

SANDIA REPORT

SAND2011-6678

Unlimited Release

Printed September, 2011

A Toolbox for a Class of Discontinuous Petrov-Galerkin Methods Using Trilinos

Nathan V. Roberts, Denis Ridzal, Pavel B. Bochev, Leszek D. Demkowicz

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.
Approved for public release; further dissemination unlimited.



Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



SAND2011-6678
Unlimited Release
Printed September, 2011

A Toolbox for a Class of Discontinuous Petrov-Galerkin Methods Using Trilinos

Nathan V. Roberts
The University of Texas at Austin
Institute for Computational Engineering and Science
201 E 24th St
Austin, TX 78712
nroberts@ices.utexas.edu

Denis Ridzal
Optimization and Uncertainty Quantification, MS-1320
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1320
dridzal@sandia.gov

Pavel B. Bochev
Numerical Analysis and Applications, MS-1320
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1320
pbboche@sandia.gov

Leszek D. Demkowicz
The University of Texas at Austin
Institute for Computational Engineering and Science
201 E 24th St
Austin, TX 78712
leszek@ices.utexas.edu.

Abstract

The class of discontinuous Petrov-Galerkin finite element methods (DPG) proposed by L. Demkowicz and J. Gopalakrishnan [4, 5] guarantees the optimality of the solution in an energy norm and produces a symmetric positive definite stiffness matrix, among other desirable properties. In this paper, we describe a toolbox, implemented atop Sandia's Trilinos library, for rapid development of solvers for DPG methods. We use this toolbox to develop solvers for the Poisson and Stokes problems.

Acknowledgments

Nathan Roberts and Leszek Demkowicz thank The Center for Predictive Engineering and Computational Sciences (PECOS) for its continued funding and support. PECOS is a DOE-funded Center of Excellence within the Institute for Computational Engineering and Sciences (ICES) at The University of Texas at Austin.

Denis Ridzal and Pavel Bochev thank the DOE Office of Science Advanced Scientific Computing Research (ASCR) for its continued support.

Contents

1	Introduction	9
2	DPG Method	10
2.1	Energy Norm	10
2.2	Optimal Test Functions	10
2.3	Optimal Test Space for U_n	11
2.4	Practical Realization	11
3	Poisson Formulation	13
4	Two Standard Choices for Test Space Norm	14
4.1	Mathematician’s Norm	14
4.2	Optimal and Quasi-Optimal Test Norms	14
5	Code Overview	15
6	Poisson Implementation	17
6.1	Poisson Bilinear Form	17
6.2	Poisson Right-Hand Side	23
6.3	Poisson Boundary Conditions	24
6.4	Manufactured Solutions	26
6.5	Test Space Inner Product Specification	27
7	Meshing and Solving	28
7.1	The Mesh Class	28
7.2	The ExactSolution Class	29
7.3	The Solution Class	30
7.4	The HConvergenceStudy Class	30
7.5	Visualization	31
8	Stokes Formulations	32
8.1	Stokes Formulation I: VSP Formulation	32
8.2	Stokes Formulation II: Velocity-Vorticity-Pressure Formulation	34
9	Numerical Results	35
9.1	Convergence Studies on Uniform Meshes	35
9.2	Meshes of Multiple Polynomial Orders	35
9.3	“Hybrid” Mesh Convergence Studies	37
10	Conclusions and Future Work	38

Appendix

A	Results of Convergence Studies on Uniform Meshes	40
B	Results of Convergence Studies on Hybrid Meshes	46
C	Solution Plots	49

1 Introduction

Recently, L. Demkowicz and J. Gopalakrishnan have proposed a new class of discontinuous Petrov-Galerkin (DPG) methods [4, 5, 6, 9, 3], which compute test functions that are adapted to the problem of interest to produce stable discretization schemes. An important choice that must be made in the application of the method is the definition of the inner product on the test space. In this paper, we describe a toolbox for rapid development of solvers for DPG methods, implemented atop Sandia's Trilinos Project [7], using especially the Intrepid package of interoperable tools for compatible discretizations [2]. We use our DPG toolbox to develop solvers for the Poisson and Stokes problems.

Whereas traditional Galerkin methods use the same space for test and trial spaces, Petrov-Galerkin methods allow the test and trial spaces to differ. The DPG approach computes test functions that are *optimal*, in a sense that we make precise in Section 2. One consequence of this choice of test functions is that the stiffness matrix for a continuous, weakly coercive variational formulation is symmetric (hermitian, for complex-valued problems) and positive definite. Of course, the determination of test functions is an extra step compared with traditional methods; it is important that these can be determined cheaply. By using discontinuous Galerkin (DG) formulations, DPG achieves this, reducing the computation of the test functions to a local problem. Our method bears some resemblance to the MDG method [8] in that a local problem is solved on each element. The key difference with that paper is that in MDG the local problem is restriction of the original equations whereas in DPG the local problem is implied by the selected test space inner product. Furthermore, in MDG the local problem is used to express DG degrees of freedom in terms of continuous degrees of freedom, i.e., to effect static condensation on the element.

We have implemented a toolbox for solving PDEs using DPG. Our eventual goal is to produce an *hp*-adaptive implementation of DPG which allows implementation of a solver for a new variational formulation with minimal effort. Currently, *p*-refinements are functional and the code is designed in anticipation of *h*-refinement, although *hp*-adaptivity is not yet completed. The code supports both triangular and quadrilateral elements of arbitrary order.

The paper is structured as follows. In Section 2, we give an introduction to the basic features of the DPG method. In Section 3, we derive the weak formulation of the Poisson problem. We describe two commonly used test space norms which the code explicitly supports in Section 4. In Section 5, we give a brief overview of the code. In Section 6 we give complete details on implementing the Poisson formulation using our toolbox. We then discuss mesh generation and solving in Section 7. We briefly discuss two Stokes formulations in Section 8. In Section 9 we present some numerical results produced using the code. We conclude in Section 10.

2 DPG Method

Here, we sketch some of the main features of the DPG method. For details, we refer the reader to a series of papers by Demkowicz et al., in particular the second ICES Report [5], from which most of this section is derived. We begin with theoretical definitions and results, and then describe the approach to practical realization. Consider the abstract variational boundary-value problem:

$$\text{Find } u \in U : b(u, v) = l(v) \quad \forall v \in V. \quad (2.1)$$

We take U and V to be real Hilbert spaces. We assume $b(\cdot, \cdot)$ is continuous, i.e.

$$|b(u, v)| \leq M \|u\|_U \|v\|_V, \quad (2.2)$$

for some real M . We assume also that $b(\cdot, \cdot)$ is weakly coercive, that is

$$\inf_{\|u\|_U=1} \sup_{\|v\|_V=1} b(u, v) > \gamma, \quad (2.3)$$

for some $\gamma > 0$. If we additionally assume that

$$\{v \in V : b(u, v) = 0 \quad \forall u \in U\} = \{0\}, \quad (2.4)$$

then it is well known that the problem (2.1) has a unique solution provided that $l \in V'$, the dual of V .

2.1 Energy Norm

We define an alternate norm, called the *energy norm*, on the trial space U by

$$\|u\|_E \stackrel{\text{def}}{=} \sup_{\|v\|_V=1} b(u, v). \quad (2.5)$$

This norm is the one in which the optimality is guaranteed by the selection of optimal test functions. It is an equivalent norm to the standard norm on U , i.e.

$$\gamma \|u\|_U \leq \|u\|_E \leq M \|u\|_U \quad \forall u \in U. \quad (2.6)$$

2.2 Optimal Test Functions

We are now prepared to give a definition of the optimal test functions. Define a map $T : U \rightarrow V$ from the trial space to the test space by: For $u \in U$, define Tu , the *optimal test function* corresponding to u , as the unique solution to

$$(Tu, v)_V = b(u, v) \quad \forall v \in V.$$

By the Riesz representation theorem, T is well-defined. Note that

$$\|u\|_E = \sup_{\|v\|_V=1} b(u, v) = \sup_{\|v\|_V=1} (Tu, v)_V = \frac{1}{\|Tu\|_V} (Tu, Tu)_V = \|Tu\|_V.$$

Thus the energy norm is generated by the inner product on V , i.e.

$$(u, u)_E \stackrel{\text{def}}{=} (Tu, Tu)_V. \quad (2.7)$$

In practice, we approximate T by a discrete operator T_n , described in Section 2.4.

2.3 Optimal Test Space for U_n

Take a finite-dimensional trial space $U_n \subset U$. Define the *optimal test space* for U_n as $V_n = \text{span}\{Te_j : j = 1, \dots, n\}$, where the e_j form a basis for U_n .

Solve the discrete problem

$$\text{Find } u_n \in U_n : b(u_n, v) = l(v) \quad \forall v \in V_n. \quad (2.8)$$

Then the error is the best approximation error in the energy norm,

$$\|u - u_n\|_E = \inf_{w_n \in U_n} \|u - w_n\|_E, \quad (2.9)$$

and this is the sense in which the test space is *optimal*.

2.4 Practical Realization

The method involves two steps: first, find the optimal test functions; second, use the optimal test functions to solve the discrete problem 2.8. The optimal test functions are not in general polynomials. In practice, we approximate them with an “enriched” polynomial space — a space of polynomials of slightly higher degree than the trial space. This is done to provide a higher-fidelity approximation to the continuous space of optimal test functions. The best choice for the amount of “enrichment” is determined experimentally for each problem.

In general, we apply the following procedure:

1. Given a boundary value problem, develop mesh-dependent $b(\cdot, \cdot)$ with test space V that allows inter-element discontinuities (hence *Discontinuous* Petrov-Galerkin). We develop this in Section 8.
2. Choose trial space U_n (in particular the norm of interest in U_n), and the inner product on V , which will be motivated by the choice of trial space.

3. Compute optimal test functions. Approximate T by $T_n : U_n \rightarrow \tilde{V}_n \subset V$. We use an enriched space of piecewise polynomials for \tilde{V}_n . Defining $t_j = T_n e_j$, we solve

$$(t_j, \tilde{e}_i)_V = b(e_j, \tilde{e}_i)$$

for t_j , where the \tilde{e}_i form the basis for \tilde{V}_n .

4. Use the optimal test functions to solve the problem on $U_n \times \tilde{V}_n$. We note that the stiffness matrix here is symmetric positive definite (hermitian, for a complex-valued problem),

$$\begin{aligned} b(e_j, t_i) &= (T_n e_j, t_i)_V = (T_n e_j, T_n e_i)_V = \overline{(T_n e_i, T_n e_j)}_V \\ &= \overline{(T_n e_i, t_j)}_V = \overline{b(e_i, t_j)}. \end{aligned}$$

Also, note that this means that we may compute the stiffness matrix in terms of the inner product on the test space V , without explicit recourse to the bilinear form.

3 Poisson Formulation

Our general approach to variational formulations in DPG is as follows. First, rewrite the strong form of the problem as a system of first-order partial differential equations. Then, multiply by test functions and integrate by parts, moving all derivatives to the test functions, introducing fluxes and traces wherever the trial space variables appear in boundary integrals. We thus arrive at the *ultra-weak* form of the problem, a formulation in which all solution variables are in L^2 .

We apply the DPG method to the Poisson problem in two dimensions:

$$\nabla \cdot \nabla \phi = f.$$

We define $\psi = \nabla \phi$ and rewrite as a first-order system:

$$\begin{aligned} \nabla \phi - \psi &= 0 \\ \nabla \cdot \psi &= f. \end{aligned}$$

Multiply by test functions and integrate:

$$\begin{aligned} \int_K \nabla \phi \cdot \mathbf{q} - \int_{\mathbf{K}} \psi \cdot \mathbf{q} &= 0 \\ \int_K \nabla \cdot \psi v &= \int_K f v. \end{aligned}$$

Integrate by parts, introducing flux $\widehat{\psi}_n$ and trace $\widehat{\phi}$ on the boundary:

$$\begin{aligned} - \int_K \phi \nabla \cdot \mathbf{q} - \int_{\mathbf{K}} \psi \cdot \mathbf{q} + \int_{\partial \mathbf{K}} \widehat{\phi} \mathbf{q} \cdot \mathbf{n} &= 0 \\ - \int_K \psi \cdot \nabla v + \int_{\partial K} \widehat{\psi}_n v &= \int_K f v. \end{aligned}$$

4 Two Standard Choices for Test Space Norm

In this section, we describe two commonly used norms on the test space — recall that the choice of test space norm will determine the energy norm, and thus the sense in which test functions are optimal. Our implementation makes usage of these norms simple.

4.1 Mathematician’s Norm

In general, test functions in a variational form each belong to some functional space on which a norm is already defined. Therefore, a mathematically natural choice for a norm on the test space will be the Euclidean combination of these norms. In the case of our Poisson formulation, $\mathbf{q} \in H(\text{div})$ and $v \in H(\text{grad})$, so define

$$\|(\mathbf{q}, v)\|_V^2 \stackrel{\text{def}}{=} \int_{\Omega} ((\nabla \cdot \mathbf{q})^2 + \mathbf{q} \cdot \mathbf{q} + \nabla v \cdot \nabla v + v^2)$$

as the *mathematician’s* norm on the test space.

4.2 Optimal and Quasi-Optimal Test Norms

We know from the analysis that DPG delivers the best solution in the *energy norm*, and that the choice of norm on the test space determines the energy norm. We can then ask whether it is possible to select the test space norm in such a way that the energy norm is exactly the norm of interest (L^2 , in our case) — and the answer is yes; this is given in the abstract setting by

$$\|v\|_V \stackrel{\text{def}}{=} \sup_{\|u\|_U=1} b(u, v),$$

where $\|\cdot\|_U$ is the norm of interest. $\|\cdot\|_V$ is then called the *optimal test norm*. To determine the optimal test norm, we can collect terms by trial space variable, and then take the supremum using the Cauchy-Schwarz inequality. When we do so, we will have some element boundary terms arising from the fluxes and traces. To make the test norm *localizable*, we then replace these terms with L^2 terms on the element interior, often with some weight β . The resultant norm we call the *quasi-optimal test norm*.

5 Code Overview

The directory layout in our code repository is as follows:

```
||-Drivers
||---DPGTests
||---MultiOrderStudy
||---Poisson
||---Stokes
||-SummerProceedings2011
||-build
||-presentation
||-src
||---Basis
||---ConvergenceStudy
||---DofOrdering
||---InnerProduct
||---Mesh
||---Problem
||---Solution
||---visualization
```

The core code is in the `src` directory. Within this, `Basis` contains classes related to basis functions. `ConvergenceStudy` contains the `HConvergenceStudy` class, discussed below in Section 7.4, which provides support for studying convergence rates. `DofOrdering` contains a utility class for maintaining the ordering of coefficients related to the bases of interest. `InnerProduct` contains classes related to implementation of the test-space inner product, discussed in detail in Section 6.5. `Mesh` contains classes related to meshing and elements; Section 7.1 details mesh construction. The `Problem` directory contains classes for the specification of the bilinear form, right hand side, and boundary conditions, discussed in Sections 6.1, 6.2, and 6.3, respectively, using the Poisson problem as a motivating example. `Solution` contains classes for solving such problems, as well as for the specification of exact (manufactured) solutions; these are discussed in Sections 7.2 and 7.3. Finally, the `visualization` directory contains a MATLAB script, discussed in Section 7.5, for plotting solutions.

Several sample drivers can be found in the `Drivers` directory; `DPGTests` runs a test suite against the core code, `StokesStudy`, `StokesStudyHybridMesh` and `PoissonStudy` are drivers built with the `HConvergenceStudy` class, used to produce the results in Section 9.

The `SummerProceedings2011` directory contains the TeX source files, plots, and raw data used used to produce this document. The `presentation` directory contains a presentation covering much the same material.

Finally, the `build` directory contains a makefile that can be used to build the core code and the drivers.

6 Poisson Implementation

We now detail the implementation of Poisson using the toolbox we have developed. To specify a PDE to solve, we need to define the following:

- a bilinear form,
- a right-hand side, and
- boundary conditions.

Our DPG implementation is intended to make it simple to specify each of these, once a first-order weak formulation has been derived.

6.1 Poisson Bilinear Form

Our bilinear form is given by

$$b(\cdot, \cdot) = - \int_K \phi \nabla \cdot \mathbf{q} - \int_{\mathbf{K}} \psi \cdot \mathbf{q} + \int_{\partial \mathbf{K}} \widehat{\phi} \mathbf{q} \cdot \mathbf{n} \\ - \int_K \psi \cdot \nabla v + \int_{\partial K} \widehat{\psi}_n v.$$

Bilinear forms should subclass `BilinearForm`. This abstract class provides the routines required for the rest of the toolbox to determine the optimal test functions, produce the global stiffness matrix, and solve the system. The first step in subclassing is to populate two `vector<int>` structures provided by `BilinearForm`, `_trialIDs` and `_testIDs`. These are simply integer identifiers that the subclass uses for each of the trial and test variables. A typical approach is to define an enumeration for each of these in the subclass header file, and populate the vectors with these in the subclass constructor.

In the Poisson formulation, we have:

- two test functions (\mathbf{q} and v),
- one scalar trial “field” variable (ϕ),
- one vector trial “field” variable (ψ , which our implementation requires us to split into scalars ψ_1 and ψ_2),
- one flux ($\widehat{\psi}_n$),
- and one trace ($\widehat{\phi}$).

Here is a snippet, defining the identifiers we wish to use for test and trial functions, from `PoissonBilinearForm.h`:

```
enum ETestIDs {
    Q_1 = 0,
    V_1
};

enum ETrialIDs {
    PHI_HAT = 0,
    PSI_HAT_N,
    PHI,
    PSI_1,
    PSI_2
};
```

The constructor for `PoissonBilinearForm` simply populates `_trialIDs` and `_testIDs` with these enumeration values:

```
PoissonBilinearForm::PoissonBilinearForm() {
    _testIDs.push_back(Q_1);
    _testIDs.push_back(V_1);

    _trialIDs.push_back(PHI_HAT);
    _trialIDs.push_back(PSI_HAT_N);
    _trialIDs.push_back(PHI);
    _trialIDs.push_back(PSI_1);
    _trialIDs.push_back(PSI_2);
}
```

`BilinearForm` also contains several virtual member functions that subclasses must override:

1. `const string & testName(int testID),`
2. `const string & trialName(int trialID),`
3. `EFunctionSpaceExtended`
`functionSpaceForTest(int testID),`
4. `EFunctionSpaceExtended`
`functionSpaceForTrial(int trialID),`
5. `bool isFluxOrTrace(int trialID),`
6. either `bool trialTestOperator(...),` or
`void trialTestOperators(...),`
7. and one of the two forms of `void applyBilinearFormData(...).`

For `testName()` and `trialName()`, we recommend using LaTeX-compatible strings, because `BilinearForm` can output a TeX-friendly string representation of the bilinear form as implemented, a valuable debugging tool. Here is a snippet defining these, from `PoissonBilinearForm.cpp`:

```
// trial variable names:
static const string & S_PHI = "\\phi";
static const string & S_PSI_1 = "\\psi_1";
static const string & S_PSI_2 = "\\psi_2";
static const string & S_PHI_HAT = "\\hat{\\phi}";
static const string & S_PSI_HAT_N = "\\hat{\\psi}_n";
static const string & S_DEFAULT_TRIAL = "invalid_trial";

// test variable names:
static const string & S_Q_1 = "q_1";
static const string & S_V_1 = "v_1";
static const string & S_DEFAULT_TEST = "invalid_test";

const string & PoissonBilinearForm::testName(int testID) {
    switch (testID) {
        case Q_1:
            return S_Q_1;
        break;
        case V_1:
            return S_V_1;
        break;
        default:
            return S_DEFAULT_TEST;
    } }

const string & PoissonBilinearForm::trialName(int trialID) {
    switch(trialID) {
        case PHI:
            return S_PHI;
        break;
        case PSI_1:
            return S_PSI_1;
        break;
        case PSI_2:
            return S_PSI_2;
        break;
        case PHI_HAT:
            return S_PHI_HAT;
        break;
        case PSI_HAT_N:
            return S_PSI_HAT_N;
        break;
        default:
            return S_DEFAULT_TRIAL;
    } }
```

Next, we need to define the continuous spaces to which the test and trial functions belong. These are defined by enumeration `EFunctionSpaceExtended` in

namespace `IntrepidExtendedTypes` (specified in `BilinearForm.h`); this mirrors names and values in `IntrepidTypes::EFunctionSpace`, with the intent of adding spaces to support features such as local conservation. In general in DPG, trial space functions will be in L^2 , with the exception of traces, which will be in H^1 . The appropriate test space for a given test function is determined by the derivatives applied to that test function — thus $\mathbf{q} \in \mathbf{H}(\text{div})$ and $v \in H(\text{grad})$. Here is the code for the Poisson bilinear form:

```

EFunctionSpaceExtended
PoissonBilinearForm::functionSpaceForTrial(int trialID) {
    // Field variables, and fluxes, are all L2.
    if (trialID != PHI_HAT) {
        return IntrepidExtendedTypes::FUNCTION_SPACE_HVOL;
    } else {
        return IntrepidExtendedTypes::FUNCTION_SPACE_HGRAD;
    } }

EFunctionSpaceExtended
PoissonBilinearForm::functionSpaceForTest(int testID) {
    switch (testID) {
        case Q_1:
            return IntrepidExtendedTypes::FUNCTION_SPACE_HDIV;
        break;
        case V_1:
            return IntrepidExtendedTypes::FUNCTION_SPACE_HGRAD;
        break;
        default:
            throw "Error:_unknown_testID";
    } }

```

We also need to specify which of our trial variables are field variables and which are fluxes or traces. This is accomplished by overriding `isFluxOrTrace()`:

```

bool PoissonBilinearForm::isFluxOrTrace(int trialID) {
    if ((PHI_HAT==trialID) || (PSI_HAT_N==trialID)) {
        return true;
    } else {
        return false;
    } }

```

We have now entirely defined the variables that enter the bilinear form. If we now call `BilinearForm's printTestTrialInteractions()` method, we get the following output:

```

***** Interactions with test variable q_1 *****
\int_{\partial K} \hat{\phi} q_1 - \int_K \phi \nabla \cdot q_1
- \int_K \psi_1 \mathbf{i} \cdot q_1 - \int_K \psi_2 \mathbf{j} \cdot q_1

***** Interactions with test variable v_1 *****
\int_{\partial K} \hat{\psi}_n v_1
- \int_K \psi_1 \frac{\partial}{\partial x} v_1
- \int_K \psi_2 \frac{\partial}{\partial y} v_1

```

Each of these lines can be pasted into a TeX document and rendered for comparison with the intended bilinear form. Note that this method will only work approximately when material data varies in space, but for constant material data, it should provide an exact representation of the bilinear form as implemented.

To complete the specification of the bilinear form, we need to define the way that test and trial variables interact, as well as apply material data. The former is specified in terms of operators that should be applied a test and trial pair. In most simple bilinear forms, a given (test, trial) pair will appear only once. In this case, the single-operator `bool trialTestOperator()` method may be used. The subclass should return `true` if the pair appears in the bilinear form, and `false` otherwise. If the pair does appear, then the subclass should set the test and trial operators passed in. Note that because our approach splits the vector ψ into two scalars ψ_1 and ψ_2 , the $\psi \cdot \nabla v$ term must be split into $\psi_1 \frac{\partial}{\partial x} v + \psi_2 \frac{\partial}{\partial y} v$. The code appears below.

```

bool PoissonBilinearForm::trialTestOperator(int trialID,
                                             int testID,
                                             EOperatorExtended &trialOperator,
                                             EOperatorExtended &testOperator) {
    // being DPG, trialOperator will always be OPERATOR_VALUE
    trialOperator = IntrepidExtendedTypes::OPERATOR_VALUE;
    // unless we specify otherwise,
    // trial and test don't interact:
    bool returnValue = false;
    switch (testID) {
        case Q_1:
            switch (trialID) {
                case PSI_1:
                    returnValue = true;
                    // x component of q1 against ps1 (dot product):
                    testOperator = IntrepidExtendedTypes::OPERATOR_X;
                    break;
                case PSI_2:
                    returnValue = true;
                    // y component of q1 against ps1 (dot product)
                    testOperator = IntrepidExtendedTypes::OPERATOR_Y;
                    break;
                case PHI:
                    returnValue = true;
                    testOperator = IntrepidExtendedTypes::OPERATOR_DIV;
                    break;
                case PHI_HAT:
                    returnValue = true;
                    testOperator = IntrepidExtendedTypes::OPERATOR_VALUE;
                    break;
                default:
                    break;
            }
        break;
        case V_1:
            switch (trialID) {
                case PSI_1:

```

```

        returnValue = true;
        testOperator = IntrepidExtendedTypes::OPERATOR_DX;
        break;
    case PSI_2:
        returnValue = true;
        testOperator = IntrepidExtendedTypes::OPERATOR_DY;
        break;
    case PSI_HAT_N:
        returnValue = true;
        testOperator = IntrepidExtendedTypes::OPERATOR_VALUE;
        break;
    default:
        break;
}
default:
    break;
}
return returnValue;
}

```

The multi-operator version is similar, with multiple `EOperatorExtended` values being specified for each test and trial function.

We now specify the material data by overriding `applyBilinearFormData`; there is a variant for both the multi- and single-operator implementations. This will be called only for (test, trial) pairs that have non-zero interaction in the bilinear form. The arguments to the single-argument variant of `applyBilinearFormData` are as follows:

- `int trialID`: the trial variable identifier
- `int testID`: the test variable identifier
- `FieldContainer<double> &testTrialValuesAtPoints`: values to which the material data should be applied
- `FieldContainer<double> &points`: the spatial points at which the material data is applied

There is a static convenience function, `multiplyFCByWeight()` implemented in `BilinearForm`, which simply multiplies all the values in an Intrepid `FieldContainer` by a given weight. This is convenient in a case like our Poisson problem, in which the material data is constant.

```

void PoissonBilinearForm::applyBilinearFormData(int trialID,
        int testID,
        FieldContainer<double> &testTrialValuesAtPoints,
        FieldContainer<double> &points) {
    switch (testID) {
        case Q_1:

```

```

// - (phi, div q1)_K + (phi_hat, q_1n)_dK - (psi, q1)
switch (trialID) {
  case PHI:
    // negate
    multiplyFCByWeight (testTrialValuesAtPoints, -1.0);
    break;
  case PSI_1:
    // negate
    multiplyFCByWeight (testTrialValuesAtPoints, -1.0);
    break;
  case PSI_2:
    // negate
    multiplyFCByWeight (testTrialValuesAtPoints, -1.0);
    break;
  case PHI_HAT:
    // do nothing -- testTrialValuesAtPoints already
    // has the right values
    break;
}
break;
case V_1:
  switch (trialID) {
    // -(psi, grad v1)_K + (psi_hat_n, v1)_dK
    case PHI:
      throw "Error: no (v1, phi) term";
      break;
    case PSI_1:
      // negate
      multiplyFCByWeight (testTrialValuesAtPoints, -1.0);
      break;
    case PSI_2:
      // negate
      multiplyFCByWeight (testTrialValuesAtPoints, -1.0);
      break;
    case PSI_HAT_N:
      // do nothing -- testTrialValuesAtPoints already
      // has the right values
      break;
  }
break;
} }

```

6.2 Poisson Right-Hand Side

To specify the right hand side, we subclass RHS, overriding two functions:

1. `bool nonZeroRHS(int testVarID)`, and
2. `void rhs(int testVarID, FC &physicalPoints, FC &values)`.

There are several versions of such implementations for Poisson. Below, we quote from `PoissonRHSCubic`. A call to `nonZeroRHS(int testVarID)` should return `true` for any test variable which has a non-zero right-hand side. For Poisson, the only such test variable is v ; we integrate a function f against it. A call to `rhs(testVarID, physicalPoints, values)` is responsible for providing the values of the function f at points `physicalPoints` by populating `values`. Taking an exact solution $\phi = x^3 + 2y^3$, we have $f = \nabla \cdot \nabla \phi = 6x + 12y$. Thus our subclass implements the right hand side as follows:

```

bool PoissonRHSCubic::nonZeroRHS(int testVarID) {
    if (testVarID == PoissonBilinearForm::Q_1) {
        // the vector test function, zero RHS:
        return false;
    } else if (testVarID == PoissonBilinearForm::V_1) {
        return true;
    } else {
        return false; // could throw an exception here
    } }

void PoissonRHSCubic::rhs(int testVarID,
                          FieldContainer<double> &physicalPoints,
                          FieldContainer<double> &values) {
    // for an exact solution of x^3 + 2y^3, f = 6x + 12y
    int numCells = physicalPoints.dimension(0);
    int numPoints = physicalPoints.dimension(1);
    int spaceDim = physicalPoints.dimension(2);
    if (testVarID == PoissonBilinearForm::V_1) {
        values.resize(numCells, numPoints);
        for (int cellIndex=0; cellIndex<numCells; cellIndex++) {
            for (int ptIndex=0; ptIndex<numPoints; ptIndex++) {
                double x = physicalPoints(cellIndex, ptIndex, 0);
                double y = physicalPoints(cellIndex, ptIndex, 1);
                values(cellIndex, ptIndex) = 6.0*x + 12.0*y;
            } } } }

```

6.3 Poisson Boundary Conditions

Because we are interested the Poisson problem as a prototype for Stokes, we would like to apply Dirichlet conditions to ψ , leaving ϕ to enter the original, strong form of the equation only through a gradient, analogous to the way that pressure enters the Stokes equations. In DPG, we apply boundary conditions through the fluxes and traces, because these are the only variables formally defined on the boundary. Thus a boundary condition $\psi \cdot n = g$ on $\partial\Omega$ becomes $\hat{\psi}_n = g$ on $\partial\Omega$. To make the problem well-posed, we do need to pin down ϕ . Two ways of doing this are supported by the code — first, a single point value can be imposed as a Dirichlet-like condition on a field variable. This is a discrete trick that may bring some numerical difficulties such as bad conditioning with it — such has been our experience. A second way to make the problem well-posed is to impose a zero-mean condition on ψ . This carries with it

slightly more computational cost than the other approach, but has better numerical properties. The implementation for the latter follows the procedure described in [1].

All the boundary conditions described above can be defined by subclassing the BC class. The methods available for overriding are:

- `bool bcsImposed(int varID):`
specifies whether BCs are anywhere imposed for `varID`;
- `void imposeBC(...):`
requests imposition of BCs at boundary points;
- `bool singlePointBC(int varID):`
specifies whether to impose a single-point BC on a field variable;
- `bool imposeZeroMeanConstraint(int varID):`
specifies whether to impose a zero-mean constraint on a field variable.

The `imposeBC()` method takes arguments:

- `int varID:`
the variable for which BCs are requested;
- `FieldContainer<double> &physicalPoints:`
the points at which BCs might be imposed, dimensions are (C,P,D);
- `FieldContainer<double> &unitNormals:`
outward unit normals at the specified points, dimensions are (C,P,D);
- `FieldContainer<double> &dirichletValues:`
the values to impose at the specified points, dimensions are (C,P);
- `FieldContainer<bool> &imposeHere:`
whether to impose the BC at the specified point, dimensions are (C,P).

The final `imposeHere` argument allows boundary conditions to be specified only along part of the boundary.

In the `PoissonBCCubic` class, we take the single-point option for the condition on ϕ , and impose a Dirichlet condition everywhere along the boundary for $\hat{\psi}_n$. Since $\phi = x^3 + 2y^3$, $\psi = \nabla\phi = \begin{pmatrix} 3x^2 \\ 6y^2 \end{pmatrix}$, so that $\psi \cdot n = 3x^2n_1 + 6y^2n_2$; this is the value we impose on $\hat{\psi}_n$. The code appears below, somewhat redacted, for space and clarity:

```
bool PoissonBCCubic::bcsImposed(int varID) {
    return varID == PoissonBilinearForm::PSI_HAT_N;
}
```

```

bool PoissonBCCubic::singlePointBC(int varID) {
    return varID == PoissonBilinearForm::PHI;
}

void PoissonBCCubic::imposeBC(int varID,
                              FieldContainer<double> &physicalPoints,
                              FieldContainer<double> &unitNormals,
                              FieldContainer<double> &dirichletValues,
                              FieldContainer<bool> &imposeHere) {
    int numCells = physicalPoints.dimension(0);
    int numPoints = physicalPoints.dimension(1);
    int spaceDim = physicalPoints.dimension(2);

    if (varID == PoissonBilinearForm::PHI) {
        for (int cellIndex=0; cellIndex<numCells; cellIndex++) {
            for (int ptIndex=0; ptIndex<numPoints; ptIndex++) {
                double x = physicalPoints(cellIndex,ptIndex,0);
                double y = physicalPoints(cellIndex,ptIndex,1);
                dirichletValues(cellIndex,ptIndex) = x*x*x + 2.0*y*y*y;
                // impose everywhere:
                imposeHere(cellIndex,ptIndex) = true;
            }
        }
    }
    else if (varID == PoissonBilinearForm::PSI_HAT_N) {
        for (int cellIndex=0; cellIndex<numCells; cellIndex++) {
            for (int ptIndex=0; ptIndex<numPoints; ptIndex++) {
                // value = n1 * (3x^2) + n2 * (6y^2)
                double x = physicalPoints(cellIndex,ptIndex,0);
                double y = physicalPoints(cellIndex,ptIndex,1);
                double n1 = unitNormals(cellIndex,ptIndex,0);
                double n2 = unitNormals(cellIndex,ptIndex,1);
                double value = (3.0*x*x)*n1 + (6.0*y*y)*n2;
                dirichletValues(cellIndex,ptIndex) = value;
                // impose everywhere:
                imposeHere(cellIndex,ptIndex) = true;
            }
        }
    }
}
}
}
}

```

6.4 Manufactured Solutions

Using the Sacado package in Trilinos, it is possible to specify exact solutions and apply automatic differentiation to these. This is an ideal way to implement a class of manufactured solutions for a variational problem. While a detailed discussion of Sacado is beyond the scope of this paper, there are two example manufactured solutions, `PoissonExactSolution` and `StokesManufacturedSolution`, included with the code. Each of these uses multiple inheritance to subclass `RHS`, `BC`, as well as `ExactSolution`, an abstract class which provides solution values at a point, and offers facilities for measuring the error in computed solutions.

6.5 Test Space Inner Product Specification

In Section 4, we described two standard test norms, the mathematician’s norm and the quasi-optimal test norm. We provide implementations in `MathInnerProduct` and `OptimalInnerProduct`. The constructors for these simply take a `BilinearForm` object as argument. The value of β in the quasi-optimal test norm is taken to be 1.

If a different test norm is desired, one can subclass `DPGInnerProduct`, and override the operators `()` and `applyInnerProductData()` methods. The interface is very much like that for the `BilinearForm`. One limitation is worth mentioning: at present, `applyInnerProductData()` receives only information about the physical points where the inner product is to be computed; some of our past research has involved inner products that depend on the location relative to an element boundary or the element’s size. Although these are relatively simple extensions, they are not presently supported by the DPG Trilinos toolbox.

7 Meshing and Solving

Once we have a bilinear form specified with a test space inner product, we are ready to produce a mesh on which to solve the variational problem, and examine the error of that solution. There are four classes provided to support this:

- `Mesh`: defines elements, vertices, and test and trial bases.
- `ExactSolution`: defines the exact solution values at a point.
- `Solution`: solves the linear system and stores solution coefficients.
- `HConvergenceStudy`: solves the variational problem on a series of meshes, and computes convergence rates for trial space variables.

In this section, we discuss each of these in turn.

7.1 The Mesh Class

The `Mesh` class represents a bilinear form together with a set of quadrilaterals and triangles on which polynomial bases are defined for each test and trial variable. The basic `Mesh` constructor takes arguments

- `vector<FieldContainer<double> > &vertices`,
- `vector< vector<int> > &elementVertices`,
- `Teuchos::RCP< BilinearForm > bilinearForm`,
- `int pTrial`, and
- `int pToAddTest`.

The `vertices` vector has `FieldContainer` entries with dimensions (`spaceDim`), the spatial dimension of the problem, each of which specifies the physical locations of an element vertex. Each physical vertex should be listed exactly once. (We do assume $(\text{spaceDim}) = 2$ in a few places in the code, so that is a limitation for the moment.)

The `elementVertices` vector contains a vector of 3 or 4 vertex indices for each element—these are the indices of the element’s vertices in the `vertices` vector, and they should be specified in counterclockwise order (clockwise will work as well, but all elements must be specified in the same order, clockwise or counterclockwise). The

Mesh class will automatically determine which element edges lie along the boundary, as well as which edges are shared between elements.

The `bilinearForm` argument is a reference-counted pointer to the bilinear form for the variational problem to be solved.

`pTrial` is the polynomial order of approximation to be used for trial space variables belonging to H^1 (that is, in function space `FUNCTION_SPACE_HGRAD` — typically, these are just the traces), and `pToAddToTest` is the polynomial order to add to this to obtain the order for the test space. Trial space variables in L^2 (function space `FUNCTION_SPACE_HVOL`, typically field variables and fluxes), then these will have order `pTrial-1`.

In addition to the basic mesh constructor, there are two static constructors to build meshes within an axis-aligned rectangle. These are `buildQuadMesh` and `buildQuadMeshHybrid`, with the following signatures:

```
Teuchos::RCP<Mesh> buildQuadMesh(
    const FieldContainer<double> &quadBoundaryPoints,
    int horizontalElements, int verticalElements,
    Teuchos::RCP< BilinearForm > bilinearForm,
    int pTrial, int pTest, bool triangulate=false);
Teuchos::RCP<Mesh> buildQuadMeshHybrid(
    const FieldContainer<double> &quadBoundaryPoints,
    int horizontalElements, int verticalElements,
    Teuchos::RCP< BilinearForm > bilinearForm,
    int pTrial, int pTest);
```

Both methods will construct a regularly spaced grid of *horizontalElements* × *verticalElements* rectangular elements, with `pToAddTest = pTest - pTrial`. `quadBoundaryPoints` is a `FieldContainer` with dimensions (4,2) containing four axis-aligned vertices. (We do hope to relax the axis-alignment constraint in the near future.) In the first method, if `triangulate` is true, these rectangles are cut along a diagonal to form triangular elements. In the second method, half of the elements are split; the mesh is thus a “hybrid” of triangles and quads.

7.2 The ExactSolution Class

In order to measure the L^2 error of a computed solution, we need to know what the exact solution is. The `ExactSolution` class is a mechanism for defining this. Subclasses should override the `solutionValue()` methods, which have signatures:

```
double solutionValue(int trialID,
                    FieldContainer<double> &physicalPoint);
double solutionValue(int trialID,
                    FieldContainer<double> &physicalPoint,
                    FieldContainer<double> &unitNormal);
```

where `unitNormal` and `physicalPoint` are containers with dimension equal to the spatial dimension (2, for the present). The `unitNormal` argument is provided in the second method so that fluxes, which depend on the outward normal, may be computed. As discussed above, the `PoissonExactSolution` class demonstrates an implementation of `ExactSolution` using Sacado for automatic differentiation.

7.3 The Solution Class

A `Solution` object can be constructed and solved as follows:

```
Solution solution(mesh, bc, rhs, innerProduct);
solution.solve();
```

At present, the `solve()` method solves using the KLU implementation provided by Trilinos. The `Solution` class also provides support for computing solution values at points (through the `solutionValues()` methods), as well as outputting a data file representing the mesh solution for a given trial space variable (the `writeToFile()` method).

7.4 The HConvergenceStudy Class

The `HConvergenceStudy` class provides a simple mechanism for studying convergence rates on the meshes supported by the constructors `buildQuadMesh()` and `buildQuadMeshHybrid()`. Its constructor interface is as follows:

```
HConvergenceStudy(Teuchos::RCP<ExactSolution> exactSolution,
                  Teuchos::RCP<BilinearForm> bilinearForm,
                  Teuchos::RCP<RHS> rhs,
                  Teuchos::RCP<BC> bc,
                  Teuchos::RCP<DPGInnerProduct> ip,
                  int minLogElements, int maxLogElements,
                  int H1Order, int pToAdd,
                  bool randomRefinements,
                  bool useTriangles, bool useHybrid)
```

The logarithms in `minLogElements` and `maxLogElements` are base 2, and the number of elements is measured in one axis direction only — for example, specifying 0 and 5 will compute solutions for meshes varying in size from 1×1 to 32×32 . `H1Order` is the polynomial order for H^1 space, which is one higher than that used for L^2 . The `randomRefinements` flag will vary the `H1Order` throughout the mesh; these are not actually randomly chosen — the flag is meant for exercising p -refinements. The other arguments are self-explanatory.

Calling `solve(quadBoundaryPoints)`, where the `quadBoundaryPoints` argument is such as was used in `buildQuadMesh()`, will compute solutions on

a series of meshes. Calling `writeToFiles(filePathPrefix, trialID)` will print error values and convergence rates for `trialID` to console, and write both the error values and the visualization data for the solution to a set of files starting with `filePathPrefix`. Example drivers for `HConvergenceStudy` can be found in `PoissonStudy` and `StokesStudy`.

7.5 Visualization

For visualization in MATLAB, first call the `Solution` object's `writeToFile()` method, passing in the trial variable identifier and the file path as arguments. Our toolbox provides a MATLAB script, `plotSolution.m`, for displaying the solution. Calling this is as simple as ensuring that the MATLAB script is in your MATLAB path, and then calling `plotSolution(filePath)` from MATLAB. The visualization is approximate: at present, we only write out solution values at vertices, and MATLAB is responsible for interpolating between these. We do hope to provide higher-fidelity visualization soon.

8 Stokes Formulations

In Section 3, we developed an ultra-weak variational formulation for the Poisson problem. In this section, we do the same for two formulations of the Stokes problem.

8.1 Stokes Formulation I: VSP Formulation

We begin with the formulation we have used in previous work, a velocity-stress-pressure (VSP) formulation. Our past motivation for selecting this one over the velocity-vorticity-pressure (VVP) formulation which we discuss below had to do with details of implementation. The VVP formulation requires considerably fewer solution variables, so now that we have it implemented, we prefer it.

The strong form of the Stokes problem with which we begin is as follows:

$$-2\mu\nabla \cdot \underline{\boldsymbol{\epsilon}} + \nabla p = \mathbf{f} \quad \text{in } \Omega, \quad (8.1)$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega, \quad (8.2)$$

$$\mathbf{u} = \mathbf{g}_D \quad \text{on } \partial\Omega, \quad (8.3)$$

where $\Omega \subset \mathbb{R}^2$, μ is viscosity, $\underline{\boldsymbol{\epsilon}} = \nabla^{\text{sym}}\mathbf{u}$ is strain, p is pressure, \mathbf{u} velocity, and \mathbf{f} a vector forcing function.

We introduce stress $\boldsymbol{\sigma}$ and vorticity $\boldsymbol{\omega}$ by

$$\begin{aligned} \underline{\boldsymbol{\sigma}} &= 2\mu\underline{\boldsymbol{\epsilon}} - p\underline{\mathbf{I}} \\ \underline{\boldsymbol{\omega}} &= \frac{1}{2}(\nabla\mathbf{u} - \nabla\mathbf{u}^T) \end{aligned}$$

so that equation (8.1) becomes simply $-\nabla \cdot \underline{\boldsymbol{\sigma}} = \mathbf{f}$. We also have

$$\underline{\boldsymbol{\epsilon}} = \frac{1}{2\mu}(\underline{\boldsymbol{\sigma}} + p\underline{\mathbf{I}}).$$

Since $\underline{\boldsymbol{\epsilon}} = \nabla^{\text{sym}}\mathbf{u} = \nabla\mathbf{u} - \underline{\boldsymbol{\omega}}$, the entire system is

$$\begin{aligned} \frac{1}{2\mu}(\underline{\boldsymbol{\sigma}} + p\underline{\mathbf{I}}) - \nabla\mathbf{u} + \underline{\boldsymbol{\omega}} &= \underline{\mathbf{0}} && \text{in } \Omega, \\ -\nabla \cdot \underline{\boldsymbol{\sigma}} &= \mathbf{f} && \text{in } \Omega, \\ \nabla \cdot \mathbf{u} &= 0 && \text{in } \Omega, \\ \mathbf{u} &= \mathbf{g}_D && \text{on } \partial\Omega. \end{aligned}$$

Note that the antisymmetric part of the first equation recovers the definition of $\underline{\omega}$, so that it need not enter the system separately. Define scalar $\omega = \omega_{21} = \frac{1}{2}(u_{1,2} - u_{2,1})$. Our strong formulation is

$$\begin{aligned}
\frac{1}{2\mu} \begin{pmatrix} \sigma_{11} + p \\ \sigma_{21} \end{pmatrix} - \nabla u_1 + \begin{pmatrix} 0 \\ \omega \end{pmatrix} &= \mathbf{0} && \text{in } \Omega, \\
\frac{1}{2\mu} \begin{pmatrix} \sigma_{12} \\ \sigma_{22} + p \end{pmatrix} - \nabla u_2 - \begin{pmatrix} \omega \\ 0 \end{pmatrix} &= \mathbf{0} && \text{in } \Omega, \\
-\nabla \cdot \begin{pmatrix} \sigma_{11} \\ \sigma_{21} \end{pmatrix} &= f_1 && \text{in } \Omega, \\
-\nabla \cdot \begin{pmatrix} \sigma_{12} \\ \sigma_{22} \end{pmatrix} &= f_2 && \text{in } \Omega, \\
\nabla \cdot \mathbf{u} &= 0 && \text{in } \Omega, \\
\mathbf{u} &= \mathbf{g}_D && \text{on } \partial\Omega.
\end{aligned}$$

Multiplying the first two equations by vector test functions \mathbf{q}_i and the following three by scalar test functions v_i , and integrating by parts over an element K , we obtain

$$\begin{aligned}
\int_K \left(\frac{1}{2\mu} \begin{pmatrix} \sigma_{11} + p \\ \sigma_{21} \end{pmatrix} + \begin{pmatrix} 0 \\ \omega \end{pmatrix} \right) \cdot \mathbf{q}_1 + \int_K u_1 \nabla \cdot \mathbf{q}_1 - \int_{\partial K} \hat{u}_1 \mathbf{q}_1 \cdot \mathbf{n} &= \mathbf{0} \\
\int_K \left(\frac{1}{2\mu} \begin{pmatrix} \sigma_{12} \\ \sigma_{22} + p \end{pmatrix} - \begin{pmatrix} \omega \\ 0 \end{pmatrix} \right) \cdot \mathbf{q}_2 + \int_K u_2 \nabla \cdot \mathbf{q}_2 - \int_{\partial K} \hat{u}_2 \mathbf{q}_2 \cdot \mathbf{n} &= \mathbf{0} \\
\int_K \begin{pmatrix} \sigma_{11} \\ \sigma_{21} \end{pmatrix} \cdot \nabla v_1 - \int_{\partial K} \hat{\sigma}_{1n} v_1 &= \int_K f_1 v_1 \\
\int_K \begin{pmatrix} \sigma_{12} \\ \sigma_{22} \end{pmatrix} \cdot \nabla v_2 - \int_{\partial K} \hat{\sigma}_{2n} v_2 &= \int_K f_2 v_2 \\
-\int_K \mathbf{u} \cdot \nabla v_3 + \int_{\partial K} \hat{\mathbf{u}} v_3 \cdot \mathbf{n} &= 0,
\end{aligned}$$

where the hatted variables (\hat{u}_1 , e.g.) are fluxes and traces introduced by relaxing the continuity requirement at element boundaries. These differ from the *numerical fluxes* that appear in other DG methods, in that they are not constructed a priori, but simply enter the variational problem as additional unknowns. We solve for them at the same time as we solve the rest of the unknowns. As in other DG methods, the fluxes will approach the corresponding unhatted solution variables (which we call *field variables*) as the latter approach the exact solution.

Conceptually, this requires nothing more than the Poisson formulation we described above — in particular, the test functions $\mathbf{q}_i \in H(\text{div})$ and $v_i \in H(\text{grad})$, so we do not require any test spaces or operators different than discussed above. An implementation of this bilinear form can be found in `StokesBilinearForm`.

8.2 Stokes Formulation II: Velocity-Vorticity-Pressure Formulation

The standard velocity-vorticity-pressure (VVP) Stokes formulation is:

$$\begin{aligned}\nabla \times \omega + \nabla P &= \mathbf{f} \\ \omega - \nabla \times \mathbf{u} &= 0 \\ \nabla \cdot \mathbf{u} &= 0.\end{aligned}$$

(Note that the value of ω here differs by a factor of $-\frac{1}{2}$ from the ω as defined in the previous formulation.) Multiplying by test functions, integrating by parts, and substituting fluxes and traces for boundary values, we obtain:

$$\begin{aligned}\int_{\partial K} \widehat{\omega} \mathbf{q} \cdot \begin{pmatrix} n_2 \\ -n_1 \end{pmatrix} + \int_{\partial K} \widehat{P} q_n - \int_K \omega \nabla \times \mathbf{q} - \int_K P \nabla \cdot \mathbf{q} &= \int_K \mathbf{f} \cdot \mathbf{q} \\ \int_{\partial K} \widehat{u}_{\times n} v_1 - \int_K \mathbf{u} \cdot (\nabla \times v_1) + \int_K \omega v_1 &= 0 \\ \int_{\partial K} \widehat{u}_n v_2 - \int_K \mathbf{u} \cdot (\nabla v_2) &= 0.\end{aligned}$$

Whereas the original Stokes formulation required 7 field variables, 3 traces, and 2 fluxes, the VVP formulation requires just 4 field, 2 trace, and two flux variables. However, there are a few new operators and spaces required. While $v_i \in H(\text{grad})$ as before, now we require $\mathbf{q} \in H(\text{div})$ and $\mathbf{q} \in H(\text{curl})$. We can support this by requiring $\mathbf{q} \in H(\text{grad}) \times H(\text{grad})$. In the code, we define the function space for \mathbf{q} as `FUNCTION_SPACE_VECTOR_HGRAD`. The curl operator is given by `OPERATOR_CURL`, and the

$$\cdot \begin{pmatrix} n_2 \\ -n_1 \end{pmatrix} = \times n$$

operator is given by `OPERATOR_CROSS_N`. The implementation for this formulation can be found in `StokesVVPBilinearForm`.

9 Numerical Results

For each of our three formulations — Poisson, VSP Stokes, and VVP Stokes — we have run a set of numerical experiments:

- For L^2 polynomial orders 1, 2, and 3, run convergence studies on meshes varying from 1×1 to 32×32 .
- For a 16×16 mesh, vary the polynomial orders of the elements across the mesh.
- For L^2 polynomial orders 1, 2, and 3, run convergence studies on “hybrid” meshes (quads and triangles together) varying in size from 1×1 to 32×32 .

The first two experiments we ran with triangles and quads. For all of the experiments, we used the zero mean constraint and the domain $(-1, 1) \times (-1, 1)$, and the mathematician’s inner product. We used the Poisson manufactured solution

$$\phi = e^{x \sin y} - \frac{1}{4} \int_{-1}^1 \int_{-1}^1 e^{x \sin y} dx dy,$$

where the subtracted integral is chosen to ensure that ϕ does indeed have a zero mean. Following our previous work, the Stokes experiments used a manufactured solution

$$\begin{aligned} u_1 &= -e^x (y \cos y + \sin y) \\ u_2 &= e^x y \sin y \\ p &= 2\mu e^x \sin y. \end{aligned}$$

9.1 Convergence Studies on Uniform Meshes

For polynomial order k , we expect a convergence rate of $k + 1$. This is indeed what we see in our experiments. The Poisson convergence studies with triangular elements can be found in Table A.1; with quads, in Table A.2. The Stokes VSP studies with triangles can be found in Table A.3; with quads, in Table A.4. The VVP studies with triangles are shown in Table A.5; with quads, in Table A.6.

9.2 Meshes of Multiple Polynomial Orders

To confirm that our code works well with meshes that include elements of varying degree, we took a 16×16 mesh with L^2 polynomial degree assigned according to the following pattern, repeated 4 times:

4	4	4	4	1	1	1	1	2	2	2	2	3	3	3	3
3	3	3	3	4	4	4	4	1	1	1	1	2	2	2	2
2	2	2	2	3	3	3	3	4	4	4	4	1	1	1	1
1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4

We hope to see errors as good as, or better than, our first-order 16×16 mesh for the same problem, although we can perhaps explain a small amount of increased error as due to worse conditioning of the matrices for higher-order polynomials.

The results for Poisson are:

Triangles					
ϕ		ψ_1		ψ_2	
$k = 1$	mixed k	$k = 1$	mixed k	$k = 1$	mixed k
2.0e-3	9.1e-4	3.4e-3	1.7e-3	2.4e-3	1.1e-3
Quads					
ϕ		ψ_1		ψ_2	
$k = 1$	mixed k	$k = 1$	mixed k	$k = 1$	mixed k
1.0e-3	3.7e-4	2.3e-3	6.6e-4	2.9e-3	1.2e-3

For Poisson, the multi-order mesh has lower error than the first-order mesh in every variable, just as we would like.

The results for Stokes VSP are:

Triangles					
p		u_1		u_2	
$k = 1$	mixed k	$k = 1$	mixed k	$k = 1$	mixed k
1.6e-2	1.4e-2	5.1e-3	2.4e-3	4.4e-3	2.1e-3
Quads					
p		u_1		u_2	
$k = 1$	mixed k	$k = 1$	mixed k	$k = 1$	mixed k
5.3e-3	1.0e-2	4.9e-3	1.6e-3	2.5e-3	1.3e-3

For Stokes VSP, the multi-order mesh has lower error than the first-order mesh in every variable, except for pressure in the quad mesh, where the first-order mesh does better by about a factor of 2.

The results for Stokes VVP are:

Triangles					
p		u_1		u_2	
$k = 1$	mixed k	$k = 1$	mixed k	$k = 1$	mixed k
6.9e-2	5.8e-2	6.9e-3	3.1e-3	6.8e-3	2.7e-3
Quads					
p		u_1		u_2	
$k = 1$	mixed k	$k = 1$	mixed k	$k = 1$	mixed k
8.0e-3	2.1e-2	3.9e-3	1.7e-3	3.8e-3	1.4e-3

Again, the behavior is just as we would like, except in the case of pressure for quads, where the error in the first-order mesh is better than that for the multi-order, this time by a factor of about 2.5. We believe that this is due to round-off error, but we do not at present have a precise explanation.

9.3 “Hybrid” Mesh Convergence Studies

We studied the convergence of the Poisson solution for “hybrid” meshes, containing half triangles and half quads, ranging from 1×1 to 32×32 , with L^2 polynomial orders $k = 1, 2, 3$. The results can be seen in Table B.1; again, the convergence rates for all variables are optimal. Plots of ϕ , ψ_1 and ψ_2 for the 16×16 mesh can be found in Figure C.1.

We studied the same for the VSP and VVP Stokes formulations. VSP results can be found in Table B.2; VVP, Table B.3 — the rates are optimal for u_1 and u_2 across the board, but for reasons unknown the $k = 1$ VVP pressure is somewhat sub-optimal, while for $k = 2$ and 3, the rate is somewhat super-optimal. Something similar happens in the VSP pressure, although there the effect is considerably less pronounced. Plots of P , u_1 and u_2 for the 16×16 mesh can be found in Figure C.2. These were generated using the VVP formulation; the VSP plots are visually identical.

10 Conclusions and Future Work

We have implemented a general toolbox for solving DPG problems, together with proof-of-concept implementations of solvers for the Poisson and Stokes problems. We look forward to extending this further. Here is a short list of features we would like to add, and experiments we would like to perform, going forward:

- Quotient space norm computations — when we impose the pressure at a point rather than imposing a zero-mean condition, the L^2 norm of the error is not the correct measure. We need to subtract off the mean value of both the exact solution and that of the computed solution before taking the L^2 norm of the difference.
- Support for solving nonlinear problems using Newton-Raphson — in the end, we hope to use this toolbox to solve the compressible Navier-Stokes equations in two dimensions.
- p -adaptivity — we have p -refinements in place, so this is just a matter of computing the error and refining where the error is highest.
- h -adaptivity — this will require us to add support for h -refinements. We have a good idea how we will do so.
- Mesh-dependent inner products — at present, our inner products are computed without any knowledge of element sizes; in some other DPG explorations, we have used various kinds of weighted inner product, so we would like to support these.
- Support for 1D computations. At present, only 2D is supported by the mesh, but it should be a relatively simple matter to add support for 1D. 3D will be more involved, but is tractable.
- MPI support. At present, the bulk of the computational cost in our numerical experiments is devoted to computing the optimal test functions. Because these are local computations, they should scale easily to many processors.

References

- [1] P. BOCHEV AND R. B. LEHOUCQ, *On the finite element solution of the pure Neumann problem*, SIAM Review, 47 (2005), pp. 55–66.
- [2] P. B. BOCHEV, R. C. KIRBY, K. J. PETERSON, AND D. RIDZAL, *Intrepid project*, <http://trilinos.sandia.gov/packages/intrepid/>.
- [3] J. CHAN, L. DEMKOWICZ, R. MOSER, AND N. ROBERTS, *A new discontinuous Petrov-Galerkin method with optimal test functions. Part V: Solution of 1D Burgers' and Navier-Stokes equations*, ICES Technical Report, (2010).
- [4] L. DEMKOWICZ AND J. GOPALAKRISHNAN, *A class of discontinuous Petrov-Galerkin methods. Part I: The transport equation*, Computer Methods in Applied Mechanics and Engineering, 199 (2010), pp. 1558 – 1572.
- [5] L. D. DEMKOWICZ AND J. GOPALAKRISHNAN, *A class of discontinuous Petrov-Galerkin methods. Part II: Optimal test functions.*, ICES Technical Report, (2009).
- [6] L. D. DEMKOWICZ, J. GOPALAKRISHNAN, AND A. NIEMI, *A class of discontinuous Petrov-Galerkin methods. Part III: Adaptivity*, (2010).
- [7] M. A. HEROUX, R. A. BARTLETT, V. E. HOWLE, R. J. HOEKSTRA, J. J. HU, T. G. KOLDA, R. B. LEHOUCQ, K. R. LONG, R. P. PAWLOWSKI, E. T. PHIPPS, A. G. SALINGER, H. K. THORNQUIST, R. S. TUMINARO, J. M. WILLENBRING, A. WILLIAMS, AND K. S. STANLEY, *An overview of the Trilinos project*, ACM Trans. Math. Softw., 31 (2005), pp. 397–423.
- [8] T. J. HUGHES, G. SCOVAZZI, P. B. BOCHEV, AND A. BUFFA, *A multiscale discontinuous Galerkin method with the computational structure of a continuous Galerkin method*, Computer Methods in Applied Mechanics and Engineering, 195 (2006), pp. 2761 – 2787.
- [9] J. ZITELLI, I. MUGA, L. DEMKOWICZ, J. GOPALAKRISHNAN, D. PARDO, AND V. CALO, *A class of discontinuous Petrov-Galerkin methods. Part IV: Wave propagation*, (2010).

A Results of Convergence Studies on Uniform Meshes

k=1						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	4.2e-1	-	5.2e-1	-	3.4e-1	-
2 × 2	1.3e-1	1.71	2.0e-1	1.38	1.5e-1	1.24
4 × 4	3.2e-2	2.00	5.2e-2	1.93	3.9e-2	1.92
8 × 8	8.0e-3	2.01	1.3e-2	1.96	9.6e-3	2.00
16 × 16	2.0e-3	2.00	3.4e-3	1.99	2.4e-3	2.01
32 × 32	5.0e-4	2.00	8.4e-4	1.99	6.0e-4	2.00
k=2						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	7.9e-2	-	2.5e-1	-	1.4e-1	-
2 × 2	1.2e-2	2.77	3.1e-2	2.97	3.6e-2	1.96
4 × 4	1.4e-3	3.00	4.1e-3	2.95	4.9e-3	2.86
8 × 8	1.8e-4	2.99	5.1e-4	2.98	6.0e-4	3.02
16 × 16	2.3e-5	3.00	6.5e-5	2.99	7.5e-5	3.01
32 × 32	2.8e-6	3.00	8.1e-6	3.00	9.3e-6	3.01
k=3						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	2.5e-2	-	2.9e-2	-	4.3e-2	-
2 × 2	1.5e-3	3.99	3.4e-3	3.09	5.2e-3	3.06
4 × 4	1.1e-4	3.82	2.3e-4	3.92	3.4e-4	3.95
8 × 8	7.1e-6	3.94	1.5e-5	3.99	2.1e-5	3.98
16 × 16	4.5e-7	3.99	9.2e-7	3.99	1.3e-6	4.00
32 × 32	2.8e-8	4.00	5.8e-8	4.00	8.4e-8	4.00

Table A.1. Poisson: Triangles, L^2 Error and h -Convergence Rates. We observe optimal convergence.

k=1						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	1.4e-1	-	5.4e-1	-	3.8e-1	-
2 × 2	5.2e-2	1.41	1.3e-1	2.00	1.4e-1	1.41
4 × 4	1.5e-2	1.77	3.5e-2	1.92	4.3e-2	1.70
8 × 8	4.1e-3	1.93	9.1e-3	1.97	1.2e-2	1.91
16 × 16	1.0e-3	1.98	2.3e-3	2.00	2.9e-3	1.98
32 × 32	2.6e-4	2.00	5.7e-4	2.00	7.3e-4	2.00
k=2						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	4.9e-2	-	8.5e-2	-	1.1e-1	-
2 × 2	6.6e-3	2.91	1.7e-2	2.34	1.6e-2	2.78
4 × 4	7.8e-4	3.08	2.2e-3	2.90	1.8e-3	3.12
8 × 8	9.3e-5	3.05	2.6e-4	3.11	2.0e-4	3.20
16 × 16	1.2e-5	3.01	3.1e-5	3.06	2.3e-5	3.11
32 × 32	1.4e-6	3.00	3.8e-6	3.03	2.8e-6	3.05
k=3						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	1.2e-2	-	3.0e-2	-	2.6e-2	-
2 × 2	6.6e-4	4.16	2.6e-3	3.54	2.0e-3	3.66
4 × 4	3.3e-5	4.33	1.2e-4	4.39	1.1e-4	4.18
8 × 8	2.1e-6	3.99	7.3e-6	4.09	6.6e-6	4.10
16 × 16	1.3e-7	3.99	4.4e-7	4.04	3.9e-7	4.07
32 × 32	8.1e-9	4.00	2.7e-8	4.02	2.4e-8	4.04

Table A.2. Poisson: Quads, L^2 Error and h -Convergence Rates. We observe optimal convergence.

k=1						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	3.2e-0	-	9.4e-1	-	1.6e-0	-
2×2	1.1e-0	1.53	3.1e-1	1.62	2.8e-1	2.05
4×4	3.0e-1	1.89	8.0e-2	1.94	7.0e-2	1.99
8×8	6.9e-2	2.11	2.0e-2	1.99	1.8e-2	2.00
16×16	1.6e-2	2.12	5.1e-3	2.00	4.4e-3	2.00
32×32	3.8e-3	2.08	1.3e-3	2.00	1.1e-3	2.00
k=2						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	2.3e-0	-	2.6e-1	-	4.3e-1	-
2×2	2.8e-1	3.02	4.3e-2	2.63	5.1e-2	3.05
4×4	3.4e-2	3.06	5.8e-3	2.88	6.6e-3	2.97
8×8	4.0e-3	3.10	7.4e-4	2.97	8.3e-4	2.99
16×16	4.7e-4	3.06	9.3e-4	2.99	1.0e-4	3.00
32×32	5.8e-5	3.03	1.2e-5	3.00	1.3e-5	3.00
k=3						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	3.7e-1	-	5.0e-2	-	3.7e-2	-
2×2	3.4e-2	3.46	4.0e-3	3.66	3.4e-3	3.44
4×4	2.6e-3	3.70	2.6e-4	3.95	2.2e-4	3.94
8×8	1.9e-4	3.82	1.6e-5	3.99	1.4e-5	4.00
16×16	1.1e-5	4.06	1.0e-6	4.00	8.8e-7	4.00
32×32	5.8e-7	4.27	6.4e-8	4.00	5.5e-8	4.00

Table A.3. Stokes VSP: Triangles, L^2 Error and h -Convergence Rates. We observe optimal convergence.

k=1						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	1.1e-0	-	4.7e-1	-	8.3e-1	-
2×2	6.1e-1	0.84	3.0e-1	0.65	1.7e-1	2.33
4×4	1.4e-1	2.07	7.8e-2	1.94	4.0e-2	2.03
8×8	2.9e-2	2.33	2.0e-2	1.99	1.0e-2	2.01
16×16	5.4e-3	2.41	4.9e-3	2.00	2.5e-3	2.00
32×32	1.0e-3	2.38	1.2e-3	2.00	6.3e-4	2.00
k=2						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	3.3e-1	-	3.0e-1	-	1.7e-1	-
2×2	1.8e-1	0.82	3.2e-2	3.22	2.0e-2	3.06
4×4	1.5e-2	3.67	3.9e-3	3.04	2.5e-3	3.02
8×8	1.2e-3	3.58	4.8e-4	3.01	3.1e-4	3.01
16×16	1.0e-4	3.54	6.0e-5	3.00	3.9e-5	3.00
32×32	9.5e-6	3.46	7.6e-6	3.00	4.9e-6	3.00
k=3						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	2.7e-1	-	1.3e-2	-	3.3e-2	-
2×2	2.0e-2	3.74	1.7e-3	2.91	1.8e-3	4.17
4×4	1.3e-3	4.02	1.1e-4	3.96	1.1e-4	4.05
8×8	7.0e-5	4.16	6.8e-6	3.99	6.9e-6	4.02
16×16	3.6e-6	4.29	4.3e-7	4.00	4.3e-7	4.00
32×32	1.6e-7	4.52	2.7e-8	4.00	2.7e-8	4.00

Table A.4. Stokes VSP: Quads, L^2 Error and h -Convergence Rates. We observe optimal convergence.

k=1						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	3.0e-0	-	1.6e-0	-	2.0e-0	-
2×2	1.5e-0	1.00	4.9e-1	1.72	4.2e-1	2.23
4×4	6.4e-1	1.23	1.1e-1	2.14	1.1e-1	1.90
8×8	2.3e-1	1.51	2.8e-2	2.00	2.8e-2	2.02
16×16	6.9e-2	1.70	6.9e-3	2.00	6.8e-3	2.02
32×32	2.0e-2	1.77	1.7e-3	2.00	1.7e-3	2.01
k=2						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	2.1e-0	-	3.3e-1	-	4.6e-1	-
2×2	8.5e-2	4.65	5.5e-2	2.58	5.6e-2	3.04
4×4	8.3e-3	3.35	7.3e-3	2.90	7.7e-3	2.85
8×8	7.7e-4	3.44	9.3e-4	2.97	1.0e-3	2.96
16×16	6.8e-5	3.51	1.2e-4	2.99	1.3e-4	2.99
32×32	6.9e-6	3.30	1.5e-5	3.00	1.6e-5	3.00
k=3						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	1.7e-1	-	5.9e-2	-	5.4e-2	-
2×2	1.3e-2	3.74	4.4e-3	3.74	4.2e-3	3.68
4×4	8.9e-4	3.83	2.9e-4	3.93	2.7e-4	3.96
8×8	4.8e-5	4.22	1.8e-5	3.99	1.7e-5	3.99
16×16	2.0e-6	4.60	1.1e-6	4.00	1.1e-6	4.00
32×32	6.7e-8	4.89	7.2e-8	4.00	6.4e-8	4.00

Table A.5. Stokes VVP: Triangles, L^2 Error and h -Convergence Rates. We observe optimal convergence.

k=1						
Mesh Size	p	rate	u_1	rate	u_2	rate
1 × 1	1.5e-0	-	4.9e-1	-	1.5e-0	-
2 × 2	1.5e-0	0.02	3.6e-1	0.45	3.1e-1	2.26
4 × 4	1.0e-1	3.90	6.5e-2	2.45	6.3e-2	2.28
8 × 8	3.0e-2	1.74	1.6e-2	2.05	1.5e-2	2.04
16 × 16	8.0e-3	1.91	3.9e-3	2.01	3.8e-3	2.01
32 × 32	2.0e-3	1.97	9.8e-4	2.00	9.6e-4	2.00
k=2						
Mesh Size	p	rate	u_1	rate	u_2	rate
1 × 1	2.1e-0	-	3.9e-1	-	2.7e-1	-
2 × 2	5.1e-2	5.33	3.3e-2	3.56	2.2e-2	3.60
4 × 4	3.7e-3	3.80	3.9e-3	3.07	2.6e-3	3.06
8 × 8	3.5e-4	3.41	4.9e-4	3.02	3.3e-4	3.01
16 × 16	3.8e-5	3.18	6.1e-5	3.00	4.1e-5	3.00
32 × 32	4.4e-6	3.12	7.6e-6	3.00	5.1e-6	3.00
k=3						
Mesh Size	p	rate	u_1	rate	u_2	rate
1 × 1	1.9e-1	-	1.3e-2	-	3.7e-2	-
2 × 2	1.3e-2	3.82	1.8e-3	2.85	2.0e-3	4.23
4 × 4	4.7e-4	4.83	1.1e-4	3.97	1.1e-4	4.11
8 × 8	1.9e-5	4.59	7.1e-6	4.00	7.1e-6	4.02
16 × 16	1.0e-6	4.24	4.4e-7	4.00	4.4e-7	4.00
32 × 32	6.3e-8	4.04	2.8e-8	4.00	2.8e-8	4.00

Table A.6. Stokes VVP: Quads, L^2 Error and h -Convergence Rates. We observe optimal convergence.

B Results of Convergence Studies on Hybrid Meshes

k=1						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	4.2e-1	-	5.2e-1	-	3.4e-1	-
2 × 2	9.6e-2	2.12	1.7e-1	1.58	1.4e-1	1.27
4 × 4	2.4e-2	1.99	4.5e-2	1.93	4.0e-2	1.84
8 × 8	6.1e-3	2.00	1.2e-2	1.97	1.0e-2	1.96
16 × 16	1.5e-3	2.00	2.9e-3	1.99	2.6e-3	1.99
32 × 32	3.8e-4	2.00	7.3e-4	2.00	6.4e-4	2.00
k=2						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	7.9e-2	-	2.5e-1	-	1.4e-1	-
2 × 2	9.5e-3	3.06	2.5e-2	3.29	2.8e-2	2.33
4 × 4	1.2e-3	3.03	3.3e-3	2.94	3.7e-3	2.90
8 × 8	1.4e-4	3.01	4.1e-4	3.01	4.5e-4	3.04
16 × 16	1.8e-5	3.00	5.1e-5	3.01	5.5e-5	3.02
32 × 32	2.3e-6	3.00	6.3e-6	3.00	6.9e-6	3.01
k=3						
Mesh Size	ϕ	rate	ψ_1	rate	ψ_2	rate
1 × 1	2.5e-2	-	2.9e-2	-	4.3e-2	-
2 × 2	1.2e-3	4.37	3.0e-3	3.27	4.0e-3	3.45
4 × 4	8.1e-5	3.88	1.8e-4	4.05	2.5e-4	3.98
8 × 8	5.2e-6	3.95	1.2e-5	3.99	1.6e-5	3.99
16 × 16	3.3e-7	3.99	7.2e-7	4.00	9.9e-7	4.00
32 × 32	2.1e-8	4.00	4.4e-8	4.00	6.2e-8	4.00

Table B.1. Poisson: “Hybrid” Mesh, L^2 Error and h -Convergence Rates. We observe optimal convergence.

k=1						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	3.2e-0	-	9.4e-1	-	1.2e-0	-
2×2	1.1e-0	1.58	3.1e-1	1.62	2.7e-1	2.10
4×4	2.9e-1	1.88	8.0e-2	1.94	6.8e-2	1.99
8×8	6.7e-2	2.12	2.0e-2	1.99	1.7e-2	2.00
16×16	1.5e-2	2.14	5.0e-3	2.00	4.2e-3	2.00
32×32	3.6e-3	2.09	1.3e-3	2.00	1.1e-3	2.00
k=2						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	2.3e-0	-	2.6e-1	-	4.3e-1	-
2×2	2.8e-1	3.02	4.1e-2	2.67	4.9e-2	3.12
4×4	3.2e-2	3.13	5.6e-3	2.89	6.3e-3	2.98
8×8	3.8e-3	3.11	7.1e-4	2.97	7.9e-4	2.99
16×16	4.5e-4	3.07	9.0e-6	2.99	9.9e-5	3.00
32×32	5.5e-5	3.03	1.1e-5	3.00	1.2e-5	3.00
k=3						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	3.7e-1	-	5.0e-2	-	3.7e-2	-
2×2	3.3e-2	3.51	3.8e-3	3.73	3.3e-3	3.50
4×4	2.5e-3	3.69	2.5e-4	3.95	2.1e-4	3.94
8×8	1.8e-4	3.80	1.6e-5	3.99	1.3e-5	4.00
16×16	1.1e-5	4.06	9.7e-7	4.00	8.4e-7	4.00
32×32	5.5e-7	4.30	6.1e-8	4.00	5.2e-8	4.00

Table B.2. Stokes VSP: “Hybrid” Mesh, L^2 Error and h -Convergence Rates. Rates are close to optimal.

k=1						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	3.0e-0	-	1.6e-0	-	2.0e-0	-
2×2	1.2e-0	1.30	4.6e-1	1.80	4.1e-1	2.25
4×4	6.2e-1	0.97	1.1e-1	2.12	1.1e-1	1.93
8×8	2.2e-1	1.48	2.7e-2	1.99	2.7e-2	2.03
16×16	6.9e-2	1.70	6.6e-3	2.00	6.6e-3	2.02
32×32	2.0e-2	1.77	1.7e-3	2.00	1.6e-3	2.01
k=2						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	2.1e-0	-	3.3e-1	-	4.6e-1	-
2×2	8.8e-2	4.60	5.2e-2	2.64	5.3e-2	3.12
4×4	8.6e-3	3.35	7.0e-3	2.91	7.3e-3	2.85
8×8	7.7e-4	3.48	8.9e-4	2.98	9.4e-4	2.96
16×16	6.7e-5	3.53	1.1e-4	2.99	1.2e-4	2.99
32×32	6.7e-6	3.32	1.4e-5	3.00	1.5e-5	3.00
k=3						
Mesh Size	p	rate	u_1	rate	u_2	rate
1×1	1.7e-1	-	5.9e-2	-	5.4e-2	-
2×2	1.3e-2	3.74	4.2e-3	3.81	4.0e-3	3.75
4×4	8.9e-4	3.83	2.8e-4	3.93	2.6e-4	3.97
8×8	4.8e-5	4.22	1.7e-5	3.99	1.6e-5	4.00
16×16	2.0e-6	4.59	1.1e-6	4.00	1.0e-6	4.00
32×32	6.7e-8	4.89	6.8e-8	4.00	6.3e-8	4.00

Table B.3. Stokes VVP: “Hybrid” Mesh, L^2 Error and h -Convergence Rates. Rates are close to optimal.

C Solution Plots

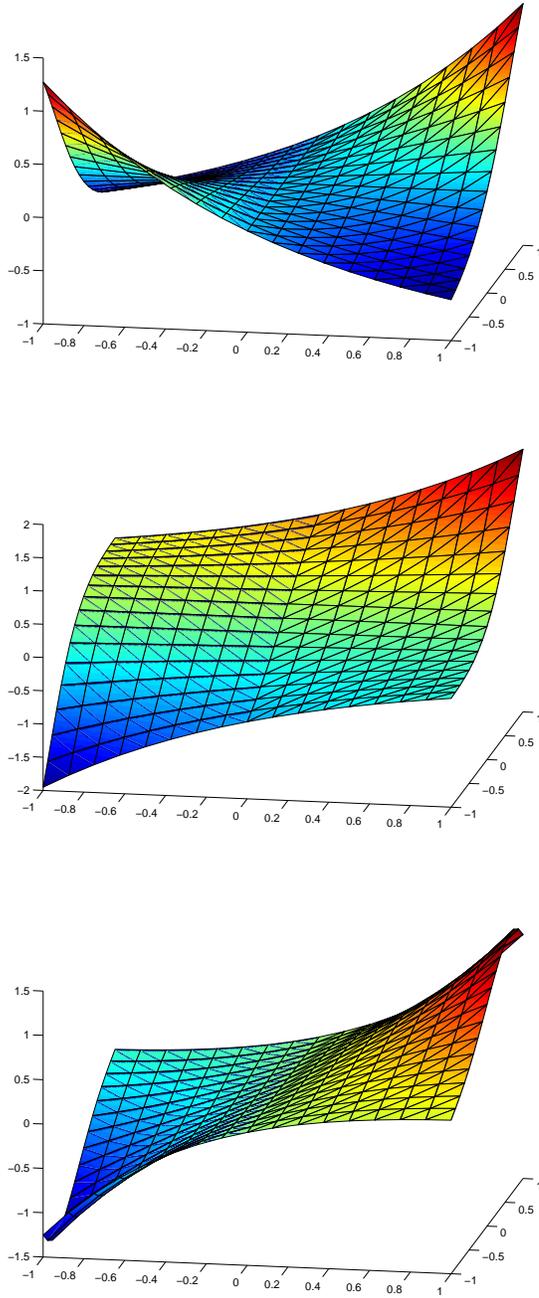


Figure C.1. Solution of Poisson with quads and triangles (cubic elements, 16×16 mesh): ϕ (top pane), ψ_1 (middle pane) ψ_2 (bottom pane).

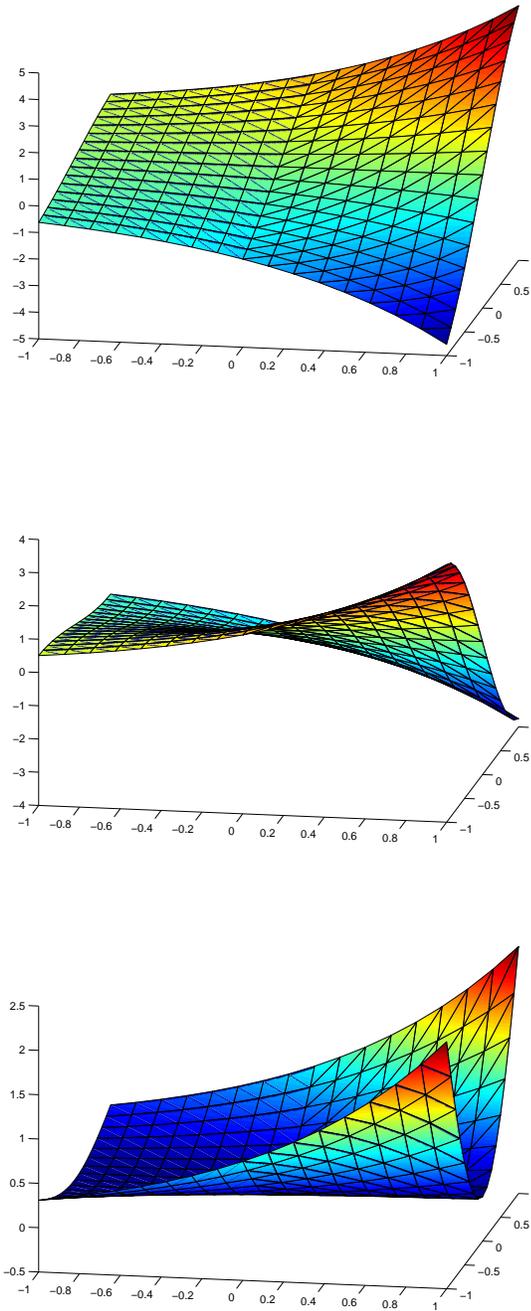


Figure C.2. Solution of Stokes (VVP) with quads and triangles (cubic elements, 16×16 mesh): P (top pane), u_1 (middle pane), u_2 (bottom pane).

DISTRIBUTION:

- 1 MS 1320
Pavel Bochev, 1442
- 1 MS 1320
Eric Cyr, 1426
- 1 MS 1320
Denis Ridzal, 1441
- 1 MS 1320
Joseph Young, 1442
- 1 MS 0899
Technical Library
(electronic copy), 9536

