

SAND REPORT

SAND2004-4793
Unlimited Release
July 2004

Sundance 2.0 Tutorial

Kevin Long
Computational Science and Mathematics Research Department
Sandia National Laboratories
Livermore, CA

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



Sandia National Laboratories

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

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

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

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

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

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

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/ordering.htm>



Abstract

Sundance is a system of software components that allows construction of an entire parallel simulator and its derivatives using a high-level symbolic language. With this high-level problem description, it is possible to specify a weak formulation of a PDE and its discretization method in a small amount of user-level code; furthermore, because derivatives are easily available, a simulation in Sundance is immediately suitable for accelerated PDE-constrained optimization algorithms.

This paper is a tutorial for setting up and solving linear and nonlinear PDEs in Sundance. With several simple examples, we show how to set up mesh objects, geometric regions for BC application, the weak form of the PDE, and boundary conditions. Each example then illustrates use of an appropriate solver and solution visualization.

Acknowledgement

The author would like to acknowledge the support of the LDRD and CSRF programs that funded development of Sundance. In addition, a number of people made valuable contributions to the design and development of Sundance 2.0. Paul Boggs, Ross Bartlett, and Bart van Bloemen Waanders provided many useful suggestions for applications to PDE-constrained inverse problems, and Vicki Howle had numerous suggestions for applications to solver algorithm development. Ross Bartlett's ideas on disciplined interface design helped significantly in making the internal code for version 2.0 considerably cleaner than version 1.0. Jill Reese wrote several TSF classes making the use of NOX solvers in Sundance possible, and moreover suffered through the buggy pre-release editions of Sundance and caught many glitches and syntactical quirks.

SAND2004-4793
Unlimited Release
Printed July 2004

Sundance 2.0 Tutorial

Kevin Long

Computational Science and Mathematics Research Department
Sandia National Laboratories
Livermore, CA

March 21, 2005

Contents

1	Getting Started	6
1.1	About the Tutorial	6
1.2	Installation	7
1.2.1	Third-Party Software	7
1.2.2	Configuring and Building Trilinos for Sundance	7
1.2.3	Configuring Sundance	9
1.2.4	Setting up makefiles to develop a Sundance application	9
1.3	About the code and the documentation	10
1.3.1	Typographical conventions in the source code and examples	10
1.3.2	Handles and polymorphism in Sundance	10
1.3.3	Memory management	11
1.3.4	Parallelism	12
1.4	Using Sundance effectively	12
2	Examples	13
2.1	Example: Potential Flow Past an Elliptical Post	13
2.1.1	Step-by-step explanation	15
2.1.2	Complete Code for the Potential Flow Problem	20
2.2	Example: Radiation Diffusion on a Line	23
2.2.1	Step-by-Step Explanation	23
2.3	Example: Navier-Stokes Flow in a Lid-Driven Cavity	25
2.3.1	Step-by-Step Explanation	26
Appendix		
A	Controlling Diagnostic Output	31
	Verbosity Control In the Settings File	31
	Verbosity Control in Source Code	31
B	Troubleshooting	33
	Tracing with a Debugger	33
C	Meshing with Cubit for Sundance	35
	Simplicial Meshes in CUBIT	35

Labeling Regions	35
Generating NetCDF Files	35
Example: Generating the Wind Tunnel & Post Mesh	35
Example: Generating the Square Cavity Mesh	36
D Solver Parameters	39
Linear Solver Parameters	39
Nonlinear Solver Parameters	39
E Websites for Related Software	41

Chapter 1

Getting Started

Sundance is a system of software components that allows construction of an entire parallel simulator and its derivatives using a high-level symbolic language. With this high-level problem description, it is possible to specify a weak formulation of a PDE and its discretization method in a small amount of user-level code; furthermore, because derivatives are easily available, a simulation in Sundance is immediately suitable for accelerated PDE-constrained optimization algorithms[9].

Version 2.0 of Sundance represents a major internal redesign from version 1.0; however, most user-level syntax is unchanged. The significant advances over version 1.0 include:

- Use of a new algorithm, automatic functional differentiation[5] for efficient, scalable evaluation of integrands appearing in weak forms.
- Use of an abstract mesh interface allowing interoperability with third-party meshes.
- Use of the TSFCore[1]/TSFExtended[6] interfaces for solvers.
- A design allowing use of Sundance symbolic weak forms in other FE frameworks.

This tutorial will not explore use of all new features. In particular, all examples use a particular FE framework called the Sundance **standard framework**, and all use a particular concrete mesh class called the `BasicSimplicialMesh`.

1.1 About the Tutorial

In this tutorial we explain how to install Sundance, and then present several examples of how to set up and solve linear and nonlinear forward PDEs. In the Appendices we describe some rudimentary diagnostic and debugging utilities. For a complete survey of Sundance features and for examples of more advanced capabilities such as PDE-constrained optimization, see the Sundance User's Guide.

The examples here use TSF for linear solves, and TSF-NOX for nonlinear solves. We do not present a detailed description of the configuration options for these solvers; for that you are referred to the TSF and NOX documentation.

1.2 Installation

1.2.1 Third-Party Software

Sundance depends on certain other software libraries for solvers, parallel communication, and various other utilities.

1.2.1.1 Required Software

In order to use Sundance, you **must** first have

- **Trilinos** – Sandia’s high-performance parallel solver components[4], freely available from <http://software.sandia.gov/Trilinos/>.

Note that many popular solver libraries, such as Petsc, UMFPACK, and SuperLU, are available through Trilinos interfaces. To see how to enable such third-party solvers, see the Trilinos documentation.

Trilinos in turn requires

- An implementation of the **BLAS** (Basic Linear Algebra Subroutines). A reference BLAS is available from www.netlib.org, and many vendors supply optimized BLAS libraries.
- **LAPACK** A reference LAPACK is available from www.netlib.org.

1.2.1.2 Optional Software

The following software packages are optional, and need be installed only if certain features are desired.

- To enable parallel Sundance, you must install an implementation of **MPI**. In this case, Trilinos should be built with MPI enabled; see the Trilinos documentation for information on how to do this.

1.2.2 Configuring and Building Trilinos for Sundance

The installation procedure for Trilinos is well-documented at

http://software.sandia.gov/Trilinos/installation_manual.html

so here we merely provide a summary of the settings relevant to working with Sundance. The following flags **must** be set when building Trilinos for use with Sundance:

```
--enable-tsfbextended \  
--enable-tsfbcore \  
--enable-tsfbcore-epetra \  
--enable-tsfbcore-mpi \  
--enable-tsfbcore-aztecoo \  

```

```
--enable-teuchos \  
--enable-nox \  
--enable-nox-tsf
```

Several other optional flags impact the behavior of Sundance:

- `--enable-ml` turns on support for the ML algebraic multilevel solvers and preconditioners
- `--enable-expat` turns on support for the Expat XML parser, allowing parameters to be read from XML files
- `--enable-teuchos-abc` turns on array bounds checking in Teuchos, which will check every access to Teuchos Array objects for bounds errors. This is very useful for debugging, however, it causes a significant performance loss (about a factor of 3 for system assembly).

The Trilinos distribution comes with two sample scripts for configuring Trilinos for use with Sundance:

- `Trilinos/sampleScripts/sundance-linux-mpi-debug` builds a parallel Trilinos on Linux with no compiler optimization, and with debugging information and array bounds checking turned on.
- `Trilinos/sampleScripts/sundance-linux-mpi-opt` builds a parallel Trilinos on Linux with aggressive compile-time optimization and with debugging information and array bounds checking turned off.

1.2.2.1 Example Trilinos Installation

Here we show an example of installing Trilinos. We will configure it for use with Sundance with settings appropriate to debugging.

It is assumed that you have installed BLAS, LAPACK, and any optional libraries such as MPI. It is assumed you have downloaded Trilinos and unpacked the source into some directory; in this example, we will work in a Trilinos directory `$HOME/Projects/Trilinos`. As is explained in the Trilinos installation guide, Trilinos should always be configured and built in a build directory, **not** in the Trilinos source directory. In this example, we work in a build directory `$HOME/Projects/Trilinos/LINUX-DEBUG`.

The installation process is listed below.

```
<kevin@rusalka:Projects/Trilinos> mkdir LINUX-DEBUG  
<kevin@rusalka:Projects/Trilinos> cd LINUX-DEBUG  
<kevin@rusalka:Projects/Trilinos/LINUX-DEBUG> ../sampleScripts/sundance-linux-mpi-debug  
<kevin@rusalka:Projects/Trilinos/LINUX-DEBUG> make  
<kevin@rusalka:Projects/Trilinos/LINUX-DEBUG> make install
```

The final step puts libraries and headers into the installation directory specified in the configuration script; the sample script used here installs into the build directory. Thus in this example, headers and libraries are found in the `include` and `lib` subdirectories of `$HOME/Projects/Trilinos/LINUX-DEBUG`. The installation directory can be changed with the `--prefix` configure option.

1.2.2.2 Note on Concurrent Development of Trilinos and Sundance

The default installation behavior updates timestamps on header files so that if any Trilinos header file is changed, many Sundance files will have to be rebuilt. This is inconvenient for those people who use Sundance while doing

Trilinos development, for instance solver researchers using Sundance as a testbed, because it forces many unnecessary rebuilds of the entire Sundance distribution. On some systems, timestamps on unchanged files can be preserved through installation, so that only those Sundance files dependent on those Trilinos files that have actually changed need be rebuilt. On Linux, timestamps can be preserved with the `-p` option to `install`, which can be imposed by adding the option `--with-install="/usr/bin/install -c -p"` to the configure command line. This is done in the sample scripts in the Trilinos distribution.

1.2.3 Configuring Sundance

Once Trilinos is built and installed, we can configure, build, and install Sundance. During the Sundance build process, `configure` and `make` must find out where to find headers and libraries for Trilinos. This is done through the `--with-trilinos` option to `configure`. With the example Trilinos installation above, you would add `--with-trilinos=$HOME/Projects/Trilinos/LINUX-DEBUG` to the `configure` command line. If you have installed Trilinos somewhere else, you should change your `configure` line accordingly.

There is a sample configuration script in the `build-scripts` subdirectory of Sundance. This assumes Trilinos is installed in `$HOME/Projects/Trilinos/LINUX-DEBUG`, and that you are building with MPI. This script can be used as a starting point for other modes of building Sundance.

An example installation process is listed below.

```
<kevin@rusalka:Projects/Sundance> mkdir LINUX-DEBUG
<kevin@rusalka:Projects/Sundance> cd LINUX-DEBUG
<kevin@rusalka:Projects/Sundance/LINUX-DEBUG> ../build-scripts/linux-mpi-debug
<kevin@rusalka:Projects/Sundance/LINUX-DEBUG> make
<kevin@rusalka:Projects/Sundance/LINUX-DEBUG> make install
```

1.2.4 Setting up makefiles to develop a Sundance application

The previous installation steps have configured and built the Sundance libraries, tests, and examples. You are probably interested in using Sundance to develop your own application, for which you will need to write a Makefile that can access the Sundance libraries.

1.2.4.1 A simple application makefile example

A template for a simple application makefile is in `Makefile.sample`, found in the root directory of the Sundance distribution. For many Sundance-based applications it will be sufficient to copy this makefile and set one variable: `SUNDANCE_INSTALL_DIR`, which should point to the directory where you have installed Sundance.

In the following example, you are building the `PostPotentialFlow` simulator (described in detail below) in the directory `MyApps`. First put the application source (and any other required files such as meshes) in your application directory:

```
<kevin@rusalka:MyApps> cp ~/Projects/Sundance/examples-tutorial/PostPotentialFlow.cpp .
<kevin@rusalka:MyApps> cp ~/Projects/Sundance/examples-tutorial/post.ncdf .
<kevin@rusalka:MyApps> cp ~/Projects/Sundance/Makefile.sample Makefile
```

Now, edit `Makefile` and set `SUNDANCE_INSTALL_DIR` to the location of your Sundance build. The next step is to build the application:

```
<kevin@rusalka:MyApps> make PostPotentialFlow.exe
```

Your application is now ready to run.

If your application requires more than one source file, additional objects to be compiled and linked can be specified in the `EXTRA_OBJS` macro. For more complicated application structures, see the next section.

1.2.4.2 Rolling your own application Makefile

If the sample makefile described above is not sufficient for your application, you can access the `make` macros for Sundance directly by including in your makefile the file `Makefile.export`, which is written into your build subdirectory at configure time. This makefile fragment contains definitions of the Sundance and Trilinos header and library search paths, compiler options, and so on. If you are rolling your own Makefile, I assume you know enough about makefiles to set it up without further instruction from me.

1.3 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 basics of object-oriented programming, but need not be an expert C++ designer. You will need to know how to use objects, 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 developer's documentation.

1.3.1 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`, `DiscreteFunction`. method 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_`

1.3.2 Handles and polymorphism 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 user-level work with objects in a heirachy 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);
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`.

1.3.3 Memory management

Sundance has been designed so that memory management is transparent to the user; that is, the user should never have to worry about deleting memory that has been allocated. With the exception of new pointers that are immediately passed to handles, user-level code is normally entirely free of pointers. 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" 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.

1.3.4 Parallelism

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.

1.4 Using Sundance effectively

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.

Chapter 2

Examples

In this chapter we show how to use Sundance to solve several linear and nonlinear PDEs. Features are discussed as they arise in the problems. For a more systematic survey of features and syntax, see the User's Guide. In each of the examples that follow, we show how to set up a problem by defining the mesh, the locations where BCs are to be applied, all variables and operators, and the equation set and BCs. We then show configuration of a solver (linear or nonlinear depending on the problem), solution of the problem, and output.

Source code as well as associated files for meshes and solver parameters for all of these examples can be found in the subdirectory

`examples-tutorial`

of the Sundance distribution.

2.1 Example: Potential Flow Past an Elliptical Post

We'll begin with what is perhaps the simplest PDE to solve numerically: Laplace's equation $\nabla^2 u = 0$. Though simple, we can illustrate much of Sundance syntax with the problem. The Laplace equation is ubiquitous in applications (see, e.g, [2]), one of which is incompressible, irrotational flow. Incompressibility and irrotationality require that the velocity field \mathbf{u} satisfy

$$\nabla \cdot \mathbf{u} = 0 \tag{2.1}$$

$$\nabla \times \mathbf{u} = 0, \tag{2.2}$$

from which it follows that the velocity is the gradient of a scalar potential

$$\mathbf{u} = \nabla \phi \tag{2.3}$$

and

$$-\nabla^2 \phi = 0. \tag{2.4}$$

We have used a negative sign above in order to get a positive definite operator.

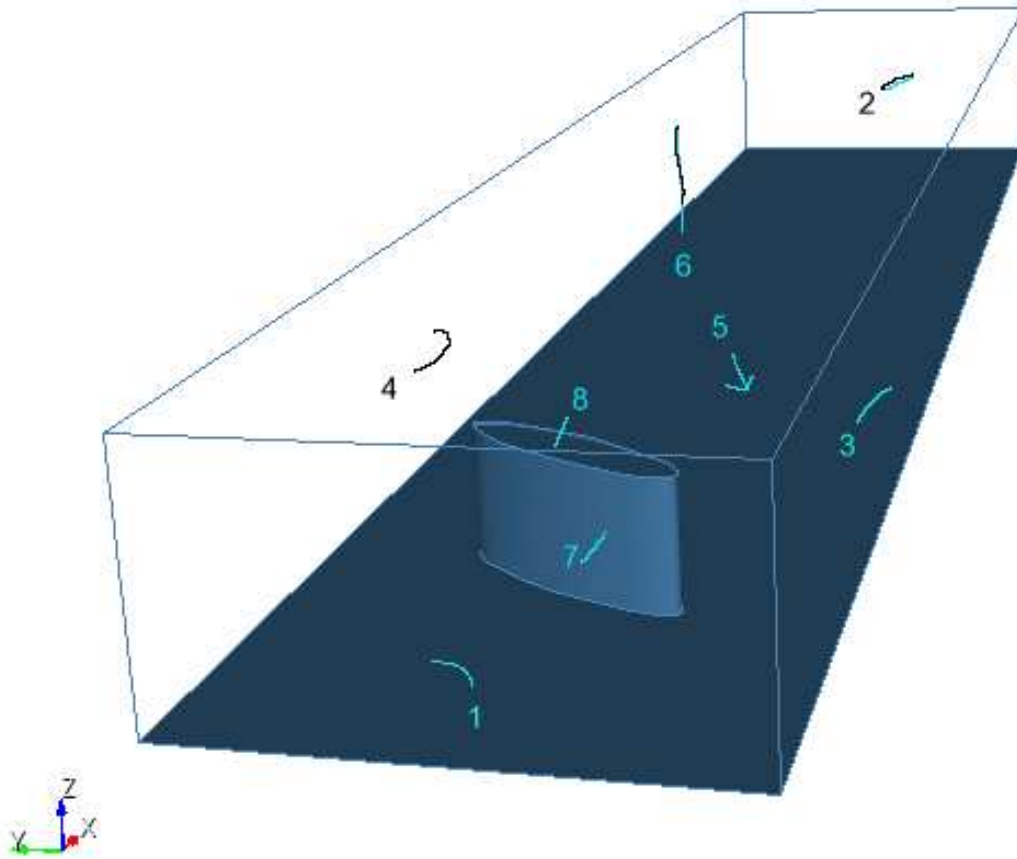


Figure 2.1. Domain for potential flow example. The numbers correspond to the side set labels.

We will consider flow past the post shown in figure 2.1. We imagine the post placed in a wind tunnel with impenetrable walls and a constant inlet potential ϕ_0 which we can choose to be zero. We also need a downstream boundary condition. The correct downstream boundary condition is constant velocity at infinity; a reasonable approximation usable in a finite domain is to assume the potential increases linearly towards some constant potential ϕ_{out} some distance L downstream from the outlet.¹ We therefore have BCs

$$\phi = 0 \text{ at inlet,} \quad (2.5)$$

$$\hat{\mathbf{n}} \cdot \nabla \phi = 0 \text{ at walls,} \quad (2.6)$$

$$\hat{\mathbf{n}} \cdot \nabla \phi = \frac{\phi_{out} - \phi}{L} \text{ at outlet.} \quad (2.7)$$

$$(2.8)$$

With the equation and BCs in hand, we can write the potential flow problem in weak form. Multiplying by a test function $\hat{\phi}$ and integrating by parts, we have

$$\int_{\text{interior}} \nabla \hat{\phi} \cdot \nabla \phi - \int_{\text{boundary}} \hat{\phi} \nabla \phi \cdot \hat{\mathbf{n}} = 0. \quad (2.9)$$

We now use the BCs to rewrite the boundary term. On the tunnel walls, $\hat{\mathbf{n}} \cdot \nabla \phi = 0$, so the boundary term vanishes. Applying the BC at the outlet and ignoring for the moment the inlet BC, we have

$$\int_{\text{interior}} \nabla \hat{\phi} \cdot \nabla \phi - \int_{\text{outlet}} \frac{1}{L} \hat{\phi} (\phi_{out} - \phi) - \int_{\text{inlet}} \hat{\phi} \nabla \phi \cdot \hat{\mathbf{n}} = 0. \quad (2.10)$$

At the inlet, we impose the Dirichlet BC $\phi = 0$ instead of the PDE. The Dirichlet BC can be written weakly as

$$\int_{\text{inlet}} \hat{\phi} \phi = 0 \quad (2.11)$$

for all test functions $\hat{\phi}$ that are nonzero on the inlet. We therefore have two cases

$$\begin{cases} \int_{\text{interior}} \nabla \hat{\phi} \cdot \nabla \phi - \int_{\text{outlet}} \frac{1}{L} \hat{\phi} (\phi_{out} - \phi) = 0 & \forall \hat{\phi} \text{ such that } \hat{\phi} = 0 \text{ on inlet} \\ \int_{\text{inlet}} \hat{\phi} \phi = 0 & \forall \hat{\phi} \text{ such that } \hat{\phi} \neq 0 \text{ on inlet} \end{cases} \quad (2.12)$$

It will be shown below how these two cases are distinguished in Sundance.

2.1.1 Step-by-step explanation

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

2.1.1.1 Boilerplate

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

¹It is reasonable to use such a BC for the inlet as well; however, we use the Dirichlet BC $\phi = 0$ here in order to show how to set up Dirichlet BCs in Sundance.

```

#include "Sundance.hpp"

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

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

```

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

2.1.1.2 Getting the mesh

Sundance uses a `Mesh` object to represent a tessellation of the problem domain. There are many ways of getting a mesh, all abstracted with the `MeshSource` interface. Sundance is designed to work with different mesh underlying implementations, the choice of which is done by specifying a `MeshType` object. In this example, we use the `BasicSimplicialMeshType` which is a lightweight parallel simplicial mesh. The mesh was created using `Cubit` (<http://cubit.sandia.gov>) and saved to a `NetCDF` file. We make the Sundance mesh object by reading this file:

```

MeshType meshType = new BasicSimplicialMeshType();
MeshSource meshReader
    = new ExodusNetCDFMeshReader("../examples-tutorial/post.ncdf",
                                meshType);
Mesh mesh = mesher.getMesh();

```

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 `MeshType` object which is – apparently – not a pointer. That’s not a typo: the `MeshType` object is a **handle** class that stores and manages the pointer to the `BasicSimplicialMeshType` object. Handle classes are used throughout user-level Sundance code, and among other things relieve you of the need to worry about memory management.

2.1.1.3 Defining the domains of integration

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 `CellFilter` object to represent subregions of a geometric domain. A `CellFilter` can be any collection of mesh cells, for example a block of maximal cells, a set of boundary edges, or a set of points.

We need do nothing for the wall BCs because they are natural and drop out of the weak form. We need only identify the interior, the inlet, and the outlet. We first create a cell filter object for the entire domain,

```
CellFilter interior = new MaximalCellFilter();
```

which will identify all maximal cells. We next create a cell filter to identify the boundary cells,

```
CellFilter boundary = new BoundaryCellFilter();
```

and then we find the subsets of the boundary corresponding to the inlet and outlet. These have been labeled by Cubit as “1” and “2,” respectively, so we can grab the subsets having those labels:

```
CellFilter in = boundary.labeledSubset(1);  
CellFilter out = boundary.labeledSubset(2);
```

2.1.1.4 Defining unknown and test functions

We’ll use first order piecewise Lagrange interpolation to represent our unknown solution ϕ . With a Galerkin method we use a test function $\hat{\phi}$ defined using the same basis as the unknown. Expressions representing the test and unknown functions are defined easily:

```
Expr phiHat = new TestFunction(new Lagrange(1));  
Expr phi = new UnknownFunction(new Lagrange(1));
```

2.1.1.5 Creating the gradient operator

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

```
Expr dx = new Derivative(0);  
Expr dy = new Derivative(1);  
Expr dz = new Derivative(2);  
Expr grad = List(dx, dy, dz);
```

Notice that we always number directions (and all other indices) starting from zero as in C, C++, and Java, as opposed to from one as in Fortran and Matlab. 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 $\{dx*u, dy*u, dz*u\}$.

2.1.1.6 Writing the weak form

We will need to define expressions for L and ϕ_{out} . Such constant expressions are defined easily as

```
Expr L = 10.0;  
Expr phiOut = 1.0;
```

The weak form for the interior, walls, and outlet is

$$\int_{\text{interior}} \nabla \hat{\phi} \cdot \nabla \phi - \int_{\text{outlet}} \frac{1}{L} \hat{\phi} (\phi_{out} - \phi) = 0. \quad (2.13)$$

All of these integrations can be done exactly so there is no need to specify a quadrature rule, however, there is no harm in doing so; if an exactly integrable term is detected any quadrature rule will be ignored. For generality in the event we later wish to change one of the coefficients to something non-constant, we explicitly specify a quadrature rule. We'll use second-order Gaussian quadrature. The weak form with a quadrature specification is written in Sundance as

```
QuadratureFamily quad2 = new GaussianQuadrature(2);

Expr eqn = Integral(interior, (grad*phiHat)*(grad*phi))
    + Integral(out, alpha*phiHat*(phiOut-phi)/L);
```

2.1.1.7 Writing the essential BCs

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

```
Expr bc = EssentialBC(inlet, phiHat*phi);
```

2.1.1.8 Creating the linear problem object

A `LinearProblem` 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.

We will use `Epetra` as our linear algebra representation, which is specified by selecting the corresponding `VectorType` subtype,

```
VectorType<double> vecType = new EpetraVectorType();
```

We can now create a problem object

```
LinearProblem prob(mesh, poisson, bc, phiHat, phi, petra);
```

It may seem unnecessary to provide `phiHat` and `phi` 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.

2.1.1.9 Specifying the solver

A good choice of solver for this problem is BICGSTAB with ILU preconditioning.² We'll use level 1 preconditioning, and ask for a convergence tolerance of 10^{-14} within 1000 iterations. The TSF BICGSTAB solver is configured using

²The Laplacian operator is symmetric positive definite, however, the imposition of essential BCs destroys the symmetry making conjugate gradients unsuitable. Symmetry may be restored by doing block manipulations or by using Robin BCs as an approximation to Dirichlet BCs.

a Trilinos `ParameterList` object, read in from an XML file

```
/* Read the parameters for the linear solver from an XML file */
ParameterXMLFileReader reader("../examples-tutorial/bicgstab.xml");
ParameterList solverParams = reader.getParameters();

LinearSolver<double> linSolver
    = LinearSolverBuilder::createSolver(solverParams);
```

The contents of this file are

```
LinearProblem prob(mesh, poisson, bc, phiHat, phi, petra);
```

2.1.1.10 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`.

2.1.1.11 Computing Velocity Fields

We have computed the potential ϕ , however, we will often want to obtain the velocity field $\mathbf{u} = \nabla\phi$. The gradient of the solution ϕ is not a continuous function and its components do not exist in the same function space as used for ϕ . However, we can project the velocity field onto any vector space by solving the least-squares problem of minimizing the L^2 norm of the difference between the velocity field and its projection onto a discrete space. In Sundance, this is accomplished as follows:

```
/* Project the velocity onto a discrete space so we can visualize it */
DiscreteSpace discreteSpace(mesh,
    List(new Lagrange(1),
        new Lagrange(1),
        new Lagrange(1)),
    vecType);
L2Projector projector(discreteSpace, grad*soln);
Expr velocity = projector.project();
```

We now have the velocity field as a discrete function.

Note 2.1.1 *The derivatives of a C^0 function are not defined at element boundaries. Derivatives of C^0 functions can safely be used within weak forms because values on element boundaries are not required. However, in cases such as visualization where a derivative of a C^0 discrete function is to be used in such a way that its nodal values are required, it is essential to project the derivative onto a discrete space.*

2.1.1.12 Viewing the solution

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

```
FieldWriter w = new VTKWriter("Post3d");
w.addMesh(mesh);
w.addField("phi", new ExprFieldWrapper(soln[0]));
w.addField("ux", new ExprFieldWrapper(velocity[0]));
w.addField("uy", new ExprFieldWrapper(velocity[1]));
w.addField("uz", new ExprFieldWrapper(velocity[2]));

w.write();
```

The resulting VTK file, `Post3d.vtu`, can then be visualized with any VTK viewer such as Paraview. A plot of potential and particle streamlines is shown in figure 2.2.

2.1.2 Complete Code for the Potential Flow Problem

```
#include "Sundance.hpp"

/**
 * Solves the Laplace equation for potential flow past an elliptical
 * post in a wind tunnel.
 */

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

        /* We will do our linear algebra using Epetra */
        VectorType<double> vecType = new EpetraVectorType();

        /* Create a mesh. It will be of type BasisSimplicialMesh, and will
         * be built using a PartitionedRectangleMesher. */
        MeshType meshType = new BasicSimplicialMeshType();

        MeshSource mesher
            = new ExodusNetCDFMeshReader("../examples-tutorial/post.ncdf", meshType);
        Mesh mesh = mesher.getMesh();

        /* Create a cell filter that will identify the maximal cells
         * in the interior of the domain */
        CellFilter interior = new MaximalCellFilter();
        CellFilter boundary = new BoundaryCellFilter();
        CellFilter in = boundary.labeledSubset(1);
        CellFilter out = boundary.labeledSubset(2);

        /* Create unknown and test functions, discretized using first-order
```

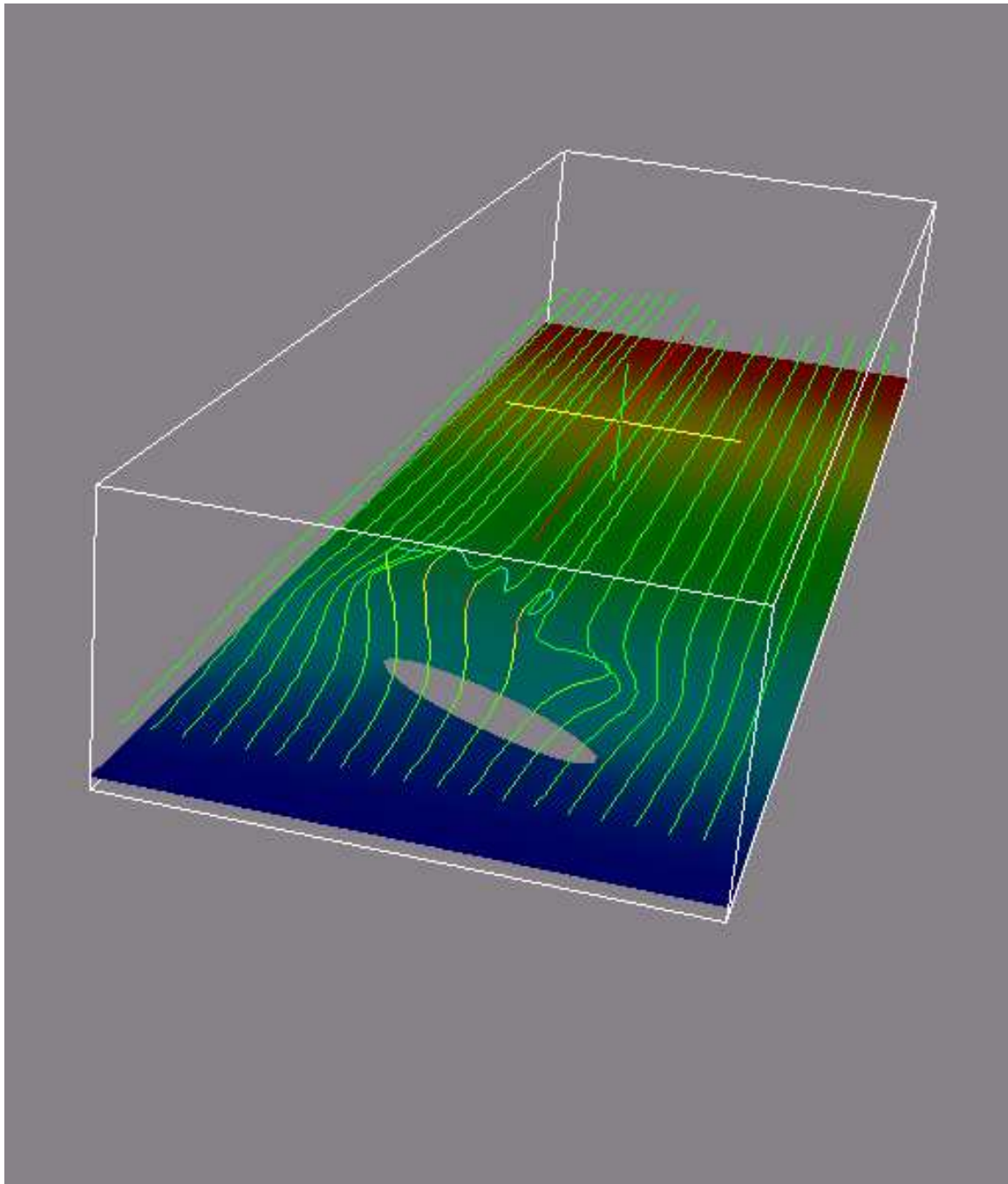


Figure 2.2. Solution for potential flow past a post in a wind tunnel. The coloring of the planar slice near the bottom of the domain corresponds to velocity potential. The curves above are streamlines; the coloring along the streamline indicates the z component of velocity.

```

    * Lagrange interpolants */
Expr phi = new UnknownFunction(new Lagrange(1), "u");
Expr phiHat = new TestFunction(new Lagrange(1), "v");

/* Create differential operator and coordinate functions */
Expr x = new CoordExpr(0);
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr dz = new Derivative(2);
Expr grad = List(dx, dy, dz);

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

double L = 1.0;
/* Define the weak form */
Expr eqn = Integral(interior, (grad*phiHat)*(grad*phi), quad2)
    + Integral(in, phiHat*(x-phi)/L, quad2) ;

/* Define the Dirichlet BC */
Expr bc = EssentialBC(out, phiHat*phi/L, quad2);

/* We can now set up the linear problem! */
LinearProblem prob(mesh, eqn, bc, phiHat, phi, vecType);

/* Read the parameters for the linear solver from an XML file */
ParameterXMLFileReader reader("../../examples-tutorial/bicgstab.xml");
ParameterList solverParams = reader.getParameters();

LinearSolver<double> linSolver
    = LinearSolverBuilder::createSolver(solverParams);

/* solve the problem */
Expr soln = prob.solve(linSolver);

/* Project the velocity onto a discrete space so we can visualize it */
DiscreteSpace discreteSpace(mesh,
    List(new Lagrange(1),
        new Lagrange(1),
        new Lagrange(1)),
    vecType);
L2Projector projector(discreteSpace, grad*soln);
Expr velocity = projector.project();

/* Write the field in VTK format */
FieldWriter w = new VTKWriter("Post3d");
w.addMesh(mesh);
w.addField("phi", new ExprFieldWrapper(soln[0]));
w.addField("ux", new ExprFieldWrapper(velocity[0]));
w.addField("uy", new ExprFieldWrapper(velocity[1]));
w.addField("uz", new ExprFieldWrapper(velocity[2]));

```



```

        w.write();

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

```

2.2 Example: Radiation Diffusion on a Line

With this example we show how to set up and solve a nonlinear PDE in Sundance.

The optical mean free path of a material is the mean distance a photon can travel before being either scattered or absorbed and reemitted. In the limit of small mean free path and in local thermodynamic equilibrium, the temperature $T(\mathbf{x})$ is given by a nonlinear diffusion equation [8],

$$\frac{\partial T}{\partial t} = s(\mathbf{x}) + \nabla \cdot K \nabla T^4 \quad (2.14)$$

In steady state and one spatial dimension with constant diffusivity K and no sources, we have

$$\frac{\partial^2}{\partial x^2} T^4 = 0 \quad (2.15)$$

which clearly has the solution

$$T(x)^4 = Ax + B \quad (2.16)$$

where A and B are to be determined by boundary conditions. Taking $T(0) = 1$ and $T(1) = 2$, we have $B = 1$ and $A = 15$, so that

$$T(x) = [15x + 1]^{1/4} \quad (2.17)$$

On physical grounds, we must restrict ourselves to the positive fourth root.

2.2.1 Step-by-Step Explanation

We begin by creating a uniform one-dimensional mesh for the problem domain

```

int nx = 10;
MeshType meshType = new BasicSimplicialMeshType();
MeshSource mesher = new PartitionedLineMesher(0.0, 1.0, nx, meshType);
Mesh mesh = mesher.getMesh();

```

The test and unknown functions are created as before,

```

/* Create test and unknown functions */
Expr u = new UnknownFunction(new Lagrange(1), "u");
Expr v = new TestFunction(new Lagrange(1), "v");
/* Create differential operator */
Expr dx = new Derivative(0);

```

2.2.1.1 Setup of the Equations

Because of the presence of the nonlinear term, the weak form must be integrated by quadrature.

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

The weak form and BCs are then

```
/* Define the weak form */
Expr eqn = Integral(interior, u*u*u*(dx*v)*(dx*u), quad);
/* Define the Dirichlet BC */
Expr bc = EssentialBC(leftPoint, v*(u-1.0), quad)
          + EssentialBC(rightPoint, v*(u-2.0), quad);
```

2.2.1.2 Specification of the Initial Guess

The radiation diffusion equation is nonlinear and must be solved by an iterative nonlinear solver such as Newton's method. We will need an initial guess for the nonlinear solver, which must be represented in Sundance as a discrete function, discretized with the same basis as the unknown function.

A fair initial guess is $u_0 = 1 + x$. We can create this expression in continuous form using the coordinate expression x , defined as follows:

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

The initial guess must be discretized; one way of doing this is by L^2 projection onto a discrete space, accomplished in Sundance by

```
/* We will do our linear algebra using Epetra */
VectorType<double> vecType = new EpetraVectorType();

/* Create a discrete space, and project the function 1.0+x onto it */
DiscreteSpace discSpace(mesh, new Lagrange(1), vecType);
Expr u0 = L2Project(discreteSpace, 1.0+x);
```

2.2.1.3 Specification of the Nonlinear Problem

We will solve the equation using a NOX nonlinear solver, which expects a TSF nonlinear operator. We create this operator as follows:

```
NonlinearOperator<double> F = new NonlinearProblem(mesh, eqn, bc, v, u, u0, vecType);
```

2.2.1.4 Solution via NOX

It remains to configure and use the NOX solver. As with the TSF linear solver in example 2.1, NOX objects are constructed using Trilinos `ParameterList` objects, which can be built up on the fly in source code or read from a

file. We will most commonly read solver parameters from an XML file, in which case we can construct the solver with the following code:

```
ParameterXMLFileReader reader("../examples-tutorial/nox.xml");
ParameterList noxParams = reader.getParameters();

NOXSolver solver(noxParams, F);
```

With the solver configured, we can solve:

```
// Solve the nonlinear system
NOX::StatusTest::StatusType status = solver.solve();
```

2.3 Example: Navier-Stokes Flow in a Lid-Driven Cavity

In the vorticity-streamfunction (VS) formulation of the incompressible Navier-Stokes equation we write the velocity field as the curl of a streamfunction $\mathbf{u} = \nabla \times \psi$, and introduce the vorticity $\Omega = \nabla \times \mathbf{u}$ as an auxiliary variable. Working in these variables, the incompressibility constraint is satisfied automatically and we can eliminate pressure from the Navier-Stokes equations[3]. In general, it is difficult to write boundary conditions for the VS formulation; the VS formulation is most useful in two-dimensional problems on simply-connected domains.

In two dimensions the VS form of the Navier-Stokes momentum equation is

$$Re \left[\frac{\partial \psi}{\partial y} \frac{\partial \Omega}{\partial x} - \frac{\partial \psi}{\partial x} \frac{\partial \Omega}{\partial y} \right] = \nabla^2 \Omega. \quad (2.18)$$

This equation is supplemented by

$$\nabla^2 \psi + \Omega = 0, \quad (2.19)$$

a purely mathematical identity which follows from the definitions of Ω and ψ .

The “lid-driven cavity” is a standard benchmark problem in computational fluid dynamics[7]. The problem domain is a square box with three fixed no-slip walls and a top wall, or lid, that drags the fluid along at velocity u_{lid} . No body forces are present.

Specifying both components of velocity on the boundary is equivalent to specifying normal and tangential derivatives of the streamfunction,

$$u_x = \frac{\partial \psi}{\partial y} \quad (2.20)$$

and

$$u_y = -\frac{\partial \psi}{\partial x}. \quad (2.21)$$

In the cavity problem the normal velocity is zero everywhere on the boundary, which in turn implies constant streamfunction on the entire boundary. The value we choose for this constant is immaterial because shifting the streamfunction by a constant does not change the velocity. Therefore we can take $\psi = 0$ as one of our boundary conditions. For the tangential velocity, we have $\frac{\partial \psi}{\partial n} = 0$ on the fixed walls and $\frac{\partial \psi}{\partial y} = -u_0$ on the lid. There is no boundary condition for the vorticity. This unusual set of boundary conditions, two on one field and none on the other, is easily handled in Sundance notation.

2.3.1 Step-by-Step Explanation

2.3.1.1 Geometry Specification

We'll read a mesh in NetCDF format. The sides are labeled "1," "2," "3," and "4," for the bottom, right, top, and left, respectively.

```
/* Create a mesh. It will be of type BasisSimplicialMesh, and will
 * be built using a PartitionedRectangleMesher. */
MeshType meshType = new BasicSimplicialMeshType();
MeshSource meshReader = new TriangleMeshReader("../.../Meshes/square.ncdf", meshType);
Mesh mesh = mesher.getMesh();

CellFilter interior = new MaximalCellFilter();
CellFilter edges = new DimensionalCellFilter(1);

CellFilter bottom = edges.labeledSubset(1);
CellFilter right = edges.labeledSubset(2);
CellFilter top = edges.labeledSubset(3);
CellFilter left = edges.labeledSubset(4);
CellFilter walls = left+right+bottom;
CellFilter lid = top;
```

2.3.1.2 Equation and BC Specification

We discretize using first-order Lagrange elements for both fields (one advantage of the VS formulation is that equal-order discretizations are stable). All except the nonlinear term can be integrated exactly; for the nonlinear term, first-order integration will suffice with first-order basis functions.

```
/* Create unknown and test functions, discretized using first-order
 * Lagrange interpolants */
Expr psi = new UnknownFunction(new Lagrange(1), "psi");
Expr vPsi = new TestFunction(new Lagrange(1), "vPsi");
Expr omega = new UnknownFunction(new Lagrange(1), "omega");
Expr vOmega = new TestFunction(new Lagrange(1), "vOmega");

/* Create differential operator and coordinate functions */
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr grad = List(dx, dy);

/* A parameter expression for the Reynolds number */
Expr reynolds = new Parameter(20.0);

/* We need a quadrature rule for integrating the nonlinear terms */
QuadratureFamily quad1 = new GaussianQuadrature(1);

/* Define the weak form */
Expr psiEqn = Integral(interior, (grad*vPsi)*(grad*psi) + vPsi*omega)
  + Integral(lid, vPsi);
```

```

Expr omegaEqn = Integral(interior, (grad*vOmega)*(grad*omega))
    + Integral(reynolds*vOmega*((dy*psi)*dx*omega - (dx*psi)*dy*omega), quad1);

Expr eqn = omegaEqn + psiEqn;

/* Dirichlet BCs giving zero normal velocity */
Expr bc = EssentialBC(walls, vOmega*psi, quad2)
    + EssentialBC(top, vOmega*psi, quad2) ;

```

Notice that the Dirichlet boundary conditions on ψ have been associated with the vorticity test function; this lets us impose streamfunction BCs in place of the boundary terms involving vorticity.

Note 2.3.1 *We have defined the Reynolds number as a parameter, **not** as a double variable or constant-valued expression. In the continuation loop we will change the Reynolds number. Constant-valued expressions and parameter expressions differ in that the former are immutable whereas the latter can be changed through a call to `setParameterValue()`. Real-valued variables that will be changed from solve to solve, for instance in a continuation loop or in timestepping, must be represented as parameters rather than as constant-valued expressions.*

2.3.1.3 The Initial Guess

Creation of and expression for the initial guess is straightforward, the only change from the previous example being the need to define a vector-valued field rather than a scalar field. This is easily done by creating a discrete space with a vector of basis functions,

```

DiscreteSpace discSpace(mesh, List(new Lagrange(1), new Lagrange(1)), vecType);
Expr u0 = new DiscreteFunction(discSpace, List(1.0, 1.0), "u0");

```

Note 2.3.2 *In nonlinear vector problems, the initial guess must be a single vector-valued discrete function rather than a list of scalar functions. This is so that all fields share a single underlying vector object.*

2.3.1.4 The Nonlinear Problem

When constructing the nonlinear problem, the only issue is that we must be careful to choose a favorable ordering of the fields ψ and Ω . The linear equations for the Newton step converge most rapidly when the equations are ordered as $(\hat{\psi}, \hat{\Omega})$ and the unknowns are ordered as (ψ, Ω) . We can specify this through the ordering of the test and unknown functions in the `NonlinearProblem` constructor,

```

NonlinearOperator<double> F = new NonlinearProblem(mesh, eqn, bc,
    List(vPsi, vOmega),
    List(psi, omega), u0,
    vecType);

```

2.3.1.5 Configuring the Nonlinear Solver

We construct the nonlinear solver using parameters read from an XML file:

```

ParameterXMLFileReader reader("../..//examples-tutorial/nox.xml");

```

```

ParameterList noxParams = reader.getParameters();

NOXSolver solver(noxParams, F);

```

2.3.1.6 The Continuation Loop

The problem and solver are now set up. Convergence of Newton's method at high Reynolds number depends on a good initial guess, so we will proceed by continuation from a low Re to the desired final $Re = 500$. A simple continuation loop can be written directly in C++,

```

int numReynolds = 10;
double finalReynolds = 500.0;
for (int r=1; r<=numReynolds; r++)
{
    double Re = r*finalReynolds/((double) numReynolds);
    reynolds.setParameterValue(Re);
    cerr << "----- " << endl;
    cerr << " solving for Reynolds Number = " << reynolds << endl;
    cerr << "----- " << endl;
    // Solve the nonlinear system
    NOX::StatusTest::StatusType status = solver.solve();

    /* Write the field in VTK format */
    FieldWriter w = new VTKWriter("vns-r" + Teuchos::toString(Re));
    w.addMesh(mesh);
    w.addField("streamfunction", new ExprFieldWrapper(u0[0]));
    w.addField("vorticity", new ExprFieldWrapper(u0[1]));
    w.write();
}

```

Note that in the course of the loop we write the solution at each Reynolds number to a VTK file. The streamfunction for $Re = 500$ is shown in figure 2.3.

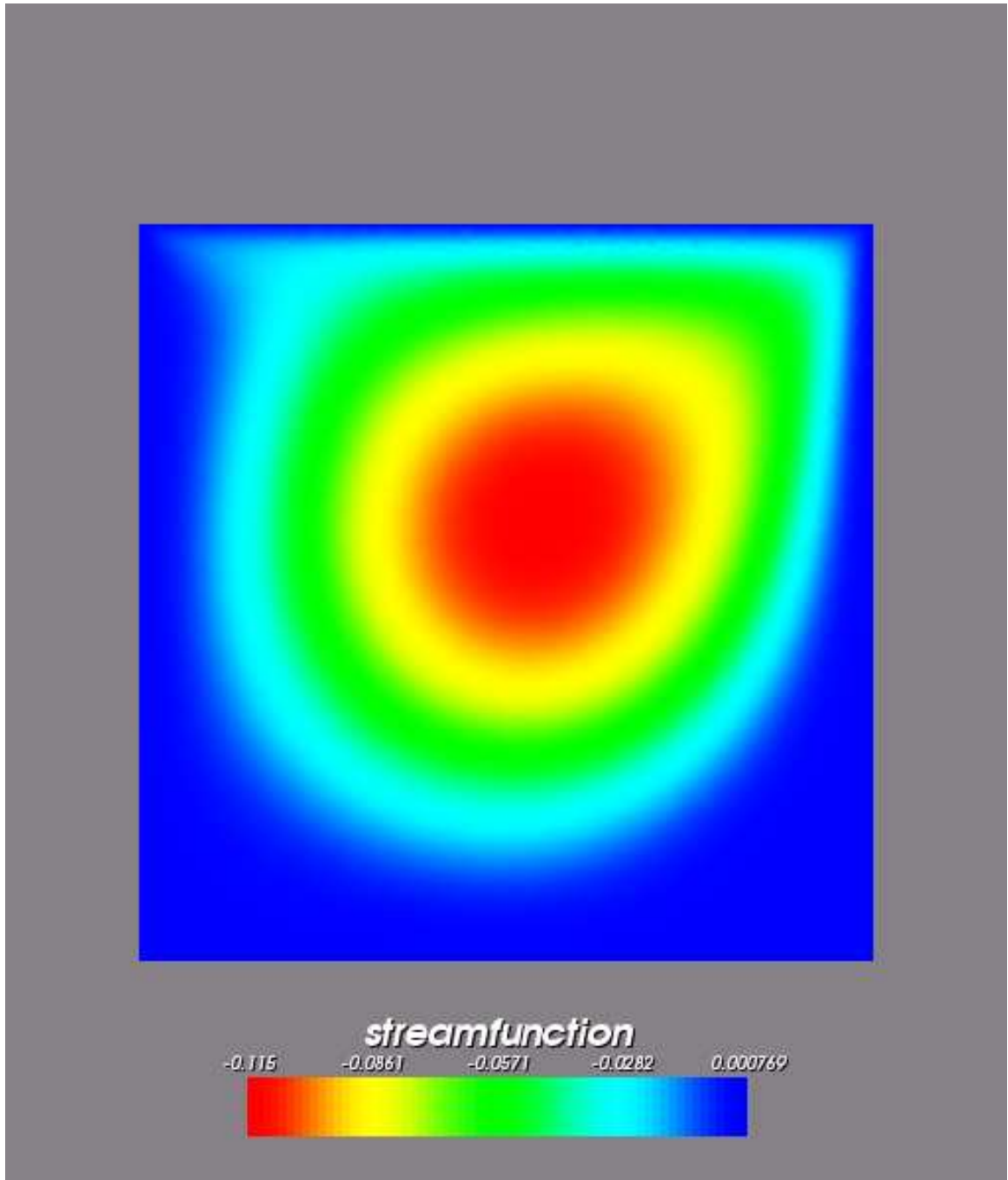


Figure 2.3. Streamfunction for flow in a lid-driven cavity at Reynolds number $Re = 500$.

A Controlling Diagnostic Output

Many Sundance classes, both user-level and internal, have user-controllable verbosity settings. Possible verbosity settings are

- `VerbSilent`
- `VerbLow`
- `VerbMedium`
- `VerbHigh`
- `VerbExtreme`

The meaning of `VerbSilent` is self-explanatory; the amount of output resulting from the other settings varies from class to class. Extreme verbosity will print *very much* output, so it is most effectively used on small problems. Verbosity settings can be specified in two ways: by function calls in source code, or by parameters in the configuration file.

Verbosity Control In the Settings File

Verbosity levels can be set in the configuration file (called `SundanceConfig.xml` by default). The format for verbosity settings is shown below:

```
<Verbosity context="Assembly" value="Silent"/>
<Verbosity context="Evaluation" value="Silent"/>
<Verbosity context="Quadrature" value="Silent"/>
<Verbosity context="Reference Integration" value="Silent"/>
<Verbosity context="DOF Mapping" value="Silent"/>
<Verbosity context="Eval Vector" value="Silent"/>
```

Verbosity Control in Source Code

Verbosity levels can also be set through function calls in the source code. Verbosity is set by calling the static `classVerbosity()` method, which returns a modifiable pointer to an internal verbosity setting. For example, to set the verbosity of the internal class `Assembler` to `VerbExtreme`, you would do

```
Assembler::classVerbosity() = VerbExtreme;
```

The following subsections list some internal classes that have useful verbosity settings.

Expression Evaluation

- `Evaluator` – traces evaluation of symbolic expressions. Extreme verbosity prints the result of evaluating every node in the expression graph.
- `EvalVector` – traces calls to vector operations encountered during evaluations.

Assembly

- `Assembler` – traces assembly of matrices and vectors. Extreme verbosity prints the matrix graph, every local matrix and vector, and then the global matrix and vector.
- `DOFMapBase` – traces construction and use of DOF numbering tables.
- `QuadratureIntegral` – traces integration by quadrature.
- `RefIntegral` – traces exact integration.
- `QuadratureEvalMediator` – traces evaluation of coordinate expressions and discrete functions at quadrature points.

Solvers

Verbosity of solvers is controlled by solver-specific flags rather than through Sundance. Refer to solver documentation for information on how to set how much information is printed out.

B Troubleshooting

Tracing with a Debugger

All Sundance runtime errors are filtered through the Teuchos `TEST_FOR_EXCEPTION` macro, which prints line number information and a descriptive message. It then calls the function

```
TestForException_break
```

Any runtime error detected by Sundance can then be traced in a debugger by setting a breakpoint at `TestForException_break`. In gdb-based debuggers such as DDD, this is done by entering

```
break TestForException_break
```

at the debugger's interactive prompt. For setting and using breakpoints in other debuggers such as Totalview, refer to the appropriate documentation.

C Meshing with Cubit for Sundance

Sundance gets meshes through the MeshSource interface, and so does not rely on a single mesher. We cannot provide extensive mesh documentation beyond referencing the mesher documentation, however, we can provide some tips for meshing for Sundance using Sandia's CUBIT mesh generation toolsuite. See the CUBIT web page <http://cubit.sandia.gov/> for information on licensing and for documentation.

Simplicial Meshes in CUBIT

Currently Sundance supports simplicial meshes only. By default, CUBIT generates quad and hex meshes, so you will need to tell CUBIT to use triangle or tet meshes.

The CUBIT command

```
volume <volume ID> scheme tetmesh+
```

will make subsequent meshing operations use the INRIA tetrahedral mesher.

There are a number of options for meshing 2D surfaces with triangles: an advancing front algorithm, a Delaunay triangulation algorithm, and conversion of a quad mesh to a triangle mesh. The scheme is specified using a command such as

```
surface <surface ID> scheme TriDelaunay
```

Labeling Regions

Side sets can be created and numbered in CUBIT. The side set number can then be used as a label to specify cell filters for defining BC regions.

Note C.1 *Older finite element codes often specify Dirichlet BCs on node sets. Because Sundance is intended to work with higher-order basis functions regardless of whether the mesh contains midside nodes, in Sundance codes we usually specify Dirichlet BCs on side sets.*

Generating NetCDF Files

CUBIT generates EXODUS files, however, the libraries to read EXODUS files are restricted release only. Therefore, we will usually convert CUBIT's EXODUS files to the text-based NetCDF format. A converter called `ncdump` is freely available from <http://www.unidata.ucar.edu>. The usage of `ncdump` is as follows.

```
<kevin@rusalka:Projects/Sundance/examples-tutorial> ncdump post.exo > post.ncdf
```

Example: Generating the Wind Tunnel & Post Mesh

The CUBIT commands for generating the wind tunnel and post mesh used in the examples are:

```

cylinder r 0.5 z 0.5
volume 1 scale y 0.25
volume 1 rotate 60 about z

brick x 6 y 2 z 1
volume 2 move z 0.25
volume 2 move x 2.0

subtract volume 1 from volume 2

volume 2 scheme tetmesh
volume 2 size 0.05
mesh volume 2

block 1 volume 2

sideset 1 surface 7
sideset 2 surface 9
sideset 3 surface 6
sideset 4 surface 8
sideset 5 surface 12
sideset 6 surface 4
sideset 7 surface 10
sideset 8 surface 11

export genesis 'post.exo' block 1

```

Example: Generating the Square Cavity Mesh

```

# mesh a square
#
# Bottom label 1
# Right label 2
# Top label 3
# Left label 4

create vertex x 0.0 y 0.0
create vertex x 1.0 y 0.0
create vertex x 1.0 y 1.0
create vertex x 0.0 y 1.0

create curve 1 2
create curve 2 3
create curve 3 4
create curve 4 1

create surface curve 1 2 3 4

```

```
surface 1 scheme tridelaunay
surface 1 size 0.01
mesh surface 1
```

```
block 1 surface 1
```

```
sideset 1 curve 1
sideset 2 curve 2
sideset 3 curve 3
sideset 4 curve 4
```

```
export genesis 'square.exo' block 1
```


D Solver Parameters

Linear Solver Parameters

The XML for the setup of the linear solver used in Example 2.1 is shown below

```
<ParameterList>
  <ParameterList name="Linear Solver">
    <Parameter name="Graph Fill" type="int" value="1"/>
    <Parameter name="Max Iterations" type="int" value="200"/>
    <Parameter name="Method" type="string" value="BICGSTAB"/>
    <Parameter name="Precond" type="string" value="ILUK"/>
    <Parameter name="Tolerance" type="double" value="1e-12"/>
    <Parameter name="Type" type="string" value="TSF"/>
    <Parameter name="Verbosity" type="int" value="4"/>
  </ParameterList>
</ParameterList>
```

Nonlinear Solver Parameters

The XML for the setup of the nonlinear solver used in Examples 2.2 and 2.3 is shown below

```
<ParameterList>
  <ParameterList name="NOX Solver">
    <Parameter name="Nonlinear Solver" type="string" value="Line Search Based"/>
    <ParameterList name="Line Search">
      <Parameter name="Method" type="string" value="More'-Thuente"/>
    </ParameterList>
    <ParameterList name="Status Test">
      <Parameter name="Max Iterations" type="int" value="20"/>
      <Parameter name="Tolerance" type="double" value="1e-12"/>
    </ParameterList>
    <ParameterList name="Linear Solver">
      <Parameter name="Graph Fill" type="int" value="1"/>
      <Parameter name="Max Iterations" type="int" value="200"/>
      <Parameter name="Method" type="string" value="BICGSTAB"/>
      <Parameter name="Precond" type="string" value="ILUK"/>
      <Parameter name="Tolerance" type="double" value="1e-12"/>
      <Parameter name="Type" type="string" value="TSF"/>
      <Parameter name="Verbosity" type="int" value="4"/>
    </ParameterList>
  </ParameterList>
</ParameterList>
```


E Websites for Related Software

- **BLAS** low-level linear algebra routines – <http://www.netlib.org>
- **Cubit** mesh generation software – <http://cubit.sandia.gov>
- **LAPACK** dense linear algebra routines – <http://www.netlib.org>
- **NetCDF** Exodus to text mesh format conversion software <http://www.unidata.ucar.edu>.
- **Paraview** Freely-available VTK viewer <http://www.paraview.org>.
- **Trilinos** High-performance solver components <http://software.sandia.gov>.

Bibliography

- [1] R.A. Bartlett, M.A. Heroux, and K. Long. Tsfcore: A package of light-weight object-oriented abstractions for the development of abstract numerical algorithms and interfacing to linear algebra libraries and applications. Technical Report SAND2003-1378, Sandia National Laboratories, 2003.
- [2] R. P. Feynman, R. B. Leighton, and M. Sands. *The Feynman Lectures on Physics, Volume II*. Addison-Wesley, 1963.
- [3] P. M. Gresho and R. L. Sani. *Incompressible Flow and the Finite Element Method*. Wiley, 1998.
- [4] M. Heroux, R. Bartlett, V. Howle, R. Heekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An overview of trilinos. Technical Report SAND2002-2729, Sandia National Laboratories, 2003.
- [5] K. Long. Efficient discretization and differentiation of partial differential equations through automatic functional differentiation. In B. Norris and P. Hovland, editors, *Automatic Differentiation 2004*, 2004.
- [6] K. Long, R.A. Bartlett, P.T. Boggs, M.A. Heroux, P.A. Howard, V.E. Howle, and J.P. Reese. TSFExtended user's guide. Technical Report In Preparation, Sandia National Laboratories, 2004.
- [7] P. N. Shankar and M. D. Deshpande. Fluid mechanics in the driven cavity. *Annual Review of Fluid Mechanics*, 32:93–136, 2000.
- [8] F. Shu. *The Physics of Astrophysics, Volume II: Radiation*. University Science Books, 1992.
- [9] B. van Bloemen Waanders, R. Bartlett, K. Long, P. Boggs, and A. Salinger. Large scale non-linear programming for PDE constrained optimization. Technical Report SAND2002-3198, Sandia National Laboratories, 2002.

DISTRIBUTION:

Omar Ghattas
 Carnegie Mellon University
 5000 Forms Ave.
 Pittsburgh, PA 15213 1

Alex Cunha
 Carnegie Mellon University
 5000 Forms Ave.
 Pittsburgh, PA 15213 1

Judy Hill
 Carnegie Mellon University
 5000 Forms Ave.
 Pittsburgh, PA 15213 1

Larry Biegler
 Department Chemical Engineering
 Carnegie Mellon University
 5000 Forms Ave.
 Pittsburgh, PA 15213 1

Matthias Heinkenschloss
 Department of Computational and Applied Mathematics
 MS 134 Rice University
 6100 S. Main Street
 Houston, TX 77005-1892 1

Bill Symes
 Department of Computational and Applied Mathematics
 MS 134 Rice University
 6100 S. Main Street
 Houston, TX 77005-1892 1

Tony Padula
 Department of Computational and Applied Mathematics
 MS 134 Rice University
 6100 S. Main Street
 Houston, TX 77005-1892 1

Mark Gockenbach
 Department of Mathematical Sciences
 Michigan Technological University
 1400 Townsend Drive
 Houghton, Michigan 49931-1295, U.S.A. 1

George Biros
 Department of Applied Mechanics and Mechanical Engineering
 University of Pennsylvania
 220 Towne Building 220 S. 33rd St.
 Philadelphia, PA 19104-6315, USA 1

David Keyes 1

Matthew Knepley 1

Robert C. Kirby 1

Patricia A. Howard 1

Paul Hovland 1

Allen S. Harvey 1