Partial Differential Equation Toolbox™
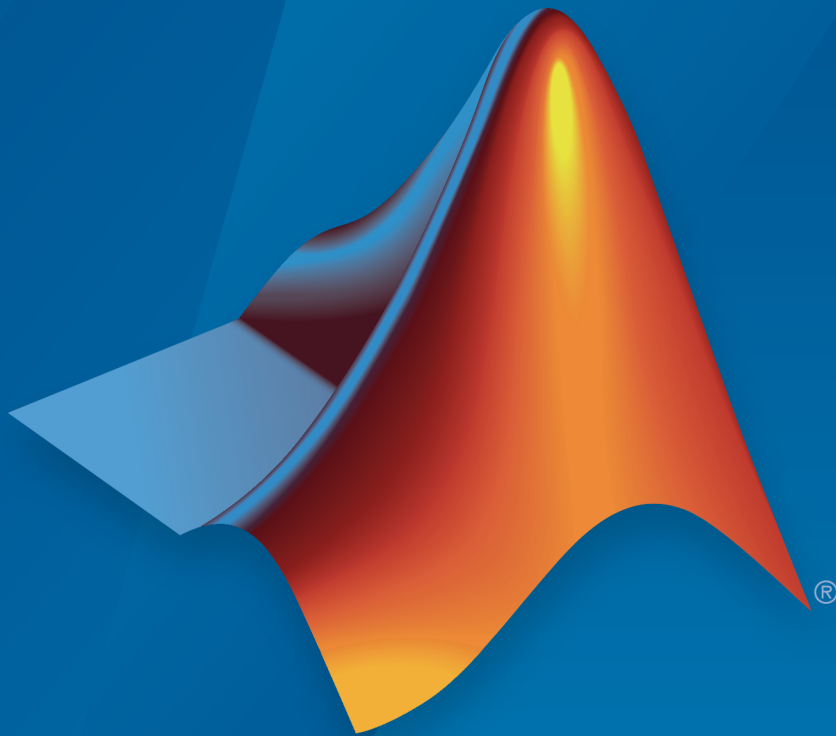
User's Guide

MATLAB®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Contents

3

PDE App

4

Finite Element Method

5

Functions — Alphabetical List

6

Getting Started

Partial Differential Equation Toolbox Product Description

Solve partial differential equations using finite element analysis

Partial Differential Equation Toolbox provides functions for solving partial differential equations (PDEs) in 2-D, 3-D, and time using finite element analysis. It lets you specify and mesh 2-D and 3-D geometries and formulate boundary conditions and equations. You can solve static, time domain, frequency domain, and eigenvalue problems over the domain of the geometry. Functions for postprocessing and plotting results enable you to visually explore the solution.

You can use Partial Differential Equation Toolbox to solve PDEs from standard problems such as diffusion, heat transfer, structural mechanics, electrostatics, magnetostatics, and AC power electromagnetics, as well as custom, coupled systems of PDEs.

Key Features

- Solvers for coupled systems of PDEs: static, time domain, frequency domain, and eigenvalue
- PDE specification for elliptic, parabolic, and hyperbolic problems
- Boundary condition specification: Dirichlet, generalized Neumann, and mixed
- Functions for 2-D geometry creation and 3-D geometry import from STL files
- Automatic meshing using tetrahedra and triangles
- Simultaneous visualization of multiple solution properties, mesh overlays, and animation

Equations You Can Solve Using Legacy Functions

> **Note:** **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "Equations You Can Solve Using Recommended Functions" on page 1-6.

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 or is a bounded 3-D region. c, a, f, and the unknown u are scalar, complex valued functions defined on Ω. c can be a matrix function on Ω (see "c Coefficient for Systems" on page 2-125). 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 of 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 PDE app, select **Parameters**. Then, select the **Use nonlinear solver** check box and click **OK**.

For eigenvalue problems, the coefficients cannot depend on the solution u or its gradient.

All solvers can handle the system case of N coupled equations. You can solve $N = 1$ or 2 equations using the PDE app, 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 2-93.

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 3-128.

Equations You Can Solve Using Recommended Functions

> **Note:** **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see "Equations You Can Solve Using Legacy Functions" on page 1-3.

Partial Differential Equation Toolbox solves scalar equations of the form

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

and eigenvalue equations of the form

$$-\nabla\cdot(c\nabla u) + au = \lambda d u$$
or
$$-\nabla\cdot(c\nabla u) + au = \lambda^2 m u.$$

It also solves systems of equations of the form

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

and eigenvalue systems of the form

$$-\nabla\cdot(\mathbf{c}\otimes\nabla\mathbf{u}) + \mathbf{au} = \lambda\mathbf{du}$$
or
$$-\nabla\cdot(\mathbf{c}\otimes\nabla\mathbf{u}) + \mathbf{au} = \lambda^2\mathbf{mu}.$$

The coefficients m, d, c, a, and f can be functions of location (x, y, and, in 3-D, z), and, except for eigenvalue problems, they also can be functions of the solution u or its gradient. For eigenvalue problems, the coefficients cannot depend on the solution u or its gradient.

For scalar equations, all the coefficients except c are scalar. The coefficient c represents a 2-by-2 matrix in 2-D geometry, or a 3-by-3 matrix in 3-D geometry. For systems of N

equations, the coefficients **m**, **d**, and **a** are *N*-by-*N* matrices, **f** is an *N*-by-1 vector, and **c** is a 2*N*-by-2*N* tensor (2-D geometry) or a 3*N*-by-3*N* tensor (3-D geometry). For the meaning of $\mathbf{c} \otimes \mathbf{u}$, see "c Coefficient for `specifyCoefficients`" on page 2-104.

When both *m* and *d* are 0, the PDE is stationary. When either *m* or *d* are nonzero, the problem is time-dependent. When any coefficient depends on the solution *u* or its gradient, the problem is called nonlinear.

Related Examples

- "Solve Problems Using PDEModel Objects" on page 2-14
- "f Coefficient for `specifyCoefficients`" on page 2-101
- "c Coefficient for `specifyCoefficients`" on page 2-104
- "m, d, or a Coefficient for specifyCoefficients" on page 2-143
- "Systems of PDEs" on page 2-65

Common Toolbox Applications

PDEs used for:

- 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
- 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 3-7
- "Structural Mechanics — Plane Strain" on page 3-13
- "Electrostatics" on page 3-32
- "Magnetostatics" on page 3-40
- "AC Power Electromagnetics" on page 3-47
- "Conductive Media DC" on page 3-53
- "Heat Transfer" on page 3-60
- "Diffusion" on page 3-74

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

Solve 2-D PDEs Using the PDE App

Solve 2-D PDE problems using the PDE app by following these steps:

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*. See "Specify Geometry Using a CSG Model" on page 2-5.

2 Define the boundary conditions.

You can have different types of boundary conditions on different boundary segments. See "Classification of Boundary Conditions" on page 2-177.

3 Define the PDE coefficients. See "Coefficients for Systems of PDEs" on page 2-93.

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.

For examples, see "Poisson's Equation with Complex 2-D Geometry" on page 1-12, "Solve Poisson's Equation on a Unit Disk" on page 3-75, "Conductive Media DC" on page 3-53, or "Minimal Surface Problem" on page 3-86.

Visualize and Animate Solutions

Partial Differential Equation Toolbox has many plot options, such as surface, mesh, contour, and arrow (quiver) plots, hidden or exposed meshes, colormaps, and so on. For command line plotting, see "Plot 2-D Solutions and Their Gradients" on page 3-135 and "Plot 3-D Solutions and Their Gradients" on page 3-145.

For 2-D PDEs, you also can use the PDE app. The app lets you plot three different solution properties at the same time, using color, height, and vector field plots. You can create plots in the app itself or in separate figures. For details, see "Plot the Solution in the PDE App" on page 4-17.

For parabolic and hyperbolic equations, you can create an animated movie of the solution. See "Wave Equation" on page 3-104.

Poisson's Equation with Complex 2-D Geometry

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

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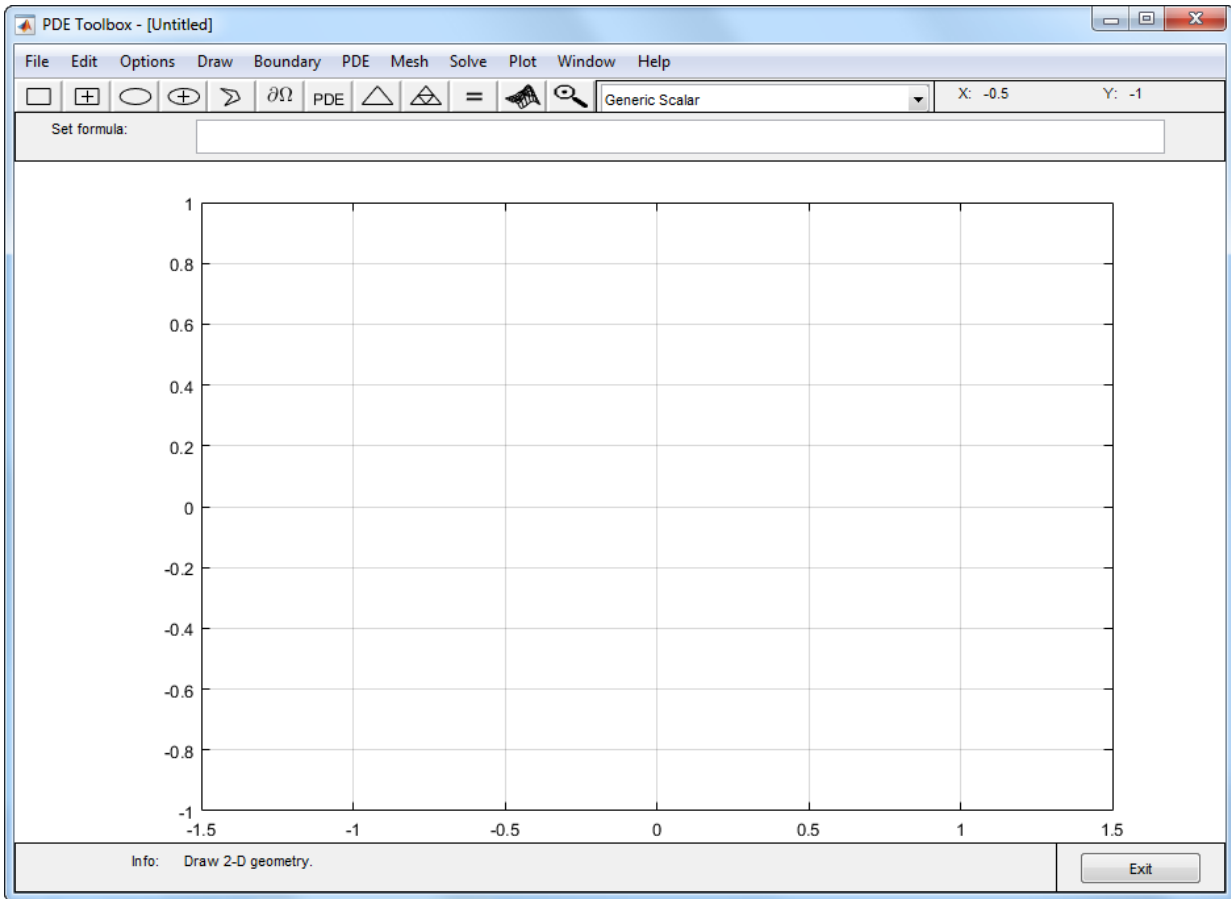
The first step is to draw the geometry on which you want to solve the PDE. The PDE app provides four basic types of *solid objects*: polygons, rectangles, circles, and ellipses. The objects are used to create a *Constructive Solid Geometry model* (CSG model). Each solid object is assigned a unique label,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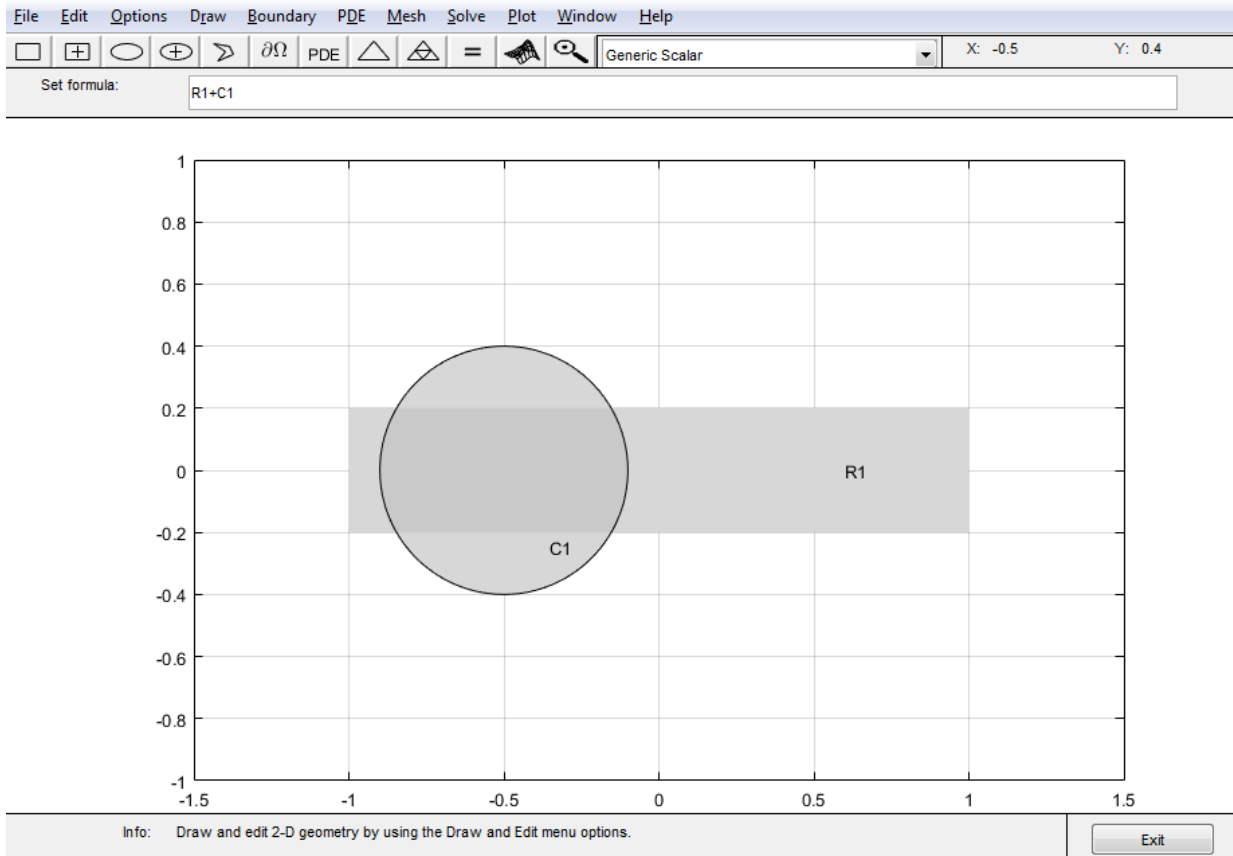
The resulting CSG model is the union of the rectangle R1 and the circle C1, described by set algebra as R1+C1. The area where the two objects overlap is clearly visible as it is drawn using a darker shade of gray. The object that you just drew—the circle—has a black border, indicating that it is selected.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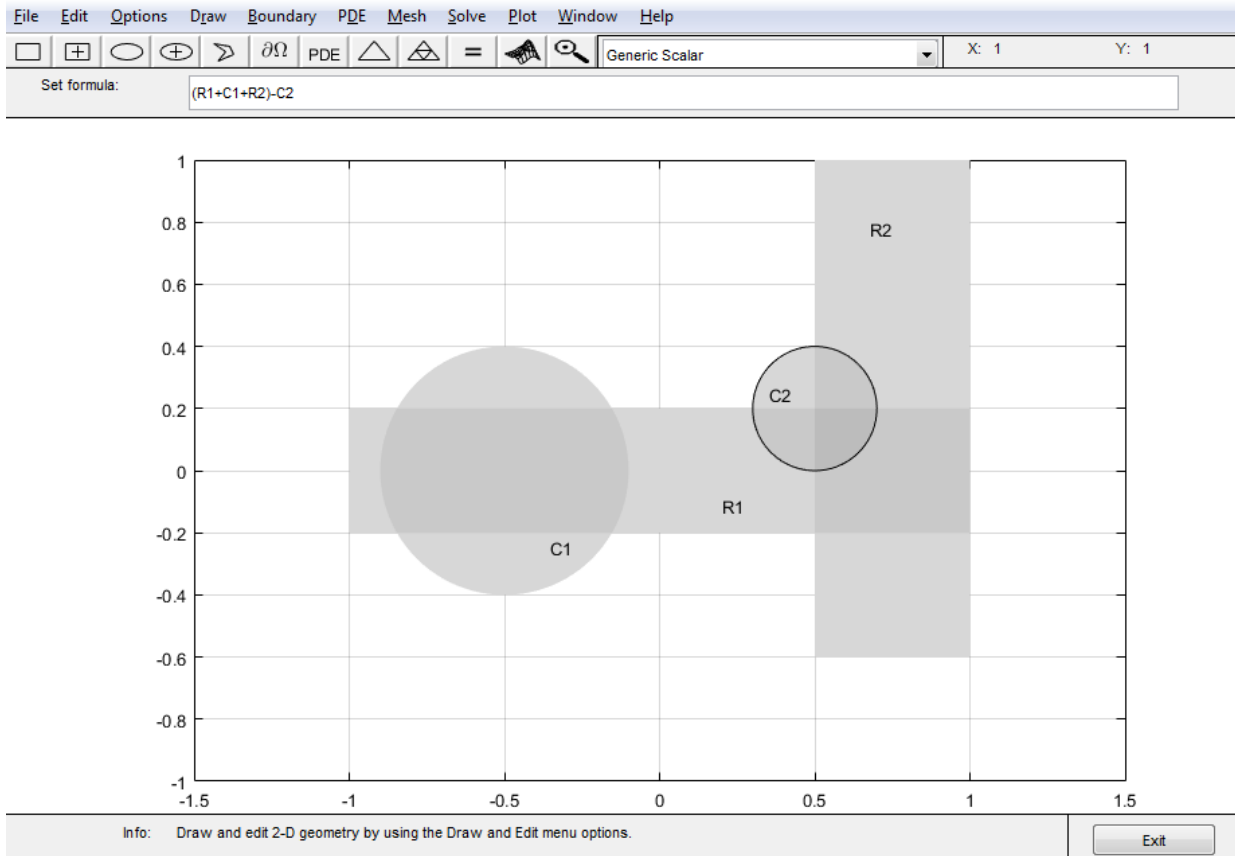
If you want, you can save this CSG model as a file. Use the **Save As** option from the **File** menu, and enter a filename of your choice. It is good practice to continue to save your model at regular intervals using **Save**. All the additional steps in the process of modeling and solving your PDE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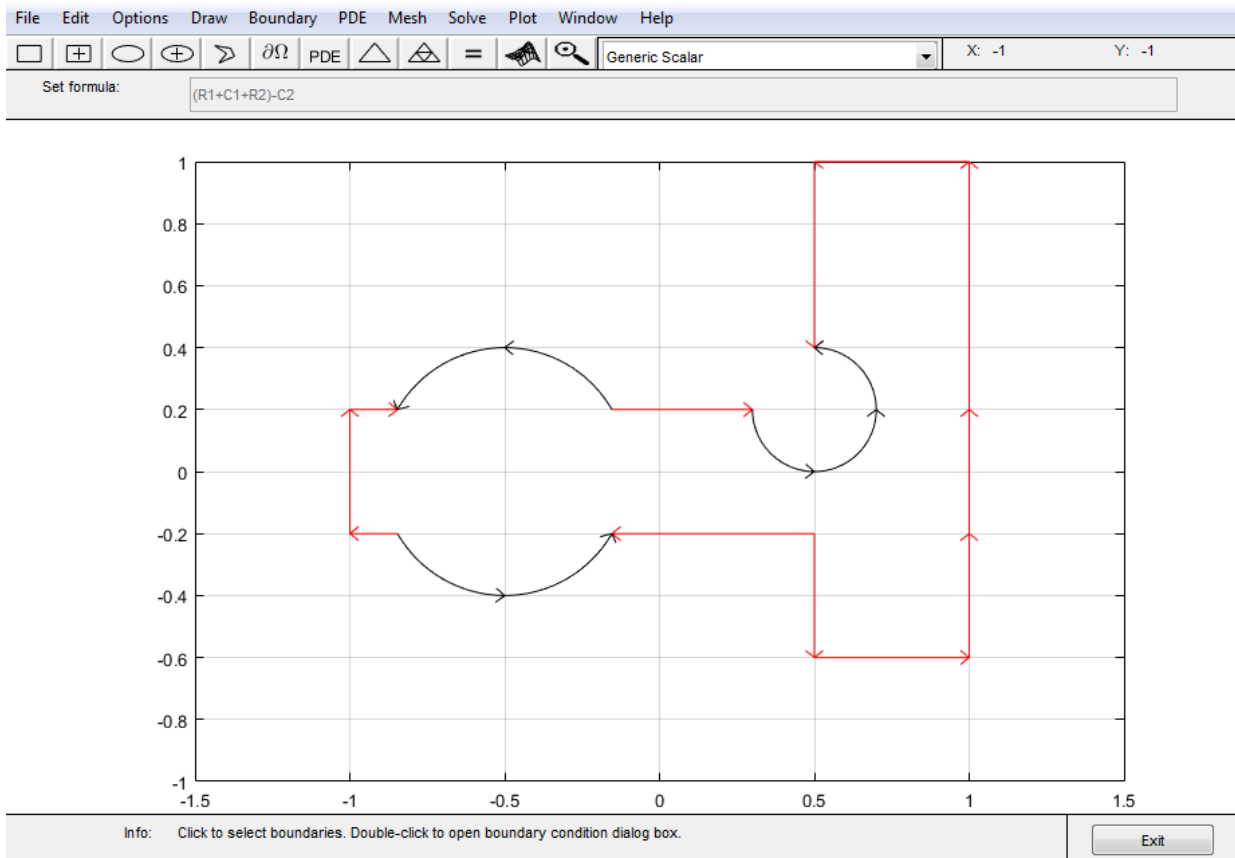
Double-clicking anywhere on the selected boundary segments opens the Boundary Condition dialog box. Here, you select the type of boundary condition, and enter the boundary condition as a MATLAB expression. Change the boundary condition along the selected boundaries to a Neumann condition, $\partial u/\partial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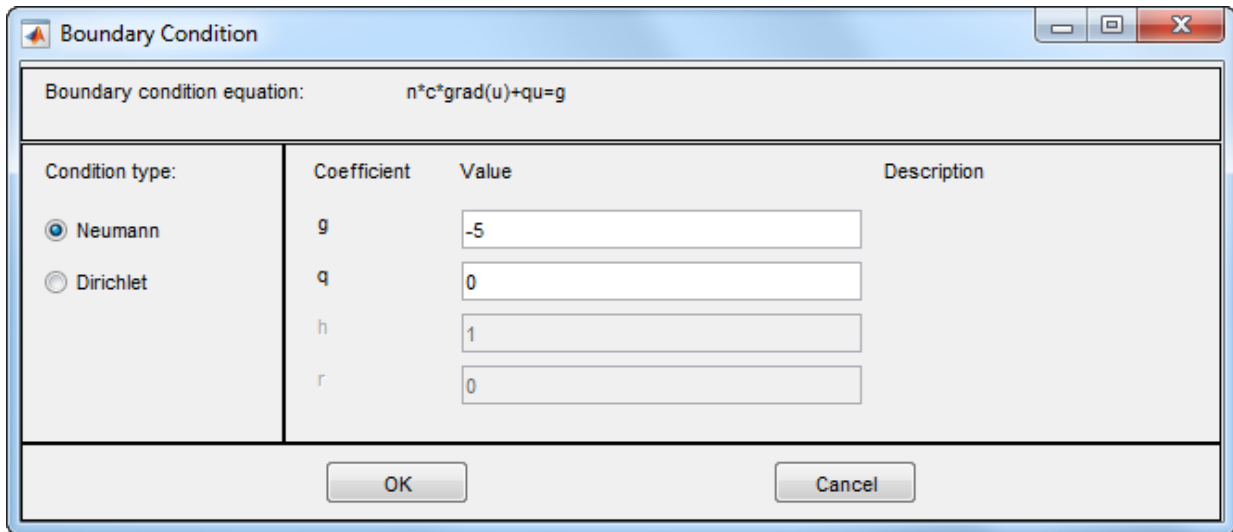
Next, specify the PDE itself through a dialog box that is accessed by clicking the button with the **PDE** icon or by selecting **PDE Specification** from the **PDE** menu. In PDE mode, you can also access the PDE Specification dialog box by double-clicking a subdomain. That way, different subdomains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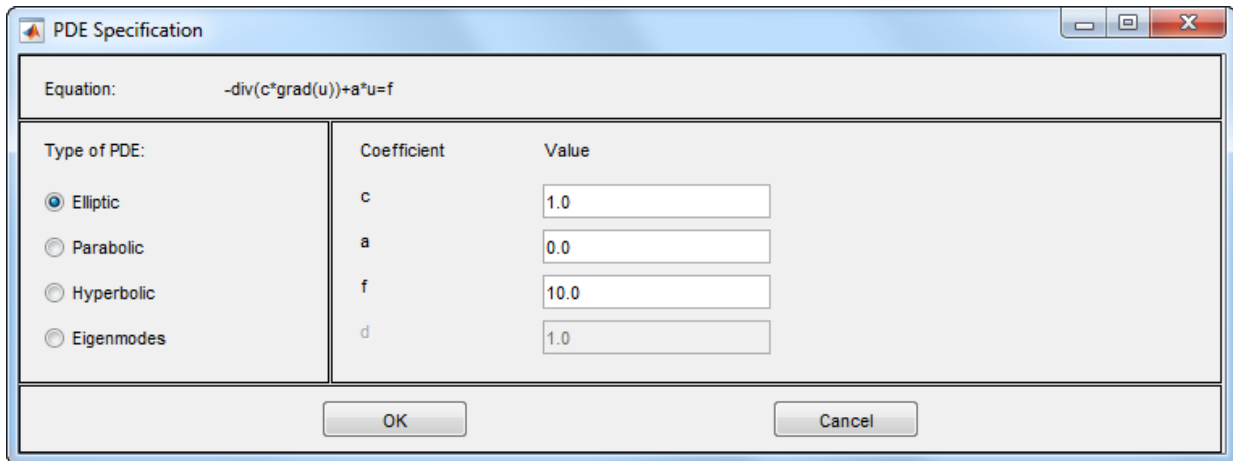
Finally, create the triangular mesh that Partial Differential Equation Toolbox software uses in the Finite Element Method (FEM) to solve the PDE. The triangular mesh is

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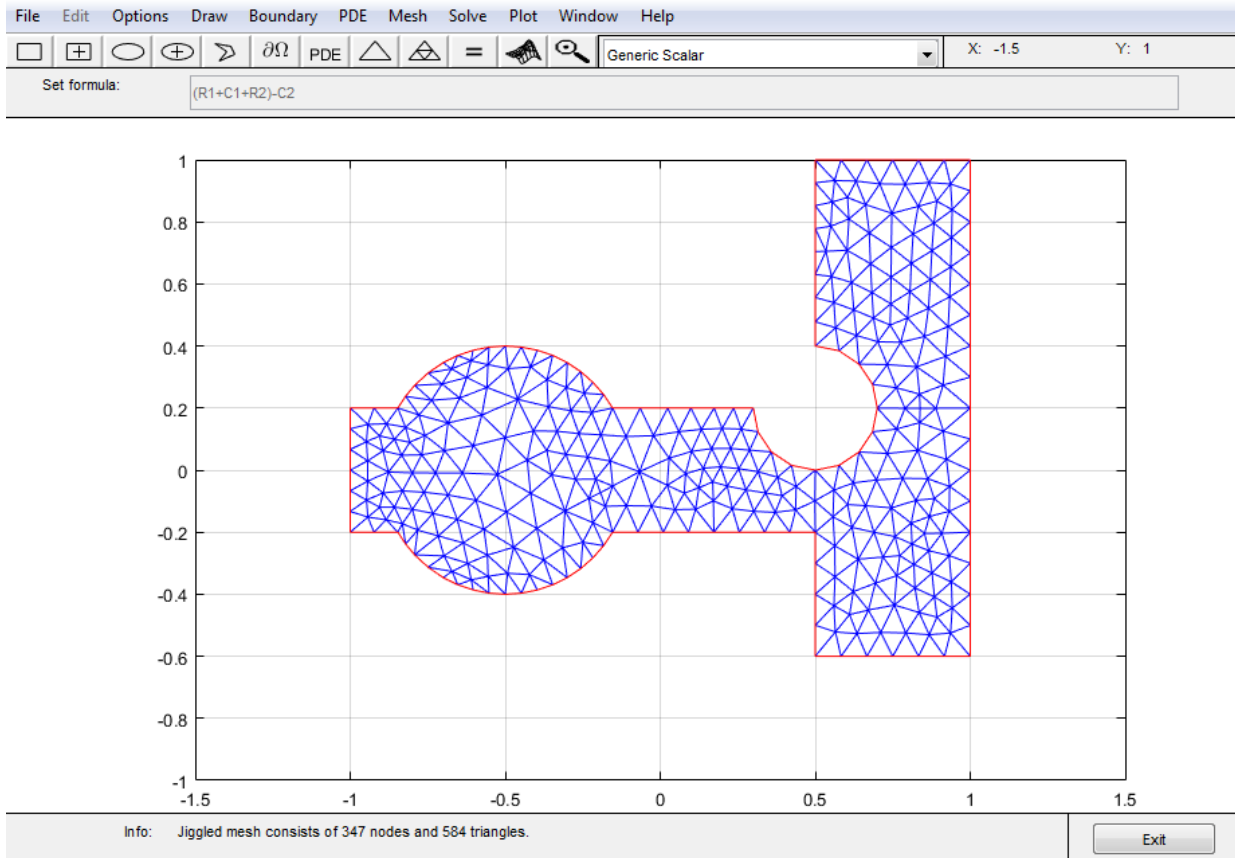
We are now ready to solve the problem. Click the **=** button or select **Solve PDE** from the **Solve** menu to solve the PDE. The solution is then plotted. By default, the plot uses interpolated coloring and a linear color map. A color bar is also provided to map the different shades to the numerical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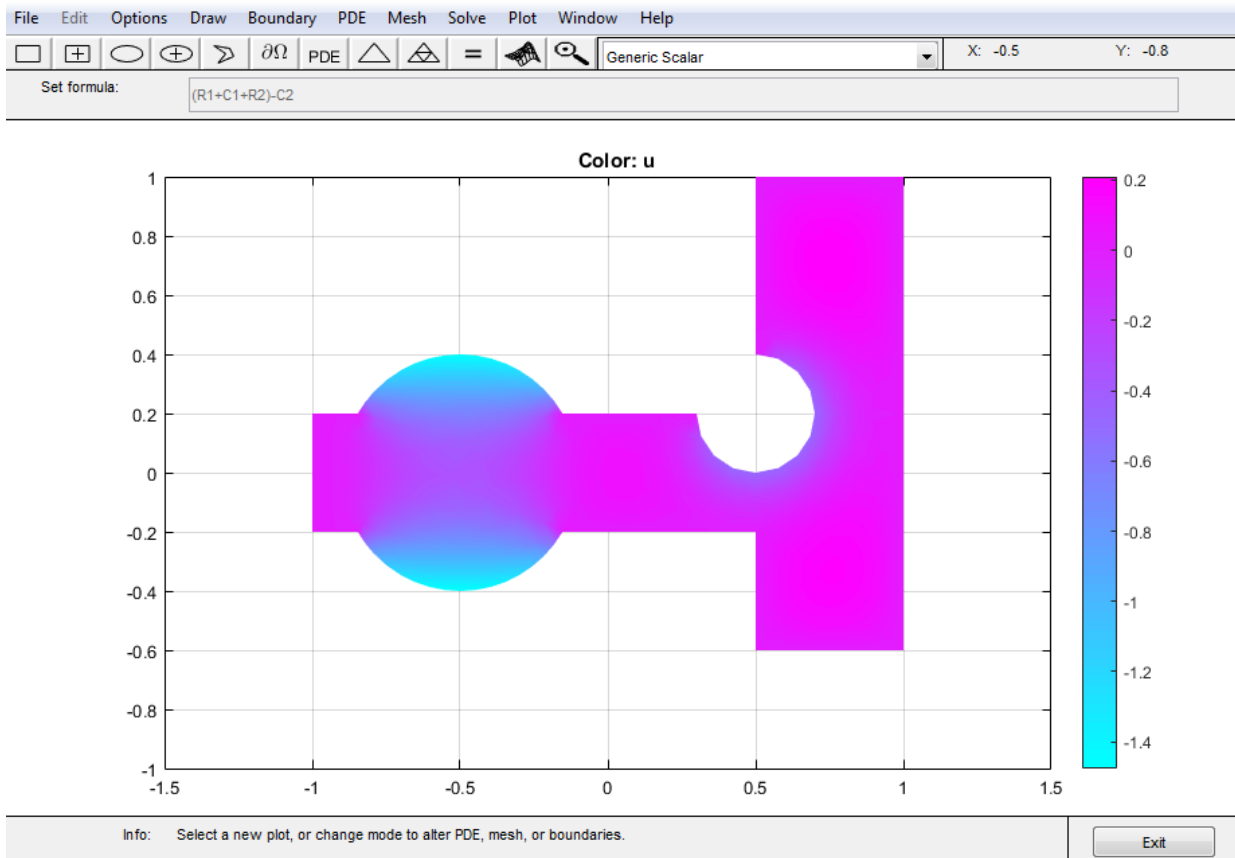
There are many more plot modes available to help you visualize the solution. Click the button with the 3-D solution icon or select **Parameters** from the **Plot** menu to access the dialog box for selection of the different plot options. Several plot styles are available, and the solution can be plotted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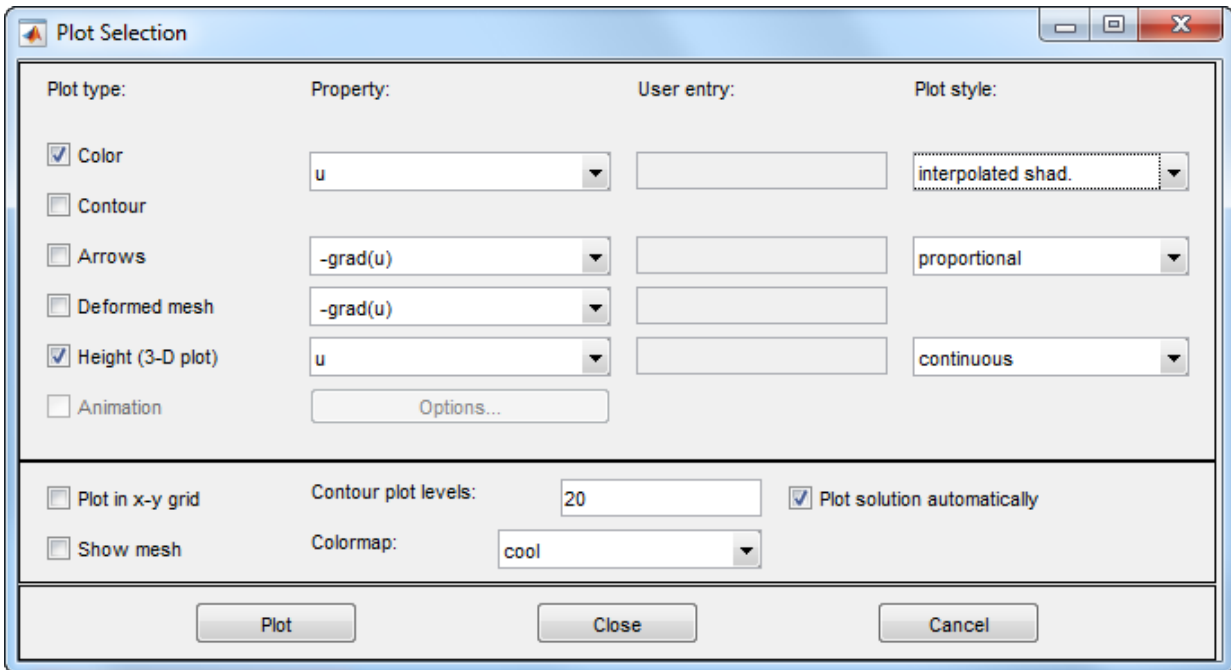
Now, select a plot where the color and the height both represent u. Choose interpolated shading and use the continuous (interpolated) height option. The default colormap is the `cool` colormap; a pop-up menu lets you select from a number of different colormaps. Finally, click the **Plot** button to plot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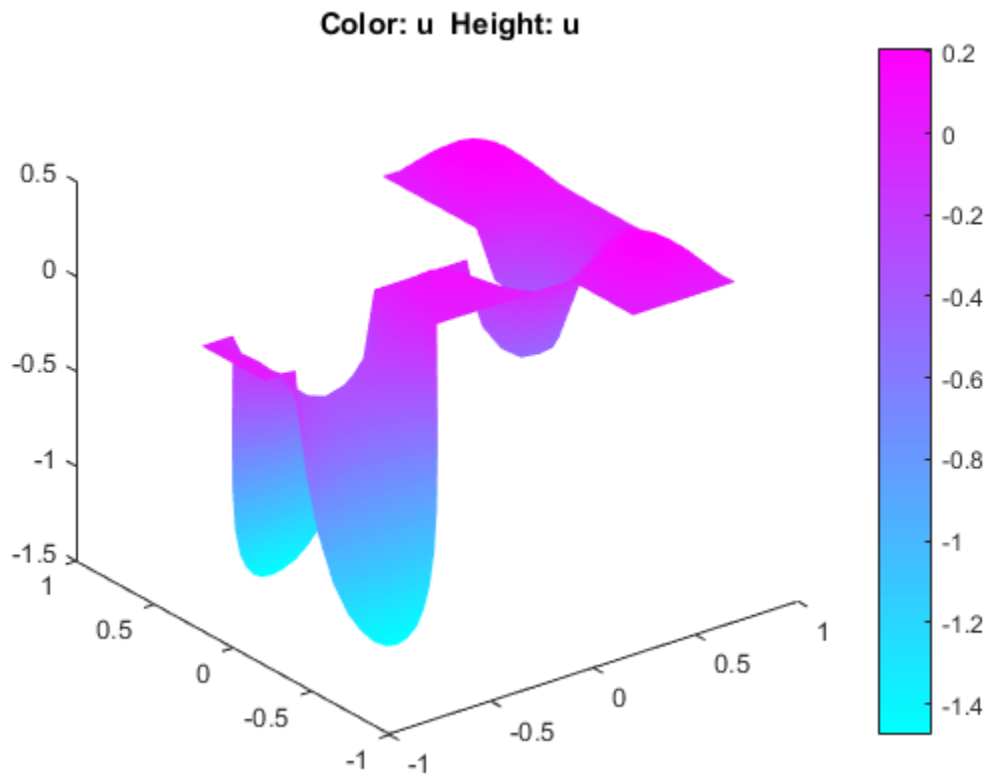
PDE App Shortcuts

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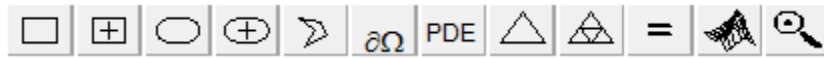
Five buttons on the left let you draw the geometry. Double-click a button makes it "stick," and you can then continue to draw solid objects of the selected type until you single-click the button to "release" it.

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:

$\partial\Omega$	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 PDE app 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.

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 PDE app 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 5-13, "Parabolic Equations" on page 5-17, "Hyperbolic Equations" on page 5-20, "Eigenvalue Equations" on page 5-22, and "Nonlinear Equations" on page 5-26.

You start by approximating the computational domain Ω with a union of simple geometric objects, in this case triangles (2-D geometry) or tetrahedra (3-D geometry). (This discussion applies to both triangles and tetrahedra, but speaks of triangles.) The triangles form a mesh and each vertex is called a node. You are in the situation of an architect designing a dome. The architect has to strike a balance between the ideal rounded forms of the original sketch and the limitations of the simple building-blocks, triangles or quadrilaterals. If the result does not look close enough to a perfect dome, the architect can always improve the result by 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.
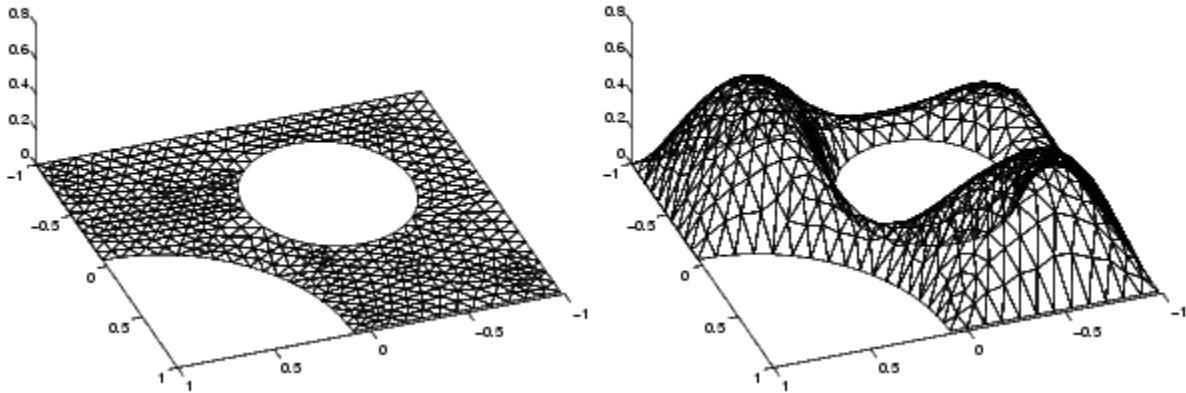
A Triangular Mesh (left) and a Continuous Piecewise Linear Function on That Mesh

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

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

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

Any continuous piecewise linear u_h is represented as a combination

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

where ϕ_i are some special piecewise linear basis functions and U_i are scalar coefficients. Choose ϕ_i like a tent, such that it has the "height" 1 at the node i and the height 0 at all other nodes. For any fixed v, the FEM formulation yields an algebraic equation in the unknowns U_i. You want to determine N unknowns, so you need N different instances of v. What better candidates than $v = \phi_i$, $i = 1, 2, \dots , N$? 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 PDE app, 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 PDE app 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

Open the PDE App

For basic information on 2-D geometry construction, see "Create 2-D Geometry" on page 2-17

Partial Differential Equation Toolbox software includes the PDE app, 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 the PDE app during a MATLAB session. The following figure shows the PDE app as it looks when you start it.
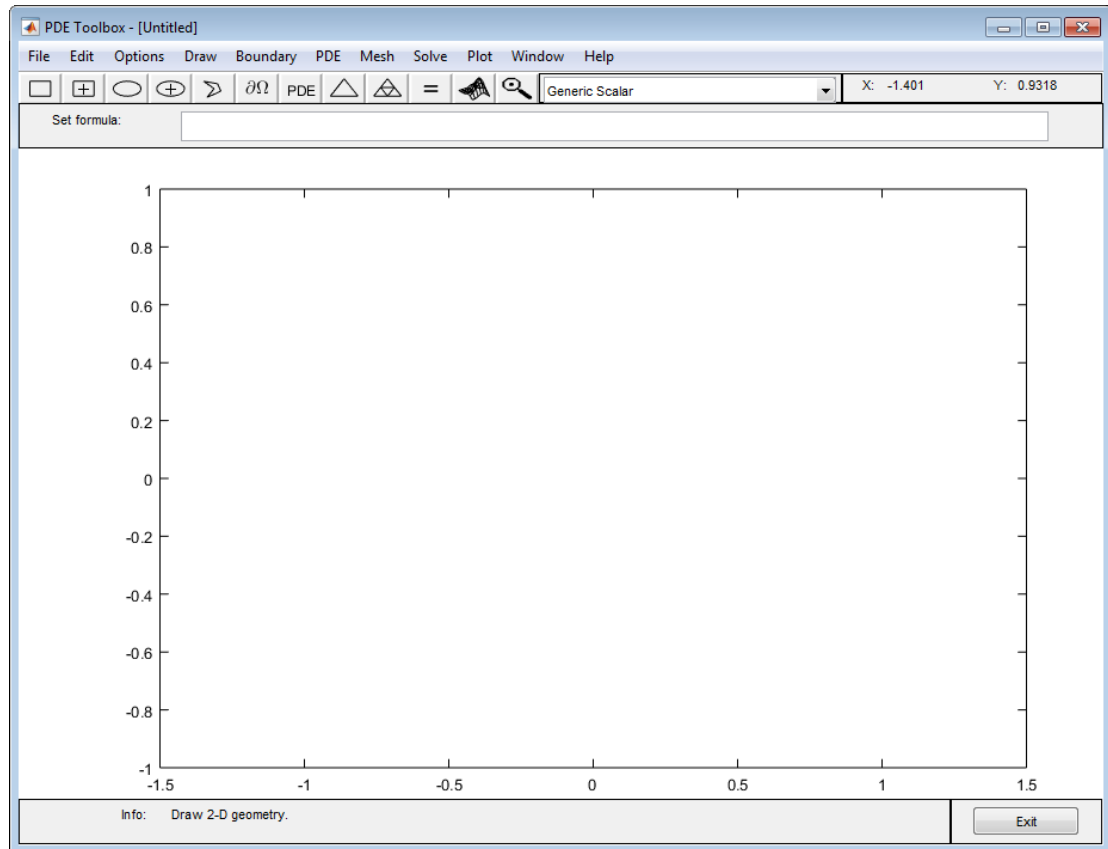
At the top, the PDE app has a pull-down menu bar that you use to control the modeling. Below the menu bar, a toolbar with icon buttons provide quick and easy access to some of the most important functions.

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 PDE app 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 PDE app.

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 PDE app, 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

For basic information on 2-D geometry construction, see "Create 2-D Geometry" on page 2-17

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 PDE app, 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.
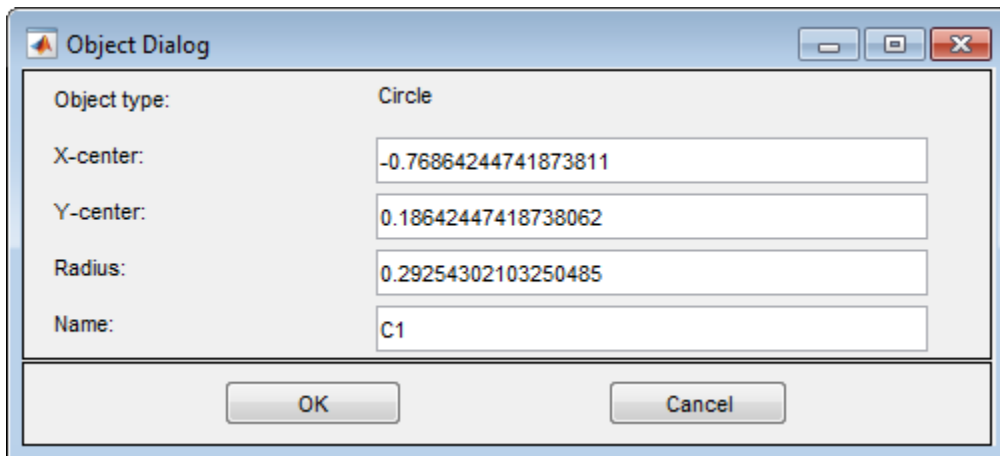
You can use the name of the object to refer to the corresponding set of points in a set formula. The operators +, *, and – are used to form the set of points Ω in the plane over which the differential equation is solved. The operators +, the set union operator, and *, the set intersection operator, have 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 PDE app, 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 PDE app 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.
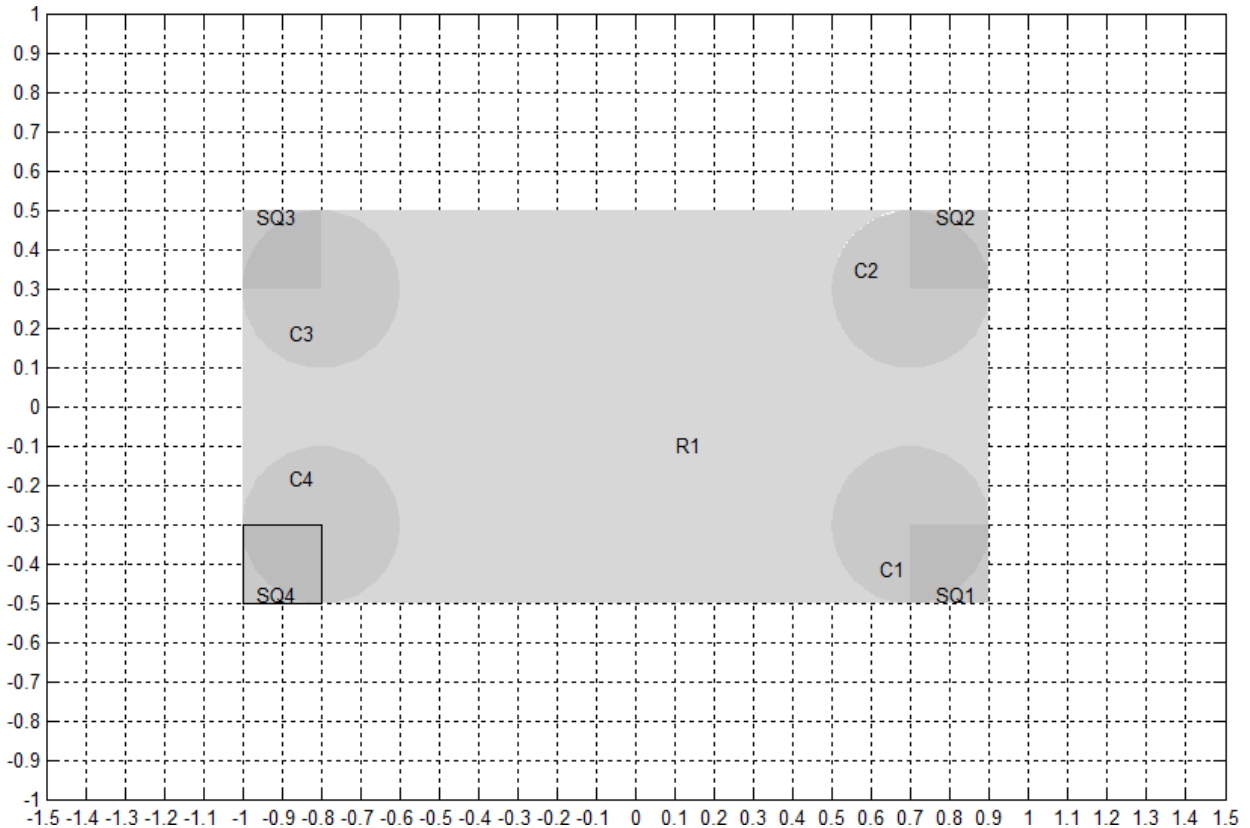
Edit the set formula. To get rounded corners, subtract the small squares from the rectangle and then add the circles. As a set formula, express this as

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

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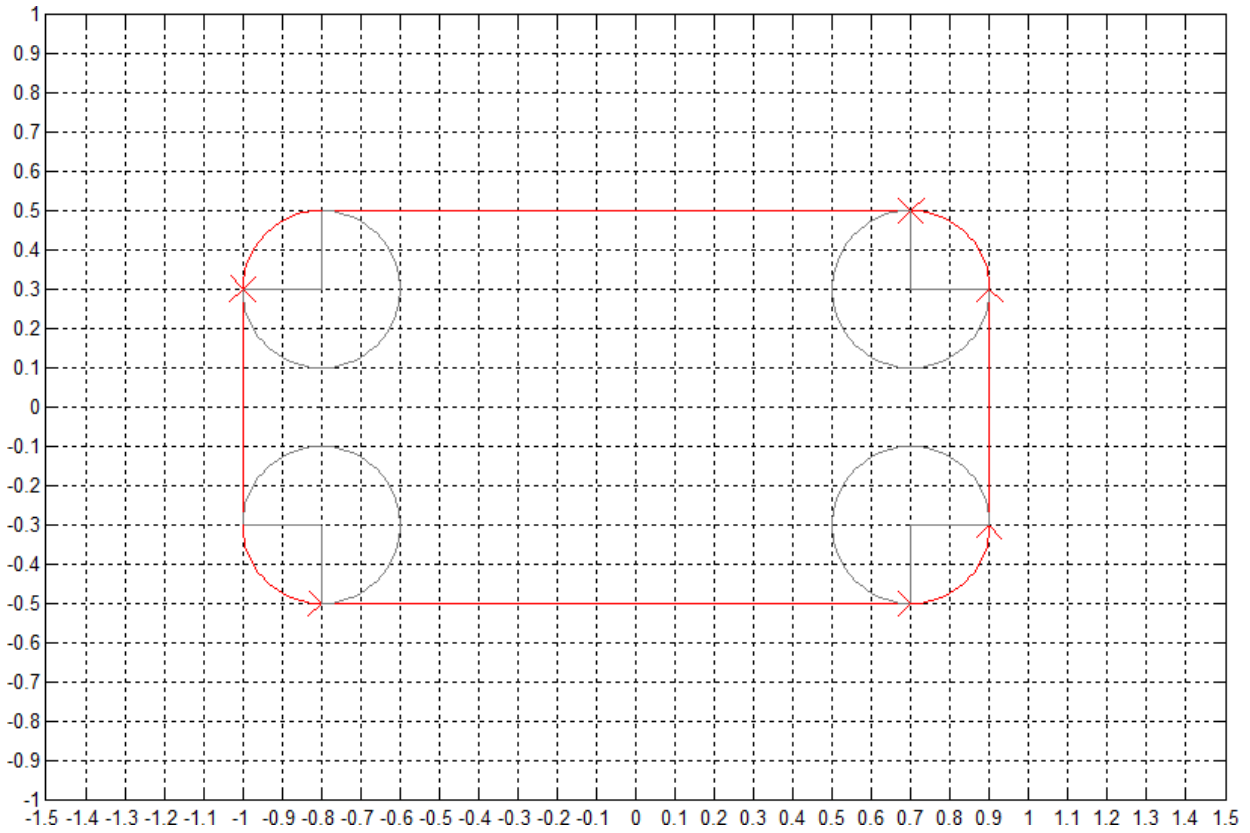
Because of the intersection of the solid objects used in the initial CSG model, a number of subdomain borders remain. They are drawn using gray lines. If this is a model of, e.g., a homogeneous plate, you can remove these borders. Select the **Remove All Subdomain Borders** option from the **Boundary**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

Solve Problems Using Legacy PDEModel Objects

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "Solve Problems Using PDEModel Objects" on page 2-14.

1   Put your problem in the correct form for Partial Differential Equation Toolbox solvers. For details, see "Equations You Can Solve Using Legacy Functions" on page 1-3. If you need to convert your problem to divergence form, see "Put Equations in Divergence Form" on page 2-62.

2   Create a `PDEModel` model container. For scalar PDEs, use `createpde` with no arguments.

```
model = createpde;
```

If *N* is the number of equations in your system, use `createpde` with input argument N.

```
model = createpde(N);
```

3   Import the geometry into `model`. For details, see "Create and View 3-D Geometry" on page 2-47 or "Create 2-D Geometry" on page 2-17. For example:

```
importGeometry(model,'geometry.stl'); % importGeometry for 3-D
geometryFromEdges(model,g); % geometryFromEdges for 2-D
```

4   View the geometry so that you know the labels of the faces. To see labels of a 3-D model, you might need to rotate the model, or make it transparent, or zoom in on it. See "Create and View 3-D Geometry" on page 2-47. For a 2-D example, see "Identify Boundary Labels" on page 2-168. For example:

```
pdegplot(model,'FaceLabels','on') % 'FaceLabels' for 3-D
pdegplot(model,'EdgeLabels','on') % 'EdgeLabels' for 2-D
```

5   Create the boundary conditions. For details, see "Specify Boundary Conditions Objects" on page 2-179. For example:

```
applyBoundaryCondition(model,'face',[2,3,5],'u',[0,0]); % 'face' for 3-D
applyBoundaryCondition(model,'edge',[1,4],'g',1,'q',eye(2)); % 'edge' for 2-D
```

For more information on boundary conditions, see "Boundary Conditions", specifically:

- To view the forms of boundary conditions, see "Classification of Boundary Conditions" on page 2-177.
- To specify constant boundary conditions, see "Specify Constant Boundary Conditions" on page 2-181 and "Solve PDEs with Constant Boundary Conditions" on page 2-185.
- To specify nonconstant boundary conditions, see "Specify Nonconstant Boundary Conditions" on page 2-190 and "Solve PDEs with Nonconstant Boundary Conditions" on page 2-193.

6 Create the PDE coefficients. For example:

```
f = [1;2];
a = 0;
c = [1;3;5];
```

- You can specify coefficients as numeric, string functions, or functions in 2-D functional form or 3-D functional form. For a 2-D example, see "Solve PDE with Coefficients in Functional Form" on page 2-79.
- For systems of PDEs, each coefficient `f`, `c`, `a`, and `d` has a specific format. See "f Coefficient for Systems" on page 2-98, "c Coefficient for Systems" on page 2-125, and "a or d Coefficient for Systems" on page 2-148.

For all information on coefficients, see "PDE Coefficients".

7 For hyperbolic or parabolic equations, create an initial condition. For nonlinear elliptic problems, create an initial guess. See "Solve PDEs with Initial Conditions" on page 2-162.

8 Create the mesh. To obtain a nondefault mesh, use `generateMesh` name-value pairs. For example:

```
generateMesh(model);
```

9 Call the appropriate solver. For example:

```
u = assempde(model,c,a,f);
```

- For elliptic problems whose coefficients do not depend on the solution u, use `assempde`.
- For elliptic problems whose coefficients depend on the solution u, use `pdenonlin`.
- For parabolic problems, use `parabolic`.
- For hyperbolic problems, use `hyperbolic`.

- For eigenvalue problems, use `pdeeig`.

For definitions of the problems that these solvers address, see "Equations You Can Solve Using Legacy Functions" on page 1-3.

10 Examine the solution. See "Plot 3-D Solutions and Their Gradients" on page 3-145 or `pdeplot`.

See Also

applyBoundaryCondition | createpde | generateMesh | geometryFromEdges | importGeometry | pdegplot | pdeplot | pdeplot3D

Solve Problems Using PDEModel Objects

Note: **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see "Solve Problems Using Legacy PDEModel Objects" on page 2-11.

---

1 Put your problem in the correct form for Partial Differential Equation Toolbox solvers. For details, see "Equations You Can Solve Using Recommended Functions" on page 1-6. If you need to convert your problem to divergence form, see "Put Equations in Divergence Form" on page 2-62.

2 Create a `PDEModel` model container. For scalar PDEs, use `createpde` with no arguments.

```
model = createpde();
```

If *N* is the number of equations in your system, use `createpde` with input argument N.

```
model = createpde(N);
```

3 Import the geometry into `model`. For details, see "Create and View 3-D Geometry" on page 2-47 or "Create 2-D Geometry" on page 2-17. For example:

```
importGeometry(model,'geometry.stl'); % importGeometry for 3-D
geometryFromEdges(model,g); % geometryFromEdges for 2-D
```

4 View the geometry so that you know the labels of the boundaries. To see labels of a 3-D model, you might need to rotate the model, or make it transparent, or zoom in on it. See "Create and View 3-D Geometry" on page 2-47. For a 2-D example, see "Identify Boundary Labels" on page 2-168. For example:

```
pdegplot(model,'FaceLabels','on') % 'FaceLabels' for 3-D
pdegplot(model,'EdgeLabels','on') % 'EdgeLabels' for 2-D
```

5 Create the boundary conditions. For details, see "Specify Boundary Conditions Objects" on page 2-179. For example:

```
applyBoundaryCondition(model,'face',[2,3,5],'u',[0,0]); % 'face' for 3-D
applyBoundaryCondition(model,'edge',[1,4],'g',1,'q',eye(2)); % 'edge' for 2-D
```

For more information on boundary conditions, see "Boundary Conditions", specifically:

- To view the forms of boundary conditions, see "Classification of Boundary Conditions" on page 2-177.

- To specify constant boundary conditions, see "Specify Constant Boundary Conditions" on page 2-181 and "Solve PDEs with Constant Boundary Conditions" on page 2-185.

- To specify nonconstant boundary conditions, see "Specify Nonconstant Boundary Conditions" on page 2-190 and "Solve PDEs with Nonconstant Boundary Conditions" on page 2-193.

6   Create the PDE coefficients. For example:

```
f = [1;2];
a = 0;
c = [1;3;5];
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);
```

- You can specify coefficients as numeric or as functions.

- Each coefficient m, d, c, a, and f, has a specific format. See "f Coefficient for specifyCoefficients" on page 2-101, "c Coefficient for specifyCoefficients" on page 2-104, and "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

For all information on coefficients, see "PDE Coefficients".

7   For time-dependent equations, or optionally for nonlinear stationary equations, create an initial condition. See "Set Initial Conditions" on page 2-155.

8   Create the mesh. For example:

```
generateMesh(model);
```

9   Call the appropriate solver. For all problems except for eigenvalue problems, call solvepde:

```
result = solvepde(model); % for stationary problems
result = solvepde(model,tlist); % for time-dependent problems
```

For eigenvalue problems, use solvepdeeig:

```
result = solvepdeeig(model);
```

10   Examine the solution. See "Plot 3-D Solutions and Their Gradients" on page 3-145 or pdeplot.

See Also
applyBoundaryCondition | createpde | generateMesh | geometryFromEdges |
importGeometry | pdegplot | pdeplot | pdeplot3D

Related Examples
· "Plot 3-D Solutions and Their Gradients" on page 3-145

Create 2-D Geometry

Note: CREATING 2-D GEOMETRIES IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Three Ways to Create 2-D Geometry

There are three ways to create 2-D geometry. Two are based on CSG (Constructive Solid Geometry) models, which combine basic shapes.

- Use the PDE app to draw basic shapes (rectangles, circles, ellipses, and polygons) and combine them with set intersection and unions to obtain the final geometry. You can then export the geometry to your MATLAB workspace, or continue to work in the app. For details, see "Open the PDE App" on page 2-3, "Specify Geometry Using a CSG Model" on page 2-5, "Select Graphical Objects Representing Your Geometry" on page 2-7, and "Rounded Corners Using CSG Modeling" on page 2-8.

- Use the `decsg` function to create geometry at the command line as follows:

  - Specify matrices that represent the basic shapes (rectangles, circles, ellipses, and polygons).

  - Give each shape a label.

  - Specify a "set formula" that describes the intersections, unions, and set differences of the basic shapes.

  `decsg` allows you to describe any geometry that you can make from the basic shapes (rectangles, circles, ellipses, and polygons). For details, see "Create CSG Geometry at the Command Line" on page 2-19.

- Specify a function that describes the geometry. The function must be in the form described in "Create Geometry Using a Geometry Function" on page 2-26.

How to Decide on a Geometry Creation Method

This table lists the advantages and disadvantages of each method for creating geometry. In general, choose the lowest-numbered method:

1 Use the PDE app if you can (simple geometry).

2 Use the `decsg` function for geometries that are somewhat complex but can be described in terms of the basic shapes.

3 Use a geometry description function if you cannot use the other methods.

Method	Advantages	Disadvantages
PDE app	Simple click-and-drag interface	Can be tedious to specify exact shapes
	See the geometry as you create it	Can fail for complex figures
	Instant feedback on subdomains, connectedness	No control of edge or subdomain labels
		Only basic shapes as building blocks: rectangles, circles, ellipses, and polygons
`decsg`	Control all basic geometry elements	Cannot see the geometry as you create it
		No control of edge or subdomain labels
		Only basic shapes as building blocks: rectangles, circles, ellipses, and polygons
Geometry function	Specify any shape	Cannot see the geometry as you create it
	Specify edge and subdomain labels	Need to write a function

Create CSG Geometry at the Command Line

> **Note:** CREATING 2-D GEOMETRIES IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Three Elements of Geometry

For basic information on 2-D geometry construction, see "Create 2-D Geometry" on page 2-17

To describe your geometry through Constructive Solid Geometry (CSG) modeling, use three data structures.

1 "Create Basic Shapes" on page 2-19 A matrix whose columns describe the basic shapes. When you export geometry from the PDE app, this matrix has the default name gd (geometry description).

2 "Create Names for the Basic Shapes" on page 2-21 A matrix whose columns contain names for the basic shapes. Pad the columns with zeros or 32 (blanks) so that every column has the same length.

3 "Set Formula" on page 2-22 A string describing the unions, intersections, and set differences of the basic shapes that make the geometry.

Create Basic Shapes

To create basic shapes at the command line, create a matrix whose columns each describe a basic shape. If necessary, add extra zeros to some columns so that all columns have the same length. Write each column using the following encoding.

Circle

Row	Value
1	1 (indicates a circle)
2	x-coordinate of circle center
3	y-coordinate of circle center
4	Radius (strictly positive)

2-19

Polygon

Row	Value
1	2 (indicates a polygon)
2	Number of line segments n
3 through 3+n-1	*x*-coordinate of edge starting points
3+n through 2*n+2	*y*-coordinate of edge starting points

Note: Your polygon cannot contain any self-intersections. To check whether your polygon satisfies this restriction, use the `csgchk` function.

Rectangle

Row	Value
1	3 (indicates a rectangle)
2	4 (Number of line segments)
3 through 6	*x*-coordinate of edge starting points
7 through 10	*y*-coordinate of edge starting points

The encoding of a rectangle is the same as that of a polygon, except that the first row is 3 instead of 2.

Ellipse

Row	Value
1	4 (indicates an ellipse)
2	*x*-coordinate of ellipse center
3	*y*-coordinate of ellipse center
4	First semiaxis length (strictly positive)
5	Second semiaxis length (strictly positive)
6	Angle in radians from *x* axis to first semiaxis

For example, this matrix has a rectangle with a circular end cap and another circular excision:

```matlab
% Create a rectangle and two adjoining circles
rect1 = [3
    4
    -1
    1
    1
    -1
    0
    0
    -0.5
    -0.5];
C1 = [1
    1
    -0.25
    0.25];
C2 = [1
    -1
    -0.25
    0.25];
% Append extra zeros to the circles
% so they have the same number of rows as the rectangle
C1 = [C1;zeros(length(rect1) - length(C1),1)];
C2 = [C2;zeros(length(rect1) - length(C2),1)];
% Combine the shapes into one matrix
gd = [rect1,C1,C2];
```

Create Names for the Basic Shapes

In order to create a formula describing the unions and intersections of basic shapes, you need a name for each basic shape. Give the names as a matrix whose columns contain the names of the corresponding columns in the basic shape matrix. Pad the columns with 0 or 32 if necessary so that each has the same length.

One easy way to create the names is by specifying a character array whose rows contain the names, and then taking the transpose. Use the char function to create the array. char pads the rows as needed so all have the same length. Continuing the example,

```matlab
% Give names for the three shapes
ns = char('rect1','C1','C2');
ns = ns';
```

Set Formula

Obtain the final geometry by writing a string that describes the unions and intersections of basic shapes. Use + for union, * for intersection, - for set difference, and parentheses for grouping. + and * have the same grouping precedence. - has higher grouping precedence.

Continuing the example,

```
% Specify the union of the rectangle and C1, and subtract C2
sf = '(rect1+C1)-C2';
```

Create Geometry and Remove Subdomains

After you have created the basic shapes, given them names, and specified a set formula, create the geometry using decsg. Often, you also remove some or all of the resulting subdomain boundaries. Completing the example,

```
[dl,bt] = decsg(gd,sf,ns); % combines the basic shapes using the set formula
```

View the geometry with and without subdomain removal.

```
pdegplot(dl,'EdgeLabels','on','SubdomainLabels','on')
xlim([-1.5,1.5])
axis equal
```
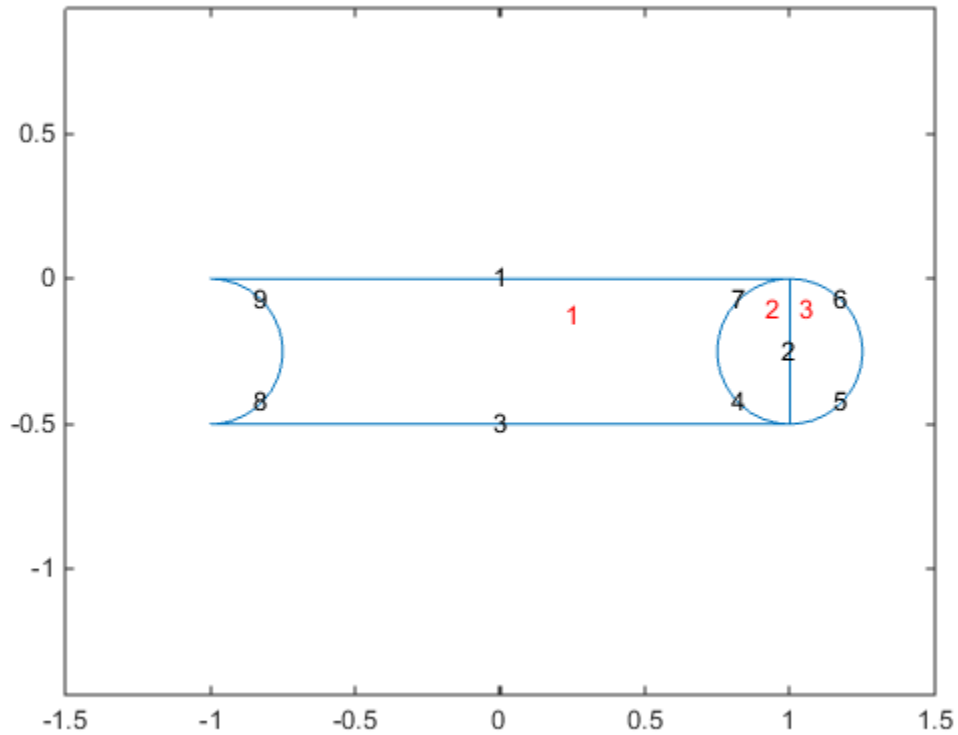
Remove the subdomains.

```
[dl2,bt2] = csgdel(dl,bt); % removes subdomain boundaries
figure
pdegplot(dl2,'EdgeLabels','on','SubdomainLabels','on')
xlim([-1.5,1.5])
axis equal
```
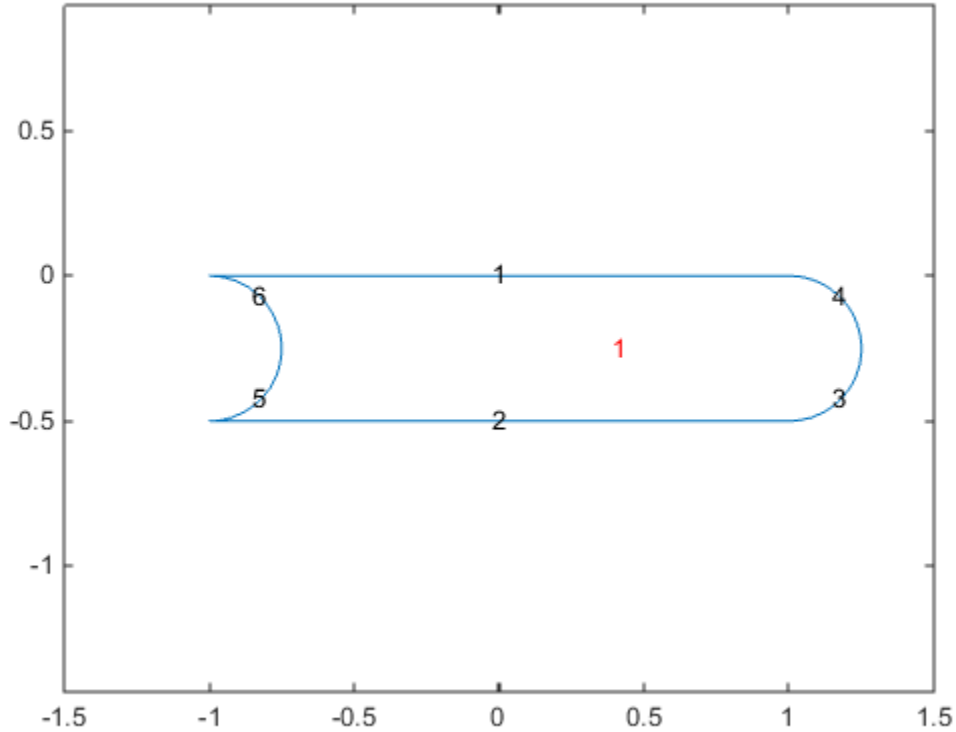
Decomposed Geometry Data Structure

A decomposed geometry matrix has the following encoding. Each column of the matrix corresponds to one boundary segment. Any 0 entry means no encoding is necessary for this row. So, for example, if only line segments appear in the matrix, then the matrix has 7 rows. But if there is also a circular segment, then the matrix has 9 rows. The extra two rows of the line columns are filled with 0.

Row	Circle	Line	Ellipse
1	1	2	4
2	Starting x coordinate	Starting x coordinate	Starting x coordinate

Row	Circle	Line	Ellipse
3	Ending x coordinate	Ending x coordinate	Ending x coordinate
4	Starting y coordinate	Starting y coordinate	Starting y coordinate
5	Ending y coordinate	Ending y coordinate	Ending y coordinate
6	Region label to left of segment, with direction induced by start and end points (0 is exterior label)	Region label to left of segment, with direction induced by start and end points (0 is exterior label)	Region label to left of segment, with direction induced by start and end points (0 is exterior label)
7	Region label to right of segment, with direction induced by start and end points (0 is exterior label)	Region label to right of segment, with direction induced by start and end points (0 is exterior label)	Region label to right of segment, with direction induced by start and end points (0 is exterior label)
8	x coordinate of circle center	0	x coordinate of ellipse center
9	y coordinate of circle center	0	y coordinate of ellipse center
10	0	0	Length of first semiaxis
11	0	0	Length of second semiaxis
12	0	0	Angle in radians between x axis and first semiaxis

Create Geometry Using a Geometry Function

Note: CREATING 2-D GEOMETRIES IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Required Syntax

For basic information on 2-D geometry construction, see "Create 2-D Geometry" on page 2-17

A geometry function describes the curves that bound the geometry regions. A curve is a parametrized function $(x(t),y(t))$. The variable t ranges over a fixed interval. For best results, t should be proportional to arc length plus a constant.

For each region you should have at least two curves. For example, the `'circleg'` geometry function, which ships with the toolbox, uses four curves to describe a circle.

Curves can intersect only at the beginning or end of parameter intervals.

Toolbox functions query your geometry function by passing in 0, 1, or 2 arguments. Conditionalize your geometry function based on the number of input arguments to return the following:

Number of Input Arguments	Returned Data
`0 (ne = pdegeom)`	`ne` is the number of edges in the geometry.
`1 (d = pdegeom(bs))`	`bs` is a vector of edge segments. Your function returns `d` as a matrix with one column for each edge segment specified in `bs`. The rows of `d` are: **1** Start parameter value **2** End parameter value **3** Left region label, where "left" is with respect to the direction from the start to end parameter value **4** Right region label Region label is the same as subdomain number. The region label of the exterior of the geometry is `0`.

Number of Input Arguments	Returned Data
2 (`[x,y] = pdegeom(bs,s)`)	`s` is an array of arc lengths, and `bs` is a scalar or an array the same size as `s` giving edge numbers. If `bs` is a scalar, then it applies to every element in `s`. Your function returns `x` and `y`, which are the `x` and `y` coordinates of the edge segments specified in `bs` at parameter value `s`. The `x` and `y` arrays have the same size as `s`.

Relation Between Parameterization and Region Labels

This figure shows how the direction of parameter increase relates to label numbering. The arrows in the following figure show the directions of increasing parameter values. The black dots indicate curve beginning and end points. The red numbers indicate region labels. The red 0 in the center of the figure indicates that the center square is a hole.

- The arrows by curves 1 and 2 show region 1 to the left and region 0 to the right.
- The arrows by curves 3 and 4 show region 0 to the left and region 1 to the right.
- The arrows by curves 5 and 6 show region 0 to the left and region 1 to the right.
- The arrows by curves 7 and 8 show region 1 to the left and region 0 to the right.
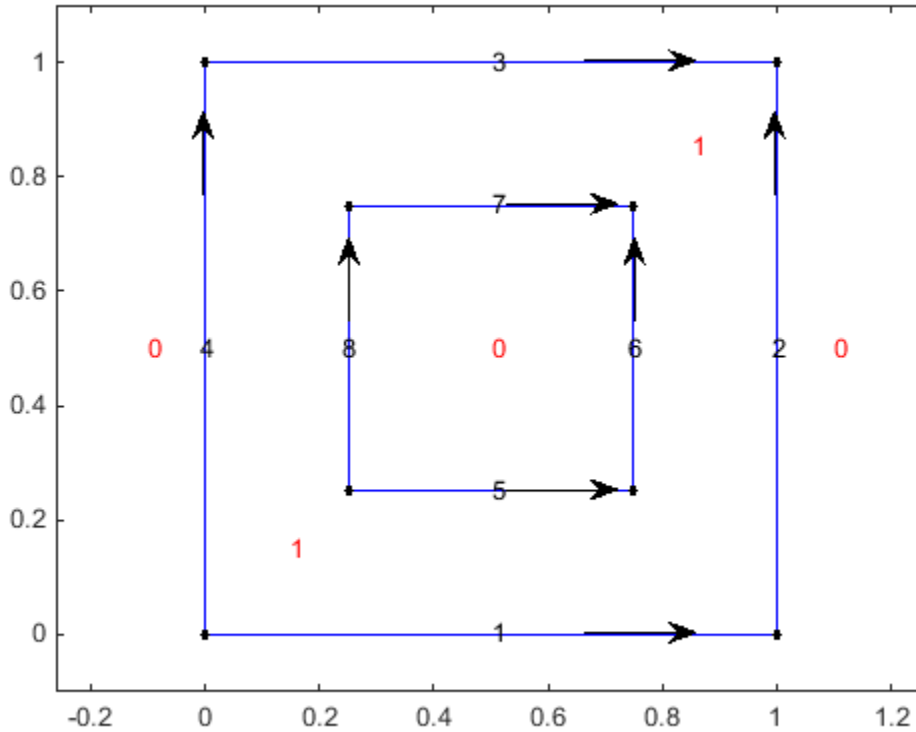
Code for Creating the Figure

```
xs = [0,1,1,0,0];
xt = [0.25,0.75,0.75,0.25,0.25];
ys = [0,0,1,1,0];
yt = [0.25,0.25,0.75,0.75,0.25];
plot(xs,ys,'b-')
hold on
axis equal
ylim([-0.1,1.1])
plot(xt,yt,'b-')
plot(xs,ys,'k.','MarkerSize',12)
plot(xt,yt,'k.','MarkerSize',12)
text(-0.1,0.5,'0','Color','r')
```

```
text(0.5,0.5,'0','Color','r')
text(1.1,0.5,'0','Color','r')
text(0.15,0.15,'1','Color','r')
text(0.85,0.85,'1','Color','r')

text(0.5,0,'1')
text(0.99,0.5,'2')
text(0.5,1,'3')
text(-0.01,0.5,'4')
text(0.5,0.25,'5')
text(0.74,0.5,'6')
text(0.5,0.75,'7')
text(0.24,0.5,'8')

annotation('arrow',[0.6,0.7],[0.18,0.18])
annotation('arrow',[0.6,0.7],[0.86,0.86])
annotation('arrow',[0.26,0.26],[0.7,0.8])
annotation('arrow',[0.77,0.77],[0.7,0.8])

annotation('arrow',[0.53,0.63],[0.35,0.35])
annotation('arrow',[0.53,0.63],[0.69,0.69])
annotation('arrow',[0.39,0.39],[0.55,0.65])
annotation('arrow',[0.645,0.645],[0.55,0.65])
```

Geometry Function for a Circle

This example shows how to use a geometry function to create a circular region. Of course, you could just as easily use a circle basic shape.

You can parametrize a circle with radius 1 centered at the origin (0,0) as follows:

$$x = \cos(t)$$
$$y = \sin(t)$$
$$0 \le t \le 2\pi.$$

A geometry function needs to have at least two segments. So break up the circle into four segments: $0 \le t \le \pi/2$, $\pi/2 \le t \le \pi$, $\pi \le t \le 3\pi/2$, and $3\pi/2 \le t \le 2\pi$.

Now that you have a parametrization, write the geometry function. Save this function file as `circlefunction.m` on your MATLAB path.

```
function [x,y] = circlefunction(bs,s)
```

```
% Create a unit circle centered at (0,0) using four segments.

switch nargin
    case 0
        x = 4; % four edge segments
        return
    case 1
        A = [0,pi/2,pi,3*pi/2; % start parameter values
            pi/2,pi,3*pi/2,2*pi; % end parameter values
            1,1,1,1; % region label to left
            0,0,0,0]; % region label to right
        x = A(:,bs); % return requested columns
        return
    case 2
        x = cos(s);
        y = sin(s);
end
```

This geometry was particularly simple to create because the parameterization did not change depending on the segment number.

Visualize the geometry, edge numbers, and domain label.

```
pdegplot(@circlefunction,'EdgeLabels','on','SubdomainLabels','on')
axis equal
```
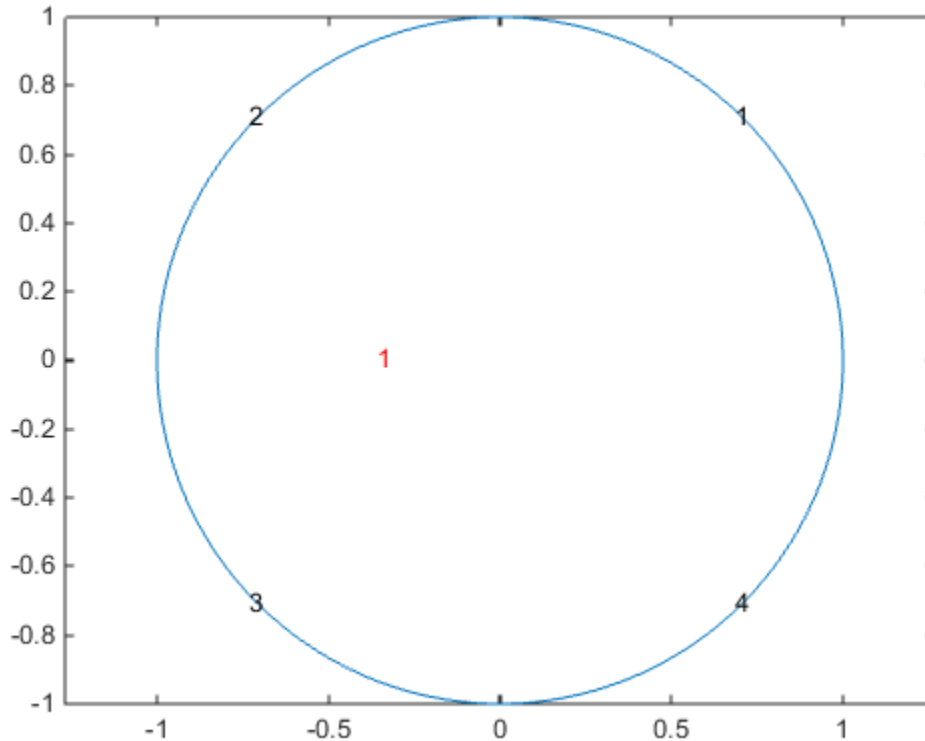
The built-in function `circleg` gives a slightly different parameterization of the circle. You might find it instructive to compare the two approaches.

Arc Length Calculations for a Geometry Function

This example shows how to create a cardioid geometry using four distinct techniques. The techniques are ways to parametrize your geometry using arc length calculations. The cardioid satisfies the equation

$$r = 2(1 + \cos(\Phi)).$$

```
ezpolar('2*(1+cos(Phi))')
```

r = 2 (1+cos(Φ))
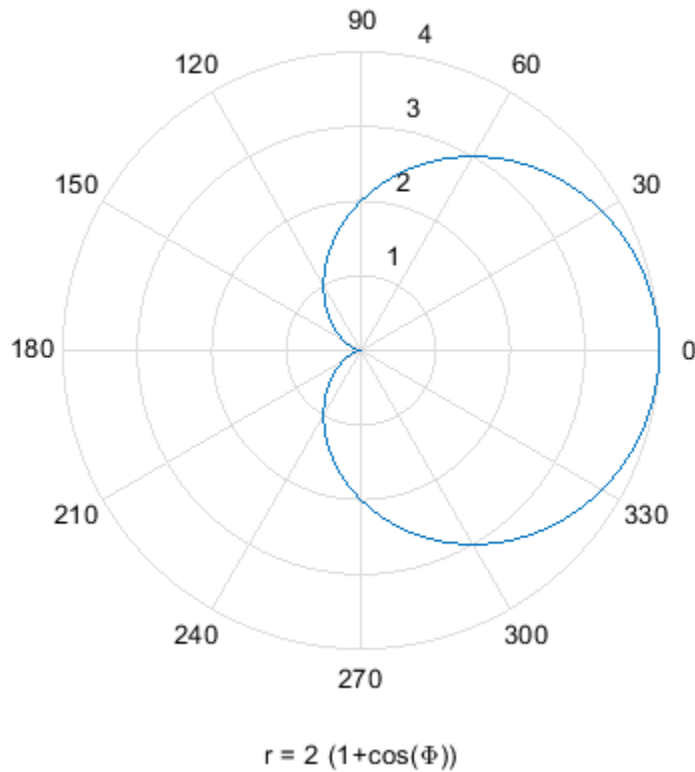
This example shows four approaches to giving a parametrization of the cardioid as a function of arc length.

- Use the `pdearcl` function with a polygonal approximation to the geometry. This approach is general, accurate enough, and computationally fast.

- Use the `integral` and `fzero` functions to compute the arc length. This approach is more computationally costly, but can be accurate without you having to choose an arbitrary polygon.

- Use an analytic calculation of the arc length. This approach is probably the best when it applies, but there are many cases where it does not apply.

- Use a parametrization that is *not* proportional to arc length plus a constant. This approach is simplest, but can yield a distorted mesh that does not give the most accurate solution to your PDE problem.

Polygonal Approximation

The finite element method uses a triangular mesh to approximate the solution to a PDE numerically. So there is no loss in accuracy by taking a sufficiently fine polygonal approximation to the geometry. The `pdearcl` function maps between parametrization and arc length in a form well-suited to a geometry function. Here is a geometry function for the cardioid.

```
function [x,y] = cardioid1(bs,s)
%CARDIOID1 Geometry File defining the geometry of a cardioid.

if nargin == 0
  x = 4; % four segments in boundary
  return
end

if nargin == 1
  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);
  return
end

x = zeros(size(s));
y = zeros(size(s));
if numel(bs) == 1 % bs might need scalar expansion
  bs = bs*ones(size(s)); % expand bs
end

nth = 400; % fine polygon, 100 segments per quadrant
th = linspace(0,2*pi,nth); % parametrization
r = 2*(1+cos(th));
xt = r.*cos(th); % Points for interpolation of arc lengths
yt = r.*sin(th);
% Compute parameters corresponding to the arc length values in s
th = pdearcl(th,[xt;yt],s,0,2*pi); % th contains the parameters
% Now compute x and y for the parameters th
r = 2*(1+cos(th));
```

```
x(:) = r.*cos(th);
y(:) = r.*sin(th);
```

Plot the geometry function.

```
pdegplot('cardioid1','EdgeLabels','on')
axis equal
```
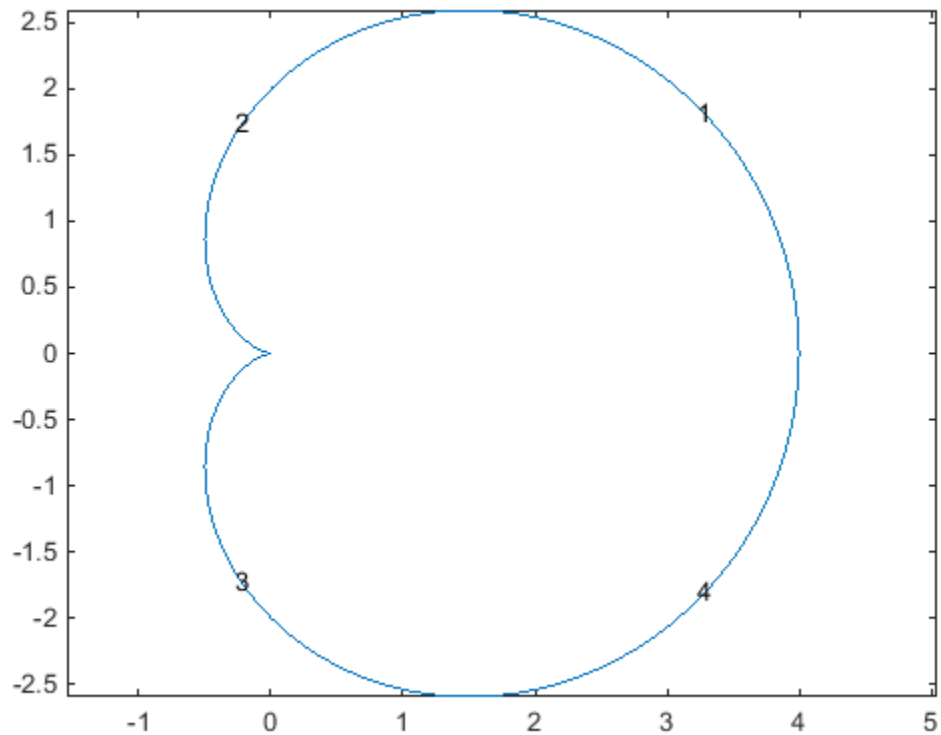

With 400 line segments, the geometry looks smooth.

The built-in `cardg` function gives a slightly different version of this technique.

Integral for Arc Length

You can write an integral for the arc length of a curve. If the parametrization is in terms of $x(u)$ and $y(u)$, then the arclength $s(t)$ is

$$s(t) = \int_{0}^{t} \sqrt{\left(\frac{dx}{du}\right)^{2} + \left(\frac{dy}{du}\right)^{2}} \, du.$$

So for a given value $s0$, you can find t as the root of the equation $s(t) = s0$. The `fzero` function solves this type of nonlinear equation.

For the present example of a cardioid, here is the calculation.

```
function [x,y] = cardioid2(bs,s)
%CARDIOID2 Geometry file defining the geometry of a cardioid.

if nargin == 0
  x = 4; % four segments in boundary
  return
end

if nargin == 1
  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);
  return
end

x = zeros(size(s));
y = zeros(size(s));
if numel(bs) == 1 % bs might need scalar expansion
  bs = bs*ones(size(s)); % expand bs
end

cbs = find(bs < 3); % upper half of cardiod
fun = @(ss)integral(@(t)sqrt(4*(1+cos(t)).^2 + 4*sin(t).^2),0,ss);
sscale = fun(pi);
for ii = cbs(:)' % ensure a row vector
    myfun = @(rr)fun(rr)-s(ii)*sscale/pi;
```

```
    theta = fzero(myfun,0,pi);
    r = 2*(1 + cos(theta));
    x(ii) = r*cos(theta);
    y(ii) = r*sin(theta);
end
cbs = find(bs >= 3); % Lower half of cardioid
s(cbs) = 2*pi - s(cbs);
for ii = cbs(:)'
    theta = fzero(@(rr)fun(rr)-s(ii)*sscale/pi,0,pi);
    r = 2*(1 + cos(theta));
    x(ii) = r*cos(theta);
    y(ii) = -r*sin(theta);
end
```

Plot the geometry function.

```
pdegplot('cardioid1','EdgeLabels','on')
axis equal
```
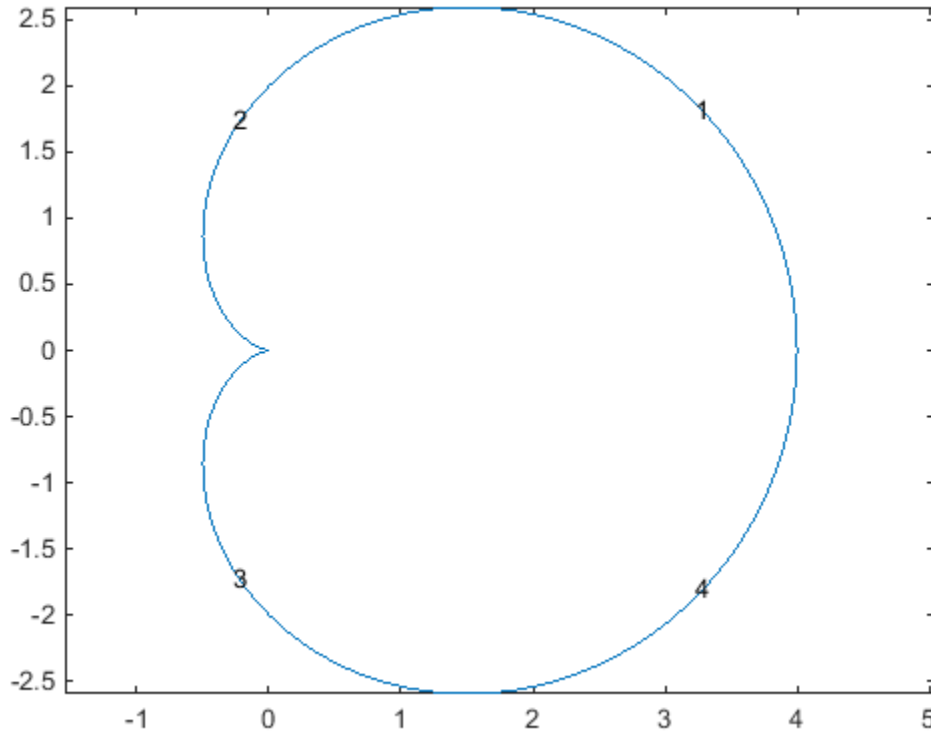
The geometry looks identical to the polygonal approximation. This integral version takes much longer to calculate than the polygonal version.

Analytic Arc Length

If you are handy with integrals, or have Symbolic Math Toolbox™, you can find an analytic expression for the arc length as a function of the parametrization. Then you can give the parametrization in terms of arc length. Here is an approach using Symbolic Math Toolbox.

```
syms t real
r = 2*(1+cos(t));
x = r*cos(t);
```

```
y = r*sin(t);
arcl = simplify(sqrt(diff(x)^2+diff(y)^2));
s = int(arcl,t,0,t,'IgnoreAnalyticConstraints',true)

s =

8*sin(t/2)
```

So you see that, in terms of arclength `s`, the parameter `t` is `t = 2*asin(s/8)` where `s` ranges from `0` to `8`, corresponding to `t` ranging from `0` to `pi`. For `s` between `8` and `16`, by the symmetry of the cardioid, `t = pi + 2*asin((16-s)/8)`.

Furthermore, you can express *x* and *y* in terms of *s* by the following analytic calculations.

```
syms s real
th = 2*asin(s/8);
r = 2*(1+cos(th));
r = expand(r)

r =

4 - s^2/16

x = r*cos(th);
x = simplify(expand(x))

x =

s^4/512 - (3*s^2)/16 + 4

y = r*sin(th);
y = simplify(expand(y))

y =

(s*(64 - s^2)^(3/2))/512
```

Now that you have analytic expressions for *x* and *y* in terms of the arclength *s*, you can write the geometry function.

```
function [x,y] = cardioid3(bs,s)
%CARDIOID3 Geometry file defining the geometry of a cardioid.

if nargin == 0
```

```
  x = 4; % four segments in boundary
  return
end

if nargin == 1
dl = [ 0    4    8   12
       4    8   12   16
       1    1    2    2
       0    0    0    0];
  x = dl(:,bs);
  return
end

x = zeros(size(s));
y = zeros(size(s));
if numel(bs) == 1 % bs might need scalar expansion
  bs = bs*ones(size(s)); % expand bs
end

cbs = find(bs < 3); % upper half of cardiod
x(cbs) = s(cbs).^4/512 - 3*s(cbs).^2/16 + 4;
y(cbs) = s(cbs).*(64 - s(cbs).^2).^(3/2)/512;
cbs = find(bs >= 3); % lower half
s(cbs) = 16 - s(cbs); % take the reflection
x(cbs) = s(cbs).^4/512 - 3*s(cbs).^2/16 + 4;
y(cbs) = -s(cbs).*(64 - s(cbs).^2).^(3/2)/512; % negate y
```

Plot the geometry function.

```
pdegplot('cardioid3','EdgeLabels','on')
axis equal
```
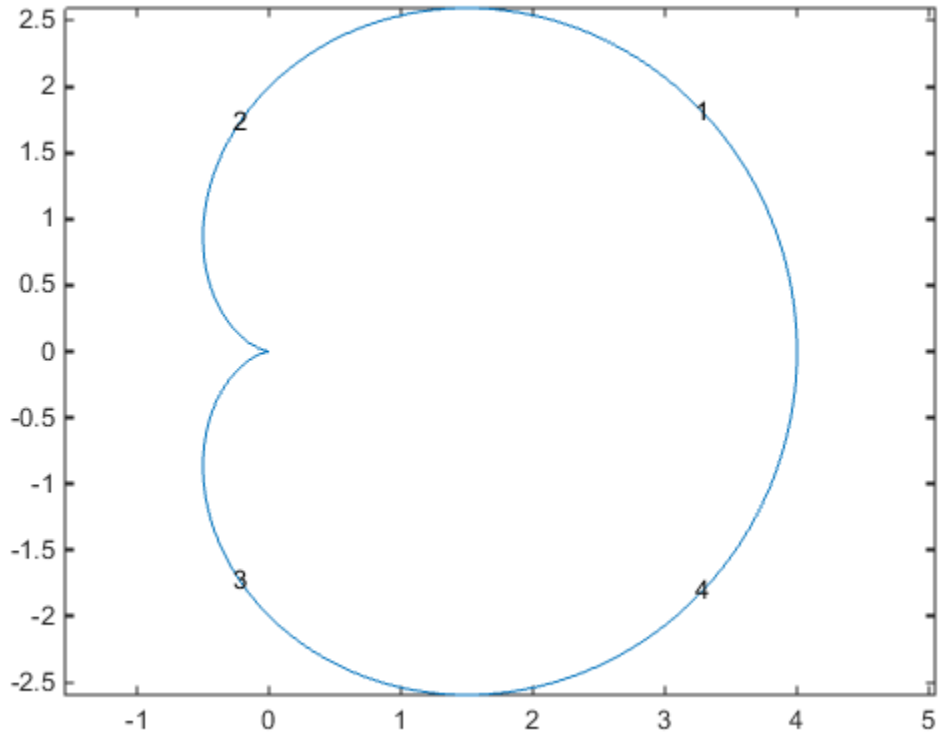
This analytic geometry looks slightly smoother than the previous versions. However, the difference is inconsequential in terms of calculations.

Geometry Not Proportional to Arc Length

You can write a geometry function where the parameter is not proportional to arc length. This can yield a distorted mesh.

```
function [x,y] = cardioid4(bs,s)
%CARDIOID4 Geometry file defining the geometry of a cardioid.

if nargin == 0
  x = 4; % four segments in boundary
  return
```

```
end

if nargin == 1
  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);
  return
end

r = 2*(1+cos(s)); % s is not proportional to arc length
x = r.*cos(s);
y = r.*sin(s);
```

Plot the geometry function.

```
pdegplot('cardioid4','EdgeLabels','on')
axis equal
```
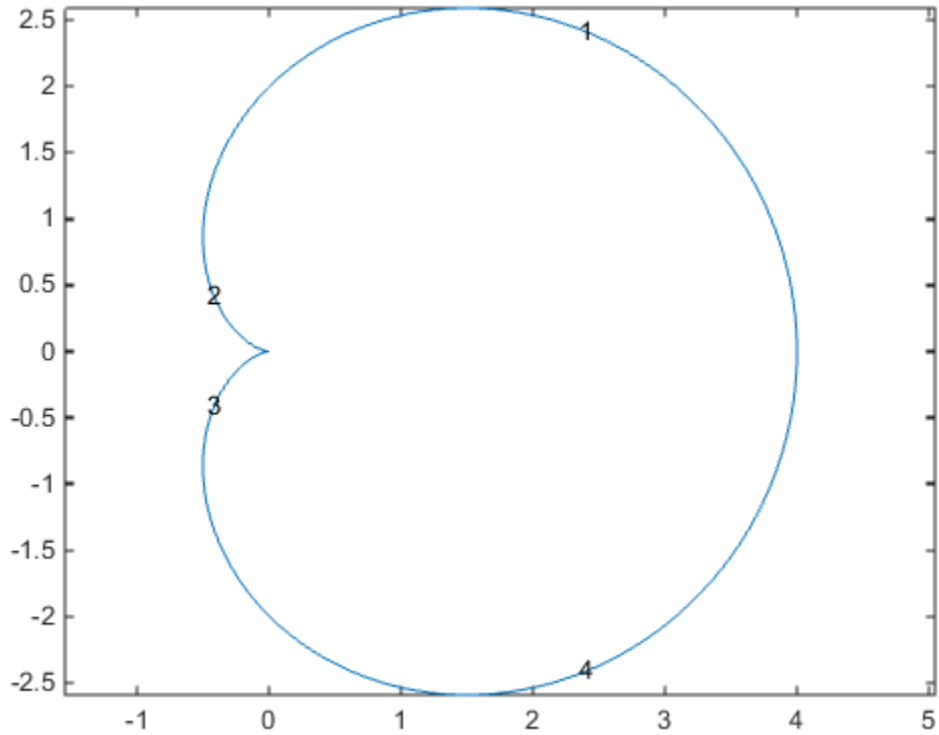
The labels are not evenly spaced on the edges because the parameter is not proportional to arc length.

Examine the default mesh for each of the four methods of creating geometry.

```
subplot(2,2,1)
[p,e,t] = initmesh(@cardioid1);
pdeplot(p,e,t)
title('Polygons')
axis equal
subplot(2,2,2)
[p,e,t] = initmesh(@cardioid2);
pdeplot(p,e,t)
title('Integral')
```

```
axis equal
subplot(2,2,3)
[p,e,t] = initmesh(@cardioid3);
pdeplot(p,e,t)
title('Analytic')
axis equal
subplot(2,2,4)
[p,e,t] = initmesh(@cardioid4);
pdeplot(p,e,t)
title('Distorted')
axis equal
```
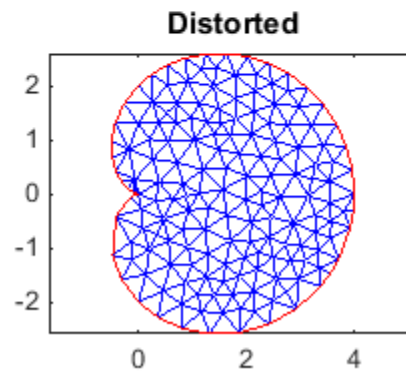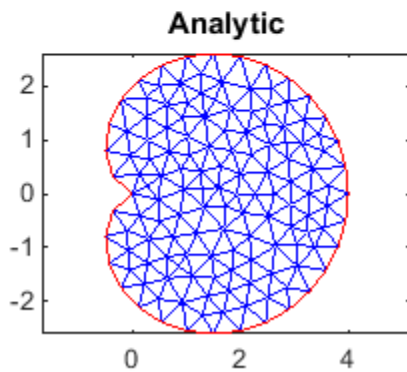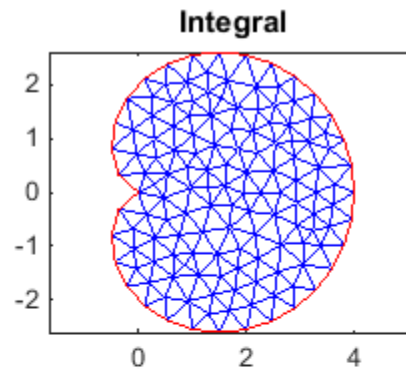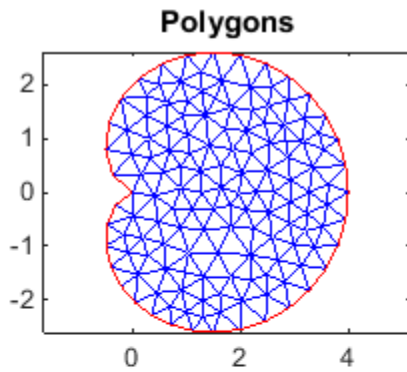
While the "Distorted" mesh looks a bit less regular than the other meshes (it has some very narrow triangles near the cusp of the cardioid), all of the meshes appear to be usable.

Geometry Function Example with Subdomains and a Hole

This example shows how to create a geometry file for a region with subdomains and a hole. It uses the "Analytic" cardioid example from "Arc Length Calculations for a Geometry Function" on page 2-31 and a variant of the circle function from "Geometry Function for a Circle" on page 2-29.

The geometry consists of an outer cardioid that is divided into an upper half called subdomain 1 and a lower half called subdomain 2. Also, the lower half has a circular hole centered at (1,–1) and of radius 1/2. Here is the code of the geometry function.

```
function [x,y] = cardg3(bs,s)
% CARDG3 Geometry File defining the geometry of a cardioid with two
% subregions and a hole.

if nargin == 0
  x = 9; % 9 segments
  return
end

if nargin == 1
    % Outer cardioid
    dl = [ 0    4    8   12
      4    8   12   16
      1    1    2    2 % Region 1 to the left in the upper half, 2 in the lower
      0    0    0    0];
    % Dividing line between top and bottom
    dl2 = [0
        4
        1 % Region 1 to the left
        2]; % Region 2 to the right
    % Inner circular hole
    dl3 = [  0      pi/2   pi      3*pi/2
        pi/2   pi     3*pi/2  2*pi
        0      0      0       0 % To the left is empty
        2      2      2       2]; % To the right is region 2
    % Combine the three edge matrices
    dl = [dl,dl2,dl3];
```

```
    x = dl(:,bs);
    return
end

x = zeros(size(s));
y = zeros(size(s));
if numel(bs) == 1 % Does bs need scalar expansion?
    bs = bs*ones(size(s)); % Expand bs
end


cbs = find(bs < 3); % Upper half of cardiod
x(cbs) = s(cbs).^4/512 - 3*s(cbs).^2/16 + 4;
y(cbs) = s(cbs).*(64 - s(cbs).^2).^(3/2)/512;
cbs = find(bs >= 3 & bs <= 4); % Lower half of cardioid
s(cbs) = 16 - s(cbs);
x(cbs) = s(cbs).^4/512 - 3*s(cbs).^2/16 + 4;
y(cbs) = -s(cbs).*(64 - s(cbs).^2).^(3/2)/512;
cbs = find(bs == 5); % Index of straight line
x(cbs) = s(cbs);
y(cbs) = zeros(size(cbs));
cbs = find(bs > 5); % Inner circle radius 0.25 center (1,-1)
x(cbs) = 1 + 0.25*cos(s(cbs));
y(cbs) = -1 + 0.25*sin(s(cbs));
```

Plot the geometry, including edge labels and subdomain labels.

```
pdegplot(@cardg3,'EdgeLabels','on','SubdomainLabels','on')
axis equal
```
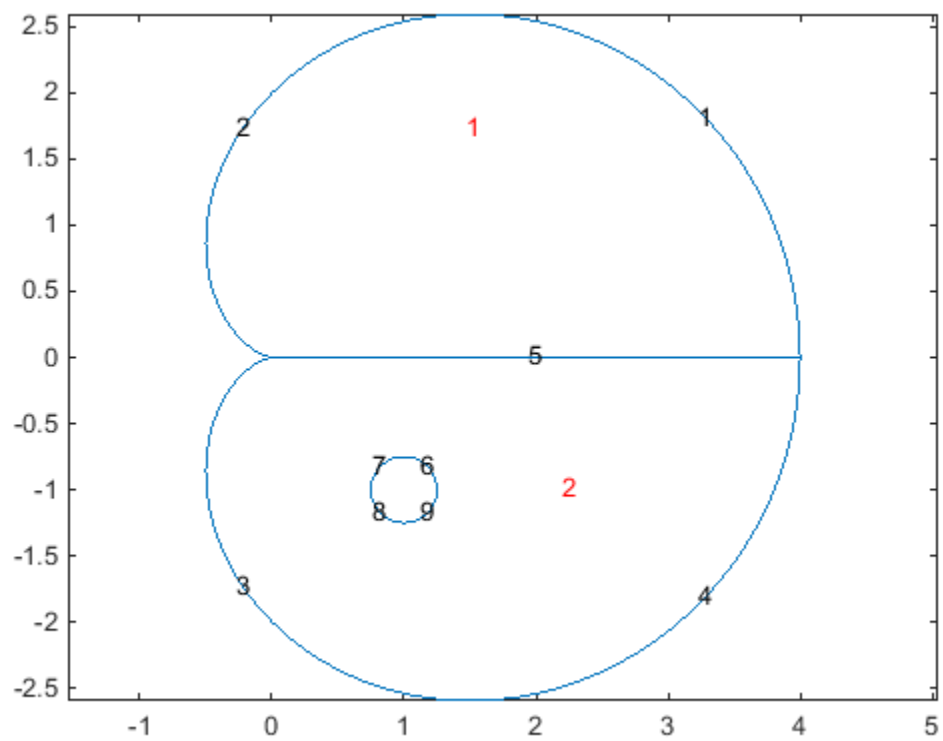
Create and View 3-D Geometry

Note: CREATING AND VIEWING 3-D GEOMETRIES IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Methods of Obtaining 3-D Geometry

Partial Differential Equation Toolbox supports the following ways to obtain 3-D geometry for a PDE model:

- Import an STL file. Generally, you create the STL file by exporting from a CAD system, such as SolidWorks®. For best results, export a fine (not coarse) STL file in binary (not ASCII) format.

- Import a tetrahedral mesh. To import a 3-D mesh, use the `geometryFromMesh` function.

- Create geometry from a 3-D point cloud or a triangulated surface mesh. Use `alphaShape` to create a surface mesh of a point cloud. Use `geometryFromMesh` to create the geometry from a surface mesh.

Import STL File

To create 3-D geometry, import an STL file.

```
model = createpde;
importGeometry(model,'geometryfile.stl');
```

Generally, you create the STL file by exporting from a CAD system, such as SolidWorks. For best results, export a fine (not coarse) STL file in binary (not ASCII) format.

After importing, view the geometry using the `pdegplot` function. To see the face IDs, set the `FaceLabels` name-value pair to `'on'`.

View the geometry examples included with Partial Differential Equation Toolbox.

```
model = createpde;
importGeometry(model,'Torus.stl');
pdegplot(model,'FaceLabels','on')
```
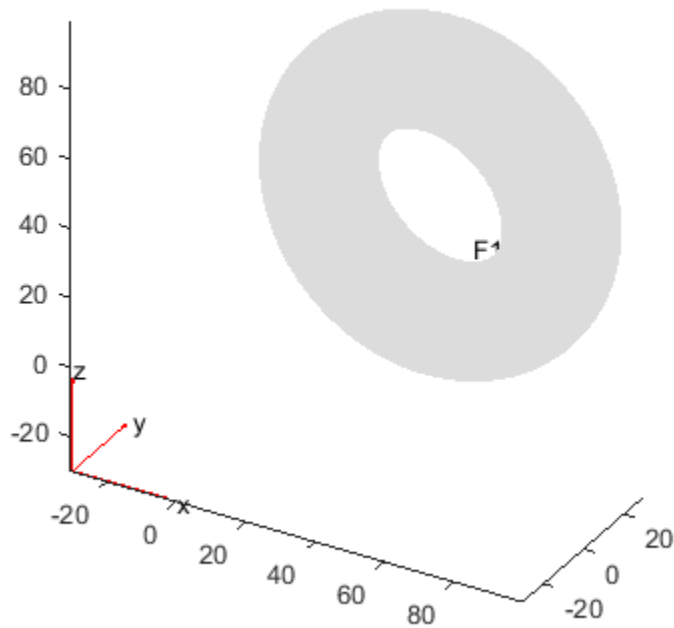
```
model = createpde;
importGeometry(model,'Block.stl');
pdegplot(model,'FaceLabels','on')
```
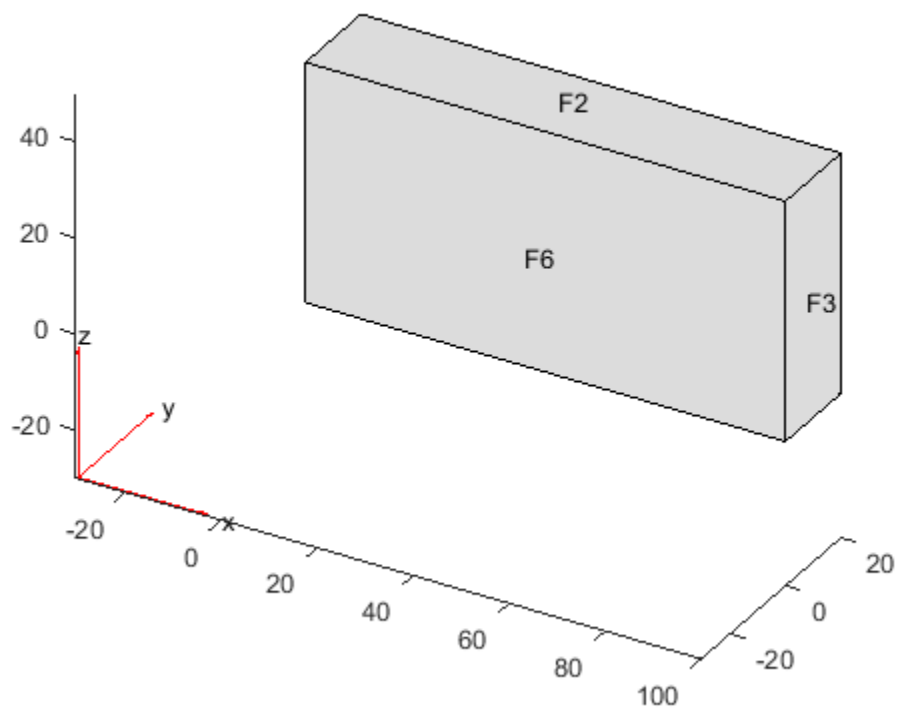
```
model = createpde;
importGeometry(model,'Plate10x10x1.stl');
pdegplot(model,'FaceLabels','on')
```
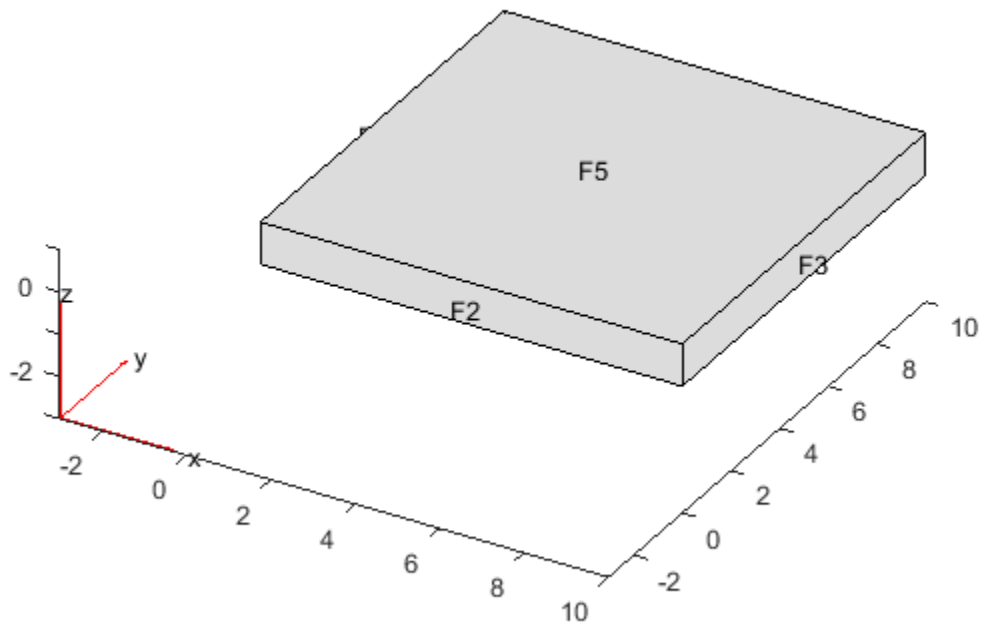
```
model = createpde;
importGeometry(model,'Tetrahedron.stl');
pdegplot(model,'FaceLabels','on')
```
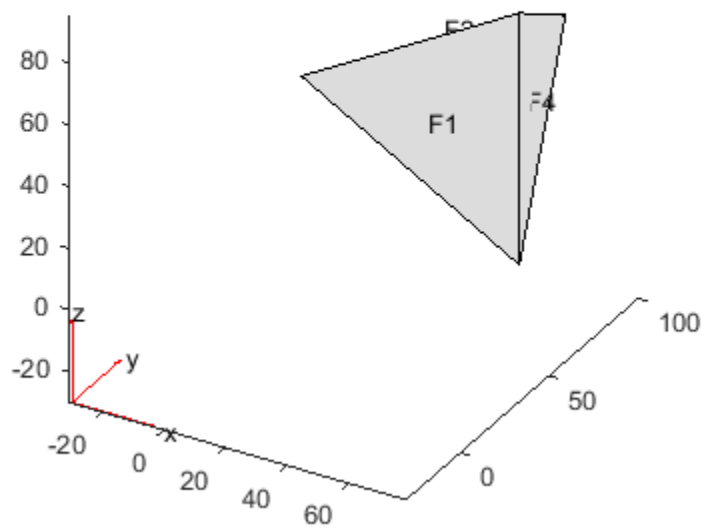
```
model = createpde;
importGeometry(model,'BracketWithHole.stl');
pdegplot(model,'FaceLabels','on')
```
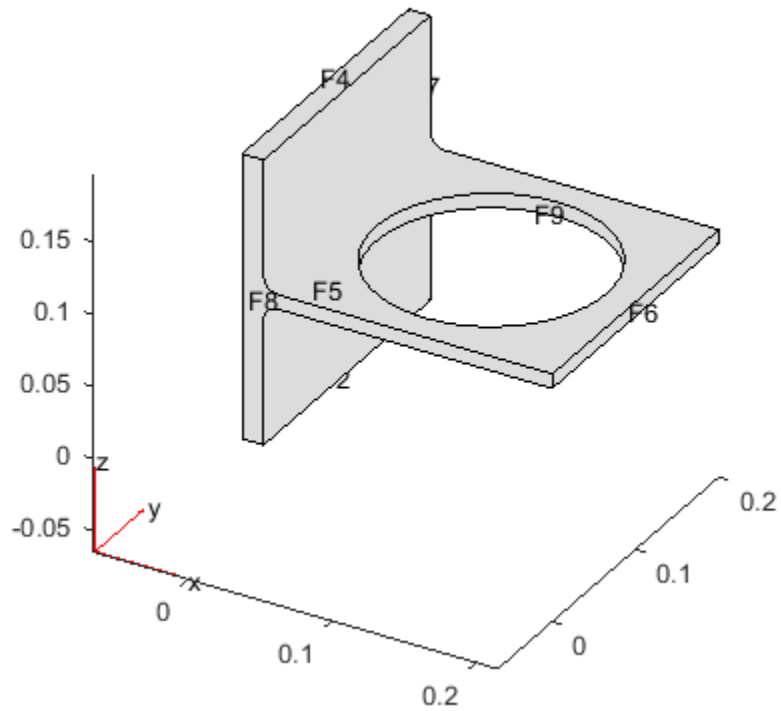
```
model = createpde;
importGeometry(model,'BracketTwoHoles.stl');
pdegplot(model,'FaceLabels','on')
```
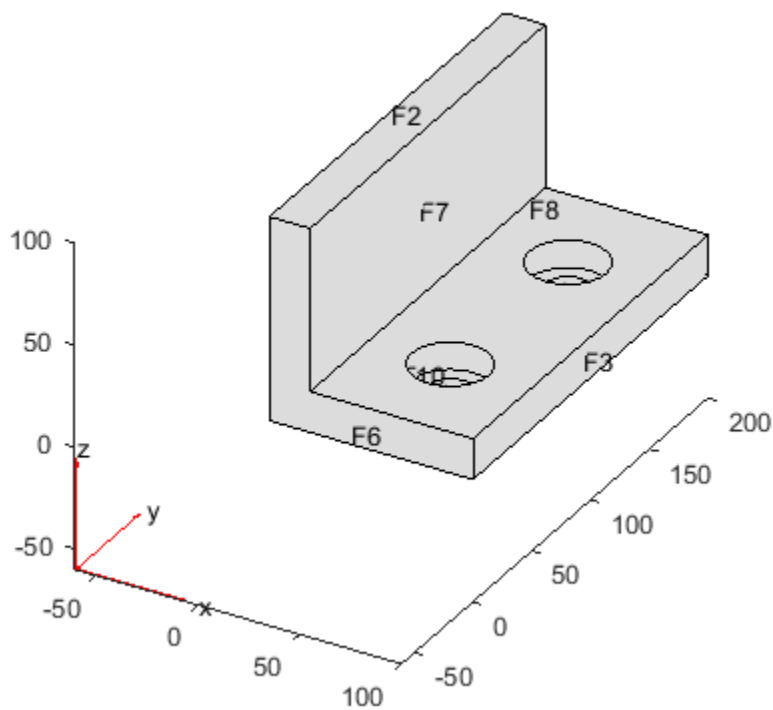
To see hidden portions of the geometry, rotate the figure using the **Rotate 3D** button
. You can rotate the angle bracket to obtain the following view.
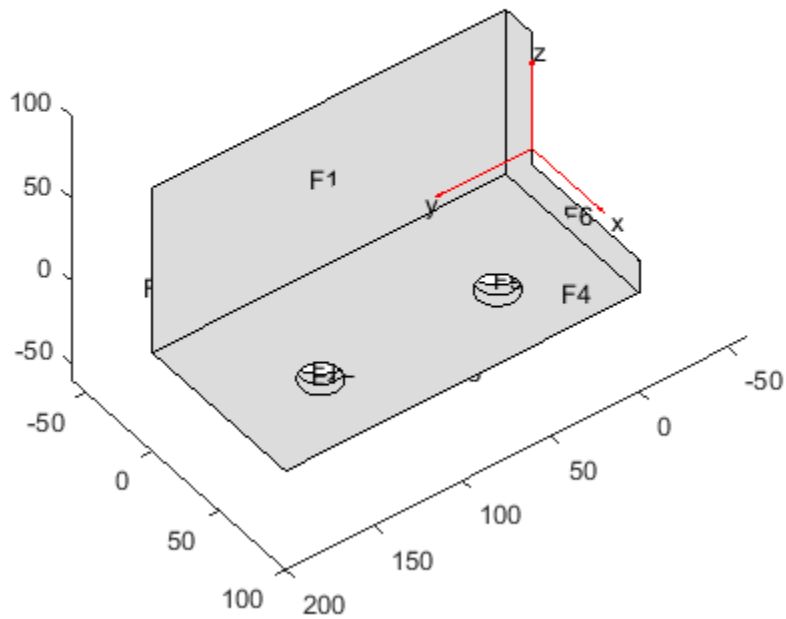
```
model = createpde;
importGeometry(model,'ForearmLink.stl');
h = pdegplot(model,'FaceLabels','on');
```
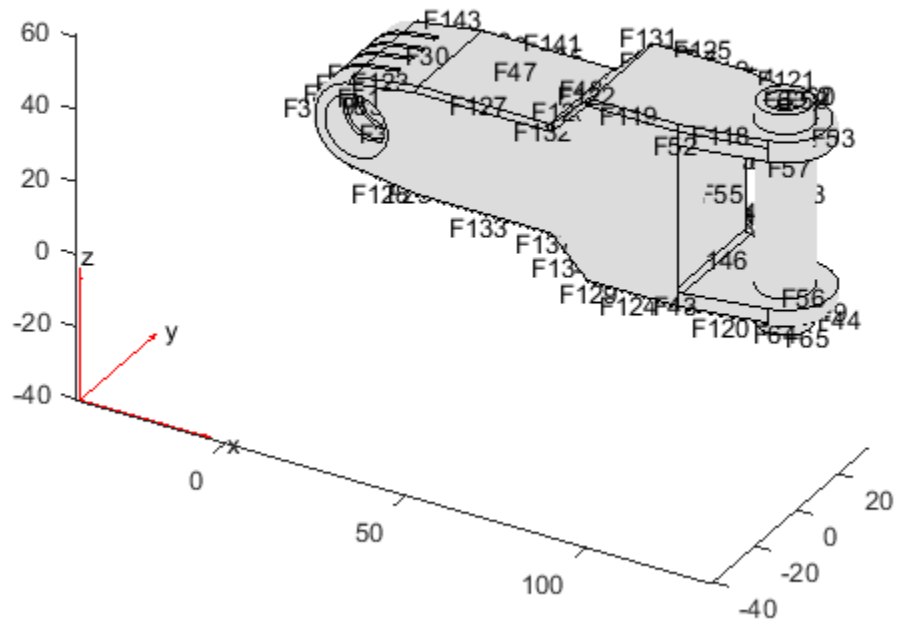
To view hidden faces, set the `FaceAlpha` property to a value less than 1, such as 0.5.

```
h(1).FaceAlpha = 0.5;
```
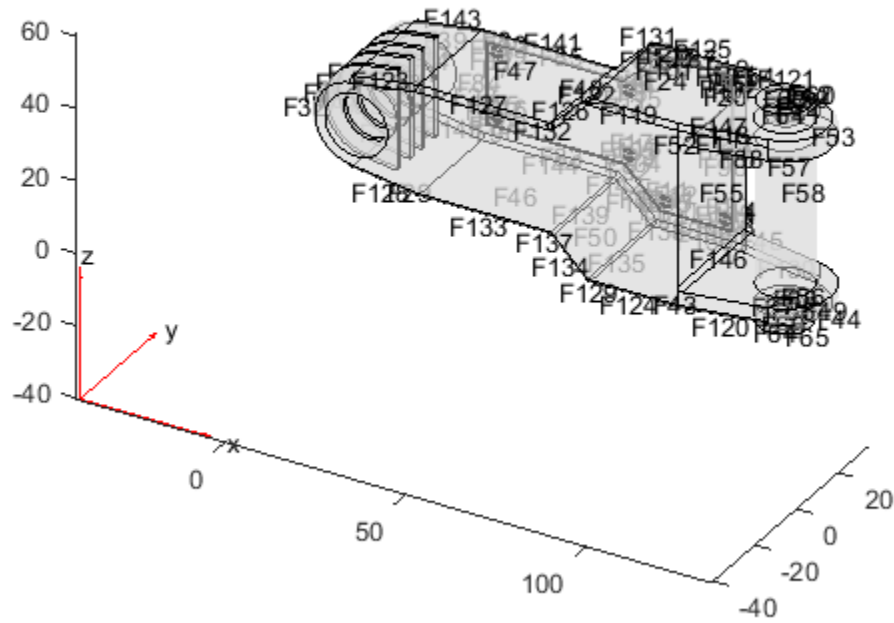
3-D Geometry from a Finite Element Mesh

This example shows how to import a 3-D mesh into a PDE model. Importing a mesh creates the corresponding geometry in the model.

The `tetmesh` file that ships with your software contains a 3-D mesh. Load the data into your Workspace.

```
load tetmesh
```

Examine the node and element sizes.

```
size(tet),size(X)
```

```
ans =

      4969              4


ans =

      1456              3
```

The data is transposed from the required form as described in `geometryFromMesh`.

Create data matrices of the appropriate sizes.

```
nodes = X';
elements = tet';
```

Create a PDE model and import the mesh.

```
model = createpde();
geometryFromMesh(model,nodes,elements);
```

The model contains the imported mesh.

```
model.Mesh
```

```
ans =

  FEMesh with properties:

            Nodes: [3x1456 double]
         Elements: [4x4969 double]
    MaxElementSize: 8.2971
    MinElementSize: 1.9044
    GeometricOrder: 'linear'
```

View the geometry and face numbers.

```
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
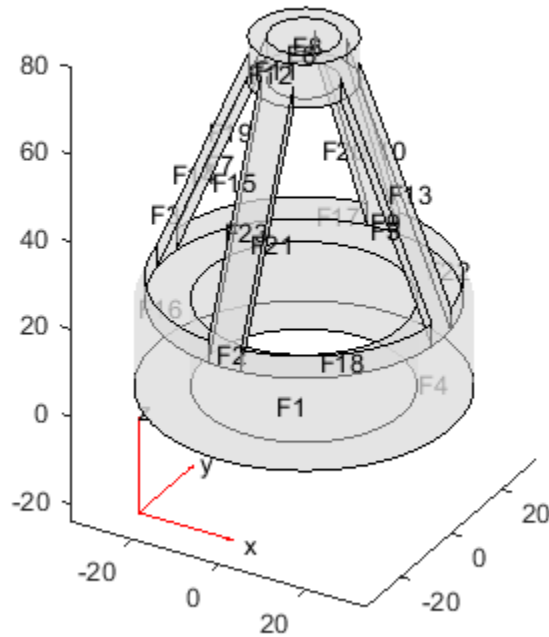
3-D Geometry from Point Cloud

This example shows how to import 3-D geometry from a point cloud.

Create an `alphaShape` object of a block with a cylindrical hole. Import the geometry into a PDE model from the `alphaShape` boundary.

To create the point cloud, create a 2-D mesh grid.

```
[xg, yg] = meshgrid(-3:0.25:3);
xg = xg(:);
yg = yg(:);
```

Create a unit disk. Remove all the mesh grid points that fall inside the unit disk, and include the unit disk points.

```
t = (pi/24:pi/24:2*pi)';
x = cos(t);
y = sin(t);
circShp = alphaShape(x,y,2);
in = inShape(circShp,xg,yg);
xg = [xg(~in); cos(t)];
yg = [yg(~in); sin(t)];
```

Create 3-D copies of the remaining mesh grid points, with the *z*-coordinates ranging from 0 through 1. Together, these points constitute a 3-D point cloud. Combine the points into an `alphaShape` object.

```
zg = ones(numel(xg),1);
xg = repmat(xg,5,1);
yg = repmat(yg,5,1);
zg = zg*(0:.25:1);
zg = zg(:);
shp = alphaShape(xg,yg,zg);
```

Obtain a surface mesh of the `alphaShape` object.

```
[elements,nodes] = boundaryFacets(shp);
```

Put the data in the correct shape for `geometryFromMesh`.

```
nodes = nodes';
elements = elements';
```

Create a PDE model and import the surface mesh.

```
model = createpde();
geometryFromMesh(model,nodes,elements);
```

View the geometry and face numbers.

```
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
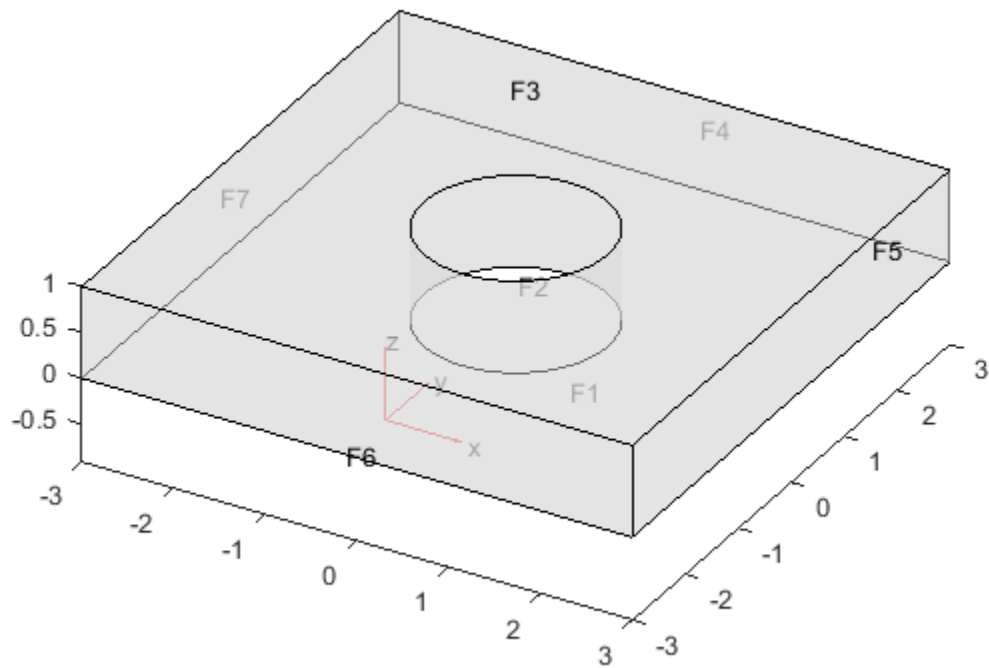
To use the geometry in an analysis, create a volume mesh.

```
generateMesh(model);
```

Functions That Support 3-D Geometry

The following functions support 3-D geometry.

Note: THIS PAGE LISTS THE FUNCTIONS FOR THE RECOMMENDED WORKFLOW.

Solver Functions	Utility and Plotting Functions
solvepde	pdegplot
solvepdeeig	pdemesh
assembleFEMatrices	pdeplot3D

Many Partial Differential Equation Toolbox functions are inherently for 2-D geometry, such as pdecirc, and so do not support 3-D geometry. For functions that are not inherently 2-D but do not support 3-D geometry, you can still obtain 3-D results by using these workarounds.

Functions	Workaround
adaptmesh	None.
jigglemesh, refinemesh	Use the Hmax name-value pair in generateMesh when creating a mesh.
pdegrad, pdecgrad	Use evaluateGradient.
pdecont, pdesurf	Use geometry slices and interpolation, as in "2-D Slices Through 3-D Geometry" on page 3-149.

Related Examples
*   "Solve Problems Using PDEModel Objects" on page 2-14

Put Equations in Divergence Form

Coefficient Matching for Divergence Form

As explained in "Equations You Can Solve Using Legacy Functions" on page 1-3, Partial Differential Equation Toolbox solvers address equations of the form

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

or variants that have derivatives with respect to time, or that have eigenvalues, or are systems of equations. These equations are in *divergence form*, where the differential operator begins $\nabla\cdot$. The coefficients a, c, and f are functions of position (x, y, z) and possibly of the solution u.

However, you can have equations in a form with all the derivatives explicitly expanded, such as

$$\left(1 + x^2\right)\frac{\partial^2 u}{\partial x^2} - 3xy\frac{\partial^2 u}{\partial x \partial y} + \frac{\left(1 + y^2\right)}{2}\frac{\partial^2 u}{\partial y^2} = 0.$$

In order to transform this expanded equation into toolbox format, you can try to match the coefficients of the equation in divergence form to the expanded form. In divergence form, if

$$c = \begin{pmatrix} c_1 & c_3 \\ c_2 & c_4 \end{pmatrix},$$

then

$$\nabla \cdot (c \nabla u) = c_1 u_{xx} + (c_2 + c_3) u_{xy} + c_4 u_{yy}$$
$$+ \left(\frac{\partial c_1}{\partial x} + \frac{\partial c_2}{\partial y} \right) u_x + \left(\frac{\partial c_3}{\partial x} + \frac{\partial c_4}{\partial y} \right) u_y.$$

Matching coefficients in the u_{xx} and u_{yy} terms in $-\nabla \cdot (c \nabla u)$ to the equation, you get

$$c_1 = -\left(1 + x^2\right)$$
$$c_4 = -\left(1 + y^2\right) / 2.$$

Then looking at the coefficients of u_x and u_y, which should be zero, you get

$$\left(\frac{\partial c_1}{\partial x} + \frac{\partial c_2}{\partial y} \right) = -2x + \frac{\partial c_2}{\partial y}$$

so

$$c_2 = 2xy.$$
$$\left(\frac{\partial c_3}{\partial x} + \frac{\partial c_4}{\partial y} \right) = \frac{\partial c_3}{\partial x} - y$$

so

$$c_3 = xy.$$

This completes the conversion of the equation to the divergence form

$$-\nabla \cdot (c \nabla u) = 0.$$

Boundary Conditions Can Affect the c Coefficient

The c coefficient appears in the generalized Neumann condition

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

So when you derive a divergence form of the c coefficient, keep in mind that this coefficient appears elsewhere.

For example, consider the 2-D Poisson equation $-u_{xx} - u_{yy} = f$. Obviously, you can take $c = 1$. But there are other c matrices that lead to the same equation: any that have $c(2) + c(3) = 0$.

$$\nabla \cdot (c\nabla u) = \nabla \cdot \left(\begin{pmatrix} c_1 & c_3 \\ c_2 & c_4 \end{pmatrix} \begin{pmatrix} u_x \\ u_y \end{pmatrix} \right)$$

$$= \frac{\partial}{\partial x}\left(c_1 u_x + c_3 u_y\right) + \frac{\partial}{\partial y}\left(c_2 u_x + c_4 u_y\right)$$

$$= c_1 u_{xx} + c_4 u_{yy} + \left(c_2 + c_3\right)u_{xy}.$$

So there is freedom in choosing a c matrix. If you have a Neumann boundary condition such as

$$\vec{n} \cdot (c\nabla u) = 2,$$

the boundary condition depends on which version of c you use. In this case, make sure that you take a version of c that is compatible with both the equation and the boundary condition.

Some Equations Cannot Be Converted

Sometimes it is not possible to find a conversion to a divergence form such as

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

For example, consider the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\cos(x+y)}{4}\frac{\partial^2 u}{\partial x \partial y} + \frac{1}{2}\frac{\partial^2 u}{\partial y^2} = 0.$$

By simple coefficient matching, you see that the coefficients c_1 and c_4 are -1 and $-1/2$ respectively. However, there are no c_2 and c_3 that satisfy the remaining equations,

$$c_2 + c_3 = \frac{-\cos(x+y)}{4}$$

$$\frac{\partial c_1}{\partial x} + \frac{\partial c_2}{\partial y} = \frac{\partial c_2}{\partial y} = 0$$

$$\frac{\partial c_3}{\partial x} + \frac{\partial c_4}{\partial y} = \frac{\partial c_3}{\partial x} = 0.$$

Systems of PDEs

As described in "Equations You Can Solve Using Recommended Functions" on page 1-6, Partial Differential Equation Toolbox can solve systems of PDEs. This means you can have N coupled PDEs, with coupled boundary conditions. The solver `solvepde` 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 geometry and mesh. It is only N, the number of equations, that can vary.

Scalar PDE Coefficients

> **Note:** **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "Equations You Can Solve Using Recommended Functions" on page 1-6.

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, for 3-D geometry, z) 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 "Specify Scalar PDE Coefficients in String Form" on page 2-68.
- MATLAB function — See "Specify 2-D Scalar Coefficients in Function Form" on page 2-74.

For an example incorporating each way to represent coefficients, see "Solve PDE with Coefficients in Functional Form" on page 2-79.

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.

Specify Scalar PDE Coefficients in String Form

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see the recommended examples on the "PDE Coefficients" page.

Write a text expression using these conventions:

- `'x'` — *x*-coordinate
- `'y'` — *y*-coordinate
- `'z'` — *z*-coordinate (3-D geometry)
- `'u'` — Solution of equation
- `'ux'` — Derivative of *u* in the *x*-direction
- `'uy'` — Derivative of *u* in the *y*-direction
- `'uz'` — Derivative of *u* in the *z*-direction (3-D geometry)
- `'t'` — Time (parabolic and hyperbolic equations)
- `'sd'` — Subdomain number (not used in 3-D geometry)

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. For 2-D geometry, 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
```

```
f2 = cos(x + y); % coefficient on subdomain 2
f(r) = f2(r); % 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:
`'2 + tanh(x.*y)!cosh(x)./(1 + x.^2 + y.^2)!x.^2 + y.^2'`

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-element column 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-element column 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-element column 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 the PDE app. 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'`, `'uy'`, or `'uz'`, you must use the `pdenonlin` solver instead of `assempde`. In the PDE app, select **Solve > Parameters > Use nonlinear solver**.

Coefficients for Scalar PDEs in PDE App

To enter coefficients for your PDE, select **PDE > PDE Specification**.

Enter text expressions 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 expression to represent a coefficient:

```
(x + y)./(x.^2 + y.^2 + 1) + 3 + sin(t)./(1 + u.^4)
```

For elliptic problems, when you include u, ux, or uy, you must use the nonlinear solver. Select **Solve > Parameters > Use nonlinear solver**.

Note:

- Do not use quotes or unnecessary spaces in your entries. 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.

- 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
f2 = cos(x + y); % coefficient on subdomain 2
f(r) = f2(r); % f on subdomain 2
```

Use `fcoeff(x,y,t,sd)` as the f coefficient in the `parabolic` solver.

Coefficient	Value
c	1.0
a	0.0
f	fcoeff(x,y,t,sd)
d	1.0

The coefficient *c* is a 2-by-2 matrix. You can give 1-, 2-, 3-, or 4-element matrix expressions. Separate the expressions for elements by spaces. These expressions mean:

- 1-element expression: $\begin{pmatrix} c & 0 \\ 0 & c \end{pmatrix}$

- 2-element expression: $\begin{pmatrix} c(1) & 0 \\ 0 & c(2) \end{pmatrix}$

- 3-element expression: $\begin{pmatrix} c(1) & c(2) \\ c(2) & c(3) \end{pmatrix}$

- 4-element expression: $\begin{pmatrix} c(1) & c(3) \\ c(2) & c(4) \end{pmatrix}$

For example, c is a symmetric matrix with constant diagonal entries and `cos(xy)` as the off-diagonal terms:

```
1.1 cos(x.*y) 5.5
```

Coefficient	Value
c	1.1 cos(x.*y) 5.5
a	0.0
f	10.0
d	1.0

This corresponds to coefficients for the parabolic equation

$$\frac{\partial u}{\partial t} - \nabla \cdot \left(\begin{pmatrix} 1.1 & \cos(xy) \\ \cos(xy) & 5.5 \end{pmatrix} \nabla u \right) = 10.$$

Related Examples

- "Enter Coefficients in the PDE App" on page 2-85

Specify 2-D Scalar Coefficients in Function Form

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see the recommended examples on the "PDE Coefficients" page.

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 "Specify Scalar PDE Coefficients in String Form" on page 2-68. For the most detailed control over coefficients, though, you can write a function form of coefficients.

A coefficient in function form for 2-D geometry 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 2-75.

- p and t are the node points and triangles of the mesh. For a description of these data structures, see "Mesh Data" on page 2-211. 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 the PDE app, pass the coefficient as a string coeffunction without quotes, because the PDE app 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 the PDE app, 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')
```

Specify 3-D PDE Coefficients in Function Form

> **Note:** **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features
> might not be compatible with the legacy workflow. For the corresponding step in the
> recommended workflow, see the recommended examples on the "PDE Coefficients" page.

Usually, the simplest way to give coefficients as the result of a program is to use a string
expression, as described in "Specify Scalar PDE Coefficients in String Form" on page
2-68. For more detailed control over coefficients, though, you can write coefficients in
function form.

A coefficient in function form for 3-D geometry uses this syntax:

```
coeff = myfun(region,state)
```

`coeff` represents any coefficient: `c`, `a`, `f`, or `d`. Partial Differential Equation Toolbox
solvers pass the `region` and `state` data to your function.

- `region` is a structure with these fields:

    - `region.x`
    - `region.y`
    - `region.z`

    The fields represent the x-, y-, and z- coordinates of points for which your function
    calculates coefficient values. The region fields are row vectors.

- `state` is a structure with these fields:

    - `state.u`
    - `state.ux`
    - `state.uy`
    - `state.uz`
    - `state.t`

    The `state.u` field represents the current value of the solution u. The `state.ux`,
    `state.uy`, and `state.uz` fields are estimates of the solution's partial derivatives ($\partial u/\partial x$, $\partial u/\partial y$, and $\partial u/\partial z$) at the corresponding points of the region structure. The solution

and gradient estimates are row vectors. The `state.t` field is a scalar representing time for the `parabolic` and `hyperbolic` solvers.

The `coeff` output of your function is an NC-by-M matrix, where

- NC is the length of a coefficient column vector.

    - `f` — NC is the same as the number of equations, N.

    - `a` or `d` — NC can be 1, N, N(N+1)/2, or N^2 (see "a or d Coefficient for Systems" on page 2-148).

    - `c` — NC can have many different values in the range 1 to $9N^2$ (see "c Coefficient for Systems" on page 2-125).

- M is the length of any of the `region` fields. This is also the length of the `state.u` fields.

Your function must compute in a vectorized fashion. In other words, it must return the matrix of values for every point in `region`. For example, in an $N = 1$ problem where the `f` coefficient is $1 + x^2$, one possible function is:

```
function fcoeff = ffunction(region,state)

fcoeff = 1 + region.x.^2;
```

To pass this coefficient to the `parabolic` solver, set the coefficient to `@ffunction`. For example:

```
f = @ffunction;
% Assume the other inputs are defined
u = parabolic(u0,tlist,model,c,a,f,d);
```

If you need a constant value, use the size of `region.x` as the number of columns of the matrix. For an $N = 3$ problem:

```
function fcoeff = ffunction(region,state)

fcoeff = ones(3,length(region.x));
```

Solve PDE with Coefficients in Functional Form

Note: THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see the recommended examples on the "PDE Coefficients" page.

This example shows how to write PDE coefficients in string form and in functional form for 2-D geometry.

Geometry

The geometry is a rectangle with a circular hole. Create a PDE model container, and incorporate the geometry into the container.

```
model = createpde(1);

% 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);

% Include the geometry in the model
geometryFromEdges(model,gd);
% View geometry
pdegplot(model,'EdgeLabels','on')
xlim([-1.1 1.1])
axis equal
```
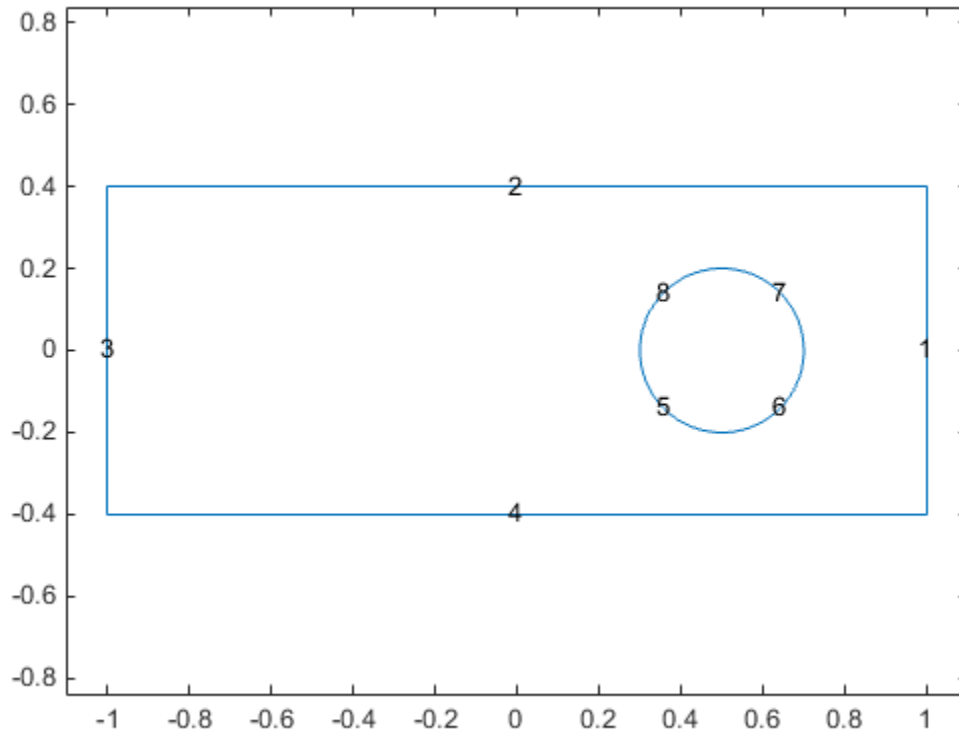
PDE Coefficients

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;
```

Boundary conditions

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 $g = x^2 + y^2$.

```
myufun = @(region,state)state.time*(region.x - region.y);
mygfun = @(region,state)(region.x.^2 + region.y.^2);
applyBoundaryCondition(model,'edge',1:4,'u',myufun,'Vectorized','on');
applyBoundaryCondition(model,'edge',5:8,'q',1,'g',mygfun,'Vectorized','on');
```

The boundary conditions are the same as in "Boundary Conditions for Scalar PDE" on page 2-199. That description uses the older function form for specifying boundary conditions, which is no longer recommended. This description uses the recommended object form.

Initial Conditions

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

```
u0 = 0;
```

Mesh

Create the mesh.

```
generateMesh(model);
```

Parabolic Solution Times

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

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

Solution

Solve the parabolic PDE.

```
d = 5;
a = 0;
f = 'framp(t)';
c = '1 + x.^2 + y.^2';
u = parabolic(u0,tlist,model,c,a,f,d);
```

View an animation of the solution.

```
for tt = 1:size(u,2) % number of steps
    pdeplot(model,'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
```
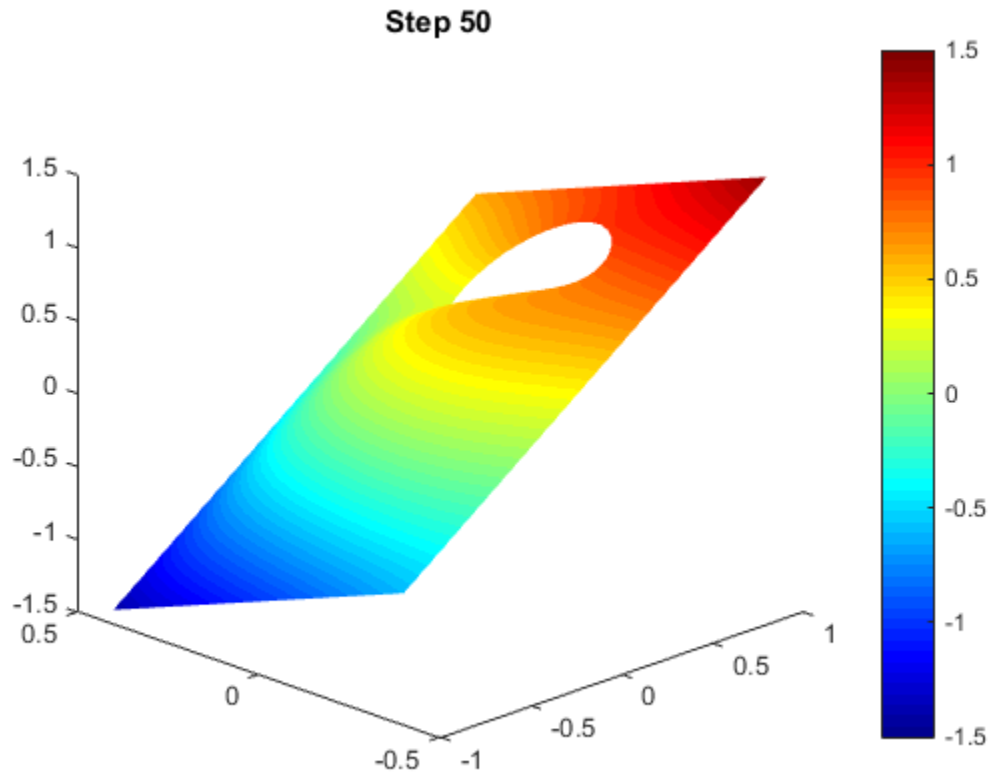
Alternative Coefficient Syntax

Equivalently, you can write a function for the coefficient *f* in the syntax described in "Specify 2-D Scalar Coefficients in Function Form" on page 2-74.

```
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(u0,tlist,model,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(u0,tlist,model,c,a,f,d);
```

Enter Coefficients in the PDE App

This example shows how to enter coefficients in the PDE app.

Caution: Do not include spaces in your coefficient strings in the PDE app. 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:

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

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

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 the PDE app, either by typing `pdetool` at the command line, or selecting **PDE** 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`
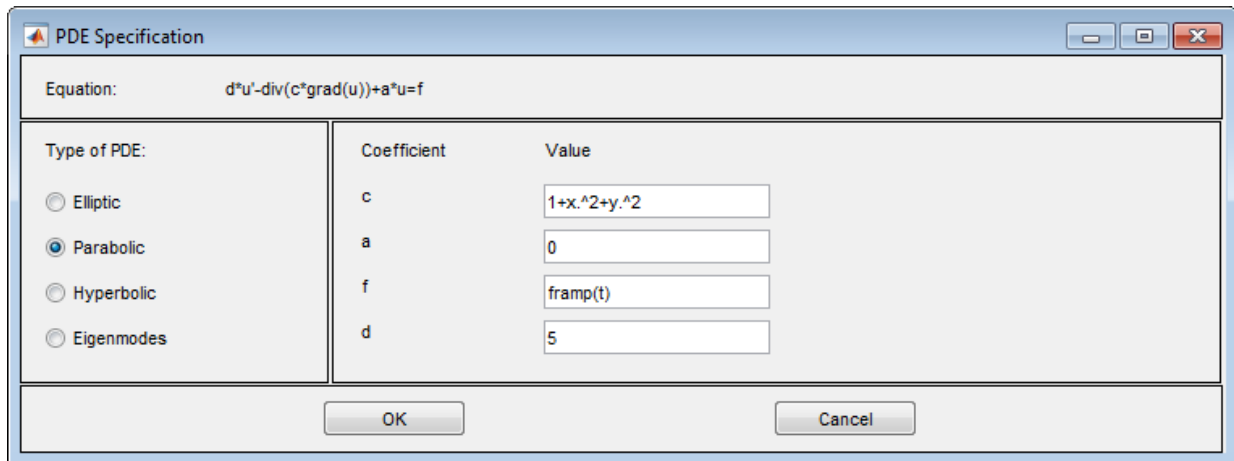
The PDE app interprets all inputs as strings. Therefore, do not include quotes for the `c` or `f` coefficients.

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`.
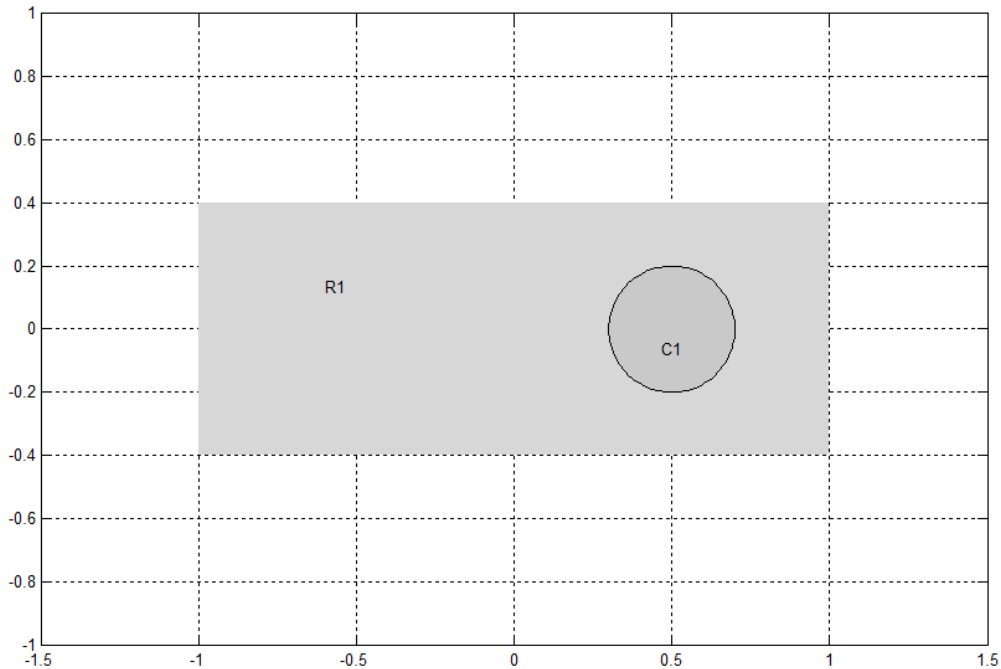
Select **Boundary > Boundary Mode**

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**.

Boundary Condition

Boundary condition equation: h*u=r

Condition type:	Coefficient	Value	Description
○ Neumann	g	0	
⦿ Dirichlet	q	0	
	h	1	
	r	t*(x-y)	

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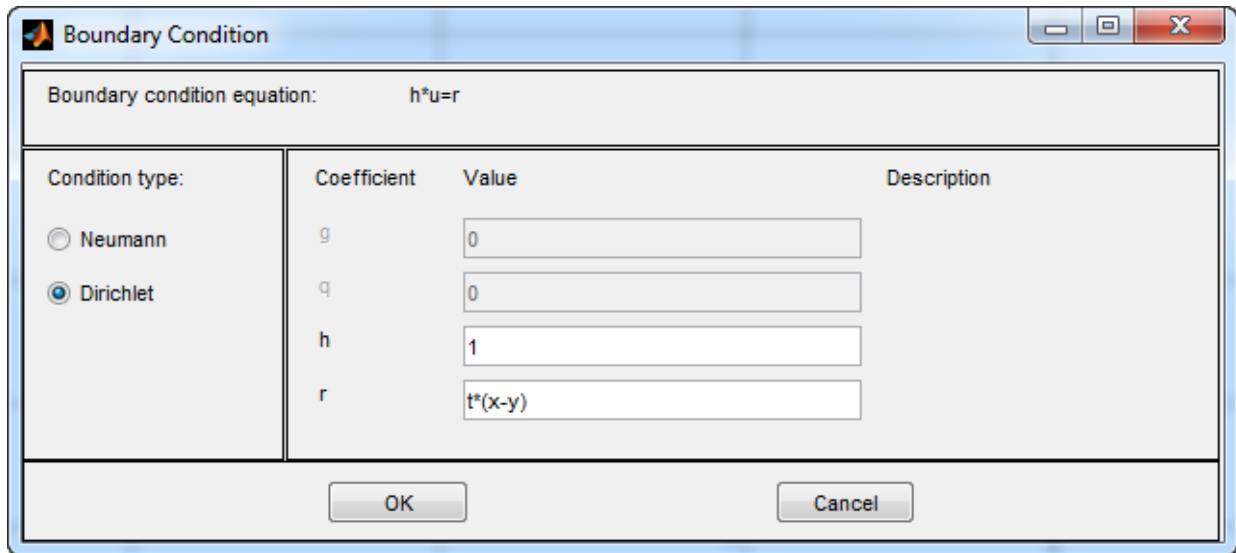
Select the four segments of the inner circle using **Shift**-click, and double-click one of the segments.

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

Boundary Condition

Boundary condition equation: n*c*grad(u)+qu=g

Condition type:	Coefficient	Value	Description
⦿ Neumann	g	x.^2+y.^2	
○ Dirichlet	q	1	
	h	1	
	r	0	

OK   Cancel
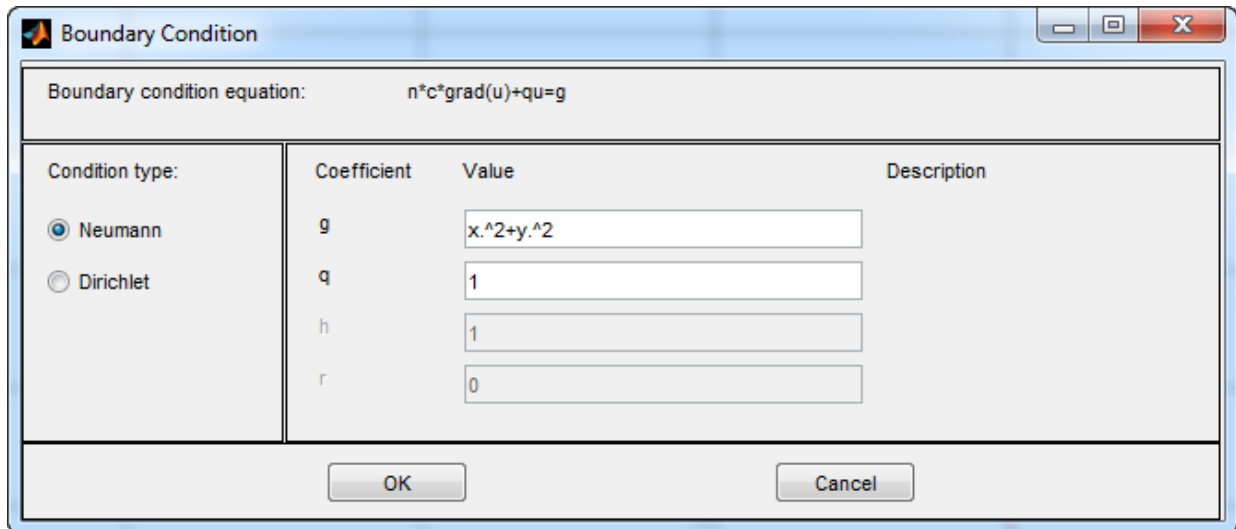
Click to initialize the mesh.

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(0,1,50)` and **u(t0)** = 0. Click **OK**.
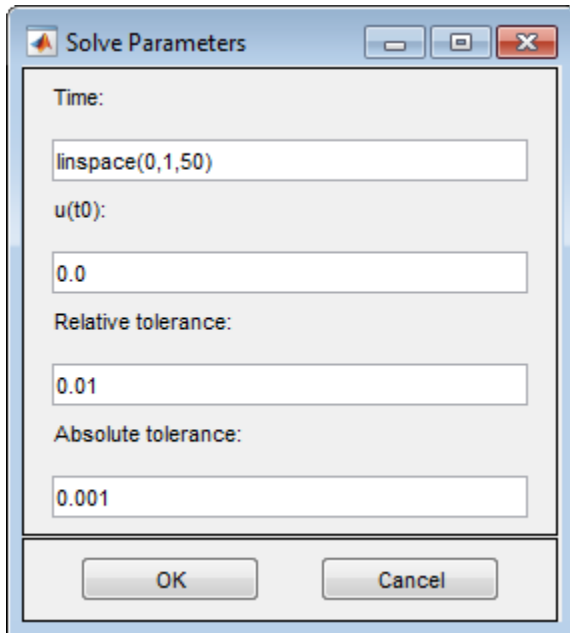
Solve and plot the equation by clicking the button.
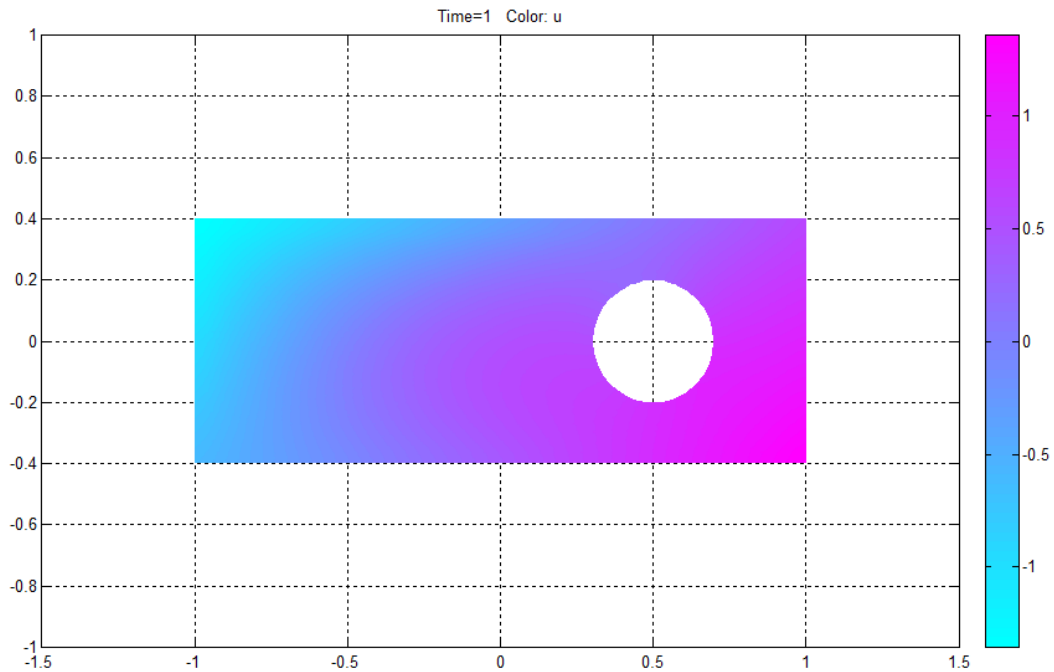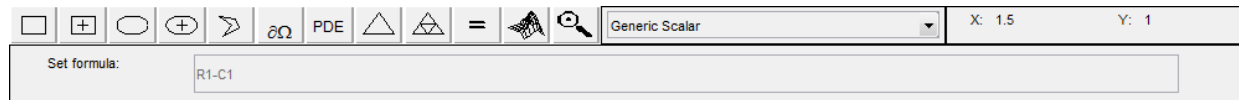
Match the following figure using **Plot > Parameters**.

Click the **Plot** button.

Time=1   Color: u   Height: u

Related Examples
- "Coefficients for Scalar PDEs in PDE App" on page 2-71

Coefficients for Systems of PDEs

As "Systems of PDEs" on page 2-65 describes, toolbox functions can address the case of systems of N PDEs. How do you represent the coefficients of your PDE in the correct form? In general, an elliptic system is

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

For 2-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an N-by-1 matrix with an $(i,1)$-component

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

For 3-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an N-by-1 matrix with an $(i,1)$-component

$$\sum_{j=1}^{N} \left(\frac{\partial}{\partial x} c_{i,j,1,1} \frac{\partial}{\partial x} + \frac{\partial}{\partial x} c_{i,j,1,2} \frac{\partial}{\partial y} + \frac{\partial}{\partial x} c_{i,j,1,3} \frac{\partial}{\partial z} \right) u_j$$

$$+ \sum_{j=1}^{N} \left(\frac{\partial}{\partial y} c_{i,j,2,1} \frac{\partial}{\partial x} + \frac{\partial}{\partial y} c_{i,j,2,2} \frac{\partial}{\partial y} + \frac{\partial}{\partial y} c_{i,j,2,3} \frac{\partial}{\partial z} \right) u_j$$

$$+ \sum_{j=1}^{N} \left(\frac{\partial}{\partial z} c_{i,j,3,1} \frac{\partial}{\partial x} + \frac{\partial}{\partial z} c_{i,j,3,2} \frac{\partial}{\partial y} + \frac{\partial}{\partial z} c_{i,j,3,3} \frac{\partial}{\partial z} \right) u_j.$$

The symbols $\mathbf{a}$ and $\mathbf{d}$ denote N-by-N matrices, and $\mathbf{f}$ denotes a column vector of length N.

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 Coefficient for Systems" on page 2-98, "c Coefficient for Systems" on page 2-125, and "a or d Coefficient for Systems" on page 2-148.

The coefficients can be functions of location (x, y, and, in 3-D, z), and, except for eigenvalue problems, they also can be functions of the solution u or its gradient. For eigenvalue problems, the coefficients cannot depend on the solution u or its gradient.

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.

Systems in the PDE App

You can enter coefficients for a system with $N = 2$ equations in the PDE app, see "Systems of PDEs" on page 2-65. To do so, open the PDE app and select `Generic System`.
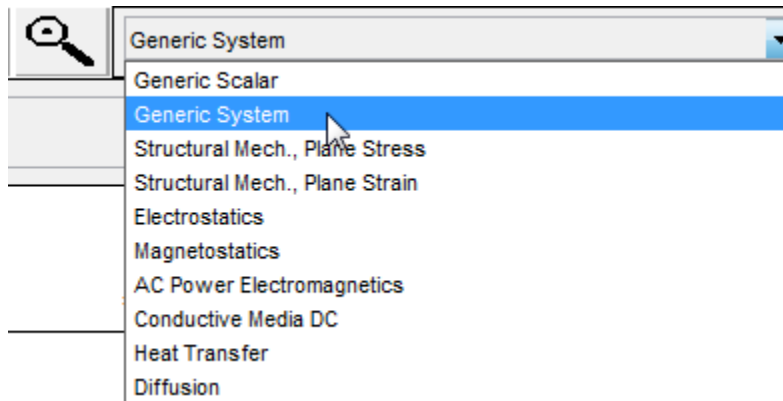
Then select **PDE > PDE Specification**.
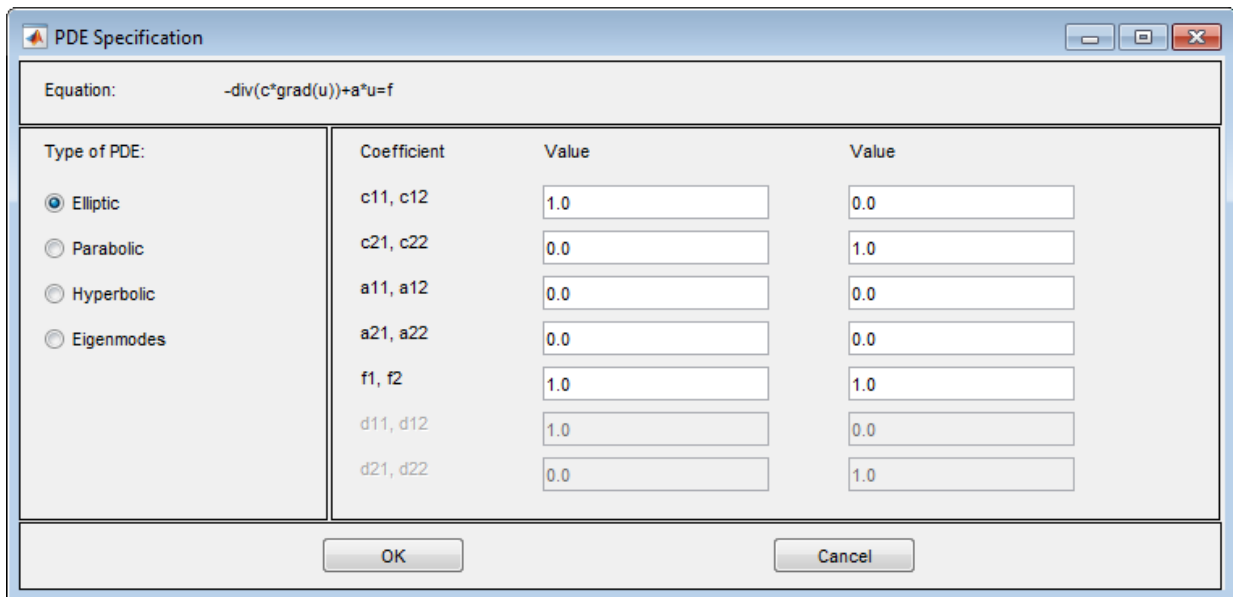
Enter string expressions for coefficients using the form in "Coefficients for Scalar PDEs in PDE App" on page 2-71, with additional options for nonlinear equations. The additional options are:

- Represent the ith component of the solution u using `'u(i)'` for i = 1 or 2.
- Similarly, represent the ith components of the gradients of the solution u using `'ux(i)'` and `'uy(i)'` for i = 1 or 2.

Note: For elliptic problems, when you include coefficients `u(i)`, `ux(i)`, or `uy(i)`, you must use the nonlinear solver. Select **Solve > Parameters > Use nonlinear solver**.

Do not use quotes or unnecessary spaces in your entries.

For higher-dimensional systems, do not use the PDE app. Represent your problem coefficients at the command line.

You can enter scalars into the c matrix, corresponding to these equations:

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

If you need matrix versions of any of the cij coefficients, enter expressions separated by spaces. You can give 1-, 2-, 3-, or 4-element matrix expressions. These mean:

- 1-element expression: $\begin{pmatrix} c & 0 \\ 0 & c \end{pmatrix}$

- 2-element expression: $\begin{pmatrix} c(1) & 0 \\ 0 & c(2) \end{pmatrix}$

- 3-element expression: $\begin{pmatrix} c(1) & c(2) \\ c(2) & c(3) \end{pmatrix}$

- 4-element expression: $\begin{pmatrix} c(1) & c(3) \\ c(2) & c(4) \end{pmatrix}$

For example, these expressions show one of each type (1-, 2-, 3-, and 4-element expressions)

These expressions correspond to the equations

$$
-\nabla\cdot\left(\begin{pmatrix} 4+\cos(xy) & 0 \\ 0 & 4+\cos(xy) \end{pmatrix}\nabla u_1\right) - \nabla\cdot\left(\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}\nabla u_2\right) = 1
$$

$$
-\nabla\cdot\left(\begin{pmatrix} .1 & .2 \\ .2 & .3 \end{pmatrix}\nabla u_1\right) - \nabla\cdot\left(\begin{pmatrix} 7 & .6 \\ .5 & \exp(x-y) \end{pmatrix}\nabla u_2\right) = 2.
$$

f Coefficient for Systems

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "f Coefficient for `specifyCoefficients`" on page 2-101.

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, see "Systems of PDEs" on page 2-65. Give f as any of the following:

- 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 "Specify Scalar PDE Coefficients in String Form" on page 2-68, with additional options for nonlinear equations. The additional options are:

  - Represent the ith component of the solution u using `'u(i)'`.
  - Similarly, represent the ith components of the gradients of the solution u using `'ux(i)'`, `'uy(i)'` and `'uz(i)'`.

  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)*(1+u(1).^2)','x.*y./(1+x.^2+y.^2)')

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

- For 2-D geometry, a function as described in "Specify 2-D Scalar Coefficients in Function Form" on page 2-74. 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 "Specify 2-D Scalar Coefficients in Function Form" on page 2-74. 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 2-75.

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

```matlab
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

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 `0`s in the case of Dirichlet boundary conditions:

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

- For 3-D geometry, a function as described in "Specify 3-D PDE Coefficients in Function Form" on page 2-77. The function should return a matrix of size *N*-by-*Nr*, where *Nr* is the number of points in the region that the solver passes. The function should evaluate `f` at these points. Give solvers the function as a function handle @*filename*, where `filename.m` is a file on your MATLAB path, or is an anonymous function.

Caution It is not reliable to specify f as a scalar or single string expression. Sometimes the toolbox can expand the single input to a vector or character array with N identical rows. But you can get an error when the toolbox fails to determine N. Instead of a scalar or single string, for reliability specify f as a column vector or character array with N rows.

f Coefficient for `specifyCoefficients`

Note: **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see "f Coefficient for Systems" on page 2-98.

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 question is how to write the coefficient f for inclusion in the PDE model via `specifyCoefficients`.

N is the number of equations, see "Systems of PDEs" on page 2-65. Give f as either of the following:

- If f is constant, give a column vector with *N* components. For example, if *N* = 3, f could be:

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

- If f is not constant, give a function handle. The function must be of the form

  ```
  fcoeffunction(region,state)
  ```

  `solvepde` passes the `region` and `state` structures to `fcoeffunction`. The function must return a matrix of size *N*-by-*Nr*, where *Nr* is the number of points in the region that `solvepde` passes. *Nr* is equal to the length of the `region.x` or any other `region` field. The function should evaluate f at these points.

  Pass the coefficient to `specifyCoefficients` as a function handle, such as

  ```
  specifyCoefficients(model,'f',@fcoeffunction,...)
  ```

  - `region` is a structure with these fields:

    - `region.x`
    - `region.y`
    - `region.z`
    - `region.subdomain`

The fields x, y, and z represent the *x*-, *y*-, and *z*- coordinates of points for which your function calculates coefficient values. The subdomain field represents the subdomain numbers, which currently apply only to 2-D models. The region fields are row vectors.

- state is a structure with these fields:

  - state.u

  - state.ux

  - state.uy

  - state.uz

  - state.time

  The state.u field represents the current value of the solution *u*. The state.ux, state.uy, and state.uz fields are estimates of the solution's partial derivatives ($\partial u/\partial x$, $\partial u/\partial y$, and $\partial u/\partial z$) at the corresponding points of the region structure. The solution and gradient estimates are *N*-by-*Nr* matrices. The state.time field is a scalar representing time for time-dependent models.

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

```
function f = fcoeffunction(region,state)

N = 3; % Number of equations
nr = length(region.x); % Number of columns
f = zeros(N,nr); % Allocate f

% Now the particular functional form of f
f(1,:) = region.x - region.y + state.u(1,:);
f(2,:) = 1 + tanh(state.ux(1,:)) + tanh(state.uy(3,:));
f(3,:) = (5 + state.u(3,:)).*sqrt(region.x.^2 + region.y.^2);
```

This represents the coefficient function

$$\mathbf{f} = \begin{bmatrix} x - y + u(1) \\ 1 + \tanh(\partial u(1)/\partial x) + \tanh(\partial u(3)/\partial y) \\ (5 + u(3))\sqrt{x^2 + y^2} \end{bmatrix}.$$

Related Examples

c Coefficient for `specifyCoefficients`

Note: **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see "c Coefficient for Systems" on page 2-125.

In this section...
"Overview of the c Coefficient" on page 2-104
"Definition of the c Tensor Elements" on page 2-105
"Some c Vectors Can Be Short" on page 2-107
"Functional Form" on page 2-121

Overview of the c Coefficient

This topic describes how to write the coefficient `c` in equations such as

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

The topic applies to the recommended workflow for including coefficients in your model using `specifyCoefficients`.

For 2-D systems, `c` is a tensor with $4N^2$ elements. For 3-D systems, `c` is a tensor with $9N^2$ elements. For a definition of the tensor elements, see "Definition of the c Tensor Elements" on page 2-105. N is the number of equations, see "Systems of PDEs" on page 2-65.

To write the coefficient `c` for inclusion in the PDE model via `specifyCoefficients`, give `c` as either of the following:

- If `c` is constant, give a column vector representing the elements in the tensor.
- If `c` is not constant, give a function handle. The function must be of the form

  ccoeffunction(region,state)

  solvepde or solvepdeeig pass the region and state structures to ccoeffunction. The function must return a matrix of size $N1$-by-Nr, where:

- $N1$ is the length of the vector representing the `c` coefficient. There are several possible values of $N1$, detailed in "Some c Vectors Can Be Short" on page 2-107. For 2-D geometry, $1 \leq N1 \leq 4N^2$, and for 3-D geometry, $1 \leq N1 \leq 9N^2$.

- Nr is the number of points in the region that the solver passes. Nr is equal to the length of the `region.x` or any other `region` field. The function should evaluate `c` at these points.

Definition of the c Tensor Elements

For 2-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an N-by-1 matrix with an $(i,1)$-component

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

For 3-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an N-by-1 matrix with an $(i,1)$-component

$$\sum_{j=1}^{N}\left(\frac{\partial}{\partial x}c_{i,j,1,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial x}c_{i,j,1,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial x}c_{i,j,1,3}\frac{\partial}{\partial z}\right)u_{j}$$
$$+\sum_{j=1}^{N}\left(\frac{\partial}{\partial y}c_{i,j,2,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial y}c_{i,j,2,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial y}c_{i,j,2,3}\frac{\partial}{\partial z}\right)u_{j}$$
$$+\sum_{j=1}^{N}\left(\frac{\partial}{\partial z}c_{i,j,3,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial z}c_{i,j,3,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial z}c_{i,j,3,3}\frac{\partial}{\partial z}\right)u_{j}.$$

All representations of the `c` coefficient begin with a "flattening" of the tensor to a matrix. For 2-D systems, the N-by-N-by-2-by-2 tensor flattens to a $2N$-by-$2N$ matrix, where the matrix is logically an N-by-N matrix of 2-by-2 blocks.

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

For 3-D systems, the N-by-N-by-3-by-3 tensor flattens to a $3N$-by-$3N$ matrix, where the matrix is logically an N-by-N matrix of 3-by-3 blocks.

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

These matrices further get flattened into a column vector. First the N-by-N matrices of 2-by-2 and 3-by-3 blocks are transformed into "vectors" of 2-by-2 and 3-by-3 blocks. Then the blocks are turned into vectors in the usual column-wise way.

The coefficient vector **c** relates to the tensor **c** as follows. For 2-D systems,

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

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

For 3-D systems,

$$
\begin{pmatrix}
c(1) & c(4) & c(7) & c(9N+1) & c(9N+4) & c(9N+7) & \cdots & c(9N(N-1)+1) & c(9N(N-1)+4) & c(9N(N-1)+7) \\
c(2) & c(5) & c(8) & c(9N+2) & c(9N+5) & c(9N+8) & \cdots & c(9N(N-1)+2) & c(9N(N-1)+5) & c(9N(N-1)+8) \\
c(3) & c(6) & c(9) & c(9N+3) & c(9N+6) & c(9N+9) & \cdots & c(9N(N-1)+3) & c(9N(N-1)+6) & c(9N(N-1)+9) \\
\\
c(10) & c(13) & c(16) & c(9N+10) & c(9N+13) & c(9N+16) & \cdots & c(9N(N-1)+10) & c(9N(N-1)+13) & c(9N(N-1)+16) \\
c(11) & c(14) & c(17) & c(9N+11) & c(9N+14) & c(9N+17) & \cdots & c(9N(N-1)+11) & c(9N(N-1)+14) & c(9N(N-1)+17) \\
c(12) & c(15) & c(18) & c(9N+12) & c(9N+15) & c(9N+18) & \cdots & c(9N(N-1)+12) & c(9N(N-1)+15) & c(9N(N-1)+18) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
c(9N-8) & c(9N-5) & c(9N-2) & c(18N-8) & c(18N-5) & c(18N-2) & \cdots & c(9N^2-8) & c(9N^2-5) & c(9N^2-2) \\
c(9N-7) & c(9N-4) & c(9N-1) & c(18N-7) & c(18N-4) & c(18N-1) & \cdots & c(9N^2-7) & c(9N^2-4) & c(9N^2-1) \\
c(9N-6) & c(9N-3) & c(9N) & c(18N-6) & c(18N-3) & c(18N) & \cdots & c(9N^2-6) & c(9N^2-3) & c(9N^2)
\end{pmatrix}.
$$

Coefficient $c(i,j,k,l)$ is in row $(9N(j-1) + 9i + 3l + k - 12)$ of the vector `c`.

Some c Vectors Can Be Short

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 for 2-D or $9N^2$ components for 3-D. The following sections give the possibilities.

- "2-D Systems" on page 2-107
- "3-D Systems" on page 2-114

2-D Systems

- "Scalar c, 2-D Systems" on page 2-108

Scalar c, 2-D Systems

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-Element Column Vector c, 2-D Systems

The software interprets a two-element column 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-Element Column Vector c, 2-D Systems

The software interprets a three-element column 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-Element Column Vector c, 2-D Systems

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

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

N-Element Column Vector c, 2-D Systems

The software interprets an N-element column 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-element column vector form takes precedence over the N-element form. 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-element form of c. Instead, you must use the $2N$-element form. If you give c as the vector `[c1;c2;c3]`, the software interprets c as a 3-element form:

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

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

2N-Element Column Vector c, 2-D Systems

The software interprets a $2N$-element column 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-element form takes precedence over the $2N$-element form. For example, if your c matrix is

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

2-111

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

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

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

3N-Element Column Vector c, 2-D Systems

The software interprets a $3N$-element column 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-Element Column Vector c, 2-D Systems

The software interprets a $4N$-element column 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-Element Column Vector c, 2-D Systems

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

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

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-Element Column Vector c, 2-D Systems

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

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

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

3-D Systems

- "Scalar c, 3-D Systems" on page 2-114
- "Three-Element Column Vector c, 3-D Systems" on page 2-115
- "Six-Element Column Vector c, 3-D Systems" on page 2-115
- "Nine-Element Column Vector c, 3-D Systems" on page 2-116
- "N-Element Column Vector c, 3-D Systems" on page 2-117
- "3N-Element Column Vector c, 3-D Systems" on page 2-118
- "6N-Element Column Vector c, 3-D Systems" on page 2-120
- "9N-Element Column Vector c, 3-D Systems" on page 2-120
- "3N(3N+1)/2-Element Column Vector c, 3-D Systems" on page 2-121
- "$9N^2$-Element Column Vector c, 3-D Systems" on page 2-121

Scalar c, 3-D Systems

The software interprets a scalar *c* as a diagonal matrix, with $c(i,i,1,1)$, $c(i,i,2,2)$, and $c(i,i,3,3)$ equal to the scalar, and all other entries 0.

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

Three-Element Column Vector c, 3-D Systems

The software interprets a three-element column vector c as a diagonal matrix, with $c(i,i,1,1)$, $c(i,i,2,2)$, and $c(i,i,3,3)$ as the three entries, and all other entries 0.

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

Six-Element Column Vector c, 3-D Systems

The software interprets a six-element column vector c as a symmetric block diagonal matrix, with
$c(i,i,1,1) = c(1)$
$c(i,i,2,2) = c(3)$

2-115

$c(i,i,1,2) = c(i,i,2,1) = c(2)$
$c(i,i,1,3) = c(i,i,3,1) = c(4)$
$c(i,i,2,3) = c(i,i,3,2) = c(5)$
$c(i,i,3,3) = c(6)$.

In the following diagram, • means the entry is symmetric.

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

Nine-Element Column Vector c, 3-D Systems

The software interprets a nine-element column vector c as a block diagonal matrix.

$$
\begin{pmatrix}
c(1) & c(4) & c(7) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
c(2) & c(5) & c(8) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
c(3) & c(6) & c(9) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
 & & & & & & & & & \\
0 & 0 & 0 & c(1) & c(4) & c(7) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & c(2) & c(5) & c(8) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & c(3) & c(6) & c(9) & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(1) & c(4) & c(7) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(2) & c(5) & c(8) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(3) & c(6) & c(3)
\end{pmatrix}
$$

N-Element Column Vector c, 3-D Systems

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

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

Caution If $N = 3$, 6, or 9, the 3-, 6-, or 9-element column vector form takes precedence over the N-element form. For example, if $N = 3$, and you have a c matrix of the form

$$
\begin{pmatrix}
c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & c(1) & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & \\
0 & 0 & 0 & c(2) & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & c(2) & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & c(2) & 0 & 0 & 0 \\
 & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & c(3) & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & c(3) & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(3)
\end{pmatrix},
$$

you cannot use the N-element form of c. If you give c as the vector `[c1;c2;c3]`, the software interprets `c` as a 3-element form:

$$
\begin{pmatrix}
c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & c(2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & c(3) & 0 & 0 & 0 & 0 & 0 & 0 \\
& & & & & & & & \\
0 & 0 & 0 & c(1) & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & c(2) & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & c(3) & 0 & 0 & 0 \\
& & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & c(1) & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & c(2) & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(3)
\end{pmatrix}.
$$

Instead, use one of these forms:

- $6N$-element form — `[c1;0;c1;0;0;c1;c2;0;c2;0;0;c2;c3;0;c3;0;0;c3]`
- $9N$-element form —
  `[c1;0;0;0;c1;0;0;0;c1;c2;0;0;0;c2;0;0;0;c2;c3;0;0;0;c3;0;0;0;c3]`

3N-Element Column Vector c, 3-D Systems

The software interprets a $3N$-element column vector c as a diagonal matrix.

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

Caution If $N = 3$, the 9-element form takes precedence over the $3N$-element form. For example, if your c matrix is

$$\begin{pmatrix} c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & c(2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c(3) & 0 & 0 & 0 & 0 & 0 & 0 \\ & & & & & & & & \\ 0 & 0 & 0 & c(4) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c(5) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c(6) & 0 & 0 & 0 \\ & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & c(7) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(8) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(9) \end{pmatrix},$$

you cannot give c as `[c1;c2;c3;c4;c5;c6;c7;c8;c9]`, because the software interprets this vector as the 9-element form

$$\begin{pmatrix} c(1) & c(4) & c(7) & 0 & 0 & 0 & 0 & 0 & 0 \\ c(2) & c(5) & c(8) & 0 & 0 & 0 & 0 & 0 & 0 \\ c(3) & c(6) & c(9) & 0 & 0 & 0 & 0 & 0 & 0 \\ & & & & & & & & \\ 0 & 0 & 0 & c(1) & c(4) & c(7) & 0 & 0 & 0 \\ 0 & 0 & 0 & c(2) & c(5) & c(8) & 0 & 0 & 0 \\ 0 & 0 & 0 & c(3) & c(6) & c(9) & 0 & 0 & 0 \\ & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & c(1) & c(4) & c(7) \\ 0 & 0 & 0 & 0 & 0 & 0 & c(2) & c(5) & c(8) \\ 0 & 0 & 0 & 0 & 0 & 0 & c(3) & c(6) & c(3) \end{pmatrix}.$$

Instead, use one of these forms:

- $6N$-element form — `[c1;0;c2;0;0;c3;c4;0;c5;0;0;c6;c7;0;c8;0;0;c9]`

- $9N$-element form — `[c1;0;0;0;c2;0;0;0;c3;c4;0;0;0;c5;0;0;0;c6;c7;0;0;0;c8;0;0;0;c9]`

6N-Element Column Vector c, 3-D Systems

The software interprets a $6N$-element column vector c as a symmetric block diagonal matrix. In the following diagram, $\cdot$ means the entry is symmetric.

$$\begin{pmatrix} c(1) & c(2) & c(4) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \bullet & c(3) & c(5) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \bullet & \bullet & c(6) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ & & & & & & & & & \\ 0 & 0 & 0 & c(7) & c(8) & c(10) & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \bullet & c(9) & c(11) & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \bullet & \bullet & c(12) & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(6N-5) & c(6N-4) & c(6N-2) \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \bullet & c(6N-3) & c(6N-1) \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \bullet & \bullet & c(6N) \end{pmatrix}$$

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

9N-Element Column Vector c, 3-D Systems

The software interprets a $9N$-element column vector c as a block diagonal matrix.

$$\begin{pmatrix} c(1) & c(4) & c(7) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ c(2) & c(5) & c(8) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ c(3) & c(6) & c(9) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ & & & & & & & & & \\ 0 & 0 & 0 & c(10) & c(13) & c(16) & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & c(11) & c(14) & c(17) & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & c(12) & c(15) & c(18) & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(9N-8) & c(9N-5) & c(9N-2) \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(9N-7) & c(9N-4) & c(9N-1) \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(9N-6) & c(9N-3) & c(9N) \end{pmatrix}$$

Coefficient $c(i,j,k,l)$ is in row $(9i + 3l + k - 12)$ of the vector `c`.

3N(3N+1)/2-Element Column Vector c, 3-D Systems

The software interprets a $3N(3N+1)/2$-element column vector c as a symmetric matrix. In the following diagram, $\cdot$ means the entry is symmetric.

$$
\begin{pmatrix}
c(1) & c(2) & c(4) & c(7) & c(10) & c(13) & \cdots & c(3(N-1)(3(N-1)+1)/2+1) & c(3(N-1)(3(N-1)+1)/2+4) & c(3(N-1)(3(N-1)+1)/2+7) \\
\cdot & c(3) & c(5) & c(8) & c(11) & c(14) & \cdots & c(3(N-1)(3(N-1)+1)/2+2) & c(3(N-1)(3(N-1)+1)/2+5) & c(3(N-1)(3(N-1)+1)/2+8) \\
\cdot & \cdot & c(6) & c(9) & c(12) & c(15) & \cdots & c(3(N-1)(3(N-1)+1)/2+3) & c(3(N-1)(3(N-1)+1)/2+6) & c(3(N-1)(3(N-1)+1)/2+9) \\
& & & & & & & & & \\
\cdot & \cdot & \cdot & c(16) & c(17) & c(19) & \cdots & c(3(N-1)(3(N-1)+1)/2+10) & c(3(N-1)(3(N-1)+1)/2+13) & c(3(N-1)(3(N-1)+1)/2+16) \\
\cdot & \cdot & \cdot & \cdot & c(18) & c(20) & \cdots & c(3(N-1)(3(N-1)+1)/2+11) & c(3(N-1)(3(N-1)+1)/2+14) & c(3(N-1)(3(N-1)+1)/2+17) \\
\cdot & \cdot & \cdot & \cdot & \cdot & c(21) & \cdots & c(3(N-1)(3(N-1)+1)/2+12) & c(3(N-1)(3(N-1)+1)/2+15) & c(3(N-1)(3(N-1)+1)/2+18) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & c(3N(3N+1)/2-5) & c(3N(3N+1)/2-4) & c(3N(3N+1)/2-2) \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot & c(3N(3N+1)/2-3) & c(3N(3N+1)/2-1) \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot & \cdot & c(3N(3N+1)/2)
\end{pmatrix}
$$

Coefficient $c(i,j,k,l)$, for $i < j$, is in row $(9(j-1)(j-2)/2 + 6(j-1) + 9i + 3l + k - 12)$ of the vector `c`. For $i = j$, coefficient $c(i,j,k,l)$ is in row $(9(i-1)(i-2)/2 + 15(i-1) + 1/2 l(l-1) + k)$ of the vector `c`.

9N²-Element Column Vector c, 3-D Systems

The software interprets a $9N^2$-element column vector c as a matrix.

$$
\begin{pmatrix}
c(1) & c(4) & c(7) & c(3N+1) & c(3N+4) & c(3N+7) & \cdots & c(9N(N-1)+1) & c(9N(N-1)+4) & c(9N(N-1)+7) \\
c(2) & c(5) & c(8) & c(3N+2) & c(3N+5) & c(3N+8) & \cdots & c(9N(N-1)+2) & c(9N(N-1)+5) & c(9N(N-1)+8) \\
c(3) & c(6) & c(9) & c(3N+3) & c(3N+6) & c(3N+9) & \cdots & c(9N(N-1)+3) & c(9N(N-1)+6) & c(9N(N-1)+9) \\
& & & & & & & & & \\
c(10) & c(13) & c(16) & c(3N+10) & c(3N+13) & c(3N+16) & \cdots & c(9N(N-1)+10) & c(9N(N-1)+13) & c(9N(N-1)+16) \\
c(11) & c(14) & c(17) & c(3N+11) & c(3N+14) & c(3N+17) & \cdots & c(9N(N-1)+11) & c(9N(N-1)+14) & c(9N(N-1)+17) \\
c(12) & c(15) & c(18) & c(3N+12) & c(3N+15) & c(3N+18) & \cdots & c(9N(N-1)+12) & c(9N(N-1)+15) & c(9N(N-1)+18) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
c(3N-8) & c(3N-5) & c(3N-2) & c(6N-8) & c(6N-5) & c(6N-2) & \cdots & c(9N^2-8) & c(9N^2-5) & c(9N^2-2) \\
c(3N-7) & c(3N-4) & c(3N-1) & c(6N-7) & c(6N-4) & c(6N-1) & \cdots & c(9N^2-7) & c(9N^2-4) & c(9N^2-1) \\
c(3N-6) & c(3N-3) & c(3N) & c(6N-6) & c(6N-3) & c(6N) & \cdots & c(9N^2-6) & c(9N^2-3) & c(9N^2)
\end{pmatrix}
$$

Coefficient $c(i,j,k,l)$ is in row $(9N(j-1) + 9i + 3l + k - 12)$ of the vector `c`.

Functional Form

If your `c` coefficient is not constant, represent it as a function of the form

```
ccoeffunction(region,state)
```

`solvepde` or `solvepdeeig` pass the `region` and `state` structures to `ccoeffunction`. The function must return a matrix of size $N1$-by-Nr, where:

- $N1$ is the number of coefficients you pass to the solver. There are several possible values of $N1$, detailed in "Some c Vectors Can Be Short" on page 2-107. For 2-D geometry, $1 \le N1 \le 4N^2$, and for 3-D geometry, $1 \le N1 \le 9N^2$.

- Nr is the number of points in the region that the solver passes. Nr is equal to the length of the `region.x` or any other `region` field. The function should evaluate `c` at these points.

Pass the coefficient to `specifyCoefficients` as a function handle, such as

```
specifyCoefficients(model,'c',@ccoeffunction,...)
```

- `region` is a structure with these fields:

  - `region.x`
  - `region.y`
  - `region.z`
  - `region.subdomain`

  The fields `x`, `y`, and `z` represent the x-, y-, and z- coordinates of points for which your function calculates coefficient values. The `subdomain` field represents the subdomain numbers, which currently apply only to 2-D models. The region fields are row vectors.

- `state` is a structure with these fields:

  - `state.u`
  - `state.ux`
  - `state.uy`
  - `state.uz`
  - `state.time`

  The `state.u` field represents the current value of the solution u. The `state.ux`, `state.uy`, and `state.uz` fields are estimates of the solution's partial derivatives ($\partial u/\partial x$, $\partial u/\partial y$, and $\partial u/\partial z$) at the corresponding points of the region structure. The solution and gradient estimates are N-by-Nr matrices. The `state.time` field is a scalar representing time for time-dependent models.

For example, suppose $N = 3$, and you have 2-D geometry. Suppose your `c` matrix is of the form

$$c = \begin{bmatrix} 1 & 2 & & & & \\ 2 & 8 & & & & \\ & & 1+x^2+y^2 & \dfrac{u(2)}{1+u(1)^2+u(3)^2} & & \\ & & \dfrac{u(2)}{1+u(1)^2+u(3)^2} & 1+x^2+y^2 & & \\ & & & & s_1(x,y) & -1 \\ & & & & -1 & s_1(x,y) \end{bmatrix},$$

where unlisted elements are zero. Here $s_1(x,y)$ is 5 in subdomain 1, and is 10 in subdomain 2.

This `c` is a symmetric, block-diagonal matrix with different coefficients in each block. So it is natural to represent `c` as a "3N-Element Column Vector c, 2-D Systems" on page 2-112:

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

For that form, the following function is appropriate.

```
function cmatrix = ccoeffunction(region,state)

n1 = 9;
```

```
nr = numel(region.x);
cmatrix = zeros(n1,nr);
cmatrix(1,:) = ones(1,nr);
cmatrix(2,:) = 2*ones(1,nr);
cmatrix(3,:) = 8*ones(1,nr);
cmatrix(4,:) = 1+region.x.^2 + region.y.^2;
cmatrix(5,:) = state.u(2,:)./(1 + state.u(1,:).^2 + state.u(3,:).^2);
cmatrix(6,:) = cmatrix(4,:);
cmatrix(7,:) = 5*region.subdomain;
cmatrix(8,:) = -ones(1,nr);
cmatrix(9,:) = cmatrix(7,:);
```

To include this function as your c coefficient, pass the function handle `@ccoeffunction`:

```
specifyCoefficients(model,'c',@ccoeffunction,...
```

Related Examples

- "Solve Problems Using PDEModel Objects" on page 2-14
- "f Coefficient for `specifyCoefficients`" on page 2-101
- "m, d, or a Coefficient for specifyCoefficients" on page 2-143
- "Deflection of a Piezoelectric Actuator" on page 3-19

c Coefficient for Systems

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "c Coefficient for `specifyCoefficients`" on page 2-104.

In this section...
"c as Tensor, Matrix, and Vector" on page 2-125
"2-D Systems" on page 2-128
"3-D Systems" on page 2-134

c as Tensor, Matrix, and Vector

This topic describes how to write the coefficient c in equations such as

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

For 2-D systems, the coefficient c is an *N*-by-*N*-by-2-by-2 tensor with components $\mathbf{c}(i,j,k,l)$. *N* is the number of equations (see "Systems of PDEs" on page 2-65). For 3-D systems, c is an *N*-by-*N*-by-3-by-3 tensor.

For 2-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an *N*-by-1 matrix with an $(i,1)$-component

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

For 3-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an *N*-by-1 matrix with an $(i,1)$-component

$$\sum_{j=1}^{N}\left(\frac{\partial}{\partial x}c_{i,j,1,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial x}c_{i,j,1,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial x}c_{i,j,1,3}\frac{\partial}{\partial z}\right)u_j$$

$$+\sum_{j=1}^{N}\left(\frac{\partial}{\partial y}c_{i,j,2,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial y}c_{i,j,2,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial y}c_{i,j,2,3}\frac{\partial}{\partial z}\right)u_j$$

$$+\sum_{j=1}^{N}\left(\frac{\partial}{\partial z}c_{i,j,3,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial z}c_{i,j,3,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial z}c_{i,j,3,3}\frac{\partial}{\partial z}\right)u_j.$$

All representations of the c coefficient begin with a "flattening" of the tensor to a matrix. For 2-D systems, the N-by-N-by-2-by-2 tensor flattens to a $2N$-by-$2N$ matrix, where the matrix is logically an N-by-N matrix of 2-by-2 blocks.

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

For 3-D systems, the N-by-N-by-3-by-3 tensor flattens to a $3N$-by-$3N$ matrix, where the matrix is logically an N-by-N matrix of 3-by-3 blocks.

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

These matrices further get flattened into a column vector. First the N-by-N matrices of 2-by-2 and 3-by-3 blocks are transformed into "vectors" of 2-by-2 and 3-by-3 blocks. Then the blocks are turned into vectors in the usual column-wise way.

The coefficient vector $\mathtt{c}$ relates to the tensor $\mathbf{c}$ as follows. For 2-D systems,

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

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

For 3-D systems,

$$
\begin{pmatrix}
c(1) & c(4) & c(7) & c(9N+1) & c(9N+4) & c(9N+7) & \cdots & c(9N(N-1)+1) & c(9N(N-1)+4) & c(9N(N-1)+7) \\
c(2) & c(5) & c(8) & c(9N+2) & c(9N+5) & c(9N+8) & \cdots & c(9N(N-1)+2) & c(9N(N-1)+5) & c(9N(N-1)+8) \\
c(3) & c(6) & c(9) & c(9N+3) & c(9N+6) & c(9N+9) & \cdots & c(9N(N-1)+3) & c(9N(N-1)+6) & c(9N(N-1)+9) \\
\\
c(10) & c(13) & c(16) & c(9N+10) & c(9N+13) & c(9N+16) & \cdots & c(9N(N-1)+10) & c(9N(N-1)+13) & c(9N(N-1)+16) \\
c(11) & c(14) & c(17) & c(9N+11) & c(9N+14) & c(9N+17) & \cdots & c(9N(N-1)+11) & c(9N(N-1)+14) & c(9N(N-1)+17) \\
c(12) & c(15) & c(18) & c(9N+12) & c(9N+15) & c(9N+18) & \cdots & c(9N(N-1)+12) & c(9N(N-1)+15) & c(9N(N-1)+18) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
c(9N-8) & c(9N-5) & c(9N-2) & c(18N-8) & c(18N-5) & c(18N-2) & \cdots & c(9N^2-8) & c(9N^2-5) & c(9N^2-2) \\
c(9N-7) & c(9N-4) & c(9N-1) & c(18N-7) & c(18N-4) & c(18N-1) & \cdots & c(9N^2-7) & c(9N^2-4) & c(9N^2-1) \\
c(9N-6) & c(9N-3) & c(9N) & c(18N-6) & c(18N-3) & c(18N) & \cdots & c(9N^2-6) & c(9N^2-3) & c(9N^2)
\end{pmatrix}.
$$

Coefficient $c(i,j,k,l)$ is in row $(9N(j-1) + 9i + 3l + k - 12)$ of the vector $\mathtt{c}$.

Express $\mathtt{c}$ as numbers, text expressions, or functions, as in "f Coefficient for Systems" on page 2-98.

Often, your tensor $\mathbf{c}$ has structure, such as symmetric or block diagonal. In many cases, you can represent $\mathbf{c}$ using a smaller vector than one with $4N^2$ components for 2-D or $9N^2$ components for 3-D.

The number of rows in the matrix can differ from $4N^2$ for 2-D or $9N^2$ for 3-D, as described in "2-D Systems" on page 2-128 and "3-D Systems" on page 2-134.

In function form for 2-D systems, the number of columns is `Nt`, which is the number of triangles or tetrahedra in the mesh. The function should evaluate `c` at the triangle or tetrahedron centroids, as in "Specify 2-D Scalar Coefficients in Function Form" on page 2-74. 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 2-75.

For the function form of coefficients of 3-D systems, see "Specify 3-D PDE Coefficients in Function Form" on page 2-77.

2-D Systems

- "Scalar c, 2-D Systems" on page 2-128
- "Two-Element Column Vector c, 2-D Systems" on page 2-129
- "Three-Element Column Vector c, 2-D Systems" on page 2-129
- "Four-Element Column Vector c, 2-D Systems" on page 2-130
- "N-Element Column Vector c, 2-D Systems" on page 2-130
- "2N-Element Column Vector c, 2-D Systems" on page 2-132
- "3N-Element Column Vector c, 2-D Systems" on page 2-133
- "4N-Element Column Vector c, 2-D Systems" on page 2-133
- "2N(2N+1)/2-Element Column Vector c, 2-D Systems" on page 2-133
- "$4N^2$-Element Column Vector c, 2-D Systems" on page 2-134

Scalar c, 2-D Systems

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-Element Column Vector c, 2-D Systems

The software interprets a two-element column 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-Element Column Vector c, 2-D Systems

The software interprets a three-element column 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-Element Column Vector c, 2-D Systems

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

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

N-Element Column Vector c, 2-D Systems

The software interprets an N-element column 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-element column vector form takes precedence over the N-element form. 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-element form of c. Instead, you must use the $2N$-element form. If you give c as the vector [c1;c2;c3], the software interprets c as a 3-element form:

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

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

2N-Element Column Vector c, 2-D Systems

The software interprets a $2N$-element column 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-element form takes precedence over the $2N$-element 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-element form

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

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

3N-Element Column Vector c, 2-D Systems

The software interprets a $3N$-element column 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-Element Column Vector c, 2-D Systems

The software interprets a $4N$-element column 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-Element Column Vector c, 2-D Systems

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

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

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.

$4N^2$-Element Column Vector c, 2-D Systems

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

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

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

3-D Systems

Scalar c, 3-D Systems

The software interprets a scalar c as a diagonal matrix, with $c(i,i,1,1)$, $c(i,i,2,2)$, and $c(i,i,3,3)$ equal to the scalar, and all other entries 0.

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

Three-Element Column Vector c, 3-D Systems

The software interprets a three-element column vector c as a diagonal matrix, with $c(i,i,1,1)$, $c(i,i,2,2)$, and $c(i,i,3,3)$ as the three entries, and all other entries 0.

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

Six-Element Column Vector c, 3-D Systems

The software interprets a six-element column vector c as a symmetric block diagonal matrix, with
$c(i,i,1,1) = c(1)$
$c(i,i,2,2) = c(3)$
$c(i,i,1,2) = c(i,i,2,1) = c(2)$
$c(i,i,1,3) = c(i,i,3,1) = c(4)$
$c(i,i,2,3) = c(i,i,3,2) = c(5)$
$c(i,i,3,3) = c(6)$.

In the following diagram, • means the entry is symmetric.

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

Nine-Element Column Vector c, 3-D Systems

The software interprets a nine-element column vector c as a block diagonal matrix.

$$
\begin{pmatrix}
c(1) & c(4) & c(7) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
c(2) & c(5) & c(8) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
c(3) & c(6) & c(9) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
 & & & & & & & & & \\
0 & 0 & 0 & c(1) & c(4) & c(7) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & c(2) & c(5) & c(8) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & c(3) & c(6) & c(9) & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(1) & c(4) & c(7) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(2) & c(5) & c(8) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(3) & c(6) & c(3)
\end{pmatrix}
$$

N-Element Column Vector c, 3-D Systems

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

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

Caution If $N = 3$, 6, or 9, the 3-, 6-, or 9-element column vector form takes precedence over the N-element form. For example, if $N = 3$, and you have a c matrix of the form

$$
\begin{pmatrix}
c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & c(1) & 0 & 0 & 0 & 0 & 0 & 0 \\
& & & & & & & & \\
0 & 0 & 0 & c(2) & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & c(2) & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & c(2) & 0 & 0 & 0 \\
& & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & c(3) & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & c(3) & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(3)
\end{pmatrix},
$$

you cannot use the N-element form of c. If you give c as the vector [c1;c2;c3], the software interprets c as a 3-element form:

$$\begin{pmatrix} c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & c(2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c(3) & 0 & 0 & 0 & 0 & 0 & 0 \\ \\ 0 & 0 & 0 & c(1) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c(2) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c(3) & 0 & 0 & 0 \\ \\ 0 & 0 & 0 & 0 & 0 & 0 & c(1) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(2) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(3) \end{pmatrix}.$$

Instead, use one of these forms:

- $6N$-element form — `[c1;0;c1;0;0;c1;c2;0;c2;0;0;c2;c3;0;c3;0;0;c3]`
- $9N$-element form —
  `[c1;0;0;0;c1;0;0;0;c1;c2;0;0;0;c2;0;0;0;c2;c3;0;0;0;c3;0;0;0;c3]`

3N-Element Column Vector c, 3-D Systems

The software interprets a $3N$-element column vector c as a diagonal matrix.

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

Caution If $N = 3$, the 9-element form takes precedence over the $3N$-element form. For example, if your c matrix is

$$\begin{pmatrix} c(1) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & c(2) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c(3) & 0 & 0 & 0 & 0 & 0 & 0 \\ \\ 0 & 0 & 0 & c(4) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c(5) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c(6) & 0 & 0 & 0 \\ \\ 0 & 0 & 0 & 0 & 0 & 0 & c(7) & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(8) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & c(9) \end{pmatrix},$$

you cannot give c as `[c1;c2;c3;c4;c5;c6;c7;c8;c9]`, because the software interprets this vector as the 9-element form

$$\begin{pmatrix} c(1) & c(4) & c(7) & 0 & 0 & 0 & 0 & 0 & 0 \\ c(2) & c(5) & c(8) & 0 & 0 & 0 & 0 & 0 & 0 \\ c(3) & c(6) & c(9) & 0 & 0 & 0 & 0 & 0 & 0 \\ \\ 0 & 0 & 0 & c(1) & c(4) & c(7) & 0 & 0 & 0 \\ 0 & 0 & 0 & c(2) & c(5) & c(8) & 0 & 0 & 0 \\ 0 & 0 & 0 & c(3) & c(6) & c(9) & 0 & 0 & 0 \\ \\ 0 & 0 & 0 & 0 & 0 & 0 & c(1) & c(4) & c(7) \\ 0 & 0 & 0 & 0 & 0 & 0 & c(2) & c(5) & c(8) \\ 0 & 0 & 0 & 0 & 0 & 0 & c(3) & c(6) & c(3) \end{pmatrix}.$$

Instead, use one of these forms:

- $6N$-element form — `[c1;0;c2;0;0;c3;c4;0;c5;0;0;c6;c7;0;c8;0;0;c9]`

- $9N$-element form —
  `[c1;0;0;0;c2;0;0;0;c3;c4;0;0;0;c5;0;0;0;c6;c7;0;0;0;c8;0;0;0;c9]`

6N-Element Column Vector c, 3-D Systems

The software interprets a $6N$-element column vector c as a symmetric block diagonal matrix. In the following diagram, $\cdot$ means the entry is symmetric.

$$
\begin{pmatrix}
c(1) & c(2) & c(4) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
\bullet & c(3) & c(5) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
\bullet & \bullet & c(6) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
\\
0 & 0 & 0 & c(7) & c(8) & c(10) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & \bullet & c(9) & c(11) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & \bullet & \bullet & c(12) & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(6N-5) & c(6N-4) & c(6N-2) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & \bullet & c(6N-3) & c(6N-1) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & \bullet & \bullet & c(6N)
\end{pmatrix}
$$

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

9N-Element Column Vector c, 3-D Systems

The software interprets a $9N$-element column vector c as a block diagonal matrix.

$$
\begin{pmatrix}
c(1) & c(4) & c(7) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
c(2) & c(5) & c(8) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
c(3) & c(6) & c(9) & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\
\\
0 & 0 & 0 & c(10) & c(13) & c(16) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & c(11) & c(14) & c(17) & \cdots & 0 & 0 & 0 \\
0 & 0 & 0 & c(12) & c(15) & c(18) & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(9N-8) & c(9N-5) & c(9N-2) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(9N-7) & c(9N-4) & c(9N-1) \\
0 & 0 & 0 & 0 & 0 & 0 & \cdots & c(9N-6) & c(9N-3) & c(9N)
\end{pmatrix}
$$

Coefficient $c(i,j,k,l)$ is in row $(9i + 3l + k - 12)$ of the vector c.

3N(3N+1)/2-Element Column Vector c, 3-D Systems

The software interprets a $3N(3N+1)/2$-element column vector c as a symmetric matrix. In the following diagram, $\cdot$ means the entry is symmetric.

$$
\begin{pmatrix}
c(1) & c(2) & c(4) & c(7) & c(10) & c(13) & \cdots & c(3(N-1)(3(N-1)+1)/2+1 & c(3(N-1)(3(N-1)+1)/2+4 & c(3(N-1)(3(N-1)+1)/2+7 \\
\cdot & c(3) & c(5) & c(8) & c(11) & c(14) & \cdots & c(3(N-1)(3(N-1)+1)/2+2 & c(3(N-1)(3(N-1)+1)/2+5 & c(3(N-1)(3(N-1)+1)/2+8 \\
\cdot & \cdot & c(6) & c(9) & c(12) & c(15) & \cdots & c(3(N-1)(3(N-1)+1)/2+3 & c(3(N-1)(3(N-1)+1)/2+6 & c(3(N-1)(3(N-1)+1)/2+9 \\
\\
\cdot & \cdot & \cdot & c(16) & c(17) & c(19) & \cdots & c(3(N-1)(3(N-1)+1)/2+10 & c(3(N-1)(3(N-1)+1)/2+13 & c(3(N-1)(3(N-1)+1)/2+16 \\
\cdot & \cdot & \cdot & \cdot & c(18) & c(20) & \cdots & c(3(N-1)(3(N-1)+1)/2+11 & c(3(N-1)(3(N-1)+1)/2+14 & c(3(N-1)(3(N-1)+1)/2+17 \\
\cdot & \cdot & \cdot & \cdot & \cdot & c(21) & \cdots & c(3(N-1)(3(N-1)+1)/2+12 & c(3(N-1)(3(N-1)+1)/2+15 & c(3(N-1)(3(N-1)+1)/2+18 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & c(3N(3N+1)/2-5) & c(3N(3N+1)/2-4) & c(3N(3N+1)/2-2) \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot & c(3N(3N+1)/2-3) & c(3N(3N+1)/2-1) \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdots & \cdot & \cdot & c(3N(3N+1)/2)
\end{pmatrix}
$$

Coefficient $c(i,j,k,l)$, for $i < j$, is in row $(9(j{-}1)(j{-}2)/2 + 6(j{-}1) + 9i + 3l + k - 12)$ of the vector c. For $i = j$, coefficient $c(i,j,k,l)$ is in row $(9(i{-}1)(i{-}2)/2 + 15(i{-}1) + 1/2l(l{-}1) + k)$ of the vector c.

9N²-Element Column Vector c, 3-D Systems

The software interprets a $9N^2$-element column vector c as a matrix.

$$
\begin{pmatrix}
c(1) & c(4) & c(7) & c(3N+1) & c(3N+4) & c(3N+7) & \cdots & c(9N(N-1)+1) & c(9N(N-1)+4) & c(9N(N-1)+7) \\
c(2) & c(5) & c(8) & c(3N+2) & c(3N+5) & c(3N+8) & \cdots & c(9N(N-1)+2) & c(9N(N-1)+5) & c(9N(N-1)+8) \\
c(3) & c(6) & c(9) & c(3N+3) & c(3N+6) & c(3N+9) & \cdots & c(9N(N-1)+3) & c(9N(N-1)+6) & c(9N(N-1)+9) \\
\\
c(10) & c(13) & c(16) & c(3N+10) & c(3N+13) & c(3N+16) & \cdots & c(9N(N-1)+10) & c(9N(N-1)+13) & c(9N(N-1)+16) \\
c(11) & c(14) & c(17) & c(3N+11) & c(3N+14) & c(3N+17) & \cdots & c(9N(N-1)+11) & c(9N(N-1)+14) & c(9N(N-1)+17) \\
c(12) & c(15) & c(18) & c(3N+12) & c(3N+15) & c(3N+18) & \cdots & c(9N(N-1)+12) & c(9N(N-1)+15) & c(9N(N-1)+18) \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
c(3N-8) & c(3N-5) & c(3N-2) & c(6N-8) & c(6N-5) & c(6N-2) & \cdots & c(9N^2-8) & c(9N^2-5) & c(9N^2-2) \\
c(3N-7) & c(3N-4) & c(3N-1) & c(6N-7) & c(6N-4) & c(6N-1) & \cdots & c(9N^2-7) & c(9N^2-4) & c(9N^2-1) \\
c(3N-6) & c(3N-3) & c(3N) & c(6N-6) & c(6N-3) & c(6N) & \cdots & c(9N^2-6) & c(9N^2-3) & c(9N^2)
\end{pmatrix}
$$

Coefficient $c(i,j,k,l)$ is in row $(9N(j{-}1) + 9i + 3l + k - 12)$ of the vector c.

m, d, or a Coefficient for `specifyCoefficients`

> **Note:** **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see "a or d Coefficient for Systems" on page 2-148.

In this section...

"Coefficients m, d, or a" on page 2-143

"Short m, d, or a vectors" on page 2-144

"Nonconstant m, d, or a" on page 2-145

Coefficients m, d, or a

This section describes how to write the **m**, **d**, or **a** coefficients in the system of equations

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

or in the eigenvalue system

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

or

$$-\nabla\cdot(\mathbf{c}\otimes\nabla\mathbf{u}) + \mathbf{a}\mathbf{u} = \lambda^2\mathbf{m}\mathbf{u}.$$

The topic applies to the recommended workflow for including coefficients in your model using `specifyCoefficients`.

If there are *N* equations in the system, then these coefficients represent *N*-by-*N* matrices.

For constant (numeric) coefficient matrices, represent each coefficient using a column vector with N^2 components. This column vector represents, for example, `m(:)`.

For nonconstant coefficient matrices, see "Nonconstant m, d, or a" on page 2-145.

Note: The d coefficient takes a special matrix form when m is nonzero. See "d Coefficient When m is Nonzero" on page 6-398.

Short m, d, or a vectors

Sometimes, your **m**, **d**, or **a** matrices are diagonal or symmetric. In these cases, you can represent **m**, **d**, or **a** using a smaller vector than one with N^2 components. The following sections give the possibilities.

- "Scalar m, d, or a" on page 2-144
- "N-Element Column Vector m, d, or a" on page 2-144
- "N(N+1)/2-Element Column Vector m, d, or a" on page 2-144
- "N^2-Element Column Vector m, d, or a" on page 2-145

Scalar m, d, or a

The software interprets a scalar **m**, **d**, or **a** 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-Element Column Vector m, d, or a

The software interprets an N-element column vector **m**, **d**, or **a** as a diagonal matrix.

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

N(N+1)/2-Element Column Vector m, d, or a

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

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

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

N^2-Element Column Vector m, d, or a

The software interprets an N^2-element column vector **m**, **d**, or **a** 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`.

Nonconstant m, d, or a

Note: If both **m** and **d** are nonzero, then **d** must be a constant scalar or vector, not a function.

If any of the **m**, **d**, or **a** coefficients is not constant, represent it as a function of the form

`dcoeffunction(region,state)`

`solvepde` or `solvepdeeig` pass the `region` and `state` structures to `dcoeffunction`. The function must return a matrix of size $N1$-by-Nr, where:

- $N1$ is the length of the vector representing the coefficient. There are several possible values of $N1$, detailed in "Short m, d, or a vectors" on page 2-144. $1 \leq N1 \leq N^2$.
- Nr is the number of points in the region that the solver passes. Nr is equal to the length of the `region.x` or any other `region` field. The function should evaluate **m**, **d**, or **a** at these points.

Pass the coefficient to `specifyCoefficients` as a function handle, such as

`specifyCoefficients(model,'d',@dcoeffunction,...)`

- `region` is a structure with these fields:

  - `region.x`
  - `region.y`
  - `region.z`
  - `region.subdomain`

  The fields `x`, `y`, and `z` represent the *x*-, *y*-, and *z*- coordinates of points for which your function calculates coefficient values. The `subdomain` field represents the subdomain numbers, which currently apply only to 2-D models. The region fields are row vectors.

- `state` is a structure with these fields:

  - `state.u`
  - `state.ux`
  - `state.uy`
  - `state.uz`
  - `state.time`

  The `state.u` field represents the current value of the solution *u*. The `state.ux`, `state.uy`, and `state.uz` fields are estimates of the solution's partial derivatives ($\partial u/\partial x$, $\partial u/\partial y$, and $\partial u/\partial z$) at the corresponding points of the region structure. The solution and gradient estimates are *N*-by-*Nr* matrices. The `state.time` field is a scalar representing time for time-dependent models.

For example, suppose $N = 3$, and you have 2-D geometry. Suppose your `d` matrix is of the form

$$\mathbf{d} = \begin{bmatrix} 1 & s_1(x,y) & \sqrt{x^2 + y^2} \\ s_1(x,y) & 4 & -1 \\ \sqrt{x^2 + y^2} & -1 & 9 \end{bmatrix},$$

where $s_1(x,y)$ is 5 in subdomain 1, and is 10 in subdomain 2.

This **d** is a symmetric matrix. So it is natural to represent **d** as a "N(N+1)/2-Element Column Vector m, d, or a" on page 2-144:

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

For that form, the following function is appropriate.

```
function dmatrix = dcoeffunction(region,state)

n1 = 6;
nr = numel(region.x);
dmatrix = zeros(n1,nr);
dmatrix(1,:) = ones(1,nr);
dmatrix(2,:) = 5*region.subdomain;
dmatrix(3,:) = 4*ones(1,nr);
dmatrix(4,:) = sqrt(region.x.^2 + region.y.^2);
dmatrix(5,:) = -ones(1,nr);
dmatrix(6,:) = 9*ones(1,nr);
```

To include this function as your **d** coefficient, pass the function handle `@dcoeffunction`:

```
specifyCoefficients(model,'d',@dcoeffunction,...
```

Related Examples

- "f Coefficient for specifyCoefficients" on page 2-101
- "c Coefficient for specifyCoefficients" on page 2-104
- "Solve Problems Using PDEModel Objects" on page 2-14
- "Deflection of a Piezoelectric Actuator" on page 3-19

a or d Coefficient for Systems

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

In this section...
"Coefficients a or d" on page 2-148
"Scalar a or d" on page 2-149
"N-Element Column Vector a or d" on page 2-149
"N(N+1)/2-Element Column Vector a or d" on page 2-149
"N²-Element Column Vector a or d" on page 2-150

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, see "Systems of PDEs" on page 2-65.

Express the coefficients as numbers, text expressions, or functions, as in "f Coefficient for Systems" on page 2-98.

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 "Specify 2-D Scalar Coefficients in Function Form" on page 2-74. 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 2-75.

Often, **a** or **d** have structure, either as symmetric or diagonal. In these cases, you can represent **a** or **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-Element Column Vector a or d

The software interprets an N-element column 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-Element Column Vector a or d

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

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

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

N^2-Element Column Vector a or d

The software interprets an N^2-element column 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.

View, Edit, and Delete PDE Coefficients

Note: **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see the legacy examples on the "PDE Coefficients" page.

View Coefficients

A PDE model stores coefficients in its `EquationCoefficients` property. Suppose `model` is the name of your model. Obtain the coefficients:

```
coeffs = model.EquationCoefficients;
```

To see the active coefficient assignment for a region, call the `findCoefficients` function. For example, create a model and view the geometry.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'SubdomainLabels','on');
ylim([-1.1,1.1])
axis equal
```
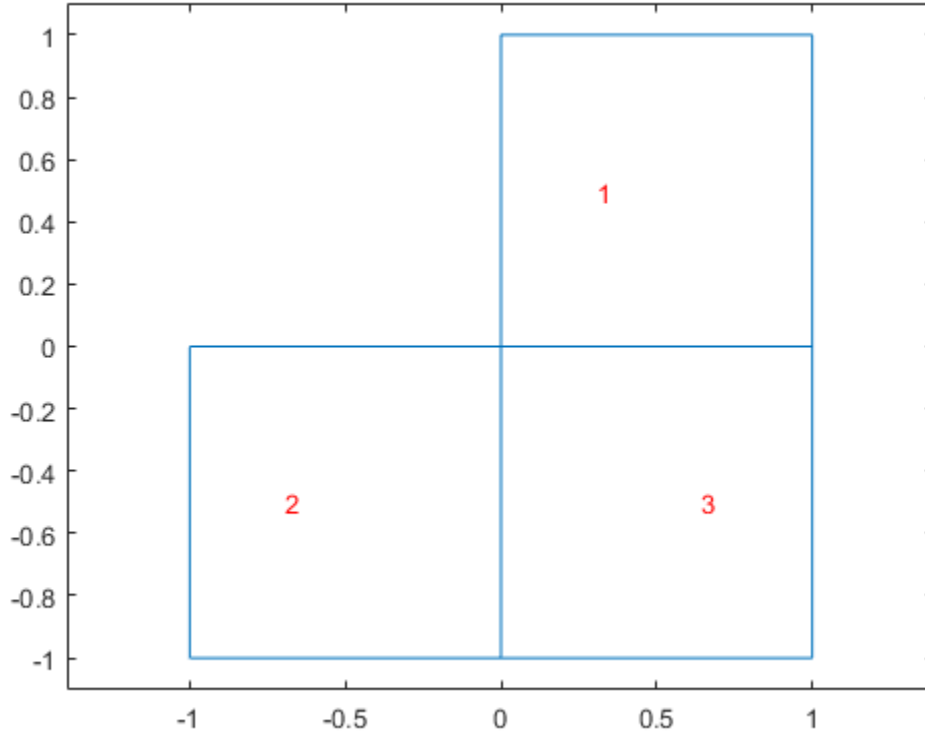
Specify constant coefficients over all the regions in the model.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',2);
```

Specify a different f coefficient on each subregion.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',3,'face',2);
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',4,'face',3);
```

Change the specification to have nonzero a on region 2.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',1,'f',3,'face',2);
```

View the coefficient assignment for region 2.

```
coeffs = model.EquationCoefficients;
findCoefficients(coeffs,'face',2)

ans =

  CoefficientAssignment with properties:

    RegionType: 'face'
      RegionID: 2
             m: 0
             d: 0
             c: 1
             a: 1
             f: 3
```

This shows the "last assignment wins" characteristic.

View the coefficient assignment for region 1.

```
findCoefficients(coeffs,'face',1)

ans =

  CoefficientAssignment with properties:

    RegionType: 'face'
      RegionID: [1 2 3]
             m: 0
             d: 0
             c: 1
             a: 0
             f: 2
```

The active coefficient assignment for region 1 includes all three regions, though this assignment is no longer active for regions 2 and 3.

Delete Existing Coefficients

To delete all the coefficients in your PDE model, use `delete`. Suppose `model` is the name of your model. To remove all coefficients from `model`,

```
delete(model.EquationCoefficients);
```

To delete specific coefficient assignments, delete them from the `model.EquationCoefficients.CoefficientAssignments` vector. For example,

```
coefv = model.EquationCoefficients.CoefficientAssignments;
delete(coefv(2));
```

Tip You do not need to delete coefficients; you can override them by calling `specifyCoefficients` again. However, deleting unused assignments can make your model smaller.

Change a Coefficient Assignment

To change a coefficient assignment, you need the coefficient handle. To get the coefficient handle:

- Retain the handle when using `specifyCoefficients`. For example,

  ```
  coefh1 = specifyCoefficients(model,'m',m,'d',d,'c',c,'a',a,'f',f);
  ```

- Obtain the handle using `findCoefficients`. For example,

  ```
  coeffs = model.EquationCoefficients;
  coefh1 = findCoefficients(coeffs,'face',2);
  ```

You can change any property of the coefficient handle. For example,

```
coefh1.RegionID = [1,3];
coefh1.a = 2;
coefh1.c = @ccoeffun;
```

Note: Editing an existing assignment in this way does not change its priority. For example, if the active coefficient in region 3 was assigned after `coefh1`, then editing `coefh1` to include region 3 does not make `coefh1` the active coefficient in region 3.

Set Initial Conditions

> **Note:** **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see "Solve PDEs with Initial Conditions" on page 2-162.

What Are Initial Conditions?

Initial condition has two meanings:

- For time-dependent problems, the initial condition is the solution u at the initial time, and also the initial time-derivative if the `m` coefficient is nonzero. Set the initial condition in the model using `setInitialConditions`.
- For nonlinear stationary problems, the initial condition is a guess or approximation of the solution u at the initial iteration of the nonlinear solver. Set the initial condition in the model using `setInitialConditions`.

  If you do not specify the initial condition for a stationary problem, `solvepde` uses the zero function for the initial iteration.

Constant Initial Conditions

For a system of N equations, you can give constant initial conditions as either a scalar or as a vector with N components. For example, if the initial condition is $u = 15$ for all components,

```
setInitialConditions(model,15);
```

If $N = 3$, and the initial condition is 15 for the first equation, 0 for the second equation, and −3 for the third equation,

```
u0 = [15,0,-3];
setInitialConditions(model,u0);
```

If the `m` coefficient is nonzero, give an initial condition for the time derivative as well. Set this initial derivative in the same form as the first initial condition. For example, if the initial derivative of the solution is `[4,3,0]`,

```
u0 = [15,0,-3];
```

2-155

```
ut0 = [4,3,0];
setInitialConditions(model,u0,ut0);
```

Nonconstant Initial Conditions

If your initial conditions are not constant, set them by writing a function of the form

```
function u0 = initfun(locations)
```

`solvepde` passes `locations` as a structure with fields `locations.x`, `locations.y`, and, for 3-D problems, `locations.z`. `initfun` must return a matrix `u0` of size *N*-by-M, where *N* is the number of equations in your PDE and `M = length(locations.x)`. The fields in `locations` are row vectors.

For example, suppose you have a 2-D problem with *N* = 2 equations:

$$\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (\nabla u) = \begin{bmatrix} 3 + x \\ 4 - x - y \end{bmatrix}$$

$$u(0) = \begin{bmatrix} 4 + x^2 + y^2 \\ 0 \end{bmatrix}$$

$$\frac{\partial u}{\partial t}(0) = \begin{bmatrix} 0 \\ \sin(xy) \end{bmatrix}.$$

This problem has $m = 1$, $c = 1$, and $f = \begin{bmatrix} 3 + x \\ 4 - x - y \end{bmatrix}$. Because m is nonzero, give both an initial value of *u* and an initial value of the derivative of *u*.

Write the following function files. Save them to a location on your MATLAB path.

```
function uinit = u0fun(locations)

M = length(locations.x);
uinit = zeros(2,M);
uinit(1,:) = 4 + locations.x.^2 + locations.y.^2;

function utinit = ut0fun(locations)

M = length(locations.x);
utinit = zeros(2,M);
```

```
utinit(2,:) = sin(locations.x.*locations.y);
```

Pass the initial conditions to your PDE model:

```
u0 = @u0fun;
ut0 = @ut0fun;
setInitialConditions(model,u0,ut0);
```

Related Examples

- "Solve Problems Using PDEModel Objects" on page 2-14
- "Wave Equation on a Square Domain"
- "Inhomogeneous Heat Equation on a Square Domain"
- "Heat Distribution in a Circular Cylindrical Rod"
- "Solving a Heat Transfer Problem With Temperature-Dependent Properties"
- "Dynamic Analysis of a Clamped Beam"

View, Edit, and Delete Initial Conditions

> **Note:** **THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.** For the corresponding step in the legacy workflow, see the legacy examples on the "Solve PDEs with Initial Conditions" on page 2-162 page.

In this section...
"View Initial Conditions" on page 2-158
"Delete Existing Initial Conditions" on page 2-160
"Change an Initial Conditions Assignment" on page 2-161

View Initial Conditions

A PDE model stores initial conditions in its `InitialConditions` property. Suppose `model` is the name of your model. Obtain the coefficients:

```
inits = model.InitialConditions;
```

To see the active coefficient assignment for a region, call the `findInitialConditions` function. For example, create a model and view the geometry.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'SubdomainLabels','on');
ylim([-1.1,1.1])
axis equal
```
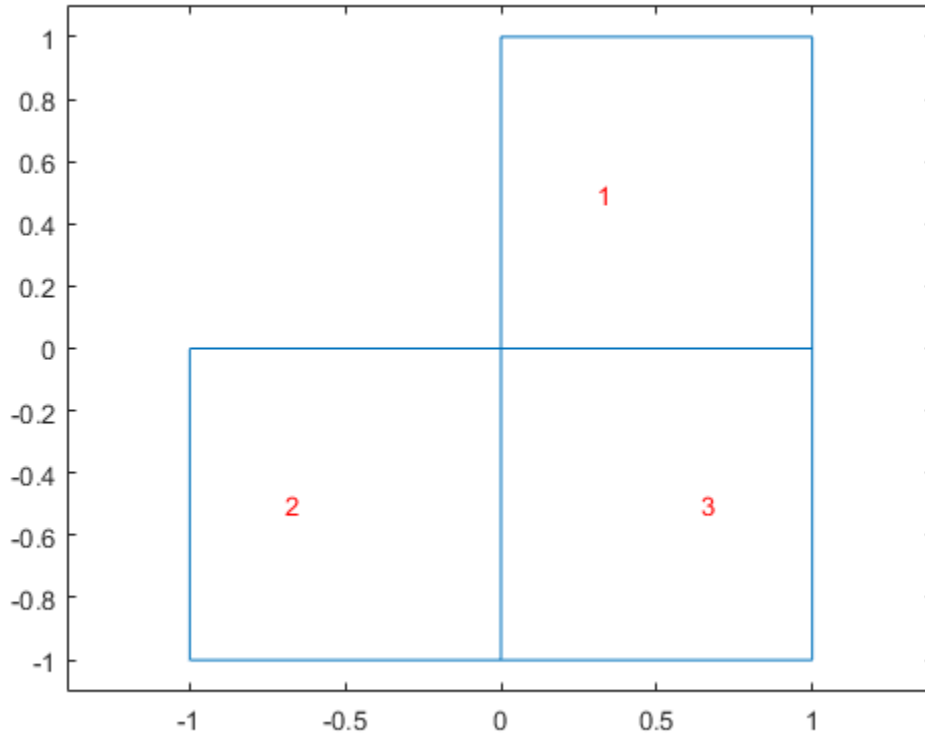
Specify constant initial conditions over all the regions in the model.

```
setInitialConditions(model,2);
```

Specify a different initial condition on each subregion.

```
setInitialConditions(model,3,'face',2);
setInitialConditions(model,4,'face',3);
```

View the coefficient assignment for region 2.

```
ics = model.InitialConditions;
findInitialConditions(ics,'face',2)

ans =
```

2-159

```
GeometricInitialConditions with properties:

        RegionType: 'face'
          RegionID: 2
      InitialValue: 3
 InitialDerivative: []
```

This shows the "last assignment wins" characteristic.

View the initial conditions assignment for region 1.

```
findInitialConditions(ics,'face',1)
```

```
ans =

  GeometricInitialConditions with properties:

        RegionType: 'face'
          RegionID: [1 2 3]
      InitialValue: 2
 InitialDerivative: []
```

The active initial conditions assignment for region 1 includes all three regions, though this assignment is no longer active for regions 2 and 3.

Delete Existing Initial Conditions

To delete all the initial conditions in your PDE model, use delete. Suppose model is the name of your model. To remove all initial conditions from model,

```
delete(model.InitialConditions);
```

To delete specific initial conditions assignments, delete them from the model.InitialConditions.InitialConditionAssignments vector. For example,

```
icv = model.InitialConditions.InitialConditionAssignments;
delete(icv(2));
```

Tip You do not need to delete initial conditions; you can override them by calling setInitialConditions again. However, deleting unused assignments can make your model smaller.

Change an Initial Conditions Assignment

To change an initial conditions assignment, you need the initial conditions handle. To get the coefficient handle:

- Retain the handle when using `setInitialConditions`. For example,

  ```
  ics1 = setInitialConditions(model,2);
  ```

- Obtain the handle using `findInitialConditions`. For example,

  ```
  ics = model.InitialConditions;
  ics1 = findInitialConditions(ics,'face',2);
  ```

You can change any property of the initial conditions handle. For example,

```
ics1.RegionID = [1,3];
ics1.InitialValue = 2;
ics1.InitialDerivative = @utOfun;
```

Note: Editing an existing assignment in this way does not change its priority. For example, if the active initial conditions in region 3 was assigned after `ics1`, then editing `ics1` to include region 3 does not make `ics1` the active initial condition in region 3.

Solve PDEs with Initial Conditions

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "Set Initial Conditions" on page 2-155.

What Are Initial Conditions?

Initial conditions has two meanings:

- For the `parabolic` and `hyperbolic` solvers, the initial condition `u0` is the solution *u* at the initial time. You must specify the initial condition for these solvers. Pass the initial condition in the first argument or arguments.

  ```
  u = parabolic(u0,...
  or
  u = hyperbolic(u0,ut0,...
  ```

  For the `hyperbolic` solver, you must also specify `ut0`, which is the value of the derivative of *u* with respect to time at the initial time. `ut0` has the same form as `u0`.

- For nonlinear elliptic problems, the initial condition `u0` is a guess or approximation of the solution *u* at the initial iteration of the `pdenonlin` nonlinear solver. You pass `u0` in the `'U0'` name-value pair.

  ```
  u = pdenonlin(b,p,e,t,c,a,f,'U0',u0)
  ```

  If you do not specify initial conditions, `pdenonlin` uses the zero function for the initial iteration.

Constant Initial Conditions

You can specify initial conditions as a constant by passing a scalar or string vector.

- For scalar problems or systems of equations, give a scalar as the initial condition. For example, set `u0` to `5` for an initial condition of 5 in every component.
- For systems of *N* equations, give a string vector initial condition with *N* rows. For example, if there are *N* = 3 equations, you can give initial conditions `u0 = char('3','-3','0')`.

Initial Conditions in String Form

You can specify text expressions for the initial conditions. The initial conditions are functions of x and y alone, and, for 3-D problems, z. The text expressions represent vectors at nodal points, so use `.*` for multiplication, `./` for division, and `.^` for exponentiation.

For example, if you have an initial condition

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

then you can use this expression for the initial condition.

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

For a system of $N > 1$ equations, use a text array with one row for each component, such as

```
char('x.^2 + 5*cos(x.*y)',...
    'tanh(x.*y)./(1 + z.^2)')
```

Initial Conditions at Mesh Nodes

Pass `u0` as a column vector of values at the mesh nodes. The nodes are either `model.Mesh.Nodes`, or the `p` data from `initmesh` or `meshToPet`. See "Mesh Data" on page 2-211.

Tip For reliability, the initial conditions and boundary conditions should be consistent.

The size of the column vector `u0` depends on the number of equations, N, and on the number of nodes in the mesh, N_p.

For scalar u, specify a column vector of length N_p. The value of element `k` corresponds to the node `p(k)`.

For a system of N equations, specify 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 node p(k). The next N_p points contain the values of component 2, etc. It can be convenient to first 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 a function myfun(x,y) that calculates the value of the initial condition u0(x,y) as a row vector of length *N* for a 2-D problem. Suppose that p is the usual mesh node data (see "Mesh Data" on page 2-211). Compute the initial conditions for all mesh nodes p.

```
% Assume N and p exist; N = 1 for a scalar problem
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
```

Specify u0 as the initial condition.

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 1-26. In the weak formulation you do not specify boundary conditions between subdomains, even if coefficients are discontinuous between subdomains. So Partial Differential Equation Toolbox does 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.
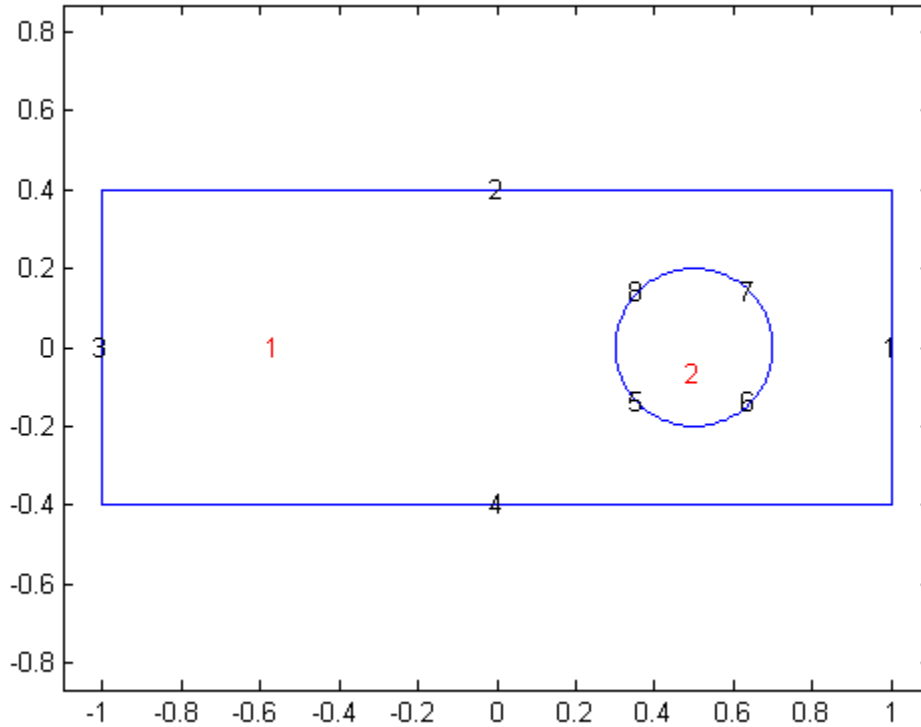
Code for generating the figure

```
% 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.

Identify Boundary Labels

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.**

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 3-D problems, set the `FaceLabels` name-value pair to `'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')
```
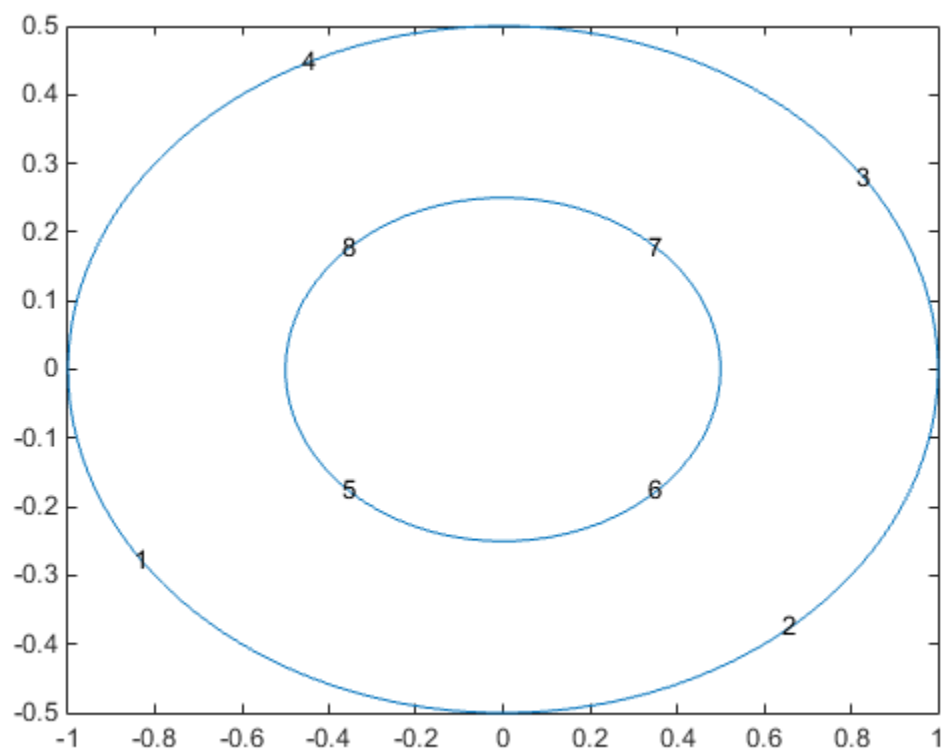
Forms of Boundary Condition Specification

There are three forms of boundary condition specifications:

- `BoundaryCondition` object — Use this form to specify boundary conditions for a `PDEModel` in a modular fashion. You can specify conditions separately for each edge or set of edges. This form allows simple specification of piecewise constant Dirichlet or Neumann boundary conditions, and also allows general functional forms of boundary conditions. For 3-D geometry, you must use this form. For details, see "Specify Boundary Conditions Objects" on page 2-179.

- Boundary matrix (2-D only) — Generally, do not attempt to write a boundary matrix manually. The main use of this form is as an export from the PDE app. For details on the matrix, see "Boundary Matrix for 2-D Geometry" on page 2-171.

- Boundary file (2-D only) — You can write a function to give boundary conditions as a function of the x and y coordinates, the solution u, and time. This syntax is not recommended, and exists primarily to support legacy code. To write a function in this form, see "Boundary Conditions by Writing Functions" on page 2-199.

Boundary Matrix for 2-D Geometry

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "Specify Boundary Conditions Objects" on page 2-179.

Boundary Matrix Specification

The Boundary Condition matrix is created internally in the PDE app (actually a function called by the PDE app) 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 as a boundary file for later use with "Boundary Conditions by Writing Functions" on page 2-199.

For each column in the Decomposed Geometry matrix (see "Decomposed Geometry Data Structure" on page 2-24) there must be a corresponding column in the Boundary Condition matrix. The format of each column is:

- 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 **q**. The lengths are stored in column-wise order with respect to **q**.

- Row $3 + N^2$ to $3 + N^2 + N - 1$ contain the lengths for the strings representing **g**.

- Row $3 + N^2 + N$ to $3 + N^2 + N + MN - 1$ contain the lengths for the strings representing **h**. The lengths are stored in column-wise order with respect to **h**.

- Row $3 + N^2 + N + MN$ to $3 + N^2 + N + MN + M - 1$ contain the lengths for the strings representing **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 column-wise 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.

One Column of a Boundary Matrix

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 ...
```

`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 the PDE app. 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.

Create Boundary Condition Matrices Programmatically

The following example shows you how to create 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;
```

2-173

```
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.

Related Examples

- "Specify Boundary Conditions Objects" on page 2-179

Classification of Boundary Conditions

Boundary Conditions for Scalar PDEs

For scalar PDEs, there are two choices of boundary conditions for each edge:

- Dirichlet — On the edge, the solution u satisfies the equation $hu = r$,

  where h and r can be functions of space (x and y), the solution u, and time. Often, you take $h = 1$, and set r to the appropriate value.

- Generalized Neumann boundary conditions — On the edge the solution u satisfies the equation

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

  The coefficient c is the same as the coefficient of the second-order differential operator in the PDE equation

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

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

To incorporate these conditions into your problem, see "Specify Boundary Conditions Objects" on page 2-179.

Boundary Conditions for Systems of PDEs

For systems of PDEs, there are generalized versions of the Dirichlet and Neumann boundary conditions:

- **hu** = **r** represents a matrix **h** multiplying the solution vector **u**, and equaling the vector **r**.

- $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{q}\mathbf{u} = \mathbf{g} + \mathbf{h}'\mu$, where 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 a total of N boundary conditions. The generalized Neumann condition contains a source $\mathbf{h}'\mu$, where the solver computes Lagrange multipliers μ such that the Dirichlet conditions are satisfied.

To incorporate these conditions into your problem, see "Specify Boundary Conditions Objects" on page 2-179.

Specify Boundary Conditions Objects

Note: SPECIFYING BOUNDARY CONDITIONS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Before you create boundary conditions, you need to create a `PDEModel` container. For details, see "Solve Problems Using Legacy PDEModel Objects" on page 2-11. These steps assume that you have a container named `model`, and that the geometry is stored in `model`.

1 Examine the geometry to see the label of each edge or face.

```
pdegplot(model,'EdgeLabels','on') % for 2-D
pdegplot(model,'FaceLabels','on') % for 3-D
```

For an example, see "Create and View 3-D Geometry" on page 2-47.

2 Specify the boundary conditions for each edge or face. A Dirichlet boundary condition is either the value of the solution u on that boundary, or it is a pair of parameters, h and r, that mean u satisfies the equation
$hu = r$.

Usually, it is easier and less error-prone to specify u directly rather than using h and r.

Neumann boundary conditions are, for given parameters q and g, the equation

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

The c argument is the same as in the PDE coefficient. See "Equations You Can Solve Using Recommended Functions" on page 1-6. For details of boundary conditions, see "Classification of Boundary Conditions" on page 2-177.

If you do not specify a boundary condition for an edge or face, the default is Neumann, with zero values for `'g'` and `'q'`.

3 For systems of equations, specify *mixed boundary conditions*, meaning different types of boundary conditions for each component of the solution on each boundary. See "Dirichlet Boundary Conditions for Systems Using u and EquationIndex" on

page 2-182. For details on Neumann boundary conditions, which use a more general syntax, see "Neumann Boundary Conditions for Systems" on page 2-183.

4 For each edge or face of the model, set a Dirichlet or Neumann boundary condition using `applyBoundaryCondition`. If you have a system of PDEs, you can set a different boundary condition for each component on each boundary edge or face.

- If the boundary condition is a constant Dirichlet or generalized Neumann condition, set boundary conditions by using the syntax in "Specify Constant Boundary Conditions" on page 2-181. These examples specify a Dirichlet condition with value `v` on boundary IDs `bid`.

```
applyBoundaryCondition(model,'face',bid,'u',v); % 3-D geometry
applyBoundaryCondition(model,'edge',bid,'u',v); % 2-D geometry
```

These examples specify Neumann conditions with values `g` and `q`.

```
applyBoundaryCondition(model,'face',bid,'g',g,'q',q); % 3-D geometry
applyBoundaryCondition(model,'edge',bid,'g',g,'q',q); % 2-D geometry
```

For systems of equations, see "Specify Constant Boundary Conditions" on page 2-181.

- If the boundary condition is a function of position, time, or the solution u, set boundary conditions by using the syntax in "Specify Nonconstant Boundary Conditions" on page 2-190.

Related Examples

Specify Constant Boundary Conditions

Note: SPECIFYING BOUNDARY CONDITIONS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Boundary Condition Parameters

Specify Dirichlet boundary conditions for edges or faces by setting the `'u'` argument in `applyBoundaryCondition`. You can also specify Dirichlet boundary conditions by giving parameters h and r for the equation
$hu = r$,

where u is the value of the solution on the edge. Usually, it is easier and less error-prone to use the `'u'` argument than the `'h'` and `'r'` arguments.

Specify Neumann boundary conditions for edges or faces by giving parameters q and g for the equation

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

For details, see "Classification of Boundary Conditions" on page 2-177.

If you do not specify a boundary condition for an edge or face, the default is Neumann with the default zero values for `'g'` and `'q'`. See "Input Arguments" on page 6-239.

Scalar Dirichlet Boundary Conditions Using u

Suppose that you have a PDE model named `model`, and edge or face labels `[e1,e2,e3]` where the solution u must equal a constant `C0`. Express this boundary condition as follows.

```
% For 3-D geometry:
applyBoundaryCondition(model,'face',[e1,e2,e3],'u',C0);
% For 2-D geometry:
applyBoundaryCondition(model,'edge',[e1,e2,e3],'u',C0);
```

Scalar Neumann Boundary Conditions

Suppose that you have a PDE model named `model`, and edge or face labels `[e1,e2,e3]` where the solution u must satisfy

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

where q and g are constants `Q0` and `G0` respectively, and c is the coefficient of the second-order differential operator in the PDE equation. Express this boundary condition as follows.

```
% For 3-D geometry:
applyBoundaryCondition(model,'face',[e1,e2,e3],'q',Q0,'g',G0);
% For 2-D geometry:
applyBoundaryCondition(model,'edge',[e1,e2,e3],'q',Q0,'g',G0);
```

Dirichlet Boundary Conditions for Systems Using u and EquationIndex

Suppose that you have a PDE model named `model`, and edge or face labels `[e1,e2,e3]` where the second and third components of the solution u must equal a constant `C0`. Express this boundary condition as follows.

```
% For 3-D geometry:
applyBoundaryCondition(model,'face',[e1,e2,e3],'u',[C0,C0],'EquationIndex',[2,3]);
% For 2-D geometry:
applyBoundaryCondition(model,'edge',[e1,e2,e3],'u',[C0,C0],'EquationIndex',[2,3]);
```

If the second component must equal `C0` and the third component must equal `C1`:

```
% For 3-D geometry:
applyBoundaryCondition(model,'face',[e1,e2,e3],'u',[C0,C1],'EquationIndex',[2,3]);
% For 2-D geometry:
applyBoundaryCondition(model,'edge',[e1,e2,e3],'u',[C0,C1],'EquationIndex',[2,3]);
```

- Generally, the `'u'` and `'EquationIndex'` arguments must have the same length.
- If you exclude the `'EquationIndex'` argument, the `'u'` argument must have length `N`.
- If you exclude the `'u'` argument, `applyBoundaryCondition` sets the components in `'EquationIndex'` to `0`.

Dirichlet Boundary Conditions for Systems Using the (r,h) Pair

Suppose that you have a PDE model named `model`, and edge or face labels `[e1,e2,e3]` where the second and third components of the solution u must equal a constant `C0`. Express this boundary condition as follows.

```
H0 = [0,0,0;
    0,1,0;
    0,0,1];
R0 = [0;C0;C0];
% For 3-D geometry:
applyBoundaryCondition(model,'face',[e1,e2,e3],'h',H0,'r',R0);
% For 2-D geometry:
applyBoundaryCondition(model,'edge',[e1,e2,e3],'h',H0,'r',R0);
```

If the second component must equal `C0` and the third component must equal `C1`, use the following code.

```
H0 = [0,0,0;
    0,1,0;
    0,0,1];
R0 = [0;C0;C1];
% For 3-D geometry:
applyBoundaryCondition(model,'face',[e1,e2,e3],'h',H0,'r',R0);
% For 2-D geometry:
applyBoundaryCondition(model,'edge',[e1,e2,e3],'h',H0,'r',R0);
```

- The `'r'` parameter must be a numeric vector of length N. If you do not include `'r'`, `applyBoundaryCondition` sets the values to `0`.

- The `'h'` parameter can be an N-by-N numeric matrix or a vector of length N^2 corresponding to the "Linear Indexing" form of the N-by-N matrix. If you do not include `'h'`, `applyBoundaryCondition` sets the value to the identity matrix.

Neumann Boundary Conditions for Systems

Suppose that you have a geometry container named `pg`, and edge or face labels `[e1,e2,e3]` where the solution u must satisfy

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

where **q** and **g** are constant matrices Q0 and G0 respectively, and **c** is the coefficient of the second-order differential operator in the PDE equation. Express this boundary condition as follows.

```
% For 3-D geometry:
applyBoundaryCondition(model,'face',[e1,e2,e3],'q',Q0,'g',G0);
% For 2-D geometry:
applyBoundaryCondition(model,'edge',[e1,e2,e3],'q',Q0,'g',G0);
```

- The 'g' parameter must be a numeric vector of length N. If you do not include 'g', applyBoundaryCondition sets the values to 0.

- The 'q' parameter can be an N-by-N numeric matrix or a vector of length N^2 corresponding to the "Linear Indexing" form of the N-by-N matrix. If you do not include 'q', applyBoundaryCondition sets the values to 0.

Related Examples

- "Solve PDEs with Constant Boundary Conditions" on page 2-185
- "Specify Boundary Conditions Objects" on page 2-179
- "Solve Problems Using PDEModel Objects" on page 2-14

Solve PDEs with Constant Boundary Conditions

Note: SPECIFYING BOUNDARY CONDITIONS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

This example shows how to apply various constant boundary condition specifications for both scalar PDEs and systems of PDEs.

Geometry

All the specifications use the same 2-D geometry, which is a rectangle with a circular hole.
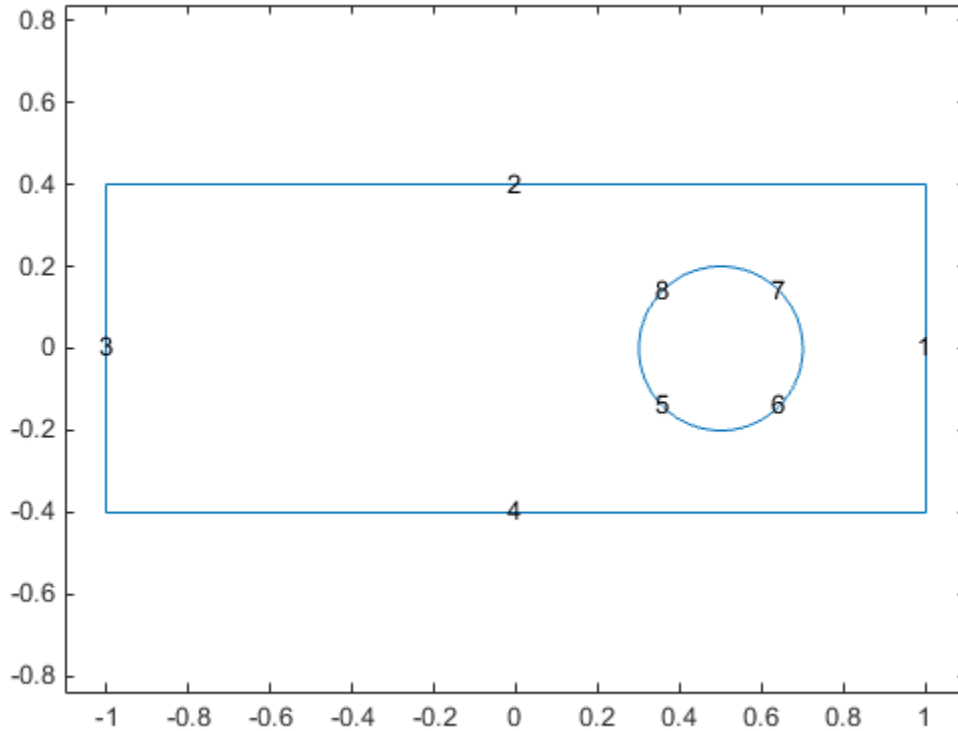
```
% Rectangle is code 3, 4 sides, followed by x-coordinates and then y-coordinates
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
% Circle is code 1, center (.5,0), radius .2
C1 = [1,.5,0,.2]';
% Pad C1 with zeros to enable concatenation with R1
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];

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

% Set formula
sf = 'R1 - C1';

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

% Create geometry model
model = createpde;

% Include the geometry in the model and view the geometry
geometryFromEdges(model,g);
pdegplot(model,'EdgeLabels','on');
xlim([-1.1 1.1])
axis equal
```

Scalar Problem

Suppose that edge 3 has Dirichlet conditions with value 32, edge 1 has Dirichlet conditions with value 72, and all other edges have Neumann boundary conditions with $q = 0, g = -1$.

```
applyBoundaryCondition(model,'edge',3,'u',32);
applyBoundaryCondition(model,'edge',1,'u',72);
applyBoundaryCondition(model,'edge',[2,4:8],'g',-1);
```

This completes the boundary condition specification.

Solve an elliptic PDE with these boundary conditions with $c = 1$, $a = 0$, and $f = 10$. Because the shorter rectangular side has length 0.8, to ensure that the mesh is not too coarse choose a maximum mesh size $\texttt{Hmax} = 0.1$.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',10);
generateMesh(model,'Hmax',0.1);
results = solvepde(model);
u = results.NodalSolution;
pdeplot(model,'xydata',u,'zdata',u)
view(-23,8)
```
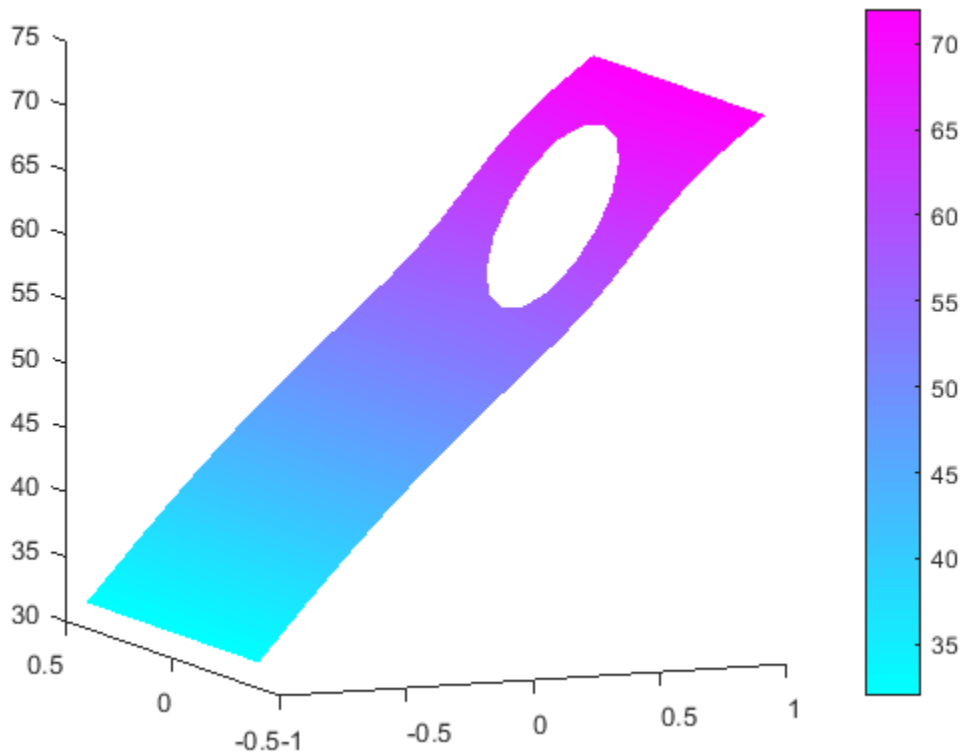

System of PDEs

Suppose that the system has $N = 2$.

- Edge 3 has Dirichlet conditions with values `[32,72]`.

- Edge 1 has Dirichlet conditions with values `[72,32]`.

- Edge 4 has a Dirichlet condition for the first component with value `52`, and has a Neumann condition for the second component with $q = 0$, $g = -1$.

- Edge 2 has Neumann boundary conditions with $q = [1,2;3,4]$ and $g = [5,-6]$.

- The circular edges (edges 5 through 8) have $q = 0$ and $g = 0$.

```
model = createpde(2);
geometryFromEdges(model,g);

applyBoundaryCondition(model,'edge',3,'u',[32,72]);
applyBoundaryCondition(model,'edge',1,'u',[72,32]);
applyBoundaryCondition(model,'edge',4,'u',52,'EquationIndex',1);
applyBoundaryCondition(model,'edge',4,'g',[0,-1]);
Q2 = [1,2;3,4];
G2 = [5,-6];
applyBoundaryCondition(model,'edge',2,'q',Q2,'g',G2);
% The next step is optional, because it sets 'g' to its default value
applyBoundaryCondition(model,'edge',5:8,'g',[0,0]);
```

This completes the boundary condition specification.

Solve an elliptic PDE with these boundary conditions using $c = 1$, $a = 0$, and $f = [10;-10]$. Because the shorter rectangular side has length 0.8, to ensure that the mesh is not too coarse choose a maximum mesh size `Hmax = 0.1`.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f', [10;-10]);
generateMesh(model,'Hmax',0.1);
results = solvepde(model);
u = results.NodalSolution;
pdeplot(model,'xydata',u(:,2),'zdata',u(:,2));
```
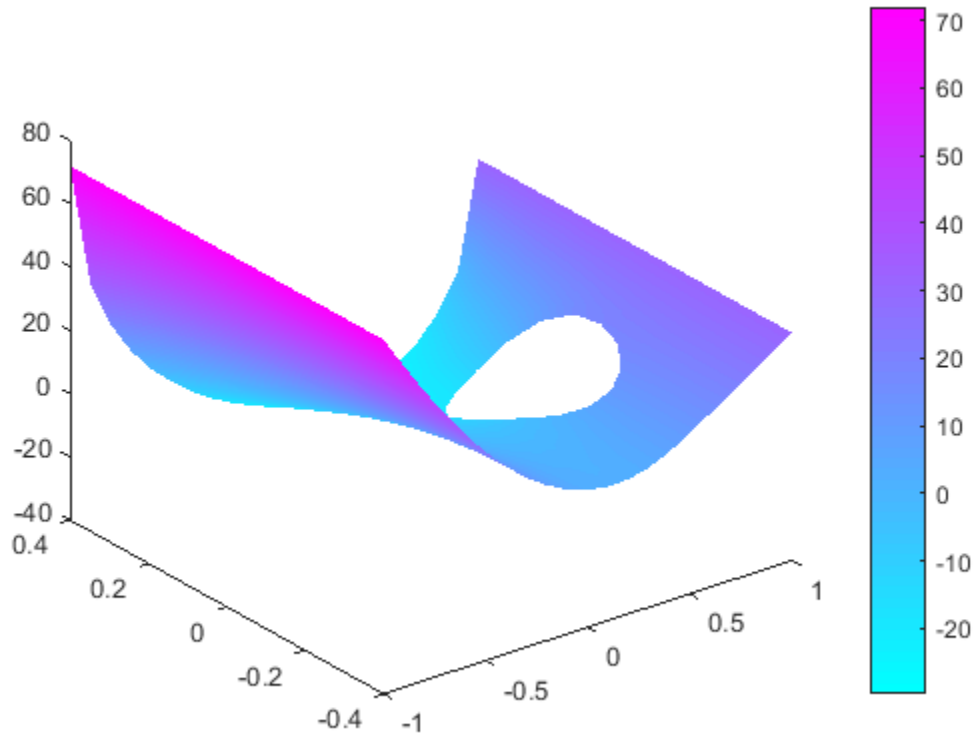
More About

- "Specify Boundary Conditions Objects" on page 2-179
- "Specify Constant Boundary Conditions" on page 2-181
- "Solve PDEs with Nonconstant Boundary Conditions" on page 2-193

Specify Nonconstant Boundary Conditions

Note: **SPECIFYING BOUNDARY CONDITIONS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.**

Specify Dirichlet boundary conditions for edges or faces by setting the `'u'` argument in `applyBoundaryCondition`. You can also specify Dirichlet boundary conditions by giving parameters h and r for the equation
$hu = r$,

where u is the value of the solution on the edge. Usually, it is easier and less error-prone to use the `'u'` argument than the `'h'` and `'r'` arguments.

Specify Neumann boundary conditions for edges or faces by giving parameters q and g for the equation

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

For details, see "Classification of Boundary Conditions" on page 2-177.

If you do not specify a boundary condition for an edge or face, the default is Neumann with the default zero values for `'g'` and `'q'`. See "Input Arguments" on page 6-239.

When you cannot express your boundary conditions by constant input arguments, write functions.

```
applyBoundaryCondition(model,'edge',1,'r',@myrfun);
applyBoundaryCondition(model,'face',2,'g',@mygfun,'q',@myqfun);
applyBoundaryCondition(model,'edge',[3,4],'u',@myufun,'EquationIndex',[2,3]);
```

Each function must have the following syntax.

```
function bcMatrix = myfun(region,state)
```

Partial Differential Equation Toolbox solvers pass the `region` and `state` data to your function.

- `region` — A structure containing the following fields. If you pass a name-value pair to `applyBoundaryCondition` with `Vectorized` set to `'on'`, then `region` can

contain several evaluation points. If you do not set `Vectorized`, or set it to `'off'`, then solvers pass just one evaluation point at a time.

- `region.x` — The *x*-coordinate of the point or points
- `region.y` — The *y*-coordinate of the point or points
- `region.z` — For 3-D geometry, the *z*-coordinate of the point or points

Furthermore, if there are Neumann conditions, then solvers pass the following data in the `region` structure.

- `region.nx` — *x*-component of the normal vector at the evaluation point or points
- `region.ny` — *y*-component of the normal vector at the evaluation point or points
- `region.nz` — For 3-D geometry, *z*-component of the normal vector at the evaluation point or points
- `state` — For transient or nonlinear problems.

  - `state.u` contains the solution vector at evaluation points. `state.u` is an N-by-M matrix, where each column corresponds to one evaluation point, and M is the number of evaluation points.
  - `state.time` contains the time at evaluation points. `state.time` is a scalar.

Your function returns `bcMatrix`. This matrix has the following form, depending on the boundary condition type.

- `'u'` — N1-by-M matrix, where each column corresponds to one evaluation point, and M is the number of evaluation points. N1 is the length of the `'EquationIndex'` argument (see EquationIndex). If there is no `'EquationIndex'` argument, then N1 = N.
- `'r'` or `'g'` — N-by-M matrix, where each column corresponds to one evaluation point, and M is the number of evaluation points.
- `'h'` or `'q'` — N^2-by-M matrix, where each column corresponds to one evaluation point via "Linear Indexing" of the underlying N-by-N matrix, and M is the number of evaluation points. Alternatively, an N-by-N-by-M array, where each evaluation point is an N-by-N matrix.

Related Examples

- "Solve PDEs with Nonconstant Boundary Conditions" on page 2-193

- "Specify Boundary Conditions Objects" on page 2-179
- "Solve Problems Using PDEModel Objects" on page 2-14

Solve PDEs with Nonconstant Boundary Conditions

Note: **SPECIFYING BOUNDARY CONDITIONS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.**

This example shows how to write functions for a nonconstant boundary condition specification. All the specifications use the same geometry, which is a rectangle with a circular hole.

```
% 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
g = decsg(geom,sf,ns);

% Create geometry model
model = createpde;

% Include the geometry in the model and view the geometry
geometryFromEdges(model,g);
pdegplot(model,'EdgeLabels','on');
xlim([-1.1 1.1])
axis equal
```

Scalar Problem

- Edge 3 has Dirichlet conditions with value 32.
- Edge 1 has Dirichlet conditions with value 72.
- Edges 2 and 4 have Dirichlet conditions that linearly interpolate between edges 1 and 3.
- The circular edges (5 through 8) have Neumann conditions with $q = 0$, $g = -1$.

```
applyBoundaryCondition(model,'edge',3,'u',32);
applyBoundaryCondition(model,'edge',1,'u',72);
applyBoundaryCondition(model,'edge',5:8,'g',-1); % q = 0 by default
```

Edges 2 and 4 need functions that perform the linear interpolation. Each edge can use the same function that returns the value
$u(x,y) = 52 + 20*x$.

You can implement this simple interpolation in an anonymous function.

```
myufun = @(region,state)52 + 20*region.x;
```

Include the function for edges 2 and 4. To help speed the solver, allow a vectorized evaluation.

```
applyBoundaryCondition(model,'edge',[2,4],'u',myufun,'Vectorized','on');
```

Solve an elliptic PDE with these boundary conditions, using the parameters $c = 1$, $a = 0$, and $f = 10$. Because the shorter rectangular side has length 0.8, to ensure that the mesh is not too coarse choose a maximum mesh size Hmax = 0.1.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',10);
generateMesh(model,'Hmax',0.1);
results = solvepde(model);
u = results.NodalSolution;
pdeplot(model,'xydata',u)
```
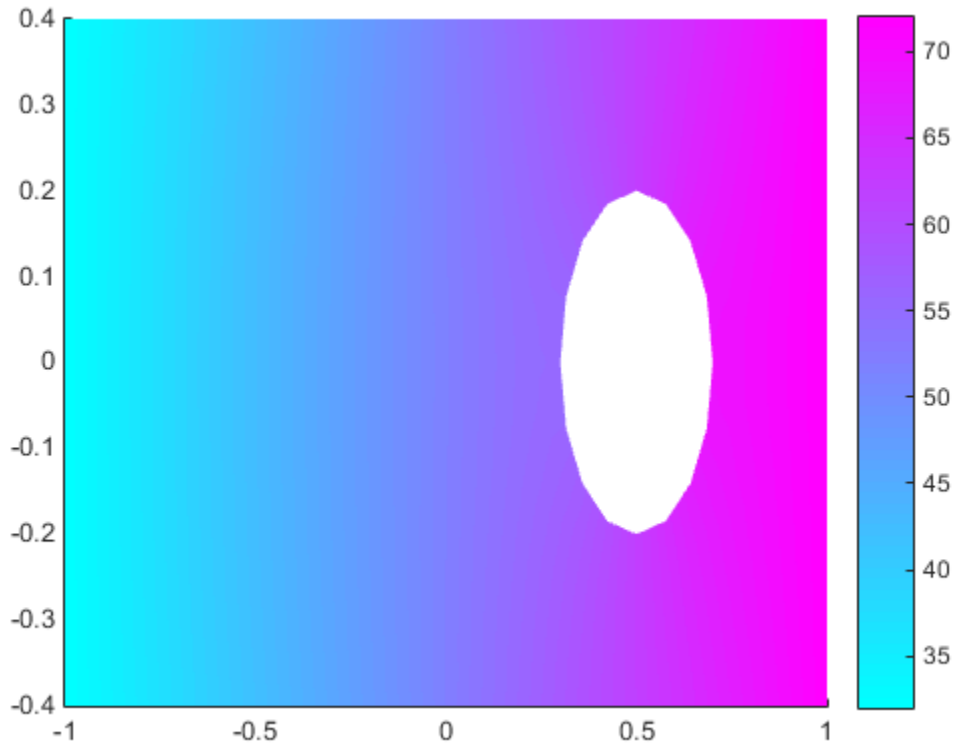
System of PDEs

Suppose that the system has N = 2.

- Edge 3 has Dirichlet conditions with values `[32,72]`.
- Edge 1 has Dirichlet conditions with values `[72,32]`.
- Edges 2 and 4 have Dirichlet conditions that interpolate between the conditions on edges 1 and 3, and include a sinusoidal variation.
- Circular edges (edges 5 through 8) have $q = 0$ and $g = -10$

```
model = createpde(2);
```

```
geometryFromEdges(model,g);

applyBoundaryCondition(model,'edge',3,'u',[32,72]);
applyBoundaryCondition(model,'edge',1,'u',[72,32]);
applyBoundaryCondition(model,'edge',5:8,'g',[-10,-10]);
```

The first component of edges 2 and 4 satisfies the equation
$u_1(x) = 52 + 20*x + 10*\sin(\pi x^3)$.

The second component satisfies
$u_2(x) = 52 - 20*x - 10*\sin(\pi x^3)$.

Write a function file `myufun.m` that incorporates these equations in the syntax from
"Specify Nonconstant Boundary Conditions" on page 2-190.

```
function bcMatrix = myufun(region,state)
bcMatrix = [52 + 20*region.x + 10*sin(pi*(region.x.^3));
    52 - 20*region.x - 10*sin(pi*(region.x.^3))]; % OK to vectorize
```

Include this function in the edge 2 and edge 4 boundary condition.

```
clear myufun % In case you have myufun in your workspace from the scalar case
applyBoundaryCondition(model,'edge',[2,4],'u',@myufun,'Vectorized','on');
```

Solve an elliptic PDE with these boundary conditions, with the parameters $c = 1$, $a = 0$,
and $f = (10,-10)$. Because the shorter rectangular side has length 0.8, to ensure that the
mesh is not too coarse choose a maximum mesh size `Hmax = 0.1`.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',[10;-10]);
generateMesh(model,'Hmax',0.1);
results = solvepde(model);
u = results.NodalSolution;

subplot(1,2,1)
pdeplot(model,'xydata',u(:,1),'zdata',u(:,1),'colorbar','off')
view(-9,24)
subplot(1,2,2)
pdeplot(model,'xydata',u(:,2),'zdata',u(:,2),'colorbar','off')
view(-9,24)
```
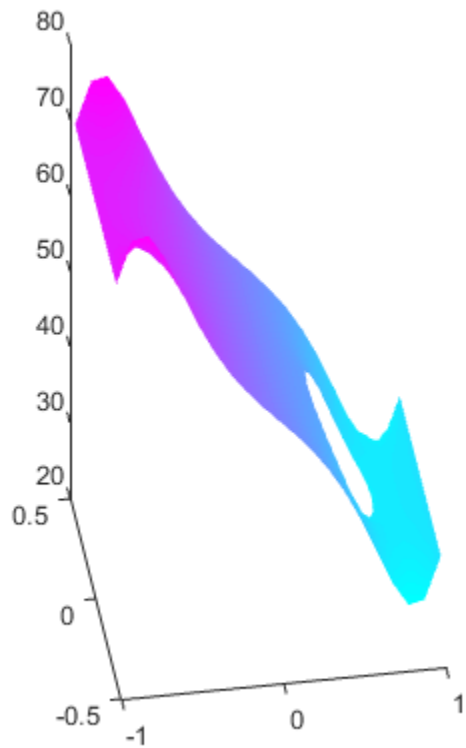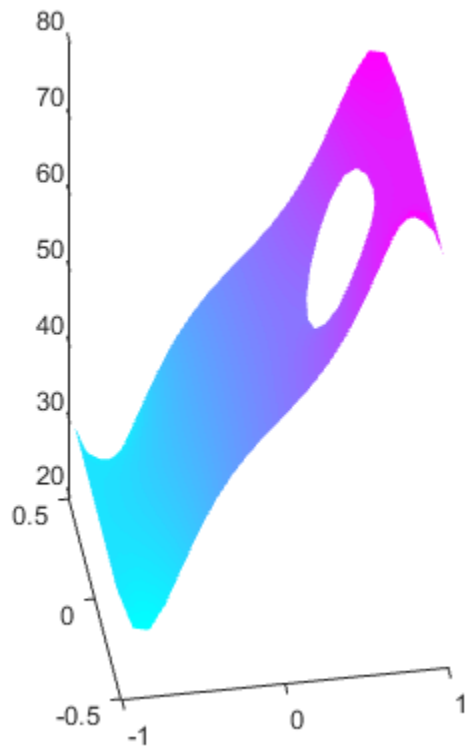
Boundary Conditions by Writing Functions

Note: **THIS PAGE DESCRIBES THE LEGACY WORKFLOW.** New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see "Specify Boundary Conditions Objects" on page 2-179.

About Boundary Conditions by Writing Functions

This section shows how to express boundary conditions for 2-D geometry using the legacy function syntax. However, the recommended way to express boundary conditions is to use "Specify Boundary Conditions Objects" on page 2-179.

To use this legacy syntax, write the functions using the templates in "Boundary Conditions for Scalar PDE" on page 2-199 or "Boundary Conditions for PDE Systems" on page 2-204.

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.

To write a function file, say `pdebound.m`, use 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 2-211)
- e — Finite element edges in the mesh, a subset of all the edges ("Mesh Data" on page 2-211)
- 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 2-168.

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 2-168).

    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;
```

```matlab
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:

```matlab
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:

```matlab
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.
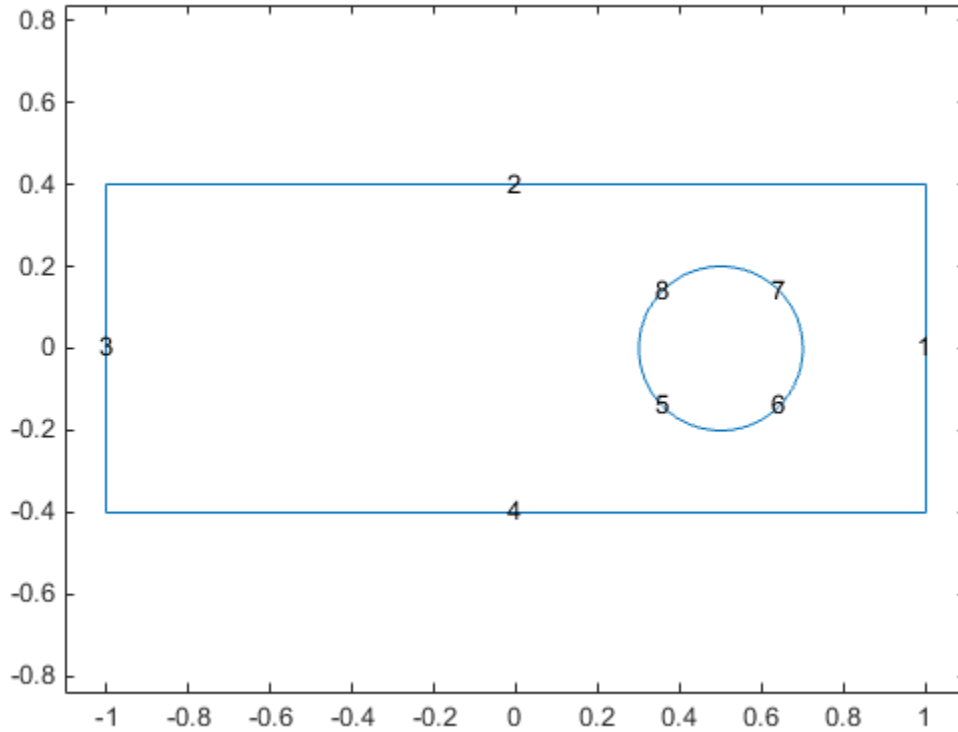
2-201

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'))';
```

```matlab
% 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:

```matlab
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
```

Boundary Conditions for PDE Systems

The general mixed-boundary conditions for PDE systems of N equations (see "Systems of PDEs" on page 2-65) 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.

To write a function file, say `pdebound.m`, use 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 2-211)
- `e` — Finite element edges in the mesh, a subset of all the edges ("Mesh Data" on page 2-211)
- `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 2-168.

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 2-168).

  `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 2-65. 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.
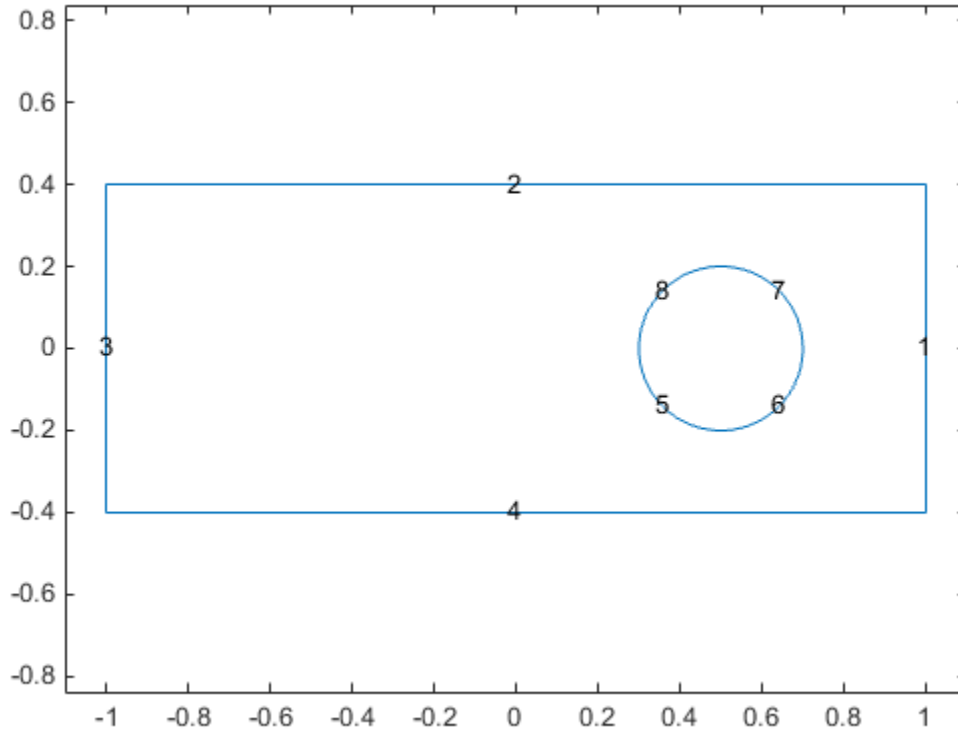
Code for generating the figure

```matlab
% 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'))';
```

2-207

```
% 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
```

Mesh Data

In this section...
"What Is Mesh Data?" on page 2-211
"Mesh Data for FEMesh" on page 2-211
"Mesh Data for [p,e,t] Triples: 2-D" on page 2-211
"Mesh Data for [p,e,t] Triples: 3-D" on page 2-212

What Is Mesh Data?

A *mesh* consists of either an FEMesh object or a [p,e,t] triple.

- Create a FEMesh object using generateMesh. This object is the Mesh property of the PDEModel object.
- For a 2-D mesh created using initmesh, the mesh is a [p,e,t] triple.
- You can convert an FEMesh object to a [p,e,t] triple using the meshToPet function.

Mesh Data for FEMesh

A FEMesh object contains the nodes of the mesh as well as the elements (triangles for 2-D, tetrahedra for 3-D) and other data. For details, see FEMesh.

Mesh Data for [p,e,t] Triples: 2-D

For a 2-D mesh produced using either initmesh or meshToPet, the mesh data is as follows:

- p (points, the mesh nodes) is a 2-by-Np matrix of nodes, where Np is the number of nodes 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. There is a one-to-one correspondence between a mesh edges in e and the edges of the geometry. That is, e represents the discrete edges of the geometry in the same manner as t represents the discrete faces. Each column in the e matrix represents one edge, with the following data:

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

- e(2,k) is the index of the second point in mesh 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 geometric edge.
- e(4,k) is the parameter value at the second point of edge k.
- e(5,k) is the ID of the geometric edge containing the mesh edge. You can see edge IDs using the command pdegplot(geom,'EdgeLabels','on').
- e(6,k) is the subdomain number on the left side of the edge (subdomain 0 is the exterior of the geometry), where direction along the edge is given by increasing parameter values.
- e(7,k) is the subdomain number on the right side of the edge.

- t (triangles) is either a 4-by-Nt matrix of triangles or a 7-by-Nt matrix of triangles, depending on whether you called generateMesh with the GeometricOrder name-value pair set to 'quadratic' or 'linear', respectively. initmesh creates only 'linear' elements, which have size 4-by-Nt. Nt is the number of triangles in the mesh. Each column of t contains the indices of the points in p that form the triangle. The exception is the last entry in the column, which is the subdomain number. Triangle points are ordered as shown.
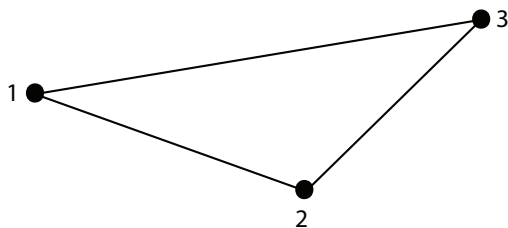
2-D linear element
showing node numbering
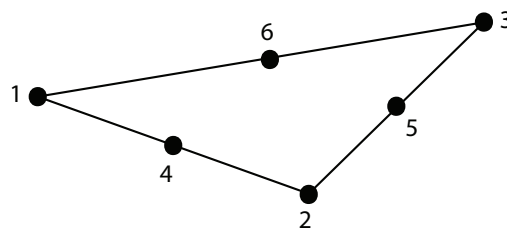
2-D quadratic element
showing node numbering

Note: Only the solvepde and solvepdeeig solvers use quadratic 2-D elements. Other solvers can only accept a linear triangular mesh.

Mesh Data for [p,e,t] Triples: 3-D

For a 3-D mesh produced using meshToPet, the mesh data is as follows:

- p (points, the mesh nodes) is a 3-by-Np matrix of nodes, where Np is the number of nodes in the mesh. Each column p(:,k) consists of the *x*-coordinate of point k in p(1,k), the *y*-coordinate of point k in p(2,k), and the *z*-coordinate of point k in p(3,k).

- e is an object that associates the mesh faces to the geometry boundary. Partial Differential Equation Toolbox functions use this association when converting the boundary conditions, which you set on geometry boundaries, to the mesh boundary faces.

- t (tetrahedra) is either an 11-by-Nt matrix of tetrahedra or a 5-by-Nt matrix of tetrahedra, depending on whether you called generateMesh with the GeometricOrder name-value pair set to 'quadratic' or 'linear', respectively. Nt is the number of tetrahedra in the mesh. Each column of t contains the indices of the points in p that form the tetrahedron. The exception is the last element in the column, which is the subdomain number. Tetrahedra points are ordered as shown.
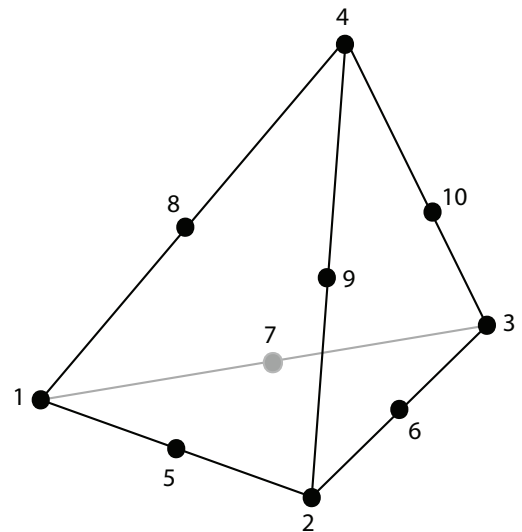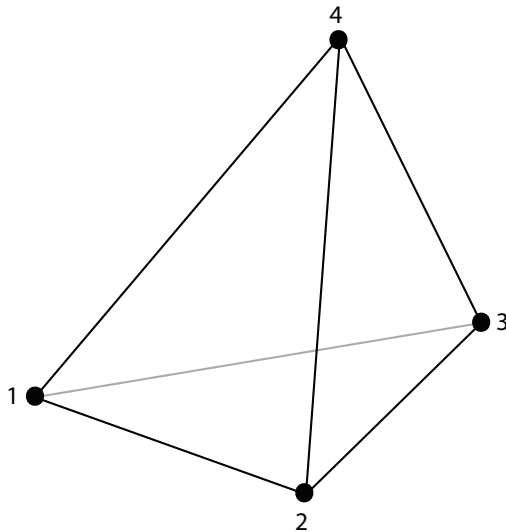
Related Examples

- "Solve PDE with Coefficients in Functional Form" on page 2-79
- "Solve Poisson's Equation on a Unit Disk" on page 3-75

Adaptive Mesh Refinement

In this section...
"Improving Solution Accuracy Using Mesh Refinement" on page 2-214
"Error Estimate for the FEM Solution" on page 2-215
"Mesh Refinement Functions" on page 2-216
"Mesh Refinement Termination Criteria" on page 2-216

Improving Solution Accuracy Using Mesh Refinement

Partial Differential Equation Toolbox software has a function for global, uniform mesh refinement for 2-D geometry. 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, when each triangle contributes less than a preset tolerance, or when an iteration limit is reached. You can provide an initial mesh, or let `adaptmesh` call `initmesh` automatically. You also choose selection and termination criteria parameters. 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 $\|\cdot\|$:

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

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

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

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 general form of the error indicator function for the elliptic equation $-\nabla \cdot (c\nabla u) + au = f$

is

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

where $\mathbf{n}_\tau$ is the unit normal of edge τ and the braced term is the jump in flux across the element edge. 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 or when the iteration limit is reached. 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

Solve 2-D PDEs Using the PDE App

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

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

---

1   Start the PDE app using `pdetool`.

    At this point, the PDE app 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 the PDE app 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 PDE app 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 PDE app 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 $\partial\Omega$ button. The outer boundaries of the decomposed geometry are displayed with the default boundary condition indicated. If the outer boundaries do not match the geometry of your problem, reenter the draw mode. You can then correct your CSG model by

adding, removing or altering any of the solid objects, or change the set formula used to evaluate the CSG model.

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

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

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

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 PDE app adds the necessary steps automatically.

- If you have not yet defined a CSG model, and leave the draw mode with an empty model, the PDE app 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, the PDE app 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, the PDE app 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, the PDE app initializes a mesh before solving the PDE.

- If you select a plot type and choose to plot the solution, the PDE app checks to see if there is a solution to the current PDE available. If not, the PDE app first solves the current PDE. The solution is then displayed using the selected plot options.

- If you have not defined your PDE, the PDE app 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

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

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 Coefficient for `specifyCoefficients`" on page 2-104), 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 $\mathbf{k}$ 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:

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

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

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

- $$v_y = \frac{\partial v}{\partial y}$$

- `exx`, the x-direction strain (ε_x)

- `eyy`, the y-direction strain (ε_y)

- `exy`, the shear strain (γ_{xy})

- `sxx`, the x-direction stress (σ_x)

- `syy`, the y-direction stress (σ_y)

- `sxy`, the shear stress (τ_{xy})

- `e1`, the first principal strain (ε_1)

- `e2`, the second principal strain (ε_2)

- `s1`, the first principal stress (σ_1)

- `s2`, the second principal stress (σ_2)

- `von Mises`, the von Mises effective stress

$$\sqrt{\sigma_1^2 + \sigma_2^2 - \sigma_1 \sigma_2}.$$

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

Example

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

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

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

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

Using the PDE App

Using the PDE app, 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`:

```
pdepoly([0 2/3 1 1 1/3 1/3 0],[1 1 2/3 0 0 1/3 1/3])
pdecirc(2/3,2/3,1/3)
```

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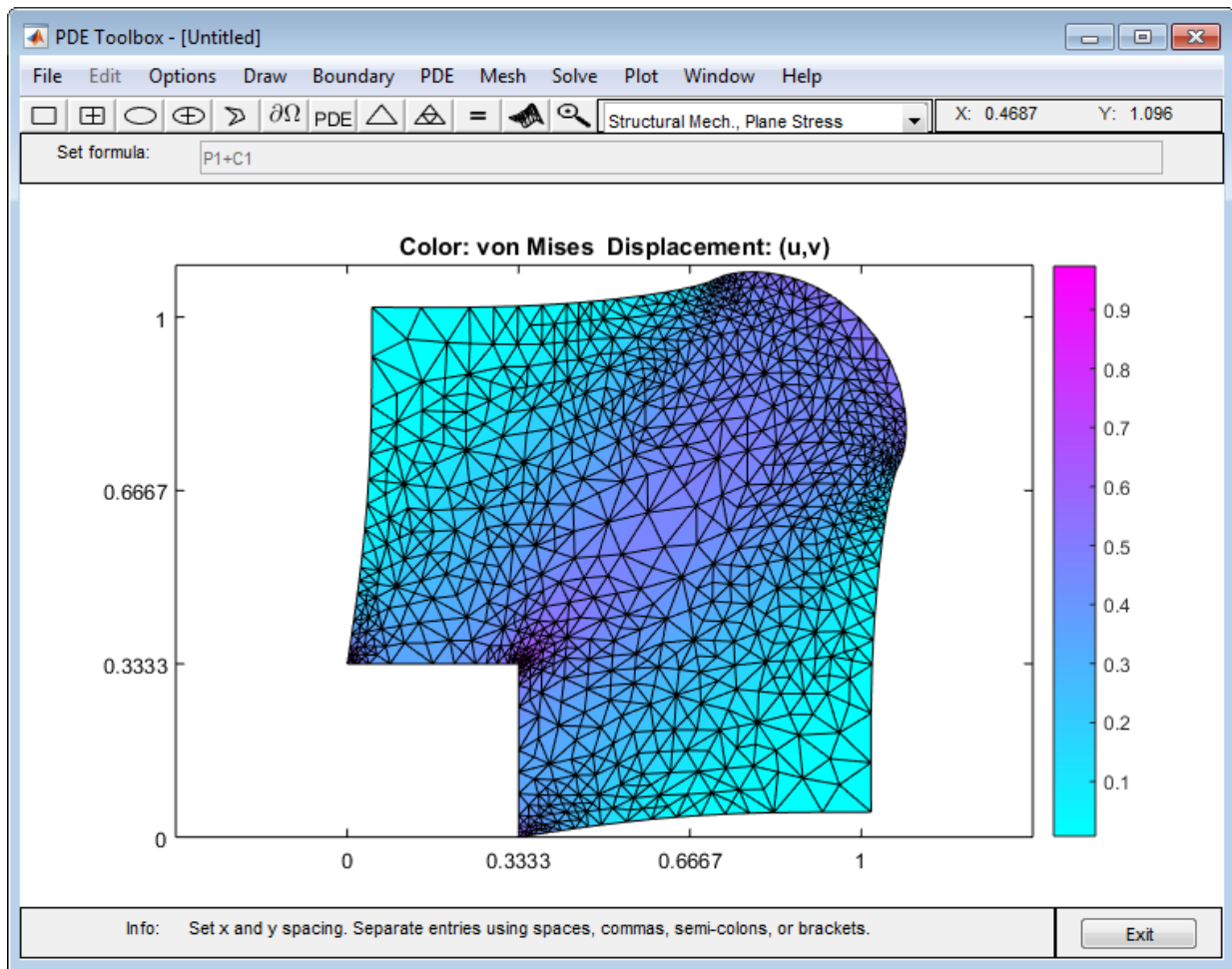
Visualization of the von Mises Effective Stress and the Displacements Using Deformed Mesh

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 in a direction 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. However, it is not obvious how to specify boundary conditions for this second order system. We cannot directly specify boundary conditions for both w and w'. Instead, we directly prescribe w' to be zero and use the following technique to define v' in such a way to insure that w also equals zero on the boundary. Stiff "springs" that apply a transverse shear force to the plate edge are distributed along the boundary. The shear force along the boundary due to these springs can be written $n \cdot D\nabla v = -kw$ where n is the normal to the boundary and k is the stiffness of the

springs. The value of k must be large enough that w is approximately zero at all points on the boundary but not so large that numerical errors result because the stiffness matrix is ill-conditioned. This expression is a generalized Neumann boundary condition supported by Partial Differential Equation Toolbox™

In the Partial Differential Equation Toolbox™ definition for an elliptic system, the w and v dependent variables are u(1) and u(2). The two second order partial differential equations can be rewritten as

$$-\nabla^2 u_1 + u_2 = 0$$

$$-D\nabla^2 u_2 = p$$

which is the form supported by the toolbox. The input corresponding to this formulation is shown in the sections below.

Create the PDE Model

Create a pde model for a PDE with two dependent variables

```
numberOfPDE = 2;
pdem = createpde(numberOfPDE);
```

Problem Parameters

```
E = 1.0e6; % modulus of elasticity
nu = .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 - nu^2));
pres = 2; % external pressure
```

Geometry Creation

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

```
gdm = [3 4 0 len len 0 0 0 len len]';
g = decsg(gdm, 'S1', ('S1')');
% Create a geometry entity
geometryFromEdges(pdem,g);
```

```
% Plot the geometry and display the edge labels for use in the boundary
% condition definition.
figure;
pdegplot(pdem, 'edgeLabels', 'on');
axis equal
title 'Geometry With Edge Labels Displayed';
```
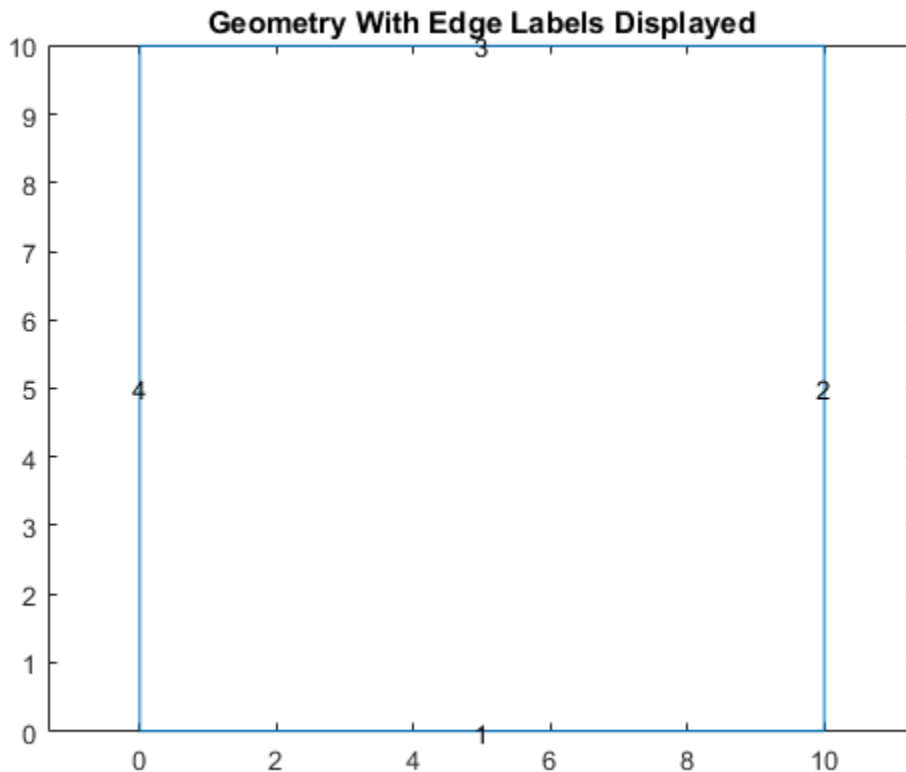

Coefficient Definition

The documentation on PDE coefficients shows the required formats for the a and c matrices. 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 c tensor, in the form of an $N \times N$ matrix of 2×2 submatrices is shown below.

$$
\left[\begin{array}{cc|cc}
c(1) & c(2) & \cdot & \cdot \\
\cdot & c(3) & \cdot & \cdot \\
\hline
\cdot & \cdot & c(4) & c(5) \\
\cdot & \cdot & \cdot & c(6)
\end{array} \right]
$$

The six-row by one-column c matrix is defined below. The entries in the full 2×2 a matrix and the 2×1 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]';
specifyCoefficients(pdem,'m', 0, 'd', 0, 'c', c, 'a', a, 'f', f);
```

Boundary Conditions

```
k = 1e7; % spring stiffness
% Define distributed springs on all four edges
bOuter = applyBoundaryCondition(pdem,'Edge',(1:4), 'g', [0 0], 'q', [0 0; k 0]);
```

Mesh generation

```
generateMesh(pdem, 'Hmax', hmax);
```

Finite Element and Analytical Solutions

The solution is calculated using the `solvepde` 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.

```
res = solvepde(pdem);
u = res.NodalSolution;
numNodes = size(pdem.Mesh.Nodes,2);
figure
pdeplot(pdem, 'xydata', u(1:numNodes), 'contour', 'on');
title 'Transverse Deflection'

numNodes = size(pdem.Mesh.Nodes,2);
fprintf('Transverse deflection at plate center(PDE Toolbox) = %12.4e\n', min(u(1:numNod
% compute analytical solution
wMax = -.0138*pres*len^4/(E*thick^3);
fprintf('Transverse deflection at plate center(analytical) = %12.4e\n', wMax);

Transverse deflection at plate center(PDE Toolbox) =  -2.7563e-01
```

Transverse deflection at plate center(analytical) = -2.7600e-01
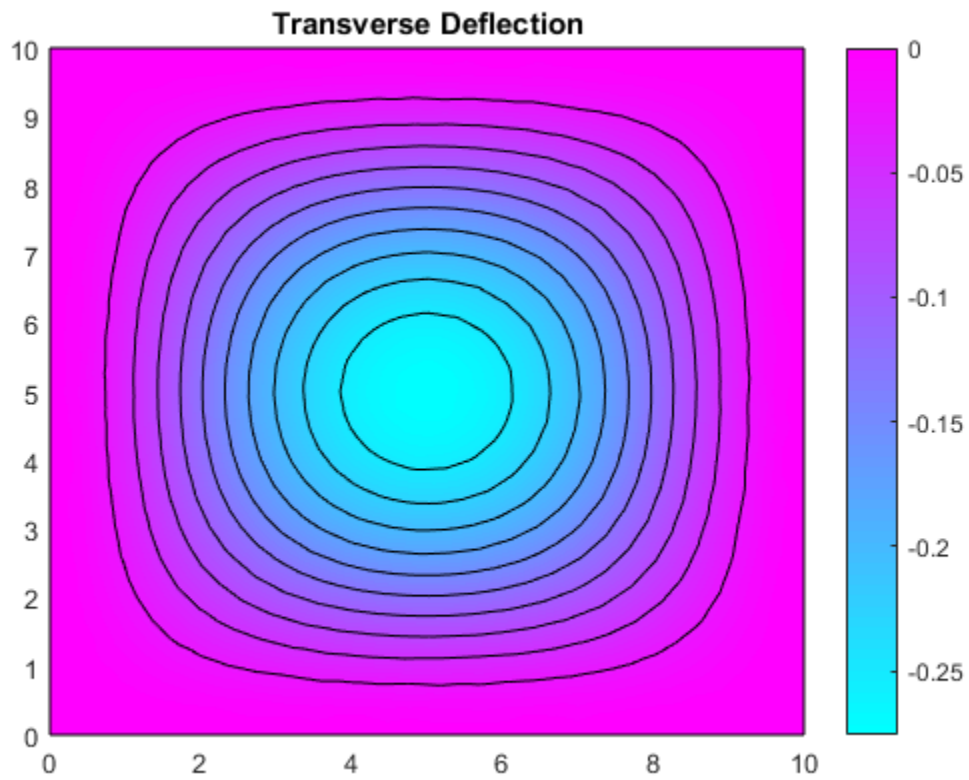
Deflection of a Piezoelectric Actuator

This example shows how to solve a coupled elasticity-electrostatics problem using 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 elastic behavior of the solid is described by the equilibrium equations

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

where σ is the stress tensor and f is the body force vector. The electrostatic behavior of the solid is described by 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 $\sigma_{11}, \sigma_{22},$ 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

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

3-19

where C_{ij} are the elastic coefficients, ε_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 described in detail in the PDE Toolbox 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

$$
\left[
\begin{array}{cccc|cc}
c_{1111} & c_{1112} & c_{1211} & c_{1212} & c_{1311} & c_{1312} \\
\cdot & c_{1122} & c_{1221} & c_{1222} & c_{1321} & c_{1322} \\
\hline
\cdot & \cdot & c_{2211} & c_{2212} & c_{2311} & c_{2312} \\
\cdot & \cdot & \cdot & c_{2222} & c_{2321} & c_{2322} \\
\hline
\cdot & \cdot & \cdot & \cdot & c_{3311} & c_{3312} \\
\cdot & \cdot & \cdot & \cdot & \cdot & c_{3322}
\end{array}
\right]
\left\{
\begin{array}{c}
\frac{\partial u}{\partial x} \\
\frac{\partial u}{\partial y} \\
\hline
\frac{\partial v}{\partial x} \\
\frac{\partial v}{\partial y} \\
\hline
\frac{\partial \phi}{\partial x} \\
\frac{\partial \phi}{\partial y}
\end{array}
\right\}
$$

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.

$$
\left[
\begin{array}{cccc|cc}
C_{11} & \cdot & \cdot & C_{12} & e_{11} & e_{31} \\
\cdot & G_{12} & G_{12} & \cdot & e_{14} & e_{34} \\
\hline
\cdot & \cdot & G_{12} & \cdot & e_{14} & e_{34} \\
\cdot & \cdot & \cdot & C_{22} & e_{13} & e_{33} \\
\hline
\cdot & \cdot & \cdot & \cdot & -\mathcal{E}_1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -\mathcal{E}_2
\end{array}
\right]
\left\{
\begin{array}{c}
\frac{\partial u}{\partial x} \\
\frac{\partial u}{\partial y} \\
\hline
\frac{\partial v}{\partial x} \\
\frac{\partial v}{\partial y} \\
\hline
\frac{\partial \phi}{\partial x} \\
\frac{\partial \phi}{\partial y}
\end{array}
\right\}
$$

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 down (minus y direction) and in the bottom layer, it points up. A schematic of the cantilever beam is shown in the figure below.
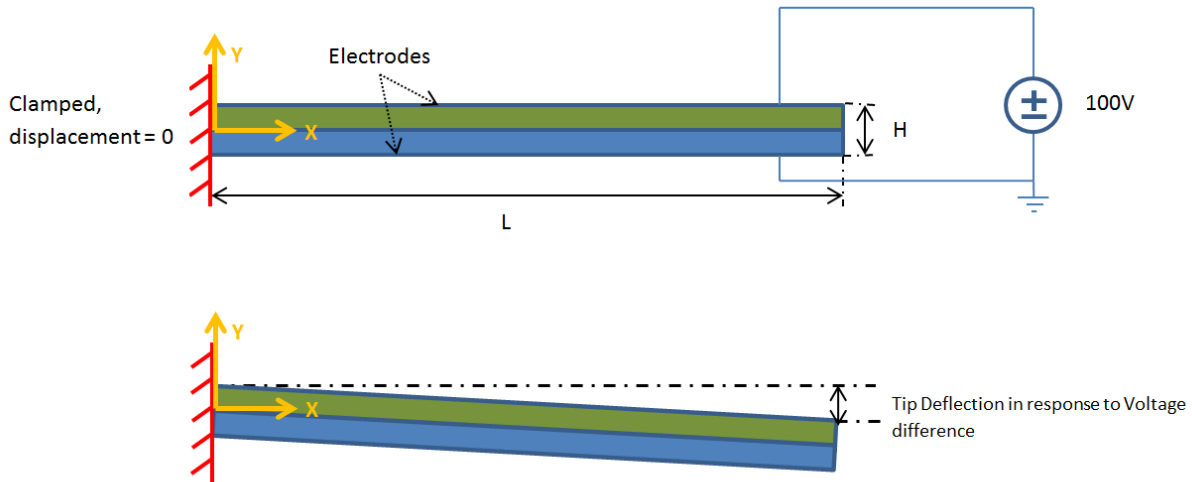
This figure is not to scale; the actual thickness/length ratio is 100 so the beam is very slender. When a voltage is applied between the lower and upper surfaces of the beam, it deflects in the y-direction; one layer shortens and the other layer lengthens. Devices of this type can be designed to provide the required motion or force for different applications.

Create a PDE Model with three dependent variables

The first step in solving a PDE problem is to create a PDE Model. This is a container that holds the number of equations, geometry, mesh, and boundary conditions for your PDE. The equations of linear elasticity have three components, so the number of equations in this model is three.

```
N = 3;
model = createpde(N);
```

Geometry Creation

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

```
L = 100e-3; % beam length in meters
H = 1e-3; % overall height of the beam
```

```matlab
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];
gdm = [topLayer; bottomLayer]';
g = decsg(gdm, 'R1+R2', ['R1'; 'R2']');

% Create a geometry entity and append to the PDE Model
geometryFromEdges(model,g);

figure;
pdegplot(model, 'edgeLabels', 'on', 'subdomainLabels', 'on');
title('Two-layer Piezoelectric Cantilever Beam (with edge labels)')
xlabel('X-coordinate, meters')
ylabel('Y-coordinate, meters')
axis([-.1*L, 1.1*L, -4*H2, 4*H2])
```
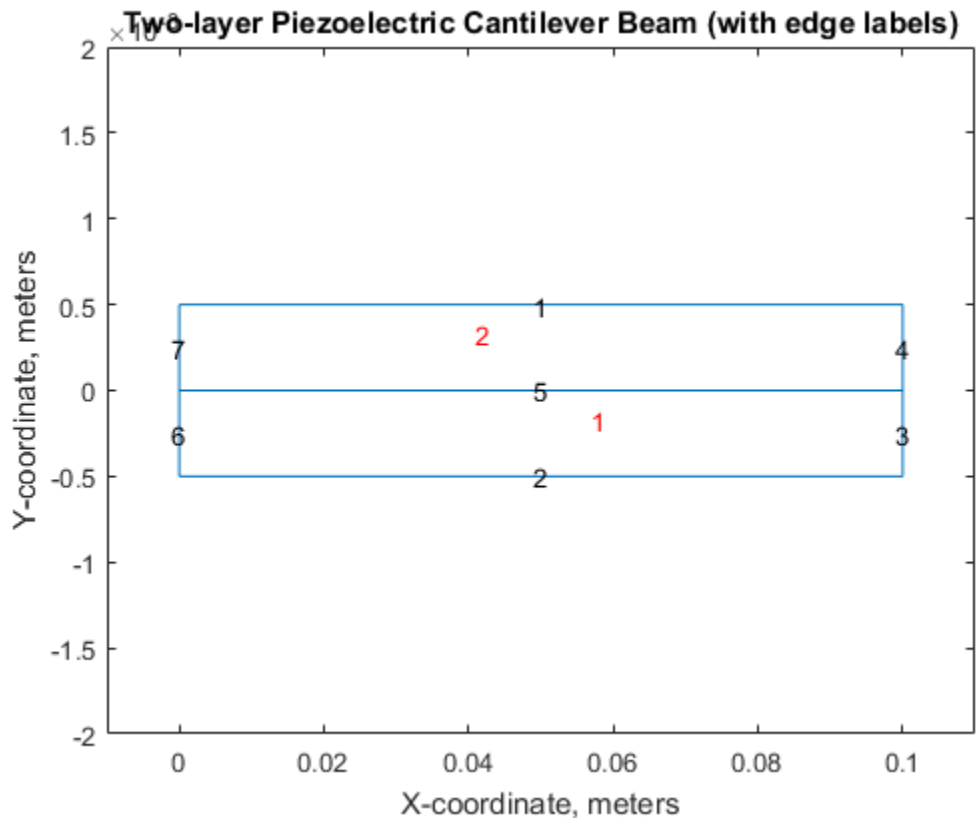
Material Properties and Coefficient Specification

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; % at constant stress
% electrical permittivity of vacuum
permittivityFreeSpace = 8.854187817620e-12; % 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;
D_const_stress = [relPermittivity 0; 0 relPermittivity]*permittivityFreeSpace;
% Convert dielectric matrix from constant stress to constant strain
D_const_strain = D_const_stress - pzeD'*pzeE;
% 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 = [D_const_strain(1,1) D_const_strain(2,1) D_const_strain(2,2)];
ctop = [c11(:); c12(:); c22(:); -c13(:); -c23(:); -c33(:)];
cbot = [c11(:); c12(:); c22(:);  c13(:);  c23(:); -c33(:)];

f = [0 0 0]';
specifyCoefficients(model, 'm', 0,'d', 0,'c', ctop, 'a', 0, 'f', f,'face',2);
specifyCoefficients(model, 'm', 0,'d', 0,'c', cbot, 'a', 0, 'f', f,'face',1);
```

Boundary Condition Definition

For this example, the top geometry edge (edge 1) has the voltage prescribed as 100 volts. The bottom geometry edge (edge 2) has the voltage prescribed as 0 volts (i.e. grounded). The left geometry 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 geometry edge (i.e. q = 0).

```
V = 100;
% Set the voltage (solution component 3) on the top edge to V.
voltTop = applyBoundaryCondition(model,'Edge',1, 'u', V, 'EquationIndex', 3);
% Set the voltage (solution component 3) on the bottom edge to zero.
voltBot = applyBoundaryCondition(model,'Edge',2, 'u', 0, 'EquationIndex', 3);
% Set the x and y displacements (solution components 1 and 2)
% on the left end (geometry edges 6 and 7) to zero.
```

```
clampLeft = applyBoundaryCondition(model,'Edge',6:7, 'u', [0 0], 'EquationIndex', 1:2);
```

Mesh Generation

We need a relatively fine mesh with maximum element size roughly equal H/16 to accurately model the bending of the beam.

```
hmax = H/16;
msh = generateMesh(model, 'Hmax', hmax, 'MesherVersion', 'R2013a');
```

Warning: Approximately 51200 triangles will be generated.

Finite Element Solution

```
result = solvepde(model);
```

Extract the NodalSolution property from the result, this has the x-deflection in column 1, the y-deflection in column 2, and the electrical potential in column 3. Find the minimum y-deflection, and plot the solution components.

```
rs = result.NodalSolution;
feTipDeflection = min(rs(:,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(model, 'xydata', rs(:,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: -3.2772e-05
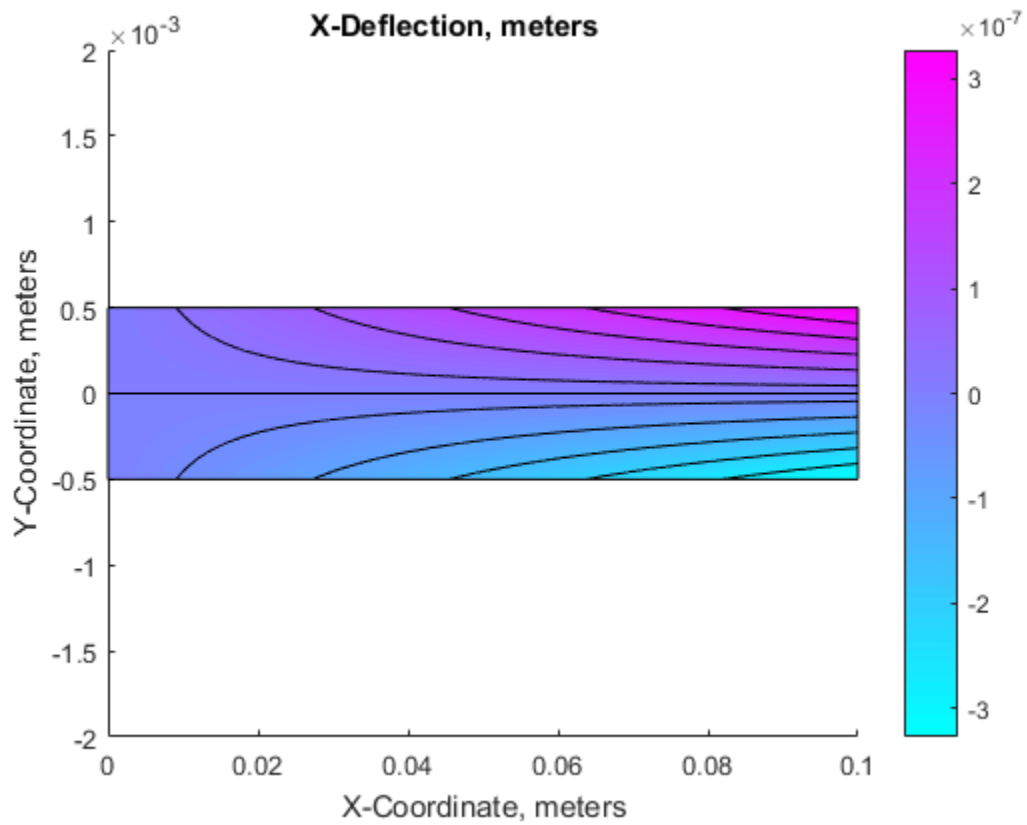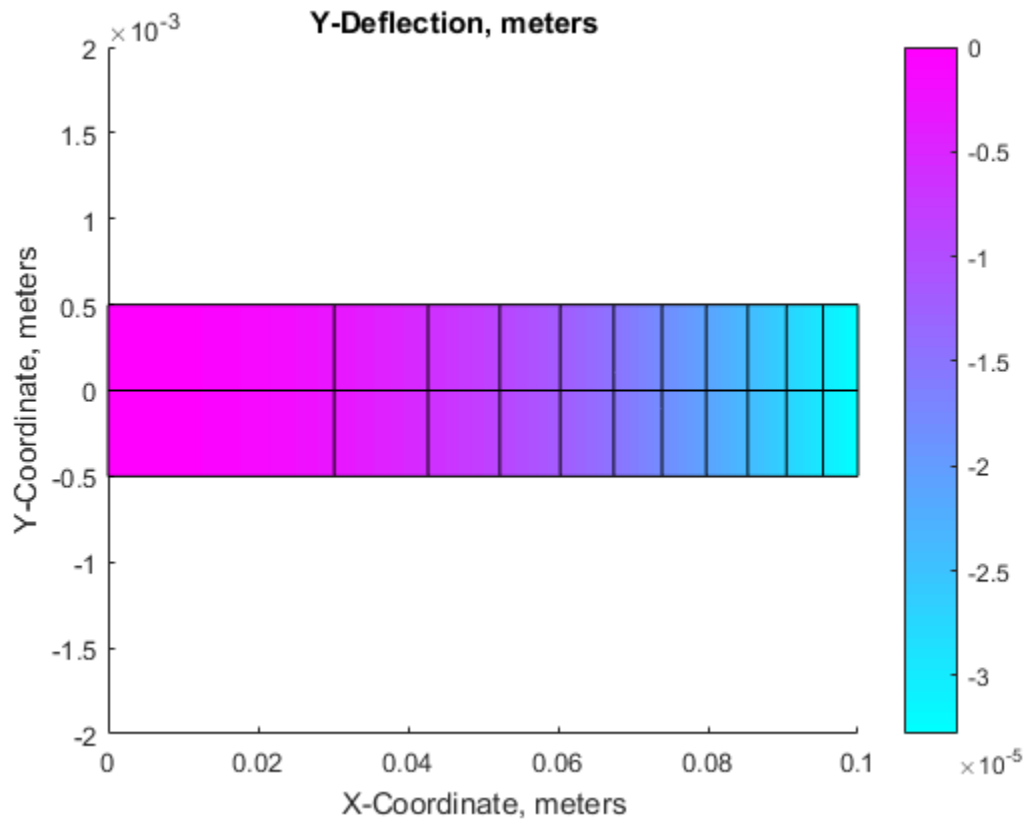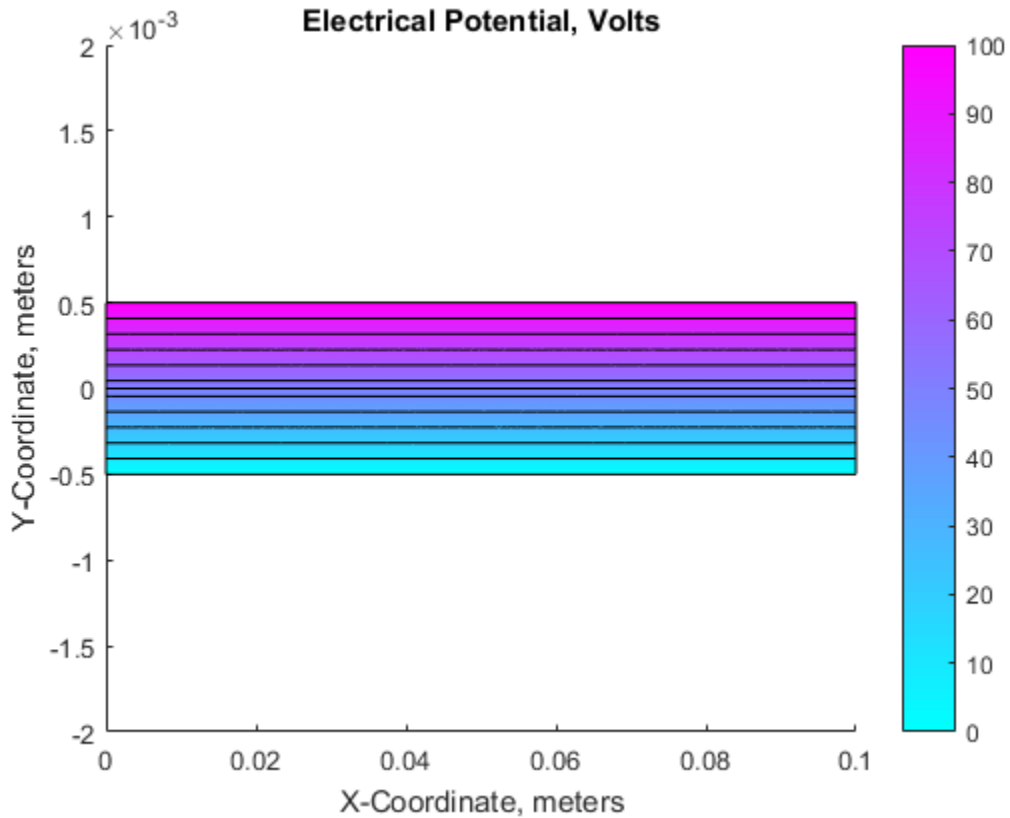
Analytical Solution

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:  -3.3000e-05
```

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 PDE App

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.
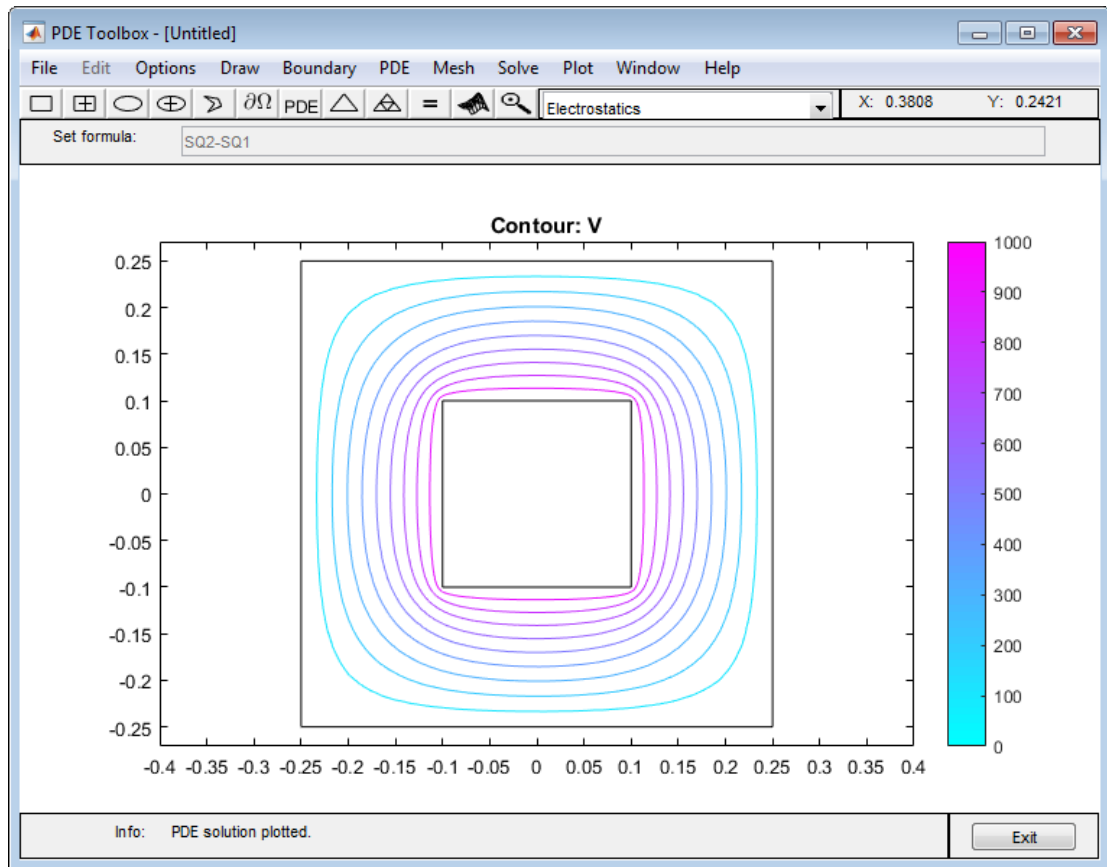
Equipotential Lines in Air-Filled Frame

3-D Linear Elasticity Equations in Toolbox Form

In this section...

How to Express Coefficients

The stiffness matrix of linear elastic isotropic material contains two parameters:

- E, the elastic modulus
- ν, Poisson's ratio

To include these parameters in a 3-D problem, you can use the `elasticityC3D(E,nu)` function (included in your software) as the `c` coefficient. This function uses the linearized, small-displacement assumption for an isotropic material. For examples that use this function, see Deflection Analysis of a Bracket and Vibration of a Square Plate.

The remainder of this section derives the `c` coefficient in `elasticityC3D(E,nu)` from the equations of linear elasticity.

Summary of the Equations of Linear Elasticity

Define the following quantities.

σ = stress
f = body force
ε = strain
u = displacement

The equilibrium equation is

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

The linearized, small-displacement strain-displacement relationship is

$$\varepsilon = \frac{1}{2}\left(\nabla u + \nabla u^T\right).$$

The balance of angular momentum states that stress is symmetric:

$$\sigma_{ij} = \sigma_{ji}.$$

The Voigt notation for the constitutive equation of the linear isotropic model is

$$
\begin{bmatrix}
\sigma_{11} \\
\sigma_{22} \\
\sigma_{33} \\
\sigma_{23} \\
\sigma_{13} \\
\sigma_{12}
\end{bmatrix}
= \frac{E}{(1+v)(1-2v)}
\begin{bmatrix}
1-v & v & v & 0 & 0 & 0 \\
v & 1-v & v & 0 & 0 & 0 \\
v & v & 1-v & 0 & 0 & 0 \\
0 & 0 & 0 & 1-2v & 0 & 0 \\
0 & 0 & 0 & 0 & 1-2v & 0 \\
0 & 0 & 0 & 0 & 0 & 1-2v
\end{bmatrix}
\begin{bmatrix}
\varepsilon_{11} \\
\varepsilon_{22} \\
\varepsilon_{33} \\
\varepsilon_{23} \\
\varepsilon_{13} \\
\varepsilon_{12}
\end{bmatrix}.
$$

In expanded form, using all the entries in σ and ε taking symmetry into account,

$$
\begin{bmatrix}
\sigma_{11} \\
\sigma_{12} \\
\sigma_{13} \\
\sigma_{21} \\
\sigma_{22} \\
\sigma_{23} \\
\sigma_{31} \\
\sigma_{32} \\
\sigma_{33}
\end{bmatrix}
= \frac{E}{(1+v)(1-2v)}
\begin{bmatrix}
1-v & 0 & 0 & 0 & v & 0 & 0 & 0 & v \\
\bullet & 1-2v & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\bullet & \bullet & 1-2v & 0 & 0 & 0 & 0 & 0 & 0 \\
\bullet & \bullet & \bullet & 1-2v & 0 & 0 & 0 & 0 & 0 \\
\bullet & \bullet & \bullet & \bullet & 1-v & 0 & 0 & 0 & v \\
\bullet & \bullet & \bullet & \bullet & \bullet & 1-2v & 0 & 0 & 0 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-2v & 0 & 0 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-2v & 0 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-v
\end{bmatrix}
\begin{bmatrix}
\varepsilon_{11} \\
\varepsilon_{12} \\
\varepsilon_{13} \\
\varepsilon_{21} \\
\varepsilon_{22} \\
\varepsilon_{23} \\
\varepsilon_{31} \\
\varepsilon_{32} \\
\varepsilon_{33}
\end{bmatrix}.
$$

In the preceding diagram, $\bullet$ means the entry is symmetric.

Conversion to Toolbox Form

The toolbox form for the equation is

$$-\nabla \cdot (c \otimes \nabla u) = f.$$

But the equations in the summary do not have ∇u alone, it appears together with its transpose:

$$\varepsilon = \frac{1}{2}\left(\nabla u + \nabla u^T\right).$$

It is a straightforward exercise to convert this equation for strain ε to ∇u. In column vector form,

$$\nabla u = \begin{bmatrix} \partial u_x / \partial x \\ \partial u_x / \partial y \\ \partial u_x / \partial z \\ \partial u_y / \partial x \\ \partial u_y / \partial y \\ \partial u_y / \partial z \\ \partial u_z / \partial x \\ \partial u_z / \partial y \\ \partial u_z / \partial z \end{bmatrix}.$$

Therefore, you can write the strain-displacement equation as

$$\varepsilon = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \nabla u \equiv A \nabla u,$$

where *A* stands for the displayed matrix. So rewriting Equation 3-1, and recalling that $\cdot$ means an entry is symmetric,

$$
\sigma = \frac{E}{(1+v)(1-2v)}
\begin{bmatrix}
1-v & 0 & 0 & 0 & v & 0 & 0 & 0 & v \\
\cdot & 1-2v & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\cdot & \cdot & 1-2v & 0 & 0 & 0 & 0 & 0 & 0 \\
\cdot & \cdot & \cdot & 1-2v & 0 & 0 & 0 & 0 & 0 \\
\cdot & \cdot & \cdot & \cdot & 1-v & 0 & 0 & 0 & v \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1-2v & 0 & 0 & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1-2v & 0 & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1-2v & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1-v
\end{bmatrix}
A\nabla u
$$

$$
= \frac{E}{(1+v)(1-2v)}
\begin{bmatrix}
1-v & 0 & 0 & 0 & v & 0 & 0 & 0 & v \\
0 & 1/2-v & 0 & 1/2-v & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1/2-v & 0 & 0 & 0 & 1/2-v & 0 & 0 \\
0 & 1/2-v & 0 & 1/2-v & 0 & 0 & 0 & 0 & 0 \\
v & 0 & 0 & 0 & 1-v & 0 & 0 & 0 & v \\
0 & 0 & 0 & 0 & 0 & 1/2-v & 0 & 1/2-v & 0 \\
0 & 0 & 1/2-v & 0 & 0 & 0 & 1/2-v & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1/2-v & 0 & 1/2-v & 0 \\
v & 0 & 0 & 0 & v & 0 & 0 & 0 & 1-v
\end{bmatrix}
\nabla u.
$$

Make the definitions

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

$$
c_1 = \frac{E(1-v)}{(1+v)(1-2v)}
$$

$$
c_{12} = \frac{Ev}{(1+v)(1-2v)}
$$

and the equation becomes

$$\sigma = \begin{bmatrix} c_1 & 0 & 0 & 0 & c_{12} & 0 & 0 & 0 & c_{12} \\ 0 & G & 0 & G & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & G & 0 & 0 & 0 & G & 0 & 0 \\ 0 & G & 0 & G & 0 & 0 & 0 & 0 & 0 \\ c_{12} & 0 & 0 & 0 & c_1 & 0 & 0 & 0 & c_{12} \\ 0 & 0 & 0 & 0 & 0 & G & 0 & G & 0 \\ 0 & 0 & G & 0 & 0 & 0 & G & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & G & 0 & G & 0 \\ c_{12} & 0 & 0 & 0 & c_{12} & 0 & 0 & 0 & c_1 \end{bmatrix} \nabla u \equiv c \nabla u.$$

To express the coefficient c for a toolbox solver, notice that it is a symmetric matrix. With $N = 3$ equations, you can write this in the form "3N(3N+1)/2-Element Column Vector c, 3-D Systems" on page 2-142. This is the form of the `c` argument in `elasticityC3D(E,nu)`.

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,

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

You can impose the common gauge assumption (Lorenz gauge or Coulomb gauge, see Wikipedia® [2])

$$\nabla \cdot \mathbf{A} = 0,$$

and then the equation for $\mathbf{A}$ in terms of $\mathbf{J}$ 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.

References

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

[2] Wikipedia entries on Gauge fixing.

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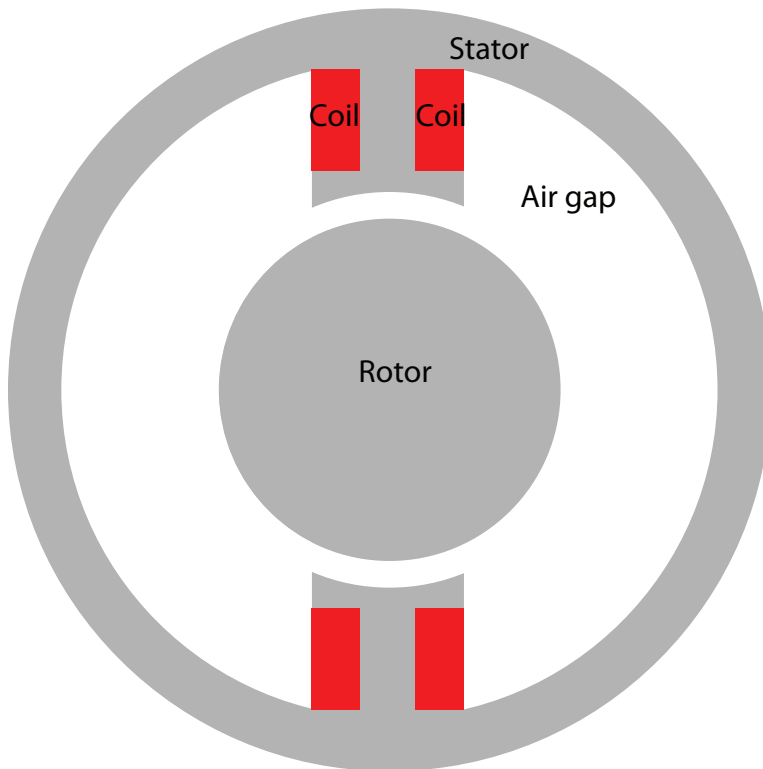
The magnetic permeability μ is 1 in the air and in the coil. In the stator and the rotor, μ is defined by

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

$\mu_{\max} = 5000$, $\mu_{\min} = 200$, 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 PDE App

The geometry is complex, involving five circular arcs and two rectangles. Using the PDE app, 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.
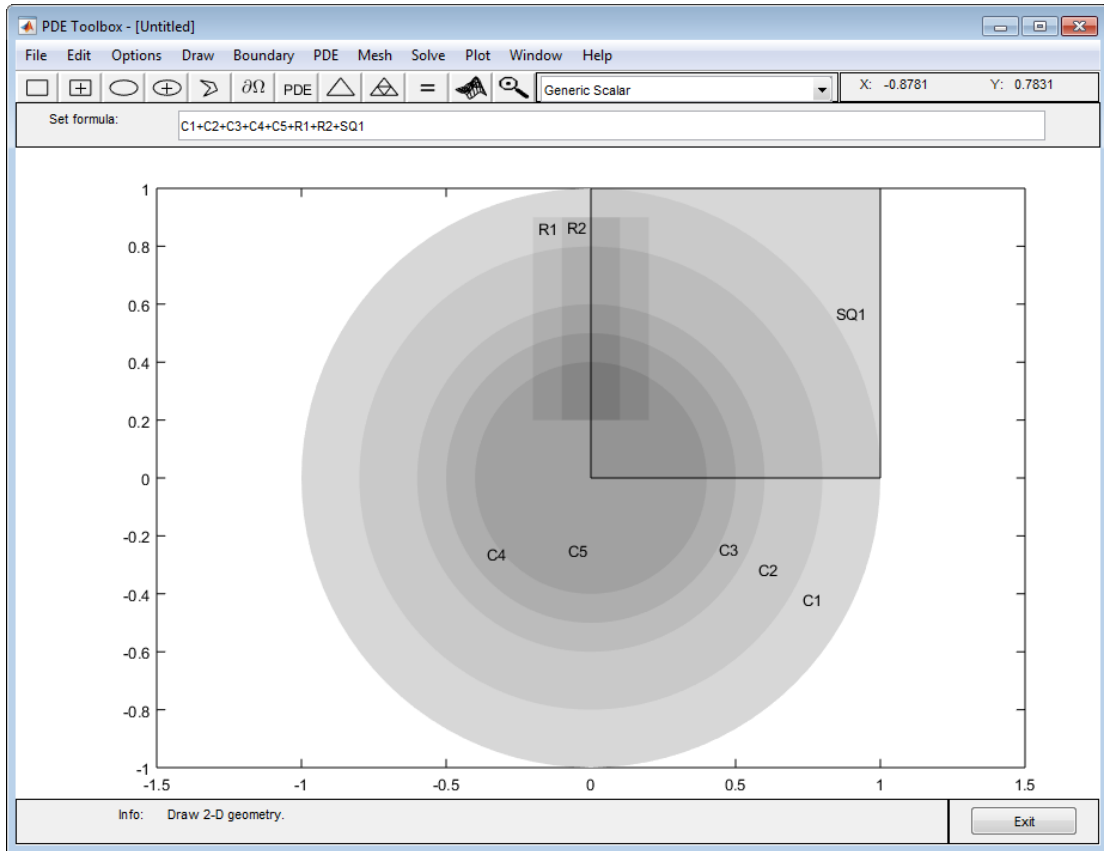
Enter the following set formula to reduce the model to the first quadrant:

```
(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 rotor is subdomain 1,

the stator is subdomain 2, the air gap is subdomain 3, and the coil is subdomain 4. The numbering of your subdomains may be different.
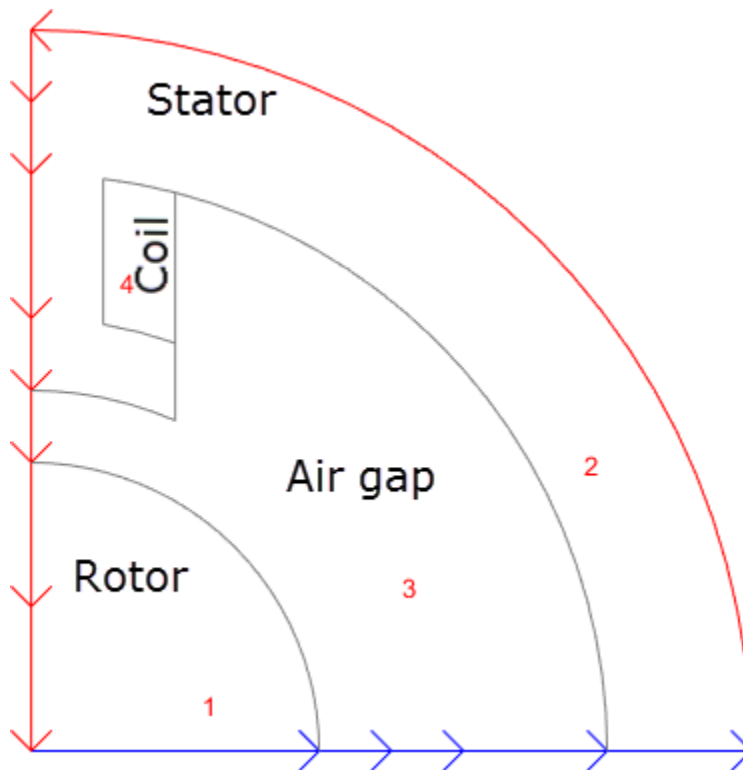
Before moving to the PDE mode, select the boundaries along the *x*-axis and set the boundary condition to a Neumann condition with *g* = 0 and *q* = 0. In the PDE mode, turn on the labels by selecting the **Show Subdomain Labels** option from the **PDE** menu. Double-click each subdomain to define the 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.
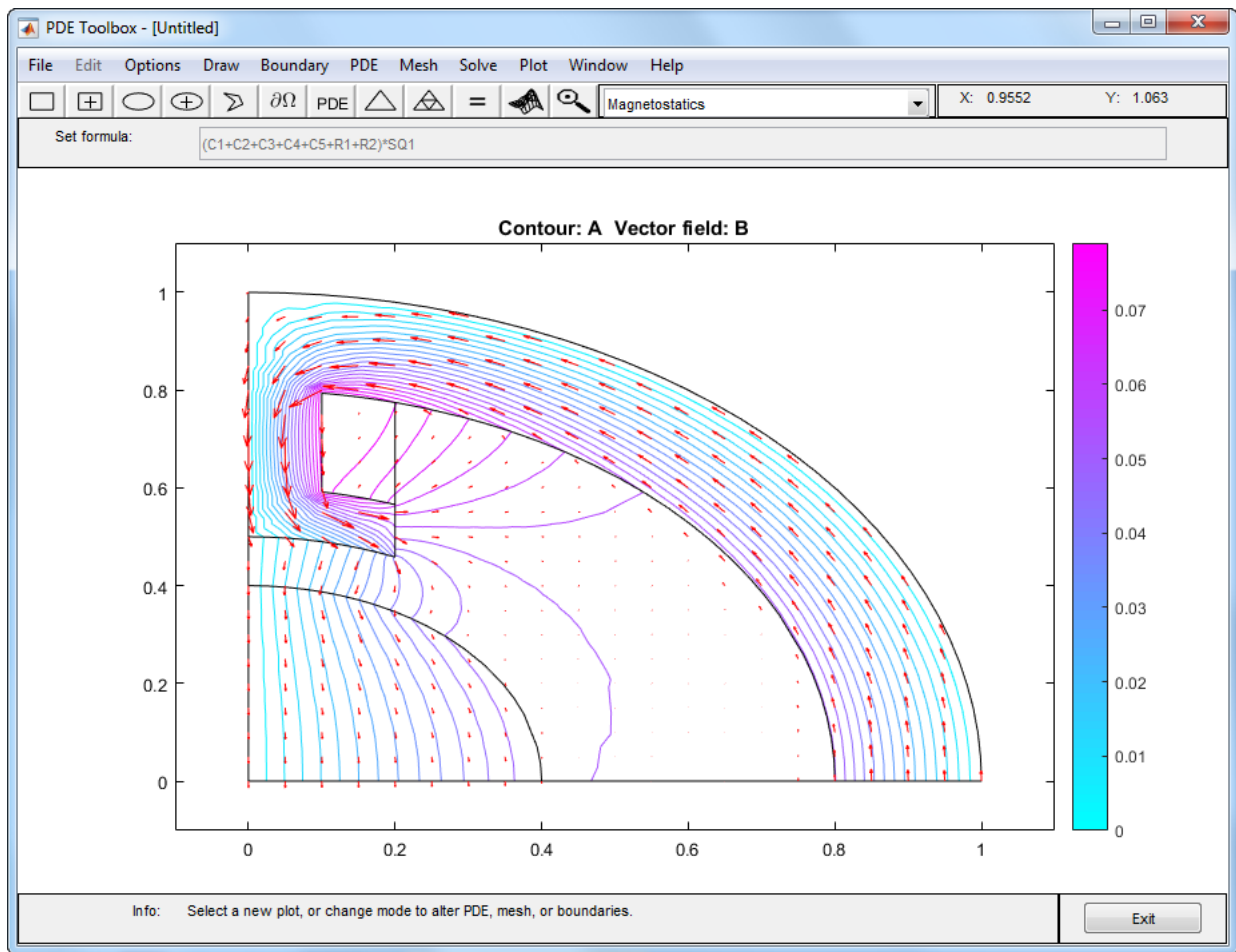
Equipotential Lines and Magnetic Flux in a Two-Pole Motor

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 o. 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 $\mathbf{E}_c = (0, 0, E_c)$, $\mathbf{J} = \left(0, 0, Je^{j\omega t}\right)$, 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 PDE App

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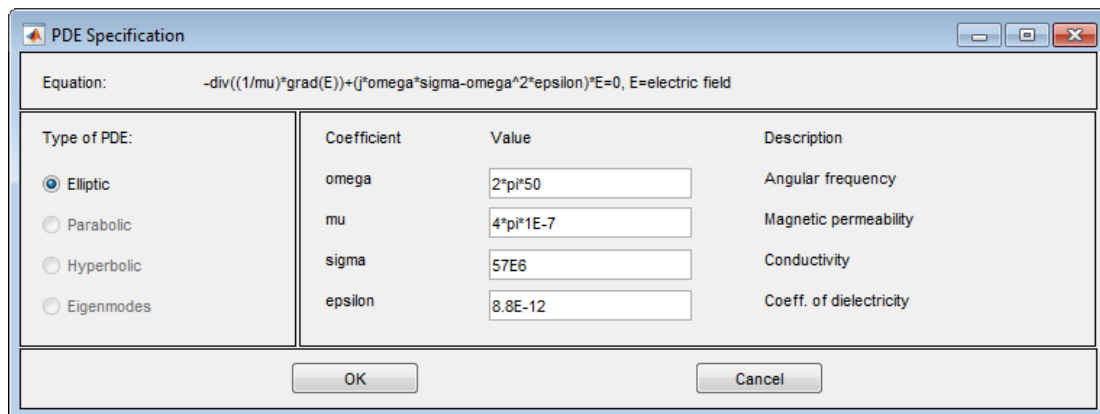
Initialize the mesh and solve the equation. Due to the skin effect, the current density at the surface of the conductor is much higher than in the conductor's interior. This is clearly visualized by plotting the current density J as a 3-D plot. To improve the accuracy of the solution close to the surface,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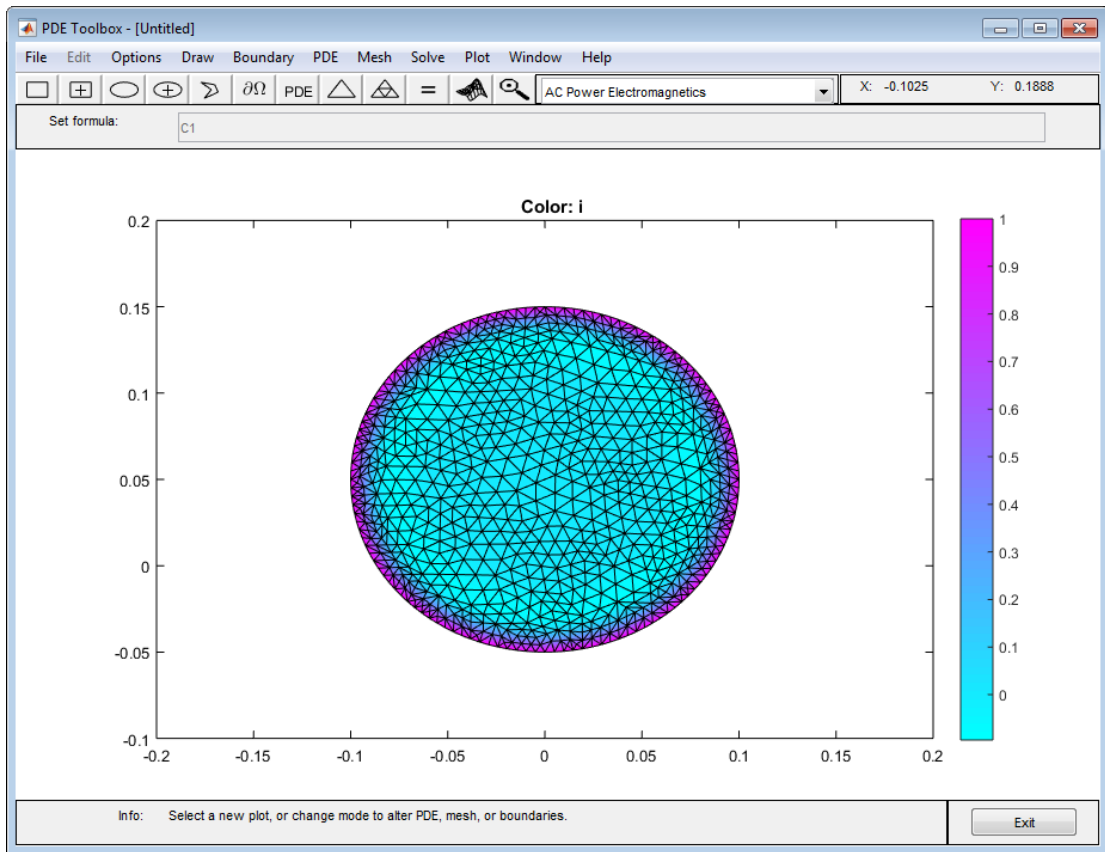
The Adaptively Refined Mesh

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).
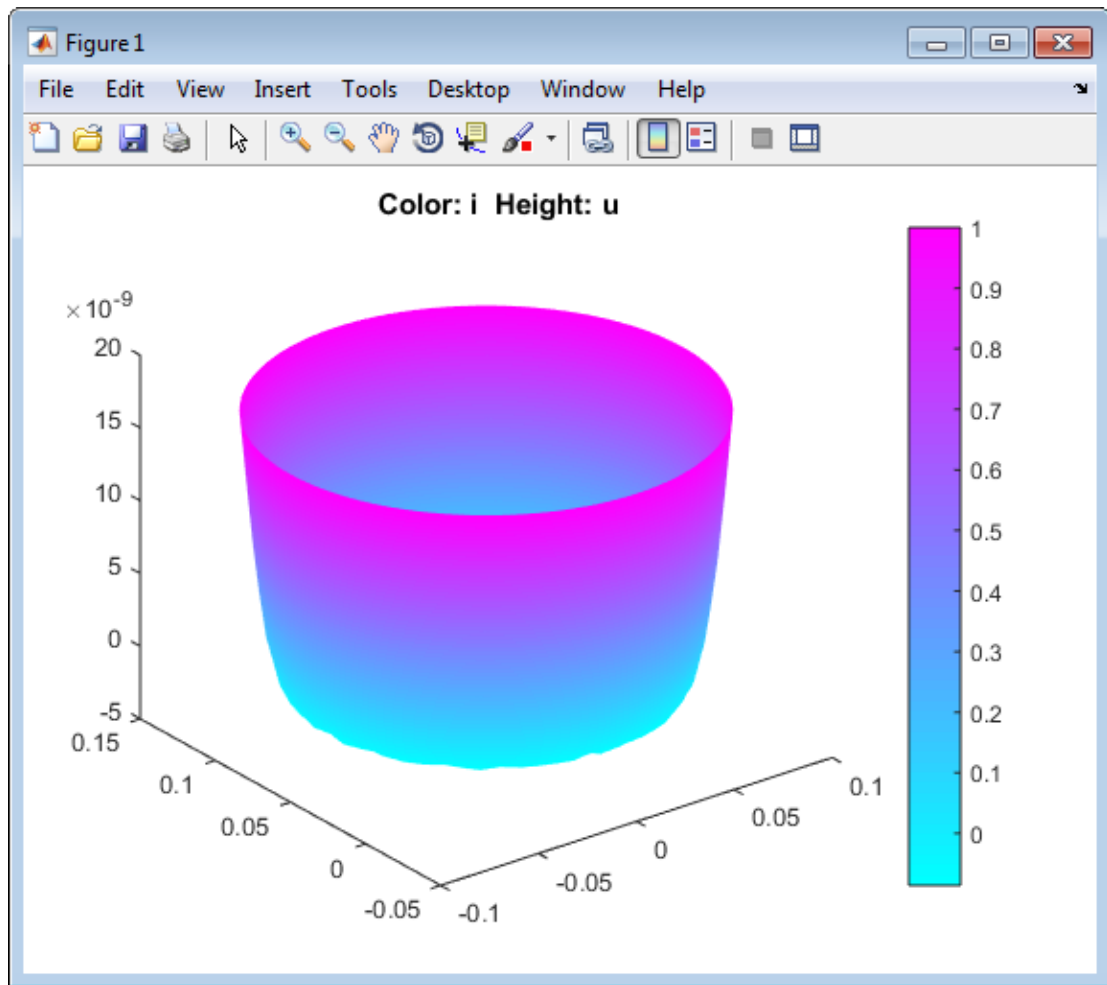
The Current Density in an AC Wire

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 PDE app 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**.
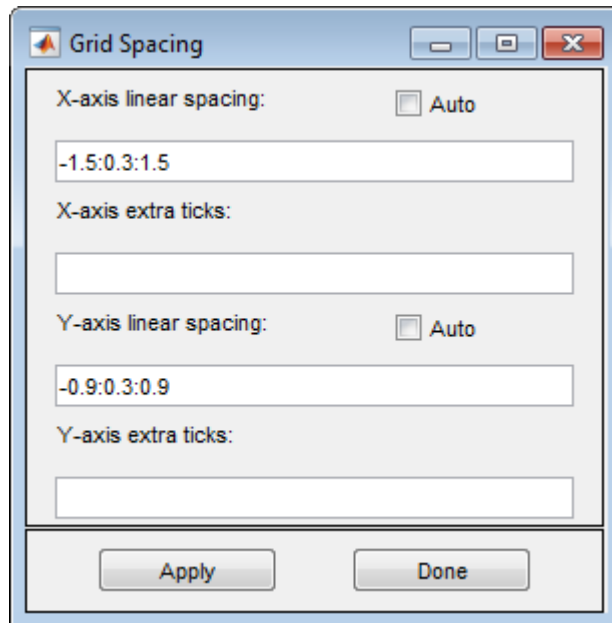
4 Click **Options** > **Snap**

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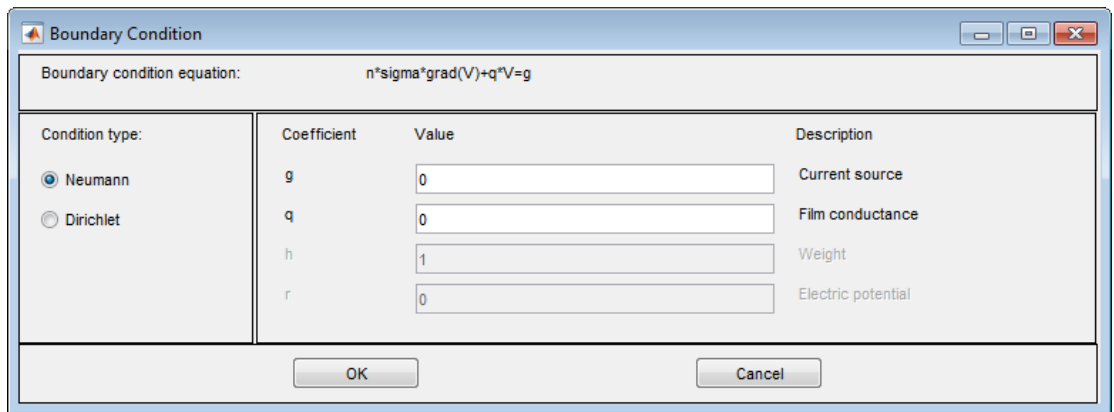
11 Hold down **Shift** and click to select the left circular conductor boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

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

- **Condition type** = **Dirichlet**
- **h** = 1
- **r** = 1
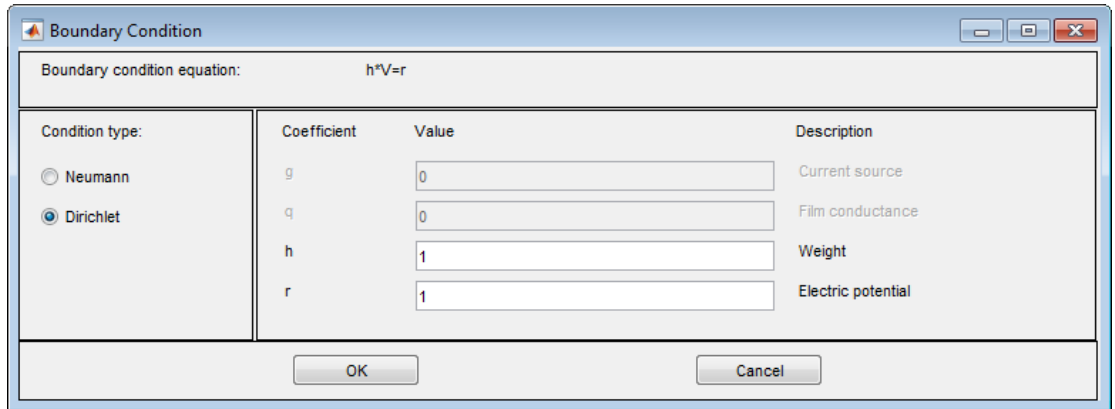
13 Hold down **Shift** and click to select the right circular conductor boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

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

- **Condition type** = **Dirichlet**

- **h** = 1

- **r** = -1
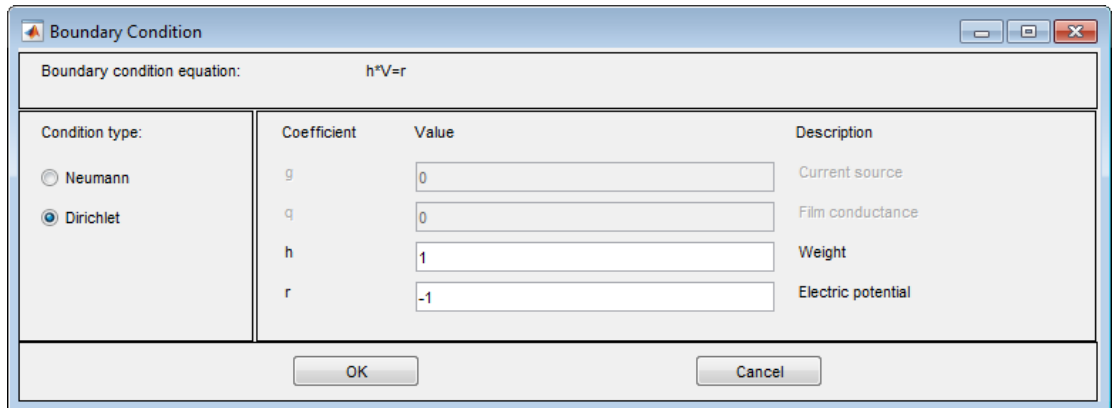
15 Open the PDE Specification dialog box by clicking **PDE** > **PDE Specification**.

16 Set the current source, **q**, parameter to 0.
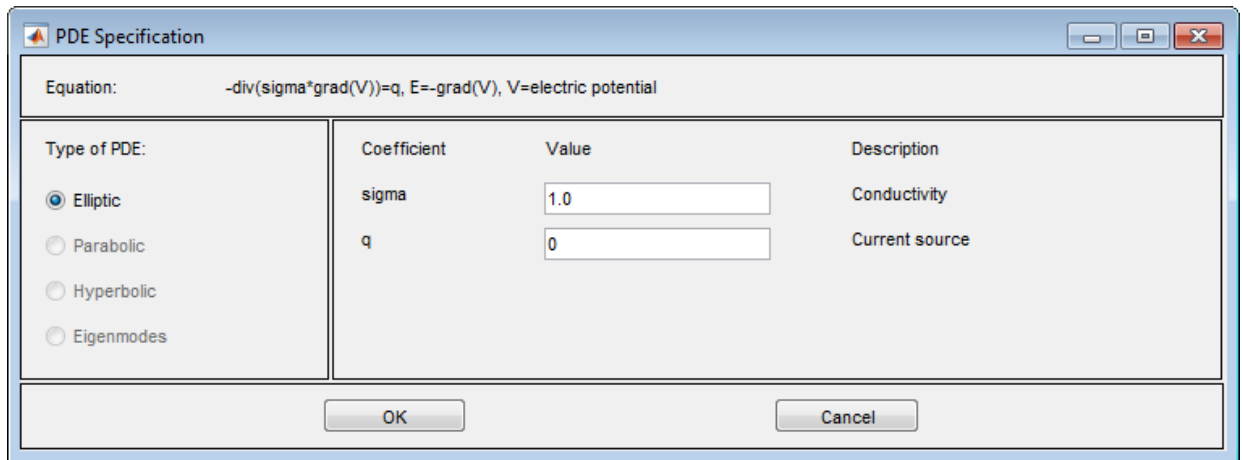
17 Initialize the mesh by clicking **Mesh** > **Initialize Mesh**.

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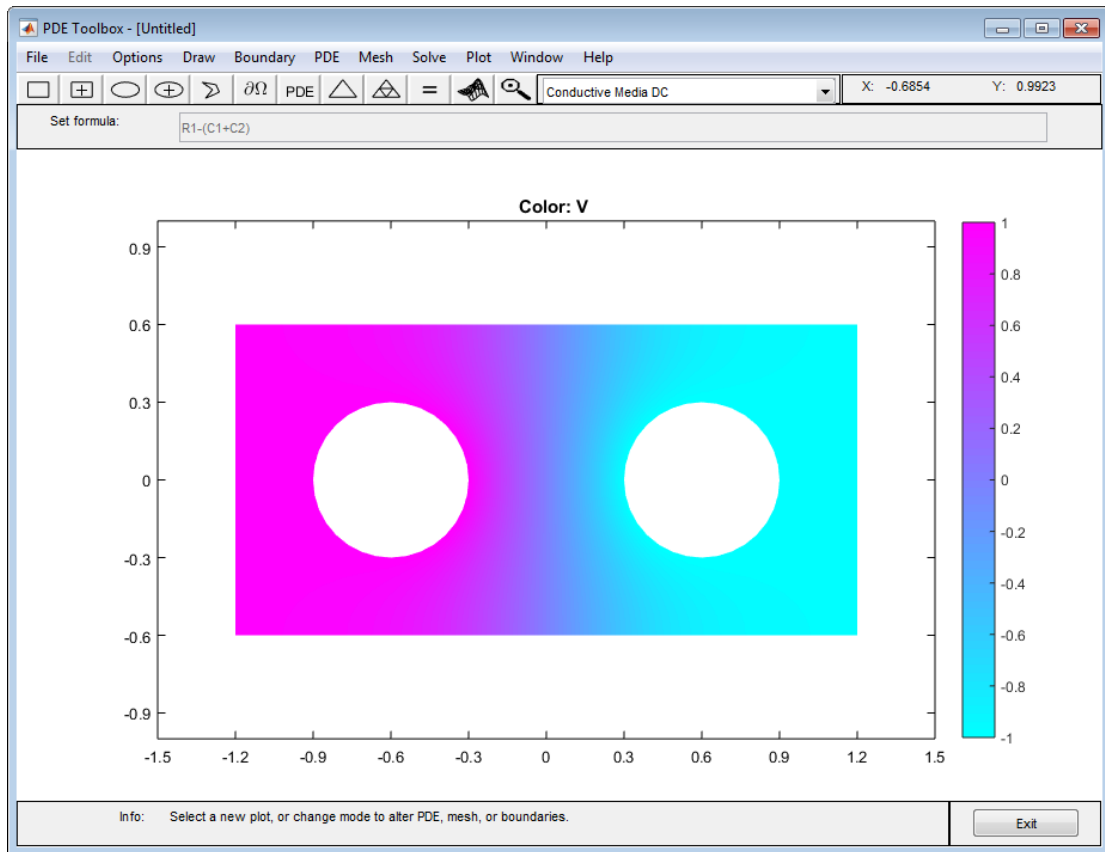
21 Visualize the current density **J** by clicking **Plot** > **Parameters**, selecting **Contour** and **Arrows** check box, and clicking **Plot**.

The current flows, as expected, from the conductor with a positive potential to the conductor with a negative potential.
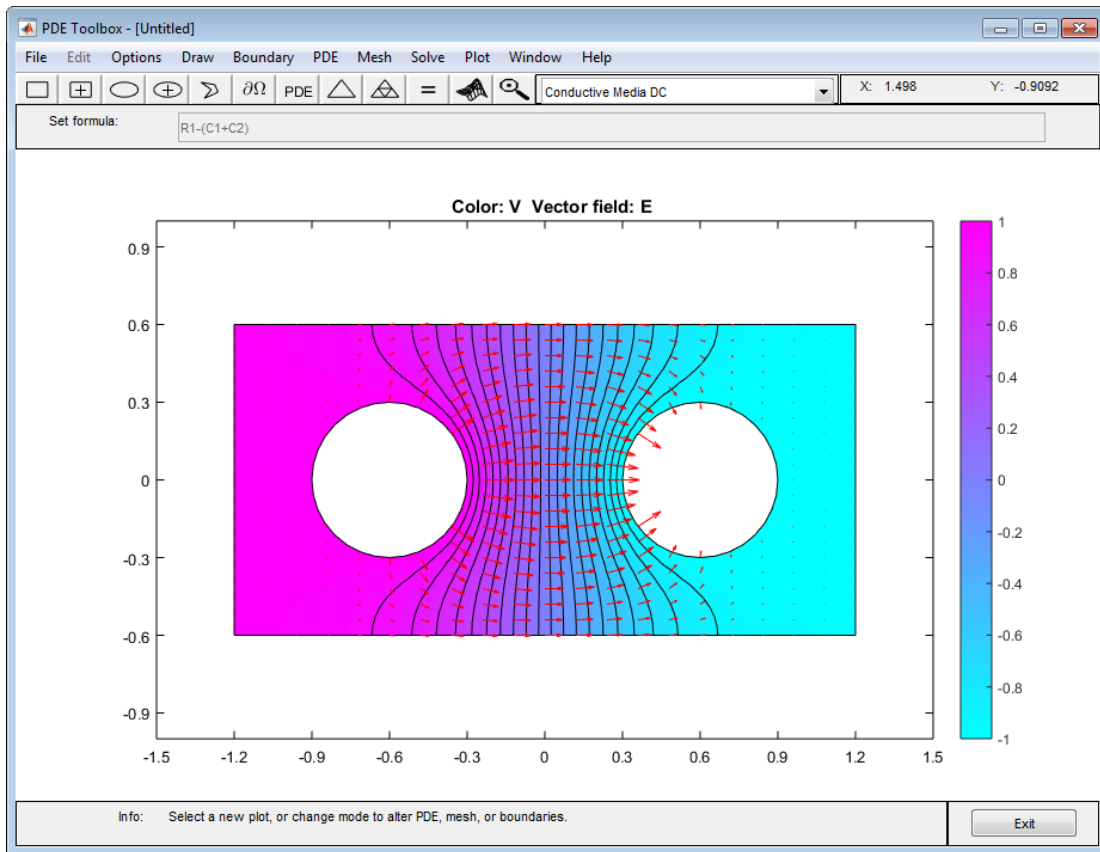
The Current Density Between Two Metallic Conductors

Heat Transfer

The heat equation is a parabolic PDE:

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

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

- Density ρ
- Heat capacity C
- Coefficient of heat conduction k
- Heat source Q
- Convective heat transfer coefficient h
- External temperature T_{ext}

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

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

is also available.

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

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

Example

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

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

Using the PDE App

Start the PDE app and set the application mode to **Heat Transfer**. Using the **Options** menu, set the *x*-axis limit to `[-1.5 4.5]` and *y*-axis limit to `[-0.5 3.5]`. Create the square region with corners in (0,0), (3,0), (3,3), and (0,3) and the diamond-shaped region with corners in (1.5,0.5), (2.5,1.5), (1.5,2.5), and (0.5,1.5).

```
pderect([0 3 0 3])
pdepoly([1.5 2.5 1.5 0.5],[0.5 1.5 2.5 1.5])
```

The temperature is kept at 0 on all the outer boundaries, so you do not have to change the default boundary conditions. Define the PDE parameters for each of the two regions. For this, click each region and select **PDE Specification** from the **PDE** menu. Since you are solving the parabolic heat equation, select the **Parabolic**. 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, you can select the **Plot in x-y grid** option to use a rectangular grid instead of a triangular grid which is used by default. 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.
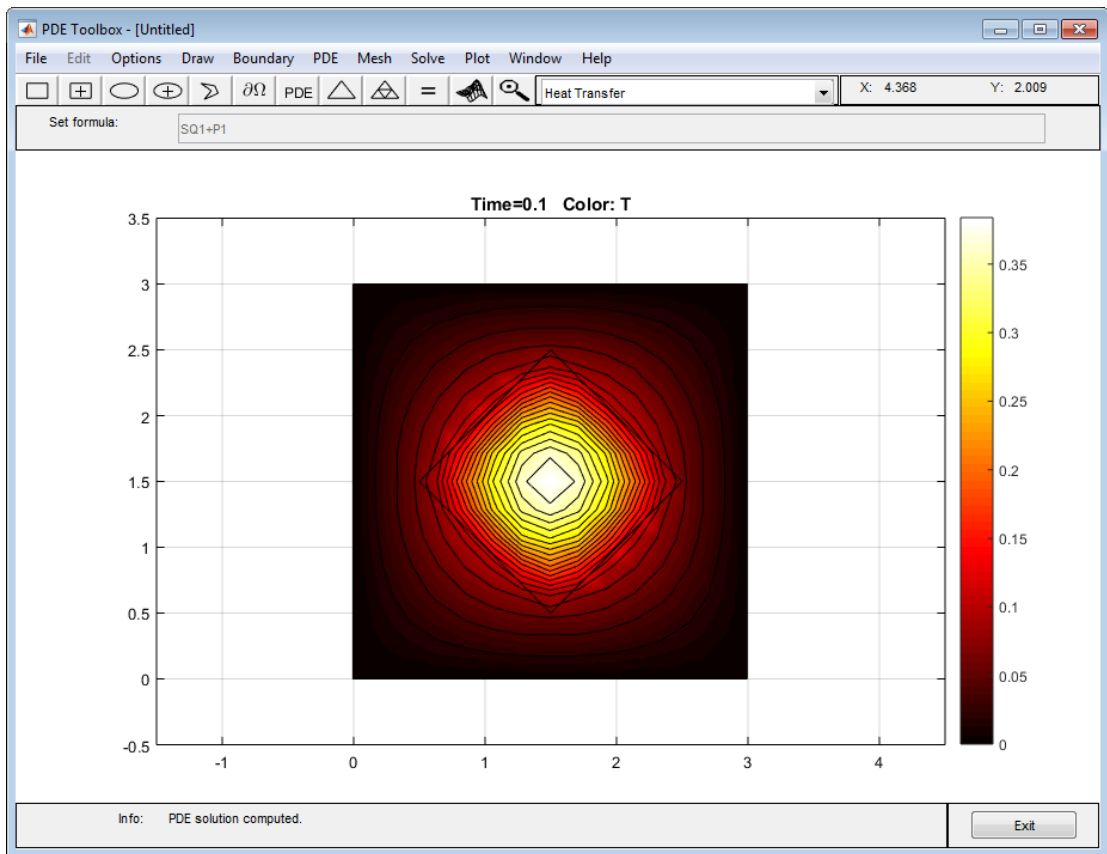
Isothermal Lines
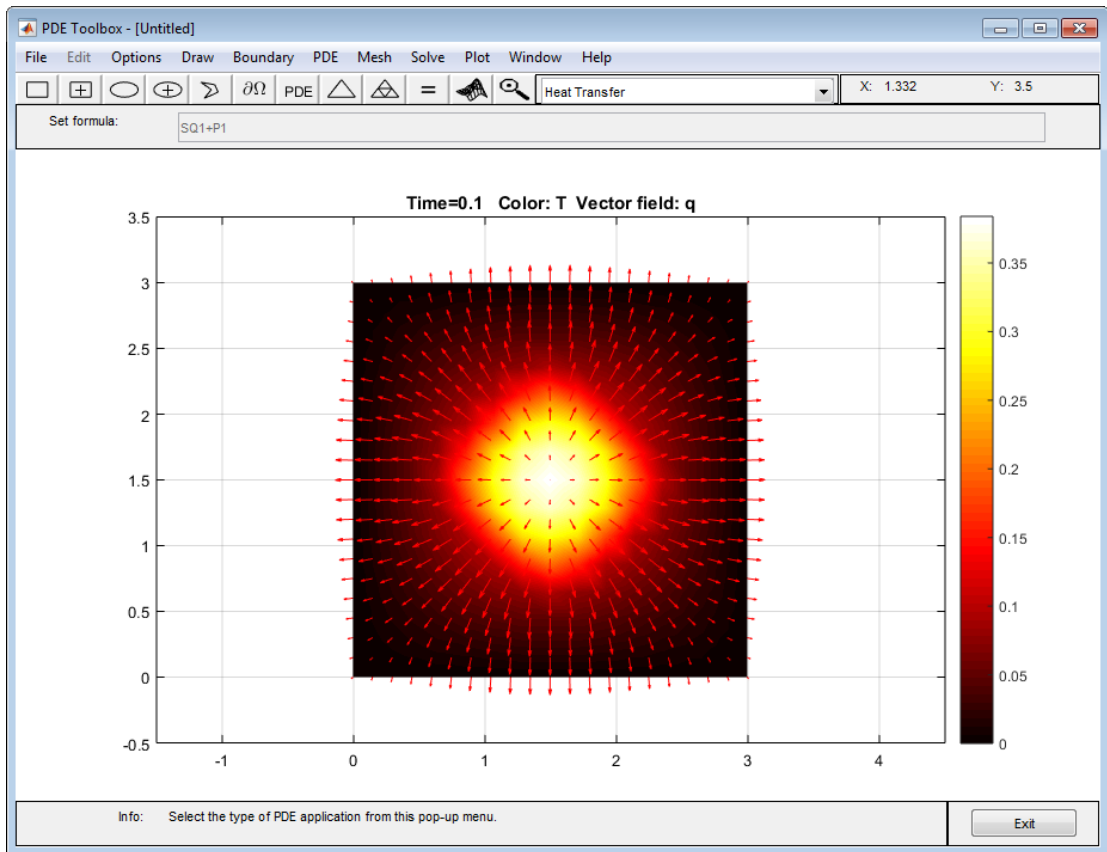
Temperature and Heat Flux

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

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

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

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

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
```

Create the PDE Model with a single dependent variable

```
numberOfPDE = 1;
pdem = createpde(numberOfPDE);
```

Geometry

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
```

```matlab
% y-locations of the corners.
gdm = [3 4 0 width width 0 0 0 height height]';
g = decsg(gdm, 'S1', ('S1')');

% Convert the DECSG geometry into a geometry object
% on doing so it is appended to the PDEModel
geometryFromEdges(pdem,g);

% Plot the geometry and display the edge labels for use in the boundary
% condition definition.
figure;
pdegplot(pdem,'edgeLabels','on');
axis([-.1 1.1 -.1 1.1]);
title 'Geometry With Edge Labels Displayed';
```


Geometry With Edge Labels Displayed
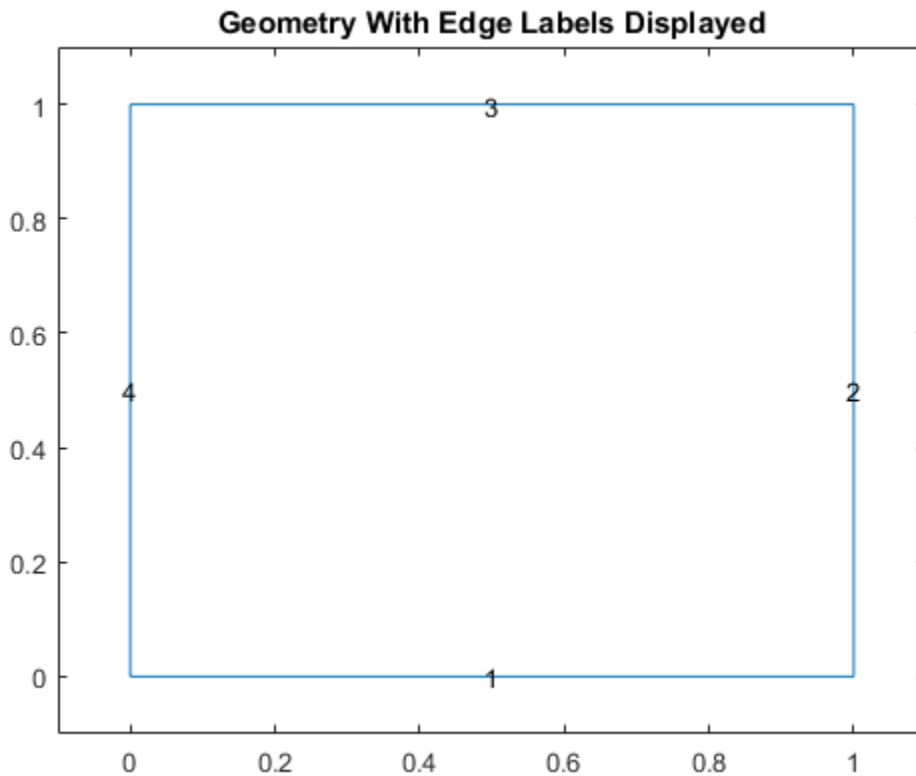
Definition of PDE Coefficients

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 = @(~,state) 2*hCoeff + 2*emiss*stefanBoltz*state.u.^3;
f = 2*hCoeff*ta + 2*emiss*stefanBoltz*ta^4;
d = thick*rho*specificHeat;
specifyCoefficients(pdem,'m',0,'d',0,'c',c,'a',a,'f',f);
```

Boundary Conditions

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

The boundary conditions are defined below. 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,

```
applyBoundaryCondition(pdem,'Edge',1,'u',1000);
```

Initial guess

The initial guess is defined below.

```
setInitialConditions(pdem,0);
```

Mesh

Create the triangular mesh on the square with approximately ten elements in each direction.

```
hmax = .1; % element size
msh = generateMesh(pdem,'Hmax',hmax);
figure;
pdeplot(pdem);
axis equal
```

```
title 'Plate With Triangular Element Mesh'
xlabel 'X-coordinate, meters'
ylabel 'Y-coordinate, meters'
```
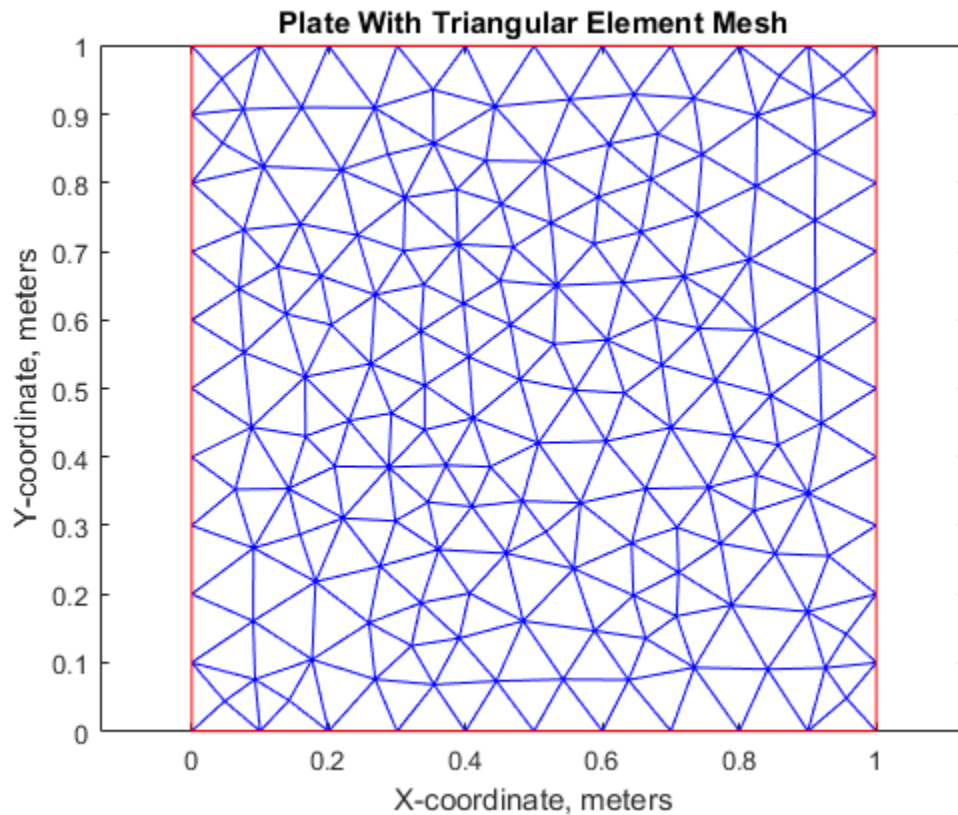

Steady State Solution

Because the a and f coefficients are functions of temperature (due to the radiation boundary conditions), solvepde automatically picks the nonlinear solver to obtain the solution.

```
R = solvepde(pdem);
u = R.NodalSolution;
figure;
```

```
pdeplot(pdem,'xydata',u,'contour','on','colormap','jet');
title 'Temperature In The Plate, Steady State Solution'
xlabel 'X-coordinate, meters'
ylabel 'Y-coordinate, meters'
axis equal
p=msh.Nodes;
plotAlongY(p,u,0);
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 = 449.2 degrees-K

Temperature As a Function of the Y-Coordinate
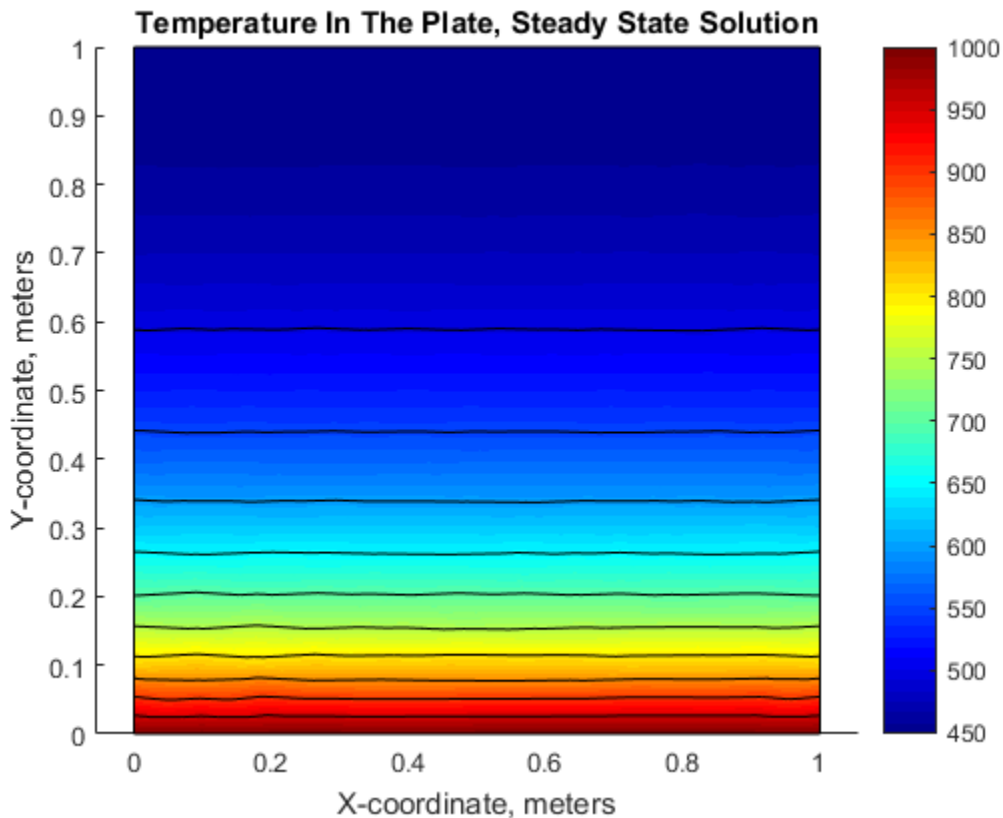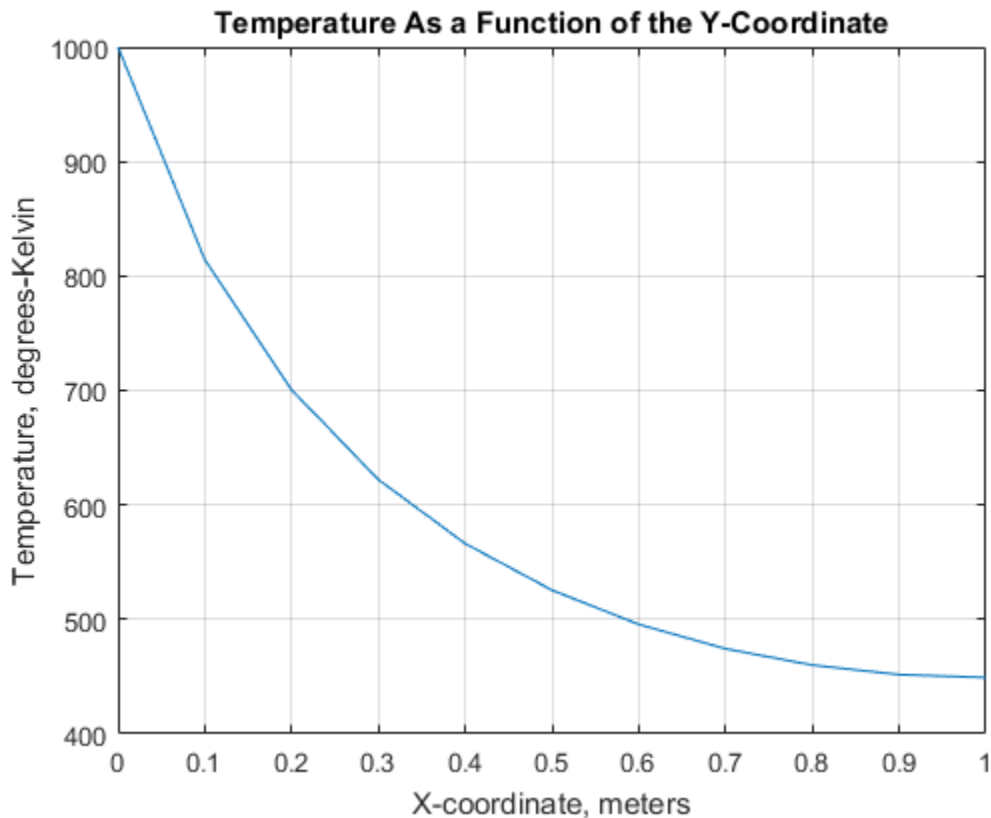
Transient Solution

Include the "d" coefficient

```
specifyCoefficients(pdem,'m',0,'d',d,'c',c,'a',a,'f',f);
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;
```

```
ic = @(~) u0;
setInitialConditions(pdem,ic);
% Set solver options
pdem.SolverOptions.RelativeTolerance = 1.0e-3;
pdem.SolverOptions.AbsoluteTolerance = 1.0e-4;
% |solvepde| automatically picks the parabolic solver to obtain the solution.
R = solvepde(pdem,tlist);
u = R.NodalSolution;
figure;
plot(tlist,u(3, :));
grid on
title 'Temperature Along the Top Edge of the Plate as a Function of Time'
xlabel 'Time, seconds'
ylabel 'Temperature, degrees-Kelvin'
%
figure;
pdeplot(pdem,'xydata',u(:,end),'contour','on','colormap','jet');
title(sprintf('Temperature In The Plate, Transient Solution( %d seconds)\n', ...
  tlist(1,end)));
xlabel 'X-coordinate, meters'
ylabel 'Y-coordinate, meters'
axis equal;
%
fprintf('\nTemperature at the top edge of the plate(t = %5.1f secs) = %5.1f degrees-K\n
  tlist(1,end), u(4,end));


Temperature at the top edge of the plate(t = 5000.0 secs) = 447.6 degrees-K
```

Temperature Along the Top Edge of the Plate as a Function of Time
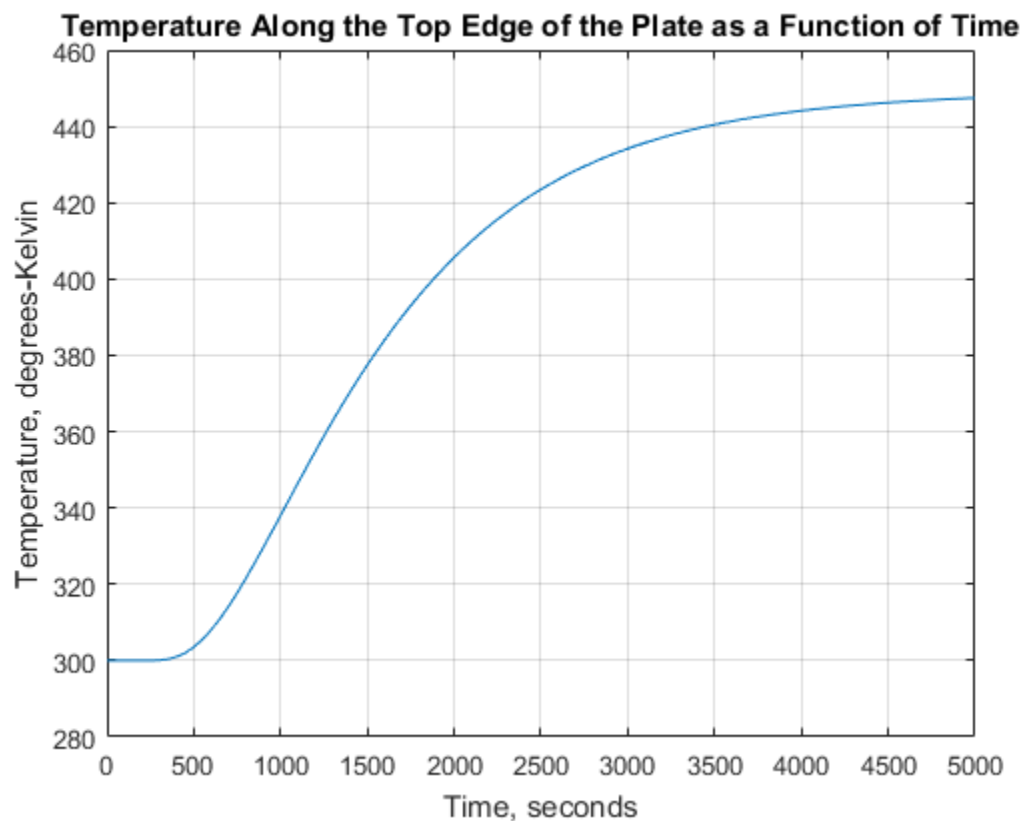
Temperature In The Plate, Transient Solution(5000 seconds)
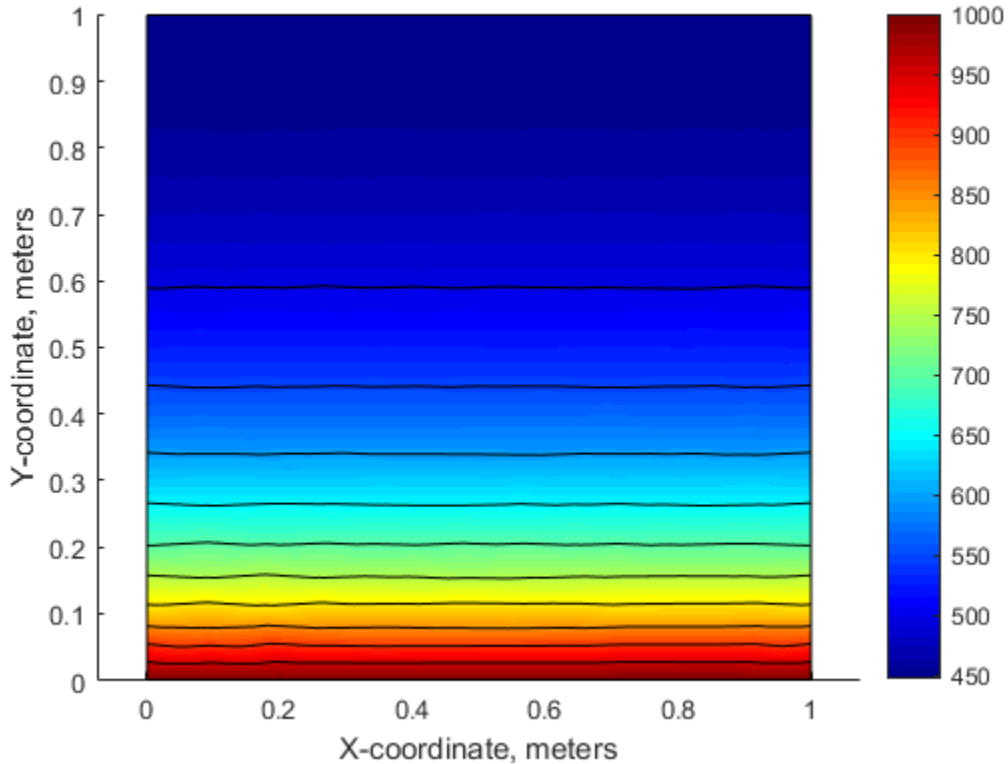
Summary

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 PDE app. For more information, see "Specify Coefficients in the PDE App" on page 4-8.

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.

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
$-\Delta U = 1$ in Ω, $U = 0$ 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 PDE App

With the PDE app started, perform the following steps using the generic scalar mode:

1   Using the **Option** menu, 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.
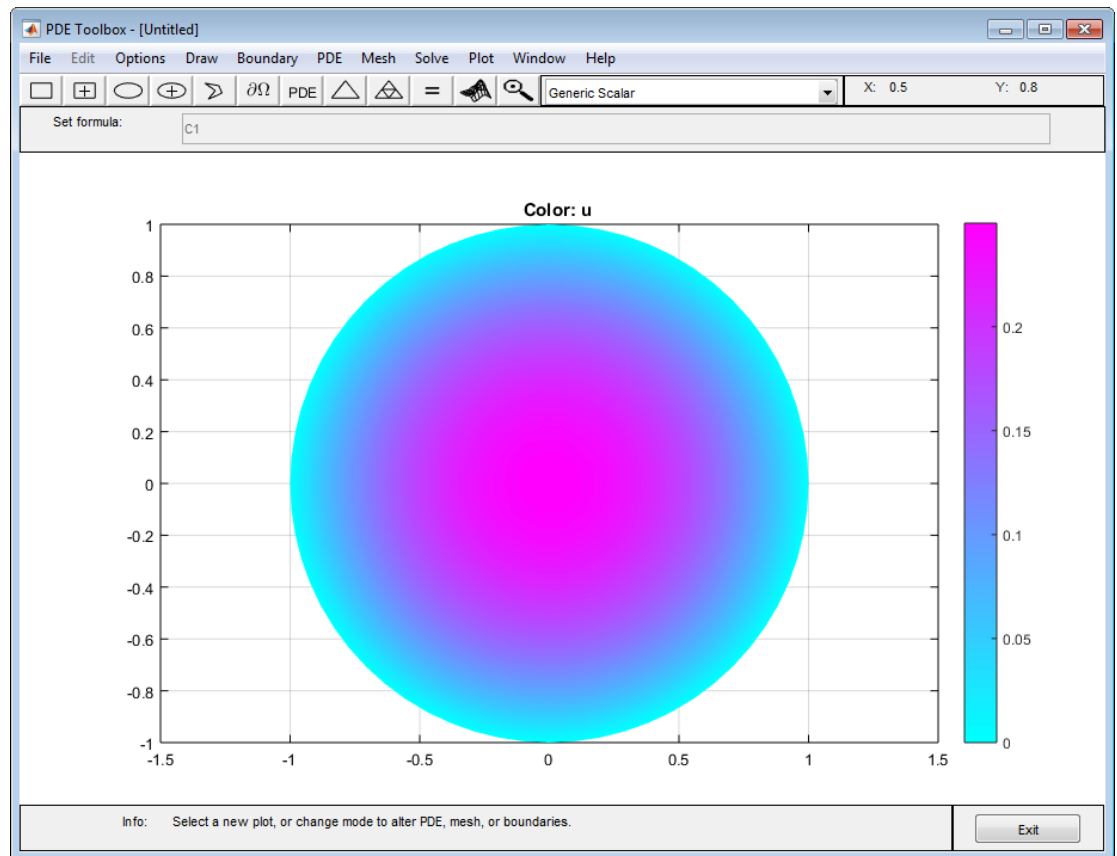
7 To compare the numerical solution to the exact solution, select the `user entry` in the **Property** pop-up menu for **Color** in the Plot Selection dialog box. Then input the MATLAB expression `u-(1-x.^2-y.^2)/4` in the **user entry** edit field. You obtain a plot of the absolute error in the solution.
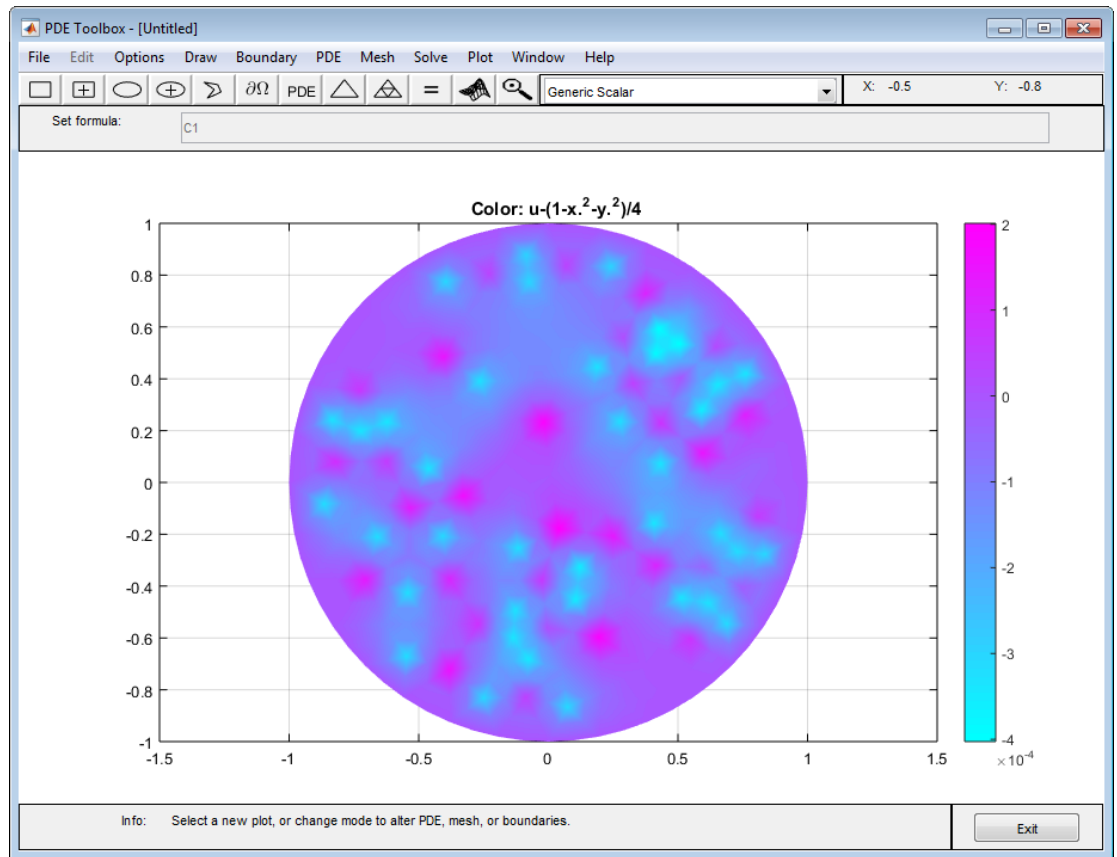
You can also compare the numerical solution to the exact solution by entering some simple command-line-oriented commands. It is easy to export the mesh data and the solution to the MATLAB main workspace by using the **Export** options from the **Mesh** and **Solve** menus. To refine the mesh and solve the PDE successively, simply click the **refine** and **=** buttons until the desired accuracy is achieved. Another possibility is to use the adaptive solver.

Solve Poisson's Equation Using Command-Line Functions

This example shows how to solve Poisson's equation using command-line functions. The code compares the solution with an analytic solution, and refines the mesh until the solutions are close.

First you must create a 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. You can display the file by typing `type circleg`.

The boundary conditions are Dirichlet, zero at each edge.

Now you can start working at the command line:

```matlab
model = createpde();
geometryFromEdges(model,@circleg);
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',0);
c = 1;
a = 0;
f = 1;
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);
hmax = 1;
generateMesh(model,'Hmax',hmax);
error = []; err = 1;
while err > 0.001, % run until error <= 0.001
    hmax = hmax/2;
    generateMesh(model,'Hmax',hmax);% refine mesh
    results = solvepde(model);
    u = results.NodalSolution;
    p = model.Mesh.Nodes;
    exact = -(p(1,:).^2+p(2,:).^2-1)/4;
    err = norm(u-exact',inf); % compare with exact solution
    error = [error err]; % keep history of err
end
plot(error,'rx','MarkerSize',12);
ax = gca;
ax.XTick = 1:numel(error);
title('Error History');
xlabel('Iteration');
ylabel('Norm of Error');
figure
pdeplot(model,'xydata',u-exact') % plot error
title('Final Error');
```

```
xlabel('x')
ylabel('y')
```
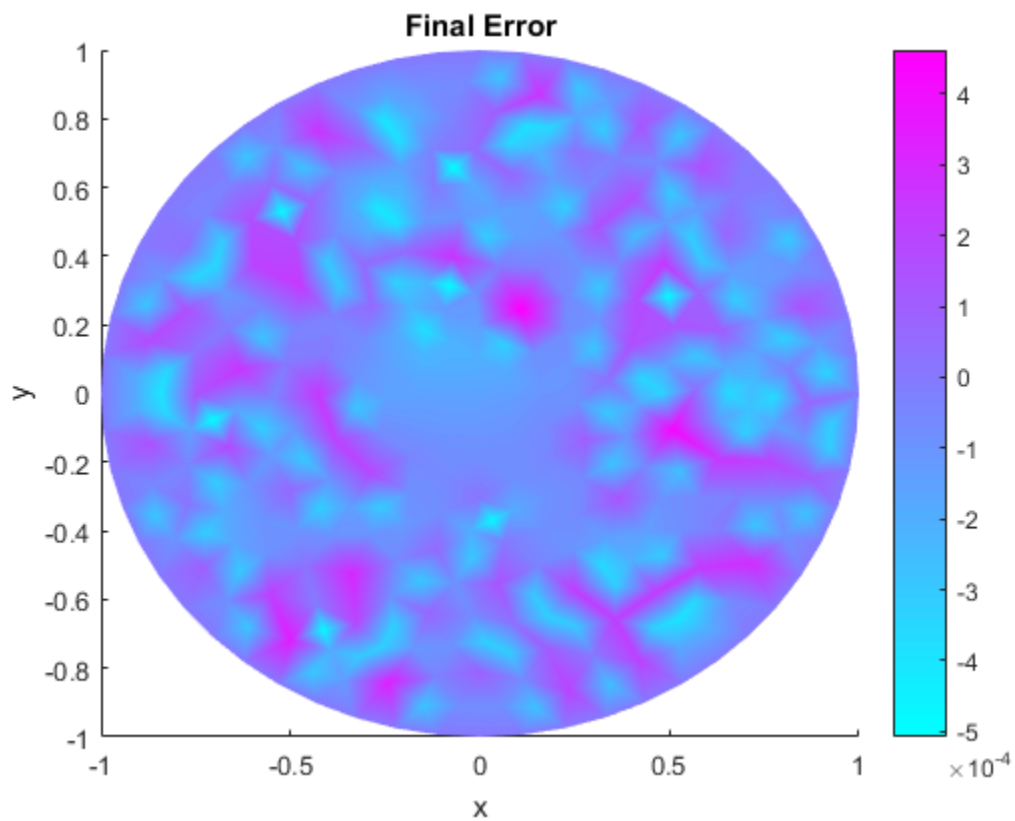
pdedemo1 also shows the solution to this problem.

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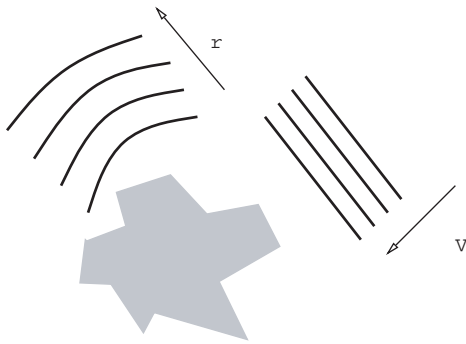
We assume that the medium is homogeneous so that the wave speed is constant, c.

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 PDE App

You can now use the PDE app 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 = -k^2 = -3600, and f = 0. Open the PDE Specification dialog box and enter these values.

The problem can now be solved, and the solution is complex. For a complex solution, the real part is plotted and a warning message is issued.
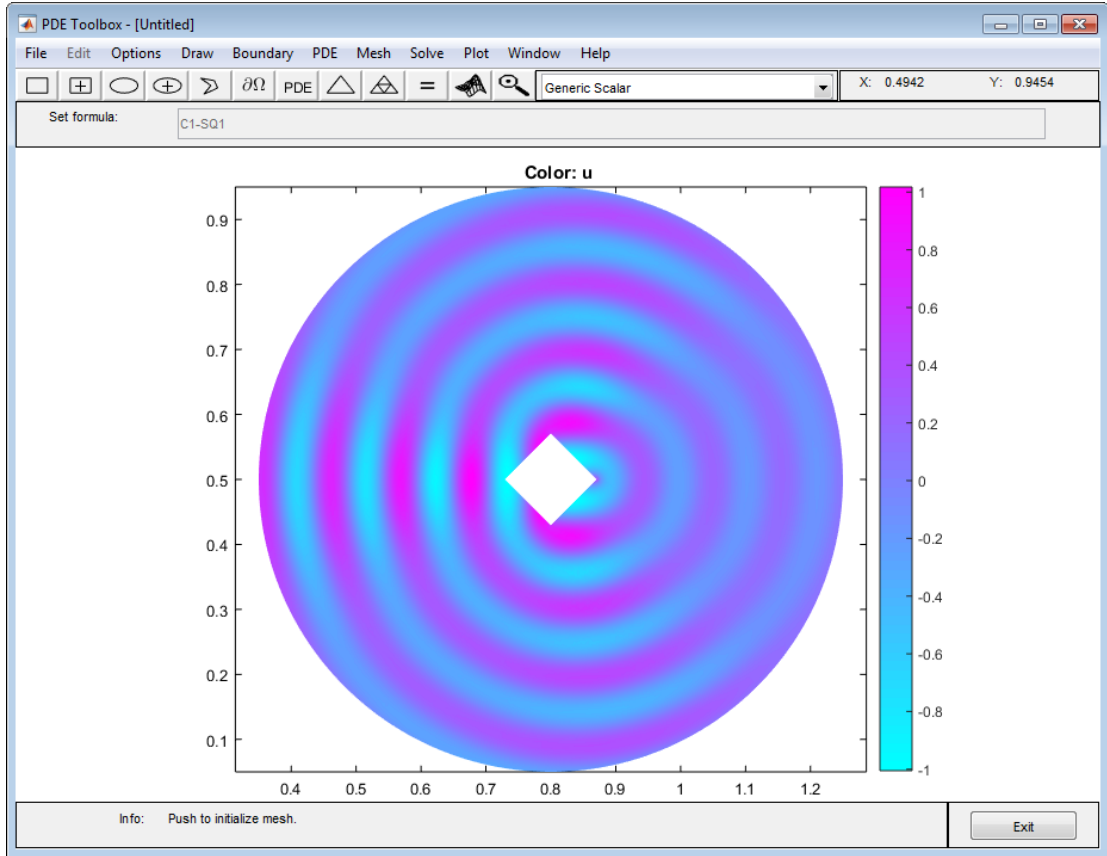
The propagation of the reflected waves is computed as
$\mathrm{Re}(r(x,y)e^{-i\omega t})$,

which is the reflex of

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

To see the whole field, plot

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

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

To make an animation of the reflected wave, first export the solution and the mesh data to the MATLAB workspace. To export the mesh, select **Export Mesh** from the **Mesh** menu. To export the solution, select **Export Solution** from the **Solve** menu.

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

This example shows how to solve this minimal surface problem using both the PDE app and command-line functions.

Using the PDE App

Make sure that the application mode in the PDE app 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.

Minimal Surface Problem on the Unit Disk

This example shows how to solve a nonlinear elliptic problem.

A Nonlinear PDE

A nonlinear problem is one whose coefficients not only depend on spatial coordinates, but also on the solution itself. An example of this is the minimal surface equation

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

on the unit disk, with

$$u(x,y) = x^2$$

on the boundary. To express this equation in toolbox form, note that the elliptic equation in toolbox syntax is

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

The PDE coefficient `c` is the multiplier of ∇u, namely

$$c = \frac{1}{\sqrt{1 + |\nabla u|^2}}.$$

c is a function of the solution u, so the problem is nonlinear. In toolbox syntax, you see that the a and f coefficients are 0.

Geometry

Create a PDE Model with a single dependent variable, and include the geometry of the unit disk. The `circleg` function represents this geometry. Plot the geometry and display the edge labels.

```
numberOfPDE = 1;
model = createpde(numberOfPDE);
geometryFromEdges(model,@circleg);
pdegplot(model,'EdgeLabels','on');
axis equal
```

```
title 'Geometry with Edge Labels';
```


Geometry with Edge Labels
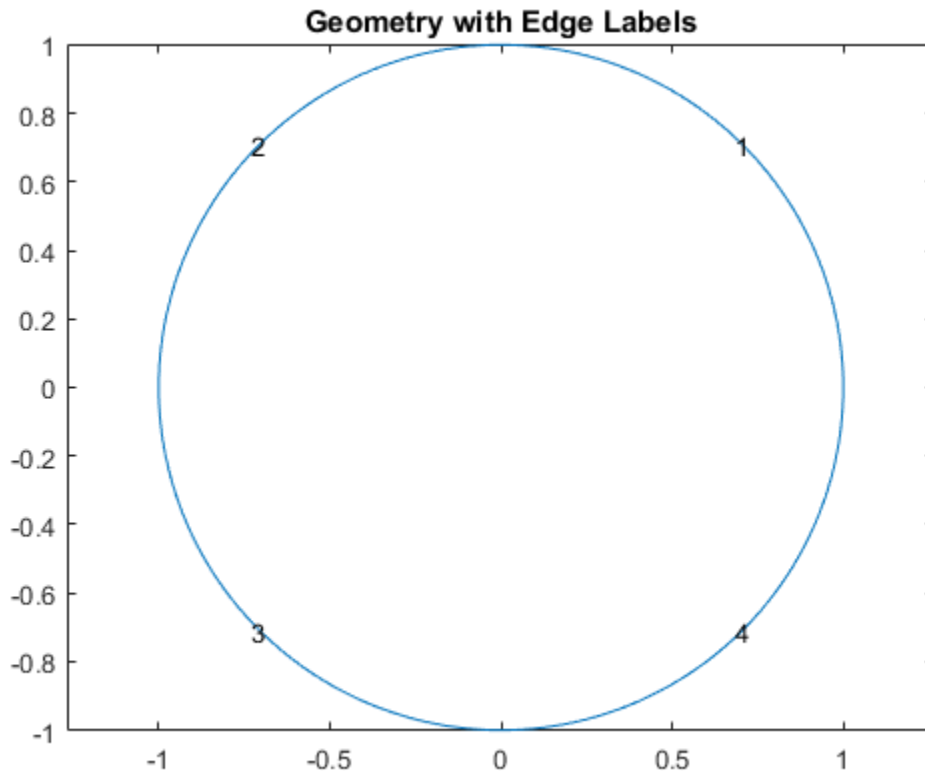
Specify PDE Coefficients

```
a = 0;
f = 0;
cCoef = @(region,state) 1./sqrt(1+state.ux.^2 + state.uy.^2);
specifyCoefficients(model,'m',0,'d',0,'c',cCoef,'a',a,'f',f);
```

Boundary Conditions

Create a function that represents the boundary condition $u(x, y) = x^2$.

```
bcMatrix = @(region,~)region.x.^2;
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',bcMatrix);
```

Generate Mesh

```
generateMesh(model,'Hmax',0.1);
figure;
pdemesh(model);
axis equal
```
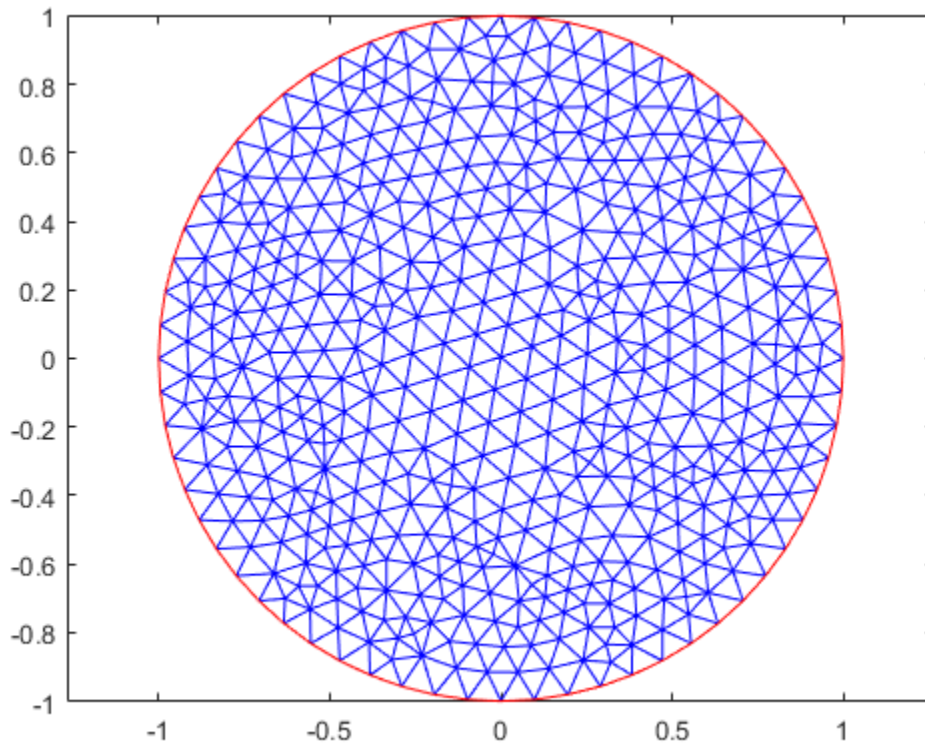

Solve PDE

Because the problem is nonlinear, solvepde invokes nonlinear solver. Observe the solver progress by setting the `SolverOptions.ReportStatistics` property of `model` to `'on'`.

```
model.SolverOptions.ReportStatistics = 'on';
result = solvepde(model);
```

```
u = result.NodalSolution;
```

```
Iteration      Residual      Step size  Jacobian: Full
   0           2.7254e-02
   1           3.5437e-04    1.0000000
   2           1.1057e-06    1.0000000
```

Plot Solution

```
figure;
pdeplot(model,'xydata',u,'zdata',u);
xlabel 'x'
ylabel 'y'
zlabel 'u(x,y)'
title 'Minimal surface'
```
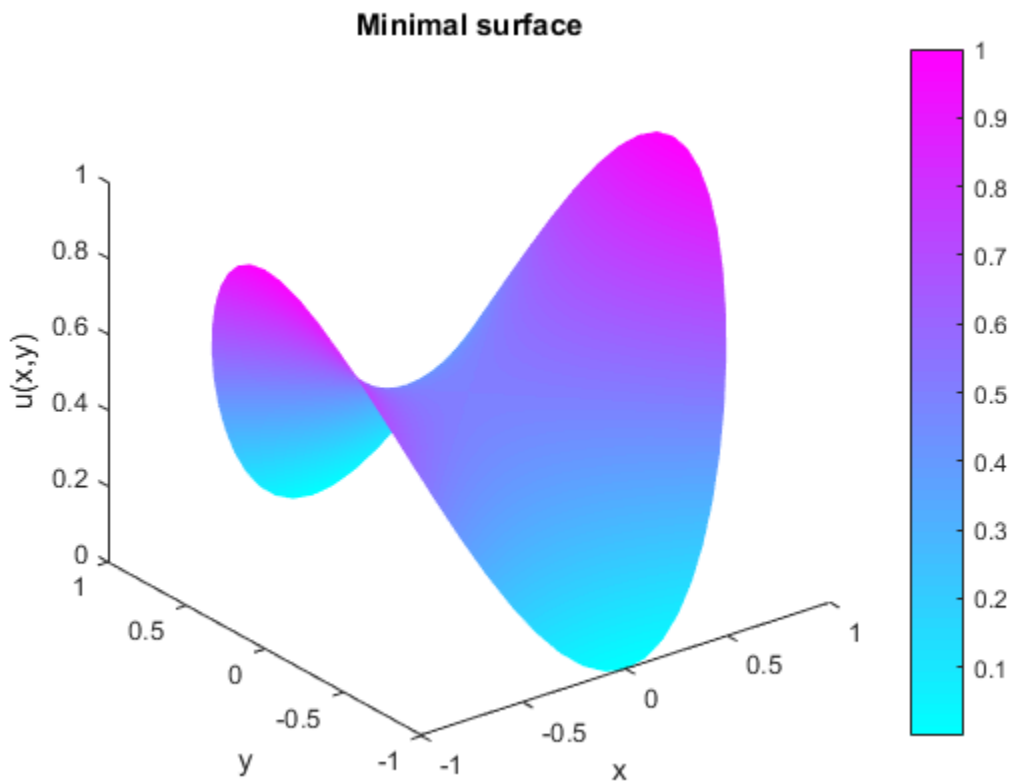
Domain Decomposition Problem

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 the PDE app.

Assume now that Ω is the disjoint union of some subdomains $\Omega_1, \Omega_2, \ldots, \Omega n$. 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)
```
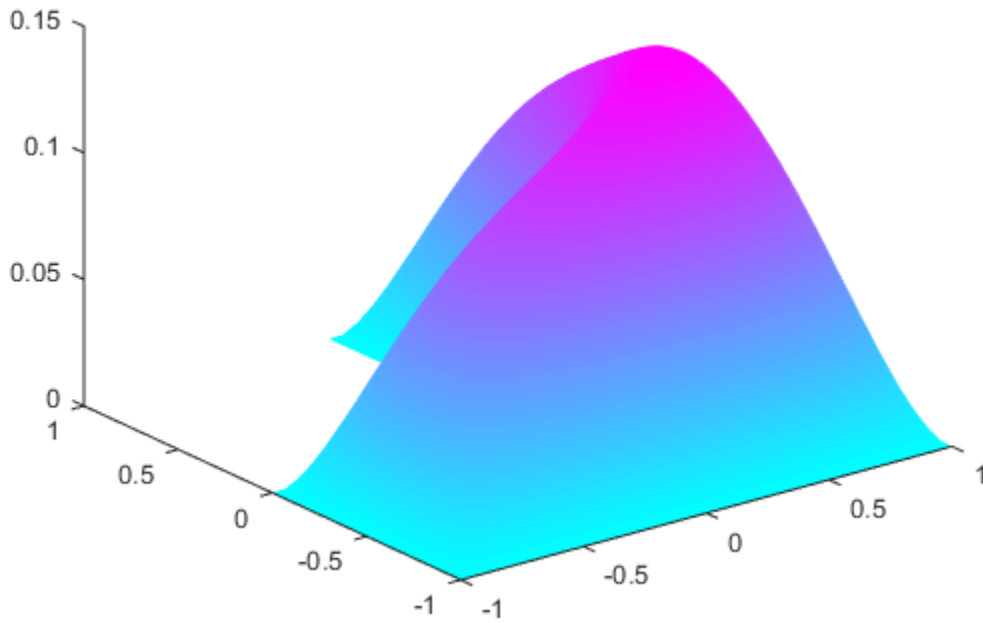
For the entire example, see Poisson's Equation Using Domain Decomposition.

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

- $u = 100$ on the left side (Dirichlet condition)
- $\partial u/\partial n = -10$ 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 PDE App

Once you have started the PDE app 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].

```
pderect([-0.5 0.5 -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 select

both **Grid** and **Snap** from the **Options** menu. A rectangular cavity with the correct dimensions is then easy to draw.
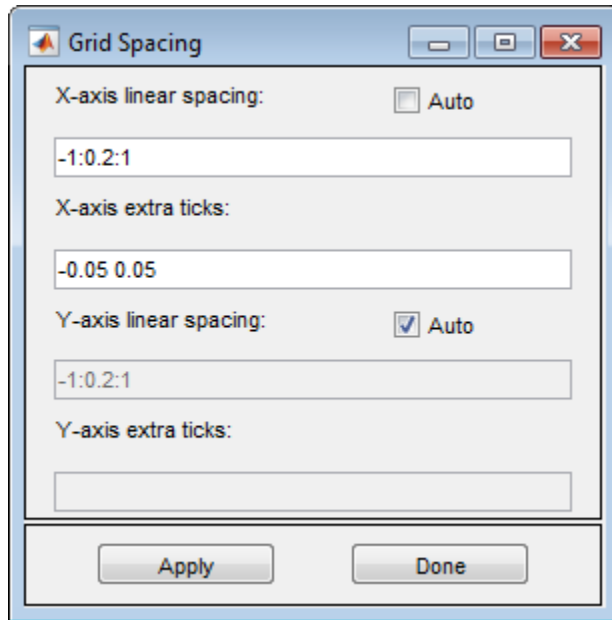
The CSG model of the metal block is now simply expressed as the set formula R1-R2.

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.
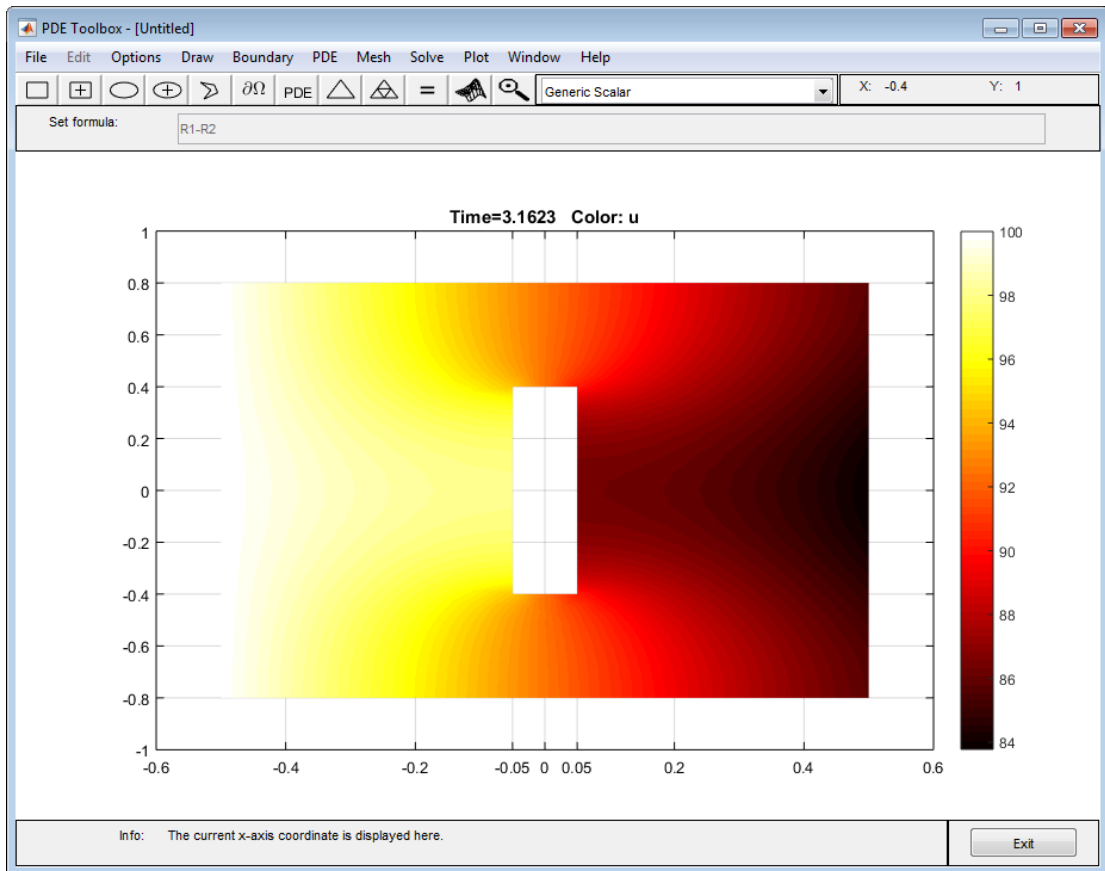
Metal Block Using Command-Line Functions

This example shows how to solve for the heat distribution in a metal block with cavity using command-line functions. The `crackg.m` file describes the geometry of the metal block.

```
model = createpde();
geometryFromEdges(model,@crackg);
```

Plot the geometry with edge labels.

```
pdegplot(model,'EdgeLabels','on');
```

```
ylim([-1,1])
axis equal
```
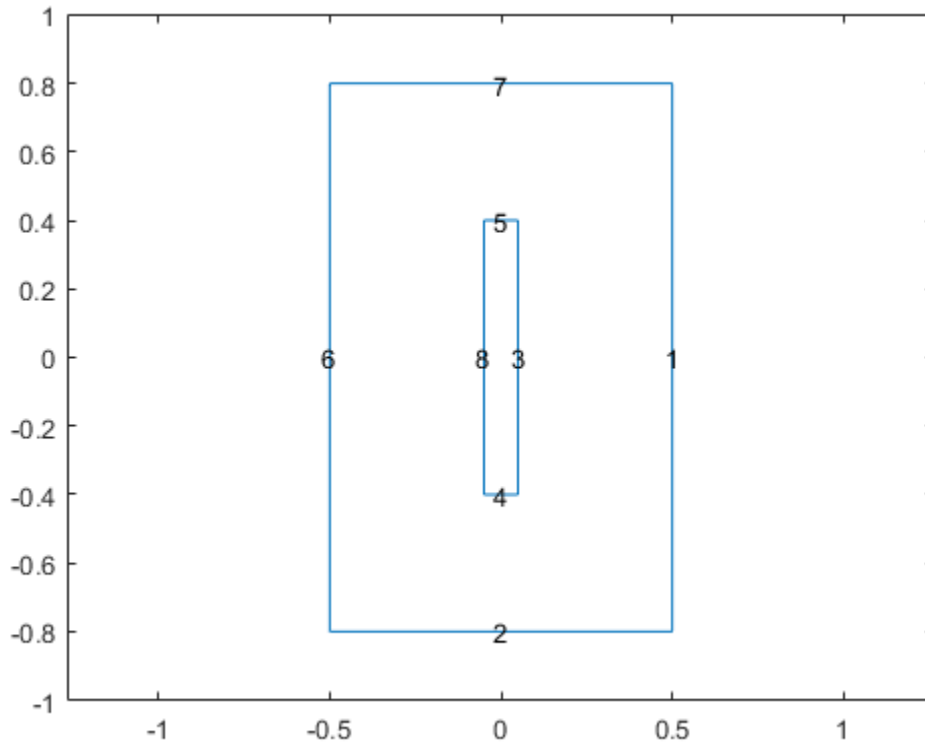

Set the boundary conditions to have the solution u equal to 100 on the left edge (edge 6), and g equal to -10 on the right edge (edge 1). The boundary condition on edge 1 corresponds to constant heat flow to the exterior.

```
applyBoundaryCondition(model,'Edge',6,'u',100);
applyBoundaryCondition(model,'Edge',1,'g',-10);
```

Set an initial value of 0 for the temperature.

```
setInitialConditions(model,0);
```

Set the coefficients to the equation of heat flow.

```
d = 1;
c = 1;
a = 0;
f = 0;
specifyCoefficients(model,'m',0,'d',d,'c',c,'a',a,'f',f);
```

Set solution times to be 0 to 5 in steps of 1/2.

```
tlist = 0:0.5:5;
```

Create a mesh and solve the problem.

```
generateMesh(model);
results = solvepde(model,tlist);
```

Plot the solution at $t = 5.0$ seconds. Use the hot colormap:

```
u = results.NodalSolution(:,end);
pdeplot(model,'xydata',u,'colormap','hot')
```
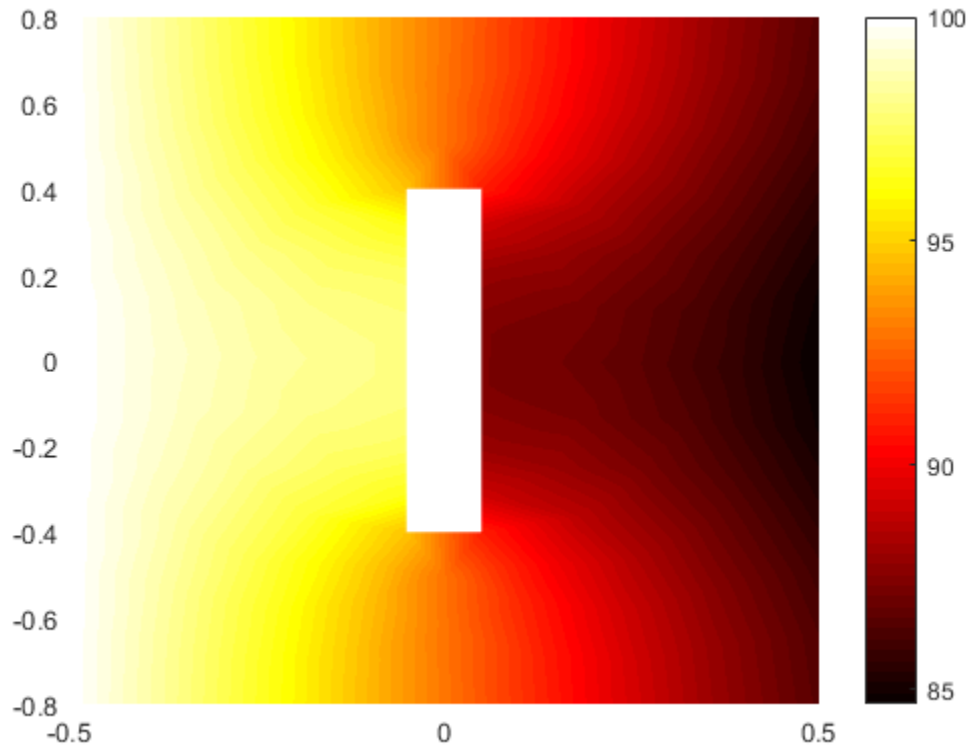
Heat Distribution in a Radioactive Rod

This example shows how to solve a 3-D parabolic PDE problem by reducing the problem to 2-D using coordinate transformation. For a step-by-step command-line solution, see Heat Distribution in a Circular Cylindrical Rod.

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 PDE App

Solve this problem using the PDE app. 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.

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 PDE App

Use the PDE app 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))
```
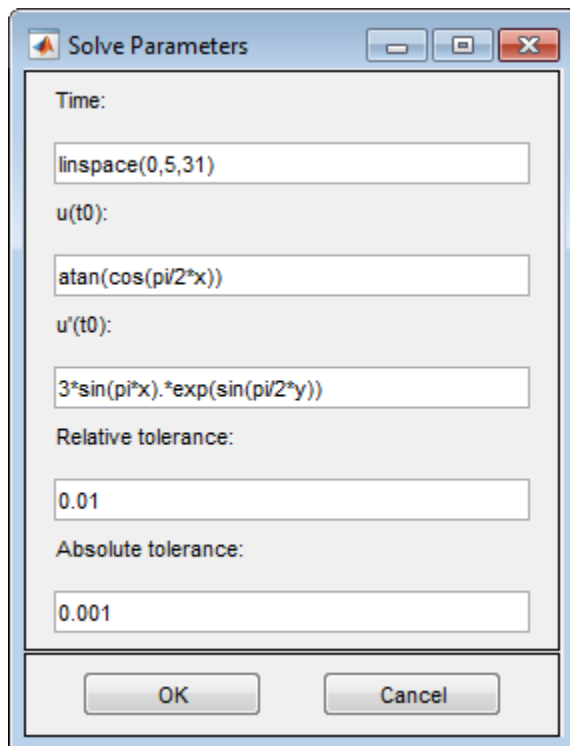

Finally, click the **=** button to compute the solution. The best plot for viewing the waves moving in the x and y directions is an animation of the whole sequence of solutions.

Animation is a very memory-consuming feature, so you may have to cut down on the number of times you 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.

Wave Equation Using Command-Line Functions

This example shows how to solve the wave equation using command-line functions. It solves the equation with boundary conditions $u = 0$ at the left and right sides, and $\partial u / \partial n = 0$ at the top and bottom. The initial conditions are

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

and

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

Calculate the solution every 0.05 seconds for five seconds.

The geometry is described in the file `squareg.m`.

First, create the geometry.

```
model = createpde();
geometryFromEdges(model,@squareg);
```

View the geometry

```
pdegplot(model,'EdgeLabels','on');
ylim([-1.1,1.1])
axis equal
```
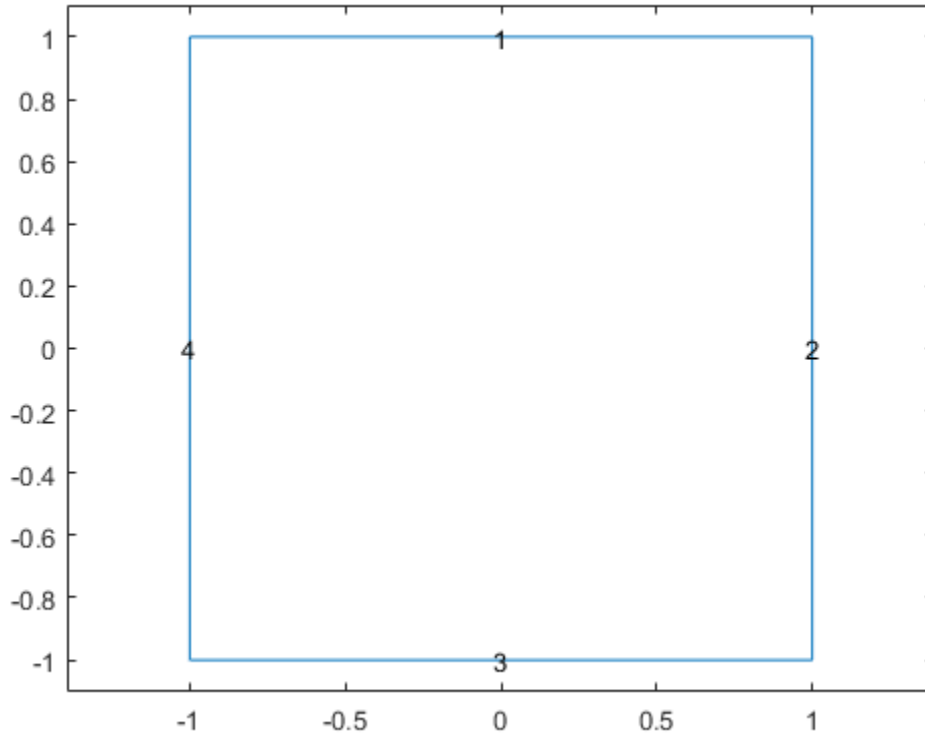
Set the boundary conditions to $u = 0$ on the left and right boundaries (edges 2 and 4). The default boundary conditions are appropriate for edges 1 and 3, so you do not need to set them.

```
applyBoundaryCondition(model,'Edge',[2,4],'u',0);
```

Set the initial conditions.

```
u0 = @(locations)atan(cos(pi/2*locations.x));
ut0 = @(locations)3*sin(pi*locations.x).*exp(sin(pi/2*locations.y));
setInitialConditions(model,u0,ut0);
```

Create the model coefficients. The equation is

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

where $m = 1$, $c = 1$, and $f = 0$.

```
m = 1;
c = 1;
f = 0;
specifyCoefficients(model,'m',m,'d',0,'c',c,'a',0,'f',f);
```

Set the solution times.

```
n = 31; % number of frames in eventual animation
tlist = linspace(0,5,n); % list of times
```

You are now ready to solve the wave equation. Create a mesh and call the solver.

```
generateMesh(model);
results = solvepde(model,tlist);
```

Animate the solution.

```
u = results.NodalSolution;

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

figure;
M = moviein(n);
for i=1:n,
    pdeplot(model,'xydata',u(:,i),'zdata',u(:,i),...
    'xygrid','on','colorbar','off');
    axis([-1 1 -1 1 umin umax]);
    caxis([umin umax]);
    M(:,i) = getframe;
end
```
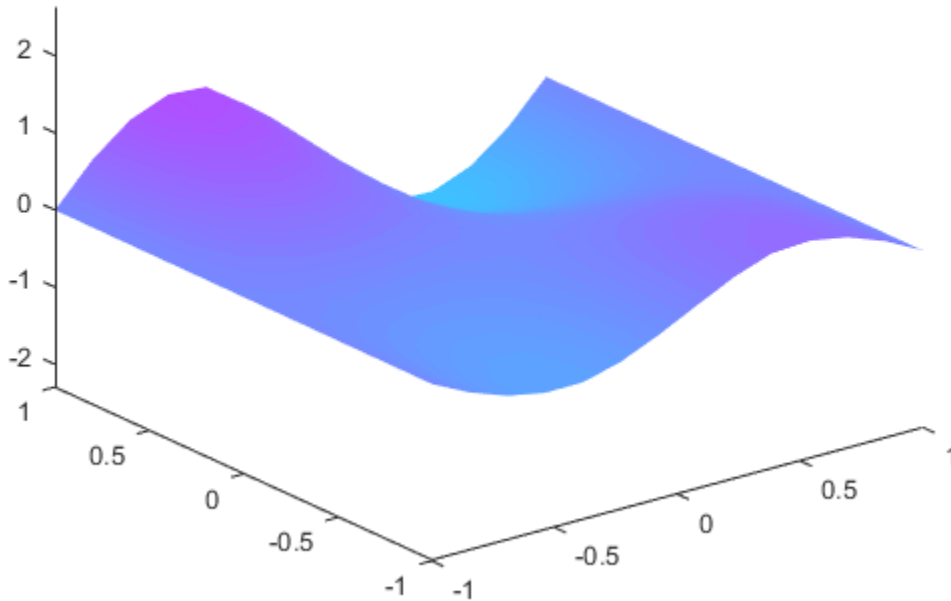
You can review the movie by executing `movie(M,10)`. There is a complete solution of this problem, including animation, in `pdedemo6`. If you have lots of memory, you can try increasing `n`, the number of frames in the movie.

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 PDE App

With the PDE app 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 PDE app. 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.

Eigenvalues for the L-Shaped Membrane Using Command-Line Functions

This example shows how to calculate eigenvalues and eigenvectors using command-line functions. The geometry of the L-shaped membrane is described in the file `lshapeg`. Create a model and include this geometry.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
```

Set zero Dirichlet boundary conditions on all edges.

```
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',0);
```

Recall the general eigenvalue PDE problem description:

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

Create coefficients for the problem

$$-\nabla \cdot (\nabla u) = \lambda u.$$

This has coefficients `d` = 1 and `c` = 1, and all other coefficients equal to zero.

```
specifyCoefficients(model,'m',0,'d',1,'c',1,'a',0,'f',0);
```

Set the interval `[0,100]` as the region for the eigenvalues in the solution.

```
r = [0,100];
```

Create a mesh and solve the problem.

```
generateMesh(model,'Hmax',0.05);
results = solvepdeeig(model,r);
```

```
            Basis= 10,  Time=  0.14,  New conv eig=  0
            Basis= 13,  Time=  0.14,  New conv eig=  0
            Basis= 16,  Time=  0.14,  New conv eig=  0
            Basis= 19,  Time=  0.14,  New conv eig=  3
            Basis= 22,  Time=  0.34,  New conv eig=  3
```

```
                Basis= 25,  Time=   0.34,  New conv eig=  5
                Basis= 28,  Time=   0.34,  New conv eig=  6
                Basis= 31,  Time=   0.34,  New conv eig=  8
                Basis= 34,  Time=   0.55,  New conv eig= 12
                Basis= 37,  Time=   0.55,  New conv eig= 14
                Basis= 40,  Time=   0.55,  New conv eig= 14
                Basis= 43,  Time=   0.55,  New conv eig= 15
                Basis= 46,  Time=   0.80,  New conv eig= 16
                Basis= 49,  Time=   0.80,  New conv eig= 17
                Basis= 52,  Time=   1.05,  New conv eig= 20
End of sweep: Basis= 52,  Time=   1.05,  New conv eig= 20
                Basis= 30,  Time=   1.26,  New conv eig=  0
                Basis= 33,  Time=   1.26,  New conv eig=  0
                Basis= 36,  Time=   1.51,  New conv eig=  0
End of sweep: Basis= 36,  Time=   1.51,  New conv eig=  0
```

See how many solutions you obtained.

```
length(results.Eigenvalues)
```

```
ans =

    19
```

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

```
u = results.Eigenvectors;
pdeplot(model,'xydata',u(:,1),'zdata',u(:,1));
figure
membrane(1,20,9,9)
```
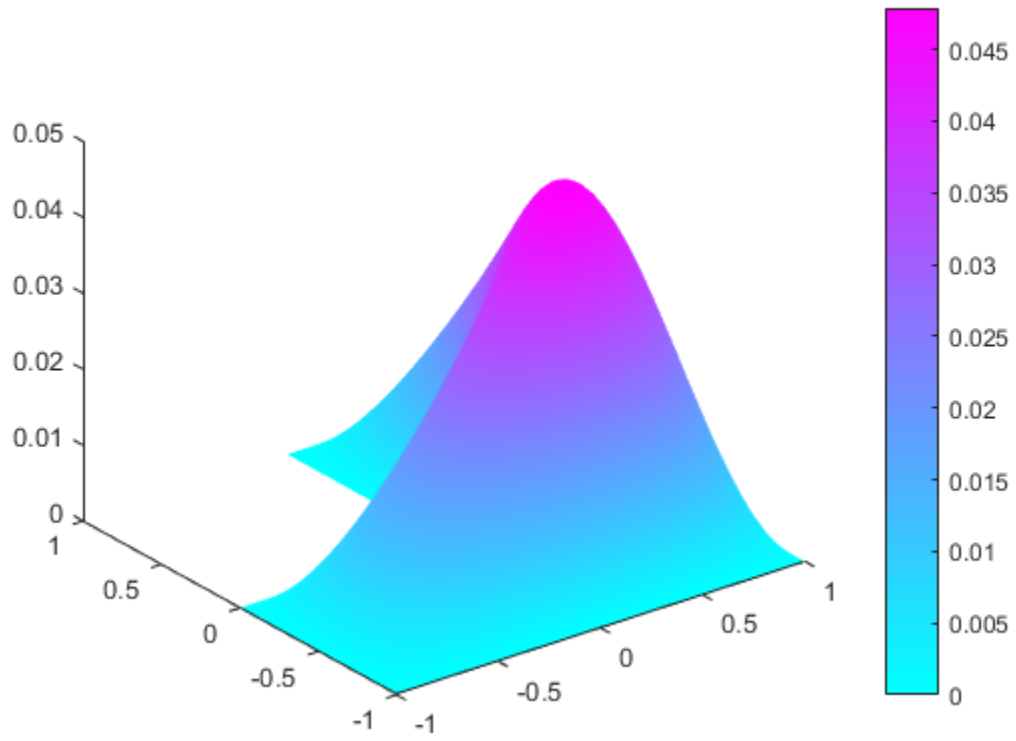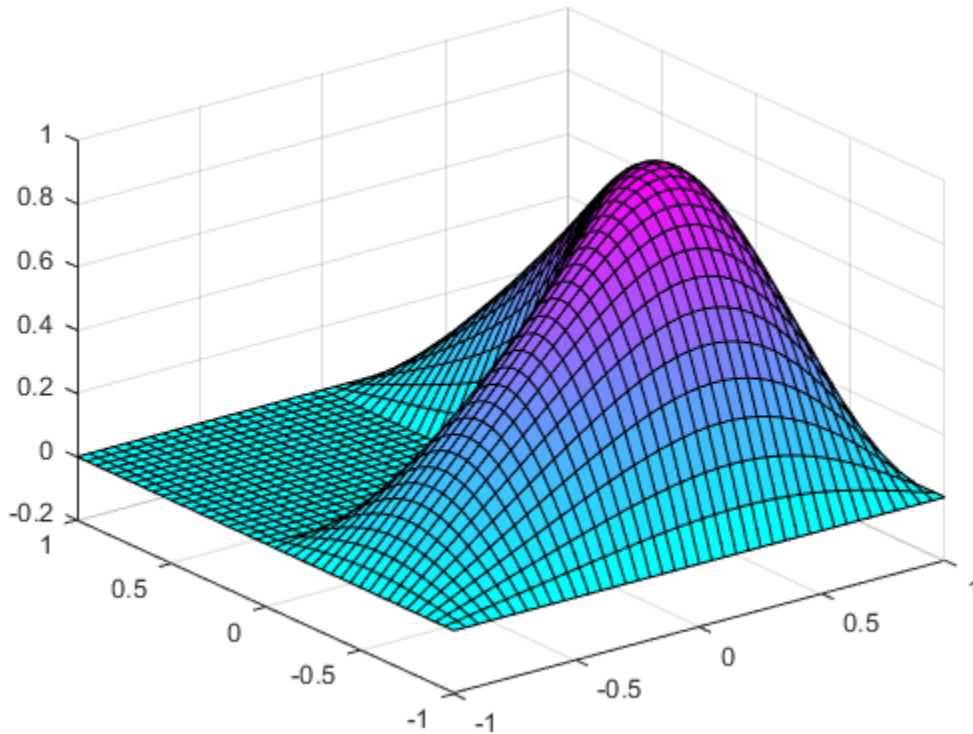
Eigenvectors can be multiplied by any scalar and remain eigenvectors. This explains the difference in scale that you see.

`membrane` can produce the first 12 eigenfunctions for the L-shaped membrane. Compare also the 12th eigenmodes. Multiply the PDE solution by -1 to have the plots look similar instead of inverted.

```
figure
pdeplot(model,'xydata',-u(:,12),'zdata',-u(:,12));
figure
membrane(12,20,9,9)
```
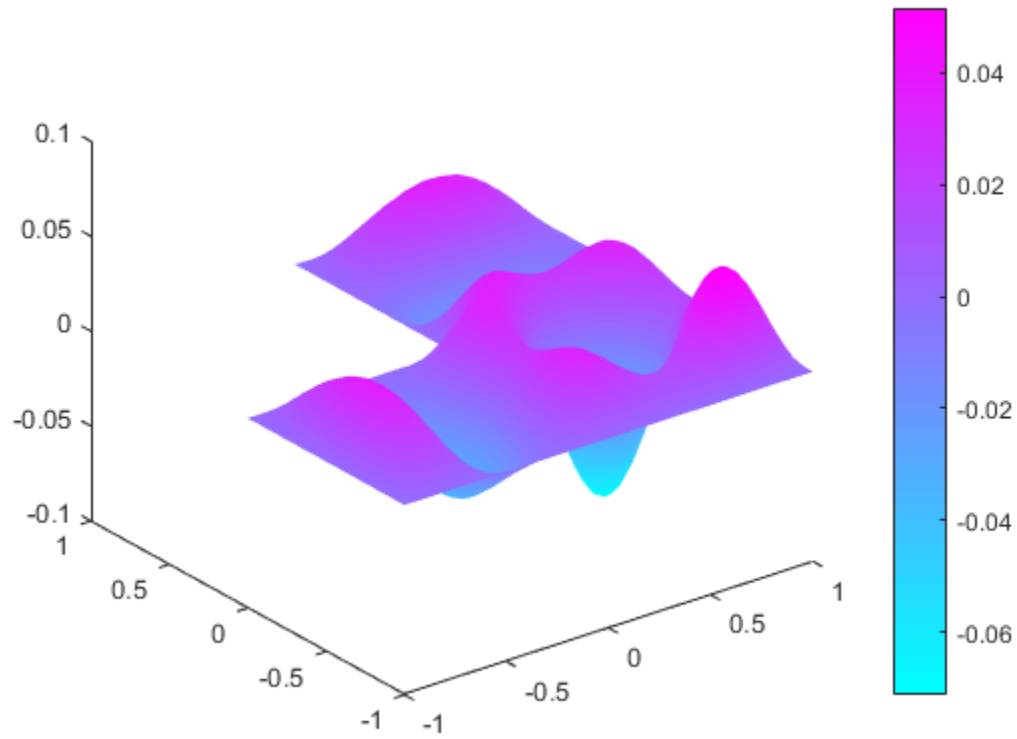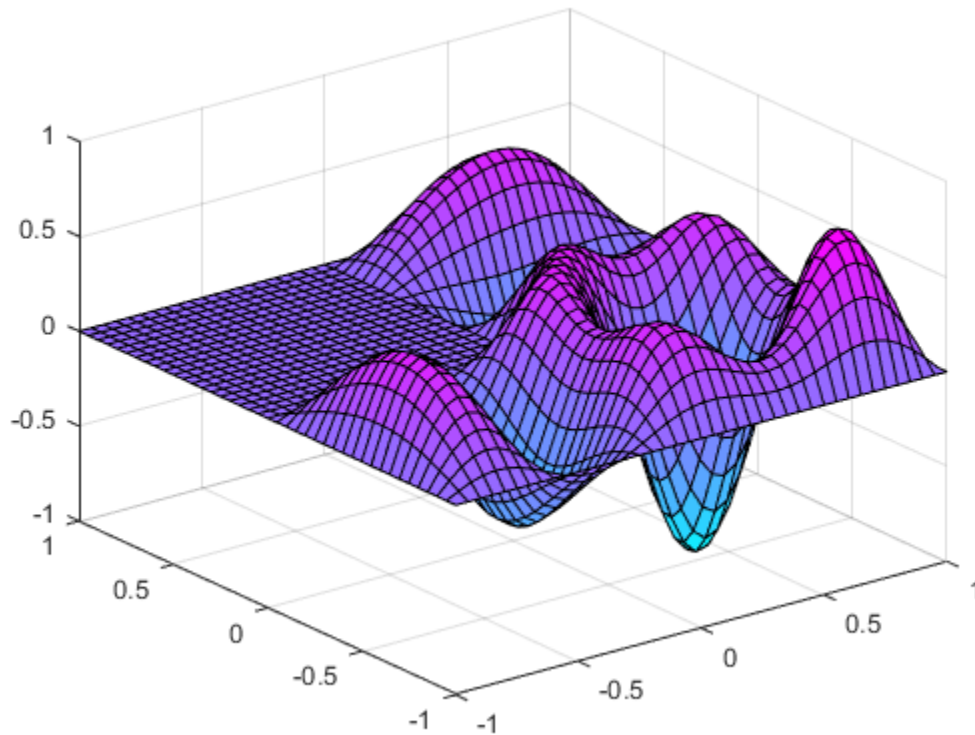
Actually, the eigenvalues of the square can be computed exactly. They are

$$\left(m^2 + n^2\right) \pi^2,$$

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

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 the PDE app. 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.
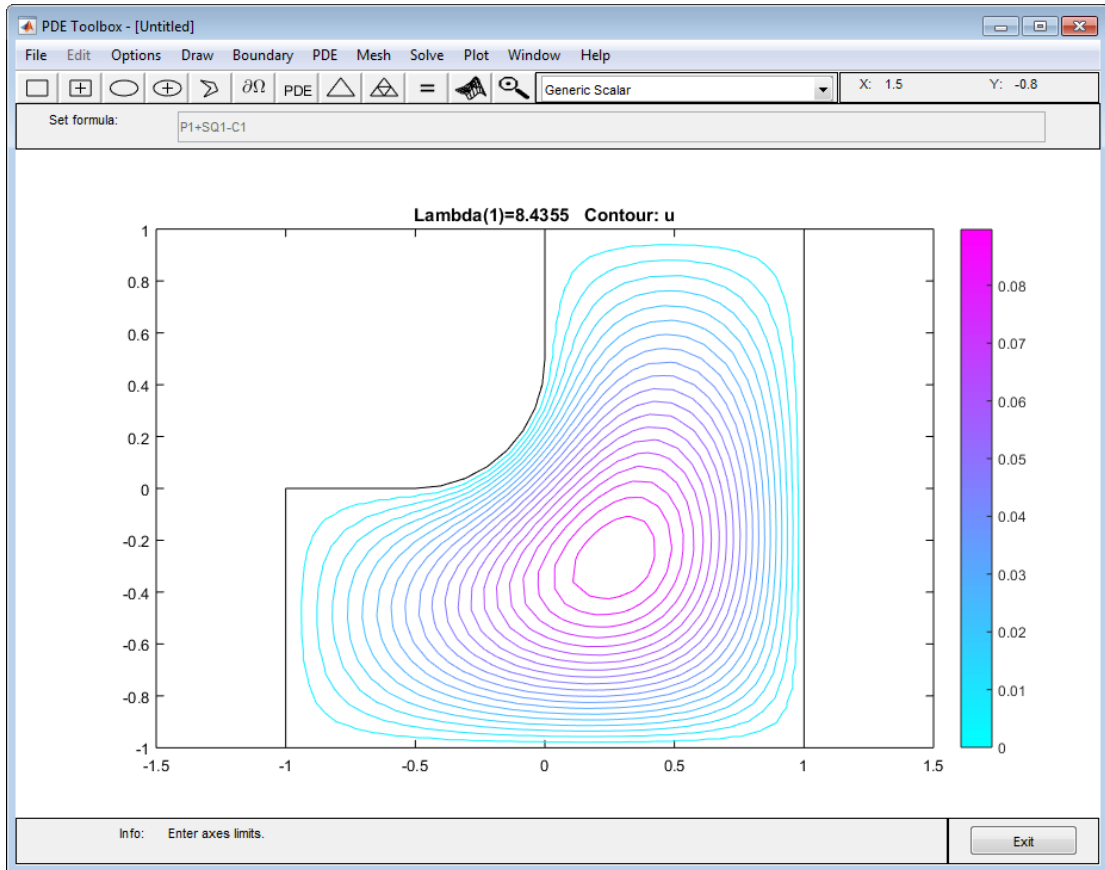
First Eigenmode for an L-Shaped Membrane with a Rounded Corner

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 PDE App

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

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.

Eigenvalues of a Square Using Command-Line Functions

This example shows how to compute the eigenvalues and eigenmodes of a square domain using command-line functions. The geometry description file file for this problem is called `squareg.m`.

Create a model and import the geometry.

```
model = createpde();
geometryFromEdges(model,@squareg);
```

Set zero Neumann boundary conditions on all edges. These are the default conditions, so this step is optional.

```
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'g',0);
```

The eigenvalue PDE coefficients for this problem are $c = 1$, $a = 0$, and $d = 1$. You can enter the eigenvalue range `r` as the vector `[-Inf 10]`.

```
specifyCoefficients(model,'m',0,'d',1,'c',1,'a',0,'f',0);
r = [-Inf,10];
```

Create a mesh and solve the problem.

```
generateMesh(model,'Hmax',0.05);
results = solvepdeeig(model,r);
```

```
                Basis= 10,  Time=   0.47,  New conv eig=  0
                Basis= 12,  Time=   0.47,  New conv eig=  2
                Basis= 14,  Time=   0.64,  New conv eig=  2
                Basis= 16,  Time=   0.64,  New conv eig=  2
                Basis= 18,  Time=   0.64,  New conv eig=  2
                Basis= 20,  Time=   0.67,  New conv eig=  4
```

```
              Basis= 22,  Time=    0.69,  New conv eig=  5
              Basis= 24,  Time=    0.69,  New conv eig=  5
              Basis= 26,  Time=    0.69,  New conv eig=  6
              Basis= 28,  Time=    0.69,  New conv eig=  9
End of sweep: Basis= 28,  Time=    0.69,  New conv eig=  9
              Basis= 19,  Time=    0.90,  New conv eig=  0
End of sweep: Basis= 19,  Time=    0.90,  New conv eig=  0
```

Plot the fourth eigenfunction as a surface plot.

```
u = results.Eigenvectors;
pdeplot(model,'xydata',u(:,4),'zdata',u(:,4));
```
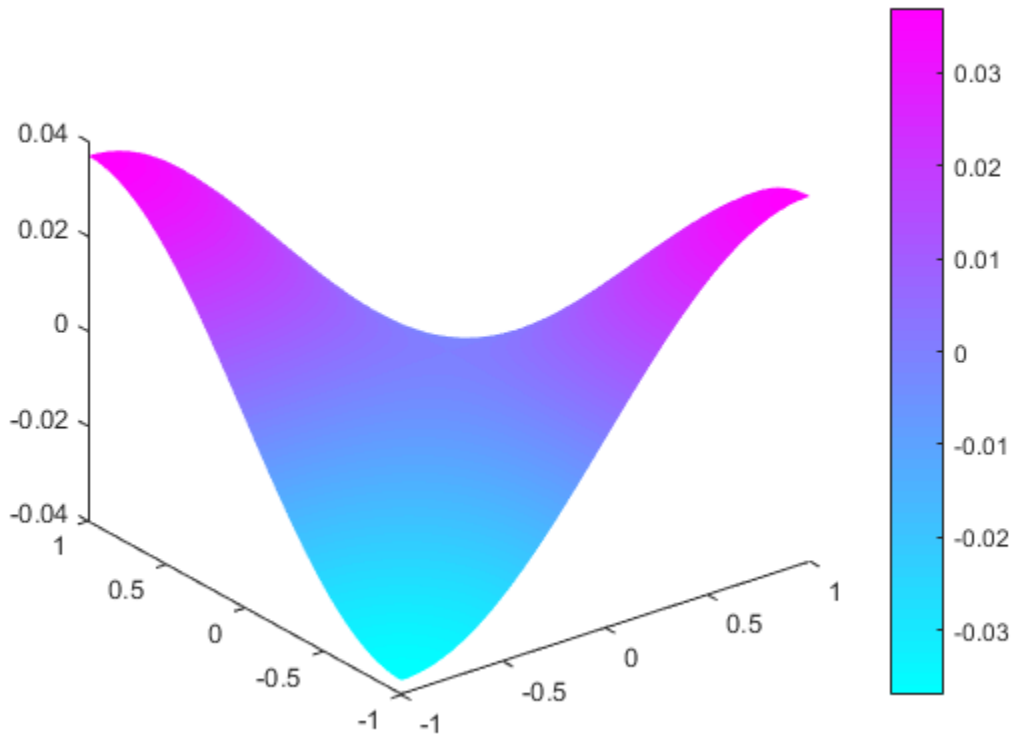

This problem is *separable*, meaning

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

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

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

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 compared to $\pi^2/4$:

```
l = results.Eigenvalues;
l(2)-l(1)
```

```
ans =

    2.4681
```

```
pi^2/4
```

```
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:

```
l(5) - l(1) - pi^2
```

```
ans =

    0.0114
```

You can explore higher modes by increasing the search range to include eigenvalues greater than 10.

Vibration Of a Circular Membrane Using The MATLAB eigs Function

This example shows the calculation of the vibration modes of a circular membrane. The calculation of vibration modes requires the solution of the eigenvalue partial differential equation (PDE). In this example the solution of the eigenvalue problem is performed using both the PDE Toolbox™ `solvepdeeig` solver and the core MATLAB™ `eigs` eigensolver.

The main objective of this example is to show how `eigs` can be used with PDE Toolbox. Generally, the eigenvalues calculated by `solvepdeeig` and `eigs` are practically identical. However, sometimes, it is simply more convenient to use `eigs` than `solvepdeeig`. One example of this is when it is desired to calculate a specified number of eigenvalues in the vicinity of a user-specified target value. `solvepdeeig` requires that a lower and upper bound surrounding this target value be specified. `eigs` requires only that the target eigenvalue and the desired number of eigenvalues be specified.

Create a pde entity for a PDE with a single dependent variable

```
numberOfPDE = 1;
pdem = createpde(numberOfPDE);
```

Geometry And Mesh

The geometry for a circle can easily be defined as shown below.

```
radius = 2;
g = decsg([1 0 0 radius]','C1',('C1')');
% Create a geometry object and append it to the PDE Model
geometryFromEdges(pdem,g);

% Plot the geometry and display the edge labels for use in the boundary
% condition definition.
figure;
pdegplot(pdem,'edgeLabels','on');
axis equal
title 'Geometry With Edge Labels Displayed';
```
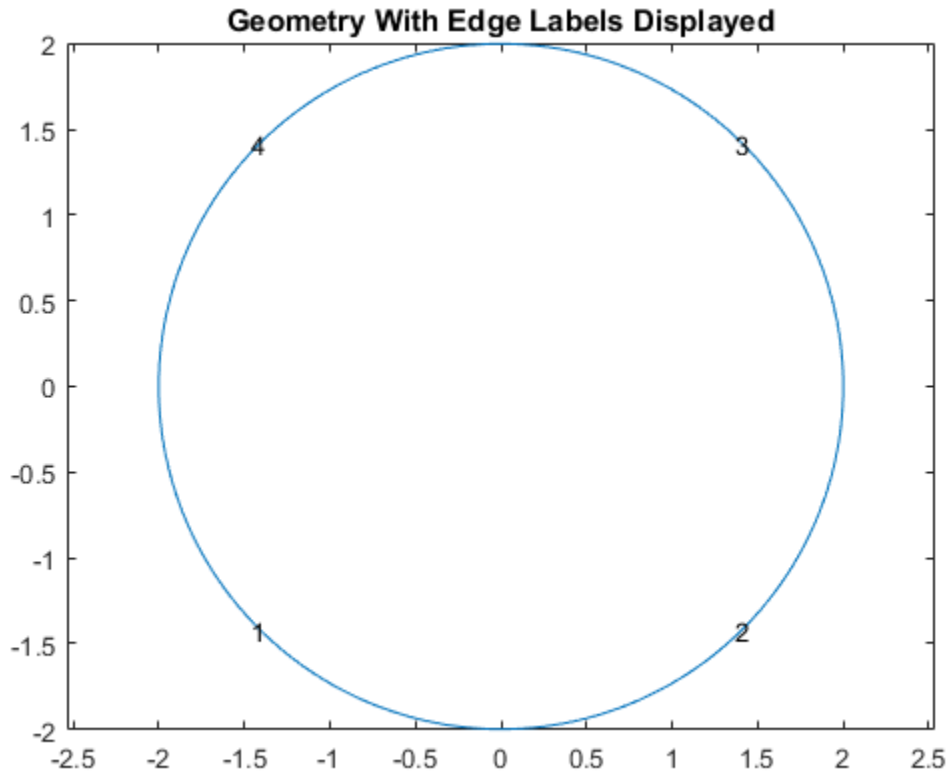
Define the PDE Coefficients and Boundary Conditions

```
c = 1e2;
a = 0;
f = 0;
d = 10;
specifyCoefficients(pdem,'m',0,'d',d,'c',c,'a',a,'f',f);
% Solution is zero at all four outer edges of the circle
bOuter = applyBoundaryCondition(pdem,'Edge',(1:4),'u',0);
```

Generate Mesh

```
generateMesh(pdem,'Hmax',0.2);
```

Solve the eigenvalue problem using `eigs`

Use `assembleFEMatrices` to calculate the global finite element mass and stiffness matrices with boundary conditions imposed using nullspace approach.

```
FEMatrices = assembleFEMatrices(pdem,'nullspace');
K = FEMatrices.Kc;
B = FEMatrices.B;
M = FEMatrices.M;
sigma = 1e2;
numberEigenvalues = 5;
[eigenvectorsEigs,eigenvaluesEigs] = eigs(K,M,numberEigenvalues,sigma);
% Reshape the diagonal eigenvaluesEigs matrix into a vector.
eigenvaluesEigs = diag(eigenvaluesEigs);
% Find the largest eigenvalue and its index in the eigenvalues vector.
[maxEigenvaluesEigs, maxIndex] = max(eigenvaluesEigs);
% Transform the eigenvectors with constrained equations removed to the full
% eigenvector including constrained equations.
eigenvectorsEigs = B*eigenvectorsEigs;
```

Solve the eigenvalue problem using `solvepdeeig`

```
%Set the range for |solvepdeeig| to be slightly larger than the range from
%|eigs|.
r = [min(eigenvaluesEigs)*.99 max(eigenvaluesEigs)*1.01];
result = solvepdeeig(pdem,r);
eigenvectorsPde = result.Eigenvectors;
eigenvaluesPde = result.Eigenvalues;
```

```
              Basis= 10,  Time=   0.02,  New conv eig=  0
              Basis= 19,  Time=   0.02,  New conv eig=  2
              Basis= 28,  Time=   0.02,  New conv eig=  7
End of sweep: Basis= 28,  Time=   0.02,  New conv eig=  7
              Basis= 17,  Time=   0.27,  New conv eig=  0
End of sweep: Basis= 17,  Time=   0.27,  New conv eig=  0
```

Compare the solutions computed by `eigs` and `solvepdeeig`

```
eigenValueDiff = sort(eigenvaluesPde) - sort(eigenvaluesEigs);
fprintf('Maximum difference in eigenvalues from solvepdeeig and eigs: %e\n', ...
  norm(eigenValueDiff,inf));
%
% As can be seen, both functions calculate the same eigenvalues. For any
% eigenvalue, the eigenvector can be multiplied by an arbitrary scalar.
```

```matlab
% eigs and solvepdeeigs choose a different arbitrary scalar for normalizing
% their eigenvectors as shown in the figure below.
%
h = figure;
h.Position = [1 1 2 1].*h.Position;
subplot(1,2,1);
axis equal
pdeplot(pdem,'xydata', eigenvectorsEigs(:,maxIndex),'contour','on');
title(sprintf('eigs eigenvector, eigenvalue: %12.4e', eigenvaluesEigs(maxIndex)));
xlabel('x');
ylabel('y');
subplot(1,2,2);
axis equal
pdeplot(pdem,'xydata',eigenvectorsPde(:,end),'contour','on');
title(sprintf('solvepdeeig eigenvector, eigenvalue: %12.4e',eigenvaluesPde(end)));
xlabel('x');
ylabel('y');
```

Maximum difference in eigenvalues from solvepdeeig and eigs: 1.421085e-13
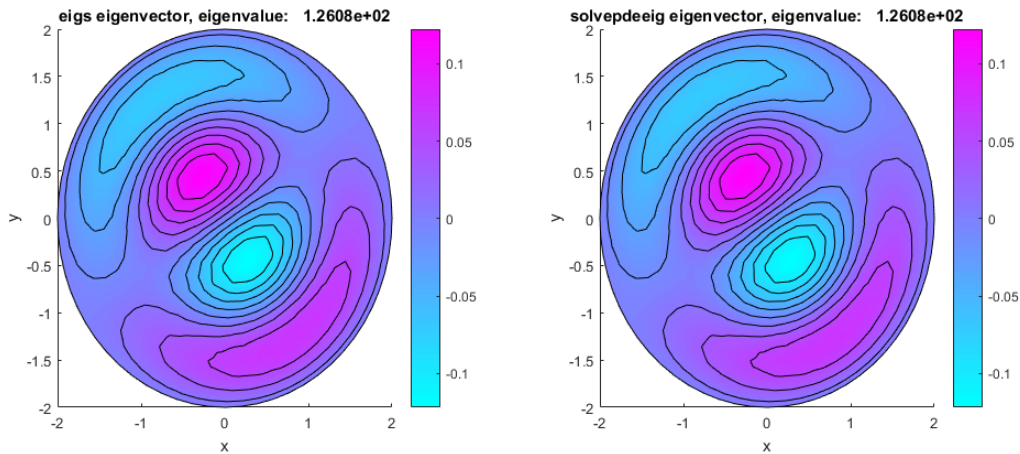
Solve PDEs Programmatically

> **Note:** **THIS PAGE DESCRIBES AN ALTERNATIVE LEGACY WORKFLOW APPLICABLE ONLY FOR 2-D PROBLEMS.** For the recommended workflow, see "Solve Problems Using PDEModel Objects" on page 2-14. The workflow described on this page is an alternative to the legacy workflow described in "Solve Problems Using Legacy PDEModel Objects" on page 2-11.

When You Need Programmatic Solutions

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

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

Data Structures in Partial Differential Equation Toolbox

The process of defining your problem and solving it is reflected in the design of the PDE app. 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.
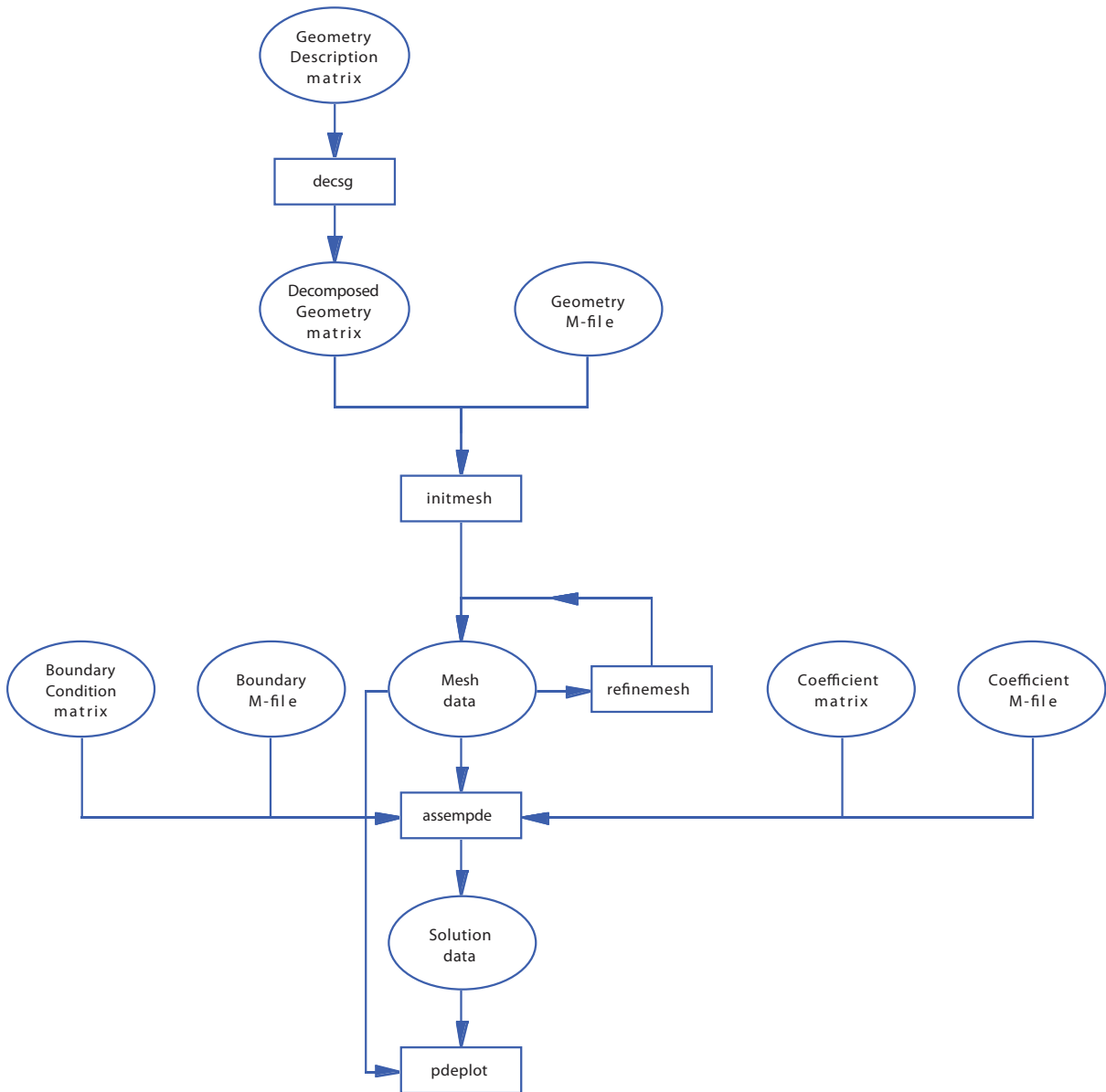
Constructive Solid Geometry Model

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 PDE app 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 PDE app 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 reference page for `decsg` and "2-D Geometry".

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 PDE app 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 reference pages for `assemb` and see "Boundary Conditions".

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 PDE app 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`, and see "PDE Coefficients".

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` and see "Mesh Data" on page 2-211.

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. Use `pdeInterpolant` and `evaluate` for more general interpolation. `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 PDE app 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 Poisson's Equation with Point Source and Adaptive Mesh Refinement 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 Poisson's Equation with Point Source and Adaptive Mesh Refinement. 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 $-\Delta u = f$ in Ω

with Dirichlet boundary conditions, where Ω is a rectangle.

For the fast solution algorithms to work, the mesh on the rectangle must be a *regular mesh*. In this context it means that the first side of the rectangle is divided into N_1 segments of length h_1, the second into N_2 segments of length h_2, and ($N_1 + 1$) by ($N_2 + 1$) points are introduced on the regular grid thus defined. The triangles are all congruent with sides h_1, h_2 and a right angle in between.

The Dirichlet boundary conditions are eliminated in the usual way, and the resulting problem for the interior nodes is $Kv = F$. If the interior nodes are numbered from left to right, and then from bottom to top, the K matrix is block tridiagonal. The $N_2 - 1$ diagonal blocks, here called T, are themselves tridiagonal ($N_1 - 1$) by ($N_1 - 1$) matrices, with $2(h_1/h_2 + h_2/h_1)$ on the diagonal and $-h_2/h_1$ on the subdiagonals. The subdiagonal blocks, here called I, are $-h_1/h_2$ times the unit $N_1 - 1$ matrix.

The key to the solution of the problem $Kv = F$ is that the problem $Tw = f$ is possible to solve using the *discrete sine transform*. Let S be the ($N_1 - 1$) by ($N_1 - 1$) matrix with $S_{ij} = \sin(\pi ij/N_1)$. Then $S^{-1}TS = \Lambda$, where Λ is a diagonal matrix with diagonal entries $2(h_1/h_2 + h_2/h_1) - 2h_2/h_1 \cos(\pi i/N_1)$. $w = S\Lambda^{-1}S^{-1} f$, but multiplying with S is nothing more than taking the discrete sine transform, and multiplying with S^{-1} is the same as taking the inverse discrete sine transform. The discrete sine transform can be efficiently calculated using the fast Fourier transform on a sequence of length $2N_1$.

Solving $Tw = f$ using the discrete sine transform would not be an advantage in itself, since the system is tridiagonal and should be solved as such. However, for the full system $Ky = F$, 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.

Plot 2-D Solutions and Their Gradients

Note: PLOTTING SOLUTIONS AND GRADIENTS IS THE SAME FOR THE
RECOMMENDED AND THE LEGACY WORKFLOWS.

Plot Solutions Without Explicit Interpolation

To quickly visualize a 2-D scalar PDE solution, use the `pdeplot` function. This function
lets you plot the solution without explicitly interpolating the solution. For example,
solve the scalar elliptic problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet
boundary conditions and plot the solution.

Create the PDE model, 2-D geometry, and mesh. Specify boundary conditions and
coefficients. Solve the PDE problem.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
c = 1;
a = 0;
f = 1;
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);
generateMesh(model);

results = solvepde(model);
```

Use `pdeplot` to plot the solution.

```
u = results.NodalSolution;
pdeplot(model,'xydata',u,'zdata',u,'mesh','on')
xlabel('x')
ylabel('y')
```
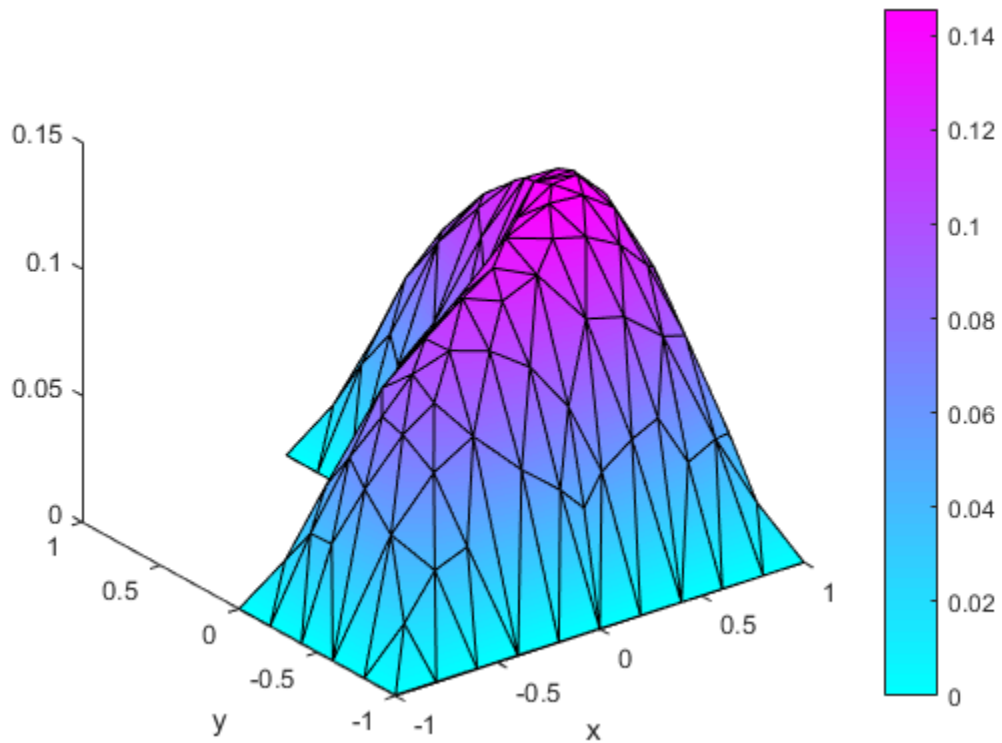
To get a smoother solution surface, specify the maximum size of the mesh triangles by using the `Hmax` argument. Then solve the PDE problem using this new mesh, and plot the solution again.

```
generateMesh(model,'Hmax',0.05);
results = solvepde(model);
u = results.NodalSolution;

pdeplot(model,'xydata',u,'zdata',u,'mesh','on')
xlabel('x')
ylabel('y')
```
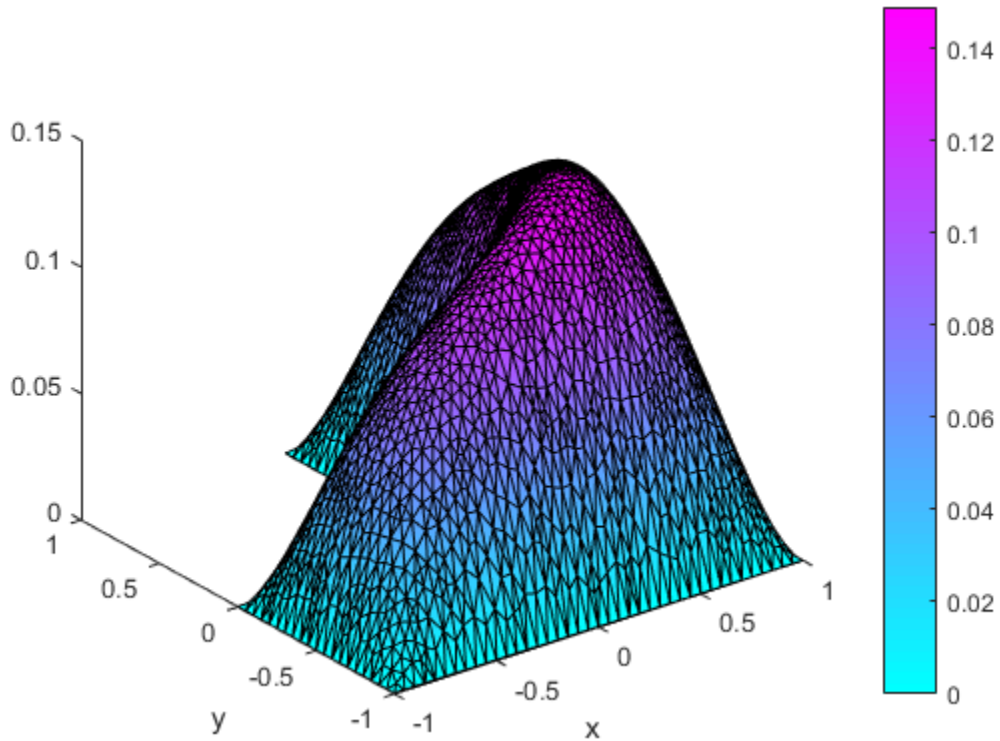
Interpolate and Plot Solutions and Gradients

Alternatively, you can interpolate the solution and, if needed, its gradient in separate steps, and then plot the results by using MATLAB functions, such as surf, mesh, quiver, and so on. For example, solve the same scalar elliptic problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions. Interpolate the solution and its gradient, and then plot the results.

Create the PDE model, 2-D geometry, and mesh. Specify boundary conditions and coefficients. Solve the PDE problem.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
c = 1;
a = 0;
```

```
f = 1;
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);
generateMesh(model,'Hmax',0.05);
results = solvepde(model);
```

Interpolate the solution and its gradients to a dense grid from −1 to 1 in each direction.

```
v = linspace(-1,1,101);
[X,Y] = meshgrid(v);
querypoints = [X(:),Y(:)]';
uintrp = interpolateSolution(results,querypoints);
```

Plot the resulting solution on a mesh.

```
uintrp = reshape(uintrp,size(X));
mesh(X,Y,uintrp)
xlabel('x')
ylabel('y')
```
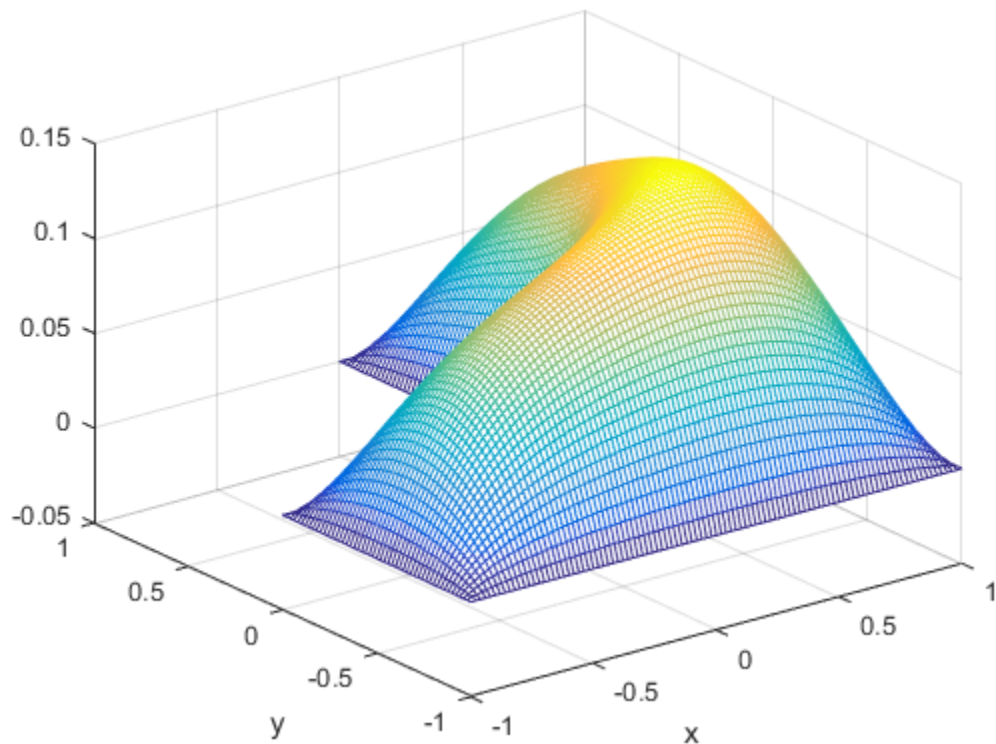
Interpolate gradients of the solution to the grid from -1 to 1 in each direction. Plot the result using `quiver`.

```
[gradx,grady] = evaluateGradient(results,querypoints);
figure
quiver(X(:),Y(:),gradx,grady)
xlabel('x')
ylabel('y')
```
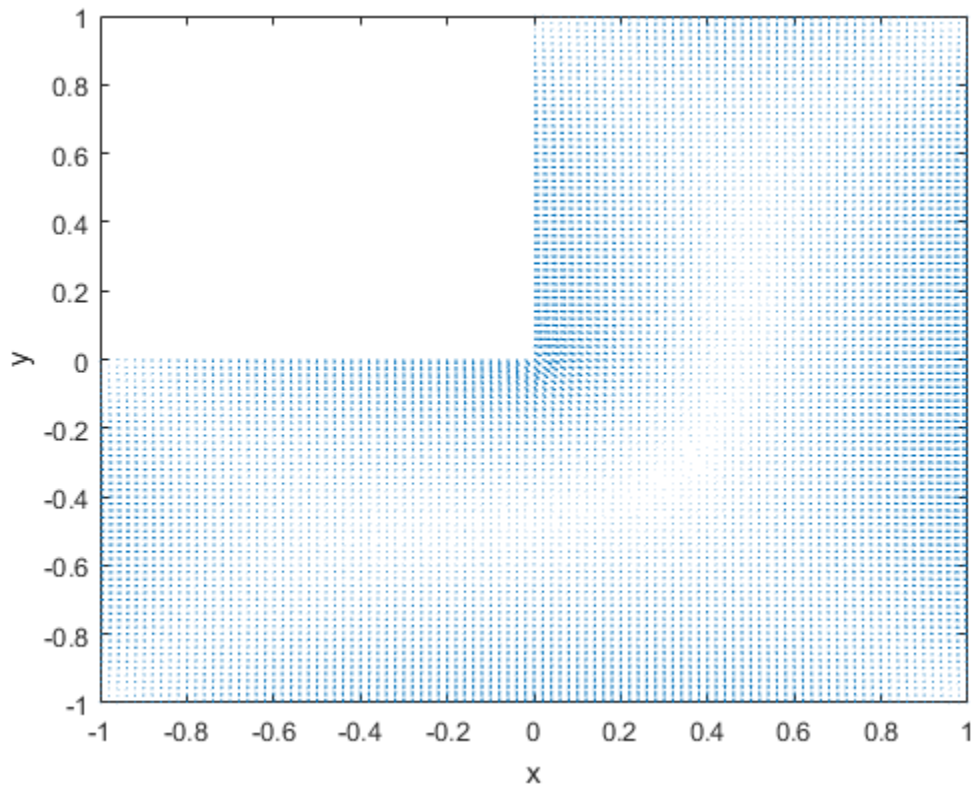

Zoom in to see more details. For example, restrict the range to [−0.2,0.2] in each direction.

```
axis([-0.2 0.2 -0.2 0.2])
```
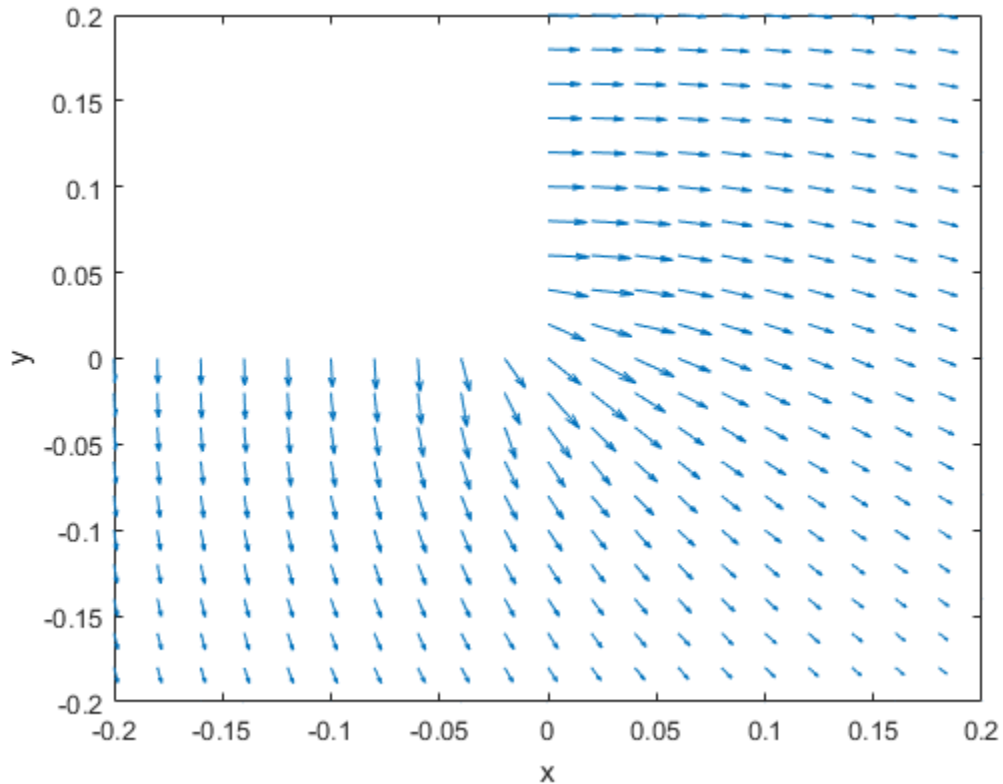
Plot the solution and the gradients on the same range.

```
figure
h1 = meshc(X,Y,uintrp);
set(h1,'FaceColor','g','EdgeColor','b')
xlabel('x')
ylabel('y')
alpha(0.5)
hold on

Z = -0.05*ones(size(X));
gradz = zeros(size(gradx));

h2 = quiver3(X(:),Y(:),Z(:),gradx,grady,gradz);
set(h2,'Color','r')
axis([-0.2,0.2,-0.2,0.2])
```
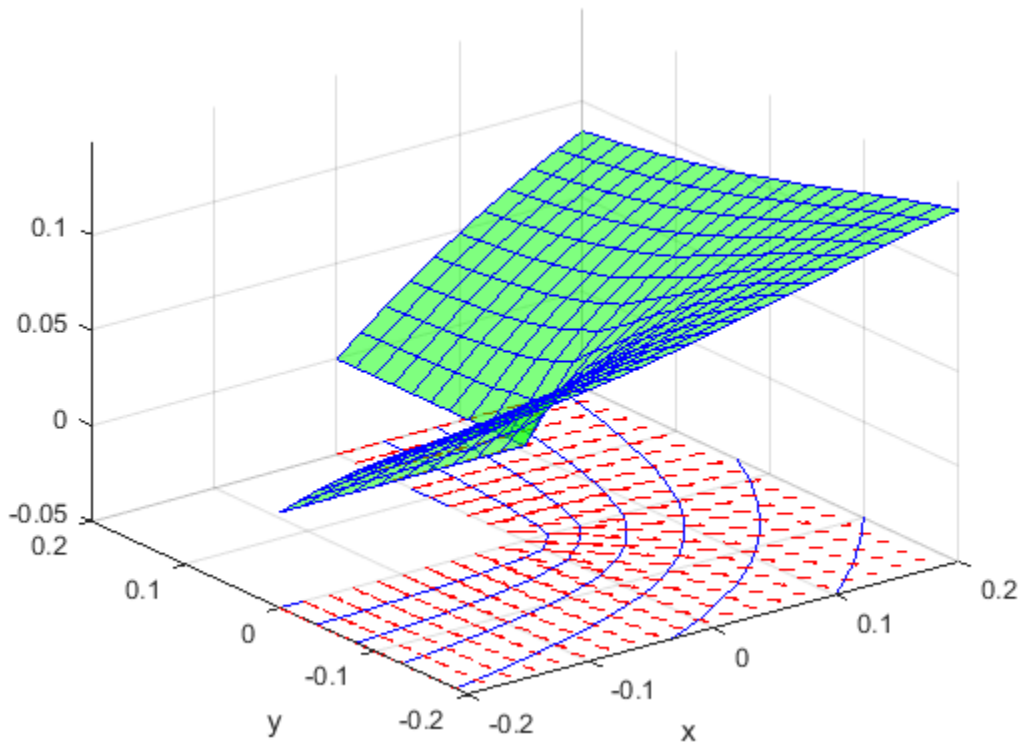
Slice of the solution plot along the line $x = y$.

```
figure
mesh(X,Y,uintrp)
xlabel('x')
ylabel('y')
alpha(0.25)
hold on

z = linspace(0,0.15,101);
Z = meshgrid(z);
surf(X,X,Z')

view([-20 -45 15])
colormap winter
```
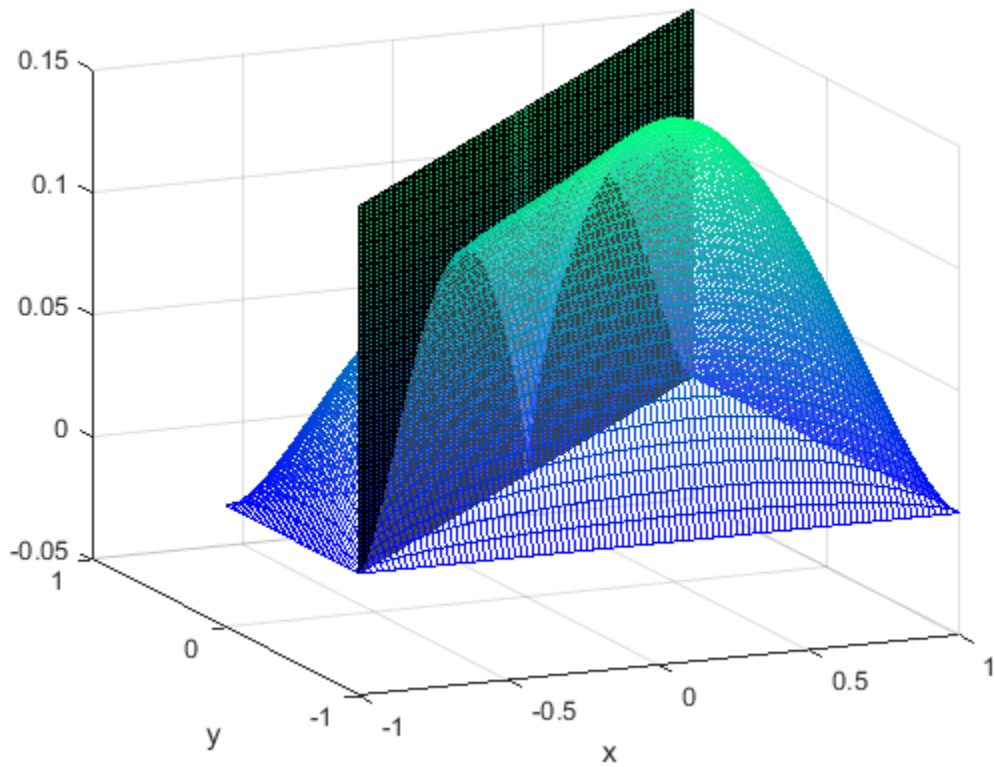
Plot the interpolated solution along the line.

```
figure
xq = v;
yq = v;
uintrp = interpolateSolution(results,xq,yq);

plot3(xq,yq,uintrp)
grid on
xlabel('x')
ylabel('y')
```
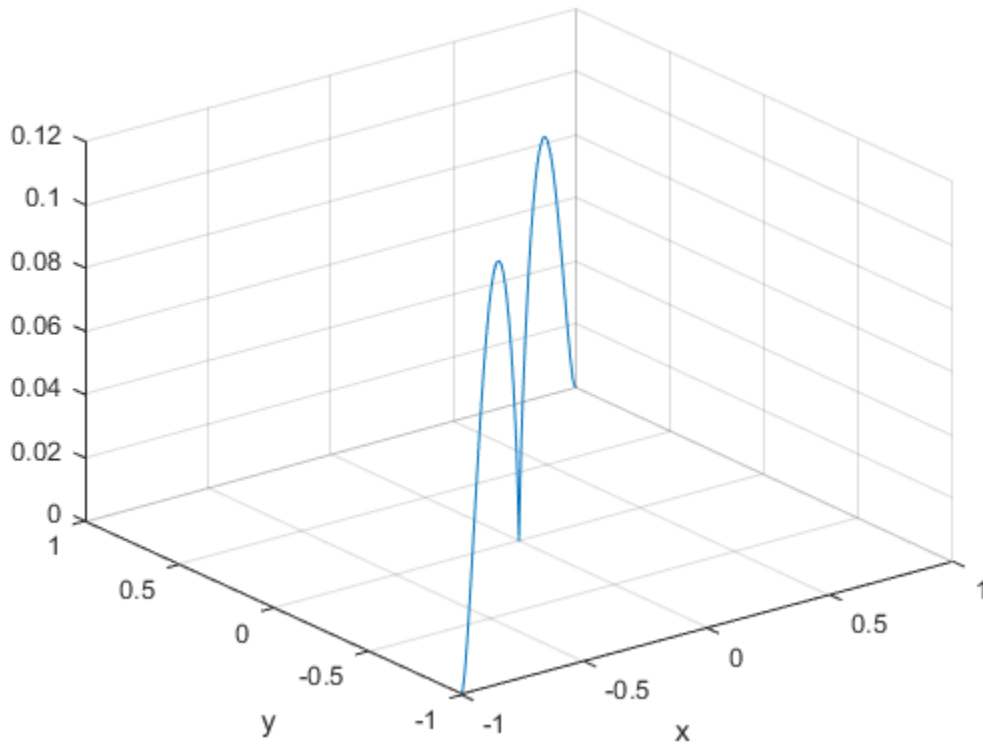
Interpolate gradients of the solution along the same line and add them to the solution plot.

```
[gradx,grady] = evaluateGradient(results,xq,yq);

gradx = reshape(gradx,size(xq));
grady = reshape(grady,size(yq));

hold on
quiver(xq,yq,gradx,grady)
view([-20 -45 75])
```
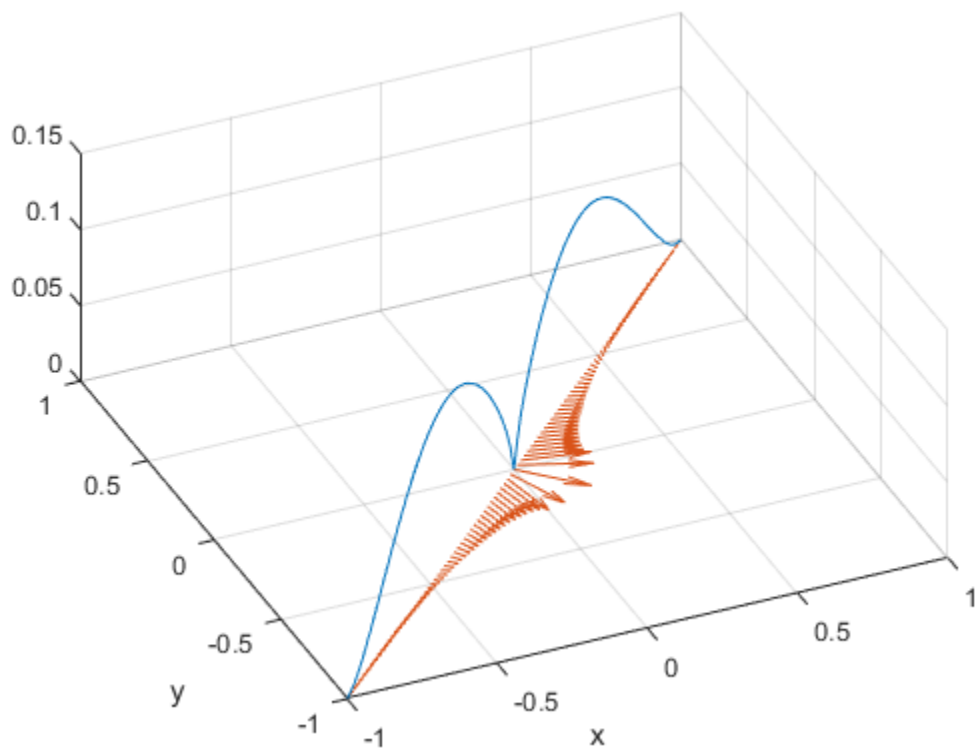
Plot 3-D Solutions and Their Gradients

Note: PLOTTING SOLUTIONS AND GRADIENTS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Types of 3-D Solution Plots

There are several types of plots for solutions when you have 3-D geometry.

- Surface plot — Sometimes you want to examine the solution on the surface of the geometry. For example, in a stress or strain calculation, the most interesting data can appear on the geometry surface. For an example, see "Surface Plot" on page 3-145.

  For a colored surface plot of a scalar solution, set the `pdeplot3D colormapdata` to the solution `u`:

  ```
  pdeplot3D(model,'colormapdata',u);
  ```

- Plot on a 2-D slice — To examine the solution on the interior of the geometry, define a 2-D grid that intersects the geometry, and interpolate the solution onto the grid. For examples, see "2-D Slices Through 3-D Geometry" on page 3-149 and "Contour Slices Through a 3-D Solution" on page 3-154. While these two examples show planar grid slices, you can also slice on a curved grid.

- Streamline or quiver plots — Plot the gradient of the solution as streamlines or a quiver. See "Plots of Gradients and Streamlines" on page 3-161.

- You can use any MATLAB plotting command to create 3-D plots. See "Techniques for Visualizing Scalar Volume Data", "Visualizing Vector Volume Data", and "Graphics".

For other plot types, see the `pdeplot3D` reference page.

Surface Plot

This example shows how to obtain a surface plot of a solution with 3-D geometry and $N > 1$.

Import a tetrahedral geometry to a model with $N = 2$ equations and view its faces.

```
model = createpde(2);
```

```
importGeometry(model,'Tetrahedron.stl');
hc = pdegplot(model,'FaceLabels','on');
hc(1).FaceAlpha = 0.5;
view(-40,24)
```
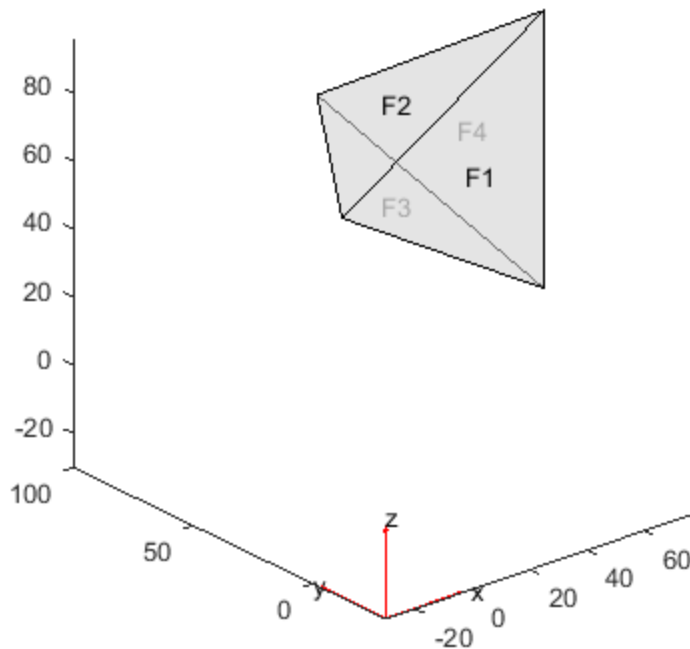

Create a problem with zero Dirichlet boundary conditions on face 4.

```
applyBoundaryCondition(model,'face',4,'u',[0,0]);
```

Create coefficients for the problem, where f = [1;10] and c is a symmetric matrix in $6N$ form.

```
f = [1;10];
```

```
a = 0;
c = [2;0;4;1;3;8;1;0;2;1;2;4];
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);
```

Create a mesh for the solution.

```
generateMesh(model);
```

Solve the problem.

```
results = solvepde(model);
u = results.NodalSolution;
```

Plot the two components of the solution.

```
pdeplot3D(model,'colormapdata',u(:,1));
view(-175,4)
title('u(1)')
figure
pdeplot3D(model,'colormapdata',u(:,2));
view(-175,4)
title('u(2)')
```

u(1)

u(2)
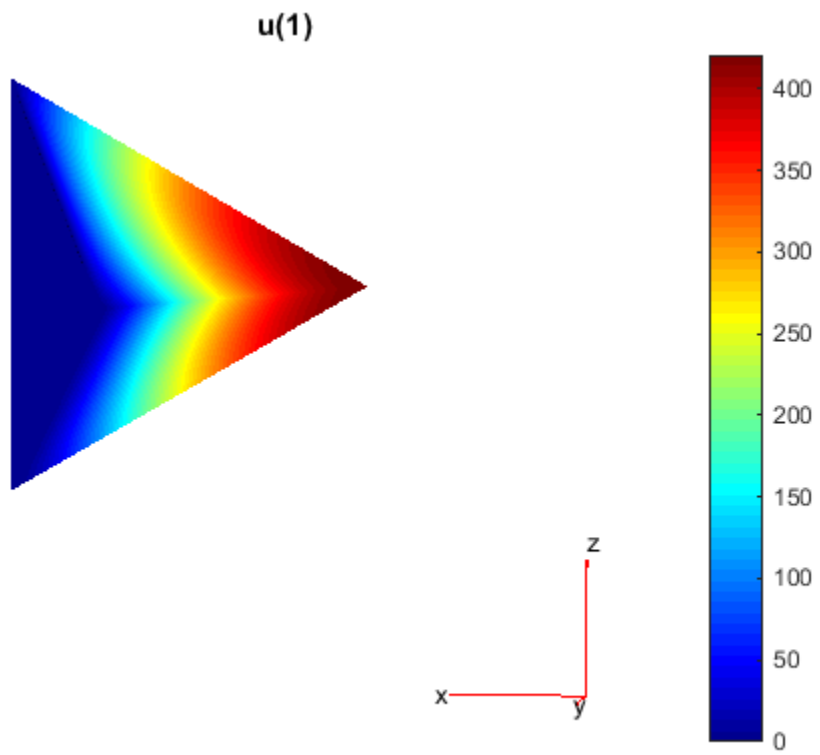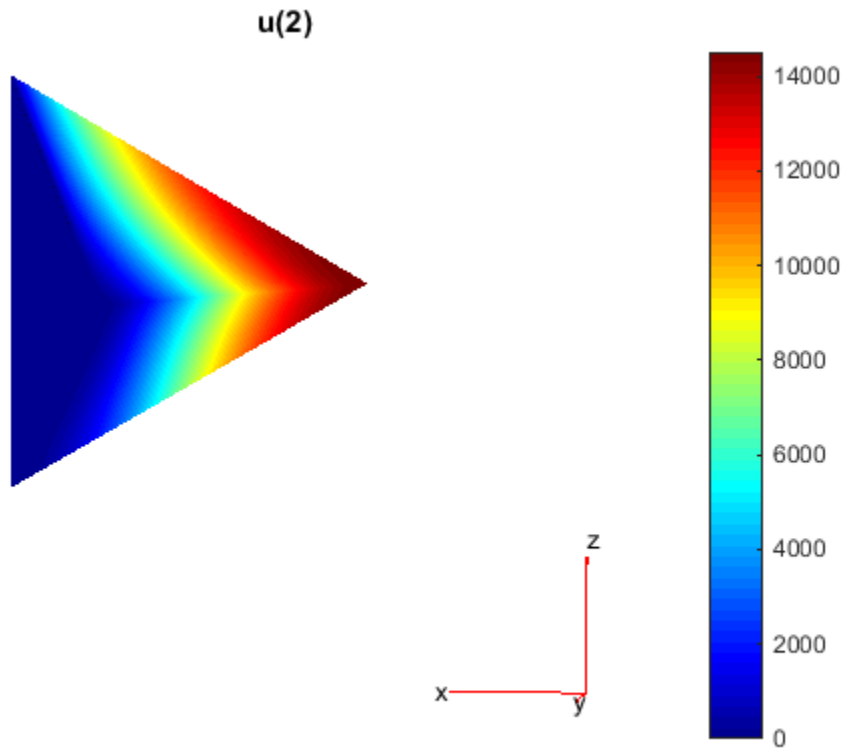
2-D Slices Through 3-D Geometry

This example shows how to obtain plots from 2-D slices through a 3-D geometry.

The problem is

$$\frac{\partial u}{\partial t} - \Delta u = f$$

on a 3-D slab with dimensions 10-by-10-by-1, where $u = 0$ at time $t = 0$, boundary conditions are Dirichlet, and

$$f(x, y, z) = 1 + y + 10z^2.$$

Set Up and Solve the PDE

Define a function for the nonlinear f coefficient in the syntax as given in "f Coefficient for `specifyCoefficients`" on page 2-101.

```
function bcMatrix = myfffun(region,state)

bcMatrix = 1+10*region.z.^2+region.y;
```

Import the geometry and examine the face labels.

```
model = createpde;
g = importGeometry(model,'Plate10x10x1.stl');
hc = pdegplot(g,'FaceLabels','on');
hc(1).FaceAlpha = 0.5;
```
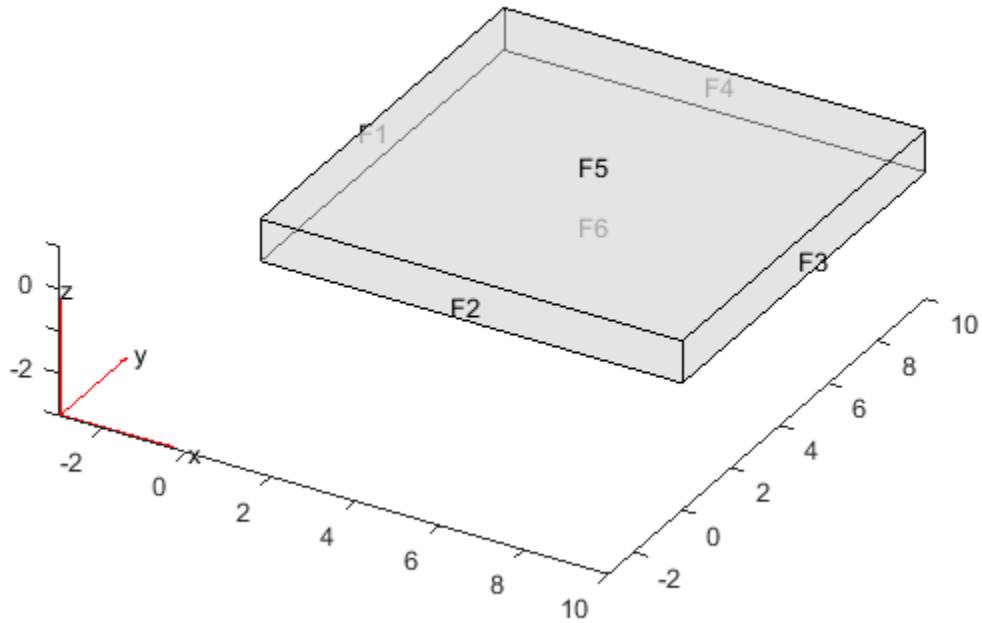
The faces are numbered 1 through 6.

Create the coefficients and boundary conditions.

```
c = 1;
a = 0;
d = 1;
f = @myfffun;
specifyCoefficients(model,'m',0,'d',d,'c',c,'a',a,'f',f);

applyBoundaryCondition(model,'face',1:6,'u',0);
```

Set a 0 initial condition.

```
setInitialConditions(model,0);
```

3-151

Create a mesh with sides no longer than 0.3.

```
generateMesh(model,'Hmax',0.3);
```

Set times from 0 through 0.2 and solve the PDE.

```
tlist = 0:0.02:0.2;
results = solvepde(model,tlist);
```

Plot Slices Through the Solution

Create a grid of (x, y, z) points, where $x = 5$, y ranges from 0 through 10, and z ranges from 0 through 1. Interpolate the solution to these grid points and all times.

```
yy = 0:0.5:10;
zz = 0:0.25:1;
[YY,ZZ] = meshgrid(yy,zz);
XX = 5*ones(size(YY));
uintrp = interpolateSolution(results,XX,YY,ZZ,1:length(tlist));
```

The solution matrix `uintrp` has 11 columns, one for each time in `tlist`. Take the interpolated solution for the second column, which corresponds to time `0.02`.

```
usol = uintrp(:,2);
```

The elements of `usol` come from interpolating the solution to the XX, YY, and ZZ matrices, which are each 5-by-21, corresponding to z-by-y variables. Reshape `usol` to the same 5-by-21 size, and make a surface plot of the solution. Also make surface plots corresponding to times 0.06, 0.10, and 0.20.

```
figure
usol = reshape(usol,size(XX));
subplot(2,2,1)
surf(usol)
title('t = 0.02')
zlim([0,1.5])
xlim([1,21])
ylim([1,5])

usol = uintrp(:,4);
usol = reshape(usol,size(XX));
subplot(2,2,2)
surf(usol)
title('t = 0.06')
zlim([0,1.5])
xlim([1,21])
```

```
ylim([1,5])

usol = uintrp(:,6);
usol = reshape(usol,size(XX));
subplot(2,2,3)
surf(usol)
title('t = 0.10')
zlim([0,1.5])
xlim([1,21])
ylim([1,5])

usol = uintrp(:,11);
usol = reshape(usol,size(XX));
subplot(2,2,4)
surf(usol)
title('t = 0.20')
zlim([0,1.5])
xlim([1,21])
ylim([1,5])
```
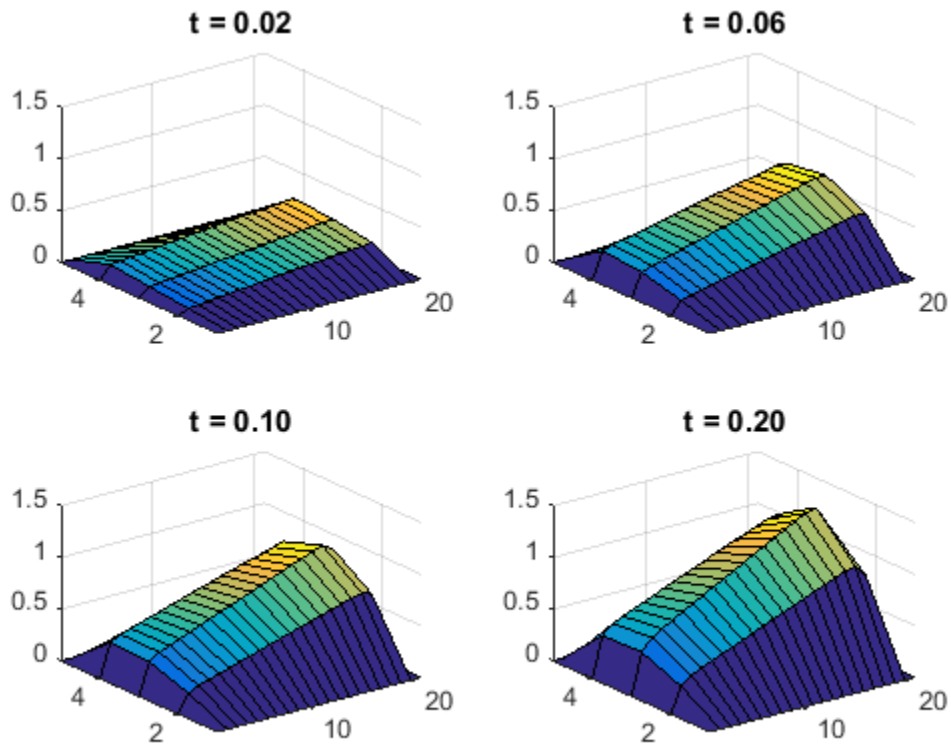
Contour Slices Through a 3-D Solution

This example shows how to create contour slices in various directions through a solution in 3-D geometry.

Set Up and Solve the PDE

The problem is to solve Poisson's equation with zero Dirichlet boundary conditions for a complicated geometry. Poisson's equation is

$$-\nabla \cdot \nabla u = f.$$

Partial Differential Equation Toolbox™ solves equations in the form

$$-\nabla \cdot \nabla(cu) + au = f.$$

So you can represent the problem by setting $c = 1$ and $a = 0$. Arbitrarily set $f = 10$.

```
c = 1;
a = 0;
f = 10;
```

The first step in solving any 3-D PDE problem is to create a PDE Model. This is a container that holds the number of equations, geometry, mesh, and boundary conditions for your PDE. Create the model, then import the 'ForearmLink.stl' file and view the geometry.

```
N = 1;
model = createpde(N);
importGeometry(model,'ForearmLink.stl');
h = pdegplot(model);
h(1).FaceAlpha = 0.5;
view(-42,24)
```
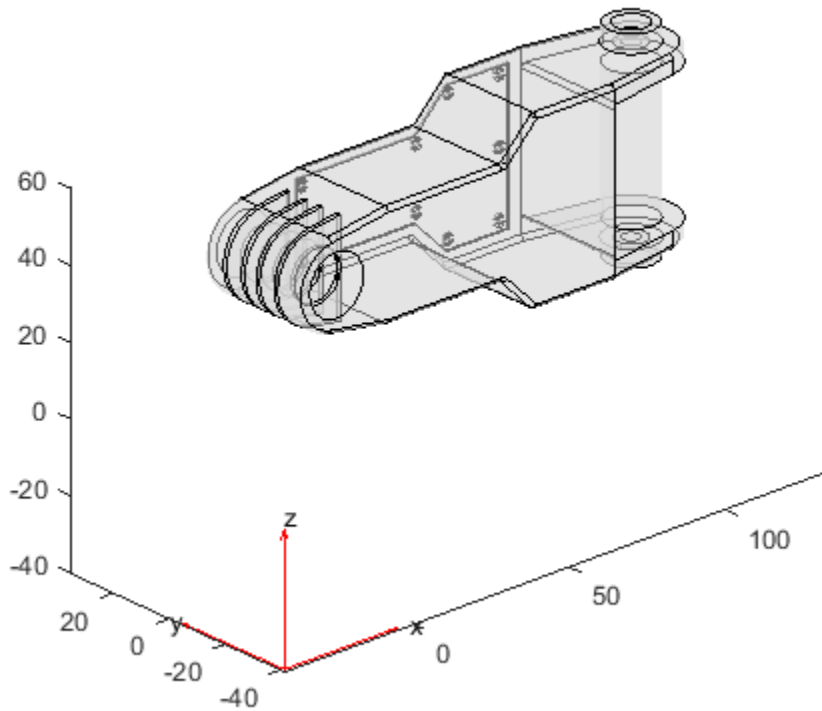
Specify PDE Coefficients

Include the PDE COefficients in `model`.

```
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);
```

Create zero Dirichlet boundary conditions on all faces.

```
applyBoundaryCondition(model,'Face',1:model.Geometry.NumFaces,'u',0);
```

Create a mesh and solve the PDE.

```
generateMesh(model);
result = solvepde(model);
```

Plot the Solution as Contour Slices

Because the boundary conditions are $u = 0$ on all faces, the solution u is nonzero only in the interior. To examine the interior, take a rectangular grid that covers the geometry with a spacing of one unit in each coordinate direction.

```
[X,Y,Z] = meshgrid(0:135,0:35,0:61);
```

For plotting and analysis, create a `PDEResults` object from the solution. Interpolate the result at every grid point.

```
V = interpolateSolution(result,X,Y,Z);
V = reshape(V,size(X));
```

Plot contour slices for various values of z.

```
figure
colormap jet
contourslice(X,Y,Z,V,[],[],0:5:60)
xlabel('x')
ylabel('y')
zlabel('z')
colorbar
view(-11,14)
axis equal
```
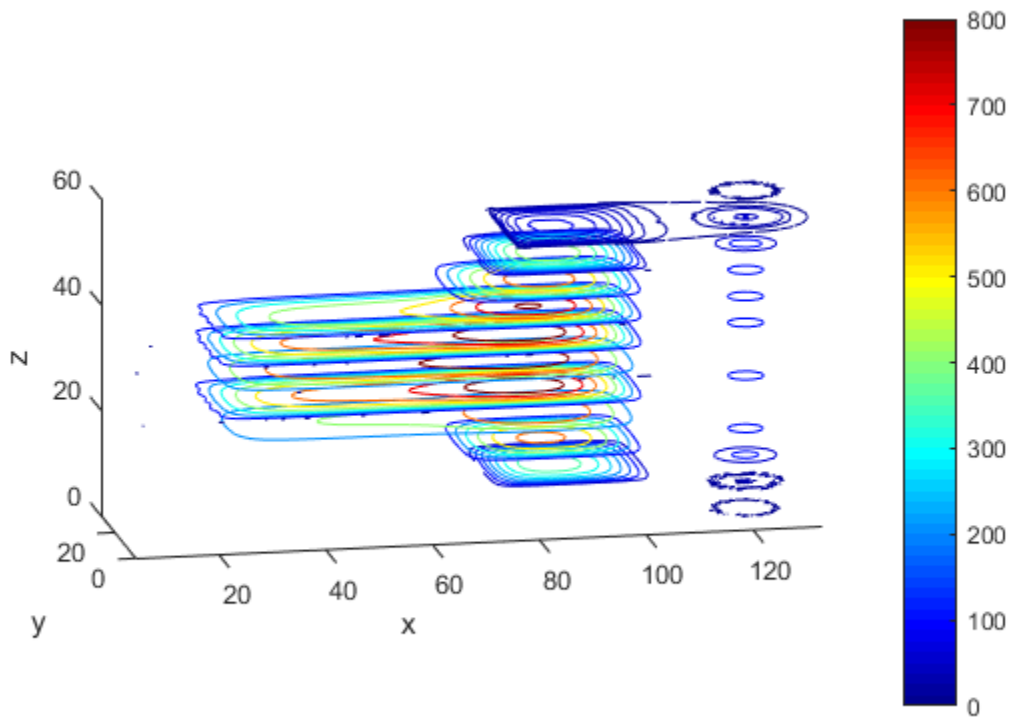
Plot contour slices for various values of y.

```
figure
colormap jet
contourslice(X,Y,Z,V,[],1:6:31,[])
xlabel('x')
ylabel('y')
zlabel('z')
colorbar
view(-62,34)
axis equal
```
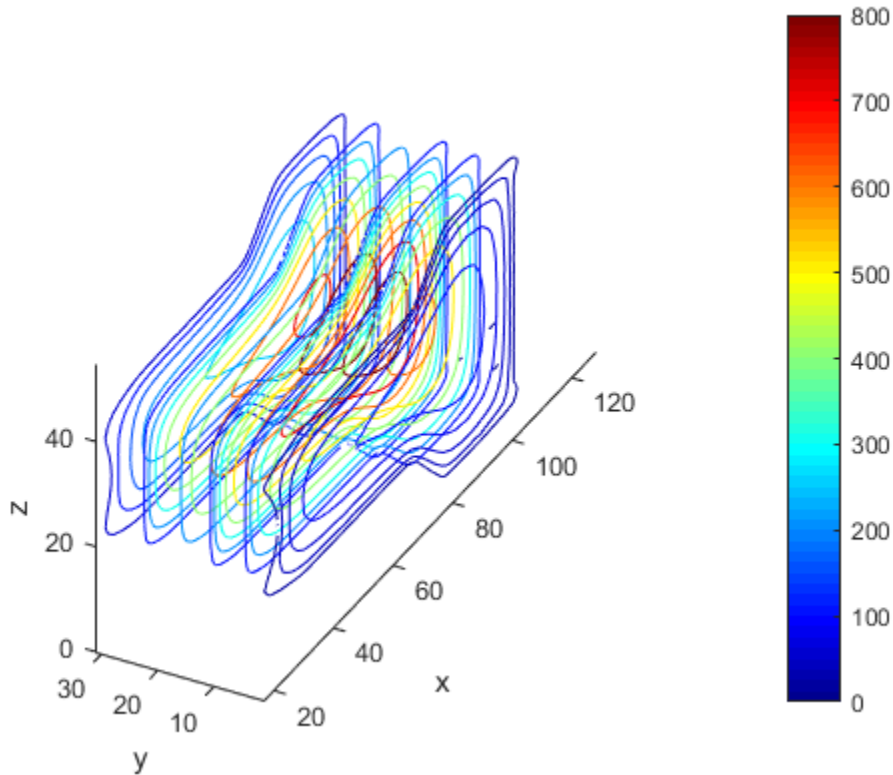
Save Memory by Evaluating As Needed

For large problems you can run out of memory when creating a fine 3-D grid. Furthermore, it can be time-consuming to evaluate the solution on a full grid. To save memory and time, evaluate only at the points you plot. You can also use this technique to interpolate to tilted grids, or to other surfaces.

For example, interpolate the solution to a grid on the tilted plane $0 \leq x \leq 135$, $0 \leq y \leq 35$, and $z = x/10 + y/2$. Plot both contours and colored surface data. Use a fine grid, with spacing 0.2.

```
[X,Y] = meshgrid(0:0.2:135,0:0.2:35);
Z = X/10 + Y/2;
V = interpolateSolution(result,X,Y,Z);
```

```
V = reshape(V,size(X));
figure
subplot(2,1,1)
contour(X,Y,V);
axis equal
title('Contour Plot on Tilted Plane')
xlabel('x')
ylabel('y')
colorbar
subplot(2,1,2)
surf(X,Y,V,'LineStyle','none');
axis equal
view(0,90)
title('Colored Plot on Tilted Plane')
xlabel('x')
ylabel('y')
colorbar
```

Contour Plot on Tilted Plane

Colored Plot on Tilted Plane
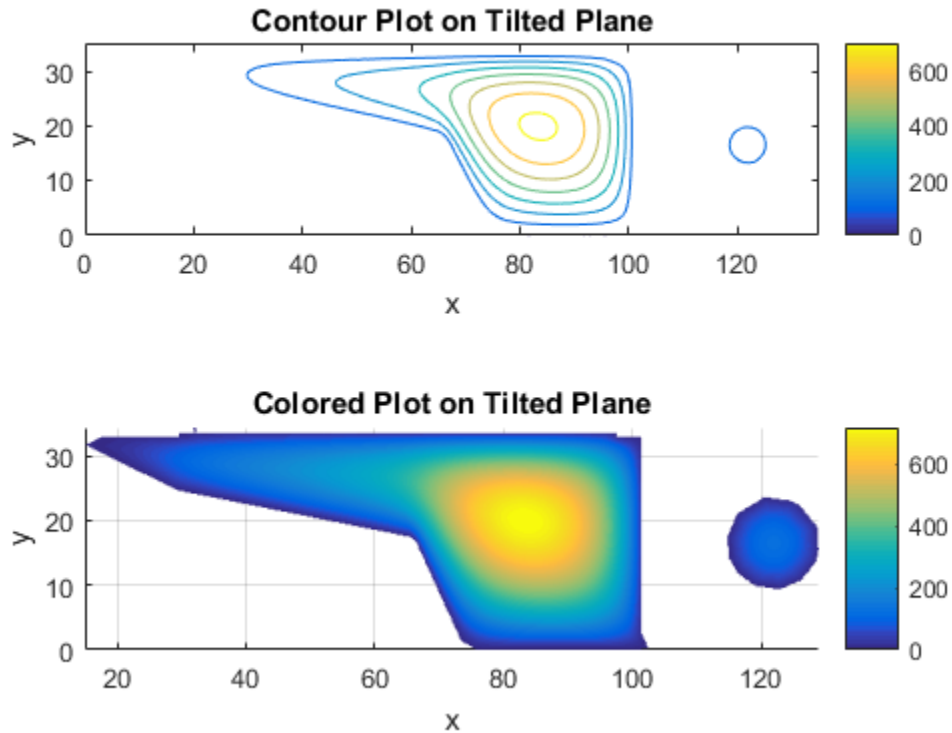
Plots of Gradients and Streamlines

This example shows how to calculate the approximate gradients of a solution, and how to use those gradients in a quiver plot or streamline plot.

The problem is the calculation of the mean exit time of a Brownian particle from a region that contains absorbing (escape) boundaries and reflecting boundaries. For more information, see Narrow escape problem. The PDE is Poisson's equation with constant coefficients. The geometry is a simple rectangular solid. The solution $u(x,y,z)$ represents the mean time it takes a particle starting at position (x,y,z) to exit the region.

Import and View the Geometry

```
model = createpde;
importGeometry(model,'Block.stl');
handl = pdegplot(model,'FaceLabels','on');
view(-42,24)
handl(1).FaceAlpha = 0.5;
```
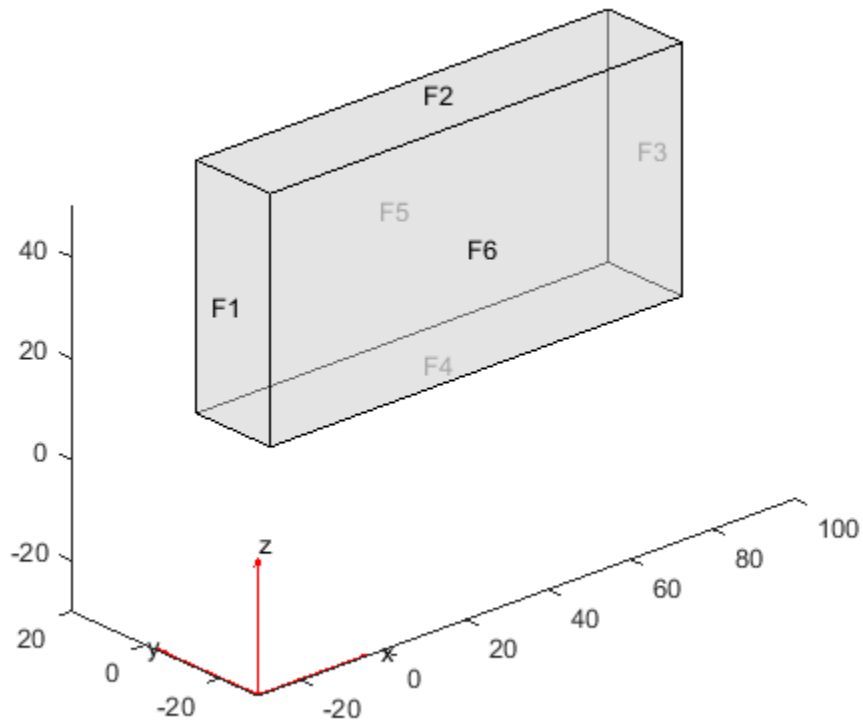

Set Boundary Conditions

Set faces 1, 2, and 5 to be the places where the particle can escape. On these faces, the solution $u = 0$. Keep the default reflecting boundary conditions on faces 3, 4, and 6.

```
applyBoundaryCondition(model,'face',[1,2,5],'u',0);
```

Create PDE Coefficients

The PDE is

$$-\Delta u = -\nabla\cdot\nabla u = 2.$$

In Partial Differential Equation Toolbox syntax,

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

This equation translates to coefficients $c = 1$, $a = 0$, and $f = 2$. Enter the coefficients.

```
c = 1;
a = 0;
f = 2;
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);
```

Create Mesh and Solve PDE

Initialize the mesh.

```
generateMesh(model);
```

Solve the PDE.

```
results = solvepde(model);
```

Examine the Solution in a Contour Slice Plot

Create a grid and interpolate the solution to the grid.

```
[X,Y,Z] = meshgrid(0:135,0:35,0:61);
V = interpolateSolution(results,X,Y,Z);
V = reshape(V,size(X));
```

Create a contour slice plot for five fixed values of the *y* coordinate.

```
figure
colormap jet
contourslice(X,Y,Z,V,[],0:4:16,[])
xlabel('x');
ylabel('y');
zlabel('z')
xlim([0,100])
ylim([0,20])
zlim([0,50])
axis equal
```

```
view(-50,22)
colorbar
```
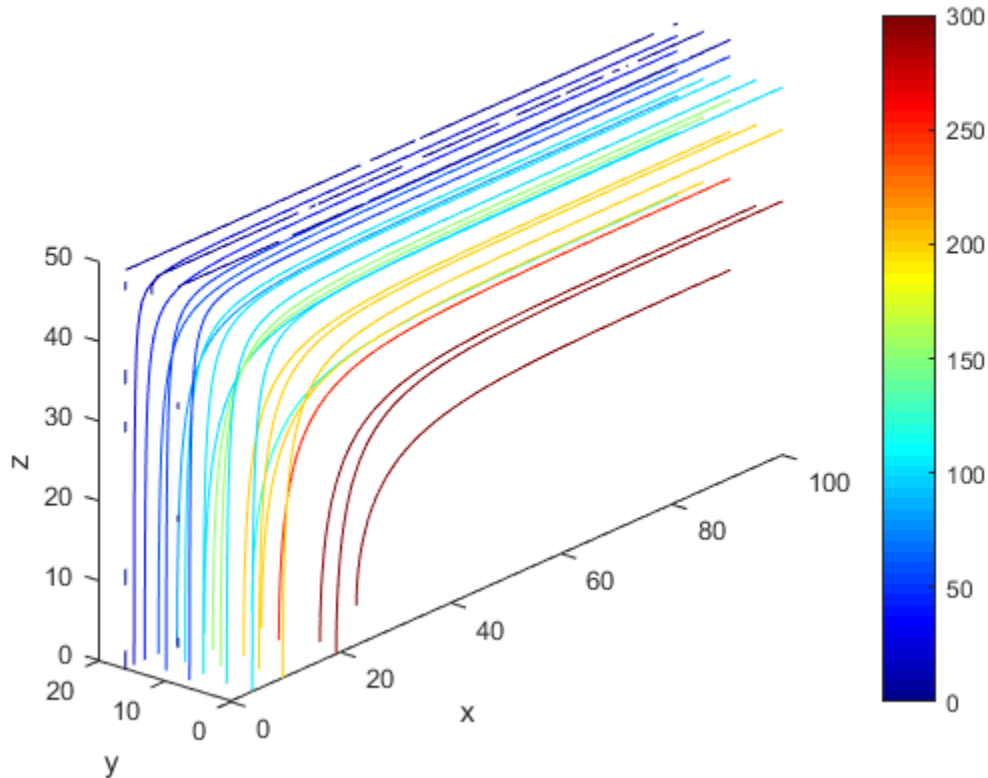

The particle has the largest mean exit time near the point $(x,y,z) = (100,0,0)$.

Use Gradients for Quiver and Streamline Plots

Examine the solution in more detail by evaluating the gradient of the solution. Use a rather coarse mesh so that you can see the details on the quiver and streamline plots.

```
[X,Y,Z] = meshgrid(1:9:99,1:3:20,1:6:50);
[gradx,grady,gradz] = evaluateGradient(results,X,Y,Z);
```

Plot the gradient vectors. First reshape the approximate gradients to the shape of the mesh.

```
gradx = reshape(gradx,size(X));
grady = reshape(grady,size(Y));
gradz = reshape(gradz,size(Z));

figure
quiver3(X,Y,Z,gradx,grady,gradz)
axis equal
xlabel 'x'
ylabel 'y'
zlabel 'z'
title('Quiver Plot of Estimated Gradient of Solution')
```


Quiver Plot of Estimated Gradient of Solution
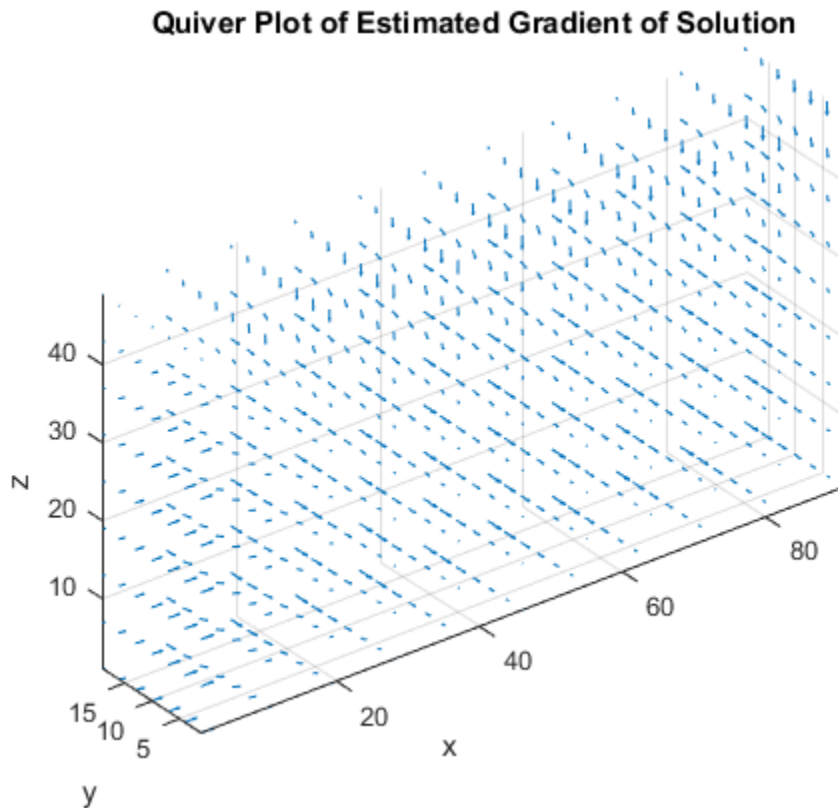
Plot the streamlines of the approximate gradient. Start the streamlines from a sparser set of initial points.

```
hold on
[sx,sy,sz] = meshgrid([1,46],1:6:20,1:12:50);
streamline(X,Y,Z,gradx,grady,gradz,sx,sy,sz)
title('Quiver Plot with Streamlines')
hold off
```
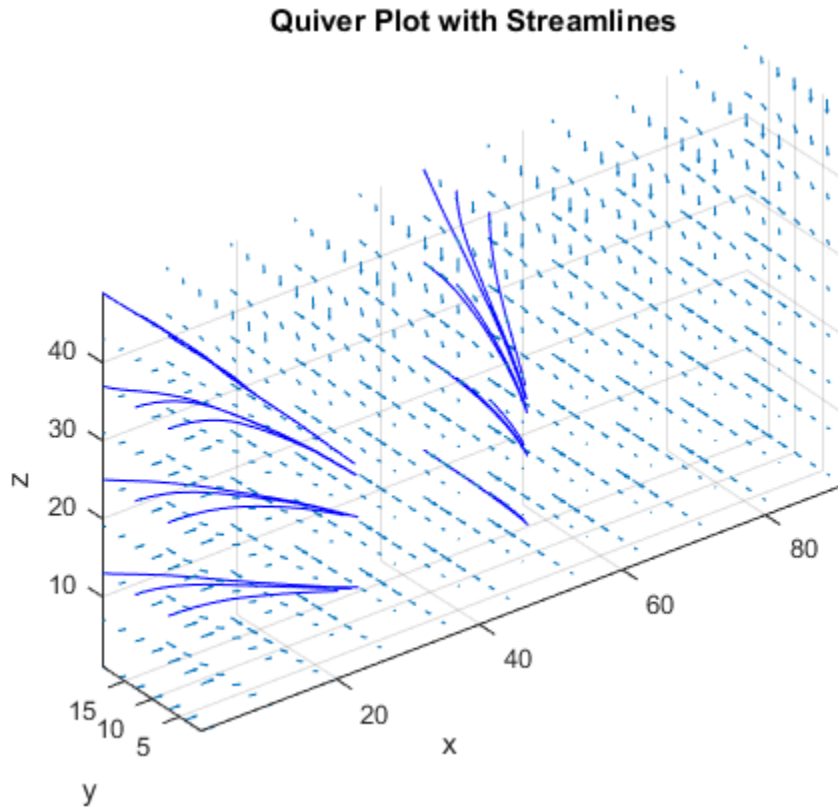

The streamlines show that small values of y and z give larger mean exit times. They also show that the x-coordinate matters has a significant effect on u when x is small, but when x is greater than 40, the larger values have little effect on u. Similarly, when z is less than 20, its values have little effect on u.

Related Examples

- "Solve Problems Using PDEModel Objects" on page 2-14

Dimensions of Solutions and Gradients

`solvepde` returns a `StationaryResults` or `TimeDependentResults` object whose properties contain the solution and its gradient at the mesh nodes. You can interpolate the solution and its gradient to other points in the geometry by using `interpolateSolution` and `evaluateGradient`.

Similarly, `solvepdeeig` returns an `EigenResults` object whose properties contain the solution eigenvectors calculated at the mesh nodes. You can interpolate the solution to other points by using `interpolateSolution`.

The dimensions of the solution and gradient depend on:

- The number of geometric evaluation points.

    - For results returned by `solvepde` or `solvepdeeig`, this is the number of mesh nodes.

    - For results returned by `interpolateSolution` or `evaluateGradient`, this is the number of query points.

- The number of equations.

    - For results returned by `solvepde` or `solvepdeeig`, this is the number of equations in the system.

    - For results returned by `interpolateSolution` or `evaluateGradient`, this is the number of query equation indices.

- The number of times for a time-dependent problem or number of modes for an eigenvalue problem.

    - For results returned by `solvepde`, this is the number of solution times (specified as an input to `solvepde`).

    - For results returned by `solvepdeeig`, this is the number of eigenvalues.

    - For results returned by `interpolateSolution` or `evaluateGradient`, this is the number of query times for time-dependent problems or query modes for eigenvalue problems.

Stationary
Scalar Problem
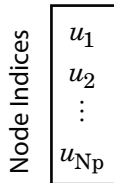
Time-Dependent or Eigenvalue
Scalar Problem

Node Indices

$$\begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_{\mathrm{Np}} \end{array}$$

Node Indices

$$\begin{array}{c} u_1(t_1) \\ u_2(t_1) \\ \vdots \\ u_{\mathrm{Np}}(t_1) \end{array} \quad \begin{array}{c} u_1(t_2) \\ u_2(t_2) \\ \vdots \\ u_{\mathrm{Np}}(t_2) \end{array} \quad \cdots \quad \begin{array}{c} u_1(t_{\mathrm{Nt}}) \\ u_2(t_{\mathrm{Nt}}) \\ \vdots \\ u_{\mathrm{Np}}(t_{\mathrm{Nt}}) \end{array}$$

Time or Mode Indices

Stationary System

Node Indices

$$\begin{array}{c} u_1^1 \\ u_2^1 \\ \vdots \\ u_{\mathrm{Np}}^1 \end{array} \quad \begin{array}{c} u_1^2 \\ u_2^2 \\ \vdots \\ u_{\mathrm{Np}}^2 \end{array} \quad \cdots \quad \begin{array}{c} u_1^N \\ u_2^N \\ \vdots \\ u_{\mathrm{Np}}^N \end{array}$$

Equation Number

Time-Dependent or Eigenvalue
System

Node lindices

Equation Number

Time or Mode Indices
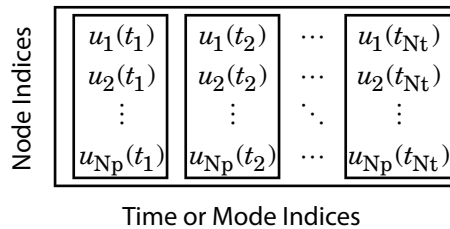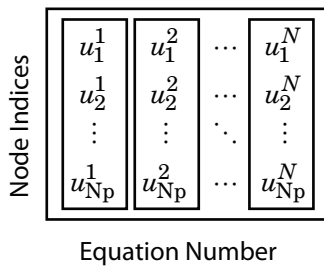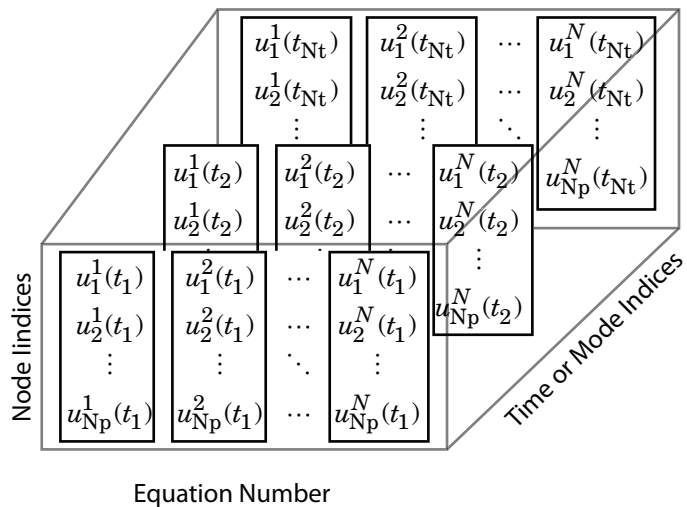
Suppose you have a problem in which:

- Np is the number of nodes in the mesh.

- Nt is the number of times for a time-dependent problem or number of modes for an eigenvalue problem.

- N is the number of equations in the system.

Suppose you also interpolate the solution and gradient to other points ("query points") in the geometry by using interpolateSolution and evaluateGradient. Here:

- Nqp is the number of query points.
- Nqt is the number of query times for a time-dependent problem or number of query modes for an eigenvalue problem.
- Nq is the number of query equations indices.

The tables show how to index into the solution returned by solvepde or solvepdeeig, where:

- iP contains the indices of nodes.
- iT contains the indices of times for a time-dependent problem or mode numbers for an eigenvalue problem.
- iN contains the indices of equations.

The tables also show the dimensions of solutions and gradients at nodal locations (returned by solvepde and solvepdeeig) and the dimensions of interpolated solutions and gradients (returned by interpolateSolution and evaluateGradient).

Stationary PDE problem	Access solution and components of gradient	Size of NodalSolution, XGradients, YGradients, and ZGradients	Size of interpolated solution and components of gradient
Scalar	result.NodalSolution(iP) result.XGradients(iP) result.YGradients(iP) result.ZGradients(iP)	Np-by-1	Nqp-by-1
System, N > 1	result.NodalSolution(iP,iN) result.XGradients(iP,iN) result.YGradients(iP,iN) result.ZGradients(iP,iN)	Np-by-N	Nqp-by-N

Time-dependent PDE problem	Access solution and components of gradient	Size of `NodalSolution`, `XGradients`, `YGradients`, and `ZGradients`	Size of interpolated solution and components of gradient
Scalar	result.NodalSolution(iP,iT) result.XGradients(iP,iT) result.YGradients(iP,iT) result.ZGradients(iP,iT)	Np-by-Nt	Nqp-by-Nqt
System, N > 1	result.NodalSolution(iP,iN,iT) result.XGradients(iP,iN,iT) result.YGradients(iP,iN,iT) result.ZGradients(iP,iN,iT)	Np-by-N-by-Nt	Nqp-by-Nq-by-Nqt

PDE eigenvalue problem	Access eigenvectors	Size of `Eigenvectors`	Size of interpolated eigenvectors
Scalar	result.Eigenvectors(iP,iT)	Np-by-Nt	Nqp-by-Nqt
System, N > 1	result.Eigenvectors(iP,iN,iT)	Np-by-N-by-Nt	Nqp-by-Nq-by-Nqt

4

PDE App

You open the PDE app by entering `pdetool` at the command line. The main components of the PDE app are the menus, the dialog boxes, and the toolbar.

Specify 2-D Geometry in the PDE App

To draw basic solid objects, such as circles and polygons, use the **Draw** menu. This menu also lets you rotate the solid objects and export the geometry to the MATLAB workspace. To cut, clear, copy, and paste the solid objects, use the **Edit** menu.

To rotate an object, select **Rotate** from the **Rotate** menu and 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].
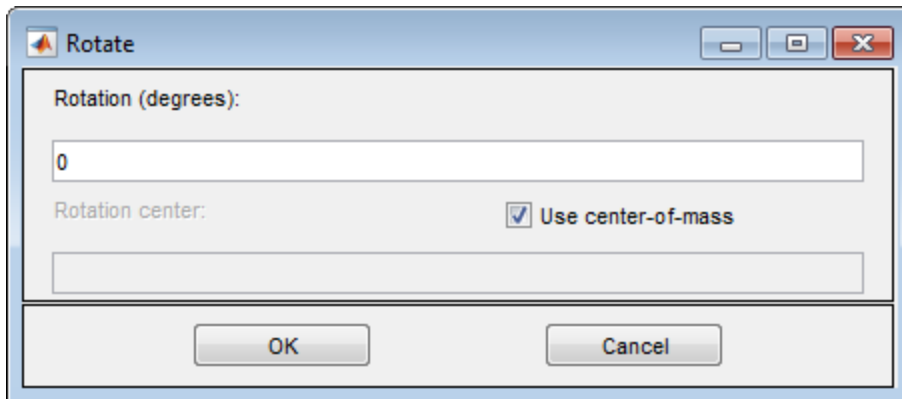
For toggling the axis grid, a "snap-to-grid" feature, and zoom, use the **Options** menu. Here you also can adjust the axes limits and the grid spacing, choose the application mode, and refresh the PDE app.

To adjust the grid spacing, select **Grid Spacing** from the **Options** menu. 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.

Clicking the **Apply** button applies the entered grid spacing. Clicking the **Done** button closes the Grid Spacing dialog.
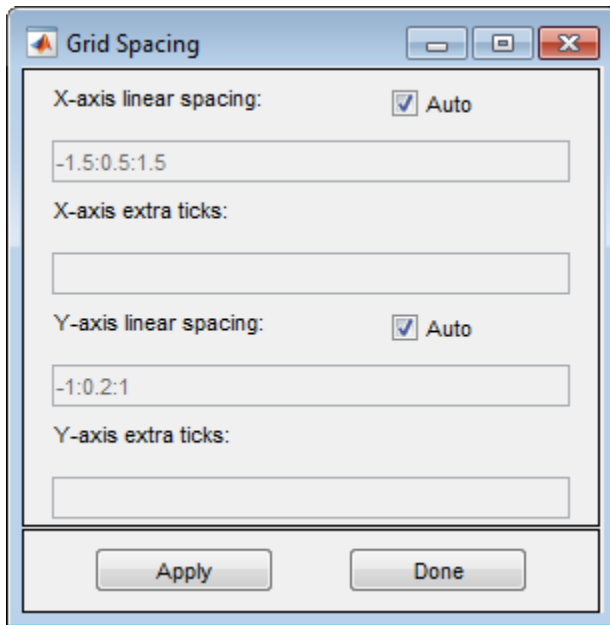
To adjust axes limits, select **Axes Limits** from the **Options** menu. In the Axes Limits dialog box, the range of the *x*-axis and the *y*-axis can be adjusted. The axis range should be entered as a 1-by-2 MATLAB vector such as [-10 10]. If you select the **Auto** check box, automatic scaling of the axis is used.

Clicking the **Apply** button applies the entered axis ranges. Clicking the **Close** button closes the Axes Limits dialog.
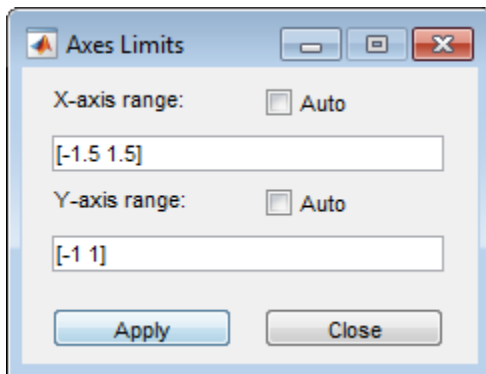
To choose the application mode, select **Application** from the **Options** menu. You also can choose the application modes using the pop-up menu in the upper right corner of the PDE app.

✓	Generic Scalar
	Generic System
	Structural Mechanics, Plane Stress
	Structural Mechanics, Plane Strain
	Electrostatics
	Magnetostatics
	AC Power Electromagnetics
	Conductive Media DC
	Heat Transfer
	Diffusion
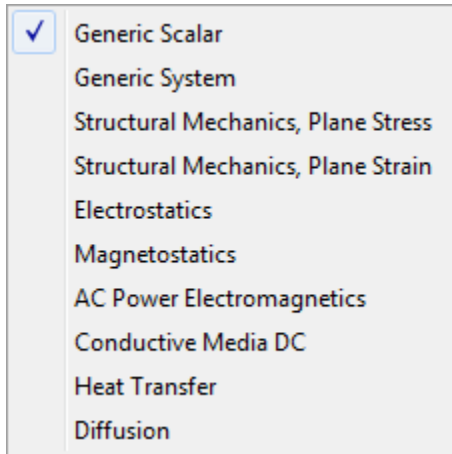
Specify Boundary Conditions in the PDE App
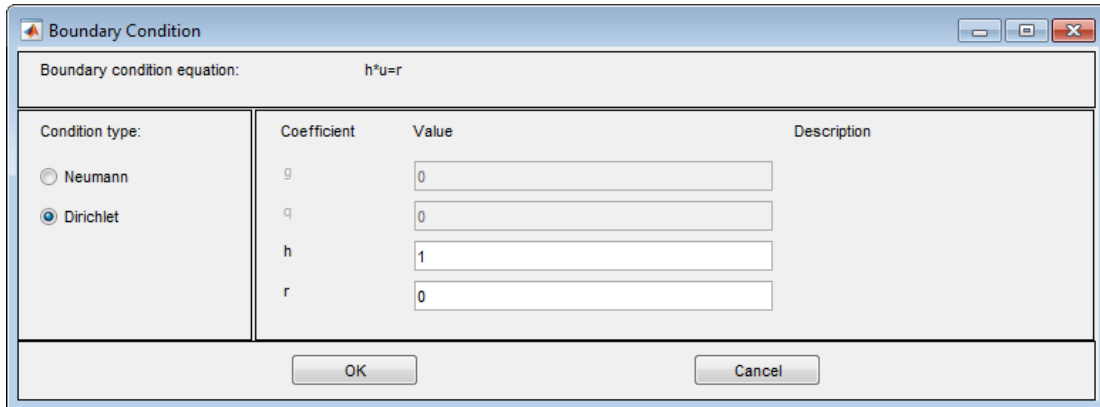
Specify Boundary Conditions opens a dialog box where you can specify the boundary condition for the selected boundary segments. There are three different condition types:

- 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.

Specify Coefficients in the PDE App
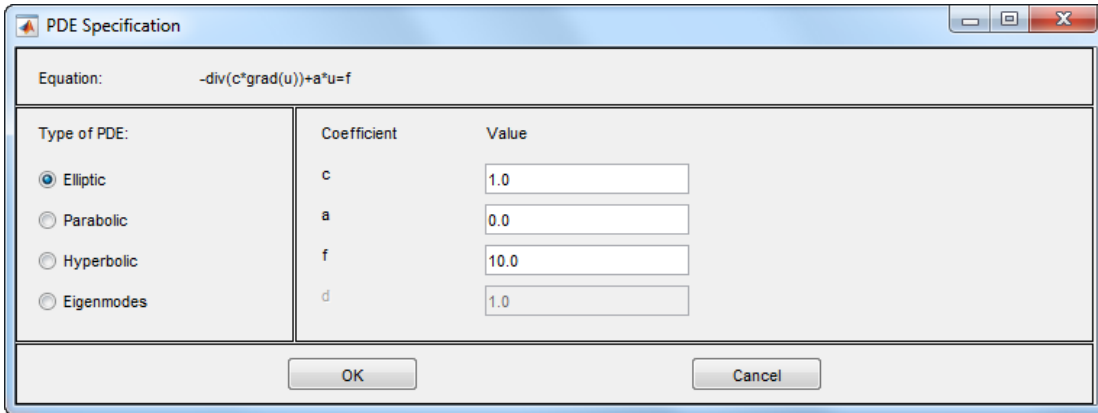
PDE Specification opens a dialog box where you enter the type of partial differential equation and the applicable parameters. The dimension of the parameters depends on the dimension of the PDE. The following description applies to scalar PDEs. If you select a nongeneric application mode, application-specific 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. These variables are available:

- x and y — The *x*- and *y*-coordinates
- u — The solution
- sd — The subdomain number
- ux and uy — The *x* and *y* derivatives of the solution
- t — The time

For details, see "Coefficients for Scalar PDEs in PDE App" on page 2-71 and "Systems in the PDE App" on page 2-95.

Specify Mesh Parameters in the PDE App

Select **Parameters** from the **Mesh** menu to open the following dialog box containing mesh generation parameters.

The parameters used by the mesh initialization algorithm 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%.

- **Mesher version:** Choose the geometry triangulation algorithm. R2013a is faster, and can mesh more geometries. preR2013a gives the same mesh as previous toolbox versions.

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

The parameters used by the mesh jiggling algorithm 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.

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.

Adjust Solve Parameters in the PDE App

To specify parameters for solving a PDE, select **Parameters** from the **Solve** menu. The set of solve parameters differs depending on the type of PDE.

Elliptic Equations

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).

  - **Relative tolerance**. This method picks triangles using a relative tolerance criterion (default: 1E-3).

  - **User-defined function**. Enter the name of a user-defined triangle selection method. See Poisson's Equation with Point Source and Adaptive Mesh Refinement 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 "Specify Mesh Parameters in the PDE App" on page 4-9.

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 Equations
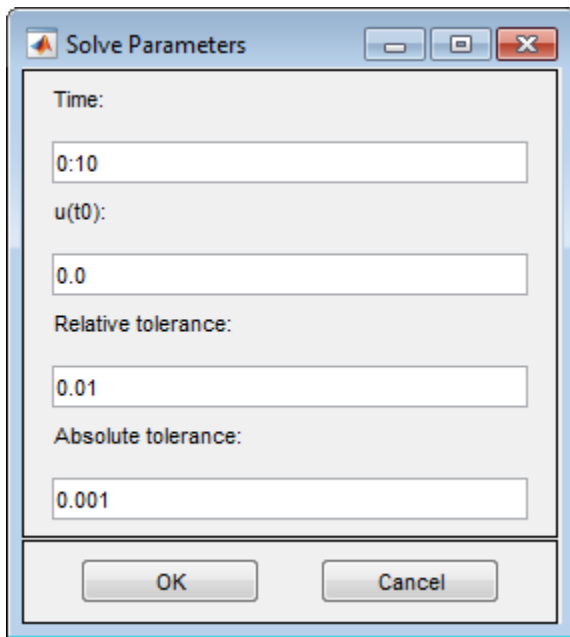
The solve parameters for the parabolic PDEs are:

- **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

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

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]`
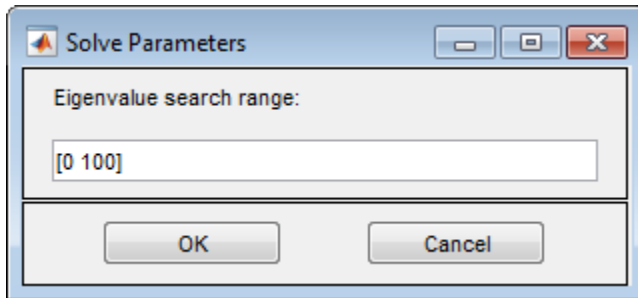
Plot the Solution in the PDE App

To plot a solution property, use the **Plot** menu. Use the **Plot Selection** dialog box to 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.

To open the **Plot Selection** dialog box, select **Parameters** from the **Plot** menu.
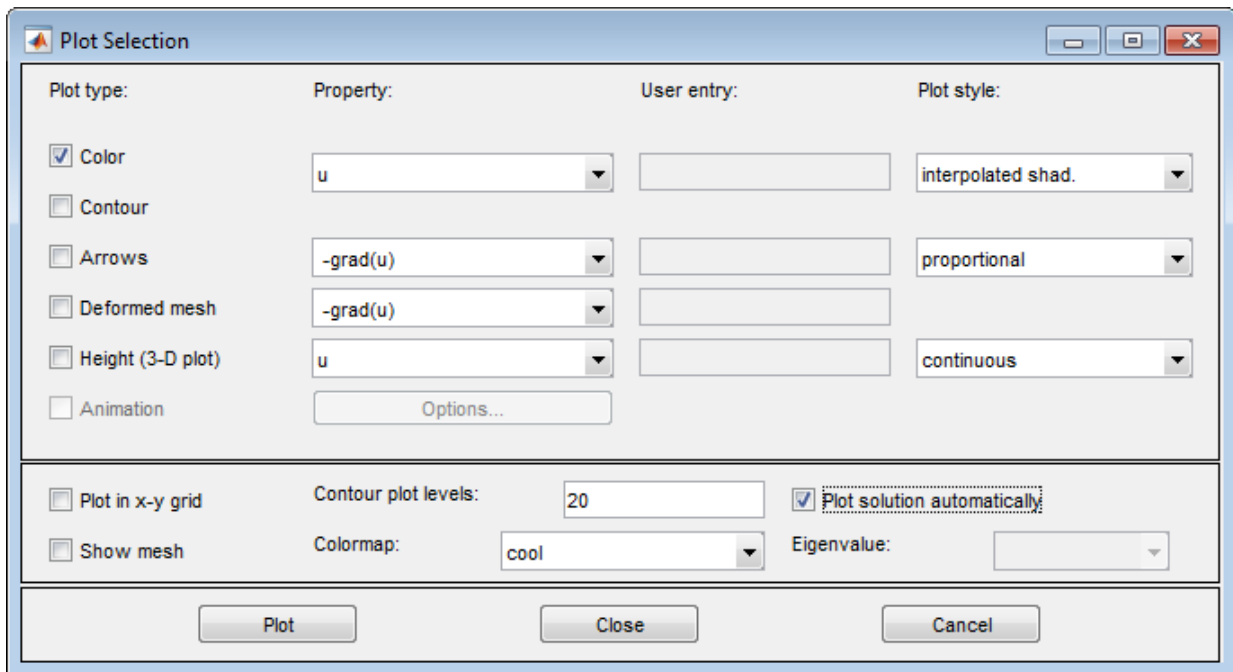
Parameters opens a dialog box containing options controlling the plotting and visualization.

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 PDE app. 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`, `prism`, and `parula`.

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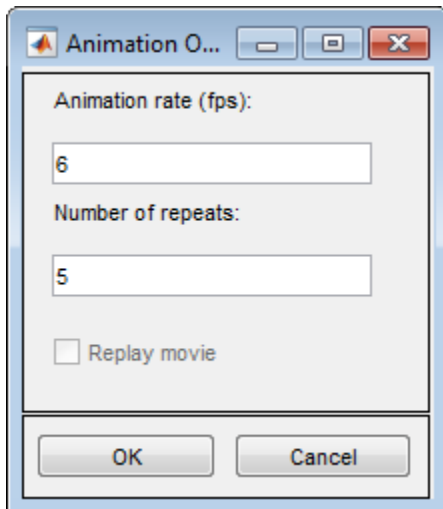
Also, the **Animation** plot type is enabled. In its property field you find an **Options** button. If you press it, an additional dialog box appears. It contains parameters that control the animation:

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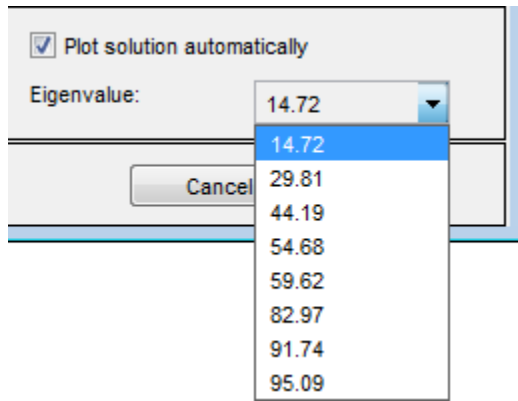
For eigenvalue problems, the bottom right part of the dialog box contains a pop-up menu with all eigenvalues. The plotted solution is the eigenvector associated with the selected eigenvalue. By default, the smallest eigenvalue is selected.

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.

5

Finite Element Method

Elliptic Equations

Partial Differential Equation Toolbox solves equations of the form

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

When the m and d coefficients are 0, this reduces to

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

which the documentation calls an *elliptic* equation, whether or not the equation is elliptic in the mathematical sense. The equation holds in Ω, where Ω is a bounded domain in two or three dimensions. 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. For 2-D geometry, create geometry using the PDE app or through MATLAB files. For 3-D geometry, import the geometry in STL file format. See "2-D Geometry", "Create and View 3-D Geometry" on page 2-47, and "Boundary Conditions".

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 elements.

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 5-13.

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:

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

Replace the original problem with *Find u such that*

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

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 $N_p{\rightarrow}\infty$. 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.,

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

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

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

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 element of a 2-D mesh, and are linear or quadratic on elements of a 3-D 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 an element 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} in 2-D is the set of "tent" or "hat" functions ϕ_i. These are linear on each element and take the value 0 at all nodes x_j except for x_i. For the definition of basis functions for 3-D geometry, see "Finite Element Basis for 3-D" on page 5-10. Requesting $\phi_i(x_i) = 1$ yields the very pleasant property

$$u(x_i) = \sum_{j=1}^{N_p} U_j \phi_j(x_i) = U_i.$$

That is, by solving the FEM system we obtain the nodal values of the approximate solution. The basis function ϕ_i vanishes on all the elements 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 elements 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 element 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 element to the corresponding entries (i.e., only if the corresponding mesh point is a vertex of the element). This process is commonly called *assembling*, hence the name of the function `assempde`.

The assembling routines scan the elements of the mesh. For each element they compute the so-called local matrices and add their components to the correct positions in the sparse matrices or vectors.

The discussion now specializes to triangular meshes in 2-D. 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.
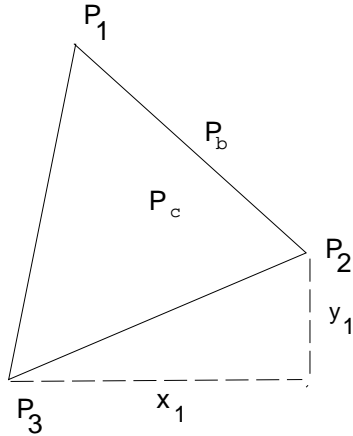
The Local Triangle P1P2P3

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:

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

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 $\phi(P_i) = 1$. 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)

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

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

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

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.

Finite Element Basis for 3-D

The finite element method for 3-D geometry is similar to the 2-D method described in "Elliptic Equations" on page 5-2. The main difference is that the elements in 3-D geometry are tetrahedra, which means that the basis functions are different from those in 2-D geometry.

It is convenient to map a tetrahedron to a canonical tetrahedron with a local coordinate system (r,s,t).
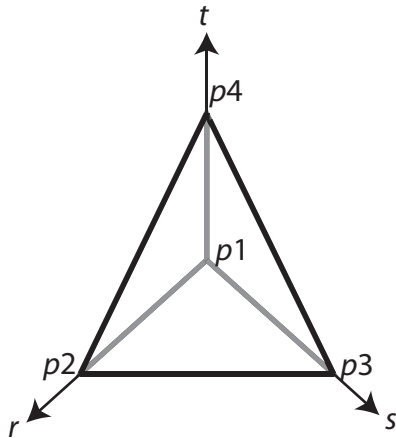
In local coordinates, the point $p1$ is at $(0,0,0)$, $p2$ is at $(1,0,0)$, $p3$ is at $(0,1,0)$, and $p4$ is at $(0,0,1)$.

For a linear tetrahedron, the basis functions are

$$\phi_1 = 1 - r - s - t$$
$$\phi_2 = r$$
$$\phi_3 = s$$
$$\phi_4 = t.$$

For a quadratic tetrahedron, there are additional nodes at the edge midpoints.
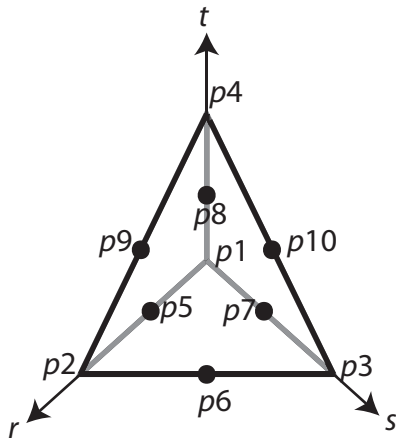
The corresponding basis functions are

$$\phi_1 = 2(1-r-s-t)^2 - (1-r-s-t)$$
$$\phi_2 = 2r^2 - r$$
$$\phi_3 = 2s^2 - s$$
$$\phi_4 = 2t^2 - t$$
$$\phi_5 = 4r(1-r-s-t)$$
$$\phi_6 = 4rs$$
$$\phi_7 = 4s(1-r-s-t)$$
$$\phi_8 = 4t(1-r-s-t)$$
$$\phi_9 = 4rt$$
$$\phi_{10} = 4st.$$

As in the 2-D case, a 3-D basis function ϕ_i takes the value 0 at all nodes j, except for node i, where it takes the value 1.

See Also
FEMesh

More About

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-D-by-D tensor, and D is the geometry dimensions, 2 or 3.

For 2-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an N-by-1 matrix with an $(i,1)$-component

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

For 3-D systems, the notation $\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an N-by-1 matrix with an $(i,1)$-component

$$\sum_{j=1}^{N}\left(\frac{\partial}{\partial x}c_{i,j,1,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial x}c_{i,j,1,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial x}c_{i,j,1,3}\frac{\partial}{\partial z}\right)u_j$$

$$+\sum_{j=1}^{N}\left(\frac{\partial}{\partial y}c_{i,j,2,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial y}c_{i,j,2,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial y}c_{i,j,2,3}\frac{\partial}{\partial z}\right)u_j$$

$$+\sum_{j=1}^{N}\left(\frac{\partial}{\partial z}c_{i,j,3,1}\frac{\partial}{\partial x}+\frac{\partial}{\partial z}c_{i,j,3,2}\frac{\partial}{\partial y}+\frac{\partial}{\partial z}c_{i,j,3,3}\frac{\partial}{\partial z}\right)u_j.$$

The symbols **a** and **d** denote N-by-N matrices, and **f** denotes a column vector of length N.

The elements c_{ijkl}, a_{ij}, d_{ij}, and f_i of **c**, **a**, **d**, and **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,

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

For 2-D systems, the notation $\mathbf{n}\cdot(\mathbf{c}\otimes\nabla\mathbf{u})$ represents an N-by-1 matrix with $(i,1)$-component

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

where the outward normal vector of the boundary is $\mathbf{n}=(\cos(\alpha),\sin(\alpha))$.

For 3-D systems, the notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ represents an N-by-1 matrix with $(i,1)$-component

$$\sum_{j=1}^{N} \left(\cos(\alpha)c_{i,j,1,1} \frac{\partial}{\partial x} + \cos(\alpha)c_{i,j,1,2} \frac{\partial}{\partial y} + \cos(\alpha)c_{i,j,1,3} \frac{\partial}{\partial z} \right) u_j$$

$$+ \sum_{j=1}^{N} \left(\cos(\beta)c_{i,j,2,1} \frac{\partial}{\partial x} + \cos(\beta)c_{i,j,2,2} \frac{\partial}{\partial y} + \cos(\beta)c_{i,j,2,3} \frac{\partial}{\partial z} \right) u_j$$

$$+ \sum_{j=1}^{N} \left(\cos(\gamma)c_{i,j,3,1} \frac{\partial}{\partial x} + \cos(\gamma)c_{i,j,3,2} \frac{\partial}{\partial y} + \cos(\gamma)c_{i,j,3,3} \frac{\partial}{\partial z} \right) u_j,$$

where the outward normal to the boundary is

$$\mathbf{n} = \left(\cos(\alpha), \cos(\beta), \cos(\gamma) \right).$$

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

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 mesh. Then the number of unknowns is $N_pN = N_u$. When Dirichlet boundary conditions fix some of the unknowns, the linear system can be correspondingly reduced. This is easily done by removing rows and columns when u values are given, but here we must treat the case when some linear combinations of the components of u are given, $\mathbf{hu} = \mathbf{r}$. 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`.

Partial Differential Equation Toolbox solves equations of the form

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

When the m coefficient is 0, but d is not, the documentation refers to the equation as *parabolic*, whether or not it is mathematically in parabolic form.

A parabolic problem is to solve the equation

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

with the initial condition
$u(\mathbf{x},0) = u_0(\mathbf{x})$ for $\mathbf{x}\in\Omega$

where $\mathbf{x}$ represents a 2-D or 3-D point and there are 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 mesh on Ω and $t \geq 0$, expand the solution to the PDE (as a function of $\mathbf{x}$) in the Finite Element Method basis:

$$u(\mathbf{x},t) = \sum_i U_i(t)\phi_i(\mathbf{x}).$$

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

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

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
$U_i(0) = u_0(\mathbf{x}_i)$

yields the solution to the PDE at each node $\mathbf{x}_i$ and time t. Note that K and F are the stiffness matrix and the right-hand side of the elliptic problem
$-\nabla \cdot (c\nabla u) + au = f$ in Ω

with the original boundary conditions, while M is just the mass matrix of the problem
$-\nabla \cdot (0\nabla u) + du = 0$ in Ω.

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.

Hyperbolic Equations

Partial Differential Equation Toolbox solves equations of the form

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

When the d coefficient is 0, but m is not, the documentation calls this a *hyperbolic* equation, whether or not it is mathematically of the hyperbolic form.

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

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

for $\mathbf{x}$ in Ω, where $\mathbf{x}$ represents a 2-D or 3-D point, with the initial conditions

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

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

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

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

with the initial conditions

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

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

$-\nabla \cdot (c\nabla u) + au = f$ and $-\nabla \cdot (0\nabla u) + mu = 0$.

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 $a(\mathbf{x})$, where $\mathbf{x}$ represents a 2-D or 3-D point.

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.,

$$M_{i,j} = \int_{\Omega} d(\mathbf{x})\phi_j(\mathbf{x})\phi_i(\mathbf{x}) \, d\mathbf{x}$$

The matrices K and M are produced by calling `assema` for the equations
$-\nabla \cdot (c\nabla u) + au = 0$ and $-\nabla \cdot (0\nabla u) + du = 0$

In the most common case, when the function $d(\mathbf{x})$ is positive, the mass matrix M is positive definite symmetric. Likewise, when $c(\mathbf{x})$ 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 examples "Eigenvalues and Eigenfunctions for the L-Shaped Membrane" on page 3-110 or "Eigenvalues and Eigenmodes of a Square" on page 3-119, where actual runs are 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 $h_{j+1}v_{j+1} = Av_j - V_jh_j$, 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

$$AV_j = V_jH_{j,j} + v_{j+1}h_{j+1,j}e_j^T$$

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 $H_{j,j}$,

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

Then $y_i = V_js_i$ is an approximate eigenvector of A, and its residual is

$$r_i = Ay_i - y_i\theta_i = AV_js_i - V_js_i\theta_i = (AV_j - V_jH_{j,j})s_i = v_{j+1}h_{j+1,j}s_{i,j}$$

This residual has to be small in norm for θ_i to be a good eigenvalue approximation. The norm of the residual is

$$\|r_i\| = \left|h_{j+1,j}s_{j,i}\right|,$$

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

Before solving a nonlinear elliptic PDE, from the **Solve** menu in the PDE app, select **Parameters**. Then, select the **Use nonlinear solver** check box and click **OK**. At the command line, use `solvepde`.

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

$$u(\mathbf{x}) = \sum U_j \phi_j$$

where $\mathbf{x}$ represents a 2-D or 3-D point. 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

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

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 $\left\| \cdot \right\|$ 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 $\rho\left(U^{(n)} + \alpha p_n\right)$.

An important point of this strategy is that when $U^{(n)}$ approaches the solution, then $\alpha \rightarrow 1$ and thus the convergence rate increases. If there is a solution to $\rho(U) = 0$, the scheme ultimately recovers the quadratic convergence rate of the standard Newton iteration.

Closely related to the preceding problem is the choice of the initial guess $U^{(0)}$. By default, the solver sets $U^{(0)}$ and then assembles the FEM matrices K and F and computes
$$U^{(1)} = K^{-1}F$$

The damped Gauss-Newton iteration is then started with $U^{(1)}$, which should be a better guess than $U^{(0)}$. If the boundary conditions do not depend on the solution u, then $U^{(1)}$ satisfies them even if $U^{(0)}$ does not. Furthermore, if the equation is linear, then $U^{(1)}$ is the exact FEM solution and the solver does not enter the Gauss-Newton loop.

There are situations where $U^{(0)} = 0$ makes no sense or convergence is impossible.

In some situations you may already have a good approximation and the nonlinear solver can be started with it, avoiding the slow convergence regime. This idea is used in the adaptive mesh generator. It computes a solution $\tilde{U}$ on a mesh, evaluates the error, and may refine certain triangles. The interpolant of $\tilde{U}$ is a very good starting guess for the solution on the refined mesh.

In general the exact Jacobian

$$J_n = \frac{\partial \rho\left(U^{(n)}\right)}{\partial U}$$

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

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

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

$$U^{(n+1)} = U^{(n)} - J_n^{-1}\rho\left(U^{(n)}\right) = U^{(n)} - K^{-1}\left(KU^{(n)} - F\right) = K^{-1}F.$$

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.

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

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

$$\delta_{i,j} \sum_l \int_\Omega \frac{\partial c}{\partial u} \nabla\phi_l \nabla\phi_i \ d\mathbf{x} U_l$$

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 $U \rightarrow MU$. The derivative of the mapping $U \rightarrow -F$ is exactly

$$-\int_\Omega \frac{\partial f}{\partial u} \phi_i \phi_j \ d\mathbf{x}$$

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

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

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.

Functions — Alphabetical List

adaptmesh

Adaptive 2-D mesh generation and PDE solution

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. In the recommended workflow, see `generateMesh` for mesh generation and `solvepde` for 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 for elliptic problems with 2-D geometry. 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 $(x,y) \in \Omega$, 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`. If the PDE is scalar, meaning only one equation, then `u` is a column vector representing the solution u at each node

in the mesh. `u(i)` is the solution at the `ith` column of `model.Mesh.Nodes` or the `ith` column of `p`. If the PDE is a system of $N > 1$ equations, then `u` is a column vector with N*`Np` elements, where `Np` is the number of nodes in the mesh. The first `Np` elements of `u` represent the solution of equation 1, then next `Np` elements represent the solution of equation 2, etc.

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 geometry of the PDE problem. `g` can be a Decomposed Geometry matrix, the name of a Geometry file, or a function handle to a Geometry file. For details, see "2-D Geometry".

`b` describes the boundary conditions of the PDE problem. For the recommended way of specifying boundary conditions, see "Specify Boundary Conditions Objects" on page 2-179. For all methods of specifying boundary conditions, see "Forms of Boundary Condition Specification" on page 2-170.

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 "Mesh Data" on page 2-211.

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 2-66 and "Coefficients for Systems of PDEs" on page 2-93.

The following table lists the property name/value value pairs, their default values, and descriptions of the properties.

Property	Value	Default	Description
Maxt	positive integer	inf	Maximum number of new triangles

Property	Value	Default	Description
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
Jac	'fixed | 'lumped' | 'full'	'fixed'	Nonlinear Jacobian calculation
norm	numeric | inf	inf	Nonlinear residual norm
MesherVersion	'R2013a' | 'preR2013a'	'preR2013a'	Algorithm for generating initial mesh

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 `MesherVersion` property chooses the algorithm for mesh generation. The `'R2013a'` algorithm runs faster, and can triangulate more geometries than the `'preR2013a'` algorithm. Both algorithms use Delaunay triangulation.

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`.

Examples

Solve the Laplace equation over a circle sector, with Dirichlet boundary conditions $u = \cos(2/3\mathrm{atan2}(y, x))$ along the arc, and $u = 0$ along the straight lines, and compare to the exact solution. Set options so that `adaptmesh` refines the triangles using the worst error criterion until it obtains 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))

Number of triangles: 197
Number of triangles: 201
Number of triangles: 216
Number of triangles: 233
Number of triangles: 254
Number of triangles: 265
Number of triangles: 313
Number of triangles: 344
Number of triangles: 417
```

```
Number of triangles: 475
Number of triangles: 629

Maximum number of triangles obtained.

ans =

    0.0028


size(t,2)


ans =

   629
```

The maximum absolute error is 0.0028, with 629 triangles.

```
pdemesh(p,e,t)
```
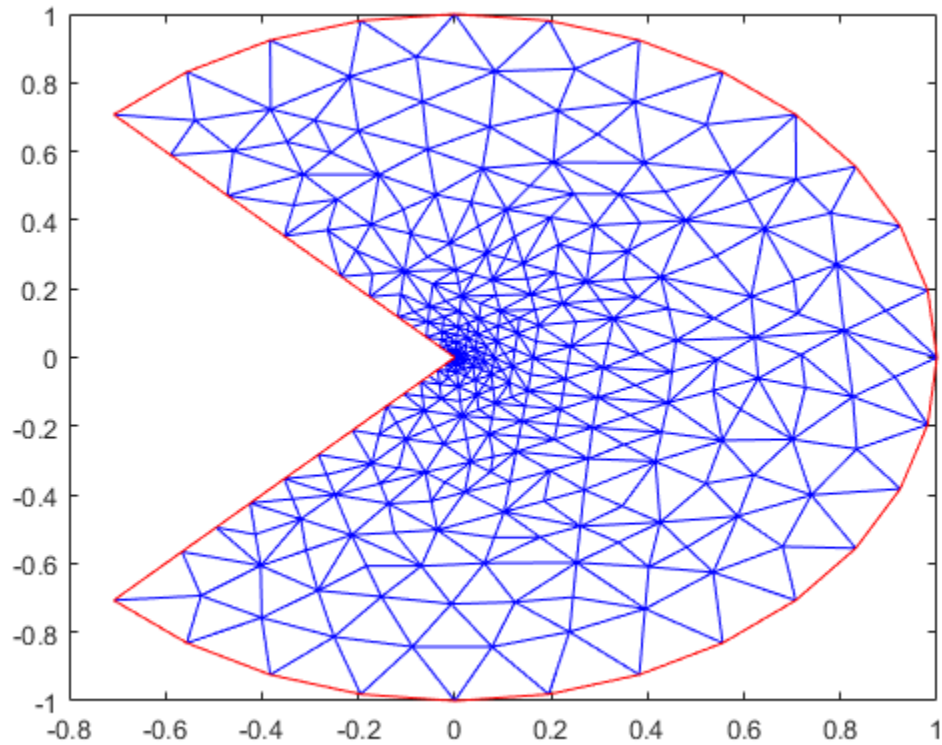
Test how many refinements you have to use with a uniform triangle net:

```
[p,e,t]=initmesh('cirsg');
[p,e,t]=refinemesh('cirsg',p,e,t);
u=assempde('cirsb',p,e,t,1,0,0);
x=p(1,:); y=p(2,:);
exact=((x.^2+y.^2).^(1/3).*cos(2/3*atan2(y,x)))';
max(abs(u-exact))

ans =

    0.0121
```

```
size(t,2)

ans =

   788


[p,e,t]=refinemesh('cirsg',p,e,t);
u=assempde('cirsb',p,e,t,1,0,0);
x=p(1,:); y=p(2,:);
exact=((x.^2+y.^2).^(1/3).*cos(2/3*atan2(y,x)))';
max(abs(u-exact))

ans =

   0.0078


size(t,2)

ans =

      3152


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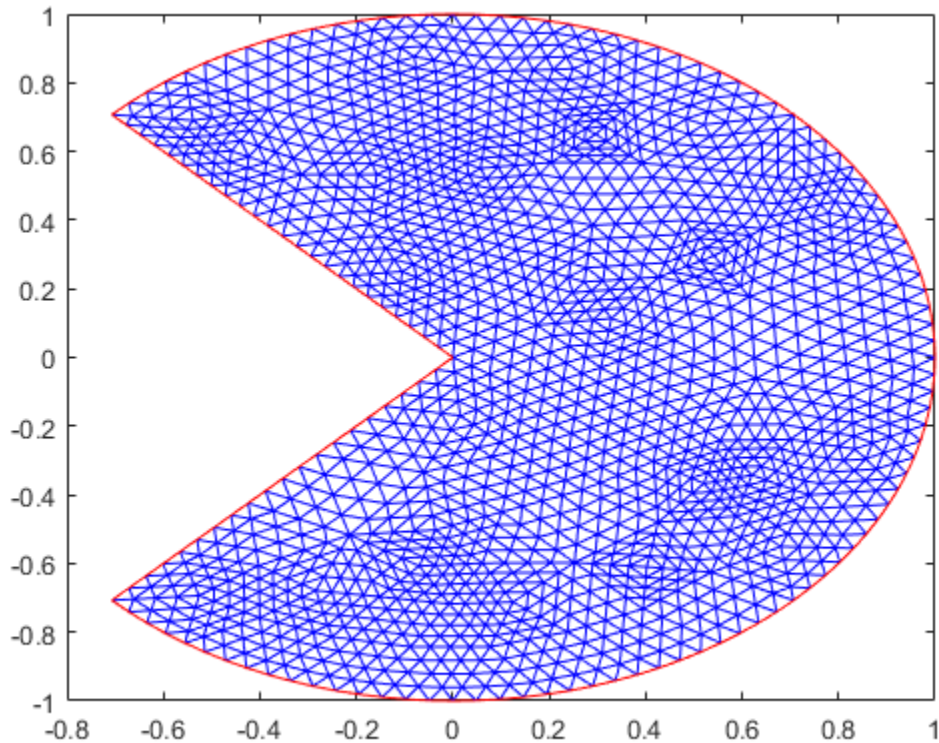
Uniform refinement with 3152 triangles produces an error of 0.0078. This error is over three times as large as that produced by the adaptive method (0.0028) with many fewer triangles (629). For a problem with regular solution, we expect an $O(h^2)$ error, but this solution is singular since $u \approx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

AnalyticGeometry Properties

2-D geometry description

Compatibility

THIS PAGE DESCRIBES BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Description

`AnalyticGeometry` describes 2-D geometry in the form of an object. A `PDEModel` object has a `Geometry` property. For 2-D geometry, the `Geometry` property is an `AnalyticGeometry` object.

Specify a 2-D geometry for your model using the `geometryFromEdges` function.

Properties

`NumEdges` — Number of geometry edges
positive integer

Number of geometry edges, returned as a positive integer.

Data Types: `double`

`NumFaces` — Number of geometry faces
positive integer

Number of geometry faces, returned as a positive integer. If your geometry is one connected region, then `NumFaces = 1`.

Data Types: `double`

`NumVertices` — Number of geometry vertices
positive integer

Number of geometry vertices, returned as a positive integer.

Data Types: `double`

See Also
`geometryFromEdges` | `PDEModel`

More About

- "Solve Problems Using PDEModel Objects" on page 2-14

Introduced in R2015a

applyBoundaryCondition

Add boundary condition to `PDEModel` container

Compatibility

SPECIFYING BOUNDARY CONDITIONS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
applyBoundaryCondition(model,RegionType,RegionID,Name,Value)
bc = applyBoundaryCondition(model,RegionType,RegionID,Name,Value)
```

Description

`applyBoundaryCondition(model,RegionType,RegionID,Name,Value)` adds a boundary condition to `model`. The boundary condition applies to boundary regions of type `RegionType` with ID numbers in `RegionID`, and with values specified in the `Name,Value` pairs.

`bc = applyBoundaryCondition(model,RegionType,RegionID,Name,Value)` returns the boundary condition object.

Examples

Apply Dirichlet and Neumann Boundary Conditions

Apply both types of boundary conditions to a scalar problem.

Create a PDE model and import a simple block geometry.

```
model = createpde;
importGeometry(model,'Block.stl');
```

View the face labels.

```
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
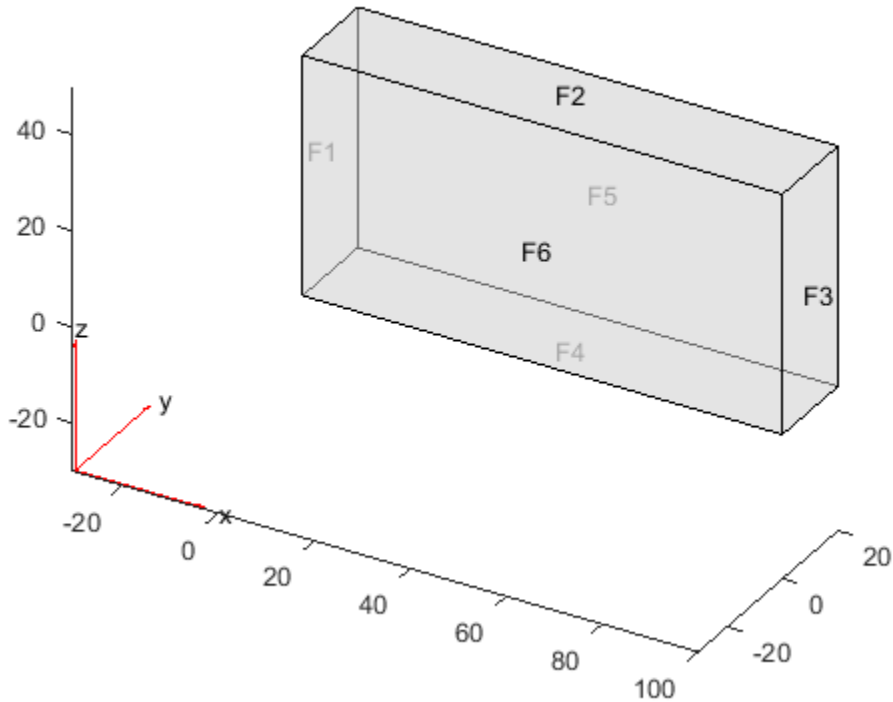

Set zero Dirichlet conditions on the narrow edges, which are labeled 1 through 4.

```
applyBoundaryCondition(model,'face',1:4,'u',0);
```

Set Neumann boundary conditions with opposite signs on faces 5 and 6.

```
applyBoundaryCondition(model,'face',5,'g',1);
applyBoundaryCondition(model,'face',6,'g',-1);
```

Solve an elliptic PDE with these boundary conditions, and plot the result.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',0);
```

```
generateMesh(model);
results = solvepde(model);
u = results.NodalSolution;
pdeplot3D(model,'colormapdata',u)
```
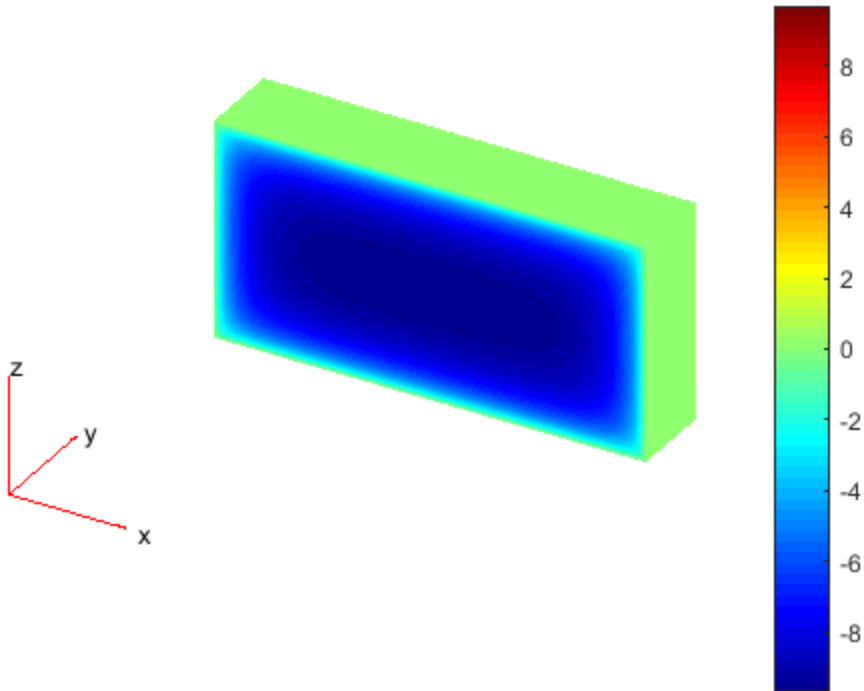

Input Arguments

model — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

`RegionType` — Boundary type
`'face'` for 3-D geometry | `'edge'` for 2-D geometry

Boundary type, specified as `'face'` for 3-D geometry or `'edge'` for 2-D geometry.

Example: `applyBoundaryCondition(model,'face',3,'u',0)`

Data Types: `char`

`RegionID` — Boundary ID
vector of positive integers

Boundary ID, specified as a vector of positive integers. To determine which ID corresponds to which portion of the geometry boundary, use the `pdegplot` function. Set the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) name-value pair set to `'on'`.

Example: `applyBoundaryCondition(model,'face',3:6,'u',0)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Note: You can set only one type of boundary condition in a call to `applyBoundaryCondition`: a `'u'`,`EquationIndex` pair, or an `'r'`,`'h'` pair, or a `'g'`,`'q'` pair. If you set only one member of a pair, the other takes its default value.

Example: `applyBoundaryCondition(model,'face',1:4,'u',0)`

`'r'` — Dirichlet condition `h*u = r`
`zeros(N,1)` (default) | vector with N elements | function handle

Dirichlet condition `h*u = r`, specified as a vector with N elements or as a function handle. N is the number of PDEs in the system. For the syntax of the function handle form of `r`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'r',[0;4;-1]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'h'` — Dirichlet condition `h*u = r`
`eye(`*N*`)` (default) | *N*-by-*N* matrix | vector with N^2 elements | function handle

Dirichlet condition `h*u = r`, specified as an *N*-by-*N* matrix, as a vector with N^2 elements, or as a function handle. *N* is the number of PDEs in the system. For the syntax of the function handle form of `h`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'h',[2,1;1,2]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'g'` — Generalized Neumann condition `n·(c×∇u) + qu = g`
`zeros(`*N*`,1)` (default) | vector with *N* elements | function handle

Generalized Neumann condition `n·(c×∇u) + qu = g`, specified as a vector with *N* elements or as a function handle. *N* is the number of PDEs in the system. For the syntax of the function handle form of `g`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'g',[3;2;-1]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'q'` — Generalized Neumann condition `n·(c×∇u) + qu = g`
`zeros(`*N*`)` (default) | *N*-by-*N* matrix | vector with N^2 elements | function handle

Generalized Neumann condition `n·(c×∇u) + qu = g`, specified as an *N*-by-*N* matrix, as a vector with N^2 elements, or as a function handle. *N* is the number of PDEs in the system. For the syntax of the function handle form of `q`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'q',eye(3)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'u'` — Dirichlet conditions
`zeros(`*N*`,1)` (default) | vector of up to *N* elements | function handle

Dirichlet conditions, specified as a vector of up to N elements or as a function handle. `EquationIndex` and u must have the same length. For the syntax of the function handle form of u, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `applyBoundaryCondition(model,'face',[2,4,11],'u',0)`

Data Types: `double`
Complex Number Support: Yes

`'EquationIndex'` — Index of specified u components
1:N (default) | vector of integers with entries from 1 to N

Index of specified u components, specified as a vector of integers with entries from 1 to N. `EquationIndex` and u must have the same length.

Example: `applyBoundaryCondition(model,'face',[2,4,11],'u',[3,-1],'EquationIndex',[2,3])`

Data Types: `double`

`'Vectorized'` — Vectorized function evaluation
`'off'` (default) | `'on'`

Vectorized function evaluation, specified as `'on'` or `'off'`. This evaluation applies when you pass a function handle as an argument. To save time in function handle evaluation, specify `'on'`, assuming that your function handle computes in a vectorized fashion. See "Vectorization". For details of this evaluation, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `applyBoundaryCondition(model,'face',[2,4,11],'u',@ucalculator,'Vectorized','on')`

Data Types: `char`

Output Arguments

bc — Boundary condition
`BoundaryCondition` object

Boundary condition, returned as a `BoundaryCondition` object. The `model` object contains a vector of `BoundaryCondition` objects. `bc` is the last element of this vector.

More About

- "Solve Problems Using PDEModel Objects" on page 2-14

See Also

BoundaryCondition Properties | PDEModel

Introduced in R2015a

assema

Assemble area integral contributions

Compatibility

assema **IS NOT RECOMMENDED.** Use `assembleFEMatrices` instead.

Syntax

```
[K,M,F] = assema(model,c,a,f)
[K,M,F] = assema(p,t,c,a,f)
```

Description

`[K,M,F] = assema(model,c,a,f)` assembles the stiffness matrix K, the mass matrix M, and the load vector F using the mesh contained in `model`, and the PDE coefficients c, a, and f.

`[K,M,F] = assema(p,t,c,a,f)` assembles the matrices from the mesh data in p and t.

Examples

Assemble Finite Element Matrices

Assemble finite element matrices for an elliptic problem on complicated geometry.

The PDE is Poisson's equation,

$$-\nabla \cdot \nabla u = 1.$$

Partial Differential Equation Toolbox™ solves equations of the form

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

So, represent Poisson's equation in toolbox syntax by setting c = 1, a = 0, and f = 1.

```
c = 1;
a = 0;
f = 1;
```

Create a PDE model container. Import the `ForearmLink.stl` file into the model and examine the geometry.

```
model = createpde;
importGeometry(model,'ForearmLink.stl');
h = pdegplot(model);
h(1).FaceAlpha = 0.5;
```


Create a mesh for the model.

```
generateMesh(model);
```
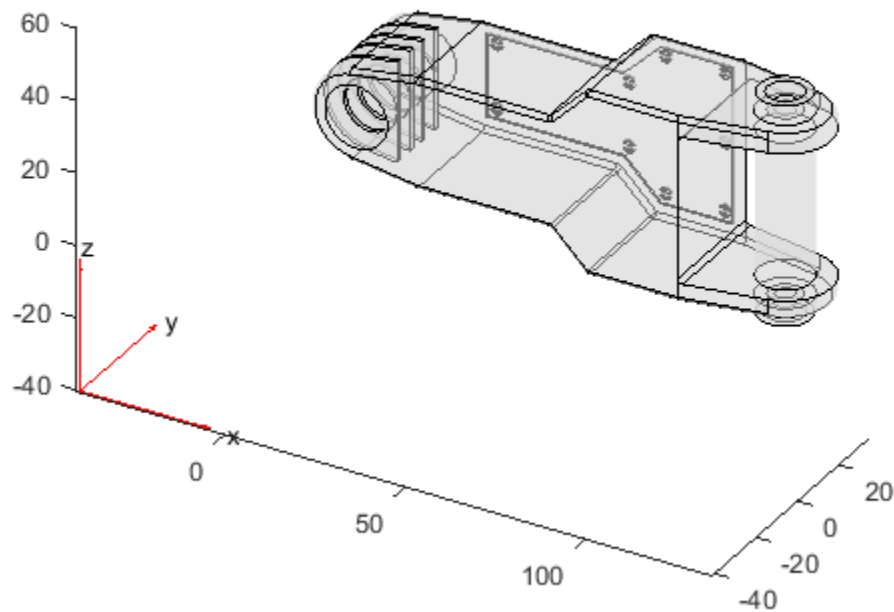
Create the finite element matrics from the mesh and the coefficients.

```
[K,M,F] = assema(model,c,a,f);
```

The returned matrix K is quite sparse. M has no nonzero entries.

```
disp(['Fraction of nonzero entries in K is ',num2str(nnz(K)/numel(K))])
disp(['Number of nonzero entries in M is ',num2str(nnz(M))])
```

```
Fraction of nonzero entries in K is 0.00071285
Number of nonzero entries in M is 0
```

Assemble Finite Element Matrices Using [p,e,t] Mesh

Assemble finite element matrices for the 2-D L-shaped region, using the [p,e,t] mesh representation.

Define the geometry using the lshapeg function included your software.

```
g = @lshapeg;
```

Use coefficients c = 1, a = 0, and f = 1.

```
c = 1;
a = 0;
f = 1;
```

Create a mesh and assemble the finite element matrices.

```
[p,e,t] = initmesh(g);
[K,M,F] = assema(p,t,c,a,f);
```

The returned matrix M has all zeros. The K matrix is quite sparse.

```
disp(['Fraction of nonzero entries in K is ',num2str(nnz(K)/numel(K))])
disp(['Number of nonzero entries in M is ',num2str(nnz(M))])
```

```
Fraction of nonzero entries in K is 0.042844
Number of nonzero entries in M is 0
```

Input Arguments

model — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

c — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. c represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify c in various ways, detailed in "c Coefficient for Systems" on page 2-125. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: 'cosh(x+y.^2)'

Data Types: double | char | function_handle
Complex Number Support: Yes

a — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. a represents the *a* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify a in various ways, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page

2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double | char | function_handle`
Complex Number Support: Yes

f — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. `f` represents the *f* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify `f` in various ways, detailed in "f Coefficient for Systems" on page 2-98. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double | char | function_handle`
Complex Number Support: Yes

p — Mesh nodes
output of `initmesh` | output of `meshToPet`

Mesh nodes, specified as the output of `initmesh` or `meshToPet`. For the structure of a `p` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

t — Mesh elements
output of `initmesh` | output of `meshToPet`

Mesh elements, specified as the output of `initmesh` or `meshToPet`. Mesh elements are the triangles or tetrahedra that form the finite element mesh. For the structure of a `t` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

Output Arguments

K — Stiffness matrix
sparse matrix

Stiffness matrix, returned as a sparse matrix. See "Elliptic Equations" on page 5-2.

Typically, you use `K` in a subsequent call to `assempde`.

M — Mass matrix
sparse matrix

Mass matrix. returned as a sparse matrix. See "Elliptic Equations" on page 5-2.

Typically, you use `M` in a subsequent call to a solver such as `assempde` or `hyperbolic`.

F — Load vector
vector

Load vector, returned as a vector. See "Elliptic Equations" on page 5-2.

Typically, you use `F` in a subsequent call to `assempde`.

See Also
`assembleFEMatrices`

Introduced before R2006a

assemb

Assemble boundary condition contributions

Compatibility

`assemb` **IS NOT RECOMMENDED.** Use `assembleFEMatrices` instead.

Syntax

```
[Q,G,H,R] = assemb(model)
[Q,G,H,R] = assemb(b,p,e)
[Q,G,H,R] = assemb( ___ ,[],sdl)
```

Description

`[Q,G,H,R] = assemb(model)` 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.

`[Q,G,H,R] = assemb(b,p,e)` assembles the matrices based on the boundary conditions specified in `b` and the mesh data in `p` and `e`.

`[Q,G,H,R] = assemb( ___ ,[],sdl)`, for any of the previous input arguments, restricts the finite element matrices to those that include the subdomain specified by the subdomain labels in `sdl`. The empty argument is required in this syntax for historic and compatibility reasons.

Examples

Assemble Boundary Condition Matrices

Assemble the boundary condition matrices for an elliptic PDE.

The PDE is Poisson's equation,

$$-\nabla \cdot \nabla u = 1.$$

Partial Differential Equation Toolbox™ solves equations of the form

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

So, represent Poisson's equation in toolbox syntax by setting $c = 1$, $a = 0$, and $f = 1$.

```
c = 1;
a = 0;
f = 1;
```

Create a PDE model container. Import the `ForearmLink.stl` file into the model and examine the geometry.

```
model = createpde;
importGeometry(model,'Block.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
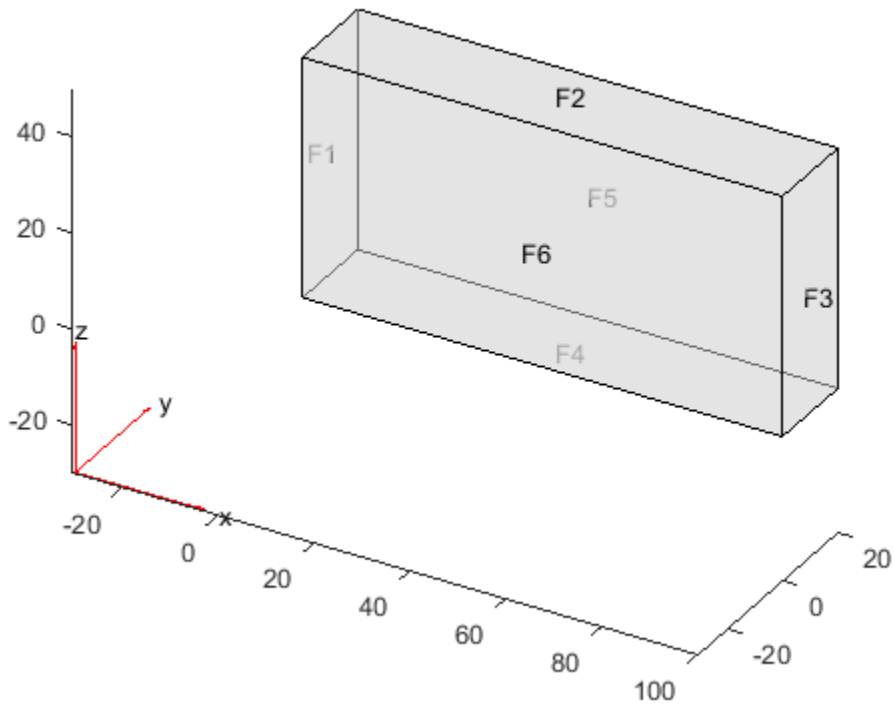
Set zero Dirichlet boundary conditions on the narrow faces (numbered 1 through 4).

```
applyBoundaryCondition(model,'Face',1:4,'u',0);
```

Set a Neumann condition with $g = -1$ on face 6, and $g = 1$ on face 5.

```
applyBoundaryCondition(model,'Face',6,'g',-1);
applyBoundaryCondition(model,'Face',5,'g',1);
```

Create a mesh for the model.

```
generateMesh(model);
```

Create the boundary condition matrices for the model.

```
[Q,G,H,R] = assemb(model);
```

The H matrix is quite sparse. The Q matrix has no nonzero entries.

```
disp(['Fraction of nonzero entries in H is ',num2str(nnz(H)/numel(H))])
disp(['Number of nonzero entries in Q is ',num2str(nnz(Q))])
```

```
Fraction of nonzero entries in H is 3.4709e-05
Number of nonzero entries in Q is 0
```

Assemble Boundary Matrices Using [p,e,t] Mesh

Assemble boundary condition matrices for the 2-D L-shaped region with Dirichlet boundary conditions, using the [p,e,t] mesh representation.

Define the geometry and boundary conditions using functions included in your software.

```
g = @lshapeg;
b = @lshapeb;
```

Create a mesh for the geometry.

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

Create the boundary matrices.

```
[Q,G,H,R] = assemb(b,p,e);
```

Only one of the resulting matrices is nonzero, namely H. The H matrix is quite sparse.

```
disp(['Fraction of nonzero entries in H is ',num2str(nnz(H)/numel(H))])
```

```
Fraction of nonzero entries in H is 0.0066667
```

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

b — Boundary conditions
boundary matrix | boundary file

Boundary conditions, specified as a boundary matrix or boundary file. Pass a boundary file as a function handle or as a string naming the file.

- A boundary matrix is generally an export from the PDE app. For details of the structure of this matrix, see "Boundary Matrix for 2-D Geometry" on page 2-171.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-199.

For more information on boundary conditions, see "Forms of Boundary Condition Specification" on page 2-170.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

p — Mesh nodes
output of `initmesh` | output of `meshToPet`

Mesh nodes, specified as the output of `initmesh` or `meshToPet`. For the structure of a `p` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

e — Mesh edges
output of `initmesh` | output of `meshToPet`

Mesh edges, specified as the output of `initmesh` or `meshToPet`. For the structure of `e`, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

sdl — Subdomain labels
vector of positive integers

Subdomain labels, specified as a vector of positive integers. For 2-D geometry only. View the subdomain labels in your geometry using the command

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

Example: `sdl = [1,3:5];`

Data Types: `double`

Output Arguments

Q — Neumann boundary condition matrix
sparse matrix

Neumann boundary condition matrix, returned as a sparse matrix. See "Elliptic Equations" on page 5-2.

Typically, you use Q in a subsequent call to a solver such as `assempde` or `hyperbolic`.

G — Neumann boundary condition vector
sparse vector

Neumann boundary condition vector, returned as a sparse vector. See "Elliptic Equations" on page 5-2.

Typically, you use G in a subsequent call to a solver such as `assempde` or `hyperbolic`.

H — Dirichlet matrix
sparse matrix

Dirichlet matrix, returned as a sparse matrix. See "Algorithms" on page 6-31.

Typically, you use H in a subsequent call to `assempde`.

R — Dirichlet vector
sparse vector

Dirichlet vector, returned as a sparse vector. See "Algorithms" on page 6-31.

Typically, you use R in a subsequent call to `assempde`.

More About

Algorithms

As explained in "Elliptic Equations" on page 5-2, the finite element matrices and vectors correspond to the *reduced linear system* and are the following.

- Q is the integral of the q boundary condition against the basis functions.
- G is the integral of the g boundary condition against the basis functions.
- H is the Dirichlet condition matrix representing $hu = r$.
- R is the Dirichlet condition vector for Hu = R.

For more information on the reduced linear system form of the finite element matrices, see the assempde "Definitions" on page 6-56 section, and the linear algebra approach detailed in "Systems of PDEs" on page 5-13.

See Also
assembleFEMatrices

Introduced before R2006a

assembleFEMatrices

Assemble finite element matrices

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW. For the corresponding step in the legacy workflow, see `assempde`, `assema`, or `assemb` for matrix assembly.

Syntax

```
FEM = assembleFEMatrices(model)
FEM = assembleFEMatrices(model,bcmethod)
```

Description

`FEM = assembleFEMatrices(model)` returns a structure containing finite element matrices for the PDE problem in `model`.

`FEM = assembleFEMatrices(model,bcmethod)` assembles finite element matrices and imposes boundary conditions using the method specified by `bcmethod`.

Examples

Assemble Finite Element Matrices for a 2-D Problem

Create a `PDEModel` for the Poisson equation on the L-shaped membrane with zero Dirichlet boundary conditions.

```
model = createpde(1);
geometryFromEdges(model,@lshapeg);
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',1);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
```

Generate a mesh and obtain the default finite element matrices for the problem and mesh.

```
generateMesh(model,'Hmax',0.2);
FEM = assembleFEMatrices(model)

FEM =

    K: [150x150 double]
    A: [150x150 double]
    F: [150x1 double]
    Q: [150x150 double]
    G: [150x1 double]
    H: [40x150 double]
    R: [40x1 double]
    M: [150x150 double]
```

Assemble Finite Element Matrices for a 2-D Problem With Boundary Conditions Imposed Using 'nullspace' Method and Obtain PDE Solution

Create a PDEModel for the Poisson equation on the L-shaped membrane with zero Dirichlet boundary conditions.

```
model = createpde(1);
geometryFromEdges(model,@lshapeg);
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',1);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
```

Generate a mesh and obtain the 'nullspace' finite element matrices for the problem and mesh.

```
generateMesh(model,'Hmax',0.2);
FEM = assembleFEMatrices(model,'nullspace')

FEM =

    Kc: [110x110 double]
    Fc: [110x1 double]
     B: [150x110 double]
    ud: [150x1 double]
     M: [110x110 double]
```

Obtain the solution to the PDE.

```
u = FEM.B*(FEM.Kc\FEM.Fc) + FEM.ud;
```

Compare to the solution using `solvepde`. The two solutions are identical:

```
u1 = solvepde(model);
norm(u - u1.NodalSolution)

ans =

     0
```

Input Arguments

`model` — PDE model
`PDEModel` object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde(1)`

`bcmethod` — Method for including boundary conditions
`'none'` (default) | `'nullspace'` | `'stiff-spring'`

Method for including boundary conditions, specified as `'none'`, `'nullspace'`, or `'stiff-spring'`. For the meaning and use of `bcmethod`, see "Algorithms" on page 6-36.

Example: `FEM = assembleFEMatrices(model,'nullspace')`

Data Types: `char`

Output Arguments

`FEM` — Finite element matrices
structure

Finite element matrices, returned as a structure. The fields in the structure depend on `bcmethod`. (These fields correspond to the legacy `assempde` outputs with the same names, except that there are now both A and M matrices.)

- `'none'` or no `bcmethod` given — The fields are K, A, F, Q, G, H, R, M.
- `'nullspace'` — The fields are Kc, Fc, B, ud, M.

- `'stiff-spring'` — The fields are Ks, Fs, M.

For the meaning and use of FEM, see "Algorithms" on page 6-36.

More About

Tips

- The mass matrix M is nonzero when the model is time-dependent. By using this matrix, you can solve a model with Raleigh damping. See "Dynamics of a Damped Cantilever Beam".

Algorithms

As explained in "Elliptic Equations" on page 5-2, the full finite element matrices and vectors are the following.

- K is the stiffness matrix, the integral of the c coefficient against the basis functions.
- M is the mass matrix, the integral of the m or d coefficient against the basis functions.
- A is the integral of the a coefficient against the basis functions.
- F is the integral of the f coefficient against the basis functions.
- Q is the integral of the q boundary condition against the basis functions.
- G is the integral of the g boundary condition against the basis functions.
- The H and R matrices come directly from the Dirichlet conditions and the mesh. See "Systems of PDEs" on page 5-13.

Given these matrices, the `'nullspace'` technique generates the combined finite element matrices [Kc,Fc,B,ud] as follows. The combined stiffness matrix is for the reduced linear system, Kc = K + M + Q. The corresponding combined load vector is Fc = F + G. The B matrix spans the null space of the columns of H (the Dirichlet condition matrix representing $hu = r$). The R vector represents the Dirichlet conditions in Hu = R. The ud vector represents boundary condition solutions for the Dirichlet conditions.

From the `'nullspace'` matrices, you can compute the solution u as
u = B*(Kc\Fc) + ud.

See "Systems of PDEs" on page 5-13 for details on the `'nullspace'` approach used to eliminate Dirichlet conditions.

> **Note:** Internally, for time-independent problems, `solvepde` uses the `'nullspace'` technique, and calculates solutions using `u = B*(Kc\Fc) + ud`.

Alternatively, the `'stiff-spring'` technique returns a matrix `Ks` and a vector `Fs` that together represent a different type of combined finite element matrices. The approximate solution `u` is `u = Ks\Fs`.

The `'stiff-spring'` technique generates matrices more quickly than the `'nullspace'` technique, but the `'stiff-spring'` technique generally gives less accurate solutions. For details of the stiff-spring approximation, see "Elliptic Equations" on page 5-2 and "Systems of PDEs" on page 5-13.

- "PDE Problem Setup"
- "Elliptic Equations" on page 5-2
- "Finite Element Basis for 3-D" on page 5-10
- "Systems of PDEs" on page 5-13

See Also
`solvepde`

Introduced in R2016a

assempde

Assemble finite element matrices and solve elliptic PDE

Compatibility

assempde IS NOT RECOMMENDED. Use `solvepde` instead.

Syntax

```
u = assempde(model,c,a,f)
u = assempde(b,p,e,t,c,a,f)

[Kc,Fc,B,ud] = assempde( ___ )
[Ks,Fs] = assempde( ___ )

[K,M,F,Q,G,H,R] = assempde( ___ )
[K,M,F,Q,G,H,R] = assempde( ___ ,[],sdl)

u = assempde(K,M,F,Q,G,H,R)
[Ks,Fs] = assempde(K,M,F,Q,G,H,R)
[Kc,Fc,B,ud] = assempde(K,M,F,Q,G,H,R)
```

Description

`u = assempde(model,c,a,f)` solves the PDE

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

with geometry, boundary conditions, and finite element mesh in `model`, and coefficients c, a, and f. If the PDE is a system of equations (`model.PDESystemSize` > 1), then `assempde` solves the system of equations

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

`u = assempde(b,p,e,t,c,a,f)` solves the PDE with boundary conditions `b`, and finite element mesh (`p`,`e`,`t`).

`[Kc,Fc,B,ud] = assempde( ___ )`, for any of the previous input syntaxes, assembles finite element matrices using the *reduced linear system* form, which eliminates any Dirichlet boundary conditions from the system of linear equations. You can calculate the solution `u` at node points by the command `u = B*(Kc\Fc) + ud`. See "Reduced Linear System" on page 6-56.

`[Ks,Fs] = assempde( ___ )` assembles finite element matrices that represent any Dirichlet boundary conditions using a *stiff-spring* approximation. You can calculate the solution `u` at node points by the command `u = Ks\Fs`. See "Stiff-Spring Approximation" on page 6-56.

`[K,M,F,Q,G,H,R] = assempde( ___ )` assembles finite element matrices that represent the PDE problem. This syntax returns all the matrices involved in converting the problem to finite element form. See "Algorithms" on page 6-56.

`[K,M,F,Q,G,H,R] = assempde( ___ ,[],sdl)` restricts the finite element matrices to those that include the subdomain specified by the subdomain labels in `sdl`. The empty argument is required in this syntax for historic and compatibility reasons.

`u = assempde(K,M,F,Q,G,H,R)` returns the solution `u` based on the full collection of finite element matrices.

`[Ks,Fs] = assempde(K,M,F,Q,G,H,R)` returns finite element matrices that approximate Dirichlet boundary conditions using the stiff-spring approximation. See "Algorithms" on page 6-56.

`[Kc,Fc,B,ud] = assempde(K,M,F,Q,G,H,R)` returns finite element matrices that eliminate any Dirichlet boundary conditions from the system of linear equations. See "Algorithms" on page 6-56.

Examples

Solve a Scalar PDE

Solve an elliptic PDE on an L-shaped region.

Create a scalar PDE model. Incorporate the geometry of an L-shaped region.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
```

Apply zero Dirichlet boundary conditions to all edges.

```
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
```

Generate a finite element mesh.

```
generateMesh(model);
```

Solve the PDE

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

with parameters $c = 1$, $a = 0$, and $f = 5$.

```
c = 1;
a = 0;
f = 5;
u = assempde(model,c,a,f);
```

Plot the solution.

```
pdeplot(model,'xydata',u)
```
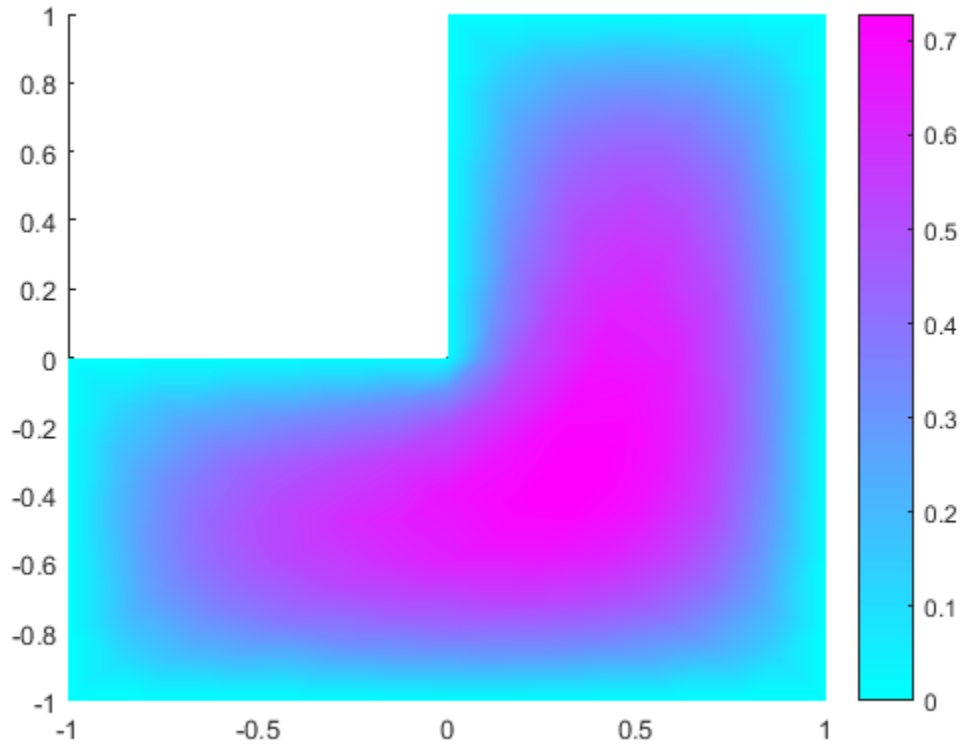
3-D Elliptic Problem

Solve a 3-D elliptic PDE using a PDE model.

Create a PDE model container, import a 3-D geometry description, and view the geometry.

```
model = createpde;
importGeometry(model,'Block.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
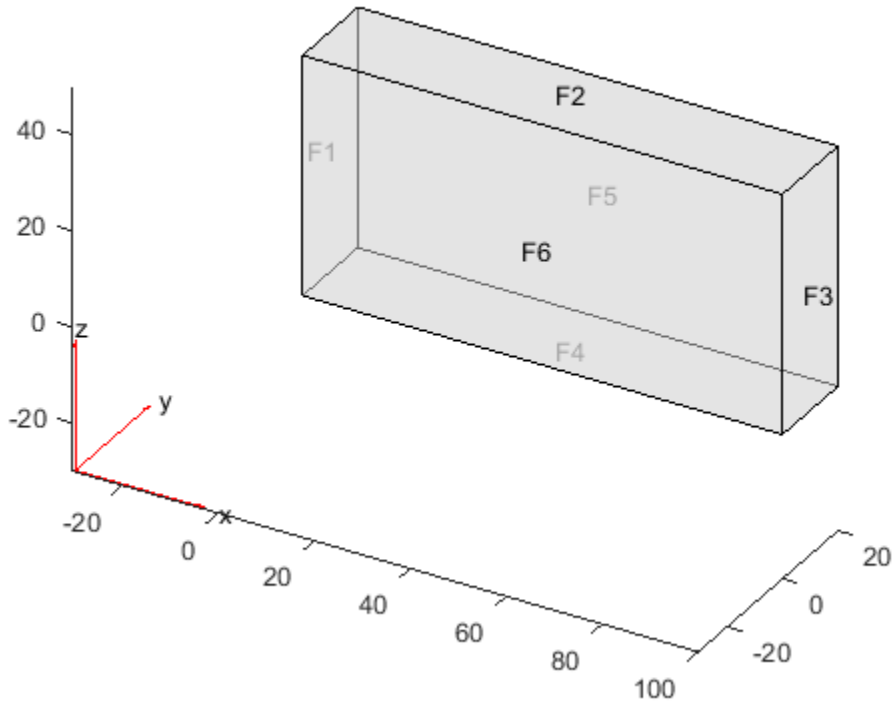
Set zero Dirichlet conditions on faces 1 through 4 (the edges). Set Neumann conditions with $g = -1$ on face 6 and $g = 1$ on face 5.

```
applyBoundaryCondition(model,'face',1:4,'u',0);
applyBoundaryCondition(model,'face',6,'g',-1);
applyBoundaryCondition(model,'face',5,'g',1);
```

Set coefficients $c = 1$, $a = 0$, and $f = 0.1$.

```
c = 1;
a = 0;
f = 0.1;
```

Create a mesh and solve the problem.

```
generateMesh(model);
u = assempde(model,c,a,f);
```

Plot the solution on the surface.

```
pdeplot3D(model,'colormapdata',u);
```
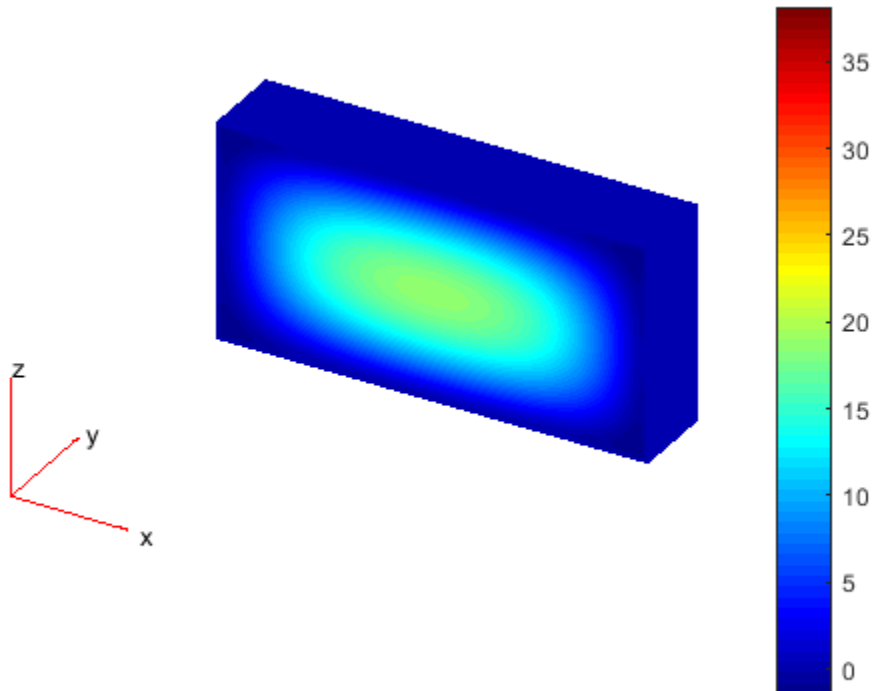

2-D PDE Using [p,e,t] Mesh

Solve a 2-D PDE using the older syntax for mesh.

Create a circle geometry.

```
g = @circleg;
```

Set zero Dirichlet boundary conditions.

```
b = @circleb1;
```

Create a mesh for the geometry.

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

Solve the PDE

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

with parameters $c = 1$, $a = 0$, and $f = \sin(x)$.

```
c = 1;
a = 0;
f = 'sin(x)';
u = assempde(b,p,e,t,c,a,f);
```

Plot the solution.

```
pdeplot(p,e,t,'xydata',u)
```
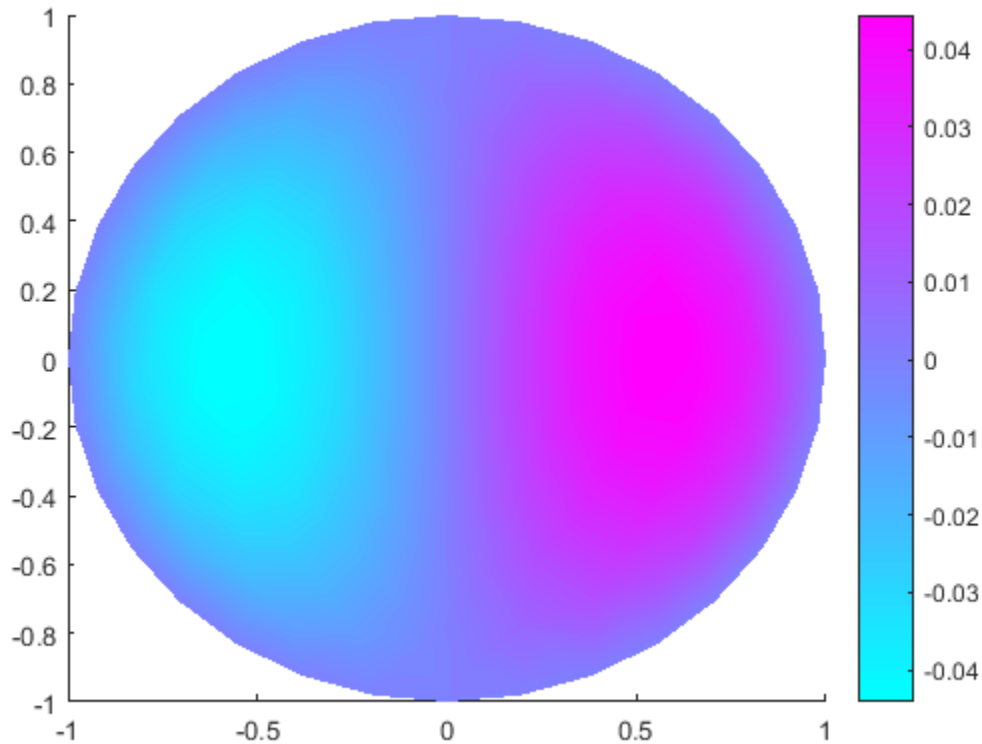
Finite Element Matrices

Obtain the finite-element matrices that represent the problem using a reduced linear algebra representation of Dirichlet boundary conditions.

Create a scalar PDE model. Import a simple 3-D geometry.

```
model = createpde;
importGeometry(model,'Block.stl');
```

Set zero Dirichlet boundary conditions on all the geometry faces.

```
applyBoundaryCondition(model,'face',1:model.Geometry.NumFaces,'u',0);
```

Generate a mesh for the geometry.

```
generateMesh(model);
```

Obtain finite element matrices K, F, B, and ud that represent the equation

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

with parameters $c = 1$, $a = 0$, and $f = \log(1 + x + y/(1+z))$.

```
c = 1;
a = 0;
f = 'log(1+x+y./(1+z))';
[K,F,B,ud] = assempde(model,c,a,f);
```

You can obtain the solution u of the PDE at mesh nodes by executing the command

```
u = B*(K\F) + ud;
```

Generally, this solution is slightly more accurate than the stiff-spring solution, as calculated in "Stiff-Spring Finite Element Solution" on page 6-46.

Stiff-Spring Finite Element Solution

Obtain the stiff-spring approximation of finite element matrices.

Create a scalar PDE model. Import a simple 3-D geometry.

```
model = createpde;
importGeometry(model,'Block.stl');
```

Set zero Dirichlet boundary conditions on all the geometry faces.

```
applyBoundaryCondition(model,'face',1:model.Geometry.NumFaces,'u',0);
```

Generate a mesh for the geometry.

```
generateMesh(model);
```

Obtain finite element matrices Ks and Fs that represent the equation

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

with parameters $c = 1$, $a = 0$, and $f = \log(1 + x + y/(1+z))$.

```
c = 1;
a = 0;
f = 'log(1+x+y./(1+z))';
[Ks,Fs] = assempde(model,c,a,f);
```

You can obtain the solution u of the PDE at mesh nodes by executing the command

```
u = Ks\Fs;
```

Generally, this solution is slightly less accurate than the reduced linear algebra solution, as calculated in "Finite Element Matrices" on page 6-45.

Full Collection of Finite Element Matrices

Obtain the full collection of finite element matrices for an elliptic problem.

Import geometry and set up an elliptic problem with Dirichlet boundary conditions. The 'Torus.stl' geometry has only one face, so you need set only one boundary condition.

```
model = createpde();
importGeometry(model,'Torus.stl');
applyBoundaryCondition(model,'face',1,'u',0);
c = 1;
a = 0;
f = 1;
generateMesh(model);
```

Create the finite element matrices that represent this problem.

```
[K,M,F,Q,G,H,R] = assempde(model,c,a,f);
```

Most of the resulting matrices are quite sparse. G, M, Q, and R are all zero sparse matrices.

```
howsparse = @(x)nnz(x)/numel(x);
disp(['Maximum fraction of nonzero entries in K or H is ',...
       num2str(max(howsparse(K),howsparse(H)))])
```

```
Maximum fraction of nonzero entries in K or H is 0.00086641
```

To find the solution to the PDE, call assempde again.

```
u = assempde(K,M,F,Q,G,H,R);
```

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

`c` — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. c represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify c in various ways, detailed in "c Coefficient for Systems" on page 2-125. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: 'cosh(x+y.^2)'

Data Types: double | char | function_handle
Complex Number Support: Yes

`a` — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. a represents the *a* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify a in various ways, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

f — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. `f` represents the *f* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify f in various ways, detailed in "f Coefficient for Systems" on page 2-98. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

b — Boundary conditions
boundary matrix | boundary file

Boundary conditions, specified as a boundary matrix or boundary file. Pass a boundary file as a function handle or as a string naming the file.

- A boundary matrix is generally an export from the PDE app. For details of the structure of this matrix, see "Boundary Matrix for 2-D Geometry" on page 2-171.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-199.

For more information on boundary conditions, see "Forms of Boundary Condition Specification" on page 2-170.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

p — Mesh nodes
output of `initmesh` | output of `meshToPet`

Mesh nodes, specified as the output of `initmesh` or `meshToPet`. For the structure of a `p` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

e — Mesh edges
output of `initmesh` | output of `meshToPet`

Mesh edges, specified as the output of `initmesh` or `meshToPet`. For the structure of `e`, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

t — Mesh elements
output of `initmesh` | output of `meshToPet`

Mesh elements, specified as the output of `initmesh` or `meshToPet`. Mesh elements are the triangles or tetrahedra that form the finite element mesh. For the structure of a `t` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

K — Stiffness matrix
sparse matrix | full matrix

Stiffness matrix, specified as a sparse matrix or full matrix. Generally, you obtain K from a previous call to `assema` or `assempde`. For the meaning of stiffness matrix, see "Elliptic Equations" on page 5-2.

Example: `[K,M,F,Q,G,H,R] = assempde(model,c,a,f)`

Data Types: `double`
Complex Number Support: Yes

M — Mass matrix
sparse matrix | full matrix

Mass matrix, specified as a sparse matrix or full matrix. Generally, you obtain M from a previous call to `assema` or `assempde`. For the meaning of mass matrix, see "Elliptic Equations" on page 5-2.

Example: `[K,M,F,Q,G,H,R] = assempde(model,c,a,f)`

Data Types: `double`
Complex Number Support: Yes

F — Finite element f representation
vector

Finite element `f` representation, specified as a vector. Generally, you obtain F from a previous call to `assema` or `assempde`. For the meaning of this representation, see "Elliptic Equations" on page 5-2.

Example: `[K,M,F,Q,G,H,R] = assempde(model,c,a,f)`

Data Types: `double`
Complex Number Support: Yes

Q — Neumann boundary condition matrix
sparse matrix | full matrix

Neumann boundary condition matrix, specified as a sparse matrix or full matrix. Generally, you obtain Q from a previous call to `assemb` or `assempde`. For the meaning of this matrix, see "Elliptic Equations" on page 5-2.

Example: `[K,M,F,Q,G,H,R] = assempde(model,c,a,f)`

Data Types: `double`
Complex Number Support: Yes

`G` — Neumann boundary condition vector
sparse vector | full vector

Neumann boundary condition vector, specified as a sparse vector or full vector. Generally, you obtain `G` from a previous call to `assemb` or `assempde`. For the meaning of this vector, see "Elliptic Equations" on page 5-2.

Example: `[K,M,F,Q,G,H,R] = assempde(model,c,a,f)`

Data Types: `double`
Complex Number Support: Yes

`H` — Dirichlet boundary condition matrix
sparse matrix | full matrix

Dirichlet boundary condition matrix, specified as a sparse matrix or full matrix. Generally, you obtain `H` from a previous call to `assemb` or `assempde`. For the meaning of this matrix, see "Algorithms" on page 6-56.

Example: `[K,M,F,Q,G,H,R] = assempde(model,c,a,f)`

Data Types: `double`
Complex Number Support: Yes

`R` — Dirichlet boundary condition vector
sparse vector | full vector

Dirichlet boundary condition vector, specified as a sparse vector or full vector. Generally, you obtain `R` from a previous call to `assemb` or `assempde`. For the meaning of this vector, see "Algorithms" on page 6-56.

Example: `[K,M,F,Q,G,H,R] = assempde(model,c,a,f)`

Data Types: `double`
Complex Number Support: Yes

`sdl` — Subdomain labels
vector of positive integers

Subdomain labels, specified as a vector of positive integers. For 2-D geometry only. View the subdomain labels in your geometry using the command

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

Example: `sdl = [1,3:5];`

Data Types: `double`

Output Arguments

u — PDE solution
vector

PDE solution, returned as a vector.

- If the PDE is scalar, meaning only one equation, then `u` is a column vector representing the solution *u* at each node in the mesh. `u(i)` is the solution at the `ith` column of `model.Mesh.Nodes` or the `ith` column of `p`.
- If the PDE is a system of $N > 1$ equations, then `u` is a column vector with N*`Np` elements, where `Np` is the number of nodes in the mesh. The first `Np` elements of `u` represent the solution of equation 1, then next `Np` elements represent the solution of equation 2, etc.

To obtain the solution at an arbitrary point in the geometry, use `pdeInterpolant`.

To plot the solution, use `pdeplot` for 2-D geometry, or see "Plot 3-D Solutions and Their Gradients" on page 3-145.

Kc — Stiffness matrix
sparse matrix

Stiffness matrix, returned as a sparse matrix. See "Elliptic Equations" on page 5-2.

`u1 = Kc\Fc` returns the solution on the non-Dirichlet points. To obtain the solution `u` at the nodes of the mesh,

```
u = B*(Kc\Fc) + ud
```

Generally, `Kc`, `Fc`, `B`, and `ud` make a slower but more accurate solution than `Ks` and `Fs`.

Fc — Load vector
vector

Load vector, returned as a vector. See "Elliptic Equations" on page 5-2.

```
u = B*(Kc\Fc) + ud
```

Generally, Kc, Fc, B, and ud make a slower but more accurate solution than Ks and Fs.

B — Dirichlet nullspace
sparse matrix

Dirichlet nullspace, returned as a sparse matrix. See "Algorithms" on page 6-56.

```
u = B*(Kc\Fc) + ud
```

Generally, Kc, Fc, B, and ud make a slower but more accurate solution than Ks and Fs.

ud — Dirichlet vector
vector

Dirichlet vector, returned as a vector. See "Algorithms" on page 6-56.

```
u = B*(Kc\Fc) + ud
```

Generally, Kc, Fc, B, and ud make a slower but more accurate solution than Ks and Fs.

Ks — Stiffness matrix corresponding to the stiff-spring approximation for Dirichlet boundary condition
sparse matrix

Finite element matrix for stiff-spring approximation, returned as a sparse matrix. See "Algorithms" on page 6-56.

To obtain the solution u at the nodes of the mesh,
u = Ks\Fs.

Generally, Ks and Fs make a quicker but less accurate solution than Kc, Fc, B, and ud.

Fs — Load vector corresponding to the stiff-spring approximation for Dirichlet boundary condition
vector

Load vector corresponding to the stiff-spring approximation for Dirichlet boundary condition, returned as a vector. See "Algorithms" on page 6-56.

To obtain the solution u at the nodes of the mesh,
u = Ks\Fs.

Generally, `Ks` and `Fs` make a quicker but less accurate solution than `Kc`, `Fc`, `B`, and `ud`.

K — Stiffness matrix
sparse matrix

Stiffness matrix, returned as a sparse matrix. See "Elliptic Equations" on page 5-2.

K represents the stiffness matrix alone, unlike Kc or Ks, which are stiffness matrices combined with other terms to enable immediate solution of a PDE.

Typically, you use `K` in a subsequent call to a solver such as `assempde` or `hyperbolic`.

M — Mass matrix
sparse matrix

Mass matrix. returned as a sparse matrix. See "Elliptic Equations" on page 5-2.

Typically, you use `M` in a subsequent call to a solver such as `assempde` or `hyperbolic`.

F — Load vector
vector

Load vector, returned as a vector. See "Elliptic Equations" on page 5-2.

F represents the load vector alone, unlike Fc or Fs, which are load vectors combined with other terms to enable immediate solution of a PDE.

Typically, you use `F` in a subsequent call to a solver such as `assempde` or `hyperbolic`.

Q — Neumann boundary condition matrix
sparse matrix

Neumann boundary condition matrix, returned as a sparse matrix. See "Elliptic Equations" on page 5-2.

Typically, you use `Q` in a subsequent call to a solver such as `assempde` or `hyperbolic`.

G — Neumann boundary condition vector
sparse vector

Neumann boundary condition vector, returned as a sparse vector. See "Elliptic Equations" on page 5-2.

Typically, you use `G` in a subsequent call to a solver such as `assempde` or `hyperbolic`.

H — Dirichlet matrix
sparse matrix

Dirichlet matrix, returned as a sparse matrix. See "Algorithms" on page 6-56.

Typically, you use H in a subsequent call to a solver such as `assempde` or `hyperbolic`.

R — Dirichlet vector
sparse vector

Dirichlet vector, returned as a sparse vector. See "Algorithms" on page 6-56.

Typically, you use R in a subsequent call to a solver such as `assempde` or `hyperbolic`.

More About

Reduced Linear System

This form of the finite element matrices eliminates Dirichlet conditions from the problem using a linear algebra approach. The finite element matrices reduce to the solution u = B*(Kc\Fc) + ud, where B spans the null space of the columns of H (the Dirichlet condition matrix representing $hu = r$). R is the Dirichlet condition vector for Hu = R. ud is the vector of boundary condition solutions for the Dirichlet conditions. u1 = Kc\Fc returns the solution on the non-Dirichlet points.

See "Systems of PDEs" on page 5-13 for details on the approach used to eliminate Dirichlet conditions.

Stiff-Spring Approximation

This form of the finite element matrices converts Dirichlet boundary conditions to Neumann boundary conditions using a stiff-spring approximation. Using this approximation, `assempde` returns a matrix Ks and a vector Fs that represent the combined finite element matrices. The approximate solution u is u = Ks\Fs.

See "Elliptic Equations" on page 5-2. For details of the stiff-spring approximation, see "Systems of PDEs" on page 5-13.

Algorithms

`assempde` performs the following steps to obtain a solution u to an elliptic PDE:

1 Generate the finite element matrices [K,M,F,Q,G,H,R]. This step is equivalent to calling `assema` to generate the matrices `K`, `M`, and `F`, and also calling `assemb` to generate the matrices `Q`, `G`, `H`, and `R`.

2 Generate the combined finite element matrices [Kc,Fc,B,ud]. The combined stiffness matrix is for the reduced linear system, `Kc = K + M + Q`. The corresponding combined load vector is `Fc = F + G`. The `B` matrix spans the null space of the columns of `H` (the Dirichlet condition matrix representing $hu = r$). The `R` vector represents the Dirichlet conditions in `Hu = R`. The `ud` vector represents boundary condition solutions for the Dirichlet conditions.

3 Calculate the solution `u` via
`u = B*(Kc\Fc) + ud.`

`assempde` uses one of two algorithms for assembling a problem into combined finite element matrix form. A *reduced linear system* form leads to immediate solution via linear algebra. You choose the algorithm by the number of outputs. For the reduced linear system form, request four outputs:
`[Kc,Fc,B,ud] = assempde(_)`

For the *stiff-spring approximation*, request two outputs:
`[Ks,Fs] = assempde(_)`

For details, see "Reduced Linear System" on page 6-56 and "Stiff-Spring Approximation" on page 6-56.

As explained in "Elliptic Equations" on page 5-2, the full finite element matrices and vectors are the following.

- `K` is the stiffness matrix, the integral of the `c` coefficient against the basis functions.
- `M` is the mass matrix, the integral of the `a` coefficient against the basis functions.
- `F` is the integral of the `f` coefficient against the basis functions.
- `Q` is the integral of the `q` boundary condition against the basis functions.
- `G` is the integral of the `g` boundary condition against the basis functions.
- The `H` and `R` matrices come directly from the Dirichlet conditions and the mesh. See "Systems of PDEs" on page 5-13.

See Also
assembleFEMatrices | solvepde

Introduced before R2006a

BoundaryCondition Properties

Boundary condition for PDE model

Compatibility

THIS PAGE DESCRIBES BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Description

A `BoundaryCondition` object specifies one type of PDE boundary condition on a set of geometry boundaries. A `PDEModel` object contains a vector of `BoundaryCondition` objects in its `BoundaryConditions` property.

Specify boundary conditions for your model using the `applyBoundaryCondition` function.

Properties

`RegionType` — Boundary type
`'face'` for 3-D geometry | `'edge'` for 2-D geometry

Boundary type, specified as `'face'` for 3-D geometry or `'edge'` for 2-D geometry.

Example: `applyBoundaryCondition(model,'face',3,'u',0)`

Data Types: `char`

`RegionID` — Boundary ID
vector of positive integers

Boundary ID, specified as a vector of positive integers. To determine which ID corresponds to which portion of the geometry boundary, use the `pdegplot` function. Set the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) name-value pair set to `'on'`.

Example: `applyBoundaryCondition(model,'face',3:6,'u',0)`

Data Types: `double`

r — Dirichlet condition `h*u = r`
`zeros(N,1)` (default) | vector with N elements | function handle

Dirichlet condition `h*u = r`, specified as a vector with N elements or as a function handle. N is the number of PDEs in the system. For the syntax of the function handle form of `r`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'r',[0;4;-1]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

h — Dirichlet condition `h*u = r`
`eye(N)` (default) | N-by-N matrix | vector with N^2 elements | function handle

Dirichlet condition `h*u = r`, specified as an N-by-N matrix, as a vector with N^2 elements, or as a function handle. N is the number of PDEs in the system. For the syntax of the function handle form of `h`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'h',[2,1;1,2]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

g — Generalized Neumann condition `n·(c×∇u) + qu = g`
`zeros(N,1)` (default) | vector with N elements | function handle

Generalized Neumann condition `n·(c×∇u) + qu = g`, specified as a vector with N elements or as a function handle. N is the number of PDEs in the system. For the syntax of the function handle form of `g`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'g',[3;2;-1]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

q — Generalized Neumann condition `n·(c×∇u) + qu = g`
`zeros(N)` (default) | N-by-N matrix | vector with N^2 elements | function handle

Generalized Neumann condition `n·(c×∇u) + qu = g`, specified as an N-by-N matrix, as a vector with N^2 elements, or as a function handle. N is the number of PDEs in the system. For the syntax of the function handle form of `q`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `'q',eye(3)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`u` — Dirichlet conditions
`zeros(N,1)` (default) | vector of up to N elements | function handle

Dirichlet conditions, specified as a vector of up to N elements or as a function handle.
`EquationIndex` and `u` must have the same length. For the syntax of the function handle
form of `u`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `applyBoundaryCondition(model,'face',[2,4,11],'u',0)`

Data Types: `double`
Complex Number Support: Yes

`EquationIndex` — Index of specified u components
`1:N` (default) | vector of integers with entries from 1 to N

Index of specified `u` components, specified as a vector of integers with entries from 1 to N.
`EquationIndex` and `u` must have the same length.

Example: `applyBoundaryCondition(model,'face',[2,4,11],'u',`
`[3,-1],'EquationIndex',[2,3])`

Data Types: `double`

`Vectorized` — Vectorized function evaluation
`'off'` (default) | `'on'`

Vectorized function evaluation, specified as `'on'` or `'off'`. This evaluation applies when
you pass a function handle as an argument. To save time in function handle evaluation,
specify `'on'`, assuming that your function handle computes in a vectorized fashion.
See "Vectorization". For details of this evaluation, see "Specify Nonconstant Boundary
Conditions" on page 2-190.

Example: `applyBoundaryCondition(model,'face',`
`[2,4,11],'u',@ucalculator,'Vectorized','on')`

Data Types: `char`

See Also
`applyBoundaryCondition` | `PDEModel`

Related Examples

- "Solve PDEs with Constant Boundary Conditions" on page 2-185
- "Solve PDEs with Nonconstant Boundary Conditions" on page 2-193

More About

- "Solve Problems Using PDEModel Objects" on page 2-14
- "Specify Constant Boundary Conditions" on page 2-181
- "Specify Nonconstant Boundary Conditions" on page 2-190

Introduced in R2015a

CoefficientAssignment Properties

Coefficient assignments

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.

Description

A `CoefficientAssignment` object contains a description of the PDE coefficients. A `PDEModel` container has a vector of `CoefficientAssignment` objects in its `EquationCoefficients.CoefficientAssignments` property.

Coefficients are the m, d, c, a, and f variables in the PDE

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

or the eigenvalue problem

$$-\nabla\cdot(c\nabla u) + au = \lambda du$$
or
$$-\nabla\cdot(c\nabla u) + au = \lambda^2 mu.$$

Create coefficients for your model using the `specifyCoefficients` function.

Properties

`RegionType` — Region type
`'face'` | `'cell'`

Region type, returned as `'face'` for a 2-D region, or `'cell'` for a 3-D region.

Data Types: `char`

`RegionID` — Region ID
vector of positive integers

Region ID, returned as a vector of positive integers. To determine which ID corresponds to which portion of the geometry, use the `pdegplot` function. Set the `'SubdomainLabels'` name-value pair to `'on'`.

Data Types: `double`

`m` — Second-order time derivative coefficient
scalar | column vector | function handle

Second-order time derivative coefficient, returned as a scalar, column vector, or function handle. For details of the `m` coefficient specification, see "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`d` — First-order time derivative coefficient
scalar | column vector | function handle

First-order time derivative coefficient, returned as a scalar, column vector, or function handle. For details of the `d` coefficient specification, see "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`c` — Second-order space derivative coefficient
scalar | column vector | function handle

Second-order space derivative coefficient, returned as a scalar, column vector, or function handle. For details of the `c` coefficient specification, see "c Coefficient for `specifyCoefficients`" on page 2-104.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`a` — Solution multiplier coefficient
scalar | column vector | function handle

Solution multiplier coefficient, returned as a scalar, column vector, or function handle. For details of the a coefficient specification, see "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

f — Source coefficient
scalar | column vector | function handle

Source coefficient, returned as a scalar, column vector, or function handle. For details of the f coefficient specification, see "f Coefficient for `specifyCoefficients`" on page 2-101.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

See Also
`findCoefficients` | `specifyCoefficients`

More About
- "PDE Coefficients"
- "Solve Problems Using PDEModel Objects" on page 2-14

Introduced in R2016a

createpde

Create PDE model

Compatibility

CREATING PDE MODELS IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
model = createpde(N)
model = createpde
```

Description

`model = createpde(N)` returns a PDE model container for a system of N equations.

`model = createpde` returns a PDE model for one equation, meaning a scalar PDE.

Examples

Create a PDE Model

Create a model for a system of three equations.

```
model = createpde(3)

model =

  PDEModel with properties:

          PDESystemSize: 3
        IsTimeDependent: O
               Geometry: []
    EquationCoefficients: []
```

```
        BoundaryConditions: [0x0 BoundaryCondition]
          InitialConditions: []
                       Mesh: []
              SolverOptions: [1x1 PDESolverOptions]
```

Create a Scalar PDE Model

Create a model for a single (scalar) PDE.

```
model = createpde

model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: 0
                 Geometry: []
      EquationCoefficients: []
        BoundaryConditions: [0x0 BoundaryCondition]
          InitialConditions: []
                       Mesh: []
              SolverOptions: [1x1 PDESolverOptions]
```

• "Solve Problems Using PDEModel Objects" on page 2-14

Input Arguments

N — Number of equations
1 (default) | positive integer

Number of equations, specified as a positive integer.

Example: model = createpde(3);

Data Types: double

Output Arguments

model — PDE model container
PDEModel object

PDE model container, returned as a `PDEModel` object.

See Also
`PDEModel`

Introduced in R2015a

createPDEResults

Create solution object

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `solvepde` and `solvepdeeig`.

The original (R2015b) version of `createPDEResults` had only one syntax, and created a `PDEResults` object. Beginning with R2016a, you generally do not need to use `createPDEResults`, because the `solvepde` and `solvepdeeig` functions return solution objects. Furthermore, `createPDEResults` returns an object of a newer type than `PDEResults`. If you open an existing `PDEResults` object, it is converted to a `StationaryResults` object.

If you use one of the older solvers such as `adaptmesh`, then you can use `createPDEResults` to obtain a solution object. Stationary and time-dependent solution objects have gradients available, whereas `PDEResults` did not include gradients.

Syntax

```
results = createPDEResults(model,u)
results = createPDEResults(model,u,'stationary')
results = createPDEResults(model,u,utimes,'time-dependent')
results = createPDEResults(model,eigenvectors,eigenvalues,'eigen')
```

Description

`results = createPDEResults(model,u)` creates a `StationaryResults` solution object from `model` and its solution `u`.

This syntax is equivalent to `results = createPDEResults(model, u,'stationary')`.

results = createPDEResults(model,u,utimes,'time-dependent') creates a TimeDependentResults solution object from model, its solution u, and the times utimes.

results = createPDEResults(model,eigenvectors,eigenvalues,'eigen') creates an EigenResults solution object from model, its eigenvector solution eigenvectors, and its eigenvalues eigenvalues.

Examples

Results From an Elliptic Problem

Create a StationaryResults object from the solution to an elliptic system.

Create a PDE model for a system of three equations. Import the geometry of a bracket and plot the face labels.

```
model = createpde(3);
importGeometry(model,'BracketWithHole.stl');
figure
h = pdegplot(model,'FaceLabels','on');
view(30,30);
title('Bracket with Face Labels')
h(1).FaceAlpha = 0.5;
```

Bracket with Face Labels
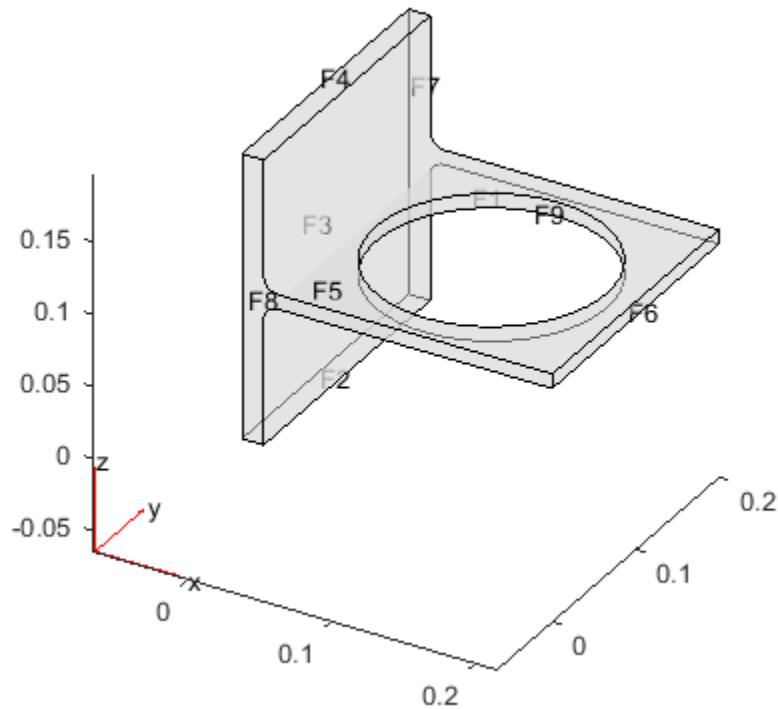
Set boundary conditions: face 3 is immobile, and there is a force in the negative z direction on face 6.

```
applyBoundaryCondition(model,'face',3,'u',[0,0,0]);
applyBoundaryCondition(model,'face',6,'g',[0,0,-1e4]);
```

Set coefficients that represent the equations of linear elasticity. See "3-D Linear Elasticity Equations in Toolbox Form" on page 3-35.

```
E = 200e9;
nu = 0.3;
c = elasticityC3D(E,nu);
a = 0;
f = [0;0;0];
```

Create a mesh and solve the problem.

```
generateMesh(model,'Hmax',1e-2);
u = assempde(model,c,a,f);
```

Create a `StationaryResults` object from the solution.

```
results = createPDEResults(model,u)

results =

  StationaryResults with properties:

    NodalSolution: [13437x3 double]
       XGradients: [13437x3 double]
       YGradients: [13437x3 double]
       ZGradients: [13437x3 double]
             Mesh: [1x1 FEMesh]
```

Plot the solution for the *z*-component, which is component 3.

```
pdeplot3D(model,'colormapdata',results.NodalSolution(:,3));
```

Results from a Time-Dependent Problem

Obtain a solution from a parabolic problem.

The problem models heat flow in a solid.

```
model = createpde();
importGeometry(model,'Tetrahedron.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
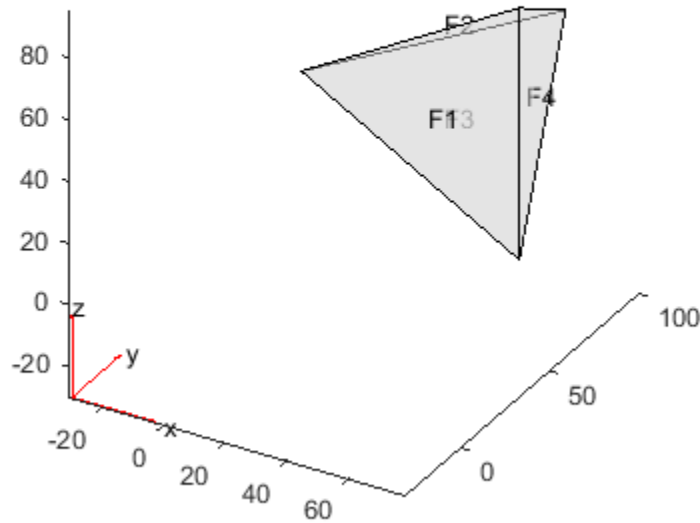
Set the temperature on face 2 to 100. Leave the other boundary conditions at their default values (insulating).

```
applyBoundaryCondition(model,'face',2,'u',100);
```

Set the coefficients to model a parabolic problem with 0 initial temperature.

```
d = 1;
c = 1;
a = 0;
f = 0;
u0 = 0;
```

Create a mesh and solve the PDE for times from 0 through 200 in steps of 10.

```
tlist = 0:10:200;
generateMesh(model);
u = parabolic(u0,tlist,model,c,a,f,d);

186 successful steps
0 failed attempts
348 function evaluations
1 partial derivatives
31 LU decompositions
347 solutions of linear systems
```

Create a `TimeDependentResults` object from the solution.

```
results = createPDEResults(model,u,tlist,'time-dependent');
```

Plot the solution on the surface of the geometry at time 100.

```
pdeplot3D(model,'colormapdata',results.NodalSolution(:,11))
```
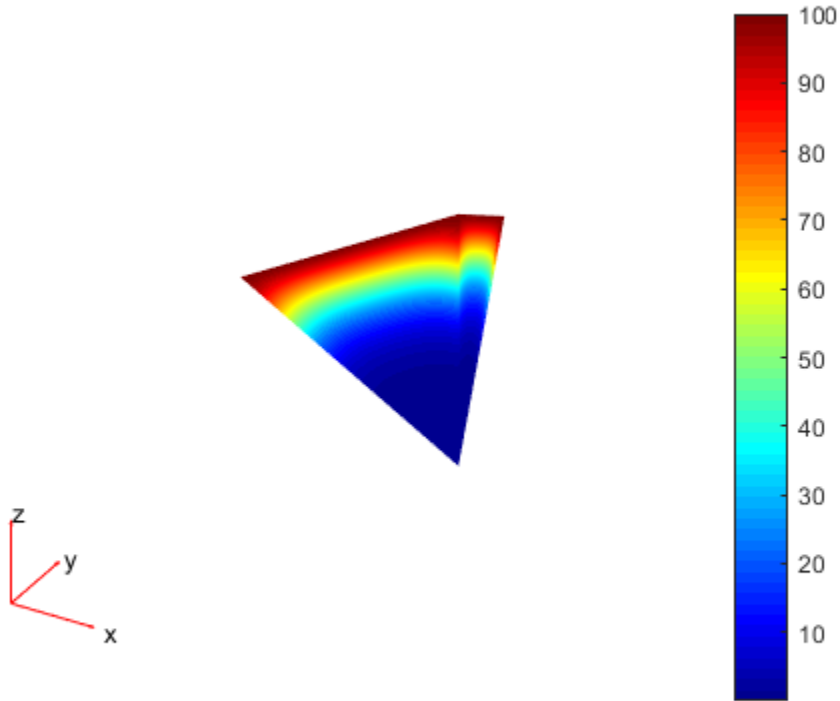
Results from an Eigenvalue Problem

Create an `EigenResults` object from the solution to an eigenvalue problem.

Create the geometry and mesh for the L-shaped membrane. Apply Dirichlet boundary conditions to all edges.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
generateMesh(model,'Hmax',0.05);
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',0);
```

Solve the eigenvalue problem for coefficients c = 1, a = 0, and d = 1. Obtain solutions for eigenvalues from 0 through 100.

```
c = 1;
a = 0;
d = 1;
r = [0,100];
[eigenvectors,eigenvalues] = pdeeig(model,c,a,d,r);
```

```
               Basis= 10,  Time=   0.12,  New conv eig=  0
               Basis= 13,  Time=   0.12,  New conv eig=  0
               Basis= 16,  Time=   0.12,  New conv eig=  0
               Basis= 19,  Time=   0.12,  New conv eig=  1
               Basis= 22,  Time=   0.31,  New conv eig=  2
               Basis= 25,  Time=   0.31,  New conv eig=  3
               Basis= 28,  Time=   0.31,  New conv eig=  3
               Basis= 31,  Time=   0.31,  New conv eig=  5
               Basis= 34,  Time=   0.53,  New conv eig=  5
               Basis= 37,  Time=   0.78,  New conv eig=  7
               Basis= 40,  Time=   0.78,  New conv eig=  7
               Basis= 43,  Time=   0.78,  New conv eig= 10
               Basis= 46,  Time=   1.01,  New conv eig= 10
               Basis= 49,  Time=   1.01,  New conv eig= 11
               Basis= 52,  Time=   1.19,  New conv eig= 13
               Basis= 55,  Time=   1.39,  New conv eig= 14
               Basis= 58,  Time=   1.39,  New conv eig= 14
               Basis= 61,  Time=   1.62,  New conv eig= 16
               Basis= 64,  Time=   1.83,  New conv eig= 16
               Basis= 67,  Time=   2.04,  New conv eig= 18
               Basis= 70,  Time=   2.28,  New conv eig= 21
End of sweep: Basis= 70,  Time=   2.28,  New conv eig= 21
               Basis= 31,  Time=   2.62,  New conv eig=  0
               Basis= 34,  Time=   2.79,  New conv eig=  0
               Basis= 37,  Time=   2.79,  New conv eig=  0
               Basis= 40,  Time=   2.89,  New conv eig=  0
End of sweep: Basis= 40,  Time=   2.89,  New conv eig=  0
```

Create an EigenResults object from the solution.

```
results = createPDEResults(model,eigenvectors,eigenvalues,'eigen')


results =
```

```
EigenResults with properties:

  Eigenvectors: [2124x19 double]
   Eigenvalues: [19x1 double]
          Mesh: [1x1 FEMesh]
```

Plot the solution for mode 10.

```
pdeplot(model,'xydata',results.Eigenvectors(:,10))
```
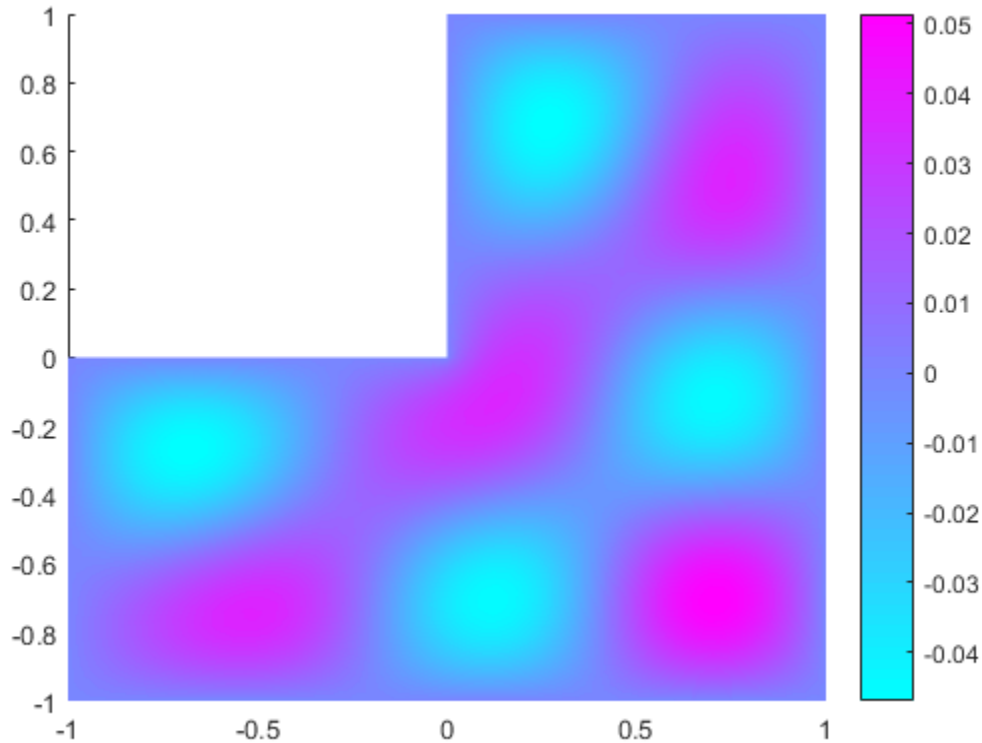
Input Arguments

`model` — PDE model
`PDEModel` object

PDE model, specified as a `PDEModel` object.

Example: model = createpde(1)

`u` — PDE solution
vector | matrix

PDE solution, specified as a vector or matrix.

Example: `u = assempde(model,c,a,f);`

`utimes` — Times for a PDE solution
monotone vector

Times for a PDE solution, specified as a monotone vector. These times should be the same as the `tlist` times that you specified for the solution by the `hyperbolic` or `parabolic` solvers.

Example: `utimes = 0:0.2:5;`

`eigenvectors` — Eigenvector solution
matrix

Eigenvector solution, specified as a matrix. Suppose

- `Np` is the number of mesh nodes
- `N` is the number of equations
- `ev` is the number of eigenvalues specified in `eigenvalues`

Then `eigenvectors` has size `Np`-by-`N`-by-`ev`. Each column of `eigenvectors` corresponds to the eigenvectors of one eigenvalue. In each column, the first `Np` elements correspond to the eigenvector of equation 1 evaluated at the mesh nodes, the next `Np` elements correspond to equation 2, and so on.

`eigenvalues` — Eigenvalue solution
vector

Eigenvalue solution, specified as a vector.

Output Arguments

`results` — PDE solution
`StationaryResults` object (default) | `TimeDependentResults` object | `EigenResults` object

PDE solution, specified as a `StationaryResults` object, a `TimeDependentResults` object, or an `EigenResults` object. Create `results` using `solvepde`, `solvepdeeig`, or `createPDEResults`.

Example: `results = solvepde(model)`

More About

Tips

- Dimensions of the returned solutions and gradients are the same as those returned by `solvepde` and `solvepdeeig`. For details, see "Dimensions of Solutions and Gradients" on page 3-167.

Algorithms

The procedure for evaluating gradients at nodal locations is as follows:

1 Calculate the gradients at the Gauss points located inside each element.

2 Extrapolate the gradients at the nodal locations.

3 Average the value of the gradient from all elements that meet at the nodal point. This step is needed because of the inter-element discontinuity of gradients. The elements that connect at the same nodal point give different extrapolated values of the gradient for the point. `createPDEResults` performs area-weighted averaging for 2-D meshes and volume-weighted averaging for 3-D meshes.
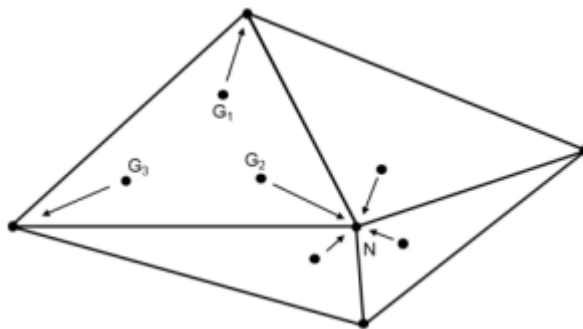
See Also

`EigenResults` | `evaluateGradient` | `interpolateSolution` | `StationaryResults` | `TimeDependentResults`

Introduced in R2015b

csgchk

Check validity of Geometry Description matrix

Compatibility

2-D GEOMETRY FUNCTIONS ARE THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

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 the PDE app.

`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

Delete borders between minimal regions

Compatibility

2-D GEOMETRY FUNCTIONS ARE THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

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

decsg

Decompose Constructive Solid Geometry into minimal regions

Compatibility

2-D GEOMETRY FUNCTIONS ARE THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

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 PDE app uses `decsg` for many purposes. Each time a new solid object is drawn or changed, the PDE app 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 non-intersecting 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 the PDE app. 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 the PDE app.

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 (in radians) 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 the PDE app 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.

Diagnostics

NaN is returned if the set formula sf cannot be evaluated.

More About

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.

See Also

csgchk | csgdel | pdecirc | pdeellip | pdepoly | pderect | pdetool | wgeom

DiscreteGeometry Properties

3-D geometry description

Compatibility

THIS PAGE DESCRIBES BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Description

`DiscreteGeometry` describes 3-D geometry in the form of an object. A `PDEModel` object has a `Geometry` property. For 3-D geometry, the `Geometry` property is a `DiscreteGeometry` object.

Specify a 3-D geometry for your model using the `importGeometry` function or the `geometryFromMesh` function.

Properties

`NumEdges` — Number of geometry edges
positive integer

Number of geometry edges, returned as a positive integer.

Data Types: `double`

`NumFaces` — Number of geometry faces
positive integer

Number of geometry faces, returned as a positive integer.

Data Types: `double`

`NumVertices` — Number of geometry vertices
positive integer

Number of geometry vertices, returned as a positive integer.

Data Types: `double`

See Also
`geometryFromMesh` | `importGeometry` | `PDEModel`

More About
- "Solve Problems Using PDEModel Objects" on page 2-14

Introduced in R2015a

dstidst

Discrete sine transform

Compatibility

dst and **idst** ARE NOT RECOMMENDED.

Syntax

```
y = dst(x)
y = dst(x,n)
x = idst(y)
x = idst(y,n)
```

Description

The `dst` function implements the following equation:

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

`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:

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

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.

Using EigenResults Objects

Contain PDE eigenvalue solution and derived quantities

Compatibility

An `EigenResults` OBJECT IS USED IN BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS. It replaces a `PDEResults` object returned in R2015b.

Description

An `EigenResults` object contains the solution of a PDE eigenvalue problem in a form convenient for plotting and postprocessing.

- Eigenvector values at the nodes appear in the `Eigenvectors` property.
- The eigenvalues appear in the `Eigenvalues` property.

Examples

Results from an Eigenvalue Problem

Obtain an `EigenResults` object from `solvepdeeig`.

Create the geometry for the L-shaped membrane. Apply zero Dirichlet boundary conditions to all edges.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge,1:model.Geometry.NumEdges,'u',0);
```

Create coefficients `c = 1`, `a = 0`, and `d = 1`.

```
specifyCoefficients(model,'m',0,...
                          'd',1,...
                          'c',1,...
```

```
                            'a',0,...
                            'f',0);
```

Create the mesh and solve the eigenvalue problem for eigenvalues from 0 through 100.

```
generateMesh(model,'Hmax',0.05);
ev = [0,100];
results = solvepdeeig(model,ev)

                Basis= 10,  Time=   0.02,  New conv eig=  0
                Basis= 13,  Time=   0.08,  New conv eig=  0
                Basis= 16,  Time=   0.08,  New conv eig=  0
                Basis= 19,  Time=   0.12,  New conv eig=  3
                Basis= 22,  Time=   0.12,  New conv eig=  3
                Basis= 25,  Time=   0.12,  New conv eig=  5
                Basis= 28,  Time=   0.12,  New conv eig=  6
                Basis= 31,  Time=   0.12,  New conv eig=  8
                Basis= 34,  Time=   0.17,  New conv eig= 12
                Basis= 37,  Time=   0.17,  New conv eig= 14
                Basis= 40,  Time=   0.22,  New conv eig= 14
                Basis= 43,  Time=   0.22,  New conv eig= 15
                Basis= 46,  Time=   0.22,  New conv eig= 16
                Basis= 49,  Time=   0.27,  New conv eig= 17
                Basis= 52,  Time=   0.30,  New conv eig= 20
End of sweep: Basis= 52,  Time=   0.30,  New conv eig= 20
                Basis= 30,  Time=   0.34,  New conv eig=  0
                Basis= 33,  Time=   0.34,  New conv eig=  0
                Basis= 36,  Time=   0.34,  New conv eig=  0
End of sweep: Basis= 36,  Time=   0.34,  New conv eig=  0

results =

  EigenResults with properties:

    Eigenvectors: [2124x19 double]
     Eigenvalues: [19x1 double]
            Mesh: [1x1 pde.FEMesh]
```

Plot the solution for mode 10.

```
pdeplot(model,'xydata',results.Eigenvectors(:,10));
```
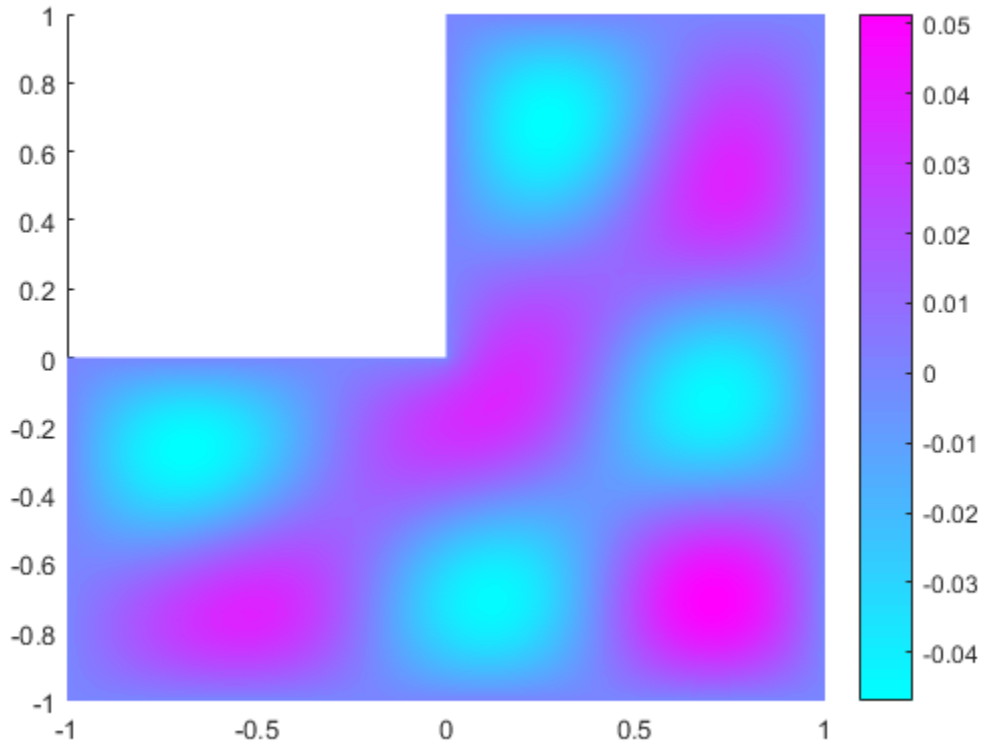
- "Eigenvalues and Eigenfunctions for the L-Shaped Membrane" on page 3-110
- "Eigenvalues and Eigenmodes of a Square" on page 3-119

Properties

Mesh — Finite element mesh
FEMesh object

Finite element mesh, specified as a FEMesh object.

Eigenvectors — Solution eigenvectors
matrix | 3-D array

Solution eigenvectors, returned as a matrix or 3-D array. The solution is a matrix for scalar eigenvalue problems, and a 3-D array for eigenvalue systems. For details, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: `double`
Complex Number Support: Yes

`Eigenvalues` — Solution eigenvalues
vector

Solution eigenvalues, returned as a vector. The vector is in order by the real part of the eigenvalues from smallest to largest.

Data Types: `double`
Complex Number Support: Yes

Object Functions

interpolateSolution   Interpolate PDE solution to arbitrary points

Create Object

`solvepdeeig` returns PDE eigenvalue solutions as an `EigenResults` object.

`createPDEResults` returns an `EigenResults` object from a PDE eigenvalue solution as returned by `pdeeig`.

See Also
`solvepdeeig` | `StationaryResults` | `TimeDependentResults`

More About
·   "Solve Problems Using PDEModel Objects" on page 2-14

Introduced in R2016a

evaluate

Interpolate data to selected locations

Compatibility

THIS FUNCTION SUPPORTS THE LEGACY WORKFLOW. Using the [p,e,t] representation of FEMesh data is not recommended. Use interpolateSolution and evaluateGradient to interpolate a PDE solution and its gradient to arbitrary points without switching to a [p,e,t] representation.

Syntax

```
uOut = evaluate(F,pOut)
uOut = evaluate(F,x,y)
uOut = evaluate(F,x,y,z)
```

Description

uOut = evaluate(F,pOut) returns the interpolated values from the interpolant F at the points pOut.

Note: If a query point is outside the mesh, evaluate returns NaN for that point.

---

uOut = evaluate(F,x,y) returns the interpolated values from the interpolant F at the points [x(k),y(k)], for k from 1 through numel(x). This syntax applies to 2-D geometry.

uOut = evaluate(F,x,y,z) returns the interpolated values from the interpolant F at the points [x(k),y(k),z(k)], for k from 1 through numel(x). This syntax applies to 3-D geometry.

Examples

Interpolate to a matrix of values

This example shows how to interpolate a solution to a scalar problem using a `pOut` matrix of values.

Solve the equation $-\Delta u = 1$ on the unit disk with zero Dirichlet conditions.

```
g0 = [1;0;0;1]; % circle centered at (0,0) with radius 1
sf = 'C1';
g = decsg(g0,sf,sf'); % decomposed geometry matrix
problem = allzerobc(g); % zero Dirichlet conditions
[p,e,t] = initmesh(g);
c = 1;
a = 0;
f = 1;
u = assempde(problem,p,e,t,c,a,f); % solve the PDE
```

Construct an interpolator for the solution.

```
F = pdeInterpolant(p,t,u);
```

Generate a random set of coordinates in the unit square. Evaluate the interpolated solution at the random points.

```
rng default % for reproducibility
pOut = rand(2,25); % 25 numbers between 0 and 1
uOut = evaluate(F,pOut);
numNaN = sum(isnan(uOut))


numNaN =

     9
```

`uOut` contains some `NaN` entries because some points in `pOut` are outside of the unit disk.

Interpolate to x, y values

This example shows how to interpolate a solution to a scalar problem using `x`, `y` values.

Solve the equation $-\Delta u = 1$ on the unit disk with zero Dirichlet conditions.

```
g0 = [1;0;0;1]; % circle centered at (0,0) with radius 1
sf = 'C1';
g = decsg(g0,sf,sf'); % decomposed geometry matrix
problem = allzerobc(g); % zero Dirichlet conditions
[p,e,t] = initmesh(g);
c = 1;
a = 0;
f = 1;
u = assempde(problem,p,e,t,c,a,f); % solve the PDE
```

Construct an interpolator for the solution.

```
F = pdeInterpolant(p,t,u); % create the interpolant
```

Evaluate the interpolated solution at grid points in the unit square with spacing `0.2`.

```
[x,y] = meshgrid(0:0.2:1);
uOut = evaluate(F,x,y);
numNaN = sum(isnan(uOut))
```

```
numNaN =

    12
```

`uOut` contains some `NaN` entries because some points in the unit square are outside of the unit disk.

Interpolate a solution with multiple components

This example shows how to interpolate the solution to a system of $N = 3$ equations.

Solve the system of equations $-\Delta \mathbf{u} = \mathbf{f}$ with Dirichlet boundary conditions on the unit disk, where

$$\mathbf{f} = \left[\sin(x) + \cos(y), \cosh(xy), \frac{xy}{1 + x^2 + y^2} \right]^T.$$

```
g0 = [1;0;0;1]; % circle centered at (0,0) with radius 1
```

```
sf = 'C1';
g = decsg(g0,sf,sf'); % decomposed geometry matrix
problem = allzerobc(g,3); % zero Dirichlet conditions, 3 components
[p,e,t] = initmesh(g);
c = 1;
a = 0;
f = char('sin(x) + cos(y)','cosh(x.*y)','x.*y./(1+x.^2+y.^2)');
u = assempde(problem,p,e,t,c,a,f); % solve the PDE
```

Construct an interpolant for the solution.

```
F = pdeInterpolant(p,t,u); % create the interpolant
```

Interpolate the solution at a circle.

```
s = linspace(0,2*pi);
x = 0.5 + 0.4*cos(s);
y = 0.4*sin(s);
uOut = evaluate(F,x,y);
```

Plot the three solution components.

```
npts = length(x);
plot3(x,y,uOut(1:npts),'b')
hold on
plot3(x,y,uOut(npts+1:2*npts),'k')
plot3(x,y,uOut(2*npts+1:end),'r')
hold off
view(35,35)
```
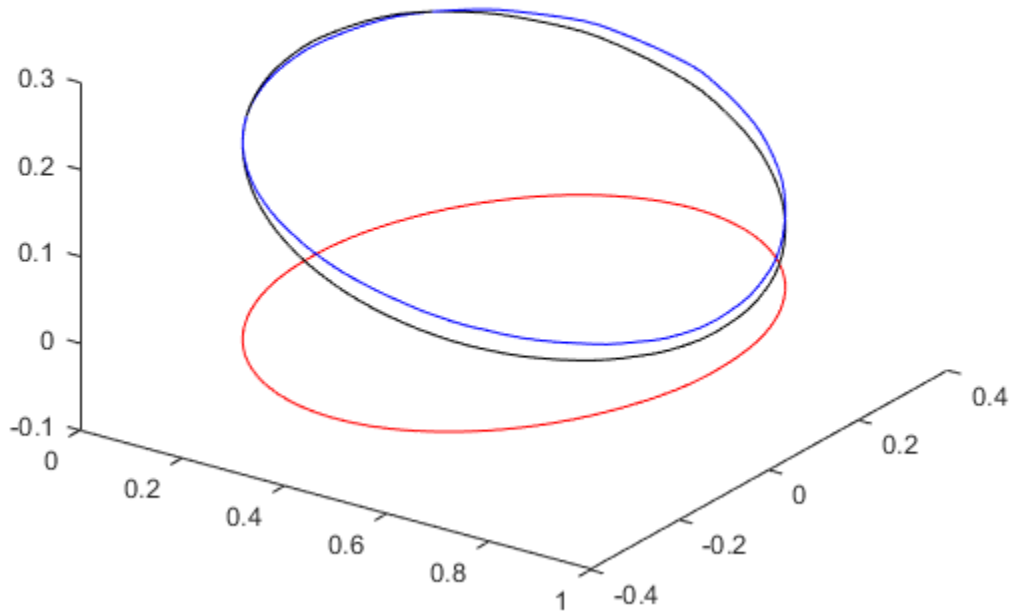
Interpolate a time-varying solution

This example shows how to interpolate a solution that depends on time.

Solve the equation

$$\frac{\partial u}{\partial t} - \Delta u = 1$$

on the unit disk with zero Dirichlet conditions and zero initial conditions. Solve at five times from 0 to 1.

```
g0 = [1;0;0;1]; % circle centered at (0,0) with radius 1
```

```
sf = 'C1';
g = decsg(g0,sf,sf'); % decomposed geometry matrix
problem = allzerobc(g); % zero Dirichlet conditions
[p,e,t] = initmesh(g);
c = 1;
a = 0;
f = 1;
d = 1;
tlist = 0:1/4:1;
u = parabolic(0,tlist,problem,p,e,t,c,a,f,d);

52 successful steps
0 failed attempts
106 function evaluations
1 partial derivatives
13 LU decompositions
105 solutions of linear systems
```

Construct an interpolant for the solution.

```
F = pdeInterpolant(p,t,u);
```

Interpolate the solution at `x = 0.1`, `y = -0.1`, and all available times.

```
x = 0.1;
y = -0.1;
uOut = evaluate(F,x,y)


uOut =

         0    0.1809    0.2278    0.2388    0.2413
```

The solution starts at 0 at time 0, as it should. It grows to about 1/4 at time 1.

Interpolate to a Grid

This example shows how to interpolate an elliptic solution to a grid.

Define and Solve the Problem

Use the built-in geometry functions to create an L-shaped region with zero Dirichlet boundary conditions. Solve an elliptic PDE with coefficients $c = 1$, $a = 0$, $f = 1$, with zero Dirichlet boundary conditions.

```
[p,e,t] = initmesh('lshapeg'); % Predefined geometry
u = assempde('lshapeb',p,e,t,1,0,1); % Predefined boundary condition
```

Create an Interpolant

Create an interpolant for the solution.

```
F = pdeInterpolant(p,t,u);
```

Create a Grid for the Solution

```
xgrid = -1:0.1:1;
ygrid = -1:0.2:1;
[X,Y] = meshgrid(xgrid,ygrid);
```

The resulting grid has some points that are outside the L-shaped region.

Evaluate the Solution On the Grid

```
uout = evaluate(F,X,Y);
```

The interpolated solution uout is a column vector. You can reshape it to match the size of X or Y. This gives a matrix, like the output of the tri2grid function.

```
Z = reshape(uout,size(X));
```

Input Arguments

F — Interpolant
output of pdeInterpolant

Interpolant, specified as the output of pdeInterpolant.

Example: F = pdeInterpolant(p,t,u)

pOut — Query points
matrix with two or three rows

Query points, specified as a matrix with two or three rows. The first row represents the x component of the query points, the second row represents the y component, and, for 3-D geometry, the third row represents the z component. evaluate computes the interpolant at each column of pOut. In other words, evaluate interpolates at the points pOut(:,k).

Example: `pOut = [-1.5,0,1; 1,1,2.2]`

Data Types: `double`

x — Query point component
vector or array

Query point component, specified as a vector or array. `evaluate` interpolates at either 2-D points `[x(k),y(k)]` or at 3-D points `[x(k),y(k),z(k)]`. The x and y, and z arrays must contain the same number of entries.

`evaluate` transforms query point components to the linear index representation, such as `x(:)`.

Example: `x = -1:0.2:3`

Data Types: `double`

y — Query point component
vector or array

Query point component, specified as a vector or array. `evaluate` interpolates at either 2-D points `[x(k),y(k)]` or at 3-D points `[x(k),y(k),z(k)]`. The x and y, and z arrays must contain the same number of entries.

`evaluate` transforms query point components to the linear index representation, such as `y(:)`.

Example: `y = -1:0.2:3`

Data Types: `double`

z — Query point component
vector or array

Query point component, specified as a vector or array. `evaluate` interpolates at either 2-D points `[x(k),y(k)]` or at 3-D points `[x(k),y(k),z(k)]`. The x and y, and z arrays must contain the same number of entries.

`evaluate` transforms query point components to the linear index representation, such as `z(:)`.

Example: `z = -1:0.2:3`

Data Types: `double`

Output Arguments

`uOut` — Interpolated values
array

Interpolated values, returned as an array. `uOut` has the same number of columns as the data `u` used in creating `F`. The number of rows of `uOut` is N times the number of query points. N is the number of components in the training data `u`.

If a query point is outside the mesh, `evaluate` returns `NaN` for that point.

More About

Element

An *element* is a basic unit in the finite-element method.

For 2-D problems, an element is a triangle `t` in the `[p,e,t]` "Mesh Data" on page 2-211 structure or in the `model.Mesh.Element` property. If the triangle represents a linear element, it has nodes only at the triangle corners. If the triangle represents a quadratic element, then it has nodes at the triangle corners and edge centers.

For 3-D problems, an element is a tetrahedron with either four or ten points. A four-point (linear) tetrahedron has nodes only at its corners. A ten-point (quadratic) tetrahedron has nodes at its corners and at the center point of each edge. For a sketch of the two tetrahedra, see "Mesh Data" on page 2-211.

The `[p,e,t]` data structure for an element `t` has the form `[p1;p2;...;pn;sd]`, where the `p` values are indexes of the nodes (points `p` in `t`), and `sd` is the subdomain number.

Algorithms

For each point where a solution is requested (`pOut`), there are two steps in the interpolation process. First, the *element* containing the point must be located and second, interpolation within that element must be performed using the element shape functions and the values of the solution at the element's node points.

- "Mesh Data" on page 2-211

See Also
pdeInterpolant

Introduced in R2014b

evaluateGradient

Evaluate gradients of PDE solutions at arbitrary points

Compatibility

EVALUATING GRADIENTS AT ARBITRARY POINTS IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
[gradx,grady] = evaluateGradient(results,xq,yq)
[gradx,grady,gradz] = evaluateGradient(results,xq,yq,zq)
[ ___ ] = evaluateGradient(results,querypoints)

[ ___ ] = evaluateGradient( ___ ,iU)
[ ___ ] = evaluateGradient( ___ ,iT)
[ ___ ] = evaluateGradient( ___ ,iU,iT)
```

Description

`[gradx,grady] = evaluateGradient(results,xq,yq)` returns the interpolated values of gradients of the PDE solution `results` at the 2-D points specified in `xq` and `yq`.

`[gradx,grady,gradz] = evaluateGradient(results,xq,yq,zq)` returns the interpolated gradients at the 3-D points specified in `xq`, `yq`, and `zq`.

`[ ___ ] = evaluateGradient(results,querypoints)` returns the interpolated values of the gradients at the points specified in `querypoints`.

`[ ___ ] = evaluateGradient( ___ ,iU)` returns the interpolated values of the gradients for the *system* of stationary equations for equation indices (components) `iU`. When solving a system of elliptic PDEs, specify `iU` after the input arguments in any of the previous syntaxes.

`[ ___ ] = evaluateGradient( ___ ,iT)` returns the interpolated values of the gradients for the time-dependent equation at times `iT`. When solving a time-dependent PDE, specify `iT` after the input arguments in any of the previous syntaxes.

[___] = evaluateGradient(___ ,iU,iT) returns the interpolated values of the gradients of the solutions to the system of time-dependent equations for equation indices (components) iU at times iT.

Examples

Evaluate Gradients for Scalar Elliptic Problem

Evaluate gradients of the solution to a scalar elliptic problem along a line. Plot the results.

Create the solution to the problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
generateMesh(model,'Hmax',0.05);
results = solvepde(model);
```

Evaluate gradients of the solution along the straight line from $(x,y) = (-1,-1)$ to $(1,1)$. Plot the results as a quiver plot by using quiver.

```
xq = linspace(-1,1,101);
yq = xq;
[gradx,grady] = evaluateGradient(results,xq,yq);

gradx = reshape(gradx,size(xq));
grady = reshape(grady,size(yq));

quiver(xq,yq,gradx,grady)
xlabel('x')
ylabel('y')
```
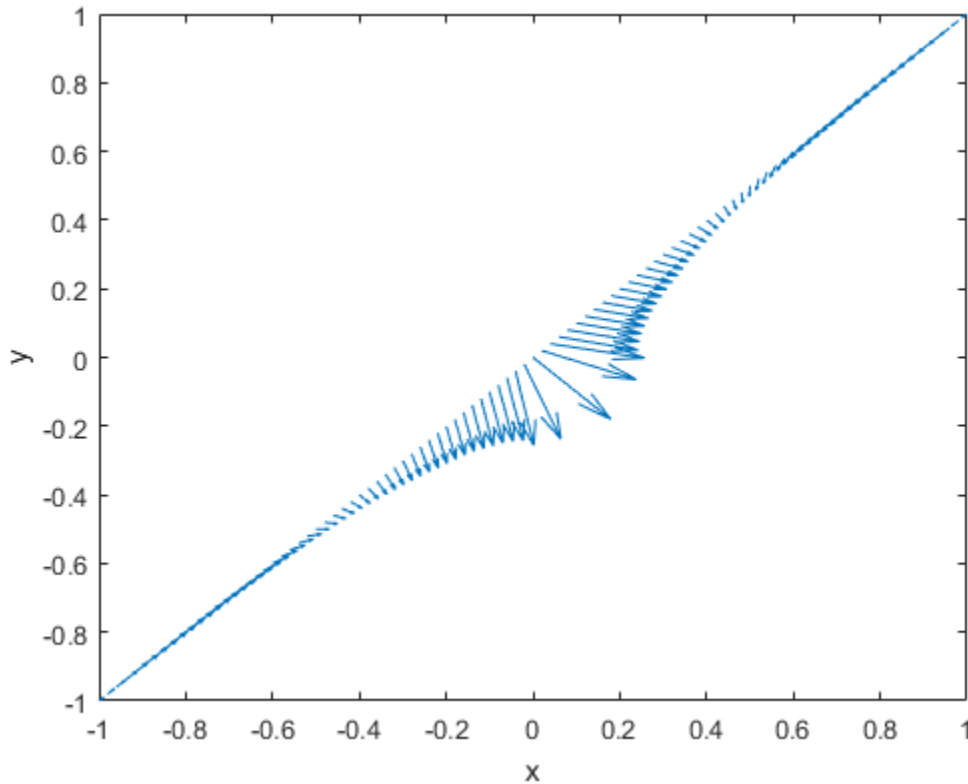
Evaluate Gradients for Poisson's Equation

Calculate gradients for the mean exit time of a Brownian particle from a region that contains absorbing (escape) boundaries and reflecting boundaries. Use the Poisson's equation with constant coefficients and 3-D rectangular block geometry to model this problem.

Create the solution for this problem.

```
model = createpde;
importGeometry(model,'Block.stl');
applyBoundaryCondition(model,'face',[1,2,5],'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
```

```
                              'c',1,...
                              'a',0,...
                              'f',2);
generateMesh(model);
results = solvepde(model);
```

Create a grid and interpolate gradients of the solution to the grid.

```
[X,Y,Z] = meshgrid(1:16:100,1:6:20,1:7:50);
[gradx,grady,gradz] = evaluateGradient(results,X,Y,Z);
```

Reshape the gradients to the shape of the grid and plot the gradients.

```
gradx = reshape(gradx,size(X));
grady = reshape(grady,size(Y));
gradz = reshape(gradz,size(Z));

quiver3(X,Y,Z,gradx,grady,gradz)
axis equal
xlabel('x')
ylabel('y')
zlabel('z')
```
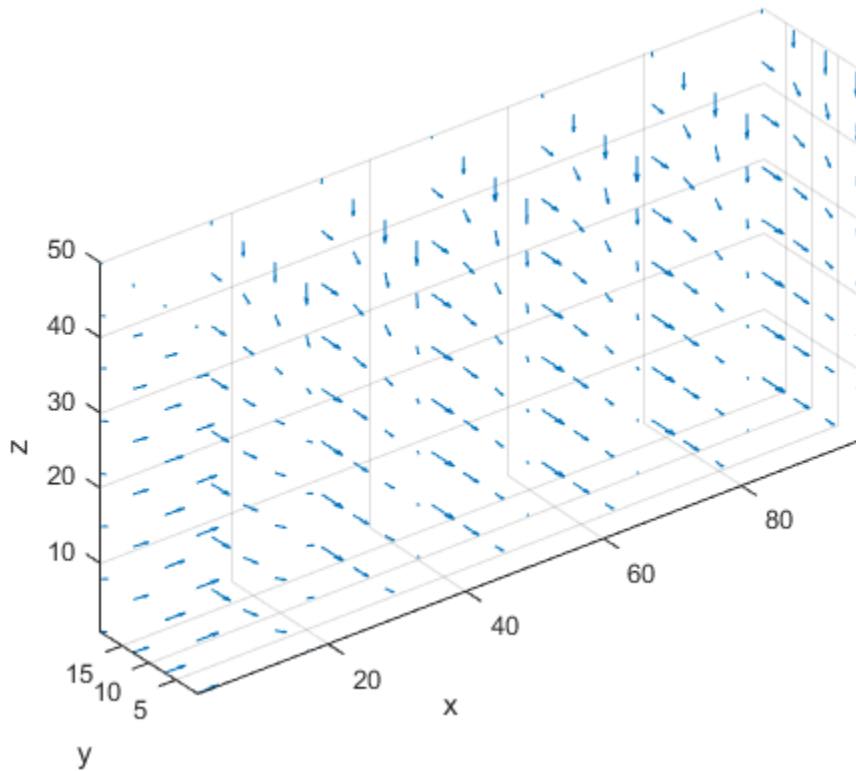
Evaluate Gradients Using Query Matrix

Solve a scalar elliptic problem and interpolate gradients of the solution to a dense grid. Use a query matrix to specify the grid.

Create the solution to the problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
```

```
                                 'f',1);
generateMesh(model,'Hmax',0.05);
results = solvepde(model);
```

Interpolate gradients of the solution to the grid from -1 to 1 in each direction. Plot the result using the `quiver` plotting function.

```
v = linspace(-1,1,101);
[X,Y] = meshgrid(v);
querypoints = [X(:),Y(:)]';

[gradx,grady] = evaluateGradient(results,querypoints);
quiver(X(:),Y(:),gradx,grady)
xlabel('x')
ylabel('y')
```
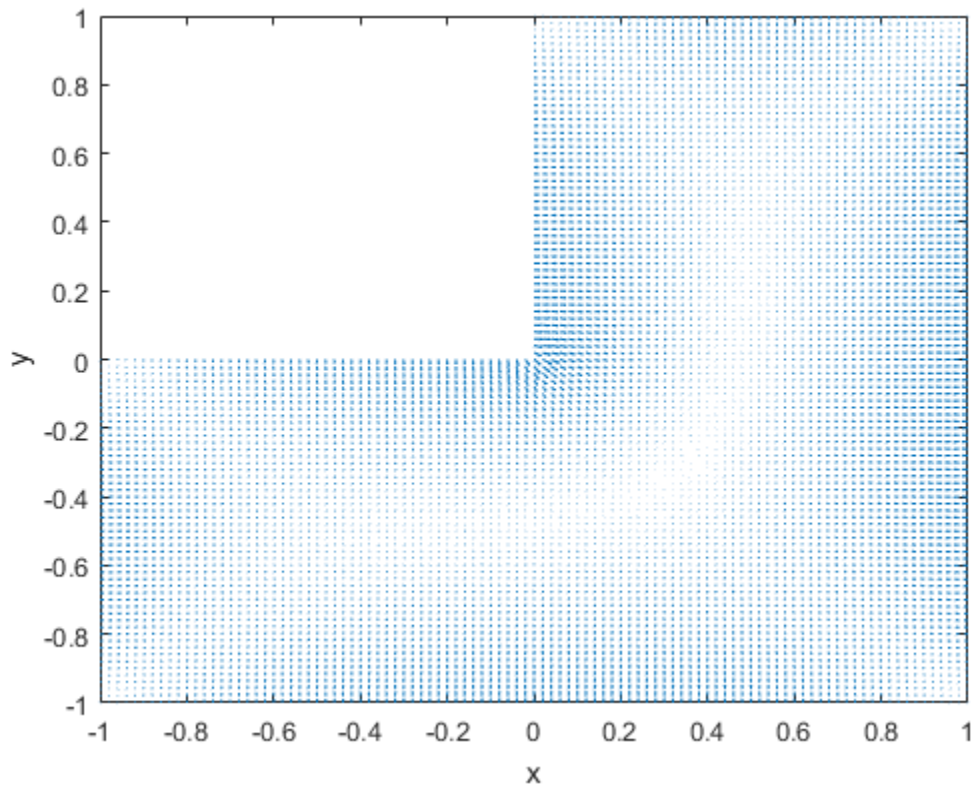
Zoom in on a particular part of the plot to see more details. For example, limit the plotting range to 0.2 in each direction.

```
axis([-0.2 0.2 -0.2 0.2])
```
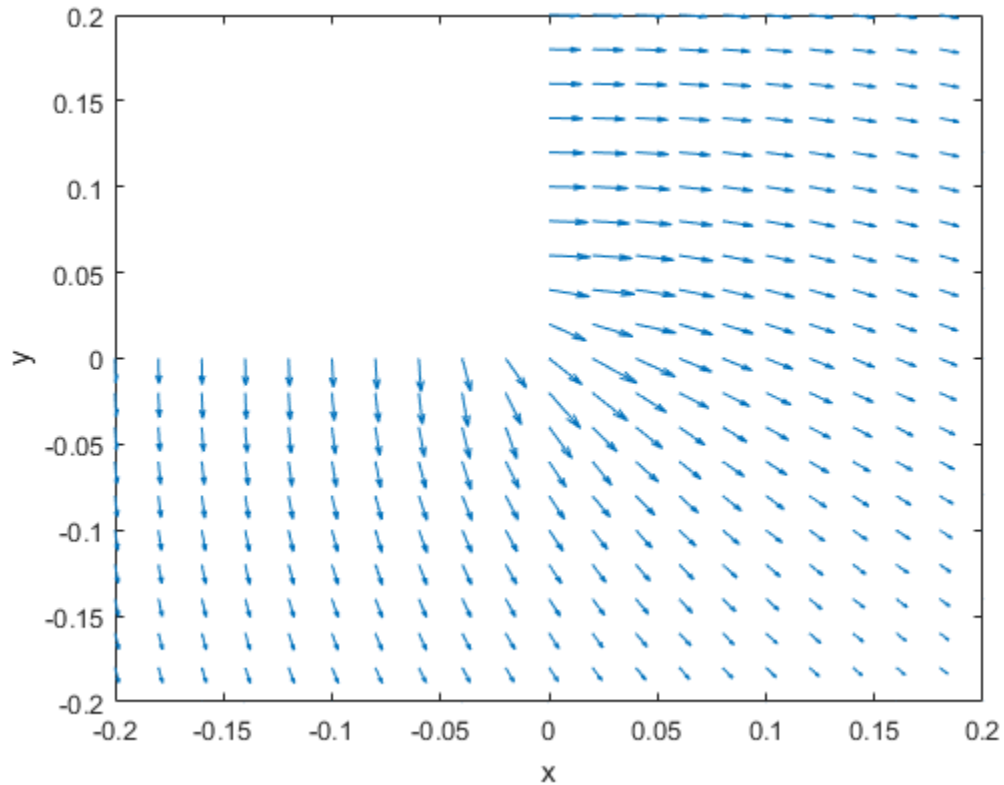

Evaluate Gradients of Solution of Elliptic System

Evaluate gradients of the solution to a two-component elliptic system and plot the results.

Create a PDE model for two components.

```
model = createpde(2);
```

Create the 2-D geometry as a rectangle with a circular hole in its center. For details about creating the geometry, see the example in "Solve PDEs with Constant Boundary Conditions" on page 2-185.

```
R1 = [3,4,-0.3,0.3,0.3,-0.3,-0.3,-0.3,0.3,0.3]';
C1 = [1,0,0,0.1]';
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];
ns = (char('R1','C1'))';
sf = 'R1 - C1';
g = decsg(geom,sf,ns);
```

Include the geometry in the model and view the geometry.

```
geometryFromEdges(model,g);
pdegplot(model,'EdgeLabels','on');
axis equal
axis([-0.4,0.4,-0.4,0.4])
```
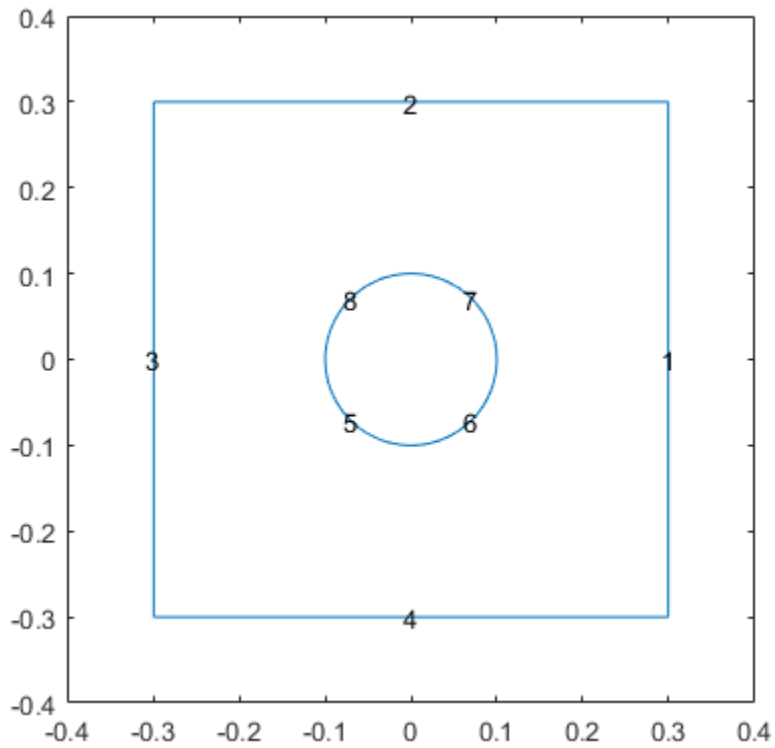
Set the boundary conditions and coefficients.

```
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',[2; -2]);

applyBoundaryCondition(model,'edge',3,'u',[-1,1]);
applyBoundaryCondition(model,'edge',1,'u',[1,-1]);
applyBoundaryCondition(model,'edge',[2,4:8],'g',[0,0]);
```

Create a mesh and solve the problem.

```
generateMesh(model,'Hmax',0.1);
results = solvepde(model);
```

Interpolate the gradients of the solution to the grid from -0.3 to 0.3 in each direction for each of the two components.

```
v = linspace(-0.3,0.3,15);
[X,Y] = meshgrid(v);

[gradx,grady] = evaluateGradient(results,X,Y,[1,2]);
```

Plot the gradients for the first component.

```
figure
gradx1 = gradx(:,1);
grady1 = grady(:,1);
quiver(X(:),Y(:),gradx1,grady1)
title('Component 1')
axis equal
xlim([-0.3,0.3])
```
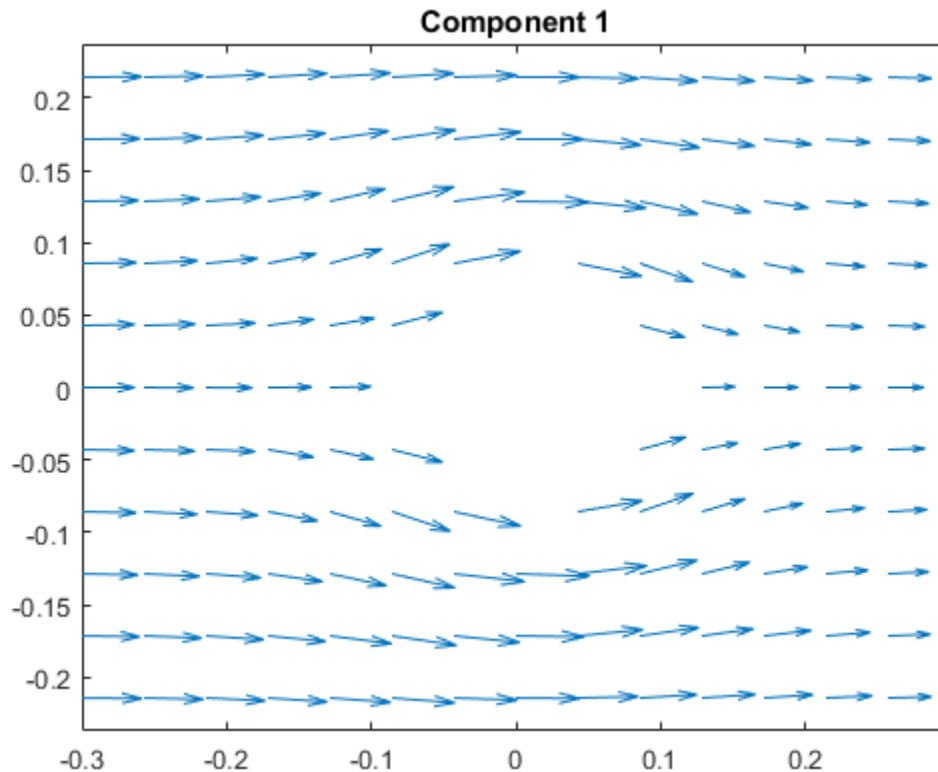

Component 1

Plot the gradients for the second component.

```
figure
gradx2 = gradx(:,2);
grady2 = grady(:,2);
quiver(X(:),Y(:),gradx2,grady2)
title('Component 2')
axis equal
xlim([-0.3,0.3])
```
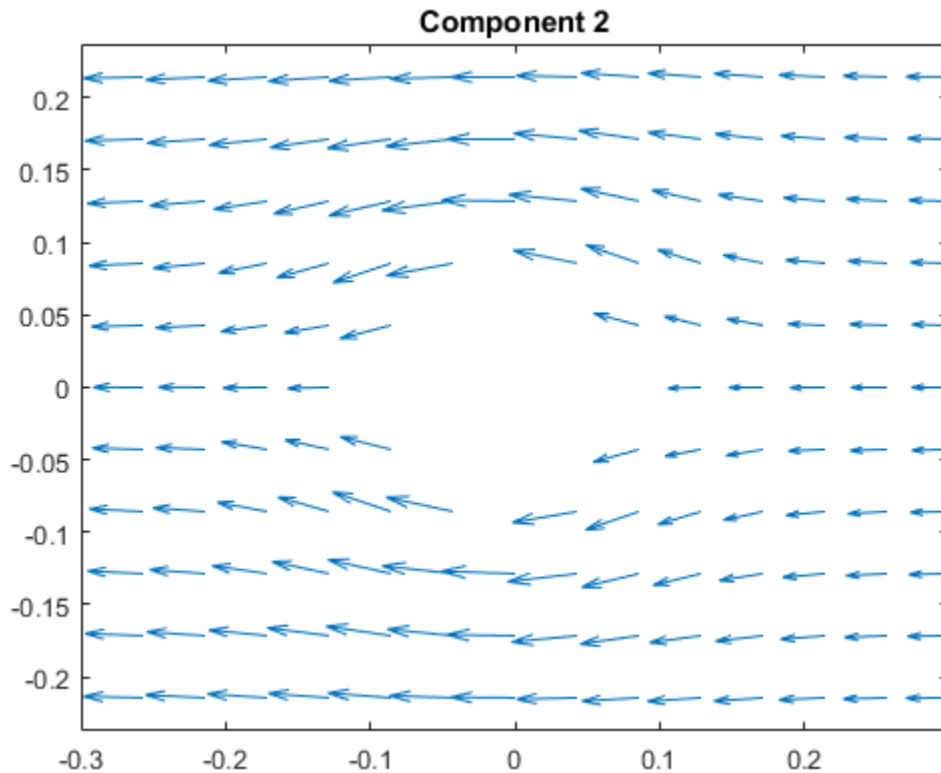

Evaluate Gradients of Solution of Hyperbolic System

Solve a system of hyperbolic PDEs and interpolate the solution.

Import slab geometry for a 3-D problem with three solution components. Plot the geometry.

```
model = createpde(3);
importGeometry(model,'Plate10x10x1.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```


Set boundary conditions such that face 2 is fixed (zero deflection in any direction) and face 5 has a load of 1e3 in the positive *z*-direction. This load causes the slab to bend upward. Set the initial condition that the solution is zero, and its derivative with respect to time is also zero.

```
applyBoundaryCondition(model,'face',2,'u',[0,0,0]);
applyBoundaryCondition(model,'face',5,'g',[0,0,1e3]);
setInitialConditions(model,0,0);
```

Create PDE coefficients for the equations of linear elasticity. Set the material properties to be similar to those of steel. See "3-D Linear Elasticity Equations in Toolbox Form" on page 3-35.

```
E = 200e9;
nu = 0.3;
specifyCoefficients(model,'m',1,...
                          'd',0,...
                          'c',elasticityC3D(E,nu),...
                          'a',0,...
                          'f',[0;0;0]);
```

Generate a mesh, setting `Hmax` to `1`.

```
generateMesh(model,'Hmax',0.45);
```

Solve the problem for times `0` through `5e-3` in steps of `1e-4`. You might have to wait a few minutes for the solution.

```
tlist = 0:1e-4:5e-3;
results = solvepde(model,tlist);
```

Evaluate the gradients of the solution at fixed *x*- and *z*-coordinates in the centers of their ranges, `5` and `0.5` respectively. Evaluate for *y* from `0` through `10` in steps of `0.2`. Obtain just component `3`, the *z*-component.

```
yy = 0:0.2:10;
zz = 0.5*ones(size(yy));
xx = 10*zz;
component = 3;
[gradx,grady,gradz] = evaluateGradient(results,xx,yy,zz,component,1:length(tlist));
```

The three projections of the gradients of the solution are 51-by-1-by-51 arrays. Use `squeeze` to remove the singleton dimension. Removing the singleton dimension transforms these arrays to 51-by-51 matrices which simplifies indexing into them.

```
gradx = squeeze(gradx);
grady = squeeze(grady);
gradz = squeeze(gradz);
```

Plot the interpolated gradient component `grady` along the *y* axis for the first eight values of the time interval `tlist`.

```
figure
t = [1:8];
```

```
for i = t
  p(i) = plot(yy,grady(:,i),'DisplayName', strcat('t=', num2str(tlist(i))));
  hold on
end
legend(p(t))
xlabel('y')
ylabel('grady')
```
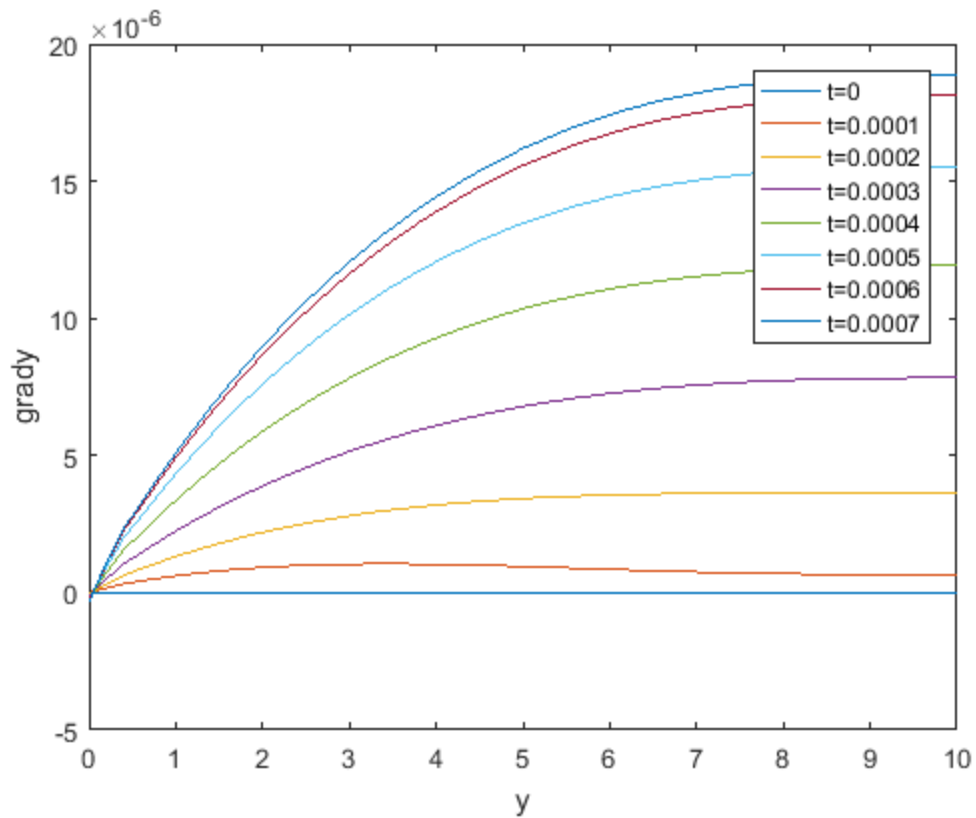

Input Arguments

results — PDE solution
StationaryResults object | TimeDependentResults object

PDE solution, specified as a `StationaryResults` object or a `TimeDependentResults` object. Create `results` using `solvepde` or `createPDEResults`.

Example: `results = solvepde(model)`

xq — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `evaluateGradient` evaluates the gradients of the solution at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. So `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`evaluateGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The returned gradients are column vectors of the same size. To ensure that the dimensions of the gradient components are consistent with the dimensions of the original query points, use `reshape`. For example, use `gradx = reshape(gradx,size(xq))`.

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `evaluateGradient` evaluates the gradients of the solution at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. So `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`evaluateGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The returned gradients are column vectors of the same size. To ensure that the dimensions of the gradient components are consistent with the dimensions of the original query points, use `reshape`. For example, use `grady = reshape(grady,size(yq))`.

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `evaluateGradient` evaluates the gradients of the solution at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. So `xq`, `yq`, and `zq` must have the same number of entries.

`evaluateGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The returned gradients are column vectors of the same size. To ensure that the dimensions of the gradient components are consistent with the dimensions of the original query points, use `reshape`. For example, use `gradz = reshape(gradz,size(zq))`.

Data Types: `double`

`querypoints` — Query points
real matrix

Query points, specified as a real matrix with either two rows for 2-D geometry, or three rows for 3-D geometry. `evaluateGradient` evaluates the gradients of the solution at the coordinate points `querypoints(:,i)`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For 2-D geometry, `querypoints = [0.5,0.5,0.75,0.75; 1,2,0,0.5]`

Data Types: `double`

`iU` — Equation indices
vector of positive integers

Equation indices, specified as a vector of positive integers. Each entry in `iU` specifies an equation index.

Example: `iU = [1,5]` specifies the indices for the first and fifth equations.

Data Types: `double`

`iT` — Time or mode indices
vector of positive integers

Time or mode indices, specified as a vector of positive integers. Each entry in `iT` specifies a time index for parabolic or hyperbolic solutions, or a mode index for eigenvalue solutions.

Example: `iT = 1:5:21` specifies the time or mode for every fifth solution up to 21.

Data Types: `double`

Output Arguments

gradx — *x*-component of the gradient
array

x-component of the gradient, returned as an array. For query points that are outside the geometry, gradx = NaN. For information about the size of gradx, see "Dimensions of Solutions and Gradients" on page 3-167.

grady — *y*-component of the gradient
array

y-component of the gradient, returned as an array. For query points that are outside the geometry, grady = NaN. For information about the size of grady, see "Dimensions of Solutions and Gradients" on page 3-167.

gradz — *z*-component of the gradient
array

z-component of the gradient, returned as an array. For query points that are outside the geometry, gradz = NaN. For information about the size of gradz, see "Dimensions of Solutions and Gradients" on page 3-167.

More About

Tips

The results object contains the solution and its gradient calculated at the nodal points of the triangular or tetrahedral mesh. You can access the solution and three components of the gradient at nodal points by using dot notation.

interpolateSolution and evaluateGradient let you interpolate the solution and its gradient to a custom grid, for example, specified by meshgrid.

See Also
contour | interpolateSolution | PDEModel | quiver | quiver3 |
StationaryResults | TimeDependentResults

Introduced in R2016a

FEMesh Properties

Mesh object

Compatibility

THIS PAGE DESCRIBES BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Description

An `FEMesh` object contains a description of the finite element mesh. A `PDEModel` container has an `FEMesh` object in its `Mesh` property.

Generate a mesh for your model using the `generateMesh` function.

Properties

`Nodes` — Mesh nodes
matrix

Mesh nodes, returned as a matrix. `Nodes` is a D-by-`Nn` matrix, where `D` is the number of geometry dimensions (2 or 3), and `Nn` is the number of nodes in the mesh. Each column of `Nodes` contains the x, y, and in 3-D, z coordinates for that mesh node.

2-D meshes have nodes at the mesh triangle corners for linear elements, and at the corners and edge midpoints for `'quadratic'` elements. 3-D meshes have nodes at tetrahedral vertices, and the `'quadratic'` elements have additional nodes at the center points of each edge. See "Mesh Data" on page 2-211.

Data Types: `double`

`Elements` — Mesh elements
matrix

Mesh elements, returned as an M-by-`Ne` matrix, where `Ne` is the number of elements in the mesh, and `M` is:

- 3 for 2-D triangles with `'linear'` `GeometricOrder`

- **6** for 2-D triangles with `'quadratic'` GeometricOrder
- **4** for 3-D tetrahedra with `'linear'` GeometricOrder
- **10** for 3-D tetrahedra with `'quadratic'` GeometricOrder

Each column in `Elements` contains the indices of the nodes for that mesh element.

Data Types: `double`

`MaxElementSize` — Target maximum mesh element size
positive real number

Target maximum mesh element size, returned as a positive real number. The maximum mesh element size is the length of the longest edge in the mesh. The `generateMesh` `Hmax` name-value pair sets the target maximum size at the time it creates the mesh. `generateMesh` can occasionally create a mesh with some elements that exceed `MaxElementSize` by a few percent.

Data Types: `double`

`MinElementSize` — Target minimum mesh element size
positive real number

Target minimum mesh element size, returned as a positive real number. The minimum mesh element size is the length of the shortest edge in the mesh.

- For a 2-D mesh, `MinElementSize` is the minimum mesh element size.
- For a 3-D mesh, the `generateMesh` `Hmin` name-value pair sets the target minimum size the at the time it creates the mesh. `generateMesh` can occasionally create a mesh with some elements that are a few percent smaller than `MinElementSize`.

Data Types: `double`

`GeometricOrder` — Element polynomial order
`'linear'` | `'quadratic'`

Element polynomial order, returned as `'linear'` or `'quadratic'`. See `Elements` or "Mesh Data" on page 2-211.

Data Types: `double`

See Also
generateMesh | meshToPet | PDEModel

More About

- "Solve Problems Using PDEModel Objects" on page 2-14
- "Finite Element Basis for 3-D" on page 5-10
- "Mesh Data" on page 2-211

Introduced in R2015a

findCoefficients

Locate active PDE coefficients

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW. For the corresponding step in the legacy workflow, see the legacy examples on the "PDE Coefficients" page.

Syntax

```
ca = findCoefficients(coeffs,regiontype,regionid)
```

Description

`ca = findCoefficients(coeffs,regiontype,regionid)` returns the active coefficient assignment `ca` for the coefficients in the specified region.

Examples

Find the Active Coefficients for a Region

Create a PDE model that has a few subdomains.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'SubdomainLabels','on');
ylim([-1.1,1.1])
axis equal
```

Set coefficients on each pair of regions.

```
specifyCoefficients(model,'m',0,'d',0,'c',12,'a',0,'f',1,'face',[1,2]);
specifyCoefficients(model,'m',0,'d',0,'c',13,'a',0,'f',2,'face',[1,3]);
specifyCoefficients(model,'m',0,'d',0,'c',23,'a',0,'f',3,'face',[2,3]);
```

Check the coefficient specification for region 1.

```
coeffs = model.EquationCoefficients;
ca = findCoefficients(coeffs,'face',1)

ca =

  CoefficientAssignment with properties:
```

```
    RegionType: 'face'
      RegionID: [1 3]
             m: 0
             d: 0
             c: 13
             a: 0
             f: 2
```

- "View, Edit, and Delete PDE Coefficients" on page 2-151
- "Solve Problems Using PDEModel Objects" on page 2-14

Input Arguments

coeffs — Model coefficients
EquationCoefficients property of a PDE model

Model coefficients, specified as the EquationCoefficients property of a PDE model. Coefficients can be complex numbers.

Example: model.EquationCoefficients

regiontype — Geometric region type
'face' for a 2-D model | 'cell' for a 3-D model

Geometric region type, specified as 'face' for a 2-D model, or 'cell' for a 3-D model.

Example: ca = findCoefficients(coeffs,'face',[1,3])

Data Types: char

regionid — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. View the subdomain labels for a 2-D model using pdegplot(model,'SubdomainLabels','on'). Currently, there are no subdomains for 3-D models, so the only acceptable value for a 3-D model is 1.

Example: ca = findCoefficients(coeffs,'face',[1,3])

Data Types: double

Output Arguments

ca — Coefficient assignment
CoefficientAssignment object

Coefficient assignment, returned as a CoefficientAssignment object.

More About

· "PDE Coefficients"

See Also

CoefficientAssignment Properties | specifyCoefficients

Introduced in R2016a

findInitialConditions

Locate active initial conditions

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW. For the corresponding step in the legacy workflow, see the legacy examples on the "Initial Conditions" page.

Syntax

```
ic = findInitialConditions(ics,regiontype,regionid)
```

Description

`ic = findInitialConditions(ics,regiontype,regionid)` returns the active initial condition assignment `ic` for the initial conditions in the specified region.

Examples

Find the Active Initial Conditions

Find the active initial conditions for a region.

Create a PDE model that has a few subdomains.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'SubdomainLabels','on');
ylim([-1.1,1.1])
axis equal
```
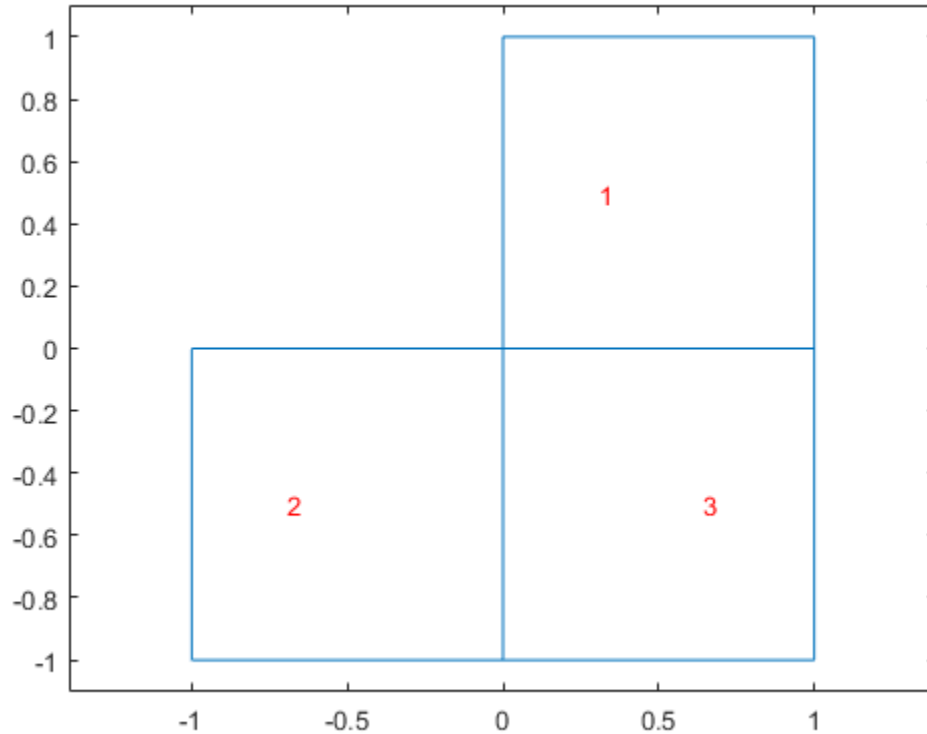
Set initial conditions on each pair of regions.

```
setInitialConditions(model,12,'face',[1,2]);
setInitialConditions(model,13,'face',[1,3]);
setInitialConditions(model,23,'face',[2,3]);
```

Check the initial conditions specification for region 1.

```
ics = model.InitialConditions;
ic = findInitialConditions(ics,'face',1)

ic =

  GeometricInitialConditions with properties:
```

```
         RegionType: 'face'
           RegionID: [1 3]
       InitialValue: 13
  InitialDerivative: []
```

- "View, Edit, and Delete Initial Conditions" on page 2-158
- "Solve Problems Using PDEModel Objects" on page 2-14

Input Arguments

`ics` — Model initial conditions
`InitialConditions` property of a PDE model

Model initial conditions, specified as the `InitialConditions` property of a PDE model. Initial conditions can be complex numbers.

Example: `model.InitialConditions`

`regiontype` — Geometric region type
`'edge'` for a 2-D model | `'face'` for a 2-D model or 3-D model | `'cell'` for a 3-D model

Geometric region type, specified as `'edge'` for a 2-D model, `'face'` for a 2-D model or 3-D model, or `'cell'` for a 3-D model.

Example: `ca = findInitialConditions(ics,'face',[1,3])`

Data Types: `char`

`regionid` — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. View the subdomain labels for a 2-D model using `pdegplot(model,'SubdomainLabels','on')`. Currently, there are no subdomains for 3-D models, so the only acceptable value for a 3-D model is `1`.

Example: `ca = findInitialConditions(ics,'face',[1,3])`

Data Types: `double`

Output Arguments

`ic` — Initial condition assignment
GeometricInitialConditions object

Initial condition assignment, returned as a GeometricInitialConditions object.

More About

- "Initial Conditions"

See Also

GeometricInitialConditions Properties | setInitialConditions

Introduced in R2016a

generateMesh

Create triangular or tetrahedral mesh

Compatibility

`generateMesh` APPLIES TO BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
generateMesh(model)
generateMesh(model,Name,Value)
mesh = generateMesh( ___ )
```

Description

`generateMesh(model)` creates a mesh and stores it in the `model` object. `model` must contain geometry. To include 2-D geometry in a model, use `geometryFromEdges`. To include 3-D geometry, use `importGeometry` or `geometryFromMesh`.

`generateMesh(model,Name,Value)` modifies the mesh creation according to the `Name,Value` arguments.

`mesh = generateMesh( ___ )` additionally returns the mesh to the MATLAB workspace, using any of the previous syntaxes.

Examples

Generate 2-D Mesh

Generate the default 2-D mesh for the L-shaped geometry.

Create a `pde` model and include the L-shaped geometry.

```
model = createpde(1);
geometryFromEdges(model,@lshapeg);
```

Generate the default mesh for the geometry.

```
generateMesh(model);
```

View the mesh.

```
pdeplot(model)
```
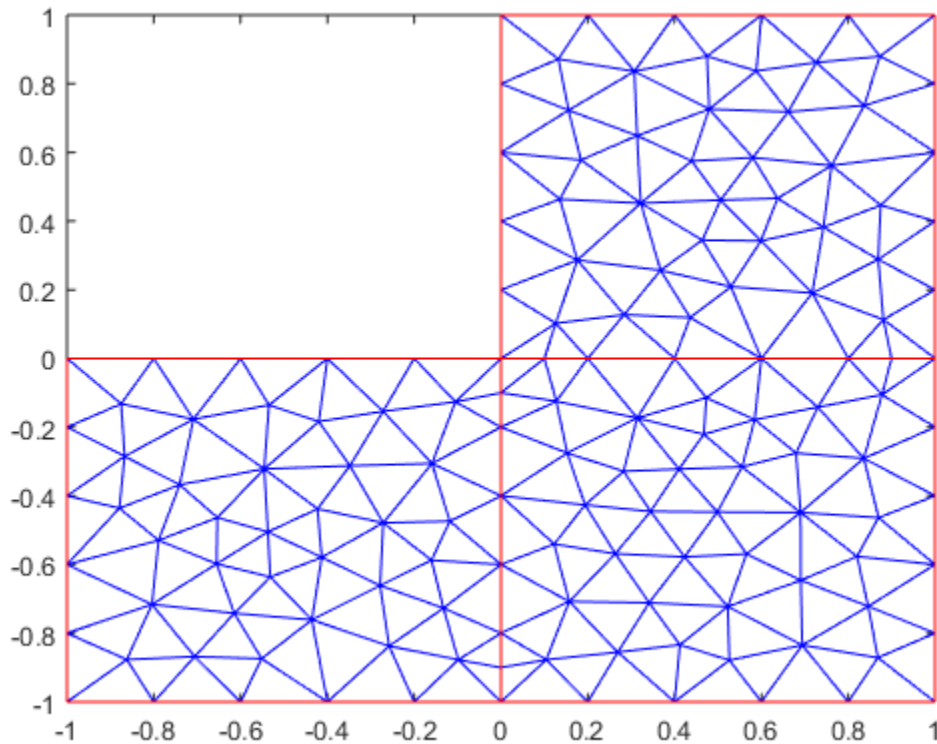

Generate 3-D Mesh

Create a mesh that is finer than the default.

Create a `pde` model and include `BracketTwoHoles` geometry.

```
model = createpde(1);
importGeometry(model,'BracketTwoHoles.stl');
```

Generate a default mesh for comparison.

```
generateMesh(model)

ans =

  pdeFEMesh with properties:

            Nodes: [3x19274 double]
         Elements: [10x11479 double]
    MaxElementSize: 7.3485
    MinElementSize: 2.9394
    GeometricOrder: 'quadratic'
```

View the mesh.

```
pdeplot3D(model)
```
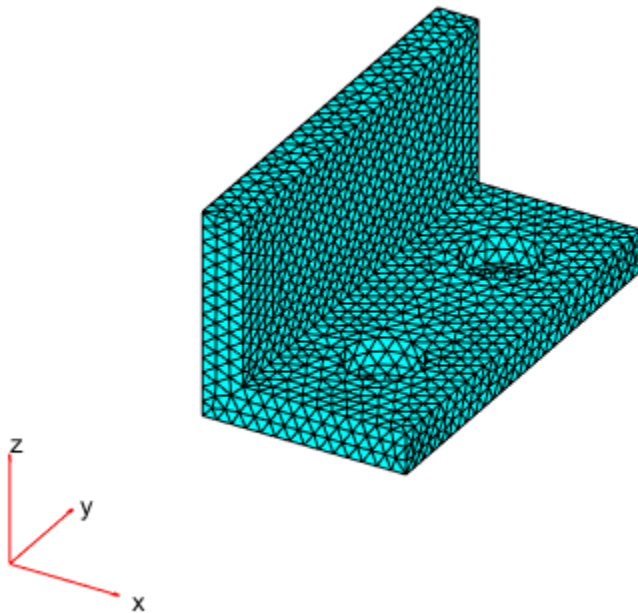

Create a mesh with target maximum element size 5 instead of the default 7.3485.

```
generateMesh(model,'Hmax',5)
```

```
ans =

  pdeFEMesh with properties:

            Nodes: [3x65926 double]
         Elements: [10x43212 double]
    MaxElementSize: 5
    MinElementSize: 2
    GeometricOrder: 'quadratic'
```
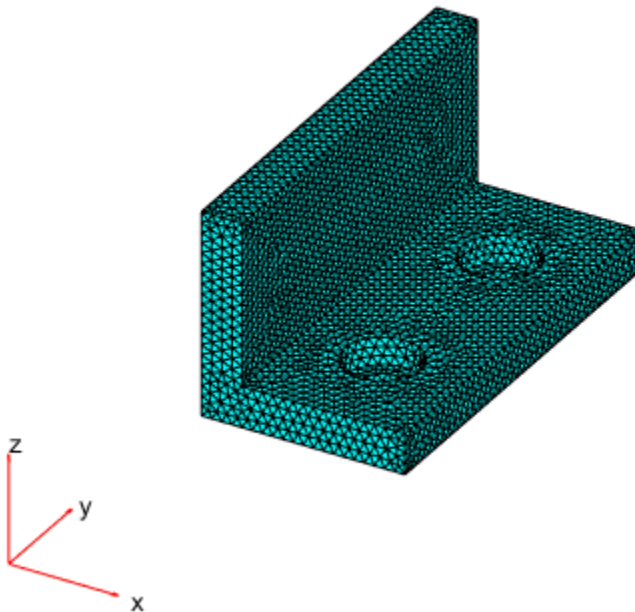
View the mesh.

```
pdeplot3D(model)
```


Input Arguments

model — PDE model
PDEModel object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde(1)`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `generateMesh(model,'Hmax',0.25);`

3-D Geometry

`'GeometricOrder'` — Element type
`'quadratic'` (default) | `'linear'`

Element type, specified as the string `'linear'` or `'quadratic'`.

In general, `'quadratic'` elements produce more accurate solutions, but use more memory. Override the default `'quadratic'` only to save memory.

Example: `generateMesh(model,'GeometricOrder','linear');`

Data Types: `char`

`'Hmax'` — Target maximum mesh edge length
positive real number

Target maximum mesh edge length, specified as a positive real number. `Hmax` is an approximate upper bound on the mesh edge lengths. `generateMesh` can occasionally create a mesh with some elements that exceed `Hmax` by a few percent. `generateMesh` estimates the default value of `Hmax` from the geometry.

Example: `generateMesh(model,'Hmax',0.25);`

Data Types: `double`

`'Hmin'` — Target minimum mesh edge length
`0` (default) | nonnegative real number

Target minimum mesh edge length, specified as a nonnegative real number.

Hmin is an approximate lower bound on the mesh edge lengths. generateMesh can occasionally create a mesh with some elements that are a few percent smaller than Hmin. generateMesh estimates the default value of Hmin from the geometry.

Example: generateMesh(model,'Hmin',0.05);

Data Types: double

2-D Geometry

`'GeometricOrder'` — Element type
'linear' (default) | 'quadratic'

Element type, specified as the string 'linear' or 'quadratic'.

In general, 'quadratic' elements produce more accurate solutions, but use more memory.

Note: Only the solvepde and solvepdeeig solvers use quadratic 2-D elements. Other solvers can only accept a linear triangular mesh.

Example: generateMesh(model,'GeometricOrder','quadratic');

Data Types: char

`'Hgrad'` — Mesh growth rate
1.3 (default) | scalar strictly between 1 and 2

Mesh growth rate, specified as a scalar strictly between 1 and 2.

Example: generateMesh(model,'Hgrad',1.5);

Data Types: double

`'Hmax'` — Target maximum mesh edge length
positive real number

Target maximum mesh edge length, specified as a positive real number. Hmax is an approximate upper bound on the mesh edge lengths. generateMesh can occasionally create a mesh with some elements that exceed Hmax by a few percent. generateMesh estimates the default value of Hmax from the geometry.

Example: `generateMesh(model,'Hmax',0.25);`

Data Types: `double`

`'Jiggle'` — Mesh quality improvement
`'mean'` (default) | `'on'` | `'off'` | `'minimum'`

Mesh quality improvement, specified as `'mean'`, `'off'`, `'minimum'`, or `'on'`.

After creating the mesh, the meshing algorithm calls `jigglemesh`, with the `Opt` name-value pair set to the stated value. Exceptions: `'off'` means do not call `jigglemesh`, and `'on'` means call `jigglemesh` with `Opt = 'off'`.

Example: `generateMesh(model,'Jiggle','on');`

Data Types: `char`

`'JiggleIter'` — Maximum jiggle iterations
10 (default) | positive integer

Maximum jiggle iterations, specified as a positive integer.

Example: `generateMesh(model,'JiggleIter',5);`

Data Types: `double`

`'MesherVersion'` — Meshing algorithm
`'preR2013a'` (default) | `'R2013a'`

Meshing algorithm, specified as `'preR2013a'` or `'R2013a'`.

Example: `generateMesh(model,'MesherVersion','R2013a');`

Data Types: `char`

Output Arguments

`mesh` — Mesh description
`FEMesh` object

Mesh description, returned as an `FEMesh` object. `mesh` is the same as `model.Mesh`.

More About

Element

An *element* is a basic unit in the finite-element method.

For 2-D problems, an element is a triangle t in the [p,e,t] "Mesh Data" on page 2-211 structure or in the model.Mesh.Element property. If the triangle represents a linear element, it has nodes only at the triangle corners. If the triangle represents a quadratic element, then it has nodes at the triangle corners and edge centers.

For 3-D problems, an element is a tetrahedron with either four or ten points. A four-point (linear) tetrahedron has nodes only at its corners. A ten-point (quadratic) tetrahedron has nodes at its corners and at the center point of each edge. For a sketch of the two tetrahedra, see "Mesh Data" on page 2-211.

The [p,e,t] data structure for an element t has the form [p1;p2;...;pn;sd], where the p values are indexes of the nodes (points p in t), and sd is the subdomain number.

- "Solve Problems Using PDEModel Objects" on page 2-14
- "Finite Element Basis for 3-D" on page 5-10
- "Mesh Data" on page 2-211

See Also
FEMesh Properties | geometryFromEdges | importGeometry | PDEModel

Introduced in R2015a

GeometricInitialConditions Properties

Initial conditions

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW.

Description

A `GeometricInitialConditions` object contains a description of the initial conditions. A `PDEModel` container has a vector of `GeometricInitialConditions` objects in its `InitialConditions.InitialConditionAssignments` property.

Set initial conditions for your model using the `setInitialConditions` function.

Properties

`RegionType` — Region type
`'face'` | `'cell'`

Region type, returned as `'face'` for a 2-D region, or `'cell'` for a 3-D region.

Data Types: `char`

`RegionID` — Region ID
vector of positive integers

Region ID, returned as a vector of positive integers. To determine which ID corresponds to which portion of the geometry, use the `pdegplot` function. Set the `'SubdomainLabels'` name-value pair to `'on'`.

Data Types: `double`

`InitialValue` — Initial value
scalar | vector | function handle

Initial value, returned as a scalar, vector, or function handle. For details, see `setInitialConditions`.

Data Types: `double | function_handle`
Complex Number Support: Yes

`InitialDerivative` — Initial derivative
scalar | vector | function handle

Initial derivative, returned as a scalar, vector, or function handle. For details, see `setInitialConditions`.

Data Types: `double | function_handle`
Complex Number Support: Yes

See Also
`findInitialConditions` | `setInitialConditions`

Related Examples
- "Set Initial Conditions" on page 2-155
- "View, Edit, and Delete Initial Conditions" on page 2-158
- "Solve Problems Using PDEModel Objects" on page 2-14

Introduced in R2016a

geometryFromEdges

Create 2-D geometry

Compatibility

CREATING 2-D GEOMETRIES IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
geometryFromEdges(model,g)
pg = geometryFromEdges(model,g)
```

Description

`geometryFromEdges(model,g)` adds the 2-D geometry described in `g` to the `model` container.

`pg = geometryFromEdges(model,g)` additionally returns the geometry to the Workspace.

Examples

Geometry from Decomposed Solid Geometry

Create a decomposed solid geometry model and include it in a PDE model.

Create a default scalar PDE model.

```
model = createpde;
```

Define a circle in a rectangle, place these in one matrix, and create a set formula that subtracts the circle from the rectangle.

```
R1 = [3,4,-1,1,1,-1,0.5,0.5,-0.75,-0.75]';
C1 = [1,0.5,-0.25,0.25]';
C1 = [C1;zeros(length(R1) - length(C1),1)];
gm = [R1,C1];
sf = 'R1-C1';
```

Create the geometry.

```
ns = char('R1','C1');
ns = ns';
g = decsg(gm,sf,ns);
```

Include the geometry in the model and plot it.

```
geometryFromEdges(model,g);
pdegplot(model,'EdgeLabels','on')
axis equal
xlim([-1.1,1.1])
```
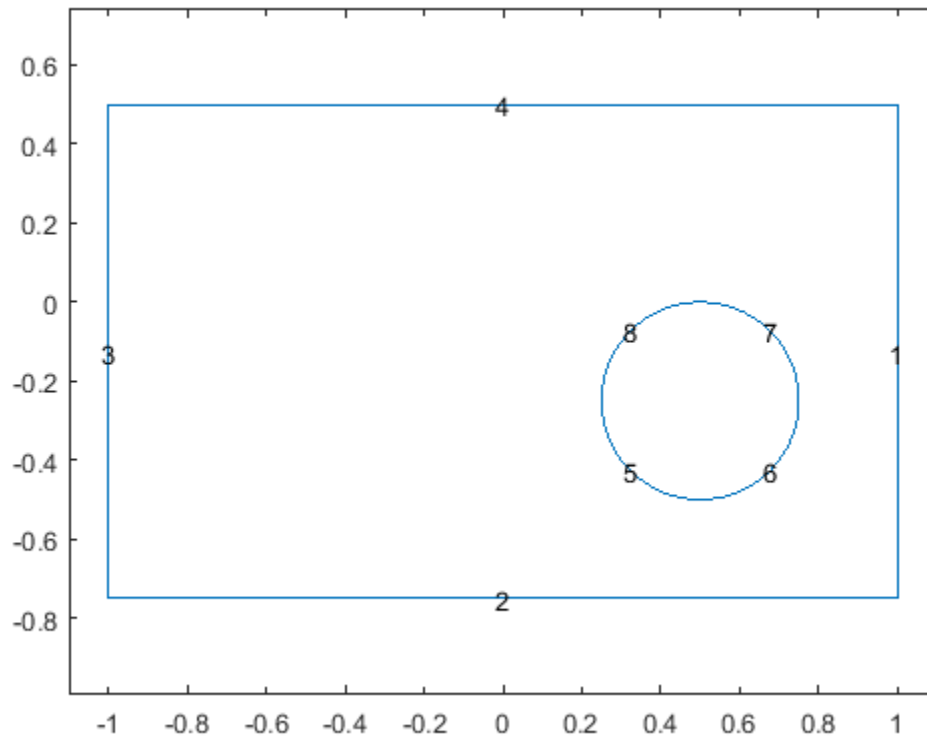
- "Solve PDEs with Constant Boundary Conditions" on page 2-185

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

g — Geometry description
decomposed geometry matrix | name of a geometry function | handle to a geometry function

Geometry description, specified as a decomposed geometry matrix, as the name of a geometry function, or as a handle to a geometry function. For details, see "Create 2-D Geometry" on page 2-17.

Example: `geometryFromEdges(model,@circleg)`

Data Types: `double` | `char` | `function_handle`

Output Arguments

pg — Geometry object
`AnalyticGeometry` object

Geometry object, returned as an `AnalyticGeometry` object. This object is stored in `model.Geometry`.

More About

- "Solve Problems Using PDEModel Objects" on page 2-14
- "2-D Geometry"

See Also

AnalyticGeometry Properties | `PDEModel`

Introduced in R2015a

geometryFromMesh

Create 3-D geometry from a triangulated mesh

Compatibility

CREATING 3-D GEOMETRIES IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
geometryFromMesh(model,nodes,elements)
[G,mesh] = geometryFromMesh(model,nodes,elements)
```

Description

`geometryFromMesh(model,nodes,elements)` creates geometry within `model`. If `elements` represents a tetrahedral volume mesh, `geometryFromMesh` incorporates `nodes` and `elements` in the `model.Mesh.Nodes` and `model.Mesh.Elements` properties respectively.

To replace the imported mesh with a mesh having different target element size, use `generateMesh`.

If elements represents a boundary triangular mesh, `geometryFromMesh` creates only the geometry from the mesh. To generate a mesh in this case, use `generateMesh`.

`[G,mesh] = geometryFromMesh(model,nodes,elements)` returns a handle `G` to the geometry in `model.Geometry`, and a handle `mesh` to the mesh in `model.Mesh`.

Examples

Geometry from Volume Mesh

Import a tetrahedral mesh into a PDE model.

Load a tetrahedral mesh into your workspace. The `tetmesh` file ships with your software. Put the data in the correct shape for `geometryFromMesh`.

```
load tetmesh
nodes = X';
elements = tet';
```

Create a PDE model and import the mesh into the model.

```
model = createpde();
geometryFromMesh(model,nodes,elements);
```

View the geometry and face numbers.

```
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
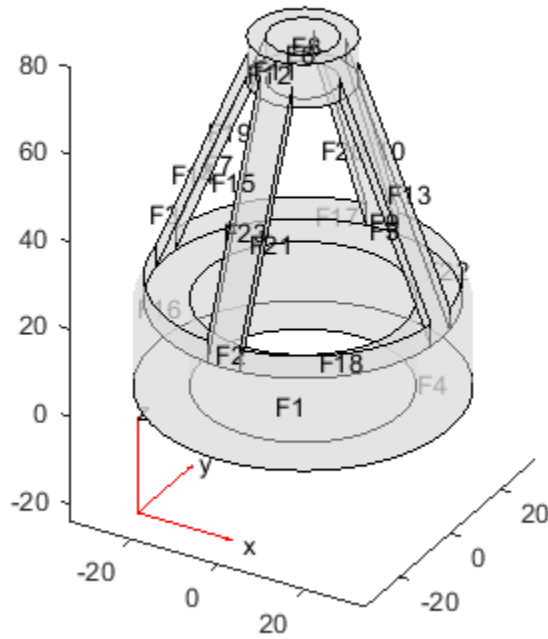
Geometry from Convex Hull

Create a geometric block from the convex hull of a mesh grid of points.

Create a 3-D mesh grid.

```
[x,y,z] = meshgrid(-2:4:2);
```

Create the convex hull.

```
x = x(:);
y = y(:);
z = z(:);
```

```
K = convhull(x,y,z);
```

Put the data in the correct shape for `geometryFromMesh`.

```
nodes = [x';y';z'];
elements = K';
```

Create a PDE model and import the mesh.

```
model = createpde();
geometryFromMesh(model,nodes,elements);
```

View the geometry and face numbers.

```
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
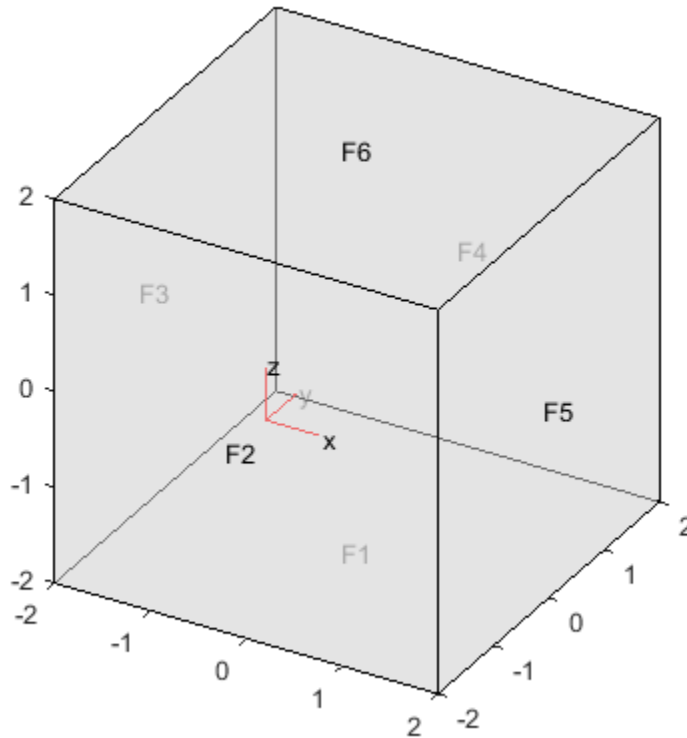
Geometry from alphaShape

Create an `alphaShape` object of a block with a cylindrical hole. Import the geometry into a PDE model from the `alphaShape` boundary.

Create a 2-D mesh grid.

```
[xg, yg] = meshgrid(-3:0.25:3);
xg = xg(:);
yg = yg(:);
```

Create a unit disk. Remove all the mesh grid points that fall inside the unit disk, and include the unit disk points.

```
t = (pi/24:pi/24:2*pi)';
x = cos(t);
y = sin(t);
circShp = alphaShape(x,y,2);
in = inShape(circShp,xg,yg);
xg = [xg(~in); cos(t)];
yg = [yg(~in); sin(t)];
```

Create 3-D copies of the remaining mesh grid points, with the *z*-coordinates ranging from 0 through 1. Combine the points into an alphaShape object.

```
zg = ones(numel(xg),1);
xg = repmat(xg,5,1);
yg = repmat(yg,5,1);
zg = zg*(0:.25:1);
zg = zg(:);
shp = alphaShape(xg,yg,zg);
```

Obtain a surface mesh of the alphaShape object.

```
[elements,nodes] = boundaryFacets(shp);
```

Put the data in the correct shape for geometryFromMesh.

```
nodes = nodes';
elements = elements';
```

Create a PDE model and import the surface mesh.

```
model = createpde();
geometryFromMesh(model,nodes,elements);
```

View the geometry and face numbers.

```
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
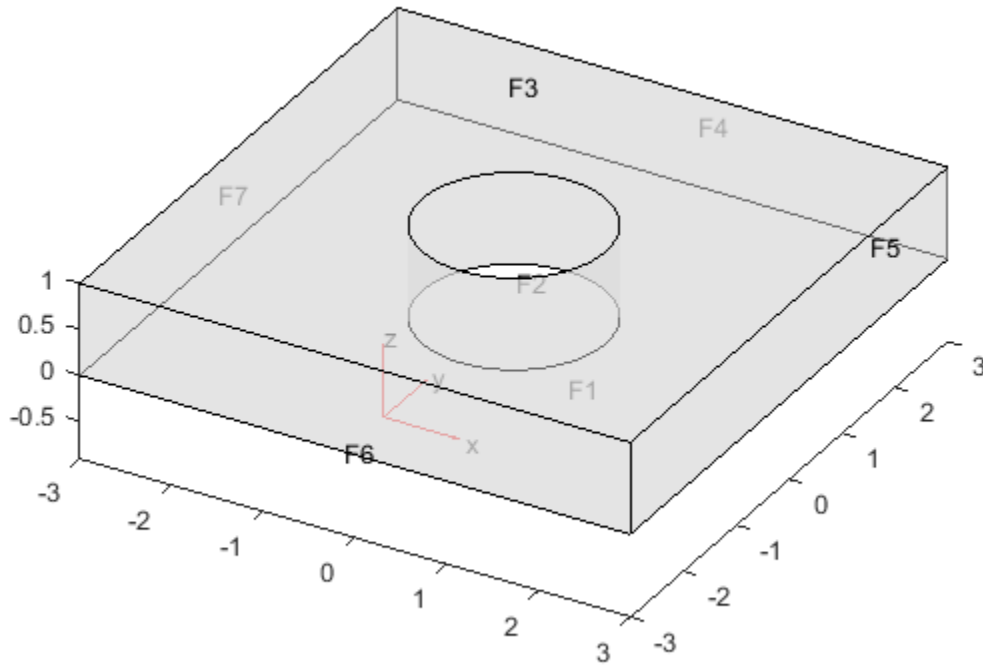
To use the geometry in an analysis, create a volume mesh.

```
generateMesh(model);
```

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

nodes — Mesh nodes
3-by-Nnodes real matrix

Mesh nodes, specified as a 3-by-Nnodes real matrix, where Nnodes is the number of nodes in the mesh. Node j has *x*, *y*, and *z* coordinates in column j of nodes.

Data Types: double

elements — Mesh elements
3-by-Nelements integer matrix | 4-by-Nelements integer matrix | 10-by-Nelements integer matrix

Mesh elements, specified as an integer matrix with 3, 4, or 10 rows, and Nelements columns, where Nelements is the number of elements in the mesh.

- A mesh on the geometry surface has size 3-by-Nelements. Each column of elements contains the indices of the triangle corner nodes for a surface element. In this case, the resulting geometry does not contain a full mesh. Create the mesh using the generateMesh function.
- Linear elements have size 4-by-Nelements. Each column of elements contains the indices of the tetrahedral corner nodes for an element.
- Quadratic elements have size 10-by-Nelements. Each column of elements contains the indices of the tetrahedral corner nodes and the tetrahedral edge midpoint nodes for an element.

For mesh node numbering details of linear or quadratic elements, see "Mesh Data" on page 2-211.

Data Types: double

Output Arguments

G — Geometry
handle to model.Geometry

Geometry, returned as a handle to model.Geometry. This geometry is of class DiscreteGeometry.

mesh — Finite element mesh
handle to model.Mesh

Finite element mesh, returned as a handle to `model.Mesh`.

- If `elements` represent a surface mesh (`elements` is a 3-by-`Nelements` matrix) then `mesh` is `[]`. In this case, create a mesh for the geometry using the `generateMesh` function.

- If `elements` represent a volume mesh (`elements` has more than three rows), then `mesh` has the same nodes and elements as the inputs. You can get a different mesh for the geometry by using the `generateMesh` function.

See Also

Using alphaShape Objects | DiscreteGeometry Properties | `generateMesh` | `importGeometry`

Introduced in R2015b

hyperbolic

Solve hyperbolic PDE problem

Hyperbolic equation solver

Solves PDE problems of the type

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

on a 2-D or 3-D region Ω, or the system PDE problem

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

The variables c, a, f, and d can depend on position, time, and the solution u and its gradient.

Compatibility

hyperbolic IS NOT RECOMMENDED. Use `solvepde` instead.

Syntax

```
u = hyperbolic(u0,ut0,tlist,model,c,a,f,d)
u = hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d)
u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M)
u = hyperbolic( ___ ,rtol)
u = hyperbolic( ___ ,rtol,atol)
u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M, ___ ,'DampingMatrix',D)
u = hyperbolic( ___ ,'Stats','off')
```

Description

`u = hyperbolic(u0,ut0,tlist,model,c,a,f,d)` produces the solution to the FEM formulation of the scalar PDE problem

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

on a 2-D or 3-D region Ω, or the system PDE problem

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

with geometry, mesh, and boundary conditions specified in `model`, with initial value `u0` and initial derivative with respect to time `ut0`. The variables c, a, f, and d in the equation correspond to the function coefficients `c`, `a`, `f`, and `d` respectively.

`u = hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d)` solves the problem using boundary conditions `b` and finite element mesh specified in `[p,e,t]`.

`u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M)` solves the problem based on finite element matrices that encode the equation, mesh, and boundary conditions.

`u = hyperbolic( ___ ,rtol)` and `u = hyperbolic( ___ ,rtol,atol)` modify the solution process by passing to the ODE solver a relative tolerance `rtol`, and optionally an absolute tolerance `atol`.

`u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M, ___ ,'DampingMatrix',D)` modifies the problem to include a damping matrix `D`.

`u = hyperbolic( ___ ,'Stats','off')` turns off the display of internal ODE solver statistics during the solution process.

Examples

Hyperbolic Equation

Solve the wave equation

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

on the square domain specified by `squareg`.

Create a PDE model and import the geometry.

```
model = createpde;
geometryFromEdges(model,@squareg);
pdegplot(model,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```


Set Dirichlet boundary conditions $u = 0$ for $x = \pm 1$, and Neumann boundary conditions

$$\nabla u \cdot \mathbf{n} = 0$$
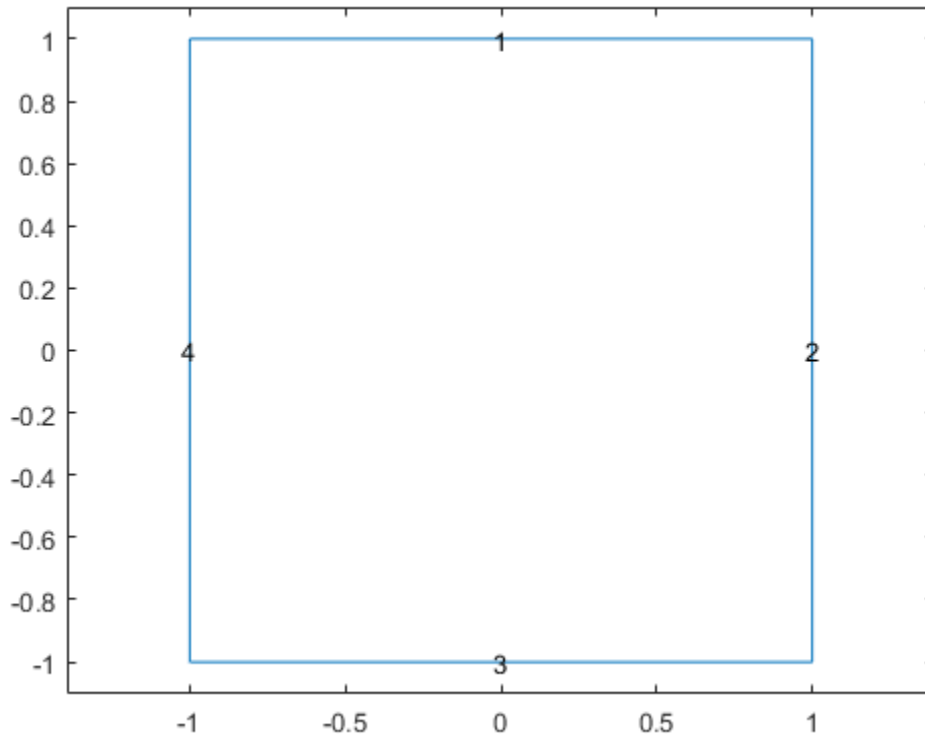
for $y = \pm 1$. (The Neumann boundary condition is the default condition, so the second specification is redundant.)

```
applyBoundaryCondition(model,'Edge',[2,4],'u',0);
applyBoundaryCondition(model,'Edge',[1,3],'g',0);
```

Set the initial conditions

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

Set the solution times.

```
tlist = linspace(0,5,31);
```

Give coefficients for the problem.

```
c = 1;
a = 0;
f = 0;
d = 1;
```

Generate a mesh and solve the PDE.

```
generateMesh(model,'Hmax',0.1);
u1 = hyperbolic(u0,ut0,tlist,model,c,a,f,d);
```

```
549 successful steps
69 failed attempts
1238 function evaluations
1 partial derivatives
172 LU decompositions
1237 solutions of linear systems
```

Plot the solution at the first and last times.

```
figure
pdeplot(model,'xydata',u1(:,1))
figure
pdeplot(model,'xydata',u1(:,end))
```
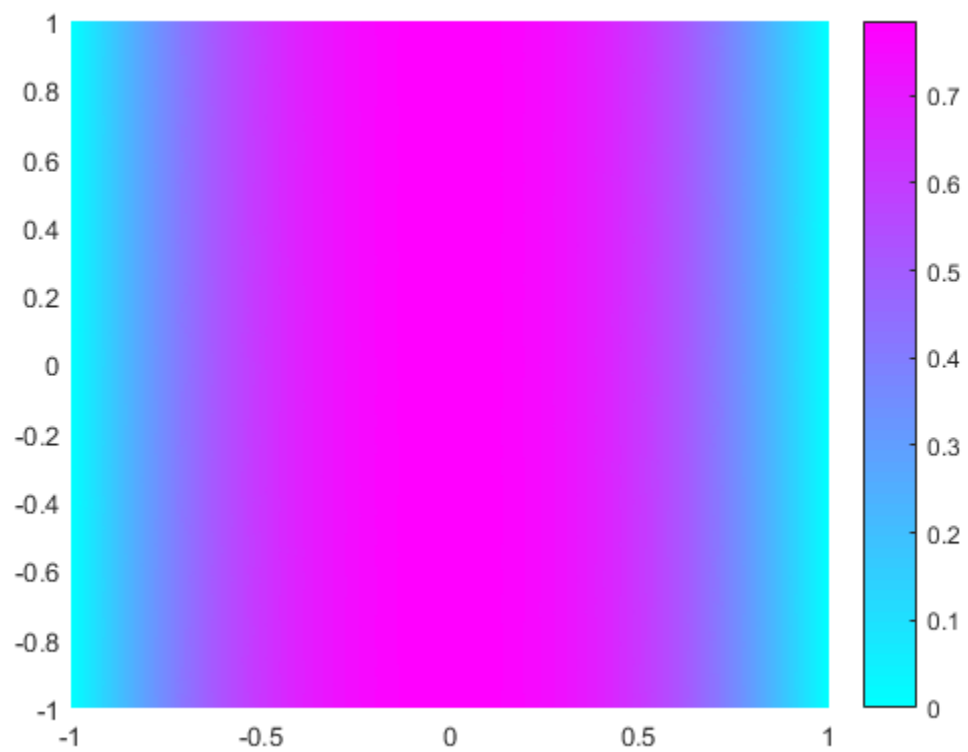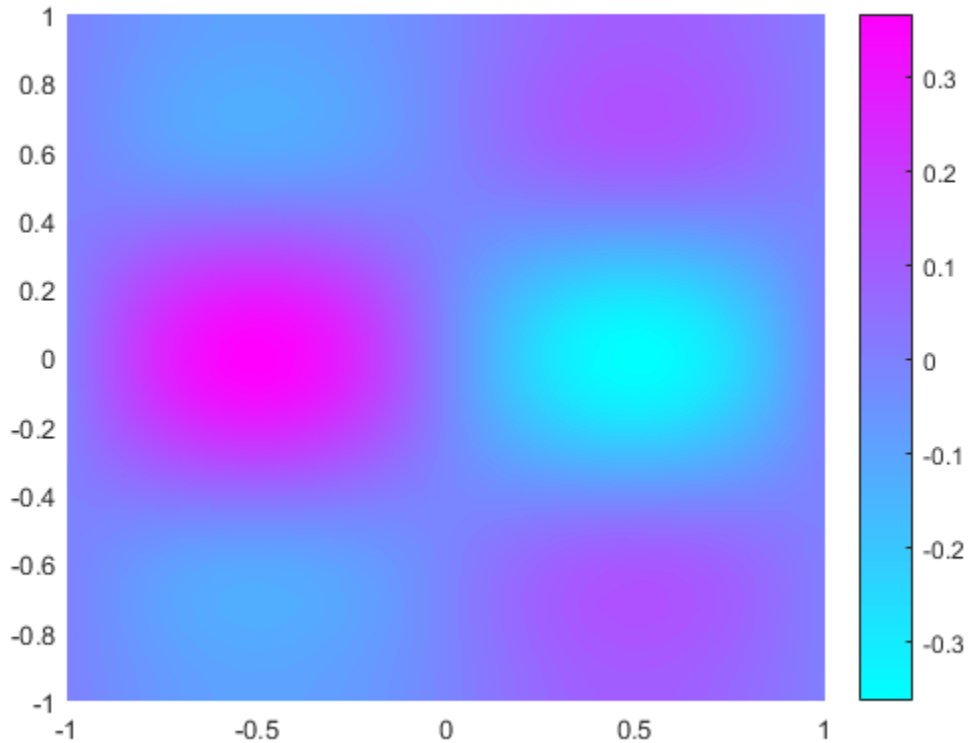
For a version of this example with animation, run `pdedemo6`.

Hyperbolic Equation using Legacy Syntax

Solve the wave equation

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

on the square domain specified by `squareg`, using a geometry function to specify the geometry, a boundary function to specify the boundary conditions, and using `initmesh` to create the finite element mesh.

Specify the geometry as `@squareg` and plot the geometry.

```
g = @squareg;
pdegplot(g,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```


Set Dirichlet boundary conditions $u = 0$ for $x = \pm 1$, and Neumann boundary conditions

$$\nabla u \cdot \mathbf{n} = 0$$
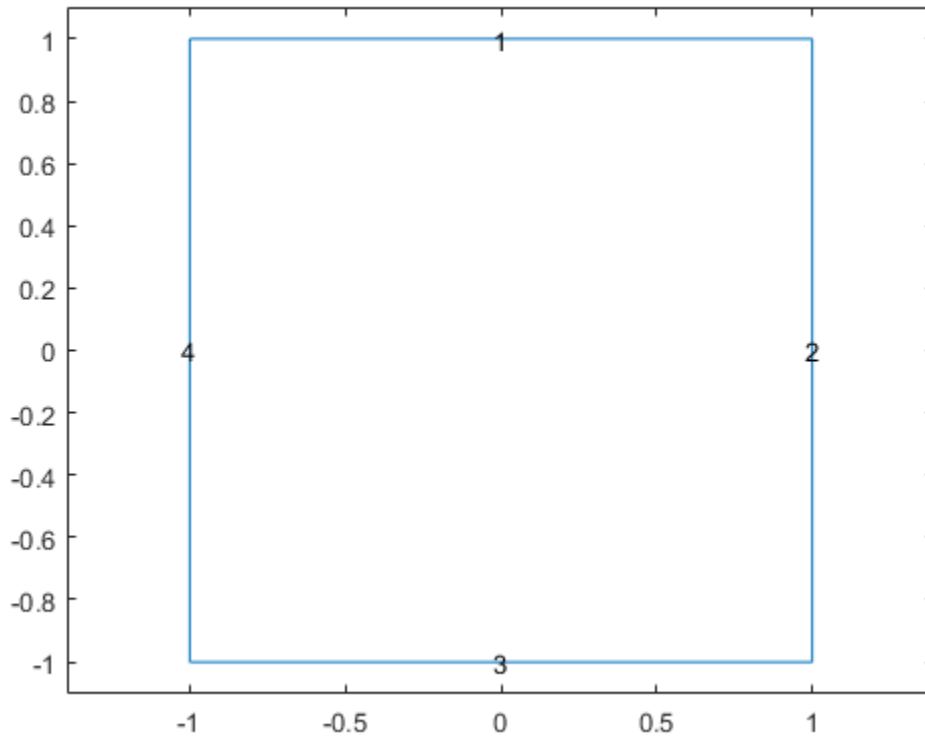
for $y = \pm 1$. (The Neumann boundary condition is the default condition, so the second specification is redundant.)

The `squareb3` function specifies these boundary conditions.

```
b = @squareb3;
```

Set the initial conditions

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

Set the solution times.

```
tlist = linspace(0,5,31);
```

Give coefficients for the problem.

```
c = 1;
a = 0;
f = 0;
d = 1;
```

Create a mesh and solve the PDE.

```
[p,e,t] = initmesh(g);
u = hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d);

462 successful steps
70 failed attempts
1066 function evaluations
1 partial derivatives
156 LU decompositions
1065 solutions of linear systems
```

Plot the solution at the first and last times.

```
figure
pdeplot(p,e,t,'xydata',u(:,1))
figure
pdeplot(p,e,t,'xydata',u(:,end))
```
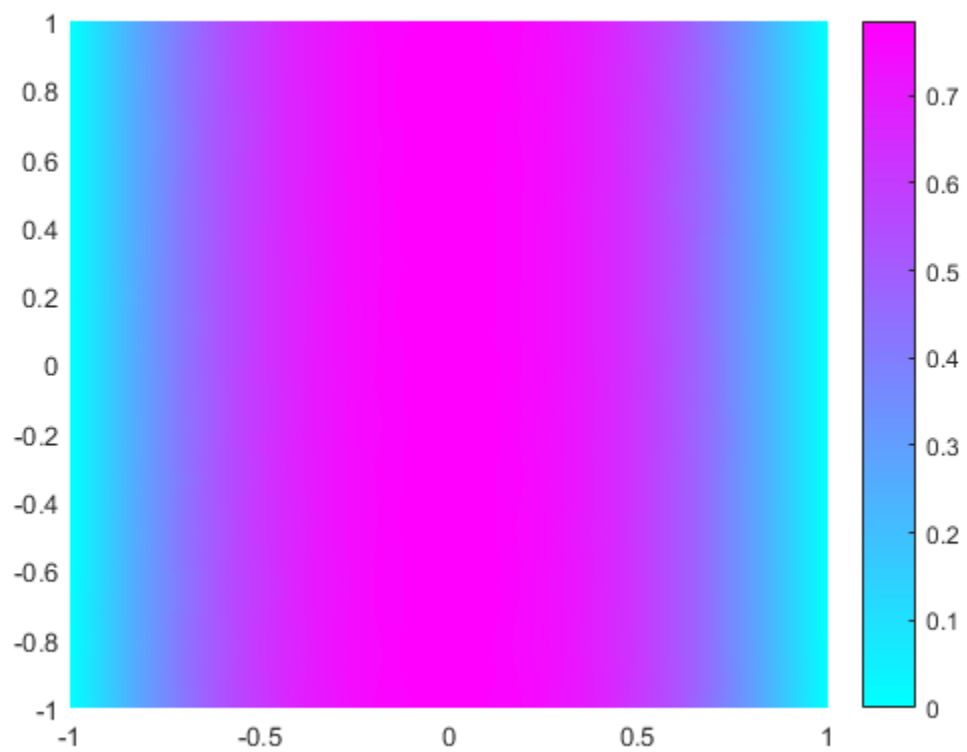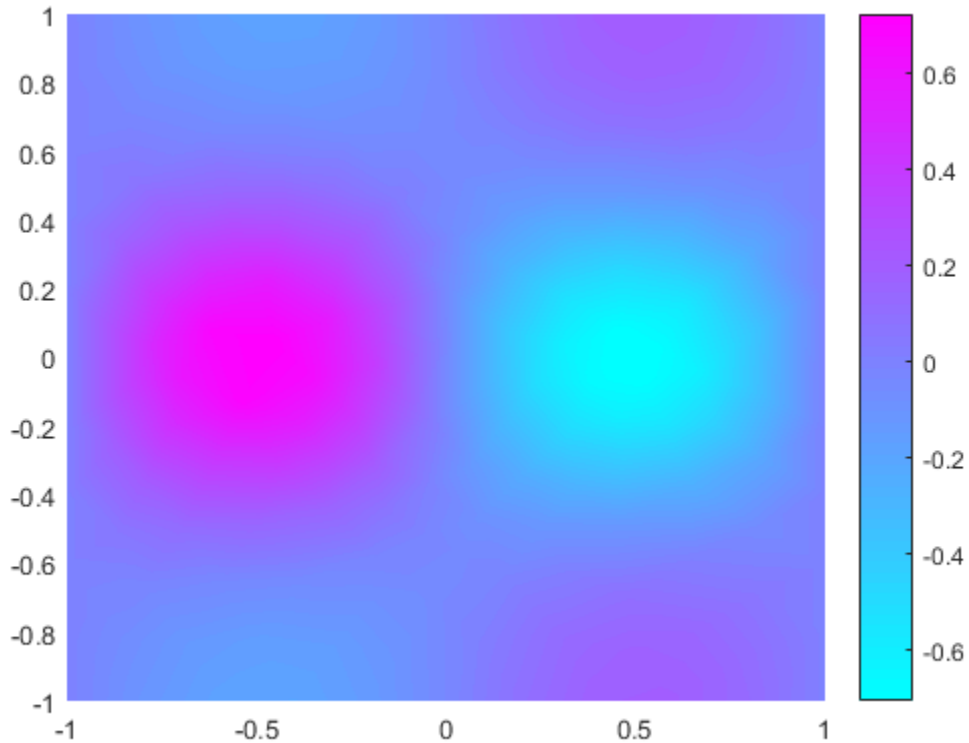
For a version of this example with animation, run `pdedemo6`.

Hyperbolic Solution Using Finite Element Matrices

Solve a hyperbolic problem using finite element matrices.

Create a model and import the `'BracketWithHole.stl'` geometry.

```
model = createpde();
importGeometry(model,'BracketWithHole.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```

Set coefficients $c = 1$, $a = 0$, $f = 0.5$, and $d = 1$.

```
c = 1;
a = 0;
f = 0.5;
d = 1;
```

Generate a mesh for the model.

```
generateMesh(model);
```

Create initial conditions and boundary conditions. The boundary condition for the rear face (face 3) is Dirichlet with value 0. All other faces have the default boundary condition.

The initial condition is $u(0) = 0$, $du/dt(0) = x/2$. Give the initial condition on the derivative by calculating the x-position of each node in `xpts`, and passing $x/2$.

```
applyBoundaryCondition(model,'face',3,'u',0);
u0 = 0;
xpts = model.Mesh.Nodes(1,:);
ut0 = xpts(:)/2;
```

Create the associated finite element matrices.

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Solve the PDE for times from 0 to 2.

```
tlist = linspace(0,5,50);
u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M);
```

```
1486 successful steps
64 failed attempts
2981 function evaluations
1 partial derivatives
278 LU decompositions
2980 solutions of linear systems
```

View the solution at a few times. Scale all the plots to have the same color range by using the `caxis` command.

```
umax = max(max(u));
umin = min(min(u));

subplot(2,2,1)
pdeplot3D(model,'colormapdata',u(:,5));
caxis([umin umax]);
title('Time 1/2')
subplot(2,2,2)
pdeplot3D(model,'colormapdata',u(:,10));
caxis([umin umax]);
title('Time 1')
subplot(2,2,3)
pdeplot3D(model,'colormapdata',u(:,15));
caxis([umin umax]);
title('Time 3/2')
subplot(2,2,4)
```

```
pdeplot3D(model,'colormapdata',u(:,20));
caxis([umin umax]);
title('Time 2')
```
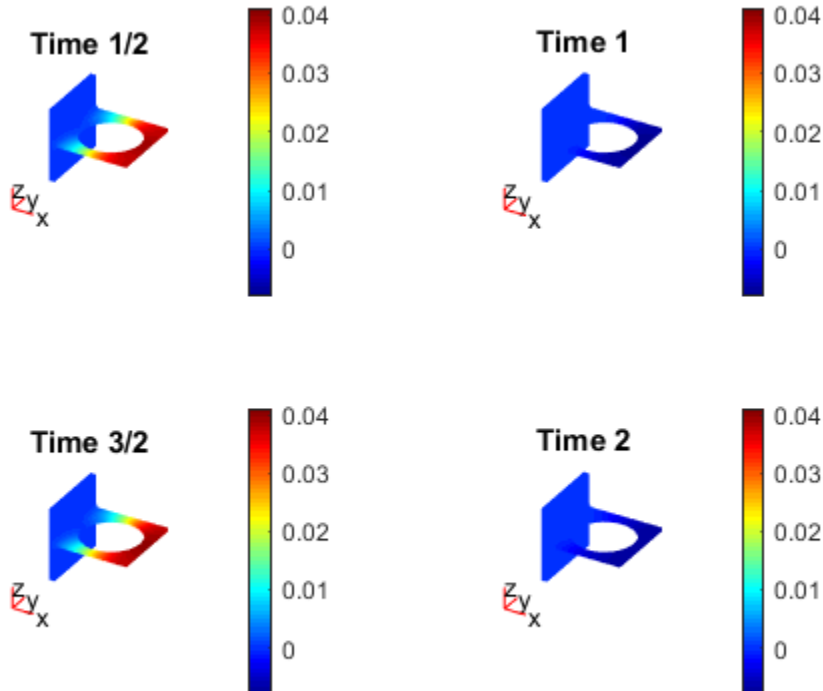

The solution seems to have a frequency of one, because the plots at times 1/2 and 3/2 show maximum values, and those at times 1 and 2 show minimum values.

Hyperbolic Equation with Damping

Solve a hyperbolic problem that includes damping. You must use the finite element matrix form to use damping.

Create a model and import the 'BracketWithHole.stl' geometry.

6-171

```
model = createpde();
importGeometry(model,'BracketWithHole.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```


Set coefficients $c = 1$, $a = 0$, $f = 0.5$, and $d = 1$.

```
c = 1;
a = 0;
f = 0.5;
d = 1;
```

Generate a mesh for the model.

```
generateMesh(model);
```

Create initial conditions and boundary conditions. The boundary condition for the rear face (face 3) is Dirichlet with value 0. All other faces have the default boundary condition. The initial condition is $u(0) = 0$, $du/dt(0) = x/2$. Give the initial condition on the derivative by calculating the *x*-position of each node in `xpts`, and passing *x*/2.

```
applyBoundaryCondition(model,'face',3,'u',0);
u0 = 0;
xpts = model.Mesh.Nodes(1,:);
ut0 = xpts(:)/2;
```

Create the associated finite element matrices.

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Use a damping matrix that is 10% of the mass matrix.

```
Damping = 0.1*M;
```

Solve the PDE for times from 0 to 2.

```
tlist = linspace(0,5,50);
u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M,'DampingMatrix',Damping);
```

```
1419 successful steps
65 failed attempts
2745 function evaluations
1 partial derivatives
268 LU decompositions
2744 solutions of linear systems
```

Plot the maximum value at each time.

```
plot(max(u))
xlabel('Time')
ylabel('Maximum value')
title('Maximum of Solution')
```
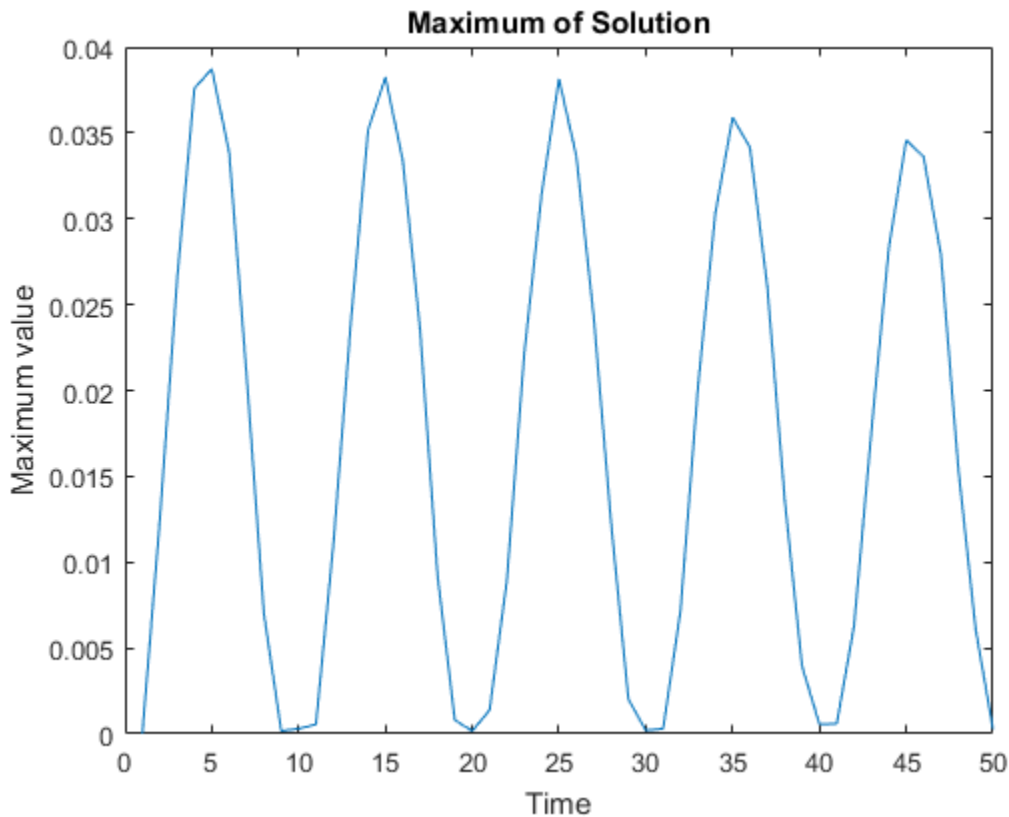
The oscillations damp slightly as time increases.

Input Arguments

u0 — Initial condition
vector | text expression

Initial condition, specified as a scalar, vector of nodal values, or text expression. The initial condition is the value of the solution u at the initial time, specified as a column vector of values at the nodes. The nodes are either p in the [p,e,t] data structure, or are model.Mesh.Nodes. For details, see "Solve PDEs with Initial Conditions" on page 2-162.

- If the initial condition is a constant scalar `v`, specify `u0` as `v`.

- If there are `Np` nodes in the mesh, and *N* equations in the system of PDEs, specify `u0` as a column vector of `Np`*N* elements, where the first `Np` elements correspond to the first component of the solution `u`, the second `Np` elements correspond to the second component of the solution `u`, etc.

- Give a text expression of a function, such as `'x.^2 + 5*cos(x.*y)'`. If you have a system of *N* > 1 equations, give a text array such as

```
char('x.^2 + 5*cos(x.*y)',...
     'tanh(x.*y)./(1+z.^2)')
```

Example: `x.^2+5*cos(y.*x)`

Data Types: `double` | `char`
Complex Number Support: Yes

`ut0` — Initial derivative
vector | text expression

Initial derivative, specified as a vector or text expression. The initial gradient is the value of the derivative of the solution `u` at the initial time, specified as a vector of values at the nodes. The nodes are either `p` in the `[p,e,t]` data structure, or are `model.Mesh.Nodes`. See "Solve PDEs with Initial Conditions" on page 2-162.

- If the initial derivative is a constant value `v`, specify `u0` as `v`.

- If there are `Np` nodes in the mesh, and *N* equations in the system of PDEs, specify `ut0` as a vector of `Np`*N* elements, where the first `Np` elements correspond to the first component of the solution `u`, the second `Np` elements correspond to the second component of the solution `u`, etc.

- Give a text expression of a function, such as `'x.^2 + 5*cos(x.*y)'`. If you have a system of *N* > 1 equations, use a text array such as

```
char('x.^2 + 5*cos(x.*y)',...
     'tanh(x.*y)./(1+z.^2)')
```

For details, see "Solve PDEs with Initial Conditions" on page 2-162.

Example: `p(1,:).^2+5*cos(p(2,:).*p(1,:))`

Data Types: `double` | `char`
Complex Number Support: Yes

`tlist` — Solution times
real vector

Solution times, specified as a real vector. The solver returns the solution to the PDE at the solution times.

Example: `0:0.2:4`

Data Types: `double`

`model` — PDE model
PDEModel object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde(1)`

`c` — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. `c` represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify `c` in various ways, detailed in "c Coefficient for Systems" on page 2-125. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

a — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. a represents the a coefficient in the scalar PDE

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

or in the system of PDEs

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

There are a wide variety of ways of specifying a, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

f — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. f represents the f coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify f in various ways, detailed in "f Coefficient for Systems" on page 2-98. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

d — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. d represents the *d* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify d in various ways, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

b — Boundary conditions
boundary matrix | boundary file

Boundary conditions, specified as a boundary matrix or boundary file. Pass a boundary file as a function handle or as a string naming the file.

- A boundary matrix is generally an export from the PDE app. For details of the structure of this matrix, see "Boundary Matrix for 2-D Geometry" on page 2-171.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-199.

For more information on boundary conditions, see "Forms of Boundary Condition Specification" on page 2-170.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

p — Mesh nodes
output of `initmesh` | output of `meshToPet`

Mesh nodes, specified as the output of `initmesh` or `meshToPet`. For the structure of a `p` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

e — Mesh edges
output of `initmesh` | output of `meshToPet`

Mesh edges, specified as the output of `initmesh` or `meshToPet`. For the structure of `e`, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

t — Mesh elements
output of `initmesh` | output of `meshToPet`

Mesh elements, specified as the output of `initmesh` or `meshToPet`. Mesh elements are the triangles or tetrahedra that form the finite element mesh. For the structure of a `t` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

6-179

Kc — Stiffness matrix
sparse matrix | full matrix

Stiffness matrix, specified as a sparse matrix or as a full matrix. See "Elliptic Equations" on page 5-2. Typically, Kc is the output of assempde.

Fc — Load vector
vector

Load vector, specified as a vector. See "Elliptic Equations" on page 5-2. Typically, Fc is the output of assempde.

B — Dirichlet nullspace
sparse matrix

Dirichlet nullspace, returned as a sparse matrix. See "Algorithms" on page 6-56. Typically, B is the output of assempde.

ud — Dirichlet vector
vector

Dirichlet vector, returned as a vector. See "Algorithms" on page 6-56. Typically, ud is the output of assempde.

M — Mass matrix
sparse matrix | full matrix

Mass matrix. specified as a sparse matrix or a full matrix. See "Elliptic Equations" on page 5-2.

To obtain the input matrices for pdeeig, hyperbolic or parabolic, run both assema and assempde:

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Note: Create the M matrix using assema with d, not a, as the argument before f.

Data Types: double
Complex Number Support: Yes

`rtol` — Relative tolerance for ODE solver
`1e-3` (default) | positive real

Relative tolerance for ODE solver, specified as a positive real.

Example: `2e-4`

Data Types: `double`

`atol` — Absolute tolerance for ODE solver
`1e-6` (default) | positive real

Absolute tolerance for ODE solver, specified as a positive real.

Example: `2e-7`

Data Types: `double`

`D` — Damping matrix
matrix

Damping matrix, specified as a matrix. `D` has the same size as the stiffness matrix `Kc` or the mass matrix `M`. When you include `D`, `hyperbolic` solves the following ODE for the variable v:

$$B^T M B \frac{d^2 v}{dt^2} + B^T D B \frac{dv}{dt} + Kv = F,$$

with initial condition `u0` and initial derivative `ut0`. Then `hyperbolic` returns the solution `u = B*v + ud`.

For an example using `D`, see "Dynamics of a Damped Cantilever Beam".

Example: `alpha*M + beta*K`

Data Types: `double`
Complex Number Support: Yes

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example:

`'Stats'` — Display ODE solver statistics
`'on'` (default) | `'off'`

Display ODE solver statistics, specified as `'on'` or `'off'`. Suppress the display by setting `Stats` to `'off'`.

Example: `x = hyperbolic(u0,ut0,tlist,model,c,a,f,d,'Stats','off')`

Data Types: `char`

Output Arguments

u — PDE solution
matrix

PDE solution, returned as a matrix. The matrix is `Np`*N-by-`T`, where `Np` is the number of nodes in the mesh, *N* is the number of equations in the PDE (*N* = 1 for a scalar PDE), and `T` is the number of solution times, meaning the length of `tlist`. The solution matrix has the following structure.

- The first `Np` elements of each column in `u` represent the solution of equation 1, then next `Np` elements represent the solution of equation 2, etc. The solution `u` is the value at the corresponding node in the mesh.
- Column `i` of `u` represents the solution at time `tlist(i)`.

To obtain the solution at an arbitrary point in the geometry, use `pdeInterpolant`.

To plot the solution, use `pdeplot` for 2-D geometry, or see "Plot 3-D Solutions and Their Gradients" on page 3-145.

More About

Algorithms

`hyperbolic` internally calls `assema`, `assemb`, and `assempde` to create finite element matrices corresponding to the problem. It calls `ode15s` to solve the resulting system of ordinary differential equations. For details, see "Hyperbolic Equations" on page 5-20.

- "PDE Problem Setup"
- "Hyperbolic Equations" on page 5-20
- "Finite Element Basis for 3-D" on page 5-10
- "Systems of PDEs" on page 5-13

See Also
solvepde

Introduced before R2006a

importGeometry

Import 3-D geometry

Compatibility

IMPORTING 3-D GEOMETRIES IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
importGeometry(model,geometryfile)
gd = importGeometry(model,geometryfile)
```

Description

`importGeometry(model,geometryfile)` creates a 3-D geometry container from the specified STL geometry file, and includes the geometry in the `model` container.

`gd = importGeometry(model,geometryfile)` additionally returns the geometry to the MATLAB Workspace.

Examples

Import Geometry into PDE Container

Import STL geometry into a PDE model.

Create a `PDEModel` container for a system of three equations.

```
model = createpde(3);
```

Import geometry into the container.

```
importGeometry(model,'ForearmLink.stl');
```

View the geometry with face labels.

```
pdegplot(model,'FaceLabels','on')
```
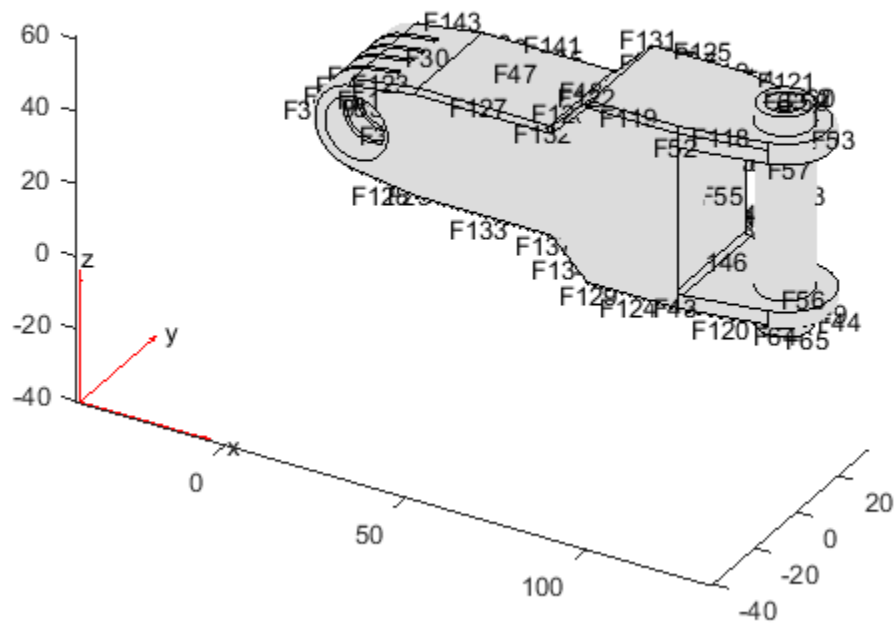

- "Create and View 3-D Geometry" on page 2-47
- "Solve Problems Using PDEModel Objects" on page 2-14

Input Arguments

model — PDE model
PDEModel object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde(1)`

geometryfile — Path to STL file
string

Path to STL file, specified as a string. The string ends with the file extension `.stl` or `.STL`.

Example: `'../geometries/Carburetor.stl'`

Data Types: `char`

Output Arguments

gd — Geometry description
`DiscreteGeometry` object

Geometry description, returned as a `DiscreteGeometry` object.

See Also
DiscreteGeometry Properties | `geometryFromMesh` | `pdegplot` | `PDEModel`

Introduced in R2015a

initmesh

Create initial 2-D mesh

Compatibility

THIS PAGE DESCRIBES THE LEGACY APPROACH. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `generateMesh`.

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 2-D geometry specification `g`. `initmesh` uses a Delaunay triangulation algorithm. The mesh size is determined from the shape of the geometry and from name-value pair settings.

`g` describes the geometry of the PDE problem. `g` can be a Decomposed Geometry matrix, the name of a Geometry file, or a function handle to a Geometry file. For details, see "2-D Geometry".

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.

`initmesh` accepts the following name/value pairs.

Name	Value	Default	Description
Hmax	numeric	estimate	Maximum edge size
Hgrad	numeric, strictly between 1 and 2	1.3	Mesh growth rate
Box	'on' | 'off'	'off'	Preserve bounding box
Init	'on' | 'off'	'off'	Edge triangulation
Jiggle	'off' | 'mean' | 'minimum' | 'on'	'mean'	Call `jigglemesh` after creating the mesh, with the `Opt` name-value pair set to the stated value. Exceptions: 'off' means do not call `jigglemesh`, and 'on' means call `jigglemesh` with `Opt` = 'off'.
JiggleIter	numeric	10	Maximum iterations
MesherVersion	'R2013a' | 'preR2013a'	'preR2013a'	Algorithm for generating initial mesh

The `Hmax` property controls the size of the triangles on the mesh. `initmesh` creates a mesh where triangle edge lengths are approximately `Hmax` or less.

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` cannot be equal to either of its bounds, `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.

The `MesherVersion` property chooses the algorithm for mesh generation. The `'R2013a'` algorithm runs faster, and can triangulate more geometries than the `'preR2013a'` algorithm. Both algorithms use Delaunay triangulation.

Examples

Make a simple triangular mesh of the L-shaped membrane in the PDE app. Before you do anything in the PDE app, 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.
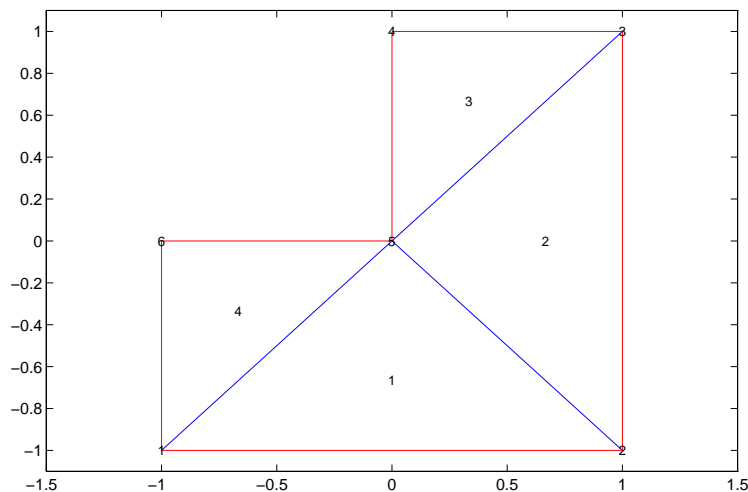
The corresponding mesh data structures can be exported to the main workspace by selecting the **Export Mesh** option from the **Mesh** menu.

```
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
```

More About

- "Mesh Data" on page 2-211

References

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

See Also

```
decsg | jigglemesh | refinemesh
```

interpolateSolution

Interpolate PDE solution to arbitrary points

Compatibility

INTERPOLATING PDE SOLUTION TO ARBITRARY POINTS IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
uintrp = interpolateSolution(results,xq,yq)
uintrp = interpolateSolution(results,xq,yq,zq)
uintrp = interpolateSolution(results,querypoints)

uintrp = interpolateSolution( ___ ,iU)
uintrp = interpolateSolution( ___ ,iT)
uintrp = interpolateSolution( ___ ,iU,iT)
```

Description

`uintrp = interpolateSolution(results,xq,yq)` returns the interpolated values of the solution to the scalar *elliptic* equation specified in `results` at the 2-D points specified in `xq` and `yq`.

`uintrp = interpolateSolution(results,xq,yq,zq)` returns the interpolated values at the 3-D points specified in `xq`, `yq`, and `zq`.

`uintrp = interpolateSolution(results,querypoints)` returns the interpolated values at the points in `querypoints`.

`uintrp = interpolateSolution( ___ ,iU)`, for any previous syntax, returns the interpolated values of the solution to the *system* of *elliptic* equations for equation indices `iU`.

`uintrp = interpolateSolution( ___ ,iT)` returns the interpolated values of the solution to the *parabolic*, *hyperbolic*, or *eigenvalue* equation or *system* of equations at times or modal indices `iT`.

`uintrp = interpolateSolution( ___ ,iU,iT)` returns the interpolated values of the solution to the system of *parabolic*, *hyperbolic*, or *eigenvalue* equations for equation indices `iU` at times or modal indices `iT`.

Examples

Interpolate Scalar Elliptic Results

Interpolate the solution to a scalar elliptic problem along a line and plot the result.

Create the solution to the problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
generateMesh(model,'Hmax',0.05);
results = solvepde(model);
```

Interpolate the solution along the straight line from $(x,y) = (-1,-1)$ to $(1,1)$. Plot the interpolated solution.

```
xq = linspace(-1,1,101);
yq = xq;

uintrp = interpolateSolution(results,xq,yq);
plot(xq,uintrp)

xlabel('x')
ylabel('u(x)')
```
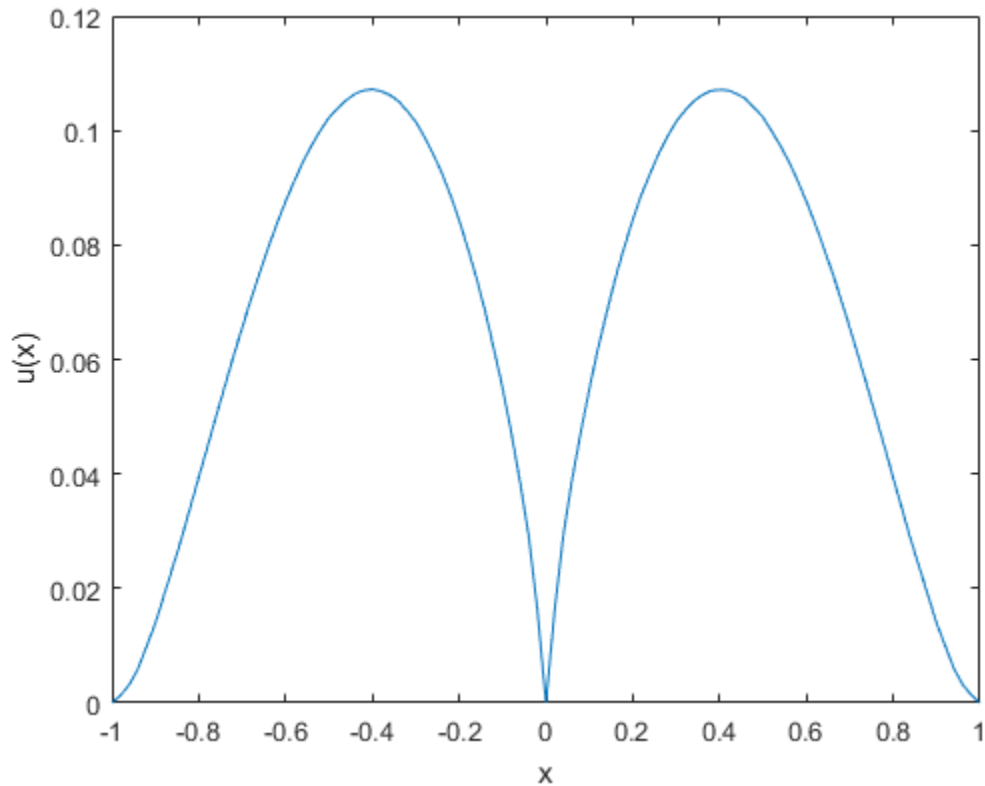
Interpolate Solution of Poisson's Equation

Calculate the mean exit time of a Brownian particle from a region that contains absorbing (escape) boundaries and reflecting boundaries. Use the Poisson's equation with constant coefficients and 3-D rectangular block geometry to model this problem.

Create the solution for this problem.

```
model = createpde;
importGeometry(model,'Block.stl');
applyBoundaryCondition(model,'face',[1,2,5],'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
```

```
                              'a',0,...
                              'f',2);
generateMesh(model);
results = solvepde(model);
```

Create a grid and interpolate the solution to the grid.

```
[X,Y,Z] = meshgrid(0:135,0:35,0:61);
uintrp = interpolateSolution(results,X,Y,Z);
uintrp = reshape(uintrp,size(X));
```

Create a contour slice plot for five fixed values of the *y* coordinate.

```
contourslice(X,Y,Z,uintrp,[],0:4:16,[])
colormap jet
xlabel('x');
ylabel('y');
zlabel('z')
xlim([0,100])
ylim([0,20])
zlim([0,50])
axis equal
view(-50,22)
colorbar
```
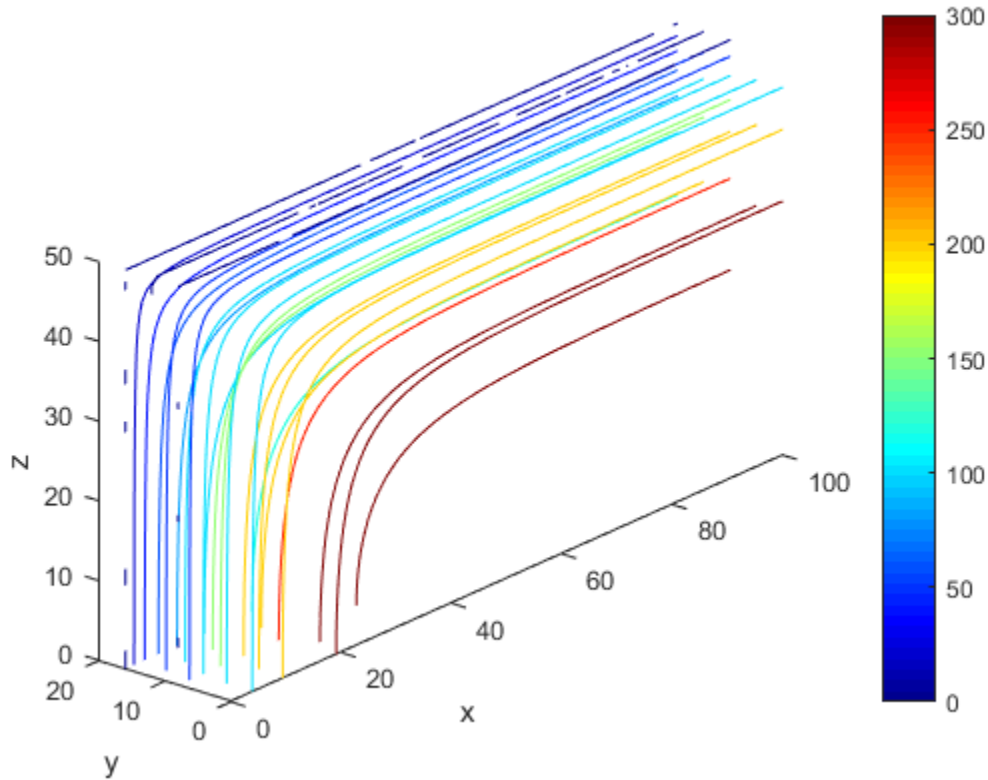
Interpolate Scalar Elliptic Results Using Query Matrix

Solve a scalar elliptic problem and interpolate the solution to a dense grid.

Create the solution to the problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
specifyCoefficients(model,'m',0,'...
                          d',0,...
                          'c',1,...
```

```
                                    'a',0,...
                                    'f',1);
        generateMesh(model,'Hmax',0.05);
        results = solvepde(model);
```

Interpolate the solution on the grid from −1 to 1 in each direction.

```
v = linspace(-1,1,101);
[X,Y] = meshgrid(v);
querypoints = [X(:),Y(:)]';
uintrp = interpolateSolution(results,querypoints);
```

Plot the resulting interpolation on a mesh.

```
uintrp = reshape(uintrp,size(X));
mesh(X,Y,uintrp)
xlabel('x')
ylabel('y')
```
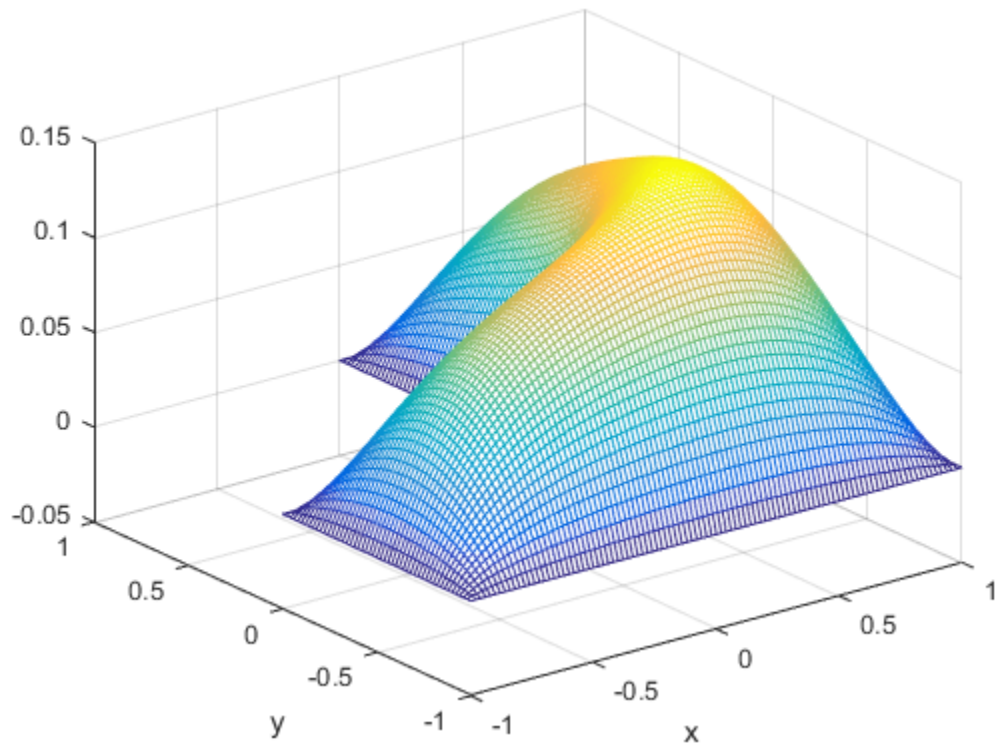
Interpolate Elliptic System

Create the solution to a two-component elliptic system and plot the two components along a planar slice through the geometry.

Create a PDE model for two components. Import the geometry of a torus.

```
model = createpde(2);
importGeometry(model,'Torus.stl');
pdegplot(model,'FaceLabels','on');
```
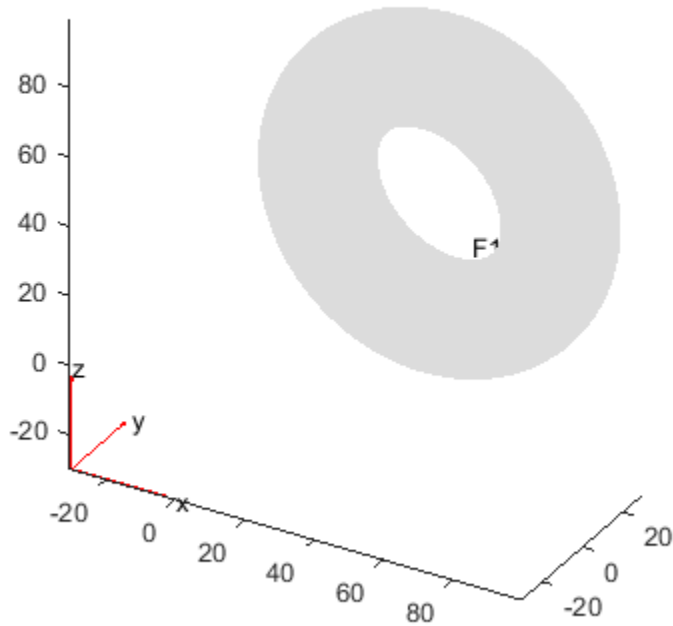
Set boundary conditions.

```
gfun = @(region,state)[0,region.z-40];
applyBoundaryCondition(model,'face',1,'g',gfun);
ufun = @(region,state)[region.x-40,0];
applyBoundaryCondition(model,'face',1,'u',ufun);
```

Set the problem coefficients.

```
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',[1;0;1;0;0;1;0;0;1;0;1;0;1;0;0;1;0;1;0;0;1],...
                          'a',0,...
                          'f',[1;1]);
```

Create a mesh and solve the problem.

```
generateMesh(model);
results = solvepde(model);
```

Interpolate the results on a plane that slices the torus for each of the two components.

```
[X,Z] = meshgrid(0:100);
Y = 15*ones(size(X));
uintrp = interpolateSolution(results,X,Y,Z,[1,2]);
```

Plot the two components.

```
sol1 = reshape(uintrp(:,1),size(X));
sol2 = reshape(uintrp(:,2),size(X));
figure
surf(X,Z,sol1)
title('Component 1')
figure
surf(X,Z,sol2)
title('Component 2')
```

Component 1

Component 2
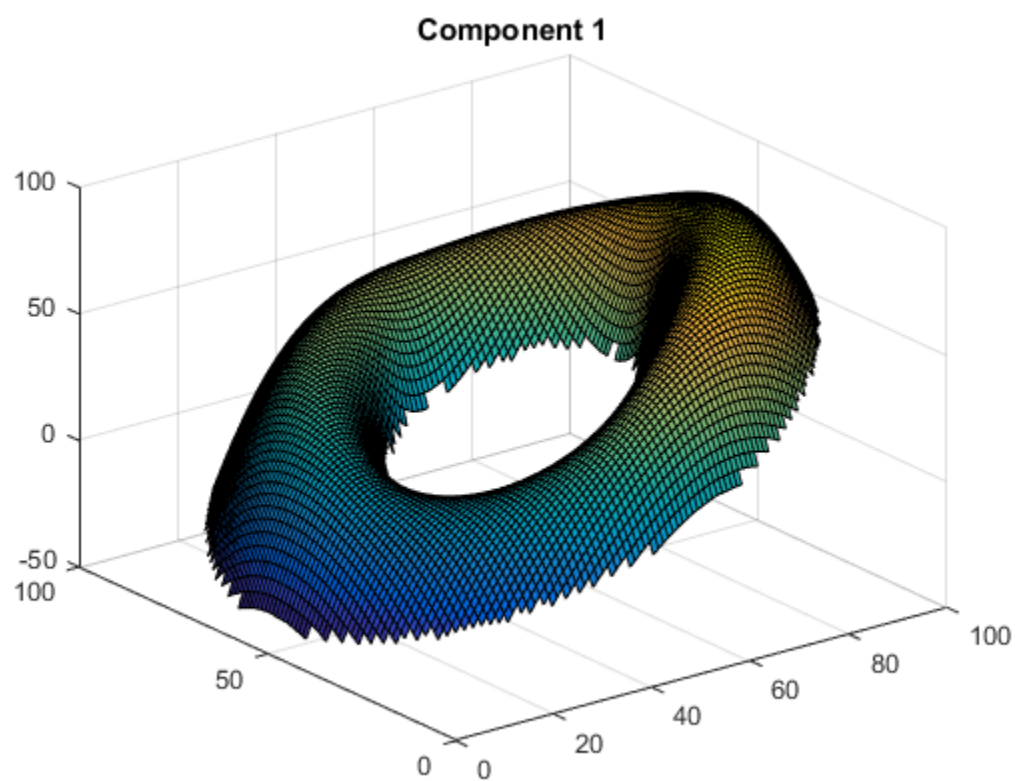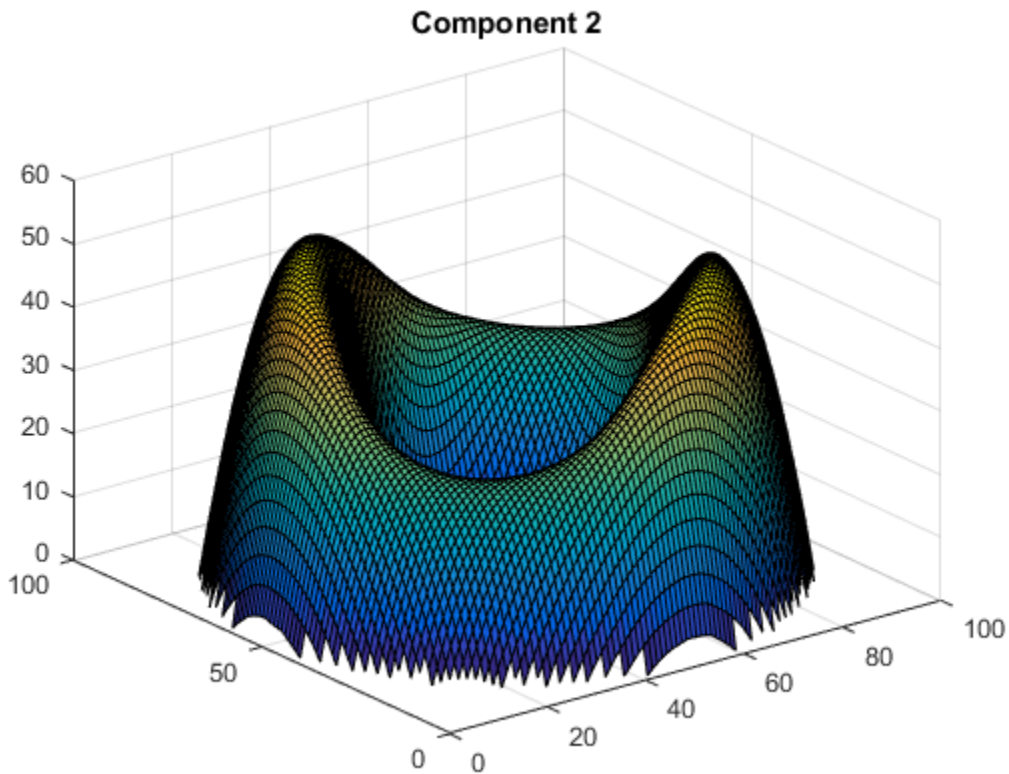
Interpolate Scalar Eigenvalue Results

Solve a scalar eigenvalue problem and interpolate one eigenvector to a grid.

Find the eigenvalues and eigenvectors for the L-shaped membrane.

```
model = createpde(1);
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
specifyCoefficients(model,'m',0,...
                          'd',1,...
                          'c',1,...
                          'a',0,...
```

```
                              'f',0);
r = [0,100];
generateMesh(model,'Hmax',1/50);
results = solvepdeeig(model,r);
```

Interpolate the eigenvector corresponding to the fifth eigenvalue to a coarse grid and plot the result.

```
[xq,yq] = meshgrid(-1:0.1:1);
uintrp = interpolateSolution(results,xq,yq,5);
uintrp = reshape(uintrp,size(xq));
surf(xq,yq,uintrp)
```
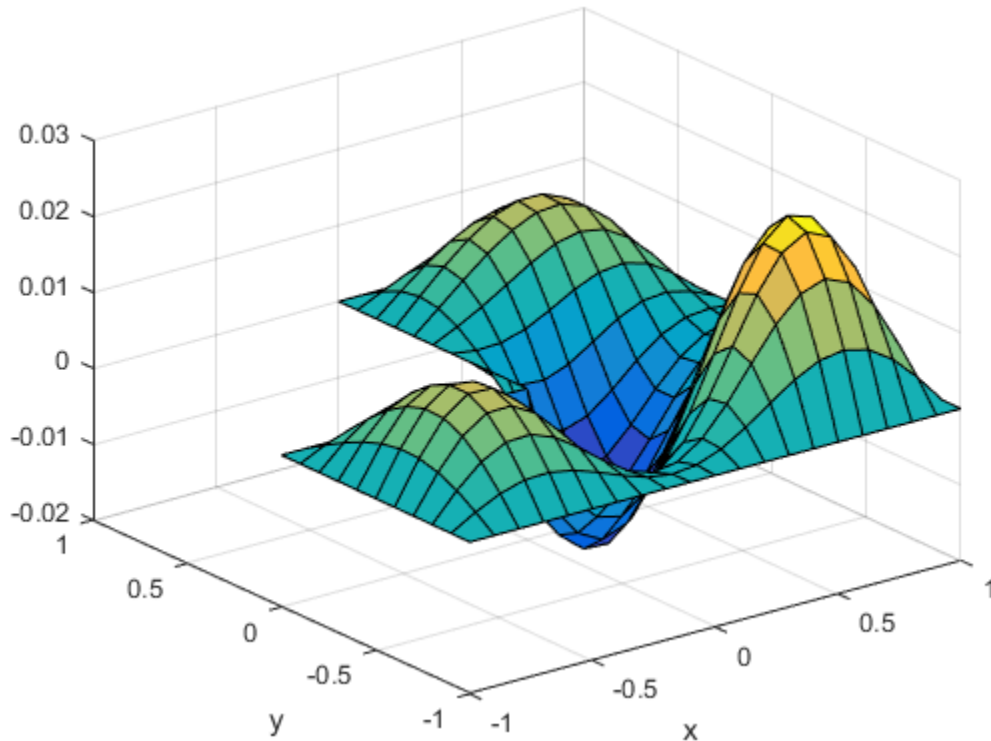

Interpolate Hyperbolic System

Solve a system of hyperbolic PDEs and interpolate the solution.

Import slab geometry for a 3-D problem with three solution components. Plot the geometry.

```
model = createpde(3);
importGeometry(model,'Plate10x10x1.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```


Set boundary conditions such that face 2 is fixed (zero deflection in any direction) and face 5 has a load of 1e3 in the positive *z*-direction. This load causes the slab to bend upward. Set the initial condition that the solution is zero, and its derivative with respect to time is also zero.

```
applyBoundaryCondition(model,'face',2,'u',[0,0,0]);
```

```
applyBoundaryCondition(model,'face',5,'g',[0,0,1e3]);
setInitialConditions(model,0,0);
```

Create PDE coefficients for the equations of linear elasticity. Set the material properties to be similar to those of steel. See "3-D Linear Elasticity Equations in Toolbox Form" on page 3-35.

```
E = 200e9;
nu = 0.3;
specifyCoefficients(model,'m',1,...
                          'd',0,...
                          'c',elasticityC3D(E,nu),...
                          'a',0,...
                          'f',[0;0;0]);
```

Generate a mesh, setting `Hmax` to `1`.

```
generateMesh(model,'Hmax',1);
```

Solve the problem for times `0` through `5e-3` in steps of `1e-4`.

```
tlist = 0:1e-4:5e-3;
results = solvepde(model,tlist);
```

Interpolate the solution at fixed *x*- and *z*-coordinates in the centers of their ranges, `5` and `0.5` respectively. Interpolate for *y* from `0` through `10` in steps of `0.2`. Obtain just component `3`, the *z*-component of the solution.

```
yy = 0:0.2:10;
zz = 0.5*ones(size(yy));
xx = 10*zz;
component = 3;
uintrp = interpolateSolution(results,xx,yy,zz,component,1:length(tlist));
```

The solution is a 51-by-1-by-51 array. Use `squeeze` to remove the singleton dimension. Removing the singleton dimension transforms this array to a 51-by-51 matrix which simplifies indexing into it.

```
uintrp = squeeze(uintrp);
```

Plot the solution as a function of *y* and time.

```
[X,Y] = ndgrid(yy,tlist);
figure
surf(X,Y,uintrp)
```

```
xlabel('Y')
ylabel('Time')
title('Deflection at x = 5, z = 0.5')
zlim([O,14e-5])
```


Deflection at x = 5, z = 0.5

Input Arguments

results — PDE solution
StationaryResults object (default) | TimeDependentResults object | EigenResults object

PDE solution, specified as a StationaryResults object, a TimeDependentResults object, or an EigenResults object. Create results using solvepde, solvepdeeig, or createPDEResults.

Example: `results = solvepde(model)`

xq — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `interpolateSolution` evaluates the solution at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. So `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`intepolateSolution` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The returned solution is a column vector of the same size. To ensure that the dimensions of the returned solution is consistent with the dimensions of the original query points, use `reshape`. For example, use `uintrp = reshape(gradxuintrp,size(xq))`.

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateSolution` evaluates the solution at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. So `xq`, `yq`, and (if present) `zq` must have the same number of entries. Internally, `interpolateSolution` converts query points to the column vector `yq(:)`.

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateSolution` evaluates the solution at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. So `xq`, `yq`, and `zq` must have the same number of entries. Internally, `interpolateSolution` converts query points to the column vector `zq(:)`.

Data Types: `double`

`querypoints` — Query points
real matrix

Query points, specified as a real matrix with either two rows for 2-D geometry, or three rows for 3-D geometry. `interpolateSolution` evaluates the solution at the coordinate

points `querypoints(:,i)`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For 2-D geometry, `querypoints = [0.5,0.5,0.75,0.75; 1,2,0,0.5]`

Data Types: `double`

`iU` — Equation indices
vector of positive integers

Equation indices, specified as a vector of positive integers. Each entry in `iU` specifies an equation index.

Example: `iU = [1,5]` specifies the indices for the first and fifth equations.

Data Types: `double`

`iT` — Time or mode indices
vector of positive integers

Time or mode indices, specified as a vector of positive integers. Each entry in `iT` specifies a time index for parabolic or hyperbolic solutions, or a mode index for eigenvalue solutions.

Example: `iT = 1:5:21` specifies the time or mode for every fifth solution up to 21.

Data Types: `double`

Output Arguments

`uintrp` — Solution at query points
array

Solution at query points, returned as an array. For query points that are outside the geometry, `uintrp = NaN`. For details about dimensions of the solution, see "Dimensions of Solutions and Gradients" on page 3-167.

See Also
`evaluateGradient` | `PDEModel` | `StationaryResults` | `TimeDependentResults`

Introduced in R2015b

jigglemesh

Jiggle internal points of triangular mesh

Compatibility

THIS PAGE DESCRIBES THE LEGACY APPROACH. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `generateMesh`.

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'` | `'minimum'`	`'mean'`	Optimization method, described in the following bullets
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 'minimum' 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')
```

More About

- "Mesh Data" on page 2-211

See Also

initmesh | pdetriq

meshToPet

[p,e,t] representation of FEMesh data

Compatibility

THIS FUNCTION SUPPORTS THE LEGACY APPROACH. New features might not be compatible with the [p,e,t] representation of FEMesh data.

Syntax

```
[p,e,t] = meshToPet(mesh)
```

Description

[p,e,t] = meshToPet(mesh) extracts the legacy [p,e,t] mesh representation from a FEMesh object.

Examples

Convert 2-D Mesh to [p,e,t] Form

This example shows how to convert a mesh in object form to [p,e,t] form.

Create a 2-D PDE geometry and incorporate it into a model object. View the geometry.

```
model = createpde(1);

R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
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];
ns = (char('R1','C1'))';
```

```
sf = 'R1-C1';
gd = decsg(geom,sf,ns);

geometryFromEdges(model,gd);
pdegplot(model,'EdgeLabels','on')
xlim([-1.1 1.1])
axis equal
```
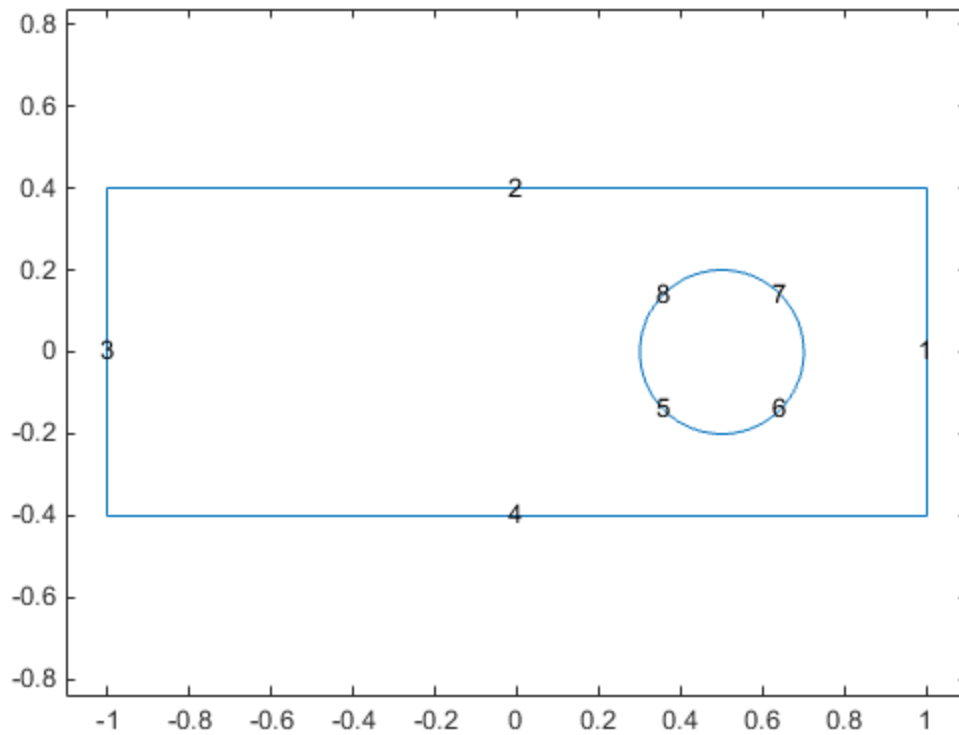

Create a mesh for the geometry. View the mesh.

```
generateMesh(model);
pdemesh(model)
axis equal
```
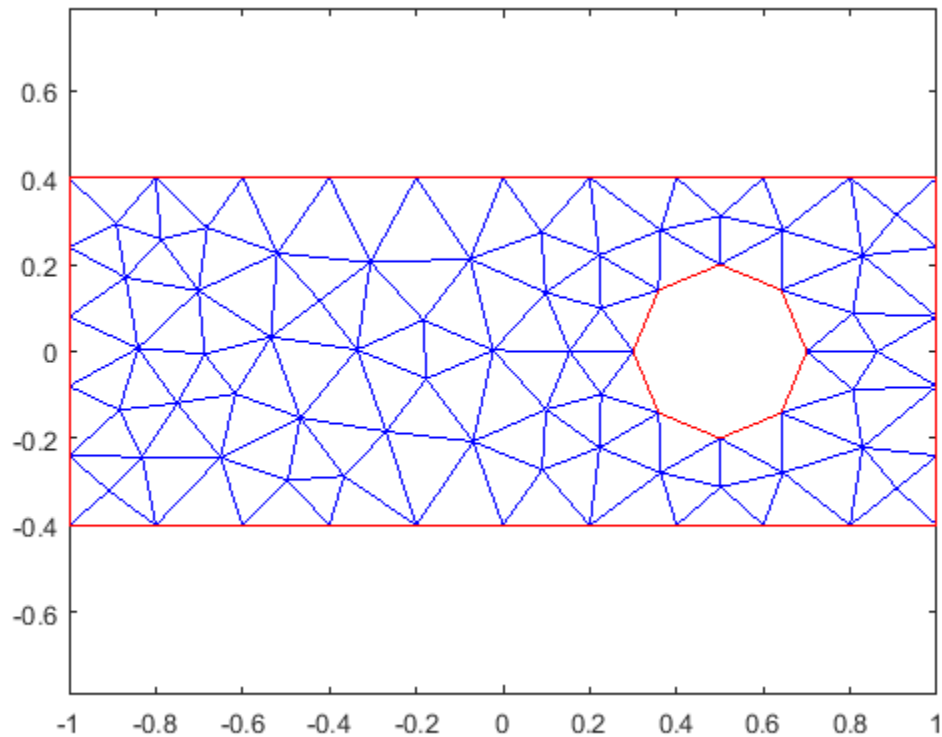
Convert the mesh to `[p,e,t]` form.

```
[p,e,t] = meshToPet(model.Mesh);
```

View the sizes of the `[p,e,t]` matrices.

```
size(p)

ans =

     2    87
```

```
size(e)

ans =
```

```
     7    38

size(t)

ans =

     4    136
```

Input Arguments

mesh — Mesh object
Mesh property of a `PDEModel` object | output of `generateMesh`

Mesh object, specified as the `Mesh` property of a `PDEModel` object or as the output of `generateMesh`.

Example: `model.Mesh`

Output Arguments

p — Mesh points
2-by-`Np` matrix | 3-by-`Np` matrix

Mesh points, returned as a 2-by-`Np` matrix (2-D geometry) or a 3-by-`Np` matrix (3-D geometry). `Np` is the number of points (nodes) in the mesh. Column `k` of `p` consists of the *x*-coordinate of point `k` in `p(1,k)`, the *y*-coordinate of point `k` in `p(2,k)`, and, for 3-D, the *z*-coordinate of point `k` in `p(3,k)`. For details, see "Mesh Data" on page 2-211.

e — Mesh edges
7-by-`Ne` matrix | mesh associativity object

Mesh edges, returned as a 7-by-`Ne` matrix (2-D), or a mesh associativity object (3-D). `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. For details, see "Mesh Data" on page 2-211.

t — Mesh elements
4-by-`Nt` matrix | 7-by-`Nt` matrix | 5-by-`Nt` matrix | 11-by-`Nt` matrix

Mesh elements, returned as a 4-by-`Nt` matrix (2-D with linear elements), a 7-by-`Nt` matrix (2-D with quadratic elements), a 5-by-`Nt` matrix (3-D with linear elements), or an 11-by-`Nt` matrix (3-D with quadratic elements). `Nt` is the number of triangles or tetrahedra in the mesh.

The `t(i,k)`, with `i` ranging from 1 through `end - 1`, contain indices to the corner points and possibly edge centers of element `k`. For details, see "Mesh Data" on page 2-211. The last row, `t(end,k)`, contains the subdomain number of the element.

More About

Tips

- Use `meshToPet` to obtain the `p` and `t` data for interpolation using `pdeInterpolant`.

- "Mesh Data" on page 2-211

See Also

FEMesh Properties | `generateMesh`

Introduced in R2015a

parabolic

Solve parabolic PDE problem

Parabolic equation solver

Solves PDE problems of the type

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

on a 2-D or 3-D region Ω, or the system PDE problem

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

The variables c, a, f, and d can depend on position, time, and the solution u and its gradient.

Compatibility

parabolic IS NOT RECOMMENDED. Use solvepde instead.

Syntax

```
u = parabolic(u0,tlist,model,c,a,f,d)
u = parabolic(u0,tlist,b,p,e,t,c,a,f,d)
u = parabolic(u0,tlist,Kc,Fc,B,ud,M)
u = parabolic( ___ ,rtol)
u = parabolic( ___ ,rtol,atol)
u = parabolic( ___ ,'Stats','off')
```

Description

u = parabolic(u0,tlist,model,c,a,f,d) produces the solution to the FEM formulation of the scalar PDE problem

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

on a 2-D or 3-D region Ω, or the system PDE problem

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

with geometry, mesh, and boundary conditions specified in `model`, and with initial value `u0`. The variables c, a, f, and d in the equation correspond to the function coefficients `c`, `a`, `f`, and `d` respectively.

`u = parabolic(u0,tlist,b,p,e,t,c,a,f,d)` solves the problem using boundary conditions `b` and finite element mesh specified in `[p,e,t]`.

`u = parabolic(u0,tlist,Kc,Fc,B,ud,M)` solves the problem based on finite element matrices that encode the equation, mesh, and boundary conditions.

`u = parabolic( ___ ,rtol)` and `u = parabolic( ___ ,rtol,atol)`, for any of the previous input arguments, modify the solution process by passing to the ODE solver a relative tolerance `rtol`, and optionally an absolute tolerance `atol`.

`u = parabolic( ___ ,'Stats','off')`, for any of the previous input arguments, turns off the display of internal ODE solver statistics during the solution process.

Examples

Parabolic Equation

Solve the parabolic equation

$$\frac{\partial u}{\partial t} = \Delta u$$

on the square domain specified by `squareg`.

Create a PDE model and import the geometry.

```
model = createpde;
geometryFromEdges(model,@squareg);
```

```
pdegplot(model,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```
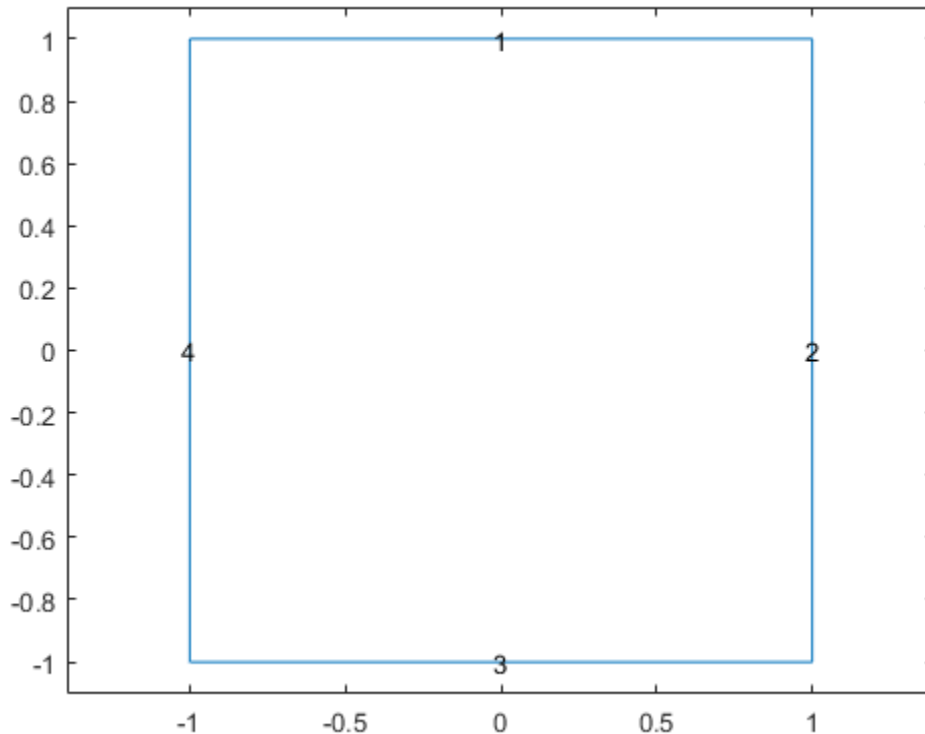

Set Dirichlet boundary conditions $u = 0$ on all edges.

```
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',0);
```

Generate a relatively fine mesh.

```
generateMesh(model,'Hmax',0.02);
```

Set the initial condition to have $u(0) = 1$ on the disk $x^2 + y^2 \leq 0.4^2$ and $u(0) = 0$ elsewhere.

```
p = model.Mesh.Nodes;
u0 = zeros(size(p,2),1);
ix = find(sqrt(p(1,:).^2 + p(2,:).^2) <= 0.4);
u0(ix) = ones(size(ix));
```

Set solution times to be from 0 to 0.1 with step size 0.005.

```
tlist = linspace(0,0.1,21);
```

Create the PDE coefficients.

```
c = 1;
a = 0;
f = 0;
d = 1;
```

Solve the PDE.

```
u = parabolic(u0,tlist,model,c,a,f,d);
```

```
147 successful steps
0 failed attempts
296 function evaluations
1 partial derivatives
28 LU decompositions
295 solutions of linear systems
```

Plot the initial condition, the solution at the final time, and two intermediate solutions.

```
figure
subplot(2,2,1)
pdeplot(model,'xydata',u(:,1));
axis equal
title('t = 0')
subplot(2,2,2)
pdeplot(model,'xydata',u(:,5))
axis equal
title('t = 0.02')
subplot(2,2,3)
pdeplot(model,'xydata',u(:,11))
axis equal
title('t = 0.05')
subplot(2,2,4)
pdeplot(model,'xydata',u(:,end))
axis equal
```

```
title('t = 0.1')
```
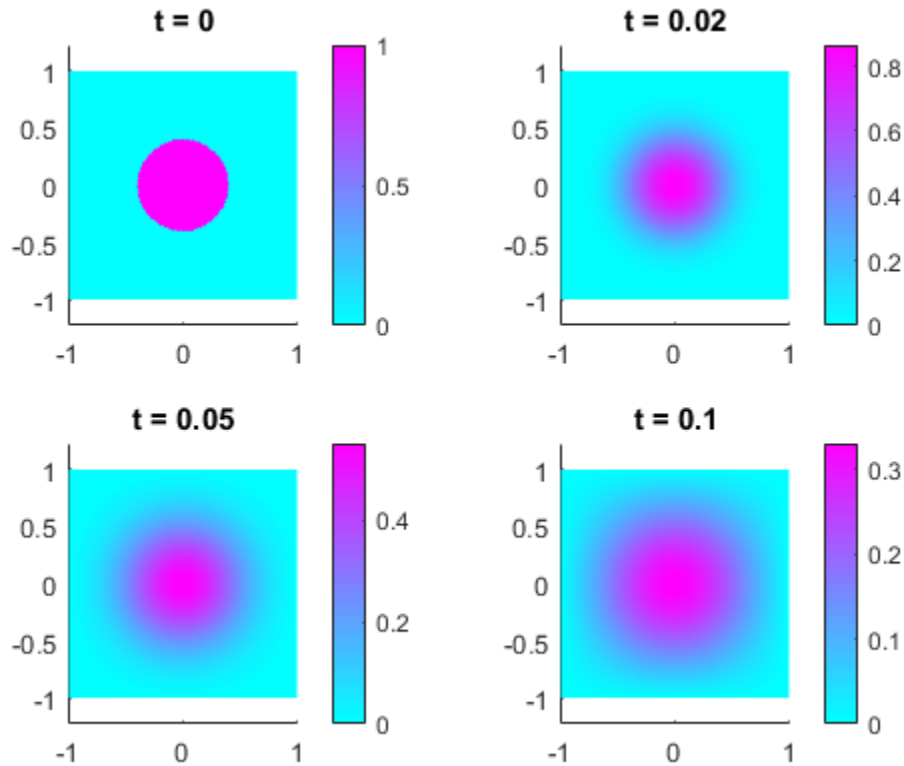

Parabolic Equation Using Legacy Syntax

Solve the parabolic equation

$$\frac{\partial u}{\partial t} = \Delta u$$

on the square domain specified by squareg, using a geometry function to specify the geometry, a boundary function to specify the boundary conditions, and using initmesh to create the finite element mesh.

Specify the geometry as @squareg and plot the geometry.

```
g = @squareg;
pdegplot(g,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```
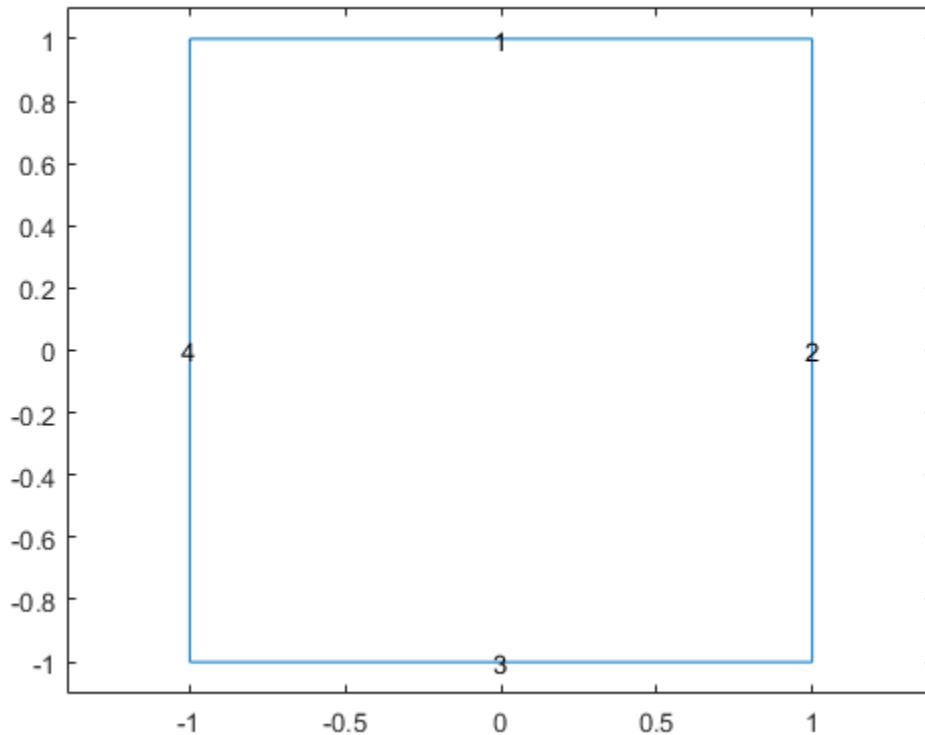

Set Dirichlet boundary conditions $u = 0$ on all edges. The `squareb1` function specifies these boundary conditions.

```
b = @squareb1;
```

Generate a relatively fine mesh.

```
[p,e,t] = initmesh(g,'Hmax',0.02);
```

Set the initial condition to have $u(0) = 1$ on the disk $x^2 + y^2 \leq 0.4^2$ and $u(0) = 0$ elsewhere.

```
u0 = zeros(size(p,2),1);
ix = find(sqrt(p(1,:).^2 + p(2,:).^2) <= 0.4);
u0(ix) = ones(size(ix));
```

Set solution times to be from 0 to 0.1 with step size 0.005.

```
tlist = linspace(0,0.1,21);
```

Create the PDE coefficients.

```
c = 1;
a = 0;
f = 0;
d = 1;
```

Solve the PDE.

```
u = parabolic(u0,tlist,b,p,e,t,c,a,f,d);
```

```
147 successful steps
0 failed attempts
296 function evaluations
1 partial derivatives
28 LU decompositions
295 solutions of linear systems
```

Plot the initial condition, the solution at the final time, and two intermediate solutions.

```
figure
subplot(2,2,1)
pdeplot(p,e,t,'xydata',u(:,1));
axis equal
title('t = 0')
subplot(2,2,2)
pdeplot(p,e,t,'xydata',u(:,5))
axis equal
title('t = 0.02')
subplot(2,2,3)
pdeplot(p,e,t,'xydata',u(:,11))
axis equal
title('t = 0.05')
subplot(2,2,4)
pdeplot(p,e,t,'xydata',u(:,end))
```

```
axis equal
title('t = 0.1')
```
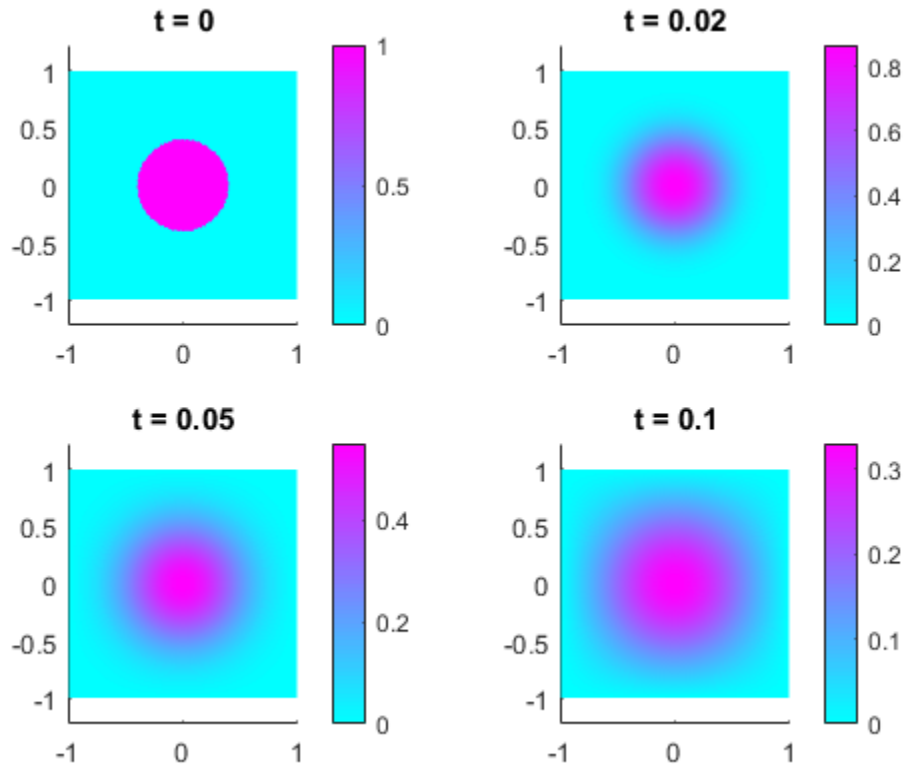

Parabolic Problem Using Matrix Coefficients

Create finite element matrices that encode a parabolic problem, and solve the problem.

The problem is the evolution of temperature in a conducting block. The block is a rectangular slab.

```
model = createpde(1);
importGeometry(model,'Block.stl');
handl = pdegplot(model,'FaceLabels','on');
view(-42,24)
handl(1).FaceAlpha = 0.5;
```
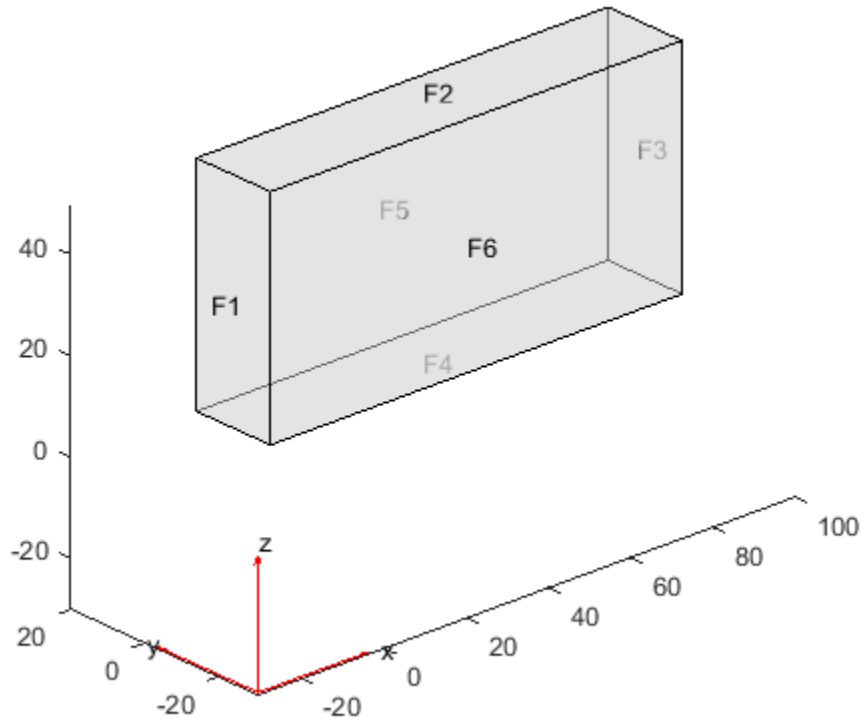
Faces 1, 4, and 6 of the slab are kept at 0 degrees. The other faces are insulated. Include the boundary condition on faces 1, 4, and 6. You do not need to include the boundary condition on the other faces because the default condition is insulated.

```
applyBoundaryCondition(model,'Face',[1,4,6],'u',0);
```

The initial temperature distribution in the block has the form

$$u_0 = 10^{-3}xyz.$$

```
generateMesh(model);
p = model.Mesh.Nodes;
x = p(1,:);
y = p(2,:);
z = p(3,:);
```

```
u0 = x.*y.*z*1e-3;
```

The parabolic equation in toolbox syntax is

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

Suppose the thermal conductivity of the block leads to a c coefficient value of 1. The values of the other coefficients in this problem are $d = 1$, $a = 0$, and $f = 0$.

```
d = 1;
c = 1;
a = 0;
f = 0;
```

Create the finite element matrices that encode the problem.

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Solve the problem at time steps of 1 for times ranging from 0 to 40.

```
tlist = linspace(0,40,41);
u = parabolic(u0,tlist,Kc,Fc,B,ud,M);

38 successful steps
0 failed attempts
78 function evaluations
1 partial derivatives
10 LU decompositions
77 solutions of linear systems
```

Plot the solution on the outside of the block at times 0, 10, 25, and 40. Ensure that the coloring is the same for all plots.

```
umin = min(min(u));
umax = max(max(u));
subplot(2,2,1)
pdeplot3D(model,'colormapdata',u(:,1))
colorbar off
view(125,22)
title 't = 0'
caxis([umin umax]);
subplot(2,2,2)
pdeplot3D(model,'colormapdata',u(:,11))
colorbar off
```

```
view(125,22)
title 't = 10'
caxis([umin umax]);
subplot(2,2,3)
pdeplot3D(model,'colormapdata',u(:,26))
colorbar off
view(125,22)
title 't = 25'
caxis([umin umax]);
subplot(2,2,4)
pdeplot3D(model,'colormapdata',u(:,41))
colorbar off
view(125,22)
title 't = 40'
caxis([umin umax]);
```

Input Arguments

u0 — Initial condition
vector | text expression

Initial condition, specified as a scalar, vector of nodal values, or text expression. The initial condition is the value of the solution u at the initial time, specified as a column vector of values at the nodes. The nodes are either p in the [p,e,t] data structure, or are model.Mesh.Nodes. For details, see "Solve PDEs with Initial Conditions" on page 2-162.

- If the initial condition is a constant scalar v, specify u0 as v.

- If there are Np nodes in the mesh, and *N* equations in the system of PDEs, specify u0 as a column vector of Np*N elements, where the first Np elements correspond to the first component of the solution u, the second Np elements correspond to the second component of the solution u, etc.

- Give a text expression of a function, such as `'x.^2 + 5*cos(x.*y)'`. If you have a system of *N* > 1 equations, give a text array such as

```
char('x.^2 + 5*cos(x.*y)',...
     'tanh(x.*y)./(1+z.^2)')
```

Example: `x.^2+5*cos(y.*x)`

Data Types: `double` | `char`
Complex Number Support: Yes

`tlist` — Solution times
real vector

Solution times, specified as a real vector. The solver returns the solution to the PDE at the solution times.

Example: `0:0.2:4`

Data Types: `double`

`model` — PDE model
`PDEModel` object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde(1)`

`c` — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. `c` represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify c in various ways, detailed in "c Coefficient for Systems" on page 2-125. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

a — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. a represents the a coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify a in various ways, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

f — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. f represents the *f* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify f in various ways, detailed in "f Coefficient for Systems" on page 2-98. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

d — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. d represents the *d* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify d in various ways, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double | char | function_handle`
Complex Number Support: Yes

b — Boundary conditions
boundary matrix | boundary file

Boundary conditions, specified as a boundary matrix or boundary file. Pass a boundary file as a function handle or as a string naming the file.

- A boundary matrix is generally an export from the PDE app. For details of the structure of this matrix, see "Boundary Matrix for 2-D Geometry" on page 2-171.

- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-199.

For more information on boundary conditions, see "Forms of Boundary Condition Specification" on page 2-170.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

p — Mesh nodes
output of `initmesh` | output of `meshToPet`

Mesh nodes, specified as the output of `initmesh` or `meshToPet`. For the structure of a `p` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

e — Mesh edges
output of `initmesh` | output of `meshToPet`

Mesh edges, specified as the output of `initmesh` or `meshToPet`. For the structure of `e`, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

t — Mesh elements
output of `initmesh` | output of `meshToPet`

Mesh elements, specified as the output of `initmesh` or `meshToPet`. Mesh elements are the triangles or tetrahedra that form the finite element mesh. For the structure of a `t` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

Kc — Stiffness matrix
sparse matrix | full matrix

Stiffness matrix, specified as a sparse matrix or as a full matrix. See "Elliptic Equations" on page 5-2. Typically, `Kc` is the output of `assempde`.

Fc — Load vector
vector

Load vector, specified as a vector. See "Elliptic Equations" on page 5-2. Typically, `Fc` is the output of `assempde`.

B — Dirichlet nullspace
sparse matrix

Dirichlet nullspace, returned as a sparse matrix. See "Algorithms" on page 6-56. Typically, `B` is the output of `assempde`.

ud — Dirichlet vector
vector

Dirichlet vector, returned as a vector. See "Algorithms" on page 6-56. Typically, `ud` is the output of `assempde`.

M — Mass matrix
sparse matrix | full matrix

Mass matrix. specified as a sparse matrix or a full matrix. See "Elliptic Equations" on page 5-2.

To obtain the input matrices for `pdeeig`, `hyperbolic` or `parabolic`, run both `assema` and `assempde`:

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Note: Create the M matrix using `assema` with `d`, not `a`, as the argument before `f`.

Data Types: `double`
Complex Number Support: Yes

`rtol` — Relative tolerance for ODE solver

`1e-3` (default) | positive real

Relative tolerance for ODE solver, specified as a positive real.

Example: `2e-4`

Data Types: `double`

`atol` — Absolute tolerance for ODE solver

`1e-6` (default) | positive real

Absolute tolerance for ODE solver, specified as a positive real.

Example: `2e-7`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example:

`'Stats'` — Display ODE solver statistics

`'on'` (default) | `'off'`

Display ODE solver statistics, specified as `'on'` or `'off'`. Suppress the display by setting `Stats` to `'off'`.

Example: `x = parabolic(u0,tlist,model,c,a,f,d,'Stats','off')`

Data Types: `char`

Output Arguments

u — PDE solution
matrix

PDE solution, returned as a matrix. The matrix is `Np`*`N`-by-`T`, where `Np` is the number of nodes in the mesh, *N* is the number of equations in the PDE (*N* = 1 for a scalar PDE), and `T` is the number of solution times, meaning the length of `tlist`. The solution matrix has the following structure.

- The first `Np` elements of each column in `u` represent the solution of equation 1, then next `Np` elements represent the solution of equation 2, etc. The solution `u` is the value at the corresponding node in the mesh.
- Column `i` of `u` represents the solution at time `tlist(i)`.

To obtain the solution at an arbitrary point in the geometry, use `pdeInterpolant`.

To plot the solution, use `pdeplot` for 2-D geometry, or see "Plot 3-D Solutions and Their Gradients" on page 3-145.

More About

Algorithms

`parabolic` internally calls `assema`, `assemb`, and `assempde` to create finite element matrices corresponding to the problem. It calls `ode15s` to solve the resulting system of ordinary differential equations. For details, see "Parabolic Equations" on page 5-17.
- "PDE Problem Setup"
- "Parabolic Equations" on page 5-17
- "Diffusion" on page 3-74

See Also
`solvepde`

Introduced before R2006a

pdeadgsc

Select triangles using relative tolerance criterion

Compatibility

pdeadgsc IS NOT RECOMMENDED.

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 2-211.

`c`,`a`, and `f` are PDE coefficients. For details, see "Scalar PDE Coefficients" on page 2-66 and "Coefficients for Systems of PDEs" on page 2-93.

`u` is the current solution, given as a column vector.

`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

`generateMesh`

pdeadworst

Select triangles relative to worst value

Compatibility

pdeadworst IS NOT RECOMMENDED.

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 2-211.

`c`, `a`, and `f` are PDE coefficients. For details, see "Scalar PDE Coefficients" on page 2-66.

`u` is the current solution, given as a column vector.

`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
generateMesh

pdearcl

Interpolation between parametric representation and arc length

Compatibility

2-D GEOMETRY FUNCTIONS ARE THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

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.

See the examples in "2-D Geometry".

pdeBoundaryConditions class

Boundary conditions

Compatibility

`pdeBoundaryConditions` **WILL BE REMOVED IN A FUTURE RELEASE.** Use `BoundaryCondition` instead.

Description

Define boundary conditions for each portion of the geometry boundary. Specify boundary conditions in one of three ways:

- Explicitly set the value of components of the solution on certain geometry edges by setting the `'u'` name-value pair, possibly including an `EquationIndex` name-value pair for PDE systems.

- Implicitly set the value of components of the solution on certain geometry edges by setting the `'h'` and `'r'` name-value pairs, which represent the equation $h*u = r$.

- Set generalized Neumann conditions on certain geometry edges by setting the `'g'` and `'q'` name-value pairs, which represent the equation

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

$\vec{n}$ is the outward unit normal on the boundary.

For systems of $N > 1$ equations, the generalized Neumann conditions are

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

See "Boundary Conditions for PDE Systems" on page 2-204.

Note: You can set only one type of boundary condition in a call to
`pdeBoundaryConditions`: a `'u'`, `EquationIndex` pair, or an `'r'`, `'h'` pair, or a `'g'`,
`'q'` pair. If you set only one member of a pair, the other takes its default value.

Construction

`bc = pdeBoundaryConditions(ApplicationRegion,Name,Value)` creates
boundary conditions for the geometry edge or edges in `ApplicationRegion`. The
`Name,Value` pairs specify the boundary conditions. `Name` can also be a property
name and `Value` is the corresponding value. `Name` must appear inside single
quotes (`' '`). You can specify several name-value pair arguments in any order as
`Name1,Value1,...,NameN,ValueN`.

Input Arguments

`ApplicationRegion` — Geometry edges for boundary conditions
vector of geometry edge entities

Geometry edges for boundary conditions, specified as a vector of geometry edge entities.

Before creating boundary conditions, first create geometry using the `decsg` function
or by writing a geometry file. Then call `pdeGeometryFromEdges` to create a geometry
container. Obtain the edges for the boundary conditions from the `Edges` property in the
geometry container.

Example: `ApplicationRegion = pg.Edges([1:4,10])`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

`'r'` — Dirichlet condition h*u = r
`zeros(N,1)` (default) | vector with `N` elements | function handle

Dirichlet condition `h*u = r`, specified as a vector with `N` elements or as a function
handle. `N` is the number of PDEs in the system. See "Systems of PDEs" on page 2-65.
For the syntax of the function handle form of `r`, see "Specify Nonconstant Boundary
Conditions" on page 2-190.

Note: You can set only one type of boundary condition in a call to
`pdeBoundaryConditions` (see "Description" on page 6-238): a `'u'`, `EquationIndex`
pair, or an `'r'`, `'h'` pair, or a `'g'`, `'q'` pair. If you set only one member of a pair, the
other takes its default value.

Example: `[0;4;-1]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'h'` — Dirichlet condition `h*u = r`
`eye(N)` (default) | N-by-N matrix | vector with N^2 elements | function handle

Dirichlet condition `h*u = r`, specified as an N-by-N matrix, a vector with N^2 elements,
or a function handle. N is the number of PDEs in the system. See "Systems of PDEs"
on page 2-65. For the syntax of the function handle form of `h`, see "Specify Nonconstant
Boundary Conditions" on page 2-190.

Note: You can set only one type of boundary condition in a call to
`pdeBoundaryConditions` (see "Description" on page 6-238): a `'u'`, `EquationIndex`
pair, or an `'r'`, `'h'` pair, or a `'g'`, `'q'` pair. If you set only one member of a pair, the
other takes its default value.

Example: `[2,1;1,2]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'g'` — Generalized Neumann condition $n \cdot (c \times \nabla u) + qu = g$
`zeros(N,1)` (default) | vector with N elements | function handle

Generalized Neumann condition $n \cdot (c \times \nabla u) + qu = g$, specified as a vector with N
elements or as a function handle. N is the number of PDEs in the system. See "Systems
of PDEs" on page 2-65. For the syntax of the function handle form of `g`, see "Specify
Nonconstant Boundary Conditions" on page 2-190.

Note: You can set only one type of boundary condition in a call to
`pdeBoundaryConditions` (see "Description" on page 6-238): a `'u'`, `EquationIndex`

pair, or an `'r'`, `'h'` pair, or a `'g'`, `'q'` pair. If you set only one member of a pair, the other takes its default value.

Example: `[3;2;-1]`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'q'` — Generalized Neumann condition `n·(c×∇u) + qu = g`
zeros(N) (default) | N-by-N matrix | vector with N^2 elements | function handle

Generalized Neumann condition `n·(c×∇u) + qu = g`, specified as an N-by-N matrix, a vector with `N^2` elements, or a function handle. `N` is the number of PDEs in the system. See "Systems of PDEs" on page 2-65. For the syntax of the function handle form of `q`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Note: You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-238): a `'u'`, `EquationIndex` pair, or an `'r'`, `'h'` pair, or a `'g'`, `'q'` pair. If you set only one member of a pair, the other takes its default value.

Example: `eye(3)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'u'` — Dirichlet conditions
zeros(N,1) (default) | vector of up to N elements | function handle

Dirichlet conditions, specified as a vector of up to `N` elements or a function handle. `EquationIndex` and `'u'` must have the same length. For the syntax of the function handle form of `u`, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Note: You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-238): a `'u'`, `EquationIndex` pair, or an `'r'`, `'h'` pair, or a `'g'`, `'q'` pair. If you set only one member of a pair, the other takes its default value.

Example: `bc = pdeBoundaryConditions(ApplicationRegion,'u',0)`

Data Types: `double`
Complex Number Support: Yes

`'EquationIndex'` — Index of specified `'u'` components
`1:N` (default) | vector of integers with entries from `1` to `N`

Index of specified `'u'` components, specified as a vector of integers with entries from `1` to `N`. `EquationIndex` and `'u'` must have the same length.

Note: You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-238): a `'u'`, `EquationIndex` pair, or an `'r'`, `'h'` pair, or a `'g'`, `'q'` pair. If you set only one member of a pair, the other takes its default value.

Example: `bc = pdeBoundaryConditions(ApplicationRegion,'u', [3;-1],'EquationIndex',[2,3])`

Data Types: `double`

`'Vectorized'` — Vectorized function evaluation
`'off'` (default) | `'on'`

Vectorized function evaluation, specified as `'on'` or `'off'`. This applies when you pass a function handle for an argument. To save time in function handle evaluation, specify `'on'`, assuming that your function handle computes in a vectorized fashion. See "Vectorization". For details, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Example: `bc = pdeBoundaryConditions(ApplicationRegion,'u',@ucalculator,'Vectorized','on')`

Data Types: `char`

Properties

`ApplicationRegion` — Geometry edges for boundary conditions
vector of geometry edge entities

Geometry edges for boundary conditions, specified as a vector of geometry edge entities.

`EquationIndex` — Index of specified `'u'` components
`1:N` (default) | vector of integers with entries from 1 to N

Index of specified `'u'` components, specified as a vector of integers with entries from 1 to N. The length of `EquationIndex` must equal the length of `'u'`.

g — Generalized Neumann condition n·(c×∇u) + qu = g
`zeros(N,1)` (default) | vector with N elements | function handle

Generalized Neumann condition $n \cdot (c \times \nabla u) + qu = g$, specified as a vector with N elements or as a function handle. N is the number of PDEs in the system. See "Systems of PDEs" on page 2-65. For details of the function handle syntax, see "Specify Nonconstant Boundary Conditions" on page 2-190.

h — Dirichlet condition h*u = r
`eye(N)` (default) | N-by-N matrix | vector with N^2 elements | function handle

Dirichlet condition $h*u = r$, specified as an N-by-N matrix, a vector with N^2 elements, or a function handle. N is the number of PDEs in the system. See "Systems of PDEs" on page 2-65. For details of the function handle syntax, see "Specify Nonconstant Boundary Conditions" on page 2-190.

q — Generalized Neumann condition n·(c×∇u) + qu = g
`zeros(N,N)` (default) | N-by-N matrix | vector with N^2 elements | function handle

Generalized Neumann condition $n \cdot (c \times \nabla u) + qu = g$, specified as an N-by-N matrix, a vector with N^2 elements, or as a function handle. N is the number of PDEs in the system. See "Systems of PDEs" on page 2-65. For details of the function handle syntax, see "Specify Nonconstant Boundary Conditions" on page 2-190.

r — Dirichlet condition h*u = r
`zeros(N,1)` (default) | vector with N elements | function handle

Dirichlet condition $h*u = r$, specified as a vector with N elements, or a function handle. N is the number of PDEs in the system. See "Systems of PDEs" on page 2-65. For details of the function handle syntax, see "Specify Nonconstant Boundary Conditions" on page 2-190.

u — Dirichlet condition u = Ubdry
`zeros(N,1)` (default) | vector with up to N elements | function handle

Dirichlet condition `u = Ubdry`, specified as a vector with up to `N` elements or a function handle. `N` is the number of PDEs in the system. See "Systems of PDEs" on page 2-65. The length of `u` must equal the length of `EquationIndex`. For details of the function handle syntax, see "Specify Nonconstant Boundary Conditions" on page 2-190.

`Vectorized` — Vectorized function evaluation
`'off'` (default) | `'on'`

Vectorized function evaluation, specified as `'off'` or `'on'`. This applies when you pass a function handle for an argument. For details of the function handle syntax, see "Specify Nonconstant Boundary Conditions" on page 2-190.

Examples

Constant Boundary Conditions for a Scalar Problem

This example sets Dirichlet conditions on two edges of a rectangle, and Neumann conditions on the other two edges.

Create a rectangle and view its edge labels.

```
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
sf = 'R1';
ns = sf';
% Create geometry
g = decsg(R1,sf,ns);
pg = pdeGeometryFromEdges(g); % Create pdeGeometry object
pdegplot(g,'edgeLabels','on')
xlim([-1.1 1.1])
axis equal
```
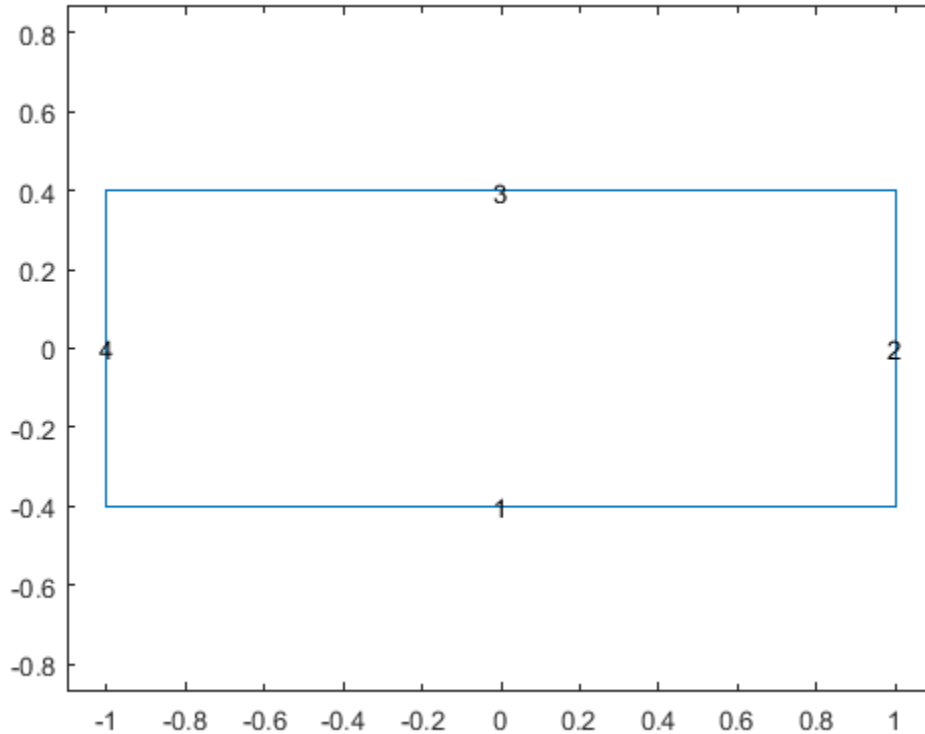
Set Dirichlet conditions so the solution $u = 3$ on edges 1 and 4.

```
bc1 = pdeBoundaryConditions(pg.Edges([1,4]),'u',3);
```

Set Neumann conditions with $q = 1$ and $g = -1$ on edges 2 and 3.

```
bc2 = pdeBoundaryConditions(pg.Edges([2,3]),'q',1,'g',-1);
```

Solve an elliptic equation with these boundary conditions, using coefficients $c = 1$, $a = 0$, and $f = 10$.

```
c = 1;
a = 0;
f = 10;
```

```
problem = pde();
problem.BoundaryConditions = [bc1,bc2];
[p,e,t] = initmesh(g);
[p,e,t] = refinemesh(g,p,e,t);
u = assempde(problem,p,e,t,c,a,f);
pdeplot(p,e,t,'xydata',u,'zdata',u)
view(174,2)
```
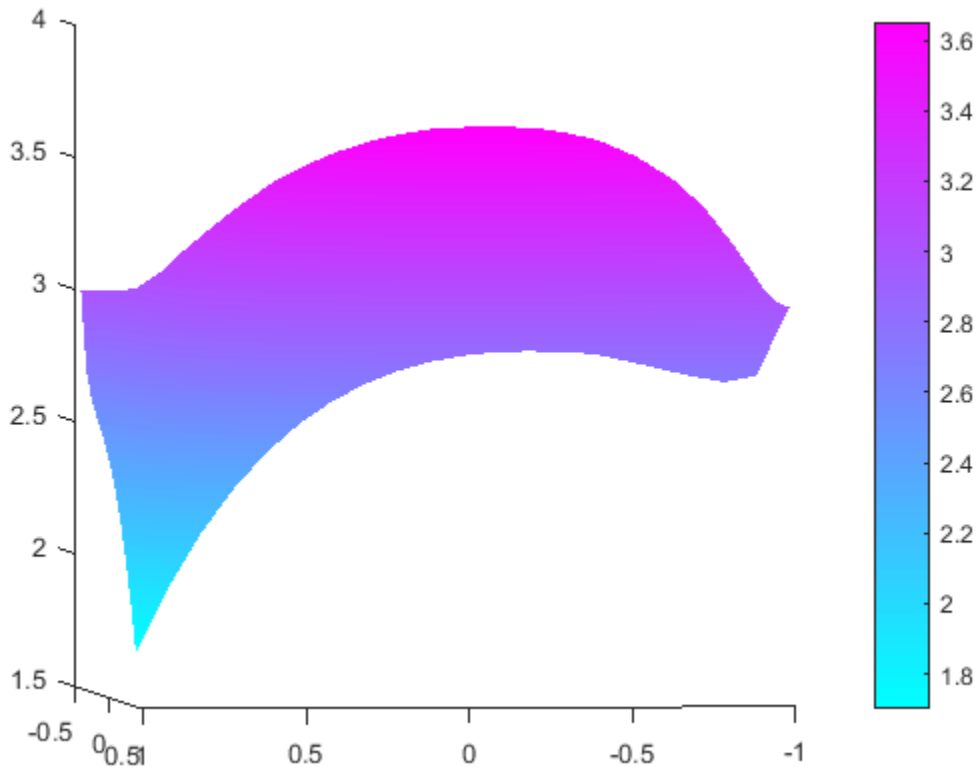

Constant Boundary Conditions for a System of Equations

This example creates constant boundary conditions for a PDE system.

Create a rectangle and view its edge labels.

```
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
sf = 'R1';
ns = sf';
% Create geometry
g = decsg(R1,sf,ns);
pg = pdeGeometryFromEdges(g); % Create pdeGeometry object
pdegplot(g,'EdgeLabels','on')
xlim([-1.1 1.1])
axis equal
```
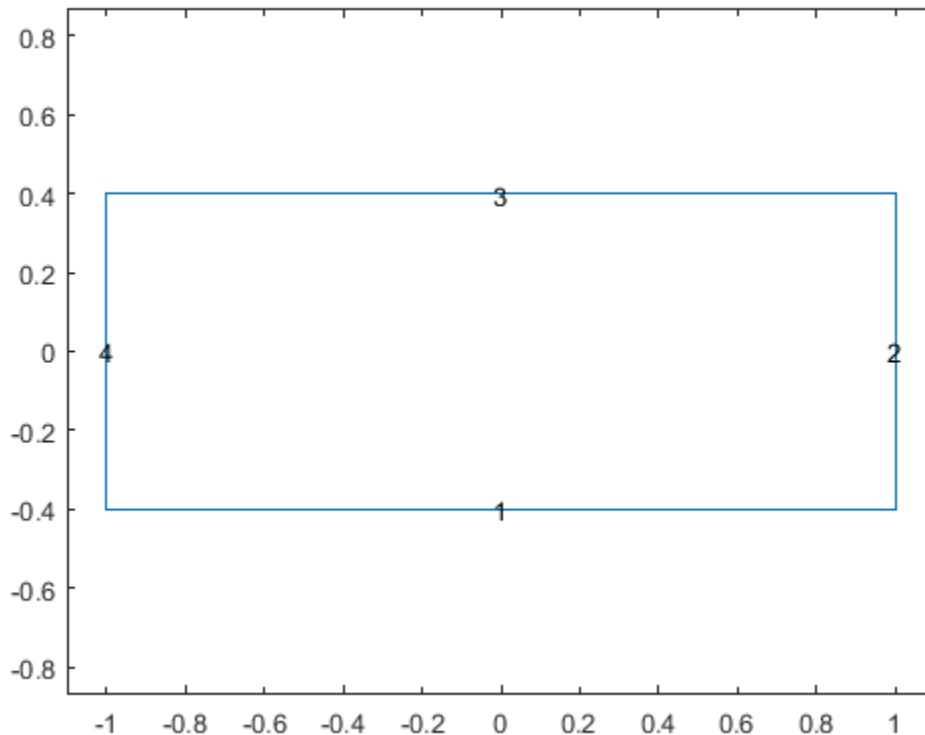

Create Dirichlet conditions for an N = 2 system. Set the Dirichlet condition values at edge 2 to [0;1]. Set the $u(2)$ component on edge 4 to the value 3.

```
bc2 = pdeBoundaryConditions(pg.Edges(2),'u',[0;1]);
```

```
bc4 = pdeBoundaryConditions(pg.Edges(4),'u',3,'EquationIndex',2);
```

Create Neumann conditions for edges 1 and 3 to have $q = 0$, $g = $ [-2,2].

```
bc13 = pdeBoundaryConditions(pg.Edges([1,3]),'g',[-2,2]); % q = 0 by default
```

Set a Neumann boundary condition on edge 4.

```
bc41 = pdeBoundaryConditions(pg.Edges(4),'g',[-2,0]);
```

Solve an elliptic problem with these boundary conditions, and with $f = $ [2;4], $a = 0$, $c = 1$.

```
[p,e,t] = initmesh(g);
[p,e,t] = refinemesh(g,p,e,t);
problem = pde(2); % N = 2
problem.BoundaryConditions = [bc2,bc4,bc13,bc41];
f = [2;4];
a = 0;
c = 1;
u = assempde(problem,p,e,t,c,a,f);
u2 = reshape(u,[],2); % Each column of u2 has one component of the solution
subplot(1,2,1)
pdeplot(p,e,t,'xydata',u2(:,1),'zdata',u2(:,1),'colorbar','off')
subplot(1,2,2)
pdeplot(p,e,t,'xydata',u2(:,2),'zdata',u2(:,2),'colorbar','off')
```
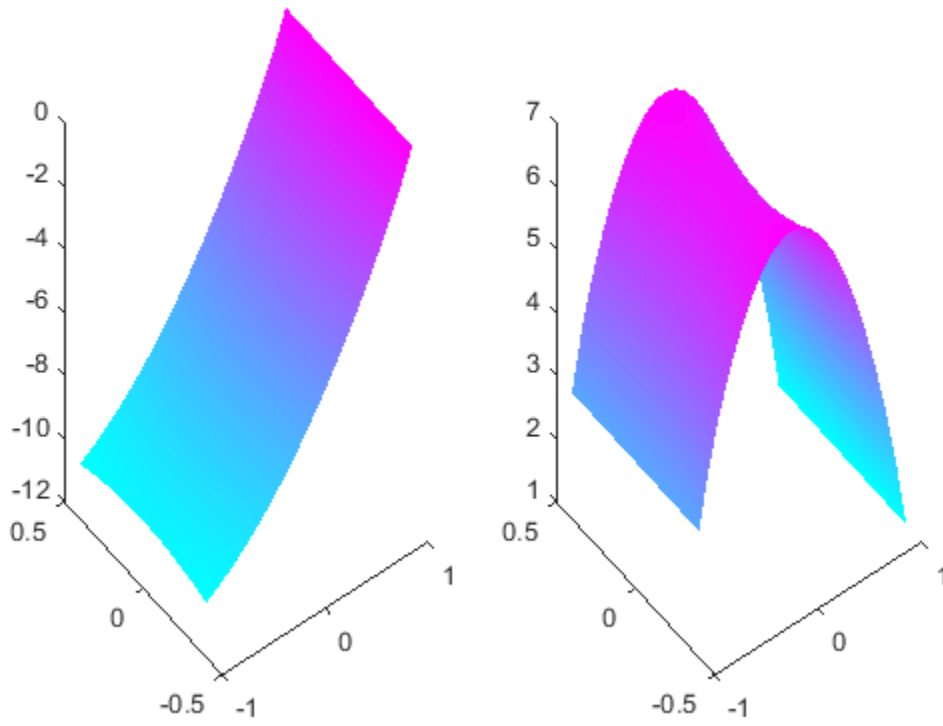
See Also
pdeGeometryFromEdges

pdecgrad

Flux of PDE solution

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `evaluateGradient`.

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. See "PDE Coefficients".

The scalar optional argument `time` is used for parabolic and hyperbolic problems, if `c` depends on `t`, the time.

The optional argument `sdl` restricts the computation to the subdomains in the list `sdl`.

See Also
`evaluateGradient`

pdecirc

Draw circle

pdecirc opens the PDE app and draws a circle. If, instead, you want to draw circles in a MATLAB figure, use the plot function such as t = linspace(0,2*pi);plot(cos(t),sin(t)) or plot(0,0,'o','MarkerSize',100), or the rectangle function with the Curvature name-value pair set to [1 1], or the Image Processing Toolbox™ viscircles function.

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 PDE app 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 the PDE app is updated to include the circle. You can export the Geometry Description matrix from the PDE app 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 the PDE app and draws a unit circle.

pdecirc(0,0,1)

See Also
pdeellip | pdepoly | pderect | pdetool

pdecont

Shorthand command for contour plot

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow.

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.

The geometry of the PDE problem is given by the mesh data `p` and `t`. For details on the mesh data representation, see "Mesh Data" on page 2-211.

`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

Solve eigenvalue PDE problem

Compatibility

pdeeig IS NOT RECOMMENDED. Use solvepdeeig instead.

Syntax

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

Description

`[v,l] = pdeeig(model,c,a,d,r)` produces the solution to the FEM formulation of the scalar PDE eigenvalue problem

$$-\nabla \cdot (c \nabla u) + au = \lambda d u \text{ on } \Omega$$

or the system PDE eigenvalue problem

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{a u} = \lambda d u \text{ on } \Omega,$$

with geometry, boundary conditions, and mesh specified in `model`, a `PDEModel` object. See "Solve Problems Using Legacy PDEModel Objects" on page 2-11.

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.

`[v,l] = pdeeig(b,p,e,t,c,a,d,r)` solves for boundary conditions described in `b`, and the finite element mesh in `[p,e,t]`.

`[v,l] = pdeeig(Kc,B,M,r)` produces the solution to the generalized sparse matrix eigenvalue problem

$Kc\ u_i = \lambda B'MBu_i$

$u = Bu_i$

with Real(λ) in the interval r.

Examples

Eigenvalues and Eigenvectors of the L-Shaped Membrane

Compute the eigenvalues that are less than 100, and compute the corresponding eigenmodes for

$-\nabla u = \lambda u$

on the geometry of the L-shaped membrane.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
generateMesh(model,'Hmax',0.02);
c = 1;
a = 0;
d = 1;
r = [-Inf 100];
[v,l] = pdeeig(model,c,a,d,r);
l(1)                      % first eigenvalue

ans =

    9.6481
```

Display the first eigenmode, and compare it to the built-in `membrane` plot.

```
pdeplot(model,'xydata',v(:,1),'zdata',v(:,1));
```
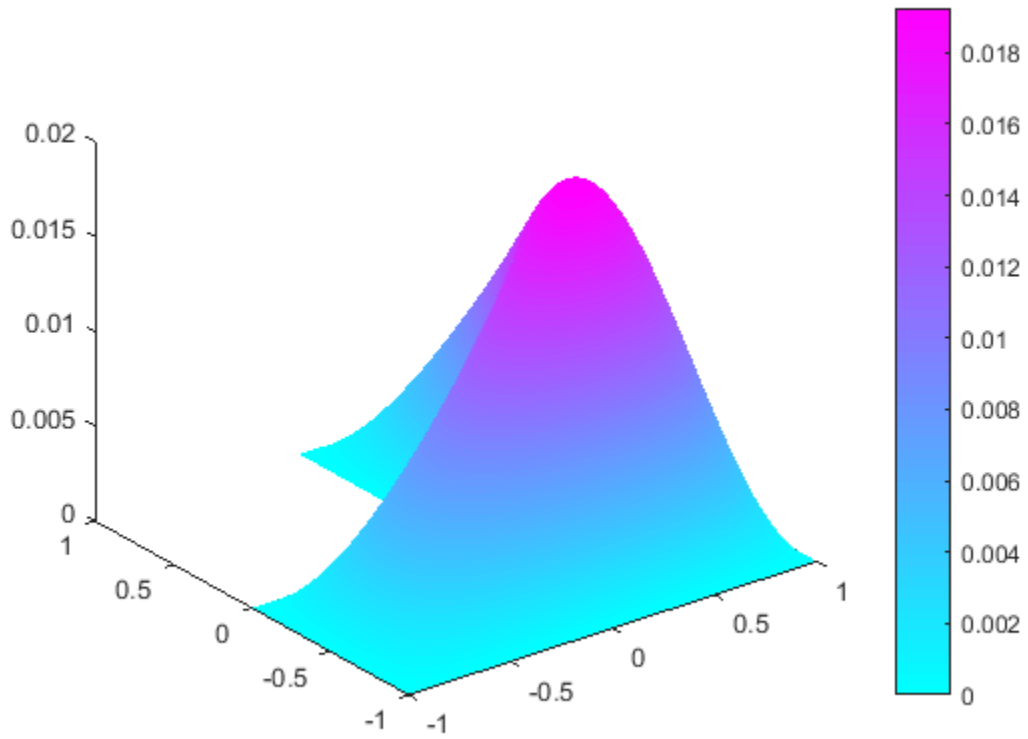
```
figure
membrane(1,20,9,9)        % the MATLAB function
```
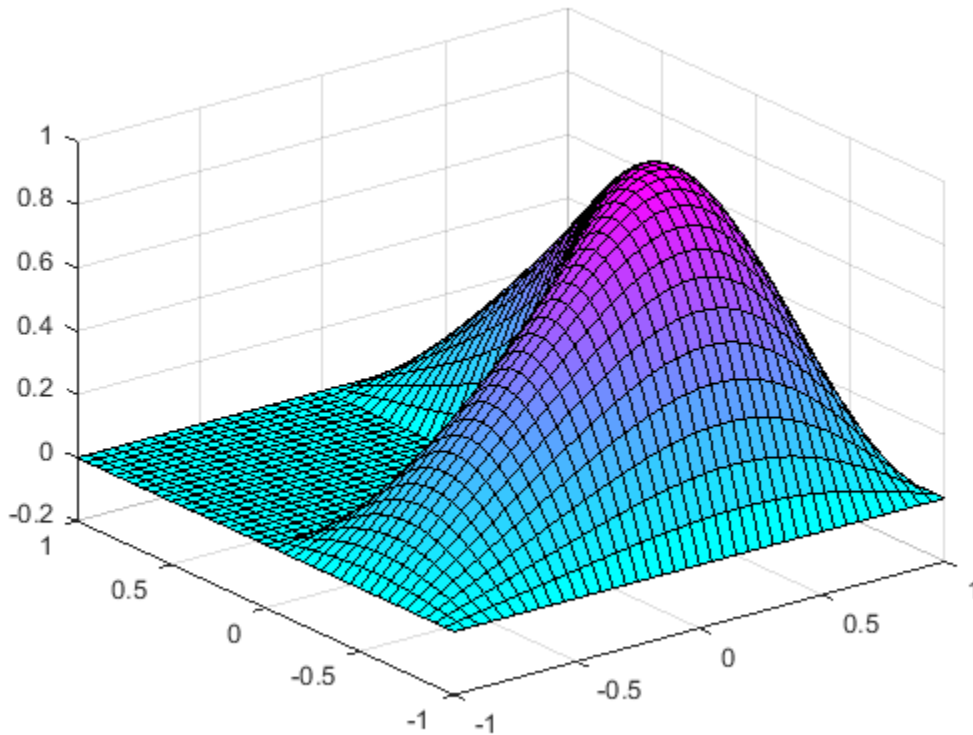
Compute the sixteenth eigenvalue, and plot the sixteenth eigenmode.

```
l(16)                       % sixteenth eigenvalue

ans =

   92.4658

figure
pdeplot(model,'xydata',v(:,16),'zdata',v(:,16));    % sixteenth eigenmode
```
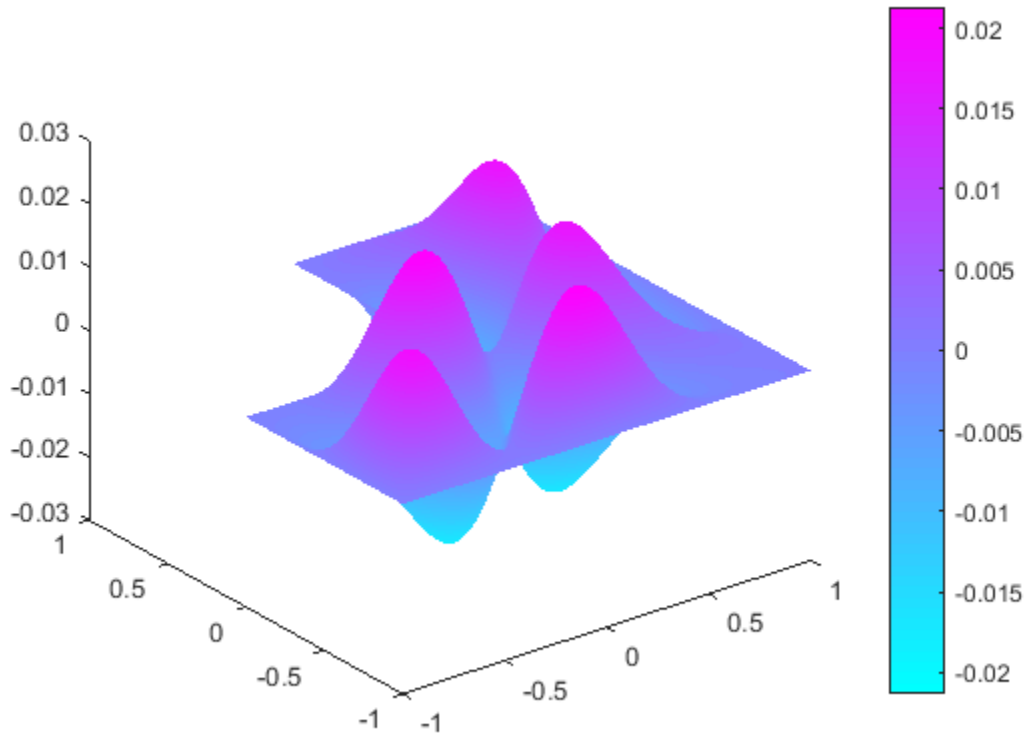
Eigenvalues and Eigenvectors of the L-Shaped Membrane Using Legacy Syntax

Compute the eigenvalues that are less than 100, and compute the corresponding eigenmodes for

$-\nabla u = \lambda u$

on the geometry of the L-shaped membrane, using the legacy syntax.

Use the geometry in `lshapeg`. For more information about this syntax, see "Create Geometry Using a Geometry Function" on page 2-26.

```
g = @lshapeg;
```

```
pdegplot(g,'EdgeLabels','on')
axis equal
ylim([-1.1,1.1])
```

Set zero Dirichlet boundary conditions using the `lshapeb` function. For more information about this syntax, see "Boundary Conditions by Writing Functions" on page 2-199.

```
b = @lshapeb;
```

Set coefficients $c = 1$, $a = 0$, and $d = 1$. Collect eigenvalues from 0 through 100.

```
c = 1;
a = 0;
d = 1;
r = [-Inf 100];
```

Generate a mesh and solve the eigenvalue problem.

```
[p,e,t] = initmesh(g,'Hmax',0.02);
[v,l] = pdeeig(b,p,e,t,c,a,d,r);
```

Find the first eigenvalue.

```
l(1)

ans =

    9.6481
```

Eigenvalues and Eigenvectors Using Finite Element Matrices

Import a simple 3-D geometry and find eigenvalues and eigenvectors from the associated finite element matrices.

Create a model and import the `'BracketWithHole.stl'` geometry.

```
model = createpde();
importGeometry(model,'BracketWithHole.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
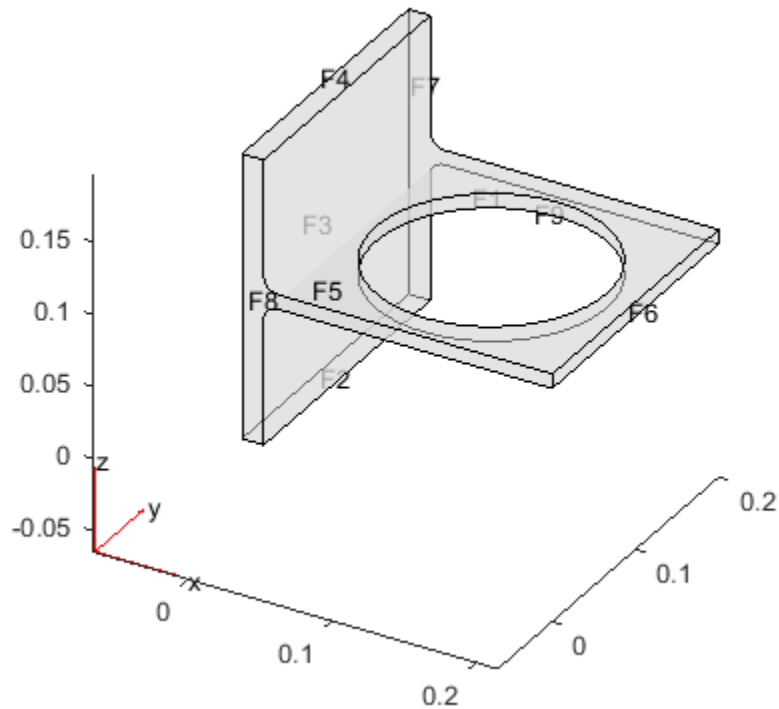
Set coefficients $c = 1$, $a = 0$, and $d = 1$. Collect eigenvalues from 0 through 100.

```
c = 1;
a = 0;
d = 1;
r = [-Inf 100];
```

Generate a mesh for the model.

```
generateMesh(model);
```

Create the associated finite element matrices.

```
[Kc,~,B,~] = assempde(model,c,a,0);
```

```
[~,M,~] = assema(model,0,d,0);
```

Solve the eigenvalue problem.

```
[v,l] = pdeeig(Kc,B,M,r);
```

Look at the first two eigenvalues.

```
l([1,2])

ans =

   -0.0000
   42.7872
```

Plot the solution corresponding to eigenvalue 2.

```
pdeplot3D(model,'colormapdata',v(:,2))
```
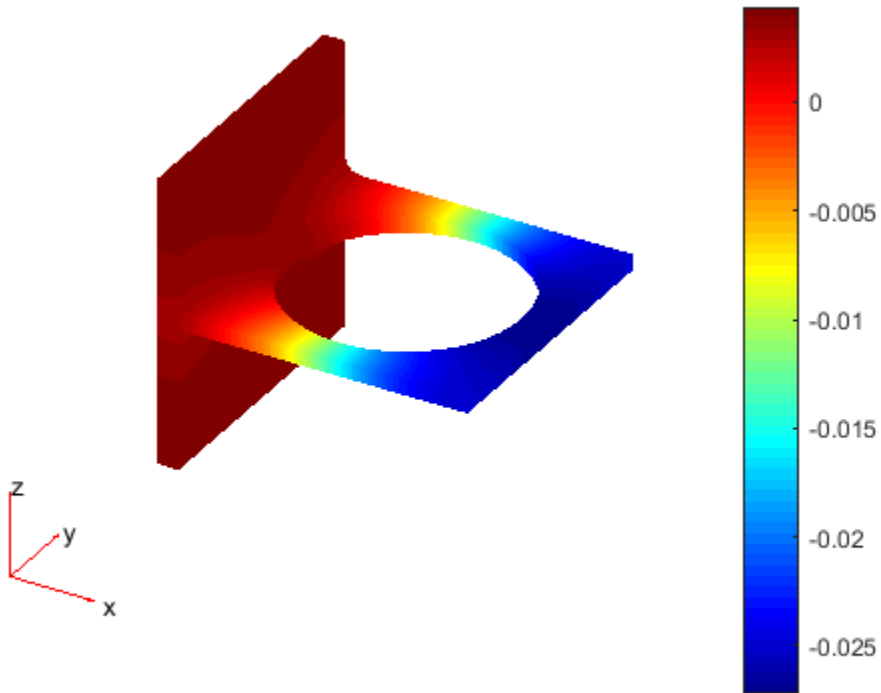
Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

`c` — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. c represents the *c* coefficient in the scalar PDE

$$-\nabla \cdot (c\nabla u) + au = \lambda du \text{ on } \Omega$$

or the system PDE eigenvalue problem

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

There are a wide variety of ways of specifying c, detailed in "c Coefficient for Systems" on page 2-125. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

a — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. a represents the *a* coefficient in the scalar PDE

$$-\nabla \cdot (c\nabla u) + au = \lambda du \text{ on } \Omega$$

or the system PDE eigenvalue problem

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

There are a wide variety of ways of specifying a, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

d — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. d represents the d coefficient in the scalar PDE

$$-\nabla \cdot (c\nabla u) + au = \lambda du \text{ on } \Omega$$

or the system PDE eigenvalue problem

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

There are a wide variety of ways of specifying d, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

r — Eigenvalue range
two-element real vector

Eigenvalue range, specified as a two-element real vector. Real parts of eigenvalues λ fall in the range $r(1) \leq \lambda \leq r(2)$. `r(1)` can be `-Inf`. The algorithm returns all eigenvalues in this interval in the `l` output, up to a maximum of 99 eigenvalues.

Example: `[-Inf,100]`

Data Types: `double`

b — Boundary conditions
boundary matrix | boundary file

Boundary conditions, specified as a boundary matrix or boundary file. Pass a boundary file as a function handle or as a string naming the file.

- A boundary matrix is generally an export from the PDE app. For details of the structure of this matrix, see "Boundary Matrix for 2-D Geometry" on page 2-171.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-199.

For more information on boundary conditions, see "Forms of Boundary Condition Specification" on page 2-170.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

p — Mesh nodes
output of `initmesh` | output of `meshToPet`

Mesh nodes, specified as the output of `initmesh` or `meshToPet`. For the structure of a `p` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

e — Mesh edges
output of `initmesh` | output of `meshToPet`

Mesh edges, specified as the output of `initmesh` or `meshToPet`. For the structure of `e`, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

t — Mesh elements
output of `initmesh` | output of `meshToPet`

Mesh elements, specified as the output of `initmesh` or `meshToPet`. Mesh elements are the triangles or tetrahedra that form the finite element mesh. For the structure of a `t` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

Kc — Stiffness matrix
sparse matrix | full matrix

Stiffness matrix, specified as a sparse matrix or as a full matrix. See "Elliptic Equations" on page 5-2. Typically, Kc is the output of assempde.

B — Dirichlet nullspace
sparse matrix

Dirichlet nullspace, returned as a sparse matrix. See "Algorithms" on page 6-56. Typically, B is the output of assempde.

M — Mass matrix
sparse matrix | full matrix

Mass matrix. specified as a sparse matrix or a full matrix. See "Elliptic Equations" on page 5-2.

To obtain the input matrices for pdeeig, hyperbolic or parabolic, run both assema and assempde:

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Note: Create the M matrix using assema with d, not a, as the argument before f.

Data Types: double
Complex Number Support: Yes

Output Arguments

v — Eigenvectors
matrix

Eigenvectors, returned as a matrix. Suppose

- Np is the number of mesh nodes
- *N* is the number of equations
- ev is the number of eigenvalues returned in l

Then v has size Np*N-by-ev. Each column of v corresponds to the eigenvectors of one eigenvalue. In each column, the first Np elements correspond to the eigenvector of equation 1 evaluated at the mesh nodes, the next Np elements correspond to equation 2, etc.

Note: Eigenvectors are determined only up to multiple by a scalar, including a negative scalar.

---

1 — Eigenvalues
vector

Eigenvalues, returned as a vector. The real parts of l are in the interval r. The real parts of l are monotone increasing.

Limitations

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.

More About

Tips

- The equation coefficients cannot depend on the solution u or its gradient.

- "Eigenvalue Equations" on page 5-22

See Also

```
solvepdeeig
```

Introduced before R2006a

pdeellip

Draw ellipse

`pdeellip` opens the PDE app and draws an ellipse. If, instead, you want to draw ellipses in a MATLAB figure, use the `plot` function such as `t = linspace(0,2*pi);plot(2*cos(t),sin(t-pi/6))`, or the `rectangle` function with the `Curvature` name-value pair set to `[1 1]`.

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 PDE app 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 the PDE app is updated to include the ellipse. You can export the Geometry Description matrix from the PDE app 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 the PDE app and draws an ellipse.

```
pdeellip(0,0,1,0.3,pi/4)
```

See Also
`pdecirc` | `pdepoly` | `pderect` | `pdetool`

pdeent

Indices of triangles neighboring given set of triangles

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `generateMesh`.

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
generateMesh

pdeGeometryFromEdges

Create geometry object

Compatibility

Note: pdeGeometryFromEdges WILL BE REMOVED IN A FUTURE RELEASE. Use
`geometryFromEdges` instead.

Syntax

```
pg = pdeGeometryFromEdges(g)
```

Description

`pg = pdeGeometryFromEdges(g)` returns a geometry object from a decomposed
geometry description or a geometry file.

Examples

Geometry from Decomposed Solid Geometry

This example geometry is a rectangle with a circular hole.

Create a rectangle and a circle. Combine them using the set formula `'R1-C1'`, which
subtracts the circle from the rectangle.

```
% 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
g = decsg(geom,sf,ns);
```

Create the geometry object.

```
pg = pdeGeometryFromEdges(g);
```

Geometry from a Geometry Function

This example creates geometry from a function.

The `circleg` function ships with Partial Differential Equation Toolbox software. It describes a circle centered at (0,0) with radius 1.

```
pg = pdeGeometryFromEdges(@circleg);
```

Input Arguments

g — Geometry description
decomposed geometry matrix | function handle to a geometry file

Geometry description, specified as a decomposed geometry matrix or a function handle to a geometry file.

Specify g as one of the following:

- Decomposed geometry matrix:

    - Export from the PDE app
    - Output of `decsg`

- Function handle to a geometry file (see "Create Geometry Using a Geometry Function" on page 2-26)

Example: `pg = pdeGeometryFromEdges(@circleg)`

Data Types: `double` | `function_handle`

Output Arguments

pg — Geometry container
edges

Geometry container, returned as edges. `pg` contains the edge objects, `pg.Edges`, that together make the geometry.

See Also
`geometryFromEdges`

Introduced in R2014b

pdegplot

Plot PDE geometry

Compatibility

PLOTTING PDE GEOMETRIES IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
pdegplot(g)
h = pdegplot(g)
h = pdegplot(g,Name,Value)
```

Description

`pdegplot(g)` plots the geometry of a PDE problem, as described in `g`.

`h = pdegplot(g)` returns handles to the figure axes.

`h = pdegplot(g,Name,Value)` plots with additional options specified by one or more `Name,Value` pair arguments.

Examples

Plot 2-D 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 0.17;
1 -0.5 -0.45 -0.5 -0.55 0.5 0.55 0.5 0.45 0.17 1;
0 -0.25 -0.3 -0.25 -0.2 -0.25 -0.3 -0.25 -0.2 0 -0.74;
0 -0.3 -0.25 -0.2 -0.25 -0.3 -0.25 -0.2 -0.25 -0.74 0;
```

```
0 0 0 0 0 0 0 0 0 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)
```
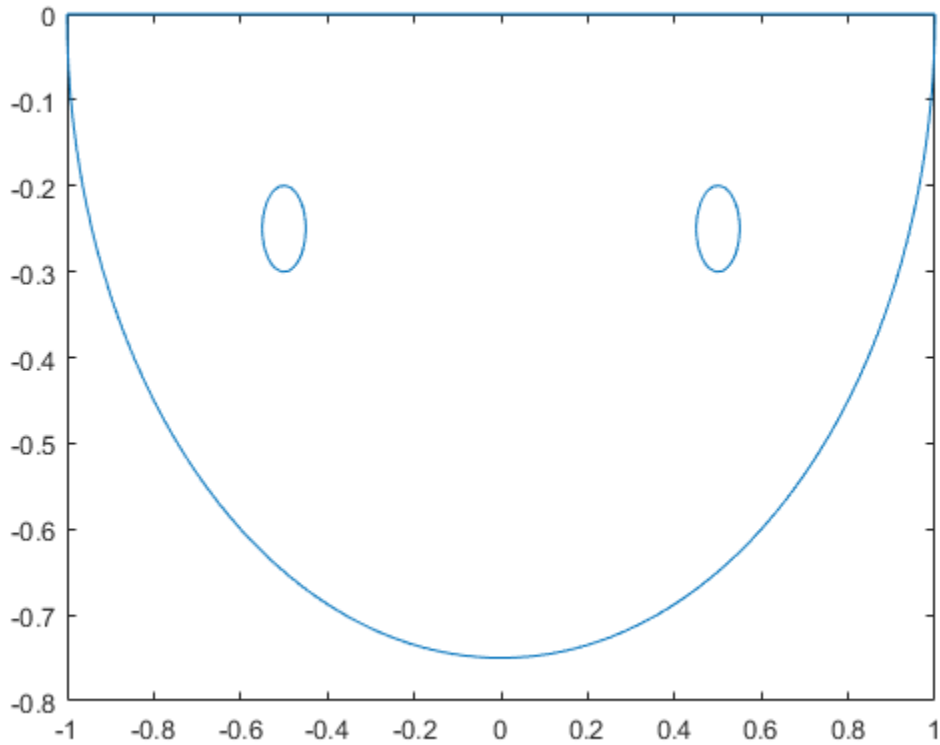

View the edge labels and the subdomain label. Add space at the top of the plot to see the top edge clearly.

```
pdegplot(g,'EdgeLabels','on','SubdomainLabels','on')
ylim([-.8,.1])
```
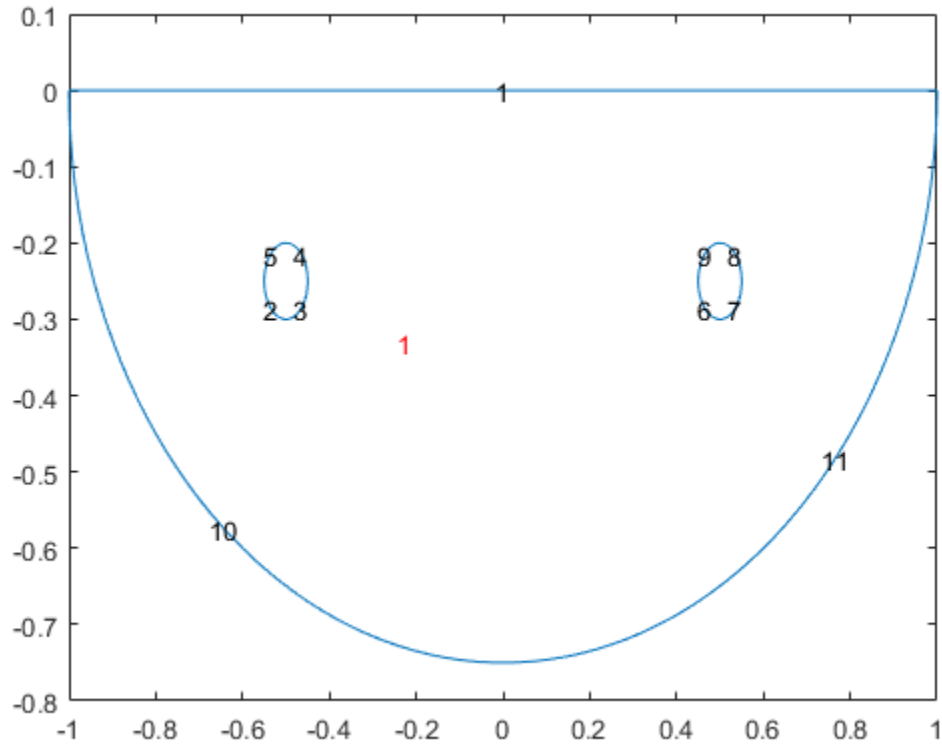
Plot 3-D Geometry

Import a 3-D geometry file. Plot the geometry and turn on face labels.

```
model = createpde;
importGeometry(model,'BracketTwoHoles.stl');
pdegplot(model,'FaceLabels','on')
```
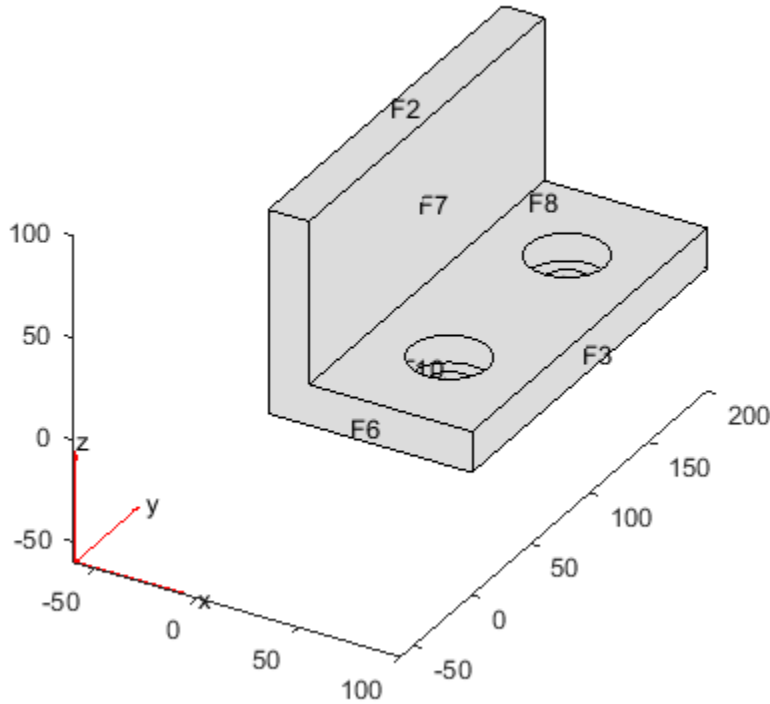
- "Solve PDE with Coefficients in Functional Form" on page 2-79
- "Create Geometry and Remove Subdomains" on page 2-22
- "Create and View 3-D Geometry" on page 2-47

Input Arguments

g — Geometry description
PDEModel object | output of decsg | decomposed geometry matrix | name of geometry file | function handle to geometry file

Geometry description, specified by one of the following:

- `PDEModel` object
- Output of `decsg`
- Decomposed geometry matrix (see "Decomposed Geometry Data Structure" on page 2-24)
- Name of geometry file (see "Create Geometry Using a Geometry Function" on page 2-26)
- Function handle to geometry file (see "Create Geometry Using a Geometry Function" on page 2-26)

Data Types: `double` | `char` | `function_handle`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example:

`'EdgeLabels'` — Boundary edge labels of 2-D geometry
`'off'` (default) | `'on'`

Boundary edge labels of 2-D geometry, specified as the comma-separated pair consisting of `'EdgeLabels'` and `'off'` or `'on'`.

Example: `pdegplot(g,'EdgeLabels','on')`

Data Types: `char`

`'SubdomainLabels'` — Subdomain labels of 2-D geometry
`'off'` (default) | `'on'`

Subdomain labels of 2-D geometry, specified as the comma-separated pair consisting of `'EdgeLabels'` and `'off'` or `'on'`.

Example: `pdegplot(g,'SubdomainLabels','on')`

Data Types: `char`

`'FaceLabels'` — Boundary face labels of 3-D geometry
`'off'` (default) | `'on'`

Boundary face labels of 3-D geometry, specified as the comma-separated pair consisting of `'FaceLabels'` and `'off'` or `'on'`.

Example: `pdegplot(g,'FaceLabels','on')`

Data Types: `char`

Output Arguments

h — Handles to the figure axes
vector

Handles to the figure axes, returned as a vector.

Alternative Functionality

App

If you create 2-D geometry in the PDE app, you can view the geometry from Boundary Mode. To see the edge labels, select **Boundary > Show Edge Labels**. To see the subdomain labels, select **PDE > Show Subdomain Labels**.

More About

See Also
`decsg` | `importGeometry` | `pdetool` | `wgeom`

Introduced before R2006a

pdegrad

Gradient of PDE solution

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `evaluateGradient`.

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`.

The optional argument `sdl` restricts the computation to the subdomains in the list `sdl`.

See Also
`evaluateGradient`

Using pdeInterpolant Objects

Interpolant for nodal data to selected locations

Compatibility

A `pdeInterpolant` OBJECT SUPPORTS THE LEGACY WORKFLOW.
Using the [p,e,t] representation of FEMesh data is not recommended. Use
`interpolateSolution` and `evaluateGradient` to interpolate a PDE solution and its
gradient to arbitrary points without switching to a [p,e,t] representation.

Description

An interpolant allows you to evaluate a PDE solution at any point within the geometry.

Partial Differential Equation Toolbox solvers return solution values at the nodes,
meaning the mesh points. To evaluate an interpolated solution at other points within the
geometry, create a `pdeInterpolant` object, and then call the `evaluate` function.

Examples

Create interpolant

This example shows how to create a `pdeInterpolant` from the solution to a scalar PDE.

Solve the equation $-\Delta u = 1$ on the unit disk with zero Dirichlet conditions.

```
g0 = [1;0;0;1]; % circle centered at (0,0) with radius 1
sf = 'C1';
g = decsg(g0,sf,sf'); % decomposed geometry matrix
problem = allzerobc(g); % zero Dirichlet conditions
[p,e,t] = initmesh(g);
c = 1;
a = 0;
f = 1;
u = assempde(problem,p,e,t,c,a,f);
```

Construct an interpolant for the solution.

```
F = pdeInterpolant(p,t,u);
```

Evaluate the interpolant at the four corners of a square.

```
pOut = [0,1/2,1/2,0;
    0,0,1/2,1/2];
uOut = evaluate(F,pOut)
```

```
uOut =

    0.2485
    0.1854
    0.1230
    0.1852
```

The values `uOut(2)` and `uOut(4)` are nearly equal, as they should be for symmetric points in this symmetric problem.

Object Functions

evaluate   Interpolate data to selected locations

Create Object

`F = pdeInterpolant(p,t,u)` returns an interpolant `F` based on the data points `p`, elements `t`, and data values at the points, `u`.

Argument Name	Description
p	Data point locations, specified as a matrix with two or three rows. Each column of p is a 2-D or 3-D point. For details, see "Mesh Data" on page 2-211.
	For 2-D problems, construct p using the initmesh function, or export from the **Mesh** menu of the PDE app. For 2-D or 3-D geometry using a PDEModel object, obtain p using the meshToPet function on model.Mesh. For example, [p,e,t] = initmesh(g) or [p,e,t] = meshToPet(model.Mesh).

Argument Name	Description
t	Triangulation elements, specified as a matrix. For details, see "Mesh Data" on page 2-211. For 2-D problems, construct t using the initmesh function, or export from the **Mesh** menu of the PDE app. For 2-D or 3-D geometry using a PDEModel object, obtain t using the meshToPet function on model.Mesh. For example, [p,e,t] = initmesh(g) or [p,e,t] = meshToPet(model.Mesh).
u	Data values to interpolate, specified as a vector or matrix. Typically, u is the solution of a PDE problem returned by assempde, parabolic, hyperbolic, or another solver. For example, u = assempde(b,p,e,t,c,a,f). You can also export u from the **Solve** menu of the PDE app. The dimensions of the matrix u depend on the problem. If np is the number of columns of p, and N is the number of equations in the PDE system, then u has N*np rows. The first np rows correspond to equation 1, the next np rows correspond to equation 2, etc. For parabolic or hyperbolic problems, u has one column for each solution time; otherwise, u is a column vector.

Tip Use meshToPet to obtain the p and t data for interpolation using pdeInterpolant.

See Also
evaluate | tri2grid

More About
· "Mesh Data" on page 2-211

Introduced in R2014b

pdeintrp

Interpolate from node data to triangle midpoint data

Compatibility

pdeintrp IS NOT RECOMMENDED. Use `interpolateSolution` and `evaluateGradient` instead.

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
`interpolateSolution` | `solvepde` | `evaluateGradient`

pdejmps

Error estimates for adaptation

Compatibility

pdejmps IS NOT RECOMMENDED.

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 2-66 and "Coefficients for Systems of PDEs" on page 2-93 for details. c, a, and f must be expanded, so that columns correspond to triangles.

The formula for computing the error indicator $E(K)$ for each triangle K is

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

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 2-214.

pdemdlcv

Convert Partial Differential Equation Toolbox 1.0 model files to 1.0.2 format

Compatibility

pdemdlcv WILL BE REMOVED IN A FUTURE RELEASE.

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.

pdemesh

Plot PDE mesh

Compatibility

MESHING IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
pdemesh(p,e,t)

pdemesh(p,e,t,u)

pdemesh(model)

pdemesh(model,u)

h = pdemesh( ___ )
```

Description

pdemesh(p,e,t) plots the mesh specified by the mesh data p, e, and t.

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 "Mesh Data" on page 2-211.

This command is just shorthand for the calls

```
pdeplot(p,e,t) % 2-D mesh
pdeplot3D(p,e,t) % 3-D mesh
pdeplot(p,e,t,'zdata',u) % 2-D only
```

If you want to have more control over your mesh plot, use `pdeplot` or `pdeplot3D` instead of `pdemesh`.

`pdemesh(model)` plots the mesh contained in a 2-D or 3-D `model` object of type `PDEModel`.

For 2-D geometry only, `pdemesh(model,u)` plots solution data `u` as a 3-D plot.

For any input arguments, `h = pdemesh( ___ )` additionally returns handles to the plotted axes objects.

Examples

Plot the mesh for the geometry of the L-shaped membrane.

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

Now solve Poisson's equation $-\Delta u = 1$ over the geometry defined by the L-shaped membrane. Use Dirichlet boundary conditions $u = 0$ on $\partial\Omega$, and plot the result.

```
u = assempde('lshapeb',p,e,t,1,0,1);
pdemesh(p,e,t,u)
```

More About

- "Mesh Data" on page 2-211
- "Solve PDE with Coefficients in Functional Form" on page 2-79

See Also
`pdecont` | `pdeplot` | `pdesurf`

Introduced before R2006a

Using PDEModel Objects

PDE model container

Compatibility

A PDEModel OBJECT IS VALID IN BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Description

A PDEModel object contains information about a PDE problem: the number of equations, geometry, mesh, and boundary conditions.

Examples

Create and Populate a PDE Model

Create and populate a PDEModel object.

Create a container for a scalar PDE ($N = 1$).

```
model = createpde();
```

Include a torus geometry, zero Dirichlet boundary conditions, coefficients for Poisson's equation, and the default mesh.

```
importGeometry(model,'Torus.stl');
applyBoundaryCondition(model,'face',1,'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
generateMesh(model);
```

Solve the PDE.

```
results = solvepde(model)
```

```
results =

  StationaryResults with properties:

    NodalSolution: [30656x1 double]
       XGradients: [30656x1 double]
       YGradients: [30656x1 double]
       ZGradients: [30656x1 double]
             Mesh: [1x1 FEMesh]
```

- "Solve Problems Using PDEModel Objects" on page 2-14

Properties

`PDESystemSize` — Number of equations
1 (default) | positive integer

Number of equations, *N*, returned as a positive integer. See "Systems of PDEs" on page 2-65.

Example: 1

Data Types: `double`

`BoundaryConditions` — PDE boundary conditions
vector of `BoundaryCondition` objects

PDE boundary conditions, returned as a vector of `BoundaryCondition` objects. You create boundary conditions using the `applyBoundaryCondition` function

`Geometry` — Geometry description
geometry object

Geometry description, returned as a geometry object.

- `AnalyticGeometry` object for 2-D geometry. You create this geometry using the `geometryFromEdges` function.
- `DiscreteGeometry` object for 3-D geometry. You create this geometry using the `importGeometry` function or the `geometryFromMesh` function.

`Mesh` — Mesh for solution
`FEMesh` object

Mesh for solution, returned as an `FEMesh` object. You create the mesh using the `generateMesh` function.

`IsTimeDependent` — Indicator if model is time-dependent
`0` (false) (default) | `1` (true)

Indicator if model is time-dependent, returned as `1` (`true`) or `0` (`false`). The property is `true` when the m or d coefficient is nonzero, and is `false` otherwise.

`EquationCoefficients` — PDE coefficients
vector of `CoefficientAssignment` objects

PDE coefficients, returned as a vector of `CoefficientAssignment` objects. See `specifyCoefficients`.

`InitialConditions` — Initial conditions or initial solution
`GeometricInitialConditions` object

Initial conditions or initial solution, returned as a `GeometricInitialConditions` object. For time-dependent problems, you must give one or two give initial conditions: one if the m coefficient is zero, and two if the m coefficient is nonzero. For nonlinear stationary problems, you can optionally give an initial solution that `solvepde` uses to start its iterations. See `setInitialConditions`.

Object Functions

applyBoundaryCondition	Add boundary condition to PDEModel container
generateMesh	Create triangular or tetrahedral mesh
geometryFromEdges	Create 2-D geometry
geometryFromMesh	Create 3-D geometry from a triangulated mesh
importGeometry	Import 3-D geometry
setInitialConditions	Give initial conditions or initial solution
specifyCoefficients	Specify coefficients in a PDE model

Create Object

`createpde` returns a `PDEModel` container. Initially, the only property that is nonempty is `PDESystemSize`, which is `1` for scalar problems.

See Also

applyBoundaryCondition | createpde | generateMesh | geometryFromEdges | geometryFromMesh | importGeometry | pdegplot | pdeplot | pdeplot3D | setInitialConditions | specifyCoefficients

Introduced in R2015a

pdenonlin

Solve nonlinear elliptic PDE problem

Compatibility

pdenonlin IS NOT RECOMMENDED. Use solvepde instead.

Syntax

```
u = pdenonlin(model,c,a,f)
u = pdenonlin(b,p,e,t,c,a,f)
u = pdenonlin( ___ ,Name,Value)
[u,res] = pdenonlin( ___ )
```

Description

u = pdenonlin(model,c,a,f) solves the nonlinear PDE

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

with geometry, boundary conditions, and finite element mesh in model, and coefficients c, a, and f. In this context, "nonlinear" means some coefficient in c, a, or f depends on the solution u or its gradient. If the PDE is a system of equations (model.PDESystemSize > 1), then pdenonlin solves the system of equations

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

u = pdenonlin(b,p,e,t,c,a,f) solves the PDE with boundary conditions b, and finite element mesh (p,e,t).

u = pdenonlin(___ ,Name,Value), for any previous arguments, modifies the solution process with Name, Value pairs.

[u,res] = pdenonlin(___) also returns the norm of the Newton step residuals res.

Examples

Minimal Surface Problem

Solve a minimal surface problem. Because this problem has a nonlinear *c* coefficient, use `pdenonlin` to solve it.

Create a model and include circular geometry using the built-in `circleg` function.

```
model = createpde;
geometryFromEdges(model,@circleg);
```

Set the coefficients.

```
a = 0;
f = 0;
c = '1./sqrt(1+ux.^2+uy.^2)';
```

Set a Dirichlet boundary condition with value x^2.

```
boundaryfun = @(region,state)region.x.^2;
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,...
    'u',boundaryfun,'Vectorized','on');
```

Generate a mesh and solve the problem.

```
generateMesh(model,'Hmax',0.1);
u = pdenonlin(model,c,a,f);
pdeplot(model,'xydata',u,'zdata',u);
```
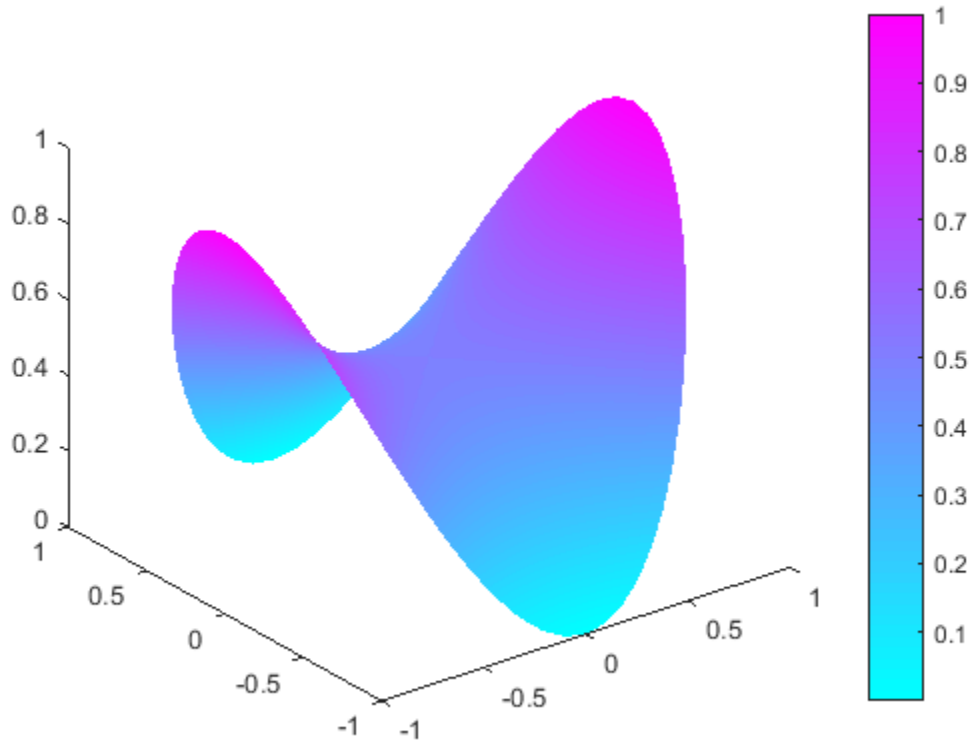
Minimal Surface Problem Using [p,e,t] Mesh

Solve the minimal surface problem using the legacy approach for creating boundary conditions and geometry.

Create the geometry using the built-in `circleg` function. Plot the geometry to see the edge labels.

```
g = @circleg;
pdegplot(g,'EdgeLabels','on')
axis equal
```
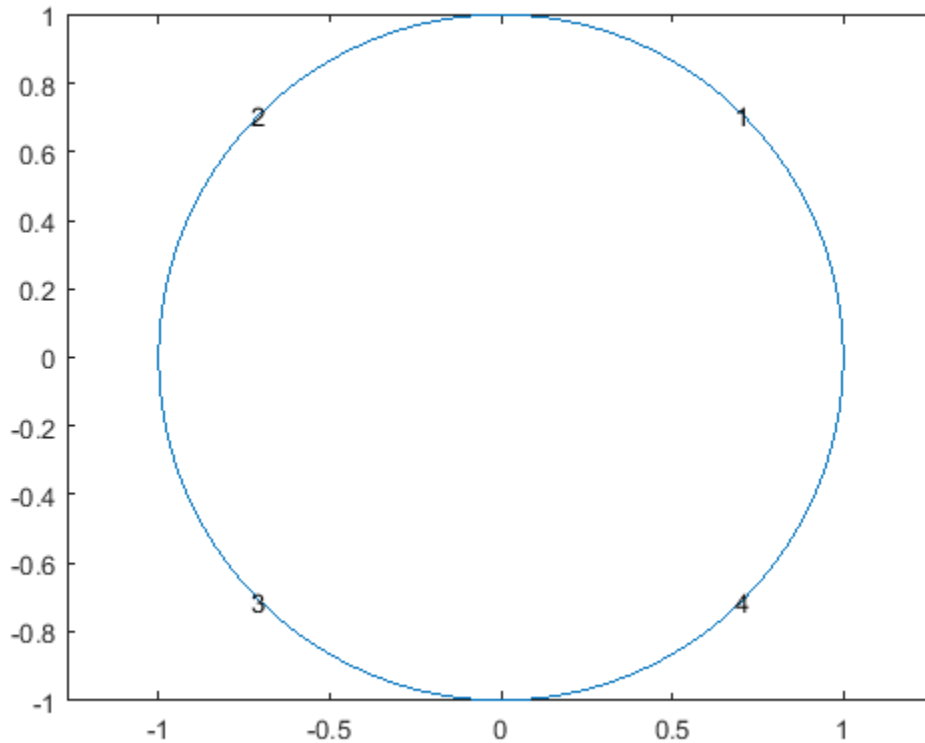
Create Dirichlet boundary conditions with value x^2. Create the following file and save it on your MATLAB path. For details of this approach, see "Boundary Conditions by Writing Functions" on page 2-199.

```matlab
function [qmatrix,gmatrix,hmatrix,rmatrix] = pdex2bound(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}
            hmatrix(k) = 1;
            hmatrix(k+ne) = 1;
            rmatrix(k) = x1^2;
            rmatrix(k+ne) = x2^2;
    end
end
```

Set the coefficients and boundary conditions.

```
a = 0;
f = 0;
c = '1./sqrt(1+ux.^2+uy.^2)';
b = @pdex2bound;
```

Generate a mesh and solve the problem.

```
[p,e,t] = initmesh(g,'Hmax',0.1);
u = pdenonlin(b,p,e,t,c,a,f);
pdeplot(p,e,t,'xydata',u,'zdata',u);
```
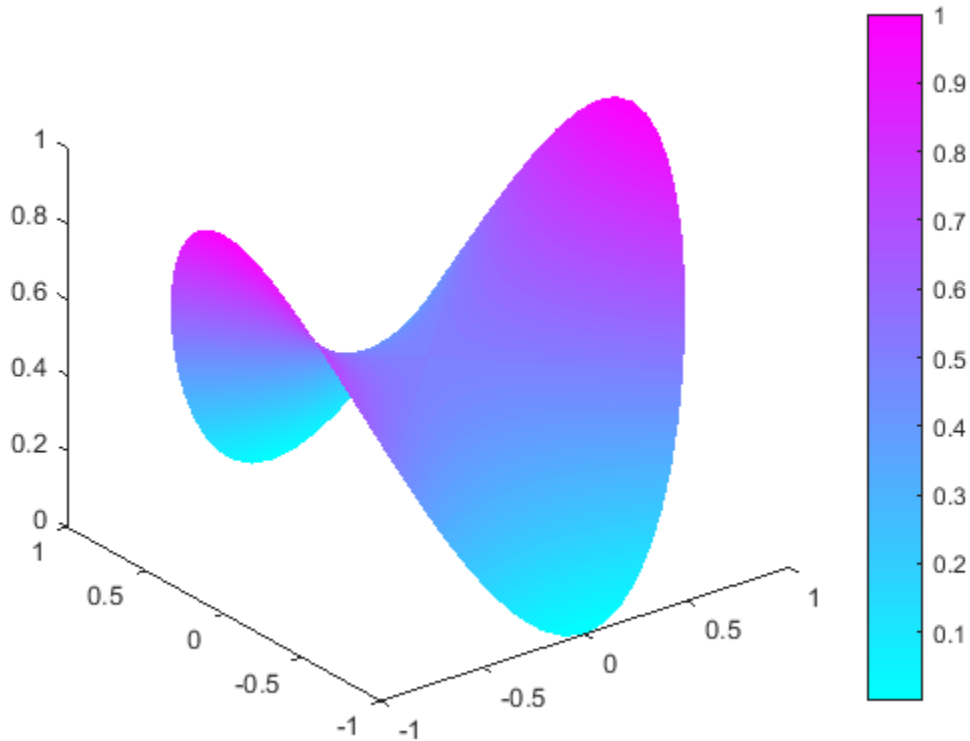
Nonlinear Problem with 3-D Geometry

Solve a nonlinear 3-D problem with nontrivial geometry.

Import the geometry from the `BracketWithHole.stl` file. Plot the geometry and face labels.

```
model = createpde();
importGeometry(model,'BracketWithHole.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
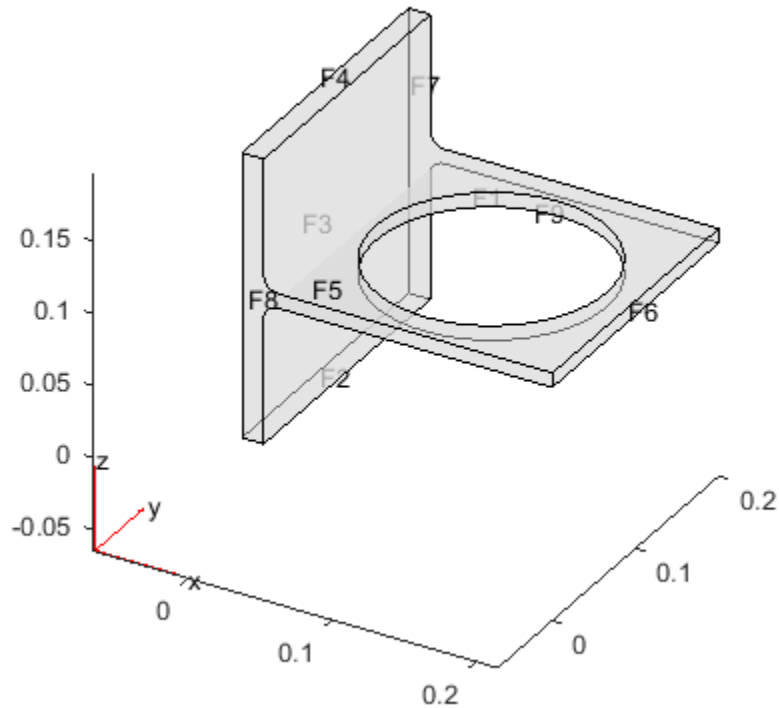
Set a Dirichlet boundary condition with value 1000 on the back face, which is face 3. Set the large faces 1 and 5, and also the circular face 9, to have Neumann boundary conditions with value g = −10. Do not set boundary conditions on the other faces. Those faces default to Neumann boundary conditions with value g = 0.

```
applyBoundaryCondition(model,'face',3,'u',1000);
applyBoundaryCondition(model,'face',[1,5,9],'g',-10);
```

Set the c coefficient to 1, f to 0.1, and a to the nonlinear value '0.1 + 0.001*u.^2'.

```
c = 1;
f = 0.1;
a = '0.1 + 0.001*u.^2';
```

Generate the mesh and solve the PDE. Start from the initial guess `u0` = 1000, which matches the value you set on face `3`. Turn on the `Report` option to observe the convergence during the solution.

```
generateMesh(model);
u = pdenonlin(model,c,a,f,'U0',1000,'Report','on');

Iteration     Residual      Step size  Jacobian: full
   0          3.5985e-01
   1          1.0182e-01    1.0000000
   2          3.0318e-02    1.0000000
   3          8.7499e-03    1.0000000
   4          1.9023e-03    1.0000000
   5          1.5581e-04    1.0000000
   6          1.2424e-06    1.0000000
```

Plot the solution on the geometry boundary.

```
pdeplot3D(model,'colormapdata',u);
```

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

`c` — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. `c` represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify `c` in various ways, detailed in "c Coefficient for Systems" on page 2-125. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `function_handle`
Complex Number Support: Yes

a — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. `a` represents the *a* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify `a` in various ways, detailed in "a or d Coefficient for Systems" on page 2-148. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `2*eye(3)`

Data Types: `double | char | function_handle`
Complex Number Support: Yes

f — PDE coefficient
scalar or matrix | character array | coefficient function

PDE coefficient, specified as a scalar or matrix, as a character array, or as a coefficient function. f represents the *f* coefficient in the scalar PDE

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

or in the system of PDEs

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

You can specify f in various ways, detailed in "f Coefficient for Systems" on page 2-98. See also "Scalar PDE Coefficients" on page 2-66, "Specify Scalar PDE Coefficients in String Form" on page 2-68, "Specify 2-D Scalar Coefficients in Function Form" on page 2-74, "Specify 3-D PDE Coefficients in Function Form" on page 2-77, and "Coefficients for Systems of PDEs" on page 2-93.

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double | char | function_handle`
Complex Number Support: Yes

b — Boundary conditions
boundary matrix | boundary file

Boundary conditions, specified as a boundary matrix or boundary file. Pass a boundary file as a function handle or as a string naming the file.

- A boundary matrix is generally an export from the PDE app. For details of the structure of this matrix, see "Boundary Matrix for 2-D Geometry" on page 2-171.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-199.

For more information on boundary conditions, see "Forms of Boundary Condition Specification" on page 2-170.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

p — Mesh nodes
output of `initmesh` | output of `meshToPet`

Mesh nodes, specified as the output of `initmesh` or `meshToPet`. For the structure of a `p` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

e — Mesh edges
output of `initmesh` | output of `meshToPet`

Mesh edges, specified as the output of `initmesh` or `meshToPet`. For the structure of `e`, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

t — Mesh elements
output of `initmesh` | output of `meshToPet`

Mesh elements, specified as the output of `initmesh` or `meshToPet`. Mesh elements are the triangles or tetrahedra that form the finite element mesh. For the structure of a `t` matrix, see "Mesh Data for [p,e,t] Triples: 2-D" on page 2-211 and "Mesh Data for [p,e,t] Triples: 3-D" on page 2-212.

Example: `[p,e,t] = initmesh(g)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example:

'Jacobian' — Approximation of Jacobian
`'full'` (3-D default) | `'fixed'` (2-D default) | `'lumped'`

Approximation of Jacobian, specified as `'full'`, `'fixed'`, or `'lumped'`.

- `'full'` means numerical evaluation of the full Jacobian based on the sparse version of the `numjac` function. 3-D geometry uses only `'full'`, any other specification yields an error.
- `'fixed'` specifies a fixed-point iteration matrix where the Jacobian is approximated by the stiffness matrix. This is the 2-D geometry default.
- `'lumped'` specifies a "lumped" approximation as described in "Nonlinear Equations" on page 5-26. This approximation is based on the numerical differentiation of the coefficients.

Example: u = pdenonlin(model,c,a,f,'Jacobian','full')

Data Types: char

'U0' — Initial solution guess
0 (default) | scalar | string | vector

Initial solution guess, specified as a scalar, string, or vector. For details, see "Solve PDEs with Initial Conditions" on page 2-162.

- A scalar specifies a constant initial condition for either a scalar or PDE system.
- For scalar problems, write a string using the same syntax as "Specify Scalar PDE Coefficients in String Form" on page 2-68.
- For systems of N equations, write a character array with N rows, where each row has the syntax of "Specify Scalar PDE Coefficients in String Form" on page 2-68.
- For systems of N equations, and a mesh with `Np` nodes, give a column vector with $N*Np$ components. The nodes are either `model.Mesh.Nodes`, or the `p` data from `initmesh` or `meshToPet`. See "Mesh Data" on page 2-211.

  The first N_p elements contain the values of component 1, where the value of element `k` corresponds to node `p(k)`. The next N_p points contain the values of component 2, etc. It can be convenient to first 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(:)`.

Example: u = pdenonlin(model,c,a,f,'U0','x.^2-y.^2')

Data Types: `double` | `char`
Complex Number Support: Yes

`'Tol'` — Residual size at termination
1e-4 (default) | positive scalar

Residual size at termination, specified as a positive scalar. `pdenonlin` iterates until the residual size is less than `'Tol'`.

Example: `u = pdenonlin(model,c,a,f,'Tol',1e-6)`

Data Types: `double`

`'MaxIter'` — Maximum number of Gauss-Newton iterations
25 (default) | positive integer

Maximum number of Gauss-Newton iterations, specified as a positive integer.

Example: `u = pdenonlin(model,c,a,f,'MaxIter',12)`

Data Types: `double`

`'MinStep'` — Minimum damping of search direction
1/2^16 (default) | positive scalar

Minimum damping of search direction, specified as a positive scalar.

Example: `u = pdenonlin(model,c,a,f,'MinStep',1e-3)`

Data Types: `double`

`'Report'` — Print convergence information
`'off'` (default) | `'on'`

Print convergence information, specified as `'off'` or `'on'`.

Example: `u = pdenonlin(model,c,a,f,'Report','on')`

Data Types: `char`

`'Norm'` — Residual norm
Inf (default) | p value for L^p norm | `'energy'`

Residual norm, specified as the `p` value for L^p norm, or as the string `'energy'`. `p` can be any positive real value, `Inf`, or `-Inf`. The `p` norm of a vector `v` is `sum(abs(v)^p)^(1/p)`. See `norm`.

Example: `u = pdenonlin(model,c,a,f,'Norm',2)`

Data Types: `double` | `char`

Output Arguments

u — PDE solution
vector

PDE solution, returned as a vector.

- If the PDE is scalar, meaning only one equation, then `u` is a column vector representing the solution u at each node in the mesh. `u(i)` is the solution at the `ith` column of `model.Mesh.Nodes` or the `ith` column of `p`.

- If the PDE is a system of $N > 1$ equations, then `u` is a column vector with N*`Np` elements, where `Np` is the number of nodes in the mesh. The first `Np` elements of `u` represent the solution of equation 1, then next `Np` elements represent the solution of equation 2, etc.

To obtain the solution at an arbitrary point in the geometry, use `pdeInterpolant`.

To plot the solution, use `pdeplot` for 2-D geometry, or see "Plot 3-D Solutions and Their Gradients" on page 3-145.

`res` — Norm of Newton step residuals
scalar

Norm of Newton step residuals, returned as a scalar. For information about the algorithm, see "Nonlinear Equations" on page 5-26.

More About

Tips

- If the Newton iteration does not converge, `pdenonlin` displays the error message `Too many iterations` or `Stepsize too small`.

- If the initial guess produces matrices containing `NaN` or `Inf` elements, `pdenonlin` displays the error message `Unsuitable initial guess U0 (default: U0 = 0)`.

- If you have very small coefficients, or very small geometric dimensions, `pdenonlin` can fail to converge, or can converge to an incorrect solution. If so, you can sometimes obtain better results by scaling the coefficients or geometry dimensions to be of order one.

Algorithms

The `pdenonlin` algorithm is a Gauss-Newton iteration scheme applied to the finite element matrices. For details, see "Nonlinear Equations" on page 5-26.

See Also
`solvepde`

Introduced before R2006a

Using PDEResults Objects

PDE results object

Compatibility

createPDEResults NO LONGER RETURNS AN OBJECT OF TYPE PDEResults. Now PDE results are returned as `StationaryResults`, `TimeDependentResults`, and `EigenResults` objects.

Description

`PDEResults` is the superclass for the specific variants of results objects `StationaryResults`, `TimeDependentResults`, and `EigenResults`. These objects are returned by the `solvepde` and `solvepdeeig` functions of `PDEModel` and by the `createPDEResults` function. In R2015b, the `createPDEResults` function returned the generic form of a `PDEResults` object. Now it returns one of the specific variants.

If you are using R2015b, see Using PDEResults Objects in the R2015b documentation.

Introduced in R2015b

pdeplot

Plot solution in 2-D geometry

Compatibility

PLOTTING IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
pdeplot(model)
pdeplot(model,Name,Value)

pdeplot(p,e,t)
pdeplot(p,e,t,Name,Value)

h = pdeplot( ___ )
```

Description

pdeplot(model) plots the mesh specified in model.

pdeplot(model,Name,Value) plots data on the model mesh using one or more Name,Value pair arguments.

Specify at least one of the flowdata (vector field plot), xydata (colored surface plot), or zdata (3-D height plot) name-value pairs. Otherwise, pdeplot plots the mesh with no data. You can combine any number of plot types.

pdeplot(p,e,t) plots the mesh described by p,e, and t.

pdeplot(p,e,t,Name,Value) plots data on the (p,e,t) mesh using one or more Name,Value pair arguments.

Give at least one of the flowdata (vector field plot), xydata (colored surface plot), or zdata (3-D height plot) name-value pairs. Otherwise, pdeplot plots the mesh with no data. You can combine any number of plot types.

h = pdeplot(___) returns handles to the axis objects using any of the input arguments in the previous syntaxes.

Examples

Mesh Plot

Create a PDE model. Include the geometry of the built-in function lshapeg. Mesh the geometry and plot it.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
generateMesh(model);
pdeplot(model);
```
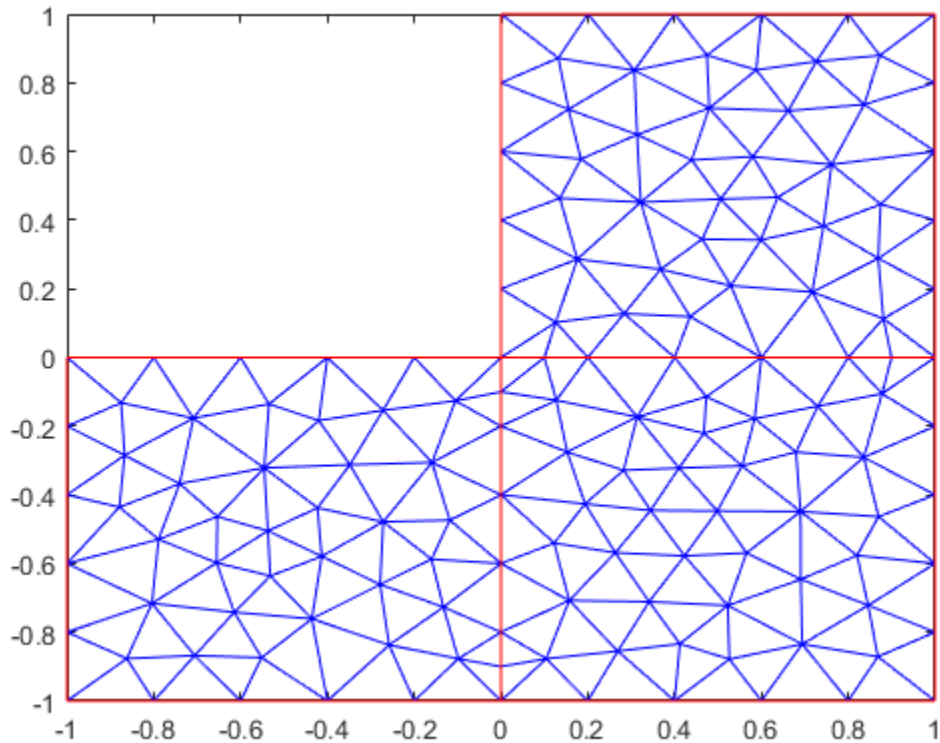
Solution Plots

Create colored 2-D and 3-D plots of a solution to a PDE model.

Create a PDE model. Include the geometry of the built-in function `lshapeg`. Mesh the geometry.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
generateMesh(model);
```

Set zero Dirichlet boundary conditions on all edges.

```
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',0);
```

Specify coefficients and solve the PDE.

```
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
results = solvepde(model)


results =

  StationaryResults with properties:

    NodalSolution: [150x1 double]
        XGradients: [150x1 double]
        YGradients: [150x1 double]
        ZGradients: []
              Mesh: [1x1 FEMesh]
```

Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

Plot the 2-D solution.

```
pdeplot(model,'xydata',u)
```

Plot the 3-D solution.

```
pdeplot(model,'xydata',u,'zdata',u)
```
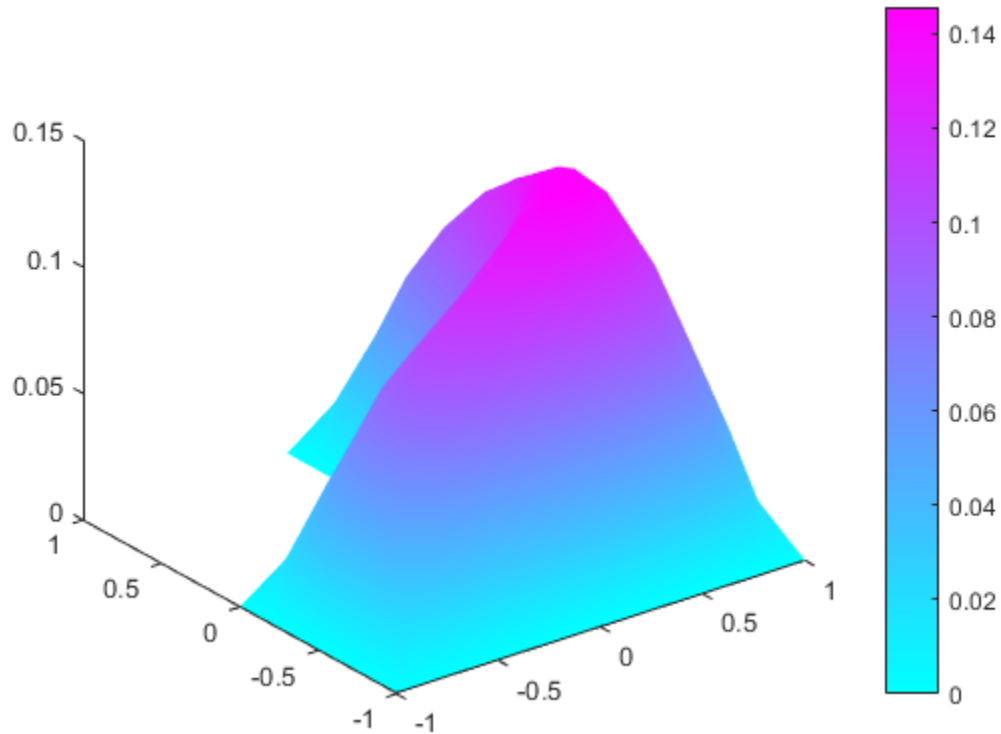
Solution Quiver Plot

Plot the gradient of a PDE solution as a quiver plot.

Create a PDE model. Include the geometry of the built-in function `lshapeg`. Mesh the geometry.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
generateMesh(model);
```

Set zero Dirichlet boundary conditions on all edges.

```
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',0);
```

Specify coefficients and solve the PDE.

```
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
results = solvepde(model)


results = 

  StationaryResults with properties:

    NodalSolution: [150x1 double]
        XGradients: [150x1 double]
        YGradients: [150x1 double]
        ZGradients: []
              Mesh: [1x1 FEMesh]
```

Access the gradient of the solution at the nodal locations.

```
ux = results.XGradients;
uy = results.YGradients;
```

Plot the gradient as a quiver plot.

```
pdeplot(model,'flowdata',[ux,uy])
```
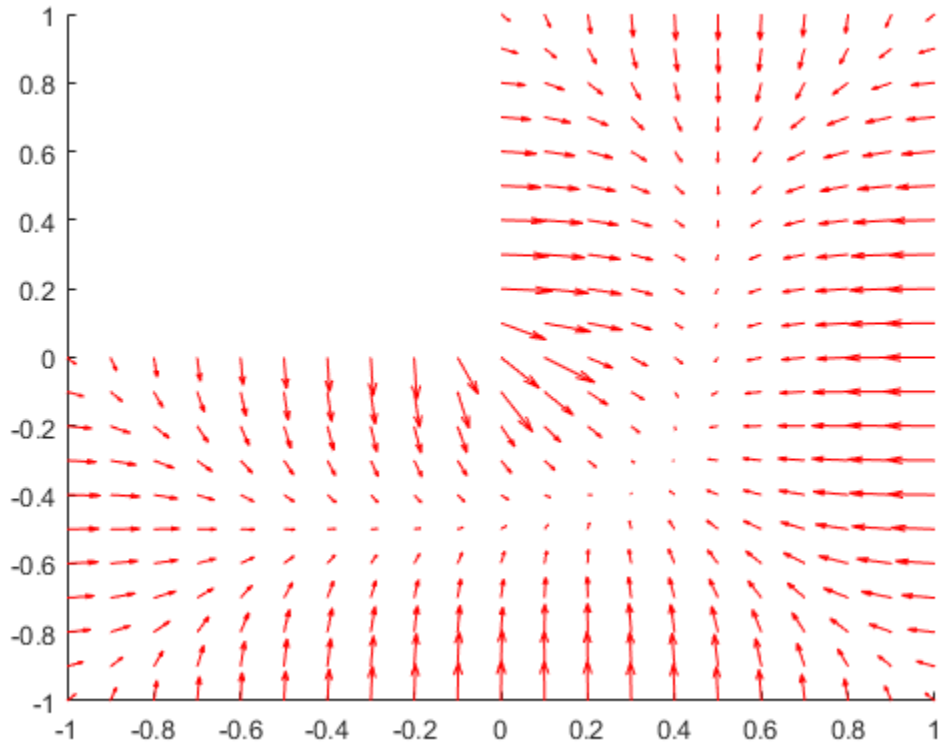
Composite Plot

Plot the solution of a PDE in 3-D with the `'jet'` coloring and a mesh, and include a quiver plot. Get handles to the axis objects.

Create a PDE model. Include the geometry of the built-in function `lshapeg`. Mesh the geometry.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
generateMesh(model);
```

Set zero Dirichlet boundary conditions on all edges.

```
applyBoundaryCondition(model,'Edge',1:model.Geometry.NumEdges,'u',0);
```

Specify coefficients and solve the PDE.

```
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
results = solvepde(model)


results =

  StationaryResults with properties:

    NodalSolution: [150x1 double]
       XGradients: [150x1 double]
       YGradients: [150x1 double]
       ZGradients: []
             Mesh: [1x1 FEMesh]
```
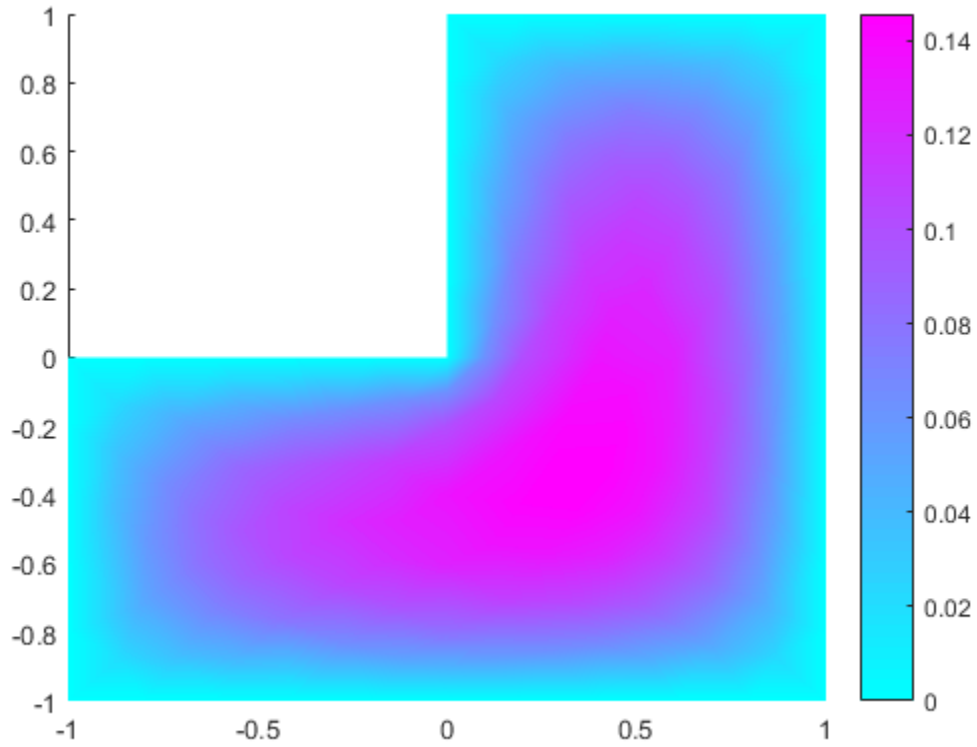
Access the solution and its gradient at the nodal locations.

```
u = results.NodalSolution;
ux = results.XGradients;
uy = results.YGradients;
```

Plot the solution in 3-D with the `'jet'` coloring and a mesh, and include the gradient as a quiver plot.

```
h = pdeplot(model,'xydata',u,'zdata',u,...
                  'flowdata',[ux,uy],...
                  'colormap','jet','mesh','on')


h =

  3x1 graphics array:

  Patch
  Quiver
  ColorBar
```
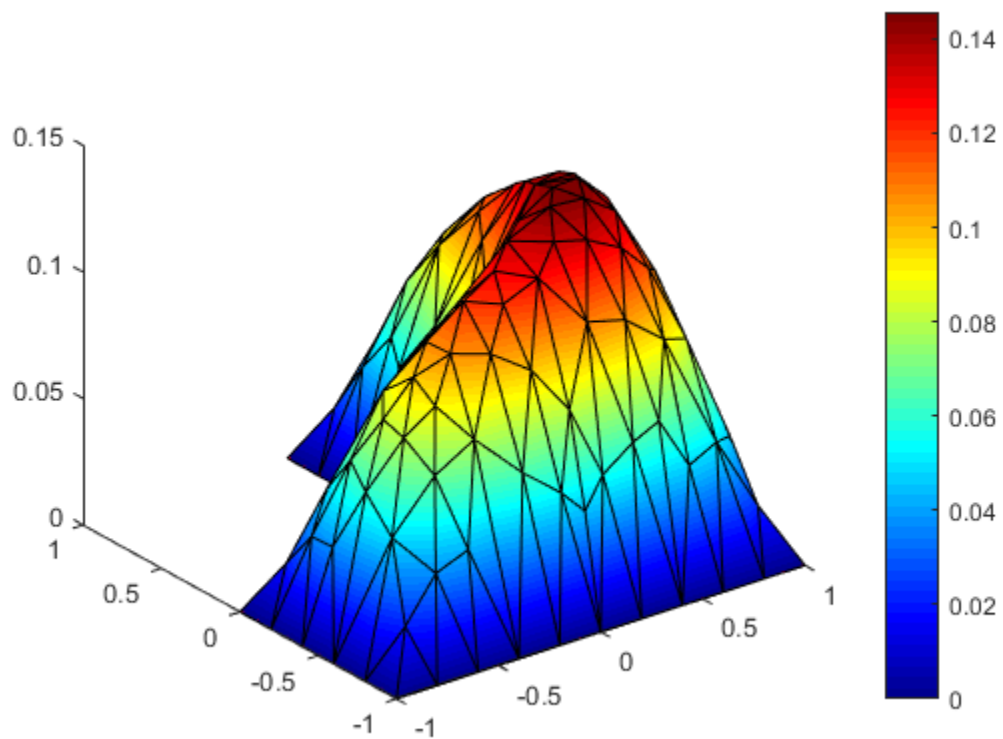
Look underneath to see the quiver plot.

```
view(20,-20)
```
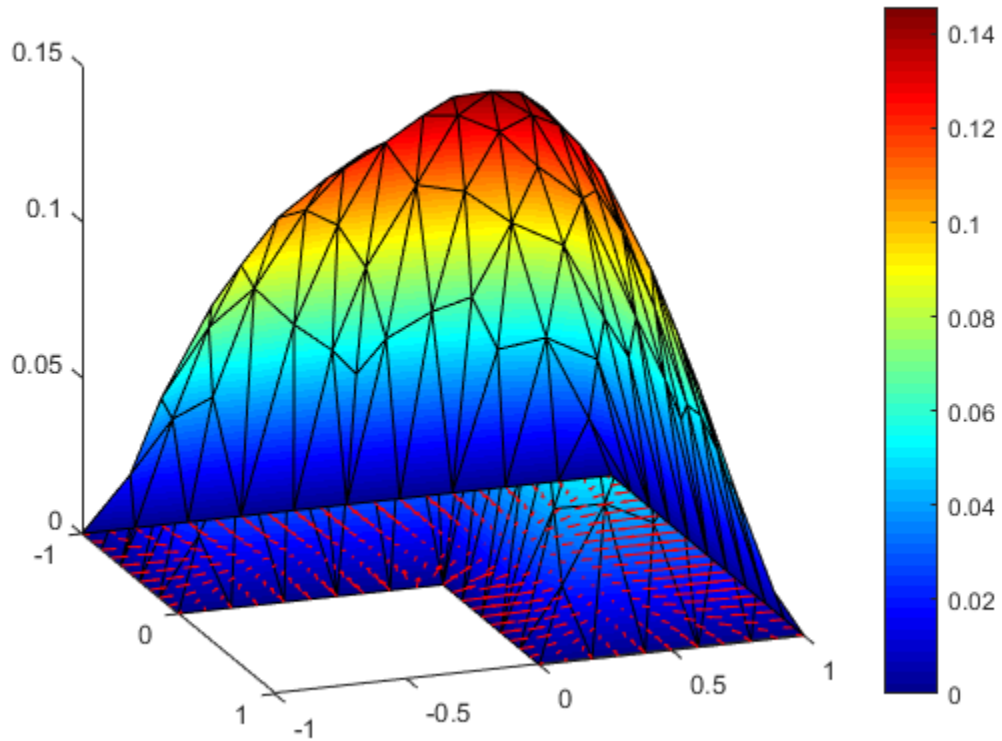
[p,e,t] Mesh and Solution Plots

Plot the `p,e,t` mesh.

Create the geometry, mesh, boundary conditions, PDE coefficients, and solution.

```
[p,e,t] = initmesh('lshapeg');
u = assempde('lshapeb',p,e,t,1,0,1);
```

Plot the mesh.

```
pdeplot(p,e,t)
```
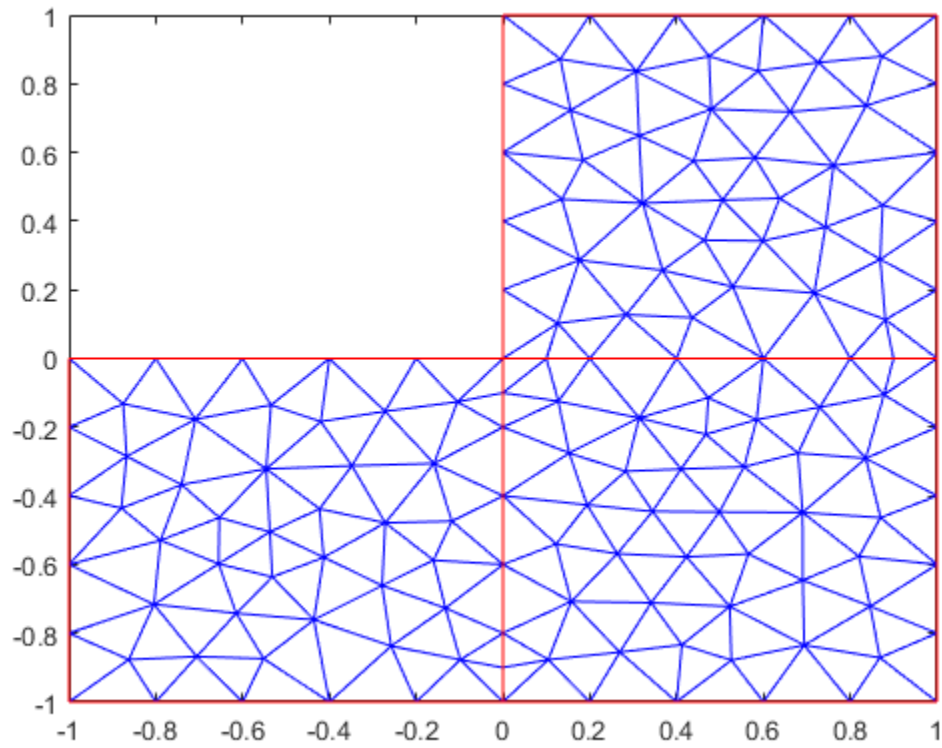
Plot the solution as a 2-D colored plot.

```
pdeplot(p,e,t,'xydata',u)
```

Plot the solution as a 3-D colored plot.

```
pdeplot(p,e,t,'xydata',u,'zdata',u)
```
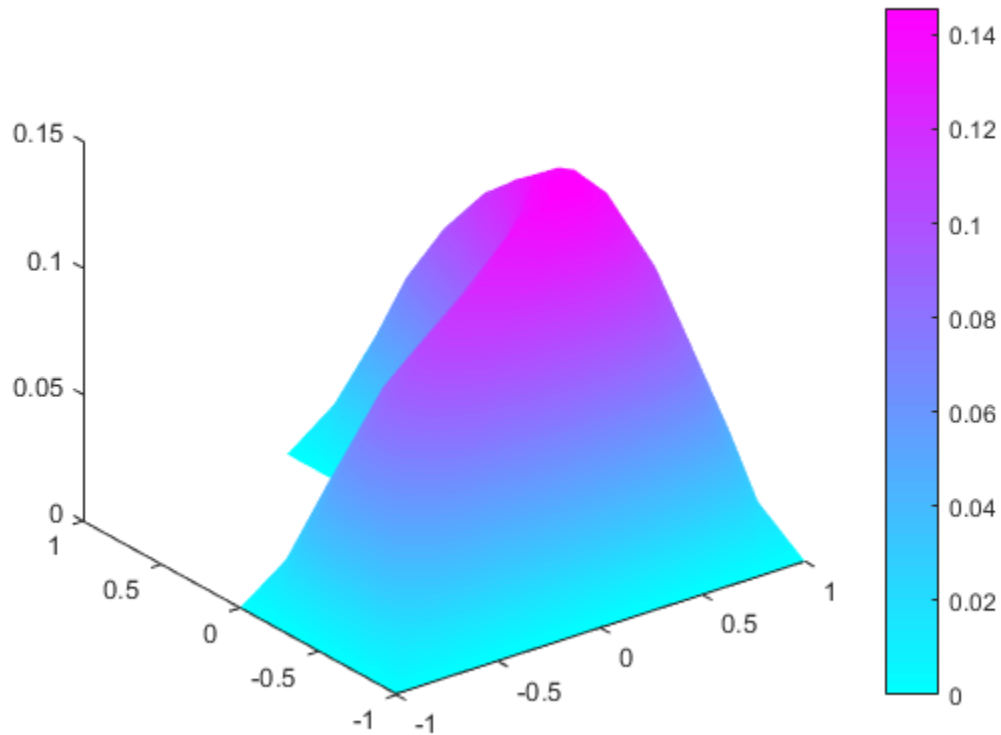
- "Plot 2-D Solutions and Their Gradients" on page 3-135
- "Deflection of a Piezoelectric Actuator" on page 3-19

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

p — Mesh points
matrix

2-by-Np matrix of points, where Np is the number of points in the mesh. For a description of the (p,e,t) matrices, see "Mesh Data" on page 2-211.

Typically, you use the p, e, and t data exported from the PDE app, or generated by initmesh or refinemesh.

Example: [p,e,t] = initmesh(gd)

Data Types: double

e — Mesh edges
matrix

7-by-Ne matrix of edges, where Ne is the number of edges in the mesh. For a description of the (p,e,t) matrices, see "Mesh Data" on page 2-211.

Typically, you use the p, e, and t data exported from the PDE app, or generated by initmesh or refinemesh.

Example: [p,e,t] = initmesh(gd)

Data Types: double

t — Mesh triangles
matrix

4-by-Nt matrix of triangles, where Nt is the number of triangles in the mesh. For a description of the (p,e,t) matrices, see "Mesh Data" on page 2-211.

Typically, you use the p, e, and t data exported from the PDE app, or generated by initmesh or refinemesh.

Example: [p,e,t] = initmesh(gd)

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. 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`.

Example: When you use a `PDEModel` object, `pdeplot(model,'xydata',u,'zdata',u)` sets surface plot coloring to the solution `u`, and sets the heights for a 3-D plot to `u`. Here `u` is a `NodalSolution` property of the PDE results returned by `solvepde` or `solvepdeeig`.

When you use a `[p,e,t]` representation, `pdeplot(p,e,t,'xydata',u,'zdata',u)` sets surface plot coloring to the solution `u`, and sets the heights for a 3-D plot to the solution `u`. Here `u` is a solution returned by a legacy solver, such as `assempde`.

Tip Give at least one of the `flowdata` (vector field plot), `xydata` (colored surface plot), or `zdata` (3-D height plot) name-value pairs. Otherwise, `pdeplot` plots the mesh with no data.

`'colorbar'` — Indicator to include color bar
`'on'` (default) | `'off'`

Indicator to include color bar, specified as the comma-separated pair consisting of `'colorbar'` and a string. `'on'` displays a bar giving the numeric values of colors in the plot. For details, see `colorbar`. `pdeplot` uses the colormap specified in the `colormap` name-value pair.

Example: `'colorbar','off'`

Data Types: `char`

`'colormap'` — Colormap
`'cool'` (default) | `colormap` string or matrix

Colormap, specified as the comma-separated pair consisting of `'colormap'` and a string representing a built-in colormap, or a colormap matrix. For details, see `colormap`.

`colormap` relates to the `xydata` name-value pair.

Example: `'colormap','jet'`

Data Types: `double | char`

`'contour'` — Indicator to plot level curves
`'off'` (default) | `'on'`

Indicator to plot level curves, specified as the comma-separated pair consisting of `'contour'` and a string. `'on'` plots level curves for the `xydata` data. Specify the levels with the `levels` name-value pair.

Example: `'contour','on'`

Data Types: `char`

`'flowdata'` — Data for quiver plot
matrix

Data for quiver plot, specified as the comma-separated pair consisting of `'flowdata'` and a matrix of vector data. `flowdata` can be M-by-2 or 2-by-M, where M is the number of mesh points `p` or the number of triangles `t`. `flowdata` contains the *x* and *y* values of the field at the mesh points or at the triangle centroids.

Typically, you set `flowdata` to the gradient of the solution. For example, when you use a `PDEModel` object, set `flowdata` as follows:

```
results = solvepde(model);
ux = results.XGradients;
uy = results.YGradients;
pdeplot(model,'flowdata',[ux,uy])
```

When you use a `[p,e,t]` representation, set `flowdata` as follows:

```
[ux,uy] = pdegrad(p,t,u); % Calculate gradient
pdeplot(p,e,t,'flowdata',[ux;uy])
```

In a 3-D plot, the quiver plot appears in the *z* = 0 plane.

`pdeplot` plots the real part of complex data.

Example: `'flowdata',[ux;uy]`

Data Types: `double`

`'flowstyle'` — Indicator to show quiver plot
`'arrow'` (default) | `'off'`

Indicator to show quiver plot, specified as the comma-separated pair consisting of `'flowstyle'` and a string. `'arrow'` displays the quiver plot specified by the `flowdata` name-value pair.

Example: `'flowstyle','off'`

Data Types: char

`'gridparam'` — Customized grid for `xygrid` name-value pair
`[tn;a2;a3]` from an earlier call to `tri2grid`

Customized grid for the `xygrid` name-value pair, specified as the comma-separated pair consisting of `'gridparam'` and the matrix `[tn;a2;a3]`. For example:

```
[~,tn,a2,a3] = tri2grid(p,t,u,x,y);
pdeplot(p,e,t,'xygrid','on','gridparam',[tn;a2;a3],'xydata',u)
```

For details on the grid data and its x and y arguments, see `tri2grid`. Note that `tri2grid` does not work with `PDEModel` objects.

Example: `'gridparam',[tn;a2;a3]`

Data Types: double

`'levels'` — Levels for contour plot
10 (default) | positive integer | vector of level values

Levels for contour plot, specified as the comma-separated pair consisting of `'levels'` and a positive integer or a vector.

- Positive integer — Plot `levels` equally spaced contours.
- Vector — Plot contours at the values in `levels`.

To obtain a contour plot, set the `contour` name-value pair to `'on'`.

Example: `'levels',16`

Data Types: double

`'mesh'` — Indicator to show mesh
`'off'` (default) | `'on'`

Indicator to show mesh, specified as the comma-separated pair consisting of `'mesh'` and a string. `'on'` shows the mesh in the plot.

Example: `'mesh','on'`

Data Types: char

`'title'` — Title of plot
string

Title of plot, specified as the comma-separated pair consisting of `'title'` and a string.

Example: `'title','Solution Plot'`

Data Types: `char`

`'xydata'` — Colored surface plot data
vector

Colored surface plot data, specified as the comma-separated pair consisting of `'xydata'` and a vector. If you use a `[p,e,t]` representation, give data for points in a vector of length `size(p,2)`, or data for triangles in a vector of length `size(t,2)`.

Typically, you set `xydata` to `u`, the solution. `pdeplot` uses `xydata` for coloring both 2-D and 3-D plots.

`pdeplot` uses the colormap specified in the `colormap` name-value pair, using the style specified in the `xystyle` name-value pair.

When the `contour` name-value pair is `'on'`, `pdeplot` also plots level curves of `xydata`.

`pdeplot` plots the real part of complex data.

To plot the `kth` component of a solution to a PDE system, extract the relevant part of the solution. For example, when you use a `PDEModel` object:

```
results = solvepde(model);
u = results.NodalSolution; % each column of u has one component of u
pdeplot(model,'xydata',u(:,k)) % data for column k
```

When you use a `[p,e,t]` representation:

```
np = size(p,2); % number of node points
uk = reshape(u,np,[]); % each uk column has one component of u
pdeplot(p,e,t,'xydata',uk(:,k)) % data for column k
```

Example: `'xydata',u`

Data Types: `double`

`'xygrid'` — Indicator to convert to *x-y* grid before plotting
`'off'` (default) | `'on'`

Indicator to convert mesh data to *x-y* grid before plotting, specified as the comma-separated pair consisting of `'xygrid'` and a string.

Note: This conversion can change the geometry, and can lessen the quality of the plot.

By default, the grid has about `sqrt(size(t,2))` elements in each direction. Exercise more control over the *x-y* grid by generating it with the `tri2grid` function, and passing it in with the `gridparam` name-value pair.

Example: `'xygrid','on'`

Data Types: `char`

`'xystyle'` — Coloring choice
`'interp'` (default) | `'off'` | `'flat'`

Coloring choice, specified as the comma-separated pair consisting of `'xystyle'` and a string.

- `'off'` — No shading, shows the mesh only.
- `'flat'` — Each triangle in the mesh has a uniform color.
- `'interp'` — Plot coloring is smoothly interpolated.

The coloring choice relates to the `xydata` name-value pair.

Example: `'xystyle','flat'`

Data Types: `char`

`'zdata'` — Data for 3-D plot heights
matrix

Data for 3-D plot heights, specified as the comma-separated pair consisting of `'zdata'` and a vector. If you use a `[p,e,t]` representation, give data for points in a vector of length `size(p,2)`, or data for triangles in a vector of length `size(t,2)`.

Typically, you set `zdata` to `u`, the solution. The `xydata` name-value pair sets the coloring of the 3-D plot. The `zstyle` name-value pair specifies whether the plot is continuous or discontinuous.

`pdeplot` plots the real part of complex data.

To plot the `kth` component of a solution to a PDE system, extract the relevant part of the solution. For example, when you use a `PDEModel` object:

```
results = solvepde(model);
```

```
u = results.NodalSolution; % each column of u has one component of u
pdeplot(model,'xydata',u(:,k),'zdata',u(:,k)) % data for column k
```

When you use a [p,e,t] representation:

```
np = size(p,2); % number of node points
uk = reshape(u,np,[]); % each uk column has one component of u
pdeplot(p,e,t,'xydata',uk(:,k),'zdata',uk(:,k)) % data for column k
```

Example: 'zdata',u

Data Types: double

'zstyle' — 3-D plot style
'continuous' (default) | 'off' | 'discontinuous'

3-D plot style, specified as the comma-separated pair consisting of 'zstyle' and a string.

- 'off' — No 3-D plot.
- 'discontinuous' — Each triangle in the mesh has a uniform height in a 3-D plot.
- 'continuous' — 3-D surface plot is continuous.

zstyle relates to the zdata name-value pair.

Example: 'zstyle','discontinuous'

Data Types: char

Output Arguments

h — Handles to axis objects in the plot
vector of handles

Handles to axis objects in the plot, returned as a vector.

More About

Quiver Plot

Plot of vector field

Plot of a vector field, also called a flow plot. Arrows show the direction of the field, with the lengths of the arrows showing the relative sizes of the field strength. For details on quiver plots, see quiver.

- "Mesh Data" on page 2-211
- "Solve Problems Using PDEModel Objects" on page 2-14

See Also
PDEModel | pdeplot3D

Introduced before R2006a

pdeplot3D

Plot 3-D solution or surface mesh

Compatibility

PLOTTING IS THE SAME FOR BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS.

Syntax

```
pdeplot3D(model,'colormapdata',u)
pdeplot3D(model)
pdeplot3D(model,Name,Value)
h = pdeplot3D( ___ )
```

Description

pdeplot3D(model,'colormapdata',u) plots the data u as colors, using the 'jet' colormap on the surface of the geometry in model.

pdeplot3D(model) plots the surface mesh. This plot is the same plot as the one produced by pdemesh.

pdeplot3D(model,Name,Value) plots the surface mesh, modified with the Name,Value pair.

h = pdeplot3D(___) returns handles to the graphics, using any of the previous syntaxes.

Examples

Solution Plot on Surface

Plot a PDE solution on the geometry surface.

Create a PDE model and import a 3-D geometry file. Specify boundary conditions and coefficients. Mesh the geometry and solve the problem.

```
model = createpde;
importGeometry(model,'Block.stl');
applyBoundaryCondition(model,'face',[1:4],'u',0);
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',2);
generateMesh(model);
results = solvepde(model)

results =

  StationaryResults with properties:

    NodalSolution: [28811x1 double]
       XGradients: [28811x1 double]
       YGradients: [28811x1 double]
       ZGradients: [28811x1 double]
             Mesh: [1x1 FEMesh]
```

Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

Plot the solution u on the geometry surface.

```
pdeplot3D(model,'colormapdata',u)
```
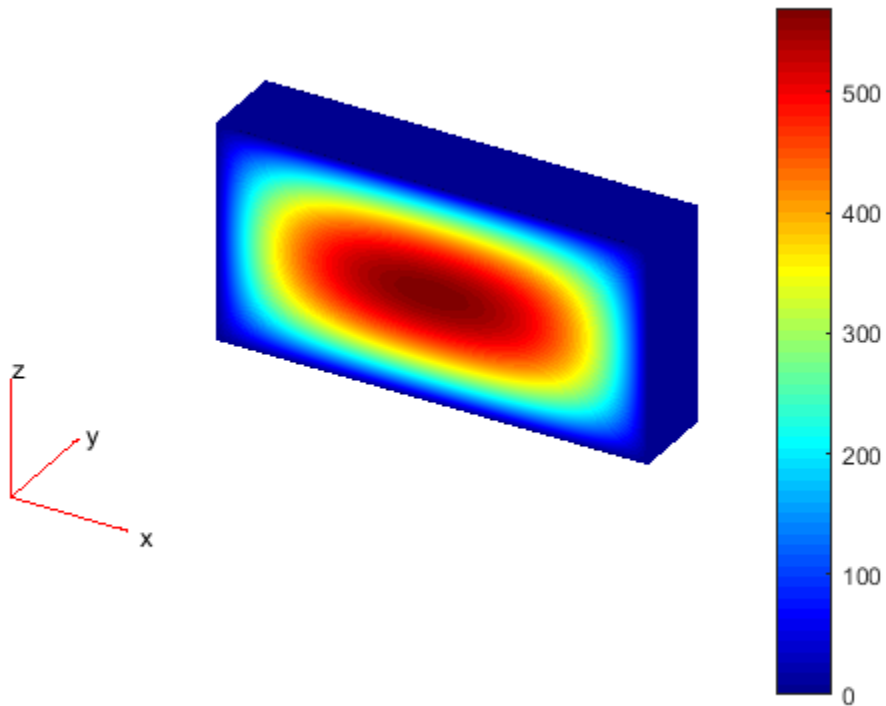
Plot Mesh Nodes with Labels

View the node labels on the surface of a simple mesh.

```
model = createpde;
importGeometry(model,'Tetrahedron.stl');
generateMesh(model,'Hmax',20,'GeometricOrder','linear');
pdeplot3D(model,'NodeLabels','on');
view(101,12)
```
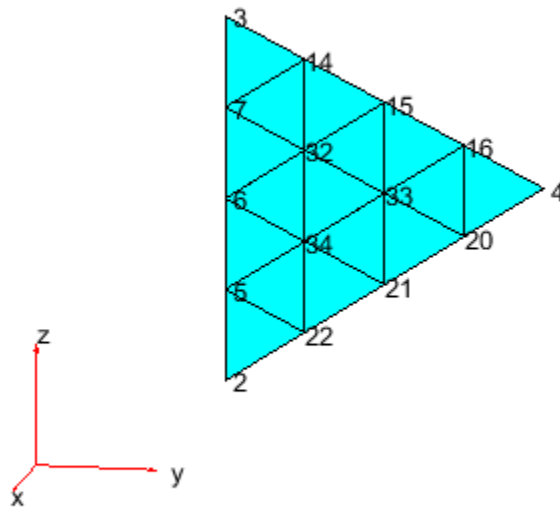
- "Plot 3-D Solutions and Their Gradients" on page 3-145

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

u — Function to plot
np*N column vector

Function to plot, specified as an `np*N` column vector. `np` is the number of points in the mesh, and `N` is the number of equations in the PDE. Typically, `u` is the solution returned by a solver function, such as `assempde` or `hyperbolic`.

Example: `u = assempde(model,c,a,f)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Example: `pdeplot3D(model,'NodeLabels','on')`

`'NodeLabels'` — Node labels
`'off'` (default) | `'on'`

Node labels, specified as `'off'` or `'on'`.

Example: `pdeplot3D(model,'NodeLabels','on')`

Data Types: `char`

`'FaceAlpha'` — Surface transparency
1 (default) | real number from 0 through 1

Surface transparency, specified as a real number from 0 through 1. The default value 1 indicates no transparency. The value 0 indicates complete transparency.

Example: `pdeplot3D(model,'FaceAlpha',0.5)`

Data Types: `double`

Output Arguments

h — Handles to graphics objects
vector of handles

Handles to graphics objects, returned as a vector of handles.

More About

- "Solve Problems Using PDEModel Objects" on page 2-14

See Also

PDEModel | pdeplot

Introduced in R2015a

pdepoly

Draw polygon

pdepoly opens the PDE app and draws a polygon. If, instead, you want to draw polygons in a MATLAB figure, use the `plot` function such as `x = [-1,-0.5,-0.5,0,1.5,-0.5,-1];y = [-1,-1,-0.5,0,0.5,0.9,-1];plot(x,y,'.-').`

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 PDE app 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 the PDE app is updated to include the polygon. You can export the Geometry Description matrix from the PDE app 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

Interpolate from triangle midpoint data to node data

Compatibility

pdeprtni IS NOT RECOMMENDED. Use `interpolateSolution` and `evaluateGradient` instead.

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
`solvepde` | `interpolateSolution`

pderect

Draw rectangle

`pderect` opens the PDE app and draws a rectangle. If, instead, you want to draw rectangles in a MATLAB figure, use the `rectangle` function such as `rectangle('Position',[1,2,5,6])`.

Syntax

```
pderect(xy)
```

```
pderect(xy,label)
```

Description

`pderect(xy)` draws a rectangle with corner coordinates defined by `xy = [xmin xmax ymin ymax]`. If the PDE app 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 the PDE app is updated to include the rectangle. You can export the Geometry Description matrix from the PDE app 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 the PDE app 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

pdesdppdesdepdesdt

Indices of points/edges/triangles in set of subdomains

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be
compatible with the legacy workflow. For the corresponding step in the recommended
workflow, see `generateMesh`.

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.

pdesmech

Calculate structural mechanics tensor functions

Compatibility

`pdesmech` IS NOT RECOMMENDED. Use the PDE app instead.

Syntax

```
ux = pdesmech(p,t,c,u,'PropertyName',PropertyValue,...)
```

Description

`ux = pdesmech(p,t,c,u,'PropertyName',PropertyValue,...)` 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 PDE app, 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'` | `{'vonmises'}`	Tensor expression
`application`	`{'ps'}` | `'pn'`	Plane stress | plane strain
`nu`	Scalar | vector | string expression | `{0.3}`	Poisson's ratio. Applies to calculating von Mises (`'vonmises'`)

Property Name	Property Value/Default	Description
		effective stress in plane strain mode (`'pn'`). Specify a scalar if the value is constant over the entire geometry. Specify a vector as a row vector whose length is equal to the number of elements. Specify a string expression in coefficient form: "Specify Scalar PDE Coefficients in String Form" on page 2-68.

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)

- `'vonmises'`, 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 3-7, 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','vonmises',...
   'application','pn','nu',0.3);
```

pdesurf

Shorthand command for surface plot

Compatibility

THIS PAGE DESCRIBES THE LEGACY APPROACH. Use it when you work with legacy code and do not plan to convert it to use the recommended approach. Otherwise, use `pdeplot`.

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

Open PDE app

Syntax

```
pdetool
```

Description

`pdetool` starts the PDE app. You should not call `pdetool` with arguments.

The PDE app 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 PDE app.

The PDE app 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. the PDE app 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 PDE app on the solid object itself. The

names refer to the object in the set formula. More specifically, in the set formula, the name refers to the set of points inside the object. The resulting geometrical model is the set of points for which the set formula evaluates to true. (For a description of the syntax of the set formula, see `decsg`.) By default, the resulting geometrical model is the union of all objects.

A "snap-to-grid" function is available. This means that objects align to the grid. The grid can be turned on and off, and the scaling and the grid spacing can be changed.

In boundary mode, you can specify the boundary conditions. You can have different types of boundary conditions on different boundaries. In this mode, the original shapes of the solid building objects constitute borders between subdomains of the model. Such borders can be eliminated in this mode. The outer boundaries are color coded to indicate the type of boundary conditions. A red outer boundary corresponds to Dirichlet boundary conditions, blue to generalized Neumann boundary conditions, and green to mixed boundary conditions. You can return to the boundary condition display by clicking the $\partial\Omega$ button or by selecting **Boundary Mode** from the **Boundary** menu.

In PDE mode, you can specify the type of PDE problem, and the coefficients c, a, f and d. You can specify the coefficients for each subdomain independently. This makes it easy to specify, e.g., various material properties in one PDE model. The PDE to be solved can be specified by clicking the **PDE** button or by selecting **PDE Specification** from the **PDE** menu. This brings up a dialog box.

In mesh mode, you can control the automated mesh generation and plot the mesh. An initial mesh can be generated by clicking the Δ button or by selecting **Initialize Mesh** from the **Mesh** menu. Choose the meshing algorithm using the **Mesh > Parameters > Mesher version** menu. The `'R2013a'` algorithm runs faster, and can triangulate more geometries than the `'preR2013a'` algorithm. 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 PDE app 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 the PDE app. 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 "Specify Boundary Conditions in the PDE App" on page 4-5.)

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 "Specify Coefficients in the PDE App" on page 4-8.)

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 the PDE app. If the PDE app has already been started, the current model is cleared. The following commands set up the scaling and tick marks of the axis of the PDE app 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, and so on.

```
% 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.

6-355

See Also

solvepde | solvepdeeig | pdeplot

pdetrg

Triangle geometry data

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `generateMesh`.

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`.

pdetriq

Triangle quality measure

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `generateMesh`.

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.

poiasma

Boundary point matrix contributions for fast solvers of Poisson's equation

Compatibility

poiasma IS NOT RECOMMENDED. To solve Poisson's equations, use solvepde. For details, see "Solve Problems Using PDEModel Objects" on page 2-14.

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.

poicalc

Fast solver for Poisson's equation on rectangular grid

Compatibility

poicalc IS NOT RECOMMENDED. To solve Poisson's equations, use `solvepde`. For details, see "Solve Problems Using PDEModel Objects" on page 2-14.

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.

poiindex

Indices of points in canonical ordering for rectangular grid

Compatibility

poiindex IS NOT RECOMMENDED. To solve Poisson's equations, use `solvepde`. For details, see "Solve Problems Using PDEModel Objects" on page 2-14.

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]`.

poimesh

Make regular mesh on rectangular geometry

Compatibility

poimesh IS NOT RECOMMENDED. To solve Poisson's equations, use solvepde. For details, see "Solve Problems Using PDEModel Objects" on page 2-14.

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.

poisolv

Fast solution of Poisson's equation on rectangular grid

Compatibility

poisolv IS NOT RECOMMENDED. To solve Poisson's equations, use `solvepde`. For details, see "Solve Problems Using PDEModel Objects" on page 2-14.

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.

refinemesh

Refine triangular mesh

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow. For the corresponding step in the recommended workflow, see `generateMesh`.

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 2-211.

`[p1,e1,t1,u1] = refinemesh(g,p,e,t,u)` refines the mesh and also extends the function `u` to the new mesh by linear interpolation. The number of rows in `u` should correspond to the number of columns in `p`, and `u1` has as many rows as there are points in `p1`. Each column of `u` is interpolated separately.

An extra input argument `it` is interpreted as a list of subdomains to refine, if it is a row vector, or a list of triangles to refine, if it is a column vector.

The default refinement method is regular refinement, where all of the specified triangles are divided into four triangles of the same shape. Longest edge refinement, where the longest edge of each specified triangle is bisected, can be demanded by giving `longest` as a final parameter. Using `regular` as a final parameter results in regular refinement. Some triangles outside of the specified set may also be refined to preserve the triangulation and its quality.

Examples

Refine the mesh of the L-shaped membrane several times. Plot the mesh for the geometry of the L-shaped membrane.

```
[p,e,t] = initmesh('lshapeg','hmax',inf);
subplot(2,2,1), pdemesh(p,e,t)
[p,e,t] = refinemesh('lshapeg',p,e,t);
subplot(2,2,2), pdemesh(p,e,t)
[p,e,t] = refinemesh('lshapeg',p,e,t);
subplot(2,2,3), pdemesh(p,e,t)
[p,e,t] = refinemesh('lshapeg',p,e,t);
subplot(2,2,4), pdemesh(p,e,t)
subplot
```

More About

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.

- "Mesh Data" on page 2-211

See Also
initmesh | pdeent | pdesdt

setInitialConditions

Give initial conditions or initial solution

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW. For the corresponding step in the legacy workflow, see "Solve PDEs with Initial Conditions" on page 2-162.

Syntax

```
setInitialConditions(model,u0)
setInitialConditions(model,u0,ut0)
setInitialConditions( ___ ,regiontype,regionid)
ic = setInitialConditions( ___ )
```

Description

`setInitialConditions(model,u0)` sets initial conditions in `model`. Use this syntax for stationary nonlinear problems or time-dependent problems where the time derivative is first order, such as a parabolic problem.

Note: Include geometry in `model` before using `setInitialConditions`.

`setInitialConditions(model,u0,ut0)` sets initial conditions in `model` for a time-dependent problem. Use this syntax for time-dependent problems where a time derivative is second order, such as a hyperbolic problem.

`setInitialConditions( ___ ,regiontype,regionid)`, for any previous syntax, sets initial conditions on a geometry region.

`ic = setInitialConditions( ___ )`, for any previous syntax, returns a handle to the initial conditions object.

Examples

Set Constant Initial Conditions

Create a PDE model, import geometry, and set the initial condition to 50 on the entire geometry.

```
model = createpde();
importGeometry(model,'BracketWithHole.stl');
setInitialConditions(model,50);
```

Set Constant Initial Conditions for a System

Set different initial conditions for each component of a system of PDEs.

Create a PDE model for a system with five components. Import the 'Block.stl' geometry.

```
model = createpde(5);
importGeometry(model,'Block.stl');
```

Set the initial conditions for each component to twice the component number.

```
u0 = [2:2:10]';
setInitialConditions(model,u0)

ans =

  GeometricInitialConditions with properties:

           RegionType: 'cell'
             RegionID: 1
         InitialValue: [5x1 double]
    InitialDerivative: []
```
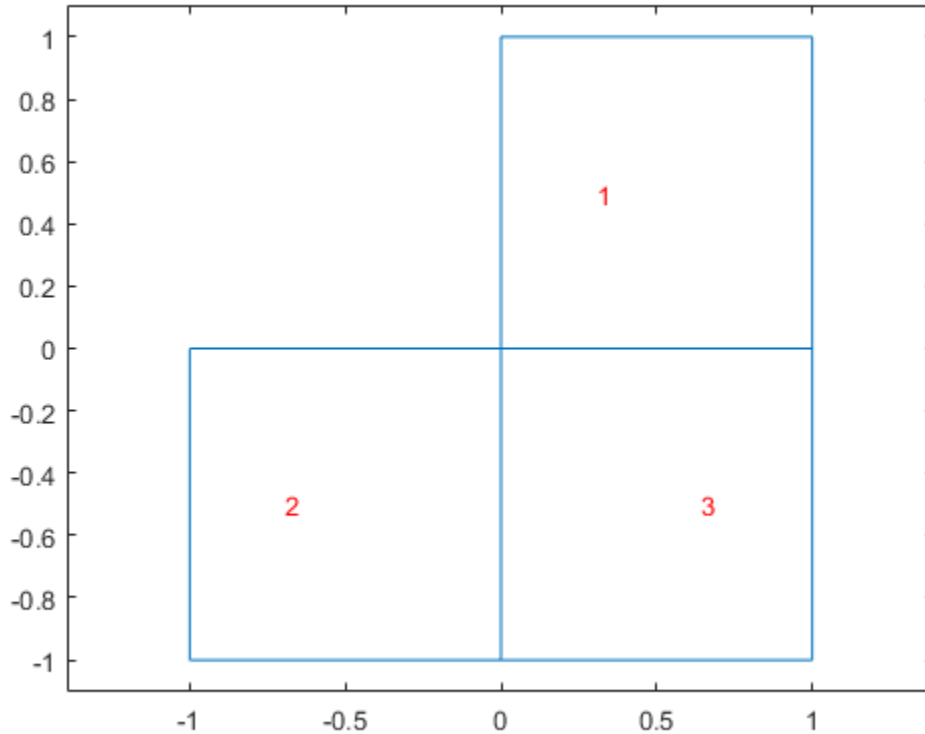
Set Different Initial Conditions on Subdomains

Set different initial conditions on each portion of the L-shaped membrane geometry.

Create a model, set the geometry function, and view the subdomain labels.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'SubdomainLabels','on')
axis equal
ylim([-1.1,1.1])
```

Set subdomain 1 to initial value -1, subdomain 2 to initial value 1, and subdomain 3 to initial value 5.

```
setInitialConditions(model,-1);
setInitialConditions(model,1,'face',2);
setInitialConditions(model,5,'face',3);
```

The initial setting applies to the entire geometry. The subsequent settings override the initial settings for regions 2 and 3.

Set Nonconstant Initial Conditions that are Functions of Position

Set initial conditions for the L-shaped membrane geometry to be $x^2 + y^2$, except in the lower-left square where it is $x^2 - y^4$.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'SubdomainLabels','on')
axis equal
ylim([-1.1,1.1])
```
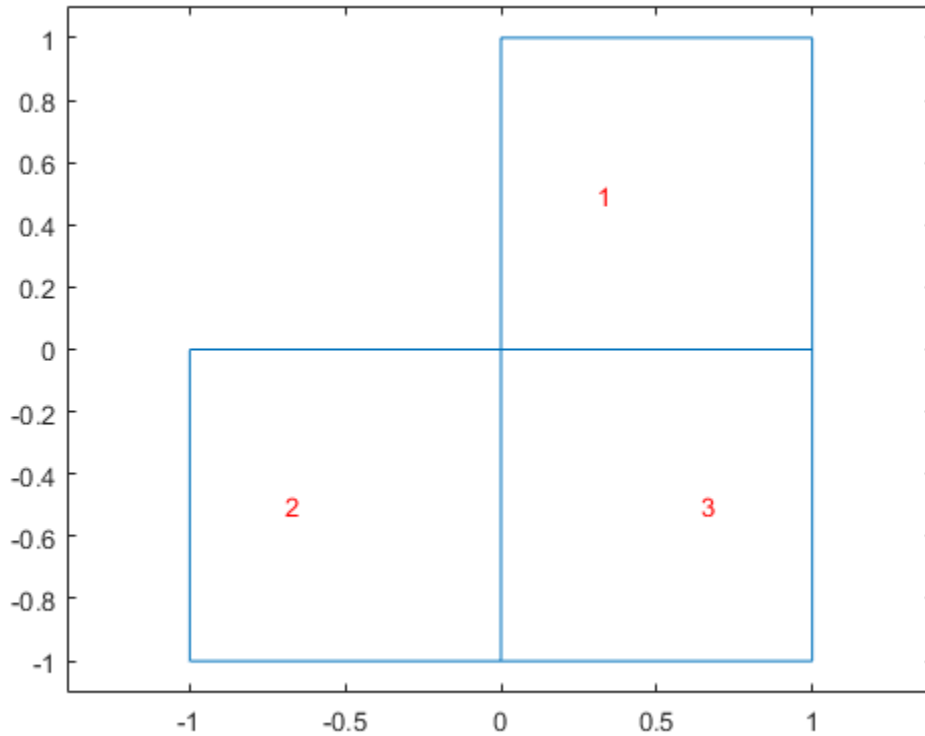

Set the initial conditions to $x^2 + y^2$.

```
initfun = @(locations)locations.x.^2 + locations.y.^2;
setInitialConditions(model,initfun);
```

Set the initial conditions on region 2 to $x^2 - y^4$. This setting overrides the first setting because you apply it after the first setting.

```
initfun2 = @(locations)locations.x.^2 - locations.y.^4;
setInitialConditions(model,initfun2,'face',2);
```

Set Initial Conditions for Hyperbolic Equation

Hyperbolic equations have nonzero m coefficient, so you must set both the u0 and ut0 arguments.

Import the 'Block.stl' to a PDE model with $N = 3$ components.

```
model = createpde(3);
importGeometry(model,'Block.stl');
```

Set the initial condition to be 0 for all components. Set the initial gradient to be the following:

$$ut0 = \begin{bmatrix} 4 + \dfrac{x}{x^2 + y^2 + z^2} \\ 5 - \tanh(z) \\ 10\dfrac{y}{x^2 + y^2 + z^2} \end{bmatrix}.$$

To create this initial gradient, write a function file, and ensure that the function is on your MATLAB path.

```
function ut0 = ut0fun(locations)

M = length(locations.x);
utinit = zeros(3,M);
denom = locations.x.^2+locations.y.^2+locations.z.^2;
ut0(1,:) = 4 + locations.x./denom;
ut0(2,:) = 5 - tanh(locations.z);
ut0(3,:) = 10*locations.y./denom;
```

Set the initial conditions.

```
setInitialConditions(model,0,@ut0fun)

ans =
```

```
GeometricInitialConditions with properties:

        RegionType: 'cell'
          RegionID: 1
      InitialValue: O
  InitialDerivative: @utOfun
```

- "Set Initial Conditions" on page 2-155

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a `PDEModel` object.

Example: model = `createpde(1)`

u0 — Initial condition
scalar | column vector of length N | function handle

Initial conditions, specified as a scalar, a column vector of length N, or a function handle. N is the size of the system of PDEs. See "Systems of PDEs" on page 2-65.

- Scalar — Represents a constant initial value for all solution components throughout the domain.
- Column vector — Represents a constant initial value for each of the N solution components throughout the domain.
- Function handle — Represents the initial conditions as a function of position. The function must be of the form

  ```
  u0 = initfun(locations)
  ```

  Solvers pass `locations` as a structure with fields `locations.x`, `locations.y`, and, for 3-D problems, `locations.z`. `initfun` must return a matrix `u0` of size N-by-M, where `M = length(locations.x)`.

Example: `setInitialConditions(model,10)`

Data Types: `double` | function_handle
Complex Number Support: Yes

ut0 — Initial condition for time derivative
scalar | column vector of length *N* | function handle

Initial condition for time derivative, specified as a scalar, a column vector of length *N*, or a function handle. *N* is the size of the system of PDEs. See "Systems of PDEs" on page 2-65. You must specify `ut0` when there is a nonzero second-order time derivative coefficient m.

- Scalar — Represents a constant initial value for all solution components throughout the domain.
- Column vector — Represents a constant initial value for each of the *N* solution components throughout the domain.
- Function handle — Represents the initial conditions as a function of position. The function must be of the form

  ```
  u0 = initfun(locations)
  ```

  Solvers pass `locations` as a structure with fields `locations.x`, `locations.y`, and, for 3-D problems, `locations.z`. `initfun` must return a matrix `u0` of size *N*-by-M, where `M = length(locations.x)`.

Example: `setInitialConditions(model,10,@initfun)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

regiontype — Geometric region type
`'face'` | `'edge'` | `'cell'`

Geometric region type, specified as `'face'`, `'edge'`, or `'cell'`.

When there are multiple initial condition assignments, solvers use the following precedence rules for determining the initial condition.

- If there are multiple assignments to a geometric region, solvers use the last applied setting.
- If there are assignments to a geometric region and the boundaries of that region, solvers use the lowest-dimensional boundary assignment. In other words, solvers give `'edge'` precedence over `'face'`, and `'face'` precedence over a 3-D region, even if you specify `'face'` conditions after `'edge'` conditions.

Example: `setInitialConditions(model,10,'face',1:4)`

Data Types: char

`regionid` — Geometric region number
vector of positive integers

Geometric region number, specified as a vector of positive integers. Find the region numbers using pdegplot, as shown in "Create Geometry and Remove Subdomains" on page 2-22 or "Create and View 3-D Geometry" on page 2-47.

Example: setInitialConditions(model,10,'face',1:4)

Data Types: double

Output Arguments

`ic` — Handle to initial condition
object

Handle to initial condition, returned as an object. ic associates the initial condition with the geometric region.

More About

Tips

- Prefer specifying coefficients before setting initial conditions. When you specify the entries in this order, the model has the correct TimeDependent property setting before you set the initial conditions.

- "Solve Problems Using PDEModel Objects" on page 2-14

See Also
findInitialConditions | pdegplot | PDEModel

Introduced in R2016a

solvepde

Solve PDE specified in a PDEModel

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW. For the corresponding step in the legacy workflow, see `assempde`, `parabolic`, `hyperbolic`, or `pdenonlin`.

Syntax

```
result = solvepde(model)
result = solvepde(model,tlist)
```

Description

`result = solvepde(model)` returns the solution to the stationary PDE represented in `model`. A stationary PDE has the property `model.IsTimeDependent = false`. That is, the time-derivative coefficients `m` and `d` in `model.EquationCoefficients` must be `0`.

`result = solvepde(model,tlist)` returns the solution to the time-dependent PDE represented in `model` at the times `tlist`. At least one time-derivative coefficient `m` or `d` in `model.EquationCoefficients` must be nonzero.
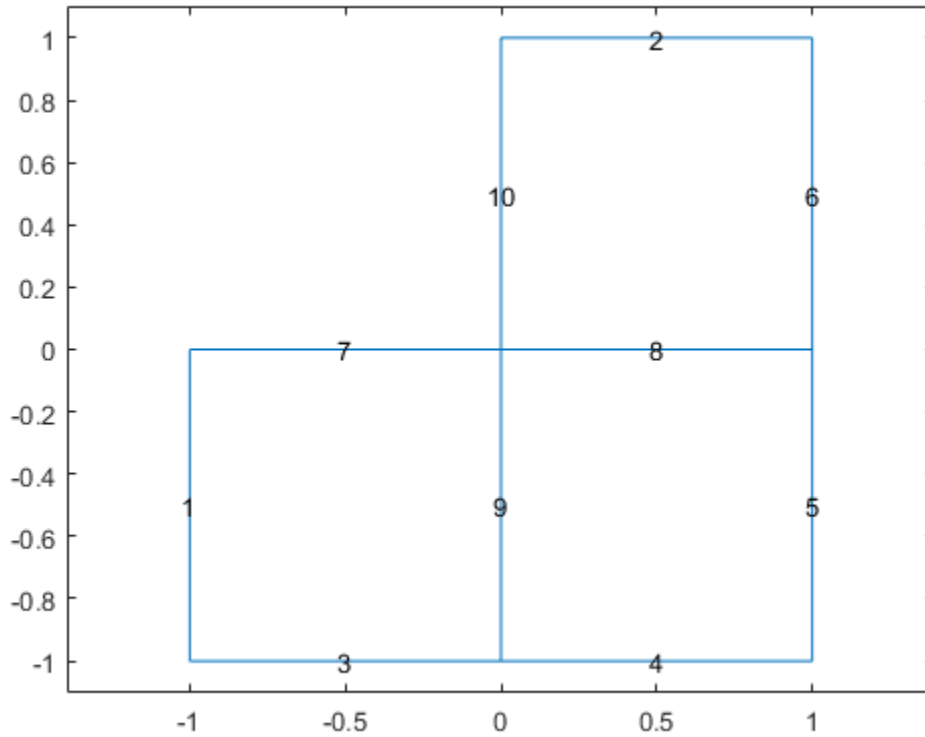
Examples

Solve a Stationary Problem: Poisson's Equation for the L-shaped Membrane

Create a PDE model, and include the geometry of the L-shaped membrane.

```
model=createpde();
geometryFromEdges(model,@lshapeg);
```

View the geometry with edge labels.

```
pdegplot(model,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```


Set zero Dirichlet conditions on all edges.

```
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
```

Poisson's equation is

$$-\nabla \cdot \nabla u = 1.$$

Toolbox solvers address equations of the form

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

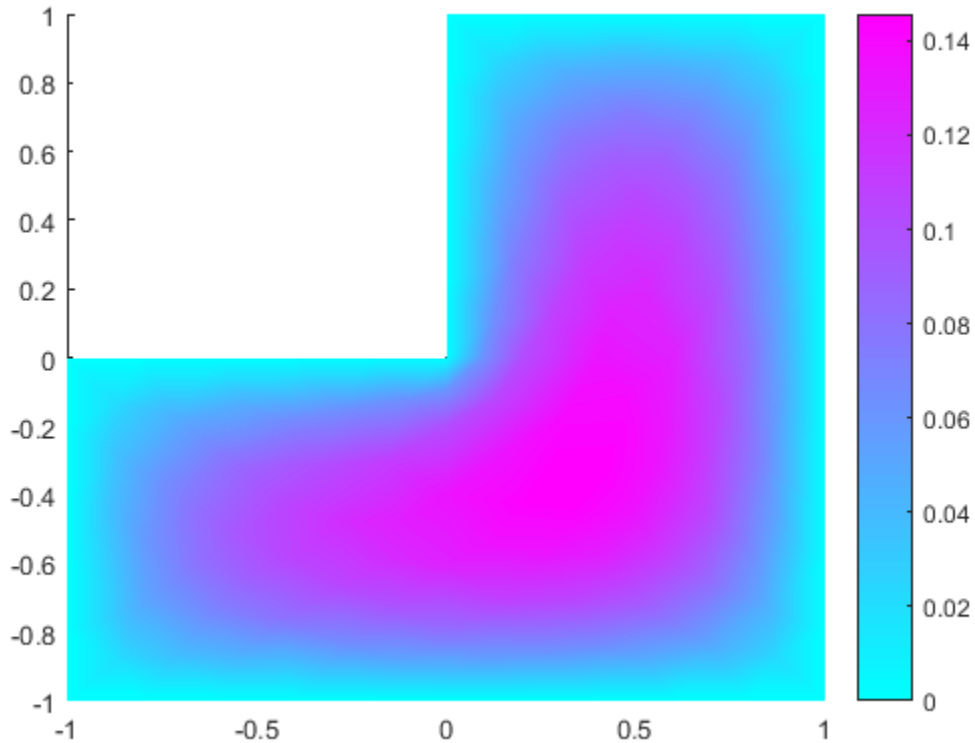Include the coefficients for Poisson's equation in the model.

```
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
```

Mesh the model and solve the PDE.

```
generateMesh(model,'Hmax',0.25);
results = solvepde(model);
```

View the solution.

```
pdeplot(model,'xydata',results.NodalSolution)
```

Solve a Time-Dependent Parabolic Equation with Nonconstant Coefficients

Create a model with 3-D rectangular block geometry.

```
model = createpde();
importGeometry(model,'Block.stl');
```

Suppose that radiative cooling causes the solution to decrease as the cube of temperature on the surface of the block.

```
gfun = @(region,state)-state.u.^3*1e-6;
applyBoundaryCondition(model,'face',1:model.Geometry.NumFaces,'g',gfun);
```

The model coefficients have no source term.

```
specifyCoefficients(model,'m',0,...
                          'd',1,...
                          'c',1,...
                          'a',0,...
                          'f',0);
```

The block starts at a constant temperature of 350.
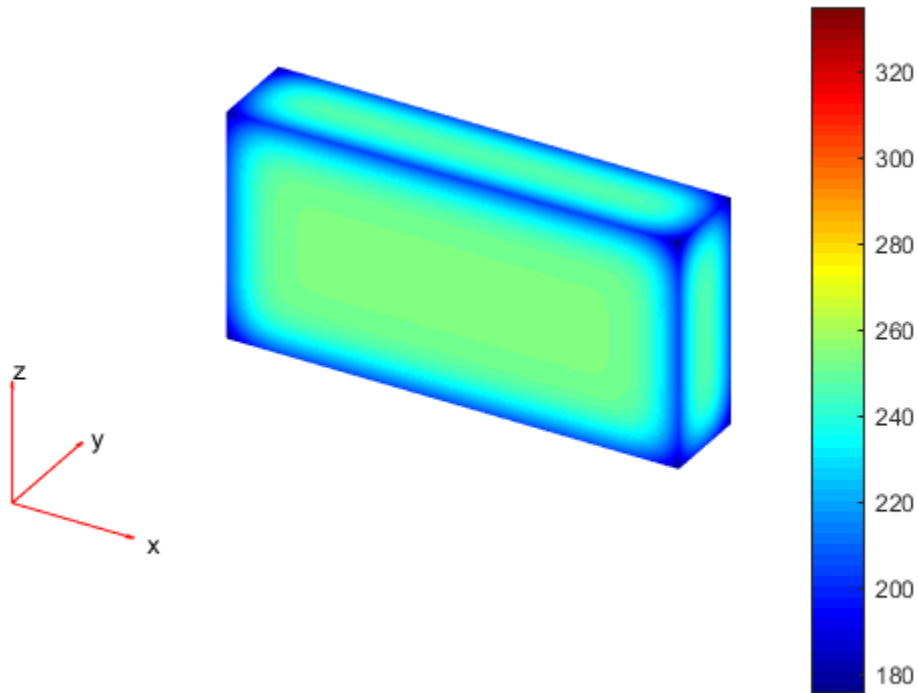
```
setInitialConditions(model,350);
```

Mesh the geometry and solve the model for times 0 through 20.

```
generateMesh(model);
tlist = 0:20;
results = solvepde(model,tlist);
```

Plot the solution on the surface of the block at times 1 and 20.

```
pdeplot3D(model,'colormapdata',results.NodalSolution(:,2));
figure
pdeplot3D(model,'colormapdata',results.NodalSolution(:,21));
```
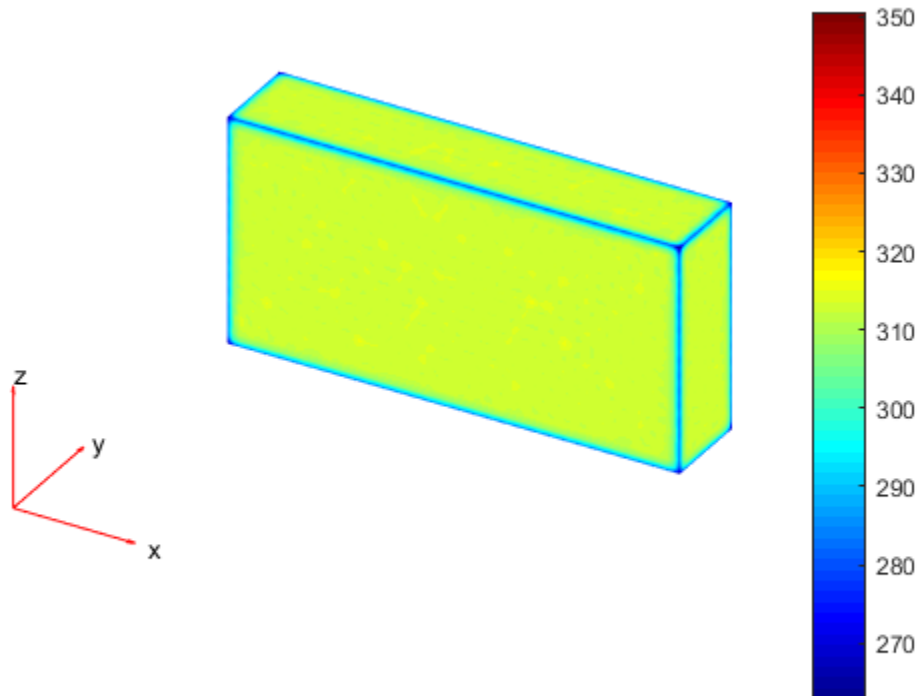
- "Partial Differential Equation Toolbox Examples"
- "Solve Problems Using PDEModel Objects" on page 2-14

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object. The model contains the geometry, mesh, and problem coefficients.

Example: model = createpde(1)

`tlist` — Solution times
real vector

Solution times, specified as a real vector. `tlist` must be a monotone vector (increasing or decreasing).

Example: `0:20`

Data Types: `double`

Output Arguments

`result` — PDE results
`StationaryResults` object | `TimeDependentResults` object

PDE results, returned as a `StationaryResults` object or as a `TimeDependentResults` object. The type of `result` depends on whether `model` represents a stationary problem (`model.IsTimeDependent = false`) or a time-dependent problem (`model.IsTimeDependent = true`).

See Also
`applyBoundaryCondition` | `PDEModel` | `setInitialConditions` | `solvepdeeig` | `specifyCoefficients`

Introduced in R2016a

solvepdeeig

Solve PDE eigenvalue problem specified in a PDEModel

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW. For the corresponding step in the legacy workflow, see `pdeeig`.

Syntax

```
result = solvepdeeig(model,evr)
```

Description

`result = solvepdeeig(model,evr)` solves the PDE eigenvalue problem in `model` for eigenvalues in the range `evr`.

Examples

Solve an Eigenvalue Problem With 3-D Geometry

Solve for several vibrational modes of the `BracketTwoHoles` geometry.

The equations of elasticity have three components. Therefore, create a PDE model that has three components. Import and view the `BracketTwoHoles` geometry.

```
model = createpde(3);
importGeometry(model,'BracketTwoHoles.stl');
h = pdegplot(model,'FaceLabels','on');
h(1).FaceAlpha = 0.5;
```
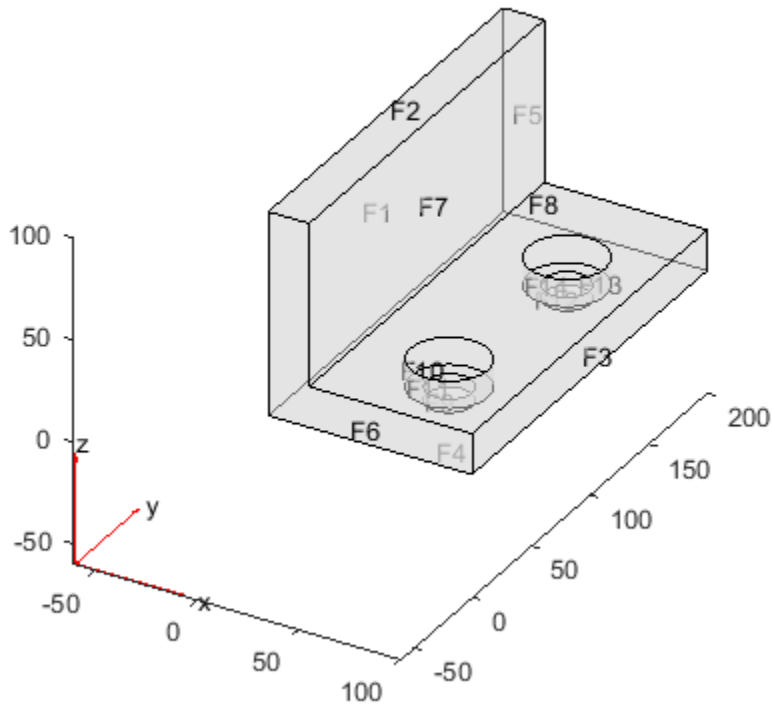
Set F1, the rear face, to have zero deflection.

```
applyBoundaryCondition(model,'face',1,'u',[0;0;0]);
```

Set the model coefficients to represent a steel bracket. For details, see "3-D Linear Elasticity Equations in Toolbox Form" on page 3-35.

```
E = 200e9; % elastic modulus of steel in Pascals
nu = 0.3; % Poisson's ratio
specifyCoefficients(model,'m',0,...
                          'd',1,...
                          'c',elasticityC3D(E,nu),...
                          'a',0,...
                          'f',[0;0;0]); % Assume all body forces are zero
```

Find the eigenvalues up to `1e7`.

```
evr = [-Inf,1e7];
```

Mesh the model and solve the eigenvalue problem.

```
generateMesh(model);
results = solvepdeeig(model,evr);
```

How many results did `solvepdeeig` return?

```
length(results.Eigenvalues)
```

```
ans =
     3
```

Plot the solution on the geometry boundary for the lowest eigenvalue.

```
V = results.Eigenvectors;
subplot(2,2,1)
pdeplot3D(model,'colormapdata',V(:,1,1));
title('x Deflection, Mode 1')
subplot(2,2,2)
pdeplot3D(model,'colormapdata',V(:,2,1));
title('y Deflection, Mode 1')
subplot(2,2,3)
pdeplot3D(model,'colormapdata',V(:,3,1));
title('z Deflection, Mode 1')
```
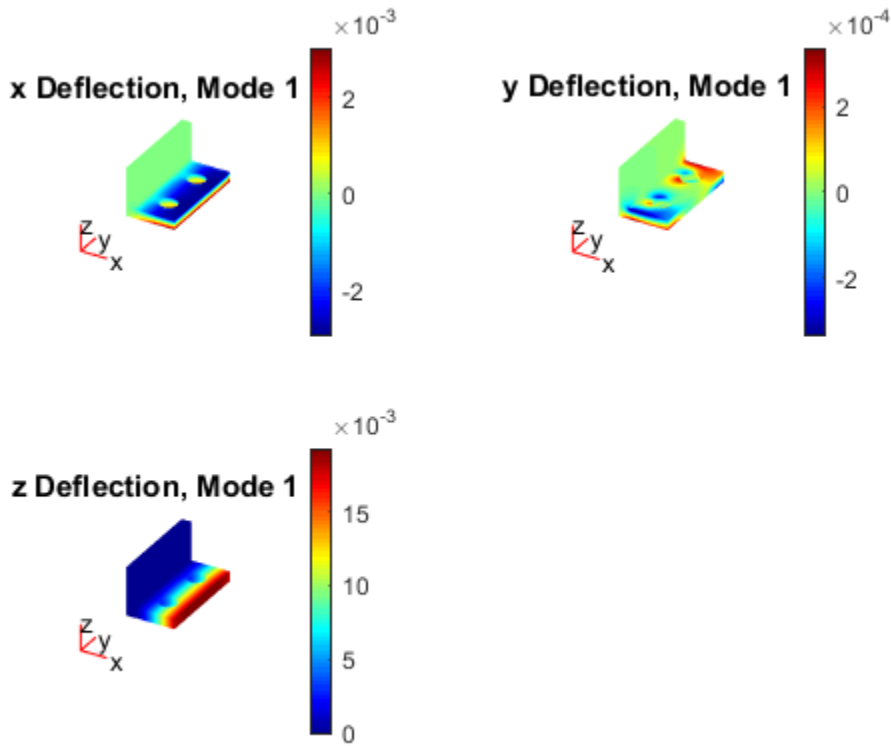
Plot the solution for the highest eigenvalue.

```
figure
subplot(2,2,1)
pdeplot3D(model,'colormapdata',V(:,1,3));
title('x Deflection, Mode 3')
subplot(2,2,2)
pdeplot3D(model,'colormapdata',V(:,2,3));
title('y Deflection, Mode 3')
subplot(2,2,3)
pdeplot3D(model,'colormapdata',V(:,3,3));
title('z Deflection, Mode 3')
```
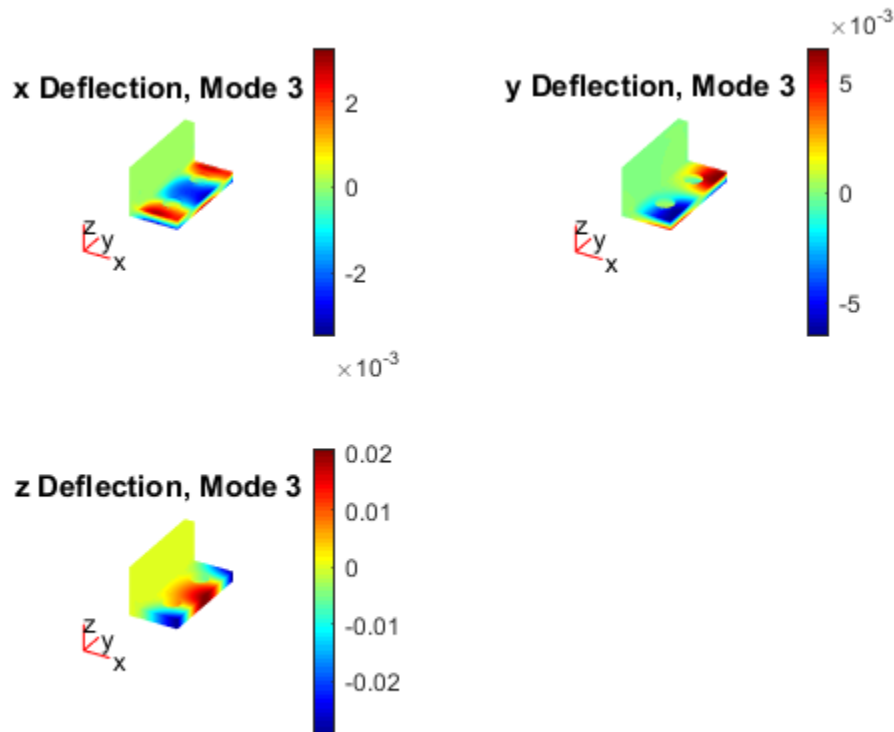
Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object. The model contains the geometry, mesh, and problem coefficients.

Example: model = createpde(1)

`evr` — Eigenvalue range
two-element real vector

Eigenvalue range, specified as a two-element real vector. `evr(1)` specifies the lower limit of the range of the real part of the eigenvalues, and may be `-Inf`. `evr(2)` specifies the upper limit of the range, and must be finite.

Example: `[-Inf;100]`

Data Types: `double`

Output Arguments

`result` — Eigenvalue results
`EigenResults` object

Eigenvalue results, returned as an `EigenResults` object.

More About

Tips

• The equation coefficients cannot depend on the solution `u` or its gradient.

See Also

`applyBoundaryCondition` | `PDEModel` | `solvepde` | `specifyCoefficients`

Introduced in R2016a

specifyCoefficients

Specify coefficients in a PDE model

Coefficients of a PDE

`solvepde` solves PDEs of the form

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

`solvepdeeig` solves PDE eigenvalue problems of the form

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

or

$$-\nabla\cdot(c\nabla u) + au = \lambda^2 mu.$$

`specifyCoefficients` defines the coefficients m, d, c, a, and f in the PDE model.

Compatibility

THIS PAGE DESCRIBES THE RECOMMENDED WORKFLOW. For the corresponding step in the legacy workflow, see the legacy examples on the "PDE Coefficients" page.

Syntax

```
specifyCoefficients(model,Name,Value)
CA = specifyCoefficients(model,Name,Value)
```

Description

`specifyCoefficients(model,Name,Value)` defines the specified coefficients in each `Name` to each associated `Value`, and includes them in `model`. You must specify all of these names: `m`, `d`, `c`, `a`, and `f`. If you do not specify a `'face'` name, the coefficients apply to the entire geometry.

Note: Include geometry in `model` before using `specifyCoefficients`.

`CA = specifyCoefficients(model,Name,Value)` returns a handle to the coefficient assignment object in `model`.

Examples

Specify Poisson's Equation

Specify the coefficients for Poisson's equation

$$-\nabla\cdot\nabla u = 1.$$

`solvepde` addresses equations of the form

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

Therefore, the coefficients for Poisson's equation are `m = 0`, `d = 0`, `c = 1`, `a = 0`, `f = 1`. Include these coefficients in a PDE model of the L-shaped membrane.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
```
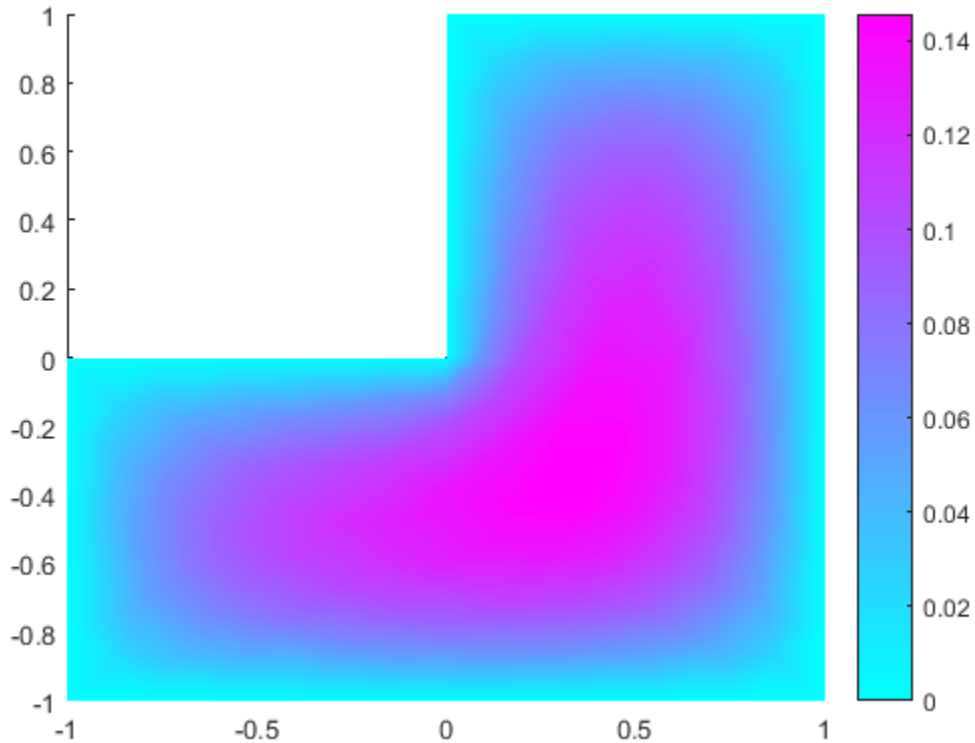
Specify zero Dirichlet boundary conditions, mesh the model, and solve the PDE.

```
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
generateMesh(model,'Hmax',0.25);
results = solvepde(model);
```

View the solution.

```
pdeplot(model,'xydata',results.NodalSolution)
```

Coefficient Handle for Nonconstant Coefficients

Specify coefficients for Poisson's equation in 3-D with a nonconstant source term, and obtain the coefficient object.

As in the example "Specify Poisson's Equation" on page 6-390, the equation coefficients are `m = 0`, `d = 0`, `c = 1`, `a = 0`. For the nonconstant source term, take $f = y^2 \tanh(z)/1000$.

```
f = @(region,state)region.y.^2.*tanh(region.z)/1000;
```

Set the coefficients in a 3-D rectangular block geometry.

```
model = createpde();
importGeometry(model,'Block.stl');
CA = specifyCoefficients(model,'m',0,...
                               'd',0,...
                               'c',1,...
                               'a',0,...
                               'f',f)

CA =

  CoefficientAssignment with properties:

    RegionType: 'cell'
      RegionID: 1
             m: 0
             d: 0
             c: 1
             a: 0
             f: @(region,state)region.y.^2.*tanh(region.z)/1000
```
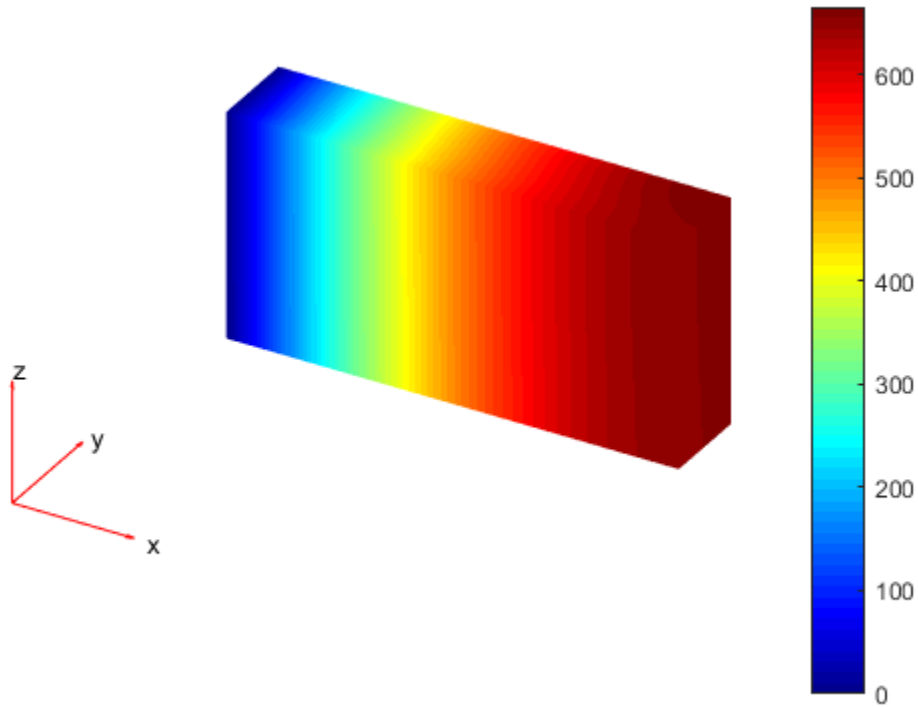
Set zero Dirichlet conditions on face 1, mesh the geometry, and solve the PDE.

```
applyBoundaryCondition(model,'face',1,'u',0);
generateMesh(model);
results = solvepde(model);
```
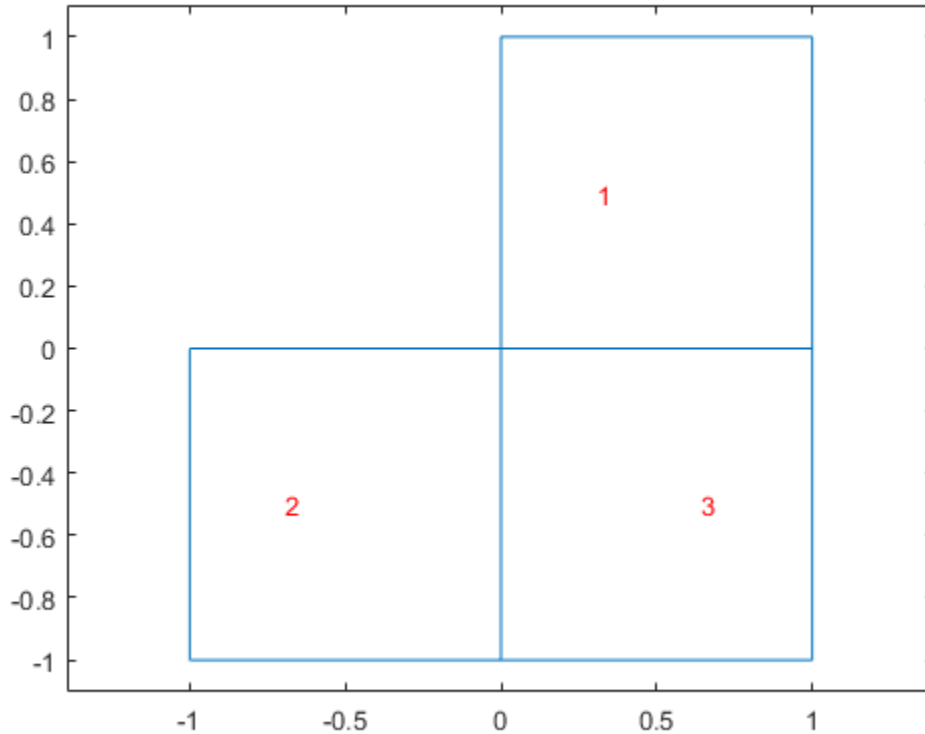
View the solution on the surface.

```
pdeplot3D(model,'colormapdata',results.NodalSolution);
```

Specify Coefficients Depending On Subdomain

Create a scalar PDE model with the L-shaped membrane as the geometry. Plot the geometry and subdomain labels

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'SubdomainLabels','on')
axis equal
ylim([-1.1,1.1])
```

Set the c coefficient to 1 in all domains, but the f coefficient to 1 in subdomain 1, 5 in subdomain 2, and −8 in subdomain 3. Set all other coefficients to 0.

```
specifyCoefficients(model,'m',0,'d',0,...
    'c',1,'a',0,'f',1,'face',1);
specifyCoefficients(model,'m',0,'d',0,...
    'c',1,'a',0,'f',5,'face',2);
specifyCoefficients(model,'m',0,'d',0,...
    'c',1,'a',0,'f',-8,'face',3);
```

Set zero Dirichlet boundary conditions to all edges. Create a mesh, solve the PDE, and plot the result.

```
applyBoundaryCondition(model,'edge',1:model.Geometry.NumEdges,'u',0);
```

```
generateMesh(model,'Hmax',0.25);
results = solvepde(model);
pdeplot(model,'xydata',results.NodalSolution)
```
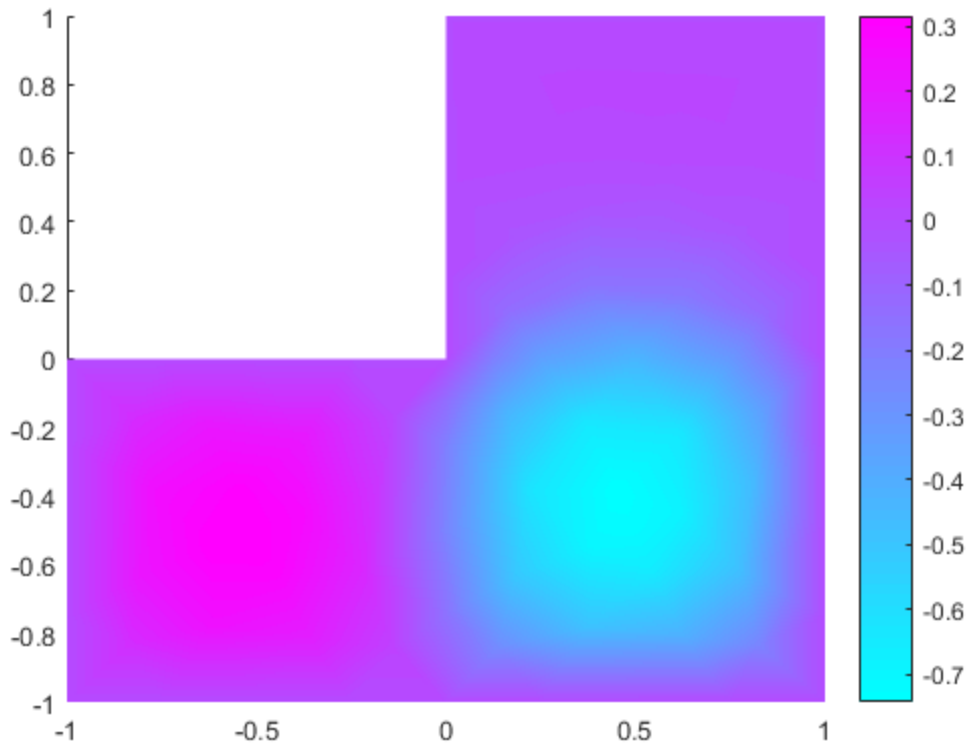

Input Arguments

`model` — PDE model
PDEModel object

PDE model, specified as a `PDEModel` object.

Example: model = createpde(1)

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `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`.

Note: You must specify all of these names: `m`, `d`, `c`, `a`, and `f`.

Example:
`specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',@fcoeff)`

`'m'` — Second-order time derivative coefficient
scalar | column vector | function handle

Second-order time derivative coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

Specify `0` if the term is not part of your problem.

Example:
`specifyCoefficients('m',@mcoef,'d',0,'c',1,'a',0,'f',1,'face',1:4)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'d'` — First-order time derivative coefficient
scalar | column vector | function handle

First-order time derivative coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

Note: If the `m` coefficient is nonzero, `d` must be `0` or a matrix, and not a function handle. See "d Coefficient When m is Nonzero" on page 6-398.

Specify `0` if the term is not part of your problem.

Example:
```
specifyCoefficients('m',0,'d',@dcoef,'c',1,'a',0,'f',1,'face',1:4)
```

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'c'` — Second-order space derivative coefficient
scalar | column vector | function handle

Second-order space derivative coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see "c Coefficient for `specifyCoefficients`" on page 2-104.

Example:
```
specifyCoefficients('m',0,'d',0,'c',@ccoef,'a',0,'f',1,'face',1:4)
```

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'a'` — Solution multiplier coefficient
scalar | column vector | function handle

Solution multiplier coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see "m, d, or a Coefficient for specifyCoefficients" on page 2-143.

Specify 0 if the term is not part of your problem.

Example:
```
specifyCoefficients('m',0,'d',0,'c',1,'a',@acoef,'f',1,'face',1:4)
```

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`'f'` — Source coefficient
scalar | column vector | function handle

Source coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see "f Coefficient for `specifyCoefficients`" on page 2-101.

Specify 0 if the term is not part of your problem.

Example:
```
specifyCoefficients('m',0,'d',0,'c',1,'a',0,'f',@fcoeff,'face',1:4)
```

Data Types: `double | function_handle`
Complex Number Support: Yes

`'face'` — Geometry faces
vector of face IDs

Geometry faces, specified as a vector of face IDs. Applies to 2-D geometry with subdomains. The face IDs are the subdomain labels.

Tip View the subdomain labels by executing

`pdegplot(model,'SubdomainLabels','on')`

If you do not specify `'face'`, then the resulting coefficients apply to the entire geometry.

Example: `specifyCoefficients('m',0,'d',0,'c',1,'a',0,'f',1,'face',1:4)`

Data Types: `double`

Output Arguments

CA — Coefficient assignment
`CoefficientAssignment` object

Coefficient assignment, returned as a `CoefficientAssignment` object.

More About

d Coefficient When m is Nonzero

The `d` coefficient takes a special matrix form when `m` is nonzero. You must specify `d` as a matrix of a particular size, and not as a function handle.

`d` represents a damping coefficient in the case of nonzero `m`. To specify `d`, perform these two steps:

1. Call `results = assembleFEMatrices(...)` for the problem with your original coefficients and using `d = 0`. Use the default `'none'` method for `assembleFEMatrices`.

2 Take the d coefficient as a matrix of size `results.M`. Generally, d is either proportional to `results.M`, or is a linear combination of `results.M` and `results.K`.

See "Dynamics of a Damped Cantilever Beam".

Tips

- For eigenvalue equations, the coefficients cannot depend on the solution u or its gradient.

See Also
`findCoefficients` | `PDEModel`

Introduced in R2016a

sptarn

Solve generalized sparse eigenvalue problem

Compatibility

sptarn IS NOT RECOMMENDED. Use `solvepdeeig` instead.

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.

More About

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.

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
solvepdeeig

Using StationaryResults Objects

Contain time-independent PDE solution and derived quantities

Compatibility

A `StationaryResults` OBJECT IS USED IN BOTH THE RECOMMENDED AND THE LEGACY WORKFLOWS. It replaces a PDEResults object returned in R2015b.

Description

A `StationaryResults` object contains the solution of a PDE and its gradients in a form convenient for plotting and postprocessing.

- A `StationaryResults` object contains the solution and its gradient calculated at the nodes of the triangular or tetrahedral mesh, generated by `generateMesh`.
- Solution values at the nodes appear in the `NodalSolution` property.
- The three components of the gradient of the solution values at the nodes appear in the `XGradients`, `YGradients`, and `ZGradients` properties.
- The array dimensions of `NodalSolution`, `XGradients`, `YGradients`, and `ZGradients` enable you to extract solution and gradient values for specified equation indices in a PDE system.

To interpolate the solution or its gradient to a custom grid (for example, specified by `meshgrid`), use `interpolateSolution` or `evaluateGradient`.

Examples

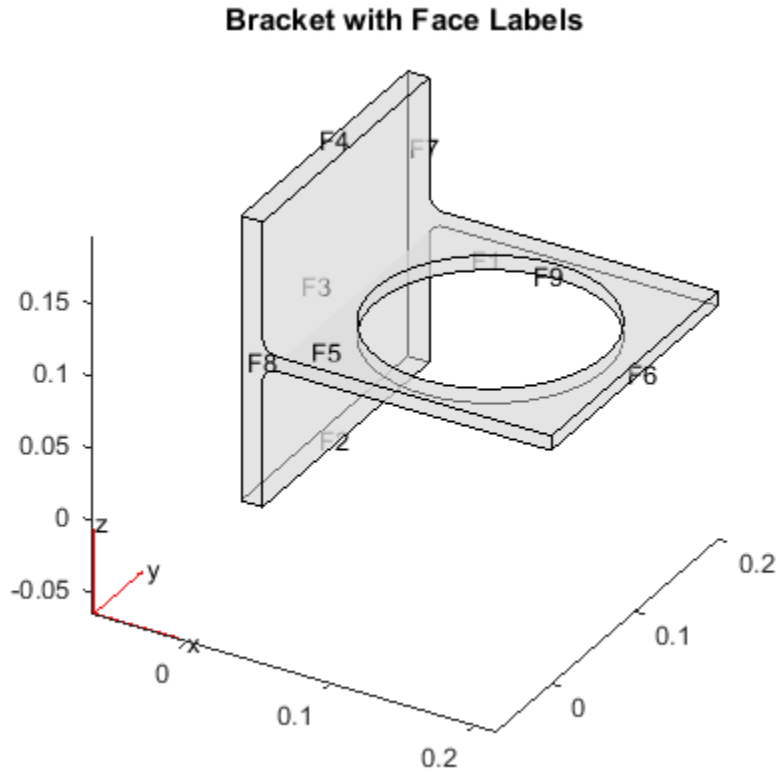### Obtain a `StationaryResults` Object From `solvepde`

Create a PDE model for a system of three equations. Import the geometry of a bracket and plot the face labels.

```
model = createpde(3);
importGeometry(model,'BracketWithHole.stl');
```

```
figure
h = pdegplot(model,'FaceLabels','on');
view(30,30);
title('Bracket with Face Labels')
h(1).FaceAlpha = 0.5;
```


Bracket with Face Labels

Set boundary conditions such that face **3** is immobile, and face **6** has a force in the negative *z* direction.

```
applyBoundaryCondition(model,'face',3,'u',[0,0,0]);
applyBoundaryCondition(model,'face',6,'g',[0,0,-1e4]);
```

Set coefficients that represent the equations of linear elasticity. See "3-D Linear Elasticity Equations in Toolbox Form" on page 3-35.

```
E = 200e9;
nu = 0.3;
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',elasticityC3D(E,nu),...
                          'a',0,...
                          'f',[0;0;0]);
```

Create a mesh.

```
generateMesh(model,'Hmax',1e-2);
```

Solve the PDE.

```
results = solvepde(model)

results =

  StationaryResults with properties:

    NodalSolution: [13437x3 double]
        XGradients: [13437x3 double]
        YGradients: [13437x3 double]
        ZGradients: [13437x3 double]
             Mesh: [1x1 pde.FEMesh]
```
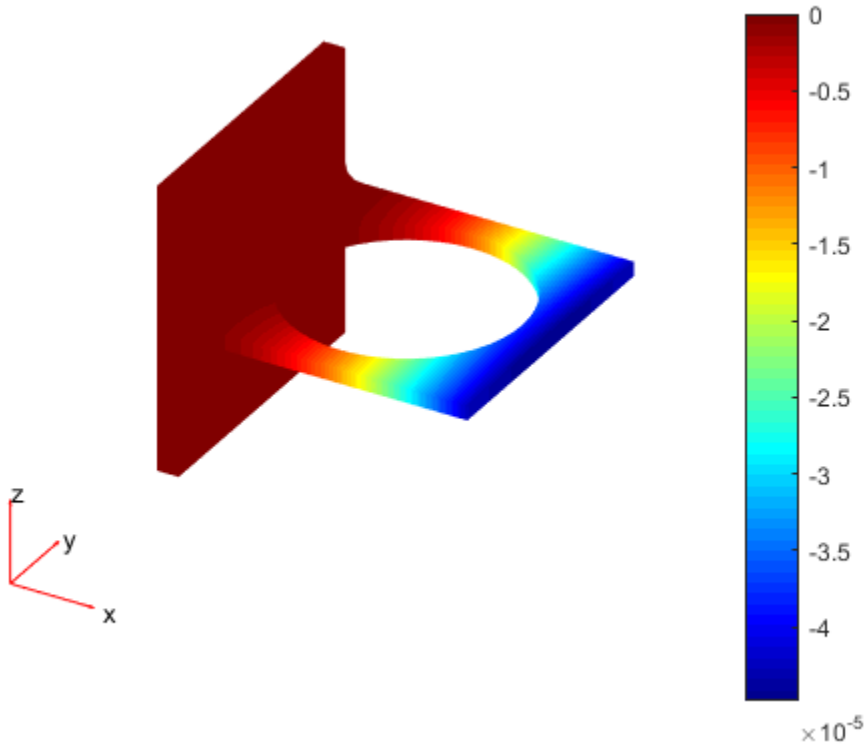
Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

Plot the solution for the *z*-component, which is component 3.

```
pdeplot3D(model,'colormapdata',u(:,3))
```

Results from `createPDEResults`

Obtain a `StationaryResults` object from a legacy solver together with `createPDEResults`.

Create a PDE model for a system of three equations. Import the geometry of a bracket and plot the face labels.

```
model = createpde(3);
importGeometry(model,'BracketWithHole.stl');
figure
h = pdegplot(model,'FaceLabels','on');
view(30,30);
```

```
title('Bracket with Face Labels')
h(1).FaceAlpha = 0.5;
```

Bracket with Face Labels
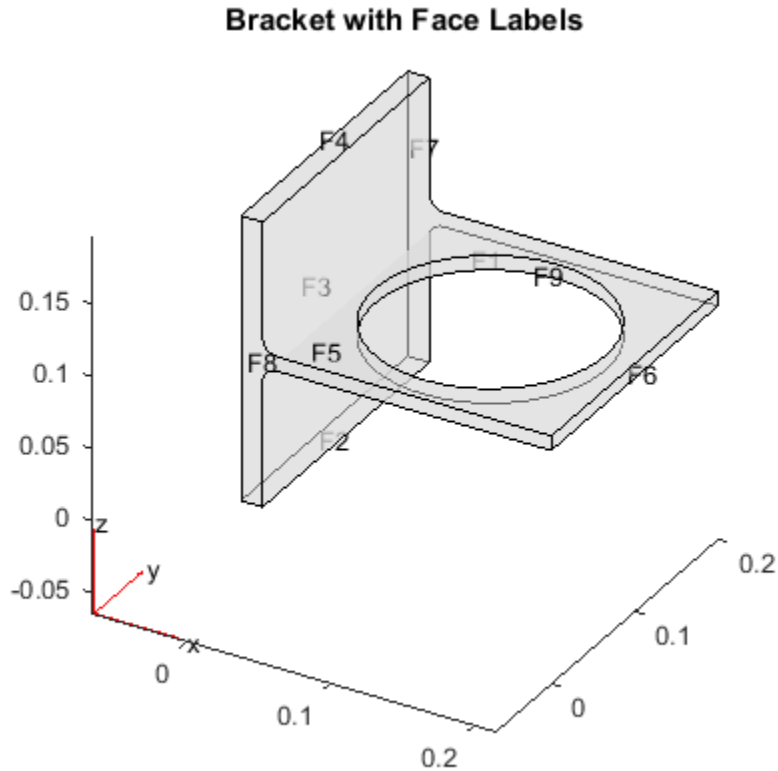
Set boundary conditions such that **F3** is immobile, and **F6** has a force in the negative z direction.

```
applyBoundaryCondition(model,'face',3,'u',[0,0,0]);
applyBoundaryCondition(model,'face',6,'g',[0,0,-1e4]);
```

Set coefficients for a legacy solver that represent the equations of linear elasticity. See "3-D Linear Elasticity Equations in Toolbox Form" on page 3-35.

```
E = 200e9;
```

```
nu = 0.3;
c = elasticityC3D(E,nu);
a = 0;
f = [0;0;0];
```

Create a mesh.

```
generateMesh(model,'Hmax',1e-2);
```

Solve the problem using a legacy solver.

```
u = assempde(model,c,a,f);
```

Create a `StationaryResults` object from the solution.

```
results = createPDEResults(model,u)

results =

  StationaryResults with properties:

    NodalSolution: [13437x3 double]
       XGradients: [13437x3 double]
       YGradients: [13437x3 double]
       ZGradients: [13437x3 double]
             Mesh: [1x1 pde.FEMesh]
```
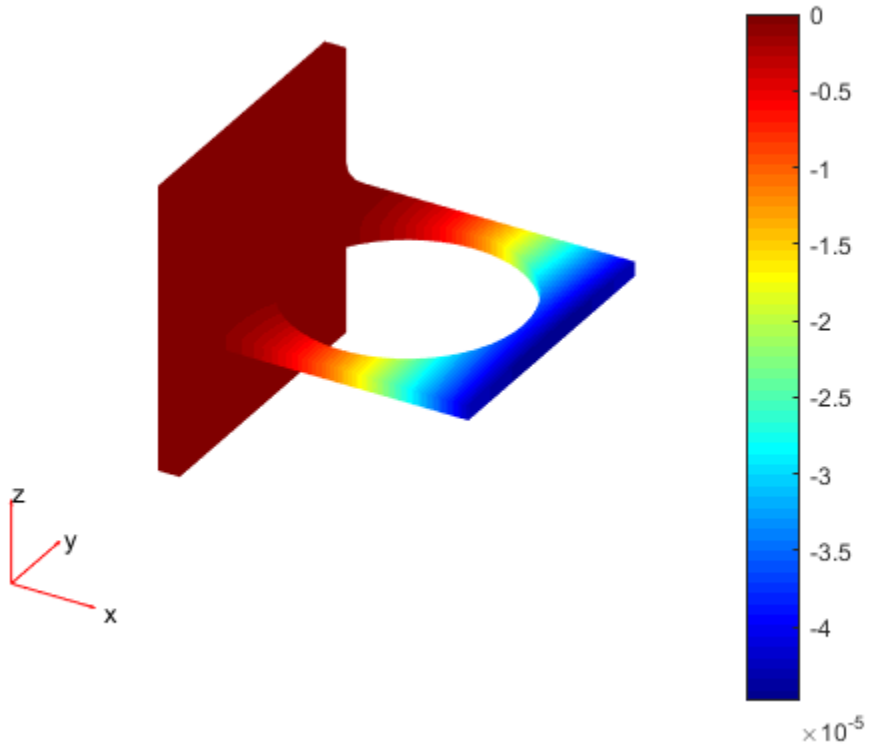
Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

Plot the solution for the *z*-component, which is component 3.

```
pdeplot3D(model,'colormapdata',u(:,3))
```

Properties

`Mesh` — Finite element mesh
FEMesh object

Finite element mesh, specified as a FEMesh object.

`NodalSolution` — Solution values at the nodes
vector | array

Solution values at the nodes, returned as a vector or array. For details about the dimensions of `NodalSolution`, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: `double`
Complex Number Support: Yes

XGradients — *x*-component of gradient at the nodes
vector | array

x-component of the gradient at the nodes, returned as a vector or array. For details about the dimensions of `XGradients`, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: `double`
Complex Number Support: Yes

YGradients — *y*-component of gradient at the nodes
vector | array

y-component of the gradient at the nodes, returned as a vector or array. For details about the dimensions of `YGradients`, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: `double`
Complex Number Support: Yes

ZGradients — *z*-component of gradient at the nodes
vector | array

z-component of the gradient at the nodes, returned as a vector or array. For details about the dimensions of `ZGradients`, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: `double`
Complex Number Support: Yes

Object Functions

evaluateGradient	Evaluate gradients of PDE solutions at arbitrary points
interpolateSolution	Interpolate PDE solution to arbitrary points

Create Object

`solvepde` returns time-independent PDE solutions as a `StationaryResults` object.

`createPDEResults` returns a `StationaryResults` object from a PDE solution as returned by `assempde` or `pdenonlin`.

See Also
`EigenResults` | `evaluateGradient` | `interpolateSolution` | `solvepde` | `TimeDependentResults`

Introduced in R2016a

Using TimeDependentResults Objects

Contain time-dependent PDE solution and derived quantities

Compatibility

A `TimeDependentResults` **OBJECT IS USED IN BOTH THE RECOMMENDED AND THE LEGACY APPROACH.** It replaces a `PDEResults` object returned in R2015b.

Description

A `TimeDependentResults` object contains the solution of a PDE and its gradients in a form convenient for plotting and postprocessing.

- A `TimeDependentResults` object contains the solution and its gradient calculated at the nodes of the triangular or tetrahedral mesh, generated by `generateMesh`.
- Solution values at the nodes appear in the `NodalSolution` property.
- The solution times appear in the `SolutionTimes` property.
- The three components of the gradient of the solution values at the nodes appear in the `XGradients`, `YGradients`, and `ZGradients` properties.
- The array dimensions of `NodalSolution`, `XGradients`, `YGradients`, and `ZGradients` enable you to extract solution and gradient values for specified time indices, and for the equation indices in a PDE system.

To interpolate the solution or its gradient to a custom grid (for example, specified by `meshgrid`), use `interpolateSolution` or `evaluateGradient`.

Examples

Solution of a Parabolic Problem

Solve a parabolic problem with 2-D geometry.

Create and view the geometry: a square with a circular subdomain.

```
% Square centered at (1,1), circle centered at (1.5,0.5).
```

```
rect1 = [3;4;0;2;2;0;0;0;2;2];
circ1 = [1;1.5;.75;0.25];
% Append extra zeros to the circle;
circ1 = [circ1;zeros(length(rect1)-length(circ1),1)];
gd = [rect1,circ1];
ns = char('rect1','circ1');
ns = ns';
sf = 'rect1+circ1';
[dl,bt] = decsg(gd,sf,ns);
pdegplot(dl,'EdgeLabels','on','SubdomainLabels','on')
axis equal
ylim([-0.1,2.1])
```
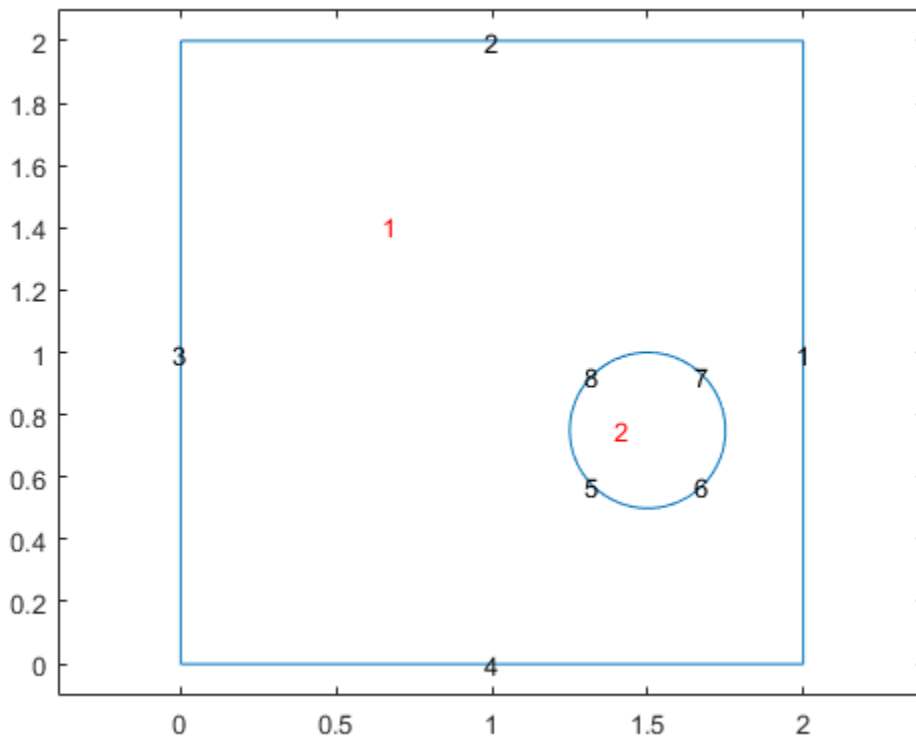

Include the geometry in a PDE model.

```
model = createpde();
geometryFromEdges(model,dl);
```

Set boundary conditions that the upper and left edges are at temperature 10.

```
applyBoundaryCondition(model,'edge',[2,3],'u',10);
```

Set initial conditions that the square region is at temperature 0, and the circle is at temperature 100.

```
setInitialConditions(model,0);
setInitialConditions(model,100,'face',2);
```

Define the model coefficients.

```
specifyCoefficients(model,'m',0,...
                          'd',1,...
                          'c',1,...
                          'a',0,...
                          'f',0);
```

Solve the problem for times 0 through 1/2 in steps of 0.01.

```
generateMesh(model,'Hmax',0.05);
tlist = 0:0.01:0.5;
results = solvepde(model,tlist);
```

Plot the solution for times 0.02, 0.04, 0.1, and 0.5.

```
sol = results.NodalSolution;
subplot(2,2,1)
pdeplot(model,'xydata',sol(:,3))
title('Time 0.02')
subplot(2,2,2)
pdeplot(model,'xydata',sol(:,5))
title('Time 0.04')
subplot(2,2,3)
pdeplot(model,'xydata',sol(:,11))
title('Time 0.1')
subplot(2,2,4)
pdeplot(model,'xydata',sol(:,51))
title('Time 0.5')
```
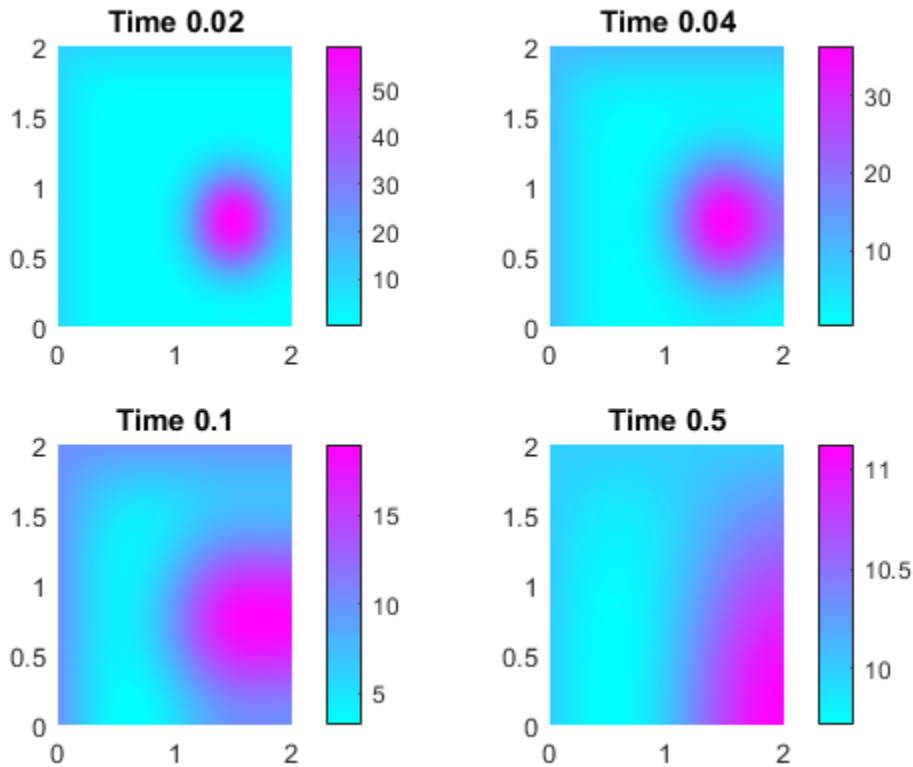
- "Metal Block Using Command-Line Functions" on page 3-98
- "Wave Equation Using Command-Line Functions" on page 3-106
- "Solve Problems Using PDEModel Objects" on page 2-14

Properties

Mesh — Finite element mesh
FEMesh object

Finite element mesh, specified as a FEMesh object.

NodalSolution — Solution values at the nodes
vector | array

Solution values at the nodes, returned as a vector or array. For details about the dimensions of NodalSolution, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: double
Complex Number Support: Yes

SolutionTimes — Solution times
real vector

Solution times, returned as a real vector. SolutionTimes is the same as the tlist input to solvepde, or the tlist input to the legacy parabolic or hyperbolic solvers.

Data Types: double

XGradients — *x*-component of gradient at the nodes
vector | array

x-component of the gradient at the nodes, returned as a vector or array. For details about the dimensions of XGradients, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: double
Complex Number Support: Yes

YGradients — *y*-component of gradient at the nodes
vector | array

y-component of the gradient at the nodes, returned as a vector or array. For details about the dimensions of YGradients, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: double
Complex Number Support: Yes

ZGradients — *z*-component of gradient at the nodes
vector | array

z-component of the gradient at the nodes, returned as a vector or array. For details about the dimensions of ZGradients, see "Dimensions of Solutions and Gradients" on page 3-167.

Data Types: `double`
Complex Number Support: Yes

Object Functions

evaluateGradient   Evaluate gradients of PDE solutions at
                                     arbitrary points
interpolateSolution   Interpolate PDE solution to arbitrary points

Create Object

`solvepde` returns time-dependent PDE solutions as a `TimeDependentResults` object.

`createPDEResults` returns a `TimeDependentResults` object from a PDE solution as returned by the legacy `parabolic` or `hyperbolic` solvers.

See Also
`EigenResults` | `evaluateGradient` | `interpolateSolution` | `StationaryResults`

Introduced in R2016a

tri2grid

Interpolate from PDE triangular mesh to rectangular grid

Compatibility

`tri2grid` IS NOT RECOMMENDED. Use `interpolateSolution` instead.

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. `u` must be a vector. For systems of equations, `uxy` interpolates only the first component. For solutions returned by `hyperbolic` or `parabolic`, pass `u` as the vector of values at one time, `u(:,k)`.

`[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, such as interpolating hyperbolic or parabolic solutions at multiple times.

All `tri2grid` output arguments are `ny`-by-`nx` matrices, where `nx` and `ny` are the lengths of the vectors `x` and `y` respectively. At grid points outside of the triangular mesh, all `tri2grid` output arguments are `NaN`.

See Also
solvepde | interpolateSolution

wbound

Write boundary condition specification file

Compatibility

wbound IS NOT RECOMMENDED. Use `applyBoundaryCondition` instead.

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.

The output file `[mn,'.m']` is the name of a Boundary file. (See "Boundary Conditions by Writing Functions" on page 2-199.)

See Also

decsg | wgeom

wgeom

Write geometry specification function

Compatibility

THIS PAGE DESCRIBES THE LEGACY WORKFLOW. New features might not be compatible with the legacy workflow.

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 "Decomposed Geometry Data Structure" on page 2-24.

The output file `[mn,'.m']` is the name of a Geometry file. For a description of the Geometry file format, see "Create Geometry Using a Geometry Function" on page 2-26.

See Also
decsg