Key words: Discontinuous Petrov Galerkin, adaptive finite elements

Camellia: A Software Framework for Discontinuous Petrov-Galerkin Methods

Nathan V. Roberts[a,*]

[a]*Argonne Leadership Computing Facility, Argonne National Laboratory, Argonne, IL.*

Abstract

The discontinuous Petrov-Galerkin (DPG) methodology of Demkowicz and Gopalakrishnan minimizes the solution residual in a user-determinable energy norm and offers a built-in mechanism for evaluating error in the energy norm, among other desirable features. However, the methodology also brings with it some additional complexity for researchers who wish to experiment with DPG in their computations. In this paper, we introduce *Camellia*, a software framework whose central design goal is to enable developers to create efficient hp-adaptive DPG solvers with minimal effort.

1. Introduction

The discontinuous Petrov-Galerkin (DPG) methodology of Demkowicz and Gopalakrishnan introduced in 2009 [12, 14] offers many desirable features, including minimizing the solution residual in a user-determinable energy norm and a built-in mechanism for evaluating error in the energy norm. In contrast to standard discontinuous Galerkin (DG) methods, DPG employs a test space that differs from the trial space, and also uses *conforming* trace discretizations to represent the unknowns on inter-element boundaries. In contrast to streamline upwind Petrov-Galerkin (SUPG) methods, DPG's test space is computed on the fly.

These features present some special challenges when implementing a DPG solver. There are several different kinds of unknowns, each of which requires appropriate treatment. For example, consider a Poisson problem $\Delta\phi = f$. A standard DPG formulation[1] for this problem involves introducing a new unknown $\boldsymbol{\psi} = \nabla\phi$, multiplying the resulting first-order system by test functions $\boldsymbol{\tau}$ and q, integrating by parts, and introducing new *trace* variables $\widehat{\phi}$ and $\widehat{\psi}_n$ on the mesh skeleton, resulting in the formulation

$$\begin{aligned} b(u,v) = &- \int_K \phi \nabla \cdot \boldsymbol{\tau} - \int_K \boldsymbol{\psi} \cdot \boldsymbol{\tau} + \int_{\partial K} \widehat{\phi}\, \boldsymbol{\tau} \cdot \mathbf{n} \\ &- \int_K \boldsymbol{\psi} \cdot \nabla q + \int_{\partial K} \widehat{\psi}_n \, q \, \mathrm{sgn}(n) \; = l(v) = \int_K f q, \end{aligned}$$

*Corresponding author

[1]See Demkowicz and Gopalakrishnan [13], e.g.

for bilinear form $b(\cdot,\cdot)$ and load $l(\cdot)$ on element K with boundary ∂K, where $u = (\phi, \boldsymbol{\psi}, \widehat{\phi}, \widehat{\psi}_n)$ is a group variable representing the solution, and $v = (\boldsymbol{\tau}, q)$ is a group variable representing a test function. The spaces[2] for the variables are as follows: $\phi \in L^2, \boldsymbol{\psi} \in \boldsymbol{L}^2, \widehat{\phi} \in \operatorname{tr}\left(H^1\right), \widehat{\psi}_n \in \operatorname{tr}\left(H(\operatorname{div})\right), \boldsymbol{\tau} \in H(\operatorname{div})$, and $q \in H^1$. The test variables are nonconforming and discontinuous, but the trial variables have continuities conforming to their functional setting. Specifically, normal continuity is enforced for $\widehat{\psi}_n$ and pointwise (vertex) continuity is enforced for $\widehat{\phi}$. The discrete test functions are determined on the fly in such a way that the solution satisfies the residual-minimizing property.

On the one hand, these requirements pose a burden for the analyst who wants to experiment with DPG to see whether it will work well for a given problem. On the other hand, the regularities within the methodology—e.g., the fact that DPG comes with a built-in error evaluation mechanism—provide an opportunity for automation. Addressing the first point while taking advantage of the second is the essential idea behind *Camellia*, which we introduce in the present work. The central design goal of Camellia is to provide a software framework in which hp-adaptive DPG solvers can be implemented for an arbitrary PDE with minimal effort.

Camellia enters an already well-populated field of finite element packages. Among the more popular open source packages are `deal.II` [3], libMesh [20], and FEniCS [22]. `deal.II` is a C++ library that supports hypercube topologies for finite element computations in one to three spatial dimensions, including support for distributed stiffness matrix computation and (anisotropic) h- and p-refinements; the `deal.II` website boasts an enormous amount of documentation, including a host of tutorials. Parallel mesh representations are available through the `p4est` library [2]. libMesh is likewise a C++ library; it supports highly flexible geometry by offering many element topologies (including simplices, hypercubes, and pyramids, and with the possibility of adding custom element types), and supports up to three dimensions and both h- and p- refinements. FEniCS offers a Python interface for rapid specification of finite element problems posed on simplicial meshes; its goal is to allow analysts to simply type in a variational form and start computing.[3]

Our aim in Camellia is to support the implementation of DPG solvers with flexible geometry (like libMesh), rapid development (like FEniCS), and scalable performance (like `deal.II`, particularly when combined with `p4est`). At present, Camellia supports 2D (potentially curvilinear) meshes of triangles and quads of variable polynomial order, provides mechanisms for rapid specification of DPG variational forms, supports h- and p- refinements, and supports distributed computation of the stiffness matrix, among other features. Work on support for 3D and space-time discretizations is underway—and we plan to release Camellia under an open source license after that work is complete.

Camellia depends on several supporting packages within Sandia's Trilinos library [19]. The Intrepid package provides conforming nodal basis functions on 1-, 2-, and 3-dimensional elements on hypercube and simplicial topologies (among others); the Shards package defines those topologies. The Teuchos package provides reference-counted pointers and a convenient MPI wrapper. Epetra offers support for linear algebra broadly, including facilities for distributed storage of the system matrix. Zoltan allows the use of space-filling curves to determine spatially local mesh partitions (which reduce the communication costs associated with data along the partition boundaries). Amesos provides an

[2]Here, we are eliding for the sake of simplicity some details of the construction—for example, the trace variables lie in spaces posed on the mesh skeleton. More detail is provided in Section 3 below. For complete details, see [25], e.g.

[3]It is worth noting that it is possible to write DPG solvers using any of these packages; indeed, we are aware of some such efforts, in the context of particular problems, for both `deal.II` and FEniCS. One of the things that distinguishes Camellia from these efforts is the ease with which an efficient DPG solver for a new, arbitrary PDE may be written.

abstraction layer that allows Camellia's code for solving the global system to be essentially independent of the choice of solver—whether this be KLU (a serial direct solver), MUMPS (a parallel direct solver) [1], or an iterative solver from Trilinos's ML package.

As a framework, Camellia continues to evolve in significant ways as we add support for 3D and space-time elements. In the present work, we therefore focus on parts of Camellia that we consider both mature and of particular interest. The structure of this paper is as follows. We offer a brief discussion of our software design philosophy in Section 2. In Section 3, we explain the mathematical steps an analyst must go through to derive a DPG formulation for a given PDE, using a Stokes formulation as a motivating example. We mirror these steps in Section 4, in which we describe the steps one must go through to implement the formulation in Camellia. In both of these sections, the focus is on execution rather than either the mathematical details or the details of Camellia's implementation. In Section 5, we go into more detail regarding some of the design choices within Camellia. We describe our approach to verification in Section 6, and discuss some of the research projects that have been undertaken using Camellia in Section 7. We conclude in Section 8.

2. Software Design Philosophy

Our first goal in Camellia is to enable developer efficiency. With that in mind, several principles guide our software design strategy; we want a framework that is

- easy to use: it should be possible to write succinct code that hews close to the mathematical formulation;
- general: the framework avoids making assumptions about which PDEs are of interest; and
- safe: among other things, the framework should check that method arguments appear reasonable, and throw exceptions when they are not.

These principles are mutually reinforcing; for example, because Camellia embeds a distinction between test and trial spaces, the code reflects the mathematics and it is possible for Camellia to check for programmatic errors such as adding a test variable to a trial variable, which may be nonsense mathematically if these belong to different spaces. We are generally willing to sacrifice a certain amount of computational efficiency for the sake of developer efficiency; however, it is worth noting that the usual result of the above principles is a relatively small number of highly reused computational kernels, which are then susceptible to optimization.

3. DPG Step by Step

3.1. Abstract DPG Method

We will now briefly derive DPG, motivating it as a minimum residual method. Suppose that U is the trial space, and V the test space (both Hilbert) for a well-posed variational problem $b(u, v) = l(v)$. Writing this in the operator form $Bu = l$, where $B : U \to V'$, we seek to minimize the residual for the discrete space $U_h \subset U$:

$$u_h = \underset{w_h \in U_h}{\arg\min} \, \frac{1}{2} \left\| B w_h - l \right\|_{V'}^2 .$$

Now, the dual space V' is not especially easy to work with (the norm on the space requires the evaluation of a supremum); we would prefer to work with V itself. Recalling that the Riesz operator $R_V : V \to V'$ defined by

$$\langle R_V v, \delta v\rangle = (v, \delta v)_V, \quad \forall \delta v \in V,$$

where $\langle \cdot, \cdot\rangle$ denotes the duality pairing between V' and V, is an *isometry*—that is, $\|R_V v\|_{V'} = \|v\|_V$—we can rewrite the term we want to minimize as a norm in V:

$$\frac{1}{2}\|Bu_h - l\|_{V'}^2 = \frac{1}{2}\left\|R_V^{-1}(Bu_h - l)\right\|_V^2 = \frac{1}{2}\left(R_V^{-1}(Bu_h - l), R_V^{-1}(Bu_h - l)\right)_V. \tag{1}$$

The first-order optimality condition requires that the Gâteaux derivative of (1) be equal to zero for minimizer u_h; we have

$$\left(R_V^{-1}(Bu_h - l), R_V^{-1}B\delta u_h\right)_V = 0, \quad \forall \delta u_h \in U_h.$$

By the definition of R_V, the preceding equation is equivalent to

$$\langle Bu_h - l, R_V^{-1}B\delta u_h\rangle = 0 \quad \forall \delta u_h \in U_h. \tag{2}$$

Now, if we identify $v_{\delta u_h} = R_V^{-1}B\delta u_h$ as a test function, we can rewrite (2) as

$$b(u_h, v_{\delta u_h}) = l(v_{\delta u_h}).$$

Note that the last equation is exactly the original variational form, tested with a special function $v_{\delta u_h}$ that corresponds to $\delta u_h \in U_h$; we call $v_{\delta u_h}$ an *optimal test function*. The DPG method is then to solve the problem $b(u_h, v_{\delta u_h}) = l(v_{\delta u_h})$ with optimal test functions $v_{\delta u_h} \in V$ that solve the problem

$$(v_{\delta u_h}, \delta v)_V = \langle R_V v_{\delta u_h}, \delta v\rangle = \langle B\delta u_h, \delta v\rangle = b(\delta u_h, \delta v), \quad \forall \delta v \in V. \tag{3}$$

In standard conforming methods, test functions are continuous over the entire domain, which would mean that solving (3) would require computations on the global mesh, making the method impractical. In DPG, we use test functions that are discontinuous across elements, so that (3) becomes a local problem—that is, it can be solved element by element. Of course, solving (3) exactly would still require inversion of the infinite-dimensional Riesz map; we approximate this by using an "enriched" test space V_h of polynomial order higher than that of the trial space U_h. Note that the test functions $v_{\delta u_h}$ immediately give rise to a hermitian positive definite stiffness matrix; if $\{e_i\}$ is a basis for U_h, we have:

$$b(e_i, v_{e_j}) = (v_{e_i}, v_{e_j})_V = \overline{(v_{e_j}, v_{e_i})_V} = \overline{b(e_j, v_{e_i})}.$$

It should be pointed out that we have not made any assumptions about the inner product on V. An important point is that by an appropriate choice of test space inner product, the induced energy norm on the trial space can be made to coincide with the norm of interest [26]; DPG then delivers the best approximation error in that norm. In practice this optimal test space inner product is approximated by a "localizable" inner product, and DPG delivers the best approximation error up to a mesh-independent constant. That is,

$$\|u - u_h\|_U \leq \frac{M}{\gamma_{DPG}} \inf_{w_h \in U_h} \|u - w_h\|_U,$$

where $M = O(1)$ and γ_{DPG} is mesh-independent, and γ_{DPG} is of the order of inf-sup constants for the strong operator and its adjoint (see [25]). We therefore say that DPG is *automatically stable*, modulo any error in solving for the test functions $v_{\delta u_h}$. When the norm of interest on the trial space is the L^2 norm, we have a standard approach[4] to localization of the inner product; we refer to the norm induced on the test space by this localizable inner product as the *graph norm*. We return to the graph norm below, in the discussion of the choice of test space inner product for a Stokes formulation.

3.2. Stokes on a Fixed Mesh with Straight Edges

We begin with the classical strong form of the Stokes equations:

$$\begin{aligned} \nabla p - \mu \Delta \boldsymbol{u} &= \boldsymbol{f} \\ \nabla \cdot \boldsymbol{u} &= 0, \end{aligned}$$

on some domain Ω, where $\boldsymbol{u}$ is the velocity, p is the pressure, and $\boldsymbol{f}$ is a vector-valued forcing function. By appropriate non-dimensionalization, we may take $\mu = 1$ without loss of generality. We assume that Dirichlet boundary conditions $\boldsymbol{u} = \boldsymbol{g}$ are specified on the boundary $\partial\Omega$. We also assume a zero-mean condition on the pressure:

$$\int_\Omega p = 0.$$

The steps to solve the system using DPG are as follows:

1. Determine the variational formulation.
2. Specify boundary conditions.
3. Specify the test space inner product.
4. Define discrete trial and test spaces (including the mesh).
5. Compute optimal test functions.
6. Assemble the stiffness matrix.
7. Solve the global problem.

We treat each of these in turn.

Variational Formulation. In DPG, our standard practice is to use an *ultra-weak* variational formulation, a first-order system in which all derivatives have been moved onto test functions. Generally speaking, there are several ways of doing this. Here, we derive the velocity-gradient-pressure (VGP) formulation for the Stokes equations. We begin by introducing $\boldsymbol{\sigma} = \nabla \boldsymbol{u}$ to get a first-order system:

$$\begin{aligned} \nabla p - \nabla \cdot \boldsymbol{\sigma} &= \boldsymbol{f}, \\ \boldsymbol{\sigma} - \nabla \boldsymbol{u} &= \boldsymbol{0}, \\ \nabla \cdot \boldsymbol{u} &= 0. \end{aligned}$$

[4]It is worth noting that, particularly in the presence of small parameters, this standard approach may not be the best one. See, for example, Chan et al. on robust norms for convection-diffusion [8].

Testing with $(\boldsymbol{v}, q, \boldsymbol{\tau})$, and integrating by parts on a mesh Ω_h with skeleton Γ_h, we have

$$
\begin{aligned}
(\boldsymbol{\sigma} - p\boldsymbol{I}, \nabla \boldsymbol{v})_{\Omega_h} - \langle (\boldsymbol{\sigma} - p\boldsymbol{I})\boldsymbol{n}, \boldsymbol{v}\rangle_{\Gamma_h} &= (\boldsymbol{f}, \boldsymbol{v})_{\Omega_h},\\
(\boldsymbol{u}, \nabla q)_{\Omega_h} - \langle \boldsymbol{u}\cdot\boldsymbol{n}, q\rangle_{\Gamma_h} &= 0,\\
(\boldsymbol{\sigma}, \boldsymbol{\tau})_{\Omega_h} + (\boldsymbol{u}, \nabla\cdot\boldsymbol{\tau})_{\Omega_h} - \langle \boldsymbol{u}, \boldsymbol{\tau}\boldsymbol{n}\rangle_{\Gamma_h} &= 0.
\end{aligned}
$$

In the ultra-weak formulation, since we take no derivatives of our trial variables, we take our so-called *field* variables to be in $L^2(\Omega_h)$, which means that we cannot speak of these on the mesh skeleton Γ_h. We therefore introduce new variables $\widehat{\boldsymbol{t}}_n \overset{\text{def}}{=} (\boldsymbol{\sigma} - p\boldsymbol{I})\boldsymbol{n}$ and $\widehat{\boldsymbol{u}}$. The hat notation indicates that these variables are only defined on the mesh skeleton—$\widehat{\boldsymbol{t}}_n \in \boldsymbol{H}^{-1/2}(\Gamma_h)$ is a trace of an $H(\text{div})$ function (which we call a *flux* variable), and $\widehat{\boldsymbol{u}} \in \boldsymbol{H}^{1/2}(\Gamma_h)$ is a trace of an $\boldsymbol{H}^1$ function. Defining group variables $u = (\boldsymbol{u}, p, \boldsymbol{\sigma}), \widehat{u} = (\widehat{\boldsymbol{u}}, \widehat{\boldsymbol{t}}_n)$ and $v = (\boldsymbol{v}, q, \boldsymbol{\tau})$, we arrive at our ultra-weak variational formulation:

$$
\begin{aligned}
b((u, \widehat{u}), v) = (\boldsymbol{\sigma} - p\boldsymbol{I}, \nabla\boldsymbol{v})_{\Omega_h} - \left\langle \widehat{\boldsymbol{t}}_n, \boldsymbol{v}\right\rangle_{\Gamma_h}&\\
+ (\boldsymbol{u}, \nabla q)_{\Omega_h} - \langle \widehat{\boldsymbol{u}}\cdot\boldsymbol{n}, q\rangle_{\Gamma_h}&\\
+ (\boldsymbol{\sigma}, \boldsymbol{\tau})_{\Omega_h} + (\boldsymbol{u}, \nabla\cdot\boldsymbol{\tau})_{\Omega_h} - \langle \widehat{\boldsymbol{u}}, \boldsymbol{\tau}\boldsymbol{n}\rangle_{\Gamma_h} = (\boldsymbol{f}, \boldsymbol{v})_{\Omega_h} = l(v).&
\end{aligned}
\tag{4}
$$

Stokes Discretization. DPG does not impose any particular constraints on the basis functions used to represent the various components of the discrete solution. However, we do have the guarantee that the error in the solution is minimized in the energy norm, and under certain modest assumptions (see [25]) if we choose the graph norm discussed below, we have an inequality of the form

$$
\begin{aligned}
&\left(\|u - u_h\|^2 + \|\widehat{u} - \widehat{u}_h\|^2_{\hat{H}_A(\Gamma_h)}\right)^{1/2}\\
&\le \frac{M}{\gamma_{DPG}} \inf_{(w_h, \widehat{w}_h)} \left(\|u - w_h\|^2 + \|\widehat{u} - \widehat{w}_h\|^2_{\hat{H}_A(\Gamma_h)}\right)^{1/2},
\end{aligned}
\tag{5}
$$

where the group variables u and $\widehat{u}$, defined above, are the exact solution, while u_h and $\widehat{u}_h$ are their discrete solution counterparts, M and γ_{DPG} are mesh-independent constants, and $\|\cdot\|_{\hat{H}_A(\Gamma_h)}$ is the "natural" norm on the traces, the minimum energy extension norm.

Assuming u is sufficiently smooth, for a discrete L^2 space comprised of polynomials of order k, we expect best h-convergence rates of $k+1$; that is, we have[5]

$$
\inf_{w_h \in U_h} \|u - w_h\| \le C_1 h^{k+1} \tag{6}
$$

for some mesh-independent constant C_1. It can be shown that, for traces $\widehat{w}_h$ whose $H^{-1/2}(\Gamma_h)$ and $H^{1/2}(\Gamma_h)$ components are approximated by polynomials of orders k and $k+1$, respectively,

$$
\inf_{\widehat{w}_h \in \hat{H}_A(\Gamma_h)} \|\widehat{u} - \widehat{w}_h\|_{\hat{H}_A(\Gamma_h)} \le C_2 h^{k+1}
$$

[5]Note that the U_h in equation 6, in contrast to that in Section 3.1, is comprised of the polynomial spaces corresponding just to the *field* variables $\boldsymbol{u}, p, \boldsymbol{\sigma}$.

for some mesh-independent constant C_2. For details and further references, see [13, pp. 7-8]. Combining this with equations (5) and (6), we then have the bound

$$\|u - u_h\| \leq Ch^{k+1}$$

for $C = \max(C_1, C_2)$.

We can also motivate the choice of polynomial orders for the trial space intuitively from the relative orders of H^1 and L^2 discretizations. If we define k as the polynomial order of approximation of field variables, because these belong to L^2, it is natural to choose $k+1$ as the H^1 order. The traces of $H^1(K)$ functions ($\widehat{u}_1$ and $\widehat{u}_2$) belong to $H^{1/2}(\partial K)$, a stronger space than $L^2(K)$, so that $k+1$ is a natural order of approximation for these. The traces of $H(\text{div, K})$ functions $\widehat{\boldsymbol{t}}_n$ belong to $H^{-1/2}(\partial K)$, a weaker space than L^2, so that k is a natural order of approximation for these.

Thus, the natural choice for our trial space discretization is a polynomial space such that

$$\begin{aligned} &\boldsymbol{u} \in \mathbf{P}^k(K), \widehat{\boldsymbol{u}} \in \mathbf{P}^{k+1}(\partial K), \\ &p \in \mathbf{P}^k(K), \\ &\boldsymbol{\sigma} \in \mathbf{P}^k(K), \widehat{\boldsymbol{t}}_n \in \mathbf{P}^k(\partial K), \end{aligned}$$

for each element K. With this choice of space, we can expect our discrete solution to converge at a rate of $k+1$, provided that the exact solution is smooth.

Note that the fact that $\widehat{\boldsymbol{u}} \in H^{1/2}(\Gamma_h)$ suggests that we should enforce continuity at vertices; while the fact that $\widehat{\boldsymbol{t}}_n \in H^{-1/2}(\Gamma_h)$ suggests that we should allow discontinuities at the vertices.

Finally, we must choose a discretization for our test space. Again we have considerable freedom, but a natural choice is an element-wise basis that conforms to spaces supporting the differential operators taken on the test space. That is, since we take the divergence of $\boldsymbol{\tau}$, a natural choice for $\boldsymbol{\tau}$ is a vector $H(\text{div})$-conforming basis. In terms of polynomial order, we take the maximum $k+1$ order of discretization used in our trial space, and "enrich" it by some amount Δk. Our usual choice in 2D is $\Delta k = 2$. The essential tradeoff is between local computational costs (the choice of test space polynomial order only affects the determination of optimal test functions, a local operation) and accurate determination of the optimal test functions.

Thus we select a test space such that

$$\begin{aligned} &\boldsymbol{v} \in \mathbf{P}^{k+1+\Delta k}(K) \cap \boldsymbol{H}^1(K), \\ &q \in \mathbf{P}^{k+1+\Delta k}(K) \cap H^1(K), \\ &\boldsymbol{\tau} \in \mathbf{P}^{k+1+\Delta k}(K) \cap \boldsymbol{H}(\text{div, K}). \end{aligned}$$

By default, Camellia employs nodal conforming bases provided by Trilinos's Intrepid package.

Stokes Boundary Conditions. As indicated above, we take our field variables—that is, $\boldsymbol{u}, p$, and $\boldsymbol{\sigma}$—to be in L^2. We therefore replace the boundary condition $\boldsymbol{u} = \boldsymbol{g}$ on $\partial\Omega_h$ with the condition $\widehat{\boldsymbol{u}} = \boldsymbol{g}$ on $\partial\Omega_h$. We discuss imposition of the zero-mean condition on the pressure below, in the context of stiffness matrix assembly.

Stokes Test Space Inner Product. The test space inner product is a crucial choice in DPG; it determines the energy norm, in which the method is optimal. Suppose we want a method optimal in the L^2 norm of our field variables; that is, we wish to minimize

$$\|\boldsymbol{u}-\boldsymbol{u}_h\|^2+\|\boldsymbol{\sigma}-\boldsymbol{\sigma}_h\|^2+\|p-p_h\|^2 .$$

As we have previously discussed in some depth [25], an appropriate choice for the test norm in this case is the *graph norm.* Because our focus in the present discussion is on execution, here we limit ourselves to the determination of the graph norm. Grouping equation (4) by the field variables, we have

$$\begin{aligned} b((u,\widehat{u}),v) = &(\boldsymbol{u},\nabla q+\nabla\cdot\boldsymbol{\tau})_{\Omega_h}+(-p,\nabla\cdot\boldsymbol{v})_{\Omega_h}+(\boldsymbol{\sigma},\nabla\boldsymbol{v}+\boldsymbol{\tau})_{\Omega_h}\\ &+\langle\text{boundary terms}\rangle\end{aligned}$$

The graph norm is then given by the Euclidean combination of the test terms for each field variable, plus L^2 norms[6] of each test variable:

$$\|(\boldsymbol{v},q,\boldsymbol{\tau})\|^2_{\text{graph}}=\|\nabla q+\nabla\cdot\boldsymbol{\tau}\|^2+\|\nabla\cdot\boldsymbol{v}\|^2+\|\nabla\boldsymbol{v}+\boldsymbol{\tau}\|^2+\|\boldsymbol{v}\|^2+\|q\|^2+\|\boldsymbol{\tau}\|^2 .$$

Optimal Test Function Determination. As discussed above, each trial space basis function e_i has corresponding to it an optimal test function v_{e_i}, where v_{e_i} solves the equation

$$(v_{e_i},\delta v)_V=b(e_i,\delta v)\quad\forall\delta v\in V,$$

where V is the (discrete) test space. Because we employ test functions that are allowed to be discontinuous at inter-element boundaries, this is an element-local problem. Note that because of the discretizations we have selected, the test space will have more degrees of freedom—if we take the test space to have m degrees of freedom per element and the trial space to have n degrees of freedom, then the left hand side of this system will be a square $m\times m$ matrix, and the right a rectangular $m\times n$ matrix (expanding the equation column-wise in the trial space index i), and the solution will be n vectors of length m: the m coefficients of each optimal test function. If our basis for the enriched test space consists of functions v_j and we define the $m\times m$ Gram matrix

$$G_{jk}=(v_j,v_k)_V$$

and the $n\times m$ bilinear form matrix

$$B_{ij}=b(e_i,v_j),$$

then the optimal test coefficients are the columns of $G^{-1}B^T$.

Stiffness Matrix Assembly. Once the optimal test functions for an element are determined, the element stiffness matrix K_{ij} is relatively simple to compute; we have

$$K_{ij}=b(e_i,v_{e_j})=(v_{e_i},v_{e_j})_V,$$

[6]The L^2 terms may be weighted by constants; the best choice of constants will depend on the problem. Often, we simply take unit weights.

by the very problem that we solved to determine the optimal test functions. Note that we can compute the inner product of an arbitrary pair of test basis functions $\alpha_i v_i$ and $\beta_j v_j$ by means of the Gram matrix:

$$(\alpha_i v_i, \beta_j v_j)_V = \boldsymbol{\alpha}^T (v_i, v_j)_V \boldsymbol{\beta} = \boldsymbol{\alpha}^T G \boldsymbol{\beta},$$

so that the element stiffness matrix is given by

$$(v_{e_i}, v_{e_j})_V = (G^{-1}B^T)^T G G^{-1} B^T = B G^{-1} B^T.$$

The element stiffness matrix entries can then be assembled into a global stiffness matrix. Note that the only inter-element coupling comes through the trace variables (the trace $\widehat{\boldsymbol{u}}$ and flux $\widehat{\boldsymbol{t}}_n$, in the case of our Stokes formulation).

To impose the zero-mean condition on the pressure, we follow a technique described by Bochev and Lehoucq [4], which preserves symmetric positive definiteness of the stiffness matrix and allows a simple implementation for nodal bases (which is what we use for field variables in Camellia).

Global Problem Solution. As noted above, the only entries in our stiffness matrix that are coupled between elements belong to trace variables. We can therefore employ static condensation to reduce the size of the global system. That is, the matrix can be reordered to take the form

$$\begin{pmatrix} K_{11} & K_{12} \\ K_{12}^T & K_{22} \end{pmatrix} \begin{pmatrix} u \\ t \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \end{pmatrix}$$

where K_{11} is block diagonal, u represents the degrees of freedom corresponding to field variables, and t those corresponding to the trace variables. Noting that $u = K_{11}^{-1}(F_1 - K_{12}t)$, we can substitute this into the equation $K_{12}^T u + K_{22} t = F_2$ to obtain an equation for the trace degrees of freedom:

$$(K_{22} - K_{12}^T K_{11}^{-1} K_{12}) t = F_2 - K_{12}^T K_{11}^{-1} F_1.$$

Since K_{11} is block diagonal, its inversion can be carried out element-wise and in parallel; since K_{12} is a significantly smaller matrix, the computational cost of the global solve is reduced. For simplicity, we have generally employed direct solvers for both the local and the global solves.

3.3. Riesz Representations and Adaptivity

DPG minimizes the error in the energy norm; that is, for a problem with exact solution u and discrete solution u_h,

$$\|u - u_h\|_E = \sup_{\|v\|_V = 1} b(u - u_h, v)$$

is minimized. Now, we may rewrite this as

$$\begin{aligned} \sup_{\|v\|_V = 1} b(u - u_h, v) &= \sup_{\|v\|_V = 1} \left(b(u, v) - b(u_h, v)\right) \\ &= \sup_{\|v\|_V = 1} \left(l(v) - b(u_h, v)\right) \\ &= \|l - Bu_h\|_{V'}, \end{aligned}$$

where the operator B is defined by $Bu = b(u, \cdot)$. The error $\|l - Bu_h\|_{V'}$ is computable by virtue of the Riesz representation theorem, which tells us that there exists a function $e \in V$ such that

$$(e, v)_V = l(v) - b(u_h, v) \quad \forall v \in V,$$

and moreover that $\|e\|_V = \|l - Bu_h\|_{V'}$. We call this e the *error representation function*, and compute it by defining a residual vector

$$r_i = l(v_i) - b(u_h, v_i),$$

then inverting the Gram matrix to solve for $e = G^{-1}r$. We can then compute

$$\|e\|_V^2 = (e, e)_V = e^T Ge = (G^{-1}r)^T Ge = r^T e.$$

Note that this computation is local to the element, making it susceptible to parallel execution. Once we have computed the error, any number of refinement strategies might be employed. Our basic strategy is a greedy one, determining the maximum element error $\|e_K\|_V$ in the mesh, and marking for refinement any element with error greater than $\theta \|e_K\|_V$, where $\theta \in (0, 1)$ is a threshold parameter, which in most of our computations to date we have taken to be 0.20. The refinements might be either h- or p-refinements.

When one element is more refined than its neighbor (in either h or p), we must decide what discretization to use for the trace variables along the interface between the elements. While other choices are possible,[7] our approach to date has been to adopt the finer element's discretization along the shared interface—the so-called *maximum rule*.

3.4. Navier-Stokes Formulation

DPG minimizes the residual of a linear problem. While more sophisticated approaches are possible, for the present when solving a nonlinear problem we first linearize the problem, then use DPG to solve the linearized problem. For example, we may write the steady incompressible Navier-Stokes equations as:

$$\begin{aligned} -\nabla p + \mu \nabla \cdot \boldsymbol{\sigma} &= \boldsymbol{f} + \boldsymbol{u} \cdot \nabla \boldsymbol{u}, \\ \boldsymbol{\sigma} - \nabla \boldsymbol{u} &= 0, \\ \nabla \cdot \boldsymbol{u} &= 0. \end{aligned}$$

Observing that the nonlinear, convective term may be written $\boldsymbol{u} \cdot \nabla \boldsymbol{u} = \boldsymbol{u} \cdot \boldsymbol{\sigma}$, we immediately see that a Navier-Stokes formulation corresponding to our VGP Stokes formulation is

$$\begin{aligned} \langle \widehat{\boldsymbol{t}}_{\boldsymbol{n}}, \boldsymbol{v} \rangle_{\Gamma_h} + (p, \nabla \cdot \boldsymbol{v})_{\Omega_h} + (\boldsymbol{\sigma}, \nabla(\mu \boldsymbol{v}))_{\Omega_h} - (\boldsymbol{u} \cdot \boldsymbol{\sigma}, \boldsymbol{v})_{\Omega_h} &= (\boldsymbol{f}, \boldsymbol{v})_{\Omega_h}, \\ (\boldsymbol{\sigma}, \boldsymbol{\tau})_{\Omega_h} - \langle \widehat{\boldsymbol{u}}, \boldsymbol{\tau n} \rangle_{\Gamma_h} + (\boldsymbol{u}, \nabla \cdot \boldsymbol{\tau})_{\Omega_h} &= 0, \\ \langle \widehat{\boldsymbol{u}} \cdot \boldsymbol{n}, q \rangle_{\Gamma_h} - (\boldsymbol{u}, \nabla q)_{\Omega_h} &= 0. \end{aligned}$$

[7] Indeed, Camellia's in-progress support for meshes of arbitrary spatial dimensions employs the minimum rule; for various reasons, we believe this to be the simpler choice, especially in three or more dimensions.

If we define the Stokes bilinear formulation as $b_{\text{Stokes}}(u,v) = l_{\text{Stokes}}(v)$, and linearize about $(\boldsymbol{u}+\Delta\boldsymbol{u}, \boldsymbol{\sigma}+\Delta\boldsymbol{\sigma}, p+\Delta p)$, we then have

$$b_{\text{Stokes}}(\Delta u, v) - (\Delta\boldsymbol{u}\cdot\boldsymbol{\sigma} + \boldsymbol{u}\cdot\Delta\boldsymbol{\sigma}, \boldsymbol{v})_{\Omega_h} = (\boldsymbol{f},\boldsymbol{v})_{\Omega_h} - b_{\text{Stokes}}(u,v) + (\boldsymbol{u}\cdot\boldsymbol{\sigma},\boldsymbol{v})_{\Omega_h}.$$

Our strategy for solving this is a standard Newton iteration: we start from some initial guess $u = u_0$, solve for the increment Δu, set $u_{i+1} = u_i + \Delta u$, and iterate until some measure of the increment is below a desired threshold. It is worth noting, however, that in general our test norm will now depend on the background flow[8] u_i.

3.4.1. *Stopping Criterion for Nonlinear Iteration*

Our stopping criterion for the Newton iteration in the context of adaptivity is as follows. We define an L^2 tolerance ϵ_0 for the initial mesh (that is, refinement 0). After we converge sufficiently that the L^2 norm of the field variables in the Newton update is below ϵ_0, we measure the energy norm of the solution, as well as the energy norm of the error, to determine a relative energy error e^0_{rel} for refinement $i = 0$, defined as:

$$e^i_{\text{rel}} = \frac{\|e^i\|_E}{\|u^i_h\|_E} = \frac{\|u - u^i_h\|_E}{\|u^i_h\|_E} = \frac{\|l - Bu^i_h\|_{V'}}{\|u^i_h\|_E}.$$

We then refine according to the greedy refinement strategy described above, and for refinement $i > 0$ we set the L^2 tolerance for our stopping criterion as

$$\epsilon_i = e^{i-1}_{\text{rel}}\epsilon_0.$$

The computed error on the adaptive mesh thus automatically guides the L^2 convergence that we attempt to achieve in the Newton iteration.

3.5. *Curvilinear Meshes*

When using curved geometries, the order of finite element solution convergence is generally limited by the order of approximation of the geometry. Thus to achieve higher-order convergence we require higher-order geometry representations. We choose to employ an isoparametric representation of the geometry, in which the basis functions used in finite element computations are also used to represent the geometry. Compared with representing the geometry exactly in our computations, this is an appealing choice for two reasons: first, the isoparametric representation will in general be cheaper to compute; second, isoparametric geometry allows the exact representation of *linearized rigid-body motion*, a fact of engineering interest. Our approach here essentially follows Demkowicz et al. [11, pp. 195-210].

Consider a connected domain $\Omega \subset \mathbb{R}^2$. We assume that the domain can be partitioned into curvilinear triangles and quadrilaterals. An example of a domain with such a partition is shown in Figure 1. In order to work with such geometries, we must determine a systematic mechanism for constructing maps $\boldsymbol{x}(\boldsymbol{t})$ from reference elements to physical space.

Now, typically an engineer or analyst will not have ready to hand a representation of the interior of the domain; rather, there will be some description of the boundary of the domain. In what follows we assume that this description

[8]That is, the previous solution.

is parametrically defined. This assumption does impose some constraints—in some cases, geometry will be known implicitly, perhaps defined by the intersection of two surfaces. For further details on implicitly defined geometry, see [11, pp. 198-199] and [16, pp. 96-97].

We proceed as follows. Since we assume a parametrization of the domain boundary, it is natural further to assume a parametrization of the edges on the domain boundary (interior edges might simply be taken to be straight lines). From the edge parametrizations it is possible to construct a *transfinite interpolant* [18], a mapping from a reference quadrilateral or triangle onto its curvilinear counterpart which is exact on the edges. The transfinite interpolant in hand—a representation of the exact geometry—we construct a *projection-based interpolant*, which approximates the geometry using precisely the polynomial basis employed in our finite element discretization.

Suppose that for an element K, for each edge $e_i \subset \partial K$ connecting vertices $\boldsymbol{v}_i$ and $\boldsymbol{v}_{i+1}$ there is some given parametrization $\boldsymbol{x}_i : [0,1] \to e_i$ onto the edge such that $\boldsymbol{x}_i(0) = \boldsymbol{v}_i$ and $\boldsymbol{x}_i(1) = \boldsymbol{v}_{i+1}$. (We implicitly use modular arithmetic in vertex numbering here and below—thus on a quadrilateral vertex 0 is identified with vertex 4, and similarly 0 and 3 are identified on the triangle.) The vertex and edge numbering for the reference quadrilateral are shown in Figure 2. We now construct transfinite interpolants for quadrilateral elements; an analogous construction exists for triangles; similar constructions are also possibly for three-dimensional topologies.

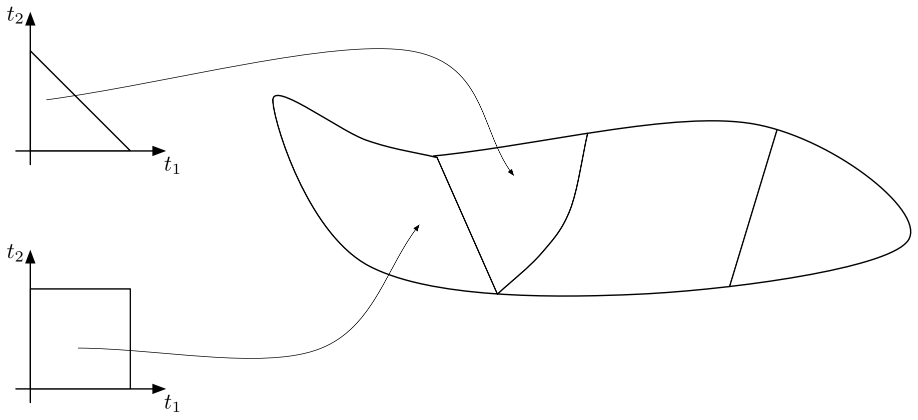


Figure 1: An example curvilinear domain partitioned into triangular and quadrilateral elements.

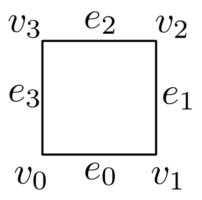


Figure 2: Vertex and edge numbering on reference quadrilateral.

Transfinite Interpolation on the Quadrilateral. We define the edge bubble functions[9] $\Delta \boldsymbol{x}_i$ by subtracting the vertex contributions from the edge parametrizations in an appropriately blended fashion:

$$\Delta \boldsymbol{x}_i \overset{\text{def}}{=} \boldsymbol{x}_i - (t-1)\,\boldsymbol{v}_i - t\,\boldsymbol{v}_{i+1}.$$

We define a bilinear interpolant $\boldsymbol{b}(t_1,t_2)$ determined by the vertices:

$$\boldsymbol{b}(t_1,t_2) \overset{\text{def}}{=} (1-t_1)\,(1-t_2)\,\boldsymbol{v}_0 + t_1\,(1-t_2)\,\boldsymbol{v}_1 + t_1\,t_2\,\boldsymbol{v}_2 + (1-t_1)\,t_2\,\boldsymbol{v}_3.$$

The transfinite interpolant is then given by

$$\begin{aligned}\boldsymbol{x}(t_1,t_2) \overset{\text{def}}{=} \boldsymbol{b}(t_1,t_2) &+ (1-t_2)\,\Delta x_0(t_1) + t_2\,\Delta x_2(t_1)\\ &+ (1-t_1)\,\Delta x_3(t_2) + t_1 \Delta x_1(t_2).\end{aligned}$$

It is clear from the construction that this agrees exactly with the curves $\boldsymbol{x}_i$ on the element boundary, and that it blends the $\boldsymbol{x}_i$ in a continuous fashion on the element interior.

$\boldsymbol{H}^1$ *projection-based interpolation.* Thus far, our representation of the geometry has been exact. For reasons suggested above, we would like to determine a polynomial approximation to the geometry which will also allow us to maintain convergence rates. The correct way to do so is known as projection-based interpolation [11, pp. 180-183]. In our finite element discretization, it is desirable that elements should be *compatible* in the sense that they agree exactly on the location of shared vertices (and vertices on the approximate boundary should coincide with the exact boundary) as well as on the approximation of shared edges. Further, we would like our approximate geometry to closely approximate both location (that is, value) and curvature (derivative) information. We thus arrive at a constrained projection problem, which we can divide into steps:

1. Interpolate the vertices (i.e. use the vertex locations to set weights for the vertex basis functions).
2. Project the exact transfinite interpolant bubble—that is, $\Delta \boldsymbol{x} \overset{\text{def}}{=} \boldsymbol{x}(t_1,t_2) - \boldsymbol{b}(t_1,t_2)$—into the discrete space of edge bubble functions. Call the sum of the weighted edge and vertex functions $\widetilde{\boldsymbol{x}}_{\text{edge}}$.
3. Project the difference of the function thus far and the transfinite interpolant—that is, $\Delta \boldsymbol{x} - \widetilde{\boldsymbol{x}}_{\text{edge}}$—into the discrete space of face bubble functions.

We have thus determined a function belonging to our finite element discretization (specifically, one that belongs to the vector $\boldsymbol{H}^1$ discrete space on the element) that interpolates the exact geometry at the vertices, agrees with its neighbors on the geometry representation along shared edges, and (subject to those constraints) minimizes the geometric error in the sense of the $\boldsymbol{H}^1$ norm.

Refinements. It remains to specify how refinements should be handled: specifically, should the edges interior to the refined element be curves defined as above by the transfinite interpolant, or will straight edges suffice? For standard conforming elements, the usual practice is to use curved edges on the interior to guarantee optimal convergence rates.

[9]A bubble function on a topological entity is defined to be a function that vanishes on that entity's boundary. Thus the edge bubbles vanish at the vertices, face bubbles vanish on edges, and so on.

We were unsure whether this would be necessary for DPG in the curvilinear geometry of immediate interest to us, so we simply experimented with using straight edges on element interiors—the computational advantage being that we reduce the number of elements for which we need to compute the curvilinear geometry, and the refinements are somewhat simpler to implement this way. It is worth noting that we do use the transfinite interpolant to compute the location of the new vertices. We have verified the approach and the code using a Poisson manufactured solution on a domain with an embedded circle, for which our solution matches the best approximation in the space. This is not a proof, and we do hope in the future to implement refinements that are curvilinear on their interiors so that we can study the relative costs and benefits of the two approaches.[10]

4. Camellia Step By Step

In this section, we describe how to implement a DPG solver using Camellia. We broadly follow the outline of Section 3, describing how to use Camellia to define and invoke the mathematical and computational apparatus specified there. As there, we begin with the steps required to solve the Stokes equations using DPG on a fixed mesh with straight edges, then consider adaptivity, nonlinear equations, and meshes with curvilinear boundaries.

The essential design goal for Camellia is to make DPG research and experimentation as simple as possible, without sacrificing too much by way of performance. Our ideal is to write code that expresses the DPG formulation—the variational form and the inner product on the test space—in a way that makes the mathematics transparent, and requires minimal overhead beyond this to specify and solve problems. We aim at excellent software design, following software engineering principles such as encapsulation and using tests and parameter checking to verify the code on an ongoing basis.

Key features of Camellia include:

- simple implementation of systems of arbitrary first-order PDEs,
- simple implementation of arbitrary test space inner products,
- distributed stiffness matrix determination,
- distributed solve (using MUMPS [1]),
- support for h-, p-, and hp-adaptivity,
- support for meshes made up of quads and triangles,
- support for meshes of arbitrary irregularity, and
- H(grad)-, $H(\text{div})$-, and $H(\text{curl})$-conforming basis functions.

Reference-counted pointers. Throughout the code, and in this section, we make liberal use of reference-counted pointers (RCPs, instances of the `Teuchos::RCP` template class), which allow us to create objects without much concern about reclaiming the memory allocated for them: when the last reference to the object goes out of scope, the memory will automatically be reclaimed. Because `Teuchos::RCP<`*`ClassName`*`>` can be a lot to type, in the code and in this section, we adopt a convention that *`ClassName`*`Ptr` will mean the same thing.

[10]It is worth noting in this context that for certain meshes and geometries, clearly refinements with straight-edged interiors will *not* suffice—for example, if a sufficiently thin, sufficiently curved shell element is refined, the straight edges on the interior of a refinement will intersect the curved edges on the element's boundary. Practically, one can usually work around this limitation simply by using a sufficiently fine initial mesh.

4.1. Building Blocks for DPG Formulations: Var, LinearTerm, Function, IP, BF

Here, we introduce a few key classes which provide building blocks to specify a DPG formulation.

`Var`. The `Var` class identifies a trial or test space variable. The `VarFactory` class manages creation of `Var`s, ensuring unique identifiers for each variable, which in turn are used in the ordering of corresponding degree of freedom coefficients in the `DofOrdering` class. The `Var` class also keeps track of a linear operator applied to the variables—the default is a value operator; first-order differential operators, operators involving unit normals, and operators to select a component of a vector function are also available. `Var`s also have *types*: field, trace, flux, and test types are defined. Each `Var` has a tensorial *rank*—0 for scalar variables, 1 for vector variables, and so on.

`Function`. The `Function` class defines a function that may vary in space, and may take values of arbitrary tensorial rank. Among other things, functions may be used to provide material data or forcing functions in the formulation of a PDE problem; they may also be used for post-processing and visualization output of computed solutions. Many `Function` subclasses are provided; here, we name a few representative examples. `PreviousSolutionFunction` takes as its argument a `LinearTerm` (see below), so that solution variables may be combined in an arbitrary fashion and used elsewhere. This is exactly the mechanism by which nonlinear problems are solved in Camellia. The subclass `hFunction` allows simple definition of functions that depend on the diameter of mesh cells. `MeshPolyOrderFunction` has as its value the polynomial order trial space on each cell of the mesh; this is useful for visualization of variable-order meshes.

`LinearTerm`. The `LinearTerm` class represents variational terms that are linear in either trial or test variables. As such, `LinearTerm`s are defined as the product of a `Function` and a `Var`. `LinearTerm`s have types and ranks determined by their component functions and variables. Operator overloading allows simple syntax for some commonly used patterns—if `f1, f2` are `FunctionPtr`s and `v1, v2` are `VarPtr`s referring to test variables, for example, each of the following expressions results in a corresponding `LinearTermPtr`:

- `f1 * v1`,
- `v1 / f1`,
- `f1 / f2 * v1`,
- `v1 + v2`,
- `f1 * v1 + f2 * v2`,
- `3 * v1 + v2 / 5`, and
- `- v1`.

Moreover, application of first-order differential operators to variables is just a matter of calling the appropriate method on the `VarPtr`s; e.g. `v1->grad()` means ∇v_1.

`IP`. The `IP` class represents an inner product; its primary purpose is to define the test space inner product. Its `addTerm()` method takes a `LinearTermPtr` as argument. The square of the norm generated by the inner product is the sum of the squares of its terms. For example, the following code will specify an inner product that induces the standard H^1 norm $\|v\|_{H^1}^2 \overset{\text{def}}{=} \|v\|^2 + \|\nabla v\|^2$.

```
IPPtr ip =  = Teuchos::rcp(new IP());
ip->addTerm(v);
ip->addTerm(v->grad());
```

BF. The `BF` class represents a bilinear form; its `addTerm()` method takes as arguments trial and test space `LinearTerms`. Each term is the L^2 inner product of its arguments, taken either on the interior of each element (for field variables) or along the element boundary (for fluxes and traces).

4.2. Stokes on a Fixed Mesh with Straight Edges

A classic test case for Stokes flow is the lid-driven cavity flow problem. Consider a square cavity with an incompressible, viscous fluid, with a lid that moves at a constant rate. The resulting flow will be vorticular; as sketched in Figure 3, there will also be so-called *Moffatt eddies* at the corners; in fact, the exact solution will have an infinite number of such eddies, visible at progressively finer scales [23]. Note that the problem as described will have a discontinuity in the fluid velocity at the top corners, and hence its solution will not conform to the spaces we use in our analysis; for this reason, in our experiment we approximate the problem by introducing a thin ramp in the boundary conditions—we have chosen a ramp of width $\frac{1}{64}$. This makes the boundary conditions continuous,[11] so that the solution conforms to the spaces used in the analysis. We will use the cavity flow problem below as a motivating

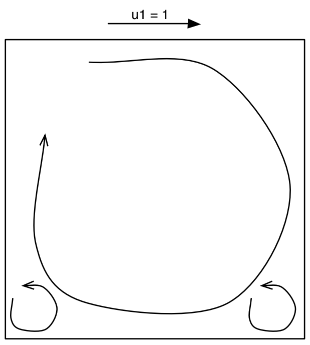


Figure 3: Sketch of lid-driven cavity flow.

example.

4.2.1. Stokes Variational Formulation

Recall the VGP formulation we defined for Stokes in Section 3:

$$\begin{aligned} b((u,\widehat{u}),v) = (\boldsymbol{\sigma} - p\boldsymbol{I}, \nabla \boldsymbol{v})_{\Omega_h} - \left\langle \widehat{\boldsymbol{t}}_n, \boldsymbol{v} \right\rangle_{\Gamma_h} \\ + (\boldsymbol{u}, \nabla q)_{\Omega_h} - \langle \widehat{\boldsymbol{u}} \cdot \boldsymbol{n}, q \rangle_{\Gamma_h} \\ + (\boldsymbol{\sigma}, \boldsymbol{\tau})_{\Omega_h} + (\boldsymbol{u}, \nabla \cdot \boldsymbol{\tau})_{\Omega_h} - \langle \widehat{\boldsymbol{u}}, \boldsymbol{\tau} \boldsymbol{n} \rangle_{\Gamma_h} = (\boldsymbol{f}, \boldsymbol{v})_{\Omega_h} = l(v). \end{aligned} \tag{7}$$

[11] It is worth noting that these boundary conditions are not exactly representable by many of the coarser meshes used in our experiments. Camellia interpolates the boundary conditions in the discrete space—at present it does so via an L^2 projection, though optional support for H^1 projection-based interpolation is also planned.

We begin by identifying the variables involved; we have scalar field variables u_1, u_2, p, vector fields $\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2$ (the columns of tensor $\boldsymbol{\sigma}$), traces $\widehat{u}_1, \widehat{u}_2$, and fluxes $\widehat{t}_{1n}, \widehat{t}_{2n}$. The test functions are $v_1, v_2, q \in H^1$ and $\boldsymbol{\tau}_1, \boldsymbol{\tau}_2 \in H(\text{div})$. We declare an instance of the `VarFactory` class, which keeps track of the trial and test space variables, and use this to create `VarPtrs` corresponding to each of these variables:

```
VarFactory varFactory;
// traces:
VarPtr u1hat = varFactory.traceVar("\\widehat{u}_1");
VarPtr u2hat = varFactory.traceVar("\\widehat{u}_2");
VarPtr t1_n = varFactory.fluxVar("\\widehat{t}_{1n}");
VarPtr t2_n = varFactory.fluxVar("\\widehat{t}_{2n}");
// fields:
VarPtr u1 = varFactory.fieldVar("u_1", L2);
VarPtr u2 = varFactory.fieldVar("u_2", L2);
VarPtr sigma1 = varFactory.fieldVar("\\sigma_1", VECTOR_L2);
VarPtr sigma2 = varFactory.fieldVar("\\sigma_2", VECTOR_L2);
VarPtr p = varFactory.fieldVar("p");
// test functions:
VarPtr tau1 = varFactory.testVar("\\tau_1", HDIV);  // tau_1
VarPtr tau2 = varFactory.testVar("\\tau_2", HDIV);  // tau_2
VarPtr v1 = varFactory.testVar("v_1", HGRAD);       // v_1
VarPtr v2 = varFactory.testVar("v_2", HGRAD);       // v_2
VarPtr q = varFactory.testVar("q", HGRAD);          // q
```

The strings in each variable definition are human- and/or TeX-readable identifiers for each variable, which can be used in debugging output, or to generate, say, TeX tabulations of numerical results. The `Var` class stores a unique ID for each variable, defines the function space for the variable, and allows various operators to be applied to the variable (first-order differential operators and operators involving the unit normal along the element boundary).

We rewrite (7) in terms of the variables thus defined, as:

$$\begin{aligned} b(u,v) = {} & (\boldsymbol{\sigma}_1, \nabla v_1)_{\Omega_h} - \left(p, \frac{\partial}{\partial x} v_1\right)_{\Omega_h} - \left\langle \widehat{t}_{1n}, v_1 \right\rangle_{\Gamma_h} \\ & + (\boldsymbol{\sigma}_2, \nabla v_2)_{\Omega_h} - \left(p, \frac{\partial}{\partial y} v_2\right)_{\Omega_h} - \left\langle \widehat{t}_{2n}, v_2 \right\rangle_{\Gamma_h} \\ & + \left(u_1, \frac{\partial}{\partial x} q\right)_{\Omega_h} + \left(u_2, \frac{\partial}{\partial y} q\right)_{\Omega_h} - \left\langle \begin{pmatrix} \widehat{u}_1 \\ \widehat{u}_2 \end{pmatrix} \cdot \boldsymbol{n}, q \right\rangle_{\Gamma_h} \\ & + (\boldsymbol{\sigma}_1, \boldsymbol{\tau}_1)_{\Omega_h} + (u_1, \nabla \cdot \boldsymbol{\tau}_1)_{\Omega_h} - \langle \widehat{u}_1, \boldsymbol{\tau}_1 \cdot \boldsymbol{n} \rangle_{\Gamma_h} \\ & + (\boldsymbol{\sigma}_2, \boldsymbol{\tau}_2)_{\Omega_h} + (u_2, \nabla \cdot \boldsymbol{\tau}_2)_{\Omega_h} - \langle \widehat{u}_2, \boldsymbol{\tau}_2 \cdot \boldsymbol{n} \rangle_{\Gamma_h} . \end{aligned} \tag{8}$$

Next, we use `varFactory` to create a bilinear form; that is, a `BF` object:

```
BFPtr stokesBF = Teuchos::rcp( new BF(varFactory) );
```

We then use `BF`'s `addTerm` method to add each pair of trial and test terms in (8)—thus, for example, the term $(\boldsymbol{\sigma}_1, \nabla v_1)_{\Omega_h}$ may be specified by calling `addTerm(sigma1, v1->grad())`. The complete bilinear form is thus specified in the following:

```
// v1:
stokesBF->addTerm(sigma1, v1->grad()); // $(\boldsymbol{\sigma}_1, \nabla v_1) - (p, \frac{\partial}{\partial x} v_1) + \langle \hat{t}_{1n}, v_1 \rangle$
stokesBF->addTerm( - p, v1->dx() );
stokesBF->addTerm( t1_n, v1);
```

```
// v2:
stokesBF->addTerm(sigma2, v2->grad()); // $(\boldsymbol{\sigma}_2, \nabla v_2) - (p, \frac{\partial}{\partial y} v_2) + \langle \hat{t}_{2n}, v_2 \rangle$
stokesBF->addTerm( - p, v2->dy());
stokesBF->addTerm( t2_n, v2);

// q:
stokesBF->addTerm(-u1,q->dx()); // $(-u_1, \frac{\partial}{\partial x} q) + (-u_2, \frac{\partial}{\partial y} q) + \langle \hat{u}_1 n_x + \hat{u}_2 n_y, q \rangle$
stokesBF->addTerm(-u2,q->dy());
FunctionPtr n = Function::normal();
stokesBF->addTerm(u1hat * n->x() + u2hat * n->y(), q);

// tau1:
stokesBF->addTerm(sigma1, tau1); // $(\boldsymbol{\sigma}_1, \boldsymbol{\tau}_1) + (u_1, \nabla \cdot \boldsymbol{\tau}_1) - \langle \hat{u}_1, \boldsymbol{\tau}_1 \cdot \boldsymbol{n} \rangle$
stokesBF->addTerm(u1, tau1->div());
stokesBF->addTerm(-u1hat, tau1->dot_normal());

// tau2:
stokesBF->addTerm(sigma2, tau2); // $(\boldsymbol{\sigma}_2, \boldsymbol{\tau}_2) + (u_2, \nabla \cdot \boldsymbol{\tau}_2) - \langle \hat{u}_2, \boldsymbol{\tau}_2 \cdot \boldsymbol{n} \rangle$
stokesBF->addTerm(u2, tau2->div());
stokesBF->addTerm(-u2hat, tau2->dot_normal());
```

Thus, in 32 lines of code, we have specified the entire bilinear form. By way of contrast, an earlier version of Camellia—prior to the introduction of `LinearTerm`—required over 300 lines of code for the same purpose.

We also need to specify the right-hand side for our variational form. This is managed by the `RHS` class; right-hand sides are linear functionals on the test space, so these can be specified as the sum of linear terms. In the case of Stokes cavity flow, the right-hand side is zero, but the general form for specifying the right-hand side is as follows:

```
FunctionPtr zero = Function::zero();
RHSPtr rhs = RHS::rhs();
rhs->addTerm( zero * v1 + zero * v2 );
```

4.2.2. *Stokes Discretization*

Camellia's discretizations are handled by the `Mesh` class;[12] one has simply to define the "H^1 order" in the trial space, given by $k+1$ according to the notation in Section 3, and the Δk enrichment for the test space, and Camellia will determine the appropriate orders for each variable.[13]

Our philosophy in DPG is to start with a coarse mesh, and allow adaptivity to refine in regions of largest error. Here, we create a 2×2 initial quadrilateral mesh on a unit square with quadratic field variables and quintic test functions using a convenience constructor provided by `MeshFactory`:

```
int k = 2;            // poly order for field variables
int H1Order = k + 1;  // L^2 order plus 1
int delta_k = 2;      // test space enrichment
double width = 1.0, height = 1.0;
int horizontalCells = 2, verticalCells = 2;
```

[12]It is perhaps worth noting that the `Mesh` class defines the entire discretization: the geometry, element boundaries, *and* the discrete trial and test spaces on each element. This is the reason why the `stokesBF` argument is required here. This is something we hope to change in a redesign, as there are many situations in which it is desirable to solve multiple problems on a single mesh, which the present design makes inconvenient. In lid-driven cavity flow, for example, we solve for streamlines using the solution to the `stokesBF` variational form as data.

[13]There are mechanisms for overriding the default choices if, for instance, one desired order $k+1$ for the velocity field variable and k for the pressure.

```
  MeshPtr mesh = MeshFactory::quadMesh(stokesBF, H1Order, delta_k, width, height,
                                       horizontalCells, verticalCells);
```

The `mesh` object, on construction, selects appropriate discrete bases according to the functional spaces used in the bilinear form. By default, the basis functions used are nodal bases (with spectrally distributed nodes) defined by Intrepid. Camellia also provides hierarchical $H(\text{div})$ and H^1 bases derived from the Lobatto (that is, the integrated Legendre) polynomials which may be used for test space discretization on quadrilateral elements. These can be selected by calling the following methods early on[14] in the execution of the driver.

```
  BasisFactory::setUseLobattoForQuadHDiv(true);
  BasisFactory::setUseLobattoForQuadHGrad(true);
```

4.2.3. Stokes Boundary Conditions

Camellia defines a `BC` class to allow specification of boundary conditions as well as zero-mean constraints. Let us specify boundary conditions for the lid-driven cavity flow problem described at the beginning of this section. Recall that our implementation of the problem approximates the boundary conditions to make them continuous, introducing a “ramp” of width $\epsilon = \frac{1}{64}$. We require:

- along the top boundary, $u_1 = \begin{cases} \frac{x}{\epsilon} & x \le \epsilon, \\ 1 & \epsilon < x < 1-\epsilon, \\ \frac{1-x}{\epsilon} & 1-\epsilon \le x, \end{cases}$
- along the left, right, and bottom walls of the cavity, $u_1 = 0$, and
- along the entire boundary, $u_2 = 0$.

Furthermore, because the pressure p only enters the formulation through a gradient, it will only be determined up to a constant. To fix its value, we impose a zero-mean constraint, $\int_\Omega p = 0$.

To specify the velocity BCs, we begin by implementing functions defined on the boundary. Camellia provides a `SimpleFunction` class, which is itself a subclass of `Function`: both classes define spatially varying functions; `SimpleFunction` presents a simpler interface that assumes the function is scalar and depends *only* on spatial coordinates. Below, we define a subclass of `SimpleFunction` for the BCs on u_1 along the top boundary.

```
class RampBoundaryFunction_U1 : public SimpleFunction {
  double _eps; // ramp width
public:
  RampBoundaryFunction_U1(double eps) {
    _eps = eps;
  }
  double value(double x, double y) {
    if ( (abs(x) < _eps) ) { // top left
      return x / _eps;
    } else if ( abs(1.0-x) < _eps) { // top right
      return (1.0-x) / _eps;
    } else { // top middle
      return 1;
    }
  }
};
```

[14]The methods here invoked are static, and the `BasisFactory` class itself is also statically defined, which is why early invocation is important: the invocation should occur before the static `BasisFactory` class is otherwise used. We hope to make this more flexible in a future version of Camellia.

Next, we need a mechanism to specify the portion of the boundary along which each BC is imposed—for example, some BCs might be imposed along an inflow boundary, while others are imposed only along the outflow boundary. Camellia allows the boundary to be restricted by means of a `SpatialFilter` subclass. In the case of lid-driven cavity flow, we want to impose one set of BCs along the top boundary (the lid), and another set along all other boundaries (the fixed walls). Below, we define a `SpatialFilter` that matches the top boundary. Camellia's `SpatialFilter::negatedFilter()` method provides a mechanism for taking the complement of a defined `SpatialFilter`; we use this to define the wall boundaries.

```
class TopBoundary : public SpatialFilter {
public:
  bool matchesPoint(double x, double y) {
    double tol = 1e-14;
    return (abs(y-1.0) < tol);
  }
};
```

Now, in the main body of the code, we need to create instances of our `Function` and `SpatialFilter` subclasses, as well as a new `BC` object. We then add Dirichlet conditions for $\widehat{u}_1$ and $\widehat{u}_2$, as well as the zero-mean constraint on p. Note that the boundary condition for $\widehat{u}_2$ is defined using `Function::zero()`, a statically defined `FunctionPtr` subclass; there are several such functions available in `Function`, including constants, select trigonometric functions, mechanisms for "vectorizing" other functions, and more.

```
  BCPtr bc = BC::bc();
  SpatialFilterPtr topBoundary = Teuchos::rcp( new TopBoundary );
  SpatialFilterPtr wallBoundaries = SpatialFilter::negatedFilter(topBoundary);

  // top boundary:
  FunctionPtr u1_bc_fxn = Teuchos::rcp( new RampBoundaryFunction_U1(1.0/64) );
  FunctionPtr zero = Function::zero();
  bc->addDirichlet(u1hat, topBoundary, u1_bc_fxn);
  bc->addDirichlet(u2hat, topBoundary, zero);

  // everywhere else:
  bc->addDirichlet(u1hat, wallBoundaries, zero);
  bc->addDirichlet(u2hat, wallBoundaries, zero);

  bc->addZeroMeanConstraint(p);
```

4.2.4. Stokes Test Space Inner Product

In Section 3, we demonstrated a procedure for determining the graph norm from a bilinear form. Camellia uses this very procedure to determine the graph norm automatically, requiring just a single line of code:

```
  IPPtr ip = bf->graphNorm();
```

Suppose another norm is desired—for example, elsewhere we have discussed a weighted graph norm motivated by a scaling argument [24], given by

$$\begin{aligned}\|(\boldsymbol{v},q,\boldsymbol{\tau})\|_V^2 &:= \|\nabla\cdot\boldsymbol{v}\|^2 + \|\nabla\boldsymbol{v}+\boldsymbol{\tau}\|^2 \\ &\quad + \|h\nabla\cdot\boldsymbol{\tau} - h\nabla q\|^2 + \left\|\frac{\boldsymbol{v}}{h}\right\|^2 + \|q\|^2 + \|\boldsymbol{\tau}\|^2 ,\end{aligned}$$

where h is the local element diameter. Camellia's `IP` class allows definition of an inner product in terms of linear terms—by calling `addTerm()` with a `LinearTermPtr` argument, one can indicate that the square of the desired norm should include as summand the square of the specified term. Thus, one can specify the weighted graph norm in Camellia as follows:

```
FunctionPtr h = Function::h(); // element diameter
IPPtr ip = Teuchos::rcp( new IP ); // create inner product

ip->addTerm( v1->dx() + v2->dy() );           // pressure
ip->addTerm( v1->grad() + tau1 );             // sigma1
ip->addTerm( v2->grad() + tau2 );             // sigma2
ip->addTerm( h * tau1->div() - h * q->dx() ); // u1
ip->addTerm( h * tau2->div() - h * q->dy());  // u2

// L^2 terms:
ip->addTerm( v1 / h );
ip->addTerm( v2 / h );
ip->addTerm( q );
ip->addTerm( tau1 );
ip->addTerm( tau2 );
```

4.3. Solving

The effort of determining optimal test functions, assembling the global stiffness matrix, and solving for the DPG degrees of freedom is encapsulated in the `Solution` class. Moreover, the optimal test function determination and stiffness matrix assembly—both of which are embarrassingly parallel operations—are performed in a distributed fashion, using a spatially-local mesh partitioning provided by the `Mesh` class (the `Mesh` class in turn uses Trilinos's Zoltan package for constructing the partitioning). One can construct a `Solution` object and solve as follows:[15]

```
SolutionPtr solution = Solution::solution(mesh, bc, rhs, ip);
solution->condensedSolve();
```

After the solve is complete, there are mechanisms for constructing `Function` objects based on the `Solution`, which can be used for further computations. There are also mechanisms, implemented by Truman Ellis, for outputting the solution to a VTK file for visualization.

If direct access to the assembled stiffness matrix is desired (one might, e.g., wish to examine its conditioning), this is possible via `Solution`'s getStiffnessMatrix() method, which returns a pointer to an `Epetra_FECrsMatrix` object.

The standard solvers supported by Camellia are direct solvers; both the KLU serial direct solver [10] and MUMPS parallel direct solver [1] are supported; when building Camellia with MPI support, MUMPS is the default solver. However, if another solution strategy is desirable (e.g. an iterative solution strategy), it is a relatively simple matter to specify this. One simply subclasses `Solver`, overriding its `solve()` method. The abstract superclass provides a member variable, `_problem`, which is a pointer to an `Epetra_LinearProblem`. The latter provides access to the

[15]One can solve either with or without static condensation; generally, static condensation is recommended. In addition to being faster, the condensed stiffness matrix generally is better conditioned. However, Lagrange constraints, which are supported by the standard solve (and which are useful for, say, enforcing local conservation), are not yet supported by the static condensation solve.

containers for the matrix, the right-hand side, and the solution containers for the system to be solved. One may then provide a pointer to a `Solver` as an argument to `Solution`'s `condensedSolve()` or `solve()` to use that as the solver in place of the standard direct solvers.

4.4. Adaptivity

Camellia provides a simple mechanism for h-, p-, and hp-adaptivity using a greedy refinement strategy through the `RefinementStrategy` class. Given a `SolutionPtr` object `solution`, one may define a greedy refinement strategy by specifying

```
double threshold = 0.20;
RefinementStrategy refStrategy( solution, threshold );
```

where the value of 0.20 indicates that elements with error greater than 20% of the maximum element error will be refined in a given refinement step. The default implementation performs h-refinements. One may implement other refinement types by subclassing `RefinementStrategy` and overriding the method

```
virtual void refineCells(vector<int> &cellIDs);
```

The superclass then handles the determination of which cells should be refined, and the subclass simply needs to determine what sorts of refinements should be performed on a per-cell basis. Once a refinement strategy is in hand and a solve is completed, a refinement step can be accomplished in a single line of code:

```
refStrategy.refine();
```

4.5. Riesz Representations

As we saw in Section 3, Riesz representation functions play a crucial role in DPG; an optimal test function, for example, is precisely the Riesz representation of a bilinear form in which the trial space argument has been fixed, understood as a functional on the test space. Similarly, the error representation function used to determine the energy error and drive adaptivity is the Riesz representation of the residual. Recognizing that the `LinearTerm` class defines functionals on the test and trial spaces, Camellia allows Riesz representations to be defined in terms of `LinearTermPtr` objects, together with an `IPPtr` object representing the inner product relative to which the Riesz representation should be taken, and a `MeshPtr` object corresponding to the mesh on which the representation should be computed. Riesz representations are vector-valued, with components corresponding to each variable in the test (or trial) space. Camellia provides a `Function` subclass, `RepFunction`, which allows one to perform arbitrary computations with the component of the Riesz representation corresponding to a variable in the test (trial) space. The code below demonstrates construction of a such a function corresponding to the `var` component of a functional `linearTerm`, using the `RieszRep` class, which was contributed to Camellia by Jesse Chan.

```
RieszRepPtr rieszRep = RieszRep::rieszRep( mesh, ip, linearTerm );
FunctionPtr repFxn   = RieszRep::repFunction( var, rieszRep );
```

4.6. *Nonlinear Problems*

For the Navier-Stokes equations, we will employ a bilinear form based on the linearized Navier-Stokes equations. This will then be solved by means of a Newton iteration, using the L^2 norm of field variables in the incremental solution as a stopping criterion. Thus we require mechanisms for:

- representing previous solutions as material data,
- evaluating L^2 norms of the incremental solution, and
- summing the previous solution and the incremental solution.

Camellia provides convenient mechanisms for each of these. The first two come by way of the `Function::solution(VarPtr var, SolutionPtr soln)` method, which returns a `FunctionPtr` representing the `var` component of solution `soln`. The third is handled by `Solution::addSolution(SolutionPtr soln)`. Each of these is demonstrated in the code snippets below, which assume that the `SolutionPtr` object `prevSoln` represents the background flow (that is, the accumulated solution) and `incrSoln` represents the incremental solution.

```
// ...
// define previous solution functions for x-velocity u1, y-velocity u2:
FunctionPtr u1_prev = Function::solution(u1, prevSoln);
FunctionPtr u2_prev = Function::solution(u2, prevSoln);
// combine these into a single vector-valued function:
FunctionPtr u_prev = Function::vectorize(u1_prev, u2_prev);
// add convective terms with u_prev to bilinear form:
navierStokesForm->addTerm( - Re * u_prev * sigma1, v1);
navierStokesForm->addTerm( - Re * u_prev * sigma2, v2);
// ...
incrSoln->solve();
// add incremental solution to prevSoln
prevSoln->addSolution(incrSoln);
// check whether we're done:
FunctionPtr u1_incr = Function::solution(u1, incrSoln);
// ...
FunctionPtr fiedsSquared = u1_incr * u1_incr + u2_incr * u2_incr + p_incr * p_incr
                         + sigma1_incr * sigma1_incr + sigma2_incr * sigma2_incr;
double l2incr = fieldsSquared->integrate(incrSoln->mesh());
if (l2incr < 1e-8) {
  break;
}
// ...
```

4.7. *Curvilinear Meshes*

As discussed in Section 3, when using curved geometries, the order of finite element solution convergence is generally limited by the order of approximation of the geometry. Thus to achieve higher-order convergence we require higher-order geometry representations. Several requirements present themselves:

1. Provide a mechanism for *specifying* curvilinear geometry. Because we may vary the polynomial order, the best way to approach this is by allowing exact specification of the geometry, rather than any particular polynomial approximation thereof.
2. Internally, compute a polynomial approximation of the geometry.
3. Internally, account for the curved mesh geometry by changing Jacobians to account for the polynomial geometry approximation, as well as recomputing geometry on mesh refinement.

At present, Camellia supports curvilinear quadrilateral elements. We hope to support triangles in a similar fashion.

4.8. *Geometry Specification*

Most elements in the mesh will not require curved edges; only those that are on the boundary may need curved edges. For this reason, we first define a straight-edge mesh, then allow any edge to be replaced by a parametrically specified curve $\boldsymbol{f}(t)$. The mesh requires that $\boldsymbol{f}(0)$ interpolates the first vertex in the edge and $\boldsymbol{f}(1)$ interpolates the second. Camellia provides several mechanisms for specifying parametric functions—the core class is the `ParametricCurve` class, instances of which can be defined either through subclassing or by specifying the x and y components as `FunctionPtr`s, in which the x variable is taken as the parametric argument. For example, to specify a unit circle centered at the origin, one might write:

```
FunctionPtr cos_2pi_t = Teuchos::rcp( new Cos_ax(2.0*PI) );
FunctionPtr sin_2pi_t = Teuchos::rcp( new Sin_ax(2.0*PI) );
ParametricCurvePtr circle = ParametricCurve::curve(cos_2pi_t, sin_2pi_t);
```

`ParametricCurve` allows definition a parametric curve as a segment of another via its `subCurve` method; for example, to define the quarter circle in the first quadrant of the plane, one could write:

```
ParametricCurvePtr arc = ParametricCurve::subCurve(circle, 0, 0.25);
```

`ParametricCurve` also provides built-in definitions of line segments, circles, and circular arcs.

Once the parametric curves are defined, one only has to associate these with edges in the mesh. Suppose that one has a mesh whose cell 0's first edge goes from (1,0) to (0,1), and that the curvilinear mesh desired is this mesh with the that edge replaced by the arc defined above. Executing that replacement is accomplished by the following code:

```
int cellID = 0;
vector<int> vertices = mesh->vertexIndicesForCell(cellID);
pair<int,int> edge = make_pair(vertices[0],vertices[1]);
map< pair<int,int>, ParametricCurvePtr > edgeToCurveMap;
edgeToCurveMap[edge] = arc;
mesh->setEdgeToCurveMap(edgeToCurveMap);
```

4.9. *Accounting for Curved Geometry in Computations*

Transparent to the user writing a driver using Camellia, there are several choices and mechanisms relating to the use of curved geometry that we relate here. Because of the transparency to the user—refinements for curved elements, for example, are invoked in precisely the same way as for straight-edged ones—in this subsection, we omit any sample code.

4.9.1. *Geometry Approximation*

We use the specified curves to compute a two-dimensional *transfinite interpolant* which exactly matches the edges.

We take an isoparametric approach to geometry approximation, in that we use exactly our $\boldsymbol{H}^1$ basis to represent the transformation from the straight-edge mesh to the curvilinear one. This transformation is determined using *projection-based interpolation* of the transfinite interpolant.

In order to avoid limiting the accuracy of the approximation of the test functions, we use the test degree as the order of our geometric approximation.

4.9.2. Jacobians

We already compute a Jacobian, an inverse Jacobian, and a Jacobian determinant for the straight-edge mesh. To compute the Jacobian from reference to physical space, we simply need to multiply the existing Jacobian by the Jacobian of our transformation from straight-edge to physical space. The resulting Jacobian (and its inverse and determinant) can then replace the existing one in all computations.

4.9.3. Refinements

When an element with a curved edge is refined, we need to recreate the curve. Because of the parametric nature of the curve, this only requires remapping the input—and this can be done algebraically, without any need for recursion. (We do also have to compute the new vertex according to the exact geometry.) We can then use the existing apparatus to recompute the transformation function for the newly created elements.

At present, all newly introduced edges are straight lines with the newly introduced vertices corresponding to the exact geometry. This allows us to minimize the computational cost associated with curved geometry (by keeping the number of elements with curved edges to a minimum); however, we may get better results by instead introducing curved edges corresponding to the transfinite interpolant. This will require defining "patches" of the transfinite interpolant, which is just the two-dimensional analog to subdivisions of parametric curves; by similar logic, it should be possible to define these without resorting to a recursive strategy.

As mentioned in Section 3, the evidence to date suggests that our convergence rates do not suffer due to these straight-edged-interior refinements, and Camellia can take advantage of some savings in computation thanks to this choice—and the amount of savings increases as the mesh is refined: elements with curves will lie only along the curved boundaries. However, a full exploration of the costs and benefits of our approach to refinements—as well as support for more general geometries, including highly curved elements—will require implementation of the more standard approach (refining with curved edges in the interior); we hope to add such a feature to Camellia in the future.

5. Selected Implementation Details

In the above, we have focused on how Camellia can be used, omitting discussion of how it works. Here, we give a few details regarding our implementation—our aim is not a complete specification, but to provide enough detail to give a sense for what we have done. Recall from the above that our essential building blocks are *variables* (represented by the `Var` class) and *functions* (`Function`). A function multiplied by a variable gives a *linear term* (`LinearTerm`, which may be interpreted as a functional on the space of test or trial functions). By summing trial and test space linear terms paired by the L^2 inner product, a *bilinear form* may be constructed. By summing symmetric pairs of linear terms on the test space, an *inner product* and corresponding norm may be specified.

Therefore, in this section, we focus on the implementation of the `Function` class. The basic idea of the class is to allow representation of any functions that might be useful in the course of finite element computations—we therefore support functions that depend only on spatial coordinates, as well as those that depend on the discretization; similarly, we support both functions that are defined on element interiors and those that are only defined on element boundaries (such as element outward-facing normals). We support functions of arbitrary rank, though most support for higher-ranked functions does assume a particular shape and length for their values—e.g., in 3D the values of a vector-valued function are in places assumed to be of length 3, and a matrix-valued function is assumed to have 3×3 matrices as its values.

One design choice for the `Function` class would be to implement a `value()` method that takes as arguments the spatial coordinates of a point of interest, and any other data that might be required (regarding the mesh, for example), and returns data for the function's value at that point. While we do adopt such a design for some of our subclasses (more on this in a moment), we do not do so for the base class for two reasons. First, this design will incur some extra method dispatching costs—the `value()` method will be called once per quadrature point during integration, for example; if we instead adopt a design that allows values for multiple points to be computed within a single method, we can amortize the dispatch costs over many points. Second, because of the generality we aim for—e.g., function values might or might not depend on the mesh, and they might be defined on a domain of any number of spatial dimensions—explicitly providing arguments for any and all data that might be of interest to subclasses will unduly clutter the interface. We therefore adopt as the core method that subclasses need to override a `values()` method, with the following signature:

```
void values(FieldContainer<double> &fxnValues, BasisCachePtr basisCache);
```

The subclass is responsible for filling in the multi-dimensional array `fxnValues` with values that are appropriate according to the data contained in `basisCache`. `BasisCache` defines the computational context broadly—so, depending on the computational context, it may have a list of cell IDs and a pointer to the mesh, as well as multi-dimensional arrays containing the spatial points of interest. It is the caller's responsibility to ensure that the `basisCache` provided has the information required by the `Function` subclass. As its name suggests, `BasisCache` also caches previously computed basis values, which allows some `Function` instances to compute their values more quickly—for example, `PreviousSolutionFunction` uses computed solution coefficients to determine solution values, and this requires summing over various basis functions.

It is worth noting that the `values()` method in the base class involves a bit more effort to override than would the single-point `value()` method proposed above—among other things, an implementation of `values()` will generally require a loop over the points of interest. To keep things simple for cases where the costs of method dispatch are not of concern and where functions are scalar-valued and depend only on spatial coordinates, we offer a `SimpleFunction` subclass that provides abstract `value()` methods for one, two, and three space dimensions—by overriding the appropriate method, a `Function` subclass can be implemented with minimal code.

As described above, Camellia provides a number of useful `Function` subclasses, many of which are available through static methods of the `Function` class. Among these are subclasses representing sums, products, and quotients of functions; Camellia also provides operator overloads so that if `a` and `b` are both `FunctionPtr` objects, then `a * b`, for example, results in a `FunctionPtr` representing their product. Camellia also optionally supports differential operators applied to functions, of which subclasses can take advantage by overriding `dx()`, `dy()`, and `dz()` methods, as appropriate; when the operands to products, quotients, and sums override these methods, then the products, quotients and sums also support the differential operators. It is worth noting that, as these are meant primarily as a convenience, we do not at present go to any great lengths to optimize the functions that result from such operations—we do not, for example, manipulate the order in which products and sums are evaluated in an effort to reduce computational costs. Such things are possible, but for the present our recommendation is that users concerned about such costs should implement their own `Function` subclasses to represent precisely the functions of interest to them, whenever Camellia's provided implementation is unsatisfactory. For simple function operations, however, Camellia's implementation is likely to be efficient enough that any differences in cost will not be noticeable in the overall execution time of applications.

Taken together, the facilities provided by `Function` and its subclasses in Camellia make for painless implementations of manufactured solutions. One simply has to define the exact solution for the primary variables (that is, those that appear in the strong form of the equations) in terms of `Function` subclasses that support differential operators, and then define both the right-hand side and the exact solution for secondary variables (that is, those introduced when rewriting in first-order form) in terms of the functions for the primary variables. One may then use the right-hand side thus determined to derive a discrete solution on any mesh of interest, and compare the computed solution to the exact one by means of the `Function::l2norm()` method, for example. The code listing below demonstrates the use of these facilities to derive a right-hand side corresponding to the classical Kovasznay flow solution for Navier-Stokes [21].

```
    // define Kovasznay flow solution
    const double PI  = 3.14159265358979323846;
    double lambda = Re / 2 - sqrt ( (Re / 2) * (Re / 2) + (2 * PI) * (2 * PI) );

    FunctionPtr exp_lambda_x = Teuchos::rcp( new Exp_ax( lambda ) );
    FunctionPtr exp_2lambda_x = Teuchos::rcp( new Exp_ax( 2 * lambda ) );
    FunctionPtr sin_2pi_y = Teuchos::rcp( new Sin_ay( 2 * PI ) );
    FunctionPtr cos_2pi_y = Teuchos::rcp( new Cos_ay( 2 * PI ) );

    u1_exact = 1.0 - exp_lambda_x * cos_2pi_y;
    u2_exact = (lambda / (2 * PI)) * exp_lambda_x * sin_2pi_y;

    p_exact = 0.5 * exp_2lambda_x;

    // RHS is given by (1/Re) * \Delta u - grad p - u grad u
    FunctionPtr f1 = -p_exact->dx() + (1.0/Re) * (u1_exact->dx()->dx() + u1_exact->dy()->dy())
                     - u1_exact * u1_exact->dx() - u2_exact * u1_exact->dy();
    FunctionPtr f2 = -p_exact->dy() + (1.0/Re) * (u2_exact->dx()->dx() + u2_exact->dy()->dy())
                     - u1_exact * u2_exact->dx() - u2_exact * u2_exact->dy();
```

6. Verification

We verify the code in several ways, including:

- assertions,
- unit tests,
- integration tests, and
- use in applications.

Below, we describe the first three in more detail; the fourth is discussed in Section 7.

Assertions. Throughout Camellia, and throughout the Trilinos packages on which Camellia relies, assertions are employed to check the reasonableness of method arguments and method results. For example, when multi-dimensional arrays are employed, we check that they have the expected number of dimensions and that the indices employed fall within appropriate bounds. Similarly, when two `Function`s are added together, their ranks are checked, and an exception is thrown if they disagree.

By means of these assertions, coding errors can be found with relative ease: in our experience, most bugs lead to exceptional states, allowing them to be quickly identified and corrected.

Unit and integration tests. We have a host of tests in a home-grown test suite, `DPGTests`—at the time of this writing, there are 151 tests, implemented in about 17,000 lines of code. (This compares to approximately 35,000 lines of code for the Camellia library.) There are both unit tests and integration tests—while we do not make a hard distinction between these, we understand unit tests to be lower-level, verifying the correctness of individual methods, while integration tests verify several classes working in concert. An example of a unit test would be one that checks that the integral of a unit `Function` over a mesh returns the area of the mesh domain; a test that verifies that a Navier-Stokes solve converges to the manufactured solution due to Kovasznay would be an integration test.

We run these tests often in the course of adding new features or refactoring the code, providing us with immediate feedback on incidental bugs introduced to the code in the course of such changes. For example, we are now in the process of rewriting the discretization-handling code to support the minimum rule[16] in arbitrary dimensions, where Camellia previously has supported the maximum rule in two dimensions. This has required significant revision and refactoring; however, by adopting a unified interface that supports both the older maximum rule and the new minimum rule, we are able to take advantage of many existing tests against the older code to expedite the debugging of the changes.

In short, by employing assertions and unit and integration tests, we can maintain a codebase that is *robust* in the sense that it allows significant revisions to be made while verifying the correctness of the changes relative to previous versions of the code.

7. Some Applications

Camellia has been used in a variety of applications. The fact that the results agree with the predictions of the analysis further verifies the code; the fact that the results have many appealing properties (in particular, the ability to solve linear and nonlinear PDE problems adaptively starting with a very coarse mesh) validates both the code and the DPG methodology more broadly. Camellia has been used to solve the following PDE problems, among others:

- convection-dominated diffusion [8, 17],

[16]In finite element meshes, the minimum rule states that when neighbors disagree on the discretization, a minimal basis (the intersection of the discretizations) should be used along the shared boundary. This contrasts with the maximum rule, in which a maximal basis is employed.

- Burgers's equations [7],
- compressible Navier-Stokes [7],
- Stokes [25, 17], and
- incompressible Navier-Stokes [24].

To provide a taste of the kinds of results that DPG and Camellia allow, here we include results from one numerical experiment, the classical lid-driven cavity problem for Stokes flow. This is the motivating example we use throughout Sections 3 and 4. A complete code listing for the driver can be found in AppendixA; including comments and formatting, this requires less than 150 lines of code.

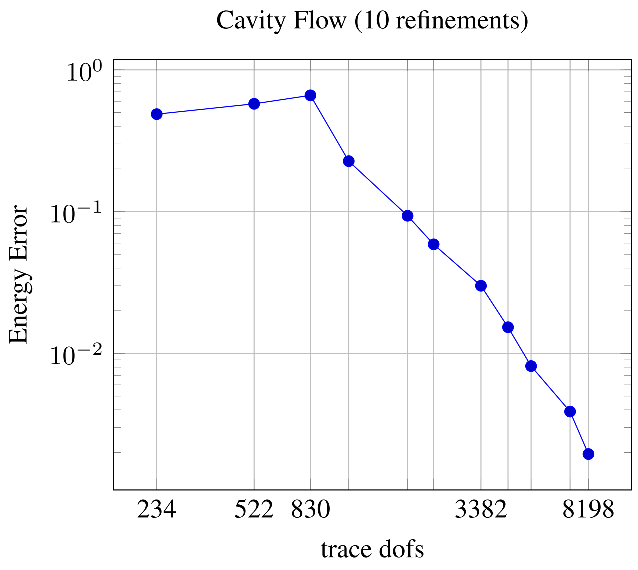
We solve the driven cavity problem with the "ramp" boundary conditions described in Section 4, starting from a coarse 2×2 quartic mesh, performing 10 h-refinement steps. We use our static condensation solver (which solves for the field variables locally, in parallel, and poses the global problem entirely in terms of trace variables) with MUMPS for the global solve, recording the number of trace degrees of freedom (a measure of the cost of the global problem) and the energy error after each refinement. Recalling that the "ramp" in the BCs has a width of $\frac{1}{64}$ the domain width, only after the fifth refinement step will we resolve the BCs exactly—this explains the fact that for early refinement steps the driver records a spurious increase in energy error; the true energy error, relative to the problem with exact BCs, must monotonically decrease as we refine.

Figure 4(a) shows the reduction of energy error under mesh refinement. Neglecting the spurious results for the first few meshes, we observe the expected exponential decrease in energy error relative to the number of degrees of freedom, and we see that the error diminishes by about three orders of magnitude relative to the coarse mesh solution. The final mesh is shown in Figure 4(b); this has 172 elements and 8198 trace degrees of freedom. The net mass flux for the final mesh—defined as the sum of the integrals of $\binom{\widehat{u}_1}{\widehat{u}_2} \cdot \widehat{\mathbf{n}}$ over each element boundary—is of order 10^{-17}.

When running on a single core of a 2013 MacBook Pro with a quad-core 2.7 GHz Intel Core i7 CPU, the driver completes in approximately 75 seconds; on 2 cores, it takes 45 seconds; on 4 cores, 35 seconds. The imperfect parallel speedup is due to parts of the code that are not distributed, including MUMPS's analysis stage (we are not yet using the optional parallel analysis libraries) and Camellia's mesh construction and storage. As the code matures, we hope to address each of the bottlenecks that stand in the way of efficient parallel execution.

8. Conclusions and Future Work

In this paper, we have described work to date on Camellia, focusing on our support for two-dimensional, steady-state DPG solvers. Camellia offers a general software framework for rapid development of DPG solvers which we developed for our DPG research—offering convenient, modular interfaces for the definition of DPG bilinear forms and test space norms, providing simple, extensible mechanisms for h-, p-, and hp-adaptivity, and taking advantage of the embarrassingly parallel nature of optimal test function determination to provide scalable computation of the stiffness matrix, among other features.



(a) Cavity Flow, Energy Error

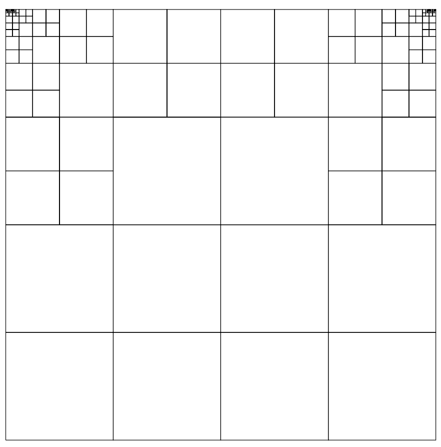

(b) Cavity Flow, Final Mesh

Figure 4: Left: energy error for Stokes cavity flow starting with a quartic 2×2 mesh, for 10 automatic h-refinements. The early solutions—prior to the fifth refinement—under-report the error, due to under-resolved boundary conditions. Right: the final mesh after 10 refinement steps.

Our research efforts thus far suggest several fruitful areas for future exploration, which will guide our ongoing development of Camellia.

8.1. DPG and HPC

DPG has several features that make it a good candidate for future high-performance computing (HPC) applications. First, it is a high-order method; high-order methods are known to provide better computational *intensity*—a measure of the amount of computational effort expended relative to the amount of communication required. Since the progression in HPC is toward higher relative communications costs—"flops will be free" has become a tagline in discussions of exascale computing—higher computational intensity is a desirable feature. DPG's optimal test function computation, being an element-local operation, similarly adds to the computational intensity—DPG's optimal test functions allow one to trade local effort for improved accuracy. Camellia's use of static condensation to reduce the global system to traces and flux coefficients also adds to the computational intensity: no information about field coefficients needs to be communicated between computational nodes.

The robust adaptivity offered by DPG also lends itself to HPC applications, in that it affords a level of *automaticity* uncommon in many PDE applications—moreover, in many applications, the PDE solve is just one component in an optimization loop, which makes failures that require human intervention costly. HPC increases the speed at which computations can in principle be completed, magnifying the cost of any events that halt the computation.

One of the key areas that must be addressed for DPG to be applied to many HPC-scale problems is the development of a DPG solver that scales to thousands—if not tens of thousands, or hundreds of thousands—of processors. As mentioned above, Camellia already provides a highly scalable mechanism for computing the optimal test functions and the global stiffness matrix; however, how to solve the global matrix system in a robust, efficient way remains an open question for most problems.

8.2. Extending Camellia to More Dimensions: 3D and Time

Several of our studies in our research applying DPG to fluid dynamics problems have been regime-limited in the sense that at a critical Reynolds number a flow becomes three-dimensional or transient. Furthermore, certain fluid phenomena—notably vortex stretching—only arise in three-dimensional flows. For these and other reasons, we plan to add support for 3D elements of various topological types—tetrahedra, hexahedra, and pyramids, perhaps—to Camellia.

We would also like to provide some mechanism for transient solves. We plan to add support for *space-time elements*,[17] which compute on a time slab—a tensor product structure, extruding the spatial mesh in the time dimension. The time slab may be refined in both time and space dimensions. Solution coefficients are only stored for the current time slab—a relatively modest additional cost, given that by this mechanism one can bring the entire DPG apparatus to bear on the time dimension. Our immediate plan in this regard is to add support for space-time elements to Camellia, by adding a mechanism for temporal extrusion of arbitrary element discretizations—thereby allowing transient computations for all element types supported by Camellia.

8.3. Variations on the DPG Theme

We would like to add support for continuous finite element spaces and a Python interface to Camellia, to enhance the breadth of Camellia's applicability and the ease with which end users may take advantage of it. This will allow both easier comparison with existing finite element technologies, as well as exploration of new variations on the DPG theme—the usual DPG methodology requires a discontinuous test space, for example, but there is no essential requirement that the trial space be discontinuous—see, for example, Demkowicz and Gopalakrishnan on the *primal* DPG method [15]. It is even possible to pursue a DPG-like scheme that involves continuous test functions—such an approach has recently been investigated by Dahmen et al. [9], as well as Broersen and Stevenson [5]. Having continuous finite elements available within Camellia would also facilitate certain forms of post-processing, such as solving for a streamfunction in an incompressible flow simulation.

8.4. Conclusion

In this paper, we have introduced Camellia, a software framework that aims to make DPG research and experimentation simple, and thereby facilitate efforts to deliver on DPG's great promise of accurate solutions, automatic stability, and robust adaptivity. We look forward to continuing this work, extending Camellia to support larger-scale computations and problems in three-dimensional and transient regimes.

[17] In 2011, Chan et al. solved a transient 1D convection-dominated diffusion problem using DPG with space-time elements [6].

Acknowledgments. This work is supported by the U.S. Department of Energy, Basic Energy Sciences, Office of Science, under Contract No. DE-AC02-06CH11357. Roberts was supported in his dissertation work, of which the present work is an outgrowth, by the Department of Energy [National Nuclear Security Administration] under Award No. DE-FC52-08NA28615. We also thank Pavel Bochev and Denis Ridzal for hosting and collaborating with Roberts in the summers of 2010 and 2011, when he was supported by internships at Sandia National Laboratories (Albuquerque). We thank Jesse Chan and Truman Ellis for their contributions to Camellia—both provided significant feedback and testing as they developed applications using Camellia; Chan also contributed the `RieszRep` implementation and the static condensation solver, while Ellis wrote the VTK output mechanisms which we use for nearly all of our visualization.

Provenance. Portions of this work are closely derived from the author's dissertation [24]. In particular, Section 3 is derived from Chapter 2 of the dissertation, Section 4 from Chapter 3, and Section 8 from Chapter 10.

AppendixA. Stokes Driver for Lid-Driven Cavity Flow

Here, we provide source code for a driver that solves the lid-driven cavity flow problem using the Stokes equations. This is the driver we used to generate the results in Section 7, and of which we provided snippets in Section 4.

```
#include "RefinementStrategy.h"
#include "PreviousSolutionFunction.h"
#include "MeshFactory.h"
#include "SolutionExporter.h"
#include <Teuchos_GlobalMPISession.hpp>
#include "GnuPlotUtil.h"

class TopBoundary : public SpatialFilter {
public:
  bool matchesPoint(double x, double y) {
    double tol = 1e-14;
    return (abs(y-1.0) < tol);
  }
};

class RampBoundaryFunction_U1 : public SimpleFunction {
  double _eps; // ramp width
public:
  RampBoundaryFunction_U1(double eps) {
    _eps = eps;
  }
  double value(double x, double y) {
    if ( (abs(x) < _eps) ) { // top left
      return x / _eps;
    } else if ( abs(1.0-x) < _eps) { // top right
      return (1.0-x) / _eps;
    } else { // top middle
      return 1;
    }
  }
};

int main(int argc, char *argv[]) {
  Teuchos::GlobalMPISession mpiSession(&argc, &argv);
  int rank = Teuchos::GlobalMPISession::getRank();
  VarFactory varFactory;
  // traces:
  VarPtr u1hat = varFactory.traceVar("\\widehat{u}_1");
  VarPtr u2hat = varFactory.traceVar("\\widehat{u}_2");
  VarPtr t1_n = varFactory.fluxVar("\\widehat{t}_{1n}");
  VarPtr t2_n = varFactory.fluxVar("\\widehat{t}_{2n}");
  // fields:
  VarPtr u1 = varFactory.fieldVar("u_1", L2);
  VarPtr u2 = varFactory.fieldVar("u_2", L2);
  VarPtr sigma1 = varFactory.fieldVar("\\sigma_1", VECTOR_L2);
  VarPtr sigma2 = varFactory.fieldVar("\\sigma_2", VECTOR_L2);
  VarPtr p = varFactory.fieldVar("p");
  // test functions:
  VarPtr tau1 = varFactory.testVar("\\tau_1", HDIV);  // tau_1
  VarPtr tau2 = varFactory.testVar("\\tau_2", HDIV);  // tau_2
  VarPtr v1 = varFactory.testVar("v1", HGRAD);        // v_1
  VarPtr v2 = varFactory.testVar("v2", HGRAD);        // v_2
  VarPtr q = varFactory.testVar("q", HGRAD);          // q

  BFPtr stokesBF = Teuchos::rcp( new BF(varFactory) );
  double mu = 1.0; // viscosity
  // tau1 terms:
  stokesBF->addTerm(u1, tau1->div());
  stokesBF->addTerm(sigma1, tau1); // (sigma1, tau1)
  stokesBF->addTerm(-u1hat, tau1->dot_normal());

  // tau2 terms:
  stokesBF->addTerm(u2, tau2->div());
  stokesBF->addTerm(sigma2, tau2);
  stokesBF->addTerm(-u2hat, tau2->dot_normal());

  // v1:
  stokesBF->addTerm(mu * sigma1, v1->grad()); // (mu sigma1, grad v1)
  stokesBF->addTerm( - p, v1->dx() );
  stokesBF->addTerm( t1_n, v1);
  // v2:
  stokesBF->addTerm(mu * sigma2, v2->grad()); // (mu sigma2, grad v2)
  stokesBF->addTerm( - p, v2->dy());
  stokesBF->addTerm( t2_n, v2);

  // q:
  stokesBF->addTerm(-u1,q->dx()); // (-u, grad q)
  stokesBF->addTerm(-u2,q->dy());
  FunctionPtr n = Function::normal();
  stokesBF->addTerm(u1hat * n->x() + u2hat * n->y(), q);

  int k = 4; // poly order for field variables
  int H1Order = k + 1;
  int delta_k = 2;   // test space enrichment
  double width = 1.0, height = 1.0;
  int horizontalCells = 2, verticalCells = 2;
  MeshPtr mesh = MeshFactory::quadMesh(stokesBF, H1Order, delta_k, width, height,
                                       horizontalCells, verticalCells);

  RHSPtr rhs = RHS::rhs(); // zero right-hand side

  BCPtr bc = BC::bc();
  SpatialFilterPtr topBoundary = Teuchos::rcp( new TopBoundary );
  SpatialFilterPtr otherBoundary = SpatialFilter::negatedFilter(topBoundary);

  // top boundary:
  FunctionPtr u1_bc_fxn = Teuchos::rcp( new RampBoundaryFunction_U1(1.0/64) );
  FunctionPtr zero = Function::zero();
  bc->addDirichlet(u1hat, topBoundary, u1_bc_fxn);
  bc->addDirichlet(u2hat, topBoundary, zero);

  // everywhere else:
  bc->addDirichlet(u1hat, otherBoundary, zero);
  bc->addDirichlet(u2hat, otherBoundary, zero);

  bc->addZeroMeanConstraint(p);

  IPPtr graphNorm = stokesBF->graphNorm();

  SolutionPtr solution = Solution::solution(mesh, bc, rhs, graphNorm);

  double energyThreshold = 0.20;
  RefinementStrategy refinementStrategy( solution, energyThreshold );

  Teuchos::RCP<Solver> mumpsSolver = Teuchos::rcp( new MumpsSolver );
  solution->condensedSolve(mumpsSolver);
  int refCount = 10;
  for (int refIndex=0; refIndex < refCount; refIndex++) {
    double energyError = solution->energyErrorTotal();
    if (rank==0) {
      cout << "Before refinement " << refIndex << ", energy error = " << energyError;
      cout << " (using " << mesh->numFluxDofs() << " trace degrees of freedom)." << endl;
    }
    refinementStrategy.refine();
    solution->condensedSolve(mumpsSolver);
  }
  double energyErrorTotal = solution->energyErrorTotal();

  FunctionPtr massFlux = Teuchos::rcp(new PreviousSolutionFunction(solution, u1hat * n->x() + u2hat * n->y()));
  double netMassFlux = massFlux->integrate(mesh); // integrate over the mesh skeleton

  if (rank==0) {
    cout << "Final mesh has " << mesh->numActiveElements() << " elements and ";
    cout << mesh->numFluxDofs() << " trace dofs.\n";
    cout << "Final energy error: " << energyErrorTotal << endl;
    cout << "Net mass flux: " << netMassFlux << endl;
  }

  VTKExporter solnExporter(solution,mesh,varFactory);
  solnExporter.exportSolution("stokesCavityFlowSolution");

  GnuPlotUtil::writeComputationalMeshSkeleton("cavityFlowRefinedMesh", mesh);

  return 0;
}
```

[1] P. R. 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*, 23(1):15–41, 2001.

[2] Wolfgang Bangerth, Carsten Burstedde, Timo Heister, and Martin Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Transactions on Mathematical Software*, 38(2), 2011.

[3] Wolfgang Bangerth and Guido Kanschat. Concepts for object-oriented finite element software – the deal.II library. *IWR*, 1999.

[4] Pavel Bochev and R. B. Lehoucq. On the finite element solution of the pure Neumann problem. *SIAM Review*, 47(1):55–66, March 2005.

[5] Dirk Broersen and Rob Stevenson. A Petrov-Galerkin discretization with optimal test space of a mild-weak formulation of convection-diffusion equations in mixed form. `http://staff.science.uva.nl/~rstevens/papers/DPG.pdf`, November 2012.

[6] J. Chan, L. Demkowicz, and M. Shashkov. Space-time DPG for shock problems. Technical Report Technical Report LA-UR 11-05511, LANL, September 2011.

[7] Jesse Chan, Leszek Demkowicz, and Robert Moser. A DPG method for steady viscous compressible flow. *Computers & Fluids*, (0):–, 2014.

[8] Jesse Chan, Norbert Heuer, Tan Bui-Thanh, and Leszek Demkowicz. A robust DPG method for convection-dominated diffusion problems II: Adjoint boundary conditions and mesh-dependent test norms. *Computers & Mathematics with Applications*, 67(4):771 – 795, 2014. High-order Finite Element Approximation for Partial Differential Equations.

[9] W. Dahmen, A. Cohen, and G. Welper. Adaptivity and variational stabilization for convection-diffusion equations. *ESAIM: Mathematical Modelling and Numerical Analysis*, 46(5):1247–1273, 2012.

[10] Timothy A. Davis and Ekanathan Palamadai Natarajan. Algorithm 907: Klu, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw.*, 37(3):36:1–36:17, September 2010.

[11] L. Demkowicz. *Computing with hp Finite Elements. I. One- and Two-Dimensional Elliptic and Maxwell Problems*. Chapman & Hall/CRC Press, Taylor and Francis, October 2006.

[12] L. Demkowicz and J. Gopalakrishnan. A class of discontinuous Petrov-Galerkin methods. Part I: The transport equation. *Comput. Methods Appl. Mech. Engrg.*, 199:1558–1572, 2010. See also ICES Report 2009-12.

[13] L. Demkowicz and J. Gopalakrishnan. Analysis of the DPG method for the Poisson problem. *SIAM J. Num. Anal.*, 49(5):1788–1809, 2011.

[14] L. Demkowicz and J. Gopalakrishnan. A class of discontinuous Petrov-Galerkin methods. Part II: Optimal test functions. *Numer. Meth. Part. D. E.*, 27(1):70–105, January 2011.

[15] L. Demkowicz and J. Gopalakrishnan. A primal DPG method without a first-order reformulation. *Computers & Mathematics with Applications*, 66(6):1058 – 1064, 2013.

[16] L. Demkowicz, J. Kurtz, D. Pardo, M. Paszyński, W. Rachowicz, and A. Zdunek. *Computing with hp Finite Elements. II. Frontiers: Three-Dimensional Elliptic and Maxwell Problems with Applications*. Chapman & Hall/CRC, October 2007.

[17] T. E. Ellis, L. F. Demkowicz, and J. L. Chan. Locally conservative discontinuous Petrov-Galerkin finite elements for fluid problems. Technical report, ICES, December 2013.

[18] W.J. Gordon and C.A. Hall. Transfinite element methods: blending function interpolation over arbitrary curved element domain. *Numer. Math.*, 21:109–129, 1973.

[19] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[20] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey. libMesh: a C++ library for parallel adaptive mesh refinement/-coarsening simulations. *Eng. with Comput.*, 22(3):237–254, December 2006.

[21] L. I. G. Kovasznay. Laminar flow behind a two-dimensional grid. *Mathematical Proceedings of the Cambridge Philosophical Society*, 44(01):58–62, 1948.

[22] A. Logg, K.-A. Mardal, and G. N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012.

[23] H.K. Moffatt. Viscous and resistive eddies near a sharp corner. *Journal of Fluid Mechanics*, 18(1):1–18, 1964.

[24] Nathan V. Roberts. *A Discontinuous Petrov-Galerkin Methodology for Incompressible Flow Problems*. PhD thesis, University of Texas at Austin, 2013.

[25] Nathan V. Roberts, Tan Bui-Thanh, and Leszek F. Demkowicz. The DPG method for the Stokes problem. *Computers and Mathematics with Applications*, 2014.

[26] J. Zitelli, I. Muga, L. Demkowicz, J. Gopalakrishnan, D. Pardo, and V. Calo. A class of discontinuous Petrov-Galerkin methods. Part IV: Wave propagation problems. *J. Comp. Phys.*, 230:2406–2432, 2011.