Getting started with Sundance

Kevin Long

June 21, 2012

In this document we show the support code common to many examples, and then walk through the development of a program to solve Laplace's equation.

Contents

1 Boilerplate code

A dull but essential first step is to show the skeleton 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(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 goes in place of the comment `code body goes here`.

1.1 A minimalist example

An example of the boilerplate code plus a small amount of code body is in the source file **Skeleton.cpp**. This program simply does a few MPI calls to get the processor rank and the total number of processors, does a simple sanity check, and ends. Here's the code body.

```
/* The main simulation code goes here. In this example, all we do
 * is to print some information about the processor ranks. */
MPIComm comm = MPIComm::world();
```

```
/* Print a header from the root processor only. Although this executes on
* all processors, anything written to the output stream Out::root()
* is ignored on all non-root processors (rank != 0).
* After writing, synchronize to keep this message from getting jumbled
* together with the subsequent messages.       */
Out::root() << "Example: getting started" << endl;
comm.synchronize();

/* Every processor now speaks up and identifies itself */
int myRank = comm.getRank();
int nProc = comm.getNProc();
Out::os() << "Processor " << myRank << " of " << nProc << " checking in" << endl
    ;

/* Test success or failure. Most examples you'll see will do this
* as part of the Trilinos regression testing system.
* If you write a simulation code that won't become part of Trilinos,
* you often can bypass this step.
* Here the test is a trival one: every processor's rank must be
* smaller than the total number of processors. If this fails,
* your MPI installation is probably broken!
* */
Sundance::passFailTest(myRank < nProc);
```

Output from a run on four processors is shown.

```
Simulation built using Sundance version 2.4.0 (10 June 2012)
Sundance is copyright
 (C) 2005-2012 Sandia National Laboratories
 (C) 2007-2012 Texas Tech University
and is licensed under the GNU Lesser General Public License, version 2.1

Example: getting started
p=0 | Processor 0 of 4 checking in
p=3 | Processor 3 of 4 checking in
p=1 | Processor 1 of 4 checking in
p=2 | Processor 2 of 4 checking in
test PASSED
```

1.2 Miscellaneous preliminaries

If you want to get on with solving differential equations, skip ahead to section 2.

1.2.1 Reading parameters from command-line arguments

If you want to get on with solving differential equations, skip ahead to section 2.

Sometimes you'll want to set program options using command-line arguments. The Teuchos CommandLineProcessor system provides a number of utitilies for parsing command-line arguments; Sundance provides a simplified interface to that.

Example program: **CommandLineOptions.cpp**.

Here's the code body. *Notice that the setting up of command-line option parsing must be done before the call to Sundance::init().* This is one of the very few cases where code should precede the `init()` call.

```
// Declare variables whose values are to be read from the command line
// Set default values
int someInt = 137;
double someDouble = 3.14159;
string someString = "blue";
bool someBool = false;

// Register option names, variables, and help string with
// the command-line processor
Sundance::setOption("integer", someInt, "An integer");
Sundance::setOption("alpha", someDouble, "A double");
Sundance::setOption("color", someString, "What is your favorite color?");
Sundance::setOption("lie", "truth", someBool, "I am lying.");

// Now call init
Sundance::init(&argc, &argv);

Out::root() << "User input:" << endl;
Out::root() << "An integer: " << someInt << endl;
Out::root() << "A double-precision number: " << someDouble << endl;
Out::root() << "Favorite color: " << someString << endl;
Out::root() << "I am lying: " << someBool << endl;
```

With `./Sundance_CommandLineOptions.exe` and no command-line arguments, the default values are used:

```
User input:
An integer: 137
A double-precision number: 3.14159
Favorite color: blue
I am lying: 0
test PASSED
```

With the command line `./Sundance_CommandLineOptions.exe --color=red`, the string argument is set to red

```
User input:
An integer: 137
A double-precision number: 3.14159
Favorite color: red
I am lying: 0
test PASSED
```

A few further points about command-line parsing are:

- Command-line options should use the format `--name=value` when values are given, or simply `--name` when no value is needed.
- To see all command-line options and their default values, run your program with the `--help` option.
- To access the lower-level command-line processor object, use the function `Sundance::clp()` which returns the command-line processor to be used during the call to `init()`. See the Teuchos documentation for information about low-level command-line handling capabilities.

1.2.2 Reading parameters from XML files

When you write an applications code you'll often want to read problem parameters from a data file. XML together with the Trilinos `ParameterList` utility is a convenient way to do this. Even in toy example problems, most Trilinos solvers are initialized through `ParameterList` objects and it's convenient to read these from an XML file.

In the example program **XMLParameterList.cpp** a `ParameterList` is read from an XML file. The default filename is `paramExample.xml` but an alternate filename can be given as a command line option `--xml-file=[filename]`.

Here's the contents of the XML file

```
<!-- An example parameter list in XML format -->
<ParameterList>
  <ParameterList name="Widget">
    <Parameter name="Region"  type="int" value="1"/>
    <Parameter name="Material" type="string" value="Kryptonite"/>
    <Parameter name="Density"  type="double" value="3.14159"/>
  </ParameterList>
  <ParameterList name="Gizmo">
    <Parameter name="Region"  type="int" value="2"/>
    <Parameter name="Material" type="string" value="Dilithium"/>
    <Parameter name="Density"  type="double" value="2.718"/>
  </ParameterList>
</ParameterList>
```

The body of the code is shown next. The `ParameterXMLFileReader` object does the XML parsing, returning a `ParameterList` object via the `getParameters()` function.

```
/* Read the XML filename as a command-line option */
string xmlFilename = "paramExample.xml";
Sundance::setOption("xml-file", xmlFilename, "XML filename");

/* Initialize */
Sundance::init(&argc, &argv);
```

```
/* Read a parameter list from the XML file */
ParameterXMLFileReader reader(xmlFilename);
ParameterList params = reader.getParameters();

/* Get the parameters for the "Widget" sublist */
const ParameterList& widget = params.sublist("Widget");
Out::root() << "widget region label: " << widget.get<int>("Region") << endl;
Out::root() << "widget material: " << widget.get<string>("Material") << endl;
Out::root() << "widget density: " << widget.get<double>("Density") << endl;

/* Get the parameters for the "Gizmo" sublist */
const ParameterList& gizmo = params.sublist("Gizmo");
Out::root() << "gizmo region label: " << gizmo.get<int>("Region") << endl;
Out::root() << "gizmo material: " << gizmo.get<string>("Material") << endl;
Out::root() << "gizmo density: " << gizmo.get<double>("Density") << endl;
```

See the Teuchos documentation for more information on the use of parameter lists.

1.2.3 Reading a solver from an XML file

One of the most common uses of XML and `ParameterList` objects is to configure linear and nonlinear solvers. The `LinearSolverBuilder` object can create a variety of linear solver types (including Amesos, Aztec, Belos, and Playa solvers) through a single function call to the static member `createMember()`, as shown in the next two code fragments.

The `createMember()` function can be given an XML filename,

```
LinearSolver<double> solver = LinearSolverBuilder::createSolver("mySolver.xml");
```

or a parameter list,

```
ParameterList solverParams = bigList.sublist("LinearSolver");
LinearSolver<double> solver = LinearSolverBuilder::createSolver(solverParams);
```

2 Example: Laplace's equation on a 3D plate with a hole

With those preliminaries out of the way, let's solve a differential equation. Out first example will be to 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 on one surface,

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

inhomogeneous Neumann conditions on the opposite surface,

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

and homogeneous Neumann conditions

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

on all other surfaces.

2.1 Weak form

The Galerkin weak form of this problem is

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

where H_0^1 is the subspace of H^1 such that

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

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. On the surfaces where homogeneous Neumann BCs hold, the surface integral is zero, and the BCs are imposed weakly by simply omitting those integrals. 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 linear solvers 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 of subtype `DiscreteFunction`. 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.

3 Programming Laplace's equation

3.1 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.

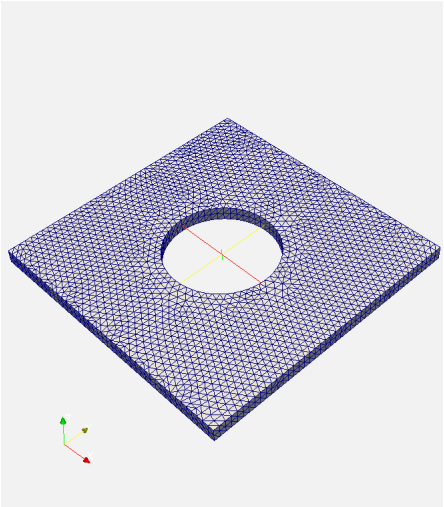

Figure 1: 3D view of meshed plate with hole

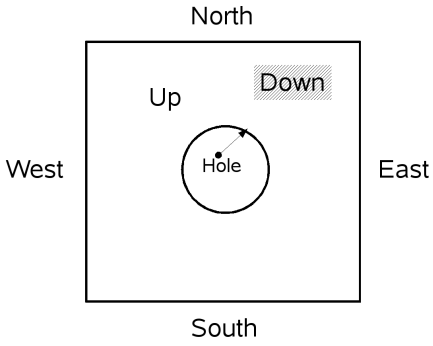

Figure 2: Schematic of labeled surfaces on the plate with hole

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.

3.2 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();
```

3.3 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.

3.4 Defining symbolic expressions

An equation is built out of mathematical expressions. Expressions, represented by `Expr` objects, can be combined using arithmetic operators, function composition, and differential operators. Expressions can be aggregated into lists.

An `Expr` object is a RCH to an expression subtype.

3.4.1 Test and unknown functions

Unknown and test functions are a vital part of every weak form. Each unknown or test function needs to have a basis function specified through choice of a `BasisFamily` object.

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

The basis object is given as an argument to the test and unknown function constructors, as shown.

```
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.

3.4.2 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 operator.

```
/* 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 ∇ */
```

3.5 Equations and boundary conditions

3.5.1 Numerical integration rules

Integrals appearing in weak forms and in postprocessing steps are done by quadrature. The family of quadrature rules to be used is specified by selection of a `QuadratureFamily` object. Different terms can use different quadrature rules. Here we create two Gaussian quadrature objects, one of order 1 (for use in integrating $\nabla v \cdot \nabla u$) and one of order 2 (for use in integrating vu on the boundary).

```
/* We need a quadrature rule for doing the integrations */
QuadratureFamily quad1 = new GaussianQuadrature(1);
QuadratureFamily quad2 = new GaussianQuadrature(2);
```

These objects are called quadrature *families* rather than quadrature *rules* because they aren’t just quadrature rules; rather, they can produce different quadrature rules for cells of different dimensions. For example, the Gaussian quadrature family will produce a Gauss-Legendre rule when used on a one-dimensional cell, or a 2D or 3D Gauss-Dunavant rule when used on a two-dimensional cell.

3.5.2 Integrals

We now have everything needed to write the weak form: a domain of integration, an integrand, and a specification of quadrature.

```
/* Write the weak form */
Expr eqn = Integral(interior, (grad*u)*(grad*v), quad1);
```

3.5.3 Essential boundary conditions

Imposition of Dirichlet boundary conditions can be a tricky aspect of finite element methods. In this example, we use the most straightforward approach, which is to replace the rows associated with boundary nodes by the boundary condition. Division of these terms by h, the local cell diameter, is done so that the terms

$$\int_{\Omega} \nabla v \cdot \nabla u \, dV$$

and

$$\int_{\text{west}} h^{-1} v u \, dA$$

scale identically with h; this helps the conditioning of the resulting linear system of equations.

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

3.6 Creating and solving a linear problem

Everything is in place to build the linear problem object. Here's the constructor.

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

Don't confuse the Sundance `LinearProblem` object with the `LinearProblem` objects in Epetra and Belos; it is quite different. The Sundance LP object is responsible for *building* a system of equations. The Epetra and Belos LP objects are encapsulations of a system of equations provided by a user.

Implementation note: LinearProblem is a lightweight user interface to a lower-level Assembler object that actually does the work of building matrices and vectors. Assembler is also used under the hood for the assembly of Jacobians and residuals for nonlinear problems and for the calculation of functional values and gradients. LinearProblem ensures that the Assembler is constructed properly, controls the call to Assembler for building the matrix and vector, invokes the linear solver and checks for convergence, and wraps the solution vector in a DiscreteFunction object so that it can be used in symbolic specification of future problems.

3.6.1 Getting a solver

The solver object can be created in a number of ways; most often it will be read from an XML file as described above.

3.6.2 Doing the solve

Invocation of the solver is simple:

```
Expr soln = prob.solve(solver);
```

The result, `soln`, is an expression with derived type `DiscreteFunction`. As an `Expr`, it can be used in further symbolic calculations; some simple examples are shown below in the section on postprocessing.

While this is the simplest way to invoke the solver, there are two issues with this syntax in complex problems in which multiple solves or error handling may be needed.

- If a problem occurs, the only feedback to the user is a thrown exception.
- A new discrete function object, `soln`, is created for every solve. While the price of allocation is relatively small, it is nonetheless an efficiency loss.

There is a version of the `solve()` function that returns diagnostics and writes the solution into an existing discrete function. This alternate version is described in the more advanced documentation.

3.7 Visualization output

To see the solution, use a `FieldWriter` to send to file. In 2D or 3D, the file formats currently supported are VTK and Exodus. Here we write to a VTK file.

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

3.8 Postprocessing

In real applications you'll want to do some computations to analyze the solution. This section gives several examples of postprocessing computations using the solution expression `soln`.

3.8.1 Flux calculation and definite integrals

The first example is the computation of the flux

$$\int_{\partial\Omega} \mathbf{n} \cdot \nabla u \, dA.$$

With no internal source, the flux should be zero to within $O(h)$; this provides a minimal validity check on the solution.We set up an expression for the flux, then call the `evaluateIntegral()` function to compute it.

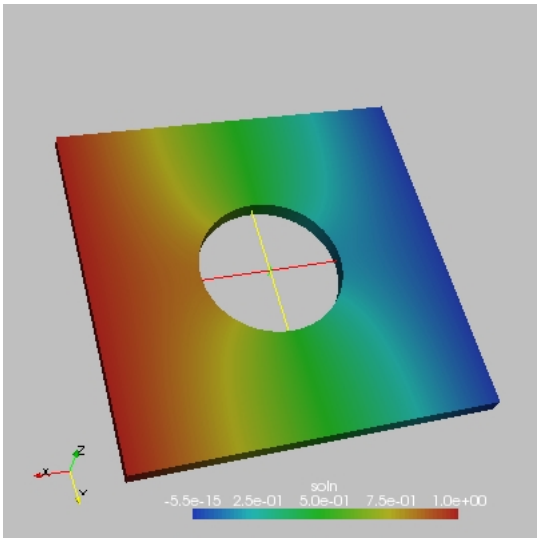

Figure 3: Solution of Laplace's equation on the holed plate.

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

3.8.2 Moments and coordinate functions

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}.

Position-dependent functions can be written using coordinate expressions.While used only in a postprocessing step here, you'll often use coordinate functions when setting up position-dependent sources and boundary conditions. Here's the construction of the coordinate expressions,

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

and their use in the integrals for the CM position.

```
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.

4 Exercises

1. Change the BC on the hole to
$$\frac{\partial u}{\partial n} = x^2.$$
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.

3. By running on a sequence of refined meshes, verify that the computations of the flux and of the first Fourier moment A_1 are converging at the correct rates.