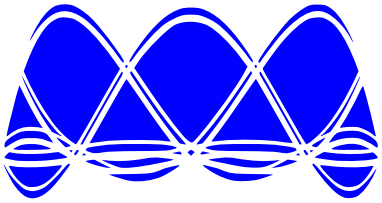
Spectral Element Libraries in Fortran



SELF *(v 3.0)*

Technical Documentation

Joseph Schoonover, Yuliana Zamora, Jenniffer Estrada

The SELF and this documentation were produced in part under the support of Florida State University and the National Science Foundation through Grant OCE-1049131 during 2015 and in part the support of the Center for Nonlinear Studies and the Department of Energy through the LANL/LDRD program in 2016.

This documentation has been cleared to be released to the general public under the LA-UR # .....

Contents

Introduction

As high performance computer architectures continue to advance, scientists and engineers continue to adapt algorithms and software to make effective use of emerging platforms. Numerical simulation of physical systems remain a cornerstone of research and development. To simulate physical systems, a mathematical model is first devised, and typically takes the form of a partial differential equation (PDE). It is often the case that analytical solutions to these models are intractable and, instead, approximate solutions are sought using numerical methods.

Spectral Element Methods (SEMs) result in discretizations that result in many independent dense matrix-matrix operations. The local nature of SEM operations naturally results in SIMD parallelism that can be favorable for many flavors of parallelization (MPI, OpenMP, OpenACC). Aside from the computational pro's of SEMs, they can naturally be applied to complicated geometries, and have the ability to obtain exponential convergence in the discretization errors. The (relatively) recent introduction of SEMs to domain scientists is partly responsible for the small size of the community that makes use of them. In comparison to the more familiar finite-difference and finite-volume methods, the algorithm structure of SEMs appears rather foreign and comes with a learning curve, making it a challenge for seasoned scientists and engineers to effectively apply them to their research.

The Spectral Element Libraries in Fortran (SELF) provide scientists and researchers with a set of data structures and routines to rapidly construct solvers for partial differential equations using SEMs. This is accomplished by posing partial differential equations in a standard conservation law form and developing routines to approximate differential operations of these generic equations. The SELF developers aim to provide high-end solvers to the scientific community that are parallelized using OpenMP, OpenACC, and MPI; this is an avenue of current exploration. Currently, the SELF offers solvers for the shallow-water equations (2-D) and the compressible Euler equations (3-D). Current work is focused on the application of the SELF routines

to turbulent compressible and incompressible fluid flows. We hope that, by using the SELF, researchers are able to (1) reduce the amount of coding needed to perform their research activities, (2) produce scientific results that are reproducible underneath a common framework, and (3) explore the benefits and limitations of SEMs to further this branch of research in numerical methods.

This document (the Technical Documentation) serves as documentation for the treatment of geometry, the mathematical framework of SEMs, implementation details within the SELF, and a description of the high-end solvers that are already provided. The treatment of an isoparametric unstructured mesh focuses on the necessary book-keeping for describing an unstructured mesh and the differential geometry required for curvilinear elements (see Chapter 2). The mathematical framework focuses on approximating solutions to PDEs with *Nodal* SEMs based on approximating solutions as piecewise polynomials (see Chapter 3). Practical implementation details describes the data-structures, routines, and support programs provided by the SELF and gives an overview of the software structure (see Chapter 9). Lastly, an appendix provides more detailed derivations and explanations of the mathematics where necessary.

The Unstructured Spectral Element Mesh

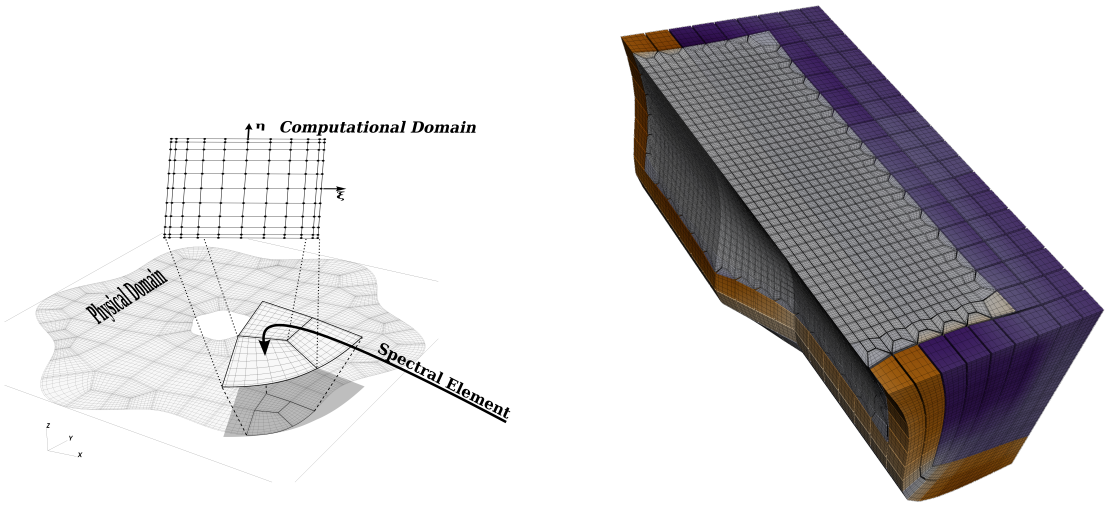


Figure 2.1: A schematic of a spectral element mesh in two-dimensions (left) and an example 3-D unstructured mesh of hexahedrals.

An unstructured spectral element mesh is comprised of an arrangement of logical quadrilateral (in 2-D) or hexahedral (3-D) elements. In the SELF, the elements are arranged so that they are non-overlapping and *conformal*, meaning that elements corners only intersect with other element corners and the polynomial degree is the same for each element (see Fig. 2.2). Each element is mapped to a reference computational domain using transfinite interpolation with linear blending(**?**). This allows each element to have curved sides and permits a high order representation of a model boundary. The elements are called *isoparametric* when the mapping is approximated by a polynomial interpolant that is the same order as the solution's interpolant. The mapping establishes metric terms that are necessary for computing derivatives in mapped coordinate systems. Together, the unstructured

domain decomposition with isoparametric elements allows for flexible and high order representation of complicated geometries. In the following sections, the unstructured mesh and its attributes are described followed by differential geometry used to calculate derivative operations in mapped coordinates.

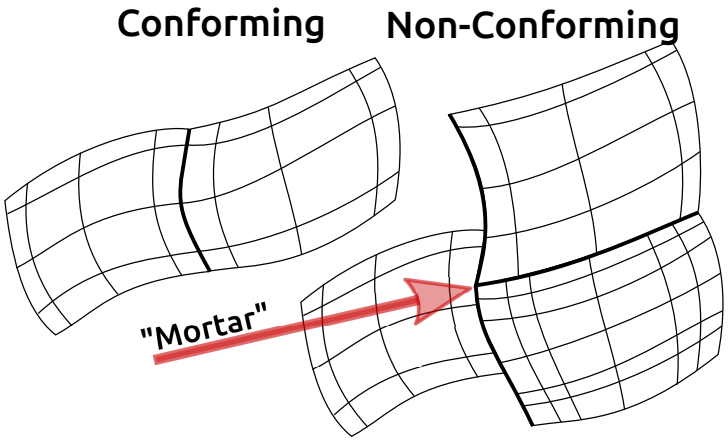


Figure 2.2: A schematic of conformal and non-conformal decompositions. In conformal decompositions, elements only have one neighbor across an edge or face, and all of the elements have the same polynomial degree. In non-conformal decompositions, elements can be "mortared" together, and the polynomial degree can vary from element to element. The SELF currently only supports conformal decompositions.

Mesh Primitives and Connectivity

An unstructured mesh can be described by a collection of nodes, which have a physical position and a unique integer identifier, and elements which are described by its corner nodes and its own unique integer identifier. This is the least amount of information required to describe an unstructured mesh. From this information, unique edges and faces (in 3-D) can be constructed. A summary of these mesh primitives is given in Table 2.1.

An example in 2-D

Figure 2.3 depicts a domain decomposition in 2-D with 8 nodes and 4 elements. Aside from the physical position of the nodes, the information about the mesh connectivity is encoded in the node

Table 2.1: A table of primitives needed to describe an unstructured mesh and the mesh connectivity.

Primitive	**Attributes**
Node	ID, x,y,z
Edge	ID, Node IDs (2), Element IDs, Element (local) sides
Face	ID, Node IDs (4), Element IDs, Element (local) sides
Element	ID, Node IDs (4,8), Neighbors

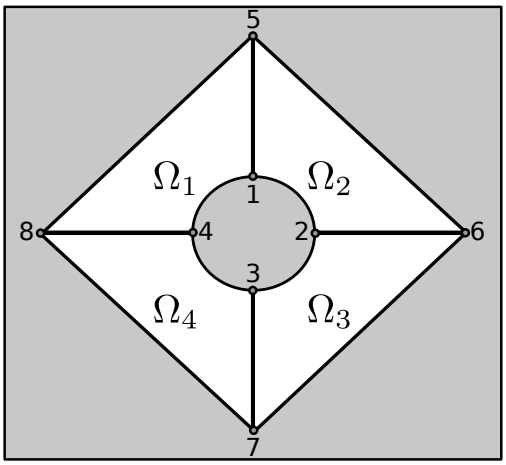


Figure 2.3: A simple schematic of a spectral element mesh. This schematic shows four spectral elements, with isoparametric (curved) sides.

IDs and the elements that they make up. Element 1 has corner nodes 4, 1, 5, 8. The element data structure for this element is

```
element(1) % ID      = 1
element(1) % nodeIDs = (/ 4, 1, 5, 8 /)
```

Similarly, the remaining elements are

```
element(2) % ID      = 2
element(2) % nodeIDs = (/ 1, 2, 6, 5 /)
element(3) % ID      = 3
element(3) % nodeIDs = (/ 3, 2, 6, 7 /)
element(4) % ID      = 4
element(4) % nodeIDs = (/ 4, 3, 7, 8 /)
```

Differential Geometry

Domain Decomposition

Spectral Element Discretizations

In the SELF, a conservation law is viewed in the general form

$$\vec{s}_t + \nabla \cdot \vec{f} = \vec{q}, \tag{3.1}$$

where $\vec{s}$ is a vector of *prognostic* variables (or solution), $\vec{f}$ is a vector of *conservative flux vectors* and $\vec{q}$ is a vector of *non-conservative source terms*. Some examples include

1. Linear advection-diffusion :

$$\vec{s} = c \text{ (scalar)} \tag{3.2a}$$

$$\vec{f} = \vec{u}c - \kappa \nabla c \tag{3.2b}$$

$$\vec{q} = 0 \text{ (scalar)} \tag{3.2c}$$

2. Inviscid Euler Equations :

$$\vec{s} = \begin{pmatrix} \rho u \\ \rho v \\ \rho w \\ \rho \\ \rho\theta \end{pmatrix} \tag{3.3a}$$

$$\vec{f} = \begin{pmatrix} \rho\vec{u}u + p\hat{x} \\ \rho\vec{u}v + p\hat{y} \\ \rho\vec{u}w + p\hat{z} \\ \rho\vec{u} \\ \vec{u}\rho\theta \end{pmatrix} \tag{3.3b}$$

$$\vec{q} = \vec{0} \tag{3.3c}$$

Figure 3.1: The first five Legendre polynomials are shown over the line segment $[-1, 1]$.

In practice, the conservative flux and the non-conservative source terms are known and will often depend on the solution. The goal is to approximate the solution at a later time given an initial estimate of the solution. For our purposes, the Spectral Element discretization deals only with the *spatial* discretization which converts the PDE (3.1) into a system of Ordinary Differential Equations (ODEs) that can be integrated forward in time with a suitable time integrator.

Interpolation, Integration, and Differentiation

Solutions to (3.1) are real-valued functions. Any real-valued function can represented as a linear combination of *basis functions*. The SELF uses Lagrange interpolating polynomials. In 1-D,

$$f(\xi) = \sum_{i=0}^{N} f_i l_i(\xi) \tag{3.4}$$

where f_i are the *nodal* values of the function at the interpolation nodes $\{\xi_i\}_{i=0}^{N}$.

Table 3.1: A summary of the types of quadrature points used in the SELF and their accuracy.

Gauss	2N+1	Endpoints not included
Gauss-Radau	2N	One endpoint included
Gauss-Lobatto	2N-1	Both endpoints included

In spectral element methods, integrals are often approximated by discrete quadrature. Discrete quadrature evaluates integrals by computing a weighted sum of function nodal values. The locations where the nodal values are obtained are the quadrature nodes, and the weights are the quadrature weights. Spectral element methods conveniently choose the interpolation nodes to be the quadrature nodes and the SELF provides options for the Legendre Gauss, Gauss-Radau,

Figure 3.2: A depiction of Gauss, Gauss-Lobatto, and Gauss-Radau quadrature nodes in one dimensions.

and Gauss-Lobatto quadratures. $\mathcal{O}$(N) quadrature requires (N+1) weights and nodal values. A summary of the quadrature accuracies is provided in Table 3.1. Gauss, Gauss-Radau, and Gauss-Lobatto quadratures provide exact integration rules for polynomials of degree 2N+1, 2N, and 2N-1, respectively.

Differentiation can be done using a "pseudo-spectral" approach that approximates the derivative of a function by taking a derivative of the interpolant (3.4).

DGSEM

A conservation law can be written in the form

$$\vec{s}_t + \nabla \cdot \vec{f} + \vec{q} = 0, \tag{3.5}$$

where $\vec{s}$ is a vector of *prognostic* variables, $\vec{f}$ is a vector of *conservative flux vectors* and $\vec{q}$ is a vector of *non-conservative source terms*.

The DGSEM discretizes (3.5) in its weak form. To obtain the weak form, (3.5) is weighted with a test function and integrated over the physical domain, call it Ω. To approximate the integrals in the weak form, the domain is first divided into non-overlapping elements (Ω^κ) . Integration is performed over each element and the solution and test function are permitted to be piecewise discontinuous across elements. These assumptions lead to the statement

$$\int_{\Omega^\kappa} (\vec{s}_t + \vec{q})\phi \, d\Omega^\kappa - \int_{\Omega^\kappa} \vec{f} \cdot \nabla\phi \, d\Omega^\kappa + \oint_{\partial\Omega^\kappa} \phi \vec{f} \cdot \hat{n} \, dA^\kappa = 0, \quad \forall\phi \in \mathbb{C}_0(\Omega_\kappa), \quad \kappa = 1, 2, ..., K \tag{3.6}$$

where $\mathbb{C}_0(\Omega^\kappa)$ is the space of functions that are continuous over Ω^κ.

Equation (3.6) assumes that the integration over each element is independent; this *compactness* results from allowing piecewise discontinuous solutions. The third term in (3.6) is the integral of the conservative flux over the element boundary. This is the only term that involves communication with other elements. For hyperbolic problems, this communication is only between *neighboring elements* which share a common node (1-D), edge (2-D), or face (3-D).

The formulation presented in (3.6) only requires that we know the geometry of each element and the connectivity of a collection of elements. This allows for the use of either structured or unstructured mesh frameworks. Additionally, the elements which comprise the mesh can have curvilinear geometry. Define the mapping from physical space $\vec{x}$ to computational space $\vec{\xi}$ using

$$\vec{x} = \vec{x}(\vec{\xi}). \tag{3.7}$$

Section **??** provides the details on the metric terms that are introduced along with the form of the divergence, gradient, and curl under such a mapping. For simplicity, the computational domain is formed from tensor products of intervals $[-1, 1]$ in each coordinate direction. This criteria restricts the elements to be logically segments (1-D), quadrilaterals (2-D), or hexahedrons (3-D). Under the mapping (3.7), the weak form (3.6) becomes

$$\int_{\Omega^\xi} (\tilde{s}_t^\kappa + \tilde{q}^\kappa)\phi \, d\Omega^\xi - \int_{\Omega^\xi} \tilde{f}^\kappa \cdot \nabla_{\vec{\xi}} \phi \, d\Omega^\xi + \oint_{\partial\Omega^\xi} \phi \tilde{f}^{\kappa,*} \cdot \hat{n}^\xi \, dA^\xi = 0, \;\; \forall \phi \in \mathbb{C}^0(\Omega^\xi) \tag{3.8}$$

where $\tilde{s} = J^\kappa \vec{s}$, $\tilde{q} = J^\kappa \vec{q}$, $\tilde{f} = (J^\kappa \vec{a}^{\kappa,i} \cdot \vec{f})\hat{a}^{\kappa,i}$, J^κ is the Jacobian of the transformation over the κ^{th} element, and $\vec{a}^{\kappa,i}$ are the contravariant basis vectors associated with the transformation of element κ (the $\hat{a}$ denotes a unit vector).

Given a conservative flux, a non-conservative source, internal element metrics, and global element connectivity, the discrete algorithm solves (3.8) by approximating the integrands with interpolants and the integrals by discrete quadratures. First, the prognostic solution, source term,

conservative flux, and mapping are approximated by Lagrange interpolants of degree N.

$$\vec{s} \approx I^N(\vec{s}) = \sum_{i,j,k=0}^{N} \vec{S}_{i,j,k} l_i(\xi^1) l_j(\xi^2) l_k(\xi^3) \tag{3.9a}$$

$$\vec{q} \approx I^N(\vec{q}) = \sum_{i,j,k=0}^{N} \vec{Q}_{i,j,k} l_i(\xi^1) l_j(\xi^2) l_k(\xi^3) \tag{3.9b}$$

$$\vec{f} \approx I^N(\vec{f}) = \sum_{i,j,k=0}^{N} \vec{F}_{i,j,k} l_i(\xi^1) l_j(\xi^2) l_k(\xi^3) \tag{3.9c}$$

$$\vec{x} \approx I^N(\vec{x}) = \sum_{i,j,k=0}^{N} \vec{X}_{i,j,k} l_i(\xi^1) l_j(\xi^2) l_k(\xi^3) \tag{3.9d}$$

An additional simplification is to approximate the product of interpolants by an interpolant of degree N which incurs an additional aliasing error.

$$\tilde{s} \approx I^N(I^N(J) I^N(\vec{s})) = \tilde{S} = \sum_{i,j,k=0}^{N} (J\vec{S}_{i,j,k}) l_i(\xi^1) l_j(\xi^2) l_k(\xi^3) \tag{3.10a}$$

$$\tilde{q} \approx I^N(I^N(J) I^N(\vec{q})) = \tilde{Q} = \sum_{i,j,k=0}^{N} (J\vec{Q})_{i,j,k} l_i(\xi^1) l_j(\xi^2) l_k(\xi^3) \tag{3.10b}$$

$$\tilde{f}_{(i,n)} \approx I^N(I^N(Ja^i_{(n)}) I^N(\vec{f})) = \tilde{F}_{(i,n)} = \sum_{i,j,k=0}^{N} (Ja^i_{(n)} \vec{F}_{(n)})_{i,j,k} l_i(\xi^1) l_j(\xi^2) l_k(\xi^3) \tag{3.10c}$$

Last, the space of test functions ($\mathbb{C}_0$) is approximated by the $\mathbb{P}^N$, the space of polynomials of degree N. Thus, the test function ϕ is replaced by each of the Lagrange interpolating polynomials,

$$\phi_{m,n,p} = l_m(\xi^1) l_n(\xi^2) l_p(\xi^3) \tag{3.11}$$

With this, equation (3.8) becomes

$$\int_{\Omega^\xi} (\tilde{S}^\kappa_t + \tilde{Q}^\kappa) \phi_{m,n,p} \, d\Omega^\xi - \int_{\Omega^\xi} \tilde{F}^\kappa \cdot \nabla_\xi \phi_{m,n,p} \, d\Omega^\xi + \oint_{\partial\Omega^\xi} \phi_{m,n,p} \tilde{F}^{\kappa,*} \cdot \hat{n}^\xi \, dA^\xi = 0, \quad m,n,p = 0,1,...,N \tag{3.12}$$

The final step is to replace the integrals in (3.12) with discrete quadrature. For this we use the Legendre-Gauss quadrature, which yields exact integration for each term in (3.12). Additionally, the interpolation nodes are specified as the Legendre-Gauss nodes, which simplifies the integration. The algorithm is now presented in one, two, and three dimesnions.

Algorithm in One-Dimensions

In one-dimension, the global physical domain is partitioned into a set of line segments. The first term in (3.12) becomes

$$\int_{\Omega^\xi} \tilde{S}_t^\kappa \phi_m \, d\Omega^\xi = \sum_{\alpha=0}^{N} \left[\sum_{i=0}^{N} \left((\tilde{S}_i^\kappa)_t l_i(\xi_\alpha) \right) l_m(\xi_\alpha) \omega_\alpha \right] \tag{3.13}$$

where the ξ_α are the Legendre-Gauss nodes and the ω_m are the Legendre-Gauss weights. Since the interpolation nodes are equivalent to the quadrature nodes, (3.13) is greatly simplified by the use of the Kronecker-delta property of Lagrange interpolating polynomials,

$$l_i(\xi_m) = \delta_{i,m}. \tag{3.14}$$

This yields (3.13) as

$$\int_{\Omega^\xi} \tilde{S}_t^\kappa \phi_\alpha \, d\Omega^\xi = (\tilde{S}_\alpha^\kappa)_t \omega_\alpha. \tag{3.15}$$

Similarly,

$$\int_{\Omega^\xi} \tilde{Q}^\kappa \phi_\alpha \, d\Omega^\xi = \tilde{Q}_\alpha^\kappa \omega_\alpha. \tag{3.16}$$

The third term in (3.12) becomes

$$\int_{\Omega^\xi} \tilde{F}^\kappa \cdot \nabla_{\vec{\xi}} \phi_m \, d\Omega^\xi = \sum_{i=0}^{N} \tilde{F}_i^\kappa l'_m(\xi_i) \omega_i. \tag{3.17}$$

Lastly, the boundary flux becomes

$$\oint_{\partial\Omega^\xi} \phi_\alpha \tilde{F}^{\kappa,*} \cdot \hat{n}^\xi \, dA^\xi = \tilde{F}^{\kappa,*}(1) l_\alpha(1) - \tilde{F}^{\kappa,*}(-1) l_\alpha(-1). \tag{3.18}$$

Thus, the spatially discretized system can be written

$$\left(J_m \vec{S}_m \right)_t = - \left[\sum_{i=0}^{N} \hat{D}_{m,i} \tilde{F}_i + \left(\frac{l_m(1)}{w_m} \tilde{F}^*(1) - \frac{l_m(-1)}{w_m} \tilde{F}^*(-1) \right) \right] + J_m \vec{Q}_m; \qquad m = 0, 1, ..., N, \tag{3.19}$$

where

$$\hat{D}_{m,i} = -\frac{l'_m(\xi_i) w_i}{w_m^{\xi^1}} \tag{3.20}$$

is the DG-Derivative matrix.

Up to this point, the treatment of the boundary flux has been relatively vague. In practice, the boundary flux is calculated using an approximate Riemman solver. This uses the solution state from the neighboring elements to approximate the flux across the shared boundary. The specification of the Riemman solver depends on the PDE system. Because of this, a discussion of the boundary flux is delayed until specific PDE systems are discussed.

Algorithm in Two-Dimensions

$$\begin{aligned}\left(J_{m,n}\vec{S}_{m,n}\right)_t = &-\left[\sum_{i=0}^{N}\hat{D}_{m,i}^{(\xi^1)}\tilde{F}_{i,n}^{(\xi^1)} + \left(\frac{l_m(1)}{w_m^{(\xi^1)}}\tilde{F}^*(1,\xi_n^2) - \frac{l_m(-1)}{w_m^{(\xi^1)}}\tilde{F}^*(-1,\xi_n^2)\right)\cdot\hat{\xi^1}\right] \\ &-\left[\sum_{j=0}^{N}\hat{D}_{n,j}^{(\xi^2)}\tilde{F}_{m,j}^{(\xi^2)} + \left(\frac{l_n(1)}{w_n^{(\xi^2)}}\tilde{F}^*(\xi_m^1,1) - \frac{l_n(-1)}{w_n^{(\xi^2)}}\tilde{F}^*(\xi_m^1,-1)\right)\cdot\hat{\xi^2}\right] \\ &+ J_{m,n}\vec{Q}_{m,n}; \qquad m,n = 0,1,...,N\end{aligned} \tag{3.21}$$

Computing the divergence of the conservative flux in this framework can be viewed as calculating a sequence of derivatives in each computational direction. Two steps are required to compute the derivative in each direction. The first is an internal matrix-vector multiply, and the second is computing the weighted Riemann fluxes at the element boundaries. The latter is the only step which requires element-to-element communication.

Algorithm in Three-Dimensions

$$\begin{aligned}\left(J_{m,n,p}\vec{S}_{m,n,p}\right)_t = &-\left[\sum_{i=0}^{N}\hat{D}_{m,i}^{(\xi^1)}\tilde{F}_{i,n,p}^{(\xi^1)} + \left(\frac{l_m(1)}{w_m^{(\xi^1)}}\tilde{F}^*(1,\xi_n^2,\xi_p^3) - \frac{l_m(-1)}{w_m^{(\xi^1)}}\tilde{F}^*(-1,\xi_n^2,\xi_p^3)\right)\cdot\hat{\xi^1}\right] \\ &-\left[\sum_{j=0}^{N}\hat{D}_{n,j}^{(\xi^2)}\tilde{F}_{m,j,p}^{(\xi^2)} + \left(\frac{l_n(1)}{w_n^{(\xi^2)}}\tilde{F}^*(\xi_m^1,1,\xi_p^3) - \frac{l_n(-1)}{w_n^{(\xi^2)}}\tilde{F}^*(\xi_m^1,-1,\xi_p^3)\right)\cdot\hat{\xi^2}\right] \\ &-\left[\sum_{k=0}^{N}\hat{D}_{m,n,k}^{(\xi^3)}\tilde{F}_{m,n,k}^{(\xi^2)} + \left(\frac{l_p(1)}{w_p^{(\xi^3)}}\tilde{F}^*(\xi_m^1,\xi_n^2,1) - \frac{l_p(-1)}{w_p^{(\xi^3)}}\tilde{F}^*(\xi_m^1,\xi_n^2,-1)\right)\cdot\hat{\xi^3}\right] \\ &+ J_{m,n,p}\vec{Q}_{m,n,p}; \qquad m,n,p = 0,1,...,N\end{aligned} \tag{3.22}$$

Riemann Flux

This section needs some attention!

Regardless of which system we are solving, the DG approximation requires that we compute an estimate of the flux across an element's edge given the solution on either side of the edge. In general, the solution is discontinuous across the edges. Let $\vec{s}_L$ and $\vec{s}_R$ denote the solution to the "left" and to the "right" of the edge as depicted in Fig. 3.3. The goal is to compute the flux across an edge given the left and right states. The conservation law, (**??**), can be written

$$\vec{s}_t + \frac{\partial f^n}{\partial \vec{s}}\frac{\partial \vec{s}}{\partial n} = 0, \tag{3.23}$$

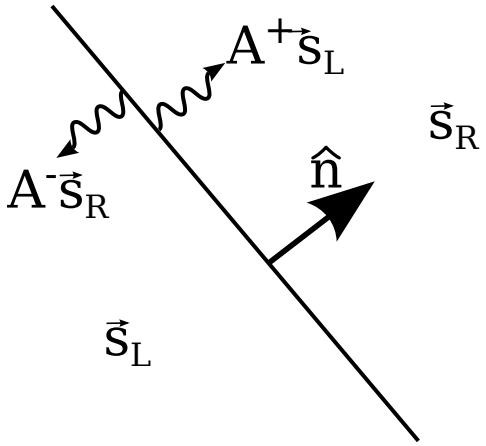


Figure 3.3: A depiction of the setup for computing the flux across an edge. The flux is split by upwinding the characteristic variables of the Jacobian matrix.

where, for the sake of exposition, the source term has been dropped. The flux in the edge-normal direction is $f^n = \vec{f} \cdot \hat{n}$ and the directional derivative of the solution is $\frac{\partial \vec{s}}{\partial n}$. For a short period of time, Δt,

$$\vec{s}_t + \frac{\partial f}{\partial \vec{s}}|_{t=t_0} \frac{\partial \vec{s}}{\partial n} = \mathcal{O}(\Delta t). \tag{3.24}$$

In (3.24), the *Jacobian* of the flux, $\frac{\partial f^n}{\partial \vec{s}}$, is evaluated at the fixed time $t = t_0$. For hyperbolic problems, like the shallow water equations, the Jacobian has real eigenvalues and can be diagonalized. Let

$$\frac{\partial f}{\partial \vec{s}}|_{t=t_0} = \boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{-1}, \tag{3.25}$$

define the diagonalization, where $\boldsymbol{P}$ is a matrix whose columns are the eigenvectors and $\boldsymbol{D}$ is a diagonal matrix whose diagonal elements are the corresponding eigenvalues of the Jacobian. Substituting (3.25) into (3.24) and multiplying on the left by $\boldsymbol{P}^{-1}$ gives

$$\vec{w}_t + \boldsymbol{D}\boldsymbol{P}^{-1} \frac{\partial \vec{s}}{\partial n} = \mathcal{O}(\Delta t), \tag{3.26}$$

where $\vec{w} = P^{-1}\vec{s}$ are the *characteristic* variables. Equation (3.26) can be rewritten, approximately as

$$\vec{w}_t + \boldsymbol{D} \frac{\partial \vec{w}}{\partial n} \approx \mathcal{O}(\Delta t). \tag{3.27}$$

where variations in the eigenvectors with n have been ignored. Equation (3.27) has approximate solutions

$$w^i = w_0^i(n - \lambda_i(t - t_0)) \tag{3.28}$$

where w^i and λ_i are the i^{th} eigenvector and eigenvalue, w_0^i is the characteristic variable at time $t = t_0$, and n is the physical distance normal to the edge. To evaluate the flux at the edge, we need to know the solution at the edge ($n = 0$). At time $t_0 + \Delta t$,

$$w^i(0, \Delta t) = w_0^i(-\lambda_i \Delta t) \tag{3.29}$$

so that if $\lambda_i > 0$, the solution depends on the state to the left of the edge, and if $\lambda_i < 0$, the solution depends on the initial state to the right of the edge. Because of this, we split the diagonalization into two components,

$$\frac{\partial f}{\partial \vec{s}}|_{t=t_0} = \boldsymbol{P}\boldsymbol{D}^+\boldsymbol{P}^{-1} + \boldsymbol{P}\boldsymbol{D}^-\boldsymbol{P}^{-1} = \boldsymbol{A}^+ + \boldsymbol{A}^-, \tag{3.30}$$

where $\boldsymbol{D}^+$ is the diagonal matrix with only positive eigenvalues and $\boldsymbol{D}^-$ is the diagonal matrix with only negative eigenvalues. The compact notation $\boldsymbol{A}^+$ and $\boldsymbol{A}^-$ is used for the Jacobian matrix associated with the splitting of the positive and negative eigenvalues.

Under similar assumptions used to obtain (3.28), the flux at the boundary can be approximated

$$\vec{f}^n \approx \vec{f}^{n,*} = A^+\vec{s}_L + A^-\vec{s}_R \tag{3.31}$$

It can be shown that

$$A^+ = \frac{A + |A|}{2} \tag{3.32a}$$

$$A^- = \frac{A - |A|}{2} \tag{3.32b}$$

so that (3.31) can be written

$$\vec{f}^{n,*} = \frac{1}{2}\left(\vec{f}^n(\vec{s}_L) + \vec{f}^n(\vec{s}_R) - |A|(\vec{s}_L - \vec{s}_R)\right) \tag{3.33}$$

Equation (3.33) is the approximate Riemann flux. The choice of approximation for $|A|$ yields different linear flux schemes. In the shallow water software, we use the Lax-Friedrich's flux where $|A|$ is approximated by the maximum eigenvalue using either the left or the right state,

$$|A| = max(\lambda_i(\vec{s}_L), \lambda_i(\vec{s}_R)). \tag{3.34}$$

Algorithm sketch

In any number of dimensions, the basic DGSEM can be broken into four main steps

1. For each element, interpolate the solutions to the element boundaries.
2. For each boundary edge/face, update the external states to apply boundary conditions.
3. For each edge/face, calculate the Riemann flux.
4. For each element, apply the DG-derivative matrix and add the weighted Riemann fluxes and source terms to produce a tendency.

In all of the DG modules in the SELF (e.g. `src/highend/euler/Euler_Class.f90`), there is a routine called `GlobalTimeDerivative` that performs these four steps. It takes a DG-data structure as input and uses the solution to update the tendency according to the above recipe. For each step, the operations can be done in parallel over the elements and edges/faces, allowing exploitation of SIMD parallelism through OpenMP and the OpenACC. Additionally, this parallelism is trivially exposed using domain decomposition and MPI. In this implementation, pairwise sends and receives are conducted during the external state update,

CGSEM

The Continuous Galerkin Spectral Element Method (CGSEM) can be derived similarly to the DGSEM. Instead of having Riemann fluxes that couple neighboring elements, continuity is enforced across corner-nodes, edges, and faces.

Time Integrators

Low-Storage 3rd Order Runge-Kutta

Discontinuous Galerkin in Time

After performing a spatial discretization to a system of partial differential equations, we are left with a large system of ordinary differential equations, e.g.,

$$\frac{d\vec{s}}{dt} = \vec{A}(\vec{s}, t). \tag{4.1}$$

In general, $\vec{A}$ is a nonlinear function of the solution variable $\vec{s}$. To solve (4.1) numerically, the time dimension is split into intervals $[t_i, t_{i+1}]$. Over each interval the prognostic variable is approximated as a linear superposition of basis functions (similar to the spatial discretization),

$$\vec{s}(t) \approx \sum_{i=0}^{N} \vec{s}_i \phi_i(t). \tag{4.2}$$

The solution is obtained by solving

$$\sum_{k=0}^{N_t} \left[\int_{t_k}^{t_{k+1}} \frac{d\vec{s}}{dt} \phi_j^k(t)\, dt \right] = \sum_{k=0}^{N_t} \left[\int_{t_k}^{t_{k+1}} \vec{A} \phi_j^k(t)\, dt \right], \text{ for j=0,1,2,...,N.} \tag{4.3}$$

The basis functions are allowed to be discontinuous across each interval, so that the integration over each interval can be performed independently. Thus, for each k,

$$\int_{-1}^{1} \frac{d\vec{s}}{d\xi} \phi_j(\xi)\, d\xi = \frac{\Delta t_k}{2} \int_{-1}^{1} \vec{A} \phi_j(\xi)\, d\xi, \text{ for j=0,1,2,...,N.} \tag{4.4}$$

Integration by parts on the first term gives

$$\sum_{i=0}^{N} \vec{s}_i \phi_i(1) \phi_j(1) - \int_{-1}^{1} \phi_j'(\xi) \vec{s}\, d\xi - \frac{1}{\Delta t_k} \int_{-1}^{1} \vec{A} \phi_j(\xi)\, d\xi = \vec{s}^{\star k} \phi_j(-1), \text{ for j=0,1,2,...,N,} \tag{4.5}$$

where the initial condition $\vec{s}(t = t_k) = \vec{s^k}$ has been applied. In the SELF, the basis functions are the Lagrange interpolating polynomials and integration over ξ is approximated using Legendre-Gauss quadrature. This yields the discrete system for integration

$$\sum_{i=0}^{N} \left(\frac{l_i(1)l_j(1) - \phi_j'(\xi_i)\omega_i}{\omega_j} \right) \vec{s}_i - \frac{\Delta t_k}{2} \vec{A}(\vec{s}_j, t_j) = \frac{\vec{s^k}\phi_j(-1)}{\omega_j}, \text{ for j=0,1,2,...,N.} \tag{4.6}$$

Once the $\vec{s}_i$ are known, the solution at $t = t_{k+1}$ can be estimated using the interpolation formula

$$\vec{s}(t_{k+1}) = \sum_{i=0}^{N} \vec{s}_i l_i(1). \tag{4.7}$$

The region of stability for the integration scheme (4.6) can be found by considering the single dimensioned ODE

$$s_t = \lambda s, \tag{4.8}$$

for which the discrete integration scheme is

$$\sum_{i=0}^{N} \left(\frac{l_i(1)l_j(1) - \phi_j'(\xi_i)\omega_i}{\omega_j} \right) s_i - \frac{\lambda \Delta t}{2} s_j = \frac{s^k \phi_j(-1)}{\omega_j}, \text{ for j=0,1,2,...,N.} \tag{4.9}$$

which can be compactly written as

$$\boldsymbol{G}(\lambda^*)\vec{s} = \vec{\Phi} s^k \tag{4.10}$$

where $\lambda^* = \frac{\lambda \Delta t}{2}$ and

$$\boldsymbol{G}_{i,j}(\lambda^*) = \left(\frac{l_i(1)l_j(1) - \phi_j'(\xi_i)\omega_i}{\omega_j} \right) - \delta_{i,j}\lambda^* \tag{4.11}$$

The solution at t_k is estimated using (4.7),

$$s(t_{k+1}) = \sum_{i,j=0}^{N} l_i(1) \left(\boldsymbol{G}^{-1}(\lambda^*) \right)_{i,j} \frac{\phi_j(-1)}{\omega_j} s^k \tag{4.12}$$

The region of stability is defined as the region where the *amplification factor* is less than one,

$$\left| \frac{s^{k+1}}{s^k} \right| \leq 1. \tag{4.13}$$

Fig. 4.1 shows the amplification factor (for N=1,2,3,4) as a function of the real and complex components of the eigenvalue λ^*. The red line marks the locations where the amplification factor is equal to one. This integration scheme is stable for high complex frequencies and negative real frequencies, indicating that a large time step can be taken while still remaining stable. The fact that the amplification factor is less than one for high frequencies and large time steps suggests that sufficient dissipation is provided for high frequency modes. Additionally, this method is easily

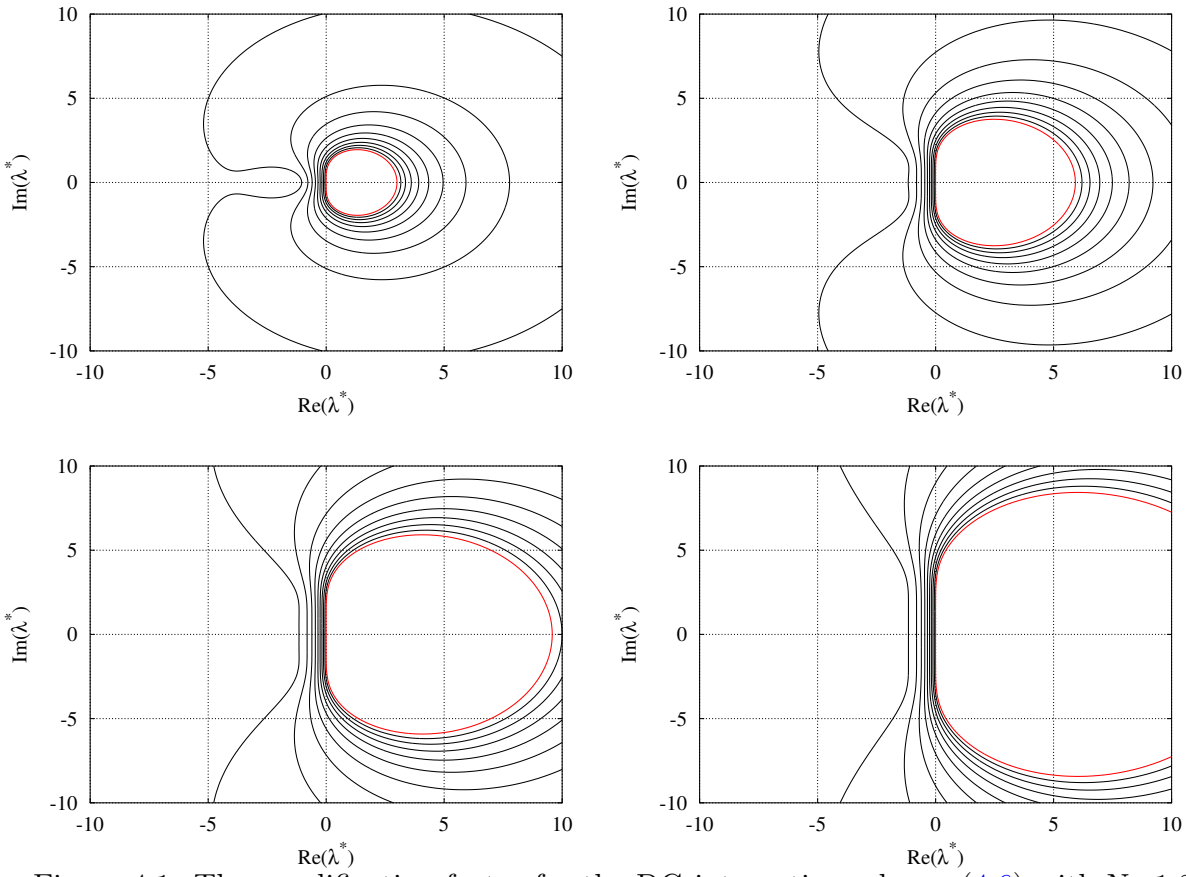


Figure 4.1: The amplification factor for the DG integration scheme (4.6) with N=1,2,3,4 as a function of the real and complex components of the eigenvalue λ^*. The region of stability is *outside* of the red contour where the amplification factor is less than one.

extensible to high order integration given the formulation presented in (4.6), though it should be noted that the memory requirements grow linearly with the polynomial degree.

To solve the general nonlinear problem (4.6), we start with an initial guess for each $\vec{s}_{i,(0)} = \vec{s}^{\star}$. Let $\vec{s}_{i,(m+1)} = \vec{s}_{i,(m)} + \Delta\vec{s}$. Linearizing (4.6) about $\vec{s}_{i,(m)}$ gives

$$\sum_{i=0}^{N} \left(\frac{l_i(1)l_j(1) - \phi_j'(\xi_i)\omega_i}{\omega_j} \right) \Delta\vec{s}_i - \frac{\Delta t_k}{2} \frac{\partial \vec{A}}{\partial \vec{s}}|_{(\vec{s}_{j,(m)},t_j)} \Delta\vec{s}_j = \frac{\vec{s}^{\star}\phi_j(-1)}{\omega_j} - \boldsymbol{B}_{j,(m)} \tag{4.14}$$

where

$$\boldsymbol{B}_{j,(m)} = \left(\sum_{i=0}^{N} \left(\frac{l_i(1)l_j(1) - \phi_j'(\xi_i)\omega_i}{\omega_j} \right) \vec{s}_{i,(m)} - \frac{\Delta t_k}{2} \vec{A}(\vec{s}_{j,(m)}, t_j) \right) \tag{4.15}$$

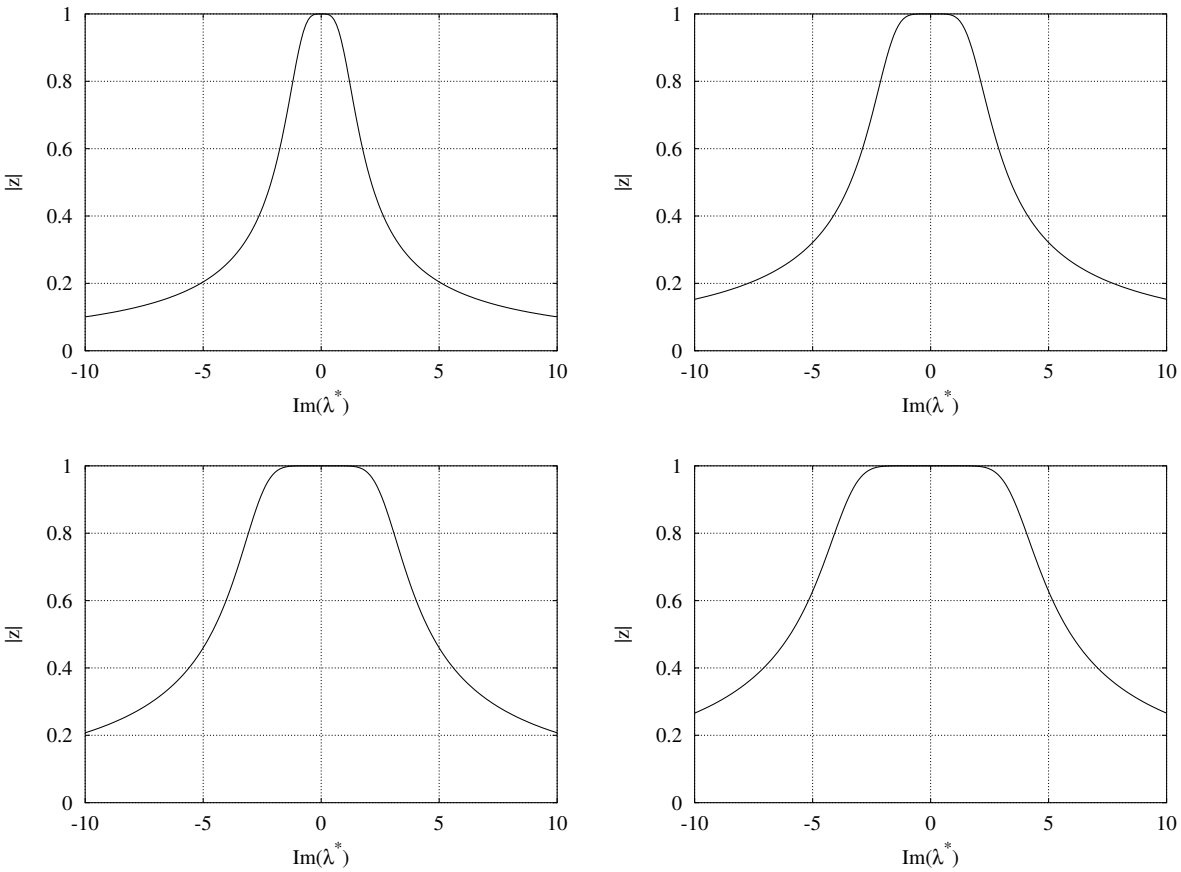


Figure 4.2: The amplification factor for the DG integration scheme (4.6) with N=1,2,3,4 along the complex-axis of Fig. 4.1.

DGIT Algorithm

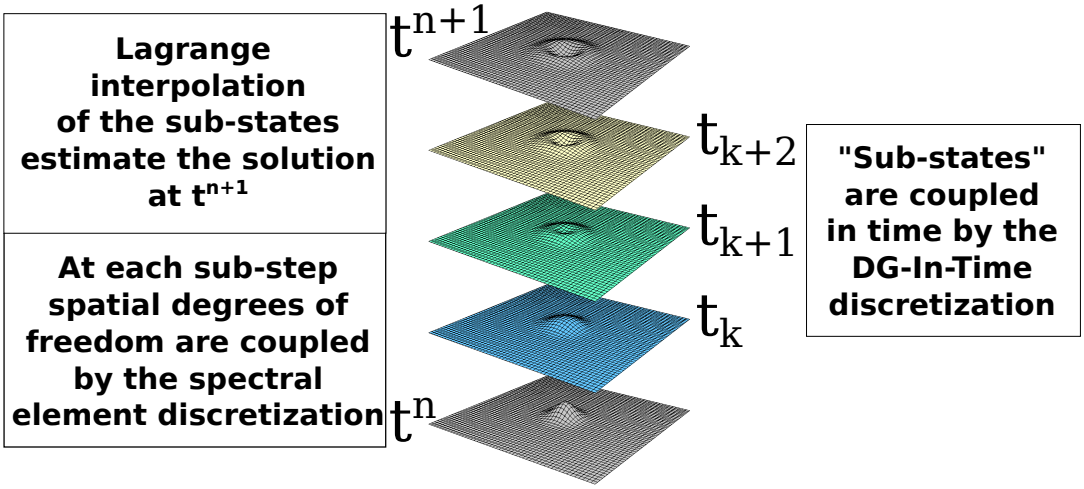


Figure 4.3: The amplification factor for the DG integration scheme (4.6) with N=1,2,3,4 as a function of the real and complex components of the eigenvalue λ^*. The region of stability is *outside* of the red contour where the amplification factor is less than one.

In solving the system (4.14), we need to compute the tendency for $N+1$ "sub-states", and each substate is coupled by the DG-in-time discretization. Once the substates are known the solution at the next time level can be estimated using Lagrange interpolation to t^{n+1}. This is depicted in the schematic shown in Fig. 4.3. To invert this system, it is clear that $N+1$ substates must be stored in memory to perform this integration. This can result in a substantial memory footprint for this algorithm for high N, and in practice it is recommended that $N \leq 2$. For linear problems and a single nonlinear iteration, we need to solve

$$\boldsymbol{G}\vec{s} = \vec{r} \tag{4.16}$$

where $\vec{s}$ is a correct that corresponds to all spatial degrees of freedom in addition to each substate. For a 3-D problem with spatial polynomial degree M, temporal degree N, with $nElem$ elements, and nEq PDE's in the PDE system, this vector has $D = (M+1)^3(nElem)(nEq)(N+1)$ degrees of freedom. In the SELF, (4.16) is solved with the preconditioned GMRES for each substate. For nonlinear problems, outer nonlinear iterations cycle over inner GMRES solves, until the change in the substate corrections is small enough.

Shallow Water

Compressible Euler (3-D)

Equations

The inviscid compressible Euler equations are

$$(\rho u)_t + (u\rho u + P)_x + (v\rho u)_y + (w\rho u)_z = 0 \tag{6.1a}$$

$$(\rho v)_t + (u\rho v)_x + (v\rho v + P)_y + (w\rho v)_z = 0 \tag{6.1b}$$

$$(\rho w)_t + (u\rho w)_x + (v\rho w)_y + (w\rho w + P)_z = -\rho g \tag{6.1c}$$

$$\rho_t + (u\rho)_x + (v\rho)_y + (w\rho)_z = 0 \tag{6.1d}$$

$$(\rho\theta)_t + (u\rho\theta)_x + (v\rho\theta)_y + (w\rho\theta)_z = 0 \tag{6.1e}$$

$$P = P(\rho, \theta) \tag{6.1f}$$

The prognostic variables for the compressible Euler equations are

$$\vec{s} = \begin{pmatrix} \rho u \\ \rho v \\ \rho w \\ \rho \\ \rho\theta \end{pmatrix} \tag{6.2}$$

where ρ is the fluid density, $\vec{u} = u\hat{x} + v\hat{y} + w\hat{z}$ is the fluid velocity, and θ is the potential temperature. The conservative flux is

$$\vec{f} = \begin{pmatrix} \rho\vec{u}u + P\hat{x} \\ \rho\vec{u}v + P\hat{y} \\ \rho\vec{u}w + P\hat{z} \\ \rho\vec{u} \\ \rho\vec{u}\theta \end{pmatrix} \tag{6.3}$$

Potential Temperature and Internal Energy

In this form, the potential temperature takes place of the internal energy equation.

Riemann Solver

Boundary Conditions

Topgraphic Waves

Stommel Model

Equations

The Stommel Equations are based on a particular scaling of the Boussinesq hydrostatic primitive equations. The momentum balance is steady state, nonlinear momentum advection is neglected, and the fluid is regarded as barotropic. These equations have historically been used to describe the large scale mean flow in the oceans and atmosphere. The lateral momentum balance is *geostrophic balance*,

$$f\hat{z} \times \vec{u} = -\nabla P \tag{8.1}$$

where $\vec{u}$ is the lateral fluid velocity components, P is the pressure per unit mass, and f is the coriolis parameter. Taking the (vertical component of the) curl of (8.1) gives the vorticity balance

$$\vec{u} \cdot \nabla f + f \nabla \cdot \vec{u} = 0 \tag{8.2}$$

For an incompressible fluid

$$\nabla \cdot \vec{u} = -w_z, \tag{8.3}$$

which gives (8.2) as

$$\vec{u} \cdot \nabla f - f w_z = 0. \tag{8.4}$$

Integrating over the interior of the fluid gives

$$h\vec{u} \cdot \nabla f - f(w_{surf} + \vec{u} \cdot \nabla h - C_d \zeta) = 0, \tag{8.5}$$

where $\vec{u}$ now represents the average fluid velocity. At the fluid surface ($z = 0$), the vertical velocity is due to the convergence of flow in a surface Ekman boundary layer. At the variable bottom ($z = -h(x, y)$), no-normal flow is applied, and an additional flow arises due to the convergence of flow in a frictional bottom Ekman boundary layer. Eq. (8.5) can be written

$$\vec{u} \cdot \nabla \left(\frac{f}{h} \right) + \frac{f C_d}{h} \zeta = \frac{f}{h} w_{surf} \tag{8.6}$$

Using (8.1), the velocity can be written in terms of a stream function Ψ, provided variations of f are *small enough*. With this, the vorticity balance can be approximated

$$\nabla \cdot \left(\frac{f}{h} \hat{z} \times \nabla \Psi + C_d^* \nabla \Psi \right) = \frac{f}{h} w_{surf} \tag{8.7}$$

where $C_d^* = \frac{f C_d}{h_0} + \mathcal{O}\left(\frac{\Delta h}{h_0}\right)$ is a constant.

Discretization

Eq. (8.7) is solved using the Continuous Galerkin Spectral Element Method (see Section 3.3). The gradient of the stream function in physical space is calculated by rotating the gradient in the computational space,

$$\frac{\partial \Psi}{\partial \xi^1}|_{i,j} = \sum_{m=0}^{N} \Psi_{m,j} l'_m(\xi_i^1) = \boldsymbol{D}_{i,m} \Psi_{m,j} \tag{8.8a}$$

$$\frac{\partial \Psi}{\partial \xi^2}|_{i,j} = \sum_{n=0}^{N} \Psi_{i,n} l'_n(\xi_j^2) = \Psi_{i,n} \boldsymbol{D}_{j,n} \tag{8.8b}$$

where a sum is implied over repeated indices. To calculate the gradient in physical space, the computational gradients are rotated along the contravariant

$$\begin{pmatrix} \frac{\partial \Psi}{\partial x} \\ \frac{\partial \Psi}{\partial y} \end{pmatrix} = \frac{1}{J} \begin{pmatrix} \frac{\partial y}{\partial \xi^2} & -\frac{\partial y}{\partial \xi^1} \\ -\frac{\partial x}{\partial \xi^2} & \frac{\partial x}{\partial \xi^1} \end{pmatrix} \begin{pmatrix} \frac{\partial \Psi}{\partial \xi^1} \\ \frac{\partial \Psi}{\partial \xi^2} \end{pmatrix} \tag{8.9}$$

where J is the Jacobian of the mapping from computational to physical space.

Under the discretization, (8.7) is approximated by the linear matrix system

$$\boldsymbol{A} \vec{\Psi} = \vec{q}. \tag{8.10}$$

The matrix $\boldsymbol{A}$ is asymmetric; this feature is expected due to the hyperbolic nature of Rossby waves that are supported by the unsteady form of (8.7).

SELF Organization

Modules

Common/Support

Linked-List

Hash-Table

Notched-Key and Key-Ring

Interpolation, Differentiation, and Integration

As indicated in Chapter **??**, functions are approximated by Lagrange interpolating polynomials in nodal spectral element methods. The interpolation nodes are equivalent to either the Legendre-Gauss (for DG methods) or Legendre-Gauss-Lobatto (for CG methods) quadrature nodes in the SELF. The interpolation modules, found under the `SELF/src/interp/` directory, provide :

1. Routines to calculate the quadrature nodes and (integration) weights
2. Data structures for Lagrange interpolating polynomials
3. Routines for interpolating data
4. Routines for approximating derivatives using Lagrange interpolating polynomials

Geometry

Solution Storage

Nodal SEM Storage

Template DGSEM

Spectral Filters

Models are just that, they're models. We cannot currently hope to explicitly and accurately represent all of the spatial and temporal scales for a problem. Instead, models focus in on the spatial and temporal scales that are important and an informed decision is made to deal with the interactions with smaller scales. In the prototypical problem of shock formation (a la Burger's equation), small scales are forced into existence from the large scale. This nonlinear transfer of energy to small scales, called the "forward cascade", poses challenges for numerical models that can become unstable when too much energy is forced into the smallest resolvable scale. The comic shown in Fig. 9.1 shows what happens when you don't remove that energy from the grid scale (the red line). The blue line shows the same shock formation occurring, except that the numerical algorithm now includes a "filter" that turns on when too much energy appears in the grid-scale and violates the principles of the forward cascade (it is called "adaptive" because of its conditional action). The filter effectively removes the oscillations, so-called "grid-scale noise", and removes enough energy to maintain stable numerical integration. The filter used in generating the stable solution in Fig. 9.1 is a "Roll-off" filter, meaning that it damps marginally resolved modes much stronger than the well resolved modes. This Adaptive Roll-off Filter (ARF) is a parameterization for shock formation. Motivated by the need for maintaining numerical stability in a justifiable manner, a set of filters are provided with the SELF as one option towards meeting this goal. In this section, the theory behind using filters is presented for 1-D. The advantage that polynomial filtering has over other parameterizations (e.g. Laplacian diffusion), is that the additional error incurred by such a filter is quantifiable.

In the "standard" nodal discretization of conservation laws presented in Eqns. (3.19) (3.21) and (3.22), additional aliasing errors can arise in approximating the flux by a polynomial of the same degree as the solution. This is always the case when the flux depends nonlinearly on the solution. The introduction of aliasing errors directly implies inexactness of the discrete integration and hence stability of the spatial discretization does not follow. However, the quadrature order

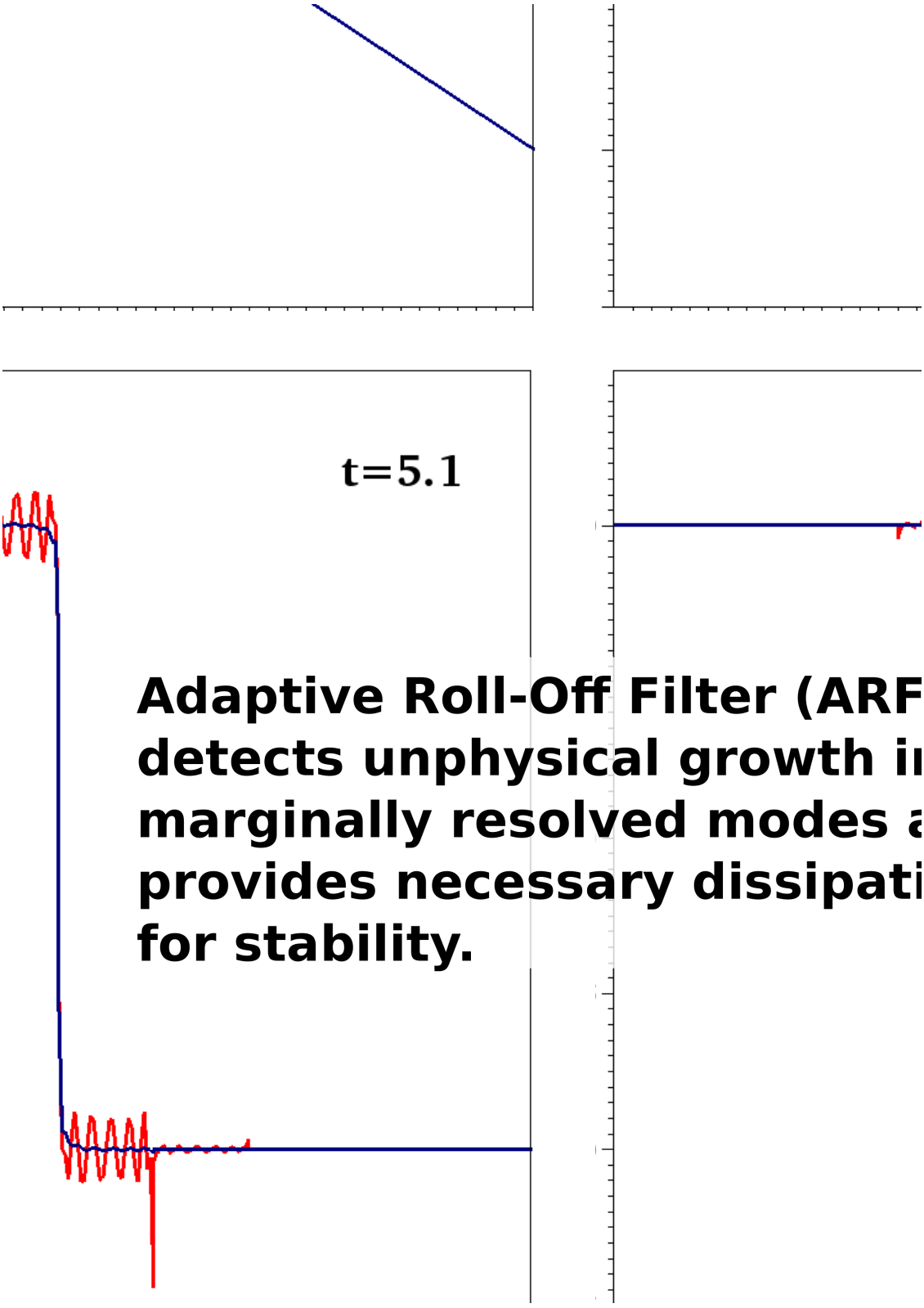

t=5.1
Adaptive Roll-Off Filter (ARF
detects unphysical growth i
marginally resolved modes a
provides necessary dissipati
for stability.

Figure 9.1:

can be increased in order to remove these aliasing errors which can then restore stability (provided the continuous problem is well posed). This approach is known as *over-integration*. In theory, over-integration would involve interpolation of the solution onto a higher order mesh, integration of the flux, and mapping back to the lower order mesh. In practice, this process can be done more efficiently by *polynomial de-aliasing* (**??**). In the de-aliasing framework, over-integration is achieved by applying a *modal-cutoff filter* to the solution variable that effectively projects the solution to a polynomial of low-enough degree so that the integration of the flux is exact.

As an example, suppose we are interested in solving Burger's equation in 1-D, as in Fig. 9.1,

$$u_t + \left(\frac{u^2}{2}\right)_x = 0 \tag{9.1a}$$

$$u(0,t) = 1 \tag{9.1b}$$

$$u(10,t) = -1 \tag{9.1c}$$

$$u(x,0) = -\frac{2x}{10} + 1 \tag{9.1d}$$

If u is approximated by a polynomial of degree N, then the integrand in the discretized system (3.19) for the flux integral is of degree $3N$. Gauss quadrature of order M is exact for integrands of degree $2M+1$, so

$$3N = 2M + 1 \tag{9.2}$$

guarantees exactness of the discrete integration and stability of the spatial discretization. In practice, the discretization of (9.1) would be performed at degree M, and at every iteration the solution would be projected onto a polynomial of degree N that satisfies the exactness criteria (9.2).

How to de-alias

Recall that the approximate solution can be expressed in *nodal* or *modal* form :

$$u \approx I_M[u] = \sum_{i=0}^{M} U_i l_i(\xi) = \sum_{j=0}^{M} \hat{U}_j L_j(\xi) \tag{9.3}$$

where the U_i are the nodal values of the interpolant, l_i are the Lagrange interpolating polynomials and $\hat{U}_j$ are the modal coefficients of the Legendre basis functions L_j. Other polynomial bases can be used, but here our attention is restricted to Legendre polynomials. Gauss quadrature of order M guarantees discrete orthogonality between the Legendre polynomials up to degree M. Because

of this, weighting (9.3) with each Legendre polynomial and performing discrete integration provides a formula for each of the modal coefficients,

$$\hat{U}_m = \sum_{k=0}^{M} U_k \frac{L_m(\xi_k)\sigma_k}{||L_m||_M^2}. \tag{9.4}$$

Eq. (9.4) can be conveniently be written in matrix form

$$\hat{U} = T\vec{U} \tag{9.5}$$

where $\hat{U}$ is a vector of the modal coefficients, $\vec{U}$ is a vector of the nodal values, and $T_{m,k} = \frac{L_m(\xi_k)\sigma_k}{||L_m||_M^2}$ is the nodal-to-modal transformation matrix. The inverse of T transforms from the modal space to the nodal space and is easily obtained by evaluating the Legendre polynomials at the interpolation nodes :

$$T_{m,k}^{-1} = L_k(\xi_m) \tag{9.6}$$

Filtering is done by transforming the nodal values to the modal coefficients, modifying the modal coefficients, and applying the inverse to map the "filtered" interpolant back to the nodal values.

1. Obtain the modal coefficients : $\hat{U} = T\vec{U}$
2. Apply a filter on the modal coefficents : $\tilde{U} = P\hat{U} = PT\vec{U}$
3. Map from the filtered modal coefficients to the nodal coefficients : $\vec{U}_f = (T^{-1}PT)\vec{U}$

When de-aliasing, the modal coefficients associated with Legendre polynomials higher than degree N are set to zero identically, so that the interpolant is effectively projected onto a lower order polynomial. The modal filter matrix that does this is a diagonal matrix with

$$P_{i,i} = \begin{cases} 1, & \text{if } i \leq N \\ 0, & \text{otherwise} \end{cases} \tag{9.7}$$

The matrix $T^{-1}PT$ with P given by (9.7) is called a *modal-cutoff filter*.

Support Programs

This section provides a description of the SELF-provided support programs. For "highend" programs (e.g. Shallow-Water programs), see Section 9.3

Geometry

GenerateMeshFiles.f90

DecomposeQuadMesh.f90

GenerateMeshFiles_3D.f90

DecomposeHexMesh.f90

High End Programs

Shallow Water Solver (2-D DGSEM)

Compressible Euler Solver (3-D DGSEM)

Software Verification and Timing

Lagrange Interpolation

Guassian Bump in 1-D

This example tests the functionality of the following modules :

1. Legendre.f90
2. Lagrange_1D_Class.f90

By interpolating through the Legendre-Gauss nodes, the error in the interpolating function should decay spectrally with increasing interpolating polynomial degree (**?**). The interpolant can be expressed as the sum of the actual function and an interpolation error,

$$I_N[f(s)] = f(s) + \epsilon(N), \tag{10.1}$$

where $I_N[f(s)]$ denotes the interpolation of $f(s)$ by a degree N. When the evaluation nodes are the Legendre Gauss nodes, $\epsilon(N)$ is a spectrally small error; it's rate of decay with N depends on the smoothness of f. Additionally, the derivative of the interpolant approximates the derivative of the function.

$$\frac{d}{ds}[I_N(f)] = \frac{df}{ds} + \delta(N) \tag{10.2}$$

where $\delta(N)$ is also a spectrally small error. For the interpolation of smooth functions, the error decays exponentially with N.s

A test program (∼`/src/interp/Testing/TestLagrangeInterpolation_1D.f90`) has been written to interpolate the function

$$f(s) = e^{-s^2} \tag{10.3}$$

and estimate the $\mathbb{L}_2$ error of the interpolation and differentiation. Differentiation is carried out by building the derivative matrix (`S/R CalculateDerivativeMatrix`) and performing a matrix vector multiply (`MATMUL`). The $\mathbb{L}_2$ error is estimated using Legendre-Gauss quadrature with 50 evaluation nodes, ie,

$$||f - I_N[f]||_{\mathbb{L}_2} = \sum_{k=0}^{K} \left(f(s_k) - I_N[f(s_k)]\right)^2 \sigma_k \tag{10.4}$$

where s_k and σ_k are the degree K Legendre Gauss quadrature nodes quadrature weights respectively.

Figures **??** and **??** show the interpolants and the interpolant derivatives respectively for polynomial degrees $N = 2$ through $N = 7$. Qualtitatively, these plots illustrate the convergence of the interpolants to the exact function and derivative. Figure **??** shows the $\mathbb{L}_2$ error norm in approximating the function and the derivative for polynomial degrees 2 through 17. Both errors exhibit exponential convergence, in agreement with the theoretical error bounds. Notice that the error in approximating the function is always smaller than in approximating its derivatine (for each polynomial degree). This indicates that approximating the derivative by taking the derivative of the interpolant incurs an additional error; the derivative of the interpolant is not always equal to the interpolant of the derivative.

Given that the function being interpolated is complete (we can take as many derivatives as we like), the theoretical error decay is exponential. This is in agreement with the error trend observed numerically, providing evidence that the interpolation and differentiation routines at the Legendre Gauss nodes are working.

History of the SELF

The SELF began as a series of homework assignments from a graduate course in spectral element methods taught by David Kopriva (Spring 2013) at the Florida State University. Many of the subroutines are adapted from the pseudo-code presented in Kopriva's book, "Implementing Spectral Element Methods for Scientists and Engineers". The SELF became a modularized Fortran embodiment of Kopriva's ideas. Originally, this software was coined "Software for the Computation of Hydrodynamics Objects and Operations, Achieving Numerical Efficieny and Reliability (SCHOONER)" by fellow graduate student Carlowen Smith at the Geophysical Fluid Dynamics Institute. This acronym pointed towards the original goal of the software, namely, to produce a set of tools that one could use to solve problems related to fluid flow.

Towards the end of graduate school, it became apparent to me that the scientific community would benefit from a single "toolbox" that can be used, with a trivial amount of effort, to implement spectral element methods. I believe that such a toolbox has the potential to enhance reproducibility of research that requires numerical methods and can reduce the development time when solving complex PDE systems. The name "Spectral Element Libraries in Fortran (SELF)" represents these beliefs.

The software has undergone much organization and optimization since the original development. During the summer of 2016, four students from the Parallel Computing Summer Research Internship (PCSRI) at Los Alamos National Laboratory refined the (serial) shallow-water solver modules, and helped redesign the SELF for a Version 3 Release. They additionally explored parallelization schemes through threading with OpenMP, OpenACC, and domain decomposition with message-passing. Their contributions have greatly improved the efficiency of the software and have spurred a redesign of many other portions of the code. I am greatly indebted to Bob Robey, Hai

Ah Nam, and Gabe Rockefeller at Los Alamos for including me in the PCSRI program which led to significant improvements in the code.

Currently, there are officially three developers for the SELF, and a small set of users. It is my hope that this will expand over time. I am glad that you are taking part in the history of the SELF and I look forward to hearing about your research and the role that this software has for your work.

- Dr. J. Schoonover

SEM Fundamentals

Interpolation

To illustrate how the interpolation works, let's start with interpolation in 1-D. A function f is approximated by a Lagrange interpolant of degree N,

$$F = \sum_{i=0}^{N} f_i l_i(\xi), \tag{A.1}$$

where the f_i are nodal values of f at the interpolation nodes $\xi_i{}_{i=0}^{N}$. The $l_i(\xi)$ are the Lagrange interpolating polynomials that have the property

$$l_i(\xi_j) = \delta_{i,j}, \tag{A.2}$$

where $\delta_{i,j}$ is the Kronecker-delta function. The property (A.2) indicates that the interpolating polynomial $l_i(\xi) = 1$ at ξ_i and has roots at all of the other interpolation nodes. Show a figure illustrating the Lagrange interpolating polynomials. These functions are written mathematically as

$$l_i(\xi) = \prod_{j \neq i} \frac{(\xi - \xi_j)}{(\xi_i - \xi_j)} \tag{A.3}$$

Following **?**, the interpolating polynomials are written in the "Barycentric Formulation", which has favorable round-off error properties.

$$l_i(\xi) = \frac{\frac{w_i}{\xi - \xi_i}}{\sum_{j=0}^{N} \frac{w_j}{\xi - \xi_j}}, \tag{A.4}$$

where w_j are the "barycentric-weights",

$$w_j = \left(\prod_{i \neq j} (\xi_j - \xi_i) \right)^{-1} \tag{A.5}$$

Interpolation in multiple dimensions is achieved by using tensor products of Lagrange-interpolating polynomials. In 2-D

$$F = \sum_{j=0}^{N} \sum_{i=0}^{N} f_{i,j} l_i(\xi^1) l_j(\xi^2), \tag{A.6}$$

and in 3-D

$$F = \sum_{k=0}^{N} \sum_{j=0}^{N} \sum_{i=0}^{N} f_{i,j,k} l_i(\xi^1) l_j(\xi^2) l_k \xi^3. \tag{A.7}$$

The superscripts in ξ^1, ξ^2, and ξ^3 correspond to each computational dimension. In general, given a set of nodes in each computational direction, the barycentric weights are computed using eq. (A.5) and the interpolating polynomials can be evaluated using (A.4).

It is often the case in application that we want to interpolate functions from a "computational grid" to a "plotting grid". In 1-D, we want to take $\{F(\xi_i)\}_{i=0}^{N}$ and map it to $\{F(\xi_m)\}_{m=0}^{M}$. From the interpolant definition (A.1),

$$F(\xi_m) = \sum_{i=0}^{N} f_i l_i(\xi_m) = \sum_{i=0}^{N} T_{m,i} f_i. \tag{A.8}$$

The "interpolation" matrix $T_{m,j} = l_i(\xi_m)$ maps an interpolant from one set of nodes $\{\xi_i\}_{i=0}^{N}$ to another $\{\xi_m\}_{m=0}^{M}$. In 2-D,

$$F(\xi_m^1, \xi_n^2) = \sum_{j=0}^{N} \sum_{i=0}^{N} T_{n,j} T_{m,i} f_{i,j}, \tag{A.9}$$

and 3-D

$$F(\xi_m^1, \xi_n^2, \xi_p^3) = \sum_{k=0}^{N} \sum_{j=0}^{N} \sum_{i=0}^{N} T_{p,k} T_{n,j} T_{m,i} f_{i,j,k}, \tag{A.10}$$

Integration