Automated Finite Elements

by

Teodoro Fields Collin

B.S., University of Chicago, 2018

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2025

Authored by: Teodoro Fields Collin
Department of Electrical Engineering and Computer Science
August 15, 2025

Certified by: Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Automated Finite Elements

by

Teodoro Fields Collin

Submitted to the Department of Electrical Engineering and Computer Science
on August 15, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Finite element methods (FEMs) are a powerful and ubiquitous tool for solving engineering problems. Experimenting with different finite elements can improve the quality and efficiency of solutions. In some cases, the wrong (but nonetheless most common) choice of finite element will produce solutions which converge to the wrong answer regardless of mesh resolution. However, in practice, the choice of finite element is not explored due to the complexity of re-deriving and re-implementing finite element methods. Trying a new finite element is challenging because practitioners must manually deduce formulas to use these elements and they must implement these formulas within the context of a potentially complex system. We address this problem by introducing ElementForge, a finite element system that is parametric over the literate mathematical specification of a finite element in a domain-specific language (DSL). The ElementForge compiler reasons about tensor spaces, tensors, and tensor bases from first principles to derive implementations of finite elements. The ElementForge compiler is able to automatically derive implementations of finite elements previously only derived by hand. Further, ElementForge minimally couples several key mathematical concepts, mainly tensor fields, mesh topologies, sparse tensors, and assembled finite element operators, to produce a complete finite element system that is parametric over the choice of element. Consequently, the elements derived by the compiler can be applied parametrically to new meshes, PDEs, and boundary conditions. We evaluate our system by implementing several simulations with different finite elements, demonstrating that our system can explore tradeoffs in generality, accuracy, speed, and representational complexity. For example, we are able to implement the Morley, Bell, Argyris, and Hermite like elements with less than 50 lines of code and use them all in a single simulation.

Thesis supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

A thesis so delayed and so long could not be managed without the support of a large community. First among this community is my advisor, Saman Amarasinghe. Throughout this time, Saman has struggled a lot with me and we've both found ourselves far outside our comfort zones, but we have persevered through his patience and determination. Saman is the most supportive advisor that I know of and I couldn't have done all of this without his support, care, advice, and drive. I think that we have a lot more fun ahead of us.

Similarly, I must thank two of my close collaborators, friends, mentors: Gilbert Bernstein and Oded Stein. Without either of them, this project could not have been born had Gilbert not introduced me one day to Oded. Both have supported me as this has dragged on and provided valuable technical advice on different aspects of this project (loosely, Oded on geometry and Gilbert on topology). They have also both been my entryway into the computer graphics community (though Justin Solomon, Oded's post-doc mentor, deserves initial credit for this through his great class, Shape Analysis, and also I suppose for conveniently bringing Oded to MIT). They are both brilliant and I look forward to continuing our collaborations.

More broadly, a wide array of academics at MIT have provided me with interesting technical conversations and support in a variety of contexts, often nothing to do with this project. Among them include professors, post-docs, and research scientists: Will Detmold, Phiala Shanahan, Riyadh Baghdadi, Andrew Pochinsky, Justin Solomon, Adam Chlipala, Youssef Marzouk, Tao B Schardl, Jaime Peraire, Joel Emer, and Cuong Nguyen. I'd like to add to this list two external collaborators in this category: Mike Wagman and Anthony Grebe of FNAL. I also want to separately thank the graduate students (some now moved on to other things) who aided and were also my friends outside of MIT, including: William Moses, Robert L Van Heyningen, Chris McNally, Edan Orzech, Farid Arthaud, Logan Webber, Jesse Michel, Willow Ahrenes, Ellie Cheng, Tom Chen, Katherine Mohr, Ariya Shajii, Filippo Carloni, Francesco Peverelli, Jane Lang, Charles Yuan, Manya Bansal, Jaeyeon Won, and Kavi Gupta. I've also had the pleasure of mentoring a number of students, who have also contributed incredibly to my time thus far at MIT, including my MEng students—Richard Sollee, Ricardo Gayle Jr., and Steven Raphael—as well as many UROPs. This is surely a non-exhaustive list as I am writing this rather last minute and surely leaving out many important names. I am also lucky to belong to many academic communities at MIT, including PSAAP 3, PSAAP 4, LQCD, and Vertical AI.

Additionally, I'd like to thank people at MIT, some of whom are academics but who I've primarily known through other means. I'd especially like to thank Mary McDavitt, our local administrator, who has always supported me and made things feel easy. I'd also like to thank all the members of the MIT Graduate Student Sci-fi Book Club, but especially the people who founded it, Clara Ziran and Maya Felice Harris (who now runs the club with me and Laura Landon). I'd also like to thank Lauren McLean of the GSC for making the club possible and being very helpful on the administrative side. More broadly, I'd like to thank all the people who contribute to social life at MIT (e.g., the recently created Bike Lab or the EECS board game nights).

Of course, I'd like to thank my family for supporting me for way longer than the time it took to complete this thesis. My family has always unconditionally supported me, especially when it comes to education. They have sacrificed particularly for that in quite a few ways. I particularly want to thank my grandmother and my father, my fellow academics in the family, for their continually amusing conversations in addition to the usual support.

Lastly, in a similar spirit but beyond family, I'd like to thank things beyond MIT. In the first place, I'd like to thank my close friends who have supported me in various ways through this process, including Aliyah Bixby-Driesen, Michael Howland-Dewar, Michael Haas, Charlie Hudgins, Elisabeth Stockinger, Andy Feldman, Yueheng Zhang, Meg Doucette, Allie Segal, Bethany Sanchez, Samuel Judson, and probably many more who I am forgetting or who don't like being named in things. In the second place, I'd like to note that I am part of many communities (including two other book clubs) outside of MIT and that I feel like a part of the broader boscamberville community, which MIT cannot exist without. I'd love to name all the people involved in the various enterprises, but I'd run out of space and most of them wouldn't appreciate it anyway. Lastly, I'd like to thank one individual who provided truly invaluable friendship and support: Sofia Chandler-Freed, who has been an incredibly close friend, who has supported me throughout this entire project, and who edited parts of this thesis.

Contents

List of Figures

List of Tables

List of Source Code Listings

Chapter 1

Introduction

Finite Element Methods (FEMs) have become an important tool in a variety of fields, including applied mathematics, scientific computing, engineering, and computer graphics [1]–[3]. The FEM is powerful because one can vary the mesh, the problem formulation, the solver, and the *element*, a discrete local function space. One can choose from more than 100 families of finite elements [4],each of which provide specific quantitative and qualitative advantages and disadvantages. Within computer graphics specifically, practitioners have examined the trade-offs between convergence and computational cost of various higher order Lagrange elements [5], [6].

However, practitioners face three challenges if they want to determine and exploit the trade-offs that different finite elements (FEs) offer. The first challenge is that practitioners must deduce, from the definition of the FE, various mathematical formulas that software packages require to implement a FE [7], [8]. The second challenge is that practitioners must understand global FE operations (e.g., methods to build a sparse matrix or enforce a boundary condition) at both a mathematical and software level so that they can reimplement these operations for their class of FE. The third challenge is that for practitioners to explore these trade-offs, they must experiment with multiple FEs, encountering the first two challenges each time, and they must also experiment with their problem formulation to properly exploit some FEs.

Currently, no system offers a solution for the first challenge. Many systems offer partial solutions to the second challenge because they internally provide an interface to FEs and use this interface to implement many global FEM operations [9]–[11]. In software packages that offer solutions to a set of narrow problems, interfaces to a FE might not be present (the FE is baked into the script) or the interface only serves a narrow set of problems (e.g., it only supports certain derivatives or boundary conditions (BCs)). Finally, many systems address third problem via libraries or domain specific languages (DSLs) to vary the problem formulation [11]–[16]. These DSLs offer a high degree of flexibility for the problem specifications and they can even work with different FEs, but they do not provide a deductive system to implement FEs automatically. Our work, ElementForge, offers such a deductive system among other abstractions to meet all three challenges. As a practical consequence of the prior lack of automation, many communities stick to varying the problem to work with linear Lagrange elements, hiding a large space of trade-offs involving both higher order Lagrange elements and *non-Lagrangian elements*.

We propose to significantly lower the cost of varying FEs and problems with Element-Forge, the first FE DSL that is fully parametric over a mathematical definition of a FE. We believe that ElementForge will allow many to find and take advantage of trade-offs among FEs and associated problem formulations. To answer the first challenge, Element-Forge contains a language for specifying FEs and a compiler that automatically deduces the formulas required to be implemented by hand in other packages, thus automating work previously found in mathematical journals. To answer the second challenge, ElementForge offers several core primitives that are sufficient for users to systematically generate global finite element operations that translate local formulas to useful code for building and manipulating sparse matrices. To answer the third challenge, ElementForge allows users to specify suitable problems with the same language that it provides to specify FEs. Finally, ElementForge agglomerates these solutions into a single package that allows users to define, reuse, and combine various FEs, problem specifications, and global routines to rapidly explore FEMs.

The core design principle of ElementForge is to systematically model the discrete mathematical objects of FEMs. In contrast to prior work, our software architecture does not emphasize either the global theory of PDEs or the structure of a specific class of problems. ElementForge emphasizes the discrete mathematical objects at the core of FEM *implementations* as these are the tools require to save labor through automation. Consequently, ElementForge pays special care to pointwise manipulation of tensor fields on manifolds, the definition of an FE, the construction of a discrete mesh, and the algebraic construction of global operations via the interaction of pointwise expressions and discrete meshes. These new tools provide the means to go from a mathematically specified FE and operator to a sparse matrix, automatically, and, hopefully, understandably.

Our implementation is not without some limitations, especially as we currently do not pay attention to low level performance issues; many FEM packages have outlined techniques to optimize code generated for FEs with special structures and these techniques could eventually be employed here [17]–[19]. Even with these limitations, we are able to implement novel simulations of independent interest to different areas of computer graphics. Moreover, we view these simulations and our software as part of an effort to make FEs more accessible. We make the following contributions:

1. To the best of our knowledge, the first FE DSL that is parametric over the definition of a FE, enabling for the first time rapid exploration of many different FEMs for the same problem.

2. A novel DSL and compiler for specifying FEs and operators via coordinate-free tensor algebra. This system enables the automated deduction of formulas needed to implement FEs at the local level.

3. An interface for the systematic construction of global FE operations, enabling core and

new global FE operations to be built on a wide variety of elements.

4. An overall compiler taking as input (i) a FE description, (ii) operator specifications, and (iii) a mesh, and producing discretization operators (and interpolation functions) as output.

5. Several novel simulations of interest to computer graphics:

   (a) A more efficient yet simpler data interpolation/smoothing method based on the natural BCs of the biharmonic equation, identified in [20], and the Morley element [21]. Moreover, we have explored a variety of FEs for this class of problems, leading to characterization of some of the tradeoffs involved.

   (b) The first implementation of the finite element exterior calculus (FEEC) that is derived purely and automatically from the mathematical specification, which allows us to reimplement the analogues of all the discrete exterior calculus (DEC) operators, leading to FEM versions of classic DEC applications.

   (c) Several FEs for Stokes flows problems in 2D+time and 3D+time, illustrating the phenomena of locking incompressible fluids (in contrast to prior illuminations focused on elasticity).

6. We show that deductive symbolic computation is a practical tool for declarative scientific computing, even when the deductions potentially involve inverting large symbolic matrices, as is often the case. In particular, we show that for FEs, the complexity of inverting a symbolic Vandermonde matrix, a core step in the deduction of elements, is practical by ensuring that the structure of the matrix is deduced and exploited in the inversion.

7. A programming language for multi-linear algebra that properly separates bases from tensors, the way they are presented in most textbooks. Though this idea has motivated

some other languages (e.g., UFL [12] and Albert [22]), we believe this is the purest implementation and the one that most carefully captures the textbook representation.

8. A novel separation between computations within mesh cells and the mesh cells themselves, encoded into the design of a novel language for integrated pointwise tensor expressions on manifolds. We carefully manage the availability of geometric information to enable automation that helps deduce important information about FEs, reducing their code size.

9. A mathematical model for global FE computations. In particular, we design:

    (a) A succinct mathematical model of meshes, sub-meshes, data on meshes, and global fields sufficient for FEMs.

    (b) A novel mathematical model/semantics for a wide array of computations on meshes, single valued assembled operators (SVAO), which when paired with a small set of sparse array operations, models most global FEM computation.

The model built from these ideas is parsimonious, malleable, and robust to changes in user code, and expressive.

Chapter 2

Examples

To illustrate the scope of our system and our way of thinking, we run through a few examples that use our highest-level interface.

2.1 Laplace equation

Given a surface Ω and a boundary condition $f\colon \partial\Omega \to \mathbb{R}$, we want to find a function $u\colon \Omega \to \mathbb{R}$ such that

$$\begin{aligned} \Delta u(x) &= 0 \text{ for } x \in \Omega \\ u(x) &= f(x) \text{ for } x \in \partial\Omega\ . \end{aligned} \tag{2.1}$$

Our program uses linear Lagrange finite elements to produce the familiar cotangent Laplace matrix.

```
from EF import LaplaceOperator, TriangleVertexListReader, assemble, LinearLagrange

% Load a mesh.
mesh = TriangleVertexListReader(filename)



@pointwiseIntegral
def laplace(u: field(R), v: field(R)):
    return grad(u).dot(grad(v))

% Make a sparse matrix, the cotan Laplacian.
laplaceMatrix = assemble(mesh, [laplace], [LinearLagrange,LinearLagrange])
```

Source Code Listing 2.1: Solving the Laplace equation with FEM in ElementForge.

Listing listing 2.1 shows how this works in our system: we import necessary functions and objects, read a mesh, express an integral, and then assemble the corresponding Laplacian matrix. The `laplace` function is an integral: it takes as input two fields u and v, and integrates over the inner product of their gradients, $\text{laplace}(u, v) = \int \nabla u \cdot \nabla v$. The assemble call builds the Cotan (or stiffness) matrix, $L_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\psi_j$, where Ω is proscribed by the `mesh` argument and the functions $\{\phi_i\}$ and $\{\psi_j\}$ are proscribed by the two `LinearLagrange` arguments.

2.2 Weak Formulations: Smoothness and Boundary Conditions

Our Laplace function in listing 2.1 does not directly implement Δ as seen in eq. (2.1) (the *strong* Laplacian), but instead implements the *weak* Laplacian. To utilize linear Lagrange functions to build a cotan Laplacian, we must transform the initial problem. This has been described many times but we will belabor some of the details of derivation to illustrate the interaction between vector spaces of functions (function spaces) and aspects of the PDE.

By choosing an appropriate space of functions, V, we can integrate by parts to go from eq. (2.1) to the weak formulation [3]:

$$\int_\Omega v0 = \int_\Omega v\Delta u = \int_\Omega \nabla v \cdot \nabla u - \int_{\partial\Omega} \frac{\partial u}{\partial n} v. \tag{2.2}$$

The idea here is that an arbitrary $v \in V$ is *testing* whether $\Delta u = 0$ for some given $u \in V$. However, we know that $u|_{\partial\Omega} = f$, so we don't need to test there, justifying that we weaken $v \in V$ to $v \in V_0 := \{v \in V : v|_{\partial\Omega} = 0\}$. This leads us to the `laplace` operator from the previous section:

$$\int_{\Omega} \nabla v \cdot \nabla u - \cancelto{0}{\int_{\partial\Omega} \frac{\partial u}{\partial n} v} = \int_{\Omega} \nabla v \cdot \nabla u. \tag{2.3}$$

Via our choice of V the problem becomes: find a $u \in V$ that satisfies the BCs in eq. (2.1) such that for all $v \in V_0$:

$$\int_{\Omega} \nabla v \cdot \nabla u = 0. \tag{2.4}$$

This is the weak formulation, and it mirrors our Laplace function in listing 2.1. Note that this is an infinite-dimensional problem, much like eq. (2.1). To make the problem discrete, one replaces V with a suitable finite-dimensional space such as the linear Lagrange elements on a triangle mesh. Having done this, one can, as many have observed [23], retrieve the traditional cotan Laplacian. Thus, we are able to solve a second order problem with functions that only have a single derivative. In general, using integration by parts allows one to transform a strong form problem with $2n$ derivatives to a weak one with n derivatives, which allows one to use functions that only need n derivatives [3].

However, to use the weak formulation, we observe that two practical matters have not been resolved[1]:

1. Can we actually integrate by parts on some finite dimensional V? (Smoothness)

2. Can we determine a subspace V_0? (BCs)

In the case of linear Lagrange FEs for Laplace's equation, these questions are easy to resolve. We now turn to other useful operators for which this question is not as clear.

[1]these are necessary, but not sufficient

2.3 Weak Formulation of the Biharmonic Equation

Let's examine the biharmonic equation, a slightly more complicated version of the Laplace equations where the nuances of the weak formulation begin to show. Given a domain Ω and BCs $f_i\colon \partial\Omega \to \mathbb{R}$ $(i = 0, 1)$, we seek a $u\colon \Omega \to \mathbb{R}$ such that

$$\Delta\Delta u = 0 \text{ on } \Omega \tag{2.5}$$

and

$$u = f_0 \text{ and } \frac{\partial u}{\partial n} = f_1 \text{ on } \partial\Omega. \tag{2.6}$$

We transform the problem into its weak form with $v \in V \subset H^2$:

$$\begin{aligned}\int_\Omega 0v &= -\int_\Omega (\Delta\Delta u)v \\ &= \int_\Omega (\nabla\Delta u)\cdot\nabla v - \int_{\partial\Omega} v\frac{\partial\Delta u}{\partial n} \\ &= \int_\Omega \Delta u\Delta v - \int_{\partial\Omega} \Delta u\frac{\partial v}{\partial n} - \int_{\partial\Omega} v\frac{\partial\Delta u}{\partial n}.\end{aligned}$$

Similar to Laplace, we now formulate a V_0-like subspace to erase the boundary terms. We set $V_0 = \{v \in V\colon v|_{\partial\Omega} = \frac{\partial v}{\partial n}|_{\partial\Omega} = 0\}$.

Our weak formulation then simplifies to: find a $u \in V$ subject to (2.6) such that for all $v \in V_0$:

$$0 = \int_\Omega \Delta u\Delta v. \tag{2.7}$$

Returning to a practical finite dimensional choice for V, we observe that linear Lagrange (or even higher order Lagrange) cannot possibly work based on our first two requirements:

1. Even higher-order Lagrange basis functions are only continuous so they must contain

functions that break two rounds of integration by parts (via Sobolev's inequality [3]).

2. If we represent a function with Lagrange basis functions, we cannot identify a subset of the basis functions ϕ_i so that $\phi_i = \frac{\partial \phi_i}{\partial n} = 0$.

We can easily express the Biharmonic operator in our system (although the choice of element is not clear yet, since our existing elements are still only continuous):

```
@pointwiseIntegral
def biharmonic(c: chart, u: field(R, 2), v: field(R, 2)):
    # trace of Hessian with respect to metric
    # because the inner product of tensors is a trace
    deltau = (c.CT().inner()).dot(u(2))
    deltav = (c.CT().inner()).dot(v(2))
    return deltau * deltav

biharmonicMatrix = assemble(mesh, [biharmonic], [UnknownElement,UnknownElement])
```

Source Code Listing 2.2: The Biharmonic Operator.

2.4 Redux: Other Weak Formulations of Laplace's equation

To design other elements, we revisit Laplace's equation to consider how it could be solved with other operations, i.e., through variations on the weak form. We consider three variations: discontinuous Galerkin methods, Nitsche's method, and mixed methods. These variations offer ways to overcome deficiencies in an element (e.g., in some circumstances, a mixed method formulation of the biharmonic equation can be solved via linear Lagrange). Additionally, these variations also teach us how to define elements that let us use simpler weak formulations.

A discontinuous Galerkin formulation offers a way to understand smoothness and weak formulations. To derive it, suppose for a moment that $\Omega = T_0 \cup T_1$ where functions in V

can be integrated by parts on T_0 and T_1. Then we can redo our derivation of the weak formulation of Laplace's equation with some careful manipulations on the set $\partial T_0 \cap \partial T_1$:

$$
\begin{aligned}
\int_\Omega v0 = \int_\Omega \Delta u v &= \int_{T_0} \Delta u v + \int_{T_1} \Delta u v \\
&= -\int_{\partial\Omega} \frac{\partial u}{\partial n} v - \int_{\partial T_0 - \partial\Omega} \frac{\partial u}{\partial n} v - \int_{\partial T_1 - \partial\Omega} \frac{\partial u}{\partial n} v + \int_{T_0 \cup T_1} \nabla u \cdot \nabla v \\
&= -\int_{\partial\Omega} \frac{\partial u}{\partial n} v - \int_{\partial T_0 \cap \partial T_1} \left(\frac{\partial u}{\partial n} v\right)|_{\partial T_0} - \left(\frac{\partial u}{\partial n} v\right)|_{\partial T_1} + \int_\Omega \nabla u \cdot \nabla v
\end{aligned}
$$

We note that if our functions in V were continuous, the new middle term would vanish. We can confirm this empirically via listing 2.3 where we introduce integrals over sub-facets and an interface to geometry via chart objects that refer to parts of the mesh in an abstract manner.

```python
from EF import LaplaceOperator, TriangleVertexListReader, assemble, LinearLagrange,
    difference

# Load a mesh.
mesh = TriangleVertexListReader(filename)

# Setup integral on boundaries of cells
# Take differences between the same edge on difference cells
@pointwiseIntegral(BoundaryChart, reduction=difference)
def interior(u: field(CellChart), v: field(CellChart)):
    n0 = u.chartOf().normalVector()
    return grad(u).dot(n0) * v(0)

interiorMatrix = assemble(mesh, [interior], [LinearLagrange,LinearLagrange])
# This matrix should be == 0 except for boundary vertices
```

Source Code Listing 2.3: An Interior Facet Integral for Testing Continuity.

At this point, the DG method becomes less intuitive as the weak formulation is still not usable without a penalty term to encourage continuity. Ignoring the boundary terms and symmetry, the DG formulation might resemble:

$$\int_\Omega \nabla u \cdot \nabla v + \int_{\partial T_0 \cap \partial T_1} \left(\frac{\partial u}{\partial n} v\right)|_{\partial T_0} - \left(\frac{\partial u}{\partial n} v\right)|_{\partial T_1}$$
$$+\sigma \int_{\partial T_0 \cap \partial T_1} (u|_{\partial T_0} - u|_{\partial T_1})\,(v|_{\partial T_0} - v|_{\partial T_1})$$

where $\sigma \in \mathbb{R}$. We can implement such terms, but our purposes here warrant one remark: the penalty term is a constraint where the other interior integral term emerges from integration by parts. The core lesson is that performing the above integration by parts exercise yields a condition that a finite element must pass for integration by parts to work.

Nitsche's method operates similarly to DG methods, but it is a useful technique for enforcing general boundary conditions. Nitsche's method saves us from having to identify V_0 within V. If we go back to before we utilize $v \in V_0$ in integration by parts then append a penalty parameter $\sigma > 0$, we get: find a $u \in V$ such that for all $v \in V$:

$$\int_\Omega \nabla v \cdot \nabla u - \int_{\partial\Omega} \frac{\partial u}{\partial n} v + \sigma \int_{\partial\Omega} (u - f)v.$$

This formulation can be symmetrized and rewritten as a bilinear and linear part without issue:

$$\int_\Omega \nabla v \cdot \nabla u - \int_{\partial\Omega} \frac{\partial u}{\partial n} v - \int_{\partial\Omega} \frac{\partial v}{\partial n} u + \sigma \int_{\partial\Omega} uv = \sigma \int_{\partial\Omega} fv.$$

This weak formulation is obviously more complex than the original formulation and features a penalty parameter that one must determine. However, the formulation works for more general V where we cannot easily determine V_0. Further, this formulation teaches us how to identify V_0. The core lesson of the formulation is that to identify the boundary (degrees of freedom) DOFs, V_0, we must determine which basis functions are necessary and sufficient for the terms $\int_{\partial\Omega} uv$ and $\int_{\partial\Omega} \frac{\partial v}{\partial n} u$ to be zero. Further, if we wish to find a good approximation to boundary data, we need to investigate the system $\int_{\partial\Omega} uv = \int_{\partial\Omega} fv$. A simple modification to listing 2.3, shown in listing 2.4, can help us with these tasks in a way that generalizes easily to non-linear Lagrange elements and other PDEs.

```python
from EF import LaplaceOperator, TriangleVertexListReader, assemble, LinearLagrange,
    difference

# Load a mesh.
mesh = TriangleVertexListReader(filename)
# Get the boundary
boundary = mesh.topologicalBoundary()
# Setup integral on boundary
@pointwiseIntegral(BoundaryChart)
def interior(u: field(CellChart), v: field(CellChart)):
    n0 = u.chartOf().normalVector()
    return grad(u).dot(n0) * v(0)

interiorMatrix = assemble(mesh, [interior], [LinearLagrange,LinearLagrange],
    restrictIntegrals=boundary)
# the non-zero rows that correspond to boundary vertices make up V_{0}
V0Entities = interiorMatrix.nonZeroEntities([boundary, None])
```

Source Code Listing 2.4: A Boundary Facet Integral for Nitsche's Method.

A final strategy to overcome smoothness requirements and enforce boundary conditions is mixed methods. Mixed methods introduce intermediate variables with weaker smoothness requirements. For Laplace, we can utilize that $\Delta = \operatorname{div} \nabla \cdot$ to introduce another variable, $\sigma = \nabla u$. We can thus rewrite the strong form of the PDE to finding a $u \colon \Omega \to \mathbb{R}$ and $\sigma \colon \Omega \to \mathbb{R}^n$ such that

$$\sigma - \nabla u = 0, \operatorname{div} \sigma = 0 \tag{2.8}$$

with boundary condition

$$u(x) = f(x) \text{ for } x \in \partial\Omega. \tag{2.9}$$

If we had a Neumann condition, we would now write it as a condition on σ, which means the condition can be directly enforced like a Dirichlet condition (as opposed to a variational enforcement in the original formulation). Turning back to the weak form, we see that we now have two functions to solve for. So we now need two spaces, V and Σ. The weak formulation can be derived by multiplying the first equation in the split strong form PDE by a function in $\gamma \in \Sigma$ and the second equation by a function in $v \in V$:

$$\int_\Omega \gamma \cdot (\sigma - \nabla u) = 0, \int_\Omega v \operatorname{div} \sigma = 0 \tag{2.10}$$

The second term in the first equation allows for integration by parts, which reduces the smoothness requirements on u. The final formulation reads: find $u \in V$ and $\sigma \in \Sigma$ such that for all $v \in V_0$ and $\gamma \in \Sigma_0$:

$$\int_\Omega \sigma \cdot \gamma + (\operatorname{div} \gamma)u + v(\operatorname{div} \sigma) = 0. \tag{2.11}$$

We observe that we do not take derivatives on u or v, so these functions need not be continuous across cell boundaries. Similarly, we only take a divergence on γ and σ, so these vector functions need not be continuous either. However, the functions in Σ cannot be totally discontinuous, as integration by parts for a divergence requires that functions be continuous in the normal direction at the boundary between two cells (as one can see by repeating the analysis in the DG section for a term that looks like $\sigma \cdot \nabla u$). Naturally, this is also why we expect to be able to enforce a Neumann boundary condition directly. Since we have not yet introduced any function spaces of vector elements, we simply sketch the code for the $(\operatorname{div} \gamma)u$ in listing 2.5.

```python
cty = TopChart # Highest chart in the mesh.
# Vector field and scalar field
@pointwiseIntegral
def incompressibility(u: field(cty.Rk()), p: field(R)):
    jac = D(u)
    div = jac.trace(0, 1)
    return div * p

incompressibilityMatrix = assemble(mesh, [incompressibility], [unknownVectorElement,
    DGLinearLagrange])
```

Source Code Listing 2.5: A Divergence Operator.

We also note that $(\operatorname{div}\gamma)u$ also plays a critical role in incompressible Stokes and elasticity, especially with respect to locking phenomena. A term in an elastic energy might be of the form:

$$I(F,p) = \int_\Omega pC(J(F)) \tag{2.12}$$

where F is a displacement vector field, p is a scalar pressure field, J is the Jacobin Determinant operator (to extract the third invariant of the right Cauchy-Green deformation tensor), and $C\colon \mathbb{R} \to \mathbb{R}$ is a constraint function minimized by $J(F) = 1$. Minimizing the energy via variational derivatives will yield a weak formulation with a divergence term, $(\operatorname{div}\gamma)u$. Choosing FEs to approximate F and p is a critical problem which can often lead to ill-posed discrete problems. A core contribution of the FEEC is to supply a collection of FEs to handle such problems, which is another reason to investigate a wide variety of FEs and operators [24].

2.5 Other Finite Elements

As we have illustrated in the previous subsections, advancing beyond the simplest PDE quickly yields weak formulations whose form calls for different FEs. Moreover, we saw that the structure of the weak formulation (exposed via various transformations) implies that the solvability, convergence, or stability of the problem depends on utilizing FEs with specific properties. Additionally, the choice of the FE basis heavily influences the structure of the resulting matrices, which impacts the efficiency of solvers along many axes. We now explore how to specify elements intuitively and with code. We will describe finite elements more formally in section 3.2.

As is already known by many, higher-order Lagrange elements provide higher degrees of convergence, but the resulting sparse matrices also have larger dense blocks, which can allow greater degrees of efficiency on platforms such as GPUs, leading to a better accuracy per flop [25]. If we use discontinuous Lagrange elements, some matrices (e.g., a mass matrix) might also be block diagonal. The linear Lagrange element in our system is mostly the specification of a point evaluation at a vertex coupled with a domain and a polynomial degree.

```
cty = ChartType(0) # on objects of dimension 0
@pointwiseEval(cty, barycentricCoordinate=0.0)
def vertexEval(u: field(cty, R, 0)):
    return u(0)


# element type, degree, List[Dof]
elementEdge = Sybil.FE(1, 1, [vertexEval])
elementTri = Sybil.FE(2, 1, [vertexEval])
elementQuad = Sybil.FE((1,1), 1, [vertexEval])

# Discontinuous variant:
@pointwiseEval(cty, barycentricCoordinate=0.0, storeAt=TopChart)
def vertexEvalDG(u: field(cty, R, 0)):
    return u(0)
elementEdgeDG = Sybil.FE(1, 1, [vertexEvalDG])
elementTriDG = Sybil.FE(2, 1, [vertexEvalDG])
```

Source Code Listing 2.6: A Linear Lagrange Dof, specifying a pointwise evaluation at a vertex, used to define linear Lagrange FEs

Up to this point, we have not addressed the ChartType in our examples except for the obvious BoundaryChart or CellChart, but here the meaning is especially critical: the chart type in the pointwise macro indicates that the pointwise object is parametric over a vertex (as opposed to an edge, triangle, or hex). The pointwiseEval indicates an evaluation at a single point (specified via barycentric coordinates) in an object associated to a given chart. The chart type in field indicates that an incoming field needs to provide pointwise values in R on vertices. Finally, the storeAt attribute associates the evaluation to a different geometric object, allowing for the creation of a vertex evaluation that is repeated per edge, triangle, or quad. Critically, since the chart indicates that this object can be run on a vertex and not just some particular vertex, we can consider all the possible representatives on a triangle: three point evaluations, one for each vertex. This is roughly the description of linear Lagrange elements found graphically in fig. 3.5.

Pointwise evaluations of quadratic elements (also depicted in fig. 3.5) is similar.

```
cty = ChartType(1) # on objects of dimension 1
@pointwiseEval(cty, barycentricCoordinate=(0.5, 0.5))
def edgeMidEval(u: field(cty, R, 0)):
    return u(0)

# element type, degree, List[Dof]
elementEdge = Sybil.FE(1, 2, [vertexEval, edgeMidEval])
elementTri = Sybil.FE(2, 2, [vertexEval, edgeMidEval])
elementQuad = Sybil.FE((1,1), 2, [vertexEval, edgeMidEval])
```

Source Code Listing 2.7: A Quadratic Lagrange DOF, specifying a pointwise evaluation on an edge.

In this case, the `ChartType` refers to an edge and the barycentric coordinate refers to the midpoint of the edge. By considering all of the possible instances of this object of a triangle and combining them with the instances of the `vertexEval`, we get the quadratic Lagrange element found in fig. 3.5.

As is revealed in the graphic and the code, a core part of a FE is a list of DOFs, which are often defined via point evaluations of formulas of functions and derivatives. These DOFs determine properties of basis functions. We are particularly interested in how the DOFs relate to the properties of basis functions that we can specify with a weak form, such as the continuity penalty term in the DG formulation of Laplace's equation. For example, the continuous Lagrange element ensures that extra terms in the DG formulation of the Laplace's equation vanish. We can now return to the biharmonic equation and examine several plausible finite elements.

Performing the DG analysis on the weak form of the biharmonic equation yields:

$$\int_{\partial T_0 \cap \partial T_1} (\Delta u \frac{\partial v}{\partial n} - v\frac{\partial \Delta u}{\partial n})|_{\partial T_0} - (\Delta u \frac{\partial v}{\partial n} - v\frac{\partial \Delta u}{\partial n})|_{\partial T_1} = 0$$

Roughly, we need that either Δu or $\frac{\partial u}{\partial v}$ are continuous on the boundaries between cells and similarly for either u or $\frac{\partial \Delta u}{\partial n}$. Just as the Lagrange element ensures continuity by using point evaluations on cell boundaries, we might seek to do the same with gradients. Starting with a cubic Lagrange element, one can replace point evaluations on edges with gradients on vertices in at least two ways: using the x and y to make two evaluations or using the edge tangent directions. These options yield two types of Hermite DOFs, depicted in code in listing 2.8 and listing 2.9. Contrasting these two code snippets yields some important differences: the first is C^1 at vertices while the latter is C^1 at edges in the tangential direction, which is signaled in the code based on where the derivatives used in the DOFs are defined. Based on this analysis, we do not expect the Hermite element to always be able to solve the biharmonic equation as it will never provide a continuous Laplacian and $\frac{\partial u}{\partial v}$ will only be continuous when all cell normals are identified with x/y directions or edge tangents. However, both options can be used to solve Laplace's equation. In our results, we will examine the Morley element. Morley is quadratic, ensuring $\frac{\partial \Delta u}{\partial n} = 0$. Morley uses normal gradient DOFs at edge points to provide continuous normal gradients. The Morley element is the minimum viable element that can be used with the standard weak formulation of the biharmonic problem. In contrast to the Hermite element, though, the Morley element does not offer a continuous function so cannot be used to solve Laplace type problems that Hermite or Lagrange can solve.

```
cty = ChartType(0) # on objects of dimension 0
# Since we take derivatives at vertices, we represent the derivatives in Euclidean space as
    we do not have access to a tangent space.
@pointwiseEval(barycentricCoordinate=0.0)
def hermiteDof1(u: (field(cty, R, euclideanDerivative=True), field(cty, cty.Rk))):
    return u(1)[0]
@pointwiseEval(barycentricCoordinate=0.0)
def hermiteDof2(u: (field(cty, R, euclideanDerivative=True), field(cty, cty.Rk, 0))):
    return u(1)[1]

# element type, degree, List[Dof]
hermLine= Sybil.FE(1, 3, [vertexEval, hermiteDof1])
hermTri = Sybil.FE(2, 3, [vertexEval, cellCenterEval, hermiteDof1, hermiteDof2])
hermQuad = Sybil.FE((1,1), 3, [vertexEval, cellCenterEval, hermiteDof1, hermiteDof2])
```

Source Code Listing 2.8: Hermite Dofs, version 1. Compare and contrast the handling of the derivative values with version 2.

```
cty = ChartType(1) # on objects of dimension 1
# Since we are on edges, we have edge tangent spaces, which can represent derivatives.
@pointwiseEval(barycentricCoordinate=(1, 0))
def hermiteDof1(c: chart, u: field(cty, R)):
    return u(1).dot(c.tangentVector(0))
@pointwiseEval(barycentricCoordinate=(0, 1))
def hermiteDof2(u: field(cty, R)):
    return u(1).dot(c.tangentVector(0))

# element type, degree, List[Dof]
hermLine= Sybil.FE(1, 3, [vertexEval, hermiteDof1])
hermTri = Sybil.FE(2, 3, [vertexEval, cellCenterEval, hermiteDof1, hermiteDof2])
hermQuad = Sybil.FE((1,1), 3, [vertexEval, cellCenterEval, hermiteDof1, hermiteDof2])
```

Source Code Listing 2.9: Hermite DOFs, version 2. Compare and contrast the handling of the derivative values with version 1.

For our divergence constraint that appears in the mixed formulation of Laplace (as well as in vector Poisson equations and incompressible Stokes equations, which we will study in the results section), we offer a simple vector element as an example. The Brezzi-Douglas-Marini (BDM) element provides continuity of the normal component of a piecewise affine vector field, as depicted in listing 2.10. We also depict this visually in fig. 3.5.

```
cty = ChartType(1) # on objects of dimension 1
R2D = R^2**None # Roughly, \mathbb{R}^2
AffineVectorPolysR2D = P(2, 1, space=R2D)

@pointwiseEval(barycentricCoordinate=(1, 0))
```

```
def bdmDOF1(c: chart, u: field(cty, R2D)):
    return u(0).dot(c.normalVector(0))
@pointwiseEval(barycentricCoordinate=(0, 1))
def bdmDOF2(c: chart, u: field(cty, R2D)):
    return u(0).dot(c.normalVector(0))
#element type, space of polys, List[Dof]
BDMTRI = Sybil.FE(2,
AffineVectorPolysR2D,
[bdmDOF2, bdmDOF1])
```

Source Code Listing 2.10: A BDM Element.

2.6 Boundary Conditions

Any high level FE software must have a comprehensible and usable system for enforcing boundary conditions (among other constraints). As we saw by examining weak formulations of Laplace's equations, boundary conditions can appear in several different ways depending on how a PDE is formulated. BCs and other constraints can be enforced either by adding to the weak formulation (e.g., Nitsche's method) or by elimination methods. An elimination method applies when the subspace identified by a homogeneous boundary condition corresponds exactly to a span of a subset of basis functions. In such cases, linear operators can often be decomposed to eliminate the BC from the system of equations. Since elimination methods exactly enforce boundary conditions and easily ensure the solvability of systems of equations, we must support them.

As far as we know, no systems have consistent support for elimination methods on more complex FEs, and for good reason: determining the relationship between basis functions and boundary conditions can be highly non-trivial.[2] We offer a solid middle ground: given

[2]Consider two familiar families of functions: Lagrange polynomials and orthonormal polynomials. Enforcing Dirichlet BCs (or constraints on any point value) on the former is a relatively trivial matter because they are defined via pointwise evaluations. In the simplest case of linear Lagrange functions, one can simply remove the DOFs associated with vertices on the boundaries and trivially adjust the problem to compensate. In contrast, many papers and systems that use orthogonal polynomials struggle to support anything but periodic boundary conditions because the relationship between coefficients on orthogonal polynomial bases and point-values is non-trivial.

a mesh, a FE, a BC, and potentially a right hand side for a BC (f), we offer a variety of utilities to check if elimination methods can work or to utilize alternatives.

In listing 2.11, we show a typical workflow. First, a mesh subset is identified, with the built-in topological boundary or via external data (specified with a pointwise condition or by loading external data). Second, we can specify constraints in the same language we use to write DOFs or weak formulations. Then we can combine these to produce an object representing constraints. We can query the constraints to see if they can be factorized. If constraints can be factorized, we get an object that can be used to slice a sparse matrix/solution vector to extract the boundary part. If constraints cannot be factorized, we can query the constraints with a right hand side to build a matrix that enforces the conditions. The constraint matrix and vector are not guaranteed to ensure the correct enforcement of these constraints (as sometimes the only viable approach is to fix the weak form), but these can be useful for a first attempt or to debug.

```

meshSubset = mesh.topologicalBoundary()
# Alternatively, mesh subsets can be identified via testing points
# e.g., we can ask what vertices are on x == 0
# meshSubset = assemble(mesh, LinearLagrange, [lambda x: x[0] == 0]).asSubset()

cty = BoundaryChart
@pointwise
def dirichletConstraint(u: field(cty, R, 0)):
    return u(0)
@pointwise
def neummanConstraint(c: cty, u: field(cty, R, 1)):
    return c.normalVector().dot(u(1))

dirichletMat = assembleConstraints(mesh, meshSubset, dirichletConstraint, linearLagrange)
# Get DOFs that can be factored out of a linear system
dirichletEntities = dirichletMat.factorize()
# dirichletEntities can be used to extract the boundary part of a function or slice a matrix

neummanMat = assembleConstraints(mesh, meshSubset, neummanConstraint, linearLagrange)
neumannEntities = neummanMat.factorize()
```

```
# will be None, you must use the sparse matrix for these constraints in your solver.
(constraintsMat, vec) = neumannMat.asConstraintMatVec(lambda x: ...)
# This still might not work - you must analyze how this will interact with the weak form.
```

Source Code Listing 2.11: Building Constraints and Factoring Them Out (or Not).

2.7 Loading External Data

As alluded to in listing 2.1, loading a triangle mesh into ElementForge is easy as it is in all FE systems. Similarly, in listing 2.11, we show that data defined in Python functions can be parlayed into a subset of a mesh. These two examples represent two classes of data input: array-based and function-based. ElementForge supports these inputs at each level of data defined on a given mesh. In a simpler system, loading data on that mesh (e.g., for a right hand side, to define a boundary via a distance function, for proscribed boundary values, or for testing against analytical solutions) would be simple too: just load a value per vertex or evaluate a function at each vertex. However, just as with BCs, we must ask a more complex question: how do we load data over general meshes and elements? We solve this problem with three capabilities. First, our system reasons about the topologies of data to use external arrays for various tasks. Second, our system supports general *interpolation* from outside data and to finite elements or other groups of expressions, which via the first mechanism can be repurposed to represent mesh subsets or functions. Third, our system supports using general Python functions in the assembly of operators into sparse matrices or dense vectors. As a preview, the first two features are implemented via a single code generation approach.

To load data from arrays, ElementForge generalizes the mechanisms by which we load meshes. Meshes are loaded first as cell vertex lists, but other objects (e.g., edges) and relations (cell edge lists) are created. Mesh subsets can be loaded as well as data on meshes using representations of objects as ordered list of vertex tuples. We depict a simple workflow in listing 2.12.

```
V = np.load(...)
F = np.load(...)

mesh = meshFromVertexCellList(V, F)

Esubset = np.load(...) # |E| by |2| array

# Determine how these edges relate to edges in the system
meshSubset = mesh.orderedSubset(Esubset)

dataPerEdge = np.load(...)

# Reorder data for the mesh in our system:
edgeDataOnMesh = dataPerEdge[meshSubset]
```

Source Code Listing 2.12: Associating Array Data to Meshes.

Interpolation is an overloaded word; in ElementForge, interpolation refers to evaluating DOFs on a function to get coefficients for a function represented in a FE basis. The evaluations for some elements are again depicted graphically in fig. 3.5 and a code snippet is provided in listing 2.13. Our interpolation can almost be thought of in a black box manner as it is highly general. So long as the types are compatible, ElementForge allows interpolation to occur, though it will produce warnings if the interpolation might be ill-defined due to discontinuities in a function or derivative. For example, if we interpolated from a Lagrange element to a Hermite element, a discontinuity in the gradient of the former could result in two different potential values for a Hermite DOF. Our system issues a warning and then picks one (though a lower-level interface described later allows the user to determine other ways to resolve such issues). Additionally, some FE coefficients require computing integrals, which means resorting to numerical quadrature. The quadrature degree can be configured and sometimes introduces another layer of approximation.

```

@jax.jit
```

```
def outsideFunc(x):
    return # Arbitrary jittable python

data1 = interpolate(mesh, unknownElement1, outsideFunc)
data2 = interpolate(mesh, unknownElement2, data1)
# Interpolated data can sometimes be used as a mesh subset:
mesubSubet = data1.submerse()
```

Source Code Listing 2.13: Using Interpolation to Define a Boundary Condition.

While interpolation is useful for representing outside data with a FE, interpolation sometimes introduces error early in the process. If we add a right hand side to Laplace's equation ($\Delta u = f$), we introduce a right hand side in the weak formulation, $\int vf$. If we supply f as an interpolated function that approximates the true f at a linear rate, then a method that solves u at a quadratic rate might still converge at a linear rate. Thus, our third way to use external data: we allow Python functions to be provided in the assembly process directly, as depicted in listing 2.14. We note that an important application of this feature is computing error against analytic examples, as error can be computed inaccurately without it.

```

# Define rhs
def f(x):
    ...
# Setup mass:
@pointwiseIntegral
def mass(u: field(R), v: field(R)):
    return u.dot(v)
# Directly compute the RHS:
rhsSide1 = assemble(mesh, mass, [LinearLagrange, f])
# Alternative: interpolate and compute
interp =  assemble(mesh, LinearLagrange, f)
rhsSide2 = assemble(mesh, mass, [LinearLagrange, interp])
# Third option: assemble mass and use dot product against interpolated function data
mass = assemble(mesh, mass, [LinearLagrange, LinearLagrange])
rhsSide3 = mass.dot(interp.array)
```

Source Code Listing 2.14: Different ways external functions can interact with operator assembly: directly, with interpolation, and via linear algebra. The latter two should be identical up to floating point if the correct quadrature degree is used.

Chapter 3

Background

3.1 Vectors, Tensors, and Tensor Calculus

We require various notions from linear algebra and calculus. At an algorithmic level, we need this material to explain intuitively the core difficulty of using general FEs: transforming elements between mesh entities. At a higher level, we need this material to explain our language, as another core difficulty is also simply writing down FEs independently of mesh entities. These two core difficulties are related, and we explore this successively through vectors, tensors, and then tensor-valued functions. In particular, we are interested in the phenomenon of the naturalness of linear algebra objects: vectors that behave nicely under change of basis/transformation are precisely those that can be written down independently of either the underlying vector space or a specific basis for it. (As a simple example, we can always write down a zero vector in any vector space). Furthermore, if we understand the naturalness (or lack thereof) of linear algebra objects, we understand how to derive formulas for many linear algebra phenomena. The same applies to the specific case of FEs. Those familiar with the issues of transforming vectors, tensors, and tensor fields can consult the table of notation (table 3.1) and move on.

Notation	Definition
$\mathcal{F}(A, V)$	Space of functions from a set A to another set V, often a vector space.
$P(A)$	The space of real valued polynomials on the set A.
$P_r(A)$	The space of real valued polynomials of degree at most r on the set A.
$C^k(A)$	The space of k times continuously differentiable real valued functions from a set A. For $k = 0$, this is the set of continuous functions.
$L^1(A)$	The space of integrable real valued functions from a set A.
$\mathcal{C}(A, V)$ or $\mathcal{C}V(A)$	Given a class of scalar functions $\mathcal{C}$, a set A, and a finite dimensional vector space V, this denotes a subspace of $\mathcal{F}(A, V)$ where every scalar coordinate function is a member of $\mathcal{C}(A)$. Several examples are below.
$C^k(A, V)$	The space of k times continuously differentiable functions from a set A to another set V, often a vector space.
$P(A, V)$	The space of polynomial functions from A to V. When V is a vector space, the interpretation is that each coordinate of the function is a scalar polynomial.
$L(V, W)$	Space of linear functions from a vector space V to a vector space W.
$L(V_0, \dots, V_n; W)$	Space of multi-linear functions from vector spaces $V_0, \dots, V_n$ to vector space W.
$V^\star := L(V, \mathbb{R})$	The dual space of V. We call V the primal space to the dual space $V^\star$.
$\mathcal{T}_x N$	The tangent space at a point x on a surface N.
$V_1 \otimes \cdots \otimes V_N$	The tensor product of N vector spaces.
$V^{p,q}$	The tensor product of p copies of $V^\star$ and q copies of V.
$N^{k,p,q}$	The space $C^k(N, (\mathcal{T}N)^{p,q})$, i.e., the space of k times continuously differentiable functions from the surface N to (p, q) tensors of the tangent space.

Table 3.1: Vector Spaces Used in This Paper

3.1.1 Transforming Vectors

To understand the naturalness of vectors, we need to understand a few example spaces. In particular, we ask: how do we write down vectors in this space? Vector spaces are typically introduced via the prototypical example $\mathbb{R}^n$. The space $\mathbb{R}^n$ lets us write down a vector as a list of numbers. This is just the definition of $\mathbb{R}^n$, though it is easy to conflate this with the standard Euclidean basis. Given a point x on the surfaces of the sphere S^2, we can consider the tangent space $\mathcal{T}_x S^2$ the set of vectors tangent to the surface at x. A vector in the tangent space can be described as a list of numbers only after a choice of basis. Alternatively, without a choice of basis or any information about the tangent space, we can refer to vectors in the tangent space by taking the derivative of scalar functions at a point x. Most vector spaces that concern us are like the tangent space rather than Euclidean space: we cannot write down vectors with lists of numbers without some contortion. Since lists of numbers otherwise simplify writing down vectors and doing calculations (using index notation), we must ask: how can we sensibly write down vectors and do calculations in such vector spaces?

We can identify vectors that are easy to write down by considering a few more abstract spaces. Given an arbitrary set A and a vector space V, the set of functions $f\colon A \to V$, denoted $\mathcal{F}(A, V)$, can be made into a vector space via pointwise extension of vector addition and scaling. A useful specialization of $\mathcal{F}(A, V)$ occurs when A is a vector space; we define $L(A, V)$ as the space of linear functions from A to V. We need the case $A^\star := L(A, \mathbb{R})$, the dual space of a vector space A. Usually, we can more easily write down elements in the dual space. For example, for any vector space $S \subset \mathcal{F}(A, \mathbb{R})$ and $a \in A$, we can write down $F \in S^\star$ as $F(f) = f(a)$ without any reference to a basis. Moreover, we can use vectors such as F to make bases, which allow us to use lists of numbers to describe vectors while maintaining natural description of the vector (insofar as the vectors like F feel natural to us). We will now formalize this notion and introduce one more, dagger maps. Dagger maps are the essence of basis-dependent computations that are not invariant under arbitrary change of basis.

Definition 3.1.1 (Primal and Dual Basis, Dagger)**.** Given a vector space V with a basis $\{v_i\}$, we say that $\{v'_j\}$, a basis for $V^\star$ is a **dual basis** for $\{v_i\}$ if $v'_j(v_i) = \delta_{ij}$. We call $\{v_i\}$ the **primal basis** to $\{v'_j\}$. Together, $\{v_i\}$ and $\{v'_j\}$ are a **primal-dual basis pair**. A primal dual basis pair defines a **dagger isomorphism** between V and $V^\star$:

$$(v_i)^\dagger = v'_i. \tag{3.1}$$

We denote the inverse of the dagger mapping as $(\cdot)^{-\dagger}$. When necessary, we can subscript the dagger mapping with a primal or dual or primal dual pair to indicate its origin. Since dual or primal bases uniquely determine each other, either part of the pair identifies the mapping.

The dagger map is critical to many calculations. Given a matrix A_{ij}, we know that it can be associated to a map $(y_i, x_j) \mapsto y_i A_{ij} x_j$ and a map $(x_j) \mapsto A_{ij} x_j$. However, finding an equivalent matrix for such maps is not so simple because the latter map requires a dagger map. Given a linear transform $A \in L(V, W)$ and bases $\{v_j\}$ and $\{w_i\}$, then the corresponding matrix is given by a formula involving the dagger: $A_{ij} = (w_i)^\dagger(A(v_j))$. In contrast, given a **bilinear map** from V and W to the real numbers (denoted $A \in L(V, W; \mathbb{R})$), then we don't need a basis-dependent dagger map to write $A_{ij} = A(v_j, w_i)$. The consequence of this contrast is that the formula for a matrix vector product has one more sensitivity to the bases involved that the formula for bilinear map does not. (In differential geometry, we often see this with the presence of metric terms.) To show how transformations might preserve the dagger map and how vectors represented with dual vectors behave under transformation, we study how a transformation on V behaves on $V^\star$ with adjoints.

Definition 3.1.2 (Transformation, Inverse, Adjoint, Adjoint-Inverse, System of transformations)**.** Suppose V and W are dual spaces. We say that $T \in L(V, W)$ is a **transformation** if it is an isomorphism. Then we can define $T^{-1} \in L(W, V)$, the inverse transformation. For any $T \in L(V, W)$, we can also define a map, **the adjoint**, $T^\star \in L(W^\star, V^\star)$ via

$$T^\star(w') = v \mapsto w'(T(v)).$$

If T is an isomorphism, then $T^\star$ is. Finally, when T is an isomorphism, then $(T^\star)^{-1} = (T^{-1})^\star$ so we can denote $(T^\star)^{-1}$ as $T^{-\star} \in L(V^\star, W^\star)$, which we call the **adjoint inverse**. We note that V and W are finite dimensional so that $V^{\star\star} \cong V$ and $W^{\star\star} \cong W$, then any of $T, T^{-1}, T^\star, T^{-\star}$ defines the other three. Thus any one identifies the other three as a **system of transformations**.

Now, we can easily show that if two primal dual basis pairs are related via a transformation, then the dagger mapper will behave well under transformation.

Proposition 3.1.3. *Suppose V and W are finite-dimensional vector spaces. Suppose we have a system of transformations $T, T^{-1}, T^\star, T^{-\star}$. Suppose we have a primal dual basis pair for V, $\{v_i\}$ and $\{v_i'\}$. Then $\{Tv_i\}$ and $\{T^{-\star}v_i'\}$ are a primal dual basis pair for W. Further, we have that for all $v \in V$*

$$T^\star(T(v)^\dagger) = v^\dagger. \tag{3.2}$$

Though this result is simple, we cannot say much more with just abstract linear algebra. We explore a pragmatic example, which happens to be a microcosm for some issues that we will encounter in transforming elements.

Suppose we are given two triangles, K and K' related by a mapping $T \in C^1(K, K')$. Given a point on an edge labeled with $\Box$, we have tangent vector t and normal vector n. These are defined via the property that $||t|| = ||n|| = 1$ and $t \cdot n = 0$. Similarly, at the point $T(\Box)$, we have a tangent t' and a normal n' similarly defined. We depict the situation in fig. 3.1. These definitions feel simple and natural, so we ask how the definitions behave under transformation. We ask: can we come up with a formula for n' with $DT_\Box$, $DT_\Box^{-\star}$, t, and n? Similarly, we want to know what type of formula are available (e.g., linear or non-linear in each of the four inputs).

To find a formula, we note that both sets of normal and tangent vectors define bases. Further, there are associated dual bases $\{t^\dagger, n^\dagger\}$ and $\{(t')^\dagger, (n')^\dagger\}$. Unfortunately, unless $DT_\square$ is a rotation map, we know that the primal dual pair at K is not mapped to the primal dual pair of K'. With just abstract linear algebra, we do not see a coordinate independent linear formula for n' via the transformations, t, and n. Without geometric information, the best we can say is that while we do not have that $DT_\square n \cdot DT_\square t \neq 0$, we do get that $DT_\square^{-\star} n^\dagger (DT_\square t) = 0$. Thus, we are limited in our ability to write down n' in terms of information in K unless we use geometric information. In particular, since geometrically $DT_\square(\operatorname{span}\{t\}) = \operatorname{span}\{t'\}$, we already have the direction of t' from just $DT_\square$ and t. Thus, an appropriate way to recover the direction of the normal vector in this scenario is to set $n'' = (DT^{-\star}(n^\dagger))^{-\dagger}$. In particular, we have that $n'' \cdot t' = 0$. So with geometric information, we can write down a linear relation for a vector in the same direction of n' using just t, n, and $DT^{-\star}$. Finally, to recover the size of n', that $|n'| = 1$, we must non-linearly normalize n''. Linearly in each argument with geometric information, the best that we have is $DT_\square^{-\star}(\operatorname{span}\{n^\dagger\})^{-\dagger} = \operatorname{span}\{n'\}$.

The recapitulate the above: we wrote down certain vectors in two contexts related by transformation. Then we hoped to find nice (linear, polynomial) formulas for one vector in terms of the others and transformations. We found that the transformations did not preserve the precise relationship between the vectors, but we could get closer to writing down the desired formula by using that the transformations did preserve the relationships between primal and dual bases. With geometric information, we get that some information in t is transformed into that of t', mainly the space spanned by it. In this sense, there is something natural about the definition of t. With this and the transformation of primal-dual bases, we could recover the direction of the normal vector, the space spanned by it, with a formula that is linear in the transformations and the vectors n and t. A complete formula for n' in terms of information in K is necessarily non-linear. The lesson is that specifying an abstract vector or vector subspace in a coordinate independent manner is difficult without additional context and that, without that context, nice linear formulas for the vector are unlikely to behave well under change of basis. In contrast, tensor spaces provide a rich family of vectors and vector subspaces that can be written down independently of even the underlying vector spaces. In essence, tensor spaces provide many analogues of t and t' where there is a bespoke relationship under the transformation. With enough machinery, we can elegantly capture our efforts to write down n' via anti-symmetric tensors and the Hodge star operator.

3.1.2 Transforming Tensors

We start with an algebraic definition of a tensor product as an operation on vectors and vector spaces.

Definition 3.1.4 (Tensor Product)**.** Suppose we have N vector spaces $V_0, \ldots, V_{N-1}$. An N-ary operator $\otimes$ and a vector space $Z := V_0 \otimes \cdots \otimes V_{N-1}$ are a tensor product and tensor product space if

1. $\otimes \in L(V_0, \ldots, V_{N-1}; Z)$,
2. $\otimes$ is associative,
3. and if for each i, $\{v_{ij}\}$ is a basis for V_i, then $\{v_{0i_0} \otimes \cdots \otimes V_{N-1i_{N-1}}\}$ is a basis for Z.

One can think of this as a formal operation and simply compute algebraically with it. For example, $e_i \otimes e_j + e_k \otimes e_j = (e_i + e_k) \otimes e_j$. These manipulations utilize the first two properties of the tensor product, but the depth of the product comes from the third property. Simply, this says that any linear map $T \in L(V_1 \otimes \cdots \otimes V_N, W)$ can be described by its action on elements of the form $(v_{i_1} \otimes \cdots \otimes v_{i_N})$. This yields an isomorphism that has surprising amounts of depth:

Proposition 3.1.5. *Suppose we have N finite-dimensional vector spaces $V_0, \ldots, V_{N-1}$ and*

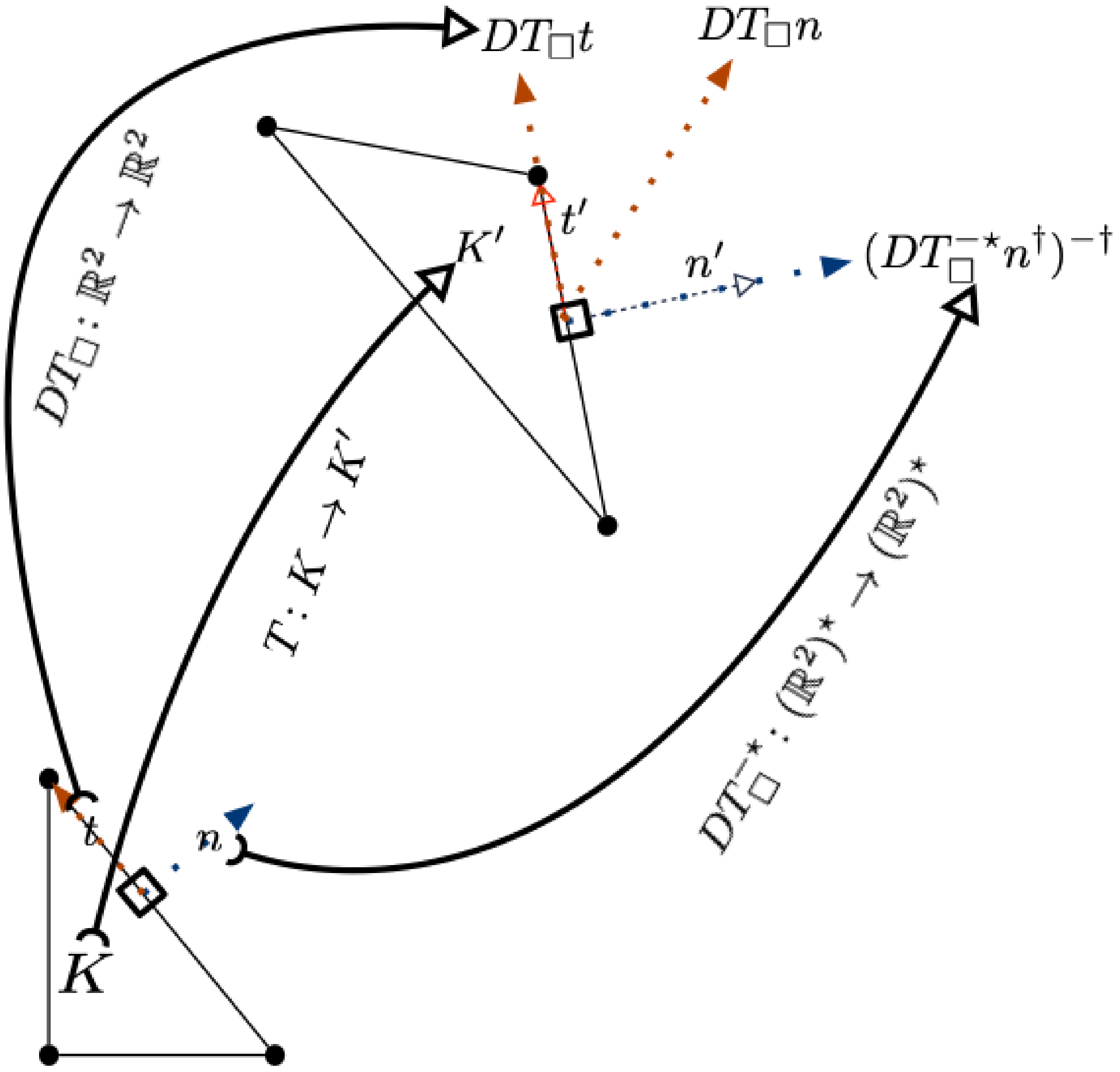

$DT_\square t$
$DT_\square n$
t'
K'
n'
$(DT_\square^{-\star} n^\dagger)^{-\dagger}$
$DT_\square : \mathbb{R}^2 \to \mathbb{R}^2$
$T : K \to K'$
$DT_\square^{-\star} : (\mathbb{R}^2)^\star \to (\mathbb{R}^2)^\star$
t
n
K

Figure 3.1: Suppose we are given two triangles, K and K' related by a mapping $T \in C^1(K, K')$. Given a point on an edge labeled with $\square$, we have tangent vector t and normal vector n. These are defined via the property that $||t|| = ||n|| = 1$ and $t \cdot n = 0$. Similarly, at the point $T(\square)$, we have a tangent t' and a normal n' similarly defined. Both of these define bases and there are associated dual bases $\{t^\dagger, n^\dagger\}$ and $\{(t')^\dagger, (n')^\dagger\}$. We use this situation to study the transformation of vectors.

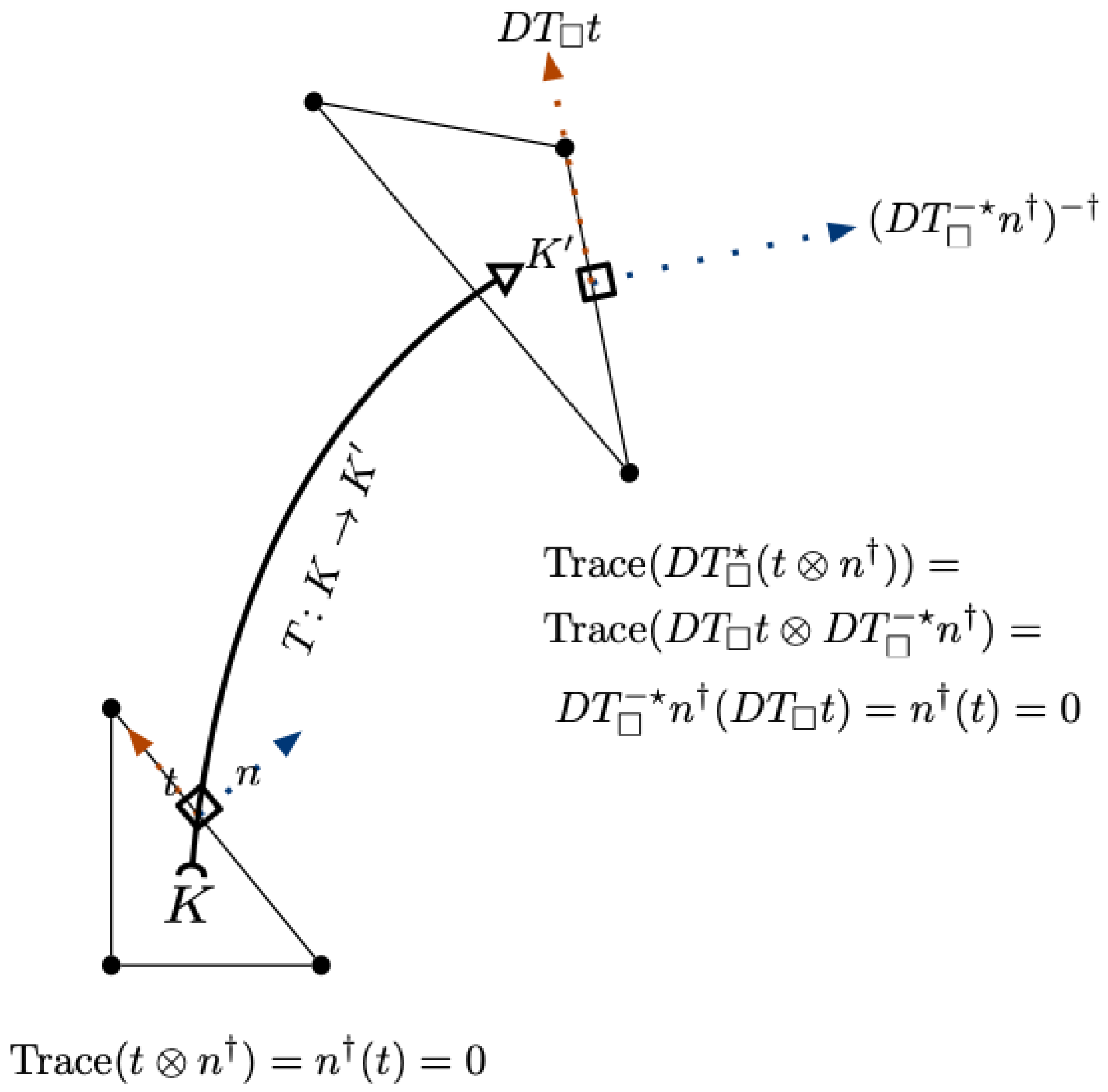

$DT_{\square}t$
$(DT_{\square}^{-\star}n^{\dagger})^{-\dagger}$
K'
$T\colon K \to K'$
$\mathrm{Trace}(DT_{\square}^{\star}(t \otimes n^{\dagger})) =$
$\mathrm{Trace}(DT_{\square}t \otimes DT_{\square}^{-\star}n^{\dagger}) =$
$DT_{\square}^{-\star}n^{\dagger}(DT_{\square}t) = n^{\dagger}(t) = 0$
t
n
$\hat{K}$
$\mathrm{Trace}(t \otimes n^{\dagger}) = n^{\dagger}(t) = 0$

Figure 3.2: Transforming tensors to preserve certain properties like trace is much simpler than preserving vectors.

a vector space W. Then $L(V_0, \cdots, V_{N-1}, W) \cong V_0^\star \otimes \cdots \otimes V_{N-1}^\star \otimes W$ via the identification:

$$f(v_0' \otimes \cdots \otimes v_{N-1}' \otimes w) = (v_0, \ldots, v_{N-1}) \mapsto w \prod_{i=0}^{N-1} (v_i'(v_i)). \tag{3.3}$$

What is interesting about this isomorphism is that we write it down using property three of tensor products but without reference in the formula to a choice of basis for each V_i. This hints at a class of vectors that can be defined on many tensor spaces simultaneously and that are preserved under certain transformations, called pullbacks. We define pullbacks for a certain class of tensor spaces to avoid needless notation.

Definition 3.1.6 ($V^{p,q}$ Tensor Space, Pullback)**.** Suppose V is a finite-dimensional vector space. Then $V^{p,q} = \otimes_{i=1}^{p} V^\star \otimes \otimes_{j=1}^{q} V$ is a **tensor space**. Suppose we have another finite-dimensional space W and a transformation $T \in L(V, W)$. The pullback can be defined using property three of tensor products as a mapping in $L(V^{p,q}, W^{p,q})$ given by

$$\begin{gathered}(T^\star)(v_0' \otimes \cdots v_{p-1}' \otimes v_0 \otimes \cdots \otimes v_{q-1}) = \\ T^{-\star}(v_0') \otimes \cdots \otimes T^{-\star}(v_{p-1}') \otimes T(v_0) \otimes \cdots \otimes T(v_{q-1}).\end{gathered} \tag{3.4}$$

Compared to vectors, pullbacks in general preserve a greater variety of subspaces of tensor spaces. In particular, we can specify a preserved subspace as the zero set of a collection of linear operators that are defined without reference to a particular basis.

Proposition 3.1.7. *Let V and W be a finite-dimensional vector spaces with an isomorphism $T \in L(V, W)$ and let $q, p \geq 0$ with $p + q > 0$. Pick $q' \geq 0$ and $p' \geq 0$ such that $q \geq q'$ and $p \geq p'$ and $q - q' = p - p'$. Suppose we have two functions $\alpha \colon S^{q'} \to \{0, 1, -1\}$ and $\beta \colon S^{p'} \to \{0, 1, -1\}$. Suppose we have an element $\gamma \in S^{q-q'}$. Let $K_W \in L(W^{p,q}, W^{q',p'})$ and*

$K_V \in L(V^{p,q}, V^{q',p'})$ be both be given by a formula:

$$\begin{aligned}
&K_H(h'_1, \dots, h'_{q'}, h'_{q'+1}, \dots h'_q, h_1, \dots, h_{p'}, h_{p'+1}, \dots, h_p) \\
&= (\prod_{j=1}^{q-q'} h'_{q'+\gamma(j)}(h_{p'+j}))(\sum_{\sigma \in S_{q'}} \alpha(\sigma) h'_{\sigma(1)} \otimes \cdots \otimes h'_{\sigma(q)'}) \\
&\qquad \bigotimes (\sum_{\sigma' \in S_{p'}} \beta(\sigma') h_{\sigma'(1)} \otimes \cdots \otimes h_{\sigma'(p')}).
\end{aligned} \tag{3.5}$$

Consider the subspace $V_K = \ker K_V$ and $W_K = \ker K_W$. We have that $T^\star|_{V_K} \in L(V_K, W_K)$ is an isomorphism.

For the cases $V^{0,2} \cong V^{2,0} \cong V^{1,1} \cong L(V, V)$, the above proposition states that symmetric, anti-symmetric, and trace-free matrices are preserved under change of basis. Further, this is a much richer space of subspaces preserved under transformation than the situation depicted by fig. 3.1, where we need information outside of linear algebra to show that the subspace was preserved (contrast with fig. 3.2) . Similarly to our analysis of fig. 3.1, we would also not expect pullbacks to preserve sizes of tensors within preserved subspaces.

3.1.3 Transforming Tensor Fields

We now advance one step closer to transforming FEs by transforming sufficiently smooth functions and dual functions on certain spaces of smooth functions. To define the specific spaces that we will transform and the transformation, we need some notation. Specifically, we consider the derivative of a transformation $T : N \to M$ where N and M are surfaces.[1] Specifically, for every $x \in N$, we recall that the derivative is a map between tangent spaces $DT_x \in L(\mathcal{T}_x N, \mathcal{T}_x M)$. To work with functions that resemble the derivatives, we will suppress the dependence on x, allowing us to write a white lie: $DT \in \mathcal{F}(N, L(\mathcal{T}N, \mathcal{T}M))$. We can now define the space functions that we will transform.

Definition 3.1.8 (Tensor Fields)**.** Let N be a surface and $p, q \geq 0$. Let $N^{k,p,q} := C^k(N, (\mathcal{T}N)^{p,q})$

[1]We wish to de-emphasize manifolds as a concept here.

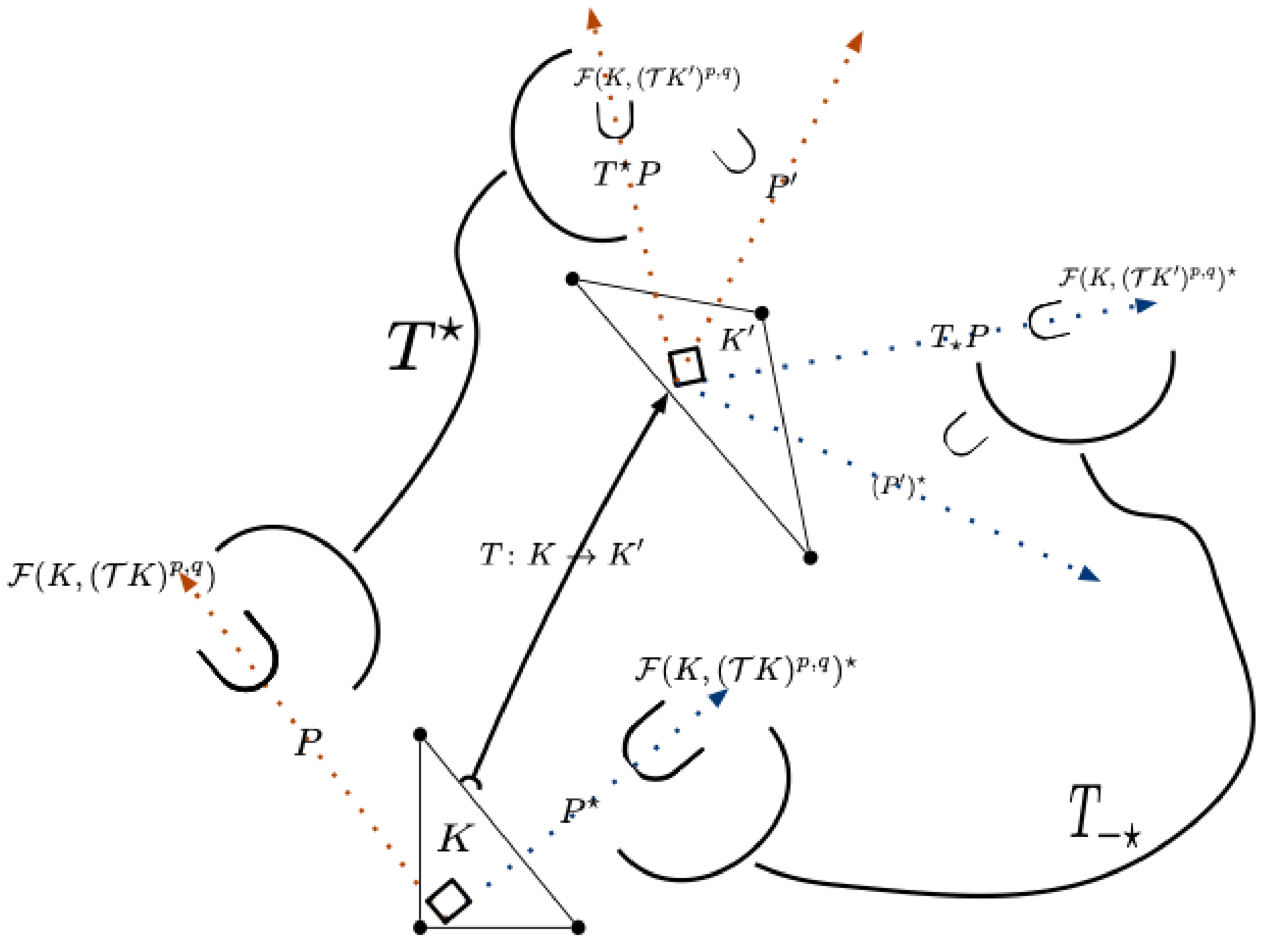

$\mathcal{F}(K,(\mathcal{T}K')^{p,q})$
$\cup$
$T^\star P$
$\cup$
P'
$T^\star$
K'
$\mathcal{F}(K,(\mathcal{T}K')^{p,q})^\star$
$T_\star P$
$\subset$
$\cup$
$(P')^\star$
$T\colon K \to K'$
$\mathcal{F}(K,(\mathcal{T}K)^{p,q})$
$\cup$
P
$\mathcal{F}(K,(\mathcal{T}K)^{p,q})^\star$
$\cup$
$P^\star$
K
$T_{-\star}$

Figure 3.3: A more abstract analogue of fig. 3.1 for spaces of functions. The pushforward and pullback map a finite-dimensional subspace P and its dual space to spaces with the same duality relationship, depicted by the right angle. However, we often hope that P would be mapped to an analogous space P' and similarly for the dual. But there is no reason for this to be the case generically, especially when we do not use the geometric information that was present in fig. 3.1.

be the space of tensor fields. Let $N^{p,q} := \mathcal{F}(N, (\mathcal{T}N)^{p,q}))$ be the space of tensor-valued functions.

Tensor spaces have one unified transformation, the pullback, in contrast to vector spaces, which have separate transforms for the primal and dual space. Tensor fields return to the vector field model via the definition of the pullback (transformation) and pushforward (adjoint transformation).

Definition 3.1.9 (Pullback and Pushforward)**.** Let N and M be surfaces. Let $T \in C^1(N, M)$ have an inverse $T^{-1} \in C^1(M, N)$. The pullback along T is a mapping in $L(N^{p,q}, M^{p,q})$ given by

$$(T^\star f)(x) = (DT_x)^\star(f(T^{-1}(x)) \text{ for all } f \in N^{p,q}. \tag{3.6}$$

The pushforward along T is a mapping in $L((M^{p,q})^\star, (N^{p,q})^\star)$ given by

$$(T_\star \phi)(f) = \phi(T^\star f) \text{ for all } \phi \in (M^{p,q})^\star \text{ and } f \in N^{p,q}. \tag{3.7}$$

As with the inverse adjoint, we have that $(T_\star)^{-1} = (T_\star^{-1})$ as mappings in $L((N^{p,q})^\star, (M^{p,q})^\star))$ so we use the notation $T_{-\star}$ for $(T^{-1})_\star$ and similarly $T^{-\star}$ for $(T^{-1})^\star$.

An even richer and more complex situation exists for preservation of vectors, subspaces, and properties under pullback and pushforward. We consider three aspects of the situation. First, we also note that something close to proposition 3.1.3 is true though these spaces are not finite dimensional: we still have that if $f \in N^{p,q}$ and $\phi \in (N^{p,q})^\star$, then $(T_{-\star}\phi)(T^\star f) = \phi(f)$. Thus, duality relationships between sets of functions and dual functions are still preserved. Second, we note that proposition 3.1.7 can be directly lifted to the context of pullbacks and pushforwards. Thus, subspaces identified via pointwise extensions of natural tensor subspaces behave similarly under pullback. Third, ignoring the the tensor aspect, we can see that the properties defining a finite-dimensional subspace of functions are often not preserved. To see this, we consider a simple example. Suppose we are given two triangles,

K and K', related by a mapping $T \in C^1(K, K')$. On both triangles, we can consider polynomials that are at most degree 2 in x and degree 1 in y. An algebraic calculation can show that this subspace is not generically preserved under pullback (consider $(x+y)^2$ which could occur via simple shear $T(x,y) = (x+y, y)$). In summary, the general situation is depicted in fig. 3.3: although we still have duality relationships and a lifted version of proposition 3.1.7, we cannot depend on pullbacks and pushforwards to match spaces of functions and dual functions that we define on different triangles, even if we define both pairs of space through seemingly natural means. In the next section, we will see the consequences of this for transforming FEs. For now, we briefly introduce exterior calculus and an application of proposition 3.1.7 to an interesting class of tensor fields, forms. With some additional machinery, we can even characterize a collection of finite-dimensional subspaces of tensor-valued polynomial spaces that are preserved under pullback by affine functions.

3.1.4 Interlude on Exterior Calculus

To handle several geometric issues, most notably integration on volumes and surfaces, we utilize exterior calculus. Additionally, exterior calculus allows us to exercise our knowledge of pullbacks. Finally, exterior calculus allows us to understand a result about a class of spaces preserved by pullback of affine functions. This will later allow us to understand the FEEC. We start by briefly defining forms and their connection to integration.

Definition 3.1.10 (Exterior Power, Forms)**.** Suppose V is a finite-dimensional vector space. We can define the symmetrizer of $V^{0,k}$ or $V^{k,0}$ as

$$\mathrm{Sym}^k(v_1 \otimes \cdots \otimes v_k) = \frac{1}{k!} \sum_{\sigma \in S^k} v_{\sigma(1)} \otimes \cdots \otimes v_{\sigma(k)}. \tag{3.8}$$

The kth **exterior power** of V is $\ker \mathrm{Sym}^k$ and it is denoted $\Lambda^k V$. The space of k **forms** on a surface N is the space $C^\infty(N, \Lambda^k(\mathcal{T}N^\star))$, which is often denoted $C^\infty \Lambda^k N$.

If a surface is k dimensional (the tangent space is a k-dimensional space), then we can

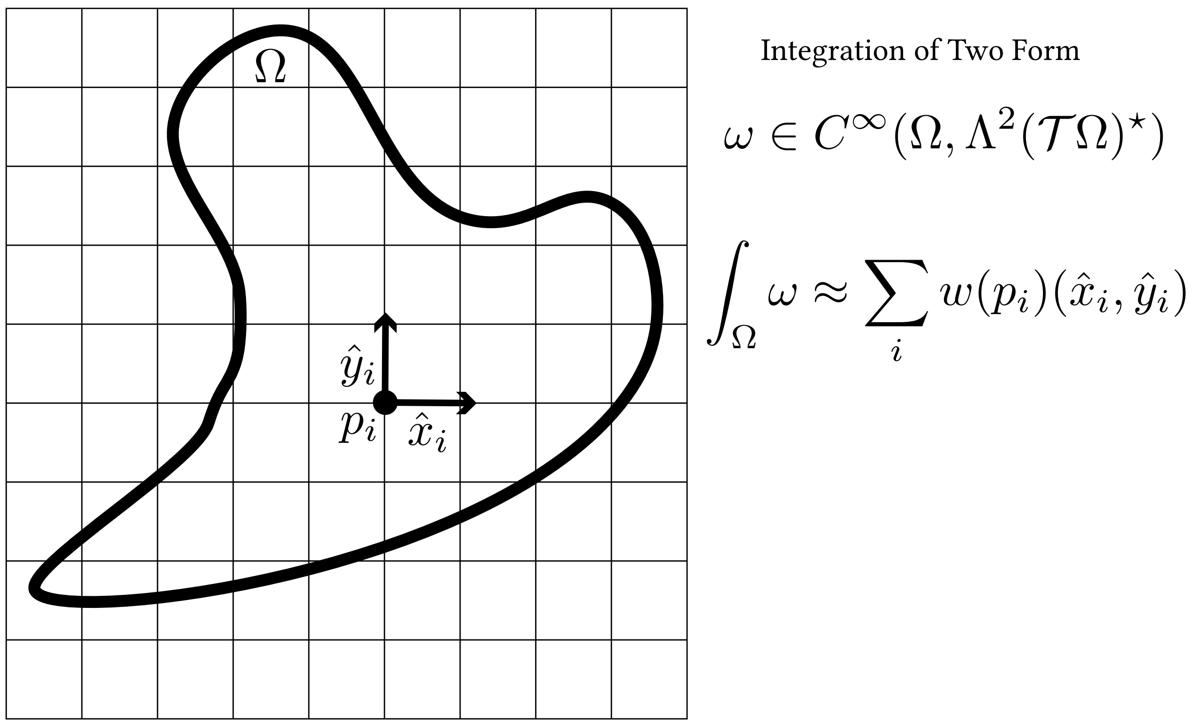

Integration of Two Form
$\omega \in C^{\infty}(\Omega, \Lambda^2(\mathcal{T}\Omega)^\star)$
$\int_\Omega \omega \approx \sum_i w(p_i)(\hat{x}_i, \hat{y}_i)$
Ω
$\hat{y}_i$
p_i
$\hat{x}_i$

Figure 3.4: A rough visual definition of integration of differential forms. Forms locally consume tangent vectors spanning a local area to produce area measurements. This figure was modeled after Keenan Crane's lectures on exterior calculus.

integrate k forms on it. A sketch of this definition of integration is depicted in fig. 3.4. In essence, anti-symmetric objects locally capture the area of parallelepipeds and forms allow us to define integrals by summing up all the local areas. The space of forms is preserved under pullback via an application of proposition 3.1.7 with the operator Sym^k. This leads to a key property of the pullback:

Proposition 3.1.11. *Suppose M and N are manifolds of dimension k and $T \in C^1(M, N)$ has an inverse $T^{-1} \in C^1(N, M)$. Suppose $w \in L^1(M, \Lambda^k(\mathcal{T}M)^\star)$. Then*

$$\int_M w = \int_N T^\star w. \tag{3.9}$$

A similar property is that two other operations on forms commute with pullbacks in some cases. Derivatives commute against pullback with sufficiently smooth functions. A class of formal anti-derivatives, called Koszul derivatives, commutes only against affine transformations. We define both of these:

Definition 3.1.12 (Exterior and Koszul Derivative)**.** Let N be a surface. Consider the space of k forms $C^k(N, \Lambda^k V^\star)$. The exterior derivative of a $k+1$ form is simply the anti-symmetrization of the derivative:

$$dw := P_{\ker \operatorname{Sym}^{k+1}}(Dw) \text{ for all } w \in C^k(N, \Lambda^k V^\star). \tag{3.10}$$

If $N \subset \mathbb{R}^N$, we can define a vector field $X \colon N \to \mathcal{T}N$ so that $X(x)$ points in the direction of x and satisfies $||X(x)|| = 1$. Then the Koszul derivative is a $k-1$ form given by

$$\kappa w(x)(t_1, \ldots, t_{k-1}) := w(x)(X(x), t_1, \ldots, t_{k-1}) \text{ for all } w \in C^k(N, \Lambda^k V^\star). \tag{3.11}$$

The Koszul derivative is probably new to most readers, but a simple intuition exists for it to those familiar with forms already: the Koszul derivative is the interior product of a form with the vector x at every point x. We suggest several references for translations of exterior calculus to the language of vector fields [24], [26]–[30]. By translating the interior product to the language of vector fields, one can consider several examples of the Koszul derivative. For example, for a 1 form in $\mathbb{R}^3$, the Koszul derivative would amount to taking a vector field $V \colon \mathbb{R}^3 \to \mathbb{R}^3$ and defining a scalar field $f(x) = V(x) \cdot x$. The exterior and Koszul derivatives allow a more sophisticated statement about the types of spaces preserved under pullback by affine functions:

Proposition 3.1.13 (See Section 3.4 in [26])**.** *Consider $P(M, \Lambda^k(\mathcal{T}M)^\star)$, the space of polynomial differential forms. The following spaces characterizes all subspaces preserved under pullback by affine function:*

$$X(r, s, k, M) = dP_{r+1}(M, \Lambda^{k-1}(\mathcal{T}M)^\star) + \kappa P_s(M, \Lambda^{k+1}(\mathcal{T}M)^\star)$$

Suppose we have a mapping A that maps surfaces M to subspaces of $P(M, \Lambda^k(\mathcal{T}M)^\star)$. Then $A(M) = X(r, s, k, M)$ if and only if for all affine transformations $T\colon N \to M$, $T^\star(A(M)) = A(N)$.

3.2 The Mathematical Definition of a Finite Element

Before providing the definition of a FE and illuminating the resulting computational issues, especially those relating to transforming elements, we recast the linear Lagrange element on a triangle. The two key properties of the linear Lagrange basis functions are:

1. The basis functions f_0, f_1, f_2 are affine functions on the triangle, i.e., $f_i \in P(K)$ the *vector space* of affine functions defined on a triangle K.

2. Each basis function corresponds to a triangle vertex, v_0, v_1, v_2, so that if $f = \alpha_0 f_0 + \alpha_1 f_1 + \alpha_2 f_2$, then $f(v_i) = \alpha_i$.

We can recast this definition by means of a dual basis. This may seem circuitous. Why not just define a primal basis, the functions f_0, f_1, f_2, directly? Consider our triangle, K and the linear Lagrange basis functions again. Depending on the coordinates of our triangle, f_0, f_1, f_2 may have all different sorts of arbitrary coefficients. However, consider the following *dual* basis functions: $\phi_0(f) = f(v_0), \phi_1(f) = f(v_1), \phi_2(f) = f(v_2)$. These these dual basis functions just say "the value at each of the vertices," which is a simple, coordinate-independent concept. Finding the coefficients of the primal basis functions then reduces to solving n linear systems:

$$\phi_i(f'_j) f_{jk} = \delta_{ik}. \tag{3.12}$$

where f'_0, f'_1, f'_2 is some arbitrary basis for the function space (e.g., monomials or Legendre polynomials). Note as a consequence of eq. (3.12), that $f_k = \sum_j f'_j f_{jk}$. Some readers may recognize the matrix $\phi_i(f'_j)$ as a generalized Vandermonde matrix [31].

The key step in the above exercise is that we no longer write down the primal basis

f_0, f_1, f_2 explicitly, but we instead write down the dual basis. This has a few key advantages: the mathematical theory works mostly on the dual basis, the dual basis functions make the interaction between neighboring triangles easier to reason about, interpolation methods come for free via the dual basis, and the dual basis is easier to write down. Thus the mathematical definition centers the dual basis:

Definition 3.2.1 (Finite Element)**.** A **finite element** (K, V, P, Σ) consists of

1. $K \subseteq \mathbb{R}^n$, the domain of the element (e.g. a triangle, quad, hex, tetrahedron),
2. V a vector space of values (e.g., real numbers $\mathbb{R}$ or vectors $\mathbb{R}^3$)
3. $P(K)$ is a finite dimensional vector space of functions from K to V (usually some class of polynomial functions)
4. $\Sigma(K)$ is a dual basis for $P(K)$ (e.g., point evaluations)

This definition dates back to Ciarlet [1] and variations of it have been reproduced in many FEM textbooks [3], [14], [32], [33]. Typically a diagram is used to depict a given FE, representing various dual basis functions via glyphs (such as dots and circles) as we do in fig. 3.5.

These diagrams raise a critical issue in the definition: we have instantiated this FE on a particular set K, but our definition is coordinate free and independent of K. Moreover, as practitioners will have anticipated, we want to instantiate a FE on arbitrary K. And if we simply use the recipe of using eq. (3.12) to find basis functions for many different K, we suffer from cubic scaling in solutions to eq. (3.12), not to mention the sub-optimal performance qualities of linear solvers. To avoid this issue, we observe that a single formula, dependent on K, might capture many different related FEs on different sets K. To see how this might work, we return to the linear Lagrange element on a triangle.

Let K_0 be the unit triangle and K_1 be some arbitrary non-degenerate triangle, related to K_0 via an affine map[2] $T\colon K_0 \to K_1$. Suppose we are given two finite elements, $(K_i, \mathbb{R}, P_i, \Sigma_i)$

[2]This could be a more general diffeomorphism instead.

for $i = 0, 1$ where P_i are the affine functions on K_i and Σ_i Let p_0, p_1, p_2 be the primal basis corresponding to $\Sigma(K_0)$. With the formula for Lagrange interpolating polynomials on a triangle, we can verify that if $T\colon K_0 \to K_1$ is a diffeomorphism and q_0, q_1, q_2 is the primal basis corresponding to $\Sigma(K_1)$, then $q_i = p_i \circ T^{-1}$. In particular, if we consider the pullback map on scalar functions $T^\star\colon P(K_0) \to P(K_1) = f \mapsto f \circ T^{-1}$, we can see that it is linear and its matrix representation in the bases p_0, p_1, p_2 and q_0, q_1, q_2 is the identity matrix, i.e.,

$$c_{ij} T^\star(p_j) = q_i \text{ iff } c_{ij} = \delta_{ij}. \tag{3.13}$$

The consequence of eq. (3.13) is that any computation on the primal basis corresponding to an arbitrary K_1 can be done in terms of p_i and T, avoiding the need to actually find q_i for each cell. This possibility is referred to as **affine equivalence** [1]:

Definition 3.2.2 (Affine Equivalence for Scalar-Valued Elements)**.** Given two finite elements: $(K_i, \mathbb{R}, P_i, \Sigma_i)$ for $i = 0, 1$ are said to be affine equivalent if whenever there is a affine map T between K_0 and K_1, then $T^\star = f \mapsto f \circ T^{-1}$ maps the primal basis corresponding to Σ_0 to the primal basis corresponding to Σ_1.

However, many FEs are not affine equivalent, and so other notions of equivalence have been developed, including affine-interpolation equivalence for the Hermite element [1] and compatible nodal completion equivalent for the Morley and Argyris elements [34]. We do not deeply investigate these notions here, but the key idea is that the matrix representation of $T^\star$, the c_{ij} in eq. (3.13), is not the identity matrix but now takes some other forms, possibly non-linearly dependent on the map T. In particular, for the Morley element, the matrix representation of the pullback map was computed by [34] as follows:

Example 3.2.3 (Morley Pullback Matrix)**.** *Let K be a non-degenerate triangle and K' be the unit triangle. For each edge e_i of K and each edge e_i' of K', we can define the edge tangents and normals t, t' and n, n'. Each edge has a midpoint m_i and length l_i. Then given* *the map $T\colon K \to K'$, we can define*

$$B^i = \begin{bmatrix} t & n \end{bmatrix} DT^T(m_i) \begin{bmatrix} t' \\ n' \end{bmatrix}.$$

Then the matrix would be roughly:

$$c = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -B^1_{12}/l_1 & B^1_{12}/l_1 & B^1_{11} & 0 & 0 \\ -B^2_{12}/l_2 & B^2_{12}/l_2 & 0 & 0 & B^2_{11} & 0 \\ B^3_{12}/l_3 & 0 & -B^3_{12}/l_3 & 0 & 0 & B^3_{11} \end{bmatrix} \tag{3.14}$$

This matrix c from eq. (3.14), the matrix representation of the pullback map $T^\star$, again lets us compute with the Morley element basis functions on an arbitrary triangle by only computing the matrix and the Morley element basis functions on a single reference triangle.

At present, these matrices are computed by hand, as in [9], [34]–[37]. One of the core aspects of our system is that we will compute these automatically from a software specification of definition 3.2.1. Naturally, these are formulas that are available in these papers, but there are several virtues to automation. First, for an unknown element, a formula is not already available. Second, as we will describe in the next two sections, the formula is not enough. The formula must be integrated into a FE code, which is a finicky task that requires awareness of the various conventions and methods a given code employs. Third, the first two reasons compound: exploring new elements requires changing other aspects of the system in subtle ways, and changing the system might require one to revisit and change existing formulas. To facilitate rapid exploration of FEMs, we need to automate the discovery of the relevant formula in the context of an expressive system for other aspects of FEs.

With this motivation and background in hand, we can now frame the core problem that we will solve in chapter 6. We are interested in finding formulas for the primal basis functions of a class of FEs (K', P', V', Σ') so that the primal basis functions depend (linearly) on a fixed FE (K, P, V, Σ) and (non-linearly) on a transformation $T\colon K \to K'$. We can now formally define the object that we will compute in chapter 6.

Definition 3.2.4 (Transformation of a Finite Element)**.** Fix a finite element (K, V, P, Σ). Suppose we have a collection of sets $\{K'\}$ that are isomorphic to K via a C^1 mapping $T_{K'}$. Suppose further that we have a collection of finite elements $\{(K', V', P', \Sigma')\}$ indexed by K'. Let $\{f_i\}$ be the primal basis for (K, V, P, Σ). Let $\{f_i^{K'}\}$ be the primal basis for (K', V', P', Σ'). We seek a function $\mathrm{P}\colon \{T_{K'}\} \to L(\mathbb{R}^{\dim P}, \mathbb{R}^{\dim T^\star P'})$ such that

$$T_{K'}^\star f_i^{K'} = \sum_j \mathrm{P}(T)_{ij} f_j. \tag{3.15}$$

The matrix $\mathrm{P}(T)$ is the representation of the pullback as a linear map from the basis $\{f_i^{K'}\}$ to the basis $\{f_i\}$.

Ultimately our system will compute this from a software implementation of the mathematical definition of a FE. Based on our discussion of pullbacks, we can now anticipate some difficulties and the tools that we have to solve them. First, the above formulation assumes that $T_{K'}^\star f_i^{K'} \in P$, which in general does not need to be the case, as discussed in section 3.1.3 and depicted in fig. 3.3. Thus, P does not obviously exist in many cases, though there is hope in the ideas in section 3.1.2 and section 3.1.4, as they illustrate some cases for P where P exists. Second, even if it does exist, P might be a complex symbolic object, especially because pullbacks depend polynomially on derivatives and inverse matrices turn polynomials into large complex rational functions. Thus, we should seek to use the ideas of section 3.1.2 and section 3.1.1 to identify objects related to FEs that transform nicely, alleviating this problem. As noted in fig. 3.1, geometric information allows one to extract the direction of normals and tangents on another triangle up to normalization using the normal and tangent of a reference triangle and a transformation. We hint that many variants of the Lagrange element defined above resemble tangent vectors in this respect. We return to this in chapter 6 and for now move on to the second big challenge of using FEs: the global FE space and operations on it.

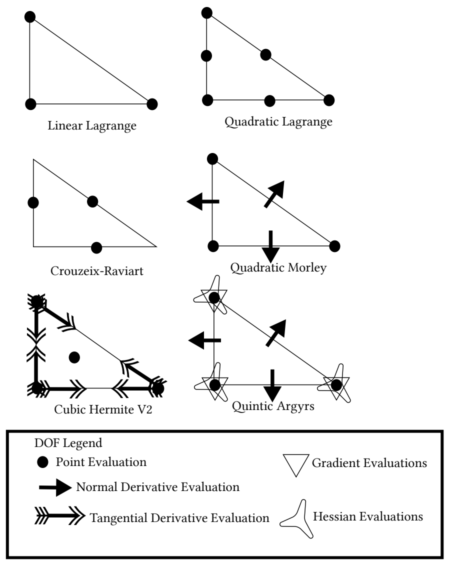
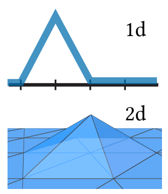


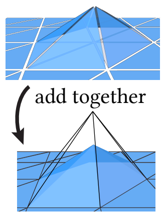
Figure 3.5: Examples of finite elements depicted graphically via their dual basis functions. This figure was inspired by graphical depictions going back to the Ciarlet definition of the finite element, but especially those in [14].

3.3 The Mathematical Definition of Global Finite Element Spaces and Associated Operations



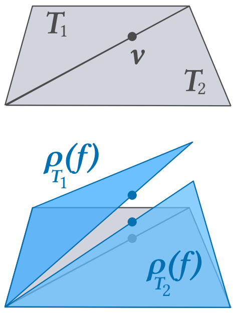
A FE defines a local space of functions on individual mesh elements. Now we seek to construct a global FE space, a space of functions defined on the whole (global) mesh. Like the individual FEs, this construction will consist of a domain, a space of functions, and a dual basis. Returning to the linear Lagrange example from section 3.2, the primal bases of the corresponding global space are the well-known linear hat functions, one for each vertex in the mesh.

A hat function associated to the vertex v matches a primal basis function in each local linear Lagrange FE space on a triangle incident to the vertex. However, the hat function is not simply the sum of these local functions, as this would result in over-counting values on the shared edges between triangles. The problem of how to handle boundaries between elements also arises when we seek to describe spaces of discontinuous functions, whose values at such boundaries are ill-defined. For instance, this occurs when modeling fractures, creases, or interfaces between different materials.



We need a few auxiliary notations from topology in order to talk coherently about the boundary vs. interior of our elements. We write $\overline{S}$ to mean the closure of the set S, and S° to mean the interior of the set S. We say that a set S is **regular closed** if $\overline{S^\circ} = S$.

Definition 3.3.1 (Restriction of Functions)**.** Suppose $T \subseteq \Omega$ is a regular closed set and V is a vector space. Given $f \in \mathcal{F}(\Omega, V)$ such that f is continuous and bounded on T°, then $\rho_T(f)$, the **restriction** of f to T, is the unique continuous extension of $f|_{T^\circ}$ to T.



A key application of this concept is the construction of discontinuous functions: consider two triangles $T_0, T_1 \subseteq \Omega$ sharing a vertex v, as well as a function f defined on Ω. If f is discontinuous at v, then we can evaluate $\rho_{T_1}(f)$ and $\rho_{T_2}(f)$ at v to get two distinct values. This allows us to have two different DOFs at v and thus to construct *global* FE spaces that include discontinuous functions.

We can generalize the preceding construction to extend local dual functions back to operating on the global space.

Definition 3.3.2 (Extension of Dual Functions)**.** Suppose $T \subseteq \Omega$ is a regular closed set and V is a vector space. Given a $\phi \in \mathcal{C}^\star(T, V)$, the **extension** to Ω is a dual function $E_T\phi \in \mathcal{F}(T, V)^\star$ defined via $E_T(\phi)(f) = \phi(\rho_T(f))$ for all f. More concisely, $E_T(\phi) = \phi \circ \rho_T$. Hence E_T is dual (a.k.a. adjoint) to ρ_T.

This machinery is sufficient to define a broken global FE space, which models a fully discontinuous space.

Definition 3.3.3 (Broken Global Finite Element Space)**.** Let $\Omega \subset \mathbb{R}^n$ be a domain meshed by a finite collection of elements $\{T_i\}$, each a regular closed subset of Ω and forming a partition of Ω in the sense that $\Omega = \cup T_i$ and $T_i^\circ \cap T_j^\circ = \emptyset$ if $i \neq j$. Furthermore, let (T_i, V, P_i, Σ_i) be the FEs associated to each T_i. Then define

1. a global space of piecewise functions $P = \{p \colon \Omega \to V \mid \forall i. \rho_{T_i}(p) \in P_i\}$
2. a global set of dual function $\Sigma = \cup_i \{E_{T_i}\phi \colon \phi \in \Sigma_i\}$ so that $\Sigma \subset P^\star$.

We call (Ω, V, P, Σ) the **broken global finite element space** associated to a collection of finite elements.

Note that the union of local dual basis functions in the above construction is disjoint. We have exactly one global dual function for each local dual function on each element. Further note that the construction of P given above is intuitively accurate, but requires tricky functional analysis to specify precisely[3].

The broken space does not yet allow us to produce our hat functions. Intuitively, we still need to ensure that all the DOFs corresponding to point evaluation at the same vertex evaluate to the same value for all functions in P. To do this, we must group or equate DOFs to construct the global FE space. We accomplish this via an equivalence relationship on the global broken degrees of freedom.

Definition 3.3.4 (Global Finite Element Space)**.** Suppose we have a collection of finite elements (T_i, V, P_i, Σ_i) that form a broken global finite element space (Ω, V, P, Σ). Further

[3] Let us consider an example of the issue that occurs. Consider a mesh of the interval $[0, 2]$ consisting of two sub-intervals $[0, 1]$ and $[1, 2]$. Even if the local function spaces consist of affine functions, P as constructed above does not specify the value of $f(1)$ for $f \in P$; it can be arbitrarily modified. This problem occurs in much of functional analysis and is resolved by drawing functions from Lebesgue spaces, such as L_1. These spaces technically consist of equivalence classes of functions, rather than individual functions. This is almost entirely a distraction from any of our development here, so we choose to not needlessly complicate our presentation further.

suppose $\equiv$ is an equivalence relationship on Σ such that basis functions from the same element are not made equivalent: if $\phi_1, \phi_2 \in \Sigma_i$ for some i and $\phi_1 \neq \phi_2$, then $E_\Omega \phi_1 \not\equiv E_\Omega \phi_2$. Then we can define $P_\equiv \subset P$ via

$$P_\equiv = \{p \in P \colon \forall \phi, \phi' \in \Sigma \colon \phi \equiv \phi' \implies \phi(p) = \phi'(p)\},$$

and we can define

$$\Sigma_\equiv = \Sigma / \equiv .$$

The **global finite element space** is then $(\Omega, V, P_\equiv, \Sigma_\equiv)$.

We make two observations and a remark. First, a good choice of equivalence relationship can sometimes be generated via the space of continuous functions, $C^0(\Omega)$:

$$\phi_1 \equiv \phi_2 \leftrightarrow \forall f \in C^0(\Omega, V) \colon \phi_1(f) = \phi_2(f).$$

This requires that all dual basis functions are well defined on functions in $C^0(\Omega, V)$. In the FE literature, equivalence relations are used implicitly by way of a choice of function space. Not all equivalence relationships provide sensible spaces for PDEs, but we outline a combinatorial guideline in appendix A. Second, although $\Sigma_\equiv$ is technically a set of equivalence classes, the construction of $P_\equiv$ as a quotient allows us to use any set of representatives of the classes $\Sigma_\equiv$ as a dual basis for $P_\equiv$. Third, just as with the broken global FE space, we can easily characterize the primal basis functions, which will allow us to characterize how we compute using global spaces.

Remark 3.3.5 (The Primal Basis for the Global Spaces). Fix an arbitrary choice of representatives ϕ_i for the equivalence classes $\varphi_i \in \Sigma_\equiv$. This $\{\phi_i\}$ is a basis for $P^\star_\equiv$. Further, suppose $\{f_j\}$ is the corresponding primal basis ($\phi_i(f_j) = \delta_{ij}$). Suppose we are given a T_k. We can characterize the behavior of $f_j(x)$ on $x \in T_j$ with two cases:

1. If there is a $\psi_i \in \varphi_j$ such that there is a $\psi_{i,k} \in \Sigma_k$ where $\psi_i = E_{T_k} \psi_{i,k}$, then $\rho_{T_k}(f_j) = g_{i,k}$

where $g_{i,k} \in P_i$ is the primal basis function corresponding to $\psi_{i,k}$.

2. Otherwise, $\rho_{T_k}(f_j) = 0$.

In light of this construction, it is useful to introduce a bit of notation to deal with equivalence classes. Given $\psi \in \Sigma_i$, let $[\psi]_\equiv$ be the equivalence class in $\Sigma_\equiv$ that contains $E_{T_i}\psi$.

The idea in the above remark proves a key lemma that will help us compute on functions defined in the global FE bases:

Lemma 3.3.6. Fix a global finite element space $(\Omega, V, P_\equiv, \Sigma_\equiv)$ and the corresponding local finite element spaces $(T_i, V_i, P_i, \Sigma_i)$. Let $\Sigma_\equiv = \{\varphi\}$ and let $\{f_\varphi\}$ be the corresponding primal basis. Also, for each i and $\psi \in \Sigma_i$, let $f_{i,\psi}$ be the primal basis function corresponding to ψ. Suppose $f \in P_\equiv$ is such that there is $\alpha_\varphi \in \mathbb{R}^{|\Sigma_\equiv|}$ so that

$$f = \sum_{\varphi \in \Sigma_\equiv} \alpha_\varphi f_\varphi.$$

Then for all i, we have that

$$\rho_{T_i} f = \sum_{\psi \in \Sigma_i} \alpha_{[\psi]} f_{i,\psi} \tag{3.16}$$

.

As an initial application of the above machinery of global FE spaces, we can define a notion of interpolation. Traditionally, interpolation uses a set of points $\{x_i\}$ and a corresponding set of measurements $\{y_i\}$. Interpolation seeks to find a function that satisfies $f(x_i) = y_i$. For FEs, we can replace the pairs (x_i, y_i) with dual basis functions, ϕ_i, and values for these dual basis functions, α_i. In line with the above remark, given a global FE space $(\Omega, V, P_\equiv, \Sigma_\equiv)$, we can define an $f \in P$ with a vector $\alpha \in \mathbb{R}^{|\Sigma_\equiv|}$. In this setting, given an α_i associated to each ϕ_i, we can define a function f using the notation of the above remark:

$$f = \sum_{\phi_i \in \Sigma_\equiv} \alpha_i f_i.$$

And, of course, via the primal and dual basis property, we have that $\phi_i(f) = \alpha_i$. However, we are not often given a set of values α_i as actual numbers. Instead, we often are given some other function h, not necessarily in the FE space, that we use to compute the α_i, which then defines a function f that approximates h.

Definition 3.3.7 (Finite Element Interpolation)**.** Let $E = (\Omega, V, P_{\equiv}, \Sigma_{\equiv})$ be a global finite element space. Let $h \in \mathcal{F}(\Omega, V)$ be such that for all $\phi \in \Sigma_{\equiv}$, $\phi(h)$ is well defined. Then the interpolation of h into the global finite element space is a function $I_E h \in P_{\equiv}$ given by

$$I_E h = \sum_{\phi_i \in \Sigma_{\equiv}} \phi_i(h) f_i. \tag{3.17}$$

FE interpolation lets one approximate a wide variety of functions with FE functions. Practically, interpolation allows one to get data into the system in a consistent way. In particular, if some FEM converges at an order k with some given FE, then interpolating incoming data will often not disrupt this order of convergence.[4]

3.4 Global Operations on Finite Element Spaces

We now deduce how to use global FE spaces in computations. We examine three examples of using the global FE spaces: integration over a domain, integration over a subset, and FE style interpolation. We explore these methods by example to present the issues involved in working with global FE spaces and to motivate a more general method that will handle all of these examples. In particular, we seek to reduce our computations to be cell local in the sense that the data needed to compute the quantities is accessible from a cell or a local basis function that corresponds to some global basis function.

Our first example is analogous to the action of a PDE operator or some derived integral quantity of a solution. This example shows how to make such computations cell local. We

[4]In some systems, e.g., Firedrake/FEniCS, inappropriate representations of DOFs have led to interpolation changing the order of convergence.

also extend the example to the case of integrating over the entire mesh for each basis function. This shows how we can construct an array that represents the operator in the primal basis of the global FE space.

Example 3.4.1 (Integration of Finite Element Functions)**.** *Fix a global finite element space* $(\Omega, V, P_{\equiv}, \Sigma_{\equiv})$ *with local elements* $E_i = (T_i, V_i, P_i, \Sigma_i)$*. Set* $\{\varphi_k\} = \Sigma$*. Let* $\{f_{\varphi_k}\}$ *be the associated primal basis. Given* $f \in P_{\equiv}$*, we have that there is a vector* α_{φ_k} *so that*

$$f = \sum_{\varphi_k \in \Sigma_{\equiv}} \alpha_{\varphi_k} f_{\varphi_k}. \tag{3.18}$$

Given any integrable function $F\colon \mathbb{R} \to \mathbb{R}$*, we have that*

$$\begin{aligned}
\int_{\Omega} F(f(x))dx &= \sum_i \int_{T_i} \rho_{T_i}(F(f))(x)dx \\
&= \sum_i \int_{T_i} \rho_{T_i}(F(f))(x)dx \\
&= \sum_i \int_{T_i} F((\rho_{T_i}(f))(x))dx
\end{aligned}$$

Applying lemma 3.3.6 to $\rho_{T_i}(f(x))$*, we have that*

$$\begin{aligned}
\rho_{T_i}(f) &= \rho_{T_i}(\sum_{\varphi_k \in \Sigma_{\equiv}} \alpha_{\varphi_k} f_{\varphi_k}) \\
&= \rho_{T_i}(\sum_{\phi \in \Sigma_i} \alpha_{[\phi]_{\equiv}} f_{[\phi]_{\equiv}}) \\
&= \sum_{\phi \in \Sigma_i} \alpha_{[\phi]_{\equiv}} \rho_{T_i}(f_{[\phi]_{\equiv}}) \\
&= \sum_{\phi \in \Sigma_i} \alpha_{[\phi]_{\equiv}} f_{\phi}.
\end{aligned}$$

So we have that

$$\int_\Omega F(f(x))dx = \sum_i \int_{T_i} F(\sum_{\phi \in \Sigma_i} \alpha_{[\phi]_\equiv} f_\phi(x))dx.$$

This formula allows one to compute an integral of a function defined in a finite element space using two cell-local quantities.

First, we need the local finite element basis functions of each element. Second, we need the coefficients defined via the equivalence class associated to each primal basis function via duality.

If in addition F is also linear, then

$$\begin{aligned}\int_\Omega F(f(x))dx &= \sum_{\varphi_k \in \Sigma_\equiv} \alpha_{\varphi_k} \int_\Omega F(f_{\varphi_k}(x))dx \\ &= \sum_{\varphi_k \in \Sigma_\equiv} \alpha_{\varphi_k} \sum_i \int_{T_i} (\rho_{T_i}(F(f_{\varphi_k})))(x)dx \\ &= \sum_{\varphi_k \in \Sigma_\equiv} \alpha_{\varphi_k} \sum_{E_{T_i \phi_j} \in \varphi_k} \int_{T_i} F(\phi_j(x))dx.\end{aligned}$$

Thus if

$$F_k := \int_\Omega F(\varphi_k(x))dx = \sum_{E_{T_i \phi_j} \in \varphi_k} \int_{T_i} F(\phi_j(x))dx$$

then

$$\int_\Omega F(f(x))dx = \sum_k F_k \alpha_{\varphi_k}.$$

Thus, we have a formula for an array that lets us compute $\int_\Omega F(f(x))dx$ on $f \in P_\equiv$ using standard linear algebra. Again, this array can be computed in a cell-local fashion modulo aggregation between cells that share a dual basis function. This example is simple, but more complex examples on the interior of elements will remain cell local modulo some aggregation. If we added more finite element basis functions (a bilinear operator) or more fixed functions *(a non-linear argument), the same steps would lead to a cell local formula.*

A simple variation of the above computation is an integral over a union of a set of edges as opposed to a union of a set of triangles. This occurs in the form of boundary integrals, which are part and parcel of the weak form of many PDEs as well as in integrals over interior facets.

Integrals over interior edges occur in various methods for the biharmonic equation, in FE style interpolation, in tests to validate FEs, and at interfaces between different materials in simulations [38].

Example 3.4.2 (Facet Integrals of Finite Element Functions)**.** *Fix a global finite element space $(\Omega, V, P_{\equiv}, \Sigma_{\equiv})$ with local elements $E_i = (T_i, V, P_i, \Sigma_i)$. Let $E \subset \partial T_k \subset \Omega$ for some k. If $E \subset \partial\Omega$, then for some $f \in P_{\equiv}$, we can follow the previous steps to obtain*

$$\int_E F(f(x))dx = \int_E F(\sum_{\phi\in\Sigma_k} \alpha_{[\phi]_\equiv} f_\phi(x))dx. \tag{3.19}$$

However, if E is in the interior, then we cannot reach this point. There is some $k' \neq k$ so that $E \subset \partial T_{k'}$. Even if this is the only k', it may be that

$$\begin{aligned}\int_E F(f(x))dx \neq &\int_{\partial T_k \cap E} \rho_{T_k}(F(f))(x)dx \\ &+ \int_{\partial T_{k'} \cap E} \rho_{T_{k'}}(F(f))(x)dx\end{aligned}$$

In fact, $\int_E F(f(x))dx$ is not necessarily well defined. There are two approaches that make mathematical sense. First, one can compute $\int_{\partial T_k \cap E} \rho_{T_k}(F(f))(x)dx$ and $\int_{\partial T_{k'} \cap E} \rho_{T_{k'}}(F(f))(x)dx$ as separate values. These can be combined to various ends (e.g., if the functions are continuous then we could just pick one as the quantity is well defined). This approaches extends to lower-dimensional objects like points in $\mathbb{R}^2$ shared by many cells. Second, one can define F on an interior boundary as a function of two values, one from each cell. This approach *is no longer cell local as it requires an adjacency relationship to combine basis functions defined in different cells in the same pointwise computation. Further, this only works on $d-1$-dimensional objects shared between two cells. In either case, the complexity of these approaches multiplies with more arguments as there will be separately defined quantities for every combination of values that could exist. For example, the assembly of a bilinear function into a 2d array might need to account for the cases of two values coming from the same side or two values from different sides, leading to four distinct well-defined quantities to compute per edge [39].*

Our final example computation is FE interpolation as defined in definition 3.3.7.

Example 3.4.3 (Computing Interpolation Operations)**.** *Fix a global finite element space $(\Omega, V, P_{\equiv}, \Sigma_{\equiv})$ with local elements $E_i = (T_i, V, P_i, \Sigma_i)$. Suppose we have some function $f \in \mathcal{F}(\Omega, V)$ such that for each i and each $\phi \in \Sigma_i$, $E_{T_i}(\phi)(f)$ is well defined. Computing each $E_{T_i}(\phi)(f)$ is a simple matter as this is already a cell local computation. However, for some $\psi \in \Sigma_{\equiv}$ with $E_{T_i}(\phi)(f) \in \psi$, there might also be an i' and $\phi' \in \Sigma_{i'}$ such that $E_{T_{i'}}\phi' \in \psi$. There is no guarantee that $E_{T_i}(\phi)(f) = E_{T_{i'}}(\phi')(f)$. Thus, to ensure that interpolation is well defined, we must compute every cell-local degree of freedom for each $\psi \in \Sigma_{\equiv}$ and then we must check that they are all equal. If they are not equal, interpolation is not well defined without additional information. In such cases, one can investigate procedures to combine the different cell-local values (e.g., averaging them). Thus this situation is fairly similar to the previous example, though integration might be replaced with point evaluation and we have an output per dual function. However, we also note that within the finite element literature, one can find various interpolation schemes to solve variants of this problem. Oftentimes, these interpolation schemes use alternative sets of dual vectors to compute values that are then used as the α_i for some finite element. Most notably Scott-Zhang interpolation handles "rough" functions by integrating functions against Lagrange basis functions (as opposed to evaluating at cell vertices where functions might be discontinuous) [40].*

In each of these examples, we tried to use the structure of a global FE space to make the computation local to cells. Our ability to do so is moderated by the location of the computation (interior edge vs. interior cell), the continuity of the functions in the global FE space, and the specific properties of the formula F that we are integrating. Additionally, sometimes we must handle that computations are produced per combinations of basis functions in cases where the formula F is multi-linear or when we produce an output per basis function as in various interpolation algorithms. These example computations belong to a class called *assembly operations*, which we will more precisely define in section 7.1. Further, in section 7.1, we will provide a universal algorithm for computing assembly operations, which will address the various issues encountered in the above examples and which will guide the code generation for assembly operations in our system.

Chapter 4

Related Work

4.1 Finite Elements in Graphics

FEMs are a standard part of the toolkit of computer graphics, especially piece-wise linear FEs. The piece-wise linear discretization of the Laplacian operator appears ubiquitously in computer graphics in the form of the cotangent Laplacian [23]. Moreover, piece-wise linear Lagrange FEs are used to simulate fluids, simulate solids, simulate fluid structure interaction, mesh objects, render objects, smooth meshes, and more [41]–[48].

In several subfields of computer graphics, alternatives to piece-wise linear Lagrange FEs are under consideration due to various problem specific needs. The primary alternatives under consideration are simply higher-order Lagrange elements or similar constructions for squares and hexahedra (such as B-splines) [5], [6], [49]. Most notably, higher-order Lagrange elements (typically not beyond cubic) have been employed to deal with locking phenomena in elasticity problems [49], [50]. In elasticity in particular, more exotic elements have been used specifically to deal with locking or other more specific phenomena [51], [52]. In fluids in particular, B-Spline constructions have been employed as FEs, though with staggering to allow for smoother bases [53]. Finally, the lower-order non-linear Lagrange elements have been employed in a variety of other problems, such as meshing and surface deformation [54], [55].

Beyond higher order Lagrange, a few have used the FEEC elements (mainly the Whitney elements via subdivision representations) and their relatives (Crouzeix-Raviart), especially when dealing with the structure of harmonic functions [27], [28], [56]–[61]. In the latter case, the elements are always of the lowest orders. Even more rarely in these cases, authors have used elements that we cannot find implemented elsewhere, such as the Fraeijs de Veubeke's element [62]. The same work also uses Morley's element but does not provide any related details on its useage

4.2 Automation for the Finite Element

In this section, we survey the related work with respect to the ability of other FE software to use definition 3.2.1 of a FE automatically and parametrically. To the best of our knowledge, to add a new FE to an existing FEM implementation, all existing systems require some mathematical derivations and some manual coding (if it is possible at all to add new FEs). In particular, all systems require that the user manually ensure that the basis functions of a FE can be evaluated on an arbitrary domain K - you must verify some version of definition 3.2.2 or eq. (3.13). For non-affine equivalent elements (definition 3.2.2), the most flexible and sophisticated libraries, often built out of multiple DSLs, allow users to manually provide matrix representations of the pullback matrix as a function of the geometry of a domain. As far as we know, only Firedrake, FEniCS, NGSolve, Dune, scikit-fem, and GetFEM provide such overrides [9], [10], [13], [63], [64]. These libraries offer a wide variety of FEs, with Firedrake supporting the greatest variety [9], [65]–[68]. In the cases where (definition 3.2.2) holds, FIAT, a library used in Firedrake, can automatically generate basis functions for a wide variety of DOFs, polynomial spaces, and domains, while FINAT facilitates the manual specification overrides and other symbolic data useful for finite elements [7]–[9], [15], [34]. In contrast, ElementForge will do all that FIAT can do while automatically deducing the overrides FINAT provides for inverse Vandermonde matrices, among other tasks.

Outside of providing basis function values, several other aspects of using FEs have been partially automated in a few libraries. The systems that support non-affine equivalent elements support using their DOFs to perform interpolation or to enforce BCs to varying degrees, though in all cases the support is partial [9], [10], [13], [63], [64]. Similarly, some of these systems make assumptions on the mesh to make certain types of FEs work [69] while others, mainly FEniCS via the Basisx library for basis functions, allow users to specify how basis functions are impacted by transformations [70].

The overwhelming majority of libraries live between FEniCS and systems like PolyFEM, which only provide Lagrange elements up to a certain order [11], [16], [71]–[78]. These systems shy away from the problems of using non-affine equivalent elements, managing permutations of basis functions, enforcing element specific boundary conditions, or performing interpolation operations other than point evaluations. Finally, even SymFEM, the symbolic FE library and the tool behind the website defelement.com, does not symbolically determine the pullback matrix [79], so that the Morley element, for example, can only be constructed on a single specific triangle in that system. In contrast, we seek to provide a far greater degree of automation for elements by ensuring that our specifications work on symbolic geometries, enabling the automatic generation of formulas for Vandermonde matrices among many other tasks.

4.3 Automation for Other Aspects of the Finite Element Method

Outside of the FE itself, many other aspects of the FEM have been somewhat automated, often via DSLs. Discussions of meshing software are not germane to our purposes, but the specification of weak formulations is heavily related to our enterprise.

Essentially three approaches have emerged to allow users of FEM software to specify their problems. First, systems such as PolyFEM or nekRS offer parameterized PDEs that users can configure with expressions for fixed tensors or BCs or other parameters [6], [71]. Second, systems such as MFEM and Nektar++ offer libraries of operators that can be combined to specify the variational formulation of a PDE [11], [78]. Third, several systems such as Firedrake, FEniCS, and Dune rely on the Unified Form Language, a DSL for specifying weak forms as expressions [12], [14]–[16].

Form languages are the line of work closest to our own as the specification form is similar to the specification of elements, but they are not used this way. Form language compilers produce quadrature loops, which are heavily optimized for performance, as opposed to symbolic outputs that can be used to compute pullback matrices [14], [19], [80], [81]. As a minor example of this, when users wanted to use FEEC elements in form compilers, form compilers had to be modified to deal with pullbacks associated with FEEC elements as opposed to symbolically deducing such pullbacks [82]. However, we note that form compilers have other symbolic capabilities such as various forms of automatic differentiation [12], [83], [84] and the ability to express many highly specialized time-steppers and preconditions [39], [85]–[87]. At this time, we do not pursue this higher-level work, but we do produce a form language (the same language we use to specify elements). Our form language inherits some advantages and disadvantages from a non-specialized form language: our language can be more concise for many forms that operate on high-level tensor expressions, especially those involving exterior calculus, but our language can also be more verbose as we do not build in specialized constructs for form differentiation or for discontinuous Galerkin methods.

4.4 Less Well-Known Finite Elements

We highlight the many FEs that appear in the literature, briefly commenting on their various utilities. For example, there are many families of FEs that offer various forms of continuity [21], [88]–[94]. Sometimes FEs are defined because they allow elegant and correct solutions to certain problems [95]–[100]. Some FEs offer superior convergence speeds or conditioning properties over easier to use options [101]–[107] or sometimes they offer highly efficient code for the same convergence [108]–[112]. Some FEs are uniquely suited for certain applications using some portion of the above properties [95], [96], [113]–[120]. We believe that ElementForge can implement most of these elements with the exception of the small minority of elements relying on rational function spaces. We do require an extension for piecewise polynomial spaces in the element space, but these are known to be simple.

Chapter 5

Software Representation of DOFs and Operators: Integrated Pointwise Tensor Expressions on Manifolds

We capture both DOFs and operators via Integrated Pointwise Tensor Expressions on Manifolds (IPTEM). We define IPTEM to decompose the expression into various components: integrals, pointwise expressions, tensor expressions, and an interface to geometry motivated by differential geometry. IPTEM are flexible enough to express most known DOFs and operators, but, critically for our automation efforts, each component of IPTEM models a continuous concept in a manner amenable to algorithmic reasoning for code generation. We discuss each component in turn from lowest to highest level, building to the full interface, but considering how each part facilitates or balances the expression of FEM concepts and the automatic generation of formulas.

5.1 Tensor Expressions: Tensors, Tensor Spaces, Basis

5.1.1 Design Rationale

Along the lines of our discussion of tensors in section 3.1, we provide a grammar in listing 5.1 that balances writing down tensors and generating formulas with natural constructions. In section 3.1, we argued that certain natural tensor objects produce simpler formulas, so we organized listing 5.1 around natural constructions in tensor algebra. We also observed that sometimes basis-dependent computation is necessary, but can be carefully isolated via dagger maps. Though other more familiar tensor representations such as Einstein index notation or Numpy arrays are trivial for computers to implement, our representation can produce better formulas because natural constructions and careful usage of basis dependent computations via inner products and daggers can expose simpler computer formulas via exploiting choice of basis. Additionally, our language naturally expresses everything that Einstein index notation can express while also providing a more natural class of operations such as reshapes.

5.1.2 Semantics

A full denotational semantics for the language is superfluous for our language as it is a direct specification of mathematics. Here we seek to clarify the meaning of the terms, illustrate how they are grouped by natural constructions, and argue that each space is a Hilbert space.

Our first space is $\mathbb{R}$, which is associated to the operations available in every vector space 0, $+$ and scalar multiplication via $\times$. This space is a Hilbert space under the inner product $(v, w)_{\mathbb{R}} \mapsto v \times w$.

Our next two space constructions, dual spaces and tensor products, are intertwined via the universal property for tensor algebra (essentially proposition 3.1.5). As a start, all vector spaces V get an identity vector map, which necessarily is a member of the tensor space $V^\star \otimes V \cong L(V, V)$. Any vector space $v' \in V^\star$ can evaluate a vector $v \in V$, which we represent via Eval(v', v). Similarly, given any two vectors $v \in V$ and $w \in W$, we can construct $v \otimes w$. Dual vectors and multi-linear maps can be constructed naturally via a function abstraction (lambda) and a variable construct. The Evaluation, Abstract, and variable terms allow users to realize one direction of the isomorphism in proposition 3.1.5 and the natural isomorphism from V to $V^{\star\star}$. The opposite direction of the natural isomorphism is offered via the J map. The other direction of proposition 3.1.5 (going from $V^\star \otimes W^\star$ to $V \otimes W$ as opposed to the other way around) corresponds to the reshape term, which groups tensor products of dual spaces (at the given indices) into a dual space of a tensor product space (at a specific location in the original tensor product). The reshape term facilitates the core consequences of proposition 3.1.5 such as the associativity of tensor products, which corresponds to a reshaping operation.

If V and W have inner products $(\cdot, \cdot)_V$ and $(\cdot, \cdot)_W$, assumes inner products on $V^\star$ and $V \otimes W$. For the latter case, we can define $(v_0 \otimes w_0, v_1 \otimes w_1)_{V \otimes W} = (v_0, v_1)_V (w_0, w_1)_W$ with an appeal to proposition 3.1.5. In our language, we can construct this vector directly with eval, abstraction, variable, inner, and reshape (the appeal to proposition 3.1.5). For the former case, we appeal to a formulation of the dagger map (definition 3.1.1) that works for any orthonormal basis, the inverse of the map $v_0 \mapsto v_1 \mapsto (v_0, v_1)_V$. With such a map, the inner product on $V^\star$ is given by $(v_0', v_1')_{V^\star} = ((v_0')^\dagger, (v_1')^\dagger)_V$.

If we have two spaces V and W, we can construct $V \oplus W$. The universal property of products supplies projection operations from $V \oplus W$ to V or W. We also naturally have embeddings from V or W to $V \oplus W$. Finally, if V and W have inner products, we have an inner product $\langle z_0, z_1 \rangle_{V \oplus W} = (P_{V \oplus W, V} z_0, P_{V \oplus W, V} z_1)_V + (P_{V \oplus W, W} z_0, P_{V \oplus W, W} z_1)_W$.

Kernel, co-kernel, image, and co-image behave similarly, in that given a linear $v \in V^\star \otimes W$, they provide projection and embedding maps into/from V or W. For convenience, we also allow these spaces to be annotated with information about the map, mainly if the map is surjective and/or injective. The embedding maps of the kernel and image fulfill the universal properties of those spaces while the projection maps of the co-kernel and co-image fulfill their universal properties. We can access the isomorphism from the co-kernel to image via the restriction operation. Similarly, these maps allow existence of an inverse map: a mapping from the image of a map to the co-kernel. The inner products for these spaces are simply the inner products of the source or target of the map, utilized via embedding maps.

Finally, for expressibility, we note that we support evaluation of external non-linear functions such as square root, division, or trigonometric functions. We note that such computation cannot be done with variables bound in abstraction nodes, as these define multi-linear functions. Our system checks that Abstract nodes define truly multi-linear functions.

$\langle \textit{TensorSpace} \rangle = \mathbb{R}$

$\quad | \quad \langle \textit{TensorSpace} \rangle^{\star}$

$\quad | \quad \langle \textit{TensorSpace} \rangle \otimes \langle \textit{TensorSpace} \rangle$

$\quad | \quad \langle \textit{TensorSpace} \rangle \oplus \langle \textit{TensorSpace} \rangle$

$\quad | \quad$ kernel($\langle \textit{TensorExpr} \rangle$, ?$\langle \textit{TensorSpace} \rangle$, ?$\langle \textit{TensorSpace} \rangle$,?bool, ?bool)

$\quad | \quad$ cokernel($\langle \textit{TensorExpr} \rangle$, ?$\langle \textit{TensorSpace} \rangle$, ?$\langle \textit{TensorSpace} \rangle$, ?bool,?bool)

$\quad | \quad$ image($\langle \textit{TensorExpr} \rangle$, ?$\langle \textit{TensorSpace} \rangle$, ?$\langle \textit{TensorSpace} \rangle$, ?bool, ?bool)

$\quad | \quad$ coimage($\langle \textit{TensorExpr} \rangle$, ?$\langle \textit{TensorSpace} \rangle$, ?$\langle \textit{TensorSpace} \rangle$, ?bool, ?bool)

$\langle \textit{TensorExpr} \rangle =$ Zero($\langle \textit{TensorSpace} \rangle$)

$\quad | \quad$ Scalar($\langle \textit{num} \rangle$)

$\quad | \quad \langle \textit{TensorExpr} \rangle + \langle \textit{TensorExpr} \rangle$

$\quad | \quad \langle \textit{TensorExpr} \rangle \times \langle \textit{TensorExpr} \rangle$

$\quad | \quad$ Identity($\langle \textit{TensorSpace} \rangle$)

$\quad | \quad$ Eval($\langle \textit{TensorExpr} \rangle$, ?$\langle \textit{int} \rangle$, $\langle \textit{TensorExpr} \rangle$)

$\quad | \quad \langle \textit{TensorExpr} \rangle \otimes \langle \textit{TensorExpr} \rangle$

$\quad | \quad$ Abstract(+$\langle \textit{str} \rangle$, +?$\langle \textit{TensorSpace} \rangle$, $\langle \textit{TensorExpr} \rangle$, ?$\langle \textit{TensorSpace} \rangle$)

$\quad | \quad$ Var($\langle \textit{str} \rangle$)

```
| J(⟨TensorSpace⟩)
| Reshape(⟨TensorExpr⟩, *⟨int⟩, ⟨int⟩)
| Inner(⟨TensorSpace⟩)
| ⟨TensorExpr⟩†
| ⟨TensorExpr⟩ ⊕ ⟨TensorExpr⟩
| Project(⟨TensorSpace⟩, ⟨TensorSpace⟩)
| Embed(⟨TensorSpace⟩, ⟨TensorSpace⟩)
| Inverse(⟨TensorExpr⟩)
| Restrict(⟨TensorExpr⟩)
| NonLinear(⟨str⟩, ⟨TensorExpr⟩)
```

Source Code Listing 5.1: A Grammar to Describe Tensor Expressions and Tensor Spaces.

5.1.2.1 Simple Examples

To make the language immediately concrete, we build some simple examples. The space $\mathbb{R}$ is well known, as is the space $\mathbb{R} \oplus \mathbb{R} \cong \mathbb{R}^2$, which has elements of the form $a \oplus b$ where $a, b \in \mathbb{R}$. A more complex example is $\mathbb{R}^2 \otimes \mathbb{R}^2$, which is isomorphic to the space of 2 by 2 matrices. We might write an element in this space via $(a_1 \oplus b_1) \otimes (a_2 \oplus b_2)$, which can be written as the matrix $\begin{bmatrix} a_1a_2 & a_1b_2 \\ b_1a_2 & b_1b_2. \end{bmatrix}$. To write an element of a kernel, image, or their co-variants, we need to use the dual space. Thus, we might instead consider the element $v = (a_1 \oplus b_1)^\dagger \otimes (a_2 \oplus b_2)$, which is a member of the space $(\mathbb{R}^2)^\star \otimes \mathbb{R}^2$. Because of the presence of the dual element, this vector can be evaluated. We could, for example, write `Eval(v, 0, (`$a_1 \oplus b_1$`))`, which mathematically is $v(a_1 \oplus b_1) = ((a_1 \oplus b_1)^\dagger \otimes (a_2 \oplus b_2))(a_1 \oplus b_1) = (a_1 \oplus b_1)^\dagger((a_1 \oplus b_1)) \otimes (a_2 \oplus b_2) = (a_1^2 + b_1^2) \otimes (a_2 \oplus b_2) = (a_1^2 + b_1^2) \times (a_2 \oplus b_2)$. The vector $v \in (\mathbb{R}^2)^\star \otimes \mathbb{R}^2$ is paired with a vector in $\mathbb{R}^2$ to make another vector in $\mathbb{R}^2$. Since v defines a linear map from $\mathbb{R}^2$ to $\mathbb{R}^2$, we can use it to define the various subspace constructors, and vectors can be defined by projecting vectors from $\mathbb{R}^2$ into this space.

5.1.3 Validity of Programs

Our grammar has several context-dependent constraints on its validity. For example, a construction $v \times w$, meaning scalar multiplication, requires that one of the items be a scalar, a member of $\mathbb{R}$. Similarly, both Eval and Reshape only work on tensor product spaces where the integer (in the former) or list of integers (in the latter) identifies elements of a tensor product space that are dual spaces. In the case of Eval(v,j,w), the dual space within the tensor product space of v identified by the integer j must be the same space that w is a member of. Another relatively simple requirement is that the tensor expressions used by the subspace constructors, Inverse, or Restrict all are members of spaces of the form $V^\star \otimes W$, as these all expect linear maps which are identified by $V^\star \otimes W$. Finally, the Abstract(vs, sps, expr, sp) construct has perhaps the most complex context dependent restriction: the expression must be a linear function in each of the inputs. This is similar to context-dependent typing systems such as linear types found in languages such as Rust [121], [122].

5.1.4 Expressing Generalized Pullbacks

To demonstrate the expressiveness of our language and to provide a utility for future uses, we show that our language can express a generalization of the notion of pullbacks from $V^{p,q}$ spaces. We assume that we have a set of base cases, pairs of spaces (V_i, W_i) with tensors $T_i \in V_i^\star \otimes W_i$. If we assume that T_i is invertible, then we can clearly express the adjoint inverse in the language via $T_i^{-\star}(x) = y \mapsto x(T_i^{-1}(y))$. Technically, we can assume T_i is just injective and replace every W_i with the image of T_i. We construct the generalized pullback by destructing a tensor expression until we hit a base case. We sketch this by sketching a match on a space Z.

1. If $Z = V^\star$ and V is not V_i, then we recurse on V to get a mapping on V and use the

adjoint inverse construction from above.

2. If $Z = P \otimes Q$, we can recurse onto P and Q to get mappings T_P and T_Q and then write

$$\mathrm{Reshape}(\mathrm{Abstract}(u, (\mathrm{Abstract}(v, T_P(u) \otimes T_Q(v))), 0, 1, 0)$$

   to get a mapping on Z.

3. If $Z = P \oplus Q$, we can recurse to get a mapping T_P and T_Q so that we can write

$$\mathrm{Embed}(P, Z) \circ T_P \circ \mathrm{project}(Z, P) + \mathrm{Embed}(Q, Z) \circ T_Q \circ \mathrm{project}(Z, Q)$$

   to get a mapping on Z.

4. For each of the subspace constructions, if Z is a subspace of Z' and we get a mapping $T_{Z'}$, then we can write

$$\mathrm{Abstract}(z', \mathrm{project}(Z', Z)(T_{Z'}(\mathrm{embed}(Z, Z')(z'))))$$

   to get a mapping on Z.

In the case of only one base space, it is not hard to check that this reproduces the $V^{p,q}$ space pullbacks, but this construction can provide pullbacks for more complex spaces.

5.1.5 Formulas

We now must be careful about what a formula is. Our answer is depicted in listing 5.2. In particular, our formulas balance easy computation and symbolic manipulation. Another option, Einstein index notation formulas, are trivially computed but don't expose the internal structure of arrays (e.g., what we see in the Morley matrix in example 3.2.3) and can't easily express computation such as inverse matrices. We use a different notion to facilitate a greater degree of symbolic manipulation: our formulas are straight-line array programs that produce arrays of values where each value is a rational function in intermediate values and the formula inputs. We choose rational functions because they can (sometimes) be symbolically analyzed efficiently and they can enable direct symbolic representation of the inverses of matrices, which we would like to analyze in the case of Vandermonde matrices. Note that we force all formulas to return arrays of rational functions to prioritize possible future simplification. Further, we provide a few external functions to manage linear algebra where rational functions will not do and where we don't expect a need for greater symbolic manipulation.

$\langle \mathit{shape} \rangle = *\langle \mathit{int} \rangle \; \langle \mathit{var} \rangle = (\langle \mathit{str} \rangle, \langle \mathit{shape} \rangle)$

$\langle \mathit{rf} \rangle = \text{VarAcc}(\langle \mathit{var} \rangle, *\langle \mathit{int} \rangle)$

$\quad | \quad \langle \mathit{rf} \rangle + \langle \mathit{rf} \rangle$

$\quad | \quad \langle \mathit{rf} \rangle * \langle \mathit{rf} \rangle$

$\quad | \quad \langle \mathit{num} \rangle * \langle \mathit{rf} \rangle$

$\quad | \quad - \langle \mathit{rf} \rangle$

$\quad | \quad \langle \mathit{num} \rangle >$

$\langle \mathit{array} \rangle = \text{Array}(*\langle \mathit{rf} \rangle, \langle \mathit{shape} \rangle)$

$\langle \mathit{stmt} \rangle = \langle \mathit{var} \rangle = \text{Einsum}(\langle \mathit{str} \rangle, *\langle \mathit{var} \rangle)$

$\quad | \quad (\langle \mathit{var} \rangle, \langle \mathit{var} \rangle) = \text{QR}(\langle \mathit{var} \rangle))$

$\quad | \quad (\langle \mathit{var} \rangle, \langle \mathit{var} \rangle, \langle \mathit{var} \rangle) = \text{SVD}(\langle \mathit{var} \rangle))$

$\quad | \quad \langle \mathit{var} \rangle = \text{Inv}(\langle \mathit{var} \rangle)$

$\quad | \quad (+\langle \mathit{var} \rangle) = \text{func}(\langle \mathit{str} \rangle, \langle \mathit{var} \rangle)$

$\quad | \quad \langle \mathit{var} \rangle = \langle \mathit{Array} \rangle$

$\quad | \quad \text{Return}(\langle \mathit{Array} \rangle)$

$\langle \mathit{formula} \rangle = (*\langle \mathit{var} \rangle, \langle \mathit{stmt} \rangle)$

Source Code Listing 5.2: A grammar to describe formulas, code that computes arrays that represent a tensor expression in a given basis.

5.1.5.1 Simple Example

To clarify the scope of formulas and their utility as an intermediate representation favoring symbolic simplification, we provide a simple example using Sympy [123] in listing 5.3. This example is a literal representation of a formula with a single input A, a 2 by 2 by 2 array. The grammar of rational functions forces the return of an array of rational functions similar to maintaining the rational vector at the end of listing 5.3. Since the structure of these rational functions is evident in the program at the return, other programs that are combined with this one have an opportunity to symbolically optimize based on the structure. Internally, our representation of formulas is an array of rational functions similar to the construction in listing 5.3, representing the final return of an array in the formula, but with additional information about the list of inputs and intermediate statements.

```python
import sympy as sp

# Step 1: Create a symbolic array "A" with shape (2, 2, 2), the input var to the formula
A = sp.IndexedBase('A', shape=(2, 2, 2))

# Step 2: Extract entries explicitly
x000 = A[0, 0, 0]
x001 = A[0, 0, 1]
x010 = A[0, 1, 0]
x011 = A[0, 1, 1]
x100 = A[1, 0, 0]
x101 = A[1, 0, 1]
x110 = A[1, 1, 0]
x111 = A[1, 1, 1]

# Step 3: Build rational functions explicitly
S_all = x000 + x001 + x010 + x011 + x100 + x101 + x110 + x111

r0 = (x000**2 + S_all) / (1 + x000)
r1 = (x001**2 + S_all) / (1 + x001)
```

```
r2 = (x010**2 + S_all) / (1 + x010)
r3 = (x011**2 + S_all) / (1 + x011)
r4 = (x100**2 + S_all) / (1 + x100)
r5 = (x101**2 + S_all) / (1 + x101)
r6 = (x110**2 + S_all) / (1 + x110)
r7 = (x111**2 + S_all) / (1 + x111)

# Step 4: Assemble into a vector, the return
rational_vector = sp.Matrix([r0, r1, r2, r3, r4, r5, r6, r7])
```

Source Code Listing 5.3: A simple Python program creating a vector of rational functions over an array input. This example was generated with ChatGPT.

5.1.6 Bases

A tensor expressed in listing 5.1 always has mathematical meaning, but we cannot assign it to a formula in listing 5.2 unless we select bases. A basis is a collection of linearly independent vectors that span a set; via their linear independence, they allow vectors to be written uniquely as a given array where each array entry corresponds to a given basis element. Given an array, we cannot give it meaning as a tensor unless each entry corresponds to an element of a basis. Further, to produce formulas with inputs, the elements in those inputs only make sense given a basis. Thus, to specify formula generation, we must specify bases, and our formula generator will start with initial mappings from tensors to bases and formulas for those bases.

Our basis language is depicted in listing 5.4. Just as our language for tensors and spaces was grouped by constructions, so is our language for bases. Furthermore, just as each space construction has an inner product, the basis language can construct a particular orthogonal basis in that space, which we call the standard basis. For $\mathbb{R}$, the standard basis is, of course, $\{1\}$. For any other space, we can utilize suitable lists of vectors as a basis. Alternatively, for a dual space, tensor product space, or a tensor sum space, we can construct such lists algorithmically while keeping the label to recognize the structure of the basis. For example, given two spaces, V and W with bases $\{v_i\}$ and $\{w_j\}$, we have $\{v_i \otimes w_j\}$ for $V \otimes W$ and $\{v_i \oplus 0_W\} \cup \{0_V \oplus w_j\}$ for $V \oplus W$. For $V^\star$, we have the dual and primal basis construction from definition 3.1.1. Each of these constructs will transform orthogonal bases to an orthogonal basis and so they transform the standard bases for the components to the standard basis. Finally, for our various subspaces (kernel, image, co-kernel, co-image) of a space V, we always have a projected basis, a basis derived from the QR factorization (co-kernel, image) or SVD factorization (kernel, co-image) realized with a given source and target basis. If the source and target basis are orthogonal (standard), the SVD or QR will produce an orthogonal (standard) basis in the inner products of the sub-spaces. We note that in the case of an injective (surjective) mapping, we can simplify in the case of a co-kernel (image). Finally, in the case of an injective linear mapping $F\colon V \to W$, another option is available for the image space: the image basis is the basis $\{F(b_i)\}$ given the basis $\{b_i\}$ for V.

$\langle \mathit{Basis} \rangle = \{1\}$

$\quad|\quad \text{RawBasis}(\langle \mathit{TensorExpr} \rangle+)$

$\quad|\quad \text{DualBasis}(\langle \mathit{Basis} \rangle)$

$\quad|\quad \text{PrimalBasis}(\langle \mathit{Basis} \rangle)$

$\quad|\quad \langle \mathit{Basis} \rangle \otimes \langle \mathit{Basis} \rangle$

$\quad|\quad \langle \mathit{Basis} \rangle \oplus \langle \mathit{Basis} \rangle$

$\quad|\quad \text{ProjectedBasis}(\langle \mathit{Basis} \rangle, \langle \mathit{Basis} \rangle, \langle \mathit{TensorSpace} \rangle)$

$\quad|\quad \text{ImageBasis}(\langle \mathit{Basis} \rangle, \langle \mathit{TensorSpace} \rangle)$

Source Code Listing 5.4: A grammar to describe bases for tensor spaces.

5.1.6.1 Basis Semantics

We reiterate that the semantics of a basis are simply the list of vectors in the basis. For example, the first two bases in listing 5.4 clearly correspond to lists of vectors. Dual and primal bases were defined in definition 3.1.1. The constructors $\otimes$ and $\oplus$ follow the recipes of the standard bases given above. Finally, the last two simply apply the mappers they are associated to to each element of the source basis.

5.1.6.2 Basis Examples

We consider two examples of bases. First, we consider the vectors $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. These form the standard basis for $\mathbb{R}^2$. For any $a \neq 0$, we also have $f_1 = \begin{bmatrix} a \\ 0 \end{bmatrix}$ and $f_2 = \begin{bmatrix} 0 \\ a \end{bmatrix}$, two formulas with a dependence on a. (To actually specify this in our system, we would specify f_1 and f_2 as vectors given by the above formulas in the basis $\{e_1, e_2\}$). Now, we consider the identity mapping on the space $\mathbb{R}^2$. This is a vector in $(\mathbb{R}^2)^\star \otimes \mathbb{R}^2$, as it is a linear map from $\mathbb{R}^2$ to $\mathbb{R}^2$. If we realize this with our first basis as the source and target, we get the identity matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Despite the formulas for f_1 and f_2, if we use them for source and target, then we get the same identity matrix. Finally, if we considered representing the identity map using the first basis as the source and the second as a target, we get: $\begin{bmatrix} 1/a & 0 \\ 0 & 1/a \end{bmatrix}$ because $\mathrm{id}(e_i) = e_i = 1/a f_i$ for $i = 1, 2$. Though this example is simple, we hope it illustrates that a vector is given meaning by a basis and that the complexity of the meaning (the complexity of the formulas, in terms of the number of statements or the degrees of the involved rational functions), can be useful or surprising based on special properties of the vector (e.g., the identity vector is the identity matrix so long as the source and target basis are the same).

5.1.7 Formula Generation

Formula generation can now be formalized. Given a tensor expression, suppose that we have a mapping (an environment) from the free variables of the expression to pairs of bases and either arrays of numbers or input variables (symbolic inputs). Further, suppose we have a basis for the tensor space that the tensor expression lives in. Formula generation combines the expression, mapping, and basis to produce a program which, given arrays for every input, produces an array that represents the expressions in the given basis under the assumptions of the mapping (that certain variables have certain representations).

An algorithm for formula generation is easy to reason out as a recursive algorithm on the tensor expression and basis expression. For each tensor expression, only a few types of bases can be used, typically separated into those that match the structure of the term and those that do not (and thus require a representation in another basis followed by a change of basis). For our purposes, simpler formulas typically result from the former case, as we avoid extra terms due to the change of basis. Simpler formulas also result from the former case because the basis matches either the universal property or the basis matches the basis dependent property. We loosely work out a few examples.

1. Given $v \otimes w$ and a basis $b_v \otimes b_w$, the algorithm can work mechanically. First, we might produce a formula f_v for v represented in b_v and a formula f_w for w represented in b_w. Second, we can combine the two formulas to a formula for $v \otimes w$ in $b_v \otimes b_w$ into a new one that returns an array of rational function entities corresponding to the tensor product (allowing us to eliminate structural zeros immediately) or by using an Einstein summation notation to compute the tensor product and then returning an array of accesses to the result.

2. Given $v \otimes w$ and a basis of other tensor expressions, we cannot easily represent the term unless $v \otimes w$ is an explicit linear combination of basis entries. Thus, we must utilize our freedom to choose some basis for v and w (e.g., the standard basis or a basis that produces a simple formula for v), proceed as above, and then apply a change of basis in the resulting formula. This is the freedom of the universal construction: any basis built via $\otimes$ will do. Again see definition 3.1.4.

3. Given $v(w)$, evaluation of w against v, we can proceed similarly to our first case (replace tensor product with tensor contraction) if we can represent w and v in a dual primal pair (one is the dual basis of the other, or the other is the primal basis of the dual). Otherwise, we will need to convert the basis of one of the terms. This is where the greater freedom in universal terms below might be useful. A practical example of this that occurs in many FEs is the evaluation of one term v in an image basis of a map J via another term W in a product with the dual basis (this could represent a gradient or Jacobian evaluated against a tangent vector); early conversion of either term to the standard basis would force an unnecessary change of basis, leading to a generated formula $vJJ^{-}W$ as opposed to the simpler vW.

4. Given an inner product, we can return an identity matrix if the basis is the standard basis for the space; otherwise, we must convert to that basis. We have little freedom due to the core connection between the inner product and certain bases.

Provided we have a change of basis, strategies like this work for all tensor expression constructors. With these examples, the overall approach up to formula generation is to recursively build formulas, exploiting where possible the freedom of some constructions to avoid unnecessary change of basis and the corresponding accumulation of complex rational function terms.

An algorithm for change of basis follows a similar structure. In particular, a conversion to and from the standard basis can be computed recursively on the structure of a basis. If we can convert b_0 to and from b'_0 and similarly for b_1 and b'_1, then we can make a formula for the conversion to and from $b_0 \otimes b_1$ and $b'_0 \otimes b'_1$. For a dual or primal basis construction, it is essentially that track conversions to and from a basis because converting such bases involves an inversion, requiring either the to or from construct depending on context. A similar argument holds for almost every basis except a basis of raw tensor expressions. We note that the formula generator and basis conversion algorithm must be mutually recursive on each other due to the presence of the raw basis. If a tensor expression in a raw basis cannot be realized in the standard basis, then the change of basis will fail at this point. If we examine the expressions, almost all expressions are representable in the standard basis via the bottom up approach given that the subterms are representable except for a variable with a representation supplied externally (as opposed to one bound in an abstraction). Thus, the algorithm for change of basis will succeed unless a basis utilizes an externally defined tensor via a variable. To prevent this, we can simply require that any expression used in a basis must be representable in the standard basis.

5.1.7.1 Important Formula Generation Example

We illuminate three examples of formula generation in greater detail to emphasize the interaction of this component with components defined above this. Suppose we have an injective linear transformation $F \in (\mathbb{R}^2)^\star \otimes \mathbb{R}^2$ and a given representation in the standard basis: $F = \begin{bmatrix} F_{00} & F_{01} \\ F_{10} & F_{11} \end{bmatrix}$. Then we can define $V = \text{Image}(F, None, None, True, True)$, marking that the mapping is injective and surjective. Suppose we have a vector $v \in V$ represented in the image basis as $v_R = \begin{bmatrix} 0 & 1 \end{bmatrix}$. In other words, $v = 0F(e_1) + F(e_2)$ where $\{e_1, e_2\}$ is the standard basis for $\mathbb{R}^2$. Suppose we also have a $v' \in V^\star$ with a representation in the dual basis of the image basis given by $v'_R = \begin{bmatrix} 1 & 0 \end{bmatrix}$. Via definition 3.1.1: $v' = F^{-T}(e_1) + 0F^{-T}(e_2)$. If we want to represent the vector $v'(v)$, the duality relationship between the bases means we compute the representation as $v'_R \cdot v_R = 0$. Now suppose in contrast that v was represented in the projected basis so $v = v_R$ directly because the mapping is injective and surjective. Then to compute $v'(v)$, we need to convert one basis to another because one basis is not the primal basis of the other. To write v in the image basis, we need to write v as some other vector in the image of F. We can write this representation as $v_{RR} = F^{-1}(v_R)$ as then $F(v_{RR}) = v_R$. In this situation, the representation of $v'(v)$ is $v'_R \cdot F^{-1}(v_R)$. A rough formula for this situation is given by listing 5.5. Looking at this formula, the complex part is not only the presence of the symbols but also the presence of the inverse. We note that if the situation was reversed and v' was in the projected basis while the other was in the image basis, we could have instead accumulated an F^T.

```python
from sympy import Matrix, MatrixSymbol, Inverse

# 1) 2x2 symbolic array named F
F = MatrixSymbol('F', 2, 2)

# 2) Symbolic inverse of F
F_inv = Inverse(F)

# 3) Basis vectors e1, e2
e1 = Matrix([1, 0])
e2 = Matrix([0, 1])

# 4) Compute e1 dot (F_inv dot e2)
dot_val = (e1.T * F_inv * e2)
```

Source Code Listing 5.5: A formula using an inverse due to a primal-dual basis mismatch. This example was partially generated with ChatGPT.

5.1.8 Miscellaneous: Expressibility, Hacks, and Programmability

We quickly enumerate some helpful observations on the system. Clearly, any tensor operation expressible with traditional Einstein summation notation is expressible within the language. We further note that the tensor aspects of exterior calculus are completely expressible within this framework, utilizing the kernel of the symmetrization operator to define Λ^k. The system can express wedge products, inner products on forms, interior evaluation, Hodge stars, and volume elements. For the Hodge star, though, an inverse is required in the naive approach, which produces messy symbolics. Thus our system also supports a hack: we allow externally defined tensors that are simply a space and a collection of bases formula pairs. This allows us to add the Hodge star via the traditional formula on an orthonormal basis. Similarly, for terms we have simpler representations of, we can always add in known collections of bases and array pairs. Users will not use this feature, as it is a lower level interface to support easy

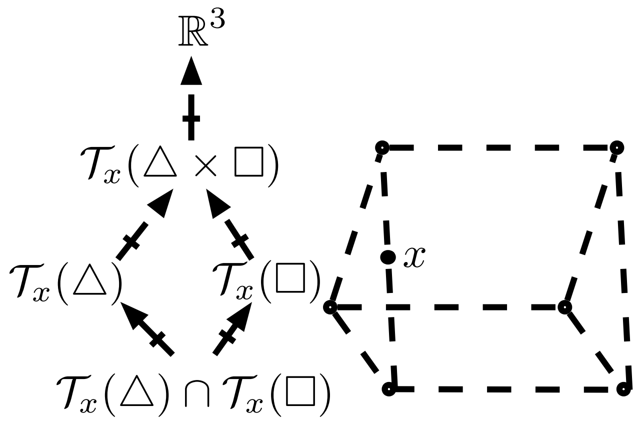

$\mathbb{R}^3$
$\mathcal{T}_x(\triangle \times \square)$
$\mathcal{T}_x(\triangle)$
$\mathcal{T}_x(\square)$
$\mathcal{T}_x(\triangle) \cap \mathcal{T}_x(\square)$
x

Figure 5.1: Various tangent spaces and relationships available at a point x.

extendability without compromising formula generation. We also note the implementation of this system is as an embedded DSL in Python, which allows for meta-programmability. Thus, for example, we can define a general wedge product that produces the right tensor expression via Python code.

5.2 Charts: Geometry for Tensor Expressions at a Point

5.2.1 Putting Geometry into Tensor Expressions: Charts

Our model of tensor expressions currently takes place without geometry, but in practice, tensor calculations can occur at points that are within multiple related geometries. For example, in fig. 5.1, a point x on the edge of a wedge could be used for four different tangent spaces, on an edge, triangle, square, and the overall wedge. From this example, our language, and earlier examples of FEM computation, we can synthesize several requirements: tangent spaces, projections to and from Euclidean space or other tangent spaces, and Riemann metrics (as all spaces must be inner product spaces). We add all of these to our language with minor modifications.

We observe that in a special case of these nested geometries where each space is locally parametrized by some function (a *chart* in the language of differential geometry)

$F\colon \mathbb{R}^m \to \mathbb{R}^n$ at a point $x = F(y)$, our system can already produce and reason about the required objects from a single external requirement: DF_y. The tangent space is the image space of DF_y and cotangent space is the dual (which would be the inverse adjoint image too). Such a space has a natural immersed metric, which corresponds to our choice of inner product for the image space. We already have projections to and from Euclidean space which can be chained together to get between spaces. We also note that mapping from the parameterization tangent space to the parametrized tangent space can be captured via the restriction of DF_y (though we do not expose this to users as we do not want to expose the parameterizations; they are just convenient to supply geometry to tensors and our generated formulas). Obviously, we will be supplying various DF_y for the various geometric regions (e.g., both $DF_y^{\square}, DF_y^{\triangle}$ in fig. 5.1). Overall, though, given a variable representing DF_y, our tensor expressions can represent all the required geometry at a point $x = F(y)$. Moreover, we then benefit from the formula generation infrastructure, especially the image basis for the image space, which enables simplifications based on reference spaces (such as the aforementioned $WJJ^{-1}v = Wv$).

5.2.2 Charts for Users and Types of Cells

Though DF_y is sufficient, we do not think users should deal with explicit parameterization or geometries at a point because this encumbers the user and the automation capabilities of our system. For the purposes of generating formulas, explicit parameterization would constrain the usage of tensor expressions. By changing parameterization or geometries, we can compute useful information about tensor expressions, such as how representations change when we change geometries, which is critical to the FE transformation problem. Similarly, users should not be writing code specialized to a particular parameterization; this flies in the face of coordinate independence or natural ways of dealing with differential geometry. Moreover, it prevents one from using DOFs that can be used in a variety of situations. In the next section on pointwise tensor expressions, we will show how users actually access geometry without referring to a particular chart or geometry, but this will compile down to tensor expression code that uses explicit parameterizations.

In the rest of this section, we will specify how to compute all the DF_y that we need. Thus, we must now specify the types of cells that we support, which is the start of the boundary between user specification of geometric information and internal automated reasoning using geometric information. To simplify the interface and the implementation effort to compute all the DF_y while supporting many geometries, we support cells that are products of simplices. A grammar for these is depicted in fig. 5.2 and a formal definition for product of simplices is definition 5.2.1.

Definition 5.2.1 (Simplex, Product of simplices, Entity)**.** A **simplex** is a set of vertices V with faces $P(V)$, where $P(V)$ is the power set of V, a partially ordered set under $\subset$. Given two simplices with vertices V_1 and V_2 as well as with faces $F_1 \subset P(V_1)$ and $F_2 \subset P(V_2)$, we can define a **product of simplices** as a poset as follows. Note the real key to this construction is that F_1 and F_2 are posets. Let $V = \{v_1 \oplus v_2 \colon v_1 \in V_1, v_2 \in V_2\}$. The set of faces is $F = \{\{(v_1 \oplus v_2) \colon v_1 \in f_1, v_2 \in f_2\} \colon f_1 \in F_1, f_2 \in F_2\}$. The set of faces is ordered by $\subset$ on $P(V)$, making $(F, \subset)$ a poset, which corresponds to a product of simplices. An object is an **entity** if it is either a simplex or product of simplices. Later, we will say an entity is a mesh entity if it is part of some collection of entities called a mesh (see section 7.2.1).

To avoid dealing with actual simplices, we first use standard simplices and then we signify their families with a type.

Definition 5.2.2 ((Geometric) Standard Entities)**.** The standard d simplex uses the vertices $\{0, e_1, \ldots, e_d\}$. The standard product of simplices is the product of the standard simplices.

We note that in some cases, especially the discrete aspects of meshes and data on meshes, we require a more topological representation of the standard entities that does not deal with Euclidean space. We note this mainly to put the content in chapter 7 on a less geometric foundation, but the two definitions are essentially the same:

$$\begin{aligned} &(\textit{topology type}) \quad C ::= \Delta^n \mid C_1 \times C_2 \\ &\qquad\qquad\qquad\quad\ n \ \in \mathbb{N}_{\geq 0} \end{aligned}$$

Figure 5.2: The grammar for cells that we support. Since $\times$ is associative and commutative, there are only seven possible primitives in dimension 3 or less. These are the simplices: points (Δ^0), edges (Δ^1), triangles (Δ^2), and tetrahedra (Δ^3); and product cells: quads ($\Delta^{1,1} = \Delta^1 \times \Delta^1$), hexes ($\Delta^{1,1,1} == \Delta^1 \times \Delta^1 \times \Delta^1$), and triangular prisms ($\Delta^{2,1}$).

Definition 5.2.3 ((Topological) Standard Entities)**.** The standard d simplex uses the vertices $\{0, 1, \ldots, d\}$. The standard product of simplices is the product of the standard simplices.

Additionally, this gives us an easy way to say what the type of an entity is:

Definition 5.2.4 (Type of an Entity)**.** An entity has a topological type (in the sense of fig. 5.2) if and only if it is isomorphic (as a poset) to the unique standard entity of that type.

With this definition, we can also formally define a chart for the purposes of our implementation.

Definition 5.2.5 (Chart, Topological Type of a Chart)**.** A chart is a triple (S_1, S_2, F) where S_i for $i = 0, 1$ are two entities whose vertices are either real numbers or formulas and F is a formula describing an affine map between the vertices of S_1 into the vertices of S_2. In particular, for every vertex v in S_1, we require there be a vertex v' in S_2 so that $F(v) = v'$. The usage of formulas for values and the transformation is clarified in section 5.2.3. The topological type of a chart is the topological type of S_1.

Definition 5.2.6 (Reference and World Space)**.** Given a chart (S_1, S_2, F), we call S_1 the reference space or domain while we call S_2 the world space or domain.

5.2.2.1 Topological Dimension and Type Ordering

Any entity has a topological dimension, defined recursively. The topological dimension of a d simplex is d and the topological dimension of a product of simplices is the sum of the dimensions of the constituent simplices. This allows us to partially order the topology types via the dimensions of their standard entities. We will utilize this fact implicitly and frequently.

5.2.3 Chart Internals: Symbolic and Numerical Charts

We wish to emphasize a critical consequence of supplying geometry at x via supplying various DF_y. In particular, if we supply an external tensor expression, we must provide a *formula* for it. Thus, DF_y can be any bit of code that fits into the grammar for the formula. Therefore, DF_y can be an array of numbers, which would allow us to simplify a formula based exactly on those numbers, or DF_y can be another input. For example, when we evaluate how a FE behaves on a reference element or a permutation of it, we would use an array of numbers since we know the exact geometry, but when we compute the symbolic Vandermonde matrix, we obviously do not know the geometry, and DF_y is just an array of symbols, waiting to be bound to some numbers. Similarly, if we compute a matrix corresponding to an operator on a reference element, we can get numerical values and use a numerical chart, but to generate code to do this on a mesh, we need to generate a formula with symbolic values for the yet unknown geometry. Thus, our choice of geometry interface allows for our tensor expression to formula pipeline to be used for a wide variety of purposes.

5.2.4 Chart Internals: Computing All Charts Given a Cell Type

For the purposes of this paper, we still need to provide parameterizations for cells that are products of simplices (i.e., products of affine maps). For simplices, we can easily compute a mapping between any two given by a list of vertices. For a product of k simplices, we can compute k affine maps using facets and then take a product of the affine maps: given $F_1(x) = A_1 x + b_1$ and $F_2(y) = A_2 y + b_2$, we can define $F(z) = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} z + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$. Given cell types, we can compute all the parameterizations that we need in advance. For every cell type present in a mesh, we need a parameterization of it (to facilitate computations over any type of cell) as well as a parameterization of all sub-cells in that cell (to facilitate reasoning about a tangent space positioned inside another tangent space). Furthermore, to ensure consistent calculations between shared cells, we need to consider every valid[1] permutation of the vertices of the cells. Computing these parameterizations allows us to figure out how DOFs and operators change when cells are permuted to ensure shared cells are consistently ordered. Finally, we need to compute a parameterization that is symbolic (uses symbolic entries) and one that is the numeric chart of the reference cell. By enumerating all of these possibilities, we can compute all parameterizations that we need in advance. We depict an example of this in fig. 5.3 where we compute various parameterizations and the closure under composition and pseudo-inverses yields all parameterizations that we need.

5.2.5 Chart Systems: Top Charts with Permutation Data

By enumerating all possible charts from one single chart or geometry, we simplify passing around many possible parameterizations, but in some cases we want to pick out specific subsets. In particular, we usually want to pick out specific permutations for each entity in the initial geometry. Given a simple (V, F), we can assign a particular permutation σ_f for every $f \in F$. We can package these permutations into a chart and specify that when we generate other charts, these entities should have that permutation applied to their reference space. This is necessary for one system-specific automation, detailed in section 6.4, which removes many conditions on the ordering of mesh entities

[1]For simplices, this is simply all permutations, but for products of k cells this is the semi-direct product of the direct product of the permutations for each symmetry group with the group $\mathbb{Z}/k\mathbb{Z}$. For example, the dihedral group of order 4 is the semi-direct product of $\mathbb{Z}/2\mathbb{Z}$ and $\mathbb{Z}/2\mathbb{Z}$ and the octahedral group is the semi-direct product of $(\mathbb{Z}/2\mathbb{Z})^3$ with S_3.

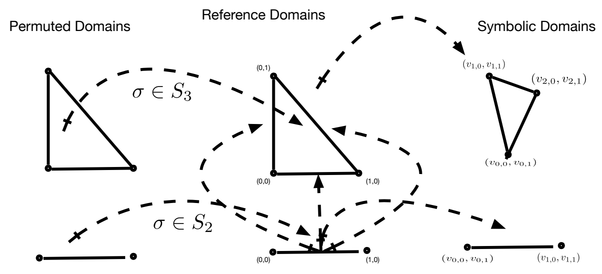


Figure 5.3: An illustration of all the parameterization needed for a triangular mesh in $\mathbb{R}^2$, modulo pseudo-inverses, composition, and vertices.

5.3 Pointwise Tensor Expressions and Fields on Manifolds

5.3.1 Design Rationale

To express virtually all DOFs and operators, users need to use tensor expressions pointwise, operating directly on fields, derivatives, and pointwise-varying geometric quantities. Mathematically, this is to say that most DOFs and operators utilize expressions of the form $F(x, u(x), Du(x), \ldots, t(x), \ldots)$ where F only has tensor operations, x is a point, u is some field, Du is the derivative, and $t(x)$ is geometric information such as the tangent vector. We will achieve a good balance between automation and expression by carefully reasoning about the exposure of geometric content to the user (geometries and parameterization) in the arguments, x, u, Du, t. To balance expression with automatic generation of formulas in our tensor expression system, our model of pointwise expressions must ultimately produce tensor expressions with explicit parameterizations utilized in x, u, Du, t, though allowing users to produce these explicitly is undesirable for several reasons, including more verbose and duplicated code. For example, automation of FEs gains from pointwise tensor expressions where we can change the parameterizations, as the various options (e.g., symbolic vs numerical charts) tend to correspond to different requirements for implementing an element. Similarly, if a user defines a DOF for a tensor field defined on an edge, ideally automation should be able to lift that to a reasonable DOF for a field defined on a triangle or square. Also, when there are multiple choices of geometries, enumeration of the possible options is useful. For example, to write down a higher order Lagrange element on a triangle, it is convenient to write down the DOFs defined on the edge only once as abstract DOFs defined on some edge, as opposed to writing them down three times for each edge. Thus, our language for pointwise tensor expressions exposes pointwise tensor expressions with only limited control over the types of geometries and parameterizations involved while our system for reasoning about pointwise tensor expressions automatically enumerates the various ways pointwise tensor expressions can utilize explicit geometries, parameterizations, and fields.

5.3.2 Grammar for Pointwise Tensor Expressions

Our grammar for pointwise tensor expressions is in listing 5.6 and seeks to offer an interface to geometry for tensor expressions occurring at a point. The first nodes in the grammar expose various types of geometries to the user: simplices or products of simplices as in fig. 5.2.

The next node, the chart type, offers control over the types of geometries that might be used, including the chart for a top cell (the largest topology types of entities in an unknown mesh), a chart that could be a boundary of the largest topology types, a chart of an object of a particular topological dimension, or a chart for a specific geometric object. These two nodes summarize the extent to which users can explicitly pair a (part of) a pointwise tensor expression to a geometry.

The rest of the grammar for pointwise tensor expressions seeks to provide geometric objects and fields to tensor expressions based on the limited geometric interface, facilitating the expression of DOFs and operators that utilize geometry pointwise without preventing automation by tying implementations to a geometry. The grammar extends the tensor space grammar with a tangent space and Euclidean space node that depend on a chart type so that these can be utilized in field types. Field types take a chart type (the geometry on which the field is defined with a default choice of the cell), a tensor space that depends only on that chart type (the default is a scalar field) and that makes limited use of spaces that depend on vectors[2], and a boolean indicating if derivatives of the field should be in dual spaces of Euclidean spaces or cotangent spaces.[3] Finally, a pointwise tensor expression takes a list of arguments that are either fields or charts. We note that default arguments are allowed via the fields described below, which is a useful convenience notion to access geometric information computed earlier (such as cell size indicators). Tensor expressions are extended to utilize these arguments to provide tangent vectors, cotangent vectors, normal vectors, access to fields and their derivatives, and the point x. All of these, except for the derivative and the point x, can be differentiated by the last optional integer argument, which ensures the language is closed under differentiation. Since many tensor expressions utilize a space argument, we also extend spaces to utilize these arguments (so the user can use the tangent space associated to a chart argument) and add a utility to get the space that an expression lives in.[4]. Finally, a pointwise tensor expression consumes a list of arguments, linear and non-linear arguments, to produce a field with a particular tensor expression.[5]

$$
\begin{array}{l}
\langle \mathit{tdim} \rangle = \langle \mathit{int} \rangle \ \langle \mathit{topType} \rangle = \mathrm{Simplex}(\langle \mathit{tdim} \rangle) \\
\quad | \quad \mathrm{Product}(+\langle \mathit{topType} \rangle) \ \langle \mathit{chartTy} \rangle = \mathrm{TopChart} \\
\quad | \quad \mathrm{BoundaryChart} \\
\quad | \quad \mathrm{Chart}(\langle \mathit{tdim} \rangle) \\
\quad | \quad \mathrm{Chart}(\langle \mathit{topType} \rangle)
\end{array}
$$

[2]Of the four, we only allow kernel, and the kernel cannot explicitly use any chart dependent spaces except as the super space.

[3]Practically, this is the difference between a field defined on $N \subset \mathbb{R}^k$ taking values in a tensor space V having a first derivative in in $\mathcal{T}_x N^\star \otimes V$ vs $(\mathbb{R}^k)^\star \otimes V$. This models derivatives defined multiply vs. singly on shared mesh entities/internal boundaries

[4]In practice, the Python embedding implementation does not really need this, as we provide methods to tensor expression objects such as dot and these query arguments with spaceOf. However, the availability of such information is a tricky issue that deserves attention.

[5]In actuality, this is Python code and not just a tensor expression. These will be equivalent if the code strictly uses an interface that simply elaborates on our grammars. In practice this means that one has a bit more flexibility, though. We also distinguish linear arguments and non-linear ones via position and keyword only arguments.

```
⟨TensorSpace⟩ = TangentSpace(⟨chartTy⟩)
  | TangentSpace(⟨str⟩)
  | EuclideanSpace(⟨chartTy⟩)
  | EuclideanSpace(⟨str⟩)
  | spaceOf(⟨tensorExpr⟩)

⟨fieldTy⟩ = field(?⟨chartType⟩, ?⟨tensorSpace⟩, ?⟨bool⟩)

⟨argTy⟩ = ⟨fieldTy⟩
  | ⟨chartTy⟩ ⟨arg⟩ = ⟨str⟩ : ⟨argTy⟩
  | ⟨str⟩ : ⟨argTy⟩ = ⟨defaultArg⟩

⟨TensorExpr⟩ = TangentVectorBasis(⟨str⟩, ⟨int⟩, ?⟨int⟩)
  | CoTangentVectorBasis(⟨str⟩, ⟨int⟩, ?⟨int⟩)
  | NormalVector(⟨str⟩, ⟨int⟩, ?⟨int⟩)
  | CoNormalVector(⟨str⟩, ⟨int⟩, ?⟨int⟩)
  | D(⟨TensorExpr⟩)
  | X()
  | FieldAccess(⟨str⟩, ?⟨int⟩)

⟨PointwiseTensorExpr⟩ = Pointwise(*⟨arg⟩, *⟨arg⟩, ?⟨fieldTy⟩, ⟨TensorExpr⟩)
```

Source Code Listing 5.6: A grammar to describe pointwise tensor expressions on manifolds, operating on tensor fields.

5.3.3 Fields: Reference and World Space Formula Queries

At this stage a field is a rather simple interface intended to ensure that formulas can be generated and that the simplest possible information can be supplied to the pointwise tensor expression for the formula pipeline. For fields, we achieve this aim with two means, old and new: nu supplying tensors, formulas and bases so simplifications can occur, and by, utilizing both reference and world space coordinates to simplify the outputs. For an example of the latter means, many polynomial spaces can best be sampled via reference space followed by a tensor pullback on the outputs whereas some inputs can only be sampled in world space. Thus a field must satisfy a simple interface: given a chart c of the right type, and a formula for the reference space and world space of a point x, called x_r and $c(x_r)$, and some number of derivatives, a field should supply the field values and derivative values at x. In particular, these values should be supplied either as a formula and basis pair or as a tensor expression, both of which should only depend on x_r, $c(x_r)$, and the chart. We stress that this interface is really no more than that and that this facilitates other upstream methods to provide fields of their own: e.g., externally defined functions, compiled pointwise expressions with no field inputs, or basis functions defined with FEs.

5.3.4 Single Compilation of Pointwise Tensor Expression

We now have all the ingredients to compile a pointwise tensor expression down to a formula at some point x (supplied as $x_r, c(x_r)$ as we could always supply a formula for one or the other), but we must precisely match fields to field arguments chart arguments to charts. Given a pointwise tensor expression, suppose that for each argument we provided the following data:

1. If the argument is a chart type, we supply a chart that matches this in the sense of definition 5.2.5.
2. If the argument is a field, we supply a chart that matches a chart type and use this to provide the data of a field.

With this information, we can almost convert a pointwise tensor expression into a tensor expression and then a formula. However, we must work in stages to deal with points and the derivatives. In particular, we must proceed as follows:

1. Using the chart for each argument, we can create a tensor space for it.

2. We can now replace every tangent space or Euclidean space in the pointwise tensor expression with a pure tensor expression space.

3. Using a symbolic differentiation procedure, we can now apply all derivatives, pushing all derivatives down to derivatives of the pointwise tensor expression values (i.e., to derivatives to tensor expressions in listing 5.6 but not listing 5.1).

4. Using x_r and every chart c', we can provide a formula for a point $x_{r,c'}$ that represents x in the reference coordinates of that chart.

5. We can now collect the derivatives that we require from each argument and use them with the charts and $\{x_{r,c'}\}$ to query fields for tensor expressions or formulas plus bases for each argument.

6. With the required inputs, our pointwise tensor expression can become a standard tensor expression by substituting information from the previous step for each argument. For linear arguments, we use abstractions and evaluation to introduce the value, whereas for non-linear arguments, we introduce a free variable bound with a definition in terms of the formula and basis in an environment for tensor expression evaluation.

7. The pointwise tensor expression has become a tensor expression that can be compiled down to a formula.

Our single compilation procedure is fine for producing a formula, but it requires too much from the user: matching a chart and field precisely to every argument. We now simplify things via two automations: adapting pointwise tensor expressions to new fields and automatically matching charts to arguments in reasonable ways via topological enumeration.

5.3.5 Adapting Pointwise Tensor Expressions to Fields

To facilitate reuse of tensor expressions in different contexts, we provide a way to automatically adapt a tensor expression to take a different field input. We can change a tensor expression to use a new field input by changing the field type in two ways:

1. The tangent and Euclidean spaces in a field type can be arbitrarily interchanged.
2. The chart types in the field type (including the tensor space in the field type) can be replaced with a larger chart type.

The first change can be accomplished by replacing the old field accesses with a new one followed by a generalized pullback (as specified in section 5.1.4). The second change can be accomplished by adding a new chart argument for the old chart type, replacing the field type to use the new chart type, and then replacing old field accesses to new ones with pullbacks that map the new tensor spaces to the old tensor spaces.

5.3.6 Topological Enumeration of Pointwise Tensor Expressions

We can now explicitly define topological enumeration of pointwise tensor expressions. Pointwise tensor expressions are associated to a number of chart types via their arguments. If we are given a collection of charts, we can use these to supply actual geometric information and make a tensor expression by pairing each argument with a chart. A simple principle defines the valid pairings and therefore the enumeration: since a pointwise expression takes the form $F(x, u(x), Du(x), \ldots, t(x), \ldots)$, then x must be in all the charts. For example, in the case of $F(x, u(x), v(x))$, then x should be in the domain of both u and v. By formalizing this notion into a constraint on pairs of chart types from a pointwise tensor expression to charts, we define validity of a pairing and the enumeration, which is the essence of the automation that we offer for interpreting pointwise tensor expressions as collections of formulas.

Definition 5.3.1 (Valid Pairings of Pointwise Arguments to charts)**.** Suppose we are given a simplex S_0 and a chart (S_0^r, S_0, F_0), which we call the top chart. Suppose we have a collection of charts $C_i = (S_i^r, S_i, F_i)$ with $S_i \cap S_0 \neq \emptyset$ for $i = 1, \ldots, C$. (In other words, the charts all parametrize geometries inside or related to S_0). Further, suppose we are given a pointwise tensor expression whose K arguments each use (or are) a chart type c_i. Finally, suppose the pointwise tensor expression has an output that uses an argument c_{C+1}. A **pairing** of chart arguments to charts is a mapping $f\colon \{1,\dots,K\} \to \{0,\dots,C\}$ such that the type of $S_{f(i)}$ is c_i. For the top chart and the boundary charts, this is defined by the type of S_0. A pairing is **valid** if

1. whenever $c_i = c_j$ and $c_i < c_{C+1}$, then $S_{f(i)} = S_{f(j)}$
2. whenever $i \neq i'$, either $S_{f(i)} \subseteq S_{f(i')}$ or $S_{f(i')} \subseteq S_{f(i)}$.

In other words, though different charts might be used for the same simplex/product of simplices, arguments of the same chart type should target the same simplex or product of simplices whenever these charts use topological types less than the output topological type (the domain of x). The content of this restriction is that whenever two arguments of the same type appear and are less than the output type, the fields they define have the exact same domains.

We consider four rough examples.

1. Suppose we are given an operator on the top chart with arguments from the top chart with only one chart for the top chart. Then we only get one output, which makes sense for the assembly of operators.
2. Suppose we are given a pointwise tensor expression that utilizes just an edge field (e.g., a Lagrange DOF or a boundary integral) which we attempt to pair with a chart for every entity in a triangle. We get three implementations, which is how we would like to replicate a DOF for an edge Lagrange element in a triangle. Another interpretation of this can be gleaned from our adaptation of field types to larger charts: if we have an expression that consumes an edge field, we can adapt it to the face field and also add an edge chart so that we must enumerate ways that face fields could interact with an edge chart.

3. Suppose we are given a pointwise tensor expression that utilizes a vertex, edge, and triangle chart, which we attempt to pair with a chart for every entity in a triangle: we get six implementations. These represent the six ways a vector field on a cell could be paired with an edge tangent vector and a vertex position. Though this example feels unnecessary, this situation can occur: a cell field might be restricted to an edge tangent to be evaluated at a vertex in exterior calculus DOFs, and there are six valid ways to do this (e.g., in Nédélec DOFs of the second kind).

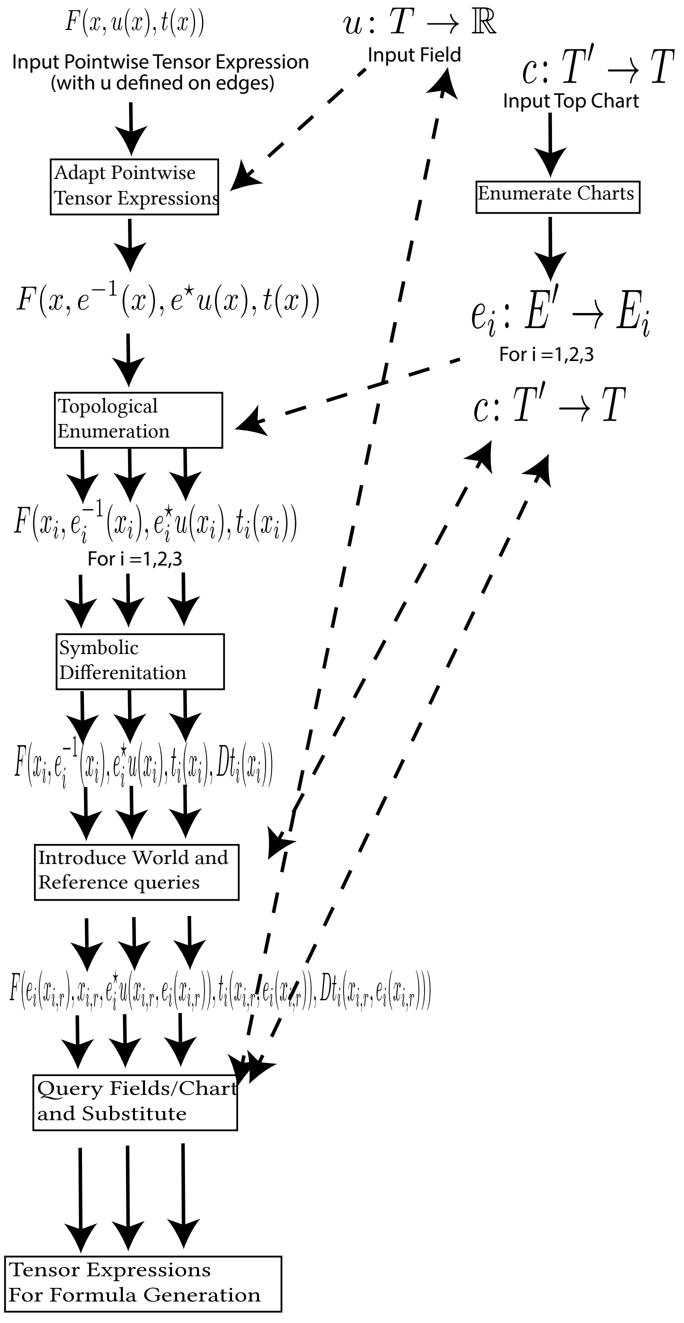
4. Suppose we are given a pointwise tensor expression that utilizes an edge chart and two triangle charts, but the output is an edge. Suppose we provide the charts for two triangles that share one edge. Then we enumerate four possible options; this closely mirrors the structure of DG edge integrals.

5.3.7 Full Pipeline for Compilation of Pointwise Tensor Expressions

We now have all the ingredients to spell out the full pipeline of producing all relevant formulas from a pointwise tensor expression and a single chart parameterization of the local cell where the computation should occur. We describe this graphically in fig. 5.4. The essential strength of this pipeline is the ability to take minimal inputs, restricted to coordinate-free tensor expressions with fairly unrestricted geometric information, and then generate exactly the collection of expression that various downstream automation tasks require.

5.3.7.1 Example

We now present an example that aims to illustrate concretely the essentials of each process without focusing on needless details. In particular, we are going to present fields and pointwise expressions as concrete and specific here though they are not. Similarly, we will return a more abstract expression rather than the specific pointwise tensor expression. We seek to emphasize and accurately present how these move through the pipeline. Let us enumerate three inputs:



$F(x, u(x), t(x))$
Input Pointwise Tensor Expression (with u defined on edges)
$u\colon T \to \mathbb{R}$
Input Field
$c\colon T' \to T$
Input Top Chart
Adapt Pointwise Tensor Expressions
Enumerate Charts
$F(x, e^{-1}(x), e^{\star}u(x), t(x))$
$e_i\colon E' \to E_i$
For i =1,2,3
$c\colon T' \to T$
Topological Enumeration
$F(x_i, e_i^{-1}(x_i), e_i^{\star}u(x_i), t_i(x_i))$
For i =1,2,3
Symbolic Differenitation
$F(x_i, e_i^{-1}(x_i), e_i^{\star}u(x_i), t_i(x_i), Dt_i(x_i))$
Introduce World and Reference queries
$F(e_i(x_{i,r}), x_{i,r}, e_i^{\star}u(x_{i,r}, e_i(x_{i,r})), t_i(x_{i,r}, e_i(x_{i,r})), Dt_i(x_{i,r}, e_i(x_{i,r})))$
Query Fields/Chart and Substitute
Tensor Expressions For Formula Generation

Figure 5.4: A pointwise expression goes through the pointwise expression compilation pipeline given input fields and a chart. The expression takes an edge chart, an edge field, and some geometry. Since the field is defined on an entire triangle, the expression adds a new chart and projects a triangle field to the edge chart. After charts are generated from the top chart, we enumerate all the pairings of pointwise expressions with charts via topological enumeration. Then we apply symbolic differentiation, resulting in some derivative queries. Finally, we query expressions, formulas, and bases from the input fields and charts to create tensor expressions for formula generation.

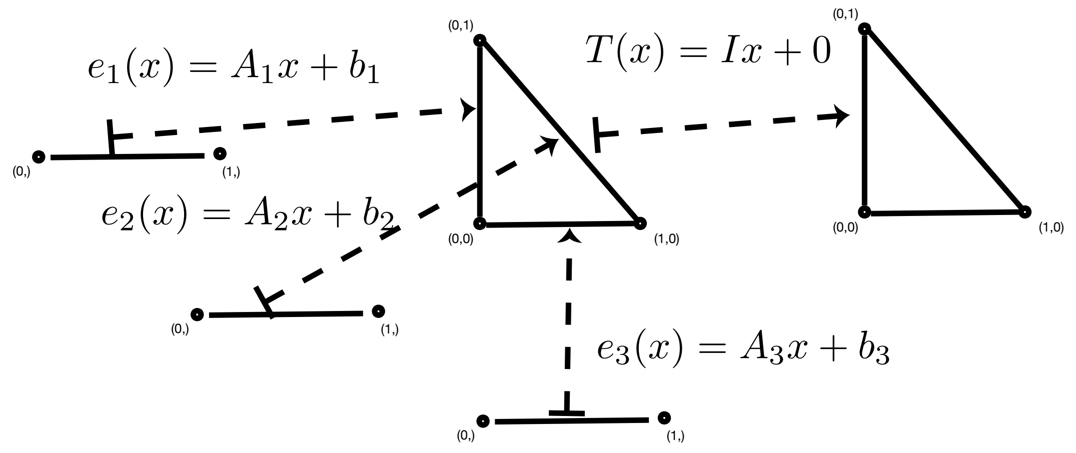


Figure 5.5: An example of enumerated charts for an example.

1. Suppose we have a pointwise tensor expression on an unknown edge: $F(x, u(x)) = x * (\nabla u(x))^2$. Note that since we are on an edge, we can square the derivative since it is in a one dimensional space. Further, this is associated to an unknown edge with a chart $e(x_r) = Ax_r + b$ where $x_r \in [0, 1]$, A is a vector and b is a vector.

2. Now for our other inputs, we suppose that $u(a, b) = ab$ on a *triangle*.

3. Finally, our top chart is the triangle of the unit triangle: a very concrete geometry, parameterized by itself via $T(x) = I(x) + 0$.

To get visual intuition, we first consider the chart enumeration. We have visually depicted this in fig. 5.5. Therein we see the triangle as well as parameterizations of each edge, $e_i(x) = A_ix + b_i$ for $i = 1, 2, 3$. We can practically compute:

1. $A_1 = \begin{bmatrix} 0 & 1 \end{bmatrix}$ and $b_1 = \begin{bmatrix} 0 & 0 \end{bmatrix}$.

2. $A_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}$ and $b_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$.

3. $A_3 = \begin{bmatrix} 1 & 0 \end{bmatrix}$ and $b_3 = \begin{bmatrix} 0 & 0 \end{bmatrix}$.

Now, we consider adapting the expression F to the triangle. This means we have to use the yet unknown edge chart to rewrite F to

$$F_e(e(x_r), x_r, u(e(x_r))) = e(x_r) * (\nabla u(e(x_r)))^2.$$

Note that we add a point x_r in the reference interval $[0, 1]$, get the point x as $e(x_r)$ in the triangle and since u is defined on a triangle, use the point in the triangle to evaluate u. Now, we topologically enumerate. Since there are three edges in a triangle, we must consider all of them for $i = 1, 2, 3$. We get

$$F_{e_i}(e_i(x_r), x_r, u(e_i(x_r))) = e(x_r) * (\nabla u(e_i(x_r)))^2.$$

Then we symbolically differentiate, which is to say we write the expression so it takes in the derivatives separately. With one application of the chain rule and one lifting to an argument, we get

$$F_{e_i}(x_r, e_i(x_r), De_i(x_r), u(e_i(x_r)), \nabla u(e_i(x_r)) = e(x_r) * (De_i(x_r) \cdot \nabla u(e_i(x_r)))^2.$$

Then to prepare to query fields, we add queries to the reference and world space of the inputs. To do this, we need to use the reference space of the triangle (in this case, trivially the original space) so that we can query u in its reference space. We introduce $x_{r,i} = T^{-1}(e_i(x_r))$, the reference position of the point $e_i(x_r)$ and rewrite arguments to u:

$$F_{e_i}(x_r, e_i(x_r), x_{i,r}, De_i(x_r), u(e_i(x_r), x_{i,r}), \nabla u(e_i(x_r), x_{i,r})) = e(x_r)*(De_i(x_r)\cdot\nabla u(e_i(x_r), x_{i,r}))^2.$$

Finally, we can substitute in the actual charts and values to get something that depends on just x and charts:

$$F(x_r, A_i, b_i, I) = (A_i(x_r) + b_i) * (A_i^T \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot I^{-1}(A_i(x_r) + b_i))^2$$

where we used that $\nabla u(a, b) = (b, a)$ as $u(a, b) = ab$.

5.4 Integrated Pointwise Tensor Expressions on Manifolds (IPTEM)

5.4.1 Design Rationale and Semantics

Given pointwise tensor expressions on manifolds, DOFs and operators are missing one component: integration. We augment this concept via reductions as a general mechanism to combine many pointwise evaluations to a single scalar and then store that scalar appropriately. With these concepts, we are primarily concerned with expression, though the implementation of integration requires some care. Our mechanism facilitates many useful utilities beyond integration: pointwise evaluations (for DOFs), combinations of topologically enumerated expressions (e.g., summing expressions of volumes across boundaries to produce perimeters for cell size indicators or a DOF defined on the boundary of a cell), and DOFs of different continuities (for DG spaces).

Our grammar is provided in listing 5.7. We provide integrals or pointwise evaluations on topologies of a known type via a barycentric coordinate. For the purposes of topological enumeration, the integral counts as another chart argument, similar to augmenting pointwise tensor expressions with fields. This leads to the need for reductions.

We observe that topological enumeration could create multiple integrals per integration domain so a default reduction is chosen to produce one integral: sum for integrals and choiceAllEq[6] for pointwise evaluations. However, we allow users to override this behavior

[6]Pick one value out of the possible options, but check that they are all identical.

with reductions as this has utility (e.g., a DOF combining the midpoints of every boundary face of a cell via some linear reduction). Moreover, we find that this allows users to control how expressions should be treated on a mesh. A reduction combines multiple values that emerge from topological enumeration, from multiple field inputs (e.g., from a FEM basis), or even from multiple geometries/basis functions on a mesh (e.g., in section 2.4). A specific reduction with a specified operation can apply to multiple values emerging from a single argument (or by default from multiple integrals) to a pointwise expression or from all values. Integrated pointwise tensor expressions can take multiple reductions and apply them in order. Finally, to support DG or permitter calculations, integrated pointwise tensor expressions can specify a final chart type that indicates if a value should be computed uniquely per this chart type. For the purposes of topological enumeration, this is yet another chart, though we require it to be greater than or equal to all other charts in our expression, but we mainly provide it to enable DG methods in a context insensitive manner.

$$
\begin{aligned}
\langle \mathit{integral} \rangle \;=\;& \text{PointEval}(\langle \mathit{topTy} \rangle, +\langle \mathit{float} \rangle) \\
\mid\;& \text{Integral}(\langle \mathit{chartTy} \rangle) \\
\langle \mathit{op} \rangle \;=\;& \text{min} \\
\mid\;& \text{max} \\
\mid\;& \text{sum} \\
\mid\;& \text{difference} \\
\mid\;& \text{avg} \\
\mid\;& \text{choice} \\
\mid\;& \text{choiceAllEq} \\
\langle \mathit{over} \rangle \;=\;& \text{LocalTop} \\
\mid\;& \text{LocalBasis} \\
\mid\;& \text{GlobalTop} \\
\mid\;& \text{GlobalBasis}
\end{aligned}
$$

```
  |  Top
  |  Basis
  |  ALL
⟨argref⟩ = ⟨str⟩
  |  ALL
  |  ⟨chartTy⟩
⟨reduction⟩ = Argument(⟨argref⟩, ⟨op⟩, ?⟨over⟩)
⟨IntegratedPointwiseTensorExpr⟩ = IPTX(⟨PointwiseTensorExpr⟩,            ⟨integral⟩,
    *⟨reduction⟩, ?⟨chartTy⟩)
```

Source Code Listing 5.7: A grammar to describe integrated pointwise tensor expressions with reductions.

5.4.2 Implementation of Integration

We implement all integrals via quadrature that can integrate d degree polynomials exactly[7]. For simplicity of implementation, this is quite appealing. For a given chart c with a domain that we assume is numeric, a quadrature rule gives us points x_i and weights w_i. With these, we can evaluate the integral of a formula via

$$\sum w_i |\det Dc(x_i)| F(x_i, c(x_i), \dots).$$

For some symbolic programs, we might symbolically compute this sum (in particular if we are computing Vandermonde matrices or if all the formula return values happen to be numeric), but for other purposes we might place this sum in generated code. However, all of this requires us to choose a quadrate degree d, which is potentially error prone and not ergonomic.

[7]For product simplices, it is a product of degrees

5.4.3 Quadrature Degree Estimation

Given a formula acting on inputs[8] that all have some polynomial dependence on the point x, we can estimate the polynomial dependence of the output of the formula on the point. To do so is fairly trivial for any formulas that are polynomial. For non-polynomial functions, we rely on one important simplification or else we just fail: for 0 degree dependence, non-linear functions still have 0 degree dependence. This simplification means that formulas for all operators and DOFs can determine quadrature degrees automatically on affine geometries. When quadrature estimation fails, users at a higher level on the interface (section 7.6) must specify the polynomial dependence of some inputs or simply specify the quadrature degree. We note that this analysis can be interpreted via a simple lattice, $\mathbb{N} \cup \{\infty\}$ with ∞ also being our fail state.

5.4.4 Integration of Forms

Integration typically assumes that a function is scalar, but we will integrate the appropriate forms correctly too. We simply apply the theory of forms as in section 3.1.4, constructing the appropriate tensor expressions for relevant exterior calculus operations. This alleviates the specification of some DOFs. In this case, though, we modify the tensor expression so that simplifications based on this definition of integration can occur.

5.5 Complete Pipeline and Output Formats

5.5.1 Input and Output Format

In presenting the complete pipeline that handles integration, we also offer additional extensions. An easy extension is multiple formulas with the same input and output signatures[9],

[8] A simple extension to the field interface provides an optional polynomial degree estimate.

[9] Input signatures are the arguments and their types modulo default arguments. Output signatures correspond to reductions over these inputs.

but a more important extension is to operate on collections of fields. The final input is multiple IPTEMs with reductions operating on collections of input fields per argument and with a top chart[10] to generate a collection of parameterizations. The final output is a formula for a tensor that has a dimension to index the formulas, a dimension for each input argument that is not reduced over, and a dimension for each chart type that is not reduced over. Additionally, for the purposes of global code generation, we maintain some metadata:

1. for each input argument, the derivatives, points, and charts that we sampled at;
2. the quadrature rule and weights corresponding to each formula;
3. and the integration and storage charts that were used for a given formula.

5.5.2 Complete Pipeline

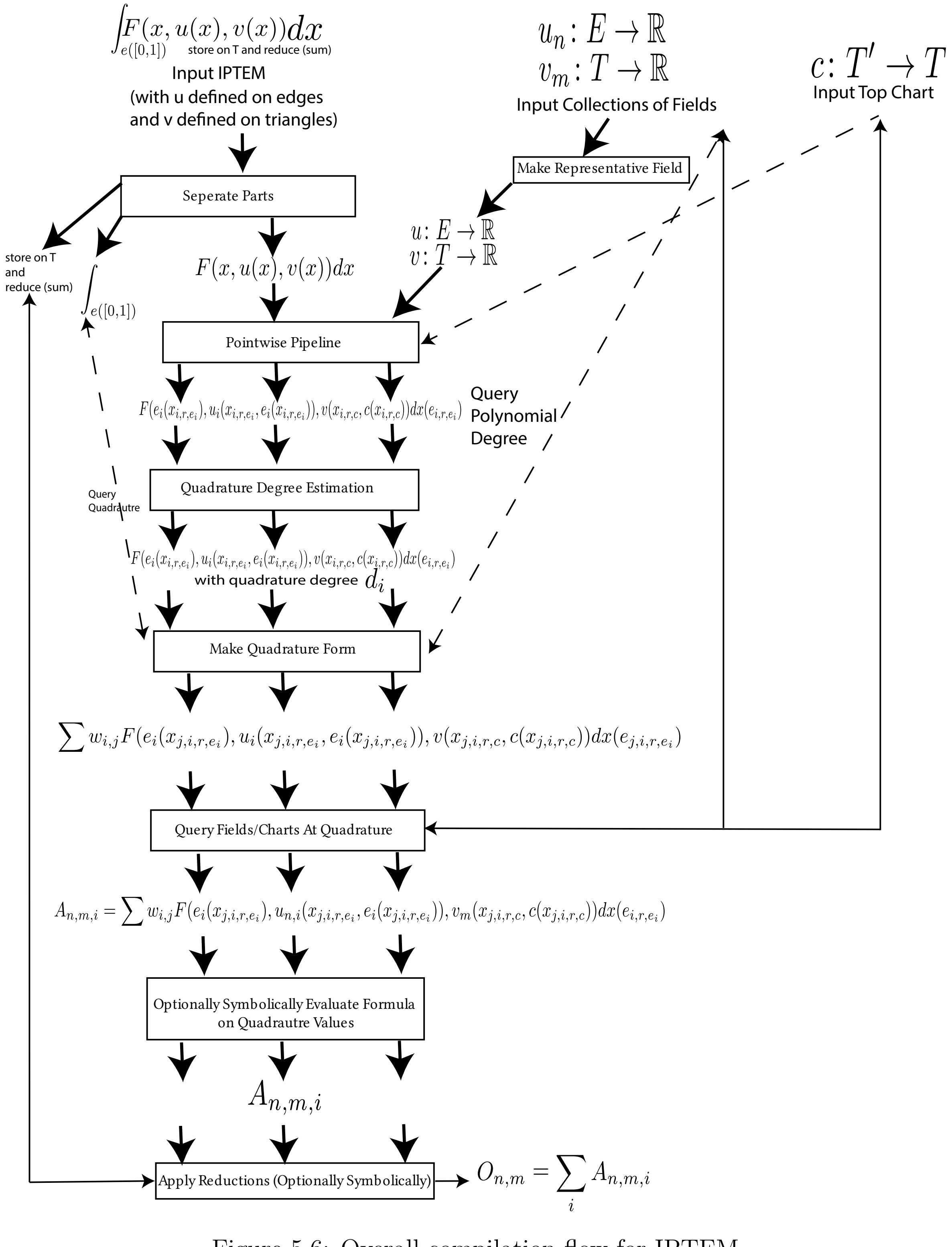
Encapsulating our pointwise pipeline, we can now provide a complete pipeline for a single formula with an integral and reduction, acting on a collection of input fields. The overall pipeline for a single formula and batched inputs is depicted in fig. 5.6, and we discuss it here. The pipeline can simply be batched over multiple formulas. For the first part of the pipeline, we assume that the field collection can be summarized via a single field in the sense that we can provide a formula and basis for the values and derivatives at some symbolic point, potentially depending on other symbolic values such as a reference space value of a field. This is always the case in ElementForge. By using a single representative field, we can then do quadrature degree estimation using integral specification and make a symbolic quadrature sum. Then we can query the fields and charts at all the actual quadrature points to get formulas for each field to substitute into our formula for the pointwise expression. For certain tasks, such as implementing an element, we would symbolically evaluate this formula down so the formula output a rational function, but for other tasks such as computations of operators on meshes, we would generate code that uses Einstein summation notation to

[10]This could be replaced with an arbitrary collection of chart inputs.

perform the sum at runtime. Similarly, we can then apply the reductions, potentially by manipulating the formula or doing the reduction in generated code. Our output is a formula for a tensor with a dimension for each non-reduced over input and each non-reduced over chart type.

5.5.2.1 Extension of Pointwise Example

We extend section 5.3.7.1. In particular, suppose that we actually evaluate fields $u_j(a,b) = c_j ab$ for real constants c_j. Suppose also that our integral is a simple point evaluation at the midpoint of each edge, associated to the point $0.5 \in [0,1]$. Then our final output would become, after integration and usage over all fields, $A_{i,k} = (A_i(0.5) + b_i) * (A_i^T \cdot \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot c_j I^{-1}(A_i(0.5) + b_i))^2$, where i runs over the edges and k runs over the incoming fields. Either index could be eliminated via a reduction over the charts or a reduction over the incoming fields.



$\int_{e([0,1])} F(x,u(x),v(x))dx$
store on T and reduce (sum)
Input IPTEM (with u defined on edges and v defined on triangles)
$u_n\colon E \to \mathbb{R}$
$v_m\colon T \to \mathbb{R}$
Input Collections of Fields
$c\colon T' \to T$
Input Top Chart
Make Representative Field
Seperate Parts
store on T and reduce (sum)
$\int_{e([0,1])}$
$F(x,u(x),v(x))dx$
$u\colon E \to \mathbb{R}$
$v\colon T \to \mathbb{R}$
Pointwise Pipeline
$F(e_i(x_{i,r,e_i}),u_i(x_{i,r,e_i},e_i(x_{i,r,e_i})),v(x_{i,r,c},c(x_{i,r,c}))dx(e_{i,r,e_i})$
Query Polynomial Degree
Quadrature Degree Estimation
Query Quadrautre
$F(e_i(x_{i,r,e_i}),u_i(x_{i,r,e_i},e_i(x_{i,r,e_i})),v(x_{i,r,c},c(x_{i,r,c}))dx(e_{i,r,e_i})$
with quadrature degree d_i
Make Quadrature Form
$\sum w_{i,j}F(e_i(x_{j,i,r,e_i}),u_i(x_{j,i,r,e_i},e_i(x_{j,i,r,e_i})),v(x_{j,i,r,c},c(x_{j,i,r,c}))dx(e_{j,i,r,e_i})$
Query Fields/Charts At Quadrature
$A_{n,m,i} = \sum w_{i,j}F(e_i(x_{j,i,r,e_i}),u_{n,i}(x_{j,i,r,e_i},e_i(x_{j,i,r,e_i})),v_m(x_{j,i,r,c},c(x_{j,i,r,c}))dx(e_{i,r,e_i})$
Optionally Symbolically Evaluate Formula on Quadrautre Values
$A_{n,m,i}$
Apply Reductions (Optionally Symbolically)
$O_{n,m} = \sum_i A_{n,m,i}$

Figure 5.6: Overall compilation flow for IPTEM.

Chapter 6

Software Representation of Finite Elements

6.1 Overall Specification

Our software definition of a FE naturally mirrors the Ciarlet Triple definition, but our definition is carefully constructed to balance expressing many elements against the automatic generation of formulas, specifically via the problem specified in definition 3.2.4. Indeed, our software construct (section 6.1) takes three objects resembling a domain, a function space, and a dual basis, but to solve definition 3.2.4, our software construct represents a class of FEs parametrized by a domain.

```
class FiniteElement:
    V: TopType
    P: PSpace
    Sigma: Sequence[IPTEM]
```

Source Code Listing 6.1: Interface for constructing finite elements.

Our construct contains two familiar objects that represent classes of objects parametrized by a domain: a topological type for the domain and a list of IPTEM for the dual basis functions. While our infrastructure for IPTEM handles many of the problems of representing many dual basis functions and generating formulas for them on symbolic domains, we must resolve similar problems for spaces of functions, especially as they relate to the problems set out in definition 3.2.4. In particular, two function spaces defined similarly on two domains are unlikely to be isomorphic via pullback, which makes computing the Vandermonde matrix impossible or symbolically messy. To understand our representation of classes of polynomial function spaces for FEs (PSpace), we first detail a software interface and algorithms for using the finite element, solving definition 3.2.4. Then we can discuss the requirements on (PSpace) and other useful utilities for elements.

6.1.1 PSpace Interface

Our interface to polynomial spaces parametrized by domains is given in section 6.1.1. At the core, the interface is simple: on a given domain (specified via a parameterization), a given set of points, and a given basis at each point, we can sample some unknown basis of functions and its derivatives.

We are not given the precise set of basis functions for the entire space, but just the basis of the output value at a point, i.e., just a basis for the value space.[1]

We will return to the question of what the basis of the entire space of functions is. For now, the main purpose of this interface is to provide fields as arguments to our compiler, which is why we also include a maximum polynomial degree.

```
class PSpace:
    def tabulateChart(self, chart: Chart, derivatives: int, refPoints: NDArray, basis:
        Optional[Basis] = None) -> Tuple[Formula, Basis]: ...
    def valueSpace(self, chart: Chart) -> Space: ...
    def dim(self) -> int: ...
    def maxDegree(self) -> int: ...
    def asFields(self) -> List[Fields]
    def validChartTypes(self) -> List[ChartType]:...
```

Source Code Listing 6.2: Interface that local spaces, PSpaces, must provide.

[1](Derivatives are taken in the tangent space and thus utilize the image basis featured in listing 5.4).

6.2 Inverse Vandermonde Formula Generation

We now solve definition 3.2.4 up to the details of the space representation. The overall algorithm is depicted in section 6.2. The algorithm consumes a FE, a reference chart, and a symbolic domain given by a chart. The algorithm conducts some validity checks (section 6.2.1), checking that the symbolic chart maps from the range of the reference chart and that the FE type checks. Then the algorithm constructs a Vandermonde matrix (section 6.2.2) and uses this to construct basis functions that can be used on the symbolic chart, exploiting that the matrix is numerical and the basis functions are code. Then using the Vandermonde algorithm again, the algorithm can construct a symbolic Vandermonde matrix from the symbolic chart and the reference function basis functions. Then, a symbolic inversion routine (section 6.2.3) exploits the expected structure of this matrix to produce a formula representing the inverse. The symbolic output matrix provides the symbolic Vandermonde matrix problem (definition 3.2.4), expressing the FE basis functions on the symbolic domain in terms of the reference domain. We prove the correctness of this approach up to an assumption on the space (section 6.2.4) and sketch an alternative approach (section 6.2.4.2) that illuminates the expected output structure.

```
def vandermondeInverse(dofs: FiniteElement, refChart: Chart, symbolicChart: Chart) ->
    Formula:
    checkValid(refChart, symbolicChart, dofs)
    refVander = vandermoneAlgo(dofs.Sigma, refChart, dofs.space.asFields())
    VRef = refVander.asNumericArray()
    assert VRef.dtype = np.float64
    vanderInv = np.linalg.inv(VRef)
    assert vanderInv.shape == (dofs.P.dim(), dofs.P.dim())
    refFields = vanderInv.T.dot(dofs.space.asFields())
    symbolicVander = vandermoneAlgo(dofs.Sigma, symbolicChart, refFields)
    symbolicVanderInv = symbolicInverse(symbolicVander)
    return symbolicVanderInv
```

Source Code Listing 6.3: Overall Algorithm for Vandermonde Calculation.

6.2.1 Validity

For a finite element, `E=FiniteElement(V,P, Sigma)` to be valid, the DOFs must consume elements of `E.P.valueSpace()` and must not use any chart types larger than the chart type `E.V`. Similarly, the chart type `E.V` must be in `E.P.validChartTypes()`. And naturally the DOFs must all be linear in a single non-default argument. The function `checkValid` checks all of this and further checks that the two chart inputs are compatible: they must both map between domains of type `E.V` and the range of the reference space chart should be the domain of the symbolic chart.

6.2.2 Vandermonde Algorithm

The Vandermonde Algorithm is the compilation procedure for a list of IPTEM. The function `vandermoneAlgo(dofs, chart, fields)` runs the procedure outlined in section 5.5. The result will be a matrix that is the number of DOFs[2] by the dimension of the polynomial space, which should match. This illustrates a convenient bit of reuse as this function will be reused in global code generation.

6.2.3 Symbolic Inversion

Our second Vandermonde matrix is symbolic, meaning the formula returns an array of rational numbers, potentially depending on other statements. However, we expect that that this matrix will be roughly block triangular. We will recursively exploit the Schur complement to invert such a matrix:

$$\begin{pmatrix} A & 0 \\ C & D \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ -D^{-1}CA^{-1} & D^{-1} \end{pmatrix}. \tag{6.1}$$

A very close sketch of the algorithm is depicted in section 6.2.3. The algorithm directly inverts small matrices, though it potentially introduces intermediate variables for non-zero

[2]Note that the actual DOFs are defined via topological enumeration.

entries if the rational expressions are too large. The algorithm tries to find a choice for A and C that maximizes the size of the zero block. Then the algorithm recuses on the blocks and recombines them, potentially introducing intermediates for non-zeros in the triple matrix multiplication to avoid very large rational functions. For the complexity of rational functions, we can use the number of non-zero terms in the numerator and the denominator degree. The actual implementation features a few other features: an environmental variable to time out and use a direct inverse (we do not need this for any of our results) and other options to potentially use small linear algebra operations or to tune when rational functions are too complex to directly manipulate. We include these features for the case where an element does not yield a reasonable symbolic Vandermonde matrix so we can fall back to the inefficient option - numerically inverting the Vandermonde C matrix for each cell, with a warning to tell the users that this is going to happen. Inverting such Vandermonde-like matrices per cell is an established practice for some methods where this is the case - such as in polygonal FEMs or virtual element methods [124], [125].

```
def symbolicInverse(V: Formula):
    (mat, stmts) = V
    N = mat.shape[0]
    if N <= 6: # largest reasonable symbolic inverse
        (matInv, stmtsp) = directSymbolicInverse(mat, stmts)
        # Depending on the complexity of rational functions, we might introduce intermediate
            variables
        return Formula(matInv, stmtsp)
    else:
        (blockP, blockQ) = choosePQ(mat)
        A = mat[0:blockP, :][:, 0:blockQ]
        B = mat[0:blockP, :][:, blockQ:]

        D = mat[blockP:, :][:, blockQ:]
        C = mat[blockP:, :][:, 0:blockQ]
        assert allSymbolicZero(B)
        (Ainv, AinvStmts) = symbolicInverse(Formula(A))
        (Dinv, DinvStmts) = symbolicInverse(Formula(D))
        (schur, schurStmts) = tripleMatMul(Dinv ,C, Ainv)
        matInv = bmat([[Ainv, np.zeros(blockP, blockQ)], [-schur, Dinv]])
        return Formula(matInv, stmts + AinvStmts + DinvStmts + schurStmts)
```

Source Code Listing 6.4: Symbolic Inverse Algorithm

6.2.4 Correctness

We now seek to show mathematically what our algorithm does and under what conditions it is correct, specifically with respect to the function spaces. First, we link the formalism of definition 3.2.4 to our code, roughly: Suppose we are given a FE $E := (K, V, P, \Sigma)$, a reference element, and another element $E' := (K', V', P', \Sigma')$ such that K is diffeomorphic to K'. In the language of definition 3.2.4, we are computing the pullback of the primal basis functions of E' in terms of E. Our algorithm, depicted in section 6.2 pulls the reference space basis functions of E to the world element E' via the creation of the reference space basis (lines 3-8), enabling the sampling of these fields from the world space (in line 9). This is to say we use the pullback of the reference space basis as a basis for the world space function space. Then in the last call to the Vandermonde matrix creation algorithm (line 9), we evaluate the world space dual basis on this basis. The inverse transpose then represents the pullback of the world space basis in terms of the reference space basis.

We now prove that we solve definition 3.2.4 under certain conditions, and our proof mirrors the above steps exactly, precisely revealing the ambiguities leading to requirements on our polynomial spaces. We now need some intermediate results, the proofs of which are simple elaborations on ideas in section 3.1.

Proposition 6.2.1. *Let $E := (K, V, P, \Sigma)$ and $E' = (K', V', P', \Sigma')$ be finite elements where V and V' are compatible (p, q) tensor spaces, i.e., that they are subspaces of $\mathcal{T}(K)^{(p,q)}$ and $\mathcal{T}(K')^{(p,q)}$ respectively. Let $T \in C^1(K, K')$ have an inverse in $C^1(K', K)$. Then the following are true in $\mathcal{F}(K, V)$:*

1. $T_\star \circ T_\star^{-1} = Id$

2. $T^\star \circ (T^{-1})^\star = Id$

Further, if $T^\star|_P$ is an isomorphism onto P', then

1. $T_\star|_{P^\star}$ is an isomorphism onto $(P')^\star$.

2. For any $\phi \in P^\star$ and any $f \in P$,

$$T_\star(\phi)((T^{-1})^\star(f)) = \phi(f). \tag{6.2}$$

This says the pushforward is analogous to a transpose of a pullback.

3. Given any primal-dual basis pair $(\{f_a\}, \{\phi_b\})$ for P (i.e $\phi_b(f_a) = \delta_{ab}$), we have that $(\{T^\star f_a\}, \{T_\star \phi_b)\})$ is a primal-dual basis pair for P'.

4. Similarly any primal-dual basis pair $(\{f'_a\}, \{\phi'_b\})$ for P' (i.e $\phi'_b(f'_a) = \delta_{ab}$), we have that $(\{(T^{-1})^\star f'_a\}, \{(T^{-1})_\star \phi'_b)\})$ is a primal-dual basis pair for P.

Our core result, theorem 6.2.1.1, is our ability to solve definition 3.2.4. This result now closely mirrors our algorithm and provides the main condition on the spaces.

Theorem 6.2.1.1. *Let $E := (K, V, P, \Sigma)$ and $E' = (K', V', P', \Sigma')$ be finite elements where V and V' are compatible (p, q) tensor spaces, i.e., that they are subspaces of $\mathcal{T}(K)^{(p,q)}$ and $\mathcal{T}(K')^{(p,q)}$ respectively. Let $T \in C^1(K, K')$ have an inverse in $C^1(K', K)$. Let $\Sigma = \{\phi_b\}$ and $\Sigma' = \{\phi'_b\}$. Let $\{f_b\}$ and $\{f'_b\}$ be the associated primal bases. Further, suppose*

1. $T^\star_{|P}$ is an isomorphism onto P'.

2. There exist points $x_{b,q} \in K$ and rational functions $w_{b,q}$ such that for any $f \in P$,

$$\phi'_b(T^\star f) = \sum_q w_{b,q}(DT(x_{b,q}))(T^\star f)(x_{bq}).$$

This boils down to the existence of a quadrature rule for integrating functions $f \in P$ on K.

Then there exists a matrix P of shape $|\Sigma| \times |\Sigma'|$ such that

1. $\mathsf{P}_{b,b'}$ *is a rational function in the values* $\{DT(x_{b,q})\}$,

2. *the matrix* P *is a representation of the pullback* $(T_i)^\star$ *in a particular basis so that*

$$(f'_{b'})(T(x)) = \sum_b \mathsf{P}_{bb'}(x)(T^{-1})^\star(f_b)((x)). \tag{6.3}$$

Proof. Solving the problem symbolically requires a basis for P', for which there is only one choice[3], the primal basis for E under the image of $(T_i^{-1})^\star$. We call this basis $f_b^T := ((T^{-1})^\star(f_b)))$. We also have the dual basis $\phi_b^T := T_\star(\phi_b)$. We can compute ϕ_b^T in terms of $\phi'_{b'}$ by setting

$$\mathsf{C}_{b'b} := (T^{-1})_\star(\phi'_b)(f_{b'}) = \phi'_b((T^{-1})^\star f_{b'}) = \phi'_b(f_{b'}^T) \tag{6.4}$$

as in a generalized Vandermonde matrix. Finally, via duality, a representation of a dual basis in terms of another dual basis yields a representation of a primal basis in terms of another primal basis via an inverse transpose:

$$\mathsf{P} = \mathsf{C}^{-T}. \tag{6.5}$$

Thus, we simply need to invert and transpose C to yield a representation of f'_b in terms of f_b^T. Then, since $T^\star$ is the identity matrix, as a mapping from f_b to f_b^T, this yields a representation of f'_b in terms of f_b as desired. This also shows that P is a representation of $(T_i^{-1})^\star$ from the basis f_b to the basis f'_b.

By the definition of pullback, we observe that for $y = T(x)$, then $f_{b'}^T(y)$ is a polynomial in the coefficients $DT(x)$ and $DT^{-1}(y)$. By the conditions of the theorem, ϕ'_b will evaluate $f_{b'}^T$ at points $\{w_{bq}\}$ and will weight each value by a rational function in $DT(x_{b,q})$. Thus, each entry of C is a rational function $\{DT(x_{b,q})\}$. Since the inverse of a matrix of rational functions is a matrix of rational functions, this completes the proof. $\square$

[3]If we have the primal basis from the element E' already, we would not be doing this.

The core extra step is that $T^{\star}_{|P}$ is an isomorphism onto P'. Thus, our representation of function spaces must provide this or something that serves an equivalent function. We discuss this the next section after two additional matters.

6.2.4.1 Relation to prior work

We note that a key difference between our methods here and those employed by hand in [34] is that our methods operate on $P(K', V')^{\star}$ rather than $C^k(K', V')^{\star}$; in [34], certain definitions[4] to facilitate manipulations of DOFs by hand arise out of concerns about DOFs' behaviors as objects in $C^k(K', V')^{\star}$ whereas we avoid this at the cost of having to use f_b explicitly to manipulate objects in $P(K', V')^{\star}$. In particular, we note that the Morley DOFs on a physical element cannot be expressed as linear combinations of push-forwarded DOFs if we consider them as objects in $C^k(K', V')^{\star}$, but this is the case for the Morley DOFs as objects in $P(K', V')^{\star}$

6.2.4.2 Structure of Vandermonde Matrices

Another proof this results can validate our earlier claim that matrices should expose a lower triangular block structure, making them amenable to recursive inversion via Schur complements (section 6.2.3). We present this alternative as theorem A.0.2.1 in appendix A. This alternative computes P directly without an inverse at the end. The core idea is that the support of a basis function and its derivatives tends not to be changed too much by pullback. Thus, based on the structure of the original proof, we expect world space DOFs and reference space DOFs to only relate if their basis functions and relevant derivatives have similar supports. Thus, if we partially order DOFs by the topological supports and dimensions associated to their integrals, we reasonably expect that the transformed DOFs are only non-zero on the basis functions associated to DOFs that come earlier in the order. Our alternative inversion strategy formalizes this with more machinery, neatly separating out aspects of the

[4]Mainly compatible nodal completion, which is a stronger and harder to construct facility than our basis cover.

structure of the matrix determined by the sparsity patterns in basis functions (the triangular pattern of blocks) and the parts determined by the pointwise structure of DOFs (structure within the blocks, such as diagonal vs. non-diagonal blocks or numerical entry or a symbolic entry).

6.3 Specification of Polynomial Spaces

Our specification of polynomial spaces must satisfy the following requirements:

1. provide a tabulate function as in the interface depicted in section 6.1.1.

2. ensure that two polynomial spaces P and P' instantiated at two different isomorphic domains K and K' are isomorphic under pullback[5], allowing theorem 6.2.1.1.

3. be expressive enough to capture a wide variety of polynomial spaces, such as the affine invariant exterior calculus spaces (proposition 3.1.13).

The most challenging issue is balancing the first and last against the middle. To see why, we first recall the analysis from section 3.1.3: though all finite-dimensional spaces of a given dimension are isomorphic, rarely are function spaces constructed through similar mechanisms isomorphic via the pullback. If we want to express many types of functions, we exploit a trick that risks making our formulas much more complex: any finite-dimensional space of polynomial tensor fields is a subspace that is naturally isomorphic under pullback, typically polynomials of a certain degree and a tensor sum of tensor products of tangent spaces. With this idea, the same proofs and algorithms go through with an extra symbolic formula included to extra the subspace that we actually want from the one that is preserved. Of course, this formula might get in the way of simplification. Thus, our compromise is a grammar for tensor-valued polynomials parameterized via a chart where we can easily find

[5]or something that lets us write world space functions in terms of pullbacks of reference spae functions

an appropriate super space and where some options provide tricks to reduce the complexity of extracting the appropriate space.

```
⟨degree⟩ = Pure(⟨int⟩)
  |  Range(⟨int⟩, ⟨int⟩)
  |  Tensor(+⟨int⟩) ⟨PSpace⟩ = Pure(⟨topType⟩, ⟨degree⟩, ⟨tensorSpace⟩)
  |  Sum(+⟨PSpace⟩)
  |  Kernel(+⟨IPTEM⟩, ⟨PSpace⟩)
  |  InvariantKernel(+⟨IPTEM⟩, ⟨PSpace⟩, ⟨PSpace⟩)
  |  DofsKernel(+⟨IPTEM⟩, +⟨IPTEM⟩, ⟨PSpace⟩)
```

Source Code Listing 6.5: A grammar to tensor-valued polynomial spaces, parameterized by an unknown geometry.

Our overall grammar is in listing 6.5. Polynomial degrees come in three kinds, up to and including a given degree, a range of degrees, and a product of degrees. For pure tensor-valued polynomial spaces, the first two degrees only work with a topological type that is not a product and the product degree only works for the product. A sum of polynomial spaces is simply a direct sum again. The flexibility comes for the options of expressing a kernel, which can be expressed as:

1. a kernel of a collection of DOFs, requiring the kernel to be computed via an SVD or QR in the code,

2. a kernel of a collection of DOFs that we promise is invariant, as a hack,

3. an effective intermediate (due to [34]), where we provide a full dual basis for the target space and DOFs that identify the null space that we want to use, allowing us to extract the subspace via our preexisting infrastructure.

Lastly, we note that the range degree construct is actually not affine invariant and defaults to the middle case, as we mainly intend this as a hack. From this grammar, it is straightforward to compute the super space recursively and to implement tabulate recursively. We handle the base case via orthogonal polynomials, as in [8].

6.4 Permutations

A final task that our software representation of FEs must handle is computing how elements respond to permutations of vertices on the reference cell. We follow an approach similar to [126], but with a greater degree of automation. For a given way of permuting a mesh element, drawn by selecting parameterizations from our description in section 5.2.5, we simply rerun our algorithm for computing our symbolic Vandermonde inverse matrices, but with the world space chart now a numerical chart with a permuted domain.

Chapter 7

Global FEM Data and Computation

See example 3.4.1:

```
outScalar = 0
for cell in cells:
    # Setup cell geometry
    geometry = setupGeometry(cell)
    # Setup quadrature points
    points, weights = quadrature(cell, geometry)
    # Load basis function values
    # Based on element
    bfvals = computeVals(geometry, points)
    # Load function data:
    # Based on data of function represented in element:
    fdata = computeVals(cell)
    fvals = bfvals @ fdata
    # Compute F
    functionVals = map(F, fvals)
    # Integrate:
    outScalar += weights @ functionVals
```

See example 3.4.2:

```
outScalarPerEdgeCell = np.zeros(nedges, 2)
outIndicies = np.zeros(nedges, 2)
for edge in edges:
    edgeGometry = setupGeometry(edge)
    # Load cells on this edge
    cells = incidentCells(edge)
    assert 1 <= len(cells) <= 2
    # Setup quadrature:
    points, weights = quadrature(edge, edgeGometry)
    for (cellIdx, cell) in cells:
        edgePoints = edgeToCell(points)
        geometry = setupGeometry(cell)
        # Load basis function values
        # Based on element
        bfvals = computeVals(geometry, points)
        # Load function data:
        # Based on data of function represented in element:
        fdata = computeVals(cell)
        fvals = bfvals @ fdata
        # Compute F
        functionVals = map(F, fvals)
        # Integrate:
          outScalarPerEdgeCell[edge, cellIdx] += weights @ functionVals
          outIndicies[edge, cellIdx] = cell

```

See example 3.4.3:

```
externalFunction = lambda x: ...
outputFunction = np.array(nverts + nedges + nfaces):
for vertex in vertices:
    geometry = setupGeometry(vertex)

    outputFunction[vertex] = f(geometry.point)

for edge in edges:
    geometry = setupGeometry(edge)

    outputFunction[nverts + edge] = f(geometry.midpoint)

for cell in cells:
    geometry = setupGeometry(cell)

    outputFunction[nverts + nedge + cell] = f(geometry.midpoint)
```

Source Code Listing 7.1: Three different global kernel structures (sketches of the codes) that many FEM systems implement as three (or more) different kernels. See section 3.4 for corresponding reasons for these structures to exist

Many FE systems offer a wide variety of global data constructions and kernels or code generation paths. This is due to a wide variety of important capabilities needed in FEM systems, as described in section 3.3 and section 3.4. In particular, global FE spaces have a great deal of variety, and, when combined with the variety of operators, lead to many different capabilities, many of which require different global loop structures. For example, for each of the three examples in section 3.4, we have a different loop structure described roughly in listing 7.1. We reproduce these capabilities with one common code generation path corresponding to a single versatile global object and a minimal collection of other global objects that interact with it. Moreover, we do so in a way that improves the malleability or adaptability of our system: since we have one common code generation path targeting one versatile object, changing something and rerunning it often produces the desired result without further downstream code changes.

Our global model of data is built around a mathematical solution to the problems around multivalued quantities raised in section 3.4. By solving this problem first, we yield an idea that addresses our challenges parsimoniously:

1. the quantity is always defined so changes to code still yield something mathematically reasonable,
2. a relatively small collection of global objects is needed to construct the object,
3. and most FEM quantities can be computed via reasonable post-processing of the object.

This confluence is perhaps expected: the many different types of FEM kernels often deal with different ways quantities can be multiply defined (as detailed in section 3.3 and section 3.4).

To go along this path, we first deduce our notion of a global output mathematically. Then we develop a discrete infrastructure to build the various inputs/related global quantities and a code generation infrastructure to compute the output from the inputs.

7.1 Design Rationale

We now mathematically develop a single global output for our code generation process. To do this, we solve the problem of multiply defined quantities in FEM. From this, a single code pipeline that is malleable and parsimonious will follow.

First, we consider a class of functions that closely resemble IPTEMs.

Definition 7.1.1 (Integrated Pointwise Operator)**.** Suppose we have a domain S and $N+M$ finite-dimensional function spaces P_k, each of which is defined on some set that includes S. Then an $N+M$ pointwise operator is a N linear function with M non-linear arguments given by

$$\begin{aligned}a(u_1\colon P_1,\dots,u_N\colon P_N; u_{N+1}\colon P_{N+1},\dots,u_{N+M}\colon P_{N+M}) = \\ \int_S F(x,u_1,\dots,u_N;u_{N+1},\dots,u_{N+M})dS(x)\end{aligned}$$

where

1. $dS(x)$ is an appropriate measure for the set S such as a delta measure for a point or Lebesgue measure for a surface or domain, and

2. F is a function such that F depends pointwise on all its arguments and derivatives;

more formally, this is to say that F takes the form

$$\begin{aligned}F(x, u_1, \dots, u_N; u_{N+1}, \dots, u_{N+M})\\ = G(x, u_1(x), Du_1(x), D^2u_1(x), \dots,\\ u_2(x), Du_2(x), D^2u_1(x), \dots,\\ \dots,\\ u_{N+M}(x), Du_{N+M}(x), D^2u_{N+M}(x) \dots).\end{aligned}$$

for some function G.

These can be extended to multiples, mirroring that IPTEMs can be grouped when their arguments are similar and that they can be topologically enumerated.

Definition 7.1.2 (Integrated Pointwise Operators)**.** Suppose we have a collection of labels L and for each label $\ell \in S$, we have a collection of sets $\{S_{\ell i}\}$. Suppose for each $\ell \in L$, we have a pointwise function F_ℓ. Suppose we have $N + M$ finite dimensional function spaces P_k, each of which is defined on some set that includes all $S_{\ell i}$. A collection of pointwise operators is a list of pointwise operators $a_{\ell i}$ of the form

$$a_{\ell i}(u_1 \colon P_1, \dots, u_N \colon P_N; u_{N+1} \colon P_{N+1}, \dots, u_{N+M} \colon P_{N+M}) = \\ \int_{S_{\ell i}} F_\ell(x, u_1, \dots, u_N; u_{N+1}, \dots, u_{N+M}) dS_{\ell i}(x).$$

The primary FEM task with such computations is to assemble them into sparse matrices/tensors, which we can formalize as follows:

Definition 7.1.3 (Assembled Pointwise Operator)**.** Given an $N + M$ multilinear pointwise operator a defined on spaces P_k with bases $\{p_{m_k}\} \subset P_k$ for $1 \le k \le N$, and a list of M functions $u_k \in P_k$ for $N \le k \le N + M$, then an assembled pointwise operator is an array of numbers of shape $\dim P_1 \times \cdots \times \dim P_N$ given by:

$$F_{m_1,\ldots,m_N} = a(p_{m_1}, \ldots, p_{m_N}; u_{N+1}, \ldots, u_{N+M}). \tag{7.1}$$

However, as we saw in section 3.4, this operation might not be reasonably defined due to the structure of the FEM space, or we might want to compute various quantities such as interpolation or a boundary that do not quite fit this model. We solve by overcoming the discontinuity of FEM spaces. As described in section 3.3, given a multilinear form a with a domain set $S = \cup K_i$ where the K_i correspond to FEs used to define the input spaces, then we can decompose the computation of the assembled form onto each element K_i. However, if S does not decompose along the elements used to define the function spaces, then we need some more sophisticated machinery because a function or its derivatives might be discontinuous along some part of S. In particular, we label the combinations of single values that need to be computed and store them in a large assembled object, which can be reduced to recover an assembled form or other things. We call this construction the Single Valued Assembled Operators.

Definition 7.1.4 (Single Valued Assembled Operators)**.** Let $\Omega \subset \mathbb{R}^n$. Suppose we are given $N+M$ spaces P_k wherein functions are C^∞ on sets K_{jk}. Let $L_k = |\{E_{kj}\}|$ and $K_k := \cup_j K_{kj}$. Suppose we are given corresponding $N+M$ multilinear pointwise operators $a_{\ell i}$ over domain sets $S_{\ell i}$ with $\cup_{\ell i} S_{\ell i} \subset K_j$ for all j. Let L be the number of operators, the range of ℓ, and let I be the maximum range of i for all ℓ. Define $SK_{k_1,\ldots,k_{M+N},\ell,i} = \cap_{j=1,\ldots,N+M} K_{k_j j} \cap S_{\ell i}$, the intersections of the domains used for each FE space and the domains of some pointwise operator. We note that either $S_{\ell i} \subset SK_{k_1,\ldots,k_{M+N},\ell,i}$ or the set is empty. Then the single valued assembled operator is an array F of size $\dim P_1 \times \cdots \times \dim P_N \times L_1 \times \cdots \times L_{N+M} \times L \times I$

given by:

$$F_{m_1,\ldots,m_N,k_1,\ldots,k_{N+M}} = a_{\ell i}(p_{m_1}|_{K_{k_1,1}}, \ldots, p_{m_N}|_{K_{k_N,1}};$$
$$u_{N+1}|_{K_{k_{N+1},N+1}}, \ldots, u_{N+M}|_{K_{k_{N+M},N+M}}).$$

Each entry of the array can be computed via integral over the domain $SK_{k_1,\ldots,k_{M+N},\ell,i}$.

Critically, definition 7.1.4 provides an object to compute using the ideas of section 3.3, i.e., it can be computed per combination of cell interiors, which is where DOFs and derivatives are continuous and singly defined. And even better: we can recover the various other FEM quantities from this. Given definition 7.1.4, we can recover an assembled operator as defined in definition 7.1.3 by reducing along equivalence classes of basis functions when this makes sense. This could also produce scalar or vector quantities with similar meanings. We can recover other objects via other procedures: we can use slicing to isolate parts with certain meanings, or we can use other reductions to build other quantities (such as the original input fields), or we can filter one input based on the entire tensor to identify certain subsets of the inputs. These ideas can aid with DG methods, interpolation, boundary elimination, or boundary identification respectively. More generally, we need the following common sparse array primitives: general reduction across axes, slicing out a matrix, setting a slice to a value or array of values, and a where construct (as in `np.where`) for filtering other constructs with the sparse array.

Our core interface and loop for doing global FEM computation is similar to the above structure, using it to support a wide variety of FEM operations. IPTEMs and global fields are turned into SVAO over some mesh. The SVAO can be turned into other objects that capture operators, sub meshes, spaces, sub-spaces, other global fields, and more These are used to construct or filter more SVAO. This loop supports a single path for many FEM operations: assemble a SAVO and then post-process it with sparse array operations. In what follows, we develop the machinery necessary to specify a discrete SAVO and the various other global objects that we can turn it into. We develop this machinery in sufficient abstraction that it is easily adapted to a variety of changes, making our codes malleable.

7.2 Global Meshes

7.2.1 Definition

We have already mathematically modeled local mesh geometry, entities, as simplices or products of simplices (via definition 5.2.1). In particular, we modeled these as posets over some set of vertices. Software-wise, this model is very close to an implementation, module some ordering of the posets for array based storage. Mathematically, our global meshes are also posets with a regularity condition that makes them meshes of entities.

Definition 7.2.1 (Mesh Topology)**.** Let V be an abstract finite set called the vertices of a mesh topology. Let $F \subset P(V)$ be a poset under $\subset$, called the set of faces. Then (V, F) is a mesh topology if

1. For every $v \in V$, $\{v\} \in F$,
2. and if for every $f \in F$, $F_f := \{f' \in F \colon w \subseteq f\}$, a sub-poset of $P(V)$, is isomorphic to either a simplex or a product of simplices as a poset.

For the software model aspect, we observe that storing this is achieved by storing a vertex list, a list of facets of each type, and a list of isomorphisms to a chosen standard entity (definition 5.2.3) for every face.

7.2.2 Orientations

Given a mesh topology (V, F), an orientation of an element $f \in F$ is an ordering of the vertices of f. There are several orientations that we store. The orientation on each top level entity given by the input to our process is often geometrically relevant for computing outwards normals or similar information. The orientation induced by a global ordering of vertices allows DOFs to be correctly ordered between cells. (To see that these two are different, see [70]). Finally, each standard entity (definition 5.2.3) has a natural ordering of the vertices so for every pair of faces $f_1, f_2 \in F$ where $f_1 \subset f_2$, then f_2 induces an orientation on f_1 via the ordering of the vertices of f_2. In particular, we are interested in the induced orientation in the case of the ordering of f_1 according to the isomorphism to a standard entity that stores f_2. We store the first orientation and the second is trivial to store. For the latter orientations, we can compute how induced ones compare to the global orientation on-demand to determine how degrees of freedom need to be permuted when transforming them between world and reference spaces. These orientations would be supplied to the process specified in section 6.4 through the global field interface specified in section 7.4.3.

7.2.3 Implementation Aspects

7.2.3.1 Construction

Our mesh topologies seem to require much more than the typical face to vertex map that determines a Tri or a Tet mesh, but a process, known in the literature as topological interpolation [127], can automatically construct a global discrete topology from analogous data. In particular, given a mapping from the top level cells of a mesh to vertices and given a choice of isomorphism to the standard entities of each top level cell, we can construct new identifiers for intermediate topological objects corresponding to unique sets of vertices. This is analogous to the quotient/group-by procedure used in the construction of a global FE space as in definition 3.3.4. Once we have all the new topological objects and their vertices, we can construct all the isomorphisms as they are determined by comparing vertices in an entity map to the vertices in the standard entity.

7.2.3.2 Different Mesh Formats

Meshes can be stored with a variety of local orderings for vertices in facets, edges in faces, vertices in edges, etc. Our system allows users to specify these orderings via arrays that list automorphisms standard entities. Our system then uses these orderings to determine the orientations of reference entities so that any properties of these orderings are available. The most common example of such a property is the orientation of normal vectors.

7.2.3.3 Entity Storage and Relationships

We store entities as a uniform list of numbers (e.g., if we store vertices and edges, vertices will be identifiers 0 through $|V|$ and edges $|V|$ through $|V| + |E|$). This choice simplifies isomorphism storage as multiple types of entities in an array cannot be confused. This choice also simplifies storing entity relationships. Since entities are subsets of vertices, the most important entity relationships are the $\subset$ and $\supset$ relationship between vertices. Our mesh stores two matrices for these relations, computed from the isomorphism arrays, that capture these relationships for all types of entities all at once. Two key uses of these relations are the computation of boundary entities (especially in meshes involving product cells with multiple types of boundary entities) and the computation of maps from interior boundary edges to cells.

7.2.3.4 External Mesh Data

We can load external data on any class of mesh entities via the association between vertices and entities. If we preserve vertex identifiers in construction[1], then in a simple search procedure can convert incoming lists of entities as vertices to entity identifiers. This facilitates the interface in section 2.7.

[1]or at least store the re-orderings created by processes such as RCM

7.3 Sections: Usage and Storage of Global Fields

Our IPTEM compilation process presumed field inputs (section 5.3.3) which, given a parameterization of an entity, produce values and derivatives at points. And these field inputs could be given as a collection, representing a basis of fields that would be supplied to an IPTEM. However, we cannot model global FE spaces in this manner. The idea of a field or basis of them fails to model a key discrete aspect of global FE spaces: how basis functions and coefficients on basis functions relate to the mesh topology, which is critical to how we define the SVAO. We introduce a new concept, sections[2], that captures the discrete essence of a global FE spaces and enables us to model global fields constructed in global FE spaces as well discrete assembled FEM operators and basically all other data that we store on meshes

7.3.1 Sections: Data on Meshes for Global Finite Element Spaces

Just as meshes model a structure similar to an entity and that locally literally is an entity, sections model data placed on meshes that locally conforms to a specific structure. Our model of local structure, definition 7.3.1, neatly matches with outputs of the IPTEM compilation process, section 5.5.1, and enables a global structure for global spaces (or other mesh data).

Definition 7.3.1 (Schema for data on a mesh)**.** Let L be an abstract set of labels (ideally IPTEMs). Let $\mathcal{T}$ be the collection of all topology types, defined by the grammar fig. 5.2. A function $f\colon L \to \mathcal{T}^2$ is a schema for a mesh if for all $l \in L$ then $(t_1, t_2) = f(l)$ satisfies $t_1 \leq t_2$.

A schema indicates that data with a given label should be placed on all pairs of entities of two given types where one entity is completely included in the other. Given a list of IPTEMs, we can automatically construct a schema via mapping from the IPTEMs to their integration topological type and their storage topological type.

[2]named for a similar construct in PETSc's DMPLEX library

Schema give rise to sections when paired with a mesh topology. Sections have the property that sections made with the same schema are isomorphic iff the mesh topologies are isomorphic. Thus, a section on an entity over a schema is always isomorphic to a section on the standard entity.

Definition 7.3.2 (Section, Isomorphism of Sections)**.** Let $\mathcal{T}$ be the set of topology types and let L be an abstract set of labels. Further, let $f\colon L \to \mathcal{T}^2$ be a schema. If (V, F) is a mesh topology, let $\mathrm{Ty}\colon F \to \mathcal{T}$ be a mapping from faces to the topology types corresponding to the standard entities a given face is isomorphic to. A section $S_{V,F,f}$ is the set $S_{V,F,f} := \{(l, h, g)\colon l \in L, h \in F, g \in F, h \subset g, (t_1, t_2) = (f(l)), \mathrm{Ty}(h) = t_1, \mathrm{Ty}(g) = t_2\}$. An isomorphism between two sections with the same underlying schema and vertices, $S_{V,F,f}$ and $S_{V,G,f}$, is a mapping $i((l, f, g)) = (l, i'(f), i'(g))$ where i' is an isomorphism between posets F and G.

7.3.1.1 Representing Global Finite Element Spaces

Sections can represent global FE spaces. In particular, the elements of a section correspond to global basis function after DOFs have been quotiented based on their identification as a unique DOF type (a label) and a correspondence with a pair of nested facets. We can even directly construct an equivalent to the broken FE space and the equivalence relationship construction of the global space (See section 3.3). We do this explicitly in the case of a mesh of a single largest cell type, for simplicity. Given a schema $f\colon L \to \mathcal{T}^2$ and the largest topological type T, we can construct $f'(l) = (t_1, T)$ where $(t_1, t_2) = f(l)$. This generates a section $S_{V,F,f'}$ where for every $q \in F$ where $\mathrm{Ty}(q) = T$, the subsets $L_q = S_{V,\{h \in F\colon h \subset q\},f'\}}$ are a partition of $S_{V,F,f'}$. This is the broken FE space where pairs of labels and faces identify local DOFs. We can then define an equivalence relationship on $S_{V,F,f'}$ where $(l_1, t_1, t_2) \sim (l'_1, t'_1, t'_2)$ if and only if $l_1 = l'_1$, $t_1 = t'_1$, and there is some $t \in F$ such that $(\mathrm{Ty}(t_1), \mathrm{Ty}(t)) = f(l)$ and such that $t \subset t_2$ and $t \subset t'_2$. Then $S_{V,F,f'}/\sim \cong S_{V,F,f}$, meaning we have recreated the global FE space via a similar quotienting process, provided the equivalence relationship relies on glue DOFs based on their relationship to specific topological entities (such as the support of a DOF (definition B.0.2)).

7.3.1.2 Software Representation and Computation of Sections

Sections are stored in a manner similar to how meshes are stored, mainly via isomorphisms to standard entities. Suppose a mesh (V, F), that we represent via storing an isomorphism i_e for every $e \in F$ to some standard entity chosen per topological type. Given a mesh (V, F) and a section $S_{V,F,f}$ over some schema f, then every entity $e \in F$ corresponds to a subset $S_{e,\{f \in F \colon f \subset e\},f}$ and this will be isomorphic to $S_{i_e(e),\{i_e(f) \colon f \subset e,\},f}$. This essentially creates standard sections per standard entity, and we store the isomorphisms to the standard section. To realize this as arrays requires ordering everything via a dictionary ordering induced via some ordering on L (typically the order in which IPTEM were supplied). These arrays are the typical cell to node mappings of FE systems.

Sections additionally can store mappings that relate the section to and from the mesh. In particular, a section can store two mesh entity to section maps, defined rather obviously via converting the section definition to CSR tensors. And similarly, a section can store two section to entity maps, also defined rather obviously as arrays.

Sections are also computed similarly to meshes and along the lines the quotienting procedure for global FE spaces. Given a schema and a mesh, a standard section is constructed per each top level topological type. We create an isomorphic copy of this section for every top level entity. Then we quotient the like entities based on the definition of a section (exactly as in the construction used to model the global space). Then, we generate isomorphisms for all entities that are not of top level topological type. Finally, we create the section to entity maps by iterating over the isomorphism maps. Entity to section maps can be created on demand.

7.4 Global Fields

A global field is analogous to a local field input in an IPTEM. However, unlike the local field input, the global field is for more than just code generation. Our global fields must provide local fields to generate code via IPTEMs, and they must be a container for the global data on the mesh, such as sections and arrays representing coefficients. They must also represent a single field as well as a collection or basis of fields. We provide a core global field interface that serves as a clear semantics of a global field and then separately indicate the data and code generation aspects of the interface. These must be separable to facilitate generating code independently of the underlying data.

7.4.1 Global Field: Core

In the context of global FE spaces, the semantics of a global field are very intuitive: they are the semantics of a tensor field (or collection of fields) that you can sample at a point after restricting it to an appropriate part of the mesh. We capture this idea with an interface (section 7.4.1) that offers a field type, a mesh topology, the number of fields involved locally, and the ability to sample on appropriate cells. The main contract is that the topological type of the field is present in the mesh and that we can only sample entities of that topological type.

```python
class GlobalFieldCore:
    fld: fieldType
    def tabulate(entity: MeshEntity, points: NDArray, derivatives: int, basis: Optional[
        Basis] = None) -> Tuple[NDArray, basis]: ...
    localDim: Optional[int]
```

Source Code Listing 7.2: Semantic interface for global fields

7.4.2 Global Field: Data

Global fields also hold global data, which can be semi-systematically categorized. We can assume that input arrays to global fields should be paired with a section though we might have a section without an array. However, for other inputs, we might have closures or really arbitrary other arrays (for extensibility).

```python
class GlobalFieldData:
    section: Optional[Section]
    data: Optional[NDArray]
    callables: Optional[List[Callable]]
    other: Any
    mesh: Mesh

    def isCollection(self):
        return self.section is not None and self.data is None

    def isReprersentation(self):
        return self.section is None and self.data is notNone
```

Source Code Listing 7.3: Runtime data that a global field must supply

7.4.3 Global Field: Code Generation

To generate global code on a global field, we will need to plug into our code from IPTEMs, but we also need to supply information and get information for a global field. To plug into IPTEMs, we just need to provide an appropriate symbolic field, but to plug that generated code into the global context, we need to wire in information such as the current mesh entity and so forth. Additionally, in the case of data with a section, we need to know the section for the standard entity of the field (to keep code generation independent of data). This interchange of information between a global field and code generation is hard to understand out of context so here we provide a more abstracted interface.

```python
class GlobalFieldCode:
    def symbolicField(self) -> Field | List[Field]: ..
    def queryCode(self, inputEnv: FieldCodeQuery) -> FieldCodeResults: ...
    def localSection(self, standardEntity) -> Optional[Section]: ...
```

Source Code Listing 7.4: Data provided by a global field at code generation time.

In particular, our code query interface supplies a `FieldCodeQuery`, which provides many values that are used in code generation such as the number of derivatives we require and the symbolic charts in use, but more importantly abstractly provides code for (at least):

- the world and reference positions that a field is being sampled at,
- the mesh entities (indices) that the samples live in,
- code for the parameterization of these entities (associated to symbolic charts),
- the orientations of the entities in terms of the relationship between the induced orientation and global orientation,
- code to access any input global section via the given mesh indices,
- and the code to access the relevant parts of the field data with the global section indices.

To be clear, by code we mean literal code as a data structure that cannot be executed but is part of a code generation process; the indices of mesh entities are not literal, but some snippet of code which when executed as part of a large program will be that value. The code query interface then returns code for:

1. values of the fields at the given positions,
2. the global section indices (if they exist) corresponding to the field.

In addition to this code query interface, the interface also supplies a section, a standard section (the same schema on the standard entity associated to the topological type of the field), that can be used to construct a global section on a yet unknown mesh. This allows the code generation process to figure out how to access the global section data.

7.4.4 Global Field: Examples

We roughly sketch a few possible global fields that our system offers. These use the internal infrastructure of IPTEM as well as the machinery described above and code generation infrastructure described later (section 7.8). We do not describe them in great detail, just to sketch how they can conform to our above interface. These interfaces are highly extensible, but would be hard to extend without extensive internal knowledge.

7.4.4.1 DOFs

Given a software representation of a FE, we can build a global field representing the basis. The core comes directly from the definition and the data is the section you get from compiling the DOFs (section 5.5.1). The code generation interface uses the geometry (charts, mesh entities) to generate code that constructs the Vandermonde inverse matrix. The code generation interface simply makes use of the orientations to emit code to construct a matrix to adjust the DOFs for re-orderings of the vertices of mesh entities. Finally, the code generation interface combines these with tabulations of values in a reference space to supply the desired values. The code generation interface also returns access to the global section based on the section, which will be supplied in the context of the code.

7.4.4.2 DofFunction

Naturally, in addition to the FE basis functions, we want arbitrary linear combinations of these. A DofFunction is like DOFs with an array to specify the linear combination and with a slight change to the code generation process: we must also emit code to access the coefficients and then contract these against the sampled basis functions.

7.4.4.3 Pointwise Function

Given an arbitrary Python function and a field type, we can construct a pointwise function (provided that the field only deals in Euclidean spaces). The core is the field type and the ability to sample the pointwise function. The code generation simply emits code to call the function at world points. Obviously, no section is outputted.

7.4.4.4 Pointwise Fields from Other Fields

Given other global fields that represent a single field (as opposed to a basis), we can utilize pointwise tensor expressions on manifolds to make another global field. We do this to support taking a gradient of a global field outside of an integral, facilitating code reuse, The code generation will compile the pointwise tensor expression to determine how to query the input fields, use the query in the pointwise tensor expression, and then supply the output pointwise values. We again exploit the idea of pointwise here so that when we query the field at some pointwise values the values are simply propagated back to the inputs, along with any required geometry/index information.

7.4.4.5 Vector Functions

Piecewise constant fields are sufficiently important that we highlight them. These are useful for defining lots of mesh quantities such as cell permitter or cell size indicators or cell normals. We need to realize some of these on the mesh to compute them as some quantities are averages across cells, most notably cell size indicators, which can be useful for DG methods, Nitsche's method, and measuring convergence. Given an IPTEM with no non-default field arguments and a Euclidean tensor output, we essentially have a mechanism to specify a vector on the associated mesh entity, and these vectors can be reduced (listing 5.7). The core is simply the field type of the output of the IPTEM. The data has an output section describing where vectors are on the mesh and an array describing the vectors. The code generation simply emits code to access the appropriate vector depending on the mesh entities.

7.5 Sub-Meshes and Sub-Sections

Meshes and sections have subsets, representing sub meshes (including boundaries) and subspaces. These are simple to define and our main focus is on constructing subsets, as this is where our generality shines.

Definition 7.5.1 (Sub-Mesh)**.** Given a mesh (V, F), a sub-mesh is simply some other mesh (V', F') with $V' \subset V$ and $F' \subset F$.

We note that this is differently from simply a subset of mesh:

Definition 7.5.2 (Mesh Subset)**.** Given a mesh (V, F), a mesh subset is simply a subset $F' \subset F$.

For sections, sub-sections are like mesh subsets:

Definition 7.5.3 (Sub-Section)**.** Suppose we have a mesh (V, F) and a schema $f : L \to \mathcal{T}^2$, generating a section $S_{V,F,f}$. A sub-section is simply a subset of $S_{V,F,f}$.

Sub-sections and sub-meshes can be constructed in a variety of ways, showcasing how various global aspects of the system come together. We depict the situation in fig. 7.1. Sub-meshes and sub-sections can be constructed from external data or the special case of the topological boundary constructed from the relationships between mesh entities. Critically, a sub-mesh (V', F') can be **propagated** to a sub-section:

Definition 7.5.4 (Propagation)**.** Given a sub-mesh (V', F') and a section $S_{V,F,f}$, the **propagation** of the sub-mesh to a sub-section is given by

$$\{(l, t1, t2) : (l, t1, t2) \in S_{V,F,f}, t1 \in F'\}.$$

This simply extracts the portion of the section where the entities are on the sub-mesh; this captures the usual boundary basis functions of a FE space. However, critically, we can also go the other way sometimes. If we have a schema $f\colon L \to \mathcal{T}^2$ where $\pi_1 \circ f$ is injective, then any section over f can be converted into a sub-mesh.

Definition 7.5.5 (Submersion)**.** Given a subset S of a section $S_{V,F,f}$ where $\pi_1 \circ f$ is injective, we can define a subset mesh via first defining a subset $F' \subset F$ via:

$$F' = \{f \in F \colon \exists!(l, t_1, t_2) \in S_{V,F,f} \text{ s.t.} t_1 = f\}.$$

and then we can define a sub-mesh (V', F'') as the smallest mesh such that $F' \subset F''$. We call this sub-mesh a **submersion**.

This little bit of topological ingenuity allows all sorts of data to be transported to sub-meshes, such as using Lagrange elements to identify a boundary. Finally, assembled outputs with a given section can be used with a predicate to filter a section or sub-section, via a where constructed to be discussed in section 7.6.2. Additionally, we note that subsections and sub-meshes can be trivially combined with boolean operations, including and/or/complement, as they are simply masks. We now proceed to discuss the assembled outputs, the runtime output of generated code, which can be used to create these sub-meshes via filtering or global fields using reductions. We will see that these can also be sliced and filtered via sub-meshes and subsections, facilitating the construction of many algorithms, such as elimination of boundary DOFs.

7.6 Global Assembly Interface and Semantics

With the necessary global discrete machinery in place, we can produce our global interface. We do this in two stages, the combination of which will yield many possible global computations. First, we discuss the genericAssembly function, which produces SVAO (definition 7.1.4) like objects. Then we discuss operations on these.

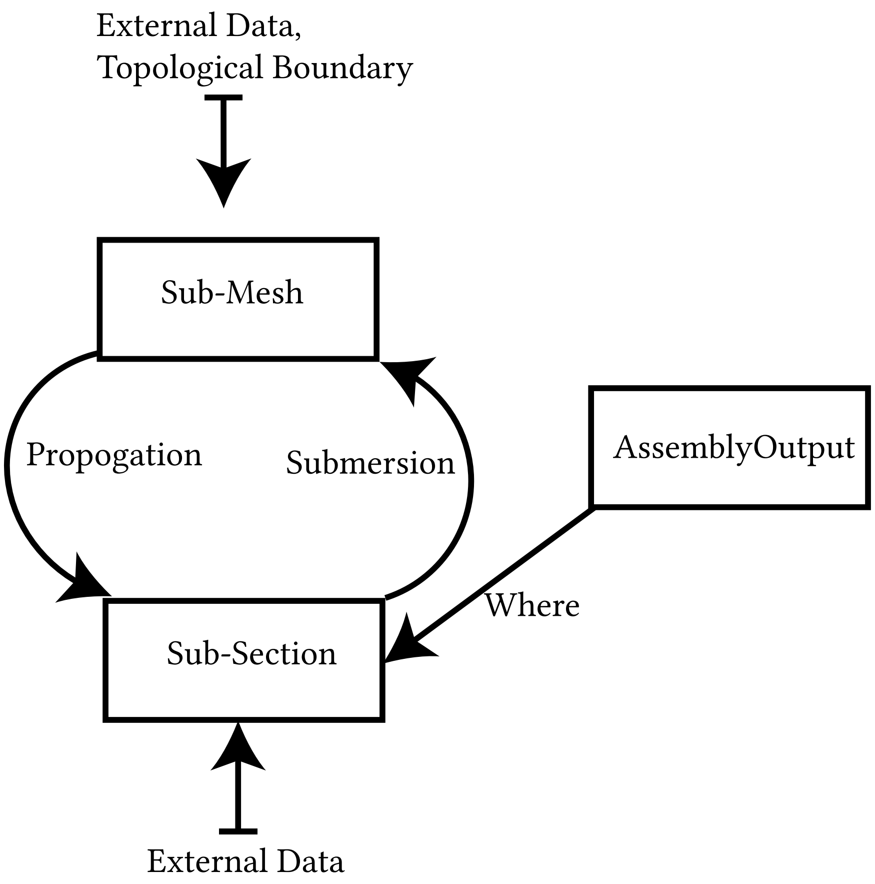


Figure 7.1: Mechanisms for constructing sub-meshes and sub-sections

7.6.1 Generic Assembly

The signature of genericAssembly is depicted in listing 7.5.

```python

def genericAssembly(
    mesh: mesh,
    expressions: List[IntegratedPointwiseExpression] | FiniteElement,
    linearInputs: List[GlobalField],
    nonLinearInputs: List[GlobalField] | dict[str, [GlobalField]],
    ) -> AssemblyOutput:
```

Source Code Listing 7.5: The Generic Assembly Interface

The interface mirrors the requirements for a SVAO: a mesh, IPTEMs (potentially from an element), a list of global fields that match the linear inputs of all IPTEMs, and a list of non-linear functions that match arguments via keywords in various IPTEM. The output, AssemblyOutput, is essentially a high dimensional COO tensor with some meta-data per dimension. An AssemblyOutput has dimensions that closely mirror those of a SVAO:

1. The first three dimensions, corresponding to a schema over the IPTEMs, the integration domains (mesh entities) of the IPTEMs, and the section generated by the IPTEMs over a mesh.

2. The next dimension is associated to the mesh entity in which the calculation was performed (the top chart in our IPTEM).

3. Then there are three dimensions per linear input that has a section, mirroring those of the first three dimensions (the mesh entity used to sample the field, the global section index, and the local index of the global section index according to the section isomorphism on the mesh entity).

4. Finally, for each linear input without a section or non-linear, we include a dimension for the mesh entity where the input was sampled.

The meaning of an entry of this tensor closely mirrors that of an entry of a SVAO. The only major deviation is the second dimension, which is an implementation aspect: we seek to compute the IPTEMs together so for example a group of edge and cell integrals for a cell field can be computed on a cell together, which we can record. However, this turns out to be more computable than the SVAO because our meshes are conforming and functions are C^∞ on certain faces; in particular, this means that the intersection of integration domains and the domains where inputs are smooth is always a given mesh entity. In particular, by construction of the IPTEM, the integration domain must be a subset of all incoming domains (see section 5.4). Thus, the generic assembly interface produces a SVAO with some additional metadata and where every known domain is exactly represented by a discrete mesh entity.

To be a bit more precise semantically, we can exactly write down the sparsity pattern with a bit of machinery and provide meaning to each entry. Suppose the mesh is (V, F). And suppose that the output schema is f so we have a section $S_{V,F,f}$. Further suppose we also have input schema f_i with S_{V,F,f_i} for $i = 1, \ldots, N$. Finally, since each input has a field type, we know it is sampled on a domain of a given type ty_i for $i = 1 \ldots, N + M$, and we call the corresponding subset $F_i = \{f \colon f \in F, \mathrm{Ty}(f) = ty_i\}$. For the linear inputs, we know that the section S_{V,F,f_i} grants us, for each $c_i \in F_i$, an isomorphism S_{c_i} from the section on the standard entity of c_i to the section S_{V,c_i,f_i}. Finally, let ty be the largest type of all inputs and let $F' = \{f \colon f \in F, \mathrm{Ty}(f) = ty\}$. Now we define:

Definition 7.6.1 (Subset-N relationship)**.** Given a mesh topology (V, F), we say that $(f_1, \ldots, f_N) \in \subset^N_{V,F}$ if and only if there is a $1 \leq j \leq N$ such that $f_j = \cap_i f_i$.

The sparsity pattern of the assembly routine is given by:

$$
\begin{aligned}
P = \{(&l, t_0, s_0, t, t_1, s_1, l_i \ldots, t_N, s_N, l_N, t_{N+1}, \ldots, t_{N+M}) : \\
&s_0 = (l, t_0, t_0') = t_0 \in S_{V,F,f}, s_i \in S_{V,F,f_i}, t_i \in F_i, t \in F', \\
&(t, t_0, \ldots, t_{N+M}) \in \subset^{1+N+M} (V, F) \\
&c_i(l_i) = s_i \\
&\}.
\end{aligned}
$$

And the meaning of each entry can be made precise: the semantics of global fields plus the semantics of IPTEM give us a recipe to calculate an entry. As a final note on semantics, we note that at a call to generic assembly, the system:

1. Generates code that is independent of the underlying mesh or data or sections of the input fields. The code depends only on their code aspects, section 7.4.3.
2. For the given mesh, computes the output section.
3. Calls the generated code, producing the sparsity pattern and values for each non-zero.

We note this to state that code generation is appropriately separated from the input data. We also note this to state that the construction of sections occurs via generic assembly and thus emerges naturally rather than through a separate construction process.

7.6.2 AssemblyOutput Operations

The output to generic assembly, AssemblyOutput, supports a variety of post-processing methods: reductions, where, slice, and slice-set. Additionally, meta-data from the IPTEMs allow sensible defaults on these. In particular, the choice of reductions can be indicated in the IPTEM and good defaults exist if they are not specified which produce the usually desired behavior for operators and DOFs. We now outline the interface to each operation in turn. For reductions, we support reduction over the non-specified axes and the reduction can be chosen up to indices from the specified axes. The defaults are inferred from the IPTEM's default reductions, which can be overridden in the global case (see listing 5.7). Via the metadata for reductions and IPTEMs included, we support (when possible) automatic conversion to a global field (e.g., for interpolation) or automatic conversion to a scalar or dense vector or sparse matrix. The interfaces are summarized in listing 7.6. Slicing functionality is similar to slicing in numpy except that dimensions are sliced with our global constructions: subsections, sub-meshes, list of IPTEM, or indices for local linear inputs. Slicing can simply extract a slice, but the non-zeros of a slice can also be set. As a convenience, we offer an option to set the diagonal over two axes, useful for boundary conditions. Slicing is detailed in listing 7.7. Finally, we offer a where-functionality that allows us to construct subsections, sub-meshes or just lists of indices. The interface is detailed in listing 7.8. As a final addendum, we note that users can extract the output section and dimension information, which is how we expect users to get such information. This is detailed in listing 7.9.

```
# Reduction optionally using indices and/or the number of non-zeros in the reduction region.
Reducer = Callable[[NDArray], NDArray] | Callable[[int, NDArray], NDArray] | Callable[[
    NDArray, NDArray], NDArray] | Callable[[int, NDArray, NDArray], NDArray]

def reduce(ao: AssemblyOutput, axes: Optional[List[str]] = None, reducer: Optional[Reducer]
    = None) -> AssemblyOutput: ...
def asScalar(ao: AssemblyOutput) -> Optional[float]: ...
def asDenseVector(ao: AssemblyOutput) -> Optional[NDArray]: ...
def asScipySparseMatrix(ao : AssemblyOutput, format: str = "coo") -> AssemblyOutput: ...
```

Source Code Listing 7.6: Reduction Interfaces

```

sliceIndex = List[IPTEM] | SubSection | SubMesh | NDArray | IPTEM | int

def slice(ao: AssemblyOutput, dimensions: List[str], subsets: List[sliceIndex]) ->
    AssemblyOutput

Setter = Zero | flat | Callable[[NDArray, NDArray, NDArray], NDArray] | AssemblyOutput
```

```
# The second to last option takes slice indices, current values of the slice, and the non-
    zeros of the slice.
# The last option must have been sliced on the appropriate dimensions with the same subsets.

def setSlice(ao: AssemblyOutput, dimensions: List[str], subsets: List[sliceIndex], setter:
    Setter = Zero) -> AssemblyOutput: ...

def setDiagonal(ao: AssemblyOutput, dimension1: str, dimension2: str, subset1: sliceIndex,
    subset2: sliceIndex, val: float | NDArray) -> AssemblyOutput: ...
```

Source Code Listing 7.7: Slice Interfaces

```

sliceIndex = List[IPTEM] | SubSection | SubMesh | NDArray | IPTEM | int

FilterTy = allZero | allNonZero | Callable[[int, NDArray, NDArray], bool]

def where(ao: AssemblyOutput, filter: Optional[FilterTy] = None) -> sliceIndex: ...
```

Source Code Listing 7.8: Where Interfaces

```

def shape(ao: AssemblyOutput, dim: str) -> int: ...

def outSection(ao: AssemblyOutput) -> Section: ...
```

Source Code Listing 7.9: Metadata Interfaces

7.7 Overall User Workflow

We depict an overall user workflow in fig. 7.2. As stated previously, the idea is to start with a few bespoke objects and then use IPTEMs and assembly to progressively build others that various post-processing operations convert to more objects to build or post-process further global objects. In this version, we also suggest how this would interact with a full pipeline, involving solvers and other external user code.

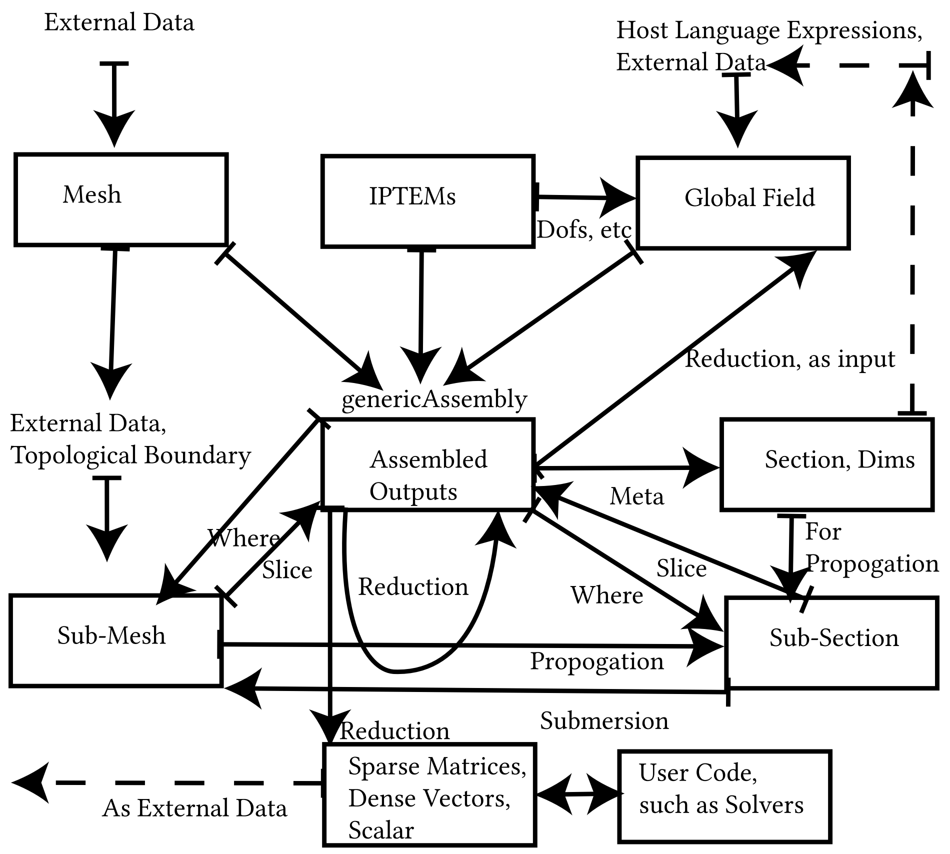


Figure 7.2: Overall Global Work Flow

7.8 Code Generation

We provide a parsimonious global abstraction for a wide variety of FEM computations through our modeling and decomposition of global FEM objects and through code generation to produce a SAVO like object from meshes and global fields. In particular, our code generation system leverages the IPTEM pipeline to generate the local part of the code while the stitching the rest together through a demand driven query system where the code generation models of the inputs are queried from the IPTEM local code to generate code to supply the IPTEMs with inputs and to stitch together the outputs into an appropriate sparsity pattern. By design, this code generation process finishes the process of addressing the main challenges of global code generation (section 3.4), demonstrating our system's ability to adapt various FEM computations to different FEM spaces or other FEM inputs. Additionally, our code generation process ensures a careful separation between runtime and compile-time data: facilitated by the IPTEM compiler, the standard mesh entities, and the code portions of global fields, we can compile and reuse code on multiple meshes/arrays/etc. Before detailing our demand driven code generation process, we quickly detail the target.

7.8.1 Code Generation Target

In practice, we target Jax [128]. Jax supported a fair variety of COO operations via its BCOO tensor, making it suitable for the runtime. For the code generation target, we needed the ability to compile formulas (listing 5.2) as well as loops, a few numpy-like operations (meshgrid, ones, zeros, selection, shape management), and gather operations. Additionally, forward mode automatic differentiation is not strictly needed, but useful for querying the derivatives of global fields provided by pointwise operations. Jax provides all of this. We could re-target the backend to other projects such as Codon or PyTorch 2 in the future [129], [130].

7.8.2 Overall Pipeline

The overall code generation process is summarized via fig. 7.3 and listing 7.10. The former figure depicts our demand driven pipeline for adapting IPTEM, meshes, and global fields to compute our AssembledOutput, producing several different bits of data to facilitate this. The latter figure depicts how the output of the demand driven pipelines is stitched together into code that computes the SAVO. We do not go into detail as most of the steps are not that revealing - they are mainly glue; rather it is their arrangement in this manner that facilitates a single pipeline for all the various FEM tasks. We describe them abstractly, showing the information flow adopting to the inputs. Perhaps the best way to understand this is that first we set up a symbolic context for local computation, and then we extrapolate back from the compiled local computation to a global loop nest, generating code backwards from the order in which it appears.

1. First, from all the inputs, we compute the topological type and symbolic chart where computation will occur and where various fields will be queried. We just gather all topological types used in input fields and the IPTEM. Naturally we check this against the types of entities that a mesh has. However, there is one subtle decision process that must occur here: if there is an IPTEM with an integration topological type that is smaller than the topological type associated to the inputs, then for the other inputs, we might need charts to represent nearby cells; in particular, we only need them if the reduction is not choice (as in choice we can just pick one value for multiple fields). For every chart type present in an input above the integration chart, we must add one external chart that the topological enumeration process can use to represent the possible other mesh element that also includes the integration domain. See section 5.3.6 for the accompanying details on topological enumeration.

2. Second, from the inputs, we can query for symbolic field(s).

3. Third, with the charts and the symbolic fields that global fields provide, we can call

our IPTEM pipeline, providing us the local code that we will call in the innermost loop of our global code. We do this favoring the non-simplifying mode discussed in (section 5.5.2) - in other words, we maintain an output representation that looks like Einstein summation notation plus a pointwise function.

4. Fourth, from the local code, we can infer a section on a standard mesh entity, which can be used to generate the global section for the output data (section 5.5.1). In particular, we use the meta-data produced in the IPTEM process to build a schema and to track how the various output formulas map to the schema. This will be used to build the output section and to determine how to assign local outputs global section indices.

5. Fifth, also from the local code and its metadata, we can see what fields and derivatives are queried in what charts at what points, which allows us to query the tabulation interface for the code of global fields. This gets us code that computes fields and derivative values depending on the mesh entities (for indices, chart data) and other global input arrays. In some sense, this step is simply piping metadata from the IPTEM compilation to the field code generation interface.

6. Sixth, we also query the field interface for code to query the sections (where applicable) on input fields so these can be placed into the indices of the output appropriately.

7. Seventh, we can examine the usage of charts in the compiled IPTEMs and in the field code queries to determine what mesh entities and mesh relationships are queried. We can use this to set up the outermost loop that goes over some collection of mesh entities and inner loops that go over queries of mesh entities that are related via $\subset$ to the entity from the outermost loop. The inner loops over entities only exist in the case where back in step 1 we had to query inputs from neighboring cells.

8. Eighth, with the code to query mesh entities, input sections, and output sections, we

can generate code for the indices of the output.

9. Ninth, from the input fields and the required mesh relationships/entities, we can determine the various input arguments to our generated code.

10. Tenth, we can now stitch all of this together into a single output template listing 7.10. We have code for output values, output indices (mesh, sections), field queries, mesh loop structures, and arguments to the field. These are all glued together in the template.

.

```

def outputSection(mesh: Mesh, localSection: Section, location: TopType) -> Section:
    ... # Utilize (1) and section construction from schema and meshes

def compute(outSectionArrays,  # (1)
            meshTopologyArrays, # (6)
            meshVertices,
            meshRelationships,
            nTopCell,
            inputFieldSections,  inputFieldArrays, inputFieldOtherArgs,  # (4)
            ) -> Tuple[Array, Array]:
    for topCell in range(nTopCell): # vmap
        # Use (6) to get vertice/topologies of mesh
        # Use (6) to loop other related cells:
        for otherCells in ...: # vmap
            # Use (6,5) to set up geometry data
            # (6,5, 2) to set up quadrature rules/points
            # (6,5, 3) to query fields
            # (0, 5,4) to compute block of output values
            # Use (1,4,6) to compute block of indices of output values
    # Return all blocks of values and indices

def outputCode(mesh, inputLinearFields, inputNonLinearGlobalFields):
    localSection = # Use (1)
    topType = # Use 1
    outputSection = outputSection(mesh, localSection, topType)
    # Do setup for mesh
    meshTopologyArrays = # From mesh and (5)
    meshVertices = mesh.vertices
```

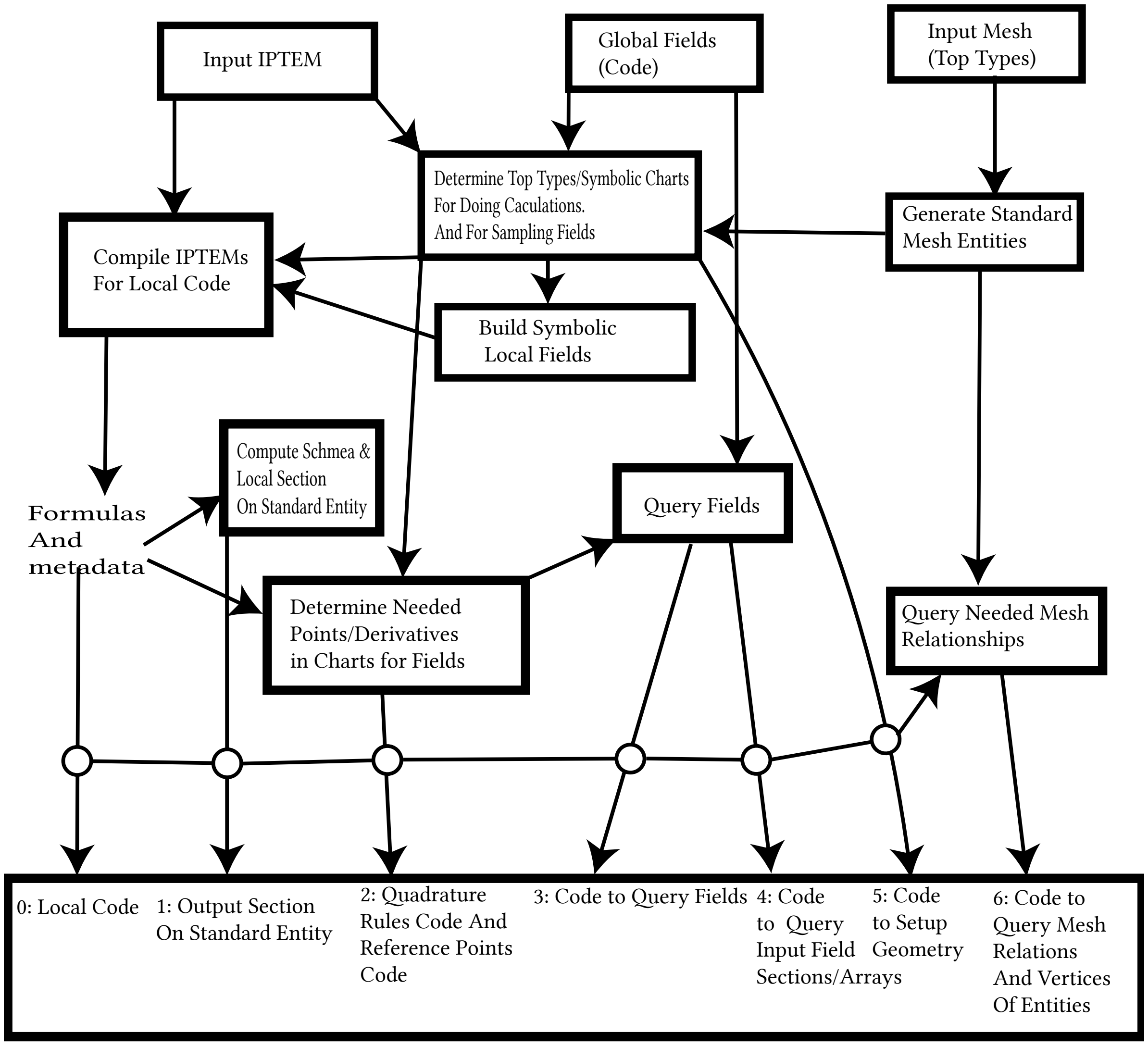

Input IPTEM
Global Fields (Code)
Input Mesh (Top Types)
Determine Top Types/Symbolic Charts For Doing Caculations. And For Sampling Fields
Generate Standard Mesh Entities
Compile IPTEMs For Local Code
Build Symbolic Local Fields
Compute Schmea & Local Section On Standard Entity
Query Fields
Formulas And metadata
Determine Needed Points/Derivatives in Charts for Fields
Query Needed Mesh Relationships
0: Local Code
1: Output Section On Standard Entity
2: Quadrature Rules Code And Reference Points Code
3: Code to Query Fields
4: Code to Query Input Field Sections/Arrays
5: Code to Setup Geometry
6: Code to Query Mesh Relations And Vertices Of Entities

Figure 7.3: Our global code generation pipeline up to the final stitching together of the code.

```
    meshRelationships = # From mesh and (5)
    nTopCell = # From mesh and (5)
    # For each input, set up, creating a section if it doesn't exist:
    inputFieldSections = []
    inputFieldArrays = []
    inputFieldOtherArgs = []
    for field in inputLinearFields:
        # Append args
    for field in inputNonLinearGlobalFields:
        # Append args
    (vals, indicies) = compute(outSectionArrays, meshTopologyArrays, meshVertices,
        meshRelationships, nTopCell, inputFieldSections, inputFieldArrays,
        inputFieldOtherArgs)
    return (outputSection, vals, indices) # To be wrapped in AssembledOutput with metadata.
```

Source Code Listing 7.10: A visual depiction of how the various intermediate outputs of our global code generation process are stitched togeather to produce a global kernel function.

7.9 Applications

As an application of our infrastructure and evidence of its adaptability and expressivity, we show how to use it to accomplish various tasks. Some tasks, such as interpolation for general elements or assembly of general operators, emerge straightforwardly from our interface. Interpolation can always be accomplished via `genericAssembly(mesh, finiteElement, [inputField ]).asInput()` and operator assembly via `genericAssembly(mesh, operatorIPTEM, [inputDofsField1, inputDofsField]) .asScipyMatrix(format="csr")`. A more impressive task is general boundary conditions, as we explained in section 2.6. We can chain our interface to build a boundary condition enforcement infrastructure that can adapt to the incoming mesh and element. To demonstrate this, we provide a version of `assembleConstraints` that detects a set of boundary conditions for elimination.

See listing 7.11. Our algorithm uses the idea from Nitsche's method described in section 2.4; we observe that enforcing a boundary condition $Bu = f$ via an elimination method requires that the term, $\int_{\partial\Omega} BvBu$, be zero on DOFs not associated to the boundary. When this is the case, we expect the extra terms in a weak formulation associated to BCs to vanish. Thus, we can assemble the operator on the boundary and use our where functionality to extract the DOFs that impact boundary integrals. Then we can compare this to the DOFs that are associated to the boundary topologically, using propagation of a sub-mesh to a section to create a subsection. If the relevant DOFs are all topologically associated to the boundary, we can eliminate them, returning the subsection; otherwise, we must return None. The assembleConstraints function uses this subsection or lack-thereof to provide an object users can interrogate to enforce BCs.

```python


def assembleConstraintsFindSubSection(mesh, meshSubset, operatorPointwise, dofs):

    # Form Boundary operator:
    integral = Integral(BoundaryChart)
    operatorPre = Pointwise([("u", dofs.field()), ("v", dofs.field())], lambda u,v:
        operatorPointwise(u).dot(operatorPointwise(v)))
    IPTEM = IntegratedPointwiseTensorExpr(operatorPre, integral, BoundaryChart)

    # Assembly boundary matrix
    oper = genericAssembly(mesh, IPTEM, [dofs, dofs]).reduce(["InputDofs0", "InputDofs1", "
        IntegrationDomain"])
    # Set all non−boundary integrals to zero.
    oper[:, :, ~meshSubset] = 0.0
    # Extract boundary DOFs that impact boundary integral:
    oper = oper.reduce(["InputDofs0", "InputDofs1"])
    usedEntities = oper.where(["inputDofs0"])
    # Propagate Dofs to get the Dofs based on the sub−mesh.
    standardBoundaryDofs = propagate(meshSubset, oper.inputSections[0])
    # an elimination method requires that every entity that impacts boundary condition is on
         the boundary
    if usedEntities <= standardBoundaryDofs::
        return usedEntities
    else:
        return None
```

Source Code Listing 7.11: An algorithm to determine if the DOFs that influence a boundary condition are associated to the boundary, allowing them to be eliminated from the linear system.

Chapter 8

Evaluation

To evaluate our system, we construct several applications of interest to computer graphics where we can utilize a variety of elements: a biharmonic smoothing energy, a Hodge Decomposition, and a Stokes flow. For each application and relevant element, we validate the application via convergence plots and we also supply a simplistic analysis of some of the trade-offs between elements. Thus, these applications demonstrate the potential utility of changing elements to computer graphics applications. In some cases, these results are known, though they have never been achieved in a truly automated fashion and are not widely available to the graphics community.

To evaluate how our system achieves the above, we conduct three experiments to show that:

1. Our system can specify FEs concisely compared to other systems, in large part due to our automatic capabilities.

2. Our system can specify FEs in a malleable manner: the code required to implement new elements given the infrastructure of old elements is small, so the code to vary elements is small.

3. Our system's automation does not cost us when it comes to the FE transformation

theory - the code generated by our system require roughly the same work as compared to the transformation theory deduced by hand.

Through the variety of applications, we hope to demonstrate that our system is sufficiently expressive to specify most elements that are available in the world.

Taken together with the applications we believe these demonstrate that:

1. Our system offers a complete and concise mathematical specification of a FE.

2. Through our system's specification of FEs and operators, we can achieve novel automation of the use of FEs for a variety of problems.

3. Our system creates simulations of broader interest to a variety of communities.

8.1 Applications

8.1.1 Biharmonic Energies

8.1.1.1 Problem Setup

Fourth-order problems are a central motivator for the development of scalar elements beyond the Lagrange element. The model fourth-order problem is the biharmonic equation:

$$\Delta^2 u = f. \tag{8.1}$$

The biharmonic equation emerges in many contexts, and we study several representing geometry processing and simulation concerns. Our contexts include a variety of BCs, including Dirichlet conditions:

$$u|_{\partial\Omega} = h \text{ and } \frac{\partial u}{\partial n} = g \tag{8.2}$$

for some given scalar functions h, g. In simulation contexts, these are sometimes known as clamped plate BCs. Simulation contexts also utilize a form of natural BC given by

$$\frac{\partial u}{\partial n} = g \text{ and } \frac{\partial \Delta u}{\partial n} = h. \tag{8.3}$$

However, we also study the natural BCs, which occur in smoothing applications [20], [58]:

$$nH_u n = 0 \text{ and } \frac{\partial \Delta u}{\partial n} + \frac{\partial u}{\partial ntn} = 0 \tag{8.4}$$

where t is a vector tangent to the boundary. In smoothing applications, we also see interpolation constraints:

$$u(x_i) = f_i \tag{8.5}$$

where $x_i \in \Omega$ (typically vertices of the mesh) and $f_i \in \mathbb{R}$.

Finally, any of these BCs could be combined with a biharmonic eigenvalue problem:

$$\Delta^2 u = \lambda u. \tag{8.6}$$

For example, the solution to the eigenvalue problem under the Dirichlet conditions provides the vibrational modes of a surface clamped at its boundaries.

8.1.1.2 Operators

Using our system, we implement the following operators to represent the biharmonic equation. We have discussed these before in the examples, so we will not provide code here. We use a Hessian-Hessian inner product for the equation:

$$a_H(u, v) := \int_\Omega (H_u, H_v). \tag{8.7}$$

Naturally, for the right hand side or the eigenvalue problem, we need a mass operator:

$$m(u,v) = \int_\Omega uv. \tag{8.8}$$

To implement various BCs variationally (Neumann or Dirichlet via Nitsche's method [131]), we utilize terms from the following operator:

$$a_B(u,v) := \int_{\partial\Omega} v\frac{\partial \Delta u}{\partial n} + \int_{\partial\Omega} \frac{\partial v}{\partial n}\Delta u + \int_{\partial\Omega} uv + \int_{\partial\Omega} \frac{\partial u}{\partial n}\frac{\partial v}{\partial n}. \tag{8.9}$$

For natural BCs in smoothing, we do not need operators to enforce them. To enforce point constraints, we need linear operators that correspond to point evaluation at a fixed set of points $\{x_i\}_{i=1}^N$:

$$P(u) = (u(x_1), \dots, u(x_N)) \in \mathbb{R}^N. \tag{8.10}$$

Also, to validate the BCs numerically[1], it is useful to have the following non-linear operator:

$$e_B(u) = \int_{\partial\Omega} (nH_u n)^2 + (\frac{\partial \Delta u}{\partial n} + \frac{\partial u}{\partial ntn})^2. \tag{8.11}$$

Similarly, to check if an element is valid to use for a biharmonic problem on a given mesh (some unconditionally work, but others do not), it is useful to consider the patch test operator that operates on the interior edges $\cup e_i \subset \Omega$ with the notation that $K_{0,i}$ and $K_{1,i}$ are the two cells that include e_i:

$$t(u,v) = \sum_i \int_{e_i} (\nabla u|_{K_{0,i}} - \nabla u|_{K_{1,i}}) \cdot v. \tag{8.12}$$

If v is drawn from discontinuous piecewise linear functions defined the set of edges, then $t \equiv 0$ when u is drawn from a given global finite element space is a necessary condition to use an element to solve a biharmonic problem.

Since we wish to compare to prior work, mainly [20], [132], we need operators to imple-

[1]As a minor note, we used this to provide the first numerical validation of the BC offered in [20]

ment a mixed method for the biharmonic equation. These come from the Laplace equation except for smoothing, where we need a few additional operators, including an inner product on matrix-valued functions that represent Hessians and an integration of the divergence of a matrix against the gradient of a scalar function:

$$a_{\mathrm{Div},\nabla}(\Lambda, u) = \int_\Omega \nabla \cdot \Lambda \cdot \nabla u. \tag{8.13}$$

8.1.1.3 Elements

We use the following elements on this problem: affine Lagrange, Morley, Hermite, Bell, Argyris. For the purpose of later analysis, we provide code. The Lagrange element in our system is:

```
cty0 = ChartType(0)
@pointwiseEval(, barycentricCoordinate=0.0)
def vertexEval(u: field(cty0, R)):
    return u(0)
LagrangeTri = Sybil.FE(2, 1, [vertexEval])
```

Source Code Listing 8.1: Affine Lagrange DOFs

The Morley element uses quadratic polynomials, the affine Lagrange DOFs, and evaluations of the gradient normal at midpoints. We note that the normal direction is computed with a global Hodge star on Euclidean space to ensure the value is consistent on edges examined from different cells:

```
cty1 = ChartType(1) # On objects of dimension 0
@pointwiseEval(cty1, barycentricCoordinate=(0.5, 0.5))
def morleyDof(c: cty1, u: field(cty, R, 1, euclideanDerivative=True)):
    return P.hodge(u(1)).dot(c.E(c.tangentVectorBasis(0)))
Morley = EF.FE(2, 2, [vertexEval, morleyDof])
```

Source Code Listing 8.2: Morley DOFs

Hermite DOFs are cubic and use Lagrange DOFs plus edge gradients and a center evaluation.

```python
fld = field(cty0, R, 1, euclideanDerivative=True)
@pointwiseEval(cty0, barycentricCoordinate=0.0)
def xDervDof(u: fld):
    return u(1)[0]
@pointwiseEval(cty0, barycentricCoordinate=0.0)
def yDervDof(u: fld):
    return u(1)[1]
cty2 = ChartType(2)
@pointwiseEval(cty2, barycentricCoordinate=(1/3, 1/3, 1/3))
def centerDof(u: field(cty2, R, 0)):
    return u(0)
HermiteDofs = EF.FE(2, 3, [vertexEval, xDervDof, yDervDof, centerDof])
```

Source Code Listing 8.3: Hermite Dofs

The Argyris DOFs utilize the Hermite gradient DOFs plus the Morley DOFs and three additional Hessian DOFs per vertex:

```python
fld = field(cty0, R, 2, euclideanDerivative=True)
@pointwiseEval(cty0, barycentricCoordinate=0.0)
def xxDervDof(u: fld):
    return u(2)[0,0]
@pointwiseEval(cty0, barycentricCoordinate=0.0)
def yyDervDof(u: fld):
    return u(2)[1,1]
@pointwiseEval(cty0, barycentricCoordinate=0.0)
def xyDervDof(u: fld):
    return u(2)[0,1]

ArgyrisDof = EF.FE(2, 5, [vertexEval, xDervDof, yDervDof, morleyDof, xxDervDof, yyDervDof,
    xyDervDof])
```

Source Code Listing 8.4: Argyris DOFs

Finally, the Bell DOFs uses the Argyris DOFs except the Morley DOF. Instead, the Bell space consists in quintic polynomials whose normal gradients are cubic polynomials on each edges. To enforce this, we add Morley-like DOF that use the Legendre DOFs to the space:

```
@pointwiseIntegral(cty1)
def bellSpaceDof(c: cty1,
                 u: field(cty1, R, 1, euclideanDerivative=True),
                 v = legendreDofs(5).basisFunction(4)):
    return u(1).dot(c.normalBasisVector(0)) * v(0)
BellDofs = EF.FE(2, P(5, nullDofs=[bellSpaceDof]), [vertexEval, xDervDof, yDervDof,
    xxDervDof, yyDervDof, xyDervDof])
```

Source Code Listing 8.5: Bell Dofs

The Legendre DOFs are just L2 duals of the monomials:

```
def legendreDofs(deg: int):
    dofs = []
    for d in range(deg + 1):
        @pointwiseIntegral(ChartType(1))
        def legendreDof(c: ChartType(1), u: field(1, R)):
            return (c.x**d) * u(0)
        dofs.append(legendreDof)
    return EF.FE(1, P(deg), dofs)
```

Source Code Listing 8.6: Legendre DOFs

8.1.1.4 Validation

To validate our system, we provide a convergence study on a model problem: the biharmonic equation with homogeneous Dirichlet BCs on the unit square. We use the analytical solution $u(x, y) = (x(x-1)y(y-1))^2$. We compared the mixed method from [20], [132] to an implementation using the operators in (8.7) and (8.8) with boundaries enforced by querying our system for the nodes corresponding to the Dirichlet BCs. This amounts to a few lower level calls made automatically in the system to the last two terms of the operator in (8.9). By comparing to other literature [63], we have validated that the DOFs selected for the BCs are correct. Since our particular variation of the Argyris and Bell elements only supports BCs on certain types of meshes (compare to [133], [134]), we also provide convergence data for a Nitsche method formulation [131] that utilizes all of the terms in (8.9). The results are

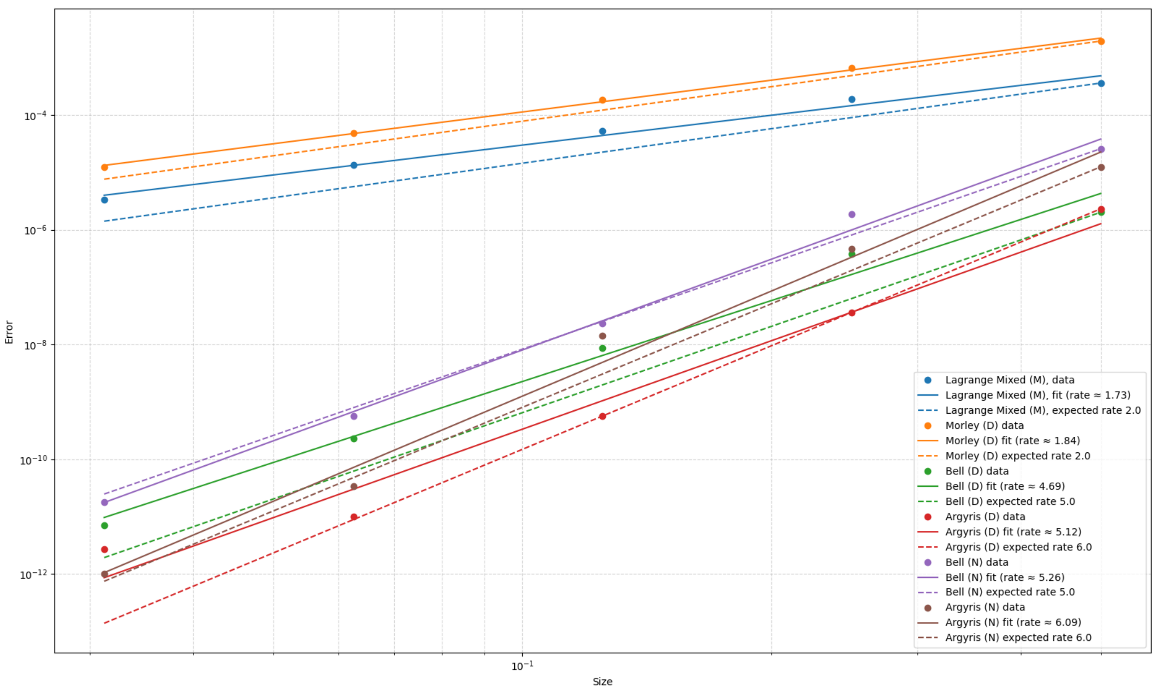


Figure 8.1: Convergence data for fig. 8.2.

depicted in fig. 8.2 and fig. 8.1. The results validate the system, as convergence is in line with theoretical predictions.

To validate the system on a more realistic mesh, we provide the first six eigenvectors of the Hessian operator with natural BCs, allowing a visual comparison with [20]. To make the comparison possible here, we also provide the first six eigenvectors via the mixed method described in [20]. The results in fig. 8.3 show that the first six eigenvectors have roughly the expected structure on a more complex mesh. We also include Hermite elements in this example to show that they provide visually reasonable results, although they do not pass the patch test and do not converge on meshes used in our convergence validation.

8.1.1.5 Trade-Offs

To visually motivate our discussion of the trade-offs between elements for Biharmonic problems, we consider the interpolation problems from [20]. We replicate the mixed method described in [20] and utilize a direct method via (8.7) with four elements: Morley, Hermite,

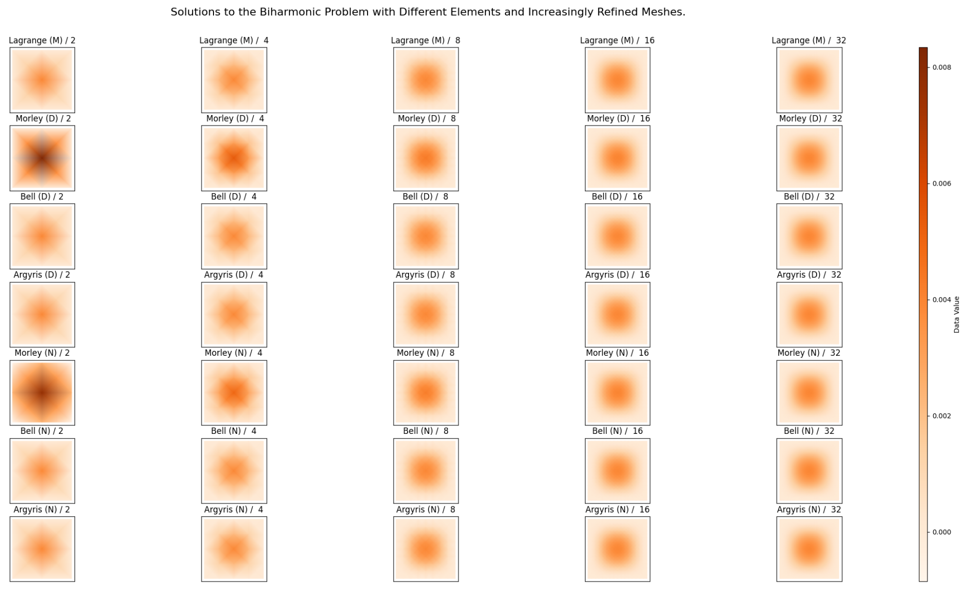


Figure 8.2: Visualization of Solutions to the Biharmonic equation on the square with different elements and meshes. Meshes were refined by subdividing into squares and converting each square into four triangles. Methods labeled M are mixed, N use Nitsche, and D use standard Dirichlet boundary conditions. All function were rendered via evaluating at vertex values and linearly interpolating between these.

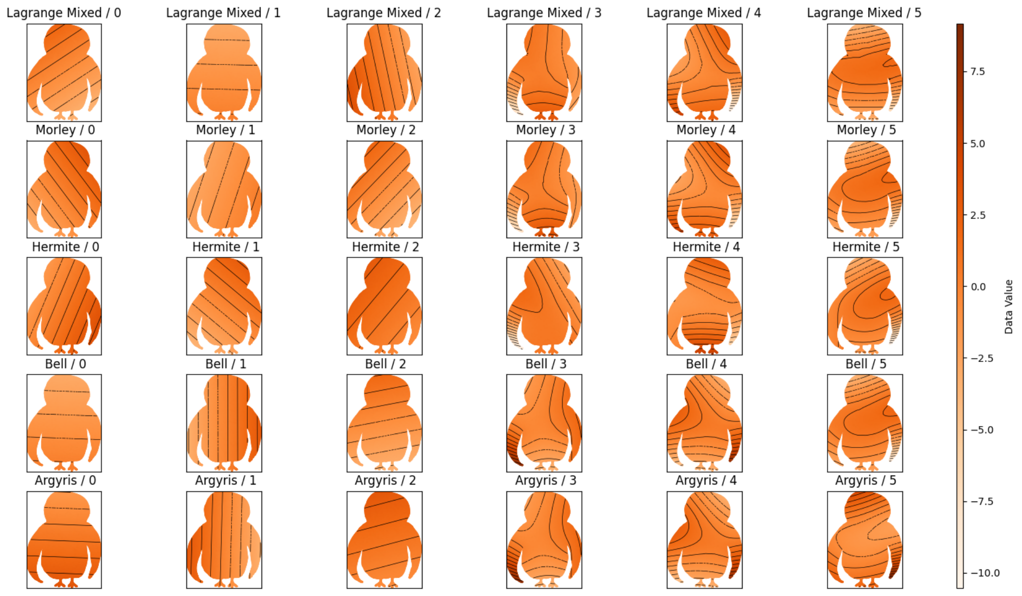


Figure 8.3: The first six eigenfunction of the Hessian energy operator with natural boundary conditions as described in [20]. All functions were rendered via evaluating at vertex values and linearly interpolating between these.

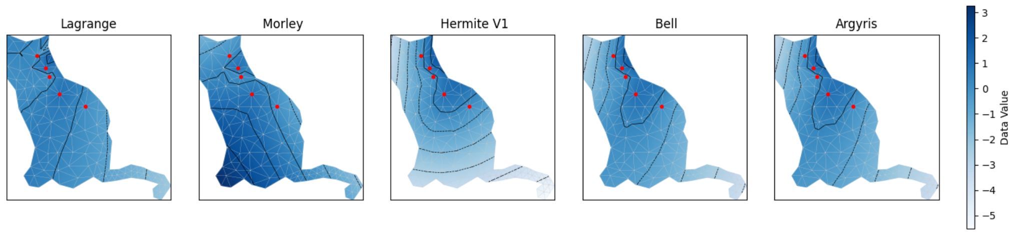


Figure 8.4: Biharmonic interpolation on a low poly mesh via different elements. The five points on the figure were interpolated from top to bottom to the values $1, 0, 1, 0, 1$. All functions were rendered via evaluating at vertex values and linearly interpolating between these.

Bell, and Argyris. We apply all methods to a low-resolution mesh (210 elements). We note that constructing a low-resolution mesh that worked on the mixed method was a struggle, as the mixed method fails when two edges of a triangle are on the boundary. In contrast, the Morley, Bell, and Argyris elements are valid on any conforming meshes for this problem. The results are depicted in fig. 8.4, where all are visualized by evaluating the solution at each vertex (so all solutions are still rendered as linear functions). Visually, we see that distortions due to low poly count disappear as we use higher-order elements. What is striking about this result is that via various machine-independent metrics, several higher-order elements were cheaper: they used fewer DOFs per element and thus required smaller sparse matrices. Of course, this is just one problem; the results are much more striking than the standard convergence results in fig. 8.2; there, the cost calculation might be more complex, especially if we explore variations of BCs. There are many vicissitudes to explore, but to tease and conserve space, we summarize some trade-offs between elements for biharmonic problems in table 8.1.

Table 8.1: A table of various trade-offs between elements. The first three rows give the number of DOFs used per triangle to solve certain types of problems. Then come rows indicating which formulations are supported (e.g., mixed methods, naive forms, IPO). The next three rows concern enforcement of boundary conditions. Transform NNZ is the number of non-zeros in the transform matrix.

Trade Off/Element	**Lagrange 1**	**Lagrange 2**	**Lagrange 3**	**Morley**	**Hermite V1**	**Hermite V2**	**Bell**	**Argyris**
DOFs Per Tri	3	6	10	6	10	10	18	21
DOFs Per Tri BiLaplce	6	12	10	6	10	10	18	21
DOFs Per Tri Hessian	15	30	10	6	10	10	18	21
Supports Form (Mixed, IPO, Naive)	Mixed	Mixed	IPO	Naive	All, but Naive depends on geometry	All, but Naive depends on geometry	All	All
Supports Homogeneous Dirichlet BCs Naively	Yes	Yes	Yes	Yes	Depends on geometry	Only $u = 0$, but not $\frac{\partial u}{\partial n} = 0$	Depends on geometry	Depends on geometry
Supports In-Homogeneous Dirichlet BCs Naively	Unknown	No	No	Yes	Depends on geometry	No	Depends on geometry	Depends on geometry
Supports In-Homogeneous Dirichlet BCs via Nitsche	No	No	Yes	Yes	Yes	No	Yes	Yes
Transform NNZ	3	6	10	12	16	10	45	63
Continuity	C^0	C^0	C^0	Minimal element that passes the patch test	C^1 at verts	C^1 at edges	C^1	C^1
Ideal Conv Rate	h^2	Unknown	h^4	h^2	h^4	h^4	h^5	h^6
Convergence Defects	Fails on meshes with corners	Unknown	Unknown		Fails patch test on some meshes	Fails patch test on some meshes		

8.1.2 Hodge Decomposition

8.1.2.1 Problem Setup

To set up this problem, we must freely utilize the language of exterior calculus. Beyond what we have already discussed, we need two additional notions, the Hodge star and the co-differential. These build on two notions described in prior sections: the volume form and the inner product on forms.

Definition 8.1.1 (Hodge Star, Co-Differential)**.** Let V be an n-dimensional vector space and $0 \leq k \leq n$. Let $w \in \Lambda^k V$ be given. Let $dV \in \Lambda^n V$ be the volume form. The **Hodge star** of w, $\star w$ is an element of $\Lambda^{n-k} V$ such that for all $v \in \Lambda^k V$,

$$v \wedge \star w =< v, w > dV. \tag{8.14}$$

If N is an n surface, then for $\omega \in C^1(N, \Lambda^k \mathcal{T} N)$, the **Hodge star** is given by

$$(\star\omega)(x) := \star(w(x)). \tag{8.15}$$

Finally, the co-differential of a k form is a $k-1$ form given by:

$$\delta\omega = \star d \star w. \tag{8.16}$$

These notions are sufficient to allow us to define the Hodge decomposition on a manifold with and without boundary. The former is sometimes called the five term decomposition while the latter is sometimes called the three term decomposition. Further variations on these decompositions are available when more complex varieties of BCs are used [135], but we stick to the standard variations.

Definition 8.1.2 (Hodge Decomposition)**.** Suppose N is an n manifold without boundary. Let ω be a k form with $0 \leq k \leq n$. The Hodge decomposition of ω is a triple of k forms, α, β, η, such that

1. $d\alpha = \delta\beta = (d\delta + \delta d)\eta = 0$,
2. and $\omega = \alpha + \beta + \eta$.

If N has boundary, then the Hodge decomposition is a tuple of 5 k forms, $\alpha, \beta, \eta_t, \eta_n, \eta$ such that

1. $d\alpha = \delta\beta = (d\delta + \delta d)\eta = (d\delta + \delta d)\eta_t = (d\delta + \delta d)\eta_n = 0$,
2. and $\omega = \alpha + \beta + \eta_t + \eta_n + \eta$,
3. and $\alpha|_{\partial\Omega} = \star\beta|_{\partial\Omega} = \eta_t|_{\partial\Omega} = \star\eta_n|_{\partial\Omega} = 0$.

Computing the Hodge decomposition has many applications, most notably as a primitive for solving many PDEs in fluids and electromagnetics [136]. Additionally, the operators used to define the Hodge decomposition are at the core of vector field design problems [137].

8.1.2.2 Operators

Traditionally, each Hodge decomposition amounts to the solution of two vector Poisson equations and two vector Poisson eigenvalue problems. The Poisson equations allow us to compute the α and β terms while the eigenvalue problems allow us to compute η_t, η_n, η by projecting $\omega - \alpha - \beta$ to the null space of the two vector Poisson equations. Further, we need two differential operators to set up the right hand side of these equations $(d\omega, \delta\omega)$ and two of the same differential operators to process the results to form α and β. A recent example of such an approach is [138]. However, this involves a huge array of operators because for each choice of k and n, a variety of differential operators might be involved. We will avoid this complexity by simply specifying the operators involved using only d and the inner product on forms. We need only four operators that operate on pairs of either two k forms, or k and $k-1$ forms, or two $k-1$ forms:

$$m(u, v) = \int_N < u, v > \tag{8.17}$$

$$m_{\star,d}(u, v) = \int_N < u, dv > \tag{8.18}$$

$$m_{d,\star}(u, v) = \int_N < du, v > \tag{8.19}$$

$$m_{d,d}(u, v) = \int_N < du, dv > . \tag{8.20}$$

With the right choice of element, the BCs are Dirichlet or natural. Next, we need to use d and δ pointwise to set up the right hand sides. Since the first four are similar, we provide code for $m_{d,d}$:

```
def ddoperator(formDeg: int, cell: int):
    fld = field(ChartType(cell), Lambda(formDeg, cell))
    @pointwiseIntegral(ChartType(cell))
    def mdd(u: fld, v: fld):
        return P.antiSymmeterize(u(1)).dot(P.antiSymmeterize(v(1)))
    return mdd
```

Source Code Listing 8.7: Example operator for FEEC

We also provide an example with δ:

```
def deltaOperator(formDeg: int, cell: int):
    fld = field(ChartType(cell), Lambda(formDeg, cell))
    fldp = field(ChartType(cell), Lambda(formDeg - 1, cell))
    @pointwise(ChartType(cell))
    def delta(u: fld) -> fldp:
        return P.hodge(P.antiSymmetrize(P.D(P.hodge(u(0)))))
    return delta
```

Source Code Listing 8.8: Implementation of a pointwise delta operator.

8.1.2.3 Elements

Just as with operators, there are elements for different choices of k and n. Also as with operators, these can be succinctly unified into a few choices, parametrized by dimension of the space/surface N, form degree (k), polynomial degree (r), Hodge duality ($\star$), and trim ($-$). These elements are labeled $P_r\Lambda^k(N)$, $P_r^-\Lambda^k(\mathcal{T}N)$, $P_r^\star\Lambda^k(N)$, $P_r^{-\star}\Lambda^k(N)$. Together these capture Lagrange elements, discontinuous Lagrange elements, Nedelec (type one and two) (face and edge) elements, Brezzi-Douglas-Marini elements, and Raviart Thomas elements.

We first describe these mathematically. To describe the spaces, we need the Koszul operator κ and also the space of homogeneous polynomials (all terms have the same degree except 0) H_r. The latter gives rise to the space of homogeneous polynomial form spaces $H_r\Lambda^k$. The first polynomial space was defined in the background, and we provide the other three here:

1. $P_r^-\Lambda^k(\mathcal{T}N) := P_{r-1}\Lambda^k(\mathcal{T}N) + \ker_{H_{r+1}\Lambda^{k-1}(\mathcal{T}N)}\kappa$, the degree $r-1$ k forms plus the

kernel of the Koszul derivative on the homogeneous polynomial degree $r+1$ $k-1$ forms. These are the so called trimmed spaces.

2. $P_r^{\star}\Lambda^k(\mathcal{T}N) := \star P_r\Lambda^{n-k}(\mathcal{T}N)$, the space of k forms represented via Hodge stars of $n-k$ forms.

3. $P_r^{-\star}\Lambda^k(\mathcal{T}N) := \star P_r^{-}\Lambda^{n-k}(\mathcal{T}N)$, similar to the above but using trimmed spaces.

For each of these elements, the DOFs can be defined via mutual recursion. The base case is the Lagrange elements. We can define the DOFs on k forms and simplices f of dimension d using forms of smaller degree defined on smaller simplices:

1. $P_r\Lambda^k(\mathcal{T}f)$: for every face $f' \subset f$, let $\{\eta_{f',i}\}$ be a basis for $P^{-}_{r+k-\dim f'}\Lambda^{\dim f'-k}(\mathcal{T}f')$. The DOFs are
$$\phi_{f',i}(u) = \int_{f'} (u|_{f'}) \wedge \eta_{f',i}$$
for all $f' \subset f$ and all valid i.

2. $P_r^{-}\Lambda^k(\mathcal{T}f)$: for every face $f' \subset f$, let $\{\eta_{f',i}\}$ be a basis for $P_{r+k-\dim f'-1}\Lambda^{\dim f'-k}(\mathcal{T}f')$. The DOFs are
$$\phi_{f',i}(u) = \int_{f'} (u|_{f'}) \wedge \eta_{f',i}$$
for all $f' \subset f$ and all valid i.

3. $P_r^{\star}\Lambda^k(\mathcal{T}f)$ or $P_r^{-\star}\Lambda^k(\mathcal{T}f)$: Similar to the previous formula but
$$\phi_{f',i}(u) = \int_{f'} (\star u|_{f'}) \wedge \star\eta_{f',i}$$

The first two spaces capture the tangential components with forms, while the latter two capture the orthogonal complements of the tangent space. (Practically, the first two favor Dirichlet BCs on u, while the latter two favor Dirichlet BCs on $\star u$). The trimmed spaces provide smaller, more economical versions that are harder to interpret.

We provide code to capture all of these cases in stages. First, we provide the DOFs to define the Koszul space.

```python
def koszulDofs(formDeg: int, degree:int, dim: int):
    evals = []
    dsf = filter(
        lambda x: sum(x) == degree + 1,
        product(*[range(degree + 2) for _ in range(dim)]),
    )
    for ds in dsfl:
        for k in range(int(binom(dim, formDegree - 1))):
            @P.pointwiseIntegrate(CellType(dim))
            def koszul_eval(c:  ChartType(dim), f: field(dim, Lambda(formDegree, dim)):
                x = c.P()(c.x)
                xx = np.prod([c.x[j] ** ds[j] for j in (range(dim))])
                r = P.interiorEval(f(0), [0], [x])
                return (
                    r[(k,)] * xx * c.dV()
                )
        evals.append(koszul_eval)
    return evals
```

Source Code Listing 8.9: Implementation of Koszul DOFs.

We define the Koszul DOFs to isolate the kernel of the Koszul operator. We do this by testing the Koszul operator with every relevant monomial and form component. Then, we can define our spaces:

```python
def feecSpace(formDeg: int, degree:int, dim: int, hodge: bool, trim: bool):
    if not trim:
        if not hodge:
            return PImpl(dim, degree, Lambda(formDeg, dim))
        else:
            return PImpl(dim, degree, Lambda(dim-formDeg, dim))
    else:
        Prm1 = feecSpace(formDeg if not hodge else dim - formDeg , max(degree - 1, 0), dim,
            hodge, trim)
        if formDegree  == 0 or formDegree == dim:
            return Prm1
        kdofs = koszulDofs(formDeg if not hodge else dim - formDeg, degree, dim)
        return Prm1 + NullSpace(dim, degree, minDegree=degree, nullDofs=kdofs)
```

Source Code Listing 8.10: Implementation of FEEC spaces, utilizing Koszul DOFs for the Koszul space.

Finally, we can define our elements modulo a few lines of code to specify the base cases:

```
def feec(formDeg: int, degree:int, dim: int, hodge: bool, trim: bool):
    if formDeg == 0 or formDegree = dim and hodge:
        return langrangeDofs(dim, degree if not trim else degree - 1)
    assert dim >= formDeg >=0 and degree >=0
    space = feecSpace(formDeg, degree, dim, hodge, trim)
    md = min(dim, formDegree + degree - 1)
    evals = []
    for j in range(formDegree, md + 1):
        ct = ChartType(j)
        field = field(j, P.Lambda(formDegree, ct)
        recDegree = degree - j + formDegree - 1 if trim else degree - j + formDegree
        recDofs = feec(j, recDegree, j - formDegree, trim=not trim, hodge=hodge)
        for basisFunc in recDofs:
            if not hodge:
                @P.pointwiseIntegrate(ChartType(j), reduction=choice)
                def form_eval(u: field, v: recDofs.field() = basisFunc):
                    return P.hodge(P.wedge([u(0), v(0)]))
            else:
                @P.pointwiseIntegrate(ChartType(j), reduction=choice)
                def form_eval(u: field, v: recDofs.field() = basisFunc):
                    return P.hodge(P.wedge([P.hodge(u(0)), P.hodge(v(0))]))
            evals.append(form_eval)
    return Dofs(space, evals)
```

Source Code Listing 8.11: Implementation of FEEC elements.

For our base case, we must also provide arbitrary order Lagrange DOFs. We provide these in the Stokes section.

8.1.2.4 Validation

We use two problems to validate our solvers and provide convergence data. First, we consider a flat annulus of outer radius 2 and inner radius 1. We decompose the vector field:

$$\omega(x, y) = (2(x - y) + 1, 2(x + y) + 1). \tag{8.21}$$

The analytical form of the Hodge decomposition is known. Set $vx(x, y) = \frac{x}{||(x,y)||^2}$ and $vy(x, y) = \frac{y}{||(x,y)||^2}$. Then we can write the decomposition via

$$d\alpha(x, y) = (2x - \frac{15}{\log 4} vx(x, y), 2y - \frac{15}{\log 4} vy(x, y)) \tag{8.22}$$

and

$$d\beta(x, y) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} d\alpha(x, y) \tag{8.23}$$

and

$$hn(x, y) = (\frac{15}{\log 4} vx(x, y), \frac{15}{\log 4} vy(x, y)) \tag{8.24}$$

and

$$ht(x, y) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} hn(x, y) \tag{8.25}$$

and finally,

$$\eta(x, y) = (1, 1). \tag{8.26}$$

Second, we consider a cylinder of radius 1 and height 1. We further specify that this is a manifold with a metric inherited from $\mathbb{R}^3$. To write this down with clarity, we use two bits of notation. We utilize cylindrical coordinates, (θ, z), and Cartesian, (x, y, z), simultaneously. We also utilize an orthonormal frame for the tangent space via

$$tu(\theta, z) = (-\sin\theta, \cos\theta, 0) \tag{8.27}$$

and

$$tv(\theta, z) = (0, 0, 1). \tag{8.28}$$

Then the vector field

$$\omega(\theta, z) = (\theta^2 + \theta - z^2 + 2z)tu(\theta, z) - 2(z + z\theta)tv(\theta, z) \tag{8.29}$$

has a decomposition as follows:

$$d\alpha(\theta, z) = (0, 0, z - 0.5). \tag{8.30}$$

and

$$d\beta(\theta, z) = (-2y(z - 0.5), 2x(z - 0.5), 0) \tag{8.31}$$

and

$$hn(\theta, z) = -tv(\theta, z) \tag{8.32}$$

and

$$ht(\theta, z) = (-4y, 4x, 0), \tag{8.33}$$

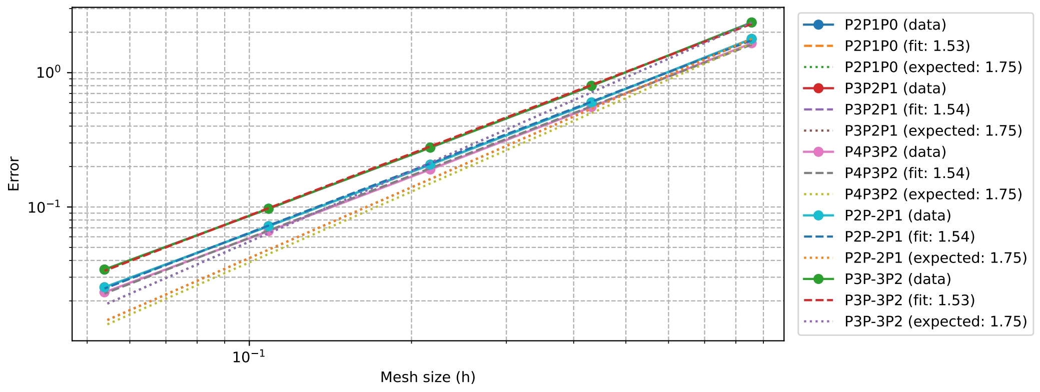
and finally we can infer eta from the rest.

For both cases, we must utilize the theory of variational crimes to temper the validation experiments; we should not expect convergence higher than second order because we are using a linear approximation of the geometry [139]. We provide convergence plots for the annulus scenario in fig. 8.5. For the convergence in the cylinder scenario, we need a further proviso. Literature on convergence in the surface scenario of individual components of the algorithm is sparse, and most articles on the decomposition do not perform this analysis, sometimes reporting a difficulty in the convergence of the individual components of the harmonic part of the decomposition [135], [138], [140]–[143]. This might be a Babuška paradox type situation [144], but it might also be that the convergence in the theory applies to the PDEs but not the classic Hodge decomposition algorithms. Thus, in this scenario we report the error in the overall harmonic part, which does converge. Further, we checked that each component has the correct angle with an interpolation of the true solution (zero to machine precision). These indicate that the elements and solvers are sound, but the steps to produce the five term Hodge decomposition (as found in many places) are not sound. With these provisos, convergence is reported in fig. 8.6.

We also provide samples of the lowest and highest resolution solution for the five components and for a few elements. True solutions are in fig. 8.7 and fig. 8.8.

8.1.2.5 Trade Offs

The trade-offs for higher order representations of forms have been noted throughout the literature [27], [28], [56], [145], including applications such as vector field design, parameterization (fixed boundary, seamless), vector heat methods, and solutions to fluidic equations. In each of these, the story is roughly the same: higher-order elements provide smoother and more accurate solutions that benefit downstream applications at the expense of more costly elements. We can provide another data point by computing one of the zero eigenvectors of the Hodge Laplacian on a low resolution mesh via a variety of elements. We compare this to a higher resolution mesh and low order element scenario in fig. 8.9. We follow [145] in this comparison. We note that not only are there visual distortions due to the lower order elements (jumps in color), but that topology of the vector field is not accurately represented: the lowest orders do not capture the critical point in the center of the field and heavily distort the flow along the lines of the coarse geometry.



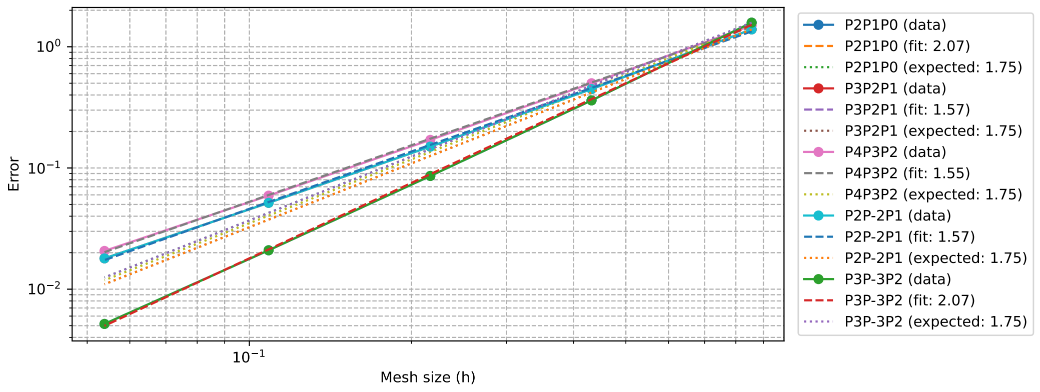
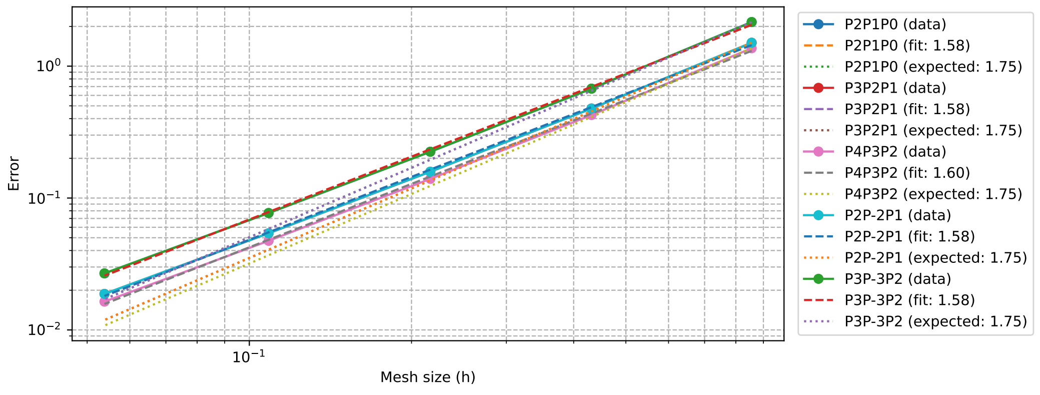


Figure 8.5: Convergence plots for the exact, co-exact, harmonic exact, harmonic co-exact, and eta components on an annulus geometry.



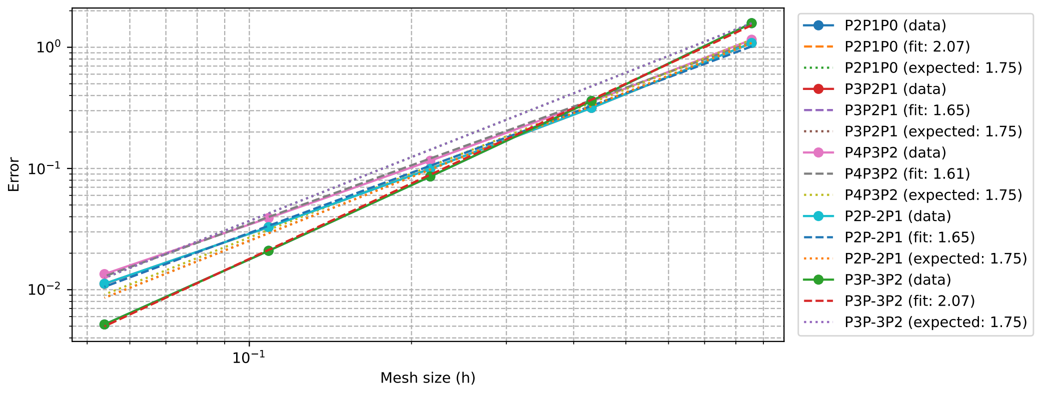


Figure 8.5: Convergence plots for the exact, co-exact, harmonic exact, harmonic co-exact, and eta components on an annulus geometry (continued).

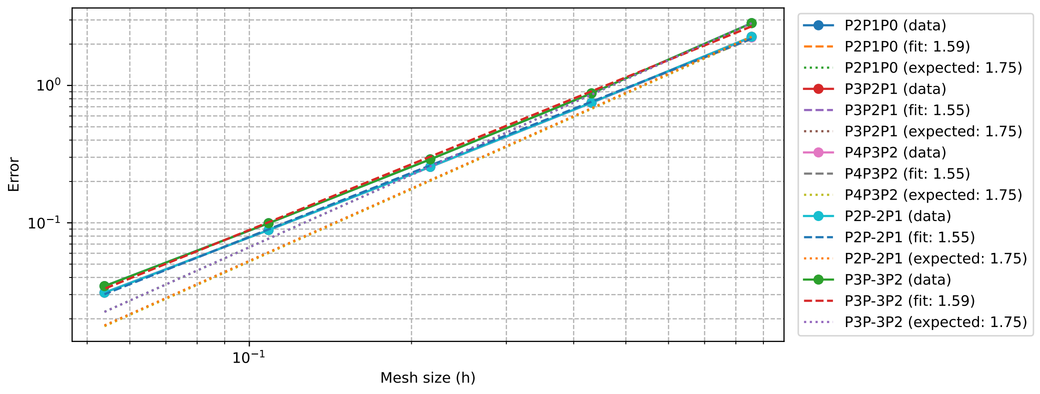
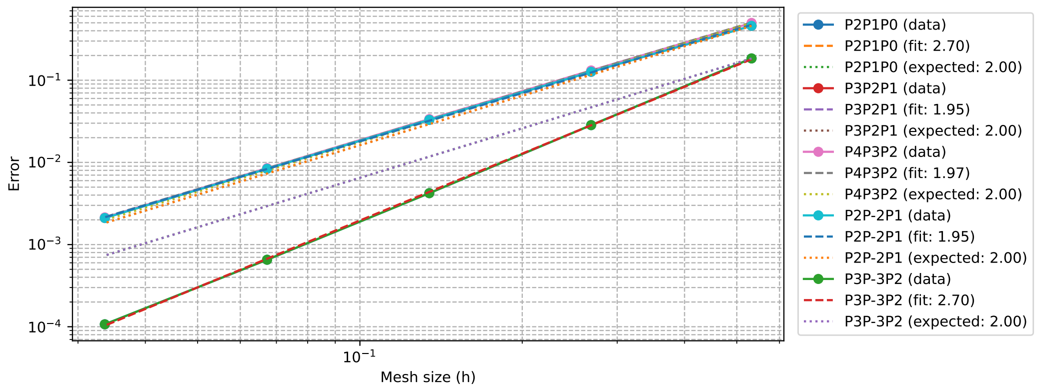


Figure 8.5: Convergence plots for the exact, co-exact, harmonic exact, harmonic co-exact, and eta components on an annulus geometry (continued).



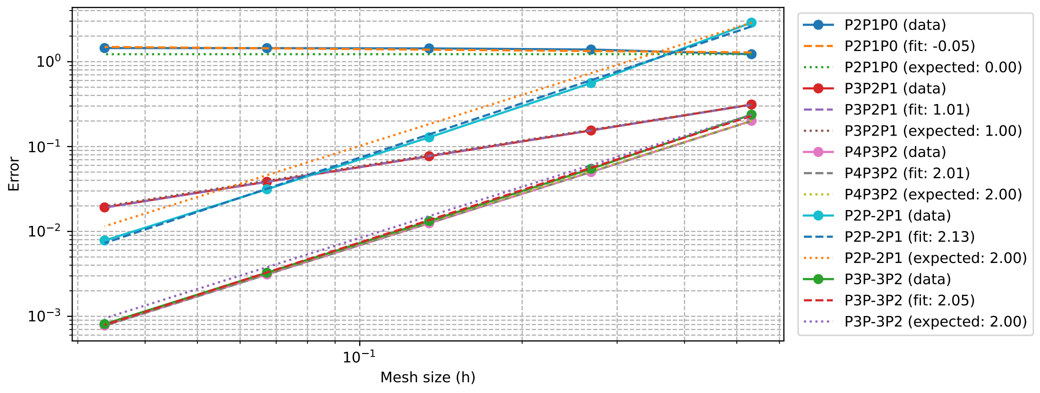
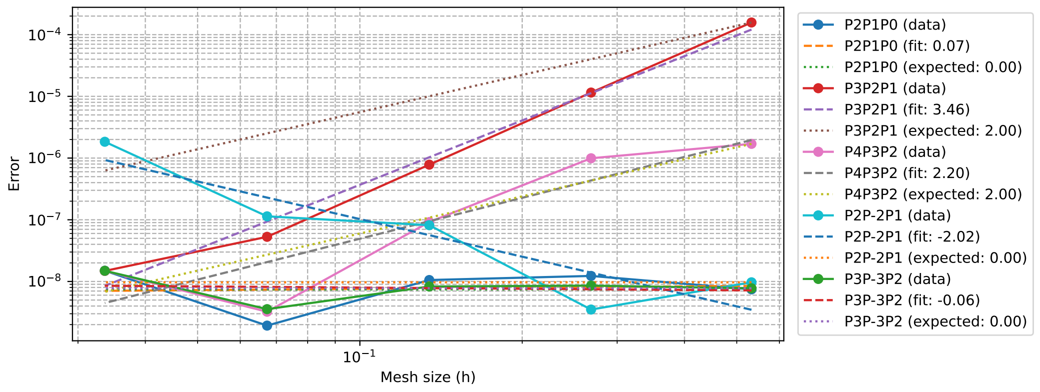


Figure 8.6: Convergence plots for the exact, co-exact, harmonic exact, harmonic co-exact, eta, and total harmonic components on a cylinder geometry.



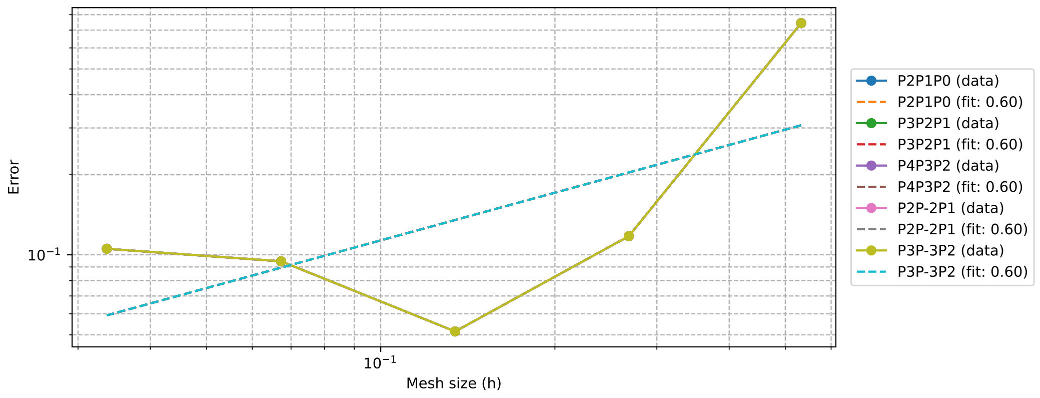
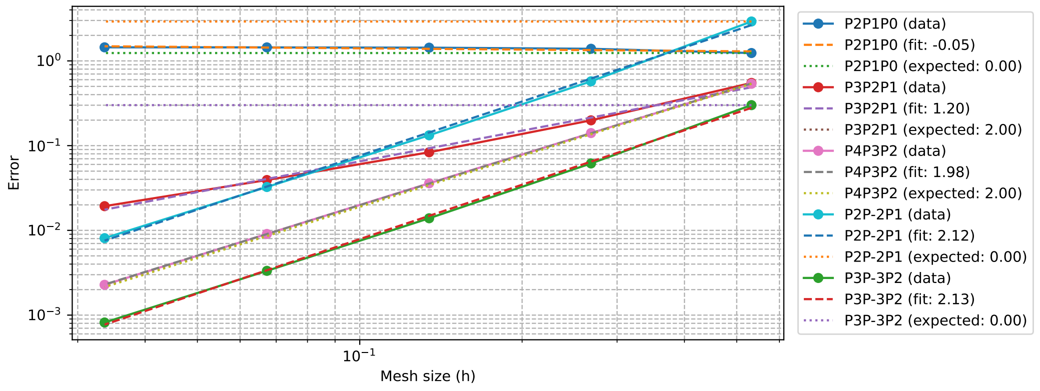


Figure 8.6: Convergence plots for the exact, co-exact, harmonic exact, harmonic co-exact, eta, and total harmonic components on a cylinder geometry (continued).



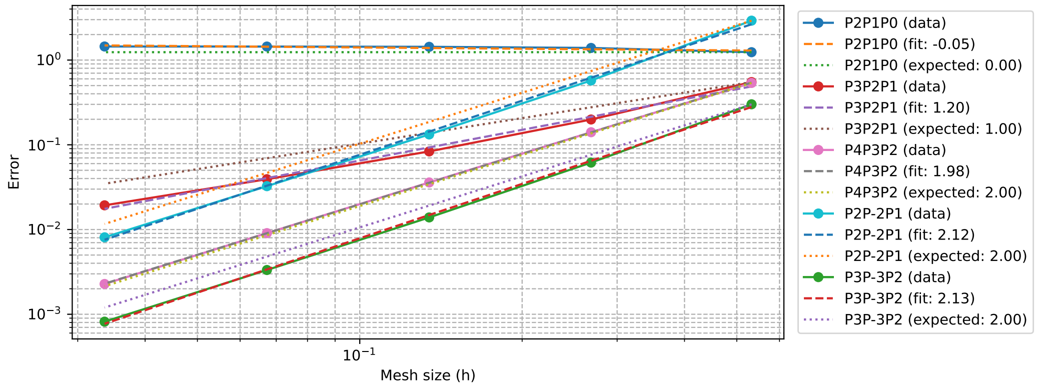
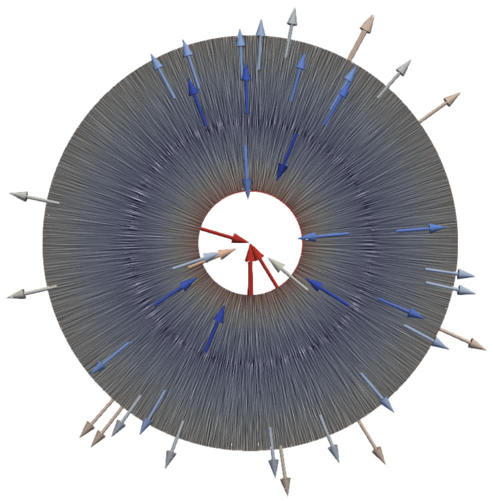


Figure 8.6: Convergence plots for the exact, co-exact, harmonic exact, harmonic co-exact, eta, and total harmonic components on a cylinder geometry (continued).



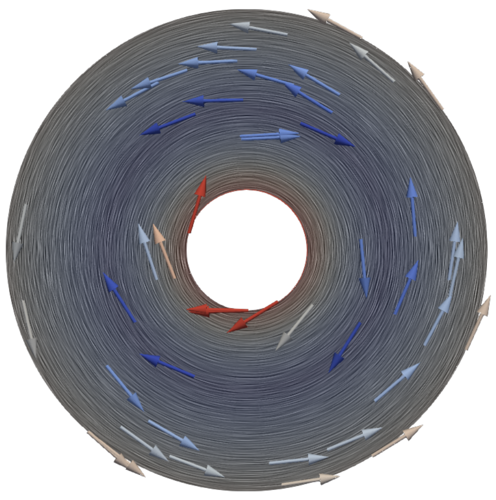
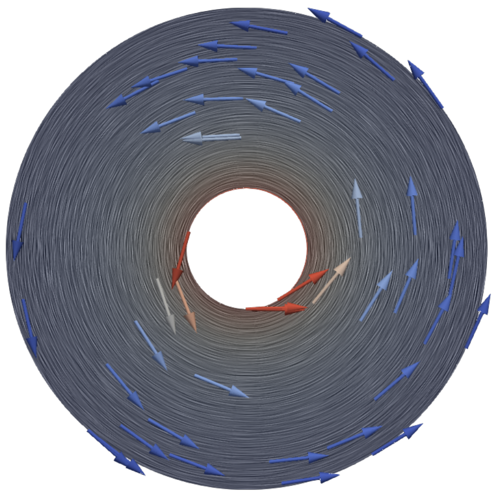


Figure 8.7: True solutions in the annulus scenario, for the exact, co-exact, harmonic exact, harmonic co-exact, and fully harmonic components.



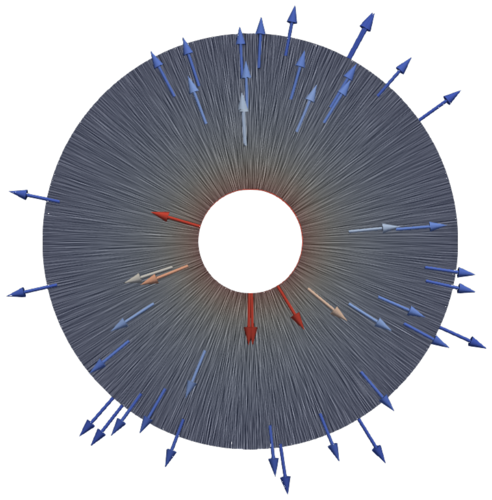


Figure 8.7: True solutions in the annulus scenario, for the exact, co-exact, harmonic exact, harmonic co-exact, and fully harmonic components (continued).

8.1.3 Stokes Flow

8.1.3.1 Problem Setup

Let $d = 2$ or $d = 3$ and $\Omega \subset \mathbb{R}^d$. Suppose we have a forcing function $f\colon [0,\infty) \times \Omega \to \mathbb{R}^3$ and a viscosity parameter $\nu \in \mathbb{R}$. The Stokes equations seek a velocity field $u\colon [0,\infty) \times \Omega \to \mathbb{R}^d$ and a pressure field $p\colon [0,\infty) \times \Omega \to \mathbb{R}$ such that

$$u_t - \nu \Delta u + \nabla p = f \tag{8.34}$$

and

$$\nabla \cdot u = 0. \tag{8.35}$$

We are primarily interested in no-slip BCs and other proscribed time-varying velocities:

$$u|_{\partial\Omega} = 0 \tag{8.36}$$

or

$$u|_{\partial\Omega} = w \tag{8.37}$$

for some $w\colon [0,\infty) \times \partial\Omega \to \mathbb{R}^d$. We additionally enforce that the pressure is mean zero, i.e., $\int_\Omega p = 0$ for all time t.

We are primarily interested in the contrast between cases where $\nu \approx O(1)$ and $\nu \approx O(h)$ where h is the characteristic edge length of the mesh. The former resembles the case of a fluid like water while the latter resembles the case of a superfluid.

8.1.3.2 Operators and Time Stepping

The standard weak formulation for stationary stokes is to find a u and p such that for all $v\colon \Omega \to \mathbb{R}^3$ and $q\colon \Omega \to \mathbb{R}$:

$$F(u,p,q,v,f) = \int_\Omega \nabla u_t \cdot \nabla v - \int_\Omega p\nabla \cdot v + \int_\Omega q\nabla \cdot u - \int_\Omega fv \tag{8.38}$$

We considered implicit Euler and Crank Nicholson time stepping. For a time step size dt and a $\theta \in [0,1]$, we solve for u_{n+1}, p_{n+1} at time t_{n+1} in terms of u_n and p_n at time t_n via finding u_{n+1} and p_{n+1} such that

$$\int_\Omega v(u_{n+1} - u_n)/dt + \theta F(u_{n+1}, p_{n+1}, v, q, f(t_{n+1})) + (1-\theta)F(u_n, p_n, v, q, f(t_n)) = 0 \tag{8.39}$$

holds for all v, q. Since the resulting discrete system is a saddle point system, we must be careful with the solver. For all purposes in this section, we get away with PETSc' GMRES with LU (via MUMMPS) as a pre-conditioner [146]–[148].

8.1.3.3 Elements

We follow the element choices offered in the experiments in [149], including various pairings of Lagrange elements (Taylor-Hood and Scott-Vogelius), mini elements, and Crouzeix-Raviart elements. The Taylor-Hood (TH) element is simply the use of a k-degree vector Lagrange element for the velocity space and a $k-1$-degree scalar Lagrange element for the pressure space. This requires a more general bit of code for the Lagrange element, which we provide here; this version allows for Lagrange elements on arbitrary Euclidean-like spaces and for discontinuous spaces:

```python
def lagrangeDofs(dim: int, degree: int, space = P.R, dg: bool = False):
    PSpace = PImpl(dim, degree, space)
    dofs = []
    for dimp in range(dim + 1):
        for point in equispacedBarycentric(degree, boundary=False):
            for vector in stdEuclideanBasis(space):
                @pointwiseEval(barycentricCoordinate=point, storeAt=dimp if not dg else dim)
                def dof(u: field(dimp, space)): return u(0).dot(vector)
                dofs.append(dof)
    return Dofs(PSpace, dofs)
```

Source Code Listing 8.12: Tensor Lagrange Elements

Scott-Vogelius (SV) can be characterized as a Taylor-Hood element where the pressure space is discontinuous, but this characterization is facile due to the circumstances under which the element works. In particular, the SV element requires that there are no singular vertices (vertices incident to edges that all lie on two lines (or three planes)) and so the SV element is sometimes phrased as TH with discontinuous pressure on a barycentrically refined or macro triangle/tetrahedral meshes [150]–[153]. In the future, we will model with these explicit macro elements, but for now we model this by just refining the mesh barycentrically. A bubble element is a Lagrange element where we include extra basis functions that are zero on the boundary of the geometry. The mini element on a d-dimensional space uses linear Lagrange for the pressure and the sum of linear Lagrange plus a d-degree bubble element for the velocity. We can define a mini space similarly:

```
def mini(dim: int, degree: int, bdegree: int, space = P.R, dg: bool = False):
    assert bdegree > degree and bdegree >= dim + 1
    PSpace, BSpace = PImpl(dim, degree, space), PImpl(dim, bdegree, space)
    idofs = []
    ndofs = []
    dofs = []
    for dimp in range(dim + 1):
        for point in equispacedBarycentric(bdegree, boundary=False):
            for vector in stdEuclideanBasis(space):
                @pointwiseEval(barycentricCoordinate=point, storeAt=dimp if not dg else dim)
                def dof(u: field(dimp, space)): return u(0).dot(vector)
                if dim == dimp:
                    idofs.append(dof)
                else:
                    ndofs.append(dof)
    PSpace =  PImpl(dim, degree, space) + NullSpace(dim, bdegree, space, nullDofs=ndofs)
    return Dofs(PSpace, lagrangeDofs(dim, degree, space).dofs + idofs)
```

Source Code Listing 8.13: Mini Elements

Finally, we do not bother with the CR element as it is very similar to the Lagrange element.

8.1.3.4 Validation and Trade-Offs

We set up two analytical problems to test our solver, which is sufficient to reveal a trade-off. In particular, we follow [149] to show that cheaper elements (fewer DOFs) tend to be non-robust to low viscosities, sometimes inducing locking behaviors similar to those found in elasticity simulation.

In two dimensions, we consider analytical velocities and pressures given by

$$u(x,y,t) = (\cos(y), \sin(x))(1+t) \tag{8.40}$$

and

$$p(x,y,t) = \sin(x+y) - (2\sin(1) - \sin(2)). \tag{8.41}$$

In three dimensions, we consider

$$u(x,y,z,t) = (\cos(y), \sin(z), cos(x))(1+t) \tag{8.42}$$

and

$$p(x,y,z,t) = \sin(x+y+z) - 8(1+2\cos(1)\sin(\frac{1}{2})^4. \tag{8.43}$$

We use these to construct a forcing function according to the Stokes equations and solve until time 0.01 in ten time steps on the unit square or unit cube. We enforce Dirichlet conditions on the velocity while for the pressure we require that $\int p = 0$. We do this for $\nu = 1$ and $\nu = 10^{-6}$. We use the Taylor-Hood elements and the Scott-Vogelius elements at order $k = 2$. In 2D, we use Crank-Nicolson time stepping with $\theta = 0.5$ whereas in

3D we used Euler. Convergence results for velocities and pressures in 2D are plotted in fig. 8.10. We immediately observe that the convergence in velocity of the Taylor-Hood elements is massively impacted by lowering the viscosity while Scott-Vogelius is not impacted. Interestingly, we observe that the pressure convergence is not impacted. As it turns out in the analysis of [149], as viscosity gets smaller, the error in velocity becomes dominated by the error in the gradient of pressure if an element is not discretely divergence free in a specific sense. This theory is borne out in the velocity and pressure plots in fig. 8.10, noting that convergence of a gradient is typically an order lower than convergence of the function. In this example, we do not bear this out visually (velocities at the start and final times are plotted in fig. 8.11), but it is perhaps worth noting that there is a visual price in the pressure for good convergence in velocity: check-boarding patterns (See fig. 8.12).

Though [149] does not show results for the Stokes equation in 3D, we show that similar results hold in 3D via fig. 8.13. We use $k = 3$ order elements.

Finally, to show that this difference can produce visually striking results, we set up a simulation on the Bob mesh. We start from zero initial conditions. We add inflow at the top of a head that looks like

$$u(x, y, z, t) = (0, 0, -1)(1 + 10t)^2 \tag{8.44}$$

and an outflow that looks like

$$u(x, y, z, t) = (0, 1, 0)(1 + 10t)^2. \tag{8.45}$$

We assume a force due to gravity. We compute 200 time steps starting at $t = 0$ and ending at $t = 0.2$. We compare the mini element to the Scott-Vogelius element of order $k = 3$. We note that the former uses 12 velocity DOFs per element while the latter uses 60. We again contrast the behavior for $\nu = 1$ vs $\nu = 10^{-6}$.

The results are seen in fig. 8.14. When $\nu = 1$, both simulations show reasonable results.

We observe that for Scott-Vogelius, the simulation changes dramatically with ν, which is expected. When ν is small, the fluid responds slowly to nearby changes in velocity so the inflow and outflows are slow to spread to the inner regions of the geometry. For Scott-Vogelius, we can plainly see that away from the inflow/outflow, the fluid is mainly responding to gravity. However, for the mini element, changing ν has little qualitative impact. We can see this in a more quantitative manner in fig. 8.13 by showing how the velocity norms change over time for $\nu = 1$ vs. $\nu = 10^{-6}$. In fig. 8.15, we see that the mini element's velocity norm grows dramatically irrespective of the velocity whereas the Scott-Vogelius element responds at a slower pace.

8.2 Claim: Finite elements are specified concisely due to automation

See table 8.2. In particular, we show that we are able to dramatically improve on the SOTA for specifying FEs within systems capable of assembling FE matrices. We do not compare against systems like the symbolic finite elements library as not all of their specifications (e.g., Morley) can be used on multiple domains [79]. Moreover, our improvements in lines of code come from two of our automations. First, our efforts to automatically deduce transformations by simplifying formulas and symbolically inverting matrices represents non-trivial portions of code counts for many elements. Not only does this portion represent a non-trivial number of lines, but these lines are hard to produce, requiring manual mathematical reasoning. Second, our efforts to build a high-level language for IPTEMs, open in particular to the automation of topological enumeration, also dramatically reduce the lines of code for all elements. This is best shown for Lagrange and FEEC elements: though the transformation is trivial, we still make the specification smaller by relying on topological enumeration and our high-level language for tensor expressions.

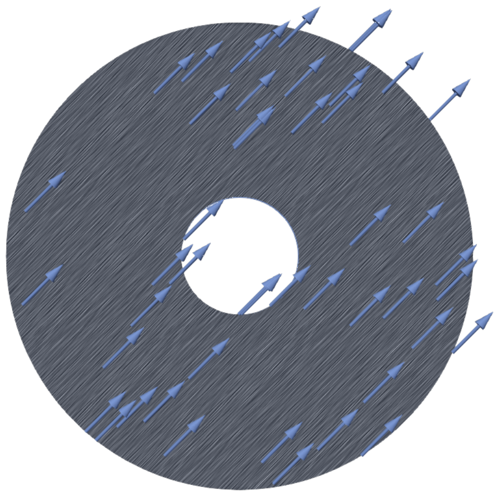
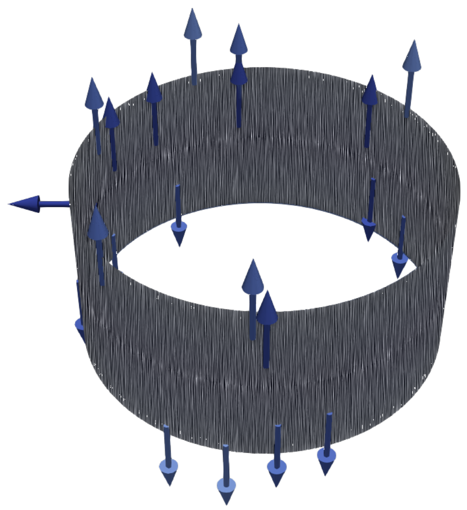


Figure 8.7: True solutions in the annulus scenario, for the exact, co-exact, harmonic exact, harmonic co-exact, and fully harmonic components (continued).

Element	Sybil SLOC	FINAT/FIAT SLOC (Transform)
Morley	∼10	84 (31)
Hermite	∼17	88 (23)
Argyris	∼31	221 (117)
Bell	∼44	138 (63)
(RT + Nédéc (first kind) + DG Volume Lagrange) + (BDM + Nédéc (second kind))	∼52	756 (0)
Lagrange	∼10	80 (0)
Mini	∼17	28 (0)

Table 8.2: SLOC to specify all the elements used in our experiments and available in both FINAT, currently the most comprehensive practical system for using different FEs. Note that for our system, RT, Nédéc (first kind), and DG Volume Lagrange all share an implementation and so do BDM + Nédéc (second kind), so we group these together.



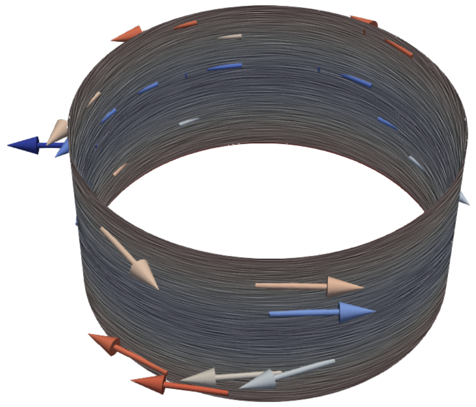
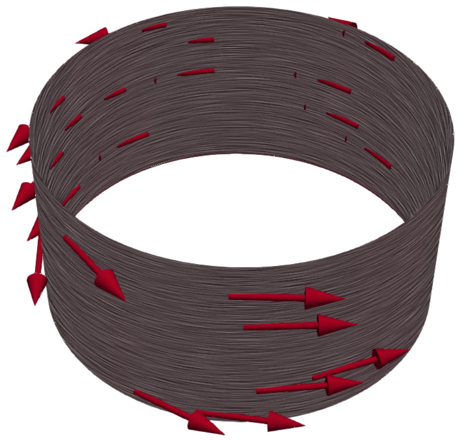


Figure 8.8: True solutions in the cylinder scenario, for the exact, co-exact, harmonic exact, harmonic co-exact, and fully harmonic components.



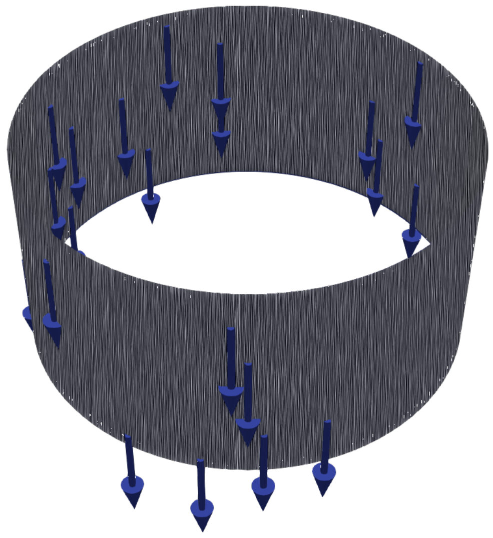


Figure 8.8: True solutions in the cylinder scenario, for the exact, co-exact, harmonic exact, harmonic co-exact, and fully harmonic components (continued).

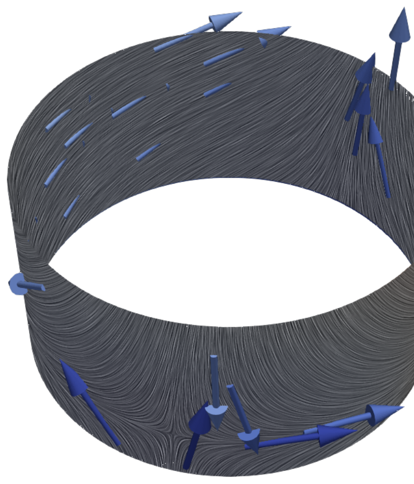


Figure 8.8: True solutions in the cylinder scenario, for the exact, co-exact, harmonic exact, harmonic co-exact, and fully harmonic components (continued).

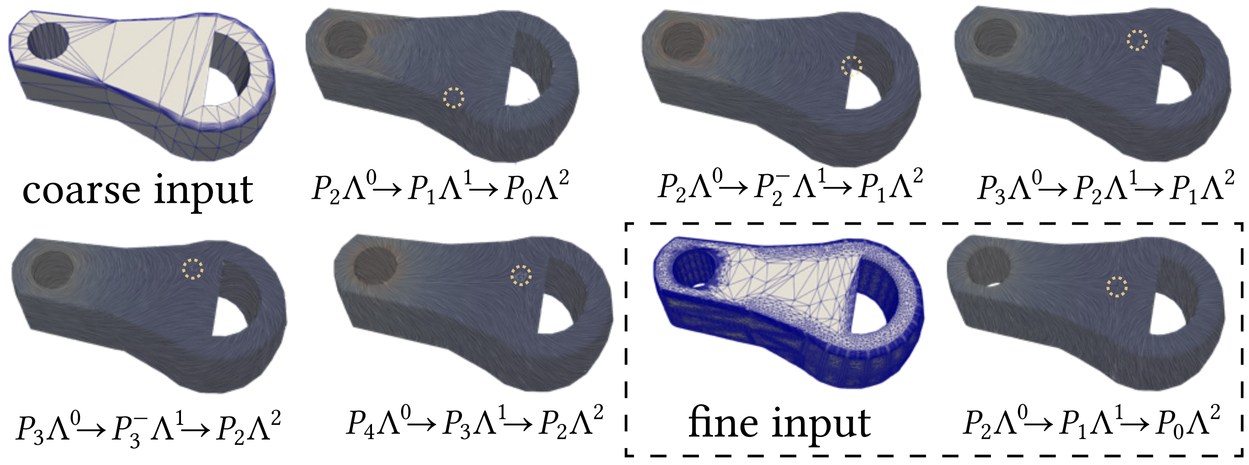
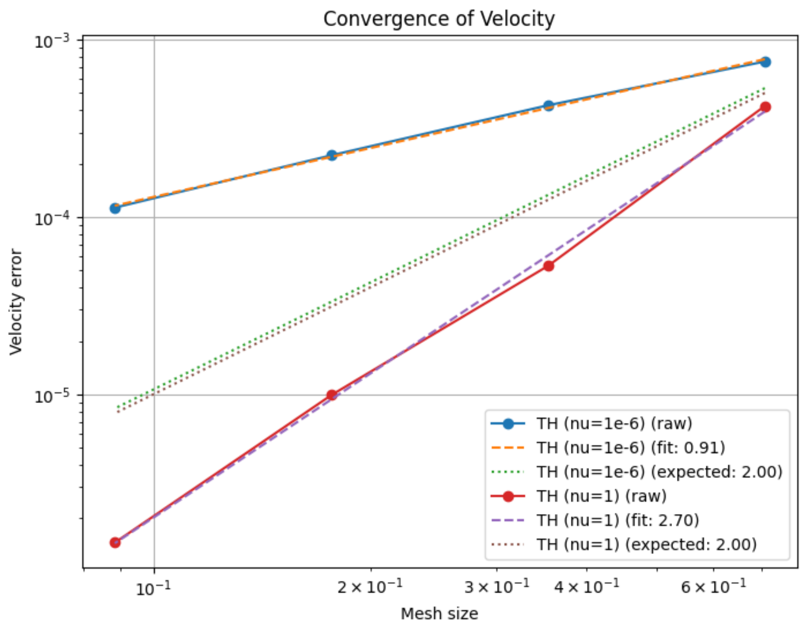

coarse input
$P_2\Lambda^0 \to P_1\Lambda^1 \to P_0\Lambda^2$
$P_2\Lambda^0 \to P_2^-\Lambda^1 \to P_1\Lambda^2$
$P_3\Lambda^0 \to P_2\Lambda^1 \to P_1\Lambda^2$
$P_3\Lambda^0 \to P_3^-\Lambda^1 \to P_2\Lambda^2$
$P_4\Lambda^0 \to P_3\Lambda^1 \to P_2\Lambda^2$
fine input
$P_2\Lambda^0 \to P_1\Lambda^1 \to P_0\Lambda^2$

Figure 8.9: Zero eigenvectors of the Hodge Laplacian on a low-resolution mesh with a variety of elements and a comparison to a lower order element on a highly refined mesh (282 vertices vs. 29,000 vertices).



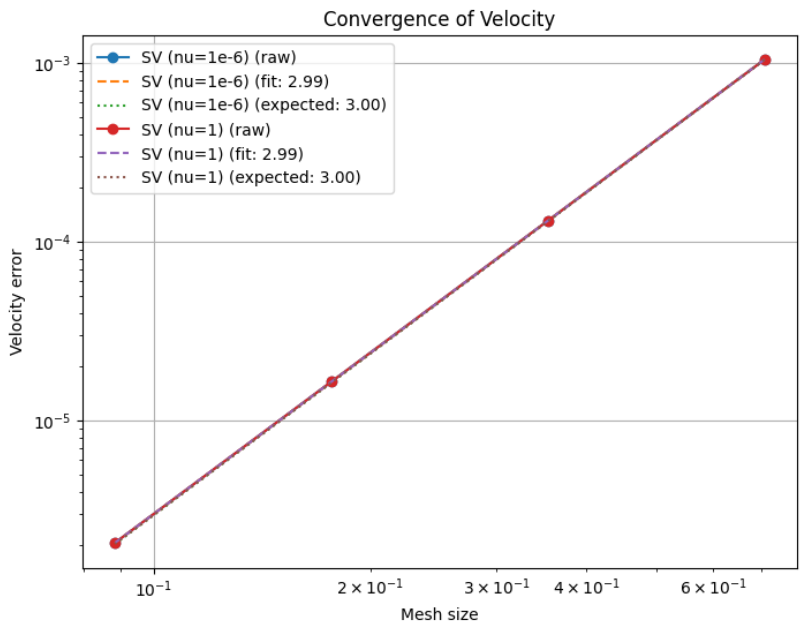
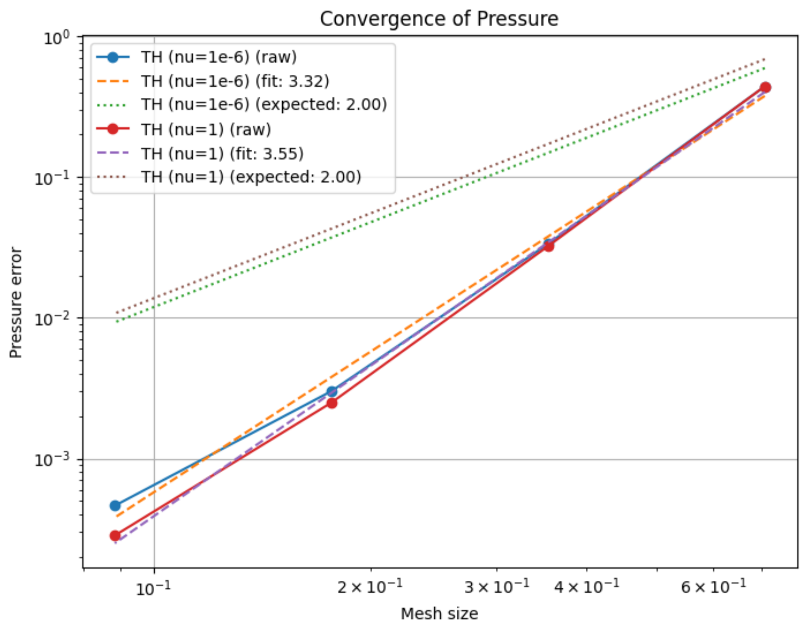


Figure 8.10: Velocity and pressure convergence for a 2D analytical Stokes problem using two different elements and two different viscosities.



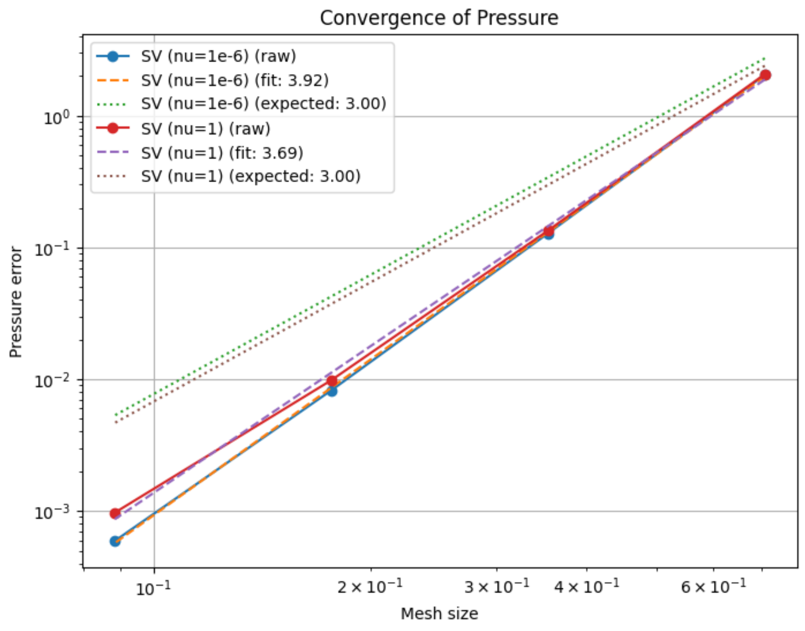


Figure 8.10: Velocity and pressure convergence for a 2D analytical Stokes problem using two different elements and two different viscosities (continued).

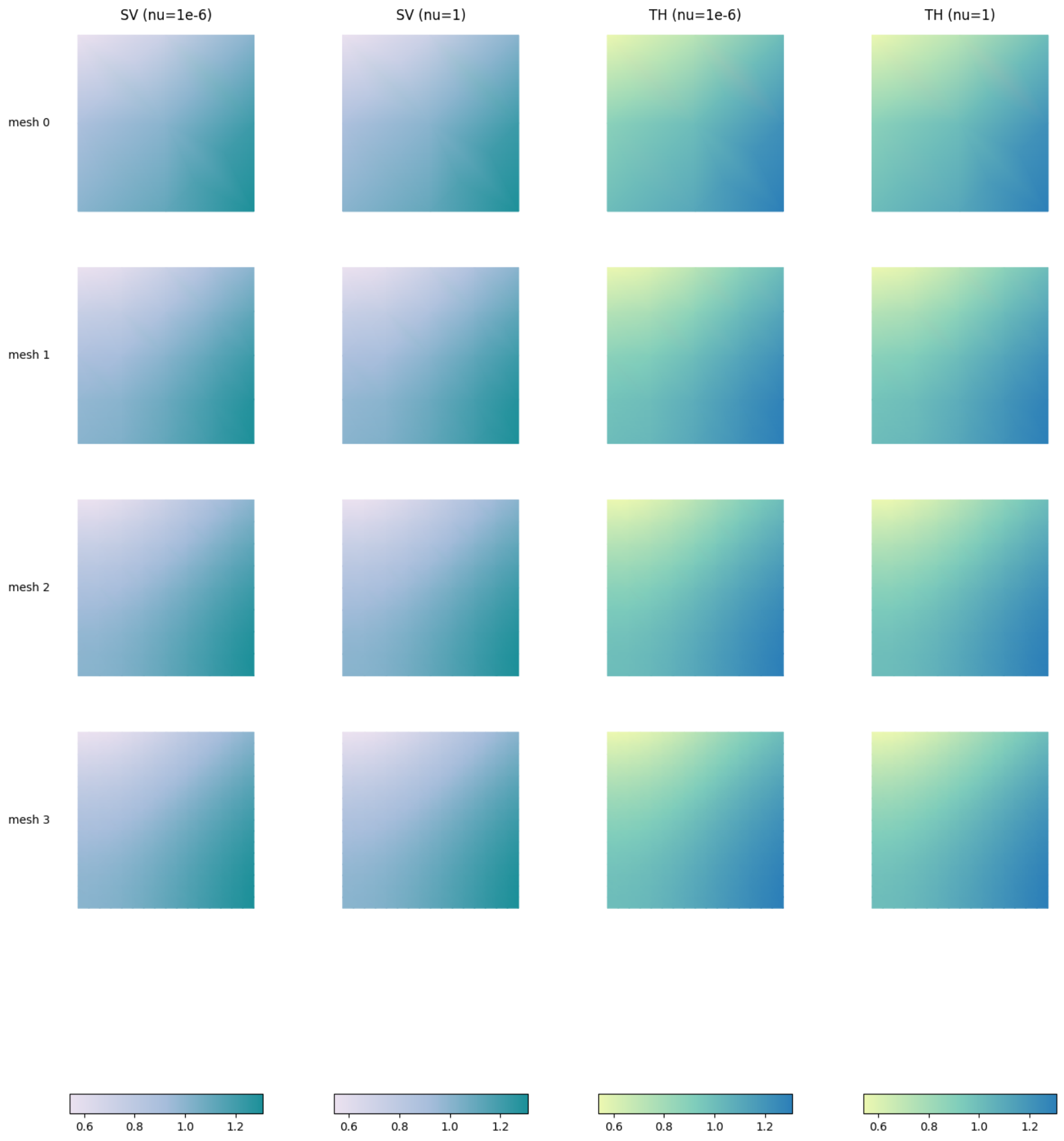


Figure 8.11: Computed pressures on tested meshes with different elements and viscosities at start and end times.

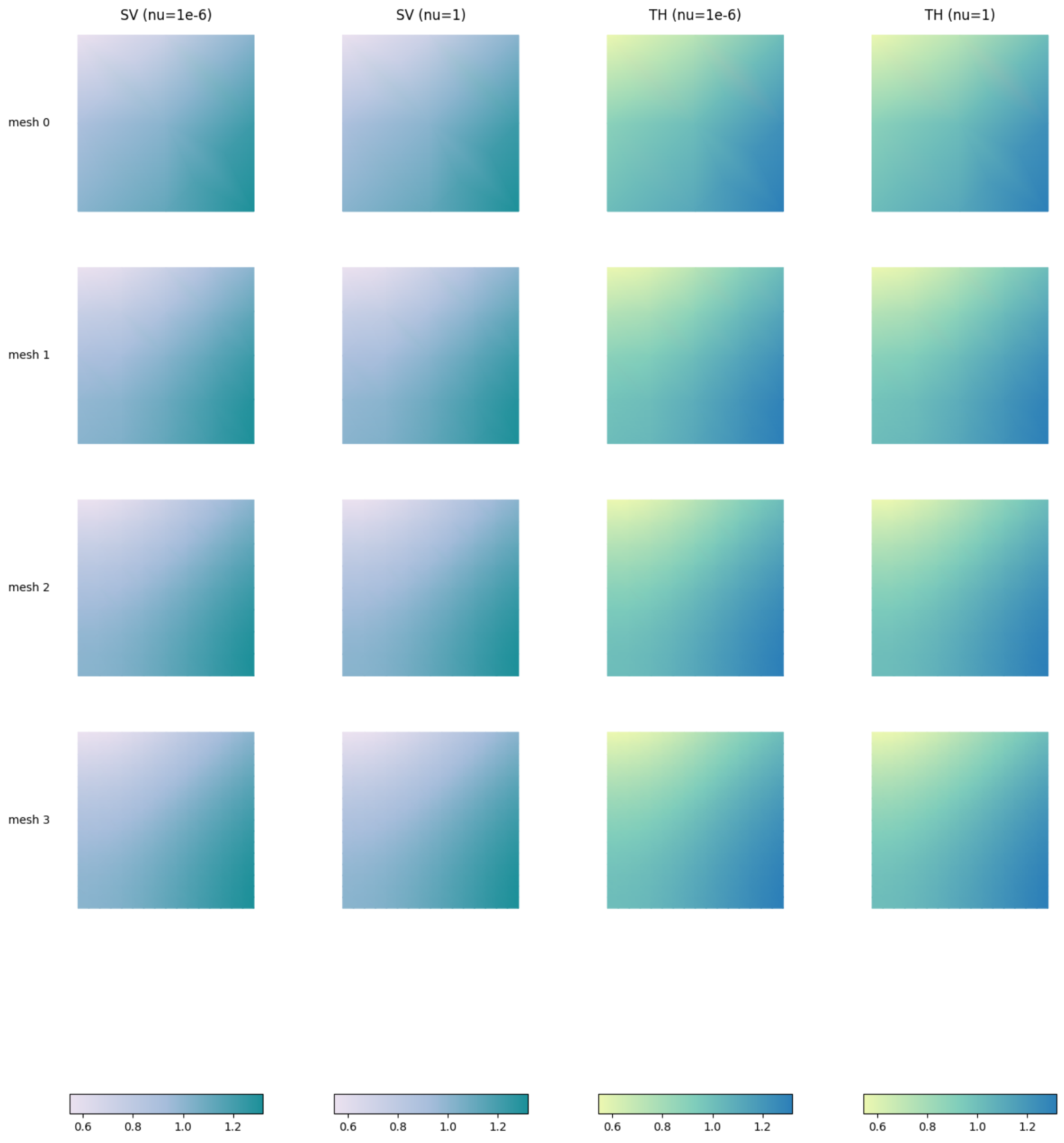


Figure 8.11: Computed velocity on tested meshes with different elements and viscosities at start and end times (continued).

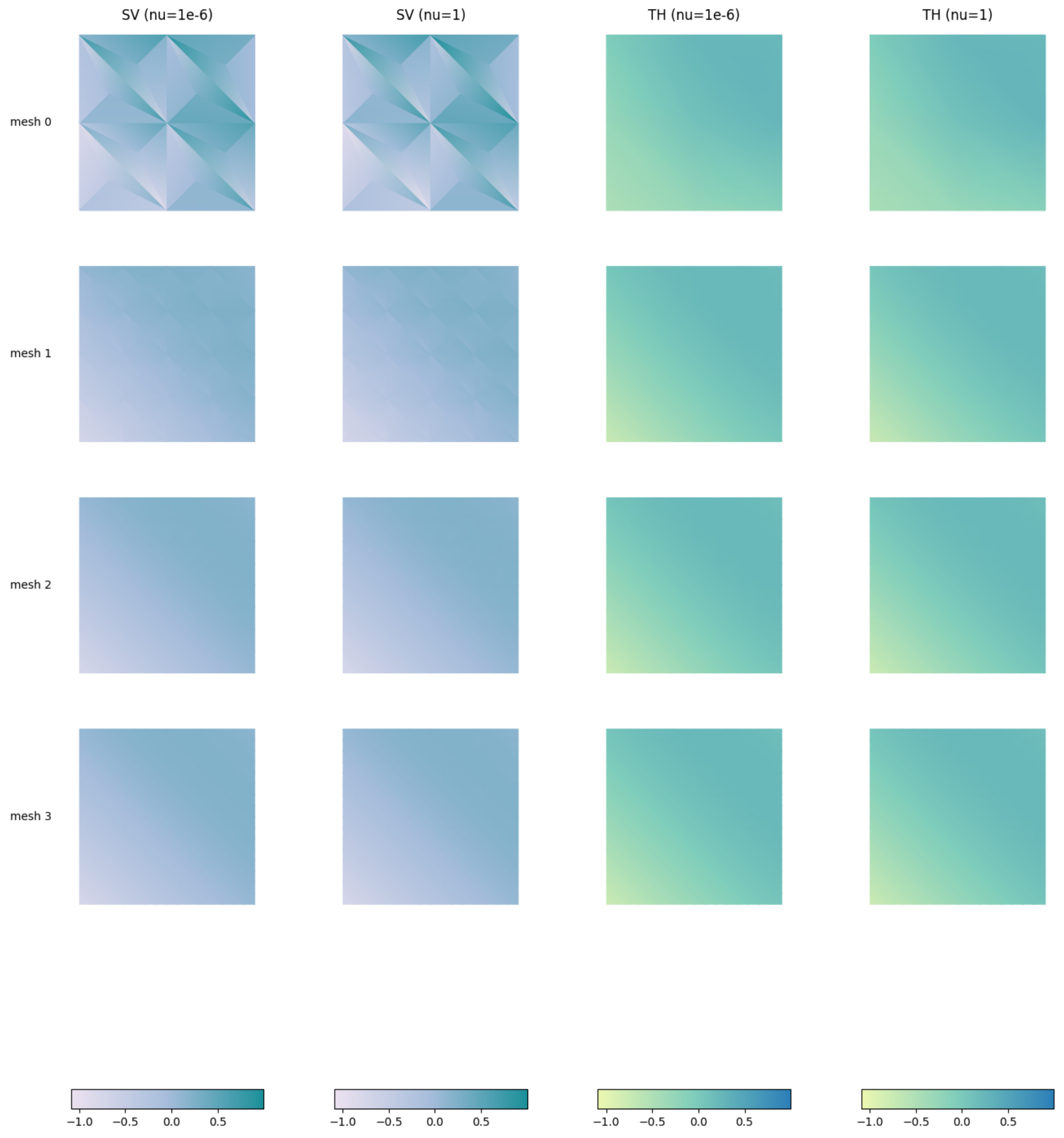


Figure 8.12: Computed pressure on tested meshes with different elements and viscosities at start and end times.

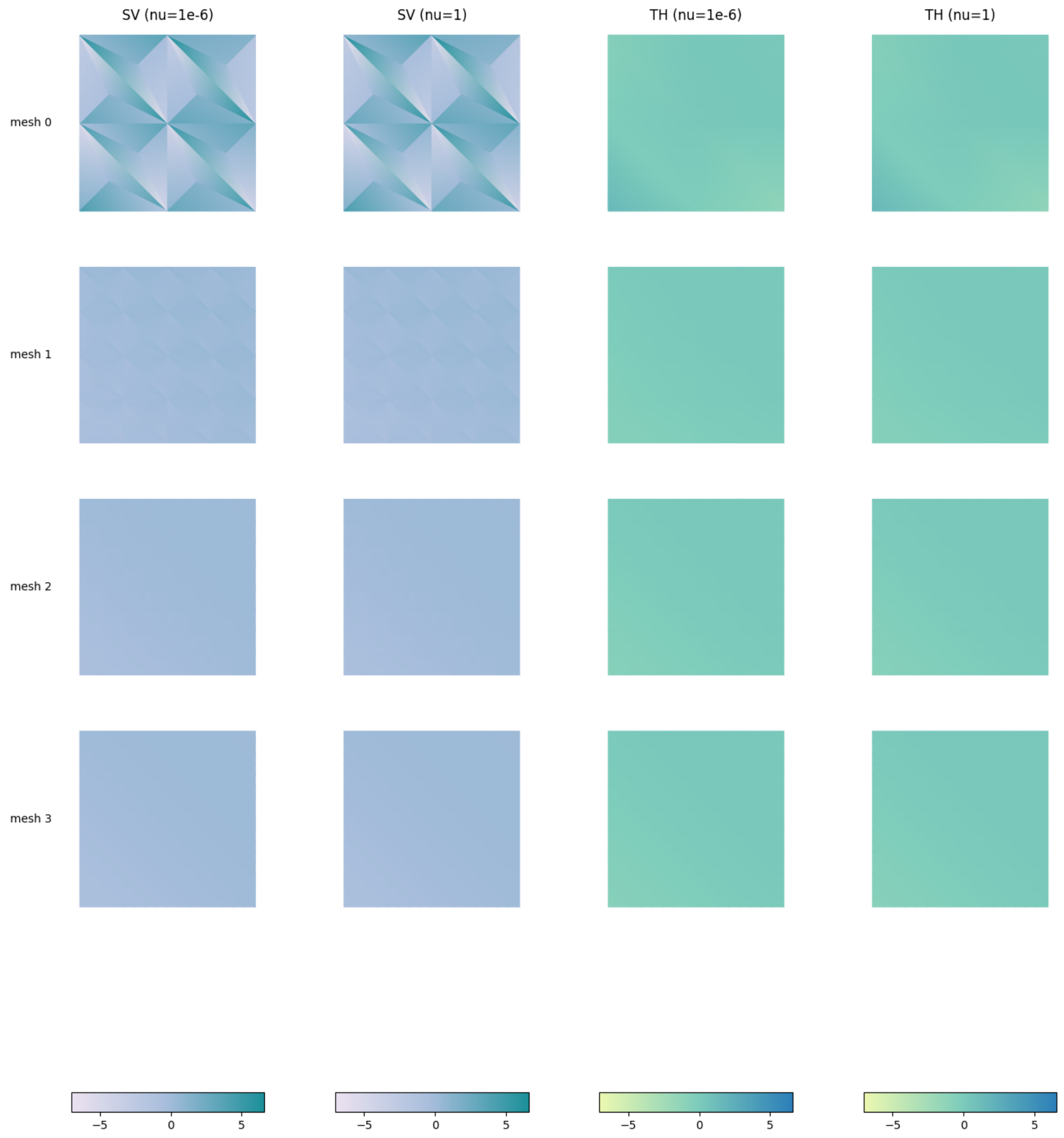


Figure 8.12: Computed pressure on tested meshes with different elements and viscosities at start and end times (continued).

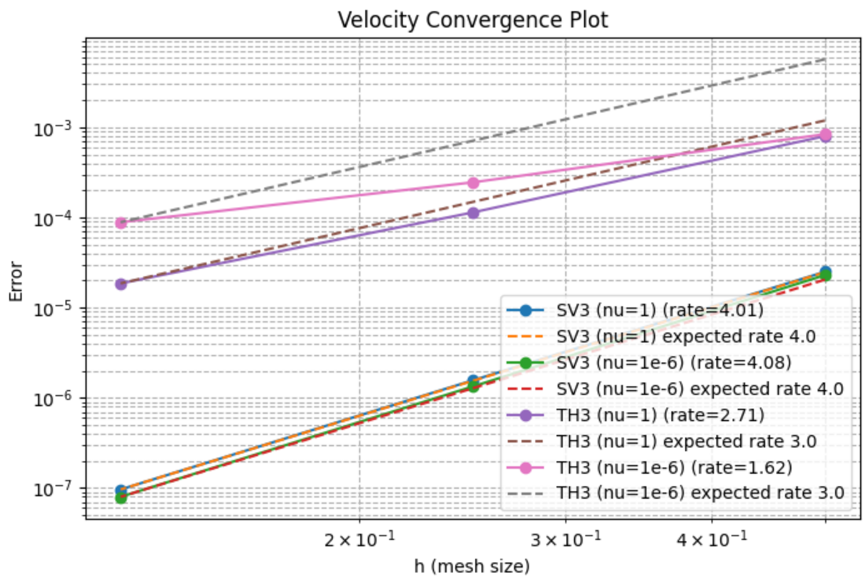


Figure 8.13: Velocity and pressure convergence for a 3D analytical Stokes problem using two different elements and two different viscosities.

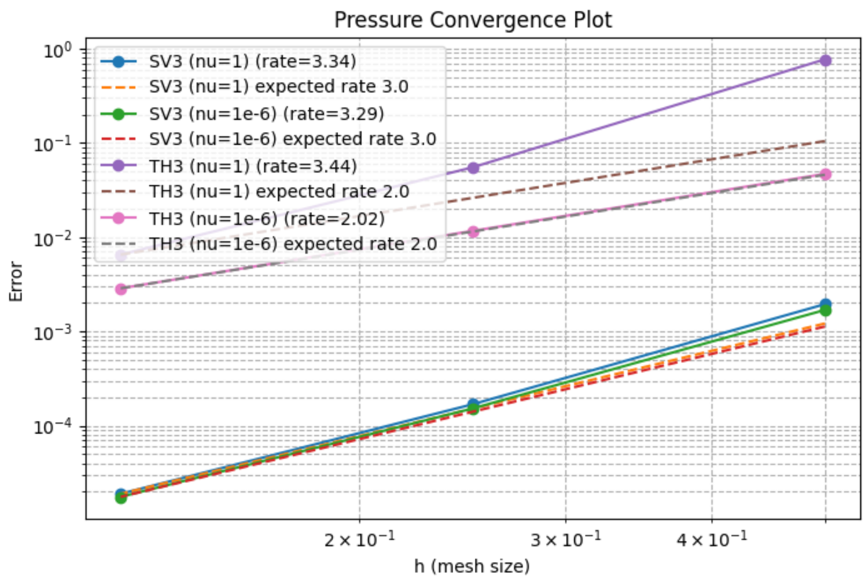


Figure 8.13: Velocity and pressure convergence for a 3D analytical Stokes problem using two different elements and two different viscosities (continued).

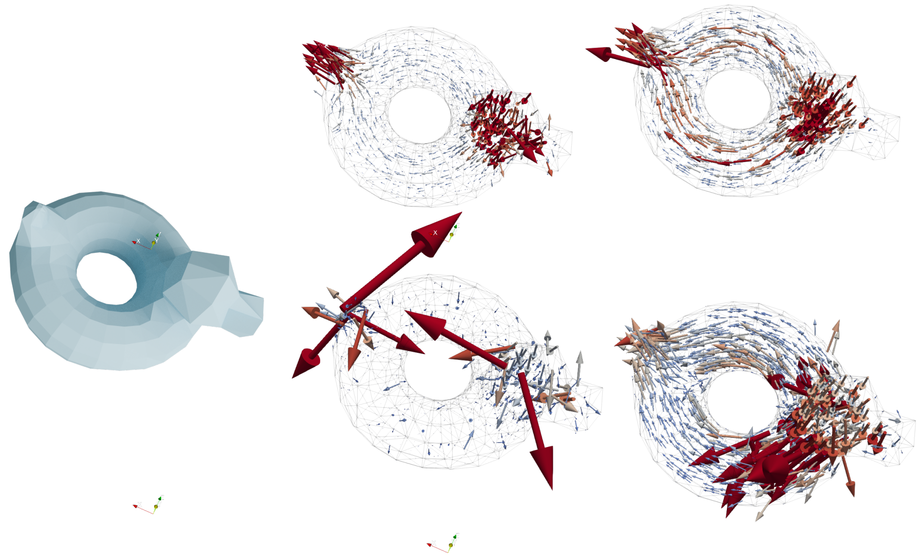


Figure 8.14: Bob Stokes simulation at the 50th time step with different elements and viscosities. The mesh is shown on the left. Columns are elements (SV vs. Mini) and rows are viscosities ($\nu = 1$ vs $\nu = 10^{-6}$).

8.2.1 Claim: The automation produces optimal implementations of the finite element transformation theory.

In [34], a key measure of the cost of using a more complex element is the number of non-zeros in the transformation. Our automations and saved lines of code would not be as useful if the transformations were considerably more expensive. While a direct comparison of runtime is not reasonable, a suitable proxy is to symbolically count the number of non-zeros. We do this in table 8.3, where we find that we have the number of non-zeros in matrices computed by hand in the literature. Additionally, this shows that our inversion procedure does not default to a matrix inverse: ElementForge enables the inverse to be computed more like a formula and with the optimal sparsity pattern visible at compile time.

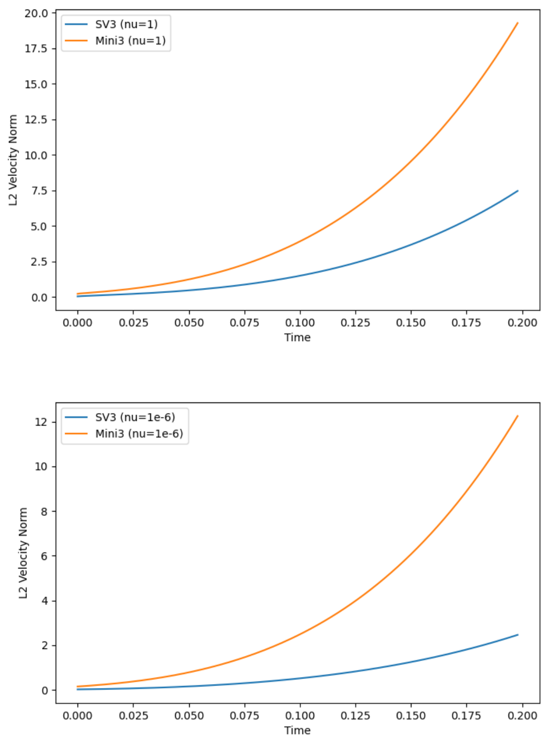


Figure 8.15: Velocity norm over time for both elements, first with $\nu = 1$ then with $\nu = 10^{-6}$

Element	Transformation NNZ
Morley	12
Hermite	16
Argyris	81
Bell	42
(RT + Nédéc (first kind) + DG Volume Lagrange) + (BDM + Nédéc (second kind))	$O(N)$
Lagrange	$O(N)$
Mini	$O(N)$

Table 8.3: Transformation DOFs

Element	Additional LOC
Morley	4
Hermite	11
Argyris	11
Bell	13

Table 8.4: Extra LOCs for biharmonic elements against affine Lagrange

Element	Additional LOC
Mini	10
CR	1

Table 8.5: Extra LOCs for Stokes elements against general Lagrange

8.3 Claim: Finite element specifications and usage are malleable

To evaluate malleability, we ask: how many extra lines of code are required to change elements? First, to literally add the element in the two simulations where the code changes, we find that the number of lines of code is modest (table 8.4 and table 8.5). Similarly, in the case of the FEEC elements, we have a much more succinct implementation, especially for trimmed elements where we more directly translate the math than compared to other systems [8]. Second, we ask how much do programs change when we change elements? The answer is that they do not unless an element requires a different formulation (e.g., the mixed Lagrange method in the biharmonic equation). Compared to other systems, our advantage here lies in our construction of the front end; no other systems that we know of support interpolation into elements such as Hermite, Morley, or Argyris[2], but this naturally emerges in our front end. Similarly, ElementForge supports direct enforcement of Dirichlet BCs when possible on these spaces, whereas that is typically a special case in other systems [63].

[2]See https://www.firedrakeproject.org/interpolation.html

Appendix A

Alternative Analysis of Finite Element Transformations

We now prove a variant of theorem 6.2.1.1 that does not require a symbolic matrix inversion and that illuminates the structure of the Vandermonde matrices, to justify our symbolic inversion algorithm. We need two definitions to start.

Definition A.0.1 (Dual Basis Cover)**.** For each i, fix a basis $\{w_d\}$ for the tangent space of K_i and a basis $\{v_k\}$ for the dual space of $V_i^\star$. The former implies a basis $\{w_{d,m}\}$ for $\otimes^m T_y$ for each $m \geq 0$. Given a point $y = T_i(x)$, an element $w_{d,m} \in \otimes^m T_y$, and an element v_k for $V_i^\star$, then define a functional on $C^m(K_i, V_i)$

$$\gamma^{K_i}_{x,m,d,k}(p) = v_k((D^m p)(T_i(y))(w_{d,m})). \tag{A.1}$$

A dual basis cover for our collection of elements E_i is a collection of tuples $L = \{(x_q, m_q, d_q, k_q)\}_q$ such that for all i,

1. $\{\gamma_l^{K_i}\}_{l \in L}$, span the space $P(K_i, V_i)^\star$,

2. and for every x_q and m_q, $\{(x_q, m_q, d, k) \colon w_{d,m}, v_k\} \subset L$.

The second condition mandates that the at each point x_q, we include enough γ to model every element of the tensor corresponding to any relevant derivatives, D^{m_q}.

For a collection of finite elements $\{E_i\}$ relative to E that satisfy the conditions of theorem 6.2.1.1, a natural choice of dual basis cover exists: use the quadrature points at each relevant derivative against a Euclidean basis and pair them against elements of a Euclidean basis or derived basis thereof. To make a claim about the sparsity of matrices for these problems, it helps to have a notion of support for functionals:

Definition A.0.2 (Support of a Dual Basis Function)**.** Given a functional f of a function space $P(K, V)$, the support is the minimal non-empty set $S \subset K$ such that $S \subset \{x \in K \colon g(x) \neq 0\}$ for all $g \in P(K, V)$ where $f(g) \neq 0$. We write $S = \mathrm{Support}(f)$.

We now state our main theorem:

Theorem A.0.2.1. *Assume the conditions of theorem 6.2.1.1. Given a dual basis cover L, then for any i, there are three matrices:*

1. *E of size $|\Sigma| \times |L|$, representing the dual basis of E_i in terms of the dual basis cover in the space $P(K_i, V_i)^\star$,*

2. *F of size $|L| \times |L|$, representing the action of $T_\star^{-1}$ on the dual basis cover of E in the space $C^\infty(K_i, V_i)$,*

3. *and G of size $|L| \times |\Sigma|$, representing the dual basis cover in terms of the dual basis of E in the space $P(K, V)^\star$*

such that $P^{K_i} = (EFG)^T$ i.e., the representation of the push-forward of the DOFs by T_i^{-1} is decomposed into the above three stages. Furthermore, F can be written as a block diagonal matrix and E can be written as a sparse block triangular matrix where the blocks are determined by the supports of the dual basis functions.

This theorem gives us an alternative algorithm that involves no direct symbolic inverse and it also gives us a way to analyze the structure of P^{K_i}; if G has a sparse triangular structure, then we expect P^{K_i} to have one as well. Similarly, the sparsity of G will highly influence the sparsity of P^{K_i}, which then controls the overhead of using a given element.

Proof of theorem A.0.2.1. By definition, the dual basis cover L automatically supplies a matrix E that satisfies

$$\phi_b^{K_i}(f) = \sum_{l \in L} E_{bl} \gamma_l^{K_i}. \tag{A.2}$$

We observe that E_{bl} can be organized as a sparse triangular matrix if we topologically sort the set L using the dictionary order with points partially ordered based on membership in support$(\phi_b^{K_i})$.

Next, we observe that $(T_\star^{-1})$ is easily computed on $\gamma_{(x,m,d,v)}^{K_i}(f)$ in terms of all $\gamma_{(x,m,d',v')}^{K}(f)$ via the chain rule for any valid $f \in C(K_i, V_i)$ because a basis cover includes all entries of a given derivative a point. Thus $(T_\star^{-1})$, represented as an operator on the span sets of the basis covers, is a block matrix, which we call F. In the simplest cases, the matrix F will be block diagonal, consisting of tensor products of DT_i and DT_i^{-1}.

Next, we observe that we can use f_b^K to compute γ_j^K in terms of ϕ_b^K, via solving for G such that

$$\gamma_j^K(f) = \sum_j G_{bj} \phi_b^K(f) \tag{A.3}$$

for all $f \in P(K, V)$, which is guaranteed to exist because ϕ_b^K is a basis for $P(K, V)^\star$ and γ_j^K is a spanning set for $P(K, V)^\star$.

Finally, since via theorem 6.2.1.1, P^{K_i} is a representation of the pullback, we observe that we have $P^{K_i} = (EFG)^T$ because EFG represents $(T_i^{-1})_\star$ via the above constructions. □

For the purposes of our original proof, the most important aspect of this proof is the sparsity pattern. In particular, the matrices E and G have sparsity patterns defined based on interactions between the dual basis cover and the original dual bases. For example, dual basis functions can only share a relationship with a dual basis cover function if they have intersecting supports. And vice-versa for G. Thus, the sparsity pattern of E and G is built on the sparsity pattern of the supports of dual basis functions and dual basis cover functions. Since we expect supports to coincide with entities in simplices, we expect the supports to interact in a triangular pattern. Thus, we get a triangular blocked pattern in E and G.

Appendix B

Additional Details on Global Function Spaces

If we group DOFs in the broken space, we can produce FE spaces such as the global linear Lagrange space. To group basis functions, we need an inverse restriction operator, which requires another definition and motivates restrictions on how we group DOFs:

Definition B.0.1 (Group Extension Operator)**.** Given a finite set of bounded domains T_i such that $\cup\overline{T_i}$ is connected and $T_j^\circ \cap T_i^\circ = \emptyset$ and $\overline{T_j} \cap \overline{T_i} \neq \emptyset$ if $i \neq j$. Further, suppose we have a vector space V and a a collection of continuous functions $f_i \colon T_i \to V$. Then $g := |^{-1}\{f_i\}$ is an integrable function[1] such that for all i, $g(x)|_{T_i} = f_i(x)$ for all $x \in T_i^\circ$ and $g(x) = 0$ if $x \notin \cup_i \overline{T_i}$. We call g the **extension** of the set of functions $\{f_i\}$ and $|^{-1}$ the **group extension operator**.

We phrase the group extension operator as $|^{-1}$ to encourage thinking about this as an inverse of the restriction. The idea of a global FE space is to group the basis functions into groups to apply $|^{-1}$ to. To ensure we do so validly in the general case, we must meet the conditions to use $|^{-1}$, which requires another definition:

[1]Even Riemannian integrability restricts us to functions that are only discontinuous on sets of measure zero; technically though for uniqueness in this definition, we want functions in the space $L^1(\Omega, V)$, the space of equivalence classes of integrable Lebesgue measurable functions

Definition B.0.2 (Support of a Dual Basis Function)**.** Given a domain T and a vector space V, the support of a function $f\colon T \to V$ is the non-zero set:

$$\operatorname{Support} f := \{x\colon x \in T, f(x) \neq 0\}$$

Given a function space $Q \subset \mathcal{C}^0(T, V)$ and a dual vector $\phi \in Q^\star$, the support of ϕ is the largest set $T' \subset T$ such that if $f \in Q$ satisfies $\phi(f) \neq 0$ then $T' \cap \operatorname{Support} f$ is an open non-empty subset of T'. We denote this support as $\operatorname{Support}_Q f$

We note that for any point evaluation dual vector, the support will be the point, and the support of dual vectors defined via integration will tend to be the domain of integration.

References

[1] P. G. Ciarlet, *Finite Element Method for Elliptic Problems*. USA: Society for Industrial and Applied Mathematics, 2002, ISBN: 0898715148.

[2] Hughes, *The Finite Element Method* (Dover Civil and Mechanical Engineering). Mineola, NY: Dover Publications, Aug. 2000.

[3] S. C. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*. New York City, NY: Springer New York, 2008, ISBN: 9780387759340. DOI: 10.1007/978-0-387-75934-0.

[4] M. W. Scroggs, *DefElement: An encyclopedia of finite element definitions*, Poster presented at FEniCS 2023, Cagliari, Italy, 2023. DOI: 10.6084/m9.figshare.23294939.v1.

[5] T. Schneider, Y. Hu, X. Gao, J. Dumas, D. Zorin, and D. Panozzo, "A large-scale comparison of tetrahedral and hexahedral elements for solving elliptic pdes with the finite element method," *ACM Trans. Graph.*, vol. 41, no. 3, Mar. 2022, ISSN: 0730-0301. DOI: 10.1145/3508372.

[6] T. Schneider, J. Dumas, X. Gao, M. Botsch, D. Panozzo, and D. Zorin, "Poly-spline finite-element method," *ACM Trans. Graph.*, vol. 38, no. 3, Mar. 2019, ISSN: 0730-0301. DOI: 10.1145/3313797.

[7] D. Ham, M. Homolya, M. Lange, R. Kirby, and L. Mitchell, *Finat/finat: A smarter library of finite elements*, 2025. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.597531.

[8] R. C. Kirby, "Algorithm 839: Fiat, a new paradigm for computing finite element basis functions," *ACM Trans. Math. Softw.*, vol. 30, no. 4, pp. 502–516, Dec. 2004, ISSN: 0098-3500. DOI: 10.1145/1039813.1039820.

[9] R. C. Kirby and L. Mitchell, "Code generation for generally mapped finite elements," *ACM Transactions on Mathematical Software*, vol. 45, no. 4, pp. 1–23, Dec. 2019, ISSN: 1557-7295. DOI: 10.1145/3361745.

[10] Y. Renard and K. Poulios, "Getfem: Automated fe modeling of multiphysics problems based on a generic weak form language," *ACM Transactions on Mathematical Software*, vol. 47, no. 1, pp. 1–31, Dec. 2020, ISSN: 1557-7295. DOI: 10.1145/3412849.

[11] R. Anderson, J. Andrej, A. Barker, *et al.*, "MFEM: A modular finite element methods library," *Computers & Mathematics with Applications*, vol. 81, pp. 42–74, 2021. DOI: 10.1016/j.camwa.2020.06.009.

[12] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Transactions on Mathematical Software*, vol. 40, no. 2, pp. 1–37, Feb. 2014, ISSN: 1557-7295. DOI: 10.1145/2566630.

[13] P. Gangl, K. Sturm, M. Neunteufel, and J. Schöberl, *Fully and semi-automated shape differentiation in ngsolve*, 2020. eprint: arXiv:2004.06783.

[14] A. Logg, K.-A. Mardal, and G. Wells, Eds., *Automated solution of differential equations by the finite element method* (Lecture Notes in Computational Science and Engineering), en. Berlin, Germany: Springer, Aug. 2016.

[15] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the finite element method by composing abstractions," *ACM Trans. Math. Softw.*, vol. 43, no. 3, Dec. 2016, ISSN: 0098-3500. DOI: 10.1145/2998441.

[16] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger, "Dune-fem: A general purpose discretization toolbox for parallel and adaptive scientific computing," in *Advances in DUNE*. Berlin, Germany: Springer Berlin Heidelberg, 2012, pp. 17–31, ISBN: 9783642285899. DOI: 10.1007/978-3-642-28589-9_2.

[17] D. Kempf, R. Heß, S. Müthing, and P. Bastian, "Automatic code generation for high-performance discontinuous galerkin methods on modern architectures," *ACM Transactions on Mathematical Software*, vol. 47, no. 1, pp. 1–31, Dec. 2020, ISSN: 1557-7295. DOI: 10.1145/3424144.

[18] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. Kelly, "A study of vectorization for matrix-free finite element methods," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 629–644, Jul. 2020, ISSN: 1741-2846. DOI: 10.1177/1094342020945005.

[19] M. Homolya, R. C. Kirby, and D. A. Ham, *Exposing and exploiting structure: Optimal code generation for high-order finite element methods*, 2017.

[20] O. Stein, E. Grinspun, M. Wardetzky, and A. Jacobson, "Natural boundary conditions for smoothing in geometry processing," *ACM Transactions on Graphics*, vol. 37, no. 2, pp. 1–13, Apr. 2018. DOI: 10.1145/3186564.

[21] D. Li, C. Wang, and J. Wang, *Weak galerkin methods based morley elements on general polytopal partitions*, 2022. eprint: arXiv:2210.17518.

[22] J.-P. BERNARDY and P. JANSSON, "Domain-specific tensor languages," *Journal of Functional Programming*, vol. 35, 2025, ISSN: 1469-7653. DOI:

10.1017/s0956796825000048. [Online]. Available:
http://dx.doi.org/10.1017/S0956796825000048.

[23] M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr, "Discrete differential-geometry operators for triangulated 2-manifolds," in *Visualization and Mathematics III*, H.-C. Hege and K. Polthier, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 35–57, ISBN: 978-3-662-05105-4.

[24] D. N. Arnold, *Finite Element Exterior Calculus* (CBMS-NSF Regional Conference Series in Applied Mathematics). New York, NY: Society for Industrial & Applied Mathematics, Jan. 2019.

[25] A. Abdelfattah, V. Barra, N. Beams, *et al.*, "Gpu algorithms for efficient exascale discretizations," *Parallel Computing*, vol. 108, p. 102 841, Dec. 2021, ISSN: 0167-8191. DOI: 10.1016/j.parco.2021.102841.

[26] D. N. Arnold, R. S. Falk, and R. Winther, "Finite element exterior calculus, homological techniques, and applications," *Acta Numerica*, vol. 15, pp. 1–155, 2006. DOI: 10.1017/S0962492906210018.

[27] K. Wang, Weiwei, Y. Tong, M. Desbrun, and P. Schröder, "Edge subdivision schemes and the construction of smooth vector fields," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 1041–1048, Jul. 2006, ISSN: 0730-0301. DOI: 10.1145/1141911.1141991.

[28] B. Custers and A. Vaxman, "Subdivision directional fields," *ACM Trans. Graph.*, vol. 39, no. 2, Feb. 2020, ISSN: 0730-0301. DOI: 10.1145/3375659.

[29] H. Yin, M. S. Nabizadeh, B. Wu, S. Wang, and A. Chern, "Fluid cohomology," *ACM Transactions on Graphics*, vol. 42, no. 4, pp. 1–25, Jul. 2023, ISSN: 1557-7368. DOI: 10.1145/3592402.

[30] S. Wang, M. S. Nabizadeh, and A. Chern, "Exterior calculus in graphics: Course notes for a siggraph 2023 course," in *ACM SIGGRAPH 2023 Courses*,

ser. SIGGRAPH '23, Los Angeles, California: Association for Computing Machinery, 2023, ISBN: 9798400701450. DOI: 10.1145/3587423.3595525.

[31] L. N. Trefethen, *Approximation Theory and Approximation Practice, Extended Edition*. Philadelphia, PA: Society for Industrial and Applied Mathematics, Jan. 2019, ISBN: 9781611975949. DOI: 10.1137/1.9781611975949.

[32] D. Braess, *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge, England: Cambridge University Press, Apr. 2007, ISBN: 9780511618635. DOI: 10.1017/cbo9780511618635.

[33] P. Gatto, *Mathematical Foundations of Finite Elements and Iterative Solvers*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2022. DOI: 10.1137/1.9781611977097. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611977097.

[34] R. C. Kirby, "A general approach to transforming finite elements," *The SMAI journal of computational mathematics*, vol. 4, pp. 197–224, Apr. 2018, ISSN: 2426-8399.

[35] F. R. A. Aznaran, P. E. Farrell, and R. C. Kirby, "Transformations for piola-mapped elements," *The SMAI Journal of computational mathematics*, vol. 8, pp. 399–437, Jul. 2023, ISSN: 2426-8399. DOI: 10.5802/smai-jcm.91.

[36] J. Grošelj and M. Knez, "A construction of edge b-spline functions for a c1 polynomial spline on two triangles and its application to argyris type splines," *Computers & Mathematics with Applications*, vol. 99, pp. 329–344, 2021.

[37] A. Sky, M. Neunteufel, J. S. Hale, and A. Zilian, "A reissner–mindlin plate formulation using symmetric hu-zhang elements via polytopal transformations," *Computer Methods in Applied Mechanics and Engineering*, vol. 416, p. 116 291, Nov. 2023, ISSN: 0045-7825. DOI: 10.1016/j.cma.2023.116291.

[38] J.-M. Hong and C.-H. Kim, "Discontinuous fluids," *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, pp. 915–920, 2005.

[39] K. B. Ølgaard, A. Logg, and G. N. Wells, "Automated code generation for discontinuous galerkin methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 849–864, 2009.

[40] L. R. Scott and S. Zhang, "Finite element interpolation of nonsmooth functions satisfying boundary conditions," *Mathematics of computation*, vol. 54, no. 190, pp. 483–493, 1990.

[41] A. W. Bargteil, C. Wojtan, J. K. Hodgins, and G. Turk, "A finite element method for animating large viscoplastic flow," *ACM Trans. Graph.*, vol. 26, no. 3, 16–es, Jul. 2007, ISSN: 0730-0301. DOI: 10.1145/1276377.1276397.

[42] G. Irving, C. Schroeder, and R. Fedkiw, "Volume conserving finite element simulations of deformable models," *ACM Trans. Graph.*, vol. 26, no. 3, 13–es, Jul. 2007, ISSN: 0730-0301. DOI: 10.1145/1276377.1276394.

[43] P. Clausen, M. Wicke, J. R. Shewchuk, and J. F. O'Brien, "Simulating liquids and solid-liquid interactions with lagrangian meshes," *ACM Trans. Graph.*, vol. 32, no. 2, Apr. 2013, ISSN: 0730-0301. DOI: 10.1145/2451236.2451243.

[44] B. Smith, F. D. Goes, and T. Kim, "Stable neo-hookean flesh simulation," *ACM Trans. Graph.*, vol. 37, no. 2, Mar. 2018, ISSN: 0730-0301. DOI: 10.1145/3180491.

[45] P. Alliez, D. Cohen-Steiner, O. Devillers, B. Lévy, and M. Desbrun, "Anisotropic polygonal remeshing," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 485–493, Jul. 2003, ISSN: 1557-7368. DOI: 10.1145/882262.882296. [Online]. Available: http://dx.doi.org/10.1145/882262.882296.

[46] W. Jakob, A. Arbree, J. T. Moon, K. Bala, and S. Marschner, "A radiative transfer framework for rendering materials with anisotropic structure," *ACM Trans. Graph.*, vol. 29, no. 4, Jul. 2010, ISSN: 0730-0301. DOI: 10.1145/1778765.1778790.

[47] A. Gruber and E. Aulisa, "Computational p-willmore flow with conformal penalty," *ACM Trans. Graph.*, vol. 39, no. 5, Aug. 2020, ISSN: 0730-0301. DOI: 10.1145/3369387.

[48] M. Kazhdan and H. Hoppe, "Screened poisson surface reconstruction," *ACM Trans. Graph.*, vol. 32, no. 3, Jul. 2013, ISSN: 0730-0301. DOI: 10.1145/2487228.2487237.

[49] A. Longva, F. Löschner, T. Kugelstadt, J. A. Fernández-Fernández, and J. Bender, "Higher-order finite elements for embedded simulation," *ACM Trans. Graph.*, vol. 39, no. 6, Nov. 2020, ISSN: 0730-0301. DOI: 10.1145/3414685.3417853.

[50] A. W. Bargteil and E. Cohen, "Animation of deformable bodies with quadratic bézier finite elements," *ACM Trans. Graph.*, vol. 33, no. 3, Jun. 2014, ISSN: 0730-0301. DOI: 10.1145/2567943.

[51] M. Frâncu, A. Asgeirsson, K. Erleben, and M. J. L. Rønnow, "Locking-proof tetrahedra," *ACM Transactions on Graphics*, vol. 40, no. 2, pp. 1–17, Apr. 2021. DOI: 10.1145/3444949.

[52] O. Rémillard and P. G. Kry, "Embedded thin shells for wrinkle simulation," *ACM Trans. Graph.*, vol. 32, no. 4, Jul. 2013, ISSN: 0730-0301. DOI: 10.1145/2461912.2462018.

[53] Y. Cao, Y. Chen, M. Li, Y. Yang, X. Zhang, M. Aanjaneya, and C. Jiang, "An efficient b-spline lagrangian/eulerian method for compressible flow, shock waves, and fracturing solids," *ACM Trans. Graph.*, vol. 41, no. 5, May 2022, ISSN: 0730-0301. DOI: 10.1145/3519595.

[54] R. Ling, J. Huang, B. Jüttler, F. Sun, H. Bao, and W. Wang, "Spectral quadrangulation with feature curve alignment and element size control," *ACM Trans. Graph.*, vol. 34, no. 1, Dec. 2015, ISSN: 0730-0301. DOI: 10.1145/2653476.

[55] Q. Le, Y. Deng, J. Bu, B. Zhu, and T. Du, "Second-order finite elements for deformable surfaces," in *SIGGRAPH Asia 2023 Conference Papers*, ser. SA '23, , Sydney, NSW, Australia, Association for Computing Machinery, 2023, ISBN: 9798400703157. DOI: 10.1145/3610548.3618186.

[56] M. Desbrun, E. Kanso, and Y. Tong, "Discrete differential forms for computational modeling," in *ACM SIGGRAPH 2006 Courses*, ser. SIGGRAPH '06, Boston, Massachusetts: Association for Computing Machinery, 2006, pp. 39–54, ISBN: 1595933646. DOI: 10.1145/1185657.1185665.

[57] A. Jacobson, E. Tosun, O. Sorkine, and D. Zorin, "Mixed finite elements for variational surface modeling," *Computer Graphics Forum*, vol. 29, no. 5, pp. 1565–1574, Jul. 2010, ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2010.01765.x.

[58] O. Stein, A. Jacobson, M. Wardetzky, and E. Grinspun, "A smoothness energy without boundary distortion for curved surfaces," *ACM Transactions on Graphics*, vol. 39, no. 3, pp. 1–17, Mar. 2020. DOI: 10.1145/3377406.

[59] E. English and R. Bridson, "Animating developable surfaces using nonconforming elements," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–5, Aug. 2008, ISSN: 0730-0301. DOI: 10.1145/1360612.1360665.

[60] J. Huang, Y. Tong, H. Wei, and H. Bao, "Boundary aligned smooth 3d cross-frame field," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 1–8, Dec. 2011, ISSN: 0730-0301. DOI: 10.1145/2070781.2024177.

[61] C. Brandt, L. Scandolo, E. Eisemann, and K. Hildebrandt, "Modeling n-symmetry vector fields using higher-order energies," *ACM Trans. Graph.*, vol. 37, no. 2, Mar. 2018, ISSN: 0730-0301. DOI: 10.1145/3177750.

[62] S. Boyé, P. Barla, and G. Guennebaud, "A vectorial solver for free-form vector gradients," *ACM Trans. Graph.*, vol. 31, no. 6, Nov. 2012, ISSN: 0730-0301. DOI: 10.1145/2366145.2366192.

[63] M. Porrmann, “C1 Finite Elements for Dune,” M.S. thesis, TU Dresden, 2022.

[64] T. Gustafsson and G. McBain, “Scikit-fem: A Python package for finite element assembly,” *Journal of Open Source Software*, vol. 5, no. 52, p. 2369, Aug. 2020, ISSN: 2475-9066. DOI: 10.21105/joss.02369. (visited on 02/14/2025).

[65] P. D. Brubeck and R. C. Kirby, *Fiat: Enabling classical and modern macroelements*, 2025.

[66] P. D. Brubeck, R. C. Kirby, F. Laakmann, and L. Mitchell, *Fiat: Improving performance and accuracy for high-order finite elements*, 2024.

[67] D. A. Ham, P. H. Kelly, L. Mitchell, C. J. Cotter, R. C. Kirby, K. Sagiyama, N. Bouziani, S. Vorderwuelbecke, T. J. Gregory, J. Betteridge, *et al.*, *Firedrake user manual*, 2023.

[68] P. D. Brubeck and P. E. Farrell, “Multigrid solvers for the de rham complex with optimal complexity in polynomial degree,” *SIAM Journal on Scientific Computing*, vol. 46, no. 3, A1549–A1573, 2024.

[69] M. Homolya and D. A. Ham, “A parallel edge orientation algorithm for quadrilateral meshes,” *SIAM Journal on Scientific Computing*, vol. 38, no. 5, S48–S61, 2016.

[70] M. W. Scroggs, J. S. Dokken, C. N. Richardson, and G. N. Wells, “Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes,” *ACM Trans. Math. Softw.*, vol. 48, no. 2, May 2022, ISSN: 0098-3500. DOI: 10.1145/3524456.

[71] P. Fischer, S. Kerkemeier, M. Min, *et al.*, “Nekrs, a gpu-accelerated spectral element navier–stokes solver,” *Parallel Computing*, vol. 114, p. 102 982, Dec. 2022, ISSN: 0167-8191. DOI: 10.1016/j.parco.2022.102982.

[72] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells, "The deal.ii finite element library: Design, features, and insights," *Computers and Mathematics with Applications*, vol. 81, pp. 407–422, Jan. 2021, ISSN: 0898-1221. DOI: 10.1016/j.camwa.2020.02.022.

[73] M. R. Tonks, D. Gaston, P. C. Millett, D. Andrs, and P. Talbot, "An object-oriented finite element framework for multiphysics phase field simulations," *Computational Materials Science*, vol. 51, no. 1, pp. 20–29, Jan. 2012, ISSN: 0927-0256. DOI: 10.1016/j.commatsci.2011.07.028.

[74] H. Jasak, A. Jemcov, Z. Tukovic, *et al.*, *Openfoam: A c++ library for complex physics simulations*, IUC, Dubrovnik, 2007.

[75] S. A. Maas, B. J. Ellis, G. A. Ateshian, and J. A. Weiss, "Febio: Finite elements for biomechanics," *Journal of Biomechanical Engineering*, vol. 134, no. 1, p. 011 005, Jan. 2012, ISSN: 1528-8951. DOI: 10.1115/1.4005694.

[76] J. Shadid, H. Moffat, S. Hutchinson, G. Hennigan, K. Devine, and A. Salinger, *MP Salsa: a finite element computer program for reacting flow problems. Part 1–theoretical development*. Oak Ridge, TN: Office of Scientific and Technical Information (OSTI), May 1996. DOI: 10.2172/237399.

[77] F. Hecht, "New development in freefem++," *Journal of Numerical Mathematics*, vol. 20, no. 3–4, pp. 251–266, Jan. 2012, ISSN: 1570-2820. DOI: 10.1515/jnum-2012-0013.

[78] D. Moxey, C. D. Cantwell, Y. Bao, *et al.*, "Nektar++: Enhancing the capability and application of high-fidelity spectral element methods," *Computer Physics Communications*, vol. 249, p. 107 110, Apr. 2020, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2019.107110.

[79] M. Scroggs, "Symfem: A symbolic finite element definition library," *Journal of Open Source Software*, vol. 6, no. 64, p. 3556, Aug. 2021, ISSN: 2475-9066. DOI: 10.21105/joss.03556.

[80] M. S. Alnæs and K.-A. Mardal, "On the efficiency of symbolic computations combined with code generation for finite element methods," *ACM Trans. Math. Softw.*, vol. 37, no. 1, Jan. 2010, ISSN: 0098-3500. DOI: 10.1145/1644001.1644007.

[81] R. C. Kirby and A. Logg, "A compiler for variational forms," *ACM Trans. Math. Softw.*, vol. 32, no. 3, pp. 417–444, Sep. 2006, ISSN: 0098-3500. DOI: 10.1145/1163641.1163644.

[82] M. E. Rognes, R. C. Kirby, and A. Logg, "Efficient assembly of h(div) and h(curl) conforming finite elements," *SIAM Journal on Scientific Computing*, vol. 31, no. 6, pp. 4130–4151, 2010. DOI: 10.1137/08073901X. eprint: https://doi.org/10.1137/08073901X.

[83] P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes, "Automated derivation of the adjoint of high-level transient finite element programs," *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C369–C393, Jan. 2013, ISSN: 1095-7197. DOI: 10.1137/120873558.

[84] A. Paganini and F. Wechsung, "Fireshape: A shape optimization toolbox for firedrake," *Structural and Multidisciplinary Optimization*, vol. 63, no. 5, pp. 2553–2569, Feb. 2021, ISSN: 1615-1488. DOI: 10.1007/s00158-020-02813-y.

[85] T. H. Gibson, L. Mitchell, D. A. Ham, and C. J. Cotter, "Slate: Extending firedrake's domain-specific abstraction to hybridized solvers for geoscience and beyond," *Geoscientific Model Development*, vol. 13, no. 2, pp. 735–761, Feb. 2020, ISSN: 1991-9603. DOI: 10.5194/gmd-13-735-2020.

[86] R. C. Kirby and L. Mitchell, "Solver composition across the pde/linear algebra barrier," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C76–C98, Jan. 2018, ISSN: 1095-7197. DOI: 10.1137/17m1133208.

[87] P. E. Farrell, R. C. Kirby, and J. Marchena-Menéndez, "Irksome: Automating runge–kutta time-stepping for finite element methods," *ACM Trans. Math. Softw.*, vol. 47, no. 4, Sep. 2021, ISSN: 0098-3500. DOI: 10.1145/3466168.

[88] S. Wu and J. Xu, "Nonconforming Finite Element Spaces for $2m$-th Order Partial Differential Equations on $\mathbb{R}n^$ Simplicial Grids When $m=n+1$," *Mathematics of Computation*, vol. 88, no. 316, pp. 531–551, May 2018, ISSN: 1088-6842. DOI: 10.1090/mcom/3361. [Online]. Available: http://dx.doi.org/10.1090/mcom/3361.

[89] M. Wang and J. Xu, "Minimal finite element spaces for $2m$-th-order partial differential equations in $Rn^$," *Mathematics of Computation*, vol. 82, no. 281, pp. 25–43, Jun. 2012, ISSN: 0025-5718, 1088-6842.

[90] P. G. Ciarlet, "Interpolation error estimates for the reduced hsieh-clough-tocher triangle," *Mathematics of Computation*, vol. 32, no. 142, pp. 335–344, 1978, ISSN: 1088-6842. DOI: 10.1090/s0025-5718-1978-0482249-1. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-1978-0482249-1.

[91] J. H. Argyris, I. Fried, and D. W. Scharpf, "The TUBA Family of Plate Elements for the Matrix Displacement Method," *The Aeronautical Journal*, vol. 72, no. 692, pp. 701–709, Aug. 1968, ISSN: 0001-9240, 2059-6464.

[92] J. Hu and S. Zhang, "The minimal conforming $Hk^$ finite element spaces on $Rn^$ rectangular grids," *Mathematics of Computation*, vol. 84, no. 292, pp. 563–579, Aug. 2014, ISSN: 0025-5718, 1088-6842.

[93] K. Bell, "A refined triangular plate bending finite element," *International Journal for Numerical Methods in Engineering*, vol. 1, no. 1, 101–122, 1969. DOI: 10.1002/nme.1620010108.

[94] J. Hu, T. Lin, and Q. Wu, *A Construction of $Cr^$ Conforming Finite Element Spaces in Any Dimension*, Mar. 2021. arXiv: 2103.14924 `[cs, math]`.

[95] D. N. Arnold and R. Winther, "Mixed finite elements for elasticity," *Numerische Mathematik*, vol. 92, no. 3, pp. 401–419, Sep. 2002, ISSN: 0029-599X, 0945-3245.

[96] D. N. Arnold and S. W. Walker, "The Hellan-Herrmann-Johnson method with curved elements," *SIAM Journal on Numerical Analysis*, vol. 58, no. 5, pp. 2829–2855, Jan. 2020, ISSN: 1095-7170. DOI: 10.1137/19m1288723. [Online]. Available: http://dx.doi.org/10.1137/19M1288723.

[97] D. Arnold and G. Awanou, "Finite element differential forms on cubical meshes," *Mathematics of Computation*, vol. 83, no. 288, pp. 1551–1570, Oct. 2013, ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-2013-02783-4.

[98] X. Tai and R. Winther, "A discrete de Rham complex with enhanced smoothness," *CALCOLO*, vol. 43, no. 4, pp. 287–306, Dec. 2006, ISSN: 0008-0624, 1126-5434.

[99] J. Guzmán and M. Neilan, "Inf-Sup Stable Finite Elements on Barycentric Refinements Producing Divergence–Free Approximations in Arbitrary Dimensions," *SIAM Journal on Numerical Analysis*, vol. 56, no. 5, pp. 2826–2844, Jan. 2018, ISSN: 0036-1429, 1095-7170. DOI: 10.1137/17M1153467.

[100] B. Cockburn and G. Fu, "A systematic construction of finite element commuting exact sequences," *SIAM Journal on Numerical Analysis*, vol. 55, no. 4, pp. 1650–1688, Jan. 2017, ISSN: 1095-7170. DOI: 10.1137/16m1073352. [Online]. Available: http://dx.doi.org/10.1137/16M1073352.

[101] P. Ciarlet Jr., C. F. Dunkl, and S. A. Sauter, *A Family of Crouzeix-Raviart Finite Elements in 3D*, Mar. 2017. arXiv: 1703.03224 `[math]`.

[102] G. Karniadakis and S. J. Sherwin, *Spectral/Hp Element Methods for CFD* (Numerical Mathematics and Scientific Computation). New York: Oxford University Press, 1999, ISBN: 978-0-19-510226-0.

[103] M. Kronbichler, K. Kormann, N. Fehn, P. Munch, and J. Witte, *A Hermite-like basis for faster matrix-free evaluation of interior penalty discontinuous Galerkin operators*, Jul. 2019. arXiv: 1907.08492 `[cs, math]`.

[104] M. Ainsworth and S. Jiang, "A systematic approach to constructing preconditioners for the hp-version mass matrix on unstructured and hybrid finite element meshes," *SIAM Journal on Scientific Computing*, vol. 44, no. 2, A901–A934, 2022. DOI: 10.1137/20M1382519.

[105] N. Nigam and J. Phillips, *High-order finite elements on pyramids: Approximation spaces, unisolvency and exactness*, Oct. 2010. arXiv: math/0610206.

[106] J. Guzmán, D. Leykekhman, and M. Neilan, "A family of non-conforming elements and the analysis of nitsche's method for a singularly perturbed fourth order problem," *Calcolo*, vol. 49, no. 2, pp. 95–125, Sep. 2011, ISSN: 1126-5434. DOI: 10.1007/s10092-011-0047-8. [Online]. Available: http://dx.doi.org/10.1007/s10092-011-0047-8.

[107] A. Gillette, T. Kloefkorn, and V. Sanders, "Computational Serendipity and Tensor Product Finite Element Differential Forms," *The SMAI journal of computational mathematics*, vol. 5, pp. 1–21, Mar. 2019, ISSN: 2426-8399.

[108] R. C. Kirby, "Fast simplicial finite element algorithms using Bernstein polynomials," *Numerische Mathematik*, vol. 117, no. 4, pp. 631–652, Apr. 2011, ISSN: 0029-599X, 0945-3245.

[109] J. Chan and T. Warburton, "A Short Note on a Bernstein–Bezier Basis for the Pyramid," *SIAM Journal on Scientific Computing*, vol. 38, no. 4, A2162–A2172, Jan. 2016, ISSN: 1064-8275, 1095-7197.

[110] C. Bernardi and G. Raugel, "Analysis of some finite elements for the stokes problem," *Mathematics of Computation*, vol. 44, no. 169, pp. 71–79, 1985, ISSN: 1088-6842. DOI: 10.1090/s0025-5718-1985-0771031-7. [Online]. Available: http://dx.doi.org/10.1090/S0025-5718-1985-0771031-7.

[111] J. Chan and T. Warburton, "Orthogonal Bases for Vertex-Mapped Pyramids," *SIAM Journal on Scientific Computing*, vol. 38, no. 2, A1146–A1170, Jan. 2016, ISSN: 1064-8275, 1095-7197.

[112] A. Buffa and S. H. Christiansen, "A dual finite element complex on the barycentric refinement," *Comptes Rendus Mathematique*, vol. 340, no. 6, pp. 461–464, Mar. 2005, ISSN: 1631073X. DOI: 10.1016/j.crma.2004.12.022.

[113] T. Regge, "General relativity without coordinates," *Il Nuovo Cimento*, vol. 19, no. 3, 558–571, 1961. DOI: 10.1007/BF02733251.

[114] K. A. Mardal, X.-C. Tai, and R. Winther, "A Robust Finite Element Method for Darcy–Stokes Flow," *SIAM Journal on Numerical Analysis*, vol. 40, no. 5, pp. 1605–1631, Jan. 2002, ISSN: 0036-1429, 1095-7170.

[115] J. Arf and B. Simeon, *Structure-preserving Discretization of the Hessian Complex based on Spline Spaces*, Sep. 2021. arXiv: 2109.05293 [cs, math].

[116] M. Neunteufel, "Mixed Finite Element Methods For Nonlinear Continuum Mechanics And Shells," Ph.D. dissertation, TU Wien, 2021, p. 189.

[117] B. Cockburn and W. Qiu, "Commuting diagrams for the TNT elements on cubes," *Mathematics of Computation*, vol. 83, 603–633, 2014. DOI: 10.1090/S0025-5718-2013-02729-9.

[118] B. Cockburn and G. Fu, "A systematic construction of finite element commuting exact sequences," *SIAM journal on numerical analysis*, vol. 55, 1650–1688, 2017. DOI: 10.1137/16M1073352.

[119] D. N. Arnold, G. Awanou, and R. Winther, "NONCONFORMING TETRAHEDRAL MIXED FINITE ELEMENTS FOR ELASTICITY," *Mathematical Models and Methods in Applied Sciences*, vol. 24, no. 04, pp. 783–796, Apr. 2014, ISSN: 0218-2025, 1793-6314. DOI: 10.1142/S021820251350067X.

[120] G. Matthies and L. Tobiska, "Inf-sup stable non-conforming finite elements of arbitrary order on triangles," *Numerische Mathematik*, vol. 102, no. 2, pp. 293–309, Dec. 2005, ISSN: 0029-599X, 0945-3245.

[121] P. Wadler, "Linear types can change the world!" In *Programming concepts and methods*, North-Holland, Amsterdam, vol. 3, 1990, p. 5.

[122] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*, 2014, pp. 103–104.

[123] A. Meurer, C. P. Smith, M. Paprocki, *et al.*, "Sympy: Symbolic computing in python," *PeerJ Computer Science*, vol. 3, e103, Jan. 2017, ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. [Online]. Available: https://doi.org/10.7717/peerj-cs.103.

[124] L. Perumal, "A brief review on polygonal/polyhedral finite element methods," *Mathematical Problems in Engineering*, vol. 2018, no. 1, p. 5 792 372, 2018. DOI: https://doi.org/10.1155/2018/5792372. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2018/5792372.

[125] O. J. Sutton, "The virtual element method in 50 lines of matlab," *Numerical Algorithms*, vol. 75, no. 4, pp. 1141–1159, Dec. 2016, ISSN: 1572-9265. DOI: 10.1007/s11075-016-0235-3.

[126] M. W. Scroggs, J. S. Dokken, C. N. Richardson, and G. N. Wells, "Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes," *ACM Trans. Math. Softw.*, vol. 48, no. 2, May 2022, ISSN: 0098-3500. DOI: 10.1145/3524456. [Online]. Available: https://doi.org/10.1145/3524456.

[127] V. Hapla, M. G. Knepley, M. Afanasiev, C. Boehm, M. van Driel, L. Krischer, and A. Fichtner, "Fully parallel mesh i/o using petsc dmplex with an application to waveform modeling," *SIAM Journal on Scientific Computing*, vol. 43, no. 2, pp. C127–C153, 2021.

[128] J. Bradbury, R. Frostig, P. Hawkins, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, version 0.3.13, Google, 2018. [Online]. Available: http://github.com/jax-ml/jax.

[129] J. Ansel, E. Yang, H. He, *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 929–947, ISBN: 9798400703850. DOI: 10.1145/3620665.3640366. [Online]. Available: https://doi.org/10.1145/3620665.3640366.

[130] A. Shajii, G. Ramirez, H. Smajlović, J. Ray, B. Berger, S. Amarasinghe, and I. Numanagić, "Codon: A compiler for high-performance pythonic applications and dsls," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC '23, New York City, NY: ACM, Feb. 2023, pp. 191–202. DOI: 10.1145/3578360.3580275. [Online]. Available: http://dx.doi.org/10.1145/3578360.3580275.

[131] J. Benzaken, J. A. Evans, and R. Tamstorf, "Constructing nitsche's method for variational problems," *Archives of Computational Methods in Engineering*, vol. 31, no. 4, pp. 1867–1896, Apr. 2024, ISSN: 1886-1784. DOI: 10.1007/s11831-023-09953-6.

[132] O. Stein, E. Grinspun, A. Jacobson, and M. Wardetzky, *A mixed finite element method with piecewise linear elements for the biharmonic equation on surfaces*, 2019. DOI: 10.48550/ARXIV.1911.08029.

[133] B. Gräßle, "Optimal multilevel adaptive fem for the argyris element," *Computer Methods in Applied Mechanics and Engineering*, vol. 399, p. 115 352, Sep. 2022, ISSN: 0045-7825. DOI: 10.1016/j.cma.2022.115352.

[134] N. Ferraro, S. Jardin, and X. Luo, "Essential boundary conditions with straight c1 finite elements in curved domains," Princeton Plasma Physics Lab., Tech. Rep., 2010.

[135] R. Zhao, M. Desbrun, G.-W. Wei, and Y. Tong, "3d hodge decompositions of edge- and face-based vector fields," *ACM Transactions on Graphics*, vol. 38, no. 6, pp. 1–13, Nov. 2019, ISSN: 1557-7368. DOI: 10.1145/3355089.3356546.

[136] H. Bhatia, G. Norgard, V. Pascucci, and P.-T. Bremer, "The helmholtz-hodge decomposition—a survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 8, pp. 1386–1404, Aug. 2013, ISSN: 1077-2626. DOI: 10.1109/tvcg.2012.316.

[137] P. Leopardi and A. Stern, "The abstract hodge–dirac operator and its stable discretization," *SIAM Journal on Numerical Analysis*, vol. 54, no. 6, pp. 3258–3279, 2016.

[138] Z. Su, Y. Tong, and G. Wei, "Hodge decomposition of vector fields in cartesian grids," in *SIGGRAPH Asia 2024 Conference Papers*, ser. SA '24, New York City, NY: ACM, Dec. 2024, pp. 1–10. DOI: 10.1145/3680528.3687602.

[139] M. Holst and A. Stern, "Geometric variational crimes: Hilbert complexes, finite element exterior calculus, and problems on hypersurfaces," *Foundations of Computational Mathematics*, vol. 12, no. 3, pp. 263–293, Apr. 2012, ISSN: 1615-3383. DOI: 10.1007/s10208-012-9119-7.

[140] K. Poelke and K. Polthier, "Boundary-aware hodge decompositions for piecewise constant vector fields," *Computer-Aided Design*, vol. 78, pp. 126–136, Sep. 2016, ISSN: 0010-4485. DOI: 10.1016/j.cad.2016.05.004.

[141] Z. Su, Y. Tong, and G.-W. Wei, *Topology-preserving hodge decomposition in the eulerian representation*, 2024. eprint: arXiv:2408.14356.

[142] Y. Tong, S. Lombeyda, A. N. Hirani, and M. Desbrun, "Discrete multiscale vector field decomposition," *ACM transactions on graphics (TOG)*, vol. 22, no. 3, pp. 445–452, 2003.

[143] K. Poelke, "Hodge-type decompositions for piecewise constant vector fields on simplicial surfaces and solids with boundary," Ph.D. dissertation, FU Berlin, 2017.

[144] C. Davini, "Gaussian curvature and babuška's paradox in the theory of plates," in *Rational Continua, Classical and New*. Milan, Italy: Springer Milan, 2003, pp. 67–87, ISBN: 9788847022317. DOI: 10.1007/978-88-470-2231-7_6.

[145] I. Boksebeld and A. Vaxman, "High-order directional fields," *ACM Transactions on Graphics*, vol. 41, no. 6, pp. 1–17, Nov. 2022, ISSN: 1557-7368. DOI: 10.1145/3550454.3555455.

[146] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, *et al.* "Petsc users manual," Argonne National Laboratory. (2019).

[147] P. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent, "A fully asynchronous multifrontal solver using distributed dynamic scheduling," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.

[148] P. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary, "Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures," *ACM Transactions on Mathematical Software*, vol. 45, 2:1–2:26, 1 2019.

[149] A. Linke and L. G. Rebholz, "Pressure-induced locking in mixed methods for time-dependent (navier–)stokes equations," *Journal of Computational Physics*, vol. 388, pp. 350–356, Jul. 2019, ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.03.010.

[150] J. Guzmán and L. R. Scott, "The scott-vogelius finite elements revisited," *Mathematics of Computation*, vol. 88, no. 316, pp. 515–529, 2019.

[151] P. E. Farrell, L. Mitchell, L. R. Scott, and F. Wechsung, "A reynolds-robust preconditioner for the scott-vogelius discretization of the stationary incompressible navier-stokes equations," *The SMAI journal of computational mathematics*, vol. 7, pp. 75–96, 2021.

[152] K. Vacek and P. Sváček, "Finite element approximation of fluid structure interaction using taylor-hood and scott-vogelius elements," in *Topical Problems of Fluid Mechanics 2024*, ser. TPFM, Prague, Czech Republic: Institute of Thermomechanics of the Czech Academy of Sciences; CTU in Prague Faculty of Mech. Engineering Dept. Tech. Mathematics, 2024, pp. 232–239. DOI: 10.14311/tpfm.2024.031. [Online]. Available: http://dx.doi.org/10.14311/TPFM.2024.031.

[153] A. Linke, G. Matthies, and L. Tobiska, "Non-nested multi-grid solvers for mixed divergence-free scott–vogelius discretizations," *Computing*, vol. 83, no. 2, pp. 87–107, 2008.