Solving PDEs with Intrepid

P. Bochev[1,5], H. C. Edwards[3,5], R. C. Kirby[4], K. Peterson[1,5], D. Ridzal[2,5]

December 7, 2011

1 Introduction

Intrepid is a Trilinos package for advanced discretizations of Partial Differential Equations (PDEs). The package provides a comprehensive set of tools for local, cell-based construction of a wide range of numerical methods for PDEs. This paper describes the mathematical ideas and software design principles incorporated in the package. We also provide representative examples showcasing the use of Intrepid both in the context of numerical PDEs and the more general context of data analysis.

The mathematical roots of Intrepid are in the abstract framework for compatible discretizations [5]. This framework prompted a reevaluation of traditional software designs for PDE discretizations, which usually target a single discretization paradigm such as Finite Element Methods (FEM), Finite Volume Methods (FVM) or Finite Difference Methods (FDM). Intrepid aims to translate these mathematical similarities into software-based similarities.

Finding software abstractions that unify these disparate families of methods differentiates Intrepid from existing PDE software. Software packages such as Chombo (seesar.lbl.gov/anag/chombo), Overture (computation.llnl.gov/casc/Overture) and ClawPack (www.clawpack.org) target FDM and FVM on adaptive block structured meshes, whereas deal.ii (www.dealii.org), HERMES (hpfem.org/hermes), Sundance (www.math.ttu.edu/~kelong/Sundance/html/), and FEniCS (www.fenicsproject.org) support primarily FEM, based on weak variational forms of the PDEs.

Additionally, such packages attempt to integrate the entire user experience, providing tools for meshes, global assembly, and interfaces to linear algebra. Some packages, such as FEniCS and Sundance, even provide an embedded description language of weak forms and automate the construction of stiffness matrices. Intrepid expresses a much lower level of abstraction, targeting the elementwise aspects of discretization via stateless methods on flat data structures. Existing large-scale application codes, such as the multiphysics simulation environments Drekar [30] and Albany [25], currently use Intrepid, but one could also envision Intrepid as a back-end for FEniCS or Sundance. As such, Intrepid represents a kind of middleware between higher-level software architecture and lower-level numerics that provides basis functions, transformations, and integration to diverse client codes.

We organize the paper as follows. Section 2 briefly reviews the key mathematical concepts incorporated in Intrepid. Section 3 explains the basic design principles adopted by Intrepid, and

[1]Numerical Analysis and Applications Department, Sandia National Laboratories, Mail Stop 1320, Albuquerque, New Mexico, 87185-1320.

[2]Optimization and Uncertainty Quantification Department, Sandia National Laboratories, Mail Stop 1320, Albuquerque, New Mexico, 87185-1320.

[3]Multiphysics Simulation Technologies Department, Sandia National Laboratories, Mail Stop 1318, Albuquerque, New Mexico, 87185-1320.

[4]Department of Mathematics and Statistics, Texas Tech University. The author acknowledges support from a contract from Sandia National Laboratories and NSF award 1117794.

[5]Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Section 4 explains the structure of the package. Section 5 contains examples of computational patterns arising in the discretization of differential operators. The paper concludes with a brief summary of additional capabilities and future directions in Section 6.

2 Mathematical Foundations of Intrepid

Stable and accurate numerical solution of PDEs is required for predictive computational simulations in science and engineering. A critical step in this task is the discretization, which translates PDEs into algebraic models that can be solved on computers. Inevitably, discretization leads to information losses about the mathematical structure of the original problem.

Compatible discretizations transform PDEs into algebraic equations that mimic their fundamental structural properties. This is especially relevant for multiphysics problems where unphysical modes from one component may cause instability in the other components. To achieve compatibility, FEM, FVM and FDM follow different paths, reflecting their distinct mathematical roots.

FEM are variational methods that approximate spaces rather than (weak) PDE equations. Inf-sup conditions ensure the existence of bounded solution operators [3, 8], and govern compatibility of FEM. FVM replace PDEs by integral equations expressing equilibriums of global quantities on arbitrary subdomains of the problem domain. Restriction of these equilibrium relations to a finite number of subdomains (the finite volumes) yields the discretization. Application of the Stokes theorem to derive the equilibrium relations is key to the compatibility of FVM [24, 33]. Finally, FDM approximate directly the differential operators in the PDE. Physically motivated locations of variables on the grid ensure the compatibility of these methods [28, 38, 16].

However, in spite of their differences, compatible FEM, FVM and FDM tend to produce discrete problems with strikingly similar properties. This observation led to the discovery of a common trait that underwrites compatible FEM, FVM and FDM, namely the notion of a "discrete vector calculus" structure, i.e., mutually consistent operations for integration and differentiation, which obey vector calculus identities and theorems [7, 5, 28, 2].

The existence of such a common trait across different discretizations motivates the development of the Intrepid package. Intrepid aims to provide a common API for mesh-based discretization methods, which enables virtually unlimited extensibility of the package to new FEM, FVM and FDM technologies. To this end, the abstract framework in [5] guides Intrepid's API design. This section reviews the key junctures of the framework and how they influence the Intrepid package.

2.1 Exterior Calculus

We assume that the reader is familiar with the basic notions of exterior differential calculus [14]. For brevity, we restrict attention to bounded, simply connected regions Ω in three-dimensions. The symbol $\Lambda^k(\Omega)$, $k = 0, 1, 2, 3$, stands for smooth differential k-forms, or simply forms, $x \mapsto \omega(x) \in \Lambda(T_x\Omega)$, where $T_x\Omega$ is the tangent manifold to Ω at x. We recall the wedge product $\wedge : \Lambda^k(\Omega) \times \Lambda^l(\Omega) \mapsto \Lambda^{k+l}(\Omega)$ and the exterior derivative $d : \Lambda^k(\Omega) \mapsto \Lambda^{k+1}(\Omega)$. The relation $dd = 0$ gives rise to the exact sequence

$$\mathbf{R} \longrightarrow \Lambda^0 \xrightarrow{d} \Lambda^1 \xrightarrow{d} \Lambda^2 \xrightarrow{d} \Lambda^3 \longrightarrow 0\,, \tag{1}$$

called DeRham complex. Assuming that Ω is a Riemannian manifold, the metric structure gives rise to an inner product $(\cdot,\cdot)$ on $\Lambda^k(\Omega)$ and an adjoint $d^* : \Lambda^{k+1} \mapsto \Lambda^k$, viz.:

$$(d\omega, \eta) = (\omega, d^*\eta)\,. \tag{2}$$

The completion of $\Lambda^k(\Omega)$ with respect to the inner product is the Hilbert space of square integrable differential forms $\Lambda^k(L^2, \Omega)$. We also have the Sobolev spaces $\Lambda^k(d, \Omega) = \{\omega \in \Lambda^k(L^2, \Omega) \,|\, d\omega \in \Lambda^{k+1}(L^2, \Omega)$.

The DeRham complex (1) and its dual relative to d^* are exterior calculus notions that can encode the structure of a large class of PDEs composed from d, d^*, and their higher-order products such as dd^* and d^*d. Consequently, building finite dimensional "exterior calculus" structures for FEM, FVM and FDM is now the standard approach to design and analyze compatible discretization methods for PDEs [2, 5].

2.2 Discrete Exterior Calculus

The algebraic topology framework for compatible discretizations [5] provides discrete "exterior calculus", which includes the classes of FEM [7, 36], FVM [23] and FDM [28] as particular cases.

Interpretation of the grid and the data as chain and cochain complexes, respectively, is essential for the construction of the discrete exterior calculus structures in the framework. Specifically, we consider computational grids Ω^h consisting of 0-cells (nodes), 1-cells (edges), 2-cells (faces), and 3-cells (volumes). Formal linear combinations of k-cells are called k-chains [12]. The sets of k-chains forming Ω^h are denoted by C_k. We will assume[1] that Ω^h is such that the collection $\{C_0, C_1, C_2, C_3\}$ is a chain complex, i.e., for any $c \in C_k$, $\partial_k c \in C_{k-1}$, where $\partial_k : C_k \mapsto C_{k-1}$ is the boundary operator on k-chains [10]. Together with the identity $\partial_k \partial_{k+1} = 0$ this gives rise to the exact sequence

$$0 \longleftarrow C_0 \xleftarrow{\partial_1} C_1 \xleftarrow{\partial_2} C_2 \xleftarrow{\partial_3} C_3 \longleftarrow 0\,. \tag{3}$$

The dual of C_k, i.e., the space of all bounded linear functionals $C_k \mapsto \mathbf{R}$, is denoted by C^k. The elements of C^k are called co-chains and the duality pairing between chains and cochains is $\langle \cdot, \cdot \rangle$. The identity $\langle \partial c_{k+1}, c^k \rangle = \langle c_{k+1}, \delta c^k \rangle$ defines an adjoint $\delta : C^k \mapsto C^{k+1}$ of ∂, called coboundary. It satisfies $\delta_{k+1}\delta_k = 0$ and gives rise to the exact sequence

$$\mathbb{R} \longrightarrow C^0 \xrightarrow{\delta_0} C^1 \xrightarrow{\delta_1} C^2 \xrightarrow{\delta_2} C^3 \longrightarrow 0\,. \tag{4}$$

The chain and cochain complexes represent the mesh and the data structures, respectively, in our framework. Two basic operators define the rest of the structures necessary for a self-consistent discrete exterior calculus. The *reduction* operator $\mathcal{R} : \Lambda^k(d, \Omega) \mapsto C^k$ translates forms to cochains and is given by the DeRham map

$$\langle \mathcal{R}\omega, c_k \rangle = \int_{c_k} \omega\,.$$

The DeRham map establishes discrete representation of k-forms in terms of global quantities associated with the chain complex (the computational grid). It has the important Commuting Diagram Property (CDP) $\mathcal{R}d = \delta\mathcal{R}$. The *reconstruction* operator $\mathcal{I} : C^k \mapsto \Lambda^k(L^2, \Omega)$ translates cochains back to forms. The range of this operator is a finite dimensional subspace of $\Lambda^k(L^2, \Omega)$. When the range of $\mathcal{I}$ is in the Sobolev space $\Lambda^k(d, \Omega)$ we call $\mathcal{I}$ *conforming*. We denote $\Lambda_h^k(\mathcal{I}, \Omega) = \text{range}(\mathcal{I})$ and call it *discrete space* associated with $\mathcal{I}$. Let $\dim C^k = N_k$ and $\{\mathbf{e}_i\}$ be the canonical basis of $\mathbb{R}^{N_k}$. Because $C^k \cong \mathbb{R}^{N_k}$ and $\Lambda_h^k(\mathcal{I}, \Omega) \cong C^k$, it follows that $\{\mathbf{e}_i\}$ is a basis of C^k and $\{\mathcal{I}\mathbf{e}_i\}$ is a basis of $\Lambda_h^k(\mathcal{I}, \Omega)$. For obvious reasons we refer to the latter as the canonical basis of the discrete space.

There are many possible ways to define $\mathcal{I}$, however, to obtain an accurate and self-consistent discrete exterior calculus, $\mathcal{I}$ must satisfy

$$\mathcal{R}\mathcal{I} = I \quad \text{and} \quad \mathcal{I}\mathcal{R} = I + O(h^r)\,, \tag{5}$$

where h is average cell size in the chain complex and r is a positive integer. In other words a "good" reconstruction operator $\mathcal{I}$ is a right inverse and approximate left inverse of the reduction operator.

A reconstruction operator $\mathcal{I}$ also gives rise to a notion of a functional basis for any discretization. Let $\dim C^k = N_k$ and $\{\mathbf{e}_i\}$ denote the canonical basis of $\mathbb{R}^{N_k}$. Because $C^k \cong \mathbb{R}^{N_k}$, $\{\mathbf{e}_i\}$ is also a basis of C^k. The set of fields $\{\mathcal{I}\mathbf{e}_i\}$

[1] This assumption can be easily satisfied for most grids, including grids with hanging nodes, by representing k-cells that have hanging nodes as unions of k-cells.

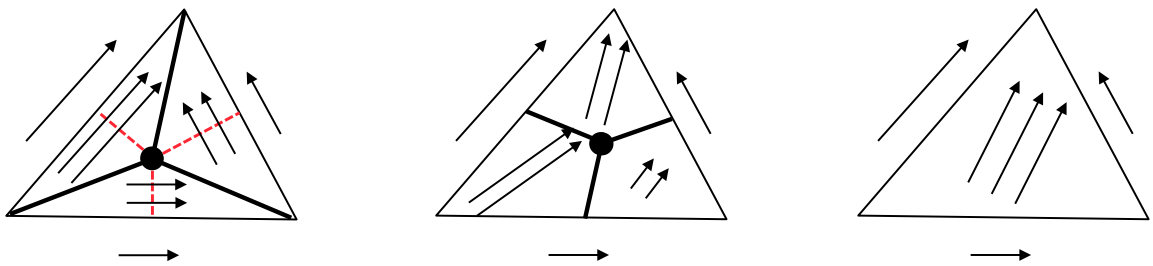


Figure 1: Examples of three different reconstruction operators for 1-cochains, corresponding to compatible FVM, FDM and FEM. The covolume $\mathcal{I}$ (leftmost plot) divides the simplex into three subsimplices by connecting the circumcenter of with its vertices. Each subsimplex is bordered by exactly one of the edges. The covolume reconstruction operator maps the 1-cochain into a 1-form whose associated vector field is piecewise constant on each subsimplex. Mimetic reconstruction (center plot) acts in a similar way to recover a form with a piecewise constant vector field. The subregions are associated with the vertices, have quadrilateral shapes, and are bordered by the edges adjacent to each vertex. The finite element reconstruction (rightmost plot) is an example of a conforming mimetic reconstruction operator. It maps the 1-cochain to a polynomial 1-form using a basis of polynomial 1-forms $\lambda_i \nabla \lambda_j$ associated with the edges e_{ij} of the simplex. Here λ_i are the barycentric coordinates of the simplex.

Discrete operations. The exactness of (4) implies that δ is a compatible approximation of the exterior derivative d, i.e., it provides a discrete gradient, curl and divergence, which satisfy the standard vector calculus identities $\nabla \times \nabla = 0$ and $\nabla \cdot \nabla \times = 0$. The discretization of d^* requires an inner product on cochains. We define this inner product using the reconstruction operator and the inner product on Λ^k:

$$(a, b)_{C^k} := (\mathcal{I}a, \mathcal{I}b)\,.$$

The identity $(\delta^* a, b)_{C^k} = (a, \delta b)_{C^{k+1}}$ defines the adjoint $\delta^* : C^{k+1} \mapsto C^k$. This operator satisfies $\delta^*\delta^* = 0$ and so it provides a second set of compatible discretizations for grad, curl and div. Using δ and δ^* one can obtain compatible discretizations of further differential operators such as the discrete Hodge Laplacian $\Delta_k : C^k \mapsto C^k$, $\Delta_k = \delta\delta^* + \delta^*\delta$. For additional operations such as discrete integral and exterior product we refer to [5].

Mimetic properties. To sum it up, our abstract framework comprises of a computational domain, represented by a chain complex $\{C_0, C_1, C_2, C_3\}$, discrete fields, represented by a cochain complex $\{C^0, C^1, C^2, C^3\}$, reduction and reconstruction operators, and notions of a discrete inner product on cochains, integral, derivative, adjoint derivative, and wedge product, to name a few. The framework represents a self-consistent discrete exterior calculus structure. Some of its key *mimetic* properties are as follows:

1. Exactness: $\delta\delta = \delta^*\delta^* = 0$;
2. A discrete Stokes theorem: $\langle \delta c^{k-1}, c_k \rangle = \langle c^{k-1}, \partial c_k \rangle$;
3. A discrete Hodge decomposition: $a = \delta b + \delta^* c + h$ for $a \in C^k$, where $\delta h = \delta^* h = 0$.

For proofs and further properties we refer to [5].

Application of the framework to API design. One of the key conclusions in [5] is that for a large class of compatible discretizations the distinctions between FEM, FVM or FDM result from specific implementations of $\mathcal{I}$, i.e., *the choice of the reconstruction operator defines the method.* Figure 1 shows examples of reconstruction operators corresponding to compatible FVM, FDM and FEM, which yield numerical methods with very similar properties.

Because in our framework the type of the compatible discretization is selected through a choice of the reconstruction operator, translation of PDEs into structure-preserving algebraic problems

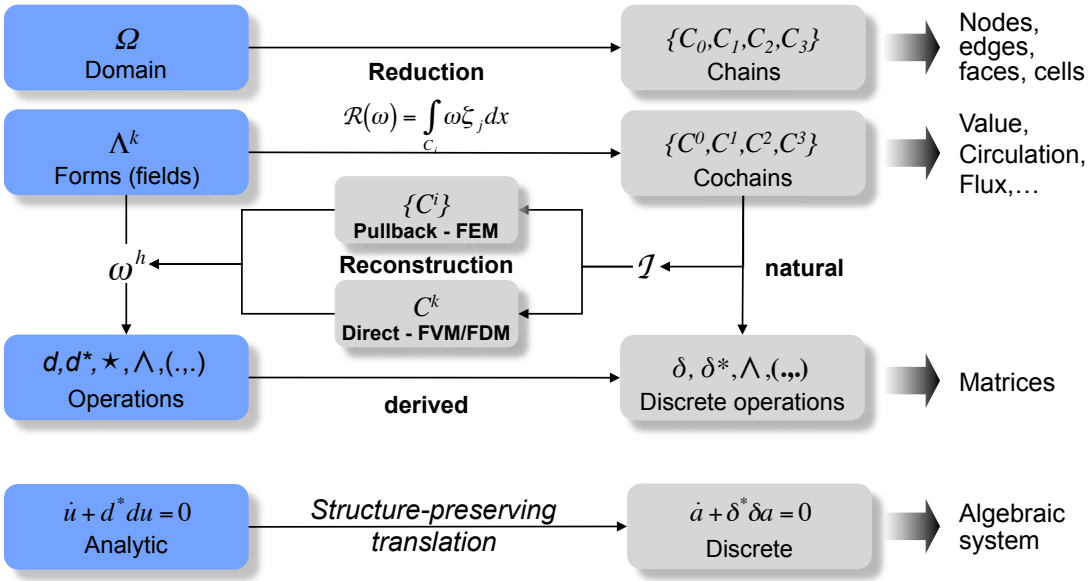


Figure 2: In the abstract framework, the formulation of compatible FEM, FVM and FDM is a generic process in which a particular method results from a particular choice of the reconstruction operator. The conforming reconstruction via pullback to a *reference cell* is typical of finite element methods. Finite volume and finite difference schemes usually employ direct reconstruction in physical coordinates.

assumes the generic form in Figure 2. In other words, our framework reveals a generic discretization pattern shared by a large class of FEM, FVM and FDM. This pattern motivates the design of Intrepid's API.

Intrepid is designed to operate locally on cells or batches of cells having the same topology and data type. In order to support reconstruction operators for a range of data types represented by k-cochains, Intrepid separates cell topology from the reconstruction process; see Figure 3. In other words, the reconstruction "basis" and its evaluation points are not tied to a particular cell topology[2].

This design strategy allows us to "mix and match" cell topologies with reconstruction operators and evaluation points and enables a virtually unlimited generation of new discretization methods from a relatively small number of basic components.

3 Software Design Principles

Intrepid provides extensive numerical functionality for basis functions, geometric transformations, and integration. These routines work on a very wide range of cells (including simplicial and rectangular reference cells). Intrepid does not make any assumptions about the client code's global discretization data structures, e.g., the global mesh connectivity data structure or global sparse linear system data structure.

Our goal is to provide client codes with a robust, extensible, and reusable infrastructure of discretization computations. Our strategy is to provide this functionality through an "expert interface" consisting largely of stateless functions that stream over large chunks of data stored in multidimensional arrays. These arrays contain data associated with homogeneous collections of cells, i.e., cell of the same type, shape, bases, and cubature.

[2]Traditional FE for structural analysis often tie element shape together with a basis type and a set of integration points. For example, Quad4 typically refers to the combination of (1) a quadrilateral cell topology with (2) bilinear Lagrange reconstruction of 0-cochains (vertex values) and (3) 2×2 Gauss integration rule. Intrepid decouples these three notions.

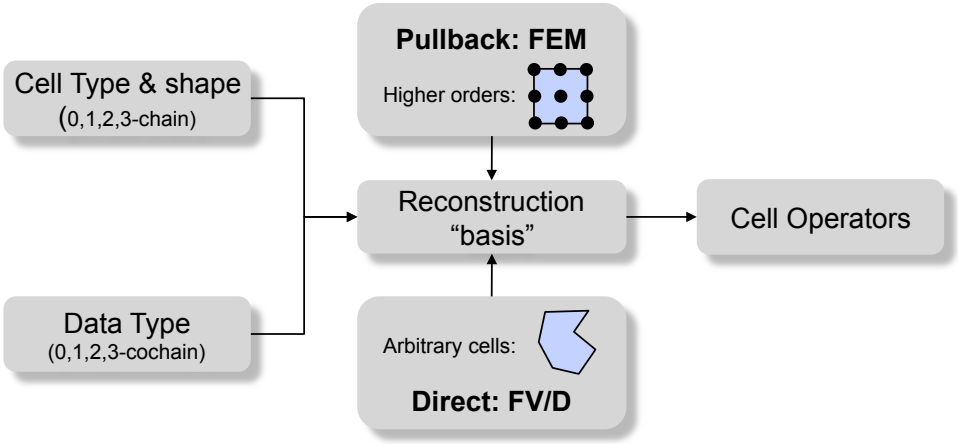


Figure 3: Intrepid's design enables the implementation of a generic discretization pattern that includes compatible FEM, FVM and FDM. Because Intrepid operates locally on cells, it can also support a wider class of discretization methods, such as stabilized and discontinuous FEM as well as spectral methods.

Our strategy for stateless functions with "flat" interfaces has several advantages. First, Intrepid does not control memory and contains no specialized data structures. As such there is no hidden state information that the client code has to be concerned with. Second, Intrepid functions operate over batches of cells, receiving and returning data through a generalized multidimensional array interface. Clients may use Intrepid's extensive library of basis functions and cubature rules, or load their own basis evaluations and cubature rules into arrays. For client codes that use multidimensional arrays this interface eliminates the need to marshal between specialized data structures, which can have a significant performance penalty. Third, this array-based interface allows calculations to be outsourced to level-3 BLAS routines or to optimized tensor contraction functions. Finally, our stateless, array-based compute-intensive functions can be easily migrated to multicore implementations. Our focus on this particular data layout and the resulting simplification of interfaces and program structure (now just loops over arrays) provides an example of data-driven design [26].

In fact, many of our computational functions work at an abstraction level similar to FORTRAN-77. However, a FORTRAN implementation would require users to explicitly (1) manage pointers to memory (with its potential safety pitfalls), (2) create versions of functions for each desired data type (no templates), and (3) conform to a particular multidimensional data layout. Our generalized multidimensional array interface is agnostic with respect to data layout – allowing client code to control or tailor the layout for performance. Furthermore, most of our users architect the higher-level portions of their code in C++, so the C++ interface is appealing to avoid inter-language issues.

We have emphasized testing and verification in Intrepid. We enforce strict preconditions in debug mode and have an extensive suite of unit tests for control flow, statement, decision, condition, modified condition/decision, and loop coverage. We also include higher-level tests tying together multiple functions, such as the testing of higher-order basis functions and numerical integration rules by solving a simple PDE on a single cell and comparing the results to exact solutions.

3.1 Multidimensional Arrays

The multidimensional array (MDArray) is a fundamental concept for managing a wide range of numerical data arising in physics-based simulations in general and the numerical solution of PDEs in particular. An MDArray is a collection of members of a given scalar data type that are identified through a multi-index. A scalar data type is typically a fundamental mathematical type of the computational language, e.g., in C or C++ the double, float, complex, or possibly integer types.

The data type can also be a sophisticated C++ class that supports the standard mathematical behavior of a scalar type. For example, mathematical algorithms can be augmented with automatic differentiation capabilities through "scalar types" that aggregate the evaluation of a mathematical expression and selected partial derivatives of that evaluation. The data members of an MDArray are typically stored in a contiguous block of memory. Contiguous memory is not necessary to the concept of an array; however, contiguity of member storage has proven to be essential to achieve the best performance from cache-based computer memory systems.

The following are typical examples of MDArrays encountered in mesh-based numerical methods for PDEs:

- MDArray of vertex coordinates for mesh cells with multi-index (*cell,vertex,space-dimension*);
- MDArray of coordinates of reference-to-physical cell mapping with multi-index (*cell,space-dimension,coefficient*);
- MDArray of values of a scalar field evaluated at a set of points with multi-index (*point*);
- MDArray of values of a vector field evaluated at a set of points with multi-index (*point,space-dimension*);
- MDArray of values of a scalar finite element basis function evaluated at points with multi-index (*function,point*).

We continue with a brief review of the basic MDArray notions.

Multi-indices and ordinate associations. Each member of a multidimensional array is uniquely identified by a multi-index or ordered list of ordinates; e.g. $(1, 3, 5)$ or $(7, 3, 5, 1)$. Each ordinate of a particular multidimensional array is associated with an entity that arises in the numerical solution of the mathematical problem for which the array is defined. Referring to the examples above, an ordinate may be associated with the axes of a Cartesian space, the number of evaluation points for numerical integration of a cell, the number of cells in a mesh, the number of vertices in a cell, or the number of coefficients in a polynomial function.

Rank. The rank of a multidimensional array is the number of ordinate-indices by which a member of the array is referenced; e.g. the multi-index $(1, 3, 5)$ is rank 3 with three ordinates and $(7, 5, 3, 1)$ is rank 4 with four ordinates. All members of a particular multidimensional array are references with multi-indices of the same rank.

Multi-dimension. The value of each ordinate index i within a multi-index of a particular multidimensional array is defined to be within a span of integer values; for example $i \in [0 \ldots N-1]$ for zero-based indices or $i \in [1 \ldots N]$ for one-based indices. The number N is called the dimension of the ordinate index. The multi-index of all dimensions $(N_1, N_2, \ldots, N_k)$ is called the multi-dimension of the multidimensional array. Given a multi-dimensional array with multi-dimension $(N_1, N_2, \ldots, N_k)$ a multi-index $(i_1, i_2, \ldots, i_k)$ is valid when every ordinate is within its admissible range, and out of bounds otherwise. For example, for zero-based indices the valid indices are as follows:

$$(0, 0, \ldots, 0) \leq (i_1, i_2, \ldots, i_k) < (N_1, N_2, \ldots, N_k)\,,$$

where the comparison is taken ordinate-wise.

Size. The size of a multidimensional array is the product of its multi-dimension: $N_1 \times N_2 \times \ldots \times N_k$. The size gives the total number of members that can be referenced by a valid multi-index. The capacity of the contiguous storage must be equal to or greater than the size of the multidimensional array.

Multi-index mapping. A multi-index into a given array is mapped to a unique members, or offset into the array's contiguous block of memory. The multi-index mapping may be optimized for a particular architecture, for brevity here we review two types of ordering that are currently in widespread use: (1) the lexicographical order and (2) the lexicographical order with reversed significance. For brevity we refer to the former as the *natural* order and the latter as the *reverse* order. Each multi-index ordering induces an associated mapping.

The natural multi-index mapping corresponds to the natural ordering of the multi-index, i.e., the order induced by comparing ordinates in the left-to-right order; for example $(1, 6, 7) < (2, 3, 5) < (2, 3, 6)$. The resulting multi-index map for an array of size (N_1, N_2, N_3) has the following analytic expression:

$$\text{offset} = \text{naturalMap}(i_1, i_2, i_3) = i_1 * N_2 * N_3 + i_2 * N_3 + i_3 \,.$$

The reverse multi-index mapping is, as the name implies, induced by the reverse ordering of the multi-index, i.e., the order induced by comparing ordinates in the right-to-left order. In this case, referring to the example above, $(2, 3, 5) < (2, 3, 6) < (1, 6, 7)$. The resulting multi-index map for an array of size (N_1, N_2, N_3) has the following analytic expression:

$$\text{offset} = \text{reverseMap}(i_1, i_2, i_3) = i_1 + i_2 * N_1 + i_3 * N_1 * N_2 \,.$$

Reverse multi-index mapping is used in the FORTRAN programming language.

3.1.1 MDArrays in Intrepid

Multidimensional arrays of scalar types are the *only* data abstraction in Intrepid. Most C++ functions in Intrepid are templated on the scalar and MDArray types. An algorithm's performance depends on its multi-index mapping to contiguous storage. As such a multi-index mapping is typically chosen for the best algorithm performance. Currently, software projects for numerical PDEs rely almost exclusively on the natural and reverse mappings described above. However, with the advent of new multicore and manycore architectures it is likely that other mappings, optimized for these architectures, will be required.

Intrepid assumes the natural multi-index ordering convention for the MDArray template arguments. As a result, client codes that adhere to the same convention can use Intrepid through a minimal interface, which provides the following methods:

- **int** rank() — returns array rank;
- **int** dimension(dim_i) — returns the ith array dimension (dimensions are dim_1, dim_2, ..., dim_k);
- **int** size () — returns array size, i.e., dim_1∗dim_2∗...∗dim_k;
- **const** Scalar& **operator**(i_1, i_2 ,..., i_k) — const accessor using multi-index;
- Scalar& **operator**(i_1, i_2 ,..., i_k) — non-const accessor using multi-index;
- **const** Scalar& **operator**[i] — const accessor using the ordinal of the array element;
- Scalar& **operator**[i] — non-const accessor using the ordinal of the array element.

The interoperability with codes that employ alternative ordering conventions requires an additional layer which translates their multi-indices to multi-indices with the natural order.

For example, consider a multidimensional array associated with entities A, B, and C with dimensions N_A, N_B, and N_C. Assume that this array uses the reverse ("FORTRAN-style") ordering:

$$\text{offset} = \text{reverseMap}(i_C, i_B, i_A) = i_C + i_B * N_C + i_A * N_C * N_B \,.$$

Note that the reverse multi-index mapping that orders the entities $C - B - A$ defines the same layout as a a natural multi-index mapping that orders the entities $A - B - C$:

$$\text{offset} = \text{naturalMap}(i_A, i_B, i_C) = i_A * N_B * N_C + i_B * N_C + i_C \,.$$

Thus the contiguous memory storage for natural and reverse multi-index mappings is identical and interoperable as long as one mapping's ordinate associations are reversed when compared to the other mapping. Interfacing between Intrepid and clients that use other ordering conventions requires an appropriate multi-index permutation, which maps their multi-indices to Intrepid's default lexicographical multi-index ordering.

In addition to the generic index and dimension notation i_k and dim_k Intrepid uses a naming convention for data-specific indices and dimensions of MDArrays that arise in mesh-based numerical methods for PDEs; see Table 1. Table 2 shows a few MDArrays used typically in numerical methods for PDEs.

Index type	Dimension	Description
point	P	number of points stored in an MDArray
vertex	V	number of nodes stored in an MDArray
(basis) field	F	number of fields stored in an MDArray
bilinear form field	L/R	number of left/right basis fields stored in an MDArray
cell	C	number of cells stored in an MDArray
field coordinate	D	space dimension
derivative ordinal	K	cardinality of the set of kth derivatives

Table 1: Data-specific indices in Intrepid.

Rank	Multi-dimension	Multi-index	Description
1	(P)	(p)	Scalar (rank 0) field evaluated at P points
2	(P,D)	(p, d)	Vector (rank 1) field evaluated at P points
3	(P,D,D)	(p, d, d)	Tensor (rank 2) field evaluated at P points
2	(F,P)	(f, p)	F scalar fields evaluated at P points
3	(F,P,D)	(f, p, d)	F vector fields evaluated at P points
4	(F,P,D,D)	(f, p, d, d)	F tensor fields evaluated at P points
3	(F,P,K)	(f, p, k)	kth derivative of F scalar fields evaluated at P points
4	(F,P,D,K)	(f, p, d, k)	kth derivative of F vector fields evaluated at P points
5	(F,P,D,D,K)	(f, p, d, d, k)	kth derivative of F tensor fields evaluated at P points
3	(C,V,D)	(c, v, d)	Vertex coordinates of C cells having V vertices each
3	(C,P,D)	(c, p, d)	Coordinates of points in C cells, P per cell

Table 2: Typical MDArrays arising in mesh-based methods for PDEs.

3.2 Cell Topology

FEM, FVM and FDM rely on grids comprised of D-dimensional polytopes (lines in 1D, polygons in 2D and polyhedrons in 3D). Different disciplines refer to the mesh polytopes as finite elements, finite volumes, zones, or cells. In Intrepid we use the term "cell" for any valid mesh polytope.

With every polytope (cell) we associate a default chain complex comprised of the D-cell itself and its $0, 1, \ldots, D-1$ dimensional subcells. We refer to this complex as the *cell topology*. Specification of cell topology for a given cell shape requires a choice of (local) 0-subcell (vertex) numbering and definition of upward and downward adjacency relations between the cell and its $0, 1, \ldots, D-1$ dimensional subcells. A cell topology defines a polytope in terms of vertex connectivity (e.g., line, triangle, quadrilateral, tetrahedron). However, the cell topology does not define a specific geometric realization of that entity, i.e., it does not associate spatial coordinates with any of the vertices or other points potentially defined for the polytope.

A cell topology implies a topological dimension D which is the smallest spatial dimension in which that cell can be realized. For example, a point has $D = 0$, line has $D = 1$, triangles and quadrilaterals have $D = 2$, and tetrahedron has $D = 3$. Note that for some discretizations the actual dimension of a cell may be greater than its implied topological dimension, as in shell elements. A cell can only be realized in a space with dimension equal to or greater than the cell dimension; i.e., a tetrahedron cannot be realized in a 2D space.

The Shards package implements the default cell topologies that are used in Intrepid. The standard Shards cell topologies are line, triangle, quadrilateral, hexahedron, tetrahedron, wedge, and pyramid. Polygon and polyhedron are non-standard topologies. Every standard cell topology in Shards has a *base* version and an *extended* version, whereas non-standard topologies have only base versions. Extended cell topologies enable the mapping of reference cells to curvilinear cells in physical coordinates. In some classical finite element methods such topologies are also used as a means of identifying degrees of freedom. Intrepid does not support this viewpoint and enforces strict separation of cell topology, basis and integration objects.

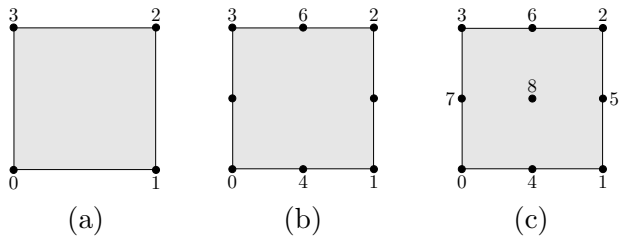


Figure 4: (a) Base Quadrilateral <4>, (b) eight-node extended Quadrilateral <8>, and (c) nine-node extended Quadrilateral <9> topologies in Shards.

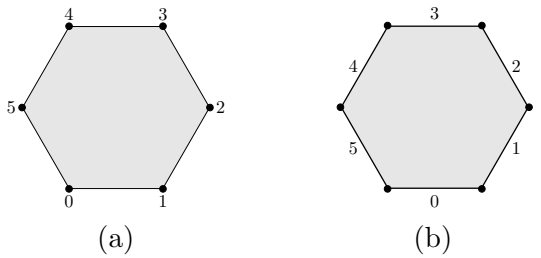


Figure 5: (a) Base Hexagon<> topology and (b) edge (1-cell) numbering in Shards.

Table 3 summarizes the standard topologies in the Shards package. Figure 4 illustrates a typical standard cell topology, whereas Figure 5 shows an example of a non-standard hexagon topology and its 1-cell (edge) numbering. Only a base topology is provided for this cell.

4 Structure of Intrepid

In this section we describe the functionality of key Intrepid classes. We distinguish two main types of classes: (1) classes designed for the one-time tabulation and subsequent reuse of mathematical fields and (2) classes designed for the high-performance processing of field data, via stream computing.

The classes of the first type are Intrepid's Basis and Cubature classes. Their implementations rely on abstract (pure virtual) base classes to provide consistent interfaces, and carry varying amounts of data. The classes of the second type are RealSpaceTools, ArrayTools, CellTools and FunctionSpaceTools. They are implemented as collections of static methods, fully stateless and geared toward the fast processing of data stored in MDArrays.

Additionally, Intrepid provides an implementation of the MDArray, called FieldContainer.

Dim.	Base Topology	Extended Topology
1	Line<>	Line<3>
2	Triangle<>	Triangle<6>
3	ShellTriangle<>	ShellTriangle<6>
2	Quadrilateral<>	Quadrilateral<8>, Quadrilateral<9>
3	ShellQuadrilateral<>	ShellQuadrilateral<8>, ShellQuadrilateral<9>
3	Hexahedron<>	Hexahedron<20>, Hexahedron<27>
3	Tetrahedron<>	Tetrahedron<10>
3	Pyramid<>	Pyramid<13>, Pyramid<14>
3	Wedge<>	Wedge<15>, Wedge<18>

Table 3: Summary of standard Shards cell topologies.

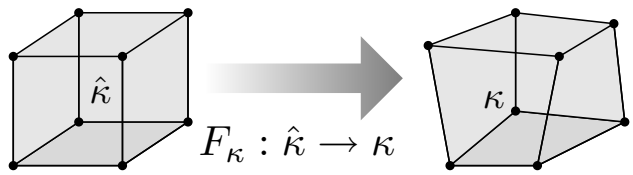


Figure 6: The reference-to-physical mapping F_κ is used commonly in FEM and defines the *pullback* mechanism, see Figures 2 and 3 and Section 4.6. We refer to the cell $\widetilde{\kappa}$ in reference frame as the *reference cell*, while the cell κ in physical frame is called the *physical cell*.

4.1 The Basis Class

Using the nomenclature from Section 2, the main purpose of the Basis class is to evaluate reconstruction operators $\mathcal{I}$ at a specified set of points given inside a cell. We recall that the abstract framework in Section 2 associates a particular discretization method with a particular choice of a reconstruction operator acting on cochains. Our Basis is a pure virtual class, with nearly 40 concrete implementations available as of Trilinos 10.8 (2011). As an example, our basis collection includes finite element bases of orders 1 through 10 in $H(\text{grad})$, $H(\text{div})$ and $H(\text{curl})$ on triangles, quadrilaterals, tetrahedrons and hexahedrons. The evaluation of fields is performed using the getValues() methods. While a number of other routines comprise the Basis interface, in this section we focus on the description of the getValues() methods. Additionally, we discuss the naming conventions behind a few concrete Basis implementations.

As emphasized in Section 2, Intrepid is designed to support a variety of PDE discretizations, including FEM, FVM, FDM and hybrids[3]. FEM typically evaluate fields on a reference cell and employ an explicit mapping between the reference cell and physical cells, see Figure 6, to define the so-called *pullback* mechanism. In contrast, FVM, FDM and polygonal FEM evaluate fields directly at points inside physical cells. For this reason, our abstract Basis class accommodates two distinct field evaluation interfaces.

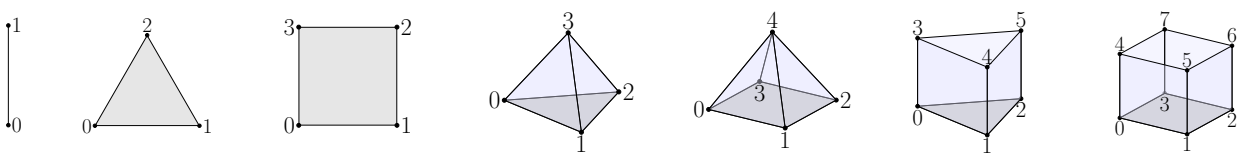
```
template<class Scalar, class ArrayScalar>
class Basis {
protected:
  shards::CellTopology basisCellTopology_;
  ...
public:

  virtual void getValues(ArrayScalar &         outputValues,
                         const ArrayScalar &   inputPoints,
                         const EOperator       operatorType) const = 0;
```

[3]For an example of hybrid methods, see control-volume FEM (CVFEM), Section 6.3.

```
  virtual void getValues(ArrayScalar &          outputValues,
                         const ArrayScalar &    inputPoints,
                         const ArrayScalar &    cellVertices,
                         const EOperator        operatorType) const = 0;
  ...
};
```

We note that in both cases, the topology of the cell in which the fields are evaluated is given by the basisCellTopology_ data member. In the case of methods based on the reference-to-physical mapping, this is sufficient to infer the spatial coordinates of cell vertices, as we follow the reference cell convention given in Table 4. In the second case, cell vertices are supplied as an additional argument, and follow the multi-index convention (V,D). The $H(\text{grad})$ polygonal finite element basis in Intrepid is an example of a reconstruction operator that relies on the second field evaluation interface. We refer to Section 6.1 for additional details about polygonal FEM in Intrepid. The MDArrays inputPoints are always indexed by (P,D), while outputValues are MDArrays whose rank depends on the basis type and the selected EOperator input: OPERATOR_VALUE, OPERATOR_GRAD, OPERATOR_CURL, OPERATOR_DIV or OPERATOR_D1 through OPERATOR_D10.[4] For example, for an $H(\text{grad})$ finite element basis and the input operator OPERATOR_GRAD the MDArray outputValues is always indexed by (F,P,D), while the operator OPERATOR_VALUE would yield the (F,P) indexing.



Dim.	Base Topology	Vertex Coordinates
1	Line<>	(-1,0,0), (1,0,0)
2	Triangle<>	(0,0,0), (1,0,0), (0,1,0)
2	Quadrilateral<>	(-1,-1,0), (1,-1,0), (1,1,0), (-1,1,0)
3	Tetrahedron<>	(0,0,0), (1,0,0), (0,1,0), (0,0,1)
3	Pyramid<>	(-1,-1,0), (1,-1,0), (1,1,0), (-1,1,0), (0,0,1)
3	Wedge<>	(0,0,-1), (1,0,-1), (0,1,-1), (0,0,1), (1,0,1), (0,1,1)
3	Hexahedron<>	(-1,-1,-1), (1,-1,-1), (1,1,-1), (-1,1,-1), (-1,-1,1), (1,-1,1), (1,1,1), (-1,1,1)

Table 4: Geometric definitions of basic reference cells in Intrepid. Reference cells based on extended topologies from the Shards package are also supported.

The naming of most Intrepid's bases follows a simple convention:

FUNCTIONSPACE_CELLTYPE_(C)OMPLETEor(I)NCOMPLETEandORDER_FEMorFVMorFDM.

The third field requires an explanation. The keywords (C)OMPLETE and (I)NCOMPLETE denote whether the basis forms a complete or an incomplete polynomial space, of given order. The keyword ORDER is either a number, 1 or 2, or the letter n, in which case the order can be set by the user. For example, HGRAD_TRI_C2_FEM is an FE basis in $H(\text{grad})$, defining a complete polynomial space of order 2 on triangles — in other words: *nodal*, piecewise quadratic basis on triangles. Another example is HCURL_HEX_In_FEM, an FE basis in $H(\text{curl})$, defining an incomplete polynomial space of user-supplied order on hexahedrons — in other words: *edge element*[5] basis of variable order on hexahedra.

[4]The compatibility of operators and bases follows the mathematics of Section 2 and is verified at runtime.

[5]Intrepid implements Nedelec edge bases of the first kind, see [21, 22].

The higher-order finite element bases for simplicial domains are C++ realizations of the generic basis construction procedure developed in FIAT [17] for reference element bases. This reimplementation allows Intrepid to remain pure C++ and avoid embedding Python within a C++ library. The FIAT framework, based on the Ciarlet triple [11] of a cell, a finite-dimensional function space, and a dual basis, constructs the nodal basis from linear combinations of orthogonal polynomials via a kind of generalized Vandermonde matrix. This approach allows construction of high-order bases for the de Rham complex of spaces on the simplex. Basis functions for the rectangular domains rely on appropriate tensor products of one-dimensional bases. In all cases, users may specify the nodal points (e.g. equispaced lattice points, Gauss-Lobatto and Warburton's warp-blend points [34]) used as degrees of freedom, which can strongly influence the accuracy and conditioning of the basis as order increases.

4.2 The Cubature Class

The Cubature class provides a simple abstract interface for the implementation of numerical integration (cubature) rules in Intrepid. In addition to the abstract interface, a variety of concrete implementations are provided. For example, as of Trilinos 10.8 (2011), the exact integration of polynomials of degree 20 is possible for triangles and tetrahedrons, while polynomials of degree 60 can be accurately integrated on lines, quadrilaterals and hexahedrons.

The Cubature interface is specified below. The key method is getCubature(), which returns MDArrays with cubature points and cubature weights, indexed by (P,D) and (P), respectively.

```
template<class Scalar, class ArrayPoint, class ArrayWeight>
class Cubature {
  public:

  virtual void getCubature(ArrayPoint  & cubPoints,
                           ArrayWeight & cubWeights) const = 0;

  virtual int getNumPoints() const = 0;

  virtual int getDimension() const = 0;

  virtual void getAccuracy(std::vector<int> & accuracy) const = 0;

};
```

Examples of implementations of this interface are the CubatureDirect and CubatureTensor classes. CubatureDirect is a partial interface implementation with concrete subclasses tied to simplicial cell topologies, such as CubatureDirectLineGauss, CubatureDirectTriDefault and CubatureDirectTetDefault. CubatureTensor is a full implementation of the base class, designed specifically for the efficient assembly of tensor products of arbitrary integration rules.

While the concrete Cubature implementations can be used directly, Intrepid provides a single interface for the construction of integration rules, called DefaultCubatureFactory. This factory class facilitates a vast majority of use cases for cubatures. Its create() methods allow the user to select cubatures based only on cell topology and the desired order of exact polynomial integration.

```
  Teuchos::RCP<Cubature<Scalar,ArrayPoint,ArrayWeight> >
      create(const shards::CellTopology & cellTopology,
             const std::vector<int> & degree);

  Teuchos::RCP<Cubature<Scalar,ArrayPoint,ArrayWeight> >
      create(const shards::CellTopology & cellTopology,
             int   degree);
```

In this excerpt, the first create()[6] method is specific to tensor products of cubature rules, enabling

[6]We note that Teuchos::RCP is a reference-counted pointer provided by Trilinos' Teuchos package.

varying degrees of accuracy in each component rule. The second method enables the instantiation of direct and uniform-degree tensor-product rules, and can be used with all basic cell shapes.

4.3 The RealSpaceTools Class

The RealSpaceTools class is responsible for basic linear algebraic operations in $\mathbb{R}^1$, $\mathbb{R}^2$ and $\mathbb{R}^3$, such as vector addition, matrix-vector multiplications, matrix transposition and inversion, and several others. It is a stateless class, comprised only of static (again, stateless) member functions. The computational kernels supplied by RealSpaceTools are used primarily by the class CellTools.

4.4 The ArrayTools Class

The class ArrayTools is one of two computational cores of Intrepid. It provides methods for efficient, stream-based algebraic operations on user-defined MDArrays, such as tensor contraction, scaling and replication. The ArrayTools class services the high-level class FunctionSpaceTools, and is therefore not exposed to the end user within a typical Intrepid workflow. Nonetheless, it is important to discuss its design, as it exemplifies the principles outlined in Section 3.

The ArrayTools class is a collection of stateless methods with flat interfaces. To ensure compatibility with generic, user-controlled data containers, it employs polymorphism based on (i) C++ templates and (ii) the MDArray abstraction, which defines container access. As an example, we quote the signature of the tensor contraction method contractFieldFieldScalar (), which is called internally when computing matrix discretizations of bilinear forms, resulting in cell mass matrices.

```
template<class Scalar, class ArrayOutFields,
         class ArrayInFieldsLeft, class ArrayInFieldsRight>
static void contractFieldFieldScalar(ArrayOutFields &            outputFields,
                                     const ArrayInFieldsLeft &   leftFields,
                                     const ArrayInFieldsRight &  rightFields,
                                     const ECompEngine           compEngine,
                                     const bool                  sumInto = false);
```

We remark that this method is templated on all its array arguments. This facilitates a generic implementation of the contraction, independent of the layout of user data.

```
...
for (int cl = 0; cl < numCells; cl++) {
  for (int lbf = 0; lbf < numLeftFields; lbf++) {
    for (int rbf = 0; rbf < numRightFields; rbf++) {
      Scalar tmpVal(0);
      for (int qp = 0; qp < numPoints; qp++) {
        tmpVal += leftFields(cl, lbf, qp)*rightFields(cl, rbf, qp);
      }
      outputFields(cl, lbf, rbf) = tmpVal;
    }
  }
}
...
```

While this is a commonly used implementation, enabled by setting the value of the ECompEngine argument to COMP_CPP, Intrepid provides optimized implementations, for MDArrays with contiguous physical layouts, via the BLAS. For our BLAS implementations of contraction routines, set the ECompEngine argument to COMP_BLAS.

4.5 The CellTools Class

The CellTools class is the second of two computational cores of Intrepid. It provides efficient methods for a variety of geometric operations on reference cells and physical cells. As a rule, these methods are *fully decoupled* from the notions of basis and cubature, but may use Intrepid bases

for internal computations and may use user-supplied cubature points as evaluation points. The methods include:

- computation of Jacobian matrices DF_κ for reference-to-physical frame mappings F_κ, see Figure 6, their inverses DF_κ^{-1} and determinants $\det(DF_\kappa)$,
- application of the reference-to-physical frame mapping F_κ and its inverse F_κ^{-1} to points within reference and physical cells, respectively,
- parametrization of edges and faces of reference cells,
- computation of edge/face tangents and face normals in reference and physical frames, and
- inclusion tests for point sets in reference and physical cells.

Similar to ArrayTools, the CellTools class is a collection of stateless methods with flat interfaces, which efficiently process user-controlled MDArrays. In contrast to ArrayTools, the CellTools class comprises an important part of Intrepid's main user interface, in other words typical Intrepid workflow involves direct calls to the CellTools methods. As of Trilinos 10.8 (2011) there are more than 20 such methods. We describe in detail one that is used very often. The descriptions of all methods, treated with similar rigor and level of detail, are given in Intrepid's documentation [1].

The setJacobian() method computes Jacobian matrices of reference-to-physical frame mappings that are commonly used in FEM.

```
template<class ArrayJac, class ArrayPoint, class ArrayCell>
static void setJacobian(ArrayJac &                    jacobian,
                        const ArrayPoint &            points,
                        const ArrayCell  &            cellWorkset,
                        const shards::CellTopology &  cellTopo,
                        const int &                   whichCell = -1);
```

There are three use cases:

(i) computation of Jacobians DF_κ, stored as the jacobian(P,D,D) array, for a specified physical cell κ from the cell workset array cellWorkset(C,V,D) on a single set of reference points stored in the points(P,D) array;

(ii) computation of Jacobians DF_κ, stored as the jacobian(C,P,D,D) array, for all physical cells in the cell workset array cellWorkset(C,V,D) on a single set of reference points stored in the points(P,D) array;

(iii) computation of Jacobians DF_κ, stored as the jacobian(C,P,D,D) array, for all physical cells in the cell workset array cellWorkset(C,V,D) on multiple reference point sets having the same number of points, given by the points(C,P,D) array.

In case (i), for a single point set and the parameter whichCell set to a valid cell ordinal relative to the cell workset, the method returns a Jacobian array such that

$$\mathsf{jacobian}(p,\cdot,\cdot) = [DF_\kappa(\mathsf{points}(p)] \quad \text{for a fixed cell } \kappa \text{ with index } 0 \le c < \mathsf{C}-1.$$

In case (ii), for a single point set and the parameter whichCell set to -1 (default value), the method returns a Jacobian array such that

$$\mathsf{jacobian}(c,p,\cdot,\cdot) = [DF_\kappa(\mathsf{points}(p)] \quad \text{for all cells } \kappa \text{ with indices } c = 0,\ldots,\mathsf{C}-1.$$

In case (iii), for multiple sets of reference points and the parameter whichCell set to -1 (default value), the method returns a Jacobian array such that

$$\mathsf{jacobian}(c,p,\cdot,\cdot) = [DF_\kappa(\mathsf{points}(c,p)] \quad \text{for all cells } \kappa \text{ with indices } c = 0,\ldots,\mathsf{C}-1.$$

In all of the above, a default reference-to-physical map F_κ is assumed by Intrepid for the selected cell topology argument cellTopo.

4.6 The FunctionSpaceTools Class

The FunctionSpaceTools class combines the computational kernels provided by the ArrayTools class into a high-level user-friendly interface with mathematical semantics. There are four types of methods:

- transformation of fields from reference frame to physical frame,
- computation of integral measures needed for edge, face and cell integration,
- scalar, vector and tensor multiplication, and
- numerical integration and evaluation (interpolation).

They are designed primarily for operations on finite element subspaces of $H(\text{grad})$, $H(\text{div})$, $H(\text{curl})$ and $H(\text{vol}) = L^2$. A subset of measure computation, numerical integration and multiplication routines can be applied directly to user data, i.e. in non-FEM contexts, such as the one described in Section 6.4. Here we focus on the FEM use case, for which the typical workflow, discussed in detail in Section 5, is:

$$\boxed{\text{Field Transformation (Pullback)}} \longrightarrow \boxed{\text{Measure Computation}} \longrightarrow \boxed{\text{Multiplication}} \longrightarrow \boxed{\text{Integration}}\ .$$

4.6.1 Field Transformation (Pullback)

In Ciarlet's notation [11], given a reference cell $\{\widehat{\kappa}, \widehat{P}, \widehat{\Lambda}\}$ with a basis $\{\widehat{u}_i\}_{i=1}^n$, the basis $\{u_i\}_{i=1}^n$ of $\{\kappa, P, \Lambda\}$ is defined as follows:

$$u_i = \sigma_i \Phi^*(\widehat{u}_i), \qquad i = 1, \ldots, n\,.$$

In this formula, $\{\sigma_i\}_{i=1}^n$ are the *field signs*, with $\sigma_i = \pm 1$, and Φ^* is the pullback, i.e. the "change of variables" transformation. For scalar spaces such as $H(\text{grad})$ and $H(\text{vol})$ the field signs are always equal to 1 and can be disregarded. For vector field spaces such as $H(\text{curl})$ or $H(\text{div})$, the field sign of a basis function can be $+1$ or -1, depending on the orientation of the physical edge or face associated with the basis function.

The form of the pullback depends on which one of the four function spaces $H(\text{grad})$, $H(\text{div})$, $H(\text{curl})$ or $H(\text{vol})$ is approximated. Let F_κ denote the reference-to-physical map, let DF_κ denote its Jacobian and let $J_\kappa = \det(DF_\kappa)$. Then the pullbacks are defined, and given by the FunctionSpaceTools methods, as follows:

$$\begin{array}{lll}
\Phi^*_G : H(\text{grad}, \widehat{\kappa}) \mapsto H(\text{grad}, \kappa) & \Phi^*_G(\widehat{u}) = \widehat{u} \circ F_\kappa^{-1} & \left\{ \begin{array}{l} \text{HGRAD_transform_VALUE()} \end{array} \right. \\[2ex]
\Phi^*_C : H(\text{curl}, \widehat{\kappa}) \mapsto H(\text{curl}, \kappa) & \Phi^*_C(\widehat{\mathbf{u}}) = \big((DF_\kappa)^{-\mathsf{T}} \cdot \widehat{\mathbf{u}}\big) \circ F_\kappa^{-1} & \left\{ \begin{array}{l} \text{HGRAD_transform_GRAD()} \\ \text{HCURL_transform_VALUE()} \end{array} \right. \\[2ex]
\Phi^*_D : H(\text{div}, \widehat{\kappa}) \mapsto H(\text{div}, \kappa) & \Phi^*_D(\widehat{\mathbf{u}}) = \big(J_\kappa^{-1} DF_\kappa \cdot \widehat{\mathbf{u}}\big) \circ F_\kappa^{-1} & \left\{ \begin{array}{l} \text{HCURL_transform_CURL()} \\ \text{HDIV_transform_VALUE()} \end{array} \right. \\[2ex]
\Phi^*_V : H(\text{vol}, \widehat{\kappa}) \mapsto H(\text{vol}, \kappa) & \Phi^*_V(\widehat{u}) = \big(J_\kappa^{-1} \widehat{u}\big) \circ F_\kappa^{-1} & \left\{ \begin{array}{l} \text{HDIV_transform_DIV()} \\ \text{HVOL_transform_VALUE()} \end{array} \right. \quad .
\end{array}$$

As an example, we consider the HGRADtransformGRAD() method.

```
template<class Scalar, class ArrayTypeOut, class ArrayTypeJac, class ArrayTypeIn>
static void HGRADtransformGRAD(ArrayTypeOut       & outVals,
                               const ArrayTypeJac & jacobianInverse,
                               const ArrayTypeIn  & inVals,
                               const char           transpose = 'T');
```

This method computes the pullbacks of the gradients of $H(\text{grad})$ functions,

$$\Phi^*(\nabla \widehat{u}_f) = \Big((DF_\kappa)^{-\mathsf{T}} \cdot \nabla \widehat{u}_f \Big) \circ F_\kappa^{-1} \,,$$

for points in physical cells that are images of a given set of points in the reference cell,

$$\{x_{c,p}\}_{p=0}^{\mathsf{P}-1} = \{F_\kappa(\widehat{x}_p)\}_{p=0}^{\mathsf{P}-1},$$

for all cells κ with indices $c = 0, \ldots, \mathsf{C}-1$. The user-provided array $\mathsf{inVals}(\mathsf{F},\mathsf{P},\mathsf{D})$ must contain the gradients of the function set $\{\widehat{u}_f\}_{f=0}^{\mathsf{F}-1}$ at the reference points,

$$\mathsf{inVals}(f,p,\cdot) = \nabla \widehat{u}_f(\widehat{x}_p) \,.$$

If the argument transpose is set to 't' or 'T', the user-provided array $\mathsf{jacobianInverse}(\mathsf{C},\mathsf{P},\mathsf{D},\mathsf{D})$ must be an array of inverses of Jacobians of the mapping F_κ computed at the reference points,

$$\mathsf{jacobianInverse}\,(c,p,\cdot,\cdot) = (DF_\kappa)^{-1}(\widehat{x}_p) \,,$$

otherwise, if transpose is set to 'n' or 'N', we must have

$$\mathsf{jacobianInverse}\,(c,p,\cdot,\cdot) = (DF_\kappa)^{-\mathsf{T}}(\widehat{x}_p) \,,$$

for all cells κ with indices $c = 0, \ldots, \mathsf{C}-1$. The method returns the array $\mathsf{outVals}(\mathsf{C},\mathsf{F},\mathsf{P},\mathsf{D})$ such that

$$\mathsf{outVals}(c,f,p,\cdot) = \Big((DF_\kappa)^{-\mathsf{T}} \cdot \nabla \widehat{u}_f \Big) \circ F_\kappa^{-1}(x_{c,p}) = (DF_\kappa)^{-\mathsf{T}}(\widehat{x}_p) \cdot \nabla \widehat{u}_f(\widehat{x}_p) \,,$$

for all cells κ with indices $c = 0, \ldots, \mathsf{C}-1$.

Other field transformation methods follow a similar pattern and are described in great detail in Intrepid's documentation [1]. For the application of field signs in function spaces $H(\text{curl})$ and $H(\text{div})$, Intrepid offers the applyFieldSigns (), applyLeftFieldSigns () and applyRightFieldSigns () methods.

4.6.2 Measure Computation

Integrals of finite element functions over cells, 2-subcells (faces) and 1-subcells (edges) are computed via the change of variables from physical to reference frame and require three types of integral measures. The integral of a scalar function over a cell κ,

$$\int_\kappa f(x)dx = \int_{\widehat{\kappa}} f(F_\kappa(\widehat{x}))|J_\kappa| d\widehat{x} \,,$$

requires the volume measure defined by the determinant of the Jacobian. This measure is computed by the computeCellMeasure() method. The integral of a scalar function over 2-subcell c_2,

$$\int_{c_2} f(x)dx = \int_R f(\Phi(u,v)) \left\| \frac{\partial \Phi}{\partial u} \times \frac{\partial \Phi}{\partial v} \right\| du\, dv \,,$$

requires the surface measure defined by the norm of the vector product of the surface tangents. This measure is computed by the computeFaceMeasure() method. In this formula R is a *parametrization domain* for the 2-subcell c_2, in other words a reference triangle or a reference quadrilateral, while Φ is a map from R to the physical 2-subcell c_2. The integral of a scalar function over a 1-subcell c_1,

$$\int_{c_1} f(x)dx = \int_R f(\Phi(s)) \|\Phi'\| ds \,,$$

requires the arc measure defined by the norm of the arc tangent vector. This measure is computed by the computeEdgeMeasure() method. In this formula R is the parametrization domain for the 1-subcell c_1, in other words a reference line, while Φ is a map from R to the physical 1-subcell c_1.

To enable the computation of cell measures the user is required to provide an array containing the determinants of Jacobians. For face and edge measures the user must supply the array of Jacobian matrices, the face or edge ordinal and the topology of the parent cell.

```
template<class Scalar, class ArrayOut, class ArrayJac, class ArrayWeights>
static void computeFaceMeasure(ArrayOut                    & outVals,
                               const ArrayJac              & inJac,
                               const ArrayWeights          & inWeights,
                               const int                     whichFace,
                               const shards::CellTopology  & parentCell);
```

Finally, it is important to note that in all three cases the measure computation is combined with a multiplication by the weights of the cubature rule that is used to subsequently evaluate the integral. In other words, the resulting measures are always *weighted measures*.

4.6.3 Multiplication

The FunctionSpaceTools class provides a variety of methods for algebraic operations on MDArrays with user data and finite element function values. These methods are used to 'prepare' such MDArrays for the integration routines. They include

(i) scalar multiplication: scalarMultiplyDataField () and scalarMultiplyDataData();

(ii) dot product: dotMultiplyDataField() and dotMultiplyDataData();

(iii) vector (cross or outer) product: vectorMultiplyDataField () and vectorMultiplyDataData();

(iv) tensor product: tensorMultiplyDataField () and tensorMultiplyDataData(); and

(v) measure application: multiplyMeasure() .

The separation of multiplication routines from integration is key to Intrepid's expressiveness.

4.6.4 Integration and Evaluation

Intrepid's treatment of integration and evaluation is quite simple — they are algebraic contractions of exactly *two* MDArrays.

In general, integration takes the input arguments leftValues and rightValues, indexed by cell (and, optionally, field) indices and point (and, optionally, dimension) indices, and contracts the point and dimension indices. The output array outputValues retains the cell index and any field indices from the input arrays, and loses the point and dimension indices.

```
template<class Scalar, class ArrayOut, class ArrayInLeft, class ArrayInRight>
static void integrate(ArrayOut            & outputValues,
                      const ArrayInLeft   & leftValues,
                      const ArrayInRight  & rightValues,
                      const ECompEngine     compEngine,
                      const bool            sumInto = false);
```

In addition to the high-level routine integrate (), three low-level methods are provided:

operatorIntegral (), functionalIntegral () and dataIntegral ().

Computational patterns for integration are presented in detail in Sections 5.1 and 5.2.

Similar to integration, evaluation takes two input arguments. The array inCoeffs is indexed by a cell index and a field index, while the input array inFields additionally includes a point index and, optionally, one or two dimension indices. Evaluation contracts away the field index. In other words, the output array outPointVals retains the cell index and the point (and dimension) indices, and loses the field index.

```
template<class Scalar, class ArrayOutPointVals,
         class ArrayInCoeffs, class ArrayInFields>
static void evaluate(ArrayOutPointVals     & outPointVals,
                     const ArrayInCoeffs   & inCoeffs,
                     const ArrayInFields   & inFields);
```

A computational pattern for evaluation is presented in detail in Section 5.3.

4.7 The FieldContainer Class

Intrepid's FieldContainer class provides an implementation of the MDArray abstraction — as such, its user interface is defined in Section 3.1. The primary purpose of FieldContainer is to enable efficient development and testing of Intrepid's functionality. In other words, it is first and foremost a tool for Intrepid developers. Nonetheless, the FieldContainer class can be used in high-performance application codes that do not provide their own MDArray implementation.

A FieldContainer object is a lexicographically ordered container that stores a multi-indexed scalar quantity: the rightmost index changes first and the leftmost index changes last. FieldContainer can be viewed as a dynamic multidimensional array whose values can be accessed in two ways: by their multi-index or by their enumeration, using an overloaded [ ] operator. The enumeration of a value gives the sequential order of the multi-indexed value in the container. The number of indices, i.e., the rank of the container is unlimited. For containers with ranks up to five many of the methods are optimized for faster execution. An overloaded () operator is also provided for low-rank containers to allow element access by multi-index without having to create an auxiliary ordinal array for the multi-index.

The FieldContainer class offers some convenience features such as the ability to resize and reshape FieldContainer objects at runtime. The Trilinos package Shards provides an implementation of the MDArray whose dimensions are set through template arguments at compile time. The Shards MDArray enables strong typing of the MDArray object but does not allow it to be reshaped or resized at runtime.

Finally, we note that the FieldContainer objects are stored contiguously in memory. For this reason, they can be used safely and efficiently with the COMP_BLAS computational engine accessible through the FunctionSpaceTools interface methods.

5 Examples of Computational Patterns

Section 4 describes an extensive range of computational tools with very flexible interfaces. With the help of these tools a user can implement a variety of discretization and non-discretization tasks. In most cases, implementation of these tasks follows a generic pattern that is independent of, e.g., cell shape, integration order or basis type. In this section we examine three typical computational patterns in Intrepid.

5.1 Building a Finite Element Operator

Finite element methods assemble discretization matrices ("finite element operators") for PDEs from element-wise contributions. Computation of these element-wise contributions is the first example of a generic computational pattern in Intrepid. We focus on finite element operators for second-order self-adjoint differential operators such as div(grad), curl(curl) and grad(div). However, with minor modifications the computational pattern extends to other PDEs, such as non-symmetric advection-diffusion equations. For self-adjoint differential operators contributions from element κ_s to the global finite element matrix have the generic form

$$A^s_{lr} = \int_{\kappa_s} \mathcal{L}\phi_l(x)\mathcal{L}\phi_r(x)dx\,, \tag{6}$$

where $\{\phi_i\}_{i=1}^n$ are basis functions, and $\mathcal{L}$ is a linear differential operator.

In a standard finite element setting the basis functions are defined on a reference cell $\widehat{\kappa}$ and their values and derivatives are transformed to physical coordinates using an appropriate pullback transform Φ^*; see Section 4.6.1. As a result, the integral (6) becomes

$$A_{lr}^s = \int_{\kappa_s} \sigma_l \sigma_r \Big(\Phi^*(\widehat{\mathcal{L}}\widehat{\phi}_l)\Big)(x)\Big(\Phi^*(\widehat{\mathcal{L}}\widehat{\phi}_r)\Big)(x)dx\,, \tag{7}$$

where $\sigma_i = \pm 1$ are *field signs*. These are required for vector spaces such as $H(\text{curl})$ and $H(\text{div})$ where the orientation of the physical edge or face may be different from the orientation of the reference edge or face. In the case of $H(\text{grad})$ or $H(\text{vol})$ spaces the signs are always equal to one.

From Section 4.6.1 it follows that the pullback has the form

$$\Phi^* = T(\widehat{\mathcal{L}}\widehat{\phi}_i) \circ F^{-1}\,,$$

where T is a transformation that can depend on the Jacobian DF of the reference-to-physical map, and/or its determinant, J. After a change of variables to reference domain, i.e., application of F^{-1}, the integral in (7) assumes the form

$$A_{lr}^s = \int_{\widehat{\kappa}} \sigma_l \sigma_r \Big(T(\widehat{\mathcal{L}}\widehat{\phi}_l)(\widehat{x})\Big)\Big(T(\widehat{\mathcal{L}}\widehat{\phi}_r)(\widehat{x})\Big)J(\widehat{x})d\widehat{x}. \tag{8}$$

The approximation of (8) using a cubature rule with cubature points $\{\widehat{x}_p\}_{p=1}^{nPt}$ and weights $\{\omega_p\}_{p=1}^{nPt}$ produces the algebraic formula

$$A_{lr}^s \approx \sum_{p=1}^{nPt} \sigma_l \sigma_r \Big(T(\widehat{\mathcal{L}}\widehat{\phi}_l)(\widehat{x}_p)\Big)\Big(T(\widehat{\mathcal{L}}\widehat{\phi}_r)(\widehat{x}_p)\Big)J(\widehat{x}_p)\omega_p. \tag{9}$$

This formula represents the contraction of several multi-indexed scalar quantities along the dimension of the cubature point, i.e., it is ideally suited for the MDArray philosophy of Intrepid. The Basis, Cubature, CellTools, and FunctionSpaceTools classes in Section 4 contain methods to compute MDArrays corresponding to the multi-indexed scalars in (9). Figure 7 shows the generic Intrepid assembly pattern using these methods. An important trait of this pattern is its invariance with respect to cell topology, cubature order, or basis function type.

For a concrete example, consider building an $H(\text{grad})$ stiffness matrix from a nodal element with basis $\{N_i\}$. The element contributions from κ_s to this operator are

$$A_{lr}^s = \int_{\kappa_s} \nabla N_l(x) \cdot \nabla N_r(x) dx\,, \tag{10}$$

and the transformation T is the transpose inverse Jacobian DF^{-T}. As a result (9) specializes to

$$A_{lr}^s \approx \sum_{p=1}^{nPt} \Big(DF^{-T}\widehat{\nabla}\widehat{N}_l(\widehat{x}_p)\Big)\Big(DF^{-T}\widehat{\nabla}\widehat{N}_r(\widehat{x}_p)\Big)J(\widehat{x}_p)\omega_p. \tag{11}$$

Figure 8 shows listing of a code that implements (11) for tetrahedral elements using the lowest-order (piecewise linear) nodal element. The sample code uses FieldContainer objects as the default MDArray type. The assembly process requires the following key steps:

1. Choose cell topology for reference cell $\widehat{\kappa}$;
2. Define integration rule using cubature factory and get cubature points $(\widehat{x}_p)$ and weights (ω_p);
3. Select basis function and evaluate basis function gradients at cubature points, $\widehat{\nabla}\widehat{N}_l(\widehat{x}_p)$;

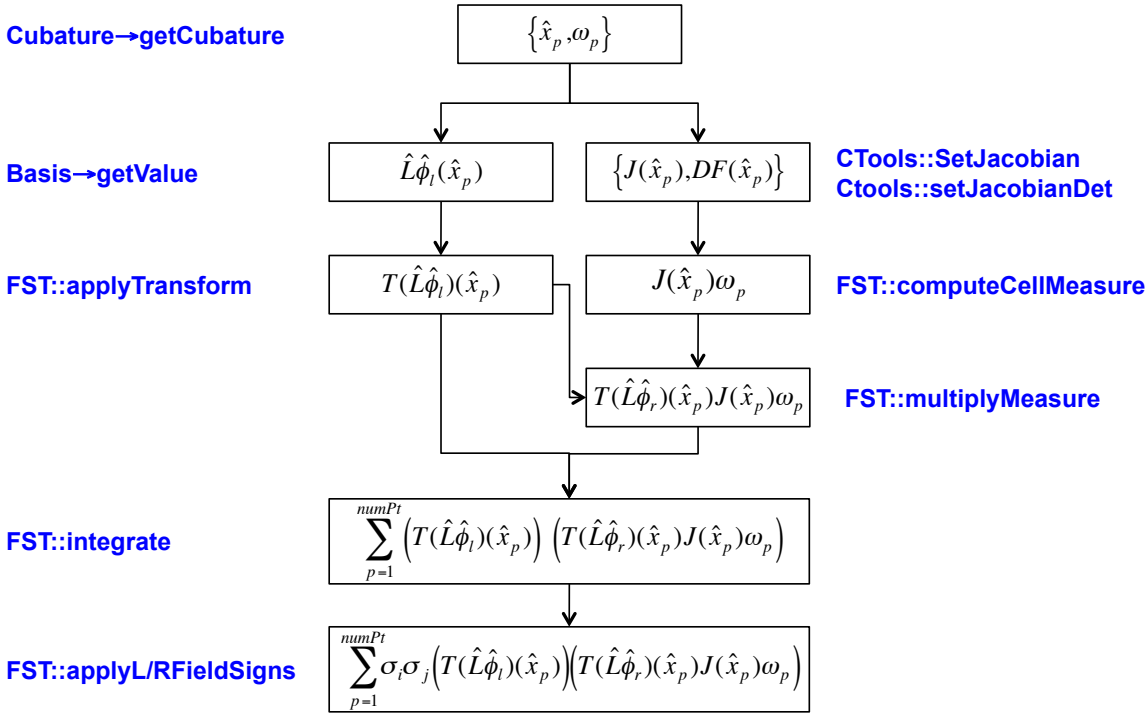


Figure 7: Intrepid assembly pattern for second order elliptic operators.

4. Calculate Jacobian (DF), its inverse (DF^{-1}) and determinant (J);
5. Transform basis function gradients to physical space, $DF^{-T}\widehat{\nabla}\widehat{N}_i(\widehat{x}_p)$;
6. Calculate cell measure, $J(\widehat{x}_p)\omega_p$;
7. Multiply transformed basis function gradients by cell measure, $DF^{-T}\widehat{\nabla}\widehat{N}_r(\widehat{x}_p)J(\widehat{x}_p)\omega_p$;
8. Integrate quantities from step 5 and step 7 to obtain the cell stiffness matrix in Equation 11.

The above operations are performed on a batch or workset of physical cells and the result is a set of operators defined on each cell in the workset. The code in Figure 8 references the multidimensional array (worksetCoords) of vertex coordinates on physical cells, which is assumed to have been filled externally. Example code that creates a sample cell workset containing a single tetrahedral cell is shown in Figure 9. In most practical applications, the physical cell coordinates will be provided by the client code. Once the individual cell operators are built they may be scattered to the appropriate locations in a global operator matrix. Note that the client code will also coordinate the scattering operation.

To illustrate the flexibility of the operator assembly framework, consider changing the example in Figure 8 to the assembly of an $H(\text{grad})$ stiffness matrix on a set of hexahedral cells using the lowest-order (trilinear) nodal basis. The assembly on a different cell topology only requires changes in step 1 and step 2. In particular the definition of the reference cell topology on line 2 must be changed to

```
2  shards::CellTopology cellType(shards::getCellTopologyData<shards::Hexahedron<8> >());
```

and the definition of the first order $H(\text{grad})$ basis function on line 16 must be changed to

```
16 Intrepid::Basis_HGRAD_HEX_C1_FEM<double,Intrepid::FieldContainer<double> > HGradBasis;
```

The code that defines the cubature rule in step 2 may be reused since the cubature factory automatically selects the correct cubature rule based on the cell topology and degree.

```
// Step 1. Define cell topology and get dimensions
 shards::CellTopology cellType(shards::getCellTopologyData<shards::Tetrahedron<4> >());
 int numNodesPerCell = cellType.getNodeCount();
 int spaceDim = cellType.getDimension();

// Step 2. Select numerical integration rule and get points and weights on reference cell
 Intrepid::DefaultCubatureFactory<double> cubFactory;
 int cubDegree = 2;
 Teuchos::RCP<Intrepid::Cubature<double> > cellCub = cubFactory.create(cellType, cubDegree);
 int numCubPoints = cellCub->getNumPoints();
 Intrepid::FieldContainer<double> cubPoints(numCubPoints, spaceDim);
 Intrepid::FieldContainer<double> cubWeights(numCubPoints);
 cellCub->getCubature(cubPoints, cubWeights);

// Step 3. Select basis and evaluate gradients at cubature points
 Intrepid::Basis_HGRAD_TET_C1_FEM<double,Intrepid::FieldContainer<double> > HGradBasis;
 int numFields = HGradBasis.getCardinality();
 Intrepid::FieldContainer<double> basisGrads(numFields,numCubPoints, spaceDim);
 HGradBasis.getValues(basisGrads, cubPoints,Intrepid::OPERATOR_GRAD);

// Step 4. Calculate Jacobian, inverse, and determinant for cell workset
 Intrepid::FieldContainer<double> worksetJacobian(worksetSize, numCubPoints, spaceDim,
                                                                                spaceDim);
 Intrepid::FieldContainer<double> worksetJacobInv(worksetSize, numCubPoints, spaceDim,
                                                                                spaceDim);
 Intrepid::FieldContainer<double> worksetJacobDet(worksetSize,numCubPoints);
 Intrepid::CellTools<double>::setJacobian(worksetJacobian, cubPoints, worksetCoords,
                                                                          cellType);
 Intrepid::CellTools<double>::setJacobianInv(worksetJacobInv, worksetJacobian);
 Intrepid::CellTools<double>::setJacobianDet(worksetJacobDet, worksetJacobian);

// Step 5. Combine Jacobian determinant and cubature weight to get cell measure
 Intrepid::FieldContainer<double> worksetCubWeights(worksetSize, numCubPoints);
 Intrepid::FunctionSpaceTools::computeCellMeasure<double>(worksetCubWeights,
                                                 worksetJacobDet, cubWeights);

// Step 6. Transform basis gradients to physical frame
 Intrepid::FieldContainer<double> worksetBasisGrads(worksetSize, numFields,
                                                     numCubPoints, spaceDim);
 Intrepid::FunctionSpaceTools::HGRADtransformGRAD<double>(worksetBasisGrads,
                                                 worksetJacobInv, basisGrads);

// Step 7. Multiply transformed basis gradients by cell measure
 Intrepid::FieldContainer<double> worksetBasisGradsWeighted(worksetSize, numFields,
                                                                 numCubPoints, spaceDim);
 Intrepid::FunctionSpaceTools::multiplyMeasure<double>(worksetBasisGradsWeighted,
                                             worksetCubWeights, worksetBasisGrads);

// Step 8. Integrate to get stiffness matrices for workset cells
 Intrepid::FieldContainer<double> worksetStiffMatrix(worksetSize,numFields, numFields);
 Intrepid::FunctionSpaceTools::integrate<double>(worksetStiffMatrix,
                    worksetBasisGradsWeighted, worksetBasisGrads, Intrepid::COMP_CPP);
```

Figure 8: Example code listing for the assembly of an $H(\text{grad})$ stiffness matrix on tetrahedral cells using linear basis functions.

```
// Define a workset of physical cells
 int worksetSize = 1;
 Intrepid::FieldContainer<double> worksetCoords(worksetSize, numNodesPerCell, spaceDim);
 worksetCoords(0,0,0) = 0.0; worksetCoords(0,0,1) = 0.0; worksetCoords(0,0,2) = 0.0;
 worksetCoords(0,1,0) = 0.5; worksetCoords(0,1,1) = 0.0; worksetCoords(0,1,2) = 0.0;
 worksetCoords(0,2,0) = 0.0; worksetCoords(0,2,1) = 0.5; worksetCoords(0,2,2) = 0.0;
 worksetCoords(0,3,0) = 0.0; worksetCoords(0,3,1) = 0.0; worksetCoords(0,3,2) = 0.5;
```

Figure 9: Example code listing to define a workset with a single tetrahedral cell.

For another example consider changing the order of the basis function, which is also straightforward using this framework. If a second order nodal basis function is desired in place of a first order basis function the only changes required are in step 2 where a higher degree cubature rule must be chosen and in step 3 where the basis function is selected. In particular line 8 should be changed to

```
8  int cubDegree = 4;
```

and line 16 should be changed to

```
16  Intrepid::Basis_HGRAD_TET_C2_FEM<double,Intrepid::FieldContainer<double> > HGradBasis;
```

This flexibility in the design of Intrepid allows users to easily mix and match different basis functions, cell topologies, and cubature rules without rewriting significant parts of their code.

5.2 Building a Finite Element Functional

Equation residuals and right hand sides in PDE discretizations are examples of *discrete functionals*. Finite element methods assemble such functionals from local element contributions. The computation of these contributions is the second example of a generic computational pattern in Intrepid. The element contribution from κ_s to a finite element functional has the general form

$$f_i^s = \int_{\kappa_s} f(x)\mathcal{L}\phi_i(x)dx, \tag{12}$$

where $f(x)$ is a given scalar or vector function. Assuming the basis, pullback, and cubature notation from the previous section, the algebraic formula

$$f_i^s \approx \sum_{p=1}^{nPt} f(F(\widehat{x}_p))\sigma_i\big(T(\widehat{\mathcal{L}\phi}_i)(\widehat{x}_p)\big)J(\widehat{x}_p)\omega_p. \tag{13}$$

gives the approximation of the finite element functional (12).

For a concrete example consider the task of computing the finite element functional corresponding to the right hand side in the weak Galerkin formulation of the Poisson equation $-\nabla\cdot\nabla u = f$. A finite element discretization using $H(\text{grad})$ basis functions will lead to a global linear system $A_{kl}u_l = f_k$ where u_l are the nodal coefficients, A_{kl} are the components of the stiffness matrix, and f_k are the components of the right-hand side vector. The element contributions from element κ_s to the right-hand side vector are generated from the functional

$$f_i^s \approx \sum_{p=1}^{nPt} f(F(\widehat{x}_p))\widehat{N}_i(\widehat{x}_p)J(\widehat{x}_p)\omega_p. \tag{14}$$

The assembly of this functional using Intrepid is similar to that of the finite element operator and includes the following steps:

1. Choose cell topology for reference cell $\widehat{\kappa}$;
2. Define integration rule using cubature factory and get cubature points $(\widehat{x}_p)$ and weights (ω_p);
3. Select basis function and evaluate basis function values at cubature points, $\widehat{N}_i(\widehat{x}_p)$;
4. Calculate Jacobian (DF) and determinant (J);
5. Transform basis function values to physical space, $\widehat{N}_i(\widehat{x}_p)$;
6. Calculate cell measure, $J(\widehat{x}_p)\omega_p$;

7. Multiply transformed basis function gradients by cell measure, $\widehat{N}_i(\widehat{x}_p)J(\widehat{x}_p)\omega_p$;

8. Map cubature points to physical space, $x_p = F(\widehat{x}_p)$;

9. Get source function at cubature points, $f(x_p)$;

10. Integrate source function against basis function to obtain final cell contributions to right-hand side vector in Equation 14.

These steps are illustrated in the code listing in Figure 10. This code includes an additional piece of Intrepid functionality in Step 8 where the mapToPhysicalFrame() method of the CellTools class is used to get cubature point coordinates in a physical cell. Note that the code assumes that a method called getRHSData() has been defined by the user to get the source function value at the cubature points. As in the previous example, any changes in the basis and cell topology are easily implemented without changing the overall flow of the assembly procedure.

5.3 Evaluation of Finite Element Fields

Given a set of basis functions $\{\phi_i\}_{i=1}^n$ on a cell, a finite element function on this cell is a linear combination of the basis functions:

$$\mathbf{u}^h = \sum_{i=1}^{n} u_i \phi_i(x). \tag{15}$$

The meaning of the coefficients $\{u_i\}_{i=1}^n$ depends on the basis. For nodal $H(\text{grad})$ basis functions u_i correspond to the values of the function at the element vertices. For $H(\text{curl})$ and $H(\text{div})$ bases the coefficients can be edge circulations and face fluxes, respectively.

If the basis function is defined on a reference cell as was assumed in the previous examples then the finite element field may be written as

$$\mathbf{u}^h(x) = \sum_{i=1}^{n} u_i \sigma_i \Phi^*(\widehat{\phi}_i)(x) = \sum_{i=1}^{n} u_i \sigma_i T(\widehat{\phi}_i) \circ F^{-1}(x)\,. \tag{16}$$

Consider an example of evaluating a finite element field using an $H(\text{div})$ basis $\{\varphi_i\}$. In this case $\Phi^*(\widehat{\varphi}) = J^{-1} DF \widehat{\varphi} \circ F^{-1}$ and the finite element field has the form

$$\mathbf{u}^h(x) = \sum_{i=1}^{n} u_i \sigma_i J^{-1} DF \widehat{\varphi}_i \circ F^{-1}(x). \tag{17}$$

If we would like to evaluate the divergence of the field as well, the expression we need to compute is (see Section 4.6.1)

$$\nabla \cdot \mathbf{u}^h(x) = \sum_{i=1}^{n} u_i \sigma_i J^{-1} \widehat{\nabla} \cdot \widehat{\varphi}_i \circ F^{-1}(x). \tag{18}$$

Note that the $H(\text{div})$ basis functions are vectors and the field signs (σ_i) must be included to correctly transform between the reference cell and the physical cell. In summary, evaluation of finite element fields involves the following steps:

1. Choose cell topology for reference cell $\widehat{\kappa}$;

2. Define evaluation points $\{x_p\}$ in cell κ;

3. Map the evaluation points to reference element, $\widehat{x}_p = F^{-1}(x_p)$;

4. Select basis function and evaluate basis function values at evaluation points, $\widehat{\varphi}_i(\widehat{x}_p)$;

5. Calculate Jacobian (DF) and determinant (J);

```
// Step 1. Define cell topology and get dimensions
 shards::CellTopology cellType(shards::getCellTopologyData<shards::Tetrahedron<4> >());
 int numNodesPerCell = cellType.getNodeCount();
 int spaceDim = cellType.getDimension();

// Step 2. Select numerical integration rule and get points and weights on reference cell
 Intrepid::DefaultCubatureFactory<double> cubFactory;
 int cubDegree = 2;
 Teuchos::RCP<Intrepid::Cubature<double> > cellCub = cubFactory.create(cellType, cubDegree);
 int numCubPoints = cellCub->getNumPoints();
 Intrepid::FieldContainer<double> cubPoints(numCubPoints, spaceDim);
 Intrepid::FieldContainer<double> cubWeights(numCubPoints);
 cellCub->getCubature(cubPoints, cubWeights);

// Step 3. Select basis and evaluate values at cubature points
 Intrepid::Basis_HGRAD_TET_C1_FEM<double,Intrepid::FieldContainer<double> > HGradBasis;
 int numFields = HGradBasis.getCardinality();
 Intrepid::FieldContainer<double> basisValues(numFields,numCubPoints);
 HGradBasis.getValues(basisValues, cubPoints,Intrepid::OPERATOR_VALUE);

// Step 4. Calculate Jacobian and determinant for cell workset
 Intrepid::FieldContainer<double> worksetJacobian(worksetSize, numCubPoints, spaceDim,
                                                                                spaceDim);
 Intrepid::FieldContainer<double> worksetJacobDet(worksetSize,numCubPoints);
 Intrepid::CellTools<double>::setJacobian(worksetJacobian, cubPoints, worksetCoords,
                                                                        cellType);
 Intrepid::CellTools<double>::setJacobianDet(worksetJacobDet, worksetJacobian);

// Step 5. Combine Jacobian determinant and cubature weight to get cell measure,
 Intrepid::FieldContainer<double> worksetCubWeights(worksetSize, numCubPoints);
 Intrepid::FunctionSpaceTools::computeCellMeasure<double>(worksetCubWeights,
                                                 worksetJacobDet, cubWeights);

// Step 6. Transform basis values to physical frame
 Intrepid::FieldContainer<double> worksetBasisValues(worksetSize, numFields,
                                                       numCubPoints);
 Intrepid::FunctionSpaceTools::HGRADtransformVALUE<double>(worksetBasisValues,
                                                              basisValues);

// Step 7. Multiply transformed basis values by cell measure
 Intrepid::FieldContainer<double> worksetBasisValuesWeighted(worksetSize, numFields,
                                                                        numCubPoints);
 Intrepid::FunctionSpaceTools::multiplyMeasure<double>(worksetBasisValuesWeighted,
                                          worksetCubWeights, worksetBasisValues);

// Step 8. Map cubature points to physical space
 Intrepid::FieldContainer<double> worksetCubPoints(worksetSize, numCubPoints,
                                                                     spaceDim);
 Intrepid::CellTools<double>::mapToPhysicalFrame(worksetCubPoints, cubPoints,
                                                   worksetCoords, cellType);

// Step 9. Get source term (RHS) data at cubature points
 Intrepid::FieldContainer<double> worksetRHSData(worksetSize, numCubPoints);
 getRHSData(worksetRHSData, worksetCubPoints);

// Step 10. Integrate to get right hand side vector for workset cells
 Intrepid::FieldContainer<double> worksetRHSVector(worksetSize, numFields);
 Intrepid::FunctionSpaceTools::integrate<double>(worksetRHSVector,
              worksetRHSData, worksetBasisValuesWeighted, Intrepid::COMP_CPP);
```

Figure 10: Example code listing for the assembly of an $H(\text{grad})$ functional on tetrahedral cells using linear basis functions.

6. Transform basis function values to physical space, $J^{-1}DF\widehat{\varphi}_i(\widehat{x}_p)$;
7. Apply field signs to transformed basic function values, $\sigma_i J^{-1}DF\widehat{\varphi}_i(\widehat{x}_p)$;
8. Evaluate finite element field $\mathbf{u}^h = \sum_{i=1}^{n} u_i\sigma_i J^{-1}DF\widehat{\varphi}_i(\widehat{x})$.

Figure 11 shows an example code to compute the values of a finite element vector field and its

```
1  // Step 1. Define cell topology and get dimensions
2   shards::CellTopology cellType(shards::getCellTopologyData<shards::Hexahedron<8> >());
3   int numNodesPerCell = cellType.getNodeCount();
4   int numFacesPerCell = cellType.getSideCount();
5   int spaceDim = cellType.getDimension();
6
7  // Step 2. Define evaluation points (here use cell centers)
8   int numEvalPoints = 1;
9   Intrepid::FieldContainer<double> evalPoints(numEvalPoints, spaceDim);
10  evalPoints(0,0)=0.0; evalPoints(0,1)=0.0; evalPoints(0,2)=0.0;
11
12 // Step 3. Select basis and evaluate values and divergence at evaluation points
13  Intrepid::Basis_HDIV_HEX_I1_FEM<double,Intrepid::FieldContainer<double> > HDivBasis;
14  int numFields = HDivBasis.getCardinality();
15  Intrepid::FieldContainer<double> basisValues(numFields, numEvalPoints, spaceDim);
16  Intrepid::FieldContainer<double> basisDivs(numFields, numEvalPoints);
17  HDivBasis.getValues(basisValues, evalPoints, Intrepid::OPERATOR_VALUE);
18  HDivBasis.getValues(basisDivs, evalPoints, Intrepid::OPERATOR_DIV);
19
20 // Step 4. Calculate Jacobian and determinant for cell workset
21  Intrepid::FieldContainer<double> worksetJacobian(worksetSize, numEvalPoints, spaceDim,
22                                                                                  spaceDim);
23  Intrepid::FieldContainer<double> worksetJacobDet(worksetSize,numEvalPoints);
24  Intrepid::CellTools<double>::setJacobian(worksetJacobian, evalPoints, worksetCoords,
25                                                                          cellType);
26  Intrepid::CellTools<double>::setJacobianDet(worksetJacobDet, worksetJacobian);
27
28 // Step 5. Transform basis values and divergence to physical frame
29  Intrepid::FieldContainer<double> worksetBasisValues(worksetSize, numFields,
30                                                  numEvalPoints, spaceDim);
31  Intrepid::FieldContainer<double> worksetBasisDivs(worksetSize, numFields,
32                                                  numEvalPoints);
33  Intrepid::FunctionSpaceTools::HDIVtransformVALUE<double>(worksetBasisValues,
34                                 worksetJacobian, worksetJacobDet, basisValues);
35  Intrepid::FunctionSpaceTools::HDIVtransformDIV<double>(worksetBasisDivs,
36                                          worksetJacobDet, basisDivs);
37
38 // Step 6. Apply field signs
39  Intrepid::FunctionSpaceTools::applyFieldSigns<double>(worksetBasisValues,
40                                                          worksetFaceSigns);
41  Intrepid::FunctionSpaceTools::applyFieldSigns<double>(worksetBasisDivs,
42                                                          worksetFaceSigns);
43
44 // Step 7. Evaluate finite element field and its divergence at evaluation points
45  Intrepid::FieldContainer<double> worksetu(worksetSize, numEvalPoints, spaceDim);
46  Intrepid::FieldContainer<double> worksetdivu(worksetSize, numEvalPoints);
47  Intrepid::FunctionSpaceTools::evaluate<double>(worksetu, uCoefficients,
48                                                 worksetBasisValues);
49  Intrepid::FunctionSpaceTools::evaluate<double>(worksetdivu, uCoefficients,
50                                                   worksetBasisDivs);
```

Figure 11: Example code listing for the evaluation of a finite element field and its divergence using lowest-order $H(\text{div})$ basis functions.

divergence using $H(\text{div})$ basis functions. The code listing refers to the arrays uCoefficients and worksetFaceSigns that are both sized by (worksetSize, numFacesPerCell). These arrays contain the coefficients of the basis function used to generate the finite element field and the signs (± 1) that describe the orientation of the physical cell face compared with the orientation of the reference cell face. If the faces have the same orientation the sign is $+1$.

In this example the single evaluation point on each hexahedral cell is its center $x_C = F(0,0,0)$, i.e., the image of the center $\widehat{x}_C = (0,0,0)$ of the reference Hexahedron $[-1,1]^3$. Because the pre-image of x_C is known, we can skip Step 3 and directly define the evaluation point on the reference element; see line 10 in Figure 11. In general, the preimages $\widehat{x}_p$ of the evaluation points are not know in advance. The method mapToReferenceFrame() of the CellTools class implements the action of F^{-1} for all standard finite element topologies defined in Shards.

6 Additional Capabilities and Future Directions

The design of Intrepid allows for unprecedented flexibility and extensibility of its core functionality by new cell topologies, integration rules, basis definitions, and discretization methods. In this section we briefly review some recent additions to Intrepid, which illustrate this trait.

6.1 Polygonal Elements

The solution of PDEs on polygonal cells is attractive in multiple contexts. Such cells provide greater flexibility in the meshing of complex geometries and can serve as "glue" between conventional elements. Polygonal cells are attractive for material design [13], multi-material calculations [18], and arbitrary Lagrangian-Eulerian methods for meshes whose connectivity may change [20, 19].

Currently Intrepid supports $H(\text{grad})$ finite elements on arbitrary convex polygons [32]. The class Basis_HGRAD_POLY_C1_FEM implements polygonal basis functions ϕ_i with the following properties:

- Partition of unity: $\sum_{i=1}^{n} \phi_i(\mathbf{x}) = 1$;
- Interpolatory basis: for any vertex $\mathbf{v}_j$ of a polygon κ: $\phi_i(\mathbf{v}_j) = \delta_{ij}$;
- Linear completeness: $\sum_{i=1}^{n} \phi_i(\mathbf{x})\mathbf{x}_i = \mathbf{x}$;
- The basis functions $\phi_i \in C^\infty(\kappa)$ and their restriction to $\partial\kappa$ are C^0, piecewise linear functions.

The implementation of this class follows the length and area metric approach of [29]. Given a convex polygon κ let a_i be the area of triangle formed by vertex $\mathbf{v}_i$ and its adjacent two vertices and $r_{jk}(\mathbf{x})$ be the area of triangle formed by vertices $\mathbf{v}_j$ and $\mathbf{v}_k$ with point $\mathbf{x}$. The rational function

$$\phi_i(x) = \frac{w_i(\mathbf{x})}{\sum_{j=1}^{n} w_j(\mathbf{x})},$$

where w_i are the *weight* functions

$$w_i(\mathbf{x}) = a_i \prod_{jk \neq i} r_{jk}(\mathbf{x}),$$

defines the basis function associated with vertex $\mathbf{v}_i$ of the polygon κ. The evaluation of these basis functions requires the vertex coordinates of the polygonal cell. For this reason, the class Basis_HGRAD_POLY_C1_FEM uses the second field evaluation interface of the Basis class, in which the vertex coordinates of the cell are one of the input arguments; see Section 4.

```
template<class Scalar, class ArrayScalar>
class Basis_HGRAD_POLY_C1_FEM : public Basis<Scalar, ArrayScalar> {
  ...
public:
  ...
  Basis_HGRAD_POLY_C1_FEM(const shards::CellTopology& cellTopology);

  void getValues(ArrayScalar& outputValues,
                 const ArrayScalar& inputPoints,
                 const ArrayScalar& cellVertices,
                 const EOperator operatorType);
  ...
};
```

The computation of OPERATOR_GRAD for the basis functions uses Sacado for automatic differentiation. Figure 12 shows typical polygonal basis functions and their gradient fields.

The class CubaturePolygon generates integration points on polygonal cells in *physical coordinates*. Consequently, this class requires access to the vertex coordinates of the polygon. To this end, the CubaturePolygon constructor has an additional MDArray argument to pass the cell vertices.

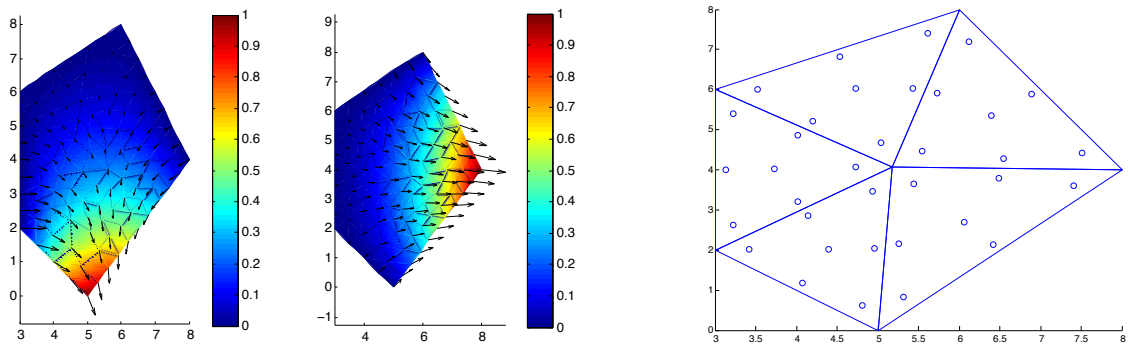


Figure 12: The two left plots show representative Intrepid polygonal basis functions and their gradient fields. The right plot shows cubature points generated by the CubaturePolygon class.

```
template<class Scalar, class ArrayPoint=FieldContainer<Scalar>,
         class ArrayWeight=ArrayPoint>
class CubaturePolygon : public Intrepid::Cubature<Scalar,ArrayPoint,ArrayWeight> {
  ...
public:
  ...
  CubaturePolygon(const shards::CellTopology& cellTopology,
                  const ArrayPoint& cellVertices,
                  int degree);

  void getCubature(ArrayPoint& cubPoints,
                   ArrayWeight& cubWeights) const;
  ...
};
```

The getCubature() method first triangulates the polygon using its centroid. Subsequently, it uses a CubatureDirectTriDefault object to generate integration points on a reference triangle. Finally, these points are mapped individually to the physical triangles forming the triangulation of the polygonal cell; see Figure 12.

6.2 Adaptive Stochastic Collocation

The solution of PDEs with random input data is critical to the quantification of uncertainty in national security applications, energy and climate research, and engineering design. The challenge in solving such PDEs is in the potentially large number of random fields that enter the equations. For example, a deterministic advection-diffusion equation in two spatial dimensions may be extended to the stochastic PDE

$$-\epsilon(\gamma,x)\,\Delta u(\gamma,x) + \mathbf{V}(\gamma,x)\cdot\nabla u(\gamma,x) = z(\gamma,x) \qquad \gamma\in\Gamma,\ x\in D \tag{19a}$$

$$u(\gamma,x) = d(\gamma,x) \qquad \gamma\in\Gamma,\ x\in\partial D_D \tag{19b}$$

$$\nabla u(\gamma,x)\cdot n = g(\gamma,x) \qquad \gamma\in\Gamma,\ x\in\partial D_N\,, \tag{19c}$$

where the PDE solution $u(\gamma,x)$ and the input fields $\epsilon(\gamma,x)$ (diffusion), $\mathbf{V}(\gamma,x)$ (advection), $z(\gamma,x)$ (source field), $d(\gamma,x)$ (Dirichlet boundary field) and $g(\gamma,x)$ (Neumann boundary field) are random fields that depend on the points x in a physical domain $D\subset\mathbb{R}^2$ and the points γ in a probability space $\Gamma\subset\mathbb{R}^M$; here ∂D_D denotes the Dirichlet boundary, ∂D_N denotes the Neumann boundary. In this particular case, if one random variable is used to represent each input field, the total dimension of the problem is seven (two spatial and five stochastic variables). As the stochastic dimension M increases, the discretization and subsequent solution of the PDE quickly become computationally intractable — unless special structure is (a) present and (b) can be exploited computationally.

Intrepid implements one such structure-exploiting discretization of the stochastic variables, called *stochastic collocation*. Unlike Monte-Carlo methods that are rooted in probability theory and often exhibit slow convergence for problems of type (19), stochastic collocation is based on approximation theory and exploits PDE regularity to accelerate convergence [37, 35, 4]. Stochastic collocation interpolates the random PDE variables on a set of integration points in high-dimensional spaces. For this reason, Intrepid's Cubature classes are its proper building blocks.

To support stochastic collocation, Intrepid adds a variety of one-dimensional cubature rules, based on John Burkardt's work [9], provided by the CubatureLineSorted class. The available line cubatures, selected via the rule parameter, include Gauss-Chebyshev, Clenshaw-Curtis, Fejer, Gauss-Legendre, Gauss-Patterson, composite trapezoidal, Gauss-Hermite, Hermite-Genz-Keister and Gauss-Laguerre rules. A constructor for the CubatureLineSorted class is given below.

```
  CubatureLineSorted(int degree = 0, EIntrepidBurkardt rule = BURK_CLENSHAWCURTIS,
                     bool isNormalized = false);
```

Tensor-products of the one-dimensional cubatures can be built using the CubatureTensorSorted class.

To make stochastic collocation computationally tractable in high dimensions, Intrepid provides an implementation of classic *sparse grids* [31], which generate sets of integration points that are small in size yet feature good polynomial accuracy. In addition, a *dimension-adaptive framework* [15] for stochastic collocation is available.

Classic sparse grids are built using the buildSparseGrid() method of the AdaptiveSparseGrid class.

```
  AdaptiveSparseGrid<Scalar, Vector>::buildSparseGrid(
    CubatureTensorSorted<Scalar> & output, // Cubature Points/Weights
    int dimension,                         // Dimension of Stochastic Space
    int maxlevel,                          // Maximum Sparse Grid Level
    std::vector<EIntrepidBurkardt> rule1D, // 1D Cubature Rules
    std::vector<EIntrepidGrowth> growth1D, // 1D Growth Rules
    bool isNormalized);                    // Normalize Weights?
```

A variety of *growth rules* are supported and include slow linear, moderate linear, slow exponential, moderate exponential and full exponential growth [9], chosen via the growth1D parameter.

Dimension-adaptive sparse grids can be realized using one of several refine_grid() methods.

```
template<class Scalar, class UserVector>
class AdaptiveSparseGrid {
public:
  ...
  static Scalar refine_grid(
    typename std::multimap<Scalar,std::vector<int> >
      & activeIndex,                                  // active (new) grid indices
    std::set<std::vector<int> > & oldIndex,           // previous set of indices
    UserVector & integralValue,                       // vector of integral values
    CubatureTensorSorted<Scalar> & cubRule,           // sparse grid points and weights
    Scalar globalErrorIndicator,                      // error indicator for adaptivity
    AdaptiveSparseGridInterface<Scalar,UserVector>
      & problem_data                                  // user-implemented ASG interface
  );
  ...
};
```

The input parameter problem_data is a user-supplied object that implements the abstract class AdaptiveSparseGridInterface. This object defines the methods for the evaluation of, e.g., stochastic integrands and error indicators. Other methods, such as eval_cubature() come with default implementations (in this case a simple summation of integrand values multiplied by the weights), but can be overloaded with user-defined implementations.

```
template<class Scalar, class UserVector>
class AdaptiveSparseGridInterface {
```

```
  ...
public:
  virtual void eval_integrand(UserVector & output,
                              std::vector<Scalar> & input) = 0;

  virtual void eval_cubature(UserVector & output,
                             CubatureTensorSorted<Scalar> & cubRule);

  virtual Scalar error_indicator(UserVector & input) = 0;

  virtual bool max_level(std::vector<int> index);
  ...
```

Given these building blocks, an adaptive sparse grid algorithm may be implemented as follows.

```
double adaptSG(Vector<double> & value,
   std::multimap<double,std::vector<int> > & activeIndex,
   std::set<std::vector<int> > & oldIndex,
   AdaptiveSparseGridInterface<double,StdVector<double> > & problem_data,
   CubatureTensorSorted<double> & cubRule,
   double TOL)
{
  // Construct container for adapted rule
  int dimension = problem_data.getDimension();
  std::vector<int> index(dimension,1);

  // Initialize global error indicator
  double eta = 1.0;

  // Initialize active index set
  activeIndex.insert(std::pair<double,std::vector<int>>(eta,index));

  // Perform adaptation
  while (eta > TOL) {
    eta = AdaptiveSparseGrid<double,Vector<double>>::refine_grid(
       activeIndex, oldIndex, value, cubRule, eta, problem_data);
  }
  cubRule.normalize();
  return eta;
}
```

Intrepid's adaptive stochastic collocation capability has been used to solve optimal design and inverse problems governed by PDEs with random inputs. Consider the stochastic PDE (19) with $z(\gamma, x) = z(x)$, $\epsilon(\gamma, x) = \epsilon$ and $g(\gamma, x) = 0$. Let u_d denote a set of given discrete measurements of the state $u(\gamma, x)$ and let $\{x^1, \ldots, x^N\}$ denote the locations of the measurements. We solve the source inversion problem

$$\min_{u,\,z} \frac{1}{2} \sum_{i=1}^{N} E\Big[|u(\cdot, x^i) - u_d(x^i)|^2\Big] + \frac{\alpha}{2}\|z\|^2$$

subject to the PDE (19). We note that in this setting z plays the role of a (deterministic) control variable whose optimal value reveals the locations and amplitudes of sources that generate the measurements u_d. The results of source inversion for a problem with three stochastic dimensions (random Dirichlet condition, advection amplitude and advection angle) are shown in Figure 13.

6.3 Control Volume Finite Element Method

Consider the scalar advection-diffusion equation

$$\frac{\partial u}{\partial t} - \nabla \cdot (\mu \nabla u - \mathbf{b} u) = f \quad \text{in } \Omega \quad \text{and} \quad u = 0 \quad \text{on } \partial\Omega\,, \tag{20}$$

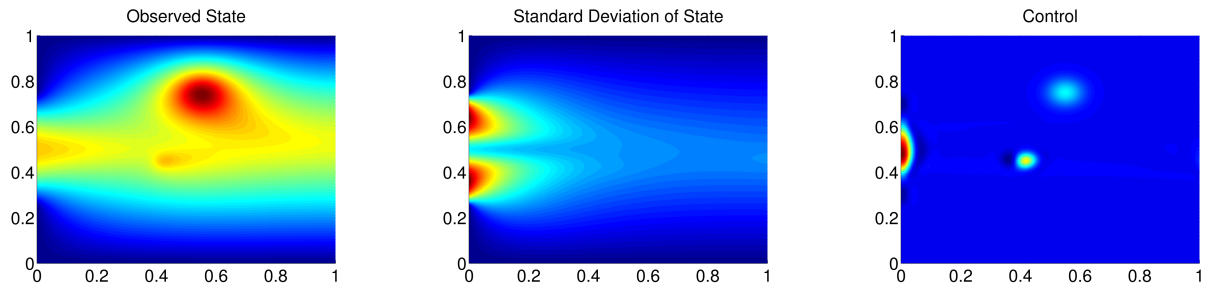


Figure 13: Left to right: observed state (measurements); standard deviation of optimal state; optimal control (sources), i.e., the result of source inversion. In this example the stochastic variables govern (1) the Dirichlet boundary condition (posed on the left boundary of the domain), (2) the amplitude of the advection field and (3) the angle of the advection field with respect to the outward facing normal on the left boundary. The random variables are distributed uniformly on $[-1, 1]^3$.

where μ is diffusion coefficient and $\mathbf{b}$ is velocity field. The control volume finite element method (CVFEM) for (20) is a hybrid spatial discretization method that combines finite element representation of the discrete solution with definition of the discrete equations through integration of the conservative form of the governing equations on dual volumes C_i, associated with the vertices $\mathbf{v}_i$ of the finite element mesh. Left plot in Figure 14 shows the geometry of a typical control volume on an unstructured quadrilateral grid. Let $\{\phi_i\}$ be an $H(\text{grad})$-conforming basis and assume that

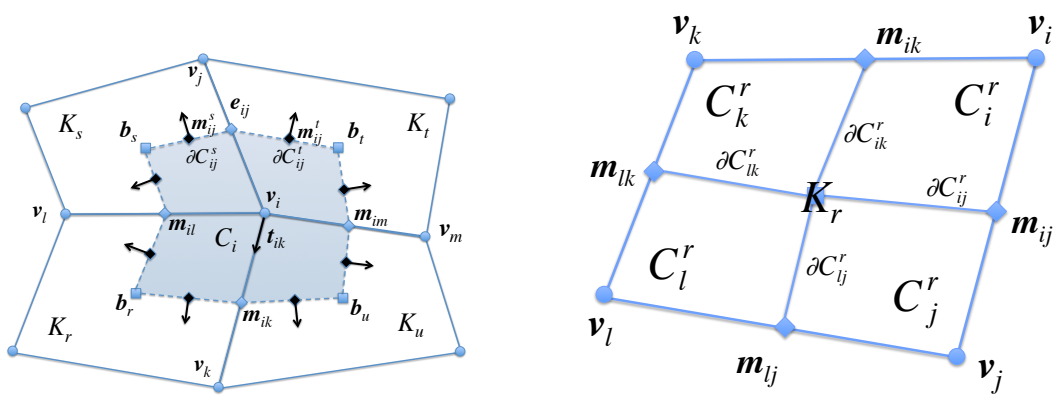


Figure 14: Left plot: unstructured primal grid with typical control volume (C_i) for the CVFEM. Right plot: CVFEM assembly requires computation of volume and surface integrals on subdomains of the finite element defined by its intersection with the control volumes.

(20) has been discretized in time using the implicit Euler scheme with time step Δt. Let u_h^o denote the finite element solution at the old time step. Application of CVFEM to the semidiscrete in time equation yields an algebraic system with matrix and right hand side given by

$$A_{ij} = \int_{C_i} \phi_j dV - \Delta t \int_{\partial \dot{C}_i} (\mu \nabla \phi_j - \mathbf{b}\phi_j) \cdot \mathbf{n} dS \quad \text{and} \quad f_i = \Delta t \int_{C_i} f dS + \int_{C_i} u_h^o dV \,, \tag{21}$$

respectively. Assembly of "operator" and "functional" elements A_{ij} and f_i requires integration over the control volumes and their boundaries. Because each control volume intersects one or more elements, A_{ij} and f_i can be assembled from element contributions A_{ij}^r and f_i^r computed on element κ_r. Assume that κ_r has vertices $\mathbf{v}_i$, $\mathbf{v}_j$, $\mathbf{v}_k$, and $\mathbf{v}_l$; see Figure 14. The control volumes associated with the vertices are C_i, C_j, C_k, and C_l. Their intersections with κ_r yield the element subdomains (sub-control volumes) C_i^r, C_j^r, C_k^r, and C_l^r, respectively. The element contributions from κ_r are

$$A_{ij}^r = \int_{C_i^r} \phi_j dV - \Delta t \int_{\partial \dot{C}_i^r} (\mu \nabla \phi_j - \mathbf{b}\phi_j) \cdot \mathbf{n} dS \quad \text{and} \quad f_i^r = \Delta t \int_{C_i^r} f dS + \int_{C_i^r} u_h^o dV \tag{22}$$

respectively. From (22) we see that computation of the element contributions A^r_{ij} and f^r_i is completely local to κ_r, i.e., it does not require any information about the neighbors of κ_r. This means that the CVFEM assembly naturally fits the standard Intrepid workflow where operations are carried on per cell basis on unrelated cells stored in cell worksets.

CVFEM assembly requires several additional steps that are not present in conventional FEM codes:

1. Find the center $\mathbf{b}_r$ of element κ_r;
2. Compute unit normals to the segments of the sub-control volume boundaries ∂C^r_i, ∂C^r_j, ∂C^r_k, and ∂C^r_l;
3. Compute the measures of sub-control volumes C^r_i, C^r_j, C^r_k, and C^r_l;
4. Compute the measures of the sub-control volume boundaries ∂C^r_i, ∂C^r_j, ∂C^r_k, and ∂C^r_l;
5. Define cubature points on the sub-control volumes and their boundaries.

The methods in CellTools and FunctionSpaceTools classes provide all the necessary functionality to carry out these steps. We have implemented a CVFEM for (20) on quadrilateral grids for which the sub-control volumes C^r_i, C^r_j, C^r_k, and C^r_l are also quadrilaterals. We store their coordinates in FieldContainer objects:

```
// container for coordinates of subcontrol volume nodes
int numNodesPerSubCV = 4;
FieldContainer<double> subCVNodes(numNodesPerElem, numNodesPerSubCV, spaceDim);
```

The computation of, e.g., sub-control volume measures in Step 3 uses the standard Jacobian methods from CellTools:

```
// calculate Jacobian and determinant for each subCV
// (Cell type is set to quad for subCV)
CellTools::setJacobian(subCVJacobian, evalPoint, subCVNodes, cellType);
CellTools::setJacobianDet(subCVJacobDet, subCVJacobian );
.........................................
         subCVArea(ielem,inode) = 4.0 * subCVJacobDet(inode,0);
.........................................
```

The implementation of the remaining steps above uses similarly the standard Intrepid functionality.

6.4 Visualization of Ultra-Large Climate Data Sets

In the field of climate research the existing tools and algorithms are not sufficient to process and visualize the the large amounts of data generated from codes and observations efficiently. Often the final visualization is less computationally expensive than the calculations that must be performed on the data before it can be visualized. This may involve interpolating data onto another grid or computing derived quantities from the data. To improve performance in data analysis and visualization, Intrepid tools are being leveraged for the computational engine in the Parallel Climate Analysis Library (ParCAL) (trac.mcs.anl.gov/projects/parvis). For a given set of function values on a grid, Intrepid contains all the necessary tools to obtain function approximations at points in a cell for use in interpolation and for computing derived quantities.

One example use case is to compute the vorticity of a vector velocity field on a latitude (ϕ) - longitude (λ) grid. In one widely-used analysis tool, NCL (www.ncl.ucar.edu), this computation is done using a spherical harmonic representation. The drawback of this method is that the approximation is global and requires data that cover the entire Earth. In contrast, Intrepid can perform the same computation using local basis functions for a data set that is regional or global.

The method that Intrepid uses to compute the vorticity begins with an approximation of the velocity field over a cell using nodal basis functions (N_i) and nodal velocity values ($\mathbf{v}_i$)

$$\mathbf{v}(\lambda, \phi) \approx \sum_i N_i(\lambda, \phi)\mathbf{v}_i. \tag{23}$$

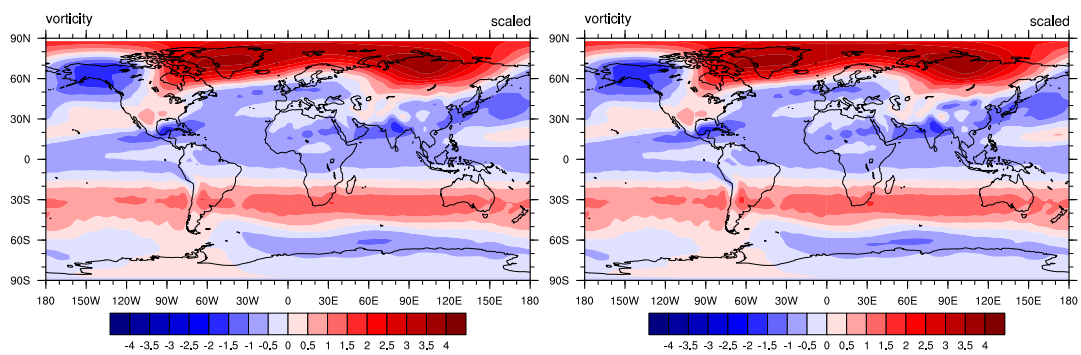
Approximate values of the gradient $\mathbf{v} = (u, v)$, can be determined using the gradients of the basis functions as

$$\nabla u(\lambda, \phi) \approx \sum_i u_i \nabla N_i(\lambda, \phi), \quad \nabla v(\lambda, \phi) \approx \sum_i v_i \nabla N_i(\lambda, \phi). \tag{24}$$

The procedures in steps 3 - 5 of the stiffness matrix assembly discussed in Section 5 are used to obtain the gradients of the shape functions, ∇N_i, and the FunctionSpaceTools::evaluate() method is used to obtain the gradients of the velocity components. Once the gradients are obtained, the vorticity or divergence can be computed by combining partial derivatives and metric terms for vorticity in physical space as shown in equation 25, where r is the radius of the Earth.

$$vorticity = \frac{1}{r\cos\phi}\frac{\partial v}{\partial \lambda} - \frac{1}{r}\frac{\partial u}{\partial \phi} + \frac{u}{r}\tan\phi. \tag{25}$$

This approach is easily parallelizable and generates a vorticity field that is quite similar to that computed with NCL using spherical harmonics as shown in Figure 15.



(a) (b)

Figure 15: Vorticity plotted with NCL and computed using (a) local cell approach with Intrepid and (b) spherical harmonics using the NCL function uv2vrG_wrap.

Acknowledgments

Crucial support for the research in advanced discretization methods, which prompted the inception of Intrepid, has been provided by the ASCR program of DoE's Office of Science. Further research and development of the package was supported in part by the ASCR and the ASC program of the NNSA.

Our work benefited tremendously from formal and informal discussions with Eric Cyr, Roger Pawlowski, Kevin Long, and Jake Ostien, among others. Our summer students M. Keegan, N. Roberts, J. Lai, D. Kouri and C. Beni bravely adopted Intrepid for their research projects and flushed a number of bugs from the package. They also contributed to the further development of the package and successfully applied it to challenging problems such as Discontinuous Petrov-Galerkin methods [27], discontinuous least-squares finite element methods [6] and adaptive stochastic collocation methods, among others.

References

[1] *Intrepid user guide.* http://trilinos.sandia.gov/packages/intrepid/documentation.html, October 2011.

[2] D. N. Arnold, R. S. Falk, and R. Winther, *Finite element exterior calculus, homological techniques, and applications*, Acta Numerica, 15 (2006), pp. 1–155.

[3] A. Aziz, ed., *The Mathematical Foundations of the Finite Element Method with Applications to Partial Differential Equations*, Academic Press, New York, 1972.

[4] I. Babuška, F. Nobile, and R. Tempone, *A Stochastic Collocation Method for Elliptic Partial Differential Equations with Random Input Data*, SIAM Review, 52 (2010), pp. 317–355.

[5] P. Bochev and M. Hyman, *Principles of compatible discretizations*, in Compatible Discretizations, Proceedings of IMA Hot Topics Workshop on Compatible Discretizations, D. N. Arnold, P. Bochev, R. Lehoucq, R. Nicolaides, and M. Shashkov, eds., vol. IMA 142, Springer Verlag, 2006, pp. 89–120.

[6] P. Bochev, J. Lai, and L. Olson, *A locally conservative, discontinuous least-squares finite element method for the stokes equations*, International Journal for Numerical Methods in Fluids, (2011).

[7] A. Bossavit, *Whitney forms: A class of finite elements for three-dimensional computations in electromagnetism*, IEE Proc., 135 (1988), pp. 493–500.

[8] F. Brezzi and M. Fortin, *Mixed and Hybrid Finite Element Methods*, Springer, Berlin, 1991.

[9] J. Burkardt, *1D Quadrature Rules for Sparse Grids*, tech. rep., Interdisciplinary Center for Applied Mathematics and Information Technology Department, Virginia Tech, 2010.

[10] S. S. Cairns, *Introductory topology*, Ronald Press, New York, 1961.

[11] P. Ciarlet, *The Finite Element Method for Elliptic Problems*, SIAM Classics in Applied Mathematics, SIAM, Philadelphia, 2002.

[12] A. Dezin, *Multidimensional Analysis and Discrete Models*, CRC Presss, Boca Raton, FL, 1995.

[13] A. R. Diaz and A. Benard, *Designing materials with prescribed elastic properties using polygonal cells*, International Journal for Numerical Methods in Engineering, 57 (2003), pp. 301–314.

[14] H. Flanders, *Differential Forms with Applications to the Physical Sciences*, Dover, New York, 1989.

[15] T. Gerstner and M. Griebel, *Dimension-adaptive tensor-product quadrature*, Computing, 71 (2003), pp. 65–87.

[16] F. H. Harlow, *The particle-in-cell computing method for fluid dynamics*, in Methods in Computational Physics, B. Alder, S. Fernbach, and M. Rotenberg, eds., vol. 3, Academic Press, New York, 1964.

[17] R. C. Kirby, *FIAT: A new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516.

[18] M. Kucharik, J. Breil, S. Galera, P.-H. Maire, M. Berndt, and M. Shashkov, *Hybrid remap for multi-material ALE*, Computers & Fluids, In Press, Corrected Proof (2010), pp. –.

[19] M. Kucharik and M. Shashkov, *Extension of efficient, swept-integration-based conservative remapping method for meshes with changing connectivity*, International Journal for Numerical Methods in Fluids, 56 (2008), pp. 1359–1365.

[20] R. Loubere and M. J. Shashkov, *A subcell remapping method on staggered polygonal grids for arbitrary-Lagrangian-Eulerian methods*, Journal of Computational Physics, 209 (2005), pp. 105 – 138.

[21] J. C. Nedelec, *Mixed finite elements in r^3*, Numerische Mathematik, 35 (1980), pp. 315–341. 10.1007/BF01396415.

[22] ——, *A new family of mixed finite elements in r^3*, Numerische Mathematik, 50 (1986), pp. 57–81. 10.1007/BF01389668.

[23] R. Nicolaides, *Direct discretization of planar div-curl problems*, SIAM J. Numer. Anal., 29 (1992), pp. 32–56.

[24] R. Nicolaides and K. Trapp, *Covolume discretizations of differential forms*, in Compatible Discretizations, Proceedings of IMA Hot Topics workshop on Compatible Discretizations, D. N. Arnold, P. Bochev, R. Lehoucq, R. Nicolaides, and M. Shashkov, eds., vol. IMA 142, Springer Verlag, 2006, pp. 161–172.

[25] R. P. Pawlowski, E. T. Phipps, A. G. Salinger, S. J. Owen, C. M. Siefert, and M. L. Staten, *Applying template-based generic programming to the simulation and analysis of partial differential equations*, Scientific Programming, (2011). Submitted.

[26] E. S. Raymond, *The Art of UNIX Programming*, Addison-Wesley Professional, September 23 2004.

[27] N. Roberts, D. Ridzal, P. Bochev, L. Demkowicz, K. Peterson, and C. Siefert, *Application of a discontinuous Petrov-Galerkin method to the Stokes equations*, in Computer Science Research Institute Summer Proceedings, E. Cyr and S. Collis, eds., no. SAND2010-8783P, 2010, pp. 32–46.

[28] M. Shashkov, *Conservative Finite Difference Methods on General Grids*, CRC Press, Boca Raton, FL, December 05 1995.

[29] D. Shepard, *A two-dimensional interpolation function for irregularly-spaced data*, in Proceedings of the 1968 23rd ACM national conference, ACM '68, New York, NY, USA, 1968, ACM, pp. 517–524.

[30] T. M. Smith, J. N. Shadid, R. P. Pawlowski, E. C. Cyr, and P. D. Weber, *Reactor Core Sub-Assembly Simulations Using a Stabilized Finite Element Method*, in The 14th International Topical Meeting on Nuclear Reactor Thermalhydraulics, NURETH-14, 2011.

[31] S. A. Smoljak, *Quadrature and interpolation formulae on tensor products of certain function classes*, Soviet Math. Dokl., 4 (1963), pp. 240–243.

[32] N. Sukumar and E. Malsch, *Recent advances in the construction of polygonal finite element interpolants*, Archives of Computational Methods in Engineering, 13 (2006), pp. 129–163. 10.1007/BF02905933.

[33] F. L. Teixeira and W. C. Chew, *Lattice electromagnetic theory from a topological viewpoint*, J. Math. Phys., 40 (1999), pp. 169–187.

[34] T. Warburton, *An explicit construction of interpolation nodes on the simplex*, Journal of Engineering Mathematics, 56 (2006), pp. 247–262.

[35] C. Webster, *Sparse Grid Stochastic Collocation Techniques for the Numerical Solution of Partial Differential Equations with Random Input Data*, PhD thesis, Department of Mathematics and School of Computational Science, Florida State University, Tallahassee, FL, 2007.

[36] J. S. V. Welij, *Calculation of eddy currents in terms of H on hexahedra*, IEEE Trans. Magnetics, 21 (1985), pp. 2239–2241.

[37] D. Xiu and J. S. Hesthaven, *High-order collocation methods for differential equations with random inputs*, SIAM Journal on Scientific Computing, 27 (2005), pp. 1118–1139.

[38] K. S. Yee, *Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media*, IEEE Trans. Ant. Propa., 14 (1966), pp. 302–307.