Solving Partial Differential Equations with Sundance

Kevin Long
Computational Science and Mathematics Research Department
Sandia National Laboratories
Livermore, California
krlong@sandia.gov

January 29, 2003

Contents

Chapter 1

Introduction

Sundance is a system for rapid development of high-performance parallel finite-element solutions of partial differential equations. The motivation for writing Sundance is a conviction that you should be able to code a finite element problem using the same level of abstraction you would use to describe the problem on a blackboard. Sundance provides a set of high-level components with which you can specify and solve a problem. The high-level nature of the components means that you need not worry about tedious and error-prone bookkeeping details. In addition to the advantage of conceptual simplicity and freedom from bookkeeping, this component-based approach allows a high degree of flexibility in the formulation, discretization, and solution of a problem.

Sundance solves PDEs using the finite-element method: a general, powerful, and quite elegant method for turning a PDE into a discrete system of algebraic equations. To use Sundance, you will need to understand the fundamentals of the finite-element method. Many introductory textbooks are available, such as Hughes[**?**]. A nice one-chapter introduction to the finite-element method set in the broader context of numerical solution of differential equations can be found in Iserles[**?**]. Some of the more subtle aspects of the finite element method are described in the books by Gresho and Sani[**?**] (with an emphasis on fluid mechanics), Belytscheko et al (emphasizing nonlinear solid mechanics), and Brenner and Scott (for the mathematician). Gresho and Sani[**?**] have an excellent literature review with references to books at all levels.

This is not the place to teach you the finite-element method, but it is worth pointing out some salient features of the finite-element method and their impact on how you will use Sundance.

- A finite-element model of a PDE is based on a **weak form** of the PDE. You will describe a PDE in Sundance code using a high-level symbolic notation for writing weak forms. There can be more than one way to derive a weak form for a given PDE.

- The geometric domain for your problem will decomposed into a discrete **mesh** of cells. Meshing a continuous geometric figure represented by an equation or CAD drawing is a difficult problem in both theory and practice. Sundance has some simple mesh generation capability built in; however, for all but the simplest toy problems you will instead want to use a third-party mesher and import that mesh into Sundance.

- A finite-element model approximates a PDE by a discrete system of algebraic equations: linear algebraic equations for a linear PDE, nonlinear equations for a nonlinear PDE. This process of going from a PDE to an system of algebraic equations is called **discretization**. There are many factors that influence the final discrete form of the equation: choice of weak form, method of imposing BCs, basis functions for the unknowns in the problem, and more.

- Designing algorithms, or `solvers`, for solving large system of linear equations is another difficult area of computational mathematics. There is definitely *not* a one-size-fits-all solver for $\mathbf{Kx} = \mathbf{f}$. The optimal solver for a given problem can depend critically on the structure of the stiffness matrix[1] K. Sundance lets you choose a solver to suit the structure of your problem, and provides an interface for plugging in high-performance linear algebra software.

[1]The matrix K and vector f the in the linear system arising from a finite-element model are often called the stiffness matrix and load vector because of the historical origins of the finite-element method in structural mechanics. We'll often use those names even in problems where K and f do not have the physical interpretation of stiffness and load.

- Solution of a nonlinear problem can be reduced to solving a sequence of linear problems. There are many choices of linearization method and iteration method for a nonlinear problem, each of which will result in a different sequence of linear problems.
- Solution of a time-dependent problem can be also reduced to solving a sequence of linear problems, by timestepping or **marching**. Again, there are many possible marching algorithms.

After reading the above list you should have the idea that solving PDEs with the finite-element method is a very open-ended business, and that in designing a simulation you will be faced with a large number of choices. Sundance is intended to let you make those choices by selecting and combining high-level objects rather than by writing low-level code.

Note that Sundance cannot help you *make* good choices – for that you will have to rely on your understanding of the problem. Unfortunately, Sundance has no way to detect if you use its components to build an unstable or innaccurate discretization or choose a solver that is inefficient for your problem.

1.1 About the code and the documentation

Sundance is written in the C++ programming language. To write your own Sundance simulation, you will have to write and compile C++. However, you need not be an expert C++ programmer, because you can do most things using Sundance's predefined objects. The trick to writing Sundance code is to understand the behavior of the user-level objects. The main purpose of this document is to show you how to use Sundance's user-level objects to set up and solve PDEs.

1.1.1 Typographical conventions in the source code and examples

Class names begin with capital letters, and each word within the name also begins capitalized. For example: `MeshReader` and `DiscreteFunction` are classes. Method names and variables begin with lower-case letters, but subsequent words within the name are capitalized. For example: `getCells()` and `numCells()` are methods. If you forge on to read the developer's documentation or source code, you will need to know that data member names end with an underscore, for example: `myName_`.

1.2 Example: Poisson on a plate

Enough preliminaries, let's do a problem. We'll solve the Poisson equation with a unit source

$$\nabla^2 u = 1 \tag{1.1}$$

on the rectangle $[0,0]$ - $[1,2]$. The sides of the rectangle will be labeled left, right, bottom, top. For boundary conditions, we will choose

- **left** Homogeneous Neumann, $\nabla u \cdot \hat{n} = 0$
- **bottom** Dirichlet, $u = \frac{1}{2}x^2$
- **right** Robin, $u + \nabla \mathbf{u} \cdot \hat{\mathbf{n}} = \frac{3}{2} + \frac{y}{3}$
- **top** Neumann, $\nabla u \cdot \hat{\mathbf{n}} = 1/3$

It is easy to check that the solution is

$$u = \frac{1}{2}x^2 + \frac{1}{3}y. \tag{1.2}$$

The solution is in the subspace spanned by second-order Lagrange polynomials, so if we choose that as our basis family we can expect to obtain the exact solution. We can compute the error norm at the end of the calculation as a check that the code is working properly.

This is a simple problem, but it in fact uses most of the components used by Sundance to do more complex problems.

1.2.1 Step-by-step explanation

We start with a step-by-step walkthrough of the code for solving the Poisson problem. When finished, there will be a summary and then the complete Poisson solver code will be listed for reference.

Boilerplate

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

```
#include "Sundance.h"

int main(int argc, void** argv)
{
  try
    {
      Sundance::init(argc, argv);

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

The body of the code – everything else we discuss here – goes in place of the comment `code body goes here`.

Getting the mesh

Sundance uses a `Mesh` object to represent a discretization of the problem domain. There are two ways to get a `Mesh` object:

- Create it using Sundance's built-in mesh generation capability. This is limited to meshing very simple domains such as rectangles.
- Read a mesh that has been produced using a third-party mesh generator. The `MeshReader` class provides an interface for reading arbitrary file formats.

For this simple problem, we can use Sundance to generate the mesh.

```
MeshGenerator mesher = new RectangleMesher(0.0, 1.0, nx, 0.0, 2.0, ny);
Mesh mesh = mesher.buildMesh();
```

If you know a little C++ – just enough to be dangerous – you might think it odd that the result of the `new` operator, which returns a pointer, is being assigned to a `MeshGenerator` object which is – apparently – not a pointer. That's not a typo: the `MeshGenerator` object is a **handle** class that stores and manages the pointer to the `RectangleMesher` object. Handle classes are used throughout user-level Sundance code, and among other things relieve you of the need to worry about memory management.

Defining coordinate functions

In the Poisson example, the boundary conditions involve functions of the coordinates x and y. We will create objects to represent the coordinate functions x and y.

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

You have probably guessed that the integer argument to the `CoordExpr` constructor gives the coordinate direction: 0 for x, 1 for y, 2 for z.

The coordinate functions are wrapped in `Expr` handle objects. Class `Expr` is used for all symbolic objects in Sundance. `Expr`s can be operated on with the usual mathematical operators. With our coordinate functions represented as `Expr` objects, we can build complicated functions of position.

Defining the cell sets

We've already read a mesh. We need a way to specify *where* on the mesh equations or boundary conditions are to be applied. Sundance uses a `CellSet` object to represent subregions of a geometric domain. A `CellSet` can be any collection of mesh cells, for example a block of maximal cells, a set of boundary edges, or a set of points.

The `CellSet` class has a `subset()` method that can be used as a "filter" that identifies cells that are in a subset defined by the arguments to the `subset` method.

We will apply different boundary conditions on the four sides of the rectangle, so we will want four `CellSets`, one for each side. We first create a cell set object for the entire boundary,

```
CellSet boundary = new BoundaryCellSet();
```

and then we find the four sides as subsets of the boundary cell set. The four sides of the rectangle can be specified with logical operations on coordinate expressions, as shown in the following code:

```
CellSet left = boundary.subset( x == 0.0 );
CellSet right = boundary.subset( x == 1.0 );
CellSet bottom = boundary.subset( y == 0.0 );
CellSet top = boundary.subset( y == 2.0 );
```

Creating a discrete function

Symbolic expressions have already been introduced. We will use discrete functions several places in this problem. A discrete function takes as a constructor argument a vector space object that specifies the mesh, basis, and vector representation to be used in discretizing the function.

The first step is to create a vector space factory object that tells us what kind of vector representation will be used. We'll use Petra vectors, so we create a `PetraVectorType`.

```
TSFVectorType petra = new PetraVectorType();
```

We can now create a `SundanceVectorSpace` containing the mesh, a basis (2nd order Lagrange in this case) and the vector space factory.

```
TSFVectorSpace discreteSpace = new SundanceVectorSpace(mesh, new Lagrange(2), petra);
```

Finally, we can create discrete functions to represent the source term $f = 1.0$ and the expression $\frac{3}{2} + \frac{y}{3}$ that appears in the right BC. Note that there's no particular need to to use discrete functions for those terms; we do so here simply to provide an example of constructing a discrete function.

```
Expr f = new DiscreteFunction(discreteSpace, 1.0);
Expr rightBCExpr = new DiscreteFunction(discreteSpace, 1.5 + y/3.0);
```

Defining unknown and test functions

We'll use 2nd order piecewise Lagrange interpolation to represent our unknown solution u. With a Galerkin method we use a test function v defined using the same basis as the unknown. Expressions representing the test and unknown functions are defined easily:

```
Expr v = new TestFunction(new Lagrange(2));
Expr u = new UnknownFunction(new Lagrange(2));
```

Creating the gradient operator

The gradient operator is formed by making a `List` containing the partial differentiation operators in the x and y directions.

```
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr grad = List(dx, dy);
```

The gradient thus defined is treated as a vector with respect to the overloaded multiplication operator used to apply the gradient, so that an operation such as `grad*u` expands correctly to {`dx*u, dy*u`}.

Writing the weak form

We will use the Galerkin method to construct a weak form. Begin by multiplying Poisson's equation Equation 1.1 by a test function v and integrating

$$-\int_{\Omega} v\nabla^2 u - \int_{\Omega} vf = 0. \tag{1.3}$$

The next step is to integrate by parts, which has the effects of lowering the order of differentiation (and thus relaxing the differentiability requirements on the unknown and test functions) and also making manifest the boundary flux. The resulting weak form is

$$-\int_{\Omega} \nabla v \cdot \nabla u - \int_{\Omega} vf + \int_{\partial\Omega} v\nabla \mathbf{u} \cdot \hat{\mathbf{n}} = 0 \tag{1.4}$$

and we will require that this equation hold for any test function v in the space of 2nd order Lagrange interpolants on our mesh. The boundary term gives us a way to apply certain boundary conditions: we can apply the Neumann and Robin BCs by substituting an appropriate value for $\nabla \mathbf{u} \cdot \hat{\mathbf{n}}$ in the boundary term. Referring to the boundary conditions above and our definition of the discrete function `rightBCExpr`, the weak form is written in Sundance as

```
Expr poisson = Integral(-(grad*v)*(grad*u)  - f*v)
                    + Integral(top, v/3.0)
                    + Integral(right, v*(rightBCExpr - u));
```

Notice that the homogeneous BC on the left side does not need to be written explicitly because that boundary term is zero.

Writing the essential BCs

The weak form contains the physics in the body of the domain plus the Neumann and Robin boundary conditions. We still need to apply the Dirichlet boundary condition on the bottom edge, which we do with an `EssentialBC` object

```
EssentialBC bc = EssentialBC(bottom, v*(u - 0.5*x*x));
```

The first argument gives the region on which the boundary condition holds, and the second gives an expression that is to be set to zero. Notice that there is a test function in the BC; this identifies the row space on which the BC is to be applied.

Creating the linear problem object

A `StaticLinearProblem` object contains everything that is needed to assemble a discrete approximation to our PDE: a mesh, a weak form, boundary conditions, specification of test and unknown functions, and a specification of the low-level matrix and vector representation to be used. All of this information is given to the constructor to create a problem object

```
StaticLinearProblem prob(mesh, poisson, bc, v, u, petra);
```

It may seem unnecessary to provide v and u as constructor arguments here; after all, the test and unknown functions could be deduced from the weak form. In more complex problems with vector-valued unknowns, however, we will want to specify the order in which the different unknowns and test functions appear, and we may want to group unknowns and test functions into blocks to create a block linear system. Such considerations can make a great difference in the performance of linear solvers for some problems. The test and unknown slots in the linear problem constructor are used to pass information about the function ordering and blocking to the linear problem; these features will be used to effect in subsequent examples.

Specifying the solver

A good choice of solver for this problem is BICGSTAB with ILU preconditioning. We'll use level 2 preconditioning, and ask for a convergence tolerance of 10^-14 within 500 iterations.

```
TSFPreconditionerFactory precond = new ILUKPreconditionerFactory(2);
TSFLinearSolver solver = new BICGSTABSolver(precond, 1.e-14, 500);
```

Solving the problem

The syntax of Sundance makes the next step look simpler than it really is:

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

What is happening under the hood is that the problem object prob builds a stiffness matrix and load vector, feeds that matrix and vector into the linear solver solver. If all goes well, a solution vector is returned from the solver, and that solution vector is captured into a discrete function wrapped in the expression object soln.

Viewing the solution

We next write the solution in a form suitable for viewing by Matlab.

```
FieldWriter writer = new MatlabWriter("heat2D.dat");
writer.writeScalar(mesh, "temperature", soln);
```

Checking the error norm

Finally, we compare to the exact solution by computing the error norm. The solution has been returned as a Sundance expression, so we can form an expression for the error

```
Expr exactSoln = 0.5*x*x + y/3.0;
Expr error = exactSoln - soln;
```

and then take the L^2 norm

```
double errorNorm = error.norm();
```

1.2.2 Complete code for the poisson problem

```
#include "Sundance.h"

/** \example heat2D.cpp
 * Solve Poisson's equation with a unit source term on the
 * rectangle [0,1] x [0, 2] with the following boundary conditions:
 *
 * Left:   Natural, du/dx = 0
 * Bottom: Dirichlet, u= 0.5 x^2
 * Right:  Robin, u + du/dx = 3/2 + y/3
 * Top:    Neumann, du/dy = 1/3
 *
 * The solution is u(x,y) = 0.5*x^2 + y/3.
 *
 * This problem can be solved exactly in the space of second-order polynomials.
 */

int main(int argc, void** argv)
{
  try
    {
      Sundance::init(argc, argv);

      /* create a simple mesh on the rectangle */
      int nx = 20;
      int ny = 20;
      MeshGenerator mesher = new RectangleMesher(0.0, 1.0, nx, 0.0, 2.0, ny);
      Mesh mesh = mesher.getMesh();

      /* define coordinate functions for x and y coordinates */
      Expr x = new CoordExpr(0);
      Expr y = new CoordExpr(1);

      /* define cells sets for each of the four sides of the rectangle */
      CellSet boundary = new BoundaryCellSet();
      CellSet left = boundary.subset( x == 0.0 );
      CellSet right = boundary.subset( x == 1.0 );
      CellSet bottom = boundary.subset( y == 0.0 );
      CellSet top = boundary.subset( y == 2.0 );

      /* Create a vector space factory, used to
       * specify the low-level linear algebra representation */
      TSFVectorType petra = new PetraVectorType();

      /* create a discrete space on the mesh */
      TSFVectorSpace discreteSpace = new SundanceVectorSpace(mesh, new Lagrange(2), petra);


      /* We'll use a discrete function to represent the
       * source term, providing a test
       * of our ability to evaluate discrete functions on maximal cells */
      Expr f = new DiscreteFunction(discreteSpace, 1.0);

      /* We'll use a discrete function to represent the imposed
       * boundary value on the right-hand boundary.  This provides a
       * test of our ability to evaluate discrete functions on
       * lower-dimensional cells. */
      Expr rightBCExpr = new DiscreteFunction(discreteSpace, 1.5 + y/3.0);


      /* create symbolic objects for test and unknown functions */
      Expr v = new TestFunction(new Lagrange(2));
      Expr u = new UnknownFunction(new Lagrange(2));

      /* create symbolic differential operators */
      Expr dx = new Derivative(0);
      Expr dy = new Derivative(1);
      Expr grad = List(dx, dy);

      /* Write symbolic weak equation and Neumann and Robin BCs */
      Expr poisson = Integral(-(grad*v)*(grad*u)  - f*v, new GaussianQuadrature(2))
        + Integral(top, v/3.0) + Integral(right, v*(rightBCExpr - u));


      /* Write essential BCs:
       * Bottom: u=x^2
       */
      EssentialBC bc = EssentialBC(bottom, v*(u - 0.5*x*x),
                                   new GaussianQuadrature(4));

      /* Assemble everything into a problem object, with a specification that
       * Petra be used as the low-level linear algebra representation */
      StaticLinearProblem prob(mesh, poisson, bc, v, u, petra);

      /* create a preconditioner and solver */
      TSFPreconditionerFactory precond = new ILUKPreconditionerFactory(1);
      TSFLinearSolver solver = new BICGSTABSolver(precond, 1.e-14, 500);

      /* solve the problem and return the solution as a symbolic object */
      Expr soln = prob.solve(solver);

      /* write to matlab */
      FieldWriter writer = new MatlabWriter("heat2D.dat");
      writer.writeField(soln);

      /* compare to known solution */
      Expr exactSoln = 0.5*x*x + y/3.0;

      // compute the norm of the error
      double errorNorm = (soln-exactSoln).norm(2);
      double tolerance = 1.0e-9;

      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
    }
  catch(exception& e)
    {
      Sundance::handleError(e, __FILE__);
    }
  Sundance::finalize();
}
```

1.3 Handles and memory management in Sundance

Understanding handle classes and how they are used in Sundance is important for reading and writing Sundance code and browsing the source and class documentation. Handle classes are used in Sundance to simplify user-level polymorphism and provide transparent memory management.

Polymorphism is a buzzword meaning the representation of different but related object types (derived classes, or subclasses) through a common interface (the base class). In C++, you can't use a base-class object to represent a derived class; you have to use a pointer to the base class object to represent a pointer to the derived class. That leads to a rather awkward syntax and also requires attention to memory management. To simplify the interface and make memory management automatic, all user-level polymorphism is done with handle classes. A handle class is simply a class that contains a pointer to a base class, along with an interface providing user-callable methods, and a (presumably) intelligent scheme for memory management.

So if you want to work with a family of Sundance objects, for instance the different flavors of symbolic objects, you need only use:

- the methods of the handle class for that family of classes
- the constructors for the derived classes.

You do not need to, and shouldn't, use any methods of the derived classes; all work with the family should be done with methods of the handle class.

For example, Sundance symbolic objects are represented with a handle class called `Expr`. The different symbolic types derive from a class called `ExprBase`, but they are never used directly after construction; they are used only through the Expr handle class. The code fragment below shows some Exprs being constructed through subclass constructors and then being used in Expr operations.

```
Expr x = new CoordExpr(0, "x");
Expr f = x + 3.0*sin(x);
Expr dx = new Derivative(0);
Expr df = dx*f;
```

Notice that a pointer to a subclass object is created using the new operator, and then given to the handle. The handle object assumes responsibility for that pointer: it does all memory management, any copying that might occur, and will eventually delete it. You, the user, should **never** delete a pointer that has been passed to a handle. Memory management is the responsibility of the handle. Code such as this will seem familiar to Java programmers, who call `new` but never `delete`.

Thanks to handles, when writing Sundance code you can always assume that

- User-level classes have well-defined behavior for copying and assignment.
- User-level classes have well-defined destructors, and take care of their own memory management.

Data structures in a PDE simulation can become rather large; for this reason, objects such as meshes, matrices, and degree-of-freedom maps are shallow-copied so that both the original and the copy refer to the same chunk of memory. A reference counted "smart pointer" inside the handle is used to ensure that data is deleted only when necessary. It is important to understand that such a copying scheme leads to side effects: when a copy is modified, the original is modified as well.

1.4 Parallel computation

Sundance can both assemble and solve linear systems in parallel. Parallel Sundance uses the SPMD paradigm, in which the same code is run on all processors. Communication is done using an object wrapper for MPI. To use Sundance's parallel capabilities, the CPPUtilities package supporting Sundance has to be built with MPI enabled, and you must use a parallel solver such as Trilinos. See the installation documentation for help in installing parallel Sundance.

One of the design goals was to make parallel solves look to the user as much as possible like serial solves. In particular, the symbolic description of an equation set and boundary conditions is completely unchanged from serial to parallel runs. To run a problem in parallel, you simply need to use a parallel solver (such as trilinos) and use a partitioned mesh.

Operations such as norms and definite integrals on discrete functions are done such that the result is collected from all processors.

Chapter 2

Symbolic Expressions

In Sundance one writes equations and boundary conditions and does auxiliary steps such as postprocessing and error estimation using a family of symbolic objects.

2.1 Overview of the Expr class

Class `Expr`, short for expression, is the user-level handle for all symbolic objects. An `Expr` can be an atomic entity such as a constant, a function, or a differential operator; or an expression can be a compound object such as a sum, project, or list of expressions. A comprehensive list of user-level methods can be found in section A.1.1.

Many different kinds of objects, with different behaviors, are represented with Expr objects. In object-oriented design jargon, the `Expr` family is weakly-typed. This is key to giving `Expr` the behavior needed, but does have an important drawback: is is quite possible to form nonsensical expressions, and such errors cannot be detected at compile-time.

2.1.1 Lists of expressions

`Expr`s can be aggregated into lists with arbitrarily heterogeneous structure. Lists of expressions have many uses, such as representing vector fields

```
Expr position = List(x, y, x);   // form position vector from coordinates
```

or simply gathering together a number of expressions. Lists can be heterogeneous, so that it is possible to form lists such as

```
Expr myList = List(a, List(b, c), List(List(d, List(e, f), g))); // {a, {b,c}, {d, {e,f}, g}
```

An important use of expression lists is in controlling the row and column structure of a linear operator to be assembled by Sundance.

2.1.2 Elementary mathematical operations

Operator overloading has been used to define mathematical operations on `Expr` objects. Mathematical operations on expression objects return expression objects. The usual arithmetic operations are defined for `Expr`s.

- `a + b` adds `Expr a` and `Expr b`. The two operands must have identical list structures.
- `a - b` subtracts `Expr b` from `Expr a`. The two operands must have identical list structures.
- `-a` returns the additive inverse of `a`.
- `a * b` takes the product of `Expr a` and `Expr b`. The meaning of this operation depends on the list structure of `a` and `b`. If either `a` or `b` is scalar-valued, the product means multiplication by a scalar. For example,

```
Expr a = 2.0;
Expr position = List(x, y);
Expr f = a*position;         // results in {a*x, a*y}
```

The multiplication operator is also meaningful when the two operands have a list structure such that multiplication can be interpreted as a tensor contraction operation (generalized dot product). For example,

```
Expr force = List(fx, fy);
Expr velocity = List(vx, vy);
Expr power = force * velocity; // results in fx*vx + fy*vy;
```

- `a / b` divides `Expr a` by `Expr b`. The denominator must be a scalar. For example,

```
Expr position = List(x, y);
Expr force = -position / pow(position*position, 1.5); // Coulomb force
```

Elementary functions

In addition to the arithmetic operations, the usual elementary functions are defined for Sundance expressions. Arguments of elementary functions must be scalar-valued; violation of this rule will result in a runtime error. Since Sundance does not support complex numbers, the arguments of each function are restricted to the domain that maps to real principal values; for example, the argument of the natural logarithm function `log` must be a positive number. The names used for the elementary functions conform to those in the standard C math library, e.g. `sqrt` for the square root. One "gotcha" for fortran and matlab programmers is that the absolute value function is called `fabs` in accord with the C standard. See section A.1.1 for a complete listing of the elementary functions known to Sundance. Should you need a function that cannot be written easily or efficiently in terms of elementary functions, don't despair: you can add your own user-defined function to Sundance's symbolic system as a plug-in (see section **??**

Integrals and norms

You will sometimes need to compute the definite integral of an expression over a domain, for example in computing the value of an objective function in an optimization problem. A very common special case is computing the L^p norm of a function over a region Ω, defined as

$$\|f\|_{L^p} \equiv \left[\int_\Omega |f(x)|^p dx\right]^{1/p}. \tag{2.1}$$

Note that the L^∞ norm is the maximum value of a function.

Sundance has methods to compute integrals and norms. The most general method to compute an integral of an expression is

```
double Expr::integral(const Mesh& mesh,
                      const CellSet& cellSet, const QuadratureFamily& quad) const ;
```

which will integrate over the subdomain of `mesh` defined by the cell set `cellSet`, using quadrature rules derived from the quadrature family `quad`. Any of those arguments can be omitted in favor of default arguments, but if the mesh argument is omitted the expression in question must be a discrete function. There are also methods to do the special case of a norm, for instance

```
double Expr::norm(int p, const Mesh& mesh,
                  const CellSet& cellSet, const QuadratureFamily& quad) const ;
```

is the most general method for computing the L^p norm. Since we cannot give $p = \infty$ as an argument to the standard norm methods, there is a special method, `maxNorm()`, for computing the L^∞ norm. The default `norm()` method computes the L^2 norm of a discrete function over the entire domain. Finally, the method `quickNorm()` computes the norm of the vector underlying a discrete function; note that this will not in general be equal to the norm of the discrete function but when a "quick and dirty" norm is all that is needed, it can be somewhat faster to compute. A summary of all integral and norm methods is given in section A.1.1

2.2 Expression types

The simplest type of Expr to create is a constant real-valued Expr, for example:

```
Expr solarMass = 2.0e33; // mass of the Sun in grams
```

Any constant that appears in an expression, for example the constant 2.0 in the expression below,

```
Expr f = 2.0*g;
```

will also be turned into a constant-valued expression. It is important to understand that once created and used in an expression, a constant's value is immutable. If you want to change the constant, you should instead use a `Parameter`.

The following code will not work as you intend:

```
Expr time = 0.0;

for (int i=0; i<10; i++)
{
cerr << time << ‘‘ ‘‘ << sin(pi*time) << endl;
// update the time
time = time + 0.1;
}
```

2.2.1 Parameter expressions

Often you will form a PDE with parameters that will change during the course of a calculation. For example, in a time-marching problem both the time and the timestep can change from step to step. Or, you may want to run a fluid flow simulation at several different values of the Reynolds number. To include in your equation a parameter that is constant in space but can change with time or some other way, you should represent that parameter with a Parameter expression.

```
Expr time = new Parameter(0.0);

for (int i=0; i<10; i++)
{
cerr << time << ‘‘ ‘‘ << sin(pi*time) << endl;
// update the time
time.setValue(time.value() + 0.1);
}
```

The above assume that the parameter is known. However, in some problems a parameter might be an unknown to be determined in the course of solving a problem. In that case, use an `UnknownParameter`, described in section **??**.

2.2.2 Coordinate expressions

`CoordExpr` is an expression subtype that is hardwired to compute the value of a given coordinate. Its constructors are

```
CoordExpr(int direction);
CoordExpr(int direction, const string &name);
```

For example, to construct an Expr that represents the coordinate on the zeroth (x) axis, create a new CoordExpr object and pass it to an Expr ctor as follows:

```
Expr x = new CoordExpr(0); // represents x-coordinate value
```

Such a coordinate expression can be used to define simple position-dependent functions, for example

```
Expr f = sin(x) + 1/4.0*sin(2.0*x) + 1/8.0*sin(3.0*x);
```

2.2.3 Differential operators

The key expression subtype for forming differential operators is the `Derivative` object, representing a partial derivative in a given direction. A `Derivative` is constructed with a single integer argument giving the direction of differentiation, for example,

```
Expr dx = new Derivative(0); // differentiate with respect to 0 coordinate
```

Derivatives are applied using the multiplication (`*`) operator.

Sundance expression objects are programmed to obey the rules of differential calculus. For example,

```
Expr dx = new Derivative(0); // differentiate with respect to 0 coordinate
Expr x = new CoordExpr(0); // represents x-coordinate value
Expr y = new CoordExpr(0); // represents y-coordinate value
Expr f = x*sin(x) + y*x;
Expr df = dx*f;
cout << df << endl;  // prints sin(x) + x*cos(x) + y;
```

Differentiation of discrete functions requires special care, and is discussed in 2.2.5

2.2.4 Test and unknown functions

Expression subtypes `TestFunction` and `UnknownFunction` are used to represent test and unknown functions in weak PDEs and boundary conditions. They are constructed with a `BasisFamily` object which specifies the subspace to which solutions and test functions are restricted. For example,

```
Expr T = new UnknownFunction(new Lagrange(1));
Expr varT = new TestFunction(new Lagrange(1));
```

constructs unknown and test functions that live in the space spanned by first-order Lagrange interpolates, i.e., all piecewise linear functions.

2.2.5 Discrete functions

Discrete functions represent the value of a field that has been discretized on a space of basis functions. Discrete functions have a number of important uses:

- representing the solution of a finite-element problem
- representing a field for which no analytical expression is available

A discrete function object can be created in a number of ways: by computing the value of an expression on the nodes in a mesh, by reading it from a file, or by "capturing" a solution vector into a discrete function.

Creating a scalar-valued discrete function

To create a discrete function, we first need to know the discrete space on which the function will be defined. The construction of this space requires at minimum a mesh, a basis function, and a vector type.

```
TSFVectorType petra = new PetraVectorType();
BasisFamily basis = new Lagrange(1);
TSFVectorSpace discreteSpace = new SundanceVectorSpace(myMesh, basis, petra);
```

Once you have a discrete space, you can create a discrete function as follows:

```
Expr f = new DiscreteFunction(discreteSpace, sin(x)*sin(y));
```

Creating a vector-valued discrete function

Discrete functions representing vector-valued fields have some wrinkles that are important to understand. Consider a discrete function representing a two-component vector field, $\mathbf{u} =$
$left(u_x, u_y$
$right)$. How is the vector underlying this function stored? One can imagine creating two independent discrete functions

```
Expr ux = new DiscreteFunction(discreteSpace, sin(x)*sin(y));
Expr uy = new DiscreteFunction(discreteSpace, cos(x)*cos(y));
```

and forming a vector-valued expression using the `List` operator,

```
Expr u = List(ux, uy);
```

This is well-defined Sundance code, but it is not usually what you want. A calculation will have improved performance due to cache efficiency if both functions are aggregated into a single vector, with u_x and u_y at each cell listed together. To achieve this aggregation, we need to create a discrete space capable of representing vector-valued functions.

```
TSFVectorType petra = new PetraVectorType();
BasisFamily basis = new Lagrange(1);
TSFArray<BasisFamily> multiVariableBasis = tuple(basis, basis);
TSFVectorSpace multiVariableDiscreteSpace
= new SundanceVectorSpace(myMesh, multiVariableBasis, petra);
Expr u = DiscreteFunction::discretize(multiVariableDiscreteSpace,
  List(sin(x)*sin(y), cos(x)*cos(y)));
```

In many problems, it is necessary to use a mixed set of basis functions. For example, in the Taylor-Hood discretization of the incompressible Navier-Stokes equations, the velocity components are represented with 2nd order polynomials and the pressure with 1st order polynomials.

```
TSFVectorType petra = new PetraVectorType();
BasisFamily basis1 = new Lagrange(1);
BasisFamily basis2 = new Lagrange(2);
TSFArray<BasisFamily> multiVariableBasis = tuple(basis2, basis2, basis1);
TSFVectorSpace multiVariableDiscreteSpace
= new SundanceVectorSpace(myMesh, multiVariableBasis, petra);
Expr uAndP = DiscreteFunction::discretize(multiVariableDiscreteSpace,
  List(y, x, 0.0));
```

Reading a discrete function

Many mesh file formats have the ability to store field data along with the mesh. This field data can be associated with elements or with nodes, depending on the application and the physical meaning of the field. Different mesh file format will index fields in different ways; for example, the Exodus format associates names with fields, while Shewchuk's Triangle format simply lists attributes. Generally, we can look up fields by either a name or by a number indicating the position in an attribute list.

```
MeshReader reader = new ShewchukMeshReader(``myMesh'');
Expr temperature = reader.getNodalField(0);
Expr velocity = reader.getNodalField(1, 2, 3)
Expr pressure = reader.getElementalField(4)

MeshReader reader = new ExodusMeshReader(``myMesh.exo'');
Expr pressure = reader.getElementalField(``pressure'');
Expr velocity = reader.getNodalField(``ux'', ``uy'', ``uz'')
Expr temperature = reader.getNodalField(``temperature'');
```

Derivatives of discrete function

Many basis functions used in finite elements calculations are only piecewise differentiable: the function is continuous everywhere and differentiable in the interior of each cell, but the derivative is not defined at boundaries between cells. Such basis functions, and functions represented with them, are said to have C^0 continuity. Since the derivative of such a function will not be continuous at element boundaries, the derivative of a C^0 function is not necessarily C^0. Thus, the derivative of a discrete function defined with a particular discrete space cannot be represented exactly with another discrete function defined with that same space.

For this reason, it is impossible to create directly a discrete function from the derivative of another discrete function. The following will result in a runtime error:

```
Expr f = new DiscreteFunction(discreteSpace, sin(x));
Expr dfdx = new DiscreteFunction(discreteSpace, dx*f);
```

If f is a C^0 function, it is possible to *integrate* derivatives of f. The integral is well-defined since the region on which f is nondifferentiable have no volume. Numerically, it is usually possible to do such integrals because the quadrature points are usually in the interiors of cells. So it's perfectly sensible, and quite common, to write a weak PDE that includes derivatives of discrete functions.

What is not possible is to obtain pointwise values of the derivative of a discrete function. This is not a common operation during the solution of a PDE, but you may often want to see derivative values during postprocessing and analysis. Because pointwise values are not available, it is impossible to create directly a discrete function from the derivative of another discrete function. The following will result in a runtime error:

```
Expr f = new DiscreteFunction(discreteSpace, sin(x));
Expr dfdx = new DiscreteFunction(discreteSpace, dx*f);
```

If you really want to look at pointwise derivative values, the best that can be done is to approximate the derivative by projecting into a C^0 space. There are many ways to do this; one of the most common is a least-squares projection, in which you choose coefficients such as to minimize the squared residual.

This is a common enough operation that Sundance has a predefined method for least-squares projection:

```
// f0 is a discrete function
Expr gradF = L2Projection(discreteSpace, List(dx, dy)*f0);
```

Note that since this operation requires the solution of a linear system, it is time-consuming. Again, it usually needs to be done only as a postprocessing step.

Finally, it should be pointed out that the difference between a derivative and its L^2 projection will decrease as the function becomes smoother. For this reason, the L^2 residual of a derivative can be used as an error estimator.

2.2.6 Test and unknown parameters

Expression subtypes `TestParameters` and `UnknownParameter` are used to represent test and unknown functions that are independent of space.

2.3 Functionals

2.3.1 Assignment and copy semantics

It is very important to understand the behavior of the assignment operator and copy constructor for `Expr` objects. Assignment and copy are by reference, so that if you create an expression

```
f = a + b + c;
```

and then change the value of `c`,

```
c = sin(x);
```

that change will instantly propagate to the copy of `c` in the sum `f`.

```
cout << f << endl; // will print a + b + sin(x)
```

This feature lets you automatically update expressions in timestepping or nonlinear solve loops, making the code for such problems more efficient and much easier to understand. The case when the lhs of an assignment operator appears on the rhs is handled as follows: the sequence of operations

```
x = 5;
x = y + x;
```

leaves `x` set to `y + 5`.

Chapter 3

Geometry

3.1 Meshes

Sundance can use unstructured meshes in 1, 2, or 3 dimensions. To Sundance, a `Mesh` object is a connected complex of cells. A **zero-cell** is a point. A **maximal** cells is a cell with dimension equal to the spatial dimension of the mesh. Each facet of a maximal cell is itself a cell, and so on down to zero cells. Every discrete geometric entity in Sundance is a cell; there is no distinction between “elements”, “edges”, and “nodes”. All are represented by `Cell` objects.

Sundance currently supports the following cell types:

- zero-cells: points
- one-cells: lines
- two-cells: triangles and quadrilaterals (“quads”)
- three-cells: tetrahedra (“tets”) and hexahedra (“bricks” or “hexes”)

The system for representing cells is extensible, so that an advanced user can add additional cell types such as prisms.

Most of the methods of the `Mesh` class are for Sundance’s internal use and will almost never appear at the user level. You will sometimes work with `Cell` objects directly, for instance when probing the value of a function at a point during postprocessing.

3.1.1 Mesh I/O

There are almost as many mesh file formats as there are engineers, and it would be foolish to try to build support for file I/O directly into the `Mesh` object. Sundance uses an extensible `MeshReader` class heirarchy to provide an interface for reading from mesh formats. The current version of Sundance supports readers for three mesh formats: a native Sundance text format, Shewchuk’s Triangle format, and Sandia’s Exodus II format. If you want to support some other mesh format you will have to implement your own `MeshReaderBase` subtype.

Using a `MeshReader` is very simple. You create a `MeshReader` object as a handle to an appropriate subtype, and then you call the `readMesh()` method to return a `Mesh` object. The following code reads a mesh in Shewchuk’s Triangle format from files `tBird.1.poly` and `tBird.1.ele`:

```
MeshReader reader = new ShewchukMeshReader("tBird.1");
Mesh mesh = reader.getMesh();
```

Similarly, to write a mesh to Triangle format one does

```
MeshWriter writer = new ShewchukMeshWriter("myMesh");
writer.writeMesh();
```

3.1.2 Mesh generator interface

Class `MeshGenerator` provides an interface for mesh generators, and there are implementations for building several simple mesh types. In principle it would be possible to connect a powerful third-party mesh generator to Sundance through the mesh generator interface, but it is generally simpler to have the mesher write the mesh to a file which can be read by a `MeshReader` object.

The most common use of `MeshGenerator` is to build toy meshes for test problems. The three built-in `MeshGenerator` subtypes are

- `LineMesher` meshes a line
- `RectangleMesher` meshes a rectangle with triangles
- `RectanglerQuadMesher` meshes a rectangle with quadrilaterals

3.2 Cell sets

A `CellSet` object is used to define a set of cells on which an equation or boundary condition is to be applied. A `CellSet` can be defined independently of any particular mesh; instead of a list of cells, it is a condition or set of condition that can be used to extract a list of cells from a mesh.

3.2.1 The set of all maximal cells

The `MaximalCellSet` object identifies all maximal cells in a mesh. The constructor has no arguments:

```
CellSet maxCells = new MaximalCellSet();
```

3.2.2 The set of all boundary cells

A `BoundaryCellSet` object identifies all boundary cells of dimension $N - 1$. For example, in a 3D problem a `BoundaryCellSet` will contain all 2D cells on the boundary, but not lines or points that happen to lie on the boundary.

The constructor has no arguments:

```
CellSet boundaryCells = new BoundaryCellSet();
```

3.2.3 Defining subsets

Given a cell set, we can use the `subset()` method to define a condition that can extract a subset of the original cell set. The condition can be a mathematical equation or inequality that must be satisfied by any cell to be accepted into the set, or it can be a string label. In "real world" problems the most common condition for defining a cell set will be a label that is associated with the cells by the code that produced the mesh.

```
CellSet boundary = new BoundaryCellSet();
CellSet wall = boundary.subset(``wall'');
CellSet arc = boundary.subset(x*x + y*y == 1.0 && x < 0.5);
```

3.2.4 Logical operations on cell sets

Cell sets can be created by doing set operations – union and intersection – on two or more existing cell sets. Union and intersection are represented by the overloaded addition (+) and logical AND (`&&`) operators.

3.2.5 Material properties

3.2.6 Creating meshes from scratch

This section can be skipped on a first reading.

3.3 Cells

Class `Cell` provides a unified interface for all mesh entities such as points, lines, triangles, and tets.

Chapter 4

Weak Equations

4.1 Specifying the domain on which an equation holds

4.2 Quadrature Families

The integrals in a Sundance weak form are done by numerical integration, or quadrature. With quadrature, one approximates an integral by a finite sum of weighted function values,

$$\int f(x) \quad dx \approx \sum_{i=1}^{N} w_i f(x_i) \tag{4.1}$$

where the evaluation points x_i and weights w_i are chosen with care and cleverness to provide an accurate approximation.

What is relevant to user-level Sundance code is how one can specify a suite of quadrature rules to be used for a given weak form. Notice that it will not suffice to specify a quadrature rule, because a given term may be integrated on several different cell types. For example, a mesh may contain both quad cells and triangle cells, and the two different cell types will require two different quadrature rules. What is needed is a specification of a *family* of quadrature rules rather than a single rule. The user-level specifier of a family of quadrature rules is the `QuadratureFamily` object. The `buildQuadraturePoints()` method of `QuadratureFamily` returns a set of quadrature points and weights appropriate to a given cell type. The user picks a quadrature family by selecting the appropriate subtype of `QuadratureFamilyBase` and supplying the desired constructor arguments. For example,

```
QuadratureFamily gauss4 = new GaussianQuadrature(4);
```

creates an object that can produce a 4-th order Gaussian quadrature rule for any cell type.

4.2.1 Gaussian Quadrature

Gaussian quadrature rules specify both points and weights to give optimal accuracy for all polynomials through a given degree. Gaussian quadrature rules for a line can be derived from the properties of the Legendre polynomials; see any textbook on numerical analysis for a discussion. Gaussian quadrature rules for quadrilaterals and bricks can be formed as "tensor products" of line rules. The development of Gaussian quadrature rules for triangles and tetrahedra is an ongoing research area; an online literature survey through 1998 can be found at Steve Vavasis' quadrature and cubature page. Symmetric Gaussian quadrature rules through moderate order have been developed for triangles by Dunavant[?] and for tetrahedra by Jinyun[?]. A summary of the quadrature rules that will be generated by Sundance's `GaussianQuadrature` object is given in the table below.

Cell type	**Available orders**	**Reference**	Comments
Line	all	e.g. Hughes[**?**]	
Triangle	1-12	Dunavant[**?**]	Orders 3,7, and 11 have negative weights.
Quad	any		Tensor project of two line rules.
Tet	1-6	Jinyun[**?**]	Order 3 has a negative weight.
Brick	any		Tensor project of three line rules.

Chapter 5

Boundary Conditions

There are many ways to apply boundary conditions in a finite element simulation. To begin with, the way a boundary condition gets written depends strongly on the way the weak problem has been formulated; boundary conditions will be written quite differently in least-squares formulations than in Galerkin formulations.

5.1 General considerations

5.1.1 Specifying where a boundary condition gets applied

In Sundance, geometric subdomains are identified using `CellSet` objects. The surface on which a BC is to be applied is specified by passing as an argument the `CellSet` representing that surface.

5.1.2 Nonlinear Boundary Conditions

Some problems will have nonlinear boundary conditions, for example in radiative heat transfer from a convex surface the heat flux is σT^4 where σ is the Stefan-Boltzmann constant. Sundance requires you to handle nonlinear BCs in the same way you handle nonlinear PDEs: by iterative solution of a linearized problem. The linearization will depend on the iterative method used.

5.1.3 Nonlocal Boundary Conditions

Nonlocal boundary conditions arise naturally in problems such as radiative heat transfer from non-convex surfaces, and can also occur as far-field boundary conditions in acoustics, electromagnetics, or fluid mechanics. Nonlocal boundary conditions are handled with integral equations, and are not supported in the current version of Sundance.

5.2 BCs for Galerkin and Petrov-Galerkin Formulations

In a Galerkin or Petrov-Galerkin formulation, integration by parts yields boundary terms involving a test function and the derivative of an unknown function.

5.2.1 Neumann Boundary Conditions

Neumann BCs specify the value of a normal derivative, or some combination of derivatives, along a boundary surface. They arise in problems where a flux has been specified on a boundary; for example, a heat flux in heat transfer or a surface traction (momentum flux) in solid mechanics. An important and quite common special case is a homogeneous Neumann BC, where the boundary flux is zero; examples are insulating surfaces in heat transfer and free surfaces in solid mechanics. Homogeneous Neumann BCs are often called **natural BCs** in the finite elements literature, and they have a particularly simple representation.

Writing Neumann BCs in Galerkin formulations is extremely simple. After integration by parts, we will have boundary terms involving test functions times derivatives of our unknowns. To apply a Neumann BC, we simply replace the derivatives appearing in those boundary terms with their value determined by the BC. Homogeneous Neumann BCs, are a particularly simple special case because the boundary integrals go to zero, letting us satisfy the BC by simply ignoring the boundary integrals.

5.2.2 Robin Boundary Conditions

Robin BCs, often called boundary conditions of the third kind, specify a linear combination of a field value and its normal derivative. Robin BCs occur, for example, on a surface from which heat is carried by convection. Robin BCs are handled similarly to Neumann BCs, in that we replace the derivative in a surface term with its value computed from the BC; however, with a Robin BC we replace the derivative with a linear expression involving the unknown field rather than with a known expression.

5.2.3 Dirichlet Boundary Conditions

Dirichlet boundary conditions specify the value of a field on a boundary segment. Some physical examples are specifying the temperature on a surface that is in contact with a heat bath, or specifying that a viscous fluid "sticks" to a surface.

In Galerkin formulations with bases where degrees of freedom correspond to nodal values, we can enforce Dirichlet boundary conditions on certain nodes by replacing the PDE at those nodes with an equation representing the Dirichlet BCs. This can be done nodewise or weakly over the boundary cell; note that nodewise BC application is a special case of weak BC application obtained by using a nodal quadrature rule.

In Sundance we write such a boundary condition using the `EssentialBC` object. The most general constructor for an `EssentialBC` takes as arguments

- the `CellSet` on which the BC is to be applied
- a weak form of the BC.
- a quadrature family.

5.2.4 Optional: More about Dirichlet BCs

The replacement scheme for Dirichlet boundary conditions described in the previous subsection above may seem ad-hoc, however it can be given a sound mathematical foundation as a limit of a Robin boundary condition.

Without loss of generality, we'll use as a model problem the Laplace equation with Dirichlet conditions $u = u_0$ on the entire boundary:

$$\nabla^2 u = 0 \text{ in } \Omega \tag{5.1}$$

$$u = u_0 \text{ on } \Gamma. \tag{5.2}$$

As usual, we form a weak equation by multiplying by a test function v and integrating, giving

$$-\int_\Omega \nabla v \cdot \nabla u + \int_\Gamma v \frac{\partial u}{\partial n} = 0 \tag{5.3}$$

which must hold for all v in a suitable subspace. We now impose a Robin boundary condition

$$\epsilon \frac{\partial u}{\partial n} = u - u_0, \tag{5.4}$$

which in the limit $\epsilon \to 0$ approaches the Dirichlet condition $u = u_0$. With this BC, the weak equation becomes

$$-\int_\Omega \nabla v \cdot \nabla u + \frac{1}{\epsilon} \int_\Gamma v\,(u - u_0) = 0 \tag{5.5}$$

At this point, we could simply take a small but nonzero ϵ and solve the problem. This is equivalent to a penalty method for imposing the BC, and unfortunately leads to poorly conditioned linear systems. We want to do better, and take the limit $\epsilon \to 0$ so that the Dirichlet BC is satisfied exactly, yet we also want to avoid ill-conditioning. As we will show, we can often have it both ways, taking the limit without ill-conditioning, but the argument must be developed carefully because the way in which that limit is taken is important.

We will discretize the system before taking the limit. We expand u in a series of basis functions,

$$u(x) = \sum_j u_j \phi_j(x) \tag{5.6}$$

and evaluate **??** wiuth each member of $\{\phi\}$ as test functions. The discrete equations are

$$\left[\mathbf{A} + \frac{1}{\epsilon}\mathbf{B}\right] \cdot \mathbf{u} = \frac{1}{\epsilon}\mathbf{B} \cdot \mathbf{u}_0 \tag{5.7}$$

where the matrices are defined as

$$\mathbf{A}_{ij} = -\int_\Omega k\nabla\phi_i \cdot \nabla\phi_j \tag{5.8}$$

$$\mathbf{B}_{ij} = \int_\Gamma \phi_i \phi_j \tag{5.9}$$

At this point we make a critical assumption: that we are using either Lagrange or Serendipity basis functions. These basis functions are *nodal*, by which we mean we can make an association between basis functions and mesh nodes, and then define the basis functions in such a way that the i-th basis function $\phi_i(x)$ is zero at every node but the i-th. Furthermore, these basis sets have the important property that *only those basis functions ϕ_i associated with nodes on the surface Γ have nonzero values on the surface Γ.* We can now partition the equations and variables according to whether each row or column is associated with a node on Γ. Notice that the dangerous denominator appears only in terms involving B, whose elements are obtained by integrating over Γ. Thus all elements of $\mathbf{B}$ involving nodes not on Γ are zero. Marking elements associated with Γ with the superscript D and all others with the superscript I, we can write each matrix as a block matrix, e.g.,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}^{II} & \mathbf{A}^{ID} \\ \mathbf{A}^{DI} & \mathbf{A}^{DD} \end{bmatrix}. \tag{5.10}$$

In particular, from the argument above we know that all off-boundary elements of $\mathbf{B}$ are zero,

$$\mathbf{B} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}^{DD} \end{bmatrix}. \tag{5.11}$$

With this notation, we can write out the system,

$$\begin{bmatrix} \mathbf{A}^{II} & \mathbf{A}^{ID} \\ \mathbf{A}^{DI} & \mathbf{A}^{DD} + \frac{1}{\epsilon}\mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{u}^I \\ \mathbf{u}^D \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \frac{1}{\epsilon}\mathbf{B}\cdot\mathbf{u}_0 \end{bmatrix} \tag{5.12}$$

Multiplying the second row by ϵ removes the explosive denominator

$$\begin{bmatrix} \mathbf{A}^{II} & \mathbf{A}^{ID} \\ \epsilon\mathbf{A}^{DI} & \epsilon\mathbf{A}^{DD} + \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{u}^I \\ \mathbf{u}^D \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{B}\cdot\mathbf{u}_0 \end{bmatrix} \tag{5.13}$$

so that it is safe to take the limit $\epsilon \to 0$, yielding the well-behaved system

$$\begin{bmatrix} \mathbf{A}^{II} & \mathbf{A}^{ID} \\ \mathbf{0} & \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{u}^I \\ \mathbf{u}^D \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{B}\cdot\mathbf{u}_0 \end{bmatrix} \tag{5.14}$$

This entire process can be summarized as such: for surfaces on which Dirichlet BCs apply, we simply *replace* all equations by the Dirichlet boundary term. The bookkeeping for this would be daunting if we had to keep track of off-boundary and on-boundary nodes in the equation specification. However, in Sundance the specification that a term is to be handled in this special way, via replacement or, more precisely, via limit and rescaling, is done by putting the term inside an `EssentialBC` object rather than an `Integral` object. The Sundance computational kernel will figure out which rows get replaced and rescaled.

Discontinous Dirichlet BCs

It is not uncommon to encounter boundary conditions in which a field is set to a function that is discontinuous. An example is the lid-driven cavity problem of fluid mechanics, in which the x-component of velocity is zero along the side walls and nonzero along the top lid. The x velocity is discountinuous at the two top corners, and it is necessary to choose a scheme for assigning the boundary value at the points of discontinuity. Several ideas suggest themselves:

- Pick one of the two boundary conditions and apply it at the corner point.
- Average the two boundary conditions.

5.3 BCs for Least-Squares Formulations

BCs in least-squares formulation can be applied either as essential BCs or by asking that the BC be satisfied in a least-squares sense.

5.3.1 Dirichlet Boundary Conditions

5.3.2 Neumann Boundary Conditions

In a first-order least-squares formulation, the fluxes have been introduced explicitly as new unknowns. Thus a flux boundary condition becomes a Dirichlet boundary condition for a flux variable. This Dirichlet boundary condition can be applied either as an essential boundary condition or in a least-squares sense.

5.3.3 Robin Boundary Conditions

As with Neumann boundary conditions, in a first-order least squares problem we can replace the derivative term in a Robin boundary condition with the appropriate flux variable, yielding a Dirichlet boundary condition involving a field variable and a flux variable. This Dirichlet boundary condition can be applied either as an essential boundary condition or in a least-squares sense.

Chapter 6

Linear Problem Assembly and Solution

The core capability of Sundance is the construction of a discrete system of linear equations based on information contained in a symbolic representation of a problem. The class of equations that can be solved directly using this core capability are static (meaning time-independent) linear problems. Time-dependent problems and nonlinear problems are solved by iterating a sequence of static linear problems; the solution of such problems is described later in **??** and **??**.

This chapter describes how to set up a static linear problem in Sundance, how to control the formation of the discrete linear system of equations, and how to interface to software components that can solve that system. The object encapsulating a static linear problem is `StaticLinearProblem`.

6.1 The StaticLinearProblem object

A `StaticLinearProblem` is constructed with all information needed to form a discrete linear system of equations. The most obvious things that will be needed for that specification are the equation, boundary conditions, and a mesh on which the problem will be solved. Additionally, you must specify the *order* in which the equations must be assembled and in which the unknowns will be listed, and you must also specify the vector and matrix representation that will be used to store the system matrix and vector.

6.2 Specification of vector and matrix representation

There are many high-quality numerical linear algebra packages in use, so Sundance is designed to allow third-party linear algebra packages to be imported as plugins. All numerical linear algebra in Sundance is done using the Trilinos Solver Framework (TSF), and the TSF in turn supports plugins of third-party types.

User specification of a linear algebra representation is done by means of a `TSFVectorType` object. This object knows how to build a vector space given a mesh and set of functions, and the vector space in turn knows how to build a vector of the appropriate type.

6.3 Specification of row and column space ordering

The order in which equations and unknowns are written can make a difference in the performance of a linear solver, and in keeping with the goal of flexibility, Sundance gives you the ability to specify this ordering. In order to understand how Sundance's ordering specification works, let's look into how Sundance decides unknown and equation numbering.

Given a mesh and a set of unknowns, the Sundance discretization engine will traverse the mesh one maximal cell at a time and find all unknowns associated with that cell and its facets. In a problem with multiple unknowns, say velocity, pressure, and temperature, there can be more than one unknown associated with a cell; if so, the unknowns are assigned in the order that their associated `UnknownFunction` objects are listed in the `StaticLinearProblem` constructor. This scheme gives us two ways to control the unknown ordering:

- **Cell ordering** specifies the order in which cells are encountered as the mesh is traversed.
- **Function ordering** specifies the order in which different functions are listed within a singlecell.

6.3.1 Cell ordering

Cell ordering is controlled by giving the linear problem constructor a `CellReorderer` object. Currently, there are two subtypes of `CellReorderer`,

- `RCMCellReorderer` uses the reverse Cuthill-McKee reordering algorithm (e.g., Saad [**?**]). The RCM algorithm is a modified breadth-first search with desirable behavior during matrix factoring.
- `IdentityCellReorderer` uses the original ordering used by the mesh, i.e., it does no reordering.

The default is `RCMCellReorderer`, and it is a good general choice. You might use `IdentityCellReorderer` in cases where your mesh already has a favorable cell ordering, saving the (small) expense of doing an unnecessary reordering.

The cell reordering system is extensible; your favorite reordering algorithm can be added to Sundance by writing a new `CellReorderer` subtype.

The same cell reordering scheme is used for equation numbering (rows) and unknown numbering (columns). Thus, cell reorderings are always symmetric.

6.3.2 Function ordering

Function ordering is controlled by the order in which test or unknown functions appear in the linear problem constructor. For example, if `ux`, `uy`, and `p` are unknowns we can order them as: `List(ux, uy, p)`, or as `List(p, ux, uy)` or any of the other permutations. A list with the desired ordering is given to the `StaticLinearProblem` constructor,

```
StaticLinearProblem problem(mesh, eqn, bc, List(vx, vy, q), List(ux,
uy, p), vecType);
```

Notice that the test functions need not have the same ordering as their corresponding unknowns: a nonsymmetric ordering such as

```
StaticLinearProblem problem(mesh, eqn, bc, List(vx, vy, q), List(p,
ux, uy), vecType);
```

is possible.

6.4 Block structuring

It is possible to group unknowns and equations into **blocks**, in which case the stiffness matrix becomes a block matrix with each block being an independent object.

As with function ordering, block structuring is specified by organization of the unknown and test function arguments to the `StaticLinearProblem` constructor.

```
Array<Block> unkBlocks = List(Block(List(U, V), petraType), Block(P, petraType));
```

6.5 Solving the problem

Once the problem has been specified and assembled, the final step of getting the solution is quite simple to code: one just calls the `solve()` method, which returns a Sundance expression containing the solution in the form of a discrete function. The solve method can be called with no arguments, in which case the matrix type's default solver is used. Alternatively `solve()` can be called with a solver as an argument.

6.5.1 Specification of solver

The solver is specified by passing a `TSFLinearSolver` object to the linear problem's `solve()` method. The solver must be one that works with the matrix representation you've chosen.

6.6 Accessing the stiffness matrix and load vectors

In solver development or debugging one will sometimes want to bypass the problem's `solve()` method and work directly with the problem's stiffness matrix and load vector. These can be accessed with the `getOperator()` and `getRHS()` methods which return a TSFLinearOperator and TSFVector respectively.

```
// create a SLP given mesh, eqn, and bc defined somewhere
StaticLinearProblem prob(mesh, eqn, bc, u, v);

// get the stiffness matrix and load vector as TSF objects
TSFLinearOperator K = prob.getOperator();
TSFVector f = prob.getRHS();

// write to matlab
TSFMatrixWriter writer = new MatlabMatrixWriter(‘‘myProb.m’’);
writer.write(K);
writer.write(f);
```

Chapter 7

Solving Nonlinear Problems

The general approach to the numerical solution of a nonlinear PDE is iterative: find a linearized approximation to the PDE given some apprimxate solution, and use the solution of this linear problem to obtain an improved solution. You can write the iterative loop directly in user-level Sundance code, or you can use a nonlinear solver package.

There are a number of ways to linearize a PDE. Picard's method, or fixed-point iteration is often the simplest: in any nonlinear terms, simply replace the unknown u by a previous guess u_{k-1}. In the next iteration, the solution of the current linear problem is used as the guess u_k in the nonlinear terms. While simple, Picard's method often converges slowly and in the unlucky event the solution is an unstable fixed point, the method will fail regardless of initial guess. Newton's method is probably the best-known method for solving a nonlinear system of equations: use the current residual and the current gradient to compute a step that would drive the residual to zero were the problem linear. The linearization for Newton's method is a first-order Taylor expansion of the original PDE about a trial solution u_k. With a good initial guess Newton's method is very fast – quadratically convergent – and it can be modified to have good – though not always fast – global convergence properties. Variants of both these schemes are possible; for example damping can be added to stabilize a fixed-point iteration, or an approximate Jacobian can be used in Newton's method.

To illustrate several approaches to solving a nonlinear PDE with Sundance, we consider the Poisson-Boltzmann equation in one dimension

$$\nabla^2 u = e^{-\beta u}. \tag{7.1}$$

The Poisson-Boltzmann equation is used to find the equilibrium potential of a large system of particles mututally interacting with a Coulomb force. The one-dimensional equation has been applied to calculating the self-consistent gravitational potential of the galactic disk (Spitzer [**?**]). The potential is assumed to be symmetric about the midplane $x = 0$ and has the asymptotic behavior $u \sim |x|$ at large $|x|$. It can be shown that the solution to the one-dimensional problem with these boundary conditions is

$$u(x) = 2 \log \cosh \left(2^{-1/2} x\right) \tag{7.2}$$

In the example code, we will solve the problem on a finite interval $[0, L]$ and use the known solution to give the upper boundary condition $u(L) = 2 \log \cosh \left(2^{-1/2} L\right)$. The symmetry boundary condition at $x = 0$ is imposed by setting an homogeneous Neumann BC at that point.

We will show several methods of solving the Poisson-Boltzmann equation in Sundance. The first two examples show Picard and Newton's method coded "inline" in user-level Sundance code. The next two show Picard and a modified, globally convergent Newton's method wrapped in high-level objects. Unless you are a nonlinear solver developer or have a problem that requires a custom solver, you will most often use the high-level nonlinear solver objects.

Picard iteration

Assume we have a function u_k that is a good approximation to the solution u. A next approximation u_{k+1} is given by the linear equation

$$\nabla^2 u_{k+1} = \exp\left(-\beta u_k\right). \tag{7.3}$$

In this problem boundary conditions on the linearized equation are simply our original BCs applied to u_{k+1}

Newton's method and variants

In Newton's method, we use gradient information to compute a step between a trial soluton u_k and its update u_{k+1}. The full Newton step δu is given by

$$F(u_k) + \nabla F(u_k) \cdot \delta u = 0. \tag{7.4}$$

For the Poisson-Boltzmann problem, the Newton linearization is

$$\nabla^2(u_k + \delta u) = \exp\left(-\beta u_k\right)\left(1 - \beta\delta u\right) \tag{7.5}$$

Linearization

It is possible to have Sundance automate the linearization of a nonlinear equation. Automated linearization is restricted to full Newton linearization; alternative linearization schemes such as Oseen must be done by hand.

The `linearization(u, u0)` methods of `Expr` and `EssentialBC` are used to return a new linear expression or BC. Linearization is always done about an initial guess `u0`, which must be a discrete function with the same structure as the unknown argument `u`. The new expression has a new unknown function for the Newton step, or differential, which will have the same structure as the original unknown `u`. Calling `linearization()` on a linear expression simply obtains the same linear expression, but in terms of the Newton step for the original unknown. Note that if *either* the PDE or BC are nonlinear, both must be linearized in order to transform both into equations for the Newton step.

Example: Poisson-Boltzmann Equation

The Poisson-Boltzmann equation can be linearized as follows.

```
Expr eqn = Integral((grad*u)*(grad*v) + exp(-u)*v);
EssentialBC bc = EssentialBC(top, (u - uBC)*v);

Expr linearizedEqn = eqn.linearization(u, u0);
EssentialBC linearizedBC = bc.linearization(u, u0);
```

The resulting expression and BC are equations for the Newton step, accessible as an unknown function through the `differential()` method on the original unknown,

```
Expr du = u.differential();
```

Complete code for the solution of the Poisson-Boltzmann equation (7.1) is shown below.

```
#include "Sundance.h"

/** \example inlinePoissonBoltzmann1D.cpp
 * Solve the Poisson-Boltzmann equation \f$\nabla^2 u = e^-u$ on the unit
 * line with boundary conditions:
 * Left: Natural, du/dx=0
 * Right: Dirichlet u = 2 log(cosh(1/sqrt(2)))
 *
 * The solution is 2 log(cosh(x/sqrt(2))).
 *
 * The problem is nonlinear, so we use Newton's method to iterate
 * towards a solution.
 *
 */

int main(int argc, void** argv)
{
  try
    {

      Sundance::init(&argc, &argv);

      /* create a simple mesh on the unit line */
      double L=1.0;
      int n = 10;
      MeshGenerator mesher = new PartitionedLineMesher(0.0, L, n);
      Mesh mesh = mesher.getMesh();

      /* define an expression representing the x-coordinate function */
      Expr x = new CoordExpr(0);

      /* create a cell set representing the right boundary */
      CellSet boundary = new BoundaryCellSet();
      CellSet right = boundary.subset( x == L );

      /* create a discrete space on the mesh */
      TSFVectorSpace discreteSpace
        = new SundanceVectorSpace(mesh, new Lagrange(2));

      /* create an expression for the initial guess. This will be reused as the
       * starting point for each newton step. Assume u(x)=x as an initial
       * guess, and discretize it.
       */
      Expr u0 = new DiscreteFunction(discreteSpace, x);

      /* create symbolic objects for test and unknown functions. At each newton
       * step we will solve a linearized equation for a step du, so our
       * unknown is du. */
      Expr u = new UnknownFunction(new Lagrange(2), "du");
      Expr v = new TestFunction(new Lagrange(2), "du");

      /* create a differential operator representing the x-derivative. */
      Expr dx = new Derivative(0);

      /* linearized weak equation for the step du */
      Expr nonlinearEqn = Integral((dx*u)*(dx*v) + v*exp(-u));
      Expr linearizedEqn = nonlinearEqn.linearization(u, u0);
      Expr du = u.differential();

      /* Dirichlet boundary condition */
      double uBC = 2.0*log(cosh(L/sqrt(2.0)));
      EssentialBC bc = EssentialBC(right, (u-uBC)*v) ;
      EssentialBC linearizedBC = bc.linearization(u, u0);

      /* linear problem for the step du */
      StaticLinearProblem prob(mesh, linearizedEqn, linearizedBC, v, du);

      /* create linear solver */
      TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
      TSFLinearSolver solver = new BICGSTABSolver(1.0e-12, 1000);


      NewtonLinearization newton(prob, u0, solver);
      Expr soln = newton.solve(NewtonSolver(solver, 8, 1.0e-12, 1.0e-12));

      // compare to exact solution
      Expr exactSoln = 2.0*log(cosh(x/sqrt(2.0)));
      Expr error = new DiscreteFunction(discreteSpace, soln-exactSoln);

      /* write to matlab */
      string filename = "pb1D." + TSF::toString(MPIComm::world().getRank())
        + ".dat";
      FieldWriter writer = new MatlabWriter(filename);
      writer.writeField(soln);

      // compute the norm of the error
      double errorNorm = (exactSoln - soln).norm(2);
      double tolerance = 1.0e-4;
      TSFOut::printf("error = %g\n", errorNorm);

      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
    }
  catch(exception& e)
    {
      Sundance::handleError(e, __FILE__);
    }
  Sundance::finalize();

}
```

Chapter 8

PDE-Constrained Optimization

Traditional PDE codes solve one of a specific class of PDEs with little hope of obtaining the gradients, adjoints, or Hessians needed for PDE-constrained optimization. Even with modern PDE frameworks such as SIERRA and Nevada, it will require considerable development effort to obtain these quantities. Thus, for optimization with existing PDE codes, one must use the PDE solver as a "black box,"and we are restricted to relatively inefficient optimization algorithms. One of the principal reasons for designing Sundance is to "open up" the PDE solver in such a way that higher-performance algorithms become practical.

8.1 A PDE-constrained optimization example

We now show a simple example of how to use Sundance to set up an optimization problem with a PDE constraint. Consider the Poisson equation with source terms parameterized with a design variable α,

$$\nabla^2 u = \sum_k \alpha_k \sin k\pi x. \tag{8.1}$$

A simple optimization problem is to choose α such that the state function u is a good fit to a target function $\hat{u}$. This target-fitting problem can be posed as a least-squares problem with objective function

$$f(\alpha) = \frac{1}{2}\int_\Omega (u(\alpha) - \hat{u})^2 + \frac{R}{2}\sum_k \alpha_k^2 \tag{8.2}$$

where R sets the control cost. As written, we could solve this problem with a pattern search method in which we solve 8.1 for u at each function evaluation. Alternatively, we can let the states become independent variables, but impose equation 8.1 as a constraint. In that case, we have a Lagrangian

$$L(\alpha, u, \lambda) = \frac{1}{2}\int_\Omega (u - \hat{u})^2 + \frac{R}{2}\sum_k \alpha_k^2 - \int_\Omega \nabla u \cdot \nabla \lambda - \sum_k \alpha_k \int_\Omega \lambda \sin k\pi x \tag{8.3}$$

where λ is a Lagrange multiplier. The necessary condition for solving the optimization problem is that the variations of the Lagrangian with respect to α, u, and λ are all zero.

This example is a quadratic program with an equality constraint, and the solution is obtained with a single linear solve of the KKT system. However, the KKT system is indefinite and is most efficiently solved using a block Schur complement method.

8.1.1 Sundance problem specification

With Sundance, all we need do to pose this problem is to write the Lagrangian using Sundance symbolic objects.

Note that in this problem, the state variable u and Lagrange multiplier λ are unknown functions defined with a finite-element basis. However, the design parameters are unknown "global" parameters, defined independently of the mesh. In Sundance, mesh-based unknowns are `UnknownFunction` expression subtypes and global unknowns are `UnknownParameter` expression subtypes. To optimize performance in parallel, Sundance imposes the restriction that all global unknowns must appear in a separate block from any meshed unknowns; that block is then replicated across processors while blocks containing meshed unknowns are distributed. Matrix blocks mapping between the global unknown space and a meshed unknown space are implemented with multivectors, in which each row (or column, depending on the orientation of the block) is a distributed vector. The specification of the unknowns and block structure for this problem is done with the following Sundance code:

```
Expr u = new UnknownFunction(new Lagrange(2));
Expr v = u.variation();

Expr lambda = new UnknownFunction(new Lagrange(2));
Expr mu = lambda.variation();

Expr alpha1 = new UnknownParameter();
Expr alpha2 = new UnknownParameter();
Expr alpha3 = new UnknownParameter();
Expr alpha = List(alpha1, alpha2, alpha3);
Expr beta = alpha.variation();

TSFVectorType petra = new PetraVectorType();
TSFVectorType dense = new DenseSerialVectorType();

TSFArray<Block> unks = tuple(Block(alpha, dense), Block(u, lambda, petra));

TSFArray<Block> vars = tuple(Block(beta, dense), Block(mu, v, petra));
```

Once the unknowns have been specified, we can write out the objective function and Lagrangian in symbolic form:

```
Expr objectiveFunction = 0.5*Integral(pow(u-target, 2.0))
      + 0.5*alpha*alpha;
Expr lagrangian = objectiveFunction - Integral((dx*u)*(dx*lambda))
      - Integral(lambda*forcing);
```

The equation set can be obtained by taking symbolic variations of the Lagrangian.

```
Expr eqn = lagrangian.variation(List(u, lambda, alpha));
```

We will solve the system using a Schur complement solver, using TSF's block manipulation capabilities. The user-level code to specify a Schur complement solver for a 2 by 2 block system is

```
TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
TSFLinearSolver innerSolver = new BICGSTABSolver(1.0e-12, 1000);
TSFLinearSolver outerSolver = new BICGSTABSolver(1.0e-10, 1000);

TSFLinearSolver solver = new SchurComplementSolver(innerSolver, outerSolver);
```

Finally, we show complete source code for the PDE-constrained optimization example.

```
#include "Sundance.h"


/**
 *
 */
```

```
int main(int argc, void** argv)
{
  try
    {
      Sundance::init(&argc, &argv);

      /*
        Create a mesh object. In this example, we will use a built-in method
        to create a uniform mesh on the unit line. In more realistic problems
        we would use a mesher to create a mesh, and then read the mesh using
        a MeshReader object.
      */
      int n = 10;
      const double pi = 4.0*atan(1.0);
      MeshGenerator mesher = new LineMesher(0.0, pi, n);
      Mesh mesh = mesher.getMesh().getSubmesh();

      /* Define a symbolic object to represent the x coordinate function. */
      Expr x = new CoordExpr(0);

      Expr psi = List(sin(x), sin(2.0*x), sin(3.0*x));

      Expr target = sin(x);

      /*
       * Define a cell set that contains all boundary cells
       */
      CellSet boundary = new BoundaryCellSet();
      /*
       *  Define a cell set that includes all cells at position x=0.
       */
      CellSet left = boundary.subset( fabs(x - 0.0) < 1.0e-10 );

      /*
       *  Define a cell set that includes all cells at position x=1.
       */
      CellSet right = boundary.subset( fabs(x - pi) < 1.0e-10 );



      /*
        Define an unknown function and its variation. The constructor
        argument is the basis family with which the function will be
        represented, in this case second-order Lagrange (nodal) polynomials.
      */
      Expr u = new UnknownFunction(new Lagrange(2));
      Expr v = u.variation();

      Expr lambda = new UnknownFunction(new Lagrange(2));
      Expr mu = lambda.variation();

      Expr alpha1 = new UnknownParameter();
      Expr alpha2 = new UnknownParameter();
      Expr alpha3 = new UnknownParameter();
Expr alpha = List(alpha1, alpha2, alpha3);
Expr beta = alpha.variation();

TSFVectorType petra = new PetraVectorType();
TSFVectorType dense = new DenseSerialVectorType();

TSFArray<Block> unks = tuple(Block(alpha, dense), Block(u, lambda, petra));

TSFArray<Block> vars = tuple(Block(beta, dense), Block(mu, v, petra));

Expr forcing = alpha * psi;



/*
  Define the differentiation operator of order 1 in direction 0.
*/
Expr dx = new Derivative(0);


Expr objectiveFunction = 0.5*Integral(pow(u-target, 2.0))
  + 0.5*alpha*alpha;

Expr lagrangian = objectiveFunction - Integral((dx*u)*(dx*lambda))
  - Integral(lambda*forcing);

Expr eqn = lagrangian.variation(List(u, lambda, alpha));



/*
  Now specify the boundary conditions on the left and right CellSets.
 */

EssentialBC bc =
  EssentialBC(left, u*mu + v*lambda) && EssentialBC(right, u*mu + v*lambda);




/*
  Create a solver object: stablized biconjugate gradient solver
*/
TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
TSFLinearSolver innerSolver = new BICGSTABSolver(1.0e-12, 1000);
TSFLinearSolver outerSolver = new BICGSTABSolver(1.0e-10, 1000);


TSFLinearSolver solver = new SchurComplementSolver(innerSolver, outerSolver);


/*
  Combine the geometry, the variational form, the BCs, and the solver
  to form a complete problem.
```

```
      */
      StaticLinearProblem prob(mesh, eqn, bc, vars, unks);
      prob.printRowMaps();
      mesh.printCells();

      /*
        solve the problem, obtaining the solution as a (discrete) Expr object
      */
      Expr soln = prob.solve(solver);

      /*
        write the solution in a form readable by matlab
      */
      FieldWriter writer = new MatlabWriter();
      cerr << "u" << endl;
      writer.writeField(soln[1][0]);
      cerr << "lambda" << endl;
      writer.writeField(soln[1][1]);

      cerr << soln[0] << endl;

      /*
        compute the error and represent as a discrete function
      */
      Expr exactSoln = sin(x);

      /*
        compute the norm of the error
      */
      double errorNorm = (soln[1][0] - exactSoln).norm(2);
      double tolerance = 1.0e-10;

      /*
        decide if the error is within tolerance
      */
      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
      Testing::timeStamp(__FILE__, __DATE__, __TIME__);

    }
  catch(exception& e)
    {
      TSFOut::println(e.what());
      Testing::crash(__FILE__);
      Testing::timeStamp(__FILE__, __DATE__, __TIME__);
    }
  Sundance::finalize();
}
```

Chapter 9

Solving timestepping problems

Currently, Sundance has no high-level support for transient simulations. However, it is not difficult to code simple timestepping schemes directly in Sundance.

Consider Crank-Nicolson (BE) time discretization for the transient heat equation. If we discretize in time but leave space undiscretized for the moment, the step from u_i to u_{i+1} is given by the PDE

$$u_{i+1} - u_i = \frac{1}{2}\delta t \left[\nabla^2 u_{i+1} + \nabla^2 u_i\right] \tag{9.1}$$

plus associated BCs. We can now solve this equation using Sundance, and the solution may be used as the starting value for the next step.

```
#include "Sundance.h"

/**
 * \example timeStepHeat1D.cpp
 *
 * This example shows how to do timestepping in Sundance. We solve the
 * transient heat equation in one dimension using Crank-Nicolson time
 * discretization. The time discretization is done at the symbolic level.
 * Spatial discretization is done via StaticLinearProblem, yielding system
 * matrices and vectors that can be used to march the problem in time.
 *
 * We solve the heat equation u_xx = u_t with boundary conditions
 * u(0)=u(1)=0 and initial conditions u(x,t=0)=sin(pi x). The solution
 * is u(x,t)=exp(-pi^2 t) sin(pi x).
 */

int main(int argc, void** argv)
{
  try
    {
      Sundance::init(&argc, &argv);

      /* create a simple mesh on the unit line */
      int n = 100;
      MeshGenerator mesher = new LineMesher(0.0, 0.5, n);
      Mesh mesh = mesher.getMesh();

      /* create unknown and variational functions */
      Expr delU = new TestFunction(new Lagrange(1));
Expr U = new UnknownFunction(new Lagrange(1));

/* create a differentiation operator */
Expr dx = new Derivative(0);

/* the initial conditions will be u0(x,t=0) = sin(pi*x).
 * create a coordinate expression to represent x, then
 * create sin(pi*x), and then project it onto a discrete function. */
Expr x = new CoordExpr(0);

double pi = 4.0*atan(1.0);

TSFVectorSpace discreteSpace
  = new SundanceVectorSpace(mesh, new Lagrange(1));

Expr u0 = new DiscreteFunction(discreteSpace, sin(pi*x));


/*
   set up crank-nicolson stepping with timestep = 0.02. The time
   discretization is done at the symbolic level, yielding
   an elliptic problem that we solve repeatedly for the updated
   solution at each time level.
*/

double deltaT = 0.02;
Expr cnStep = delU*(U - u0) + deltaT*(dx*delU)*(dx*(U + u0)/2.0);
Expr eqn = Integral(cnStep);

/* Define BCs to be zero at both ends */
CellSet boundary = new BoundaryCellSet();
CellSet left = boundary.subset( fabs(x - 0.0) < 1.0e-10 );
CellSet right = boundary.subset( fabs(x - 1.0) < 1.0e-10 );
EssentialBC bc = EssentialBC(left, delU*U);

/* create a solver object */
TSFPreconditionerFactory prec = new ILUKPreconditionerFactory(1);
TSFLinearSolver solver = new BICGSTABSolver(prec, 1.0e-14, 300);

/*
   put the time-discretized eqn into a StaticLinearProblem object
   which will do the spatial discretization.
*/
StaticLinearProblem prob(mesh, eqn, bc, delU, U);

/*
   Now, loop over timesteps, solving the elliptic problem for u at each
   step. At the end of each step, assign the solution solnU into u0.
   Because Exprs are stored by reference, the updating of u0 propagates
   to the copies of u0 in the equation set and in the
   StaticLinearProblem. The same StaticLinearProblem can be reused
   at all timesteps.
*/
int nSteps = 100;
```

```
      for (int i=0; i<nSteps; i++)
        {
          /* solve the problem */
          Expr soln = prob.solve(solver);
          TSFVector solnVec;
          soln.getVector(solnVec);
          u0.setVector(solnVec);
          /* write the solution at step i to a file */
          char fName[20];
          sprintf(fName, "timeStepHeat%d.dat", i);
          ofstream of(fName);
          FieldWriter writer = new MatlabWriter(fName);
          writer.writeField(u0);
          cerr << "[" << i << "]";
          /* flush the matrix and RHS values */
          prob.flushMatrixValues();
        }
      cerr << endl;

      /* compute the exact solution and the error */
      double tFinal = nSteps * deltaT;
      Expr exactSoln = exp(-pi*pi*tFinal) * sin(pi*x);

      /*
        compute the norm of the error
      */
      double errorNorm = (exactSoln-u0).norm(2);
      double tolerance = 1.0e-4;

      /*
        decide if the error is within tolerance
      */
      Testing::passFailCheck(__FILE__, errorNorm, tolerance);
    }
  catch(exception& e)
    {
      Sundance::handleError(e, __FILE__);
    }
  Sundance::finalize();
}
```

Chapter 10

Examples

The best way to learn Sundance is by example.

- Elastic deformation shows how to set up a multivariable problem, vector and tensor manipulation.
- Cavity flow shows several approaches to incompressible flow
- Shock tube shows how to do upwinding

10.1 Elastic Deformation

10.2 Flow in a lid-driven cavity

Incompressible viscous flow in a lid-driven cavity has been studied extensively and is used as a standard benchmark for computational fluid dynamics algorithms and codes. See the review article by Shankar and Deshpande[**?**] for recent references on computational and experimental work on this problem.

The configuration is sketched in Figure 10.1: fluid in a two-dimensional box is driven by a uniformly moving lid. The governing equations are the incompressible Navier-Stokes equations

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \frac{1}{\mathrm{Re}}\nabla^2 \mathbf{u} \tag{10.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{10.2}$$

where $\mathbf{u}$ is the fluid velocity, p is the pressure, and Re is the Reynolds number. Boundary conditions are no-slip, and are as indicated in the figure. Since the lid is moving the fluid in contact with the lid moves with the same velocity as the lid. The velocity on the bottom and sides is zero. There are no boundary conditions for pressure. This leaves the system indeterminate, since equation 10.2 unchanged by adding a constant to the pressure, but it is consistent and the indeterminacy will cause no problem for iterative solvers.

Here we will consider only the steady-state case. One can find the steady-state solution in a number of ways: by marching the time-dependent Navier-Stokes equations forward until equilibrium is reached, or by applying a nonlinear solver to the time-independent Navier-Stokes equations

$$(\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \frac{1}{\mathrm{Re}}\nabla^2 \mathbf{u} \tag{10.3}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{10.4}$$

In the examples that follow, we will show several approaches to solving the nonlinear steady-state problem: time-marching to equilibrium, Picard iteration on the steady-state equations, and Newton iteration on the steady-state equation.

There are a number of ways to rewrite the Navier-Stokes equations, and yet more ways to discretize them. There are many potential pitfalls in formulating a Navier-Stokes solver – the naive discretization methods are unstable, some methods will converge slowly without specialized preconditioners, and boundary conditions are tricky – and if you are new to the game you are advised to read and think first and code second (Gresho and Sani[**?**] is a good place to start).

With Sundance, you can easily develop solvers with different formulations, discretizations, and solution methods; in this section we present four examples of Navier-Stokes solvers:

- A mixed-order discretization of the primitive equations using Taylor-Hood elements. This discretization is stable, but gives rise to an indefinite saddle-point linear system that requires special preconditioning. In this example we do the nonlinear solve using Picard iteration.
- A pressure-stabilized equal-order discretization using bilinear elements for both velocity and pressure. Equal-order methods, in which the velocity and pressure are discretized with the same basis functions, are never stable with the unmodified Navier-Stokes equations. The method is stabilized by adding a regularization term to the Navier-Stokes equation. In this example we do the nonlinear solve using Newton's method with backtracking.
- A streamfunction-vorticity formulation of the Navier-Stokes equations, in which we rewrite the equations in such a way that the incompressibility constraint is automatically satisfied and the pressure drops out. This leads to a very elegant and well-behaved formulation; unfortunately, writing boundary conditions in three dimensions is difficult. In this example we converge to equilibrium by marching the time-dependent equations.

10.2.1 Mixed-order discretization of primitive formulation

Naively, one would proceed to discretize the Navier-Stokes equations by picking a basis and using it to represent both components of velocity and the pressure in a Galerkin method. Alas, life is not so simple; the mathematical theory behind the Navier-Stokes equations places strong constraints on the bases that are allowed for velocity and pressure; the two bases cannot be chosen independently of one another. Choose poorly, and spurious oscillations in pressure appear; choose poorly another way, and all your displacement "lock" to zero. The condition that must be satisfied for stability is called the Ladyshenskaya-Babuska-Brezzi (LBB) condition, and several generations of applied mathematicians and engineers have worked on finding basis combinations ("elements") that satisfy the strict demands of LBB. A summary of popular element types for Navier-Stokes and their status with respect to LBB are given in Gresho and Sani[**?**]. The simplest LBB-stable element to code in Sundance is the trianglular element P2-P1 element, also known as the Taylor-Hood element: 2nd-order Lagrange interpolation for velocity, 1st-order Lagrange interpolation for pressure.

Now that we have a stable element, life is good, right? Not so fast. The linearized Navier-Stokes equations as formulated discretize to a **saddle-point** system

$$\begin{bmatrix} B & C \\ C^T & 0 \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}. \tag{10.5}$$

Such a system requires care to solve. We use a preconditioner developed for this system by Wathen, Silvester, and Elman.

10.2.2 Pressure-stabilized equal-order discretization of primitive formulation

The Taylor-Hood discretization in the previous example required mixed-order interpolation and a specialized proeconditioner. Is there hope for solving the Navier-Stokes equations with equal-order interpolation, or better yet without a specialized preconditioner? The answer is "yes" to both questions, but we will have to add an unphysical regularization term to the incompressibility equation. We parameterize this term by a scaling factor β; if we choose β too large, our model is unphysical; if we choose β too small, we lose its stabilizing effect. So we gain simplicity and stability, but at the price of introducing a unknown but important control parameter.

An unstabilized equal-order discretization of NS is susceptible to the checkerboard instability, in which the pressure oscillates wildly on the scale of one element. You might think to squelch that oscillation by smoothing the pressure. Modify the continuity equation as follows:

$$\nabla \cdot \mathbf{u} + \epsilon(x)\nabla^2 p = 0 \tag{10.6}$$

where ϵ is an as yet unspecified scaling function.

Not only have we gained stability, but the discrete form of our system is now

$$\begin{bmatrix} B & C \\ C^T & \beta S \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}. \tag{10.7}$$

where both B and S are positive-definite matrices. This system can be solved quite easily by standard solvers and preconditioners, and needs no special treatment.

10.2.3 Streamfunction-vorticity formulation

The streamfunction-vorticity formulation is In the streamfunction-vorticity formulation, we notice that any divergence-free velocity field can be written as the curl of another vector field called the **streamfunction**. In two dimensions, the streamfunction reduces to a scalar ψ, and we have

$$u_x = \frac{\partial \psi}{\partial y} \tag{10.8}$$

$$u_y = -\frac{\partial \psi}{\partial x} \tag{10.9}$$

which is easily shown to have zero divergence. Taking the curl of the velocity field, we find an equation relating the streamfunction and vorticity

$$\nabla^2 \psi = \omega \tag{10.10}$$

We next take the curl of the Navier-Stokes momentum equation to obtain

$$-\frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} + \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} = \frac{1}{\mathrm{Re}}\nabla^2 \omega \tag{10.11}$$

Notice that the pressure has dropped out of the problem, and we are left with two scalar equations for the two scalar unknowns ψ and ω.

Linearization

We will solve the nonlinear problem using Newton's method with backtracking (e.g., Kelley [?]) and a full linearization. For large values of the Reynolds number, the convergence rate of Newton's method on this problem proves to be quite sensitive to the initial guess. To obtain a good initial guess we will embed the Newton solver in a continuation method, in which we solve a sequence of problems starting at small Reynolds numbers.

Expanding to first order about a trial solution $[\psi_0, \omega_0]$, we have

$$\nabla^2 \left[\psi_0 + \delta\psi\right] = \omega_0 + \delta\omega \tag{10.12}$$

and

$$-\frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} + \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} = \frac{1}{\mathrm{Re}}\nabla^2 \omega \tag{10.13}$$

Boundary conditions

Boundary conditions in vorticity-streamfunction formulations are a subtle issue. In the driven cavity problem we have specified both velocity components at the boundary, which means we must specify both normal and tangential derivatives of ψ, or equivalently, *both* Neumann and Dirichlet boundary conditions for ψ. In particular, ψ is constant everywhere on the boundary, and has unit flux on the top and zero flux elsewhere. There are no boundary conditions for the vorticity ω.

Notice that the nature of the boundary conditions rules out the otherwise appealing strategy of solving equation 10.11 for ω given an estimate of ψ and subsequently using that ω in equation 10.10 to obtain an updated estimate for ψ. This won't work: treated as separate problems, there are too few boundary conditions on equation 10.11 and too many on equation 10.10. The equations must be considered as a single system; see Gresho and Sani [?] for elaboration on this point.

Once we have accepted that we have to solve equations 10.10 and 10.11 as a coupled system, the immediate followup question is how we specify these boundary conditions in Sundance. The obvious place to start is to impose the Neumann BCs on ψ in the usual manner of assigning a value to the normal derivative in the boundary integral. However, if we then try to apply the Dirichlet condition in the usual way,

```
EssentialBC dirichletBC(wholeBoundary, varPsi*psi);
```

we replace the weak equations for ψ on the boundary with these boundary conditions. The problem is that `varPsi` on the boundary identifies exactly those same rows on which we just applied the Neumann condition. The resolution is simple: apply the Dirichlet BC in the rows corresponding to `varOmega` on the boundary, which is accomplished very simply by

```
EssentialBC dirichletBC(wholeBoundary, varOmega*psi);
```

Recall that this will replace exactly those rows including the boundary integral over $\nabla\omega\cdot\hat{n}$, so that we need not include that term.

Code

10.3 A source inversion problem

Most of the PDEs presented in this guide are **forward problems**: given a PDE and associated BCs, we solve for an unknown field variable. However, in some contexts the parameters of the PDE or BC are unknown and are to be determined by minimizing some objective function. In this case, we have an **inverse problem**, which we may broadly classify into parameter estimation problems in which parameters are determined by fitting a model to experimental data, and optimal control or design problems where parameters are chosen to satisfy some desired design criteria.

In this example, we will set up an inverse problem to determine the source of a pollutant that is being transported in a known flow field. In a more realistic problem, the flow field might also be determined as part of the inverse problem. We begin with the forward problem: let the contaminant be released with source density p and transported via advection and diffusion in velocity field $\mathbf{u}$. The steady-state concentration of the contaminant is given by the forced advection-diffusion equation

$$\mathrm{Pe}\,(\mathbf{u}\cdot\nabla)\,c = \nabla^2 c + p \tag{10.14}$$

We can synthesis some "experimental" data by solving the forward problem with a known source and taking measurements at N_s sensor locations.

Choose p to minimize the objective function

$$f(p) = \frac{1}{2}\sum_{i=0}^{N_s}\left(c(p) - c_i\right)^2. \tag{10.15}$$

Unfortunately, this formulation is ill-posed because there will be an unknown p at every grid point, but only N_s measurements. We can deal with this problem by regularizing the source field, for instance by augmenting the objective function with a term that penalizes large gradients in the source

$$f(p) = \frac{1}{2}\sum_{i=0}^{N_s}\left(c(p) - c_i\right)^2 + \frac{1}{2}|\nabla p|^2 \tag{10.16}$$

Now that we have a well-posed problem, how do we proceed? One simple and very widely-used approach is to compute the objective function from the formula above, which requires we solve the forward problem for each choice of parameter p.

When the space of source parameters is very large, these approaches are not feasible.

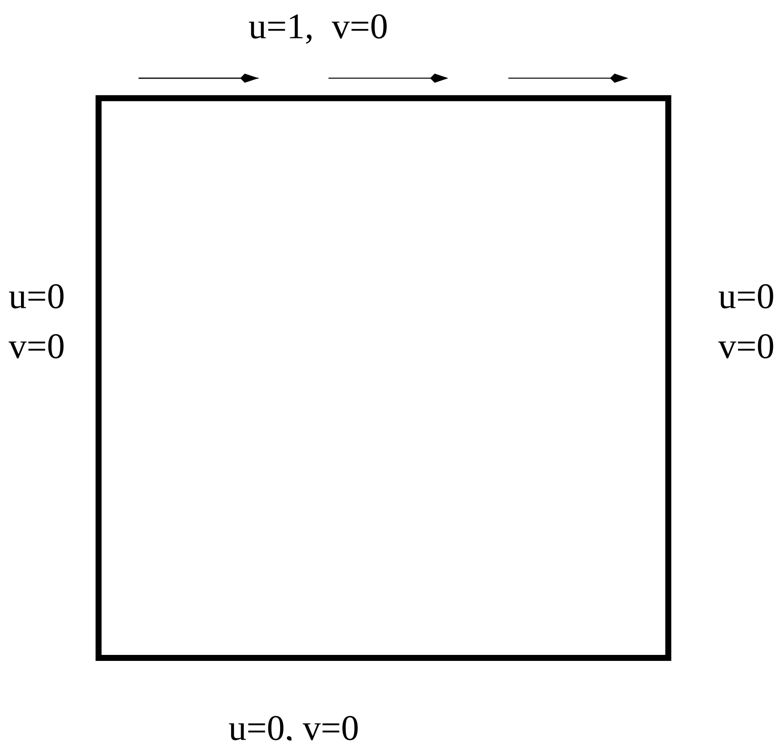

Figure 10.1: Schematic of lid-driven cavity flow problem. Boundary conditions on velocity are indicated with text labels.

Appendix A

Listing of user-level classes and methods

This chapter contains detailed information about user-level classes and methods. This is intended as a reference for Sundance users rather than Sundance developers; Sundance developers should consult the Doxygen-generated class documentation in which all classes and methods are described.

For each class, we describe

- Copy and assignment behavior.
- User-level subclasses.
- User-level member methods. Where appropriate, we will subdivide this list into groups of related methods.
- User-level global methods (if any).

A.1 Symbolic objects

A.1.1 Expr

Class `Expr` is the user-level handle for all objects that can be used in symbolic operations, e.g. functions, constants, and differential operators. See Chapter 2 for a guide to how symbolic mathematics is used in Sundance.

Constructors

Constructor	Comments
`Expr()`	Creates an empty expression
`Expr(const double& value)`	Creates an expression with a constant real value. Once constructed, the value cannot be changed, so this ctor is not appropriate for changeable constant parameters such as design variables or timesteps. In that case, use a `ParameterExpr` instead.
`Expr(ExprBase* subclass)`	Creates an expression by capturing a subclass pointer into a handle.

Copy and assignment behavior

Copy behavior is shallow. The case in which an object appears on both sides of an assignment operator, e.g.,

$$\texttt{x} = \texttt{x} + \texttt{y}; \tag{A.1}$$

is handled specially. See subsection 2.3.1 for more information.

Expression subclasses

The user-level expression subtypes are listed below.

Constructor	Comments
`CoordExpr(int d, const string& name)`	function evaluating to the d-th coordinate
`Derivative(int d, int order=1)`	Partial differentiation operator of order=*order* in direction d.
`TestFunction(const BasisFamily& basis, const string& name)`	Test function with the specified basis
`UnknownFunction(const BasisFamily& basis, const string& name)`	Unknown function with the specified basis
`DiscreteFunction(const TSFVectorSpace& space, const Expr& expr, const string& name)`	Function discretizing the value of *expr* onto the given discrete space
`ParameterExpr(const double& value)`	Changeable parameter. This will be treated as a constant over the mesh, but can be changed between steps in a computation.

Overloaded math operators (methods)

The following math operations are defined between expressions. In the case of binary operators, both arguments must have a compatible list structures as described in the comments below.

Return type	Method	Comments
`Expr`	`operator-() const`	Unary minus, returns additive inverse
`const Expr&`	`operator+=(const Expr& other)`	Reflexive addition. Both operands must have identical list structures.
`const Expr&`	`operator-=(const Expr& other)`	Reflexive subtraction. Both operands must have identical list structures.
`const Expr&`	`operator*=(const Expr& other)`	Reflexive multiplication. The meaning of this operation depends on the list structure of `this` and `other`. If either `this` or `other` is scalar-valued, the product means multiplication by a scalar. The multiplication operator is also meaningful when the two operands have a list structure such that multiplication can be interpreted as a tensor contraction operation (generalized dot product).
`const Expr&`	`operator/=(const Expr& other)`	Reflexive division. The right operand must be a scalar.

Overloaded math operators (global methods)

See the comments in the previous section A.1.1 for restrictions on the list structure of operands to binary operators.

Return type	Method	Comments
`Expr`	`operator+(const Expr& a, const Expr& b)`	Addition
`Expr`	`operator-(const Expr& a, const Expr& b)`	Subtraction
`Expr`	`operator*(const Expr& a, const Expr& b)`	Multiplication
`Expr`	`operator/(const Expr& a, const Expr& b)`	Division
`Expr`	`pow(const Expr& a, const double& p)`	Raise to scalar power
`Expr`	`sqrt(const Expr& a)`	Square root. Argument must be a positive scalar.
`Expr`	`exp(const Expr& a)`	Exponential. Argument must be a scalar.
`Expr`	`log(const Expr& a)`	Natural logarithm. Argument must be a strictly positive scalar.
`Expr`	`sin(const Expr& a)`	Trigonometric sine . Argument must be a scalar.
`Expr`	`cos(const Expr& a)`	Trigonometric cosine. Argument must be a scalar.
`Expr`	`tan(const Expr& a)`	Trigonometric tangent. Argument must be a scalar.
`Expr`	`sinh(const Expr& a)`	Hyperbolic sine. Argument must be a scalar.
`Expr`	`cosh(const Expr& a)`	Hyperbolic cosine. Argument must be a scalar.
`Expr`	`tanh(const Expr& a)`	Hyperbolic tangent. Argument must be a scalar.
`Expr`	`fabs(const Expr& a)`	Absolute value. Argument must be a scalar.

Probing values at a cell or point

Return type	Method	Comments
`ExprValue`	`average(const Cell& cell) const`	Returns average value over a cell

Definite integrals and norms

Return type	Method	Comments
`double`	`norm() const`	Returns L^2 norm
`double`	`norm(int p) const`	Returns L^p norm
`double`	`maxNorm() const`	Returns L^∞ norm
`double`	`norm(const CellSet& region) const`	Returns L^2 norm computed over a subdomain defined by the cell set *region*
`double`	`norm(int p, const CellSet& region) const`	Returns L^p norm computed over a subdomain defined by the cell set *region*
`double`	`maxNorm(const CellSet& region) const`	Returns L^∞ norm computed over a subdomain defined by the cell set *region*

Listing

Return type	Method	Comments
`const Expr&`	`operator[](int i) const`	Returns read-only reference to i-th element
`Expr&`	`operator[](int i)`	Returns writeable reference to i-th element
`int`	`length() const`	Returns number of entries in the current level of list
`int`	`size() const`	Returns total number of entries in list
`Expr`	`append(const Expr& other)`	Puts the argument expr at the end of the list.

A.1.2 BasisFamily

Class `BasisFamily` is used to specify the basis function used in defining a discrete, unknown, or test function. Currently, the only subtype implemented is `Lagrange` which represents the family of Lagrange interpolants. There are few user-level methods; at the user level this class' role is as a specifier of basis type, and it appears only as an argument to constructors of functions and vector spaces.

Copy and assignment behavior

Copies and assignments are shallow.

BasisFamily subclasses

Constructor	Comments
`Lagrange(int order)`	Lagrange interpolation with polynomial order=*order*.

User-level methods

Return type	Method	Comments
`XMLObject`	toXML() const	write XML representation

A.1.3 QuadratureFamily

Class `QuadratureFamily` is used to specify the family of quadrature rule used in integrating a weak form or doing a definite integral or norm of an expression. Currently, the only subtype implemented is `GaussianQuadrature` which represents the family of Gaussian quadrature rules. There are few user-level methods; at the user level this class' role is as a specifier of quadrature type, and it will appears only as an argument to weak form and boundary condition constructors and integral and norm functions.

Copy and assignment behavior

Copies and assignments are shallow.

QuadratureFamily subclasses

Constructor	Comments
`GaussianQuadrature(int order)`	Gaussian quadrature of order=*order*.

User-level methods

Return type	Method	Comments
`XMLObject`	toXML() const	write XML representation

A.2 Geometric objects

A.2.1 Mesh

Class `Mesh` is Sundance's unstructured mesh object. In most Sundance problems your interaction with the `Mesh` will be very limited: you will generally read it using a `MeshReader`'s `getMesh()` method, and then pass it as an argument to functions that need it. You'll rarely need to call `Mesh` methods.

Copy and assignment behavior

Copy behavior is shallow.

Subclasses

`Mesh` has no subclasses.

Constructors

Constructor	Comments
`Mesh()`	Creates empty mesh
`Mesh(int dim)`	Creates empty mesh of spatial dimension dim

User-level methods

Return type	Method	Comments
int	spatialDim() const	returns the spatial dimension of the mesh
int	numPoints() const	returns the number of points in the mesh
int	numCells(int d) const	returns the number of cells of dimension *d* in the mesh
void	printCells() const	Writes a whole bunch of information about the mesh to standard error. This is sometimes useful for very low-level debugging.

Methods for creating a mesh

These methods are necessary only if you are generating a mesh by hand or writing your own `MeshReader` subclass.

Return type	Method	Comments
int	addPoint(const Point& point, int ownerProcID=0, const Array<int>& sharerProcID, int globalIndex=-1)	Adds a point to the mesh. Optional arguments specify the processor that owns the point as well as a list of processors that need the point.
Cell	createMaximalCell(const CellFactory& cellFactory, const string& label, int ownerProcID=0, int globalIndex=-1)	Adds a maximal cell to the mesh. Any facet cells are created during the course of this operation.

A.2.2 MeshReader

Copy and assignment behavior

Copy behavior is shallow.

User-level methods

Return type	Method	Comments
`Mesh`	`getMesh() const`	Returns a mesh

MeshReader subclasses

Constructor	Comments
`ShewchukMeshReader(const string& filename)`	Reader for Shewchuk's Triangle format[**?**]. Expects to find the mesh in the files *filename*`.poly` and *filename*`.ele`
`ExodusMeshReader(const string& filename)`	

A.2.3 MeshGenerator

Copy and assignment behavior

Copy behavior is shallow.

User-level methods

Return type	Method	Comments
`Mesh`	`getMesh() const`	Creates and returns a mesh

MeshGenerator subclasses

Constructor	Comments
`LineMesher(double a, double b, int n)`	Creates and returns a uniform mesh with n elements on the line segment $[a, b]$.
`RectangleMesher(double ax, double bx, int nx, double ay, double by, int ny)`	Creates and returns a uniform triangular mesh with n_x by n_y elements on the rectangle $[a_x, b_x] \otimes [a_y, b_y]$.
`RectangleQuadMesher(double ax, double bx, int nx, double ay, double by, int ny)`	Creates and returns a uniform quadrilateral mesh with n_x by n_y elements on the rectangle $[a_x, b_x] \otimes [a_y, b_y]$.

A.2.4 CellSet

Class `CellSet` represents a set of cells, and is used to specify the cells on which an equation or boundary condition will be applied or on which an integral or norm will be computed. A `CellSet` object is not in itself a list of cells, rather it is an abstract rule that can be used to identify the desired cells. For simplicity, however, we will usually talk of it as if it were a set rather than a rule used to identify a set.

Copy and assignment behavior

Copy behavior is shallow.

Cell set subclasses

Constructor	Comments
`MaximalCellSet()`	set of all cells with dimension equal to D, the spatial dimension of the mesh
`BoundaryCellSet()`	set of all $D - 1$-dimensional cells on the boundary
`LabeledCellSet(const string& label)`	set of all cells having a given label

User-level methods

Return type	Method	Comments
`CellSet`	`operator+(const CellSet& other) const`	Union operator, returning a cell set that is the set union of *this* and *other*
`CellSet`	`operator&&(const CellSet& other) const`	Intersection operator, returning a cell set that is the set intersection of *this* and *other*
`CellSet`	`subset(const LogicalExpr& filter) const`	Return a new cell set defined by applying the given filter to this cell set.
`XMLObject`	`toXML() const`	Return an XML representation of this object

A.2.5 Cell

Copy and assignment behavior

Copy behavior is shallow.

Cell subclasses and constructors

There are no user-level cell subclasses or constructors. Cells are constructed only inside the `createCell()` method of `Mesh`.

User-level methods

Return type	Method	Comments
`int`	`numFacets(int dim) const`	Returns the number of facets of dimension `dim`
`const Cell&`	`facet(int dim, int index) const`	Returns the `index`-th facet of dimension `dim`
`int`	`numCofacets(int dim) const`	Returns the number of cofacets of dimension `dim`
`const Cell&`	`cofacet(int dim, int index) const`	Returns the `index`-th cofacet of dimension `dim`
`const Point&`	`point(int index) const`	Returns the `index`-th node