Partial Differential Equation Toolbox™
User's Guide

R2012b

MATLAB®

How to Contact 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 1995–2012 The MathWorks, Inc.

Trademarks

Patents

Revision History

Contents

Solving PDEs

3

Graphical User Interface

4

Finite Element Method

5

Function Reference

6

Functions — Alphabetical List

7

Index

1

Getting Started

Product Description

Solve partial differential equations using finite element methods

The Partial Differential Equation Toolbox™ product contains tools for the study and solution of partial differential equations (PDEs) in two-space dimensions (2-D) and time. A set of command-line functions and a graphical user interface let you preprocess, solve, and postprocess generic 2-D PDEs for a broad range of engineering and science applications.

Key Features

- Complete GUI for pre- and post-processing 2-D PDEs

- Automatic and adaptive meshing

- Geometry creation using constructive solid geometry (CSG) paradigm

- Boundary condition specification: Dirichlet, generalized Neumann, and mixed

- Flexible coefficient and PDE problem specification using MATLAB® syntax

- Fully automated mesh generation and refinement

- Nonlinear and adaptive solvers handle systems with multiple dependent variables

- Simultaneous visualization of multiple solution properties, FEM-mesh overlays, and animation

Prerequisite Knowledge for Using This Toolbox

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. "Poisson's Equation with Complex 2-D Geometry" on page 1-12 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. You can use functions to, for example, generate meshes, discretize your problem, interpolate, and plot data on unstructured grids.

Types of PDE Problems You Can Solve

This toolbox applies to the following PDE type:

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

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, on the solution *u*, and on its gradient ∇u. A nonlinear solver (`pdenonlin`) 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* and on its gradient ∇u. The parabolic and hyperbolic equation solvers also solve nonlinear and time-dependent problems.

> **Note** Before solving a nonlinear elliptic 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 of *N* coupled equations. You can solve *N* = 1 or 2 equations using pdetool, and any number of equations using command-line functions. For example, *N* = 2 elliptic equations:

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

For a description of *N* > 1 PDE systems and their coefficients, see "Coefficients for Systems of PDEs" on page 2-38.

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.

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 "Solve PDEs Programmatically" on page 3-108.

Common Toolbox Applications

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
- Steady state of wave equations

Hyperbolic equation is used for:

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

Eigenvalue problems are used for:

- Determining natural vibration states in membranes and structural mechanics problems

In addition to solving generic scalar PDEs and generic systems of PDEs with vector valued u, Partial Differential Equation Toolbox provides tools for solving PDEs that occur in these common applications in engineering and science:

- "Structural Mechanics — Plane Stress" on page 3-6
- "Structural Mechanics — Plane Strain" on page 3-13
- "Electrostatics" on page 3-34
- "Magnetostatics" on page 3-37
- "AC Power Electromagnetics" on page 3-44
- "Conductive Media DC" on page 3-50
- "Heat Transfer" on page 3-57
- "Diffusion" on page 3-70

The PDE Toolbox GUI lets you specify PDE coefficients and boundary conditions in terms of physical entities. For example, you can specify Young's modulus in structural mechanics problems.

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.

The PDE Toolbox GUI lets you solve problems with vector valued u of dimension two. However, you can use functions to solve problems for any dimension of u.

Typical Steps to Solve PDEs

Partial Differential Equation Toolbox provides the PDE Toolbox GUI that you can use to:

1 Define the 2-D geometry.

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*.

2 Define the boundary conditions.

You can have different types of boundary conditions on different boundary segments. See "Types of Boundary Conditions" on page 2-56.

3 Define the PDE coefficients. See "Scalar PDE Coefficients" on page 2-14 and "Coefficients for Systems of PDEs" on page 2-38.

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.

4 Create the triangular mesh.

Generate the mesh to a fineness that adequately resolves the important features in the geometry, but is coarse enough to run in a reasonable amount of time and memory.

5 Solve the PDE.

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.

6 Plot the solution and other physical properties calculated from the solution (post processing).

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, `adaptmesh`. This option tries to find a mesh that fits the solution.

Visualize and Animate Solutions

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.

Poisson's Equation with Complex 2-D Geometry

This example shows how to solve the Poisson's equation, $-\Delta u = f$ using `pdetool`. This problem requires configuring a 2-D geometry with Dirichlet and Neumann boundary conditions.

To start the GUI, type the command `pdetool` at the MATLAB prompt. 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.
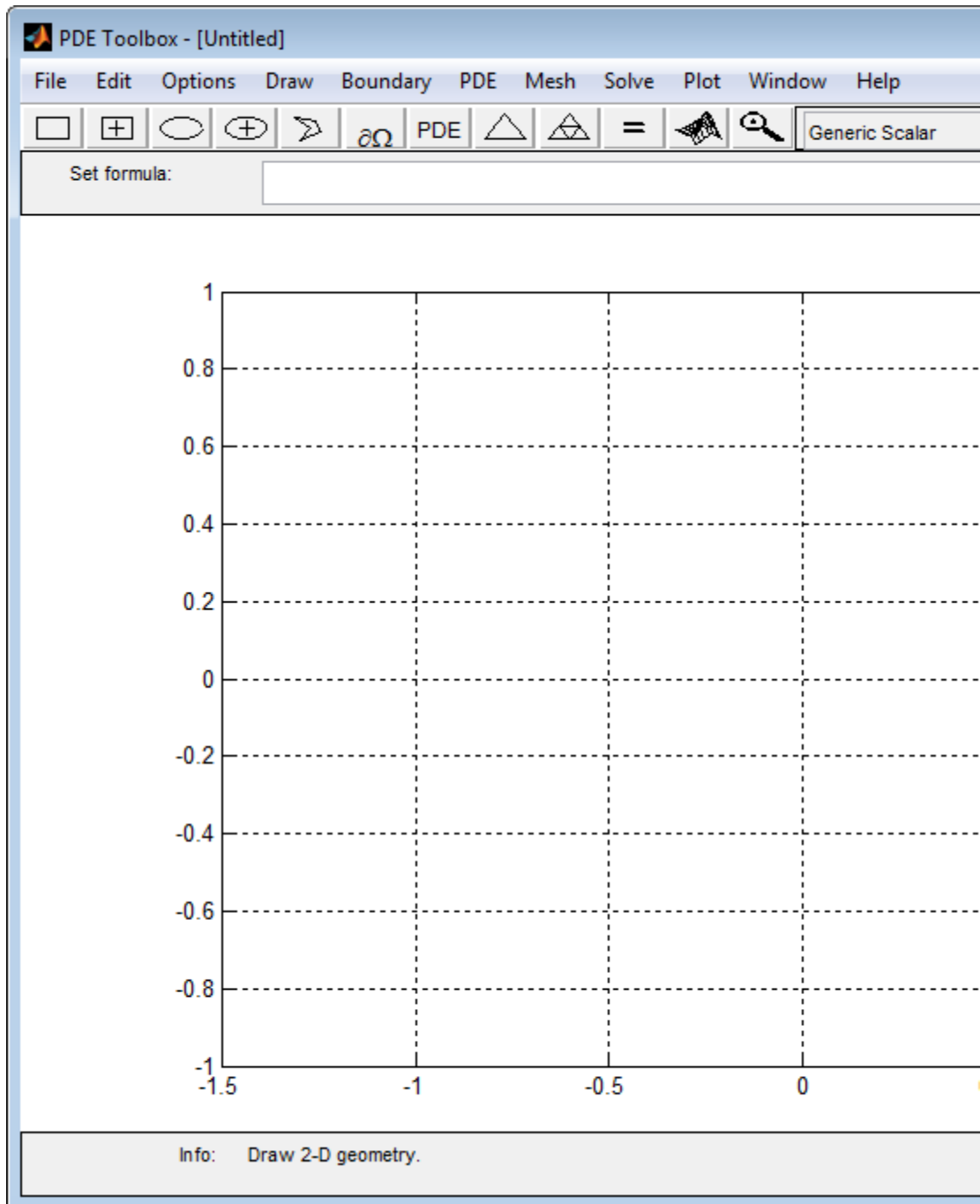
The first step is to draw the geometry on which you want to solve the PDE. The GUI provides four basic types of *solid objects*: polygons, rectangles, circles, and ellipses. The objects are used to create a *Constructive Solid Geometry model* (CSG model). Each solid object is assigned a unique label, 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.) Click and drag from (–1,.2) to (1,–.2). 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, starting near the point (–.5,0) with radius .4, using the *right* mouse button, starting at the circle center.
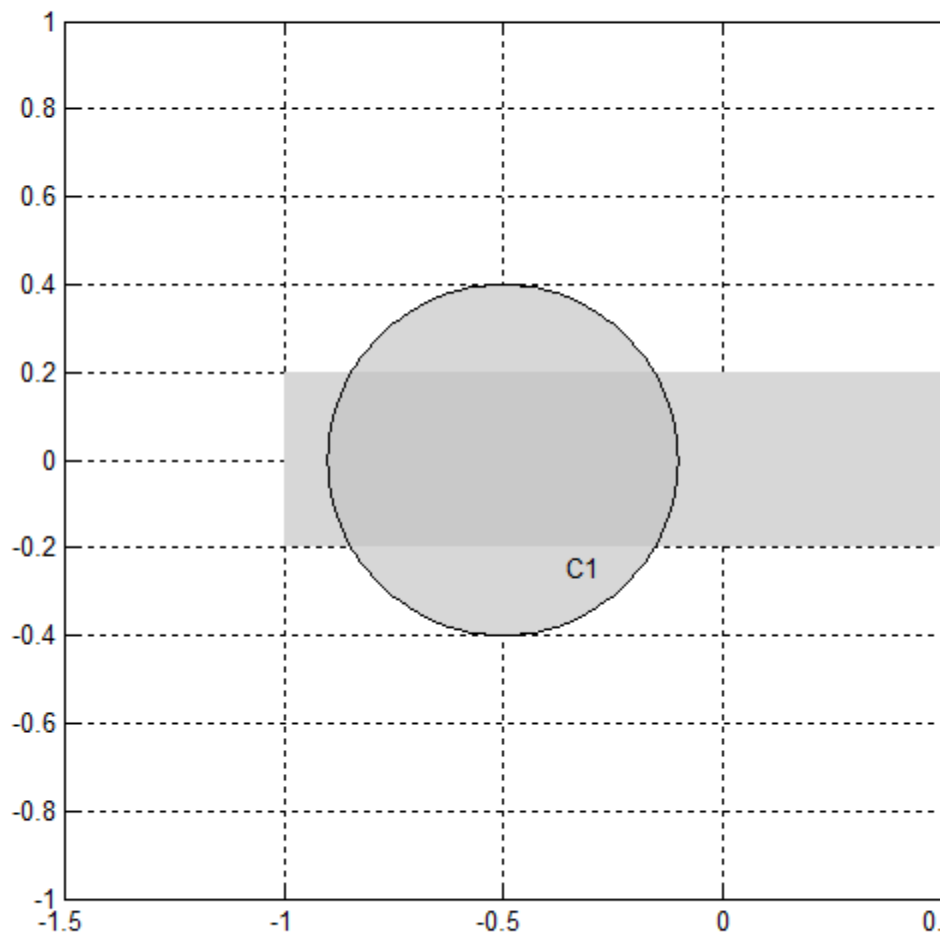
The resulting CSG model is the union of the rectangle R1 and the circle C1, described by set algebra as R1+C1. The area where the two objects overlap is clearly visible as it is drawn using a darker shade of gray. The object that you just drew—the circle—has a black border, indicating that it is selected. 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 from (.5,–.6) to (1,1), and a circle C2 centered at (.5,.2) with radius .2. 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
```
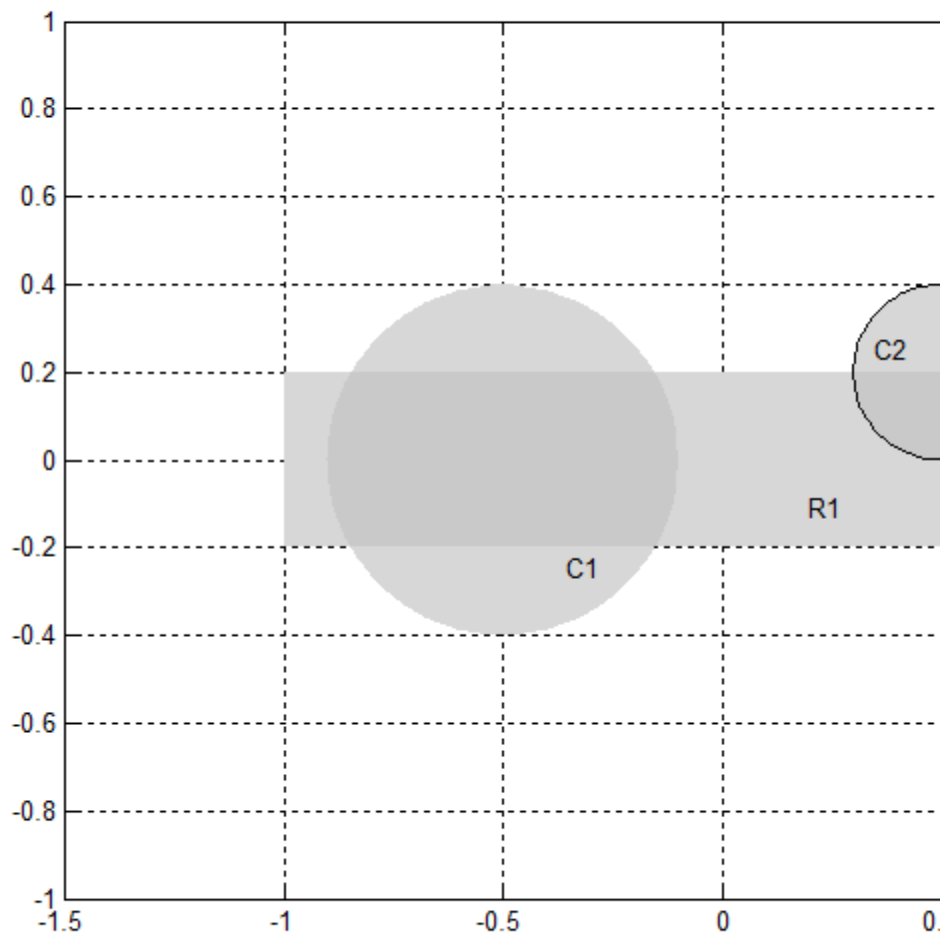
If you want, you can save this CSG model as a file. Use the **Save As** option from the **File** menu, and enter a filename of your choice. It is good practice to continue to save your model at regular intervals using **Save**. All the additional steps in the process of modeling and solving your PDE 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.
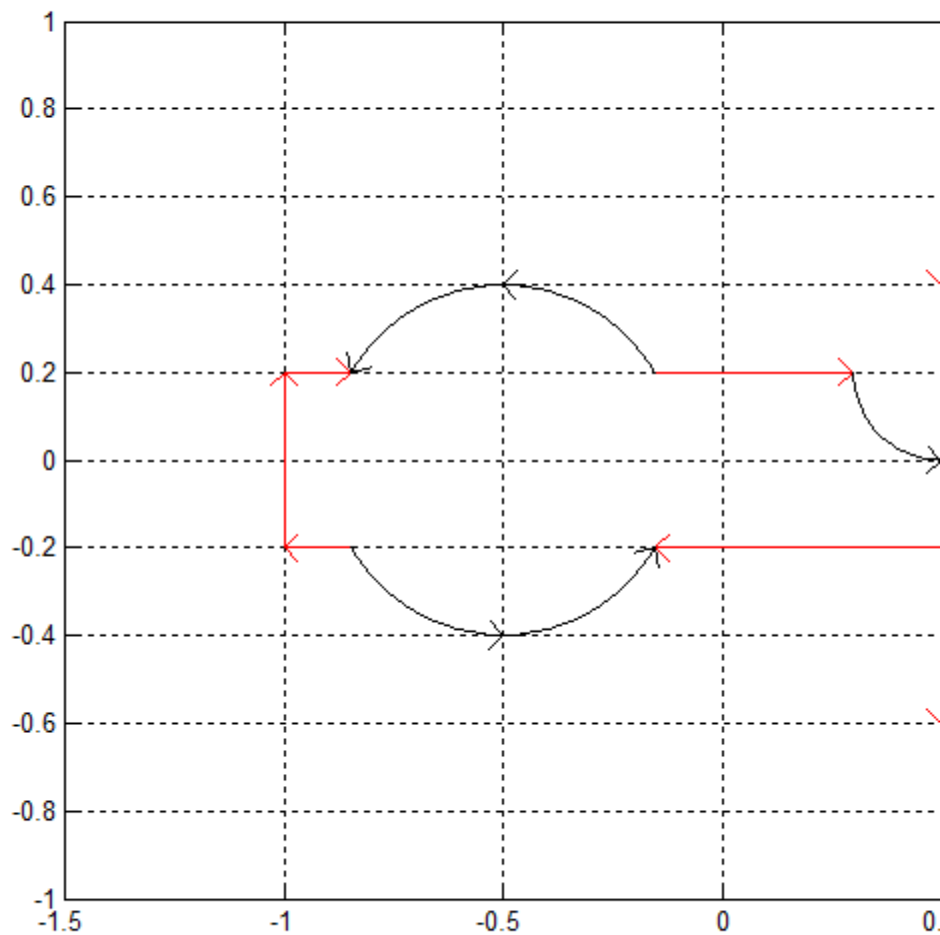
Double-clicking anywhere on the selected boundary segments opens the Boundary Condition dialog box. Here, you select the type of boundary condition, and enter the boundary condition as a MATLAB expression. Change the boundary condition along the selected boundaries to a Neumann condition, $\partial u/\partial n = -5$. 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.
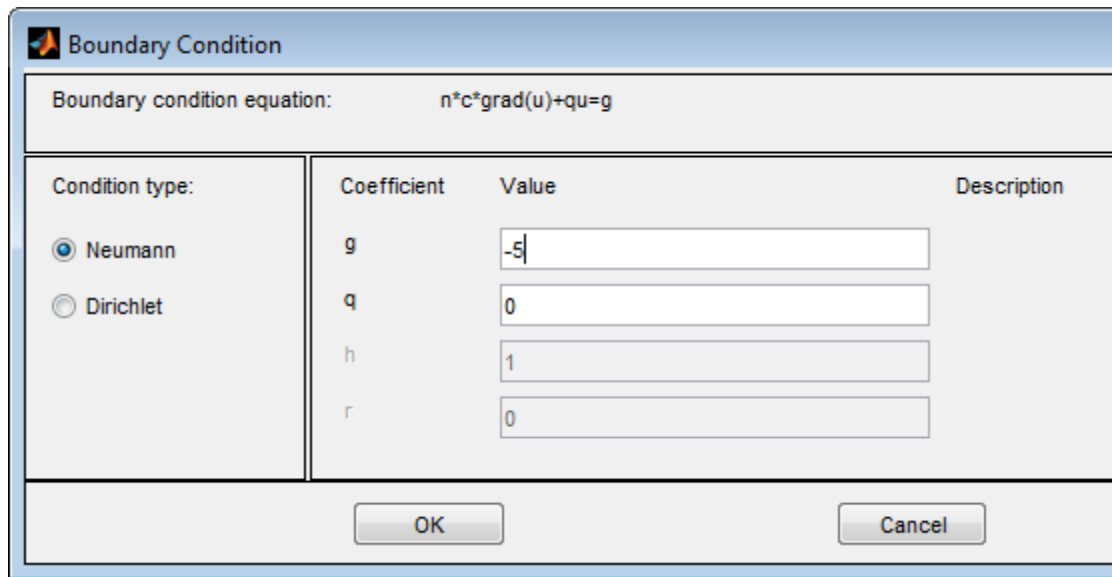
Next, specify the PDE itself through a dialog box that is accessed by clicking the button with the **PDE** icon or by selecting **PDE Specification** from the **PDE** menu. In PDE mode, you can also access the PDE Specification dialog box by double-clicking a subdomain. That way, different subdomains 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 $f = 10.0$.
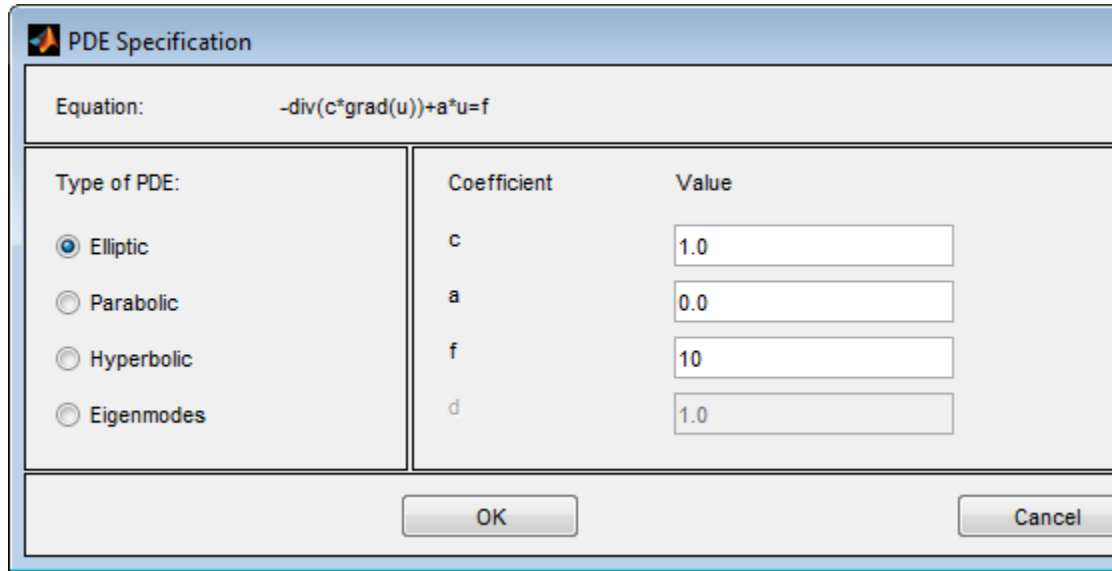
Finally, create the triangular mesh that Partial Differential Equation Toolbox software uses in the Finite Element Method (FEM) to solve the PDE. The triangular mesh is created and displayed when clicking the button with the

△ 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.
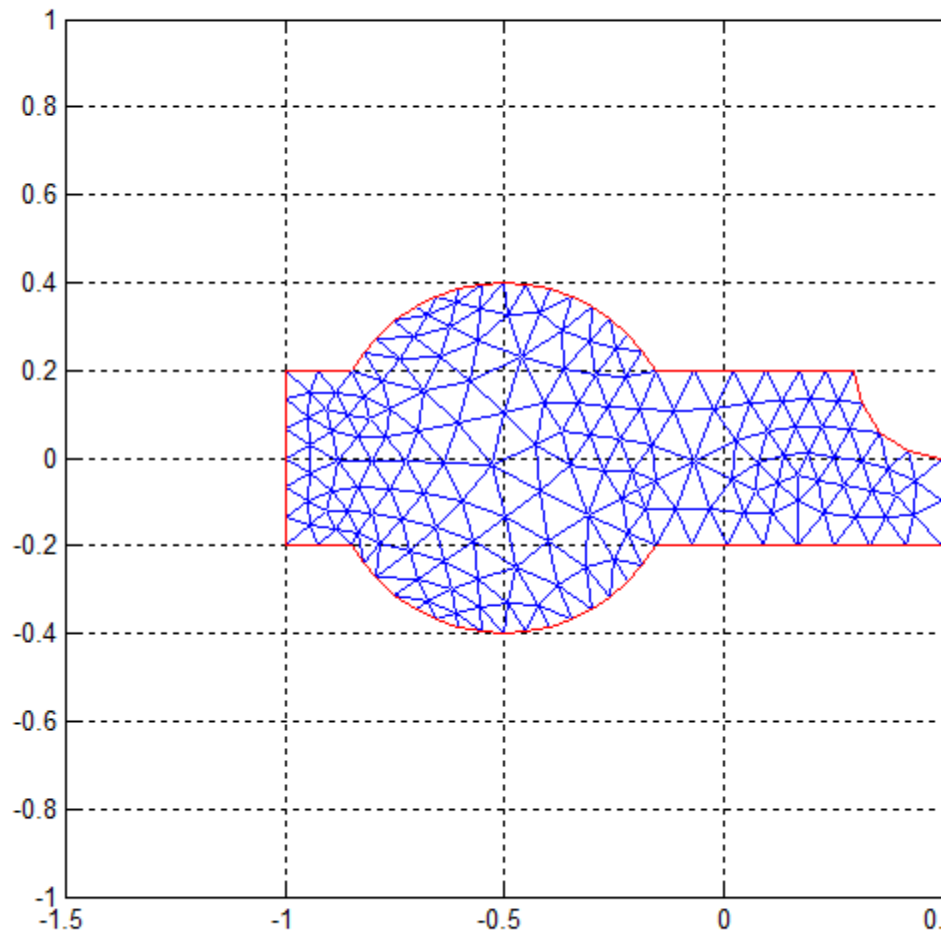
We are now ready to solve the problem. Click the **=** button or select **Solve PDE** from the **Solve** menu to solve the PDE. The solution is then plotted. By default, the plot uses interpolated coloring and a linear color map. A color bar is also provided to map the different shades to the numerical values of the solution. If you want, the solution can be exported as a vector to the MATLAB main workspace.
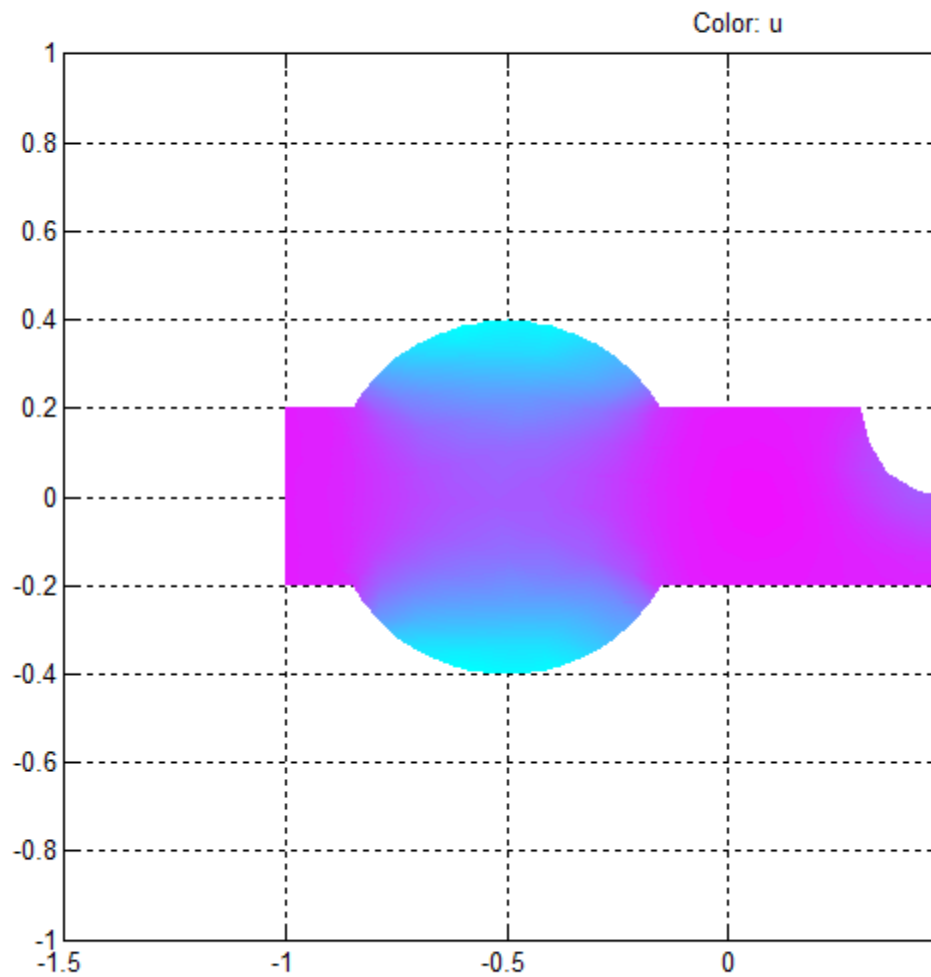
There are many more plot modes available to help you visualize the solution. Click the button with the 3-D solution icon or select **Parameters** from the **Plot** menu to access the dialog box for selection of the different plot options. Several plot styles are available, and the solution can be plotted in the GUI or in a separate figure as a 3-D plot.
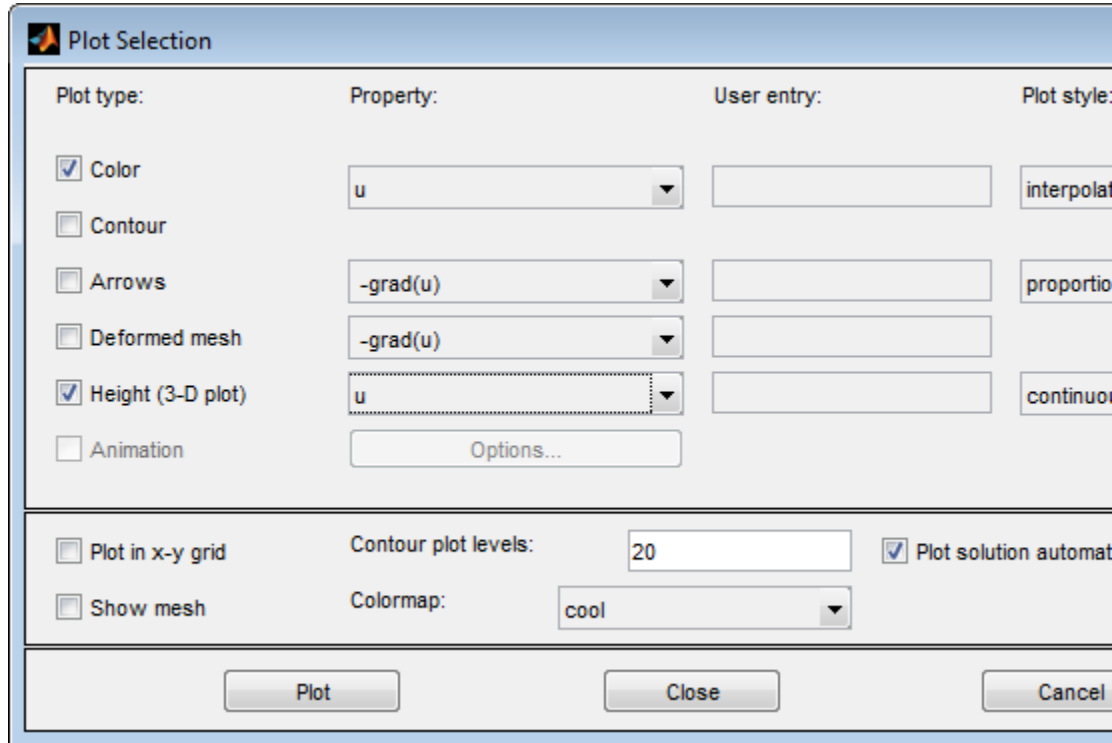
Now, select a plot where the color and the height both represent u. Choose interpolated shading and use the continuous (interpolated) height option. The default colormap is the cool colormap; a pop-up menu lets you select from a number of different colormaps. Finally, click the **Plot** button to plot the solution; click the **Close** 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.

Color: u Height: u
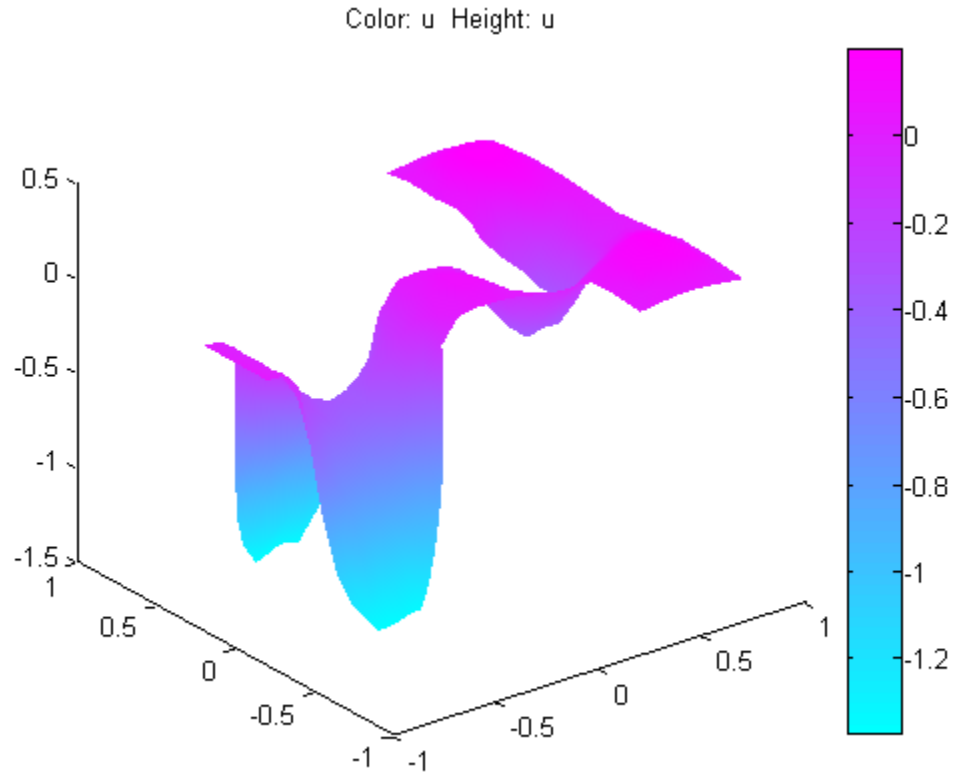
PDE Toolbox GUI Shortcuts

PDE Toolbox GUI toolbar provide quick access to key operations that are also available in the menus.

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 the rightmost button for activating the zoom feature.
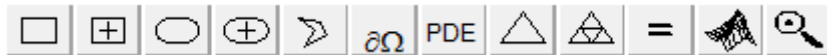
Five buttons on the left let you draw the geometry. Double-click a button makes it "stick," and you can then continue to draw solid objects of the selected type until you single-click the button to "release" it.

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.

	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 remaining buttons represent, from left to right:

∂Ω	Enters the boundary mode. 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.
PDE	Opens the PDE Specification dialog box. 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.
△	Initializes the triangular mesh In mesh mode, you can control the automated mesh generation and plot the mesh.
⚠	Refines the triangular mesh.
=	Solves the PDE. 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.

	3-D solution opens the Plot Selection dialog box.
	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.
	Toggles zoom.

Solving 3-D Problems Using 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.

Finite Element Method (FEM) Basics

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.

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 "Elliptic Equations" on page 5-2, with variants for specific types in "Systems of PDEs" on page 5-10, "Parabolic Equations" on page 5-13, "Hyperbolic Equations" on page 5-18, "Eigenvalue Equations" on page 5-19, and "Nonlinear Equations" on page 5-24.

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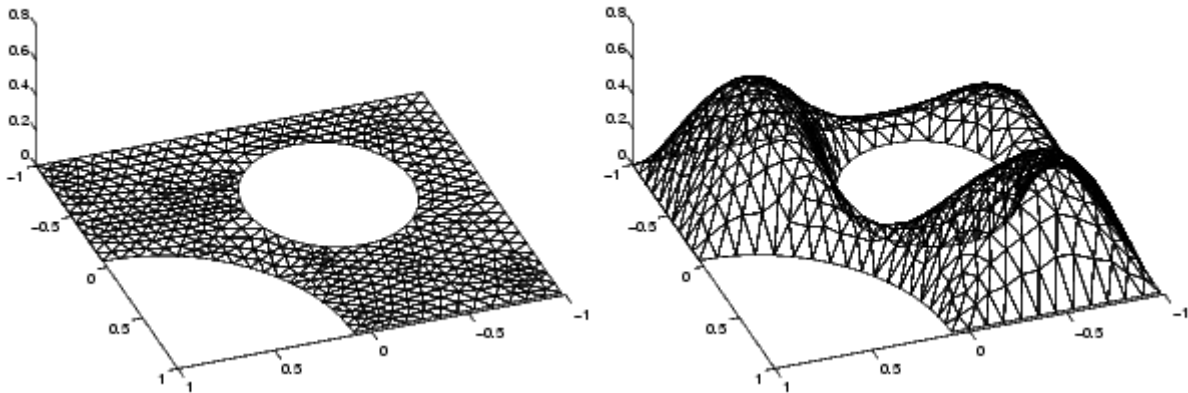
A Triangular Mesh (left) and a Continuous Piecewise Linear Function on That Mesh

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

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

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

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

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

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., $(c\nabla u_h) \cdot \vec{n} + qu_h = g$. The FEM formulation reads: Find u_h such that

$$\int_\Omega \left((c\nabla u_h) \cdot \nabla v + au_h v\right)dx + \int_{\partial\Omega_1} qu_h v\,ds = \int_\Omega fv\,dx + \int_{\partial\Omega_1} gv\,ds \quad \forall v,$$

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 $v = \varphi_i$, $i = 1, 2, \dots, N$? 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. The function 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)

$$M\frac{dU}{dt} + KU = F,$$

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

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

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 or ∇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. For elliptic equations, the toolbox provides a nonlinear solver called `pdenonlin` using a damped Gauss-Newton method. The `parabolic` and `hyperbolic` functions call the nonlinear solver automatically.

- 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.

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

Setting Up Your PDE

- "Boundary Conditions for PDE Systems" on page 2-68
- "Tooltip Displays for Mesh and Plots" on page 2-75
- "Mesh Data" on page 2-76
- "Adaptive Mesh Refinement" on page 2-77

Open the PDE Toolbox GUI

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.
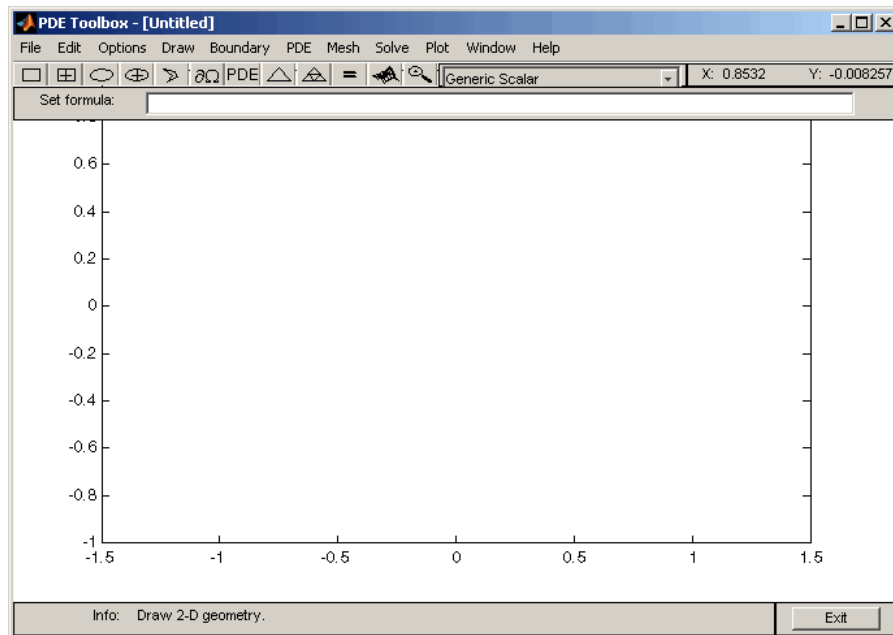
At the top, the GUI has a pull-down menu bar that you use to control the modeling. Below the menu bar, a toolbar with icon buttons provide quick and easy access to some of the most important functions.

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. This position 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, and plot the solution.

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.

Specify Geometry Using a CSG Model

You can specify complex geometries by overlapping solid objects. This approach to representing geometries is called Constructive Solid Geometry (CSG).

Use these four solid objects to specify a geometry for your problem:

- **Circle** — Represents the set of points inside and on a circle.

- **Polygon** — Represents the set of points inside and on a polygon given by a set of line segments.

- **Rectangle** — Represents the set of points inside and on a rectangle.

- **Ellipse** — Represents the set of points inside and on an ellipse. The ellipse can be rotated.

When you draw a solid object in the GUI, each solid object is automatically given a unique name. 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.
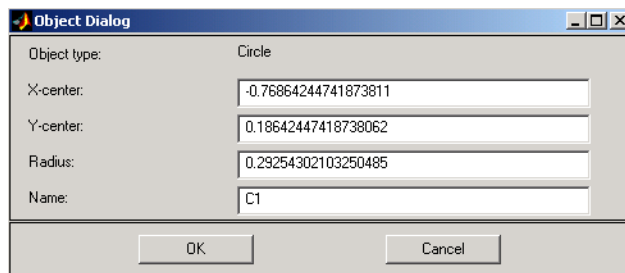
You can use the name of the object to refer to the corresponding set of points in a set formula. The operators +, *, and – are used to form the set of points Ω in the plane over which the differential equation is solved. The operators +, the set union operator, and *, the set intersection operator, have 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*.

Select Graphical Objects Representing Your Geometry

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.

Rounded Corners Using CSG Modeling

This example shows how to represent a geometry that includes rounded corners (fillets) using Constructive Solid Geometry (CSG) modeling. You learn how to draw several overlapping solid objects, and specify how these objects should combine to produce the desired geometry.

Start the GUI using `pdetool` 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 0.2.

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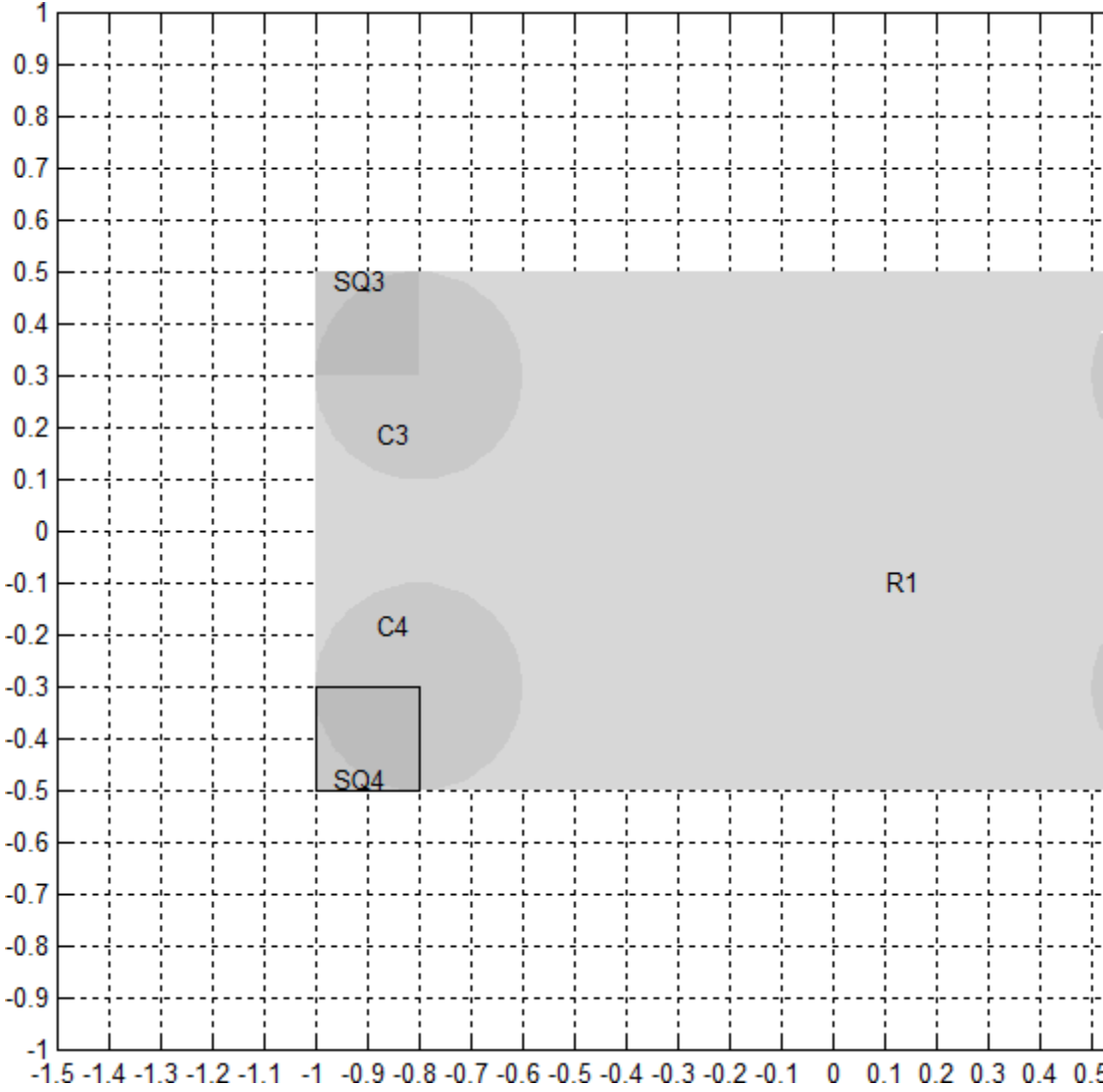
Now you have to edit the set formula. To get the rounded corners, subtract the small squares from the rectangle and then add the circles. As a set formula, this is expressed as

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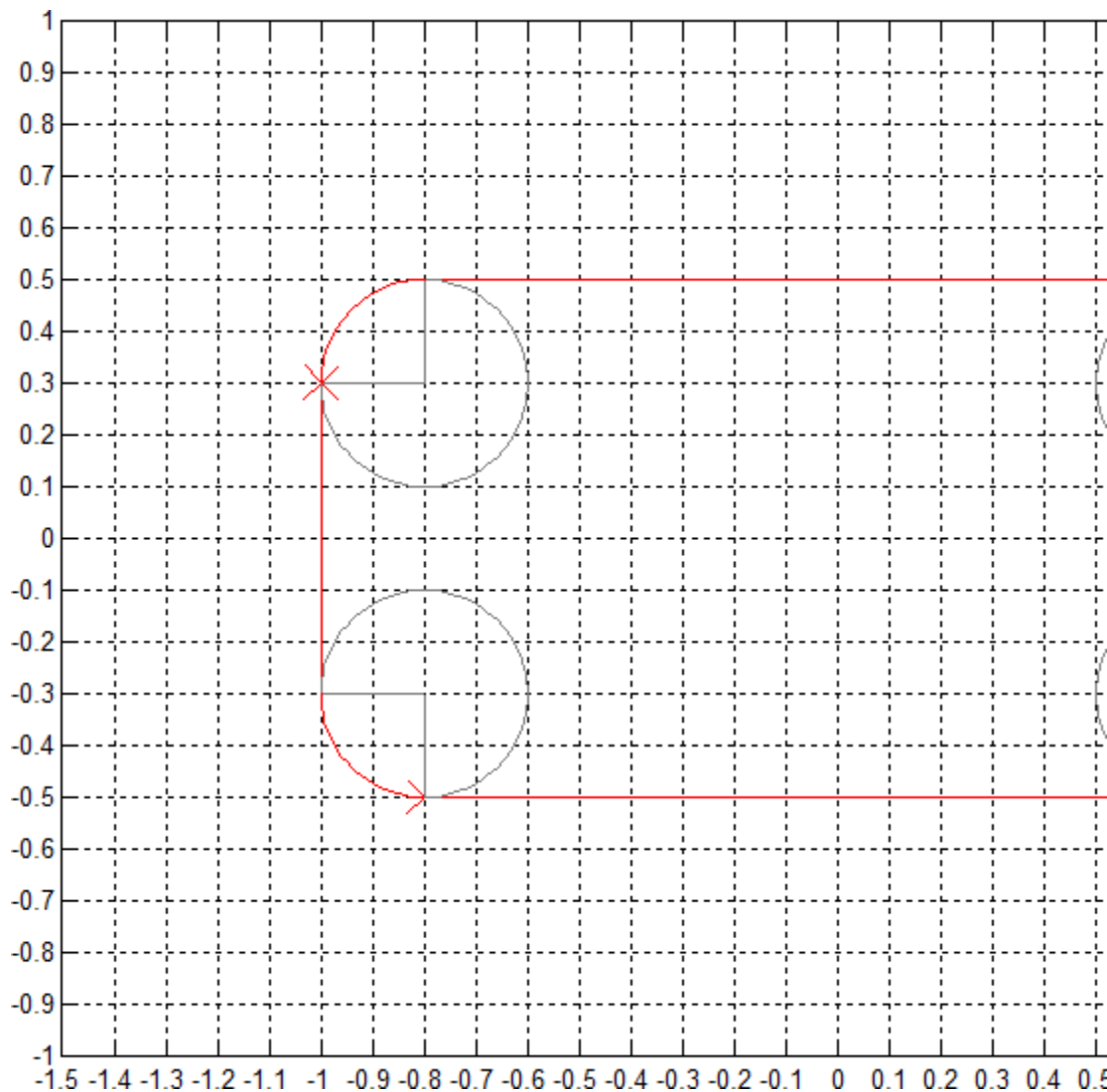
Because of the intersection of the solid objects used in the initial CSG model, a number of subdomain borders remain. They are drawn using gray lines. If this is a model of, e.g., a homogeneous plate, you can remove them. Select the **Remove All Subdomain Borders** option from the **Boundary** menu.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

Enter 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*xy* should be entered as `4*x.*y` and not as `4*x*y`, which normally is not a valid MATLAB expression.

Systems of PDEs

As described in "Types of PDE Problems You Can Solve" on page 1-4, Partial Differential Equation Toolbox can solve systems of PDEs. This means you can have N coupled PDEs, with coupled boundary conditions. The solvers such as `assempde` and `hyperbolic` can solve systems of PDEs with any number N of components.

Scalar PDEs are those with $N = 1$, meaning just one PDE. Systems of PDEs generally means $N > 1$. The documentation sometimes refers to systems as multidimensional PDEs or as PDEs with vector solution u.

In all cases, PDE systems have a single 2-D geometry and mesh. It is only N, the number of equations, that can vary.

Scalar PDE Coefficients

A scalar PDE is one of the following:

- Elliptic

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

- Parabolic

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

- Hyperbolic

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

- Eigenvalue

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

In all cases, the coefficients *d*, *c*, *a*, and *f* can be functions of position (*x* and *y*) and the subdomain index. For all cases except eigenvalue, the coefficients can also depend on the solution *u* and its gradient. And for parabolic and hyperbolic equations, the coefficients can also depend on time.

The question is how to represent the coefficients for the toolbox.

There are three ways of representing each coefficient. You can use different ways for different coefficients.

- Numeric — If a coefficient is numeric, give the value.
- String formula — See "Scalar PDE Coefficients in String Form" on page 2-16.
- MATLAB function — See "Scalar PDE Coefficients in Function Form" on page 2-19.

For an example incorporating each way to represent coefficients, see "Scalar PDE Functional Form and Calling Syntax" on page 2-22.

Note If any coefficient depends on time or on the solution u or its gradient, then that coefficient should be NaN when either time or the solution u is NaN. This is the way that solvers check to see if the equation depends on time or on the solution.

Scalar PDE Coefficients in String Form

Write a text expression using these conventions:

- `'x'` — *x*-coordinate

- `'y'` — *y*-coordinate

- `'u'` — Solution of equation

- `'ux'` — Derivative of *u* in the *x*-direction

- `'uy'` — Derivative of *u* in the *y*-direction

- `'t'` — Time (parabolic and hyperbolic equations)

- `'sd'` — Subdomain number

For example, you could use this string to represent a coefficient:

```
'(x+y)./(x.^2 + y.^2 + 1) + 3 + sin(t)./(1+u.^4)'
```

Note Use `.*`, `./`, and `.^` for multiplication, division, and exponentiation operations. The text expressions operate on row vectors, so the operations must make sense for row vectors. The row vectors are the values at the triangle centroids in the mesh.

You can write MATLAB functions for coefficients as well as plain text expressions. For example, suppose your coefficient f is given by the file fcoeff.m:

```
function f = fcoeff(x,y,t,sd)

f = (x.*y)./(1+x.^2+y.^2); % f on subdomain 1
f = f + log(1+t); % include time
r = (sd == 2); % subdomain 2
f(r) = cos(x+y); % f on subdomain 2
```

Represent this function in the `parabolic` solver, for example:

```
u1 = parabolic(u0,tlist,b,p,e,t,c,a,'fcoeff(x,y,t,sd)',d)
```

> **Caution** In function form, t represents triangles, and time represents time. In string form, t represents time, and triangles do not enter into the form.

There is a simple way to write a text expression for multiple subdomains without using 'sd' or a function. Separate the formulas for the different subdomains with the '!' character. Generally use the same number of expressions as subdomains. However, if an expression does not depend on the subdomain number, you can give just one expression.

For example, an expression for an input (a, c, f, or d) with three subdomains:

```
'(1+x.^2)./(1+x.^2+y.^2)!(1+(x+y).^2)./(1+x.^2+y.^2)!(1+x.^2)./(1+x.^2+y.
```

The coefficient c is a 2-by-2 matrix. You can give c in any of the following forms:

- Scalar or single string — The software interprets c as a diagonal matrix:

$$\begin{pmatrix} c & 0 \\ 0 & c \end{pmatrix}$$

- Two-row vector or two-row text array — The software interprets c as a diagonal matrix:

$$\begin{pmatrix} c(1) & 0 \\ 0 & c(2) \end{pmatrix}$$

- Three-row vector or three-row text array — The software interprets c as a symmetric matrix:

$$\begin{pmatrix} c(1) & c(2) \\ c(2) & c(3) \end{pmatrix}$$

- Four-row vector or four-row text array — The software interprets c as a full matrix:

$$\begin{pmatrix} c(1) & c(3) \\ c(2) & c(4) \end{pmatrix}$$

For example, *c* as a symmetric matrix with cos(*xy*) on the off-diagonal terms:

```
c = char('x.^2+y.^2',...
    'cos(x.*y)',...
    'u./(1+x.^2+y.^2)')
```

To include subdomains separated by '!', include the '!' in each row. For example,

```
c = char('1+x.^2+y.^2!x.^2+y.^2',...
    'cos(x.*y)!sin(x.*y)',...
    'u./(1+x.^2+y.^2)!u.*(x.^2+y.^2)')
```

Caution   Do not include spaces in your coefficient strings in pdetool. The string parser can misinterpret a space as a vector separator, as when a MATLAB vector uses a space to separate elements of a vector.

For elliptic problems, when you include 'u', 'ux', or 'uy', you must use the pdenonlin solver instead of assempde. In pdetool, select **Solve > Parameters > Use nonlinear solver**.

Related Examples	• "Scalar PDE Functional Form and Calling Syntax" on page 2-22 • "Enter Coefficients in pdetool" on page 2-28 • "Scalar PDE Coefficients in Function Form" on page 2-19
Concepts	• "Scalar PDE Coefficients" on page 2-14

Scalar PDE Coefficients in Function Form

In this section...

Coefficients as the Result of a Program

Usually. the simplest way to give coefficients as the result of a program is to use a string expression as described in "Scalar PDE Coefficients in String Form" on page 2-16. For the most detailed control over coefficients, though, you can write a function form of coefficients.

A coefficient in function form has the syntax

```
coeff = coeffunction(p,t,u,time)
```

`coeff` represents any coefficient: `c`, `a`, `f`, or `d`.

Your program evaluates the return `coeff` as a row vector of the function values at the centroids of the triangles `t`. For help calculating these values, see "Calculate Coefficients in Function Form" on page 2-20.

- `p` and `t` are the node points and triangles of the mesh. For a description of these data structures, see "Mesh Data" on page 2-76. In brief, each column of `p` contains the *x*- and *y*-values of a point, and each column of `t` contains the indices of three points in `p` and the subdomain label of that triangle.

- `u` is a row vector containing the solution at the points `p`. `u` is `[]` if the coefficients do not depend on the solution or its derivatives.

- `time` is the time of the solution, a scalar. `time` is `[]` if the coefficients do not depend on time.

Caution In function form, `t` represents triangles, and `time` represents time. In string form, `t` represents time, and triangles do not enter into the form.

Pass the coefficient function to the solver as a string `'coeffunction'` or as a function handle `@coeffunction`. In `pdetool`, pass the coefficient as a string `coeffunction` without quotes, because `pdetool` interprets all entries as strings.

If your coefficients depend on `u` or `time`, then when `u` or `time` are `NaN`, ensure that the corresponding `coeff` consist of a vector of `NaN` of the correct size. This signals to solvers, such as `parabolic`, to use a time-dependent or solution-dependent algorithm.

For elliptic problems, if any coefficient depends on `u` or its gradient, you must use the `pdenonlin` solver instead of `assempde`. In `pdetool`, select **Solve > Parameters > Use nonlinear solver**.

Calculate Coefficients in Function Form

X- and Y-Values
The *x*- and *y*-values of the centroid of a triangle `t` are the mean values of the entries of the points `p` in `t`. To get row vectors `xpts` and `ypts` containing the mean values:

```
% Triangle point indices
it1=t(1,:);
it2=t(2,:);
it3=t(3,:);

% Find centroids of triangles
xpts=(p(1,it1)+p(1,it2)+p(1,it3))/3;
ypts=(p(2,it1)+p(2,it2)+p(2,it3))/3;
```

Interpolated u
The `pdeintrp` function linearly interpolates the values of `u` at the centroids of `t`, based on the values at the points `p`.

```
uintrp = pdeintrp(p,t,u); % Interpolated values at centroids
```

The output `uintrp` is a row vector with the same number of columns as `t`. Use `uintrp` as the solution value in your coefficient calculations.

Gradient or Derivatives of u

The pdegrad function approximates the gradient of u.

```
[ux,uy] = pdegrad(p,t,u); % Approximate derivatives
```

The outputs ux and uy are row vectors with the same number of columns as t.

Subdomains

If your coefficients depend on the subdomain label, check the subdomain number for each triangle. Subdomains are the last (fourth) row of the triangle matrix. So the row vector of subdomain numbers is:

```
subd = t(4,:);
```

You can see the subdomain labels by using the pdegplot function with the subdomainLabels name-value pair set to 'on':

```
pdegplot(g,'subdomainLabels','on')
```

Related Examples

- "Scalar PDE Functional Form and Calling Syntax" on page 2-22
- "Enter Coefficients in pdetool" on page 2-28
- "Scalar PDE Coefficients in String Form" on page 2-16
- "Deflection of a Piezoelectric Actuator" on page 3-19

Concepts

- "Scalar PDE Coefficients" on page 2-14

Scalar PDE Functional Form and Calling Syntax

This example shows how to write PDE coefficients in string form and in functional form.

The geometry is a rectangle with a circular hole.
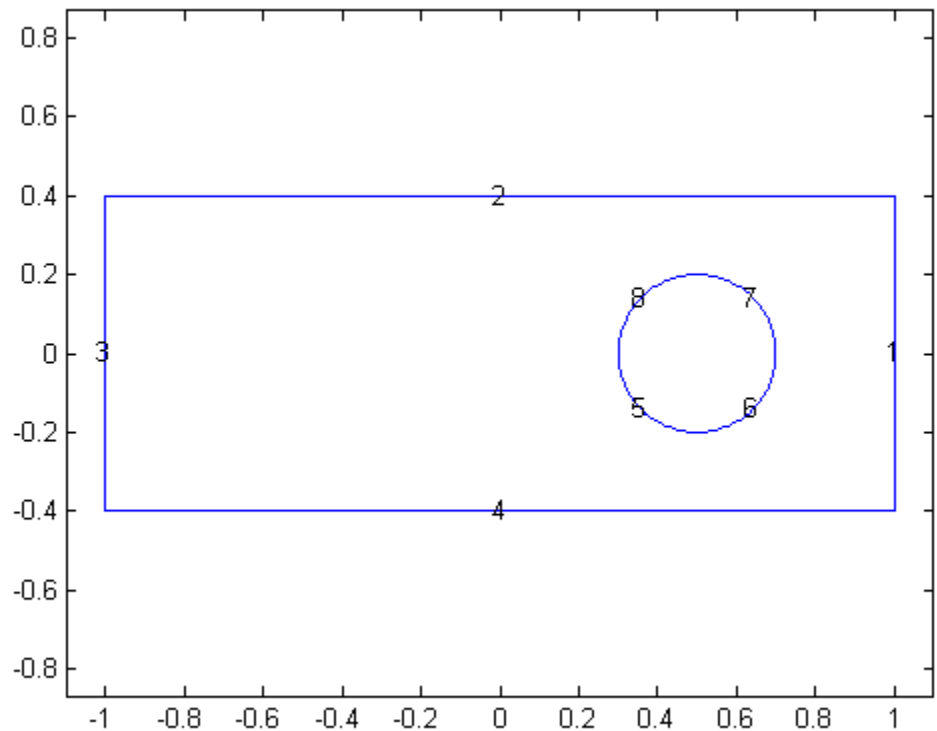
Code for generating the figure

```
% Rectangle is code 3, 4 sides,
% followed by x-coordinates and then y-coordinates
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
```

```
% Circle is code 1, center (.5,0), radius .2
C1 = [1,.5,0,.2]';
% Pad C1 with zeros to enable concatenation with R1
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];

% Names for the two geometric objects
ns = (char('R1','C1'))';

% Set formula
sf = 'R1-C1';

% Create geometry
gd = decsg(geom,sf,ns);

% View geometry
pdegplot(gd,'edgeLabels','on')
xlim([-1.1 1.1])
axis equal
```

The PDE is parabolic,

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

with the following coefficients:

- $d = 5$

- $a = 0$

- f is a linear ramp up to 10, holds at 10, then ramps back down to 0:

$$f = 10 * \begin{cases} 10t & 0 \le t \le 0.1 \\ 1 & 0.1 \le t \le 0.9 \\ 10 - 10t & 0.9 \le t \le 1 \end{cases}$$

- $c = 1 + .x^2 + y^2$

Write a function for the f coefficient.

```
function f = framp(t)

if t <= 0.1
    f = 10*t;
elseif t <= 0.9
    f = 1;
else
    f = 10-10*t;
end
f = 10*f;
```

The boundary conditions are the same as in "Boundary Conditions for Scalar PDE" on page 2-63.

Boundary conditions

Suppose the boundary conditions on the outer boundary (segments 1 through 4) are Dirichlet, with the value $u(x,y) = t(x - y)$, where t is time. Suppose the circular boundary (segments 5 through 8) has a generalized Neumann condition, with $q = 1$ and $g = x^2 + y^2$.

```
function [qmatrix,gmatrix,hmatrix,rmatrix] = pdebound(p,e,u,time)

ne = size(e,2); % number of edges
qmatrix = zeros(1,ne);
gmatrix = qmatrix;
hmatrix = zeros(1,2*ne);
rmatrix = hmatrix;

for k = 1:ne
    x1 = p(1,e(1,k)); % x at first point in segment
    x2 = p(1,e(2,k)); % x at second point in segment
    xm = (x1 + x2)/2; % x at segment midpoint
    y1 = p(2,e(1,k)); % y at first point in segment
    y2 = p(2,e(2,k)); % y at second point in segment
    ym = (y1 + y2)/2; % y at segment midpoint
    switch e(5,k)
        case {1,2,3,4} % rectangle boundaries
            hmatrix(k) = 1;
            hmatrix(k+ne) = 1;
```

```
                    if ~isempty(time)
                        rmatrix(k) = time*(x1 - y1);
                        rmatrix(k+ne) = time*(x2 - y2);
                    end
            otherwise % same as case {5,6,7,8}, circle boundaries
                    qmatrix(k) = 1;
                    gmatrix(k) = xm^2 + ym^2;
        end
end
```

The initial condition is $u(x,y) = 0$ at $t = 0$.

After running the code for creating the geometry, create the mesh, refine it twice, and jiggle it once.

```
[p,e,t] = initmesh(gd);
[p,e,t] = refinemesh(gd,p,e,t);
[p,e,t] = refinemesh(gd,p,e,t);
p = jigglemesh(p,e,t);
```

Set the time steps for the parabolic solver to 50 steps from time 0 to time 1.

```
tlist = linspace(0,1,50);
```

Solve the parabolic PDE.

```
b = @pdebound;
d = 5;
a = 0;
f = 'framp(t)';
c = '1+x.^2+y.^2';
u = parabolic(0,tlist,b,p,e,t,c,a,f,d);
```

View an animation of the solution.

```
for tt = 1:size(u,2) % number of steps
    pdeplot(p,e,t,'xydata',u(:,tt),'zdata',u(:,tt),'colormap','jet')
    axis([-1 1 -1/2 1/2 -1.5 1.5 -1.5 1.5]) % use fixed axis
    title(['Step ' num2str(tt)])
    view(-45,22)
    drawnow
    pause(.1)
```

```
end
```


Step 50
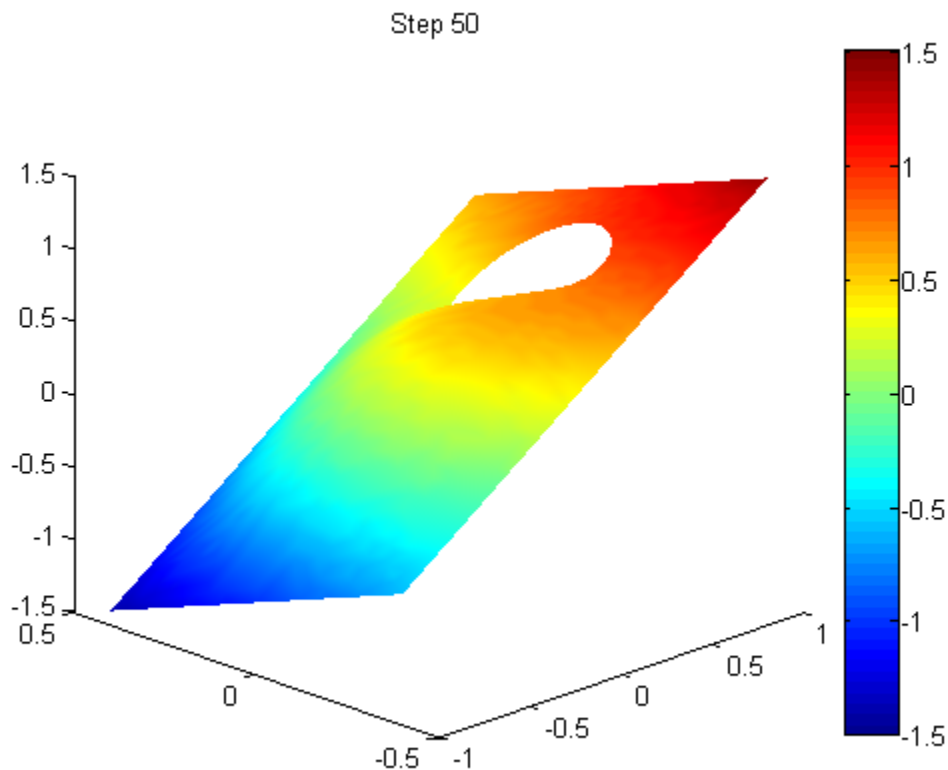
Equivalently, you can write a function for the coefficient *f* in the syntax described in "Scalar PDE Coefficients in Function Form" on page 2-19.

```
function f = framp2(p,t,u,time)

if time <= 0.1
    f = 10*time;
elseif time <= 0.9
    f = 1;
else
    f = 10-10*time;
```

```
end
f = 10*f;
```

Call this function by setting

```
f = @framp2;
u = parabolic(0,tlist,b,p,e,t,c,a,f,d);
```

You can also write a function for the coefficient *c*, though it is more complicated than the string formulation.

```
function c = cfunc(p,t,u,time)

% Triangle point indices
it1=t(1,:);
it2=t(2,:);
it3=t(3,:);

% Find centroids of triangles
xpts=(p(1,it1)+p(1,it2)+p(1,it3))/3;
ypts=(p(2,it1)+p(2,it2)+p(2,it3))/3;

c = 1 + xpts.^2 + ypts.^2;
```

Call this function by setting

```
c = @cfunc;
u = parabolic(0,tlist,b,p,e,t,c,a,f,d);
```

Related Examples
- "Enter Coefficients in pdetool" on page 2-28
- "Scalar PDE Coefficients in String Form" on page 2-16
- "Scalar PDE Coefficients in Function Form" on page 2-19
- "Nonlinear Heat Transfer In a Thin Plate" on page 3-60
- "Deflection of a Piezoelectric Actuator" on page 3-19

Concepts
- "Scalar PDE Coefficients" on page 2-14

Enter Coefficients in pdetool

This example shows how to enter coefficients in `pdetool`.

Caution   Do not include spaces in your coefficient strings in `pdetool`. The string parser can misinterpret a space as a vector separator, as when a MATLAB vector uses a space to separate elements of a vector.

The PDE is parabolic,

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

with the following coefficients:

- $d = 5$

- $a = 0$

- f is a linear ramp up to 10, holds at 10, then ramps back down to 0:

$$f = 10 * \begin{cases} 10t & 0 \le t \le 0.1 \\ 1 & 0.1 \le t \le 0.9 \\ 10 - 10t & 0.9 \le t \le 1 \end{cases}$$

- $c = 1 + .x^2 + y^2$

These coefficients are the same as in "Scalar PDE Functional Form and Calling Syntax" on page 2-22.

Write the following file `framp.m` and save it on your MATLAB path.

```
function f = framp(t)

if t <= 0.1
    f = 10*t;
elseif t <= 0.9
    f = 1;
```

```
else
    f = 10-10*t;
end
f = 10*f;
```

Open `pdetool`, either by typing `pdetool` at the command line, or selecting **Partial Differential Equation** from the **Apps** menu.

Select **PDE > PDE Specification**.

Select **Parabolic** equation. Fill in the coefficients as pictured:

- **c** = `1+x.^2+y.^2`

- **a** = `0`

- **f** = `framp(t)`

- **d** = `5`
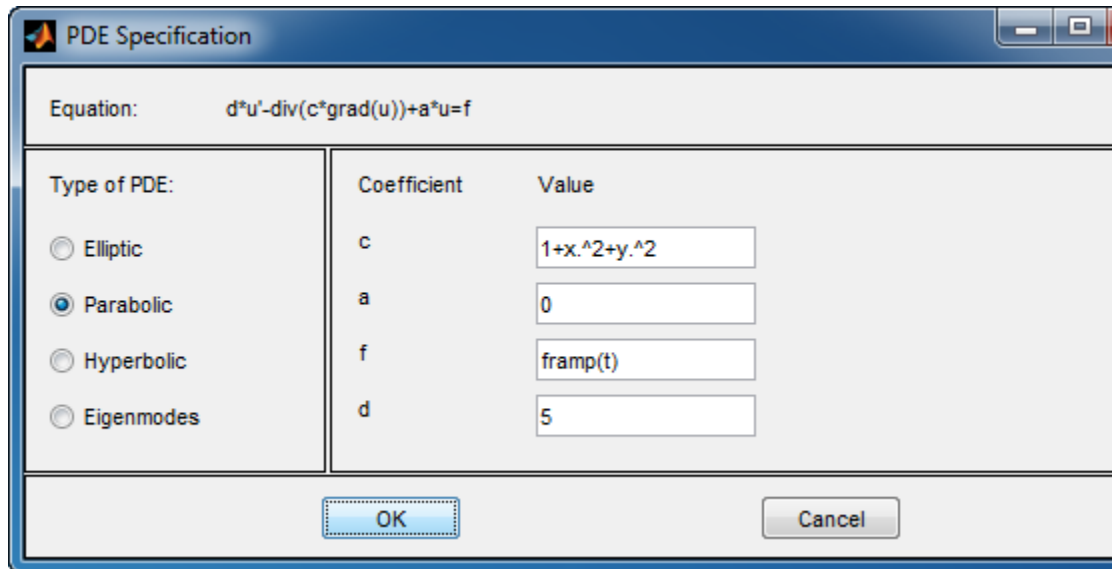
`pdetool` interprets all inputs as strings. Therefore, do not include quotes for the `c` or `f` coefficients.

Select **Options > Grid** and **Options > Snap**.

Select **Draw > Draw Mode**, then draw a rectangle centered at (0,0) extending to 1 in the *x*-direction and 0.4 in the *y*-direction.

Draw a circle centered at (0.5,0) with radius 0.2

Change the set formula to `R1-C1`.
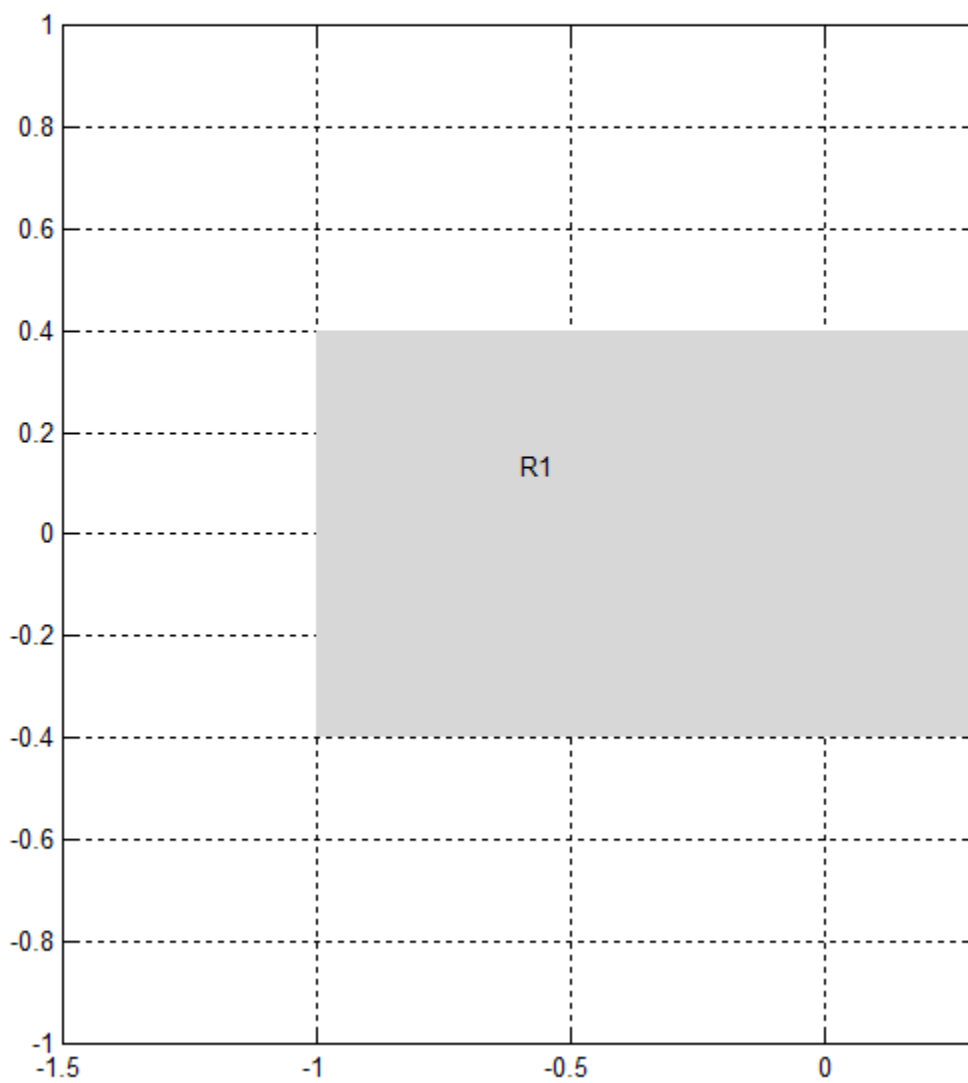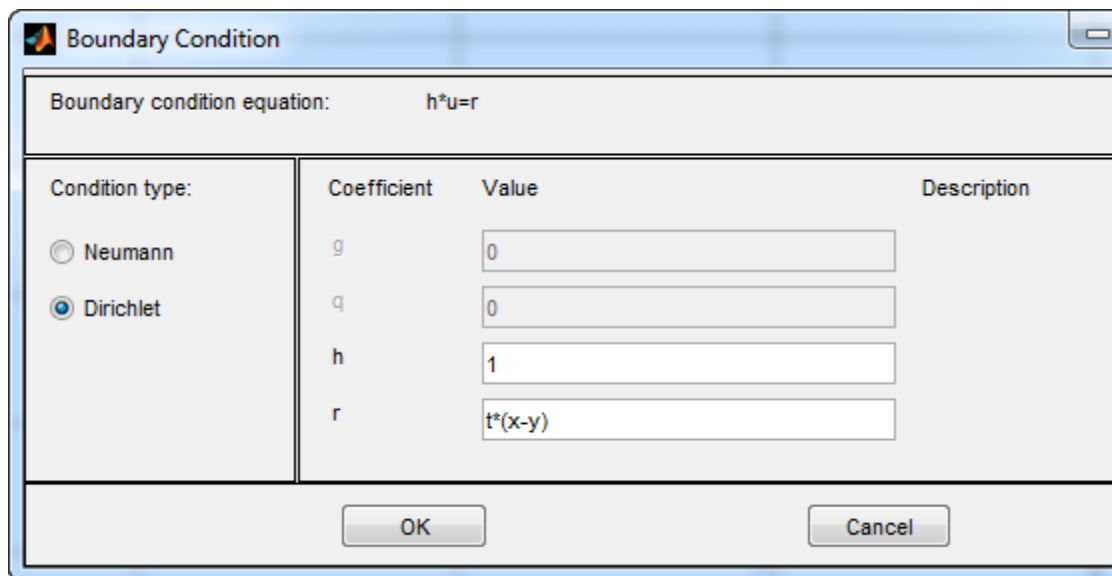
Select **Boundary > Boundary Mode**

Click a segment of the outer rectangle, then **Shift**-click the other three segments so that all four segments of the rectangle are selected.

Double-click one of the selected segments.

Fill in the resulting dialog box as pictured, with Dirichlet boundary conditions **h** = 1 and **r** = t*(x-y). Click **OK**.

Select the four segments of the inner circle using **Shift**-click, and double-click one of the segments.

Select **Neumann** boundary conditions, and set **g** = x.^2+y.^2 and **q** = 1. Click **OK**.
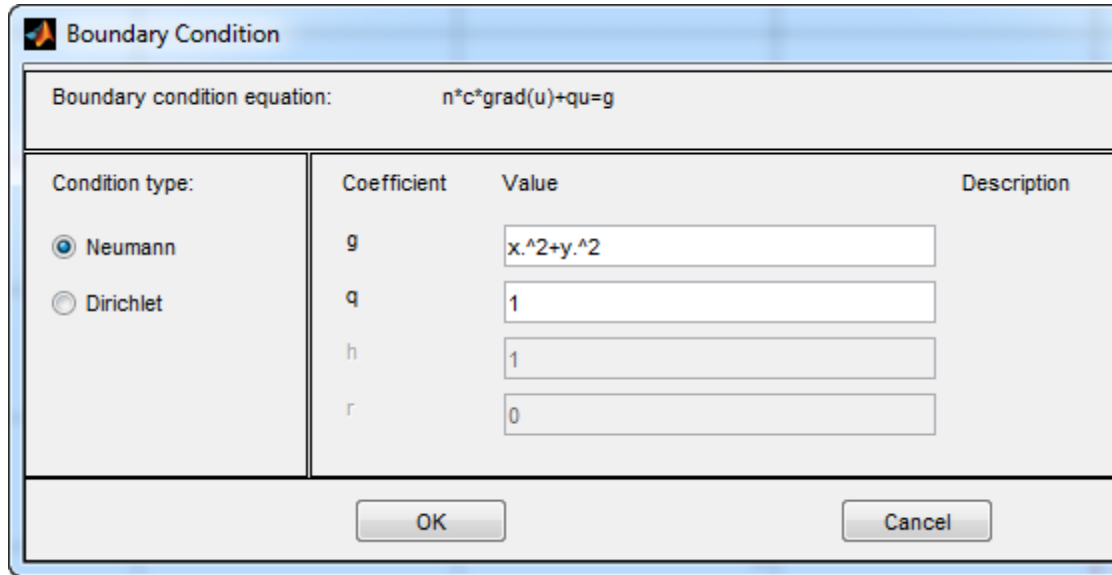
Click to initialize the mesh.

Click to refine the mesh. Click again to get an even finer mesh.

Select **Mesh > Jiggle Mesh** to improve the quality of the mesh.

Set the time interval and initial condition by selecting **Solve > Parameters** and setting **Time** = linspace(1,2,50) and **u(t0)** = 0. Click **OK**.
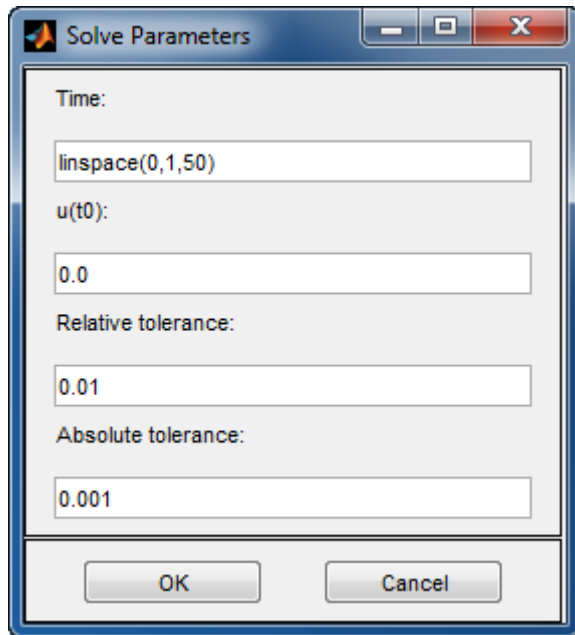
Solve and plot the equation by clicking the ⚌ button.

Time=1 Color: u
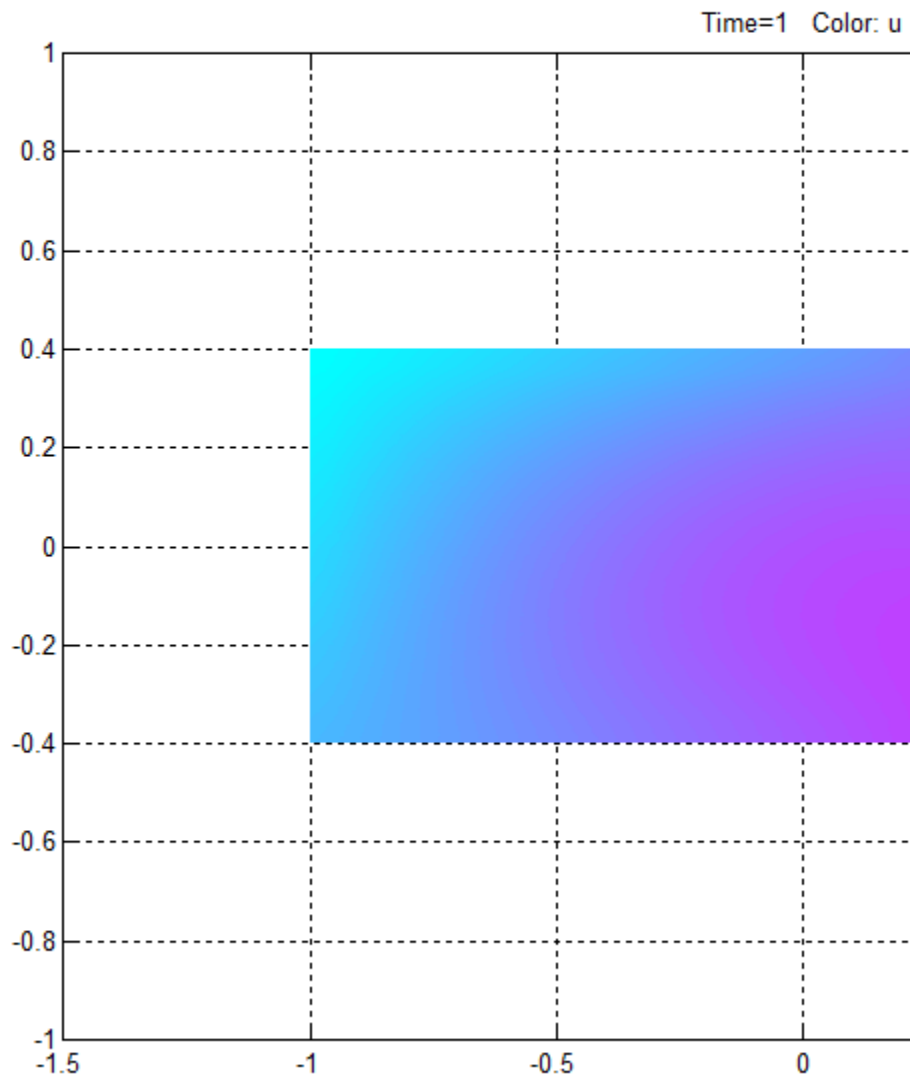
Match the following figure using **Plot > Parameters**.
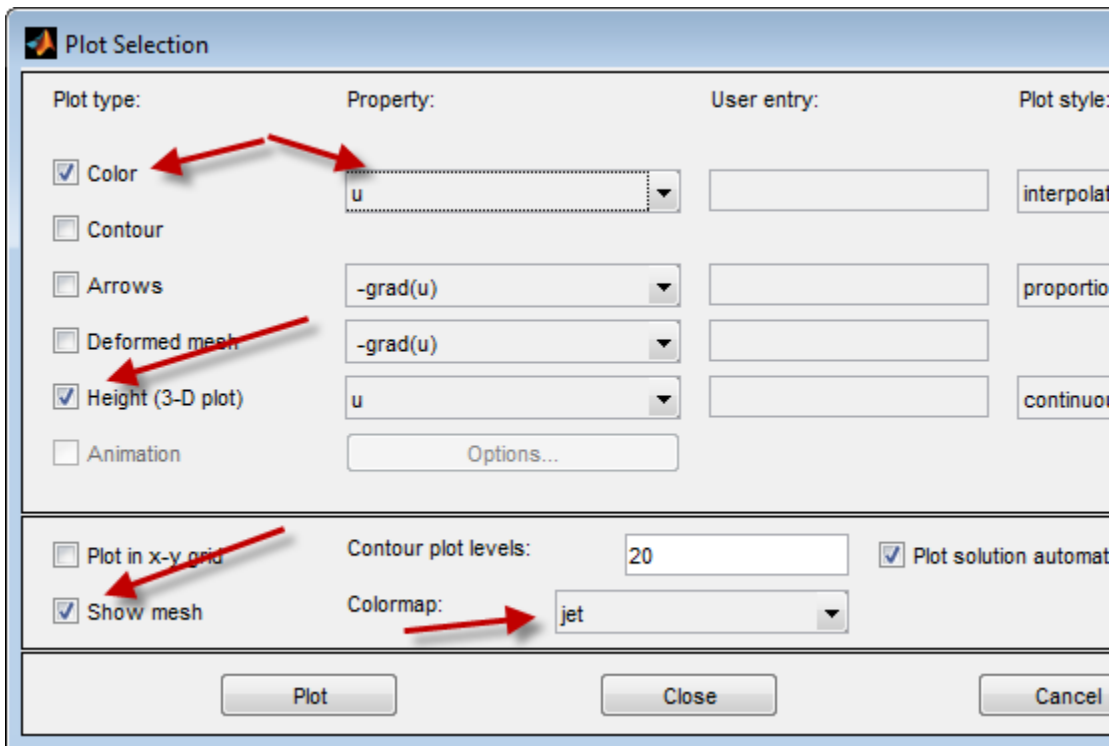
Click the **Plot** button.

Time=1   Color: u   Height: u
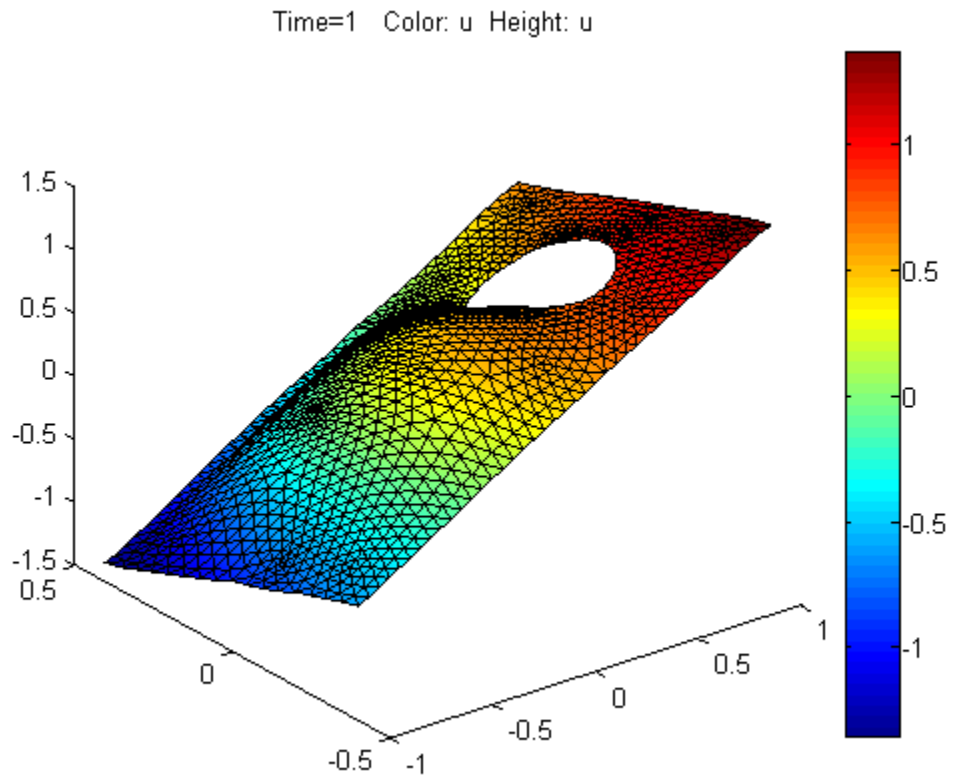
Related Examples
- "Scalar PDE Functional Form and Calling Syntax" on page 2-22
- "Scalar PDE Coefficients in String Form" on page 2-16
- "Scalar PDE Coefficients in Function Form" on page 2-19

Concepts
- "Scalar PDE Coefficients" on page 2-14

Coefficients for Systems of PDEs

As describe in "Systems of PDEs" on page 2-13, toolbox functions can address the case of systems of N PDEs. How do you represent the coefficients of your PDE in the correct form? For example, an elliptic system with two components in the solution vector **u** is

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

In general, an elliptic system is

$$-\nabla\cdot(\mathbf{c}\otimes\nabla\mathbf{u}) + \mathbf{au} = \mathbf{f},$$

The notation $\nabla\cdot(\mathbf{c}\otimes\nabla\mathbf{u})$ means the N-by-1 matrix with $(i,1)$-component

$$\sum_{j=1}^{N}\left(\frac{\partial}{\partial x}c_{i,j,1,1}\frac{\partial}{\partial x} + \frac{\partial}{\partial x}c_{i,j,1,2}\frac{\partial}{\partial y} + \frac{\partial}{\partial y}c_{i,j,2,1}\frac{\partial}{\partial x} + \frac{\partial}{\partial y}c_{i,j,2,2}\frac{\partial}{\partial y}\right)u_j$$

Other problems with $N > 1$ are the parabolic system

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

the hyperbolic system

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

and the eigenvalue system

$$-\nabla\cdot(\mathbf{c}\otimes\nabla\mathbf{u}) + \mathbf{au} = \lambda\mathbf{du}.$$

To solve a PDE using this toolbox, you convert your problem into one of the forms the toolbox accepts. Then express your problem coefficients in a form the toolbox accepts.

The question is how to express each coefficient: **d**, **c**, **a**, and **f**. For answers, see "f for Systems" on page 2-40, "c for Systems" on page 2-42, and "a or d for Systems" on page 2-51.

Note If any coefficient depends on time or on the solution u or its gradient, then all coefficients should be NaN when either time or the solution u is NaN. This is the way that solvers check to see if the equation depends on time or on the solution.

---

f for Systems

This section describes how to write the coefficient f in the equation

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \mathbf{f},$$

or in similar equations. The number of rows in f indicates *N*, the number of equations. Give f as any of the following:

- A scalar or single string expression. Solvers expand the single input to a vector of *N* elements.

- A column vector with *N* components. For example, if *N* = 3, f could be:

  ```
  f = [3;4;10];
  ```

- A character array with *N* rows. The rows of the character array are MATLAB expressions as described in "Scalar PDE Coefficients in String Form" on page 2-16. Pad the rows with spaces so each row has the same number of characters (char does this automatically). For example, if *N* = 3, f could be:

  ```
  f = char('sin(x) + cos(y)','cosh(x.*y)','x.*y./(1+x.^2+y.^2)')

  f =

  sin(x) + cos(y)
  cosh(x.*y)
  x.*y./(1+x.^2+y.^2)
  ```

- A function of the form as described in "Scalar PDE Coefficients in Function Form" on page 2-19. The function should return a matrix of size *N*-by-*Nt*, where *Nt* is the number of triangles in the mesh. The function should evaluate f at the triangle centroids, as in "Scalar PDE Coefficients in Function Form" on page 2-19. Give solvers the function name as a string '*filename*', or as a function handle @*filename*, where filename.m is a file on your MATLAB path. For details on writing the function, see "Calculate Coefficients in Function Form" on page 2-20.

  For example, if *N* = 3, f could be:

```
function f = fcoeffunction(p,t,u,time)

N = 3; % Number of equations
% Triangle point indices
it1=t(1,:);
it2=t(2,:);
it3=t(3,:);

% Find centroids of triangles
xpts=(p(1,it1)+p(1,it2)+p(1,it3))/3;
ypts=(p(2,it1)+p(2,it2)+p(2,it3))/3;

[ux,uy] = pdegrad(p,t,u); % Approximate derivatives
uintrp = pdeintrp(p,t,u); % Interpolated values at centroids
uintrp = reshape(uintrp,[],N); % matrix with N column
uintrp = uintrp'; % change to row vectors

nt = size(t,2); % Number of columns
f = zeros(N,nt); % Allocate f

% Now the particular functional form of f
f(1,:) = xpts - ypts + uintrp(1,:);
f(2,:) = 1 + tanh(ux(1,:)) + tanh(uy(3,:));
f(3,:) = (5+uintrp(3,:)).*sqrt(xpts.^2+ypts.^2);
```

Because this function depends on the solution **u**, if the equation is elliptic, use the pdenonlin solver. The initial value can be all 0s in the case of Dirichlet boundary conditions:

```
np = size(p,2); % number of points
u0 = zeros(N*np,1); % initial guess
```

Related Examples

- "a or d for Systems" on page 2-51
- "c for Systems" on page 2-42
- "Deflection of a Piezoelectric Actuator" on page 3-19

c for Systems

c as Tensor, Matrix, and Vector

This section describes how to write the coefficient c in the equation

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \mathbf{f},$$

or in similar equations. The coefficient c is an *N*-by-*N*-by-2-by-2 tensor with components $\mathbf{c}(i,j,k,l)$.

The notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ means the *N*-by-1 matrix with $(i,1)$-component.

$$\sum_{j=1}^{N} \left(\frac{\partial}{\partial x} c_{i,j,1,1} \frac{\partial}{\partial x} + \frac{\partial}{\partial x} c_{i,j,1,2} \frac{\partial}{\partial y} + \frac{\partial}{\partial y} c_{i,j,2,1} \frac{\partial}{\partial x} + \frac{\partial}{\partial y} c_{i,j,2,2} \frac{\partial}{\partial y} \right) u_j$$

There are many ways to represent the coefficient *c*. All representations begin with a "flattening" of the *N*-by-*N*-by-2-by-2 tensor to a 2*N*-by-2*N* matrix, where the matrix is logically an *N*-by-*N* matrix of 2-by-2 blocks.

$$\begin{pmatrix} c(1,1,1,1) & c(1,1,1,2) & c(1,2,1,1) & c(1,2,1,2) & \cdots & c(1,N,1,1) & c(1,N,1,2) \\ c(1,1,2,1) & c(1,1,2,2) & c(1,2,2,1) & c(1,2,2,2) & \cdots & c(1,N,2,1) & c(1,N,2,2) \\ \\ c(2,1,1,1) & c(2,1,1,2) & c(2,2,1,1) & c(2,2,1,2) & \cdots & c(2,N,1,1) & c(2,N,1,2) \\ c(2,1,2,1) & c(2,1,2,2) & c(2,2,2,1) & c(2,2,2,2) & \cdots & c(2,N,2,1) & c(2,N,2,2) \\ \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c(N,1,1,1) & c(N,1,1,2) & c(N,2,1,1) & c(N,2,1,2) & \cdots & c(N,N,1,1) & c(N,N,1,2) \\ c(N,1,2,1) & c(N,1,2,2) & c(N,2,2,1) & c(N,2,2,2) & \cdots & c(N,N,2,1) & c(N,N,2,2) \end{pmatrix}$$

The matrix further gets flattened to a vector, where the *N*-by-*N* matrix of 2-by-2 blocks is first transformed to a "vector" of 2-by-2 blocks, and then the 2-by-2 blocks are turned into vectors in the usual column-wise way.

The coefficient vector c relates to the tensor **c** as follows:

$$\begin{pmatrix} c(1) & c(3) & c(2N+1) & c(2N+3) & \cdots & c(2N(2N-1)+1) & c(2N(2N-1) \\ c(2) & c(4) & c(2N+2) & c(2N+4) & \cdots & c(2N(2N-1)+2) & c(2N(2N-1) \\ \\ c(5) & c(7) & c(2N+5) & c(2N+7) & \cdots & c(2N(2N-1)+5) & c(2N(2N-1) \\ c(6) & c(8) & c(2N+6) & c(2N+8) & \cdots & c(2N(2N-1)+6) & c(2N(2N-1) \\ \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ c(2N-3) & c(2N-1) & c(4N-3) & c(4N-1) & \cdots & c(4N^2-3) & c(4N^2-1 \\ c(2N-2) & c(2N) & c(4N-2) & c(4N) & \cdots & c(4N^2-2) & c(4N^2) \end{pmatrix}$$

Coefficient $c(i,j,k,l)$ is in row $(4N(j-1)+4i+2l+k-6)$ of the vector c.

Express c as numbers, text expressions, or functions, as in "f for Systems" on page 2-40.

Often, your tensor **c** has structure, such as symmetric or block diagonal. In many cases, you can represent **c** using a smaller vector than one with $4N^2$ components.

The number of rows in the matrix can differ from $4N^2$, as described in the next few sections.

In function form, the number of columns is `Nt`, which is the number of triangles in the mesh. The function should evaluate c at the triangle centroids, as in "Scalar PDE Coefficients in Function Form" on page 2-19. Give solvers the function name as a string `'filename'`, or as a function handle `@filename`, where `filename.m` is a file on your MATLAB path. For details on writing the function, see "Calculate Coefficients in Function Form" on page 2-20.

Scalar c

The software interprets a scalar c as a diagonal matrix, with $c(i,i,1,1)$ and $c(i,i,2,2)$ equal to the scalar, and all other entries 0.

$$\begin{pmatrix} c & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & c & 0 & 0 & \cdots & 0 & 0 \\ & & & & & & \\ 0 & 0 & c & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & c & \cdots & 0 & 0 \\ & & & & & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & c \end{pmatrix}$$

Two-Row Vector c

The software interprets a two-row vector c as a diagonal matrix, with $c(i,i,1,1)$ and $c(i,i,2,2)$ as the two entries, and all other entries 0.

$$\begin{pmatrix} c(1) & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & c(2) & 0 & 0 & \cdots & 0 & 0 \\ & & & & & & \\ 0 & 0 & c(1) & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & c(2) & \cdots & 0 & 0 \\ & & & & & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c(1) & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & c(2) \end{pmatrix}$$

Three-Row Vector c

The software interprets a three-row vector c as a symmetric block diagonal matrix, with $c(i,i,1,1) = c(1)$, $c(i,i,2,2) = c(3)$, and $c(i,i,1,2) = c(i,i,2,1) = c(2)$.

$$\begin{pmatrix} c(1) & c(2) & 0 & 0 & \cdots & 0 & 0 \\ c(2) & c(3) & 0 & 0 & \cdots & 0 & 0 \\ & & & & & & \\ 0 & 0 & c(1) & c(2) & \cdots & 0 & 0 \\ 0 & 0 & c(2) & c(3) & \cdots & 0 & 0 \\ & & & & & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c(1) & c(2) \\ 0 & 0 & 0 & 0 & \cdots & c(2) & c(3) \end{pmatrix}$$

Four-Row Vector c

The software interprets a four-row vector c as a block diagonal matrix.

2-45

$$
\begin{pmatrix}
c(1) & c(3) & 0 & 0 & \cdots & 0 & 0 \\
c(2) & c(4) & 0 & 0 & \cdots & 0 & 0 \\
\\
0 & 0 & c(1) & c(3) & \cdots & 0 & 0 \\
0 & 0 & c(2) & c(4) & \cdots & 0 & 0 \\
\\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & c(1) & c(3) \\
0 & 0 & 0 & 0 & \cdots & c(2) & c(4)
\end{pmatrix}
$$

N-Row Vector c

The software interprets an N-row vector c as a diagonal matrix.

$$
\begin{pmatrix}
c(1) & 0 & 0 & 0 & \cdots & 0 & 0 \\
0 & c(1) & 0 & 0 & \cdots & 0 & 0 \\
\\
0 & 0 & c(2) & 0 & \cdots & 0 & 0 \\
0 & 0 & 0 & c(2) & \cdots & 0 & 0 \\
\\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & c(N) & 0 \\
0 & 0 & 0 & 0 & \cdots & 0 & c(N)
\end{pmatrix}
$$

Caution   If $N = 2$, 3, or 4, the 2-, 3-, or 4-row vector form takes precedence over the N-row form. So, for example, if $N = 3$, and you have a c matrix of the form

$$\begin{pmatrix} c1 & 0 & 0 & 0 & 0 & 0 \\ 0 & c1 & 0 & 0 & 0 & 0 \\ 0 & 0 & c2 & 0 & 0 & 0 \\ 0 & 0 & 0 & c2 & 0 & 0 \\ 0 & 0 & 0 & 0 & c3 & 0 \\ 0 & 0 & 0 & 0 & 0 & c3 \end{pmatrix},$$

you cannot use the N-row form of c, and instead would have to use the $2N$-row form. If you give c as the vector [c1;c2;c3], the software interprets it as a 3-row form, namely

$$\begin{pmatrix} c1 & c2 & 0 & 0 & 0 & 0 \\ c2 & c3 & 0 & 0 & 0 & 0 \\ 0 & 0 & c1 & c2 & 0 & 0 \\ 0 & 0 & c2 & c3 & 0 & 0 \\ 0 & 0 & 0 & 0 & c1 & c2 \\ 0 & 0 & 0 & 0 & c2 & c3 \end{pmatrix}.$$

Instead, use the $2N$-row form [c1;c1;c2;c2;c3;c3].

2N-Row Vector c

The software interprets a $2N$-row vector c as a diagonal matrix.

$$
\begin{pmatrix}
c(1) & 0 & 0 & 0 & \cdots & 0 & 0 \\
0 & c(2) & 0 & 0 & \cdots & 0 & 0 \\
0 & 0 & c(3) & 0 & \cdots & 0 & 0 \\
0 & 0 & 0 & c(4) & \cdots & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \cdots & c(2N-1) & 0 \\
0 & 0 & 0 & 0 & \cdots & 0 & c(2N)
\end{pmatrix}
$$

Caution If $N = 2$, the 4-row form takes precedence over the $2N$-row form. For example, if your c matrix is

$$
\begin{pmatrix}
c1 & 0 & 0 & 0 \\
0 & c2 & 0 & 0 \\
0 & 0 & c3 & 0 \\
0 & 0 & 0 & c4
\end{pmatrix},
$$

you cannot give c as [c1;c2;c3;c4], because the software interprets this vector as the 4-row form

$$
\begin{pmatrix}
c1 & c3 & 0 & 0 \\
c2 & c4 & 0 & 0 \\
0 & 0 & c1 & c3 \\
0 & 0 & c2 & c4
\end{pmatrix}.
$$

Instead, use the $3N$-row form [c1;0;c2;c3;0;c4] or the $4N$-row form [c1;0;0;c2;c3;0;0;c4].

3N-Row Vector c

The software interprets a $3N$-row vector c as a symmetric block diagonal matrix.

$$\begin{pmatrix} c(1) & c(2) & 0 & 0 & \cdots & 0 & 0 \\ c(2) & c(3) & 0 & 0 & \cdots & 0 & 0 \\ \\ 0 & 0 & c(4) & c(5) & \cdots & 0 & 0 \\ 0 & 0 & c(5) & c(6) & \cdots & 0 & 0 \\ \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c(3N-2) & c(3N-1) \\ 0 & 0 & 0 & 0 & \cdots & c(3N-1) & c(3N) \end{pmatrix}$$

Coefficient $c(i,j,k,l)$ is in row $(3i+k+l-4)$ of the vector c.

4N-Row Vector c

The software interprets a $4N$-row vector c as a block diagonal matrix.

$$\begin{pmatrix} c(1) & c(3) & 0 & 0 & \cdots & 0 & 0 \\ c(2) & c(4) & 0 & 0 & \cdots & 0 & 0 \\ \\ 0 & 0 & c(5) & c(7) & \cdots & 0 & 0 \\ 0 & 0 & c(6) & c(8) & \cdots & 0 & 0 \\ \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & c(4N-3) & c(4N-1) \\ 0 & 0 & 0 & 0 & \cdots & c(4N-2) & c(4N) \end{pmatrix}$$

Coefficient $c(i,j,k,l)$ is in row $(4i+2l+k-6)$ of the vector c.

2N(2N+1)/2-Row Vector c

The software interprets a $2N(2N+1)/2$-row vector c as a symmetric matrix. In the following diagram, $\cdot$ means the entry is symmetric.

$$
\begin{pmatrix}
c(1) & c(2) & c(4) & c(6) & \cdots & c((N-1)(2N-1)+1) & c((N-1)(2N-1)+3) \\
\square & c(3) & c(5) & c(7) & \cdots & c((N-1)(2N-1)+2) & c((N-1)(2N-1)+4) \\
 & & & & & & \\
\square & \square & c(8) & c(9) & \cdots & c((N-1)(2N-1)+5) & c((N-1)(2N-1)+7) \\
\square & \square & \square & c(10) & \cdots & c((N-1)(2N-1)+6) & c((N-1)(2N-1)+8) \\
 & & & & & & \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\square & \square & \square & \square & \cdots & c(N(2N+1)-2) & c(N(2N+1)-1) \\
\square & \square & \square & \square & \cdots & \square & c(N(2N+1))
\end{pmatrix}
$$

Coefficient $c(i,j,k,l)$, for $i < j$, is in row $(2j^2-3j+4i+2l+k-5)$ of the vector c. For $i = j$, coefficient $c(i,j,k,l)$ is in row $(2i^2+i+l+k-4)$ of the vector c.

$4N^2$-Row Vector c

The software interprets a $4N^2$-row vector c as a matrix.

$$
\begin{pmatrix}
c(1) & c(3) & c(2N+1) & c(2N+3) & \cdots & c(2N(2N-1)+1) & c(2N(2N-1)\ldots \\
c(2) & c(4) & c(2N+2) & c(2N+4) & \cdots & c(2N(2N-1)+2) & c(2N(2N-1)\ldots \\
 & & & & & & \\
c(5) & c(7) & c(2N+5) & c(2N+7) & \cdots & c(2N(2N-1)+5) & c(2N(2N-1)\ldots \\
c(6) & c(8) & c(2N+6) & c(2N+8) & \cdots & c(2N(2N-1)+6) & c(2N(2N-1)\ldots \\
 & & & & & & \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
c(2N-3) & c(2N-1) & c(4N-3) & c(4N-1) & \cdots & c(4N^2-3) & c(4N^2-1\ldots \\
c(2N-2) & c(2N) & c(4N-2) & c(4N) & \cdots & c(4N^2-2) & c(4N^2\ldots
\end{pmatrix}
$$

Coefficient $c(i,j,k,l)$ is in row $(4N(j-1)+4i+2l+k-6)$ of the vector c.

a or d for Systems

Coefficients a or d

This section describes how to write the coefficients a or d in the equation

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

or in similar equations. $\mathbf{a}$ and $\mathbf{d}$ are N-by-N matrices, where N is the number of equations.

Express the coefficients as numbers, text expressions, or functions, as in "f for Systems" on page 2-40.

The number of rows in the matrix is either 1, N, $N(N+1)/2$, or N^2, as described in the next few sections. If you choose to express the coefficients in functional form, the number of columns is Nt, which is the number of triangles in the mesh. The function should evaluate a or d at the triangle centroids, as in "Scalar PDE Coefficients in Function Form" on page 2-19. Give solvers the function name as a string '*filename*', or as a function handle @*filename*, where filename.m is a file on your MATLAB path. For details on how to write the function, see "Calculate Coefficients in Function Form" on page 2-20.

Often, $\mathbf{a}$ or $\mathbf{d}$ have structure, either as symmetric or diagonal. In these cases, you can represent $\mathbf{a}$ or $\mathbf{d}$ using fewer than N^2 rows.

Scalar a or d

The software interprets a scalar a or d as a diagonal matrix.

$$\begin{pmatrix} a & 0 & \cdots & 0 \\ 0 & a & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a \end{pmatrix}$$

N-Row Vector a or d

The software interprets an N-row vector a or d as a diagonal matrix.

$$\begin{pmatrix} d(1) & 0 & \cdots & 0 \\ 0 & d(2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d(N) \end{pmatrix}$$

For example, if N = 3, a or d could be

```
a = char('sin(x) + cos(y)','cosh(x.*y)','x.*y./(1+x.^2+y.^2)') % or d

a =

sin(x) + cos(y)
cosh(x.*y)
x.*y./(1+x.^2+y.^2)
```

N(N+1)/2-Row Vector a or d

The software interprets an $N(N+1)/2$-row vector a or d as a symmetric matrix. In the following diagram, · means the entry is symmetric.

$$\begin{pmatrix} a(1) & a(2) & a(4) & \cdots & a(N(N-1)/2) \\ \cdot & a(3) & a(5) & \cdots & a(N(N-1)/2+1) \\ \cdot & \cdot & a(6) & \cdots & a(N(N-1)/2+2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \cdot & \cdot & \cdot & \cdots & a(N(N+1)/2) \end{pmatrix}$$

Coefficient $a(i,j)$ is in row $(j(j–1)/2+i)$ of the vector a.

N²-Row Vector a or d

The software interprets an N^2-row vector a or d as a matrix.

$$\begin{pmatrix} d(1) & d(N+1) & \cdots & d(N^2-N+1) \\ d(2) & d(N+2) & \cdots & d(N^2-N+2) \\ \vdots & \vdots & \ddots & \vdots \\ d(N) & d(2N) & \cdots & d(N^2) \end{pmatrix}$$

Coefficient $a(i,j)$ is in row $(N(j–1)+i)$ of the vector a.

Related Examples
- "f for Systems" on page 2-40
- "c for Systems" on page 2-42
- "Deflection of a Piezoelectric Actuator" on page 3-19

Initial Conditions

Initial conditions means the solution u at the initial time. Pass initial conditions for hyperbolic or parabolic equations.

Include the initial conditions in the u0 argument. For hyperbolic equations, also pass the first derivative of the solution u at the initial time in the ut0 argument. The form of the ut0 input is exactly the same as the u0 input. The discussion addresses the u0 input.

Note The initial conditions and boundary conditions should be consistent.

You can give the initial conditions in these ways:

- A scalar

- A single text expression, using the conventions in "Scalar PDE Coefficients in String Form" on page 2-16

- A column vector

- A character matrix of text expressions, using the conventions in "Scalar PDE Coefficients in String Form" on page 2-16

The size of the column vector depends on the number of equations N and on the number of points in the mesh.

For scalar u, you can give a column vector with the same number of points as the mesh contains (the points are p in the usual p, e, t mesh description, see "Mesh Data" on page 2-76). The value of element k corresponds to point p(k).

For a system of N equations, you can give a column vector of $N*N_p$ elements. The first N_p elements contain the values of component 1, where the value of element k corresponds to point p(k). The next N_p points contain the values of component 2, etc. It can be convenient initially to represent the initial conditions u0 as an N_p-by-N matrix, where the first column contains entries for component 1, the second column contains entries for component 2, etc. The final representation of the initial conditions is u0(:).

For example, suppose you have an initial condition

$$u(x, y) = \frac{xy \cos(x)}{1 + x^2 + y^2}.$$

You can give the following expression for the initial condition:

```
'x.*y.*cos(x)./(1+x.^2+y.^2)'
```

For a multicomponent example in vector form, suppose you have a function `myfun(x,y)` that calculates the value of the initial condition `u0(x,y)` for any particular x and y. Suppose that `p`, is the usual mesh point data structure ("Mesh Data" on page 2-76), and the PDE system has $N = 5$. To compute the initial conditions for all mesh points `p`:

```
N = 5;
np = size(p,2); % number of mesh points
u0 = zeros(np,N); % allocate initial matrix
for k = 1:np
    x = p(1,k);
    y = p(2,k);
    u0(k,:) = myfun(x,y); % fill in row k
end
u0 = u0(:); % convert to column form
```

Give `u0` as the initial condition.

See Also   hyperbolic | parabolic | pdenonlin

Types of Boundary Conditions

The general mixed-boundary conditions for PDE systems (see "Systems of PDEs" on page 2-13) are

$$\mathbf{hu} = \mathbf{r}$$
$$\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{qu} = \mathbf{g} + \mathbf{h}'\mu.$$

The notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ means 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 $\mathbf{n} = (\cos(\alpha), \sin(\alpha))$. For each edge segment, there are M Dirichlet conditions and the h-matrix is M-by-N, $M \geq 0$. The generalized Neumann condition contains a source $\mathbf{h}'\mu$ where the solver computes Lagrange multipliers μ such that the Dirichlet conditions are satisfied.

"Boundary Conditions Overview" on page 2-62 describes a recommended approach for writing boundary conditions.

The problem of how to specify boundary conditions is somewhat simpler in the case of scalar solutions. Find the directions for your specific problem in "Boundary Conditions for Scalar PDE" on page 2-63 or "Boundary Conditions for PDE Systems" on page 2-68.

No Boundary Conditions Between Subdomains

There are two types of boundaries:

- Boundaries between the interior of the region and the exterior of the region
- Boundaries between subdomains—these are boundaries in the interior of the region

Boundary conditions, either Dirichlet or generalized Neumann, apply only to boundaries between the interior and exterior of the region. This is because the toolbox formulation uses the weak form of PDEs; see "Finite Element Method (FEM) Basics" on page 1-32. In the weak formulation you do not specify boundary conditions between subdomains, even if coefficients are discontinuous between subdomains. So both ways of specifying boundary conditions, boundary matrix and boundary file, do not support defining boundary conditions on subdomain boundaries.

For example, look at a rectangular region with a circular subdomain. The red numbers are the subdomain labels, the black numbers are the edge segment labels.
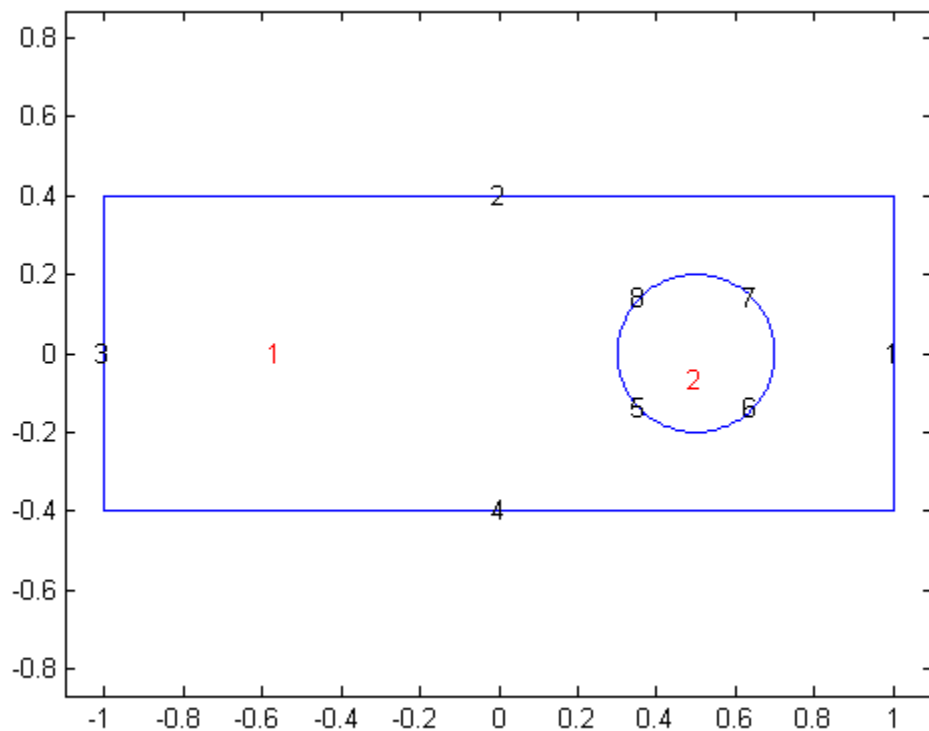
Code for generating the figure

```
% Rectangle is code 3, 4 sides,
% followed by x-coordinates and then y-coordinates
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
% Circle is code 1, center (.5,0), radius .2
C1 = [1,.5,0,.2]';
% Pad C1 with zeros to enable concatenation with R1
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];

% Names for the two geometric objects
ns = (char('R1','C1'))';
```

```
% Set formula
sf = 'R1+C1';

% Create geometry
gd = decsg(geom,sf,ns);

% View geometry
pdegplot(gd,'edgeLabels','on','subdomainLabels','on')
xlim([-1.1 1.1])
axis equal
```

You need not give boundary conditions on segments 5, 6, 7, and 8, because these are subdomain boundaries, not exterior boundaries.

However, if the circle is a hole, meaning it is not part of the region, then you do give boundary conditions on segments 5, 6, 7, and 8. For an example, see "Scalar PDE Functional Form and Calling Syntax" on page 2-22.

Identify Boundary Labels

You can see the edge labels by using the pdegplot function with the edgeLabels name-value pair set to 'on':

```
pdegplot(g,'edgeLabels','on')
```

For example, look at the edge labels for a simple annulus geometry:

```
e1 = [4;0;0;1;.5;0]; % Outside ellipse
e2 = [4;0;0;.5;.25;0]; % Inside ellipse
ee = [e1 e2]; % Both ellipses
lbls = char('outside','inside'); % Ellipse labels
lbls = lbls'; % Change to columns
sf = 'outside-inside'; % Set formula
dl = decsg(ee,sf,lbls); % Geometry now done
pdegplot(dl,'edgeLabels','on')
```
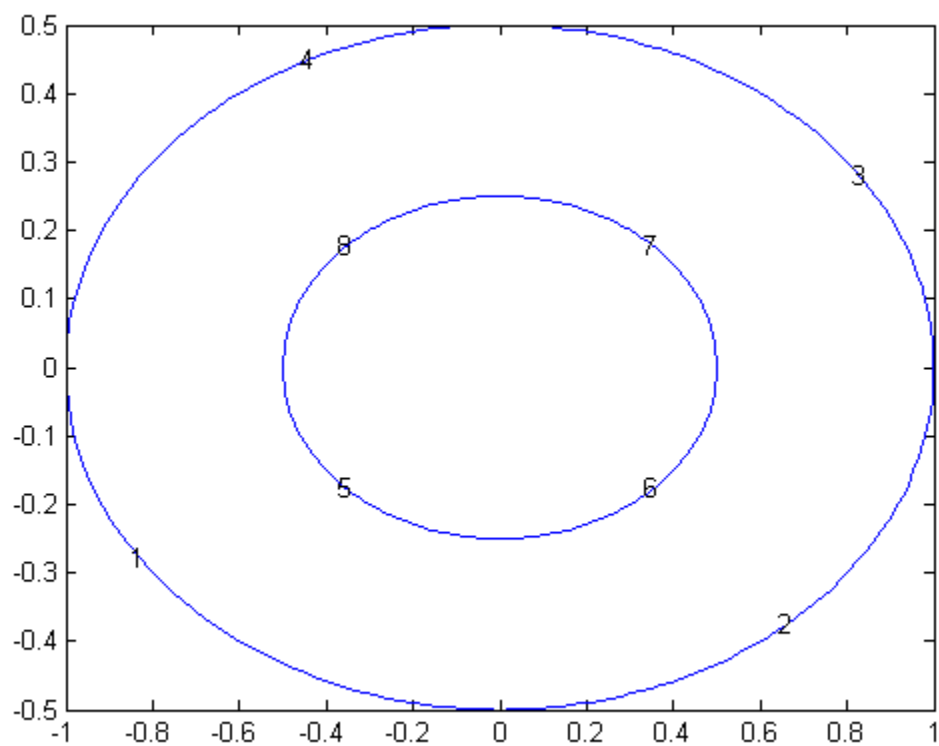
Boundary Conditions Overview

There are two ways to specify boundary conditions:

- Boundary matrix
- Boundary file

Except for the simplest cases, the easiest way to specify boundary conditions at the command line is with a boundary file (see pdebound). Write a function file, say pdebound.m, with the following syntax:

```
[qmatrix,gmatrix,hmatrix,rmatrix] = pdebound(p,e,u,time)
```

Your function returns matrices qmatrix, gmatrix, hmatrix, and rmatrix, based on these inputs:

- p — Points in the mesh ("Mesh Data" on page 2-76)
- e — Finite element edges in the mesh, a subset of all the edges ("Mesh Data" on page 2-76)
- u — Solution of the PDE
- time — Time, for parabolic or hyperbolic PDE only

If your boundary conditions do not depend on u or time, those inputs are []. If your boundary conditions do depend on u or time, then when u or time are NaN, ensure that the outputs such as qmatrix consist of matrices of NaN of the correct size. This signals to solvers, such as parabolic, to use a time-dependent or solution-dependent algorithm.

Before specifying boundary conditions, you need to know the boundary labels. See "Identify Boundary Labels" on page 2-60.

Alternatively, to generate a boundary matrix, use pdetool to draw your geometry and specify your boundary conditions, and then export the boundary conditions as a boundary matrix.

Boundary Conditions for Scalar PDE

For a scalar PDE, some boundary segments can have Dirichlet conditions, and some boundary segments can have generalized Neumann conditions.

Dirichlet boundary conditions are

$hu = r,$

where h and r can be functions of x, y, the solution u, the edge segment index, and, for parabolic and hyperbolic equations, time.

Generalized Neumann boundary conditions are $\vec{n} \cdot (c \nabla u) + qu = g$ on $\partial\Omega$.

$\vec{n}$ is the outward unit normal. g and q are functions defined on $\partial\Omega$, and can be functions of x, y, the solution u, the edge segment index, and, for parabolic and hyperbolic equations, time.

The PDE solver, such as `assempde` or `adaptmesh`, passes a matrix `p` of points and `e` of edges. `e` has seven rows and `ne` columns, where you do not necessarily know in advance the size `ne`.

- `p` is a 2-by-N_p matrix, where `p(1,k)` is the x-coordinate of point `k`, and `p(2,k)` is the y-coordinate of point `k`.
- `e` is a 7-by-`ne` matrix, where
  - `e(1,k)` is the index of the first point of edge `k`.
  - `e(2,k)` is the index of the second point of edge `k`.
  - `e(5,k)` is the label of the geometry edge of edge `k` (see "Identify Boundary Labels" on page 2-60).

  `e` contains an entry for every finite element edge that lies on an exterior boundary.

Use the following template for your boundary file.

```
function [qmatrix,gmatrix,hmatrix,rmatrix] = pdebound(p,e,u,time)

ne = size(e,2); % number of edges
```

```
qmatrix = zeros(1,ne);
gmatrix = qmatrix;
hmatrix = zeros(1,2*ne);
rmatrix = hmatrix;

for k = 1:ne
    x1 = p(1,e(1,k)); % x at first point in segment
    x2 = p(1,e(2,k)); % x at second point in segment
    xm = (x1 + x2)/2; % x at segment midpoint
    y1 = p(2,e(1,k)); % y at first point in segment
    y2 = p(2,e(2,k)); % y at second point in segment
    ym = (y1 + y2)/2; % y at segment midpoint
    switch e(5,k)
        case {some_edge_labels}
            % Fill in hmatrix,rmatrix or qmatrix,gmatrix
        case {another_list_of_edge_labels}
            % Fill in hmatrix,rmatrix or qmatrix,gmatrix
        otherwise
            % Fill in hmatrix,rmatrix or qmatrix,gmatrix
    end
end
```

For each column k in e, entry k of rmatrix is the value of rmatrix at the first point in the edge, and entry ne + k is the value at the second point in the edge. For example, if $r = x^2 + y^4$, then write these lines:

```
rmatrix(k) = x1^2 + y1^4;
rmatrix(k+ne) = x2^2 + y2^4;
```

The syntax for hmatrix is identical: entry k of hmatrix is the value of r at the first point in the edge, and entry k + ne is the value at the second point in the edge.

For each column k in e, entry k of qmatrix is the value of qmatrix at the midpoint in the edge. For example, if $q = x^2 + y^4$, then write these lines:

```
qmatrix(k) = xm^2 + ym^4;
```

The syntax for gmatrix is identical: entry k of gmatrix is the value of gmatrix at the midpoint in the edge.

If the coefficients depend on the solution u, use the element `u(e(1,k))` as the solution value at the first point of edge k, and `u(e(2,k))` as the solution value at the second point of edge k.

For example, consider the following geometry, a rectangle with a circular hole.
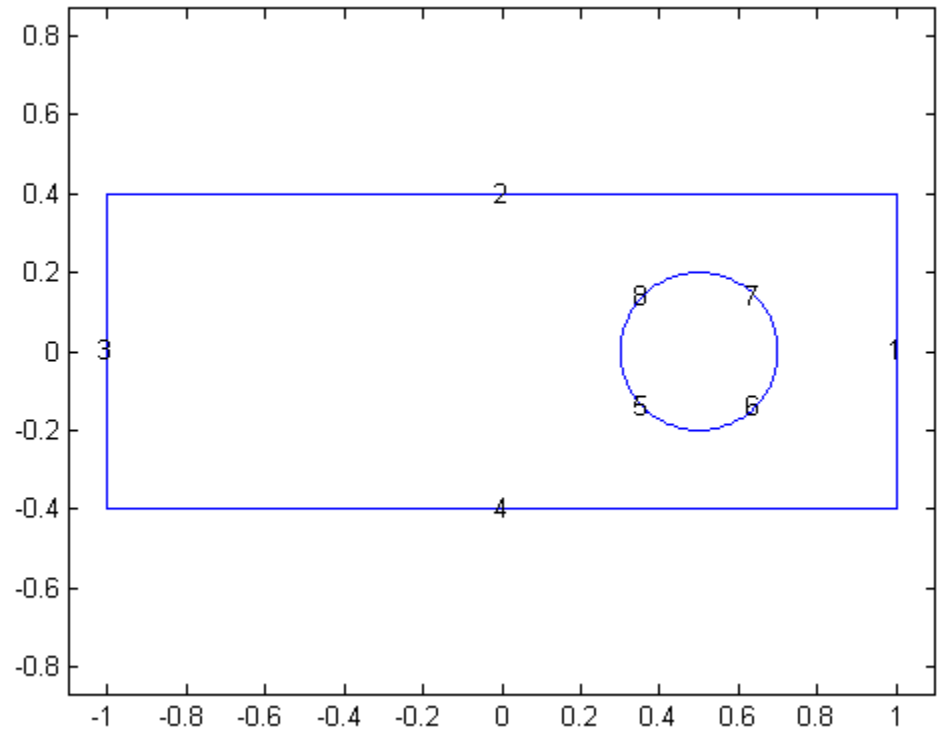
Code for generating the figure

```
% Rectangle is code 3, 4 sides,
% followed by x-coordinates and then y-coordinates
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
% Circle is code 1, center (.5,0), radius .2
C1 = [1,.5,0,.2]';
```

```
% Pad C1 with zeros to enable concatenation with R1
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];

% Names for the two geometric objects
ns = (char('R1','C1'))';

% Set formula
sf = 'R1-C1';

% Create geometry
gd = decsg(geom,sf,ns);

% View geometry
pdegplot(gd,'edgeLabels','on')
xlim([-1.1 1.1])
axis equal
```

Suppose the boundary conditions on the outer boundary (segments 1 through 4) are Dirichlet, with the value $u(x,y) = t(x - y)$, where t is time. Suppose the circular boundary (segments 5 through 8) has a generalized Neumann condition, with $q = 1$ and $g = x^2 + y^2$.

Write the following boundary file to represent the boundary conditions:

```
function [qmatrix,gmatrix,hmatrix,rmatrix] = pdebound(p,e,u,time)

ne = size(e,2); % number of edges
qmatrix = zeros(1,ne);
gmatrix = qmatrix;
hmatrix = zeros(1,2*ne);
rmatrix = hmatrix;

for k = 1:ne
    x1 = p(1,e(1,k)); % x at first point in segment
    x2 = p(1,e(2,k)); % x at second point in segment
    xm = (x1 + x2)/2; % x at segment midpoint
    y1 = p(2,e(1,k)); % y at first point in segment
    y2 = p(2,e(2,k)); % y at second point in segment
    ym = (y1 + y2)/2; % y at segment midpoint
```

```
        switch e(5,k)
            case {1,2,3,4} % rectangle boundaries
                hmatrix(k) = 1;
                hmatrix(k+ne) = 1;
                rmatrix(k) = time*(x1 - y1);
                rmatrix(k+ne) = time*(x2 - y2);
            otherwise % same as case {5,6,7,8}, circle boundaries
                qmatrix(k) = 1;
                gmatrix(k) = xm^2 + ym^2;
        end
    end
```

Related Examples
- "Scalar PDE Functional Form and Calling Syntax" on page 2-22
- "Deflection of a Piezoelectric Actuator" on page 3-19

Boundary Conditions for PDE Systems

The general mixed-boundary conditions for PDE systems (see "Systems of PDEs" on page 2-13) are

$$\mathbf{hu} = \mathbf{r}$$
$$\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{qu} = \mathbf{g} + \mathbf{h}'\mu.$$

The notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ means 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 $\mathbf{n} = (\cos(\alpha), \sin(\alpha))$. For each edge segment there are M Dirichlet conditions and the h-matrix is

M-by-N, $M \geq 0$. The generalized Neumann condition contains a source $\mathbf{h}'\mu$ where the solver computes Lagrange multipliers μ such that the Dirichlet conditions are satisfied.

A PDE solver, such as assempde or adaptmesh, passes a matrix p of points and e of edges. e has seven rows and ne columns, where you do not necessarily know in advance the size ne.

- p is a 2-by-N_p matrix, where p(1,k) is the *x*-coordinate of point k, and p(2,k) is the *y*-coordinate of point k.

- e is a 7-by-ne matrix, where

  - e(1,k) is the index of the first point of edge k.

  - e(2,k) is the index of the second point of edge k.

  - e(5,k) is the label of the geometry edge of edge k (see "Identify Boundary Labels" on page 2-60).

  e contains an entry for every finite element edge that lies on an exterior boundary.

Let N be the dimension of the system of PDEs; see "Systems of PDEs" on page 2-13. Use the following template for your boundary file.

```
function [qmatrix,gmatrix,hmatrix,rmatrix] = pdebound(p,e,u,time)

N = 3; % Set N = the number of equations
ne = size(e,2); % number of edges
qmatrix = zeros(N^2,ne);
gmatrix = zeros(N,ne);
hmatrix = zeros(N^2,2*ne);
rmatrix = zeros(N,2*ne);

for k = 1:ne
    x1 = p(1,e(1,k)); % x at first point in segment
    x2 = p(1,e(2,k)); % x at second point in segment
    xm = (x1 + x2)/2; % x at segment midpoint
    y1 = p(2,e(1,k)); % y at first point in segment
    y2 = p(2,e(2,k)); % y at second point in segment
    ym = (y1 + y2)/2; % y at segment midpoint
    switch e(5,k)
        case {some_edge_labels}
            % Fill in hmatrix,rmatrix or qmatrix,gmatrix
        case {another_list_of_edge_labels}
            % Fill in hmatrix,rmatrix or qmatrix,gmatrix
        otherwise
            % Fill in hmatrix,rmatrix or qmatrix,gmatrix

    end
end
```

For the boundary file, you represent the matrix **h** for each edge segment as a vector, taking the matrix column-wise, as hmatrix(:). Column k of hmatrix corresponds to the matrix at the first edge point e(1,k), and column k + ne corresponds to the matrix at the second edge point e(2,k).

Similarly, you represent each vector **r** for an edge as a column in the matrix rmatrix. Column k corresponds to the vector at the first edge point e(1,k), and column k + ne corresponds to the vector at the second edge point e(2,k).

Represent the entries for the matrix **q** for each edge segment as a vector, qmatrix(:), similar to the matrix hmatrix(:). Similarly, represent **g** for each edge segment is a column vector in the matrix gmatrix. Unlike **h** and **r**, which have two columns for each segment, **q** and **g** have just one column for each segment, which is the value of the function at the midpoint of the edge segment.

For example, consider the following geometry, a rectangle with a circular hole.
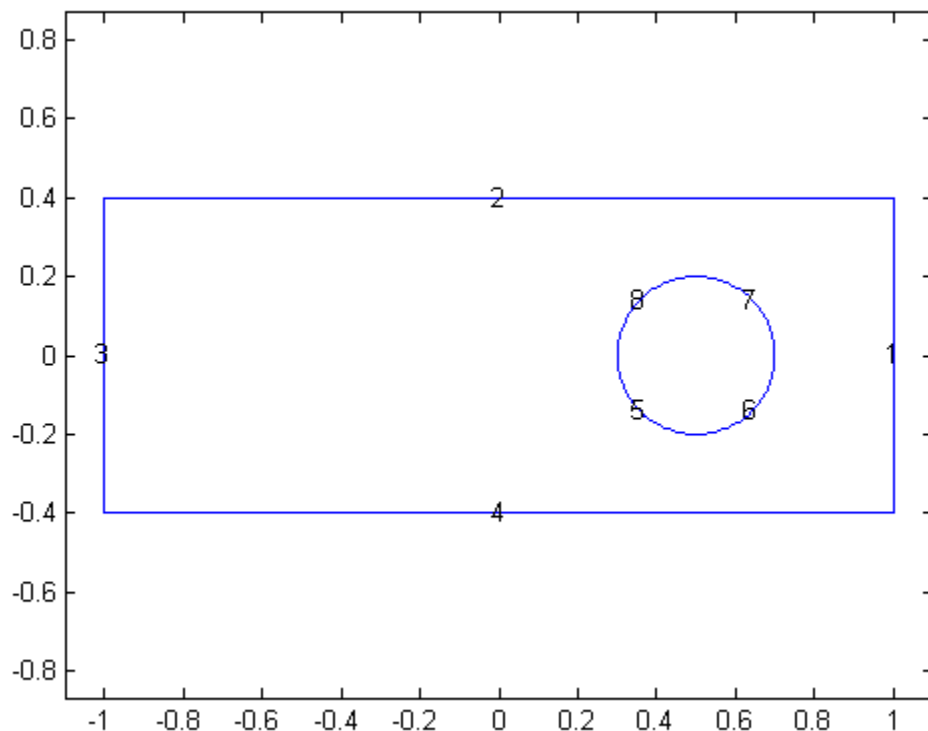
Code for generating the figure

```
% Rectangle is code 3, 4 sides,
% followed by x-coordinates and then y-coordinates
```

```
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
% Circle is code 1, center (.5,0), radius .2
C1 = [1,.5,0,.2]';
% Pad C1 with zeros to enable concatenation with R1
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];

% Names for the two geometric objects
ns = (char('R1','C1'))';

% Set formula
sf = 'R1-C1';

% Create geometry
gd = decsg(geom,sf,ns);

% View geometry
pdegplot(gd,'edgeLabels','on')
xlim([-1.1 1.1])
axis equal
```

Suppose $N = 3$. Suppose the boundary conditions are mixed. There is M = 1 Dirichlet condition:

- The first component of $u = 0$ on the rectangular segments (numbers 1–4). So $\mathbf{h}(1,1) = 1$ and $\mathbf{r}(1) = 0$ for those segments.

- The second components of $u = 0$ on the circular segments (numbers 5–8). So $\mathbf{h}(2,2) = 1$ and $\mathbf{r}(2) = 0$ for those segments.

- On the rectangular segments (numbers 1–4),

$$\mathbf{q} = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

  and

$$\mathbf{g} = \begin{pmatrix} 1 + x^2 \\ 0 \\ 1 + y^2 \end{pmatrix}$$

- On the circular segments (numbers 5–8),

$$\mathbf{q} = \begin{pmatrix} 0 & 1 + x^2 & 2 + y^2 \\ 0 & 0 & 0 \\ 1 + x^4 & 1 + y^4 & 0 \end{pmatrix}$$

and

$$\mathbf{g} = \begin{pmatrix} \cos(\pi x) \\ 0 \\ \tanh(x + y) \end{pmatrix}$$

Write the following boundary file to represent the boundary conditions:

```
function [qmatrix,gmatrix,hmatrix,rmatrix] = pdebound(p,e,u,time)

N = 3;
ne = size(e,2); % number of edges
qmatrix = zeros(N^2,ne);
gmatrix = zeros(N,ne);
hmatrix = zeros(N^2,2*ne);
rmatrix = zeros(N,2*ne);

for k = 1:ne
    x1 = p(1,e(1,k)); % x at first point in segment
    x2 = p(1,e(2,k)); % x at second point in segment
    xm = (x1 + x2)/2; % x at segment midpoint
    y1 = p(2,e(1,k)); % y at first point in segment
    y2 = p(2,e(2,k)); % y at second point in segment
    ym = (y1 + y2)/2; % y at segment midpoint
    switch e(5,k)
        case {1,2,3,4}
            hk = zeros(N);
```

```
        hk(1,1) = 1;
        hk = hk(:);
        hmatrix(:,k) = hk;
        hmatrix(:,k+ne) = hk;

        rk = zeros(N,1); % Not strictly necessary
        rmatrix(:,k) = rk; % These are already 0
        rmatrix(:,k+ne) = rk;

        qk = zeros(N);
        qk(1,2) = 1;
        qk(1,3) = 1;
        qk(3,1) = 1;
        qk(3,2) = 1;
        qk = qk(:);
        qmatrix(:,k) = qk;

        gk = zeros(N,1);
        gk(1) = 1+xm^2;
        gk(3) = 1+ym^2;
        gmatrix(:,k) = gk;

    case {5,6,7,8}
        hk = zeros(N);
        hk(2,2) = 1;
        hk = hk(:);
        hmatrix(:,k) = hk;
        hmatrix(:,k+ne) = hk;

        rk = zeros(N,1); % Not strictly necessary
        rmatrix(:,k) = rk; % These are already 0
        rmatrix(:,k+ne) = rk;

        qk = zeros(N);
        qk(1,2) = 1+xm^2;
        qk(1,3) = 2+ym^2;
        qk(3,1) = 1+xm^4;
        qk(3,2) = 1+ym^4;
        qk = qk(:);
        qmatrix(:,k) = qk;
```

```
                       gk = zeros(N,1);
                       gk(1) = cos(pi*xm);
                       gk(3) = tanh(xm*ym);
                       gmatrix(:,k) = gk;

               end
         end
```

Related Examples
- "Scalar PDE Functional Form and Calling Syntax" on page 2-22
- "Deflection of a Piezoelectric Actuator" on page 3-19

Tooltip Displays for Mesh and Plots

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. 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.

Mesh Data

A mesh consists of three data structures:

- p (points) is a 2-by-Np matrix of points, where Np is the number of points in the mesh. Each column p(:,k) consists of the *x*-coordinate of point k in p(1,k), and the *y*-coordinate of point k in p(2,k).

- e (edges) is a 7-by-Ne matrix of edges, where Ne is the number of edges in the mesh. An edge is a pair of points in p containing a boundary between subdomains, or containing an outer boundary, as follows:

  - e(1,k) is the index of the first point in edge k.

  - e(2,k) is the index of the second point in edge k.

  - e(3,k) is the parameter value at the first point of edge k. The parameter value is related to arc length along the edge.

  - e(4,k) is the parameter value at the second point of edge k.

  - e(5,k) is the segment number of the geometry containing the edge. The segment number is inherited from the edge segments in the base geometry. You can see geometry segment numbers using the command pdegplot(geom,'edgeLabels','on').

  - e(6,k) is the subdomain number on the left of the edge (subdomain 0 is the exterior of the geometry), where direction along the edge is given by increasing parameter value.

  - e(7,k) is the subdomain number on the right of the edge.

- t (triangles) is a 4-by-Nt matrix of triangles, where Nt is the number of triangles in the mesh. t(1,k), t(2,k), and t(3,k) contain indices to the three points in p that form triangle k. The points are in counterclockwise order. t(4,k) contains the subdomain number of the triangle.

Generate an initial mesh using the initmesh function. Refine the mesh using the refinemesh function. Improve mesh quality without introducing new points using the jigglemesh function.

Related Examples
- "Scalar PDE Functional Form and Calling Syntax" on page 2-22
- "Solve Poisson's Equation on a Unit Disk" on page 3-71

Adaptive Mesh Refinement

In this section...

Improving Solution Accuracy Using Mesh Refinement

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.

Error Estimate for the FEM Solution

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 elementwise 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 $\|\cdot\|$:

$$\|\nabla(u - u_h)\| \leq \alpha \|hf\| + \beta D_h(u_h),$$

where $h = h(x)$ is the local mesh size, and

$$D_h(v) = \left(\sum_{\tau \in E_1} h_\tau^2 \left[\frac{\partial v}{\partial n_\tau} \right]^2 \right)^{1/2}.$$

The braced quantity is the jump in normal derivative of v across edge τ, h_τ 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 elementwise 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

$$E(K) = \alpha \left\| h(f - au) \right\|_K + \beta \left(\frac{1}{2} \sum_{\tau \in \partial K} h_\tau^2 \left(\mathbf{n}_\tau \cdot c \nabla u_h \right)^2 \right)^{1/2},$$

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.

Mesh Refinement Functions

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.

Mesh Refinement 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.

3

Solving PDEs

Set Up and Solve Your PDE Problem

The layout of the PDE Toolbox GUI represents the sequence of steps you perform to solve a PDE. Specifically, the order of the GUI menu and toolbar items represent these actions you perform:

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

---

1 Start the PDE Toolbox GUI using pdetool.

At this point, the GUI 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.

2 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.

3 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.

4 Move to the next step in the PDE solving process by clicking the ∂Ω 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.

5 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.

6 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.

7 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.

---

8 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.

9 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.

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 *x-y* 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

$$
\begin{pmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{pmatrix} = \frac{E}{1-v^2} \begin{pmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & \dfrac{1-v}{2} \end{pmatrix} \begin{pmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{pmatrix},
$$

where o_x and o_y are the normal stresses in the *x* and *y* directions, and τ_{xy} is the *shear stress.* The material properties are expressed as a combination of E, the *elastic modulus* or *Young's modulus*, and *v*, 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

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) = \mathbf{k},$$

where $\mathbf{c}$ is a rank four tensor (see "c for Systems" on page 2-42), 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

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

and μ in turn is defined by

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

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

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

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \boldsymbol{u}) = \lambda \mathbf{d} u$$
$$\mathbf{d} = \begin{pmatrix} \rho & 0 \\ 0 & \rho \end{pmatrix}.$$

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:

- $u_x = \dfrac{\partial u}{\partial x}$

- $u_y = \dfrac{\partial u}{\partial y}$

- $v_x = \dfrac{\partial v}{\partial x}$

- $v_y = \dfrac{\partial v}{\partial y}$
- 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 *v* (`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.
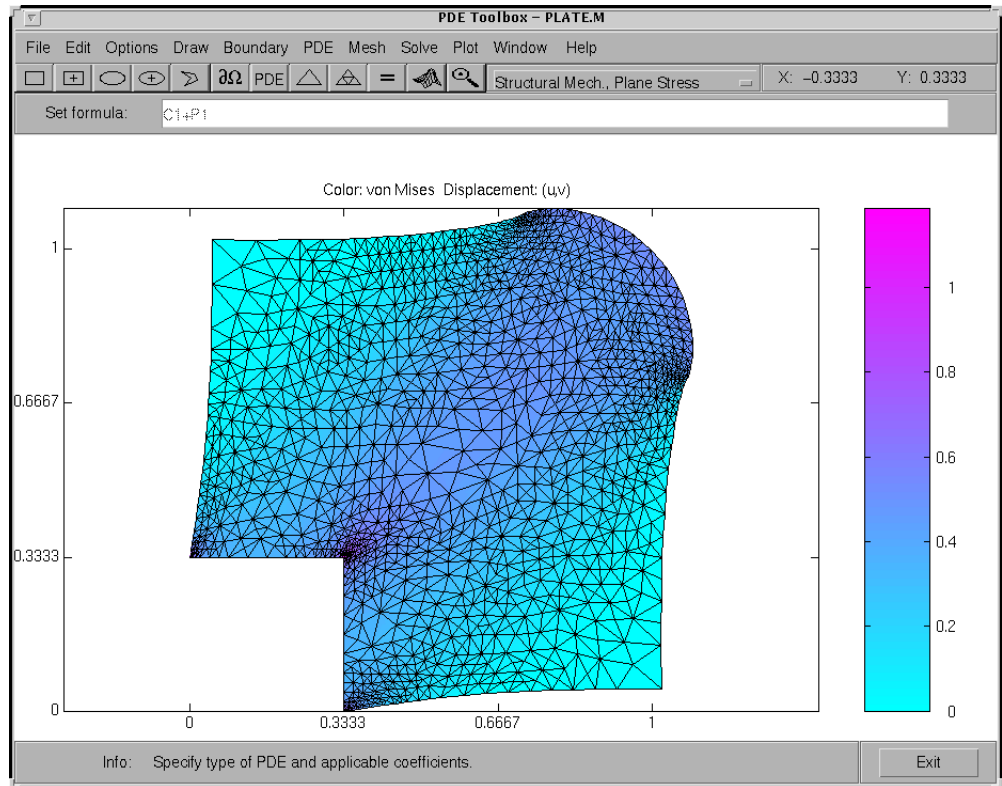
Visualization of the von Mises Effective Stress and the Displacements Using Deformed Mesh

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

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

- The von Mises effective stress is computed as

$$\sqrt{\left(\sigma_1^2 + \sigma_2^2\right)\left(v^2 - v + 1\right) + \sigma_1\sigma_2\left(2v^2 - 2v - 1\right)}.$$

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.

Clamped, Square Isotropic Plate With a Uniform Pressure Load

This example shows how to calculate the deflection of a structural plate acted on by a pressure loading using the Partial Differential Equation Toolbox™.

PDE and Boundary Conditions For A Thin Plate

The partial differential equation for a thin, isotropic plate with a pressure loading is

$$\nabla^2(D\nabla^2 w) = p$$

where D is the bending stiffness of the plate given by

$$D = \frac{Eh^3}{12(1-\nu^2)}$$

and E is the modulus of elasticity, ν is Poisson's ratio, and h is the plate thickness. The transverse deflection of the plate is w and p is the pressure load.

The boundary conditions for the clamped boundaries are $w = 0$ and $w' = 0$ where w' is the derivative of w normal to the boundary.

The Partial Differential Equation Toolbox™ cannot directly solve the fourth order plate equation shown above but this can be converted to the following two second order partial differential equations.

$$\nabla^2 w = v$$
$$D\nabla^2 v = p$$

where v is a new dependent variable. We now require boundary conditions for both w and v. The boundary conditions of the original, fourth order problem will be approximately satisfied if we define $w' = 0$ and $v' = Lv$ where L is a large number. This is the "Stiff Spring" method of boundary condition enforcement discussed in the Partial Differential Equation Toolbox™ User's Guide.

In the Partial Differential Equation Toolbox™ definition for an elliptic system, the w and v dependent variables are u(1) and u(2).

Problem Parameters

```
E=1.0e6; % modulus of elasticity
gnu=.3; % Poisson's ratio
thick=.1; % plate thickness
len=10.0; % side length for the square plate
hmax=len/20; % mesh size parameter
D=E*thick^3/(12*(1 - gnu^2));
pres=2; % external pressure
```

Geometry and Mesh

For a single square, the geometry and mesh are easily defined as shown below.

```
gdmTrans=[3 4 0 len len 0 0 0 len len];
sf = 'S1';
nsmTrans = 'S1';
g = decsg(gdmTrans', sf, nsmTrans');
[p, e, t]=initmesh(g, 'Hmax', hmax);
```

Boundary Conditions

Most of this example was created using simple command-line programming. However, it is more convenient to create the boundary condition matrix using the pdetool GUI and the procedure described below.

1 A single square is created in pdetool. The size of the square is irrelevant since, for this example, we are using the pdetool GUI only for creating the boundary condition matrix.

2 In the Options menu, the Application is set to Structural Mechanics, Plane Stress. The only reason for setting this particular application type is that the equations for plane stress and the thin plate are both two-equation systems. The boundary condition dialog box used in the next step will contain the appropriate entries for a two-equation system.

3 All four edges of the square are selected in Boundary Mode and the Specify Boundary Conditions option invoked. The Condition Type is set to Neumann and q21 is set to 1e4, the "large number" used for this example.

4 The Export Decomposed Geometry, Boundary Cond's option is invoked from the Boundary menu to add the matrix b to the MATLAB workspace. This matrix was printed and inserted directly in the script as shown below.

```
b = [
      2     2     2     2
      0     0     0     0
      1     1     1     1
      5     5     5     5
      1     1     1     1
      1     1     1     1
      1     1     1     1
      1     1     1     1
     48    48    48    48
     49    49    49    49
     46    46    46    46
     48    48    48    48
    101   101   101   101
     52    52    52    52
     48    48    48    48
     48    48    48    48
     48    48    48    48
     48    48    48    48
     48    48    48    48
     48    48    48    48
     49    49    49    49
     48    48    48    48
     48    48    48    48
     49    49    49    49
     48    48    48    48
     48    48    48    48
     ];
```

Coefficient Definition

The documentation for `assempde` shows the required formats for the a and c matrices in the section titled "PDE Coefficients for System Case". The most convenient form for c in this example is $n_c = 3N$ from the table where N is the number of differential equations. In this example $N = 2$. The six-row by one-column c matrix is defined below. The entries in the a matrix and f vector follow directly from the definition of the two-equation system shown above.

```
c=[1; 0; 1; D; 0; D];
a=[0; 0; 1; 0];
f=[0; pres];
```

Finite Element and Analytical Solutions

The solution is calculated using the `assempde` function and the transverse deflection is plotted using the `pdeplot` function. For comparison, the transverse deflection at the plate center is also calculated using an analytical solution to this problem.

```
u=assempde(b,p,e,t,c,a,f);
numNodes = size(p,2);
fprintf('Transverse deflection at plate center(PDE Toolbox)=%12.4e\n', mi
pdeplot(p, e, t, 'xydata', u(1:numNodes), 'contour', 'on');
title 'Transverse Deflection'

% compute analytical solution
wMax = -.0138*pres*len^4/(E*thick^3);
fprintf('Transverse deflection at plate center(analytical)=%12.4e\n', wMa
```

```
Transverse deflection at plate center(PDE Toolbox)= -2.7523e-01
Transverse deflection at plate center(analytical)= -2.7600e-01
```

Transverse Deflection
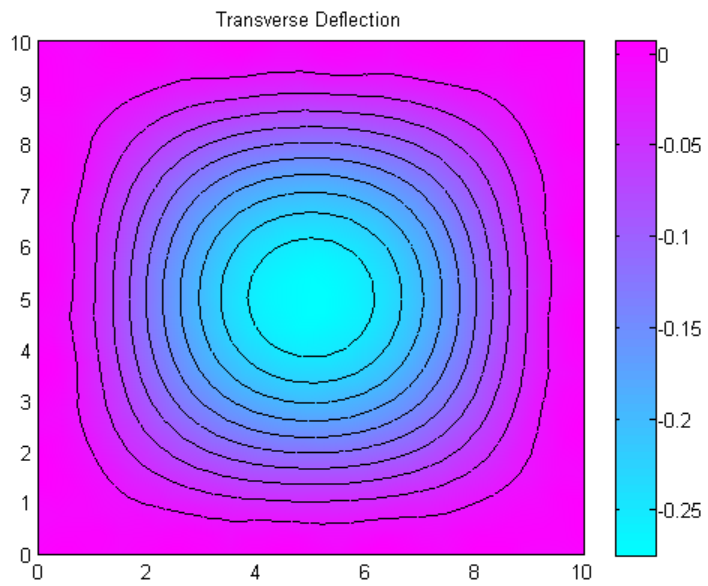
Deflection of a Piezoelectric Actuator

This example shows how to solve a coupled elasticity-electrostatics problem using the Partial Differential Equation Toolbox™. Piezoelectric materials deform when a voltage is applied. Conversely, a voltage is produced when a piezoelectric material is deformed.

Analysis of a piezoelectric part requires the solution of a set of coupled partial differential equations with deflections and electrical potential as dependent variables. One of the main objectives of this example is to show how such a system of coupled partial differential equations can be solved using PDE Toolbox.

PDE For a Piezoelectric Solid

The behavior of the solid is described by the equilibrium equations from elasticity theory

$$-\nabla \cdot \sigma = f$$

where σ is the stress tensor and f is the body force vector and Gauss' Law

$$\nabla \cdot D = \rho$$

where D is the electric displacement and ρ is the distributed, free charge. These two PDE systems can be combined into the following single system

$$-\nabla \cdot \left\{ \begin{array}{c} \sigma \\ D \end{array} \right\} = \left\{ \begin{array}{c} f \\ -\rho \end{array} \right\}$$

In 2D, σ has the components σ_{11}, σ_{22}, and $\sigma_{12} = \sigma_{21}$ and D has the components D_1 and D_2.

The constitutive equations for the material define the stress tensor and electric displacement vector in terms of the strain tensor and electric field. For a 2D, orthotropic, piezoelectric material under plane stress conditions these are commonly written as

$$
\left\{ \begin{array}{c} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \\ D_1 \\ D_2 \end{array} \right\} = \left[\begin{array}{ccccc} C_{11} & C_{12} & & e_{11} & e_{31} \\ C_{12} & C_{22} & & e_{13} & e_{33} \\ & & G_{12} & e_{14} & e_{34} \\ e_{11} & e_{13} & e_{14} & \mathcal{E}_1 & \\ e_{31} & e_{33} & e_{34} & & \mathcal{E}_2 \end{array} \right] \left\{ \begin{array}{c} \epsilon_{11} \\ \epsilon_{22} \\ \gamma_{12} \\ E_1 \\ E_2 \end{array} \right\}
$$

where C_{ij} are the elastic coefficients, $\mathcal{E}_i$ are the electrical permittivities, and e_{ij} are the piezoelectric stress coefficients. The piezoelectric stress coefficients are written to conform to conventional notation in piezoelectric materials where the z-direction (3-direction) is aligned with the "poled" direction of the material. For the 2D analysis, we want the poled direction to be aligned with the y-axis.

Finally, the strain vector can be written in terms of the x-displacement, u, and y-displacement, v as

$$
\left\{ \begin{array}{c} \epsilon_{11} \\ \epsilon_{22} \\ \gamma_{12} \end{array} \right\} = \left\{ \begin{array}{c} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{array} \right\}
$$

and the electric field written in terms of the electrical potential, ϕ, as

$$
\left\{ \begin{array}{c} E_1 \\ E_2 \end{array} \right\} = - \left\{ \begin{array}{c} \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{array} \right\}
$$

See reference 2, for example, for a more complete description of the piezoelectric equations.

The strain-displacement equations and electric field equations above can be substituted into the constitutive equations to yield a system of equations for the stresses and electrical displacements in terms of displacement and electrical potential derivatives. If the resulting equations are substituted into the PDE system equations, we have a system of equations that involve the divergence of the displacement and electrical potential derivatives. Arranging these equations to match the form required by PDE Toolbox will be the topic for the next section.

Converting the Equations To PDE Toolbox Form

The PDE Toolbox requires a system of elliptic equations to be expressed in the form

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{a}\mathbf{u} = \mathbf{f}$$

or in tensor form

$$-\frac{\partial}{\partial x_k}\left(c_{ijkl}\frac{\partial u_j}{\partial x_l}\right) + a_{ij}u_j = f_i$$

where summation is implied by repeated indices. For the 2D piezoelectric system described above, the PDE Toolbox system vector $\mathbf{u}$ is

$$\mathbf{u} = \left\{\begin{array}{c} u \\ v \\ \phi \end{array}\right\}$$

This is an $N = 3$ system. The gradient of $\mathbf{u}$ is given by

$$\nabla \mathbf{u} = \left\{\begin{array}{c} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \\ \hline \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial y} \\ \hline \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{array}\right\}$$

The documentation for the function `assempde` shows that it is convenient to view the tensor c_{ijkl} as an $N \times N$ matrix of 2×2 submatrices. The most convenient form for the c input argument for this symmetric, $N = 3$ system has 21 rows in c and is shown in complete form in the `assempde` documentation. It is repeated here for convenience.

$$\begin{bmatrix} c(1) & c(2) & c(4) & c(6) & c(11) & c(13) \\ \cdot & c(3) & c(5) & c(7) & c(12) & c(14) \\ \cdot & \cdot & c(8) & c(9) & c(15) & c(17) \\ \cdot & \cdot & \cdot & c(10) & c(16) & c(18) \\ \cdot & \cdot & \cdot & \cdot & c(19) & c(20) \\ \cdot & \cdot & \cdot & \cdot & \cdot & c(21) \end{bmatrix}$$

For the purposes of mapping terms from constitutive equations to the form required by PDE Toolbox it is useful to write the c tensor and solution gradient in the following form

$$\begin{bmatrix} c_{1111} & c_{1112} & c_{1211} & c_{1211} & c_{1311} & c_{1311} \\ \cdot & c_{1122} & c_{1211} & c_{1211} & c_{1311} & c_{1311} \\ \cdot & \cdot & c_{2211} & c_{2212} & c_{2311} & c_{2312} \\ \cdot & \cdot & \cdot & c_{2222} & c_{2321} & c_{2322} \\ \cdot & \cdot & \cdot & \cdot & c_{3311} & c_{3311} \\ \cdot & \cdot & \cdot & \cdot & \cdot & c_{3311} \end{bmatrix} \begin{Bmatrix} \frac{\partial u}{\partial z} \\ \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial z} \\ \frac{\partial v}{\partial y} \\ \frac{\partial \phi}{\partial z} \\ \frac{\partial \phi}{\partial y} \end{Bmatrix}$$

From this equation the traditional constitutive coefficients can be mapped to the form required for the PDE Toolbox c matrix. Note the minus sign in the equations for electric field. This minus must be incorporated into the c matrix to match the PDE Toolbox convention. This is shown explicitly below.

$$\begin{bmatrix} C_{11} & \cdot & \cdot & C_{12} & -e_{11} & -e_{31} \\ \cdot & G_{12} & G_{12} & \cdot & -e_{14} & -e_{34} \\ \cdot & \cdot & G_{12} & \cdot & -e_{14} & -e_{34} \\ \cdot & \cdot & \cdot & C_{22} & -e_{13} & -e_{33} \\ \cdot & \cdot & \cdot & \cdot & -\mathcal{E}_1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & -\mathcal{E}_2 \end{bmatrix} \begin{Bmatrix} \frac{\partial u}{\partial z} \\ \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial z} \\ \frac{\partial v}{\partial y} \\ \frac{\partial \phi}{\partial z} \\ \frac{\partial \phi}{\partial y} \end{Bmatrix}$$

Piezoelectric Bimorph Actuator Model

Now that we have defined the equations for a 2D piezoelectric material, we are ready to apply these to a specific model. The model is a two-layer

cantilever beam that has been extensively studied (e.g. refs 1 and 2). It is defined as a "bimorph" because although both layers are made of the same Polyvinylidene Fluoride (PVDF) material, in the top layer the polarization direction points up (y direction) and in the bottom layer, it points down. A schematic of the cantilever beam is shown in the figure below.
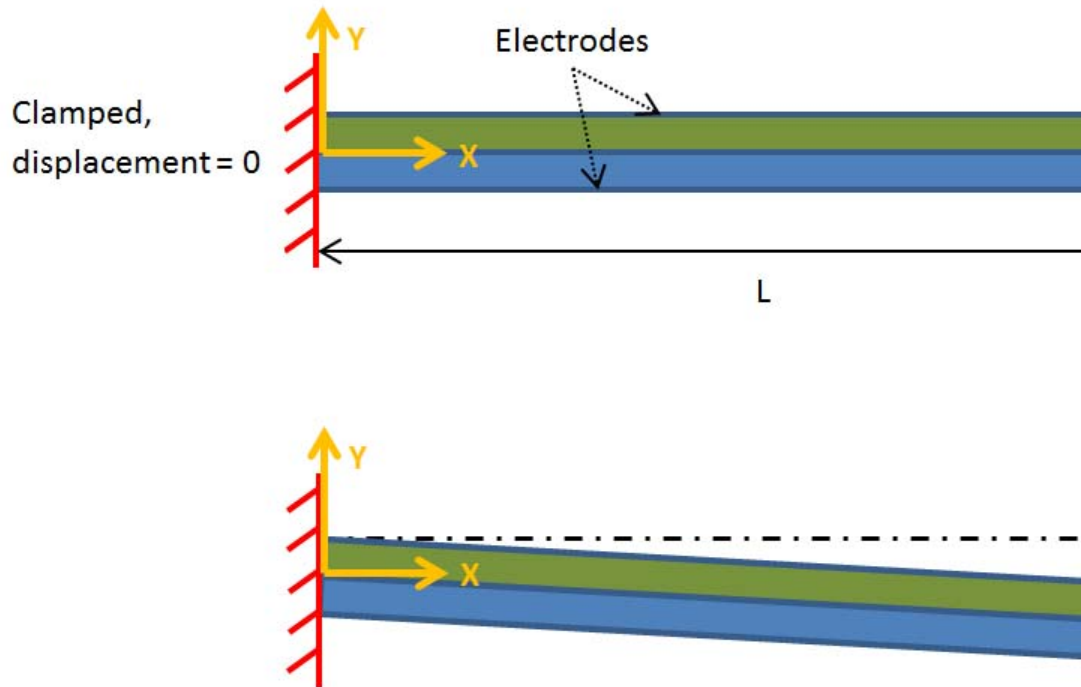
This figure is not to scale; the actual thickness/length ratio is 50 so the beam is very slender. When a voltage is applied between the lower and upper surfaces of the beam, it deflects in the y-direction; one layer shortens and the

other layer lengthens. Devices of this type can be designed to provide the required motion or force for different applications.

Geometry and Mesh

The simple two-layer geometry of the beam can be created by defining the sum of two rectangles.

```
L = 50e-3; % beam length in meters
H = 1e-3; % overall height of the beam
H2 = H/2; % height of each layer in meters
% The two lines below contain the columns of the
% geometry description matrix (GDM) for the two rectangular layers.
% The GDM  is the first input argument to decsg and describes the
% basic geometric entities in the model.
topLayer = [3 4 0 L L 0 0 0 H2 H2];
bottomLayer = [3 4 0 L L 0 -H2 -H2 0 0];
gdmTrans = [topLayer; bottomLayer];
g = decsg(gdmTrans', 'R1+R2', ['R1'; 'R2']');
pdegplot(g); axis equal; title 'Two-layer Piezoelectric Cantilever Beam'
xlabel 'X-coordinate, meters'
ylabel 'Y-coordinate, meters'
% We need a relatively fine mesh with maximum element size roughly equal H/
% to accurately model the bending of the beam.
hmax = H/16;
[p, e, t] = initmesh(g, 'Hmax', hmax);
```

```
warning: Approximately 25600 triangles will be generated.
```

Two-layer Piezoelectric Cantilever Beam
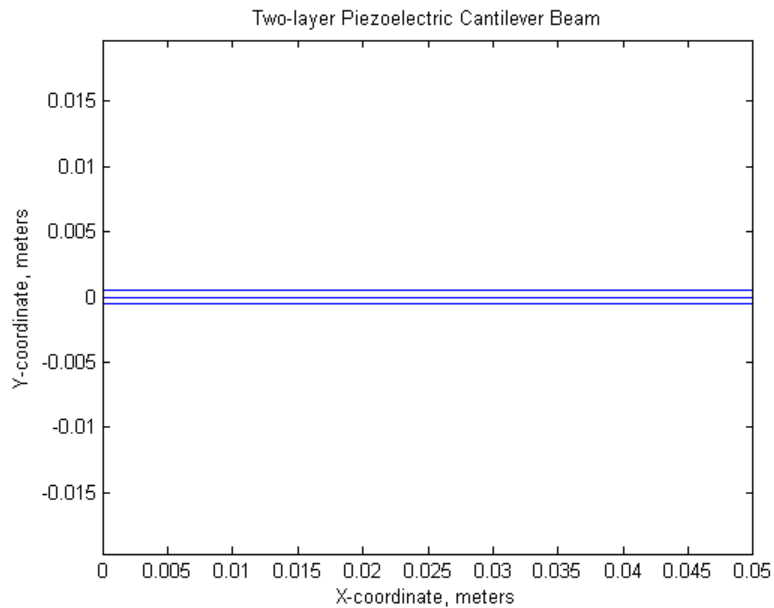
Material Properties

The material in both layers of the beam is Polyvinylidene Fluoride (PVDF), a thermoplastic polymer with piezoelectric behavior.

```
E = 2.0e9; % Elastic modulus, N/m^2
NU = 0.29; % Poisson's ratio
G = 0.775e9; % Shear modulus, N/m^2
d31 = 2.2e-11; % Piezoelectric strain coefficients, C/N
d33 = -3.0e-11;
% relative electrical permittivity of the material
relPermittivity = 12;
% electrical permittivity of vacuum
permittivityFreeSpace = 8.8541878176200e-9; % F/m
C11 = E/(1-NU^2); C12 = NU*C11;
c2d = [C11 C12 0; C12 C11 0; 0 0 G];
pzeD = [0 d31; 0 d33; 0 0];
% The piezoelectric strain coefficients for PVDF are
% given above but the constitutive relations in the
```

```
% finite element formulation require the
% piezoelectric stress coefficients. These are calculated on the next
% line (for details see, for example, reference 2).
pzeE = c2d*pzeD;
% As discussed above, it is convenient to view the 21 coefficients
% required by assempde as a 3 x 3 array of 2 x 2 submatrices.
% The cij matrices defined below are the 2 x 2 submatrices in the upper
% triangle of this array.
c11 = [c2d(1,1) c2d(1,3) c2d(3,3)];
c12 = [c2d(1,3) c2d(1,2); c2d(3,3) c2d(2,3)];
c22 = [c2d(3,3) c2d(2,3) c2d(2,2)];
c13 = [pzeE(1,1) pzeE(1,2); pzeE(3,1) pzeE(3,2)];
c23 = [pzeE(3,1) pzeE(3,2); pzeE(2,1) pzeE(2,2)];
c33 = [relPermittivity 0 relPermittivity]*permittivityFreeSpace;
```

Function To Return C Coefficients

The c-matrix for this N=3 system is symmetric. From the documentation for
assempde, we see that the most convenient form for defining the c-matrix has
21 rows defining the upper triangle of the matrix.

```
c = @(p, t, u, t0) calcCMatPiezoActuator(p, t, c11, c12, c22, c13, c23, c33
% The function shown below is called by the PDE Toolbox routines to
% return the required 21 entries in the c-matrix.
type calcCMatPiezoActuator


function c = calcCMatPiezoActuator( p, t, c11, c12, c22, c13, c23, c33 )
%CALCCMATPIEZOACTUATOR C-matrix for piezoelectric actuator example
%   c = CALCCMATPIEZOACTUATOR( p, t, c11, c12, c22, c13, c23, c33 )
%   returns the 'c' coefficient matrix for the piezoelectric actuator
%   example given the point and element matrices along with the
%   constitutive submatrices (cij) for the PVDF material.

% Copyright 2012 The MathWorks, Inc.
% $Revision: 1.1.6.1 $  $Date: 2012/05/08 20:26:11 $

numElems = size(t,2);
c=zeros(21,numElems);
%
```

```
% Although the material in both layers is PVDF, in the top layer
% the polarization direction points up (y direction) and in the
% bottom layer, it points down. That is, the top layer has d-coefficients
% that are the negative of those in the down.
%
% The code below examines the y-location of the centroid of each
% triangular element and assigns the correct material properties to
% element depending on whether it is in the top or bottom layer.
%
ctop = [c11(:); c12(:); c22(:); -c13(:); -c23(:); -c33(:)];
cbot = [c11(:); c12(:); c22(:);  c13(:);  c23(:); -c33(:)];
% calculate y-coordinate of triangle centers
yCenter=(p(2,t(1,:)) + p(2,t(2,:)) + p(2,t(3,:)))/3;
for i=1:numElems
  if(yCenter(i) < 0)
    c(:,i) = cbot;
  else
    c(:,i) = ctop;
  end
end

end
```

Boundary Condition Definition

The function below is referred to as a "boundary file" in the PDE Toolbox documentation. It returns the appropriate q and g matrices defining the Neumann boundary conditions and the h and r matrices defining Dirichlet boundary conditions on each exterior, finite element edge.

For this example, the top edge (edge 1) has the voltage prescribed as 100 volts. The bottom edge (edge 2) has the voltage prescribed as 0 volts (i.e. grounded). The left edge (edges 6 and 7) have the u and v displacements equal zero (i.e. clamped). The stress and charge are zero on the right edge (i.e. q=0).

```
V = 100;
b = @(p, e, u, time) boundaryFilePiezoActuator(p, e, V);
type boundaryFilePiezoActuator
```

```
function [ q, g, h, r ] = boundaryFilePiezoActuator( p, e, V )
%BOUNDARYFILEPIEZOACTUATOR Boundary conditions for piezoelectric actuator e
%   [ q, g, h, r ] = BOUNDARYFILEPIEZOACTUATOR( p, e, V ) returns the
%   Neumann BC (q, g) and Dirichlet BC (h, r) matrices for the
%   piezoelectric actuator example example.
%   p is the point matrix returned from INITMESH
%   e is the edge matrix returned from INITMESH
%   V is the voltage applied to the top layer of the beam

% Copyright 2012 The MathWorks, Inc.
% $Revision: 1.1.6.1 $  $Date: 2012/05/08 20:26:10 $

N = 3;
ne = size(e,2);
q = zeros(N^2, ne);
g = zeros(N, ne);
h = zeros(N^2, 2*ne);
r = zeros(N, 2*ne);
voltage = V;
for i=1:ne
  ei = e(5,i);
  if(ei == 1)
    % top edge
    % set the voltage at both vertices on the edge
    h(9,i) = 1;
    h(9,i+ne) = 1;
    r(3,i) = voltage;
    r(3,i+ne) = voltage;
  elseif(ei == 2)
    % bottom edge
    % set the voltage to zero at both vertices on the edge
    % (the entries in r have already been set to zero above)
    h(9,i) = 1;
    h(9,i+ne) = 1;
  elseif(ei == 6 || ei == 7)
    % left edge
    % set the u and v displacements to zero at both vertices on the edge
    % (the entries in r have already been set to zero above)
    h(1,i) = 1;
    h(1,i+ne) = 1;
```

```
    h(5,i) = 1;
    h(5,i+ne) = 1;
  end
end

end
```

Finite Element Solution

```
a = 0;
f = [0;0;0];
u = assempde(b, p, e, t, c, a, f);
%
% For display and plotting purposes, it is convenient to reshape the
% solution vector as three columns containing the x-displacement,
% y-displacement, and electrical potential, respectively.
%
n = size(p,2); % number of finite element nodes
uu = reshape(u, n, []);
format shortE
feTipDeflection = uu(1,2);
fprintf('Finite element tip deflection is: %12.4e\n', feTipDeflection);
varsToPlot = char('X-Deflection, meters', 'Y-Deflection, meters', ...
  'Electrical Potential, Volts');
for i=1:size(varsToPlot,1)
  figure;
  pdeplot(p, e, t, 'xydata', uu(:,i), 'contour', 'on');
  title(varsToPlot(i,:));
  % scale the axes to make it easier to view the contours
  axis([0, L, -4*H2, 4*H2]);
  xlabel 'X-Coordinate, meters'
  ylabel 'Y-Coordinate, meters'
end

Finite element tip deflection is:  -8.2068e-06
```
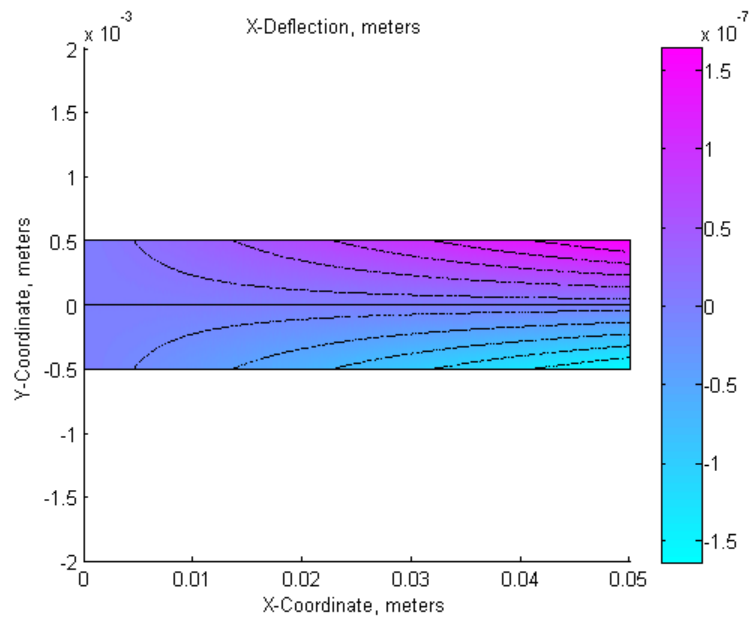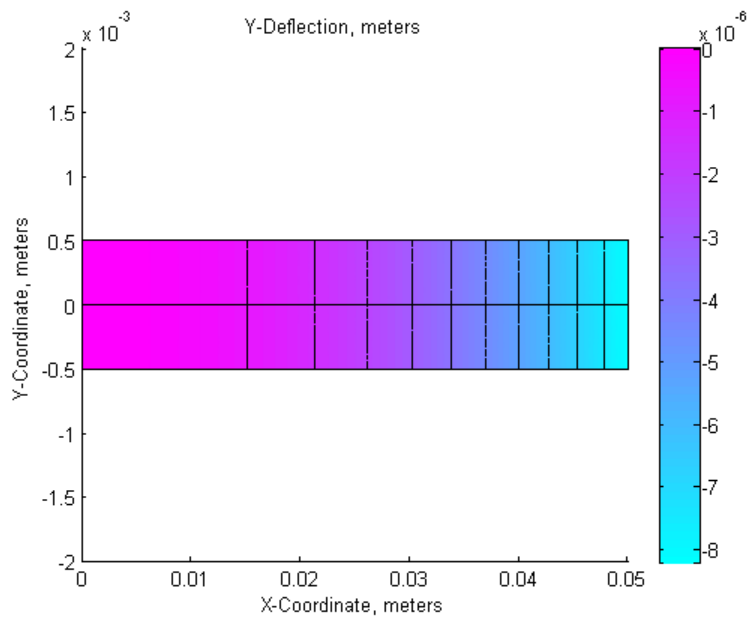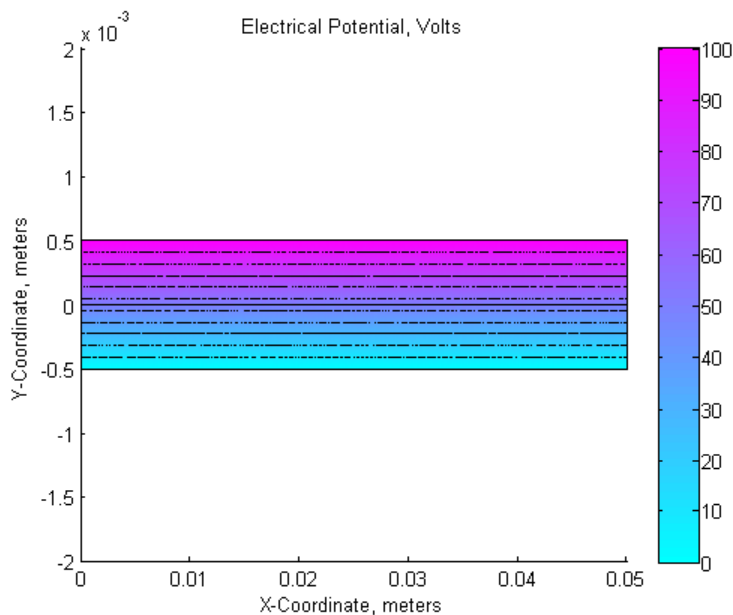
Analytical Solution

A simple, approximate, analytical solution was obtained for this problem in reference 1.

```
tipDeflection = -3*d31*V*L^2/(8*H2^2);
fprintf('Analytical tip deflection is: %12.4e\n', tipDeflection);

Analytical tip deflection is:  -8.2500e-06
```

Summary

The color contour plots of x-deflection and y-deflection show the standard behavior of the classical cantilever beam solution. The linear distribution of voltage through the thickness of the beam is as expected. There is good agreement between the PDE Toolbox finite element solution and the analytical solution from reference 1.

Although this example shows a very specific coupled elasticity-electrostatic model, the general approach here can be used for many other systems of coupled PDEs. The key to applying PDE Toolbox to these types of coupled systems is the systematic, multi-step coefficient mapping procedure described above.

References

1 Hwang, W. S.; Park, H. C; Finite Element Modeling of Piezoelectric Sensors and Actuators. AIAA Journal, 31(5), pp 930-937, 1993.

2 Pieford, V; Finite Element Modeling of Piezoelectric Active Structures. PhD Thesis, Universite Libre De Bruxelles, 2001.

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* $\mathbf{n} \cdot (\varepsilon \nabla V)$ 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.
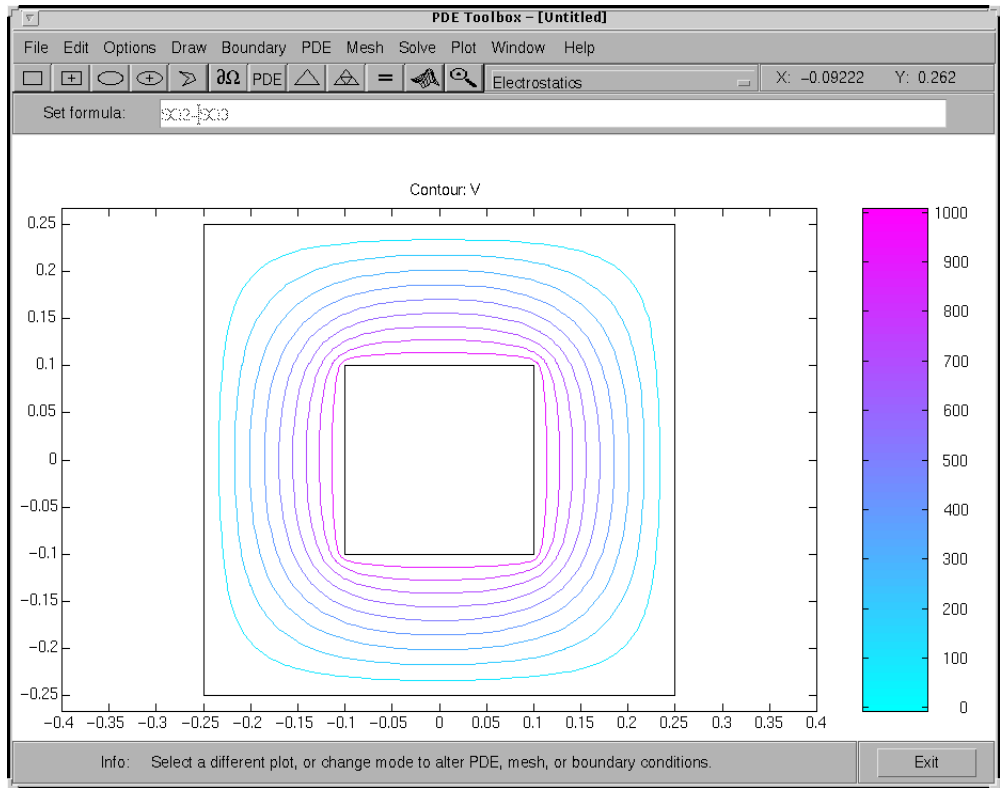
Equipotential Lines in Air-Filled Frame

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

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

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

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

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 **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

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

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

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

μ_{max} = 5000, μ_{min} = 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

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

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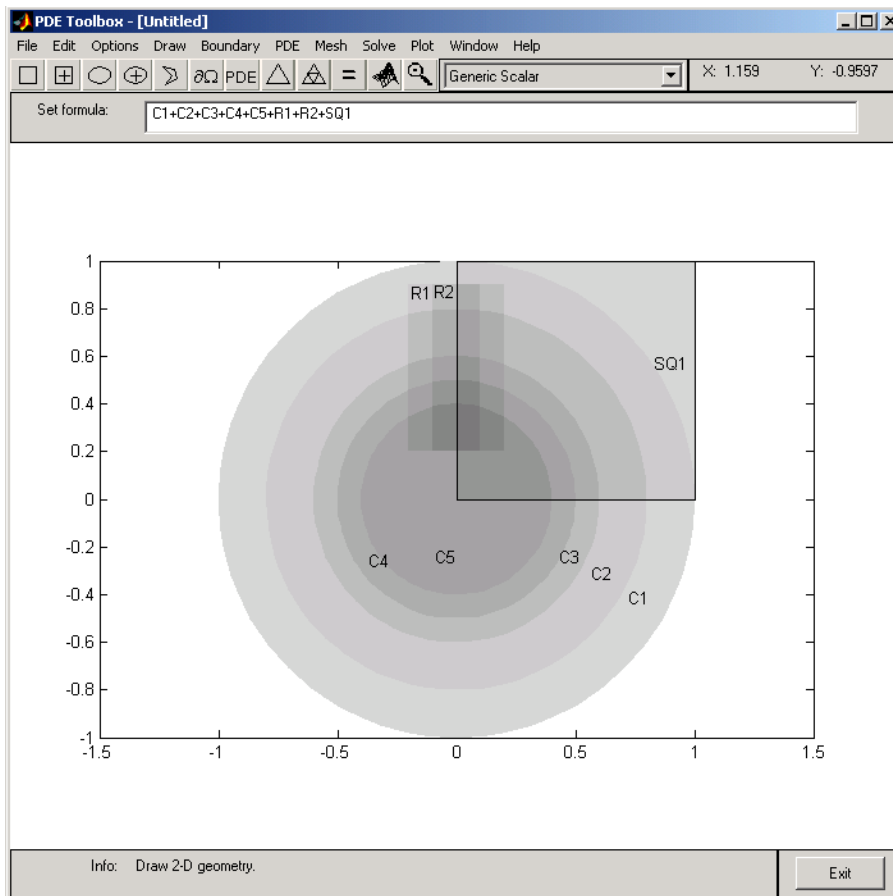
Enter the following set formula to reduce the model to the first quadrant:

```
(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.
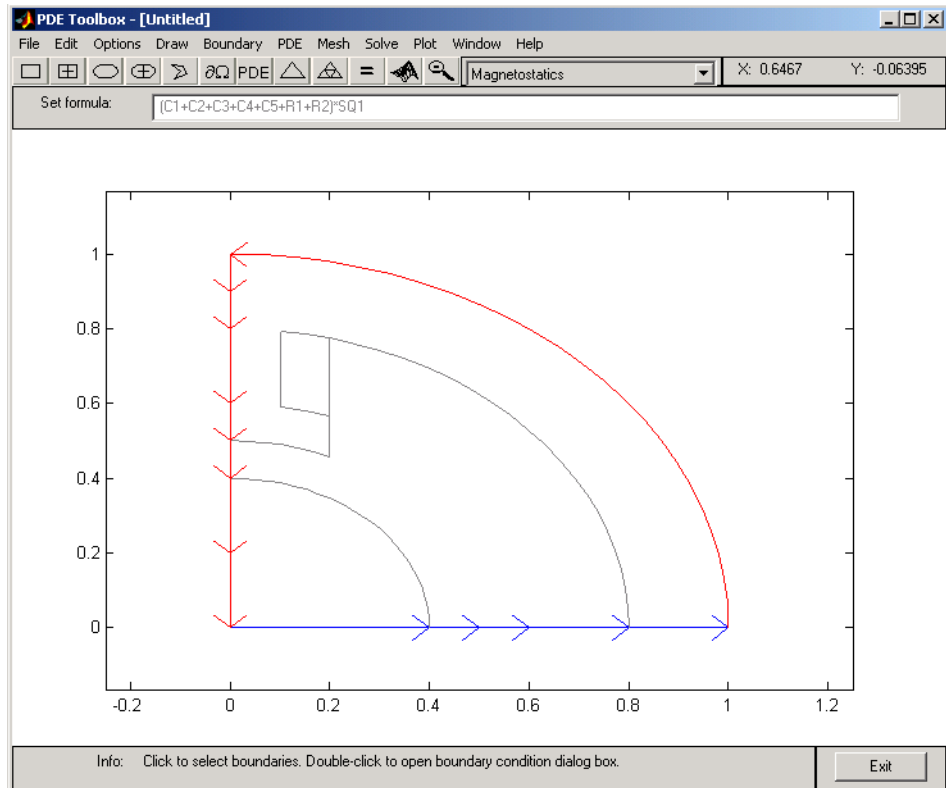
Before moving to the PDE mode, select the boundaries along the *x*-axis and set the boundary condition to a Neumann condition with $g = 0$ and $q = 0$. In the PDE mode, turn on the labels by selecting the **Show Subdomain Labels** option from the **PDE** menu. Double-click each subdomain to define the 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.
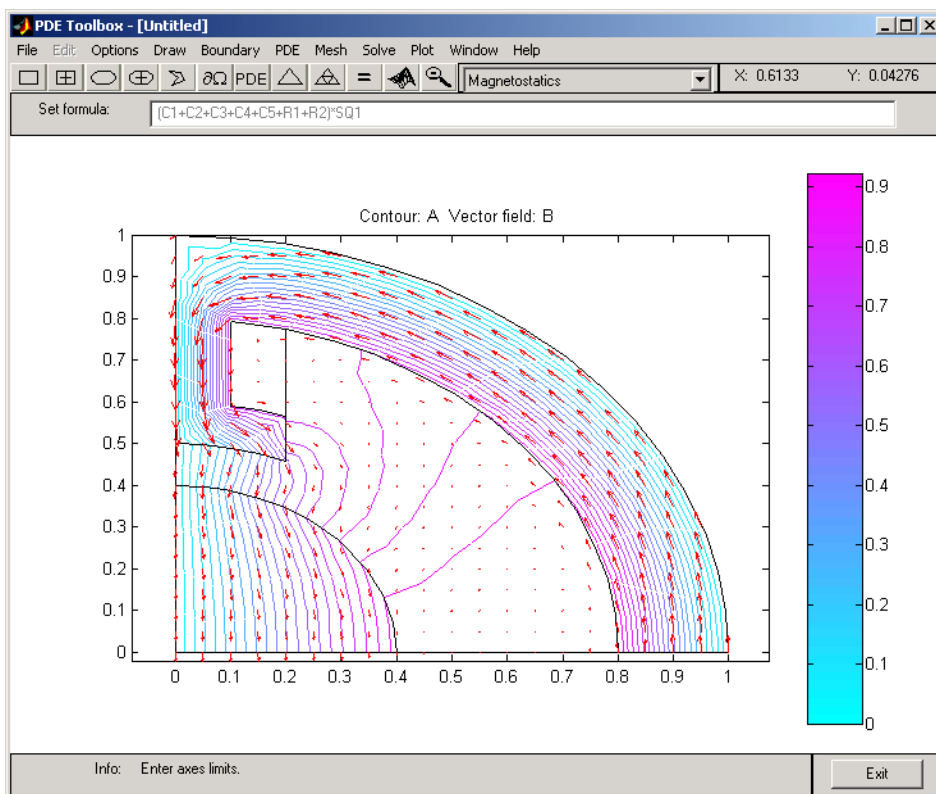
Equipotential Lines and Magnetic Flux in a Two-Pole Motor

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 **H** from the first set and **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 **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

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

$$\mathbf{H} = \left(\mathbf{H}_x, \mathbf{H}_y, 0\right) = \frac{-1}{j\mu\sigma}\nabla \times E_c.$$

The scalar equation for E_c becomes

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

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 $\varepsilon_c = \varepsilon\text{-}j\sigma/\omega$. 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 E_c. This is equivalent to specifying the tangential component of the magnetic field **H**:

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

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

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

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 shows 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 $\omega = 2\pi \cdot 50$.
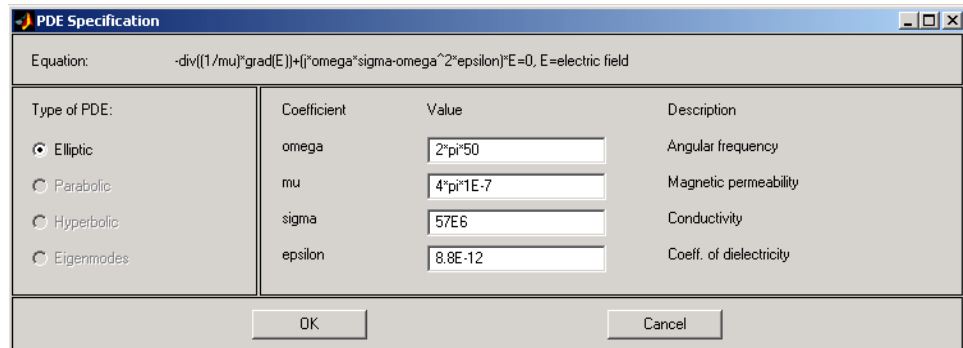
Initialize the mesh and solve the equation. Due to the skin effect, the current density at the surface of the conductor is much higher than in the conductor's interior. This is clearly visualized by plotting the current density J as a 3-D plot. To improve the accuracy of the solution close to the surface, 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.
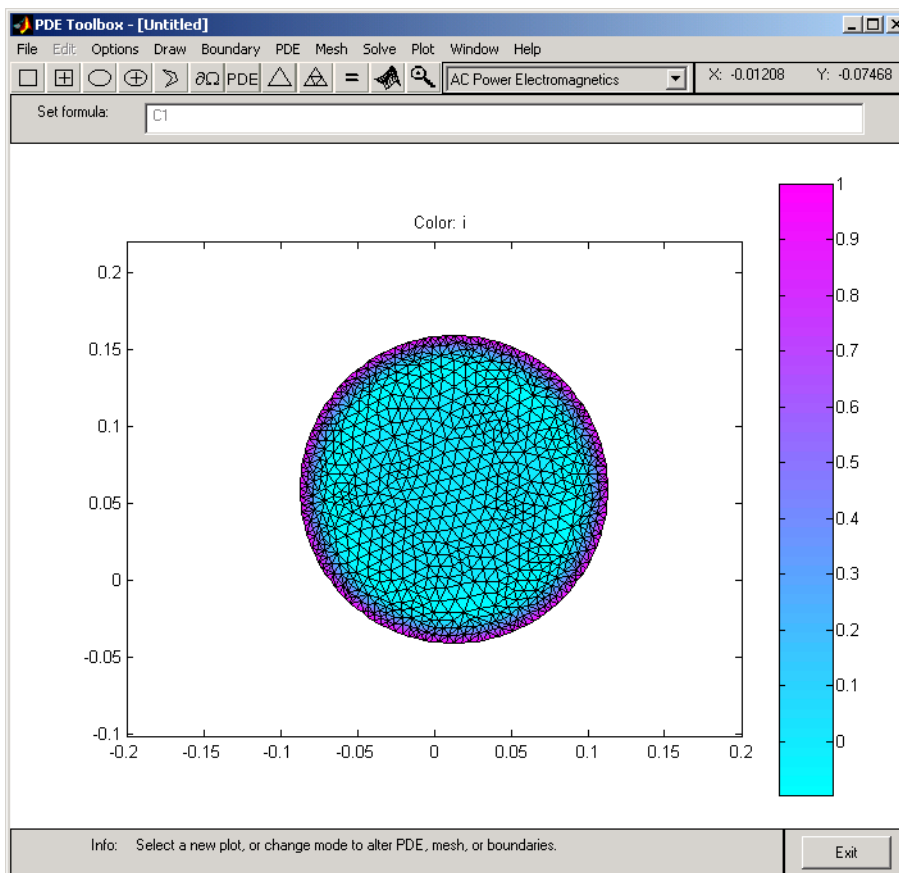
The Adaptively Refined Mesh

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).
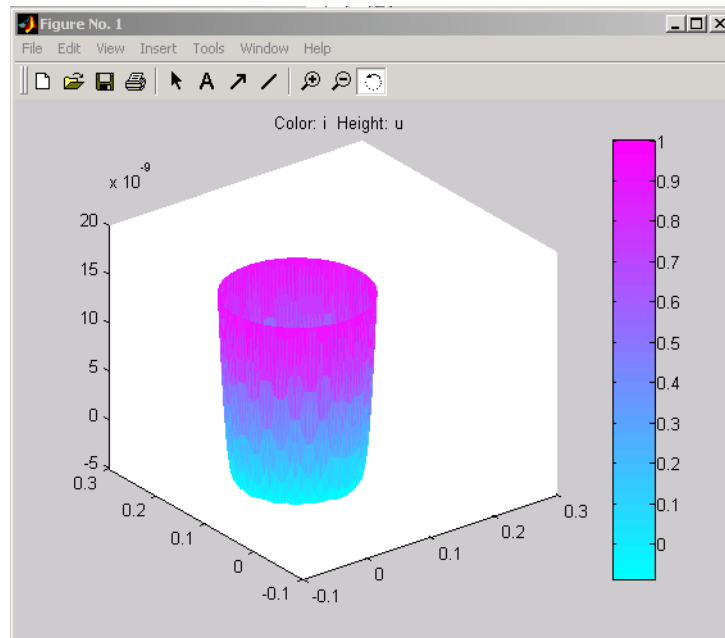
The Current Density in an AC Wire

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 $\mathbf{J} = \sigma\mathbf{E}$. 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 **E**, and the current density **J** are all available for plotting. Interesting quantities to visualize are the current lines (the vector field of **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 $\sigma = 1$ (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**.
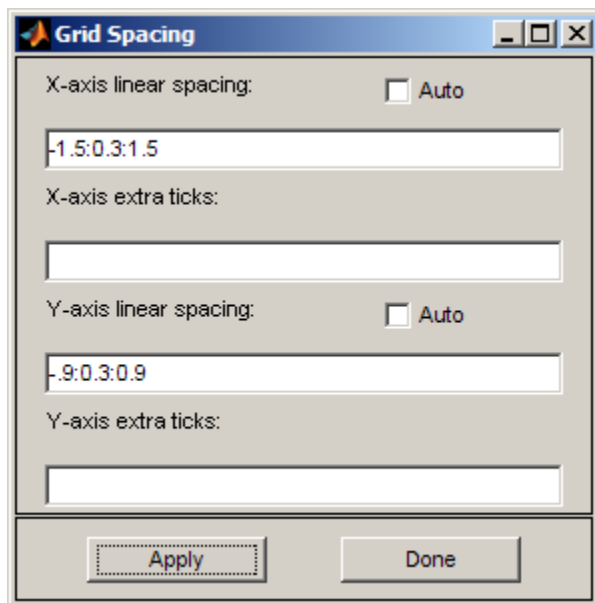
4 Click **Options > Snap**

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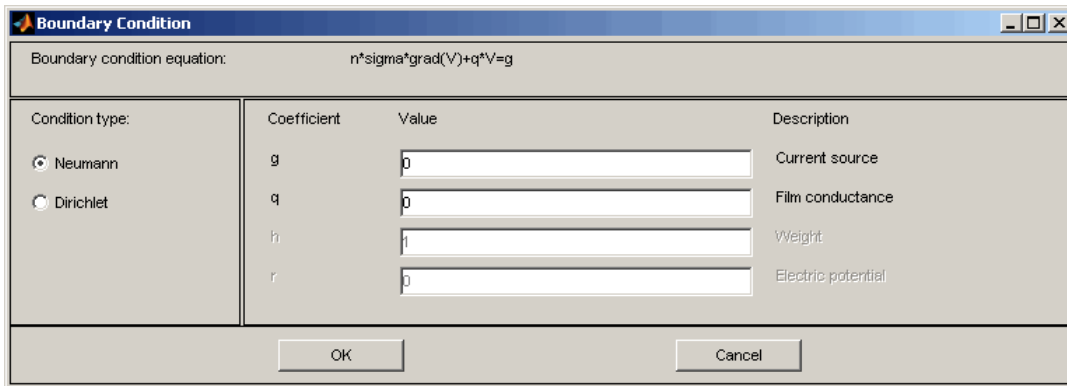
11 Hold down **Shift** and click to select the left circular conductor boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

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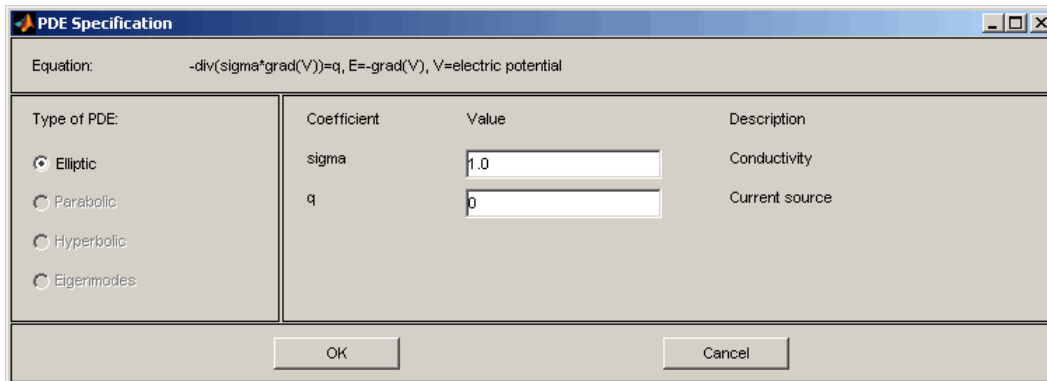
17 Initialize the mesh by clicking **Mesh > Initialize Mesh**.

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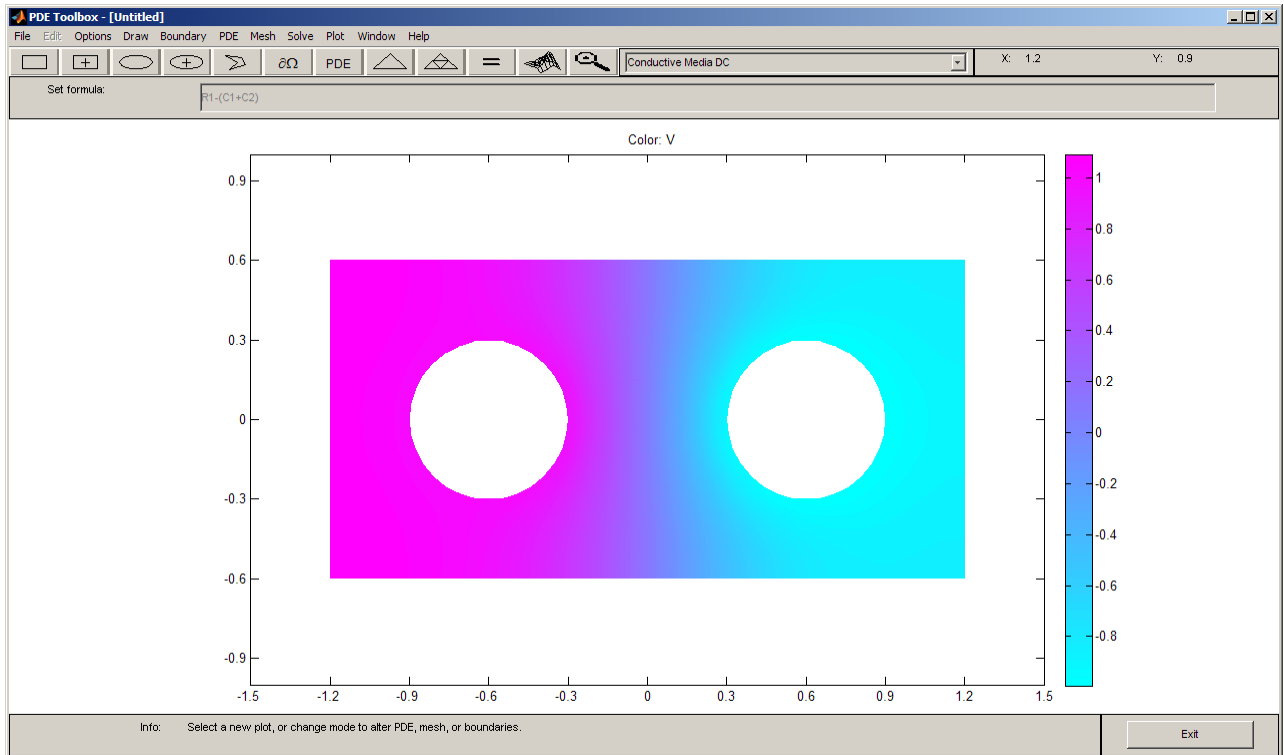
21 Visualize the current density **J** by clicking **Plot > Parameters**, selecting **Contour** and **Arrows** check box, and clicking **Plot**.

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:

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

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 $h(T_{\text{ext}} - T)$ 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,

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

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 $h(T_{\text{ext}} - T)$ 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.
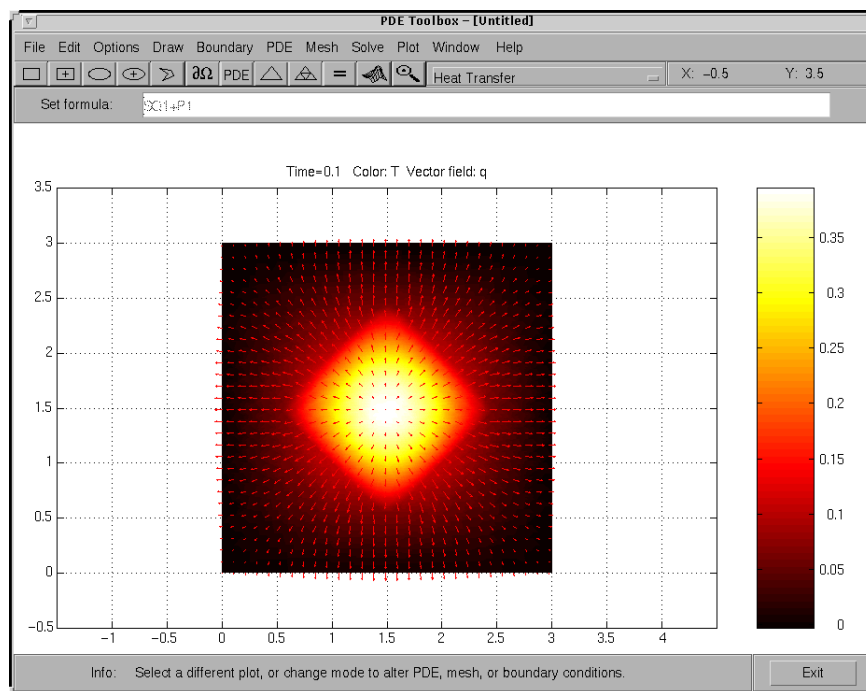
Visualization of the Temperature and the Heat Flux

Nonlinear Heat Transfer In a Thin Plate

This example shows how to perform a heat transfer analysis of a thin plate using the Partial Differential Equation Toolbox™.

The plate is square and the temperature is fixed along the bottom edge. No heat is transferred from the other three edges (i.e. they are insulated). Heat is transferred from both the top and bottom faces of the plate by convection and radiation. Because radiation is included, the problem is nonlinear. One of the purposes of this example is to show how to handle nonlinearities in PDE problems.

Both a steady state and a transient analysis are performed. In a steady state analysis we are interested in the final temperature at different points in the plate after it has reached an equilibrium state. In a transient analysis we are interested in the temperature in the plate as a function of time. One question that can be answered by this transient analysis is how long does it take for the plate to reach an equilibrium temperature.

Heat Transfer Equations for the Plate

The plate has planar dimensions one meter by one meter and is 1 cm thick. Because the plate is relatively thin compared with the planar dimensions, the temperature can be assumed constant in the thickness direction; the resulting problem is 2D.

Convection and radiation heat transfer are assumed to take place between the two faces of the plate and a specified ambient temperature.

The amount of heat transferred from each plate face per unit area due to convection is defined as

$$Q_c = h_c(T - T_a)$$

where T_a is the ambient temperature, T is the temperature at a particular x and y location on the plate surface, and h_c is a specified convection coefficient.

The amount of heat transferred from each plate face per unit area due to radiation is defined as

$$Q_r = \epsilon\sigma(T^4 - T_a^4)$$

where ϵ is the emissivity of the face and σ is the Stefan-Boltzmann constant. Because the heat transferred due to radiation is proportional to the fourth power of the surface temperature, the problem is nonlinear.

The PDE describing the temperature in this thin plate is

$$\rho C_p t_z \frac{\partial T}{\partial t} - kt_z\nabla^2 T + 2Q_c + 2Q_r = 0$$

where ρ is the material density, C_p is the specific heat, t_z is the plate thickness, and the factors of two account for the heat transfer from both plate faces.

It is convenient to rewrite this equation in the form expected by PDE Toolbox

$$\rho C_p t_z \frac{\partial T}{\partial t} - kt_z\nabla^2 T + 2h_c T + 2\epsilon\sigma T^4 = 2h_c T_a + 2\epsilon\sigma T_a^4$$

Problem Parameters

The plate is composed of copper which has the following properties

```
k = 400; % thermal conductivity of copper, W/(m-K)
rho = 8960; % density of copper, kg/m^3
specificHeat = 386; % specific heat of copper, J/(kg-K)
thick = .01; % plate thickness in meters
stefanBoltz = 5.670373e-8; % Stefan-Boltzmann constant, W/(m^2-K^4)
hCoeff = 1; % Convection coefficient, W/(m^2-K)
% The ambient temperature is assumed to be 300 degrees-Kelvin.
ta = 300;
emiss = .5; % emissivity of the plate surface
```

Definition of PDE Coefficients

The expressions for the coefficients required by PDE Toolbox can easily be identified by comparing the equation above with the scalar parabolic equation in the PDE Toolbox documentation.

```
c = thick*k;
```

```
% Because of the radiation boundary condition, the "a" coefficient
% is a function of the temperature, u. It is defined as a MATLAB
% expression so it can be evaluated for different values of u
% during the analysis.
a = sprintf('2*%g + 2*%g*%g*u.^3', hCoeff, emiss, stefanBoltz)
f = 2*hCoeff*ta + 2*emiss*stefanBoltz*ta^4;
d = thick*rho*specificHeat;


a =

2*1 + 2*0.5*5.67037e-08*u.^3
```

Geometry and Mesh

For a square, the geometry and mesh are easily defined as shown below.

```
width = 1; height = 1;
% define  the square by giving the 4 x-locations followed by the 4
% y-locations of the corners.
gdmTrans = [3 4 0 width width 0 0 0 height height];
g = decsg(gdmTrans', 'S1', ('S1')');
% Create the triangular mesh on the square with approximately
% ten elements in each direction.
hmax = .1; % element size
[p, e, t] = initmesh(g, 'Hmax', hmax);
pdeplot(p,e,t);
title 'Plate With Triangular Element Mesh'
xlabel 'X-coordinate, meters'
ylabel 'Y-coordinate, meters'
```

Plate With Triangular Element Mesh
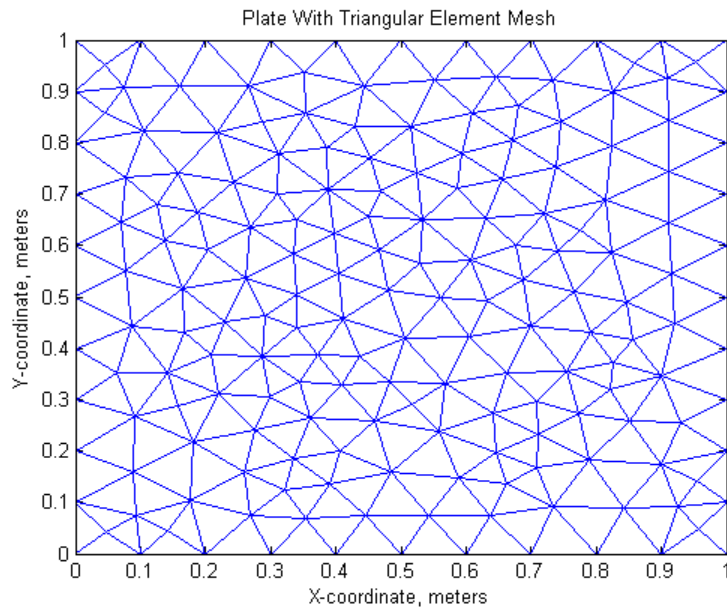
Boundary Conditions

The bottom edge of the plate is set to 1000 degrees-Kelvin.

The boundary conditions are defined in the function below which is referred to in the PDE Toolbox documentation as a boundary file. Three of the plate edges are insulated. Because a Neumann boundary condition equal zero is the default in the finite element formulation, the boundary conditions on these edges do not need to be set explicitly. A Dirichlet condition is set on all nodes on the bottom edge, edge 1,

```
b=@boundaryFileThinPlate;
```

```
type boundaryFileThinPlate
```

```
function [ q, g, h, r ] = boundaryFileThinPlate( p, e, u, time )
%BOUNDARYFILETHINPLATE Boundary conditions for heatTransferThinPlateExample
%   [ q, g, h, r ] = BOUNDARYFILETHINPLATE( p, e, u, time ) returns the
```

```
%    Neumann BC (q, g) and Dirichlet BC (h, r) matrices for the
%    heatTransferThinPlateExample example.
%    p is the point matrix returned from INITMESH
%    e is the edge matrix returned from INITMESH
%    u is the solution vector (used only for nonlinear cases)
%    time (used only for parabolic and hyperbolic cases)
%
%    See also PDEBOUND, INITMESH

%    Copyright 2012 The MathWorks, Inc.
%    $Revision: 1.1.6.1 $  $Date: 2012/05/04 00:02:05 $

N = 1;
ne = size(e,2);
q = zeros(N^2, ne);
g = zeros(N, ne);
h = zeros(N^2, 2*ne);
r = zeros(N, 2*ne);
for i=1:ne
  ei = e(5,i);
  if(ei == 1)
    % Set the temperature at both vertices on the edge
    h(1,i) = 1;
    h(1,i+ne) = 1;
    r(1,i) = 1000;
    r(1,i+ne) = 1000;
  end
end

end
```

Steady State Solution

Because the a and f coefficients are functions of temperature (due to the radiation boundary conditions), the nonlinear solver pdenonlin must be used to obtain the solution.

```
u = pdenonlin(b,p,e,t,c,a,f, 'jacobian', 'lumped');
```

```
figure;
pdeplot(p, e, t, 'xydata', u, 'contour', 'on', 'colormap', 'jet');
title 'Temperature In The Plate, Steady State Solution'
xlabel 'X-coordinate, meters'
ylabel 'Y-coordinate, meters'
plotAlongY(p, u, O);
title 'Temperature As a Function of the Y-Coordinate'
xlabel 'X-coordinate, meters'
ylabel 'Temperature, degrees-Kelvin'
fprintf('Temperature at the top edge of the plate = %5.1f degrees-K\n', .
  u(4));
```

Temperature at the top edge of the plate = 448.9 degrees-K

Temperature As a Function of the Y-Coordinate
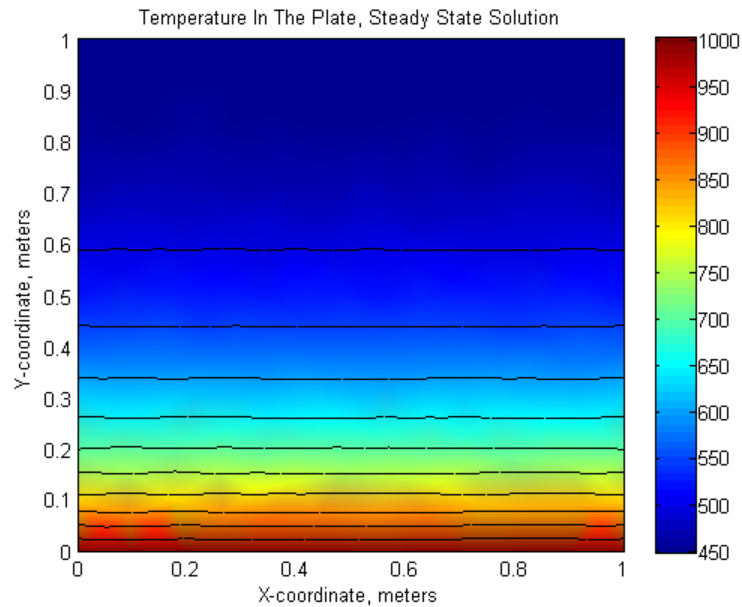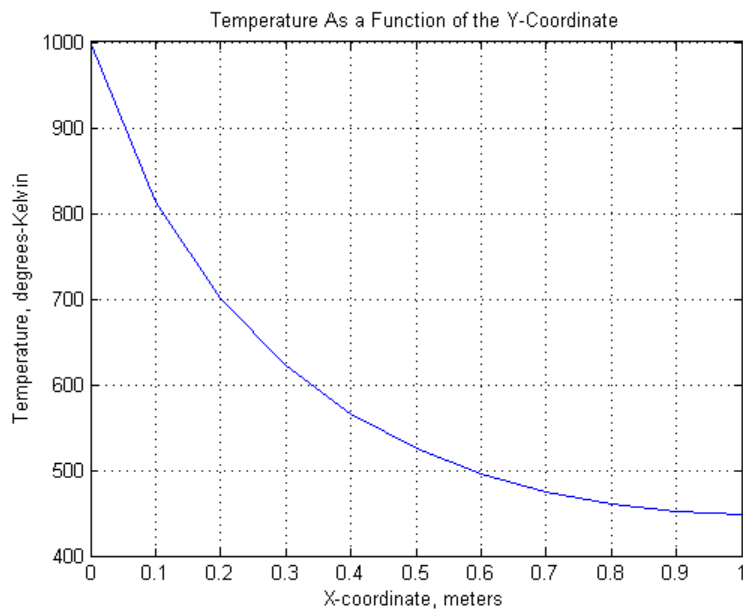
Transient Solution

```
endTime = 5000;
tlist = 0:50:endTime;
numNodes = size(p,2);
% Set the initial temperature of all nodes to ambient, 300 K
u0(1:numNodes) = 300;
% Find all nodes along the bottom edge and set their initial temperature
% to the value of the constant BC, 1000 K
nodesY0 = abs(p(2,:)) < 1.0e-5;
u0(nodesY0) = 1000;
rtol = 1.0e-3; atol = 1.0e-4;
% The transient solver parabolic automatically handles both linear
% and nonlinear problems, such as this one.
u = parabolic(u0, tlist, b,p,e,t,c,a,f,d,rtol,atol);
figure;
plot(tlist, u(3, :)); grid;
title 'Temperature Along the Top Edge of the Plate as a Function of Time'
xlabel 'Time, seconds'
```

```
  ylabel 'Temperature, degrees-Kelvin'
  %
  figure;
  pdeplot(p, e, t, 'xydata', u(:,end), 'contour', 'on', 'colormap', 'jet');
  title(sprintf('Temperature In The Plate, Transient Solution( %d seconds)\
    tlist(1,end)));
  xlabel 'X-coordinate, meters'
  ylabel 'Y-coordinate, meters'
  %
  fprintf('\nTemperature at the top edge of the plate(t=%5.1f secs) = %5.1f
    tlist(1,end), u(4,end));
```

```
65 successful steps
0 failed attempts
95 function evaluations
1 partial derivatives
16 LU decompositions
94 solutions of linear systems

Temperature at the top edge of the plate(t=5000.0 secs) = 447.2 degrees-K
```

Temperature Along the Top Edge of the Plate as a Function of Time

Temperature In The Plate, Transient Solution(5000 seconds)
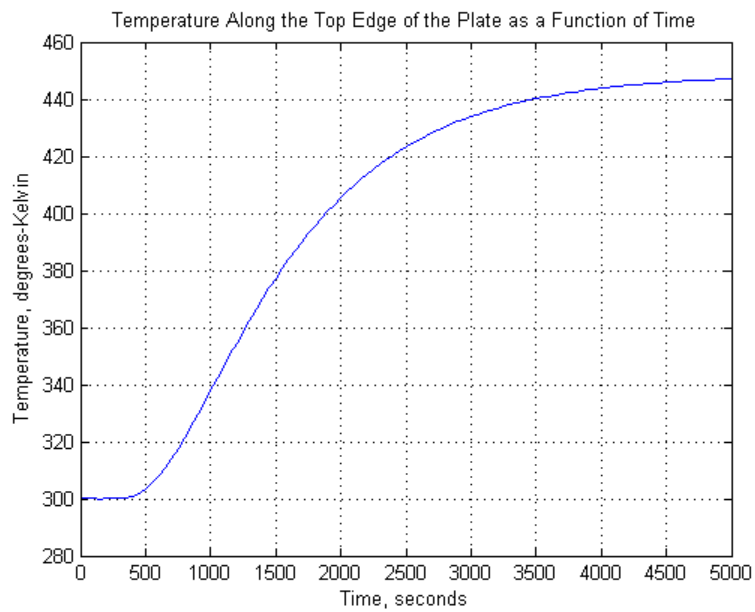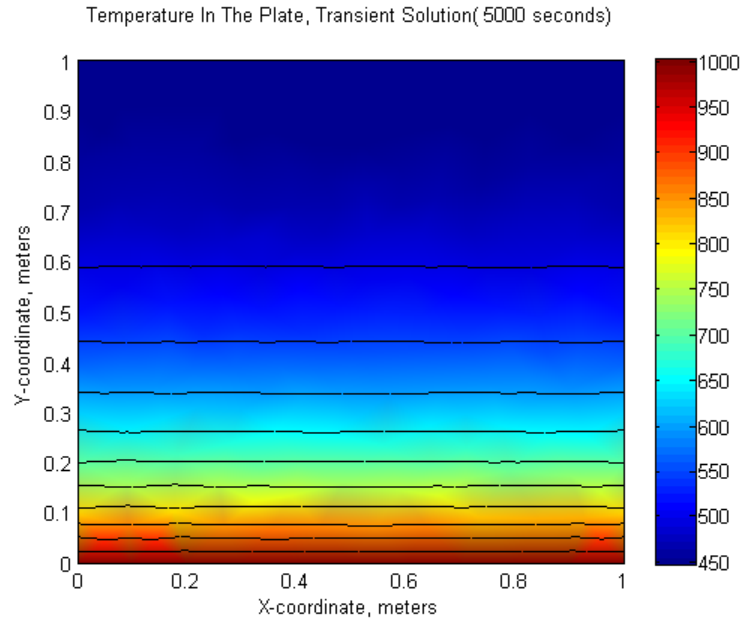
Summary

As can be seen, the plots of temperature in the plate from the steady state and transient solution at the ending time are very close. That is, after around 5000 seconds, the transient solution has reached the steady state values. The temperatures from the two solutions at the top edge of the plate agree to within one percent.

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 4-19.

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.

Elliptic PDEs

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:

In this section...
"Solve Poisson's Equation on a Unit Disk" on page 3-71
"Scattering Problem" on page 3-75
"Minimal Surface Problem" on page 3-80
"Domain Decomposition Problem" on page 3-82

Solve Poisson's Equation on a Unit Disk

This example shows how to solve a simple elliptic PDE in the form of Poisson's equation on a unit disk.

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

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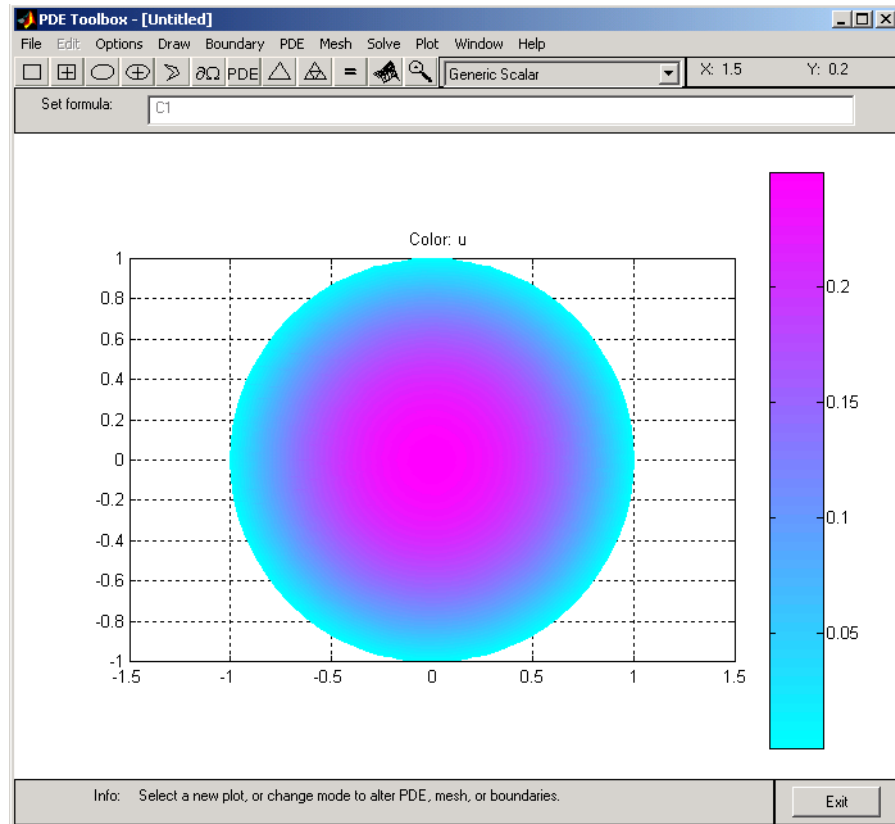
7 To compare the numerical solution to the exact solution, select the user entry in the **Property** pop-up menu for **Color** in the Plot Selection dialog box. Then input the MATLAB expression u-(1-x.^2-y.^2)/4 in the **user entry** edit field. You obtain a plot of the absolute error in the solution.

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.
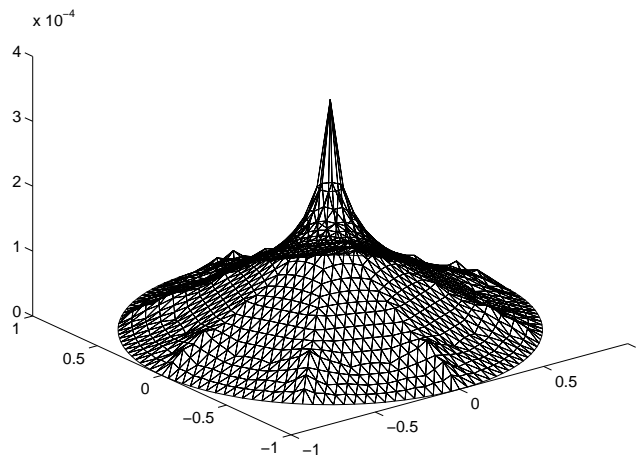
The Error Function

`pdedemo1` performs all the previous steps.

Scattering Problem

This example shows how to solve a simple scattering problem, where you compute the waves reflected from an object illuminated by incident waves. For this problem, assume an infinite horizontal membrane subjected to small vertical displacements *U*. The membrane is fixed at the object boundary.
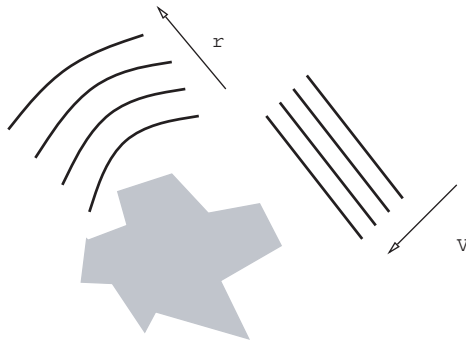
We assume that the medium is homogeneous so that the wave speed is constant, *c*.

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:

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

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 = -k² = -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

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

which is the reflex of

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

To see the whole field, plot

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

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 implementation of the scattering problem.
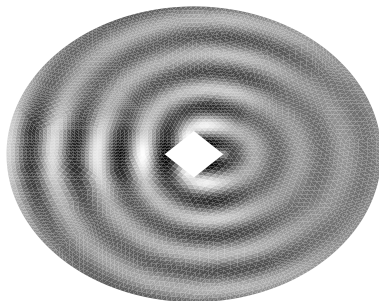
Minimal Surface Problem

This example shows how to solve a nonlinear problem for this equation:

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

where the coefficients c, a, and f do not depend only on x and y, but also on the solution u.

The problem geometry is a unit disk, specified as $\Omega = \{(x, y) \mid x^2 + y^2 \leq 1\}$, with $u = x^2$ on $\partial\Omega$.

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

This example show how to solve this minimal surface problem using both 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`.
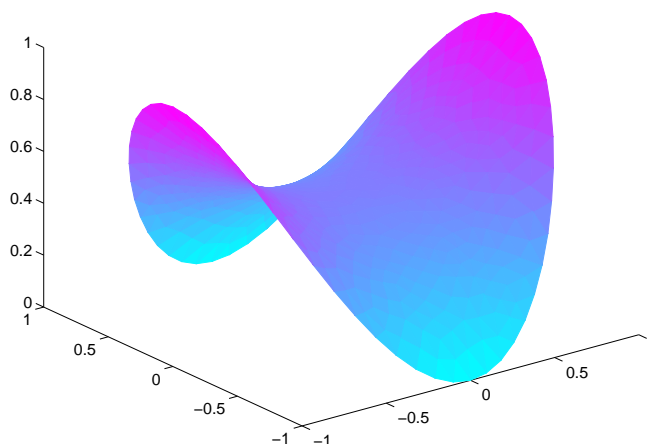
Domain Decomposition Problem

This example shows how to perform one-level domain decomposition for complicated geometries, where you can decompose this geometry 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 Ω_1, Ω_2, . . . , 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:

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

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:

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

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.
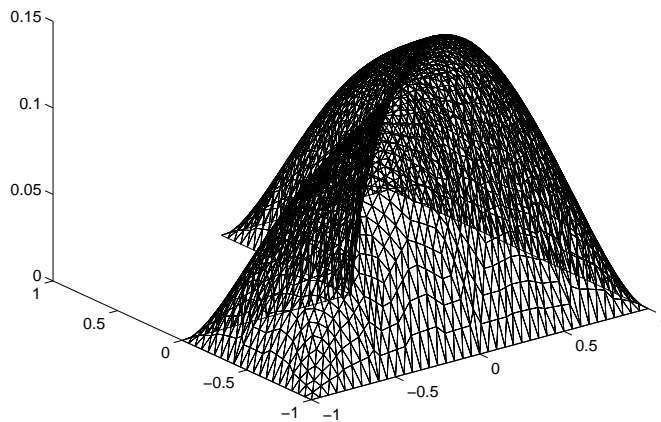
Parabolic PDEs

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:

In this section...
"Heat Equation for Metal Block with Cavity" on page 3-87
"Heat Distribution in a Radioactive Rod" on page 3-91

Heat Equation for Metal Block with Cavity

This example shows how to solve a heat equation that describes the diffusion of heat in a body. The heat equation has the form:

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

Consider a metal block containing 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)
- $\partial u/\partial n$ = –10 on the right side (Neumann condition)
- $\partial u/\partial n$ = 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.
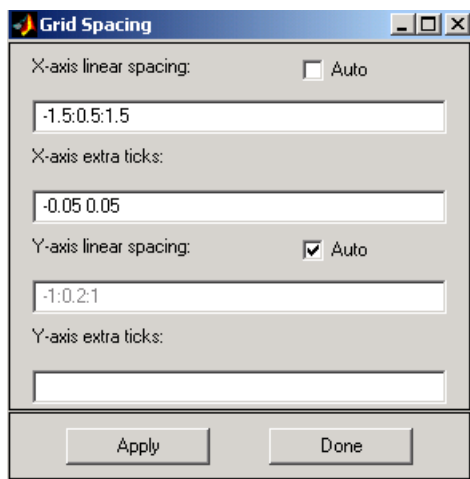
The CSG model of the metal block is now simply expressed as the set formula `R1-R2`.

Leave the draw mode and enter the boundary mode by clicking the ∂Ω 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 $u_0 = u(t_0)$ 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.
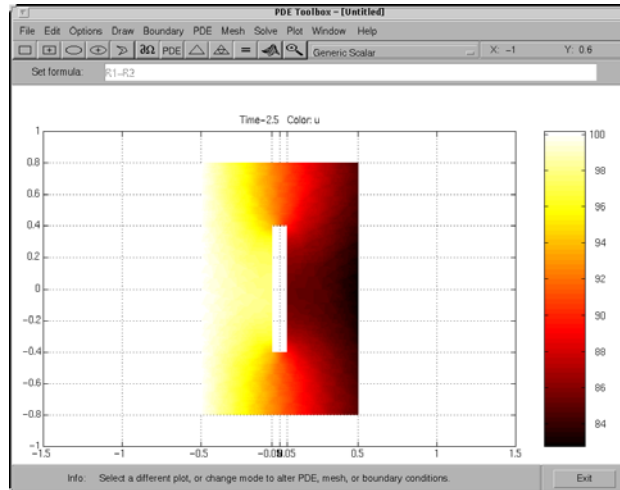
Using Command-Line Functions

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 a Radioactive Rod

This example shows how to solve a 3-D parabolic PDE problem by reducing the problem to 2–D using coordinate transformation.

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/kg^{o}C, and the thermal conductivity is 40 W/m^{o}C. The heat source is 20000 W/m^3. The temperature at the right end is 100 ^{o}C. The surrounding temperature at the outer boundary is 100 ^{o}C, and the heat transfer coefficient is 50 W/m^{2o}C. 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 $\vec{n}\cdot(k\nabla u) + qu = g$, 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).

- $\vec{n}\cdot(k\nabla u) = 50(100\text{-}u)$ 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 $\vec{n}\cdot(c\nabla u)$ 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 PDEs

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.

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 ($\partial u/\partial n = 0$) at the upper and lower sides. Additionally, we need initial values for $u(t_0)$ and $\partial u(t_0)/\partial t$

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

$$u(x,y,0) = \arctan\left(\cos\left(\frac{\pi}{2} x \right) \right)$$

and

$$\left. \frac{\partial u(x,y,t)}{\partial t} \right|_{t=0} = 3 \sin(\pi x) \exp\left(\sin\left(\frac{\pi}{2} y \right) \right)$$

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 $\partial u/\partial t$, enter

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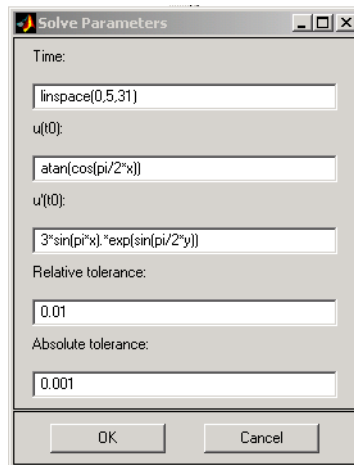

Finally, click the **=** button to compute the solution. The best plot for viewing the waves moving in the x and y directions is an animation of the whole

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 example 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.
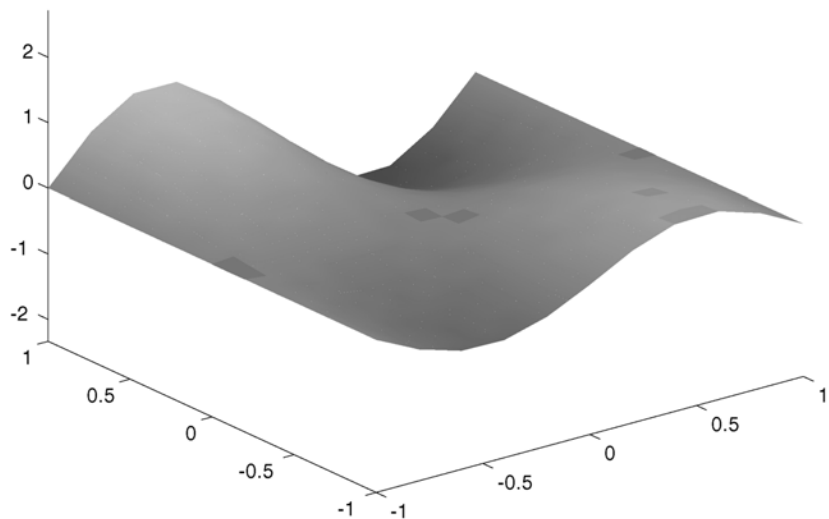
Animation of the Solution to the Wave Equation

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:

In this section...

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 MathWorks® logo. 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 a 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.
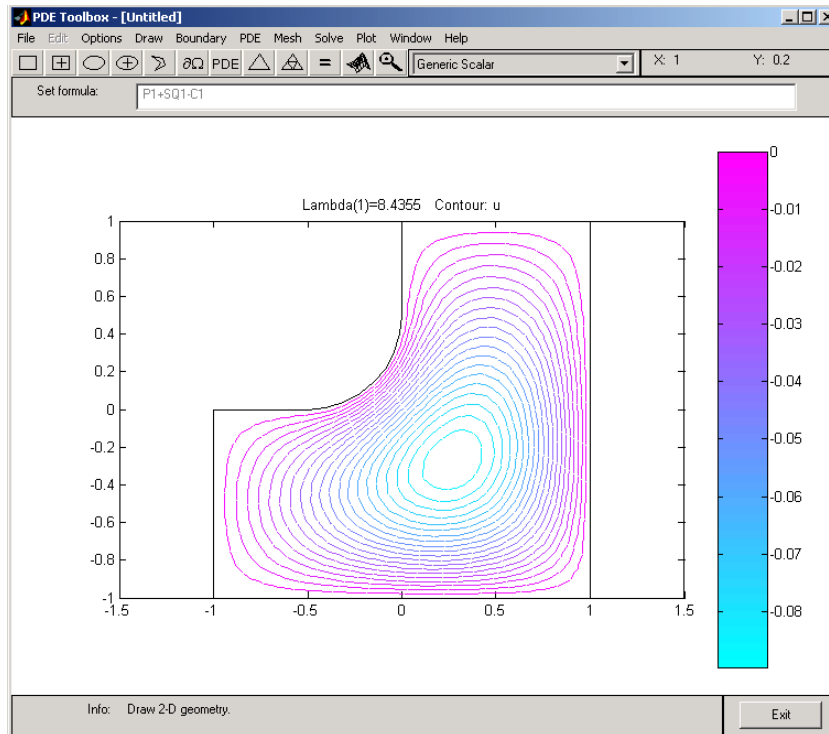
First Eigenmode for an L-Shaped Membrane with a Rounded Corner

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 ∂Ω 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 $g = -3/4$.

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.

Solve PDEs Programmatically

In this section...

When You Need Programmatic Solutions

Although the PDE Toolbox 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 in Partial Differential Equation Toolbox

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.
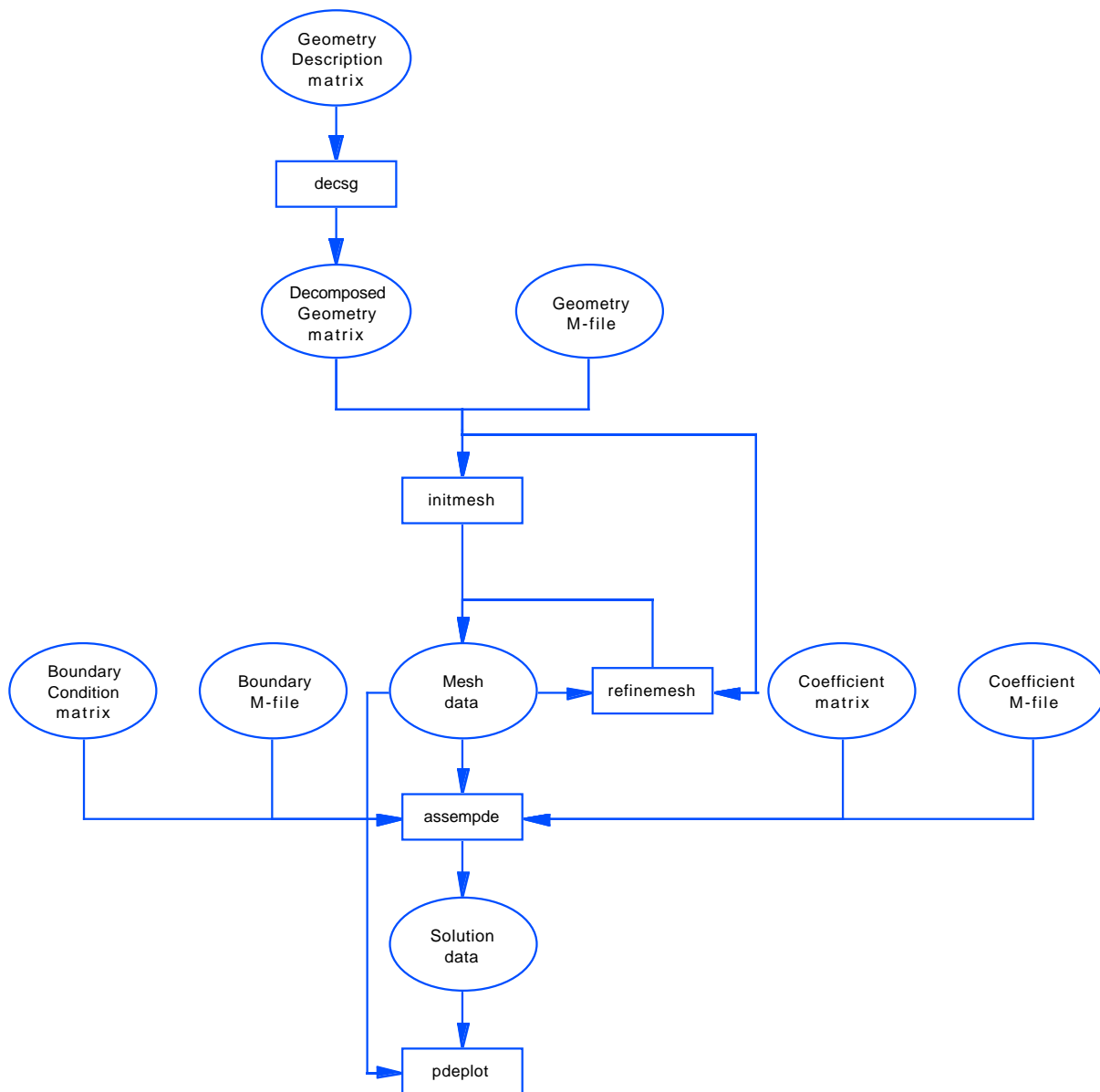
Constructive Solid Geometry Model

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. 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`.

Tips for Solving PDEs Programmatically

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.

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.

Solve Poisson's Equation on a Grid

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 $Sij = \sin(\pi ij/N_1)$. 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 $Ky = F$, 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.

4

Graphical User Interface

You open the graphical user interface (GUI) by entering `pdetool` at the command line. The main components of the GUI are the menus, the dialog boxes, and the toolbar.

PDE Toolbox GUI Menus

PDE Toolbox GUI menus let you perform the following operations:

- **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.

File Menu
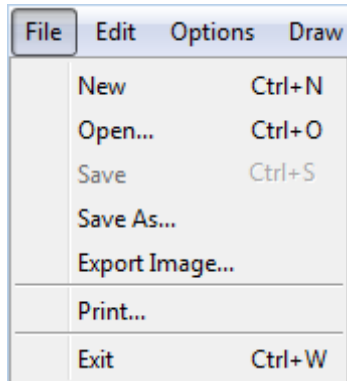
"New" on page 4-4	Create a new (empty) *Constructive Solid Geometry* (CSG) model.
"Open" on page 4-5	Load a model file from disk.
Save	Save the GUI session to a model file.
"Save As" on page 4-5	Save the GUI session to a new model file.
"Export Image" on page 4-5	Save the current figure in one of a variety of image formats.
"Print" on page 4-6	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.

Export Image

Save the current figure as a file in your choice of formats. Available formats include:

- Bitmap (`.bmp`)
- EPS (`.eps`)
- JPEG (`.jpg`)
- Portable Document Format (`.pdf`)
- Portable Network Graphics (`.png`)

• TIFF (`.tif`)

Print
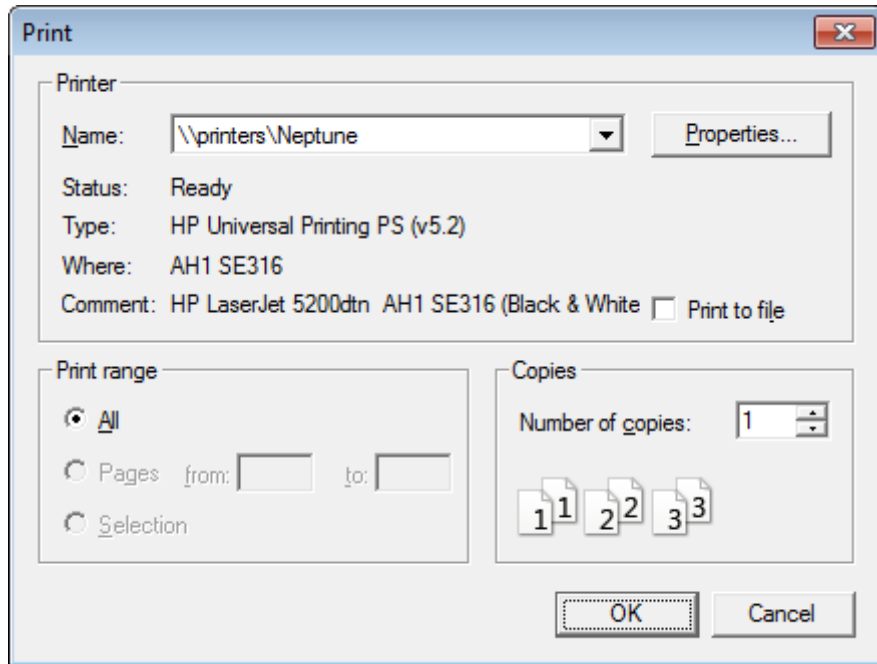
Print displays a dialog box for printing a hardcopy of a figure. Only the main part of the figure is printed, not the upper and lower menu and information parts. In the dialog box, you can enter any device option that is available for the MATLAB `print` command. The default device option is `-dps` (PostScript® 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" on page 4-8	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
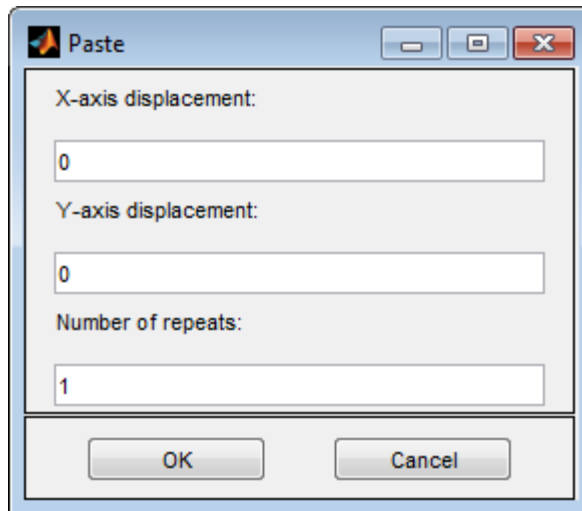
Paste displays a dialog box for pasting the contents of the Clipboard on to the current CSG model. The Clipboard contents can be repeatedly pasted adding a specified *x*- and *y*-axis displacement to the positions of the Clipboard objects.

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

Options Menu
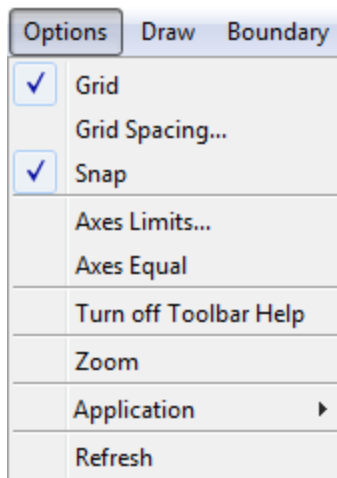
Grid	Turn grid on/off.
"Grid Spacing" on page 4-10	Adjust the grid spacing.
Snap	Turn the "snap-to-grid" feature on/off.
"Axes Limits" on page 4-11	Change the scaling of the drawing axes.
Axis Equal	Turn the "axis equal" feature on/off.
Turn off Toolbar Help	Turn off help texts for the toolbar buttons.
Zoom	Turn zoom feature on/off.
Application	Select application mode.
Refresh	Redisplay all graphical objects in the pdetool graphical user interface.

Grid Spacing
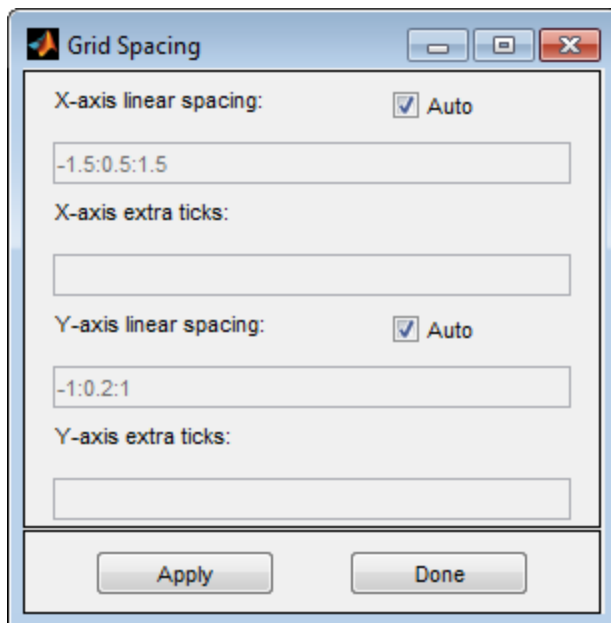
In the Grid Spacing dialog box, you can adjust the *x*-axis and *y*-axis grid spacing. By default, the MATLAB automatic linear grid spacing is used. If you turn off the **Auto** check box, the edit fields for linear spacing and extra ticks are enabled. For example, the default linear spacing `-1.5:0.5:1.5` 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
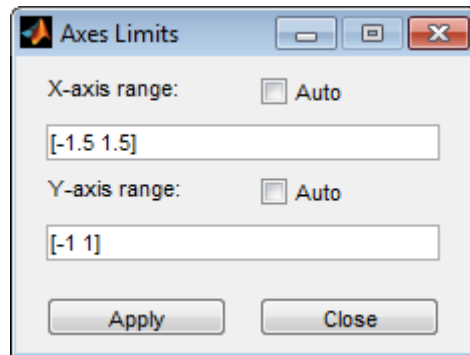
In the Axes Limits dialog box, the range of the *x*-axis and the *y*-axis can be adjusted. The axis range should be entered as a 1-by-2 MATLAB vector such as [-10 10]. If you select the **Auto** check box, automatic scaling of the axis is used.

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

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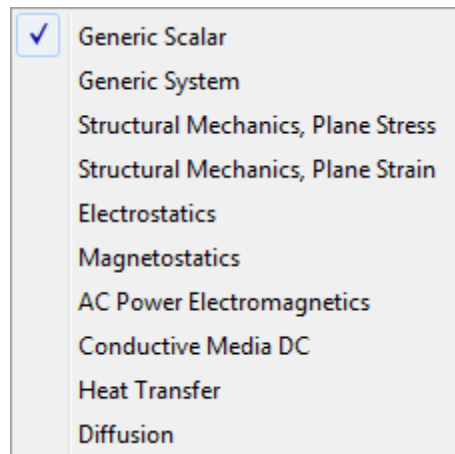
From the **Application** submenu, you can select from 10 available application modes. The application modes can also be selected using the pop-up menu in the upper right corner of the GUI.

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

Draw Menu
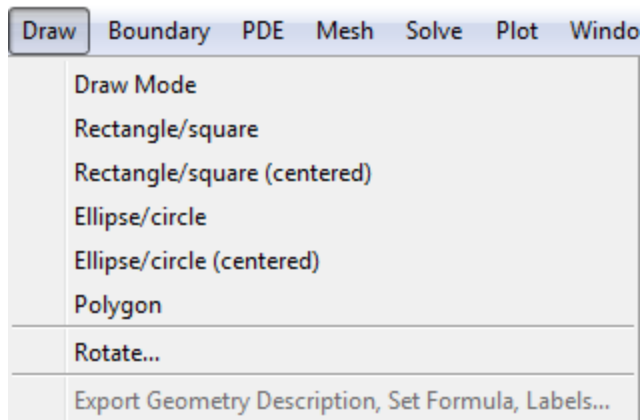
Draw Mode	Enter draw mode.
Rectangle/square	Draw a rectangle/square starting at a corner. Using the left mouse button, click-and-drag to create a rectangle. Using the right mouse button (or **Ctrl**+click), click-and-drag to create a square.
Rectangle/square (centered)	Draw a rectangle/square starting 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" on page 4-14	Rotate selected objects.
Export Geometry Description, Set Formula, Labels	Export the Geometry Description matrix gd, the set formula string sf, and the Name Space matrix ns (labels) to the main workspace.

Rotate
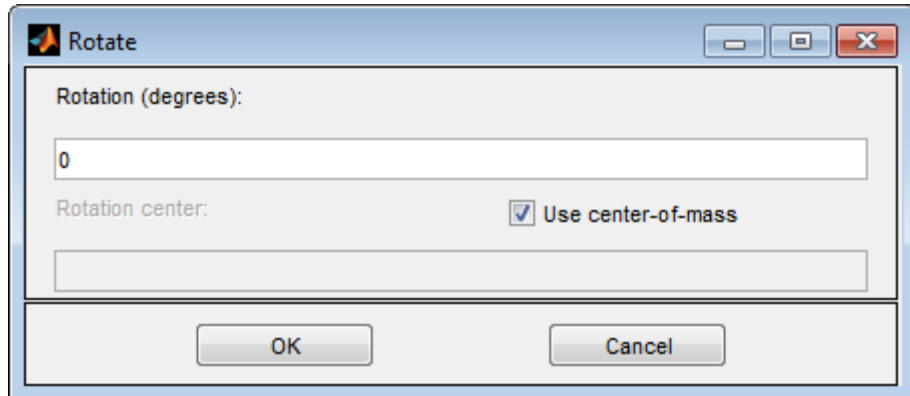
Rotate opens a dialog box where you can enter the angle of rotation in degrees. The selected objects are then rotated by the number of degrees that you specify. The rotation is done counter clockwise for positive rotation angles. By default, the rotation center is the center-of-mass of the selected objects. 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
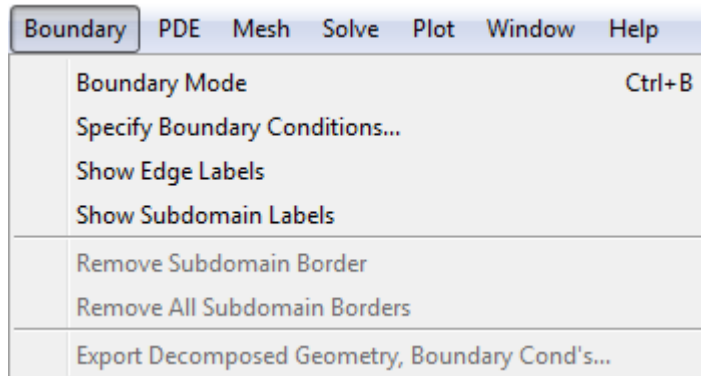
Boundary Mode	Enter the boundary mode.
Specify Boundary Conditions	Specify boundary conditions for the selected boundaries. If no boundaries are selected, the entered boundary condition applies to all boundaries.
Show Edge Labels	Toggle the labeling of the edges (outer boundaries and subdomain borders) on/off. The edges are labeled using the column number in the Decomposed Geometry matrix.
Show Subdomain 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 in pdetool
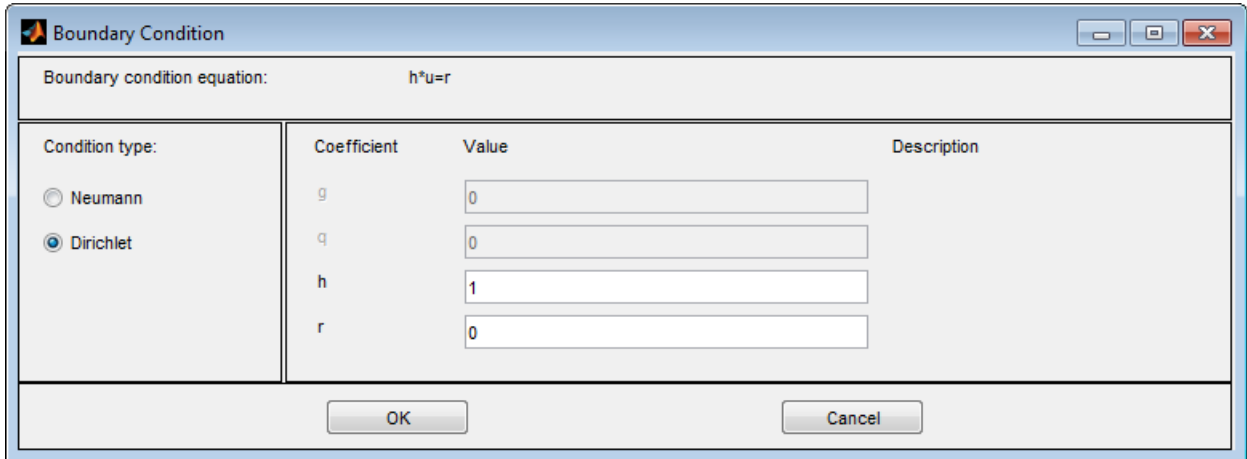
Specify Boundary Conditions opens a dialog box where you can specify the boundary condition for the selected boundary segments. There are three different condition types:

- 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 dialog box for the generic system PDE (**Options > Application > Generic System**).

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
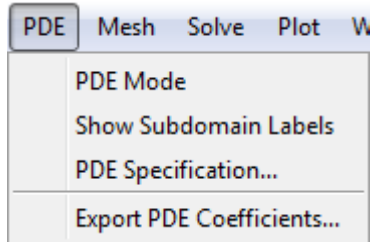
PDE Mode	Enter the partial differential equation mode.
Show Subdomain Labels	Toggle the labeling of the subdomains on/off. The subdomains are labeled using the subdomain numbering in the decomposed geometry matrix.
"PDE Specification in pdetool" on page 4-20	Open dialog box for entering PDE coefficients and types.
Export PDE Coefficients	Export current PDE coefficients to the main workspace. The resulting workspace variables are strings.

PDE Specification in pdetool
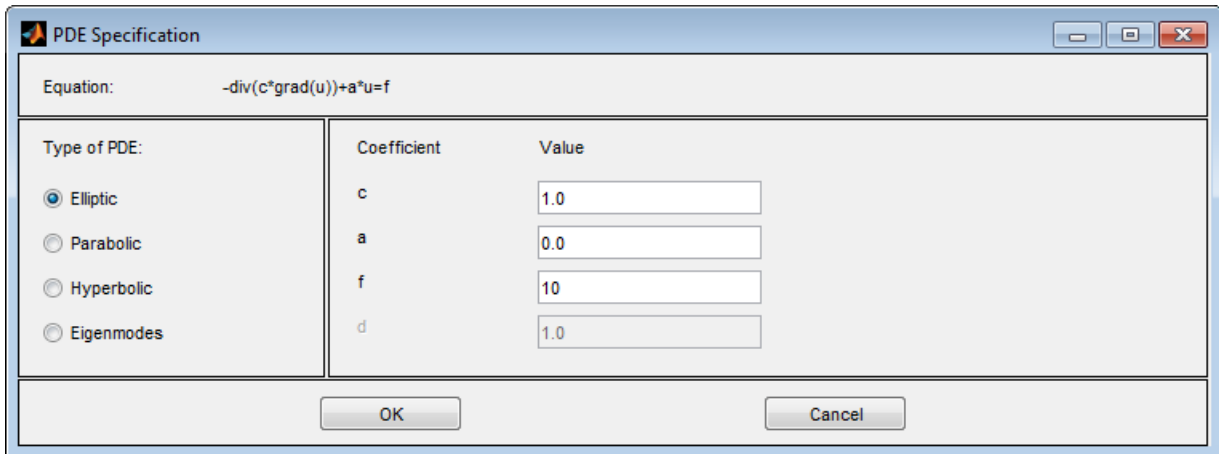
PDE Specification opens a dialog box where you enter the type of partial differential equation and the applicable parameters. The dimension of the parameters is dependent on the dimension of the PDE. The following description applies to scalar PDEs. If a nongeneric application mode is selected, application-specific PDEs and parameters replace the standard PDE coefficients.

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 = $\partial u/\partial t(t_0)$

- 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" on page 4-25	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.
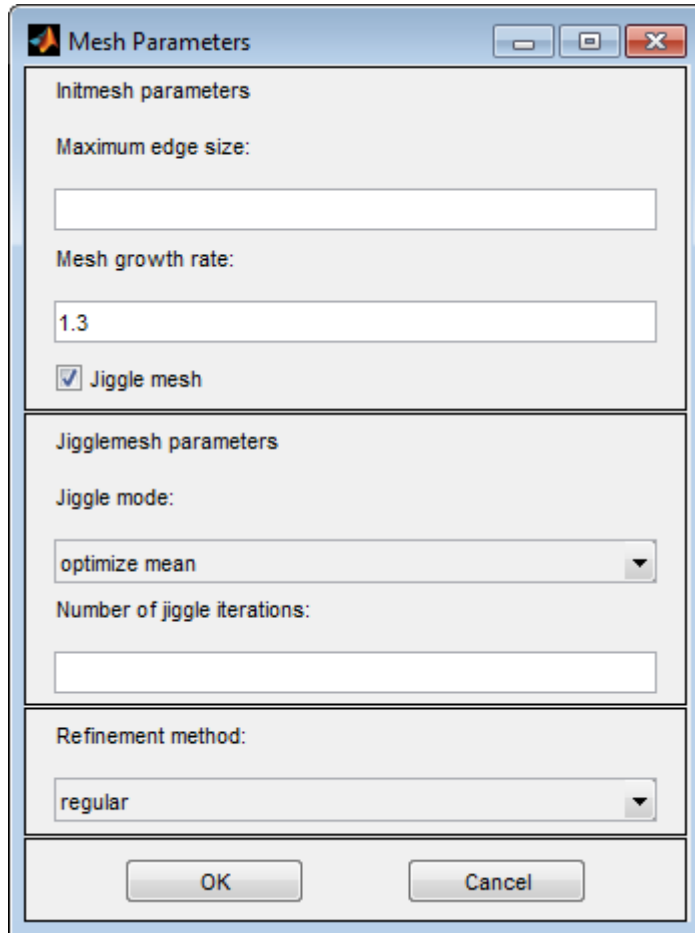
Parameters

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
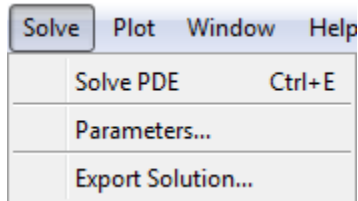
Solve PDE	Solve the partial differential equation for the current CSG model and triangular mesh, and plot the solution (the automatic solution plot can be disabled).
"Parameters" on page 4-28	Open dialog box for entry of PDE solve parameters.
Export Solution	Export the PDE solution vector u and, if applicable, the computed eigenvalues l to the main workspace.
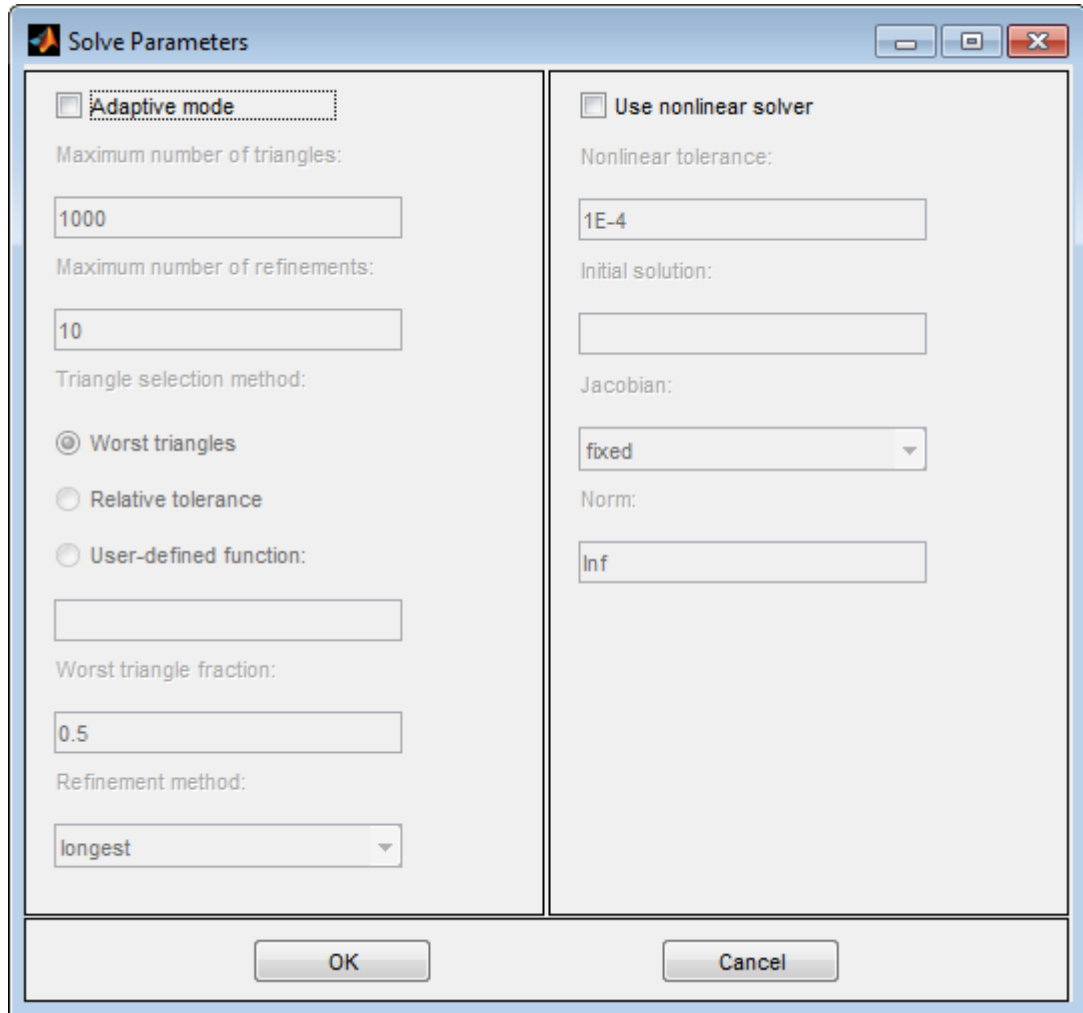
Parameters

Elliptic Equations

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 4-23.

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.
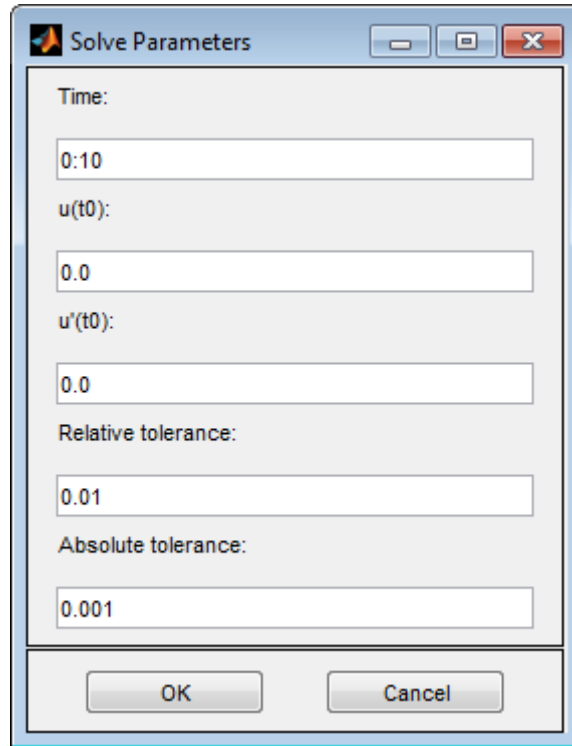
Hyperbolic Equations

- 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.
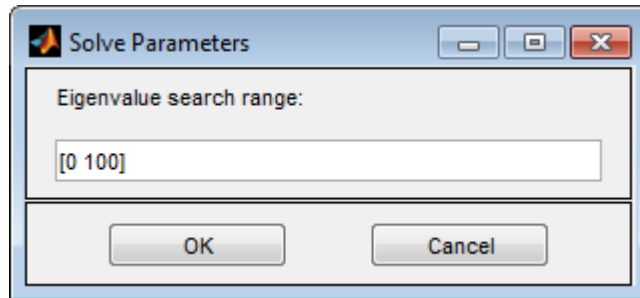
Eigenvalue Equations

- 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
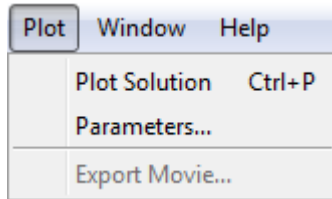
Plot Solution	Display a plot of the solution.
"Parameters" on page 4-33	Open dialog box for plot selection.
Export Movie	If a movie has been recorded, the movie matrix M is exported to the main workspace.
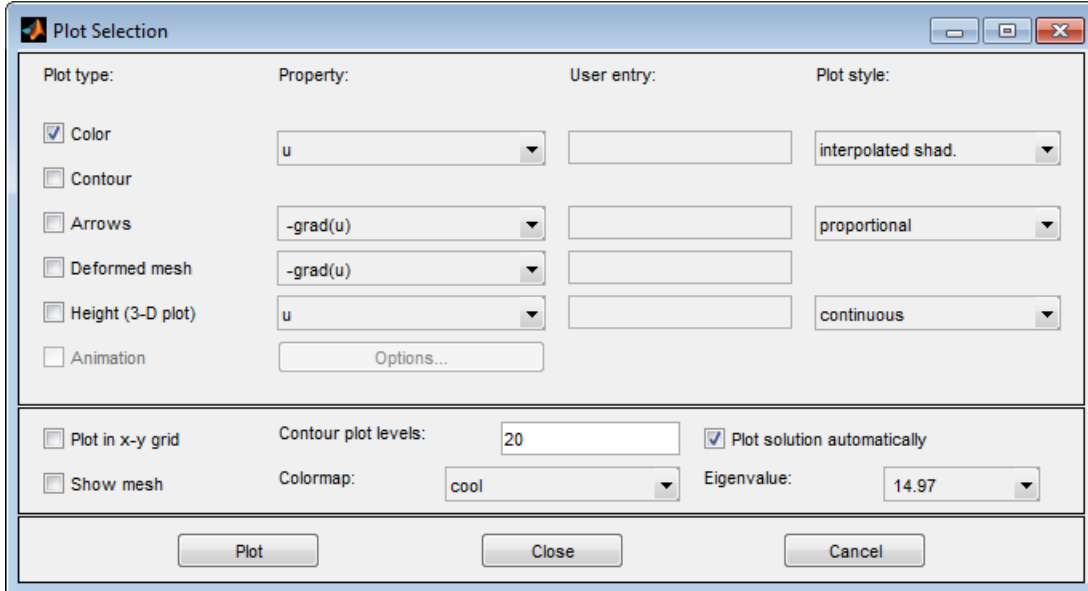
Parameters

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.

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.
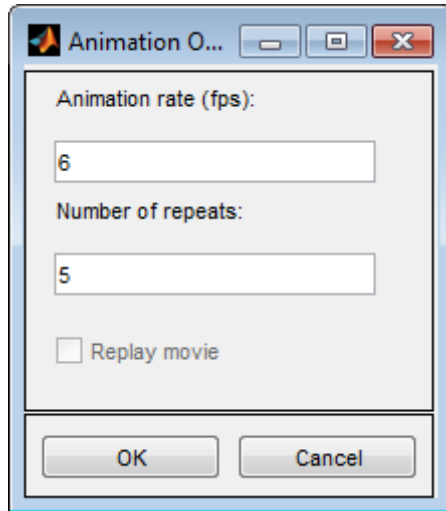
Also, the **Animation** plot type is enabled. In its property field you find an **Options** button. If you press it, an additional dialog box appears. It contains parameters that control the animation:

- **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.
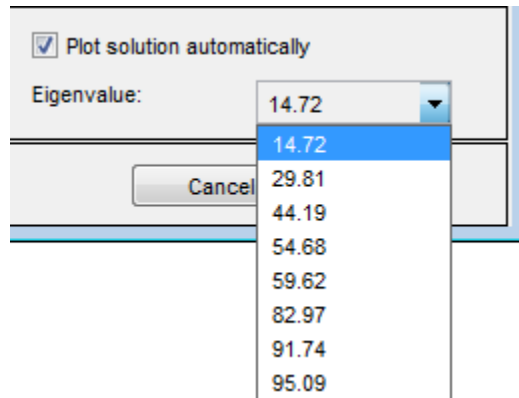
For eigenvalue problems, the bottom right part of the dialog box contains a pop-up menu with all eigenvalues. The plotted solution is the eigenvector associated with the selected eigenvalue. By default, the smallest eigenvalue is selected.

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
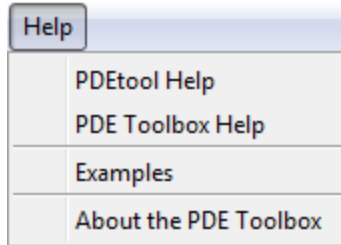
PDETool Help	Open documentation to `pdetool` entry.
PDE Toolbox Help	Open documentation to Partial Differential Equation Toolbox.
Examples	Examples using the software.
About the PDE Toolbox	Display a window with some program information.

5

Finite Element Method

Elliptic Equations

The basic elliptic equation handled by the software is

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

in Ω, 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` 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 $q \to \infty$ 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 "Systems of PDEs" on page 5-10.

Assume that u is a solution of the differential equation. Multiply the equation
with an arbitrary *test function* v and integrate on Ω:

$$\int_\Omega \left(-(\nabla \cdot c\nabla u)v + auv \right) dx = \int_\Omega fv \, dx.$$

Integrate by parts (i.e., use Green's formula) to obtain

$$\int_\Omega \left((c\nabla u) \cdot \nabla v + auv \right) dx - \int_{\partial\Omega} \vec{n} \cdot (c\nabla u)v \, ds = \int_\Omega fv \, dx.$$

The boundary integral can be replaced by the boundary condition:

$$\int_{\Omega} \left((c\nabla u) \cdot \nabla v + auv\right) dx - \int_{\partial\Omega} (-qu+g)v \, ds = \int_{\Omega} fv \, dx.$$

Replace the original problem with *Find u such that*

$$\int_{\Omega} \left((c\nabla u) \cdot \nabla v + auv - fv\right) dx - \int_{\partial\Omega} (-qu+g)v \, ds = 0 \quad \forall v.$$

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 N_p test-functions $\Phi_i \in V_{N_p}$ that form a basis, i.e.,

$$\int_{\Omega} \left((c\nabla u) \cdot \nabla \phi_i + au\phi_i - f\phi_i\right) dx - \int_{\partial\Omega} (-qu+g)\phi_i \, ds = 0, \quad i = 1,...,N_p.$$

Expand u in the same basis of V_{N_p} elements

$$u(x) = \sum_{j=1}^{N_p} U_j \phi_j(x),$$

and obtain the system of equations

$$\sum_{j=1}^{N_p} \left(\int_{\Omega} \left(\left(c\nabla\phi_j \right) \cdot \nabla\phi_i + a\phi_j\phi_i \right) dx + \int_{\partial\Omega} q\phi_j\phi_i \ ds \right) U_j = \int_{\Omega} f\phi_i \ dx + \int_{\partial\Omega} g\phi_i \ ds, \ i = 1, \ldots, N_{p.}$$

Use the following notations:

$$K_{i,j} = \int_{\Omega} \left(c\nabla\phi_j \right) \cdot \nabla\phi_i \ dx \quad \text{(stiffness matrix)}$$

$$M_{i,j} = \int_{\Omega} a\phi_j\phi_i \ dx \quad \text{(mass matrix)}$$

$$Q_{i,j} = \int_{\partial\Omega} q\phi_j\phi_i \ ds$$

$$F_i = \int_{\Omega} f\phi_i \ dx$$

$$G_i = \int_{\partial\Omega} g\phi_i \ ds$$

and rewrite the system in the form

$$(K + M + Q)U = F + G.$$

K, M, and Q are N_p-by-N_p matrices, and F and G are N_p-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 $U \neq 0$.

$$U^T (K + M + Q)U = \int_{\Omega} \left(c|u|^2 + au^2 \right) dx + \int_{\partial \Omega} qu^2 \, ds > 0, \ \text{if } U \neq 0.$$

$U^T(K + M + Q)U$ 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 x_j except for x_i. Requesting $\varphi_i(x_i) = 1$ 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.
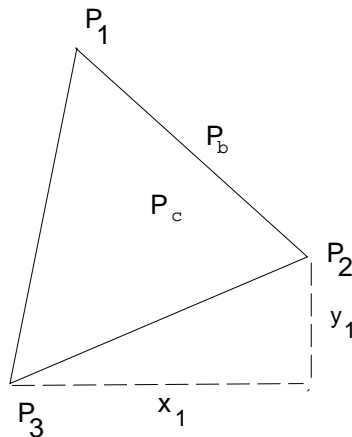
The Local Triangle P1P2P3

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:

$$m_{i,j} = \int_{\Delta P_1 P_2 P_3} a(P_c)\phi_i(x)\phi_j(x)\, dx = a(P_c)\frac{\text{area}(\Delta P_1 P_2 P_3)}{12}\big(1+\delta_{i,j}\big),$$

where P_c is the center of mass of $\Delta\, P_1 P_2 P_3$, 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 $P_1 P_2 P_3$. Since the basis functions are linear on the triangle $P_1 P_2 P_3$, the gradients are constants. Denote the basis functions φ_1, φ_2, and φ_3 such that $\varphi(P_i) = 1$. If $P_2 - P_3 = [x_1, y_1]^T$ then we have that

$$\nabla \phi_1 = \frac{1}{2 \,\text{area}(\Delta P_1 P_2 P_3)} \begin{bmatrix} y_1 \\ -x_1 \end{bmatrix}$$

and after integration (taking c as a constant matrix on the triangle)

$$k_{i,j} = \frac{1}{4 \,\text{area}(\Delta P_1 P_2 P_3)} \begin{bmatrix} y_j, -x_j \end{bmatrix} c(P_c) \begin{bmatrix} y_1 \\ -x_1 \end{bmatrix}.$$

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

$$Q_{i,j} = q(P_b) \frac{\|P_1 - P_2\|}{6} (1 + \delta_{i,j}), \quad i,j = 1,2$$

and

$$G_i = g(P_b) \frac{\|P_1 - P_2\|}{2}, \quad i = 1,2$$

where P_b is the midpoint of $P_1 P_2$.

For each triangle the vertices P_m of the local triangle correspond to the indices i_m of the mesh points. The contributions of the individual triangle are added to the matrices such that, e.g.,

$$K_{i_m,i_n} t \leftarrow K_{i_m,i_n} + k_{m,n}, \quad m,n = 1,2,3.$$

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.

Systems of PDEs

Partial Differential Equation Toolbox software can also handle systems of N partial differential equations over the domain Ω. We have the elliptic system

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \mathbf{f},$$

the parabolic system

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

the hyperbolic system

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

and the eigenvalue system

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \lambda \mathbf{du},$$

where $\mathbf{c}$ is an N-by-N-by-2-by-2 tensor. By the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$, 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 $\mathbf{a}$ and $\mathbf{d}$ 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 $\mathbf{c}$, $\mathbf{a}$, $\mathbf{d}$, 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,

$$\mathbf{h}\mathbf{u} = \mathbf{r}$$
$$\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{q}\mathbf{u} = \mathbf{g} + \mathbf{h}'\mu.$$

By the notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ we mean the N-by-1 matrix with $(i,1)$-component

$$\sum_{j=1}^{N} \left(\cos(\alpha) c_{i,j,1,1} \frac{\partial}{\partial x} + \cos(\alpha) c_{i,j,1,2} \frac{\partial}{\partial y} + \sin(\alpha) c_{i,j,2,1} \frac{\partial}{\partial x} + \sin(\alpha) c_{i,j,2,2} \frac{\partial}{\partial y} \right) \mathbf{u}_j$$

where the outward normal vector of the boundary is $\mathbf{n} = (\cos(\alpha), \sin(\alpha))$. There are M Dirichlet conditions and the $\mathbf{h}$-matrix is M-by-N, $M \geq 0$. The generalized Neumann condition contains a source $\mathbf{h}'\mu$, 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 "Elliptic Equations" on page 5-2 for the scalar case, which is equivalent to a diagonal $\mathbf{h}$-matrix. For the general case, Dirichlet conditions

$$\mathbf{h}\mathbf{u} = \mathbf{r}$$

are approximated by adding a term

$$L(\mathbf{h}'\mathbf{hu} - \mathbf{h}'\mathbf{r})$$

to the equations $\mathbf{KU} = \mathbf{F}$, where L is a large number such as 10^4 times a representative size of the elements of K.

When this number is increased, $\mathbf{hu} = \mathbf{r}$ 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 $\mathbf{h}$, 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 $N_p N = N_u$. 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, $\mathbf{hu} = \mathbf{r}$. 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.

Parabolic Equations

Reducing Parabolic Equations 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

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

with the initial condition

$$u(x,0) = u_0(x) \text{ for } x\epsilon\Omega$$

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

$$\sum_i \int_\Omega d\phi_j \phi_i \, \frac{dU_i(t)}{dt} \, dx + \sum_i \left(\int_\Omega \left(\nabla\phi_j \cdot \left(c\nabla\phi_i \right) + a\phi_j\phi_i \right) dx + \int_{\partial\Omega} q\phi_j\phi_i \, ds \right) U_i(t)$$
$$= \int_\Omega f\phi_j \, dx + \int_{\partial\Omega} g\phi_j \, ds \quad \forall j.$$

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

$$U_i(0) = u_0(x_i)$$

yields the solution to the PDE at each node x_i and time t. Note that 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 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 $\mathbf{h}(t)$ 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 $\mathbf{h}$ 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.

Solve a Parabolic Equation

This example shows how to solve a parabolic equation and 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.

Hyperbolic Equations

Using the same ideas as for the parabolic equation, `hyperbolic` implements the numerical solution of

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

for x in Ω, with the initial conditions

$$u(x,0) = u_0(x)$$
$$\frac{\partial u}{\partial t}(x,0) = v_0(x)$$

for all x in Ω, and usual boundary conditions. In particular, solutions of the equation u_{tt} - cΔu = 0 are waves moving with speed $\sqrt{c}$.

Using a given triangulation of Ω, the method of lines yields the second order ODE system

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

with the initial conditions

$$U_i(0) = u_0(x_i) \quad \forall i$$
$$\frac{d}{dt}U_i(0) = v_0(x_i) \quad \forall i$$

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

$$-\nabla \cdot (c\nabla u) + au = f \text{ and } -\nabla \cdot (0\nabla u) + du = 0.$$

Eigenvalue Equations

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 3-99, 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 $\lambda_i = \mu + 1/\theta_i$ 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,

$$AV_j = V_jH_{j,j} + E_j.$$

(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 $H_{j,j}$,

$$H_{j,j} s_i = s_i \theta_i, \;\; i = 1, ..., j.$$

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,

$$x = QU^{-1}L^{-1}PMv_j$$

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 elliptic PDE, from the **Solve** menu in the `pdetool` GUI, select **Parameters**. Then, select the **Use nonlinear solver** check box and click **OK**. At the command line, use `pdenonlin` instead of `assempde`.

The `parabolic` and `hyperbolic` functions automatically detect when they need to use a nonlinear solver.

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

$$\begin{aligned}
0 = \rho(U) = \sum_j \Bigg(&\int_\Omega \Big(c(x,U)\nabla\phi_j(x)\Big) \cdot \nabla\phi_j(x) + a(x,U)\phi_j(x)\phi_i(x)\,dx \\
&+ \int_{\partial\Omega} q(x,U)\phi_j(x)\phi_i(x)\,ds \Bigg) U_j \\
&- \int_\Omega f(x,U)\phi_i(x)\,dx - \int_{\partial\Omega} g(x,U)\phi_i(x)\,ds
\end{aligned}$$

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

$$\frac{\partial \rho\left(U^{(n)}\right)}{\partial U}\left(U^{(n+1)} - U^{(n)}\right) = -\alpha\rho\left(U^{(n)}\right),$$

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 $\min_U \left\|\rho(U)\right\|$.

It is well known that for sufficiently small α

$$\left\|\rho\left(U^{(n+1)}\right)\right\| < \left\|\rho\left(U^{(n)}\right)\right\|$$

and

$$p_n = \left(\frac{\partial \rho\left(U^{(n)}\right)}{\partial U}\right)^{-1}\rho\left(U^{(n)}\right)$$

is called a descent direction for $\left\|\rho(U)\right\|$, where $\|\cdot\|$ is the L_2-norm. The iteration is

$$U^{(n+1)} = U^{(n)} + \alpha p_n,$$

where $\alpha \leq 1$ 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:

$$\left\| \rho\left(U^{(n)}\right) \right\| - \left\| \rho\left(U^{(n)}\right) + \alpha\, p_n \right\| \geq \frac{\alpha}{2} \left\| \rho\left(U^{(n)}\right) \right\|$$

which guarantees a reduction of the residual norm by at least $1 - \alpha/2$. 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 $\alpha \to 1$ 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

$$\frac{\rho(U^{(n)} + \varepsilon\phi_i) - \rho(U^{(n)})}{\varepsilon}$$

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 $U{\rightarrow}KU$, proceed as follows. The a term has been omitted for clarity, but appears again in the final result.

$$\frac{\partial(KU)_i}{\partial U_j} = \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \sum_l \left(\int_\Omega c\left(U + \varepsilon\phi_j\right) \nabla\phi_l \nabla\phi_i \ dx\left(U_l + \varepsilon\delta_{l,j}\right)\right.$$

$$\left. - \int_\Omega c(U)\nabla\phi_l\nabla\phi_i \ dxU_l \right)$$

$$= \int_\Omega c(U)\nabla\phi_j\nabla\phi_i \ dx + \sum_l \int_\Omega \phi_j \frac{\partial c}{\partial u} \nabla\phi_l\nabla\phi_i \ dxU_l.$$

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 $\Sigma_j \varphi_j = 1$, the second term is approximated by

$$\delta_{i,j} \sum_l \int_\Omega \frac{\partial c}{\partial u} \nabla\phi_l\nabla\phi_i \ dxU_l$$

which is the ith component of $K^{(c')}U$, where $K^{(c')}$ is the stiffness matrix associated with the coefficient $\partial c/\partial u$ rather than c. The same reasoning can be applied to the derivative of the mapping $U \to MU$. The derivative of the mapping $U \to -F$ is exactly

$$-\int_\Omega \frac{\partial f}{\partial u} \phi_i\phi_j \ dx$$

which is the mass matrix associated with the coefficient $\partial f/\partial u$. Thus the Jacobian of the residual $\rho(U)$ is approximated by

$$J = K^{(c)} + M^{(a-f')} + \mathrm{diag}\left(\left(K^{(c')} + M^{(a')}\right)U\right)$$

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.

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.

6

Function Reference

2-D Geometry

csgchk	Check validity of Geometry Description matrix
csgdel	Delete borders between minimal regions
decsg	Decompose Constructive Solid Geometry into minimal regions
pdearcl	Interpolation between parametric representation and arc length
pdecirc	Draw circle
pdeellip	Draw ellipse
pdegeom	Write custom function for defining geometry
pdepoly	Draw polygon
pderect	Draw rectangle
wgeom	Write geometry specification function

Boundary Conditions

Meshing

adaptmesh	Adaptive mesh generation and PDE solution
initmesh	Create initial triangular mesh
jigglemesh	Jiggle internal points of triangular mesh
pdeadgsc	Select triangles using relative tolerance criterion
pdeadworst	Select triangles relative to worst value
pdeent	Indices of triangles neighboring given set of triangles
pdeintrp	Interpolate from node data to triangle midpoint data
pdejmps	Error estimates for adaptation
pdemesh	Plot PDE triangular mesh
pdeprtni	Interpolate from triangle midpoint data to node data
pdesde	Indices of points/edges/triangles in set of subdomains
pdesdp	Indices of points/edges/triangles in set of subdomains
pdesdt	Indices of points/edges/triangles in set of subdomains
pdetrg	Triangle geometry data
pdetriq	Triangle quality measure
refinemesh	Refine triangular mesh
tri2grid	Interpolate from PDE triangular mesh to rectangular grid

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
dst	Discrete sine transform
hyperbolic	Solve hyperbolic PDE problem
idst	Discrete sine transform
parabolic	Solve parabolic PDE problem
pdecgrad	Flux of PDE solution
pdeeig	Solve eigenvalue PDE problem
pdegrad	Gradient of PDE solution
pdenonlin	Solve nonlinear PDE problem
pdesmech	Calculate structural mechanics tensor functions
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
poimesh	Make regular mesh on rectangular geometry
poisolv	Fast solution of Poisson's equation on rectangular grid
sptarn	Solve generalized sparse eigenvalue problem

Solution Visualization and Analysis

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

PDE Toolbox GUI

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

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

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

for $(x,y) \, \epsilon \, \Omega$, or the elliptic system PDE problem

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \mathbf{f},$$

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 "Scalar PDE Coefficients" on page 2-14 and "Coefficients for Systems of PDEs" on page 2-38.

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 Nt columns, where Nt 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)
```
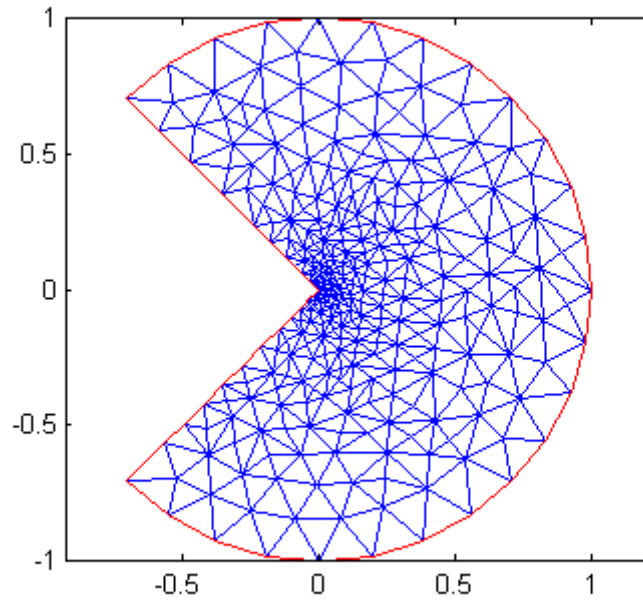
The maximum absolute error is 0.0028, with 629 triangles.

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)
```

adaptmesh
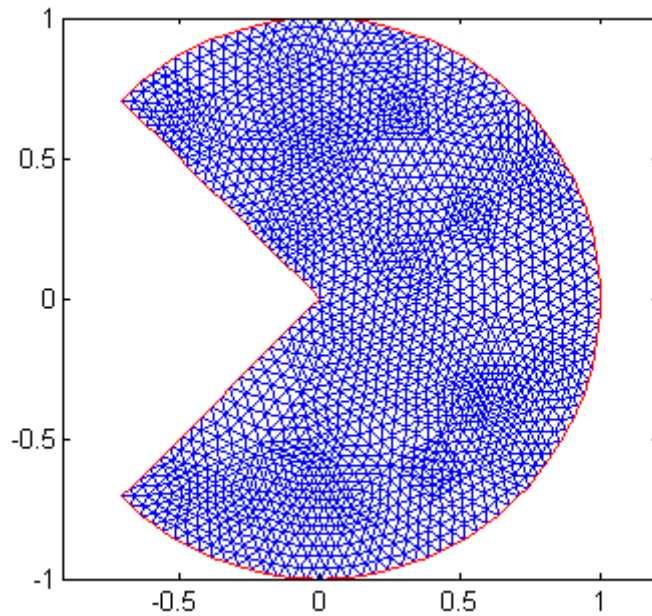
Uniform refinement with 3152 triangles produces an error of 0.0078. This error is over three times as large as that produced by the adaptive method (0.0028) with many fewer triangles (629). For a problem with regular solution, we expect a $O(h^2)$ error, but this solution is singular since $u \approx r^{1/3}$ 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 | initmesh | pdeadgsc | pdeadworst | pdejmps | refinemesh

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

How To   • "Scalar PDE Coefficients" on page 2-14

                     • "Coefficients for Systems of PDEs" on page 2-38

assemb

Purpose   Assemble boundary condition contributions

Syntax   [Q,G,H,R]=assemb(b,p,e)
                  [Q,G,H,R]=assemb(b,p,e,uO)
                  [Q,G,H,R]=assemb(b,p,e,uO,time)
                  [Q,G,H,R]=assemb(b,p,e,uO,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, uO, 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:

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

                  • 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 $\mathbf{hu} = \mathbf{r}$.

                    $$\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{qu} = \mathbf{g} + \mathbf{h}' \mu.$$

The notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ 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 $\mathbf{q}$. The lengths are stored in column-wise order with respect to $\mathbf{q}$.

- 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 $\mathbf{h}$. The lengths are stored in columnwise order with respect to $\mathbf{h}$.

- Row $3 + N^2 + N + MN$ 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.

Examples   **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 ($M = 0$)

$$\mathbf{n} \cdot (c \nabla u) = -x^2$$

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

$$u = x^2 - y^2$$

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 ($M = 1$):

$$\begin{pmatrix} h_{11} & h_{12} \end{pmatrix} \mathbf{u} = r_1$$

$$\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \begin{pmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{pmatrix} \mathbf{u} = \begin{pmatrix} g_1 \\ g_2 \end{pmatrix} + \mathbf{s}$$

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 | pdebound

How To   • "Boundary Conditions for Scalar PDE" on page 2-63

• "Boundary Conditions for PDE Systems" on page 2-68

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 "Elliptic Equations" on page 5-2. The command assempde assembles the scalar PDE problem

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

for $(x,y) \ \epsilon \ \Omega$, or the system PDE problem

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \mathbf{f}.$$

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 "Systems of PDEs" on page 5-10 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.

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 executes an adaptive solution for this problem.
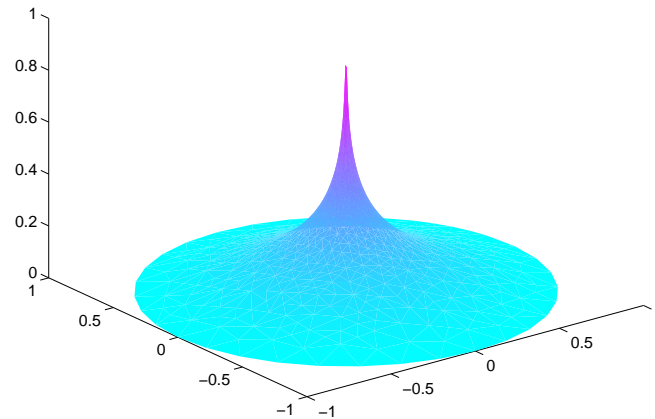
See Also   assema | assemb | initmesh | pdebound | refinemesh

How To

- "Scalar PDE Coefficients" on page 2-14
- "Coefficients for Systems of PDEs" on page 2-38
- "Boundary Conditions for Scalar PDE" on page 2-63
- "Boundary Conditions for PDE Systems" on page 2-68

csgchk

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`

csgdel

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`	`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.

Algorithms   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 | csgdel | pdebound | pdecirc | pdeellip | pdegeom | pdepoly | pderect | pdetool | wbound | wgeom

dst

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:

$$y(k) = \sum_{n=1}^{N} x(n)\sin\left(\pi\frac{kn}{N+1}\right), \quad k = 1,...,N.$$

`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:

$$y(k) = \frac{2}{N+1} \sum_{n=1}^{N} x(n)\sin\left(\pi\frac{kn}{N+1}\right), \, k = 1,...,N.$$

`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 "Solve Poisson's Equation on a Grid" on page 3-114.

See Also   poiasma | poiindex | poisolv

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

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

for $(x,y) \, \epsilon \, \Omega$, or the system PDE problem

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

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. They can also depend on `u`, the solution, and on the components of the gradient of `u`, namely `ux` and `uy`. For a complete listing of all options, see "Scalar PDE Coefficients" on page 2-14 and "Coefficients for Systems of PDEs" on page 2-38.

`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

$$B'MB\frac{d^2u_i}{dt^2} + K \cdot u_i = F$$

$$u = Bu_i + u_d$$

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

$$u(0) = \text{atan}(\cos(\pi x))$$

and

$$\frac{du}{dt}(0) = 3\sin(\pi x)\exp\big(\cos(\pi y)\big).$$

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 | parabolic

How To   • "Scalar PDE Coefficients" on page 2-14

• "Coefficients for Systems of PDEs" on page 2-38

• "Boundary Conditions for Scalar PDE" on page 2-63

• "Boundary Conditions for PDE Systems" on page 2-68

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 |

# initmesh

| 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.

Algorithms   `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 Δ 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
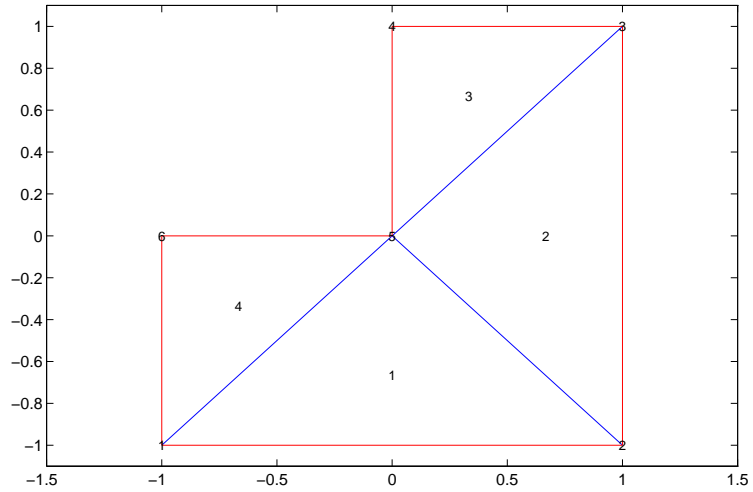

e
e =
     1     2     3     4     5     6
     2     3     4     5     6     1
     0     0     0     0     0     0
     1     1     1     1     1     1
     1     2     3     4     5     6
     1     1     1     1     1     1
     0     0     0     0     0     0
```

```
t
t =
   1   2   3   1
   2   3   4   5
   5   5   5   6
   1   1   1   1
```

References   George, P. L., *Automatic Mesh Generation — Application to Finite Element Methods,* Wiley, 1991.

See Also   decsg | jigglemesh | pdegeom | refinemesh

How To   • "Mesh Data" on page 2-76

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 | pdetriq

How To   • "Mesh Data" on page 2-76

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

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

for $(x,y) \ \epsilon \ \Omega$, or the system PDE problem

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

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. They can also depend on u, the solution, and on the components of the gradient of u, namely ux and uy. For a complete listing of all options, see "Scalar PDE Coefficients" on page 2-14 and "Coefficients for Systems of PDEs" on page 2-38.

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

$$B'MB\frac{du_i}{dt} + K \cdot u_i = F$$
$$u = Bu_i + u_d$$

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,1,0,0,1);
```

parabolic

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 | hyperbolic

How To   • "Scalar PDE Coefficients" on page 2-14

• "Coefficients for Systems of PDEs" on page 2-38

• "Boundary Conditions for Scalar PDE" on page 2-63

• "Boundary Conditions for PDE Systems" on page 2-68

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 "Mesh Data" on page 2-76.

c,a, and f are PDE coefficients. For details, see "Scalar PDE
Coefficients" on page 2-14 and "Coefficients for Systems of PDEs" on
page 2-38.

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 | pdejmps

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 "Mesh Data" on page 2-76.

                     c, a, and f are PDE coefficients. For details, see "Scalar PDE
                     Coefficients" on page 2-14.

                     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 | assempde | initmesh | pdejmps

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

pdebound

Purpose Write custom function for defining boundary conditions

Syntax `[qmatrix,gmatrix,hmatrix,rmatrix]=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

$$\mathbf{hu} = \mathbf{r}$$
$$\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{qu} = \mathbf{g} + \mathbf{h}'\mu.$$

By the notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ 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 $\mathbf{n} = (\cos(\alpha), \sin(\alpha))$. There are M Dirichlet conditions and the h-matrix is M-by-N, $M \geq 0$.

The generalized Neumann condition contains a source $\mathbf{h}'\mu$ 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 `[qmatrix,gmatrix,hmatrix,rmatrix] = 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.

Similarly, if an output argument depends on u, then the output argument must return a matrix of NaN of the correct size if the input u is NaN. This requirement signals to the solver, such a parabolic or hyperbolic, that the argument depends on time or the solution u.

The solution u is represented by the solution vector u. Details on the representation can be found in the entry on assempde.

qmatrix and gmatrix must contain the value of q and g on the midpoint of each boundary. Thus we have size(qmatrix)=[N^2 ne], where N is the dimension of the system, and ne the number of edges in e, and size(gmatrix)=[N ne]. For the Dirichlet case, the corresponding values must be zeros.

hmatrix and rmatrix 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(hmatrix)=[N^2 2*ne], where N is the dimension of the system, and ne the number of edges in e, and size(rmatrix)=[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 qmatrix and hmatrix.

Examples   For the boundary conditions

$$\begin{pmatrix} 1 & -1 \end{pmatrix}\mathbf{u} = 2$$

$$\mathbf{n} \cdot (\mathbf{c} \otimes \nabla\mathbf{u}) + \begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix}\mathbf{u} = \begin{pmatrix} 3 \\ 4 \end{pmatrix} + \mathbf{h}'\mu$$

pdebound

the following values should be stored in `qmatrix`, `gmatrix`, `hmatrix`, and `rmatrix`

```
                1
qmatrix = [ ... 2 ... ]
                2
                0

gmatrix = [ ... 3 ... ]
                4

                1     1
hmatrix = [ ... 0 ... 0 ... ]
               -1    -1
                0     0

rmatrix = [ ... 2 ... 2 ... ]
                0     0
```

See Also   initmesh | pdeent | pdegeom | pdesdt

How To   • "Boundary Conditions for Scalar PDE" on page 2-63

   • "Boundary Conditions for PDE Systems" on page 2-68

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`"Scalar PDE Coefficients" on page 2-14 and "c for Systems" on page 2-42.

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`.

pdecgrad

See Also   `assempde`

7-52

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 | pdepoly | pderect | pdetool

pdecont

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 | pdeplot | pdesurf

pdeeig

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

$$-\nabla \cdot (c\nabla u) + au = \lambda du \text{ on } \Omega$$

or the system PDE eigenvalue problem

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \lambda du \text{ on } \Omega$$

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 "Mesh Data" on page 2-76.

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 "Scalar PDE Coefficients" on page 2-14 and "Coefficients for Systems of PDEs" on page 2-38.

[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
```

Algorithms   pdeeig calls sptarn to calculate eigenvalues. For details of the algorithm, see the sptarn reference pages.

pdeeig

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

How To   • "Scalar PDE Coefficients" on page 2-14

- "Coefficients for Systems of PDEs" on page 2-38

- "Boundary Conditions for Scalar PDE" on page 2-63

- "Boundary Conditions for PDE Systems" on page 2-68

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 | pdepoly | pderect | pdetool

pdeent

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

Purpose
Write custom function for defining geometry

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 | pdearcl | refinemesh

How To   • "Boundary Conditions for Scalar PDE" on page 2-63

• "Boundary Conditions for PDE Systems" on page 2-68

- "Mesh Data" on page 2-76

Purpose	Plot PDE geometry

Syntax

```
pdegplot(g)
h = pdeplot(g)
h = pdeplot(g,Name,Value)
```

Description

pdegplot(g) plots the geometry of a PDE problem.

h = pdeplot(g) returns handles to the figure axes.

h = pdeplot(g,Name,Value) plots with additional options specified by one or more Name,Value pair arguments.

Input Arguments

g

Decomposed geometry matrix, as produced by decsg, or by selecting **Boundary > Export Decomposed Geometry, Boundary Cond's** from the pdetool. g can also be the name of a geometry file (see wgeom for details).

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments, where Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

edgeLabels

'on' shows the label for each boundary edge. These are the same as the labels that pdetool produces when you select **Boundary > Show Edge Labels**.

Default: 'off'

subdomainLabels

'on' shows the label for each subdomain. These are the same as the labels that pdetool produces when you select **PDE > Show Subdomain Labels**.

Default: 'off'

Output Arguments

h

Vector of handles to the figure axes.

Examples

Plot Geometry

Plot the geometry of a region defined by a few simple shapes.

```
g = [2 1 1 1 1 1 1 1 1 4 4;
-1 -0.55 -0.5 -0.45 -0.5 0.45 0.5 0.55 0.5 -1 0.169101978725763;
1 -0.5 -0.45 -0.5 -0.55 0.5 0.55 0.5 0.45 0.169101978725763 1;
0 -0.25 -0.3 -0.25 -0.2 -0.25 -0.3 -0.25 -0.2 0 -0.739198919740117;
0 -0.3 -0.25 -0.2 -0.25 -0.3 -0.25 -0.2 -0.25 -0.739198919740117 0;
0 0 0 0 0 0 0 0 0 1 1;
1 1 1 1 1 1 1 1 1 1 0 0;
0 -0.5 -0.5 -0.5 -0.5 0.5 0.5 0.5 0.5 0 0;
0 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 -0.25 0 0;
0 0.05 0.05 0.05 0.05 0.05 0.05 0.05 0.05 1 1;
0 0 0 0 0 0 0 0 0 0.75 0.75;
0 0 0 0 0 0 0 0 0 0 0];
pdegplot(g)
```

View the edge labels and the subdomain label.

```
pdegplot(g,'edgeLabels','on','subdomainLabels','on')
ylim([-.8,.1]) % to see the top edge clearly
```

pdegplot

Alternatives   If you create the geometry in `pdetool`, you can view the geometry from **Boundary Mode**. You can see the edge labels by selecting **Boundary > Show Edge Labels**, and you can see the subdomain labels by selecting **PDE > Show Subdomain Labels**.

See Also   `decsg` | `pdetool` | `wgeom`

Related Examples   • "Scalar PDE Functional Form and Calling Syntax" on page 2-22

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

How To   • "Gradient or Derivatives of u" on page 2-21

• "f for Systems" on page 2-40

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 | initmesh | pdeprtni
How To	• "Interpolated u" on page 2-20
	• "f for Systems" on page 2-40

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 "Scalar PDE Coefficients" on page
2-14 and "Coefficients for Systems of PDEs" on page 2-38 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

$$E(K) = \alpha \left\| h^m \left(f - au \right) \right\|_K + \beta \left(\frac{1}{2} \sum_{\tau \in \partial K} h_\tau^{2m} [\mathbf{n}_\tau \cdot (c\nabla u_h)]^2 \right)^{1/2},$$

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 2-77.

See Also   `adaptmesh` | `pdeadgsc` | `pdeadworst`

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

```
pdemesh(p,e,t,u)
```

See Also   pdecont | pdeplot | pdesurf

How To   • "Mesh Data" on page 2-76

    • "Scalar PDE Functional Form and Calling Syntax" on page 2-22

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

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \mathbf{f} \text{ on } \Omega$$

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. 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 "Scalar PDE Coefficients" on page 2-14
and "Coefficients for Systems of PDEs" on page 2-38.

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 — Use the syntax of "Initial Conditions" on page 2-54
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 "Nonlinear Equations" on page 5-24 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 $\nabla \cdot (1/u\nabla u) = 0$ 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 | pdebound

How To
- "Scalar PDE Coefficients" on page 2-14
- "Coefficients for Systems of PDEs" on page 2-38
- "Boundary Conditions for Scalar PDE" on page 2-63
- "Boundary Conditions for PDE Systems" on page 2-68
- "Initial Conditions" on page 2-54

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');
```

pdeplot

```
u=assempde('lshapeb',p,e,t,1,0,1);
pdeplot(p,e,t,'xydata',u,'zdata',u,'mesh','off');
```

See Also   pdecont | pdegplot | pdemesh | pdesurf

How To   • "Scalar PDE Functional Form and Calling Syntax" on page 2-22

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 | pderect | pdetool

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 | initmesh | pdeintrp

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 | pdeellip | pdepoly | pdetool

pdesdp

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, which is $\dfrac{\partial u}{\partial x}$

- uy, which is $\dfrac{\partial u}{\partial y}$

- vx, which is $\dfrac{\partial v}{\partial x}$

- vy, which is $\dfrac{\partial v}{\partial y}$

- 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

$$\sqrt{(\sigma_1^2 + \sigma_2^2)(v^2 - v + 1) + \sigma_1 \sigma_2 \left(2v^2 - 2v - 1\right)}$$

where v is Poisson's ratio nu.

Examples Assuming that a problem has been solved using the application mode "Structural Mechanics, Plane Stress," discussed in "Structural Mechanics — Plane Stress" on page 3-6, 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` | `pdemesh` | `pdeplot`

pdetool

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 ∂Ω 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 4-15.)

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

- Hyperbolic PDE: $d\dfrac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f$
- 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 "Scalar PDE Coefficients" on page 2-14 and "Coefficients for Systems of PDEs" on page 2-38 and "PDE Menu" on page 4-19.)

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 | initmesh | parabolic | pdecont | pdeeig | pdesurf

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 $h_1 = h_2 = h_3$.

References   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` | `jigglemesh` | `refinemesh`

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 | poisolv

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 | idst | poiasma | poiindex | poisolv

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` | `poisolv`

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 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 | poisolv

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).

References   Strang, Gilbert, *Introduction to Applied Mathematics*,
Wellesley-Cambridge Press, Cambridge, MA, 1986, pp. 453–458.

See Also   poicalc | poimesh

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 "Mesh Data" on page 2-76.

`[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.

refinemesh

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
```

Algorithms   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 | pdeent | pdegeom | pdesdt

How To   • "Mesh Data" on page 2-76

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.

Algorithms   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.

sptarn

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

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 | initmesh | refinemesh

wbound

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`. The output `fid` is `-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` \| `pdebound` \| `pdegeom` \| `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` | `pdegeom` | `wbound`

How To   • "Boundary Conditions for Scalar PDE" on page 2-63

                   • "Boundary Conditions for PDE Systems" on page 2-68

Index