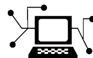
Partial Differential Equation Toolbox™ 1
User's Guide

MATLAB®

How to Contact The MathWorks



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Partial Differential Equation Toolbox™ User's Guide

© COPYRIGHT 1995–2010 The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

August 1995	First printing	New for Version 1.0
February 1996	Second printing	Revised for Version 1.0.1
July 2002	Online only	Revised for Version 1.0.4 (Release 13)
September 2002	Third printing	Minor Revision for Version 1.0.4
June 2004	Online only	Revised for Version 1.0.5 (Release 14)
October 2004	Online only	Revised for Version 1.0.6 (Release 14SP1)
March 2005	Online only	Revised for Version 1.0.6 (Release 14SP2)
August 2005	Fourth printing	Minor Revision for Version 1.0.6
September 2005	Online only	Revised for Version 1.0.7 (Release 14SP3)
March 2006	Online only	Revised for Version 1.0.8 (Release 2006a)
March 2007	Online only	Revised for Version 1.0.10 (Release 2007a)
September 2007	Online only	Revised for Version 1.0.11 (Release 2007b)
March 2008	Online only	Revised for Version 1.0.12 (Release 2008a)
October 2008	Online only	Revised for Version 1.0.13 (Release 2008b)
March 2009	Online only	Revised for Version 1.0.14 (Release 2009a)
September 2009	Online only	Revised for Version 1.0.15 (Release 2009b)
March 2010	Online only	Revised for Version 1.0.16 (Release 2010a)

Contents

1 Getting Started

Product Overview **1-2**
Introduction **1-2**
Can I Use Partial Differential Equation Toolbox Software? **1-2**
What Problems Can I Solve? **1-3**
In Which Areas Can Partial Differential Equation Toolbox Software Be Used? **1-5**
How Do I Define a PDE Problem? **1-6**
How Can I Solve a PDE Problem? **1-6**
Can I Use Partial Differential Equation Toolbox Software for Nonstandard Problems? **1-7**
How Can I Visualize My Results? **1-7**
Are There Any Applications Already Implemented? **1-7**
Can I Extend the Functionality of the Software? **1-8**
How Can I Solve 3-D Problems by 2-D Models? **1-8**

Solving a PDE **1-10**

Basics of the Finite Element Method **1-22**

Using the pdetool GUI **1-27**
Introduction **1-27**
The Menus **1-29**
The Toolbar **1-30**
The GUI Modes **1-31**
The CSG Model and the Set Formula **1-32**
Creating Rounded Corners **1-33**
Suggested Modeling Method **1-35**
Object Selection Methods **1-39**
Display Additional Information **1-40**
Entering Parameter Values as MATLAB Expressions **1-40**
Using Earlier Version Partial Differential Equation Toolbox Model Files **1-41**

Using Command-Line Functions **1-42**
Introduction **1-42**
Data Structures and Utility Functions **1-42**
Hints and Suggestions for Using Command-Line Functions **1-47**

Common PDE Problems **1-49**
Elliptic Problems **1-49**
Parabolic Problems **1-64**
Hyperbolic Problem **1-71**
Eigenvalue Problems **1-75**
Application Modes **1-84**
References **1-118**

2 Graphical User Interface

Using the pdetool Menus **2-2**
Introduction **2-2**
File Menu **2-3**
Edit Menu **2-5**
Options Menu **2-7**
Draw Menu **2-10**
Boundary Menu **2-12**
PDE Menu **2-16**
Mesh Menu **2-20**
Solve Menu **2-22**
Plot Menu **2-27**
Window Menu **2-34**
Help Menu **2-34**

The Toolbar **2-35**

3 Finite Element Method

The Elliptic Equation **3-2**

The Elliptic System **3-10**

The Parabolic Equation **3-13**
Reducing the Parabolic Equation to Elliptic Equations . . . **3-13**
Solving the Parabolic Equation in Stages **3-15**

The Hyperbolic Equation **3-18**

The Eigenvalue Equation **3-19**

Nonlinear Equations **3-23**

Adaptive Mesh Refinement **3-29**
Introduction **3-29**
The Error Indicator Function **3-30**
The Mesh Refiner **3-31**
The Termination Criteria **3-31**

Fast Solution of Poisson's Equation **3-32**

References **3-34**

4 Function Reference

PDE Algorithms **4-1**

User-Interface Algorithms **4-2**

Geometry Algorithms **4-2**

Plots 4-3

Utility Algorithms 4-3

User-Defined Algorithms 4-4

5 Functions — Alphabetical List

Index

1

Getting Started

- "Product Overview" on page 1-2
- "Solving a PDE" on page 1-10
- "Basics of the Finite Element Method" on page 1-22
- "Using the pdetool GUI" on page 1-27
- "Using Command-Line Functions" on page 1-42
- "Common PDE Problems" on page 1-49

Product Overview

In this section...

"Introduction" on page 1-2

"Can I Use Partial Differential Equation Toolbox Software?" on page 1-2

"What Problems Can I Solve?" on page 1-3

"In Which Areas Can Partial Differential Equation Toolbox Software Be Used?" on page 1-5

"How Do I Define a PDE Problem?" on page 1-6

"How Can I Solve a PDE Problem?" on page 1-6

"Can I Use Partial Differential Equation Toolbox Software for Nonstandard Problems?" on page 1-7

"How Can I Visualize My Results?" on page 1-7

"Are There Any Applications Already Implemented?" on page 1-7

"Can I Extend the Functionality of the Software?" on page 1-8

"How Can I Solve 3-D Problems by 2-D Models?" on page 1-8

Introduction

The objectives of Partial Differential Equation Toolbox™ software are to provide you with tools that:

- Define a PDE problem, e.g., define 2-D regions, boundary conditions, and PDE coefficients.
- Numerically solve the PDE problem, e.g., generate unstructured meshes, discretize the equations, and produce an approximation to the solution.
- Visualize the results.

Can I Use Partial Differential Equation Toolbox Software?

Partial Differential Equation Toolbox software is designed for both beginners and advanced users.

The minimal requirement is that you can formulate a PDE problem on paper (draw the domain, write the boundary conditions, and the PDE). At the MATLAB® command line, type

```
pdetool
```

This invokes the graphical user interface (GUI), which is a self-contained graphical environment for PDE solving. For common applications you can use the specific physical terms rather than abstract coefficients. Using `pdetool` requires no knowledge of the mathematics behind the PDE, the numerical schemes, or MATLAB. "Solving a PDE" on page 1-10 guides you through an example step by step.

Advanced applications are also possible by downloading the domain geometry, boundary conditions, and mesh description to the MATLAB workspace. From the command line, or from MATLAB files, you can call functions to do the hard work, e.g., generate meshes, discretize your problem, perform interpolation, plot data on unstructured grids, etc., while you retain full control over the global numerical algorithm.

What Problems Can I Solve?

The basic equation addressed by the software is the PDE

$$-\nabla \cdot (c\nabla u) + au = f$$

expressed in Ω, which we shall refer to as the *elliptic equation,* regardless of whether its coefficients and boundary conditions make the PDE problem elliptic in the mathematical sense. Analogously, we shall use the terms *parabolic equation* and *hyperbolic equation* for equations with spatial operators like the previous one, and first and second order time derivatives, respectively. Ω is a bounded domain in the plane. c, a, f, and the unknown u are scalar, complex valued functions defined on Ω. c can be a 2-by-2 matrix function on Ω. The software can also handle the parabolic PDE

$$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f$$

the hyperbolic PDE

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f$$

and the eigenvalue problem

$$-\nabla \cdot (c\nabla u) + au = \lambda du$$

where d is a complex valued function on Ω, and λ is an unknown eigenvalue. For the parabolic and hyperbolic PDE the coefficients c, a, f, and d can depend on time. A nonlinear solver is available for the nonlinear elliptic PDE

$$-\nabla \cdot (c(u)\nabla u) + a(u)u = f(u)$$

where c, a, and f are functions of the unknown solution u.

Note Before solving a nonlinear PDE, from the **Solve** menu in the `pdetool` GUI, select **Parameters**. Then, select the **Use nonlinear solver** check box and click **OK**.

All solvers can handle the system case

$$-\nabla \cdot (c_{11}\nabla u_1) - \nabla \cdot (c_{12}\nabla u_2) + a_{11}u_1 + a_{12}u_2 = f_1$$
$$-\nabla \cdot (c_{21}\nabla u_1) - \nabla \cdot (c_{22}\nabla u_2) + a_{21}u_1 + a_{22}u_2 = f_2$$

You can work with systems of arbitrary dimension from the command line. For the elliptic problem, an adaptive mesh refinement algorithm is implemented. It can also be used in conjunction with the nonlinear solver. In addition, a fast solver for Poisson's equation on a rectangular grid is available.

The following boundary conditions are defined for scalar u:

- *Dirichlet*: $hu = r$ on the boundary $\partial\Omega$.
- *Generalized Neumann*: $\vec{n} \cdot (c\nabla u) + qu = g$ on $\partial\Omega$.

$\vec{n}$ is the outward unit normal. g, q, h, and r are complex-valued functions defined on $\partial\Omega$. (The eigenvalue problem is a homogeneous problem, i.e., $g = 0$,

$r = 0$.) In the nonlinear case, the coefficients g, q, h, and r can depend on u, and for the hyperbolic and parabolic PDE, the coefficients can depend on time. For the two-dimensional system case, Dirichlet boundary condition is

$$h_{11}u_1 + h_{12}u_2 = r_1$$
$$h_{21}u_1 + h_{22}u_2 = r_2$$

the generalized Neumann boundary condition is

$$\vec{n} \cdot (c_{11}\nabla u_1) + \vec{n} \cdot (c_{12}\nabla u_2) + q_{11}u_1 + q_{12}u_2 = g_1$$
$$\vec{n} \cdot (c_{21}\nabla u_1) + \vec{n} \cdot (c_{22}\nabla u_2) + q_{21}u_1 + q_{22}u_2 = g_2$$

and the *mixed* boundary condition is

$$h_{11}u_1 + h_{12}u_2 = r_1$$
$$\vec{n} \cdot (c_{11}\nabla u_1) + \vec{n} \cdot (c_{12}\nabla u_2) + q_{11}u_1 + q_{12}u_2 = g_1 + h_{11}\mu$$
$$\vec{n} \cdot (c_{21}\nabla u_1) + \vec{n} \cdot (c_{22}\nabla u_2) + q_{21}u_1 + q_{22}u_2 = g_2 + h_{12}\mu$$

where μ is computed such that the Dirichlet boundary condition is satisfied. Dirichlet boundary conditions are also called *essential* boundary conditions, and Neumann boundary conditions are also called *natural* boundary conditions. See Chapter 3, "Finite Element Method" for the general system case.

In Which Areas Can Partial Differential Equation Toolbox Software Be Used?

The PDEs implemented in Partial Differential Equation Toolbox software are used as a mathematical model for a wide variety of phenomena in all branches of engineering and science. The following is by no means a complete list of examples.

The elliptic and parabolic equations are used for modeling:

- Steady and unsteady heat transfer in solids
- Flows in porous media and diffusion problems

- Electrostatics of dielectric and conductive media
- Potential flow

The hyperbolic equation is used for:

- Transient and harmonic wave propagation in acoustics and electromagnetics
- Transverse motions of membranes

The eigenvalue problems are used for:

- Determining natural vibration states in membranes and structural mechanics problems

Last, but not least, the toolbox can be used for educational purposes as a complement to understanding the theory of the FEM.

How Do I Define a PDE Problem?

The simplest way to define a PDE problem is using the GUI, implemented in `pdetool`. There are three modes that correspond to different stages of defining a PDE problem:

- In draw mode, you create Ω, the geometry, using the constructive solid geometry (CSG) model paradigm. A set of solid objects (rectangle, circle, ellipse, and polygon) is provided. You can combine these objects using *set formulas*.
- In boundary mode, you specify the boundary conditions. You can have different types of boundary conditions on different boundary segments.
- In PDE mode, you interactively specify the type of PDE and the coefficients c, a, f, and d. You can specify the coefficients for each subdomain independently. This may ease the specification of, e.g., various material properties in a PDE model.

How Can I Solve a PDE Problem?

Most problems can be solved from the GUI. There are two major modes that help you solve a problem:

- In mesh mode, you generate and plot meshes. You can control the parameters of the automated mesh generator.
- In solve mode, you can invoke and control the nonlinear and adaptive solvers for elliptic problems. For parabolic and hyperbolic problems, you can specify the initial values, and the times for which the output should be generated. For the eigenvalue solver, you can specify the interval in which to search for eigenvalues.

After solving a problem, you can return to the mesh mode to further refine your mesh and then solve again. You can also employ the adaptive mesh refiner and solver. This option tries to find a mesh that fits the solution.

Can I Use Partial Differential Equation Toolbox Software for Nonstandard Problems?

For advanced, nonstandard applications you can transfer the description of domains, boundary conditions etc. to your MATLAB workspace. From there you use Partial Differential Equation Toolbox functions for managing data on unstructured meshes. You have full access to the mesh generators, FEM discretizations of the PDE and boundary conditions, interpolation functions, etc. You can design your own solvers or use FEM to solve subproblems of more complex algorithms. See also “Using Command-Line Functions” on page 1-42.

How Can I Visualize My Results?

From the graphical user interface you can use plot mode, where you have a wide range of visualization possibilities. You can visualize both inside the `pdetool` GUI and in separate figures. You can plot three different solution properties at the same time, using color, height, and vector field plots. Surface, mesh, contour, and arrow (quiver) plots are available. For surface plots, you can choose between interpolated and flat rendering schemes. The mesh may be hidden or exposed in all plot types. For parabolic and hyperbolic equations, you can even produce an animated movie of the solution’s time dependence. All visualization functions are also accessible from the command line.

Are There Any Applications Already Implemented?

Partial Differential Equation Toolbox software is easy to use in the most common areas due to the application interfaces. Eight application interfaces

are available, in addition to the generic scalar and system (vector valued u) cases:

- "Structural Mechanics — Plane Stress" on page 1-86
- "Structural Mechanics — Plane Strain" on page 1-92
- "Electrostatics" on page 1-93
- "Magnetostatics" on page 1-96
- "AC Power Electromagnetics" on page 1-103
- "Conductive Media DC" on page 1-108
- "Heat Transfer" on page 1-114
- "Diffusion" on page 1-117

These interfaces have dialog boxes where the PDE coefficients, boundary conditions, and solution are explained in terms of physical entities. The application interfaces enable you to enter specific parameters, such as Young's modulus in the structural mechanics problems. Also, visualization of the relevant physical variables is provided.

Several nontrivial examples are included in this manual. Many examples are solved both by using the GUI and in command-line mode.

The toolbox contains a number of demonstration files. They illustrate some ways in which you can write your own applications.

Can I Extend the Functionality of the Software?

Partial Differential Equation Toolbox software is written using the MATLAB open system philosophy. There are no black-box functions, although some functions may not be easy to understand at first glance. The data structures and formats are documented. You can examine the existing functions and create your own as needed.

How Can I Solve 3-D Problems by 2-D Models?

Partial Differential Equation Toolbox software solves problems in two space dimensions and time, whereas reality has three space dimensions. The

reduction to 2-D is possible when variations in the third space dimension (taken to be z) can be accounted for in the 2-D equation. In some cases, like the plane stress analysis, the material parameters must be modified in the process of dimensionality reduction.

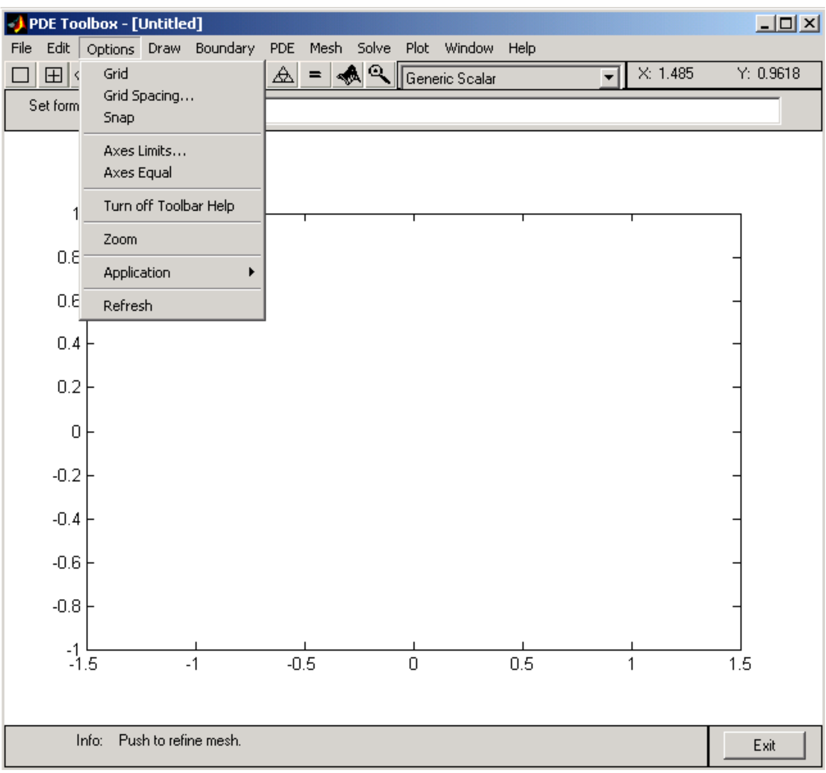
When the problem is such that variation with z is negligible, all z-derivatives drop out and the 2-D equation has exactly the same units and coefficients as in 3-D.

Slab geometries are treated by integration through the thickness. The result is a 2-D equation for the z-averaged solution with the thickness, say $D(x,y)$, multiplied onto all the PDE coefficients, c, a, d, and f, etc. For instance, if you want to compute the stresses in a sheet welded together from plates of different thickness, multiply Young's modulus E, volume forces, and specified surface tractions by $D(x,y)$. Similar definitions of the equation coefficients are called for in other slab geometry examples and application modes.

Solving a PDE

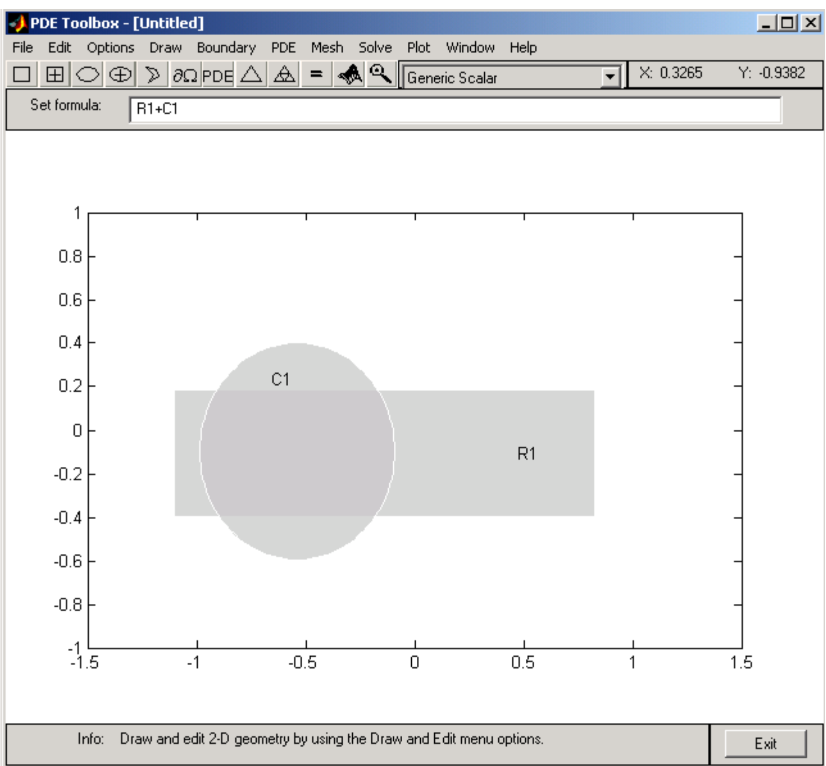
To get you started, let us use the graphical user interface (GUI) `pdetool`, which is a part of Partial Differential Equation Toolbox software, to solve a PDE step by step. The problem that we would like to solve is *Poisson's equation*, $-\Delta u = f$. The 2-D geometry on which we would like to solve the PDE is quite complex. The boundary conditions are of *Dirichlet* and *Neumann* types.

To start the GUI, type the command `pdetool` at the MATLAB prompt. It can take a minute or two for the GUI to start. The GUI looks similar to the following figure, with exception of the grid. Turn on the grid by selecting **Grid** from the **Options** menu. Also, enable the "snap-to-grid" feature by selecting **Snap** from the **Options** menu. The "snap-to-grid" feature simplifies aligning the solid objects.

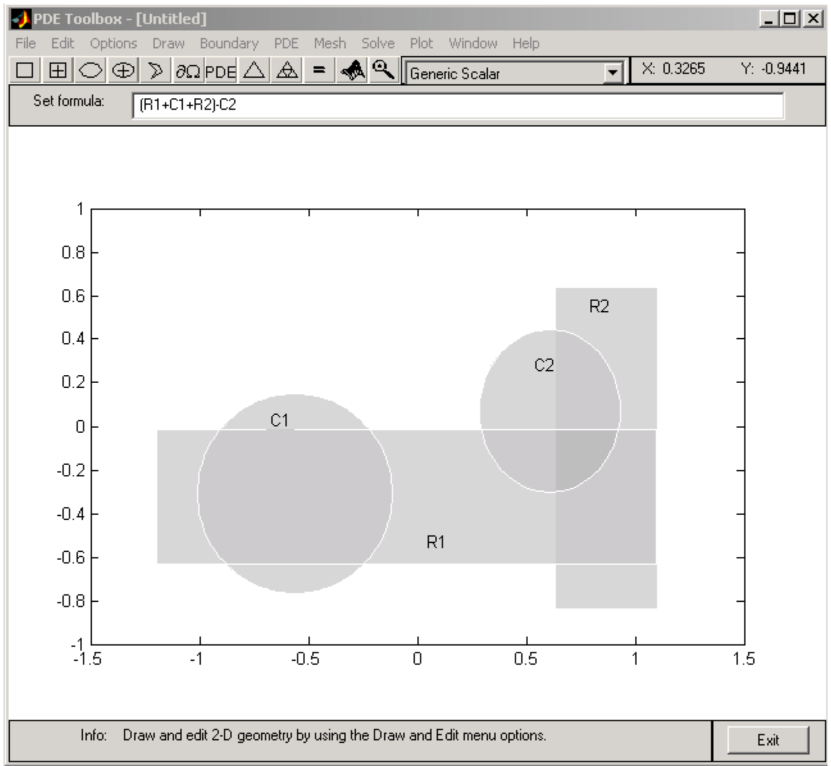


The first step is to draw the geometry on which you want to solve the PDE. The GUI provides four basic types of *solid objects*: polygons, rectangles, circles, and ellipses. The objects are used to create a *Constructive Solid Geometry model* (CSG model). Each solid object is assigned a unique label, and by the use of set algebra, the resulting geometry can be made up of a combination of unions, intersections, and set differences. By default, the resulting CSG model is the union of all solid objects.

To select a solid object, either click the button with an icon depicting the solid object that you want to use, or select the object by using the **Draw** pull-down menu. In this case, rectangle/square objects are selected. To draw a rectangle or a square starting at a corner, click the rectangle button without a + sign in the middle. The button with the + sign is used when you want to draw starting at the center. Then, put the cursor at the desired corner, and click-and-drag using the *left* mouse button to create a rectangle with the desired side lengths. (Use the right mouse button to create a square.) Notice how the “snap-to-grid” feature forces the rectangle to line up with the grid. When you release the mouse, the CSG model is updated and redrawn. At this stage, all you have is a rectangle. It is assigned the label R1. If you want to move or resize the rectangle, you can easily do so. Click-and-drag an object to move it, and double-click an object to open a dialog box, where you can enter exact location coordinates. From the dialog box, you can also alter the label. If you are not satisfied and want to restart, you can delete the rectangle by clicking the **Delete** key or by selecting **Clear** from the **Edit** menu. Next, draw a circle by clicking the button with the ellipse icon with the + sign, and then click-and-drag in a similar way, using the *right* mouse button, starting at the circle center.



The resulting CSG model is the union of the rectangle R1 and the circle C1, described by set algebra as R1+C1. The area where the two objects overlap is clearly visible as it is drawn using a darker shade of gray. The object that you just drew—the circle—has a black border, indicating that it is selected. A selected object can be moved, resized, copied, and deleted. You can select more than one object by **Shift**+clicking the objects that you want to select. Also, a **Select All** option is available from the **Edit** menu.



Finally, add two more objects, a rectangle R2 and a circle C2. The desired CSG model is formed by subtracting the circle C2 from the union of the other three objects. You do this by editing the set formula that by default is the union of all objects: C1+R1+R2+C2. You can type any other valid set formula into **Set formula** edit field. Click in the edit field and use the keyboard to change the set formula to

```
(R1+C1+R2)-C2
```

If you want, you can save this CSG model as a file. Use the **Save As** option from the **File** menu, and enter a filename of your choice. It is good practice to continue to save your model at regular intervals using **Save**. All the additional steps in the process of modeling and solving your PDE are then saved to the same file. This concludes the drawing part.

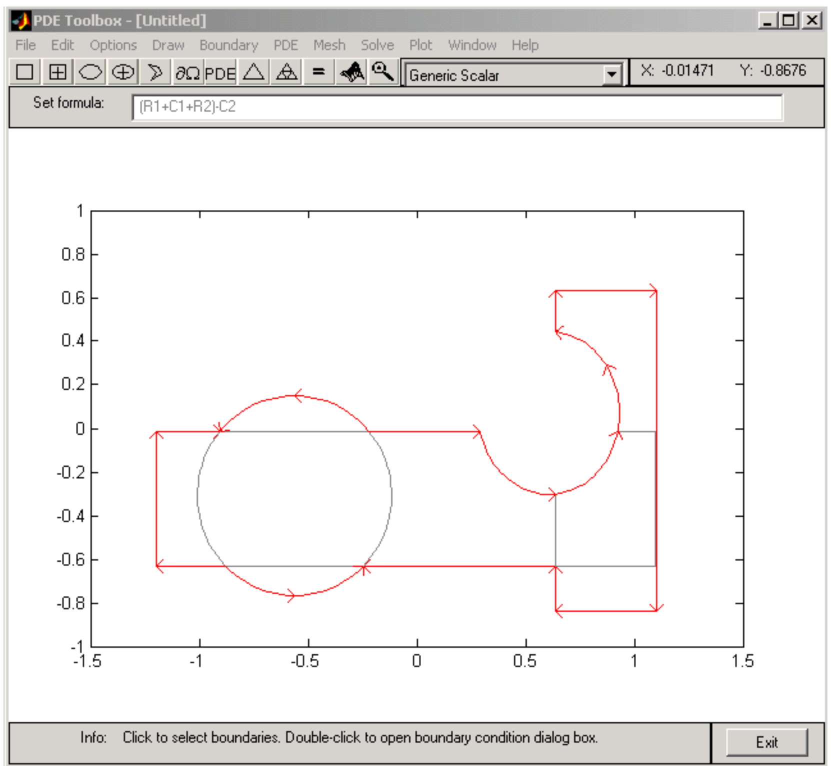
You can now define the boundary conditions for the outer boundaries. Enter the boundary mode by clicking the ∂Ω icon or by selecting **Boundary Mode** from the **Boundary** menu. You can now remove subdomain borders and define the boundary conditions.

The gray edge segments are subdomain borders induced by the intersections of the original solid objects. Borders that do not represent borders between, e.g., areas with differing material properties, can be removed. From the **Boundary** menu, select the **Remove All Subdomain Borders** option. All borders are then removed from the decomposed geometry.

The boundaries are indicated by colored lines with arrows. The color reflects the type of boundary condition, and the arrow points toward the end of the boundary segment. The direction information is provided for the case when the boundary condition is parameterized along the boundary. The boundary condition can also be a function of x and y, or simply a constant. By default, the boundary condition is of Dirichlet type: $u = 0$ on the boundary.

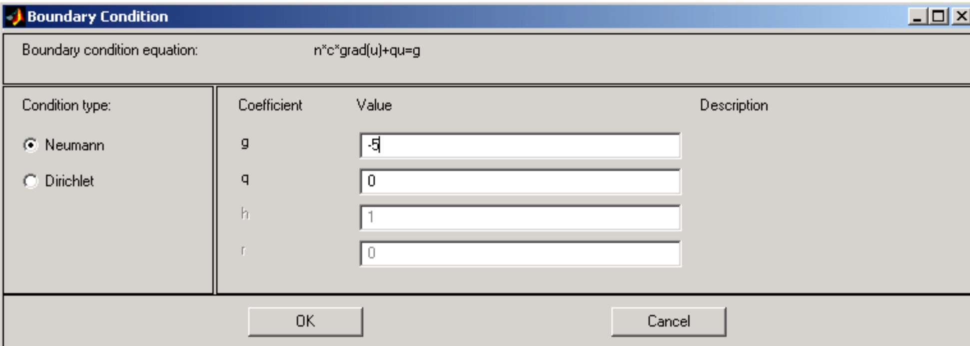
Dirichlet boundary conditions are indicated by red color. The boundary conditions can also be of a generalized Neumann (blue) or mixed (green) type. For scalar u, however, all boundary conditions are either of Dirichlet or the generalized Neumann type. You select the boundary conditions that you want to change by clicking to select one boundary segment, by **Shift**+clicking to select multiple segments, or by using the **Edit** menu option **Select All** to select all boundary segments. The selected boundary segments are indicated by black color.

For this problem, change the boundary condition for all the circle arcs. Select them by using the mouse and **Shift**+click those boundary segments.



Double-clicking anywhere on the selected boundary segments opens the Boundary Condition dialog box. Here, you select the type of boundary condition, and enter the boundary condition as a MATLAB expression. Change the boundary condition along the selected boundaries to a Neumann condition, $\partial n/\partial u = -5$. This means that the solution has a slope of -5 in the normal direction for these boundary segments.

In the Boundary Condition dialog box, select the **Neumann** condition type, and enter `-5` in the edit box for the boundary condition parameter `g`. To define a pure Neumann condition, leave the `q` parameter at its default value, `0`. When you click the **OK** button, notice how the selected boundary segments change to blue to indicate Neumann boundary condition.

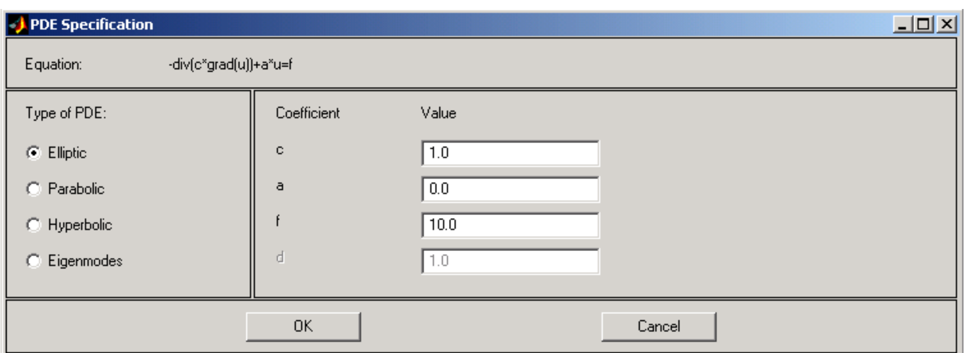


Next, specify the PDE itself through a dialog box that is accessed by clicking the button with the **PDE** icon or by selecting **PDE Specification** from the **PDE** menu. In PDE mode, you can also access the PDE Specification dialog box by double-clicking a subdomain. That way, different subdomains can have different PDE coefficient values. This problem, however, consists of only one subdomain.

In the dialog box, you can select the type of PDE (elliptic, parabolic, hyperbolic, or eigenmodes) and define the applicable coefficients depending on the PDE type. This problem consists of an elliptic PDE defined by the equation

$$-\nabla \cdot (c\nabla u) + au = f$$

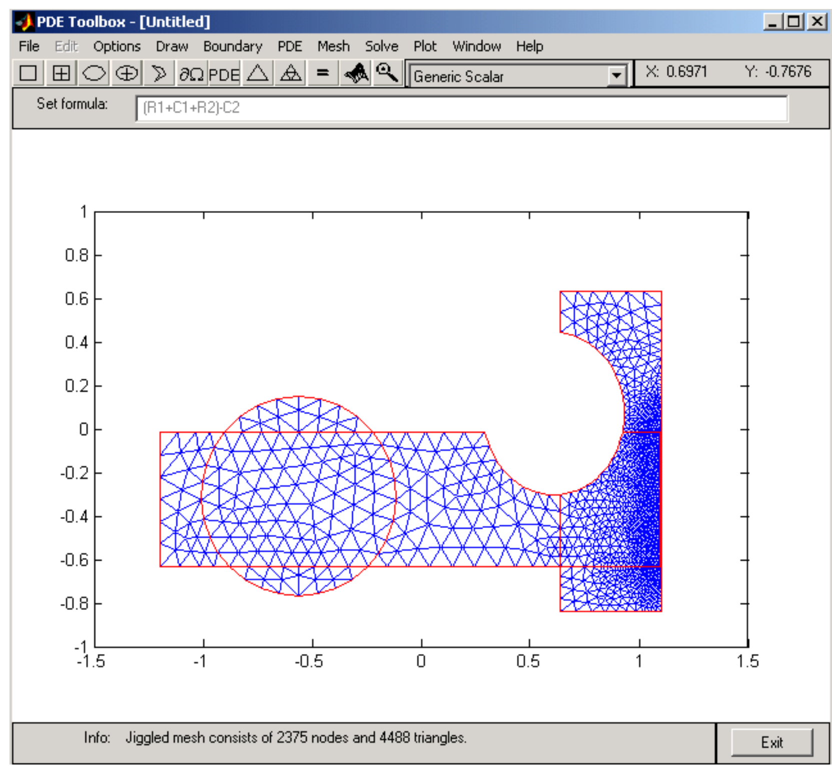
with $c = 1.0$, $a = 0.0$, and $f = 10.0$



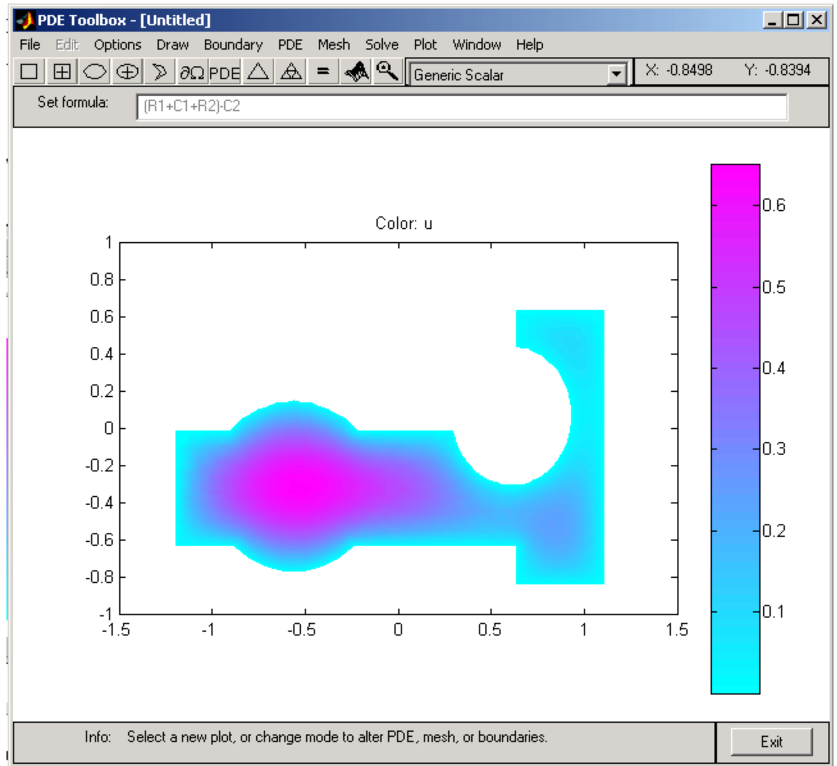
Finally, create the triangular mesh that Partial Differential Equation Toolbox software uses in the Finite Element Method (FEM) to solve the PDE. The triangular mesh is created and displayed when clicking the button with the △ icon or by selecting the **Mesh** menu option **Initialize Mesh**. If you want a more accurate solution, the mesh can be successively refined by clicking the button with the four triangle icon (the **Refine** button) or by selecting the **Refine Mesh** option from the **Mesh** menu.

Using the **Jiggle Mesh** option, the mesh can be jiggled to improve the triangle quality. Parameters for controlling the jiggling of the mesh, the refinement method, and other mesh generation parameters can be found in a dialog box that is opened by selecting **Parameters** from the **Mesh** menu. You can undo any change to the mesh by selecting the **Mesh** menu option **Undo Mesh Change**.

Initialize the mesh, then refine it once and finally jiggle it once.

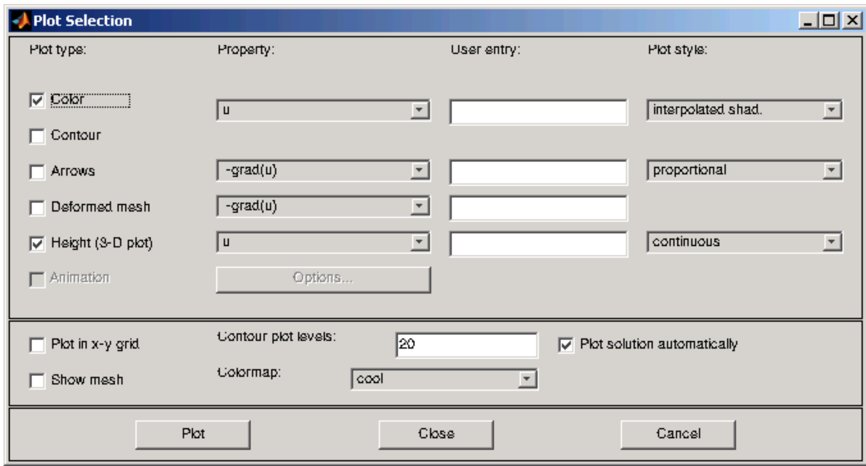


We are now ready to solve the problem. Click the **=** button or select **Solve PDE** from the **Solve** menu to solve the PDE. The solution is then plotted. By default, the plot uses interpolated coloring and a linear color map. A color bar is also provided to map the different shades to the numerical values of the solution. If you want, the solution can be exported as a vector to the MATLAB main workspace.

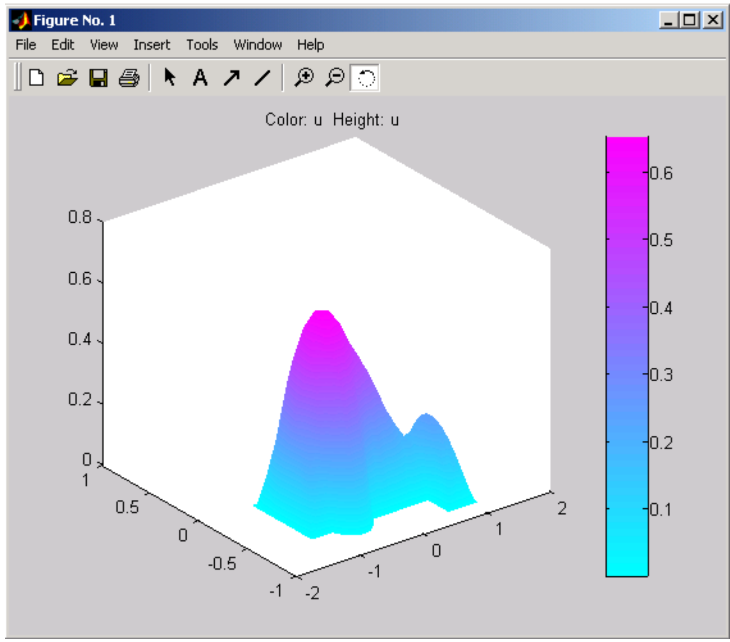


There are many more plot modes available to help you visualize the solution. Click the button with the 3-D solution icon or select **Parameters** from the **Plot** menu to access the dialog box for selection of the different plot options. Several plot styles are available, and the solution can be plotted in the GUI or in a separate figure as a 3-D plot.

Now, select a plot where the color and the height both represent u. Choose interpolated shading and use the continuous (interpolated) height option. The default colormap is the `cool` colormap; a pop-up menu lets you select from a number of different colormaps. Finally, click the **Plot** button to plot the solution; click the **Done** button to save the plot setup as the current default. The solution is plotted as a 3-D plot in a separate figure window.



The following solution plot is the result. You can use the mouse to rotate the plot in 3-D. By clicking-and-dragging the axes, the angle from which the solution is viewed can be changed.



This concludes the first example of solving a PDE by using the `pdetool` GUI. Many more examples in "Common PDE Problems" on page 1-49 focus on solving particular problems involving different kinds of PDEs, geometries and boundary conditions and covering a range of different applications.

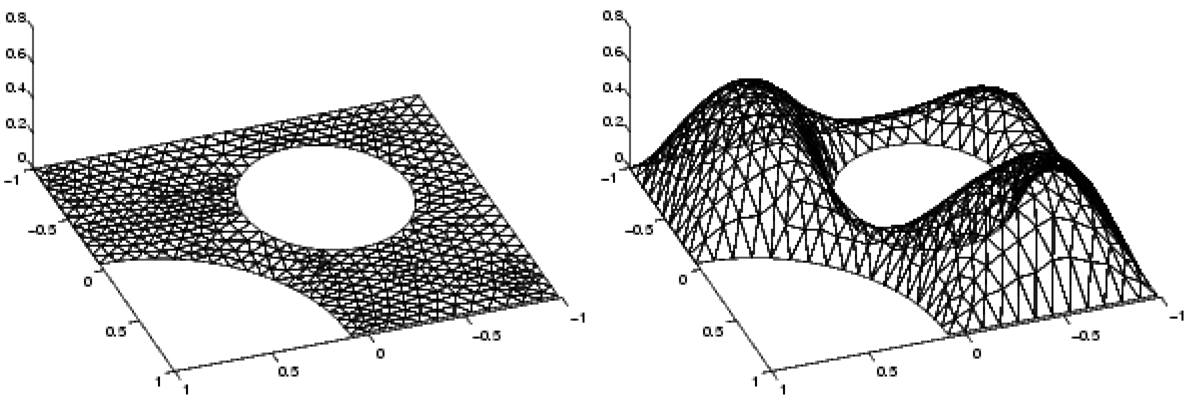
Basics of the Finite Element Method

The solutions of simple PDEs on complicated geometries can rarely be expressed in terms of elementary functions. You are confronted with two problems: First you need to describe a complicated geometry and generate a mesh on it. Then you need to discretize your PDE on the mesh and build an equation for the discrete approximation of the solution. The `pdetool` graphical user interface provides you with easy-to-use graphical tools to describe complicated domains and generate triangular meshes. It also discretizes PDEs, finds discrete solutions and plots results. You can access the mesh structures and the discretization functions directly from the command line (or from a file) and incorporate them into specialized applications.

Here is an overview of the Finite Element Method (FEM). The purpose of this presentation is to get you acquainted with the elementary FEM notions. Here you find the precise equations that are solved and the nature of the discrete solution. Different extensions of the basic equation implemented in Partial Differential Equation Toolbox software are presented. A more detailed description can be found in Chapter 3, “Finite Element Method”.

You start by approximating the computational domain Ω with a union of simple geometric objects, in this case triangles. The triangles form a mesh and each vertex is called a node. You are in the situation of an architect designing a dome. He has to strike a balance between the ideal rounded forms of the original sketch and the limitations of his simple building-blocks, triangles or quadrilaterals. If the result does not look close enough to a perfect dome, the architect can always improve his work using smaller blocks.

Next you say that your solution should be simple on each triangle. Polynomials are a good choice: they are easy to evaluate and have good approximation properties on small domains. You can ask that the solutions in neighboring triangles connect to each other continuously across the edges. You can still decide how complicated the polynomials can be. Just like an architect, you want them as simple as possible. Constants are the simplest choice but you cannot match values on neighboring triangles. Linear functions come next. This is like using flat tiles to build a waterproof dome, which is perfectly possible.



A Triangular Mesh (left) and a Continuous Piecewise Linear Function on That Mesh

Now you use the basic elliptic equation (expressed in Ω)

$$-\nabla \cdot (c\nabla u) + au = f$$

If u_h is the piecewise linear approximation to u, it is not clear what the second derivative term means. Inside each triangle, ∇u_h is a constant (because u_h is flat) and thus the second-order term vanishes. At the edges of the triangles, $c\nabla u_h$ is in general discontinuous and a further derivative makes no sense.

What you are looking for is the best approximation of u in the class of continuous piecewise polynomials. Therefore you test the equation for u_h against all possible functions v of that class. Testing means formally to multiply the residual against any function and then integrate, i.e., determine u_h such that

$$\int_{\Omega}(-\nabla \cdot (c\nabla u_h) + au_h - f)v\,dx = 0$$

for all possible v. The functions v are usually called *test functions.*

Partial integration (Green's formula) yields that u_h should satisfy

$$\int_{\Omega}\left((c\nabla u_h)\cdot\nabla v + a u_h v\right)dx - \int_{\partial\Omega}\vec{n}\cdot(c\nabla u_h)v\,ds = \int_{\Omega} f v\,dx \quad \forall v,$$

where $\partial\Omega$ is the boundary of Ω and $\vec{n}$ is the outward pointing normal on $\partial\Omega$. The integrals of this formulation are well-defined even if u_h and v are piecewise linear functions.

Boundary conditions are included in the following way. If u_h is known at some boundary points (Dirichlet boundary conditions), we restrict the test functions to $v = 0$ at those points, and require u_h to attain the desired value at that point. At all the other points we ask for Neumann boundary conditions, i.e., $(c\nabla u_h)\cdot\vec{n} + q u_h = g$. The FEM formulation reads: Find u_h such that

$$\int_{\Omega}\left((c\nabla u_h)\cdot\nabla v + a u_h v\right)dx + \int_{\partial\Omega_1} q u_h v\,ds = \int_{\Omega} f v\,dx + \int_{\partial\Omega_1} g v\,ds \quad \forall v,$$

where $\partial\Omega_1$ is the part of the boundary with Neumann conditions. The test functions v must be zero on $\partial\Omega - \partial\Omega_1$.

Any continuous piecewise linear u_h is represented as a combination

$$u_h(x) = \sum_{i=1}^{N} U_i \phi_i(x)$$

where Φ_i are some special piecewise linear basis functions and U_i are scalar coefficients. Choose Φ_i like a tent, such that it has the “height” 1 at the node i and the height 0 at all other nodes. For any fixed v, the FEM formulation yields an algebraic equation in the unknowns U_i. You want to determine N unknowns, so you need N different instances of v. What better candidates than $v = \Phi_j$, $j = 1, 2, \ldots, N$? You find a linear system $KU = F$ where the matrix K and the right side F contain integrals in terms of the test functions Φ_i, Φ_j, and the coefficients defining the problem: c, a, f, q, and g. The solution vector U contains the expansion coefficients of u_h, which are also the values of u_h at each node x_i since $u_h(x_i) = U_i$.

If the exact solution u is smooth, then FEM computes u_h with an error of the same size as that of the linear interpolation. It is possible to estimate the error on each triangle using only u_h and the PDE coefficients (but not the exact solution u, which in general is unknown).

There are Partial Differential Equation Toolbox functions that assemble K and F. This is done automatically in the graphical user interface, but you also have direct access to the FEM matrices from the command-line function `assempde`.

To summarize, the FEM approach is to approximate the PDE solution u by a piecewise linear function $u_h \cdot u_h$ is expanded in a basis of test-functions Φ_i, and the residual is tested against all the basis functions. This procedure yields a linear system $KU = F$. The components of U are the values of u_h at the nodes. For x inside a triangle, $u_h(x)$ is found by linear interpolation from the nodal values.

FEM techniques are also used to solve more general problems. The following are some generalizations that you can access both through the graphical user interface and with command-line functions.

- Time-dependent problems are easy to implement in the FEM context. The solution $u(x,t)$ of the equation

  $$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f$$

  can be approximated by

  $$u_h(x,t) = \sum_{i=1}^{N} U_i(t)\phi_i(x)$$

- This yields a system of ordinary differential equations (ODE)

  $$M\frac{d}{dt}U + KU = F$$

  which you integrate using ODE solvers. Two time derivatives yield a second order ODE

  $$M\frac{d^2}{dt^2}U + KU = F$$

  etc. The toolbox supports problems with one or two time derivatives (the functions `parabolic` and `hyperbolic`).

- Eigenvalue problems: Solve

  $$-\nabla \cdot (c\nabla u) + au = \lambda du$$

for the unknowns u and λ (λ is a complex number). Using the FEM discretization, you solve the algebraic eigenvalue problem $KU = \lambda_h MU$ to find u_h and λ_h as approximations to u and λ. A robust eigenvalue solver is implemented in `pdeeig`.

- If the coefficients *c, a, f, q,* or *g* are functions of u, the PDE is called nonlinear and FEM yields a nonlinear system $K(U)\ U= F(U)$. You can use iterative methods for solving the nonlinear system. The toolbox provides a nonlinear solver called `pdenonlin` using a damped Gauss-Newton method.
- Small triangles are needed only in those parts of the computational domain where the error is large. In many cases the errors are large in a small region and making all triangles small is a waste of computational effort. Making small triangles only where needed is called adapting the mesh refinement to the solution. An iterative adaptive strategy is the following: For a given mesh, form and solve the linear system $KU = F$. Then estimate the error and refine the triangles in which the error is large. The iteration is controlled by `adaptmesh` and the error is estimated by `pdejmps`.

Although the basic equation is scalar, systems of equations are also handled by the toolbox. The interactive environment accepts u as a scalar or 2-vector function. In command-line mode, systems of arbitrary size are accepted.

If $c \geq \delta > 0$ and $a \geq 0$, under rather general assumptions on the domain Ω and the boundary conditions, the solution u exists and is unique. The FEM linear system has a unique solution which converges to u as the triangles become smaller. The matrix K and the right side F make sense even when u does not exist or is not unique. It is advisable that you devise checks to problems with questionable solutions.

Using the pdetool GUI

In this section...

"Introduction" on page 1-27

"The Menus" on page 1-29

"The Toolbar" on page 1-30

"The GUI Modes" on page 1-31

"The CSG Model and the Set Formula" on page 1-32

"Creating Rounded Corners" on page 1-33

"Suggested Modeling Method" on page 1-35

"Object Selection Methods" on page 1-39

"Display Additional Information" on page 1-40

"Entering Parameter Values as MATLAB Expressions" on page 1-40

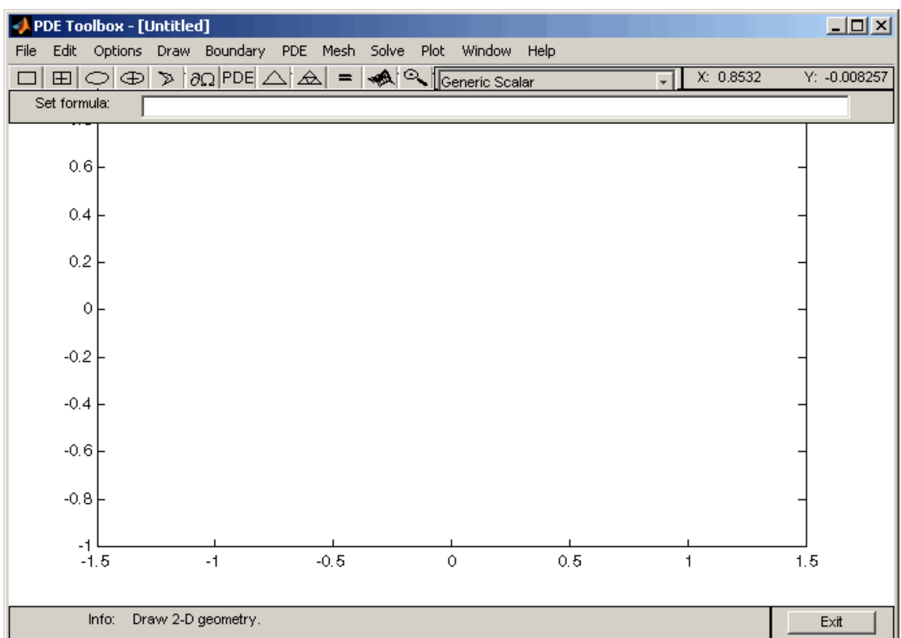
"Using Earlier Version Partial Differential Equation Toolbox Model Files" on page 1-41

Introduction

Partial Differential Equation Toolbox software includes a complete graphical user interface (GUI), which covers all aspects of the PDE solution process. You start it by typing

```
pdetool
```

at the MATLAB command line. It may take a while the first time you launch `pdetool` during a MATLAB session. The following figure shows the `pdetool` GUI as it looks when you start it.



At the top, the GUI has a pull-down menu bar that you use to control the modeling. It conforms to common pull-down menu standards. Menu items followed by a right arrow lead to a submenu. Menu items followed by an ellipsis lead to a dialog box. Stand-alone menu items lead to direct action. Below the menu bar, a toolbar with icon buttons provide quick and easy access to some of the most important functions.

To the right of the toolbar is a pop-up menu that indicates the current application mode. You can also use it to change the application mode. The upper right part of the GUI also provides the x- and y-coordinates of the current cursor position. It is updated when you move the cursor inside the main axes area in the middle of the GUI. The edit box for the set formula contains the active set formula. In the main axes you draw the 2-D geometry, display the mesh, plot the solution, etc. At the bottom of the GUI, an information line provides information about the current activity. It can also display help information about the toolbar buttons.

The Menus

There are 11 different pull-down menus in the GUI. See Chapter 2, “Graphical User Interface” for a more detailed description of the menus and the dialog boxes:

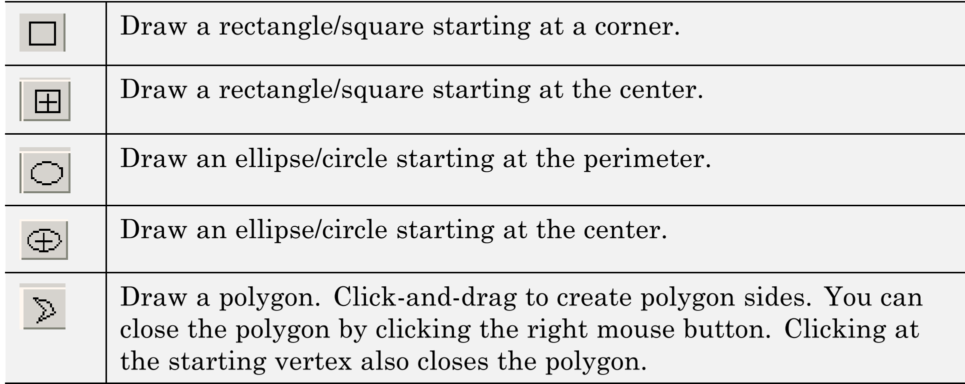
- **File** menu. From the **File** menu you can **Open** and **Save** model files that contain a command sequence that reproduces your modeling session. You can also print the current graphics and exit the GUI.
- **Edit** menu. From the **Edit** menu you can cut, clear, copy, and paste the solid objects. There is also a **Select All** option.
- **Options** menu. The **Options** menu contains options such as toggling the axis grid, a “snap-to-grid” feature, and zoom. You can also adjust the axis limits and the grid spacing, select the application mode, and refresh the GUI.
- **Draw** menu. From the **Draw** menu you can select the basic solid objects such as circles and polygons. You can then draw objects of the selected type using the mouse. From the **Draw** menu you can also rotate the solid objects and export the geometry to the MATLAB main workspace.
- **Boundary** menu. From the **Boundary** menu you access a dialog box where you define the boundary conditions. Additionally, you can label edges and subdomains, remove borders between subdomains, and export the decomposed geometry and the boundary conditions to the workspace.
- **PDE** menu. The **PDE** menu provides a dialog box for specifying the PDE, and there are menu options for labeling subdomains and exporting PDE coefficients to the workspace.
- **Mesh** menu. From the **Mesh** menu you create and modify the triangular mesh. You can initialize, refine, and jiggle the mesh, undo previous mesh changes, label nodes and triangles, display the mesh quality, and export the mesh to the workspace.
- **Solve** menu. From the **Solve** menu you solve the PDE. You can also open a dialog box where you can adjust the solve parameters, and you can export the solution to the workspace.
- **Plot** menu. From the **Plot** menu you can plot a solution property. A dialog box lets you select which property to plot, which plot style to use and several other plot parameters. If you have recorded a movie (animation) of the solution, you can export it to the workspace.

- **Window** menu. The **Window** menu lets you select any currently open MATLAB figure window. The selected window is brought to the front.
- **Help** menu. The **Help** menu provides a brief help window.

The Toolbar

The toolbar underneath the main menu at the top of the GUI contains icon buttons that provide quick and easy access to some of the most important functions.

The five leftmost buttons are draw mode buttons and they represent, from left to right:



	Draw a rectangle/square starting at a corner.
	Draw a rectangle/square starting at the center.
	Draw an ellipse/circle starting at the perimeter.
	Draw an ellipse/circle starting at the center.
	Draw a polygon. Click-and-drag to create polygon sides. You can close the polygon by clicking the right mouse button. Clicking at the starting vertex also closes the polygon.

The draw mode buttons can only be activated one at the time and they all work the same way: single-clicking a button allows you to draw one solid object of the selected type. Double-clicking a button makes it “stick,” and you can then continue to draw solid objects of the selected type until you single-click the button to “release” it. Using the right mouse button or **Ctrl**+click, the drawing is constrained to a square or a circle.

The second group of six buttons includes the following analysis buttons.

∂Ω	Enters the boundary mode.
PDE	Opens the PDE Specification dialog box.
	Initializes the triangular mesh.
	Refines the triangular mesh.
=	Solves the PDE.
	3-D solution opens the Plot Selection dialog box.

The button toggles the zoom function on/off.

The GUI Modes

The PDE solving process can be divided into several steps:

1. Define the geometry (2-D domain).
2. Define the boundary conditions.
3. Define the PDE.
4. Create the triangular mesh.
5. Solve the PDE.
6. Plot the solution and other physical properties calculated from the solution (post processing).

The `pdetool` GUI is designed in a similar way. You work in six different modes, each corresponding to one of the steps in the PDE solving process:

- In draw mode, you can create the 2-D geometry using the constructive solid geometry (CSG) model paradigm. A set of solid objects (rectangle, circle, ellipse, and polygon) is provided. These objects can be combined using set formulas in a flexible way.

- In boundary mode, you can specify the boundary conditions. You can have different types of boundary conditions on different boundaries. In this mode, the original shapes of the solid objects constitute borders between subdomains of the model. Such borders can be eliminated in this mode.
- In PDE mode, you can interactively specify the type of PDE problem, and the PDE coefficients. You can specify the coefficients for each subdomain independently. This makes it easy to specify, e.g., various material properties in a PDE model.
- In mesh mode, you can control the automated mesh generation and plot the mesh.
- In solve mode, you can invoke and control the nonlinear and adaptive solver for elliptic problems. For parabolic and hyperbolic PDE problems, you can specify the initial values, and the times for which the output should be generated. For the eigenvalue solver, you can specify the interval in which to search for eigenvalues.
- In plot mode, there is a wide range of visualization possibilities. You can visualize both in the `pdetool` GUI and in a separate figure window. You can visualize three different solution properties at the same time, using color, height, and vector field plots. There are surface, mesh, contour, and arrow (quiver) plots available. For parabolic and hyperbolic equations, you can animate the solution as it changes with time.

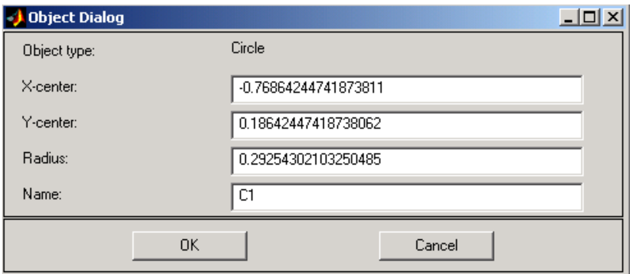
The CSG Model and the Set Formula

Partial Differential Equation Toolbox functions use the Constructive Solid Geometry (CSG) model paradigm for modeling. You can draw *solid objects* that can overlap. There are four types of solid objects:

- **Circle** object — Represents the set of points inside and on a circle.
- **Polygon** object — Represents the set of points inside and on a polygon given by a set of line segments.
- **Rectangle** object — Represents the set of points inside and on a rectangle.
- **Ellipse** object — Represents the set of points inside and on an ellipse. The ellipse can be rotated.

Each solid object is automatically given a unique name by the GUI. The default names are C1, C2, C3, etc., for circles; P1, P2, P3, etc. for polygons; R1,

R2, R3, etc., for rectangles; E1, E2, E3, etc., for ellipses. Squares, although a special case of rectangles, are named SQ1, SQ2, SQ3, etc. The name is displayed on the solid object itself. You can use any unique name, as long as it contains no blanks. In draw mode, you can alter the names and the geometries of the objects by double-clicking them, which opens a dialog box. The following figure shows an object dialog box for a circle.



You can use the name of the object to refer to the corresponding set of points in a set formula. The operators +, *, and - are used to form the set of points Ω in the plane over which the differential equation is solved. The operators +, the set union operator, and *, the set intersection operator, have the same precedence. The operator -, the set difference operator, has higher precedence. The precedence can be controlled by using parentheses. The resulting geometrical model, Ω, is the set of points for which the set formula evaluates to true. By default, it is the union of all solid objects. We often refer to the area Ω as the *decomposed geometry*.

Creating Rounded Corners

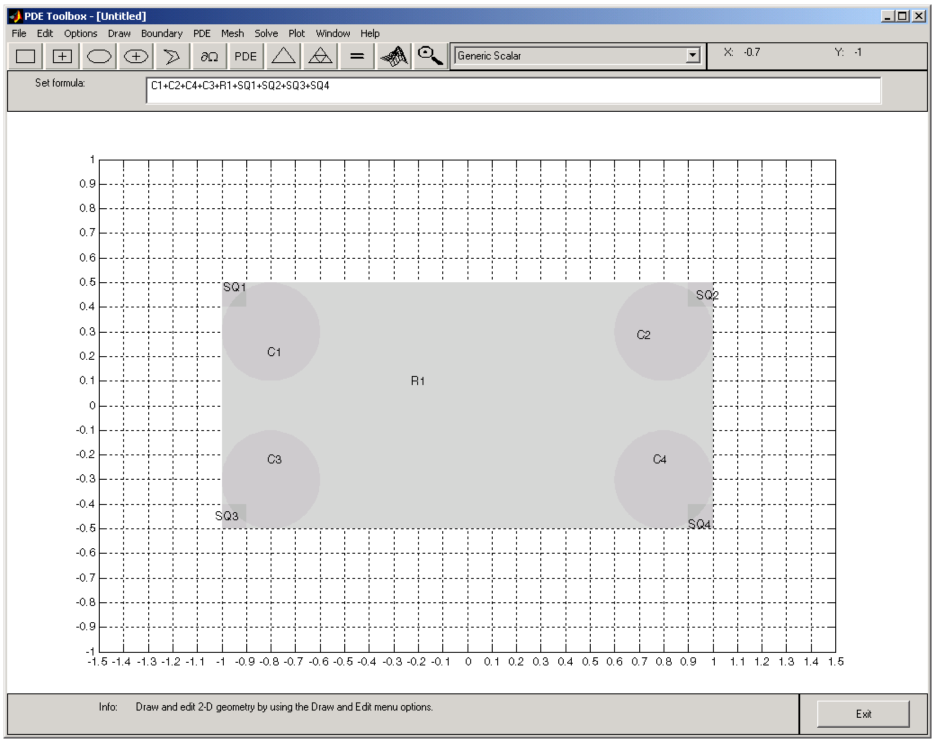
As an example of how to use the set formula, let us model a plate with rounded corners (fillets).

Start the GUI and turn on the grid and the “snap-to-grid” feature using the **Options** menu. Also, change the grid spacing to `-1.5:0.1:1.5` for the *x*-axis and `-1:0.1:1` for the *y*-axis.

Select **Rectangle/square** from the **Draw** menu or click the button with the rectangle icon. Then draw a rectangle with a width of 2 and a height of 1 using the mouse, starting at (-1,0.5). To get the round corners, add circles, one in each corner. The circles should have a radius of 0.2 and centers at

a distance that is 0.2 units from the left/right and lower/upper rectangle boundaries ((-0.8,-0.3), (-0.8,0.3), (0.8,-0.3), and (0.8,0.3)). To draw several circles, double-click the button for drawing ellipses/circles (centered). Then draw the circles using the right mouse button or **Ctrl**+click starting at the circle centers. Finally, at each of the rectangle corners, draw four small squares with a side of 0.1.

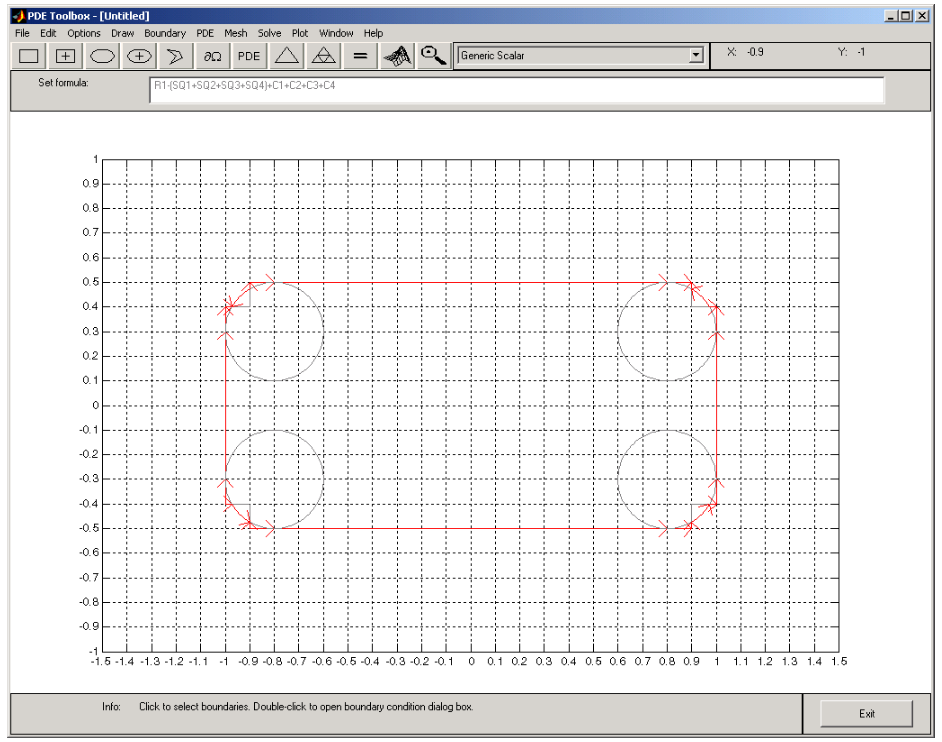
The following figure shows the complete drawing.



Now you have to edit the set formula. To get the rounded corners, subtract the small squares from the rectangle and then add the circles. As a set formula, this is expressed as

```
R1-(SQ1+SQ2+SQ3+SQ4)+C1+C2+C3+C4
```

Enter the set formula into the edit box at the top of the GUI. Then enter the **Boundary** mode by clicking the ∂Ω button or by selecting the **Boundary Mode** option from the **Boundary** menu. The CSG model is now decomposed using the set formula, and you get a rectangle with rounded corners, as shown in the following figure.



Because of the intersection of the solid objects used in the initial CSG model, a number of subdomain borders remain. They are drawn using gray lines. If this is a model of, e.g., a homogeneous plate, you can remove them. Select the **Remove All Subdomain Borders** option from the **Boundary** menu. The subdomain borders are removed and the model of the plate is now complete.

Suggested Modeling Method

Although Partial Differential Equation Toolbox software offers you a great deal of flexibility in the ways that you can approach the problems and interact

with the toolbox functions, there is a suggested method of choice for modeling and solving your PDE problems using the `pdetool` GUI. There are also a number of shortcuts that you can use in certain situations.

Note There are platform-dependent keyboard accelerators available for many of the most common `pdetool` GUI activities. Learning to use the accelerator keys may improve the efficiency of your `pdetool` sessions.

The basic flow of actions is indicated by the way the graphical buttons and the menus are ordered from left to right. You work your way from left to right in the process of modeling, defining, and solving your PDE problem using the `pdetool` GUI:

- When you start, `pdetool` is in draw mode, where you can use the four basic solid objects to draw your Constructive Solid Geometry (CSG) model. You can also edit the set formula. The solid objects are selected using the five leftmost buttons (or from the **Draw** menu).
- To the right of the draw mode buttons you find buttons through which you can access all the functions that you need to define and solve the PDE problem: define boundary conditions, design the triangular mesh, solve the PDE, and plot the solution.

The following sequence of actions covers all the steps of a normal `pdetool` session:

1. Use `pdetool` as a drawing tool to make a drawing of the 2-D geometry on which you want to solve your PDE. Make use of the four basic solid objects and the grid and the "snap-to-grid" feature. The GUI starts in the draw mode, and you can select the type of object that you want to use by clicking the corresponding button or by using the **Draw** menu. Combine the solid objects and the set algebra to build the desired CSG model.

2. Save the geometry to a model file. If you want to continue working using the same geometry at your next Partial Differential Equation Toolbox session, simply type the name of the model file at the MATLAB prompt. The `pdetool` GUI then starts with the model file's solid geometry loaded. If you save the PDE problem at a later stage of the solution process, the

model file also contains commands to recreate the boundary conditions, the PDE coefficients, and the mesh.

3 Move to the next step in the PDE solving process by clicking the $\partial\Omega$ button. The outer boundaries of the decomposed geometry are displayed with the default boundary condition indicated. If the outer boundaries do not match the geometry of your problem, reenter the draw mode. You can then correct your CSG model by adding, removing or altering any of the solid objects, or change the set formula used to evaluate the CSG model.

Note The set formula can only be edited while you are in the draw mode.

If the drawing process resulted in any unwanted subdomain borders, remove them by using the **Remove Subdomain Border** or **Remove All Subdomain Borders** option from the **Boundary** menu.

You can now define your problem's boundary conditions by selecting the boundary to change and open a dialog box by double-clicking the boundary or by using the **Specify Boundary Conditions** option from the **Boundary** menu.

4 Initialize the triangular mesh. Click the Δ button or use the corresponding **Mesh** menu option **Initialize Mesh**. Normally, the mesh algorithm's default parameters generate a good mesh. If necessary, they can be accessed using the **Parameters** menu item.

5 If you need a finer mesh, the mesh can be refined by clicking the **Refine** button. Clicking the button several times causes a successive refinement of the mesh. The cost of a very fine mesh is a significant increase in the number of points where the PDE is solved and, consequently, a significant increase in the time required to compute the solution. Do not refine unless it is required to achieve the desired accuracy. For each refinement, the number of triangles increases by a factor of four. A better way to increase the accuracy of the solution to elliptic PDE problems is to use the adaptive solver, which refines the mesh in the areas where the estimated error of the solution is largest. See the `adaptmesh` reference page for an example of how the adaptive solver can solve a Laplace equation with an accuracy that requires more than 10 times as many triangles when regular refinement is used.

6 Specify the PDE from the PDE Specification dialog box. You can access that dialog box using the **PDE** button or the **PDE Specification** menu item from the **PDE** menu.

Note This step can be performed at any time prior to solving the PDE since it is independent of the CSG model and the boundaries. If the PDE coefficients are material dependent, they are entered in the PDE mode by double-clicking the different subdomains.

---

7 Solve the PDE by clicking the = button or by selecting **Solve PDE** from the **Solve** menu. If you do not want an automatic plot of the solution, or if you want to change the way the solution is presented, you can do that from the Plot Selection dialog box prior to solving the PDE. You open the Plot Selection dialog box by clicking the button with the 3-D solution plot icon or by selecting the **Parameters** menu item from the **Plot** menu.

8 Now, from here you can choose one of several alternatives:

- Export the solution and/or the mesh to the MATLAB main workspace for further analysis.
- Visualize other properties of the solution.
- Change the PDE and recompute the solution.
- Change the mesh and recompute the solution. If you select **Initialize Mesh**, the mesh is initialized; if you select **Refine Mesh**, the current mesh is refined. From the **Mesh** menu, you can also jiggle the mesh and undo previous mesh changes.
- Change the boundary conditions. To return to the mode where you can select boundaries, use the $\partial\Omega$ button or the **Boundary Mode** option from the **Boundary** menu.
- Change the CSG model. You can reenter the draw mode by selecting **Draw Mode** from the **Draw** menu or by clicking one of the **Draw Mode** icons to add another solid object. Back in the draw mode, you are able to add, change, or delete solid objects and also to alter the set formula.

In addition to the recommended path of actions, there are a number of shortcuts, which allow you to skip over one or more steps. In general, the `pdetool` GUI adds the necessary steps automatically.

- If you have not yet defined a CSG model, and leave the draw mode with an empty model, `pdetool` creates an L-shaped geometry with the default boundary condition and then proceeds to the action called for, performing all the steps necessary.
- If you are in draw mode and click the Δ button to initialize the mesh, `pdetool` first decomposes the geometry using the current set formula and assigns the default boundary condition to the outer boundaries. After that, an initial mesh is created.
- If you click the **refine** button to refine the mesh before the mesh has been initialized, `pdetool` first initializes the mesh (and decomposes the geometry, if you were still in the draw mode).
- If you click the = button to solve the PDE and you have not yet created a mesh, `pdetool` initializes a mesh before solving the PDE.
- If you select a plot type and choose to plot the solution, `pdetool` checks to see if there is a solution to the current PDE available. If not, `pdetool` first solves the current PDE. The solution is then displayed using the selected plot options.
- If you have not defined your PDE, `pdetool` solves the default PDE, which is Poisson's equation:

  $$-\Delta u = 10$$

  (This corresponds to the generic elliptic PDE with $c = 1$, $a = 0$, and $f = 10$.) For the different application modes, different default PDE settings apply.

Object Selection Methods

Throughout the GUI, similar principles apply for selecting objects such as solid objects, subdomains, and boundaries.

- To select a single object, click it using the left mouse button.
- To select several objects and to deselect objects, **Shift**+click (or click using the middle mouse button) on the desired objects.

- Clicking in the intersection of several objects selects all the intersecting objects.
- To open an associated dialog box, double-click an object. If the object is not selected, it is selected before opening the dialog box.
- In draw mode and PDE mode, clicking outside of objects deselects all objects.
- To select all objects, use the **Select All** option from the **Edit** menu.
- When defining boundary conditions and the PDE via the menu items from the **Boundary** and **PDE** menus, and no boundaries or subdomains are selected, the entered values applies to all boundaries and subdomains by default.

Display Additional Information

In mesh mode, you can use the mouse to display the node number and the triangle number at the position where you click. Press the left mouse button to display the node number on the information line. Press the middle mouse button (or use the left mouse button and the **Shift** key) to display the triangle number on the information line.

In plot mode, you can use the mouse to display the numerical value of the plotted property at the position where you click. Press the left mouse button to display the triangle number and the value of the plotted property on the information line.

The information remains on the information line until you release the mouse button.

Entering Parameter Values as MATLAB Expressions

When entering parameter values, e.g., as a function of x and y, the entered string must be a MATLAB expression to be evaluated for x and y defined on the current mesh, i.e., x and y are MATLAB row vectors. For example, the function $4\,x\,y$ should be entered as `4*x.*y` and not as `4*x*y`, which normally is not a valid MATLAB expression.

Using Earlier Version Partial Differential Equation Toolbox Model Files

You can convert *Model files* created using an earlier version of Partial Differential Equation Toolbox software for use with the current versions of MATLAB and Partial Differential Equation Toolbox software. The old Model files cannot be used directly in the current version of Partial Differential Equation Toolbox software.

To convert your old Model files, use the conversion utility `pdemdlcv`. For example, to convert a Model file called `model42.m` to a compatible Model file called `model5.m`, type the following at the MATLAB command line:

```
pdemdlcv model42 model5
```

Using Command-Line Functions

In this section...

"Introduction" on page 1-42
"Data Structures and Utility Functions" on page 1-42
"Hints and Suggestions for Using Command-Line Functions" on page 1-47

Introduction

Although the `pdetool` GUI provides a convenient working environment, there are situations where the flexibility of using the command-line functions is needed. These include:

- Geometrical shapes other than straight lines, circular arcs, and elliptical arcs
- Nonstandard boundary conditions
- Complicated PDE or boundary condition coefficients
- More than two dependent variables in the system case
- Nonlocal solution constraints
- Special solution data processing and presentation itemize

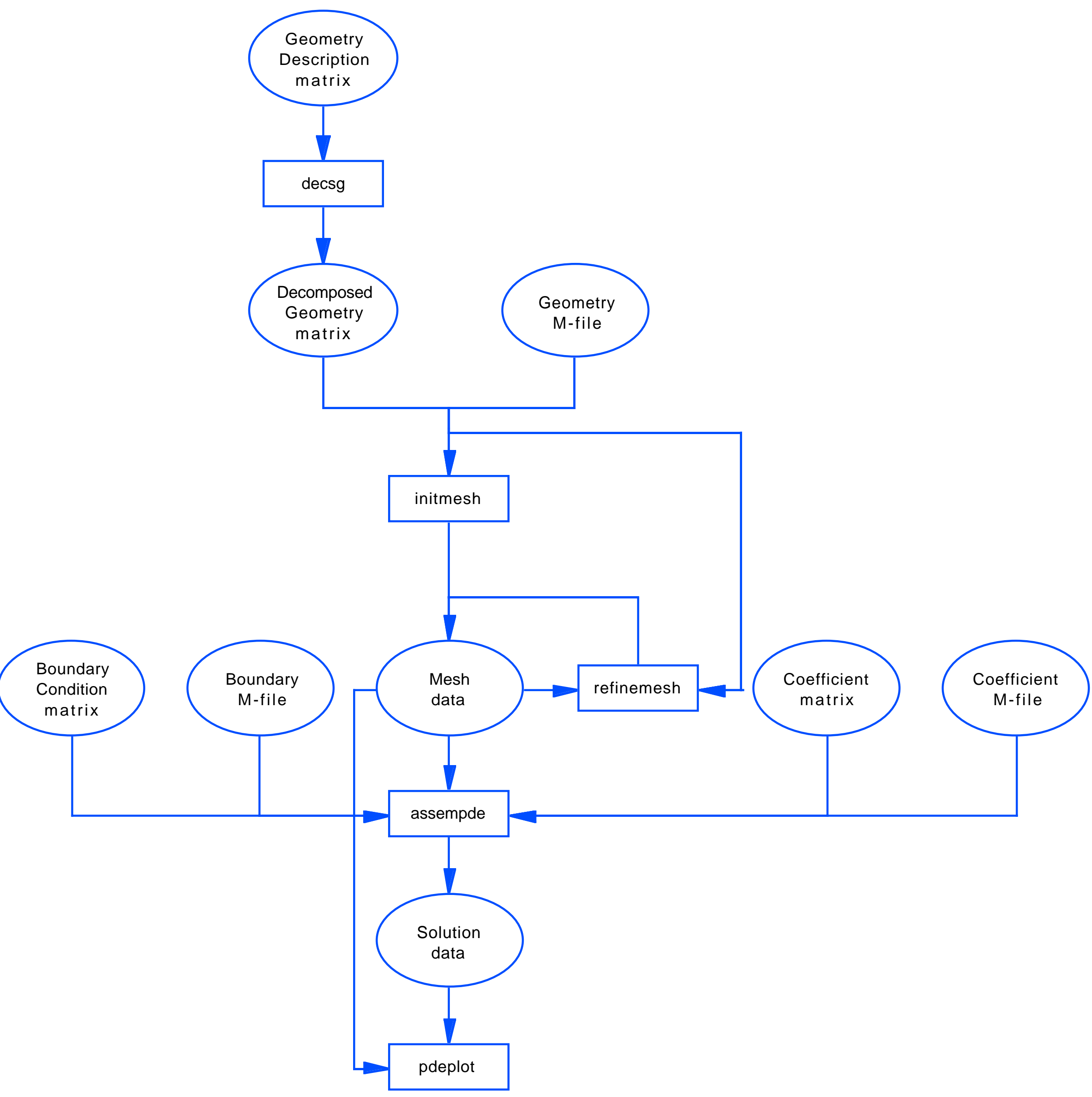
The GUI can still be a valuable aid in some of the situations presented previously, if part of the modeling is done using the GUI and then made available for command-line use through the extensive data export facilities of the GUI.

Data Structures and Utility Functions

The process of defining your problem and solving it is reflected in the design of the GUI. A number of data structures define different aspects of the problem, and the various processing stages produce new data structures out of old ones. See the following figure.

The rectangles are functions, and ellipses are data represented by matrices or files. Arrows indicate data necessary for the functions.

As there is a definite direction in this diagram, you can cut into it by presenting the needed data sets, and then continue downward. In the following sections, we give pointers to descriptions of the precise formats of the various data structures and files.


Geometry
Description
matrix
decsg
Decomposed
Geometry
matrix
Geometry
M-file
initmesh
Boundary
Condition
matrix
Boundary
M-file
Mesh
data
refinemesh
Coefficient
matrix
Coefficient
M-file
assempde
Solution
data
pdeplot

Constructive Solid Geometry Model

A Constructive Solid Geometry (CSG) model is specified by a *Geometry Description matrix*, a *set formula,* and a *Name Space matrix.* For a description of these data structures, see the reference page for `decsg`. At this level, the problem geometry is defined by overlapping solid objects. These can be created by drawing the CSG model in the GUI and then exporting the data using the **Export Geometry Description, Set Formula, Labels** option from the **Draw** menu.

Decomposed Geometry

A *decomposed geometry* is specified by either a *Decomposed Geometry matrix*, or by a *Geometry file*. Here, the geometry is described as a set of *disjoint minimal regions* bounded by *boundary segments* and *border segments.* A Decomposed Geometry matrix can be created from a CSG model by using the function `decsg`. It can also be exported from the GUI by selecting the **Export Decomposed Geometry, Boundary Cond's** option from the **Boundary** menu. A Geometry file equivalent to a given Decomposed Geometry matrix can be created using the `wgeom` function. A decomposed geometry can be visualized with the `pdegplot` function. For descriptions of the data structures of the Decomposed Geometry matrix and Geometry file, see the respective reference pages for `decsg` and `pdegeom`.

Boundary Conditions

These are specified by either a *Boundary Condition matrix*, or a *Boundary file.* Boundary conditions are given as functions on boundary segments. A Boundary Condition matrix can be exported from the GUI by selecting the **Export Decomposed Geometry, Boundary Cond's** option from the **Boundary** menu. A Boundary file equivalent to a given Boundary Condition matrix can be created using the `wbound` function. For a description of the data structures of the Boundary Condition matrix and Boundary file, see the respective reference pages for `assemb` and `pdebound`.

Equation Coefficients

The PDE is specified by either a *Coefficient matrix* or a *Coefficient file* for each of the PDE coefficients *c, a, f,* and *d*. The coefficients are functions on the subdomains. Coefficients can be exported from the GUI by selecting the

Export PDE Coefficient option from the **PDE** menu. For the details on the equation coefficient data structures, see the reference page for `assempde`.

Mesh

A triangular mesh is described by the *mesh data* which consists of a *Point matrix,* an *Edge matrix,* and a *Triangle matrix.* In the mesh, minimal regions are triangulated into subdomains, and border segments and boundary segments are broken up into edges. Mesh data is created from a decomposed geometry by the function `initmesh` and can be altered by the functions `refinemesh` and `jigglemesh`. The **Export Mesh** option from the **Mesh** menu provides another way of creating mesh data. The `adaptmesh` function creates mesh data as part of the solution process. The mesh may be plotted with the `pdemesh` function. For details on the mesh data representation, see the reference page for `initmesh`.

Solution

The solution of a PDE problem is represented by the *solution vector.* A solution gives the value at each mesh point of each dependent variable, perhaps at several points in time, or connected with different eigenvalues. Solution vectors are produced from the mesh, the boundary conditions, and the equation coefficients by `assempde`, `pdenonlin`, `adaptmesh`, `parabolic`, `hyperbolic`, and `pdeeig`. The **Export Solution** option from the **Solve** menu exports solutions to the workspace. Since the meaning of a solution vector is dependent on its corresponding mesh data, they are always used together when a solution is presented. For details on solution vectors, see the reference page for `assempde`.

Post Processing and Presentation

Given a solution/mesh pair, a variety of tools is provided for the visualization and processing of the data. `pdeintrp` and `pdeprtni` can be used to interpolate between functions defined at triangle nodes and functions defined at triangle midpoints. `tri2grid` interpolates a functions from a triangular mesh to a rectangular grid. `pdegrad` and `pdecgrad` compute gradients of the solution. `pdeplot` has a large number of options for plotting the solution. `pdecont` and `pdesurf` are convenient shorthands for `pdeplot`.

Hints and Suggestions for Using Command-Line Functions

Several examples of command-line function usage are given in "Common PDE Problems" on page 1-49.

Use the export facilities of the GUI as much as you can. They provide data structures with the correct syntax, and these are good starting points that you can modify to suit your needs.

A good way to produce a Geometry file describing a geometry outside of the possibilities provided by the GUI is to draw a similar geometry using the GUI, export the Decomposed Geometry matrix, and write a Geometry file with `wgeom`. The special segments can then be edited by hand. An example of a hand-tailored Geometry file is `cardg`. See also the reference page for `pdegeom`.

Working with the system matrices and vectors produced by `assema` and `assemb` can sometimes be valuable. When solving the same equation for different loads or boundary conditions, it pays to assemble the stiffness matrix only once. Point loads on a particular node can be implemented by adding the load to the corresponding row in the right side vector. A nonlocal constraint can be incorporated into the `H` and `R` matrices.

An example of a handwritten Coefficient file is `circlef.m`, which produces a point load. You can find the full example in `pdedemo7` and on the `assempde` reference page.

The routines for adaptive mesh generation and solution are powerful but can lead to dense meshes and thus long computation times. Setting the `Ngen` parameter to one limits you to a single refinement step. This step can then be repeated to show the progress of the refinement. The `Maxt` parameter helps you stop before the adaptive solver generates too many triangles. An example of a handwritten triangle selection function is `circlepick`, used in `pdedemo7`. Remember that you always need a decomposed geometry with `adaptmesh`.

Deformed meshes are easily plotted by adding offsets to the Point matrix `p`. Assuming two variables stored in the solution vector `u`:

```
np=size(p,2);
pdemesh(p+scale*[u(1:np) u(np+1:np+np)]',e,t)
```

The time evolution of eigenmodes is obtained by, e.g.,

```
u1=u(:,mode)*cos(sqrt(l(mode))*tlist)            % hyperbolic
```

for positive eigenvalues in hyperbolic problems, or

```
u1=u(:,mode)*exp(-l(mode)*tlist);            % parabolic
```

in parabolic problems. This makes nice animations, perhaps together with deformed mesh plots.

Common PDE Problems

In this section...

"Elliptic Problems" on page 1-49
"Parabolic Problems" on page 1-64
"Hyperbolic Problem" on page 1-71
"Eigenvalue Problems" on page 1-75
"Application Modes" on page 1-84
"References" on page 1-118

Elliptic Problems

This topic describes the solution of some elliptic PDE problems. The last problem, a minimal surface problem, is nonlinear and illustrates the use of the nonlinear solver. The problems are solved using both the Partial Differential Equation Toolbox graphical user interface and command-line functions. The topics include:

- "Poisson's Equation on Unit Disk" on page 1-49
- "A Scattering Problem" on page 1-53
- "A Minimal Surface Problem" on page 1-58
- "Domain Decomposition" on page 1-60

Poisson's Equation on Unit Disk

As a first example of an elliptic problem, let us use the simplest elliptic PDE of all—*Poisson's equation.*

The problem formulation is

$$-\Delta U = 1 \text{ in } \Omega,\ U = 0 \text{ on } \partial\Omega$$

where Ω is the unit disk. In this case, the exact solution is

$$U(x, y) = \frac{1 - x^2 - y^2}{4}$$

so the error of the numeric solution can be evaluated for different meshes.

Using the Graphical User Interface. With the `pdetool` graphical user interface (GUI) started, perform the following steps using the generic scalar mode:

1. Using some of the **Option** menu features, add a grid and turn on the "snap-to-grid" feature. Draw a circle by clicking the button with the ellipse icon with the + sign, and then click-and-drag from the origin, using the *right* mouse button, to a point at the circle's perimeter. If the circle that you create is not a perfect unit circle, double-click the circle. This opens a dialog box where you can specify the exact center location and radius of the circle.

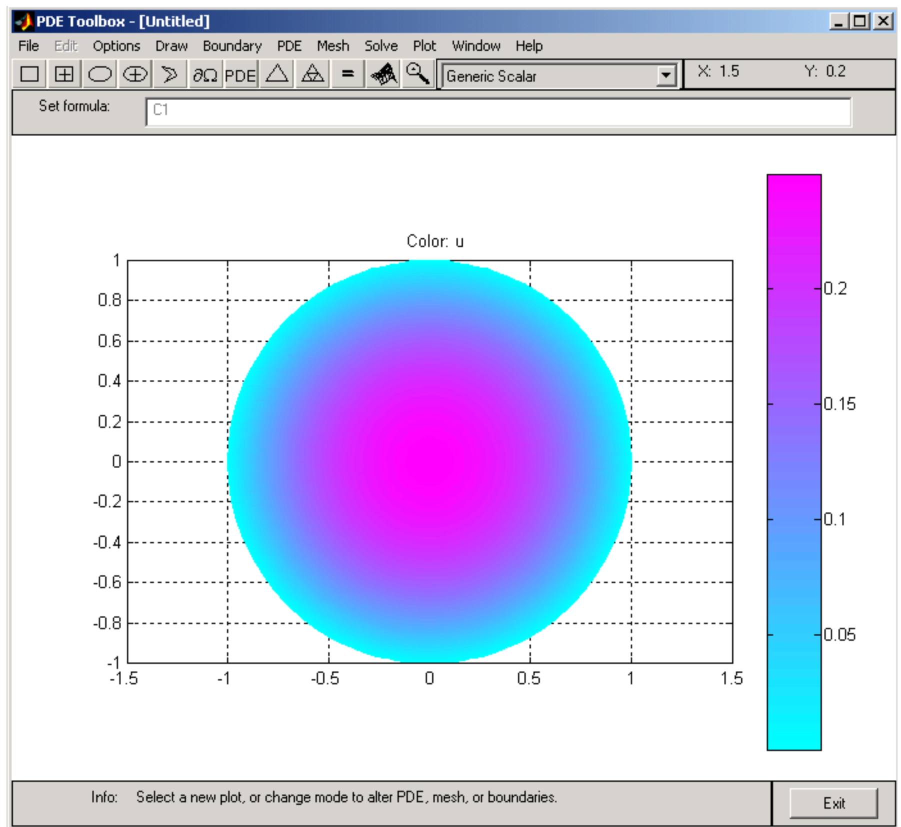
2. Enter the boundary mode by clicking the button with the $\partial\Omega$ icon. The boundaries of the decomposed geometry are plotted, and the outer boundaries are assigned a default boundary condition (Dirichlet boundary condition, $u = 0$ on the boundary). In this case, this is what we want. If the boundary condition is different, double-click the boundary to open a dialog box through which you can enter and display the boundary condition.

3. To define the partial differential equation, click the **PDE** button. This opens a dialog box, where you can define the PDE coefficients `c`, `a`, and `f`. In this simple case, they are all constants: $c = 1$, $f = 1$, and $a = 0$.

4. Click the Δ button or select **Initialize Mesh** from the **Mesh** menu. This initializes and displays a triangular mesh.

5. Click the **Refine** button or select **Refine Mesh** from the **Mesh** menu. This causes a refinement of the initial mesh, and the new mesh is displayed.

6. To solve the system, just click the **=** button. The toolbox assembles the PDE problem and solves the linear system. It also provides a plot of the solution. Using the Plot Selection dialog box, you can select different types of solution plots.



7 To compare the numerical solution to the exact solution, select the `user entry` in the **Property** pop-up menu for **Color** in the Plot Selection dialog box. Then input the MATLAB expression `u-(1-x.^2-y.^2)/4` in the **user entry** edit field. You obtain a plot of the absolute error in the solution.

You can also compare the numerical solution to the exact solution by entering some simple command-line-oriented commands. It is easy to export the mesh data and the solution to the MATLAB main workspace by using the **Export** options from the **Mesh** and **Solve** menus. To refine the mesh and solve the PDE successively, simply click the **refine** and **=** buttons until the desired accuracy is achieved. (Another possibility is to use the adaptive solver.)

Using Command-Line Functions. First you must create a MATLAB function that parameterizes the 2-D geometry—in this case a unit circle.

The `circleg.m` file returns the coordinates of points on the unit circle's boundary. The file conforms to the file format described on the reference page for `pdegeom`. You can display the file by typing `type circleg`.

Also, you need a function that describes the boundary condition. This is a Dirichlet boundary condition where $u = 0$ on the boundary. The `circleb1.m` file provides the boundary condition. The file conforms to the file format described on the reference page for `pdebound`. You can display the file by typing `type circleb1`.

Now you can start working from the MATLAB command line:

```
[p,e,t]=initmesh('circleg','Hmax',1);
error=[]; err=1;
while err > 0.001,
    [p,e,t]=refinemesh('circleg',p,e,t);
    u=assempde('circleb1',p,e,t,1,0,1);
    exact=-(p(1,:).^2+p(2,:).^2-1)/4;
    err=norm(u-exact',inf);
    error=[error err];
  end
pdemesh(p,e,t)
pdesurf(p,t,u)
pdesurf(p,t,u-exact')
```

The first MATLAB command creates the initial mesh using the parameterizing function `circleg`.

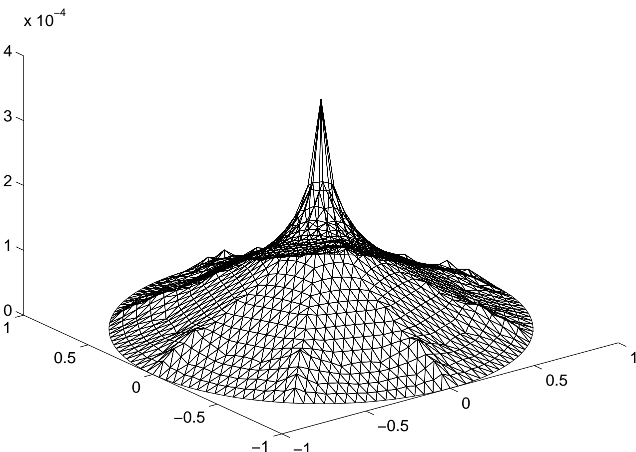
Also, initialize a vector `error` for the maximum norm errors of the successive solutions and set the initial error `err` to 1. The loop then runs until the error of the solution is smaller than 10^{-3}.

1. Refine the mesh. The current triangular mesh, defined by the geometry `circleg`, the point matrix `p`, the edge matrix `e`, and the triangle matrix `t`, is refined, and the mesh is returned using the same matrix variables.

2 Assemble and solve the linear system. The coefficients of the elliptic PDE are constants ($c = f = 1$, $a = 0$) for this simple case. `circleb1` contains a description of the boundary conditions, and `p`, `e`, and `t` define the triangular mesh.

3 Find the error of the numerical solution produced by Partial Differential Equation Toolbox solver. The vector `exact` contains the exact solution at the nodes, and what you actually find is the max-norm error of the solution at the nodes.

4 Plot the mesh, the solution, and the error. The plot function `pdesurf` as third argument can take any vector of values on the mesh given by `p` and `t`, not just the solution. In this case you are also plotting the error function.

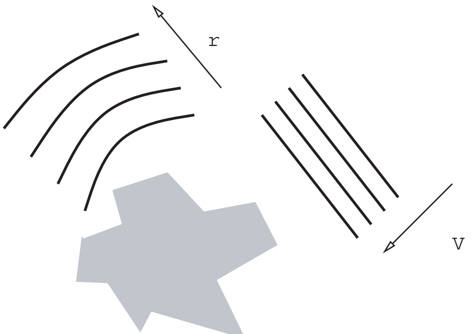


The Error Function

`pdedemo1` performs all the previous steps.

A Scattering Problem

The scattering problem is to compute the waves reflected from an object illuminated by incident waves. For this problem consider an infinite horizontal membrane subjected to small vertical displacements U. The membrane is fixed at the object boundary.



We assume that the medium is homogeneous so that the wave speed is constant, c.

Note Do not confuse this c with the parameter `c` appearing in Partial Differential Equation Toolbox functions.

When the illumination is harmonic in time, we can compute the field by solving a single steady problem. With $U(x,y,t) = u(x,y)e^{-i\omega t}$, the wave equation

$$\frac{\partial^2 U}{\partial t^2} - c^2 \Delta U = 0$$

turns into $-\omega^2 u - c^2 \Delta u = 0$ or the *Helmholtz's equation*

$$-\Delta u - k^2 u = 0,$$

where k, the *wave number,* is related to the angular frequency ω, the frequency f, and the wavelength λ by

$$k = \frac{\omega}{c} = \frac{2\pi f}{c} = \frac{2\pi}{\lambda}.$$

We have yet to specify the boundary conditions. Let the incident wave be a plane wave traveling in the direction $\vec{a} = (\cos(a), \sin(a))$:

$$V(x,y,t) = e^{i(k\vec{a}\cdot\vec{x}-\omega t)} = v(x,y)e^{-i\omega t},$$

where

$$v(x,y) = e^{ik\vec{a}\cdot\vec{x}}.$$

u is the sum of v and the reflected wave r,

$$u = v + r$$

The boundary condition for the object's boundary is easy: $u = 0$, i.e.,

$$r = -v(x,y)$$

For acoustic waves, where v is the pressure disturbance, the proper condition would be

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

The reflected wave r travels outward from the object. The condition at the outer computational boundary should be chosen to allow waves to pass without reflection. Such conditions are usually called nonreflecting, and we use the classical *Sommerfeld radiation condition.* As $|\vec{x}|$ approaches infinity, r approximately satisfies the one-way wave equation

$$\frac{\partial r}{\partial t} + c\vec{\xi} \cdot \nabla r = 0$$

which allows waves moving in the positive ξ-direction only (ξ is the radial distance from the object). With the time-harmonic solution, this turns into the generalized Neumann boundary condition

$$\vec{\xi} \cdot \nabla r = ikr$$

For simplicity, let us make the outward normal of the computational domain approximate the outward ξ-direction.

Using the Graphical User Interface. You can now use `pdetool` to solve this scattering problem. Using the generic scalar mode, start by drawing the 2-D geometry of the problem. Let the illuminated object be a square SQ1 with a side of 0.1 units and center in `[0.8 0.5]` and rotated 45 degrees, and let the computational domain be a circle C1 with a radius of 0.45 units and the same center location. The Constructive Solid Geometry (CSG) model is then given by `C1-SQ1`.

For the outer boundary (the circle perimeter), the boundary condition is a generalized Neumann condition with $q = -ik$. The wave number $k = 60$, which corresponds to a wavelength of about 0.1 units, so enter `-60i` as a constant `q` and `0` as a constant `g`.

For the square object's boundary, you have a Dirichlet boundary condition:

$$r = -v(x, y) = -e^{ik\vec{a} \cdot \vec{x}}$$

In this problem, the incident wave is traveling in the $-x$ direction, so the boundary condition is simply

$$r = -e^{-ikx}$$

Enter this boundary condition in the Boundary Condition dialog box as a Dirichlet condition: `h=1, r=-exp(-i*60*x)`. The real part of this is a sinusoid.

For sufficient accuracy, about 10 finite elements per wavelength are needed. The outer boundary should be located a few object diameters from the object itself. An initial mesh generation and two successive mesh refinements give approximately the desired resolution.

Although originally a wave equation, the transformation into a Helmholtz's equation makes it—in the Partial Differential Equation Toolbox context, but not strictly mathematically—an elliptic equation. The elliptic PDE coefficients for this problem are $c = 1$, $a = -k^2 = -3600$, and $f = 0$. Open the PDE Specification dialog box and enter these values.

The problem can now be solved, and the solution is complex. For a complex solution, the real part is plotted and a warning message is issued.

The propagation of the reflected waves is computed as

$$\mathrm{Re}(r(x,y)e^{-i\omega t}),$$

which is the reflex of

$$\mathrm{Re}\left(e^{i\left(k\vec{a}\cdot\vec{x}-\omega t\right)}\right).$$

To see the whole field, plot

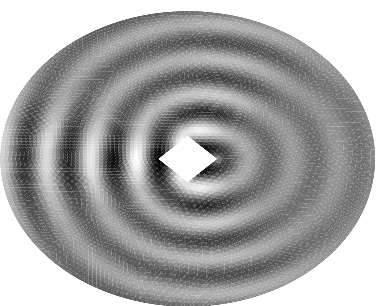
$$\mathrm{Re}\left(\left(r(x,y)+e^{ik\vec{a}\cdot\vec{x}}\right)e^{-i\omega t}\right).$$

The reflected waves and the “shadow” behind the object are clearly visible when you plot the reflected wave.

To make an animation of the reflected wave, the solution and the mesh data must first be exported to the main workspace. Then make a script file or type the following commands at the MATLAB prompt:

```
h=newplot; hf=get(h,'Parent'); set(hf,'Renderer','zbuffer')
axis tight, set(gca,'DataAspectRatio',[1 1 1]); axis off
M=moviein(10,hf);
maxu=max(abs(u));
colormap(cool)
for j=1:10,
   ur=real(exp(-j*2*pi/10*sqrt(-1))*u));
   pdeplot(p,e,t,'xydata',ur,'colorbar','off','mesh','off');
   caxis([-maxu maxu]);
   axis tight, set(gca,'DataAspectRatio',[1 1 1]); axis off
   M(:,j)=getframe;
 end
movie(hf,M,50);
```

`pdedemo2` contains a full command-line demonstration of the scattering problem.



A Minimal Surface Problem

In many problems the coefficients c, a, and f do not only depend on x and y, but also on the solution u itself. Consider the equation

$$-\nabla \cdot \left(\frac{1}{\sqrt{1+|\nabla u|^2}} \nabla u \right) = 0$$

on the unit disk $\Omega = \{(x, y) \mid x^2 + y^2 \leq 1\}$, with $u = x^2$ on $\partial\Omega$.

This problem is nonlinear and cannot be solved with the regular elliptic solver. Instead, the nonlinear solver `pdenonlin` is used.

Let us solve this *minimal surface problem* using the `pdetool` GUI and command-line functions.

Using the Graphical User Interface. Make sure that the application mode in the `pdetool` GUI is set to **Generic Scalar.** The problem domain is simply a unit circle. Draw it and move to the boundary mode to define the boundary conditions. Use **Select All** from the **Edit** menu to select all boundaries. Then double-click a boundary to open the Boundary Condition dialog box. The Dirichlet condition $u = x^2$ is entered by typing `x.^2` into the **r** edit box. Next, open the PDE Specification dialog box to define the PDE. This is an elliptic equation with

$$c = \frac{1}{\sqrt{1+|\nabla u|^2}},\ a = 0,\ \text{and } f = 0.$$

The nonlinear c is entered into the c edit box as

```
1./sqrt(1+ux.^2+uy.^2)
```

Initialize a mesh and refine it once.

Before solving the PDE, select **Parameters** from the **Solve** menu and check the **Use nonlinear solver** option. Also, set the tolerance parameter to `0.001`.

Click the = button to solve the PDE. Use the Plot Selection dialog box to plot the solution in 3-D (check `u` and `continuous` selections in the **Height** column) to visualize the saddle shape of the solution.

Using Command-Line Functions. Working from the command line, the following sequence of commands solves the minimal surface problem and plots the solution. The files `circleg` and `circleb2` contain the geometry specification and boundary condition functions, respectively.

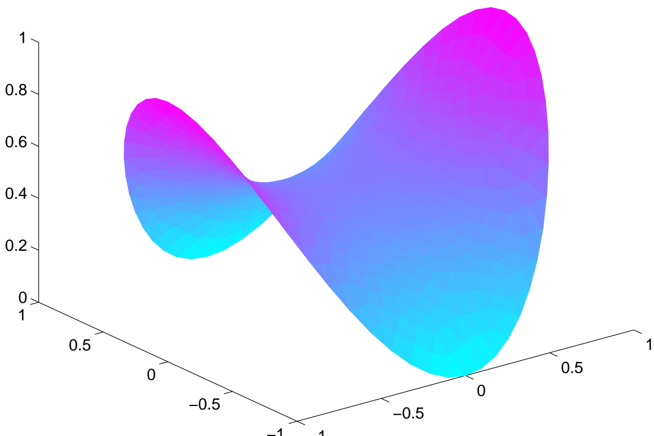
```
g='circleg';
b='circleb2';
c='1./sqrt(1+ux.^2+uy.^2)';
rtol=1e-3;

[p,e,t]=initmesh(g);
[p,e,t]=refinemesh(g,p,e,t);

u=pdenonlin(b,p,e,t,c,0,0,'Tol',rtol);

pdesurf(p,t,u)
```

You can run this example by typing `pdedemo3`.



Domain Decomposition

Partial Differential Equation Toolbox software is designed to deal with one-level domain decomposition. If Ω has a complicated geometry, it is often useful to decompose it into the union of more subdomains of simpler structure. Such structures are often introduced by `pdetool`.

Assume now that Ω is the disjoint union of some subdomains $\Omega_1, \Omega_2, \ldots,$ Ωn. Then you could renumber the nodes of a mesh on Ω such that the indices of the nodes of each subdomain are grouped together, while all the indices of nodes common to two or more subdomains come last. Since K has nonzero entries only at the lines and columns that are indices of neighboring nodes, the stiffness matrix is partitioned as follows:

$$K = \begin{pmatrix} K_1 & 0 & \ldots & 0 & B_1^T \\ 0 & K_2 & \ldots & 0 & B_2^T \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & K_n & B_n^T \\ B_1 & B_2 & \ldots & B_n & C \end{pmatrix}$$

while the right side is

$$F = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \\ f_c \end{pmatrix}$$

The Partial Differential Equation Toolbox function `assempde` can assemble the matrices *K*j*, B*j*, f*j*,* and *C* separately. You have full control over the storage and further processing of these matrices.

Furthermore, the structure of the linear system

$Ku = F$

is simplified by decomposing *K* into the partitioned matrix.

Now consider the geometry of the L-shaped membrane. You can plot the geometry of the membrane by typing

```
pdegplot('lshapeg')
```

Notice the borders between the subdomains. There are three subdomains. Thus the matrix formulas with $n = 3$ can be used. Now generate a mesh for the geometry:

```
[p,e,t]=initmesh('lshapeg');
[p,e,t]=refinemesh('lshapeg',p,e,t);
[p,e,t]=refinemesh('lshapeg',p,e,t);
```

So for this case, with $n = 3$, you have

$$\begin{pmatrix} K_1 & 0 & 0 & B_1^T \\ 0 & K_2 & 0 & B_2^T \\ 0 & 0 & K_3 & B_3^T \\ B_1 & B_2 & B_3 & C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_c \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_c \end{pmatrix}$$

and the solution is given by block elimination:

$$(C - B_1 K_1^{-1} B_1^T - B_2 K_2^{-1} B_2^T - B_3 K_3^{-1} B_3^T) u_c = f_c - B_1 K_1^{-1} f_1 - B_2 K_2^{-1} f_2 - B_3 K_3^{-1} f_3$$
$$u_1 = K_1^{-1}(f_1 - B_1^T u_c)$$
$$\ldots$$

In the following MATLAB solution, a more efficient algorithm using Cholesky factorization is used:

```
time=[];
np=size(p,2);
% Find common points
c=pdesdp(p,e,t);

nc=length(c);
C=zeros(nc,nc);
FC=zeros(nc,1);

[i1,c1]=pdesdp(p,e,t,1);ic1=pdesubix(c,c1);
[K,F]=assempde('lshapeb',p,e,t,1,0,1,time,1);
K1=K(i1,i1);d=symamd(K1);i1=i1(d);
K1=chol(K1(d,d));B1=K(c1,i1);a1=B1/K1;
C(ic1,ic1)=C(ic1,ic1)+K(c1,c1)-a1*a1';
f1=F(i1);e1=K1'\f1;FC(ic1)=FC(ic1)+F(c1)-a1*e1;

[i2,c2]=pdesdp(p,e,t,2);ic2=pdesubix(c,c2);
[K,F]=assempde('lshapeb',p,e,t,1,0,1,time,2);
K2=K(i2,i2);d=symamd(K2);i2=i2(d);
K2=chol(K2(d,d));B2=K(c2,i2);a2=B2/K2;
C(ic2,ic2)=C(ic2,ic2)+K(c2,c2)-a2*a2';
f2=F(i2);e2=K2'\f2;FC(ic2)=FC(ic2)+F(c2)-a2*e2;
```

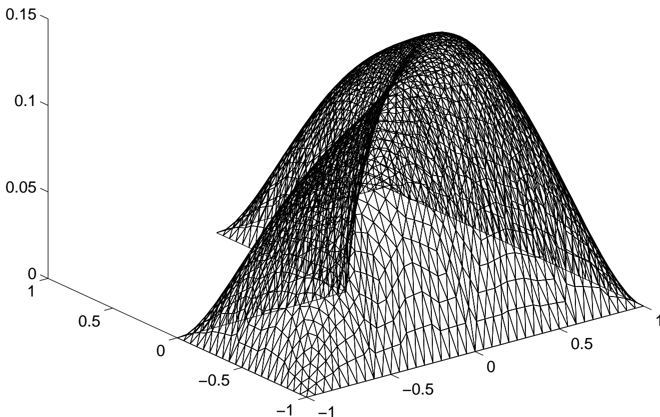
```
[i3,c3]=pdesdp(p,e,t,3);ic3=pdesubix(c,c3);
[K,F]=assempde('lshapeb',p,e,t,1,0,1,time,3);
K3=K(i3,i3);d=symamd(K3);i3=i3(d);
K3=chol(K3(d,d));B3=K(c3,i3);a3=B3/K3;
C(ic3,ic3)=C(ic3,ic3)+K(c3,c3)-a3*a3';
f3=F(i3);e3=K3'\f3;FC(ic3)=FC(ic3)+F(c3)-a3*e3;

% Solve
u=zeros(np,1);
u(c)=C\ FC;
u(i1)=K1\(e1-a1'*u(c1));
u(i2)=K2\(e2-a2'*u(c2));
u(i3)=K3\(e3-a3'*u(c3));
```

The problem can also be solved by typing

```
% Compare with solution not using subdomains
[K,F]=assempde('lshapeb',p,e,t,1,0,1);u1=K\F;
norm(u-u1,'inf')
pdesurf(p,t,u)
```

You can run this entire example by typing `pdedemo4`.



Parabolic Problems

This section describes the solution of some parabolic PDE problems. The problems are solved using both the Partial Differential Equation Toolbox graphical user interface and the command-line functions. The topics include:

- "The Heat Equation: A Heated Metal Block" on page 1-64
- "Heat Distribution in Radioactive Rod" on page 1-69

The Heat Equation: A Heated Metal Block

A common parabolic problem is the *heat equation*:

$$d\frac{\partial u}{\partial t} - \Delta u = 0$$

The heat equation describes the diffusion of heat in a body of some kind. See "Application Modes" on page 1-84 for more information about heat transfer and diffusion problems.

This first example studies a heated metal block with a rectangular crack or cavity. The left side of the block is heated to 100 degrees centigrade. At the right side of the metal block, heat is flowing from the block to the surrounding air at a constant rate. All the other block boundaries are isolated. This leads to the following set of boundary conditions (when proper scaling of t is chosen):

- $u = 100$ on the left side (Dirichlet condition)
- $\frac{\partial u}{\partial n} = -10$ on the right side (Neumann condition)
- $\frac{\partial u}{\partial n} = 0$ on all other boundaries (Neumann condition)

Also, for the heat equation we need an initial value: the temperature in the metal block at the starting time t_0. In this case, the temperature of the block is 0 degrees at the time we start applying heat.

Finally, to complete the problem formulation, we specify that the starting time is 0 and that we want to study the heat distribution during the first five seconds.

Using the Graphical User Interface. Once you have started the `pdetool` GUI and selected the **Generic Scalar** mode, drawing the CSG model can be done very quickly: Draw a rectangle (R1) with the corners in `x = [-0.5 0.5 0.5 -0.5]` and `y = [-0.8 -0.8 0.8 0.8]`. Draw another rectangle (R2) to represent the rectangular cavity. Its corners should have the coordinates `x = [-0.05 0.05 0.05 -0.05]` and `y = [-0.4 -0.4 0.4 0.4]`. To assist in drawing the narrow rectangle representing the cavity, open the Grid Spacing dialog box from the **Options** and enter *x*-axis extra ticks at `-0.05` and `0.05`. Then turn on the grid and the “snap-to-grid” feature. A rectangular cavity with the correct dimensions is then easy to draw.

The CSG model of the metal block is now simply expressed as the set formula `R1-R2`.

Leave the draw mode and enter the boundary mode by clicking the $\partial\Omega$ button, and continue by selecting boundaries and specifying the boundary conditions. Using the **Select All** option from the **Edit** menu and then defining the Neumann condition

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

for all boundaries first is a good idea since that leaves only the leftmost and rightmost boundaries to define individually.

The next step is to open the PDE Specification dialog box and enter the PDE coefficients.

The generic parabolic PDE that Partial Differential Equation Toolbox functions solve is

$$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f$$

with initial values $u0 = u(t_0)$ and the times at which to compute a solution specified in the array `tlist`.

For this case, you have $d = 1$, $c = 1$, $a = 0$, and $f = 0$.

Initialize the mesh by clicking the Δ button. If you want, you can refine the mesh by clicking the **Refine** button.

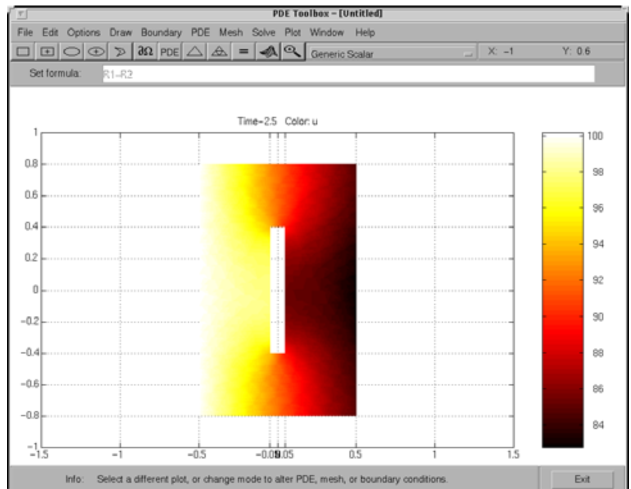
The initial values $u0 = 0$, and the list of times is entered as the MATLAB array `[0:0.5:5]`. They are entered into the Solve Parameters dialog box, which is accessed by selecting **Parameters** from the **Solve** menu.

The problem can now be solved. Pressing the **=** button solves the heat equation at 11 different times from 0 to 5 seconds. By default, an interpolated plot of the solution, i.e., the heat distribution, at the end of the time span is displayed.

A more interesting way to visualize the dynamics of the heat distribution process is to *animate* the solution. To start an animation, check the **Animation** check box in the Plot selection dialog box. Also, select the colormap `hot`. Click the **Plot** button to start a recording of the solution plots in a separate figure window. The recorded animation is then "played" five times.

The temperature in the block rises very quickly. To improve the animation and focus on the first second, try to change the list of times to the MATLAB expression `logspace(-2,0.5,20)`.

Also, try to change the heat capacity coefficient `d` and the heat flow at the rightmost boundary to see how they affect the heat distribution.



Using Command-Line Functions. First, you must create geometry and boundary condition files. The files used here were created using `pdetool`. The geometry of the metal block is described in `crackg.m`, and the boundary conditions can be found in `crackb.m`.

To create an initial mesh, call `initmesh`:

```
» [p,e,t]=initmesh('crackg');
```

The heat equation can now be solved using the Partial Differential Equation Toolbox function `parabolic`. The generic parabolic PDE that `parabolic` solves is

$$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f$$

with initial value $u0 = u(t_0)$ and the times at which to compute a solution specified in the array `tlist`. For this case, you have $d = 1$, $c = 1$, $a = 0$, and $f = 0$. The initial value $u0 = 0$, and the list of times, `tlist`, is set to the MATLAB array `0:0.5:5`.

To compute the solution, call `parabolic`:

```
u=parabolic(0,0:0.5:5,'crackb',p,e,t,1,0,0,1);
```

The solution u created this way a matrix with 11 columns, where each column corresponds to the solution at the 11 points in time 0,0.5, . . . ,4.5,5.0.

Let us plot the solution at $t = 5.0$ seconds using interpolated shading and a hidden mesh. Use the hot colormap:

```
 pdeplot(p,e,t,'xydata',u(:,11),'mesh','off',...
'colormap','hot')
```

Heat Distribution in Radioactive Rod

This heat distribution problem is an example of a 3-D parabolic PDE problem that is reduced to a 2-D problem by using cylindrical coordinates.

Consider a cylindrical radioactive rod. At the left end, heat is continuously added. The right end is kept at a constant temperature. At the outer boundary, heat is exchanged with the surroundings by transfer. At the same time, heat is uniformly produced in the whole rod due to radioactive processes. Assume that the initial temperature is zero. This leads to the following problem:

$$\rho C\frac{\partial u}{\partial t} - \nabla \cdot (k\nabla u) = f$$

where ρ is the density, C is the rod’s thermal capacity, k is the thermal conductivity, and f is the radioactive heat source.

The density for this metal rod is 7800 kg/m^3, the thermal capacity is 500 Ws/kgºC, and the thermal conductivity is 40 W/mºC. The heat source is 20000 W/m^3. The temperature at the right end is 100 ºC. The surrounding temperature at the outer boundary is 100 ºC, and the heat transfer coefficient is 50 W/m^{2}ºC. The heat flux at the left end is 5000 W/m^2.

But this is a cylindrical problem, so you need to transform the equation, using the cylindrical coordinates r, z, and θ. Due to symmetry, the solution is independent of θ, so the transformed equation is

$$r\rho C\frac{\partial u}{\partial t} - \frac{\partial}{\partial r}\left(kr\frac{\partial u}{\partial r}\right) - \frac{\partial}{\partial z}\left(kr\frac{\partial u}{\partial z}\right) = fr$$

The boundary conditions are:

- $\vec{n} \cdot (k\nabla u)$ = 5000 at the left end of the rod (Neumann condition). Since the generalized Neumann condition in Partial Differential Equation Toolbox software is $\vec{n} \cdot (c\nabla u) + qu = g$, and c depends on r in this problem ($c = kr$), this boundary condition is expressed as $\vec{n} \cdot (c\nabla u) = 5000r$
- u = 100 at the right end of the rod (Dirichlet condition).
- $\vec{n} \cdot (k\nabla u)$ = 50(100-u) at the outer boundary (generalized Neumann condition). In Partial Differential Equation Toolbox software, this must be expressed as $\vec{n} \cdot (c\nabla u) + 50r \cdot u = 50r \cdot 100$.
- The cylinder axis r = 0 is not a boundary in the original problem, but in our 2-D treatment it has become one. We must give the *artificial* boundary condition $\vec{n} \cdot (c\nabla u) = 0$ here.

The initial value is $u(t_0) = 0$.

Using the Graphical User Interface. Solve this problem using the `pdetool` GUI. Model the rod as a rectangle with its base along the x-axis, and let the x-axis be the z direction and the y-axis be the r direction. A rectangle with corners in (-1.5,0), (1.5,0), (1.5,0.2), and (-1.5,0.2) would then model a rod with length 3 and radius 0.2.

Enter the boundary conditions by double-clicking the boundaries to open the Boundary Condition dialog box. For the left end, use Neumann conditions with `0` for `q` and `5000*y` for `g`. For the right end, use Dirichlet conditions with `1` for `h` and `100` for `r`. For the outer boundary, use Neumann conditions with `50*y` for `q` and `50*y*100` for `g`. For the axis, use Neumann conditions with `0` for `q` and `g`.

Enter the coefficients into the PDE Specification dialog box: `c` is `40*y`, `a` is zero, `d` is `7800*500*y`, and `f` is `20000*y`.

Animate the solution over a span of 20000 seconds (computing the solution every 1000 seconds). We can see how heat flows in over the right and outer boundaries as long as $u < 100$, and out when $u > 100$. You can also open the PDE Specification dialog box, and change the PDE type to **Elliptic**. This shows the solution when u does not depend on time, i.e., the steady state solution. The profound effect of cooling on the outer boundary can be demonstrated by setting the heat transfer coefficient to zero.

Hyperbolic Problem

This section describes the solution of a hyperbolic PDE problem. The problem is solved using the Partial Differential Equation Toolbox graphical user interface (GUI) and command-line functions.

The Wave Equation

As an example of a hyperbolic PDE, let us solve the *wave equation*

$$\frac{\partial^2 u}{\partial t^2} - \Delta u = 0$$

for transverse vibrations of a membrane on a square with corners in (-1,-1), (-1,1), (1,-1), and (1,1). The membrane is fixed ($u = 0$) at the left and right sides, and is free

$$\left(\frac{\partial u}{\partial n} = 0\right)$$

at the upper and lower sides. Additionally, we need initial values for

$u(t_0)$ and $\frac{\partial u(t_0)}{\partial t}$

The initial values need to match the boundary conditions for the solution to be well-behaved. If we start at t=0,

$$u(0) = \text{atan}\left(\cos\left(\frac{\pi}{2}x\right)\right)$$

and

$$\frac{\partial u(0)}{\partial t} = 3\sin(\pi x)e^{\sin\left(\frac{\pi}{2}y\right)}$$

are initial values that satisfy the boundary conditions. The reason for the *arctan* and *exponential* functions is to introduce more modes into the solution.

Using the Graphical User Interface. Use the `pdetool` GUI in the generic scalar mode. Draw the square using the **Rectangle/square** option from the **Draw** menu or the button with the rectangle icon. Proceed to define the boundary conditions by clicking the $\partial\Omega$ button and then double-click the boundaries to define the boundary conditions.

Initialize the mesh by clicking the Δ button or by selecting **Initialize mesh** from the **Mesh** menu.

Also, define the hyperbolic PDE by opening the PDE Specification dialog box, selecting the hyperbolic PDE, and entering the appropriate coefficient values. The general hyperbolic PDE is described by

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f$$

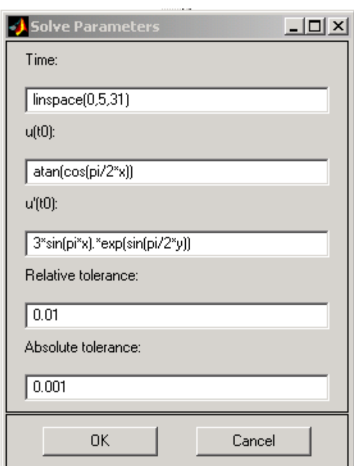
so for the wave equation you get $c = 1$, $a = 0$, $f = 0$, and $d = 1$.

Before solving the PDE, select **Parameters** from the **Solve** menu to open the Solve Parameters dialog box. As a list of times, enter `linspace(0,5,31)` and as initial values for u:

```
atan(cos(pi/2*x))
```

and for $\frac{\partial u}{\partial t}$, enter

```
3*sin(pi*x).*exp(sin(pi/2*y))
```



Finally, click the **=** button to compute the solution. The best plot for viewing the waves moving in the *x* and *y* directions is an animation of the whole sequence of solutions. Animation is a very real time and memory consuming feature, so you may have to cut down on the number of times at which to compute a solution. A good suggestion is to check the **Plot in x-y grid** option. Using an *x*-*y* grid can speed up the animation process significantly.

Using Command-Line Functions. From the command line, solve the equation with the preceding boundary conditions and the initial values, starting at time 0 and then every 0.05 seconds for five seconds.

The geometry is described in the file `squareg.m` and the boundary conditions in the file `squareb3.m`. The following sequence of commands then generates a solution and animates it. First, create a mesh and define the initial values and the times for which you want to solve the equation:

```
[p,e,t]=initmesh('squareg');

x=p(1,:)'; y=p(2,:)';

u0=atan(cos(pi/2*x));
ut0=3*sin(pi*x).*exp(sin(pi/2*y));

n=31;
```

```
tlist=linspace(0,5,n); % list of times
```

You are now ready to solve the wave equation. The general form for the hyperbolic PDE in Partial Differential Equation Toolbox software is

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f$$

so here you have $d = 1$, $c = 1$, $a = 0$, and $f = 0$:

```
uu=hyperbolic(u0,ut0,tlist,'squareb3',p,e,t,1,0,0,1);
```

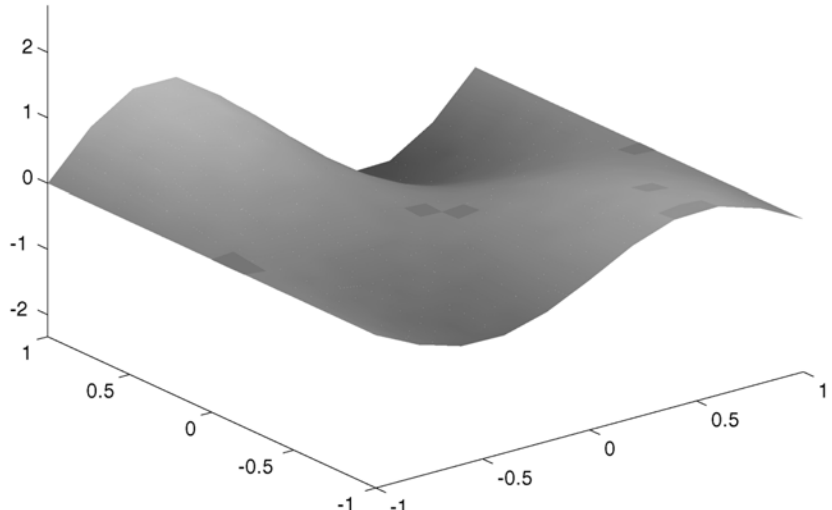
To visualize the solution, you can animate it. Interpolate to a rectangular grid to speed up the plotting:

```
delta=-1:0.1:1;
[uxy,tn,a2,a3]=tri2grid(p,t,uu(:,1),delta,delta);
gp=[tn;a2;a3];

umax=max(max(uu));
umin=min(min(uu));

newplot
M=moviein(n);
for i=1:n,
pdeplot(p,e,t,'xydata',uu(:,i),'zdata',uu(:,i),...
'mesh','off','xygrid','on','gridparam',gp,...
'colorbar','off','zstyle','continuous');
axis([-1 1 -1 1 umin umax]); caxis([umin umax]);
M(:,i)=getframe;
end
movie(M,10);
```

You can find a complete demo of this problem, including animation, in `pdedemo6`. If you have lots of memory, you can try increasing `n`, the number of frames in the movie.



Animation of the Solution to the Wave Equation

Eigenvalue Problems

This section describes the solution of some eigenvalue PDE problems. The problems are solved using thePartial Differential Equation Toolbox graphical user interface (GUI) and command-line functions. The problems include:

- "Eigenvalues and Eigenfunctions for the L-Shaped Membrane" on page 1-75
- "L-Shaped Membrane with Rounded Corner" on page 1-80
- "Eigenvalues and Eigenmodes of a Square" on page 1-81

Eigenvalues and Eigenfunctions for the L-Shaped Membrane

The problem of finding the eigenvalues and the corresponding eigenfunctions of an L-shaped membrane is of interest to all MATLAB users, since the plot of the first eigenfunction is the logo of The MathWorks™. In fact, you can compare the eigenvalues and eigenfunctions computed by Partial Differential

Equation Toolbox software to the ones produced by the MATLAB function `membrane`.

The problem is to compute all eigenmodes with eigenvalues < 100 for the *eigenmode PDE problem*

$$-\Delta u = \lambda u$$

on the geometry of the L-shaped membrane. $u = 0$ on the boundary (Dirichlet condition).

Using the Graphical User Interface. With the `pdetool` GUI active, check that the current mode is set to **Generic Scalar**. Then draw the L-shape as a polygon with corners in (0,0), (-1,0), (-1,-1), (1,-1), (1,1), and (0,1).

There is no need to define any boundary conditions for this problem since the default condition—$u = 0$ on the boundary—is the correct one. Therefore, you can continue to the next step: to initialize the mesh. Refine the initial mesh twice. Defining the eigenvalue PDE problem is also easy. Open the PDE Specification dialog box and select **Eigenmodes**. The default values for the PDE coefficients, $c = 1$, $a = 0$, $d = 1$, all match the problem description, so you can exit the PDE Specification dialog box by clicking the **OK** button.

Open the Solve Parameters dialog box by selecting **Parameters** from the **Solve** menu. The dialog box contains an edit box for entering the eigenvalue search range. The default entry is `[0 100]`, which is just what you want.

Finally, solve the L-shaped membrane problem by clicking the **=** button. The solution displayed is the first eigenfunction. The value of the first (smallest) eigenvalue is also displayed. You find the number of eigenvalues on the information line at the bottom of the GUI. You can open the Plot Selection dialog box and choose which eigenfunction to plot by selecting from a pop-up menu of the corresponding eigenvalues.

Using Command-Line Functions. The geometry of the L-shaped membrane is described in the file `lshapeg`.m and the boundary conditions in the file `lshapeb.m`.

First, initialize the mesh and refine it twice using the command line functions at the MATLAB prompt:

```
[p,e,t]=initmesh('lshapeg');
[p,e,t]=refinemesh('lshapeg',p,e,t);
[p,e,t]=refinemesh('lshapeg',p,e,t);
```

Recall the general eigenvalue PDE problem description:

$$-\nabla \cdot (c\nabla u) + au = \lambda du$$

This means that in this case you have $c = 1$, $a = 0$, and $d = 1$. The syntax of pdeeig, the Partial Differential Equation Toolbox eigenvalue solver, is

```
[v,l]=pdeeig(b,p,e,t,c,a,d,r)
```

The input argument r is a two-element vector indicating the interval on the real axis where pdeeig searches for eigenvalues. Here you are looking for eigenvalues < 100, so the interval you use is [0 100].

Now you can call pdeeig and see how many eigenvalues you find:

```
[v,l]=pdeeig('lshapeb',p,e,t,1,0,1,[0 100]);
```

There are 19 eigenvalues smaller than 100. Plot the first eigenmode and compare it to the MATLAB membrane function:

```
pdesurf(p,t,v(:,1))
figure
membrane(1,20,9,9)
```

membrane can produce the first 12 eigenfunctions for the L-shaped membrane. Compare also the 12th eigenmodes:

```
figure
pdesurf(p,t,v(:,12))
figure
membrane(12,20,9,9)
```

Looking at the following eigenmodes, you can see how the number of oscillations increases. The eigenfunctions are symmetric or antisymmetric around the diagonal from (0,0) to (1,-1), which divides the L-shaped membrane into two mirror images. In a practical computation, you could take advantage of such symmetries in the PDE problem, and solve over a region half the size. The eigenvalues of the full L-shaped membrane are the union of those of the

half with Dirichlet boundary condition along the diagonal (eigenvalues 2, 4, 7, 11, 13, 16, and 17) and those with Neumann boundary condition (eigenvalues 1, 3, 5, 6, 10, 12, 14, and 15).

The eigenvalues λ_8 and λ_9 make up a double eigenvalue for the PDE at around 49.64. Also, the eigenvalues λ_{18} and λ_{19} make up another double eigenvalue at around 99.87. You may have gotten two different but close values. The default triangulation made by `initmesh` is not symmetric around the diagonal, but a symmetric grid gives a matrix with a true double eigenvalue. Each of the eigenfunctions u_8 and u_9 consists of three copies of eigenfunctions over the unit square, corresponding to its double second eigenvalue. You may not have obtained the zero values along a diagonal of the square—any line through the center of the square may have been computed. This shows a general fact about multiple eigenvalues for symmetric matrices; namely that any vector in the invariant subspace is equally valid as an eigenvector. The two eigenfunctions u_8 and u_9 are orthogonal to each other if the dividing lines make right angles. Check your solutions for that.

Actually, the eigenvalues of the square can be computed exactly. They are

$$(m^2+n^2)\pi^2$$

e.g., the double eigenvalue λ_{18} and λ_{19} is $10\pi^2$, which is pretty close to 100.

If you compute the FEM approximation with only one refinement, you would only find 16 eigenvalues, and you obtain the wrong solution to the original problem. You can of course check for this situation by computing the eigenvalues over a slightly larger range than the original problem.

You get some information from the printout in the MATLAB command window that is printed during the computation. For this problem, the algorithm computed a new set of eigenvalue approximations and tested for convergence every third step. In the output, you get the step number, the time in seconds since the start of the eigenvalue computation, and the number of converged eigenvalues with eigenvalues both inside and outside the interval counted.

Here is what MATLAB wrote:

```
Basis= 10,  Time=   2.70,  New conv eig=  0
Basis= 13,  Time=   3.50,  New conv eig=  0
```

```
                Basis= 16,  Time=   4.36,  New conv eig=  0
                Basis= 19,  Time=   5.34,  New conv eig=  1
                Basis= 22,  Time=   6.46,  New conv eig=  2
                Basis= 25,  Time=   7.61,  New conv eig=  3
                Basis= 28,  Time=   8.86,  New conv eig=  3
                Basis= 31,  Time=  10.23,  New conv eig=  5
                Basis= 34,  Time=  11.69,  New conv eig=  5
                Basis= 37,  Time=  13.28,  New conv eig=  7
                Basis= 40,  Time=  14.97,  New conv eig=  8
                Basis= 43,  Time=  16.77,  New conv eig=  9
                Basis= 46,  Time=  18.70,  New conv eig= 11
                Basis= 49,  Time=  20.73,  New conv eig= 11
                Basis= 52,  Time=  22.90,  New conv eig= 13
                Basis= 55,  Time=  25.13,  New conv eig= 14
                Basis= 58,  Time=  27.58,  New conv eig= 14
                Basis= 61,  Time=  30.13,  New conv eig= 15
                Basis= 64,  Time=  32.83,  New conv eig= 16
                Basis= 67,  Time=  35.64,  New conv eig= 18
                Basis= 70,  Time=  38.62,  New conv eig= 22
 End of sweep:  Basis= 70,  Time=  38.62,  New conv eig= 22
                Basis= 32,  Time=  43.29,  New conv eig=  0
                Basis= 35,  Time=  44.70,  New conv eig=  0
                Basis= 38,  Time=  46.22,  New conv eig=  0
                Basis= 41,  Time=  47.81,  New conv eig=  0
                Basis= 44,  Time=  49.52,  New conv eig=  0
                Basis= 47,  Time=  51.35,  New conv eig=  0
                Basis= 50,  Time=  53.27,  New conv eig=  0
                Basis= 53,  Time=  55.30,  New conv eig=  0
 End of sweep:  Basis= 53,  Time=  55.30,  New conv eig=  0
```

You can see that two Arnoldi runs were made. In the first, 22 eigenvalues converged after a basis of size 70 was computed; in the second, where the vectors were orthogonalized against all the 22 converged vectors, the smallest eigenvalue stabilized at a value outside of the interval [0, 100], so the algorithm signaled convergence. Of the 22 converged eigenvalues, 19 were inside the search interval.

L-Shaped Membrane with Rounded Corner

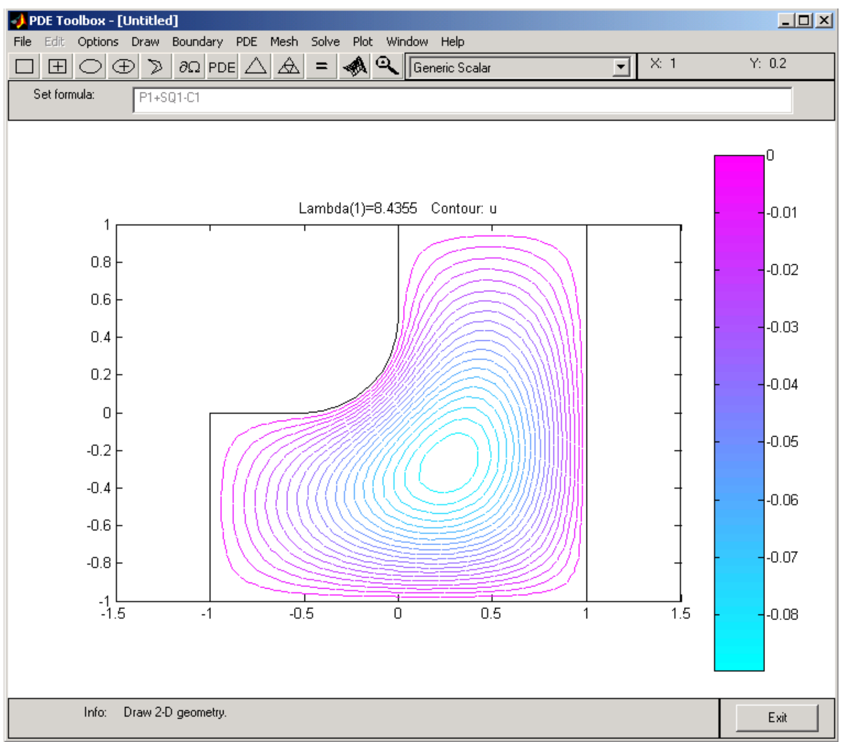
An extension of this problem is to compute the eigenvalues for an L-shaped membrane where the inner corner at the "knee" is rounded. The roundness is created by adding a circle so that the circle's arc is a part of the L-shaped membrane's boundary. By varying the circle's radius, the degree of roundness can be controlled. The `lshapec` file is an extension of an ordinary model file created using `pdetool`. It contains the lines

```
pdepoly([-1, 1, 1, 0, 0, -1],...
        [-1, -1, 1, 1, 0, 0],'P1');
pdecirc(-a,a,a,'C1');
pderect([-a 0 a 0],'SQ1');
```

The extra circle and rectangle that are added using `pdecirc` and `pderect` to create the rounded corner are affected by the added input argument a through a couple of extra lines of MATLAB code. This is possible since Partial Differential Equation Toolbox software is a part of the open MATLAB environment.

With `lshapec` you can create L-shaped rounded geometries with different degrees of roundness. If you use `lshapec` without an input argument, a default radius of 0.5 is used. Otherwise, use `lshapec(a)`, where `a` is the radius of the circle.

Experimenting using different values for the radius `a` shows you that the eigenvalues and the frequencies of the corresponding eigenmodes decrease as the radius increases, and the shape of the L-shaped membrane becomes more rounded. In the following figure, the first eigenmode of an L-shaped membrane with a rounded corner is plotted.



First Eigenmode for an L-Shaped Membrane with a Rounded Corner

Eigenvalues and Eigenmodes of a Square

Let us study the eigenvalues and eigenmodes of a square with an interesting set of boundary conditions. The square has corners in (-1,-1), (-1,1), (1,1), and (1,-1). The boundary conditions are as follows:

- On the left boundary, the Dirichlet condition $u = 0$
- On the upper and lower boundary, the Neumann condition

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

- On the right boundary, the generalized Neumann condition

$$\frac{\partial u}{\partial n} - \frac{3}{4}u = 0$$

The eigenvalue PDE problem is

$$-\Delta u = \lambda u$$

We are interested in the eigenvalues smaller than 10 and the corresponding eigenmodes, so the search range is `[-Inf 10]`. The sign in the generalized Neumann condition is such that there are negative eigenvalues.

Using the Graphical User Interface. Using the `pdetool` GUI in the generic scalar mode, draw the square using the **Rectangle/square** option from the **Draw** menu or the button with the rectangle icon. Then define the boundary conditions by clicking the $\partial\Omega$ button and then double-click the boundaries to define the boundary conditions. On the right side boundary, you have the generalized Neumann conditions, and you enter them as constants: g = 0 and q = -3/4.

Initialize the mesh and refine it once by clicking the Δ and **refine** buttons or by selecting the corresponding options from the **Mesh** menu.

Also, define the eigenvalue PDE problem by opening the PDE Specification dialog box and selecting the **Eigenmodes** option. The general eigenvalue PDE is described by

$$-\nabla \cdot (c\nabla u) + au = \lambda du$$

so for this problem you use the default values c = 1, a = 0, and d = 1. Also, in the Solve Parameters dialog box, enter the eigenvalue range as the MATLAB vector `[-Inf 10]`.

Finally, click the **=** button to compute the solution. By default, the first eigenfunction is plotted. You can plot the other eigenfunctions by selecting the corresponding eigenvalue from a pop-up menu in the Plot Selection dialog box. The pop-up menu contains all the eigenvalues found in the specified range. You can also export the eigenfunctions and eigenvalues to the MATLAB main workspace by using the **Export Solution** option from the **Solve** menu.

Using Command-Line Functions. The geometry description file and boundary condition file for this problem are called squareg.m and squareb2.m, respectively. Use the following sequence of commands to find the eigenvalues in the specified range and the corresponding eigenfunctions:

```
[p,e,t]=initmesh('squareg');
[p,e,t]=refinemesh('squareg',p,e,t);
```

The eigenvalue PDE coefficients c, a, and d for this problem are $c = 1$, $a = 0$, and $d = 1$. You can enter the eigenvalue range r as the MATLAB vector [-Inf 10]. pdeeig returns two output arguments, the eigenvalues as an array l and a matrix v of corresponding eigenfunctions:

```
[v,l]=pdeeig('squareb2',p,e,t,1,0,1,[-Inf 10]);
```

To plot the fourth eigenfunction as a surface plot, type

```
pdesurf(p,t,v(:,4))
```

This problem is *separable*, i.e.,

$$u(x, y) = f(x)g(y)$$

The functions f and g are eigenfunctions in the x and y directions, respectively. In the x direction, the first eigenmode is a slowly increasing exponential function. The higher modes include sinusoids. In the y direction, the first eigenmode is a straight line (constant), the second is half a cosine, the third is a full cosine, the fourth is one and a half full cosines, etc. These eigenmodes in the y direction are associated with the eigenvalues:

$$0, \frac{\pi^2}{4}, \frac{4\pi^2}{4}, \frac{9\pi^2}{4}, \ldots$$

There are five eigenvalues smaller than 10 for this problem, and the first one is even negative (-0.4145). It is possible to trace the preceding eigenvalues in the eigenvalues of the solution. Looking at a plot of the first eigenmode, you can see that it is made up of the first eigenmodes in the x and y directions. The second eigenmode is made up of the first eigenmode in the x direction and the second eigenmode in the y direction.

Look at the difference between the first and the second eigenvalue:

```
l(2)-l(1)
ans =
    2.4740
pi*pi/4
ans =
    2.4674
```

Likewise, the fifth eigenmode is made up of the first eigenmode in the x direction and the third eigenmode in the y direction. As expected, `l(5)-l(1)` is approximately equal to π^2. You can explore higher modes by increasing the search range to include eigenvalues greater than 10.

Application Modes

In this section we describe the application modes of the `pdetool` graphical user interface (GUI). Examples are given for a variety of applications to different engineering problems.

The Application Modes and the GUI

Partial Differential Equation Toolbox software can be applied to a great number of problems in engineering and science. To help you in using the `pdetool` GUI for some important and common applications, eight different application modes are available in addition to the generic scalar and system modes.

The available application modes are:

- Generic scalar (the default mode)
- Generic system
- "Structural Mechanics — Plane Stress" on page 1-86
- "Structural Mechanics — Plane Strain" on page 1-92
- "Electrostatics" on page 1-93
- "Magnetostatics" on page 1-96
- "AC Power Electromagnetics" on page 1-103
- "Conductive Media DC" on page 1-108

- "Heat Transfer" on page 1-114
- "Diffusion" on page 1-117

Note From the GUI, the system PDEs are restricted to problems with vector valued *u* of dimension two. Using command-line functions, there is no formal restriction on the dimension of *u*.

The application mode can be selected directly from the pop-up menu in the upper right part of the GUI or by selecting an application from the **Application** submenu in the **Options** menu. Changing the application resets all PDE coefficients and boundary conditions to the default values for that specific application mode.

When using an application mode, the generic PDE coefficients are replaced by application-specific parameters such as Young's modulus for problems in structural mechanics. The application-specific parameters are entered by selecting **Parameters** from the **PDE** menu or by clicking the **PDE** button. You can also access the PDE parameters by double-clicking a subdomain, if you are in the PDE mode. That way it is possible to define PDE parameters for problems with regions of different material properties. The Boundary condition dialog box is also altered so that the Description column reflects the physical meaning of the different boundary condition coefficients. Finally, the Plot Selection dialog box allows you to visualize the relevant physical variables for the selected application.

Note In the **User entry** options in the Plot Selection dialog box, the solution and its derivatives are always referred to as `u`, `ux`, and `uy` (`v`, `vx`, and `vy` for the system cases) even if the application mode is nongeneric and the solution of the application-specific PDE normally is named, e.g., `V` or `T`.

In the remaining part of this section, we explain each of the application modes in more detail and give examples of how to solve application specific problems using Partial Differential Equation Toolbox functions.

Structural Mechanics — Plane Stress

In structural mechanics, the equations relating stress and strain arise from the balance of forces in the material medium. *Plane stress* is a condition that prevails in a flat plate in the *xy* plane, loaded only in its own plane and without *z*-direction restraint.

The stress-strain relation can then be written, assuming isotropic and isothermal conditions

$$\begin{pmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{pmatrix} = \frac{E}{1-\nu^2} \begin{pmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & (1-\nu)/2 \end{pmatrix} \begin{pmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{pmatrix}$$

where σx and σy are the stresses in the *x* and *y* directions, and τxy is the *shear stress.* The material properties are expressed as a combination of *E*, the *elastic modulus* or *Young's modulus*, and *ν*, Poisson's ratio.

The deformation of the material is described by the displacements in the *x* and *y* directions, *u* and *v*, from which the strains are defined as

$$\varepsilon_x = \frac{\partial u}{\partial x}, \varepsilon_y = \frac{\partial v}{\partial y}, \gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}$$

The balance of force equations are

$$-\frac{\partial \sigma_x}{\partial x} - \frac{\partial \tau_{xy}}{\partial y} = K_x$$

$$-\frac{\partial \tau_{xy}}{\partial x} - \frac{\partial \sigma_y}{\partial y} = K_y$$

where *K*x and *K*y are volume forces (body forces).

Combining the preceding relations, we arrive at the displacement equations, which can be written

$$-\nabla \cdot (\underline{\boldsymbol{c}} \otimes \nabla \boldsymbol{u}) = \underline{\boldsymbol{k}}$$

where $\underline{c}$ is a rank four tensor, which can be written as four 2-by-2 matrices c_{11}, c_{12}, c_{21}, and c_{22}:

$$c_{11} = \begin{pmatrix} 2G+\mu & 0 \\ 0 & G \end{pmatrix}$$

$$c_{12} = \begin{pmatrix} 0 & \mu \\ G & 0 \end{pmatrix}$$

$$c_{21} = \begin{pmatrix} 0 & G \\ \mu & 0 \end{pmatrix}$$

$$c_{22} = \begin{pmatrix} G & 0 \\ 0 & 2G+\mu \end{pmatrix}$$

where G, the *shear modulus,* is defined by

$$G = \frac{E}{2(1+\nu)}$$

and μ in turn is defined by

$$\mu = 2G\frac{v}{1-\nu}$$

$$\underline{\mathbf{k}} = \begin{pmatrix} K_x \\ K_y \end{pmatrix}$$

are *volume forces.*

This is an elliptic PDE of system type (u is two-dimensional), but you need only to set the application mode to **Structural Mechanics, Plane Stress** and then enter the material-dependent parameters E and v and the volume forces k into the PDE Specification dialog box.

In this mode, you can also solve the eigenvalue problem, which is described by

$$-\nabla \cdot (\underline{\boldsymbol{c}} \otimes \nabla \boldsymbol{u}) = \lambda \underline{\boldsymbol{d}} u$$

$$\underline{\boldsymbol{d}} = \begin{pmatrix} \rho & 0 \\ 0 & \rho \end{pmatrix}$$

ρ, the density, can also be entered using the PDE Specification dialog box.

In the Plot Selection dialog box, the *x*- and *y*-displacements, *u* and *v*, and the absolute value of the displacement vector (*u,v*) can be visualized using color, contour lines, or *z*-height, and the displacement vector field (*u,v*) can be plotted using arrows or a deformed mesh. In addition, for visualization using color, contour lines, or height, you can choose from 15 scalar tensor expressions:

- $ux = \frac{\partial u}{\partial x}$
- $uy = \frac{\partial u}{\partial y}$
- $vx = \frac{\partial v}{\partial x}$
- $vy = \frac{\partial v}{\partial y}$
- `exx`, the *x*-direction strain (ε_x)
- `eyy`, the *y*-direction strain (ε_y)
- `exy`, the shear strain (γ_{xy})
- `sxx`, the *x*-direction stress (σ_x)
- `syy`, the *y*-direction stress (σ_y)
- `sxy`, the shear stress (τ_{xy})
- `e1`, the first principal strain (ε_1)
- `e2`, the second principal strain (ε_2)
- `s1`, the first principal stress (σ_1)
- `s2`, the second principal stress (σ_2)
- `von Mises`, the von Mises effective stress

$(\sqrt{\sigma_1^2 + \sigma_2^2 - \sigma_1\sigma_2})$

For a more detailed discussion on the theory of stress-strain relations and applications of FEM to problems in structural mechanics, see Cook, Robert D., David S. Malkus, and Michael E. Plesha, *Concepts and Applications of Finite Element Analysis*, 3rd edition, John Wiley & Sons, New York, 1989.

Example. Consider a steel plate that is clamped along a right-angle inset at the lower-left corner, and pulled along a rounded cut at the upper-right corner. All other sides are free.

The steel plate has the following properties: Dimension: 1-by-1 meters; thickness 1 mm; inset is 1/3-by-1/3 meters. The rounded cut runs from (2/3, 1) to (1, 2/3). Young's modulus: $196 \cdot 10^3$ (MN/m^2), Poisson's ratio: 0.31.

The curved boundary is subjected to an outward normal load of 500 N/m. We need to specify a surface traction; we therefore divide by the thickness 1 mm, thus the surface tractions should be set to 0.5 MN/m^2. We will use the force unit MN in this example.

We want to compute a number of interesting quantities, such as the *x*- and *y*-direction strains and stresses, the shear stress, and the von Mises effective stress.

Using the Graphical User Interface. Using the `pdetool` GUI, set the application mode to **Structural Mechanics, Plane Stress**.

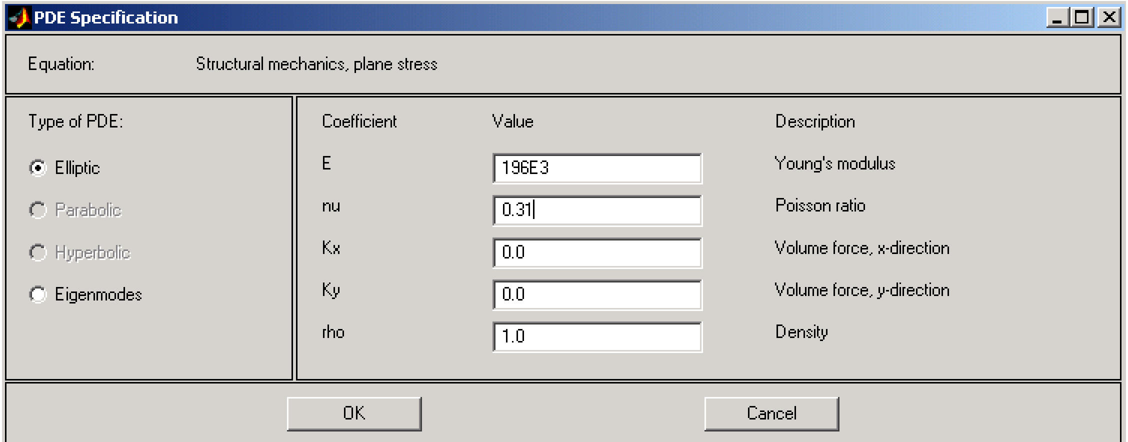
The CSG model can be made very quickly by drawing a polygon with corners in `x=[0 2/3 1 1 1/3 1/3 0]` and `y=[1 1 2/3 0 0 1/3 1/3]` and a circle with center in `x=2/3, y=2/3` and radius `1/3`.

The polygon is normally labeled P1 and the circle C1, and the CSG model of the steel plate is simply P1+C1.

Next, select **Boundary Mode** to specify the boundary conditions. First, remove all subdomain borders by selecting **Remove All Subdomain Borders** from the **Boundary** menu. The two boundaries at the inset in the lower left are clamped, i.e., Dirichlet conditions with zero displacements. The rounded cut is subject to a Neumann condition with `q=0` and `g1=0.5*nx`,

g2=0.5*ny. The remaining boundaries are free (no normal stress), that is, a Neumann condition with q=0 and g=0.

The next step is to open the PDE Specification dialog box and enter the PDE parameters.



The E and ν (nu) parameters are Young's modulus and Poisson's ratio, respectively. There are no volume forces, so Kx and Ky are zero. ρ (rho) is not used in this mode. The material is homogeneous, so the same E and [[v apply to the whole 2-D domain.

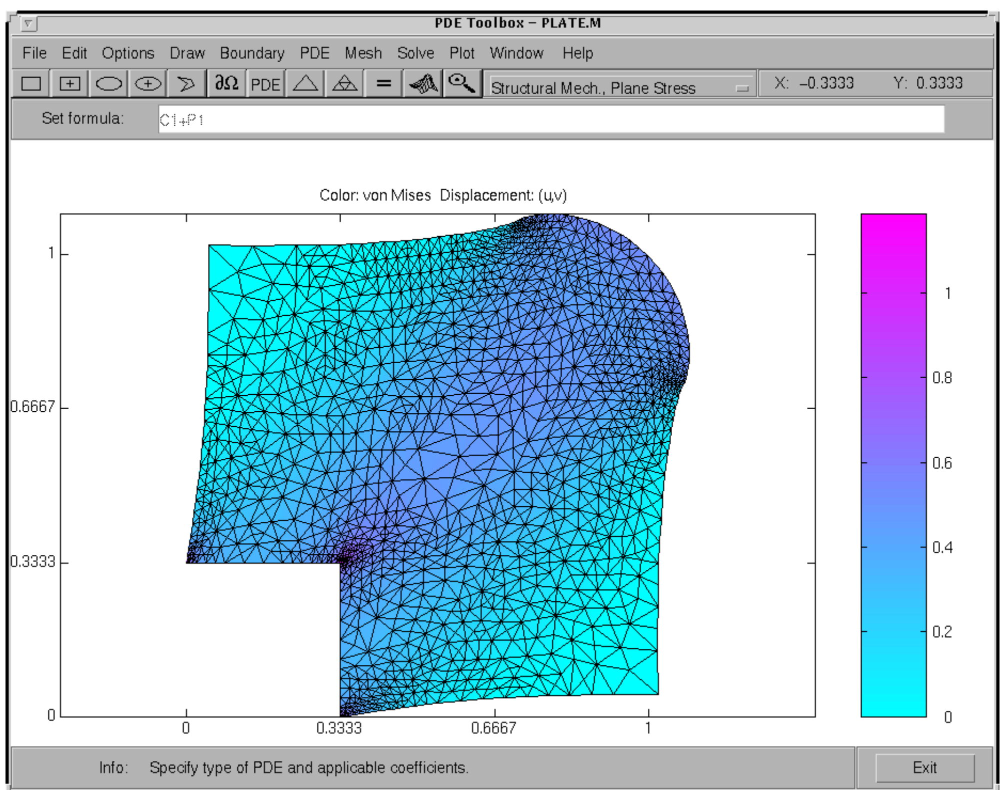
Initialize the mesh by clicking the Δ button. If you want, you can refine the mesh by clicking the **Refine** button.

The problem can now be solved by clicking the = button.

A number of different strain and stress properties can be visualized, such as the displacements u and v, the x- and y-direction strains and stresses, the shear stress, the von Mises effective stress, and the principal stresses and strains. All these properties can be selected from pop-up menus in the Plot Selection dialog box. A combination of scalar and vector properties can be plotted simultaneously by selecting different properties to be represented by color, height, vector field arrows, and displacements in a 3-D plot.

Select to plot the von Mises effective stress using color and the displacement vector field (*u,v*) using a deformed mesh. Select the **Color** and **Deformed mesh** plot types. To plot the von Mises effective stress, select `von Mises` from the pop-up menu in the **Color** row.

In areas where the gradient of the solution (the stress) is large, you need to refine the mesh to increase the accuracy of the solution. Select **Parameters** from the **Solve** menu and select the **Adaptive mode** check box. You can use the default options for adaptation, which are the **Worst triangles** triangle selection method with the **Worst triangle fraction** set to 0.5. Now solve the plane stress problem again. Select the **Show Mesh** option in the Plot Selection dialog box to see how the mesh is refined in areas where the stress is large.



Visualization of the von Mises Effective Stress and the Displacements Using Deformed Mesh

Structural Mechanics — Plane Strain

A deformation state where there are no displacements in the z-direction, and the displacements in the x- and y-directions are functions of x and y but not z is called *plane strain*. You can solve plane strain problems with Partial Differential Equation Toolbox software by setting the application mode to **Structural Mechanics, Plane Strain**. The stress-strain relation is only slightly different from the plane stress case, and the same set of material parameters is used. The application interfaces are identical for the two structural mechanics modes.

The places where the plane strain equations differ from the plane stress equations are:

- The μ parameter in the c tensor is defined as

$$\mu = 2G\frac{\nu}{1-2\nu}$$

- The von Mises effective stress is computed as

$$\sqrt{(\sigma_1^2+\sigma_2^2)(\nu^2-\nu+1)+\sigma_1\sigma_2(2\nu^2-2\nu-1)}$$

Plane strain problems are less common than plane stress problems. An example is a slice of an underground tunnel that lies along the z-axis. It deforms in essentially plane strain conditions.

Electrostatics

Applications involving electrostatics include high voltage apparatus, electronic devices, and capacitors. The "statics" implies that the time rate of change is slow, and that wavelengths are very large compared to the size of the domain of interest. In electrostatics, the electrostatic scalar potential V is related to the electric field E by $E = -\nabla V$ and, using one of *Maxwell's equations*, $\nabla \cdot D = \rho$ and the relationship $D = \varepsilon E$, we arrive at the Poisson equation

$$-\nabla \cdot (\varepsilon \nabla V) = \rho$$

where ε is the *coefficient of dielectricity* and ρ is the *space charge density*.

Note ε should really be written as $\varepsilon\varepsilon_0$, where ε_0 is the coefficient of dielectricity or permittivity of vacuum ($8.854 \cdot 10^{-12}$ farad/meter) and ε is the relative coefficient of dielectricity that varies among different dielectrics (1.00059 in air, 2.24 in transformer oil, etc.).

Using the Partial Differential Equation Toolbox electrostatics application mode, you can solve electrostatic problems modeled by the preceding equation.

The PDE Specification dialog box contains entries for ε and ρ.

The boundary conditions for electrostatic problems can be of Dirichlet or Neumann type. For Dirichlet conditions, the electrostatic potential V is specified on the boundary. For Neumann conditions, the *surface charge* $n \cdot (\varepsilon \nabla V)$ is specified on the boundary.

For visualization of the solution to an electrostatic problem, the plot selections include the electrostatic potential V, the electric field E, and the electric displacement field D.

For a more in-depth discussion of problems in electrostatics, see Popovic, Branko D., *Introductory Engineering Electromagnetics*, Addison-Wesley, Reading, MA, 1971.

Example. Let us consider the problem of determining the electrostatic potential in an air-filled quadratic "frame," bounded by a square with side length of 0.2 in the center and by outer limits with side length of 0.5. At the inner boundary, the electrostatic potential is 1000V. At the outer boundary, the electrostatic potential is 0V. There is no charge in the domain. This leads to the problem of solving the Laplace equation

$$\Delta V = 0$$

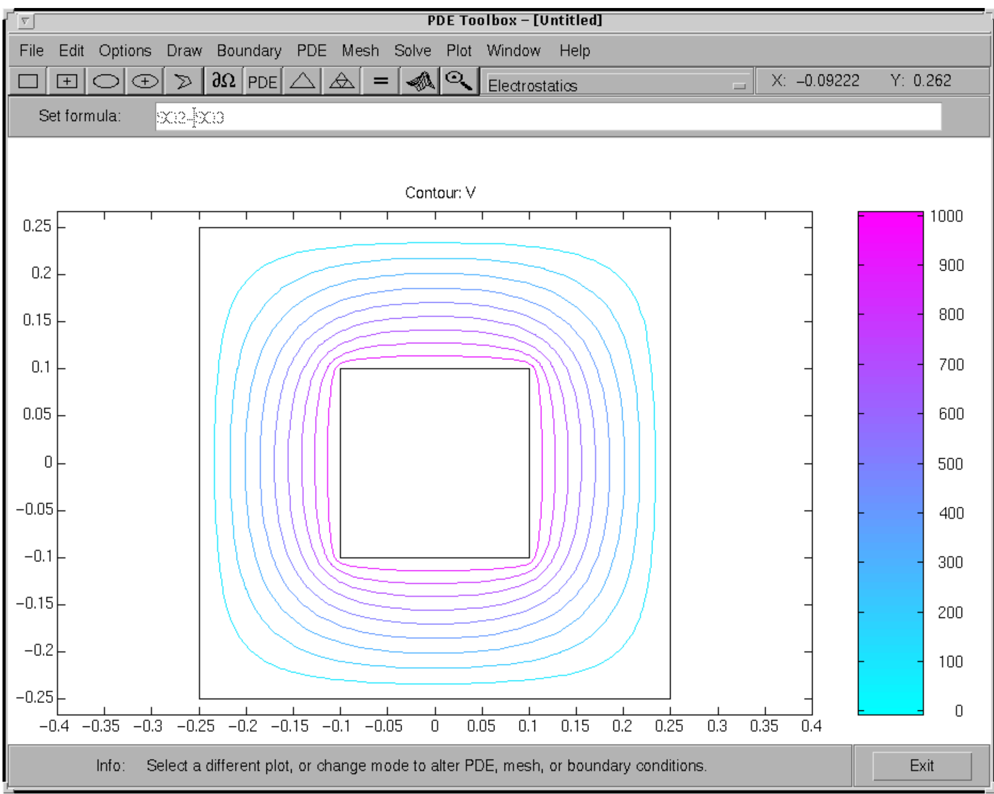
with the Dirichlet boundary conditions $V = 1000$ at the inner boundary, and $V = 0$ at the outer boundary.

Using the Graphical User Interface. After setting the application mode to **Electrostatics**, the 2-D area is most easily drawn by first drawing a square with sides of length 0.2 (use the **Snap** option and adjust the grid spacing if necessary). Then draw another square with sides of length 0.5 using the same center position. The 2-D domain is then simply SQ2-SQ1, if the first square is named SQ1 and the second square is named SQ2. Enter the expression into the **Set formula** edit box, and proceed to define the boundary conditions. Use **Shift**+click to select all the inner boundaries. Then double-click an inner boundary and enter 1000 as the Dirichlet boundary condition for the inner boundaries.

Next, open the PDE Specification dialog box, and enter `0` into the space charge density (`rho`) edit field. The coefficient of dielectricity can be left at 1, since it does not affect the result as long as it is constant.

Initialize the mesh, and click the **=** button to solve the equation. Using the adaptive mode, you can improve the accuracy of the solution by refining the mesh close to the reentrant corners where the gradients are steep. For example, use the triangle selection method picking the worst triangles and set the maximum number of triangles to 500. Add one uniform mesh refinement by clicking the **Refine** button once. Finally turn adaptive mode off, and click the **=** button once more.

To look at the equipotential lines, select a contour plot from the Plot Selection dialog box. To display equipotential lines at every 100th volt, enter `0:100:1000` into the **Contour plot levels** edit box.



Equipotential Lines in Air-Filled Frame

Magnetostatics

Magnets, electric motors, and transformers are areas where problems involving magnetostatics can be found. The "statics" implies that the time rate of change is slow, so we start with Maxwell's equations for steady cases,

$$\nabla \times \mathbf{H} = \mathbf{J}$$

$$\nabla \cdot \mathbf{B} = 0$$

and the relationship

$$\mathbf{B} = \mu\mathbf{H}$$

where $\mathbf{B}$ is the *magnetic flux density,* $\mathbf{H}$ is the *magnetic field intensity*, $\mathbf{J}$ is the *current density*, and μ is the material's *magnetic permeability*.

Since $\nabla \cdot \mathbf{B} = 0$, there exists a *magnetic vector potential* $\mathbf{A}$ such that

$$\mathbf{B} = \nabla \times \mathbf{A}$$

and

$$\nabla \times \left(\frac{1}{\mu}\nabla \times A\right) = J$$

The plane case assumes that the current flows are parallel to the z-axis, so only the z component of $\mathbf{A}$ is present,

$$\mathbf{A} = (0, 0, A), \mathbf{J} = (0, 0, J)$$

and the preceding equation can be simplified to the scalar elliptic PDE

$$-\nabla \cdot \left(\frac{1}{\mu}\nabla A\right) = J$$

where $J = J(x,y)$.

For the 2-D case, we can compute the magnetic flux density $\mathbf{B}$ as

$$\mathbf{B} = \left(\frac{\partial A}{\partial y}, -\frac{\partial A}{\partial x}, 0 \right)$$

and the magnetic field **H**, in turn, is given by

$$\mathbf{H} = \frac{1}{\mu}\mathbf{B}$$

The interface condition across subdomain borders between regions of different material properties is that H x n be continuous. This implies the continuity of

$$\frac{1}{\mu}\frac{\partial A}{\partial n}$$

and does not require special treatment since we are using the variational formulation of the PDE problem.

In ferromagnetic materials, μ is usually dependent on the field strength $|B| = |\nabla A|$, so the nonlinear solver is needed.

The Dirichlet boundary condition specifies the value of the magnetostatic potential A on the boundary. The Neumann condition specifies the value of the normal component of

$$n\left(\frac{1}{\mu}\nabla A\right)$$

on the boundary. This is equivalent to specifying the tangential value of the magnetic field H on the boundary.

Visualization of the magnetostatic potential A, the magnetic field H, and the magnetic flux density B is available. B and H can be plotted as vector fields.

For a more detailed discussion on Maxwell's equations and magnetostatics, see Popovic, Branko D., *Introductory Engineering Electromagnetics*, Addison-Wesley, Reading, MA, 1971.

Example. As an example of a problem in magnetostatics, consider determining the static magnetic field due to the stator windings in a two-pole electric motor. The motor is considered to be long, and when end effects are neglected, a 2-D computational model suffices.

The domain consists of three regions:

- Two ferromagnetic pieces, the stator and the rotor
- The air gap between the stator and the rotor
- The armature coil carrying the DC current

The magnetic permeability μ is 1 in the air and in the coil. In the stator and the rotor, μ is defined by

$$\mu = \frac{\mu_{max}}{1 + c\|\nabla(A)\|^2} + \mu_{min}$$

μmax = 5000, μmin = 200, and c = 0.05 are values that could represent transformer steel.

The current density J is 0 everywhere except in the coil, where it is 1.

The geometry of the problem makes the magnetic vector potential A symmetric with respect to y and antisymmetric with respect to x, so you can limit the domain to $x \geq 0, y \geq 0$ with the Neumann boundary condition

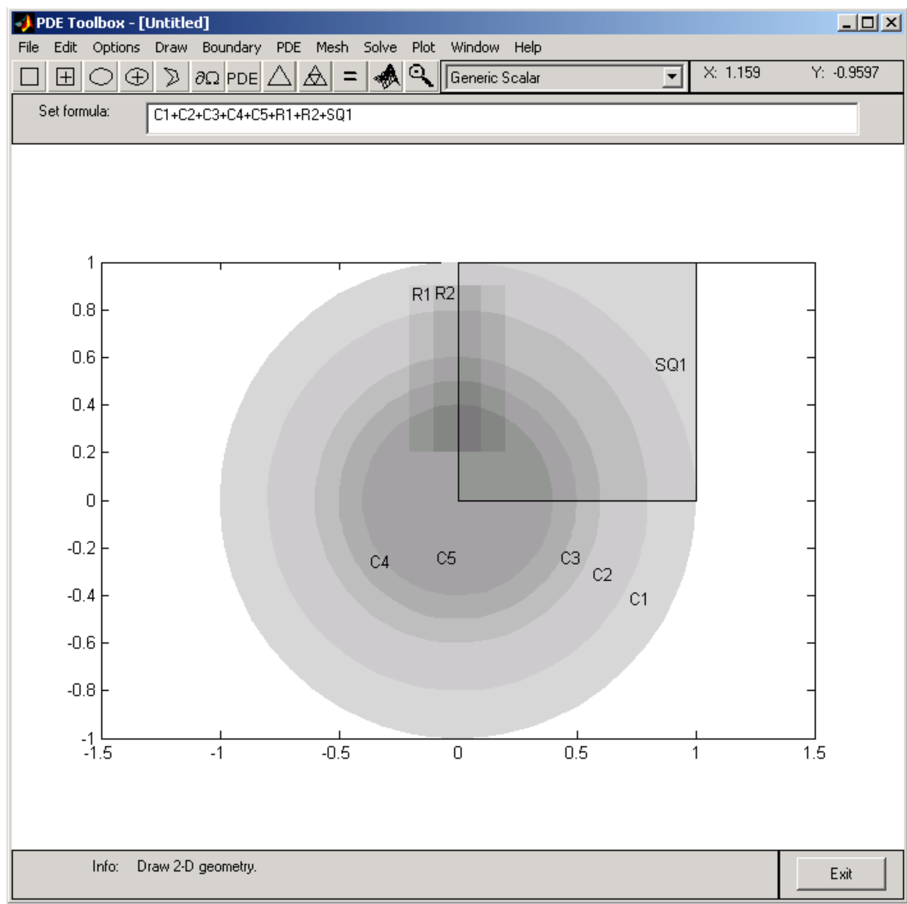
$$n \cdot \left(\frac{1}{\mu}\nabla A\right) = 0$$

on the x-axis and the Dirichlet boundary condition $A = 0$ on the y-axis. The field outside the motor is neglected leading to the Dirichlet boundary condition $A = 0$ on the exterior boundary.

Using the Graphical User Interface. The geometry is complex, involving five circular arcs and two rectangles. Using the `pdetool` GUI, set the x-axis limits to [-1.5 1.5] and the y-axis limits to `[-1 1]`. Set the application mode to **Magnetostatics**, and use a grid spacing of 0.1. The model is a union of circles and rectangles; the reduction to the first quadrant is achieved by intersection with a square. Using the "snap-to-grid" feature, you can draw the geometry using the mouse, or you can draw it by entering the following commands:

```
pdecirc(0,0,1,'C1')
pdecirc(0,0,0.8,'C2')
pdecirc(0,0,0.6,'C3')
pdecirc(0,0,0.5,'C4')
pdecirc(0,0,0.4,'C5')
pderect([-0.2 0.2 0.2 0.9],'R1')
pderect([-0.1 0.1 0.2 0.9],'R2')
pderect([0 1 0 1],'SQ1')
```

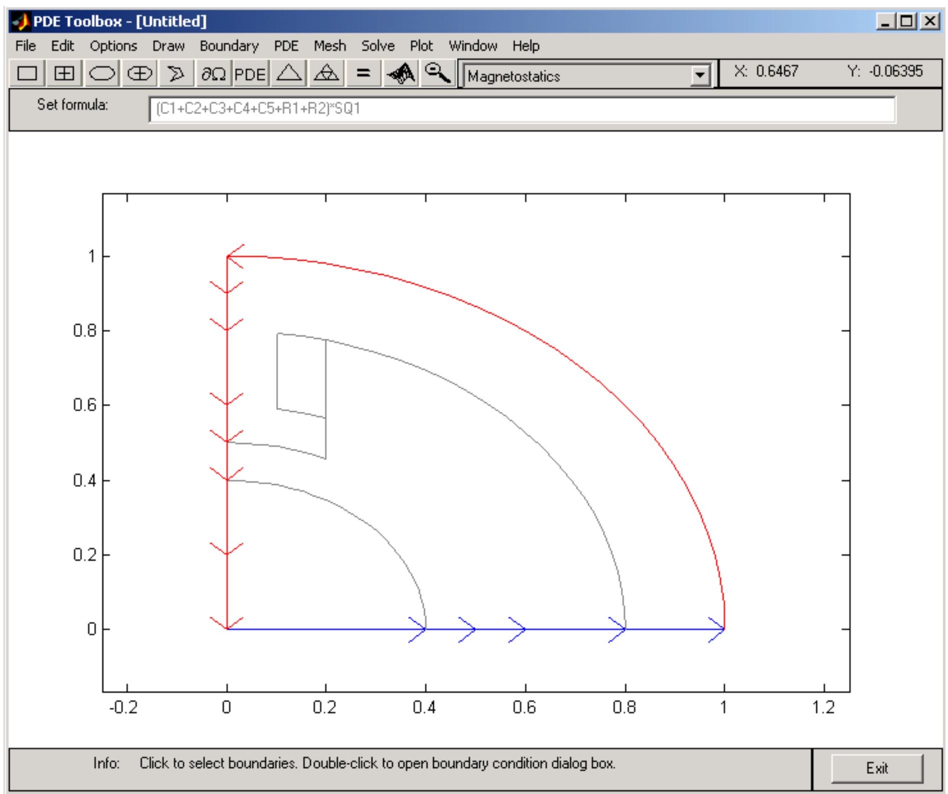
You should get a CSG model similar to the one in the following plot.



Enter the following set formula to reduce the model to the first quadrant:

```
(C1+C2+C3+C4+C5+R1+R2)*SQ1
```

In boundary mode you need to remove a number of subdomain borders. Using **Shift**+click, select borders and remove them using the **Remove Subdomain Border** option from the **Boundary** menu until the geometry consists of four subdomains: the stator, the rotor, the coil, and the air gap. In the following plot, the stator is subdomain 1, the rotor is subdomain 2, the coil is subdomain 3, and the air gap is subdomain 4. The numbering of your subdomains may be different.



Before moving to the PDE mode, select the boundaries along the x-axis and set the boundary condition to a Neumann condition with $g = 0$ and $q = 0$. In the PDE mode, turn on the labels by selecting the **Show Subdomain Labels** option from the **PDE** menu. Double-click each subdomain to define the PDE parameters:

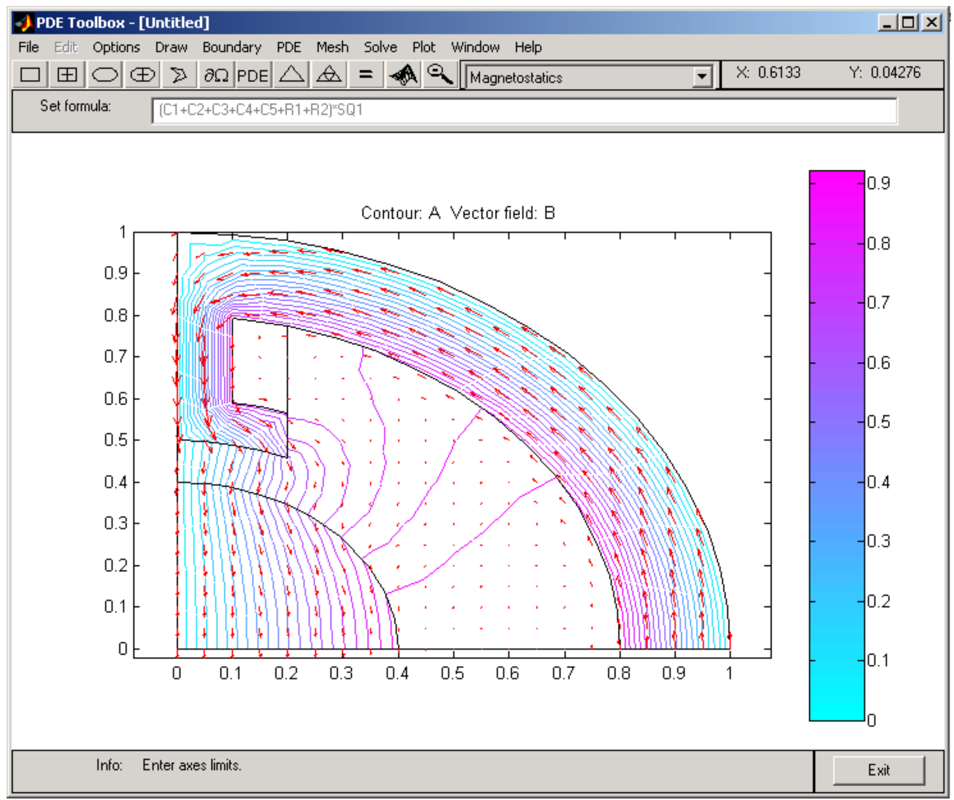
- In the coil both μ and J are 1, so the default values do not need to be changed.
- In the stator and the rotor μ is nonlinear and defined by the preceding equation. Enter μ as

```
5000./(1+0.05*(ux.^2+uy.^2))+200
```

  `ux.^2+uy.^2` is equal to $|\nabla A|^2$. J is 0 (no current).

- In the air gap μ is 1, and J is 0.

Initialize the mesh, and continue by opening the Solve Parameters dialog box by selecting **Parameters** from the **Solve** menu. Since this is a nonlinear problem, the nonlinear solver must be invoked by checking the **Use nonlinear solver**. If you want, you can adjust the tolerance parameter. The adaptive solver can be used together with the nonlinear solver. Solve the PDE and plot the magnetic flux density B using arrows and the equipotential lines of the magnetostatic potential A using a contour plot. The plot clearly shows, as expected, that the magnetic flux is parallel to the equipotential lines of the magnetostatic potential.



Equipotential Lines and Magnetic Flux in a Two-Pole Motor

AC Power Electromagnetics

AC power electromagnetics problems are found when studying motors, transformers and conductors carrying alternating currents.

Let us start by considering a homogeneous dielectric, with coefficient of dielectricity ε and magnetic permeability μ, with no charges at any point. The fields must satisfy a special set of the general Maxwell's equations:

$$\nabla \times \mathbf{E} = -\mu \frac{\partial \mathbf{H}}{\partial t}$$

$$\nabla \times \mathbf{H} = \varepsilon \frac{\partial \mathbf{E}}{\partial t} + \mathbf{J}$$

For a more detailed discussion on Maxwell's equations, see Popovic, Branko D., *Introductory Engineering Electromagnetics*, Addison-Wesley, Reading, MA, 1971.

In the absence of current, we can eliminate $\mathbf{H}$ from the first set and $\mathbf{E}$ from the second set and see that both fields satisfy wave equations with wave speed

$$\sqrt{\varepsilon\mu}$$

$$\Delta\mathbf{E} - \varepsilon\mu \frac{\partial^2 \mathbf{E}}{\partial t^2} = 0, \Delta\mathbf{H} - \varepsilon\mu \frac{\partial^2 \mathbf{H}}{\partial t^2} = 0$$

We move on to studying a charge-free homogeneous dielectric, with coefficient of dielectrics ε, magnetic permeability μ, and conductivity σ. The current density then is

$$\mathbf{J} = \sigma\mathbf{E}$$

and the waves are damped by the Ohmic resistance,

$$\Delta\mathbf{E} - \mu\sigma \frac{\partial \mathbf{E}}{\partial t} - \varepsilon\mu \frac{\partial^2 \mathbf{E}}{\partial t^2} = 0$$

and similarly for $\mathbf{H}$.

The case of time harmonic fields is treated by using the complex form, replacing **E** by

$$\mathbf{E_c} e^{j\omega t}$$

The plane case of this Partial Differential Equation Toolbox mode has

$\mathbf{E}_c = (0,0,E_c), \mathbf{J} = \left(0,0,Je^{i\omega t}\right)$, and the magnetic field

$$H = (H_x, H_y, 0) = -\frac{1}{j\mu\sigma}\nabla \times E_c$$

The scalar equation for E_c becomes

$$-\nabla \cdot \left(\frac{1}{\mu}\nabla E_c\right) + (j\omega\sigma - \omega^2\varepsilon)E_c = 0$$

This is the equation used by Partial Differential Equation Toolbox software in the AC power electromagnetics application mode. It is a complex Helmholtz's equation, describing the propagation of plane electromagnetic waves in imperfect dielectrics and good conductors ($\sigma \gg \omega\varepsilon$). A *complex permittivity* ε_c can be defined as $\varepsilon_c = \varepsilon - j\sigma/\omega$. The conditions at material interfaces with abrupt changes of ε and μ are the natural ones for the variational formulation and need no special attention.

The PDE parameters that have to be entered into the PDE Specification dialog box are the *angular frequency* ω, the magnetic permeability μ, the conductivity σ, and the coefficient of dielectricity ε.

The boundary conditions associated with this mode are a Dirichlet boundary condition, specifying the value of the electric field E_c on the boundary, and a Neumann condition, specifying the normal derivative of Ec. This is equivalent to specifying the tangential component of the magnetic field H:

$$H_t = \frac{j}{\omega} n \cdot \left(\frac{1}{\mu}\nabla E_c\right)$$

Interesting properties that can be computed from the solution—the electric field E—are the current density $J = \sigma E$ and the magnetic flux density

$$B = \frac{j}{\omega}\nabla \times E$$

The electric field E, the current density J, the magnetic field H and the magnetic flux density B are available for plots. Additionally, the resistive heating rate

$$Q = E_c^2/\sigma$$

is also available. The magnetic field and the magnetic flux density can be plotted as vector fields using arrows.

Example. The example demonstrates the *skin effect* when AC current is carried by a wire with circular cross section. The conductivity of copper is 57 $\cdot 10^6$, and the permeability is 1, i.e., $\mu = 4\pi 10^{-7}$. At the line frequency (50 Hz) the $\omega^2\varepsilon$-term is negligible.

Due to the induction, the current density in the interior of the conductor is smaller than at the outer surface where it is set to $J_S = 1$, a Dirichlet condition for the electric field, $E_c = 1/\sigma$. For this case an analytical solution is available,

$$J = J_S\frac{J_0(kr)}{J_0(kR)}$$

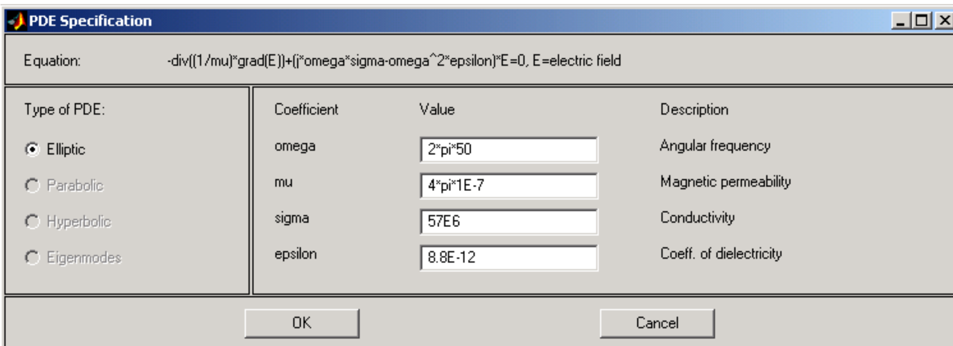
where

$$k = \sqrt{j\omega\mu\sigma}$$

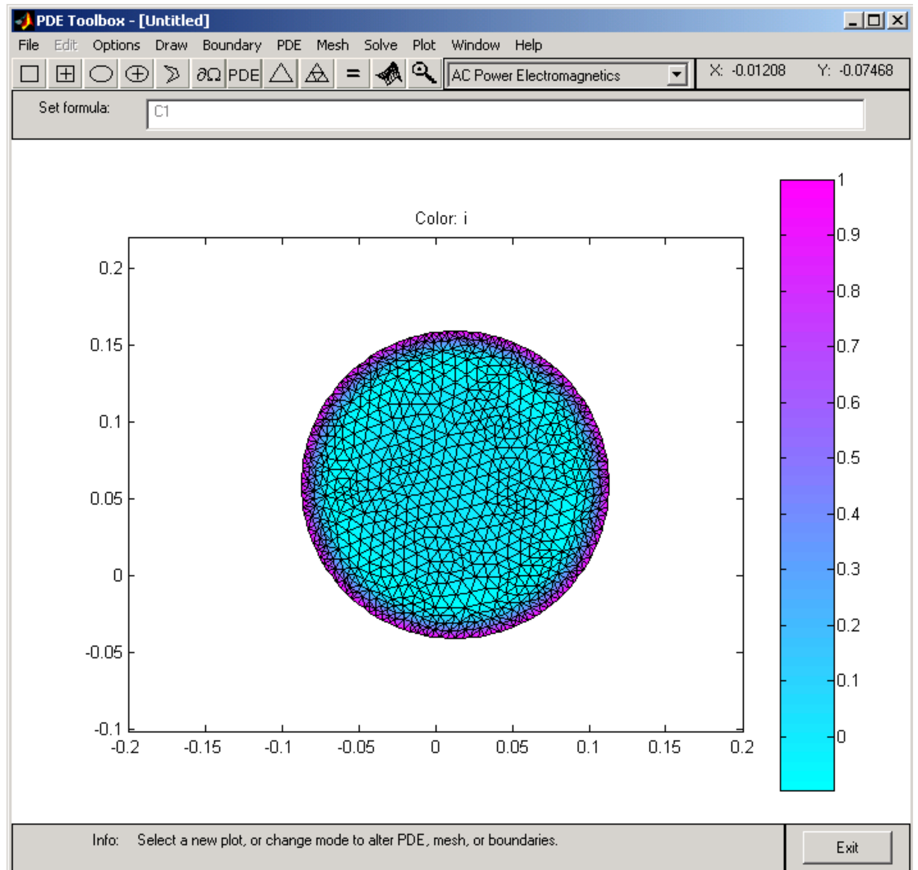
R is the radius of the wire, r is the distance from the center line, and $J_0(x)$ is the first Bessel function of zeroth order.

Using the Graphical User Interface. Start the `pdetool` GUI and set the application mode to**AC Power Electromagnetics**. Draw a circle with radius 0.1 to represent a cross section of the conductor, and proceed to the boundary mode to define the boundary condition. Use the **Select All** option to select all boundaries and enter `1/57E6` into the **r** edit field in the Boundary Condition dialog box to define the Dirichlet boundary condition ($E = J/\sigma$).

Open the PDE Specification dialog box and enter the PDE parameters. The angular frequency $\omega = 2 \cdot \pi \cdot 50$.



Initialize the mesh and solve the equation. Due to the skin effect, the current density at the surface of the conductor is much higher than in the conductor's interior. This is clearly visualized by plotting the current density J as a 3-D plot. To improve the accuracy of the solution close to the surface, you need to refine the mesh. Open the Solve Parameters dialog box and select the **Adaptive mode** check box. Also, set the maximum numbers of triangles to `Inf`, the maximum numbers of refinements to 1, and use the triangle selection method that picks the worst triangles. Recompute the solution several times. Each time the adaptive solver refines the area with the largest errors. The number of triangles is printed on the command line. The following mesh is the result of successive adaptations and contains 1548 triangles.

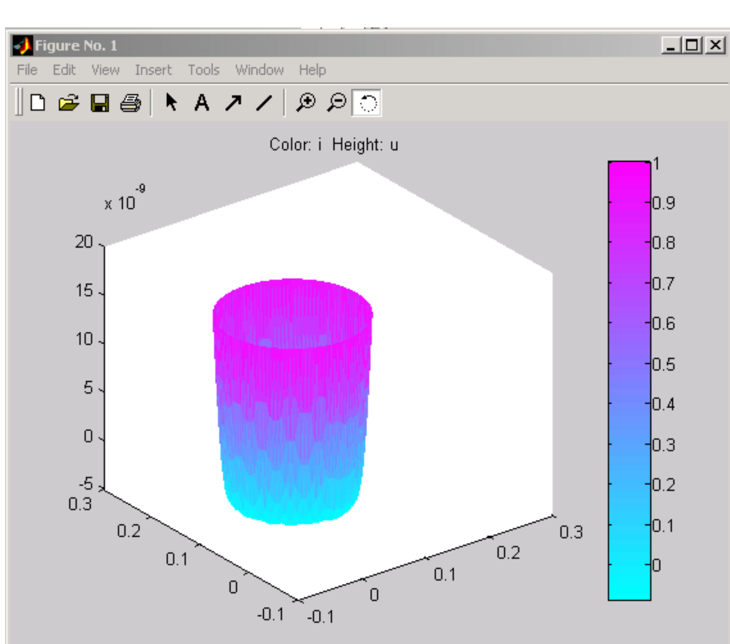


The Adaptively Refined Mesh

The solution of the AC power electromagnetics equation is complex. The plots show the real part of the solution (a warning message is issued), but the solution vector, which can be exported to the main workspace, is the full complex solution. Also, you can plot various properties of the complex solution by using the user entry. `imag(u)` and `abs(u)` are two examples of valid user entries.

The skin effect is an AC phenomenon. Decreasing the frequency of the alternating current results in a decrease of the skin effect. Approaching DC

conditions, the current density is close to uniform (experiment using different angular frequencies).



The Current Density in an AC Wire

Conductive Media DC

For electrolysis and computation of resistances of grounding plates, we have a conductive medium with conductivity σ and a steady current. The current density **J** is related to the electric field **E** through **J** = σ**E**. Combining the continuity equation $\nabla \cdot \mathbf{J} = Q$, where Q is a current source, with the definition of the electric potential V yields the elliptic Poisson's equation

$$-\nabla \cdot (\sigma \nabla V) = Q$$

The only two PDE parameters are the conductivity σ and the current source Q.

The Dirichlet boundary condition assigns values of the electric potential V to the boundaries, usually metallic conductors. The Neumann boundary condition requires the value of the normal component of the current density

($\mathbf{n} \cdot (\sigma \nabla(V))$) to be known. It is also possible to specify a generalized Neumann condition defined by $\mathbf{n} \cdot (\sigma \nabla V) + qV = g$, where q can be interpreted as a *film conductance* for thin plates.

The electric potential V, the electric field $\mathbf{E}$, and the current density $\mathbf{J}$ are all available for plotting. Interesting quantities to visualize are the current lines (the vector field of $\mathbf{J}$) and the equipotential lines of V. The equipotential lines are orthogonal to the current lines when σ is isotropic.

Example. Two circular metallic conductors are placed on a plane, thin conductor like a blotting paper wetted by brine. The equipotentials can be traced by a voltmeter with a simple probe, and the current lines can be traced by strongly colored ions, such as permanganate ions.

The physical model for this problem consists of the Laplace equation

$$-\nabla \cdot (\sigma \nabla V) = 0$$

for the electric potential V and the boundary conditions:

- $V = 1$ on the left circular conductor
- $V = -1$ on the right circular conductor
- The natural Neumann boundary condition on the outer boundaries

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

The conductivity σ = 1 (constant).

1 Open the Partial Differential Equation Toolbox GUI by typing

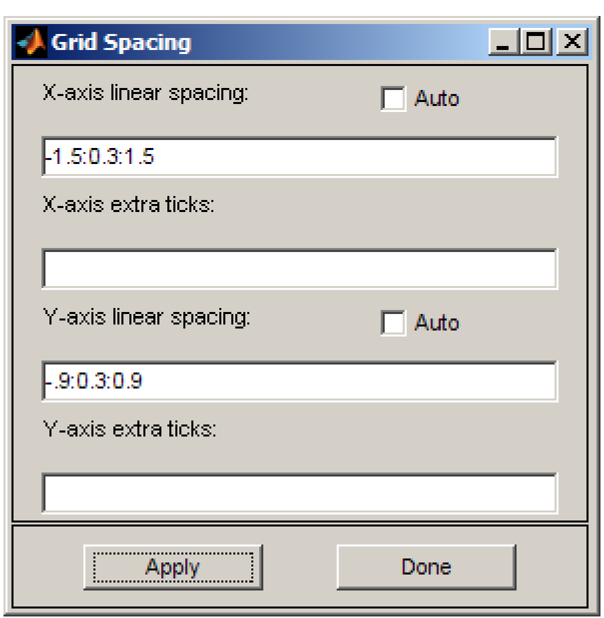
```
pdetool
```

at the MATLAB command prompt.

2 Click **Options > Application > Conductive Media DC**.

3 Click **Options > Grid Spacing...**, deselect the **Auto** check boxes for **X-axis linear spacing** and **Y-axis linear spacing**, and choose a spacing

of 0.3, as pictured. Ensure the Y-axis goes from –0.9 to 0.9. Click **Apply**, and then **Done**.



4 Click **Options > Snap**

5 Click [rectangle button] and draw the blotting paper as a rectangle with corners in (-1.2,-0.6), (1.2,-0.6), (1.2,0.6), and (-1.2,0.6).

6 Click [ellipse button] and add two circles with radius 0.3 that represent the circular conductors. Place them symmetrically with centers in (-0.6,0) and (0.6,0).

7 To express the 2-D domain of the problem, enter

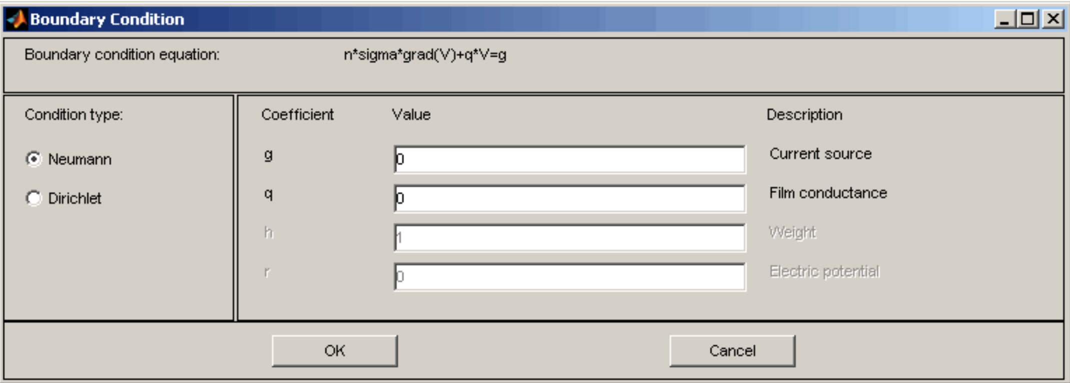
```
R1-(C1+C2)
```

for the **Set formula** parameter.

8 To decompose the geometry and enter the boundary mode, click ∂Ω.

9 Hold down **Shift** and click to select the outer boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

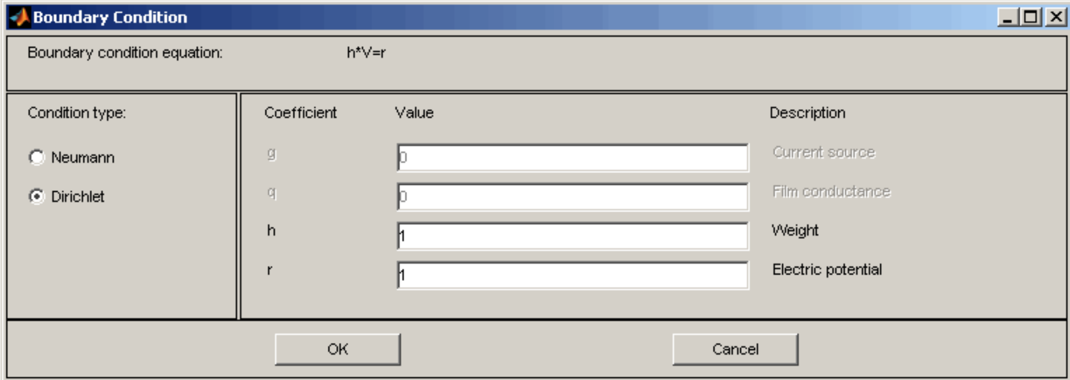
10 Select **Neumann** and click **OK**.



11 Hold down **Shift** and click to select the left circular conductor boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

12 Set the parameters as follows and then click **OK**:

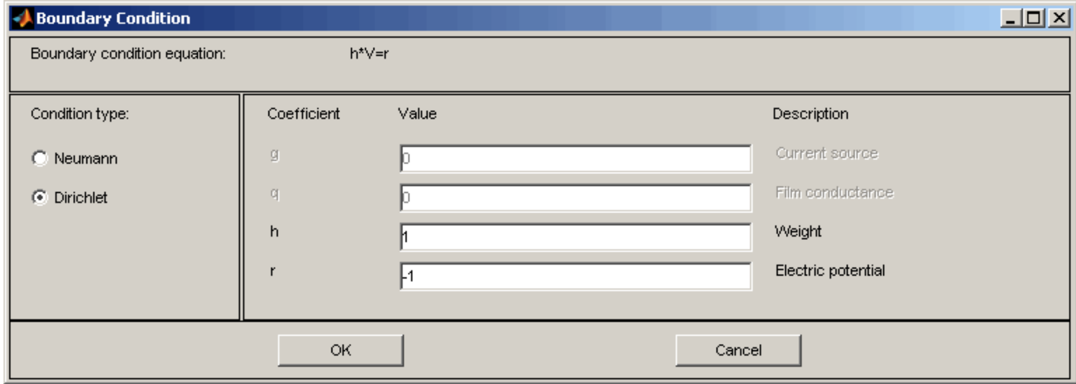
- **Condition type** = **Dirichlet**
- **h** = 1
- **r** = 1



13 Hold down **Shift** and click to select the right circular conductor boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

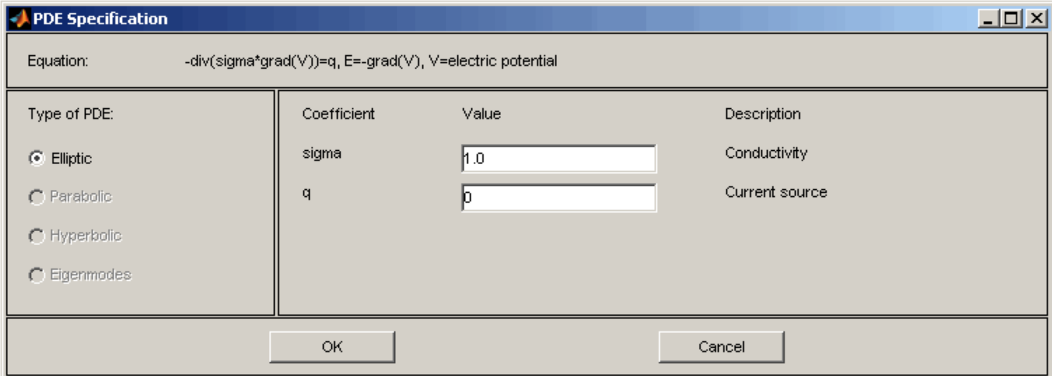
14 Set the parameters as follows and then click **OK**:

- **Condition type = Dirichlet**
- **h** = 1
- **r** = -1



15 Open the PDE Specification dialog box by clicking **PDE > PDE Specification**.

16 Set the current source, **q**, parameter to 0.



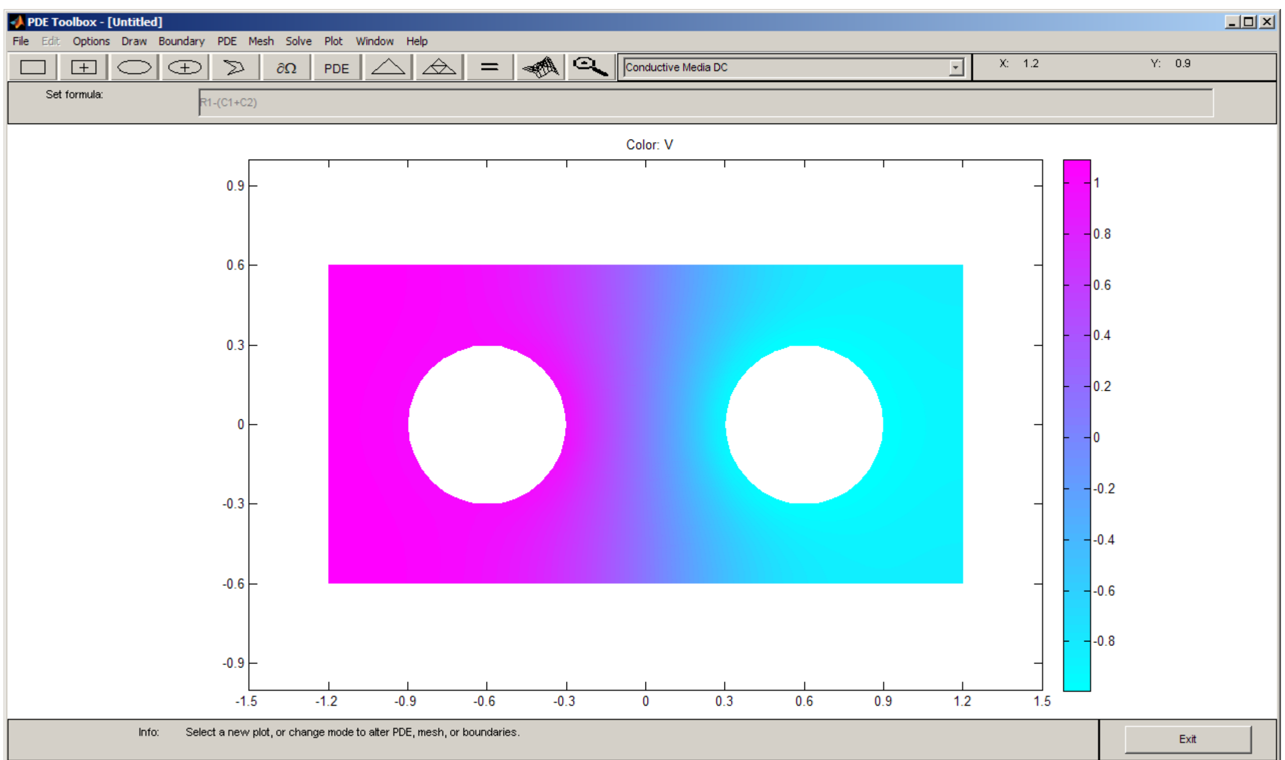
17 Initialize the mesh by clicking **Mesh > Initialize Mesh**.

18 Refine the mesh by clicking **Mesh > Refine Mesh** twice.

19 Improve the triangle quality by clicking **Mesh > Jiggle Mesh**.

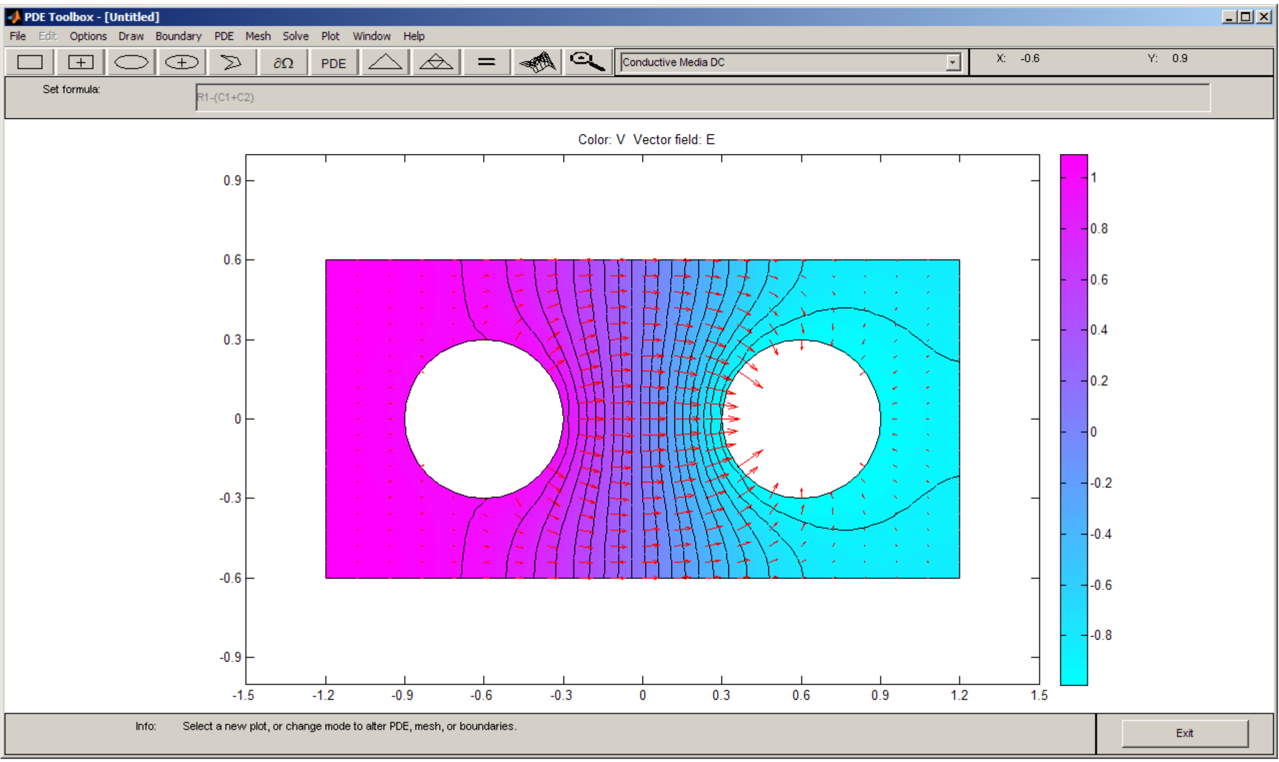
20 Solve the PDE by clicking =.

The resulting potential is zero along the *y*-axis, which is a vertical line of anti-symmetry for this problem.



21 Visualize the current density **J** by clicking **Plot > Parameters**, selecting **Contour** and **Arrows** check box, and clicking **Plot**.

The current flows, as expected, from the conductor with a positive potential to the conductor with a negative potential.



The Current Density Between Two Metallic Conductors

Heat Transfer

The heat equation is a parabolic PDE:

$$\rho C\frac{\partial T}{\partial t} - \nabla \cdot (k\nabla T) = Q + h \cdot (T_{ext} - T)$$

It describes the heat transfer process for plane and axisymmetric cases, and uses the following parameters:

- Density ρ
- Heat capacity C
- Coefficient of heat conduction k

- Heat source Q
- Convective heat transfer coefficient h
- External temperature T_{ext}

The term $h \cdot (T_{ext} - T)$ is a model of transversal heat transfer from the surroundings, and it may be useful for modeling heat transfer in thin cooling plates etc.

For the steady state case, the elliptic version of the heat equation,

$$-\nabla \cdot (k\nabla T) = Q + h \cdot (T_{ext} - T)$$

is also available.

The boundary conditions can be of Dirichlet type, where the temperature on the boundary is specified, or of Neumann type where the *heat flux*, $\mathbf{n} \cdot (k\nabla(T))$, is specified. A generalized Neumann boundary condition can also be used. The generalized Neumann boundary condition equation is $\mathbf{n} \cdot (k\nabla(T)) + qT = g$, where q is the heat transfer coefficient.

Visualization of the temperature, the temperature gradient, and the heat flux $k\nabla T$ is available. Plot options include *isotherms* and heat flux vector field plots.

Example. In the following example, a heat transfer problem with differing material parameters is solved.

The problem's 2-D domain consists of a square with an embedded diamond (a square with 45 degrees rotation). The square region consists of a material with coefficient of heat conduction of 10 and a density of 2. The diamond-shaped region contains a uniform heat source of 4, and it has a coefficient of heat conduction of 2 and a density of 1. Both regions have a heat capacity of 0.1.

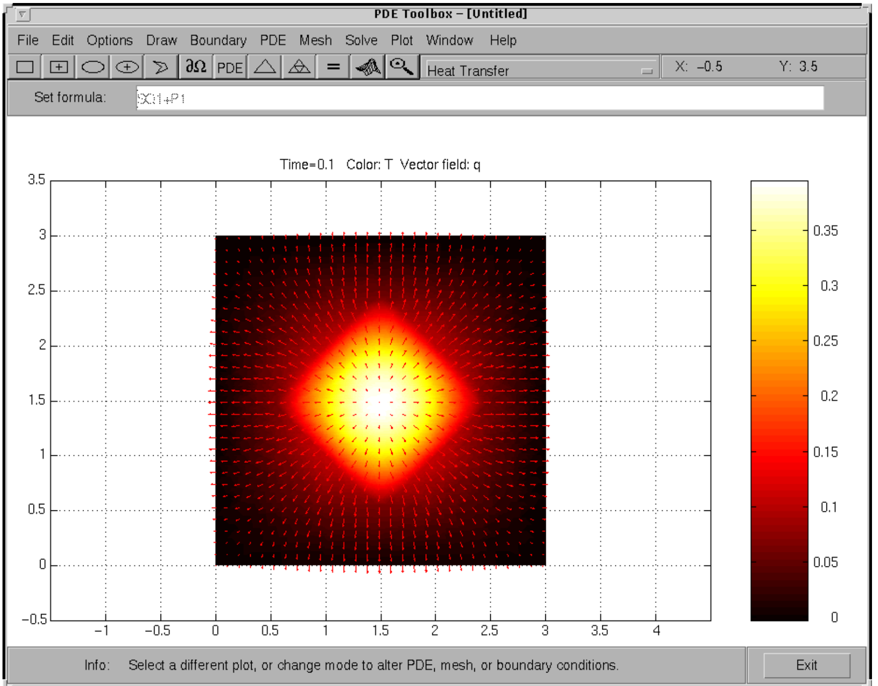
Using the Graphical User Interface. Start the `pdetool` GUI and set the application mode to **Heat Transfer**. In draw mode, set the *x*- and *y*-axis limits to `[-0.5 3.5]` and select the **Axis Equal** option from the **Options** menu. The square region has corners in (0,0), (3,0), (3,3), and (0,3). The diamond-shaped region has corners in (1.5,0.5), (2.5,1.5), (1.5,2.5), and (0.5,1.5).

The temperature is kept at 0 on all the outer boundaries, so you do not have to change the default boundary conditions. Move on to define the PDE parameters (make sure to set the application mode to **Heat Transfer** in the PDE mode by double-clicking each of the two regions and enter the PDE parameters. You want to solve the parabolic heat equation, so make sure that the **Parabolic** option is selected. In the square region, enter a density of 2, a heat capacity of 0.1, and a coefficient of heat conduction of 10. There is no heat source, so set it to 0. In the diamond-shaped region, enter a density of 1, a heat capacity of 0.1, and a coefficient of heat conduction of 2. Enter 4 in the edit field for the heat source. The transversal heat transfer term $h \cdot (T_{ext} - T)$ is not used, so set h, the convective heat transfer coefficient, to 0.

Since you are solving a dynamic PDE, you have to define an initial value, and the times at which you want to solve the PDE. Open the Solve Parameters dialog box by selecting **Parameters** from the **Solve** menu. The dynamics for this problem is very fast—the temperature reaches steady state in about 0.1 time units. To capture the interesting part of the dynamics, enter `logspace(-2,-1,10)` as the vector of times at which to solve the heat equation. `logspace(-2,-1,10)` gives 10 logarithmically spaced numbers between 0.01 and 0.1. Set the initial value of the temperature to 0. If the boundary conditions and the initial value differ, the problem formulation contains discontinuities.

Solve the PDE. By default, the temperature distribution at the last time is plotted. The best way to visualize the dynamic behavior of the temperature is to animate the solution. When animating, turn on the **Height (3-D plot)** option to animate a 3-D plot. Also, select the **Plot in x-y grid** option. Using a rectangular grid instead of a triangular grid speeds up the animation process significantly.

Other interesting visualizations are made by plotting isothermal lines using a contour plot, and by plotting the heat flux vector field using arrows.



Visualization of the Temperature and the Heat Flux

Diffusion

Since heat transfer is a diffusion process, the generic diffusion equation has the same structure as the heat equation:

$$\frac{\partial c}{\partial t} - \nabla \cdot (D \nabla c) = Q$$

where c is the concentration, D is the *diffusion coefficient* and Q is a volume source. If diffusion process is anisotropic, in which case D is a 2-by-2 matrix, you must solve the diffusion equation using the generic system application mode of the `pdetool` GUI. For more information, see "PDE Menu" on page 2-16.

The boundary conditions can be of Dirichlet type, where the concentration on the boundary is specified, or of Neumann type, where the flux, $\mathbf{n} \cdot (D \nabla(c))$, is

specified. It is also possible to specify a generalized Neumann condition. It is defined by $\mathbf{n} \cdot (D\nabla(c)) + qc = g$, where q is a transfer coefficient.

Visualization of the concentration, its gradient, and the flux is available from the Plot Selection dialog box.

References

[1] Cook, Robert D., David S. Malkus, and Michael E. Plesha, *Concepts and Applications of Finite Element Analysis*, 3rd edition, John Wiley & Sons, New York, 1989.

[2] Popovic, Branko D., *Introductory Engineering Electromagnetics*, Addison-Wesley, Reading, MA, 1971.

2

Graphical User Interface

This chapter discusses the graphical user interface (GUI) `pdetool`. The main components of the GUI are the menus, the dialog boxes, and the toolbar.

- "Using the pdetool Menus" on page 2-2
- "The Toolbar" on page 2-35

Using the pdetool Menus

In this section...

"Introduction" on page 2-2
"File Menu" on page 2-3
"Edit Menu" on page 2-5
"Options Menu" on page 2-7
"Draw Menu" on page 2-10
"Boundary Menu" on page 2-12
"PDE Menu" on page 2-16
"Mesh Menu" on page 2-20
"Solve Menu" on page 2-22
"Plot Menu" on page 2-27
"Window Menu" on page 2-34
"Help Menu" on page 2-34

Introduction

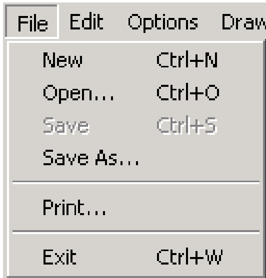
The graphical user interface (GUI) has a pull-down menu bar that you can use to control the modeling. It conforms to common pull-down menu standards.

Menu items followed by a right arrow lead to a submenu. Menu items followed by an ellipsis lead to a dialog box. Stand-alone menu items lead to direct action. Some menu items can be executed by using keyboard accelerators.

`pdetool` also contains a toolbar with icon buttons for quick and easy access to some of the most important `pdetool` functions.

The following sections describe the contents of the `pdetool` menus and the dialog boxes associated with menu items.

File Menu



New	Create a new (empty) *Constructive Solid Geometry* (CSG) model.
Open	Load a model file from disk.
Save	Save the GUI session to a model file.
Save As	Save the GUI session to a new model file.
Print	Print a hardcopy of a figure.
Exit	Exit the `pdetool` graphical user interface.

New

New deletes the current CSG model and creates a new, empty model called "Untitled."

Open

Open displays a dialog box with a list of existing files from which you can select the file that you want to load. You can list the contents of a different folder by changing the path in the **Selection** text box. You can use the scroll bar to display more filenames. You can select a file by double-clicking the filename or by clicking the filename and then clicking the **Done** button. When you select a file, the CSG model that is stored in the model file is loaded into the workspace and displayed. Also, the equation, the boundary conditions, and information about the mesh and the solution are loaded if present, and the modeling and solution process continues to the same status as when you saved the file.

Save As

Save As displays a dialog box in which you can specify the name of the file in which to save the CSG model and other information regarding the GUI session. You can also change the folder in which it is saved. If the filename is given without a `.m` extension, `.m` is appended automatically.

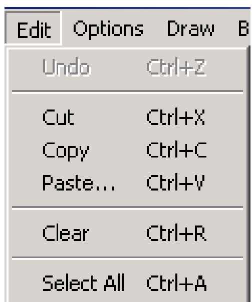
The GUI session is stored in a model file, which contains a sequence of drawing commands and commands to recreate the modeling environment (axes scaling, grid, etc.). If you have already defined boundary conditions, PDE coefficients, created a triangular mesh, and solved the PDE, further commands to recreate the modeling and solution of the PDE problem are also included in the model file. The `pdetool` GUI can be started from the command line by entering the name of a model file. The model in the file is then directly loaded into the GUI.

Print

Print displays a dialog box for printing a hardcopy of a figure. Only the main part of the figure is printed, not the upper and lower menu and information parts. In the dialog box, you can enter any device option that is available for the MATLAB `print` command. The default device option is `-dps` (PostScript®

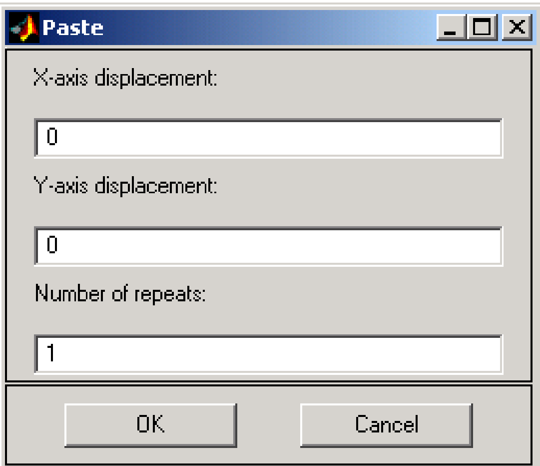
for black and white printers). The paper orientation can be set to `portrait`, `landscape`, or `tall`, and you can print to a printer or to file.

Edit Menu



Undo	Undo the last line when drawing a polygon.
Cut	Move the selected solid objects to the Clipboard.
Copy	Copy the selected objects to the Clipboard, leaving them intact in their original location.
Paste	Copy the contents of the Clipboard to the current CSG model.
Clear	Delete the selected objects.
Select All	Select all solid objects in the current CSG model. Also, select all outer boundaries or select all subdomains.

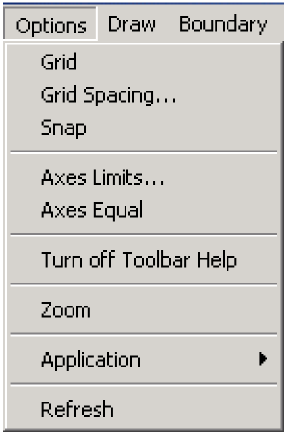
Paste



Paste displays a dialog box for pasting the contents of the Clipboard on to the current CSG model. The Clipboard contents can be repeatedly pasted adding a specified x- and y-axis displacement to the positions of the Clipboard objects.

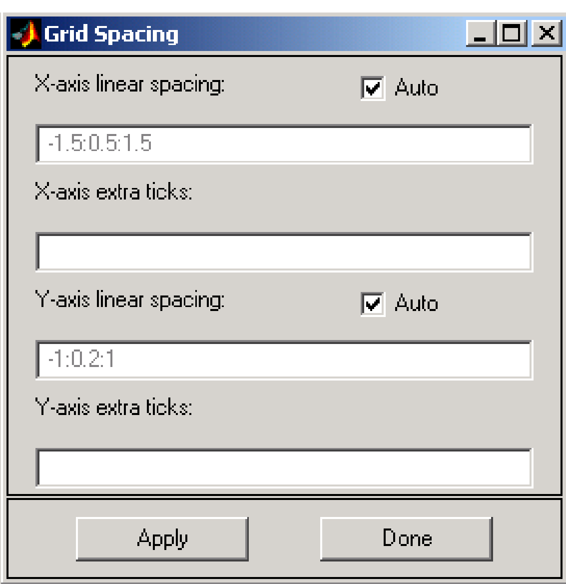
Using the default values—zero displacement and one repetition—the Clipboard contents is inserted at its original position.

Options Menu



Grid	Turn grid on/off.
Grid Spacing	Adjust the grid spacing.
Snap	Turn the “snap-to-grid” feature on/off.
Axis Limits	Change the scaling of the drawing axes.
Axis Equal	Turn the “axis equal” feature on/off.
Turn off Toolbar Help	Turn off help texts for the toolbar buttons.
Zoom	Turn zoom feature on/off.
Application	Select application mode.
Refresh	Redisplay all graphical objects in the `pdetool` graphical user interface.

Grid Spacing



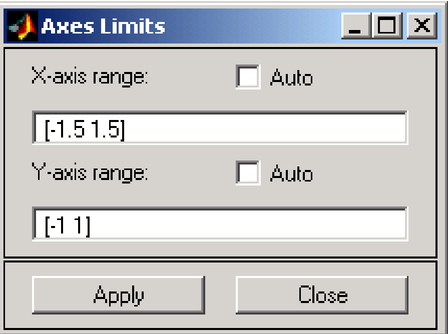
In the Grid Spacing dialog box, you can adjust the *x*-axis and *y*-axis grid spacing. By default, the MATLAB automatic linear grid spacing is used. If you turn off the **Auto** check box, the edit fields for linear spacing and extra ticks are enabled. For example, the default linear spacing `-1.5:0.5:1.5` can be changed to `-1.5:0.2:1.5`. In addition, you can add extra ticks so that the grid can be customized to aid in drawing the desired 2-D domain. Extra tick entries can be separated using spaces, commas, semicolons, or brackets.

Examples:

```
pi
2/3, 0.78, 1.1
-0.123; pi/4
```

Clicking the **Apply** button applies the entered grid spacing; clicking the **Done** button ends the Grid Spacing dialog.

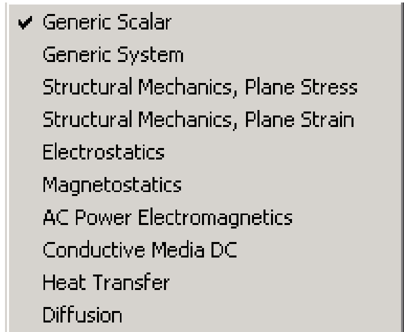
Axes Limits



In the Axes Limits dialog box, the range of the x-axis and the y-axis can be adjusted. The axis range should be entered as a 1-by-2 MATLAB vector such as `[-10 10]`. If you select the **Auto** check box, automatic scaling of the axis is used.

Clicking the **Apply** button applies the entered axis ranges; clicking the **Close** button ends the Axes Limits dialog.

Application



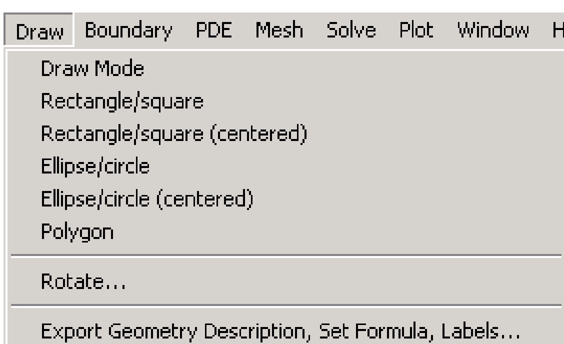
From the **Application** submenu, you can select from 10 available application modes. The application modes can also be selected using the pop-up menu in the upper right corner of the GUI.

The available application modes are:

- Generic Scalar (the default mode)
- Generic System
- Structural Mechanics — Plane Stress
- Structural Mechanics — Plane Strain
- Electrostatics
- Magnetostatics
- AC Power Electromagnetics
- Conductive Media DC
- Heat Transfer
- Diffusion

See “Application Modes” on page 1-84 for more details.

Draw Menu

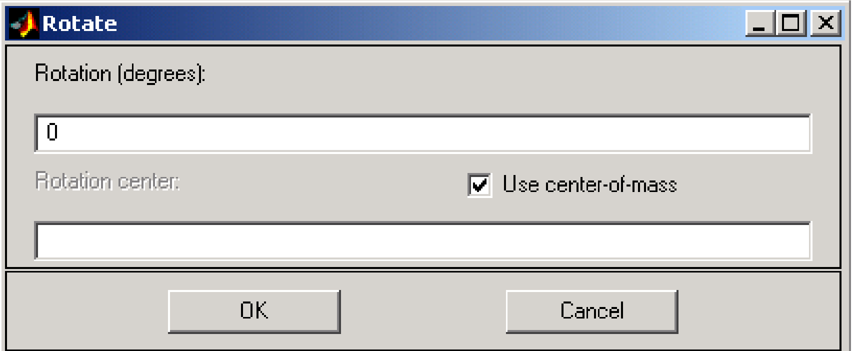


Draw Mode	Enter draw mode.
Rectangle/square	Draw a rectangle/square starting at a corner. Using the left mouse button, click-and-drag to create a rectangle. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a square.

Rectangle/square (centered)	Draw a rectangle/square starting at the center. Using the left mouse button, click-and-drag to create a rectangle. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a square.
Ellipse/circle	Draw an ellipse/circle starting at the perimeter. Using the left mouse button, click-and-drag to create an ellipse. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a circle.
Ellipse/circle (centered)	Draw an ellipse/circle starting at the center. Using the left mouse button, click-and-drag to create an ellipse. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a circle.
Polygon	Draw a polygon. You can close the polygon by pressing the right mouse button. Clicking at the starting vertex also closes the polygon.
Rotate	Rotate selected objects.
Export Geometry Description, Set Formula,	Export the Geometry Description matrix `gd`, the set formula string `sf`, and the

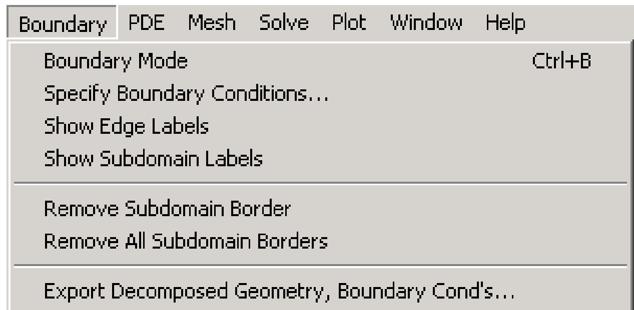
Labels	Name Space matrix ns (labels) to the main workspace.

Rotate



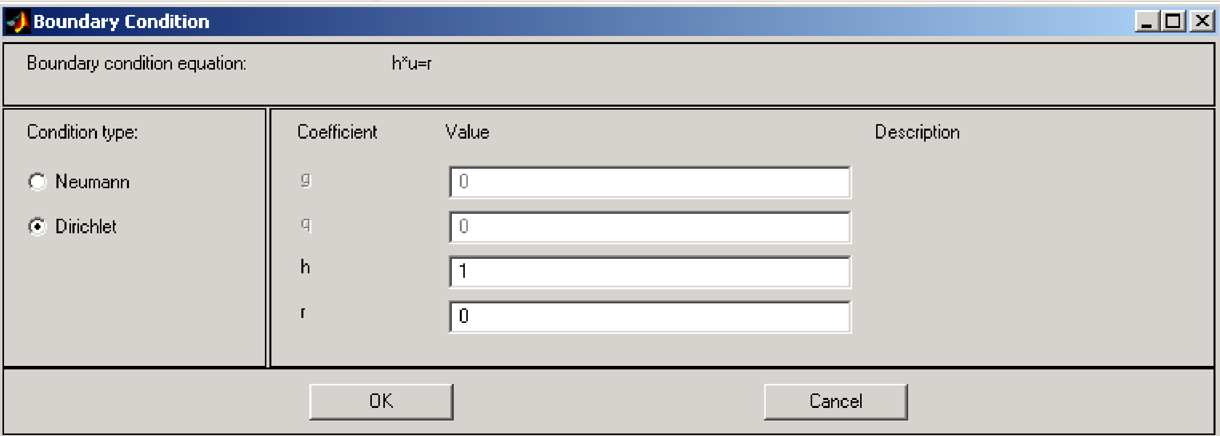
Rotate opens a dialog box where you can enter the angle of rotation in degrees. The selected objects are then rotated by the number of degrees that you specify. The rotation is done counter clockwise for positive rotation angles. By default, the rotation center is the center-of-mass of the selected objects. If the **Use center-of-mass** option is not selected, you can enter a rotation center (*xc*,*yc*) as a 1-by-2 MATLAB vector such as `[-0.4 0.3]`.

Boundary Menu



Boundary Mode	Enter the boundary mode.
Specify Boundary Conditions	Specify boundary conditions for the selected boundaries. If no boundaries are selected, the entered boundary condition applies to all boundaries.
Show Edge Labels	Toggle the labeling of the edges (outer boundaries and subdomain borders) on/off. The edges are labeled using the column number in the Decomposed Geometry matrix.
Show Subdomains Labels	Toggle the labeling of the subdomains on/off. The subdomains are labeled using the subdomain numbering in the Decomposed Geometry matrix.
Remove Subdomain Border	Remove selected subdomain borders.
Remove All Subdomain Borders	Remove all subdomain borders.
Export Decomposed Geometry, Boundary Cond's	Export the Decomposed Geometry matrix `g` and the Boundary Condition matrix `b` to the main workspace.

Specify Boundary Conditions



Specify boundary conditions displays a dialog box in which you can specify the boundary condition for the selected boundary segments. There are three different condition types:

- Generalized Neumann conditions, where the boundary condition is determined by the coefficients q and g according to the following equation:

  $$\vec{n} \cdot (c\nabla u) + qu = g$$

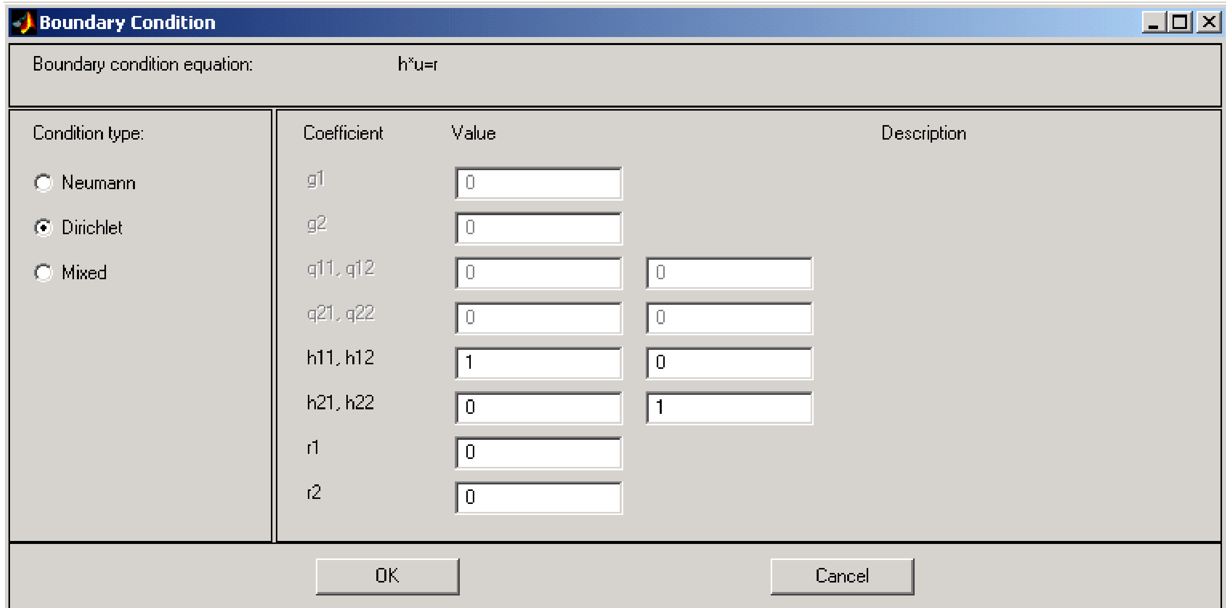
  In the system cases, q is a 2-by-2 matrix and g is a 2-by-1 vector.

- Dirichlet conditions: u is specified on the boundary. The boundary condition equation is $hu = r$, where h is a weight factor that can be applied (normally 1).

  In the system cases, h is a 2-by-2 matrix and r is a 2-by-1 vector.

- Mixed boundary conditions (system cases only), which is a mix of Dirichlet and Neumann conditions. q is a 2-by-2 matrix, g is a 2-by-1 vector, h is a 1-by-2 vector, and r is a scalar.

The following figure shows the boundary condition dialog box for the generic system PDE.



For boundary condition entries you can use the following variables in a valid MATLAB expression:

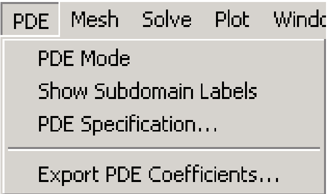
- The 2-D coordinates `x` and `y`.
- A boundary segment parameter `s`, proportional to arc length. `s` is 0 at the start of the boundary segment and increases to 1 along the boundary segment in the direction indicated by the arrow.
- The outward normal vector components `nx` and `ny`. If you need the tangential vector, it can be expressed using `nx` and `ny` since $t_x = -n_y$ and $t_y = n_x$.
- The solution `u`.
- The time `t`.

Note If the boundary condition is a function of the solution `u`, you must use the nonlinear solver. If the boundary condition is a function of the time t, you must choose a parabolic or hyperbolic PDE.

Examples: `(100-80*s).*nx`, and `cos(x.^2)`

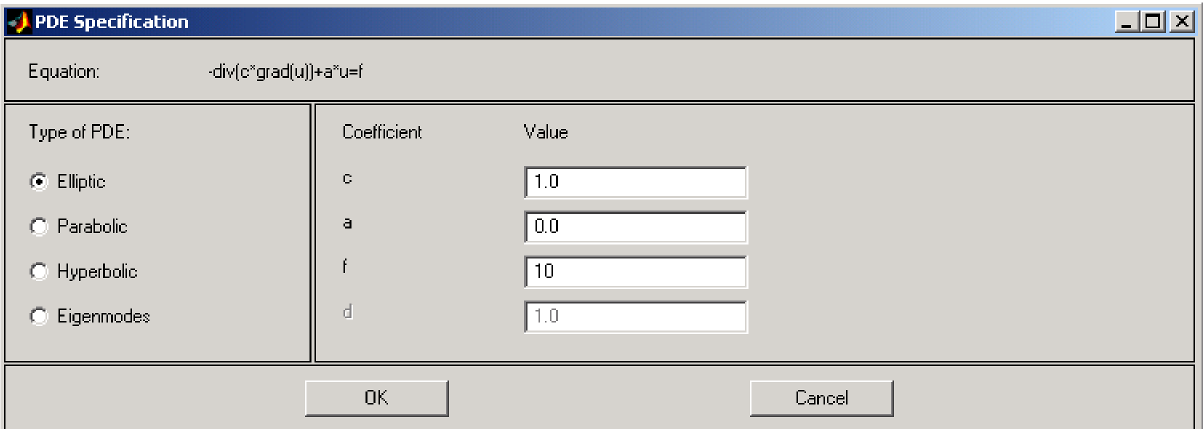
In the nongeneric application modes, the **Description** column contains descriptions of the physical interpretation of the boundary condition parameters.

PDE Menu



PDE Mode	Enter the partial differential equation mode.
Show Subdomain Labels	Toggle the labeling of the subdomains on/off. The subdomains are labeled using the subdomain numbering in the decomposed geometry matrix.
PDE Specification	Open dialog box for entering PDE coefficients and types.
Export PDE Coefficients	Export current PDE coefficients to the main workspace. The resulting workspace variables are strings.

PDE Specification



PDE Specification opens a dialog box where you enter the type of partial differential equation and the applicable parameters. The dimension of the parameters is dependent on the dimension of the PDE. The following description applies to scalar PDEs. If a nongeneric application mode is selected, application-specific PDEs and parameters replace the standard PDE coefficients. For a thorough description of the different application modes, see "Application Modes" on page 1-84.

Each of the coefficients `c`, `a`, `f`, and `d` can be given as a valid MATLAB expression for computing coefficient values at the triangle centers of mass. The following variables are available:

- `x` and `y`: The *x*- and *y*-coordinates
- `u`: The solution
- `ux, uy`: The *x* and *y* derivatives of the solution
- `t`: The time

Note If the PDE coefficient is a function of the solution `u` or its derivatives `ux` and `uy`, you must use the nonlinear solver. If the PDE coefficient is a function of the time `t`, you must choose a parabolic or hyperbolic PDE.

You can also enter the name of a user-defined MATLAB function that accepts the arguments (`p,t,u,time`). For an example, type the function `circlef`.

c can be a scalar or a 2-by-2 matrix. The matrix `c` can be used to model, e.g., problems with anisotropic material properties.

If `c` contains two rows, they are the $c_{1,1}$ and $c_{2,2}$ elements of a 2-by-2 symmetric matrix

$$\begin{pmatrix} c_{1,1} & 0 \\ 0 & c_{2,2} \end{pmatrix}$$

If c contains three rows, they are the $c_{1,1}$, $c_{1,2}$, and $c_{2,2}$ elements of a 2-by-2 symmetric matrix ($c_{2,1} = c_{1,2}$)

$$\begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix}$$

If `c` contains four rows, they are the $c_{1,1}$, $c_{2,1}$, $c_{1,2}$, and $c_{2,2}$ elements of the 2-by-2 preceding matrix.

The available types of PDEs are

- Elliptic. The basic form of the elliptic PDE is

  $$-\nabla \cdot (c\nabla u) + au = f$$

  The parameter `d` does not apply to the elliptic PDE.
- Parabolic. The basic form of the parabolic PDE is

  $$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f,$$

  with initial values `u0` = $u(t_0)$.
- Hyperbolic. The basic form of the hyperbolic PDE is

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f,$$

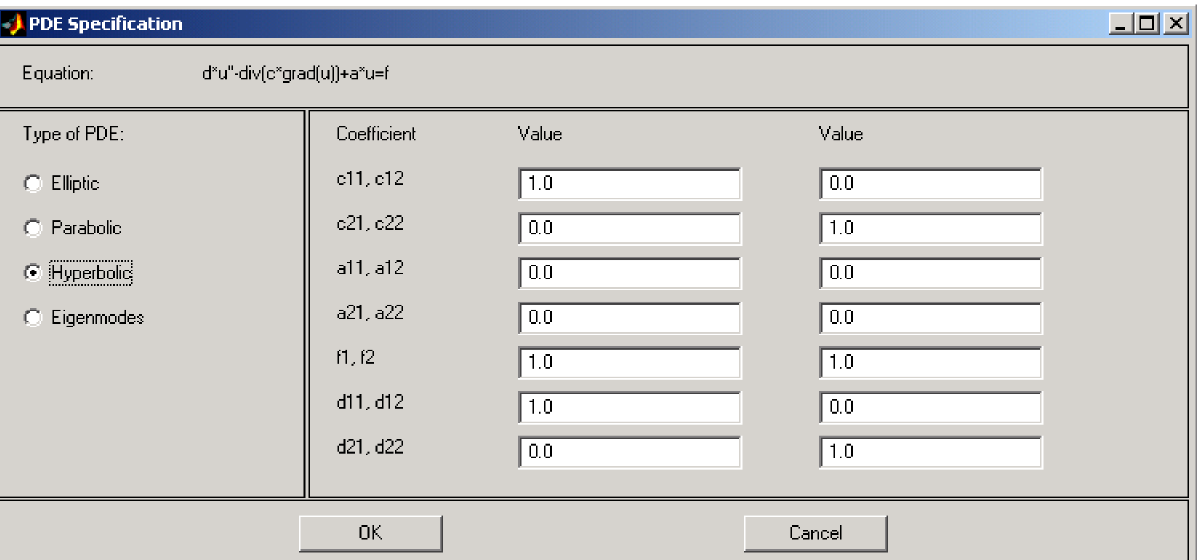
with initial values `u0` $= u(t_0)$ and `ut0` $= \frac{\partial u}{\partial t}(t_0)$

- Eigenmodes. The basic form of the PDE eigenvalue problem is

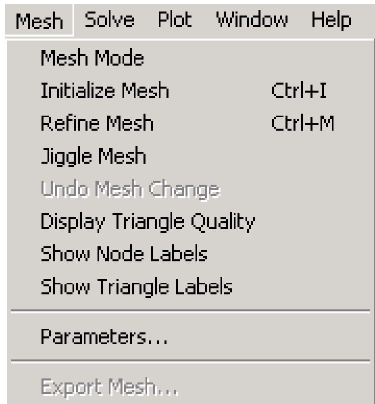
$$-\nabla \cdot (c\nabla u) + au = \lambda du$$

The parameter `f` does not apply to the eigenvalue PDE.

In the system case, `c` is a rank four tensor, which can be represented by four 2-by-2 matrices, `c11`, `c12`, `c21`, and `c22`. They can be entered as one, two, three, or four rows—see the preceding scalar case. `a` and `d` are 2-by-2 matrices, and `f` is a 2-by-1 vector. The PDE Specification dialog box for the system case is shown in the following figure.



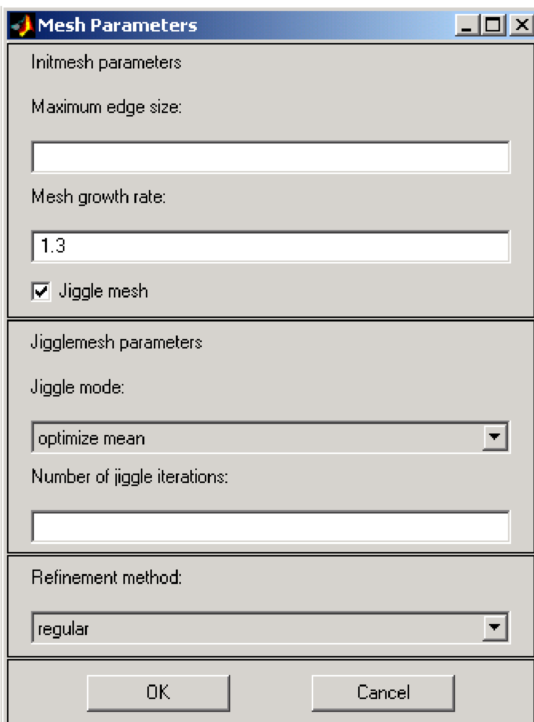
Mesh Menu



Mesh Mode	Enter mesh mode.
Initialize Mesh	Build and display an initial triangular mesh.
Refine Mesh	Uniformly refine the current triangular mesh.
Jiggle Mesh	Jiggle the mesh.
Undo Mesh Change	Undo the last mesh change. All mesh generations are saved, so repeated **Undo Mesh Change** eventually brings you back to the initial mesh.
Display Triangle Quality	Display a plot of the triangular mesh where the individual triangles are colored according to their quality. The quality measure is a number between 0 and 1, where triangles with a quality measure greater than 0.6 are acceptable. For details on the triangle quality measure, see `pdetriq`.
Show Node Labels	Toggle the mesh node labels on/off. The node labels are the column numbers in the Point matrix `p`.
Show Triangle Labels	Toggle the mesh triangle labels on/off. The triangle labels are the column numbers in the triangle matrix `t`.

Parameters	Open dialog box for modification of mesh generation parameters.
Export Mesh	Export Point matrix `p`, Edge matrix `e`, and Triangle matrix `t` to the main workspace.

Parameters



Parameters opens a dialog box containing mesh generation parameters. The parameters used by the mesh initialization algorithm `initmesh` are:

- **Maximum edge size:** Largest triangle edge length (approximately). This parameter is optional and must be a real positive number.
- **Mesh growth rate:** The rate at which the mesh size increases away from small parts of the geometry. The value must be between 1 and 2. The default value is 1.3, i.e., the mesh size increases by 30%.

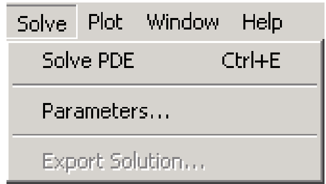
- **Jiggle mesh:** Toggles automatic jiggling of the initial mesh on/off.

The parameters used by the mesh jiggling algorithm `jigglemesh` are:

- **Jiggle mode:** Select a jiggle mode from a pop-up menu. Available modes are `on`, `optimize minimum`, and `optimize mean`. `on` jiggles the mesh once. Using the jiggle mode `optimize minimum`, the jiggling process is repeated until the minimum triangle quality stops increasing or until the iteration limit is reached. The same applies for the `optimize mean` option, but it tries to increase the mean triangle quality.
- **Number of jiggle iterations:** Iteration limit for the `optimize minimum` and `optimize mean` modes. Default: 20.

Finally, for the mesh refinement algorithm `refinemesh`, the **Refinement method** can be `regular` or `longest`. The default refinement method is `regular`, which results in a uniform mesh. The refinement method `longest` always refines the longest edge on each triangle.

Solve Menu



Solve PDE	Solve the partial differential equation for the current CSG model and triangular mesh, and plot the solution (the automatic solution plot can be disabled).
Parameters	Open dialog box for entry of PDE solve parameters.
Export Solution	Export the PDE solution vector `u` and, if applicable, the computed eigenvalues `l` to the main workspace.

Parameters

Solve Parameters Dialog Box for Elliptic PDEs

Parameters opens a dialog box where you can enter the solve parameters. The set of solve parameters differs depending on the type of PDE.

- Elliptic PDEs. By default, no specific solve parameters are used, and the elliptic PDEs are solved using the basic elliptic solver `assempde`. Optionally, the adaptive mesh generator and solver `adaptmesh` can be used. For the adaptive mode, the following parameters are available:
  - **Adaptive mode.** Toggle the adaptive mode on/off.
  - **Maximum number of triangles.** The maximum number of new triangles allowed (can be set to `Inf`). A default value is calculated based on the current mesh.

- **Maximum number of refinements.** The maximum number of successive refinements attempted.
- **Triangle selection method.** There are two triangle selection methods, described below. You can also supply your own function.
  - **Worst triangles.** This method picks all triangles that are worse than a fraction of the value of the worst triangle (default: 0.5). For more details, see `pdetriq`.
  - **Relative tolerance.** This method picks triangles using a relative tolerance criterion (default: 1E-3). For more details, see `pdeadgsc`.
  - **User-defined function.** Enter the name of a user-defined triangle selection method. See `pdedemo7` for an example of a user-defined triangle selection method.
- **Function parameter.** The function parameter allows fine-tuning of the triangle selection methods. For the worst triangle method (`pdeadworst`), it is the fraction of the worst value that is used to determine which triangles to refine. For the relative tolerance method, it is a tolerance parameter that controls how well the solution fits the PDE.
- **Refinement method.** Can be `regular` or `longest`. See the Parameters dialog box description in "Mesh Menu" on page 2-20.

If the problem is nonlinear, i.e., parameters in the PDE are directly dependent on the solution `u`, a nonlinear solver must be used. The following parameters are used:

- **Use nonlinear solver.** Toggle the nonlinear solver on/off.
- **Nonlinear tolerance.** Tolerance parameter for the nonlinear solver.
- **Initial solution.** An initial guess. Can be a constant or a function of x and y given as a MATLAB expression that can be evaluated on the nodes of the current mesh.

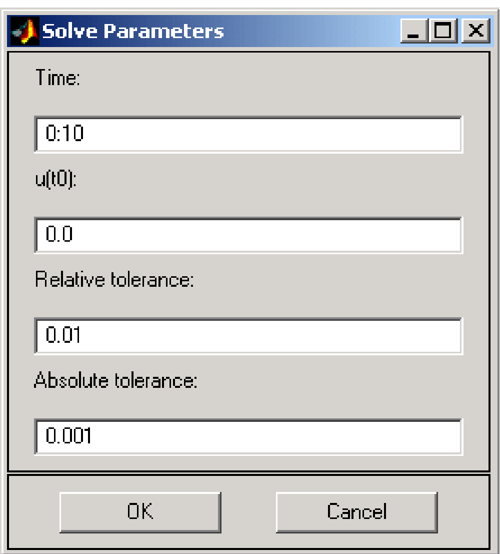
  Examples: `1`, and `exp(x.*y)`. Optional parameter, defaults to zero.
- **Jacobian.** Jacobian approximation method: `fixed` (the default), a fixed point iteration, `lumped`, a "lumped" (diagonal) approximation, or `full`, the full Jacobian.

- **Norm.** The type of norm used for computing the residual. Enter as `energy` for an energy norm, or as a real scalar p to give the lp norm. The default is `Inf`, the infinity (maximum) norm.

Note The adaptive mode and the nonlinear solver can be used together.

- Parabolic PDEs. The solve parameters for the parabolic PDEs are:
  - **Time.** A MATLAB vector of times at which a solution to the parabolic PDE should be generated. The relevant time span is dependent on the dynamics of the problem.

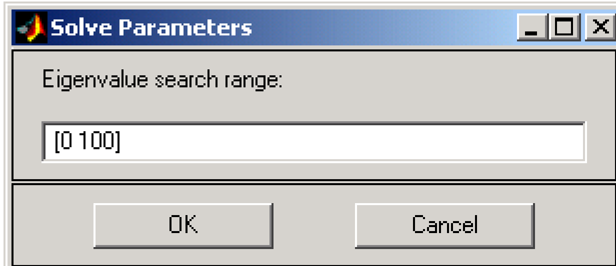
    Examples: `0:10`, and `logspace(-2,0,20)`
  - **u(t0).** The initial value $u(t_0)$ for the parabolic PDE problem The initial value can be a constant or a column vector of values on the nodes of the current mesh.
  - **Relative tolerance.** Relative tolerance parameter for the ODE solver that is used for solving the time-dependent part of the parabolic PDE problem.
  - **Absolute tolerance.** Absolute tolerance parameter for the ODE solver that is used for solving the time-dependent part of the parabolic PDE problem.



Solve Parameters Dialog Box for Hyperbolic PDEs

- Hyperbolic PDEs. The solve parameters for the hyperbolic PDEs are:
  - **Time.** A MATLAB vector of times at which a solution to the hyperbolic PDE should be generated. The relevant time span is dependent on the dynamics of the problem.

    Examples: `0:10`, and `logspace(-2,0,20)`
  - **u(t0).** The initial value $u(t_0)$ for the hyperbolic PDE problem. The initial value can be a constant or a column vector of values on the nodes of the current mesh.
  - **u'(t0).** The initial value $\dot{u}(t_0)$ for the hyperbolic PDE problem. You can use the same formats as for **u(t0)**.
  - **Relative tolerance.** Relative tolerance parameter for the ODE solver that is used for solving the time-dependent part of the hyperbolic PDE problem.
  - **Absolute tolerance.** Absolute tolerance parameter for the ODE solver that is used for solving the time-dependent part of the hyperbolic PDE problem.



Solve Parameters Dialog Box for Eigenvalue PDEs

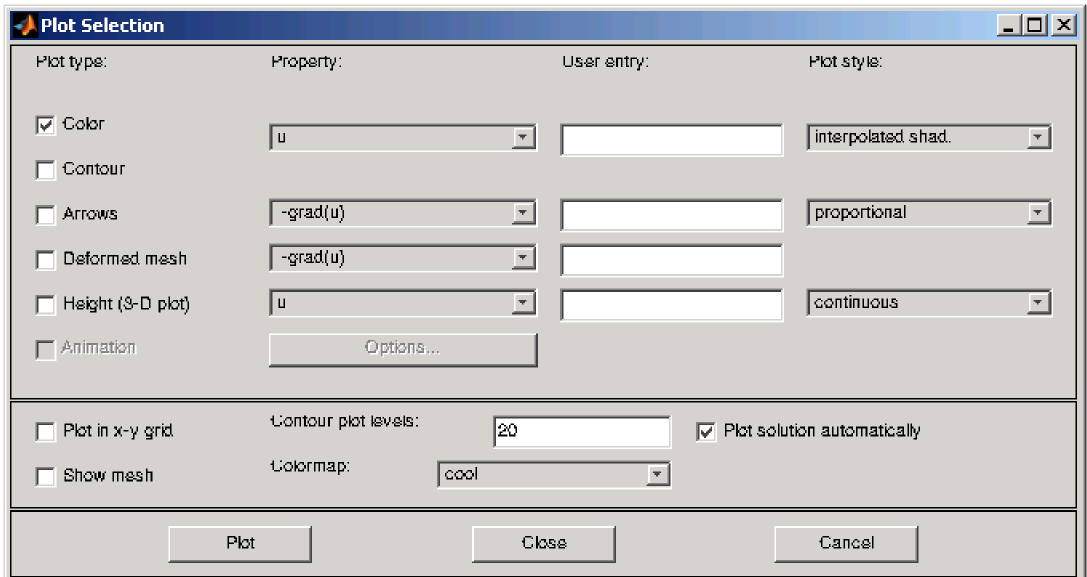
- Eigenvalue problems. For the eigenvalue PDE, the only solve parameter is the **Eigenvalue search range**, a two-element vector, defining an interval on the real axis as a search range for the eigenvalues. The left side can be `-Inf`.

  Examples: `[0 100]`, `[-Inf 50]`

Plot Menu

Plot Solution	Display a plot of the solution.
Parameters	Open dialog box for plot selection.
Export Movie	If a movie has been recorded, the movie matrix `M` is exported to the main workspace.

Parameters



Plot Selection Dialog Box

Parameters opens a dialog box containing options controlling the plotting and visualization.

The upper part of the dialog box contains four columns:

- **Plot type** (far left) contains a row of six different plot types, which can be used for visualization:
  - **Color.** Visualization of a scalar property using colored surface objects.
  - **Contour.** Visualization of a scalar property using colored contour lines. The contour lines can also enhance the color visualization when both plot types (**Color** and **Contour**) are checked. The contour lines are then drawn in black.
  - **Arrows.** Visualization of a vector property using arrows.
  - **Deformed mesh.** Visualization of a vector property by deforming the mesh using the vector property. The deformation is automatically scaled to 10% of the problem domain. This plot type is primarily intended for

visualizing x- and y-displacements (u and v) for problems in structural mechanics. If no other plot type is selected, the deformed triangular mesh is displayed.

- **Height (3-D plot).** Visualization of a scalar property using height (z-axis) in a 3-D plot. 3-D plots are plotted in separate figure windows. If the **Color** and **Contour** plot types are not used, the 3-D plot is simply a mesh plot. You can visualize another scalar property simultaneously using **Color** and/or **Contour**, which results in a 3-D surface or contour plot.
- **Animation.** Animation of time-dependent solutions to parabolic and hyperbolic problems. If you select this option, the solution is recorded and then animated in a separate figure window using the MATLAB `movie` function.

A color bar is added to the plots to map the colors in the plot to the magnitude of the property that is represented using color or contour lines.

- **Property** contains four pop-up menus containing lists of properties that are available for plotting using the corresponding plot type. From the first pop-up menu you control the property that is visualized using color and/or contour lines. The second and third pop-up menus contain vector valued properties for visualization using arrows and deformed mesh, respectively. From the fourth pop-up menu, finally, you control which scalar property to visualize using z-height in a 3-D plot. The lists of properties are dependent on the current application mode. For the generic scalar mode, you can select the following scalar properties:
  - **u.** The solution itself.
  - **abs(grad(u)).** The absolute value of ∇u, evaluated at the center of each triangle.
  - **abs(c*grad(u)).** The absolute value of $c \cdot \nabla u$, evaluated at the center of each triangle.
  - **user entry.** A MATLAB expression returning a vector of data defined on the nodes or the triangles of the current triangular mesh. The solution `u`, its derivatives `ux` and `uy`, the x and y components of $c \cdot \nabla u$, `cux` and `cuy`, and `x` and `y` are all available in the local workspace. You enter the expression into the edit box to the right of the **Property** pop-up menu in the **User entry** column.

Examples: `u.*u, x+y`

The vector property pop-up menus contain the following properties in the generic scalar case:

- **-grad(u)**. The negative gradient of u, $-\nabla u$.
- **-c*grad(u)**. c times the negative gradient of u, $-c \cdot \nabla u$.
- **user entry**. A MATLAB expression `[px; py]` returning a 2-by-*ntri* matrix of data defined on the triangles of the current triangular mesh (*ntri* is the number of triangles in the current mesh). The solution `u`, its derivatives `ux` and `uy`, the x and y components of $c \cdot \nabla u$, `cux` and `cuy`, and `x` and `y` are all available in the local workspace. Data defined on the nodes is interpolated to triangle centers. You enter the expression into the edit field to the right of the **Property** pop-up menu in the **User entry** column.

  Examples: `[ux;uy], [x;y]`

For the generic system case, the properties available for visualization using color, contour lines, or z-height are **u, v, abs(u,v)**, and a user entry. For visualization using arrows or a deformed mesh, you can choose **(u,v)** or a user entry. For applications in structural mechanics, u and v are the x- and y-displacements, respectively.

For the visualization options in the other application modes, see "Application Modes" on page 1-84. The variables available in the local workspace for a user entered expression are the same for all scalar and system modes (the solution is always referred to as u and, in the system case, `v`).

- **User entry** contains four edit fields where you can enter your own expression, if you select the user entry property from the corresponding pop-up menu to the left of the edit fields. If the user entry property is not selected, the corresponding edit field is disabled.
- **Plot style** contains three pop-up menus from which you can control the plot style for the color, arrow, and height plot types respectively. The available plot styles for color surface plots are
  - **Interpolated shading.** A surface plot using the selected colormap and interpolated shading, i.e., each triangular area is colored using a linear, interpolated shading (the default).

- **Flat shading.** A surface plot using the selected colormap and flat shading, i.e., each triangular area is colored using a constant color.

You can use two different arrow plot styles:

- **Proportional.** The length of the arrow corresponds to the magnitude of the property that you visualize (the default).
- **Normalized.** The lengths of all arrows are normalized, i.e., all arrows have the same length. This is useful when you are interested in the direction of the vector field. The direction is clearly visible even in areas where the magnitude of the field is very small.

For height (3-D plots), the available plot styles are:

- **Continuous.** Produces a "smooth" continuous plot by interpolating data from triangle midpoints to the mesh nodes (the default).
- **Discontinuous.** Produces a discontinuous plot where data and z-height are constant on each triangle.

A total of three properties of the solution—two scalar properties and one vector field—can be visualized simultaneously. If the **Height (3-D plot)** option is turned off, the solution plot is a 2-D plot and is plotted in the main axes of the `pdetool` GUI. If the **Height (3-D plot)** option is used, the solution plot is a 3-D plot in a separate figure window. If possible, the 3-D plot uses an existing figure window. If you would like to plot in a new figure window, simply type `figure` at the MATLAB command line.

Additional Plot Control Options

In the middle of the dialog box are a number of additional plot control options:

- **Plot in x-y grid.** If you select this option, the solution is converted from the original triangular grid to a rectangular *x-y* grid. This is especially useful for animations since it speeds up the process of recording the movie frames significantly.
- **Show mesh**. In the surface plots, the mesh is plotted using black color if you select this option. By default, the mesh is hidden.
- **Contour plot levels**. For contour plots, the number of level curves, e.g., `15` or `20` can be entered. Alternatively, you can enter a MATLAB vector of

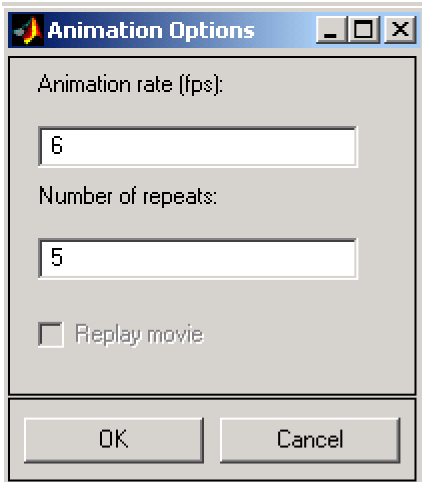
levels. The curves of the contour plot are then drawn at those levels. The default is 20 contour level curves.

Examples: `[0:100:1000]`, `logspace(-1,1,30)`

- **Colormap**. Using the **Colormap** pop-up menu, you can select from a number of different colormaps: `cool`, `gray`, `bone`, `pink`, `copper`, `hot`, `jet`, `hsv`, and `prism`.
- **Plot solution automatically.** This option is normally selected. If turned off, there will *not* be a display of a plot of the solution immediately upon solving the PDE. The new solution, however, can be plotted using this dialog box.

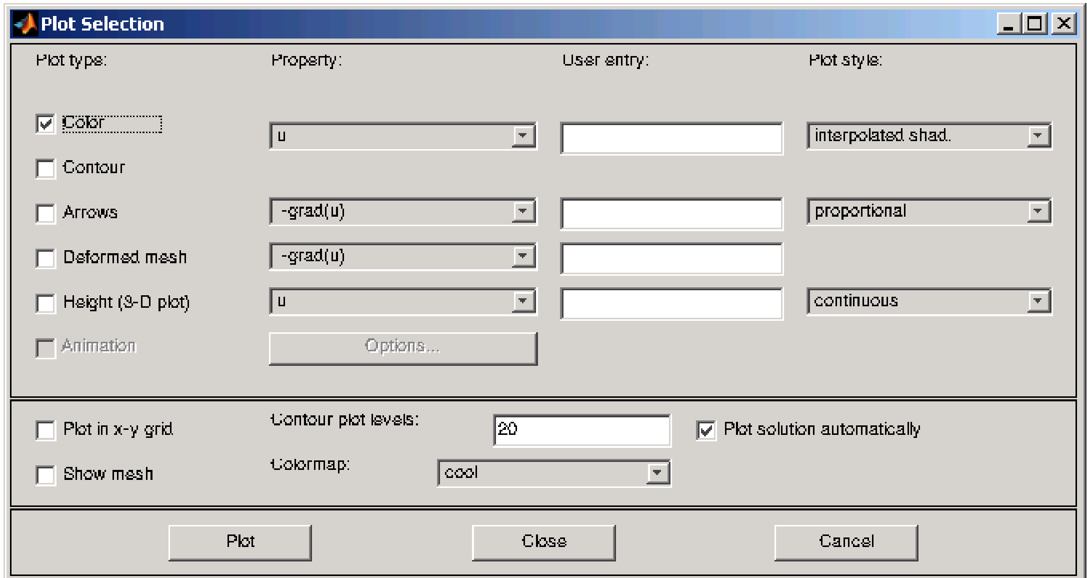
For the parabolic and hyperbolic PDEs, the bottom right portion of the Plot Selection dialog box contains the **Time for plot** parameter.

Time for plot. A pop-up menu allows you to select which of the solutions to plot by selecting the corresponding time. By default, the last solution is plotted.



Also, the **Animation** plot type is enabled. In its property field you find an **Options** button. If you press it, an additional dialog box appears. It contains parameters that control the animation:

- **Animation rate (fps)**. For the animation, this parameter controls the speed of the movie in frames per second (fps).
- **Number of repeats**. The number of times the movie is played.
- **Replay movie**. If you select this option, the current movie is replayed without rerecording the movie frames. If there is no current movie, this option is disabled.



For eigenvalue problems, the bottom right part of the dialog box contains a pop-up menu with all eigenvalues. The plotted solution is the eigenvector associated with the selected eigenvalue. By default, the smallest eigenvalue is selected.

You can rotate the 3-D plots by clicking the plot and, while keeping the mouse button down, moving the mouse. For guidance, a surrounding box appears. When you release the mouse, the plot is redrawn using the new viewpoint. Initially, the solution is plotted using -37.5 degrees horizontal rotation and 30 degrees elevation.

If you click the **Plot** button, the solution is plotted immediately using the current plot setup. If there is no current solution available, the PDE is first solved. The new solution is then plotted. The dialog box remains on the screen.

If you click the **Done** button, the dialog box is closed. The current setup is saved but no additional plotting takes place.

If you click the **Cancel** button, the dialog box is closed. The setup remains unchanged since the last plot.

Window Menu

From the **Window** menu, you can select all currently open MATLAB figure windows. The selected window is brought to the front.

Help Menu

Help	Display a brief help window.
About	Display a window with some program information.

The Toolbar

The toolbar underneath the main menu at the top of the GUI contains icon buttons that provide quick and easy access to some of the most important functions. They do not offer any additional functionality; all toolbar buttons functions are also available using menu items. The toolbar consists of three different parts: the five leftmost buttons for draw mode functions, the next six buttons for different boundary, mesh, solution, and plot functions, and finally the rightmost button for activating the zoom feature.

The draw mode buttons represent, from left to right:

- Draw a rectangle/square starting at a corner. Using the left mouse button, click-and-drag to create a rectangle. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a square.
- Draw a rectangle/square starting at the center. Using the left mouse button, click-and-drag to create a rectangle. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a square.
- Draw an ellipse/circle starting at the perimeter. Using the left mouse button, click-and-drag to create an ellipse. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a circle.
- Draw an ellipse/circle starting at the center. Using the left mouse button, click-and-drag to create an ellipse. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a circle.
- Draw a polygon. Click-and-drag to create polygon edges. You can close the polygon by pressing the right mouse button. Clicking at the starting vertex also closes the polygon.

The draw mode buttons can only be activated one at a time and they all work the same way: single-clicking a button allows you to draw one solid object of the selected type. Double-clicking a button makes it "stick," and you can then continue to draw solid objects of the selected type until you single-click the button to "release" it.

If you are not in the draw mode when you click one of the draw mode buttons, the GUI enters the draw mode automatically.

The second group of buttons includes the following buttons, from left to right:

∂Ω	Enters the boundary mode.
PDE	Opens the PDE Specification dialog box.
	Initializes the triangular mesh
	Refines the triangular mesh.
=	Solves the PDE.
	Opens the Solution Plot Selection dialog box.

The buttons in the second group are of the "flash" type; single-clicking them initiates the associated function.

The last, rightmost button with the magnifier icon toggles the zoom function on/off.

Finite Element Method

The core Partial Differential Equation Toolbox algorithm is a PDE solver that uses the *Finite Element Method* (FEM) for problems defined on bounded domains in the plane. Partial Differential Equation Toolbox software provides a user-friendly interface between the MATLAB computing environment and the FEM technical procedures.

- "The Elliptic Equation" on page 3-2
- "The Elliptic System" on page 3-10
- "The Parabolic Equation" on page 3-13
- "The Hyperbolic Equation" on page 3-18
- "The Eigenvalue Equation" on page 3-19
- "Nonlinear Equations" on page 3-23
- "Adaptive Mesh Refinement" on page 3-29
- "Fast Solution of Poisson's Equation" on page 3-32
- "References" on page 3-34

The Elliptic Equation

The basic elliptic equation handled by the software is

$$-\nabla \cdot (c\nabla u) + au = f \text{ in } \Omega$$

where Ω is a bounded domain in the plane. *c, a, f,* and the unknown solution *u* are complex functions defined on Ω. *c* can also be a 2-by-2 matrix function on Ω. The boundary conditions specify a combination of *u* and its normal derivative on the boundary:

- *Dirichlet:* $hu = r$ on the boundary $\partial\Omega$.
- *Generalized Neumann:* $\vec{n} \cdot (c\nabla u) + qu = g$ on $\partial\Omega$.
- *Mixed:* Only applicable to *systems*. A combination of Dirichlet and generalized Neumann.

$\vec{n}$ is the outward unit normal. *g, q, h,* and *r* are functions defined on $\partial\Omega$.

Our nomenclature deviates slightly from the tradition for potential theory, where a Neumann condition usually refers to the case $q = 0$ and our Neumann would be called a mixed condition. In some contexts, the generalized Neumann boundary conditions is also referred to as the *Robin boundary conditions*. In variational calculus, Dirichlet conditions are also called essential boundary conditions and restrict the trial space. Neumann conditions are also called natural conditions and arise as necessary conditions for a solution. The variational form of the Partial Differential Equation Toolbox equation with Neumann conditions is given below.

The approximate solution to the elliptic PDE is found in three steps:

1. Describe the geometry of the domain Ω and the boundary conditions. This can be done either interactively using `pdetool` (see Chapter 2, “Graphical User Interface”) or through MATLAB files (see `pdegeom` and `pdebound`).

2. Build a triangular mesh on the domain Ω. The software has mesh generating and mesh refining facilities. A mesh is described by three matrices of fixed format that contain information about the mesh points, the boundary segments, and the triangles.

3 Discretize the PDE and the boundary conditions to obtain a linear system $Ku = F$. The unknown vector u contains the values of the approximate solution at the mesh points, the matrix K is assembled from the coefficients c, a, h, and q and the right-hand side F contains, essentially, averages of f around each mesh point and contributions from g. Once the matrices K and F are assembled, you have the entire MATLAB environment at your disposal to solve the linear system and further process the solution.

More elaborate applications make use of the Finite Element Method (FEM) specific information returned by the different functions of the software. Therefore we quickly summarize the theory and technique of FEM solvers to enable advanced applications to make full use of the computed quantities.

FEM can be summarized in the following sentence: *Project the weak form of the differential equation onto a finite-dimensional function space.* The rest of this section deals with explaining the preceding statement.

We start with the *weak form of the differential equation.* Without restricting the generality, we assume generalized Neumann conditions on the whole boundary, since Dirichlet conditions can be approximated by generalized Neumann conditions. In the simple case of a unit matrix h, setting $g = qr$ and then letting $q \rightarrow \infty$ yields the Dirichlet condition because division with a very large q cancels the normal derivative terms. The actual implementation is different, since the preceding procedure may create conditioning problems. The mixed boundary condition of the system case requires a more complicated treatment, described in "The Elliptic System" on page 3-10.

Assume that u is a solution of the differential equation. Multiply the equation with an arbitrary *test function* v and integrate on Ω:

$$\int_{\Omega} -(\nabla \cdot (c\nabla u))v + auv \;\; dx = \int_{\Omega} fv dx$$

Integrate by parts (i.e., use Green's formula) to obtain

$$\int_{\Omega} (c\nabla u) \cdot \nabla v + auvdx - \int_{\partial\Omega} \vec{n} \cdot (c\nabla u)vds = \int_{\Omega} fvdx$$

The boundary integral can be replaced by the boundary condition:

$$\int_\Omega (c\nabla u)\cdot\nabla v + auv\,dx - \int_{\partial\Omega}(-qu+g)v\,ds = \int_\Omega fv\,dx$$

Replace the original problem with *Find u such that*

$$\left(\int_\Omega (c\nabla u)\cdot\nabla v + auv - fv\ \ dx - \int_{\partial\Omega}(-qu+g)v\,ds\right) = 0 \qquad \forall v$$

This equation is called the variational, or weak, form of the differential equation. Obviously, any solution of the differential equation is also a solution of the variational problem. The reverse is true under some restrictions on the domain and on the coefficient functions. The solution of the variational problem is also called the weak solution of the differential equation.

The solution u and the test functions v belong to some function space V. The next step is to choose an Np-dimensional subspace $V_{N_p} \subset V$. *Project the weak form of the differential equation onto a finite-dimensional function space* simply means requesting u and v to lie in V_{N_p} rather than V. The solution of the finite dimensional problem turns out to be the element of V_{N_p} that lies closest to the weak solution when measured in the energy norm. Convergence is guaranteed if the space V_{N_p} tends to V as $N_p \to \infty$. Since the differential operator is linear, we demand that the variational equation is satisfied for Np test-functions $\Phi_i \in V_{N_p}$ that form a basis, i.e.,

$$\int_\Omega (c\nabla u)\cdot\nabla\phi_i + au\phi_i - f\phi_i dx - \int_{\partial\Omega}(-qu+g)\phi_i ds = 0, \qquad i = 1, \ldots, N_p$$

Expand u in the same basis of V_{N_p}

$$u(x) = \sum_{j=1}^{N_p} U_j\phi_j(x),$$

and obtain the system of equations

$$\sum_{j=1}^{N_p}\left(\int_{\Omega}(c\nabla\phi_j)\cdot\nabla\phi_i + a\phi_j\phi_i dx + \int_{\partial\Omega} q\phi_j\phi_i ds\right)U_j$$

$$= \int_{\Omega} f\phi_i dx + \int_{\partial\Omega} g\phi_i ds \qquad i = 1, \ldots, N_p$$

Use the following notations:

$$K_{i,j} = \int_{\Omega}(c\nabla\phi_j)\cdot\nabla\phi_i dx \text{ (Stiffness matrix)}$$

$$M_{i,j} = \int_{\Omega} a\phi_j\phi_i dx \text{ (Mass matrix)}$$

$$Q_{i,j} = \int_{\partial\Omega} q\phi_j\phi_i ds$$

$$F_i = \int_{\Omega} f\phi_i dx$$

$$G_i = \int_{\partial\Omega} g\phi_i ds$$

and rewrite the system in the form $(K + M + Q)U = F + G$.

K, M, and Q are Np-by-Np matrices, and F and G are Np-vectors. K, M, and F are produced by `assema`, while Q, G are produced by `assemb`. When it is not necessary to distinguish K, M, and Q or F and G, we collapse the notations to $KU = F$, which form the output of `assempde`.

When the problem is *self-adjoint* and *elliptic* in the usual mathematical sense, the matrix $K + M + Q$ becomes symmetric and positive definite. Many common problems have these characteristics, most notably those that can also be formulated as minimization problems. For the case of a scalar equation, K, M, and Q are obviously symmetric. If $c(x) \geq \delta > 0$, $a(x) \geq 0$ and $q(x) \geq 0$ with $q(x) > 0$ on some part of $\partial\Omega$, then, if $U \neq 0$.

$$U^T(K + M + Q)U = \int_{\Omega} c|\nabla u|^2 + au^2 dx + \int_{\partial\Omega} qu^2 ds > 0, \quad \text{if } U \neq 0$$

UT$(K + M + Q)U$ is the *energy norm*. There are many choices of the test-function spaces. The software uses continuous functions that are linear

on each triangle of the mesh. Piecewise linearity guarantees that the integrals defining the stiffness matrix K exist. Projection onto V_{N_p} is nothing more than linear interpolation, and the evaluation of the solution inside a triangle is done just in terms of the nodal values. If the mesh is uniformly refined, V_{N_p} approximates the set of smooth functions on Ω.

A suitable basis for V_{N_p} is the set of "tent" or "hat" functions Φ_i. These are linear on each triangle and take the value 0 at all nodes xj except for xi. Requesting $\Phi_i(x_i) = 1$ yields the very pleasant property

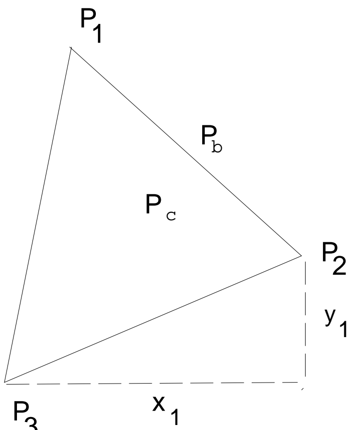
$$u(x_i) = \sum_{j=1}^{N_p} U_j \phi_j(x_i) = U_i$$

That is, by solving the FEM system we obtain the nodal values of the approximate solution. The basis function Φ_i vanishes on all the triangles that do not contain the node x_i. The immediate consequence is that the integrals appearing in $K_{i,j}$, $M_{i,j}$, $Q_{i,j}$, F_i and G_i only need to be computed on the triangles that contain the node x_i. Secondly, it means that $K_{i,j}$ and $M_{i,j}$ are zero unless x_i and x_j are vertices of the same triangle and thus K and M are very sparse matrices. Their sparse structure depends on the ordering of the indices of the mesh points.

The integrals in the FEM matrices are computed by adding the contributions from each triangle to the corresponding entries (i.e., only if the corresponding mesh point is a vertex of the triangle). This process is commonly called *assembling*, hence the name of the function `assempde`.

The assembling routines scan the triangles of the mesh. For each triangle they compute the so-called local matrices and add their components to the correct positions in the sparse matrices or vectors. (The local 3-by-3 matrices contain the integrals evaluated only on the current triangle. The coefficients are assumed constant on the triangle and they are evaluated only in the triangle barycenter.) The integrals are computed using the midpoint rule. This approximation is optimal since it has the same order of accuracy as the piecewise linear interpolation.

Consider a triangle given by the nodes P_1, P_2, and P_3 as in the following figure.



The Local Triangle P1P2P3

Note The local 3-by-3 matrices contain the integrals evaluated only on the current triangle. The coefficients are assumed constant on the triangle and they are evaluated only in the triangle barycenter.

The simplest computations are for the local mass matrix m:

$$m_{i,j} = \int_{\nabla P_1 P_2 P_3} a(P_c)\phi_i(x)\phi_j(x)dx = a(P_c)\frac{\text{area}(\Delta P_1 P_2 P_3)}{12}(1+\delta_{i,j})$$

where P_c is the center of mass of $\Delta\ P_1P_2P_3$, i.e.,

$$P_c = \frac{P_1 + P_2 + P_3}{3}$$

The contribution to the right side F is just

$$f_i = f(P_c)\frac{\text{area}(\Delta P_1 P_2 P_3)}{3}$$

For the local stiffness matrix we have to evaluate the gradients of the basis functions that do not vanish on $P_1P_2P_3$. Since the basis functions are linear on

the triangle $P_1P_2P_3$, the gradients are constants. Denote the basis functions Φ_1, Φ_2, and Φ_3 such that $\Phi(P_i) = 1$. If $P_2 - P_3 = [x_1,y_1]^T$ then we have that

$$\nabla\phi_1 = \frac{1}{2\,\mathrm{area}(\Delta P_1P_2P_3)}\begin{bmatrix} y_1 \\ -x_1 \end{bmatrix}$$

and after integration (taking c as a constant matrix on the triangle)

$$k_{i,j} = \frac{1}{4\,\mathrm{area}(\Delta P_1P_2P_3)}[y_j, -x_j]c(P_c)\begin{bmatrix} y_1 \\ -x_1 \end{bmatrix}$$

If two vertices of the triangle lie on the boundary $\partial\Omega$, they contribute to the line integrals associated to the boundary conditions. If the two boundary points are P_1 and P_2, then we have

$$Q_{i,j} = q(P_b)\frac{\|P_1 - P_2\|}{6}(1 + \delta_{i,j}), \qquad i, j = 1, 2$$

and

$$G_i = g(P_b)\frac{\|P_1 - P_2\|}{2}, \qquad i = 1, 2$$

where P_b is the midpoint of P_1P_2.

For each triangle the vertices P_m of the local triangle correspond to the indices im of the mesh points. The contributions of the individual triangle are added to the matrices such that, e.g.,

$$K_{i_m, i_n} t \leftarrow K_{i_m, i_n} + k_{m,n}, \qquad m, n = 1, 2, 3$$

This is done by the function `assempde`. The gradients and the areas of the triangles are computed by the function `pdetrg`.

The Dirichlet boundary conditions are treated in a slightly different manner. They are eliminated from the linear system by a procedure that yields a symmetric, reduced system. The function `assempde` can return matrices *K,*

F, B, and *ud* such that the solution is $u = Bv + ud$ where $Kv = F$. u is an N_p-vector, and if the rank of the Dirichlet conditions is rD, then v has N_p - rD components.

The Elliptic System

Partial Differential Equation Toolbox software can also handle systems of N partial differential equations over the domain Ω. We have the elliptic system

$$-\nabla \cdot (\underline{c} \otimes \nabla u) + \underline{a} u = f$$

the parabolic system

$$\underline{d}\frac{\partial u}{\partial t} - \nabla \cdot (\underline{c} \otimes \nabla u) + \underline{a} u = f$$

the hyperbolic system

$$\underline{d}\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (\underline{c} \otimes \nabla u) + \underline{a} u = f$$

$$\underline{d}\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u}) + \underline{a}\mathbf{u} = \mathbf{f}$$

and the eigenvalue system

$$-\nabla \cdot (\underline{c} \otimes \nabla u) + \underline{a} u = \lambda \underline{d} u$$

where $\underline{\mathbf{c}}$ is an N-by-N-by-2-by-2 tensor. By the notation $\nabla \cdot (c \otimes \nabla \mathbf{u})$, we mean the N-by-1 matrix with $(i,1)$-component.

$$\sum_{j=1}^{N}\left(\frac{\partial}{\partial x}c_{i,j,1,1}\frac{\partial}{\partial x} + \frac{\partial}{\partial x}c_{i,j,1,2}\frac{\partial}{\partial y} + \frac{\partial}{\partial y}c_{i,j,2,1}\frac{\partial}{\partial x} + \frac{\partial}{\partial y}c_{i,j,2,2}\frac{\partial}{\partial y}\right)u_j$$

The symbols $\underline{a}$ and $\underline{d}$ denote N-by-N matrices, and $\mathbf{u}$ denotes column vectors of length N.

The elements c_{ijkl}, a_{ij}, $d_{ij,}$ and f_i of $\underline{\mathbf{c}}$, $\underline{a}$, $\underline{d}$, and $\mathbf{f}$ are stored row-wise in the MATLAB matrices `c`, `a`, `d`, and `f`. The case of identity, diagonal, and symmetric matrices are handled as special cases. For the tensor c_{ijkl} this applies both to the indices i and j, and to the indices k and l.

Partial Differential Equation Toolbox software does not check the ellipticity of the problem, and it is quite possible to define a system that is *not* elliptic in the mathematical sense. The preceding procedure that describes the scalar case is applied to each component of the system, yielding a symmetric positive definite system of equations whenever the differential system possesses these characteristics.

The boundary conditions now in general are *mixed*, i.e., for each point on the boundary a combination of Dirichlet and generalized Neumann conditions,

$$\underline{h}\mathbf{u} = \mathbf{r}$$
$$\vec{n} \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u}) + \underline{q}\mathbf{u} = \mathbf{g} + \underline{h}'\mu$$

By the notation $\vec{n} = (\underline{\mathbf{c}} \otimes \nabla \mathbf{u})$ we mean the N-by-1 matrix with $(i,1)$-component

$$\sum_{j=1}^{N} \left(\cos(\alpha) c_{i,j,1,1} \frac{\partial}{\partial x} + \cos(\alpha) c_{i,j,1,2} \frac{\partial}{\partial y} + \sin(\alpha) c_{i,j,2,1} \frac{\partial}{\partial x} + \sin(\alpha) c_{i,j,2,2} \frac{\partial}{\partial y} \right) u_j$$

where the outward normal vector of the boundary is $\vec{n} = (\cos(\alpha), \sin(\alpha))$. There are M Dirichlet conditions and the $\underline{h}$ -matrix is M-by-N, $M \geq 0$. The generalized Neumann condition contains a source $\underline{h}'\mu$, where the Lagrange multipliers μ are computed such that the Dirichlet conditions become satisfied. In a structural mechanics problem, this term is exactly the reaction force necessary to satisfy the kinematic constraints described by the Dirichlet conditions.

The rest of this section details the treatment of the Dirichlet conditions and may be skipped on a first reading.

Partial Differential Equation Toolbox software supports two implementations of Dirichlet conditions. The simplest is the “Stiff Spring” model, so named for its interpretation in solid mechanics. See “The Elliptic Equation” on page 3-2 for the scalar case, which is equivalent to a diagonal h-matrix. For the general case, Dirichlet conditions

$$\underline{h}\mathbf{u} = \mathbf{r}$$

are approximated by adding a term

$$L(\underline{h}'\,\underline{h}\mathbf{u} - \underline{h}'\,\mathbf{r})$$

to the equations $KU = F$, where L is a large number such as 10^4 times a representative size of the elements of K.

When this number is increased, $\underline{h}\mathbf{u} = \mathbf{r}$ will be more accurately satisfied, but the potential ill-conditioning of the modified equations will become more serious.

The second method is also applicable to general mixed conditions with nondiagonal $\underline{h}$, and is free of the ill-conditioning, but is more involved computationally. Assume that there are N_p nodes in the triangulation. Then the number of unknowns is $N_pN = N_u$. When Dirichlet boundary conditions fix some of the unknowns, the linear system can be correspondingly reduced. This is easily done by removing rows and columns when u values are given, but here we must treat the case when some linear combinations of the components of u are given, $\underline{h}\mathbf{u} = \mathbf{r}$. These are collected into $HU = R$ where H is an M-by-N_u matrix and R is an M-vector.

With the reaction force term the system becomes

$$KU + H'\mu = F$$

$$HU = R$$

The constraints can be solved for m of the U-variables, the remaining called V, an $N_u - m$ vector. The null space of H is spanned by the columns of B, and $U = BV + u_d$ makes U satisfy the Dirichlet conditions. A permutation to block-diagonal form exploits the sparsity of H to speed up the following computation to find B in a numerically stable way. μ can be eliminated by premultiplying by B' since, by the construction, $HB = 0$ or $B'H' = 0$. The reduced system becomes

$$B'KBV = B'F - B'Ku_d$$

which is symmetric and positive definite if K is.

The Parabolic Equation

In this section...

"Reducing the Parabolic Equation to Elliptic Equations" on page 3-13

"Solving the Parabolic Equation in Stages" on page 3-15

Reducing the Parabolic Equation to Elliptic Equations

The elliptic solver allows other types of equations to be more easily implemented. In this section, we show how the parabolic equation can be reduced to solving elliptic equations. This is done using the function `parabolic`.

Consider the equation

$$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f \qquad \text{in } \Omega$$

with the initial condition

$$u(x, 0) = u_0(x) \qquad x \in \Omega$$

and boundary conditions of the same kind as for the elliptic equation on $\partial\Omega$.

The heat equation reads

$$\rho C\frac{\partial u}{\partial t} - \nabla \cdot (k\nabla u) + h(u - u_\infty) = f$$

in the presence of distributed heat loss to the surroundings. ρ is the density, C is the thermal capacity, k is the thermal conductivity, h is the film coefficient, u_∞ is the ambient temperature, and f is the heat source.

For time-independent coefficients, the steady-state solution of the equation is the solution to the standard elliptic equation

$$-\nabla \cdot (c\nabla u) + au = f$$

Assuming a triangular mesh on Ω and $t \geq 0$, expand the solution to the PDE (as a function of x) in the Finite Element Method basis:

$$u(x,t) = \sum_i U_i(t)\phi_i(x)$$

Plugging the expansion into the PDE, multiplying with a test function Φ_j, integrating over Ω, and applying Green's formula and the boundary conditions yield

In matrix notation, we have to solve the *linear, large* and *sparse* ODE system

$$M\frac{dU}{dt} + KU = F$$

This method is traditionally called *method of lines* semidiscretization.

Solving the ODE with the initial value

$$U_i(0) = u_0(x_i)$$

$$\sum_i \int_\Omega d\phi_j\phi_i dx \frac{dU_i(t)}{dt} + \sum_i \left(\int_\Omega \nabla\phi_j \cdot (c\nabla\phi_i) + a\phi_j\phi_i dx + \int_{\partial\Omega} q\phi_j\phi_i ds\right)U_i(t)$$

$$= \int_\Omega f\phi_j dx + \int_{\partial\Omega} g\phi_j ds \qquad \forall j$$

yields the solution to the PDE at each node x_i and time t. K and$F$ are the stiffness matrix and the right-hand side of the elliptic problem

$$-\nabla \cdot (c\nabla u) + au = f \text{ in } \Omega$$

with the original boundary conditions while M is just the mass matrix of the problem

$$-\nabla \cdot (0\nabla u) + du = 0 \text{ in } \Omega$$

When the Dirichlet conditions are time dependent, F contains contributions from time derivatives of $\underline{h}$ h and $\mathbf{r}$. These derivatives are evaluated by finite differences of the user-specified data.

The ODE system is ill conditioned. Explicit time integrators are forced by stability requirements to very short time steps while implicit solvers can be expensive since they solve an elliptic problem at every time step. The numerical integration of the ODE system is performed by the MATLAB ODE Suite functions, which are efficient for this class of problems. The time step is controlled to satisfy a tolerance on the error, and factorizations of coefficient matrices are performed only when necessary. When coefficients are time dependent, the necessity of reevaluating and refactorizing the matrices each time step may still make the solution time consuming, although `parabolic` reevaluates only that which varies with time. In certain cases a time-dependent Dirichlet matrix $\underline{h}(t)$ may cause the error control to fail, even if the problem is mathematically sound and the solution $u(t)$ is smooth. This can happen because the ODE integrator looks only at the reduced solution v with $u = Bv + ud$. As $\underline{h}$ changes, the pivoting scheme employed for numerical stability may change the elimination order from one step to the next. This means that B,v and ud all change discontinuously, although u itself does not.

Solving the Parabolic Equation in Stages

The following example shows you how to solve a parabolic equation in stages and how to set an initial condition as a variable:

1. At the MATLAB command prompt, type `pdetool`.

2. Draw a rectangle in the GUI axes.

3. From the **Draw** menu, select **Export Geometry Description, Set Formula, Labels**.

4. In the Export dialog box, enter `gd sf ns`. Click **OK**.

   The exported variables are available in the MATLAB workspace.

5. From the **Boundary** menu, select **Boundary Mode**.

6. From the **Boundary** menu, select **Specify Boundary Conditions**.

7. Set the Neumann and Dirichlet boundary conditions. If these conditions are not the same for all the stages, set the conditions accordingly.

8 From the **Boundary** menu, select **Export Decomposed Geometry, Boundary Cond's**.

9 In the Export dialog box, enter `g b`. Click **OK**.

10 From the **PDE** menu, select **PDE Mode**.

11 From the **PDE** menu, select **PDE Specification**.

12 Set the partial differential equation (PDE) coefficients, which are the same for any value of time.

13 From the **PDE** menu, select **Export PDE Coefficients.**

14 In the Export dialog box, enter `c a f d`. Click **OK**.

15 From the **Mesh** menu, select **Mesh Mode**.

16 From the **Mesh** menu, select **Parameters**.

17 Verify the initial mesh, jiggle mesh, and refine mesh values. The mesh is fixed for all stages.

18 From the **Mesh** menu, select **Export Mesh**.

19 In the Export dialog box, enter `p e t`. Click **OK**.

20 Save the workspace variables into a MAT-file by typing `save data.mat` at the MATLAB command prompt.

21 Save the following code as a file:

```
clear all;
close all;
load data

%For the first stage you need to specify an
%initial condition, U0.
U0 = 0; %U0 expands to the correct size automatically.

%Divide the time range into 4 stages.
time = {0:.01:1, 1:.05:3, 3:.1:5, 5:.5:20};
```

```
for i = 1:4
U1 = parabolic(U0,time{i},b,p,e,t,c,a,f,d);
for j = 1:size(U1,2)
H =pdeplot(p,e,t,'xydata',U1(:,j),'zdata',...
U1(:,j),'mesh','off');
set(gca,'ZLim',[-80 0]);
drawnow
end
%Reset the initial condition at all points.
U0 = U1(:,1);
end
```

This file uses the variables you defined in the MATLAB workspace to solve a parabolic equation in stages. Within this file, you set the initial condition as a variable.

The Hyperbolic Equation

Using the same ideas as for the parabolic equation, `hyperbolic` implements the numerical solution of

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f \quad \text{in } \Omega$$

with the initial conditions

$$u(x,0) = u_0(x) \qquad \frac{\partial u}{\partial t}(x,0) = v_0(x) \qquad x \in \Omega$$

and usual boundary conditions. In particular, solutions of the equation u_{tt} - $c\Delta u = 0$ are waves moving with speed $\sqrt{c}$.

Using a given triangulation of Ω, the method of lines yields the second order ODE system

$$M\frac{d^2 U}{dt^2} + KU = F$$

with the initial conditions

$$U_i(0) = u_0(x_i), \qquad \frac{d}{dt}U_i(0) = v_0(x_i) \qquad \forall i$$

after we eliminate the unknowns fixed by Dirichlet boundary conditions. As before, the stiffness matrix K and the mass matrix M are assembled with the aid of the function `assempde` from the problems

$-\nabla \cdot (c\nabla u) + au = f$ and $-\nabla \cdot (0\nabla u) + du = 0$

The Eigenvalue Equation

Partial Differential Equation Toolbox software handles the following basic eigenvalue problem:

$$-\nabla \cdot (c\nabla u) + au = \lambda du$$

where λ is an unknown complex number. In solid mechanics, this is a problem associated with wave phenomena describing, e.g., the natural modes of a vibrating membrane. In quantum mechanics λ is the energy level of a bound state in the potential well $a(x)$.

The numerical solution is found by discretizing the equation and solving the resulting algebraic eigenvalue problem. Let us first consider the discretization. Expand u in the FEM basis, multiply with a basis element, and integrate on the domain Ω. This yields the generalized eigenvalue equation

$$KU = \lambda MU$$

where the mass matrix corresponds to the right side, i.e.,

$$M_{i,j} = \int_{\Omega} d(x)\phi_j(x)\phi_i(x)dx$$

The matrices K and M are produced by calling `assema` for the equations

$$-\nabla \cdot (c\nabla u) + au = 0 \text{ and } -\nabla \cdot (0\nabla u) + du = 0$$

In the most common case, when the function $d(x)$ is positive, the mass matrix M is positive definite symmetric. Likewise, when $c(x)$ is positive and we have Dirichlet boundary conditions, the stiffness matrix K is also positive definite.

The generalized eigenvalue problem, $KU = \lambda MU$, is now solved by the *Arnoldi algorithm* applied to a shifted and inverted matrix with restarts until all eigenvalues in the user-specified interval have been found.

Let us describe how this is done in more detail. You may want to look at the example provided in the section “Eigenvalue Problems” on page 1-75, where an actual run is reported.

First a shift μ is determined close to where we want to find the eigenvalues. When both K and M are positive definite, it is natural to take $\mu = 0$, and get the smallest eigenvalues; in other cases take any point in the interval $[lb,ub]$ where eigenvalues are sought. Subtract μM from the eigenvalue equation and get $(K - \mu M)U = (\lambda - \mu)MU$. Then multiply with the inverse of this shifted matrix and get

$$\frac{1}{\lambda - \mu}U = (K - \mu M)^{-1}MU$$

This is a standard eigenvalue problem $AU = \theta U$, with the matrix $A = (K - \mu M)^{-1}M$ and eigenvalues

$$\theta_i = \frac{1}{\lambda_i - \mu}$$

where $i = 1, \ldots, n$. The largest eigenvalues θ_i of the transformed matrix A now correspond to the eigenvalues ºi $= \mu + 1/\theta_i$ of the original *pencil* (K,M) closest to the shift μ.

The Arnoldi algorithm computes an orthonormal basis V where the shifted and inverted operator A is represented by a Hessenberg matrix H,

$$AV_j = V_j H_{j,j} + E_j$$

(The subscripts mean that V_j and E_j have j columns and $H_{j,j}$ has j rows and columns. When no subscripts are used we deal with vectors and matrices of size n.)

Some of the eigenvalues of this Hessenberg matrix $H_{j,j}$ eventually give good approximations to the eigenvalues of the original pencil (K,M) when the basis grows in dimension j, and less and less of the eigenvector is hidden in the residual matrix E_j.

The basis V is built one column v_j at a time. The first vector v_1 is chosen at random, as n normally distributed random numbers. In step j, the first j vectors are already computed and form the $n \times j$ matrix V_j. The next vector v_{j+1} is computed by first letting A operate on the newest vector v_j, and then making the result orthogonal to all the previous vectors.

This is formulated as $h_{j+1}v_{j+1} = Av_j - V_j h_j$, where the column vector h_j consists of the Gram-Schmidt coefficients, and $h_{j+1,j}$ is the normalization factor that gives v_{j+1} unit length. Put the corresponding relations from previous steps in front of this and get

$$AV_j = V_j H_{j,j} + v_{j+1} h_{j+1,j} e_j^T$$

where $H_{j,j}$ is a $j \times j$ Hessenberg matrix with the vectors h_j as columns. The second term on the right-hand side has nonzeros only in the last column; the earlier normalization factors show up in the subdiagonal of $H_{j,j}$.

The eigensolution of the small Hessenberg matrix H gives approximations to some of the eigenvalues and eigenvectors of the large matrix operator $A_{j,j}$ in the following way. Compute eigenvalues θ_i and eigenvectors s_i of $H_{j,j}$,

$$H_{j,j}s_i = s_i\theta_i, \text{""} i = 1, \ldots, j$$

Then $y_i = V_j s_i$ is an approximate eigenvector of A, and its residual is

$$r_i = Ay_i - y_i\theta_i = AV_j s_i - V_j s_i \theta_i = (AV_j - V_j H_{j,j})s_i = v_{j+1} h_{j+1,j} s_{i,j}$$

This residual has to be small in norm for θ_i to be a good eigenvalue approximation. The norm of the residual is $\|r_i\| = |h_{j+1,j} s_{j,i}|$, the product of the last subdiagonal element of the Hessenberg matrix and the last element of its eigenvector. It seldom happens that $h_{j+1,j}$ gets particularly small, but after sufficiently many steps j there are always some eigenvectors s_i with small last elements. The long vector V_{j+1} is of unit norm.

It is not necessary to actually compute the eigenvector approximation y_i to get the norm of the residual; we only need to examine the short vectors s_i, and flag those with tiny last components as converged. In a typical case n may be 2000, while j seldom exceeds 50, so all computations that involve only matrices and vectors of size j are much cheaper than those involving vectors of length n.

This eigenvalue computation and test for convergence is done every few steps j, until all approximations to eigenvalues inside the interval $[lb,ub]$ are flagged as converged. When n is much larger than j, this is done very

often, for smaller n more seldom. When all eigenvalues inside the interval have converged, or when j has reached a prescribed maximum, the converged eigenvectors, or more appropriately *Schur vectors*, are computed and put in the front of the basis V.

After this, the Arnoldi algorithm is restarted with a random vector, if all approximations inside the interval are flagged as converged, or else with the best unconverged approximate eigenvector y_i. In each step j of this second Arnoldi run, the vector is made orthogonal to all vectors in V including the converged Schur vectors from the previous runs. This way, the algorithm is applied to a projected matrix, and picks up a second copy of any double eigenvalue there may be in the interval. If anything in the interval converges during this second run, a third is attempted and so on, until no more approximate eigenvalues θ_i show up inside. Then the algorithm signals convergence. If there are still unconverged approximate eigenvalues after a prescribed maximum number of steps, the algorithm signals nonconvergence and reports all solutions it has found.

This is a heuristic strategy that has worked well on both symmetric, nonsymmetric, and even defective eigenvalue problems. There is a tiny theoretical chance of missing an eigenvalue, if all the random starting vectors happen to be orthogonal to its eigenvector. Normally, the algorithm restarts p times, if the maximum multiplicity of an eigenvalue is p. At each restart a new random starting direction is introduced.

The shifted and inverted matrix $A = (K - \mu M)^{-1}M$ is needed only to operate on a vector v_j in the Arnoldi algorithm. This is done by computing an LU factorization,

$$P(K - \mu M)Q = LU$$

using the sparse MATLAB command `lu` (P and Q are permutations that make the triangular factors L and U sparse and the factorization numerically stable). This factorization needs to be done only once, in the beginning, then $x = Av_j$ is computed as,

$$x = QU^{-1}L^{-1}PMv\mathrm{j}$$

with one sparse matrix vector multiplication, a permutation, sparse forward- and back-substitutions, and a final renumbering.

Nonlinear Equations

The low-level Partial Differential Equation Toolbox functions are aimed at solving linear equations. Since many interesting computational problems are nonlinear, the software contains a nonlinear solver built on top of the `assempde` function.

Note Before solving a nonlinear PDE, from the **Solve** menu in the `pdetool` GUI, select **Parameters**. Then, select the **Use nonlinear solver** check box and click **OK**.

The basic idea is to use Gauss-Newton iterations to solve the nonlinear equations. Say you are trying to solve the equation

$$r(u) = -\nabla \cdot (c(u)\nabla u) + a(u)u - f(u) = 0.$$

In the FEM setting you solve the weak form of $r(u) = 0$. Set as usual

$$u(x) = \sum U_j \phi_j$$

then, multiply the equation by an arbitrary test function Φ_i, integrate on the domain Ω, and use Green's formula and the boundary conditions to obtain

$$\begin{aligned} 0 = \rho(U) = \sum_j \Big(& \int_\Omega (c(x,U)\nabla\phi_j(x)) \cdot \nabla\phi_i(x) + a(x,U)\phi_j(x)\phi_i(x)dx \\ & + \int_{\partial\Omega} q(x,U)\phi_j(x)\phi_i(x)ds \Big) U_j \\ & - \int_\Omega f(x,U)\phi_i(x)dx - \int_{\partial\Omega} g(x,U)\phi_i(x)ds \end{aligned}$$

which has to hold for all indices i.

The residual vector $\rho(U)$ can be easily computed as

$$\rho(U) = (K + M + Q)U - (F + G)$$

where the matrices K, M, Q and the vectors F and G are produced by assembling the problem

$$-\nabla \cdot (c(U)\nabla u) + a(U)u = f(U)$$

Assume that you have a guess $U^{(n)}$ of the solution. If $U^{(n)}$ is close enough to the exact solution, an improved approximation $U^{(n+1)}$ is obtained by solving the linearized problem

$$\left(\frac{\partial\rho(U^{(n)})}{\partial U}\right)(U^{(n+1)} - U^{(n)}) = -\alpha\rho(U^{(n)})$$

where α is a positive number. (It is not necessary that $\rho(U)=0$ have a solution even if $\rho(u) = 0$ has. In this case, the Gauss-Newton iteration tends to be the minimizer of the residual, i.e., the solution of minU $||\rho(U)||$).

It is well known that for sufficiently small α

$$||\rho(U^{(n+1)})|| < ||\rho(U^{(n)})||$$

and

$$p_n = \left(\frac{\partial\rho(U^{(n)})}{\partial U}\right)^{-1}\rho(U^{(n)})$$

is called a descent direction for $||\rho(U)||$, where $||\cdot||$ is the l_2-norm. The iteration is

$$U^{(n+1)} = U^{(n)} + \alpha p_n$$

where $\alpha \le 1$ is chosen as large as possible such that the step has a reasonable descent.

The *Gauss-Newton method* is local, and convergence is assured only when $U^{(0)}$ is close enough to the solution. In general, the first guess may be outside the region of convergence. To improve convergence from bad initial guesses, a *damping* strategy is implemented for choosing α, the *Armijo-Goldstein line search*. It chooses the largest damping coefficient α out of the sequence 1, 1/2, 1/4, . . . such that the following inequality holds:

$$||\rho(U^{(n)})|| - ||\rho(U^{(n)} + \alpha p_n)|| \geq \frac{\alpha}{2}||\rho(U^{(n)})||$$

which guarantees a reduction of the residual norm by at least 1 - α/2. Each step of the line-search algorithm requires an evaluation of the residual $\rho(U^{(n)} + \alpha p_n)$.

An important point of this strategy is that when $U^{(n)}$ approaches the solution, then $\alpha \rightarrow 1$ and thus the convergence rate increases. If there is a solution to $\rho(U) = 0$, the scheme ultimately recovers the quadratic convergence rate of the standard Newton iteration.

Closely related to the preceding problem is the choice of the initial guess $U^{(0)}$. By default, the solver sets $U^{(0)}$ and then assembles the FEM matrices K and F and computes

$$U^{(1)} = K^{-1}F$$

The damped Gauss-Newton iteration is then started with $U^{(1)}$, which should be a better guess than $U^{(0)}$. If the boundary conditions do not depend on the solution u, then $U^{(1)}$ satisfies them even if $U^{(0)}$ does not. Furthermore, if the equation is linear, then $U^{(1)}$ is the exact FEM solution and the solver does not enter the Gauss-Newton loop.

There are situations where $U^{(0)} = 0$ makes no sense or convergence is impossible.

In some situations you may already have a good approximation and the nonlinear solver can be started with it, avoiding the slow convergence regime. This idea is used in the adaptive mesh generator. It computes a solution $\tilde{U}$ on a mesh, evaluates the error, and may refine certain triangles. The interpolant of $\tilde{U}$ is a very good starting guess for the solution on the refined mesh.

In general the exact Jacobian

$$J_n = \frac{\partial \rho(U^{(n)})}{\partial U}$$

is not available. Approximation of J_n by finite differences in the following way is expensive but feasible. The ith column of J_n can be approximated by

$$\frac{1}{\varepsilon}(\rho(U^{(n)} + \varepsilon\phi_i) - \rho(U^{(n)}))$$

which implies the assembling of the FEM matrices for the triangles containing grid point i. A very simple approximation to J_n, which gives a fixed point iteration, is also possible as follows. Essentially, for a given $U^{(n)}$, compute the FEM matrices K and F and set

$$U^{(n+1)} = K^{-1}F$$

This is equivalent to approximating the Jacobian with the stiffness matrix. Indeed, since $\rho(U^{(n)}) = KU^{(n)} - F$, putting $J_n = K$ yields

$$U^{(n+1)} = U^{(n)} - J_n^{-1}\rho(U^{(n)}) = U^{(n)} - K^{-1}(KU^{(n)} - F) = K^{-1}F$$

In many cases the convergence rate is slow, but the cost of each iteration is cheap.

The Partial Differential Equation Toolbox nonlinear solver also provides for a compromise between the two extremes. To compute the derivative of the mapping $U \to KU$, proceed as follows. The a term has been omitted for clarity, but appears again in the final result.

$$\begin{aligned}\frac{\partial(KU)_i}{\partial U_j} &= \lim_{\varepsilon \to 0}\frac{1}{\varepsilon}\sum_l\Big(\int_\Omega c(U + \varepsilon\phi_j)\nabla\phi_l\nabla\phi_i dx(U_l + \varepsilon\delta_{l,j}) \\ &\qquad - \int_\Omega c(U)\nabla\phi_l\nabla\phi_i dx U_l\Big) \\ &= \int_\Omega c(U)\nabla\phi_j\nabla\phi_i dx + \sum_l\int_\Omega \phi_j\frac{\partial c}{\partial u}\nabla\phi_l\nabla\phi_i dx U_l\end{aligned}$$

The first integral term is nothing more than $K_{i,j}$.

The second term is "lumped," i.e., replaced by a diagonal matrix that contains the row sums. Since $\Sigma_j\Phi_j = 1$, the second term is approximated by

$$\delta_{i,j}\sum_l \int_\Omega \frac{\partial c}{\partial u}\nabla\phi_l\nabla\phi_i\,dx U_l$$

which is the ith component of $K^{(c')}U$, where $K^{(c')}$ is the stiffness matrix associated with the coefficient $\frac{\partial c}{\partial u}$ rather than c. The same reasoning can be applied to the derivative of the mapping $U \mapsto MU$. The derivative of the mapping $U \mapsto -F$ is exactly

$$-\int_\Omega \frac{\partial f}{\partial u}\phi_i\phi_j\,dx$$

which is the mass matrix associated with the coefficient $\frac{\partial f}{\partial u}$. Thus the Jacobian of the residual $\rho(U)$ is approximated by

$$J = K^{(c)} + M^{(a-f')} + \mathrm{diag} + ((K^{(c')} + M^{(a')})U)$$

where the differentiation is with respect to u K and M designate stiffness and mass matrices and their indices designate the coefficients with respect to which they are assembled. At each Gauss-Newton iteration, the nonlinear solver assembles the matrices corresponding to the equations

$$-\nabla\cdot(c\nabla u) + (a - f')u = 0$$
$$-\nabla\cdot(c'\nabla u) + a'u = 0$$

and then produces the approximate Jacobian. The differentiations of the coefficients are done numerically.

In the general setting of elliptic systems, the boundary conditions are appended to the stiffness matrix to form the full linear system:

$$\tilde{K}\tilde{U} = \begin{bmatrix} K & H' \\ H & 0 \end{bmatrix}\begin{bmatrix} U \\ \mu \end{bmatrix} = \begin{bmatrix} F \\ R \end{bmatrix} = \tilde{F}$$

where the coefficients of $\tilde{K}$ and $\tilde{F}$ may depend on the solution $\tilde{U}$. The "lumped" approach approximates the derivative mapping of the residual by

$$\begin{bmatrix} J & H' \\ H & 0 \end{bmatrix}$$

The nonlinearities of the boundary conditions and the dependencies of the coefficients on the derivatives of $\tilde{U}$ are not properly linearized by this scheme. When such nonlinearities are strong, the scheme reduces to the fix-point iteration and may converge slowly or not at all. When the boundary conditions are linear, they do not affect the convergence properties of the iteration schemes. In the Neumann case they are invisible (H is an empty matrix) and in the Dirichlet case they merely state that the residual is zero on the corresponding boundary points.

Adaptive Mesh Refinement

In this section...

"Introduction" on page 3-29

"The Error Indicator Function" on page 3-30

"The Mesh Refiner" on page 3-31

"The Termination Criteria" on page 3-31

Introduction

Partial Differential Equation Toolbox software has a function for global, uniform mesh refinement. It divides each triangle into four similar triangles by creating new corners at the midsides, adjusting for curved boundaries. You can assess the accuracy of the numerical solution by comparing results from a sequence of successively refined meshes. If the solution is smooth enough, more accurate results may be obtained by extrapolation.

The solutions of equations often have geometric features like localized strong gradients. An example of engineering importance in elasticity is the stress concentration occurring at reentrant corners such as the MATLAB L-shaped membrane. Then it is more economical to refine the mesh selectively, i.e., only where it is needed. When the selection is based on estimates of errors in the computed solutions, a posteriori estimates, we speak of *adaptive mesh refinement*. See `adaptmesh` for an example of the computational savings where global refinement needs more than 6000 elements to compete with an adaptively refined mesh of 500 elements.

The adaptive refinement generates a sequence of solutions on successively finer meshes, at each stage selecting and refining those elements that are judged to contribute most to the error. The process is terminated when the maximum number of elements is exceeded or when each triangle contributes less than a preset tolerance. You need to provide an initial mesh, and choose selection and termination criteria parameters. The initial mesh can be produced by the `initmesh` function. The three components of the algorithm are the error indicator function, which computes an estimate of the element error contribution, the mesh refiner, which selects and subdivides elements, and the termination criteria.

The Error Indicator Function

The adaptation is a feedback process. As such, it is easily applied to a larger range of problems than those for which its design was tailored. You want estimates, selection criteria, etc., to be optimal in the sense of giving the most accurate solution at fixed cost or lowest computational effort for a given accuracy. Such results have been proved only for model problems, but generally, the equidistribution heuristic has been found near optimal. Element sizes should be chosen such that each element contributes the same to the error. The theory of adaptive schemes makes use of *a priori* bounds for solutions in terms of the source function *f*. For nonelliptic problems such a bound may not exist, while the refinement scheme is still well defined and has been found to work well.

The error indicator function used in the software is an element-wise estimate of the contribution, based on the work of C. Johnson et al. [5], [6]. For Poisson's equation $-\Delta u = f$ on Ω, the following error estimate for the FEM-solution u_h holds in the L_2-norm $\|\cdot\|$:

$$\| \nabla(u - u_h) \| \le \alpha \| hf \| + \beta D_h(u_h)$$

where $h = h(x)$ is the local mesh size, and

$$D_h(v) = \left(\sum_{\tau \in E_1} h_\tau^2 [\partial v / \partial n_\tau]^2 \right)^{1/2}$$

The braced quantity is the jump in normal derivative of v across edge τ, h_r is the length of edge τ, and the sum runs over E_i, the set of all interior edges of the triangulation. The coefficients α and β are independent of the triangulation. This bound is turned into an element-wise error indicator function $E(K)$ for element K by summing the contributions from its edges. The final form for the equation

$$-\nabla \cdot (c \nabla u) + au = f$$

becomes

$$E(K) = \alpha \| h(f - au) \|_K + \beta \left(\frac{1}{2} \sum_{\tau \in \partial K} h_\tau^2 [\mathbf{n}_r \cdot c \nabla u_h]^2 \right)^{1/2}$$

where $\mathbf{n}_\tau$ is the unit normal of edge τ and the braced term is the jump in flux across the element edge. The L_2 norm is computed over the element K. This error indicator is computed by the `pdejmps` function.

The Mesh Refiner

Partial Differential Equation Toolbox software is geared to elliptic problems. For reasons of accuracy and ill-conditioning, they require the elements not to deviate too much from being equilateral. Thus, even at essentially one-dimensional solution features, such as boundary layers, the refinement technique must guarantee reasonably shaped triangles.

When an element is refined, new nodes appear on its midsides, and if the neighbor triangle is not refined in a similar way, it is said to have *hanging nodes*. The final triangulation must have no hanging nodes, and they are removed by splitting neighbor triangles. To avoid further deterioration of triangle quality in successive generations, the "longest edge bisection" scheme Rosenberg-Stenger [8] is used, in which the longest side of a triangle is always split, whenever any of the sides have hanging nodes. This guarantees that no angle is ever smaller than half the smallest angle of the original triangulation.

Two selection criteria can be used. One, `pdeadworst`, refines all elements with value of the error indicator larger than half the worst of any element. The other, `pdeadgsc`, refines all elements with an indicator value exceeding a user-defined dimensionless tolerance. The comparison with the tolerance is properly scaled with respect to domain and solution size, etc.

The Termination Criteria

For smooth solutions, error equidistribution can be achieved by the `pdeadgsc` selection if the maximum number of elements is large enough. The `pdeadworst` adaptation only terminates when the maximum number of elements has been exceeded. This mode is natural when the solution exhibits singularities. The error indicator of the elements next to the singularity may never vanish, regardless of element size, and equidistribution is too much to hope for.

Fast Solution of Poisson's Equation

While the general strategy of Partial Differential Equation Toolbox software is to use the MATLAB built-in solvers for sparse systems, there are situations where faster solution algorithms are available. One such example is found when solving Poisson's equation

$$-\Delta u = f \text{ in } \Omega$$

with Dirichlet boundary conditions, where Ω is a rectangle.

For the fast solution algorithms to work, the mesh on the rectangle must be a *regular mesh*. In this context it means that the first side of the rectangle is divided into N_1 segments of length h_1, the second into N_2 segments of length h_2, and $(N_1 + 1)$ by $(N_2 + 1)$ points are introduced on the regular grid thus defined. The triangles are all congruent with sides h_1, h_2 and a right angle in between.

The Dirichlet boundary conditions are eliminated in the usual way, and the resulting problem for the interior nodes is $Kv = F$. If the interior nodes are numbered from left to right, and then from bottom to top, the K matrix is block tridiagonal. The $N_2 - 1$ diagonal blocks, here called T, are themselves tridiagonal $(N_1 - 1)$ by $(N_1 - 1)$ matrices, with $2(h_1/h_2 + h_2/h_1)$ on the diagonal and $-h_2/h_1$ on the subdiagonals. The subdiagonal blocks, here called I, are $-h_1/h_2$ times the unit $N_1 - 1$ matrix.

The key to the solution of the problem $Kv = F$ is that the problem $Tw = f$ is possible to solve using the *discrete sine transform*. Let S be the $(N_1 - 1)$ by $(N_1 - 1)$ matrix with $S\text{ij} = \sin(\pi ij/N_1)$. Then $S^{-1}TS = \Lambda$, where Λ is a diagonal matrix with diagonal entries $2(h_1/h_2 + h_2/h_1) - 2h_2/h_1 \cos(\pi i/N_1)$. $w = S\Lambda^{-1}S^{-1} f$, but multiplying with S is nothing more than taking the discrete sine transform, and multiplying with S^{-1} is the same as taking the inverse discrete sine transform. The discrete sine transform can be efficiently calculated using the fast Fourier transform on a sequence of length $2N_1$.

Solving $Tw = f$ using the discrete sine transform would not be an advantage in itself, since the system is tridiagonal and should be solved as such. However, for the full system $Kv = F$, a transformation of the blocks in K turns it into $N_1 - 1$ decoupled tridiagonal systems of size $N_2 - 1$. Thus, a solution algorithm would look like

1. Divide F into N_2 - 1 blocks of length N_1 - 1, and perform an inverse discrete sine transform on each block.

2. Reorder the elements and solve N_1 - 1 tridiagonal systems of size N_2 - 1, with $2(h_1/h_2 + h_2/h_1) - 2h_2/h_1 \cos(\pi i/N_1)$ on the diagonal, and $-h_1/h_2$ on the subdiagonals.

3. Reverse the reordering, and perform N_2 - 1 discrete sine transforms on the blocks of length N_1 - 1.

When using a fast solver such as this one, time and memory are also saved since the matrix K in fact never has to be assembled. A drawback is that since the mesh has to be regular, it is impossible to do adaptive mesh refinement.

The fast elliptic solver for Poisson's equation is implemented in `poisolv`. The discrete sine transform and the inverse discrete sine transform are computed by `dst` and `idst`, respectively.

References

[1] Bank, Randolph E., *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations*, User's Guide 6.0, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.

[2] Dahlquist, Germund, and Björk, Åke, *Numerical Methods*, 2nd edition, 1995, in print.

[3] Golub, Gene H., and Charles F. Van Loan, *Matrix Computations*, 2nd edition, John Hopkins University Press, Baltimore, MD, 1989.

[4] George, P.L., *Automatic Mesh Generation — Application to Finite Element Methods*, Wiley, 1991.

[5] Johnson, C., *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Studentlitteratur, Lund, Sweden, 1987.

[6] Johnson, C., and Eriksson, K., *Adaptive Finite Element Methods for Parabolic Problems I: A Linear Model Problem*, SIAM J. Numer. Anal, 28, (1991), pp. 43–77.

[7] Saad, Yousef, *Variations on Arnoldi's Method for Computing Eigenelements of Large Unsymmetric Matrices*, Linear Algebra and its Applications, Vol 34, 1980, pp. 269–295.

[8] Rosenberg, I.G., and F. Stenger, *A lower bound on the angles of triangles constructed by bisecting the longest side*, Math. Comp. 29 (1975), pp 390–395.

[9] Strang, Gilbert, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Cambridge, MA, 1986.

[10] Strang, Gilbert, and Fix, George, *An Analysis of the Finite Element Method*, Prentice-Hall Englewoood Cliffs, N.J., USA, 1973.

4

Function Reference

PDE Algorithms (p. 4-1)	Solve various PDE problems
User-Interface Algorithms (p. 4-2)	Draw shapes and open the Partial Differential Equation Toolbox GUI
Geometry Algorithms (p. 4-2)	Specify geometry of PDE problems
Plots (p. 4-3)	Display PDE solutions
Utility Algorithms (p. 4-3)	Compute intermediate results
User-Defined Algorithms (p. 4-4)	Specify boundary conditions and geometry of PDE problems

PDE Algorithms

`adaptmesh`	Adaptive mesh generation and PDE solution
`assema`	Assemble area integral contributions
`assemb`	Assemble boundary condition contributions
`assempde`	Assemble stiffness matrix and right side of PDE problem
`hyperbolic`	Solve hyperbolic PDE problem
`parabolic`	Solve parabolic PDE problem
`pdeeig`	Solve eigenvalue PDE problem

`pdenonlin`	Solve nonlinear PDE problem
`poisolv`	Fast solution of Poisson's equation on rectangular grid

User-Interface Algorithms

`pdecirc`	Draw circle
`pdeellip`	Draw ellipse
`pdemdlcv`	Convert Partial Differential Equation Toolbox 1.0 model files to 1.0.2 format
`pdepoly`	Draw polygon
`pderect`	Draw rectangle
`pdetool`	Open GUI

Geometry Algorithms

`csgchk`	Check validity of Geometry Description matrix
`csgdel`	Delete borders between minimal regions
`decsg`	Decompose Constructive Solid Geometry into minimal regions
`initmesh`	Create initial triangular mesh
`jigglemesh`	Jiggle internal points of triangular mesh
`pdearcl`	Interpolation between parametric representation and arc length

`poimesh`	Make regular mesh on rectangular geometry
`refinemesh`	Refine triangular mesh
`wbound`	Write boundary condition specification file
`wgeom`	Write geometry specification function

Plots

`pdecont`	Shorthand command for contour plot
`pdegplot`	Plot PDE geometry
`pdemesh`	Plot PDE triangular mesh
`pdeplot`	Generic plot function
`pdesurf`	Shorthand command for surface plot

Utility Algorithms

`dst, idst`	Discrete sine transform
`pdeadgsc`	Select triangles using relative tolerance criterion
`pdeadworst`	Select triangles relative to worst value
`pdecgrad`	Flux of PDE solution
`pdeent`	Indices of triangles neighboring given set of triangles
`pdegrad`	Gradient of PDE solution

`pdeintrp`	Interpolate from node data to triangle midpoint data
`pdejmps`	Error estimates for adaptation
`pdeprtni`	Interpolate from triangle midpoint data to node data
`pdesdp, pdesde, pdesdt`	Indices of points/edges/triangles in set of subdomains
`pdesmech`	Calculate structural mechanics tensor functions
`pdetrg`	Triangle geometry data
`pdetriq`	Triangle quality measure
`poiasma`	Boundary point matrix contributions for fast solvers of Poisson's equation
`poicalc`	Fast solver for Poisson's equation on rectangular grid
`poiindex`	Indices of points in canonical ordering for rectangular grid
`sptarn`	Solve generalized sparse eigenvalue problem
`tri2grid`	Interpolate from PDE triangular mesh to rectangular grid

User-Defined Algorithms

`pdebound`	Boundary file
`pdegeom`	Geometry file

Functions — Alphabetical List

adaptmesh

Purpose Adaptive mesh generation and PDE solution

Syntax

```
[u,p,e,t]=adaptmesh(g,b,c,a,f)
[u,p,e,t]=adaptmesh(g,b,c,a,f,'PropertyName',PropertyValue,)
```

Description

```
[u,p,e,t]=adaptmesh(g,b,c,a,f)
[u,p,e,t]=adaptmesh(g,b,c,a,f,'PropertyName',PropertyValue,)
```

performs adaptive mesh generation and PDE solution. Optional arguments are given as property name/property value pairs.

The function produces a solution u to the elliptic scalar PDE problem

$$-\nabla \cdot (c\nabla u) + au = f \text{ on } \Omega$$

or the elliptic system PDE problem

$$-\nabla \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u}) + \underline{a}\mathbf{u} = \mathbf{f} \quad on \quad \Omega$$

with the problem geometry and boundary conditions given by `g` and `b`. The mesh is described by the `p`, `e`, and `t`.

The solution u is represented as the solution vector `u`. For details on the representation of the solution vector, see `assempde`.

The algorithm works by solving a sequence of PDE problems using refined triangular meshes. The first triangular mesh generation is obtained either as an optional argument to `adaptmesh` or by a call to `initmesh` without options. The following generations of triangular meshes are obtained by solving the PDE problem, computing an error estimate, selecting a set of triangles based on the error estimate, and then finally refining these triangles. The solution to the PDE problem is then recomputed. The loop continues until no triangles are selected by the triangle selection method, or until the maximum number of triangles is attained, or until the maximum number of triangle generations has been generated.

`g` describes the decomposed geometry of the PDE problem. `g` can either be a Decomposed Geometry matrix or the name of a Geometry file. The

formats of the Decomposed Geometry matrix and Geometry file are described in the entries on decsg and pdegeom, respectively.

b describes the boundary conditions of the PDE problem. b can be either a Boundary Condition matrix or the name of a Boundary file. The formats of the Boundary Condition matrix and Boundary file are described in the entries on assemb and pdebound, respectively.

The adapted triangular mesh of the PDE problem is given by the mesh data p, e, and t. For details on the mesh data representation, see initmesh.

The coefficients c, a, and f of the PDE problem can be given in a wide variety of ways. In the context of adaptmesh the coefficients can depend on u if the nonlinear solver is enabled using the property nonlin. The coefficients cannot depend on t, the time. For a complete listing of all options, see assempde.

The following table lists the property name/property value pairs, their default values, and descriptions of the properties.

Property	Property	Default	Description
Maxt	positive integer	inf	Maximum number of new triangles
Ngen	positive integer	10	Maximum number of triangle generations
Mesh	p1, e1, t1	initmesh	Initial mesh
Tripick	MATLAB function	pdeadworst	Triangle selection method
Par	numeric	0.5	Function parameter
Rmethod	longest|regular	longest	Triangle refinement method
Nonlin	on|off	off	Use nonlinear solver
Toln	numeric	1e-4	Nonlinear tolerance
Init	u0	0	Nonlinear initial value

Property	Property	Default	Description
Jac	fixed|lumped|full	fixed	Nonlinear Jacobian calculation
norm	numeric|inf|energy	inf	Nonlinear residual norm

Par is passed to the Tripick function, which is described later. Normally it is used as tolerance of how well the solution fits the equation.

No more than Ngen successive refinements are attempted. Refinement is also stopped when the number of triangles in the mesh exceeds Maxt.

p1, e1, and t1 are the input mesh data. This triangular mesh is used as starting mesh for the adaptive algorithm. For details on the mesh data representation, see initmesh. If no initial mesh is provided, the result of a call to initmesh with no options is used as the initial mesh.

The triangle selection method, Tripick, is a user-definable triangle selection method. Given the error estimate computed by the function pdejmps, the triangle selection method selects the triangles to be refined in the next triangle generation. The function is called using the arguments p, t, cc, aa, ff, u, errf, and par. p and t represent the current generation of triangles, cc, aa, and ff are the current coefficients for the PDE problem, expanded to triangle midpoints, u is the current solution, errf is the computed error estimate, and par, the function parameter, given to adaptmesh as optional argument. The matrices cc, aa, ff, and errf all have *N*t columns, where *N*t is the current number of triangles. The number of rows in cc, aa, and ff are exactly the same as the input arguments c, a, and f. errf has one row for each equation in the system. There are two standard triangle selection methods—pdeadworst and pdeadgsc. pdeadworst selects triangles where errf exceeds a fraction (default: 0.5) of the worst value, and pdeadgsc selects triangles using a relative tolerance criterion.

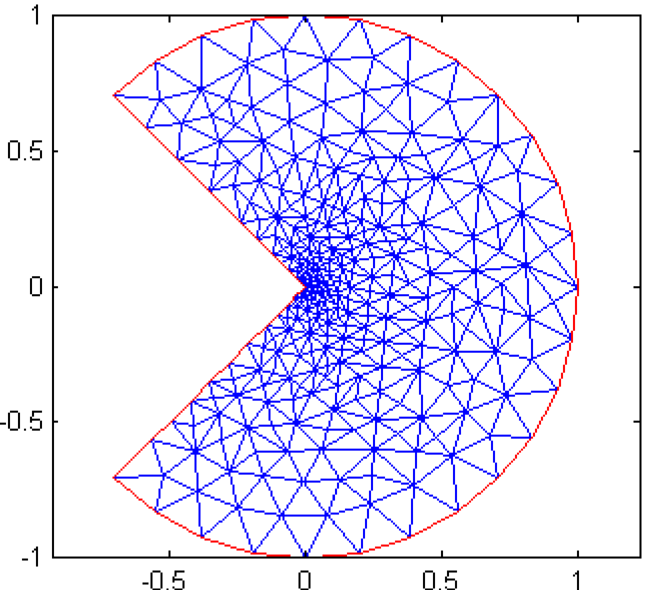
The refinement method is either longest or regular. For details on the refinement method, see refinemesh.

The adaptive algorithm can also solve nonlinear PDE problems. For nonlinear PDE problems, the `Nonlin` parameter must be set to `on`. The nonlinear tolerance `Toln`, nonlinear initial value `u0`, nonlinear Jacobian calculation `Jac`, and nonlinear residual norm `Norm` are passed to the nonlinear solver `pdenonlin`. For details on the nonlinear solver, see `pdenonlin`.

Examples

Solve the Laplace equation over a circle sector, with Dirichlet boundary conditions $u = \cos(2/3\text{atan2}(y,x))$ along the arc, and $u = 0$ along the straight lines, and compare to the exact solution. We refine the triangles using the worst error criterion until we obtain a mesh with at least 500 triangles:

```
[u,p,e,t]=adaptmesh('cirsg','cirsb',1,0,0,'maxt',500,...
                       'tripick','pdeadworst','ngen',inf);
x=p(1,:); y=p(2,:);
exact=((x.^2+y.^2).^(1/3).*cos(2/3*atan2(y,x)))';
max(abs(u-exact))
ans =
    0.0028
size(t,2)
ans =
    629
pdemesh(p,e,t)
```

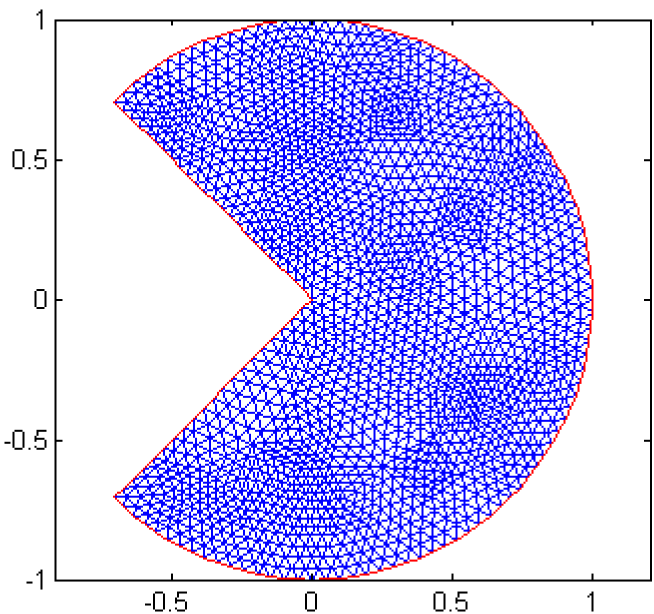


The maximum absolute error is 0.0028, with 629 triangles.

We test how many refinements we have to use with a uniform triangle net:

```
[p,e,t]=initmesh('cirsg');
[p,e,t]=refinemesh('cirsg',p,e,t);
u=assempde('cirsb',p,e,t,1,0,0);
x=p(1,:); y=p(2,:);
exact=((x.^2+y.^2).^(1/3).*cos(2/3*atan2(y,x)))';
max(abs(u-exact))
ans =
       0.0121
size(t,2)
```

```
ans =
       788
[p,e,t]=refinemesh('cirsg',p,e,t);
u=assempde('cirsb',p,e,t,1,0,0);
x=p(1,:); y=p(2,:);
exact=((x.^2+y.^2).^(1/3).*cos(2/3*atan2(y,x)))';
max(abs(u-exact))
ans =
       0.0078
size(t,2)
ans =
       3152
pdemesh(p,e,t)
```



Uniform refinement with 3152 triangles produces an error of 0.0078. This error is over three times as large as that produced by the adaptive method (0.0028) with many fewer triangles (629). For a problem with regular solution, we expect a $O(h^2)$ error, but this solution is singular since $u \approx r^{1/3}$ at the origin.

Diagnostics

Upon termination, one of the following messages is displayed:

- `Adaption completed` (This means that the `Tripick` function returned zero triangles to refine.)
- `Maximum number of triangles obtained`
- `Maximum number of refinement passes obtained`

See Also

`assempde`	Partial Differential Equation Toolbox
`initmesh`	Partial Differential Equation Toolbox
`pdeadgsc`	Partial Differential Equation Toolbox
`pdeadworst`	Partial Differential Equation Toolbox
`pdejmps`	Partial Differential Equation Toolbox
`refinemesh`	Partial Differential Equation Toolbox

Purpose

Assemble area integral contributions

Syntax

```
[K,M,F]=assema(p,t,c,a,f)
[K,M,F]=assema(p,t,c,a,f,u0)
[K,M,F]=assema(p,t,c,a,f,u0,time)
[K,M,F]=assema(p,t,c,a,f,u0,time,sdl)
[K,M,F]=assema(p,t,c,a,f,time)
[K,M,F]=assema(p,t,c,a,f,time,sdl)
```

Description

`[K,M,F]=assema(p,t,c,a,f)` assembles the stiffness matrix `K`, the mass matrix `M`, and the right-hand side vector `F`.

The input parameters `p`, `t`, `c`, `a`, `f`, `u0`, `time`, and `sdl` have the same meaning as in `assempde`.

See Also

`assempde`	Partial Differential Equation Toolbox

assemb

Purpose

Assemble boundary condition contributions

Syntax

```
[Q,G,H,R]=assemb(b,p,e)
[Q,G,H,R]=assemb(b,p,e,u0)
[Q,G,H,R]=assemb(b,p,e,u0,time)
[Q,G,H,R]=assemb(b,p,e,u0,time,sdl)
[Q,G,H,R]=assemb(b,p,e,time)
[Q,G,H,R]=assemb(b,p,e,time,sdl)
```

Description

`[Q,G,H,R]=assemb(b,p,e)` assembles the matrices `Q` and `H`, and the vectors `G` and `R`. `Q` should be added to the system matrix and contains contributions from mixed boundary conditions. `G` should be added to the right side and contains contributions from generalized Neumann and mixed boundary conditions. The equation `H*u=R` represents the Dirichlet type boundary conditions.

The input parameters `p`, `e`, `u0`, `time`, and `sdl` have the same meaning as in `assempde`.

`b` describes the boundary conditions of the PDE problem. `b` can be either a *Boundary Condition matrix* or the name of a Boundary file. The format of the Boundary Condition matrix is described later.

Partial Differential Equation Toolbox software treats the following boundary condition types:

- On a generalized Neumann boundary segment, q and g are related to the normal derivative value by:

$$\vec{n} \cdot (c \otimes \nabla u) + \underline{q}u = g$$

- On a Dirichlet boundary segment, $hu = r$.

The software can also handle systems of partial differential equations over the domain Ω. Let the number of variables in the system be N. The general boundary condition is $\underline{h}u = \mathbf{r}$.

$$\vec{n} \cdot (c \otimes \nabla u) + \underline{q}u = g + h'u$$

The notation $\vec{n} \cdot (c \otimes \nabla u)$ indicates that the N by 1 matrix with (i,1)-component

$$\sum_{j=1}^{N}\left(\cos(\alpha)c_{i,j,1,1}\frac{\partial}{\partial x}+\cos(\alpha)c_{i,j,1,2}\frac{\partial}{\partial y}+\sin(\alpha)c_{i,j,2,1}\frac{\partial}{\partial x}+\sin(\alpha)c_{i,j,2,2}\frac{\partial}{\partial y}\right)u$$

where α is the angle of the normal vector of the boundary, pointing in the direction out from Ω, the domain.

The Boundary Condition matrix is created internally in `pdetool` (actually a function called by `pdetool`) and then used from the function `assemb` for assembling the contributions from the boundary to the matrices `Q`, `G`, `H`, and `R`. The Boundary Condition matrix can also be saved onto a file as a boundary file for later use with the `wbound` function.

For each column in the Decomposed Geometry matrix there must be a corresponding column in the Boundary Condition matrix. The format of each column is according to the following list:

- Row one contains the dimension N of the system.
- Row two contains the number M of Dirichlet boundary conditions.
- Row three to $3 + N^2 - 1$ contain the lengths for the strings representing $\underline{q}$. The lengths are stored in column-wise order with respect to $\underline{q}$.
- Row $3 + N^2$ to $3 + N^2 + N - 1$ contain the lengths for the strings representing $\mathbf{g}$.
- Row $3 + N^2 + N$ to $3 + N^2 + N + MN - 1$ contain the lengths for the strings representing h. The lengths are stored in columnwise order with respect to $\underline{h}$.
- Row $3 + N^2 + N + NM$ to $3 + N^2 + N + MN + M - 1$ contain the lengths for the strings representing $\mathbf{r}$.

The following rows contain text expressions representing the actual boundary condition functions. The text strings have the lengths according to above. The MATLAB text expressions are stored in columnwise order with respect to matrices h and q. There are no separation characters between the strings. You can insert MATLAB expressions containing the following variables:

- The 2-D coordinates `x` and `y`.
- A boundary segment parameter `s`, proportional to arc length. `s` is 0 at the start of the boundary segment and increases to 1 along the boundary segment in the direction indicated by the arrow.
- The outward normal vector components `nx` and `ny`. If you need the tangential vector, it can be expressed using `nx` and `ny` since $t_x = -n_y$ and $t_y = n_x$.
- The solution `u` (only if the input argument `u` has been specified).
- The time `t` (only if the input argument `time` has been specified).

It is not possible to explicitly refer to the time derivative of the solution in the boundary conditions.

Example 1

The following examples describe the format of the boundary condition matrix for one column of the Decomposed Geometry matrix. For a boundary in a scalar PDE ($N = 1$) with Neumann boundary condition ($M = 0$)

$$\vec{n} \cdot (c\nabla u) = -x^2$$

the boundary condition would be represented by the column vector

```
[1 0 1 5 '0' '-x.^2']'
```

No lengths are stored for h or r.

Also for a scalar PDE, the Dirichlet boundary condition

$u = x^2\text{-}y^2$

is stored in the column vector

```
[1 1 1 1 1 9 '0' '0' '1' 'x.^2-y.^2']'
```

For a system ($N = 2$) with mixed boundary conditions ($M = 1$):

$$(h_{11}\ \ h_{12})u = r_1$$

$$\vec{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \begin{pmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{pmatrix} \mathbf{u} = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} + \mathbf{s}$$

the column appears similar to the following example:

```
2
1
lq11
lq21
lq12
lq22
lg1
lg2
lh11
lh12
lr1
q11 ...
q21 ...
q12 ...
q22 ...
g1 ...
g2 ...
h11 ...
h12 ...
r1 ...
```

Where `lq11`, `lq21`, . . . denote lengths of the MATLAB text expressions, and `q11`, `q21`, . . . denote the actual expressions.

You can easily create your own examples by trying out `pdetool`. Enter boundary conditions by double-clicking on boundaries in boundary mode, and then export the Boundary Condition matrix to the MATLAB workspace by selecting the **Export Decomposed Geometry, Boundary Cond's** option from the **Boundary** menu.

Example 2

The following example shows you how to find the boundary condition matrices for the Dirichlet boundary condition $u = x^2 - y^2$ on the boundary of a circular disk.

1 Create the following function in your working folder:

```
function [x,y]=circ_geom(bs,s)
%CIRC_GEOM Creates a geometry file for a unit circle.

% Number of boundary segments
nbs=4;

if nargin==0 % Number of boundary segments
    x=nbs;
elseif nargin==1 % Create 4 boundary segments
    dl=[0     pi/2  pi      3*pi/2
        pi/2  pi    3*pi/2  2*pi
        1     1     1       1
        0     0     0       0];
    x=dl(:,bs);

else % Coordinates of edge segment points
    z=exp(i*s);
    x=real(z);
    y=imag(z);
end
```

2 Create a second function in your working folder that finds the boundary condition matrices, `Q`, `G`, `H`, and `R`:

```
function assemb_example
```

```
% Use ASSEMB to find the boundary condition matrices.

% Describe the geometry using four boundary segments
figure(1)
pdegplot('circ_geom')
axis equal

% Initialize the mesh
[p,e,t]=initmesh('circ_geom','Hmax',0.4);
figure(2)

% Plot the mesh
pdemesh(p,e,t)
axis equal

% Define the boundary condition vector, b,
% for the boundary condition u=x^2-y^2.
% For each boundary segment, the boundary
% condition vector is
b=[1 1 1 1 1 9 '0' '0' '1' 'x.^2-y.^2']';

% Create a boundary condition matrix that
% represents all of the boundary segments.
b = repmat(b,1,4);

% Use ASSEMB to find the boundary condition
% matrices. Since there are only Dirichlet
% boundary conditions, Q and G are empty.
[Q,G,H,R]=assemb(b,p,e)
```

3 Run the function `assemb_example.m`.

The function returns the four boundary condition matrices.

```
Q =

   All zero sparse: 41-by-41
```

```
G =

   All zero sparse: 41-by-1

H =

    (1,1)        1
    (2,2)        1
    (3,3)        1
    (4,4)        1
    (5,5)        1
    (6,6)        1
    (7,7)        1
    (8,8)        1
    (9,9)        1
   (10,10)       1
   (11,11)       1
   (12,12)       1
   (13,13)       1
   (14,14)       1
   (15,15)       1
   (16,16)       1

R =

    (1,1)       1.0000
    (2,1)      -1.0000
    (3,1)       1.0000
    (4,1)      -1.0000
    (5,1)       0.0000
    (6,1)      -0.0000
    (7,1)       0.0000
    (8,1)      -0.0000
    (9,1)       0.7071
   (10,1)      -0.7071
   (11,1)      -0.7071
```

```
(12,1)        0.7071
(13,1)        0.7071
(14,1)       -0.7071
(15,1)       -0.7071
(16,1)        0.7071
```

`Q` and `G` are all zero sparse matrices because the problem has only Dirichlet boundary conditions and neither generalized Neumann nor mixed boundary conditions apply.

See Also

`assempde`	Partial Differential Equation Toolbox
`pdebound`	Partial Differential Equation Toolbox

assempde

Purpose

Assemble stiffness matrix and right side of PDE problem

Syntax

```
u=assempde(b,p,e,t,c,a,f)
u=assempde(b,p,e,t,c,a,f,u0)
u=assempde(b,p,e,t,c,a,f,u0,time)
u=assempde(b,p,e,t,c,a,f,time)
[K,F]=assempde(b,p,e,t,c,a,f)
[K,F]=assempde(b,p,e,t,c,a,f,u0)
[K,F]=assempde(b,p,e,t,c,a,f,u0,time)
[K,F]=assempde(b,p,e,t,c,a,f,u0,time,sdl)
[K,F]=assempde(b,p,e,t,c,a,f,time)
[K,F]=assempde(b,p,e,t,c,a,f,time,sdl)
[K,F,B,ud]=assempde(b,p,e,t,c,a,f)
[K,F,B,ud]=assempde(b,p,e,t,c,a,f,u0)
[K,F,B,ud]=assempde(b,p,e,t,c,a,f,u0,time)
[K,F,B,ud]=assempde(b,p,e,t,c,a,f,time)
[K,M,F,Q,G,H,R]=assempde(b,p,e,t,c,a,f)
[K,M,F,Q,G,H,R]=assempde(b,p,e,t,c,a,f,u0)
[K,M,F,Q,G,H,R]=assempde(b,p,e,t,c,a,f,u0,time)
[K,M,F,Q,G,H,R]=assempde(b,p,e,t,c,a,f,u0,time,sdl)
[K,M,F,Q,G,H,R]=assempde(b,p,e,t,c,a,f,time)
[K,M,F,Q,G,H,R]=assempde(b,p,e,t,c,a,f,time,sdl)
u=assempde(K,M,F,Q,G,H,R)
[K1,F1]=assempde(K,M,F,Q,G,H,R)
[K1,F1,B,ud]=assempde(K,M,F,Q,G,H,R)
```

Description

`assempde` is the basic Partial Differential Equation Toolbox function. It assembles a PDE problem by using the FEM formulation described in Chapter 3, “Finite Element Method”. The command `assempde` assembles the scalar PDE problem

$$-\nabla \cdot (c\nabla u) + au = f \text{ on } \Omega$$

or the system PDE problem

$$-\nabla \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u}) + \underline{a}\mathbf{u} = \mathbf{f} \quad on \quad \Omega$$

The command can optionally produce a solution to the PDE problem.

For the scalar case the *solution vector u* is represented as a column vector of solution values at the corresponding node points from `p`. For a system of dimension N with n_p node points, the first n_p values of `u` describe the first component of u, the following n_p values of `u` describe the second component of u, and so on. Thus, the components of u are placed in the vector `u` as N blocks of node point values.

`u=assempde(b,p,e,t,c,a,f)` assembles and solves the PDE problem by eliminating the Dirichlet boundary conditions from the system of linear equations.

`[K,F]=assempde(b,p,e,t,c,a,f)` assembles the PDE problem by approximating the Dirichlet boundary condition with stiff springs (see "The Elliptic System" on page 3-10 for details). `K` and `F` are the stiffness matrix and right-hand side, respectively. The solution to the FEM formulation of the PDE problem is `u=K\F`.

`[K,F,B,ud]=assempde(b,p,e,t,c,a,f)` assembles the PDE problem by eliminating the Dirichlet boundary conditions from the system of linear equations. `u1=K\F` returns the solution on the non-Dirichlet points. The solution to the full PDE problem can be obtained as the MATLAB expression `u=B*u1+ud`.

`[K,M,F,Q,G,H,R]=assempde(b,p,e,t,c,a,f)` gives a split representation of the PDE problem.

`u=assempde(K,M,F,Q,G,H,R)` collapses the split representation into the single matrix/vector form, and then solves the PDE problem by eliminating the Dirichlet boundary conditions from the system of linear equations.

`[K1,F1]=assempde(K,M,F,Q,G,H,R)` collapses the split representation into the single matrix/vector form, by fixing the Dirichlet boundary condition with large spring constants.

`[K1,F1,B,ud]=assempde(K,M,F,Q,G,H,R)` collapses the split representation into the single matrix/vector form by eliminating the Dirichlet boundary conditions from the system of linear equations.

b describes the boundary conditions of the PDE problem. b can be either a Boundary Condition matrix or the name of a Boundary file. The formats of the Boundary Condition matrix and Boundary file are described in the entries on assemb and pdebound, respectively.

The geometry of the PDE problem is given by the mesh data p, e, and t. For details on the mesh data representation, see initmesh.

The optional list of subdomain labels, sdl, restricts the assembly process to the subdomains denoted by the labels in the list. The optional input arguments u0 and time are used for the nonlinear solver and time stepping algorithms, respectively. The tentative input solution vector u0 has the same format as u.

PDE Coefficients for Scalar Case

The coefficients c, a, and f in the scalar PDE problem can be represented in the MATLAB variables c, a, and f in the following ways:

- A constant.
- A row vector of values at the triangle centers of mass.
- A MATLAB text expression for computing coefficient values at the triangle centers of mass. The expression is evaluated in a context where the variables x, y, sd, u, ux, uy, and t are row vectors representing values at the triangle centers of mass (t is a scalar). The row vectors contain x- and y-coordinates, subdomain label, solution, x and y derivatives of the solution, and time. u, ux, and uy can only be used if u0 has been passed to assempde. The same applies to the scalar t, which is passed to assempde as time.
- A sequence of MATLAB text expressions separated by exclamation marks !. The syntax of each of the text expressions must be according to the preceding item. The number of expressions in the sequence must equal the number of subdomain labels in the triangle list t. (This number can be checked by typing max(t(4,:)).)
- The name of a user-defined MATLAB function that accepts the arguments (p,t,u,t0). u and time are empty matrices if the corresponding parameter is not passed to assempde. p and t are

mesh data, `u` is the `u0` input argument, and `t0` is the time input to `assempde`. If time is `NaN` and the function depends on time, the function must return a matrix of correct size, containing `NaN`s in all positions.

We refer to the preceding matrices as *coefficient matrix*, and the user-defined MATLAB function as *coefficient file*.

If `c` contains two rows with data according to any of the preceding items, they are the $c_{1,1}$, and $c_{2,2}$, elements of the 2-by-2 diagonal matrix

$$\begin{pmatrix} c_{1,1} & 0 \\ 0 & c_{2,2} \end{pmatrix}$$

If `c` contains four rows, they are the $c_{1,1}$, $c_{2,1}$, $c_{1,2}$, and $c_{2,2}$ elements of a 2-by-2 matrix.

PDE Coefficients for System Case

Let N be the dimension of the PDE system. Now $\underline{\mathbf{c}}$ is an N-by-N-by-2-by-2 tensor, $\underline{a}$ an N-by-N-matrix, and $\mathbf{f}$ a column vector of length N. The elements c_{ijkl}, a_{ij}, d_{ij}, and f_i of $\underline{\mathbf{c}}$, $\underline{a}$, $\underline{d}$, and $\mathbf{f}$ are stored row-wise in the MATLAB matrices `c`, `a`, `d`, and `f`. Each row in these matrices is similar in syntax to the scalar case. There is one difference, however: At the point of evaluation of MATLAB text expressions, the variables `u`, `ux`, and `uy` contain matrices with N rows, one row for each component. The cases of identity, diagonal, and symmetric matrices are handled as special cases. For the tensor c_{ijkl} this applies both to the indices i and j, and to the indices k and l.

The number of rows in `f` determines the dimension N of the system. Row i in `f` represents the component f_i in $\mathbf{f}$.

The number of rows n_a in `a` is related to the components a_{ij} of a according to the following table. For the symmetric case assume that $j \geq i$. All elements a_{ij} that cannot be formed are assumed to be zero.

n_a	Symmetric	a_{ij}	Row in a
1	No	a_{ii}	1
N	No	a_{ii}	i
$N(N+1)/2$	Yes	a_{ij}	$j(j-1)/2+i$
$N2$	No	a_{ij}	$N(j-1)+i$

The coding of $\underline{\mathbf{c}}$ in `c` is determined by the dimension N and the number of rows nc in `c`. The number of rows nc in `c` is matched to the function of N in the first column in the following table—sequentially from the first line to the last line. The first match determines the type of coding of $\underline{\mathbf{c}}$. This actually means that for some small values, $2 \le N \le 4$, the coding is only determined by the order in which the tests are performed. For the symmetric case assume that $j \ge i$, and $l \ge k$. All elements c_{ijkl} that can not be formed are assumed to be zero.

n_c	Symmetric	c_{ijkl}	Row in c
1	No	c_{iikk}	1
2	No	c_{iikk}	k
3	Yes	c_{iikl}	$l+k-1$
4	No	c_{iikl}	$2l+k-2$
N	No	c_{iikk}	i
2N	No	c_{iikk}	$2i+k-2$
3N	Yes	c_{iikl}	$3i+l+k-4$
4N	No	c_{iikl}	$4i+2l+k-6$
$2N(2N+1)/2$	Yes	c_{iikl}	$2i^2+i+l+k-4$
		c_{ijkl}, $i<j$	$2j^2-3j+4i+2l+k-5$
$4N2$	No	c_{ijkl}	$4N(j-1)+4i+2l+k-6$

Notice

You can use MATLAB functions in expressions for boundary conditions and PDE coefficients. For example, the c coefficient can be a string containing the MATLAB function call `fun(x,y)` that interpolates measured data to the coordinates x and y. The function must return a matrix of exactly the same size as x or y, and should contain the interpolated data in the corresponding point.

Examples

Example 1

Solve the equation $-\Delta u = 1$ on the geometry defined by the L-shaped membrane. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$. Finally plot the solution.

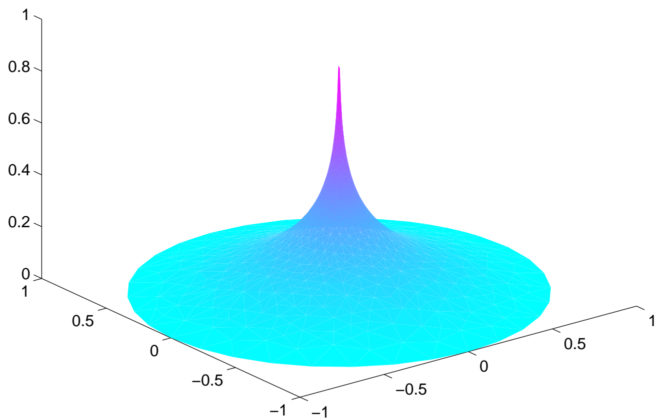
```
[p,e,t]=initmesh('lshapeg','Hmax',0.2);
[p,e,t]=refinemesh('lshapeg',p,e,t);
u=assempde('lshapeb',p,e,t,1,0,1);
pdesurf(p,t,u)
```

Example 2

Consider Poisson's equation on the unit circle with unit point source at the origin. The exact solution

$$u = -\frac{1}{2\pi}\log(r)$$

is known for this problem. We define the function `f=circlef(p,t,u,time)` for computing the right-hand side. `circlef` returns zero for all triangles except for the one located at the origin; for that triangle it returns $1/a$, where a is the triangle area. `pdedemo7` performs a full demonstration of the problem with adaptive solution.



Example 3

We study how the matrices $\underline{a}$ (and also $\underline{d}$) are stored in the MATLAB matrix a for system case $N = 3$.

$n_a = 1$:	a(1)	0	0	$n_a = 3$:	a(1)	0	0
	0	a(1)	0		0	a(2)	0
	0	0	a(1)		0	0	a(3)

The bullet symbol (•) means that the matrix is symmetric.

$n_a = 6$:	a(1)	a(2)	a(4)	$n_a = 9$:	a(1)	a(4)	a(7)
	•	a(3)	a(5)		a(2)	a(5)	a(8)
	•	•	a(6)		a(3)	a(6)	a(9)

We study how the tensor $\underline{\mathbf{c}}$ is stored in the MATLAB matrix c for the system case. N=3

$n_c = 1$:	c(1)	0	0	0	0	0
	0	c(1)	0	0	0	0
	0	0	c(1)	0	0	0
	0	0	0	c(1)	0	0
	0	0	0	0	c(1)	0
	0	0	0	0	0	c(1)

$n_c = 2$:	c(1)	0	0	0	0	0
	0	c(2)	0	0	0	0
	0	0	c(1)	0	0	0
	0	0	0	c(2)	0	0
	0	0	0	0	c(1)	0
	0	0	0	0	0	c(2)

The bullet symbol (•) means that the matrix is symmetric.

$n_c = 3$:	c(1)	c(2)	0	0	0	0
	•	c(3)	0	0	0	0
	0	0	c(1)	c(2)	0	0
	0	0	•	c(3)	0	0
	0	0	0	0	c(1)	c(2)
	0	0	0	0	•	c(3)

$n_c = 4$:						
	c(1)	c(3)	0	0	0	0
	c(2)	c(4)	0	0	0	0
	0	0	c(1)	c(3)	0	0
	0	0	c(2)	c(4)	0	0
	0	0	0	0	c(1)	c(3)
	0	0	0	0	c(2)	c(4)

The case $n_c = 3$ takes precedence over the case $n_c = N$, and the form $n_c = N$ thus cannot be used.

$n_c = 6$:						
	c(1)	0	0	0	0	0
	0	c(2)	0	0	0	0
	0	0	c(3)	0	0	0
	0	0	0	c(4)	0	0
	0	0	0	0	c(5)	0
	0	0	0	0	0	c(6)

$n_c = 9$:						
	c(1)	c(2)	0	0	0	0
	•	c(3)	0	0	0	0
	0	0	c(4)	c(5)	0	0
	0	0	•	c(6)	0	0
	0	0	0	0	c(7)	c(8)
	0	0	0	0	•	c(9)

$n_c = 12$:	c(1)	c(3)	0	0	0	0
	c(2)	c(4)	0	0	0	0
	0	0	c(5)	c(7)	0	0
	0	0	c(6)	c(8)	0	0
	0	0	0	0	c(9)	c(11)
	0	0	0	0	c(10)	c(12)

$n_c = 21$:	c(1)	c(2)	c(4)	c(6)	c(11)	c(13)
	•	c(3)	c(5)	c(7)	c(12)	c(14)
	•	•	c(8)	c(9)	c(15)	c(17)
	•	•	•	c(10)	c(16)	c(18)
	•	•	•	•	c(19)	c(20)
	•	•	•	•	•	c(21)

$n_c = 36$:	c(1)	c(3)	c(13)	c(15)	c(25)	c(27)
	c(2)	c(4)	c(14)	c(16)	c(26)	c(28)
	c(5)	c(7)	c(17)	c(19)	c(29)	c(31)
	c(6)	c(8)	c(18)	c(20)	c(30)	c(32)
	c(9)	c(11)	c(21)	c(23)	c(33)	c(35)
	c(10)	c(12)	c(22)	c(24)	c(34)	c(36)

Example 4

For structural mechanics problems, consider the following equations for plane stress

$$-\frac{\partial}{\partial x}\left(c_{1111}\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial y}\left(c_{1122}\frac{\partial u}{\partial y}\right)-\frac{\partial}{\partial x}\left(c_{1212}\frac{\partial v}{\partial y}\right)-\frac{\partial}{\partial y}\left(c_{1221}\frac{\partial v}{\partial x}\right)=F_x$$

$$-\frac{\partial}{\partial x}\left(c_{2112}\frac{\partial u}{\partial y}\right)-\frac{\partial}{\partial y}\left(c_{2121}\frac{\partial u}{\partial x}\right)-\frac{\partial}{\partial x}\left(c_{2211}\frac{\partial v}{\partial x}\right)-\frac{\partial}{\partial y}\left(c_{2222}\frac{\partial v}{\partial y}\right)=F_y$$

If you assume that $c_{1221} = c_{2112}$ and $c_{2121} = c_{1212}$, you can use the following commands to set up the variables for the *c* tensors and *f* vector:

```
c = 'c1111|0|c1122|0|c1221|0|c1212|c2211|0|c2222';
f = 'Fx|Fy';
```

See Also

`assema`	Partial Differential Equation Toolbox
`assemb`	Partial Differential Equation Toolbox
`initmesh`	Partial Differential Equation Toolbox
`pdebound`	Partial Differential Equation Toolbox
`refinemesh`	Partial Differential Equation Toolbox

Purpose

Check validity of Geometry Description matrix

Syntax

```
gstat=csgchk(gd,xlim,ylim)
gstat=csgchk(gd)
```

Description

`gstat=csgchk(gd,xlim,ylim)` checks if the solid objects in the Geometry Description matrix `gd` are valid, given optional real numbers `xlim` and `ylim` as current length of the *x*- and *y*-axis, and using a special format for polygons. For a polygon, the last vertex coordinate can be equal to the first one, to indicate a closed polygon. If `xlim` and `ylim` are specified and if the first and the last vertices are not equal, the polygon is considered as closed if these vertices are within a certain "closing distance." These optional input arguments are meant to be used only when calling `csgchk` from `pdetool`.

`gstat=csgchk(gd)` is identical to the preceding call, except for using the same format of `gd` that is used by `decsg`. This call is recommended when using `csgchk` as a command-line function.

`gstat` is a row vector of integers that indicates the validity status of the corresponding solid objects, i.e., columns, in `gd`.

For a circle solid, `gstat`=0 indicates that the circle has a positive radius, `1` indicates a nonpositive radius, and `2` indicates that the circle is not unique.

For a polygon, `gstat`=0 indicates that the polygon is closed and does not intersect itself, i.e., it has a well-defined, unique interior region. `1` indicates an open and non-self-intersecting polygon, `2` indicates a closed and self-intersecting polygon, and `3` indicates an open and self-intersecting polygon.

For a rectangle solid, `gstat` is identical to that of a polygon. This is so because a rectangle is considered as a polygon by `csgchk`.

For an ellipse solid, `gstat`=0 indicates that the ellipse has positive semiaxes, `1` indicates that at least one of the semiaxes is nonpositive, and `2` indicates that the ellipse is not unique.

If gstat consists of zero entries only, then gd is valid and can be used as input argument by decsg.

See Also

decsg Partial Differential Equation Toolbox

Purpose

Delete borders between minimal regions

Syntax

```
[dl1,bt1]=csgdel(dl,bt,bl)
[dl1,bt1]=csgdel(dl,bt)
```

Description

`[dl1,bt1]=csgdel(dl,bt,bl)` deletes the border segments in the list `bl`. If the consistency of the Decomposed Geometry matrix is not preserved by deleting the elements in the list `bl`, additional border segments are deleted. Boundary segments cannot be deleted.

For an explanation of the concepts or border segments, boundary segments, and minimal regions, see `decsg`.

`dl` and `dl1` are Decomposed Geometry matrices. For a description of the Decomposed Geometry matrix, see `decsg`. The format of the Boolean tables `bt` and `bt1` is also described in the entry on `decsg`.

`[dl1,bt1]=csgdel(dl,bt)` deletes all border segments.

See Also

`csgchk`	Partial Differential Equation Toolbox
`decsg`	Partial Differential Equation Toolbox

decsg

Purpose

Decompose Constructive Solid Geometry into minimal regions

Syntax

```
dl=decsg(gd,sf,ns)
dl=decsg(gd)
[dl,bt]=decsg(gd)
[dl,bt]=decsg(gd,sf,ns)
[dl,bt,dl1,bt1,msb]=decsg(gd)
[dl,bt,dl1,bt1,msb]=decsg(gd,sf,ns)
```

Description

This function analyzes the *Constructive Solid Geometry model* (CSG model) that you draw. It analyzes the CSG model, constructs a set of disjoint minimal regions, bounded by boundary segments and border segments, and optionally evaluates a set formula in terms of the objects in the CSG model. We often refer to the set of minimal regions as the *decomposed geometry*. The decomposed geometry makes it possible for other Partial Differential Equation Toolbox functions to “understand” the geometry you specify. For plotting purposes a second set of minimal regions with a connected boundary is constructed.

The graphical user interface `pdetool` uses `decsg` for many purposes. Each time a new solid object is drawn or changed, `pdetool` calls `decsg` to be able to draw the solid objects and minimal regions correctly. The Delaunay triangulation algorithm, `initmesh`, also uses the output of `decsg` to generate an initial mesh.

`dl=decsg(gd,sf,ns)` decomposes the CSG model `gd` into the decomposed geometry `dl`. The CSG model is represented by the Geometry Description matrix, and the decomposed geometry is represented by the Decomposed Geometry matrix. `decsg` returns the minimal regions that evaluate to true for the set formula `sf`. The Name Space matrix `ns` is a text matrix that relates the columns in `gd` to variable names in `sf`.

`dl=decsg(gd)` returns all minimal regions. (The same as letting `sf` correspond to the union of all objects in `gd`.)

`[dl,bt]=decsg(gd)` and `[dl,bt]=decsg(gd,sf,ns)` additionally return a *Boolean table* that relates the original solid objects to the

minimal regions. A column in `bt` corresponds to the column with the same index in `gd`. A row in `bt` corresponds to a minimal region index.

`[dl,bt,dl1,bt1,msb]=decsg(gd)` and `[dl,bt,dl1,bt1,msb]=decsg(gd,sf,ns)` return a second set of minimal regions `dl1` with a corresponding Boolean table `bt1`. This second set of minimal regions all have a connected boundary. These minimal regions can be plotted by using MATLAB patch objects. The second set of minimal regions have borders that may not have been induced by the original solid objects. This occurs when two or more groups of solid objects have nonintersecting boundaries.

The calling sequences additionally return a sequence `msb` of drawing commands for each second minimal region. The first row contains the number of edge segment that bounds the minimal region. The additional rows contain the sequence of edge segments from the Decomposed Geometry matrix that constitutes the bound. If the index edge segment label is greater than the total number of edge segments, it indicates that the total number of edge segments should be subtracted from the contents to get the edge segment label number and the drawing direction is opposite to the one given by the Decomposed Geometry matrix.

Geometry Description Matrix

The *Geometry Description matrix* `gd` describes the CSG model that you draw using `pdetool`. The current Geometry Description matrix can be made available to the MATLAB workspace by selecting the **Export Geometry Description, Set Formula, Labels** option from the **Draw** menu in `pdetool`.

Each column in the Geometry Description matrix corresponds to an object in the CSG model. Four types of *solid objects* are supported. The object type is specified in row 1:

- For the *circle solid,* row one contains 1, and the second and third row contain the center *x*- and *y*-coordinates, respectively. Row four contains the radius of the circle.

- For a *polygon solid,* row one contains 2, and the second row contains the number, *n*, of line segments in the boundary of the polygon. The following *n* rows contain the *x*-coordinates of the starting points of the edges, and the following *n* rows contain the *y*-coordinates of the starting points of the edges.
- For a *rectangle solid,* row one contains 3. The format is otherwise identical to the polygon format.
- For an *ellipse solid,* row one contains 4, the second and third row contains the center *x*- and *y*-coordinates, respectively. Rows four and five contain the semiaxes of the ellipse. The rotational angle of the ellipse is stored in row six.

Set Formula

`sf` contains a *set formula* expressed with the set of variables listed in `ns`. The operators '+', '*', and '-' correspond to the set operations union, intersection, and set difference, respectively. The precedence of the operators '+' and '*' is the same. '-' has higher precedence. The precedence can be controlled with parentheses.

Name Space Matrix

The *Name Space matrix* `ns` relates the columns in `gd` to variable names in `sf`. Each column in `ns` contains a sequence of characters, padded with spaces. Each such character column assigns a name to the corresponding geometric object in `gd`. This way we can refer to a specific object in `gd` in the set formula `sf`.

Decomposed Geometry Matrix

The *Decomposed Geometry matrix* `dl` contains a representation of the decomposed geometry in terms of disjointed *minimal regions* that have been constructed by the `decsg` algorithm. Each edge segment of the minimal regions corresponds to a column in `dl`. We refer to edge segments between minimal regions as *border segments* and outer boundaries as *boundary segments*. In each such column rows two and three contain the starting and ending *x*-coordinate, and rows four and five the corresponding *y*-coordinate. Rows six and seven contain left and right minimal region labels with respect to the direction induced

by the start and end points (counter clockwise direction on circle and ellipse segments). There are three types of possible edge segments in a minimal region:

- For circle edge segments row one is 1. Rows eight and nine contain the coordinates of the center of the circle. Row 10 contains the radius.
- For line edge segments row one is 2.
- For ellipse edge segments row one is 4. Rows eight and nine contain the coordinates of the center of the ellipse. Rows 10 and 11 contain the semiaxes of the ellipse, respectively. The rotational angle of the ellipse is stored in row 12.

Examples

The following command sequence starts pdetool and draws a unit circle and a unit square.

```
pdecirc(0,0,1)
pderect([0 1 0 1])
```

Insert the set formula C1-SQ1. Export the Geometry Description matrix, set formula, and Name Space matrix to the MATLAB workspace by selecting the **Export Geometry Description** option from the **Draw** menu. Then type

```
[dl,bt]=decsg(gd,sf,ns);
dl =
      2.0000    2.0000    1.0000    1.0000    1.0000
           0         0   -1.0000    0.0000    0.0000
      1.0000         0    0.0000    1.0000   -1.0000
           0    1.0000   -0.0000   -1.0000    1.0000
           0         0   -1.0000         0   -0.0000
           0         0    1.0000    1.0000    1.0000
      1.0000    1.0000         0         0         0
           0         0         0         0         0
           0         0         0         0         0
           0         0    1.0000    1.0000    1.0000
```

```
bt =
     1          0
```

There is one minimal region, with five edge segments, three circle edge segments, and two line edge segments.

Algorithm

The algorithm consists of the following steps:

1. Determine the intersection points between the borders of the model objects.
2. For each intersection point, sort the incoming edge segments on angle and curvature.
3. Determine if the induced graph is connected. If not, add some appropriate edges, and redo algorithm from step 1.
4. Cycle through edge segments of minimal regions.
5. For each original region, determine minimal regions inside it.
6. Organize output and remove the additional edges.

Note The input CSG model is not checked for correctness. It is assumed that no circles or ellipses are identical or degenerated and that no lines have zero length. Polygons must not be self-intersecting. Use the function `csgchk` to check the CSG model.

Diagnostics

`NaN` is returned if the set formula `sf` cannot be evaluated.

See Also

`csgchk`	Partial Differential Equation Toolbox
`csgdel`	Partial Differential Equation Toolbox
`pdebound`	Partial Differential Equation Toolbox

`pdecirc`	Partial Differential Equation Toolbox
`pdeellip`	Partial Differential Equation Toolbox
`pdegeom`	Partial Differential Equation Toolbox
`pdepoly`	Partial Differential Equation Toolbox
`pderect`	Partial Differential Equation Toolbox
`pdetool`	Partial Differential Equation Toolbox
`wbound`	Partial Differential Equation Toolbox
`wgeom`	Partial Differential Equation Toolbox

dst, idst

Purpose

Discrete sine transform

Syntax

```
y=dst(x)
y=dst(x,n)
x=idst(y)
x=idst(y,n)
```

Description

The `dst` function implements the following equation:

$$y(k) = \sum_{n=1}^{N} x(n)\sin(\pi\frac{kn}{N+1}), k = 1,\ldots,N$$

`y=dst(x)` computes the discrete sine transform of the columns of `x`. For best performance speed, the number of rows in `x` should be 2_m - 1, for some integer *m*.

`y=dst(x,n)` pads or truncates the vector `x` to length `n` before transforming.

If `x` is a matrix, the `dst` operation is applied to each column.

The `idst` function implements the following equation:

$$y(k) = \frac{2}{N+1}\sum_{n=1}^{N} x(n)\sin(\pi\frac{kn}{N+1}), k = 1,\ldots,N$$

`x=idst(y)` calculates the inverse discrete sine transform of the columns of `y`. For best performance speed, the number of rows in `y` should be 2^m - 1, for some integer *m*.

`x=idst(y,n)` pads or truncates the vector `y` to length `n` before transforming.

If `y` is a matrix, the `idst` operation is applied to each column.

For more information about this algorithm, see “Fast Solution of Poisson’s Equation” on page 3-32.

See Also

`poiasma`	Partial Differential Equation Toolbox
`poiindex`	Partial Differential Equation Toolbox
`poisolv`	Partial Differential Equation Toolbox

hyperbolic

Purpose Solve hyperbolic PDE problem

Syntax

```
u1=hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d)
u1=hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d,rtol)
u1=hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d,rtol,atol)
u1=hyperbolic(u0,ut0,tlist,K,F,B,ud,M)
u1=hyperbolic(u0,ut0,tlist,K,F,B,ud,M,rtol)
u1=hyperbolic(u0,ut0,tlist,K,F,B,ud,M,rtol,atol)
```

Description `u1=hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d)` produces the solution to the FEM formulation of the scalar PDE problem

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f \qquad on \;\; \Omega$$

or the system PDE problem

$$d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c \otimes \nabla u) + au = f \qquad on \;\; \Omega$$

on a mesh described by `p`, `e`, and `t`, with boundary conditions given by `b`, and with initial value `u0` and initial derivative `ut0`.

In the scalar case, each row in the solution matrix `u1` is the solution at the coordinates given by the corresponding column in `p`. Each column in `u1` is the solution at the time given by the corresponding item in `tlist`.

For a system of dimension N with n_p node points, the first n_p rows of `u1` describe the first component of u, the following n_p rows of `u1` describe the second component of u, and so on. Thus, the components of u are placed in blocks `u` as N blocks of node point rows.

`b` describes the boundary conditions of the PDE problem. `b` can be either a Boundary Condition matrix or the name of a Boundary file. The boundary conditions can depend on `t`, the time. The formats of the Boundary Condition matrix and Boundary file are described in the entries on `assemb` and `pdebound`, respectively.

The geometry of the PDE problem is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

The coefficients `c`, `a`, `d`, and `f` of the PDE problem can be given in a variety of ways. The coefficients can depend on `t`, the time. For a complete listing of all options, see `assempde`.

`atol` and `rtol` are absolute and relative tolerances that are passed to the ODE solver.

`u1=hyperbolic(u0,ut0,tlist,K,F,B,ud,M)` produces the solution to the ODE problem

$$B'MB\frac{d^2u_i}{dt^2} + K \cdot u_i = F, \qquad u = Bu_i + u_d$$

with initial values for u being `u0` and `ut0`.

Examples

Solve the wave equation

$$\frac{\partial^2 u}{\partial t^2} = \Delta u$$

on a square geometry $-1 \le x,y \le 1$ (`squareg`), with Dirichlet boundary conditions $u = 0$ for $x = \pm 1$, and Neumann boundary conditions

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

for $y = \pm 1$ (`squareb3`). Choose

$$u(0) = \mathrm{atan}(\cos(\Pi x))$$

and

$$\frac{du(0)}{dt} = 3\sin(\Pi x)\exp(\cos(\Pi y))$$

Compute the solution at times 0, 1/6, 1/3, ... , 29/6, 5.

```
[p,e,t]=initmesh('squareg');
x=p(1,:)';
y=p(2,:)';
u0=atan(cos(pi/2*x));
ut0=3*sin(pi*x).*exp(cos(pi*y));
tlist=linspace(0,5,31);
uu=hyperbolic(u0,ut0,tlist,'squareb3',p,e,t,1,0,0,1);
```

The file `pdedemo6` contains a complete example with animation.

Note In expressions for boundary conditions and PDE coefficients, the symbol `t` is used to denote time. The variable `t` is often used to store the triangle matrix of the mesh. You can use any variable to store the triangle matrix, but in the Partial Differential Equation Toolbox expressions, `t` always denotes time.

See Also

`assempde`	Partial Differential Equation Toolbox
`parabolic`	Partial Differential Equation Toolbox

Purpose

Create initial triangular mesh

Syntax

```
[p,e,t]=initmesh(g)
[p,e,t]=initmesh(g,'PropertyName',PropertyValue,...)
```

Description

`[p,e,t]=initmesh(g)` returns a triangular mesh using the geometry specification function g. It uses a Delaunay triangulation algorithm. The mesh size is determined from the shape of the geometry.

`g` describes the geometry of the PDE problem. `g` can either be a Decomposed Geometry matrix or the name of a Geometry file. The formats of the Decomposed Geometry matrix and Geometry file are described in the entries on `decsg` and `pdegeom`, respectively.

The outputs `p`, `e`, and `t` are the *mesh data*.

In the *Point matrix* `p`, the first and second rows contain *x*- and *y*-coordinates of the points in the mesh.

In the *Edge matrix* `e`, the first and second rows contain indices of the starting and ending point, the third and fourth rows contain the starting and ending parameter values, the fifth row contains the edge segment number, and the sixth and seventh row contain the left- and right-hand side subdomain numbers.

In the *Triangle matrix* `t`, the first three rows contain indices to the corner points, given in counter clockwise order, and the fourth row contains the subdomain number.

The following property name/property value pairs are allowed.

Property	Value	Default	Description
`Hmax`	numeric	estimate	Maximum edge size
`Hgrad`	numeric	`1.3`	`Mesh growth rate`
`Box`	`on|off`	`off`	Preserve bounding box
`Init`	`on|off`	`off`	Edge triangulation

Property	Value	Default	Description
`Jiggle`	`off|mean|min`	`mean`	Call `jigglemesh`
`JiggleIter`	numeric	10	Maximum iterations

The `Hmax` property controls the size of the triangles on the mesh. `initmesh` creates a mesh where no triangle side exceeds `Hmax`.

The `Hgrad` property determines the mesh growth rate away from a small part of the geometry. The default value is 1.3, i.e., a growth rate of 30%. `Hgrad` must be between 1 and 2.

Both the `Box` and `Init` property are related to the way the mesh algorithm works. By turning on `Box` you can get a good idea of how the mesh generation algorithm works within the bounding box. By turning on `Init` you can see the initial triangulation of the boundaries. By using the command sequence

```
[p,e,t]=initmesh(dl,'hmax',inf,'init','on');
[uxy,tn,a2,a3]=tri2grid(p,t,zeros(size(p,2)),x,y);
n=t(4,tn);
```

you can determine the subdomain number `n` of the point `xy`. If the point is outside the geometry, `tn` is `NaN` and the command `n=t(4,tn)` results in a failure.

The `Jiggle` property is used to control whether jiggling of the mesh should be attempted (see `jigglemesh` for details). Jiggling can be done until the minimum or the mean of the quality of the triangles decreases. `JiggleIter` can be used to set an upper limit on the number of iterations.

Algorithm

`initmesh` implements a Delaunay triangulation algorithm:

1. Place node points on the edges.
2. Enclose geometry in bounding box.

3. Triangulate edges.

4. Check that the triangulation respects boundaries.

5. Insert node points into centers of circumscribed circles of large triangles.

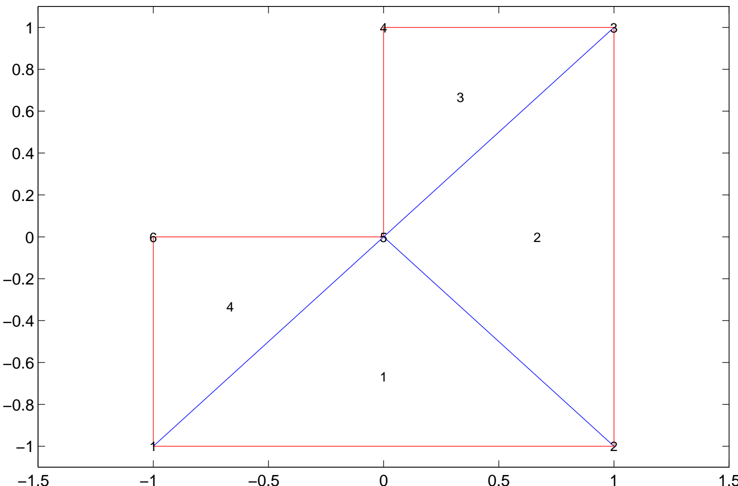
6. Repeat from step 4 if `Hmax` not yet achieved.

7. Remove bounding box.

Examples

Make a simple triangular mesh of the L-shaped membrane in `pdetool`. Before you do anything in `pdetool`, set the **Maximum edge size** to `inf` in the Mesh Parameters dialog box. You open the dialog box by selecting the **Parameters** option from the **Mesh** menu. Also select the items **Show Node Labels** and **Show Triangle Labels** in the **Mesh** menu. Then create the initial mesh by pressing the button. (This can also be done by selecting the **Initialize Mesh** option from the **Mesh** menu.)

The following figure appears.



The corresponding mesh data structures can be exported to the main workspace by selecting the **Export Mesh** option from the **Mesh** menu.

```
p
p =
    -1     1     1     0     0    -1
    -1    -1     1     1     0     0

e
e =
     1     2     3     4     5     6
     2     3     4     5     6     1
     0     0     0     0     0     0
     1     1     1     1     1     1
     1     2     3     4     5     6
     1     1     1     1     1     1
     0     0     0     0     0     0
```

```
t
t =
     1     2     3     1
     2     3     4     5
     5     5     5     6
     1     1     1     1
```

Reference

George, P. L., *Automatic Mesh Generation — Application to Finite Element Methods,* Wiley, 1991.

See Also

`decsg`	Partial Differential Equation Toolbox
`jigglemesh`	Partial Differential Equation Toolbox
`pdegeom`	Partial Differential Equation Toolbox
`refinemesh`	Partial Differential Equation Toolbox

jigglemesh

Purpose

Jiggle internal points of triangular mesh

Syntax

```
p1=jigglemesh(p,e,t)
p1=jigglemesh(p,e,t,'PropertyName',PropertyValue,...)
```

Description

`p1=jigglemesh(p,e,t)` jiggles the triangular mesh by adjusting the node point positions. The quality of the mesh normally increases.

The following property name/property value pairs are allowed.

Property	Value	Default	Description
`Opt`	`off|mean|min`	`mean`	Optimization method
`Iter`	numeric	1 or 20 (see the following bullets)	Maximum iterations

Each mesh point that is not located on an edge segment is moved toward the center of mass of the polygon formed by the adjacent triangles. This process is repeated according to the settings of the `Opt` and `Iter` variables:

- When `Opt` is set to `off` this process is repeated `Iter` times (default: 1).
- When `Opt` is set to `mean` the process is repeated until the mean triangle quality does not significantly increase, or until the bound `Iter` is reached (default: 20).
- When `Opt` is set to `min` the process is repeated until the minimum triangle quality does not significantly increase, or until the bound `Iter` is reached (default: 20).

Examples

Create a triangular mesh of the L-shaped membrane, first without jiggling, and then jiggle the mesh.

```
[p,e,t]=initmesh('lshapeg','jiggle','off');
q=pdetriq(p,t);
pdeplot(p,e,t,'xydata',q,'colorbar','on','xystyle','flat')
```

```
p1=jigglemesh(p,e,t,'opt','mean','iter',inf);
q=pdetriq(p1,t);
pdeplot(p1,e,t,'xydata',q,'colorbar','on','xystyle','flat')
```

See Also

`initmesh`	Partial Differential Equation Toolbox
`pdetriq`	Partial Differential Equation Toolbox

parabolic

Purpose

Solve parabolic PDE problem

Syntax

```
u1=parabolic(u0,tlist,b,p,e,t,c,a,f,d)
u1=parabolic(u0,tlist,b,p,e,t,c,a,f,d,rtol)
u1=parabolic(u0,tlist,b,p,e,t,c,a,f,d,rtol,atol)
u1=parabolic(u0,tlist,K,F,B,ud,M)
u1=parabolic(u0,tlist,K,F,B,ud,M,rtol)
u1=parabolic(u0,tlist,K,F,B,ud,M,rtol,atol)
```

Description

`u1=parabolic(u0,tlist,g,b,p,e,t,c,a,f,d)` produces the solution to the FEM formulation of the scalar PDE problem

$$d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f \quad on \ \Omega$$

or the system PDE problem

$$\underline{d}\frac{\partial u}{\partial t} - \nabla \cdot (\underline{c} \otimes \nabla u) + \underline{a}u = f \quad on \ \Omega$$

on a mesh described by `p`, `e`, and `t`, with boundary conditions given by `b`, and with initial value `u0`.

For the scalar case, each row in the solution matrix `u1` is the solution at the coordinates given by the corresponding column in `p`. Each column in `u1` is the solution at the time given by the corresponding item in `tlist`. For a system of dimension N with n_p node points, the first n_p rows of `u1` describe the first component of u, the following n_p rows of `u1` describe the second component of u, and so on. Thus, the components of u are placed in the vector `u` as N blocks of node point rows.

`b` describes the boundary conditions of the PDE problem. `b` can be either a Boundary Condition matrix or the name of a Boundary file. The boundary conditions can depend on `t`, the time. The formats of the Boundary Condition matrix and Boundary file are described in the entries on `assemb` and `pdebound`, respectively.

The geometry of the PDE problem is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

The coefficients `c`, `a`, `d`, and `f` of the PDE problem can be given in a variety of ways. The coefficients can depend on `t`, the time. For a complete listing of all options, see `initmesh`.

`atol` and `rtol` are absolute and relative tolerances that are passed to the ODE solver.

`u1=parabolic(u0,tlist,K,F,B,ud,M)` produces the solution to the ODE problem

$$B'MB\frac{du_i}{dt} + K \cdot u_i = F, \qquad u = Bu_i + u_d$$

with initial value for u being `u0`.

Examples

Solve the heat equation

$$\frac{\partial u}{\partial t} = \Delta u$$

on a square geometry $-1 \le x,y \le 1$ (`squareg`). Choose $u(0) = 1$ on the disk $x^2 + y^2 < 0.4^2$, and $u(0) = 0$ otherwise. Use Dirichlet boundary conditions $u = 0$ (`squareb1`). Compute the solution at times `linspace(0,0.1,20)`.

```
[p,e,t]=initmesh('squareg');
[p,e,t]=refinemesh('squareg',p,e,t);
u0=zeros(size(p,2),1);
ix=find(sqrt(p(1,:).^2+p(2,:).^2)<0.4);
u0(ix)=ones(size(ix));
tlist=linspace(0,0.1,20);
u1=parabolic(u0,tlist,'squareb1',p,e,t,1,0,1,1);
```

Note In expressions for boundary conditions and PDE coefficients, the symbol `t` is used to denote time. The variable `t` is often used to store the triangle matrix of the mesh. You can use any variable to store the triangle matrix, but in the Partial Differential Equation Toolbox expressions, `t` always denotes time.

See Also

`assempde`	Partial Differential Equation Toolbox
`hyperbolic`	Partial Differential Equation Toolbox

pdeadgsc

Purpose

Select triangles using relative tolerance criterion

Syntax

```
bt=pdeadgsc(p,t,c,a,f,u,errf,tol)
```

Description

`bt=pdeadgsc(p,t,c,a,f,u,errf,tol)` returns indices of triangles to be refined in `bt`. Used from `adaptmesh` to select the triangles to be further refined. The geometry of the PDE problem is given by the mesh data `p` and `t`. For more details, see the entry on `initmesh`.

`c`,`a`, and `f` are PDE coefficients. For details, see `assempde`.

`u` is the current solution, given as a column vector. For details, see `assempde`.

`errf` is the error indicator, as calculated by `pdejmps`.

`tol` is a tolerance parameter.

Triangles are selected using the criterion `errf>tol*scale`, where `scale` is calculated as follows:

Let `cmax`, `amax`, `fmax`, and `umax` be the maximum of `c`, `a`, `f`, and `u`, respectively. Let `l` be the side of the smallest axis-aligned square that contains the geometry.

Then `scale=max(fmax*l^2,amax*umax*l^2,cmax*umax)`. The scaling makes the `tol` parameter independent of the scaling of the equation and the geometry.

See Also

`adaptmesh`	Partial Differential Equation Toolbox
`pdejmps`	Partial Differential Equation Toolbox

pdeadworst

Purpose

Select triangles relative to worst value

Syntax

```
bt=pdeadworst(p,t,c,a,f,u,errf,wlevel)
```

Description

`bt=pdeadworst(p,t,c,a,f,u,errf,wlevel)` returns indices of triangles to be refined in `bt`. Used from `adaptmesh` to select the triangles to be further refined.

The geometry of the PDE problem is given by the mesh data `p` and `t`. For details, see `initmesh`.

`c`, a, and f are PDE coefficients. For details, see `assempde`.

`u` is the current solution, given as a column vector. For details, see `assempde`.

`errf` is the error indicator, as calculated by `pdejmps`.

`wlevel` is the error level relative to the worst error. `wlevel` must be between 0 and 1.

Triangles are selected using the criterion `errf>wlevel*max(errf)`.

See Also

`adaptmesh`	Partial Differential Equation Toolbox
`pdejmps`	Partial Differential Equation Toolbox

Purpose

Interpolation between parametric representation and arc length

Syntax

```
pp=pdearcl(p,xy,s,s0,s1)
```

Description

`pp=pdearcl(p,xy,s,s0,s1)` returns parameter values for a parameterized curve corresponding to a given set of arc length values.

`p` is a monotone row vector of parameter values and `xy` is a matrix with two rows giving the corresponding points on the curve.

The first point of the curve is given the arc length value `s0` and the last point the value `s1`.

On return, pp contains parameter values corresponding to the arc length values specified in `s`.

The arc length values `s`, `s0`, and `s1` can be an affine transformation of the arc length.

Examples

See the example `cardg` on the reference page for `pdegeom`.

See Also

`pdegeom`	Partial Differential Equation Toolbox

pdebound

Purpose

Boundary file

Syntax

```
[q,g,h,r]=pdebound(p,e,u,time)
```

Description

The Boundary file specifies the boundary conditions of a PDE problem.

The most general form of boundary conditions that we can handle is

$$\underline{h}u = \mathbf{r}$$
$$\vec{n} \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u}) + \underline{q}\mathbf{u} = \mathbf{g} + \underline{h}' \mu$$

By the notation $\vec{n} \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u})$ we mean the N-by-1 matrix, where N is the dimension of the system, with $(i,1)$-component

$$\sum_{j=1}^{N} \left(\cos(\alpha) c_{i,j,1,1} \frac{\partial}{\partial x} + \cos(\alpha) c_{i,j,1,2} \frac{\partial}{\partial y} + \sin(\alpha) c_{i,j,2,1} \frac{\partial}{\partial x} + \sin(\alpha) c_{i,j,2,2} \frac{\partial}{\partial y} \right) u_j,$$

where the outward normal vector of the boundary $\vec{n} = (\cos(\alpha), \sin(\alpha))$. There are M Dirichlet conditions and the h-matrix is M-by-N, $M \geq 0$.

The generalized Neumann condition contains a source $\underline{h}' m$ where the Lagrange multipliers μ is computed such that the Dirichlet conditions become satisfied.

The data that you specify is q, g, h, and r.

For $M = 0$ we say that we have a generalized Neumann boundary condition, for $M = N$ a Dirichlet boundary condition, and for $0 < M < N$ a mixed boundary condition.

The Boundary file `[q,g,h,r]=pdebound(p,e,u,time)` computes the values of q, g, h, and r, on the a set of edges `e`.

The matrices `p` and `e` are mesh data. `e` needs only to be a subset of the edges in the mesh. Details on the mesh data representation can be found in the entry on `initmesh`.

The input arguments u and time are used for the nonlinear solver and time stepping algorithms, respectively. u and time are empty matrices if the corresponding parameter is not passed to assemb. If time is NaN and any of the function q, g, h, and r depends on time, pdebound must return a matrix of correct size, containing NaNs in all positions, in the corresponding output argument. It is not possible to explicitly refer to the time derivative of the solution in the boundary conditions.

The solution u is represented by the solution vector u. Details on the representation can be found in the entry on assempde.

q and g must contain the value of q and g on the midpoint of each boundary. Thus we have size(q)=[N^2 ne], where N is the dimension of the system, and ne the number of edges in e, and size(g)=[N ne]. For the Dirichlet case, the corresponding values must be zeros.

h and r must contain the values of h and r at the first point on each edge followed by the value at the second point on each edge. Thus we have size(h)=[N^2 2*ne], where N is the dimension of the system, and ne the number of edges in e, and size(r)=[N 2*ne]. When $M < N$, h and r must be padded with $N - M$ rows of zeros.

The elements of the matrices q and h are stored in column-wise ordering in the MATLAB matrices q and h.

Examples

For the boundary conditions

$$\begin{pmatrix} 1 & -1 \end{pmatrix}\mathbf{u} = 2$$

$$\vec{n} \cdot (\underline{c} \otimes \nabla\mathbf{u}) + \begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix}\mathbf{u} = \begin{pmatrix} 3 \\ 4 \end{pmatrix} + \underline{h}'\mu$$

the following values should be stored in q, g, h, and r

```
          1
q=[  ...  2  ...  ]
          2
          0
```

```
g=[ ... 3 ... ]
        4

        1     1
h=[ ... 0 ... 0 ... ]
       -1    -1
        0     0

r=[ ... 2 ... 2 ... ]
        0     0
```

See Also

`initmesh`	Partial Differential Equation Toolbox
`pdeent`	Partial Differential Equation Toolbox
`pdegeom`	Partial Differential Equation Toolbox
`pdesdt`	Partial Differential Equation Toolbox

Purpose

Flux of PDE solution

Syntax

```
[cgxu,cgyu]=pdecgrad(p,t,c,u)
[cgxu,cgyu]=pdecgrad(p,t,c,u,time)
[cgxu,cgyu]=pdecgrad(p,t,c,u,time,sdl)
```

Description

`[cgxu,cgyu]=pdecgrad(p,t,c,u)` returns the flux, $\underline{\mathbf{c}} \otimes \nabla \mathbf{u}$, evaluated at the center of each triangle.

Row i of `cgxu` contains

$$\sum_{j=1}^{N} c_{ij11}\frac{\partial u_j}{\partial x} + c_{ij12}\frac{\partial u_j}{\partial y}$$

Row i of `cgyu` contains

$$\sum_{j=1}^{N} c_{ij21}\frac{\partial u_j}{\partial x} + c_{ij22}\frac{\partial u_j}{\partial y}$$

There is one column for each triangle in `t` in both `cgxu` and `cgyu`.

The geometry of the PDE problem is given by the mesh data `p` and `t`. Details on the mesh data representation can be found in the entry on `initmesh`.

The coefficient `c` of the PDE problem can be given in a variety of ways. A complete listing of all options can be found in the entry on `assempde`.

The format for the solution vector `u` is described in `assempde`.

The scalar optional argument `time` is used for parabolic and hyperbolic problems, if `c` depends on `t`, the time.

The optional argument `sdl` restricts the computation to the subdomains in the list `sdl`.

See Also

`assempde` Partial Differential Equation Toolbox

pdecirc

Purpose

Draw circle

Syntax

```
pdecirc(xc,yc,radius)
pdecirc(xc,yc,radius,label)
```

Description

`pdecirc(xc,yc,radius)` draws a circle with center in `(xc,yc)` and radius `radius`. If the `pdetool` GUI is not active, it is automatically started, and the circle is drawn in an empty geometry model.

The optional argument `label` assigns a name to the circle (otherwise a default name is chosen).

The state of the Geometry Description matrix inside `pdetool` is updated to include the circle. You can export the Geometry Description matrix from `pdetool` by using the **Export Geometry Description** option from the **Draw** menu. For a details on the format of the Geometry Description matrix, see `decsg`.

Examples

The following command starts `pdetool` and draws a unit circle.

```
pdecirc(0,0,1)
```

See Also

`pdeellip`	Partial Differential Equation Toolbox
`pdepoly`	Partial Differential Equation Toolbox
`pderect`	Partial Differential Equation Toolbox
`pdetool`	Partial Differential Equation Toolbox

Purpose

Shorthand command for contour plot

Syntax

```
pdecont(p,t,u)
pdecont(p,t,u,n)
pdecont(p,t,u,v)
h=pdecont(p,t,u)
h=pdecont(p,t,u,n)
h=pdecont(p,t,u,v)
```

Description

`pdecont(p,t,u)` draws 10 level curves of the PDE node or triangle data `u`. h=pdecont(p,t,u) additionally returns handles to the drawn axes objects.

If `u` is a column vector, node data is assumed. If `u` is a row vector, triangle data is assumed. Triangle data is converted to node data using the function `pdeprtni`.

The geometry of the PDE problem is given by the mesh data `p` and `t`. For details on the mesh data representation, see `initmesh`.

`pdecont(p,t,u,n)` plots using `n` levels.

`pdecont(p,t,u,v)` plots using the levels specified by `v`.

This command is just shorthand for the call

```
pdeplot(p,[],t,'xydata',u,'xystyle','off','contour',...
'on','levels',n,'colorbar','off');
```

If you want to have more control over your contour plot, use `pdeplot` instead of `pdecont`.

Examples

Plot the contours of the solution to the equation $-\Delta u = 1$ over the geometry defined by the L-shaped membrane. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$.

```
[p,e,t]=initmesh('lshapeg');
[p,e,t]=refinemesh('lshapeg',p,e,t);
u=assempde('lshapeb',p,e,t,1,0,1);
pdecont(p,t,u)
```

See Also

`pdemesh`	Partial Differential Equation Toolbox
`pdeplot`	Partial Differential Equation Toolbox
`pdesurf`	Partial Differential Equation Toolbox

Purpose Solve eigenvalue PDE problem

Syntax

```
[v,l]=pdeeig(b,p,e,t,c,a,d,r)
[v,l]=pdeeig(K,B,M,r)
```

Description

`[v,l]=pdeeig(b,p,e,t,c,a,d,r)` produces the solution to the FEM formulation of the scalar PDE eigenvalue problem

$$-\Delta \cdot (c\nabla u) + au = \lambda du \text{ on } \Omega$$

or the system PDE eigenvalue problem

$$-\nabla \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u}) + \underline{a}\mathbf{u} = \lambda du \quad on \quad \Omega$$

on a geometry described by `p`, `e`, and `t`, and with boundary conditions given by `b`.

`r` is a two-element vector, indicating an interval on the real axis. (The left-hand side can be `-Inf`.) The algorithm returns all eigenvalues in this interval in `l`, up to a maximum of 99 eigenvalues.

`v` is an *eigenvector matrix*. For the scalar case each column in `v` is an eigenvector of solution values at the corresponding node points from `p`. For a system of dimension N with n_p node points, the first n_p rows of `v` describe the first component of v, the following n_p rows of `v` describe the second component of v, and so on. Thus, the components of v are placed in blocks `v` as N blocks of node point rows.

`b` describes the boundary conditions of the PDE problem. `b` can be either a Boundary Condition matrix or the name of a Boundary file. The formats of the Boundary Condition matrix and Boundary file are described in the entries on `assemb` and `pdebound`, respectively. The eigenvalue PDE problem is a *homogeneous* problem, i.e., only boundary conditions where $g = 0$ and $r = 0$ can be used. The nonhomogeneous part is removed automatically.

The geometry of the PDE problem is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

The coefficients c, a, d of the PDE problem can be given in a wide variety of ways. In the context of pdeeig the coefficients cannot depend on u nor t, the time. For a complete listing of all options, see assempde.

[v,l]=pdeeig(K,B,M,r) produces the solution to the generalized sparse matrix eigenvalue problem

$$K\,u_i = \lambda B'MBu_i \; u = Bu_i$$

with Real(λ) in the interval in r.

Examples

Compute the eigenvalues less than 100 and corresponding eigenmodes for

$$-\nabla u = \lambda u,$$

on the geometry of the L-shaped membrane. Then display the first and sixteenth eigenmodes.

```
[p,e,t]=initmesh('lshapeg');
[p,e,t]=refinemesh('lshapeg',p,e,t);
[p,e,t]=refinemesh('lshapeg',p,e,t);
[v,l]=pdeeig('lshapeb',p,e,t,1,0,1,[-Inf 100]);
l(1)                      % first eigenvalue
pdesurf(p,t,v(:,1))       % first eigenmode
figure
membrane(1,20,9,9)        % the MATLAB function
figure
l(16)                     % sixteenth eigenvalue
pdesurf(p,t,v(:,16))      % sixteenth eigenmode
```

Algorithm

pdeeig calls sptarn to calculate eigenvalues. For details of the algorithm, see the sptarn reference pages.

Caution

In the standard case c and d are positive in the entire region. All eigenvalues are positive, and 0 is a good choice for a lower bound of the interval. The cases where either c or d is zero are discussed next.

- If $d = 0$ in a subregion, the mass matrix M becomes singular. This does not cause any trouble, provided that $c > 0$ everywhere. The pencil (K,M) has a set of infinite eigenvalues.
- If $c = 0$ in a subregion, the stiffness matrix `K` becomes singular, and the pencil (K,M) has many zero eigenvalues. With an interval containing zero, `pdeeig` goes on for a very long time to find all the zero eigenvalues. Choose a positive lower bound away from zero but below the smallest nonzero eigenvalue.
- If there is a region where both $c = 0$ and $d = 0$, we get a singular pencil. The whole eigenvalue problem is undetermined, and any value is equally plausible as an eigenvalue.

Some of the awkward cases are detected by `pdeeig`. If the shifted matrix is singular, another shift is attempted. If the matrix with the new shift is still singular a good guess is that the entire pencil (K,M) is singular.

If you try any problem not belonging to the standard case, you must use your knowledge of the original physical problem to interpret the results from the computation.

See Also

`sptarn`	Partial Differential Equation Toolbox

pdeellip

Purpose

Draw ellipse

Syntax

```
pdeellip(xc,yc,a,b,phi)
pdeellip(xc,yc,a,b,phi,label)
```

Description

`pdeellip(xc,yc,a,b,phi)` draws an ellipse with center in `(xc,yc)` and semiaxes `a` and `b`. The rotation of the ellipse (in radians) is given by `phi`. If the `pdetool` GUI is not active, it is automatically started, and the ellipse is drawn in an empty geometry model.

The optional argument `label` assigns a name to the ellipse (otherwise a default name is chosen.)

The state of the Geometry Description matrix inside `pdetool` is updated to include the ellipse. You can export the Geometry Description matrix from `pdetool` by selecting the **Export Geometry Description** option from the **Draw** menu. For a details on the format of the Geometry Description matrix, see `decsg`.

Examples

The following command starts `pdetool` and draws an ellipse.

```
pdeellip(0,0,1,0.3,pi/4)
```

See Also

`pdecirc`	Partial Differential Equation Toolbox
`pdepoly`	Partial Differential Equation Toolbox
`pderect`	Partial Differential Equation Toolbox
`pdetool`	Partial Differential Equation Toolbox

Purpose

Indices of triangles neighboring given set of triangles

Syntax

```
ntl=pdeent(t,tl)
```

Description

Given triangle data `t` and a list of triangle indices `tl`, `ntl` contains indices of the triangles in `tl` and their immediate neighbors, i.e., those whose intersection with `tl` is nonempty.

See Also

`refinemesh` Partial Differential Equation Toolbox

pdegeom

Purpose Geometry file

Syntax

```
ne=pdegeom
d=pdegeom(bs)
[x,y]=pdegeom(bs,s)
```

Description

We represent 2-D regions by parameterized edge segments. Both the regions and edge segments are assigned unique positive numbers as labels. The edge segments cannot overlap. The full 2-D problem description can contain several nonintersecting regions, and they can have common border segments. The boundary of a region can consist of several edge segments. All edge segment junctions must coincide with edge segment endpoints. We sometimes refer to an edge segment as a *boundary segment* or a *border segment*. A boundary segment is located on the outer boundary of the union of the minimal regions, and a border segment is located on the border between minimal regions.

There are two options for specifying the problem geometry:

- Create a Decomposed Geometry matrix with the function `decsg`. This is done automatically from `pdetool`. Using the Decomposed Geometry matrix restricts the edge segments to be straight lines, circle, or ellipse segments. The Decomposed Geometry matrix can be used instead of the Geometry file.
- Create a Geometry file. By creating your own Geometry file, you can create a geometry that follows any mathematical function exactly. The following is an example of how to create a cardioid.

`ne=pdegeom` is the number of edge segments.

`d=pdegeom(bs)` is a matrix with one column for each edge segment specified in `bs`.

- Row 1 contains the start parameter value.
- Row 2 contains the end parameter value.

- Row 3 contains the label of the left-hand region (left with respect to direction induced by start and end from row 1 and 2).
- Row 4 contains the label of the right-hand region.

The complement of the union of all regions is assigned the region number 0.

`[x,y]=pdegeom(bs,s)` produces coordinates of edge segment points. `bs` specifies the edge segments and `s` the corresponding parameter values. `bs` can be a scalar. The parameter `s` should be approximately proportional to the curve length. All minimal regions should have at least two, and preferably three, edge segments in their boundary.

Examples

The function `cardg` defines the geometry of a cardioid

$r = 2(1+\cos(\Phi))$.

```
function [x,y]=cardg(bs,s)
%CARDG Geometry File defining the geometry of a cardioid.
nbs=4;

if nargin==0
  x=nbs;
  return
end
dl=[  0      pi/2   pi       3*pi/2
      pi/2   pi     3*pi/2   2*pi
      1      1      1        1
      0      0      0        0];

if nargin==1
  x=dl(:,bs);
  return
end

x=zeros(size(s));
y=zeros(size(s));
```

```
[m,n]=size(bs);
if m==1 & n==1,
  bs=bs*ones(size(s)); % expand bs
elseif m~=size(s,1) | n~=size(s,2),
  error('bs must be scalar or of same size as s');
end

nth=400;
th=linspace(0,2*pi,nth);
r=2*(1+cos(th));
xt=r.*cos(th);
yt=r.*sin(th);
th=pdearcl(th,[xt;yt],s,0,2*pi);
r=2*(1+cos(th));
x(:)=r.*cos(th);
y(:)=r.*sin(th);
```

We use the function `pdearcl` to make the parameter `s` proportional to arc length. You can test the function by typing

```
pdegplot('cardg'), axis equal
[p,e,t]=initmesh('cardg');
pdemesh(p,e,t), axis equal
```

Then solve the PDE problem $-\Delta u = 1$ on the geometry defined by the cardioid. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$. Finally plot the solution.

```
u=assempde('cardb',p,e,t,1,0,1);
pdesurf(p,t,u);
```

Caution

The parameter `s` should be approximately proportional to the curve length. All minimal regions should have at least two, and preferably three, edge segments in their boundary.

See Also

`initmesh`	Partial Differential Equation Toolbox
`pdearcl`	Partial Differential Equation Toolbox
`refinemesh`	Partial Differential Equation Toolbox

Purpose

Plot PDE geometry

Syntax

```
pdegplot(g)
h=pdegplot(g)
```

Description

`pdegplot(g)` plots the geometry of a PDE problem.

`h=pdegplot(g)` returns handles to the plotted axes objects.

`g` describes the geometry of the PDE problem. `g` can either be a Decomposed Geometry matrix or the name of a Geometry file. The formats of the Decomposed Geometry matrix and Geometry file are described in the entries on `decsg` and `pdegeom`, respectively.

Examples

Plot the geometry of the L-shaped membrane

```
pdegplot('lshapeg')
```

See Also

`initmesh`	Partial Differential Equation Toolbox
`pdearcl`	Partial Differential Equation Toolbox
`refinemesh`	Partial Differential Equation Toolbox

Purpose

Gradient of PDE solution

Syntax

```
[ux,uy]=pdegrad(p,t,u)
[ux,uy]=pdegrad(p,t,u,sdl)
```

Description

`[ux,uy]=pdegrad(p,t,u)` returns the gradient of `u` evaluated at the center of each triangle.

Row i from 1 to N of `ux` contains

$$\frac{\partial u_i}{\partial x}$$

Row i from 1 to N of `uy` contains

$$\frac{\partial u_i}{\partial y}$$

There is one column for each triangle in `t` in both `ux` and `uy`.

The geometry of the PDE problem is given by the mesh data `p` and `t`. For details on the mesh data representation, see `initmesh`.

For a description of the format for the solution vector `u`, see `assempde`.

The optional argument `sdl` restricts the computation to the subdomains in the list `sdl`.

See Also

`assempde`	Partial Differential Equation Toolbox

pdeintrp

Purpose Interpolate from node data to triangle midpoint data

Syntax

```
ut=pdeintrp(p,t,un)
```

Description `ut=pdeintrp(p,t,un)` gives linearly interpolated values at triangle midpoints from the values at node points.

The geometry of the PDE problem is given by the mesh data `p` and `t`. For details on the mesh data representation, see `initmesh`.

Let N be the dimension of the PDE system, n_p the number of node points, and n_t the number of triangles. The components of the *node data* are stored in `un` either as N columns of length n_p or as an ordinary solution vector. The first n_p values of `un` describe the first component, the following n_p values of `un` describe the second component, and so on. The components of *triangle data* are stored in `ut` as N rows of length n_t.

Caution `pdeprtni` and `pdeintrp` are not inverse functions. The interpolation introduces some averaging.

See Also

`assempde`	Partial Differential Equation Toolbox
`initmesh`	Partial Differential Equation Toolbox
`pdeprtni`	Partial Differential Equation Toolbox

Purpose

Error estimates for adaptation

Syntax

```
errf=pdejmps(p,t,c,a,f,u,alfa,beta,m)
```

Description

`errf=pdejmps(p,t,c,a,f,u,alfa,beta,m)` calculates the error indication function used for adaptation. The columns of `errf` correspond to triangles, and the rows correspond to the different equations in the PDE system.

`p` and`t` are mesh data. For details, see `initmesh`.

`c, a,` and `f` are PDE coefficients. See `assempde` for details. `c`, `a`, and `f` must be expanded, so that columns correspond to triangles.

`u` is the solution vector. For details, see `assempde`.

The formula for computing the error indicator $E(K)$ for each triangle K is

$$E(K) = \alpha \| h^m (f - au) \|_K + \beta \left(\frac{1}{2} \sum_{\tau \in \partial K} h_\tau^{2m} [\mathbf{n}_\tau \cdot (c \nabla u_h)]^2 \right)^{1/2}$$

where $\mathbf{n}_\tau$ is the unit normal of edge τ and the braced term is the jump in flux across the element edge, where α and β are weight indices and m is an order parameter. The norm is an L_2 norm computed over the element K. The error indicator is stored in `errf` as column vectors, one for each triangle in `t`. More information can be found in the section "Adaptive Mesh Refinement" on page 3-29.

See Also

`adaptmesh`	Partial Differential Equation Toolbox
`pdeadgsc`	Partial Differential Equation Toolbox
`pdeadworst`	Partial Differential Equation Toolbox

pdemdlcv

Purpose Convert Partial Differential Equation Toolbox 1.0 model files to 1.0.2 format

Syntax `pdemdlcv(infile,outfile)`

Description `pdemdlcv(infile,outfile)` converts the Partial Differential Equation Toolbox 1.0 model file `infile` to a Partial Differential Equation Toolbox 1.0.2 compatible model file. The resulting file is saved as `outfile`. If the `.m` extension is missing in `outfile`, it is added automatically.

Examples `pdedmdlcv('model42.m','model5.m')` converts the Partial Differential Equation Toolbox 1.0 Model file `model42.m` and saves the converted model in `model5.m`.

Purpose

Plot PDE triangular mesh

Syntax

```
pdemesh(p,e,t)
pdemesh(p,e,t,u)
h=pdemesh(p,e,t)
h=pdemesh(p,e,t,u)
```

Description

`pdemesh(p,e,t)` plots the mesh specified by the mesh data `p`, `e`, and `t`.

`h=pdemesh(p,e,t)` additionally returns handles to the plotted axes objects.

`pdemesh(p,e,t,u)` plots PDE node or triangle data `u` using a mesh plot. If `u` is a column vector, node data is assumed. If `u` is a row vector, triangle data is assumed. This command plots substantially faster than the `pdesurf` command.

The geometry of the PDE problem is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

This command is just shorthand for the calls

```
pdeplot(p,e,t)
pdeplot(p,e,t,'zdata',u)
```

If you want to have more control over your mesh plot, use `pdeplot` instead of `pdemesh`.

Examples

Plot the mesh for the geometry of the L-shaped membrane.

```
[p,e,t]=initmesh('lshapeg');
[p,e,t]=refinemesh('lshapeg',p,e,t);
pdemesh(p,e,t)
```

Now solve Poisson's equation $-\Delta u = 1$ over the geometry defined by the L-shaped membrane. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$, and plot the result.

```
u=assempde('lshapeb',p,e,t,1,0,1);
```

```
pdemesh(p,e,t,u)
```

See Also

`pdecont`	Partial Differential Equation Toolbox
`pdeplot`	Partial Differential Equation Toolbox
`pdesurf`	Partial Differential Equation Toolbox

Purpose

Solve nonlinear PDE problem

Syntax

```
[u,res]=pdenonlin(b,p,e,t,c,a,f)
[u,res]=pdenonlin(b,p,e,t,c,a,f,'PropertyName', 'PropertyValue',...)
```

Description

`[u,res]=pdenonlin(b,p,e,t,c,a,f)` solves the nonlinear PDE scalar PDE problem

$$-\nabla \cdot (c\nabla u) + au = f \text{ on } \Omega$$

or the nonlinear system PDE problem

$$-\nabla \cdot (\underline{\mathbf{c}} \otimes \nabla \mathbf{u}) + \underline{a}\mathbf{u} = \mathbf{f} \quad on \quad \Omega$$

where the coefficients c, a, and f may depend on u. The algorithm solves the equation by using damped Newton iteration with the Armijo-Goldstein line search strategy.

The solution u is represented as the solution vector `u`. For details on the representation of the solution vector, see `assempde`. `res` is the norm of the Newton step residuals.

The triangular mesh of the PDE problem is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

`b` describes the boundary conditions of the PDE problem. `b` can be either a Boundary Condition matrix or the name of a Boundary file. The formats of the Boundary Condition matrix and Boundary file are described in the entries on `assemb` and `pdebound`, respectively, respectively. For the general call to `pdebound`, the boundary conditions can also depend on `u`. A fixed-point iteration strategy is employed to solve for the nonlinear boundary conditions.

The coefficients c, a, `f` of the PDE problem can be given in a wide variety of ways. In the context of `pdenonlin` the coefficients can depend on `u`. The coefficients cannot depend on `t`, the time. For a complete listing of all format options, see `assempde`.

The solver can be fine-tuned by setting some of the options described next.

Property Name	Property Value	Default	Description
Jacobian	fixed|lumped|full	fixed	Approximation of Jacobian
U0	string or numeric	0	Initial solution guess
Tol	positive scalar	1e-4	Residual size at termination
MaxIter	positive integer	25	Maximum Gauss-Newton iterations
MinStep	positive scalar	1/2^16	Minimum damping of search direction
Report	on|off	off	Print convergence information
Norm	string or numeric	Inf	Residual norm

There are three methods currently implemented to compute the Jacobian:

- Numerical evaluation of the full Jacobian based on the sparse version of the function numjac
- A "lumped" approximation described in Chapter 3, "Finite Element Method" based on the numerical differentiation of the coefficients
- A fixed-point iteration matrix where the Jacobian is approximated by the stiffness matrix

Select the desired method by setting the Jacobian property to full, lumped, or fixed, bearing in mind that the more precise methods are computationally more expensive.

U0 is the starting guess that can be given as an expression, a generic scalar, or a vector. By default it is set to 0, but this is useless in problems such as $\nabla(1/u\nabla u) = 0$ with Dirichlet boundary conditions $u = e^{x+y}$. Tol fixes the exit criterion from the Gauss-Newton iteration,

i.e., the iterations are terminated when the residual norm is less than Tol. The norm in which the residual is computed is selected through Norm. This can be any admissible MATLAB vector norm or energy for the energy norm.

MaxIter and MinStep are safeguards against infinite Gauss-Newton loops and they bound the number of iterations and the step size used in each iteration. Setting Report to on forces printing of convergence information.

Diagnostics

If the Newton iteration does not converge, the error message Too many iterations or Stepsize too small is displayed. If the initial guess produces matrices containing NaN or Inf elements, the error message Unsuitable initial guess U0 (default: U0=0) is printed.

See Also

assempde	Partial Differential Equation Toolbox
pdebound	Partial Differential Equation Toolbox

pdeplot

Purpose

Generic plot function

Syntax

```
pdeplot(p,e,t,'PropertyName',PropertyValue,)
h=pdeplot(p,e,t,'PropertyName',PropertyValue,)
```

Description

`pdeplot(p,e,t,p1,v1,...)` is the generic Partial Differential Equation Toolbox plot function. It can display several functions of a PDE solution at the same time.

The geometry of the PDE problem is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

Valid property/value pairs include the following.

Property Name	Property Value/Default	Description
`xydata`	data	Triangle data
`xystyle`	`off|flat|{interp}`	x-y data plot style
`contour`	`{off}|on`	Show contours
`zdata`	data	Node or triangle data
`zstyle`	`off|{continuous}|discontinuous`	3-D height plot style
`flowdata`	data	Node or triangle data
`flowstyle`	`off|{arrow}`	Flow plot style
`colormap`	colormap `cool`	x-y data colormap name or colormap matrix
`xygrid`	`{off}|on`	Convert to x-y grid before plotting
`gridparam`	`[tn; a2; a3]`	Triangle index and interpolation parameters from earlier call to `tri2grid`
`mesh`	`{off}|on`	Show mesh in plot
`colorbar`	`off|{on}`	Show color bar

Property Name	Property Value/Default	Description
`title`	`''`	Plot title text
`levels`	10	Number of levels or a vector specifying levels

The `pdeplot` is used both from inside the `pdetool` GUI and from the command line. It is able to display three entities simultaneously. `xydata` can be visualized by a surface plot. Either flat or interpolated (default) shading can be used for the surface plots. A contour plot can be superimposed on the surface plot (in black) or plotted independently (in colors) by setting `contour` to `on`. `zdata` is visualized by displaying height. The triangles can be either tilted by interpolation (default) or flat. Flow data can be visualized by plotting arrows like the MATLAB `quiver` plot. All data types can be either node data or triangle data (flow data can only be triangle data). *Node data* is represented by a column vector of length `size(p,2)` and *triangle data* is represented by a row vector of length `size(t,2)`. If no `xydata`, `zdata`, or `flowdata` is supplied, `pdeplot` plots the mesh specified by `p`, `e`, and `t`.

The option `mesh` displays or hides (default) the triangle mesh in the plot. The option `xygrid` first converts the data to *x-y* data (using `tri2grid`), and then uses a standard MATLAB plotting algorithm. The property `gridparam` passes the `tri2grid` data to `pdeplot`. This speeds up animation (see `pdedemo5` and `pdedemo6`). The property `colormap` renders the plot using any MATLAB colormap or color matrix. `colorbar` adds a color bar to the plot. `title` inserts a title into the plot. `levels` only applies to contour plots: Given a scalar integer value, it plots that number of equally spaced contour levels; given a vector of level values, it plots those contour lines on the levels in the vector.

`h=pdeplot(p,t,u)` additionally returns handles to the drawn axes objects.

Examples

The following command sequence plots the solution to Poisson's equation on the L-shaped membrane in 3-D.

```
[p,e,t]=initmesh('lshapeg');
```

```
u=assempde('lshapeb',p,e,t,1,0,1);
pdeplot(p,e,t,'xydata',u,'zdata',u,'mesh','off');
```

See Also

`pdecont`	Partial Differential Equation Toolbox
`pdegplot`	Partial Differential Equation Toolbox
`pdemesh`	Partial Differential Equation Toolbox
`pdesurf`	Partial Differential Equation Toolbox

Purpose

Draw polygon

Syntax

```
pdepoly(x,y)
pdepoly(x,y,label)
```

Description

`pdepoly(x,y)` draws a polygon with corner coordinates defined by `x` and `y`. If the `pdetool` GUI is not active, it is automatically started, and the polygon is drawn in an empty geometry model.

The optional argument `label` assigns a name to the polygon (otherwise a default name is chosen).

The state of the Geometry Description matrix inside `pdetool` is updated to include the polygon. You can export the Geometry Description matrix from `pdetool` by using the **Export Geometry Description** option from the **Draw** menu. For a details on the format of the Geometry Description matrix, see `decsg`.

Examples

The command

```
pdepoly([-1 0 0 1 1 -1],[0 0 1 1 -1 -1]);
```

creates the L-shaped membrane geometry as one polygon.

See Also

`pdecirc`	Partial Differential Equation Toolbox
`pderect`	Partial Differential Equation Toolbox
`pdetool`	Partial Differential Equation Toolbox

pdeprtni

Purpose

Interpolate from triangle midpoint data to node data

Syntax

```
un=pdeprtni(p,t,ut)
```

Description

`un=pdeprtni(p,t,ut)` gives linearly interpolated values at node points from the values at triangle midpoints.

The geometry of the PDE problem is given by the mesh data `p` and `t`. For details on the mesh data representation, see `initmesh`.

Let N be the dimension of the PDE system, n_p the number of node points, and n_t the number of triangles. The components of triangle data in `ut` are stored as N rows of length n_t. The components of the node data are stored in `un` as N columns of length n_p.

Caution

`pdeprtni` and `pdeintrp` are not inverse functions. The interpolation introduces some averaging.

See Also

`assempde`	Partial Differential Equation Toolbox
`initmesh`	Partial Differential Equation Toolbox
`pdeintrp`	Partial Differential Equation Toolbox

Purpose

Draw rectangle

Syntax

```
pderect(xy)
pderect(xy,label)
```

Description

`pderect(xy)` draws a rectangle with corner coordinates defined by `xy=[xmin xmax ymin ymax]`. If the `pdetool` GUI is not active, it is automatically started, and the rectangle is drawn in an empty geometry model.

The optional argument `label` assigns a name to the rectangle (otherwise a default name is chosen).

The state of the Geometry Description matrix inside `pdetool` is updated to include the rectangle. You can export the Geometry Description matrix from `pdetool` by selecting the **Export Geometry Description** option from the **Draw** menu. For details on the format of the Geometry Description matrix, see `decsg`.

Examples

The following command sequence starts `pdetool` and draws the L-shaped membrane as the union of three squares.

```
pderect([-1 0 -1 0])
pderect([0 1 -1 0])
pderect([0 1 0 1])
```

See Also

`pdecirc`	Partial Differential Equation Toolbox
`pdeellip`	Partial Differential Equation Toolbox
`pdepoly`	Partial Differential Equation Toolbox
`pdetool`	Partial Differential Equation Toolbox

pdesdp, pdesde, pdesdt

Purpose

Indices of points/edges/triangles in set of subdomains

Syntax

```
c=pdesdp(p,e,t)
[i,c]=pdesdp(p,e,t)
c=pdesdp(p,e,t,sdl)
[i,c]=pdesdp(p,e,t,sdl)
i=pdesdt(t)
i=pdesdt(t,sdl)
i=pdesde(e)
i=pdesde(e,sdl)
```

Description

`[i,c]=pdesdp(p,e,t,sdl)` given mesh data `p`, `e`, and `t` and a list of subdomain numbers `sdl`, the function returns all points belonging to those subdomains. A point can belong to several subdomains, and the points belonging to the domains in `sdl` are divided into two disjoint sets. `i` contains indices of the points that wholly belong to the subdomains listed in `sdl`, and `c` lists points that also belongs to the other subdomains.

`c=pdesdp(p,e,t,sdl)` returns indices of points that belong to more than one of the subdomains in `sdl`.

`i=pdesdt(t,sdl)` given triangle data `t` and a list of subdomain numbers `sdl`, `i` contains indices of the triangles inside that set of subdomains.

`i=pdesde(e,sdl)` given edge data `e`, it extracts indices of outer boundary edges of the set of subdomains.

If `sdl` is not given, a list of all subdomains is assumed.

Purpose

Calculate structural mechanics tensor functions

Syntax

```
ux=pdesmech(p,t,c,u,'PropertyName',PropertyValue,...)
```

Description

`ux=pdesmech(p,t,c,u,p1,v1,...)` returns a tensor expression evaluated at the center of each triangle. The tensor expressions are stresses and strains for structural mechanics applications with plane stress or plane strain conditions. `pdesmech` is intended to be used for postprocessing of a solution computed using the structural mechanics application modes of the `pdetool` GUI, after exporting the solution, the mesh, and the PDE coefficients to the MATLAB workspace. Poisson's ratio, `nu`, has to be supplied explicitly for calculations of shear stresses and strains, and for the von Mises effective stress in plane strain mode.

Valid property name/property value pairs include the following.

Property Name	Property Value/Default	Description
`tensor`	`ux|uy|vx|vy|exx|eyy|exy|sxx|syy|sxy|e1|e2|s1|s2|{von Mises}`	Tensor expression
`application`	`{ps}|pn`	Plane stress|plane strain
`nu`	Scalar or string expression {0.3}	Poisson's ratio

The available tensor expressions are

- $ux = \frac{\partial u}{\partial x}$
- $uy = \frac{\partial u}{\partial y}$
- $vx = \frac{\partial v}{\partial x}$
- $vy = \frac{\partial v}{\partial y}$
- `exx`, the x-direction strain (ε_x)

- `eyy`, the *y*-direction strain (ε_y)
- `exy`, the shear strain (γ_{xy})
- `sxx`, the *x*-direction stress (σ_x)
- `syy`, the *y*-direction stress (σ_y)
- `sxy`, the shear stress (τ_{xy})
- `e1`, the first principal strain (ε_1)
- `e2`, the second principal strain (ε_2)
- `s1`, the first principal stress (σ_1)
- `s2`, the second principal stress (σ_2)
- `von Mises`, the von Mises effective stress, for plane stress conditions

  $$\sqrt{\sigma_1^2 + \sigma_2^2 - \sigma_1\sigma_2}$$

  or for plane strain conditions

  $$\sqrt{(\sigma_1^2 + \sigma_2^2)(\upsilon^2 - \upsilon + 1) + \sigma_1\sigma_2(2\upsilon^2 - 2\upsilon - 1)}$$

Examples

Assuming that a problem has been solved using the application mode "Structural Mechanics, Plane Stress," discussed in "Structural Mechanics — Plane Stress" on page 1-86, and that the solution `u`, the mesh data `p` and `t`, and the PDE coefficient `c` all have been exported to the MATLAB workspace, the *x*-direction strain is computed as

```
sx=pdesmech(p,t,c,u,'tensor','sxx');
```

To compute the von Mises effective stress for a plane strain problem with Poisson's ratio equal to 0.3, type

```
mises=pdesmech(p,t,c,u,'tensor','von Mises',...
  'application','pn','nu',0.3);
```

pdesurf

Purpose

Shorthand command for surface plot

Syntax

```
pdesurf(p,t,u)
```

Description

`pdesurf(p,t,u)` plots a 3-D surface of PDE node or triangle data. If `u` is a column vector, node data is assumed, and continuous style and interpolated shading are used. If `u` is a row vector, triangle data is assumed, and discontinuous style and flat shading are used.

`h=pdesurf(p,t,u)` additionally returns handles to the drawn axes objects.

For node data, this command is just shorthand for the call

```
pdeplot(p,[],t,'xydata',u,'xystyle','interp',...
        'zdata',u,'zstyle','continuous',...
        'colorbar','off');
```

and for triangle data it is

```
pdeplot(p,[],t,'xydata',u,'xystyle','flat',...
        'zdata',u,'zstyle','discontinuous',...
        'colorbar','off');
```

If you want to have more control over your surface plot, use `pdeplot` instead of `pdesurf`.

Examples

Surface plot of the solution to the equation $-\Delta u = 1$ over the geometry defined by the L-shaped membrane. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$.

```
[p,e,t]=initmesh('lshapeg');
[p,e,t]=refinemesh('lshapeg',p,e,t);
u=assempde('lshapeb',p,e,t,1,0,1);
pdesurf(p,t,u)
```

See Also

`pdecont`	Partial Differential Equation Toolbox
`pdemesh`	Partial Differential Equation Toolbox
`pdeplot`	Partial Differential Equation Toolbox

Purpose

Open GUI

Syntax

```
pdetool
```

Description

`pdetool` provides the Partial Differential Equation Toolbox graphical user interface (GUI). Call `pdetool` without arguments to start the application. You should not call `pdetool` with arguments.

The GUI helps you to draw the 2-D domain and to define boundary conditions for a PDE problem. It also makes it possible to specify the partial differential equation, to create, inspect and refine the mesh, and to compute and display the solution from the GUI.

`pdetool` contains several different modes:

In draw mode, you construct a *Constructive Solid Geometry model* (CSG model) of the geometry. You can draw *solid objects* that can overlap. There are four types of solid objects:

- **Circle** object — represents the set of points inside a circle.
- **Polygon** object — represents the set of points inside the polygon given by a set of line segments.
- **Rectangle** object — represents the set of points inside the rectangle given by a set of line segments.
- **Ellipse** object — represents the set of points inside an ellipse. The ellipse can be rotated.

The solid objects can be moved and rotated. Operations apply to groups of objects by doing multiple selects. (A **Select all** option is also available.) You can cut and paste among the selected objects. The model can be saved and restored. `pdetool` can be started by just typing the name of the model. (This starts the corresponding file that contains the MATLAB commands necessary to create the model.)

The solid objects can be combined by typing a set formula. Each object is automatically assigned a unique name, which is displayed in the GUI on the solid object itself. The names refer to the object in the set

formula. More specifically, in the set formula, the name refers to the set of points inside the object. The resulting geometrical model is the set of points for which the set formula evaluates to true. (For a description of the syntax of the set formula, see `decsg`.) By default, the resulting geometrical model is the union of all objects.

A "snap-to-grid" function is available. This means that objects align to the grid. The grid can be turned on and off, and the scaling and the grid spacing can be changed.

In boundary mode, you can specify the boundary conditions. You can have different types of boundary conditions on different boundaries. In this mode, the original shapes of the solid building objects constitute borders between subdomains of the model. Such borders can be eliminated in this mode. The outer boundaries are color coded to indicate the type of boundary conditions. A red outer boundary corresponds to Dirichlet boundary conditions, blue to generalized Neumann boundary conditions, and green to mixed boundary conditions. You can return to the boundary condition display by clicking the $\partial\Omega$ button or by selecting **Boundary Mode** from the **Boundary** menu.

In PDE mode, you can specify the type of PDE problem, and the coefficients *c, a, f,* and *d*. You can specify the coefficients for each subdomain independently. This makes it easy to specify, e.g., various material properties in one PDE model. The PDE to be solved can be specified by clicking the **PDE** button or by selecting **PDE Specification** from the **PDE** menu. This brings up a dialog box.

In mesh mode, you can control the automated mesh generation and plot the mesh. An initial mesh can be generated by clicking the Δ button or by selecting **Initialize Mesh** from the **Mesh** menu. The initial mesh can be repeatedly refined by clicking the refine button or by selecting **Refine Mesh** from the **Mesh** menu.

In solve mode, you can specify solve parameters and solve the PDE. For parabolic and hyperbolic PDE problems, you can also specify the initial conditions, and the times at which the output should be generated. For eigenvalue problems, the search range can be specified. Also, the adaptive and nonlinear solvers for elliptic PDEs can be invoked. The

PDE problem is solved by clicking the **=** button or by selecting **Solve PDE** from the **Solve** menu. By default, the solution is plotted in the `pdetool` axes.

In plot mode, you can select a wide variety of visualization methods such as surface, mesh, contour, and quiver (vector field) plots. For surface plots, you can choose between interpolated and flat rendering schemes. The mesh can be hidden in all plot types. For parabolic and hyperbolic equations, you can animate the solution as it changes with time. You can show the solution both in 2-D and 3-D. 2-D plots are shown inside `pdetool`. 3-D plots are plotted in separate figure windows. Different types of plots can be selected by clicking the button with the solution plot icon or by selecting **Parameters** from the **Plot** menu. This opens a dialog box.

Boundary Condition Dialog Box

In this dialog box, the boundary condition for the selected boundaries is entered. The following boundary conditions can be handled:

- *Dirichlet:* $hu = r$ on the boundary.
- *Generalized Neumann:* $\vec{n} \cdot (c\nabla u) + qu = g$ on the boundary.
- *Mixed:* a combination of Dirichlet and generalized Neumann condition.

$\vec{n}$ is the outward unit length normal.

The boundary conditions can be entered in a variety of ways. (See `assemb` and "Boundary Menu" on page 2-12.)

PDE Specification Dialog Box

In this dialog box, the type of PDE and the PDE coefficients are entered. The following types of PDEs can be handled:

- Elliptic PDE: $-\nabla \cdot (c\nabla u) + au = f$
- Parabolic PDE: $d\frac{\partial u}{\partial t} - \nabla \cdot (c\nabla u) + au = f$

- Hyperbolic PDE: $d\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f$
- Eigenvalue PDE: $-\nabla \cdot (c\nabla u) + au = \lambda du$

for x and y on the problem's 2-D domain Ω.

The PDE coefficients can be entered in a variety of ways. (See `assempde` and "PDE Menu" on page 2-16.)

Model File

The *Model file* contains the MATLAB commands necessary to create a CSG model. It can also contain additional commands to set boundary conditions, define the PDE, create the mesh, solve the pde, and plot the solution. This type of file can be saved and opened from the **File** menu.

The Model file is a MATLAB function and not a script. This way name clashes between variables used in the function and in the main workspace are avoided. The name of the file must coincide with the model name. The beginning of the file always looks similar to the following code fragment:

```
function pdemodel

pdeinit;
pde_fig=gcf;
ax=gca;
pdetool('appl_cb',1);
setappdata(pde_fig,'currparam',...
   char('1.0','0.0','10.0','1.0'));
pdetool('snapon');
set(ax,'XLim',[-1.5 1.5]);
set(ax,'YLim',[-1 1]);
set(ax,'XTickMode','auto');
set(ax,'YTickMode','auto');
grid on;
```

The pdeinit command starts up pdetool. If pdetool has already been started, the current model is cleared. The following commands set up the scaling and tick marks of the axis of pdetool and other user parameters.

Then a sequence of drawing commands is issued. The commands that can be used are named pdecirc, pdeellip, pdepoly, and pderect. The following command sequence creates the L-shaped membrane as the union of three squares. The solid objects are given names SQ1, SQ2, SQ3, etc.

```
% Geometry description:
pderect([-1 0 0 -1],'SQ1');
pderect([0 1 0 -1],'SQ2');
pderect([0 1 1 0],'SQ3');
```

We do not intend to fully document the format of the Model file. It can be used to change the geometry of the drawn objects, since the pdecirc, pdeellip, pdepoly, and pderect commands are documented.

See Also

assempde	Partial Differential Equation Toolbox
initmesh	Partial Differential Equation Toolbox
parabolic	Partial Differential Equation Toolbox
pdecont	Partial Differential Equation Toolbox
pdeeig	Partial Differential Equation Toolbox
pdesurf	Partial Differential Equation Toolbox

pdetrg

Purpose

Triangle geometry data

Syntax

```
[ar,a1,a2,a3]=pdetrg(p,t)
[ar,g1x,g1y,g2x,g2y,g3x,g3y]=pdetrg(p,t)
```

Description

`[ar,a1,a2,a3]=pdetrg(p,t)` returns the area of each triangle in `ar` and half of the negative cotangent of each angle in `a1`, `a2`, and `a3`.

`[ar,g1x,g1y,g2x,g2y,g3x,g3y]=pdetrg(p,t)` returns the area and the gradient components of the triangle base functions.

The triangular mesh of the PDE problem is given by the mesh data `p` and `t`. For details on the mesh data representation, see `initmesh`.

Purpose Triangle quality measure

Syntax

```
q=pdetriq(p,t)
```

Description

`q=pdetriq(p,t)` returns a triangle quality measure given mesh data.

The triangular mesh is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

The triangle quality is given by the formula

$$q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2}$$

where a is the area and h_1, h_2, and h_3 the side lengths of the triangle.

If $q > 0.6$ the triangle is of acceptable quality. $q = 1$ when $h_1 = h_2 = h_3$.

Reference

Bank, Randolph E., *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, User's Guide 6.0,* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990.

See Also

`initmesh`	Partial Differential Equation Toolbox
`jigglemesh`	Partial Differential Equation Toolbox
`refinemesh`	Partial Differential Equation Toolbox

poiasma

Purpose

Boundary point matrix contributions for fast solvers of Poisson's equation

Syntax

```
K=poiasma(n1,n2,h1,h2)
K=poiasma(n1,n2)
K=poiasma(n)
```

Description

`K=poiasma(n1,n2,h1,h2)` assembles the contributions to the stiffness matrix from boundary points. `n1` and `n2` are the numbers of points in the first and second directions, and `h1` and `h2` are the mesh spacings. `K` is a sparse `n1*n2`-by-`n1*n2` matrix. The point numbering is the canonical numbering for a rectangular mesh.

`K=poiasma(n1,n2)` uses `h1=h2`.

`K=poiasma(n)` uses `n1=n2=n`.

See Also

`poiindex`	Partial Differential Equation Toolbox
`poisolv`	Partial Differential Equation Toolbox

Purpose

Fast solver for Poisson's equation on rectangular grid

Syntax

```
u=poicalc(f,h1,h2,n1,n2)
u=poicalc(f,h1,h2)
u=poicalc(f)
```

Description

`u=poicalc(f,h1,h2,n1,n2)` calculates the solution of Poisson's equation for the interior points of an evenly spaced rectangular grid. The columns of `u` contain the solutions corresponding to the columns of the right-hand side `f`. `h1` and `h2` are the spacings in the first and second direction, and `n1` and `n2` are the number of points.

The number of rows in `f` must be `n1*n2`. If `n1` and `n2` are not given, the square root of the number of rows of `f` is assumed. If `h1` and `h2` are not given, they are assumed to be equal.

The ordering of the rows in `u` and `f` is the canonical ordering of interior points, as returned by `poiindex`.

The solution is obtained by sine transforms in the first direction and tridiagonal matrix solution in the second direction. `n1` should be 1 less than a power of 2 for best performance.

See Also

`dst`	Partial Differential Equation Toolbox
`idst`	Partial Differential Equation Toolbox
`poiasma`	Partial Differential Equation Toolbox
`poiindex`	Partial Differential Equation Toolbox
`poisolv`	Partial Differential Equation Toolbox

poiindex

Purpose

Indices of points in canonical ordering for rectangular grid

Syntax

```
[n1,n2,h1,h2,i,c,ii,cc]=poiindex(p,e,t,sd)
```

Description

`[n1,n2,h1,h2,i,c,ii,cc]=poiindex(p,e,t,sd)` identifies a given grid `p`, `e`, `t` in the subdomain `sd` as an evenly spaced rectangular grid. If the grid is not rectangular, `n1` is 0 on return. Otherwise `n1` and `n2` are the number of points in the first and second directions, `h1` and `h2` are the spacings. `i` and `ii` are of length `(n1-2)*(n2-2)` and contain indices of interior points. `i` contains indices of the original mesh, whereas `ii` contains indices of the canonical ordering. `c` and `cc` are of length `n1*n2-(n1-2)*(n2-2)` and contain indices of border points. `ii` and `cc` are increasing.

In the canonical ordering, points are numbered from left to right and then from bottom to top. Thus if `n1=3` and `n2=5`, then `ii=[5 8 11]` and `cc=[1 2 3 4 6 7 9 10 12 13 14 15]`.

See Also

`poiasma`	Partial Differential Equation Toolbox
`poisolv`	Partial Differential Equation Toolbox

poimesh

Purpose

Make regular mesh on rectangular geometry

Syntax

```
[p,e,t]=poimesh(g,nx,ny)
[p,e,t]=poimesh(g,n)
[p,e,t]=poimesh(g)
```

Description

`[p,e,t]=poimesh(g,nx,ny)` constructs a regular mesh on the rectangular geometry specified by g, by dividing the “x edge” into `nx` pieces and the “y edge” into `ny` pieces, and placing `(nx+1)*(ny+1)` points at the intersections.

The “x edge” is the one that makes the smallest angle with the *x*-axis.

`[p,e,t]=poimesh(g,n)` uses `nx=ny=n`, and `[p,e,t]=poimesh(g)` uses `nx=ny=1`.

The triangular mesh is described by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

For best performance with `poisolv`, the larger of `nx` and `ny` should be a power of 2.

If `g` does not seem to describe a rectangle, `p` is zero on return.

Examples

Try the demo command `pdedemo8`. The solution of Poisson’s equation over a rectangular grid with boundary condition given by the file `squareb4` is returned. The solution time is compared to the usual Finite Element Method (FEM) approach.

See Also

`initmesh`	Partial Differential Equation Toolbox
`poisolv`	Partial Differential Equation Toolbox

poisolv

Purpose Fast solution of Poisson's equation on rectangular grid

Syntax

```
u=poisolv(b,p,e,t,f)
```

Description

`u=poisolv(b,p,e,t,f)` solves Poisson's equation with Dirichlet boundary conditions on a regular rectangular grid. A combination of sine transforms and tridiagonal solutions is used for increased performance.

The boundary conditions `b` must specify Dirichlet conditions for all boundary points.

The mesh `p`, `e`, and `t` must be a regular rectangular grid. For details on the mesh data representation, see `initmesh`.

`f` gives the right-hand side of Poisson's equation.

Apart from roundoff errors, the result should be the same as `u=assempde(b,p,e,t,1,0,f)`.

Reference

Strang, Gilbert, *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Cambridge, MA, 1986, pp. 453–458.

See Also

`poicalc`	Partial Differential Equation Toolbox
`poimesh`	Partial Differential Equation Toolbox

Purpose

Refine triangular mesh

Syntax

```
[p1,e1,t1]=refinemesh(g,p,e,t)
[p1,e1,t1]=refinemesh(g,p,e,t,'regular')
[p1,e1,t1]=refinemesh(g,p,e,t,'longest')
[p1,e1,t1]=refinemesh(g,p,e,t,it)
[p1,e1,t1]=refinemesh(g,p,e,t,it,'regular')
[p1,e1,t1]=refinemesh(g,p,e,t,it,'longest')
[p1,e1,t1,u1]=refinemesh(g,p,e,t,u)
[p1,e1,t1,u1]=refinemesh(g,p,e,t,u,'regular')
[p1,e1,t1,u1]=refinemesh(g,p,e,t,u,'longest')
[p1,e1,t1,u1]=refinemesh(g,p,e,t,u,it)
[p1,e1,t1,u1]=refinemesh(g,p,e,t,u,it,'regular')
[p1,e1,t1,u1]=refinemesh(g,p,e,t,u,it,'longest')
```

Description

`[p1,e1,t1]=refinemesh(g,p,e,t)` returns a refined version of the triangular mesh specified by the geometry `g`, Point matrix `p`, Edge matrix `e`, and Triangle matrix `t`.

The triangular mesh is given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see `initmesh`.

`[p1,e1,t1,u1]=refinemesh(g,p,e,t,u)` refines the mesh and also extends the function `u` to the new mesh by linear interpolation. The number of rows in `u` should correspond to the number of columns in `p`, and `u1` has as many rows as there are points in `p1`. Each column of `u` is interpolated separately.

An extra input argument `it` is interpreted as a list of subdomains to refine, if it is a row vector, or a list of triangles to refine, if it is a column vector.

The default refinement method is regular refinement, where all of the specified triangles are divided into four triangles of the same shape. Longest edge refinement, where the longest edge of each specified triangle is bisected, can be demanded by giving `longest` as a final parameter. Using `regular` as a final parameter results in regular refinement. Some triangles outside of the specified set may also be refined to preserve the triangulation and its quality.

Examples

Refine the mesh of the L-shaped membrane several times. Plot the mesh for the geometry of the L-shaped membrane.

```
[p,e,t]=initmesh('lshapeg','hmax',inf);
subplot(2,2,1), pdemesh(p,e,t)
[p,e,t]=refinemesh('lshapeg',p,e,t);
subplot(2,2,2), pdemesh(p,e,t)
[p,e,t]=refinemesh('lshapeg',p,e,t);
subplot(2,2,3), pdemesh(p,e,t)
[p,e,t]=refinemesh('lshapeg',p,e,t);
subplot(2,2,4), pdemesh(p,e,t)
subplot
```

Algorithm

The algorithm is described by the following steps:

1. Pick the initial set of triangles to be refined.
2. Either divide all edges of the selected triangles in half (regular refinement), or divide the longest edge in half (longest edge refinement).
3. Divide the longest edge of any triangle that has a divided edge.
4. Repeat step 3 until no further edges are divided.
5. Introduce new points of all divided edges, and replace all divided entries in `e` by two new entries.
6. Form the new triangles. If all three sides are divided, new triangles are formed by joining the side midpoints. If two sides are divided, the midpoint of the longest edge is joined with the opposing corner and with the other midpoint. If only the longest edge is divided, its midpoint is joined with the opposing corner.

See Also

`initmesh`	Partial Differential Equation Toolbox
`pdeent`	Partial Differential Equation Toolbox

`pdegeom`	Partial Differential Equation Toolbox
`pdesdt`	Partial Differential Equation Toolbox

sptarn

Purpose

Solve generalized sparse eigenvalue problem

Syntax

```
[xv,lmb,iresult] = sptarn(A,B,lb,ub)
[xv,lmb,iresult] = sptarn(A,B,lb,ub,spd)
[xv,lmb,iresult] = sptarn(A,B,lb,ub,spd,tolconv)
[xv,lmb,iresult] = sptarn(A,B,lb,ub,spd,tolconv,jmax)
[xv,lmb,iresult] = sptarn(A,B,lb,ub,spd,tolconv,jmax,maxmul)
```

Description

`[xv,lmb,iresult] = sptarn(A,B,lb,ub,spd,tolconv,jmax,maxmul)` finds eigenvalues of the *pencil* $(A - \lambda B)x = 0$ in interval $[lb,ub]$. (A matrix of linear polynomials $A_{ij} - \lambda B_{ij}$, $A - \lambda B$, is called a *pencil*.)

`A` and `B` are sparse matrices. `lb` and `ub` are lower and upper bounds for eigenvalues to be sought. We may have `lb=-inf` if all eigenvalues to the left of `ub` are sought, and `rb=inf` if all eigenvalues to the right of `lb` are sought. One of `lb` and `ub` must be finite. A narrower interval makes the algorithm faster. In the complex case, the real parts of lmb are compared to `lb` and `ub`.

`xv` are eigenvectors, ordered so that `norm(a*xv-b*xv*diag(lmb))` is small. `lmb` is the sorted eigenvalues. If `iresult>=0` the algorithm succeeded, and all eigenvalues in the intervals have been found. If `iresult<0` the algorithm has not yet been successful, there may be more eigenvalues—try with a smaller interval.

`spd` is `1` if the pencil is known to be symmetric positive definite (default `0`).

`tolconv` is the expected relative accuracy. Default is `100*eps`, where `eps` is the machine precision.

`jmax` is the maximum number of basis vectors. The algorithm needs `jmax*n` working space so a small value may be justified on a small computer, otherwise let it be the default value `jmax=100`. Normally the algorithm stops earlier when enough eigenvalues have converged.

`maxmul` is the number of Arnoldi runs tried. Must at least be as large as maximum multiplicity of any eigenvalue. If a small value of `jmax` is

given, many Arnoldi runs are necessary. The default value is `maxmul=n`, which is needed when all the eigenvalues of the unit matrix are sought.

Algorithm

The *Arnoldi algorithm* with spectral transformation is used. The shift is chosen at `ub`, `lb`, or at a random point in interval (`lb`,`ub`) when both bounds are finite. The number of steps `j` in the Arnoldi run depends on how many eigenvalues there are in the interval, but it stops at `j=min(jmax,n)`. After a stop, the algorithm restarts to find more Schur vectors in orthogonal complement to all those already found. When no more eigenvalues are found in `lb < lmb <= ub`, the algorithm stops. For small values of `jmax`, several restarts may be needed before a certain eigenvalue has converged. The algorithm works when `jmax` is at least one larger than the number of eigenvalues in the interval, but then many restarts are needed. For large values of `jmax`, which is the preferred choice, `mul+1` runs are needed. `mul` is the maximum multiplicity of an eigenvalue in the interval.

Note The algorithm works on nonsymmetric as well as symmetric pencils, but then accuracy is approximately `tol` times the Henrici departure from normality. The parameter `spd` is used only to choose between `symamd` and `colamd` when factorizing, the former being marginally better for symmetric matrices close to the lower end of the spectrum.

In case of trouble,

If convergence is too slow, try (in this order of priority):

- a smaller interval `lb`, `ub`
- a larger `jmax`
- a larger `maxmul`

If factorization fails, try again with `lb` or `ub` finite. Then shift is chosen at random and hopefully not at an eigenvalue. If it fails again, check whether pencil may be singular.

If it goes on forever, there may be too many eigenvalues in the strip. Try with a small value `maxmul=2` and see which eigenvalues you get. Those you get are some of the eigenvalues, but a negative `iresult` tells you that you have not gotten them all.

If memory overflow, try smaller `jmax`.

The algorithm is designed for eigenvalues close to the real axis. If you want those close to the imaginary axis, try `A=i*A`.

When `spd=1`, the shift is at `lb` so that advantage is taken of the faster factorization for symmetric positive definite matrices. No harm is done, but the execution is slower if `lb` is above the lowest eigenvalue.

References

[1] Golub, Gene H., and Charles F. Van Loan, *Matrix Computations*, 2nd edition, Johns Hopkins University Press, Baltimore, MD, 1989.

[2] Saad, Yousef, "Variations on Arnoldi's Method for Computing Eigenelements of Large Unsymmetric Matrices," *Linear Algebra and its Applications*, Vol. 34, 1980, pp. 269–295.

See Also

`pdeeig`	Partial Differential Equation Toolbox

Purpose

Interpolate from PDE triangular mesh to rectangular grid

Syntax

```
uxy=tri2grid(p,t,u,x,y)
[uxy,tn,a2,a3]=tri2grid(p,t,u,x,y)
uxy=tri2grid(p,t,u,tn,a2,a3)
```

Description

`uxy=tri2grid(p,t,u,x,y)` computes the function values `uxy` over the grid defined by the vectors `x` and `y`, from the function `u` with values on the triangular mesh defined by `p` and `t`. Values are computed using linear interpolation in the triangle containing the grid point. The vectors `x` and `y` must be increasing.

`[uxy,tn,a2,a3]=tri2grid(p,t,u,x,y)` additionally lists the index `tn` of the triangle containing each grid point, and interpolation coefficients `a2` and `a3`.

`uxy=tri2grid(p,t,u,tn,a2,a3)` with `tn`, `a2`, and `a3` computed in an earlier call to `tri2grid`, interpolates using the same grid as in the earlier call. This variant is, however, much faster if several functions have to be interpolated using the same grid.

For grid points outside of the triangular mesh, `NaN` is returned in `uxy`, `tn`, `a2`, and `a3`.

See Also

`assempde`	Partial Differential Equation Toolbox
`initmesh`	Partial Differential Equation Toolbox
`refinemesh`	Partial Differential Equation Toolbox

Purpose Write boundary condition specification file

Syntax

```
fid=wbound(bl,mn)
```

Description

`fid=wbound(bl,mn)` writes a Boundary file with the name `[mn,'.m']`. The Boundary file is equivalent to the Boundary Condition matrix `bl.fid` returns -1 if the file could not be written.

`bl` describes the boundary conditions of the PDE problem. `bl` is a Boundary Condition matrix. For details, see `assemb`.

The output file `[mn,'.m']` is the name of a Boundary file. (See `pdebound`.)

See Also

`decsg`	Partial Differential Equation Toolbox
`pdebound`	Partial Differential Equation Toolbox
`pdegeom`	Partial Differential Equation Toolbox
`wgeom`	Partial Differential Equation Toolbox

wgeom

Purpose Write geometry specification function

Syntax `fid=wgeom(dl,mn)`

Description `fid=wgeom(dl,mn)` writes a Geometry file with the name `[mn,'.m']`. The Geometry file is equivalent to the Decomposed Geometry matrix `dl`. `fid` returns -1 if the file could not be written.

`dl` is a Decomposed Geometry matrix. For a description of the format of the Decomposed Geometry matrix, see `decsg`.

The output file `[mn,'.m']` is the name of a Geometry file. For a description of the Geometry file format, see `pdegeom`.

See Also

`decsg`	Partial Differential Equation Toolbox
`pdegeom`	Partial Differential Equation Toolbox
`wbound`	Partial Differential Equation Toolbox

Index

A

AC power electromagnetics 1-103
adaptive mesh refinement 1-26
`adaptmesh` function 5-2
animation 1-67
Application command 2-9
application modes 1-84
Armijo-Goldstein line search 3-24
Arnoldi algorithm 5-109
`assema` function 5-9
`assemb` function 5-10
assembling 3-6
`assempde` function 5-18
Axes Limits command 2-9

B

Boolean table 5-32
border segment 5-34
Boundary Condition matrix 1-45
boundary conditions 2-14
Boundary file 1-45
Boundary menu 2-12
boundary mode 1-32
boundary segment 5-34

C

circle solid 5-33
Coefficient file 1-45
Coefficient matrix 1-45
command-line functions 1-42
conductive media DC 1-108
Constructive Solid Geometry model 1-32
CSG model 1-32
`csgchk` function 5-29
`csgdel` function 5-31
cylindrical problem 1-69

D

decomposed geometry 1-33
Decomposed Geometry matrix 1-45
`decsg` function 5-32
discrete sine transform 3-32
domain decomposition 1-60
Draw menu 2-10
draw mode 1-31

E

Edge matrix 1-46
Edit menu 2-5
eigenmodes 1-81
eigenvalue equation 3-19
eigenvalue problems 1-25
eigenvector matrix 5-63
electrostatics 1-93
ellipse solid 5-34
elliptic equation 1-3
elliptic problems 1-49
elliptic system 3-10
energy norm 3-5

F

fast solver 3-32
FEM 1-22
File menu 2-3
Finite Element Method 3-1

G

Gauss-Newton method 3-24
Geometry Description matrix 1-45
Geometry file 1-45
graphical user interface 1-10
Green’s formula 1-23
Grid Spacing command 2-8

H

heat distribution in radioactive rod 1-69
heat equation 1-64
heat transfer 1-114
Helmholtz's equation 1-54
Help menu 2-34
hyperbolic equation 1-3
`hyperbolic` function 5-40
hyperbolic problems 1-71

I

`idst` function 5-38
`initmesh` function 5-43

J

`jigglemesh` function 5-48

L

L-shaped membrane 1-75
Laplace equation 1-94

M

magnetostatics 1-96
mass matrix 3-5
Maxwell's equations 1-93
mesh data 1-46
Mesh menu 2-20
mesh mode 1-32
mesh parameters 2-21
method of lines 3-14
minimal region 1-45
minimal surface problem 1-58
Model file 1-41

N

Name Space matrix 1-45
New command 2-3
node data 5-74
nonlinear
 equation 3-23
 problem 1-58
 solver 1-26

O

Open command 2-3
Options menu 2-7

P

parabolic equation 1-3
`parabolic` function 5-50
parabolic problems 1-64
Paste command 2-6
PDE coefficients 2-17
PDE menu 2-16
PDE mode 1-32
PDE Specification 2-17
`pdeadgsc` function 5-53
`pdeadworst` function 5-54
`pdearcl` function 5-55
`pdebound` function 5-56
`pdecgrad` function 5-59
`pdecirc` function 5-60
`pdecont` function 5-61
`pdeeig` function 5-63
`pdeellip` function 5-66
`pdeent` function 5-67
`pdegeom` function 5-68
`pdegplot` function 5-72
`pdegrad` function 5-73
`pdeintrp` function 5-74
`pdejmps` function 5-75
`pdemdlcv` function 5-76
`pdemesh` function 5-77
`pdenonlin` function 5-79

`pdeplot` function 5-82
`pdepoly` function 5-85
`pdeprtni` function 5-86
`pderect` function 5-87
`pdesde` function 5-88
`pdesdp` function 5-88
`pdesdt` function 5-88
`pdesmech` function 5-89
`pdesurf` function 5-91
`pdetool` function 5-93
`pdetrg` function 5-98
`pdetriq` function 5-99
pencil 5-108
plane strain 1-92
plane stress 1-86
Plot menu 2-27
plot mode 1-32
Plot parameters 2-28
`poiasma` function 5-100
`poicalc` function 5-101
`poiindex` function 5-102
`poimesh` function 5-103
Point matrix 1-46
`poisolv` function 5-104
Poisson's equation 1-10
polygon solid 5-34
Print command 2-4

R

rectangle solid 5-34
`refinemesh` function 5-105
Robin boundary condition 3-2
Rotate command 2-12

S

Save As command 2-4
scattering problem 1-53
Schur vector 3-21
set formulas 1-6
skin effect 1-105
solid object 1-11
solution vector 1-46
Solve menu 2-22
solve mode 1-32
solve parameters 2-23
`sptarn` function 5-108
stiff springs 3-11
structural mechanics 1-86

T

test function 1-23
toolbar 1-30
`tri2grid` function 5-112
Triangle matrix 1-46
triangle quality 2-20

V

von Mises effective stress 1-88

W

wave equation 1-71
`wbound` function 5-113
weak form 3-3
`wgeom` function 5-114
Window menu 2-34