Sundance user's manual

Various contributors

June 15, 2010

Contents

Chapter 1

Introduction

Sundance is a high-level system, in which a finite-element problem is described with expressions, function spaces, and domains instead of low-level concepts such as matrix entries, elements, and nodes. If you find yourself asking things such as "how can I modify the entries in my local stiffness matrix" instead of "how can I modify my symbolic equation set," you're probably thinking about the problem the wrong way. Of course, you can write Sundance code using its low-level features directly, but such code will be harder to read and almost always less efficient. So please stick to the higher-level objects and operations. Matlab programmers who have learned to write their problems as high-level vector and matrix operations instead of low-level loops will find this way of thinking natural.

Solution of partial differential equations is a complicated endeavor with many subtle difficulties, and there can be no one-size-fits-all simulation code. Sundance is not a simulation code as much as it is a set of high-level objects that will let you, the user, build your own simulation code with a minimum of effort. These objects shield you from the rather tedious bookkeeping details required in writing a finite-element code, but they do not shield you from the need to understand how to do a proper formulation and discretization of a given problem.

1.1 About the code and the documentation

Sundance is written in the C++ programming language, with some calls to third-party codes written in C and Fortran. A user of Sundance will need to know the rudiments of C++ and should know the fundamentals of object-oriented programming, but need not be an expert C++ designer. You will need to know how to use and occasionally write classes, but not how to design them.

Only a fraction of the objects and methods that make up Sundance are ever needed in user code; most are used internally by Sundance. This user's guide concentrates on those objects and methods needed by you to write high-level code to solve a PDE using Sundance's native capabilities; if your interest is in modifying or extending Sundance or simply figuring out what goes on "under the hood," I'll refer you to the sparse and expert-friendly Doxygen documentation.

Sundance objects are declared in the Sundance namespace.

1.1.1 Typographical conventions

We show code samples, variable names, function names, and class names in `typewriter` font.

Class names begin with capital letters, and each word within the name also begins capitalized. For example: `MeshReader, DiscreteFunction`. Function names and variables begin with lower-case letters, but subsequent words within the name are capitalized. For example: `getCells()` or `numCells`. Data member names end with an underscore, for example: `myName_`.

1.2 Fundamental data structures & utilities

1.2.1 Classes and objects

Sundance is essentially a suite of *classes* and functions operating on them. A class is a data structure containing *member data* and *methods* or *member functions.* Refer to your favorite C++ book for various simple examples.

1.2.1.1 Subclasses

Polymorphism is a buzzword meaning the representation of different object types (derived classes, or subclasses) having common behavior through a common interface (the base class). For example, there are a number of file formats in which we might want to write results. In a 1D problem, a simple text file with columns of values is sufficient for plotting in a program such as Matlab. In 2D and 3D, more complex data structures are needed; numerous file formats for storing such data have been developed; two we'll use here are Exodus (a binary format for 2D/3D data) and VTK (an XML-based text format for 2D/3D data). We might want to use any one of these, but don't want switching file format to require major modifications to our simulation code. The solution to this design problem is to encapsulate the common behavior of file writers in an *abstract interface*, or *abstract class.* Specific file writers are then implemented as *subclasses* or *derived classes* of the abstract class.

In the file writer example, the abstract class is called FieldWriterBase. Several of its subclasses are VTKWriter, ExodusWriter, and MatlabWriter. A quirk of writing clean C++ code is that we "hide" the FieldWriterBase object behind a "handle" class called FieldWriter. We talk more about handles below; for now it is sufficient to say that the handle class make the code more readable by humans. Here's an example in which we create an ExodusWriter subclass, capture it into a FieldWriter handle, and then execute several operations through the FieldWriter interface.

```
/* Create a field writer that will write to filename using the Exodus format */
FieldWriter writer = new ExodusWriter(filename);

/* Prepare to write by adding data to the writer */
writer.addMesh(mesh);
writer.addField(temperature, /* some arguments */);
writer.addField(density, /* some arguments */);

/* Do the write operation */
writer.write();
```

To convert this code to write to a VTK file, all that's needed is to change the writer subclass used, replacing

```
FieldWriter writer = new ExodusWriter(filename);
```

with

```
FieldWriter writer = new VTKWriter(filename);
```

The remaining code is unchanged because the adding and writing operations are done through the common FieldWriter interface.

1.2.1.2 Handle classes

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[1] 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 generally 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.

1.3 Parallel computing

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, Trilinos and Sundance must be built with MPI enabled, and then your simulator must use a parallel-capable linear algebra representation such as Epetra. 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 parallel linear algebra 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.

[1]A *pointer* is a variable containing the memory address of some data, in other words, it "points to" the data. See your favorite C++ book for more information on pointers. The Sundance toolkit is designed to minimize your need to work directly with pointers.

Chapter 2

Some preliminaries

Herein are some dull but useful code management topics collected for reference. You can skim over this chapter for now, then return to it as you encounter output, error handling, and command-line processing in the examples.

2.1 Structure of a user's program

Your main program will usually look like this:

```
#include "Sundance.hpp"

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

    /* body of code goes here */

  }
  catch(std::exception& ex)
  {
    Sundance::handleException(e);
  }
  Sundance::finalize();
}
```

2.2 Some utilities

2.2.1 Output utilities

While it's possible to use the C++ standard output streams `cout` and `cerr` directly, the output can get cluttered when computing on multiple processors because it's difficult to tell which message comes from which processor. The `Out` class has several methods that can help keep your screen output clean and readable.

- `Out::os()` is a wrapper around cout that prepends a processor identifier to each line
- `Out::root()` is a wrapper around cout that is active only on the root (rank zero) processor. Messages to `Out::root()` on non-root processors are ignored.

To see the difference, try running the code

```
Out::root() << "Processor␣roll␣call:" << endl;
Out::os() << "rank=" << MPIComm::world().getRank() << "␣says␣hello" << endl;
```

with several processors.

2.2.2 Error checks

Use the TEST_FOR_EXCEPTION() macro, which takes three arguments:

1. An expression that evaluates to a boolean. If the expression evaluates to true, the exception will be thrown.
2. The type of exception class to be thrown. The RuntimeError class is a good general-purpose exception.
3. Code to write a descriptive error message. This code can use any `ostream` operations to format the error message can be formatted nicely.

Here's an example of using TEST_FOR_EXCEPTION() to check validity of the input to a function with a restricted domain.

```
double myFunction(const double& x)
{
  TEST_FOR_EXCEPTION( x<0.0, RuntimeError,
    "input␣to␣myFunction()␣must␣be␣positive:␣value␣was␣x=" << x);

  return sin(sqrt(x));
}
```

In addition to neatly packaging several lines of error checking code, the TEST_FOR_EXCEPTION() macro does two useful things for you:

1. It appends the filename and line number where the error occurred.
2. It calls a dummy function, TestForException_break(), intended as a place to put a breakpoint in interactive debugging. For example, when starting up gdb you can give the command

   ```
   break TestForException_break
   ```

   Then, when an error occurs the debugger will stop the breakpoint, letting you trace back to the point of failure. See A.2 for more information on configuring debuggers to work with Sundance.

2.2.3 Working with command-line arguments

Command-line arguments to main() are processed by the Teuchos `CommandLineProcessor` utility class (hereafter, CLP). The CLP class parses command-line arguments set with the Posix standard double-dash format. Parsing is done inside the `Sundance::init()` function, so all command-line setup should be done before `init()` is called. Here is an example of setting several options of different data types. Note that the syntax for `bool` options differs slightly from that for `string`, `int`, and `double`.

```
int main(int argc, char** argv)
{
  try
  {
    /* Set default values for the options */
    int nSteps = 100;
    double reynolds = 500.0;
    string outFilename = "flow";
```

```
    bool useSUPG = false;

    /* Register the options with the command-line parser */
    Sundance::setOption("steps", nSteps, "Number␣of␣timesteps");
    Sundance::setOption("Re", reynolds, "Reynolds␣number");
    Sundance::setOption("o", outFilename, "Filename␣for␣output");
    Sundance::setOption("supg", "no-supg", useSUPG,
      "Whether␣to␣use␣SUPG␣stabilization␣(recommended␣at␣high␣"
      "Reynolds␣numbers)");

    /* Call init as usual */
    Sundance::init(&argc, &argv);

    /* ----- remainder of code --- */
```

If you run the executable with flags set as shown,

```
./myCode.exe --o="flow-2000.0" --supg --Re=2000.0
```

then *after* the call to `Sundance::init()` the variable `outFilename` will be set to “`flow-2000.0`”, the variable `reynolds` will be set to 2000, and `useSUPG` will be set to true. The option `steps` was never used, so the variable `nSteps` will be left at its default value of 100. *Before* the call to Sundance::init() the command line has not been parsed so the variables still contain their default values.

The example above used high-level wrapper functions, `Sundance::setOption()`, for the lower-level functions of CLP. Should you prefer to work directly with the CLP object, it can be accessed through the static function `Sundance::clp()`.

2.2.4 Working with XML files

An alternative to command line arguments is specification of options in an XML file.

Chapter 3

Operators, vectors, and solvers

3.1 Overview

Most of the computation in PDE simulation involves operations with matrices and vectors. If you've programmed in Matlab, you'll have learned that it's a good idea – for both efficiency and human readability – to work with matrices and vectors as "objects" rather than doing operations by looping over indices. There are no built-in matrix and vector types in C++, but one can write classes to represent matrices and vectors.

It's important to understand that we will be forming systems of linear equations that are both very large and very sparse. The data structures for the matrices and vectors involved are fairly complicated, and the best choice of solution algorithm will depend strongly on the specific problem. There are a number of subpackages within Trilinos for doing sparse data structures and sparse solves. These in turn depend on lower-level libraries for dense linear algebra (LAPACK and BLAS) and for parallel communication (MPI). To provide a consistent and convenient user interface we "wrap" those capabilities in a suite of higher-level objects. This three-layer structure is shown in figure 3.1. Most of the time you'll work with the highest-level objects; occasionally you might need to delve into the Trilinos mid-level objects if you want some customized behavior. Should you need to work with the mid-level objects, all of the Trilinos libraries have Doxygen documentation available.

3.1.1 Principal user-level objects

- The `VectorType` class is responsible for creating `VectorSpace` objects of a specified size and distribution over processors. For example, the `EpetraVectorType` subclass specifies that vectors will be stored as Epetra data structures. Note that "vector type" is not a mathematical object in the sense that vector spaces and vectors are; rather, it is a concept that lets us specify what *low-level software implementation* of vectors, spaces, and operators will be used.
- The `VectorSpace` class is responsible for creating vectors. This is done using the `createMember()` function. `VectorSpace` and `VectorType` are both examples of the *abstract factory* design pattern; an abstract factory is a software design trick that provides a common interface for creating objects, specific implementations of which might be constructed in very different ways.
- The `Vector` class represents vectors. Two compatible (*i.e.*, both from the same vector space) vectors can be added and subtracted with the $+$ and $-$ operators. The $*$ operator between two vectors does the dot product. Other operations such as various norms, Hadamard products, various operations involving scalars, and element access are described in section 3.2.2.
- The `LinearOperator` class represents linear operators. However, many operators can be implemented "matrix free," meaning it is not necessary to store any matrix elements. For example, while the identity operator has a matrix representation, it is inefficient to go through a matrix-vector multiplication when the assigment $x \leftarrow Iy$ can be effected by simply copying y into x. Thus, an identity operator isn't implemented as a matrix, it's simply an instruction to copy the input directly into the output.

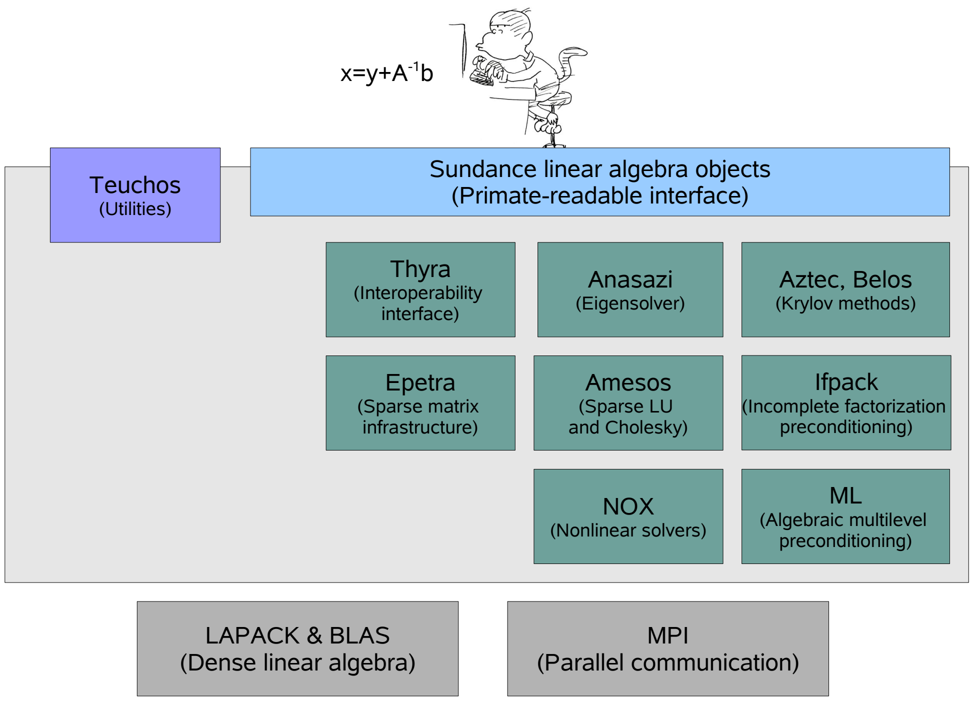

Figure 3.1: Schematic of relationship between high-level Sundance linear algebra interface, mid-level Trilinos libraries for sparse linear algebra, and low-level BLAS, LAPACK, and MPI. The Sundance linear algebra objects make it possible to write efficient code using high-level notation such as $x = y + A^{-1}b$.

- The `LinearSolver` class represents algorithms for solving linear equations.

All of these classes are templated on the scalar type, for instance, `Vector<double>` or `Vector<float>`. The Sundance PDE discretization capabilities are presently hardwired to double-precision real numbers, so we'll use `Vector<double>` in all examples.

3.1.1.1 Handles, deep and shallow copies

The five principal classes, VectorType, VectorSpace, Vector, LinearOperator, and LinearSolver are all implemented as *reference-counted handles.* An important consequence of that fact is that copies are "shallow". That means that an assignment such as

```
Vector<double> x = someSpace.createMember();
Vector<double> y = x;
```

does not create a new copy of the vector x, complete with new data. Rather, it creates a new "handle" to the same data. One advantage of this is obvious: vectors can be large, so we want to avoid making unnecessary copies. But note that any modification to y will also trigger the same modification to x, because x and y are referring to *exactly the same data in memory.* The potential for confusion and unintended side effects is obvious. Less obvious is that in certain important circumstances, such side effects are exactly what is needed for a clean user interface to efficient low-level code.

Should you want to make a "deep" copy of a vector, in which the copy has its own data and is fully independent of the original, use the copy() member function.

```
Vector<double> x = someSpace.createMember();
x.setToConstant(1.0);
Vector<double> y = x;
Vector<double> z = x.copy();
```

At this point x and y are two "handles" to the same underlying vector, whereas z is a vector that has, for now, the same elements as x. Now modify x

```
x.setToConstant(5.0);
```

and print norms of the three vectors

```
Out::root() << "||x||␣=␣" << x.norm2() << endl;
Out::root() << "||y||␣=␣" << y.norm2() << endl;
Out::root() << "||z||␣=␣" << z.norm2() << endl;
```

The norms of x and y are consistent with the updated value even though the variable y has never been *directly* modified. The norm of z remains consistent with the original value of the vector x, because modifications to x are not propagated to its deep copies.

3.1.1.2 Example: A conjugate gradient solver

Here we show how these objects are used to write a simple conjugate gradient algorithm and apply it to finite-difference solution of the Poisson equation in 1D. The creation of the finite-difference matrix is assumed to be done by a function `buildFDPoisson1D()`. We don't show the details of the matrix creation function here; filling matrices is low-level code that Sundance will usually hide from you.

```
int numPerProc = 20;

VectorType<double> vecType = new EpetraVectorType();

LinearOperator<double> A = buildFDPoisson1D(vecType, numPerProc);

Out::root() << "Matrix␣A␣=␣" << endl; // print header on root processor only
Out::os() << A << endl;               // print matrix data on all processors
```

Having created the matrix, we now make a vector of compatible size and fill it with values. We'll use this as the RHS of $Ax = b$.

```
VectorSpace<double> space = A.domain();
Vector<double> b = space.createMember();

b.setToConstant(1.0);

Out::root() << "Vector␣b␣=␣" << endl; // print header on root processor only
Out::os() << b << endl;                // print matrix data on all processors
```

Now we write the CG algorithm. For simplicity, error checking is omitted.

```
Vector<double> x = b.copy(); // NOT x=b, which would be a shallow copy
Vector<double> r = b - A*x;
Vector<double> p = r.copy();

double tol = 1.0e-12;
int maxIter = 100;

Out::root() << "Running␣CG" << endl;
Out::root() << "tolerance␣=␣" << tol << endl;
Out::root() << "max␣iters␣=␣" << maxIter << endl;
Out::root() << "-------------------------------------------------------" << endl;
for (int i=0; i<maxIter; i++)
{
  Vector<double> Ap = A*p; // save this, because we'll use it twice
  double rSqOld = r*r;
  double pAp = p*Ap;
  double alpha = rSqOld/pAp;

  x = x + alpha*p;
  r = r - alpha*Ap;

  double rSq = r*r;
  double rNorm = sqrt(rSq);
  Out::root() << "iter=" << setw(6) << i << setw(20) << rNorm << endl;

  if (rNorm < tol) break;

  double beta = rSq/rSqOld;
  p = r + beta*p;
}
```

Upon exiting the CG loop, print the solution.

```
Out::root() << "Solution:␣" << endl;
Out::os() << x << endl;
```

An industrial-strength CG solver would need preconditioning and error checking, and could be packaged up as a `LinearSolver` object.

3.1.1.3 Example: Inverse power iteration

Next we show code for calculation of the lowest eigenvalue of a matrix A by applying the power method to A^{-1}. The most important feature to look for in this example is the use of an implicit inverse operator. It's rarely a good idea to compute the matrix A^{-1}, so we want to avoid that. Instead, we create an *implicit inverse operator* that evaluates $A^{-1}y$ by using a LinearSolver object to solve the system $Ax = y$. Linear solvers are complicated enough that we'll usually build them by reading parameters from a file.

```
#include "Sundance.hpp"
#include "FDMatrixPoisson1D.hpp"

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

    int numPerProc = 1000;
    VectorType<double> vecType = new EpetraVectorType();
    LinearOperator<double> A = buildFDPoisson1D(vecType, numPerProc);
    VectorSpace<double> space = A.domain();

    Vector<double> x = space.createMember();
    x.setToConstant(1.0);

    LinearSolver<double> solver     =
      LinearSolverBuilder::createSolver("amesos.xml");
    LinearOperator<double> AInv = inverse(A, solver);

    int maxIters = 100;
    double tol = 1.0e-12;
      double mu;
    double muPrev = 0.0;

    for (int i=0; i<maxIters; i++)
    {
      Vector<double> AInvX = AInv*x;
      mu = (x*AInvX)/(x*x);
      Out::os() << "Iter␣" << setw(5) << i
        << setw(25) << setprecision(10) << mu << endl;
      if (fabs(mu-muPrev) < tol) break;
      muPrev = mu;
      double AInvXNorm = AInvX.norm2();
      x = 1.0/AInvXNorm * AInvX;
    }
    Out::root() << "Lowest␣eigenvalue␣"
      << setw(25) << setprecision(10) << 1.0/mu << endl;
  }
  catch(exception& e)
  {
    Sundance::handleException(e);
  }
  Sundance::finalize();
}
```

Both examples have concentrated on the *use* of matrix and vector objects rather than the *creation* of these objects. Sundance will automatically build matrices for rather complicated problems, and you'll rarely need to create matrices yourself. However, in writing advanced preconditioning, optimization, and solver algorithms you'll need to compose implicit operations.

3.2 Vectors

We now move on from the high-level overview to a discussion of the types and capabilities of vectors.

3.2.1 Creation of vectors

You will rarely call a vector constructor directly; the reason for this is that different vector libraries have different data requirements making it difficult to . Instead, vectors are built indirectly by calling the `createMember()` member function of `VectorSpace`. Each `VectorSpace` object contains the data needed to build vector objects, and the implementation of the `createMember()` function will use that data to invoke a constructor call.

3.2.2 Vector operations

3.2.2.1 Overloaded binary operations

The standard binary operations $\mathbf{a} \pm \mathbf{b}$, $\alpha\mathbf{a}$, and $\mathbf{a} \cdot \mathbf{b}$ are implemented via operator overloading.

Left operand	Operator	Right operand	Return type	Restrictions
`a`	`+`	`b`	`Vector`	Operands must be members of the same vector space
`a`	`-`	`b`	`Vector`	Operands must be members of the same vector space
`alpha`	`*`	`a`	`Vector`	
`a`	`*`	`b`	`Scalar`	Operands must be members of the same vector space

These operations can be combined in any way that makes mathematical sense, for example:

```
Vector<double> v = 2.0*a - 4.0*b + (a*b)*c;
```

Note that the operation `(a*b)*c` obeys the rules of precedence, so that the vector **c** is multiplied by the scalar $\mathbf{a} \cdot \mathbf{b}$.

3.2.2.2 Unary vector-valued operations

The following functions operate elementwise on a vector, returning a new vector as a result. The original vector is unchanged.

Operation	Notes
`reciprocal()`	If any element is zero, a runtime exception will be thrown
`abs()`	
`copy()`	Makes a “deep” copy of a vector

```
Vector<double> w = v.reciprocal();
```

3.2.2.3 Norms and other reduction operations

Operation	Notes
`norm1()`	Computes $\|x\|_1$
`norm2()`	Computes $\|x\|_2$
`normInf()`	Computes $\|x\|_\infty$
`max()`	Largest element of x
`min()`	Smallest element of x

If $x = \begin{pmatrix} -1 & 0 & 0 & 1 & 2 \end{pmatrix}$ then

```
Out::os() << x.norm1() << endl;   // prints 4
Out::os() << x.norm2() << endl;   // prints sqrt(6)
Out::os() << x.normInf() << endl; // prints 2
Out::os() << x.max()     << endl; // prints 2
Out::os() << x.min()     << endl; // prints -1
```

Also, there are methods to return the location of the minimum and maximum (used in algorithms for constrained optimization, to find the nearest constraint). Write these up later.

3.2.3 Block vectors

To be written

3.2.4 Access to vector elements

To be written

3.2.5 Writing your own vector operations

Need a clean interface to Bartlett's RTOp system.

3.3 Linear operators

Linear operators are represented by the `LinearOperator` class.

3.3.1 Matrices

Construction of matrices is a several-step process.

1. Create a `MatrixFactory` object
2. Call member functions of the `MatrixFactory` to configure the sparsity structure of the matrix
3. Call the createMatrix() member function of the `MatrixFactory` to allocate the matrix
4. Call member functions of the `LoadableMatrix` interface to set values of the nonzero elements.

If this seems complicated, it's because working with sparse matrices *is* complicated, and different sparse matrix implementations (there are several just in Epetra) need to be constructed in different ways. The MatrixFactory interface lets us hide the complexity of sparse matrix construction behind a common interface.

3.3.2 Implicit operators

We will frequently need to build operators out of simpler operators, without explicitly forming matrices. Several common types of implicit operators are described here.

In the following examples it is assumed that operators A and B have been created by some function.

```
LinearOperator<double> A = makeSomeMatrix();
LinearOperator<double> B = makeSomeOtherMatrix();
```

3.3.2.1 Operator arithmetic

The action of a composed operator ABx is computed implicitly by first computing $y = Bx$, then computing Ay. There is no need to form the matrix AB. Similarly, $(A \pm B)\, x$ can be evaluated implicitly by computing $y = Ax$, $z = Bx$, then doing $y \pm z$. Action of a scaled operator αAx is done implicitly as $\alpha\,(Ax)$. Any combination of these can be specified using overloaded operators, for example:

```
LinearOperator<double> C = A + B;
LinearOperator<double> D = 2.0*A - 0.5*B + 1.2*C;
LinearOperator<double> E = A*B;
```

3.3.2.2 Transposition

Most good sparse matrix packages have the ability to compute $A^T x$ without explicitly forming A^T. Given that, together with implicit composition, we can do $(AB)^T x = B^T A^T x$ implicitly as well, and with implicit addition we can do $(A \pm B)^T x = A^T x \pm B^T x$. The `transposedOperator()` function creates an operator object that knows to apply these rules.

```
LinearOperator<double> At = transposedOperator(A);
```

3.3.2.3 Diagonal operators

A diagonal operator can be represented with nothing but a vector of diagonal elements. To make a diagonal operator, call the `diagonalOperator()` function with the vector to be put on the diagonal.

```
Vector<double> d = makeSomeVector();
LinearOperator<double> D = diagonalOperator(d);
```

3.3.2.4 Zero operators

Application of the zero operator returns the zero vector of the range space of the operator. To make a zero operator, call the `zeroOperator()` function with arguments that specify the domain and range spaces of the operator.

```
LinearOperator<double> zero = zeroOperator(domain, range);
```

3.3.2.5 Identity operators

The identity operator simply returns a copy of the operand.

```
LinearOperator<double> I = identityOperator(space);
```

3.3.2.6 Implicit inverses

The operation $y = A^{-1}x$ is computed implicity by solving the system $Ay = x$. It is necessary to specify the solver algorithm that will be used to solve the system.

```
LinearOperator<double> AInv = inverse(A, solver);
```

3.3.3 Writing your own operator type

Write your class to conform to the `SimplifiedLinearOpBase` abstract interface, which basically means writing a member function that applies the operator to a vector. *To be written.*

3.3.4 Block operators

To be written

3.3.5 Access to matrix data

You usually don't want to do this! Two exceptions are: row access for creating certain preconditioners, and access to the diagonal for diagonal preconditioning and other tricks.

3.3.5.1 Access to the nonzeros in a row

3.3.5.2 Access to values on the diagonal

You can extract the diagonal from an operator A that is stored in Epetra form

```
Vector<double> d = getEpetraDiagonal(A);
```

3.3.5.3 Getting a lumped diagonal

A lumped diagonal for A is a diagonal matrix D where $D_{ii} = \sum_{j=1}^{N} A_{ij}$. This can be done with high-level operations, as follows:

```
Vector<double> ones = A.domain().createMember();
ones.setToConstant(1.0);

Vector<double> rowSums = A*ones;

LinearOperator<double> lump = diagonalOperator(rowSums);
```

Because this involves multiplications as well as addition of matrix elements, it is very slightly less efficient than a specialized row sum function. However, not all sparse matrix implementations will have a row sum function, but all will have a matrix-vector multiply, so the multiplication method is a simple solution.

This method of finding lumped diagonals is a special case of a more general idea called probing, in which properties of a matrix are determined, or at least estimated, through multiplications with a strategically-chosen sequence of vectors.

3.3.6 Matrix-matrix operations

3.3.6.1 Matrix-matrix products

Multiplication of matrices is expensive, and explicit calculation of AB should almost always be avoided if an implicit calculation of ABx will suffice. However, in some cases there is no alternative but to form AB explicitly. The function `epetraMatrixMatrixProduct` will multiply two Epetra CRS matrices, returning the result (also stored as an Epetra CRS matrix) as a LinearOperator. This example does the operation $C = AB$.

```
LinearOperator<double> C = epetraMatrixMatrixProduct(A, B);
```

An important special case where some performance optimizations are possible is that of multiplication of a matrix A with a diagonal matrix D. The product DA is equivalent to scaling the rows of A by the corresponding diagonal entry of D, whereas the product AD scales the columns. The diagonal matrix D can be represented by a vector d containing its diagonal elements. The computation of DA is done as shown:

```
LinearOperator<double> DA = epetraLeftScale(d, A);
```

The matrix A is unchanged. Calculation of AD is, similarly,

```
LinearOperator<double> AD = epetraRightScale(A,d );
```

3.3.6.2 Matrix-matrix sums

Explicit matrix-matrix addition is done with the function epetraMatrixMatrixSum as follows:

```
LinearOperator<double> APlusB = epetraMatrixMatrixSum(A, B);
```

The two operands must have compatible shapes, but need not have the same sparsity graph.

3.3.6.3 Explicit diagonal matrix formation

If for some reason you need to work with an explicit Epetra matrix representation of a diagonal operator, you can create one from a `Vector` d as shown

```
LinearOperator<double> D = makeEpetraDiagonalMatrix(d);
```

3.4 Linear solvers

A linear solver is an object that takes a matrix A, a vector b, and solves the equation $Ax = b$. There are many methods for solving systems, and even given the same method we might have several different choices of implementations of that method. The Trilinos library, for instance, has three major linear solver packages

- Amesos is a collection of direct (sparse LU and sparse Cholesky) solvers suitable for PDE problems up to a few hundred thousand unknowns.
- Belos is a collection of Krylov methods
- Aztec is a legacy collection of Krylov methods, being superseded by Belos. Aztec is (at present) somewhat faster than Belos, but is more difficult to customize to problems involving block structure, unusual preconditioners, or unusual stopping conditions.

We might want to use any one of these and it needs to be convenient to switch from one solver to another. The most common method of creating a solver is to read parameters from an XML file and invoke the `createSolver()` static method of the `LinearSolverBuilder` class.

Unfortunately, at present the solver parameters are not well documented. In some solver packages they're not documented at all outside the source code. We need to work with the Trilinos solver developers to improve this situation.

3.4.1 Preconditioners

3.5 Nonlinear solvers

To be written

3.6 Eigensolvers

To be written

3.7 Review of design concepts

Some important points to take away from this description of linear algebra objects are:

1. We try to hide complicated, implementation-dependent operations (such as configuration and filling of a sparse matrix) behind abstract factory interfaces.
   (a) This lets us change implementations simply by changing which factory we use.
2. We do as few *explicit* matrix operations as possible. For example, in computing ABx or $A^{-1}x$ with overloaded operators we never actually form the matrices AB or A^{-1}. Explicit matrix-matrix products are available for those rare cases when they are needed.

3. We avoid low-level operations on vector *elements* whenever possible, preferring high-level functions that operate on the vector as an object.

Keep these considerations in mind as we move on to discuss objects for meshes, geometric regions, symbolic expressions, quadrature, and other components of a PDE simulation.

Chapter 4

Some examples

4.1 Poisson's equation

4.2 Poisson's equation with a nonlinear source

In this example we'll solve the nonlinear equation

$$\nabla^2 u = \alpha u^2 + x \quad \text{in } \Omega \tag{4.1}$$
$$u = 0 \quad \text{on } \Gamma.$$

We can't solve a nonlinear equation directly.

4.2.1 Solution by fixed-point iteration

The simplest approach is fixed-point iteration on the problem

$$\nabla^2 u_n = \alpha u_{n-1}^2 + x \quad \text{in } \Omega$$
$$u_n = 0 \quad \text{on } \Gamma$$

in which we have replaced the nonlinear term u^2 by its value at a previous iterate, u_{n-1}^2. We solve this problem until

$$\|u_n - u_{n-1}\| \leq \epsilon$$

for some specified tolerance ϵ and norm $\|\cdot\|$. The new features of this problem are

1. We need a way to represent the previous solution u_{n-1} as a symbolic object. There is an object type, `DiscreteFunction`, designed for exactly this purpose.
2. We won't store all the previous solutions: we only need one, which we'll call `uPrev`. After each step, we'll write the solution into `uPrev`.
3. In order to check convergence, we need to compute a norm of a symbolic expression.

Let's first look at the code to create a discrete function. A discrete function is based on a `DiscreteSpace` object.

```
DiscreteSpace discSpace(mesh, basis, vecType);
Expr uPrev = new DiscreteFunction(discSpace, 0.0);
```

The discrete function has been initialized to the constant value zero. We will see later how to create a non-constant discrete function.

With the discrete function ready, we can write the weak form and the linear problem.

```
Expr eqn = Integral(interior,
  (grad*v)*(grad*v)+v*(alpha*pow(uPrev,2.0)+x), quad);
Expr bc = EssentialBC(bdry, v*u, quad);

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

We now write the fixed-point iteration loop, which involves the norm check and the updating of the solution vector. There are a number of norms we can use in the convergence check, and in the next examples we'll show several norm computation methods. In the present example, we'll use the L^2 norm.

```
int maxIters = 20;
double tol = 1.0e-12;
bool converged = false;

Expr soln;

for (int i=0; i<maxIters; i++)
{
  soln = prob.solve(solver);

  /* Set up an expression for computation of the norm */
  Expr normIntExpr = Integral(interior, pow(soln-uPrev,2.0), quad);
  /* Compute the norm */
  double stepNorm = sqrt(evaluateIntegral(mesh, normExpr));

  /* Check for convergence */
  if (stepNorm < tol)
  {
    converged = true;
    break;
  }

  /* Write the current solution into the previous solution */
  Vector<double> solnVec = DiscreteFunction::discFunc(soln)->getVector();
  DiscreteFunction::discFunc(uPrev)->getVector();
}

TEST_FOR_EXCEPTION(!converged, RuntimeError,
  "Fixed-point␣iterations␣did␣not␣converge␣after␣" << maxIters
  << "␣iterations.");
```

If the algorithm has converged, the expression `soln` now contains the approximate solution.

4.2.2 Solution by Newton's method

In Newton's method for solving a nonlinear equation $F(u) = 0$, we linearize the problem about an estimated solution u_{n-1},

$$F(u_{n-1}) + \left.\frac{\partial F}{\partial u}\right|_{u_{n-1}} (u_n - u_{n-1}) = 0.$$

This linear equation is solved for $w = (u_n - u_{n-1})$, and then $u_n = u_{n-1} + w$ is the next estimate for the solution. Provided the initial guess is sufficiently close to the solution, the algorithm will converge quadratically. Without a good initial guess, the method can fail to converge. High-quality nonlinear solvers will have a method for improving global convergence. One class of globalization methods, the *line search* methods, limit the size of the step, *i.e.*, they update the solution estimate by

$$u_n = u_{n-1} + \alpha w$$

for some $\alpha \in (0, 1]$ chosen to ensure improvement in the solution. Refer to a text on nonlinear solvers (*e.g.* Dennis and Schnabel, or Kelley) for information on globalization methods.

Our example problem is to solve

$$F(u) = u^2 + x - \nabla^2 u = 0.$$

The derivative $\frac{\partial F}{\partial u}$ is the Frechet derivative, computed implicitly through the Gateaux differential

$$d_w F = \frac{\partial F}{\partial u} w.$$

Note that the Gateaux differential is exactly what appears in the equation for the Newton step, so we can write the linearized problem as

$$F(u_{n-1}) + d_w F(u_{n-1}) = 0.$$

The Gateaux differential is defined by

$$d_w F(u_{n-1}) = \lim_{\epsilon \to 0} \frac{F(u_{n-1} + \epsilon w) - F(u_{n-1})}{\epsilon}$$

from which the usual rules of calculus can be derived. For our example problem, we find

$$d_w F(u_{n-1}) = 2u_{n-1} w - \nabla^2 w.$$

Therefore the linearized equation for the Newton step w is

$$\left[u_{n-1}^2 + x - \nabla^2 u_{n-1}\right] + \left[2u_{n-1} w - \nabla^2 w\right] = 0.$$

While we can do linearization by hand, it is difficult and error-prone for complicated problems. Sundance has a built-in automated differentiation capability so that linearized equations can be derived automatically from a symbolic specification of the nonlinear equations. We will show examples of Newton's method with hand-coded linearized equations and with automated linearization.

4.2.2.1 Newton's method with hand-coded derivatives

First, we set up Newton's method by setting up a `LinearProblem` object for the equation

$$\nabla^2 u_n + \nabla^2 w = u_n^2 + 2u_n w + x$$
$$u_n + w = 0$$

for the step w. The unknown function is the step w. The previous iterate u_n is represented by a `DiscreteFunction`. Here are the steps for creating the linear problem.

```
Expr w = new UnknownFunction(basis);

/* other symbolic code omitted */

Expr stepEqn = Integral(interior,
  (grad*v)*(grad*(uPrev+w)) + v*(uPrev*uPrev + 2.0*uPrev*w+x), quad);
Expr stepBC = EssentialBC(bdry, v*(uPrev + w), quad);

LinearProblem stepProb(mesh, stepEqn, stepBC, v, w, vecType);
```

Now we can write the Newton algorithm.

```
int maxIters = 20;
double tol = 1.0e-12;
bool converged = false;

for (int i=0; i<maxIters; i++)
{
  Expr step = stepProb.solve(solver);
  Vector<double> stepVec = DiscreteFunction::discFunc(step)->getVector();
  Vector<double> uPrevVec = DiscreteFunction::discFunc(uPrev)->getVector();
  DiscreteFunction::discFunc(uPrev)->setVector(uPrevVec + stepVec);

  if (stepVec.norm2() < tol)
  {
    converged = true;
    break;
  }
}

TEST_FOR_EXCEPTION(!converged, RuntimeError,
  "Newton's␣method␣did␣not␣converge␣after␣" << maxIters << "␣steps");

Expr soln = uPrev;
```

Newton's method is done. The remainder of the code is for output of the solution and is unchanged from the fixed-point code.

4.2.2.2 Newton's method with automated derivatives

One of the most useful features of Sundance is its built-in automatic differentiation capability. You can write a nonlinear PDE as a Sundance `Expr`, and Sundance will do the Newton linearization for you.

The weak form for equation 4.1 is written

```
Expr eqn = Integral(interior, (grad*u)*(grad*v) + (alpha*u*u + x)*v, quad);
```

and the boundary conditions are written

```
Expr bc = EssentialBC(bdry, v*u/h, quad);
```

We use these to construct a `NonlinearProblem` object,

```
NonlinearProblem nlp(mesh, eqn, bc, v, u, uPrev, vecType);
```

Now we can write the Newton loop. We obtain the Jacobian as a `LinearOperator` and the residual as a `Vector` through a member function of the `NonlinearProblem`. We then solve the equation $Jw = -F(u_{n-1})$ for the step w.

```
for (int i=0; i<maxIters; i++)
{
  Vector<double> stepVec;
  nlp.setInitialGuess(uPrev);
  nlp.computeJacobianAndFunction(J, residVec);
  solver.solve(J, -1.0*residVec, stepVec);

  double stepNorm = stepVec.norm2();
  Vector<double> uPrevVec =DiscreteFunction::discFunc(uPrev)->getVector();
  DiscreteFunction::discFunc(uPrev)->setVector(uPrevVec + stepVec);

  Out::root() << "Iter=" << setw(5) << i << "␣||Delta␣u||=" << setw(20)
<< stepNorm << endl;

  if (stepNorm < tol)
  { converged = true; break; }
}
```

Notice that in this loop we

4.2.2.3 Black-box Newton's method with automated derivatives

4.3 Steady-state Navier-Stokes equations

4.4 Time-dependent Burgers equation

4.5 Time-dependent Navier-Stokes equations

4.5.1 Fully-implicit solution

4.5.2 Pressure projection solution

4.6 Exercises

1. Let V be a vector space of functions on a spatial domain Ω. The least-squares approximation in V to a function f is given by the orthogonal projection of f onto V. The solution u to

$$\int_{\Omega} (f - u)\, v\, d\Omega = 0 \quad \forall v \in V \tag{4.2}$$

is the orthogonal projection. Let Ω be the unit square, and take $f = \sin(2x)\, e^{y}$. Let V be the first-order Lagrange polynomials defined on a mesh of Ω.

   (a) Write a program that sets up and solves a `LinearProblem` representing equation 4.2. Your program should compute the L^2 norm of the approximation error, $e_h = \|u - f\|_2$.

   (b) Run the program on a sequence of meshes with decreasing h. Plot e_h versus h on a log-log plot, and use these results to find the order of accuracy p such that $e_h = O(h^p)$.

2. Consider the steady-state radiation diffusion equation,

$$\nabla^2 \left(u^4\right) = 0$$

on the unit square with boundary conditions

$$u = (1 + \sin(\pi x))^{1/4} \quad \text{along the line } y = 1$$

$$u = 1 \;\; \text{elsewhere on } \Gamma.$$

   (a) Derive a weak form of the problem

   (b) Derive a linearized weak form for the Newton step $w = u_n - u_{n-1}$.

   (c) Suppose you were to use an initial guess $u_0 = 0$ in Newton's method. What would happen, and why?

   (d) Write a program to set up and solve this problem using Newton's method.

3. Modify the Newton loop in example 4.2.2.2 so that the convergence test uses the L^2 norm of the step, rather than the 2-norm of the step vector.

Chapter 5

Geometry

Sundance solves PDEs on geometric domains that have been discretized to meshes. With a few simple exceptions, Sundance does not create meshes itself; usually, one will mesh a domain using an external meshing program such as Cubit or Triangle, and then read the results into Sundance.

When solving a PDE on a mesh, one needs to associate equations or boundary conditions with certain parts of the mesh. Identification of mesh entities (called cells) is done using `CellFilter` objects which examine the mesh cells and then "filter" those that obey a specified condition.

5.1 Meshes

Meshes are represented by Mesh objects. These objects are typically very large, so copies are shallow; that is, if you write code such as

```
Mesh mesh1 = getMyMeshFromSomewhere();
Mesh mesh2 = mesh1;
```

in which one mesh object is assigned to another, then both objects point to the same underlying data.

When writing simulation codes, you will rarely need any of the member functions of the mesh class. Refer to the Doxygen documentation for information on member functions.

5.1.1 Mesh types

Several types of mesh implementations are available in Sundance. In this manual, we will consider only one of them, the BasicSimplicialMesh. As the name indicates, this mesh type is restricted to simplices (tetrahedra, triangles, lines, and points). Any mesher or mesh reader has to know what type of mesh to make or to read. That specification is done via the MeshType object. Here we construct a basic simplicial mesh type,

```
MeshType meshType = new BasicSimplicialMeshType();
```

The use of this object will be shown below.

5.1.2 Mesh sources

Meshes can be created at runtime or read from a file. Both modes of creation are represented by a common `MeshSource` user interface. `MeshSource` is an abstract, extensible interface for mesh readers and mesh generators; several subtypes (listed below) have been implemented. Once created, an existing mesh can be transformed into another by means of a `MeshTransformation` object.

In the following example, we use a `MeshSource` to read an `Exodus` file "`mesh.exo`" and produce a `Mesh.`

```
MeshType meshType = new BasicSimplicialMeshType();
MeshSource reader = new ExodusMeshReader("wing", meshType);
Mesh mesh = reader.getMesh();
```

Several points need to be mentioned. First is that the argument meshType specifies the low-level mesh implementation to be used (here, BasicSimplicialMeshType). Second, the suffix of the filename has been dropped: the argument "wing" is used for the file "wing.exo." There is a reason for this: a mesh that has been partitioned for use in a parallel simulation will need to be distributed among several files, the names of which must be determined from a root such as "wing." For example, a four-processor run might need to find the files

```
wing.pxo
wing.4.0.exo
wing.4.1.exo
wing.4.2.exo
wing.4.3.exo
```

all of which can be determined with the name "wing," the communicator size, and the processor's rank.

5.1.2.1 Mesh readers

Currently, reading from Exodus, NetCDF, and Triangle files is supported. This manual will consider only Exodus readers; see the Doxygen for information on the other reader types[1].

5.1.2.2 Runtime mesh generators

Sufficiently simple regions can be meshed at runtime. Rectangles in 2D can be meshed in parallel with the PartitionedRectangleMesher, and 1D intervals with the PartitionedLineMesher. Refer to the Doxygen and to the examples for the arguments.

5.2 Cell filters

A weak PDE problem is stated in terms of integrals over subsets of a geometric domain. Typically, there will be an integral over the interior plus surface terms for the boundary conditions. In other cases such as fluid-structure interactions we may apply different weak equations on distinct maximal-dimension subsets of the entire domain. There may be point forces applied, or in the case of inverse problems, measurements taken at some subset of points. In contact problems, the subdomains on which constraints are to be applied will be determined as part of the solution. It is thus necessary to have a very flexible system for specification of geometric regions.

Any specification of a domain of integration must be able to identify on a mesh the set of cells on which a particular integration is to be done. In general, then, a specification of a subregion is a specification of a *filter* that can extract from a mesh the subset of cells which satisfies some condition, i.e., those cells that "pass through" the filter.

5.2.1 Filters on meshes

The coarsest view of a mesh is simply as the set of its cells. Any subset of these cells can be produced through an appropriate filter acting on the mesh. The Sundance `CellFilter` object does this job.

The simplest cell filters are those based on cell dimension

[1]Exodus is a binary mesh file format developed at Sandia National Laboratories. The NetCDF and Triangle readers exist for historical reasons: the Exodus I/O library used to have a restrictive license, requiring use of an alternative format for use outside Sandia. The Exodus library is now released open source, so while the NetCDF and Triangle readers still exist the Exodus reader is most widely used because the Exodus format is supported natively by a number of meshing programs such as Cubit.

- `MaximalCellFilter` passes all cells of maximal dimension (*i.e.* dimension equal to the spatial dimension of the mesh).
- `DimensionalCellFilter` passes all cells of a specified dimension

These two filter types will be used in nearly every problem, because you'll construct more complex filters through operations on these two fundamental filters.

5.2.1.1 Selecting cells by label

Most mesh generation programs allow labeling of volumes, surfaces, curves, and vertices.

```
CellFilter faces = new DimensionalCellFilter(2);
CellFilter top = faces.labeledSubset(7);
```

5.2.1.2 Selecting cells by predicate

Selecting cells by some criterion other than label requires working with predicates. Predicates are represented by CellPredicate objects. CellPredicate is a handle class for CellPredicateBase, which defines an abstract interface for predicates. If you have defined some predicate type MySpecialPredicate, you would use it as follows:

```
CellPredicate myPred = new MySpecialPredicate();
CellFilter mySpecialFaces = faces.subset(myPred);
```

A shortcut is to construct within the subset() function,

```
CellFilter mySpecialFaces = faces.subset(new MySpecialPredicate());
```

which does the same thing with a little less typing.

5.2.1.3 Selecting cells having a specified coordinate value

A very common filtering operation is to select all cells whose vertices have a specified coordinate value, say, $y = 2$. The predicate to test whether the i-th coordinate x_i is within a tolerance ϵ of a is written

$$p_{i,a,\epsilon}(\mathbf{x}) = \begin{cases} 1 & |x_i - a| \leq \epsilon \\ 0 & \text{otherwise} \end{cases}$$

There is a predefined object for this, CoordinateValueCellPredicate. The constructor for this object takes two required arguments: a coordinate index i and a coordinate value a, and an optional tolerance ϵ.

```
CellFilter edges = new DimensionalCellFilter(1);

/* Select edges having vertices at y=2.0. Default tolerance used. */
CellFilter y2 = edges.subset(new CoordinateValueCellPredicate(1, 2.0));

/* Select edges having vertices at x=5.0 to a tolerance of 1.0e-4. */
CellFilter x5 = edges.subset(new CoordinateValueCellPredicate(0, 5.0, 1.0e-4));
```

The default tolerance is $\epsilon = 10^{-10}$.

5.2.1.4 Writing a user-defined position-based predicate

If you want to write a predicate that is a more complicated function of vertex position, you'll need to write a subclass of CellPredicateFunctorBase. The subclass must define a function operator() which takes a Point object; this function should return true if a vertex's position satisfies the predicate's condition.

For example, here is a predicate that tests whether a point is a specified distance a from the origin, to within a tolerance of 10^{-5}.

```
class RadiusTest: public CellPredicateFunctorBase
{
public:
  /** Constructor */
  RadiusTest(double a)
  : CellPredicateFunctorBase("RadiusTest(" + Teuchos::toString(a)+")"),
    a_(a), tol_(1.0e-10) {}

  /** */
  bool operator()(const Point& x) const {return std::fabs(x*x-a_*a_) < tol_;}

private:
  double a_;
  double tol_;
};
```

You can write arbitrarily complicated tests in this way. Once such a class has been defined, it can be used in filters,

```
CellFilter faces = new DimensionalCellFilter(2);

/* Select faces whose vertices are at r=3. */
CellFilter r3 = edges.subset(new RadiusTest(3.0));
```

5.2.1.5 Simple creation of predicates through the CELL_PREDICATE macro

CELL_PREDICATE is a helper macro that lets you streamline the creation of sufficiently simple CellPredicateFunctorBase subtypes. Suppose your problem's geometry has a feature, the "nozzle," on the disk $z = 4$, $x^2 + y^2 \leq 1$. You can then write

```
CELL_PREDICATE(NozzlePointTest,\
{\
double r2 = x[0]*x[0]+x[1]*x[1];
double z = x[2];
bool rtn = (r2 <= 1.0) && (std::fabs(z-4.0)<1.0e-6);\
return rtn;\
})
```

Use in code is then

```

CellFilter nozzle = faces.subset(new NozzlePointTest());
```

The limitation of the macro is that it is not possible to pass arguments to the predicate's constructor, *i.e.*, you cannot vary the position z of the nozzle at runtime by means of a constructor argument.

Chapter 6

Discrete vector spaces

6.0.2 Basis families

Lagrange and friends

6.0.3 DiscreteSpace objects

How integrations are done

Chapter 7

Symbolic expressions

7.1 Expression subtypes

7.1.1 Lists

Expressions can be grouped into lists with an arbitrary structure. Important special cases are scalar, vector, and tensor expressions, but lists can have heterogeneous structure as well. Here's an example; don't worry for now what the function types mean; concentrate on the list structure.

```
/* Create a vector-valued expression */
Expr vx = new TestFunction(new Lagrange(2));
Expr vy = new TestFunction(new Lagrange(2));
Expr v = List(vx, vy);

/* Create a scalar-valued expression */
Expr q = new TestFunction(new Lagrange(1));

/* Create a heterogeneous list {{vx,vy},q}. */
Expr vq = List(v, q);
```

7.1.1.1 Operations on lists

Lists with identical structures can be added and subtracted. If the structures are not identical an exception is thrown.

```
/* x={a,b},y={c,d},z={e,f,g} */
Expr sum = x+y;      // returns {a+c, b+d}
Expr diff = x-y;     // return  {a-c, b-d}
Expr bogus = x+z;    // FAILS! A runtime error will be thrown.
```

The multiplication operator (*) on lists denotes the inner product between vectors or tensors; the operands must have list structure such that an inner product is defined. Multiplication of a scalar by a list threads the multiplication over the list entries.

```
Expr x_dot_y = x*y;       // returns a*c + b*d
Expr x2 = 2.0*x;          // returns {2*a, 2*b}
Expr bogus2 = z*x;        // FAILS!
```

The * operator is also used to represent the application of a differential operator (see below).

Division of a list by a scalar threads the division over the list entries. Division *by* a list is not defined.

```
Expr x_over_4 = x/4.0;       // returns {a/4.0, b/4.0}
Expr bogus3 = x/y;           // FAILS! Division by {c,d} is nonsense.
```

7.1.1.2 Low-level list creation.

The List function works for up to ten arguments. To create a list with more than 10 elements, incrementally add elements using the append() function.

7.1.2 Constants and parameters

The simplest type of Expr to create is a real-valued constant expression. For example,

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

Any double-precision constant appearing in an expression, for instance, the constant 2.0 in the expression

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

will be represented internally by a constant-valued expression. *It is important to understant that once created and used in an expression, a constant's value is immutable. Should you want to change the expression's value during runtime, you should instead use a Parameter.*

For example, the following code to print a sequence of pairs $(t_i, \sin(\pi t_i))$ will not work as intended:

```
Expr time = 0.0;
Expr f = sin(pi*time);
for (int i=0; i<10; i++)
{
  Out::os() << time << "␣" << f << endl;
  // update the time
  time = 0.1*i;
}
```

To get the intended effect of updating time in the expression $f(t) = \sin(\pi t)$, do the following

```
Expr time = new Parameter(0.0);
Expr f = sin(pi*time);
for (int i=0; i<10; i++)
{
  Out::os() << time << "␣" << f << endl;
  // update the time
  time.setParameterValue(0.1*i);
}
```

To summarize the difference between constants and parameters: constants are constant with respect to the mesh and with respect to runtime, whereas parameters are constants with respect to the mesh but variable with respect to runtime. That is, a constant has the same value at every point in the mesh, and cannot be changed during runtime. A parameter has the same value at every point in the mesh, but that value can be changed during runtime.

7.1.3 Coordinates and derivatives

You can build position-dependent functions using coordinate functions, which represent the Cartesian coordinates. A coordinate function is created using the CoordExpr constructor with an integer zero-based index specifying the coordinate direction. Index 0 gives the coordinate function x, indices 1 and 2 gives y and z, respectively.

```
Expr x = CoordExpr(0);
Expr y = CoordExpr(1);
Expr r = sqrt(x*x+y*y);
```

Partial differential operators are created in a similar way: the operator $\frac{\partial}{\partial x}$ is represented by

```
Expr dx = Derivative(0);
```

The application of a differential operator to an expression is done using the * operator.

```
Expr dr_dx = dx*r;
```

7.1.3.1 Vector differential operators

The operations of vector calculus can be composed directly from partial derivative operators.

```
Expr curlU = curl(u);
Expr divU = div(u);
```

is equivalent to

```
Expr curlU = cross(del, u);
Expr divU = del*u;
```

Tensor expressions such as $\nabla \mathbf{v} : \nabla \mathbf{u}$ appearing in the weak Navier-Stokes equations can be carried out with the `outerProduct` and `colonProduct` functions

```
Expr gradVColonGradU = colonProduct( outerProduct(del, v), outerProduct(del, u) );
```

7.1.4 Test and unknown functions

The unknown and test functions appearing in a weak problem are represented by the UnknownFunction and TestFunction classes.

```
BasisFamily P2 = new Lagrange(2);
Expr q = new TestFunction(new Lagrange(1));
Expr u = new UnknownFunction(List(P2, P2, P2));
```

7.1.5 Discrete functions

7.2 Operations

7.2.1 Elementary functions

7.2.2 User-defined functions

Chapter 8

Weak forms and boundary conditions

8.1 Weak forms

A weak form is an expression such as

$$\int_{\Omega} \kappa \nabla v \cdot \nabla u \, d\Omega - \int_{\Gamma} g v \, d\Gamma.$$

The integrands can be composed out of objects (type `Expr`) such as `kappa`, `v`, `u`, and `g` representing the mathematical quantities κ, v, u, and g. The regions Ω and Γ are represented by cell filters, say `omega` and `gamma`. Additionally, it is necessary to specify how the integrals are to be computed; this specification is done by a QuadratureFamily object, called, say, `quad`. An object `wf` for the weak form shown above would then be created by the following code

```
Expr wf = Integral(omega, kappa*(grad*v)*(grad*u), quad)
  - Integral(gamma, g*v, quad);
```

8.1.1 Quadrature

Specification of a weak form must include specification of a method for computing the necessary integrals. This is done through the `QuadratureFamily` class hierarchy, which provides an interface for building quadrature rules appropriate to a cell type. The most widely used quadrature family is `GaussianQuadrature`, which produces Gauss rules on lines, and

- Gauss-Dunavant symmetric rules on triangles and tets, if such a rule is available for the order requested
- Collapsed (non-symmetric) tensor product Gauss rules on triangles and tets, if no Gauss-Dunavant rule is available. These rules will integrate polynomials exactly up to the specified order, but may perform poorly on other functions (including higher-order polynomials) because the points are biased towards one of the corners.
- Tensor-product Gauss rules on quads and hexes.

Other families include Fekete and low-order Newton-Cotes quadrature, sometimes preferred for specialized purposes. An example of quadrature specification is

```
QuadratureFamily quad4 = new GaussianQuadrature(4);
Expr wk = Integral(someCells, someExpr, quad4);
```

In some cases, the quadrature order required can't be determined at runtime, so a `QuadratureFamily` object can't be constructed directly. The `QuadratureType` class heirarchy provides factory objects that can build `QuadratureFamily` objects dynamically given a specification of order.

8.1.2 Integral expressions

Definite integrals play a central role in finite element methods so you will use integral expressions in nearly every simulation. Integrals are represented internally by an expression subtype that is rarely constructed directly. Almost always you will build integrals using the `Integral` function, as in the first example in this section. There is a limited set of operations on integral expressions: you can add or subtract two integrals, and you can multiply integrals by an expression that is constant in space. Other operations, even mathematically well-defined operations such as $\sqrt{\int_0^1 f^2\,dx}$, cannot be done using integral expressions. Here is an example of some operations, valid and not:

```
Expr I1 = Integral(omega1, v*f, quad1);
Expr I2 = Integral(omega2, v*g, quad2);
Expr I3 = Integral(omega1, v*g, quad2);
Expr I4 = Integral(omega2, v*f, quad2);

Expr total = I1 + 2.0*I2 - I3 + 3.0*I4;

Expr crash = I1*I2;          // mathematically valid, but not defined in code
Expr burn = sqrt(I1);        // mathematically valid, but not defined in code
Expr nonsense = dx*I3;       // mathematically undefined (I3 is definite!)
```

It is worth commenting on several points in this example.

- There are two different quadrature methods used, `quad1` and `quad2`. Therefore, although `I1` and `I3` are defined on the same cell filter, their difference `I1-I3` *cannot* be combined into a single function call

  ```
  Integral(omega1, v*(f-g), doh)
  ```

  Upon computing I1-I3, the integrands are stored separately, each associated with its unique combination of cell filter and quadrature family[1].

- Integral expressions having the same cell filters and the same quadrature rules are, when added or subtracted, collected into a single object. In the example, I2 and I4 have the same cell filter (omega1) and the same quadrature family (quad2) so that 2.0*I2 + 3.0*I4 is exactly equivalent to

  ```
  Integral(omega1, v*(2.0*f + 3.0*g), quad2)
  ```

- Be careful: Note that

  ```
  Integral(omega1, v*f, quad1)+Integral(omega2, v*f, quad2)
  ```

  is equivalent to

  ```
  Integral(omega1+omega2, v*f, quad1)
  ```

  but that

  ```
  Integral(omega1, v*f, quad1)-Integral(omega2, v*f, quad2)
  ```

  is *not* equivalent to

  ```
  Integral(omega1-omega2, v*f, quad1)
  ```

[1]Advanced users who explore Sundance's internals will encounter objects called region-quad combinations (RQC) which are used to tag integrands.

8.2 Neumann and Robin BC

With a Neumann BC we specify the normal derivative of the unknown u, for example,

$$\kappa \mathbf{n} \cdot \nabla u = g$$

In a Galerkin formulation these are easily implemented because the normal derivative appears explictly after integration by parts. For example, the weak Poisson equation is

$$\int_{\Omega} \kappa \nabla v \cdot \nabla u \, d\Omega - \int_{\Gamma} v \kappa \mathbf{n} \cdot \nabla u \, d\Gamma = 0.$$

Substituting g for $\kappa \mathbf{n} \cdot \nabla u$ in the surface term

$$\int_{\Omega} \kappa \nabla v \cdot \nabla u \, d\Omega - \int_{\Gamma} v g \, d\Gamma = 0$$

gives consistency. The surface integral is implemented just like any other integral: with the `Integral` function. For example,

```
Expr neumSurfIntegral = Integral(gammaNeum, v*g, quad);
```

Note that the expression g may be an arbitrary function of u, so that nonlinear boundary conditions such as the radiative condition

$$\kappa \mathbf{n} \cdot \nabla u = -\sigma \left(u^4 - u_0^4 \right)$$

may be implemented in this way. Simply replace g with $-\sigma \left(u^4 - u_0^4 \right)$ in the code above.

A Robin BC specifies a linear combination of the unknown and its normal derivative, for example

$$-\alpha \left(u - u_R \right) + \kappa \mathbf{n} \cdot \nabla u = 0.$$

As with a Neumann BC, we can substitute for $\kappa \mathbf{n} \cdot \nabla u$ in the surface term to obtain

$$\int_{\Omega} \kappa \nabla v \cdot \nabla u \, d\Omega - \int_{\Gamma} \alpha v \left(u - u_R \right) \, d\Gamma = 0.$$

8.3 Dirichlet BC

Imposing Dirichlet BCs is less straightforward than imposing Neumann and Robin BCs. There are several alternatives

8.3.1 Nitsche's method

Nitsche devised a clever method for applying Dirichlet BC in such a way that symmetry and coercivity are preserved. Modify the weak Poisson equation by adding the terms indicated by an underbrace.

$$\int_{\Omega} \kappa \nabla v \cdot \nabla u \, d\Omega - \int_{\Gamma_D} v \kappa \mathbf{n} \cdot \nabla u \, d\Gamma \underbrace{- \int_{\Gamma_D} \left(u - u_D \right) \kappa \mathbf{n} \cdot \nabla v \, d\Gamma + \gamma \int_{\Gamma_D} h^{-1} \kappa v \left(u - u_D \right) \, d\Gamma} = 0.$$

The additional terms are zero when the BC are satisfied, so the modified problem is consistent. Clearly the weak form is symmetric. Nitsche proved that there is a $\gamma_0 > 0$ such that coercivity is obtained for all $\gamma > \gamma_0$. Coercivity implies stability, which together with consistency and the Lax-Milgram lemma implies convergence.

One can easily code the boundary terms by hand, however, it is a common enough operation to warrant a packaged solution

```
Expr nitscheBC = NitschePoissonDirichletBC(dim, diriSurf, quad, kappa,
  v, u, uD, gamma);
```

which returns all three surface terms in the weak form above.

Note that this operation is specific to BC arising in Poisson's equation and variants thereof, for example the steady radiation diffusion equation in which case $\kappa = u^3$. Dirichlet BCs $u = u_D$ when $\kappa = u^3$ would be implemented through

```
Expr nitscheBC = NitschePoissonDirichletBC(dim, diriSurf, quad, pow(u,3),
  v, u, uD, gamma);
```

The convergence theory requires $u > 0$ (which is, of course, also required on physical grounds because the temperature must be positive).

In general, a Nitsche method must be derived for each operator. Methods for some common operators are available in the literature, for example, a Nitsche method has been formulated for no-slip BC $\mathbf{u} = \mathbf{u_{BC}}$ for the Stokes or Navier-Stokes equations. This is also available as a packaged function in the Sundance library,

```
Expr bc = NitscheStokesNoSlipBC(diriSurf, quad, nu, v, q, u, p, uBC, C1, C2);
```

where ν is the viscosity and C_1 and C_2 are positive constants.

When available, the Nitsche method preserves symmetry and obtains good scaling. A minor disadvantage is the need to estimate constants such as γ or C_1 and C_2.

8.3.2 Replacement method

When Nitsche's method cannot be used, an alternative suitable for certain problems is to force the Dirichlet boundary conditions through a side condition that replaces a subset of the discrete equations with equations that impose the Dirichlet BC.

```
Expr replaceBC = EssentialBC(diriSurf, v*(u-uD)/h, quad);
```

Division by the cell diameter h is optional, but helps improve conditioning of the resulting linear system.

The most significant drawback of this method is that it destroys any symmetry of the original problem. Nitsche's method is therefore preferred when possible. Note also that the replacement method cannot be used directly on an eigenvalue problem.

8.3.3 Lagrange multiplier method

In problems arising from a variational principle, Dirichlet boundary conditions can be enforced as a constraint through the method of Lagrange multipliers. Some disadvantages of this method are that the resulting linear system is indefinite, that one must introduce a new variable for the Lagrange multiplier, and that the basis for the multiplier function must be chosen carefully to be consistent with the LBB condition. Indefiniteness can be addressed through the use of an augmented Lagrangian method.

Because of the close connection to optimization, further discussion of this method is deferred until the section on PDE-constrained optimization.

8.4 Miscellaneous

8.4.1 Point loads and Dirac delta functions

A point source is most conveniently, and accurately, modeled when it is located at a vertex in the mesh. Let $\mathbf{a}$ be the location and q be the strength of a point load, which enters a weak form through an integral involving the Dirac delta function,

$$\int_\Omega vq\delta(\mathbf{x} - \mathbf{a})\, d\Omega$$

which evaluates to $v(\mathbf{a})q(\mathbf{a})$. Rather than representing the Dirac delta function we write this integral using a Dirac measure $d\mu_{\mathbf{a}}$ so that

$$\int_{\Omega} vq\delta\left(\mathbf{x}-\mathbf{a}\right)\, d\Omega = \int_{\Omega} vq\, d\mu_{\mathbf{a}} = \int_{\mathbf{a}} vq\, d\mu_{\mathbf{a}}.$$

Introduce a cell filter `pointA` that selects the vertex located at $\mathbf{a}$. The above integral would be written

```
Expr ptTerm = Integral(pointA, v*q, quad);
```

The quadrature argument is an unused placeholder, needed only to maintain consistent syntax. Any integral taken over a zero-cell is interpreted to use the Dirac measure.

8.4.2 Absorbing BC

8.5 Definite integrals

8.5.1 Probing values

Chapter 9

Problem specifications

9.1 Orthogonal projections

9.2 Forward problems

9.2.1 Linear problems

9.2.1.1 Blocked variables

9.2.1.2 Problems with some variables held fixed

9.2.2 Nonlinear problems

9.2.2.1 Blocked variables

9.2.2.2 Problems with some variables held fixed

9.2.3 Sensitivity analysis

9.2.4 Spectral uncertainty quantification

9.3 Optimization

9.3.1 Functionals and variations

9.3.2 Full-space optimization

9.3.3 Reduced-space optimization

9.4 Eigenvalue problems

Chapter 10

Postprocessing

10.1 Field writers

- VTK format
- Exodus format
- Column formatted (Matlab, Gnuplot)
- Triangle format
- Verbose

Appendix A

Debugging tips

A.1 Diagnostics

A.1.1 Watch flags

Any `Integral` or `EssentialBC` function can accept an optional `WatchFlag` argument that controls the amount and type of diagnostic information to be printed. The type of information requested A verbosity level of zero means no output will be printed. Amount of diagnostic information increases as the verbosity level increases; typically, level one gives a top-level view of what's being done, whereas level four or above provides considerable detail and values of intermediate calculations such as element integrations and symbolic evaluations.

```
WatchFlag watchSource("source␣term");
watchSource.setParam("evaluation", 4);

Expr eqn = Integral(interior, (grad*v)*(grad*u), quad)
  + Integral(interior, v*f, quad, watchSource);
```

A.1.1.1 Global watch flags

These diagnostic types can't be localized to a single term in a problem. Setting one of these in any term will enable it for all other terms given to a problem

Name	Description
"`solve control`"	High-level progress reports from solver drivers such as `LinearProblem`. For details of linear or nonlinear solver progress, use the solver object's verbosity setting.
"`eval mediator`"	Details of communication between symbolic objects and discrete objects.
"`assembler setup`"	High-level progress report on the construction of the Assembler object
"`dof map setup`"	Details on setup of DOF map.
"`equation set setup`"	High-level progress report on setup of the EquationSet, the object that organizes the symbolic equations, drives symbolic preprocessing, and determines maps from cell filters to equations and functions
"`matrix config`"	Configuration of sparse matrix
"`vector config`"	Configuration of vector
"`assembly loop`"	High-level progress report on assembly loop.

A.1.1.2 Term-specific watch flags

These watch flags will request increased detail for the expression in which they are used. In order to put the information in context, they may also turn on low-verbosity output tracking at the high level.

Name	Description
“`evaluation`”	Details of evaluation of symbolic expressions during assembly loop.
“`discrete function evaluation`”	Details of evaluation of discrete functions during symbolic calculations
“`symbolic preprocessing`”	Details of expression graph determination and evaluation construction during equation set setup .
“`integration setup`”	Details about construction of element integrals during assembler setup
“`integration`”	Details on evaluation of element integrals during assembly loop
“`integral transformation`”	Details on coordinate transformations during element integration
“`fill`”	Details on target (*e.g.*, matrix) loading during assembly loop

A.1.2 Viewing low-level data structures

A.1.2.1 Viewing a problem’s DOF maps

A `LinearProblem` or `NonlinearProblem` will have two arrays of DOF maps, one array for the row maps and one for the column maps. The arrays are to deal with block operators: In a problem where the test and unknown functions are grouped into blocks, there will be one row map for each block of test functions and one column map for each block of unknown functions. In the example we show how to print the row maps for a problem (linear or nonlinear) named `prob`.

```
for (int r=0; r<prob.numBlockRows(); r++)
{
  Out::root() << "showing␣DOF␣map␣for␣block␣row␣r=" << r << endl;
  prob.rowMap(r)->print(Out::os());
}
```

A.1.2.2 Viewing a discrete space’s DOF map

A DiscreteSpace has an associated DOF map, which may be accessed through the map() member function.

```
DiscreteSpace discSpace(mesh, basis, vecType);
Out::root() << "showing␣DOF␣map␣for␣discrete␣space" << endl;
discSpace.map()->print(Out::os());
```

A.1.2.3 Viewing a problem’s matrices and vectors

The matrix and RHS vector for a discretized `LinearProblem` can be obtained by the `getOperator()` and `getSingleRHS()` member functions. In sensitivity analysis, there may be multiple RHS vectors which can be obtained (as an array) by the `getRHS()` member function.

When working with a nonlinear problem, the current Jacobian and residual can be obtained by calling the `computeJacobianAndFunction()` member function. If only the function value is desired, it can be obtained (in `Vector` form) by the `computeFunctionValue()` function. The current evaluation point in `Expr` form may be obtained by the `getU0()` member function, or in `Vector` form by the `getInitialGuess()` function.

A.1.2.4 Testing differentiation

Differentiation of a functional can be tested against a finite-difference calculation by the `fdGradientCheck()` member function of `FunctionalEvaluator`. While running, it prints detailed information on the gradient vectors obtained through both finite difference and in-place differentiation. The finite difference stepsize is specified as an argument to `fdGradientCheck()`. The return value is the maximum component of the vector of errors.

A.2 Debugger helpers

A.2.1 Debugger configuration

The `gdb` debugger and its various front ends (such as DDD) can be given startup options through the `.gdbinit` file. Some useful startup options are

```
set breakpoint pending on
break TestForException_break
break abort
```

The first line is needed for deferred setting of breakpoints when working with dynamically-loaded libraries. The second and third lines set breakpoints in the standard C `abort()` function and in the Teuchos `TestForException_break()` function. All errors detected by Sundance components are handled by the TEST_FOR_EXCEPTION() macro described in 2.2.2, which internally calls `TestForException_break()`. The `abort()` function may be called by sufficiently catastrophic errors in low-level code.

A.2.2 Parallel debugging

The most convenient way to debug in parallel is to use a parallel debugger such as Totalview. If a parallel debugger is not available, it is possible to bind multiple sessions of a gdb-based debugger to multiple jobs running on a single host.

Appendix B

Miscellaneous math

B.1 Helpful formulas and identities

B.1.1 Outer product (Kronecker product)

Let **a** and **b** be vectors in $\mathbb{R}^N$. Then $\mathbf{a} \otimes \mathbf{b}$ is the $N \times N$ matrix

$$\begin{pmatrix} a_1b_1 & a_1b_2 & \cdots & a_1b_N \\ a_2b_1 & a_2b_2 & & \\ \vdots & & \ddots & \vdots \\ a_Nb_1 & & \cdots & a_Nb_N \end{pmatrix}.$$

The gradient of a vector-valued function **u**, written $\nabla\mathbf{u}$, can be expressed in outer product notation as $\nabla \otimes \mathbf{u}$.

B.1.2 Colon product (Frobenius product)

Let **A** and **B** be $N \times N$ matrices. Then the Frobenius product is

$$\mathbf{A} : \mathbf{B} = \sum_{i=1}^{N} \sum_{j=1}^{N} A_{ij} B_{ij}.$$

The Frobenius product is useful in writing the weak form of the Stokes equations.

B.1.3 Integral identities

In the following identities Ω is a smooth subset of $\mathbb{R}^N$ and Γ is its boundary. Notation: ϕ, u and v are scalar functions, **v** and **u** are vector-valued functions, and **K** is a tensor-valued function.

$$\int_\Omega v\nabla^2 u \, d\Omega = -\int_\Omega \nabla v \cdot \nabla u \, d\Omega + \int_\Gamma v\mathbf{n}\cdot\nabla u \, d\Gamma$$

$$\int_\Omega v\nabla \cdot [\kappa\nabla u] \, d\Omega = -\int_\Omega \kappa\nabla v \cdot \nabla u \, d\Omega + \int_\Gamma v\kappa\mathbf{n}\cdot\nabla u \, d\Gamma$$

$$\int_\Omega v\nabla \cdot [\mathbf{K}\nabla u] \, d\Omega = -\int_\Omega \nabla v \cdot (\mathbf{K} \cdot \nabla u) \, d\Omega + \int_\Gamma v\mathbf{n}\cdot (\mathbf{K} \cdot \nabla u) \, d\Gamma$$

$$\int_\Omega \mathbf{v} \cdot \nabla\phi \, d\Omega = -\int_\Omega \phi\nabla \cdot \mathbf{v} \, d\Omega + \int_\Gamma \phi\mathbf{v} \cdot \mathbf{n} \, d\Gamma$$

$$\int_\Omega \mathbf{v} \cdot \nabla^2\mathbf{u} \, d\Omega = -\int_\Omega \nabla\mathbf{v} : \nabla\mathbf{u} \, d\Omega + \int_\Gamma (\mathbf{n}\otimes\mathbf{v}) : \nabla\mathbf{u} \, d\Gamma$$

B.2 The formal logic of filters

Definition 1. Let S be a set. A *filter* applied to S returns a member of the power set of S.

Example 2. The identity filter I acts on S to return S: $I(S) = S$. The zero filter 0 produces the empty set: $0(S) = \emptyset$.

B.2.1 Predicates

To go beyond those trivial examples we need to find a way of selecting members of a set, which we do using a logical operation called a predicate.

Definition 3. Let S be a set. A predicate $p : S \to \{0, 1\}$ is any function that maps members of S to the booleans.

With a predicate p, we can filter a finite set S by applying the predicate to every member of S, returning the subset of members such that the predicate evaluates true.

Definition 4. A predicate p can define a filter F^p, operating on a set S as

$$F^p(S) = \{s \in S | p(s) = 1\}.$$

Example 5. Let $S = \{1, 4, 9, 16, 25, 36, 49\}$. Define $p(x) = \begin{cases} 1 & x \text{ even} \\ 0 & x \text{ odd} \end{cases}$. Then $F^p(S) = \{4, 16, 36\}$.

B.2.2 Binary operations between filters

We can define binary operations on filters in terms of binary operations on the sets they produce.

Definition 6. The union, intersection, and difference operations on two filters produce the corresponding operations on the output of the filters. Let S be a set and F_1 and F_2 be two filters. Then

$$(F_1 \cup F_2)(S) = F_1(S) \cup F_2(S)$$

$$(F_1 \cap F_2)(S) = F_1(S) \cap F_2(S)$$

$$(F_1 - F_2)(S) = F_1(S) - F_2(S).$$

When two filters are defined in terms of predicates, an equivalent definition of the binary operations can be given in terms of the binary logical operations $\vee$ (OR) and $\wedge$ (AND), and the unary operation $\sim$ (NOT). Specifically,

$$F^p \cup F^q = F^{p \vee q}$$

$$F^p \cap F^q = F^{p \wedge q}$$

$$F^p - F^q = F^{p \wedge \sim q}.$$

You can easily verify that these definitions are equivalent to the original definitions in terms of sets.

A filter is not a set, so the normal definition of subset does not apply to filters. However, it's convenient to speak (loosely) of subsets of filters.

Definition 7. A filter G is a subset of a filter F if $G(S) \subset F(S)$ for all S.

Example 8. All filters are subsets of the identity filter.

Appendix C

Complete example codes

C.1 Conjugate gradient example

```
#include "Sundance.hpp"
#include "FDMatrixPoisson1D.hpp"

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

    int numPerProc = 4;

    VectorType<double> vecType = new EpetraVectorType();
    LinearOperator<double> A = buildFDPoisson1D(vecType, numPerProc);

    Out::root() << "Matrix A = " << endl;
    Out::os() << A << endl;

    VectorSpace<double> space = A.domain();
    Vector<double> b = space.createMember();

    b.setToConstant(1.0);

    Vector<double> x = b.copy(); // NOT x=b, which would be a shallow copy
    Vector<double> r = b - A*x;
    Vector<double> p = r.copy();

    double tol = 1.0e-12;
    int maxIter = 100;

    bool converged = false;

    Out::root() << "Running CG" << endl;
    Out::root() << "tolerance = " << tol << endl;
    Out::root() << "max iters = " << maxIter << endl;
    Out::root() << "-------------------------------------------------------"
                << endl;

    for (int i=0; i<maxIter; i++)
    {
      Vector<double> Ap = A*p; // save this, because we'll use it twice

      double rSqOld = r*r;
      double pAp = p*Ap;
      double alpha = rSqOld/pAp;

      x = x + alpha*p;
      r = r - alpha*Ap;  // used Ap again

      double rSq = r*r;
      double rNorm = sqrt(rSq);
      Out::root() << "iter=" << setw(6) << i << setw(20) << rNorm << endl;

      if (rNorm < tol) { converged = true; break; }

      double beta = rSq/rSqOld;

      p = r + beta*p;
    }

    Out::root() << "Solution: " << endl;
    Out::os() << x << endl;

  }
    catch(exception& e)
  {
    Sundance::handleException(e);
  }
  Sundance::finalize();
}
```

C.1.1 Finite difference matrix for Poisson 1D