Sundance Example 1: Laplace's Equation in 3D

Kevin Long

March 29, 2011

1 Problem definition

This program will solve a linear boundary value problem in 3D: Laplace's equation

$$\nabla^2 u = 0$$

on a thin square plate with a circular through-hole in the center. The geometry of this plate is shown in figures 1 and 2. For boundary conditions, we will specify Dirichlet conditions

$$u = 0 \quad \text{on the west edge of the plate}$$

$$u = 1 \quad \text{on the east edge of the plate}$$

and homogeneous Neumann conditions

$$\frac{\partial u}{\partial n} = 0$$

on all other surfaces.

The Galerkin weak form of this problem is

$$\int_\Omega \nabla v \cdot \nabla u \, d\Omega - \int_{\partial\Omega} v\mathbf{n} \cdot \nabla u \, dA = 0 \quad \forall v \in H_0^1$$

$$u = 0 \ \text{ on west}$$

$$u = 1 \ \text{ on east.}$$

In our program we'll represent this weak form in terms of symbolic expression objects called `Exprs`. As a basis for both the unknown function u and the test function v, we will use the first-degree Lagrange functions on tetrahedral elements. The integrals will be computed using Gauss-Dunavant quadrature.

The resulting system of equations

$$K\mathbf{u} = \mathbf{b}$$

is linear and must be solved with some linear solver algorithm. Sundance interfaces with linearsolvers through the Playa `LinearSolver` interface; most Trilinos solver libraries have an adapter letting them be used through Playa.

The solution vector is returned wrapped in an `Expr` object. As such, it can be used in other symbolic expressions, for example, expressions that define post-processing steps such as flux calculations. Finally, it may be given to a `FieldWriter` object that writes the solution to an output file in a format such as VTK or Exodus.

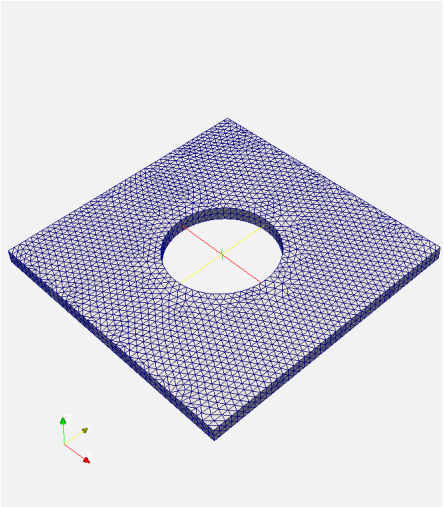

Figure 1: 3D view of meshed plate with hole

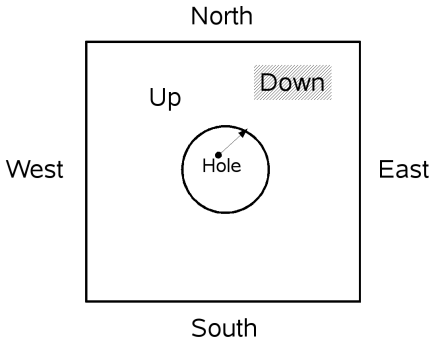

Figure 2: Schematic of labeled surfaces on the plate with hole

2 Structure of a simulation program

2.1 Initialization and finalization code

A dull but essential first step is to show the boilerplate C++ common to nearly every Sundance program:

```
#include "Sundance.hpp"
int main(int argc, void** argv)
{
  try
  {
    Sundance::init(argc, argv);

        /* code body goes here */
  }
  catch(exception& e)
  {
    Sundance::handleException(__FILE__, e);
  }
  Sundance::finalize();
}
```

These lines control initialization and result gathering for profiling timers, initializing and finalizing MPI if MPI is being used, and other administrative tasks. The body of the code – everything else we discuss here – goes in place of the comment `code body goes here`.

2.2 Overview of problem setup and solution

Before diving into code, let's take a coarse-grained look at the steps involved in setting up and solving a linear boundary value problem.

1. Do initialization steps
2. Create the objects that define the problem's geometry
3. Create the symbolic objects that will be used in the equation specification
4. Define the weak form and boundary conditions
5. Create a "problem" object that encapsulates the equations, boundary conditions, and geometry along with a specification of ordering of unknowns
6. Create a solver object
7. Solve the problem
8. Do postprocessing and/or visualization output
9. Do finalization steps

In more complex problems there may be loops over one or more of these steps; for example, a time integration will involve a loop over many solution steps, with visualization output being done at selected intervals.

2.3 Getting a mesh

Sundance uses a `Mesh` object to represent a discretization of the problem's geometric domain. There are many ways of getting a mesh; simple meshes might be built on the fly at runtime, more complex meshes will need to be build offline and read from a file. There are then numerous mesh file formats. To accomodate the diversity of mesh creation mechanisms, Sundance uses an abstract `MeshSource` interface. Different mesh creation modes are represented as subtypes that implement this abstract interface.

Sundance is designed to work with different mesh underlying implementations, the choice of which is done by specifying a `MeshType` object.

In this example we read a mesh that's been stored in the Exodus format. The file is named `plateWithHole.exo`.

```
MeshType meshType = new BasicSimplicialMeshType();
MeshSource meshReader = new ExodusMeshReader("plateWithHole", meshType);
Mesh mesh = mesher.getMesh();
```

2.4 Defining geometric subdomains

We'll need to specify subsets of the mesh on which equations or boundary conditions are defined. In many FEA codes this is done by explicit definition of element blocks, node sets, and side sets. Rather than working with sets explicitly at the user level, we instead work with *filtering rules* that produce sets of cells. These rules are represented by `CellFilter` objects. You can think of a cell filter as an operator that acts on a mesh and returns a set of cells.

First we define a cell filter that identifies all cells of maximal dimension:

```
/* Filter subtype MaximalCellFilter selects all cells having dimension equal to
   the spatial dimension of the mesh */
CellFilter interior = new MaximalCellFilter();
```

Next we define filters that identify the various boundary surfaces. In this example, boundary surfaces are specified by labels assigned to the mesh cells during the process of mesh generation. The `labeledSubset()` member function finds those cells having a specified label.

```
/* DimensionalCellFilter selects all cells of a specified dimension. Here we
   select all 2D faces. Boundary conditions will be applied on certain subsets
   of these. */
CellFilter edges = new DimensionalCellFilter(2);
```

```
CellFilter south = edges.labeledSubset(1);
CellFilter east = edges.labeledSubset(2);
CellFilter north = edges.labeledSubset(3);
CellFilter west = edges.labeledSubset(4);
CellFilter hole = edges.labeledSubset(5);
CellFilter down = edges.labeledSubset(6);
CellFilter up = edges.labeledSubset(7);
```

See figure 2 for a schematic of the various boundary surfaces. In subsequent examples we will see other mechanisms for identifying cells.

2.5 Defining symbolic expressions

An equation is built out of mathematical expressions.

2.5.1 Basis families

Discretization

```
/* Create an object representation of the first-degree Lagrange basis */
BasisFamily basis = new Lagrange(1);
```

2.5.2 Test and unknown functions

With a basis defined, we set up symbolic expressions for the unknown function and test function that appear in the weak form. They are constructed as in this code:

```
Expr u = new UnknownFunction(basis, "u");
Expr v = new TestFunction(basis, "v");
```

The string arguments "u" and "v" are optional and are used only in labeling these functions in diagnostic output. Any label can be used; there is no need for the string's value to be identical to the name of the C++ variable.

2.5.3 Differential operators

Differential operators are also represented as `Expr` objects. The next code fragment shows the construction of partial derivative operators and their aggregation into a gradient.

```
/* Create differential operators and coordinate functions. Directions are
 * indexed starting from zero. The List() function can collect
 * expressions into a vector. */
Expr dx = new Derivative(0);    /* The operator ∂/∂x */
Expr dy = new Derivative(1);    /* The operator ∂/∂y */
Expr dz = new Derivative(2);    /* The operator ∂/∂z */
Expr grad = List(dx, dy, dz);   /* The operator ∇ */
```

2.5.4 Coordinate expressions

```
Expr x = new CoordExpr(0);
Expr y = new CoordExpr(1);
Expr z = new CoordExpr(2);
```

2.6 Equations and boundary conditions

2.6.1 Numerical integration rules

```
/* We need a quadrature rule for doing the integrations */
QuadratureFamily quad2 = new GaussianQuadrature(2);
QuadratureFamily quad4 = new GaussianQuadrature(4);
```

2.6.2 Watching expressions

```
WatchFlag watchMe("watch me");
watchMe.setParam("symbolic preprocessing", 1);
watchMe.setParam("discrete function evaluation", 3);
watchMe.setParam("integration setup", 6);
watchMe.setParam("integration", 6);
watchMe.setParam("fill", 6);
watchMe.setParam("evaluation", 2);
watchMe.deactivate();
```

2.6.3 Integrals

```
/* Write the weak form */
Expr eqn = Integral(interior, (grad*u)*(grad*v), quad2, watchMe);
```

2.6.4 Essential boundary conditions

```
Expr h = new CellDiameterExpr();
Expr bc = EssentialBC(west, v*u/h, quad2)+EssentialBC(east, v*(u-1.0)/h, quad2);
```

2.7 Defining a linear problem

```
LinearProblem prob(mesh, eqn, bc, v, u, vecType);
```

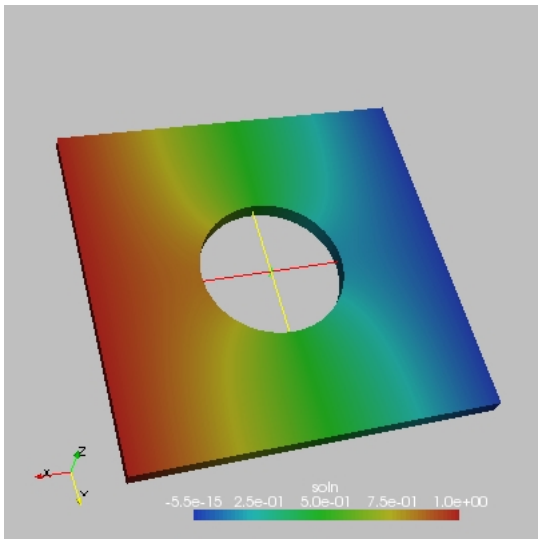

Figure 3: Solution of Laplace's equation on the holed plate.

2.8 Solving a linear problem

2.8.1 Getting a solver

2.8.2 Doing the solve

2.9 Visualization output

```
/* Write the results to a VTK file */
FieldWriter w = new VTKWriter("PoissonDemo3D");
w.addMesh(mesh);
w.addField("soln", new ExprFieldWrapper(soln[0]));
w.write();
```

2.10 Postprocessing

We now show several examples of postprocessing computations using the solution expression `soln`. The first example is the computation of the flux

$$\int_{\text{east+west}} \mathbf{n} \cdot \nabla u \, dA.$$

```
Expr n = CellNormalExpr(3, "n");
Expr fluxExpr = Integral(east + west, (n*grad)*soln, quad2);
double flux = evaluateIntegral(mesh, fluxExpr);
Out::os() << "numerical flux = " << flux << endl;
```

In the next example, we compute the center-of-mass position of the body Ω,

$$x_{CM} = \frac{1}{V(\Omega)} \int_{\Omega} x \, d\Omega$$

and similarly for y_{CM} and z_{CM}.

```
Expr volExpr = Integral(interior, 1.0, quad2);
Expr xCMExpr = Integral(interior, x, quad2);
Expr yCMExpr = Integral(interior, y, quad2);
Expr zCMExpr = Integral(interior, z, quad2);
double vol = evaluateIntegral(mesh, volExpr);
double xCM = evaluateIntegral(mesh, xCMExpr);
double yCM = evaluateIntegral(mesh, yCMExpr);
double zCM = evaluateIntegral(mesh, zCMExpr);
Out::os() << "centroid = (" << xCM << ", " << yCM << ", " << zCM << ")" << endl;
```

We next compute the first Fourier sine coefficient of the solution on the surface of the hole,

$$A_1 = \frac{\int_{\text{hole}} u \sin\phi \, d\Omega}{\int_{\text{hole}} \sin^2\phi \, d\Omega}$$

```
/* Compute sin(phi) from Cartesian coordinates (x,y) */
Expr r = sqrt(x*x + y*y);
Expr sinPhi = y/r;

/* Define expressions for the Fourier coefficients */
Expr fourierSin1Expr = Integral(hole, sinPhi*soln, quad2);
Expr fourierDenomExpr = Integral(hole, sinPhi*sinPhi, quad2);

/* Evaluate the integrals */
double fourierSin1 = evaluateIntegral(mesh, fourierSin1Expr);
double fourierDenom = evaluateIntegral(mesh, fourierDenomExpr);

/* Write the results */
Out::os() << "fourier sin m=1 = " << fourierSin1/fourierDenom << endl;
```

As the final postprocessing example, we compute the L^2 norm of the solution u,

$$\|u\|_2 = \sqrt{\int_{\Omega} u^2 \, d\Omega}.$$

```
Expr L2NormExpr = Integral(interior, soln*soln, quad2);
double l2Norm_method1 = sqrt(evaluateIntegral(mesh, L2NormExpr));
Out::os() << "method #1: ||soln|| = " << l2Norm_method1 << endl;
```

Norm computation is a common enough operation that Sundance provides several built-in functions to compute various norms. For example, the previous computation can be carried out more compactly through the code

```
double l2Norm_method2 = L2Norm(mesh, interior, soln, quad);
Out::os() << "method #2: ||soln|| = " << l2Norm_method2 << endl;
```

Similar functions exist for the computation of the H^1 norm and H^1 seminorm.

3 Exercises

1. Change the BC on the hole to
$$\frac{\partial u}{\partial n} = 1.$$
In a postprocessing step, compute and compare the fluxes
$$Q_{\text{hole}} = \int_{\text{hole}} \mathbf{n} \cdot \nabla u \, dA$$
$$Q_{\Omega \setminus \text{hole}} = \int_{\Omega \setminus \text{hole}} \mathbf{n} \cdot \nabla u \, dA.$$
Verify that the net flux is zero.
2. Define an expression that will compute the average element diameter.