AD NUMBER

ADB238737

NEW LIMITATION CHANGE

TO

Approved for public release, distribution unlimited

FROM

Further dissemination only as directed by U.S. Army Missile Command, Attn: AMSMI-RD-WS-DP-SB, Redstone Arsenal, AL, Oct 98 or higher DoD authority.

AUTHORITY

DTIC/OMI

FINAL REPORT

SEPTEMBER 30, 1998

CONTRACT TITLE: A DOMAIN Specific Library and APE for Simulation of Partial Differential Equations in Heterogeneous Environments

CONTRACT NUMBER: DAAHO1-97-C-R102

PI: Gal Berkooz, Ph.D.

Beam Technologies, Inc.
110 N. Cayuga
Ithaca, NY 14850-4331

Prepared for DEFENSE ADVANCE RESEARCH PROJECTS AGENCY
US ARMY MISSILE COMMAND
REDSTONE ARSENAL, ALABAMA

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	30 September 1998	Final 20 Mar 97 through 30 September 98

4. TITLE AND SUBTITLE
A Domain specific Library and APE for Simulation or Paratial Differential Equations in Heterogeneous Environments

5. FUNDING NUMBERS
DAAHO1-97-C-R102

6. AUTHOR(S)
Gal Berkooz, Ph.D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Beam Technologies, Inc.
110 N. Cayuga
Ithaca, NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
US ARMY COMMAND
Attn: Charles R. Piner, Technical Monitor
AMSMI-RD-WS-DP-SB
Building 7804, Room 222
Redstone Arsenal, AL 35898-5248

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

PDESolve is a C++ (object-oriented) class library that enables a high level expression of ordinary and partial differential equations, and interfaces with real-world engineering tools such as CAD systems. PDESolve is a new class of tool, offering mixed symbolic and numeric computing, open C++ architecture, easy interface with engineering tools and the potential for scalable performance. Beam has found that the reusability and exibility that come with the object-oriented and math-based architecture of PDESolve allows dramatically faster development of engineering solutions. These solutions are often two to three orders of magnitude shorter in terms of line count compared with programming in C or FORTRAN, and take accordingly less time to develop. The objective of this project was to develop parallel computing capabilities in the PDESolve substrate. The main achievements of the work were designing the architecture of the finite element engine so that it is scalable and appropriate for parallel computing. The work also had a significant component of technology transition: DoD customers at Lockheed and Boeing were exposed to the technology and Beam is following up on this interest.

14. SUBJECT TERMS
Parallel computing, C++, object oriented computing, symbolic and numeric computing.

15. NUMBER OF PAGES
75

16. PRICE CODE
NSP

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	None

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

A Domain Specific Library and APE for Simulation of Partial Differential Equations in Heterogeneous Environments

DAAHO1-97-C-R102

Scale PDESolve Final Report

Technical Achievements

The objective of this research was to develop a parallel computing version of Beam's object oriented C++ class library, PDESolve. The research carried out was the first year of a two-year research effort. In this contracting period the kernel of the PDESolve library was redesigned so as to enable efficient execution of parallel computing applications. This redesigned kernel is the core of a new release of the PDESolve library. The User's Guide for this latest release is enclosed as part of this report.

PDESOLVE C++ Library

User's Guide

Version 2.0 *Beta*

August 28, 1998

PDESOLVE Version 2.0 *Beta*

Beam Technologies, Inc.
110 North Cayuga Street
Ithaca, NY 14850-4331

phone: 607-273-????
(sales, tech support, bug reports, info, etc.)
email addresses:
(sales, tech support, bug reports, info, etc.)
Web site: http://www.beamtech.com

Contents

Foreword

What is PDESOLVE?

PDESOLVE is an environment for numerically solving partial differential equation (PDE) problems. PDESOLVE combines object-oriented programming with a natural framework for working with differential equations. PDESOLVE's fundamental objects, differential equations, simulate and optimize a wide variety of scientific and engineering products.

PDESOLVE can be used to model any system whose underlying behavior can be described with differential equations. PDESOLVE provides objects to describe the various components of a PDE problem, its discretization, and its solution.

PDEs are fundamental to many areas of science and engineering. PDESOLVE can be used to solve PDE problems in many different fields, including finance, life sciences, engineering, and physics.

Who is Beam Technologies?

Beam Technologies is a scientific computing company based in Ithaca, NY. Beam has developed new design tools and techniques that meet many of the pressing needs of scientific and engineering enterprises. The fundamental technology driving Beam's vision is the creation of open standards that allow the communication of comprehensive engineering information on IP networks.

For more information about Beam Technologies or to find out about our other products, visit our web site: http://www.beamtech.com.

How to Get Help

Sample Programs

Sample programs are located in the *(???)* subdirectory of your PDESOLVE directory.

PDESOLVE Web Page

(these are some suggestions for what might be there)

- documentation

  Tutorial

  User's Guide

- FAQ
- updates (?)
- more examples (?)
- user contributed examples

  this link contains examples of how PDESOLVE has been used by our customers for various types of problems

- ...

Customer Support

Customer support is available by:

- phone –
- email –
- web site—http://www.beamtech.com

Please have the following information available when contacting customer support:

- software version number
- platform

To display the version number of your copy of PDESOLVE, compile and run the following PDESOLVE program:

```
#include "PDESolve.h"

void main()
{
// Outputs "PDESolve version 2.0"
cout << PDEGlobal::version();
}
```

(This is what the version number function is planned to look like – it doesn't work yet as far as I can tell. What if their problem is that they can't compile? -vh)

PDESOLVEDocumentation

Organization of the Documentation Set

The PDESOLVE documentation is organized as follows:

Installation Guide:
This guide contains instructions on installing PDESOLVE and compiling an example and visualizing the results.

User's Guide:
This document contains a brief walk through of the basic ideas of FEM and their implementation in PDESOLVE using several progressively more sophisticated examples. It also contains detailed information about using the Geometry classes provided by PDESOLVE for constructing domains.

Reference Manual:
This document contains a detailed description of the entire PDESOLVE class library.

Case Studies:
For FEM experts, this guide contains several non-trivial examples with brief instructions on how to set up and solve the problems in PDESOLVE.

Documentation Conventions

italic Roman font
book and section titles, references to code and directories, etc., within the text, e.g.,

sample code is located in the directory */some/directory*.

`constant width font`
code fragments and PDESOLVEcommands. Sample names that should be replaced by your own information are in *italic font*, e.g.,

to compile your program type `make` (*filename*).

Abbreviations

Abbreviations used throughout the documents:

FD Finite Difference

FEM Finite Element Method

PDE Partial Differential Equation

References

You may want to consult the following references for background information on Finite Differences, the Finite Element Method, or other mathematical background:

A. Iserles, *A First Course in the Numerical Analysis of Differential Equations*, Cambridge University Press, 1996.

O. C. Zienkiewicz, FRS, and R. L. Taylor, *The Finite Element Method*, Fourth Ed., V. 1, McGraw–Hill, London, 1989.

The PDESOLVEdocumentation assumes that you are familiar with basic C++ syntax and programming, and PDESOLVE commands follow C++ syntax. You may want to consult one of the following references for more information on C++ and C++ syntax.

B. Stroustrup, *The C++ Programming Language*, Addison–Wesley, Reading, MA, 1987.

Chapter 1

Mathematics and PDESOLVE

This chapter describes the components of a typical PDE problem.

The first section, *Anatomy of a PDE Problem*, describes the various mathematical components of a PDE problem and introduces mathematical terminology that we will use throughout the documentation.

The second section, *Anatomy of a* PDESOLVE *Program*, discusses schematically the components necessary for a numerical solution to the PDE problem. (The *Examples* chapter provides detail on specific PDESOLVE syntax for incorporating these mathematical components into your code.)

1.1 Anatomy of a PDE Problem

A partial differential equation (PDE) problem has several necessary mathematical components:

- the PDE itself
- the domain
- appropriate boundary conditions (BC) or initial conditions (IC)

All of these components must be specified before you have a mathematically complete problem (with a unique solution).

1.1.1 The PDE

The PDE is the mathematical equation that models the behavior you are trying to predict or understand. Some typical example PDEs are:

The wave equation (in 3 dimensions):

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = \frac{\partial^2 u}{\partial t^2}$$

The heat equation (in 2 dimensions):

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial u}{\partial t}$$

Poisson's equation (in 2 dimensions):

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = F(x, y).$$

Poisson's equation, for example, can be used to represent the steady-state heat distribution over a given region.

1.1.2 The Domain

The domain is the region over which you are solving the PDE. PDESOLVE supports one, two, and three-dimensional domains. The domain involved in your problem may be a factor in choosing between using FD and FEM to solve the problem. To use FD on a non-rectangular domain (of any dimension), you must have an invertible mapping from the domain to a domain that is either rectangular or is a collection of rectangular sub-domains.

Your domain could be a simple or complex region. For example, you might want to solve a 2-dimensional PDE over the unit disk $x^2 + y^2 \leq 1$. Your domain can also be a complicated region that can be approximated by a combination of rectangles or disks. Or finally, your domain may be a very complicated region that cannot be easily approximated as a combination of simpler regions.

1.1.3 Boundary Conditions

Boundary conditions specify the required behavior of your solution on the boundaries of your domain. For a boundary-value PDE problem, you must specify the behavior of the solution on every boundary of the domain.

Warning: PDESOLVE does not check or enforce well-posedness of your boundary conditions.

In second-order PDEs, there are two main categories of boundary conditions. The first involves specifying the value of the function at the boundary. This type of boundary condition is known as a *Dirichlet, essential, geometric,* or *forced* boundary condition. We will use the term *Dirichlet boundary condition* for this category. An example of a Dirichlet boundary condition is $u(0) = 0$ or $u(0, y) = g(y)$, where $g(y)$ is some specified function.

The second category involves specifying the value of the normal derivative of the function at the boundary. This type of boundary condition is known as a *Neumann, natural,* or *force* boundary condition. We will use the term *Neumann boundary condition* for this category. An example of a Neumann boundary condition is $\partial u/\partial x = 0$ at $x = 0$.

For example, when solving the two-dimensional Poisson equation on the domain $\Omega = \{ (x, y) \mid 0 < x < 1, 0 < y < 1 \}$, your problem might require that the solution obey Dirichlet boundary conditions on the x boundaries of the square,

$$u(0, y) = 0, \qquad u(1, y) = 0,$$

and Neumann boundary conditions on the y boundaries of the square,

$$\left.\frac{\partial u}{\partial y}\right|_{y=0} = g(x), \qquad \left.\frac{\partial u}{\partial y}\right|_{y=1} = h(x),$$

where g and h are specified functions.

1.1.4 Initial Conditions

For time-dependent problems such as the heat equation and the wave equation, you must also specify initial conditions. If the time derivative is order n, then you need n initial conditions.

For example, suppose you want to solve the two-dimensional heat equation on the domain Ω as specified above for the Poisson equation. Then you must specify boundary conditions just as above *and* the initial temperature distribution

$$u(x, y, 0) = u_0(x, y) \qquad \text{for all } (x, y) \in \Omega.$$

To solve the wave equation on the same domain, we need the initial displacement

$$u(x, y, 0) = u_0(x, y) \qquad \text{for all } (x, y) \in \Omega,$$

as well as the initial velocity

$$\frac{\partial u}{\partial t}(x, y, 0) = v_0(x, y) \qquad \text{for all } (x, y) \in \Omega.$$

1.1.5 Weak Formulation of the PDE — FEM Only

To solve a PDE problem using Finite Differencing (FD), the mathematics can stop here. We have specified the mathematical problem completely and the next step is to incorporate all of the components into a PDESOLVE program.

To solve a PDE problem using the Finite Element Method (FEM), we need several further mathematical steps.

When solving with FEM, we represent the solution to the PDE as a linear combination of functions from a (possibly infinite) set of *basis* functions. As the PDE problem was originally formulated, these basis functions would have to be differentiable to the extent required by the PDE. In Poisson's equation, for example, the basis functions would have to be twice continuously differentiable so that a linear combination of them would make sense as a solution to the equation. This requirement is often more restrictive than is necessary for a good numerical solution. In FEM, we work with the "weak form" of the PDE. One advantage of the weak form is that it decreases the continuity requirements on the basis functions.

Consider the one-dimensional Poisson equation:

$$\frac{\partial^2 u}{\partial x^2} = F(x),$$

on the domain $a \leq x \leq b$, with a Neumann boundary condition $\partial u/\partial x = \pi$ at $x = a$, and a Dirichlet boundary condition $u = 0$ at $x = b$.

Let $v(x)$ be an arbitrary (variational) function. Create a functional out of the PDE and the Neumann boundary condition, $\partial u/\partial x - \pi = 0$ at $x = a$, as follows:

$$\int_a^b (v\frac{d^2u}{dx^2})dx - \int_a^b (vF(x))dx - \oint_a v(\frac{\partial u}{\partial x} - \pi) = 0.$$

The lefthand side of this equation is the functional applied to v. If u satisfies the PDE, then this equation must be satisfied for any v. Conversely, if the function u satisfies this equation for any function v, then it must also satisfy the original PDE. Integrating by parts, we get

$$\int_a^b (\frac{du}{dx}\frac{dv}{dx})dx + \int_a^b (vF(x))dx - \frac{du}{dx}v\Big|_a^b - v\frac{\partial u}{\partial x}\Big|_a + v\pi\Big|_a = 0.$$

In this example, we have a Dirichlet boundary condition at the point $x = b$, so we do not want the function v to vary at that point. Consequently, we assume that $v(b)$ is zero (i.e., assume that v is any arbitrary function such that $v(b) = 0$). The equation then becomes

$$\int_a^b (\frac{du}{dx}\frac{dv}{dx})dx + \int_a^b (vF(x))dx + \pi v(a) = 0.$$

This is the expression of the weak form that we use in the PDESOLVE program.

More generally, if we have a PDE, $Lu - F = 0$, where L is a linear operator on u and F is a function of x, over a domain Ω with a Neumann boundary condition, $\partial u/\partial n - k = 0$ on $\partial\Omega_N$, where $\partial\Omega$ is the boundary of the domain and $\partial\Omega_N$ is the subregion of the boundary on which the Neumann boundary condition holds, k is a constant, and $\partial u/\partial n$ represents the normal derivative of u at each point of $\partial\Omega_N$, the weak form of the PDE is found by

creating a functional from the PDE and the Neumann boundary condition as follows:

$$\int_{\Omega} v(Lu - F)d\Omega - \oint_{\partial\Omega_N} v(\frac{\partial u}{\partial n} - k) = 0.$$

For more information on weak formulation and on FEM techniques in general, see one of the suggested FEM references.

1.2 Anatomy of a PDESOLVE Program

Like the mathematical problem, a PDESOLVE program must contain certain components to specify the problem completely so it can be solved numerically. In this section we go through the components of a PDESOLVE program schematically. In the next chapter, we work through many specific examples.

1.2.1 Mathematical Components

A complete PDESOLVE program must contain all of the mathematical components described in the previous section. In addition, there are several components that are required for numerical solutions or for C++ programs in general. You will see how the mathematical components are turned into PDESOLVE code in the *Examples* chapter. The following sections describe the additional necessary components of a PDESOLVE program.

1.2.2 Include Files

Like any C++ program, your PDESOLVE program must contain an *include* statement for any necessary header files. In particular, the file *PDESolve.h* is required in every PDESOLVE program.

1.2.3 Solving the PDE and Viewing Results

Once you have the complete mathematical description of the PDE problem in your PDESOLVE code and have incorporated the other necessary numerical components, all that remains is to solve it and to display the results.

PDESOLVE provides a simple `solve` command to get the solutions to the problem.

There are several output formats available in PDESOLVE for viewing the results of your solution. Depending on how you intend to view the results, you will need to choose different data output formats.

For a complete description of the available output formats and provided viewing tools, see the *Viewing Your Results* section.

Chapter 2

Examples

In this chapter, we step through the Finite Element code for several increasingly complicated problems. We start with a simple one-dimensional Poisson PDE problem then proceed to solve a similar problem in two dimensions. Following this we solve two time-dependent problems: the heat equation and a first order hyperbolic equation, both in one dimension. The final example is a nonlinear problem (Burgers' equation). We explain the basics of the PDESOLVE program as they arise in these example problems. For more sophisticated examples of the kinds of problems that can be solved using PDESOLVE see the *Case Studies* document.

Complete code for example programs, including the ones in this chapter, can be found in the */pdesolve/examples* directory.

2.1 One-dimensional Poisson Equation

2.1.1 The Problem

Let us start with a one dimensional Poisson equation on a line with some simple boundary conditions. This equation could represent, for example, the steady-state heat flow in a wire.

In particular, let us work with the following Poisson equation:

$$\frac{\partial^2 u}{\partial x^2} = -\pi^2 \sin \pi x,$$

on the domain $0 < x < 1$, with a Neumann boundary condition $\partial u/\partial x = \pi$ at $x = 0$, and a Dirichlet boundary condition $u = 0$ at $x = 1$.

As we discussed in the previous chapter, the mathematical PDE problem has several necessary components: the PDE itself, the domain boundary conditions and a weak formulation.

The exact solution to this problem is

$$u(x) = \sin \pi x.$$

2.1.2 Walk-through of the FEM PDESOLVE code

The first step in solving this problem using the Finite Element Method is a mathematical step — rewriting the PDE in its weak form.

Let $v(x)$ be an arbitrary (variational) function. Multiplying the PDE by v and integrating, we get:

$$\int_0^1 v \frac{d^2u}{dx^2}\, dx + \int_0^1 v\pi^2 \sin \pi x\, dx = 0.$$

Notice that if the function u satisfies this equation for any function v, then it must also satisfy the original PDE. Integrating by parts, we get

$$\int_0^1 \frac{du}{dx}\frac{dv}{dx}\, dx - \int_0^1 v\pi^2 \sin \pi x\, dx - \frac{du}{dx} v \Big|_0^1 = 0.$$

In this example, we have a Dirichlet boundary condition at the point $x = 1$, so we assume that $v(1)$ is zero. Enforcing the Neumann boundary condition $du/dx = \pi$ at $x = 0$, the equation becomes

$$\int_0^1 \frac{du}{dx}\frac{dv}{dx}\, dx - \int_0^1 v\pi^2 \sin \pi x\, dx + \pi v(0) = 0.$$

This is the form of the problem that we put into the PDESOLVE program. Now we will put together the PDESOLVE program to solve this problem.

Outside *main*

As always, we need to include the PDESOLVE header file *PDESolve.h*.

```
#include "PDESolve.h"
```

Since this PDE is inhomogeneous, we need to set up the forcing function

$$f(x) = -\pi^2 \sin \pi x.$$

We can set it up in PDESOLVE as follows:

```
Real f(const Coords& x)
{ return -M_PI*M_PI* sin(M_PI*x[0]); }
```

`Real` declares the function `f` to be a real-valued function.

Within *main*

Next we enter the *main* part of the program. This is where most of the PDESOLVE coding will occur in this example.

First, we set up the differential operators needed to represent the PDE. The weak formulation of the PDE problem involves only first derivatives with respect to x, so the only differential operator needed is one representing a single derivative with respect to x.

```
DiffOp dx(1,0);
```

This statement declares `dx` to represent a first derivative operator with respect to the zeroth variable (x). Note that PDESOLVE follows the C/C++ convention that counting starts with zero.

Next, we declare the functions that are needed to represent the weak form of the PDE problem. In this case, we need the inhomogeneous function f, a function G representing the Neumann boundary condition, a function u representing the solution to the PDE problem, and a variational function v to use in the weak formulation of the problem.

```
const int dim = 1;
const int order = 1;
Function F(dim, Scalar, f);
Function G((Real) M_PI);
Function u(dim, Scalar, Lagrange(order));
Function v(dim, Scalar, Lagrange(order), VARIATIONAL);
```

Here we are setting F to refer to a pointer to the 1-dimensional scalar function f that we defined before *main*. G is declared to be the constant function with value π (note that some compilers define the constant M_PI to be long double, hence we must cast it to Real), u is declared to be a one-dimensional scalar-valued unknown function, and v is declared to be a one-dimensional scalar-valued variational function. (We could have used the parameter UNKNOWN in the fourth slot in the declaration of u in the same way that we declare v to be VARIATIONAL, but that is unnecessary since UNKNOWN is the default.) The parameter Lagrange(order) tells PDESOLVE to use a basis of first order (linear) Lagrange polynomials in the discretization of u and v. Currently the only basis functions PDESOLVE supports are Lagrange polynomials of any order.

The next step is to set up and discretize the domain of the problem. The information on the geometry of the elements is kept in an object of the class CellComplex which you can think of as being a collection of finite elements on a region. The following lines set the domain to be the line segment $0 < x < 1$ divided into 20 elements (using 21 vertices).

```
const int n = 21;
CellComplex mesh = rectMeshGen(n, 0.0, 1.0);
```

Now that we have all of the differential operators and functions needed, and have set up and discretized the domain, we can write out the weak form of the PDE problem.

```
Expr eqn = Integral((dx*u)*(dx*v))
           + Integral(v*F) + Integral("x=0",G*v);
```

Notice that the equation is an expression, Expr. An Expr is a symbolic expression tree which can hold Functions, DiffOps and constants as well as

common operations and operators. $G = \pi$ is the flux through the left-hand boundary (point) given in the Neumann boundary condition of the problem. Note that the third integral in the equation, `Integral("x=0",G*v)`, is the "integral" over the point $x = 0$ and is equivalent to $\pi v(0)$. The string `"x=0"` refers to the left-hand boundary in the x direction. This label was assigned by the meshing function `rectMeshGen()`.

At the right-hand boundary, we replace the weak form with the Dirichlet boundary condition $u(1) = 0$.

```
ReplaceBC bc("x=L", v*u);
```

This is the integral version of the boundary condition, where the "integral" is over the point "x=L" which was defined by `rectMeshGen`. (Note that `L` refers to the *length* of the interval.) Note that the Dirichlet boundary condition effectively forces the variational functions v to integrate to 0 at the point $x = 1$. With that restriction, the weak form that we have written into our PDESOLVE program is equivalent to the form we got from integrating by parts.

Finally, we convert the problem into a linear system of equations and solve.

```
LinearProblem poisson1d(mesh, eqn, bc, u, v);
u = poisson1d.solve();
```

The first line combines the information about the geometry, the equation and the boundary conditions to construct a `LinearProblem`; i.e., a linear system of equations. The second line solves this system and uses the result to construct a discretized function (a linear combination of the basis functions) representing the solution.

Finally we can output our results as follows:

```
cout << u << endl;
```

Since we are using Lagrange interpolating polynomials as our basis functions, the numbers we output are just the nodal values of the approximate solution. If we were using second or higher order polynomials, the first n

numbers would be the values at the vertices, and the remaining would be the values at the internal nodes.

The entire listing of the PDESOLVE code for this problem is given below. When we run this code, we get the following output:

```
FEFunc Lagrange(1)0.00323215
0.159344
0.311608
0.45627
0.589765
0.708802
0.810446
0.892189
0.952016
0.988448
1.00059
0.988125
0.951369
0.89122
0.809153
0.707186
0.587826
0.454008
0.309022
0.156435
0
```

We could of course achieve a more accurate result by increasing the number of elements or their order. Figure 2.1 displays the numerical result.

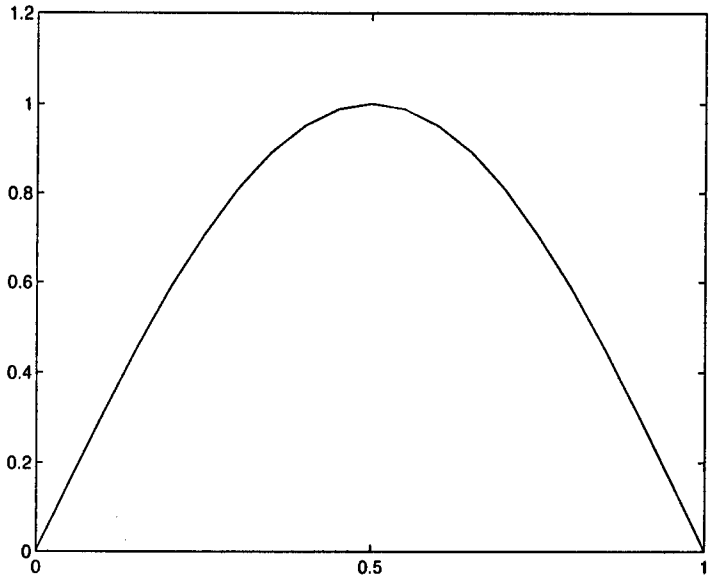

Figure 2.1: *Numerical solution to 1-D Poisson equation.*

Program: Poisson1d.cpp

```
// Copyright (c) 1998 BEAM Technologies Inc.
//
// Solve the 1D Poisson equation
// $u_{xx} = -\pi^2\sin(\pi x)$ on $(0,1)$
// $u_x(0) = \pi$, $u(1) = 0$
//
// The exact solution is $u(x) = \sin(\pi x)$.
//

#include "PDESolve.h"
#include "ReplaceBC.h"

// Set up the right-hand side of the PDE

Real f(const Coords& x)
{ return -M_PI*M_PI* sin(M_PI*x[0]); }

int main()
{
  const int order = 1;
  const int dim = 1;
  const int n = 21;
```

```
  DiffOp dx(1,0);

// Declare Functions

  Function F(dim,Scalar,f);
  Function G((Real) M_PI);
  Function u(dim,Scalar,Lagrange(order));
  Function v(dim,Scalar,Lagrange(order),VARIATIONAL);

// Discrete Mesh

  CellComplex mesh = rectMeshGen(n, 0.0, 1.0);

// Weak formulation of PDE with Neumann BC incorporated

  Expr eqn = Integral((dx*u)*(dx*v)) + Integral(F*v) + Integral("x=0",G*v);

// Dirichlet boundary condition

  ReplaceBC bc("x=L", u*v);

// Set up the problem as a linear system of equations and solve.

  LinearProblem poisson1d(mesh,eqn,bc,u,v);
  u = poisson1d.solve();

// Write output

  cout << u << endl;
}
```

2.2 Two-dimensional Poisson Equation on a Square

2.2.1 The Problem

As our second example we again use a Poisson equation, but now we consider a two-dimensional problem on a square domain. The change to two dimensions will introduce a few more PDESOLVE concepts, while keeping the basic problem familiar. In particular, let us solve the following two-dimensional Poisson equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y) = -2\pi^2 \sin \pi x \sin \pi y,$$

on the square domain $\Omega = \{ (x,y) \mid 0 < x < 1,\ 0 < y < 1 \}$ with Dirichlet boundary conditions, $u(x,y) = 0$, on all of the edges of the square except the $x = 0$ edge, and the Neumann boundary condition $\nabla u \cdot \mathbf{n} = g(y)$ on the $x = 0$ edge, where $g(y) = -\pi \sin \pi y$, and $\mathbf{n}$ is the outward unit normal vector on $\partial\Omega$ the boundary of Ω.

The exact solution to this problem is:

$$u(x,y) = \sin \pi x \sin \pi y.$$

2.2.2 Walk-through of the FEM PDESOLVE code

As in the previous example, the first step in using FEM is rewriting the PDE in its weak form. Let $v(x,y)$ be an arbitrary (variational) function. Multiplying the PDE by v and integrating we get:

$$\int_\Omega \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) v \, dx \, dy = \int_\Omega f v \, dx \, dy.$$

Integrating by parts and applying the Divergence Theorem (or applying Green's formula), we get:

$$\int_\Omega \nabla u \cdot \nabla v \, dx \, dy + \int_\Omega f v \, dx \, dy - \oint_{\partial\Omega} \frac{\partial u}{\partial \mathbf{n}} v = 0.$$

To satisfy the Dirichlet boundary conditions, we require that $v(x,y) = 0$ on all the edges except the $x = 0$ edge. Thus the line integral is zero everywhere except on the $x = 0$ edge where $\partial u/\partial \mathbf{n} = g(y)$. Labelling this part of the boundary Γ_N, the equation becomes:

$$\int_\Omega \nabla u \cdot \nabla v \, dx \, dy + \int_\Omega f v \, dx \, dy - \int_{\Gamma_N} g v \, ds = 0.$$

This is the form of the equation that we will put into our PDESOLVE program.

Outside *main*

As always, we must include the PDESOLVE header file:

```
#include "PDESolve.h"
```

As in the previous example, we set up the inhomogeneous part of the PDE outside *main*. In this case, the inhomogeneous function is:

$$f(x,y) = -2\pi^2 \sin \pi x \sin \pi y,$$

which we write in PDESOLVE as:

```
Real f(const Coords& x)
{
   return -2*M_PI*M_PI*
               sin(M_PI*x[0])*sin(M_PI*x[1]);
}
```

Note that `x[0]` represents the x variable, and `x[1]` represents the y variable. (PDESOLVE follows the C/C++ convention of counting from zero.)

Next we set up the function $g(y)$ representing the flux along the $x = 0$ edge of the unit square (the Neumann boundary condition).

```
Real g(const Coords& x)
{
   return -M_PI*sin(M_PI*x[1]);
}
```

Within *main*

In the *main* part of the program, we set up the weak form of the PDE, set up and discretize the domain, and solve the problem.

First we set up the differential operators. In this case, we need first derivatives with respect to x and y. In particular, we need to set up a gradient operator which we can form by building a vector of differential operators. In PDESOLVE, a vector is represented as a `List`. The gradient operator is then a `List` whose first component is a partial derivative operator with respect to x and whose second component is a partial derivative operator with respect to y. The syntax in PDESOLVE is:

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

Recall that the first component of the argument to `DiffOp` represents the order of the derivative, and the second component represents the variable with respect to which the derivative is taken. 0 refers to the first (x) variable, 1 refers to the second (y) variable.

Notice that the gradient is declared to be an `Expr` so that it can be applied symbolically to functions of two variables.

Next we declare the functions that will be needed in the statement of the PDE:

```
const int dim = 2;
const int order = 2;
Function F(dim, Scalar, f);
Function G(dim, Scalar, g);
Function u(dim, Scalar, Lagrange(order));
Function v(dim, Scalar, Lagrange(order), VARIATIONAL);
```

Here we are declaring F and G to be 2-dimensional scalar-valued functions. F is the inhomogeneous part of the PDE, and G represents the Neumann boundary condition on the $x = 0$ edge of the domain. u is declared to be a 2-dimensional scalar-valued unknown function, and v is declared to be a

2-dimensional scalar-valued variational function. For this example we are using second order (quadratic) Lagrange polynomials for our basis.

Next we set up and discretize the domain. In this case the domain is the unit square, which we are meshing using 11 vertices in each direction.

```
const int n = 11;
CellComplex mesh = rectMeshGen(n,n,0.0,1.0,0.0,1.0);
```

The function `rectMeshGen` can be used to specify a rectangular domain in either two or three dimensions. This function generates the mesh and assigns the labels "x=0", "x=L", "y=0" and "y=L" (and "z=0" and "z=L" in the three dimensional case) to the appropriate parts of the boundary.

Now we have all of the pieces necessary to set up the weak form of the PDE:

```
Expr eqn = Integral((grad*u)*(grad*v)) + Integral(F*v)
           - Integral("x=0",G*v);
```

Note that the star operator, `*`, is being used in three ways in this fragment: `grad*u` is applying the gradient operator to u (∇u), the `*` operating on the two gradients is a dot product, and the star in `F*v` and `G*v` is scalar multiplication.

Next we set up the Dirichlet boundary conditions on the remaining three faces of the square:

```
ReplaceBC bcs = ReplaceBC("x=L", u*v)
             && ReplaceBC("y=0", u*v)
             && ReplaceBC("y=L", u*v);
```

Finally, we convert the problem into a linear system of equations and solve:

```
LinearProblem poisson2d(mesh, eqn, bcs, u, v);
u = poisson2d.solve();
```

The entire listing of the PDESOLVE code for this problem is given below. When this code is run, the error returned is `0.000362359`.

Figure 2.2 displays the numerical result. Here we are using the MATLAB script *PDEPlot* to read the output file and display the results. See the *Viewing Your Results* section for details.

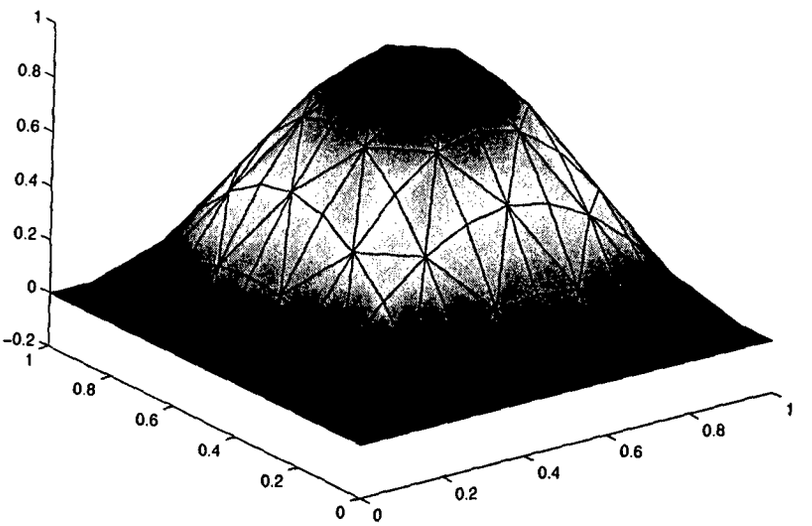

Figure 2.2: *Numerical solution to 2-D Poisson equation.*

Figure 2.3 shows the same results looking down from the z-axis. In this view you can see the undeformed mesh.

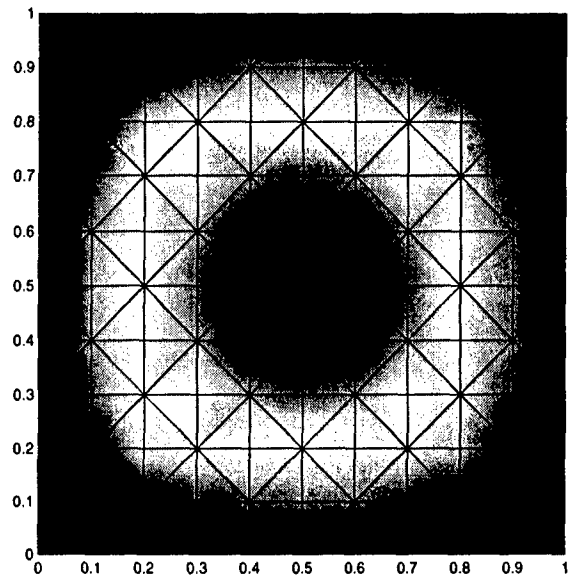

Figure 2.3: *Numerical solution to 2-D Poisson equation viewed from above.*

Program: Poisson2d.cpp

```
// Copyright (c) 1998 by BEAM Technologies, Inc.

// Solves the 2-D steady-state heat equation on the unit square.
// div(grad u) = f(x,y)
// with the boundary conditons:
// u(x,y) = 0    (Dirichlet)  on all the edges EXCEPT x = 0
// flux = g(y)   (Neumann)    on the edge x = 0
// The exact solution, f, and g are given below.

#include "PDESolve.h"
#include "ReplaceBC.h"  // should be included by PDESolve.h ?

#define DIM 2  // 2-D problem

// Actual answer.

Real ans(const Coords& x)
{ return sin(M_PI*x[0])*sin(M_PI*x[1]); }

// Internal generation.

Real f(const Coords& x)
{ return -2*M_PI*M_PI* ans(x); }
```

```
// Flux on x=0 edge. g(y) = - du/dx(x = 0).

Real g(const Coords& x)
{ return -M_PI*sin(M_PI*x[1]); }

Real myL2Norm(const Expr& func, const CellComplex& cc, const int orderGL=4);

int main()
{
  const int order = 2;
  const int n = 11;

// Gradient operator.

  DiffOp dx(1,0);
  DiffOp dy(1,1);
  Expr grad = List(dx, dy);

// Functions.

  Function F(DIM, Scalar, f);                              // internal generation
  Function G(DIM, Scalar, g);                              // boundary flux
  Function u(DIM, Scalar, Lagrange(order));                // unknown function
  Function v(DIM, Scalar, Lagrange(order), VARIATIONAL); // test functions

// Discrete mesh.

  CellComplex mesh = rectMeshGen(n, n, 0.0, 1.0, 0.0, 1.0);

// Weak formulation of the PDE.

  Expr eqn = Integral((grad*u)*(grad*v)) + Integral(F*v)
           - Integral("x=0",G*v);

// Essential (Dirichlet) boundary conditions.

  ReplaceBC bcs = ReplaceBC("x=L",u*v) && ReplaceBC("y=0", u*v)
               && ReplaceBC("y=L",u*v);

// Set up the problem as a linear system of equations.

  LinearProblem poisson2d(mesh, eqn, bcs, u, v);

// Solve the problem.

  u = poisson2d.solve();

// Compute 2-norm of error.

  Function answer(DIM, Scalar, ans);
  cout << "error norm: " << myL2Norm(answer - u, mesh, 12) << endl;
}

Real myL2Norm(const Expr& func, const CellComplex& cc, const int orderGL)
{
```

```
  Real r = Integral(cc, func*func, GaussLegendre(orderGL));
  return sqrt(r);
}
```

There are no significant conceptual differences in going from this two dimensional example to a three dimensional example. We have introduced all of the PDESOLVE code that would be needed. For a three dimensional example, see *Poisson3d.cpp* in your */pdesolve/examples* directory.

2.2.3 Specifying Dirichlet Boundary Conditions By the Method of Lagrange Multipliers

In the examples considered thus far we imposed the Dirichlet boundary conditions using `ReplaceBC`. This method entails using the boundary conditions to replace certain rows in the problem's matrix. This method works whenever the coefficients in the expansion of a function in our basis are related to the function values on a node-by-node basis. When using Lagrange interpolating polynomials, for example, the coefficients are simply the nodal values of the function. However, with a different choice of basis functions, say Legendre polynomials or trig functions, there is no simple relationship between the coefficients and the function values at the nodes. In this case, it is not simple to translate the constraints at a set of nodes to changes in a few rows of the matrix; i.e., `ReplaceBC` will not work.

When we cannot use `ReplaceBC` we can instead use the method of Lagrange multipliers. To illustrate the method, suppose we want to solve the following boundary value problem:

$$\begin{aligned} \nabla^2 u &= f \quad \text{in } \Omega \\ u &= g \quad \text{on } \Gamma_1 \subset \partial\Omega \\ \frac{\partial u}{\partial \mathbf{n}} &= h \quad \text{on } \Gamma_2 = \partial\Omega - \bar{\Gamma}_1 \end{aligned}$$

The weak form of the equation is

$$\int_\Omega \nabla u \cdot \nabla v \, dx + \int_\Omega f v \, dx = \int_{\partial\Omega} \frac{\partial u}{\partial \mathbf{n}} v \, ds,$$

which we obtain by projecting the strong form of the equation onto a space of variational functions $\{ v \}$ and demanding that the residual on this space be zero. Since the above equation must hold for all v, there are as many rows in our system of equations as there are linearly independent variational functions.

We impose the Neumann boundary condition by replacing $\partial u/\partial \mathbf{n} = \nabla u \cdot \mathbf{n}$ with the imposed flux on Γ_2:

$$\int_\Omega \nabla u \cdot \nabla v \, dx + \int_\Omega f v \, dx = \int_{\Gamma_1} \frac{\partial u}{\partial \mathbf{n}} v \, ds + \int_{\Gamma_2} h v \, ds.$$

To impose the Dirichlet boundary conditions, we introduce a new set of variational functions, $\delta\lambda$, which are linearly independent of the functions v. We then add to the above system the projection of the condition $u = g$ onto the space $\delta\lambda$:

$$\int_\Omega \nabla u \cdot \nabla v \, dx + \int_\Omega f v \, dx = \int_{\Gamma_1} \frac{\partial u}{\partial \mathbf{n}} v \, ds + \int_{\Gamma_2} h v \, ds + \int_{\Gamma_1} \delta\lambda (u - g) \, ds.$$

Just as above, we get as many equations as there are linearly independent variational functions; hence, we now have an overdetermined system. In order to be able to solve the system, we must introduce additional degrees of freedom. The obvious choice is to allow $\nabla u \cdot \mathbf{n}$ to vary independently of u on the boundary Γ_1. Replacing $\partial u/\partial \mathbf{n}$ with a new variable λ we have

$$\int_\Omega \nabla u \cdot \nabla v \, dx + \int_\Omega f v \, dx = \int_{\Gamma_1} \left(\lambda v + \delta\lambda (u - g) \right) \, ds + \int_{\Gamma_2} h v \, ds.$$

We can now solve simultaneously for the two unknown functions u and λ.

Now, let's look at the changes required in the PDESOLVE code when using this method. To illustrate, we will solve the same 2D Poisson equation as above.

First, we must add the declarations of the new unknown function λ and its variation $\delta\lambda$:

```
Function lambda(DIM, Scalar, Lagrange(order));
Function varLambda(DIM, Scalar, Lagrange(order),
                                VARIATIONAL);
```

We change the weak formulation of the PDE to include the Dirichlet BCs:

```
Expr eqn = Integral((grad*u)*(grad*v)) + Integral(F*v)
            - Integral("x=0",v*G)
            + Integral("x=L",u*varLambda+v*lambda)
            + Integral("y=0",u*varLambda+v*lambda)
            + Integral("y=L",u*varLambda+v*lambda);
```

and we eliminate the line imposing the BCs by `ReplaceBC`.

When the linear problem is defined we need to solve for both the unknown function u and the undetermined multiplier λ.

```
LinearProblem problem(mesh, eqn, List(u, lambda),
                                 List(v, varLambda));
Function soln = problem.solve();
u = soln[0];
```

Note that the `solve()` method returns a list that is arranged in the same order that the problem was set in. Hence in the last line we set the solution u to the first element in the returned list.

The rest of the code is identical to the previous example. The complete code can be found in the file *Poisson2d_lagr.cpp* in your */pdesolve/examples* directory.

2.3 Time Dependent Problems

2.3.1 The Heat Equation

As our next example, we consider the following time dependent problem:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad \text{on } 0 < x < \pi, 0 < t \leq T,$$

with Dirichlet boundary conditions, $u(0,t) = 0 = u(\pi,t)$, and the initial condition $u(x,0) = \sin(x)$.

The exact solution to this problem is:

$$u(x,t) = e^{-t} \sin x.$$

2.3.2 Walk-through of the FEM PDESOLVE code

Multiplying the PDE by the variational function v and integrating we get:

$$\int_0^\pi v \frac{\partial u}{\partial t}\, dx = \int_0^\pi v \frac{\partial^2 u}{\partial x^2}\, dx.$$

Integrating by parts and imposing the boundary conditions yields:

$$\int_0^\pi v \frac{\partial u}{\partial t}\, dx = -\int_0^\pi \frac{\partial u}{\partial x} \frac{\partial v}{\partial x}\, dx.$$

We will illustrate time stepping using a backward Euler (implicit) method. Given the solution u^c at the current time t^c, we want to approximate the solution u^+ at the time $t^+ = t^c + \Delta t$. We approximate the time derivative u_t^+ by the backward difference

$$u_t(x,t^+) \approx \frac{u^+ - u^c}{\Delta t}.$$

Substituting into the above equation and regrouping the terms, we obtain the form of the problem that we will put into our PDESOLVE program:

$$\int_0^\pi u^+ v\, dx + \Delta t \int_0^\pi \frac{\partial u^+}{\partial x} \frac{\partial v}{\partial x}\, dx = \int_0^\pi u^c v\, dx$$

Now we put together the PDESOLVE code to solve the problem.

Outside *main*

As always, we must include the PDESOLVE header file:

```
#include "PDESolve.h"
```

We also must set up the initial function:

```
Real u0(const Coords& x)
{  return sin(x[0]);  }
```

Within *main*

Much of this part of the code is quite similar to the first example. We declare the differential operator:

```
DiffOp dx(1,0);
```

Next we declare the functions that will be needed in the solution of the PDE. We need a function for the current value of u which will initially have the value of u_0, an unknown function for the next time step, and a variational function v.

```
const int order = 2;
Function u_current(1, Scalar, u0);
Function u_next(1, Scalar, Lagrange(order));
Function v(1, Scalar, Lagrange(order), VARIATIONAL);
```

Next we set up and discretize the domain. In this case the domain is the interval $[0, \pi]$, which we are meshing using 11 vertices.

```
const int n = 11;
CellComplex mesh = rectMeshGen(n, 0.0, M_PI);
```

Now we set up the time stepping loop.

```
Expr eqn;
Real h = 0.01;
for (Real t=0.; t<= 1.0; t += h)
{
   eqn = Integral( u_next*v )
           + h*Integral( (dx*u_next)*(dx*v) )
          == Integral( u_current*v );
   ReplaceBC bcs = ReplaceBC("x=0", u_next*v)
                && ReplaceBC("x=L", u_next*v);
   LinearProblem heat1d(mesh, eqn, bcs, u_next, v);
   u_current = heat1d.solve();
}
```

Warning: In the above code, the last line is mathematically equivalent to

```
u_next = heat1d.solve();
u_current = u_next;
```

Here we are solving for the next time step which will then become the current time step the next time through the loop. This code will cause the PDESOLVE program to fail, however, because the line

```
u_next = heat1d.solve();
```

causes `u_next` to become a known function. Thus, the second time through the loop, there will no longer be any UNKNOWN functions so PDESOLVE will not know how to set up the equation.

The entire listing of the PDESOLVE code for this problem is given below. When this code is run, the error returned is `0.00177394`.

Program: FEheat1d_BackwardEuler.cpp

```
//--------------------------------------------------------------------
//  This program solves the heat equation (via an IMPLICIT semi-discrete
//  finite element program).
//--------------------------------------------------------------------
```

```
#include "PDESolve.h"
#include "ReplaceBC.h"

//-------------------------------------------------------------------------
//  Initial data
//-------------------------------------------------------------------------
Real u0(const Coords& x)
{
   return sin(x[0]);
}

//-------------------------------------------------------------------------
//  Final data (used to compare numerical solution to exact solution)
//-------------------------------------------------------------------------

Real u1(const Coords& x)
{
   return exp(-1.0)*sin(x[0]);
}

Real myL2Norm(const Expr& func, const CellComplex& cc, const int orderGL=6);

int main()
{
//-------------------------------------------------------------------------
//  Finite element parameters (order:  order of the elements)
//                            (dim:    dimension of the problem)
//                            (n:      number of finite elements + 1)
//-------------------------------------------------------------------------
   const int  order     =  2;
   const int  dim       =  1;
   const int  n         = 11;
   Real h               = .01;  // time step

   DiffOp dx(1,0);

   Function u_next   (dim,Scalar,Lagrange(order));
   Function u_current(dim,Scalar,u0);
   Function uExact   (dim,Scalar,u1);
   Function v        (dim,Scalar,Lagrange(order),VARIATIONAL);

   CellComplex mesh = rectMeshGen(n, 0.0, M_PI);
   Expr w;

   Timer loop_time("loop time");
   loop_time.start();
   for (Real t=0; t<=1.0; t+=h)
   {
      w = Integral( u_next*v ) + Integral(h* (dx*u_next)*(dx*v) )
         == Integral( u_current*v );
      ReplaceBC bcs = ReplaceBC("x=0", u_next*v) && ReplaceBC("x=L",u_next*v);
      LinearProblem heat1d(mesh,w,bcs,u_next,v);
      u_current    = heat1d.solve();
   }
   loop_time.stop();
```

```
   cout << "The computed solution is:" << endl;
   Function uOut = FEInterpolation(u_current, mesh, 1);
   cout << uOut << endl;
   cout << "error norm is: " << myL2Norm(u_current-uExact), mesh, 12) << endl;

}

Real myL2Norm(const Expr& func, const CellComplex& cc, const int orderGL)
{
  Real r = Integral(cc, func*func, GaussLegendre(orderGL));
  return sqrt(r);
}
```

2.3.3 A Hyperbolic Equation

To introduce another time stepping method, we consider the first order one dimensional hyperbolic equation

$$u_t + \alpha u_x = 0 \quad \text{in } \Omega \subset (-\infty, \infty), 0 < t < T$$

where α is a constant and T is the final time. We have the initial condition $u(x,0) = u_0(x)$ where u_0 has compact support inside Ω. The exact solution is $u(t,x) = u_0(x - \alpha t)$.

Choose a time step Δt and a value θ with $0 \leq \theta \leq 1$. (θ is a weighting parameter between a purely explicit method ($\theta = 0$) and a purely implicit method ($\theta = 1$).) Set $t^* = t + \theta \Delta t$ and define the least-squares functional

$$I(t^*) = \int_\Omega [u_t(x,t^*) + \alpha u_x(x,t^*)]^2 \, dx.$$

Using Taylor series expansions about t^* to find approximations to u_t and u_x, we find

$$I(t^*) \approx \int_\Omega \left[\frac{u(t+\Delta t) - u(t)}{\Delta t} + \alpha(\theta u_x(t+\Delta t) + (1-\theta)u_x(t)) \right]^2 dx,$$

where we have suppressed the dependence on x. Now, taking variations with respect to $u(t + \Delta t)$, simplifying, and setting $\delta u = v$, we obtain the weak form of the equation:

$$\int_\Omega [u(t+\Delta t) - u(t) + \alpha \Delta t(\theta u_x(t+\Delta t) + (1-\theta)u_x(t))] \cdot (v + \alpha \Delta t \theta v_x) \, dx = 0.$$

See the paper of Carey and Jiang for additional details.

2.3.4 The PDESOLVE code

Just as in the previous example, we need functions `u_current` for the value of $u(t)$ and `u_next` for the value of $u(t + \Delta t)$. With these functions, we set up the problem as follows:

First define

```
Real tt = time_step*theta;
```

Then within the time-stepping loop set up the equation as

```
Expr w = Integral( (u_next+alpha*tt*dx*u_next)
             * (v+alpha*tt*dx*v) )
          == Integral( (u_current + alpha*tt*(theta-1)
                        *dx*u_current) * (v+tt*dx*v) );
```

This code will cause PDESOLVE to fail, though, because `u_current` is initially a pointer to the C function `u0`, and we cannot apply a `DiffOp` to a C function.

Warning: A `DiffOp` can be applied *only* to discretized functions.

In order for the above code to work, we must declare `u_current` as follows *after* the construction of the mesh:

```
Function u_0(dim,Scalar,u0);
Function u_current = FEInterpolation(u_0,mesh,order);
```

The complete code for this example is in the file *Hyperbolic.cpp*.

Reference

Graham F. Carey and B. N. Jiang. "Least-Squares Finite Elements for First-Order Hyperbolic Systems." *Intl. J. Num. Meth. Engr.* Vol. 26, (1988), pp. 81–93.

2.4 A Nonlinear Problem

2.4.1 Burgers' Equation

For our final example we consider the inviscid Burgers' equation:

$$u_t + uu_x = 0, \qquad u(x,0) = u_0(x).$$

We can use the least-squares time stepping method used in Section 2.3.3 for the hyperbolic equation to solve this problem. The weak form of the equation is (writing u^+ for $u(t+\Delta t)$ and u^c for $u(t)$)

$$\int_\Omega \left[u^+ - u^c + \Delta t\theta u^+ u_x^+ + \Delta t(1-\theta)u^c u_x^c\right] \cdot \left[v + \Delta t\theta\left(u_x^+ v + u^+ v_x\right)\right] dx = 0$$

This is a nonlinear equation in u^+. To solve it we use a fixed point iteration. Given $u^{(0)} = u^{(0)}(t+\Delta t)$, iterate $u^{(k+1)}(t+\Delta t) = u^{(k+1)}$ satisfies

$$\int_\Omega \left[u^{(k+1)} - u^c + \Delta t\theta u^{(k)} u_x^{(k+1)} + \Delta t(1-\theta)u^c u_x^c\right] \cdot \left[v + \Delta t\theta\left(u_x^{(k)} v + u^{(k)} v_x\right)\right] dx = 0$$

for $k = 0, 1, \ldots$.

2.4.2 The PDESOLVE code

Since we are doing a fixed point iteration to solve a nonlinear problem within each time step, we need several extra functions.

```
Function u_next(1,Scalar,Lagrange(order));
Function u_store(1,Scalar,Lagrange(order));
Function u_update(1,Scalar,Lagrange(order));
```

Within the fixed point iteration, `u_next` will represent the function $u^{(k)}$ (initially u^c), and `u_update` represents $u^{(k+1)}$. Thus, `u_update` must always be UNKNOWN, so when we solve the system, we will assign the solution

to u_next. To test for convergence we compare the new value of u_next to its previous value which we have stored in u_store. Here is the code for the time-stepping loop with the fixed point iteration inside.

```
for (int iStep = 1; iStep <= nSteps; iStep++)
  {
  u_next = u_current;     // set up next time step
  bool converged = false;
  int iterations = 0;
  Real eps;
  do {                    // fixed point iteration
     u_store = u_next;    // save result of previous iterate
     w = Integral( (u_update + tt*u_next*dx*u_update)
         * (v + tt*(dx*u_next*v+u_next*dx*v) ))
         == Integral( (u_current - tt*(1.-theta)
                                  *u_current*dx*u_current)
            *  (v + tt*(dx*u_next*v+u_next*dx*v) ));
     ReplaceBC bc = ReplaceBC("x=0", v*(u_update-0.5))
            && ReplaceBC("x=L", v*(u_update+0.5));
     LinearProblem burgers(mesh, w, bc, u_update, v);
     u_next = burgers.solve();
     iterations++;
     eps = myL2Norm(u_next-u_store, mesh, glOrder);
     if (eps < step_tol)
        converged = true;
     } while (!converged && iterations < max_iters);
  u_current = u_next;
  }
```

For the complete code see the file *Burgers.cpp*.

2.4.3 Use of Reference Functions.

Notice that a consequence of time-stepping and the use of a fixed point iterative algorithm is the repeated construction and assembly of the matrix and right hand side vector of the FE problem with each invocation of LinearProblem. For sufficiently large problems this can result in dramatic increases in memory and CPU usage costs. This is a terrible waste

because it should be noted that really all that is changing is the value of the function. The actual equations and boundary conditions don't alter at all. PDESOLVE has a convenient way of reducing this waste by the use of reference functions. As their name suggests these functions refer to the functions to which they are pointing. Any changes made to the parent function is immediatly reflected in the reference function. If one of these reference functions is put into the expression that is to be discretized, then as the parent function is updated in each timestep the reference function and hence the expression and discretizer is immediately updated as well, without having to be recreated! We just need to do a new solve. This also makes the resulting PDESOLVE code remarkably clean and easy to maintain.

Let us walk through the use of PDESOLVE reference functions in the solution of the Burgers' equation discussed in the last subsection. The first change is that for every function whose updating will effect the statement of the problem, a reference function is defined:

```
Function u_current_ref = Function(u_current,REFERENCE);
Function u_next_ref = Function(u_next,REFERENCE);
```

The next change is that the equations and boundary conditions are defined *before* entering the time stepping and iterative loop! The `LinearProblem` is invoked before the time stepping loop as well.

```
u_next = u_current;
Expr w = Integral(u_update*v + tt*u_next_ref*dx*u_update*v
 + tt*dx*u_next_ref*u_update*v + tt*u_next_ref*u_update*dx*v
 + tt*tt*u_next_ref*dx*u_next_ref*dx*u_update*v
 + tt*tt*u_next_ref*u_next_ref*dx*u_update*dx*v) ==
 Integral((u_current_ref
 - time_step*(1.-theta)*u_current_ref*dx*u_current_ref) *
 (v + tt*(dx*u_next_ref*v + u_next_ref*dx*v)));
ReplaceBC bc = ReplaceBC("x=0",v*(u_update-0.5)) &&
  ReplaceBC("x=L",v*(u_update+0.5));
LinearProblem burgers1d(mesh, w, bc, u_update, v);
```

Note the use of `u_current_ref` and `u_next_ref` where ever `u_current` or `u_next` would appear in the expression `w`.

And now the time stepping and iterative loop becomes extraordinarily simple.

```
for (Real time=0; time<=time_final;
     time=time+time_step) {
  u_next = u_current;
  converged = false;
  int iterations = 0;
  do
    {
 u_store = u_next;
 u_next = burgers1d.solve();
 iterations++;
 Function errFunc = myL2Norm(u_store-u_next,mesh,4);
 Real eps = l2Norm(errFunc);
 if (fabs(eps) < 1.0e-12)
   converged = true;
    }
  while (!converged && iterations < 20);
  u_current = u_next;
}
```

The full listing of this code may be found in the file *Burgers_ref.cpp*.

This use of reference functions results in CPU usage savings even for a small problem. Without them the solution to the Burger's equation on a Pentium 200 running Linux took 109 seconds to complete 20 time steps, while with them the same machine took only 78 seconds.

Chapter 3

Viewing Your Results

PDESOLVE provides a set of `DataWriter` objects which can output data in a variety of formats including ASCII, MATLAB, VTK and TecPlot. The syntax is the same for all the different `DataWriters`. Currently only first order Lagrange polynomials can be written using these objects. Suppose we have solved a problem using second order Lagrange polynomials and we wish to view our solution u. Since we used quadratic polynomials, we must first project the solution onto a first order basis. The following lines illustrate how we would use an `ASCIIDataWriter` to produce a "human-readable" output file.

```
Function view = FEInterpolation(u, mesh, 1);
ASCIIDataWriter txtout("output.dat");
txtout.addScalarData(view,"U");
txtout.writeData();
```

The first line "projects" our solution onto a basis of first order Lagrange polynomials. The second line creates an `ASCIIDataWriter` object which will write its results to the file named *output.dat*. On the third line we tell the `DataWriter` to output the values of the scalar-valued function `view` and to assign the label "U" to the results. When we look at the file that the `DataWriter` produces, we will see this label on the column with the function values. If we were using a `VTKDataWriter` or a `TECPlotDataWriter`, the labels we assign would appear when we use the visualization package to view our results. The fourth line causes the `DataWriter` to write the

data to the file. We can add more data, including vector data, if desired. For example if we have a vector-valued function named v which we wish to view, we could insert the following lines between the third and fourth lines above:

```
Function viewV1 = FEInterpolation(v[0], mesh, 1);
Function viewV2 = FEInterpolation(v[1], mesh, 1);
Function viewV3 = FEInterpolation(v[2], mesh, 1);
txtout.addVectorData(viewV1, viewV2, viewV3, "V");
```

In the output file, there will be a column labeled "U" containing the values of the function u. Beside that column there will be columns containing the components of the values of the function v.

3.1 "Human-readable" Output Format

The "human-readable" output produced by the four lines of code above contains the value of the function at all of the mesh points, and a listing of all of the triangles used. Note that in one-dimensional problems, the "triangles" are lines, and in three-dimensional problems, the triangles are on the surface of the domain. For example, suppose we have solved the Poisson equation in one dimension using a mesh with eleven equally spaced nodes, and we output the solution using an `ASCIIDataWriter` to the file named *Poisson1d.txt*. The contents of this file will be as follows:

The first section in the file is the function output section. This section displays the results of the numerical solution of the PDE. The first column of the function output section is an index number for each of the mesh points. The next column labeled "X" is the mesh point, and the third column is the value of the numerical solution at each point. Note that the heading on this column is the label we assigned using the `addScalarData()` method.

The second section is a listing of the triangles used in the FEM solution. The first column is an index number for each triangle and the next two

or three columns are the locations of the points specifying each triangle. In this example, the "triangles" are lines, so the points listed are the endpoints of each line segment. Thus, the endpoints of line segment 0 are mesh points 0 and 1, the endpoints of line segment 1 are points 1 and 2, etc. In the future when support for outputting second order functions is added, three points will be required to specify the line segments: the two endpoints and the midpoint.

The output is similar for two dimensional problems. For example, solving a problem on the unit square using three mesh points in each direction, the output file would look similar to the following:

In the function output section, the first column is again an index of the mesh points, the second and third columns are the x and y components of the mesh points, and the fourth column is the function value at each point. In the second section, each triangle is specified by its three corner points.

3.2 MATLAB Output Format

The output produced by a `MatlabDataWriter` is very similar to the format described above. The numerical data will have the same format; the only difference will be in the header lines. There are several MATLAB script files provided with your PDESOLVE distribution to help you view your output quickly. These scripts are somewhat limited; e.g., you cannot use them to view vector-valued functions. You are free to modify them to provide more sophisticated viewing capabilities. To use these scripts you must make sure they are in a directory where MATLAB can find them. For example, if they are in the directory */home/userid/pdesolve/matlab* , you must make sure this directory is in MATLAB's search path. If it isn't you can add it to the beginning of the search path using the MATLAB command:

```
path('/home/userid/pdesolve/matlab', path);
```

3.2.1 The PDEPlot Script

To display your FEM output in MATLAB, you can use the *PDEPlot* script provided with PDESOLVE.

In your PDESOLVE program, use a `MatlabDataWriter` to save your output in a file called, say, *output.matlab*. Then in MATLAB, display the results by typing

`PDEPlot('output.matlab').`

You may then want to use some of the following MATLAB commands to manipulate your plot: `color`, `colormap`, `view`, `shading`, `mesh`, `surf`, `surfc`.

You can also use the MATLAB `print` command to produce an Encapsulated Postscript file for including in a TEXdocument. For example, Figures 2.2 and 2.3 were produced as follows:

First, the following lines were added to the file Poisson2d.cpp:

```
Function view = FEInterpolation(u, mesh, 1);
MatlabDataWriter matout("poisson2d.matlab");
matout.addScalarData(view,"U");
matout.writeData();
```

Then the following commands were entered in a MATLAB session:

```
PDEPlot('poisson2d.matlab')
print poisson2d -depsc
view(2)
axis('square')
print poisson2d.view2.eps -depsc
```

For more details on the MATLAB commands available, type

```
help graph3d
```

in your MATLAB session.

3.2.2 The PDEMovie Script

This script is for animating the solution of a time-dependent problem. There is a script file called mymovie.m in /pdesolve/matlab. You can use it to animate the results of the programs described in the time dependent section of the *User's Guide*.

3.3 VTK Output Format

This format is almost identical to the Matlab or Ascii formats. With the Tcl scripts provided in /pdesolve/VTK directory these output files can be used inside the *Visualization ToolKit* Tcl/Tk interpreter to visualize the results of PDESOLVE calculations.

3.4 TECPlot Output Format

TecPlot is a menu driven commercially available graphics package. The `TECPlot` output format produces an ascii file that can be used inside the *TecPlot* environment to read in and display the data from the PDESOLVE calculations.

Chapter 4

Using More Complicated Domains

PDESOLVE provides the function `rectMeshGen()` for building a rectangular domain and generating a simple mesh on it. There are many cases however, which require that we have more control over the domain.

- We may want to specify different boundary conditions on two parts of the same edge of a square.
- We may want to generate a mesh which has more elements concentrated in some subregion of our domain.
- We may have a complicated domain (nonrectangular or having holes in its interior).
- We may want to design an optimal shape. This means that we need to be able to change the domain programmatically.

In this chapter we describe the classes provided by PDESOLVE for building a domain and generating a mesh on it. Currently there are two ways of specifying the geometry of a domain: It can be built up from individual components in a C++ program, or a description of its parts can be read in from a file.

4.1 Geometry Classes in PDESOLVE

The domain will be an instance of a `ParameterizedGeometry` class, either a `ParameterizedGeometry2D` or a `ParameterizedGeometry3D`. Currently only the two dimensional geometry classes are available. A two dimensional geometry is constructed from a collection of objects of the class `Part2D`. A `Part2D` is an object whose boundary is composed of objects derived from `Entity2D`. Currently a geometry can contain only one part. The entities which are available are: `LabeledPoint2D`, `Origin2D`, `SimpleLine2D` (either horizontal or vertical), `Line2D` (a free angle line), `Arc2D` (a section of an elliptical arc) and `Nurbs2D` (a more general curve). Each of these entities has certain parameters which determine its form. These parameters have names which we can use to set their values. In addition, we can change the names in order to distinguish a parameter belonging to one entity from the same parameter belonging to another entity of the same class. If we wish to be able to change the geometry after it is initially constructed, we must "publish" the names of these parameters so that we have access to them from within our C++ program. In this section we describe each of the entities and their parameters. In the following sections we show how to use these entities to build parts.

4.1.1 LabeledPoint2D

This class is provided as a means of assigning a label to a specific point. It has no parameters and takes up no space. Its position is the previous entity's endpoint.

4.1.2 Origin2D

An `Origin2D` is a 2D point used to parameterize the origin of either a part or a hole within a part. It allows the part builder to build parts and holes that can be moved through geometry parameters, or that refer to other parts so that they can update themselves in response to movement of other parts. If no `Origin2D` is specified, the part will begin at $(0, 0)$. Even if the part is originally intended to begin at $(0, 0)$ it is necessary to specify an `Origin2D` if the part will be moved at a later stage.

An `Origin2D` has two geometry parameters, initially named "`X`" and "`Y`". These parameters are x and y vector components or point positions. Neither parameter is published by default. An `Origin2D` may be parameterized in three ways:

- `FIXED` means `X` and `Y` represent x and y vectors from the local coordinate system. In this case `X` and `Y` cannot be changed.
- `RELATIVELOCAL` means `X` and `Y` represent x and y vectors from the local coordinate system that may be changed.
- `RELATIVEENTITY` means `X` and `Y` represent x and y vectors from a given `Entity2D`'s origin. This is the means for making one part's origin relative to another part.

Note that `Origin2D` is only logically useful as the very first `Entity2D` of a part or hole. Inserting it anywhere else has unpredictable results.

4.1.3 SimpleLine2D

A `SimpleLine2D` is either horizontal or vertical. It can be parameterized in one of two ways: `GIVENLENGTH` (the default) or `FIXEDEXTENT`. A `SimpleLine2D` has one parameter. For `GIVENLENGTH`, this parameter represents the actual length of the line, and is initially called "`length`". For `FIXEDEXTENT`, this parameter represents the endpoint of the line, in either `X` or `Y` (depending on whether the line is horizontal or vertical). For `FIXEDEXTENT` the geometry parameter is initially named "`extent`". Note that the fixed (x, y) point is referenced to the entity's local coordinate space (implicitly $(0, 0)$ if not specified by an `Origin2D` at the `Part2D` level).

4.1.4 Line2D

A `Line2D` is a line which can form any angle with the horizontal. It can be parameterized in one of three ways: `FIXED`, `FREEVECTOR` or `FREEANGLE`, with the default `FREEVECTOR`. A `Line2D` has two parameters:

- For FREEANGLE: the two parameters represent the angle of the line and its length, and are initially named "theta" and "length", respectively. The angle 0 is at the X axis, with increasing angles moving counter-clockwise. It is specified in radians, and must be between 0 and 2π.
- For FREEVECTOR: the two parameters represent the components of the vector from the line's origin and are initially named "X" and "Y", respectively.
- For FIXED: the parameters represent the fixed endpoint of the line, and are named "X" and "Y", respectively. Note that the fixed (x, y) point is referenced to the entity's local coordinate space (implictly $(0, 0)$ if not specified by an Origin2D at the Part2D level).

4.1.5 Arc2D

An Arc2D is a section of an elliptical arc. To construct an arc, you specify the size of the arc, either QUARTER, HALF, THREEQUARTER or FULL, the lengths of the semi-axes, the direction of the arc (CW for clockwise or CCW for counter-clockwise), and the orientation of the center of the ellipse from the vertex of the major axis (the center can be UP, DOWN, LEFT or RIGHT from the vertex).

An Arc2D is parameterized by the lengths of the semi-axes, given by parameter names "lengthA" and "lengthB".

The vertex of the major axis is implicitly at $(0, 0)$, but within a part, the vertex will be the previous entity's end point. The center point is computed from the given axis lengths.

4.1.6 Nurbs2D

A Nurbs2D is a nonuniform rational B-spline. Three pieces of data are required to construct a nurbs:

- The nurbs degree (must be greater than or equal to one).

- An array of weighted control points (x, y, w) 3D `Coords` (Note the constructor does *not* take homogeneous points (xw, yw, w))
- A knot vector (array of reals), if specified.

If you use the constructor that does not take knots, it will generate a clamped, uniform (equally-spaced inner knots) nurbs. If you want a (rational) Bezier curve (no inner knots), send in the appropriate control points with the correct degree (i.e., the number of control points is one greater than the degree), and this constructor will make it for you.

The (length of the knot vector - 1) must equal the (degree + the number of control points). The first control point must begin at (0,0). The control point (x, y) values represent x,y vectors from the current entity origin.

Construction assumptions:

- knots is an increasing sequence, with all knots normalized so that $0 \leq K_i \leq 1$ where K_i is the i^{th} knot. (Note that K_i may equal K_{i+1})
- control point weights are > 0
- the first control point (x, y) is at (0,0)

A `Nurbs2D` makes the following parameters available: "PiX", "PiY" and "PiW" which represent the ith (indexed from 0) control point's X, Y and Weight value, respectively. The values are not homogeneous, so, e.g., the third control point with (x,y,w) = 2,4,3 will have parameters P2X==2, P2Y==4 and P2W==3. None are published by default, so the user must publish them to change them.

For details about nurbs, see *The NURBS Book*.

4.2 Constructing the Domain Programmatically

In this section we work through several examples to show how to use the entities described above to construct a domain within a C++ program. A

`Part2D` is constructed by "stringing together" several `Entity2Ds`. For each entity we must set values of the appropriate parameters, set a label for the entire entity (for use in specifying boundary conditions) and set a pointer to the next entity in the part. Once we have constructed the domain, we can use a mesh generator to create a mesh.

4.2.1 A Square

For the first example, we take our domain to be the unit square. The twist is that we want to be able to specify a Dirichlet BC on a part of the $x = 1$ edge and a Neumann BC on the rest of that edge.

First we need to declare some pointers to `Entity2Ds`:

```
Entity2D* root;
Entity2D* current;
Entity2D* prev;
```

We use `root` to point to the first entity so that we can use it later to make a part. We use `prev` and `current` to make one entity point to the next.

The first entity is a horizontal line of unit length with its initial endpoint at the origin. We want to impose a Dirichlet BC on this part of the boundary. We create this entity as follows:

```
root = new SimpleLine2D(SimpleLine2D::HORIZONTAL);
root->getParameter("length").setValue(1.0);
root->setLabel("dirichlet");
prev = root;
```

Note that in order to set the value of a parameter, we must know how it is named initially. The `length` parameter can be set to either a positive or a negative value. A positive value means that the final endpoint is to the right of (or, for a vertical line, above) the initial endpoint, and a negative value means that the final endpoint is to the left of (or below) the initial endpoint.

The next entity is a vertical line segment of length 1/2. This is the part of the boundary on which we wish to impose the Neumann BC.

```
current = new SimpleLine2D(SimpleLine2D::VERTICAL);
current->getParameter("length").setValue(0.5);
current->setLabel("neumann");
prev->setNext(current);
prev = current;
```

Note that the initial endpoint of this line is the final endpoint of the previous entity. (This is accomplished by the `regenerate()` method of the `Part2D`. See below.)

Next we construct the other half of this edge:

```
current = new SimpleLine2D(SimpleLine2D::VERTICAL);
current->getParameter("length").setValue(0.5);
current->setLabel("dirichlet");
prev->setNext(current);
prev = current;
```

The next two edges each have a length of -1 since we move to the left and then down to complete the construction of the square. When we are done, we close the loop by setting the last entity to point to the first one (`root`) as the next entity in the loop.

```
current = new SimpleLine2D(SimpleLine2D::HORIZONTAL);
current->getParameter("length").setValue(-1.0);
current->setLabel("dirichlet");
prev->setNext(current);
prev = current;

current = new SimpleLine2D(SimpleLine2D::VERTICAL);
current->getParameter("length").setValue(-1.0);
current->setLabel("dirichlet");
prev->setNext(current);

current->setNext(root);
```

Once we have put together all the entities, we use them to form a part which we then add to the geometry:

```
Part2D square(root);

ParameterizedGeometry2D geom;
geom.addPart(square);
geom.regenerate();
```

The `regenerate()` method traverses the loop (using the pointers we set above using the `setNext()` method) and adjusts the endpoints of each entity so that each one is connected to the next.

Now that we have constructed the domain, we can create a mesh on it: Currently the only mesh generator provided by PDESOLVE is the `DelaunayMesher2D`.

```
float cellSize = 0.2;
DelaunayMesher2D mesher;
GlobalMeshSettings& settings = mesher.getMesherSettings();
settings.setValue(GlobalMeshSettings::MAX_ELEMENT_SIZE, cellSize);

CellComplex mesh = mesher.createCellComplex(geom);
```

The domain and the mesh we have generated on it are shown in Figure 4.1. Observe that there is a node at the point $(1, 0.5)$. This will always be true no matter how we set `cellSize` since that point is the endpoint of two entities.

Now that we have meshed our domain, we can solve our PDE problem just as in the examples considered in Chapter 2. For example, if we want to impose the boundary conditions

$$u = g \quad \text{on } \Gamma_1 \qquad \frac{\partial u}{\partial n} = h \quad \text{on } \Gamma_2$$

where Γ_1 and Γ_2 are the parts of the boundary labelled "dirichlet" and "neumann" above, we could use code similar to the folowing:

```
Expr eqn = Integral(...) + ... + Integral("neumann", H*v);
ReplaceBC bc("dirichlet", v*(u-G));
```

See the file *Poisson2D_geom.cpp* in the */pdesolve/examples* directory for the complete code of a problem solved using this domain.

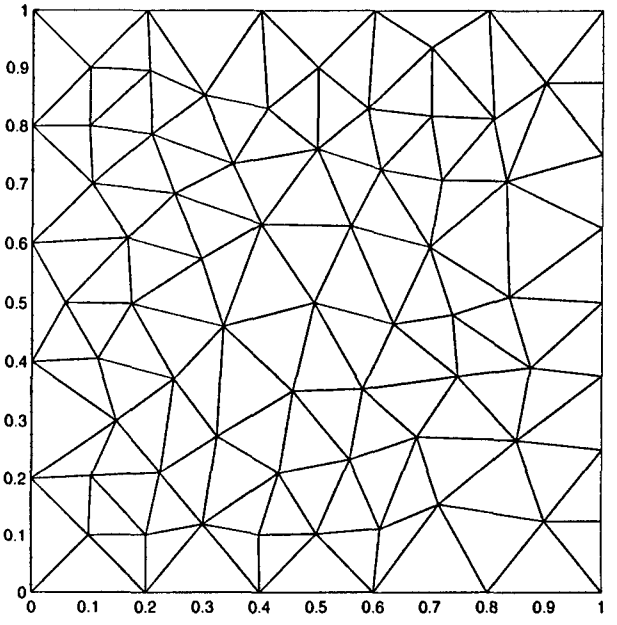

Figure 4.1: *A square domain constructed from Entity2Ds.*

4.2.2 A Wedge

As our next example, we construct a domain which is a quarter of the unit disk. Just as in the previous example, we begin by declaring `root`, `current` and `prev` to be pointers to `Entity2D` objects. The first entity is a horizontal line of length one:

```
root = new SimpleLine2D(SimpleLine2D::HORIZONTAL);
root->getParameter("length").setValue(1.0);
root->setLabel("outside");
prev = root;
```

Next we have an arc which is a quarter of a circle:

```
current = new Arc2D(1.0,1.0,Arc2D::QUARTER,Arc2D::CCW,Arc2D::LEFT);
current->setLabel("outside");
prev->setNext(current);
prev = current;
```

The first two arguments are the lengths of the semi-axes. Since they are equal we are constructing a circular arc. The vertex of the major (A) axis is at the endpoint of the previous entity; i.e., at $(1, 0)$. The last argument says that the center of the ellipse (circle) is to the left of this vertex; i.e., at the origin $(0, 0)$. To draw the arc, we move in a counter clockwise direction. Thus, the arc lies in the first quadrant, and its final endpoint is at $(0, 1)$. We complete the loop by constructing a vertical line back to the origin:

```
current = new SimpleLine2D(SimpleLine2D::VERTICAL);
current->getParameter("length").setValue(-1.0);
current->setLabel("outside");
prev->setNext(current);
prev = current;

current->setNext(root);
```

Once we have all the entities, we create a part and add it to the parameterized geometry just as in the previous example. The domain we have created (along with a mesh) are shown in Figure 4.2.

The complete code for generating this domain can be found in the file *Wedge.cpp* in the */pdesolve/examples* directory.

4.2.3 A Triangular Region

To illustrate the use of `Line2D` we construct a triangular shaped region. This example is exactly like the last one except that the arc is replaced by a line with initial endpoint at $(1, 0)$ and final endpoint at $(0, 1)$. Since we know the final endpoint of the line, we choose the parameter type to be `FIXED`. The only change from the previous example is that we replace the line of code that creates the `Arc2D` with

```
current = new Line2D(0.0, 1.0, Line2D::FIXED);
```

The region and mesh are shown in Figure 4.3.

The complete code listing can be found in *triangle.cpp*.

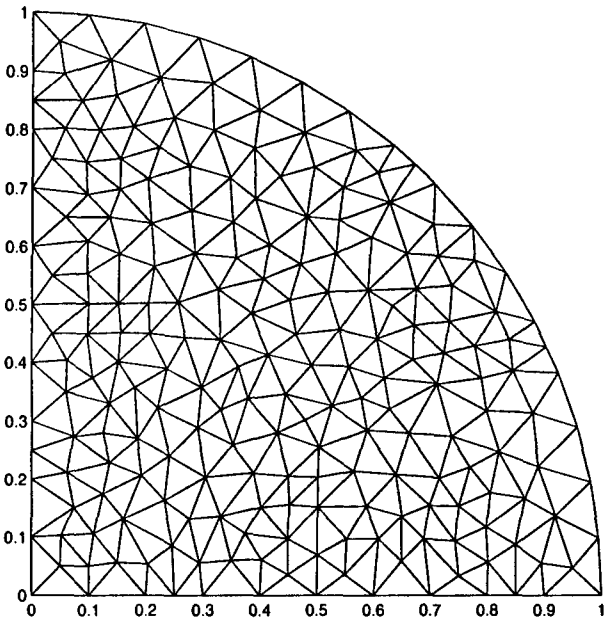

Figure 4.2: *A wedge shaped domain.*

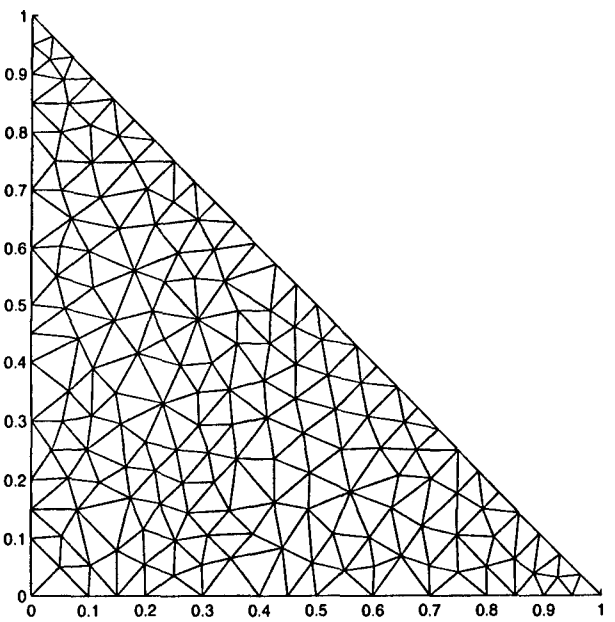

Figure 4.3: *A triangular domain.*

4.2.4 A Domain With Interior Holes

For the next example we create a domain with interior holes. We publish the parameters for the holes so that we can move them and change their sizes. We begin by creating a circular domain:

```
root = new Origin2D(1.0, 0.0);       // shift origin
root->setLabel("outside");
prev = root;

current = new Arc2D(1.0, 1.0, Arc2D::FULL, Arc2D::CCW, Arc2D::LEFT);
current->setLabel("outside");
prev->setNext(current);
current->setNext(root);
```

Since the arc will start at the origin, we shift the origin one unit to the right, then draw the arc with the center to the left of the major axis vertex. This will put the center at $(0, 0)$.

Next we create a hole which we will put inside the domain. We make the hole an ellipse, but we could make it as complicated as we like, consisting of multiple entities.

```
Entity2D* hroot;
Entity2D* hprev;
Entity2D* hcurrent;

hroot = new Origin2D(0.1,0.1);
hroot->getParameter("X").setName("X1");
hroot->publishParameter("X1");
hroot->getParameter("Y").setName("Y1");
hroot->publishParameter("Y1");
hprev = hroot;

hcurrent = new Arc2D(0.2,0.2,
                     Arc2D::FULL,Arc2D::CW,Arc2D::RIGHT);
hcurrent->setLabel("hole1");
hcurrent->setDistributionSize(0.06);
hcurrent->getParameter("lengthA").setName("A1");
```

```
hcurrent->publishParameter("A1");
hcurrent->getParameter("lengthB").setName("B1");
hcurrent->publishParameter("B1");
hprev->setNext(hcurrent);
```

We must change the names of the parameters before we publish them so that we can distinguish the parameters belonging to one hole from those belonging to other holes. By using the `setDistributionSize()` method, we can control the distribution of nodes on the hole when we generate the mesh.

Next we create an array to hold the holes:

```
Array<Entity2D*> holes(1,hroot);
```

and then make another hole. The construction of the second hole is the same as the first. We take the origin to be $(-0.2, -0.2)$ and the lengths to be 0.1 and 0.2. We then add this hole to the array:

```
holes.append(hroot);
```

We could add more holes in the same way. Next we create a `Part2D` from the outer domain and the array of holes:

```
Part2D dom(root,holes);
```

and add it to the geometry:

```
ParameterizedGeometry2D geom2D;
geom2D.addPart(dom);
```

We then `regenerate()` and create a mesh just as before. The resulting domain with the mesh is shown in Figure 4.4.

Now, using the labels we assigned, we can move and resize the holes. For this example, we leave hole two alone and change hole one as follows:

```
geom2D.getParameter("X1").setValue(.3);
geom2D.getParameter("Y1").setValue(.2);
geom2D.getParameter("A1").setValue(.2);
geom2D.getParameter("B1").setValue(.15);
```

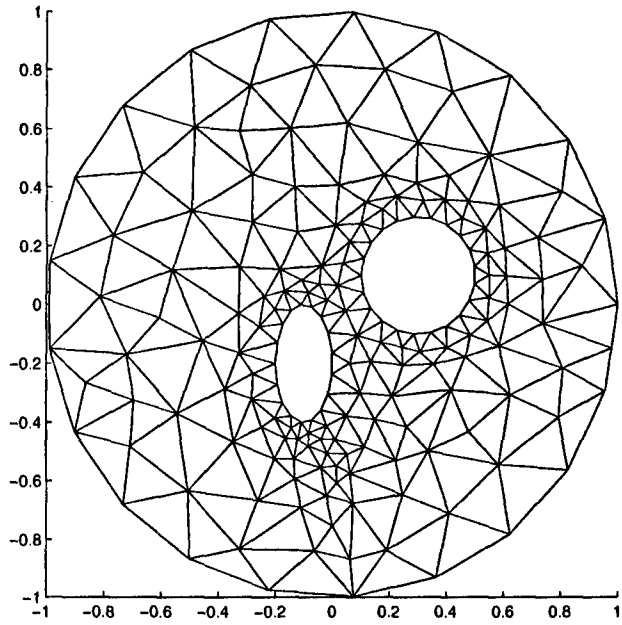

Figure 4.4: *A domain with holes.*

Again we must `regenerate()` and create a new mesh. The resulting domain is shown in Figure 4.5.

4.3 Reading in a Description of a Domain

If you want to change anything about the domain programmatically, you need to use the methods described in the previous section. To construct the domain initially, though, you can read in its description from an ASCII file as follows:

```
ifstream partFile("file.dat");
ParameterizedGeometry2D geom2D(partFile);
partFile.close();
```

where *file.dat* is a file containing the description of the part. The constructor which reads in the description of the part automatically does an initial `regenerate()`, so the part is now ready to use.

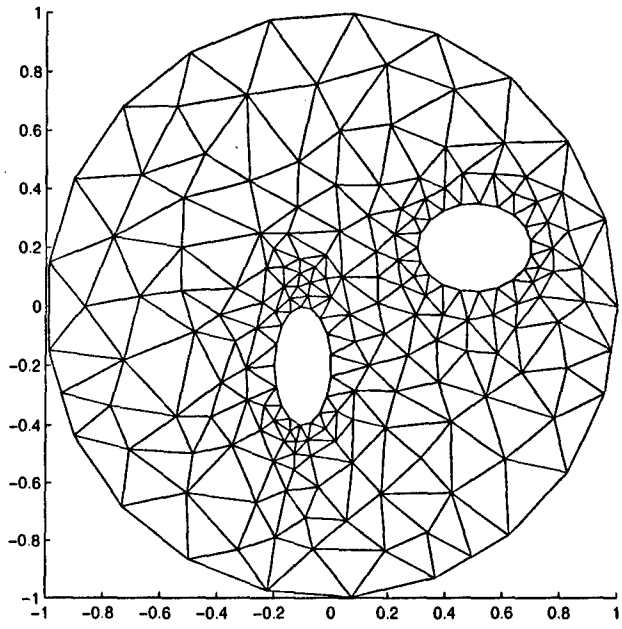

Figure 4.5: *The same domain with one hole changed.*

In the description file, blank lines and anything following a pound sign (#) are treated as comments. The file must adhere to the following format: The first line contains the number of parts (currently only one part is supported). Next comes the "part section." The first line of this section contains a character string with the name of the part. The next line has an integer value for the number of boundary entities for the part, and the third line has the number of holes in this part. For example, if the first four lines of the file look like

```
1
part1
4
0
```

then the file contains the description of one part, the part will have the label "`part1`", will be composed of four boundary entities and will contain no holes.

Within the part section there will be a section for each entity following the format:
First line – entity type:

- 0 – `Origin2D`
- 1 – `LabeledPoint2D`
- 2 – `SimpleLine2D`
- 3 – `Line2D`
- 4 – `Nurbs2D`
- 5 – `Arc2D`

Second line – entity label (string)
Third line – distribution type:

- 0 – None, use global mesher settings.
- 1 – Equal distribution. In this case you must specify the size by a real number.
- 2 – Exponential distribution. In this case, you must specify α (real), the number of points and the direction (1 for forward, 0 for backward). The number of points must be greater than or equal to two. The location of the normalized point t along the entity is given by $(e^{\alpha t} - 1)/(e^{\alpha} - 1)$.

Fourth line – number of published parameters. Following this line there is one line for each published parameter. The format of these lines is `oldname` (string) `newname` (string) `value`.
Next line(s) – entity data. The format of the entity data depends on the type:

- `Origin2D`

  `x y type entityRelativeTo`

  where `type` is `0` for `FIXED`, `1` for `RELATIVELOCAL` or `2` for `RELATIVEENTITY`. The `entityRelativeTo` is the number of the entity from the top of the file. ***(Not yet supported.)***

- `LabeledPoint2D` Nothing needed.
- `SimpleLine2D`

  `lineType parameterType parameterValue`

  where `lineType` is 0 for `HORIZONTAL`, 1 for `VERTICAL`, and `parameterType` is 0 for `GIVENLENGTH`, 1 for `FIXEDEXTENT`.
- `Line2D`

  `param1Value param2Value parameterType`

  where `parameterType` is 0 for `FREEVECTOR`, 1 for `FREEANGLE` or 2 for `FIXED`.
- `Nurbs2D`. In this case several lines are required for the entity data. The first line has the degree, the second has the number of control points, and the third has the number of knots. Following this there must be one line for each control point containing the `x`, `y` and `w` coordinates. The data for the knots follow the data for the control points.
- `Arc2D`

  `lengthA lengthB arcSize arcDirection centerOrientation`

  where `arcSize` is 0 for `QUARTER`, 1 for `HALF`, 2 for `THREEQUARTER` or 3 for `FULL`, `arcDirection` is 0 for `CW`, 1 for `CCW`, and `centerOrientation` is 0 for `UP`, 1 for `DOWN`, 2 for `LEFT` or 3 for `RIGHT`.

Here is an example of an entity section:

```
3               # Line2D
y=0             # label is "y=0"
1 0.3           # dist type: equal with size 0.3
1               # one published parameter
X X1 1.0        # publish parameter "X" under the name "X1"
1.0 1.0 2       # type = FIXED, endpoint is (1,1)
```

Note that the value of the parameter `X` is actually specified twice: once on the line specifying the published parameter and again on the last line giving the entity data. In this example the same value is specified in both places. If these values should differ, the one given on the "publish parameters" line takes precedence.

Below is the complete listing of a description file. The geometry constructed from this file is shown in Figure 4.6.

```
1                # 1 part
part1            # name of part 1
4                # 4 boundary entities
0                # no holes
3                # first entity -- Line2D --------------------------
y=0              # label "y=0"
0                # dist type - none, use global mesher settings
0                # no published parameters
1 0 2            # type = FIXED, endpoint is (1,0)
4                # second entity - Nurbs2D -------------------------
x=L              # label "x=L"
1 0.075          # dist type - equal, size = 0.075
2                # 2 published parameters
P1X myx 0.4      # old name = "P1X", new name = "myx", value = 0.4
P1Y myy 0.2      # old name = "P1Y", new name = "myy", value = 0.2
3                # degree = 3
4                # number of control points (degree + 1)
0                # number of knots
0 0 1            # first control point
0.4 0.2 3        # second (note values of parameters same as above)
0.4 0.8 7        # third
0 1 1            # fourth
2                # third entity - SimpleLine2D --------------------
y=L              # label "y=L"
0                # dist type
0                # no published parameters
0 0 -1.0         # lineType = HORIZONTAL,
                 #    parameterType = GIVENLENGTH, length -1.
2                # fourth entity - SimpleLine2D ---------------------
x=0              # label "x=0"
0                # use global mesher settings for distribution
0                # no published parameters
1 0 -1.0         # vertical, length = -1.
```

Observe that the X and Y parameters for the second control point are published so that they can be changed. The file *ReadGeometry.cpp* in the */pdesolve/examples*

directory reads in this file then uses the published parameters to change the region.

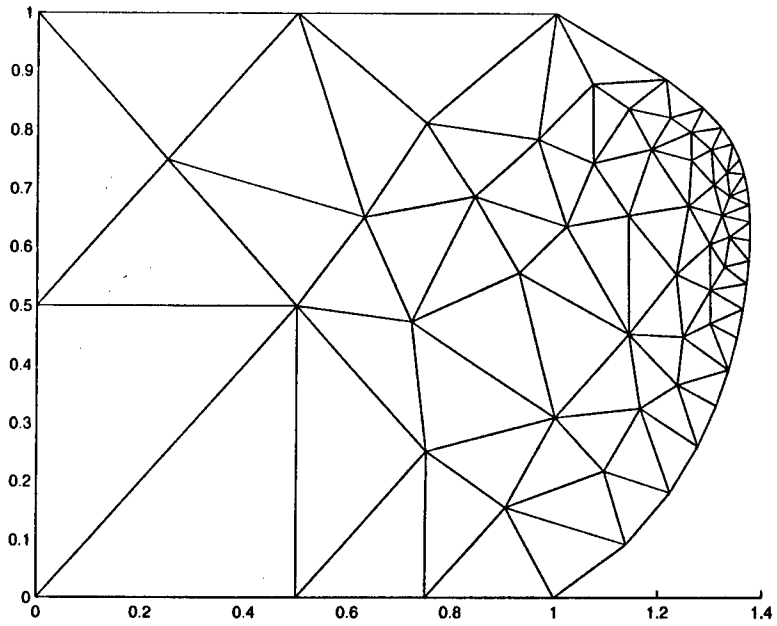

Figure 4.6: *A domain constructed by reading in an ASCII file.*

Reference

Les Piegl and Wayne Tiller. *The NURBS Book.* 2nd Ed. Springer-Verlag, Berlin, 1997.