10 confirm that the solution remains stable even for very large time steps. The accuracy in the transient is not very good when we only have 10 time steps total, but this inaccuracy does not significantly affect the final-time solution.

To determine whether BDF2 is worth the additional work per time step, we perform a convergence study. Figure 1.4.11 plots the error in the final position of the mass at $T = 100$, as a function of the number of time steps. Both RK2 and BDF2 are second-order accurate and have similar errors for a given number of time steps. While BDF2 has a higher cost per time step, it can run with a much larger Δt (small N). On the other hand, RK2 is unstable until $N = 5000$, at which point the error is already below 10^{-9}! If we need such a high-level of accuracy, then it makes sense to use RK2. However, for more reasonable error tolerances, BDF2 wins. For example, with 200 time steps, BDF2 achieves an error of 10^{-6}.

Figure 1.4.11 also shows the result of RK4. While RK4 has a slightly larger stability region than RK2, it is **explicit**, and hence its time-step restriction is only slightly better than RK2. For such a restrictive time step, the fourth-order convergence rate does not get

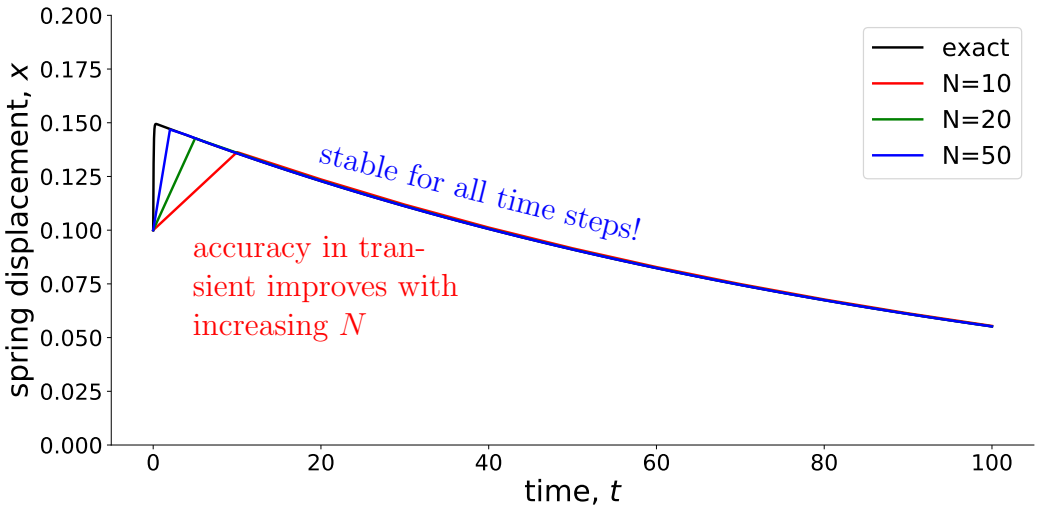


Figure 1.4.10: BDF2 applied to the spring-mass system.

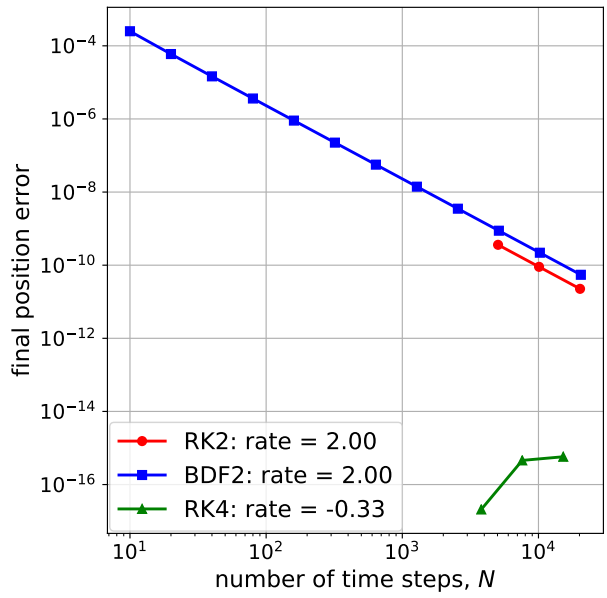


Figure 1.4.11: Final-time position convergence study of the spring-mass system.

a chance to help: by the time RK4 is stable, the error is already machine-precision. Hence, BDF2 still wins in accuracy versus cost for reasonable error levels.

In summary, stiff systems are characterized by a wide range of time scales, with the fast ones not affecting accuracy. Explicit methods are inefficient for stiff systems because of their severe time-step restriction, which is determined by the fast time scales. Implicit methods, particularly with A-stability, are not limited by time-step restrictions, and hence they can be more efficient for stiff systems. Note, however, that the cost of implicit methods becomes more expensive for nonlinear systems, in which Newton-Raphson or another nonlinear solver must be used at each time step.

1.5 Splines

Splines are often used to interpolate points in one or more dimensions. We will specifically look at using *cubic* splines, which consist of piecewise-continuous cubic polynomials.

1.5.1 One-dimensional splines

We begin in one dimension, where the goal is to interpolate N points $X_i(S_i)$ using a piecewise continuous cubic polynomial on each interval. Given the N points as parameter-position pairs, (S_i, X_i), $1 \le i \le N$, $S_i < S_{i+1}$, the goal is to solve for unknown slopes $\frac{dX}{dS}\big|_i$ at the N points. The equations arise form the requirement of continuous *second* derivatives at the N points. Figure 1.5.1 illustrates these quantities for one interval of a spline.

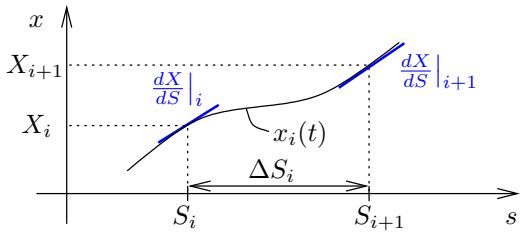


Figure 1.5.1: Givens and unknowns in a one-dimensional spline interpolation.

A spline consists of a cubic interpolant between two points. So, on each interval, we must construct a cubic given values and slopes at two end points. Consider interval i. Define $t = (s - S_i)/\Delta S_i$; note $t \in [0, 1]$ on the interval.

The interpolant on interval i is

$$x_i(t) = \underbrace{(1-t)X_i + tX_{i+1}}_{\text{linear interpolant}} + \underbrace{(t - t^2)\left[(1-t)x'_{i,0} - tx'_{i,1}\right]}_{\text{cubic bubble}}, \tag{1.5.1}$$

where the endpoint slopes are defined as

$$x'_{i,0} \equiv \left(\frac{dX}{dS}\Big|_i - \frac{X_{i+1} - X_i}{\Delta S_i}\right)\Delta S_i, \quad x'_{i,1} \equiv \left(\frac{dX}{dS}\Big|_{i+1} - \frac{X_{i+1} - X_i}{\Delta S_i}\right)\Delta S_i, \tag{1.5.2}$$

Differentiating $x_i(t)$ twice with respect to t gives

$$\frac{d^2x_i}{dt^2} = (6t-4)x'_{i,0} + (6t-2)x'_{i,1}. \tag{1.5.3}$$

Continuity of the interpolant second derivative with respect to s at node S_i requires

$$\left.\frac{d^2x_{i-1}}{ds^2}\right|_{S_i^-} = \left.\frac{d^2x_i}{ds^2}\right|_{S_i^+} \quad \Leftrightarrow \quad \left.\frac{d^2x_{i-1}}{dt^2}\right|_{t=1}\frac{1}{\Delta s_{i-1}^2} = \left.\frac{d^2x_i}{dt^2}\right|_{t=0}\frac{1}{\Delta s_i^2} \tag{1.5.4}$$

Using the expression for the second derivative from Equation 1.5.3, evaluated at the appropriate t (1 for interval $i-1$, and 0 for interval i), gives

$$\left[2x'_{i-1,0} + 4x'_{i-1,1}\right]\frac{1}{\Delta s_{i-1}^2} = \left[-4x'_{i,0} - 2x'_{i,1}\right]\frac{1}{\Delta s_i^2}. \tag{1.5.5}$$

Substituting the slope formulas from Equation 1.5.2 yields

$$\underbrace{\Delta S_i}_{B_i}\left.\frac{dX}{dS}\right|_{i-1} + \underbrace{2\left(\Delta S_{i-1} + \Delta S_i\right)}_{A_i}\left.\frac{dX}{dS}\right|_{i} + \underbrace{\Delta S_{i-1}}_{C_i}\left.\frac{dX}{dS}\right|_{i+1} = \underbrace{3\left[\frac{X_i - X_{i-1}}{\Delta S_{i-1}}\Delta S_i + \frac{X_{i+1} - X_i}{\Delta S_i}\Delta S_{i-1}\right]}_{D_i}$$

This is a linear tri-diagonal system for the unknowns $dX/dS|_i$,

$$\begin{bmatrix} A_1 & C_1 & 0 & \cdots & 0 \\ B_2 & A_2 & C_2 & \ddots & \vdots \\ 0 & B_3 & A_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & C_{N-1} \\ 0 & \cdots & 0 & B_N & A_N \end{bmatrix} \begin{bmatrix} dX/dS|_1 \\ dX/dS|_2 \\ dX/dS|_3 \\ \vdots \\ dX/dS|_{N-1} \\ dX/dS|_N \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_{N-1} \\ D_N \end{bmatrix}$$

Boundary conditions are enforced at the endpoints, $i = 1$ and $i = N$. Typical ones are

- Zero third derivative, $d^3x/ds^3 = 0$
- Zero second derivative, $d^2x/ds^2 = 0$
- Prescribed first derivative, $dx/ds = \text{const.}$

Note, the boundary conditions modify A_1, C_1, D_1 on the first node, and B_N, A_N, D_N on the last node.

1.5.2 Two-dimensional splines

In two dimensions, we need to interpolate N points (X_i, Y_i), and we do so using piecewise parametric polynomials between the points. Denote by s the arc length along the parametric curve $\vec{x}(s) \equiv (x(s), y(s))$, as illustrated in Figure 1.5.2. If we knew S_i, the arc-length value at the nodes, a priori, we could calculate one-dimensional splines for $x(s)$ and $y(s)$ independently. However, this arc-length information is generally not available to start (since we do not know the spline ahead of time!), and hence given just the pairs $\vec{X}_i \equiv (X_i, Y_i)$, we need to iterate.

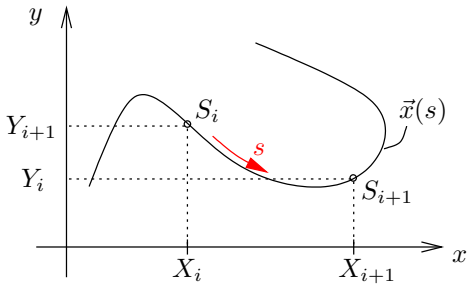


Figure 1.5.2: Two-dimensional spline interpolation.

The steps in constructing a 2D spline are as follows:

1. Estimate the arc-length parameter at the nodes using simple linear distances,
$$S_1 = 0, \quad S_{i+1} = S_i + \sqrt{(X_{i+1} - X_i)^2 + (Y_{i+1} - Y_i)^2} \tag{1.5.6}$$
2. Spline $x(s)$ and $y(s)$ independently using the current estimate for S_i.
3. Compute the true arc-length of the current curve, $(x(s), y(s))$. Call this S_i^{true} at the nodes.
$$S_1^{\text{true}} = 0, \quad S_{i+1}^{\text{true}} = S_i^{\text{true}} + \int_{S_i}^{S_{i+1}} \sqrt{\left(\frac{dx_i}{ds}\right)^2 + \left(\frac{dy_i}{ds}\right)^2}\, ds$$
where the derivative with respect to the arc length is
$$\frac{dx_i}{ds} = \frac{1}{\Delta S_i}\frac{dx_i}{dt} = \frac{1}{\Delta S_i}\left[X_{i+1} - X_i + (1 - 4t + 3t^2)x'_{i,0} + (-2t + 3t^2)x'_{i,1}\right].$$
The integral over each interval is typically done numerically, for example using quadrature.
4. Check the arc-length error, e.g. $||S_i - S_i^{\text{true}}||_{L_1}$. If the error is below a specified tolerance (usually normalized by $S_N - S_1$), the splining algorithm terminates.
5. Otherwise, set $S_i = S_i^{\text{true}}$ and return to step 2.

Boundary conditions in the form of first, second, or third derivatives can be enforced independently on the $x(s)$ and $y(s)$ 1D splines.

1.6 Discretization Approaches

In this section we review three families of discretization approaches, for a prototypical partial differential equation written as

$$r(u) = 0 \tag{1.6.1}$$

where

- $u(x)$ is an unknown scalar field (in one spatial dimension for now)
- $r(u)$ is a continuous residual operator

For example, $r(u) = \mathcal{L}u - f$ for a differential operator $\mathcal{L}$ and a known function $f(x)$.

1.6.1 The Finite Difference Method

The idea underlying the finite-difference method is to solve for point values of u at a large number of points. Referring to Figure 1.6.1, we have the following definitions,

x_j	:	j^{th} spatial node, $1 \leq j \leq N$
Δx	:	spacing between nodes (assumed same for all intervals)
$\hat{u}_j$	:	approximate solution at node x_j
N	:	number of nodes = number of unknowns
$\mathbf{U}$	:	$N \times 1$ vector of unknowns, $\{\hat{u}_j\}$

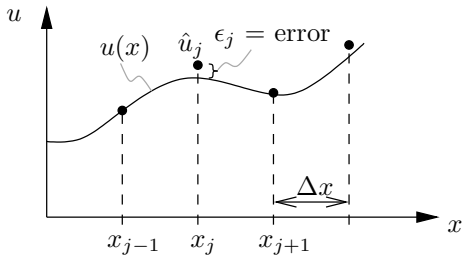


Figure 1.6.1: Unknowns in the finite-difference method.

N equations result from replacing derivative operators with difference approximations at each node x_i. For example, a typical second-derivative difference is

$$u_{xx}(x_i) = \frac{\hat{u}_{i-1} - 2\hat{u}_i + \hat{u}_{i+1}}{\Delta x^2} + \mathcal{O}(\Delta x^2). \tag{1.6.2}$$

We can derive or verify this expression using Taylor-series expansions. Let $R_i(\mathbf{U}) = 0$ be the resulting finite-difference approximation of $r(u) = 0$ at node x_i. Putting these equations together yields the discrete system

$$\mathbf{R}(\mathbf{U}) = \mathbf{0},$$

where $\mathbf{R}(\mathbf{U}) = \{R_i(\mathbf{U})\}$ and $1 \leq i \leq N$. This vector equation represents N equations for N unknowns, since we have one equation at each node.

Example 1.1 (Finite-differences for the Laplace operator)**.** Consider the linear differential operator $\mathcal{L} = -\frac{\partial^2}{\partial x^2}$ so that $r(u) = -u_{xx} - f$. Using Equation 1.6.2 on a set of N uniformly-spaced points gives

$$\mathbf{R}(\mathbf{U}) = \mathbf{A}\mathbf{U} - \mathbf{F} = \mathbf{0},$$

where for homogeneous Dirichlet boundary conditions ($\hat{u}_0 = \hat{u}_{N+1} = 0$),

$$\mathbf{A} = \frac{1}{(\Delta x)^2}\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & -1 & 1 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \quad \text{and} \quad \mathbf{F} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_N) \end{bmatrix}.$$

This system is solved by moving all knowns to the right-hand side and directly or iteratively inverting $\mathbf{A}$ to obtain the unknown vector $\mathbf{U} = \mathbf{A}^{-1}\mathbf{F}$. If the system were not linear, a Newton-Raphson approach would be required.

Boundary conditions (BCs) can be imposed in various ways. Two of these are as follows.

- Dirichlet BCs can be enforced by not including nodes on Dirichlet boundaries as unknowns and instead using the known values in the difference formulas (the values will end up on the right-hand side).
- Neumann BCs can be enforced by writing one-sided difference formulas for the given derivatives on the boundaries to express the boundary node values in terms of interior unknowns.

1.6.2 The Finite Element Method

The idea underlying the finite-element method is to solve for a function with piecewise local support that approximates the solution u. Specifically, $u(x)$ is approximated by a weighted series of basis functions,

$$u(x) \approx u_h(x) \equiv \sum_{j=1}^{N} U_j \phi_j(x). \tag{1.6.3}$$

Referring to Figure 1.6.2, we have the following definitions,

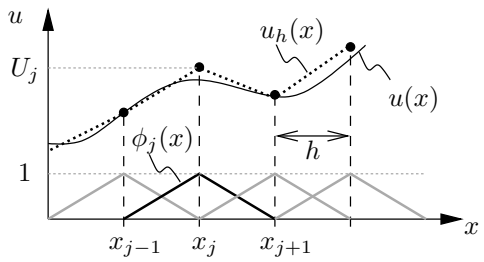


Figure 1.6.2: Unknowns in the finite-element method.

$\phi_j(x)$	:	j^{th} spatial basis function, $1 \leq j \leq N$
h	:	characteristic element (here an interval) size
U_j	:	coefficient on j^{th} basis function in approximation series
N	:	number of basis functions = number of unknowns
$\mathbf{U}$	:	$N \times 1$ vector of unknowns, $\{U_j\}$

Simplicity of the discrete equations relies on $\phi_j(x)$ having *local support* – each 1D "hat" basis function shown in Figure 1.6.2 has support on only two elements. Other functions, not necessarily nodal, could also be used (e.g. a high-order basis). Discontinuous basis functions can also be used as long as any integrations by parts are performed carefully.

N equations can be obtained using:

- Collocation: Enforce the strong-form of the equations at N selected points, x_i, $1 \leq i \leq N$,

$$r(u_h)|_{x_i} = 0.$$

  Derivatives in $r(u_h)$ are computed directly from the basis function series expansion. This requires basis functions to be of sufficient order to accommodate these derivatives.

- Weighted residuals: Set weighted integrals over the domain Ω of $r(u)$ to zero for N weight (test) functions, $w_i(x)$, $1 \leq i \leq N$,

$$R_i \equiv \int_\Omega w_i(x) r(u_h(x))\, dx = 0. \tag{1.6.4}$$

  In a *Galerkin* FEM, the N basis functions $\phi_i(x)$ are used as the weight functions. The above integrals become restricted to only over a few elements at a time when the weight functions have local support. Combined with a locally-supported basis, the result is a sparsely-coupled system of equations.

  Integration by parts is typically used in Equation 1.6.4 to transfer some derivatives off $r(u_h)$ to w_i. This allows the basis/weight functions to be of lower regularity than required to accommodate derivatives in $r(u)$ – e.g. piecewise linear $\phi_j(x)$ can be used when $r(u)$ contains second derivatives.

Example 1.2 (Finite elements for the Laplace operator)**.** Consider the linear differential operator $\mathcal{L} = -\frac{\partial^2}{\partial x^2}$ so that $r(u) = -u_{xx} - f$. Using a Galerkin FEM (weighted residuals), one integration by parts gives

$$R_i = \int_\Omega \frac{\partial \phi_i}{\partial x}\frac{\partial u_h}{\partial x}\,dx - \left[\phi_i \frac{\partial u_h}{\partial x}\right]_{x_L}^{x_R} - \int_\Omega \phi_i f\,dx, \qquad \text{where } \Omega = [x_L, x_R]. \tag{1.6.5}$$

Expanding u_h via Equation 1.6.3 then gives a linear $N \times N$ system of equations.

Boundary conditions (BCs):

- Dirichlet BCs can be enforced by not associating unknowns to basis functions that are nonzero on the boundary.
- Neumann BCs are enforced through the second term in Equation 1.6.5. Note that this term is not active for Dirichlet BCs because all ϕ_i are zero on Dirichlet boundaries. In the Neumann case, boundary basis functions need to be included (boundary values are unknown).

1.6.3 The Finite Volume Method

In the finite-volume method, we solve for cell averages that approximate the solution to a conservation law. Referring to Figure 1.6.3, we have the following definitions,

$x_{j\pm1/2}$	:	coordinates of left/right endpoints of cell j
Δx_j	:	size of j^{th} cell (here an interval)
U_j	:	approximated average of u in cell j
N	:	number of cells = number of unknowns
$\mathbf{U}$	:	$N \times 1$ vector of unknowns, $\{U_j\}$

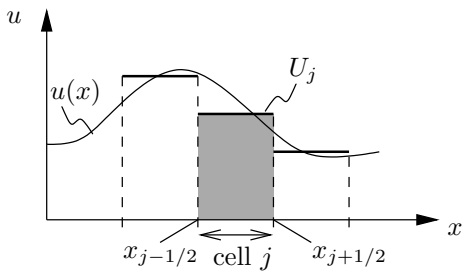


Figure 1.6.3: Unknowns in the finite-volume method.

The approximated average of u in cell j is given by

$$\frac{1}{\Delta x_j}\int_{\text{cell } j} u(x)\,dx \to U_j \tag{1.6.6}$$

N equations are obtained by requiring the integral of $r(u)$ to vanish for each cell. The finite-volume method is geared for conservation laws, for which $r(u) = \frac{\partial f}{\partial x}$ for some flux $f(u)$. The N residuals that must be driven to zero are then, after an integration by parts,

$$R_i = \int_{\text{cell } i} \frac{\partial f}{\partial x}\, dx = f_{i+1/2} - f_{i-1/2}, \qquad f_{i\pm 1/2} = \text{ fluxes on cell interfaces.}$$

The method is completed by defining *numerical fluxes*, $\hat{f}_{i\pm 1/2}$, that express the interface fluxes in terms of the unknowns U_j – here is where the approximation comes in, as we do not know the value of u on the interfaces (we just have two cell averages on either side). These numerical fluxes are typically obtained from (approximate) solutions to the Riemann problem on cell interfaces.

Example 1.3 (The finite-volume method for the advection operator)**.** Consider the linear differential operator $\mathcal{L} = a\frac{\partial}{\partial x}$, where a is constant, so that $r(u) = (au)_x$ – i.e. $f(u) = au$. An upwind (identical to Roe) flux in this case is

$$\hat{f}_{i+1/2} = \frac{1}{2}(f_i + f_{i+1}) - \frac{1}{2}|a|(U_{i+1} - U_i). \tag{1.6.7}$$

The method as described will be first-order accurate. High-order accuracy can be obtained by reconstructing the state at the interfaces using a wider stencil of neighboring cells. Shock capturing then requires non-linear limiting of these reconstructions to prevent oscillations/instabilities.

Boundary conditions are imposed through fluxes on the domain boundaries. In some cases it is convenient to define "ghost cells", with known states, just outside the domain to simplify the calculation of these boundary fluxes. For hyperbolic systems of equations, a characteristic analysis is required to determine the amount of information that should be specified on each boundary (e.g. inflow versus outflow).

1.6.4 Other Methods

- Cartesian method: typically finite volume on a Cartesian (structured) grid. Often combined with a cut-cell treatment of boundaries in which a geometry is cut out of the regular background grid leaving irregularly-shaped cells adjacent to the geometry.
- Meshless methods: typically finite difference on a cloud of points (unstructured).
- Immersed boundary: typically finite difference treatment of one or more fluids with an embedded interface that affects the fluid via a momentum source/forcing term.

Chapter 2

Mesh Generation

Meshes are used to discretize the computational domain for various numerical solution techniques, including finite-volume and finite-element methods. This chapter discusses different types of meshes and algorithms for generating them.

2.1 Definitions

In a numerical analysis problem, we seek to approximately solve a partial differential equation over certain **computational domain**. A **mesh** is a subdivision of the computational domain into smaller, usually regularly-shaped, volumes (areas in two dimensions) that fill the entire domain and do not overlap. Such volumes are called **elements** or **cells**. We will use these two terms interchangeably, though "cell" is the more common name in finite-volume methods, while "element" is the more common name in finite-element methods. Figure 2.1.1 shows a section of a mesh of a mesh with triangular elements. The figure also defines other key mesh

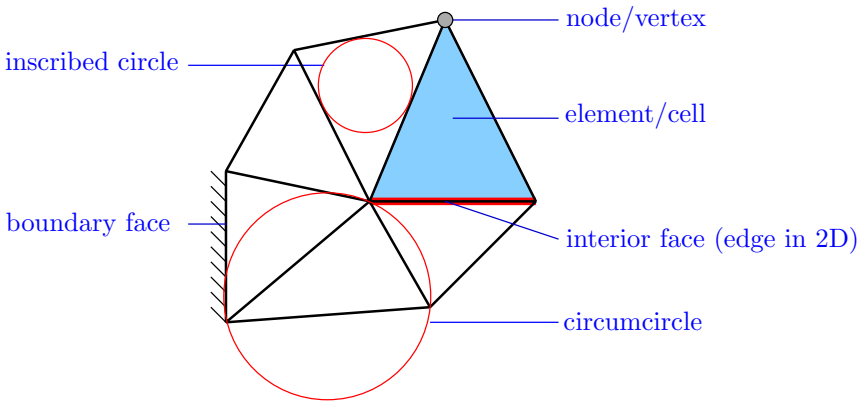


Figure 2.1.1: Definitions of mesh quantities.

constructs, including **nodes/vertices** and **interior/boundary faces/edges**. Note that in a single mesh, the elements need not be the same type. For example, in two dimensions,

meshes often contain both quadrilaterals and triangles, and such a mesh is called **hybrid**. Figure 2.1.2 defines the most commonly-used elements and their abbreviations.

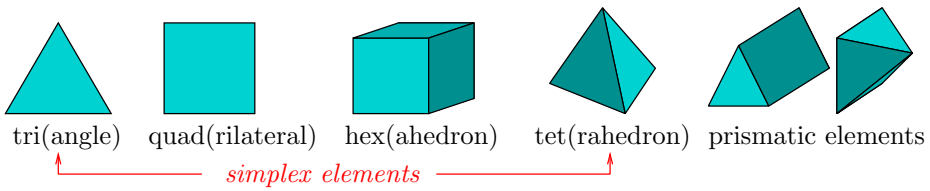
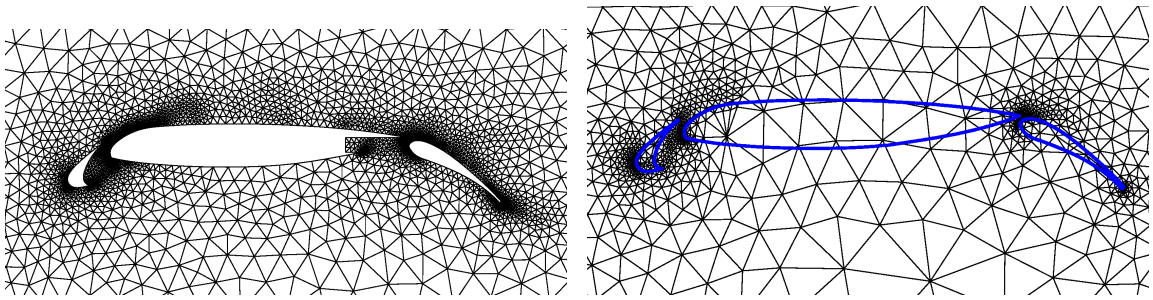


Figure 2.1.2: Mesh element types.

One high-level distinction between mesh types is whether they are **boundary-conforming** or **cut-cell**. Figure 2.1.3 illustrates the difference: namely that boundary-conforming meshes only fill the computational domain (e.g. the area where air flows over an airfoil), whereas cut-cell meshes disregard the boundaries of the domain. The motivation for cut-cells is that



(a) Boundary-conforming mesh (b) Cut-cell mesh

Figure 2.1.3: Boundary conforming and cut-cell meshes.

mesh generation around complex geometries is not trivial, and sometimes it is easier to "mesh through" the boundary of the domain. This puts an additional burden on the solver to be able to handle meshes where the computational domain boundary cuts through the elements. However, in some cases, this burden is worthwhile, e.g. for very intricate geometries or moving components where the geometry changes often.

Another distinction between meshes, particularly common in the finite-volume community, is whether a **primal** or **dual** mesh is used by the solver. A primal mesh consists of the cells that logically make up the subdivision of the domain. For example, the triangular mesh shown in black in Figure 2.1.4 is a primal mesh. The corresponding dual mesh derives from the primal mesh by connecting the centroids of the primal cells. This encloses the primal mesh nodes with new volumes, called dual cells. That is, each dual cell corresponds to one node of the primal mesh. Some numerical methods work on the dual mesh instead of the primal mesh, and one example is **node-centered** finite-volume methods, in which the control volumes are the dual cells, as opposed to the primal cells in a **cell-centered**

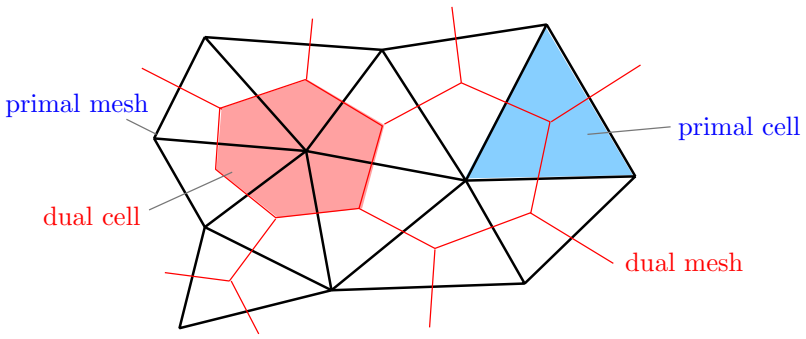


Figure 2.1.4: Primal and dual meshes.

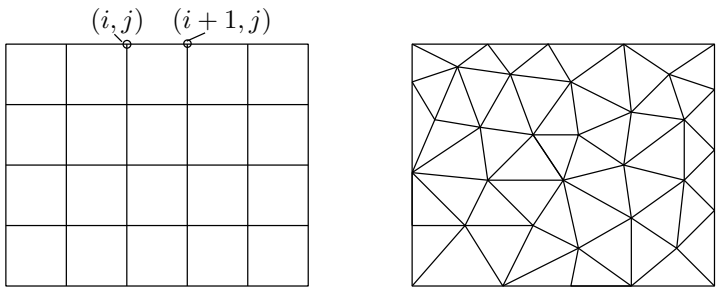
finite-volume method. A counting argument on a regular mesh (6 cells around each node) shows that there are approximately a factor of two fewer nodes than triangles in a triangular mesh. This translates to fewer unknowns (control-volume averages) for a similar resolution, at a cost of more inter-cell flux calculations.

The interior structure of elements in a mesh can also be classified as conforming or non-conforming. In a **conforming** mesh, every element is adjacent to only one neighbor element across each face. In contrast, in a **non-conforming** mesh, two distinct elements may exist adjacent to a single face of another element, as shown in Figure 2.1.5. A non-conforming

Figure 2.1.5: Conforming and non-conforming meshes.

mesh is often useful for adaptive refinement, since resolution can be added locally to one area without affecting quality of the neighboring elements.

Finally, a mesh type distinction that is critical for solvers is structured versus unstructured. In a **structured** mesh, the connectivity between nodes/elements is implied. As shown in Figure 2.1.6, the node to the right of node indexed by (i, j) is node $(i+1, j)$ – that is, incrementing the first index of a node identifier gives the index pair of the node to the right. Similar rules can be constructed for nodes above/below and to the left of the node. As such, the connectivity information does not have to be explicitly stored. In contrast, in an **unstructured** mesh, the connectivity between nodes and elements is not known ahead of time and must be stored. For example, in the unstructured mesh shown in Figure 2.1.6, there is no intuitive or systematic rule for ordering the elements such that neighbors can be identified by simple manipulation of indices. Instead, the neighbors of each element (or node) must be stored in some form. This storage adds to the overhead of an unstructured mesh, but for complex geometries, such an overhead is generally acceptable given the mesh



(a) Structured (b) Unstructured

Figure 2.1.6: Structured and unstructured mesh types.

resolution flexibility offered by unstructured meshes. Specifically, unstructured meshes allow for *local* resolution changes: finer elements can be placed in arbitrary regions of the domain without much effect on the surrounding element sizes. In contrast, in a structured mesh, additional mesh resolution can only be added in increments of entire rows or columns of elements.

2.2 Mesh Storage and Processing

As mentioned in the previous section, the connectivity information in an unstructured mesh must be stored. This storage of connectivity is not unique, and much of the information required by a particular solver can be derived from a few key pieces of connectivity data. One such complete set consist of:

1. Node coordinates $[N_n \times 2]$,
2. Node numbers for each element $[N_e \times 3]$,
3. Node numbers for each boundary face if multiple boundaries present $[N_{bf} \times 2]$.

The quantities in square brackets indicate the storage requirements for a two-dimensional triangular mesh, where N_n is the number of nodes, N_e is the number of elements, and N_{bf} is the number of boundary faces. This is one choice of "minimalistic" mesh information, and it is often used to store meshes. Figure 2.2.1 shows these matrices for a small sample mesh with eight elements and nine nodes. More information about the mesh is generally needed when solving a flow problem, or even when generating the mesh. This information can be derived from processing of the minimalistic mesh data. Note, however, that meshes in CFD may be quite big, with millions or billions of unknowns (i.e. mesh cells or nodes). To support these sizes, it is critical that mesh processing algorithms avoid $\mathcal{O}(N^2)$ complexity, where N is the number of elements or the number of nodes. Typically, all processing of mesh data can be performed using algorithms that scale linearly with the number of nodes or elements of the mesh.

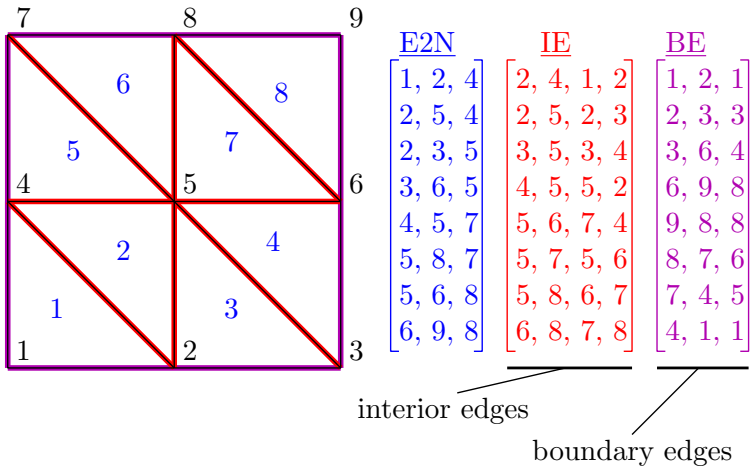


Figure 2.2.1: Sample mesh showing mesh connectivity matrices.

2.3 Unstructured Mesh Generation

We now review a few popular methods for creating unstructured meshes. These include

1. **Delaunay triangulation**: a method of connecting existing points into triangles, allowing for local addition or deletion of points.

2. **Advancing front**: the mesh is generated by propagating a front of simplex elements into domain interior from a boundary mesh. It is ideal for boundary layer meshes, where regularity near the boundary is required

3. **Grid and quadtree meshing**: an unstructured mesh is obtained by processing intersections between a Cartesian background meshes and the geometry of interest. Quad- or oct-tree storage can be used to efficiently add resolution only to areas where it is needed.

2.3.1 Algorithms

Delaunay Triangulation This method generates a unique triangulation of a set of arbitrary input points, with the key property that the circumcircle of any triangle contains no vertices other than the ones defining that triangle. Figure 2.3.1 illustrates this property. In addition, the Delaunay triangulation maximizes the minimum angle over all the triangles and extends to multiple dimensions. Furthermore, the dual mesh of a Delaunay triangulation, based on circumcenters of the triangles, becomes the Voronoi tessalation

Several algorithms are available for generating Delaunay meshes. These include:

1. Brute force: begin with any triangulation of the given points, and iteratively drive it to satisfy the Delaunay conditions. On each iteration, loop over all edges and flip the

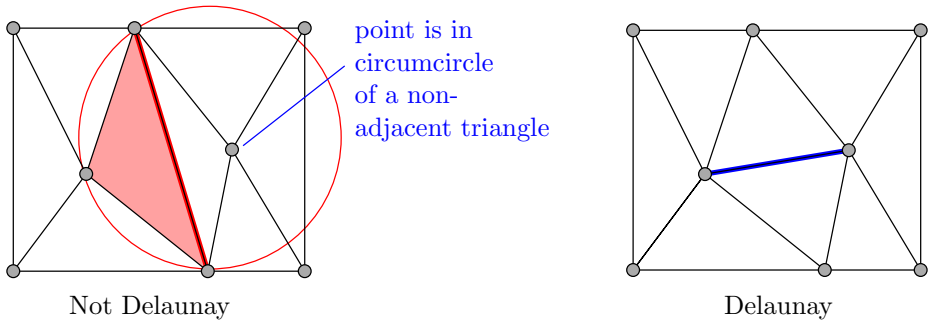


Figure 2.3.1: The Delaunay property.

edges of all triangle pairs that are not Delaunay – that is, for which the sum of the opposite angles exceeds 180°. Figure 2.3.1 shows a sample swap for a pair of triangles that did not satisfy this property. The problem with this method is that it requires $\mathcal{O}(N^2)$ operations, where N is the number of triangles, which scales proportionately to the number of edges. This is because $\mathcal{O}(N)$ iterations may be required, and at each iteration $\mathcal{O}(N)$ edges need to be visited.

2. Incremental node insertion: given a Delaunay triangulation of $N-1$ points, insert the N^{th} point in a desired region, e.g. where resolution is required. Then re-triangulate a (small) region of affected triangles, e.g. using the Bowyer-Watson algorithm [23]. Figure 2.3.2 illustrates such a node-insertion approach. Since the re-triangulations do not have to consider the entire mesh, the complexity of this method is better than the brute force approach, and turns out to be $\mathcal{O}(N \log N)$.

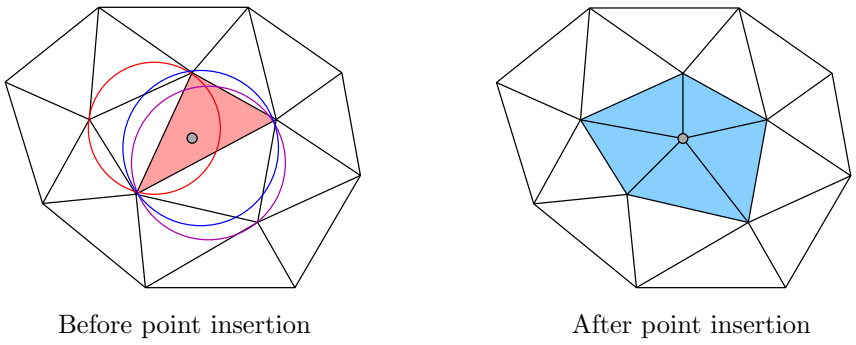


Figure 2.3.2: Delaunay refinement through point insertion and re-triangulation of a small region, identified by the triangles whose circumcircles enclose the inserted point.

3. Subdivision and merging: In this method, the set of points to be triangulated is split into two sets. Each set is triangulated separately and the two sets are then joined together. Applying the procedure recursively to the two sets results in an even better algorithmic complexity than the node insertion approach, at $\mathcal{O}(N \log \log N)$.

The node-insertion approach can also be used to adaptively improve mesh resolution, e.g. in a solution-based refinement. Such an adaptation of a Delaunay triangulation begins with picking a target "bad" triangle, which could be one that is too large, or that has undesirable angles, or for which some measure of the solution error is too high. Next, the centroid of that cell is added as the $N+1^{\text{th}}$ point. The affected region of triangles is then re-triangulated to satisfy the Delaunay property.

The Delaunay property ensures mesh uniqueness, but this is not always desired. Sometimes a mesh is desired with edges between certain points. For example, in multi-material simulations, there may be a required dividing line between two materials. Or if generating a mesh around an object, the required edges could represent the geometry boundary, so that after meshing the entire region, the interior portion could be easily removed. A Delaunay triangulation will not necessarily respect these edges, but the desired edges can be recovered relatively easily using edge swaps. Figure 2.3.3 illustrates this **constrained Delaunay** algorithm.

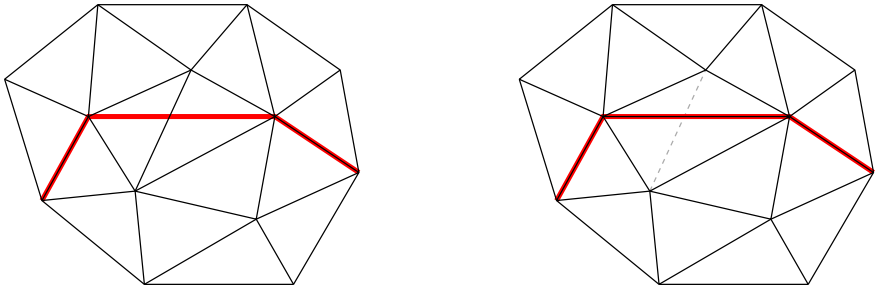


Figure 2.3.3: The constrained Delaunay algorithm: required edges are recovered through edge swaps.

Advancing Front In this approach [118, 89], the mesh is generated by starting on the computational domain boundary and working towards the interior. A **mesh front** is the set of all edges which are available at a given time to form a new triangular element. The algorithm begins by initializing the front with a distribution of nodes on the boundary. This is the *initial front* shown in Figure 2.3.4. Next, an edge is selected from the current front and a new triangle is generated by creating a new node or by adding connections to existing nodes. The desired mesh density drives the decision of whether and where a new node is added. The front is then updated with the new edges and the process repeats with the selection of a new edge. If no more edges are available, the front is empty, and the algorithm completes.

Grid and Quadree Intersection This algorithm starts with a subdivision of the *bounding box* of the computational domain with structured elements, such as squares in two dimen-

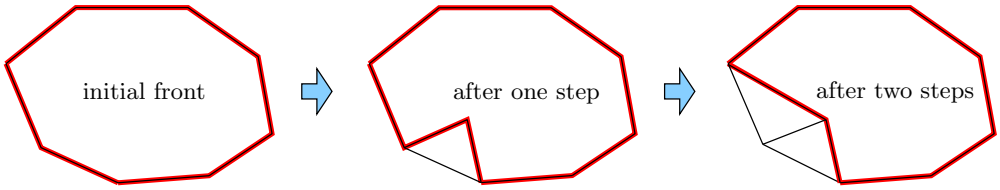


Figure 2.3.4: The advancing-front mesh-generation method.

sions or cubes in three dimensions. The bounding box refers to a box (e.g. square or cube) that encloses the computational domain. As this initial mesh does not respect the geometry boundary, the next step is to fix the mesh by separately considering the intersection of each square/cube with the boundary. An unstructured mesh is created in each case, as illustrated in Figure 2.3.5. Since the intersection of the geometry with a structured element

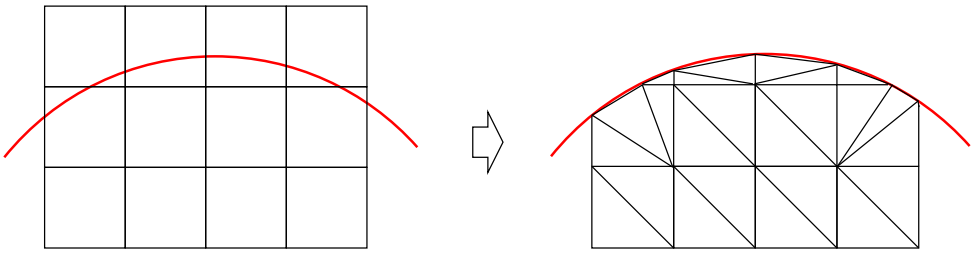


Figure 2.3.5: Cartesian grid-based mesh generation.

is relatively simple, or can be approximated as such, the individual meshing problems can be solved through the application of a few prototypical tessellations. If desired, the remainder of the structured mesh can then be converted into an unstructured mesh by dividing each square into triangles. Alternatively, a hybrid mesh could be employed with a structured set of elements away from the geometry. Since a structured Cartesian mesh has uniform mesh resolution, which may not be adequate for accurately representing the geometry, local hanging-node refinement can be used. Figure 2.3.6 illustrates such an approach, using a **quadtree**, which is a hierarchical data structure used for storing the hanging-node refinement levels. In three-dimensions, a similar structure is called an **octtree**, since each cube uniformly divides into eight cubes.

2.3.2 Size and Shape Control

Unstructured meshes offer the flexibility of efficiently controlling the size and shape of the elements. This section discusses some methods for changing mesh resolution, measuring quality, and making use of anisotropy.

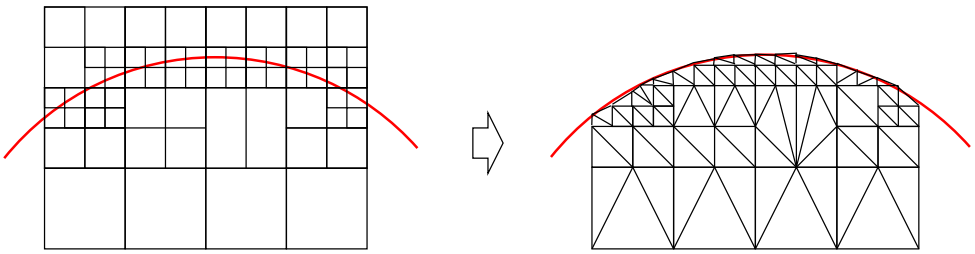


Figure 2.3.6: Quad-tree refinement of a grid-based mesh.

Unstructured Mesh Refinement One approach for changing the resolution of a computational mesh is the application of local mesh modification operators [54, 7, 112]. For triangular and tetrahedral meshes, local mesh modification operators consist of node insertion, face/edge swapping, edge collapsing, and node movement. Figure 2.3.7 illustrates these for triangles. The primary advantage of local operators is their robustness: the entire mesh is not regenerated all at once, but rather each operator affects only a prescribed number of nodes, edges, or elements. A disadvantage is that the changes are incremental, so that adding a lot of resolution or making other drastic mesh changes may require a large number of iterations.

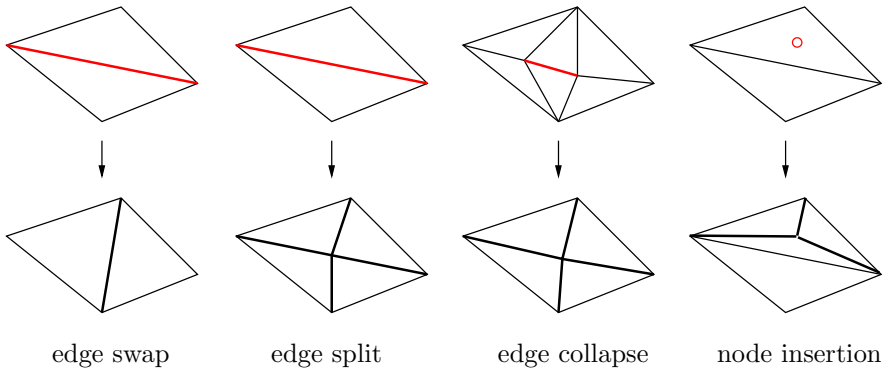


Figure 2.3.7: Local refinement and coarsening operators for triangles.

Mesh Quality Elements in unstructured meshes can take on almost arbitrary sizes and shapes. However, not all sizes and shapes may be desirable for a particular discretization. For example, explicit time marching methods must take small time steps in the presence of small elements, so unnecessarily small elements are undesirable (if possible, as often accuracy drives the size of the mesh). In addition to size, element shape can also have an impact on the solver robustness and the quality of the solution. One important characteristic of an element shape is its angles. In general,

- Large angles (close to 180^o) are problematic for most discretizations,

- Small angles (close to 0) are problematic for some discretizations.

These observations hold for many finite-volume and finite-element methods. A measure of quality, Q, that provides information similar to the smallest angle is

$$Q = \frac{4\sqrt{3}\,\text{Area}}{L_1^2 + L_2^2 + L_3^2} \quad \text{(1 for equilateral triangles)}$$

where L_i are the edge lengths. $Q = 1$ for an equilateral triangle, and for all other triangles, $Q < 1$. The smaller the smallest angle, the smaller the value of Q. Note, however, that not all discretizations are sensitive to just the smallest angle, and that in some cases small angles may be preferable for efficiency, e.g. in anisotropic boundary-layer meshes.

Numerous other quality measures have been created for mesh generation, incorporating mesh size limits, anisotropy measures, and adjacent-element area/volume ratios. Most of these measures do not incorporate properties of the solution, which ultimately provides the authoritative quality and size information for a mesh used to solve a particular problem.

Size Control The size of a mesh is an ambiguous term on several levels. First, some practitioners refer to the "mesh size" as a measure of the number of elements or nodes, so that a large mesh size refers to a mesh that has many, densely-packed, elements. Conversely, the word size can be used to describe some measure of the size, such as maximum length, of the elements. In this case, a large size refers to big elements, and correspondingly fewer of them for a given computational domain. We adopt this latter convention and use the letter h to denote the mesh size.

The **mesh size**, h, is a length measure of the size of the elements that constitute a mesh. Since a mesh consists of many elements, generally of different sizes, one could define h as specific to each element. On the other hand, one can refer to an average h of the entire mesh, a concept that is still useful when considering multiple meshes of the same family. For example, a sequence of meshes derived by uniform refinement can be referred to via h, $h/2$, $h/4$, etc.

Defining the size of an element is not unique, and several options are available. These include the diameter of the incircle or circumcircle, the shortest or longest edge length, the shortest or longest height of a triangle, the ratio of the element area to the perimeter (half of the hydraulic radius), etc. These measures do not uniquely specify an element, in that many different element shapes could have the same h. However, the goal of defining h is not to fully characterize the elements in a mesh, but rather to provide a simple measure for comparing meshes, especially those in the same family. This measure can be used to quantify a desired increase or decrease in resolution, e.g. $h \to h/2$ represents uniform refinement. h is also used in finite-element proofs related to the solution errors. For example, certain measures of the error in a finite-element solution approximation converge as $\mathcal{O}(h^{p+1})$, where $p+1$ is the formal order of accuracy of the numerical method.

When creating or adapting a mesh, a single measure of the mesh size, h, does not provide sufficient information for specifying the relative resolution requirements in different parts of the computational domain. Instead, we turn to a **mesh size field**, $h(\vec{x})$, which can be

used to provide information on the desired element size at every point $\vec{x}$ in the domain. This scalar field does not uniquely characterize the desired mesh, but it provides much more flexibility than a single quantity h. It is usually prescribed on a background mesh, which could be the current mesh in an adaptive sequence or a separate Cartesian grid. Information on setting $h(\vec{x})$ could come from a priori knowledge of the solution, geometry features, or a solution-based adaptive indicator.

Anisotropy and Metric-Based Meshing It may seem at first counter-intuitive, but stretched elements with small angles are in very important cases preferable over isotropic elements, such as nearly equilateral triangles. These stretched elements are characterized by a large **aspect ratio**, defined in Figure 2.3.8 for a triangle as the ratio of the longest to shortest edge in a fitted bounding box. Other definitions are also possible.

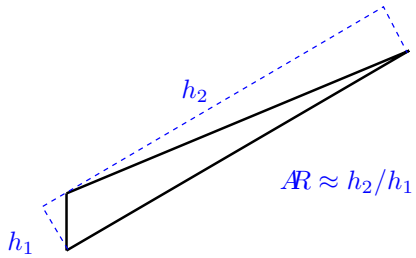


Figure 2.3.8: Aspect ratio of a triangular element.

Ansiotropic elements are needed when the solution varies significantly in one direction and relative to another, perpendicular direction. This occurs in flow features such as boundary layers, wakes, and shocks. For example, in a boundary layer, the wall-tangential velocity increases rapidly from zero at the wall to a large nonzero value close to the wall, whereas the flow quantities do not change much in the direction aligned with the wall. In such cases, element can be stretched so that the long side is aligned with the direction of little solution variation, while the short side is aligned with the direction of maximum variation.

When a solution changes a lot in only one direction, using isotropic elements would be very inefficient. For the same resolution in the important maximum-change direction, we would need $AR^{\text{dim}-1}$ times as many isotropic elements as anisotropic ones. Often, the aspect ratio can be $AR \sim \mathcal{O}(10^3)$ for high-Reynolds number simulations, so in three dimensions, the ratio of isotropic to anisotropic elements can reach $\mathcal{O}(10^6)$. Since boundary layers, wakes, and shocks consume the most elements (often 1/3 of the total element count), isotropic elements are not practical in such cases.

Anisotropic meshes can be produced using the advancing-front method, by growing a surface mesh away from the boundary using prismatic elements. This is the typical approach for generating boundary-layer meshes. A more general approach is fully-unstructured **metric-based meshing** [30]. The idea of metric-based meshing is to make a high-quality isotropic mesh, for example using Delaunay meshing, under a new length measure. The standard

Euclidian measure corresponds to an identity metric in defining the distance between two points separated by $\Delta\vec{x}$:

$$L_I^2 = \Delta\vec{x}^T \mathbf{I} \Delta\vec{x}.$$

In this equation, L_I is the length between the two points under the Euclidian measure, which is just $L_I = |\Delta\vec{x}|$. The Euclidian measure is what we're used to when we talk about the distance between points, but we don't have to measure distances this way. Instead, we could a "skewed" yardstick, in which the length measure depends on the direction. The most general case is an arbitrary metric, a dim $\times$ dim SPD matrix denoted by $\mathbf{M}$. It yields a distance measure of

$$L_M^2 = \Delta\vec{x}^T \mathbf{M} \Delta\vec{x}.$$

Note that the metric is sandwiched between the two $\Delta\vec{x}$ vectors. The role of this metric is to specify how we measure distances in different directions. L_M is called the metric measure, and if $\mathbf{M}$ is not the identity, then L_M is not the Euclidian distance between the points.

We can gain some insight into the metric by considering the set of points that are a unit metric distance away from a certain point. This unit measure requirement,

$$\Delta\vec{x}^T \mathbf{M} \Delta\vec{x} = 1,$$

describes an ellipsoid in physical space, as shown in Figure 2.3.9. The eigenvectors of $\mathbf{M}$, $\vec{e}_i$,

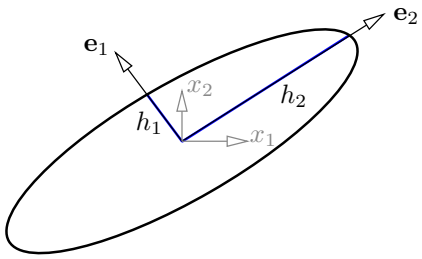
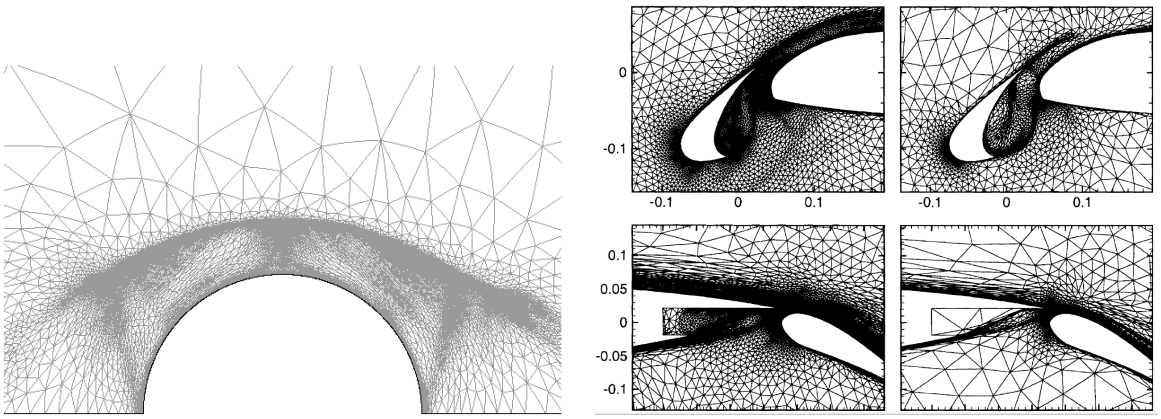


Figure 2.3.9: Ellipsoid demonstrating equal measure in metric space. All points on the ellipse are equidistant, under the metric measure, from the center of the ellipse.

are called the principal stretching directions. These correspond to the directions where the metric measure is smallest and largest. The eigenvalues of $\mathbf{M}$, λ_i, are related to the principal stretching lengths, h_i, via $h_i = 1/\sqrt{\lambda_i}$.

The metric tensor is useful for adaptive meshing, where one mesh is created from another based on some kind of error estimate. One such error estimate is based on the approximation of a scalar quantity. For example, using a second-order accurate method, the second derivatives of a scalar quantity measure the interpolation error in that quantity. A matrix formed from the second derivatives, including cross-derivatives, is called the **Hessian matrix**, and it can be interpreted as a scaled metric tensor [7]. Using the Hessian matrix as the metric

ensures that the interpolation error of a chosen scalar quantity is equidistributed throughout the domain. However, such equidistribution may not necessarily be the desired figure of merit, for example if we are more concerned with scalar output quantities. More sophisticated metric definitions based on measures of the output error exist, and these metrics are defined to produce the optimal element stretching for accurately predicting a chosen scalar output quantity [147, 47, 160]. Figure 2.3.10 shows some example meshes generated using anisotropic metric-based refinement.



(a) Cylinder in supersonic flow

(b) Multi-element airfoil

(c) Laminar-flow over a NACA 0012

(d) Laminar-flow over a NACA 0012 (cut-cell)

Figure 2.3.10: Sample anisotropic meshes.

Smoothing Mesh quality can be improved by **smoothing**, which refers to iteratively repositioning the internal nodes. The node movement is generally designed to improve some mesh quality measure. One simple technique is averaging of node coordinates with neighbors,

according to

$$\vec{x}_i' = \frac{1}{1+n_i}\left[\vec{x}_i + \sum_{j=1}^{n_i} \vec{x}_{N(i,j)}\right], \tag{2.3.1}$$

where n_i is the number of nodes adjacent to node i, and $N(i,j)$ is the global node number of the j^{th} neighbor to node i. Figure 2.3.11 illustrates the resulting mesh quality improvement. The initially-skewed set of triangles becomes more isotropic after application of this smoothing. Mesh smoothing generally improves quality, but this is not guaranteed. In some

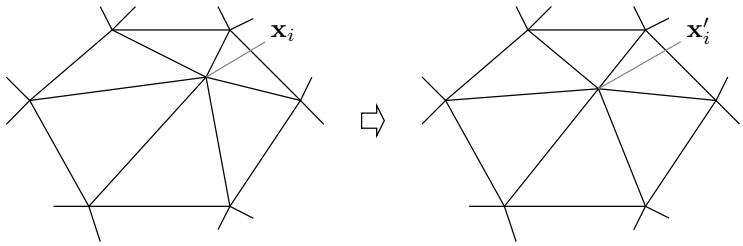


Figure 2.3.11: Mesh quality improvement through smoothing.

extreme cases, the smoothed mesh may even become invalid, as shown in Figure 2.3.12. However, this can be addressed by under-relaxing the coordinate update in Equation 2.3.1. Mesh smoothing is often used after quadtree or Delaunay meshing, as the initial node po-

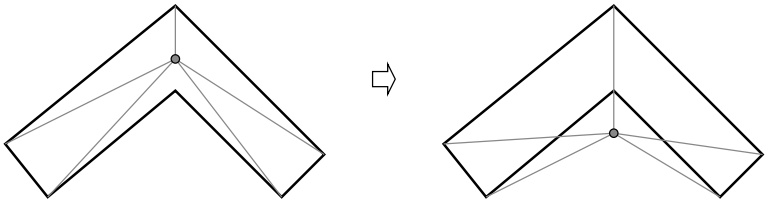


Figure 2.3.12: Invalid mesh produced by smoothing.

sitions in both of these methods may not be optimal. Many iterations may be required to improve quality to the desired level, and the termination criteria may be based on a fixed number of iterations or on a node movement tolerance.

2.4 Structured Mesh Generation

Recall that a structured mesh differs from an unstructured mesh in that the connectivity between elements or nodes is not explicitly stored in a structured mesh. This means that numerical methods based on structured meshes use less memory and can be faster, since neighbor look-ups are avoided. In addition, the regular grid pattern inherent to structured

meshes typically creates high-quality uniform elements, so that numerical methods using structured meshes tend to produce accurate results. The downsides include a lack of geometric flexibility, an inability to refine locally, and a typically user-intensive mesh generation process. However, for certain classes of problems these downsides may not be relevant, or the benefits may outweigh them.

2.4.1 Single-Block Topologies

A **block** of a structured grid is a section in which connectivity does not have to be explicitly stored. This is actually our working definition of a structured grid, but we will soon discuss multi-block grids, in which blocks can be arranged in an unstructured way, but which we still call structured because of the intra-block regularity.

A single block of a mesh is constructed by mapping a simple reference domain to the physical domain. The reference domain is usually a rectangle, as shown in Figure 2.4.1. In an

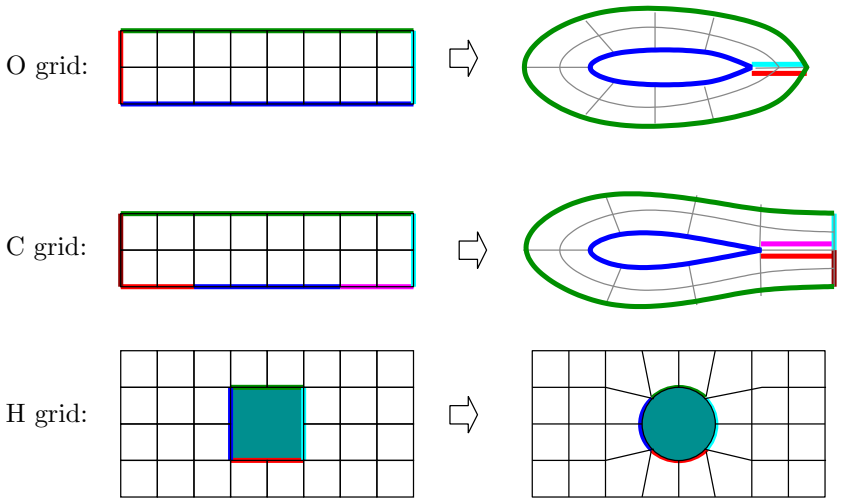


Figure 2.4.1: Single-block mesh topologies.

"O grid" and a "C grid", the mapped rectangle wraps around the object of interest, whereas in an "H grid", a hole cut out of the reference rectangle maps to the object boundary. O grids are most suitable for blunt objects requiring boundary-layer resolution. For airfoil shapes with sharp trailing edges, the O grid mapping at the trailing edge becomes highly skewed. In such cases. a C grid mapping performs better, as the mapped rectangle wraps around both the object and the wake. A C grid is therefore also useful for resolving the wake, assuming we know where it is. Figure 2.4.2 shows an example of a C grid for an airfoil. Note that boundary-layer and wake resolution can be controlled by adjusting the mesh stretching ratios. Finally, an H grid can produce high-quality elements near the object surface, and high mesh resolution can be achieved with only local refinement along each of the directions.

In general, single-block grids are limited in the types of geometries they can handle. In addition, though they offer some mesh size control, increasing resolution generally re-

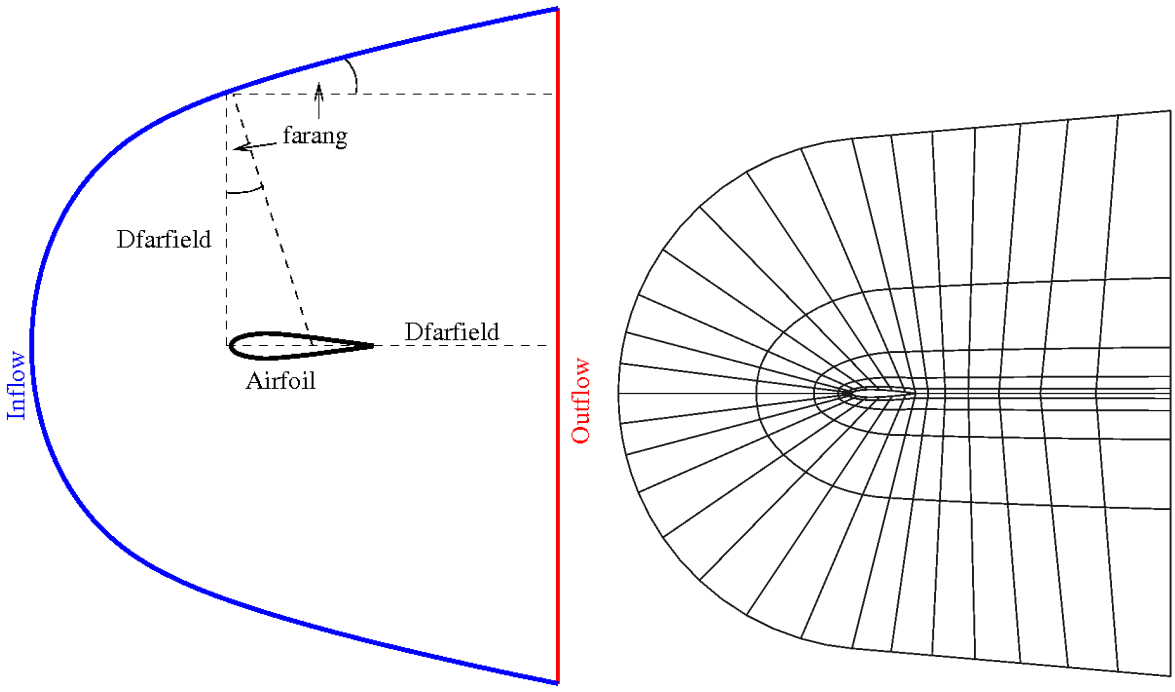


Figure 2.4.2: C-grid airfoil mesh generation.

quires non-local refinement, far from the object. For more geometric flexibility and meshing efficiency, we turn to multiblock meshes.

2.4.2 Multi-block Meshes

In a **multi-block mesh**, the computational domain is divided into a coarse "blocking" of quadrilaterals in two dimensions and hexahedra in three dimensions. Single-block mappings are used to mesh the interiors of each block. For matched multi-block meshes, the mesh nodes must be consistent along the interfaces between the blocks, as illustrated in Figure 2.4.3. Figure 2.4.4 shows more complicated examples of such meshes. Less typical are unmatched multi-block meshes, where each block can be meshed independently, leaving the task of handling non-conforming block interfaces to the solver.

Generating high-quality blockings is not easy. Indeed, a large portion of the CFD analysis process time is typically devoted to this task, which requires heavy user involvement to ensure high quality for complex geometries. One automation possibility is a blocking algorithm based on the medial-axis transform [55], which is illustrated in Figure 2.4.5. The utility and robustness of this algorithm remain to be investigated and demonstrated for complex three-dimensional geometries.

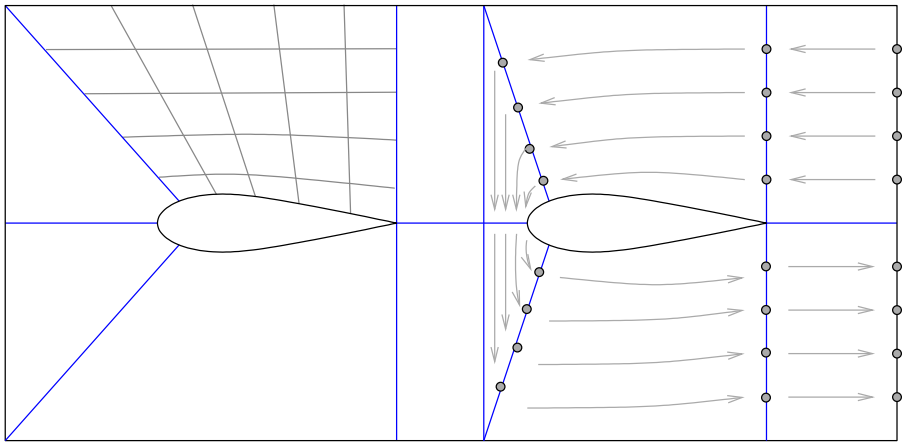


Figure 2.4.3: Generation of a matched multi-block mesh around an airfoil.

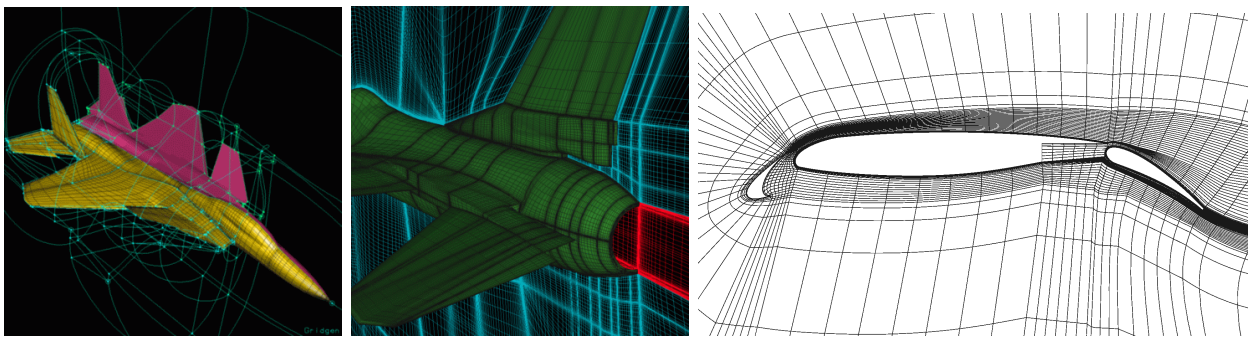


Figure 2.4.4: Sample multi-block meshes in two and three dimensions.

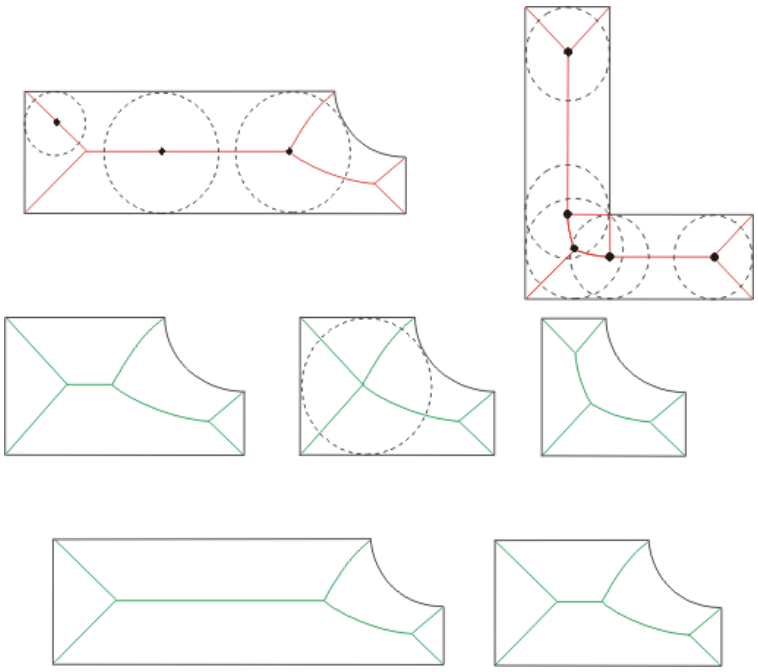


Figure 2.4.5: Blocking generation through the medial axis transform.

2.4.3 Mappings

In both single and multi-block meshes, a mapping is required from the reference domain to the physical domain. Several methods are available to construct such a mapping, including:

- **Conformal map**: an analytical mapping using complex numbers: generates high-quality orthogonal meshes, but limited to two dimensions.
- **Algebraic mapping**: a general and simple technique that can be used for propagating boundary maps to interiors. Grid quality is not guaranteed.
- **PDE-based map**: a widely-used approach that generates high-quality grids but requires solution of a partial differential equation.

Conformal Map A conformal map is defined by an analytic function of complex numbers, $z = f(\zeta)$, with $f'(\zeta) \neq 0$. Here, $z = x + iy$ is a complex number that encodes the two-dimensional coordinates in physical space, and $\zeta = \xi + i\eta$ is a complex number that encodes the coordinates in reference space.

An **analytic** function is one for which a unique derivative, f', exists. This is equivalent to saying that the individual coordinate maps, $x(\xi, \eta)$ and $y(\xi, \eta)$, must satisfy the Cauchy-Riemann equations and hence are harmonic,

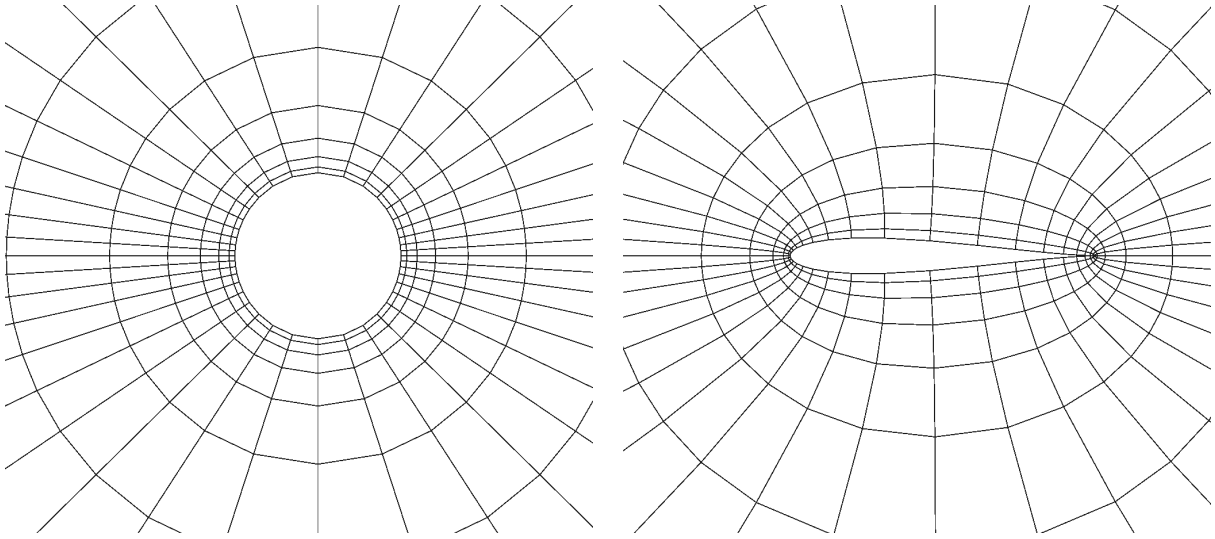
$$\frac{\partial^2 x}{\partial \xi^2} + \frac{\partial^2 x}{\partial \eta^2} = 0, \qquad \frac{\partial^2 y}{\partial \xi^2} + \frac{\partial^2 y}{\partial \eta^2} = 0.$$

A conformal map preserves angles and therefore yields high-quality meshes when the reference-domain mesh is high quality. An example map is the Joukowski transformation,

$$z = \zeta + \frac{1}{\zeta},$$

which maps a circle to a cusped airfoil. Specifically, the circle in reference space must pass through the point $\zeta = 1 + 0i$, which is a singular point of the derivative of the mapping in Equation 2.4.1. The image of this point at $z = 2 + 0i$ becomes the airfoil trailing edge.

Denote by ζ_0 the center of the circle in the ζ plane. The horizontal position of the center, i.e. $\Re(\zeta)$, controls the thickness of the resulting airfoil, while the vertical position, i.e. $\Im(\zeta)$, controls the airfoil camber. Given a mesh around a circle, which is straightforward to generate, we can obtain a mesh around the airfoil by applying the transformation in Equation 2.4.1 to all nodes in the mesh. Figure 2.4.6 shows a sample mesh generated in this way, starting from a grated/stretched mesh around the circle.



(a) Reference grated mesh around a circle in the reference ζ plane

(b) Mapped mesh around a Joukowski airfoil in the physical z plane

Figure 2.4.6: Joukowski-transform mesh generation. Note that 90° angles are preserved

Joukowski transformations are limited to generating airfoils with cusped (zero angle) trailing edges. A Karman-Trefftz mapping generates an airfoil with a nonzero trailing edge angle. It is given by

$$\frac{z-n}{z+n} = \left(\frac{\zeta - 1}{\zeta + 1}\right)^n, \tag{2.4.1}$$

where n controls the trailing-edge angle. Even more general is the set of Schwarz-Christoffel mappings, which map a regular polygon into a half plane. These have been extended to multiply-connected domains [28], for use in meshing more complex geometries.

Algebraic Map An algebraic mapping is given by defining functions that produce physical-space coordinates, (x, y) given reference-space coordinates, ξ, η. Thus, every conformal map is an algebraic map, but not the other way around, since general algebraic mappings need not use harmonic functions.

Figure 2.4.7 shows a simple functional mapping in which the vertical coordinate remains unchanged, whereas the horizontal coordinate is scaled in a non-uniform way that depends linearly on the vertical coordinate, η.

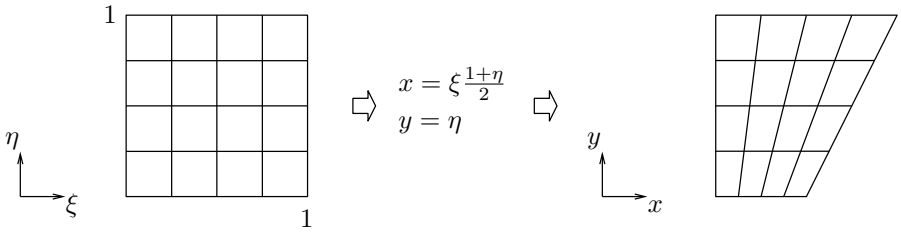


Figure 2.4.7: An algebraic function mapping.

The general setting for applying algebraic maps is one in which we know the location of the boundaries of the domain (e.g. from the geometry), and we need to mesh the interior. Figure 2.4.8 illustrates this situation in two dimensions.

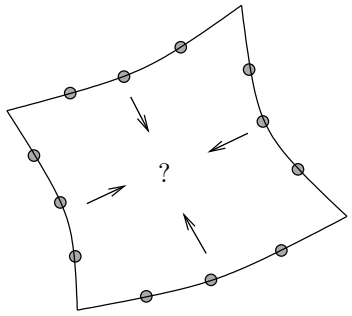


Figure 2.4.8: Boundary map interpolation requires superimposition of boundary displacement requirements.

Before presenting the general case, we consider a simplified example of meshing a variable-height channel, illustrated in Figure 2.4.9. Given coordinates of the lower and upper boundaries, i.e. node coordinates y_{i1} and y_{iN} (the first index is the horizontal position, and the second index is the vertical position), the goal is to generate a set of points inside the channel. We do this by choosing a reference domain for which the horizontal coordinate is unchanged, $\xi = x$, and for which the vertical coordinate ranges from $\eta = 0$ to $\eta = 1$. Node (i, j) in the mesh has a reference vertical coordinate of η_{ij}. The corresponding physical vertical coordinate, $y_{i,j}$ is given by the linear interpolation of the two coordinates: $y_{i,1}$ and y_{iN}, as shown in Figure 2.4.9.

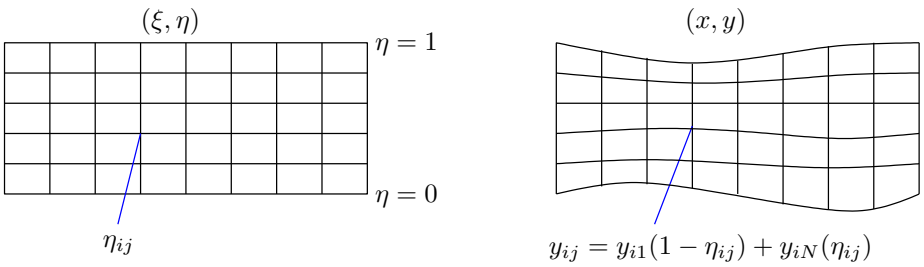


Figure 2.4.9: Algebraic mapping applied to a variable-height channel.

The channel example is simple because only the lower and upper boundary locations vary. In a general case, all boundaries may be arbitrary functions. To handle this case, we turn to **transfinite interpolation**. This is a technique for mapping boundary points to interior points. The physical domain under consideration must be topologically equivalent to a square in two dimensions, or a cube in three dimensions. The reference domain is then taken as a unit square or unit cube. The method is fast and general, though it does not guarantee properties such as orthogonality, or even validity.

The goal of transfinite interpolation is to determine the physical-space coordinates of interior points, $\vec{x}(\xi, \eta)$, given the physical-space locations of boundary points, as shown in Figure 2.4.10.

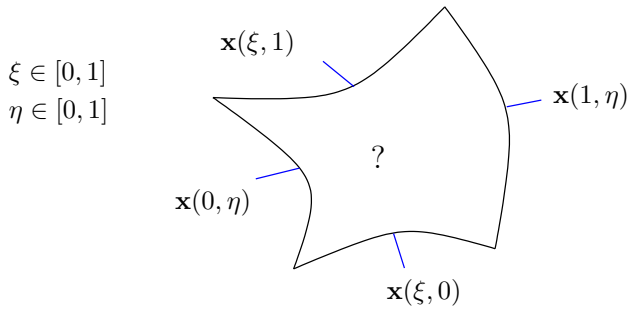
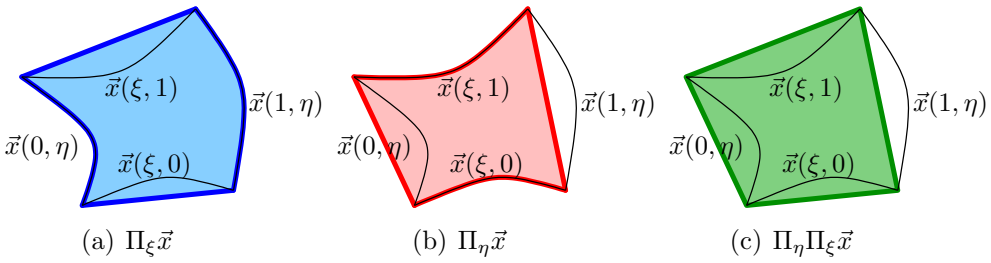


Figure 2.4.10: Boundary data for two-dimensional transfinite interpolation.

We approach the two-dimensional interpolation problem, by first defining one-dimensional edge interpolants, as shown in Figure 2.4.11. The physical-space points denoted by $\Pi_\xi \vec{x}$ linearly interpolate the left and right boundaries of the domain, without regard to the locations of the top and bottom boundaries. Conversely, the points $\Pi_\eta \vec{x}$ linearly interpolate the bottom and top boundaries of the domain, without regard to the locations of the left and right boundaries. Finally, the set of points $\Pi_\eta \Pi_\xi \vec{x}$ is a bilinear interpolation of the corner nodes –



(a) $\Pi_\xi \vec{x}$ (b) $\Pi_\eta \vec{x}$ (c) $\Pi_\eta \Pi_\xi \vec{x}$

Figure 2.4.11: One-dimensional edge interpolants and a bilinear interpolation of the four nodes.

these points does not respect any of the four boundaries. In equations, the interpolants are

$$\Pi_\xi \vec{x} \equiv (1-\xi)\vec{x}(0,\eta) + \xi \vec{x}(1,\eta) \tag{2.4.2}$$

$$\Pi_\eta \vec{x} \equiv (1-\eta)\vec{x}(\xi,0) + \eta \vec{x}(\xi,1) \tag{2.4.3}$$

$$\Pi_\eta \Pi_\xi \vec{x} \equiv (1-\xi)(1-\eta)\vec{x}(0,0) + (1-\xi)\eta \vec{x}(0,1) + \xi(1-\eta)\vec{x}(1,0) + \xi\eta \vec{x}(1,1) \tag{2.4.4}$$

The transfinite interpolation operator is given by the boolean sum of the two edge interpolation operators,

$$\text{Interior } \vec{x} = (\Pi_\xi \oplus \Pi_\eta)\, \vec{x} = (\Pi_\xi + \Pi_\eta - \Pi_\eta \Pi_\xi)\, \vec{x}. \tag{2.4.5}$$

Note that the boolean sum requires the product map $\Pi_\eta \Pi_\xi$, to take out the duplicate interpolation of the nodes between the two maps. Figure 2.4.12 illustrates this boolean sum graphically. Intuitively, the interior map formed by the boolean sum is a blended combination

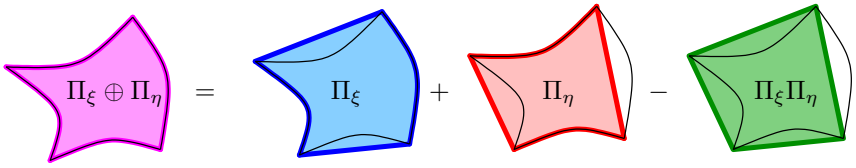


Figure 2.4.12: Transfinite interpolation as a boolean sum of edge interpolants.

of the edge maps that does not disturb the node locations.

Figure 2.4.13 shows a snippet of C-code for computing the transfinite interpolation in two dimensions. In this code, the reference-space coordinates are $X = \xi, Y = \eta$, $x[9][2]$ contains physical-space coordinates on the boundaries, and $R[k]$ is constructed to contain the interpolation weights on the other nodes.

PDE-based mapping In PDE-based mappings, we seek functions $\xi(x,y)$ and $\eta(x,y)$ that satisfy the equations

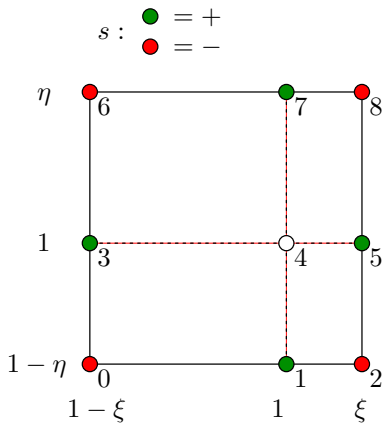
$$\nabla^2 \xi(x,y) = P(x,y), \tag{2.4.6}$$

$$\nabla^2 \eta(x,y) = Q(x,y), \tag{2.4.7}$$

```
// construct interpolation coefficients
XV[0] = 1.-X; XV[1] = 1.; XV[2] = X;
YV[0] = 1.-Y; YV[1] = 1.; YV[2] = Y;
for (j=0, k=0; j<3; j++)
  for (i=0; i<3; i++)
    R[k++] = YV[j]*XV[i];
R[4] = 0.;

// interpolate center node
for (d=0; d<dim; d++)
  for (k=0, x[4][d]=0.0, s=-1; k<9; k++, s=-s)
    x[4][d] += R[k]*x[k][d]*s;
```

(a) Code snippet



(b) Participating nodes

Figure 2.4.13: Code for a two-dimensional transfinite interpolation.

where P and Q are used for grid size control. Note, that the Laplace operator is with respect to the physical coordinates, x and y. Assuming an invertible mapping allows us to transform these equations to ones for x and y in terms of ξ and η,

$$ax_{\xi\xi} - 2bx_{\xi\eta} + cx_{\eta\eta} = 0, \tag{2.4.8}$$

$$ay_{\xi\xi} - 2by_{\xi\eta} + cy_{\eta\eta} = 0. \tag{2.4.9}$$

where

$$a = x_\eta^2 + y_\eta^2, \quad b = x_\xi x_\eta + y_\xi y_\eta, \quad c = x_\xi^2 + y_\xi^2. \tag{2.4.10}$$

These nonlinear Thompsons's equations are typically solved using an iterative method (e.g. successive over-relaxation). For example, a finite-difference discretization on a regular grid in (ξ, η) yields the desired (x, y) coordinates of a transformed mesh.

Chapter 3

The Finite Volume Method

3.1 Conservation Laws

A system of conservation laws in multiple dimensions can be written in the following general, differential form,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}} = \mathbf{0}, \tag{3.1.1}$$

where

- $\mathbf{u} \in \mathbb{R}^s$ is the conservative state vector with s components.
- $\vec{\mathbf{F}} = \vec{\mathbf{F}}(\mathbf{u}) \in [\mathbb{R}^s]^{\text{dim}}$ is a flux that tells us the direction and rate at which the conserved quantities pass through space. Note that dim is the number of spatial dimensions, and the arrow in $\vec{\mathbf{F}}$ denotes a spatial vector. The units of each flux are conserved quantity per unit area per unit time.

For example, linear advection for a conserved scalar quantity, u, in a prescribed velocity field $\vec{V}$, reads

$$\frac{\partial u}{\partial t} + \nabla \cdot \underbrace{\left(\vec{V} u\right)}_{\vec{F}} = 0. \tag{3.1.2}$$

That is, the flux in this case is $\vec{F} = \vec{V} u$.

The differential form in Equation 3.1.1 derives from an *integral form*, which we can recover by integrating Equation 3.1.1 over an area A[1]. At present, we restrict our attention to two spatial dimensions. Referring to the definitions in Figure 3.1.1, the integral form of

[1] in three dimensions, this would be a volume

Equation 3.1.1 is

$$\begin{aligned}
\int_A \left[\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}\right] dA &= \mathbf{0} \\
\int_A \frac{\partial \mathbf{u}}{\partial t} dA + \underbrace{\int_A \nabla \cdot \vec{\mathbf{F}} dA}_{\text{use divergence thm.}} &= \mathbf{0} \\
\int_A \frac{\partial \mathbf{u}}{\partial t} dA + \oint_{\partial A} \vec{\mathbf{F}} \cdot \vec{n}\, dl &= \mathbf{0} \qquad (3.1.3)
\end{aligned}$$

Note that in Equation 3.1.3 the normal vector $\vec{n}$ points outward from the enclosed area.

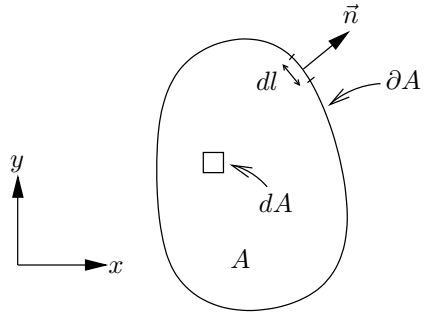


Figure 3.1.1: Definitions for application of the divergence (Gauss) theorem in two dimensions.

3.1.1 The Euler Equations

The compressible Euler equations of gas dynamics govern the flow of a perfect inviscid gas. They derive from the vanishing-viscosity (high-Reynolds number) limit of the compressible Navier-Stokes equations. The equations encode conservation statements for mass, momentum, and energy. In two spatial dimensions, the Euler equations are

$$\begin{aligned}
&\frac{\partial}{\partial t}(\rho) + \frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\rho v) = 0 && \text{(mass)} \\
&\frac{\partial}{\partial t}(\rho u) + \frac{\partial}{\partial x}(\rho u^2 + p) + \frac{\partial}{\partial y}(\rho uv) = 0 && (x\text{-momentum}) \\
&\frac{\partial}{\partial t}(\rho v) + \frac{\partial}{\partial x}(\rho vu) + \frac{\partial}{\partial y}(\rho v^2 + p) = 0 && (y\text{-momentum}) \\
&\frac{\partial}{\partial t}(\rho E) + \frac{\partial}{\partial x}(\rho uH) + \frac{\partial}{\partial y}(\rho vH) = 0 && \text{(energy)}
\end{aligned}$$

The variables appearing in this system of equations are defined as follows:

$$
\begin{array}{lcl}
\rho & = & \text{density} \\
u, v & = & x, y \text{ components of velocity, } \vec{v} \\
E & = & \text{total energy per unit mass} \\
e & = & \text{internal energy per unit mass} \\
H & = & \text{total enthalpy per unit mass} \\
h & = & \text{internal enthalpy per unit mass} \\
p & = & \text{pressure} \\
R & = & \text{gas constant for air, } c_p - c_v \\
\gamma & = & \text{ratio of specific heats, } c_p/c_v = 1.4 \\
c_p & = & \text{specific heat at constant pressure} \\
c_v & = & \text{specific heat at constant volume} \\
c & = & \text{speed of sound} \\
M & = & \text{Mach number}
\end{array}
$$

Calorically-perfect gas:

$$
\begin{aligned}
e &= c_v T \\
h &= c_p T \\
p &= \rho R T \\
&= (\gamma - 1) \underbrace{\left[\rho E - \frac{1}{2} \rho |\vec{v}|^2 \right]}_{\rho e} \\
H &= E + p/\rho \\
M &= |\vec{v}|/c = \sqrt{u^2 + v^2}/c \\
c &= \sqrt{\gamma R T} = \sqrt{\gamma p/\rho} \\
q &= \sqrt{u^2 + v^2}
\end{aligned}
$$

The Euler equations can be written in the compact form of Equation 3.1.1,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}(\mathbf{u}) = \mathbf{0},$$

by defining the state and flux vectors as

$$\text{state:} \quad \mathbf{u} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \qquad \text{flux:} \quad \vec{\mathbf{F}} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{bmatrix} \hat{x} + \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vH \end{bmatrix} \hat{y}.$$

Note that the spatial nature of the vector $\vec{\mathbf{F}}$ is evident from the two components, one along $\hat{x}$ and the other along $\hat{y}$. Each of these components is a rank $s = 4$ vector.

3.2 The Finite Volume Discretization

We now present the two-dimensional finite volume method: a technique for numerically obtaining an approximate solution to the conservation law. The starting point is the integral form in Equation 3.1.3, which we apply repeatedly to cells of the computational mesh. This is the essential part of the finite-volume method. We define the *cell average* of the state as

$$\text{cell average on cell } i \ = \mathbf{u}_i \equiv \frac{1}{A_i} \int_{A_i} \mathbf{u} \, dA,$$

where A_i refers to the area of cell i. Eqn. 3.1.3 in terms of cell averages becomes

$$\boxed{A_i \frac{d\mathbf{u}_i}{dt} + \oint_{\partial A_i} \vec{\mathbf{F}} \cdot \vec{n} \, dl = \mathbf{0}} \tag{3.2.1}$$

The cell averages are the unknowns in our discretization.

3.2.1 Scalar Flux Residual

Consider a finite volume method for a scalar problem in two dimensions. The PDE is given by Equation 3.1.2, and we use Equation 3.2.1 for our discretization, noting that the cells can be of various shapes. Since the shape of the cells does not change the key concepts, we focus our attention on triangles.

Figure 3.2.1 shows an example of a triangular cell with its nearest neighbors. Equation 3.2.1 applied to this cell, for a scalar problem, yields

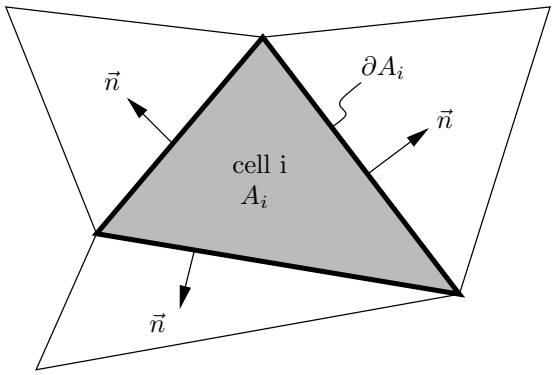


Figure 3.2.1: A triangular cell, indexed by i, in two dimensions. The edges are straight and the normal is outward-pointing and constant on each edge.

$$A_i \frac{du_i}{dt} \underbrace{+ \oint_{\partial A_i} \vec{F} \cdot \vec{n} \, dl}_{R_i} = 0, \tag{3.2.2}$$

where R_i is the flux residual. The calculation of the residual requires knowing the flux on the edges in between cells, and this is not something we have direct access to when all we store is the cell averages. This is the point where the approximation comes in: we must define a *numerical flux* that approximates the flux on the edges from an incomplete description of the state – just the cell averages. The residual calculation specifically requires only the flux dotted with the edge normal, so we approximate this directly:

$$\text{normal flux on edge } e \text{ between elements } L \text{ and } R \; = \vec{F} \cdot \vec{n} \approx \hat{F}(u_L, u_R, \vec{n}), \tag{3.2.3}$$

where L and R denote two adjacent elements with $\vec{n}$ the normal on their common edge, pointing from L to R as shown in Figure 3.2.2. To approximate an integral over the edge, we assume that either (1) the flux is constant along the edge, which is valid for first-order methods, in which the state is also assumed constant inside each cell; or (2) that the flux varies linearly along the edge, which is needed for second-order-accurate methods. In either case, the integral can be approximated as $\hat{F}$ multiplied by the edge length, with $\hat{F}$ evaluated

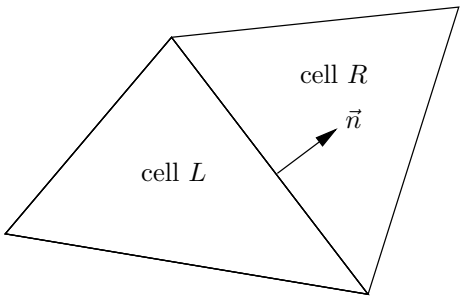


Figure 3.2.2: Two elements sharing a common edge. The L versus R designation is arbitrary, and the only requirement is that $\vec{n}$ points from L to R.

on the edge midpoint in case (2) for a single-point quadrature formula that handles linear variation. The flux residual in Equation 3.2.2 is then

$$R_i = \sum_{e=1}^{3} \hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e})\, \Delta l_{i,e}, \tag{3.2.4}$$

where, referring to Figure 3.2.1, e is an index over the three edges, $N(i,e)$ is the element adjacent to cell i across edge e, $\vec{n}_{i,e}$ is the outward-pointing normal vector from cell i on edge e, and $\Delta l_{i,e}$ is the length of edge e in cell i. Figure 3.2.3 illustrates these definitions.

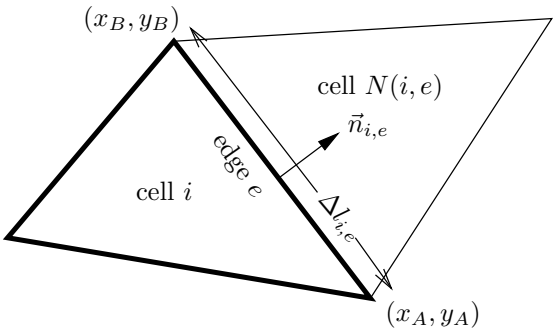


Figure 3.2.3: Definitions of edges and normals for a triangle.

For an edge between two points (x_A, y_A) and (x_B, y_B) the length and normal are

$$\begin{aligned} \Delta l_{i,e} &= \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}, \\ \vec{n}_{i,e} &= \left[(y_B - y_A)\hat{x} + (x_A - x_B)\hat{y}\right] / \Delta l_{i,e}. \end{aligned}$$

Note that this equation for the normal requires a counter-clockwise ordering of the nodes (i.e. A and B) relative to cell i.

$\hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e})$ is the flux function that tells us, in our initial scalar example, how much of u is flowing from cell i to the neighbor cell, $N(i,e)$, through edge e. For stability, we use

the *upwind* flux, where the upwinding decision is made based on the normal component of the velocity field, $\vec{V} \cdot \vec{n}_{i,e}$. That is, we write,

$$\hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e}) = \begin{cases} \vec{V} \cdot \vec{n}_{i,e} u_i & \text{for } \vec{V} \cdot \vec{n}_{i,e} \geq 0 \\ \vec{V} \cdot \vec{n}_{i,e} u_{N(i,e)} & \text{for } \vec{V} \cdot \vec{n}_{i,e} < 0 \end{cases} .$$

We can write this as one equation by using absolute values,

$$\hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e}) = \frac{1}{2} \vec{V} \cdot \vec{n}_{i,e} \left(u_i + u_{N(i,e)} \right) - \frac{1}{2} |\vec{V} \cdot \vec{n}_{i,e}| \left(u_{N(i,e)} - u_i \right) .$$

3.3 Euler Fluxes and Boundary Conditions

The finite volume method extends naturally to systems of nonlinear conservation laws, such as the Euler equations of gas dynamics. The key differences compared to our linear scalar equation is in the definition of the flux function at cell interfaces, and in the wider range of possible boundary conditions.

3.3.1 Riemann Solvers

We saw in the scalar case that the flux function must rely on the "upwind" state in order to faithfully model the convection physics. For hyperbolic systems of conservation laws, we must extend this idea to multiple waves: in effect, we have to calculate a flux that for each wave uses the correct upwind state.

A relatively simple flux function for systems is the **local Lax-Friedrichs**, or **Rusanov** flux. Given two states $\mathbf{u}_L$ and $\mathbf{u}_R$, the flux function is

$$\hat{\mathbf{F}} = \frac{1}{2} \left(\mathbf{F}_L + \mathbf{F}_R \right) - \frac{1}{2} s_{\max} \left(\mathbf{u}_R - \mathbf{u}_L \right) , \tag{3.3.1}$$

where $\mathbf{F}_L = \mathbf{F}(\mathbf{u}_L)$, $\mathbf{F}_R = \mathbf{F}(\mathbf{u}_R)$, and $s_{\max}$ is the maximum speed of any wave in the system, considering both $\mathbf{u}_L$ and $\mathbf{u}_R$. The goal of using $s_{\max}$ is to upwind every wave in the system, although for systems with different wave speeds, this flux will upwind too much thereby creating a significant amount of numerical diffusion.

An alternative flux that more carefully upwinds waves one by one is the **Roe** flux, given by

$$\hat{\mathbf{F}} = \frac{1}{2} \left(\mathbf{F}_L + \mathbf{F}_R \right) - \frac{1}{2} \left| \frac{\partial \mathbf{F}}{\partial \mathbf{u}} (\mathbf{u}^*) \right| \left(\mathbf{u}_R - \mathbf{u}_L \right) , \tag{3.3.2}$$

where $\left| \frac{\partial \mathbf{F}}{\partial \mathbf{u}} (\mathbf{u}^*) \right|$ refers to taking the absolute values of the eigenvalues, i.e. $\mathbf{R}|\mathbf{\Lambda}|\mathbf{L}$, in the eigenvalue decomposition. $\mathbf{u}^*$ is an appropriately-chosen intermediate state based on $\mathbf{u}_L$ and $\mathbf{u}_R$. This choice is only important for nonlinear problems, and the Roe flux uses the

Roe-average state, a choice that yields exact single-wave solutions to the Riemann problem. Specifically, the Roe state for the Euler equations is

$$\vec{v} = \frac{\sqrt{\rho}_L \vec{v}_L + \sqrt{\rho}_R \vec{v}_R}{\sqrt{\rho}_L + \sqrt{\rho}_R}, \quad H = \frac{\sqrt{\rho}_L H_L + \sqrt{\rho}_R H_R}{\sqrt{\rho}_L + \sqrt{\rho}_R},$$

and $\mathbf{A}(\mathbf{U})$ is the flux Jacobian,

$$\mathbf{A}(\mathbf{U}) = \frac{\partial \mathbf{F}}{\partial \mathbf{U}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -u^2 + \frac{\gamma-1}{2}q^2 & (3-\gamma)u & -(\gamma-1)v & \gamma-1 \\ -uv & v & u & 0 \\ \frac{\gamma-2}{2}uq^2 - \frac{uc^2}{\gamma-1} & H-(\gamma-1)u^2 & -(\gamma-1)uv & \gamma u \end{bmatrix}.$$

$|\mathbf{A}(\mathbf{U})|$ indicates absolute value signs around the eigenvalues,

$$|\mathbf{A}| = \mathbf{R}|\mathbf{\Lambda}|\mathbf{L},$$

where the right ($\mathbf{R} = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4]$) and left ($\mathbf{L} = \mathbf{R}^{-1} = [\mathbf{l}_1, \mathbf{l}_2, \mathbf{l}_3, \mathbf{l}_4]^T$) eigenvectors are

$$\mathbf{R} = \left[\begin{array}{c|c|c|c} 1 & 1 & 0 & 1 \\ u+c & u-c & 0 & u \\ v & v & v & v \\ H+uc & H-uc & v^2 & \frac{1}{2}q^2 \end{array}\right], \quad \mathbf{L} = \frac{\gamma-1}{2c^2}\left[\begin{array}{cccc} \frac{q^2}{2} - \frac{uc}{\gamma-1} & -u + \frac{c}{\gamma-1} & -v & 1 \\ \hline \frac{q^2}{2} + \frac{uc}{\gamma-1} & -u - \frac{c}{\gamma-1} & -v & 1 \\ \hline -\frac{2c^2}{\gamma-1} & 0 & \frac{2c^2}{(\gamma-1)v} & -1 \\ \hline -q^2 + \frac{2c^2}{\gamma-1} & 2u & 2v & -2 \end{array}\right]$$

and the corresponding eigenvalues are

$$\mathbf{\Lambda} = \mathrm{diag}([\lambda_1, \lambda_2, \lambda_3, \lambda_4]) = \mathrm{diag}([u+c, u-c, u, u]).$$

Note that all variables without L or R subscripts refer to the Roe-averaged state. Using these expressions, the Roe flux in (3.3.2) becomes, after some manipulation,

$$\hat{\mathbf{F}} = \frac{1}{2}(\mathbf{F}_L + \mathbf{F}_R) - \frac{1}{2}\begin{bmatrix} |\lambda|_3 \Delta\rho & +C_1 & \\ |\lambda|_3 \Delta(\rho\vec{v}) & +C_1\vec{v} & +C_2\vec{n} \\ |\lambda|_3 \Delta(\rho E) & +C_1 H & +C_2 u \end{bmatrix}$$

where

$$C_1 = \frac{G_1}{c^2}(s_1 - |\lambda|_3) + \frac{G_2}{c}s_2, \qquad C_2 = \frac{G_1}{c}s_2 + (s_1 - |\lambda|_3)G_2$$
$$G_1 = (\gamma-1)\left(\frac{q^2}{2}\Delta\rho - \vec{v}\cdot\Delta(\rho\vec{v}) + \Delta(\rho E)\right), \qquad G_2 = -u\Delta\rho + \Delta(\rho\vec{v})\cdot\vec{n}$$
$$s_1 = \frac{1}{2}(|\lambda|_1 + |\lambda|_2), \qquad s_2 = \frac{1}{2}(|\lambda|_1 - |\lambda|_2)$$
$$\Delta\mathbf{u} = \mathbf{u}_R - \mathbf{u}_L,$$

and $u = \vec{v} \cdot \vec{n}$. Note, $\vec{v}$ can be defined in any coordinate system; in this general case, $\mathbf{F}_L = \vec{\mathbf{F}}(\mathbf{u}_L) \cdot \vec{n}$ and $\mathbf{F}_R = \vec{\mathbf{F}}(\mathbf{u}_R) \cdot \vec{n}$. The above equations are written in a coordinate-system independent form, which means that in practice, no coordinate system rotation needs to be performed to calculate $\hat{\mathbf{F}}$.

To prevent expansion shocks, an *entropy fix* is needed. One simple choice is to keep all eigenvalues away from zero,

$$\text{if } |\lambda|_i < \epsilon \text{ then } \lambda_i = \frac{\epsilon^2 + \lambda_i^2}{2\epsilon}, \quad \text{for all } i \in [1, 4],$$

where ϵ is a small fraction of the Roe-averaged speed of sound, e.g. $\epsilon = 0.05c$. The maximum eigenvalue can be used as the maximum wave speed for time step calculations.

A more dissipative flux than Roe (but not as dissipative as Rusanov) is the **HLLE** flux,

$$\hat{\mathbf{F}} = \frac{1}{2}(\mathbf{F}_L + \mathbf{F}_R) - \frac{1}{2}\frac{s_{\max} + s_{\min}}{s_{\max} - s_{\min}}(\mathbf{F}_R - \mathbf{F}_L) + \frac{s_{\max} s_{\min}}{s_{\max} - s_{\min}}(\mathbf{u}_R - \mathbf{u}_L),$$

where

$$\begin{aligned} s_{\min} &= \min(s_{L,\min}, s_{R,\min}) \\ s_{\max} &= \max(s_{L,\max}, s_{R,\max}) \\ s_{L,\min} = \min(0, u_L - c_L), &\quad s_{L,\max} = \max(0, u_L + c_L) \\ s_{R,\min} = \min(0, u_R - c_R), &\quad s_{R,\max} = \max(0, u_R + c_R) \end{aligned}$$

where c_L, c_R are the sound speeds. The maximum wave speed for time step calculations can be taken as $\max(|u_L| + c_L, |u_R| + c_R)$. Note that $u_L = \vec{v}_L \cdot \vec{n}$ and $u_R = \vec{v}_R \cdot \vec{n}$. One advantage of this flux is its simplicity. A disadvantage is that is smears out contact discontinuities.

3.3.2 Boundary Conditions

In the finite-volume method, boundary conditions are generally imposed through fluxes on the boundary faces. These fluxes may depend on data specified on the boundaries and on the state inside the computational domain. For the Euler equations, both are often used, with the amount of information required on the boundary dependent on the type of boundary condition and on the flow state.

Figure 3.3.1 shows the key quantities used in setting a boundary condition. These include:

$$\begin{aligned} \mathbf{u}^+ &= \text{state inside the computational domain} \\ \mathbf{u}^b &= \text{state on the boundary, just outside the domain} \\ \vec{n} &= \text{normal vector pointing out of the domain} \\ \hat{\mathbf{F}}^b &= \text{numerical flux in direction of } \vec{n} \text{ on boundary} \end{aligned}$$

We now review some of the commonly used boundary conditions for the Euler equations.

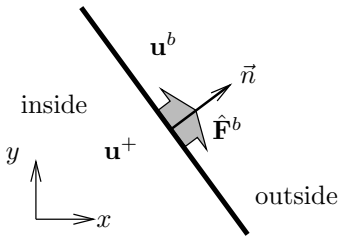


Figure 3.3.1: Interior/boundary state and boundary flux.

Full State

In this boundary condition, the entire boundary state, $\mathbf{u}^b$, is specified by the user. This does not necessarily mean that the entire state is *necessary* for specifying the boundary condition. For example a full state may be specified at a subsonic inflow, though physically there will be one characteristic (acoustic wave) leaving the domain. The full state (e.g. free-stream condition) will therefore generally be an over-specification of the boundary condition. To account for this, a numerical flux function is used to compute the boundary flux,

$$\hat{\mathbf{F}}^b = \hat{\mathbf{F}}(\mathbf{u}^+, \mathbf{u}^b, \vec{n}). \tag{3.3.3}$$

The choice of flux function is not arbitrary: it must be sufficiently accurate to properly upwind the waves of different speeds: e.g. the Roe flux. The full state boundary condition is typically used at farfield boundaries, where the state is approximately uniform but where the distinction between inflow and outflow may be cumbersome to make.

Inviscid Wall

This boundary condition is used to represent a solid object through which the flow cannot pass (no flow through). It is not a no-slip boundary condition, as such a requirement is only suitable for viscous flows. The boundary flux on the wall is given by

$$\hat{\mathbf{F}}^b = [0, p^b n_x, p^b n_y, 0]^T, \qquad p^b = (\gamma - 1)\left[\rho E^+ - \frac{1}{2}\rho^+ |\vec{v}^b|^2\right], \tag{3.3.4}$$

where $\vec{n} = n_x\hat{x} + n_y\hat{y}$, p^b represents the boundary pressure, and $\vec{v}^b$ is the boundary velocity. The boundary velocity is tangential: the interior velocity with the wall-normal component taken out,

$$\vec{v}^b = \vec{v}^+ - (\vec{v}^+ \cdot \vec{n})\vec{n}. \tag{3.3.5}$$

Note p^b is calculated based on the interior density, energy, and boundary (tangential) velocity.

Inflow T_t, p_t, α

This is an inflow boundary condition for which the flow angle α and two total stagnation quantities are specified: total temperature T_t and total pressure p_t. The boundary flux $\hat{\mathbf{F}}^b$ is determined by constructing the boundary state $\mathbf{u}^b$ from the interior state $\mathbf{u}^+$ – in particular the Riemann invariant J^+ – and the specified quantities. J^+ contains all the information from the interior that is used in constructing the exterior state. It is defined as

$$J^+ = u_n^+ + \frac{2c^+}{(\gamma - 1)}.$$

where $u_n^+ = \vec{v}^+ \cdot \vec{n}$ is the wall-normal velocity component from the interior. The inflow Mach number, M^b, is calculated from J^+ and the specified parameters by solving

$$\left(\gamma R T_t d_n^2 - \frac{\gamma - 1}{2}(J^+)^2\right)(M^b)^2 + \left(\frac{4\gamma R T_t d_n}{\gamma - 1}\right) M^b + \frac{4\gamma R T_t}{(\gamma-1)^2} - (J^+)^2 = 0,$$

where $d_n = \vec{n}_{\text{in}} \cdot \vec{n}$ and $\vec{n}_{\text{in}} = [\cos(\alpha), \sin(\alpha)]$ is the specified inflow direction. This quadratic is obtained by expressing the boundary speed of sound $c^b = \sqrt{\gamma R T^b}$ in terms of the boundary Mach number M^b and the given total temperature T_t via an isentropic relation,

$$T_t = \left(1 + 0.5(\gamma - 1)(M^b)^2\right) T^b \quad \Rightarrow \quad (c^b)^2 = \frac{\gamma R T_t}{(1 + 0.5(\gamma - 1)(M^b)^2)}.$$

This expression is then substituted for c^b into the definition of J^+,

$$J^+ = M^b c^b d_n + \frac{2c^b}{\gamma - 1}. \tag{3.3.6}$$

The physically relevant solution ($M^b \geq 0$) to this quadratic is used – if both are greater than zero, then the smaller Mach number is used. Using M and the specified stagnation quantities, the exterior state is calculated as follows:

$$\begin{aligned}
\text{exterior static temperature:} \quad T^b &= T_t/[1 + 0.5(\gamma - 1)(M^b)^2] \\
\text{exterior static pressure:} \quad p^b &= p_t(T^b/T_t)^{\gamma/(\gamma-1)} \\
\text{exterior static density:} \quad \rho^b &= p^b/(RT^b) \\
\text{exterior speed of sound:} \quad c^b &= \sqrt{\gamma p^b/\rho^b} \\
\text{exterior velocity:} \quad \vec{v}^b &= M^b c^b \vec{n}_{\text{in}} \\
\text{exterior total energy:} \quad \rho E^b &= p^b/(\gamma - 1) + \frac{1}{2}\rho^b |\vec{v}^b|^2
\end{aligned}$$

The boundary flux is then calculated according to $\hat{\mathbf{F}}^b = \vec{\mathbf{F}}(\mathbf{u}^b) \cdot \vec{n}$.

Inflow ρ_t, c_t, α

In this boundary condition, the total density (ρ_t) and speed of sound (c_t) are given together with the flow angle α. Since $c_t = \sqrt{\gamma p_t/\rho_t}$, we have $p_t = c_t^2 \rho_t/\gamma$ and $T_t = p_t/(\rho_t R)$, and we can use the same procedure as in the previous inflow boundary condition.

Subsonic Outflow

At a subsonic outflow, the boundary static pressure, p^b, is typically specified. The complete exterior state, $\mathbf{u}^b$, is calculated from the Riemann invariant $J^+(\mathbf{U}^+)$, the interior entropy $S^+ = p^+/(\rho^+)^\gamma$, and the interior tangential velocity v^+. The calculation proceeds as follows. First, the exterior density is

$$\rho^b = \left(\frac{p^b}{S^+}\right)^{1/\gamma}. \tag{3.3.7}$$

The boundary normal velocity, u_n^b, is found using J^+ and $c^b = \sqrt{\gamma p^b/\rho^b}$,

$$u_n^b = J^+ - \frac{2c^b}{\gamma - 1}. \tag{3.3.8}$$

Setting the boundary tangential velocity to the interior tangential velocity then fully defines $\vec{v}^b$, according to

$$\vec{v}^b = \vec{v}^+ - (\vec{v}^+ \cdot \vec{n})\vec{n} + (\vec{v}^b \cdot \vec{n})\vec{n}.$$

$(\rho E)^b$ is calculated as $p^b/(\gamma - 1) + \frac{1}{2}\rho^b|\vec{v}^b|^2$. The boundary flux is calculated according to $\hat{\mathbf{F}}^b = \vec{\mathbf{F}}(\mathbf{u}^b) \cdot \vec{n}$.

Supersonic Outflow

The boundary flux is determined based solely on the interior state, $\hat{\mathbf{F}}^b = \vec{\mathbf{F}}(\mathbf{u}^+) \cdot \vec{n}$. No boundary information is required.

Periodic

This is not a true boundary condition. An edge on a periodic BC should be treated as an interior edge with $\mathbf{u}^b$ equal to the state on the corresponding periodic cell.

3.4 Time Stepping

In a semi-discrete formulation, the last piece is the choice of time integration. For a first-order spatial discretization, the forward Euler method works sufficiently well. Referring to Equation 3.2.2, the complete discrete equation is

$$A_i \frac{u_i^{n+1} - u_i^n}{\Delta t^n} + \underbrace{\sum_{e=1}^{3} \hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e})\, \Delta l_{i,e}}_{R_i} = 0. \tag{3.4.1}$$

Note that Δt gets a superscript n to allow for the possibility of a variable time step. This equation can be solved for u_i^{n+1}, the updated state at time node $n+1$ for cell i,

$$u_i^{n+1} = u_i^n - \frac{\Delta t^n}{A_i} R_i. \tag{3.4.2}$$

Note that the update depends only on the states of cells that are immediately adjacent to cell i.

To set the time step, we use the CFL condition. Recall that in one spatial dimension, the CFL number was given by $CFL = |s|\Delta t/\Delta x$, where s is the wave speed (e.g. a in advection). When the cell size is not constant, which is usually the case for unstructured meshes in two dimensions, the CFL number has to be defined separately for every cell. On cell i, the definition is

$$CFL_i = \frac{\Delta t |\bar{s}|_i}{d_i}, \tag{3.4.3}$$

where d_i is a measure of the cell size, and $|\bar{s}_i|$ is the maximum wave speed. For the cell size, the hydraulic radius is typically used,

$$d_i = \frac{2A_i}{P_i}, \tag{3.4.4}$$

where A_i is the area of cell i and P_i is its perimeter. For $|\bar{s}_i|$, an edge-weighted average of wave speeds is used

$$|\bar{s}|_i = \sum_{e=1}^{3} |s|_{i,e} \Delta l_{i,e}/P_i,$$

where $|s|_{i,e}$ is the wave speed on edge e of cell i. This wave speed can be computed during the flux calculation on each edge.

When solving an unsteady problem time-accurately, every cell takes the same time step, which is set to be the smallest (most restrictive) over all of the cells,

$$\text{global } \Delta t = \min_i \left(\frac{\text{CFL}\, d_i}{|\bar{s}|_i} \right). \tag{3.4.5}$$

Note that the CFL number is a constant over all cells, prescribed by the user. When a mesh has very large and very small cells, the small cells will restrict the time step to small values. This is necessary to keep the explicit time marching stable, but it can lead to a hefty computational burden in terms of a large number of time steps. There are at least two ways to address this problem: local time stepping, and implicit methods.

First, suppose that we are not performing a time-accurate simulation, but are instead using time marching as a way to drive the solution to steady state. In such a case, *how* we get to steady state is not important, and so we dispense with time accuracy. Instead of

marching every cell with the same time step, we let each cell set its own time step, according to its own local CFL condition. That is, the time step on cell i is

$$\text{local } \Delta t_i = \frac{\text{CFL}\, d_i}{|\bar{s}|_i} = \frac{2A_i\, \text{CFL}}{\sum_{e=1}^{3} |s|_{i,e} \Delta l_{i,e}},$$

where the CFL number is set globally by the user. The above equation gives the Δt_i for **local time stepping**. These element-specific time steps are the ones from which we would pick the minimum for global time stepping. So why does using different local time steps work? It seems non-physical ... and it is. However, if all we want from the solution is steady-state, then the path taken by the solution on the way to steady state is not critical. The above choice of time steps lets big cells advect their errors quickly, which speeds up the rate at which error modes leave the domain and hence accelerates convergence.

Second, suppose that we are actually interested in a time-accurate simulation, but that the time step restriction imposed by the smallest cells is too restrictive, much below the Δt we would need for time accuracy. In such a case, the problem is *stiff*, and we could often benefit from an implicit time marching method. These come at a price of solving a system of potentially nonlinear equations at each time step, e.g. via the Newton-Raphson method.

3.5 High-Order Methods and Limiting

Using cell averages as inputs into the numerical fluxes that we have presented means that we are approximating the solution as constant in each cell. This gives a spatially first-order accurate method, since the discretization error is proportional to the cell size (e.g. diameter) to the first power. We can do better by a high-order extension — in fact the industry standard is a second-order finite-volume method.

In a second-order finite-volume method, we need a linear representation of the solution inside each cell. We cannot construct this using only the cell average. Instead we must look to cell averages on neighboring cells for extra information. For triangular cells, the states on the three adjacent neighbors will give us (more than) enough information to construct a gradient of the state. For example, we can compute the average gradient in cell i, $\nabla u\big|_i$, by using integration by parts,

$$\begin{aligned} \int_{A_i} \nabla \mathbf{u}\, dA &= \int_{\partial A_i} \hat{\mathbf{u}} \vec{n}\, dl \\ \nabla \mathbf{u}\big|_i A_i &= \int_{\partial A_i} \hat{\mathbf{u}} \vec{n}\, dl \\ \Rightarrow \qquad \nabla \mathbf{u}\big|_i &= \frac{1}{A_i} \int_{\partial A_i} \hat{\mathbf{u}} \vec{n}\, dl \end{aligned} \tag{3.5.1}$$

where $\hat{\mathbf{u}}$ is some average state on the boundary of cell i. A simple arithmetic average of the left/right face neighbors will often suffice. We can then use the cell gradient to evaluate the states at each of the edge midpoints using

$$\mathbf{u}(\vec{x}) = \mathbf{u}_i + \nabla \mathbf{u}\big|_i \cdot (\vec{x} - \vec{x}_i), \tag{3.5.2}$$

where $\vec{x}_i$ is the cell centroid and $\vec{x}$ is the location of a desired edge midpoint, as defined in Figure 3.5.1. Performing a similar reconstruction on each cell, one can use the reconstructed

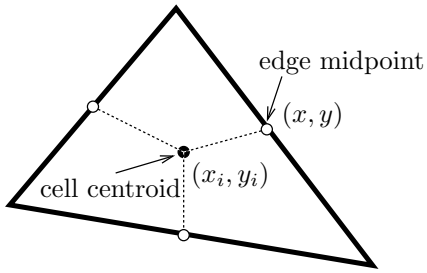


Figure 3.5.1: Centroid and edge midpoints in a triangular mesh cell.

solution values, from the left and right cells of a given edge, to compute the fluxes and increment the residuals. The remainder of the algorithm remains the same as in the first-order case. For completeness, we present the second-order finite-volume method in Algorithm 5.

Algorithm 5 The second-order finite-volume method for a steady-state simulation.

Compute/store edge normals, cell areas, etc.
Initialize the state, $\mathbf{U} = \mathbf{U}^0$, e.g. from a first-order solve.
while $|\mathbf{R}| >$ tolerance **do**
    Initialize the gradient, $\nabla u\big|_i$, to zero on each element
    for $f = 1 : N_{\text{face}}$ **do**
        Let L/R be the elements adjacent to face f
        Let $\vec{n}$ be the normal pointing out of L
        Let Δl be the length of face f
        Set $\hat{\mathbf{u}}$ to the average of the L and R cell averages
        Add $\hat{\mathbf{u}}\vec{n}\Delta l$ to $\nabla u\big|_i$ on L and subtract it from R
    end for
    Divide each $\nabla u\big|_i$ by its element area, A_i
    Initialize residual and wave speed on each cell to zero
    for $f = 1 : N_{\text{face}}$ **do**
        Let L/R be the elements adjacent to face f
        Compute the L/R states on the face midpoint using Equation 3.5.2
        Compute the flux on face f using the *edge* states
        Increment residual on L, decrement it on R
        Add wave speed to tallies on L and R cells
    end for
    Compute the time step on each cell
    Update the state: use RK2 or higher-order method
end while

Note that when converging a second-order solution, we generally start with the converged first-order solution as the initial guess. Also, it is useful to encode the residual calculation in a separate function, since the residual must be evaluated more than once per step when using a multi-stage time-marching scheme.

Even higher-order accuracy can be obtained by reconstructing the state at the interfaces using a wider stencil of neighboring cells. In all cases involving reconstruction, shock capturing techniques are needed to prevent oscillations and nonlinear instabilities.

Chapter 4

The Discontinuous Galerkin Method

4.1 High-Order Methods

4.1.1 Introduction

In many scientific and engineering disciplines a debate continues on the merits and deficiencies of low-order versus high-order algorithms, in particular for Computational Fluid Dynamics (CFD). In the International Workshop on High-Order CFD Methods [153], "high-order" discretizations were compared against their "low-order" counterparts for a suite of over a dozen test cases. The quotation marks indicate that that demarcation of low/high order is also a subject of debate. A typical answer is that the choice of order depends on the problem: smooth flows are efficiently resolved with high order methods, whereas discontinuities and singularities are better addressed by low-order approximations. However, most practical applications of CFD have both smooth and singular features, the locations of which are generally not known a priori, so that neither low nor high-order algorithms alone dominate.

In a solution-adaptive setting, high order can be applied selectively and becomes just another tool for efficient accuracy improvement. Other tools include mesh size and stretching optimization and mesh motion; one can imagine that more adaptive tools translate into a better use of degrees of freedom, and hence more efficient adaptation, as indicated in Figure 4.1.1. In aerospace engineering *hp*-refinement, refinement of both the mesh and approximation order, is important because aerodynamic flows generally exhibit both smooth and discontinuous features. *hp*-refinement is then a big step in maximizing efficiency and robustness of CFD calculations, and the ability to locally enrich the order of the method is hence particularly important.

4.1.2 Background

Although high-order methods may not be categorically better than low-order methods, high order is in many cases the most efficient refinement option for enrichment and optimization of a solution approximation space. To reap these benefits, we require a discretization

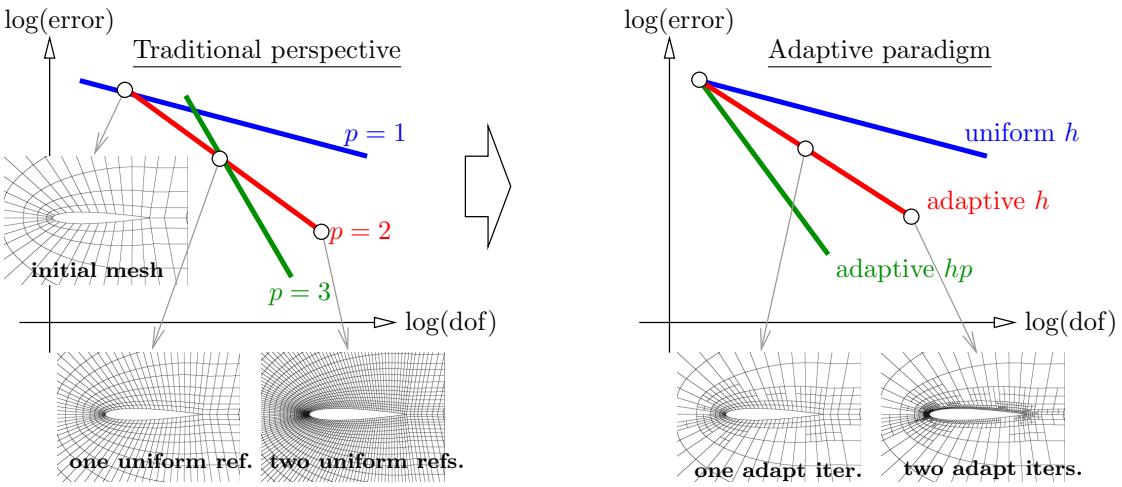


Figure 4.1.1: Schematic comparison of output convergence for uniform mesh refinement and for adaptive refinement. In aerodynamics, *hp*-refinement provides an important capability for addressing errors in flows with both smooth and discontinuous regions, specifically when targeting engineering outputs. Note that in the presence of singular features, the convergence rate of high-order approximations with uniform refinement (left) will generally be no better than that of low-order approximations, whereas this is not the case with *hp*-refinement.

that admits high order, preferably in a locally-adaptive manner. Several such choices exist, yet we turn to finite elements for their built-in variational framework, and to discontinuous approximation spaces for convective stability. Previous works have demonstrated the realizability of high-order accuracy [35, 49], error estimation [20, 73, 42], *hp*-adaptation [77], stable viscous discretization [15, 16, 6, 24, 117], and extension to variational space-time algorithms [9, 90, 144, 145, 84, 13, 97, 129, 48] using **discontinuous Galerkin (DG)** finite element methods. In this chapter we present the DG discretization, starting with a scalar advection problem in one dimension and working up to a system with convective and diffusive terms in multiple dimensiosn.

4.2 One-Dimensional Conservation Laws

We introduce DG in one dimension, starting with scalar conservation laws.

4.2.1 1D Scalar Conservation

The equation governing conservation of a scalar one-dimensional field $u(x)$, with flux $F(u)$, is

$$\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} = 0. \tag{4.2.1}$$

We consider a spatial domain $x \in [0, L]$ and a temporal domain $t \in [0, T]$. For unsteady problems we need an initial condition,

$$u(x, t = 0) = u_0(x) = \text{given initial condition.} \tag{4.2.2}$$

Finally, we must specify boundary conditions, but to simplify the initial presentation, we assume a periodic domain, so that no boundary conditions are required. Instead, the solution will be periodic, $u(x + L, t) = u(x, t)$. Figure 4.2.1 illustrates the behavior of a solution to Equation 4.2.1 for scalar advection, $F = au$, where a is a constant advection speed.

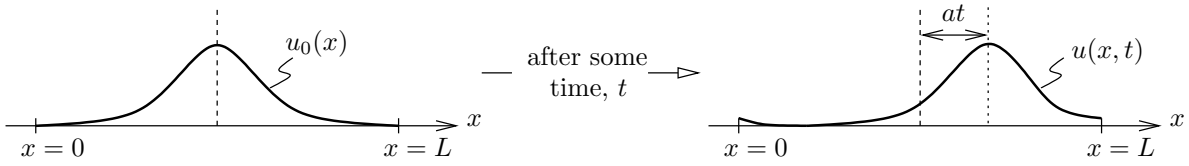


Figure 4.2.1: State evolution for one-dimensional advection.

Spatial Discretization To discretize in space, we use a Galerkin finite-element method with discontinuous basis and test functions. Figure 4.2.2 defines the domain, which we assume consists of elements of the same size, Δx.

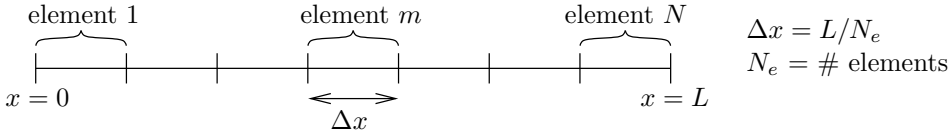


Figure 4.2.2: Uniform space discretization for one-dimensional conservation laws.

We approximate the solution as a weighted sum of basis functions over the entire domain,

$$u(x, t) \approx \sum_{m=1}^{N_e} \sum_{j=1}^{p+1} U_{m,j}(t) \phi_{m,j}(x), \tag{4.2.3}$$

where $\phi_{m,j}(x)$ is the j^{th} polynomial basis function, of order p, on element m, p is the order of spatial approximation on each element, and $U_{m,j}(t)$ are the time-varying expansion coefficients on the corresponding basis functions. Note that the basis functions within one element have local support only on that element: they are zero on the other elements. The expansion coefficients, $U_{m,j}$ are our unknowns, and we have $N_e(p + 1)$ of them.

A popular basis consists of Lagrange interpolating functions on equally-spaced nodes, as shown in Figure 4.2.3. Basis functions are usually expressed on a reference element, which we will choose as $\xi \in [-1, 1]$. Figure 4.2.4 gives formulas for the $p = 1$ and $p = 2$ functions

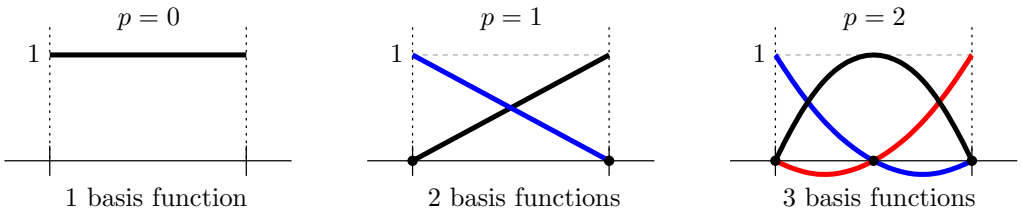


Figure 4.2.3: Lagrange basis functions for DG in 1D.

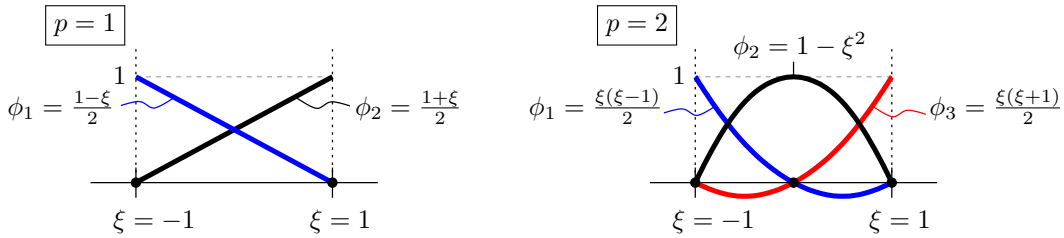


Figure 4.2.4: Reference-space formulas for 1D Lagrange basis functions.

in reference space. For a general $p > 0$, given $p + 1$ nodes ξ_j, the i^{th} 1D Lagrange basis function in reference space is

$$\phi_i(\xi) = \prod_{j=1, j\neq i}^{p+1} \frac{\xi - \xi_j}{\xi_i - \xi_j}. \tag{4.2.4}$$

The mapping between the reference element and a global element k, illustrated in Figure 4.2.5 is

$$x = x_{k-\frac{1}{2}} + \frac{\xi + 1}{2}\Delta x, \qquad \xi = 2\frac{x - x_{k-\frac{1}{2}}}{\Delta x} - 1. \tag{4.2.5}$$

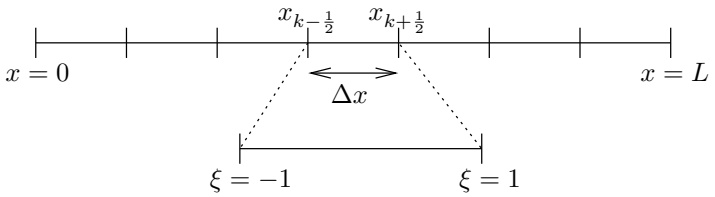


Figure 4.2.5: Mapping between reference and global elements in 1D.

Weak Form To obtain the weak form, we multiply the PDE in Equation 4.2.1 by test functions, which are the basis functions, and integrate by parts over the elements. Consider

test function $\phi_{k,i}$, where k is an element index and i enumerates the basis functions inside the element. We only need to integrate over the support of $\phi_{k,i}$, i.e. element k, since only there is $\phi_{k,i}$ nonzero. Denoting the interior of element k by Ω_k, we have

$$\begin{aligned} \int_{\Omega_k} \phi_{k,i} \left[\frac{\partial u}{\partial t} + \frac{\partial F}{\partial x} \right] dx &= 0 \\ \int_{\Omega_k} \phi_{k,i} \frac{\partial u}{\partial t} \, dx + \int_{\Omega_k} \phi_{k,i} \frac{\partial F}{\partial x} \, dx &= 0 \\ \int_{\Omega_k} \phi_{k,i} \frac{\partial u}{\partial t} \, dx - \int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} F \, dx + \left[\phi_{k,i} \hat{F} \right]_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} &= 0 \end{aligned} \tag{4.2.6}$$

The numerical flux $\hat{F}$ couples unknowns in adjacent elements. It resolves the conflict of a double-valued state at element interfaces. Figure 4.2.6 illustrates this situation on the interface between elements k and $k+1$. Fortunately, we already solved this problem in the

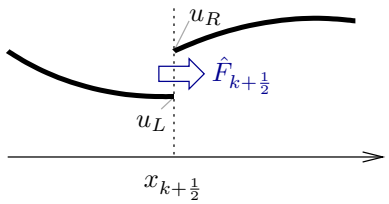


Figure 4.2.6: A flux function yields a unique flux between two different states.

finite volume method. In fact, the numerical fluxes used to solve the Riemann problems in the finite volume method can be used directly in DG.

Note that u_L and u_R in Figure 4.2.6 come from the element polynomials evaluated at the cell interface. Specifically, u_L is the state from the right boundary of the left element, i.e. element k, and u_R is the state from the left boundary of the right element, i.e. element $k+1$. Using the solution approximation formula in Equation 4.2.3, these states are

$$u_L = \sum_{j=1}^{p+1} U_{k,j} \phi_{k,j}(\xi = 1), \tag{4.2.7}$$

$$u_R = \sum_{j=1}^{p+1} U_{k+1,j} \phi_{k+1,j}(\xi = -1). \tag{4.2.8}$$

When using a Lagrange basis, on any element, there is only one basis function that is nonzero at $\xi = 1$, and this is the $(p+1)^{\text{th}}$ function. Similarly, only the first basis function is nonzero at $\xi = -1$. Both are 1 at these respective points, so that the above formulas are

$$u_L = U_{k,p+1}, \quad u_R = U_{k+1,1} \qquad \text{(for a Lagrange basis)}. \tag{4.2.9}$$

Substituting Equation 4.2.3 into Equation 4.2.6 gives a system of equations

$$\mathbf{M}\frac{d\mathbf{U}}{dt} + \mathbf{R}(\mathbf{U}) \;=\; \mathbf{0}, \tag{4.2.10}$$

where $\mathbf{U}$ is the complete "unrolled" state vector of unknowns over all elements,

$$\mathbf{U} = \Big[\underbrace{U_{1,1},\ldots,U_{1,p+1}}_{\text{element 1}},\; \underbrace{U_{2,1},\ldots,U_{2,p+1}}_{\text{element 2}},\; \ldots,\; \underbrace{U_{N_e,1},\ldots,U_{N_e,p+1}}_{\text{element } N_e}\Big]^T. \tag{4.2.11}$$

The first term in Equation 4.2.10 comes from the first term in Equation 4.2.6,

$$\int_{\Omega_k} \phi_{k,i}\frac{\partial u}{\partial t} \;=\; \sum_{j=1}^{p+1} \underbrace{\Big[\int_{\Omega_k} \phi_{k,i}\phi_{k,j}\,dx\Big]}_{[\mathbf{M}_k]_{i,j}} \frac{dU_{k,j}}{dt}, \tag{4.2.12}$$

where we have defined $\mathbf{M}_k$ as the **mass matrix** for element k. In Equation 4.2.10, we use the *global* mass matrix, $\mathbf{M}$, obtained by putting all of the elemental mass matrices together in a block diagonal form,

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_1 & 0 & 0 & \ldots & 0 \\ 0 & \mathbf{M}_2 & 0 & \ldots & 0 \\ 0 & 0 & \mathbf{M}_3 & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & \mathbf{M}_{N_e} \end{bmatrix} \tag{4.2.13}$$

The size of $\mathbf{M}$ is $N_e(p+1)\times N_e(p+1)$. By arranging the elemental mass matrices in this way, in the product $\mathbf{M}\frac{d\mathbf{U}}{dt}$, the k^{th} block of $\mathbf{M}$ acts on the unknowns in element k, as required in Equation 4.2.12.

The second term in Equation 4.2.10, $\mathbf{R}(\mathbf{U})$, is the **residual**, and it comes from the second two terms in the weak form in Equation 4.2.6. Each element has $p+1$ residuals, and the i^{th} residual on element k is given by

$$R_{k,i} = -\int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} F\,dx + \Big[\phi_{k,i}\hat{F}\Big]_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}}. \tag{4.2.14}$$

Rolling these together gives the entire residual vector

$$\mathbf{R} = \Big[\underbrace{R_{1,1},\ldots,R_{1,p+1}}_{\text{element 1}},\; \underbrace{R_{2,1},\ldots,R_{2,p+1}}_{\text{element 2}},\; \ldots,\; \underbrace{R_{N_e,1},\ldots,R_{N_e,p+1}}_{\text{element } N_e}\Big]^T. \tag{4.2.15}$$

This is a function of the state because the fluxes in Equation 4.2.14 depend on the state. Note, however, that to evaluate all the residuals in element k, i.e. the vector $\mathbf{R}_k$, we only need to know the states in element k and its immediate neighbors, $k-1$ and $k+1$ (possibly just one). The dependence on neighbors comes through the flux function $\hat{F}$, which uses states on possibly both sides of an interface.

4.2.2 1D Advection

Let's take a look at a simple scalar equation, linear advection, for which the flux in Equation 4.2.1 is $F = au$. We assume that the advection speed a is constant and positive. For the flux function, $\hat{F}$, we use an upwind flux. Referring to Figure 4.2.6, the upwind flux is defined as

$$\hat{F}_{k+\frac{1}{2}} = \begin{cases} au_L, & a \geq 0 \\ au_R, & a < 0 \end{cases} = \frac{1}{2}\left(F_L + F_R\right) - \frac{1}{2}|a|(u_R - u_L).$$

As discussed in the previous section, u_L and u_R come from the element polynomials evaluated at the cell interface. The corresponding fluxes are $F_L = F(u_L)$ and $F_R = F(u_R)$. Assuming a Lagrange basis and $a > 0$, we have

$$\hat{F}_{k+\frac{1}{2}} = au_L = aU_{k,p+1}. \tag{4.2.16}$$

The expression for the residual in Equation 4.2.14 then becomes

$$\begin{aligned} R_{k,i} &= -\int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} F \, dx + \left[\phi_{k,i}\hat{F}\right]_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} \\ &= -\int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} (au) \, dx + \left[\phi_{k,i}\hat{F}\right]_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} \\ &= -\sum_{j=1}^{p+1} \left[\int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} a\phi_{k,j} \, dx\right] U_{k,j} + \phi_{k,i}\Big|_{\xi=1} aU_{k,p+1} - \phi_{k,i}\Big|_{\xi=-1} aU_{k-1,p+1}. \end{aligned} \tag{4.2.17}$$

Note that $\hat{F}_{k-\frac{1}{2}} = aU_{k-1,p+1}$.

With the residual expression in Equation 4.2.17, we are ready to time-integrate the system of ODEs in Equation 4.2.10. If using an explicit time-marching method, such as RK4, we can re-write Equation 4.2.10 as

$$\frac{d\mathbf{U}}{dt} = -\mathbf{M}^{-1}\mathbf{R}(\mathbf{U}), \tag{4.2.18}$$

and apply the time-marching method directly to this canonical form. If using an implicit method, we need to solve a system of equations, and to start, we typically leave both terms on the left-hand side. Since advection is linear, we can write the residual vector as

$$\mathbf{R} = \mathbf{A}\mathbf{U}, \tag{4.2.19}$$

where the block-sparse **stiffness matrix** $\mathbf{A}$ is, still assuming $a > 0$,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & 0 & 0 & \dots & \mathbf{A}_{1,N_e} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & 0 & \dots & 0 \\ 0 & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \mathbf{A}_{N_e,N_e-1} & \mathbf{A}_{N_e,N_e} \end{bmatrix}. \tag{4.2.20}$$

The k^{th} block row in $\mathbf{A}$ corresponds to the residuals on element k. $\mathbf{A}$ has entries on the block diagonal because the unknowns in element k affect the residuals in element k. $\mathbf{A}$ also has unknowns on the lower block diagonal because the unknowns in element $k-1$ affect the residuals in element k, via the last term in Equation 4.2.17. The periodic domain is responsible for the wrap-around appearance of $\mathbf{A}_{1,N_e}$ on the top right in $\mathbf{A}$. The expressions for the on-diagonal and off-diagonal blocks for element k are, from Equation 4.2.17,

$$\begin{aligned} [\mathbf{A}_{k,k}]_{i,j} &= -a \int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} \phi_{k,j} \, dx + a \delta_{i,p+1} \delta_{j,p+1} \\ [\mathbf{A}_{k-1,k}]_{i,j} &= -a \delta_{i,1} \delta_{j,p+1} \end{aligned}$$

These integrals can generally be evaluated analytically. For example, for the $p = 1$ Lagrange basis,

$$\mathbf{A}_{k,k} = \frac{a}{2} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}, \qquad \mathbf{A}_{k-1,k} = \begin{bmatrix} 0 & -a \\ 0 & 0 \end{bmatrix}.$$

Note, these matrices are independent of element size, Δx! The elemental mass matrix for $p = 1$ is

$$\mathbf{M}_k = \frac{\Delta x}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix},$$

and this does depend on Δx. When evaluating these integrals, the mapping to reference space in Equation 4.2.5 is useful. For example, since $dx = (\Delta x/2) d\xi$, the expression for the (i,j) entry of the elemental mass matrix $\mathbf{M}_k$ in Equation 4.2.12 can be evaluated as,

$$[\mathbf{M}_k]_{i,j} = \int_{\Omega_k} \phi_{k,i} \phi_{k,j} \, dx = \int_{-1}^{1} \phi_{k,i} \phi_{k,j} \frac{\Delta x}{2} d\xi = \frac{\Delta x}{2} \int_{-1}^{1} \phi_{k,i}(\xi) \phi_{k,j}(\xi) \, d\xi. \tag{4.2.21}$$

The last integral can be done purely in reference space, so it is independent of the element size. This means that the mass matrices for all elements are just scaled versions of each other, with $\Delta x/2$ as the scaling factor.

With the matrix representation $\mathbf{R} = \mathbf{A}\mathbf{U}$, we can now implement an implicit time-marching method. For example, a backward Euler time discretization applied to Equation 4.2.10 is

$$\begin{aligned} \mathbf{M} \frac{\mathbf{U}^{n+1} - \mathbf{U}^n}{\Delta t} + \mathbf{A}\mathbf{U}^{n+1} &= \mathbf{0} \\ \left[\frac{\mathbf{M}}{\Delta t} + \mathbf{A} \right] \mathbf{U}^{n+1} &= \frac{\mathbf{M}}{\Delta t} \mathbf{U}^n \quad \leftarrow \quad \text{solve for } \mathbf{U}^{n+1}, \end{aligned}$$

where n is the time node index, and $n = 0$ corresponds to the initial condition.

4.2.3 1D Systems

Consider now a system of s conservation laws in 1D,

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = \mathbf{0}, \quad \mathbf{u}(x, t = 0) = \mathbf{u}_0(x) = \text{initial condition} \tag{4.2.22}$$

Now, both the state vector $\mathbf{u}$ and the flux $\mathbf{F}$ consist of s numbers. The extension of the solution approximation in Equation 4.2.3 is

$$\mathbf{u}(x, t) \approx \sum_{m=1}^{N_e} \sum_{j=1}^{p+1} \mathbf{U}_{m,j}(t) \phi_{m,j}(x), \tag{4.2.23}$$

where now $\mathbf{U}_{m,j}$ contains s expansion coefficients for basis function j on element m. This means that all state components are approximated using the same basis functions. Of course, the approximations can be different, since we have separate expansion coefficients for each component, but because the bases are the same, each component will be an order p polynomial.

We obtain the weak form by multiplying the PDE in Equation 4.2.22 by test functions, which again are the basis functions in our Galerkin method, and integrating by parts over the elements. As in the scalar case, consider test function $\phi_{k,i}$, where k is an element index and i enumerates the basis functions inside the element. We integrate only over element k, since that's where $\phi_{k,i}$ is nonzero,

$$\begin{aligned}
\int_{\Omega_k} \phi_{k,i} \left[\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} \right] dx &= \mathbf{0} \\
\int_{\Omega_k} \phi_{k,i} \frac{\partial \mathbf{u}}{\partial t} \, dx + \int_{\Omega_k} \phi_{k,i} \frac{\partial \mathbf{F}}{\partial x} \, dx &= \mathbf{0} \\
\int_{\Omega_k} \phi_{k,i} \frac{\partial \mathbf{u}}{\partial t} \, dx - \int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} \mathbf{F} \, dx + \left[\phi_{k,i} \hat{\mathbf{F}} \right]_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} &= \mathbf{0}
\end{aligned} \tag{4.2.24}$$

The numerical flux $\hat{\mathbf{F}} \in \mathbb{R}^s$ couples unknowns in adjacent elements, and here again we use the same flux functions as in the finite-volume method. Note that Equation 4.2.24 is a vector equation that actually consists of s equations.

Substituting Equation 4.2.23 into Equation 4.2.24 results in a system of ODEs that looks the same as Equation 4.2.10,

$$\mathbf{M} \frac{d\mathbf{U}}{dt} + \mathbf{R}(\mathbf{U}) = \mathbf{0}. \tag{4.2.25}$$

However, for systems, the vectors and matrices are now bigger. Figure 4.2.7 shows the unrolled storage pattern for the state vector $\mathbf{U}$, which now contains $N_e(p+1)s$ numbers. The residual vector storage is the same, with $N_e(p+1)s$ entries in $\mathbf{R}$. The mass matrix is now formally of size $N_e(p+1)s \times N_e(p+1)s$, but it contains the same numbers as the scalar

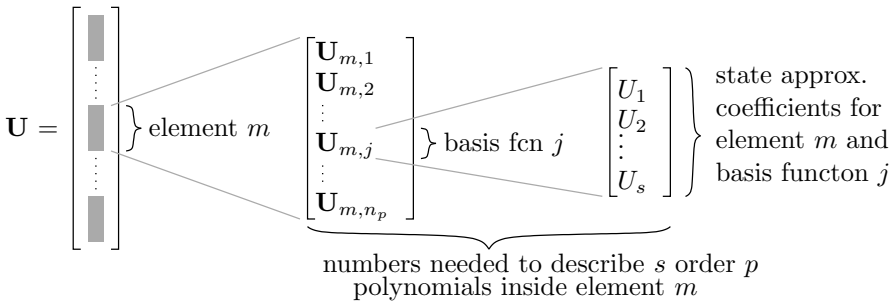


Figure 4.2.7: Unrolled storage of a the state vector for a system of equations in DG. The number of basis functions per element in 1D is $n_p = p + 1$.

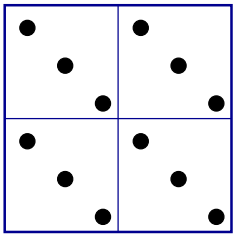


Figure 4.2.8: Sparsity pattern of a 1D elemental mass matrix for $p = 1$ and $s = 3$. The dots indicate nonzero entries, and all diagonal entries within a block are identical.

mass matrix, arranged in diagonal blocks of size $s \times s$. For example, the elemental mass matrix for $p = 1$ and $s = 3$ has the sparsity pattern shown in Figure 4.2.8. Note that the mass matrix does not couple the states in its action. The repeated entries in the mass matrix are generally not stored separately, and instead the action of the mass matrix on a vector is computed with the original scalar mass matrix acting on the state vector components individually. A convenient way to implement this mass matrix product, e.g. **MU**, is to reshape the **U** vector into a size $N_e(p+1) \times s$ matrix, act on it with the scalar mass matrix, and reshape the result back to an unrolled vector.

Similarly to Equation 4.2.14, the residual associated with test function i in element k is

$$\mathbf{R}_{k,i} \quad = \quad -\int_{\Omega_k} \frac{\partial \phi_{k,i}}{\partial x} \mathbf{F} \, dx + \left[\phi_{k,i} \hat{\mathbf{F}} \right]_{x_{k-\frac{1}{2}}}^{x_{k+\frac{1}{2}}} . \tag{4.2.26}$$

This residual now consists of s numbers. Time stepping via an explicit method could now be readily implemented as in the case of advection. If the system is linear, then an implicit method could also be implemented as described above, with **A** now a matrix of size $N_e(p+1)s \times N_e(p+1)s$. If the system is not linear, then the Newton-Raphson method can be used, requiring the linearization of **R** with respect to **U**.

4.3 Two-Dimensional Conservation Laws

4.3.1 2D Scalar Conservation

The equation governing conservation of a scalar two-dimensional field $u(\vec{x})$, with flux $\vec{F}(u)$, is

$$\frac{\partial u}{\partial t} + \nabla \cdot \vec{F} \quad = \quad 0. \tag{4.3.1}$$

For unsteady problems we need an initial condition,

$$u(\vec{x}, t = 0) = u_0(\vec{x}) = \text{given initial condition.} \tag{4.3.2}$$

Appropriate boundary conditions must also be imposed if the domain is not periodic. Figure 4.2.1 illustrates the behavior of a solution to Equation 4.3.1 for scalar advection, $F = \vec{V}u$, where $\vec{V}$ is a constant advection velocity vector.

Spatial Discretization To discretize in space, we use a Galerkin finite-element method with discontinuous basis and test functions. Figure 4.3.2 illustrates a sample unstructured mesh of triangular elements. We approximate the solution as a weighted sum of basis functions over the entire domain,

$$u(\vec{x}, t) \approx \sum_{m=1}^{N_e} \sum_{j=1}^{n_p} U_{m,j}(t) \phi_{m,j}(\vec{x}), \tag{4.3.3}$$

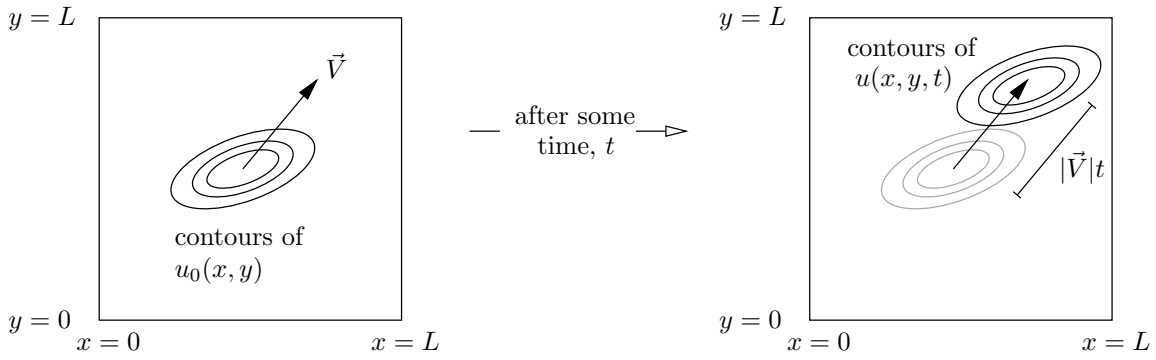


Figure 4.3.1: State evolution for two-dimensional advection.

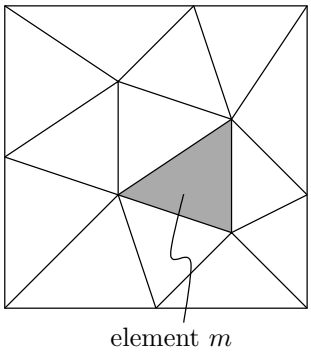


Figure 4.3.2: An unstructured mesh for a two-dimensional DG discretization.

where $\phi_{m,j}(\vec{x})$ is the j^{th} polynomial basis function, of order p, on element m, p is the order of spatial approximation on each element, n_p is the number of basis functions per element, and $U_{m,j}(t)$ are the time-varying expansion coefficients on the corresponding basis functions. The basis functions have local support, each over just one element. The coefficients $U_{m,j}$ are our unknowns, and we have $N_e n_p$ of them.

Approximation Spaces In one dimension we used polynomials of order p in x to approximate the solution. In two dimensions, we have more freedom in choosing the types of polynomials, now in x and y, used for the approximation. We present two spaces: tensor-product and full-order.

A **tensor-product** space of order p, denoted by $\mathbb{Q}^p$, consists of the span of basis functions $x^r y^s$ where $0 \leq r \leq p$ and $0 \leq s \leq p$. For example,

$$\begin{aligned} \mathbb{Q}^1 &= \text{span}\left(1, x, y, xy\right), \\ \mathbb{Q}^2 &= \text{span}\left(1, x, y, xy, x^2, y^2, x^2y, xy^2, x^2y^2\right). \end{aligned}$$

The number of basis functions, i.e. the dimension of the space, $n_p = \dim(\mathbb{Q}^p) = (p+1)^2$. This means that we have $(p+1)^2$ unknowns per element when using $\mathbb{Q}^p$. This space is often used on quadrilateral elements, a remnant of continuous finite element approximations. In DG, however, the tensor-product space could be used on arbitrary elements, since the approximations on each element are independent.

A **full-order** space of order p, denoted by $\mathbb{P}^p$, consists of the span of basis functions $x^r y^s$, where $r \geq 0$, $s \geq 0$, and $r + s \leq p$. For example,

$$\begin{aligned} \mathbb{P}^1 &= \text{span}\left(1, x, y\right), \\ \mathbb{P}^2 &= \text{span}\left(1, x, y, x^2, xy, y^2\right). \end{aligned}$$

The number of basis functions is $n_p = \dim(\mathbb{P}^p) = (p+1)(p+2)/2$ unknowns per element. A full-order space is often used on triangles, again an remnant of continuous finite-element approximation choices.

We now provide expressions for both spaces on reference elements. To define the tensor-product Lagrange basis, we use a unit-square reference element, $(\xi, \eta) \in [0,1]^2$, as illustrated in Figure 4.3.3. The j^{th} tensor-product basis function is

$$\phi_j(\xi, \eta) = \phi^{1d}_{r+1}(\xi)\, \phi^{1d}_{s+1}(\eta) \tag{4.3.4}$$

where $j = (p+1)s + r + 1$. The ranges of the indices are $j \in [1, (p+1)^2]$ and $r, s \in [0, p]$, and $\phi^{1d}_{r+1}(\xi)$ and $\phi^{1d}_{s+1}(\eta)$ are one-dimensional Lagrange basis functions. The nodes in ξ and η need not be equally spaced, and for high orders, Gauss/cosine spacing may improve conditioning.

To define a full-order basis, we use a unit right-triangle element, as shown in Figure 4.3.4, with $\xi \in [0,1]$, and $\eta \in [0,1]$. The j^{th} full-order basis function is

$$\phi_j(\xi, \eta) = \sum_{s=0}^{p} \sum_{r=0}^{p-s} c_{k(r,s),j} \xi^r \eta^s. \tag{4.3.5}$$

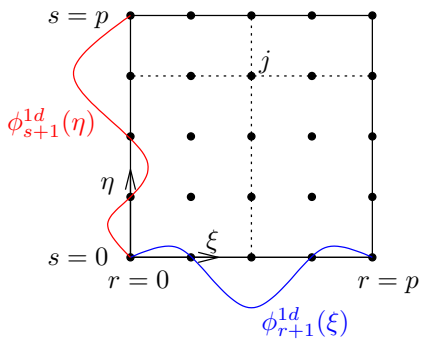


Figure 4.3.3: Tensor-product basis function construction.

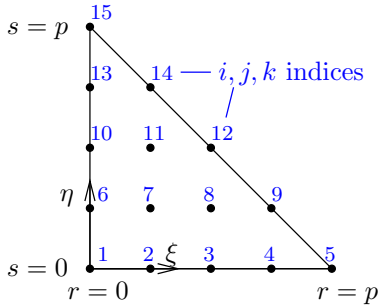


Figure 4.3.4: Full-order basis function construction.

The $c_{k,j}$ are coefficients in a monomial expansion of basis function j. The ranges of the indices are $j \in [1,(p+1)(p+2)/2]$, and $r,s \in [0,p]$, $r+s \le p$. The function $k(r,s)$ returns the full-order index associated with the coordinate pair (r,s). It is given by

$$k(r,s) = \sum_{s'=0}^{s-1}(p+1-s') + r + 1$$

For example, referring to the $p=4$ case shown in Figure 4.3.4,

$$\begin{aligned} r=0,\ s=0 &\ \Rightarrow\ k=1 \\ r=1,\ s=0 &\ \Rightarrow\ k=1 \\ r=0,\ s=1 &\ \Rightarrow\ k=6 \\ r=2,\ s=1 &\ \Rightarrow\ k=8 \end{aligned}$$

As in the tensor-product case, the nodes need not be equally spaced.

To obtain the monomial expansion coefficients $c_{k,j}$ for the j^{th} basis function, we need to solve a matrix system that enforces the Lagrange property: basis function j should be zero at all nodes except node j. The size of the matrix in this system is $n_p \times n_p$, where $n_p = (p+1)(p+2)/2$, and it takes the form

$$\sum_{k=1}^{n_p} A_{i,k}c_{k,j} = \delta_{i,j}, \qquad A_{i,k} = \xi_i^{r(k)}\eta_i^{s(k)},$$

where (ξ_i,η_i) are the reference coordinates of Lagrange node i, $r(k),s(k)$ are the "r,s" coordinates of node k, and $\delta_{i,j}$ is the Kronecker delta function that is 1 if $i=j$ and 0 otherwise. Written out in matrix form, the equations for the coefficients of the j^{th} basis function are

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n_p} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n_p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_p,1} & A_{n_p,2} & \cdots & A_{n_p,n_p} \end{bmatrix} \begin{bmatrix} c_{1,j} \\ c_{2,j} \\ \vdots \\ c_{n_p,j} \end{bmatrix} = \begin{bmatrix} \delta_{1,j} \\ \delta_{2,j} \\ \vdots \\ \delta_{n_p,j} \end{bmatrix}.$$

Row i of this matrix system represents an equation that enforces the value of basis function j at node i: all zeros except 1 when $i=j$.

Listing 4.3.1 provides a Matlab code that builds and solves this system for all basis functions, $1 \le j \le n_p$, and prints out the monomial coefficients.

Listing 4.3.1: Matlab code for calculating the monomial coefficients $c_{k,j}$ for full-order basis functions

```
1  function TriLagrange2D(p);
2  % calculates coeffs for full-order Lagrange basis of order p
3  % reference element is a unit isoceles right triangle
4
5  xi  = linspace(0,1,p+1);  eta = xi;
```

```
N = (p+1)*(p+2)/2; % number of basis functions
A = zeros(N,N); C = A;
i = 1; % build A-matrix
for iy=0:p, for ix=0:p-iy,  % loop over nodes
    k = 1;
    for s=0:p, for r=0:p-s, % loop over monomials
        A(i,k) = xi(ix+1)^r * eta(iy+1)^s;
        k = k+1;
    end, end
    i = i + 1;
end, end;
C = inv(A)

% print out  coefficients
for s=0:p, for r=0:p-s,
    fprintf('%11s', sprintf('x^%d*y^%d', r, s));
end, end
fprintf('\n'); fprintf(strcat(repmat('_%10g', 1,N), '\n'), C);
```

For $p = 1$, the resulting basis functions are

$$\phi_1 = 1 - \xi - \eta, \quad \phi_2 = \xi, \quad \phi_3 = \eta.$$

Figure 4.3.5 illustrates these functions, which are just planes.

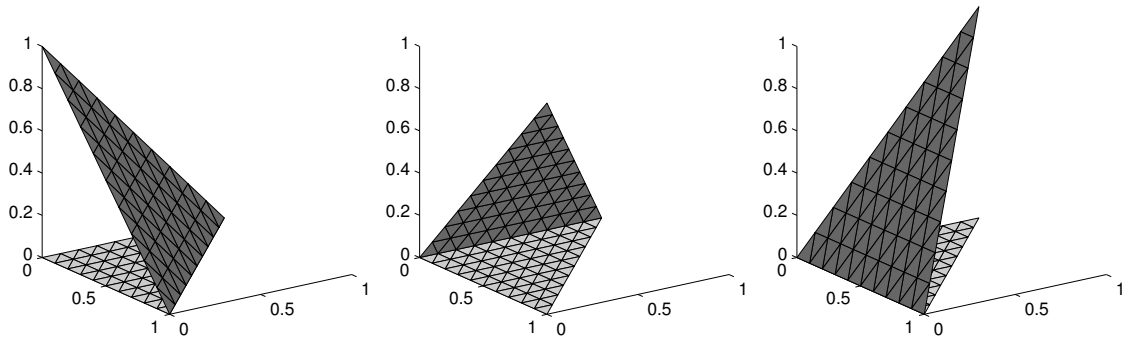


Figure 4.3.5: $p = 1$ full-order Lagrange basis functions.

For $p = 2$, the basis functions are quadratic, and there are six of them:

$$\begin{aligned}
\phi_1 &= 1 - 3\xi + 2\xi^2 - 3\eta + 4\xi\eta + 2\eta^2 \\
\phi_2 &= 4\xi - 4\xi^2 - 4\xi\eta \\
\phi_3 &= -\xi + 2\xi^2 \\
\phi_4 &= 4\eta - 4\xi\eta - 4\eta^2 \\
\phi_5 &= 4\xi\eta \\
\phi_6 &= -\eta + 2\eta^2
\end{aligned}$$

Figure 4.3.6 illustrates these functions.

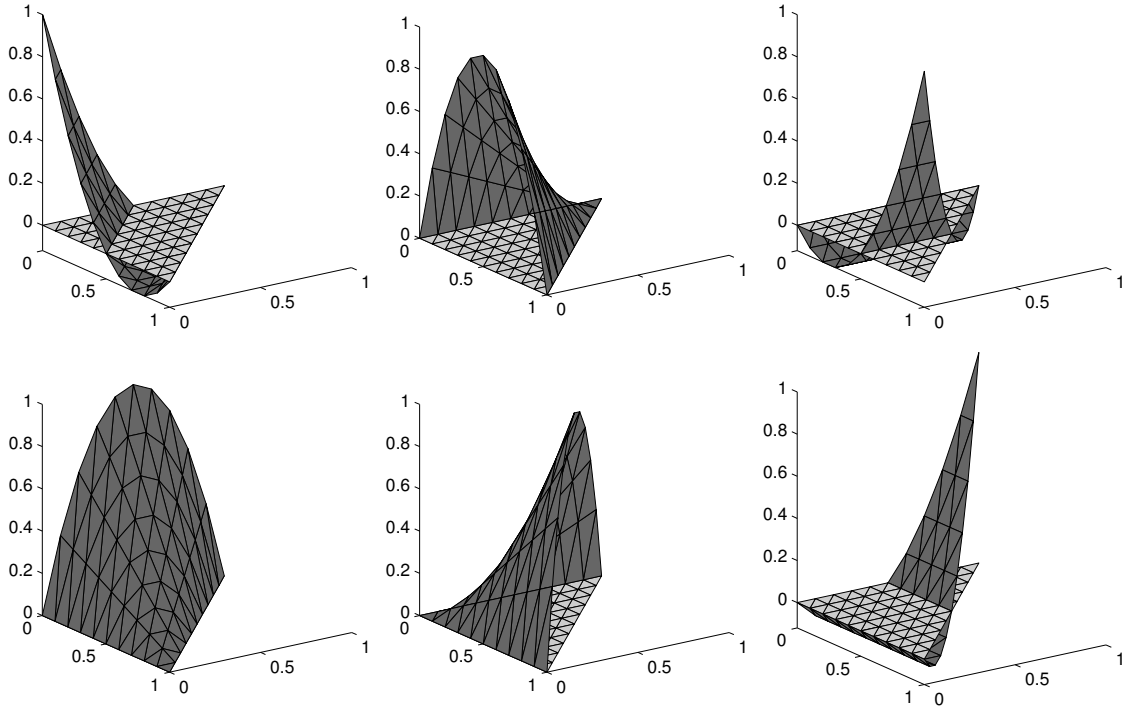


Figure 4.3.6: $p = 2$ full-order Lagrange basis functions.

Reference element calculations A reference element is useful for calculating integrals and computing gradients. For example, the gradients of a basis function are most easily calculated by expressing ϕ_j in terms of reference-space coordinates, differentiating with respect to these, and then mapping into global space.

Figure 4.3.7 illustrates the mapping from a reference element, a unit right triangle, to global space. We can express this map conveniently using the set of linear, order 1, basis functions,

$$\begin{aligned} \vec{x} &= \sum_{i=1}^{3} \vec{x}_i \phi_i(\xi,\eta) \qquad\qquad \phi = \text{order 1 Lagrange basis} \\ &= \vec{x}_1(1-\xi-\eta) + \vec{x}_2\xi + \vec{x}_3\eta \\ &= \vec{x}_1 + (\vec{x}_2 - \vec{x}_1)\xi + (\vec{x}_3 - \vec{x}_1)\eta. \end{aligned} \tag{4.3.6}$$

This equation expresses the desired map because, by the property of Lagrange basis func-

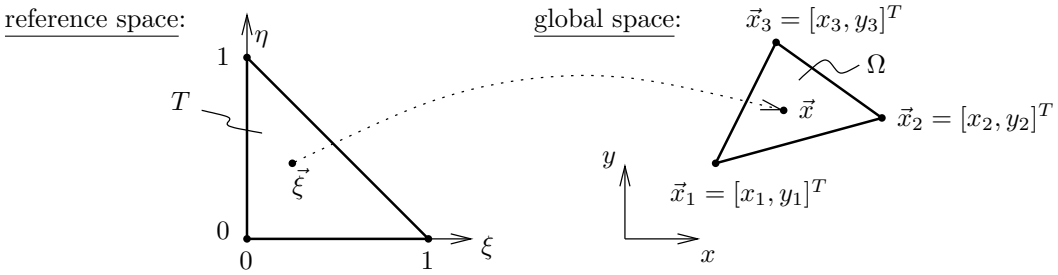


Figure 4.3.7: Mapping from a reference triangle to an arbitrary triangle in global space.

tions, each vertex of the reference triangle maps to one of the global space nodes. Linearity of the basis ensures that the resulting shape is a linear triangle. In matrix form, Equation 4.3.6 is

$$\vec{x} = \vec{x}_1 + \underline{J}\vec{\xi}, \tag{4.3.7}$$

where the **Jacobian matrix** is

$$\underline{J} = \frac{\partial \vec{x}}{\partial \vec{\xi}} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix}. \tag{4.3.8}$$

The inverse mapping is

$$\vec{\xi} = \underline{J}^{-1}(\vec{x} - \vec{x}_1), \tag{4.3.9}$$

where the inverse Jacobian matrix is

$$\underline{J}^{-1} = \frac{\partial \vec{\xi}}{\partial \vec{x}} = \frac{1}{J}\begin{bmatrix} y_3 - y_1 & x_1 - x_3 \\ y_1 - y_2 & x_2 - x_1 \end{bmatrix}, \qquad J \equiv \det(\underline{J})$$

To integrate some function $f(\vec{x})$ over the triangle in global space, Ω, we can use the mapping to express the integral in reference space,

$$\int_\Omega f(\vec{x})d\Omega = \int_T f(\vec{x}(\vec{\xi}))JdT = J\int_T f(\vec{x}(\vec{\xi}))dT. \tag{4.3.10}$$

Note the presence of the Jacobian determinant, J, in the reference-space integrand. J comes out of the integral because it is constant for a linear map from reference to global space.

To differentiate a function with respect to the global coordinates, x and y, we use the chain rule,

$$\left.\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial f}{\partial \eta}\frac{\partial \eta}{\partial x}\\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial f}{\partial \eta}\frac{\partial \eta}{\partial y}\end{aligned}\right\} \quad \frac{\partial f}{\partial \vec{x}} = \frac{\partial f}{\partial \vec{\xi}}\frac{\partial \vec{\xi}}{\partial \vec{x}} = \frac{\partial f}{\partial \vec{\xi}}\underline{J}^{-1} = [\cdot\ \cdot]\begin{bmatrix}\cdot & \cdot\\ \cdot & \cdot\end{bmatrix}. \tag{4.3.11}$$

This equation shows that the gradient with respect to the global coordinates is related to the reference-space gradient through the inverse Jacobian matrix.

Weak form To obtain the weak form, we multiply the PDE in Equation 4.3.1 by test (basis) functions and integrate by parts over each element. Consider test function $\phi_{k,i}$, where k is an element index and i enumerates the basis functions inside the element. We only need to integrate over the support of $\phi_{k,i}$ – i.e. element k, Ω_k.

$$\begin{aligned}\int_{\Omega_k}\phi_{k,i}\left[\frac{\partial u}{\partial t} + \nabla\cdot\vec{F}\right]d\Omega &= 0\\ \int_{\Omega_k}\phi_{k,i}\frac{\partial u}{\partial t}\,d\Omega + \int_{\Omega_k}\phi_{k,i}\nabla\cdot\vec{F}\,d\Omega &= 0\\ \int_{\Omega_k}\phi_{k,i}\frac{\partial u}{\partial t}\,d\Omega - \int_{\Omega_k}\nabla\phi_{k,i}\cdot\vec{F}\,d\Omega + \int_{\partial\Omega_k}\phi^+_{k,i}\hat{F}(u^+,u^-,\vec{n})\,dl &= 0\end{aligned} \tag{4.3.12}$$

In these equations, $\vec{n}$ is a normal vector that points out of Ω_k, $\hat{F}$ is the numerical flux function, and the superscript $^{+/-}$ denotes a quantity taken from the interior/exterior of element k. We assume that adjacent to each edge of the element is another element, and u^- refers to the state from the neighbor, evaluated on the interface.

Substituting Equation 4.3.3 into Equation 4.3.12 gives a system,

$$\mathbf{M}\frac{d\mathbf{U}}{dt} + \mathbf{R}(\mathbf{U}) = \mathbf{0}, \tag{4.3.13}$$

where $\mathbf{U}$ is again the unrolled state vector containing the states from all of the elements,

$$\mathbf{U} = \Big[\underbrace{U_{1,1},\ldots,U_{1,n_p}}_{\text{element 1}},\ \underbrace{U_{2,1},\ldots,U_{2,n_p}}_{\text{element 2}},\ \ldots,\ \underbrace{U_{N_e,1},\ldots,U_{N_e,n_p}}_{\text{element } N_e}\Big]^T. \tag{4.3.14}$$

$\mathbf{M}$ is the block-diagonal global mass matrix that arises from the first term in Equation 4.3.12. The block associated with element k has components

$$[\mathbf{M}_k]_{i,j} = \int_{\Omega_k} \phi_{k,i}\phi_{k,j}\, d\Omega. \tag{4.3.15}$$

Finally, $\mathbf{R}(\mathbf{U})$ is the residual vector that arises from the second two terms in Equation 4.3.12. It is also stored in unrolled form,

$$\mathbf{R} = \left[\underbrace{R_{1,1},\ldots,R_{1,n_p}}_{\text{element 1}},\ \underbrace{R_{2,1},\ldots,R_{2,n_p}}_{\text{element 2}},\ \ldots,\ \underbrace{R_{N_e,1},\ldots,R_{N_e,n_p}}_{\text{element } N_e}\right]^T. \tag{4.3.16}$$

Each element has n_p residuals, and the i^{th} residual on element k is given by

$$R_{k,i} = -\int_{\Omega_k} \nabla\phi_{k,i}\cdot\vec{F}\, d\Omega + \int_{\partial\Omega_k} \phi^+_{k,i}\hat{F}(u^+,u^-,\vec{n})\, dl. \tag{4.3.17}$$

These n_p residuals for element k are functions of the state inside element k and the states on the immediate neighbors of element k. The dependence on the neighbors comes from the boundary integral term involving the numerical flux function, $\hat{F}$.

Implementation An implementation of an explicit time-marching method for a two-dimensional discontinuous Galerkin method consists of the following steps.

1. Pre-compute $\mathbf{M}$ in reference space and, whenever needed, multiply by J to convert into physical space for a given element.
2. Initialize the state at time $t = 0$ through an interpolation or projection.
3. Begin the time marching loop.
4. At each time step, compute the residual $\mathbf{R}(\mathbf{U})$ via two loops: one loop over element interiors in order to calculate the contributions from the second term in Equation 4.3.12, and one loop over interfaces to capture the contribution of the third term in Equation 4.3.12. Note that each interface integral affects the residuals associated with elements on both sides of the interface. This latter calculation is usually performed separately for boundary faces, since these require special treatment for enforcing the boundary contributions.
5. Advance in time using the chosen time-marching scheme, applying the mass matrix inverse as needed.

When the complexity of flux $\vec{F}$ increases, the integrals in Equation 4.3.12 are typically evaluated using numerical integration, e.g. quadrature.

An example of a scalar conservation law is linear advection, where $\vec{F} = \vec{V}u$. In this case, $\hat{F}$ can be chosen as the pure upwind flux, based on the normal velocity $\vec{V}\cdot\vec{n}$. In addition, the residual is now a linear function of the state, so we can write $\mathbf{R} = \mathbf{A}\mathbf{U} + \mathbf{R}_0$, where $\mathbf{A}$ is the stiffness matrix and $\mathbf{R}_0$ contains the contribution to the residual of boundary conditions or source terms.

4.3.2 2D Systems

A system of conservation laws in two dimensions takes the form

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}} = \mathbf{0}, \tag{4.3.18}$$

where $\mathbf{u} \in \mathbb{R}^s$ and s is the state rank. The state approximation formula for an unsteady simulation is

$$\mathbf{u}(\vec{x}, t) \approx \sum_{m=1}^{N} \sum_{j=1}^{n_p} \mathbf{U}_{m,j}(t) \phi_{m,j}(\vec{x}), \tag{4.3.19}$$

where most terms are the same as in the scalar case, with the exception of the coefficients $\mathbf{U}_{m,j}(t)$, which are now vectors of length s. Each component of the state is represented by an independent order p polynomial inside each element.

To obtain the weak form, we multiply the PDE by test functions $\phi_{k,i}$, where k is an element index, and i enumerates the basis functions inside the element, and we integrate over the support of $\phi_{k,i}$ – i.e. the interior of element k, Ω_k,

$$\begin{aligned}
\int_{\Omega_k} \phi_{k,i} \left[\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}} \right] d\Omega &= \mathbf{0} \\
\int_{\Omega_k} \phi_{k,i} \frac{\partial \mathbf{u}}{\partial t} \, d\Omega + \int_{\Omega_k} \phi_{k,i} \nabla \cdot \vec{\mathbf{F}} \, d\Omega &= \mathbf{0} \\
\int_{\Omega_k} \phi_{k,i} \frac{\partial \mathbf{u}}{\partial t} \, d\Omega - \int_{\Omega_k} \nabla \phi_{k,i} \cdot \vec{\mathbf{F}} \, d\Omega + \int_{\partial \Omega_k} \phi^{+}_{k,i} \hat{\mathbf{F}}(\mathbf{u}^{+}, \mathbf{u}^{-}, \vec{n}) \, dl &= \mathbf{0}
\end{aligned} \tag{4.3.20}$$

In this equation, $\hat{\mathbf{F}}$ is a numerical flux on interior edges. On boundary edges it is a flux determined from boundary conditions. Fluxes should only be computed once by looping over edges in the mesh. Area and edge integrals are typically performed via quadrature, of accuracy order $2p + 1$ or more. This requires interpolation of the state to the integration points. Finally, wave-speed estimates for time-step restrictions should be computed/checked at all integration points.

Substituting Equation 4.3.19 into Equation 4.3.20 gives a non-linear system,

$$\mathbf{M} \frac{d\mathbf{U}}{dt} + \mathbf{R}(\mathbf{U}) = \mathbf{0}, \tag{4.3.21}$$

where $\mathbf{M}$ is the mass matrix and $\mathbf{R}(\mathbf{U})$ is a residual vector that depends on the state $\mathbf{U}$. The mass matrix is of the same form as presented in the 1D case, Section 4.2.3. The state and residual vectors are stored unrolled, and the size of these vectors is $N_e n_p s$. Each entry in $\mathbf{R}$ corresponds to one combination of: element, basis function, state component. As the fluxes only depend on the states adjacent to the edge, the residual calculation is local, involving only a small subset of $\mathbf{U}$ for each entry.

For nonlinear fluxes, the system in Equation 4.3.21 can be solved explicitly using an ODE time-marching scheme that exhibits adequate stability on the imaginary axis, or implicitly via Newton-Raphson iterations, requiring linearization of the residual.

4.3.3 Source Terms

When a source term $\mathbf{s}(\mathbf{u})$ is present in the PDE,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}} + \mathbf{s}(\mathbf{u}) = \mathbf{0}, \tag{4.3.22}$$

the residual from the weak-form of Equation 4.3.20 is

$$\mathbf{R}_{ki} = -\int_{\Omega_k} \nabla \phi_{k,i} \cdot \vec{\mathbf{F}} \, d\Omega + \int_{\partial\Omega_k} \phi^+_{k,i} \hat{\mathbf{F}}(\mathbf{u}^+, \mathbf{u}^-, \vec{n}) \, dl + \underbrace{\int_{\Omega_k} \phi_{k,i} \mathbf{s}(\mathbf{u}) \, d\Omega}_{\text{new term}}. \tag{4.3.23}$$

That is, the source term only adds one integral to the residual expression, with the integrand being the product of the test function and $\mathbf{s}(\mathbf{u})$. For general source terms, this integral is performed via quadrature.

4.4 Curved Elements

In finite-element methods, including DG, the geometry of an object is represented by the shape of adjacent elements. When elements are *linear*, i.e. have straight lines for their edges, the geometry representation consists of "panels". However, if the actual geometry is not piecewise linear, but instead curved, then the panel representation of the geometry may not be adequate. This occurs for high-order methods, including even $p = 1$ DG, and in such cases, the elements near the geometry need to be curved, as shown in Figure 4.4.1.

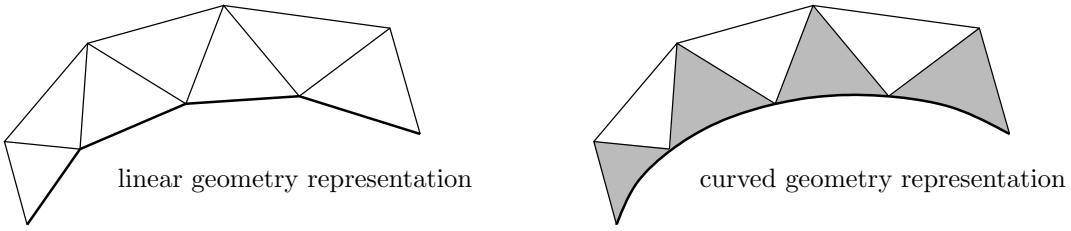
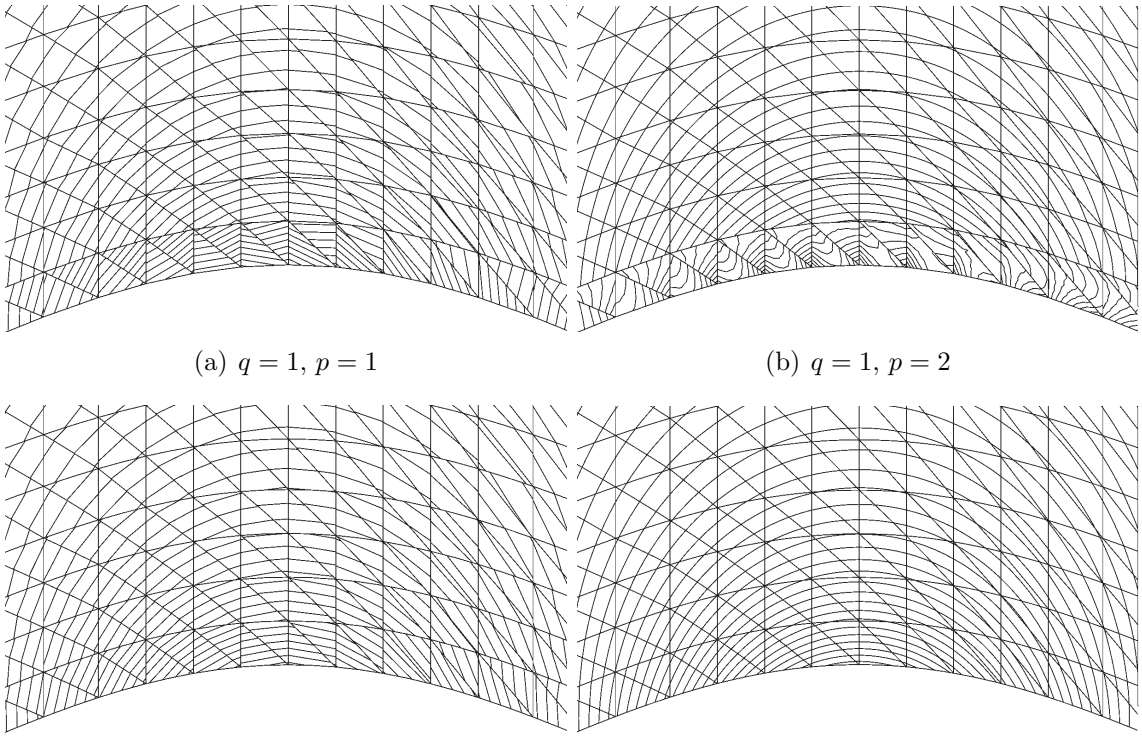


Figure 4.4.1: Linear versus curved geometry representation

The underlying problem with a linear geometry in a high-order solver is that the geometric fidelity is inconsistent with the fidelity of the solution approximation. Instead of solving for flow over a smooth surface, we instead solve for flow over a paneled geometry with sharp corners. This not only lowers the accuracy of the solution, negating the benefits of high order solution approximation, but can also cause iterative convergence problems. For example, when solving the Euler equations, sharp corners cause pressure spikes, which could lead to near-vacuum conditions and non-physical states. Figure 4.4.2 shows the effect of geometry representation on pressure contours in the solution of flow governed by the Euler equations. Note the oscillations of pressure near the boundary when the geometry order is $q = 1$, especially for $p = 2$ solution approximation.



(a) $q = 1$, $p = 1$ (b) $q = 1$, $p = 2$

(c) $q = 2$, $p = 1$ (d) $q = 2$, $p = 2$

Figure 4.4.2: Pressure contours for inviscid compressible flow in a channel with a bump on the bottom wall, using various geometry (q) and solution (p) approximation orders.

Geometry mapping One way to generate curved elements is to use a nonlinear map from the reference triangle. We specifically use a high-order polynomial to express the shape of the element in physical space. Let q be the order of this polynomial. Then, choosing a Lagrange basis to express the map, the task reduces to specifying the global-space coordinates of the high-order geometry nodes. Figure 4.4.3 illustrates this mapping.

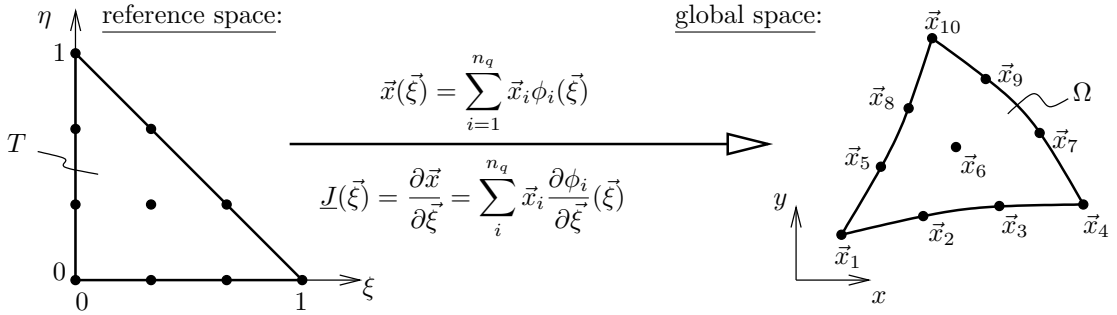


Figure 4.4.3: An order q geometry mapping for producing a curved element in global space.

The Lagrange basis provides a convenient way to represent the geometry because the Lagrange representation interpolates the geometry between specified points. In reference space the n_q nodes are equally spaced. For triangles, we have $n_q = (q+1)(q+2)/2$ points, while for quadrilaterals we have $n_q = (q+1)^2$ points. The Lagrange basis functions in Figure 4.4.3 are denoted by $\phi_i(\vec{\xi})$.

Note that $\vec{\xi} = (\xi, \eta)$ is the reference space coordinate, and $\vec{x} = (x, y)$ is the global coordinate. The mapping Jacobian, $\underline{J}$, is no longer constant for $q > 1$ triangles, and hence the determinant J must be left inside the integral for element-interior integrations. Note that $\underline{J}$ is not even constant inside a general $q = 1$ quadrilateral.

Normal calculation For linear ($q = 1$) elements, the calculation of a normal vector on an edge is easy, and the normal remains constant along an edge. However, this is no longer the case for a curved element where the direction may change along a curved edge.

We present two methods for computing the normal vector, $\vec{n}$, along a curved edge. First, let q be the geometry order of an element adjacent to the edge. If q differs between the two adjacent elements, we choose the element with the lower q (the normals must be the same for a watertight mesh, but the calculation is cheaper for lower q). As a one-dimensional structure, the edge is defined by the $q+1$ nodes along the edge, and the **edge interpolant** is

$$\vec{x}_{\text{edge}}(\sigma) = \sum_{j=1}^{q+1} \vec{x}_{\text{edge},j} \phi_j^{1d}(\sigma), \tag{4.4.1}$$

where

s = arc-length position along the edge
σ = reference position along the edge, $\sigma \in (0,1)$
$\vec{x}_{\text{edge},j}$ = $q+1$ global nodes on edge, $1 \le j \le (q+1)$
$\phi_j^{1d}(\sigma)$ = 1D Lagrange basis on $q+1$ equally-spaced nodes in $[0,1]$

The *tangent* vector along the edge is given by

$$\vec{t}\frac{ds}{d\sigma} = \frac{d\vec{x}_{\text{edge}}}{d\sigma} = \sum_{j=1}^{q+1} \vec{x}_{\text{edge},j}\frac{d\phi_j^{1d}}{d\sigma}(\sigma), \tag{4.4.2}$$

and the associated normal is

$$\vec{n} = \vec{t} \times \hat{k} = [t_y, -t_x] \tag{4.4.3}$$

Note, that

$$\frac{ds}{d\sigma} = \left|\frac{d\vec{x}_{\text{edge}}}{d\sigma}\right|. \tag{4.4.4}$$

so to get $\vec{t}$, we can normalize the direction vector $\frac{d\vec{x}_{\text{edge}}}{d\sigma}$.

Another method for computing $\frac{d\vec{x}_{\text{edge}}}{d\sigma}$, and hence the normal, that does not rely on identifying nodes on the edge uses the chain rule and a transformation to the reference element,

$$\frac{d\vec{x}_{\text{edge}}}{d\sigma} = \frac{d\vec{x}}{d\xi}\frac{d\xi}{d\sigma} + \frac{d\vec{x}}{d\eta}\frac{d\eta}{d\sigma}, \tag{4.4.5}$$

where again (ξ, η) are the element reference coordinates. Note that $\frac{d\vec{x}}{d\xi}$ and $\frac{d\vec{x}}{d\eta}$ are entries of the element mapping Jacobian and $\frac{d\xi}{d\sigma}$ and $\frac{d\eta}{d\sigma}$ depend on the local edge under consideration. For example, on the first local edge of a triangle (the hypotenuse), $\frac{d\xi}{d\sigma} = -1$, $\frac{d\eta}{d\sigma} = 1$.

Integration on a curved edge When integrating along a curved edge, we must account for the non-constant normal and arc length. The integral of a scalar function transforms as

$$\text{scalar:} \quad \int_{\text{edge}} f(\vec{x})ds = \int_0^1 f(\vec{x}(\sigma))\frac{ds}{d\sigma}d\sigma \tag{4.4.6}$$

$ds/d\sigma$ in the integrand provides information about the length of the global-space edge and can be evaluated using Equation 4.4.4. To integrate a vector dotted with a normal, we have

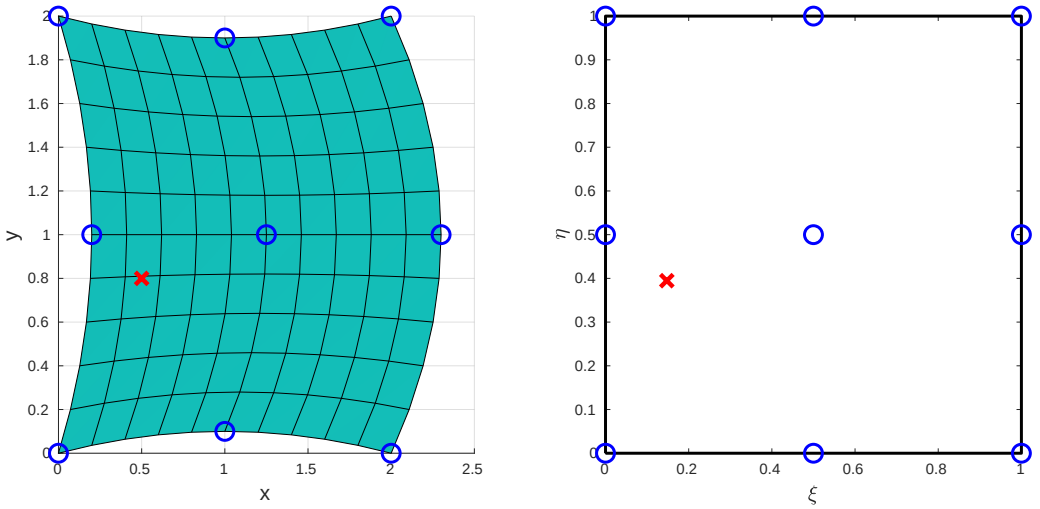
$$\int_{\text{edge}} \vec{f} \cdot \vec{n}\, ds = \int_0^1 \vec{f} \cdot \vec{n}\, \frac{ds}{d\sigma}d\sigma$$

Note, typically we compute the product $\vec{n}(\sigma_q)\frac{ds}{d\sigma}(\sigma_q)$ directly as $\frac{d\vec{x}_{\text{edge}}}{d\sigma} \times \hat{k}$.

Properties and usage Some general properties of and recommendations for using curved elements are as follows.

- We need superparametric ($q > p$) elements for $p = 1$; above $p = 1$ generally safe with isoparametric ($q = p$) or even subparametric ($q < p$) elements. Recall that p is the solution approximation order.
- The global approximation space is not a complete span of order p polynomials when using $q > 1$ curved elements.
- Curved mesh generation is not easy. One option is to obtain high-order node positions from uniformly-refined meshes.
- Need curved elements primarily on boundaries, but sometimes we need to curve several layers of elements for highly-anisotropic boundary-layer meshes.

Curved quadrilateral mapping example The mapping from reference to global space illustrated in Figure 4.4.3 is straightforward to implement, as the node coordinates $\vec{x}_i$ are specified, and the basis functions $\phi_i(\vec{\xi})$ can be evaluated for any reference-space coordinate $\vec{\xi}$. Going the other way, from global space to reference space, is harder but sometimes necessary, e.g. when interrogating a solution at an arbitrary global point. In this example, we consider a curved $q = 2$, quadrilateral, defined by 9 nodes, as shown in Figure 4.4(a). The lines in the figure are contours of constant reference-space coordinate (ξ, η), and they help to visualize the curved element.



(a) Global element (b) Reference element

Figure 4.4.4: Mapping of a point in global space to one in reference space, for a curved quadrilateral.

Given a global-space coordinate $\vec{x}_0$, we would like to obtain the corresponding reference-space coordinate $\vec{\xi}_0$. Since the reference-to-global mapping is nonlinear, we use the Newton-Raphson method to obtain the required inverse mapping. For a given guessed reference-space coordinate $\vec{\xi}$, we define the two-component residual vector as

$$\vec{R} \equiv \vec{x}(\vec{\xi}) - \vec{x}_0 = \sum_{i=1}^{n_q} \vec{x}_i \phi(\vec{\xi}) - \vec{x}_0. \tag{4.4.7}$$

To apply the Newton-Raphson method, we also need the residual Jacobian matrix, which is just the mapping Jacobian defined in Figure 4.4.3,

$$\frac{\partial \vec{R}}{\partial \vec{\xi}} = \sum_{i=1}^{n_q} \vec{x}_i \frac{\partial \phi(\vec{\xi})}{\partial \vec{\xi}} = \underline{J}(\vec{\xi}) \tag{4.4.8}$$

Listing 4.4.1 presents a snippet of code that implements the inverse mapping. The functions `basis` and `gbasis` compute $\phi\vec{\xi}$ and $\frac{\partial \phi(\vec{\xi})}{\partial \vec{\xi}}$, respectively.

Listing 4.4.1: Matlab code snippet for performing an inverse coordinate mapping: global-to-reference space, for a curved element

```
xref  = [0.5, 0.5];  % initialize guess to centroid
tol   = 1e-10;       % tolerance on Newton-Raphson
dmax  = 1.0;         % maximum update for xi and eta

iNewton = 0;
converged = 0;
while (~converged),

  % map xref to global coords -> x
  phi = basis(xn, xref);
  x = phi*xnode; % (x,y) actually

  % calculate residual: R = x(xref) - xglob
  R = x - xglob;

  % check tolerance on residual
  fprintf('iNewton = %d, xref = %.4E %.4E, norm(R) = %.10E\n', ...
          iNewton, xref(1), xref(2), norm(R));
  if (norm(R) < tol), converged = 1; break; end;

  % calculate residual linearization = mapping Jacobian
  [phi_xi, phi_eta] = gbasis(xn, xref);
  J = xnode' * [phi_xi', phi_eta'];

  % determine state update: dxref = -inv(J)*R;
  dxref = -J\(R');

  % limit update using dmax
  d = max(abs(dxref));
  if (d > dmax), dxref = dxref*dmax/d; end;

  % apply state update
  xref = xref + dxref';

  % increment Newton iteration counter
  iNewton = iNewton + 1;
  if (iNewton > 100), error 'Too many iterations: Newton is probably stuck'; end;
end
```

Listing 4.4.2 shows the code printout: the Netwon-Raphson method converges in just three iterations when starting with the reference-element centroid as a guess. Quadratic convergence of Newton is also evident. Figure 4.4(b) shows the resulting point in reference space.

Listing 4.4.2: Inverse coordinate mapping calculation output.

```
1 iNewton = 0, xref = 5.0000E−01 5.0000E−01, norm(R) = 7.7620873481E−01
2 iNewton = 1, xref = 1.4286E−01 3.8889E−01, norm(R) = 1.5508907429E−02
3 iNewton = 2, xref = 1.4738E−01 3.9470E−01, norm(R) = 3.1477657870E−05
4 iNewton = 3, xref = 1.4739E−01 3.9471E−01, norm(R) = 8.6936236138E−11
```

4.5 Diffusion

4.5.1 Formulation

So far we have shown how to discretize first-order conservation laws using DG. "First order" refers to the fact that these conservation laws involved fluxes that modeled convective physics, and hence did not include the state gradient. On the other hand, *second-order* conservation laws involve **diffusion**, which can be written as flux that depends (often linearly) on the gradient of the state.

The model problem for studying diffusion is the **Poisson equation**, which governs the diffusion of a scalar field $u(\vec{x})$,

$$-\nabla^2 u = f \qquad \forall \vec{x} \in \Omega, \qquad u = u_D \text{ on } \partial\Omega \tag{4.5.1}$$

For simplicity we assume Dirichlet boundary conditions, but Neumann or Robin boundary conditions are also possible.

The continuous Galerkin approach Before considering DG, we review the standard continuous Galerkin approach for discretizing Equation 4.5.1. Multiplying by test functions v and integrating by parts over the entire domain gives the weak form

$$a(u,v) \;=\; l(v), \tag{4.5.2}$$

where

$$a(u,v) \;=\; \int_\Omega \nabla u \cdot \nabla v d\Omega, \tag{4.5.3}$$

$$l(v) \;=\; \int_\Omega f v d\Omega. \tag{4.5.4}$$

The trial and test functions u, v lie in $\mathcal{W} = H^1(\Omega)$, which is the space of functions that have square integrable values and first derivatives. Because of the Dirichlet boundary conditions, the test functions v vanish on the domain boundary, and hence this term is not present from integration by parts. In addition, integration by parts can be applied over the entire domain

because the test functions are continuous, in contrast to DG, where the discontinuous nature of the functions restricts the integration by parts to be only over elements.

The discrete problem is obtained by defining a finite-dimensional space,

$$\mathcal{W}_h^p = \left\{ v \in H^1(\Omega) : v|_\kappa \in P^p(\kappa) \ \forall \kappa \in T_h \right\}, \tag{4.5.5}$$

where T_h is a triangulation of the domain into elements κ, and $P^p(\kappa)$ denotes the space of polynomials of order p on the elements κ. We now look for $u_h \in \mathcal{W}_h^p$ such that

$$a(u_h, v_h) = l(v_h) \quad \forall v_h \in \mathcal{W}_h^p. \tag{4.5.6}$$

Note that the bilinear form $a(\cdot, \cdot)$ is symmetric and positive-definite.

An initial attempt with DG Suppose now that we have an approximation space that admits discontinuities at element interfaces. Integration of Equation 4.5.1 by parts over one element yields

$$\begin{aligned} -\int_\kappa v \nabla^2 u d\kappa &= \int_\kappa v f d\kappa \\ -\int_{\partial\kappa} v \widehat{\nabla u} \cdot \vec{n} ds + \int_\kappa \nabla v \cdot \nabla u d\kappa &= \int_\kappa v f d\kappa. \end{aligned} \tag{4.5.7}$$

We put a hat on ∇u at the element boundary since ∇u is discontinuous there and we need to somehow combine the gradients from either side to couple elements together. Since diffusion has no preferred direction, a natural choice is

$$\widehat{\nabla u} = \{\nabla u\} \equiv \frac{1}{2}\left(\nabla u^+ + \nabla u^-\right), \tag{4.5.8}$$

where we have defined $\{\cdot\}$ as the average of quantities taken from the two sides of an interface. Summing together over all of the elements, we obtain the bilinear form,

$$\begin{aligned} B(u, v) &= \sum_{\text{elements}} \left[-\int_{\partial\kappa} v \widehat{\nabla u} \cdot \vec{n} ds + \int_\kappa \nabla v \cdot \nabla u d\kappa \right] \\ &= -\sum_{\text{faces}} \int_{\text{face}} [\![v\vec{n}]\!] \cdot \widehat{\nabla u} ds + \sum_{\text{elements}} \int_\kappa \nabla u \cdot \nabla v d\kappa \end{aligned} \tag{4.5.9}$$

As shown in Figure 4.5.1, $[\![\cdot]\!]$ is the "jump" operator when normals are involved, since $[\![v\vec{n}]\!] = v^+\vec{n}^+ + v^-\vec{n}^- = (v^+ - v^-)\vec{n}^+$.

The bilinear form in Equation 4.5.9 is *consistent* in that $B(u - u_h, v_h) = 0$ for u_h, v_h in a finite-dimensional space. This can be shown by reversing every step in the derivation of $B(\cdot, \cdot)$ for the exact solution, u. Specifically, the averaging operation can be reversed because u lies in $H^2(\Omega)$, so that ∇u is continuous across element interfaces.

However, two problems arise:

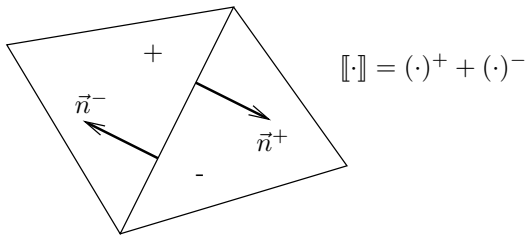


Figure 4.5.1: Definition of the jump operator, used for quantities that involve the normal vector.

1. The bilinear form is not symmetric, $B(u,v) \neq B(v,u)$. This itself is not a show-stopper, but it is not reassuring given the SPD form of the continuous discretization.
2. The bilinear form is not stable/coercive. Consider a v that is 1 on element κ and 0 everywhere else. We then have $B(v,v) = 0$. This means that the discretization is singular, i.e. that the stiffness matrix is not invertible.

Conversion to a first-order system More success is attained by converting Equation 4.5.1 to a system of first-order equations,

$$\begin{aligned} \nabla u &= \vec{\sigma}, \\ -\nabla \cdot \vec{\sigma} &= f. \end{aligned}$$

$\vec{\sigma}$ is a new variable, the viscous flux. We define finite-dimensional spaces for u_h and $\vec{\sigma}_h$, which are the discrete approximations of our unknowns,

$$\begin{aligned} u_h \in \mathcal{V}_h^p &= \left\{ v \in L^2(\Omega) : v|_\kappa \in P^p(\kappa) \;\; \forall \kappa \in T_h \right\}, \\ \vec{\sigma}_h \in \Sigma_h^p &= [\mathcal{V}_h^p]^{\dim}. \end{aligned}$$

The order of approximation is assumed to be the same (p) in both spaces. We now dot the first equation with a test function $\vec{\tau}_h \in \Sigma_h^p$, and we multiply the second equation by a test function $v_h \in \mathcal{V}_h^p$. Integrating each result once by parts gives

$$-\int_\kappa u_h \nabla \cdot \vec{\tau}_h d\kappa + \int_{\partial\kappa} \hat{u}_h \vec{\tau}_h \cdot \vec{n} ds = \int_\kappa \vec{\sigma}_h \cdot \vec{\tau}_h d\kappa, \tag{4.5.10}$$

$$\int_\kappa \vec{\sigma}_h \cdot \nabla v_h d\kappa - \int_{\partial\kappa} v_h \hat{\vec{\sigma}}_h \cdot \vec{n} ds = \int_\kappa f v_h d\kappa, \tag{4.5.11}$$

where $\hat{u}_h$ is the flux function for u_h, and $\hat{\vec{\sigma}}$ is the flux function for $\vec{\sigma}$.

First form of Bassi and Rebay (BR1) In 1997, Bassi and Rebay [14] introduced a diffusion discretization based on Equations 4.5.10 and 4.5.11 that uses the following simple average fluxes,

$$\hat{u}_h = \{u_h\}, \tag{4.5.12}$$

$$\hat{\vec{\sigma}}_h = \{\vec{\sigma}_h\}. \tag{4.5.13}$$

Boundary conditions are enforced weakly by setting $\hat{u}_h = u_D$ on the boundaries.

Since we have two variables, $u_h, \vec{\sigma}_h$, it appears that we have increased the number of unknowns by a factor of dim+1 ($\vec{\sigma}_h$ is a spatial vector). However, we can eliminate $\vec{\sigma}_h$ by solving the first equation locally. That is, in Equation 4.5.10, $\vec{\sigma}_h$ comes in only on the right-hand side, giving a mass-matrix term multiplying the unknown coefficients on $\vec{\sigma}_h$. Inverting the mass matrix gives an expression for $\vec{\sigma}_h$ in terms of u_h, which is then substituted into Equation 4.5.11. As a result, only the unknowns associated with u_h need to be stored.

One problem with BR1 is that the resulting discretization is not compact. That is, unknowns in an element are coupled to unknowns in more elements than just the immediate neighbors. From Equation 4.5.10, we see that $\vec{\sigma}_h$ on element κ is a function of u_h on κ and its neighbors. Consider then the second term in Equation 4.5.11: the averaging of $\vec{\sigma}_h$ on an element interface couples together the states on not only the elements adjacent to the interface, but also the states on the next layer of elements, as shown in Figure 4.5.2.

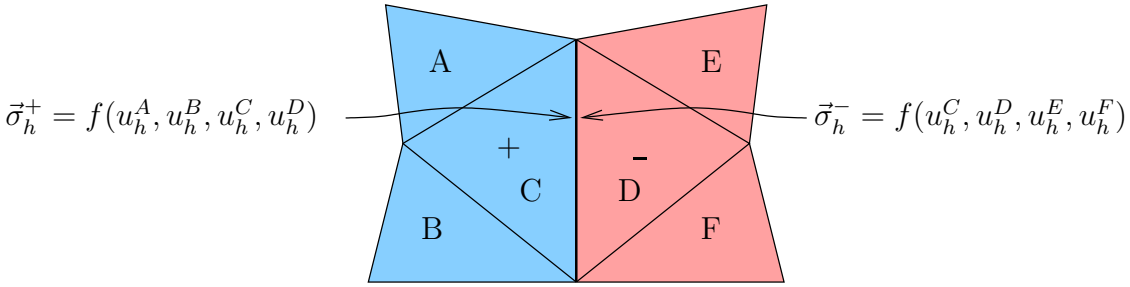


Figure 4.5.2: In BR1, the average $\vec{\sigma}_h$ on an edge depends on the states in all five elements, A-F.

Lack of a compact stencil makes application of implicit methods cumbersome and expensive, especially for parallel implementations. Furthermore, the bilinear form that results after eliminating $\vec{\sigma}_h$ is not guaranteed to be stable, as analysis shows that $B(v_h, v_h) \geq 0$ [6]. Finally, BR1 exhibits suboptimal convergence rates for even orders p: the L^2 error norm is $\mathcal{O}(h^p)$ instead of $\mathcal{O}(h^{p+1})$.

Local discontinuous Galerkin (LDG) In 1998, Cockburn and Shu [34] generalized the fluxes of BR1 as

$$\hat{u}_h = \{u\}_h - \vec{C}_{12} \cdot [\![u_h \vec{n}]\!], \tag{4.5.14}$$

$$\hat{\vec{\sigma}}_h = \{\vec{\sigma}\}_h - C_{11}[\![u_h \vec{n}]\!] + \vec{C}_{12}[\![\vec{\sigma} \cdot \vec{n}]\!], \tag{4.5.15}$$

where C_{11} is a positive constant, and $\vec{C}_{12}$ is a vector of constants specific to each face. The choice that results in the smallest stencil is $\vec{C}_{12} = \frac{1}{2}[\![S\vec{n}]\!]$, where $0 \leq S \leq 1$ is a "switch operator" that satisfies $\{S\} = \frac{1}{2}$. S is usually chosen to make the stencil as small as possible. However, the resulting stencil still cannot be made nearest-neighbor compact when the spatial dimension is higher than one.

Nevertheless, LDG was a big improvement over existing methods when it was introduced, as it is provably consistent and stable, and as it exhibits optimal convergence rates. Note

that $\vec{\sigma}_h$ can again be eliminated as in BR1 (this gave rise to the name "local" discontinuous Galerkin).

Compact discontinuous Galerkin (CDG) In 2007, Peraire and Persson [117] introduced CDG: a variant of LDG that maintains a compact stencil. $\hat{u}_h$ remains the same as in LDG, but $\hat{\vec{\sigma}}_h$ is modified to remove the troublesome $\{\vec{\sigma}_h\}$ term according to

$$\hat{\vec{\sigma}}_h = \{\vec{\sigma}_h^e\} - C_{11}[\![u_h\vec{n}]\!] + \vec{C}_{12}[\![\sigma_h^e \cdot \vec{n}]\!], \tag{4.5.16}$$

where $\vec{\sigma}_h^e = \nabla u_h + \bar{\sigma}_h^e$, and $\bar{\sigma}_h^e$ is a one-sided viscous stress field obtained by lifting jumps on a face to the element interior.

Second form of Bassi and Rebay (BR2) One of the first diffusion discretizations with a compact stencil came from Bassi and Rebay in 2000. In this "BR2" method [15], the fluxes in Equation 4.5.10 and Equation 4.5.11 are chosen as

$$\hat{u}_h = \{u_h\}, \tag{4.5.17}$$
$$\hat{\vec{\sigma}}_h = \{\nabla u_h\} - \eta_f\{\vec{\delta}_f\}, \tag{4.5.18}$$

where $\vec{\delta}_f \in \Sigma_h^p$ is a vector field associated with one face, and with support over two elements adjacent to the face. It is defined according to

$$\int_{\kappa^+} \vec{\delta}_f \cdot \vec{\tau}_h d\kappa + \int_{\kappa^-} \vec{\delta}_f \cdot \vec{\tau}_h d\kappa = \int_{\text{face f}} \{\vec{\tau}\} \cdot [\![u\vec{n}]\!] ds. \tag{4.5.19}$$

That is, $\vec{\delta}_f$ is driven by the jump of the state across the face. Note that the units of $\vec{\delta}_f$ are u per unit length, owing to the volume integrals on the left-hand side, and the face integral on the right-hand side.

A compact stencil is attained because $\hat{\vec{\sigma}}_h$ does not depend on $\vec{\sigma}_h$. The method is provably stable when the nondimensional stabilization factor, η_f, associated with face f is greater than or equal to the maximum number of faces on the two adjacent elements. It also exhibits optimal convergence rates of $\mathcal{O}(h^{p+1})$.

We give a little more detail for the BR2 discretization by eliminating $\vec{\sigma}_h$ from Equations 4.5.10 and 4.5.11. Choosing $\vec{\tau} = \nabla v_h$ in Equation 4.5.10, we see that the right-hand side of Equation 4.5.10 is the first term of Equation 4.5.11. Making the substitution, Equation 4.5.11 becomes

$$-\int_\kappa u_h \nabla \cdot (\nabla v_h) d\kappa + \int_{\partial\kappa} \hat{u}_h \nabla v_h \cdot \vec{n} ds - \int_{\partial\kappa} v_h \hat{\vec{\sigma}}_h \cdot \vec{n} ds = \int_\kappa f v_h d\kappa. \tag{4.5.20}$$

Integrating the first term by parts and using $^+$ and $^-$ to clearly denote terms taken from, respectively, the interior and exterior of element κ, we obtain the following elemental contribution to the bilinear form,

$$\int_\kappa \nabla u_h \cdot \nabla v_h d\kappa - \int_{\partial\kappa} (u_h^+ - \hat{u}_h) \nabla v_h^+ \cdot \vec{n} ds - \int_{\partial\kappa} v_h^+ \hat{\vec{\sigma}}_h \cdot \vec{n} ds = \int_\kappa f v_h d\kappa. \tag{4.5.21}$$

Interior penalty (IP) This "poor-man's" DG diffusion discretization has been used in various forms for many years. The fluxes are similar to BR2, except that the $\vec{\delta}_f$ definition is much less complicated. Specifically,

$$\vec{\delta}_f^+ = \frac{1}{h^+}[\![u\vec{n}]\!], \qquad \vec{\delta}_f^- = \frac{1}{h^-}[\![u\vec{n}]\!], \tag{4.5.22}$$

where h^+/h^- are the "heights" of the elements adjacent to face f, defined as the element volume divided by the face area[1]. This simple definition requires no pre-computation of $\vec{\delta}_f$ and hence is much easier to implement. The stencil is compact and optimal orders of convergence can be attained, for suitable factors η_f. An initial lack of rigorous bounds for η_f has been addressed [131], but a negative aspect that remains is increased stiffness (i.e. higher condition numbers) of IP discretizations compared to others such as BR2.

Recovery discontinuous Galerkin (RDG) Introduced by van Leer, Lo, and Raalte in 2007 [146], this discretization is based directly on the second-order system in Equation 4.5.1. The required values of state gradients at element interfaces are obtained by *reconstructing* (recovering) a smooth higher-order function for the state on the two adjacent elements, using the discontinuous states on each element. The resulting stencil is not compact, stability on unstructured meshes depends on aspects of the reconstruction operator. The convergence rate in $L^2(\Omega)$ is $\mathcal{O}(h^{p+1})$, while element mean quantities superconverge at up to $\mathcal{O}(h^{2p+1})$.

4.5.2 1D Implementation of BR2

For the model differential equation in 1D, we consider the scalar unsteady Laplace equation on a uniform grid with periodic boundaries,

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = 0 \tag{4.5.23}$$

where

$$t \in [0, T], \quad x \in [0, L], \quad u(x + L, t) = u(x, t), \quad u(x, t = 0) = u_0(x)$$

BR2 bilinear form Recall the BR2 bilinear form in Equation 4.5.21,

$$B(u, v) = \int_\kappa \nabla v \cdot \nabla u d\kappa - \int_{\partial\kappa} (u^+ - \hat{u}) \nabla v^+ \cdot \vec{n} ds - \int_{\partial\kappa} v^+ \hat{\vec{\sigma}} \cdot \vec{n} ds,$$

where u is the solution, and v is the test function. In one dimension, the contribution to the n^{th} discrete residual on the k^{th} element ($x \in [x_{k-1/2}, x_{k+1/2}]$) is

$$R_{k,n} = \underbrace{\int_{x_{k-1/2}}^{x_{k+1/2}} \frac{\partial \phi_{k,n}}{\partial x} \frac{\partial u}{\partial x} dx}_{\text{Term I}} \underbrace{- \left[(u^+ - \hat{u}) \frac{\partial \phi_{k,n}^+}{\partial x} \right]_{x_{k-1/2}}^{x_{k+1/2}}}_{\text{Term II}} \underbrace{- \left[\phi_{k,n}^+ \hat{\sigma} \right]_{x_{k-1/2}}^{x_{k+1/2}}}_{\text{Term III}}, \tag{4.5.24}$$

[1] In two dimensions, this is the element area divided by the edge length.

where $(\cdot)^+/(\cdot)^-$ denotes a quantity taken from the interior/exterior of element k. The fluxes are

$$\hat{u} = \frac{1}{2}\left(u^+ + u^-\right), \qquad \hat{\sigma} = \frac{1}{2}\left(\frac{\partial u^+}{\partial x} + \frac{\partial u^-}{\partial x}\right) \underbrace{-\eta\frac{1}{2}(\delta^+ + \delta^-)}_{\text{stabilization}}, \tag{4.5.25}$$

where $\eta = 2$ in our 1D case. Our convention at an interface (node) will be that $^+$ refers to the left element while $^-$ refers to the right element adjacent to the node. At $x_{k+1/2}$, δ^+ and δ^- are obtained by solving for δ_k and δ_{k+1}, which give the δ variable on each element,

$$\int_{x_{k-1/2}}^{x_{k+1/2}} \phi_{k,l}\delta_k dx = \left[\frac{1}{2}\phi_{k,l}(u^+ - u^-)\right]_{x_{k+1/2}}, \tag{4.5.26}$$

$$\int_{x_{k+1/2}}^{x_{k+3/2}} \phi_{k+1,l}\delta_{k+1} dx = \left[\frac{1}{2}\phi_{k+1,l}(u^+ - u^-)\right]_{x_{k+1/2}}. \tag{4.5.27}$$

Note that δ^+ and δ^- are approximated with the same basis as u on the two elements adjacent to the node. Then, $\delta^+ = \delta_k$ evaluated at the right node, and $\delta^- = \delta_{k+1}$ evaluated at the left node.

Stiffness matrix assembly The state $u(x)$ on element k is approximated via

$$u(x) = \sum_m U_{k,m}\phi_{k,m}(x)$$

where m ranges over all basis functions in a single element, and $U_{k,m}$ are the discrete unknowns.

The problem is defined on a uniform grid (all elements the same), with periodic boundary conditions. Since BR2 has a compact stencil, we just need to form three matrices (same for all k),

$$A^0_{n,m} = \frac{\partial R_{k,n}}{\partial U_{k,m}}, \qquad A^L_{n,m} = \frac{\partial R_{k,n}}{\partial U_{k-1,m}}, \qquad A^R_{n,m} = \frac{\partial R_{k,n}}{\partial U_{k+1,m}}.$$

We consider the three terms in Equation 4.5.24 separately.

Term I

This contribution only affects $A^0_{n,m}$,

$$R^I_{k,n} = \left[\int_{x_{k-1/2}}^{x_{k+1/2}} \frac{\partial\phi_{k,n}}{\partial x}\frac{\partial\phi_{k,m}}{\partial x}dx\right] U_{k,m} \quad \Rightarrow \quad A^{0,I}_{n,m} = \int_{x_{k-1/2}}^{x_{k+1/2}} \frac{\partial\phi_{k,n}}{\partial x}\frac{\partial\phi_{k,m}}{\partial x}dx$$

Term II

Using the given flux, $\hat{u}$, this term is

$$R^{II}_{k,n} = -\left[\frac{1}{2}(u^+ - u^-)\frac{\partial\phi^+_{k,n}}{\partial x}\right]_{x_{k-1/2}}^{x_{k+1/2}}$$

Define the following quantities, which are the same for all elements:

ϕ_m^L = basis functions evaluated on left node of an element
ϕ_m^R = basis functions evaluated on right node of an element
$\frac{\partial \phi_m^L}{\partial x}$ = basis function gradients evaluated on left node of an element
$\frac{\partial \phi_m^R}{\partial x}$ = basis function gradients evaluated on right node of an element

Linearizing the above residual contribution, we obtain

$$A_{n,m}^{0,II} = -\frac{1}{2}\frac{\partial \phi_n^R}{\partial x}\phi_m^R + \frac{1}{2}\frac{\partial \phi_n^L}{\partial x}\phi_m^L, \qquad A_{n,m}^{L,II} = -\frac{1}{2}\frac{\partial \phi_n^L}{\partial x}\phi_m^R, \qquad A_{n,m}^{R,II} = \frac{1}{2}\frac{\partial \phi_n^R}{\partial x}\phi_m^L$$

Term III

Linearizing this last term, we have

$$A_{n,m}^{0,III} = -\phi_n^R\frac{\partial \hat{\sigma}}{\partial U_m^+} + \phi_n^L\frac{\partial \hat{\sigma}}{\partial U_m^-}, \qquad A_{n,m}^{L,III} = \phi_n^L\frac{\partial \hat{\sigma}}{\partial U_m^+}, \qquad A_{n,m}^{R,III} = -\phi_n^R\frac{\partial \hat{\sigma}}{\partial U_m^-}$$

The linearization of σ is obtained from Equation 4.5.25,

$$\frac{\partial \hat{\sigma}}{\partial U_m^+} = \frac{1}{2}\frac{\partial \phi_m^R}{\partial x} - \eta\frac{1}{2}\left(\frac{\partial \delta^+}{\partial U_m^+} + \frac{\partial \delta^-}{\partial U_m^+}\right) \tag{4.5.28}$$

$$\frac{\partial \hat{\sigma}}{\partial U_m^-} = \frac{1}{2}\frac{\partial \phi_m^L}{\partial x} - \eta\frac{1}{2}\left(\frac{\partial \delta^+}{\partial U_m^-} + \frac{\partial \delta^-}{\partial U_m^-}\right) \tag{4.5.29}$$

The linearization of δ is obtained from Equations 4.5.26 and 4.5.27, by first explicitly writing the approximation via basis functions,

$$\delta^+ = D_i^+\phi_i^R, \qquad \delta^- = D_i^-\phi_i^L,$$

where summation is implied on repeated indices and

$$D_i^+ = \frac{1}{2}M_{il}^{-1}\phi_l^R(u^+ - u^-), \quad D_i^- = \frac{1}{2}M_{il}^{-1}\phi_l^L(u^+ - u^-).$$

The required derivatives are then

$$\frac{\partial \delta^+}{\partial U_m^+} = \frac{1}{2}\phi_i^R M_{il}^{-1}\phi_l^R\phi_m^R, \quad \frac{\partial \delta^+}{\partial U_m^-} = -\frac{1}{2}\phi_i^R M_{il}^{-1}\phi_l^R\phi_m^L,$$
$$\frac{\partial \delta^-}{\partial U_m^+} = \frac{1}{2}\phi_i^L M_{il}^{-1}\phi_l^L\phi_m^R, \quad \frac{\partial \delta^-}{\partial U_m^-} = -\frac{1}{2}\phi_i^L M_{il}^{-1}\phi_l^L\phi_m^L.$$

4.6 Analysis

In this section we analyze the stability of the DG discretization for one-dimensional advection and advection-diffusion.

4.6.1 1D Advection

Discretization We consider constant-velocity scalar transport with periodic boundaries,

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, \qquad x \in [0,1], \quad u(x+1,t) = u(x,t). \tag{4.6.1}$$

For the discretization, we use DG on N equal elements, each of size $\Delta x = 1/N$, Lagrange basis functions of order p on each element, and pure upwinding for advection. The resulting system of ODEs is

$$\mathbf{M}\frac{d\mathbf{U}}{dt} + \mathbf{A}\mathbf{U} = 0, \tag{4.6.2}$$

where $\mathbf{U}$ is a $Nn_p \times 1$ vector, and $\mathbf{M}$ and $\mathbf{A}$ are sparse $Nn_p \times Nn_p$ matrices. $n_p = p+1$ is the number of degrees of freedom per element.

Stability analysis Multiplying Equation 4.6.2 on the left by $\mathbf{M}^{-1}$ gives the standard ODE form,

$$\frac{d\mathbf{U}}{dt} = \underbrace{-\mathbf{M}^{-1}\mathbf{A}}_{\mathbf{C}}\mathbf{U} \tag{4.6.3}$$

To determine stability of time integration we need to look at the eigenvalues of $\mathbf{C} \equiv -\mathbf{M}^{-1}\mathbf{A}$. We calculate these eigenvalues numerically.

We could form the entire $\mathbf{M}$ and $\mathbf{A}$ matrices, but even using sparse formats, the storage could become excessive for a large number of elements.

Taking advantage of the periodic boundary conditions, we realize that the eigenvectors of $\mathbf{C}$ are sinusoidal over the elements. That is, the mode m eigenvector on element k is

$$\mathbf{V}_k = \mathbf{V}_0 e^{ik\theta_m}, \quad \theta_m = 2\pi m \Delta x, \quad m \in [-N/2+1, N/2]$$

Note that $\mathbf{V}_0$ is a vector of $p+1$ numbers. Looking at the rows associated with element k in Equation 4.6.3, the eigenvalue problem is

$$\begin{aligned} \Big[-\mathbf{M}_k^{-1}(\mathbf{A}_{k,k-1}\mathbf{V}_{k-1} + \mathbf{A}_{k,k}\mathbf{V}_k) &= \lambda \mathbf{V}_k \Big] \times e^{-ik\theta_m} \\ \underbrace{-\mathbf{M}_k^{-1}(\mathbf{A}_{k,k-1}e^{-i\theta_m} + \mathbf{A}_{k,k})}_{\mathbf{C}_m}\mathbf{V}_0 &= \lambda \mathbf{V}_0, \end{aligned}$$

where we use the structure of $\mathbf{M}$ and $\mathbf{A}$ from Equations 4.2.13 and 4.2.20. We obtain $N(p+1)$ eigenvalues by calculating $p+1$ eigenvalues of $\mathbf{C}_m$, a $(p+1)\times(p+1)$ matrix, for each value of m (N in total).

Results We now numerically compute and show the eigenvalues of $\mathbf{C} = -\mathbf{M}^{-1}\mathbf{A}$ for different combinations of approximation order p and number of elements N.

$p = 0$ versus N

Consider $p = 0$ with a varying number of elements. Figure 4.6.1 shows the eigenvalues of $\mathbf{C}$ in the complex number plane. These eigenvalues are the same as ones obtained using an upwind finite difference or volume method: for a given N, they form circles in the left-half plane. The maximum magnitude eigenvalue grows linearly with $1/\Delta x = N$. This is expected based on the analytic eigenvalues presented in Section 1.4.1. The minimum magnitude eigenvalues (excluding $\lambda = 0$) approach the analytical result $\lambda_{\text{min}} = \pm i2\pi$ at $\mathcal{O}(\Delta x^2)$. The circular distribution of the eigenvalues indicates that forward Euler is a reasonable explicit time integration method in this case.

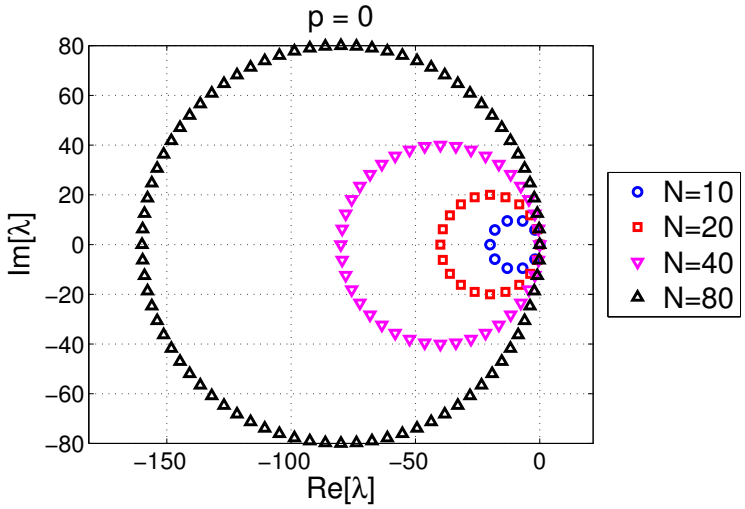


Figure 4.6.1: System eigenvalues for DG advection using $p = 0$.

$p = 1$ versus N

Figure 4.6.2 shows the system eigenvalues for $p = 1$ and various N. For a given N, the elongated shape of the eigenvalue distribution near the origin (i.e. eigenvalues closer to the imaginary axis) indicates less numerical dissipation. As in the $p = 0$ case, $\lambda_{\text{max}} \sim 1/\Delta x$. However, λ_{min} now approaches $\pm i2\pi$ faster ($\mathcal{O}(\Delta x^4)$), indicating improved accuracy over $p = 0$. Particularly for large N, forward Euler is not a reasonable time integration method in this case because it lacks stability along the imaginary axis. More suitable are higher-order time integration methods such as RK2 or RK4, with better stability along the imaginary axis.

$p = 2$ versus N

Figure 4.6.3 shows the system eigenvalues for $p = 2$. Compared to $p = 1$, the eigenvalue distribution for a given N is even more elongated near the origin, which indicates even less numerical dissipation. The maximum eigenvalue still scales as $\lambda_{\text{max}} \sim 1/\Delta x$, and the

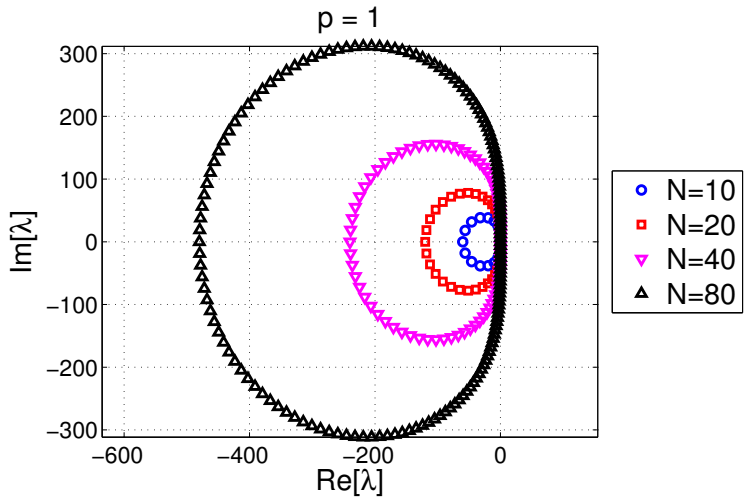


Figure 4.6.2: System eigenvalues for DG advection using $p = 1$.

minimum eigenvalue, λ_{min}, approaches $\pm i2\pi$ faster, at $\mathcal{O}(\Delta x^6)$, indicating improved accuracy over $p = 1$ and a superconvergence property. As in the case of $p = 1$, Forward Euler is not a reasonable time integration method.

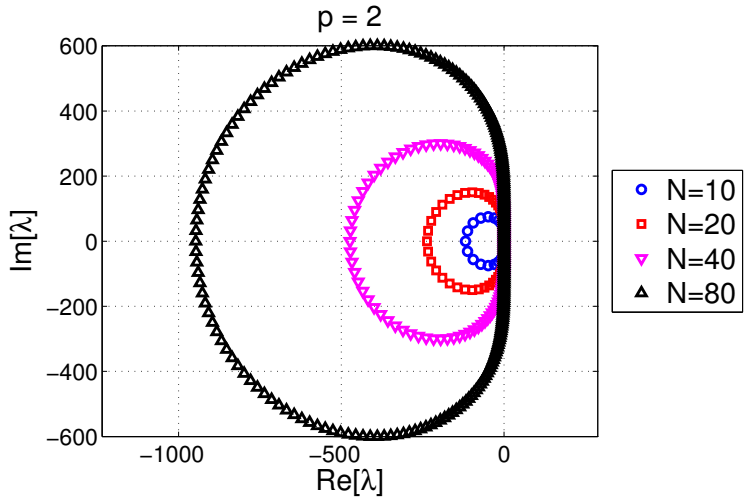
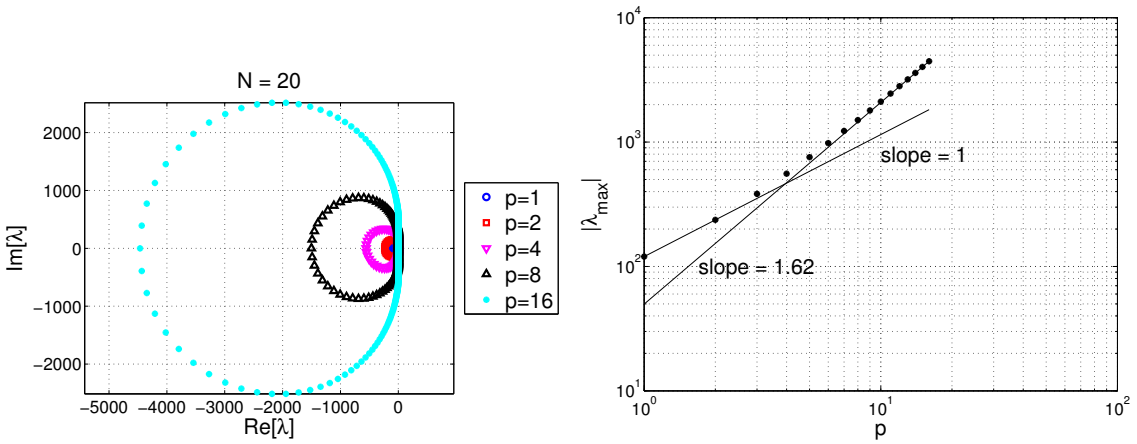


Figure 4.6.3: System eigenvalues for DG advection using $p = 2$.

$N = 20$ versus p

We now look at the behavior of eigenvalues at a fixed N, with increasing p. Figure 4.6.4 shows the distribution of eigenvalues for $p = 1, 2, 4, 8, 16$, and the scaling of the maximum eigenvalue magnitude versus p.

We see that for small p, $|\lambda_{\text{max}}| \propto p$. For larger p, $|\lambda_{\text{max}}| \propto p^{1.62}$ – this result is quite



(a) Distribution (b) Maximum magnitude

Figure 4.6.4: Distribution of eigenvalues and magnitude of the maximum eigenvalue versus order p for $N = 20$ elements.

insensitive to N. As order increases, the clustering of eigenvalues on the imaginary axis near the origin requires implicit treatment or an explicit time-marching scheme with eigenvalue stability along the imaginary axis, such as RK4.

Independence of basis The above eigenvalue analysis was carried out using a Lagrange basis on Gauss-Legendre nodes; however, the results hold for any other (polynomial) basis. Suppose we had used basis functions $\widetilde{\phi}_j$ instead of ϕ_i. Assuming both sets of basis functions span the same space, order-p polynomials, we can write

$$\phi_i = \sum_j T_{i,j} \widetilde{\phi}_j, \tag{4.6.4}$$

for some invertible transformation matrix $\mathbf{T} = T_{i,j}$. Note that the indices i, j range over all degrees of freedom (e.g. $\mathbf{U} = U_i$, $\widetilde{\mathbf{U}} = \widetilde{U}_j$). The discretized system would then read

$$\widetilde{\mathbf{M}} \frac{d\widetilde{\mathbf{U}}}{dt} + \widetilde{\mathbf{A}}\widetilde{\mathbf{U}} = \mathbf{0}, \tag{4.6.5}$$

where the original mass matrix is related to the new mass matrix via

$$M_{i,j} = \int_\Omega \phi_i \phi_j d\Omega = \sum_k \sum_l \int_\Omega T_{i,k} \widetilde{\phi}_k T_{j,l} \widetilde{\phi}_l d\Omega = \sum_k \sum_l T_{i,k} T_{j,l} \widetilde{M}_{kl}. \tag{4.6.6}$$

In matrix form, this equation is

$$\mathbf{M} = \mathbf{T}\widetilde{\mathbf{M}}\mathbf{T}^T. \tag{4.6.7}$$

Similarly, $\mathbf{A} = \mathbf{T}\widetilde{\mathbf{A}}\mathbf{T}^T$. Stability of time integration of Equation 4.6.5 is determined by the eigenvalues of

$$\widetilde{\mathbf{C}} = -\widetilde{\mathbf{M}}^{-1}\widetilde{\mathbf{A}} = -\mathbf{T}^T\mathbf{M}^{-1}\mathbf{T}\mathbf{T}^{-1}\mathbf{A}\mathbf{T}^{-T} = -\mathbf{T}^T\mathbf{M}^{-1}\mathbf{A}\mathbf{T}^{-T} = \mathbf{T}^T\mathbf{C}\mathbf{T}^{-T}$$

Since the eigenvalues of $\mathbf{T}^T\mathbf{C}\mathbf{T}^{-T}$ are the same as the eigenvalues of $\mathbf{C}$, the stability analysis is independent of basis.

4.6.2 1D Advection-Diffusion

Discretization For advection-diffusion, the model PDE is

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2} = 0, \qquad x \in [0,1], \quad u(x+1,t) = u(x,t).$$

In the DG discretization, we use N equal elements, each of size $\Delta x = L/N$, where $L = 1$, Lagrange basis functions of order p on each element, pure upwinding for the advection flux, and BR2 with $\eta = 2$ for the diffusive flux. The resulting system is of the same form as Equation 4.6.2,

$$\mathbf{M}\frac{d\mathbf{U}}{dt} + \mathbf{A}\mathbf{U} = 0,$$

where $\mathbf{U}$ is a $Nn_p \times 1$ vector, and $\mathbf{M}$ and $\mathbf{A}$ are sparse $Nn_p \times Nn_p$ matrices. $n_p = p+1$ is the number of degrees of freedom per element.

Stability analysis Following the 1D advection stability analysis, the eigenvalue problem is

$$\begin{aligned}\Big[-\mathbf{M}_k^{-1}(\mathbf{A}_{k,k-1}\mathbf{V}_{k-1} + \mathbf{A}_{k,k}\mathbf{V}_k + \mathbf{A}_{k,k+1}\mathbf{V}_{k+1}) \;&=\; \lambda\mathbf{V}_k\Big] \times e^{-ik\theta_m} \\ \underbrace{-\mathbf{M}_k^{-1}(\mathbf{A}_{k,k-1}e^{-i\theta_m} + \mathbf{A}_{k,k} + \mathbf{A}_{k,k+1}e^{i\theta_m})}_{\mathbf{C}_m}\mathbf{V}_0 \;&=\; \lambda\mathbf{V}_0, \end{aligned} \tag{4.6.8}$$

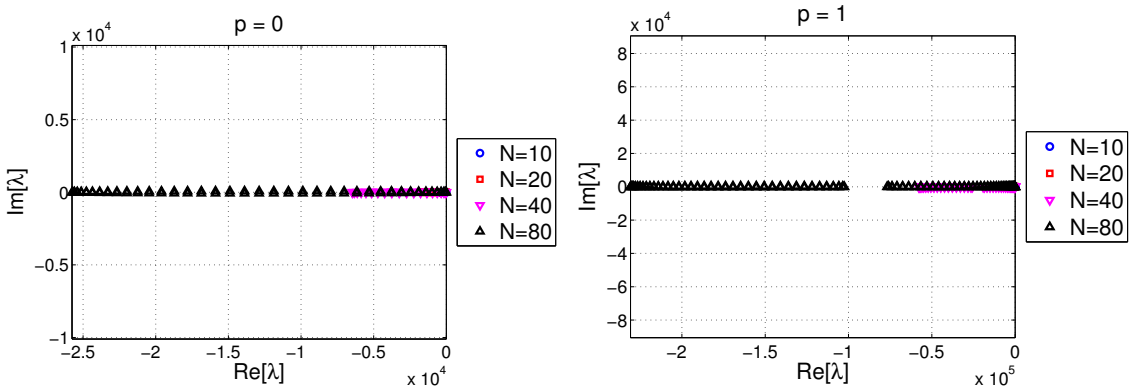
where we make use of the block structure of $\mathbf{M}$ and $\mathbf{A}$ from advection, augmented with $\mathbf{A}_{k,k+1}$ from the BR2 discretization. We obtain $N(p+1)$ eigenvalues by calculating $p+1$ eigenvalues of $\mathbf{C}_m$, a $(p+1)\times(p+1)$ matrix, for each value of m (N in total).

Results We now show results for the distribution of the eigenvalues of $\mathbf{C}$, for different **Peclet** numbers, defined as

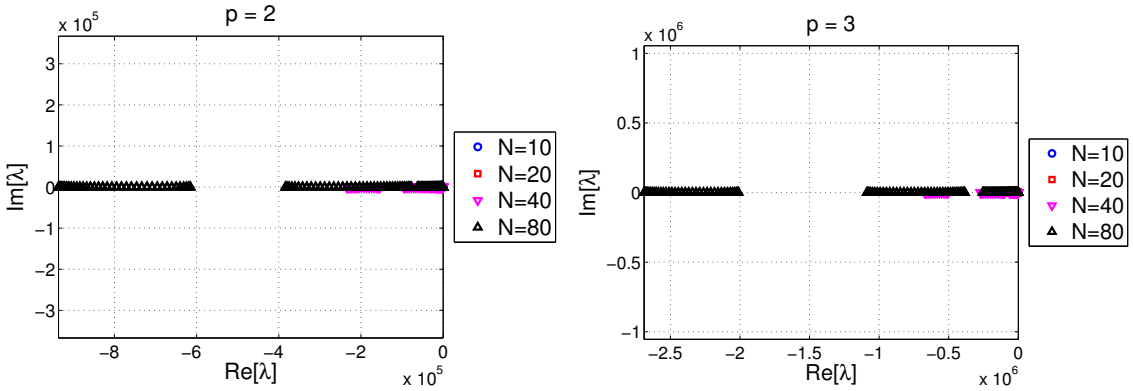
$$Pe \;\equiv\; \frac{aL}{\nu}. \tag{4.6.9}$$

$\underline{Pe = 1}$

Figure 4.6.5 shows the distribution of eigenvalues for the case $Pe = 1$. This situation corresponds to a case where the diffusive fluxes approximately balance the convective fluxes.



(a) $p = 0$ (b) $p = 1$



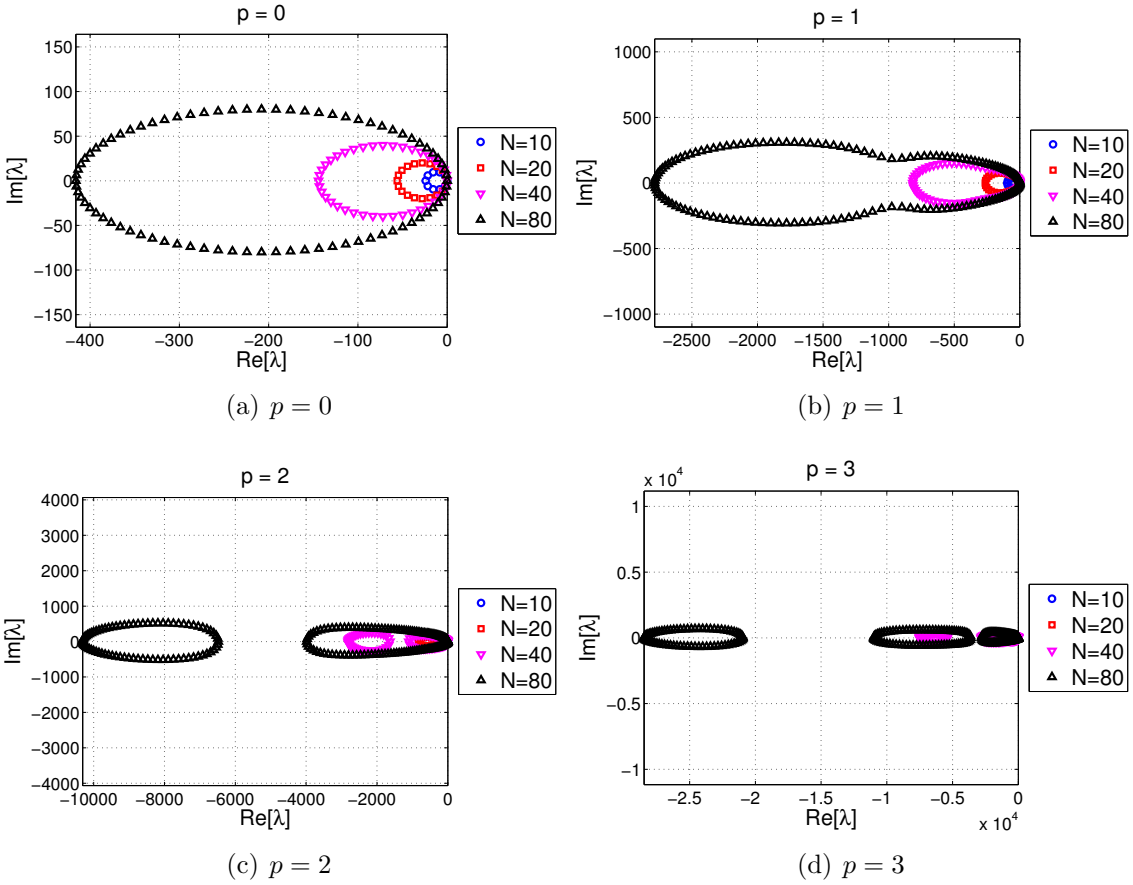
(c) $p = 2$ (d) $p = 3$

Figure 4.6.5: Distribution of eigenvalues for advection-diffusion at $Pe = 1$.

We see that the eigenvalues are distributed mostly on the negative real axis, and that they have a large magnitude real component. In particular, the maximum eigenvalue magnitude scales quadratically with the number of elements, consistent with the analysis in Section 1.4.1.

$Pe = 100$

Figure 4.6.5 shows the distribution of eigenvalues for $Pe = 100$. This corresponds to a decrease in the amount of diffusion compared to $Pe = 1$, e.g. a 100-fold reduction in the viscosity, ν. We now observe a larger magnitude imaginary component of the eigenvalues compared to $Pe = 1$. The maximum eigenvalue magnitude is smaller than for $Pe = 1$, but it still scales quadratically with the number of elements.



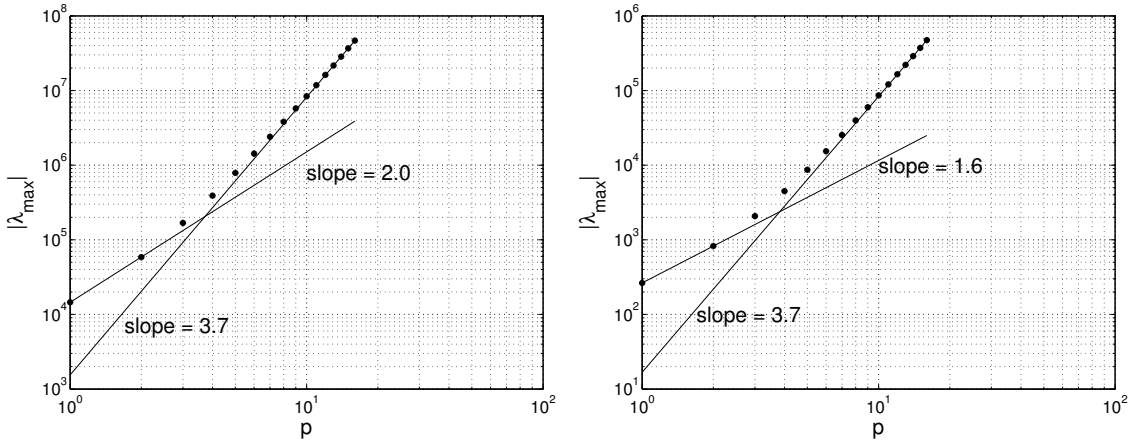
(a) $p = 0$ (b) $p = 1$

(c) $p = 2$ (d) $p = 3$

Figure 4.6.6: Distribution of eigenvalues for advection-diffusion at $Pe = 100$.

Eigenvalue behavior with p, for $N = 20$

Figure 4.6.7 shows the growth of the largest-magnitude eigenvalue with increasing p, for two Peclet numbers: $Pe = 1$ and $Pe = 100$. On the log-log scales, we see that the initial growth of the maximum eigenvalue is slower than the final growth. In addition, the initial slope decreases with increasing Pe, due to the increased effect of advection relative to diffusion. The final slope is 3.7 in both cases, and this is fairly independent of N.



(a) $Pe = 1$ (b) $Pe = 100$

Figure 4.6.7: Distribution of eigenvalues for advection-diffusion at $Pe = 100$.

Chapter 5

Adjoints, Error Estimation, and Adaptation

Several comprehensive reviews exist on the use of adjoint methods in computational fluid dynamics, including in estimating output errors [47, 44]. This chapter introduces the adjoint equations and presents the role of the adjoint in output error estimation and mesh adaptation.

5.1 Adjoint Equations

Consider a discretized PDE,

$$\mathbf{R}(\mathbf{U}) \;=\; \mathbf{0}, \tag{5.1.1}$$

where $\mathbf{U} \in \mathbb{R}^N$ is the discrete state vector, and $\mathbf{R}$ are N residuals, i.e. the N equations of the discrete system. Suppose that we are interested in a **scalar output**, J, computed from the solution, $\mathbf{U}$,

$$\text{output } J \;=\; J(\mathbf{U}). \tag{5.1.2}$$

The **discrete adjoint**, $\boldsymbol{\Psi} \in \mathbb{R}^N$, is a vector of sensitivities of the output to the N residuals. That is, each entry of the adjoint tells us the effect that a perturbation in the same entry in the residual vector, i.e. an addition to one of the equations, would have on the output J. One common source of residual perturbations is changes in input parameters for a problem, and so we ground the adjoint presentation in the context of a local sensitivity analysis.

5.1.1 Local Sensitivity Analysis

Consider a situation in which N_μ parameters, $\boldsymbol{\mu} \in \mathbb{R}^{N_\mu}$, affect the PDE in Equation 5.1.1. For example, in aerodynamics, parameters could be used to set boundary conditions or the geometry, as illustrated schematically in Figure 5.1.1. We can then write the following chain of dependence,

$$\underbrace{\boldsymbol{\mu}}_{\text{inputs} \in \mathbb{R}^{N_\mu}} \rightarrow \underbrace{\mathbf{R}(\mathbf{U}, \boldsymbol{\mu}) = 0}_{N \text{ equations}} \rightarrow \underbrace{\mathbf{U}}_{\text{state} \in \mathbb{R}^N} \rightarrow \underbrace{J(\mathbf{U})}_{\text{output (scalar)}} . \tag{5.1.3}$$

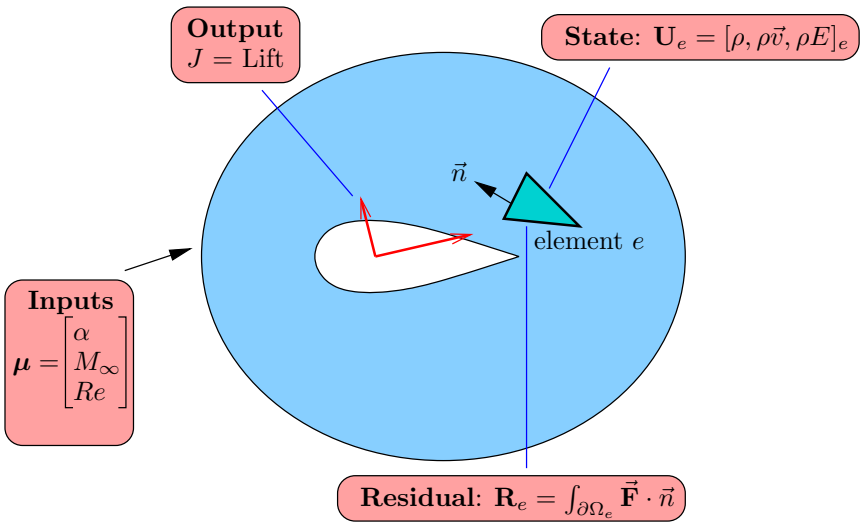


Figure 5.1.1: Schematic of typical inputs, state, residual, and output quantities in an aerodynamics simulation.

We are interested in how J changes (locally for nonlinear problems) with $\boldsymbol{\mu}$,

$$\frac{dJ}{d\boldsymbol{\mu}} \in \mathbb{R}^{1\times N_\mu} = N_\mu \text{ sensitivities.} \tag{5.1.4}$$

Note, if J depends directly on $\boldsymbol{\mu}$ we would add $\frac{\partial J}{\partial \boldsymbol{\mu}}$ to the above sensitivity, but for clarity of presentation we consider only the case when $J = J(\mathbf{U})$. Several options exist for computing these sensitivities. Two direct ones are finite differencing, in which the input parameters are perturbed one at a time, and forward linearization, in which the sequence of operations in Equation 5.1.3 is linearized. Both of these become expensive when N_μ is moderate or large, because of the need to re-solve the system $\mathbf{R}(\mathbf{U}, \boldsymbol{\mu}) = 0$ for each parameter. A third choice is the adjoint approach, which requires an inexpensive residual perturbation calculation followed by an adjoint weighting to compute the effect on the output. That is, we write

$$\frac{dJ}{d\boldsymbol{\mu}} = \boldsymbol{\Psi}^T \frac{\partial \mathbf{R}}{\partial \boldsymbol{\mu}}. \tag{5.1.5}$$

This approach is efficient for computing a large number of sensitivities for one output, as the cost is one residual perturbation calculation and one vector product per sensitivity.

The central idea in the adjoint approach is that we do not need to solve the forward problem each time we want a sensitivity. Suppose that for a given $\boldsymbol{\mu}$ we solve our discrete system and find $\mathbf{U}$ such that $\mathbf{R}(\mathbf{U}, \boldsymbol{\mu}) = \mathbf{0}$. Now perturb $\boldsymbol{\mu} \to \boldsymbol{\mu} + \delta\boldsymbol{\mu}$ ... what is the effect on J? We can re-solve for a perturbed state, but this is expensive. Instead, we can separate the effects of $\boldsymbol{\mu}$ on $\mathbf{R}$, and $\mathbf{R}$ on J, as illustrated in Figure 5.1.2. The adjoint method precomputes the effect of $\mathbf{R}$ on J, which is the expensive step. The resulting N sensitivities are stored in the vector $\boldsymbol{\Psi}$.

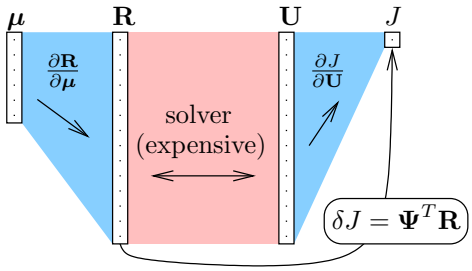


Figure 5.1.2: Bypassing the forward solve via an adjoint approach to sensitivity calculation.

5.1.2 The Adjoint System

To derive an equation for the adjoint, we consider the chain of operations we would take in computing the sensitivities via a direct approach. In the following steps, we assume small perturbations.

1. Input $\quad \boldsymbol{\mu} \to \boldsymbol{\mu} + \delta\boldsymbol{\mu}$
2. Residual $\quad \mathbf{R}(\mathbf{U}, \boldsymbol{\mu} + \delta\boldsymbol{\mu}) = \delta\mathbf{R} \neq 0 \quad \to \quad \mathbf{R}(\mathbf{U}, \boldsymbol{\mu}) + \frac{\partial \mathbf{R}}{\partial \boldsymbol{\mu}}\Big|_{\mathbf{U},\boldsymbol{\mu}} \delta\boldsymbol{\mu} = \delta\mathbf{R}$
3. State $\quad \mathbf{R}(\mathbf{U} + \delta\mathbf{U}, \boldsymbol{\mu} + \delta\boldsymbol{\mu}) = 0 \quad \to \quad \mathbf{R}(\mathbf{U}, \boldsymbol{\mu}) + \frac{\partial \mathbf{R}}{\partial \boldsymbol{\mu}}\Big|_{\mathbf{U},\boldsymbol{\mu}} \delta\boldsymbol{\mu} + \frac{\partial \mathbf{R}}{\partial \mathbf{U}}\Big|_{\mathbf{U},\boldsymbol{\mu}} \delta\mathbf{U} = 0$
4. Output $\quad J(\mathbf{U} + \delta\mathbf{U}) = J(\mathbf{U}) + \delta J \quad \to \quad \delta J = \frac{\partial J}{\partial \mathbf{U}} \delta\mathbf{U}$

Subtracting step 2 from step 3, we obtain

$$\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\Big|_{\mathbf{U},\boldsymbol{\mu}} \delta\mathbf{U} = -\delta\mathbf{R} \quad \Rightarrow \quad \delta\mathbf{U} = -\left[\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right]^{-1} \delta\mathbf{R}. \tag{5.1.6}$$

Combining this result with the output linearization in step 4 gives the output perturbation in terms of the residual perturbation,

$$\delta J = \frac{\partial J}{\partial \mathbf{U}} \delta\mathbf{U} = \underbrace{-\frac{\partial J}{\partial \mathbf{U}} \left[\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right]^{-1}}_{\boldsymbol{\Psi}^T \in \mathbb{R}^N} \delta\mathbf{R}. \tag{5.1.7}$$

Taking the transpose of the equation $\boldsymbol{\Psi}^T = -\frac{\partial J}{\partial \mathbf{U}} \left[\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right]^{-1}$ and moving everything to the left-hand side gives the *adjoint equation*,

$$\left(\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right)^T \boldsymbol{\Psi} + \left(\frac{\partial J}{\partial \mathbf{U}}\right)^T = 0. \tag{5.1.8}$$

The n^{th} component of $\boldsymbol{\Psi}$ is the sensitivity of J to changes in the n^{th} residual.

Since $\mathbf{R}(\mathbf{U}, \boldsymbol{\mu}) = 0$, from step 2 above we have $\delta\mathbf{R} = \frac{\partial \mathbf{R}}{\partial \boldsymbol{\mu}}\Big|_{\mathbf{U},\boldsymbol{\mu}} \delta\boldsymbol{\mu}$ and Equation 5.1.7 becomes

$$\delta J = \boldsymbol{\Psi}^T \frac{\partial \mathbf{R}}{\partial \boldsymbol{\mu}}\Big|_{\mathbf{U},\boldsymbol{\mu}} \delta\boldsymbol{\mu} \quad \Rightarrow \quad \frac{dJ}{d\boldsymbol{\mu}} = \boldsymbol{\Psi}^T \frac{\partial \mathbf{R}}{\partial \boldsymbol{\mu}}\Big|_{\mathbf{U},\boldsymbol{\mu}}. \tag{5.1.9}$$

Therefore, once we have $\boldsymbol{\Psi}$, no more solutions are required for new sensitivities for the same output. Note that the calculation of $\frac{\partial \mathbf{R}}{\partial \boldsymbol{\mu}}$ is typically very cheap compared to a forward solve.

Although we have presented the adjoint in the context of a parameter sensitivity analysis, we will see in the next section that residual perturbations also arise when discrete solutions are viewed from an enriched space. This will be the motivation for using adjoint solutions in output error estimation.

5.1.3 Adjoint Consistency

The solution to Equation 5.1.8 is a *discrete* adjoint, $\boldsymbol{\Psi}$, which at the simplest level we can think of as a vector of N numbers. However, the adjoint also has a continuous counterpart, call it $\boldsymbol{\psi}(\vec{x})$, and we can think of the N numbers in $\boldsymbol{\Psi}$ as expansion coefficients in an approximation of $\boldsymbol{\psi}$ using the same basis functions used for the primal problem. The accuracy of this approximation is of interest for various reasons, including error estimation.

Suppose that the exact primal solution, $\mathbf{u} \in \boldsymbol{\mathcal{V}}$, satisfies

$$\mathcal{R}(\mathbf{u}, \mathbf{v}) = 0, \qquad \forall \mathbf{v} \in \boldsymbol{\mathcal{V}}, \tag{5.1.10}$$

for an appropriately defined space $\boldsymbol{\mathcal{V}}$. The exact adjoint $\boldsymbol{\psi} \in \boldsymbol{\mathcal{V}}$ then satisfies

$$\mathcal{R}'[\mathbf{u}](\mathbf{v}, \boldsymbol{\psi}) + \mathcal{J}'[\mathbf{u}](\mathbf{v}) = 0, \qquad \forall \mathbf{v} \in \boldsymbol{\mathcal{V}}, \tag{5.1.11}$$

where the primes denote Fréchét linearization about the arguments in square brackets, and $\mathcal{R}$ and $\mathcal{J}$ are the continuous versions of the semilinear form and output functional, respectively. While we have assumed that both $\mathbf{u}$ and $\boldsymbol{\psi}$ lie in $\boldsymbol{\mathcal{V}}$, this may not always be the case [91].

The exact adjoint can be regarded as a Green's function that relates source perturbations in the original partial differential equation to perturbations in the output[58, 60], as illustrated schematically in Figure 5.1.3. A sample adjoint solution is shown in Figure 5.1.4 for Reynolds-averaged compressible flow over an airfoil. While the adjoint solution often shares qualitative characteristics similar to the primal, such as the presence of a boundary layer in a high-Reynolds number flow, it also shows marked differences, such as the "wake reversal" seen in the far-field view in Figure 5.1.4. In this case, the output is drag, and upstream of the airfoil, residual perturbations on/near the stagnation streamline (the flow that is going to hit or come closest to the airfoil) will have a larger magnitude impact on the drag than residual perturbations elsewhere; hence we see an adjoint "reversed" wake telling us that there are large sensitivities to perturbations in front of the airfoil. The figure shown tells us that this is the case for residual perturbations in the conservation of x-momentum equations, but plots of the other adjoint components show similar behavior.

Figure 5.1.5 illustrates an adjoint field for another case: supersonic flow over a diamond airfoil. The output in this case is a pressure line integral, and nonzero values of the adjoint (y-momentum component shown) indicate where perturbations to the (y-momentum) residuals will affect the output.

The adjoint field depicted in Figure 5.1.4 is the discrete adjoint solution on a fine mesh. It can only be regarded as a faithful representation of the exact adjoint if the discretization is in some manner consistent with the exact adjoint problem. *Primal consistency* in

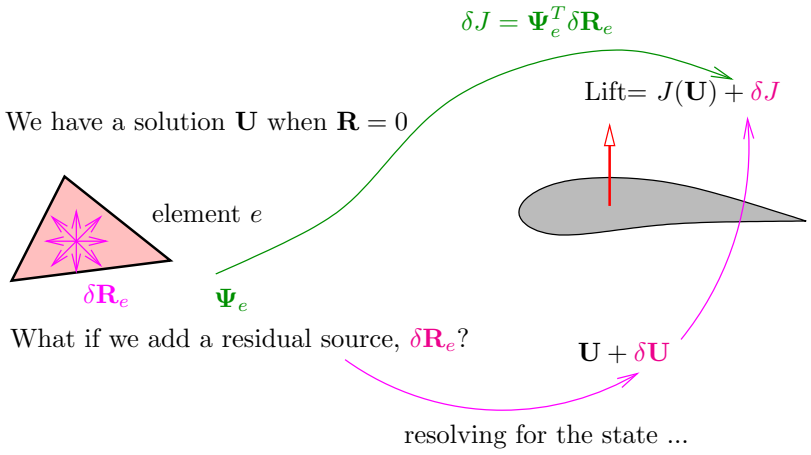
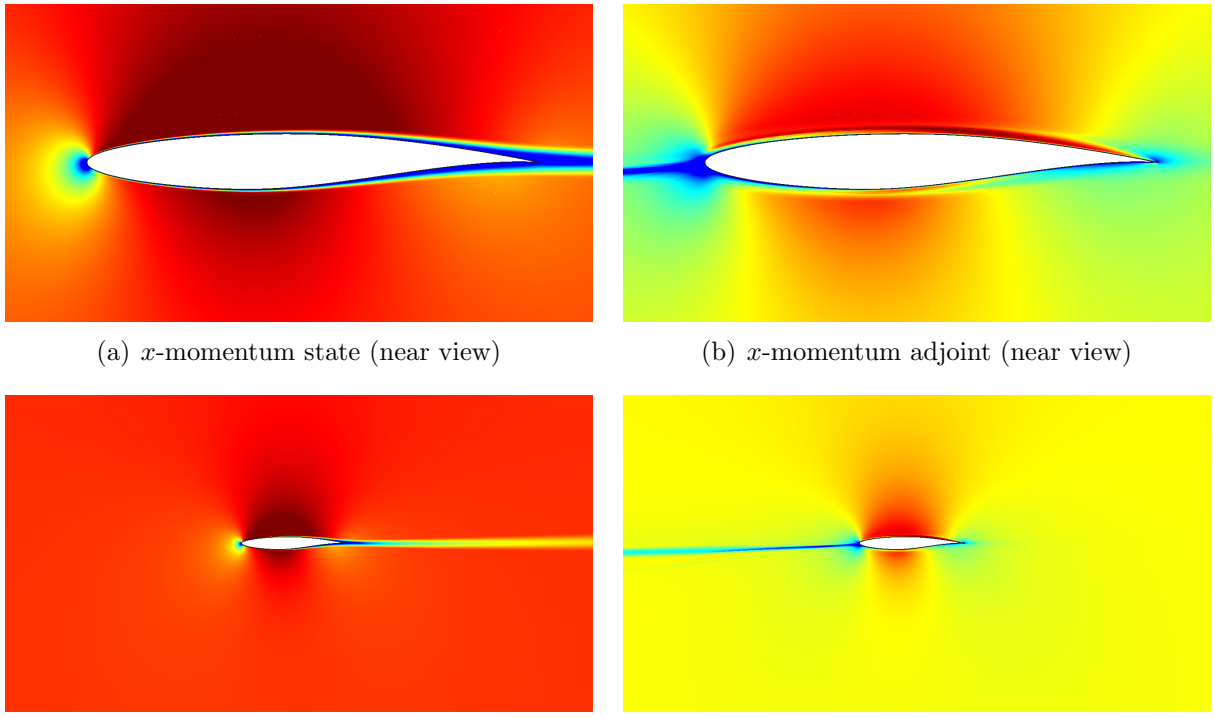


Figure 5.1.3: Interpretation of the adjoint as the sensitivity of a scalar output, such as lift, to residual source perturbations.



(a) x-momentum state (near view)

(b) x-momentum adjoint (near view)

(c) x-momentum state (far view)

(d) x-momentum adjoint (far view)

Figure 5.1.4: Comparison of the primal solution (x-momentum component) and the adjoint solution (conservation of x-momentum equation component) for a drag output in Reynolds-averaged turbulent flow over an RAE 2822 airfoil. The color scales are clipped to show the interesting features of each quantity – in the adjoint plots, yellow is near zero.

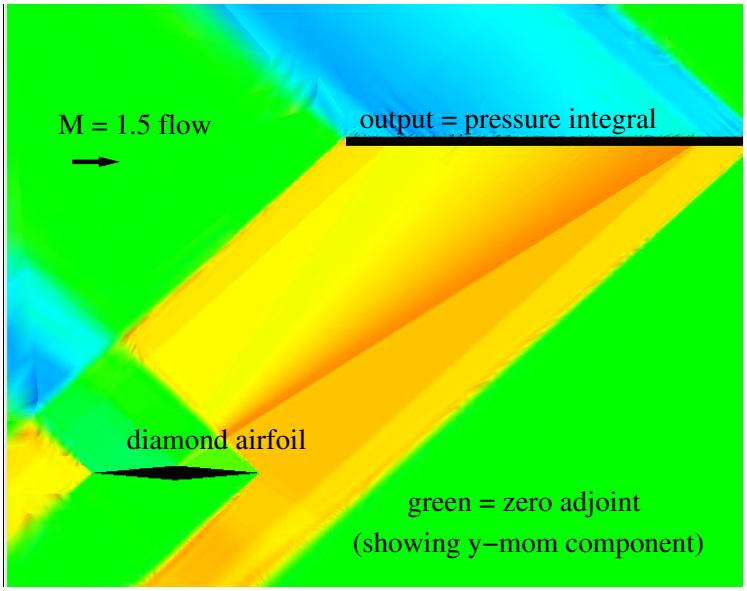


Figure 5.1.5: Sample adjoint solution for a pressure integral output in supersonic flow over a diamond airfoil. Large regions of zero adjoint indicate areas where residuals have no affect on the output – these could also be identified via a characteristic analysis.

the variational problem requires that the exact solution $\mathbf{u}$ satisfy the discrete variational statement,

$$\mathcal{R}_h(\mathbf{u}, \mathbf{v}) = 0, \qquad \forall \mathbf{v} \in \boldsymbol{\mathcal{W}}_h, \tag{5.1.12}$$

where $\boldsymbol{\mathcal{W}}_h = \boldsymbol{\mathcal{V}}_h + \boldsymbol{\mathcal{V}} = \{\mathbf{h} = \mathbf{f} + \mathbf{g} : \mathbf{f} \in \boldsymbol{\mathcal{V}}_h, \mathbf{g} \in \boldsymbol{\mathcal{V}}\}$. Similarly, the combination of the discrete semi-linear form $\mathcal{R}_h$ and the functional $\mathcal{J}_h$ is said to be *adjoint consistent* if [91, 67, 109]

$$\mathcal{R}'_h[\mathbf{u}](\mathbf{v}, \boldsymbol{\psi}) + \mathcal{J}'_h[\mathbf{u}](\mathbf{v}) = 0, \qquad \forall \mathbf{v} \in \boldsymbol{\mathcal{W}}_h. \tag{5.1.13}$$

Discretizations that are not adjoint consistent may still be *asymptotically adjoint consistent* if Eq. 5.1.13 holds in the limit $h \to 0$, by which we mean the limit of uniformly increasing resolution, over suitably normalized $\mathbf{v} \in \boldsymbol{\mathcal{W}}_h$. For non-variational discretizations, the definition of consistency must involve an approximation operator to map exact solutions into discrete spaces [37].

Adjoint consistency has an impact on the convergence of not only the adjoint approximation but also the primal approximation[6, 65, 59, 64, 91, 67, 109]. In error estimation, an adjoint-inconsistent discretization can lead to irregular or oscillatory adjoint solutions that pollute the error estimate with noise and lead to adaptation in incorrect areas [91]. Enforcing adjoint consistency imposes restrictions on the output definition and on the interior and boundary discretizations that enter into the semi-linear form. These restrictions have been studied by several authors in the context of the discontinuous Galerkin method [6, 91, 67]. In

general, discretizations that are found to be adjoint inconsistent can often be made adjoint consistent by adding terms to either the semi-linear form or the output functional.

5.1.4 Adjoint Sensitivity Tests

One way to test a discrete adjoint is to compare output sensitivities computed using the adjoint to those computed using finite differences. In this example we show this comparison for a viscous flow over a NACA 0012 airfoil at $M = 0.5$, $Re = 5000$, $\alpha = 2°$. We are interested in the sensitivity of the lift coefficient to angle of attack. Since the lift is defined as the force perpendicular to the free-stream direction, the output depends directly on the input parameters, so that we need to augment Equation 5.1.5 with the partial derivative of J with respect to the input angle of attack, α,

$$\frac{dJ}{d\alpha} = \boldsymbol{\Psi}^T \frac{\partial \mathbf{R}}{\partial \alpha} + \frac{\partial J}{\partial \alpha}. \tag{5.1.14}$$

For the present test we compute $\frac{\partial \mathbf{R}}{\partial \alpha}$ using a simple forward difference,

$$\frac{\partial \mathbf{R}}{\partial \alpha} \approx \frac{\mathbf{R}(\mathbf{U}, \alpha + \Delta\alpha) - \mathbf{R}(\mathbf{U}, \alpha)}{\Delta\alpha}, \tag{5.1.15}$$

where $\Delta\alpha$ is a small value, e.g. 0.01rad. In Figure 5.1.6, we show a comparison of the sensitivity computed using Equation 5.1.14, i.e. a local linearized sensitivity, to outputs obtained by actually perturbing the input parameter. The qualitative agreement is excellent, and a quantitative look shows convergence at the expected rate as the angle of attack perturbation goes to zero.

5.1.5 Extension to Unsteady Systems

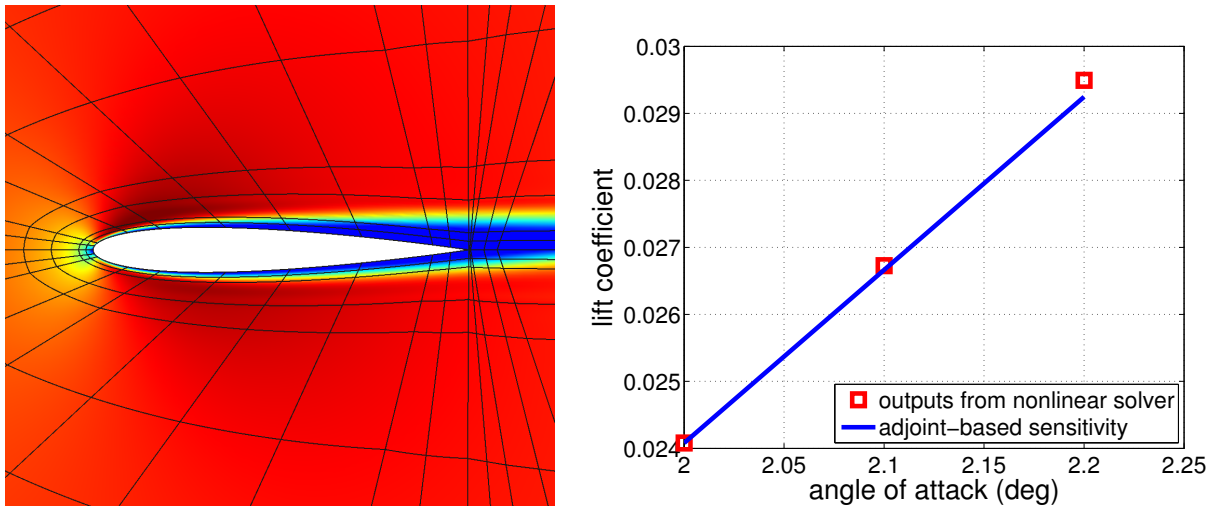
Let's look at the adjoint for an unsteady multi-step discretization. The unsteady version of the sensitivity chain in Equation 5.1.3 is

$$\boldsymbol{\mu} \rightarrow \mathbf{R}^m(\mathbf{U}^n, \boldsymbol{\mu}) = \mathbf{0} \rightarrow \mathbf{U}^n \rightarrow J(\mathbf{U}^n), \tag{5.1.16}$$

where, as in the primal form, $0 \leq n, m \leq N_t$ are the time indices in the unsteady discretization. The equations derived for the steady adjoint apply directly, but the vectors are now much larger (length $\mathbb{R}^{NN_t}$). The adjoint equation is

$$\sum_{m=1}^{N_t} \left(\frac{\partial \mathbf{R}^m}{\partial \mathbf{U}^n} \right)^T \boldsymbol{\Psi}^m + \left(\frac{\partial J}{\partial \mathbf{U}^n} \right)^T = \mathbf{0}. \tag{5.1.17}$$

Due to the transpose on the unsteady Jacobian matrix, the adjoint system is most easily solved by marching backwards in time. For example, given a backward Euler temporal discretization, the unsteady Jacobian $\frac{\partial \mathbf{R}^m}{\partial \mathbf{U}^n}$ and its transpose will look like ($\star = N \times N$ block),



(a) Mach number contours

(b) lift coefficient sensitivity

Figure 5.1.6: Verification of the discrete adjoint solver using a parameter sensitivity test. The angle of attack is varied in a viscous flow over a NACA 0012 airfoil, and the resulting data points are overlaid on a line through the baseline, $\alpha = 2°$, case, with a slope computed from the discrete lift coefficient adjoint. The agreement is excellent for small α perturbations; deviations at larger α are due to the nonlinear nature of the compressible Navier-Stokes equations.

Primal unsteady Jacobian

$$\frac{\partial \mathbf{R}^m}{\partial \mathbf{U}^n} = \begin{bmatrix} \star & & & & & & \\ \star & \star & & & & & \\ & \star & \star & & & & \\ & & \star & \star & & & \\ & & & \star & \star & & \\ & & & & \star & \star & \\ & & & & & \star & \star \end{bmatrix}$$

Adjoint unsteady Jacobian

$$\left(\frac{\partial \mathbf{R}^m}{\partial \mathbf{U}^n}\right)^T = \begin{bmatrix} \star & \star & & & & & \\ & \star & \star & & & & \\ & & \star & \star & & & \\ & & & \star & \star & & \\ & & & & \star & \star & \\ & & & & & \star & \star \\ & & & & & & \star \end{bmatrix}$$

We see that the primal Jacobian is zero above the main diagonal, making forward substitution in time the method of choice for solving the primal problem. On the other hand, the adjoint Jacobian is zero below the main diagonal, so that backward substitution is the natural solution strategy. Hence, an unsteady adjoint solution requires marching *backward* in time from the final state to the first state.

For nonlinear problems, the state history $\mathbf{U}^n$ needs to be stored for computing the linearizations in the adjoint solve. These states are typically written to disk, and solution checkpointing can alleviate storage requirements if they are burdensome [61]. Once the

unsteady adjoint solution is available, the sensitivities are calculated as

$$\frac{dJ}{d\boldsymbol{\mu}} = \sum_{m=1}^{N_t} (\boldsymbol{\Psi}^m)^T \left(\frac{\partial \mathbf{R}^m}{\partial \boldsymbol{\mu}} \right). \tag{5.1.18}$$

5.2 Error Estimation

Numerical errors due to insufficient mesh resolution can affect outputs, often in seemingly subtle but significant ways. The latter point is especially true for the convection-dominated flows common to aerodynamics. In this section, we present a method for estimating this error using adjoints.

5.2.1 Two Discretization Levels

Without access to infinite resolution, estimating the true numerical error in an output is practically out of reach for general nonlinear problems. We thus resign ourselves to estimating the output error between two finite-dimensional spaces: a coarse approximation space ($\boldsymbol{\mathcal{V}}_H$) on which we calculate the state and output, and a fine space ($\boldsymbol{\mathcal{V}}_h$) relative to which we estimate the error. The equations and output representations on these spaces are

$$\begin{aligned}
\text{coarse space:} \quad &\rightarrow \underbrace{\mathbf{R}_H(\mathbf{U}_H) = 0}_{N_H \text{ equations}} \rightarrow \underbrace{\mathbf{U}_H}_{\text{state} \in \mathbb{R}^{N_H}} \rightarrow \underbrace{J_H(\mathbf{U}_H)}_{\text{output (scalar)}} \\
\text{fine space:} \quad &\rightarrow \underbrace{\mathbf{R}_h(\mathbf{U}_h) = 0}_{N_h \text{ equations}} \rightarrow \underbrace{\mathbf{U}_h}_{\text{state} \in \mathbb{R}^{N_h}} \rightarrow \underbrace{J_h(\mathbf{U}_h)}_{\text{output (scalar)}}
\end{aligned}$$

We would like to measure the output error in the coarse solution relative to the fine space,

$$\text{output error:} \quad \delta J \equiv J_H(\mathbf{U}_H) - J_h(\mathbf{U}_h). \tag{5.2.1}$$

The fine space is typically constructed by uniformly refining each element in the coarse space, or by increasing each element's approximation order. Figure 5.2.1 illustrates a fine space obtained by uniform refinement. We assume that the fine approximation space contains the coarse approximation space, so that the following lossless state injection, $\mathbf{U}_h^H$, is possible:

$$\mathbf{U}_h^H \equiv \mathbf{I}_h^H \mathbf{U}_H, \tag{5.2.2}$$

where $\mathbf{I}_h^H$ is the coarse-to-fine state injection (prolongation) operator.

5.2.2 The Adjoint-Weighted Residual

On the fine space, the exact solution $\mathbf{U}_h \in \mathbb{R}^{N_h}$ would give us zero fine-space residuals,

$$\mathbf{R}_h(\mathbf{U}_h) = \mathbf{0}. \tag{5.2.3}$$

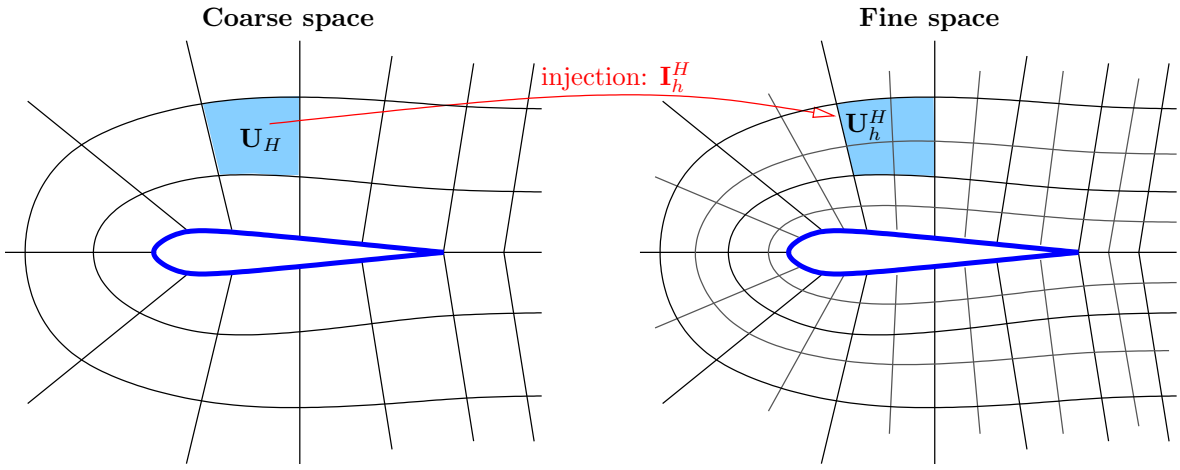


Figure 5.2.1: Sample fine approximation space obtained by uniform refinement of each element in the mesh of the coarse approximation space. An alternative fine space is obtained by incrementing the approximation order of each element.

However, the state injected from the coarse space will generally not be a fine space solution and hence will not give us zero fine-space residuals,

$$\mathbf{R}_h(\mathbf{U}_h^H) \neq \mathbf{0}. \tag{5.2.4}$$

Instead, the injected coarse state solves a *perturbed* fine-space problem,

$$\text{find } \mathbf{U}'_h \text{ such that:} \quad \mathbf{R}_h(\mathbf{U}'_h) \underbrace{-\mathbf{R}_h(\mathbf{U}_h^H)}_{\delta\mathbf{R}_h} = \mathbf{0} \quad \Rightarrow \quad \text{answer is: } \mathbf{U}'_h = \mathbf{U}_h^H. \tag{5.2.5}$$

As this is just the fine-space problem with a residual perturbation, the fine-space adjoint, $\mathbf{\Psi}_h$, tells us to expect an output perturbation given by the inner product between the adjoint and the residual perturbation,

$$\underbrace{J_h(\mathbf{U}_h^H) - J_h(\mathbf{U}_h)}_{\approx \delta J} = \mathbf{\Psi}_h^T \delta\mathbf{R}_h = -\mathbf{\Psi}_h^T \mathbf{R}_h(\mathbf{U}_h^H). \tag{5.2.6}$$

This derivation assumes small perturbations in $\mathbf{U}$ and $\mathbf{R}$ when the output or equations are nonlinear. Calling the left-hand side δJ assumes $J_H(\mathbf{U}_H) = J_h(\mathbf{U}_h^H)$, which is true if the output definition (e.g. geometry) does not change between the coarse and fine spaces.

In summary, we have

$$\boxed{\delta J \approx -\mathbf{\Psi}_h^T \mathbf{R}_h(\mathbf{U}_h^H)} \tag{5.2.7}$$

Note that this error estimate does not require the fine-space primal solution, $\mathbf{U}_h$.

5.2.3 Approximations

The error estimate in Equation 5.2.7 uses the adjoint on the fine space, $\mathbf{\Psi}_h$. Obtaining $\mathbf{\Psi}_h$ requires solving a large, possibly expensive, linear system. Suppose we have the *coarse*-space adjoint, $\mathbf{\Psi}_H$, which injected into the fine space is $\mathbf{\Psi}_h^H \equiv \mathbf{I}_h^H \mathbf{\Psi}_H$. Define the adjoint perturbation as

$$\delta\mathbf{\Psi}_h \equiv \mathbf{\Psi}_h^H - \mathbf{\Psi}_h.$$

We then re-write Equation 5.2.7 as

$$\delta J \approx \underbrace{-\left(\mathbf{\Psi}_h^H\right)^T \mathbf{R}_h(\mathbf{U}_h^H)}_{\text{computable correction}} + \underbrace{(\delta\mathbf{\Psi}_h)^T \mathbf{R}_h(\mathbf{U}_h^H)}_{\text{remaining error}}. \tag{5.2.8}$$

The computable correction is tempting to use as the sole error estimate. It does provide important information on δJ for many discretizations, including reconstructed finite volume. However, it performs poorly as an adaptive indicator because it does not incorporate any new information from the fine space. Moreover, it is zero for finite element discretizations with Galerkin orthogonality. Therefore we need some estimate of $\mathbf{\Psi}_h$. Several methods are used in practice,

1. Reconstruct $\mathbf{\Psi}_h$ from $\mathbf{\Psi}_H$ using information from neighboring elements.
2. Solve for $\mathbf{\Psi}_h$ approximately using a cheap iterative smoother on the fine space.
3. Solve for $\mathbf{\Psi}_h$ exactly on the fine space if the $N_h \times N_h$ linear system is tractable.

The third option gives the most accurate error estimates (albeit at the highest cost). When the fine-space solve becomes too expensive, one can turn to one of the first two approximations; at a cost of losing some accuracy in the error estimate. In all options, the adaptive indicator obtained from the error estimate remains similar.

For nonlinear problems the leading term not present in Equation 5.2.8 is quadratic in the state and adjoint errors. This "error in the error estimate" can be reduced to third-order in the state and adjoint errors by using [123]

$$\delta J \approx -\left(\mathbf{\Psi}_h^H\right)^T \mathbf{R}_h(\mathbf{U}_h^H) + \frac{1}{2}(\delta\mathbf{\Psi}_h)^T \mathbf{R}_h(\mathbf{U}_h^H) + \frac{1}{2}(\delta\mathbf{U}_h)^T \mathbf{R}_h^{\psi}(\mathbf{\Psi}_h^H), \tag{5.2.9}$$

where $\mathbf{R}_h^{\psi}(\mathbf{\Psi}_h^H)$ is the residual vector of the fine-space adjoint problem, Equation 5.1.8, and $\delta\mathbf{U}_h = \mathbf{U}_h^H - \mathbf{U}_h$ is the state perturbation. In practice, both the state and adjoint perturbations could be approximated using one of the three approaches outlined above.

5.2.4 Error Effectivity

The error estimate calculated above is not a bound. We measure its accuracy by defining an *effectivity*,

$$\eta_H = \frac{J_H(\mathbf{U}_H) - J_h(\mathbf{U}_h)}{J_H(\mathbf{U}_H) - J}, \tag{5.2.10}$$

where J is the exact output, i.e. calculated from the exact solution. An effectivity close to 1 is desirable. In practice this value will depend on the choice of fine space (order enrichment versus element subdivision), and on approximations made in the fine-space adjoint error estimation. However, we can make some rough a priori estimates. If the output converges as $J_H(\mathbf{U}_H) - J = CH^k$, uniform element subdivision for the fine space yields an effectivity of $\eta_H = 1 - (1/2)^k$ – this does not approach unity as $H \to 0$. On the other hand, if order enrichment is used for the fine space, then the effectivity converges as $\eta_H = 1 - C_1 H^{\delta k}$, where δk is the increase in convergence rate of the fine-space output relative to the coarse-space output. In this case, the effectivity does approach unity with mesh refinement [47].

5.2.5 Examples

We now present some examples that test the error estimates described in this section. We use uniform mesh refinement studies to determine the rate, k, at which certain errors, notably the output error, asymptotically converge,

$$\text{error} \propto h^k, \quad \text{valid as } h \to 0, \tag{5.2.11}$$

where h is a measure of the size of the mesh elements. Even though we have many elements, not all of the same size, we use a single h value that we can think of as indicating a refinement level. When we do a uniform refinement study, we are not concerned with the particular value of h; we just want to know how "uniform" changes in h for every element translate into changes in the error.

In the following studies we will uniformly subdivide mesh elements, by bisecting each edge [1], and measure the resulting effect on the error. In this case, a reasonable definition of h is $\sqrt{1/N_e}$, where N_e is the number of elements. So a uniform refinement of a two-dimensional mesh would increase N_e by a factor of 4, and decrease h by a factor of 2. Taking the logarithm of Equation 5.2.11, we obtain

$$\log(\text{error}) \quad = \quad C + k \log\left(\sqrt{\frac{1}{N_e}}\right). \tag{5.2.12}$$

Drag Error for Euler Flow over a Bump

In this example, we consider inviscid subsonic flow inside a channel that has a smooth Gaussian perturbation on the bottom wall. The compressible Euler equations govern the flow, and the output J is the drag (horizontal) force on the bottom wall. The solution on a fine mesh is shown in Figure 5.2(a).

Figure 5.2(b) shows the initial coarse mesh for the uniform refinement study. On this mesh, we perform the following steps:

[1] When using curved elements it is important to bisect curved edges as close to along the arc length as possible, and the same applies for element interiors; for example, bisecting a reference element when using a reference-to-global mapping with a lot of nonlinear stretching would not necessarily reduce the size of each element by the same amount.

1. Solve for the flow using $p = 1$. This gives us $\mathbf{U}_H$, which is the state in the coarse approximation space. Also compute $J_H = J_H(\mathbf{U}_H)$.

2. Solve the adjoint exactly using $p = 2$. This gives us $\mathbf{\Psi}_h$, which is in the fine approximation space.

3. Compute the error estimate using Equation 5.2.7 and find the associated corrected output,

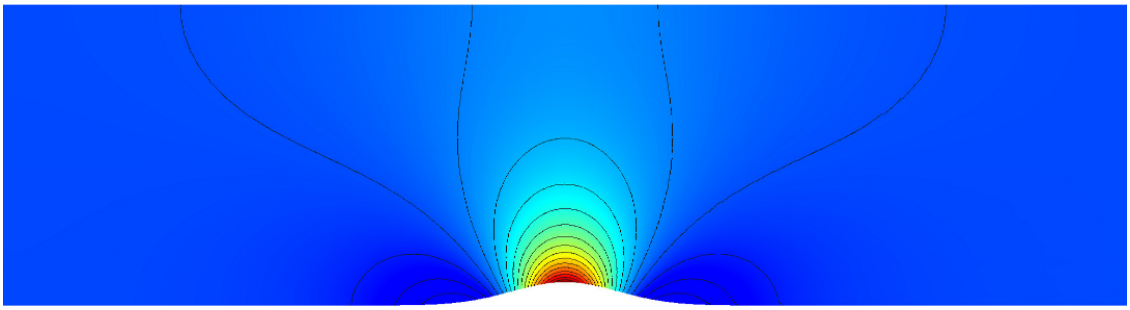
$$\text{corrected output} = J_H - \delta J. \tag{5.2.13}$$

We then repeat these steps on three successive refinements of the coarse mesh. We also obtain the "exact" output, J, by solving using $p = 3$ approximation on a mesh that is uniformly refined once more compared to the finest mesh in the study.

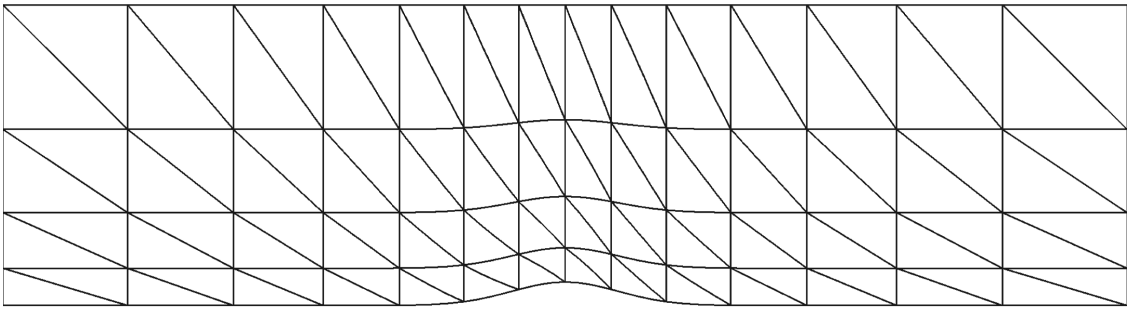
Figure 5.2(c) shows the convergence of the errors in the coarse-space output, $J_H - J$, and in the corrected output, $J_H - \delta J - J$. Based on the formula in Equation 5.2.12, we expect (asymptotically) straight lines on a log-log plot, and this is what we see. The slopes tell us the convergence rates. First, we observe a rate of $k = 3$ for the output error $J_H - J$, and for our $p = 1$ approximation, this is a super-convergent result ... approximation theory would lead us to expect a rate of $p+1$ but we are actually seeing a rate of $2p+1$ (we can verify this with data from higher p). Second, we observe a rate of 4, $2p + 2$, for the corrected output. So we see that the output correction buys us an extra order of convergence for the output, which is a reasonable result as we are using order enrichment for the fine space.

Figure 5.2(d) plots the error effectivity, Equation 5.2.10, for the solutions on the 4 mesh refinements. We actually plot two effectivities: one as defined in Equation 5.2.10 (this is relative to the exact output J), and a similar one but relative to the fine-space output J_h. The latter choice gives us an idea of the errors we make in estimating $J_H - J_h$, while the former tells us about $J_H - J$. We see that as expected (by design), we estimate the error better relative to the fine space than relative to the true output, but that both effectivities approach 1 as $H \to 0$.

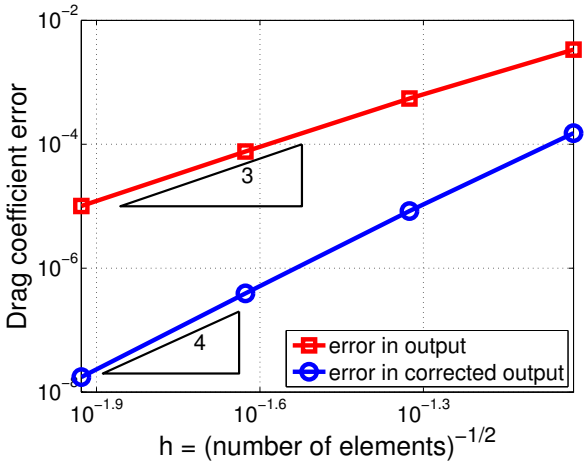
We mentioned that in practice we sometimes solve the fine-space adjoint problem approximately, to avoid the cost of a full solve on the fine space. It turns out that this approximation degrades the accuracy of the error estimates, resulting in worse convergence rates for the corrected output compared to when using an exact fine space adjoint. Figure 5.2.3 shows this degradation when using ν iterations of an element-block Jacobi solver to smooth the fine-space adjoint after injection from the coarse space. We see that when ν is small (less work on the fine space), the error estimates are not great, although the corrected output is still better than the original coarse one. As ν increases, the error estimates approach the exact adjoint solve result ($\nu \to \infty$), and the effectivities close in on 1. The "adequate" number of smoothing iterations will depend on the case and on the smoother, in addition to the desired accuracy levels, so these results should not be interpreted as a recipe for how to choose ν. Instead, one take-away message is that to get the best error estimates possible, we need the best possible fine-space adjoint; however, another one is that even approximate fine-space adjoints can lead to improved outputs. Furthermore, in an adaptive setting, the



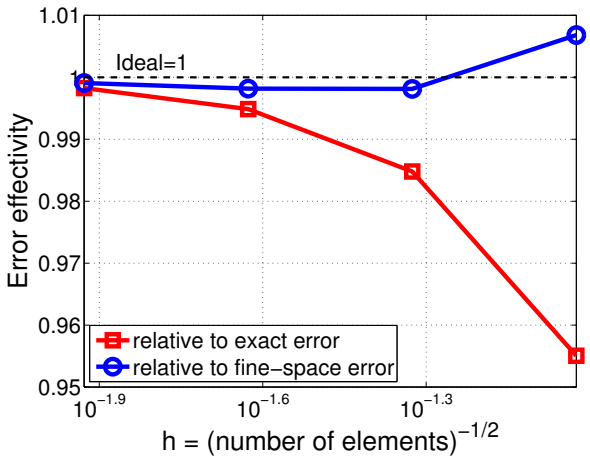
(a) Mach number contours; range is 0.27 to 0.43



(b) Coarsest mesh for uniform refinement study: quartic curved triangles

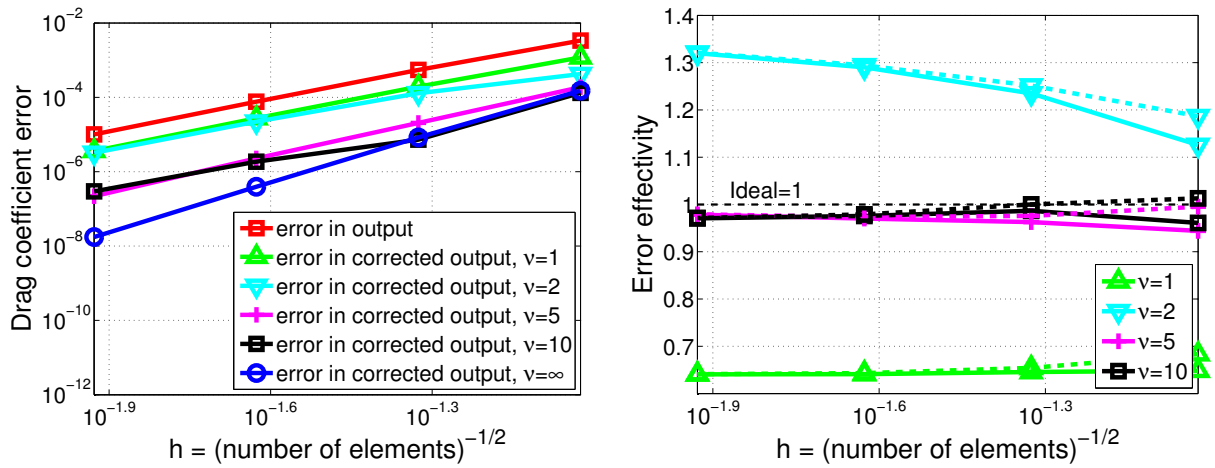


(c) Convergence of baseline and corrected output



(d) Convergence of error effectivity

Figure 5.2.2: Error estimates and effectivity for inviscid flow in a channel with a Gaussian bump, at $M_\infty = 0.3$, using $p = 1$ approximation. The output of interest is the drag force coefficient on the bottom wall. The adjoint is solved exactly on the fine space.



(a) Convergence of baseline and corrected outputs

(b) Convergence of error effectivities

Figure 5.2.3: Effect of inexact fine-space adjoint solve, using ν element-block smoothing iterations, on the error estimates for inviscid flow in a channel with a Gaussian bump, at $M_\infty = 0.3$, using $p = 1$ approximation. The output of interest is the drag force coefficient on the bottom wall.

indicators obtained from these error estimates will turn out to be very good at driving mesh refinement.

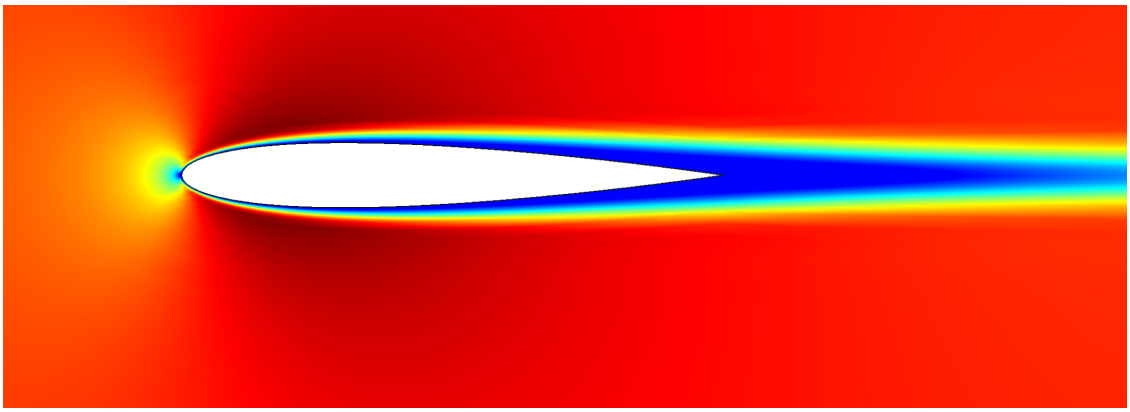
Drag Error for Viscous Flow over a NACA 0012 Airfoil

In this example, we consider viscous subsonic ($M_\infty = 0.5$) flow over a NACA 0012 airfoil (closed-trailing-edge [42]) at zero angle of attack. The compressible Navier-Stokes equations, at $Re = 5000$, $Pr = 0.71$, and constant viscosity, govern the flow, and the output J is the drag (horizontal) force on the airfoil, at which an adiabatic no-slip boundary condition is imposed. The solution on a fine mesh is shown in Figure 5.4(a).

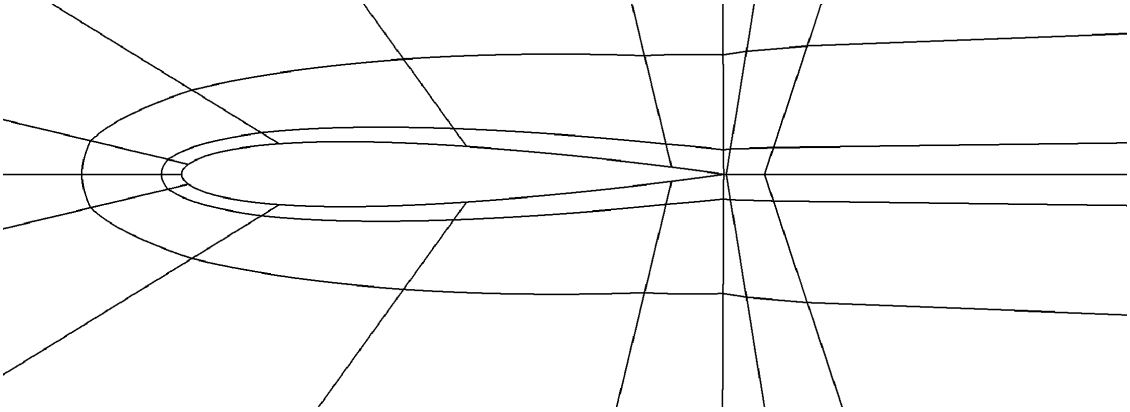
We follow the same steps as in the previous example for measuring the convergence rate of the output and the corrected output (i.e. the error estimate)[2]. Figure 5.4(c) shows the convergence of the errors in the coarse-space output, $J_H - J$, and in the corrected output, $J_H - \delta J - J$. The lines are not straight initially, indicating that the solution is not yet in the asymptotic regime. However, by the finer meshes we observe relatively straight lines on the log-log plot. We observe a rate of $k \approx 2.7$ for the output error $J_H - J$, and for our $p = 1$ approximation, this is still super-convergent relative to the expected rate of $p + 1$. For the corrected output, we observe a higher rate of 3.8, and so we see that the output correction buys us an extra order of convergence, which is again reasonable as we are using order enrichment for the fine space.

The effectivity story in Figure 5.2(d) is similar to that of the previous example. The

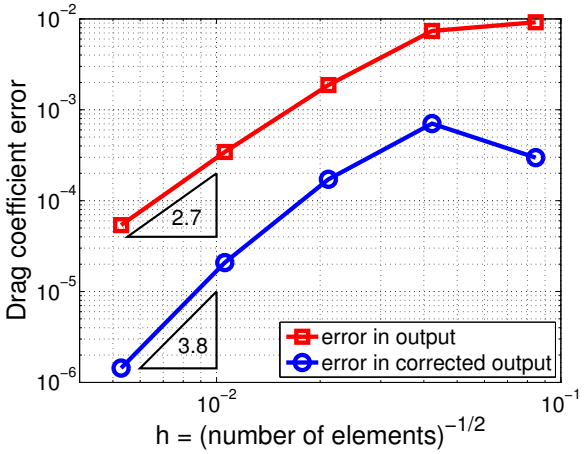
[2]We also follow the remedy for alleviating the effects of p-dependence of the BR2 residual on the error estimates, by evaluating the fine-space residual with order p integration rules and stabilization approximation. [162]



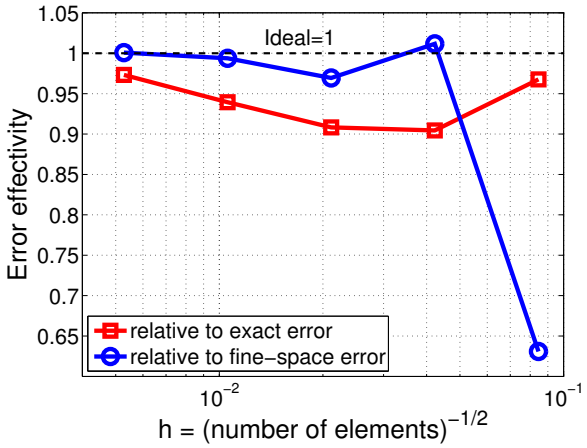
(a) Mach number contours; range is 0.0 to 0.6



(b) Coarsest mesh for uniform refinement study: quartic curved quadrilaterals



(c) Convergence of outputs



(d) Convergence of error effectivity

Figure 5.2.4: Error estimates and effectivity for viscous $Re = 5000$, $M_\infty = 0.5$ flow over a NACA 0012 airfoil at $\alpha = 0$, using $p = 1$ approximation. The output of interest is the drag force coefficient on the airfoil, and the adjoint is solved exactly on the fine space.

error estimate does a better job at predicting the error between the coarse space and the fine space than at predicting the error relative to the exact output, but both of the effectivities converge to 1 as $H \to 0$.

5.3 Mesh Adaptation

The output error estimate derived in the previous section provides error bars on quantities of interest from numerical simulations. However, the benefit of these estimates extends beyond the error bars. Because the output error estimate takes the form of a weighted residual, and because local mesh refinement decreases residuals, the error estimate provides a means of targeting for refinement areas of the computational domain that give rise to the output error. This is the key idea of output-based mesh adaptation.

5.3.1 Error Localization

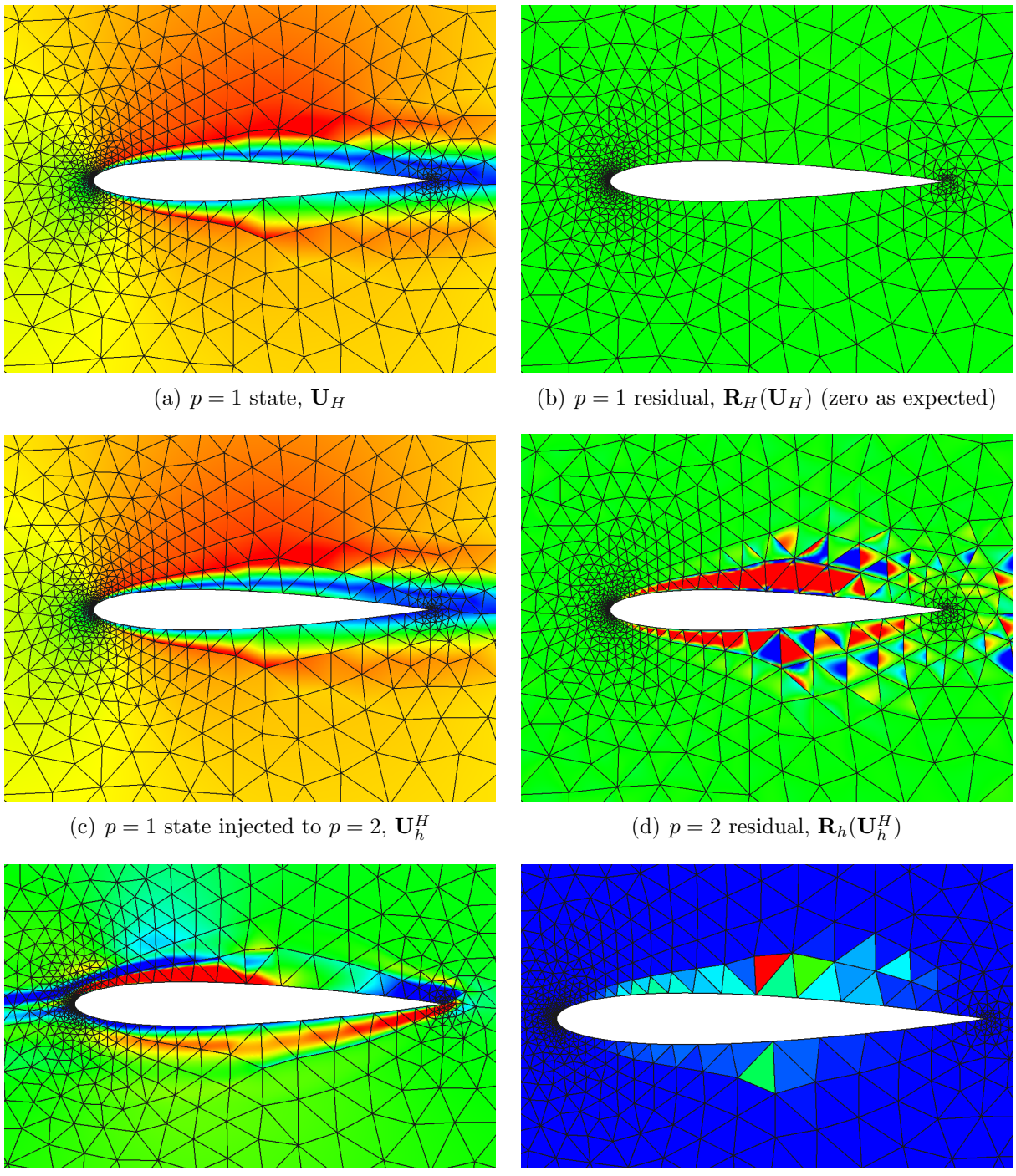
The adjoint-weighted residual error estimate in (5.2.7) can be localized to the elements by keeping track of the contributions from each fine-space element, indexed by e below,

$$\begin{aligned} J_H(\mathbf{U}_H) - J_h(\mathbf{U}_h) \approx -\boldsymbol{\Psi}_h^T \mathbf{R}_h(\mathbf{U}_h^H) &= -\sum_e \boldsymbol{\Psi}_{he}^T \mathbf{R}_{he}(\mathbf{U}_h^H) \\ \Rightarrow \quad \epsilon_e &\equiv \left| \boldsymbol{\Psi}_{he}^T \mathbf{R}_{he}(\mathbf{U}_h^H) \right|, \end{aligned}$$

where the subscript e indicates restriction to element e, and the adaptive indicator ϵ_e is obtained by taking the absolute value of the elemental contributions. When order enrichment is used for the fine space in error estimation, this ϵ_e is the adaptive indicator for each element in the current mesh and can be used directly to drive a mesh adaptation strategy. When element refinement is used for the fine space, then the indicators for the fine sub-elements of a coarse element need to be summed to obtain the coarse-element adaptive indicator. The steps involved in obtaining the adaptive indicator, ϵ_e are summarized graphically in Figure 5.3.1.

5.3.2 Adaptation Mechanics

Numerous strategies exist for translating the error indicator into a modified computational mesh. In CFD for aerospace engineering, the most popular adaptation strategy is h-adaptation, in which only the triangulation forming the mesh is modified. This modification usually consists of targeted refinement and coarsening, although pure node repositioning, sometimes called r-refinement, has also been investigated [27, 96]. For high-order methods, additional strategies include p-adaptation, in which the approximation order is changed on a fixed triangulation [142, 91], and hp-adaptation in which both the order and the triangulation are varied [62, 20, 122, 3, 75, 36, 70, 98, 74, 78]. For CFD applications, in which solutions often possess localized, singular features, h-adaptation is key to an efficient adaptation strategy.



(a) $p = 1$ state, $\mathbf{U}_H$

(b) $p = 1$ residual, $\mathbf{R}_H(\mathbf{U}_H)$ (zero as expected)

(c) $p = 1$ state injected to $p = 2$, $\mathbf{U}_h^H$

(d) $p = 2$ residual, $\mathbf{R}_h(\mathbf{U}_h^H)$

(e) $p = 2$ lift adjoint, $\mathbf{\Psi}_h$

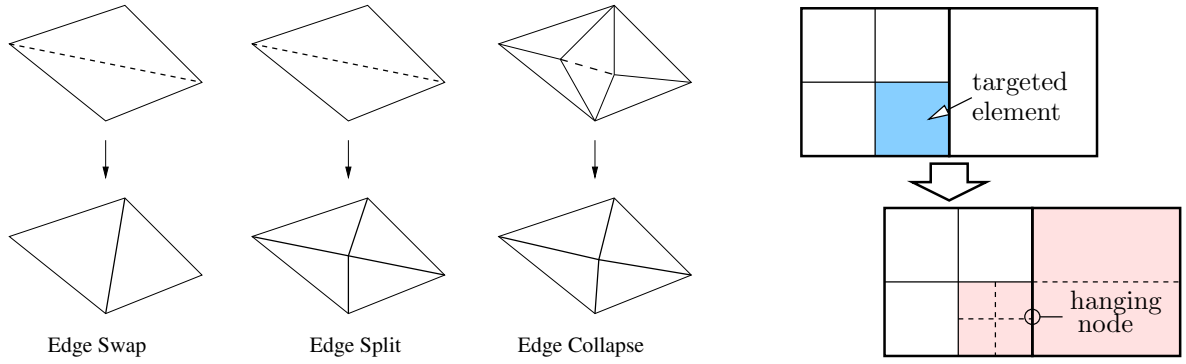
(f) Error indicator, $\epsilon_e = |\mathbf{\Psi}_{he}^T \mathbf{R}_{he}(\mathbf{U}_h^H)|$

Figure 5.3.1: Quantities involved in the calculation of the error estimate and adaptive indicator for $Re = 5000$, $\alpha = 2°$, $M = 0.1$ flow over a NACA 0012 airfoil.

With the growing availability of high-order methods, however, *hp*-adaptation can now be used to further improve efficiency.

Local Refinement

Many approaches to adapting a mesh rely upon the application of local operators through which the mesh is modified incrementally. A simple example of a local operator is element sub-division in a setting that supports non-conforming, or hanging, nodes [19, 80, 123, 36, 140]. For triangular and tetrahedral meshes, local mesh modification operators consist of node insertion, face/edge swapping, edge collapsing, and node movement, as illustrated in Figure 5.3.2(a). These operators have been studied extensively by various authors [54, 26,



(a) Unstructured local mesh operators (b) Hanging-node refinement

Figure 5.3.2: Local mesh modification adaptation operators in two dimensions.

30, 155, 63, 156, 7, 112, 113] in different contexts. The primary advantage of local operators is their robustness: the entire mesh is not regenerated all at once, but rather each operator affects only a prescribed number of nodes, edges, or elements.

Another local operation, especially relevant for discontinuous Galerkin discretizations, is hanging-node mesh refinement, illustrated for quadrilateral elements in Figure 5.3.2(b). The non-conforming nature of a hanging node mesh does not significantly affect the DG discretization, which does not enforce solution continuity between elements. Hanging node refinement could be isotropic or directional [31]. It is particularly useful for initially-structured, stretched meshes for viscous flows, in which only a few refinements of key regions may dramatically improve the accuracy of an output.

Global Re-Meshing

Another approach to adapting a mesh is global re-meshing, in which a new mesh is generated for the entire computational domain. The original, or background, mesh is used to store desired mesh characteristics during regeneration. For applications to adaptation, the desired mesh characteristics are often described using a Riemannian metric, the idea being that in an optimal mesh, all edge lengths will have unit measure under the metric [30, 63]. In a

Cartesian coordinate system of dimension d, an infinitesimal segment $\delta\mathbf{x}$ has length $\delta\Gamma$ under a Riemannian metric $\mathbf{M}$,

$$\delta\Gamma^2 = \delta\mathbf{x}^T \ \mathbf{M} \ \delta\mathbf{x} = \delta x_i \ M_{ij} \ \delta x_j, \tag{5.3.1}$$

where δx_i are the components of $\delta\mathbf{x} \in \mathbb{R}^d$, M_{ij} are the components of the symmetric, positive definite metric, $\mathbf{M} \in \mathbb{R}^{d\times d}$, and summation is implied on the repeated indices $i, j \in [1, \ldots, d]$.

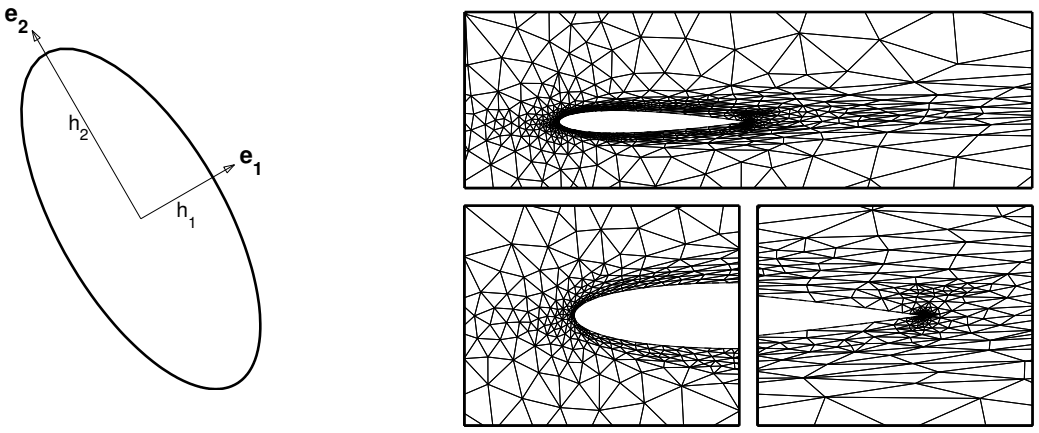
The metric $\mathbf{M}$ contains information on the desired mesh edge lengths in physical space. As $\mathbf{M}$ is symmetric and positive definite, the unit measure requirement,

$$\mathbf{x}^T \ \mathbf{M} \ \mathbf{x} = 1,$$

describes an ellipsoid in physical space that maps to a sphere under the action of the metric. The eigenvectors of $\mathbf{M}$ form the orthogonal axes of the ellipsoid – i.e. the principal directions. The corresponding eigenvalues, λ_i, are related to the lengths of the axes, h_i, via

$$\lambda_i = \frac{1}{h_i^2} \quad \Rightarrow \quad \frac{h_i}{h_j} = \left(\frac{\lambda_j}{\lambda_i}\right)^{1/2}$$

Physically, the h_i are the principal stretching magnitudes. A diagram of a possible ellipse resulting from the unit-measure requirement in two dimensions is given in Figure 5.3.3(a). Thus, the ratio of eigenvalues of $\mathbf{M}$ can be used to define a desired level of anisotropy.



(a) Riemannian metric ellipse

(b) Sample mesh obtained by global re-meshing

Figure 5.3.3: Depiction of adaptation using global metric-based re-meshing. On the left, an ellipse representing requested mesh sizes implied by equal measure under a Riemannian metric $\mathbf{M}$, together with the principal directions, $\mathbf{e}_i$, and the associated principal stretching magnitudes, h_i. On the right, sample mesh for viscous flow generated using global re-meshing.

A successful approach for generating simplex meshes based on a Riemannian metric is mapped Delaunay triangulation, in which a Delaunay mesh generation algorithm [132] is

applied in the mapped space, allowing for the creation of stretched and variable-size triangles or tetrahedra [95]. This method is implemented in the Bi-dimensional Anisotropic Mesh Generator (BAMG) [22, 69], which has been used in various finite volume [128, 147] and discontinuous Galerkin [46, 11, 108] applications requiring anisotropic meshes. An example of an output-adapted mesh obtained using BAMG is shown in Figure 5.3.3(b).

Targeting Strategies

In global re-meshing, all elements can be refined or coarsened based on their error indicators. However, when using local operators, such as hanging-node or order refinement, one must decide which elements to target. The determination of which elements to refine or coarsen has important implications, as too little refinement at each adaptive iteration may result in an unnecessary number of iterations, while too much refinement may result in an expensive solve on an overly-refined mesh. A useful tool for analyzing adaptation targeting strategies is an error distribution histogram [2], in which elements are binned according to the error indicator. A typical assumption is that in an ideal mesh, the error is equidistributed among the elements [7], and this situation yields a "delta" histogram, in which all elements lie in the same bin. In contrast, the initial coarse mesh will generally have some distribution of error indicators, and the goal of an adaptation targeting strategy is to drive the histogram towards the ideal delta distribution. This characterization of adaptation targeting strategies also holds for runs in which a maximum element count is specified instead of an error tolerance. The ideal mesh in this case is one for which the error is equidistributed among a number of elements within the element budget [161, 162].

Most adaptation targeting strategies are based on a decreasing refinement threshold [101], in which elements with the highest error are targeted for refinement first so that the mesh size grows gradually. For example, a fixed-fraction approach prescribes a fraction of elements with the highest error indicator to be refined at each adaptation iteration, such that the decreasing threshold is a function of the shape of the error histogram. Then, the elements targeted for adaptation are typically refined in a locally uniform manner, e.g. by splitting all edges in half. This simple approach has been applied to output-based adaptation in several studies[17, 68, 12, 137, 78, 100, 32]. The fixed-fraction parameter is often chosen heuristically in a trade-off between an excessive number of iterations and a risk of over-refinement. Nevertheless, the method works quite well for practical problems.

Incorporating Anisotropy

An important ingredient in h-adaptation for aerodynamics is the ability to generate stretched elements in areas such as boundary layers, wakes, and shocks, where the solution exhibits anisotropy, which refers to variations of disparate magnitudes in different directions. Anisotropy can be created to a limited, albeit often sufficient, extent with hanging-node refinement [134, 141]; yet global re-meshing with unstructured triangular and tetrahedral grids offers the most flexibility in tailoring the required stretching.

For spatially second-order methods, the dominant method for detecting anisotropy involves estimating the Hessian matrix of second derivatives of a scalar (e.g. Mach number) computed from the solution [118, 95, 30, 63]. A mesh metric is obtained from the Hessian by requiring that the approximation error estimate of the scalar quantity u be the same in any chosen spatial direction. The Hessian matrix stores precisely this information, so that this requirement leads to a metric that is directly proportional to the Hessian. The proportionality constant can be tied to the error indicator to incorporate the output error estimate [147].

The definition of a metric tensor becomes difficult for high-order methods because the standard Hessian matrix approach assumes linear approximation of the scalar quantity. One possible extension is based on constructing a metric around the direction of maximum $p+1$st derivative [42, 41, 110]. Another extension involves the calculation of the order $p+1$ derivative tensor that is analogous to the Hessian for $p = 1$, or the use of surrogate heuristics based on inter-element jumps for quadrilateral or hexahedral meshes [87].

The metric tensor may also be used to guide an adaptation procedure based on local operators, for example in an effort to make all edges approximately the same length when measured using the metric tensor [30, 38, 156, 113].

The methods mentioned so far rely on a priori analysis and heuristics to predict the desired mesh anisotropy. For example, the approximation error assumptions are made without regard to the output of interest by using a single scalar, such as the Mach number, to control the anisotropy for a system of equations. However, the assumption that the directional approximation error must be equidistributed for one or more scalar variables at each point in the domain may not be valid, especially when only a scalar output is desired. This observation has motivated research into adaptation algorithms that more directly target the error indicator.

Hessian-based approximation error estimates have been combined with output-based a posteriori error analysis to arrive at an output-based error indicator that explicitly includes the anisotropy of each element [53, 52, 51]. Other works have considered directional output error estimates for quadrilateral and hexahedral meshes [125], and direct node position optimization using an auxiliary adjoint problem [130].

For general unstructured meshes, output error estimation has been incorporated into local mesh operators of element swapping, node movement, element collapse, and element splitting [114]. A direct optimization approach for output error reduction has also been applied to quadrilateral and hexahedral elements, using anisotropic discrete refinement options [76, 66, 31]. Finally, a recent work extends the ideas of direct mesh optimization to general unstructured meshes with global re-meshing, through optimization of a parametrized metric field via local refinement sampling [162].

Adapting in Order

In order/p-adaptation [142], the approximation space is refined or coarsened by changing the order of approximation. With the discontinuous Galerkin method, changing the order is simple and can be done locally on each element [91, 43]. Advantages of p-adaptation are

that the computational mesh remains fixed and that an exponential error convergence with respect to degrees of freedom (DOF) is possible for sufficiently smooth solutions. Disadvantages include difficulty in handling singularities and areas of anisotropy and the need for a reasonable starting mesh.

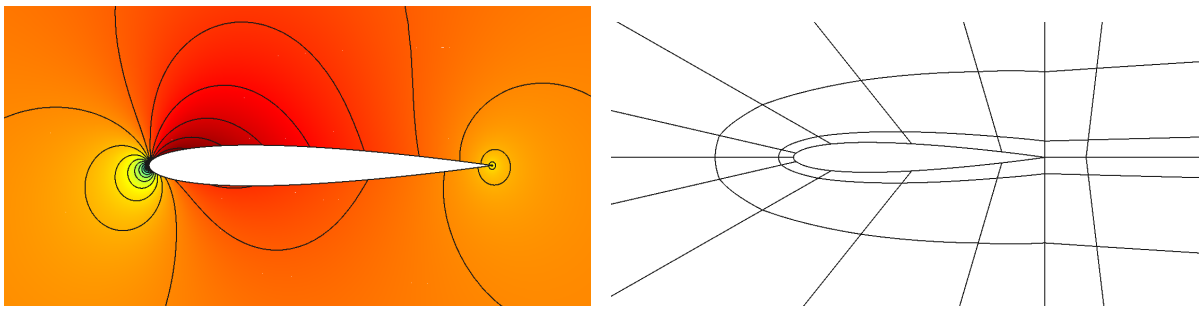
hp-adaptation strives to combine the best of both strategies, employing *p*-refinement in areas where the solution is smooth and *h*-refinement near singularities or areas of anisotropy. The motivation for this strategy is that, in smooth regions, *p*-refinement is more effective at reducing the error per unit cost, compared to *h*-refinement [148, 78]. Implemented properly, *hp*-adaptation can isolate singularities and yield exponential error convergence with respect to DOF. In practice, however, the difficulty of *hp*-adaptation lies in making the decision between *h*- and *p*-refinement, which typically requires either a solution regularity estimate or a heuristic algorithm [78, 25]. An approach that employs output error information to make the *hp* decision and to choose the appropriate element anisotropy for quadrilateral and hexahedral elements has also been presented [32].

5.3.3 Examples

The following sub-sections demonstrate applications of output-based refinement to various aerodynamic flows. We consider inviscid, viscous, and turbulent flows in two and three dimensions. Our figure of merit will generally be degrees of freedom, although we also present some results for computational time.

Inviscid Flow over an Airfoil

In this example we adapt a mesh to predict drag in inviscid flow over a NACA 0012 airfoil at $M_\infty = 0.5$ and $\alpha = 2°$. Figure 5.3.4 shows the Mach number contours and the initial mesh. initial mesh We compare uniform refinement to output-driven, isotropic, hanging-node, fixed-fraction refinement with $f^{\text{adapt}} = 0.05$.



(a) Mach contours (b) Initial mesh

Figure 5.3.4: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$: Contours of the Mach number and the initial mesh.

Figure 5.3.5 shows the results of one adaptation run, using $p = 2$. The drag coefficient error decreases as the number of elements increases, at a steady rate. At each iteration, both the estimated and actual errors are shown, and these agree very well, indicating that the error estimate is effective at predicting the true error. The figure also shows uniform refinement results, in which the error does not drop as rapidly.

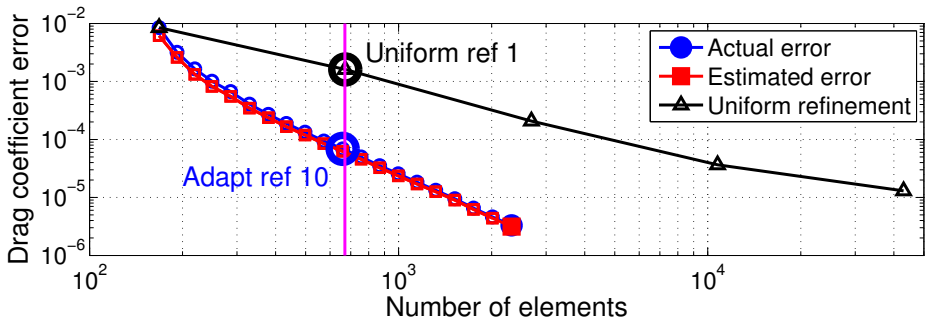
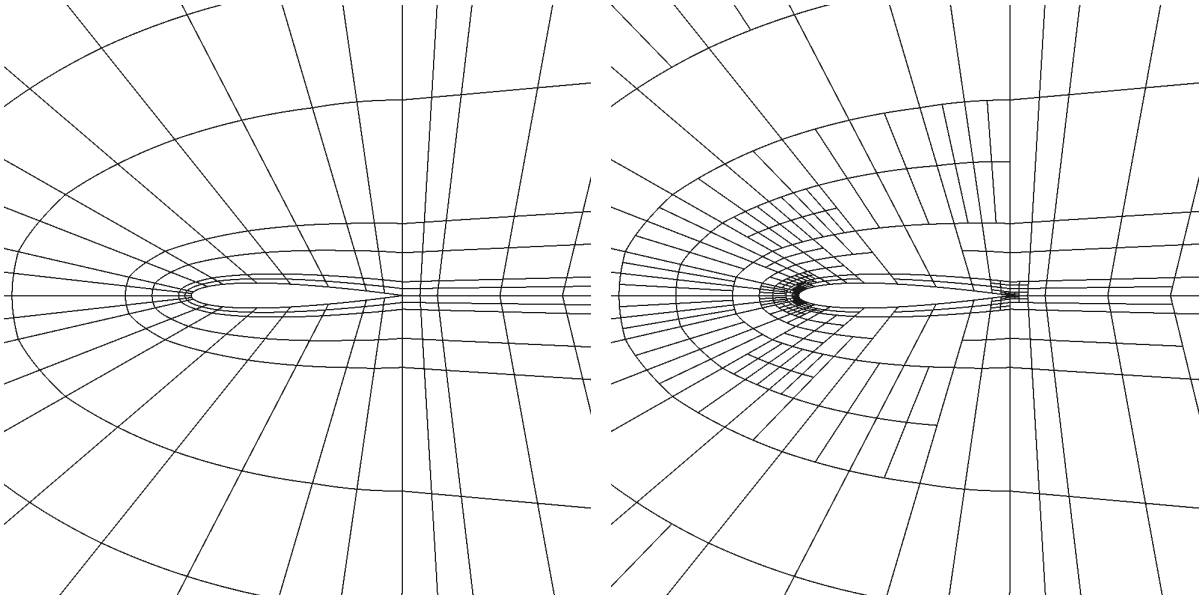


Figure 5.3.5: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$: actual and estimated drag coefficient errors obtained from $p = 2$ output-based adaptation, compared to uniform refinement.

We can gain insight into where adaptive refinement allocates degrees of freedom by comparing meshes from uniform refinement and adaptive refinement at a similar number of elements. Figure 5.3.5 identifies two such meshes, which are shown in Figure 5.3.6. We can see that for accurate drag prediction, the leading and trailing edges of the airfoil must be resolved with small elements.

Figure 5.3.7 shows the convergence of the drag coefficient error with a measure of the mesh size, $\texttt{dof}^{-1/2}$, for $p = 1$ and $p = 2$ approximation. We see that uniform refinement converges at a rate of 2.5 for $p = 1$ and $p = 2$. This rate is limited due to a singularity of the flow at the trailing edge, which then dominates the error convergence. On the other hand, we see that adaptive refinement produces meshes that make better use of degrees of freedom: the equivalent "convergence rate" for $p = 2$ adaptation is approximately 5, i.e. the expected super-convergent rate of $2p + 1$ for an inviscid simulation. We note that although we usually speak of convergence rates only for uniform refinement studies, this notion generalizes to adaptive mesh refinement [161]. For example, for a case in which the entire domain is important (e.g. a very smooth solution), adaptive refinement would (eventually, in turn) target the entire domain too so that its error versus degrees of freedom would converge at the same rate as for uniform refinement. More interestingly, when there are (isolated) singularities present, a situation in which uniform refinement would see its rate limited, adaptive refinement could "quarantine" these singularities with nominal resources and allocate the rest to resolving the remaining smooth regions at the optimal rate. This is indeed what we observe in Figure 5.3.7, where, while uniform refinement sees its rate limited by non-smooth solution features for high order, adaptive refinement attains a higher, super-convergent rate. We also note that correcting the output with the error estimate further increases the order of accuracy by 1 to $2p + 2$. In this example, the fine-space problem



(a) Uniform refinement, 672 elements (b) Adaptive refinement, 659 elements

Figure 5.3.6: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$: comparison of meshes obtained from uniform refinement and adaptive refinement, at a similar total number of elements.

was solved exactly, but solving it approximately (e.g. with a few block-Jacobi smoothing iterations) does not appreciably change the results. For more complex problems, not solving the fine-space adjoint exactly can yield poor error estimates.

Adapting using an output adjoint consumes more resources, i.e. CPU time, than just solving the primal problem because of the need for a fine-space adjoint. To determine whether this additional cost pays off, Figure 5.3.8 plots the convergence of the drag output with CPU time, for both an exact and an approximate fine-space adjoint solution. We see that the cost of an exact adjoint solve pushes the output-based results close to the uniform refinement results – the corrected results still remain more efficient, however. If we instead solve the fine-space adjoint problem approximately, we can reduce the CPU time of the output-based results by a factor of 2 or more.

Finally, Figure 5.3.9 shows the breakdown of the CPU time at each adaptive iteration when using the exact and approximate adjoint solves. The exact adjoint solve consumes more than half of the CPU time, whereas the approximate adjoint solve is much cheaper: less than 10% of the total CPU cost. In both cases, the cost of adapting the mesh is negligible compared to the primal and adjoint solutions.

Viscous Flow over an Airfoil

We consider again the case of a NACA 0012 airfoil in viscous flow, introduced in Section 5.2.5. We use $p = 2$ solution approximation and fixed-fraction, $f^{\text{adapt}} = 0.05$, and isotropic hanging-

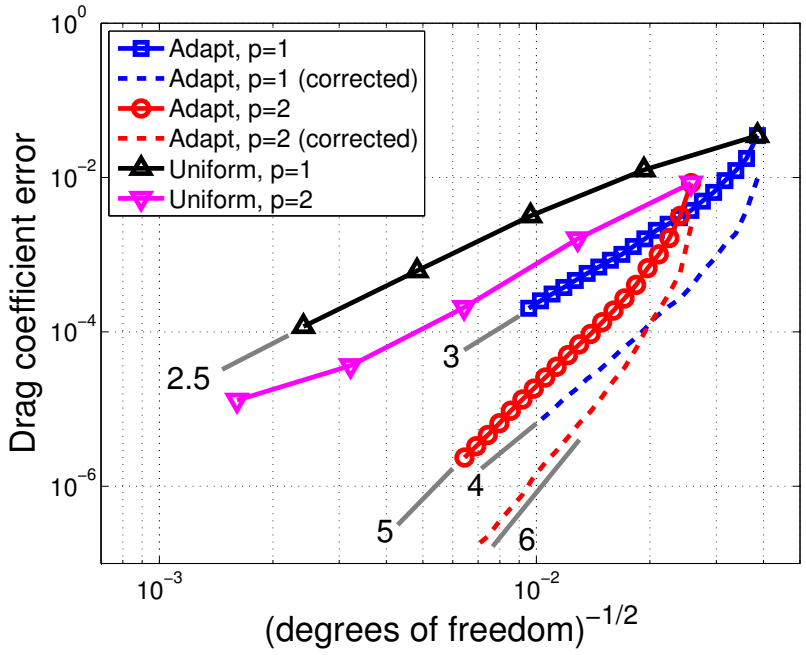
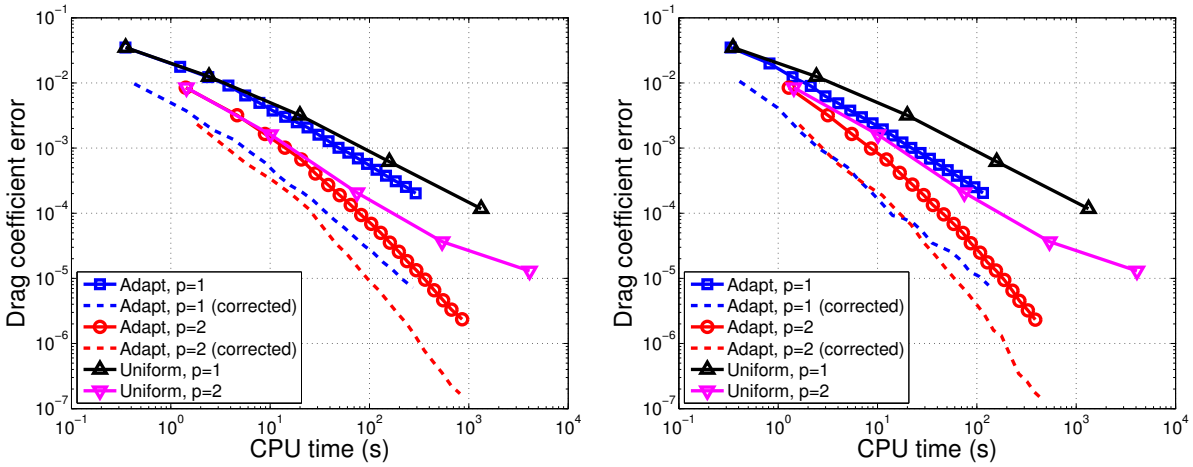
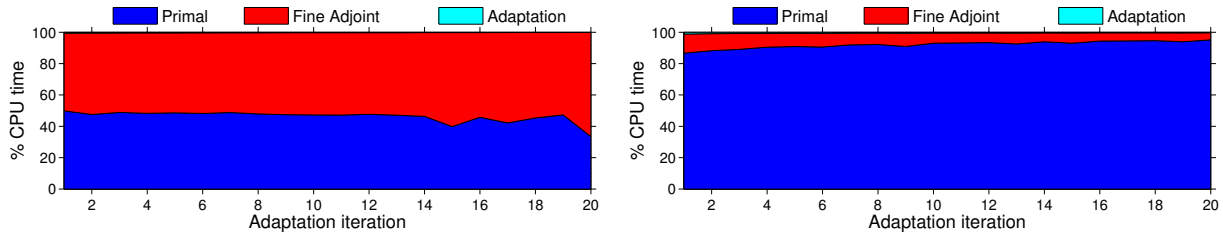


Figure 5.3.7: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$: drag coefficient convergence with mesh size, $\texttt{dof}^{-1/2}$, using uniform refinement and output-based adaptation. Orders of approximation $p = 1$ and $p = 2$ are shown. The convergence rate is identified for each run: we observe a rate of $2p + 1$ for the output-adapted results, and $2p+2$ for the adapted results corrected with the error estimate. Uniform refinement yields suboptimal rates due to a singularity at the airfoil trailing edge.



(a) Exact fine-space adjoint (b) Approximate fine-space adjoint

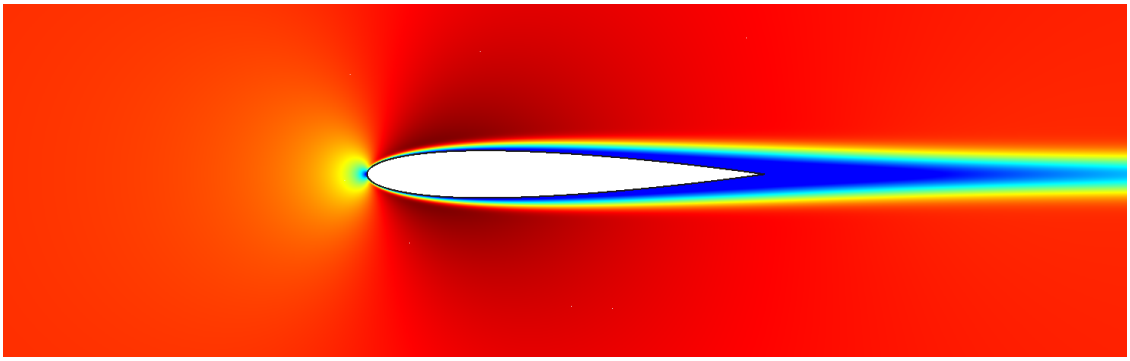
Figure 5.3.8: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$: output convergence with CPU time for different adaptation strategies. Using an approximate fine-space adjoint solve of five block-Jacobi smoothing iterations yields noticeable efficiency improvements over an exact (machine precision) adjoint solve. Correcting the output with the error estimate further improves efficiency.



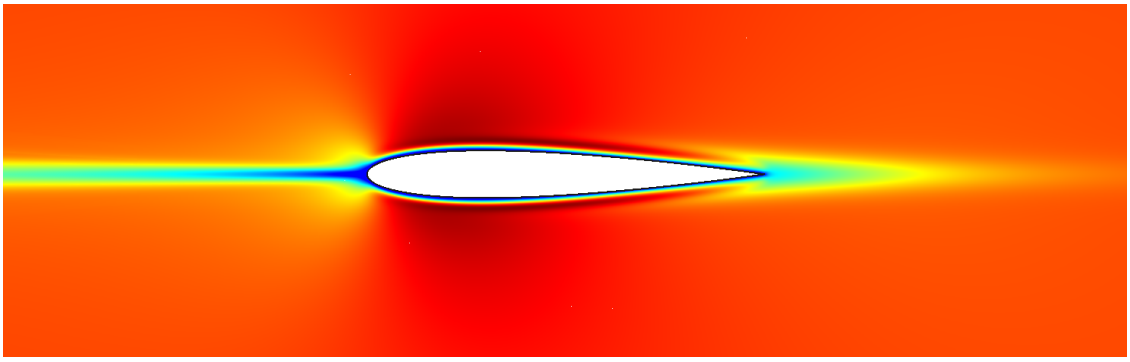
(a) Exact fine-space adjoint (b) Approximate fine-space adjoint

Figure 5.3.9: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$: timing breakdown, summed over all adaptive iterations for $p = 2$, when using exact and approximate adjoint solves. output convergence with CPU time for different adaptation strategies.

node adaptation. The output of interest is the drag coefficient, and the initial mesh is the same as in the previous example. Figure 5.3.10 shows contours of the primal solution (Mach number) and the drag adjoint solution (x-momentum component).



(a) Mach number



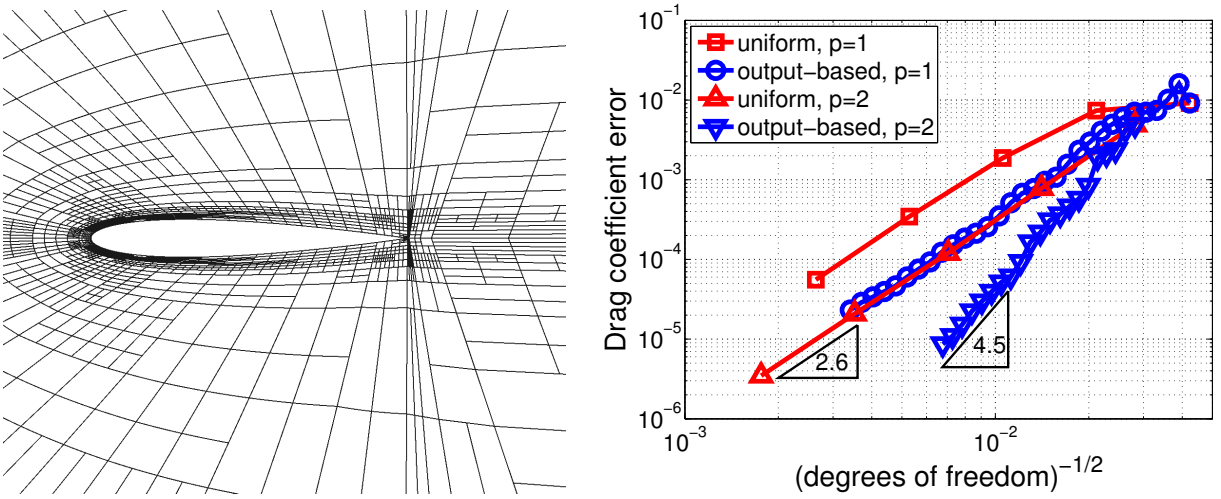
(b) x-Momentum drag adjoint

Figure 5.3.10: NACA 0012, $M_\infty = 0.5$, $\alpha = 0°$, $Re = 5000$: contours of the Mach number and the x-momentum component of the drag adjoint. Note the wake reversal in the adjoint solution, which indicates high sensitivity of the output to residual perturbations made upstream of the airfoil.

Figure 5.3.11 shows the adaptive results when using a drag-adjoint weighted residual indicator, compared to uniform refinement. As in the previous example, we see that adaptive refinement produces meshes that make better use of degrees of freedom: the equivalent convergence rate for $p = 2$ is approximately 4.5, slightly above the expected rate of $2p$ for a viscous simulation. Uniform mesh refinement at $p = 2$ only attains a rate of 2.6. Thus, as in the previous example, adaptive refinement can "uncover" the optimal convergence rates for high-order discretizations, resulting in a more efficient use of degrees of freedom.

Which Output?

In this example, we consider a NACA 0012 airfoil with a closed trailing edge and a far field approximately 40 chord-lengths away. The initial mesh of cubic quadrilateral elements is



(a) Final drag-adapted mesh for $p = 2$

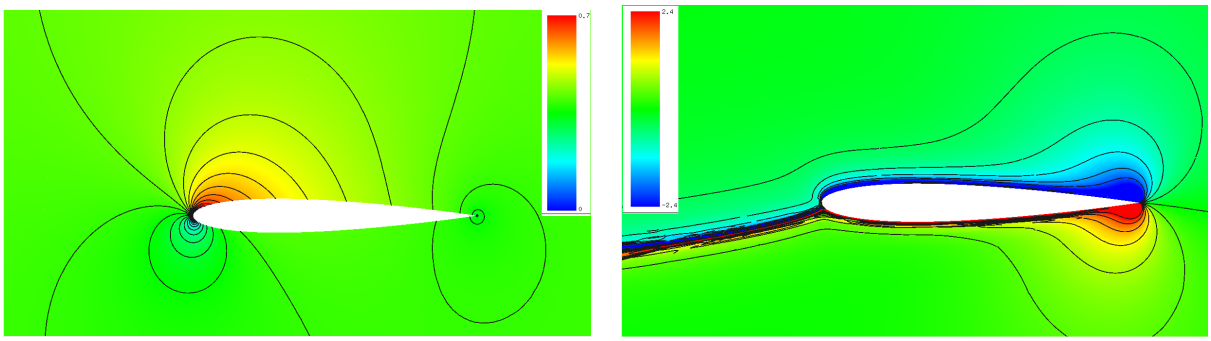
(b) Output convergence

Figure 5.3.11: Adaptation results for viscous $Re = 5000$, $M_\infty = 0.5$ flow over a NACA 0012 airfoil at $\alpha = 0$. The output of interest is the drag force coefficient on the airfoil, and, at each adaptive iteration, the adjoint is solved approximately with $\nu = 10$ element-block Jacobi smoothing iterations on the fine space. Using an exact adjoint solve does not perceptibly change the adaptive result.

illustrated in Figure 5.3.14. While the initial mesh appears structured, this structure disappears with the first adaptation iteration and the mesh storage is always fully unstructured. In the following results, quadratic solution approximation, $p = 2$, was used in the discretization, and isotropic h-adaptation was driven by a fixed-fraction strategy with $f^{\text{adapt}} = 0.1$, meaning that at each step of the adaptation, those elements lying in the top 10% of the error criterion were chosen for refinement.

Mach number contours for the airfoil in inviscid flow at $M_\infty = 0.4$, $\alpha = 5°$ are shown in Figure 5.3.12. Three different engineering outputs are considered: drag coefficient, lift coefficient, and leading-edge moment coefficient. All of these outputs were computed using integrals of the inviscid momentum flux, that is, the pressure, on the airfoil surface. Adjoint solutions associated with these outputs were used to drive three different adaptation runs. One adaptation run was also performed using an "entropy-adjoint" indicator [50], in which the entropy variables are interpreted as a "free adjoint" for an output that expresses an entropy balance statement. For comparison, an unweighted residual indicator, equivalent to summing the absolute values of the discrete fine-space residuals, was also tested.

Figure 5.3.13 shows the results of adaptation runs driven by the different indicators. Uniform mesh refinement results are given for comparison. The plots show the error in the engineering outputs versus degrees of freedom. Each "truth" output was calculated from a $p = 3$ solution on a mesh obtained by uniformly refining the finest output-adapted mesh. For all three outputs of interest, the adjoint-based adaptive strategies, including the entropy



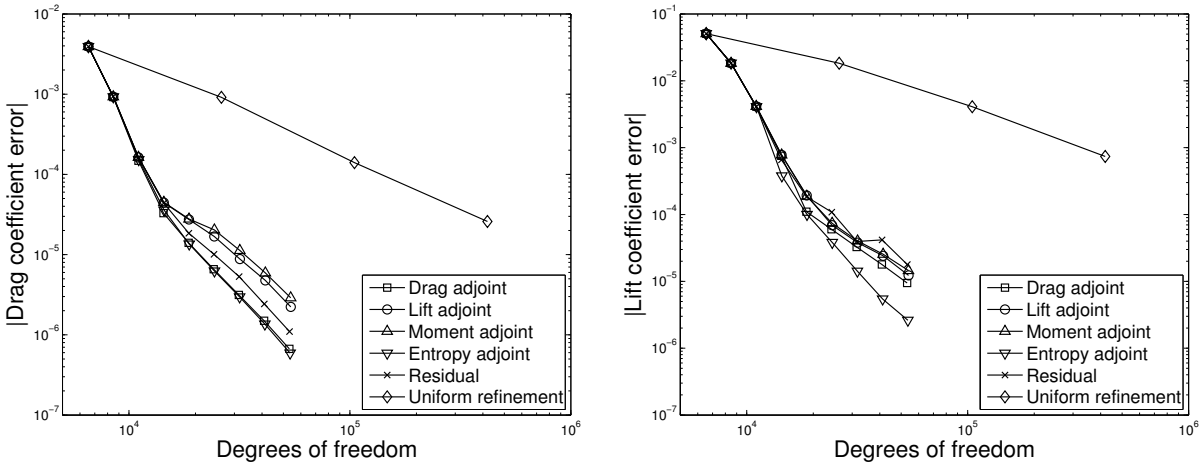
(a) Mach number contours

(b) x-momentum moment adjoint

Figure 5.3.12: NACA 0012, $M_\infty = 0.4$, $\alpha = 5°$: Contours of the Mach number and the x-momentum component of the moment adjoint.

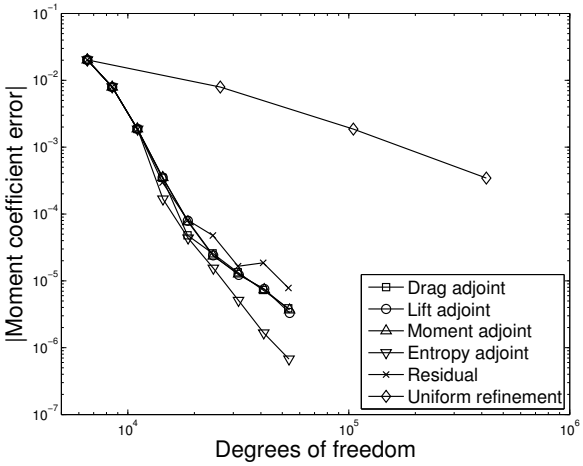
adjoint, perform similarly and are orders of magnitude better than uniform mesh refinement. The unweighted residual indicator performs well for the drag output, and similarly to the output adjoints for the lift and moment outputs. Interestingly, the refinement based on the entropy adjoint actually gives better predictions for lift and moment than the refinements that specifically target those outputs. These results are certainly surprising, but not actually paradoxical, because the procedure does have empirical elements. For example, the lift and moment adaptive indicators target the stagnation streamline in front of the airfoil, perhaps excessively so. As shown in Figure 5.3.12 for the moment output, the adjoint varies rapidly across the stagnation streamline. This behavior was suggested in the analysis of Giles and Pierce who found that a square root singularity with respect to distance from the stagnation streamline exists for sources that perturb the stagnation pressure [58]. Intuitively, a force output on the airfoil should respond differently to perturbations that affect the flow over the upper surface of the airfoil versus to those that affect the flow over the lower surface of the airfoil. The singularity is strongest for the lift and moment outputs, and for these cases the performance of the output adjoint adaptation deteriorates the most. The noise created by polynomial approximation of the adjoint on discrete finite elements in this area may be responsible for the excessive refinement.

The meshes after eight adaptation iterations of each strategy are shown in Figure 5.3.14. The leading edge, trailing edge, and upper surface of the airfoil are consistently targeted for refinement by the adjoint indicators. The unweighted residual adaptation targets the vicinity of the leading edge and the trailing edge, but not the upper surface of the airfoil, leading to errors in the lift and moment outputs. Refinement of the stagnation streamline is evident in the adjoint-based runs, especially for the lift and moment adaptations. Finally, we remark that although residual-based refinement performs well in this example, this is not always the case for more complicated flows, such as those involving higher degrees of nonlinearity arising from shocks, turbulence models, or unsteadiness. In addition, residual-based adaptation does not provide an error estimate that could be used as a stopping criterion for the adaptation.



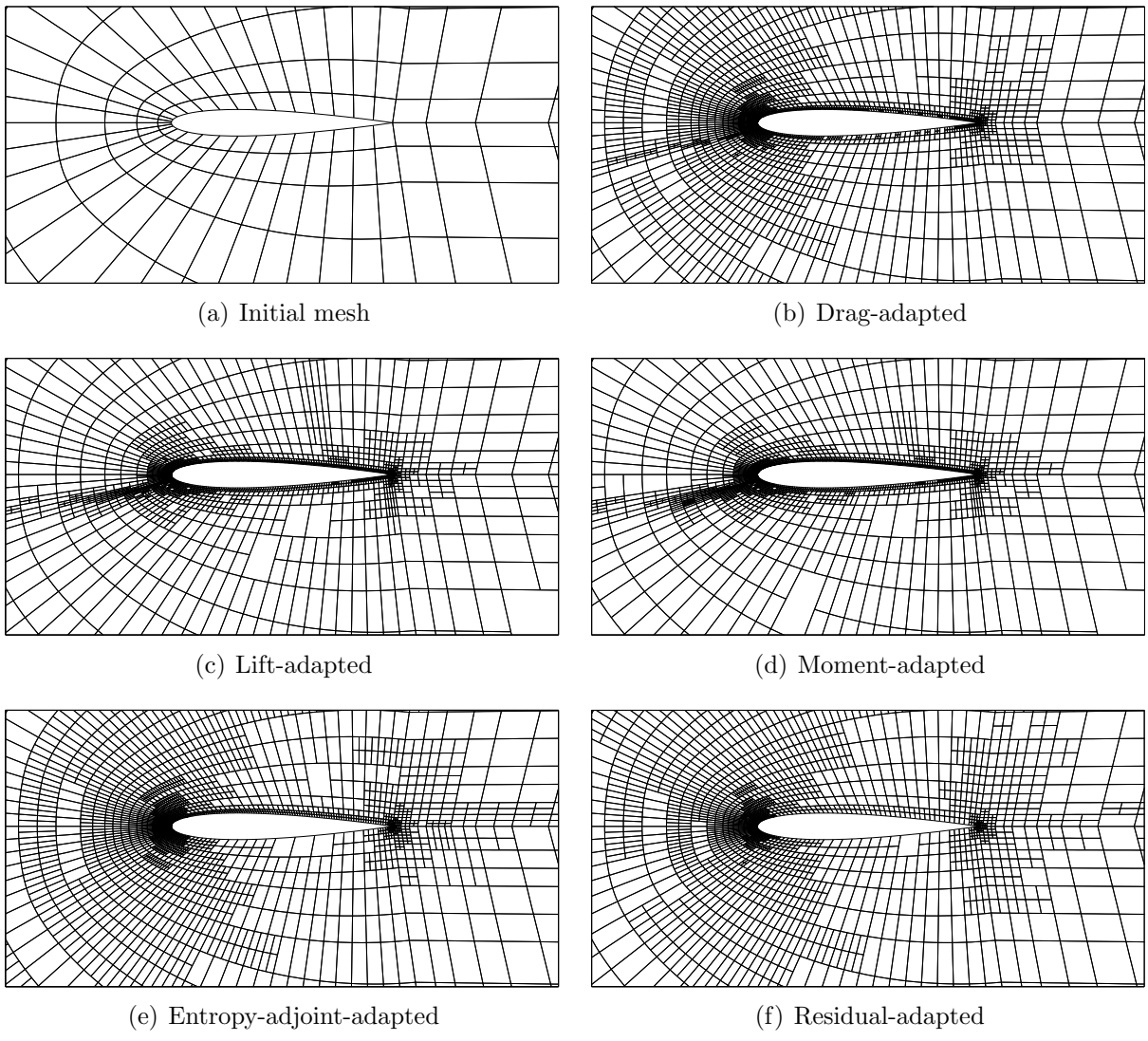
(a) Drag output

(b) Lift output



(c) Moment output

Figure 5.3.13: NACA 0012, $M_\infty = 0.4$, $\alpha = 5°$: Comparison of output convergence histories for various adaptation strategies.



(a) Initial mesh

(b) Drag-adapted

(c) Lift-adapted

(d) Moment-adapted

(e) Entropy-adjoint-adapted

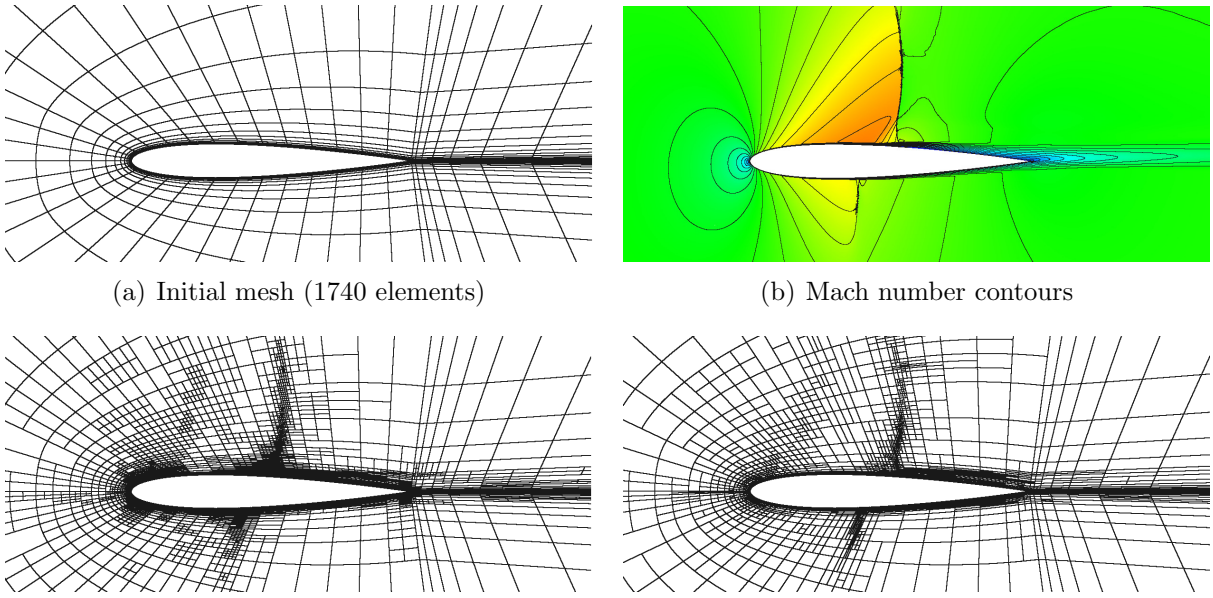
(f) Residual-adapted

Figure 5.3.14: NACA 0012, $M_\infty = 0.4$, $\alpha = 5°$: Meshes after eight adaptation iterations for the tested adaptation strategies.

The same may be argued for adaptation with the entropy adjoint, as that output is typically not of direct engineering interest. However, it turns out that the entropy adjoint output can be related to drag error [40] and hence could be used as an adaptive stopping criterion when targeting drag.

Transonic Turbulent Flow over an Airfoil

To demonstrate the importance of anisotropy, we present an example of output-based adaptation for turbulent, transonic flow over an NACA 0012 airfoil. The free-stream conditions are $M = 0.8, \alpha = 1.25°, Re = 100,000$. In this case, the flow experiences a relatively strong normal shock on the upper surface and a weak shock on the lower surface. Element-wise constant artificial viscosity is used to capture the shocks [120]. $p = 2$ is used for solution approximation on all elements, and the initial mesh consists of 1740 cubic quadrilateral elements, as shown in Figure 5.3.15.



(a) Initial mesh (1740 elements)

(b) Mach number contours

(c) 6^{th} adapted mesh, isotropic (8,736 elements)

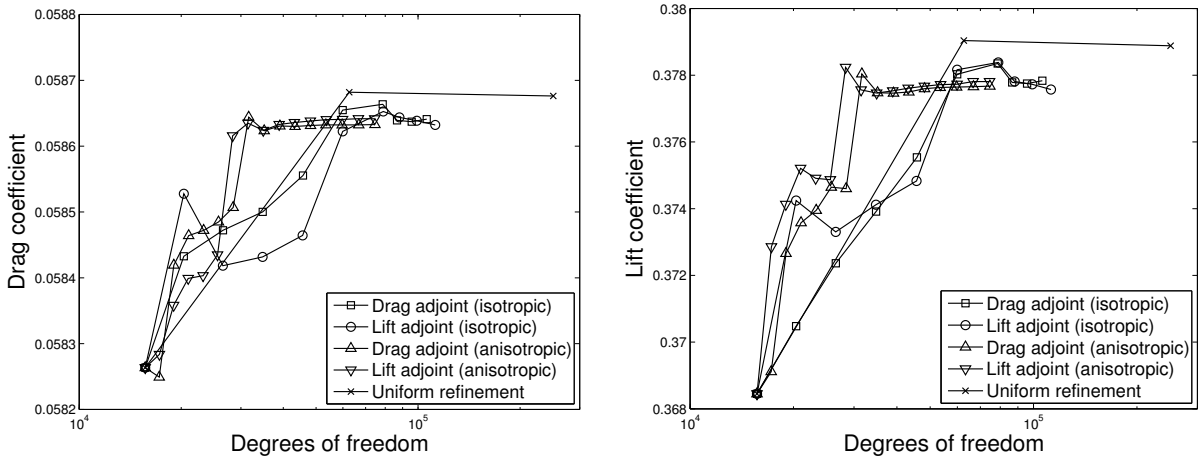
(d) 10^{th} adapted mesh, anisotropic (4,816 elements)

Figure 5.3.15: Turbulent NACA 0012, $M = 0.8, \alpha = 1.25°, Re = 100,000$: Initial mesh, solution contours, and adapted meshes. Note, under isotropic adaptation, some elements are refined anisotropically, and these arise from additional refinements required to keep a maximum refinement ratio between adjacent elements below 2. of 2-to-1 ratio between is due to

Convergence of the drag and lift coefficients is shown in Figure 5.3.16. The plots are similar in that the lift output converges as rapidly as the drag output for all of the adaptation schemes. Also, the drag adaptation performs well for the lift output and vice versa. The anisotropic adaptations, performed using a discrete-choice output-based approach [31], converge much more rapidly compared to the isotropic adaptations: the outputs do not change

much after 40,000 degrees of freedom with the anisotropic adaptation, while changes are still observed after nearly 100,000 degrees of freedom when using isotropic adaptation.

Two of the drag-adapted meshes are shown in Figure 5.3.15: one from isotropic adaptation after six iterations and one from anisotropic adaptation after ten iterations. Differences are evident in the boundary layer and wake, where anisotropic adaptation is more efficient. The shock also appears to be more tightly resolved in the anisotropically-adapted mesh.



(a) Drag output

(b) Lift output

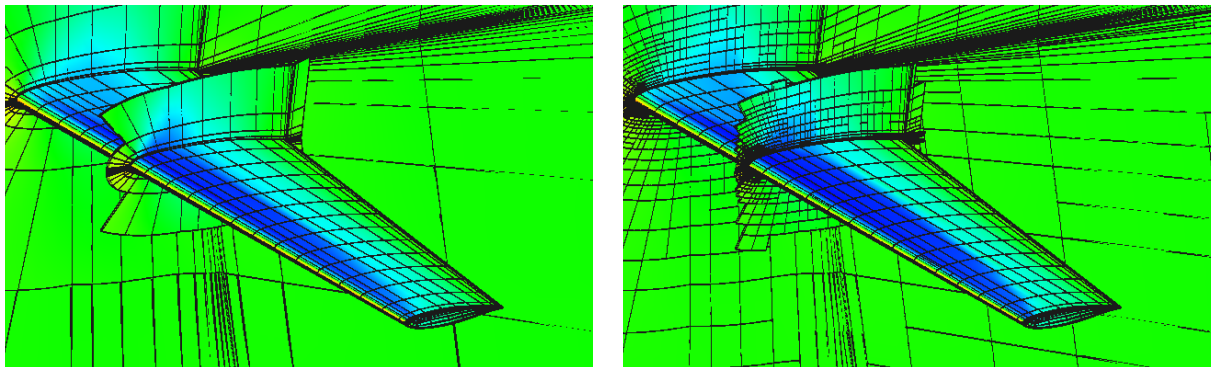
Figure 5.3.16: Turbulent NACA 0012, $M = 0.8, \alpha = 1.25°, Re = 100,000$: Drag and lift convergence with degrees of freedom using isotropic and anisotropic refinement. Note, outputs from uniform refinement overshoot the exact values.

Transonic Turbulent Flow over a Wing

In this example, we demonstrate drag-based adaptation for a steady, three-dimensional, turbulent, transonic flow. We consider the baseline wing geometry (DPW-W1) from the third AIAA Drag Prediction Workshop [57]. The initial curved mesh, shown in Figure 5.17(a), was obtained through agglomeration of cells from a finer structured linear C-grid generated specifically for this purpose. In the agglomeration, each curved hexahedral element was obtained by merging twenty seven linear elements using a distance-based Lagrange interpolation of the nodal coordinates, resulting in cubic ($q = 3$) geometry interpolation. Also, the spacing of the linear mesh is such that the agglomerated mesh presents $y^+ \approx 1$ for the first element off the wall as recommended in the workshop guidelines [56] and the outer boundary is located at 100 mean-aerodynamic-chord-lengths away from the wing.

We use the Spalart-Allmaras turbulence model without trip terms, and element-wise constant artificial viscosity for shock capturing [120]. The baseline flow solution is obtained with linear ($p = 1$) approximation order, and we study anisotropic, output-based *hp*-adaptation using discrete choices that include order enrichment [32] – in this strategy, when deciding

which discrete refinement option to choose, a figure of merit is used that takes into account the benefit (error addressed) and the cost (the increased Jacobian matrix size). The fine-space adjoint used for error estimation is obtained approximately by using $\nu^{\text{smooth}} = 5$ iterations of an element-block Jacobi smoother. All of the adaptive schemes start from the same initial solution. For the adjoint-based adaptation methods, the CPU time taken for the initial adjoint solve is also included in the initial starting time.



(a) Initial pressure contours (29310 cubic elements, $p = 1$).

(b) Pressure contours on the 7^{th} drag-adapted mesh (85377 cubic elements).

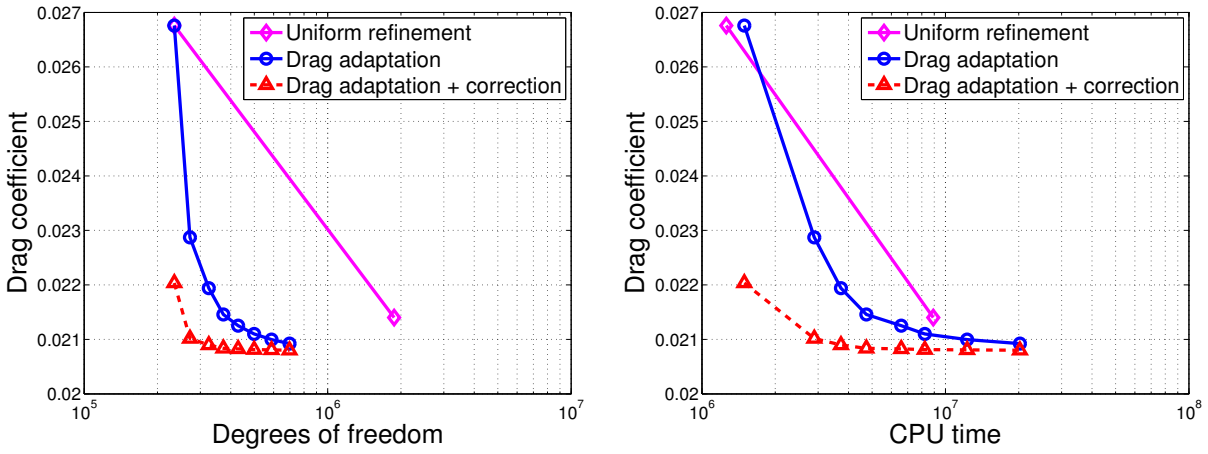
Figure 5.3.17: DPW Wing 1, $M_\infty = 0.76, \alpha = 0.5°, Re = 5 \times 10^6$: Initial and drag-adapted meshes with pressure contours.

We compare two mesh improvement strategies starting from the initial $p = 1$ solution shown in Figure 5.17(a). One of the strategies is uniform h-refinement, in which all hexahedra are divided into 8 elements. The other is drag-based hp-adaptation in which $f^{\text{adapt}} = 10\%$ of the elements is selected for refinement at each adaptation step. Additionally, we fix the overall budget of CPU wall-time for each of the three runs and the last converged solutions obtained within that budget are shown in Figure 5.3.17.

Figure 5.3.18 shows the drag coefficient convergence for the mesh refinement strategies. Note that the dashed lines indicate the output corrected with the error estimate. The difference between these corrected values for the last two adaptation steps of the output-based strategy is within 0.15 counts of drag. Note that the performance in terms of degrees of freedom and CPU time of the output-based strategy is better compared to uniform refinement, especially when using the corrected output results.

Figures 5.3.19 and 5.3.20 show two cuts at representative span-wise positions for the output-based strategy. Note the presence of anisotropic cells along the shock and on the boundary layer, and the minimal use of p-refinement in this case – this is due in part to the under-resolved nature of the mesh for this flow, and to the relatively large cost ascribed to order enrichment in the definition of the merit function.

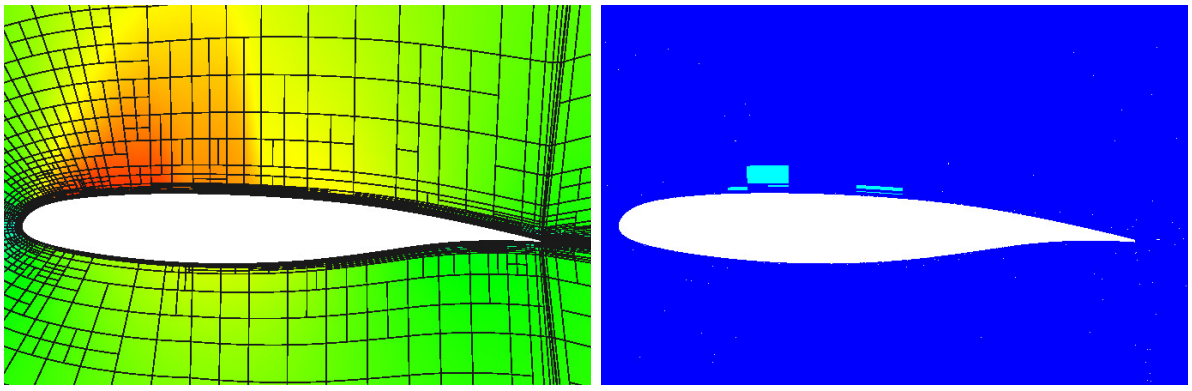
An optimization-based mesh adaptation algorithm may offer insight on "best-practice" gridding guidelines. We notice that several regions of the flow are frequently targeted for



(a) Convergence with degrees of freedom

(b) Convergence with CPU time

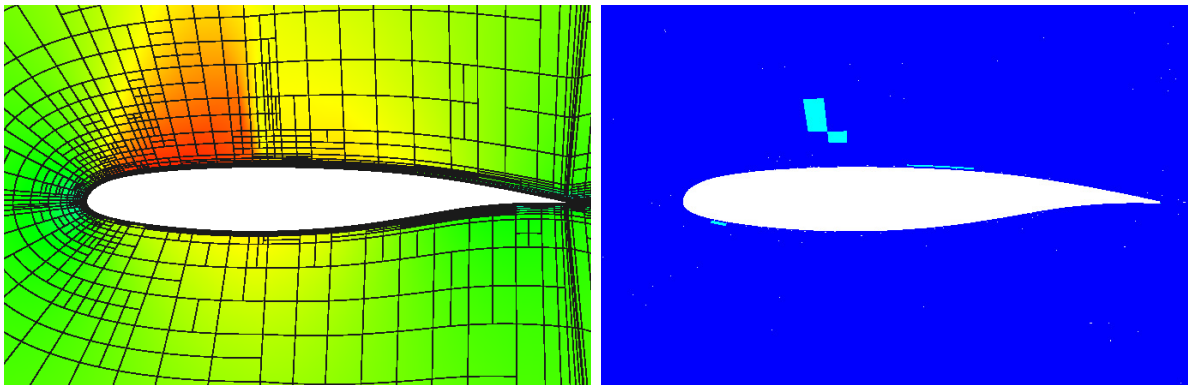
Figure 5.3.18: DPW Wing 1, $M_\infty = 0.76, \alpha = 0.5°, Re = 5 \times 10^6$: drag coefficient convergence for output-based adaptation compared to uniform refinement, using both degrees of freedom and CPU time.



(a) 7^{th} adapted mesh with Mach contours.

(b) 7^{th} adapted mesh p-order distribution; the range is $p = 1 \rightarrow 5$.

Figure 5.3.19: DPW Wing 1, $M_\infty = 0.76, \alpha = 0.5°, Re = 5 \times 10^6$: cut at $y = 220$mm of the drag-adapted mesh. Note, the reference span is $1524mm$.

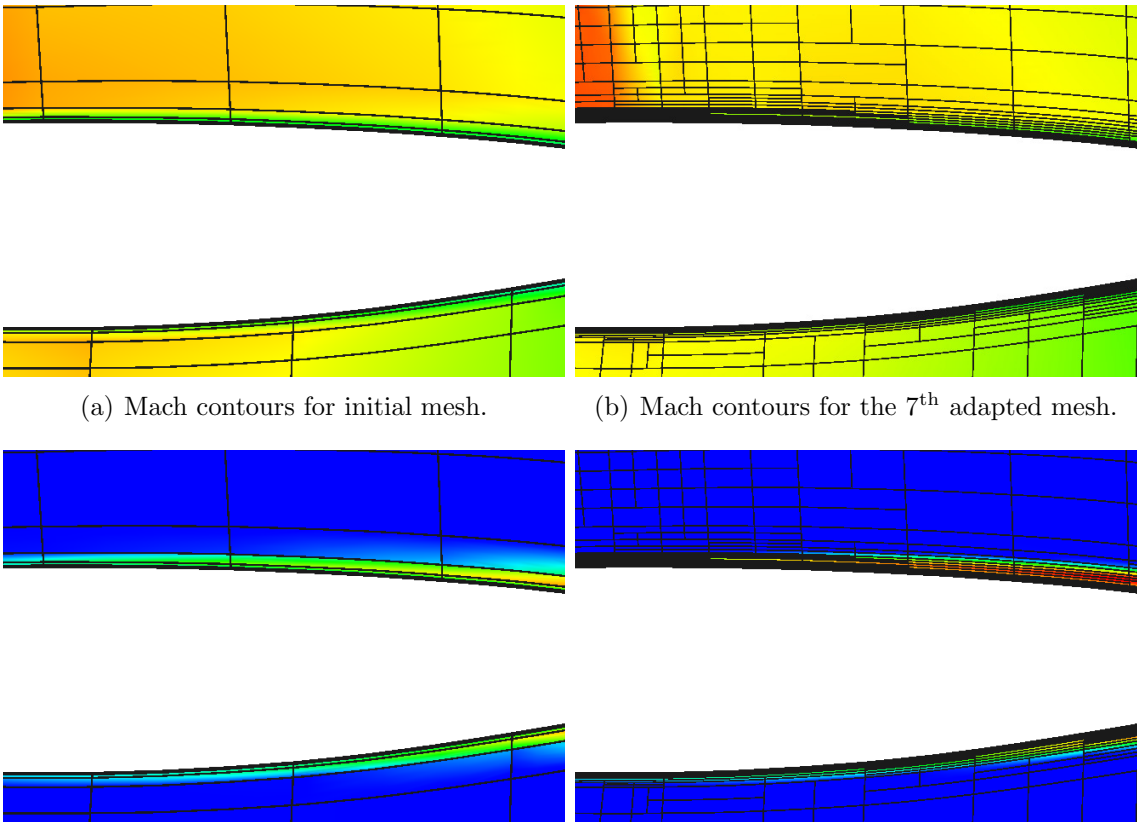


(a) 7^{th} adapted mesh with Mach contours.

(b) 7^{th} adapted mesh p-order distribution; the range is $p = 1 \rightarrow 5$.

Figure 5.3.20: DPW Wing 1, $M_\infty = 0.76, \alpha = 0.5°, Re = 5 \times 10^6$: cut at $y = 620$mm of the drag-adapted meshes.

refinement. One of these regions is near the leading edge where the flow accelerates through the sonic condition. This change in character of the flow causes strong variations in the adjoint solution, which are responsible for large error indicators. Another region is the edge of the boundary layer, where the turbulent working variable, $\tilde{\nu}$, transitions to zero rapidly. The other two regions are the location of shock-boundary-layer interaction and the trailing edge. These regions exhibit strong gradients in $\tilde{\nu}$ that contribute to the drag output. Figure 5.3.21 shows the interaction between the shock and the boundary layer. Note the concentration of cells in the boundary layer and the sharp variation of $\tilde{\nu}$. Further downstream, in the trailing edge region (Figure 5.3.22), the beginning of the turbulent wake is also adapted.



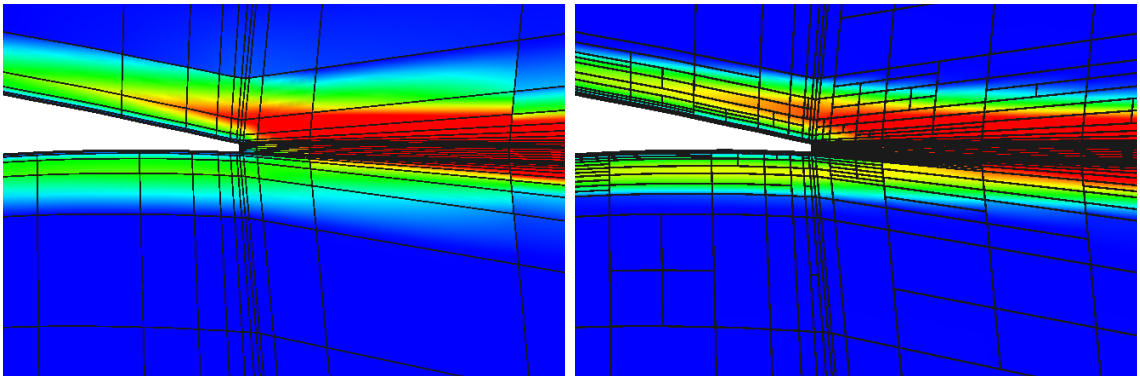
(a) Mach contours for initial mesh.

(b) Mach contours for the 7^{th} adapted mesh.

(c) $\tilde{\nu}$ contours for initial mesh.

(d) $\tilde{\nu}$ contours for the 7^{th} adapted mesh.

Figure 5.3.21: DPW Wing 1, $M_\infty = 0.76, \alpha = 0.5°, Re = 5 \times 10^6$: interaction between shock and boundary-layer at $y = 620$mm.



(a) $\tilde{\nu}$ contours for initial mesh.

(b) $\tilde{\nu}$ contours for the 7^{th} adapted mesh.

Figure 5.3.22: DPW Wing 1, $M_\infty = 0.76, \alpha = 0.5°, Re = 5 \times 10^6$: trailing edge at $y = 620$mm.

Chapter 6

Mesh Optimization

Unstructured meshes offer flexibility in mesh generation and in adaptation, since resolution can be placed only where necessary. *Resolution* refers to the size and shape of an element, as both of these affect the approximation power of the mesh. This information can be encoded in a metric field [22, 115] over the computational domain. The goal in mesh optimization is to relate this metric field to a practical objective, such as minimizing error or computational cost. This has been done through heuristic [30, 7, 26, 63], semi-heuristic [147, 113, 41, 161], and more recently, rigorous [162] ways. In this section we discuss a rigorous output-based approach [162] for determining the metric that gives the best possible mesh.

6.1 Metric-Based Mesh Optimization Algorithm

Our goal is to optimize the computational mesh, T_h, in order to minimize the output error at a prescribed computational cost. In 2012, Yano [162] introduced an approach called Mesh Optimization through Error Sampling and Synthesis (MOESS), which iteratively determines the optimal change in the mesh metric field given a prescribed metric-cost relationship and a sampling-inferred metric-error relationship. In this section, we e briefly review the key elements of this method.

6.1.1 Metric-Based Meshing

A Riemannian metric field, $\mathcal{M}(\vec{x})$, is a field of symmetric positive definite (SPD) tensors that can be used to encode information about the desired size and stretching of a computational mesh. At each point in physical space, $\vec{x}$, the metric tensor $\mathcal{M}(\vec{x})$ provides a "yardstick" for measuring the distance from $\vec{x}$ to another point infinitesimally far away, $\vec{x} + \delta\vec{x}$. This distance is

$$\delta\ell = \sqrt{\delta\vec{x}^T \mathcal{M} \delta\vec{x}}. \tag{6.1.1}$$

After choosing a Cartesian coordinate system and basis for physical space, $\mathcal{M}$ can be represented as a $d \times d$ SPD matrix. The set of points at unit metric distance from $\vec{x}$ is an ellipse, as illustrated in Figure 6.1.1: eigenvectors of $\mathcal{M}$ give directions along the principal axes,

while the length of each axis (stretching) is the inverse square root of the corresponding eigenvalue.

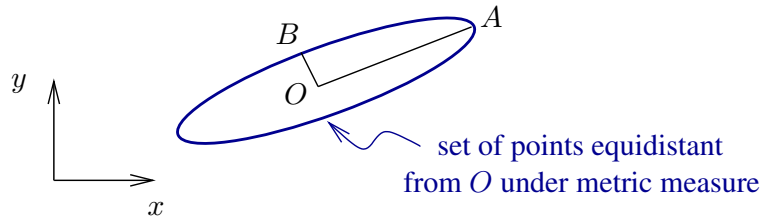
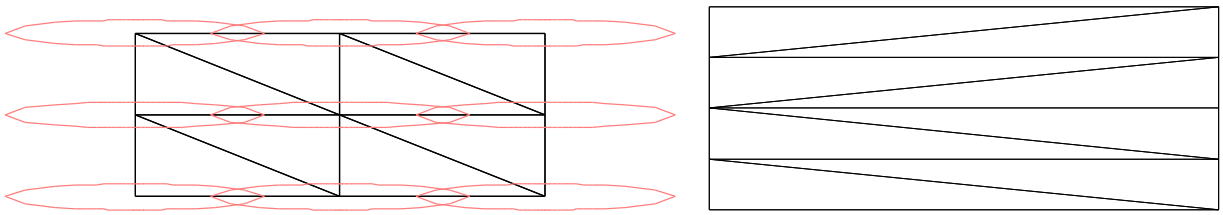


Figure 6.1.1: A metric, $\mathcal{M} \in \mathbb{R}^{d\times d}$ is a symmetric positive definite tensor field that provides a "yardstick" for measuring distances in different directions. In two dimensions, the set of points equidistant to a point O under the metric measure defines an ellipse.

A mesh that *conforms* to a metric field is one in which each edge has the same length, to some tolerance, when measured with the metric according to Equation 6.1.1, which can be integrated to obtain the distance between points that are not infinitesimally close together. An example of a two-dimensional metric-conforming mesher is the Bi-dimensional Anisotropic Mesh Generator (BAMG) [22].

BAMG generates a mesh given a metric field, which is specified at nodes of a background mesh – the current mesh in an adaptive setting. The optimization will determine *changes* to the current, mesh-implied, metric, $\mathcal{M}_0(\vec{x})$, which is obtained on each simplex element of the background mesh by solving a linear system for the $d(d+1)/2$ independent entries of $\mathcal{M}_0(\vec{x})$; the equations in this system enforce that each of the $d(d+1)/2$ edges has unit metric measure. The element-based mesh-implied metrics are then averaged to the nodes using an affine-invariant[1] algorithm [115]. Given the desired metric at each node of the background mesh, BAMG generates a new, metric-conforming mesh, as illustrated in Figure 6.1.2.



(a) Input: background mesh and desired metric

(b) Output: Metric-conforming mesh

Figure 6.1.2: Input and output to/from the Bi-dimensional Anisotropic Mesh Generator [22]. The desired metric on the background mesh is visualized via ellipses at each node.

Affine-invariant changes to the metric field are made via a symmetric *step matrix*, $\mathcal{S} \in$

[1]For example, just averaging matrix entries would be coordinate-system dependent and hence not affine invariant.

$\mathbb{R}^{d\times d}$, according to

$$\mathcal{M} = \mathcal{M}_0^{\frac{1}{2}} \exp(\mathcal{S}) \mathcal{M}_0^{\frac{1}{2}}. \tag{6.1.2}$$

Note that $\mathcal{S} = 0$ leaves the metric unchanged, while diagonal values in $\mathcal{S}$ of $\pm 2 \log 2$ halve/-double the metric stretching sizes.

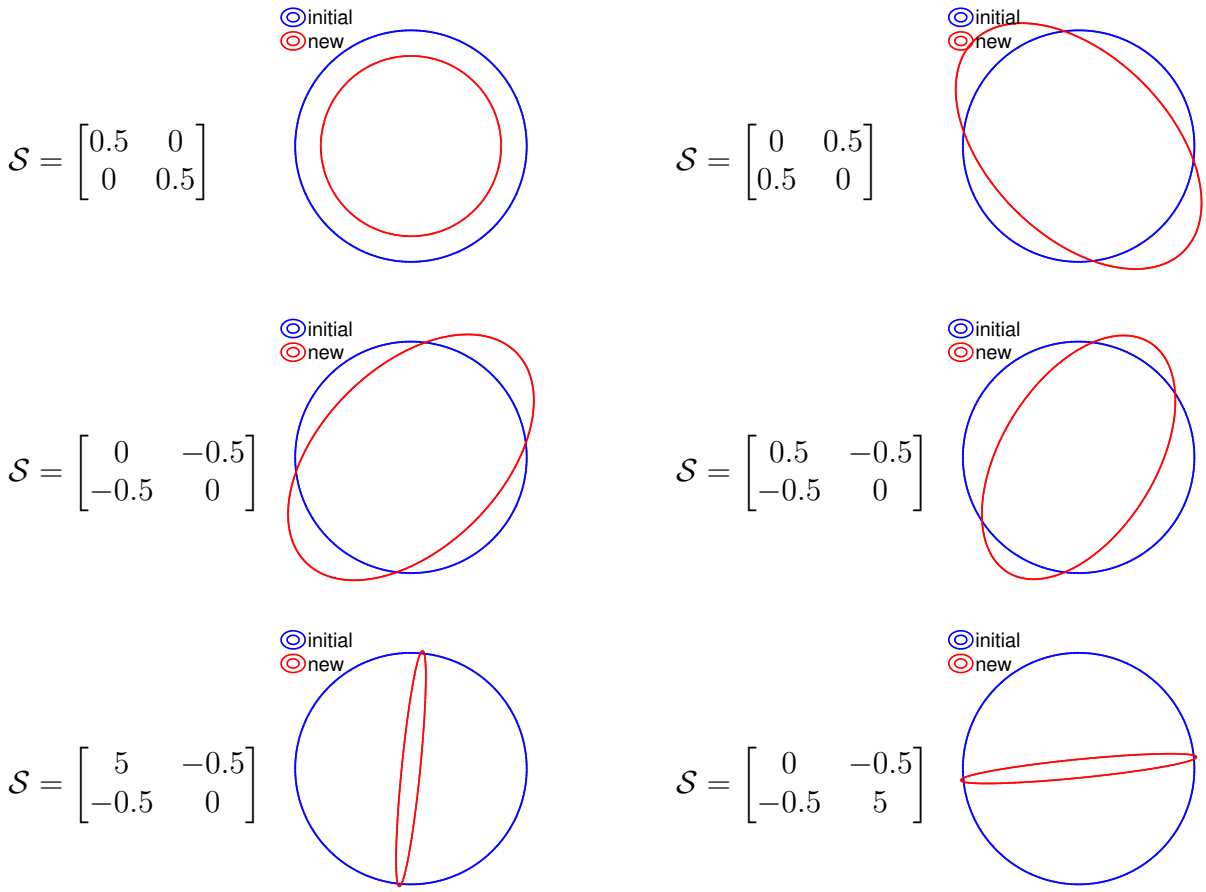


Figure 6.1.3: Examples of the effect of metric modification via Equation 6.1.2 using $\mathbf{M}_0 = \mathcal{I}$ and various step matrices, $\mathcal{S}$. The ellipses show the loci of points of unit metric measure away from the center using $\mathbf{M}_0 = \mathcal{I}$ (blue) and $\mathcal{M} = \mathcal{M}_0^{\frac{1}{2}} \exp(\mathcal{S}) \mathcal{M}_0^{\frac{1}{2}}$ (red).

6.1.2 Error Convergence Model

The mesh optimization algorithm requires a model for how the error changes as the metric changes. Consider one element, Ω_e, with a current error $\mathcal{E}_{e0}$, the absolute value of the element's contribution to the total error, e.g. the adjoint-weighted residual output error in Equation 5.2.7, and a proposed metric step matrix of $\mathcal{S}_e$. What will the new error, $\mathcal{E}_e$, be

over Ω_e following refinement with this step matrix? A typical a priori model may predict

$$\mathcal{E}_e = \mathcal{E}_{e0}\left(\frac{h}{h_0}\right)^r = \mathcal{E}_{e0}\exp\left[r\log(h/h_0)\right], \tag{6.1.3}$$

where h and h_0 are some consistent measures of the new and original element sizes, and r is the error convergence rate. A generalization to anisotropic metric tensors, for which the error may converge differently depending on the direction of change, is that S_e plays the role of $\log(h/h_0)$ and a symmetric *rate tensor* $\mathcal{R}_e$ replaces the scalar r:

$$\mathcal{E}_e = \mathcal{E}_{e0}\exp\left[\mathrm{tr}(\mathcal{R}_e\mathcal{S}_e)\right] \quad \Rightarrow \quad \frac{\partial\mathcal{E}_e}{\partial\mathcal{S}_e} = \mathcal{E}_e\mathcal{R}_e. \tag{6.1.4}$$

Note, we include a linearization with respect to $\mathcal{S}_e$ for use in Section 6.1.4. The total error over the mesh is the sum of the elemental errors,

$$\mathcal{E} = \sum_{e=1}^{N_e}\mathcal{E}_e. \tag{6.1.5}$$

During optimization we will want to keep $\mathcal{E}$ small, and we will be able to change the step matrices at the mesh vertices, $\mathcal{S}_v$. The rate tensor, $\mathcal{R}_e$, will be determined separately for each element through the sampling procedure described in Section 6.2.

6.1.3 Cost Model

To measure the cost of refinement, we use degrees of freedom, `dof`, which on each element just depends on the approximation order p, assumed constant over the elements. By Equation 6.1.2 and properties of the metric tensor, when the step matrix S_e is applied to the metric of element e, the area of the element decreases by $\exp\left[\frac{1}{2}\mathrm{tr}(S_e)\right]$. Equivalently, the number of new elements, and hence degrees of freedom, occupying the original area Ω_e *increases* by this factor. So the elemental cost model is

$$C_e = C_{e0}\underbrace{\exp\left[\frac{1}{2}\mathrm{tr}(\mathcal{S}_e)\right]}_{\mathrm{Area}_0/\mathrm{Area}} \quad \Rightarrow \quad \frac{\partial\mathcal{C}_e}{\partial\mathcal{S}_e} = \mathcal{C}_e\frac{1}{2}\mathcal{I}, \tag{6.1.6}$$

where $C_{e0} = \texttt{dof}_{e0}$ is the current number of degrees of freedom on element e, and $\mathcal{I}$ is the identity tensor. We again include a linearization with respect to $\mathcal{S}_e$ for use in Section 6.1.4. The total cost over the mesh is the sum of the elemental costs,

$$\mathcal{C} = \sum_{e=1}^{N_e} C_e. \tag{6.1.7}$$

6.1.4 Metric Optimization Algorithm

Given a current mesh with its mesh-implied metric ($\mathcal{M}_0(\vec{x})$), elemental error indicators $\mathcal{E}_{e0}$, and elemental rate tensor estimates, $\mathcal{R}_e$, the goal of the metric optimization algorithm is to determine the step matrix field, $\mathcal{S}(\vec{x})$, that minimizes the error at a fixed cost.

The step matrix field is approximated by values at the mesh vertices, $\mathcal{S}_v$, which are arithmetically-averaged to adjacent elements[2]:

$$\mathcal{S}_e = \frac{1}{|V_e|} \sum_{v \in V_e} \mathcal{S}_v, \tag{6.1.8}$$

where V_e is the set of vertices ($|V_e|$ is the number of them) adjacent to element e (see Figure 6.1.4). The optimization problem is to determine $\mathcal{S}_v$ such that the total error $\mathcal{E}$ is minimized at a prescribed total cost $\mathcal{C}$.

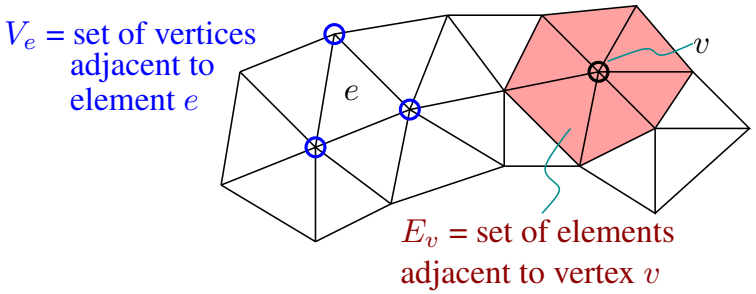


Figure 6.1.4: Definition of vertices surrounding an element and vice versa, used for expanding linearizations of the error and cost functions with respect to the step matrix at the nodes.

First-order optimality conditions require derivatives of the error and cost with respect to $\mathcal{S}_v$. From the equations in Sections 6.1.2 and 6.1.3, and using Equation 6.1.8, these are

$$\frac{\partial \mathcal{E}}{\partial \mathcal{S}_v} = \sum_{e \in E_v} \underbrace{\frac{\partial \mathcal{E}_e}{\partial \mathcal{S}_e}}_{\text{Equation 6.1.4}} \underbrace{\frac{\partial \mathcal{S}_e}{\partial \mathcal{S}_v}}_{1/|V_e|}, \qquad \frac{\partial \mathcal{C}}{\partial \mathcal{S}_v} = \sum_{e \in E_v} \underbrace{\frac{\partial \mathcal{C}_e}{\partial \mathcal{S}_e}}_{\text{Equation 6.1.6}} \underbrace{\frac{\partial \mathcal{S}_e}{\partial \mathcal{S}_v}}_{1/|V_e|}, \tag{6.1.9}$$

where E_v is the set of elements adjacent to vertex v (see Figure 6.1.4). We note that the cost only depends on the *trace* of the step matrix; i.e. the trace-free part of $\mathcal{S}_e$ stretches an element but does not alter its area. We therefore separate the vertex step matrices into trace ($s_v \mathcal{I}$) and trace-free ($\widetilde{\mathcal{S}}_v$) parts,

$$\mathcal{S}_v = s_v \mathcal{I} + \widetilde{\mathcal{S}}_v. \tag{6.1.10}$$

Derivatives of the error with respect to s_v and $\widetilde{\mathcal{S}}_v$ are

$$\frac{\partial \mathcal{E}}{\partial s_v} = \operatorname{tr}\left(\frac{\partial \mathcal{E}}{\partial \mathcal{S}_v}\right), \qquad \frac{\partial \mathcal{E}}{\partial \widetilde{\mathcal{S}}_v} = \frac{\partial \mathcal{E}}{\partial \mathcal{S}_v} - \frac{\partial \mathcal{E}}{\partial s_v} \frac{\mathcal{I}}{d}. \tag{6.1.11}$$

[2]There is no need for an affine-invariant average because entries of $\mathcal{S}$ are coordinate system independent.

The optimization algorithm originally presented by Yano [162] is then:

1. Given a mesh, solution, and adjoint, calculate $\mathcal{E}_e, \mathcal{C}_e, \mathcal{R}_e$ for each element e.
2. Set $\delta s = \delta s_{\max}/n_{\text{step}}$, $\mathcal{S}_v = 0$.
3. Begin loop: $i = 1 \dots n_{\text{step}}$
   (a) Calculate $\mathcal{S}_e$ from Equation 6.1.8, $\frac{\partial \mathcal{E}_e}{\partial \mathcal{S}_e}$ from Equation 6.1.4, and $\frac{\partial \mathcal{C}_e}{\partial \mathcal{S}_e}$ from Equation 6.1.6.
   (b) Calculate derivatives of $\mathcal{E}$ and $\mathcal{C}$ with respect to s_v and $\widetilde{\mathcal{S}}_v$ using Equation 6.1.11.
   (c) At each vertex form the ratio $\lambda_v = \frac{\partial \mathcal{E}/\partial s_v}{\partial \mathcal{C}/\partial s_v}$ and
      - Refine the metric for 30% of the vertices with the largest $|\lambda_v|$: $\mathcal{S}_v = \mathcal{S}_v + \delta s \mathcal{I}$
      - Coarsen the metric for 30% of the vertices with the smallest $|\lambda_v|$: $\mathcal{S}_v = \mathcal{S}_v - \delta s \mathcal{I}$
   (d) Update the trace-free part of S_v to enforce stationarity with respect to shape changes at fixed area: $S_v = S_v + \delta s (\partial \mathcal{E}/\partial \widetilde{\mathcal{S}}_v)/(\partial \mathcal{E}/\partial s_v)$.
   (e) Rescale $S_v \to S_v + \beta \mathcal{I}$, where β is a global constant calculated from Equation 6.1.6 to constrain the total cost to the desired `dof` value: $\beta = \frac{2}{d} \log \frac{\mathcal{C}_{\text{target}}}{\mathcal{C}}$, where $\mathcal{C}_{\text{target}}$ is the target cost.

Note, λ_v is a Lagrange multiplier in the optimization. It is the ratio of the marginal error to marginal cost of a step matrix trace increase (i.e. mesh refinement). The above algorithm iteratively equidistributes λ_v globally so that, at optimum, all elements have the same marginal error to cost ratio. Constant values that work generally well in the above algorithm are $n_{\text{step}} = 20$ and $\delta s_{\max} = 2 \log 2$.

In practice, the mesh optimization and flow/adjoint solution are performed several times at a given target cost, $\mathcal{C}_{\text{target}}$, until the error stops changing. Then the target cost is increased to reduce the error further if desired. Figure 6.1.5 illustrates this process. For reporting the final error and cost at each adaptive iteration, the values are typically averaged over the last few solution iterations at each target cost.

6.2 Element-Local Error Sampling

The mesh optimization algorithm requires that the error convergence rate tensor, $\mathcal{R}_e$ be known for each element e. We estimate this rate tensor a posteriori by *sampling* a small number of element refinements (cuts) for each element and performing a regression. This is also the approach taken by Yano [162], but the approach presented here differs in that we do not solve any primal or adjoint problems on the refined elements. Indeed, we can obtain samples without actually modifying the mesh, and this simplifies data structure handling in the implementation, especially in parallel.

For a triangular element, we consider four refinement options, as shown in Figure 6.2.1. We would like to know how much the error would decrease under each refinement option. One expensive option is to refine the element with the proposed cut, re-solve the primal and fine-space adjoint problems globally, and re-compute the error estimate. Though accurate,

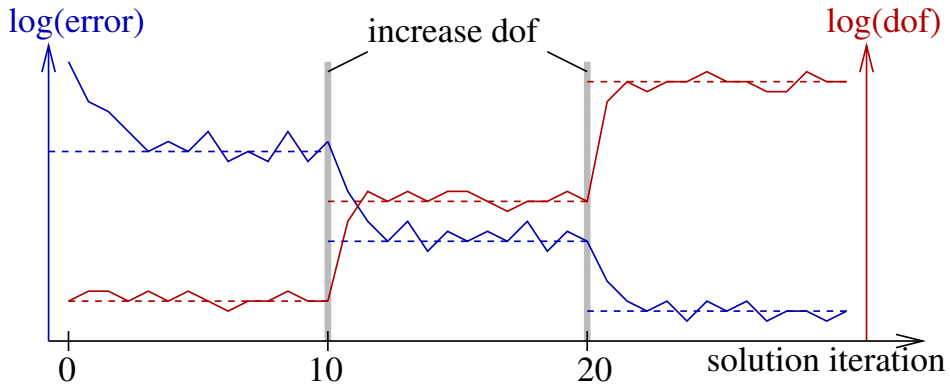


Figure 6.1.5: Illustration of the typical error (left axis) and cost/dof (right axis) history during mesh optimization. At each solution iteration, the primal and adjoint problems are solved and the mesh optimization algorithm is applied to determine the metric for the mesh for the next solution iteration. The target cost is held fixed for multiple (e.g. 10) solution iterations, during which the error eventually stabilizes as each mesh "hovers" about the optimum. Following an allowable cost increase, the error drops over a few solution/optimization iterations and settles to a lower value. Note that the primal and adjoint solutions can be transferred from one mesh to the next, making the solutions at each iteration much cheaper compared to solving from scratch (e.g. using the freestream state).

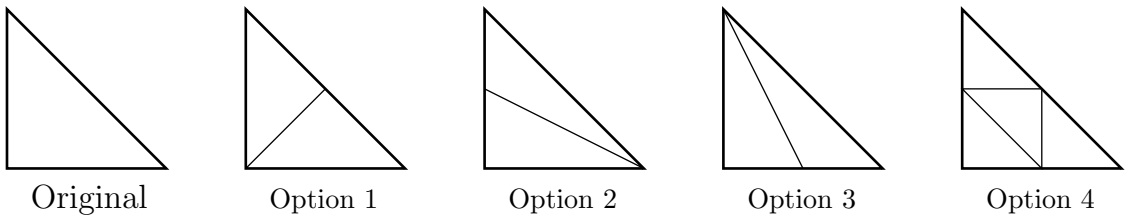


Figure 6.2.1: Four refinement options for a triangle. Each one is considered implicitly during error sampling, though the elements are never actually refined.

this would be impractically expensive. Another option is to only solve the primal/adjoint problems on a subset of the original mesh: the current element and its neighbors. This approach, taken by Yano, is less accurate but still performs very well as globally-exact primal/adjoint states are not necessary to estimate the error rate tensor. Here, we further simplify the estimation by not solving additional problems, even on a local patch of elements. Instead, our proposed algorithm uses element-local *projections* of the fine-space adjoint to semi-refined spaces associated with each refinement option.

Consider one element e. The fine space adjoint $\boldsymbol{\psi}_h^{p+1}\big|_{\Omega_e}$ gives, via Equation 5.2.7, an estimate of the output error in the current order p solution, as measured relative to a "truth" order $p+1$ solution: this is $\mathcal{E}_{e0}$. If the fine-space adjoint were of order $p+2$, we would have an estimate of the error relative to an even finer space. Now, suppose that we are looking at refinement option i in Figure 6.2.1. Refining an element with such an option creates a solution space that is finer than the original, though (we assume) not as fine as increasing the order to $p+1$ – we still use $p+1$ as the "truth" space. Suppose we have an order p adjoint on this refined space; call it $\boldsymbol{\psi}_{hi}^p\big|_{\Omega_e}$, where the i indicates that we are considering refinement option i. With this adjoint we can compute an error indicator $\Delta\mathcal{E}_{ei}$: this estimates the error between the coarse solution and that on refinement option i. The remaining error associated with refinement option i is then given by the difference,

$$\mathcal{E}_{ei} \equiv \mathcal{E}_{e0} - \Delta\mathcal{E}_{ei}. \tag{6.2.1}$$

Calculating $\Delta\mathcal{E}_{ei}$ requires an adjoint-weighted residual evaluation on the element refined under option i. We bypass this complication with an additional simplification: we project $\boldsymbol{\psi}_{hi}^p\big|_{\Omega_e}$ into the $p+1$ space on the original element and evaluate the adjoint weighted residual there. That is, we perform

$$\Delta\mathcal{E}_{ei} \equiv |\mathcal{R}_h^{p+1}(\mathbf{u}_h^p, \widetilde{\boldsymbol{\psi}}_{hi}^p\big|_{\Omega_e})|, \tag{6.2.2}$$

where $\widetilde{\boldsymbol{\psi}}_{hi}^p$ is $\boldsymbol{\psi}_{hi}^p$ projected from order p on refinement option i into order $p+1$ on the original element. The final simplification is to not actually calculate $\boldsymbol{\psi}_{hi}^p\big|_{\Omega_e}$ for refinement option i. Instead, we simply project the known fine-space order $p+1$ adjoint onto order p in refinement option i.

In summary, the error uncovered by refinement option i, $\Delta\mathcal{E}_{ei}$, is estimated by the adjoint-weighted residual in Equation 6.2.2, with all calculations occurring at order $p+1$ on the original element. Figure 6.2.2 breaks down the key computation of $\widetilde{\boldsymbol{\psi}}_{hi}^p$ into individual projections. Using least-squares projections in reference space, the combination of the steps depicted in Figure 6.2.2 can be encapsulated into one transfer matrix that converts $\boldsymbol{\psi}_h^{p+1}$ into $\widetilde{\boldsymbol{\psi}}_{hi}^p$, both represented in the order $p+1$ space on the original element:

$$\widetilde{\boldsymbol{\psi}}_{hi}^p = T_i \boldsymbol{\psi}_h^{p+1}, \tag{6.2.3}$$

$$T_i = \left[M_0(\phi_0^{p+1}, \phi_0^{p+1})\right]^{-1} \sum_{k=1}^{n_i} T_{ik}, \tag{6.2.4}$$

$$T_{ik} = M_k(\phi_0^{p+1}, \phi_k^p) \left[M_k(\phi_k^p, \phi_k^p)\right]^{-1} M_k(\phi_k^p, \phi_k^{p+1}) \tag{6.2.5}$$

$$\left[M_k(\phi_k^{p+1}, \phi_k^{p+1})\right]^{-1} M_k(\phi_k^{p+1}, \phi_0^{p+1}). \tag{6.2.6}$$

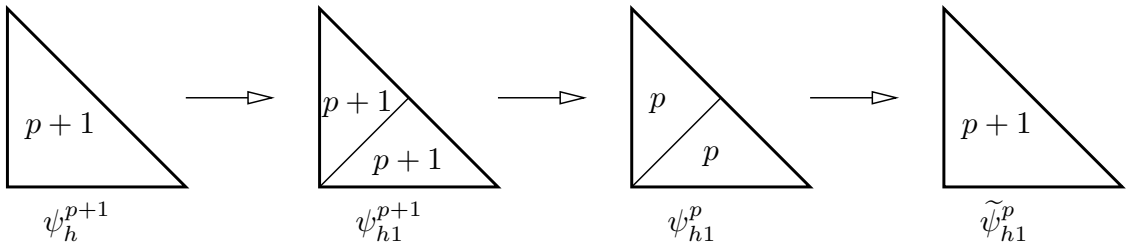


Figure 6.2.2: Projection of the fine-space adjoint, $\boldsymbol{\psi}_h^{p+1}$, down to the space created by refinement option $i = 1$, and then back up to order $p+1$ on the original element. This sequence of projections is encapsulated in a single transfer matrix in Equation 6.2.3.

In these equations, n_i is the number of sub-elements in refinement option i, k is an index over these sub-elements, ϕ_k^p, ϕ_k^{p+1} are order p and $p+1$ basis functions on sub-element k, ϕ_0^p, ϕ_0^{p+1} are order p and $p+1$ basis functions on the original element, and components of the mass-like matrices are defined as

$$M_k(\phi_l, \phi_m) = \int_{\Omega_k} \phi_l \phi_m d\Omega, \qquad M_0(\phi_l, \phi_m) = \int_{\Omega_0} \phi_l \phi_m d\Omega, \tag{6.2.7}$$

where Ω_k is sub-element k and Ω_0 is the original element. Note that the transfer matrix T_i can be calculated for each refinement option i once in reference space and then used for all elements, so that the calculation of $\Delta\mathcal{E}_{ei}$ consumes minimal additional cost – and most importantly, no solutions or residual evaluations are needed on the refined element, as these generally require cumbersome data management and transfer.

Finally, after calculating $\mathcal{E}_{ei}$, the errors remaining after each refinement option i according to Equation 6.2.1, we use least-squares regression to estimate the rate tensor $\mathcal{R}_e$. Note that for triangles, we have 4 refinement options and 3 independent entries in the symmetric $\mathcal{R}_e$ tensor. Using Equation 6.1.4, we formulate the regression to minimize the following error, summed over refinement options,

$$\mathcal{E}_{\text{regression}} = \sum_i \left[\log \frac{\mathcal{E}_{ei}}{\mathcal{E}_{e0}} - \text{tr}(\mathcal{R}_e \mathcal{S}_{ei}) \right]^2. \tag{6.2.8}$$

In this equation, $\mathcal{S}_{ei}$ is the step matrix associated with refinement option i, given by (from Equation 6.1.2),

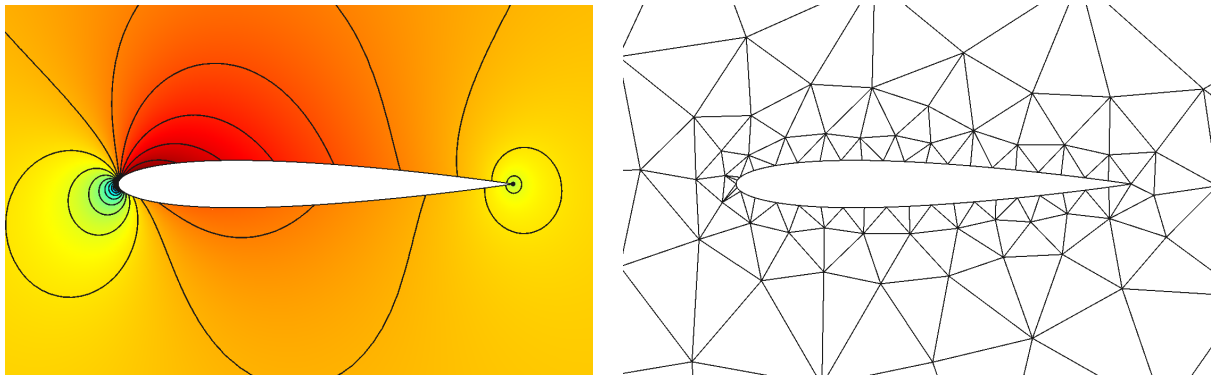
$$\mathcal{S}_{ei} = \log \left(\mathcal{M}_0^{-\frac{1}{2}} \mathcal{M}_i \mathcal{M}_0^{-\frac{1}{2}} \right), \tag{6.2.9}$$

where $\mathcal{M}_i$ is the affine-invariant metric average of the mesh-implied metrics of all sub-elements in refinement option i. Differentiating Equation 6.2.8 with respect to the independent components of $\mathcal{R}_e$ yields a linear system for these components.

6.3 Examples

6.3.1 Inviscid Flow over a NACA 0012 Airfoil

We first consider inviscid flow over a NACA 0012 airfoil at $M_\infty = 0.5$ and $\alpha = 2°$. The output of interest is the drag coefficient. Figure 6.3.1 shows the Mach number contours and the initial coarse mesh for this case. Curved elements of geometry order $q = 4$ are used adjacent to the airfoil to represent the geometry, and the farfield boundary is over 2000 chord-lengths away.



(a) Mach number contours (b) Initial mesh (356 triangles)

Figure 6.3.1: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, inviscid: Mach number contours and the initial coarse mesh for metric-based mesh optimization.

Starting from the coarse mesh, adaptive runs were performed using approximation orders $p = 1, 2, 3$, and at four `dof` targets: 2000, 4000, 8000, and 16000. Note that for a given `dof` target, we expect coarser meshes (fewer elements) at higher p, when each element consumes more `dof`. At each `dof` target, ten solution iterations were performed before moving to the next `dof` target. Figure 6.3.2 shows a sample run using $p = 2$. Note how quickly the error drops in the first ~ 3 iterations after increasing the `dof` target and after starting from the coarse mesh. Since the 356-element initial mesh has 2136 `dof` at $p = 2$, the `dof` plot shows little change in the first ten iterations, when the target is 2000 `dof`, even though the error drops as the ~ 350 elements are resized and repositioned to an optimal configuration for drag prediction.

Figure 6.3.3 compares the convergence of the error in the drag coefficient with a measure of the mesh size, $\texttt{dof}^{-1/2}$, when using the presented mesh optimization algorithm and when using uniform refinement of the initial mesh. The uniform refinement results exhibit stagnating convergence, even at high order, due to the presence of a solution singularity at the trailing edge that limits the attainable rate. Indeed there is little benefit in using $p = 3$ compared to $p = 2$ on the uniform refinement sequence. On the other hand, the adaptive results with the presented optimization algorithm show greatly-improved convergence. Orders of magnitude error reductions are achieved compared to uniform refinement, and there is a

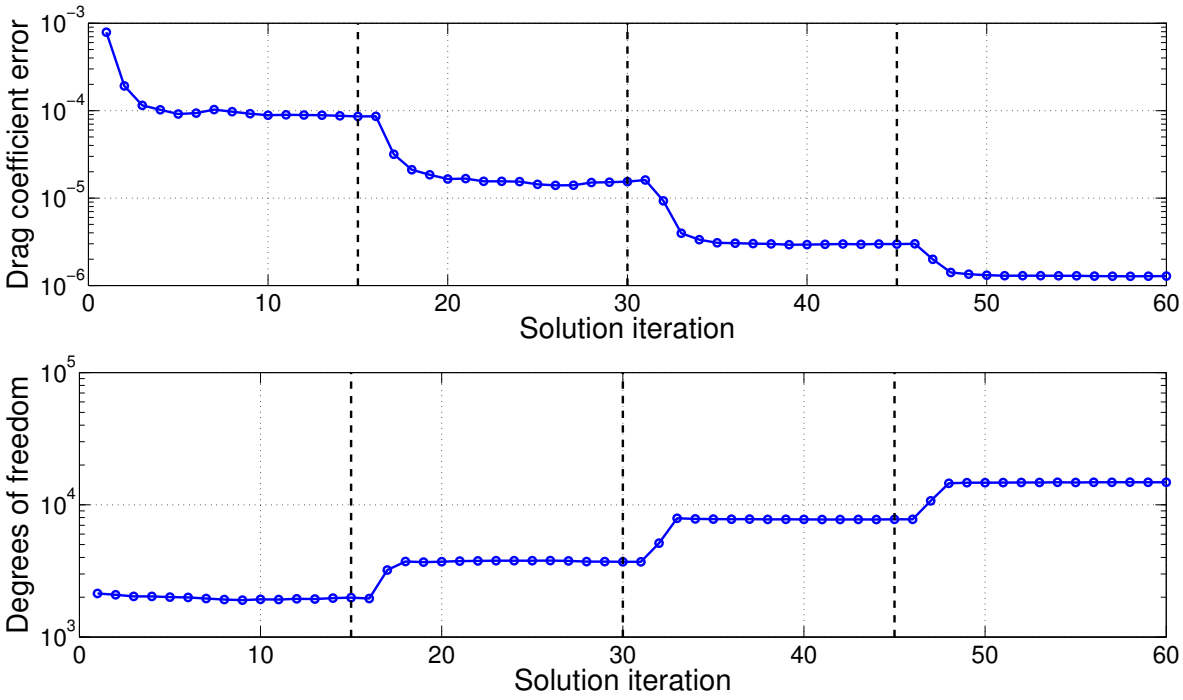


Figure 6.3.2: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, inviscid: output error and degrees of freedom histories for mesh optimization with $p = 2$ approximation and 15 solution iterations at each target `dof` value.

clear benefit to higher-order approximation. Note that Figure 6.3.3 presents data points for every solution iteration in the mesh optimization, with larger symbols denoting the output and `dof` averaged over the five iterations at each `dof` target. As shown, the 2000 initial `dof` target required refinement of the initial mesh for $p = 1$ and coarsening of the initial mesh at $p = 3$; at $p = 2$ the total number of elements remained virtually unchanged at this target.

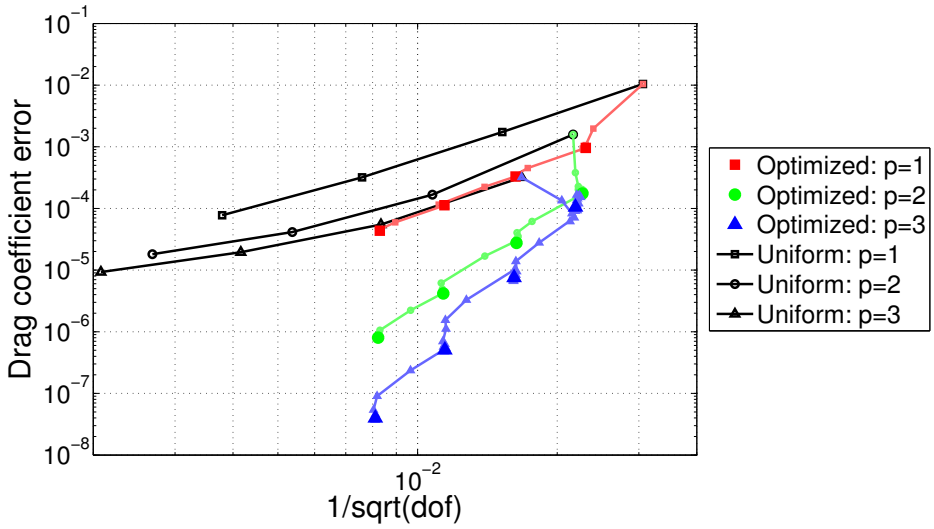


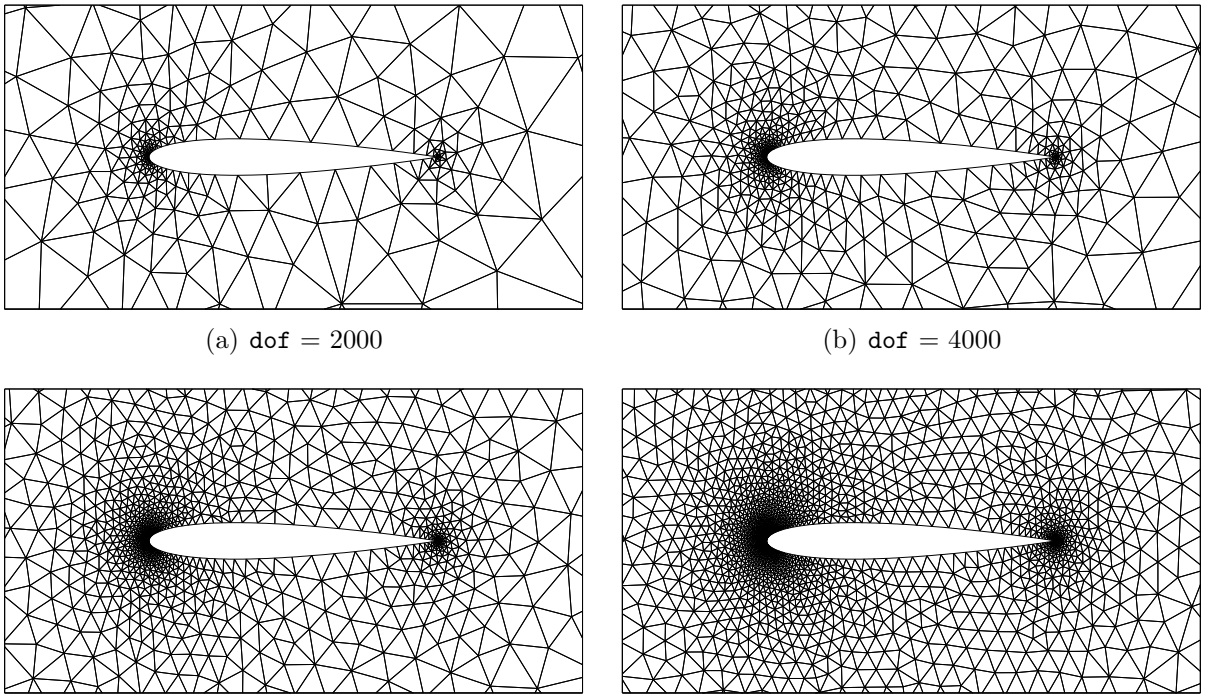
Figure 6.3.3: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, inviscid: comparison of output error convergence with `dof` for uniform refinement and mesh optimization at orders $p = 1, 2, 3$. In the optimization results, the smaller symbols denote the output at each solution iteration, whereas the larger symbols show the average over the last five iterations at each target `dof`.

Figures 6.3.4, 6.3.5, 6.3.6 show the adapted meshes for $p = 1, 2, 3$ approximation, at the four target `dof` values. The differences in the number of elements among the various meshes are evident: the $p = 1$ adapted meshes are the finest, whereas the $p = 3$ ones are the coarsest. The adaptation adds mesh resolution to the most interesting areas in the flowfield: the leading and trailing edges of the airfoil. Especially at the trailing edge, small elements persist even at $p = 3$, in order to resolve the strong singularity.

6.3.2 High-Reynolds Number Flow over a Flat Plate

This case is laminar flow over a flat plate at $M_\infty = 0.5, \alpha = 0, Re_L = 10^6$. The compressible Navier-Stokes equations are solved over the domain shown in Figure 6.3.7, where $L_H = L_V = L = 1$. The Prandtl number is $Pr = 0.72$ and the ratio of specific heats is $\gamma = 1.4$. The output of interest is the drag coefficient on the flat plate.

Figure 6.3.7 also shows the initial mesh used for uniform refinement and mesh optimization runs. This mesh was obtained from subdivision of elements in an initially structured quadrilateral mesh that was generated with extra refinement in the boundary layer and leading-edge of the flat plate. In the uniform refinement runs, each triangular element was



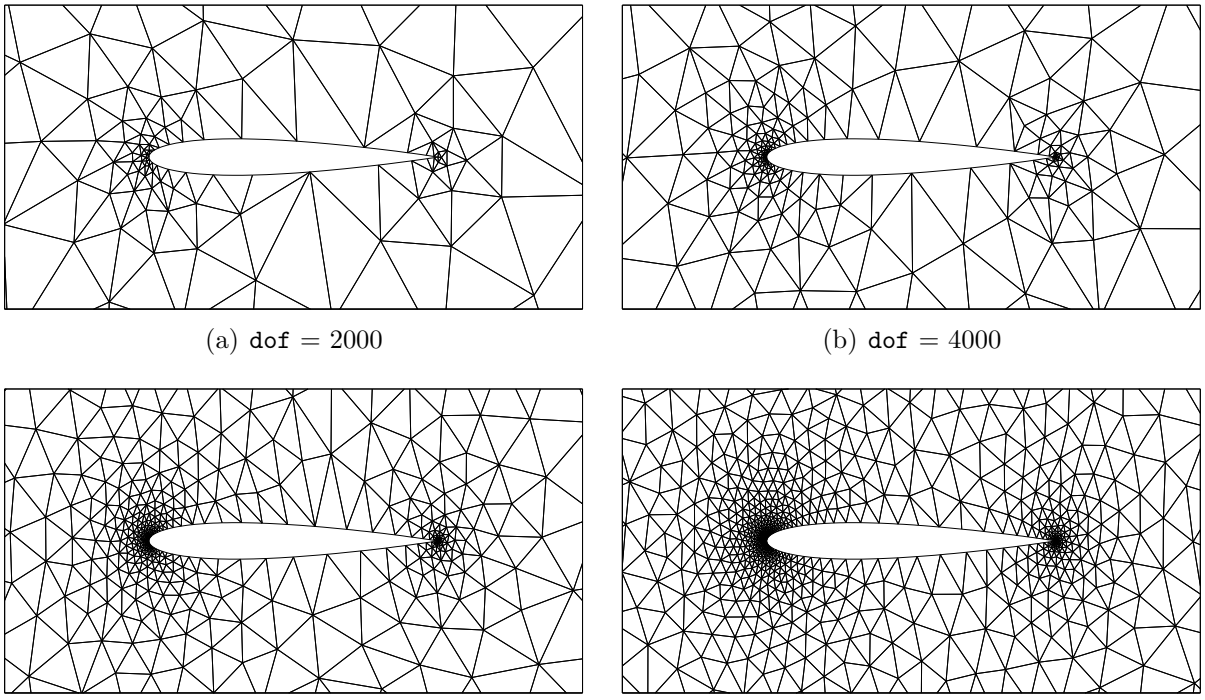
(a) `dof` = 2000

(b) `dof` = 4000

(c) `dof` = 8000

(d) `dof` = 16000

Figure 6.3.4: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, inviscid: optimized meshes (last in sequence for each target `dof`) for $p = 1$.



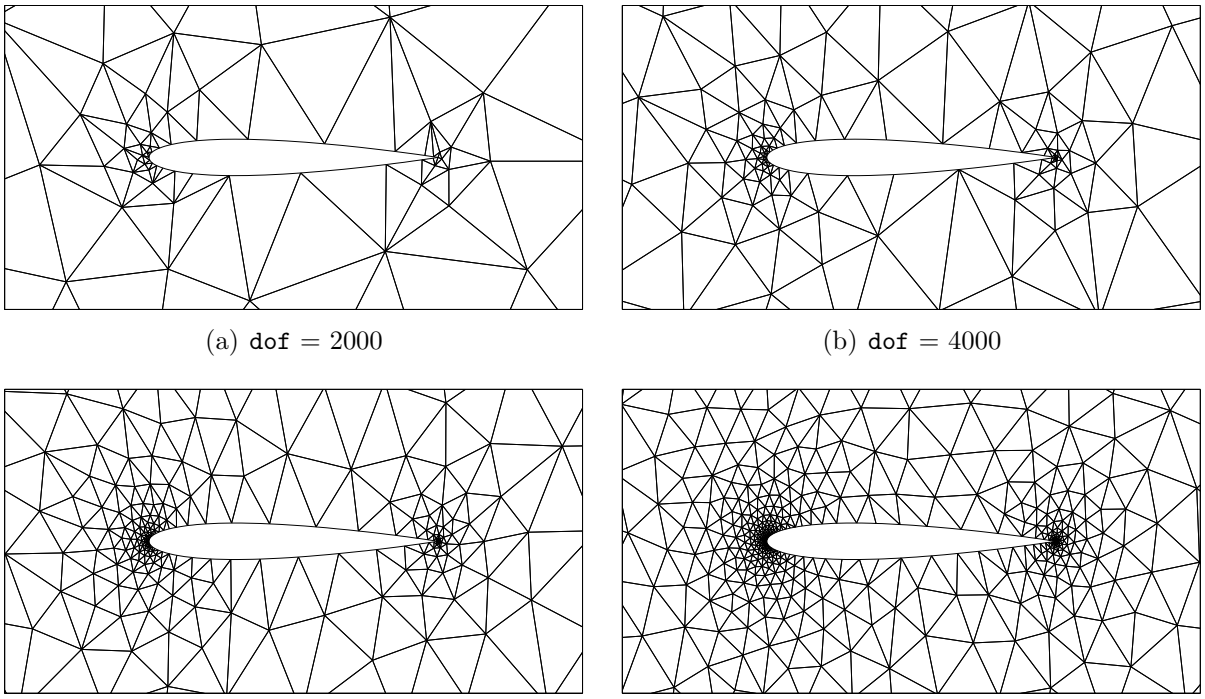
(a) `dof` = 2000

(b) `dof` = 4000

(c) `dof` = 8000

(d) `dof` = 16000

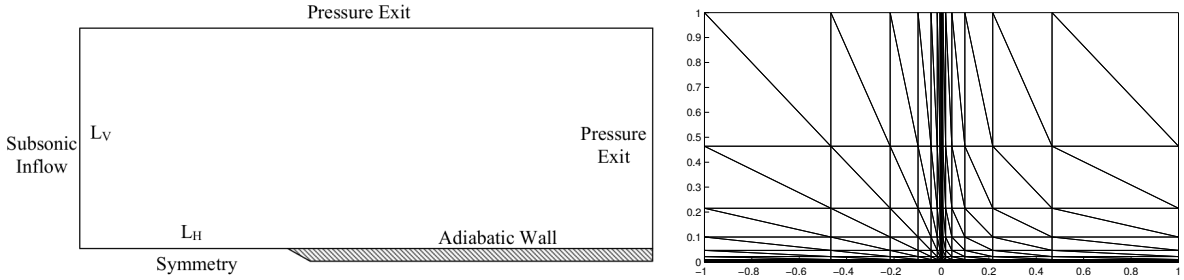
Figure 6.3.5: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, inviscid: optimized meshes (last in sequence for each target `dof`) for $p = 2$.



(a) `dof` = 2000 (b) `dof` = 4000

(c) `dof` = 8000 (d) `dof` = 16000

Figure 6.3.6: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, inviscid: optimized meshes (last in sequence for each target `dof`) for $p = 3$.



(a) Domain and boundary conditions (b) Initial mesh (336 triangles)

Figure 6.3.7: Flat plate, $M_\infty = 0.5$, $Re_L = 10^6$: problem setup and the initial coarse mesh for metric-based mesh optimization.

subdivided isotropically into four new elements to obtain the next mesh. In addition, "quasi-uniform" refinement was considered in which a hand-generated structured fine mesh, also with stretched spacing, was coarsened by removal of every other grid line. The resulting mesh sequence was no longer nested but did a better job at resolving the boundary layer and leading-edge regions.

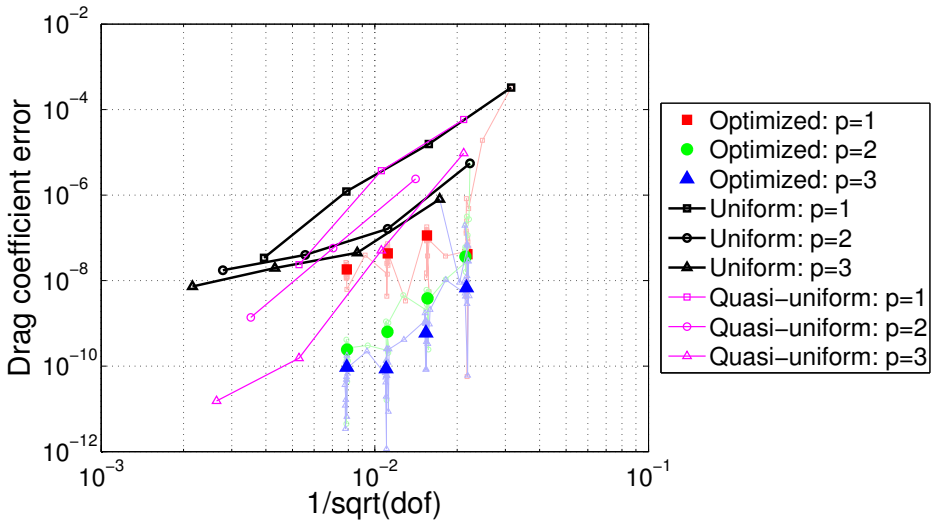


Figure 6.3.8: Flat plate, $M_\infty = 0.5$, $Re_L = 10^6$: comparison of output error convergence with `dof` for uniform refinement, quasi-uniform refinement (coarsening of a grated fine mesh), and mesh optimization at orders $p = 1, 2, 3$. In the optimization results, the smaller symbols denote the output at each solution iteration, whereas the larger symbols show the average over the last five iterations at each target `dof`.

Figure 6.3.8 shows the results of the uniform refinement and adaptive runs for approximation orders $p = 1, 2, 3$. Note that among the uniform refinement runs, increasing the order does decrease the error, but that no higher-order convergence rate is observed due to insufficient resolution of the singular region of the flow (i.e. the leading edge of the flat plate). The quasi-uniform meshes show better results because they are not nested and the finer meshes intentionally have high resolution at the leading edge and in the boundary layer. The meshes optimized using the proposed method show, for a given order, the best results. Note that in this case four cost (`dof`) values were considered: $2000, 4000, 8000, 16000$. For each cost value, fifteen iterations of mesh optimization were performed, and errors from the last five were averaged to produce the large symbols shown in Figure 6.3.8. The stall in convergence around error values of 10^{-10} is likely due to the residual tolerance used (eight orders of magnitude relative decrease).

Figures 6.3.9 and 6.3.10 show the optimized meshes for $p = 2$ and $p = 3$, respectively. The meshes are shown over the entire computational domain and zoomed-in with a stretched vertical axis to see refinement in the boundary layer. Comparing $p = 2$ and $p = 3$, we see coarser meshes for $p = 3$ since the number of degrees of freedom is the same for both orders, and $p = 3$ consumes, with full-order approximation, 10 degrees of freedom compared to 6

for $p = 2$.

From the un-zoomed figures, we see that the optimized meshes show no sign of the initial-mesh refinement pattern: not much resolution is required away from the boundary layer. In the zoomed-in figures we see very high resolution in the leading-edge region, and in an arc-shaped region extending from the leading edge to the trailing edge. The boundary layer of course does not disappear to zero height at the trailing edge, but errors made away from the wall this far down the plate have little impact on the drag. The adaptive algorithm picks up on this. In addition, we see reasonably-high anisotropy of the elements in the boundary layer region – note that the ratio of horizontal to vertical axis scaling is 100-to-1 in the zoomed-in figures. This anisotropy is detected automatically by the error sampling procedure that yields the rate tensor for each element, as described in Section 6.2.

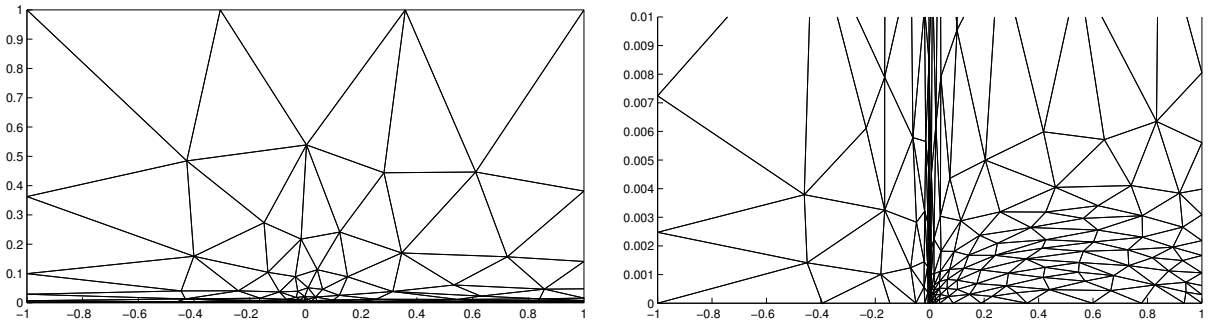
6.3.3 NACA 0012 Airfoil in Laminar Viscous Flow

In this case we consider viscous flow over a NACA 0012 airfoil at $M_\infty = 0.5$, $\alpha = 2°$, and $Re = 5000$. The output of interest is the drag coefficient. Figure 6.3.11 shows the Mach number contours and the initial coarse mesh for this case. Curved elements of geometry order $q = 4$ are used adjacent to the airfoil to represent the geometry, and the farfield boundary is over 2000 chord-lengths away.

Starting from the coarse mesh, adaptive runs were performed using approximation orders $p = 1, 2, 3$ and four `dof` targets: 2000, 4000, 8000, and 16000. Figure 6.3.12 shows drag error and `dof` histories for a sample run using $p = 2$. Compared to the inviscid case the error does not drop as quickly after each `dof` target increase, but it does plateau after 10-15 solution iterations. The output and `dof` values reported are again averages over the last 5 solution iterations at each target `dof`.

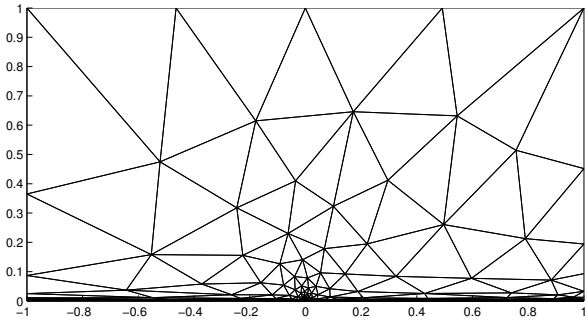
Figure 6.3.13 compares convergence of the error in the drag coefficient with a measure of the mesh size, $\texttt{dof}^{-1/2}$, when using the presented mesh optimization algorithm, when using uniform refinement of the initial mesh, and when using an alternative adaptation algorithm: one in which the output error indicator is used to set the mesh size while the anisotropy comes from the Hessian matrix associated with the Mach number [147, 41]. As in the inviscid case, the uniform refinement results exhibit stagnating convergence, even at high order, due to the presence of singular solution features – there is little benefit to high order in this scenario. On the other hand, the adaptive results show greatly-improved convergence, especially with the presented optimization algorithm. Using the Mach number Hessian in lieu of the metric optimization yields reasonable results for $p = 1$ and $p = 2$, but stalls out for $p = 3$. This should not come as a surprise, since: (1) the Hessian matrix approach was derived for linear ($p = 1$) solution approximation; and (2) the use of the Mach number Hessian is a heuristic that is not guaranteed to provide optimal meshes for the prediction of an output.

Figures 6.3.14 and 6.3.15 show the adapted meshes for $p = 2$ and $p = 3$ approximation, respectively, at the four target `dof` values. The adaptation algorithm adds resolution in the boundary layer, wake, and in the vicinity of the leading-edge stagnation streamline. The two sets of meshes, one optimized using the error sampling procedure and the other generated using the Mach number Hessian, show resolution of similar regions but with different

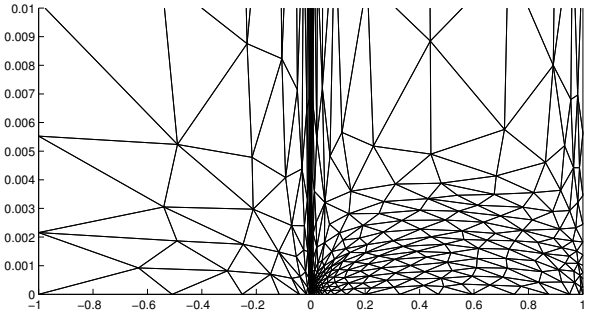


(a) `dof` = 2000

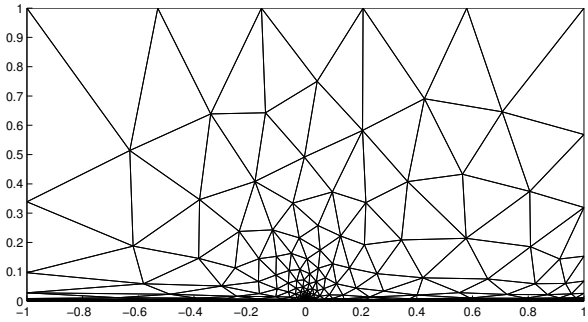
(b) `dof` = 2000 (zoom)



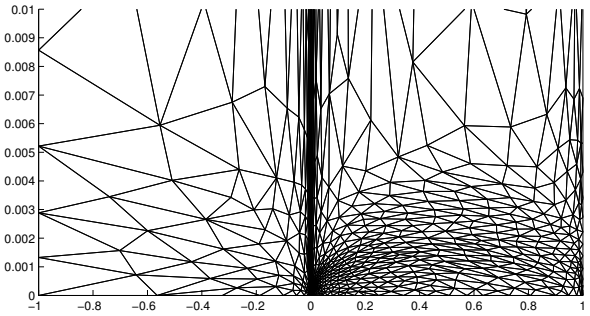
(c) `dof` = 4000



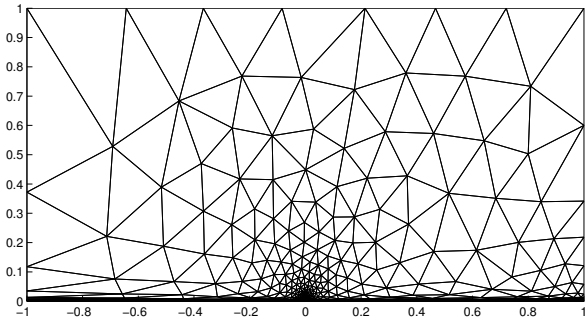
(d) `dof` = 4000 (zoom)



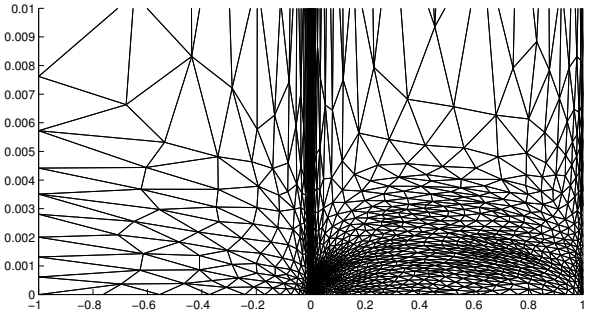
(e) `dof` = 8000



(f) `dof` = 8000 (zoom)

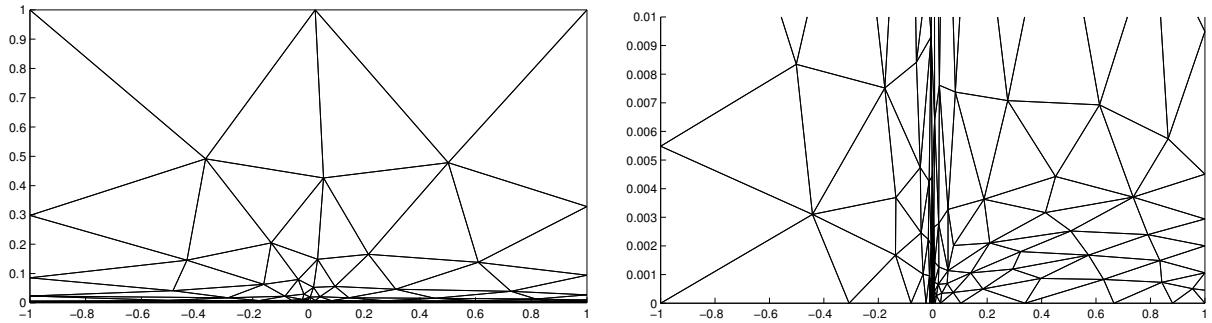


(g) `dof` = 16000



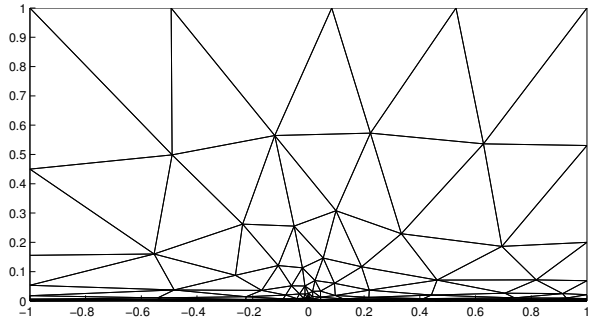
(h) `dof` = 16000 (zoom)

Figure 6.3.9: Flat plate, $M_\infty = 0.5$, $Re_L = 10^6$: optimized meshes (last in sequence for each target `dof`) for $p = 2$. Note the vertical axis stretching in the zoomed plots.

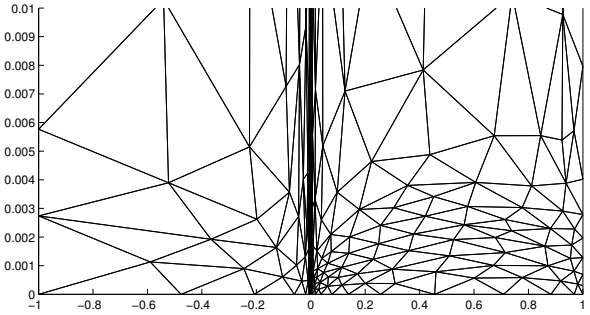


(a) `dof` = 2000

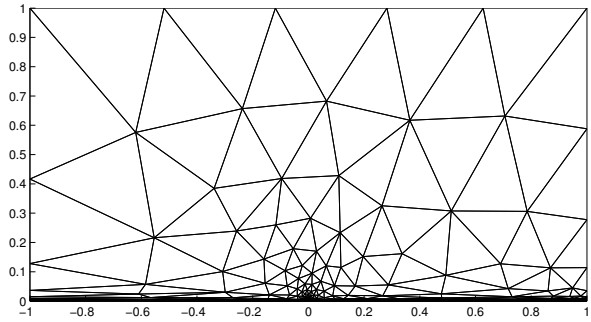
(b) `dof` = 2000 (zoom)



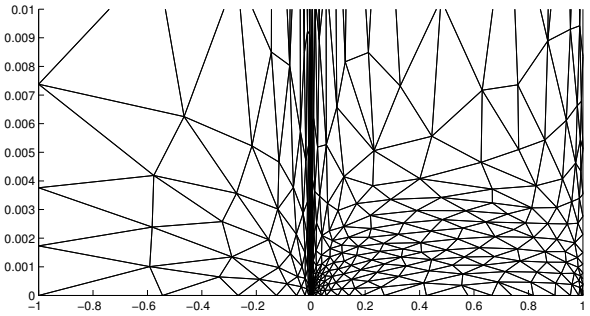
(c) `dof` = 4000



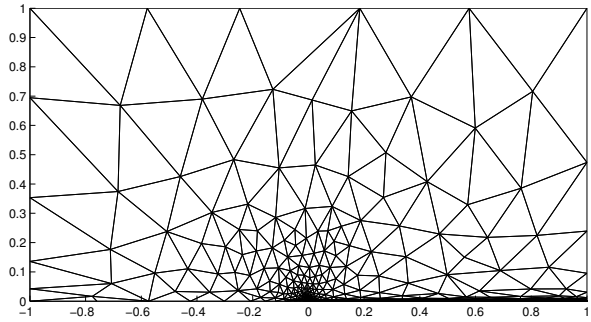
(d) `dof` = 4000 (zoom)



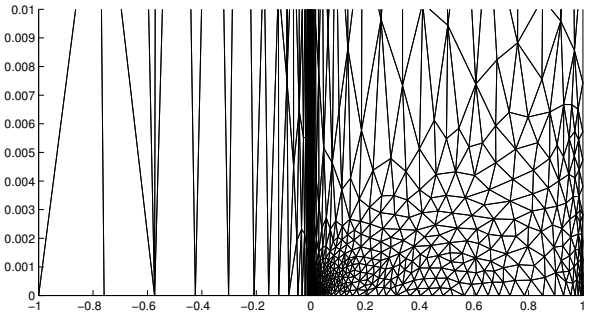
(e) `dof` = 8000



(f) `dof` = 8000 (zoom)

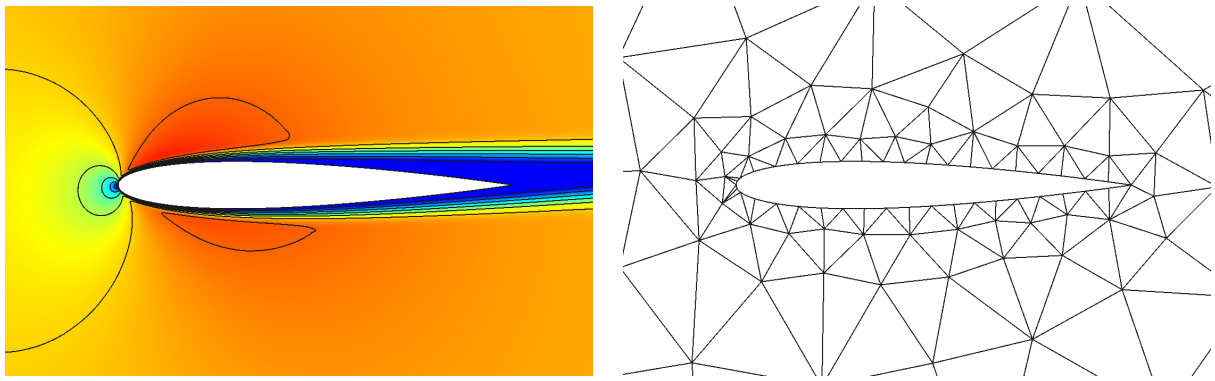


(g) `dof` = 16000



(h) `dof` = 16000 (zoom)

Figure 6.3.10: Flat plate, $M_\infty = 0.5$, $Re_L = 10^6$: optimized meshes (last in sequence for each target `dof`) for $p = 3$. Note the vertical axis stretching in the zoomed plots.



(a) Mach number contours (b) Initial mesh (356 triangles)

Figure 6.3.11: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, $Re = 5000$: Mach number contours and the initial coarse mesh for metric-based mesh optimization.

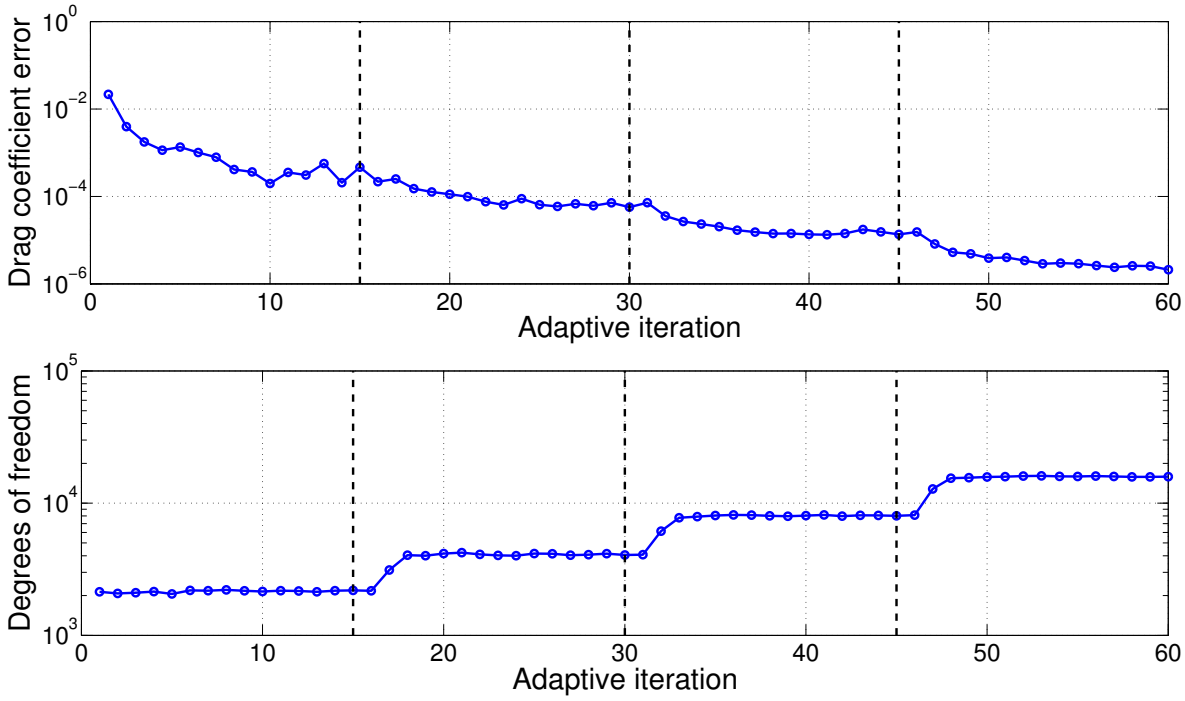


Figure 6.3.12: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, $Re = 5000$: output error and degrees of freedom histories for mesh optimization with $p = 2$ approximation and 15 solution iterations at each target `dof` value.

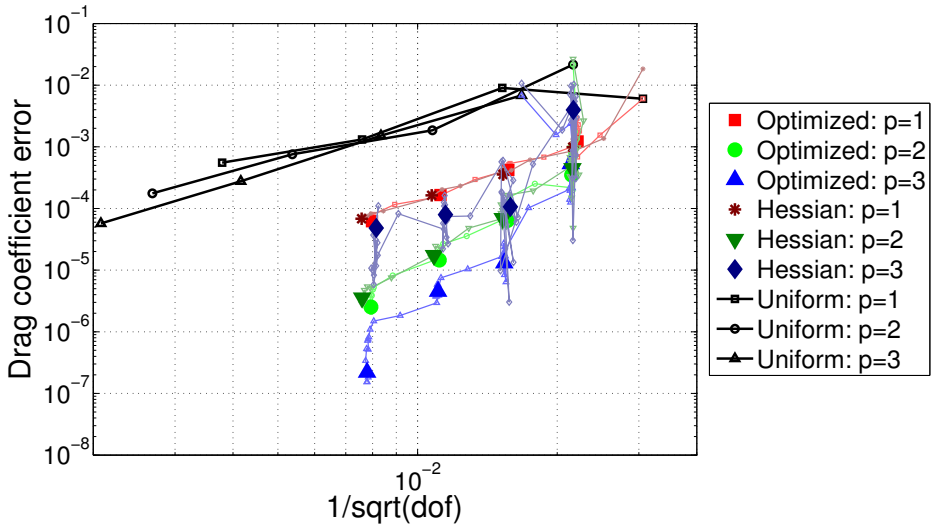


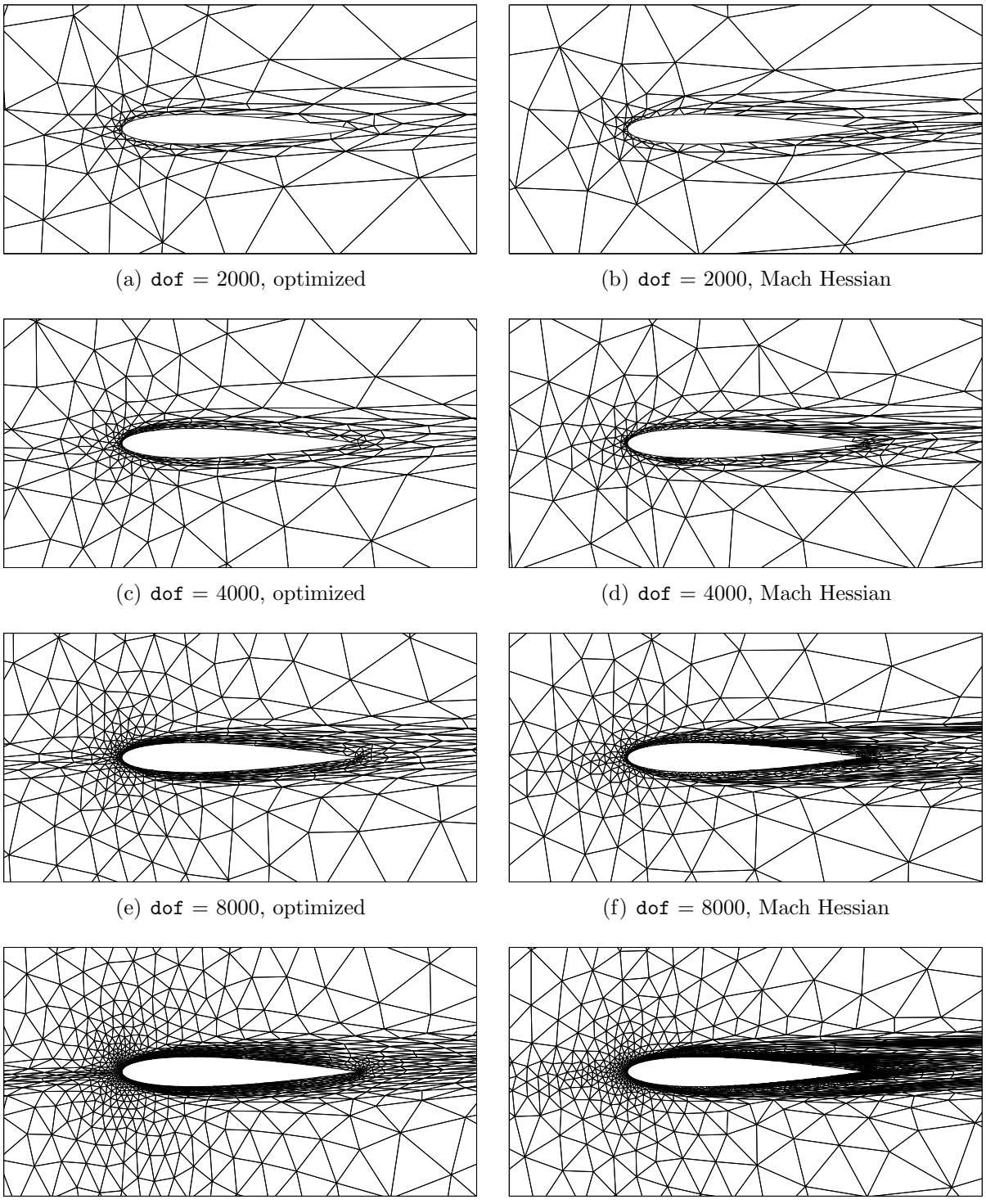
Figure 6.3.13: NACA 0012, $M_\infty = 0.5$, $\alpha = 2°$, $Re = 5000$: comparison of output error convergence with `dof` for uniform refinement, adaptation using the Mach number Hessian to determine anisotropy, and adaptation using the mesh optimization algorithm described in this section. The approximation orders are $p = 1, 2, 3$. In the adaptive results, the smaller symbols denote the output at each solution iteration, whereas the larger symbols show the average over the last five iterations at each target `dof`.

anisotropy. In the boundary layer and in the edges of the wake, using the Mach number Hessian leads to elements with somewhat larger anisotropy. However, on the leading-edge stagnation streamline, the sampling-based meshes exhibit flow-aligned anisotropy, a result of features in the adjoint solution that the Mach Hessian cannot identify. In general, the Mach number Hessian is a reasonable heuristic for identifying anisotropy, but it is not perfect, as the output convergence results show that the sampling-based optimized meshes are more efficient.

6.3.4 RAE 2822 Airfoil in Turbulent Transonic Flow

The final case is an RAE 2822 airfoil in transonic turbulent flow: $M_\infty = 0.734$, $\alpha = 2.79°$, $Re = 6.5 \times 10^6$. The Spalart-Allmaras [5] closure is used to model the turbulent flow, and element-constant artificial viscosity [120] is used to capture shocks. The output of interest is the drag coefficient. Figure 6.3.16 shows the Mach number contours and the initial coarse mesh for this case. Note that the initial mesh is much coarser in the boundary layer than required for accuracy. Curved elements of geometry order $q = 4$ are used adjacent to the airfoil to represent the geometry, and the farfield boundary is over 500 chord-lengths away. A linear elasticity analogy is used to deform anisotropic linear meshes to curve them to the $q = 4$ boundary geometry approximation.

Starting from the coarse mesh, adaptive runs were performed using approximation orders $p = 1, 2, 3$, and at four `dof` targets: 5000, 10000, 20000, and 40000. At each `dof` target, fifteen



(a) `dof` = 2000, optimized

(b) `dof` = 2000, Mach Hessian

(c) `dof` = 4000, optimized

(d) `dof` = 4000, Mach Hessian

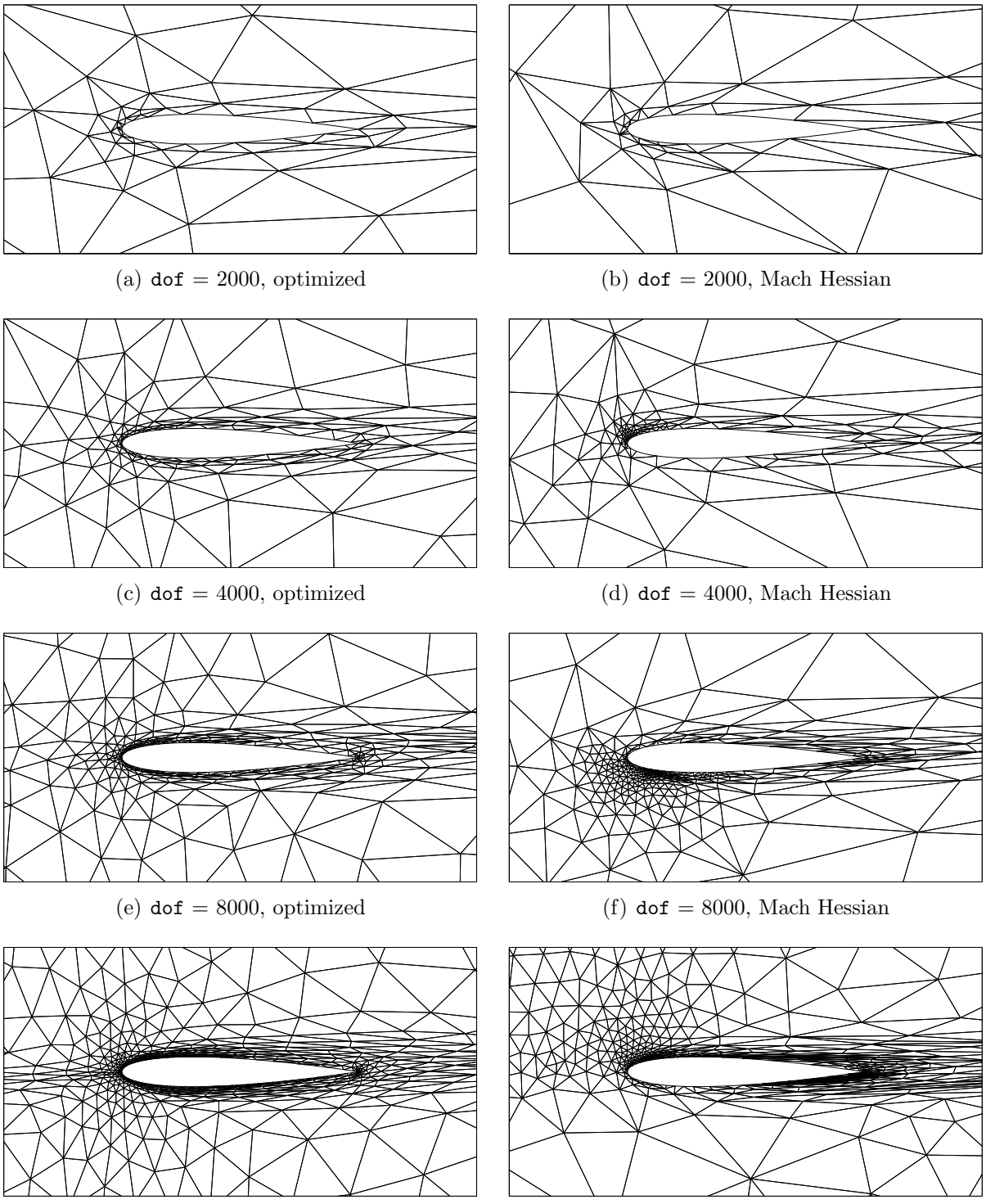
(e) `dof` = 8000, optimized

(f) `dof` = 8000, Mach Hessian

(g) `dof` = 16000, optimized

(h) `dof` = 16000, Mach Hessian

Figure 6.3.14: NACA 0012, $M_\infty = 0.5$, $\alpha = 2^\circ$, $Re = 5000$: adapted meshes (last in sequence for each target `dof`) for $p = 2$ using mesh optimization and, for comparison, the Mach number Hessian to set anisotropy.



(a) `dof` = 2000, optimized

(b) `dof` = 2000, Mach Hessian

(c) `dof` = 4000, optimized

(d) `dof` = 4000, Mach Hessian

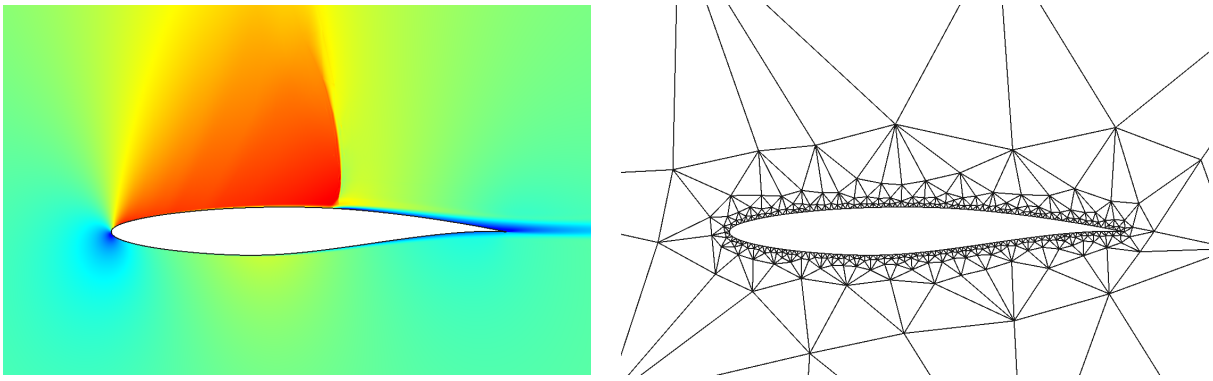
(e) `dof` = 8000, optimized

(f) `dof` = 8000, Mach Hessian

(g) `dof` = 16000, optimized

(h) `dof` = 16000, Mach Hessian

Figure 6.3.15: NACA 0012, $M_\infty = 0.5$, $\alpha = 2^\circ$, $Re = 5000$: adapted meshes (last in sequence for each target `dof`) for $p = 3$ using mesh optimization and, for comparison, the Mach number Hessian to set anisotropy.



(a) Mach number contours

(b) Initial mesh (758 triangles)

Figure 6.3.16: RAE 2822, $M_\infty = 0.734$, $\alpha = 2.79°$, $Re = 6.5 \times 10^6$: Mach number contours and the initial coarse mesh for metric-based mesh optimization.

solution iterations were performed before moving to the next `dof` target. Figure 6.3.17 shows a sample run using $p = 2$. The error drops more slowly in this case compared to the inviscid and laminar cases, and the behavior of the error from mesh to mesh is more oscillatory. This is likely due to the highly nonlinear nature of the RANS equations and the sensitivity of the output to anisotropy in the boundary layer.

Figure 6.3.18 shows the convergence of the drag coefficient error with respect to the mesh size, $\texttt{dof}^{-1/2}$, when using the presented mesh optimization algorithm, and when using uniform refinement. Here we see a dramatic benefit from the optimized meshes. Uniform refinement of the coarse mesh does not appear to consistently reduce the error at any of the orders – the refined meshes are still not fine enough to resolve the boundary layer. On the other hand, the adapted meshes show excellent error reduction: almost four orders of magnitude lower error on the finest meshes compared to the error on the initial mesh. The differences in the adapted results among the three orders are minor, with slight efficiency gains for $p = 2$ and $p = 3$ at the lower error levels.

Finally, Figures 6.3.19, 6.3.20, 6.3.21 show the adapted meshes for the different orders and `dof` targets. These meshes are much different from the initial one: we observe high levels of anisotropy in the thin boundary layer adjacent to the airfoil, and in the wake and leading-edge stagnation streamline resolution. We also see some refinement in the foot of the shock on the upper surface, and resolution of the "λ" feature of the adjoint solution above the airfoil. Note that resolution of the entire shock is not needed for accurate drag calculation: the important regions are the foot of the shock and the characteristics that connect to the shock-boundary layer junction.

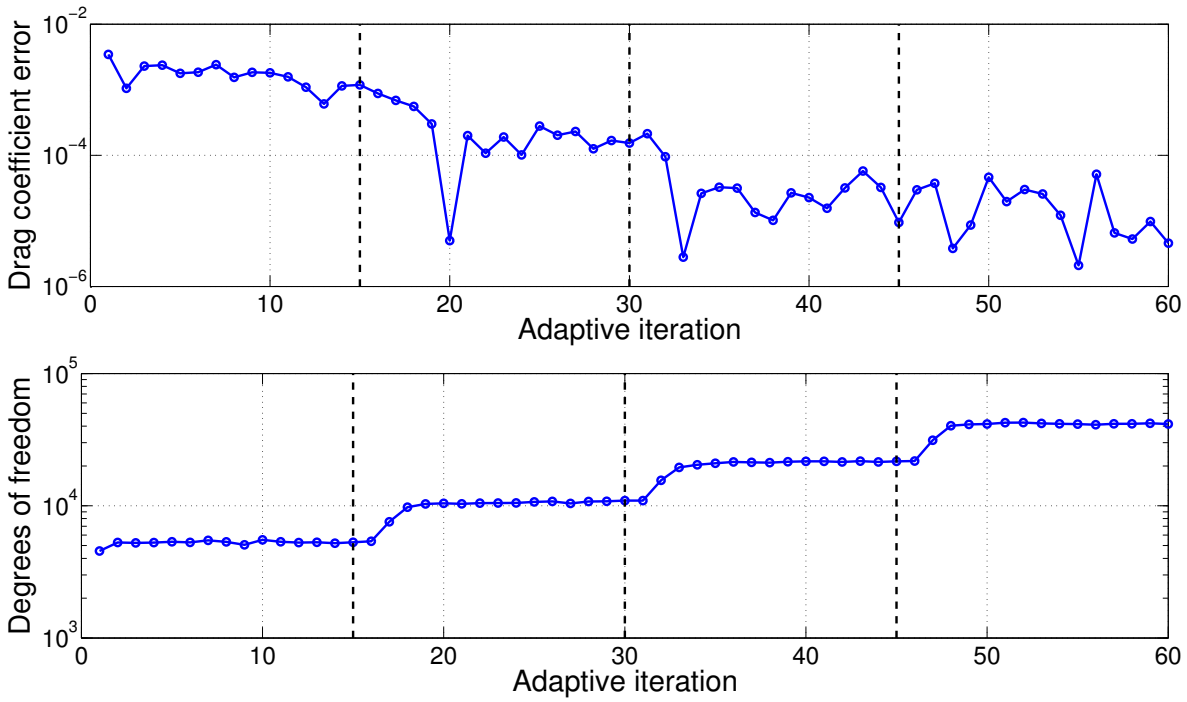


Figure 6.3.17: RAE 2822, $M_\infty = 0.734$, $\alpha = 2.79°$, $Re = 6.5 \times 10^6$: output error and degrees of freedom histories for mesh optimization with $p = 2$ approximation and 15 solution iterations at each target `dof` value.

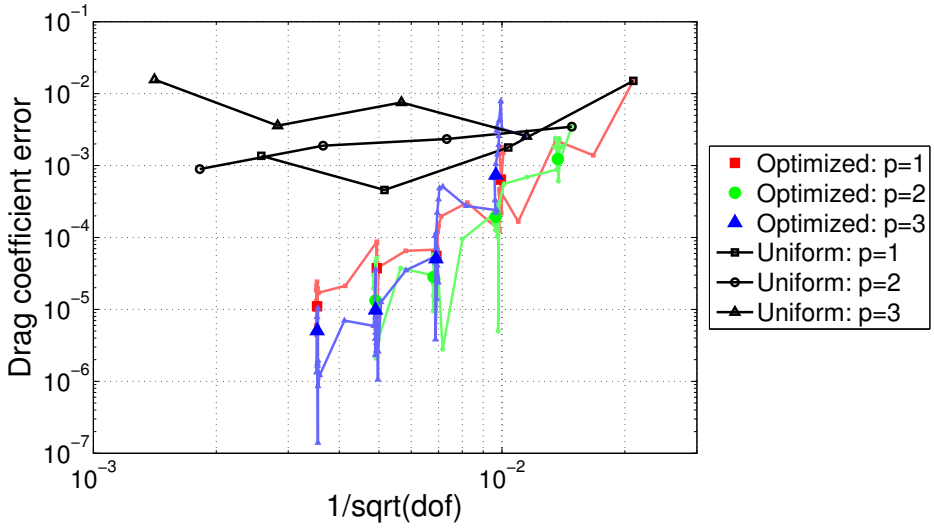
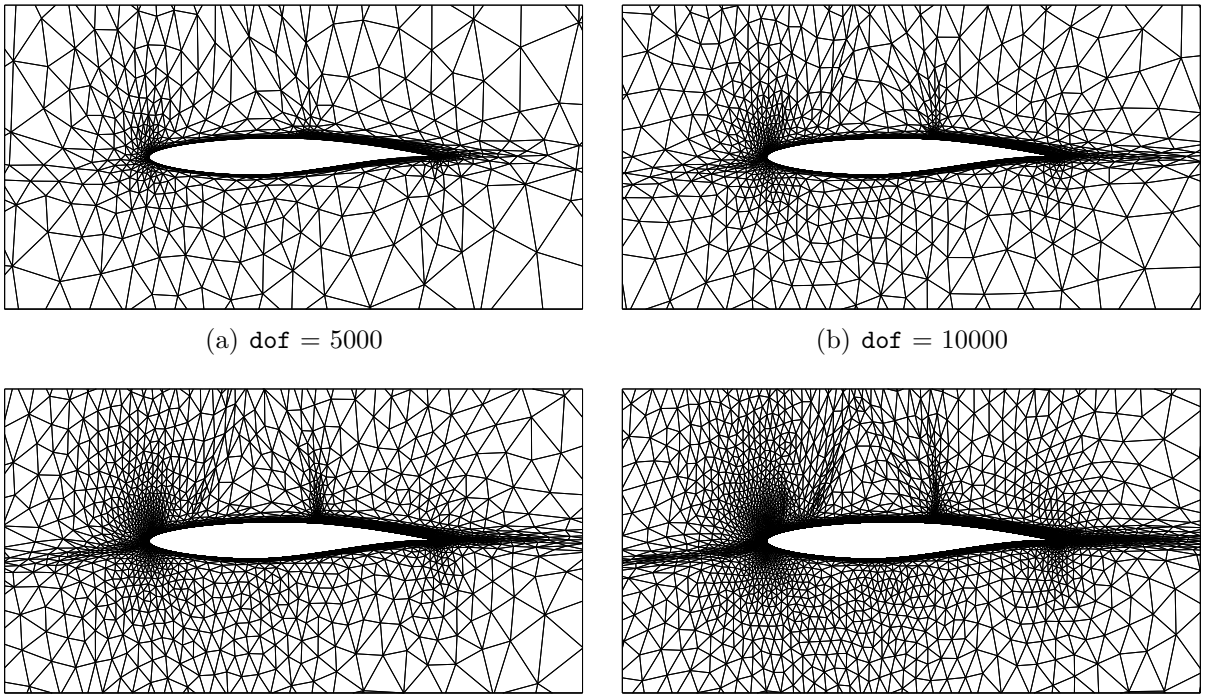


Figure 6.3.18: RAE 2822, $M_\infty = 0.734$, $\alpha = 2.79°$, $Re = 6.5 \times 10^6$: comparison of output error convergence with `dof` for uniform refinement and adaptation using the mesh optimization algorithm described in this section. The approximation orders are $p = 1, 2, 3$. In the adaptive results, the smaller symbols denote the output at each solution iteration, whereas the larger symbols show the average over the last five iterations at each target `dof`.



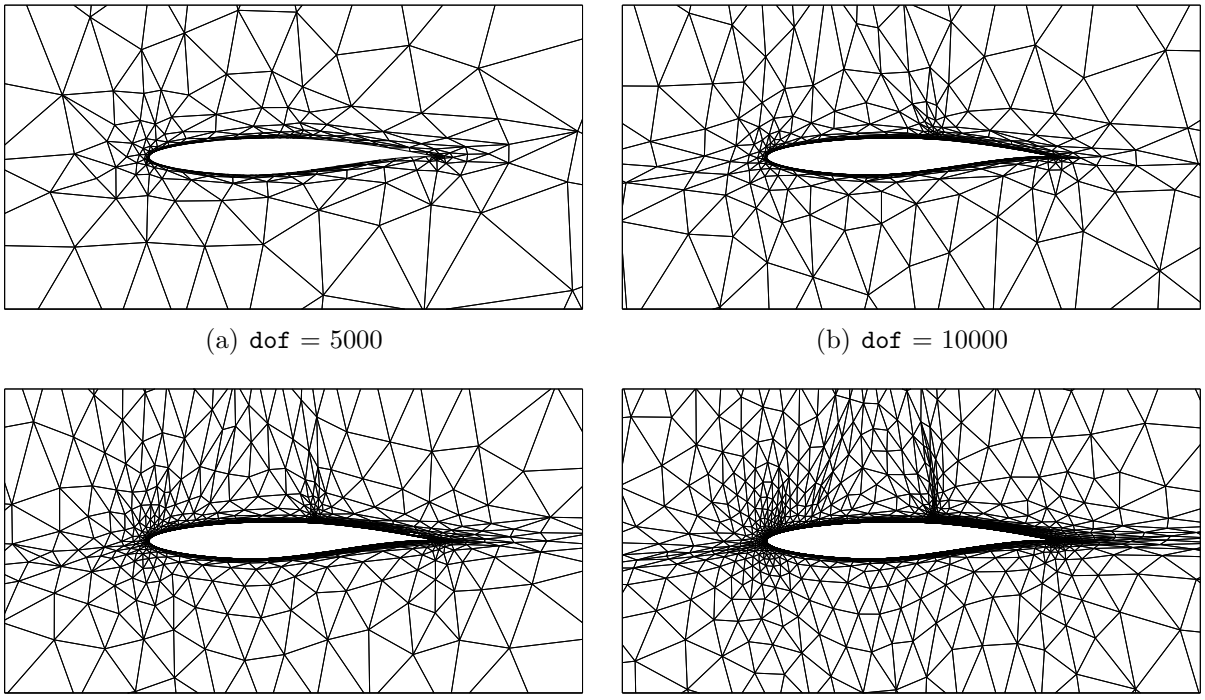
(a) `dof` = 5000

(b) `dof` = 10000

(c) `dof` = 20000

(d) `dof` = 40000

Figure 6.3.19: RAE 2822, $M_\infty = 0.734$, $\alpha = 2.79°$, $Re = 6.5 \times 10^6$: optimized meshes (last in sequence for each target `dof`) for $p = 1$.



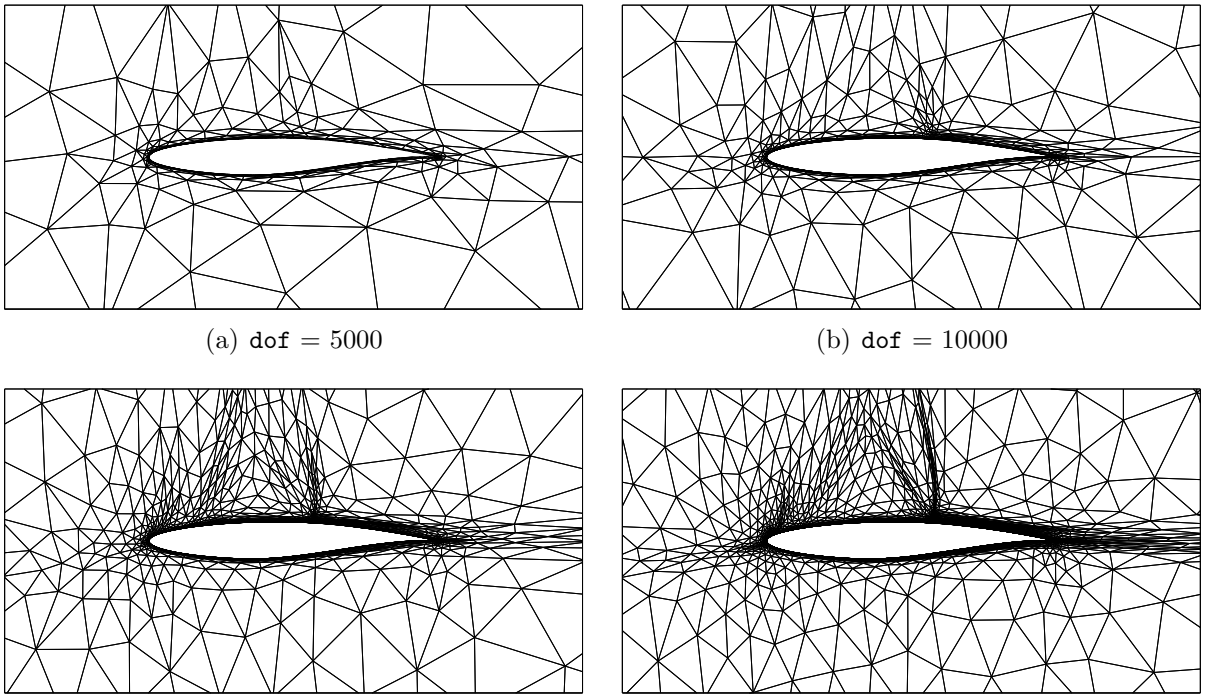
(a) `dof` = 5000

(b) `dof` = 10000

(c) `dof` = 20000

(d) `dof` = 40000

Figure 6.3.20: RAE 2822, $M_\infty = 0.734$, $\alpha = 2.79°$, $Re = 6.5 \times 10^6$: optimized meshes (last in sequence for each target `dof`) for $p = 2$.



(a) `dof` = 5000

(b) `dof` = 10000

(c) `dof` = 20000

(d) `dof` = 40000

Figure 6.3.21: RAE 2822, $M_\infty = 0.734$, $\alpha = 2.79°$, $Re = 6.5 \times 10^6$: optimized meshes (last in sequence for each target `dof`) for $p = 3$.

Chapter 7

Special Topics

7.1 Compressible Navier-Stokes Equations

This section presents, in index notation, the Euler, compressible Navier-Stokes, and Reynolds-averaged compressible Navier-Stokes (RANS) equations. Only the one-equation Spalart-Allmaras model is presented for RANS.

7.1.1 Euler Equations

The Euler equations of gas dynamics are

$$\begin{array}{lcll} \partial_t \rho & + & \partial_i(\rho u_i) & = & 0 \\ \partial_t(\rho u_j) & + & \partial_i(\rho u_i u_j + p\delta_{ij}) & = & 0 \\ \partial_t(\rho E) & + & \partial_i(\rho u_i H) & = & 0 \end{array}$$

where ρ is the density ρu_j is the j^{th} momentum component, and ρE is the total energy. The pressure and total enthalpy are given by

$$\begin{aligned} p &= (\gamma - 1)\left(\rho E - \frac{1}{2}\rho u_k u_k\right), \\ H &= E + \frac{p}{\rho}. \end{aligned}$$

Note, i, j, k index the spatial dimensions and summation is implied over repeated indices. These equations can be written in compact conservation form as

$$\partial_t \mathbf{u} + \partial_i \mathbf{F}_i = \mathbf{0},$$

where

$$\mathbf{u} = \begin{bmatrix} \rho \\ \rho u_j \\ \rho E \end{bmatrix}, \quad \mathbf{F}_i = \begin{bmatrix} \rho u_i \\ \rho u_i u_j + p\delta_{ij} \\ \rho u_i H \end{bmatrix}.$$

7.1.2 Compressible Navier-Stokes

The compressible Navier-Stokes equations are given by

$$\begin{array}{lcll} \partial_t \rho & + & \partial_i(\rho u_i) & = & 0 \\ \partial_t(\rho u_j) & + & \partial_i(\rho u_i u_j + p\delta_{ij}) & = & \partial_i \tau_{ij} \\ \partial_t(\rho E) & + & \partial_i(\rho u_i H) & = & \partial_i(\tau_{ij} u_j - q_i) \end{array}$$

where the viscous stress tensor and the heat flux vector are

$$\begin{aligned} \tau_{ij} &= \mu(\partial_i u_j + \partial_j u_i) + \delta_{ij}\lambda \partial_m u_m, \\ q_i &= -\kappa_T \partial_i T. \end{aligned}$$

Relevant physical quantities for air are,

$$\begin{aligned} \text{Dynamic viscosity:} \quad \mu &= \mu_{\text{ref}} \left(\frac{T}{T_{\text{ref}}}\right)^{1.5} \left(\frac{T_{\text{ref}} + T_{\text{s}}}{T + T_{\text{s}}}\right), \\ &\quad (\text{Sutherland's law: } T_{\text{ref}} = 288.15\text{K},\ T_{\text{s}} = 110\text{K}) \\ \text{Bulk viscosity coefficient:} \quad \lambda &= -\frac{2}{3}\mu, \\ \text{Kinematic viscosity:} \quad \nu &= \frac{\mu}{\rho}, \\ \text{Thermal conductivity:} \quad \kappa_T &= \frac{\gamma \mu R}{(\gamma - 1)Pr}, \\ \text{Specific-heat ratio:} \quad \gamma &= 1.4, \\ \text{Prandtl number:} \quad Pr &= 0.71, \\ \text{Gas constant:} \quad R. & \end{aligned}$$

To clarify a typical viscous implementation, we make the dependence of the viscous flux on the gradient of the state vector explicit by writing the equations in the form

$$\partial_t \mathbf{u} + \partial_i \mathbf{F}_i - \partial_i(\mathbf{K}_{ij} \partial_j \mathbf{u}) = \mathbf{0},$$

where $\mathbf{u} = [\rho, \rho u_j, \rho E]^T$ is the state vector and $\vec{\mathbf{F}}$ is the inviscid flux vector. The viscous flux coefficient matrices can be written as

$$\mathbf{K}_{ij} = \begin{bmatrix} 0 & \mathbf{0} & 0 \\ K_{ij}^{\rho u_d, \rho} & K_{ij}^{\rho u_d, \rho u_c} & \mathbf{0} \\ K_{ij}^{\rho E, \rho} & K_{ij}^{\rho E, \rho u_c} & K^{\rho E, \rho E} \end{bmatrix},$$

where for a given pair i, j,

$$\begin{aligned}
K_{ij}^{\rho u_d,\rho} = (\dim \times 1) \text{ matrix} &= \nu\left(u_d\delta_{ij} - u_i\delta_{jd} + \frac{2}{3}u_j\delta_{id}\right) \\
K_{ij}^{\rho u_d,\rho u_c} = (\dim \times \dim) \text{ matrix} &= \nu\left(\delta_{dc}\delta_{ij} + \delta_{dj}\delta_{ci} - \frac{2}{3}\delta_{di}\delta_{cj}\right) \\
K_{ij}^{\rho E,\rho} = (1 \times 1) \text{ matrix} &= \nu\left(-u_k u_k\delta_{ij} - \frac{1}{3}u_i u_j + \kappa_T T_\rho\right) \\
K_{ij}^{\rho E,\rho u_c} = (1 \times \dim) \text{ matrix} &= \nu\left(-\frac{2}{3}u_i\delta_{cj} + u_j\delta_{ci} + u_c\delta_{ij} + \kappa_T T_{\rho u_c}\right) \\
K_{ij}^{\rho E,\rho E} = (1 \times 1) \text{ matrix} &= \nu\left(\kappa_T T_\rho\right)
\end{aligned}$$

and the temperature derivative vector is

$$T_{\mathbf{u}} = [T_\rho, T_{\rho u_c}, T_{\rho E}] = \frac{\gamma - 1}{R\rho}\left[-E + u_k u_k, -u_c, 1\right].$$

Note that the indices d, c, k index the spatial dimension.

7.1.3 Reynolds-Averaged Compressible Navier-Stokes

An approach to simulating turbulent flows is Reynolds averaging, where the conservative state becomes an averaged state, and fluctuations about this average are not resolved. However, the fluctuations cannot be completely ignored because they do affect the averaged equations through additional stresses. Numerous closure models exist for these stresses, and one popular one in aerospace applications is the Spalart-Allmaras model [139, 5], which requires the concurrent of one additional transport equation for an eddy viscosity variable. The augmented equations are

$$\begin{array}{llllll}
\partial_t\rho & + \partial_j(\rho u_j) & & & = & 0 \\
\partial_t(\rho u_i) & + \partial_j(\rho u_j u_i + p\delta_{ij}) & - \partial_j\tau_{ij} & & = & 0 \\
\partial_t(\rho E) & + \partial_j(\rho u_j H) & - \partial_j(u_i\tau_{ij} - q_j) & & = & 0 \\
\partial_t(\rho\tilde{\nu}) & + \partial_j(\rho u_j\tilde{\nu}) & - \partial_j\left[\frac{1}{\sigma}\rho(\nu + \tilde{\nu}f_n)\partial_j\tilde{\nu}\right] & & = & S_{\tilde{\nu}}
\end{array}$$

where the source term for the turbulence model $\tilde{\nu}$ equation is

$$S_{\tilde{\nu}} = P - D - \frac{1}{\sigma}(\nu + \tilde{\nu}f_n)\partial_j\rho\partial_j\tilde{\nu} + \frac{c_{b2}\rho}{\sigma}\partial_j\tilde{\nu}\partial_j\tilde{\nu}.$$

The Reynolds stress, τ_{ij}, is

$$\tau_{ij} = 2(\mu + \mu_t)\bar{\epsilon}_{ij}, \qquad \bar{\epsilon}_{ij} = \frac{1}{2}(\partial_i u_j + \partial_j u_i) - \frac{1}{3}\partial_k u_k\delta_{ij}$$

μ is the laminar dynamic viscosity and the eddy viscosity, μ_t, is

$$\mu_t = \begin{cases} \rho\tilde{\nu} f_{v1} & \tilde{\nu} \geq 0 \\ 0 & \tilde{\nu} < 0 \end{cases} \qquad f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}, \quad \chi = \frac{\tilde{\nu}}{\nu}$$

The heat flux, q_j, is given by

$$q_j = (k + k_t)\partial_i T, \qquad k = C_p \mu / Pr, \qquad k_t = C_p \mu_t / Pr_t$$

The viscosity that appears in the SA working variable equation is given by $\frac{1}{\sigma}\rho(\nu + f_n\tilde{\nu})$ where f_n is 1 for positive χ and

$$f_n = \frac{c_{n1} + \chi^3}{c_{n1} - \chi^3} \quad \text{when} \quad \chi < 0.$$

The production term, P, is

$$P = \begin{cases} c_{b1}\tilde{S}\rho\tilde{\nu} & \chi \geq 0 \\ c_{b1} S \rho\tilde{\nu} & \chi < 0 \end{cases} \qquad \tilde{S} = \begin{cases} S + \bar{S} & \bar{S} \geq -c_{v2} S \\ S + \dfrac{S(c_{v2}^2 S + c_{v3}\bar{S})}{(c_{v3} - 2c_{v2})S - \bar{S}} & \bar{S} < -c_{v2} S \end{cases}$$

where $S = \sqrt{2\Omega_{ij}\Omega_{ij}}$ is the magnitude of vorticity, and $\Omega_{ij} = (\partial_i v_j - \partial_j v_i)/2$. Also,

$$\bar{S} = \frac{\tilde{\nu} f_{v2}}{\kappa^2 d^2}, \qquad f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}}$$

The destruction term, D, is given by

$$D = \begin{cases} c_{w1} f_w \dfrac{\rho\tilde{\nu}^2}{d^2} & \chi \geq 0 \\ -c_{w1} \dfrac{\rho\tilde{\nu}^2}{d^2} & \chi < 0 \end{cases}$$

where

$$f_w = g\left(\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right)^{1/6}, \qquad g = r + c_{w2}(r^6 - r), \qquad r = \frac{\tilde{\nu}}{\tilde{S}\kappa^2 d^2}$$

The closure coefficients are

$$\begin{array}{rclrcl} c_{b1} & = & 0.1355 & c_{w2} & = & 0.3 \\ \sigma & = & 2/3 & c_{w3} & = & 2 \\ c_{b2} & = & 0.622 & c_{v1} & = & 7.1 \\ \kappa & = & 0.41 & c_{v2} & = & 0.7 \\ c_{w1} & = & \dfrac{c_{b1}}{\kappa^2} + \dfrac{1 + c_{b2}}{\sigma} & c_{v3} & = & 0.9 \\ Pr_t & = & 0.9 & c_{n1} & = & 16 \end{array}$$

In discretizing the $\tilde{\nu}$ viscous term, we have to be aware that we store $\rho\tilde{\nu}$, so for example

$$\partial_j \tilde{\nu} = \partial_j(\rho\tilde{\nu}) - \tilde{\nu}\partial_j \rho$$

The SA working variable, $\tilde{\nu}$, will generally be orders of magnitude smaller than the other state components. Scaling or "non-dimensionalization" of $\tilde{\nu}$ is used to make the stored values of $\tilde{\nu}$ closer to the other state components, which helps during the discrete linear solves and when using finite residual convergence tolerances. Instead of $\tilde{\nu}$ we store $\tilde{\nu}'$, given by

$$\tilde{\nu}' = \frac{\tilde{\nu}}{ND}$$

where ND is a scaling factor, usually on the order of the laminar ν. The associated changes in the above expressions are obtained by re-writing all of them in terms of $\tilde{\nu}'$. In addition, the SA equation is divided by ND. This has the effect of bringing the SA equation residual to the same order as that of the other equation.

7.2 Shock Capturing

7.2.1 Oscillations

The discontinuous Galerkin method, like other high-order methods, suffers from oscillations around discontinuities. The discontinuous approximation space only helps if the shocks are aligned with element edges, which is difficult or not possible for general three-dimensional flows.

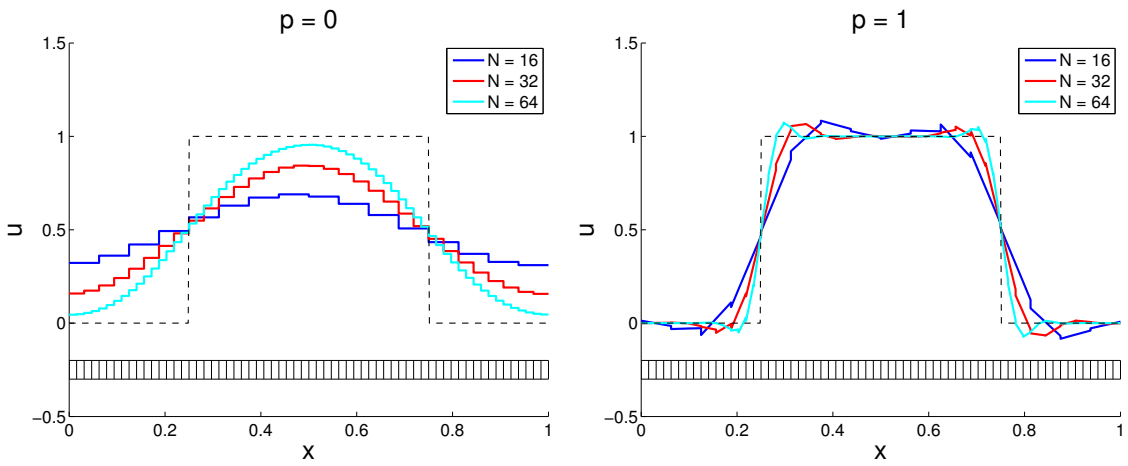
Figure 7.2.1 shows an example of DG solutions to linear advection of a square wave on three meshes, using an upwind flux and periodic boundaries. The $p = 0$ results do not exhibit oscillations, as the scheme is effectively a first-order finite-volume method, which is monotone. However, being first order, $p = 0$ is very dissipative. For higher p, we observe oscillations, and though the solutions improve with increasing order, the oscillations do not disappear. This example demonstrates **Gibbs phenomenon**: the appearance of oscillations when approximating discontinuities using high-order polynomials.

The process by which oscillations are eliminated or mitigated is referred to as **shock capturing**. The same ideas extend other under-resolved features such as boundary layers on coarse meshes. Shock capturing methods can be classified into two general categories:

1. Ones that treat the symptoms by limiting or filtering-out the oscillations

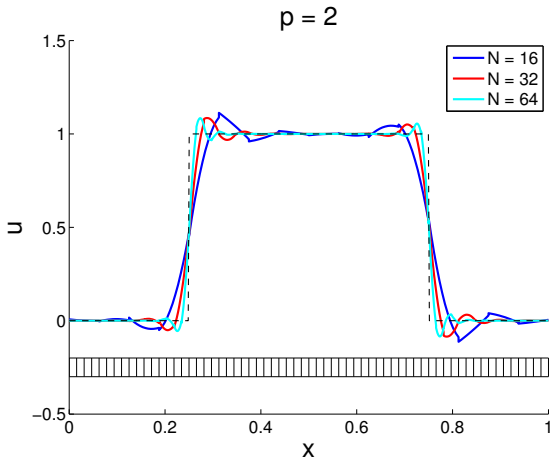
2. Ones that treat the source of the oscillations by modifying the equations

We consider both of these approaches in the following discussion.

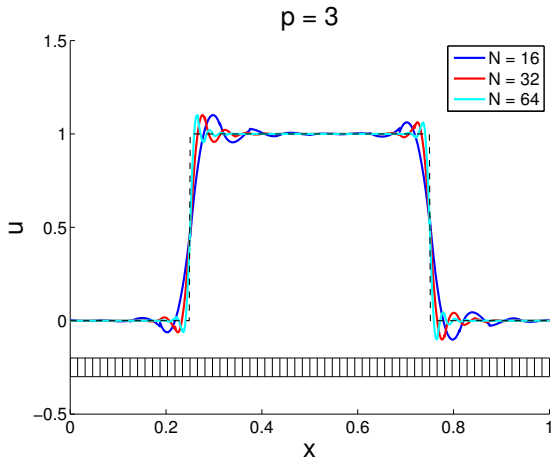


(a) $p = 0$

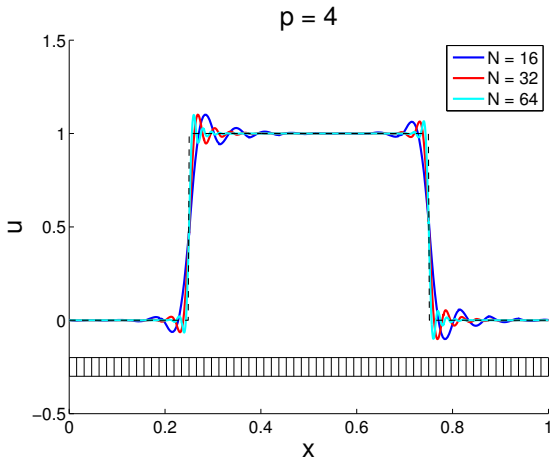
(b) $p = 1$



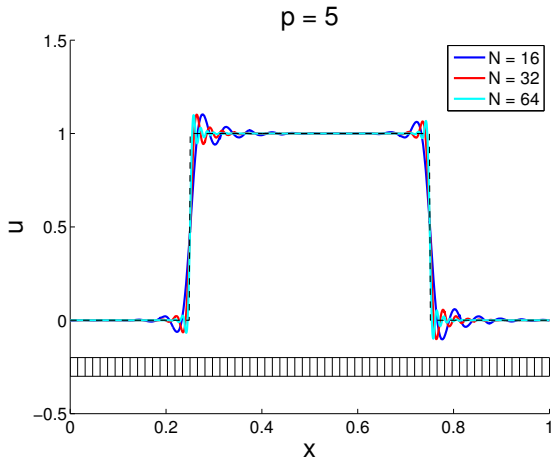
(c) $p = 2$



(d) $p = 3$



(e) $p = 4$



(f) $p = 5$

Figure 7.2.1: DG advection solutions, over one period, for a square-wave initial condition and three mesh resolutions.

7.2.2 Limiting

One method of limiting DG solutions is to apply existing finite-volume techniques to only the $p = 1$ component of the DG approximation. This technique, termed "RKDG", was made popular by Cockburn and Shu [35]. Key elements of this technique are as follows:

- The minmod limiter is used on the $p = 1$ component of the solution; high-order information is neglected.
- The limiter is only applied on "troubled elements", which are identified by oscillations in $p = 1$.
- The limiter is applied at every stage of an explicit multistage time-stepping scheme.

An extension of this technique to higher orders is given by Krivodonova [85] and has the following characteristics:

- It generalizes RKDG to limit more than just the $p = 1$ component.
- The minmod limiter is applied to highest-order derivatives first, followed by the lower-order derivatives.
- Limiting stops once an order is reached for which the limiting is not active.
- For systems of equations, the limiter should be applied to the characteristic variables.

7.2.3 Non-Oscillatory Reconstructions

A method related to limiting is reconstruction. The idea behind non-oscillatory reconstructions is to modify/reconstruct the solution polynomial within each element using information from neighboring elements, as illustrated in Figure 7.2.2. The goal of such a reconstruction should be to reduce any oscillations present in the central element.

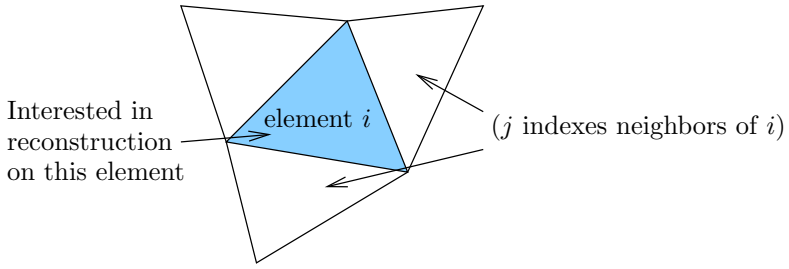


Figure 7.2.2: Solution reconstruction using data from immediate neighbors.

One technique for creating non-oscillatory reconstructions is a hierarchical approach [88]. This consists of the following steps:

1. Start by taking order $p-1$ derivative in element of interest (i) and its neighbors (j) – this will be a linear function.

2. Compute element averages of these order $p-1$ derivatives on elements i and j.
3. Use a standard finite-volume slope reconstruction procedure to obtain a new slope (linear function) on element i.
4. Use the coefficients in this new linear function as "candidates" for order p derivatives in element i.
5. Repeat this process with order $p-2, p-3$, etc. derivatives.
6. Choose the reconstructed high-order derivatives from among multiple candidates at the end of the reconstruction process via limiting.
7. The limiter can be minmod or of the weighted essentially non-oscillatory (WENO) type [157].
8. The reconstruction should only applied on troubled elements.

Another reconstruction approach is based on a direct application of WENO [92]. This method possesses the following characteristics:

- Demonstrated for $p = 1$.
- Polynomial in element i is given as a weighted combination of polynomials reconstructed from various stencils involving i and its neighbors.
- Weights in the combination are determined by an oscillation indicator that involves an integral of first derivatives of the reconstruction.
- Each polynomial is reconstructed to preserve element averages and average derivatives.
- The reconstruction should be used only on troubled elements for efficiency; however, the formal accuracy will not degrade if applied to all elements.

7.2.4 Artificial Viscosity

The method of artificial viscosity adds a diffusion term to the governing equations, where the amount of viscosity is local and depends on how oscillatory the solution is. Some general observations regarding artificial viscosity are as follows:

- All stable numerical schemes add some artificial viscosity. Artificial viscosity is needed because high-order-accurate schemes do not add enough viscosity to stabilize shocks or under-resolved features.
- The diffusion must not be applied to smooth regions to avoid destroying the accuracy.
- If the underlying PDE already contains physical diffusion, artificial viscosity still generally needs to be added. For example the small amount of viscosity in high-Reynolds number Navier-Stokes simulations will not stabilize approximations of shocks on practical mesh resolutions.

The idea of artificial viscosity is not new. It has been used throughout CFD history to stabilize various schemes, starting with the early work of Von Neumann and Richtmyer [102]. The method was made popular for DG by Persson and Peraire [120]. It consists of the following steps:

1. For systems of equations, choose a scalar, u, for detecting discontinuities (e.g. density for Euler).
2. Calculate a non-dimensional smoothness indicator, S_κ, on each element.
   - One option is a very-compact element-interior "resolution" indicator,

$$S_\kappa = R_\kappa = \frac{\int_\kappa (u - \hat{u})^2 \, dx}{\int_\kappa u^2 \, dx}, \tag{7.2.1}$$

     where $\hat{u}$ is a truncated polynomial representation of u of one lower order.
   - Another option is a non-compact "jump" indicator,

$$S_\kappa = J_\kappa = \frac{1}{|\partial\kappa|} \int_{\partial\kappa} \left| \frac{[\![u]\!]}{\{u\}} \right| ds, \quad \text{or} \quad \frac{1}{|\partial\kappa|} \int_{\partial\kappa} \frac{[\![u]\!]^2}{\{u\}^2} \, ds, \tag{7.2.2}$$

     where $|\partial\kappa|$ is the perimeter of the element κ, $[\![u]\!] = u^+ - u^-$ is the inter-element jump, and $\{u\} = (u^+ + u^-)/2$ is the inter-element average.
3. Convert the smoothness indicator to an artificial viscosity using a nonlinear switch.
   - One switch proposed by Persson and Peraire is

$$\epsilon_\kappa = \begin{cases} 0 & \text{if} \quad s_\kappa < \psi_0 - \Delta\psi \\ \frac{\epsilon_0}{2}\left(1 + \sin \frac{\pi(s_\kappa - \psi_0)}{2\Delta\Psi}\right) & \text{if} \quad \psi_0 - \Delta\psi \le s_\kappa \le \psi_0 + \Delta\psi \\ \epsilon_0 & \text{if} \quad s_\kappa > \psi_0 + \Delta\psi \end{cases} \tag{7.2.3}$$

     where $s_e = \log_{10} S_\kappa$, ψ_0 and $\Delta\psi$ are empirically-chosen parameters, and ϵ_0 is a maximum amount of viscosity that scales with h/p – mesh size divided by approximation order.
   - Another, less-nonlinear, switch is

$$\epsilon_\kappa = \epsilon_0 \frac{f^2}{f+1}, \qquad f \equiv \frac{S_\kappa}{S_0}, \tag{7.2.4}$$

     where S_0 is an empirically-defined constant whose role is similar to ψ_0 in Equation 7.2.3, and where ϵ_0 is a viscosity measure that scales as h/p. In Equation 7.2.4, the solution is deemed smooth as long as the smoothness indicator S_κ is well below S_0. If the indicator value is comparable to or greater than S_0, the amount of artificial viscosity added will be roughly proportional to S_κ/S_0.
   - One choice for ϵ_0 in the above two switches is

$$\epsilon_0 = \frac{\bar{\lambda}_{\max} h}{p},$$

     where $\bar{\lambda}_{\max}$ is a local maximum characteristic propagation speed, and h is a local mesh size. For anisotropic grids, h can be made directional by using a metric.

4. The form of the artificial viscosity term added to the PDE is typically Laplacian, which means that the same viscosity is used for all equations. For example, for the Euler equations,

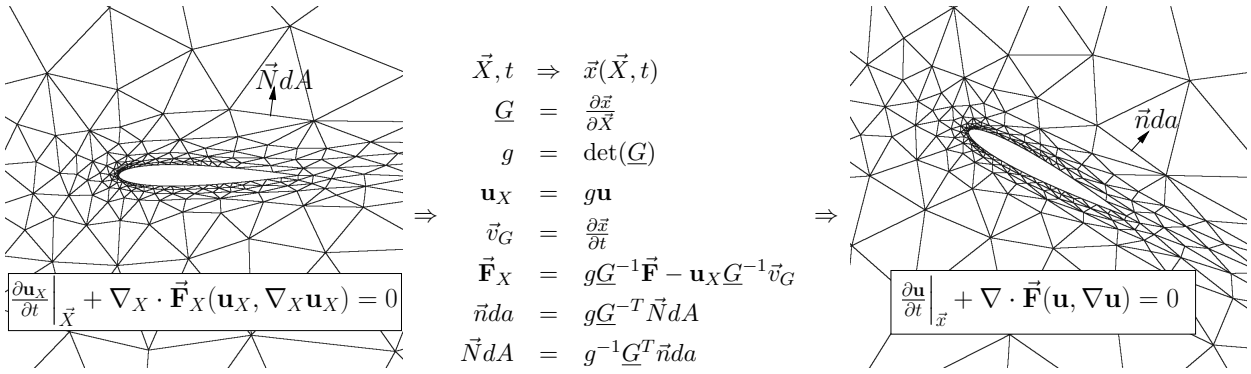
$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}(\mathbf{u}) - \nabla \cdot (\epsilon \nabla \mathbf{u}) = \mathbf{0}.$$

7.3 Mesh Motion

Mesh motion is required when solving problems that involve movement of the geometry, and hence the surrounding mesh. Discretizations need to be modified in order to correctly solve the unsteady PDE in the presence of such motion. One such modification consists of transforming the equations using the arbitrary Lagrangian-Eulerian (ALE) technique.

7.3.1 The Arbitrary Lagrangian-Eulerian Mapping

The idea of ALE is to map the original PDE on a deforming physical domain to a modified PDE on a static reference domain. Figure 7.3.1 illustrates this mapping.



(a) Reference domain: $\vec{X}, \mathbf{u}_X, \vec{\mathbf{F}}_X$ (b) Mapping (c) Physical domain: $\vec{x}, \mathbf{u}, \vec{\mathbf{F}}$

Figure 7.3.1: ALE mapping between the reference and physical domains.

Table 7.3.1 defines the relevant quantities in this mapping. Note that bold indicates a state vector, an arrow indicates a spatial vector, and an underline indicates a spatial matrix.

The expressions for the transformations of the normals are obtained using $dv = gdV$ for infinitesimal volumes and $d\vec{l} = \underline{G}d\vec{L}$ for infinitesimal vectors, as derived in Persson *et al* [119].

Our system of conservation laws (PDE) on the physical domain is

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}(\mathbf{u}, \nabla \mathbf{u}) = \mathbf{0}, \qquad \vec{\mathbf{F}} = \vec{\mathbf{F}}^i(\mathbf{u}) - \vec{\mathbf{F}}^v(\mathbf{u}, \nabla \mathbf{u}), \tag{7.3.1}$$

Table 7.3.1: Key quantities used in the ALE mapping.

$\vec{X}$	=	reference-domain coordinates	da	=	differential area on physical domain
$\vec{x}$	=	physical-domain coordinates	dA	=	differential area on reference domain
$\underline{G}$	=	mapping Jacobian matrix	$\vec{v}_G$	=	grid velocity, $\partial\vec{x}/\partial t$
g	=	determinant of Jacobian matrix	$\mathbf{u}$	=	physical state
$\vec{n}$	=	normal vector on physical domain	$\mathbf{u}_X$	=	state approximated on reference domain
$\vec{N}$	=	normal vector on reference domain	$\vec{\mathbf{F}}$	=	flux vector on physical domain
$v(t)$	=	physical domain (dynamic)	$\vec{\mathbf{F}}_X$	=	flux vector on reference domain
V	=	reference domain (static)			

where both inviscid and viscous fluxes are included. Integrating over a time-varying volume $v(t)$ yields,

$$\int_{v(t)} \frac{\partial \mathbf{u}}{\partial t} dv + \int_{\partial v(t)} \vec{\mathbf{F}} \cdot \vec{n}\, da = 0, \qquad \vec{n} \text{ is outward-pointing on } \partial v(t). \tag{7.3.2}$$

We now transform to the reference domain, V. The boundary integral of the flux is

$$\int_{\partial v(t)} \vec{\mathbf{F}} \cdot \vec{n}\, da = \int_{\partial V} \vec{\mathbf{F}} \cdot (g\underline{G}^{-T}\vec{N})\, dA = \int_{\partial V} (g\underline{G}^{-1}\vec{\mathbf{F}}) \cdot \vec{N}\, dA. \tag{7.3.3}$$

The first integral in Equation 7.3.1 transforms using Leibniz's rule (a.k.a Reynolds' transport theorem in this case),

$$\begin{aligned} \int_{v(t)} \frac{\partial \mathbf{u}}{\partial t} &= \frac{d}{dt}\int_{v(t)} \mathbf{u}\, dv - \int_{\partial v(t)} (\mathbf{u}\vec{v}_G) \cdot \vec{n}\, da \\ &= \frac{d}{dt}\int_V \mathbf{u} g\, dV - \int_{\partial V} (\mathbf{u}\vec{v}_G) \cdot (g\underline{G}^{-T}\vec{N})\, dA \\ &= \int_V \frac{\partial (g\mathbf{u})}{\partial t}\, dV - \int_{\partial V} (g\mathbf{u}\underline{G}^{-1}\vec{v}_G) \cdot \vec{N}\, dA. \end{aligned} \tag{7.3.4}$$

Substituting Equations 7.3.3 and 7.3.4 into Equation 7.3.2 and applying the divergence theorem gives the PDE on the reference domain,

$$\begin{aligned} \frac{\partial \mathbf{u}_X}{\partial t}\bigg|_X + \nabla_X \cdot \vec{\mathbf{F}}_X(\mathbf{u}_X, \nabla_X \mathbf{u}_X) &= 0, \\ \text{where} \quad \mathbf{u}_X &= g\mathbf{u}, \\ \vec{\mathbf{F}}_X &= g\underline{G}^{-1}\vec{\mathbf{F}} - \mathbf{u}_X\underline{G}^{-1}\vec{v}_G. \end{aligned} \tag{7.3.5}$$

∇_X denotes the gradient with respect to the reference coordinates. We break up the transformed flux, $\vec{\mathbf{F}}_X$, into inviscid and viscous fluxes by lumping the grid-velocity term into the inviscid flux,

$$\vec{\mathbf{F}}_X = \vec{\mathbf{F}}^i_X - \vec{\mathbf{F}}^v_X, \qquad \vec{\mathbf{F}}^i_X = g\underline{G}^{-1}\vec{\mathbf{F}}^i - \mathbf{u}_X\underline{G}^{-1}\vec{v}_G, \qquad \vec{\mathbf{F}}^v_X = g\underline{G}^{-1}\vec{\mathbf{F}}^v.$$

The gradient of the state transforms via the chain an product rules. Using implied summation,

$$\nabla \mathbf{u} = \frac{\partial \mathbf{u}}{\partial x_j} = \frac{\partial (g^{-1}\mathbf{u}_X)}{\partial X_d}\frac{\partial X_d}{\partial x_j} = \left(g^{-1}\frac{\partial \mathbf{u}_X}{\partial X_d} - g^{-2}\frac{\partial g}{\partial X_d}\mathbf{u}_X \right) G^{-1}_{dj}$$
$$= g^{-1}\left(\frac{\partial \mathbf{u}_X}{\partial X_d} - g^{-1}\frac{\partial g}{\partial X_d}\mathbf{u}_X \right) G^{-1}_{dj}, \tag{7.3.6}$$

where d and j index the reference and physical coordinates, respectively. We also have,

$$\underline{G} = G_{jd} = \frac{\partial x_j}{\partial X_d}, \qquad \delta_{ji} = \frac{\partial x_j}{\partial x_i} = \frac{\partial x_j}{\partial X_d}\frac{\partial X_d}{\partial x_i} = G_{jd}G^{-1}_{di}, \qquad \Rightarrow \quad \underline{G}^{-1} = G^{-1}_{di} = \frac{\partial X_d}{\partial x_i}.$$

7.3.2 Discretization

In a DG setting, discretization of the new reference-domain equation requires modifications to the numerical flux function on inter-element faces, to the boundary conditions, to the face normal vectors, and to the quadrature integration weights. These modifications are based on the reference-to-global mapping and its derivatives.

The weighted residual statement of Equation 7.3.5 on the reference domain is obtained from the PDE by multiplying by test functions (defined in the reference domain) and integrating over reference-domain elements. The resulting terms in the semilinear form for one reference-domain element, κ, are as follows:

$$\text{(total)} \quad \mathcal{R}_X(\mathbf{u}_X, \mathbf{v}) = \mathcal{R}^u_X(\mathbf{u}_X, \mathbf{v}) + \mathcal{R}^i_X(\mathbf{u}_X, \mathbf{v}) + \mathcal{R}^v_X(\mathbf{u}_X, \mathbf{v})$$

$$\text{(unsteady)} \quad \mathcal{R}^u_X(\mathbf{u}_X, \mathbf{v}) = \int_\kappa \frac{\partial u_{X,k}}{\partial t} v_k dV$$

$$\text{(inviscid)} \quad \mathcal{R}^i_X(\mathbf{u}_X, \mathbf{v}) = -\int_\kappa \partial_{X_d} v_k F^i_{X,dk} dV + \int_{\partial\kappa} v^+_k \widehat{F^i_{X,dk} N_d}\, dA$$

$$\text{(viscous)} \quad \mathcal{R}^v_X(\mathbf{u}_X, \mathbf{v}) = \int_\kappa \partial_{X_d} v_k F^v_{X,dk} dV - \int_{\partial\kappa} v^+_k \widehat{F^v_{X,dk} N_d}\, dA$$

where $\mathbf{v}$ is a test function in the reference domain, d indexes the reference domain spatial coordinates, and k indexes the state vector. The hats indicate numerical fluxes on element interfaces or domain boundaries, and the $^+$ superscript indicates quantities taken from the element interior.

The discretization would be straightforward were it not for the fact that fluxes and boundary conditions are specified on the physical domain. A natural approach that minimizes intrusion into the code is to express the reference-space fluxes and boundary conditions in terms of the physical fluxes and boundary conditions.

Inviscid flux

$$\vec{\mathbf{F}}^i_X = g\underline{G}^{-1}\vec{\mathbf{F}}^i - \mathbf{u}_X\underline{G}^{-1}\vec{v}_G = g\underline{G}^{-1}\left(\vec{\mathbf{F}}^i - \mathbf{u}\vec{v}_G\right). \tag{7.3.7}$$

The inviscid flux includes the standard Galilean transformation expected from changing reference frames and also a multiplication by $g\underline{G}^{-1}$, which is done by post-processing the equation-set specific flux.

On element interfaces, evaluation of the numerical flux $\widehat{F^i_{X,dk}N_d}$ also requires changes. Using

$$\vec{N}dA = g^{-1}\underline{G}^T\vec{n}da \quad \Rightarrow \quad N_d dA = g^{-1}(\underline{G}^T)_{dj}n_j da, \quad \text{and} \quad n_j da = g(\underline{G}^{-T})_{jd}N_d dA,$$

where $(\underline{G}^T)_{dj} = G_{jd}$ and $(\underline{G}^{-T})_{jd} = G^{-1}_{dj}$, we obtain

$$\begin{aligned} F^i_{X,dk}N_d\,dA &= gG^{-1}_{dj}\left(F^i_{jk} - u_k v_{G,j}\right)N_d\,dA \\ &= \left(F^i_{jk} - u_k v_{G,j}\right)gG^{-1}_{dj}N_d\,dA \\ &= \left(F^i_{jk} - u_k v_{G,j}\right)n_j\,da \end{aligned}$$

Without mesh motion the numerical flux calculation returns $\widehat{F^i_{jk}n_j}$. With mesh motion present, the flux has to be modified to operate on $\left(F^i_{jk} - u_k v_{G,j}\right)$ instead of F^i_{jk}. This is a simple but intrusive change because we need to modify equation-set specific functions (i.e. the Riemann solvers) to take as input a grid velocity, $v_{G,j}$.

For example, given two states $\mathbf{u}^L$ and $\mathbf{u}^R$, the Roe flux without mesh motion reads

$$\left[F^i_{jk}n_j\right]^{\text{Roe}} = \frac{1}{2}\left(F^L_{jk}n_j + F^R_{jk}n_j\right) - \frac{1}{2}\left|A_{kl}(\mathbf{u}^{\text{Roe}})\right|(u^R_l - u^L_l).$$

With mesh motion present, the Roe flux becomes

$$\left[(F^i_{jk} - u_k v_{G,j})n_j\right]^{\text{Roe}} = \frac{1}{2}\left(F^L_{jk}n_j + F^R_{jk}n_j\right) - \frac{1}{2}\left(u^L_k + u^R_k\right)u_G - \frac{1}{2}\left|A_{kl}(\mathbf{u}^{\text{Roe}}) - \delta_{kl}u_G\right|(u^R_l - u^L_l),$$

where $u_G = v_{G,j}n_j$ is the component of the grid velocity in the direction of the physical normal $\vec{n}$. The new terms consist of an addition to the flux of the average state multiplied by the mesh velocity, and a shift of the eigenvalues of the linearization about the Roe-average state, $\mathbf{u}^{\text{Roe}}$.

Viscous flux The reference-domain viscous flux is related to the physical viscous flux through

$$F^v_{X,dk} = gG^{-1}_{di}F^v_{ik}. \tag{7.3.8}$$

If the physical viscous flux is calculated using a diffusion matrix and the physical state gradient, $F^v_{ik} = A_{ijkl}\partial_{x_j}u_l$, then, using Equation 7.3.6 for the physical gradient, the reference-domain viscous flux is, using implied summation,

$$\begin{aligned} F^v_{X,dk} &= gG^{-1}_{di}A_{ijkl}\partial_{x_j}u_l \\ &= gG^{-1}_{di}A_{ijkl}g^{-1}\left(\partial_{X_c}u_{X,l} - u_{X,l}g^{-1}\partial_{X_c}g\right)G^{-1}_{cj} \\ &= \underbrace{G^{-1}_{di}A_{ijkl}G^{-1}_{cj}}_{A_{X,dckl}}\left(\partial_{X_c}u_{X,l} - u_{X,l}g^{-1}\partial_{X_c}g\right), \end{aligned} \tag{7.3.9}$$

where c, d index the reference domain coordinates. $A_{X,dckl}$ represents the diffusion matrix on the reference domain. It can be re-written in a more symmetrical form as

$$A_{X,dckl} = G^{-1}_{di} A_{ijkl} G^{-1}_{cj} = G^{-1}_{di} A_{ijkl} G^{-T}_{jc}.$$

7.3.3 Boundary Conditions

The physical convective boundary flux, $\vec{\mathbf{F}}^{ib}$, is modified to account for mesh motion as given in Equation 7.3.7,

$$\vec{\mathbf{F}}^{ib}_X = g\underline{G}^{-1}\left(\vec{\mathbf{F}}^{ib} - \mathbf{u}^b \vec{v}_G\right).$$

We note that the physical boundary flux must be aware of motion on the boundary, $\vec{v}_G$. For example, on a moving wall, the flow tangency boundary condition states that the normal component of the fluid velocity is equal to the normal component of the boundary motion velocity (which would be zero without mesh motion). This physical consideration is separate from the subtraction of $\mathbf{u}^b \vec{v}_G$ above – both must be included.

Calculation of the viscous contribution on a boundary requires not only the boundary state, $\mathbf{u}^b$, but also the boundary flux. For pure Dirichlet boundary conditions, the state gradient information is taken from the interior. In other cases, the physical viscous flux is prescribed on the boundary (e.g. zero heat flux for an adiabatic wall), and in these cases, the viscous flux contribution is added directly to the residual. Let's call Q^b_k the prescribed boundary viscous flux dotted with the physical normal. Then in our residual contribution, we will be integrating,

$$Q^b_k da = F^v_{ik} n_i da = g^{-1} G_{id} F^v_{X,dk}\, g G^{-1}_{ci} N_c dA = F^v_{X,dk} N_d dA.$$

This means that the prescribed boundary viscous flux is the same in both the physical and the reference domains. That is, no transformation needs to be applied to Q^b_k when adding the viscous flux contribution to the residual.

7.3.4 Analytical Mesh Motions

General mesh mapping strategies using blending functions are given in Persson *et al* [119]. The requirement is to prescribe a reference-to-physical mapping at every point. This can be achieved by *blending* a rigid-body motion, such as pitching and plunging for an airfoil, to zero at the farfield. Smooth mappings are desirable, as highly-nonlinear mappings will stress integration requirements for the residual contributions (very high quadrature rules will be required). In addition severely-distorted elements are more likely to be generated for highly-nonlinear mappings, limiting the allowable time step for explicit methods and increasing the possibility of negative element Jacobians.

7.3.5 The Geometric Conservation Law (GCL)

Due to the nonlinear and non-polynomial nature of general mappings, a constant state in the physical domain ($\mathbf{u} = \bar{\mathbf{u}} = \text{const.}$) will generally not be representable using a standard polynomial basis in the reference domain ($\mathbf{u}_X = g\bar{\mathbf{u}}$ will not be a polynomial in X). This means that in an unsteady free-stream test, the free-stream will not be preserved! Persson *et al* [119] describe one technique, a geometric conservation law (GCL), for addressing this problem. This technique relies on approximating (in reference space) $\mathbf{u}_{\bar{X}} = \bar{g}\mathbf{u} = \bar{g}g^{-1}\mathbf{u}_X$ instead of $\mathbf{u}_X$, where $\bar{g}$ is a separate variable approximated using the same basis and marched using the same unsteady solver as the state to solve the following equation:

$$\frac{\partial \bar{g}}{\partial t} - \nabla_X \cdot (g\underline{G}^{-1}\vec{v}_G) = 0.$$

Note that now a constant physical state ($\mathbf{u}_X = g\bar{\mathbf{u}}$) is representable, since $\mathbf{u}_X = \bar{g}\bar{\mathbf{u}}$ is a polynomial in the discrete approximation space.

7.4 Hybridized Discontinuous Galerkin Methods

Motivation The motivation for modifying the standard DG discretization is to reduce the cost of DG by decreasing the number of globally-coupled unknowns. DG uses extra degrees of freedom (DOFs) compared to continuous FEM for a given mesh and approximation order. The discontinuous approximation space in DG provides stability for convection-dominated problems but does not generally improve approximation of the underlying continuous functions. The cost of "duplicating" degrees of freedom at element interfaces is especially pronounced in three dimensions. Hybridized and embedded DG methods, HDG and EDG, have been recently introduced to address this DOF duplication problem [103, 116].

Additional degrees of freedom HDG and EDG methods rely on additional unknowns that approximate the state on interfaces between elements in the mesh. At first this seems counter-intuitive (we are trying to reduce DOFs!), but ultimately these "additional" unknowns will be the *only* unknowns.

Figure 7.4.1 defines key structures and spaces used in the HDG discretization. Note that W_h is the standard DG approximation space. M_h is the new approximation space defined on the mesh faces: elements on either side of a face see the same $\widehat{u}$. In HDG, no further continuity is enforced in M_h. EDG, however, takes one more step and uses $\widetilde{M}_h$ instead of M_h, where $\widetilde{M}_h$ is M_h but with continuity enforced at mesh nodes (and edges in 3D).

The equations Suppose we have a first-order system (e.g. Euler) in conservative form,

$$\nabla \cdot \vec{\mathbf{F}}(\mathbf{u}) = 0. \tag{7.4.1}$$

Define:

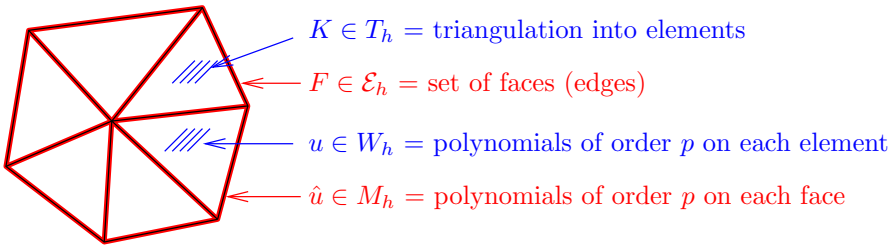


Figure 7.4.1: Definitions used in hybrid DG discretizations.

$$(\mathbf{w}, \mathbf{v})_{T_h} = \sum_{K \in T_h} (\mathbf{w}, \mathbf{v})_K = \sum_{K \in T_h} \int_K \mathbf{w}^T \mathbf{v} \, dV \qquad \text{(sum of integrals over elements)}$$

$$\langle \mathbf{w}, \mathbf{v} \rangle_{\partial T_h} = \sum_{K \in T_h} \langle \mathbf{w}, \mathbf{v} \rangle_{\partial K} = \sum_{K \in T_h} \int_{\partial K} \mathbf{w}^T \mathbf{v} \, dS \qquad \text{(sum of integrals over faces)}$$

Testing Equation 7.4.1 with functions $\mathbf{w} \in \mathbf{W}_h$, we obtain the "standard" weak form,

$$-\left(\vec{\mathbf{F}}(\mathbf{u}), \nabla \mathbf{w}\right)_{T_h} + \langle \hat{\mathbf{F}} \cdot \mathbf{n}, \mathbf{w} \rangle_{\partial T_h} = 0, \quad \forall \mathbf{w} \in \mathbf{W}_h. \tag{7.4.2}$$

One difference compared to standard DG is the definition of the numerical flux, $\hat{\mathbf{F}} \cdot \mathbf{n}$, which in HDG/EDG takes the form

$$\hat{\mathbf{F}} \cdot \mathbf{n} = \mathbf{F}(\widehat{\mathbf{u}}) \cdot \mathbf{n} + \mathbf{S}(\mathbf{u}, \widehat{\mathbf{u}})(\mathbf{u} - \widehat{\mathbf{u}}), \tag{7.4.3}$$

where $\mathbf{S}$ is a local stabilization matrix that plays a similar role to components of the flux function in standard DG. Note that the numerical flux depends on the state inside the element, $\mathbf{u}$, and the interface state, $\widehat{\mathbf{u}}$, but *not* on the state in the neighboring element. This choice will allow us to significantly reduce the number of globally-coupled degrees of freedom.

The flux computed via Equation 7.4.3 on one side of a face may now not be the same as the flux computed on the other side. We obtain additional equations by enforcing equality of these fluxes in a weak sense (this also ensures conservation),

$$\langle \hat{\mathbf{F}} \cdot \mathbf{n}, \boldsymbol{\mu} \rangle_{\partial T_h \setminus \partial \Omega} + \langle \hat{\mathbf{B}}, \boldsymbol{\mu} \rangle_{\partial \Omega} = 0, \quad \forall \boldsymbol{\mu} \in \mathbf{M}_h. \tag{7.4.4}$$

$\partial\Omega$ is the boundary of the domain and $\hat{\mathbf{B}}$ is a term that enforces boundary conditions. For example, on an inviscid wall, $\hat{\mathbf{B}} = \mathbf{b} - \widehat{\mathbf{u}}$, where $\mathbf{b}$ is the interior $\mathbf{u}$ with the normal velocity component set to zero.

Static condensation Our two equations, Equation 7.4.2 and Equation 7.4.4, are

$$\begin{aligned} -\left(\vec{\mathbf{F}}(\mathbf{u}), \nabla \mathbf{w}\right)_{T_h} &+ \langle \hat{\mathbf{F}} \cdot \mathbf{n}, \mathbf{w} \rangle_{\partial T_h} = 0, \quad \forall \mathbf{w} \in \mathbf{W}_h \\ \langle \hat{\mathbf{F}} \cdot \mathbf{n}, \boldsymbol{\mu} \rangle_{\partial T_h \setminus \partial \Omega} &+ \langle \hat{\mathbf{B}}, \boldsymbol{\mu} \rangle_{\partial \Omega} = 0, \quad \forall \boldsymbol{\mu} \in \mathbf{M}_h \end{aligned}$$

Linearizing these equations (e.g. for implicit calculations) yields the system

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{\Lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{F} \\ \mathbf{G} \end{bmatrix}, \tag{7.4.5}$$

where $\mathbf{U}$ is the discrete vector of unknowns in the approximation of the state in each element ($\mathbf{u}$), and $\mathbf{\Lambda}$ is the vector of unknowns in the approximation of the state on the interfaces ($\widehat{\mathbf{u}}$).

The system in Equation 7.4.5 is larger than the standard DG system. However, it has a special structure in that $\mathbf{A}$ is element-wise block diagonal. That is, unknowns in $\mathbf{U}$ from two separate elements are not coupled together. That means that we can invert $\mathbf{A}$ locally on each element (and in parallel too!) to compute $\mathbf{A}^{-1}$.

Specifically we can eliminate $\mathbf{U}$ as an unknown by "statically-condensing" these degrees of freedom out of the system. This just means solving the first block row in Equation 7.4.5 for $\mathbf{U}$ and substituting into the second block row to obtain

$$\mathbf{K}\mathbf{\Lambda} = \mathbf{R}, \tag{7.4.6}$$

where

$$\mathbf{K} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}, \qquad \mathbf{R} = \mathbf{G} - \mathbf{C}\mathbf{A}^{-1}\mathbf{F}$$

The system in Equation 7.4.6 is now (typically) a much smaller system for the unknowns in the state approximation on element interfaces. The stencil in $\mathbf{K}$ can be considered compact in that only unknowns on faces adjacent to the same element are coupled.

Once $\mathbf{\Lambda}$ is known, the states on each element are obtained according to

$$\mathbf{U} = \mathbf{A}^{-1}\left(\mathbf{F} - \mathbf{B}\mathbf{\Lambda}\right)$$

To recap, the key idea in HDG/EDG methods is to reduce the set of *globally-coupled* degrees of freedom – those DOFs that require a global solve – by approximating the state on element interfaces in addition to the element-interior approximation. After static condensation, these additional unknowns remain as the only globally-coupled unknowns in the problem.

Degree of freedom comparison In DG, the number of globally-coupled unknowns scales as p^{dim}, where "dim" is the spatial dimension. HDG and EDG have the advantage that the exponent is reduced by one to $p^{\text{dim}-1}$, since the unknowns are defined on a space of one lower dimension (faces instead of volumes).

To quantify this advantage, we compare standard DG, continuous FEM (CG), and HDG/EDG using degrees of freedom per vertex of the mesh. This comparison assumes typical isotropic meshes (e.g. 6 triangles or 24 tetrahedra adjacent to a vertex) and ignores boundary effects.

The numbers in the below tables indicate approximately how many degrees of freedom (per equation of a system) we need per vertex of the mesh.

Triangles:

method	$p=1$	$p=2$	$p=3$	$p=4$
DG	6	12	20	30
CG	1	4	9	16
HDG	6	9	12	15
EDG	1	4	7	10

Quadrilaterals:

method	$p=1$	$p=2$	$p=3$	$p=4$
DG	4	9	16	25
CG	1	4	9	16
HDG	4	6	8	10
EDG	1	3	5	7

Tetrahedra:

method	$p=1$	$p=2$	$p=3$	$p=4$
DG	24	60	120	210
CG	1	8.2	27.4	64.6
HDG	36	72	120	180
EDG	1	8.2	27.4	58.6

Hexahedra:

method	$p=1$	$p=2$	$p=3$	$p=4$
DG	8	27	64	125
CG	1	8	27	64
HDG	12	27	48	75
EDG	1	7	19	37

We see that for low-to-moderate orders, HDG can be as or even more expensive than standard DG. However, above $p = 2$ or $p = 3$, HDG is clearly cheaper. EDG shows a much more dramatic improvement, requiring as many or even fewer degrees of freedom compared to CG.

7.5 Field Inversion and Machine Learning

7.5.1 Introduction

Turbulent flows exist in many fluid systems, including in the aerodynamics of aerospace vehicles. Many techniques have been developed to simulate such flows, ranging from completely ignoring viscous effects, for example through potential-flow or Euler equations, to resolving every turbulent scale via direct numerical simulation (DNS). In between lie steady-state methods based on Reynolds averaging (RANS), and unsteady methods such as large-eddy and detached-eddy simulations (LES, DES). Whereas steady-state methods are generally adequate for vehicles at on-design conditions, high-fidelity simulations at off-design conditions, often characterized by regions of separated flow, generally require unsteady methods.

Unsteady models of turbulent flow complicate shape optimization in at least two ways. First, the simulations are much more expensive than steady models, so that each function evaluation, usually a statistic such as a time-averaged scalar output of interest, requires orders of magnitude more computational time and resources. This limits the applicability of many-query studies, such as optimization, particularly gradient-free methods that rely on numerous function evaluations [82]. Second, the chaotic nature of unsteady turbulent simulations prevents the application of linear sensitivity techniques for evaluation of gradients. In particular, adjoint methods, the hallmark of shape optimization [121, 99, 94, 39, 33], cannot be directly applied, except in special cases such as quasi-periodic flows [126], as the unsteady adjoint equations are unstable in such systems [81] and require expensive regularization techniques to provide meaningful answers [149, 151, 150, 21, 104, 105]. Furthermore, unsteady adjoints come with massive storage and computation requirements that become impractical for forward simulations, which already stress computational resources [61].

Unsteady adjoints are useful when computing sensitivities of deterministic events, such as maneuvers, gust interactions, or short-duration aeroelastic events [97, 93, 48, 43, 83, 45, 107, 18, 4]. However, for turbulent flows, they provide perhaps too much information: the sensitivity of an output to a flow residual at a single point in space and time loses significance when that particular state may not appear in another simulation that follows a different trajectory due the chaotic nature of the flow [86]. In such systems, of engineering interest are generally statistical quantities, such as time averages of outputs, and if a steady-state model were able to yield these quantities with sufficient accuracy, it would be much more amenable to gradient-based optimization.

Many steady-state turbulence models exist, most based on Reynolds averaging of the Navier-Stokes equations. However, as discussed previously, these models often do not provide sufficient accuracy relative to unsteady models at off-design conditions. Indeed, the creation of a general turbulence model that is accurate across multiple flow regimes continues to be an elusive quest [127]. Here, we consider foregoing generality and creating multiple steady-state models, each specific to the unsteady case being considered. We only use the model to enable an efficient calculation of the derivatives required for optimization, instead of trying to apply the model to vastly different flow regimes.

The turbulence modeling approach we take here is based on the idea of field inversion and

machine learning (FIML) [111, 135, 136, 72, 71, 158, 79]. The idea of FIML is to start with an existing steady-state model and to correct it via a PDE-level correction factor on one or more of the turbulence modeling terms. The unsteady simulation, here in lieu of experimental or other external data, provides the truth solution for the inverse problem for this correction factor, and a machine-learning approach maps local flow features to the correction factor. The result is a corrected steady-state model that can be used on different meshes or shapes. Whereas previous works have used FIML as a general turbulence modeling strategy with mixed generalizability success [127], here we only rely on FIML's local accuracy, at/near the conditions for which it was trained. This relaxation of expectations then simplifies aspects of the FIML approach, particularly the amount of training data, size of the machine learning neural network, and the generality of the inputs.

The use of machine learning here is not to create a direct input-to-output surrogate of the high-fidelity model, as has been done in many previous works [143, 159, 124, 152]. Instead, machine learning is only used to provide a correction to what is already a high-fidelity model, RANS, so as to make it better represent an even higher-fidelity model, unsteady turbulence. The corrected RANS model enables calculation of the objective gradient, which is needed for gradient-based shape optimization.

The unsteady optimization approach presented here consists of iterative turbulence model calibration, through FIML and a *limited* number of unsteady simulations, coupled with steady-state optimization using the corrected turbulence models. The number of unsteady simulations required is much lower than the function evaluations demanded by any gradient-free or even gradient-based method, since the optimization is offloaded to the steady-state model. The unsteady simulations only provide model training data, and as the model is often accurate in the vicinity of training, only a few unsteady evaluations (2-4 in the present results) are typically required. However, as will be shown in the results, the accuracy improvements of the corrected model can make a difference in the optimized shapes, particularly in cases where the uncorrected model loses applicability.

7.5.2 Field Inversion

Unsteady Solution Averaging

Let $\bar{\mathbf{U}}$ be a statistically-steady flow state computed as the time-average of the unsteady simulation after initial transients,

$$\bar{\mathbf{U}} = \frac{1}{T_f - T_i} \int_{T_i}^{T_f} \mathbf{U}(t)\, dt, \tag{7.5.1}$$

where T_i is the start time, taken sufficiently large to minimize startup transient effects, and T_f is the final time, taken sufficiently greater than T_i to yield an adequate statistical mean. The goal of field inversion is to determine a correction field, which enters into the governing equations at the PDE level, to a steady-state model such that the corrected steady-state solution reproduces $\bar{\mathbf{U}}$.

The proximity of the corrected solution to $\bar{\mathbf{U}}$ is measured by the field-inversion error, $\mathcal{E}$, which could be the error in an engineering quantity such as lift or drag, a weighted combination of errors in scalar quantities, an L_2 norm of the entire state error, an L_2 norm of a surface stress distribution error, etc. Presently, we consider the stress distribution error,

$$\mathcal{E} = \frac{1}{2}\int_{\text{airfoil}} \|\boldsymbol{\sigma}(\mathbf{U})\cdot\vec{n} - \boldsymbol{\sigma}(\bar{\mathbf{U}})\cdot\vec{n}\|^2\, ds, \tag{7.5.2}$$

where $\boldsymbol{\sigma}$ is the stress tensor and $\vec{n}$ is the unit normal vector on the airfoil surface. We note that the entire time-averaged flowfield is available and could also be used in the error definition. However, using just the surface stress distribution often yields better matching of force outputs and their sensitivities.

7.5.3 Correction-Field Inversion

The correction is a scalar field, $\beta(\vec{x})$, that multiplies the production term of the Spalart-Allmaras (SA) turbulence model. The production term appears in the SA eddy-viscosity equation,

$$\begin{aligned}\frac{\partial(\rho\tilde{\nu})}{\partial t} + \nabla\cdot(\rho\vec{v}\tilde{\nu}) - \frac{1}{\sigma}\nabla\cdot[\rho(\nu+\tilde{\nu}f_n)\nabla\tilde{\nu}] = \\ -\frac{1}{\sigma}(\nu+\tilde{\nu}f_n)\nabla\rho\cdot\nabla\tilde{\nu} + \frac{c_{b2}}{\sigma}\rho\nabla\tilde{\nu}\cdot\nabla\tilde{\nu} + \boxed{\beta}P - D.\end{aligned} \tag{7.5.3}$$

In this equation, ρ is the density, $\vec{v}$ is the velocity, $\tilde{\nu}$ turbulence working variable, f_n is a function that is active for $\tilde{\nu} < 0$, c_{b2}, σ are model constants, P is the turbulence production function, and D is the turbulence destruction function. Scaling P by β affects the entire solution as the eddy viscosity equation is coupled to the conservation equations.

The nominal value of this correction field is $\beta = 1$. The discrete representation of this field depends on the discretization, and presently, $\beta(\vec{x})$ is approximated by a $p = 1$ Lagrange DG basis on each element, even for $p > 1$ state approximations. Let $\boldsymbol{\beta}$ be the vector of basis coefficients used in the approximation of $\beta(\vec{x})$, numbering $3N_e$ and $4N_e$ values for triangles and quadrilaterals, respectively. The use of a Lagrange basis makes the nominal $\boldsymbol{\beta}$ a vector of all ones. The field inversion then becomes a discrete optimization problem,

$$\begin{aligned}\min_{\boldsymbol{\beta}} \quad & J^{\text{inv}} \equiv \mathcal{E}(\mathbf{U}(\boldsymbol{\beta})) + \frac{\gamma}{2}(\boldsymbol{\beta}-1)^T\mathbf{M}(\boldsymbol{\beta}-1) \\ \text{s.t.} \quad & \mathbf{R}(\mathbf{U},\boldsymbol{\beta}) = \mathbf{0}\end{aligned} \tag{7.5.4}$$

Here, $\mathbf{R}$ is the discrete residual vector that arises from the discretization. It depends on the state and on the correction, which enters the equations through the eddy-viscosity production term. The additional term in the minimization problem is a continuous Tikhonov regularization that makes the inverse problem well-posed by penalizing large deviations of the correction field from the nominal value of one. $\mathbf{M}$ is the mass matrix from the spatial discretization, and γ is a small user-prescribed parameter that needs to be sufficiently

large to prevent ill-conditioning but not too large so as to affect the minimization of $\mathcal{E}$. A broad range of values was typically found to be acceptable, and for the normalized problems considered here, with $\mathcal{O}(1)$ units, $\gamma = 10^{-7}$ was used.

Equation 7.5.4 can be solved using a simple steepest descent minimization algorithm or one that is more sophisticated, such as (L-)BFGS. In both cases, the PDE-constrained gradient of the objective, J^{inv}, with respect to $\boldsymbol{\beta}$ is evaluated using an adjoint,

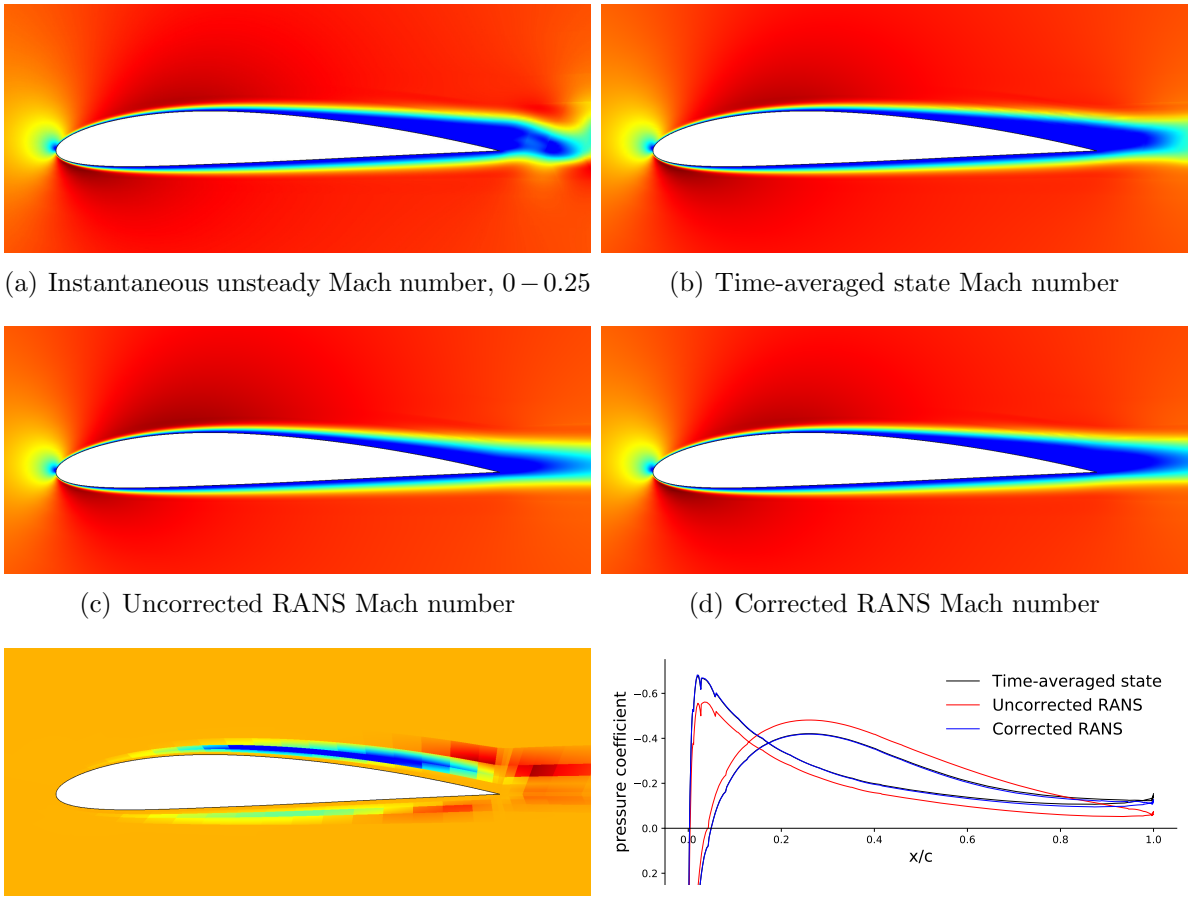
$$\frac{dJ^{\text{inv}}}{d\boldsymbol{\beta}} = \left(\boldsymbol{\Psi}^{\mathcal{E}}\right)^T \frac{\partial \mathbf{R}}{\partial \boldsymbol{\beta}} + \gamma \mathbf{M}(\boldsymbol{\beta} - 1), \qquad \left(\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right)^T \boldsymbol{\Psi}^{\mathcal{E}} + \left(\frac{\partial \mathcal{E}}{\partial \mathbf{U}}\right)^T = \mathbf{0}. \tag{7.5.5}$$

The linearization of the residual with respect to the correction, $\frac{\partial \mathbf{R}}{\partial \boldsymbol{\beta}}$, is local to the elements: by virtue of the DG discretization and the fact that the correction enters via a residual source term, coefficients of $\boldsymbol{\beta}$ for an element affect the discrete residuals only on that element. Therefore, $\frac{\partial \mathbf{R}}{\partial \boldsymbol{\beta}}$ is evaluated efficiently using finite differences. This requires only a small number of residual evaluations (e.g. 3 for triangles), as derivatives on all elements can be calculated concurrently.

7.5.4 Example

To demonstrate the accuracy of the RANS correction, we consider a NACA 3412 airfoil at $M = 0.2$, $Re = 10^4$, and angle of attack $\alpha = 0$. This low Reynolds-number flow exhibits unsteady behavior that prevents the use of direct unsteady adjoints for time-averaged outputs [133]. The case is solved using $p = 2$ solution approximation on a mesh of 1978 $q = 3$ quadrilateral elements. A third-order modified backwards difference scheme (MEBDF) [29] is used to advance the unsteady cases in time, using a time step of $\Delta t = .05c/U_\infty$, where U_∞ is the free-stream speed. After a transient, a time horizon of $T_f - T_i = 30c/U_\infty$ is used to calculate the average state according to Equation 7.5.1.

RANS models are typically tuned for high Reynolds-number turbulent flows, so we should not expect good agreement between unsteady results and a standard RANS result. This is indeed the case, as shown in Figure 7.5.1. The uncorrected RANS solution exhibits a smaller wake and a different pressure distribution compared to the averaged unsteady result. With the correction to the production term, however, the RANS model can be modified to quite-well match the averaged unsteady result. Figure 7.5.1 shows that the Mach number profile and pressure distribution of the corrected RANS result are nearly identical to the time-averaged unsteady result. The field inversion in this case is done using the stress distribution error function, Equation 7.5.2. The $\beta(\vec{x})$ plot indicates reduced turbulent production on the boundary layer edges, particularly over the upper surface, and increased production towards the trailing edge and wake.



(a) Instantaneous unsteady Mach number, $0-0.25$

(b) Time-averaged state Mach number

(c) Uncorrected RANS Mach number

(d) Corrected RANS Mach number

(e) Correction field, $\beta(\vec{x})$, $0.3-1.3$

(f) Surface pressure coefficient

Figure 7.5.1: Field-inversion results for unsteady flow around a NACA 3412 airfoil at $M = 0.2, Re = 10^4, \alpha = 0$.

7.5.5 Machine Learning

Overview

The correction field obtained from the solution of the inverse problem makes the solution of the corrected RANS equations match the unsteady average for a particular geometry and mesh. In design optimization, we would like to consider changes to the geometry and trim parameters, which lead to different meshes and states. The correction field then also needs to change, in a manner generally beyond a simple mapping, as new geometry or state features may require local changes to the correction.

One approach to realizing consistent changes to the correction field is to make this field a function of the state and gradient, i.e. to create a model for β as a function of the *local* $\mathbf{u}$ and $\nabla\mathbf{u}$. As the correction enters the equations at the residual level, propagation effects of convection-dominated systems can be modeled in this approach. For example, a discrepancy in the location of separation can be resolved through changes in the turbulent production at/before separation, where the boundary-layer profile can be controlled, instead of in the wake, where the largest state discrepancies occur but where little control is possible.

The link between the state and the correction field can be made in the form of an analytical model [138], where the coefficients of the model can be fit to features of the state. Alternatively, the analytical model can be replaced altogether by a direct mapping from the state to the correction field, through an artificial neural network [154, 10]. This idea forms the basis of the field-inversion machine-learning (FIML) approach to corrected turbulence modeling [111]. We note that the network plays a minor role in this method, in contrast to other approaches that use a network to replace the entire turbulence model [163]. In FIML, the network could be replaced by other functional mappings or closed form expressions [79].

FIML has seen success in making solutions of corrected turbulence models match unsteady or higher-fidelity data for a variety of cases [135, 72, 71, 79]. The generality of this approach is still in debate [127], and it is possible that FIML may never yield a general-purpose "new" turbulence model that performs better than existing analytical ones. In addition, the identification and selection of the most physically relevant local state features for use as inputs into the neural network has a high degree of arbitrariness [71].

These possible shortcomings of FIML do not affect our present study, which does not aim to derive a general-purpose turbulence model. Our goal is to create a locally-valid corrected turbulence model that reproduces the behavior of the unsteady system at geometries and conditions that are close in parameter space. The model is retrained as the optimization progresses and more unsteady data become available. Furthermore, the fact that the model is specific to the problem simplifies the network input selection. Instead of choosing, perhaps arbitrarily, non-dimensional, physically meaningful features, we take a more systematic approach and use the entire state vector, its spatial gradient, and the wall distance as the network input.

Network Architecture and Training

The neural network maps local flowfield information to the scalar correction field. Figure 7.5.2 shows the structure of the network used here, which is a single-hidden-layer perceptron. The hidden layer, $\mathbf{x}_1$, contains n_1 neurons, between the input layer, i.e. the features, $\mathbf{x}_0$, and the output layer, which consists of the scalar correction factor, β. The map from the in-

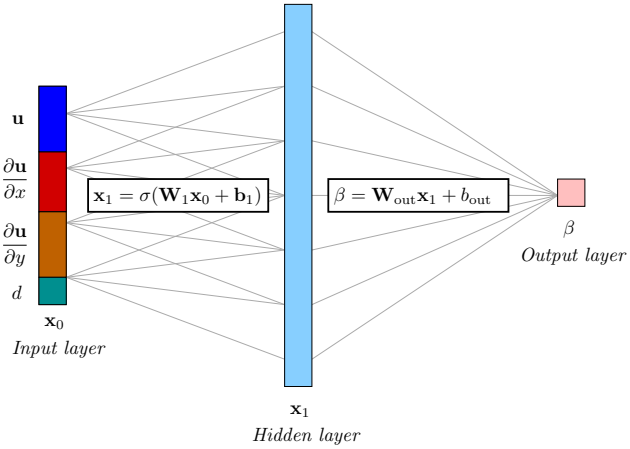


Figure 7.5.2: Structure of the artificial neural networks used to predict the correction field.

put to the hidden layer involves an entry-wise sigmoid activation function, $\sigma(x) = 1/(1+e^{-x})$, whereas no activation function is used for the output layer calculation. The parameters associated with the network consist of the weights and biases, $\mathbf{W}_i \in \mathbb{R}^{n_i \times n_{i-1}}$, $\mathbf{b}_i \in \mathbb{R}^{n_i}$, where n_i is the number of neurons in layer i.

The input into the network consists of the state, $\mathbf{u}$, its gradient, $\nabla\mathbf{u}$, and the wall distance, d, for a total of $(1 + \dim)s + 1$ neurons. After experimentation, the size of the hidden layer was set to $n_1 = 30$ for the two-dimensional problems considered here. The observed trade-off lies between over-fitting and high quadrature order requirements for large n_1 and a loss of accuracy for small n_1. A fairly wide range of values, $n_1 \in [16, 50]$, showed similar performance, so that the results are not overly sensitive to the precise value of n_1.

The parameters associated with the network consist of all of the weights and biases. The values of these parameters are determined using an optimization procedure, the adaptive moment (Adam) estimation algorithm in TensorFlow [1], that minimizes the mean squared error loss function between predicted and actual output-layer values. The actual output-layer values, i.e. values of the correction field, come from the field inversion result for one particular unsteady simulation. As $\beta(x)$ is a field, each simulation yields ostensibly an infinite amount of training data. However, the finite-dimensional representation of the state and $\beta(x)$ means that not all of these data are independent or informative for training. Presently, we sample the required fields (state, gradient, wall distance, correction) at the quadrature points of each element. Each inversion result then yields an amount of training data that depends on the mesh size and quadrature order.

The training data are broken into mini-batches of size 1000 for the optimizer, and the

learning rate is set to .001. Prior to training, the weights and biases are initialized randomly from a unit normal distribution. Experiments with different learning rates, batch sizes, and initialization showed that the final results were not overly sensitive to these choices. 500,000 optimization iterations are taken in each training session for all of the presented results, although the mean-squared error typically stabilizes well before this number. Two to three orders of magnitude drop in the loss are usually observed.

Implementation and Example

Once a network is trained, it is implemented as a physics model in the turbulence source calculation of the flow simulation code. No changes to the inputs to this calculation are required, since the turbulent source already uses the state, its gradient, and the wall distance function. All parts of the source calculation must be differentiated with respect to the state and its gradient for the forward Newton solver and the discrete adjoint solver. This includes the neural-network correction field calculation, the linearization of which is calculated using back-propagation [154]. For a single-hidden layer network, the derivative of the scalar output with respect to the input vector is calculated as follows:

$$\beta = \mathbf{W}_{\text{out}}\sigma(\mathbf{W}_1\mathbf{x}_0 + \mathbf{b}_1) + b_{\text{out}} \quad \Rightarrow \quad \frac{\partial\beta}{\partial\mathbf{x}_0} = \mathbf{W}_2 \ \text{diag}(\boldsymbol{\sigma}_1') \ \mathbf{W}_1, \tag{7.5.6}$$

where $\boldsymbol{\sigma}_1'$ is the derivative of the activation function at the hidden layer values. For the sigmoid function, this derivative is $\sigma' = \sigma(1-\sigma)$, and hence its calculation only requires storing the hidden layer post-activation values. The chain rule calculation in Equation 7.5.6 extends systematically to multiple hidden layers. Due to the nonlinear nature of the network, higher-order accurate quadrature rules are used for the corrected RANS model. Presently, an constant order increment of 1 is added to the baseline $2p+1$ order requirement in the code.

To demonstrate the ability of the neural-network model to produce an accurate correction field, we consider multiple airfoils of varying camber. For five airfoils of cambers 0 to 5% of the chord, unsteady simulations, field inversions, and neural network trainings are separately performed. The output of interest is the lift coefficient and its sensitivity to camber changes.

Figure 7.5.3 shows the lift-coefficient results of the various simulations. The uncorrected RANS results show an increasing lift coefficient with camber, and adjoint-based gradients that match the trend from the multiple simulations. However, this trend disagrees with the time-averaged unsteady results, which show an initial decrease in the averaged lift coefficient for small cambers, followed by an increase. The result is not surprising, as the RANS model is not designed for such low Reynolds numbers.

Applying field inversion to each case and using the resulting correction field $\beta(\vec{x})$ in the turbulence model yields data that track the time-averaged results well. More importantly, the neural-network model for $\beta(\vec{x})$ also performs very well, as indicated by the "FIML" data points in Figure 7.5.3. These data correspond to simulations performed by using the neural network model and not the inverted $\beta(\vec{x})$ field, which was used only for training. Furthermore, when differentiated as part of the adjoint solver, the FIML result yields a

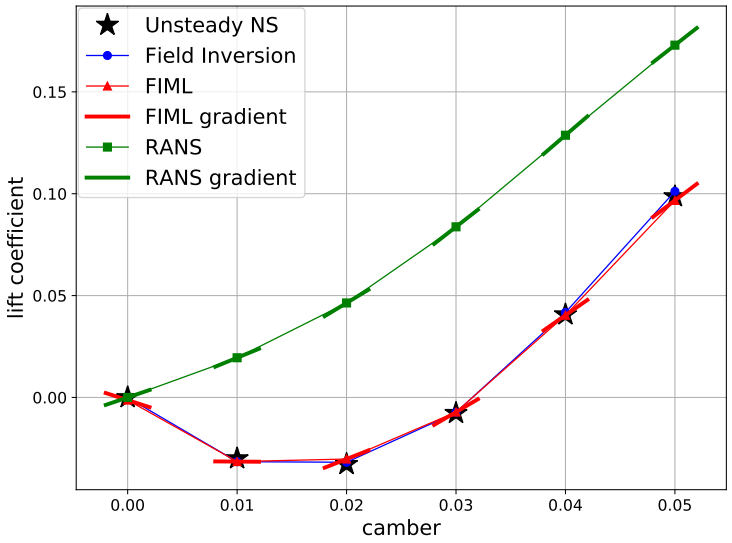


Figure 7.5.3: Field-inversion and machine learning results for lift-coefficient calculations in unsteady flow around NACA X412 airfoils at $M = 0.2, Re = 10^4, \alpha = 0$.

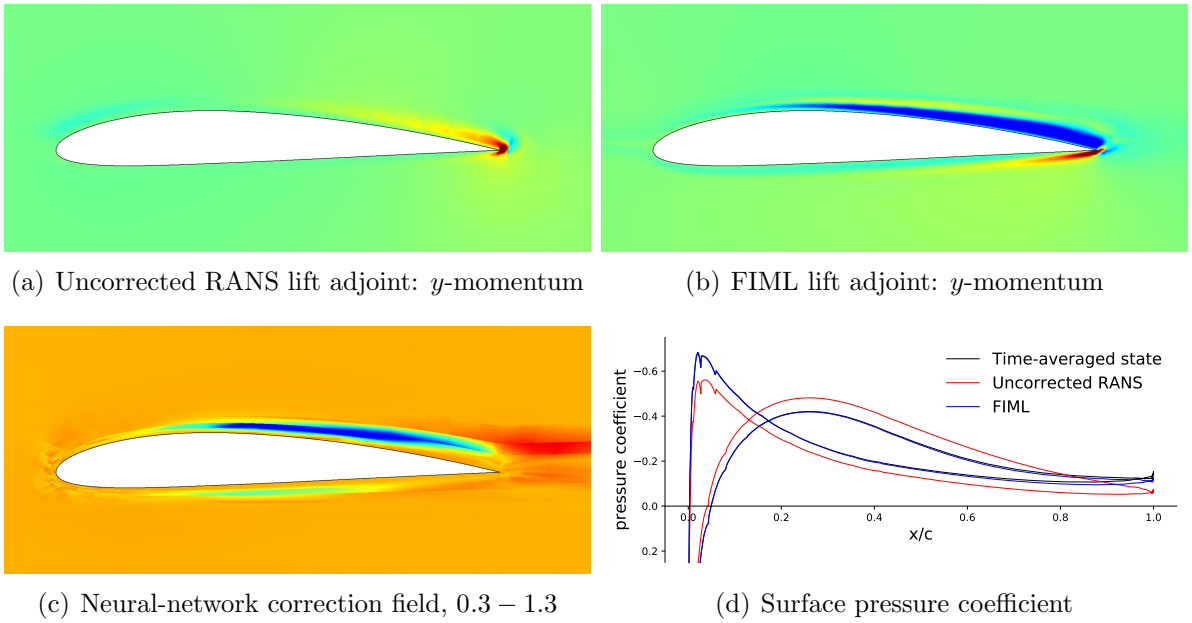
reasonably-accurate PDE-constrained gradient of the lift coefficient with respect to the camber. The ability to accurately predict the lift coefficient value and its gradient with respect to shape parameters is important when using the FIML correction model in optimization.

Figure 7.5.4 shows the differences in the lift adjoint fields, the conservation of y momentum component, obtained from the uncorrected RANS and the FIML solutions. These are shown on the same scale, and the differences are large, particularly over the upper surface of the airfoil. The correction field, $\beta(\vec{x})$, is also shown, calculated using the neural network at the converged FIML state. This field matches well with the inversion result shown in Figure 7.5.1. Finally, the pressure distribution from the FIML state matches the averaged unsteady pressude distribution just as well as the field inversion result.

7.5.6 Optimization Example: Lift Maximization at $Re = 10^4$

As an example of the use of FIML for unsteady problems where an unsteady adjoint is not possible, we consider the maximization of the lift produced by an airfoil at free-stream $M = 0.2$ and $Re = 10^4$. The airfoil has an area constraint of $A = 0.06c^2$, and the design parameters consist of seven shape parameters governing the camber and thickness profile, augmented by the angle of attack. No trim parameters are necessary for this unconstrained maximization problem. Figure 7.5.5 shows the mesh, which consists of $q = 3$ curved quadrilaterals generated by a script according to the specified shape parameters.

We first optimize the airfoil using the uncorrected RANS equations, starting from a baseline airfoil in which all shape parameters are zero, and this optimization yields the RANS airfoil. An unsteady analysis of the RANS airfoil is then performed, and time-averaged state and output information from the unsteady analysis of the RANS airfoil provide the target



(a) Uncorrected RANS lift adjoint: y-momentum

(b) FIML lift adjoint: y-momentum

(c) Neural-network correction field, $0.3 - 1.3$

(d) Surface pressure coefficient

Figure 7.5.4: FIML and RANS contour plots for unsteady flow around a NACA 3412 airfoil at $M = 0.2, Re = 10^4, \alpha = 0$.

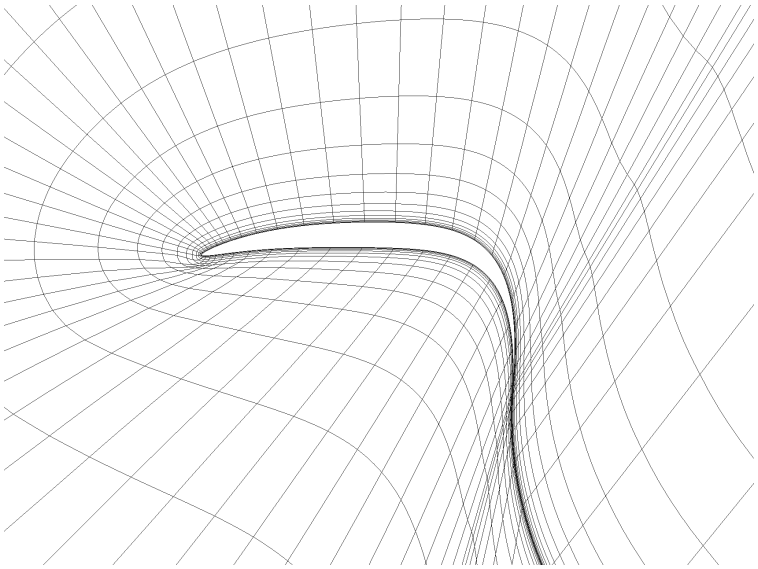


Figure 7.5.5: Mesh used in lift maximization, shown for the final optimized geometry.

data for field inversion and machine learning. The resulting FIML model is used to re-optimize the airfoil with unsteady information, using the same optimization technique as for the steady RANS case. The combination of unsteady analysis and FIML optimization constitutes one FIML optimization iteration, and a total of three such iterations (requiring three unsteady analyses) are performed to obtain the FIML airfoil. We note that the RANS and the FIML airfoils also each come with their optimized angle of attack.

Figure 7.5.6 shows the geometries of the two airfoils. Both are highly cambered, with a smooth, near right-angle flow deflection at the trailing edge, starting at approximately 75% of the chord. However, the airfoils do have noticeable differences. The FIML airfoil has a smaller radius of curvature at the leading edge and is thinner for the first 25% of the chord. Its thickness then gradually increases over the middle before tapering off at the trailing edge. In contrast, the RANS airfoil starts out thicker and maintains a relatively constant thickness distribution from 25% to 70% of the chord. Its upper surface is also flatter than that of the FIML airfoil. The optimum angles of attack for the two airfoils are not too different: 22.92°

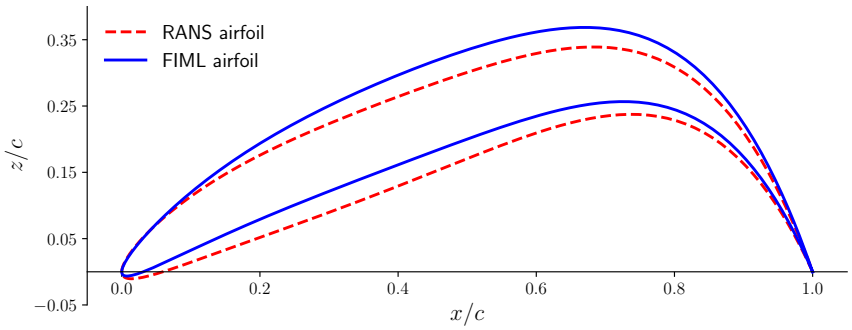


Figure 7.5.6: Airfoils optimized for maximum lift at $M = 0.2, Re = 10^4$.

for the RANS airfoil and 23.41° for the FIML airfoil. These large values align the flow with the direction of the first portion of the airfoil, so that the last 25% of the chord constitutes an effective smooth flap.

Table 7.5.1 presents the lift coefficient results for the optimized airfoils. It shows values from a steady-state RANS analysis and an unsteady analysis (time-averaged value), for both airfoils. Simulations for each airfoil are performed at the airfoil's optimized angle of attack. We see that each airfoil has the higher lift coefficient in the analysis that matches the optimization. The RANS airfoil has the higher lift coefficient in the steady analysis, as expected from the steady optimization. The FIML airfoil has the higher lift coefficient in the unsteady analysis, by over 8.5%, which indicates that the FIML model is accounting for the unsteady effects of the flow. We also note that the unsteady analyses predict much higher lift coefficients than steady RANS, by 40% to 60%.

Figure 7.5.7 shows the lift coefficient time histories from the unsteady analyses of the two airfoils. Both histories show a high degree of unsteadiness, but the range of lift coefficient variation is smaller in the FIML airfoil case: between 2.5 and 3.5 compared to 1.75 and 3.25 in the RANS airfoil. The time history of the RANS airfoil lift coefficient is punctuated

Table 7.5.1: Lift coefficient results for lift maximization at $M = 0.2, Re = 10^4$.

Airfoil	Angle of attack	Analysis	Lift coefficient
RANS airfoil	22.92°	Steady	1.93
RANS airfoil	22.92°	Unsteady	2.71
FIML airfoil	23.41°	Steady	1.85
FIML airfoil	23.41°	Unsteady	2.94

by large drops in the lift coefficient, which correspond to large vortex shedding events off the leading edge. These are not present for the FIML airfoil. In addition, the overall lower average lift coefficient for the RANS airfoil is discernible from the time histories.

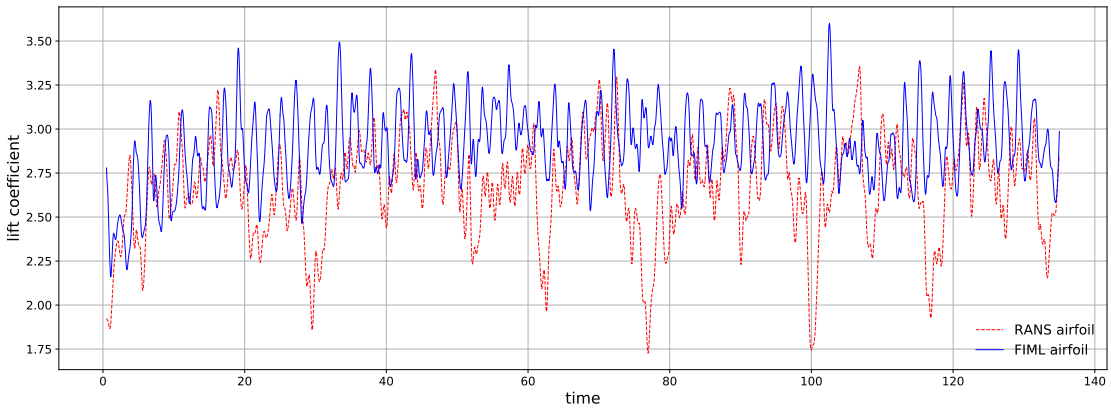
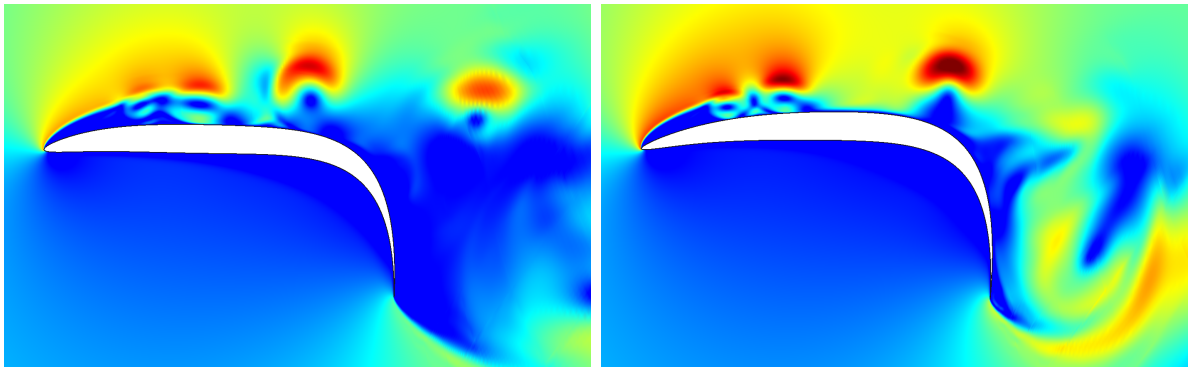


Figure 7.5.7: Lift coefficient time histories for lift maximization at $M = 0.2, Re = 10^4$.

Figure 7.5.8 compares Mach number snapshots of the unsteady flowfields for both airfoils. The large degree of unsteadiness in the flowfield is apparent, with vortices that start shortly after the leading edge on the upper surface. The flat front portion of the RANS airfoil leads to a larger region of leading-edge separation, in contrast to the more cambered FIML airfoil front portion that allows for reattachment. In both airfoils, the high effective flap angle leads to large shed vortices in the wake.

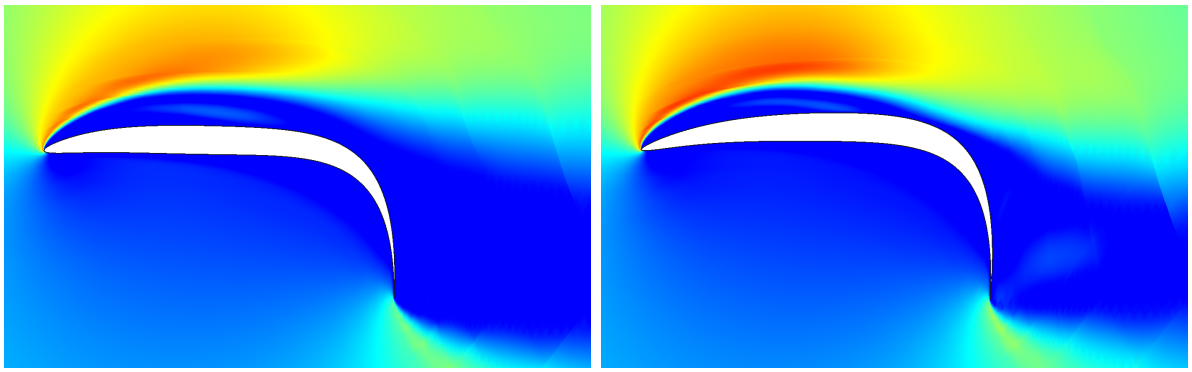
Figure 7.5.9 shows the time-averaged Mach number contours for both cases, computed from the unsteady analyses. We see that a region of separated flow on the upper surface starts almost immediately at the leading edge in both airfoils, but that it re-attaches by the start of the flap in the FIML airfoil. This then leads to more flow turning on the upper surface and hence larger lift generation. In the RANS airfoil case, the flow does not re-attach, leading to a larger wake, less flow turning and hence less lift.

Finally, to test the performance of the airfoils off-design, and to verify the angles of attack determined from the optimization, we perform an angle of attack sweep of both airfoils. For the unsteady analyses, this involves running a separate unsteady simulation at each angle of attack and computing the time-averaged lift coefficient. Figure 7.5.10 shows the resulting lift coefficient curves. In addition to the two unsteady analyses, we also include a steady



(a) RANS airfoil (b) FIML airfoil

Figure 7.5.8: Instantaneous Mach number contours (0-.5) of airfoils optimized for lift maximization at $M = 0.2, Re = 10^4$.



(a) RANS airfoil (b) FIML airfoil

Figure 7.5.9: Time-averaged Mach number contours (0-.5) of airfoils optimized for lift maximization at $M = 0.2, Re = 10^4$.

analysis result of the RANS airfoil. The optimal angles of attack for the unsteady FIML airfoil analysis and the steady RANS airfoil analysis are indeed correctly predicted by the optimization, as indicated by the large symbols. The unsteady angle of attack sweep of the RANS airfoil shows that the maximum time-averaged lift actually occurs at an angle of attack that is lower, about $\alpha = 21°$, than the 22.92° angle predicted by the steady analysis. In contrast, the FIML optimization correctly predicts the angle of attack at maximum time-averaged lift.

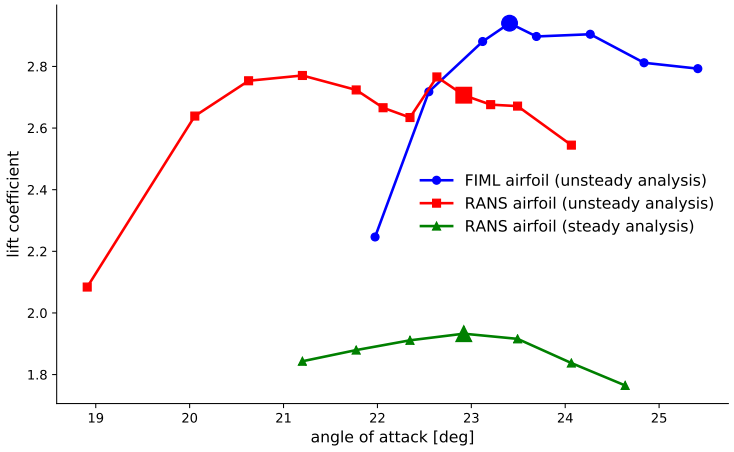


Figure 7.5.10: Lift coefficient versus angle of attack for the optimized airfoils for lift maximization at $M = 0.2, Re = 10^4$. Large symbols indicate the angle of attack determined from the optimization.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M.J. Aftosmis and M.J. Berger. Multilevel error estimation and adaptive h-refinement for Cartesian meshes with embedded boundaries. AIAA Paper 2002-14322, 2002.

[3] Mark Ainsworth and Bill Senior. An adaptive refinement strategy for *hp*-finite element computations. *Applied Numerical Mathematics*, 26:165–178, 1998.

[4] Frédéric Alauzet, Adrien Loseille, and Geraldine Olivier. Time accurate anisotropic goal-oriented mesh adaptation for unsteady flows. *Journal of Computational Physics*, 373(15):28–63, 2018.

[5] S.R. Allmaras, F.T. Johnson, and P.R. Spalart. Modifications and clarifications for the implementation of the Spalart-Allmaras turbulence model. Seventh International Conference on Computational Fluid Dynamics (ICCFD7) 1902, 2012.

[6] Douglas N. Arnold, Franco Brezzi, Bernardo Cockburn, and L. Donatella Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39(5):1749–1779, 2002.

[7] Timothy J. Baker. Mesh adaptation strategies for problems in fluid dynamics. *Finite Elements in Analysis and Design*, 25(3-4):243–273, 1997.

[8] S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith. The portable extensible toolkit for scientific computing (PETSc), release 3.1, 2010. `http://www.mcs.anl.gov/petsc/petsc-as/index.html`.

[9] Pinhas Bar-Yoseph and David Elata. An efficient L2 Galerkin finite element method for multi-dimensional non-linear hyperbolic systems. *International Journal for Numerical Methods in Engineering*, 29:1229–1245, 1990.

[10] Andrew R. Barron. Approximation and estimation bounds for artificial neural networks. *Machine Learning*, 14(1):115–133, 1994.

[11] Garrett E. Barter. *Shock Capturing with PDE-Based Artificial Viscosity for an Adaptive Higher–Order Discontinuous Galerkin Finite Element Method*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2008.

[12] Timothy Barth and Mats Larson. A posteriori error estimates for higher order Godunov finite volume methods on unstructured meshes. In R. Herban and D. Kröner, editors, *Finite Volumes for Complex Applications III*, pages 41–63, London, 2002. Hermes Penton.

[13] Timothy J. Barth. Space-time error representation and estimation in Navier-Stokes calculations. In Stavros C. Kassinos, Carlos A. Langer, Gianluca Iaccarino, and Parviz Moin, editors, *Complex Effects in Large Eddy Simulations*, pages 29–48. Springer Berlin Heidelberg, Lecture Notes in Computational Science and Engineering Vol 26, 2007.

[14] F. Bassi and S. Rebay. A high–order discontinuous finite element method for the numerical solution of the compressible Navier-Stokes equations. *Journal of Computational Physics*, 131(1–2):267–279, 1997.

[15] F. Bassi and S. Rebay. GMRES discontinuous Galerkin solution of the compressible Navier-Stokes equations. In Bernardo Cockburn, George Karniadakis, and Chi-Wang Shu, editors, *Discontinuous Galerkin Methods: Theory, Computation and Applications*, pages 197–208. Springer, Berlin, 2000.

[16] F. Bassi and S. Rebay. Numerical evaluation of two discontinuous Galerkin methods for the compressible Navier-Stokes equations. *International Journal for Numerical Methods in Fluids*, 40:197–207, 2002.

[17] R. Becker and R. Rannacher. A feed-back approach to error control in finite element methods: Basic analysis and examples. *East-West Journal of Numerical Mathematics*, 4(4):237–264, 1996.

[18] Anca Belme, Alain Dervieux, and Frédéric Alauzet. Time accurate anisotropic goal-oriented mesh adaptation for unsteady flows. *Journal of Computational Physics*, 231(19):6323–6348, 2012.

[19] M.J. Berger and A. Jameson. Automatic adaptive grid refinement for the Euler equations. *AIAA Journal*, 23:561–568, 1985.

[20] Kim S. Bey and J. Tinsley Oden. *hp*-version discontinuous Galerkin methods for hyperbolic conservation laws. *Computer Methods in Applied Mechanics and Engineering*, 133:259–286, 1996.

[21] Patrick J. Blonigan, Steven A. Gomez, and Qiqi Wang. Least squares shadowing for sensitivity analysis of turbulent fluid flows. AIAA Paper 2014–1426, 2014.

[22] H. Borouchaki, P. George, F. Hecht, P. Laug, and E Saltel. Mailleur bidimensionnel de Delaunay gouverné par une carte de métriques. Partie I: Algorithmes. INRIA-Rocquencourt, France. Tech Report No. 2741, 1995.

[23] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[24] F. Brezzi, L.D. Marini, and E. Süli. Discontinuous Galerkin methods for first-order hyperbolic problems. *Mathematical Models and Methods in Applied Sciences*, 14:1893–1903, 2004.

[25] Nicholas K. Burgess and Dimitri J. Mavriplis. An *hp*-adaptive discontinuous Galerkin, solver for aerodynamic flows on mixed-element meshes. AIAA Paper 2011-490, 2011.

[26] Gustavo C. Buscaglia and Enzo A. Dari. Anisotropic mesh optimization and its application in adaptivity. *International Journal for Numerical Methods in Engineering*, 40(22):4119–4136, November 1997.

[27] P. J. Capon and P. K. Jimack. On the adaptive finite element solution of partial differential equations using h-r refinement. Technical Report 96.03, University of Leeds, School of Computing, 1996.

[28] James Case. Breakthrough in conformal mapping. *SIAM News*, 41(1), 2008.

[29] J.R. Cash. The integration of stiff initial value problems in ODEs using modified extended backward differentiation formulae. *Computers & mathematics with applications*, 9(5):645–657, 1983.

[30] M. J. Castro-Diaz, F. Hecht, B. Mohammadi, and O. Pironneau. Anisotropic unstructured mesh adaptation for flow simulations. *International Journal for Numerical Methods in Fluids*, 25(4):475–491, 1997.

[31] Marco A. Ceze and Krzysztof J. Fidkowski. Output-driven anisotropic mesh adaptation for viscous flows using discrete choice optimization. AIAA Paper 2010-0170, 2010.

[32] Marco A. Ceze and Krzysztof J. Fidkowski. An anisotropic hp-adaptation framework for functional prediction. *AIAA Journal*, 51:492–509, 2013.

[33] Guodong Chen and Krzysztof J. Fidkowski. Discretization error control for constrained aerodynamic shape optimization. *Journal of Computational Physics*, 387:163–185, 2019.

[34] Bernardo Cockburn and Chi-Wang Shu. The local discontinuous Galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[35] Bernardo Cockburn and Chi-Wang Shu. Runge-Kutta discontinuous Galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16(3):173–261, 2001.

[36] L. Demkowicz, W. Rachowicz, and Ph. Devloo. A fully automatic hp-adaptivity. *Journal of Scientific Computing*, 17:117–142, 2002.

[37] Bruno Despres. Lax theorem and finite volume schemes. *Mathematics of Computation*, 73(247):1203–1234, 2003.

[38] Julien Dompierre, Marie-Gabrielle Vallet, Yves Bourgault, Michel Fortin, and Wagdi G. Habashi. Anisotropic mesh adaptation: towards user-independent, mesh-independent and solver-independent CFD. Part III: Unstructured meshes. *International Journal for Numerical Methods in Fluids*, 39:675–702, 2002.

[39] Thomas D. Economon, Francisco Palacios, and Juan J. Alonso. Unsteady continuous adjoint approach for aerodynamic design on dynamic meshes. *AIAA Journal*, 53(9):2437–2453, 2015.

[40] K. J. Fidkowski, M. A. Ceze, and P. L. Roe. Entropy-based drag error estimation and mesh adaptation in two dimensions. *AIAA Journal of Aircraft*, 49(5):1485–1496, September-October 2012.

[41] K. J. Fidkowski and D. L. Darmofal. A triangular cut-cell adaptive method for high-order discretizations of the compressible Navier-Stokes equations. *Journal of Computational Physics*, 225:1653–1672, 2007.

[42] Krzysztof J. Fidkowski. *A Simplex Cut-Cell Adaptive Method for High–order Discretizations of the Compressible Navier-Stokes Equations*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2007.

[43] Krzysztof J. Fidkowski. An output-based dynamic order refinement strategy for unsteady aerodynamics. AIAA Paper 2012-77, 2012.

[44] Krzysztof J. Fidkowski. Output-based error estimation and mesh adaptation for steady and unsteady flow problems. In H. Deconinck and T. Horvath, editors, *38^{th} Advanced CFD Lectures Series; Von Karman Institute for Fluid Dynamics (September 14–16 2015)*. von Karman Institute for Fluid Dynamics, 2015.

[45] Krzysztof J. Fidkowski. Output-based space-time mesh optimization for unsteady flows using continuous-in-time adjoints. *Journal of Computational Physics*, 341(15):258–277, July 2017.

[46] Krzysztof J. Fidkowski and David L. Darmofal. An adaptive simplex cut-cell method for discontinuous Galerkin discretizations of the Navier-Stokes equations. AIAA Paper 2007-3941, 2007.

[47] Krzysztof J. Fidkowski and David L. Darmofal. Review of output-based error estimation and mesh adaptation in computational fluid dynamics. *AIAA Journal*, 49(4):673–694, 2011.

[48] Krzysztof J. Fidkowski and Yuxing Luo. Output-based space-time mesh adaptation for the compressible Navier-Stokes equations. *Journal of Computational Physics*, 230:5753–5773, 2011.

[49] Krzysztof J. Fidkowski, Todd A. Oliver, James Lu, and David L. Darmofal. *p*-Multigrid solution of high–order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations. *Journal of Computational Physics*, 207:92–113, 2005.

[50] Krzysztof J. Fidkowski and Philip L. Roe. An entropy adjoint approach to mesh refinement. *SIAM Journal on Scientific Computing*, 32(3):1261–1287, 2010.

[51] L. Formaggia, S. Micheletti, and S. Perotto. Anisotropic mesh adaptation with applications to CFD problems. In H. A. Mang, F. G. Rammerstorfer, and J. Eberhardsteiner, editors, *Fifth World Congress on Computational Mechanics*, Vienna, Austria, July 7-12 2002.

[52] Luca Formaggia and Simona Perotto. New anisotropic a priori error estimates. *Numerische Mathematik*, 89(4):641–667, 2001.

[53] Luca Formaggia, Simona Perotto, and Paolo Zunino. An anisotropic a posteriori error estimate for a convection-diffusion problem. *Computing and Visualization in Science*, 4:99–104, 2001.

[54] Lori A. Freitag and Carl Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21):3979–4002, 1997.

[55] Pascal Jean Frey and Paul-Louis George. *Mesh Generation: Application to Finite Elements, Second Edition*. Wiley, New Jersey, 2010.

[56] Neal T. Frink. 3rd AIAA CFD drag prediction workshop gridding guidelines. NASA Langley, 2007. `http://aiaa-dpw.larc.nasa.gov/Workshop3/gridding_guidelines.html`.

[57] Neal T. Frink. Test case results from the 3rd AIAA drag prediction workshop. NASA Langley, 2007. `http://aiaa-dpw.larc.nasa.gov/Workshop3/workshop3.html`.

[58] M. B. Giles and N. A. Pierce. Adjoint equations in CFD: duality, boundary conditions and solution behavior. AIAA Paper 97-1850, 1997.

[59] M. B. Giles and E. Süli. Adjoint methods for PDEs: a posteriori error analysis and postprocessing by duality. In *Acta Numerica*, volume 11, pages 145–236, 2002.

[60] M.B. Giles and N.A. Pierce. Analytic adjoint solutions for the quasi-one-dimensional Euler equations. *Journal of Fluid Mechanics*, 426:327–345, 2001.

[61] Andreas Griewank and Andrea Walther. Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, 2000.

[62] W. Gui and I. Babuška. The h, p, and $h-p$ versions of the finite element method in 1 dimension. Part III. the adaptive $h-p$ version. *Numerische Mathematik*, 49(6):659–683, 1986.

[63] Wagdi G. Habashi, Julien Dompierre, Yves Bourgault, Djaffar Ait-Ali-Yahia, Michel Fortin, and Marie-Gabrielle Vallet. Anisotropic mesh adaptation: towards user-independent, mesh-independent and solver-independent CFD. Part I: general principles. *International Journal for Numerical Methods in Fluids*, 32(6):725–744, 2000.

[64] K. Harriman, D. Gavaghan, and E. Süli. The importance of adjoint consistency in the approximation of linear functionals using the discontinuous Galerkin finite element method. Technical Report Technical Report NA 04/18, Oxford University Computing Lab Numerical Analysis Group, 2004.

[65] K. Harriman, P. Houston, B. Senior, and Endre Süli. hp-version discontinuous Galerkin methods with interior penalty for partial differential equations with nonnegative characteristic form. Technical Report Technical Report NA 02/21, Oxford University Computing Lab Numerical Analysis Group, 2002.

[66] R. Hartmann and P. Houston. Error estimation and adaptive mesh refinement for aerodynamic flows. In H. Deconinck, editor, *36^{th} CFD/ADIGMA course on hp-adaptive and hp-multigrid methods: VKI Lecture Series 2010-01 (Oct. 26-30, 2009)*. von Karman Institute for Fluid Dynamics, 2010.

[67] Ralf Hartmann. Adjoint consistency analysis of discontinuous Galerkin discretizations. *SIAM Journal on Numerical Analysis*, 45(6):2671–2696, 2007.

[68] Ralf Hartmann and Paul Houston. Adaptive discontinuous Galerkin finite element methods for the compressible Euler equations. *Journal of Computational Physics*, 183(2):508–532, 2002.

[69] F. Hecht. BAMG: Bidimensional anisotropic mesh generator. INRIA–Rocquencourt, France, 1998. `www.freefem.org`.

[70] V. Heuveline and R. Rannacher. Duality-based adaptivity in the *hp*-finite element method. *Journal of Numerical Mathematics*, 11(2):95–113, 2003.

[71] Joel Ho and Alastair West. Field inversion and machine learning for turbulence modelling applied to three-dimensional separated flows. AIAA Paper 2021–2903, 2021.

[72] Jonathan R. Holland, James D. Baeder, and Karthik Duraisamy. Towards integrated field inversion and machine learning with embedded neural networks for RANS modeling. AIAA Paper 2019-1884, 2019.

[73] P. Houston, R. Hartmann, and E. Süli. Adaptive discontinuous Galerkin finite element methods for compressible fluid flows. In M. Baines, editor, *Numerical Methods for Fluid Dynamics VII, ICFD*, pages 347–353, 2001.

[74] P. Houston, B. Senior, and E. Süli. Sobolev regularity estimation for *hp*-adaptive finite element methods. In F. Brezzi, A. Buffa, S. Corsaro, and A. Murli, editors, *Numerical Mathematics and Advanced Applications*, pages 619–644. Springer-Verlag, 2003.

[75] P. Houston and E. Süli. *hp*-adaptive discontinuous Galerkin finite element methods for first-order hyperbolic problems. *SIAM Journal on Scientific Computing*, 23(4):1226–1252, 2001.

[76] Paul Houston, Emmanuil H. Georgoulis, and Edward Hall. Adaptivity and a posteriori error estimation for DG methods on anisotropic meshes. In G. Lube and G. Rapin, editors, *Proceedings of the International Conference on Boundary and Interior Layers (BAIL)*. University of Göttingen, 2006.

[77] Paul Houston, Bill Senior, and Endre Süli. hp-Discontinuous Galerkin finite element methods for hyperbolic problems: Error analysis and adaptivity. *International Journal for Numerical Methods in Fluids*, 40:153–169, 2002.

[78] Paul Houston and Endre Süli. A note on the design of hp-adaptive finite element methods for elliptic partial differential equations. *Computer Methods in Applied Mechanics and Engineering*, 194:229–243, 2005.

[79] Florian Jäckel. A closed-form correction for the Spalart-Allmaras turbulence model for separated flow. AIAA Paper 2022–0462, 2022.

[80] Hans Johansen and Philip Colella. A Cartesian grid embedded boundary method for Poisson's equation on irregular domains. *Journal of Computational Physics*, 147:60–85, 1998.

[81] Claes Johnson. On computability and error control in CFD. *International Journal for Numerical Methods in Fluids*, 20:777–788, 1995.

[82] Hamid R. Karbasian and Brian C. Vermeire. Gradient-free aerodynamic shape optimization using large eddy simulation. *Computers and Fluids*, 232:105185, 2022.

[83] Steven M. Kast and Krzysztof J. Fidkowski. Output-based mesh adaptation for high order Navier-Stokes simulations on deformable domains. *Journal of Computational Physics*, 252(1):468–494, 2013.

[84] C.M. Klaij, J.J.W. van der Vegt, and H. van der Ven. Space-time discontinuous Galerkin method for the compressible Navier-Stokes equations. *Journal of Computational Physics*, 217:589–611, 2006.

[85] Lilia Krivodonova. Limiters for high-order discontinuous Galerkin methods. *Journal of Computational Physics*, 226(1):879–896, 2007.

[86] Johan Larsson and Qiqi Wang. The prospect of using large eddy and detached eddy simulations in engineering design, and the research required to get there. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2022):20130329, 2014.

[87] Tobias Leicht and Ralf Hartmann. Anisotropic mesh refinement for discontinuous galerkin methods in two-dimensional aerodynamic flow simulations. *International Journal for Numerical Methods in Fluids*, 56:2111–2138, 2008.

[88] Yingjie Liu, Chi-Wang Shu, Eitan Tadmor, and Mengping Zhang. Central discontinuous Galerkin methods on overlapping cells with a nonoscillatory hierarchical reconstruction. *SIAM Journal on Numerical Analysis*, 45(6):2442–2467, 2007.

[89] Rainald Löhner. Extensions and improvements of the advancing-front grid generation technique. *Communications in Numerical Methods in Engineering*, 12(10):683–702, 1996.

[90] Robert B. Lowrie, Philip L. Roe, and Bram van Leer. Properties of space-time discontinuous Galerkin. Los Alamos Technical Report LA-UR-98-5561, 1998.

[91] James Lu. *An a Posteriori Error Control Framework for Adaptive Precision Optimization Using Discontinuous Galerkin Finite Element Method*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2005.

[92] Hong Luo, Joseph D. Baum, and Rainald Löhner. A hermite WENO-based limiter for discontinuous Galerkin method on unstructured grids. AIAA Paper 2007-0510, 2007.

[93] Karthik Mani and Dimitri J. Mavriplis. Error estimation and adaptation for functional outputs in time-dependent flow problems. *Journal of Computational Physics*, 229:415–440, 2010.

[94] Joaquim R. R. A. Martins and Andrew B. Lambe. Multidisciplinary design optimization: a survey of architectures. *AIAA Journal*, 51(9):2049–2075, 2013.

[95] D. J. Mavriplis. Adaptive mesh generation for viscous flows using Delaunay triangulation. *Journal of Computational Physics*, 90:271–291, 1990.

[96] D. Scott McRae. r-Refinement grid adaptation algorithms and issues. *Computer Methods in Applied Mechanics and Engineering*, 2000(4):1161–1182, 189.

[97] D. Meidner and B. Vexler. Adaptive space-time finite element methods for parabolic optimization problems. *SIAM Journal on Control Optimization*, 46(1):116–142, 2007.

[98] Peter K. Moore. Applications of lobatto polynomials to an adaptive finite element method: A posteriori error estimates for hp-adaptivity and grid-to-grid interpolation. *Numerische Mathematik*, 94:367–401, 2003.

[99] Siva K. Nadarajah and Antony Jameson. A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. AIAA Paper 2000-0667, 2000.

[100] Marian Nemec and Michael J. Aftosmis. Error estimation and adaptive refinement for embedded-boundary Cartesian meshes. AIAA Paper 2007-4187, 2007.

[101] Marian Nemec, Michael J. Aftosmis, and Mathias Wintzer. Adjoint-based adaptive mesh refinement for complex geometries. AIAA Paper 2008-0725, 2008.

[102] J. Von Neumann and R. D. Richtmyer. A method for the numerical calculation of hydrodynamic shocks. *Journal of Applied Physics*, 21:232–237, 1950.

[103] N.C. Nguyen, J. Peraire, and B. Cockburn. An implicit high-order hybridizable discontinuous Galerkin, method for linear convection-diffusion equations. *Journal of Computational Physics*, 228:3232–3254, 2009.

[104] Angxiu Ni and Qiqi Wang. Sensitivity analysis on chaotic dynamical systems by non-intrusive least squares shadowing (NILSS). *Journal of Computational Physics*, 347:56–77, 2017.

[105] Angxiu Ni, Qiqi Wang, Pablo Fernández, and Chaitanya Talnikar. Sensitivity analysis on chaotic dynamical systems by finite difference non-intrusive least squares shadowing (FD-NILSS). *Journal of Computational Physics*, 394:615–631, 2019.

[106] F. Nobile, R. Tempone, and C.G. Webster. A sparse grid stochastic collocation method for partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, 46(5):2309–2345, 2008.

[107] Vivek Ojha, Krzysztof J. Fidkowski, and Carlos E. S. Cesnik. Adaptive mesh refinement for fluid-structure interaction simulations. AIAA Paper 2021–0731, 2021.

[108] Todd A. Oliver. *A High–order, Adaptive, Discontinuous Galerkin, Finite Elemenet Method for the Reynolds-Averaged Navier-Stokes Equations*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2008.

[109] Todd A. Oliver and David L. Darmofal. Impact of turbulence model irregularity on high–order discretizations. AIAA Paper 2009-953, 2009.

[110] Doug Pagnutti and Carl Ollivier-Gooch. A generalized framework for high order anisotropic mesh adaptation. *Computers and Structures*, 87(11-12):670 – 679, 2009.

[111] Eric J. Parish and Karthik Duraisamy. A paradigm for data-driven predictive modeling using field inversion and machine learning. *Journal of Computational Physics*, 305:758–774, 2016.

[112] M. A. Park. Adjoint-based, three-dimensional error prediction and grid adaptation. AIAA Paper 2002-3286, 2002.

[113] M. A. Park. Three–dimensional turbulent RANS adjoint–based error correction. AIAA Paper 2003-3849, 2003.

[114] Michael A. Park. *Anisotropic Output-Based Adaptation with Tetrahedral Cut Cells for Compressible Flows*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2008.

[115] Xavier Pennec, Pierre Fillard, and Nicholas Ayache. A Riemannian framework for tensor computing. *International Journal of Computer Vision*, 66(1):41–66, 2006.

[116] J. Peraire, N. C. Nguyen, and B. Cockburn. An embedded discontinuous Galerkin method for the compressible Euler and Navier-Stokes equations. AIAA Paper 2011-3228, 2011.

[117] J. Peraire and P.-O. Persson. The compact discontinuous Galerkin (CDG) method for elliptic problems. *SIAM Journal on Scientific Computing*, 2007.

[118] J. Peraire, M. Vahdati, K. Morgan, and O. C. Zienkiewicz. Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, 72(2):449–466, 1987.

[119] P.-O. Persson, J. Bonet, and J. Peraire. Discontinuous Galerkin solution of the Navier-Stokes equations on deformable domains. *Computer Methods in Applied Mechanics and Engineering*, 198:1585–1595, 2009.

[120] P.-O. Persson and J. Peraire. Sub-cell shock capturing for discontinuous Galerkin methods. AIAA Paper 2006-112, 2006.

[121] Olivier Pironneau. On optimum design in fluid mechanics. *Journal of Fluid Mechanics*, 64:97–110, 1974.

[122] W. Rachowicz, L. Demkowicz, and J.T. Oden. Toward a universal $h - p$ adaptive finite element strategy, part 3. design of $h - p$ meshes. *Computer Methods in Applied Mechanics and Engineering*, 77:181–212, 1989.

[123] R. Rannacher. Adaptive Galerkin finite element methods for partial differential equations. *Journal of Computational and Applied Mathematics*, 128:205–233, 2001.

[124] S. Ashwin Renganathan, Romit Maulik, and Jai Ahuja. Enhanced data efficiency using deep neural networks and Gaussian processes for aerodynamic design optimization. *Aerospace Science and Technology*, 111:106522, 2021.

[125] Thomas Richter. A posteriori error estimation and anisotropy detection with the dual-weighted residual method. *International Journal for Numerical Methods in Fluids*, 62:90–118, 2010.

[126] Antonio Rubino, Salvatore Vitale, Piero Colonna, and Matteo Pini. Fully-turbulent adjoint method for the unsteady shape optimization of multi-row turbomachinery. *Aerospace Science and Technology*, 106:106132, 2020.

[127] C.L. Rumsey, G.N. Coleman, and L.Wang. In search of data-driven improvements to RANS models applied to separated flows. AIAA Paper 2022–0937, 2022.

[128] E. Schall, D. Leservoisier, A. Dervieux, and B. Koobus. Mesh adaptation as a tool for certified computational aerodynamics. *International Journal for Numerical Methods in Fluids*, 45(2):179–196, 2004.

[129] Michael Schmich and Boris Vexler. Adaptivity with dynamic meshes for space-time finite element discretizations of parabolic equations. *SIAM Journal on Scientific Computing*, 30(1):369–393, 2008.

[130] R. Schneider and P. K. Jimack. Toward anisotropic mesh adaptation based upon sensitivity of a posteriori estimates. Technical Report 2005.03, University of Leeds, School of Computing, 2005.

[131] Khosro Shahbazi. An explicit expression for the penalty parameter of the interior penalty method. *Journal of Computational Physics*, 205:401–407, 2005.

[132] J.R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22:21–74, 2002.

[133] Yukiko S. Shimizu and Krzysztof J. Fidkowski. Output error estimation for chaotic flows. AIAA Paper 2016-3806, 2016.

[134] Kunibert G. Siebert. An a posteriori error estimator for anisotropic refinement. *Numerische Mathematik*, 73:373–398, 1996.

[135] Anand Pratap Singh, Shivaji Medida, and Karthik Duraisamy. Machine-learning-augmented predictive modeling of turbulent separated flows over airfoils. *AIAA Journal*, 55(7):2215–2227, 2017.

[136] Anand Pratap Singh, Shaowu Pan, and Karthik Duraisamy. Characterizing and improving predictive accuracy in shock-turbulent boundary layer interactions using data-driven models. AIAA Paper 2017–0314, 2017.

[137] P. Solín and L. Demkowicz. Goal-oriented hp-adaptivity for elliptic problems. *Computer Methods in Applied Mechanics and Engineering*, 193:449–468, 2004.

[138] Philippe Spalart and Michael L. Shur. On the sensitization of turbulence models to rotation and curvature. *Aerospace Science and Technology*, 1(5):297–302, 1997.

[139] Philippe R. Spalart and Steven R. Allmaras. A one-equation turbulence model for aerodynamic flows. *La Recherche Aérospatiale*, (1):5–21, 1994.

[140] Shuyu Sun and Mary Wheeler. Mesh adaptation strategies for discontinuous Galerkin methods applied to reactive transport problems. In H.W. Chu, M. Savoie, and B. Sanchez, editors, *International Conference on Computing, Communication and Control Technologies*, volume 1, pages 223–228, Austin, Texas, August 2004.

[141] Shuyu Sun and Mary F. Wheeler. Anisotropic and dynamic mesh adaptation for discontinuous Galerkin methods applied to reactive transport. Technical Report 05-15, ICES, 2005.

[142] Barna A. Szabo. Estimation and control of error based on p convergence. In I. Babuška, O. C. Zienkiewicz, J. Gago, and E. R. de Oliveira, editors, *Accuracy Estimates and Adaptive Refinements in Finite Element Computations*, pages 61–78. John wiley & Sons Ltd., 1986.

[143] Jun Tao and Gang Sun. Application of deep learning based multi-fidelity surrogate model to robust aerodynamic design optimization. *Aerospace Science and Technology*, 92:722–737, 2019.

[144] J.J.W. van der Vegt. Space-time discontinuous Galerkin finite element method with dynamic grid motion for inviscid compressible flows. National Aerospace Laboratory NLR-TP-98239, 1998.

[145] H. van der Ven and J.J.W. van der Vegt. Space-time discontinuous Galerkin fintie element method with dynamic grid motion for inviscid compressible flows II. Efficient flux quadrature. *Computer Methods in Applied Mechanics and Engineering*, 191:4747–4780, 2002.

[146] Bram van Leer, Marcus Lo, and Marc van Raalte. A discontinuous Galerkin method for diffusion based on recovery. AIAA Paper 2007-4083, 2007.

[147] D. A. Venditti and D. L. Darmofal. Anisotropic grid adaptation for functional outputs: application to two-dimensional viscous flows. *Journal of Computational Physics*, 187(1):22–46, 2003.

[148] V. Venkatakrishnan, S. R. Allmaras, D. S. Kamenetskii, and F. T. Johnson. Higher order schemes for the compressible Navier-Stokes equations. AIAA Paper 2003-3987, 2003.

[149] Qiqi Wang. Forward and adjoint sensitivity computation of chaotic dynamical systems. *Journal of Computational Physics*, 235:1–13, 2013.

[150] Qiqi Wang. Convergence of the least squares shadowing method for computing derivative of ergodic averages. *SIAM Journal on Numerical Analysis*, 52(1):156–170, 2014.

[151] Qiqi Wang, Rui Hu, and Patrick Blonigan. Least squares shadowing sensitivity analysis of chaotic limit cycle oscillations. *Journal of Computational Physics*, 267:210–224, 2014.

[152] Yuqi Wang, Tianyuan Liu, Di Zhang, and Yonghui Xie. Dual-convolutional neural network based aerodynamic prediction and multi-objective optimization of a compact turbine rotor. *Aerospace Science and Technology*, 116:106869, 2021.

[153] Z.J. Wang, Krzysztof Fidkowski, Remi Abgrall, Francesco Bassi, Doru Caraeni, Andrew Cary, Herman Deconinck, Ralf Hartmann, Koen Hillewaert, H.T. Huynh, Norbert Kroll, Georg May, Per-Olof Persson, Bram van Leer, and Miguel Visbal. High-order CFD methods: Current status and perspective. *International Journal for Numerical Methods in Fluids*, 72:811–845, 2013.

[154] Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science*. PhD thesis, Harvard University, 1974.

[155] William A. Wood and William L. Kleb. On multi-dimensional unstructured mesh adaptation. AIAA Paper 99-3254, 1999.

[156] Guoping Xia, Ding Li, and Charles L. Merkle. Anisotropic grid adaptation on unstructured meshes. AIAA Paper 2001-0443, 2001.

[157] Zhiliang Xu, Yingjie Liu, and Chi-Wang Shu. Hierarchical reconstruction for discontinuous galerkin methods on unstructured grids with a WENO-type linear reconstruction and partial neighboring cells. *Journal of Computational Physics*, 228(6):2194–2212, 2009.

[158] Chongyang Yan, Haoran Li, Yufei Zhang, and Haixin Chen. Data-driven turbulence modeling in separated flows considering physical mechanism analysis, 2021.

[159] Xinghui Yan, Jihong Zhu, Minchi Kuang, and Xiangyang Wang. Aerodynamic shape optimization using a novel optimizer based on machine learning techniques. *Aerospace Science and Technology*, 86:826–835, 2021.

[160] M. Yano and D.L. Darmofal. An optimization framework for anisotropic simplex mesh adaptation: Application to aerodynamic flows. AIAA Paper 2012-0079, 2012.

[161] M. Yano, J.M. Modisette, and D.L. Darmofal. The importance of mesh adaptation for higher-order discretizations of aerodynamics flows. AIAA Paper 2011-3852, 2011.

[162] Masayuki Yano. *An Optimization Framework for Adaptive Higher-Order Discretizations of Partial Differential Equations on Anisotropic Simplex Meshes*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2012.

[163] Linyang Zhu, Weiwei Zhang, Xuxiang Sun, Yilang Liu, and Xianxu Yuan. Turbulence closure for high Reynolds number airfoil flows by deep neural networks. *Aerospace Science and Technology*, 110:106452, 2021.

