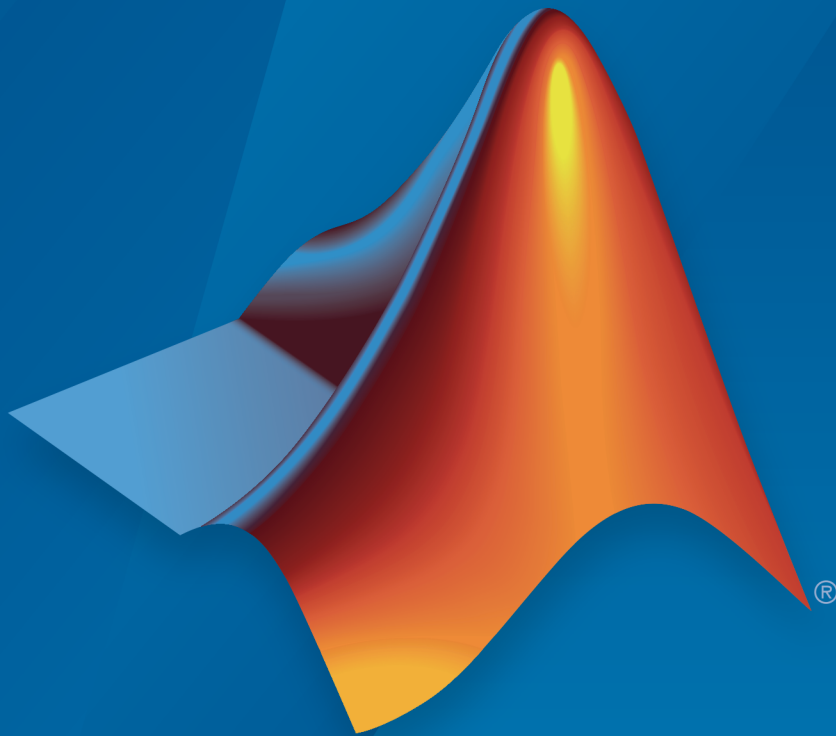# Partial Differential Equation Toolbox™

## User's Guide

**MATLAB**®

MathWorks®

# 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

# Solving PDEs

## 3

# Functions — Alphabetical List

**6**

**1**

# 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 $\Omega$, 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. $\Omega$ 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 $\Omega$. $c$ can be a matrix function on $\Omega$ (see "c Coefficient for Systems" on page 2-110). 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 $\Omega$, and $\lambda$ 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 $\nabla u$. 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 $\nabla u$. 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.

A system of PDEs with $N$ components is $N$ coupled PDEs with coupled boundary conditions. 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 a 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.

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:

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

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 $\mu$ 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 3-192.

# 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

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

or

$$-\nabla \cdot (c\nabla u) + au = \lambda^2 mu$$

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

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

  where $h$ and $r$ can be functions of space ($x$, $y$, and, in 3-D case, $z$), the solution $u$, and time. Often, you take $h = 1$, and set $r$ to the appropriate value.

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

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

  $\vec{n}$ is the outward unit normal. $q$ and $g$ are functions defined on $\partial\Omega$, and can be functions of $x$, $y$, and, in 3-D case, $z$, the solution $u$, and, for time-dependent equations, time.

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

A system of PDEs with $N$ components is $N$ coupled PDEs with coupled boundary conditions. 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 a 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.

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 $\mathbf{m}$, $\mathbf{d}$, and $\mathbf{a}$ are $N$-by-$N$ matrices, $\mathbf{f}$ is an $N$-by-1 vector, and $\mathbf{c}$ is a $2N$-by-$2N$ tensor (2-D geometry) or a $3N$-by-$3N$ tensor (3-D geometry). For the meaning of $\mathbf{c}\otimes\mathbf{u}$ , see "c Coefficient for `specifyCoefficients`" on page 2-89.

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.

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

- $\mathbf{hu} = \mathbf{r}$ represents a matrix $\mathbf{h}$ multiplying the solution vector $\mathbf{u}$, and equaling the vector $\mathbf{r}$.

- $\mathbf{n}\cdot(\mathbf{c}\otimes\nabla\mathbf{u}) + \mathbf{qu} = \mathbf{g}$ . For 2-D systems, 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 3-D systems, the notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ means the $N$-by-1 vector with $(i,1)$-component

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

where the outward normal vector of the boundary $\mathbf{n} = (\sin(\varphi)\cos(\theta), \sin(\varphi)\sin(\theta), \cos(\varphi))$.

For each edge or face segment, there are a total of $N$ boundary conditions.

## Related Examples

- "Solve Problems Using PDEModel Objects" on page 2-5
- "f Coefficient for `specifyCoefficients`" on page 2-86
- "c Coefficient for `specifyCoefficients`" on page 2-89
- "m, d, or a Coefficient for specifyCoefficients" on page 2-128

# 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:

- "Electrostatics and Magnetostatics"
- "Structural Mechanics"
- "AC Power Electromagnetics"
- "DC Conduction and Elliptic Problems"
- "Heat Transfer"

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

To solve 2-D PDE problems using the PDE app follow these steps:

1   Start the PDE app by using the **Apps** tab or typing `pdetool` in the MATLAB Command Window. For details, see "Open the PDE App" on page 4-2.

2   Choose the application mode by selecting **Application** from the **Options** menu.

3   Create a 2-D geometry by drawing, rotating, and combining the basic shapes: circles, ellipses, rectangles, and polygons. To draw and rotate shapes, use the **Draw** menu or the corresponding toolbar buttons. To combine shapes, use the **Set formula** field. See "Create 2-D Geometry in the PDE App" on page 4-3.

4   Specify boundary conditions for each boundary segment. To do this, first switch to the **Boundary Mode** by using the **Boundary** menu. Click the boundary to select it, then specify the boundary condition for that boundary. You can have different types of boundary conditions on different boundary segments. The default boundary condition is the Dirichlet condition $hu = r$ with $h = 1$ and $r = 0$. You can remove unnecessary subdomain borders by selecting **Remove Subdomain Border** or **Remove All Subdomain Borders** from the **Boundary** menu. For details, see "Specify Boundary Conditions in the PDE App" on page 4-14.

5   Specify PDE coefficients by selecting **PDE Mode** from the **PDE** menu. Then select a region or multiple regions for which you are specifying the coefficients. Select **PDE Specification** from the **PDE** menu or click the **PDE** button on the toolbar. Type the coefficients in the resulting dialog box. For details, see "Specify Coefficients in the PDE App" on page 4-17.

   You can specify the coefficients at any time before solving the PDE because the coefficients are independent of the geometry and the boundaries. If the PDE coefficients are material-dependent, specify them by double-clicking each particular region.

6   Generate a triangular mesh by selecting **Initialize Mesh** from the **Mesh** menu. Using the same menu, you can also refine mesh, display node and triangle labels, and control mesh parameters, letting you generate a mesh that is fine enough to adequately resolve the important features in the geometry, but is coarse enough to run in a reasonable amount of time and memory. See "Specify Mesh Parameters in the PDE App" on page 4-19.

7   Solve the PDE by clicking the **=** button or by selecting **Solve PDE** from the **Solve** menu. To use a solver with non-default parameters, select **Parameters** from the **Solve** menu to. The resulting dialog box lets you:

- Invoke and control the nonlinear and adaptive solvers for elliptic problems.

- Specify the initial values, and the times for which to generate the output for parabolic and hyperbolic problems.

- Specify the interval in which to search for eigenvalues for eigenvalue problems.

See "Adjust Solve Parameters in the PDE App" on page 4-21.

**8** When you solve the PDE, the app automatically plots the solution using the default settings. To customize the plot or plot other physical properties calculated using the solution, select **Parameters** from the **Plot** menu. See "Plot the Solution in the PDE App" on page 4-27.

## Tips

After solving the problem, you can:

- Export the solution or the mesh or both to the MATLAB 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. You also can use the adaptive mesh refiner and solver, `adaptmesh`. This option tries to find a mesh that fits the solution.

- 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 geometry. You can switch to the draw mode again by selecting **Draw Mode** from the **Draw** menu or by clicking one of the **Draw Mode** icons to add another shape.

The following are the shortcuts that you can use to skip one or more steps. In general, the PDE app adds the necessary steps automatically.

- If you do not create a geometry, the PDE app uses an L-shaped geometry with the default boundary conditions.

- If you initialize the mesh while in the draw mode, the PDE app first decomposes the geometry using the current set formula and assigns the default boundary condition to the outer boundaries. After that, it generate the mesh.
- If you refine the mesh before initializing it, the PDE app first initializes the mesh.
- If you solve the PDE without generating 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 if the solution to the current PDE is available. If not, the PDE app first solves the current PDE. The app displays the solution using the selected plot options.
- If do not specify the coefficients and use the default Generic Scalar application mode, 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 $f = 10$. The default PDE settings depend on the application mode.

## Related Examples

- "Poisson's Equation with Complex 2-D Geometry" on page 1-14
- "Solve Poisson's Equation on a Unit Disk" on page 3-120
- "Conductive Media DC" on page 3-96
- "Minimal Surface Problem" on page 3-129

# 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.
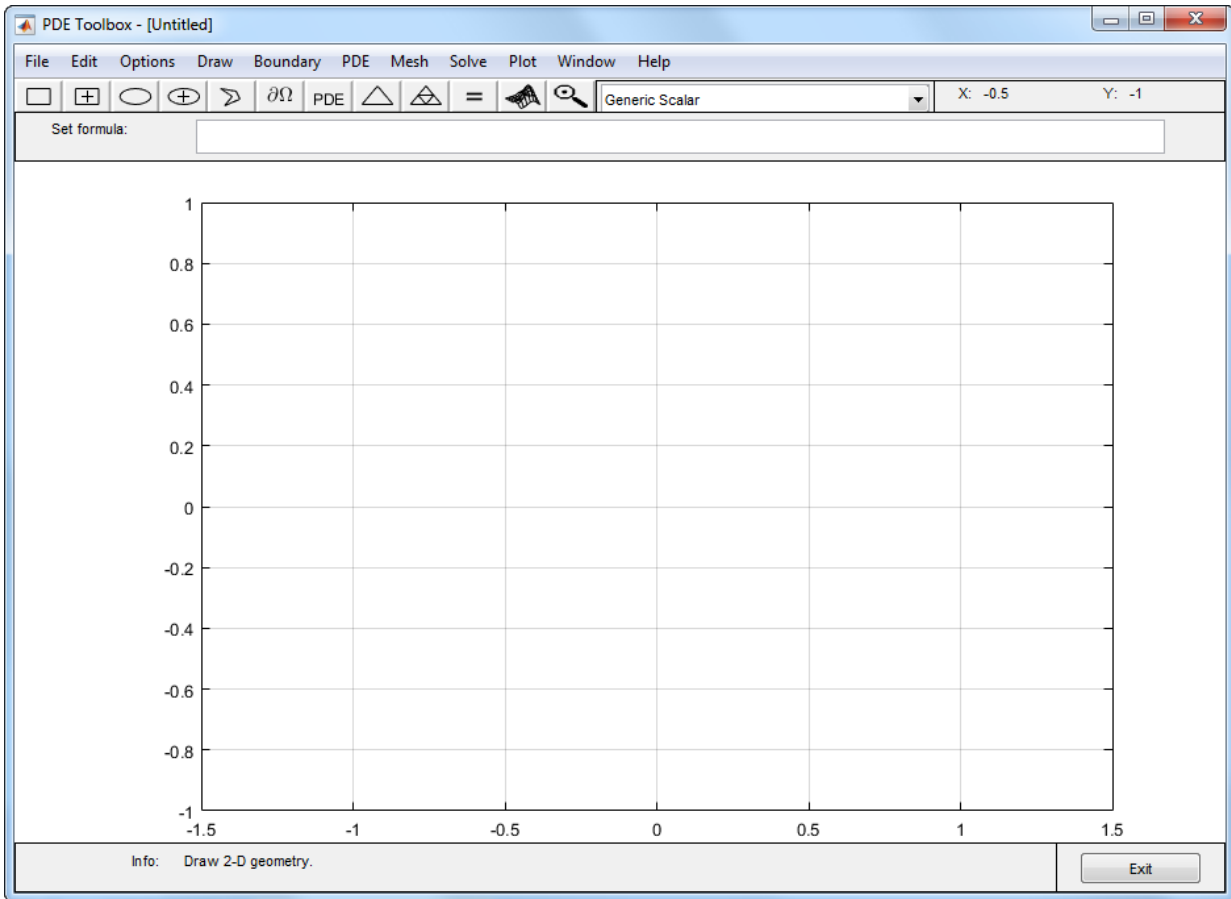
The first step is to draw the geometry on which you want to solve the PDE. The PDE app provides four basic types of *solid objects*: polygons, rectangles, circles, and ellipses. The objects are used to create a *Constructive Solid Geometry model* (CSG model). Each solid object is assigned a unique label, 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.
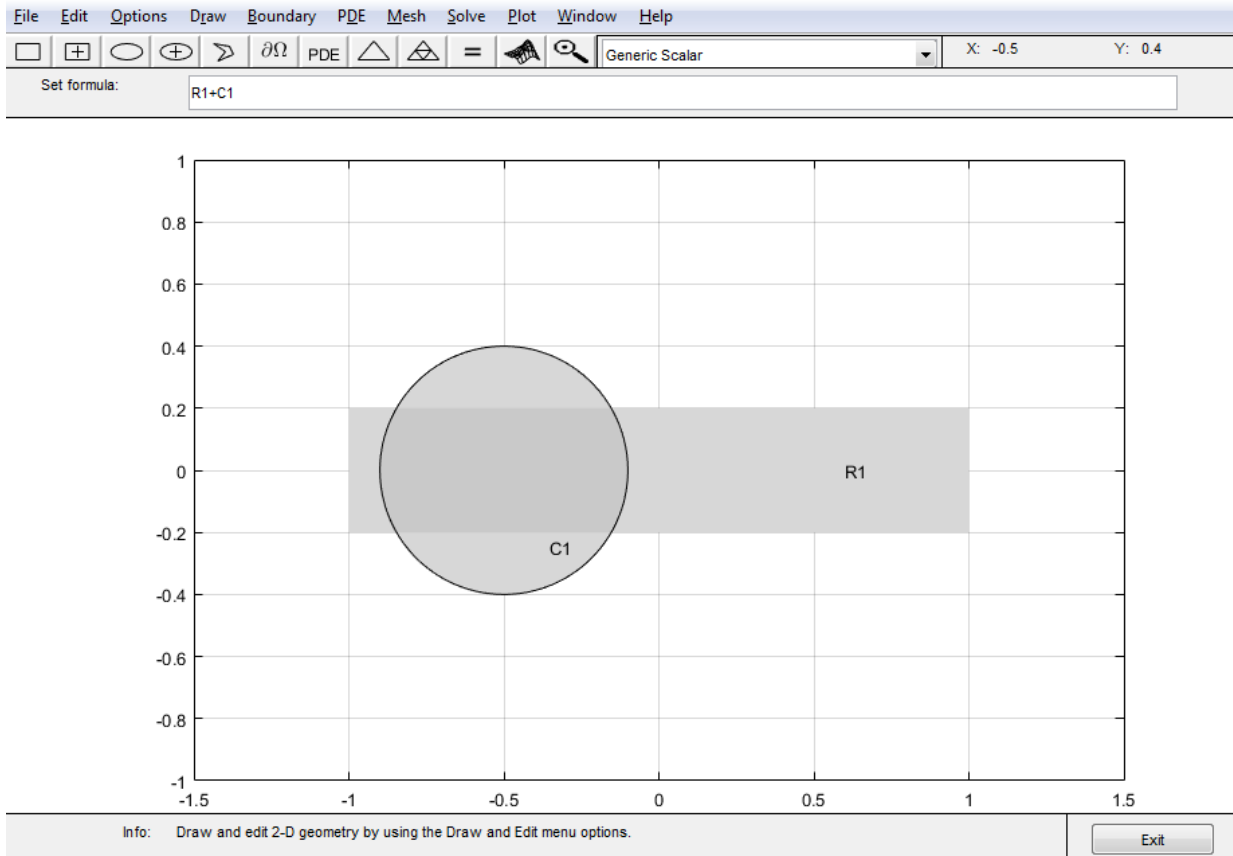
The resulting CSG model is the union of the rectangle R1 and the circle C1, described by set algebra as R1+C1. The area where the two objects overlap is clearly visible as it is drawn using a darker shade of gray. The object that you just drew—the circle—has a black border, indicating that it is selected. 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`
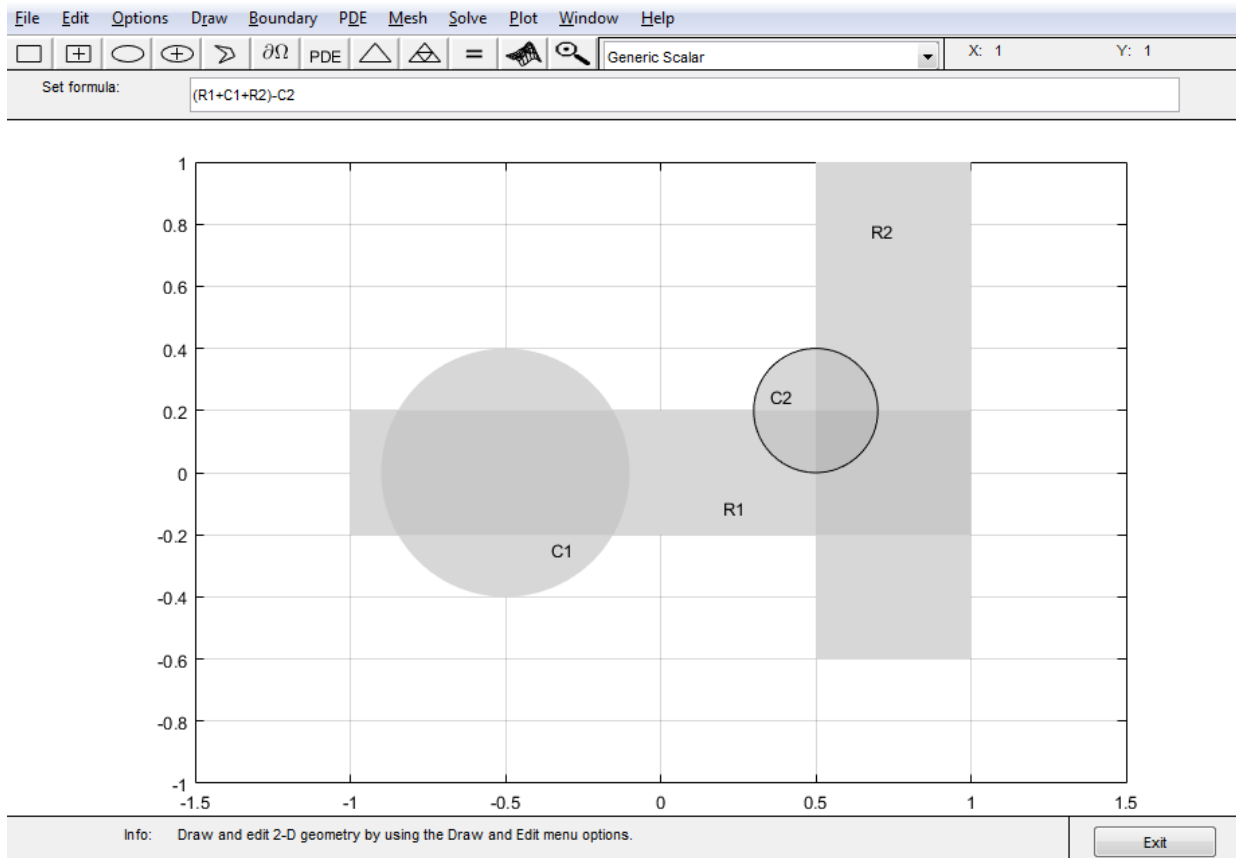


If you want, you can save this CSG model as a file. Use the **Save As** option from the **File** menu, and enter a filename of your choice. It is good practice to continue to save your model at regular intervals using **Save**. All the additional steps in the process of modeling and solving your PDE 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.
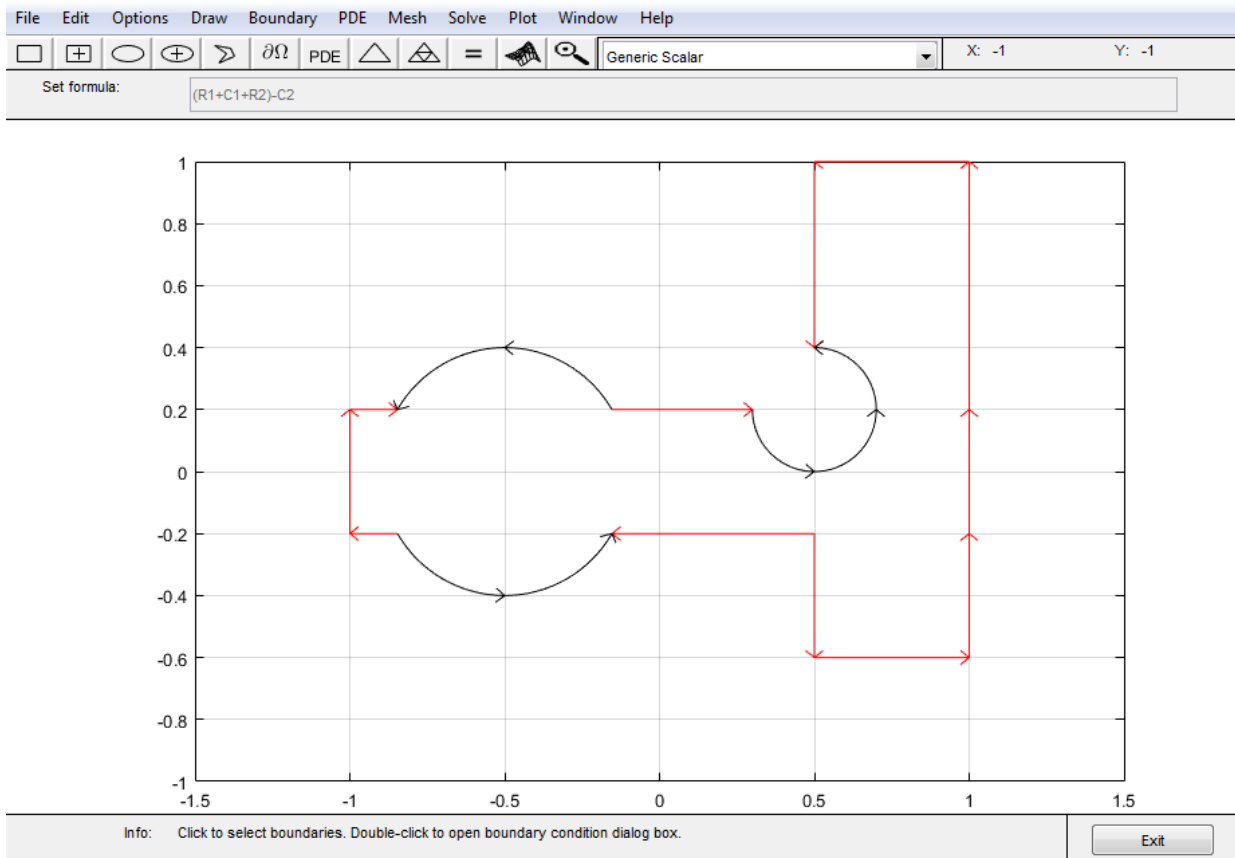
Double-clicking anywhere on the selected boundary segments opens the Boundary Condition dialog box. Here, you select the type of boundary condition, and enter the boundary condition as a MATLAB expression. Change the boundary condition along the selected boundaries to a Neumann condition, $\partial u/\partial n = -5$. 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.
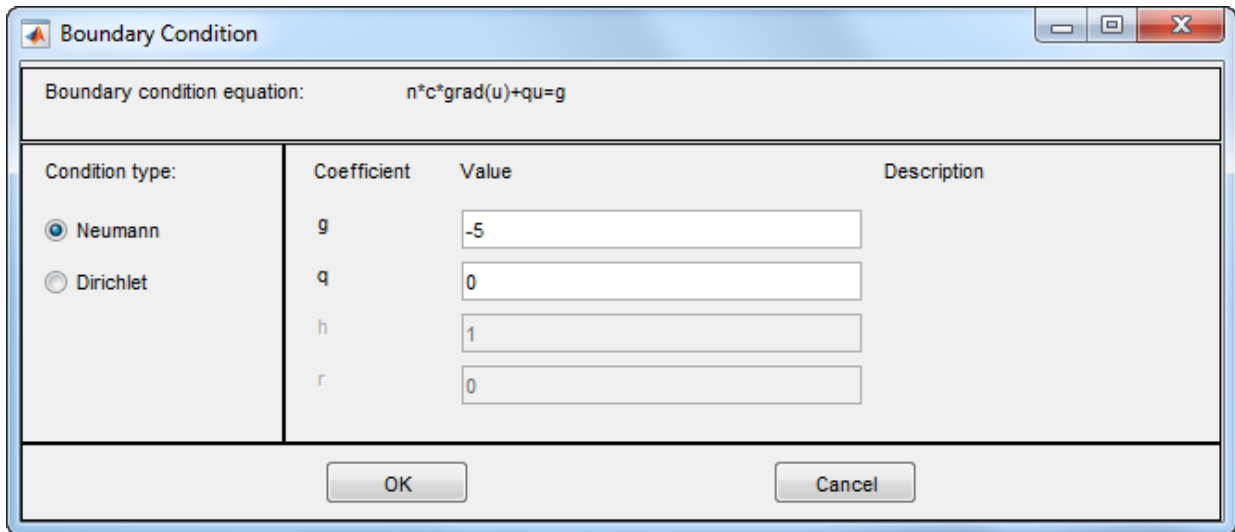
Next, specify the PDE itself through a dialog box that is accessed by clicking the button with the **PDE** icon or by selecting **PDE Specification** from the **PDE** menu. In PDE mode, you can also access the PDE Specification dialog box by double-clicking a subdomain. That way, different subdomains 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 $f = 10.0$.

**PDE Specification**

Equation:        -div(c*grad(u))+a*u=f

| Type of PDE: | Coefficient | Value |
|---|---|---|
| ⦿ Elliptic | c | 1.0 |
| ◯ Parabolic | a | 0.0 |
| ◯ Hyperbolic | f | 10.0 |
| ◯ Eigenmodes | d | 1.0 |

OK          Cancel
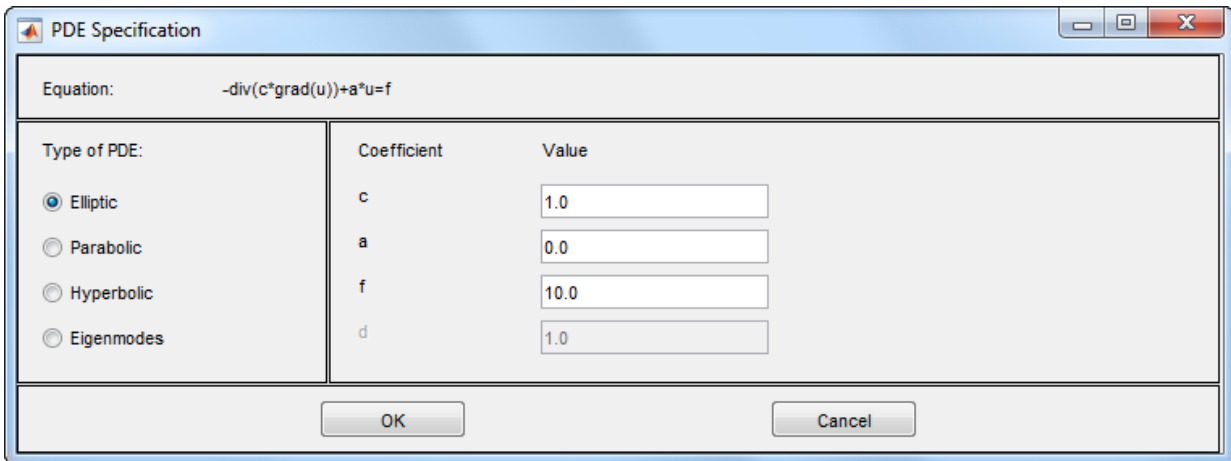
Finally, create the triangular mesh that Partial Differential Equation Toolbox software uses in the Finite Element Method (FEM) to solve the PDE. The triangular mesh is

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.

| File | Edit | Options | Draw | Boundary | PDE | Mesh | Solve | Plot | Window | Help |
|------|------|---------|------|----------|-----|------|-------|------|--------|------|

Set formula: `(R1+C1+R2)-C2`

X: -1.5    Y: 1

Generic Scalar

Info:    Jiggled mesh consists of 347 nodes and 584 triangles.

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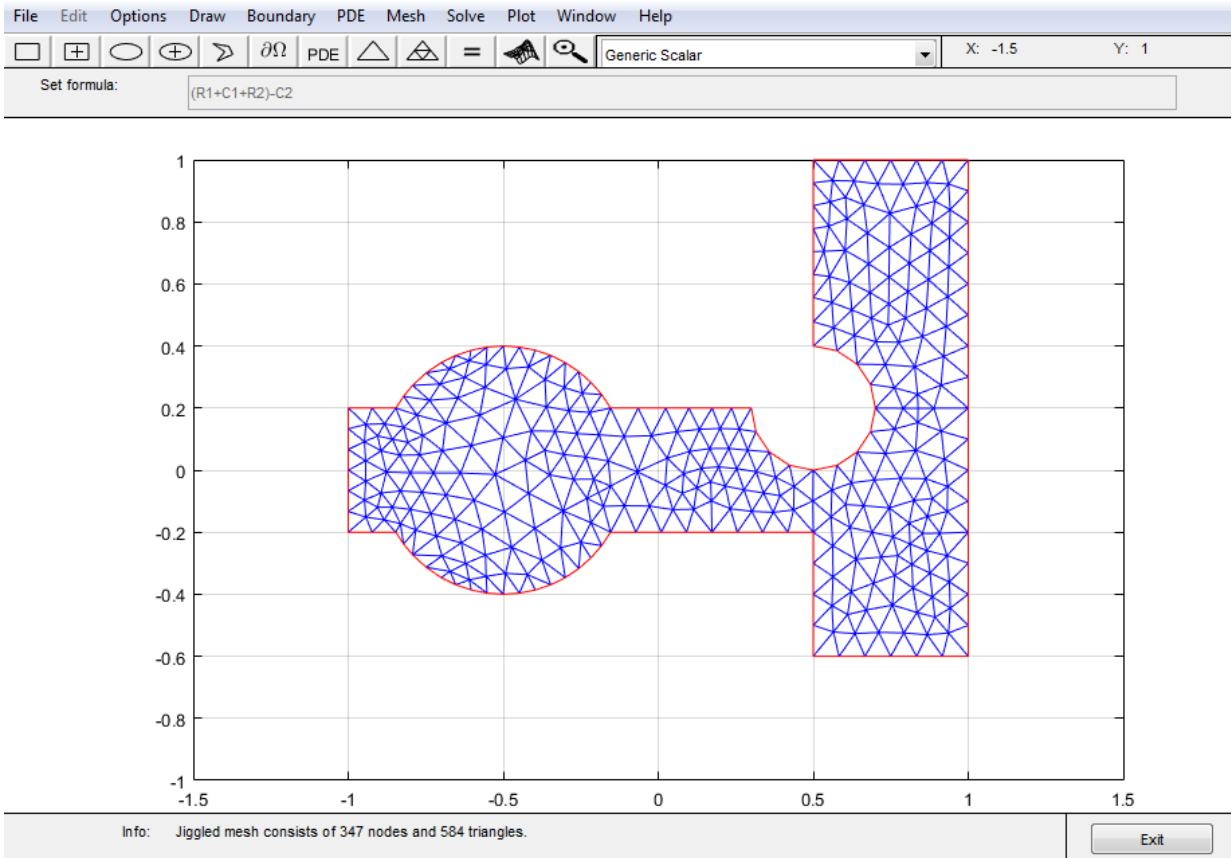
We are now ready to solve the problem. Click the **=** button or select **Solve PDE** from
the **Solve** menu to solve the PDE. The solution is then plotted. By default, the plot uses
interpolated coloring and a linear color map. A color bar is also provided to map the
different shades to the numerical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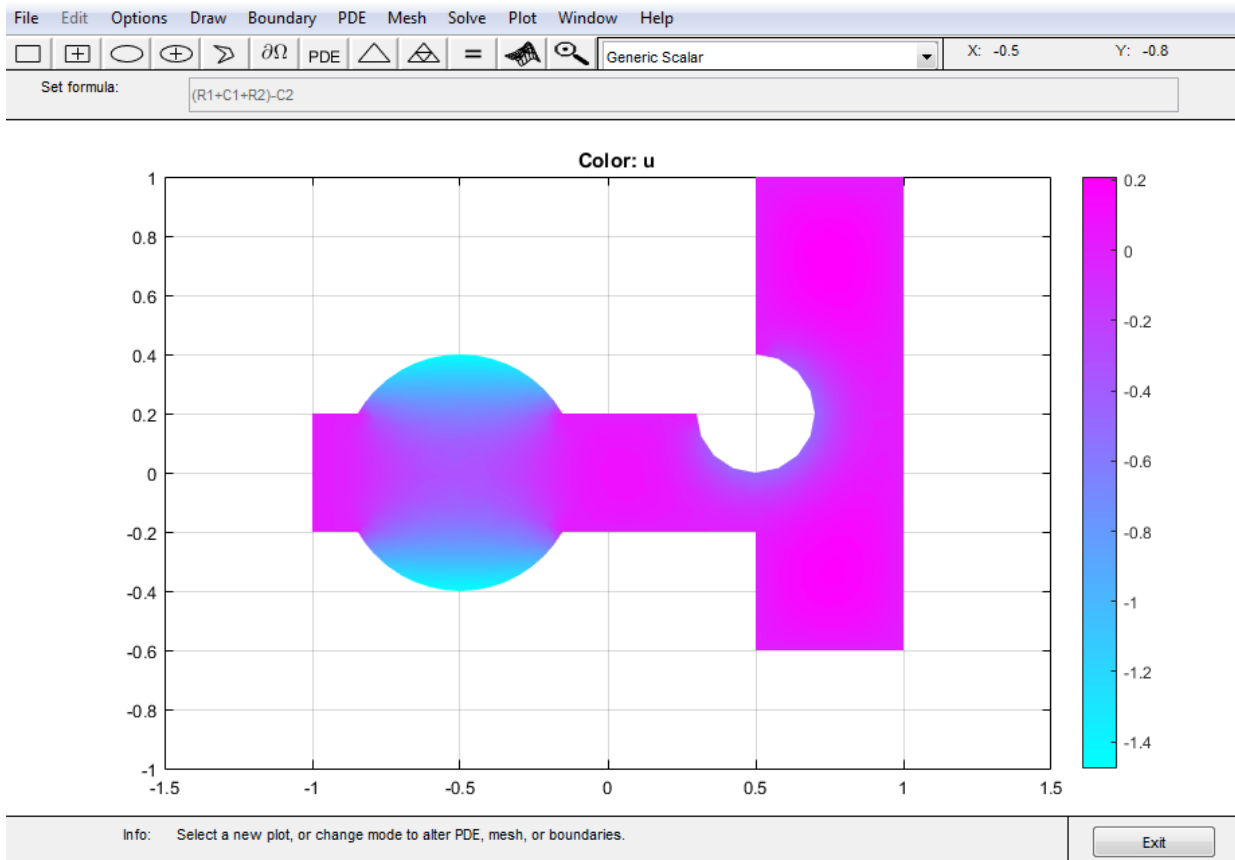
There are many more plot modes available to help you visualize the solution. Click the button with the 3-D solution icon or select **Parameters** from the **Plot** menu to access the dialog box for selection of the different plot options. Several plot styles are available, and the solution can be plotted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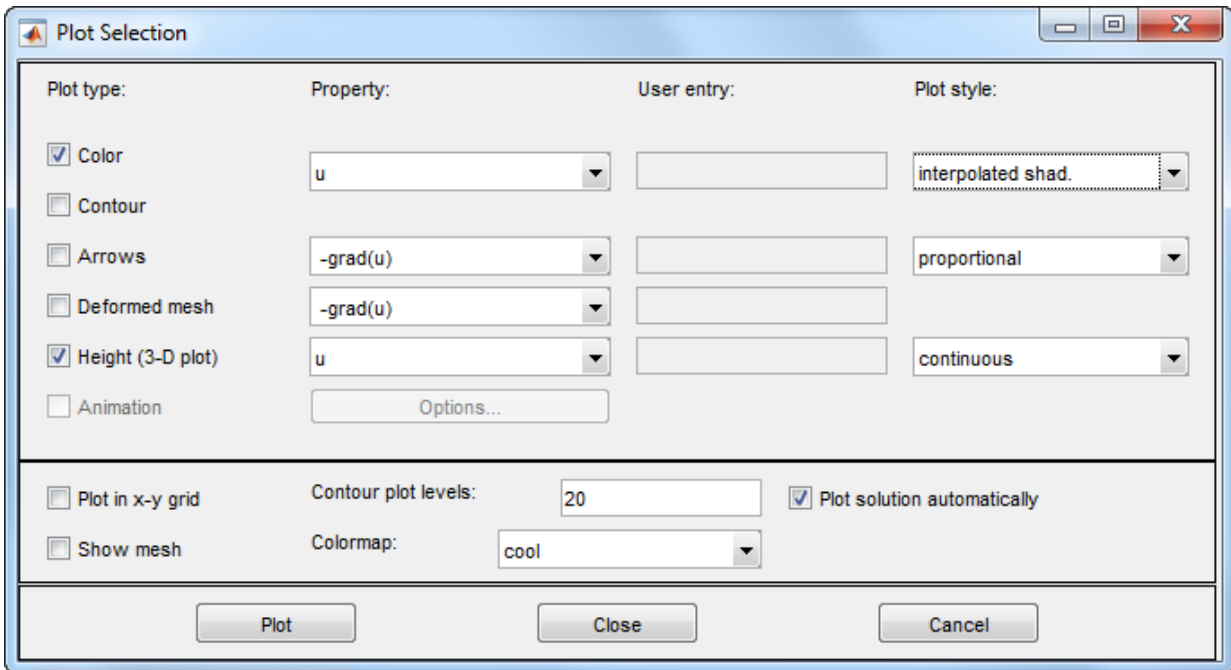
Now, select a plot where the color and the height both represent $u$. Choose interpolated shading and use the continuous (interpolated) height option. The default colormap is the `cool` colormap; a pop-up menu lets you select from a number of different colormaps. Finally, click the **Plot** button to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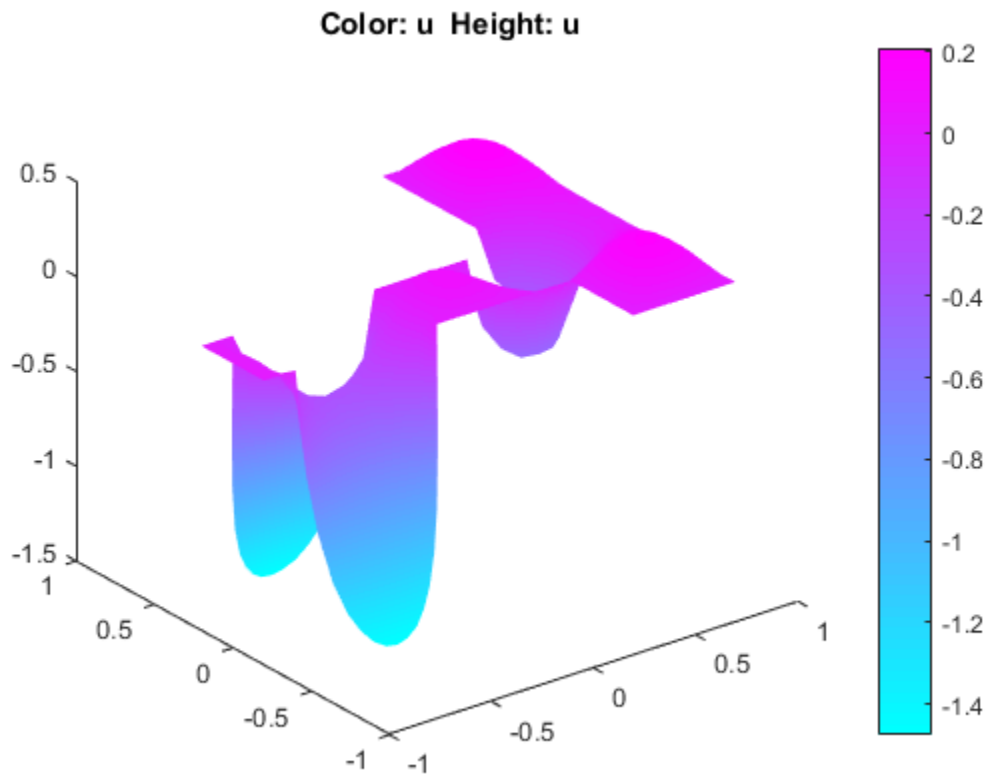
# Finite Element Method (FEM) Basics

The core Partial Differential Equation Toolbox algorithm uses the Finite Element Method (FEM) for problems defined on bounded domains in 2-D or 3-D space. In most cases, elementary functions cannot express the solutions of even simple PDEs on complicated geometries. The finite element method describes a complicated geometry as a collection of subdomains by generating a mesh on the geometry. For example, you can approximate the computational domain $\Omega$ with a union of triangles (2-D geometry) or tetrahedra (3-D geometry). The subdomains form a mesh, and each vertex is called a node. The next step is to approximate the original PDE problem on each subdomain by using simpler equations.

For example, consider the basic elliptic equation.

$$-\nabla \cdot (c\nabla u) + au = f \text{ on domain } \Omega$$

Suppose that this equation is a subject to the Dirichlet boundary condition $u = r$ on $\partial\Omega_D$ and Neumann boundary conditions on $\partial\Omega_N$. Here, $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$ is the boundary of $\Omega$.

The first step in FEM is to convert the original differential (*strong*) form of the PDE into an integral (*weak*) form by multiplying with test function $v$ and integrating over the domain $\Omega$.

$$\int_{\Omega} \left(-\nabla \cdot (c\nabla u) + au - f\right)v \, d\Omega = 0 \quad \forall v$$

The test functions are chosen from a collection of functions (functional space) that vanish on the Dirichlet portion of the boundary, $v = 0$ on $\partial\Omega_D$. Above equation can be thought of as weighted averaging of the residue using all possible weighting functions $v$. The collection of functions that are admissible solutions, $u$, of the weak form of PDE are chosen so that they satisfy the Dirichlet BC, $u = r$ on $\partial\Omega_D$.

Integrating by parts (Green's formula) the second-order term results in:

$$\int_{\Omega} (c\nabla u \nabla v + auv) d\Omega - \int_{\partial\Omega_N} \vec{n} \cdot (c\nabla u) v \, d\partial\Omega_N + \int_{\partial\Omega_D} \vec{n} \cdot (c\nabla u) v \, d\partial\Omega_D = \int_{\Omega} fv \, d\Omega \quad \forall v$$

Use the Neumann boundary condition to substitute for second term on the left side of the equation. Also, note that $v = 0$ on $\partial\Omega_D$ nullifies the third term. The resulting equation is:

$$\int_\Omega (c\nabla u \nabla v + auv)\,d\Omega + \int_{\partial\Omega_N} quv\,d\partial\Omega_N = \int_{\partial\Omega_N} gv\,d\partial\Omega_N + \int_\Omega fv\,d\Omega \quad \forall v$$

Note that all manipulations up to this stage are performed on continuum $\Omega$, the global domain of the problem. Therefore, the collection of admissible functions and trial functions span infinite-dimensional functional spaces. Next step is to discretize the weak form by subdividing $\Omega$ into smaller subdomains or elements $\Omega^e$, where $\Omega = \cup\Omega^e$. This step is equivalent to projection of the weak form of PDEs onto a finite-dimensional subspace. Using the notations $u_h$ and $v_h$ to represent the finite-dimensional equivalent of admissible and trial functions defined on $\Omega^e$, you can write the discretized weak form of the PDE as:

$$\int_{\Omega^e} (c\nabla u_h \nabla v_h + au_h v_h)\,d\Omega^e + \int_{\partial\Omega_N^e} qu_h v_h\,d\partial\Omega_N^e = \int_{\partial\Omega_N^e} gv_h\,d\partial\Omega_N^e + \int_{\Omega^e} fv_h\,d\Omega^e \quad \forall v_h$$

Next, let $\phi_i$, with $i = 1, 2, \ldots, N_p$, be the piecewise polynomial basis functions for the subspace containing the collections $u_h$ and $v_h$, then any particular $u_h$ can be expressed as a linear combination of basis functions:

$$u_h = \sum_1^{N_p} U_i \phi_i$$

Here $U_i$ are yet undetermined scalar coefficients. Substituting $u_h$ into to the discretized weak form of PDE and using each $v_h = \varphi_i$ as test functions and performing integration over element yields a system of $N_p$ equations in terms of $N_p$ unknowns $U_i$.

Note that finite element method approximates a solution by minimizing the associated error function. The minimizing process automatically finds the linear combination of basis functions which is closest to the solution $u$.

FEM yields a system $KU = F$ where the matrix $K$ and the right side $F$ contain integrals in terms of the test functions $\phi_i$, $\phi_j$, and the coefficients $c$, $a$, $f$, $q$, and $g$ defining the

problem. The solution vector $U$ contains the expansion coefficients of $u_h$, which are also the values of $u_h$ at each node $x_k$ ($k = 1,2$ for a 2-D problem or $k = 1,2,3$ for a 3-D problem) since $u_h(x_k) = U_i$.

FEM techniques are also used to solve more general problems, such as:

- Time-dependent problems. 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

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

The result is a system of ordinary differential equations (ODEs)

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

Two time derivatives result in a second-order ODE

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

- Eigenvalue problems. Solve

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

for the unknowns $u$ and $\lambda$, where $\lambda$ is a complex number. Using the FEM discretization, you solve the algebraic eigenvalue problem $KU = \lambda MU$ to find $u_h$ as an approximation to $u$. To solve eigenvalue problems, use `solvepdeeig`.

- Nonlinear problems. If the coefficients $c$, $a$, $f$, $q$, or $g$ are functions of $u$ or $\nabla u$, the PDE is called nonlinear and FEM yields a nonlinear system $K(U)U = F(U)$.

To summarize, the FEM approach:

1 Represents the original domain of the problem as a collection of elements.

2 For each element, substitutes the original PDE problem by a set of simple equations that locally approximate the original equations. Applies boundary conditions for boundaries of each element. For stationary linear problems where the coefficients do not depend on the solution or its gradient, the result is a linear system of equations. For stationary problems where the coefficients depend on the solution or its gradient, the result is a system of nonlinear equations. For time-dependent problems, the result is a set of ODEs.

3 Assembles the resulting equations and boundary conditions into a global system of equations that models the entire problem.

4 Solves the resulting system of algebraic equations or ODEs using linear solvers or numerical integration, respectively. The toolbox internally calls appropriate MATLAB solvers for this task.

## References

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

[2] Gilbert Strang and George Fix. *An Analysis of the Finite Element Method*. 2nd edition. Wellesley, MA: Wellesley-Cambridge Press, 2008.

## See Also

assembleFEMatrices | solvepde | solvepdeeig

# 2

# Setting Up Your PDE

# 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 2-5.

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 2-51.

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 2-36 or "Create 2-D Geometry" on page 2-7. 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 2-36. For a 2-D example, see "Identify Boundary Labels" on page 2-152. 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" on page 2-160. 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".

6 Create the PDE coefficients. For example:

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

- You can specify coefficients as numeric, character 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 2-66.

- For systems of PDEs, each coefficient f, c, a, and d has a specific format. See "f Coefficient for Systems" on page 2-83, "c Coefficient for Systems" on page 2-110, and "a or d Coefficient for Systems" on page 2-133.

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 2-147.

**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 3-210 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 2-3.

**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 2-51.

**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 2-36 or "Create 2-D Geometry" on page 2-7.

```
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.

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

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 2-36.

**5** Create the boundary conditions. For details, see "Specify Boundary Conditions" on page 2-160.

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

For more information on boundary conditions, see "Boundary Conditions".

**6** Create the PDE coefficients.

```
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 2-86, "c Coefficient for specifyCoefficients" on page 2-89, and "m, d, or a Coefficient for specifyCoefficients" on page 2-128.

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 2-140.

**8** Create the mesh.

```
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 2-D Solutions and Their Gradients" on page 3-199 and "Plot 3-D Solutions and Their Gradients" on page 3-210.

## See Also

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

## Related Examples

- "Plot 3-D Solutions and Their Gradients" on page 3-210

# 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 "Create 2-D Geometry in the PDE App" on page 4-3 and "Rounded Corners Using CSG Modeling" on page 4-11.

- 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 2-9.

- Specify a function that describes the geometry. The function must be in the form described in "Create Geometry Using a Geometry Function" on page 2-16.

## 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 2-7

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

1 "Create Basic Shapes" on page 2-9 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 2-11 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 2-12 A set of characters 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) |

### 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 set of characters 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 Face Boundaries

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 face boundaries. Completing the example,

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

View the geometry with and without boundary removal.

```
pdegplot(dl,'EdgeLabels','on','FaceLabels','on')
xlim([-1.5,1.5])
axis equal
```
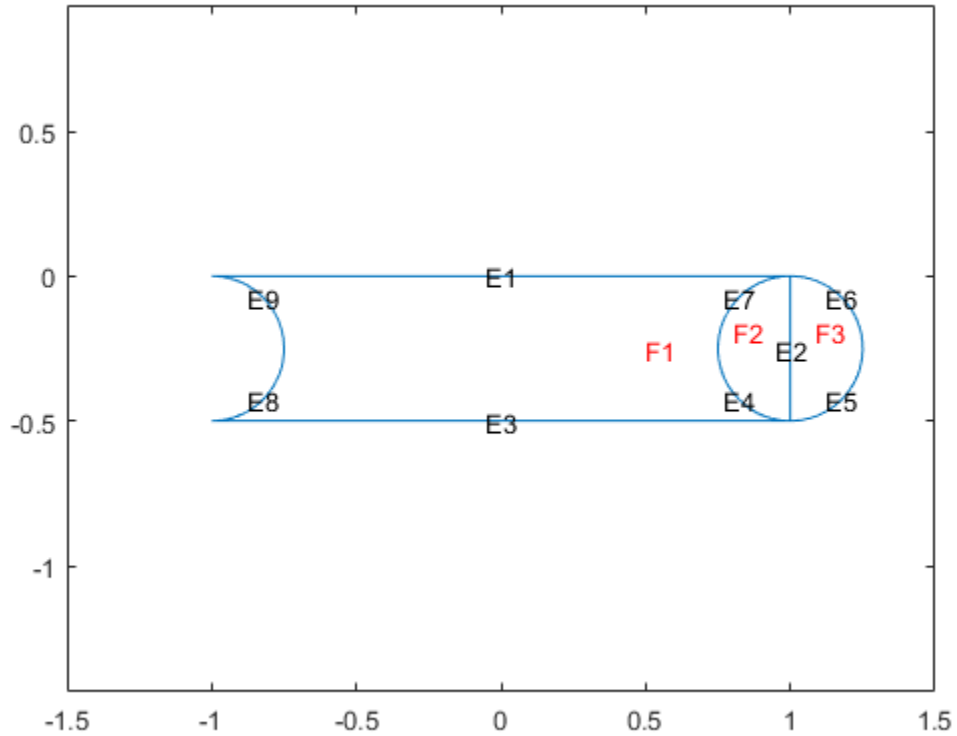
Remove the face boundaries.

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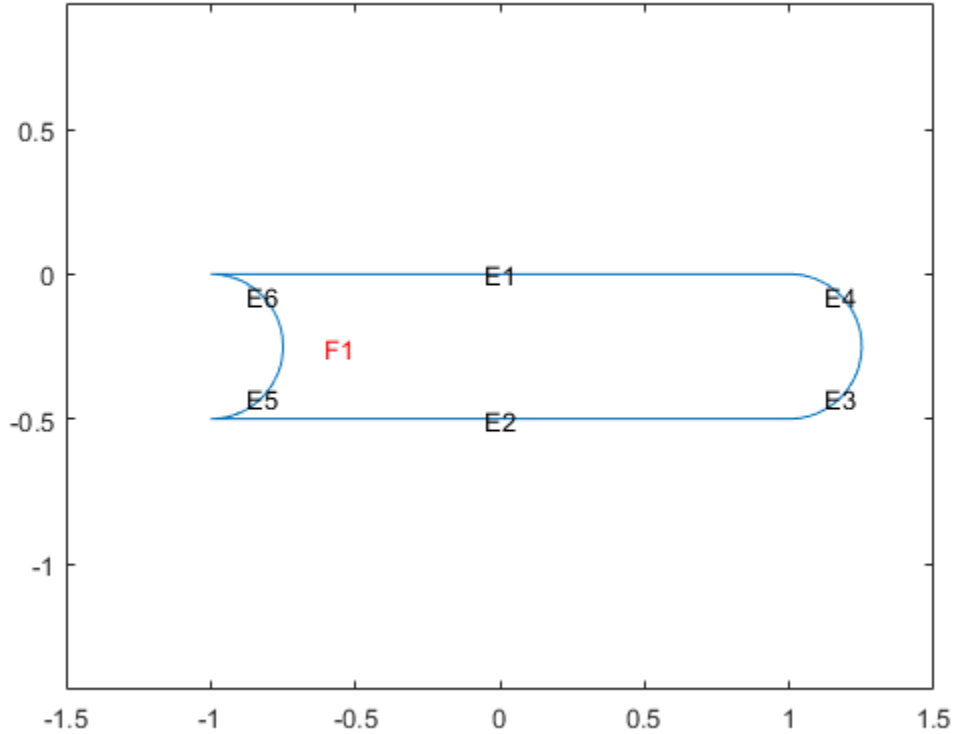
## Decomposed Geometry Data Structure

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 2-7

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:<br><br>**1**  Start parameter value<br>**2**  End parameter value<br>**3**  Left region label, where "left" is with respect to the direction from the start to end parameter value<br>**4**  Right region label<br><br>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.

```matlab
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])
```
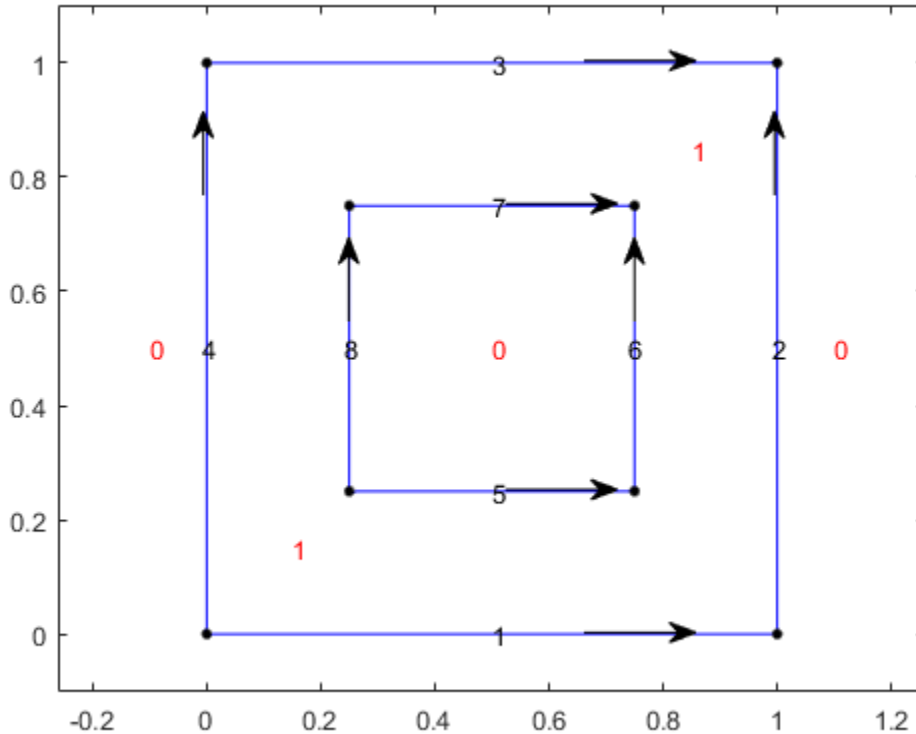
## Geometry Function for a Circle

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:

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

A geometry function needs to have at least two segments. So break up the circle into four segments:

- $0 \leq t \leq \pi/2$

- $\pi/2 \leq t \leq \pi$

- $\pi \leq t \leq 3\pi/2$

- $3\pi/2 \leq t \leq 2\pi$

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
```
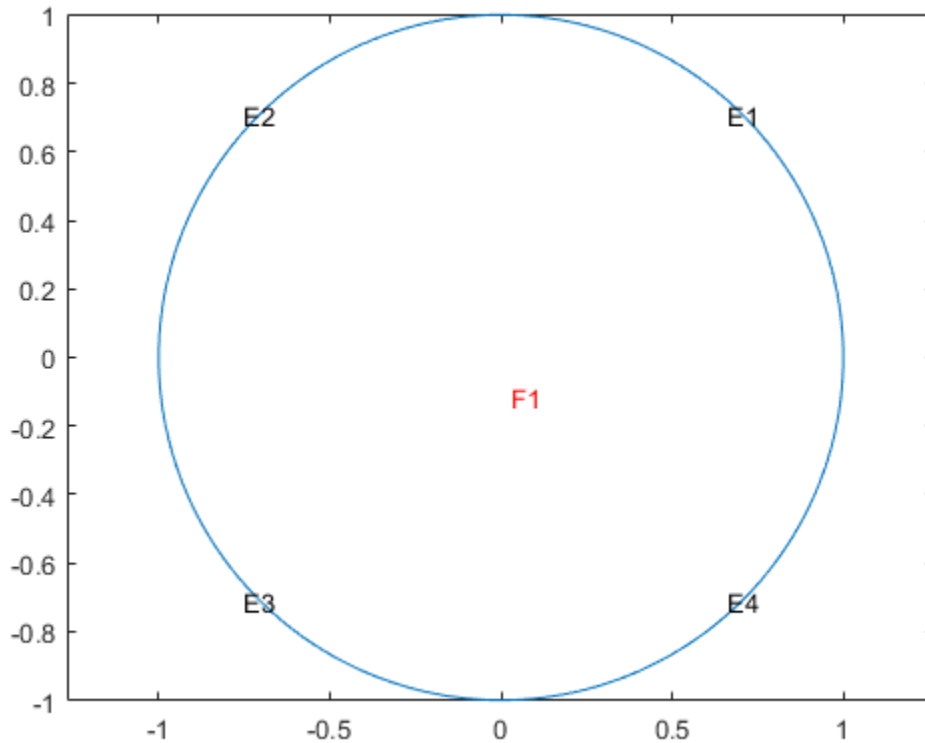
The built-in function `circleg` gives a slightly different parameterization of the circle. You might find it instructive to compare the two approaches.

## 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))')
```

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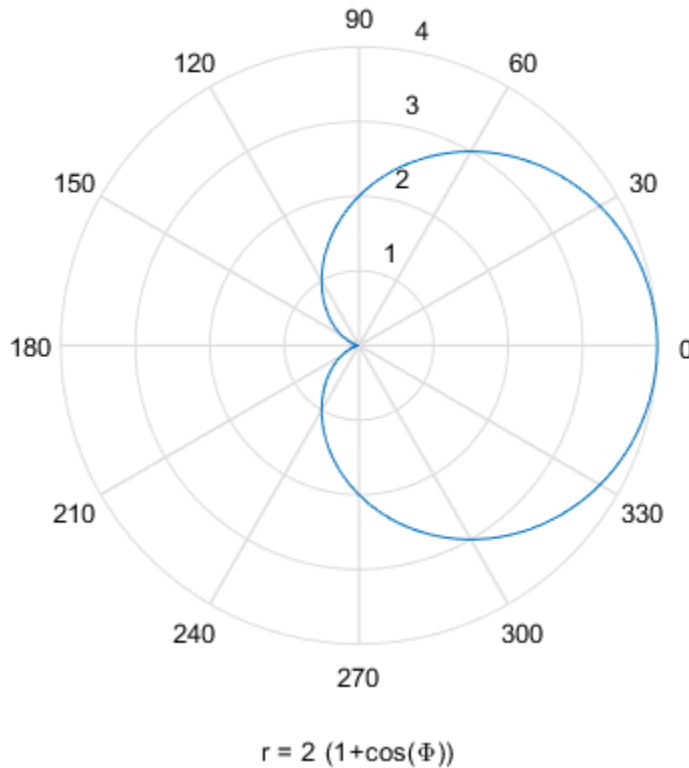
This example shows four approaches to giving a parametrization of the cardioid as a function of arc length.

- 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 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);
end
```

Plot the geometry function.

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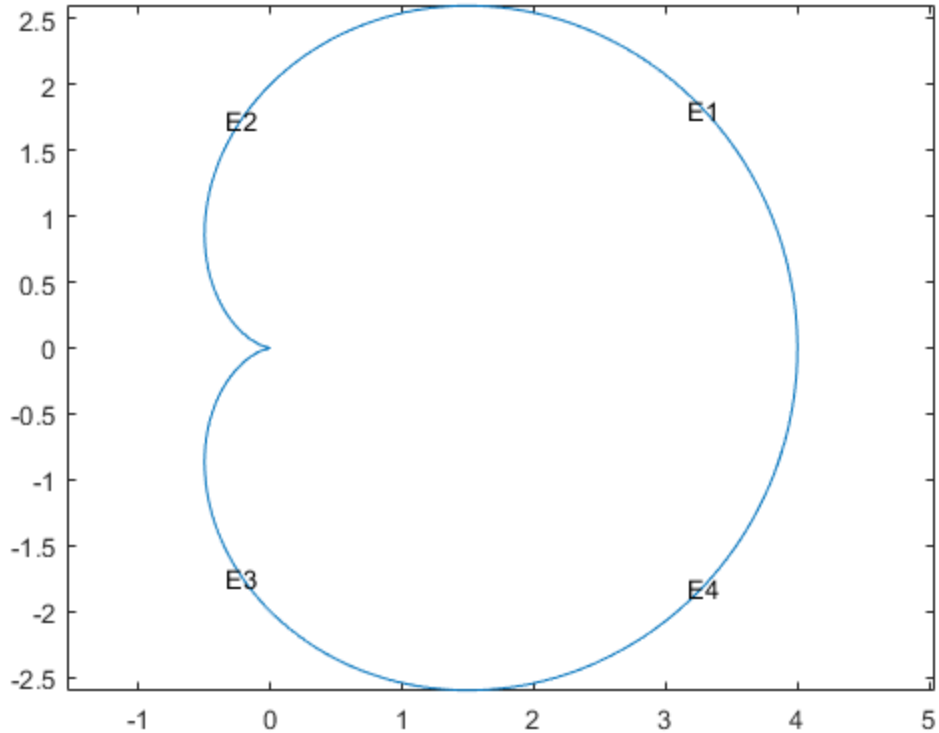


With 400 line segments, the geometry looks smooth.

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

$$s\left(t\right) = \int_0^t \sqrt{\left(\frac{dx}{du}\right)^2 + \left(\frac{dy}{du}\right)^2}\,du.$$

So for a given value $s0$, you can find $t$ as the root of the equation $s\left(t\right) = s0$. 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
end
```

Plot the geometry function.

```
pdegplot('cardioid2','EdgeLabels','on')
axis equal
```
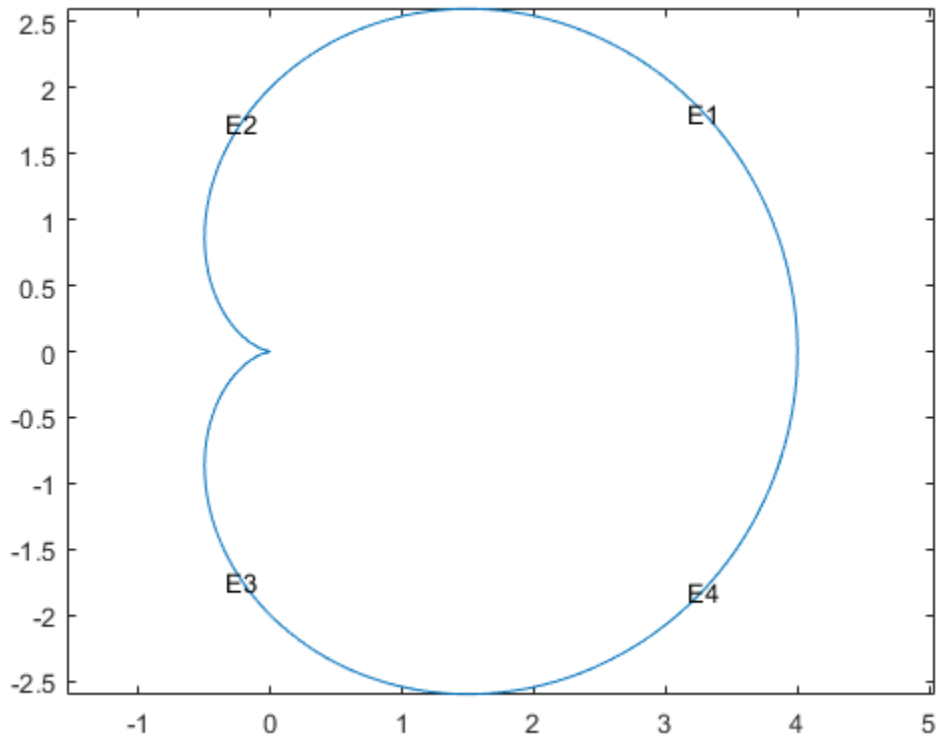


The geometry looks identical to the polygonal approximation. This integral version takes much longer to calculate than the polygonal version.

### 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 π. For s between 8 and 16, by symmetry if 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
end
```

Plot the geometry function.

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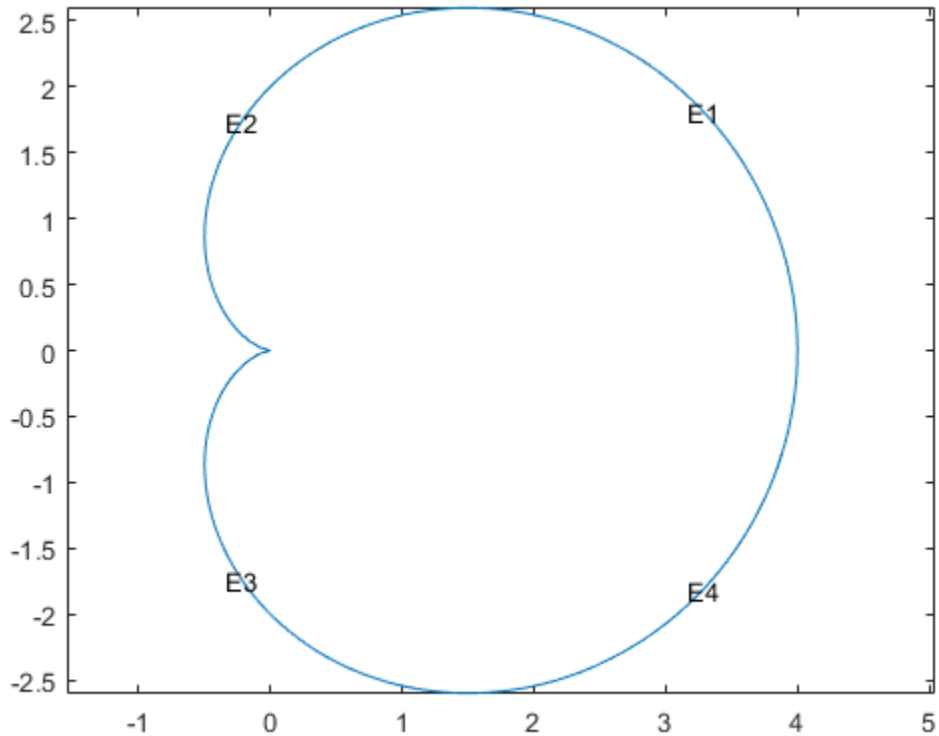
This analytic geometry looks slightly smoother than the previous versions. However, the difference is inconsequential in terms of calculations.

### 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);
      end
```

Plot the geometry function.

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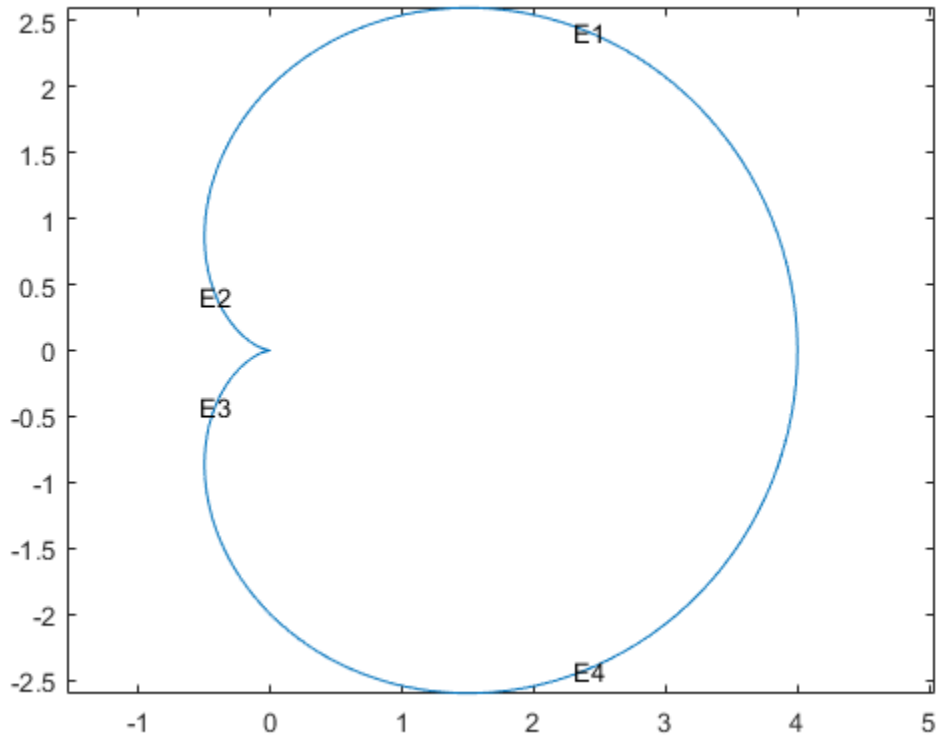
The labels are not evenly spaced on the edges because the parameter is not proportional to arc length.

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
```
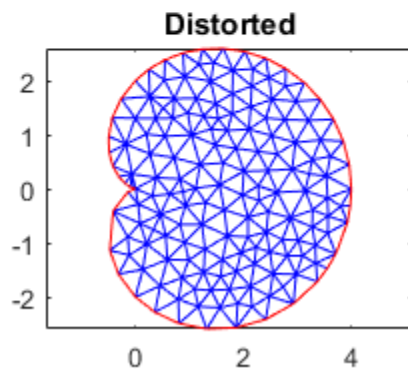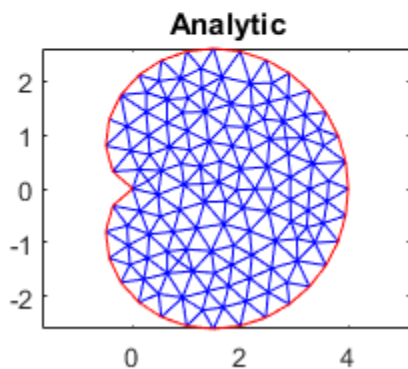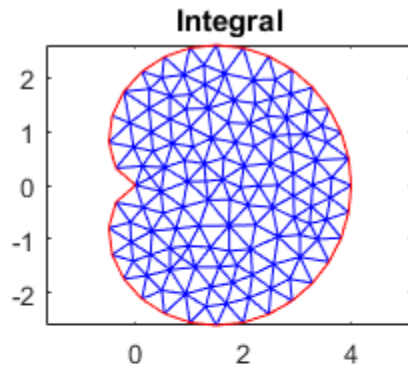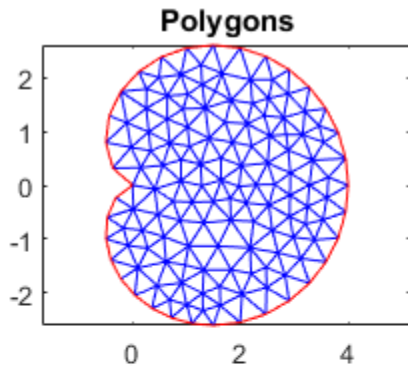
While the "Distorted" mesh looks a bit less regular than the other meshes (it has some very narrow triangles near the cusp of the cardioid), all of the meshes appear to be usable.

## 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" and a variant of the circle function from "Geometry Function for a Circle".

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));
end
```

Plot the geometry, including edge labels and subdomain labels.

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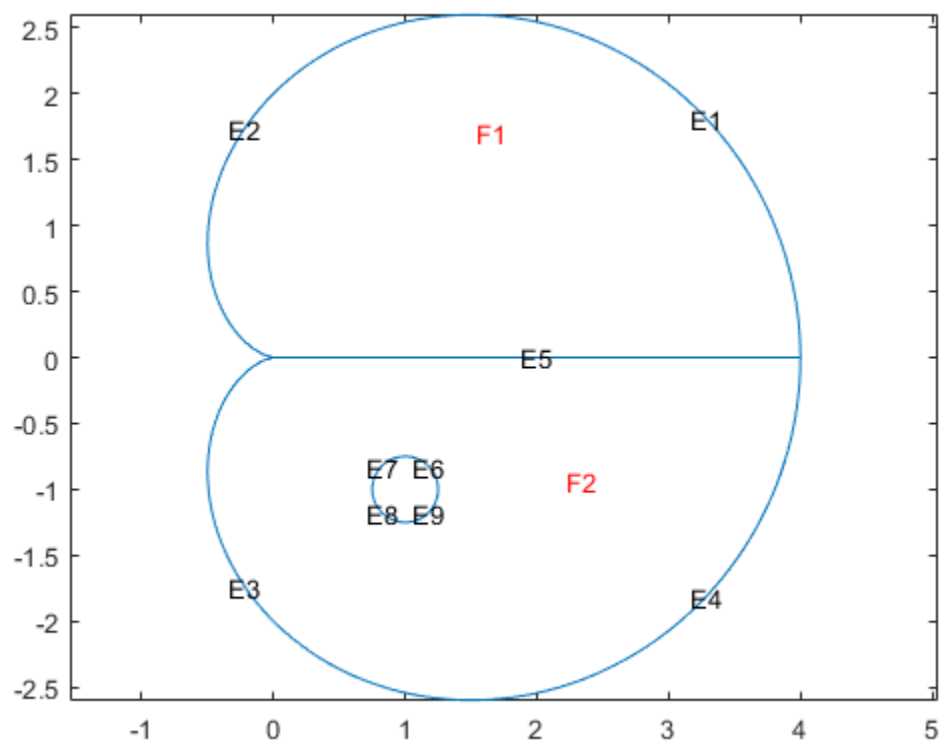
# Create and View 3-D Geometry

---
**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')
```
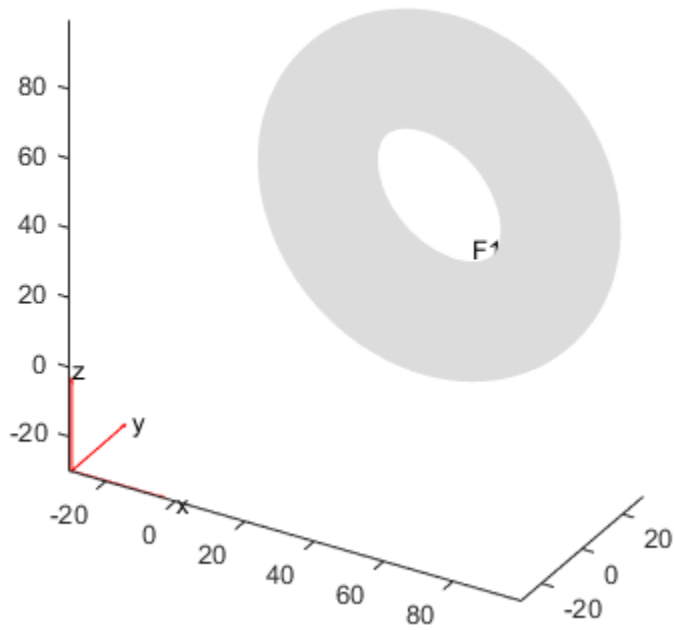
```
model = createpde;
importGeometry(model,'Block.stl');
pdegplot(model,'FaceLabels','on')
```

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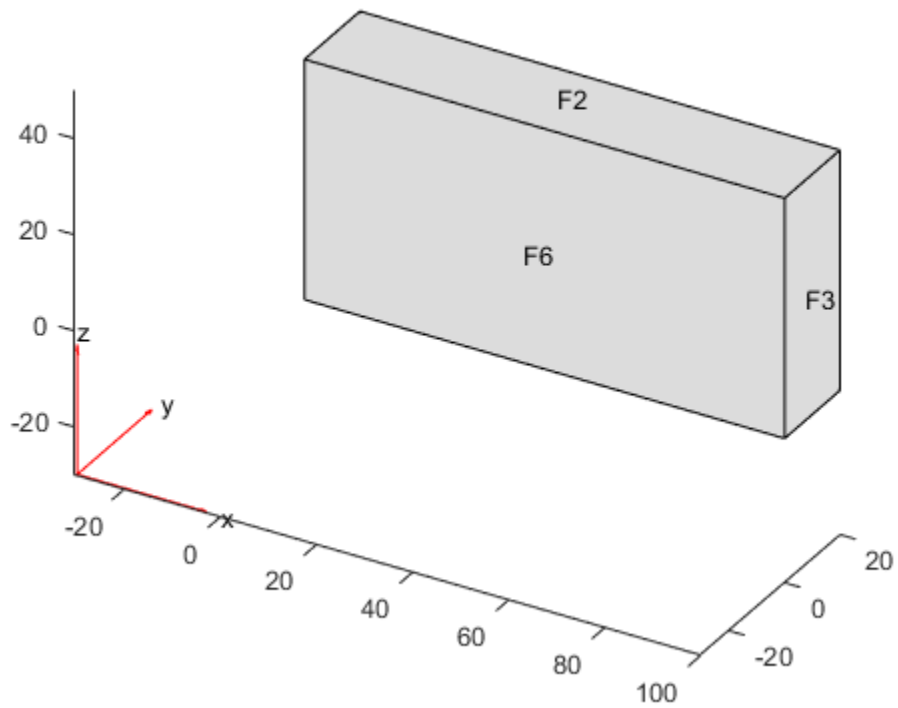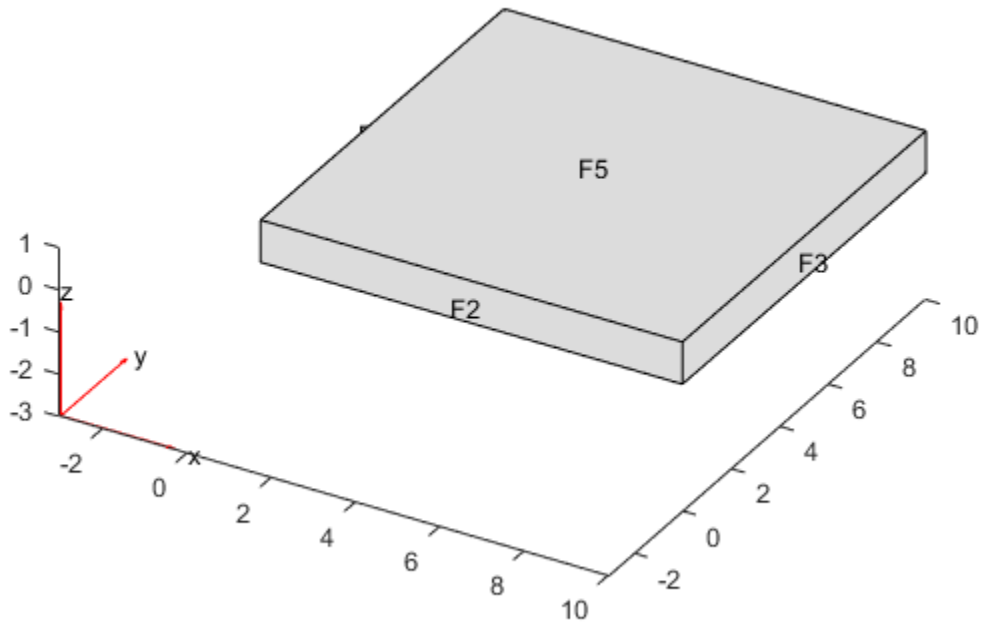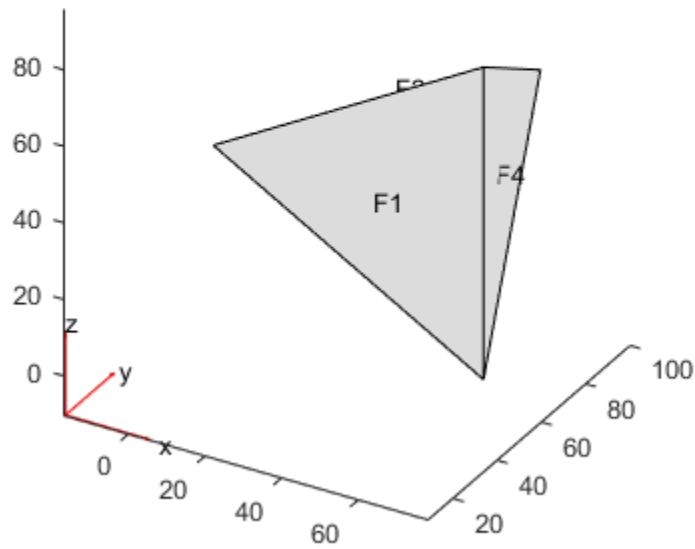
```
model = createpde;
importGeometry(model,'BracketTwoHoles.stl');
pdegplot(model,'FaceLabels','on')
```
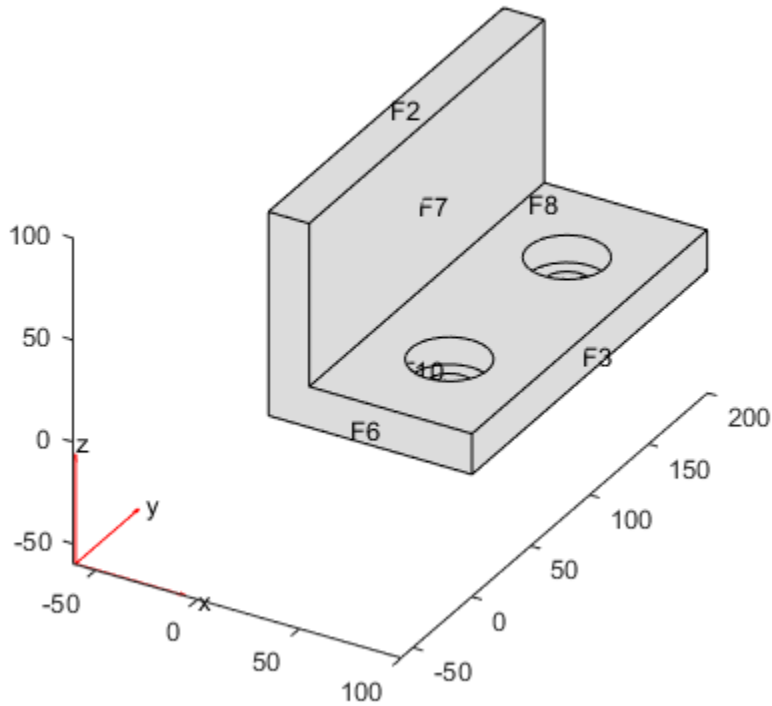
To see hidden portions of the geometry, rotate the figure using the **Rotate 3D** button
. You can rotate the angle bracket to obtain the following view.
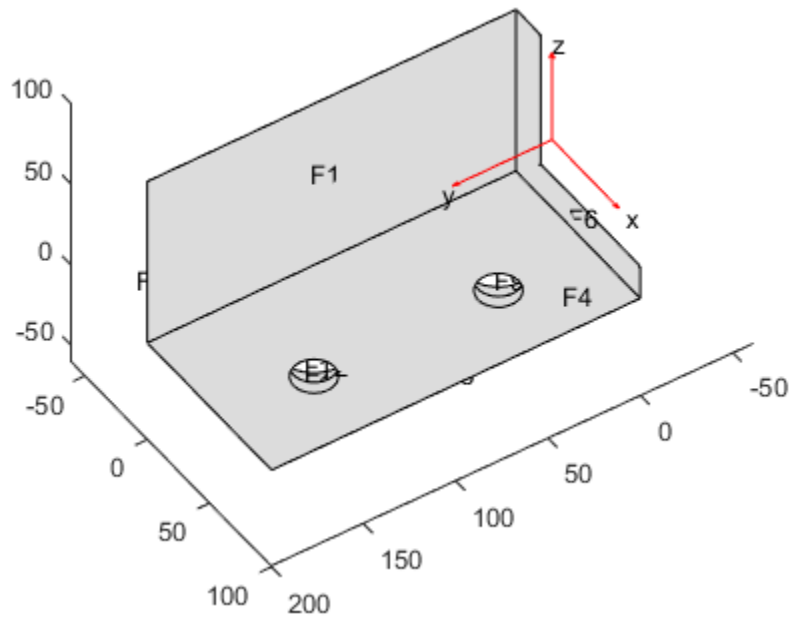
```
model = createpde;
importGeometry(model,'ForearmLink.stl');
pdegplot(model,'FaceLabels','on');
```
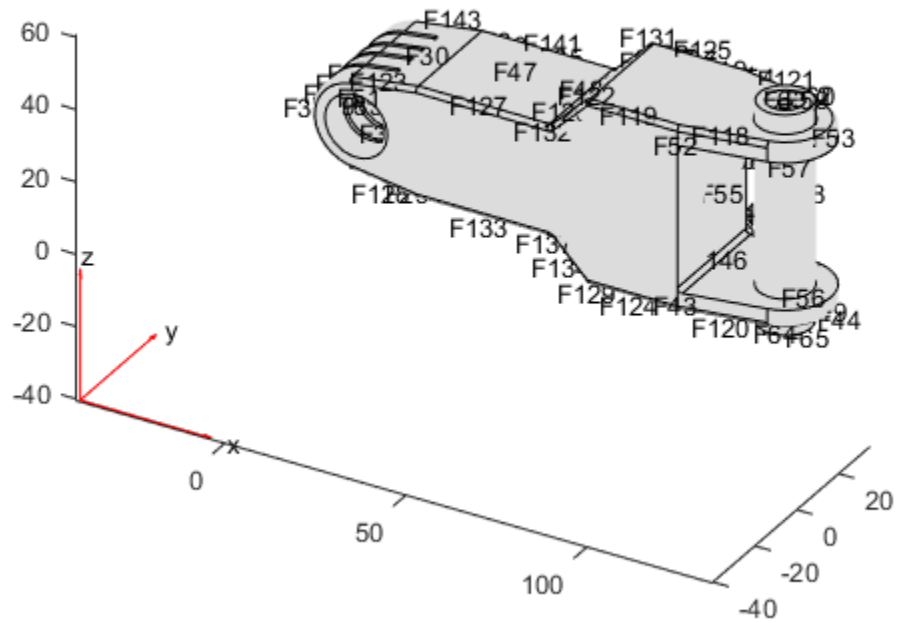
To view hidden faces, set `FaceAlpha` to a value less than 1, such as 0.5.

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
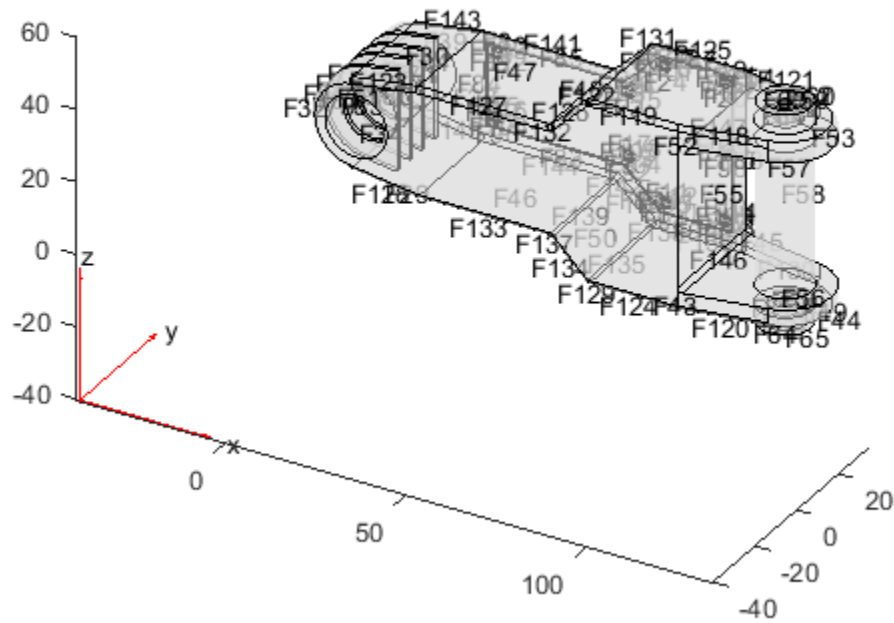
## 3-D Geometry from a Finite Element Mesh

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: [3×1456 double]
          Elements: [4×4969 double]
    MaxElementSize: 8.2971
    MinElementSize: 1.9044
    GeometricOrder: 'linear'
```

View the geometry and face numbers.

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
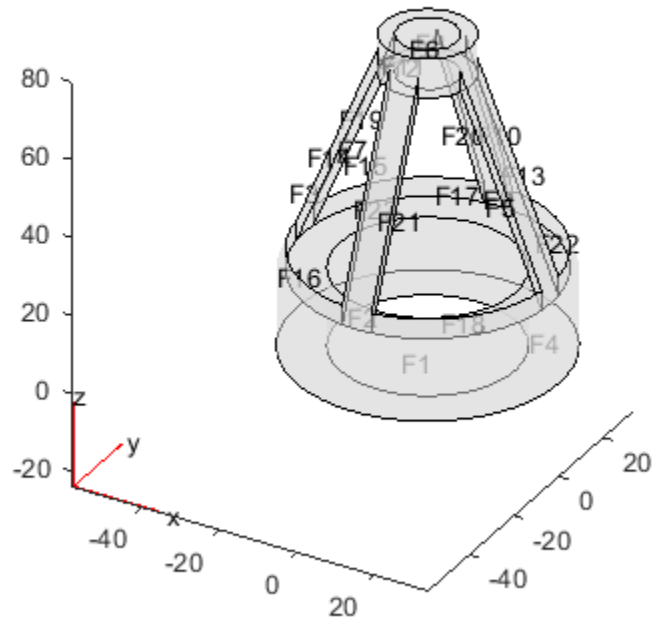
## 3-D Geometry from Point Cloud

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.

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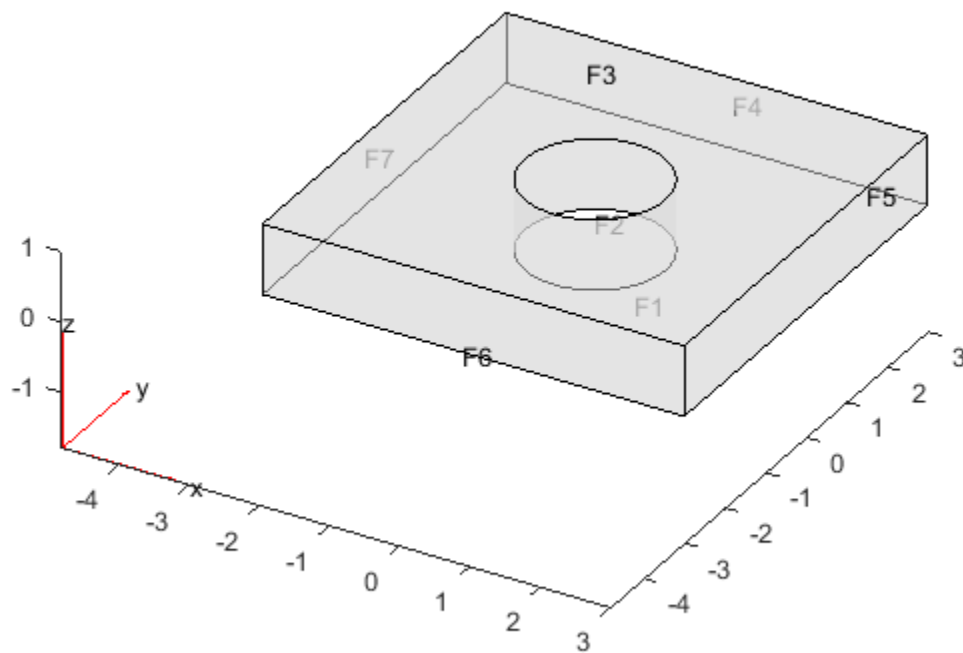
To use the geometry in an analysis, create a volume mesh.

```
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 3-214. |

## Related Examples

- "Solve Problems Using PDEModel Objects" on page 2-5

# Put Equations in Divergence Form

## Coefficient Matching for Divergence Form

As explained in "Equations You Can Solve Using Recommended Functions" on page 1-6, 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

$$c_1 = -(1 + x^2)$$
$$c_4 = -(1 + y^2)/2$$

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,

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

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

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

## Related Examples

# Specify Scalar PDE Coefficients in Character 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 vector of characters 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 character 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 vector of characters — 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 use spaces when specifying coefficients in the PDE app. The 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 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 2-72

# 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 character expression as described in "Specify Scalar PDE Coefficients in Character Form" on page 2-55. 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 2-62.

- p and t are the node points and triangles of the mesh. For a description of these data structures, see "Mesh Data" on page 2-196. 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 character form, t represents time, and triangles do not enter into the form.

---

Pass the coefficient function to the solver as 'coeffunction' or as a function handle @coeffunction. In the PDE app, pass the coefficient as coeffunction without quotes, because the PDE app interprets all entries as characters.

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 character expression, as described in "Specify Scalar PDE Coefficients in Character Form" on page 2-55. 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 2-133).
  - `c` — NC can have many different values in the range 1 to $9N^2$ (see "c Coefficient for Systems" on page 2-110).

- 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 character 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.

```matlab
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
```
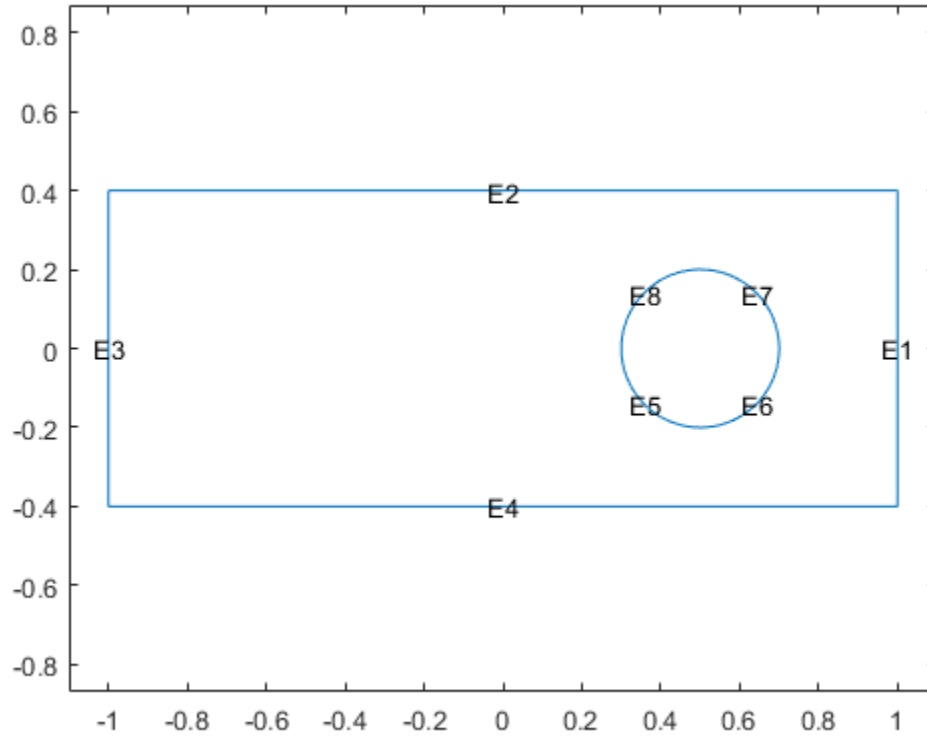
## PDE Coefficients

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:

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

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

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;
end
```

## 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". 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);

111 successful steps
9 failed attempts
242 function evaluations
1 partial derivatives
32 LU decompositions
241 solutions of linear systems
```

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
```

Step 50
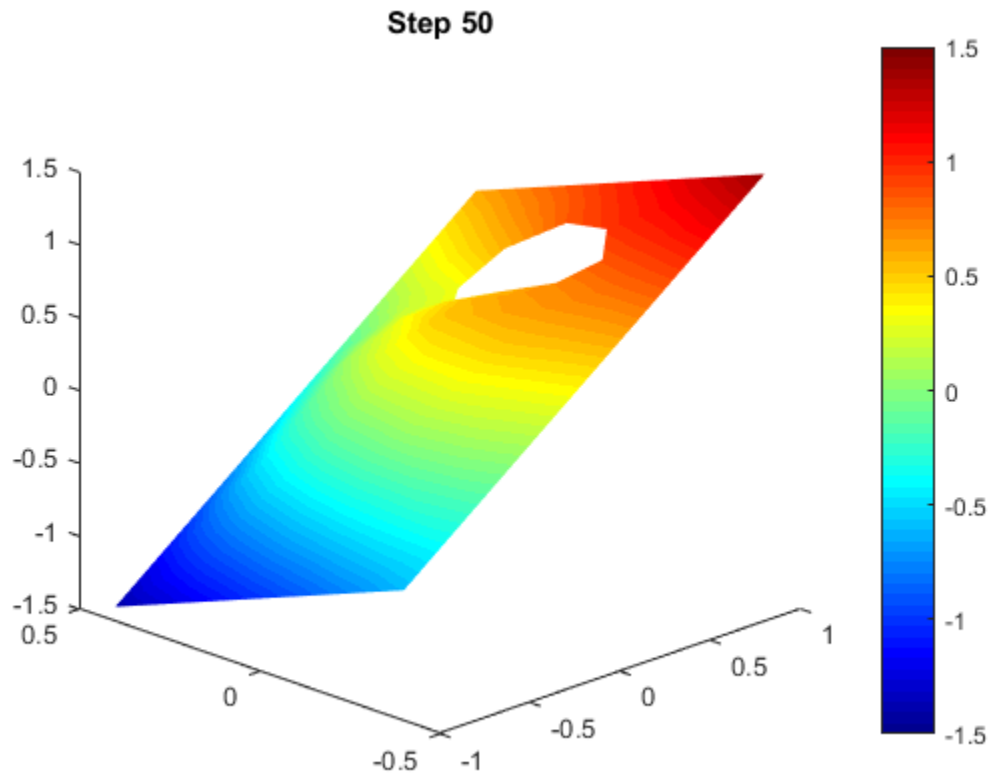
## Alternative Coefficient Syntax

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

```
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;
end
```

Call this function by setting

```
f = @framp2;
u = parabolic(u0,tlist,model,c,a,f,d);

111 successful steps
9 failed attempts
242 function evaluations
1 partial derivatives
32 LU decompositions
241 solutions of linear systems
```

You can also write a function for the coefficient c, though it is more complicated than the character 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;
end
```

Call this function by setting

```
c = @cfunc;
u = parabolic(u0,tlist,model,c,a,f,d);

111 successful steps
9 failed attempts
242 function evaluations
1 partial derivatives
32 LU decompositions
241 solutions of linear systems
```

# Enter Coefficients in the PDE App

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

Caution: Do not include spaces when you specify your coefficients the PDE app. The 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 vectors of characters. Therefore, do not include quotes for the `c` or `f` coefficients.

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

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

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

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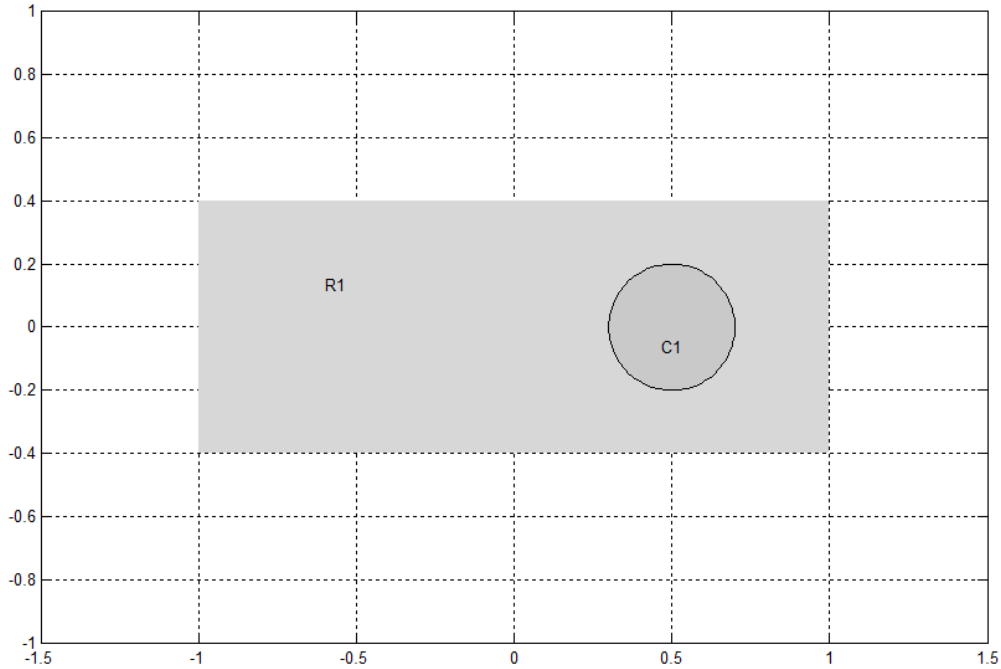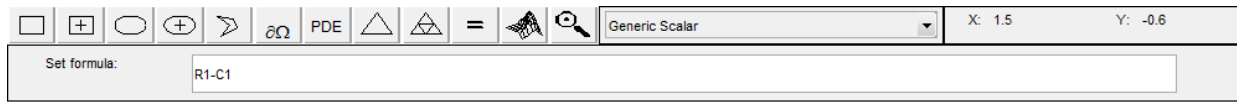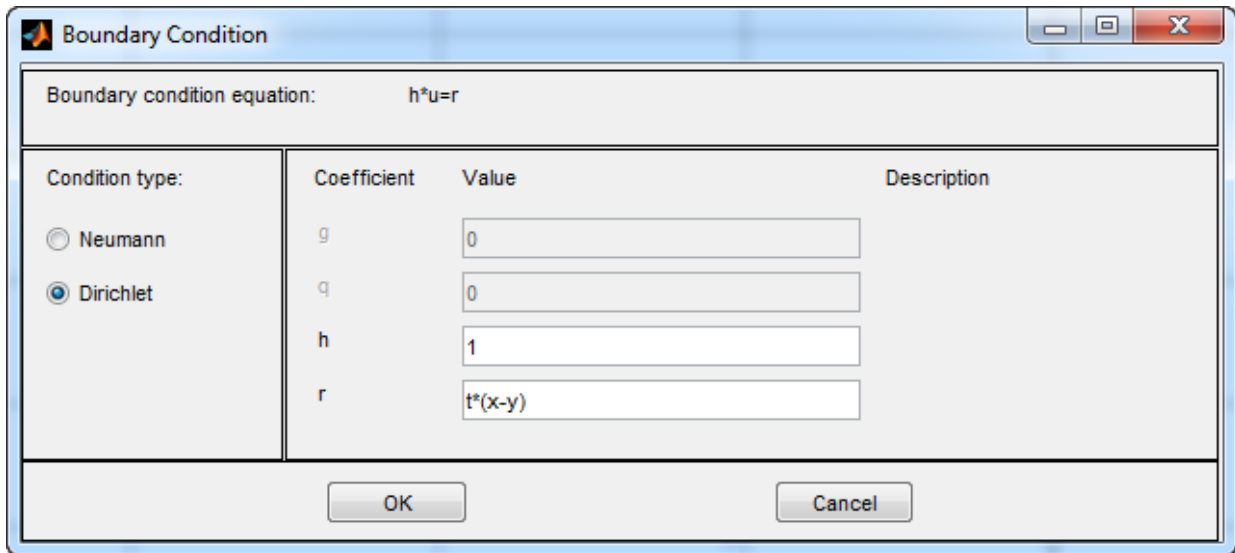
Select **Boundary > Boundary Mode**

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

Double-click one of the selected segments.

Fill in the resulting dialog box as pictured, with Dirichlet boundary conditions **h** = 1 and **r** = t*(x-y). Click **OK**.
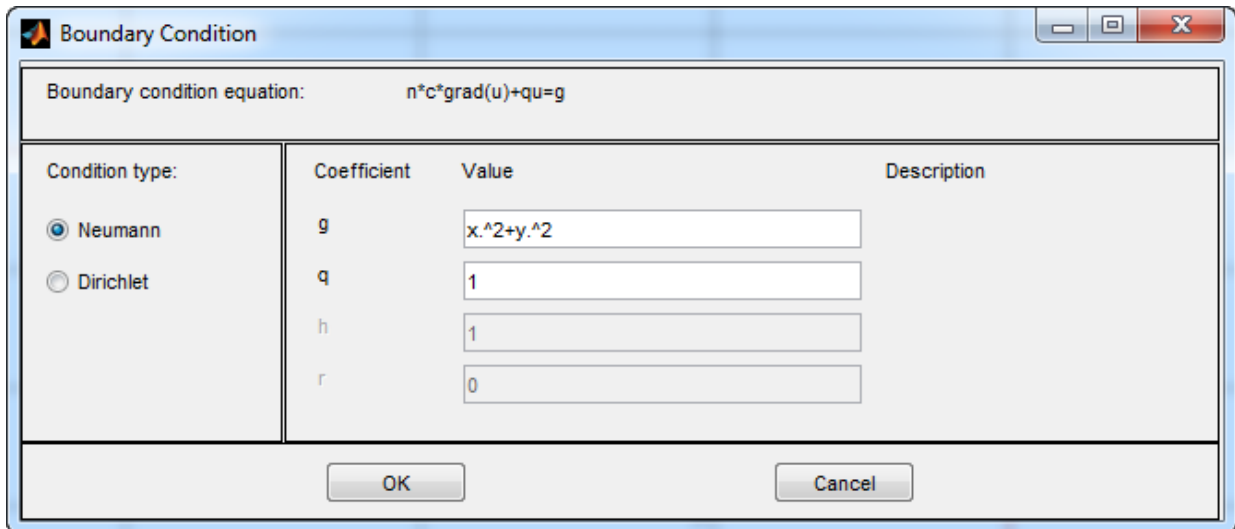
Select the four segments of the inner circle using **Shift**-click, and double-click one of the segments.

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

Click △ to initialize the mesh.

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

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

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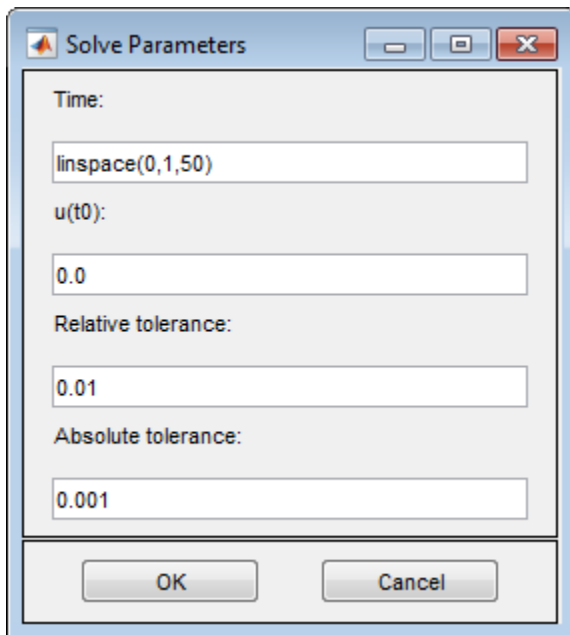


Solve and plot the equation by clicking the ═ button.

Match the following figure using **Plot > Parameters**.

Click the **Plot** button.

Time=1   Color: u   Height: u
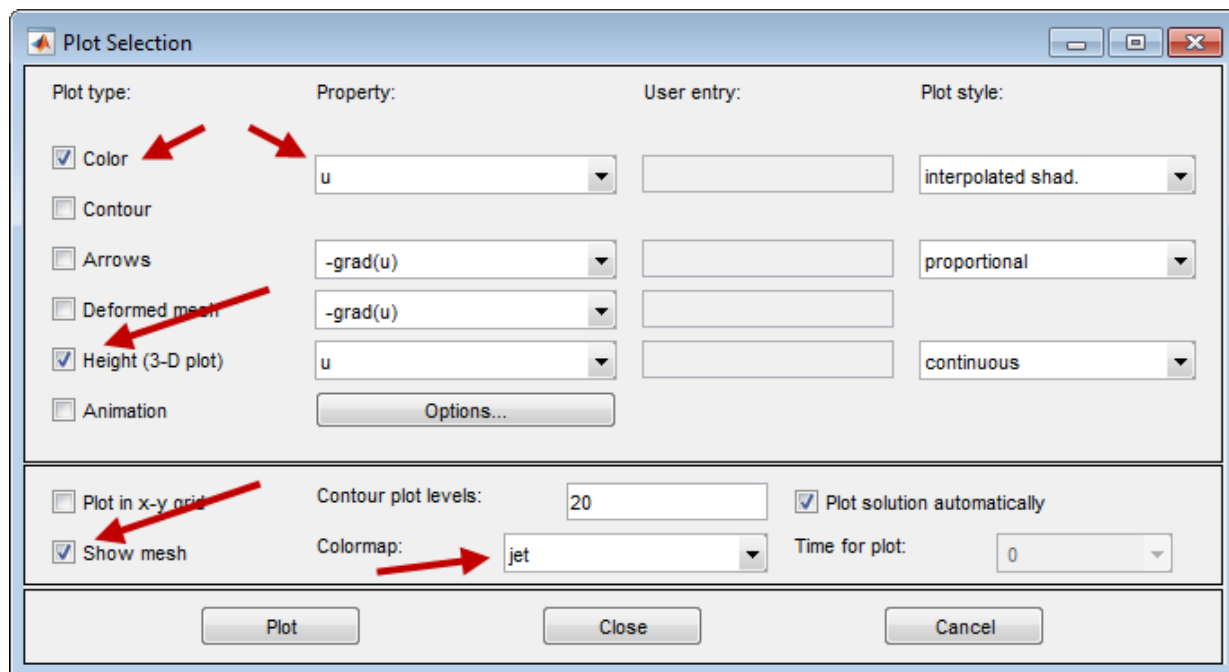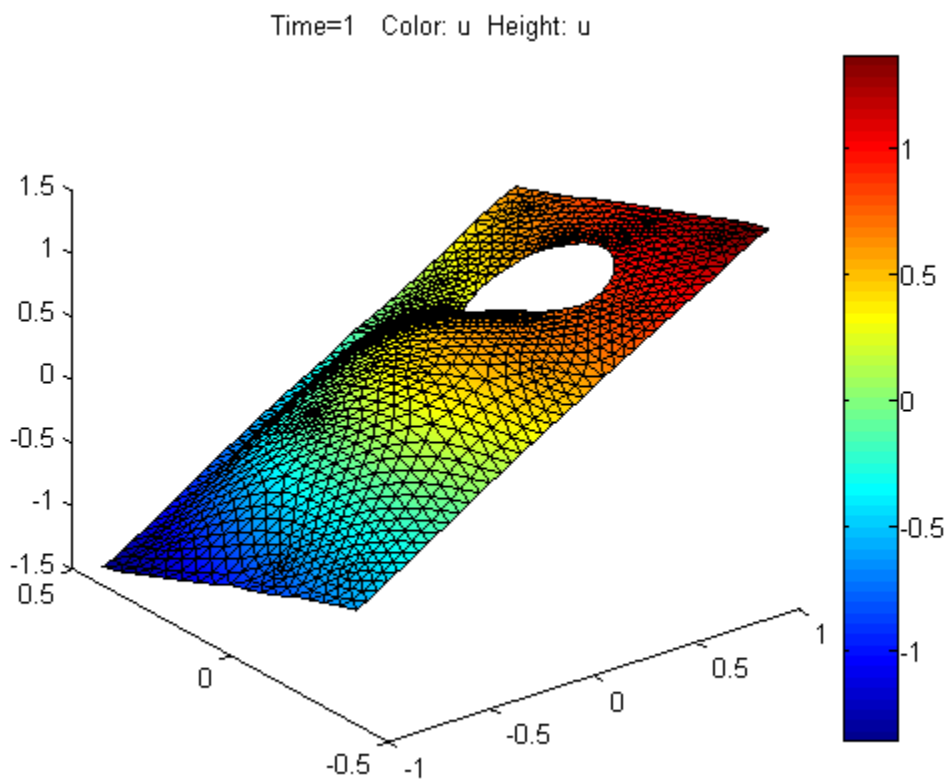
## Related Examples

- "Coefficients for Scalar PDEs in PDE App" on page 2-58

# Systems in the PDE App

You can enter coefficients for a system with $N = 2$ equations in the PDE app. To do so, open the PDE app and select `Generic System`.
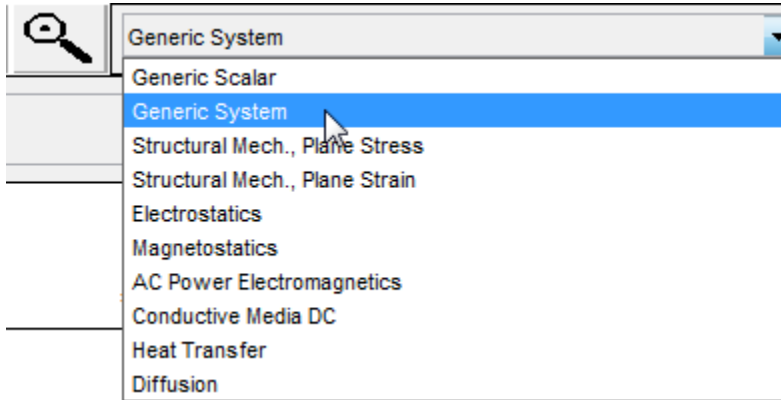


Then select **PDE > PDE Specification**.
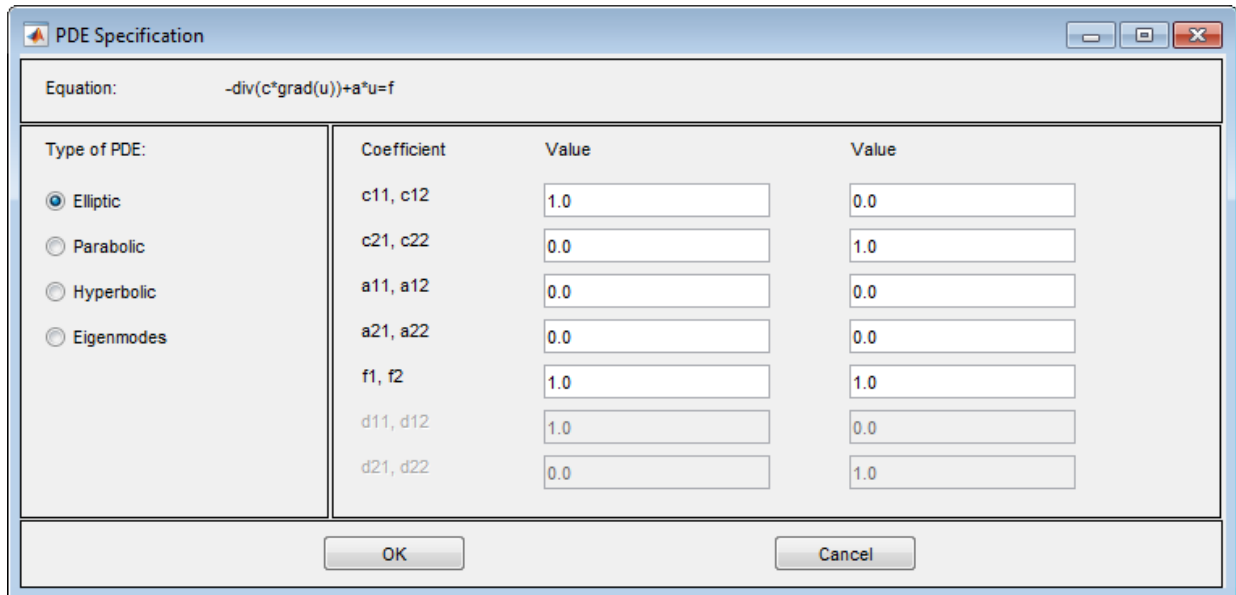
Enter character expressions for coefficients using the form in "Coefficients for Scalar PDEs in PDE App" on page 2-58, with additional options for nonlinear equations. The additional options are:

- 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)
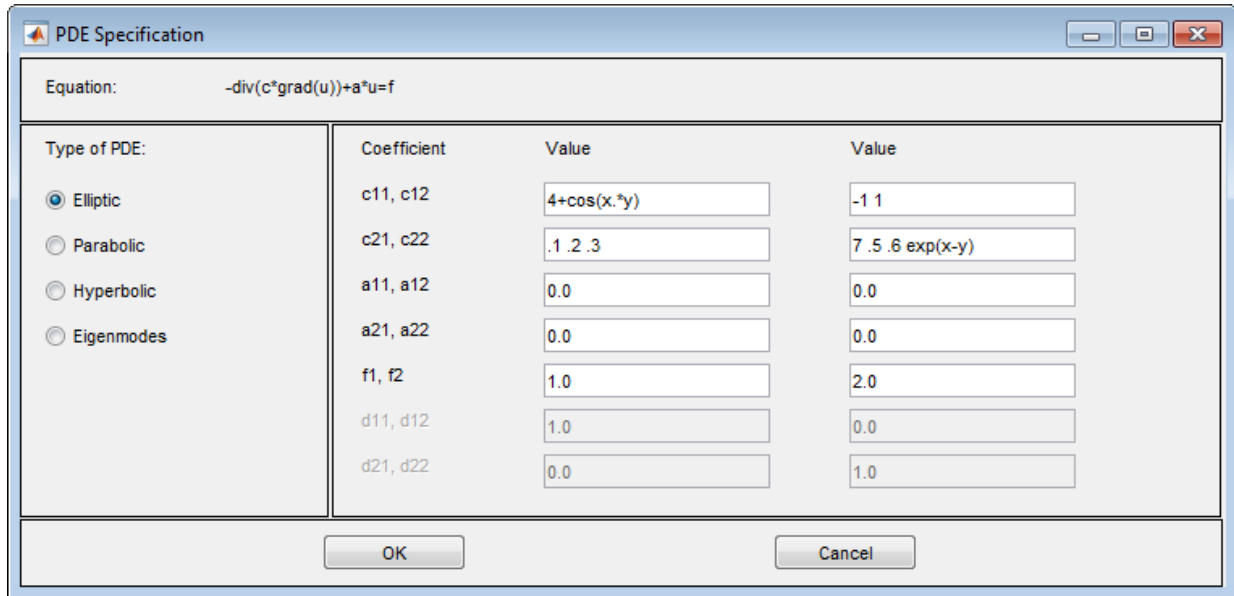
These expressions correspond to the equations

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

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

# 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 2-86.

---

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 "Equations You Can Solve Using Legacy Functions" on page 1-3. 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 Character Form" on page 2-55, 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 2-61. 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 2-61. Give solvers the function name as '*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 2-62.

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 0s 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 2-64. 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 a single vector of characters. 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 a single vector of characters, 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 2-83.

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

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

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 "Equations You Can Solve Using Recommended Functions" on page 1-6. 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 (∂*u*/∂*x*, ∂*u*/∂*y*, and ∂*u*/∂*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 2-110.

---

| In this section... |
|---|
| "Overview of the c Coefficient" on page 2-89 |
| "Definition of the c Tensor Elements" on page 2-90 |
| "Some c Vectors Can Be Short" on page 2-92 |
| "Functional Form" on page 2-106 |

## Overview of the c Coefficient

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

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

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 2-90. $N$ is the number of equations, see "Equations You Can Solve Using Recommended Functions" on page 1-6.

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 *N*1-by-*Nr*, where:

- *N*1 is the length of the vector representing the c coefficient. There are several possible values of *N*1, detailed in "Some c Vectors Can Be Short" on page 2-92. 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 2*N*-by-2*N* matrix, where the matrix is logically an *N*-by-*N* matrix of 2-by-2 blocks.

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

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

### 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 $[\texttt{c1;c2;c3}]$, the software interprets $\texttt{c}$ 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 $[\texttt{c1;c1;c2;c2;c3;c3}]$.

**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, $\cdot$ means the entry is symmetric.

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

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.

**4N²-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, $\cdot$ means the entry is symmetric.

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

**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 *N*1-by-*Nr*, where:

- *N*1 is the number of coefficients you pass to the solver. There are several possible values of *N*1, detailed in "Some c Vectors Can Be Short" on page 2-92. 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.

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 2-97:

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

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 2-5
- "f Coefficient for `specifyCoefficients`" on page 2-86
- "m, d, or a Coefficient for specifyCoefficients" on page 2-128
- "Deflection of a Piezoelectric Actuator" on page 3-15

# 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 2-89.

## 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 "Equations You Can Solve Using Legacy Functions" on page 1-3). 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 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.

Express c as numbers, text expressions, or functions, as in "f Coefficient for Systems" on page 2-83.

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 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 2-113 and "3-D Systems" on page 2-119.

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 2-61. Give solvers the function name as `'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 2-62.

For the function form of coefficients of 3-D systems, see "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

## 2-D Systems

- "Scalar c, 2-D Systems" on page 2-113
- "Two-Element Column Vector c, 2-D Systems" on page 2-114
- "Three-Element Column Vector c, 2-D Systems" on page 2-114
- "Four-Element Column Vector c, 2-D Systems" on page 2-115
- "N-Element Column Vector c, 2-D Systems" on page 2-115
- "2N-Element Column Vector c, 2-D Systems" on page 2-117
- "3N-Element Column Vector c, 2-D Systems" on page 2-118
- "4N-Element Column Vector c, 2-D Systems" on page 2-118
- "2N(2N+1)/2-Element Column Vector c, 2-D Systems" on page 2-118
- "$4N^2$-Element Column Vector c, 2-D Systems" on page 2-119

### 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, $\cdot$ means the entry is symmetric.

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

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" on page 2-120
- "Three-Element Column Vector c, 3-D Systems" on page 2-120

### 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.

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

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`

## Coefficients m, d, or a

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

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

or in the eigenvalue system

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

The 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 2-130.

---

**Note:** The d coefficient takes a special matrix form when m is nonzero. See "d Coefficient When m is Nonzero" on page 6-647.

---

## 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 2-129
- "N-Element Column Vector m, d, or a" on page 2-129
- "N(N+1)/2-Element Column Vector m, d, or a" on page 2-129
- "$N^2$-Element Column Vector m, d, or a" on page 2-130

### 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 2-129. $1 \leq N1 \leq N^2$.
- $Nr$ 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 (∂*u*/∂*x*, ∂*u*/∂*y*, and ∂*u*/∂*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.

**2-131**

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 2-129:

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

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 2-86
- "c Coefficient for `specifyCoefficients`" on page 2-89
- "Solve Problems Using PDEModel Objects" on page 2-5
- "Deflection of a Piezoelectric Actuator" on page 3-15

# 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 2-128.

---

| **In this section...** |
|---|
| "Coefficients a or d" on page 2-133 |
| "Scalar a or d" on page 2-134 |
| "N-Element Column Vector a or d" on page 2-134 |
| "N(N+1)/2-Element Column Vector a or d" on page 2-134 |
| "N$^2$-Element Column Vector a or d" on page 2-135 |

## 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. **a** and **d** are *N*-by-*N* matrices, where *N* is the number of equations, see "Equations You Can Solve Using Legacy Functions" on page 1-3.

Express the coefficients as numbers, text expressions, or functions, as in "f Coefficient for Systems" on page 2-83.

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 2-61. Give solvers the function name as `'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 2-62.

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.**
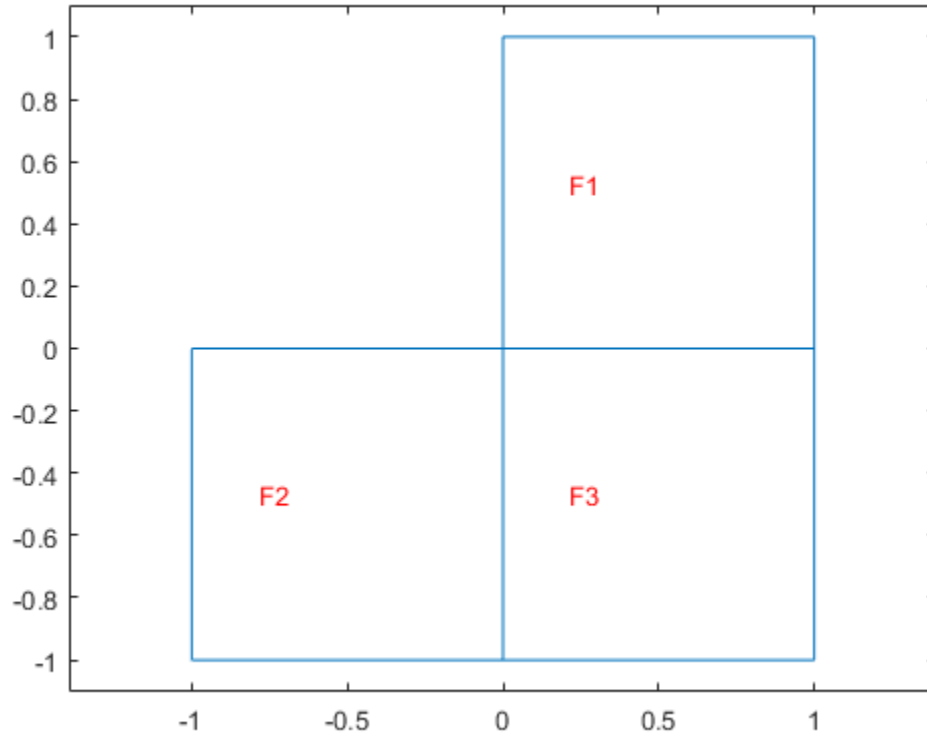
## View Coefficients

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,'FaceLabels','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. Remove all coefficients from `model`.

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

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

```
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 2-147.

---

## What Are Initial Conditions?

The term *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, use the following command.

```
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, use the following commands.

```
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]`, use the following commands.

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

```
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);
```

## Nodal Initial Conditions

You can use results of previous analysis as nodal initial conditions for your current model. The geometry and mesh of the model you used to obtain the results and the current model must be the same. For example, solve a time-dependent PDE problem for times from `t0` to `t1` with a time step `tstep`.

```
results = solvepde(model,t0:tstep:t1);
```

If later you need to solve this PDE problem for times from `t1` to `t2`, you can use `results` to set initial conditions. If you do not explicitly specify the time step, `setInitialConditions` uses `results` corresponding to the last solution time, `t1`.

```
setInitialConditions(model,results)
```

To use `results` for a particular solution time instead of the last one, specify the solution time index as a third parameter of `setInitialConditions`. For example, to use the solution at time `t0 + 10*tstep`, specify `11` as the third parameter.

```
setInitialConditions(model,results,11)
```

## Related Examples

- "Solve Problems Using PDEModel Objects" on page 2-5
- "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 2-147 page.

## View Initial Conditions

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

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

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

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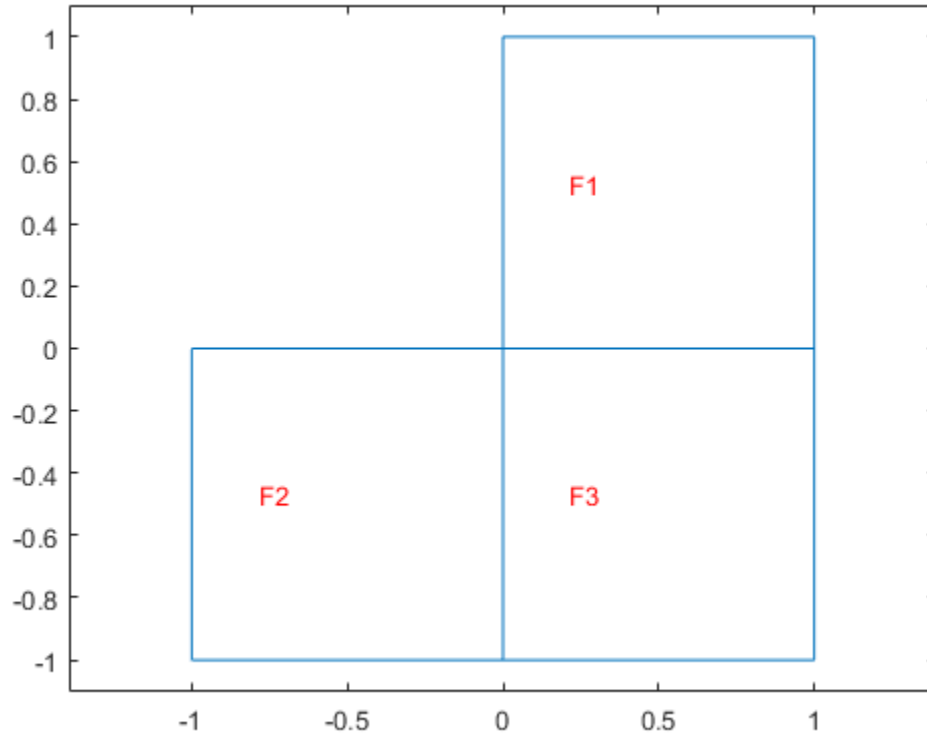
Specify constant initial conditions over all the regions in the model.

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

Specify a different initial condition on each subregion.

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

View the initial condition assignment for region 2.

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

```
ans =

  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. Remove all initial conditions from `model`.

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

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

```
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 initial condition 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 = @u0fun;
```

---

**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 2-140.

---

## 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 character 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 character 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 Character 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 2-196.

---

**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 2-196). 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". In the weak formulation you do not specify boundary conditions between subdomains, even if coefficients are discontinuous between subdomains. So &tm_partialdifferentialequationtoolbox; 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.

```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'))';

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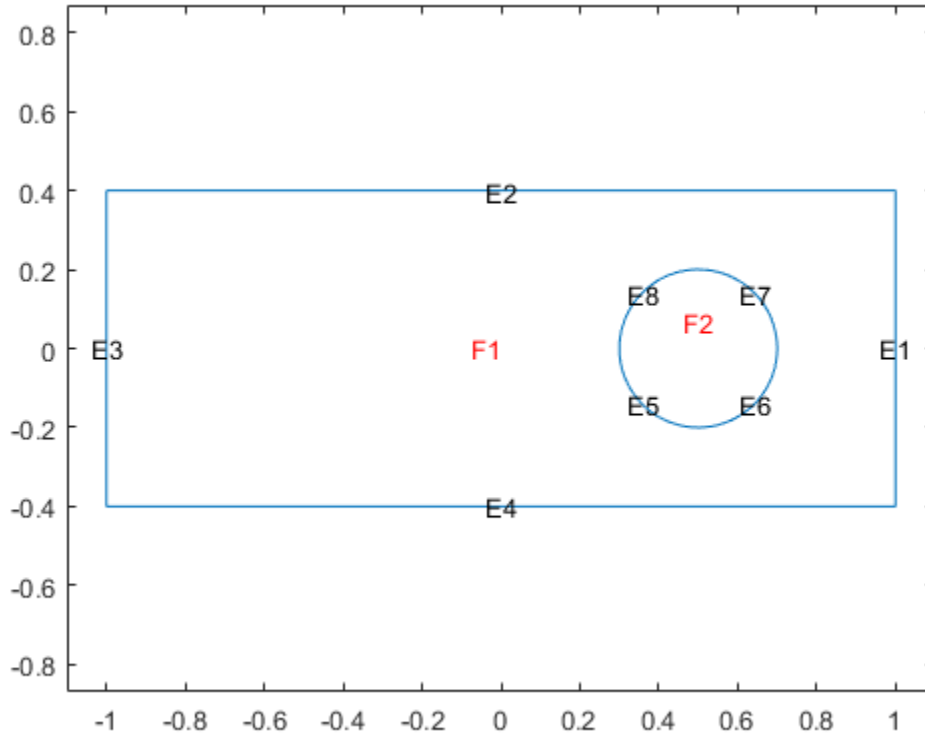
You need not give boundary conditions on segments 5, 6, 7, and 8, because these are subdomain boundaries, not exterior boundaries.

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:** **IDENTIFYING BOUNDARY CONDITIONS IS THE SAME FOR THE RECOMMENDED AND THE LEGACY WORKFLOWS.**

---

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')
```
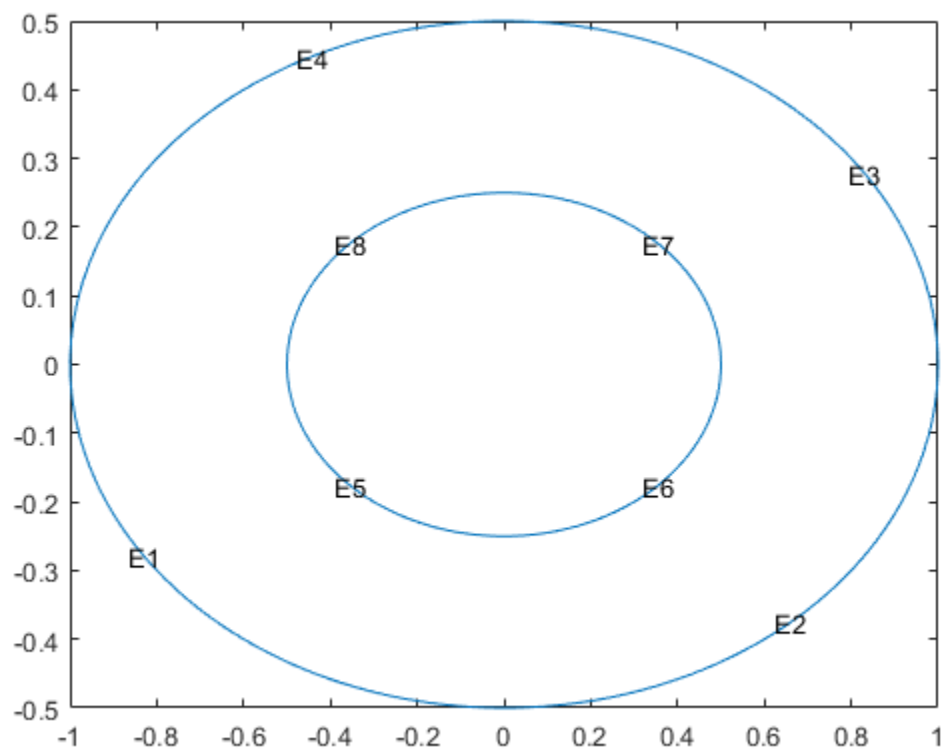
# Boundary Matrix for 2-D Geometry

---

**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" on page 2-160.

---

## 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 2-183.

For each column in the Decomposed Geometry matrix (see "Decomposed Geometry Data Structure" on page 2-14) 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 vectors of characters 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 vectors of characters representing **g**.
- Row $3 + N^2 + N$ to $3 + N^2 + N + MN - 1$ contain the lengths for the vectors of characters 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 vectors of characters representing **r**.

The following rows contain text expressions representing the actual boundary condition functions. The vectors of characters 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 vectors of characters. 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 ($M = 1$):

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

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

**2-155**

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;
```

```
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);
```

**2-157**

```
% 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" on page 2-160

# Specify Boundary Conditions

Before you create boundary conditions, you need to create a `PDEModel` container. For details, see "Solve Problems Using Legacy PDEModel Objects" on page 2-3. Suppose that you have a container named `model`, and that the geometry is stored in `model`. 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
```

Now you can specify the boundary conditions for each edge or face. 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 function of position, time, or the solution $u$, set boundary conditions by using the syntax in "Nonconstant Boundary Conditions" on page 2-165.

If you do not specify a boundary condition for an edge or face, the default is the Neumann boundary condition with the zero values for `'g'` and `'q'`.

## Dirichlet Boundary Conditions

### Scalar PDEs

The Dirichlet boundary condition implies that the solution $u$ on a particular edge or face satisfies the equation
$hu = r$,

where $h$ and $r$ are functions defined on $\partial\Omega$, and can be functions of space ($x$, $y$, and, in 3-D, $z$), the solution $u$, and, for time-dependent equations, time. Often, you take $h = 1$, and set $r$ to the appropriate value. You can specify Dirichlet boundary conditions as the value of the solution $u$ on the boundary or as a pair of the parameters $h$ and $r$.

Suppose that you have a PDE model named `model`, and edges or faces `[e1,e2,e3]`, where the solution $u$ must equal `2`. Specify this boundary condition as follows.

```
% For 3-D geometry:
```

```matlab
applyBoundaryCondition(model,'dirichlet','Face',[e1,e2,e3],'u',2);
% For 2-D geometry:
applyBoundaryCondition(model,'dirichlet','Edge',[e1,e2,e3],'u',2);
```

If the solution on edges or faces [e1,e2,e3] satisfies the equation $2u = 3$, specify the boundary condition as follows.

```matlab
% For 3-D geometry:
applyBoundaryCondition(model,'dirichlet','Face',[e1,e2,e3],'r',3,'h',2);
% For 2-D geometry:
applyBoundaryCondition(model,'dirichlet','Edge',[e1,e2,e3],'r',3,'h',2);
```

- If you do not specify 'r', applyBoundaryCondition sets its value to 0.
- If you do not specify 'h', applyBoundaryCondition sets its value to 1.

### Systems of PDEs

The Dirichlet boundary condition for a system of PDEs is **hu** = **r**, where **h** is a matrix, **u** is the solution vector, and **r** is a vector.

Suppose that you have a PDE model named model, and edge or face labels [e1,e2,e3] where the first component of the solution $u$ must equal 1, while the second and third components must equal 2. Specify this boundary condition as follows.

```matlab
% For 3-D geometry:
applyBoundaryCondition(model,'dirichlet','Face',[e1,e2,e3],...
                       'u',[1,2,2],'EquationIndex',[1,2,3]);
% For 2-D geometry:
applyBoundaryCondition(model,'dirichlet','Edge',[e1,e2,e3],...
                       'u',[1,2,2],'EquationIndex',[1,2,3]);
```

- 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.

Suppose that you have a PDE model named model, and edge or face labels [e1,e2,e3] where the first, second, and third components of the solution $u$ must satisfy the equations $2u_1 = 3$, $4u_2 = 5$, and $6u_3 = 7$, respectively. Specify this boundary condition as follows.

```matlab
H0 = [2 0 0;
      0 4 0;
```

```
      0 0 6];
R0 = [3;5;7];
% For 3-D geometry:
applyBoundaryCondition(model,'dirichlet','Face',[e1,e2,e3],'h',H0,'r',R0);
% For 2-D geometry:
applyBoundaryCondition(model,'dirichlet','Edge',[e1,e2,e3],'h',H0,'r',R0);
```

- The `'r'` parameter must be a numeric vector of length $N$. If you do not specify `'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" (MATLAB) form of the $N$-by-$N$ matrix. If you do not specify `'h'`, `applyBoundaryCondition` sets the value to the identity matrix.

## Neumann Boundary Conditions

### Scalar PDEs

Generalized Neumann boundary conditions imply that the solution $u$ on the edge or face 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

$$-\nabla \cdot (c \nabla u) + au = f \text{ on domain } \Omega$$

$\vec{n}$ is the outward unit normal. $q$ and $g$ are functions defined on $\partial \Omega$, and can be functions of space ($x$, $y$, and, in 3-D, $z$), the solution $u$, and, for time-dependent equations, time.

Suppose that you have a PDE model named `model`, and edges or faces `[e1,e2,e3]` where the solution $u$ must satisfy the Neumann boundary condition with `q = 2` and `g = 3`. Specify this boundary condition as follows.

```
% For 3-D geometry:
applyBoundaryCondition(model,'neumann','Face',[e1,e2,e3],'q',2,'g',3);
% For 2-D geometry:
applyBoundaryCondition(model,'neumann','Edge',[e1,e2,e3],'q',2,'g',3);
```

- If you do not specify `'g'`, `applyBoundaryCondition` sets its value to `0`.

- If you do not specify `'q'`, `applyBoundaryCondition` sets its value to `0`.

### Systems of PDEs

Neumann boundary conditions for a system of PDEs is $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{qu} = \mathbf{g}$. For 2-D systems, the notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ means the $N$-by-1 vector 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 3-D systems, the notation $\mathbf{n} \cdot (\mathbf{c} \otimes \nabla \mathbf{u})$ means the $N$-by-1 vector with $(i,1)$-component

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

where the outward normal vector of the boundary $\mathbf{n} = (\sin(\varphi)\cos(\theta), \sin(\varphi)\sin(\theta), \cos(\varphi))$. For each edge or face segment, there are a total of $N$ boundary conditions.

Suppose that you have a PDE model named `model`, and edges or faces `[e1,e2,e3]` where the first component of the solution $u$ must satisfy the Neumann boundary condition with `q = 2` and `g = 3`, and the second component must satisfy the Neumann boundary condition with `q = 4` and `g = 5`. Specify this boundary condition as follows.

```
Q = [2 0; 0 4];
G = [3;5];
% For 3-D geometry:
applyBoundaryCondition(model,'neumann','Face',[e1,e2,e3],'q',Q,'g',G);
% For 2-D geometry:
applyBoundaryCondition(model,'neumann','Edge',[e1,e2,e3],'q',Q,'g',G);
```

- The `'g'` parameter must be a numeric vector of length *N*. If you do not specify `'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" (MATLAB) form of the *N*-by-*N* matrix. If you do not specify `'q'`, `applyBoundaryCondition` sets the values to `0`.

## Mixed Boundary Conditions

If some equations in your system of PDEs must satisfy the Dirichlet boundary condition and some must satisfy the Neumann boundary condition for the same geometric region, use the `'mixed'` parameter to apply boundary conditions in one call. Note that `applyBoundaryCondition` uses the default Neumann boundary condition with `g = 0` and `q = 0` for equations for which you do not explicitly specify a boundary condition.

Suppose that you have a PDE model named `model`, and edge or face labels `[e1,e2,e3]` where the first component of the solution *u* must equal `11`, the second component must equal `22`, and the third component must satisfy the Neumann boundary condition with `q = 3` and `g = 4`. Express this boundary condition as follows.

```
Q = [0 0 0; 0 0 0; 0 0 3];
G = [0;0;4];
% For 3-D geometry:
applyBoundaryCondition(model,'mixed','Face',[e1,e2,e3],...
                       'u',[11,22],'EquationIndex',[1,2],...
                       'q',Q,'g',G);
% For 2-D geometry:
applyBoundaryCondition(model,'mixed',...
                          'Edge',[e1,e2,e3],'u',[11,22],...
                          'EquationIndex',[1,2],'q',Q,'g',G);
```

Suppose that you have a PDE model named `model`, and edges or faces `[e1,e2,e3]` where the first component of the solution *u* must satisfy the Dirichlet boundary condition $2u_1 = 3$, the second component must satisfy the Neumann boundary condition with `q = 4` and `g = 5`, and the third component must satisfy the Neumann boundary condition with `q = 6` and `g = 7`. Express this boundary condition as follows.

```
h = [2 0 0; 0 0 0; 0 0 0];
r = [3;0;0];
Q = [0 0 0; 0 4 0; 0 0 6];
G = [0;5;7];
% For 3-D geometry:
```

```
applyBoundaryCondition(model,'mixed','Face',[e1,e2,e3],'h',h,'r',r,'q',Q,'g',G);
% For 2-D geometry:
applyBoundaryCondition(model,'mixed','Edge',[e1,e2,e3],'h',h,'r',r,'q',Q,'g',G);
```

## Nonconstant Boundary Conditions

Use functions to express nonconstant boundary conditions.

```
applyBoundaryCondition(model,'dirichlet','Edge',1,'r',@myrfun);
applyBoundaryCondition(model,'neumann','Face',2,'g',@mygfun,'q',@myqfun);
applyBoundaryCondition(model,'mixed','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 use `Vectorized,'off'`, then solvers pass just one evaluation point in each call.

  - `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" (MATLAB) 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.

If boundary conditions depend on `state.u` or `state.time`, ensure that your function returns a matrix of `NaN` of the correct size when `state.u` or `state.time` are `NaN`. Solvers check whether a problem is nonlinear or time-dependent by passing `NaN` state values, and looking for returned `NaN` values.

See "Solve PDEs with Nonconstant Boundary Conditions" on page 2-172.

# 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
```
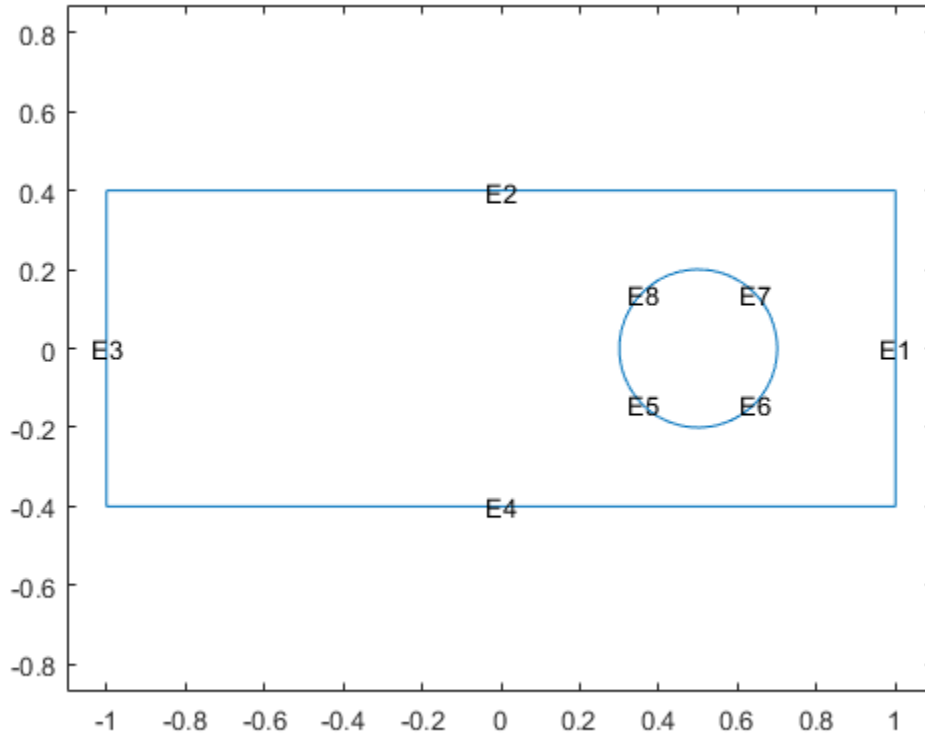
## Scalar Problem

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,'dirichlet','edge',3,'u',32);
applyBoundaryCondition(model,'dirichlet','edge',1,'u',72);
applyBoundaryCondition(model,'neumann','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 `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 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,'dirichlet','edge',3,'u',32);
applyBoundaryCondition(model,'dirichlet','edge',1,'u',72);
applyBoundaryCondition(model,'neumann','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 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)
```

## More About

- "Specify Boundary Conditions" on page 2-160
- "Solve PDEs with Nonconstant Boundary Conditions" on page 2-172

# 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.

## Geometry

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
```
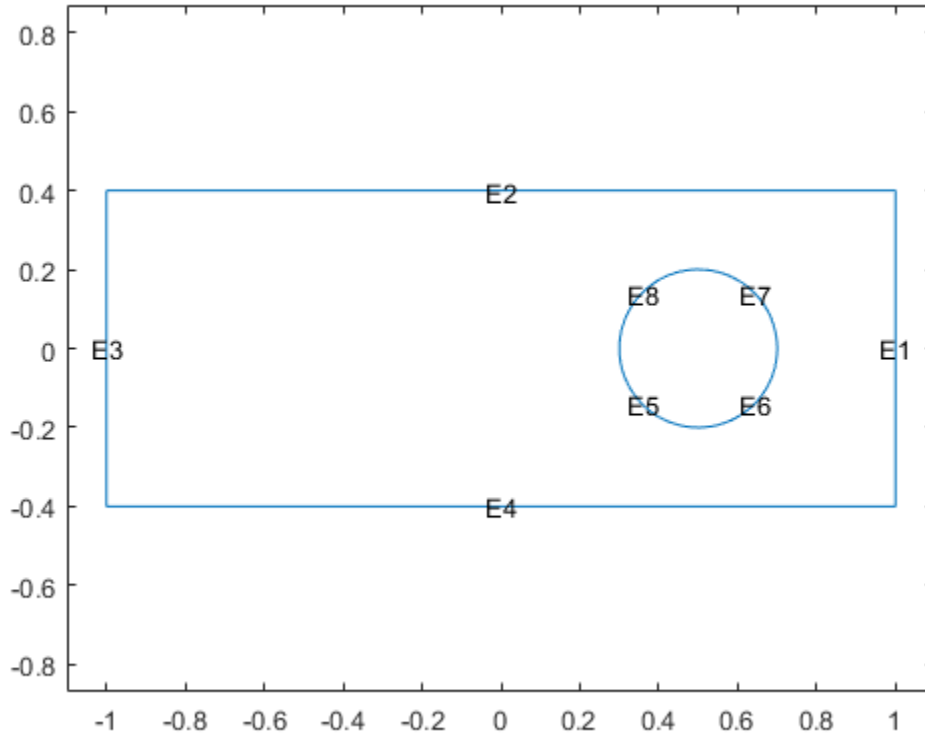
## Scalar Problem

- 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,'dirichlet','Edge',3,'u',32);
applyBoundaryCondition(model,'dirichlet','Edge',1,'u',72);
applyBoundaryCondition(model,'neumann','Edge',5:8,'g',-1); % q = 0 by default
```

2-173

Edges 2 and 4 need functions that perform the linear interpolation. Each edge can use the same function that returns the value $u\left(x, y\right) = 52 + 20x$.

You can implement this simple interpolation in an anonymous function.

```
myufunction = @(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,'dirichlet','Edge',[2,4],...
                                  'u',myufunction,...
                                  '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)
```
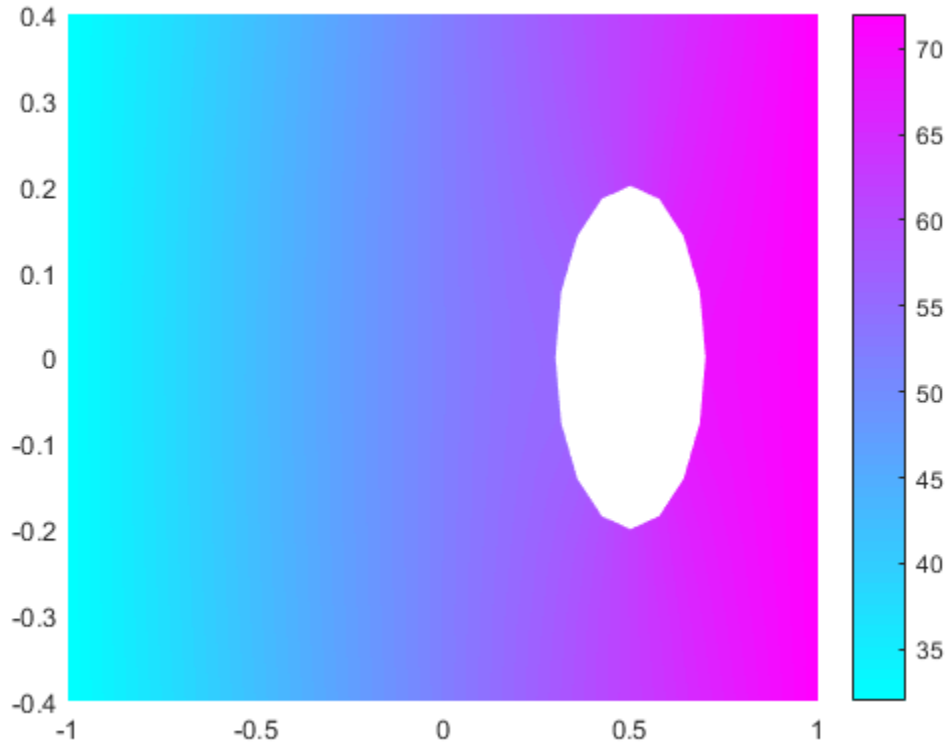
## System of PDEs

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,'dirichlet','Edge',3,'u',[32,72]);
applyBoundaryCondition(model,'dirichlet','Edge',1,'u',[72,32]);
applyBoundaryCondition(model,'neumann','Edge',5:8,'g',[-10,-10]);
```

The first component of edges 2 and 4 satisfies the equation
$$u_1\left(x\right) = 52 + 20x + 10\sin\left(\pi x^3\right)$$.

The second component satisfies $u_2\left(x\right) = 52 - 20x - 10\sin\left(\pi x^3\right)$.

Write a function file `myufun.m` that incorporates these equations in the syntax described in "Nonconstant Boundary Conditions".

```
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
end
```

Include this function in the edge 2 and edge 4 boundary condition.

```
applyBoundaryCondition(model,'dirichlet','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)
```
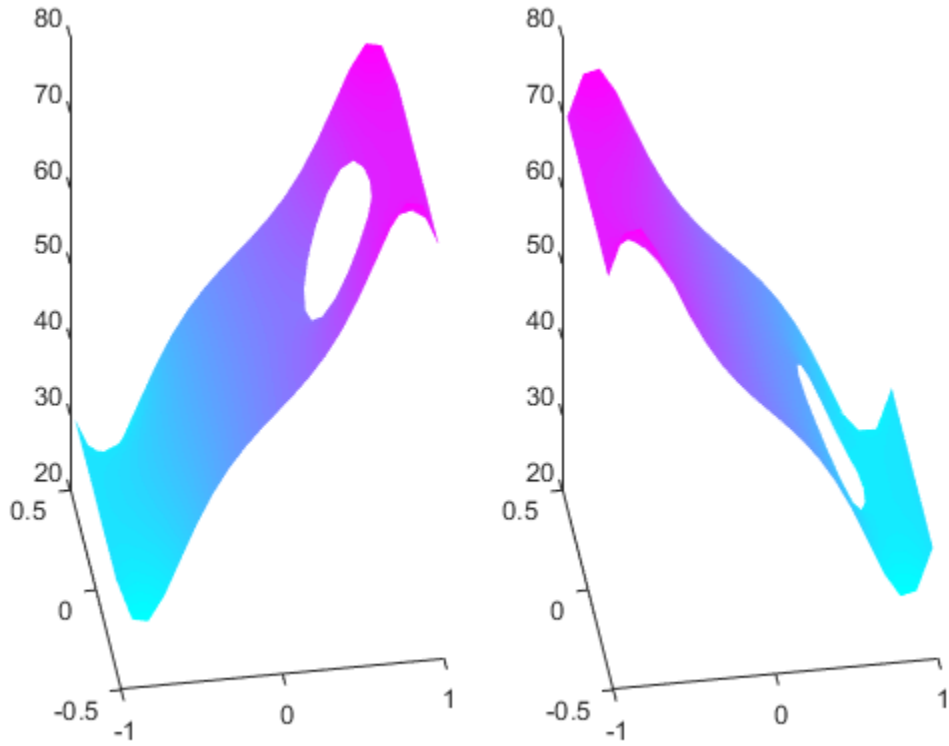
# View, Edit, and Delete Boundary Conditions

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

| In this section... |
| --- |
| "View Boundary Conditions" on page 2-178 |
| "Delete Existing Boundary Conditions" on page 2-181 |
| "Change a Boundary Conditions Assignment" on page 2-181 |

## View Boundary Conditions

A PDE model stores boundary conditions in its `BoundaryConditions` property. To obtain the boundary conditions stored in the PDE model called `model`, use this syntax:

```
BCs = model.BoundaryConditions;
```

To see the active boundary condition assignment for a region, call the `findBoundaryConditions` function.

For example, create a model and view the geometry.

```
model = createpde(3);
importGeometry(model,'Block.stl');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
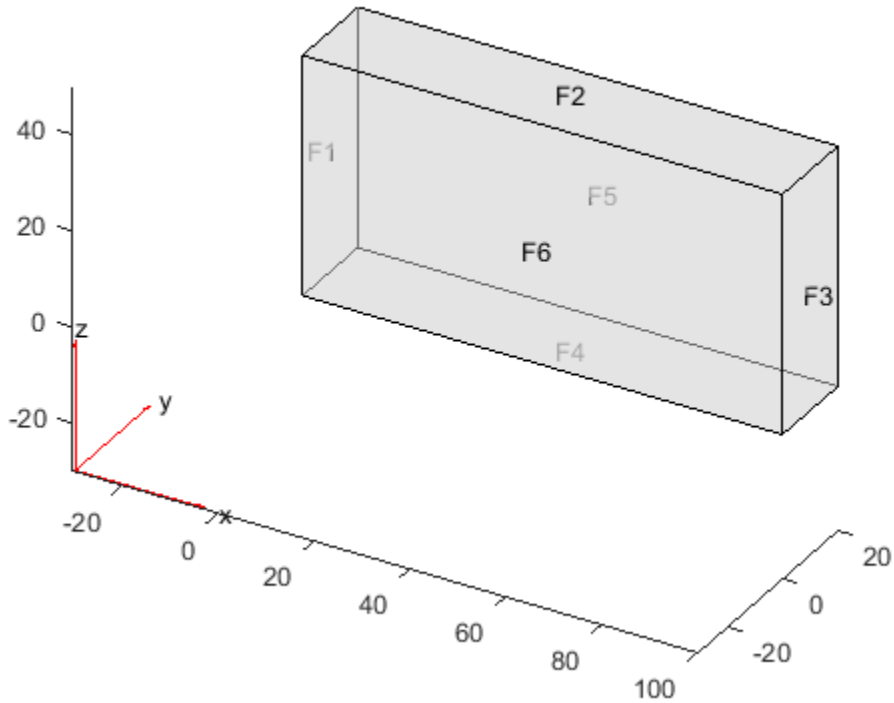
Set zero Dirichlet conditions for all equations and all regions in the model.

```
applyBoundaryCondition(model,'dirichlet','Face',1:6,'u',[0,0,0]);
```

On face 3, set the Neumann boundary condition for equation 1 and Dirichlet boundary condition for equations 2 and 3.

```
h = [0 0 0;0 1 0;0 0 1];
r = [0;3;3];
q = [1 0 0;0 0 0;0 0 0];
g = [10;0;0];
applyBoundaryCondition(model,'mixed','Face',3,'h',h,'r',r,'g',g,'q',q);
```

View the boundary condition assignment for face 3. The result shows that the active boundary condition is the last assignment.

```
BCs = model.BoundaryConditions;
findBoundaryConditions(BCs,'Face',3)
```

```
ans =

  BoundaryCondition with properties:

          BCType: 'mixed'
      RegionType: 'Face'
        RegionID: 3
               r: [3×1 double]
               h: [3×3 double]
               g: [3×1 double]
               q: [3×3 double]
               u: []
   EquationIndex: []
      Vectorized: 'off'
```

View the boundary conditions assignment for face 1.

```
findBoundaryConditions(BCs,'Face',1)
```

```
ans =

  BoundaryCondition with properties:

          BCType: 'dirichlet'
      RegionType: 'Face'
        RegionID: [1 2 3 4 5 6]
               r: []
               h: []
               g: []
               q: []
               u: [0 0 0]
   EquationIndex: []
      Vectorized: 'off'
```

The active boundary conditions assignment for face 1 includes all six faces, though this assignment is no longer active for face 3.

## Delete Existing Boundary Conditions

To remove all the boundary conditions in the PDE model called `pdem`, use `delete`.

```
delete(pdem.BoundaryConditions)
```

To remove specific boundary conditions assignments from `pdem`, delete them from the `pdem.BoundaryConditions.BoundaryConditionAssignments` vector. For example,

```
BCv = pdem.BoundaryConditions.BoundaryConditionAssignments;
delete(BCv(2))
```

---

**Tip:** You do not need to delete boundary conditions; you can override them by calling `applyBoundaryCondition` again. However, removing unused assignments can make your model more concise.

---

## Change a Boundary Conditions Assignment

To change a boundary conditions assignment, you need the boundary condition's handle. To get the boundary condition's handle:

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

  ```
  bc1 = applyBoundaryCondition(model,'dirichlet','Face',1:6,'u',[0 0 0]);
  ```

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

  ```
  BCs = model.BoundaryConditions;
  bc1 = findBoundaryConditions(BCs,'Face',2)

  bc1 = 

    BoundaryCondition with properties:

            BCType: 'dirichlet'
        RegionType: 'Face'
          RegionID: [1 2 3 4 5 6]
                 r: []
                 h: []
  ```

```
                     g: []
                     q: []
                     u: [0 0 0]
       EquationIndex: []
          Vectorized: 'off'
```

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

```
bc1.BCType = 'neumann';
bc1.u = [];
bc1.g = [0 0 0];
bc1.q = [0 0 0];
bc1

bc1 =

  BoundaryCondition with properties:

           BCType: 'neumann'
       RegionType: 'Face'
         RegionID: [1 2 3 4 5 6]
                r: []
                h: []
                g: [0 0 0]
                q: [0 0 0]
                u: []
    EquationIndex: []
       Vectorized: 'off'
```

**Note:** Editing an existing assignment in this way does not change its priority. For example, if the active boundary condition was assigned after bc1, then editing bc1 does not make bc1 the active boundary condition.

## Related Examples

- "Specify Boundary Conditions" on page 2-160

# 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" on page 2-160.

## 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" on page 2-160.

To use this legacy syntax, write the functions using the templates in "Boundary Conditions for Scalar PDE" on page 2-183 or "Boundary Conditions for PDE Systems" on page 2-188.

## 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 2-196)
- e — Finite element edges in the mesh, a subset of all the edges ("Mesh Data" on page 2-196)
- 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 2-152.

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 2-152).

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

Use the following template for your boundary file.

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

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

for k = 1:ne
    x1 = p(1,e(1,k)); % x at first point in segment
```

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

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

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

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

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

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

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

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

For example, consider the following geometry, a rectangle with a circular hole.

```
% 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
```

```matlab
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];

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

% Set formula
sf = 'R1-C1';

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

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

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

Write the following boundary file to represent the boundary conditions:

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

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

**2-187**

```
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 "Equations You Can Solve Using Legacy Functions" on page 1-3) 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 $\mu$ 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 2-196)
- `e` — Finite element edges in the mesh, a subset of all the edges ("Mesh Data" on page 2-196)
- `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 2-152.

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 2-152).

  `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 "Equations You Can Solve Using Legacy Functions" on page 1-3. 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.

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

```
C1 = [C1;zeros(length(R1)-length(C1),1)];
geom = [R1,C1];

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

% Set formula
sf = 'R1-C1';

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

% View geometry
pdegplot(gd,'EdgeLabels','on')
xlim([-1.1 1.1])
axis equal
```
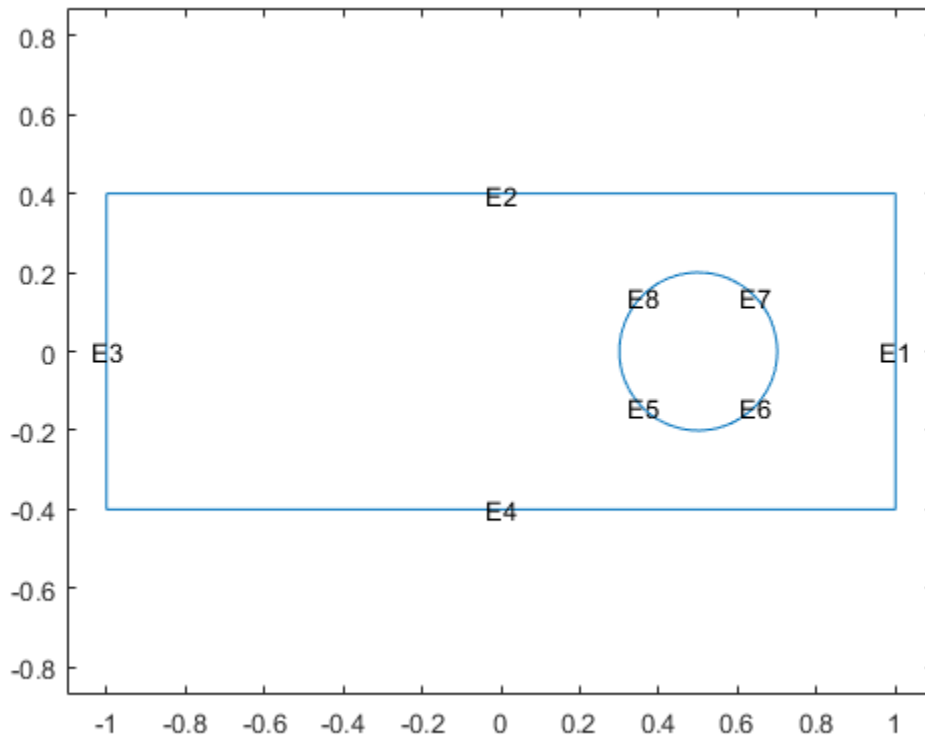
Suppose $N = 3$. Suppose the boundary conditions are mixed. There is M = 1 Dirichlet condition:

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

- The second components of $u = 0$ on the circular segments (numbers 5–8). So **h**(2,2) = 1 and **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
```

```matlab
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 2-196 |
| "Mesh Data for FEMesh" on page 2-196 |
| "Mesh Data for [p,e,t] Triples: 2-D" on page 2-196 |
| "Mesh Data for [p,e,t] Triples: 3-D" on page 2-197 |

## 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.
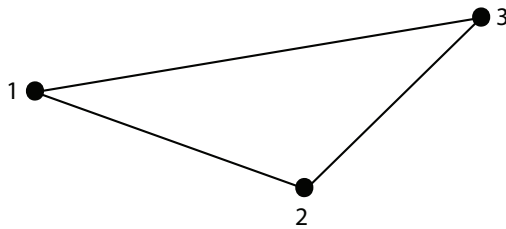


2-D linear element
showing node numbering
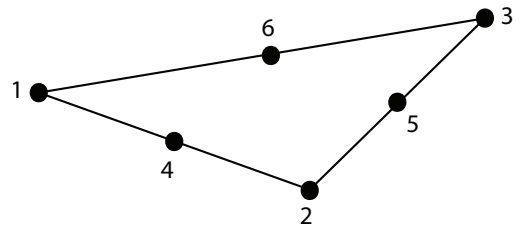
2-D quadratic element
showing node numbering

**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.
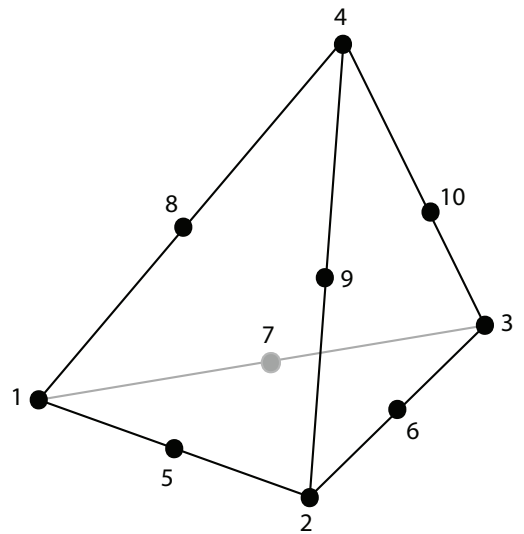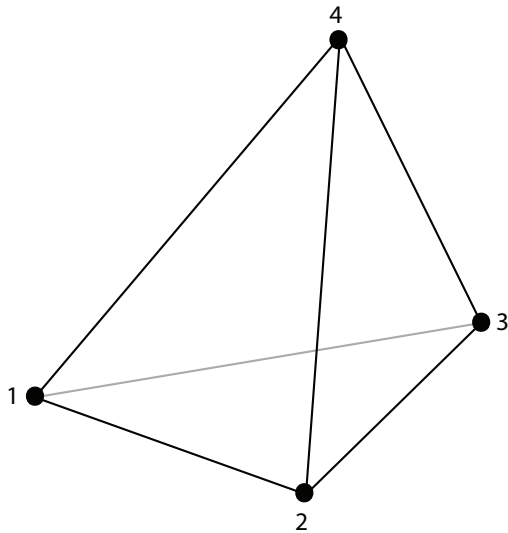


## Related Examples

- "Solve PDE with Coefficients in Functional Form" on page 2-66
- "Solve Poisson's Equation on a Unit Disk" on page 3-120

# Adaptive Mesh Refinement

| In this section... |
| --- |
| "Improving Solution Accuracy Using Mesh Refinement" on page 2-199 |
| "Error Estimate for the FEM Solution" on page 2-200 |
| "Mesh Refinement Functions" on page 2-201 |
| "Mesh Refinement Termination Criteria" on page 2-201 |

## 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 $\Omega$, the following error estimate for the FEM-solution $u_h$ holds in the $L_2$-norm $\|\cdot\|$:

$$\left\|\nabla(u - u_h)\right\| \le \alpha \left\|hf\right\| + \beta D_h(u_h)$$

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 $\tau$, $h_\tau$ is the length of edge $\tau$, and the sum runs over $E_i$, the set of all interior edges of the triangulation. The coefficients $\alpha$ and $\beta$ 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

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

where $\mathbf{n}_\tau$ is the unit normal of edge $\tau$ 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

# von Mises Effective Stress and Displacements

This example shows how to compute the displacements $u$ and $v$ and the von Mises effective stress for 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. The example uses the PDE app. The app also lets you compute and visualize other properties, such as the $x$- and $y$-direction strains and stresses and the shear stress.

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:

- Dimensions 1 m-by-1 m-by 0.001 m;
- Inset is 1/3-by-1/3 m
- 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. To specify a surface traction, divide the load by the thickness (0.001 m). Thus, the surface traction is 0.5 MN/m$^2$. The force unit in this example is MN.

To solve this problem in the PDE app, follow these steps:

1 Draw a polygon with corners (0 1), (2/3,1), (1,2/3), (1,0), (1/3,0), (1/3,1/3), (0,1/3) and a circle with the center (2/3, 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)
```

2 Set the $x$-axis limit to [-0.5 1.5] and $y$-axis limit to [0 1.2]. To do this, select **Options** > **Axes Limits** and set the corresponding ranges.

3 Model the geometry by entering P1+C1 in the **Set formula** field.

4 Set the application mode to **Structural Mechanics, Plane Stress**.

5 Remove all subdomain borders. To do this, switch to the boundary mode by selecting **Boundary** > **Boundary Mode**. Then select **Boundary** > **Remove All Subdomain Borders**.

6 Display the edge labels by selecting **Boundary** > **Show Edge Labels**.
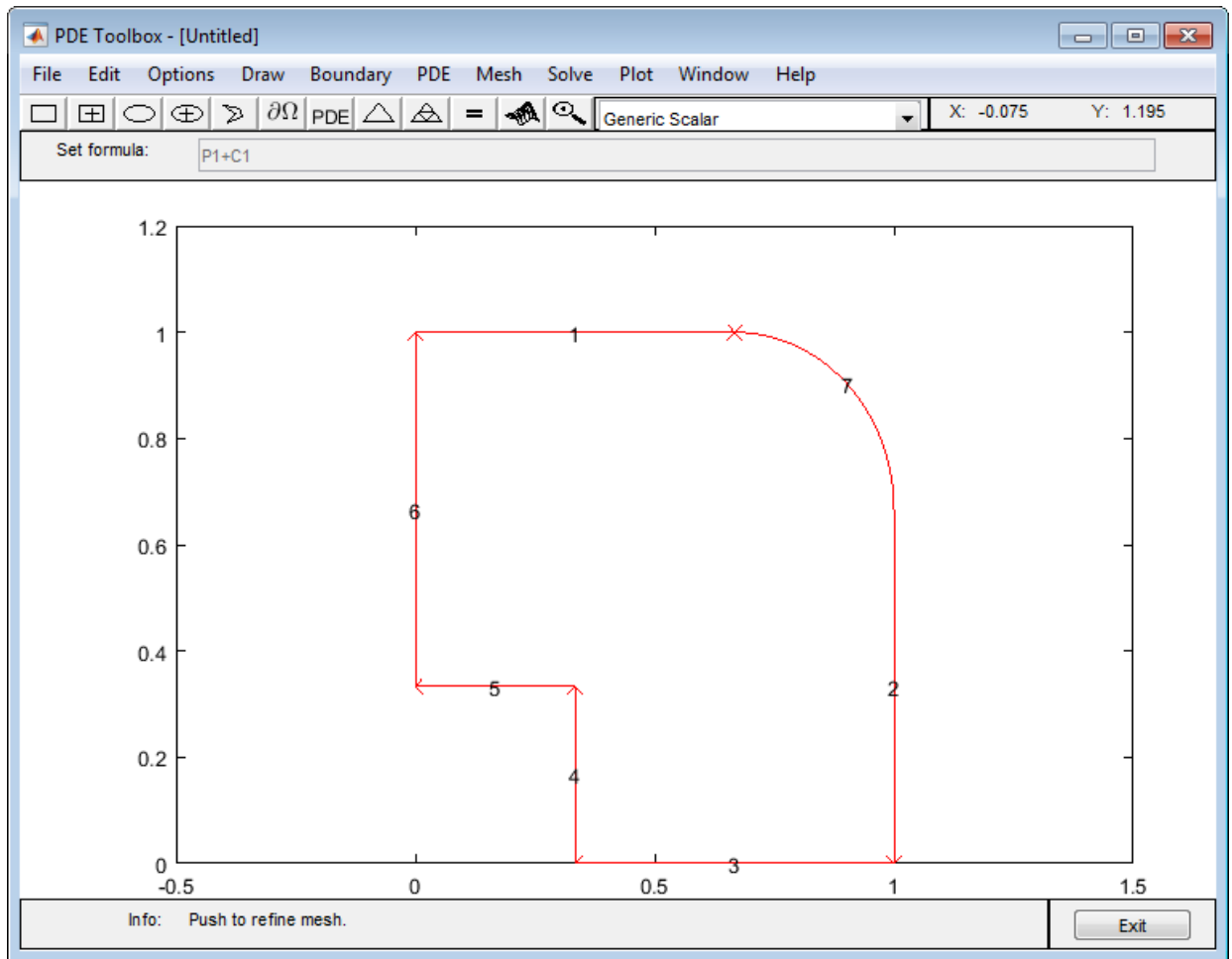
7  Specify the boundary conditions. To do this, select **Boundary** > **Specify Boundary Conditions**.

- For convenience, first specify the Neumann boundary condition g1 = g2 = 0, q11 = q12 = q21 = q22 = 0 (no normal stress) for all boundaries. Use **Edit** > **Select All** to select all boundaries.

- For the two clamped boundaries at the inset in the lower left (edges 4 and 5), specify the Dirichlet boundary condition with zero displacements: h11 = 1, h12

= 0, h21 = 0, h22 = 1, r1 = 0, r2 = 0. Use **Shift**+click to select several boundaries.

- For the rounded cut (edge 7), specify the Neumann boundary condition: g1 = 0.5*nx, g2 = 0.5*ny, q11 = q12 = q21 = q22 = 0.

**8** Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Specify E = 196E3 and nu = 0.31. The material is homogeneous, so the same values E and nu apply to the entire 2-D domain. Because there are no volume forces, specify Kx = Ky = 0. The elliptic type of PDE for plane stress does not use density, so you can specify any value. For example, specify pho = 0.

**9** Initialize the mesh by selecting **Mesh** > **Initialize Mesh**. Refine the mesh by selecting **Mesh** > **Refine Mesh**.

**10** Refining the mesh in areas where the gradient of the solution (the stress) is large. To do this, select **Solve** > **Parameters**. In the resulting dialog box, select **Adaptive mode**. Use the default adaptation options: the **Worst triangles** triangle selection method with the **Worst triangle fraction** set to 0.5.

**11** Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.

**12** Plot the von Mises effective stress using color. Plot the displacement vector field ($u$,$v$) using a deformed mesh. To do this:

**a** Select **Plot** > **Parameters**.

**b** In the resulting dialog box, select the **Color** and **Deformed mesh** options. Select von Mises from the **Color** drop-down menu. Select **Show Mesh** to observe the refined mesh in large stress areas.
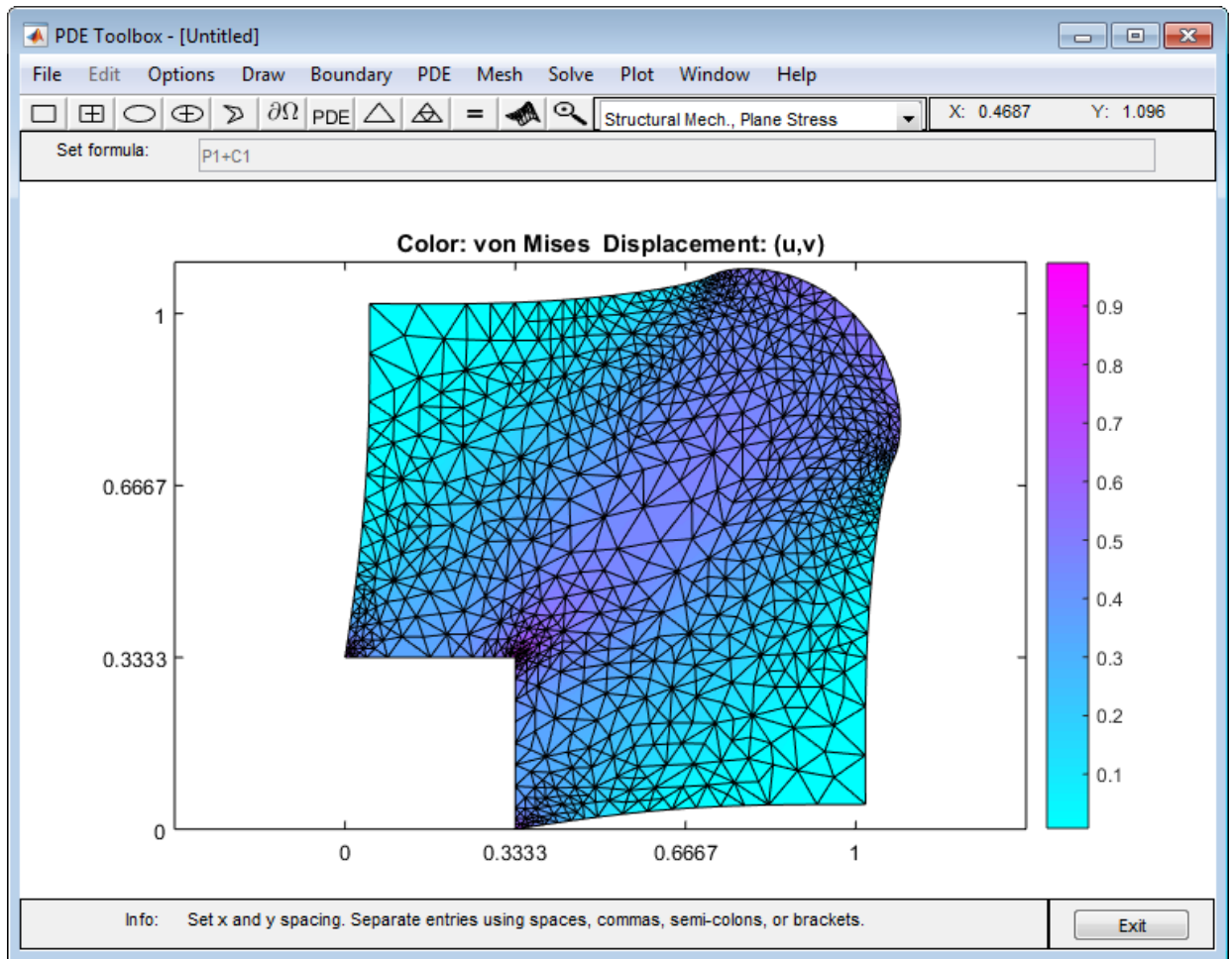
By selecting other options from the **Color** drop-down menu, you can visualize different strain and stress properties, such as the *x*- and *y*-direction strains and stresses, the shear stress, and the principal stresses and strains. You also can plot combinations of scalar and vector properties by using color, height, vector field arrows, and displacements in a 3-D plot to represent different properties.

# Plane Stress and Plane Strain

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. Assuming isotropic and isothermal conditions, the stress-strain relation can then be written as follows:

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

where $\sigma_x$ and $\sigma_y$ are the normal stresses in the $x$ and $y$ directions, and $\tau_{xy}$ 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 yields the displacement equation:

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

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

$$c_{11}=\begin{pmatrix}2G+\mu & 0\\ 0 & G\end{pmatrix},\quad c_{12}=\begin{pmatrix}0 & \mu\\ G & 0\end{pmatrix},\quad c_{21}=\begin{pmatrix}0 & G\\ \mu & 0\end{pmatrix},\quad c_{22}=\begin{pmatrix}G & 0\\ 0 & 2G+\mu\end{pmatrix}$$

$$G=\frac{E}{2(1+v)},\quad \mu=2G\frac{v}{1-v},\quad \mathbf{k}=\begin{pmatrix}K_x\\ K_y\end{pmatrix}$$

Here $G$ is the shear modulus, and **k** is a vector of volume forces. In addition, you can describe the plane stress problem as an eigenvalue problem where, $\rho$ is the material density:

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

*Plane strain* is 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*. The stress-strain relation is only slightly different from the plane stress case, and the same set of material parameters is used.

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

• The $\mu$ 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.

## References

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

# 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, $\nu$ 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;
model = 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(model,g);
```

Plot the geometry and display the edge labels for use in the boundary condition definition.

```
figure;
pdegplot(model,'EdgeLabels','on');
ylim([-1,11])
axis equal
title 'Geometry With Edge Labels Displayed';
```

**Geometry With Edge Labels Displayed**
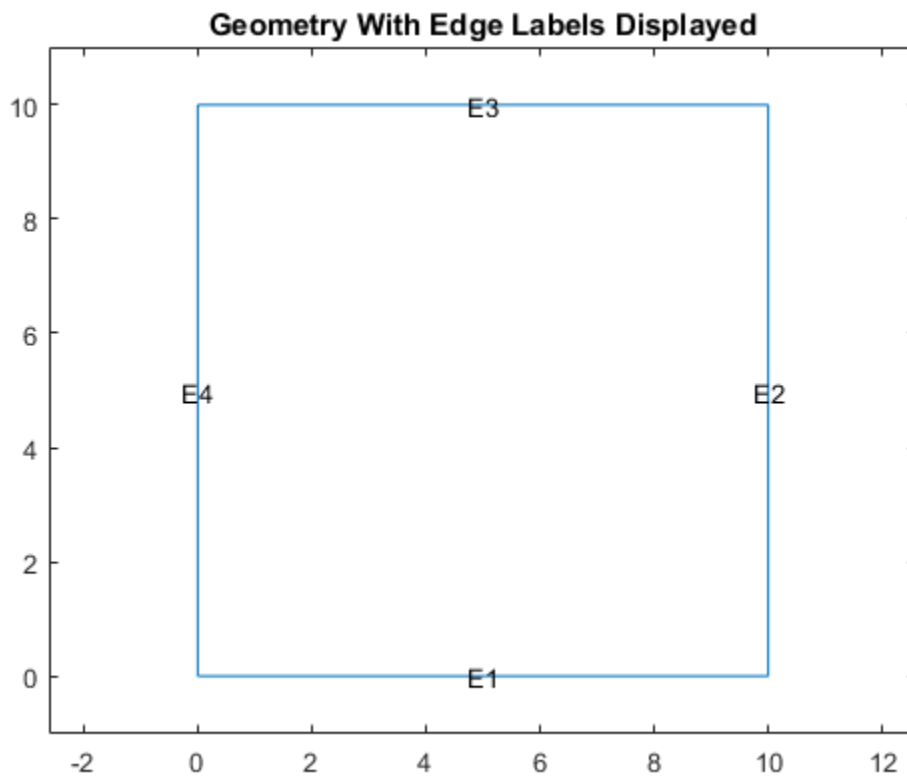


### Coefficient Definition

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 $2 \times 2$ submatrices is shown below.

$$\begin{bmatrix} c(1) & c(2) & \cdot & \cdot \\ \cdot & c(3) & \cdot & \cdot \\ \cdot & \cdot & c(4) & c(5) \\ \cdot & \cdot & \cdot & c(6) \end{bmatrix}$$

The six-row by one-column c matrix is defined below. The entries in the full $2 \times 2$ a matrix and the $2 \times 1$ 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(model,'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(model,'neumann','Edge',(1:4),...
                                'g',[0 0],'q',[0 0; k 0]);
```

### Mesh generation

```
generateMesh(model, '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(model);
u = res.NodalSolution;
numNodes = size(model.Mesh.Nodes,2);
figure
pdeplot(model,'XYData',u(1:numNodes),'Contour','on');
title 'Transverse Deflection'
```

```
numNodes = size(model.Mesh.Nodes,2);
fprintf('Transverse deflection at plate center(PDE Toolbox) = %12.4e\n',...
                                              min(u(1:numNodes,1)));
```

Transverse deflection at plate center(PDE Toolbox) =  -2.7563e-01
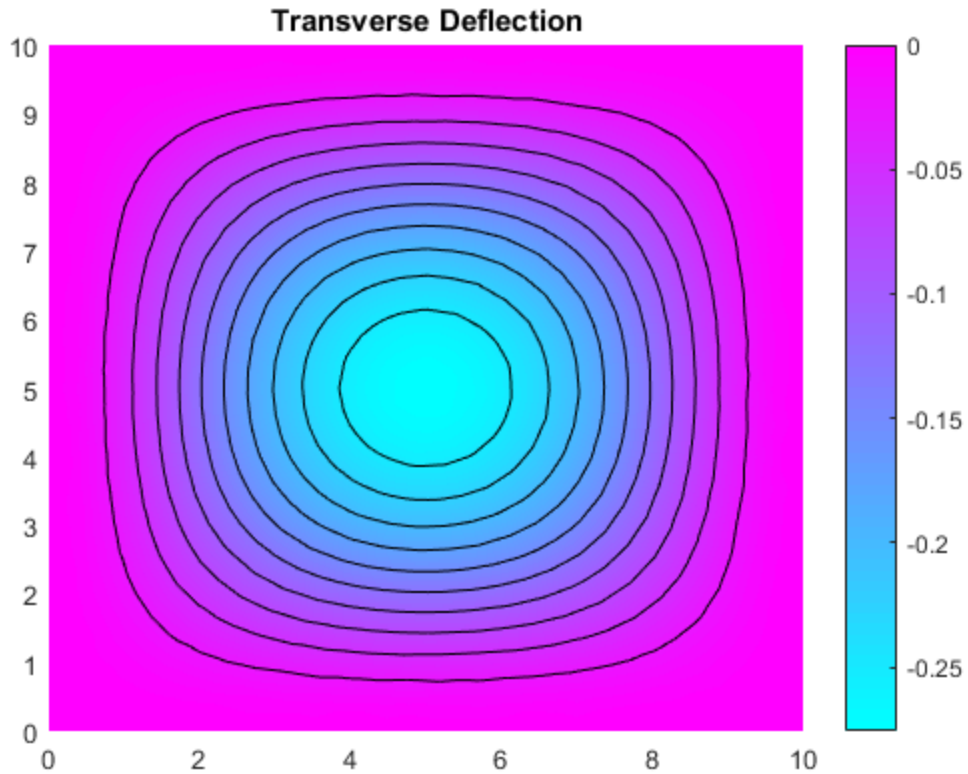


Ccompute analytical solution.

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

Transverse deflection at plate center(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 $\sigma$ 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 $\rho$ 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, $\sigma$ 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

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

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

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

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

and the electric field written in terms of the electrical potential, $\phi$, 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 $2 \times 2$ 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

$$
\begin{bmatrix}
c_{1111} & c_{1112} & c_{1211} & c_{1212} & c_{1311} & c_{1312} \\
\cdot & c_{1122} & c_{1221} & c_{1222} & c_{1321} & c_{1322} \\
\cdot & \cdot & c_{2211} & c_{2212} & c_{2311} & c_{2312} \\
\cdot & \cdot & \cdot & c_{2222} & c_{2321} & c_{2322} \\
\cdot & \cdot & \cdot & \cdot & c_{3311} & c_{3312} \\
\cdot & \cdot & \cdot & \cdot & \cdot & c_{3322}
\end{bmatrix}
\begin{Bmatrix}
\frac{\partial u}{\partial x} \\
\frac{\partial u}{\partial y} \\
\frac{\partial v}{\partial x} \\
\frac{\partial v}{\partial y} \\
\frac{\partial \phi}{\partial x} \\
\frac{\partial \phi}{\partial y}
\end{Bmatrix}
$$

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

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

### Piezoelectric Bimorph Actuator Model

Now that we have defined the equations for a 2D piezoelectric material, we are ready to apply these to a specific model. The model is a two-layer cantilever beam that has been extensively studied (e.g. refs 1 and 2). It is defined as a "bimorph" because although both layers are made of the same Polyvinylidene Fluoride (PVDF) material, in the top layer the polarization direction points down (minus y direction) and in the bottom layer, it points up. A schematic of the cantilever beam is shown in the figure below.
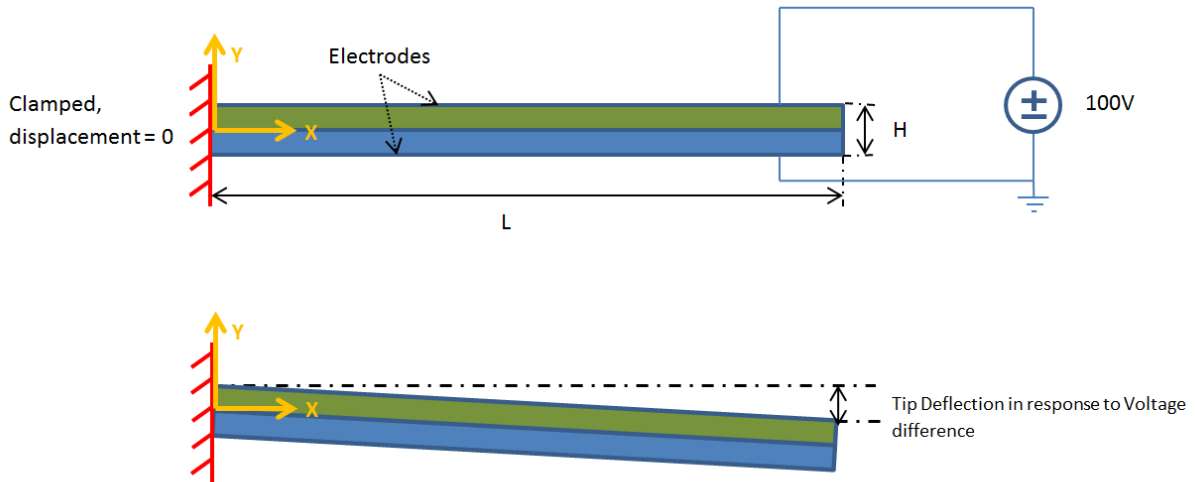
This figure is not to scale; the actual thickness/length ratio is 100 so the beam is very slender. When a voltage is applied between the lower and upper surfaces of the beam, it deflects in the y-direction; one layer shortens and the other layer lengthens. Devices of this type can be designed to provide 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', 'FaceLabels', '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])
% test
```
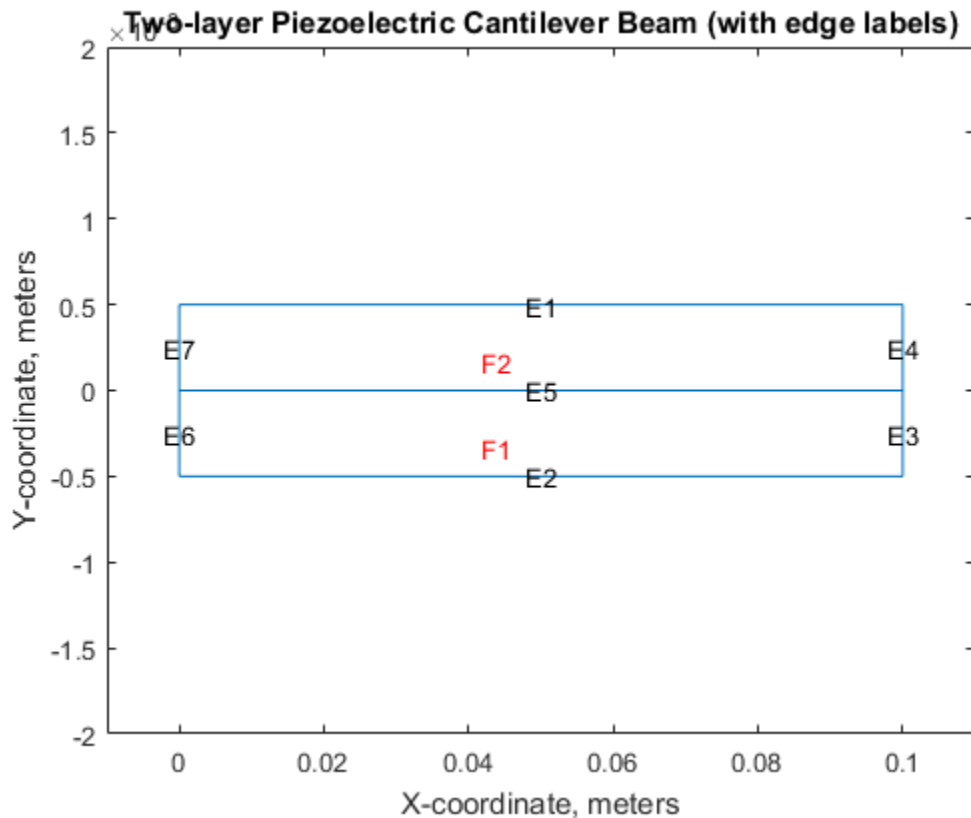
**Material Properties and Coefficient Specification**

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,'mixed','Edge',1,...
                                        'u',V,...
                                        'EquationIndex',3);
% Set the voltage (solution component 3) on the bottom edge to zero.
voltBot = applyBoundaryCondition(model,'mixed','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,'mixed','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
```
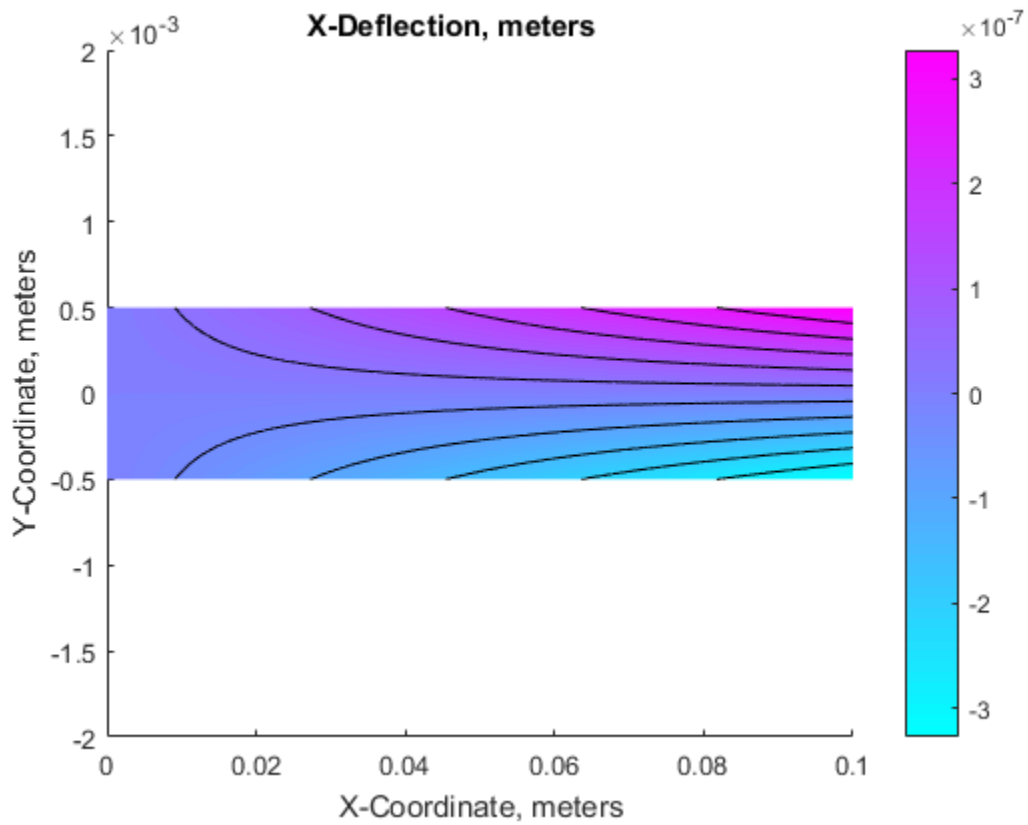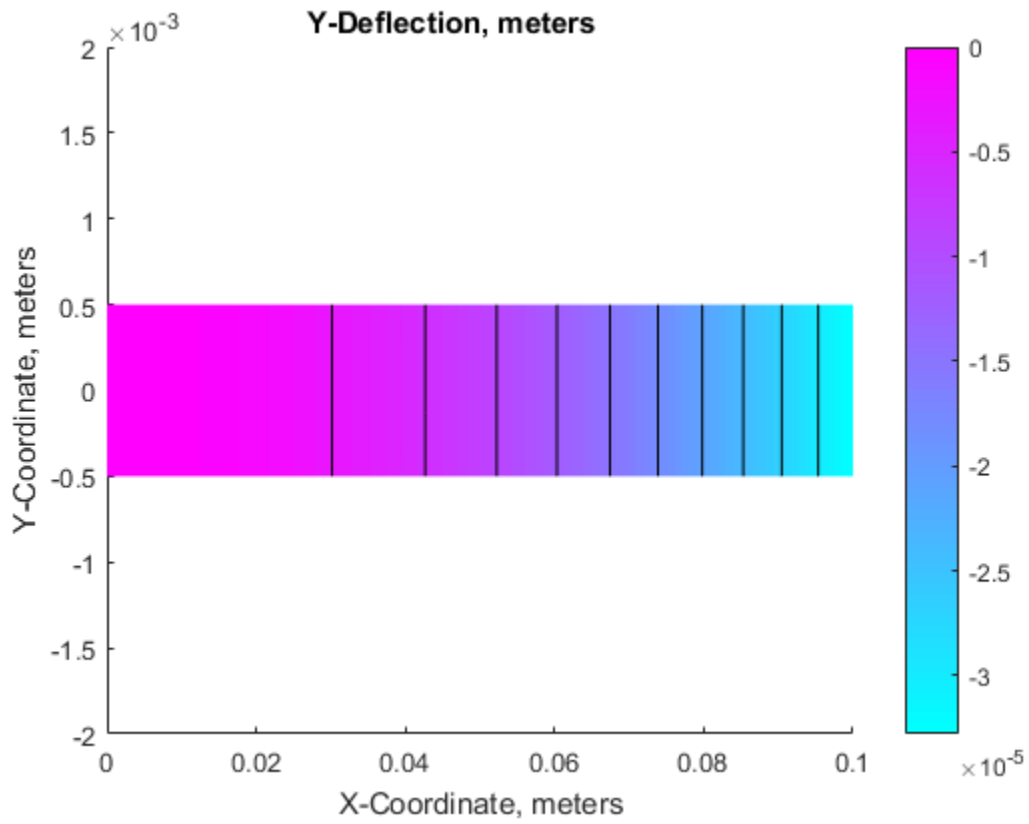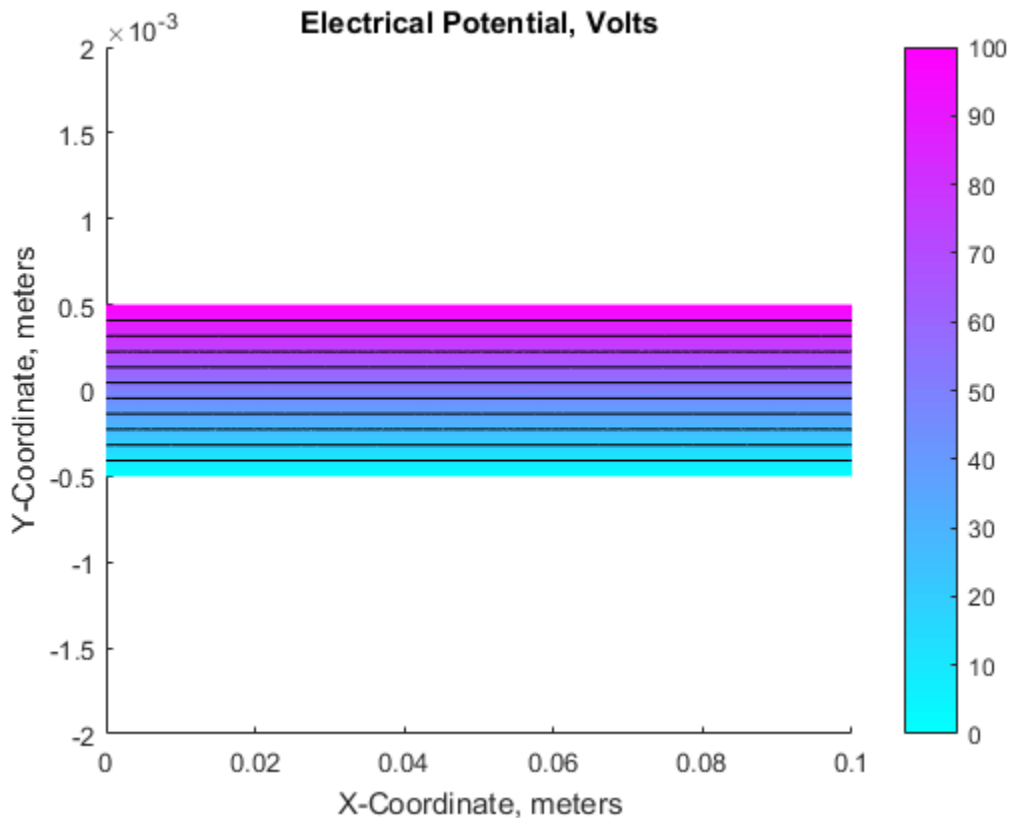
### Analytical Solution

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.

# Dynamics of a Damped Cantilever Beam

This example shows how to include damping in the transient analysis of a simple cantilever beam analyzed with the Partial Differential Equation Toolbox™. The beam is modeled with a plane stress elasticity formulation. The damping model is basic viscous damping distributed uniformly through the volume of the beam. Several transient analyses are performed where the beam is deformed into an initial shape and then released at time, $t = 0$. Analyses with and without damping are considered. Two initial displacement shapes are considered. In the first, the beam is deformed into a shape corresponding to the lowest vibration mode. In the second, the beam is deformed by applying an external load at the tip of the beam. No additional loading is applied in this example so, in the damped cases, the displacement of the beam decays as a function of time due to the damping.

The transient analyses are performed using the PDE Toolbox `hyperbolic` function. One form of this function allows a transient analysis to be performed with the stiffness, mass, and damping matrices and load vectors as input. Typically these matrices and vectors are calculated using other PDE Toolbox functions. That approach will be demonstrated in this example.

A particularly simple way to construct a damping matrix is by using what is commonly referred to as Rayleigh damping. With Rayleigh damping, the damping matrix is defined as a linear combination of the mass and stiffness matrices:

$$D = \alpha M + \beta K$$

It is common to express damping as a percentage of critical damping, $\xi$, for a selected vibration frequency. For a given frequency, $\omega_i$, the following expression relates $\xi$ to $\alpha$ and $\beta$.

$$\xi = \frac{\alpha}{2\omega_i} + \frac{\beta\omega_i}{2}$$

In this example, we will define $\xi = 0.03$ (three percent of critical damping) and $\beta$ equal zero so that $\alpha$ can be calculated as $\alpha = 2 \times .03\omega_i$

This example specifies values of parameters using the imperial system of units. You can replace them with values specified in the metric system. If you do so, ensure that you specify all values throughout the example using the same system.

### Beam Dimensions and Material Properties

The beam is 5 inches long and 0.1 inches thick.

```
width = 5;
height = 0.1;
% The material is steel
E = 3.0e7;
nu = 0.3;
rho = 0.3/386;
% Calculate the coefficient matrix from the material properties
G = E/(2.*(1+nu));
mu = 2.0*G*nu/(1-nu);
```

### Lowest Vibration Frequency From Beam Theory

```
I = height^3/12;
A = height;
% From beam theory, there is a simple expression for the lowest vibration
% frequency of the cantilever beam.
eigValAnalytical = 1.8751^4*E*I/(rho*A*width^4);
freqAnalytical = sqrt(eigValAnalytical)/(2*pi);
```

### Create a PDE Analysis Model

Create a PDEModel with two independent variables to represent the analysis

```
numberOfPDE = 2;
model = createpde(numberOfPDE);
```

### Create the Geometry

Create a simple rectangular geometry

```
gdm = [3;4;0;width;width;0;0;0;height;height];
g = decsg(gdm,'S1',('S1')');
% Plot the geometry and display the edge labels for use in the boundary
% condition definition.
figure;
pdegplot(g,'EdgeLabels','on');
axis equal
title 'Geometry With Edge Labels Displayed'

% Provide the model with a definition of the geometry
geometryFromEdges(model,g);
```

**Geometry With Edge Labels Displayed**
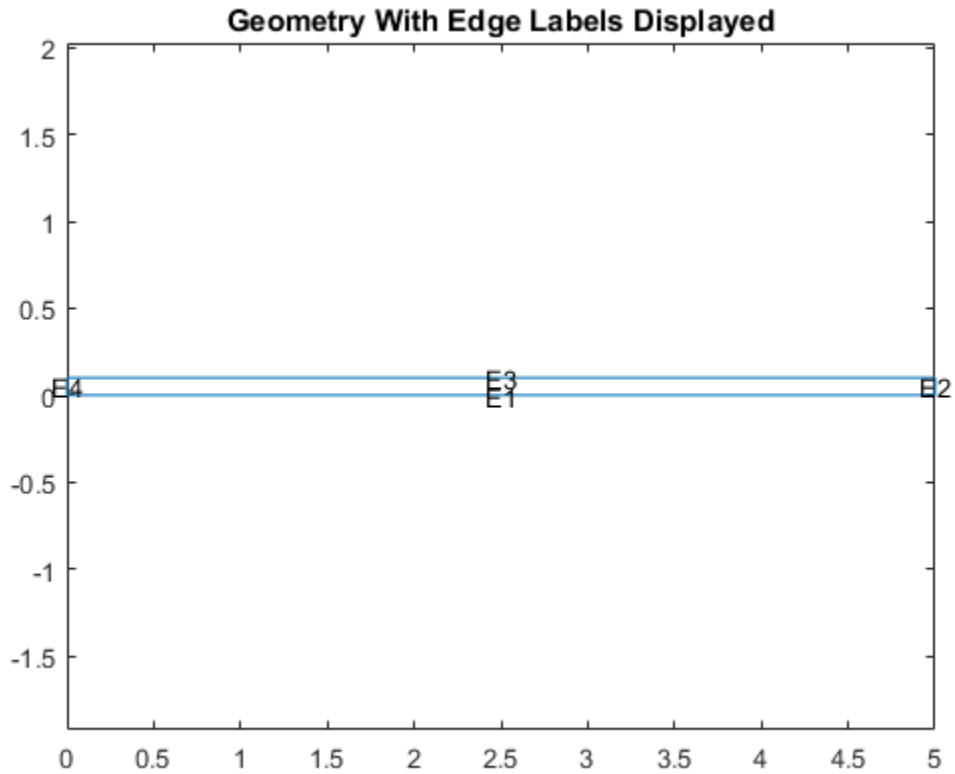


### Equation Coefficients

The equation coefficients are derived from the material properties

```
c = [2*G+mu;0;G;0;G;mu;0;G;0;2*G+mu];
f = [0;0];
a = 0;
m = rho;
specifyCoefficients(model,'m',m,'d',0,'c',c,'a',a,'f',f);
```

### Boundary Conditions

Boundary condition to clamp (displacements equal zero) the left beam-edge

```
applyBoundaryCondition(model,'dirichlet','Edge',4,'u',[0 0]);
```

### Mesh Generation

Define a maximum element size (5 elements through the beam thickness)

```
hmax = height/5;
msh=generateMesh(model,'Hmax',hmax,'MesherVersion','R2013a');
```

### Calculation of Vibration Modes and Frequencies

Use solvepdeeig and then compute the lowest-frequency vibration mode

```
res = solvepdeeig(model, [0,1e6]');
eigenVec = res.Eigenvectors;
eigenVal = res.Eigenvalues;

% Color plot of y-displacement of the lowest-frequency vibration mode
freqNumerical = sqrt(eigenVal(1))./(2*pi);
longestPeriod = 1/freqNumerical;
% Plot the deformed shape of the beam with the displacements scaled
% by an arbitrary factor.
scaleFactor = 20;
figure;
[p,e,t] = meshToPet(msh);
pdeplot(p+scaleFactor*eigenVec',e,t,'XYData',real(eigenVec(:,2)));
title('Lowest Frequency Vibration Mode');
axis equal
xlabel('Inches');
ylabel('Inches');
drawnow

fprintf('Lowest beam frequency (Hz). Analytical = %12.3e, Numerical = %12.3e\n', ...
  freqAnalytical,freqNumerical);
```

```
              Basis= 10,  Time=   0.23,  New conv eig=  3
End of sweep: Basis= 10,  Time=   0.23,  New conv eig=  3
              Basis= 13,  Time=   0.34,  New conv eig=  1
End of sweep: Basis= 13,  Time=   0.36,  New conv eig=  1
Lowest beam frequency (Hz). Analytical =    1.269e+02, Numerical =    1.297e+02
```
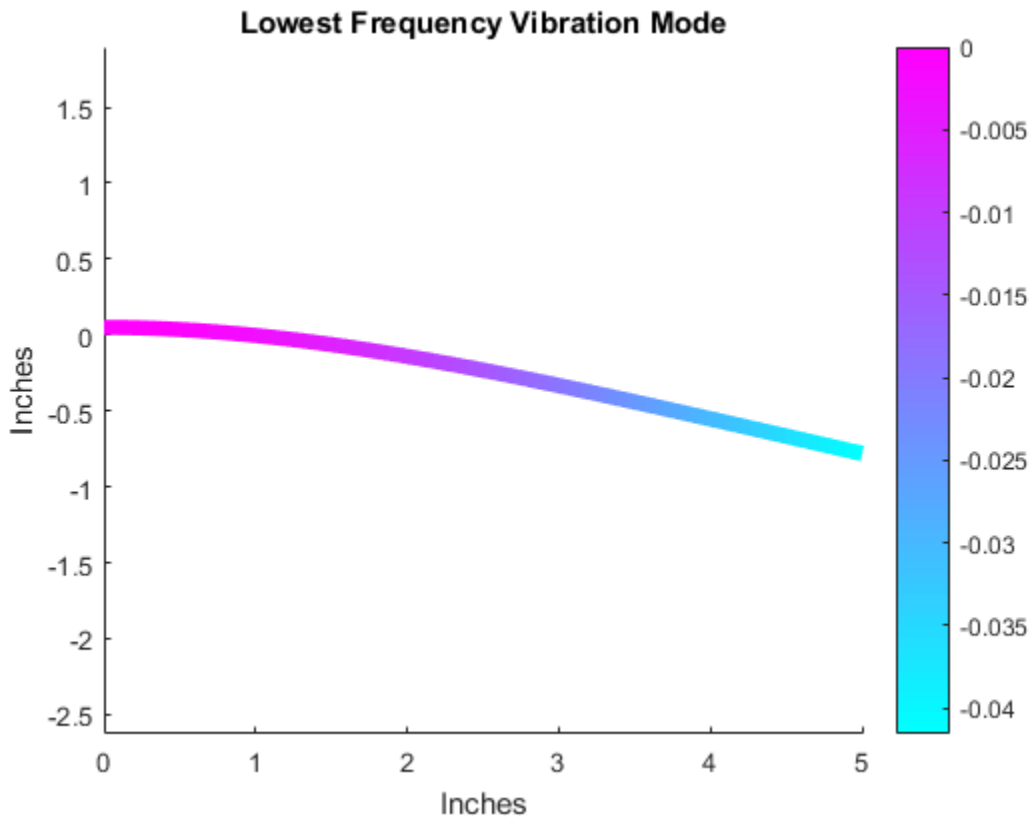
**Transient Analysis, Initial Displacement From First Mode Shape**

In the first two transient analyses, we define an initial displacement in the shape of the lowest vibration mode. By doing this, we convert the PDE to a single ODE with time as the independent variable. The solution to this ODE is the same as that of the classical spring-mass-damper system with a frequency equal the frequency of this vibration mode. Thus we are able to compare the numerical solution with the analytical solution to this well-known problem.

For convenience, we will scale the eigenvector shape so that y-displacement at the tip is .1 inches. This makes the transient history plots simpler.

```
uEigenTip = eigenVec(2,2);
u0TipDisp = .1;
```

```matlab
uO = uOTipDisp/uEigenTip*eigenVec;
%
% First solve the undamped system.
%
% Calculate the solution for three full periods
tlist = O:longestPeriod/100:3*longestPeriod;

%
% Create a function handle that can be used to provide initial conditions
R = createPDEResults(model, uO(:));
ice = icEvaluator(R);

% Set the initial conditions
setInitialConditions(model, @ice.computeIC, O);

tres = solvepde(model,tlist);
titl = 'Initial Displacements from Lowest Eigenvector, Un-Damped Solution';
cantileverBeamTransientPlot(tres,titl);

% As can be seen in the figure below, the displacement at the tip is a sinusoidal
% function of time with amplitude equal to the initial y-displacement.
% This agrees with the solution to the simple spring-mass system.
%
% Now solve the system with damping equal to 3% of critical for this
% lowest vibration frequency.
%

fem = assembleFEMatrices(model);
zeta = .O3;
omega = 2*pi*freqNumerical;
alpha = 2*zeta*omega;
dampmat = alpha*fem.M;
specifyCoefficients(model,'m',m,'d',dampmat,'c',c,'a',a,'f',f);
tres = solvepde(model,tlist);

titl = 'Initial Displacements from Lowest Eigenvector, Damped Solution';
cantileverBeamTransientPlot(tres,titl);
hold on;
plot(tlist,uOTipDisp*exp(-zeta*omega*tlist),'Color','r');
legend('PDE','ODE Amplitude Decay','Location','southeast');
%
% The figure below shows the y-displacement at the tip as a function of time.
% Superimposed on this plot is a second curve which shows the envelope
```

```
% of the maximum amplitude as a function of time, calculated from the
% solution to the single degree-of-freedom ODE. As expected, the PDE
% solution agrees well with the analytical solution.
```
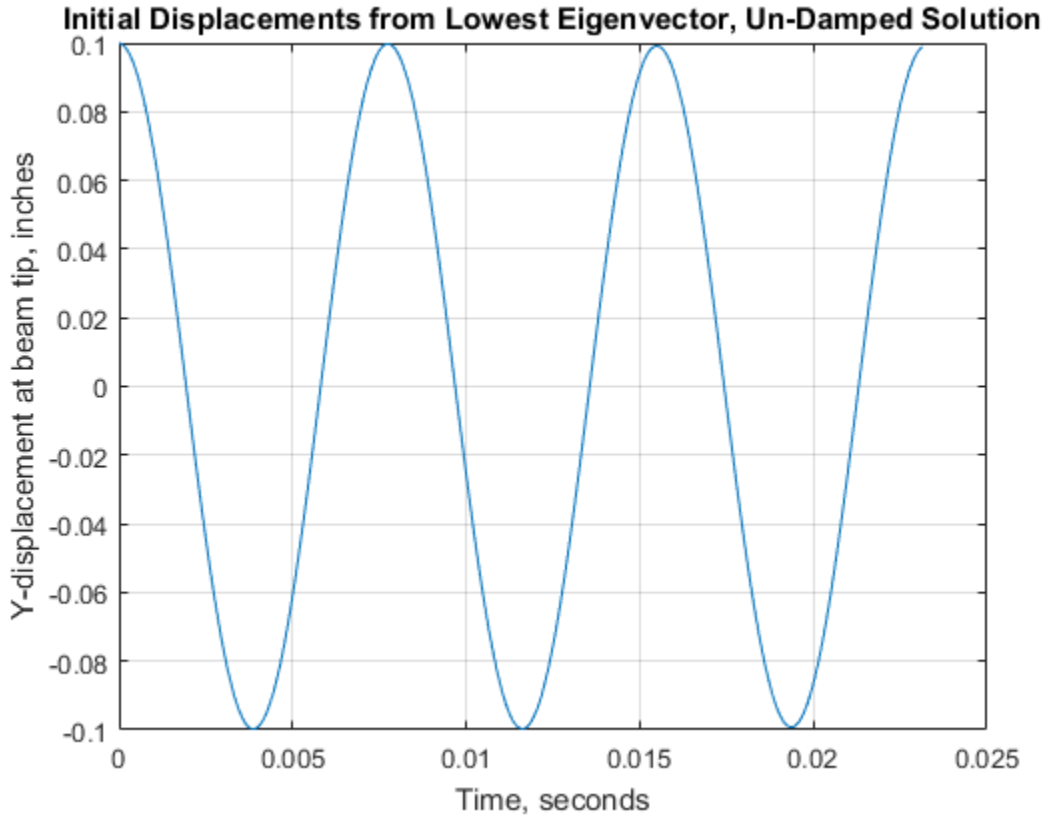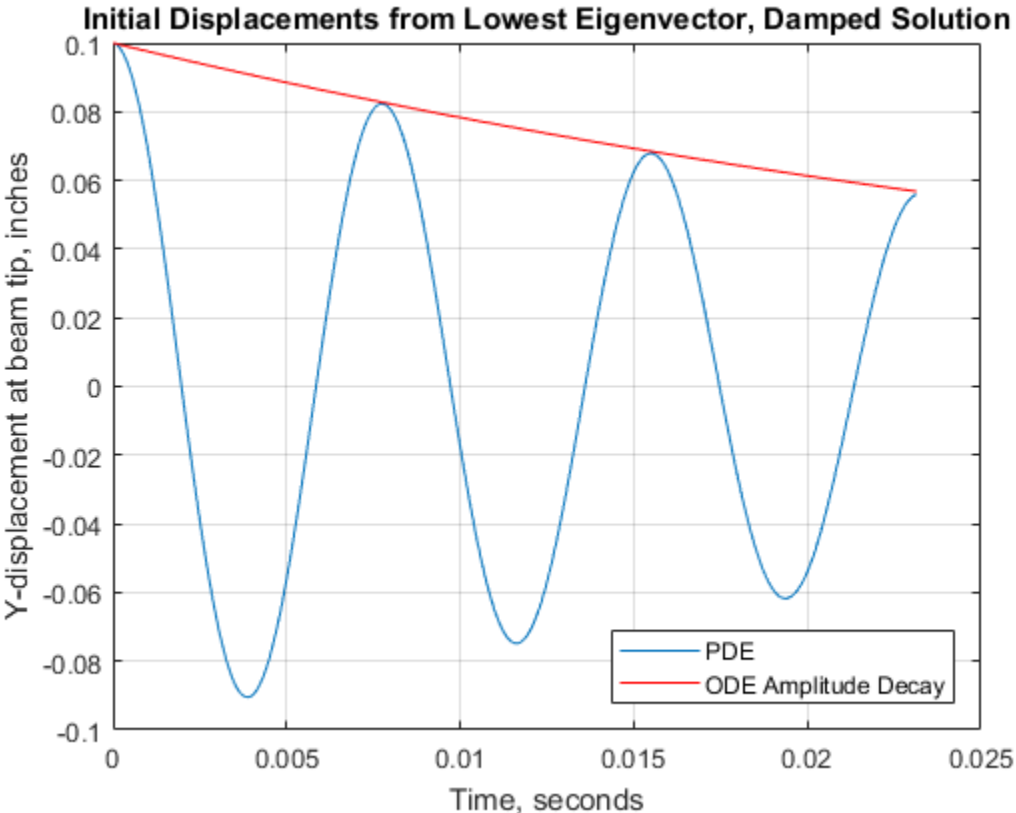
**Transient Analysis, Initial Displacement From Static Solution**

It would be very unusual for a structure to be loaded such that the displacement is equal to a multiple of one of it's vibration modes. In this more realistic example, we solve the transient equations with the initial displacement shape calculated from the static solution of the cantilever beam with a vertical load at the tip.

Perform a static analysis with vertical tip load equal one. Follow the model building steps as before.

```
P = 1.0;
pdeTipLoad = createpde(2);
pg = geometryFromEdges(pdeTipLoad,g);
% Specify the equation to solve
```

```matlab
specifyCoefficients(pdeTipLoad,'m',O,'d',O,'c',c,'a',a,'f',f);
tipLoad = applyBoundaryCondition(pdeTipLoad,'neumann','Edge',2,'g',[O P/height]);
clampedEdge = applyBoundaryCondition(pdeTipLoad,'dirichlet','Edge',4,'u',[O,O]);
msh=generateMesh(pdeTipLoad,'Hmax',hmax,'MesherVersion','R2013a');

statres = solvepde(pdeTipLoad);

% To make comparison with the eigenvector case clearer, we will also scale
% this static solution so that the maximum y-displacement at the tip equals
% .1 inches.

u = statres.NodalSolution;
uEigenTip = u(2,2);
uOTipDisp = 0.1;
uO = uOTipDisp/uEigenTip*u;

%
% Calculate the un-damped solution with the initial displacement from
% the static analysis.
%

specifyCoefficients(model, 'm', m, 'd', O, 'c', c, 'a', a, 'f', f);

% Set the initial conditions
R = createPDEResults(model, uO(:));
ice = icEvaluator(R);
setInitialConditions(model, @ice.computeIC, O);
tres = solvepde(model,tlist);
titl = 'Initial Displacements from Static Solution, Un-Damped Solution';
cantileverBeamTransientPlot(tres,titl);
% We see in the figure below that the displacement is no longer a pure sin wave.
% The static solution that we are using as the initial conditions is similar
% to the lowest eigenvector but higher-frequency modes are also contributing
% to the transient solution.

%
% Calculate the damped solution with the initial displacement from
% the static analysis. The damping matrix is the same as that used
% in the eigenvector case, above.
%
specifyCoefficients(model, 'm', m, 'd', dampmat, 'c', c, 'a', a, 'f', f);
tres = solvepde(model,tlist);

titl = 'Initial Displacements from Static Solution, Damped Solution';
```

```matlab
cantileverBeamTransientPlot(tres,titl);
hold on;
plot(tlist,uOTipDisp*exp(-zeta*omega*tlist),'Color','r');
legend('PDE','ODE Amplitude Decay','Location','southeast');
%
% In the figure below, we plot the tip displacement from this solution as
% a function of time. Again we superimpose a curve of the damped amplitude
% as a function of time obtained from an analytical solution to the
% single degree-of-freedom ODE. Because the initial condition
% differs from the lowest eigenvector,
% this analytical solution only approximates the amplitude of the PDE
% solution.
%
```
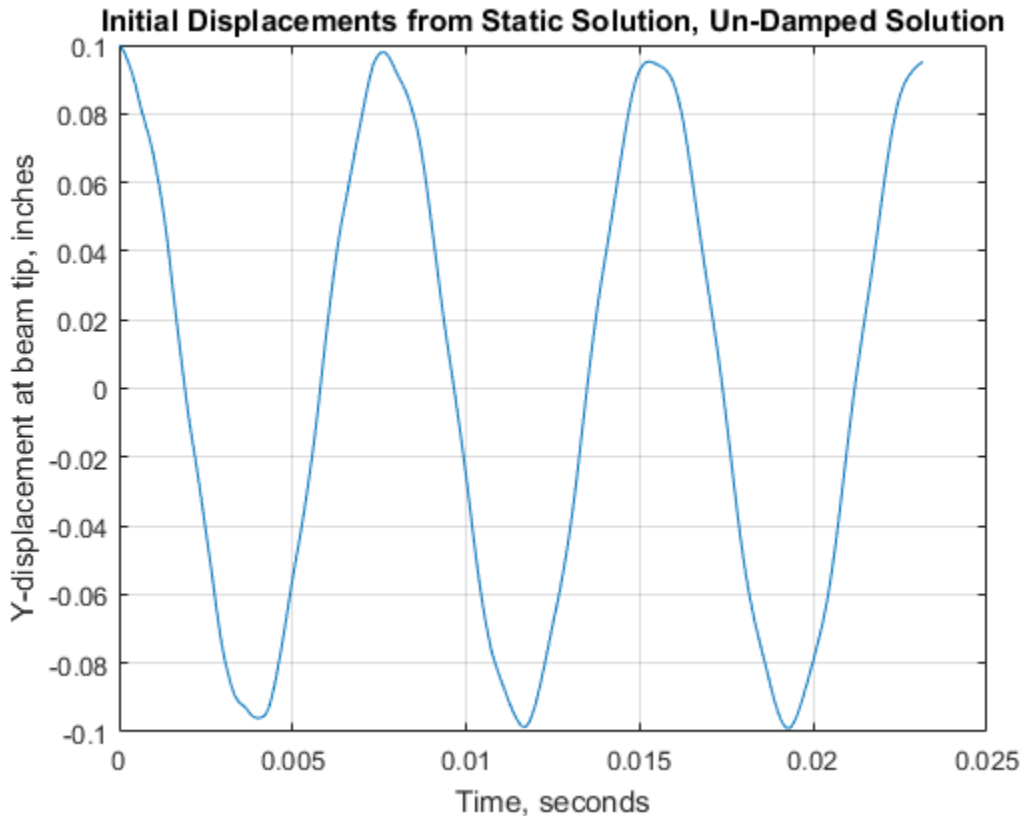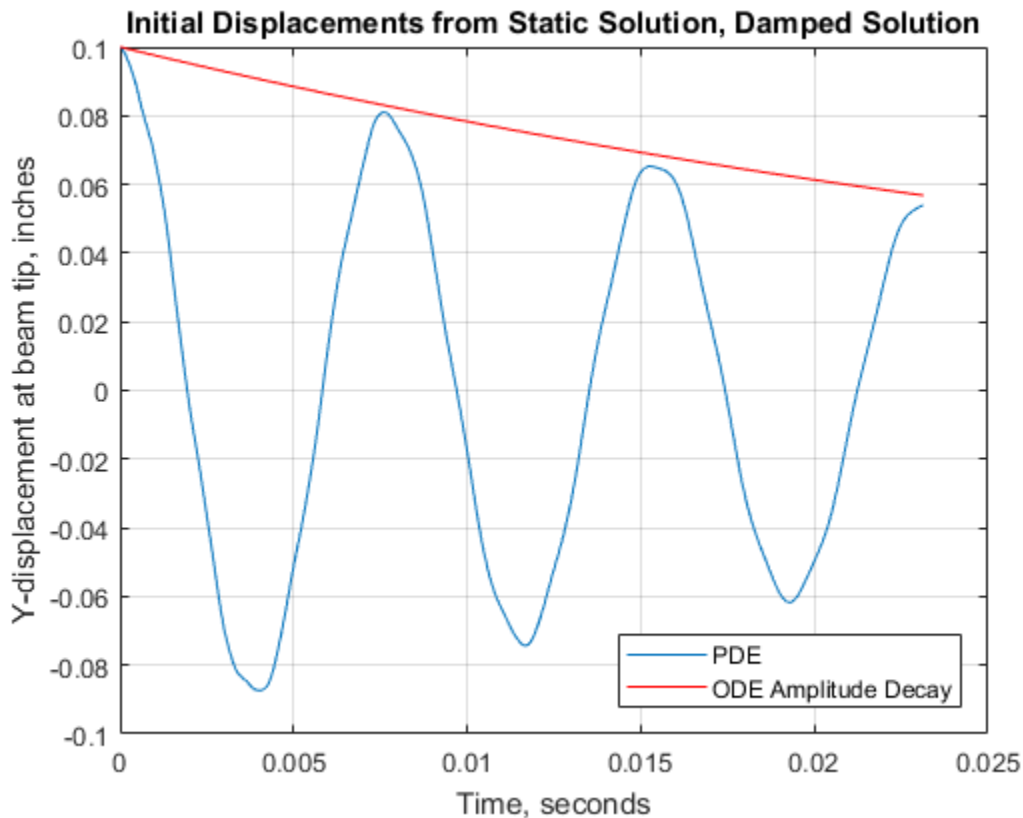
**Utility Plot Function**

Utility function for creating the plots of tip y-displacement as a function of time.

type cantileverBeamTransientPlot.m

```
function cantileverBeamTransientPlot( tdres, titl )
%CANTILEVERBEAMTRANSIENTPLOT Plot y-displacement at the beam tip
%   tdres - Time-dependent results object representing displacements as a function of t
%   titl plot title

% Copyright 2013-2015 The MathWorks, Inc.
```

```
tlist = tdres.SolutionTimes;
uu = tdres.NodalSolution;
utip = uu(2,2,:);
figure; plot(tlist, utip(:)); grid on;
title(titl); xlabel('Time, seconds');
ylabel('Y-displacement at beam tip, inches');
drawnow;
end
```

### Function Handle for Specifying Initial Condition (IC)

The evaluation function returns the value of the IC at any point within the mesh. A results object represents the values and the interpolation function that it provided is used to compute the ICs.

```
type icEvaluator.m
```

```
classdef icEvaluator
% icEvaluator Evaluates Initial Conditions data at requested locations
% ICE = icEvaluator(R) Creates an initial conditions evaluator from a
% results object. The evaluator provides a function that can be called at
% specific locations within the geometry. This class customized to represent
% a stationary solution.
%
% icEvaluator methods:
%    computeIC - Computes ICs at locations within the geometric domain
%
% Copyright 2015 The MathWorks, Inc.
    properties
        Results;
    end

    methods
        function obj = icEvaluator(R)
            obj.Results = R;
        end

        function ic = computeIC(obj,locations)
            is2d = size(obj.Results.Mesh.Nodes, 1) == 2;
            if is2d
                querypts = [locations.x; locations.y];
```

```
                    else
                         querypts = [locations.x; locations.y; locations.z];
                    end
               numeqns = size(obj.Results.NodalSolution,2);
               if numeqns == 1
                   ic = interpolateSolution(obj.Results, querypts);
               else
                   ic = zeros(numeqns, numel(locations.x));
                   for i = 1:numeqns
                       ic(i,:) = interpolateSolution(obj.Results, querypts, i);
                   end
               end
           end
       end
end
```

# Dynamic Analysis of a Clamped Beam

This example shows the Partial Differential Equation Toolbox™ analysis of the dynamic behavior of a beam clamped at both ends and loaded with a uniform pressure load. The pressure load is suddenly applied at time equal zero and the magnitude is high enough to produce deflections on the same order as the beam thickness.

Accurately predicting this type of behavior requires a geometrically-nonlinear formulation of the elasticity equations. One of the main purposes of this example is to show how PDE Toolbox can be used to solve a problem in nonlinear elasticity. The analysis will be performed with both linear and nonlinear formulations to demonstrate the importance of the latter.

This example specifies values of parameters using the imperial system of units. You can replace them with values specified in the metric system. If you do so, ensure that you specify all values throughout the example using the same system.

### Equations

This section describes the equations of geometrically nonlinear elasticity. One approach to handling the large deflections is to consider the elasticity equations in the deformed position. However, PDE Toolbox formulates the equations based on the original geometry. This motivates using a Lagrangian formulation of nonlinear elasticity where stresses, strains, and coordinates refer to the original geometry.

The Lagrangian formulation of the equilibrium equations can be written

$$\rho \ddot{u} - \nabla \cdot (F \cdot S) = f$$

where $\rho$ is the material density, $u$ is the displacement vector, $F$ is the deformation gradient, $S$ is the second Piola-Kirchoff stress tensor, and $f$ is the body force vector. This equation can also be written in the following tensor form:

$$\rho \ddot{u}_i - \frac{\partial}{\partial x_j} \left( \left( \frac{\partial u_i}{\partial x_k} + \delta_{ik} \right) S_{kj} \right) = f_i$$

Although large deflections are accounted for in this formulation, it is assumed that the strains remain small so that linear elastic constitutive relations apply. Also, the material

is assumed to be isotropic. For the 2D plane stress case, the constitutive relations may be written in matrix form:

$$
\left\{ \begin{array}{c} S_{11} \\ S_{22} \\ S_{12} \end{array} \right\} = \left[ \begin{array}{ccc} C_{11} & C_{12} & \\ C_{12} & C_{22} & \\ & & 2G_{12} \end{array} \right] \left\{ \begin{array}{c} E_{11} \\ E_{22} \\ E_{12} \end{array} \right\}
$$

where $E_{ij}$ is the Green-Lagrange strain tensor defined as

$$
E_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} + \frac{\partial u_k}{\partial x_i} \frac{\partial u_k}{\partial x_j} \right)
$$

For an isotropic material

$$
C_{11} = C_{22} = \frac{E}{1 - \nu^2}
$$

$$
C_{12} = \frac{E\nu}{1 - \nu^2}
$$

$$
G_{12} = \frac{E}{2(1 + \nu)}
$$

where $E$ is Young's modulus and $\nu$ is Poisson's ratio.

Readers who are interested in more details about the Lagrangian formulation for nonlinear elasticity can consult, for example, reference 1.

The equations presented above completely define the geometrically nonlinear plane stress problem. The work required to convert them to a form acceptable to PDE Toolbox is considerably simplified by using the MATLAB Symbolic Math Toolbox. Symbolic Math Toolbox can perform the necessary algebraic manipulations and output a MATLAB function defining the c-coefficient that can be passed to PDE Toolbox functions. This function, `cCoefficientLagrangePlaneStress`, is shown in the appendix below.

### Create the PDE Model

```
N = 2; % Two PDE in plane stress elasticity
```

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

### Define the Geometry

```
blength = 5; % Beam length, in.
height = .1; % Thickness of the beam, in.
```

A drawing of the clamped beam is shown in the figure below.
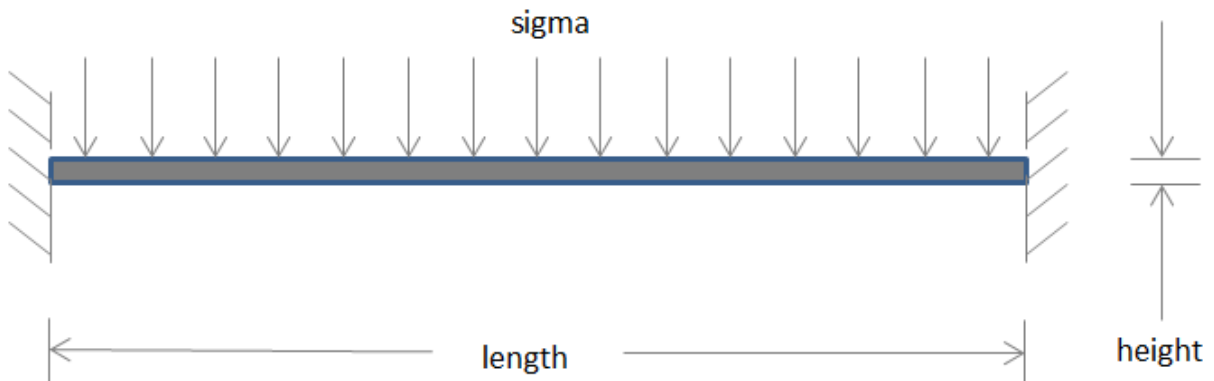


Since the beam geometry and loading are symmetric about the beam center($x =$ blength/2), the model can be simplified by considering only the right-half of the beam.

```
l2 = blength/2;
h2 = height/2;

% Create the edges of the rectangle representing the beam with these
% two statements:
rect = [3 4 0 l2 l2 0 -h2 -h2  h2 h2]';
g = decsg(rect,'R1',('R1')');

% The geometryFromEdges function creates a geometry object from the edges
% and stores it within the model.
pg = geometryFromEdges(model,g);
```

Plot the geometry and display the edge labels. The edge labels are needed for edge identification when applying boundary conditions.

```
figure
```

```
pdegplot(g,'EdgeLabels','on');
title('Geometry With Edge Labels Displayed');
axis([-.1 1.1*l2 -5*h2 5*h2]); % Scale the plot so the labels are viewable
```

**Geometry With Edge Labels Displayed**

```
 0.25

  0.2

 0.15

  0.1

 0.05                              E3

    0  E4                                                    E2

-0.05                              E1

 -0.1

-0.15

 -0.2

-0.25
        0       0.5       1       1.5       2       2.5
```
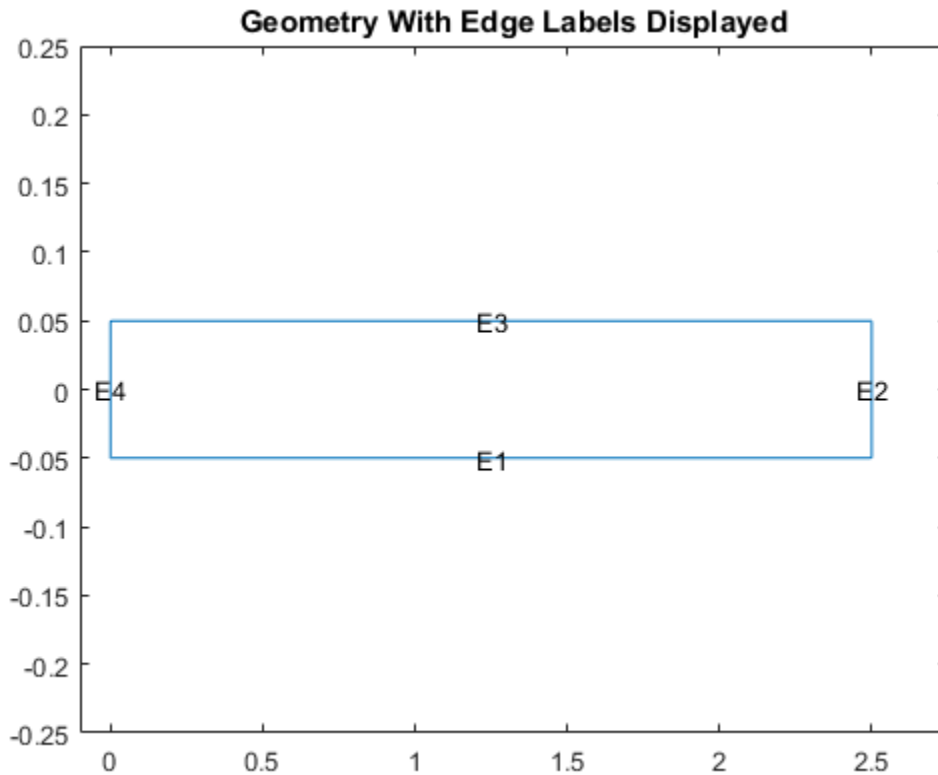
### Specify Equation Coefficients

Derive the equation coefficients using the material properties. For the linear case, the c-coefficient matrix is constant

```
E = 3.0e7; % Young's modulus of the material, lbs/in^2
gnu = .3; % Poisson's ratio of the material
rho = .3/386; % Density of the material
G = E/(2.*(1+gnu));
```

```
mu = 2*G*gnu/(1-gnu);
c = [2*G+mu; 0; G;   0; G; mu; 0;  G; 0; 2*G+mu];
f = [0 0]'; % No body forces
specifyCoefficients(model, 'm', rho, 'd', 0, 'c', c, 'a', 0, 'f', f);
```

### Apply the Boundary Conditions

Symmetry condition is x-displacement equal zero at left edge.

```
symBC = applyBoundaryCondition(model,'mixed','Edge',4,'u',0,'EquationIndex',1);
```

x- and y-displacements equal zero along right edge.

```
clampedBC = applyBoundaryCondition(model,'dirichlet','Edge',2,'u',[0 0]);
```

Apply a constant vertical stress along the top edge.

```
sigma = 2e2;
presBC = applyBoundaryCondition(model,'neumann','Edge',3,'g',[0 sigma]);
```

### Set the Initial Conditions

Zero initial displacements and velocities

```
setInitialConditions(model,0,0);
```

### Create the Mesh

Create a mesh with approximately eight elements through the thickness of the beam.

```
generateMesh(model,'Hmax',height/8,'MesherVersion','R2013a');
```

### Linear Solution

Set up the analysis timespan

```
tfinal = 3e-3; % Final time in the analysis
tlist = linspace(0,tfinal,100); % Save the output at 100 time points

% Compute the time-dependent solution
result = solvepde(model,tlist);
```

Interpolate the solution at the geometry center for the y-component (component 2) at all times.

```
xc = 1.25;
yc = 0;
u4Linear = interpolateSolution(result,xc,yc,2,1:length(tlist));
```

### Specify Equation Coefficients for Nonlinear Solution

The function `cCoefficientLagrangePlaneStress` takes the isotropic material properties and location and state structures, and returns a c-matrix for a nonlinear plane-stress analysis. Small strains are assumed; i.e. E and $\nu$ are independent of the solution. PDE Toolbox calls user-defined coefficient functions with the arguments location and state. The function `cCoefficientLagrangePlaneStress` expects the arguments E, gnu, location, state. `c` is defined below as an anonymous function to provide an interface between these two function signatures. (The function `cCoefficientLagrangePlaneStress` can be used with any geometric nonlinear plane stress analysis of a model made from an isotropic material.)

```
c  = @(location, state) cCoefficientLagrangePlaneStress(E,gnu,location,state);

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

### Nonlinear Solution

Compute the time-dependent solution.

```
result = solvepde(model,tlist);
```

As before, interpolate the solution at the geometry center for the y-component (component 2) at all times.

```
u4NonLinear = interpolateSolution(result,xc,yc,2,1:length(tlist));
```

### Plot Solutions

The figure below shows the y-deflection at the center of the beam as a function of time. The nonlinear analysis computes displacements that are substantially less than the linear analysis. This "stress stiffening" effect is also reflected in the higher oscillation frequency from the nonlinear analysis.

```
figure
plot(tlist,u4Linear(:),tlist,u4NonLinear(:));
legend('Linear','Nonlinear');
title 'Deflection at Beam Center'
xlabel 'Time, seconds'
```

```
ylabel 'Deflection, inches'
grid on
```
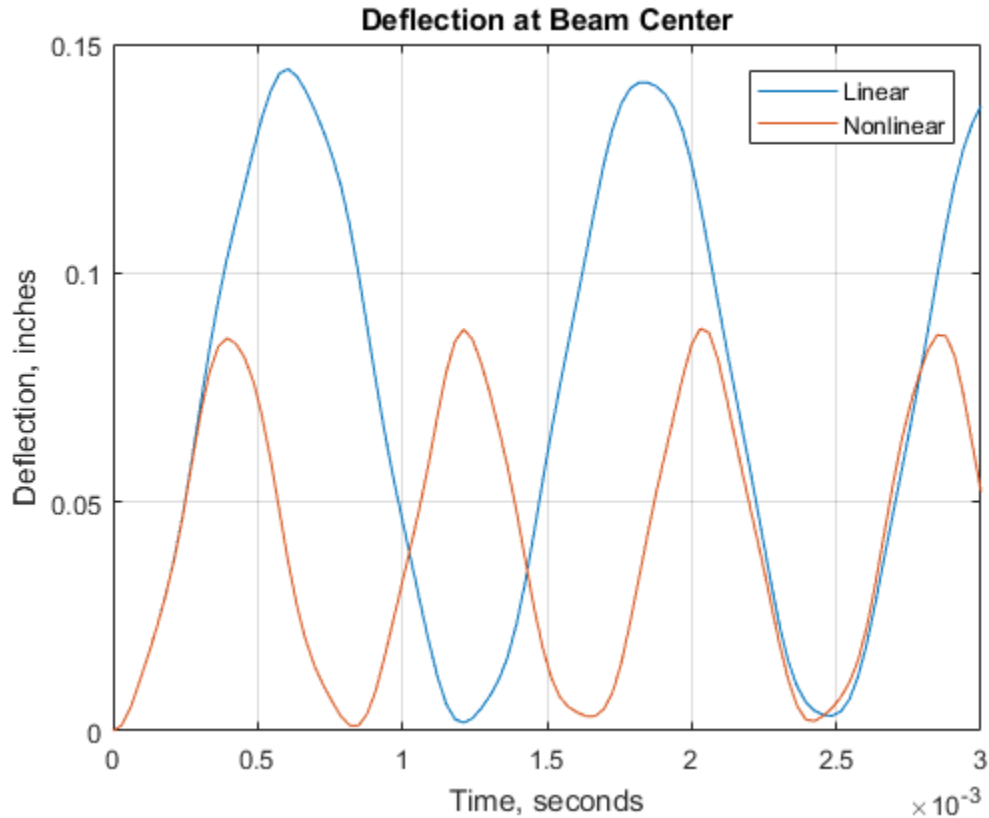


**References**

1   Malvern, Lawrence E., Introduction to the Mechanics of a Continuous Medium, Prentice Hall, 1969.

**Appendix - Nonlinear C-Coefficient Function**

The function cCoefficientLagrangePlaneStress calculates the c-coefficient matrix for a large displacement Lagrangian plane stress formulation.

```
type cCoefficientLagrangePlaneStress


function c = cCoefficientLagrangePlaneStress(E, nu, loc, state)
%cCoefficientLagrangePlaneStress Calculate c-coefficient for nonlinear plane stress
% Calculate the c-coefficient for a geometrically nonlinear Lagrangian formulation
% of plane stress elasticity. The strain measure is the Green-Lagrange strain
% tensor. The stress is the second Piola-Kirchoff stress tensor. The material
% is assumed to be isotropic with linear behavior (Hooke's law applies).
%
% E  - Young's modulus of the linear isotropic material
% nu - Poisson's ratio for the material
% p  - matrix of point (node) locations
% t  - element connectivity matrix
% u  - current displacement vector

%    This function was generated by the Symbolic Math Toolbox version 6.0.
%    31-Jan-2014 09:50:09

% Copyright 2014-2015 The MathWorks, Inc.


ux = reshape(state.ux,2,[]);
uy = reshape(state.uy,2,[]);



dudx=ux(1,:);
dvdx=ux(2,:);
dudy=uy(1,:);
dvdy=uy(2,:);

% if(~isempty(u))
%   [ux,uy] = pdegrad(p,t,u);
%   dudx=ux(1,:); dudy=uy(1,:); dvdx=ux(2,:); dvdy=uy(2,:);
% else
%   dudx = zeros(1, size(t,2)); dudy=dudx; dvdx=dudx; dvdy=dudx;
% end

t4 = 1/(nu^2-1);
t6 = 1/(1+nu);
t7 = E*dudy.*t4*.25;
t8 = dudx+1.0;
t9 = E*dudy.*t4.*t8*.25;
t10 = dvdy+1.0;
```

```
t11 = t7+t9-E*dvdx.*t6.*t10*.25;
t12 = dvdy.*2.0;
t13 = dudx.^2;
t14 = dudy.^2;
t15 = dvdy.^2;
t16 = dvdx.^2;
t17 = E*dvdx.*t4.*(1.0./2.0);
t18 = E*dudx.*dvdx.*t4*.25;
t19 = t17+t18-E*dudy.*t6*.25-E*dudy.*dvdy.*t6.*(1.0./8.0);
t20 = E*dudy.*dvdx.*nu.*t4*.25;
t21 = t20-E*t6.*(1.0./2.0)-E*dudx.*t6*.25-E*dvdy.*t6*.25-E*dudx.*dvdy.*t6.*(1.0./8.0);
t22 = dudx.*2.0;
t23 = dvdy+2.0;
t24 = nu-1.0;
t25 = E*nu.*t4;
t26 = E*dudx.*nu.*t4.*(1.0./2.0);
t27 = E*dvdy.*nu.*t4.*(1.0./2.0);
t28 = E*dudx.*dvdy.*nu.*t4*.25;
t29 = t25+t26+t27+t28-E*dudy.*dvdx.*t6.*(1.0./8.0);
t30 = E*dudy.*t4.*t23.*(1.0./8.0);
t31 = E*dudy.*dvdy.*t4.*(1.0./8.0);
t32 = t7+t30+t31-E*dvdx.*t6.*(1.0./8.0)-E*dvdx.*t4.*t8.*t24.*(1.0./8.0);
t33 = dudy.*2.0;
t34 = dvdx.*2.0;
t35 = dudx.*dudy.*2.0;
t36 = dvdx.*dvdy;
t37 = t33+t34+t35+t36;
t38 = 1.0./t24;
t39 = E*dvdx.*t23.*t38.*(1.0./8.0);
t40 = t39-E*t6.*t37.*(1.0./8.0);
out1 = [E*t4.*(dudx.*6.0+t13.*2.0+t14+t16+4.0)*.25+E*nu.*t4.*(t12+t15)*.25;
  t11;
  t19;
  t29;
  t11;
  E*t4.*(t12+t13+t14.*2.0+t15+t22+2.0)*.25+E*nu.*t4.*(t16-2.0)*.25;
  t21;
  t32;
  t19;
  t21;
  E*t4.*(t12+t13+t15+t16.*2.0+t22+2.0)*.25+E*nu.*t4.*(t14-2.0)*.25;
  t40;
  t29;
  t32;
```

```
  t40;
  E*t4.*(dvdy.*6.0+t14+t15.*2.0+t16+4.0)*.25+E*nu.*t4.*(t13+t22)*.25];

c = -out1([1 5 6 9 10 13 14 11 15 16], :);

end
```

# Deflection Analysis of a Bracket

This example shows how to analyze an idealized 3-D mechanical part under an applied load using Finite Element Analysis (FEA). The objective of the analysis is to determine the maximum deflection caused by the load.

### Create a PDE Model

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, PDE coefficients, 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);
```

### Construct the Geometry

Import an STL file of a simple bracket model using the `importGeometry` function. This function reconstructs the faces, edges and vertices of the model. It can merge some faces and edges, so the numbers can differ from those of the parent CAD model.

```
importGeometry(model,'BracketWithHole.stl');
```

Plot the geometry and turn on face labels. You will need the face labels to define the boundary conditions.

```
figure
pdegplot(model,'FaceLabels','on')
view(30,30);
title('Bracket with Face Labels')
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```
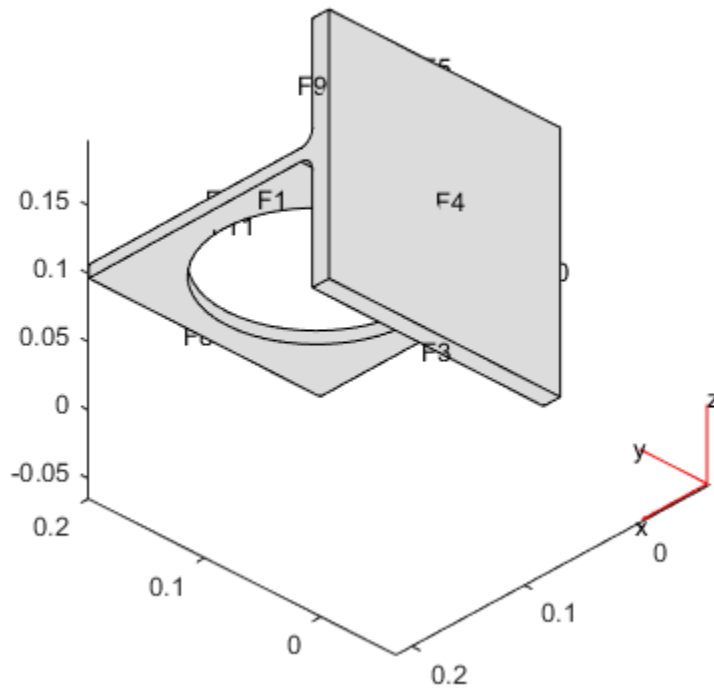
**Bracket with Face Labels**
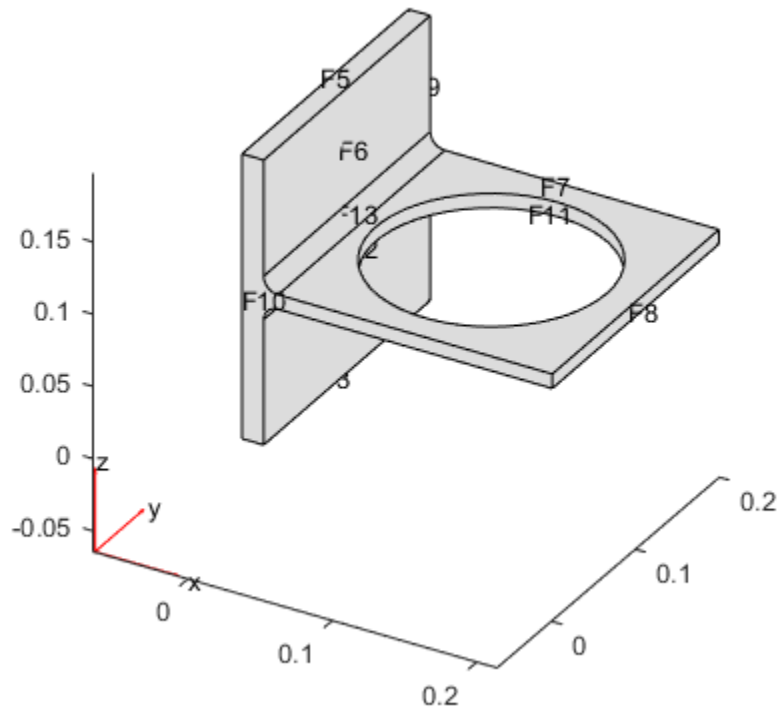
**Bracket with Face Labels, Rear View**



### Examine the Elasticity Equations

As explained in 3-D Linear Elasticity Equations in Toolbox Form, you can express the elasticity equations for the deflection $u$ of a linear isotropic solid as a three-component system

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

where $f = [0; 0; 0]$ when there are no body forces, and $c$ is the symmetric matrix

$$
c = \begin{bmatrix}
c_1 & 0 & 0 & 0 & c_{12} & 0 & 0 & 0 & c_{12} \\
\bullet & G & 0 & G & 0 & 0 & 0 & 0 & 0 \\
\bullet & \bullet & G & 0 & 0 & 0 & G & 0 & 0 \\
\bullet & \bullet & \bullet & G & 0 & 0 & 0 & 0 & 0 \\
\bullet & \bullet & \bullet & \bullet & c_1 & 0 & 0 & 0 & c_{12} \\
\bullet & \bullet & \bullet & \bullet & \bullet & G & 0 & G & 0 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & G & 0 & 0 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & G & 0 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & c_1
\end{bmatrix}.
$$

The $\bullet$ symbol means the entry is symmetric. Here $E$ is the elastic modulus, $\nu$ is Poisson's ratio, and

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

- $c_1 = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}$

- $c_{12} = c_1 \frac{\nu}{1-\nu}$

You can create the c coefficient for a linear isotropic solid using the `elasticityC3D` function, which is included in your software (see 3-D Linear Elasticity Equations in Toolbox Form).

### Define the Coefficients in Toolbox Syntax

Define the elastic modulus of steel and Poisson's ratio for this material.

```
E = 200e9; % elastic modulus of steel in Pascals
nu = 0.3; % Poisson's ratio
```

Incorporate these coefficients in toolbox syntax.

```
c = elasticityC3D(E,nu);
a = 0;
f = [0;0;0]; % Assume all body forces are zero
```

### Specify PDE Coefficients

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

**Define the Boundary Conditions**

The problem has two boundary conditions: the back face is immobile and the front face has an applied load. All other boundary conditions, by default, are free Neumann boundaries.

Constrain all three displacements to 0 on the back face, face 4.

```
applyBoundaryCondition(model,'dirichlet','Face',4,'u',[0,0,0]);
```

Apply a distributed load in the negative $z$-direction to the front face, face 8.

```
distributedLoad = 1e4; % Applied load in Pascals
applyBoundaryCondition(model,'neumann','Face',8,'g',[0,0,-distributedLoad]);
```

**Create a Mesh**

Create a mesh that uses 10-node tetrahedral elements with quadratic interpolation functions. This element type is significantly more accurate than the linear interpolation (four-node) elements, particularly in elasticity analyses that involve bending.

```
bracketThickness = 1e-2; % Thickness of horizontal plate with hole, meters
hmax = bracketThickness; % Maximum element length for a moderately fine mesh
generateMesh(model,'Hmax',hmax,'GeometricOrder','quadratic');
figure
pdeplot3D(model)
title('Mesh with Quadratic Tetrahedral Elements');
```

## Mesh with Quadratic Tetrahedral Elements
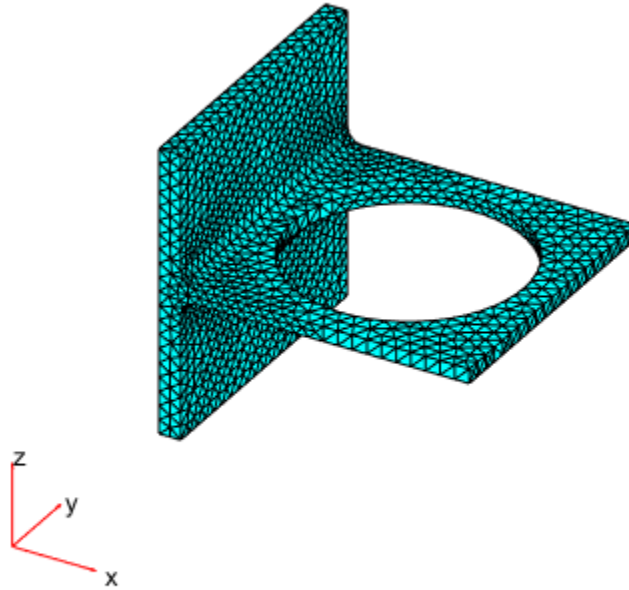


### Calculate the Solution

Use `solvepde` to calculate the solution.

```
result = solvepde(model);
```

### Examine the Solution

For plotting and analysis, extract the `NodalSolution` property.

```
rs = result.NodalSolution;
```

Find the greatest calculated deflection of the bracket. The deflection in the $z$ direction, meaning the third component of the solution.

```
minUz = min(rs(:,3));
fprintf('Largest deflection in the z-direction is %g meters.\n', minUz)
```

```
Largest deflection in the z-direction is -4.46979e-05 meters.
```

**Plot the Solution**

To see the solution, plot the components of the solution vector. The maximum deflections are in the $z$-direction. Because the part and the loading are symmetric, the $x$-displacement and $z$-displacement are symmetric and the $y$-displacement is antisymmetric about the part centerline.

By default, the plotting routine uses the `'jet'` colormap, which has blue as the color representing the lowest value and red representing the highest value. The bracket loading causes face 8 to dip down, so the maximum $z$-displacement appears blue.

```
labels = ['x','y','z'];
for i=1:3
  figure
  pdeplot3D(model,'ColorMapData',rs(:,i))
  title([labels(i),'-displacement'])
end
```

x-displacement

y-displacement
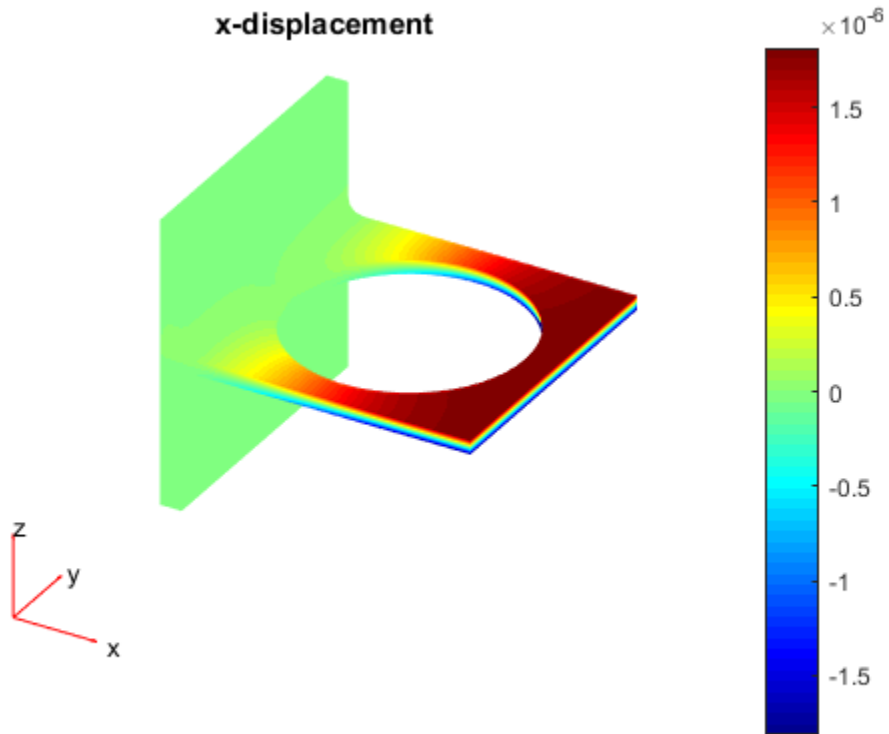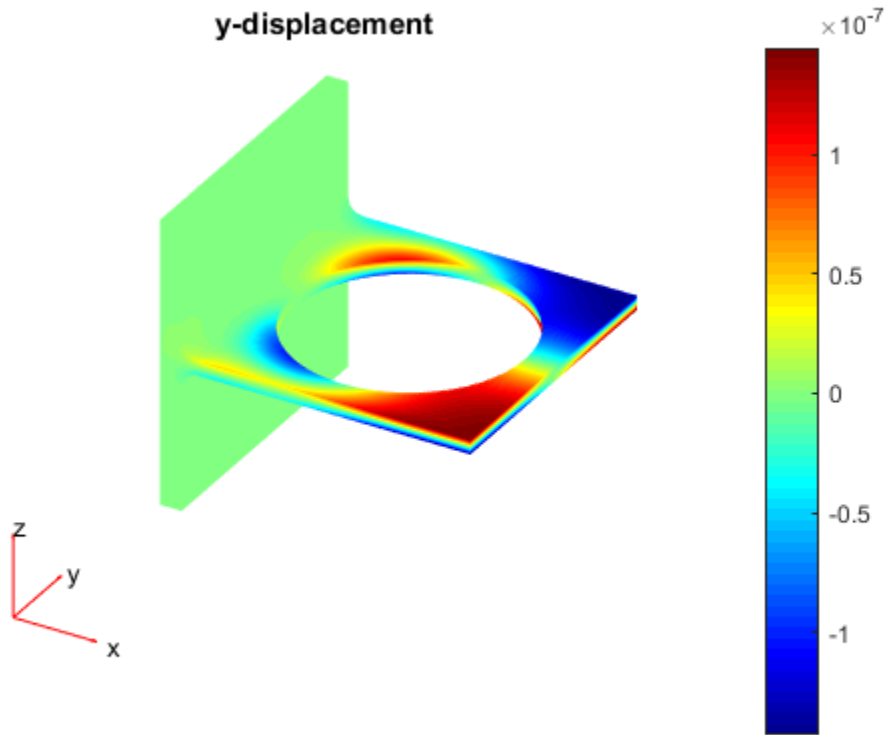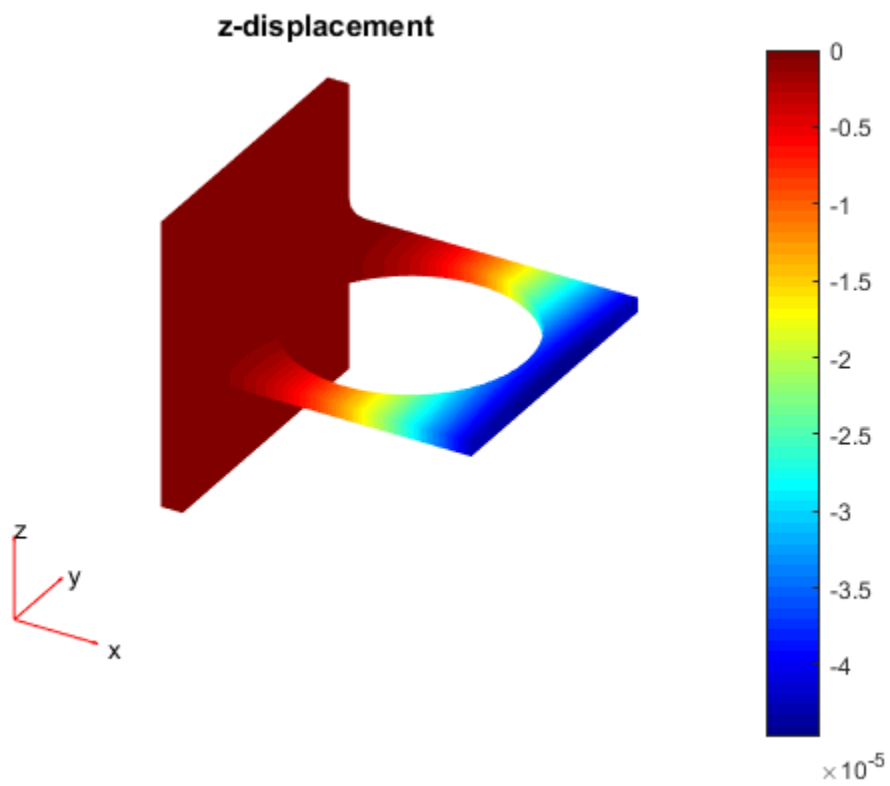
# Vibration of a Square Plate

This example shows how to calculate the vibration modes and frequencies of a 3-D simply supported, square, elastic plate. The dimensions and material properties of the plate are taken from a standard finite element benchmark problem published by NAFEMS, FV52 ([1]).

### Create a PDE Model

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, PDE coefficients, 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);
```

### Construct the Geometry

Import an STL file of a simple plate model using the `importGeometry` function. This function reconstructs the faces, edges and vertices of the model. It can merge some faces and edges, so the numbers can differ from those of the parent CAD model.

```
importGeometry(model,'Plate10x10x1.stl');
```

Plot the geometry and turn on face labels. You will need the face labels to define the boundary conditions.

```
figure
hc = pdegplot(model,'FaceLabels','on');
hc(1).FaceAlpha = 0.5;
title('Plate with Face Labels')
```

**Plate with Face Labels**

F4

F1

F5

F6

F3

F2

1
0
z
-1
-2 · y
-3

-2

0

2

4

6

8

10

10

8

6

4

2

0

-2

### Examine the Elasticity Equations

As explained in 3-D Linear Elasticity Equations in Toolbox Form, you can express the elasticity equations for the deflection $u$ of a linear isotropic solid as a three-component system

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

where $f = [0; 0; 0]$ when there are no body forces, and $c$ is the symmetric matrix

$$c = \begin{bmatrix} c_1 & 0 & 0 & 0 & c_{12} & 0 & 0 & 0 & c_{12} \\ \bullet & G & 0 & G & 0 & 0 & 0 & 0 & 0 \\ \bullet & \bullet & G & 0 & 0 & 0 & G & 0 & 0 \\ \bullet & \bullet & \bullet & G & 0 & 0 & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & c_1 & 0 & 0 & 0 & c_{12} \\ \bullet & \bullet & \bullet & \bullet & \bullet & G & 0 & G & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & G & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & G & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & c_1 \end{bmatrix} .$$

The $\bullet$ symbol means the entry is symmetric. Here $E$ is the elastic modulus, $\nu$ is Poisson's ratio, and

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

- $c_1 = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}$

- $c_{12} = c_1 \frac{\nu}{1-\nu}$

You can create the c coefficient for a linear isotropic solid using the `elasticityC3D` function, which is included in your software (see 3-D Linear Elasticity Equations in Toolbox Form).

### Define the Coefficients in Toolbox Syntax

Define the elastic modulus of steel, Poisson's ratio, and the material density.

```
E = 200e9; % Modulus of elasticity in Pascals
nu = .3; % Poisson's ratio
m = 8000; % Material density in kg/m^3
```

Incorporate these coefficients in toolbox syntax.

```
c = elasticityC3D(E,nu);
a = 0;
```

### Specify PDE Coefficients

Include the PDE COefficients in `model`.

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

**Define the Boundary Conditions**

In this example, the only boundary conditions are that the four edge faces have zero displacement in the $z$ direction. The edge faces have labels 1 through 4. All other boundary conditions, by default, are free Neumann boundaries.

Set the boundary conditions.

```
applyBoundaryCondition(model,'mixed','Face',1:4,'u',0,'EquationIndex',3);
```

**Create a Mesh**

Create a mesh that uses 10-node tetrahedral elements with quadratic interpolation functions. This element type is significantly more accurate than the linear interpolation (four-node) elements, particularly in elasticity analyses that involve bending.

```
hmax = 1.2; % Maximum element length for a moderately fine mesh
generateMesh(model,'Hmax',hmax,'GeometricOrder','quadratic');
figure
pdeplot3D(model);
title('Mesh with Quadratic Tetrahedral Elements');
```

## Mesh with Quadratic Tetrahedral Elements
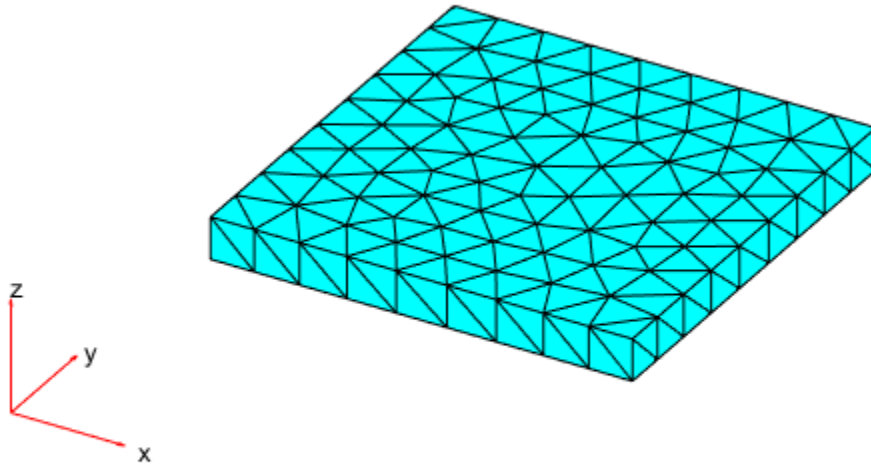


### Calculate the Solution

For comparison with published values, load the reference frequencies in Hz from [1].

```
refFreqHz = [0 0 0 45.897 109.44 109.44 167.89 193.59 206.19 206.19];
```

Use `solvepdeeig` to calculate all eigensolutions in a specified range. Define the upper value of the range slightly as slightly larger than the highest frequency from [1].

```
maxLam = (1.1*refFreqHz(end)*2*pi)^2;
r = [-.1 maxLam];
result = solvepdeeig(model,r);
eVec = result.Eigenvectors;
```

```matlab
eVal = result.Eigenvalues;
numEig = size(eVal,1);
```

```
                Basis= 10,  Time=   1.20,  New conv eig=  0
                Basis= 12,  Time=   1.20,  New conv eig=  1
                Basis= 14,  Time=   1.22,  New conv eig=  1
                Basis= 16,  Time=   1.31,  New conv eig=  1
                Basis= 18,  Time=   1.47,  New conv eig=  1
                Basis= 20,  Time=   1.62,  New conv eig=  1
                Basis= 22,  Time=   1.73,  New conv eig=  1
                Basis= 24,  Time=   1.75,  New conv eig=  4
                Basis= 26,  Time=   1.90,  New conv eig=  6
                Basis= 28,  Time=   1.98,  New conv eig=  4
                Basis= 30,  Time=   2.04,  New conv eig= 10
                Basis= 32,  Time=   2.25,  New conv eig= 10
                Basis= 34,  Time=   2.39,  New conv eig= 10
                Basis= 36,  Time=   2.54,  New conv eig= 10
                Basis= 38,  Time=   2.59,  New conv eig= 14
End of sweep: Basis= 38,  Time=   2.59,  New conv eig= 14
                Basis= 24,  Time=   3.17,  New conv eig=  0
                Basis= 26,  Time=   3.28,  New conv eig=  0
                Basis= 28,  Time=   3.37,  New conv eig=  0
                Basis= 30,  Time=   3.37,  New conv eig=  1
End of sweep: Basis= 30,  Time=   3.37,  New conv eig=  1
                Basis= 25,  Time=   3.78,  New conv eig=  0
End of sweep: Basis= 25,  Time=   3.78,  New conv eig=  0
```

Calculate the frequencies in Hz from the eigenvalues.

```matlab
freqHz = sqrt(eVal(1:numEig))/(2*pi);
```

### Compare the Solution to Reference

Compare the reference and computed frequencies for the lowest 10 modes. The lowest three mode shapes correspond to rigid-body motion of the plate. Their frequencies are close to zero.

```matlab
fprintf('Frequency(Reference 1),Hz  Frequency(solvepdeeig),Hz\n');
numToPrint = min(length(freqHz),length(refFreqHz));
for i=1:numToPrint
  fprintf('%10.5g                       %10.5g\n', ...
    refFreqHz(i),freqHz(i));
end
```

```
Frequency(Reference 1),Hz  Frequency(solvepdeeig),Hz
          0                          0
          0                          0
          0                   1.9122e-05
     45.897                     44.885
     109.44                     109.54
     109.44                     109.62
     167.89                     168.19
     193.59                     193.74
     206.19                      207.4
     206.19                     207.48
```

You see good agreement between the computed and published frequencies.

### Plot the Mode Shapes

For plotting and analysis, create a PDEResults object from the solution. Plot the third
(⌁) component of the solution for the seven lowest nonzero-frequency modes.

```
h = figure;
h.Position = [100,100,900,600];
for i = 4:numToPrint
    subplot(4,2,i-3);
    pdeplot3D(model,'ColorMapData',eVec(:,3,i));
    axis equal
    title(sprintf(['Mode=%d, z-displacement\n', ...
    'Frequency(Hz): Ref=%g FEM=%g'], ...
    i,refFreqHz(i),freqHz(i)));
end
```

Mode=4, z-displacement
Frequency(Hz): Ref=45.897 FEM=44.8855

Mode=5, z-displacement
Frequency(Hz): Ref=109.44 FEM=109.541

Mode=6, z-displacement
Frequency(Hz): Ref=109.44 FEM=109.617

Mode=7, z-displacement
Frequency(Hz): Ref=167.89 FEM=168.191

Mode=8, z-displacement
Frequency(Hz): Ref=193.59 FEM=193.743
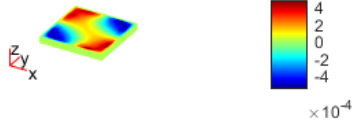
Mode=9, z-displacement
Frequency(Hz): Ref=206.19 FEM=207.398
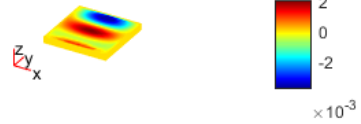
Mode=10, z-displacement
Frequency(Hz): Ref=206.19 FEM=207.479

### Reference

[1] National Agency for Finite Element Methods and Standards. *The Standard NAFEMS Benchmarks*. United Kingdom: NAFEMS, October 1990.

# Electrostatic Potential in an Air-Filled Frame

Find the electrostatic potential in an air-filled annular quadrilateral frame using the PDE app. For this example, use the following parameters:

- Inner square side is 0.2 m
- Outer square side is 0.5 m
- Electrostatic potential at the inner boundary is $1000V$
- Electrostatic potential at the outer boundary is $0V$

The PDE governing this problem is the Poisson equation
$-\nabla \cdot (\varepsilon \nabla V) = \rho$.

The PDE app uses the relative permittivity $\varepsilon_r = \varepsilon/\varepsilon_0$, where $\varepsilon_0$ is the absolute dielectric permittivity of a vacuum ($8.854 \cdot 10^{-12}$ farad/meter). The relative permittivity for the air is 1.00059. Note that the coefficient of permittivity does not affect the result in this example as long as the coefficient is constant.

Assuming that there is no charge in the domain, you can simplify the Poisson equation to the Laplace equation,
$\Delta V = 0$.

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

To solve this problem in the PDE app, follow these steps:

**1** Draw the following two squares.

```
pderect([-0.1 0.1 -0.1 0.1])
pderect([-0.25 0.25 -0.25 0.25])
```

**2** Set both *x*- and *y*-axis limits to `[-0.3 0.3]`. To do this, select **Options** > **Axes Limits** and set the corresponding ranges. Then select **Options** > **Axes Equal**.

**3** Model the frame by entering `SQ2-SQ1` in the **Set formula** field.

**4** Set the application mode to **Electrostatics**.

**5** Specify the boundary conditions. To do this, switch to the boundary mode by selecting **Boundary** > **Boundary Mode**. Use **Shift**+click to select several boundaries. Then select **Boundary** > **Specify Boundary Conditions**.

- For the inner boundaries, use the Dirichlet boundary condition with `h = 1` and `r = 1000`.

- For the outer boundaries, use the Dirichlet boundary condition with `h = 1` and `r = 0`.

**6** Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Specify `epsilon = 1` and `rho = 0`.

**7** Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.

**8** Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.

**9** Plot the equipotential lines using a contour plot. To do this, select **Plot** > **Parameters** and choose the contour plot in the resulting dialog box.

**10** Improve the accuracy of the solution by refining the mesh close to the reentrant corners where the gradients are steep. To do this, select **Solve** > **Parameters**. Select **Adaptive mode**, use the **Worst triangles** selection method, and set the maximum number of triangles to 500. Select **Mesh** > **Refine Mesh**.

**11** Solve the PDE using the refined mesh. To display equipotential lines at every 100th volt, select **Plot** > **Parameters** and enter `0:100:1000` in the **Contour plot levels** field.
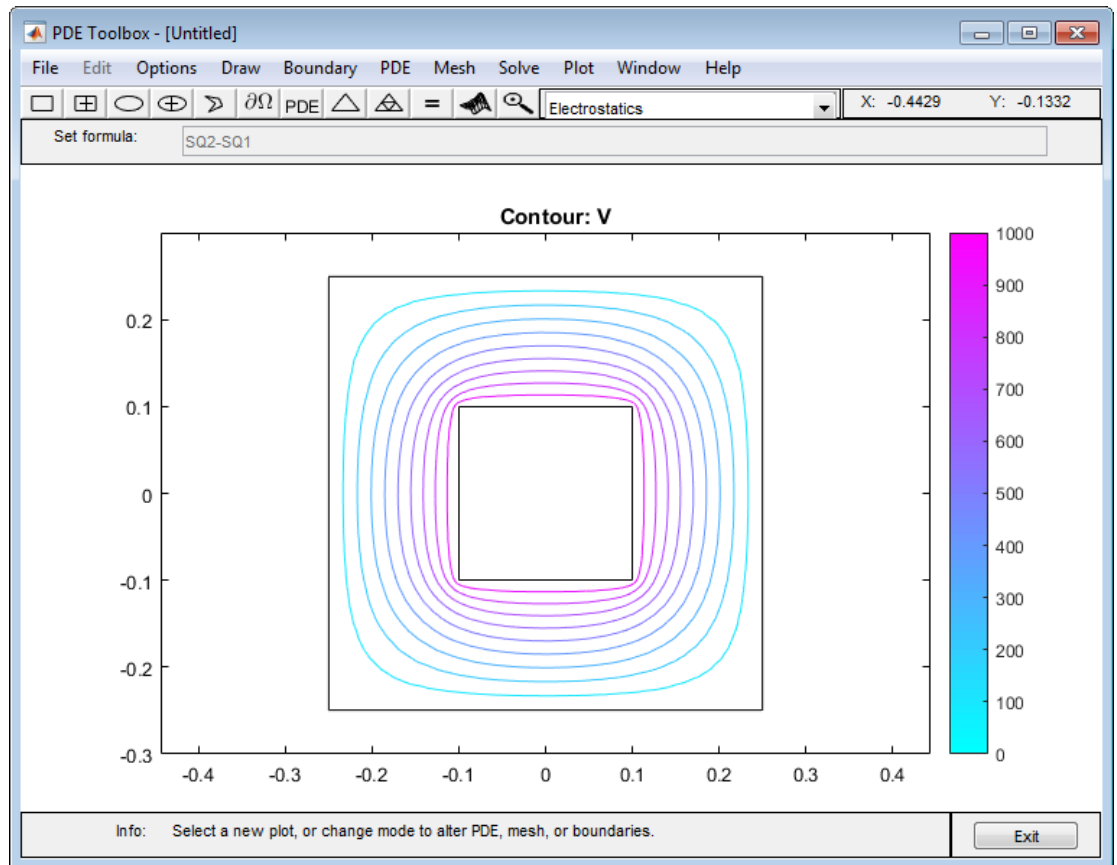
# 3-D Linear Elasticity Equations in Toolbox Form

**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.

$\sigma = \text{stress}$

$f = \text{body force}$

$\varepsilon = \text{strain}$

$u = \text{displacement}$

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 $\sigma$ and $\varepsilon$ 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 $\nabla u$ 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 $\varepsilon$ to $\nabla u$. 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,

$$\sigma = \frac{E}{(1+v)(1-2v)} \begin{bmatrix} 1-v & 0 & 0 & 0 & v & 0 & 0 & 0 & v \\ \bullet & 1-2v & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bullet & \bullet & 1-2v & 0 & 0 & 0 & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & 1-2v & 0 & 0 & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & 1-v & 0 & 0 & 0 & v \\ \bullet & \bullet & \bullet & \bullet & \bullet & 1-2v & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-2v & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-2v & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-v \end{bmatrix} A\nabla u$$

$$= \frac{E}{(1+v)(1-2v)} \begin{bmatrix} 1-v & 0 & 0 & 0 & v & 0 & 0 & 0 & v \\ 0 & 1/2-v & 0 & 1/2-v & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2-v & 0 & 0 & 0 & 1/2-v & 0 & 0 \\ 0 & 1/2-v & 0 & 1/2-v & 0 & 0 & 0 & 0 & 0 \\ v & 0 & 0 & 0 & 1-v & 0 & 0 & 0 & v \\ 0 & 0 & 0 & 0 & 0 & 1/2-v & 0 & 1/2-v & 0 \\ 0 & 0 & 1/2-v & 0 & 0 & 0 & 1/2-v & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/2-v & 0 & 1/2-v & 0 \\ v & 0 & 0 & 0 & v & 0 & 0 & 0 & 1-v \end{bmatrix} \nabla u$$

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 2-127. This is the form of the `c` argument in `elasticityC3D(E,nu)`.

# Magnetic Field in a Two-Pole Electric Motor

Find the static magnetic field induced by the stator windings in a two-pole electric motor. The example uses the PDE app. Assuming that the motor is long and end effects are negligible, you can use a 2-D model. The geometry consists of three regions:

- Two ferromagnetic pieces: the stator and the rotor (transformer steel)
- The air gap between the stator and the rotor
- The armature copper coil carrying the DC current
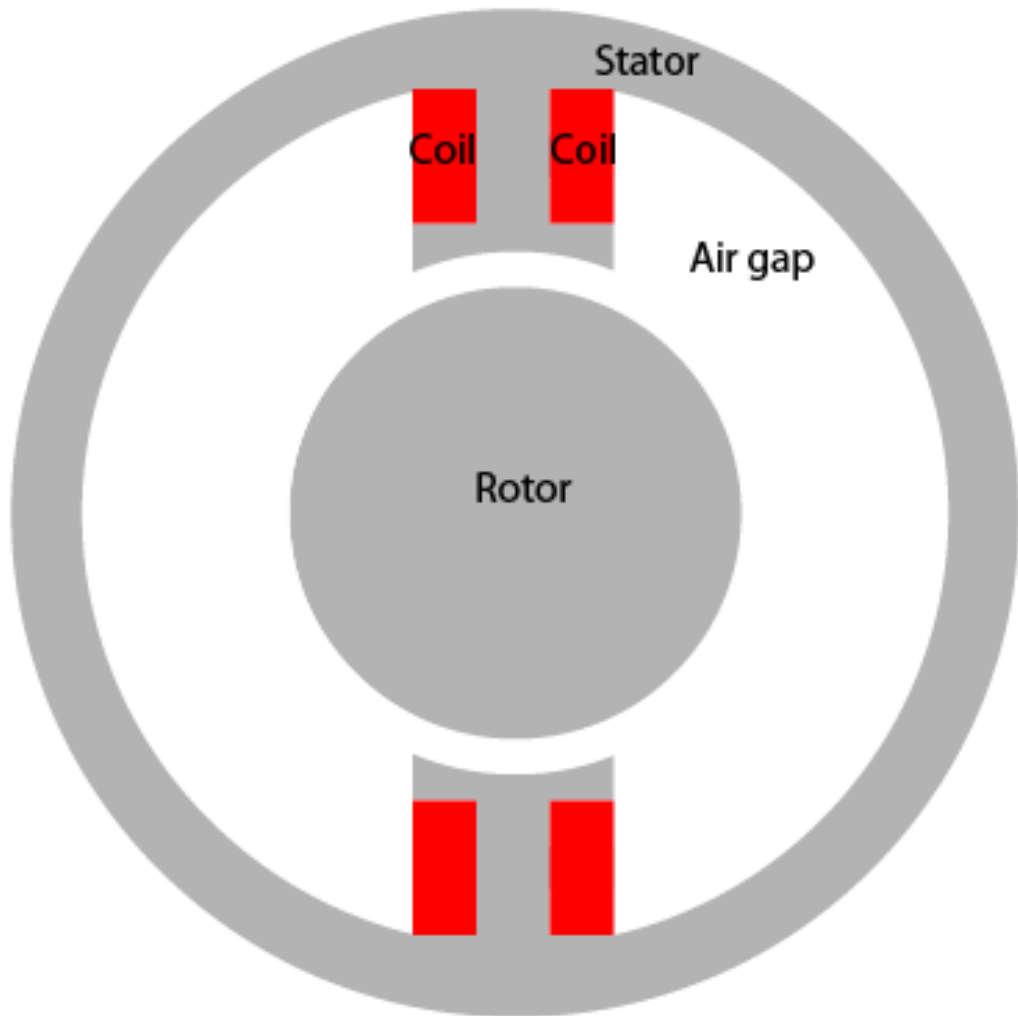
Magnetic permeability of the air and copper is close to the magnetic permeability of a vacuum, $\mu_0 = 4\pi*10^{-7}$ H/m. In this example, use the magnetic permeability $\mu = \mu_0$ for both the air gap and copper coil. For the stator and the rotor, $\mu$ is

$$\mu = \mu_0 \left( \frac{\mu_{\max}}{1 + c\|\nabla A\|^2} + \mu_{\min} \right)$$

where $\mu_{\max} = 5000$, $\mu_{\min} = 200$, and $c = 0.05$. The current density $J$ is 0 everywhere except in the coil, where it is 10 A/m$^2$.

The geometry of the problem makes the magnetic vector potential $A$ symmetric with respect to $y$ and antisymmetric with respect to $x$. Therefore, 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. Because the field outside the motor is negligible, you can use the Dirichlet boundary condition $A = 0$ on the exterior boundary.

To solve this problem in the PDE app, follow these steps:

**1** Set the $x$-axis limits to `[-1.5 1.5]` and the $y$-axis limits to `[-1 1]`. To do this, select **Options** > **Axes Limits** and set the corresponding ranges.

**2** Set the application mode to **Magnetostatics**.

**3** Create the geometry. The geometry of this electric motor is complex. The model is a union of five circles and two rectangles. The reduction to the first quadrant is achieved by intersection with a square. To draw the geometry, enter the following commands in the MATLAB Command Window:

```
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')
```
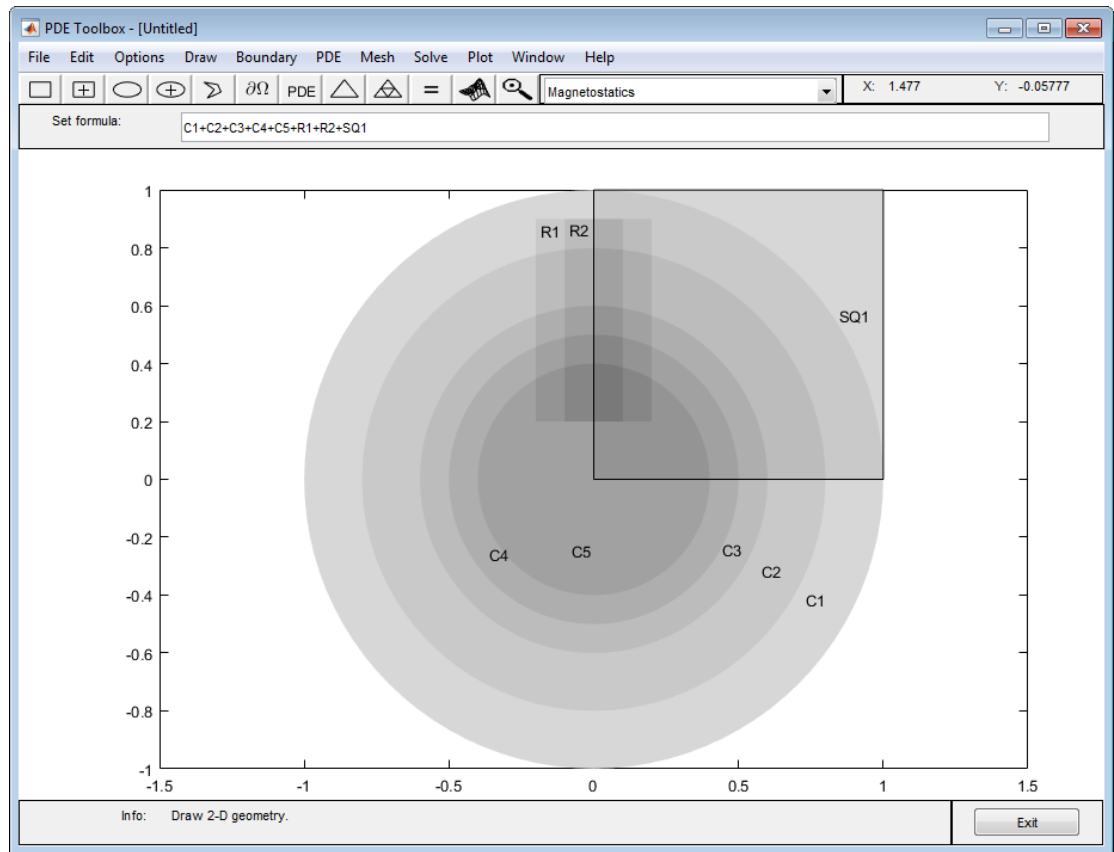
**4** Reduce the model to the first quadrant. To do this, enter
(C1+C2+C3+C4+C5+R1+R2)*SQ1 in the **Set formula** field.

**5** Remove unnecessary subdomain borders. To do this, switch to the boundary mode
by selecting **Boundary > Boundary Mode**. Using **Shift**+click, select borders, and
then select **Boundary > Remove Subdomain Border** until the geometry consists
of four subdomains: the rotor (subdomain 1), the stator (subdomain 2), the air gap
(subdomain 3), and the coil (subdomain 4). The numbering of your subdomains
can differ. If you do not see the numbers, select **Boundary > Show Subdomain
Labels**.
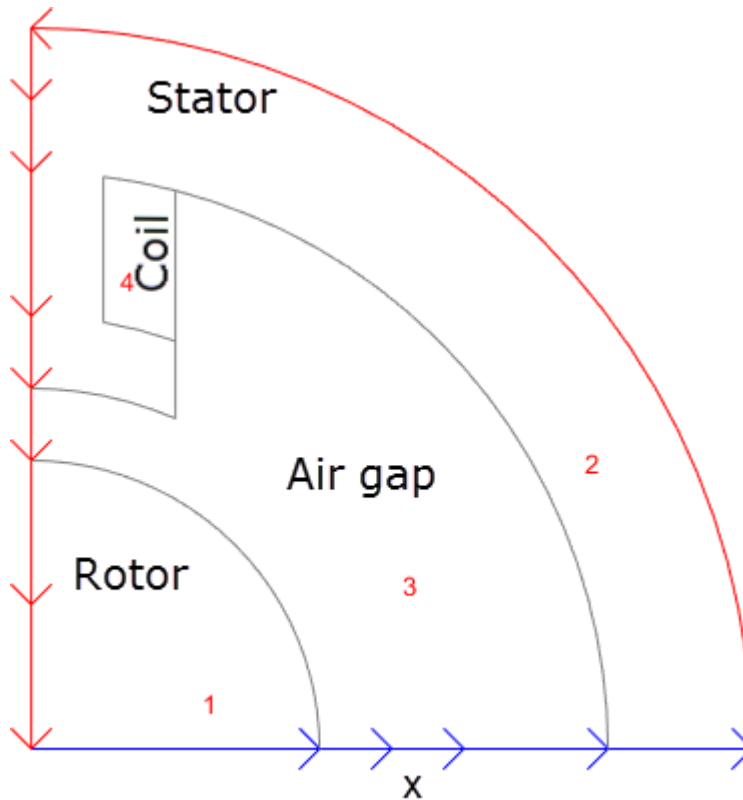
**6** Specify the boundary conditions. To do this, select the boundaries along the *x*-axis. Select **Boundary** > **Specify Boundary Conditions**. In the resulting dialog box, specify a Neumann boundary condition with $g = 0$ and $q = 0$.

All other boundaries have a Dirichlet boundary condition with $h = 1$ and $r = 0$, which is the default boundary condition in the PDE app.

**7** Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Double-click each subdomain and specify the following coefficients:

- Coil: $\mu$ = `4*pi*10^(-7)` H/m, `J = 10` A/m$^2$.

- Stator and rotor: $\mu$ = `4*pi*10^(-7)*(5000./` `(1+0.05*(ux.^2+uy.^2))+200)` H/m, where `ux.^2+uy.^2` equals to $|\nabla A|^2$, `J = 0` (no current).

- Air gap: $\mu$ = `4*pi*10^(-7)` H/m, `J = 0`.

**8** Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.

**9** Choose the nonlinear solver. To do this, select **Solve** > **Parameters** and check **Use nonlinear solver**. Here, you also can adjust the tolerance parameter and choose to use the adaptive solver together with the nonlinear solver.

**10** Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.

**11** Plot the magnetic flux density *B* using arrows and the equipotential lines of the magnetostatic potential *A* using a contour plot. To do this, select **Plot** > **Parameters** and choose the contour and arrows plots in the resulting dialog box. Using **Options** > **Axes Limits**, adjust the axes limits as needed. For example, use the **Auto** check box.

The plot shows that the magnetic flux is parallel to the equipotential lines of the magnetostatic potential.
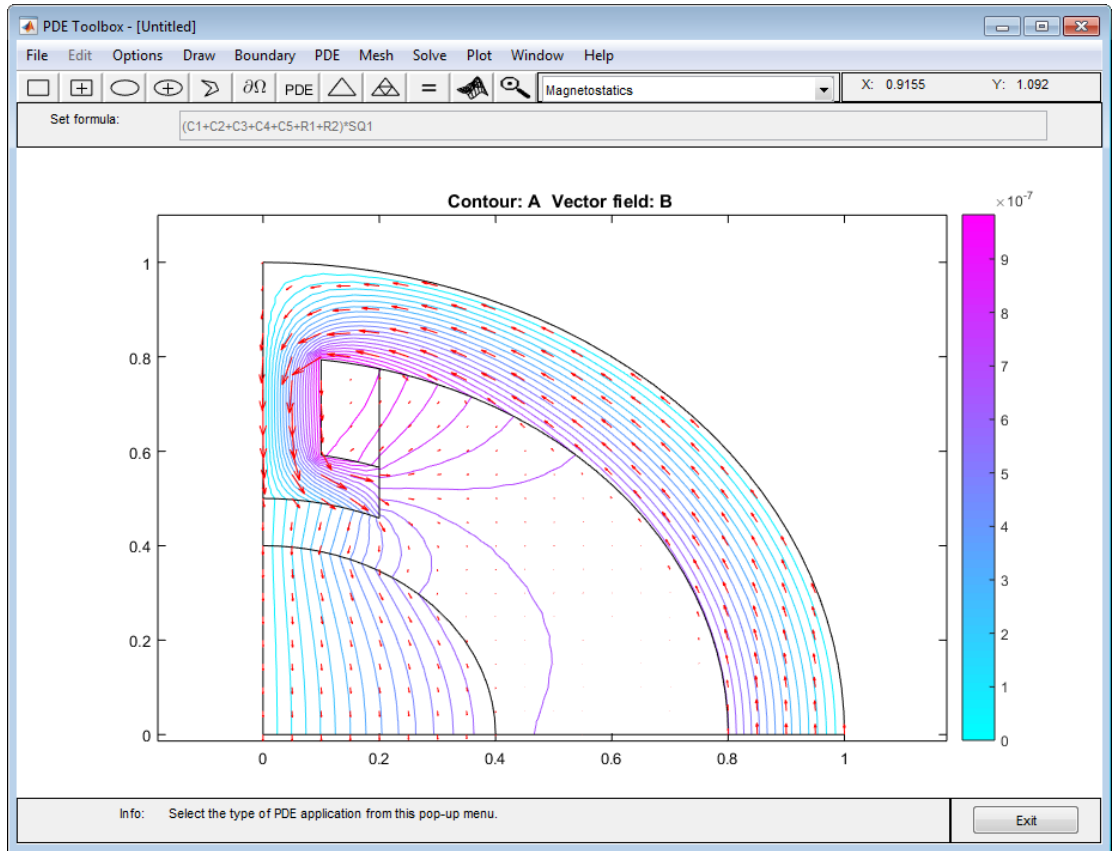
# Helmholtz's Equation on a Unit Disk with a Square Hole

This example shows how to solve a Helmholtz equation using the `solvepde` function in Partial Differential Toolbox™.

The Helmholtz equation, an elliptic equation that is the time-independent form of the wave equation, is

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

Solving this equation allows us to compute the waves reflected by a square object illuminated by incident waves that are coming from the right.

### Problem Definition

The following variables define our problem:

- `g`: A geometry specification function. For more information, see the code for `scatterg.m` and the documentation section Create Geometry Using a Geometry Function.
- `k`, `c`, `a`, `f`: The coefficients and inhomogeneous term.

```
g = @scatterg;
k = 60;
c = 1;
a = -k^2;
f = 0;
```

### Create PDE Model

Create a PDE Model with a single dependent variable.

```
numberOfPDE = 1;
model = createpde(numberOfPDE);
```

Convert the geometry and append it to the pde model.

```
geometryFromEdges(model,g);
```

### Specify PDE Coefficients

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

**Boundary Conditions**

Plot the geometry and display the edge labels for use in the boundary condition definition.

```
figure;
pdegplot(model,'EdgeLabels','on');
axis equal
title 'Geometry With Edge Labels Displayed';
ylim([0,1])
```
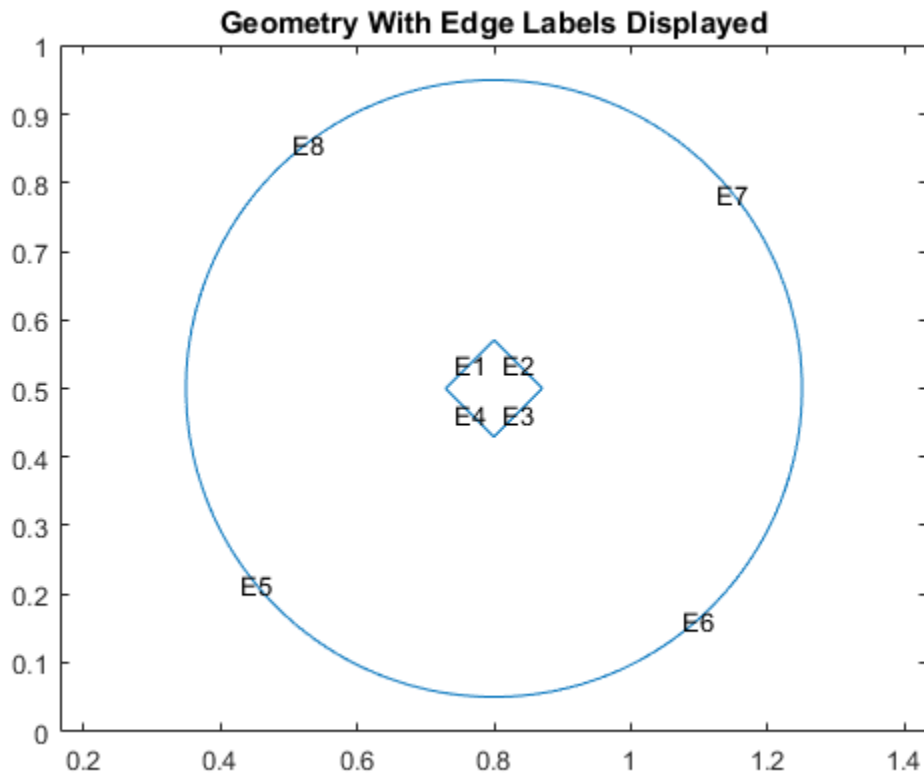


Apply the boundary conditions.

```
bOuter = applyBoundaryCondition(model,'neumann','Edge',(5:8),'g',0,'q',-60i);
innerBCFunc = @(loc,state)-exp(-1i*k*loc.x);
```

```
bInner = applyBoundaryCondition(model,'dirichlet','Edge',(1:4),'u',innerBCFunc);
```

### Create Mesh

```
generateMesh(model,'Hmax',0.02);
figure
pdemesh(model);
axis equal
```
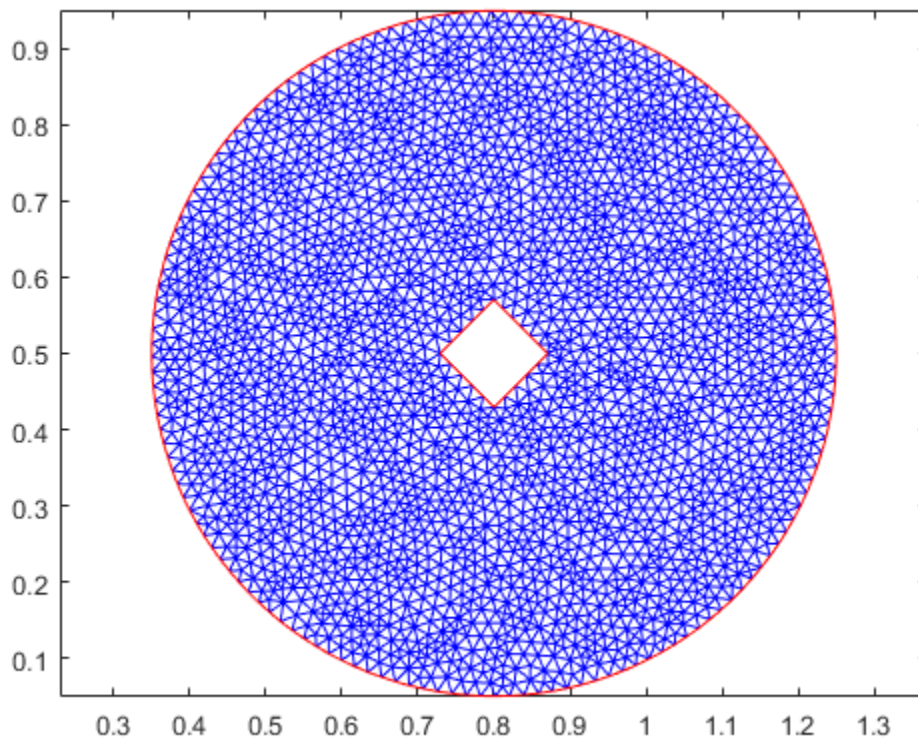


### Solve for Complex Amplitude

The real part of the vector u stores an approximation to a real-valued solution of the Helmholtz equation.

```
result = solvepde(model);
```

```
u = result.NodalSolution;
```

**Plot FEM Solution**

```
figure
pdeplot(model,'XYData',real(u),'ZData',real(u),'Mesh','off');
colormap(cool)
```
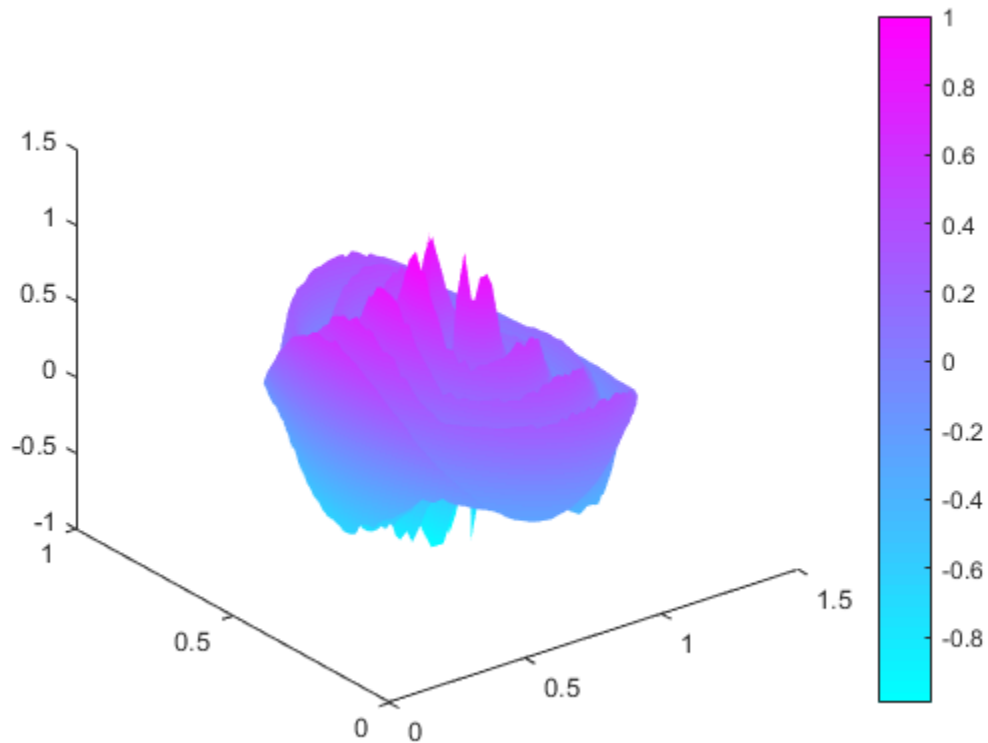


**Animate Solution to Wave Equation**

Using the solution to the Helmholtz equation, construct an animation showing the corresponding solution to the time-dependent wave equation.

```
figure
m = 10;
```

```matlab
h = newplot;
hf = h.Parent;
axis tight
ax = gca;
ax.DataAspectRatio = [1 1 1];
axis off
maxu = max(abs(u));
for j = 1:m
    uu = real(exp(-j*2*pi/m*sqrt(-1))*u);
    pdeplot(model,'XYData',uu,'ColorBar','off','Mesh','off');
    caxis([-maxu maxu]);
    axis tight
    ax = gca;
    ax.DataAspectRatio = [1 1 1];
    axis off
    M(j) = getframe(hf);
end
```

To play the movie 10 times, use the `movie(hf,M,10)` command.

# 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 $\mu$, 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 $\varepsilon$, magnetic permeability $\mu$, 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* $\varepsilon_c$ can be defined as $\varepsilon_c = \varepsilon - j\sigma/\omega$. The conditions at material interfaces with abrupt changes of $\varepsilon$ and $\mu$ 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 $\omega$*, the magnetic permeability $\mu$, the conductivity $\sigma$, and the coefficient of dielectricity $\varepsilon$.

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 $\mathbf{J} = o\mathbf{E}$ 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, $E_c = 1/o$. 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 $\omega = 2\pi \cdot 50$.
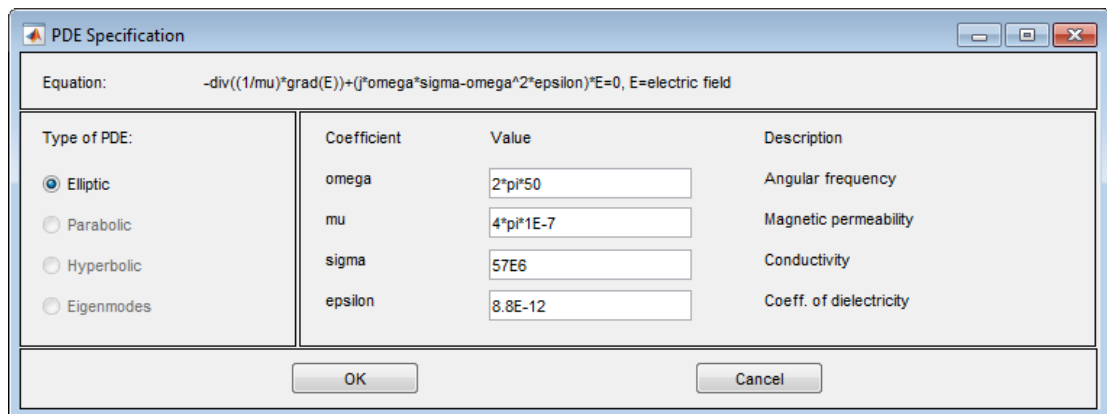


Initialize the mesh and solve the equation. Due to the skin effect, the current density at the surface of the conductor is much higher than in the conductor's interior. This is clearly visualized by plotting the current density $J$ as a 3-D plot. To improve the accuracy of the solution close to the surface,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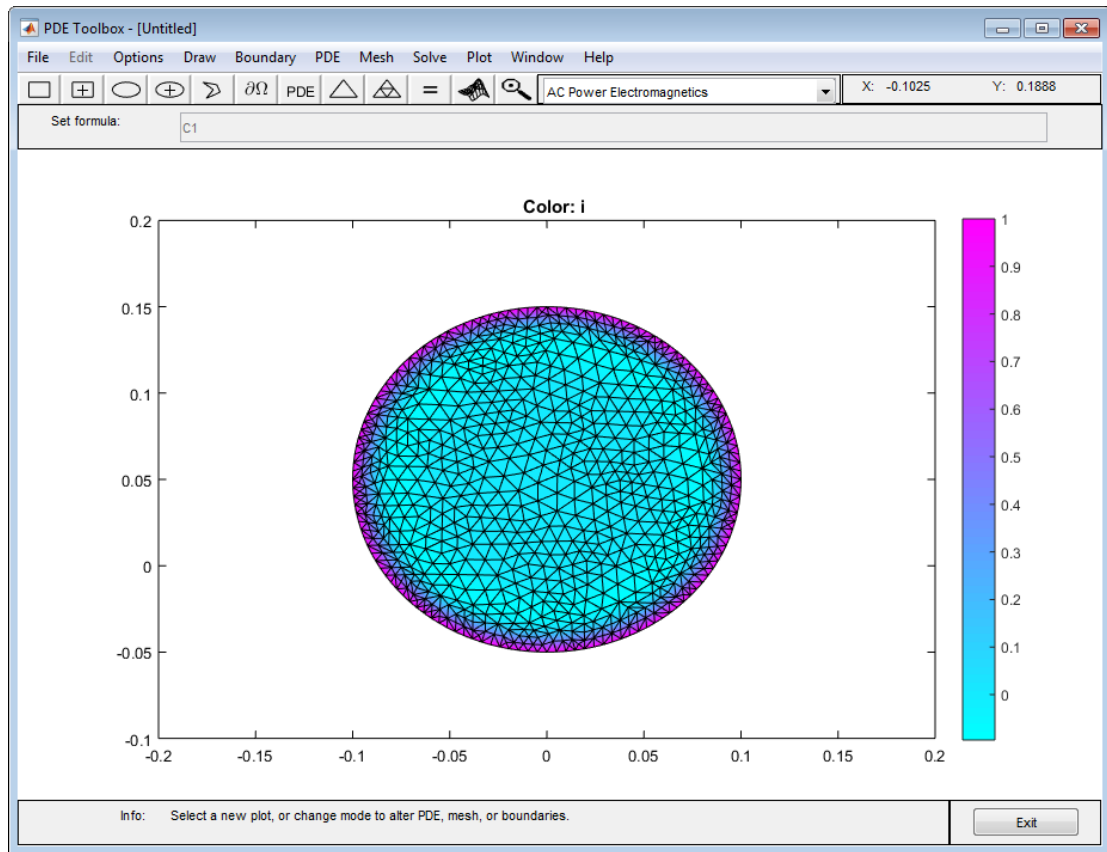
**The Adaptively Refined Mesh**

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).
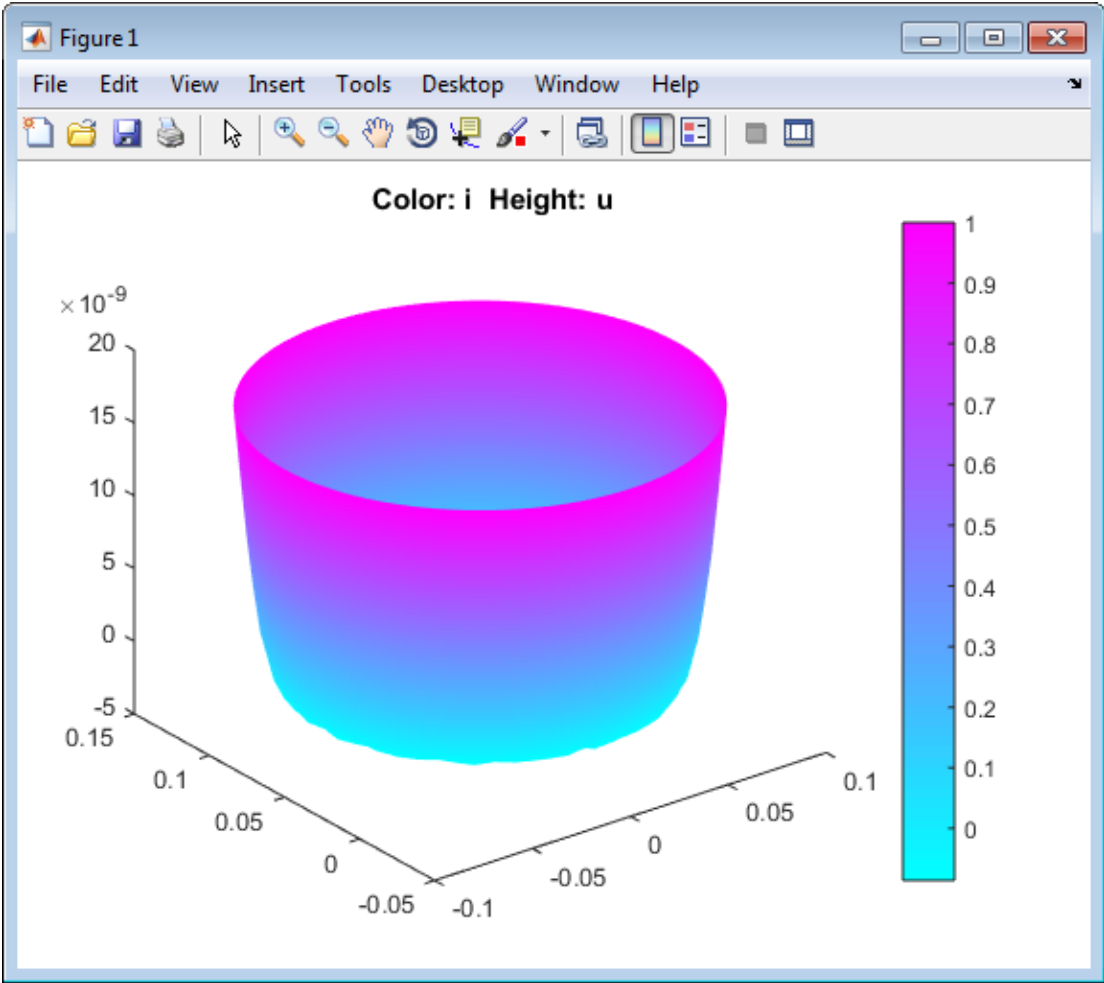
**The Current Density in an AC Wire**

# 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 $\sigma$ 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 $\sigma$ 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**.
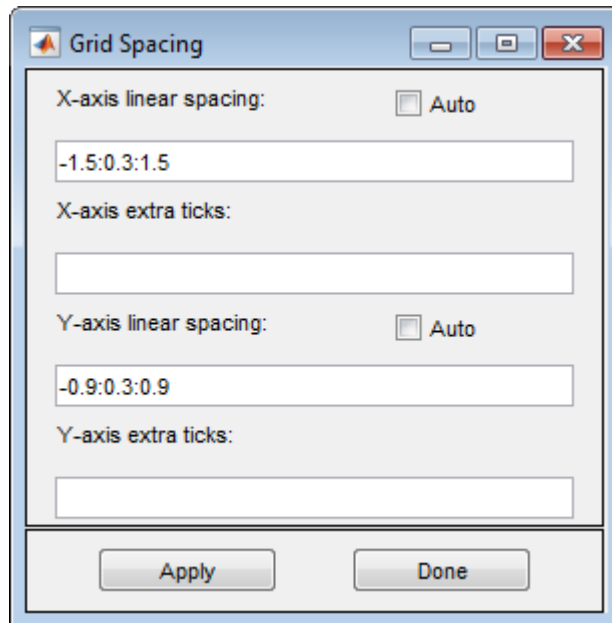


**4**  Click **Options** > **Snap**

**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 $\partial\Omega$ .

**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**.
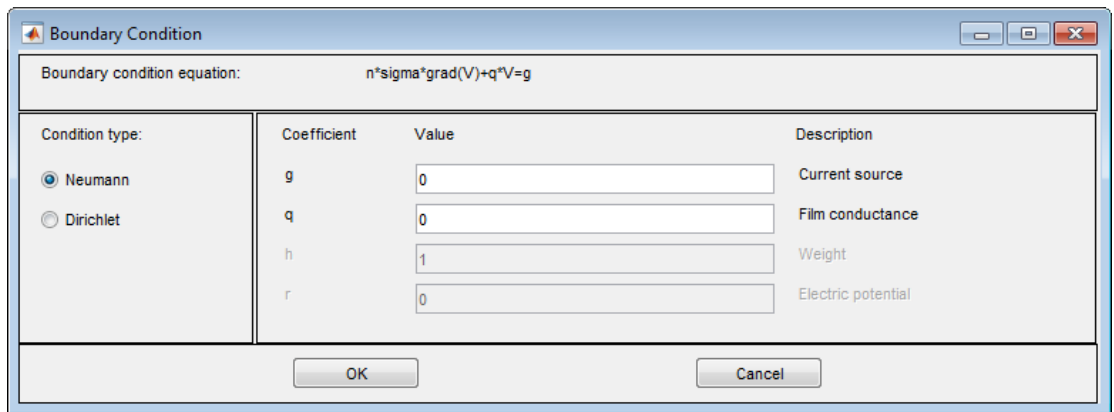


**11**  Hold down **Shift** and click to select the left circular conductor boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

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

- **Condition type** = **Dirichlet**
- **h** = 1
- **r** = 1
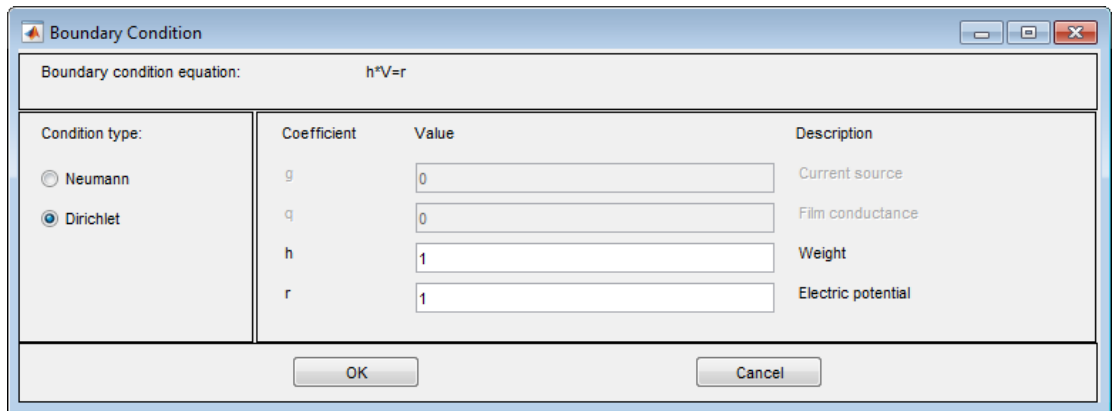
13 Hold down **Shift** and click to select the right circular conductor boundaries. Double-click the last boundary to open the Boundary Condition dialog box.

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

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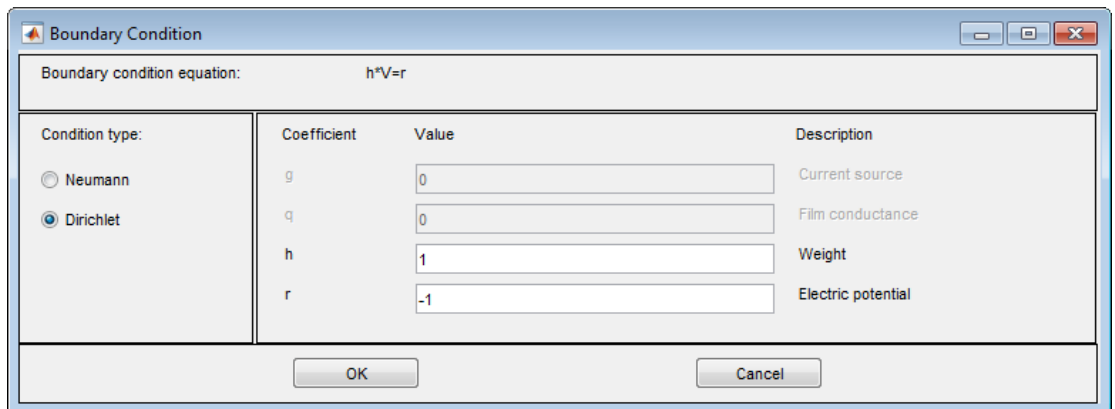


15 Open the PDE Specification dialog box by clicking **PDE** > **PDE Specification**.

16 Set the current source, **q**, parameter to 0.
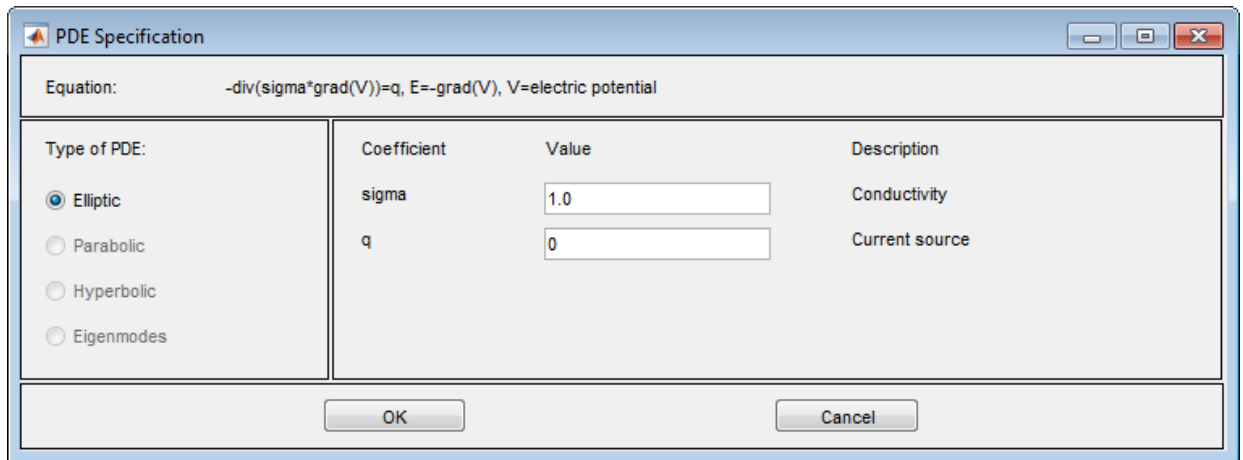
17  Initialize the mesh by clicking **Mesh** > **Initialize Mesh**.

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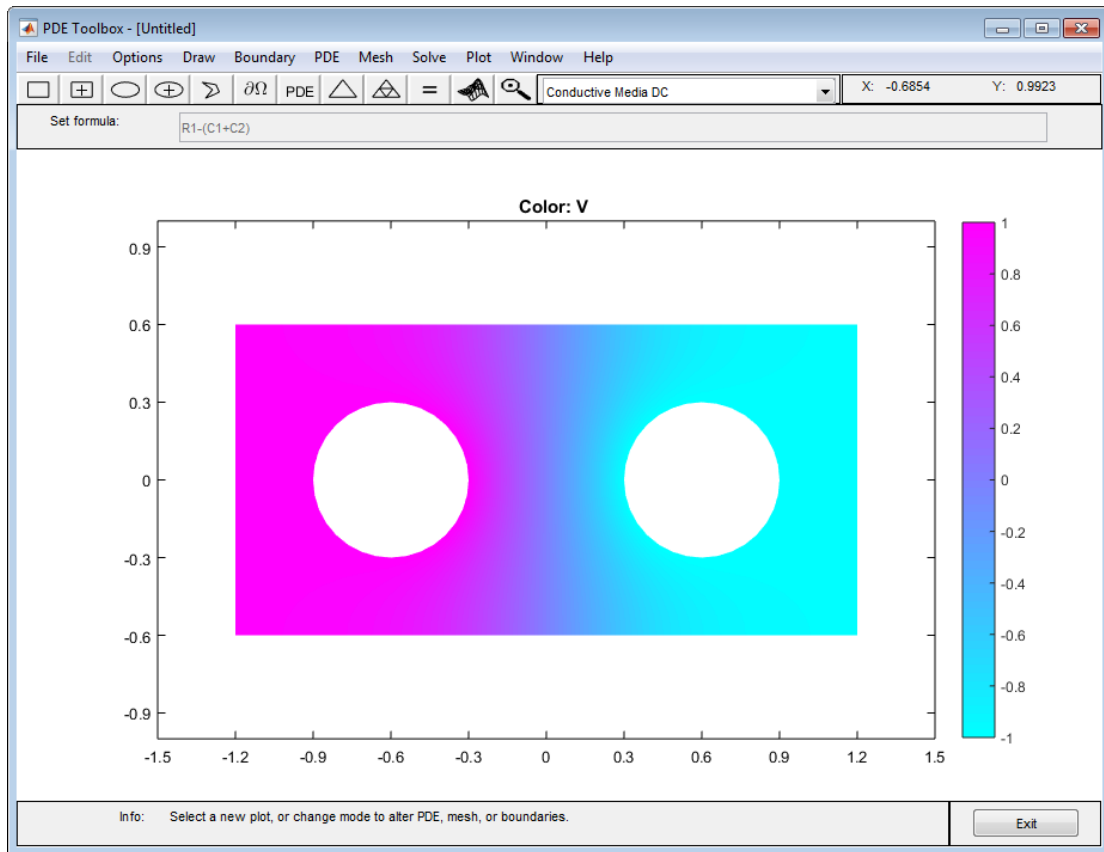
**21** Visualize the current density **J** by clicking **Plot** > **Parameters**, selecting **Contour** and **Arrows** check box, and clicking **Plot**.

The current flows, as expected, from the conductor with a positive potential to the conductor with a negative potential.
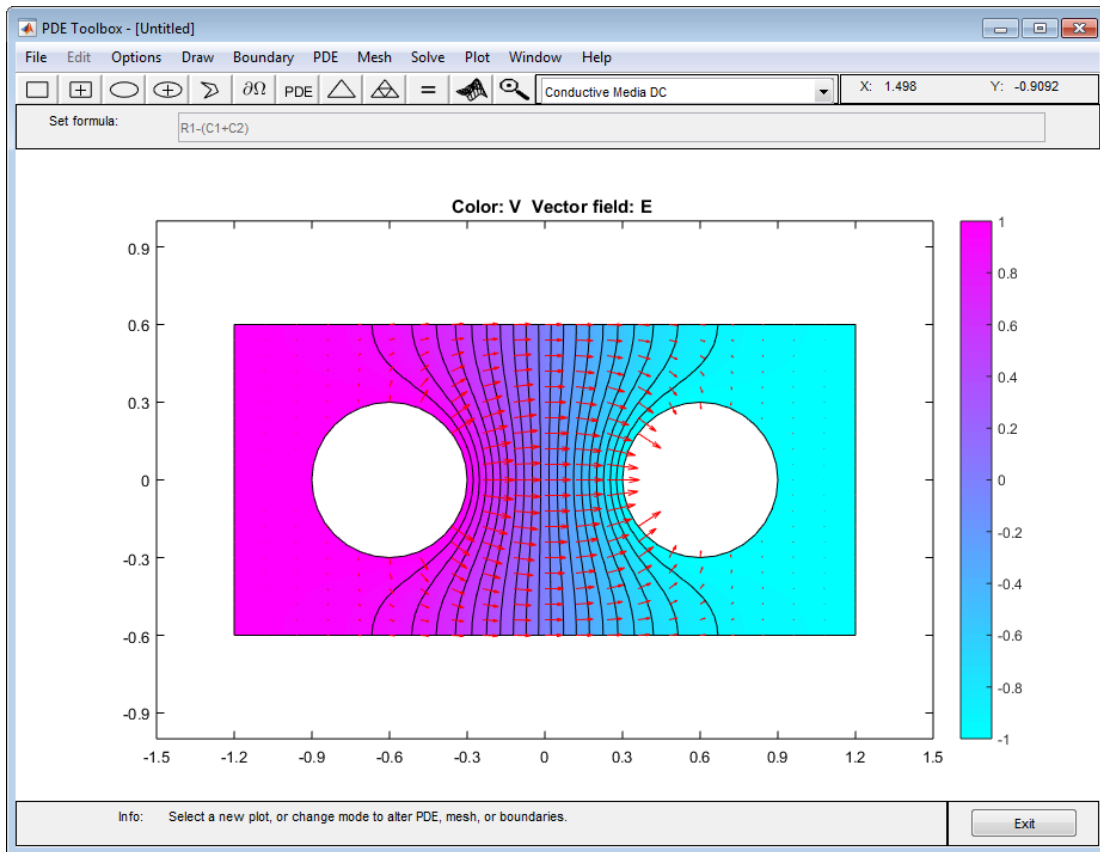
**The Current Density Between Two Metallic Conductors**

# Heat Transfer Between Two Squares Made of Different Materials: PDE App

Solve the following heat transfer problem with different material parameters. This example uses the PDE app. For the command-line solutions see "Heat Transfer Between Two Squares Made of Different Materials" on page 6-139.

The 2-D geometry for this problem is a square with an embedded diamond (a square with 45 degrees rotation). PDE governing this problem is a parabolic heat equation:

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

where $\rho$ is the density, $C$ is the heat capacity, $k$ is the coefficient of heat conduction, $Q$ is the heat source, $h$ is convective heat transfer coefficient, and $T_{\text{ext}}$ is the external temperature.

To solve this problem in the PDE app, follow these steps:

**1**   Model the geometry: draw 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])
```

**2**   Set the *x*-axis limit to [-1.5 4.5] and *y*-axis limit to [-0.5 3.5]. To do this, select **Options** > **Axes Limits** and set the corresponding ranges.

**3**   Set the application mode to **Heat Transfer**.

**4**   The temperature is kept at 0 on all the outer boundaries, so you do not have to change the default Dirichlet boundary condition T = 0.

**5**   Specify the coefficients. To do this, select **PDE** > **PDE Mode**. Then click each region and select **PDE** > **PDE Specification** or click the **PDE** button on the toolbar. Since you are solving the parabolic heat equation, select the **Parabolic** type of PDE for both regions. For the square region, specify the following coefficients:

  - Density, pho = 2
  - Heat capacity, C = 0.1
  - Coefficient of heat conduction, k = 10

**3-103**

- Heat source, `Q = 0`
- Convective heat transfer coefficient, `h = 0`
- External temperature, `Text = 0`

For the diamond-shaped region, specify the following coefficients:

- Density, `pho = 1`
- Heat capacity, `C = 0.1`
- Coefficient of heat conduction, `k = 2`
- Heat source, `Q = 4`
- Convective heat transfer coefficient, `h = 0`
- External temperature, `Text = 0`

**6** Initialize the mesh by selecting **Mesh** > **Initialize Mesh**. For a more accurate solution, refine the mesh by selecting **Mesh** > **Refine Mesh**.

**7** Set the initial value and the solution time. To do this, select **Solve** > **Parameters**.

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, set time to `logspace(-2,-1,10)`. This gives 10 logarithmically spaced numbers between 0.01 and 0.1. Set the initial value of the temperature $u(t_0)$ to `0`.

**8** Solve the equation by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.

**9** Plot the solution. By default, the app plots the temperature distribution at the last time. The best way to visualize the dynamic behavior of the temperature is to animate the solution. To do this, select **Plot** > **Parameters** and select the **Animation** and **Height (3-D plot)** options to animate a 3-D plot. Also, you can select the **Plot in x-y grid** option to use a rectangular grid instead of the default triangular grid. Using a rectangular grid instead of a triangular grid speeds up the animation process significantly.

You can also plot isothermal lines using a contour plot and the heat flux vector field using arrows.

**a** Select **Plot** > **Parameters**.

**b** In the resulting dialog box, deselect the **Animation**, and **Height (3-D plot)**, and **Plot in x-y grid** options.

**c** Change the colormap to `hot` by using the corresponding drop-down menu in the same dialog box.

**d** To obtain the first plot, select the **Color** and **Contour** options.

**e** For the second plot, select the **Color** and **Arrows** and set their values to `temperature` and `heat flux`, respectively.
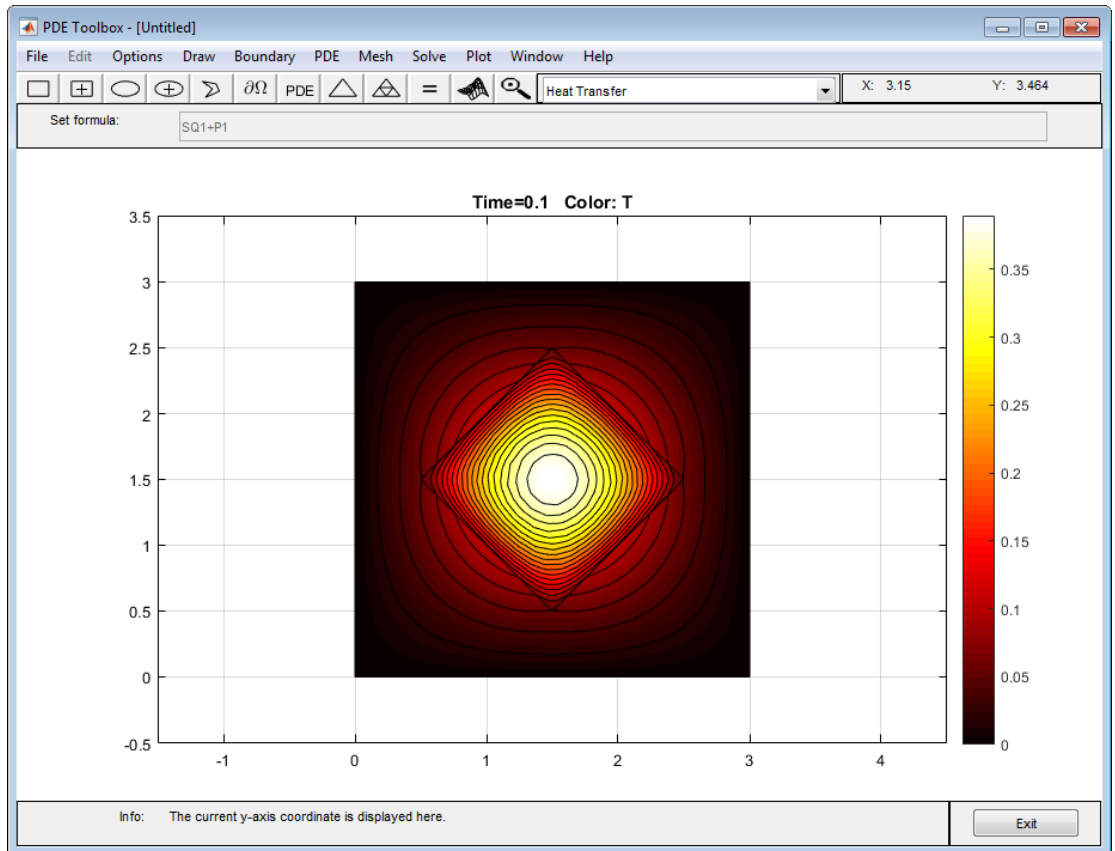


### Isothermal Lines
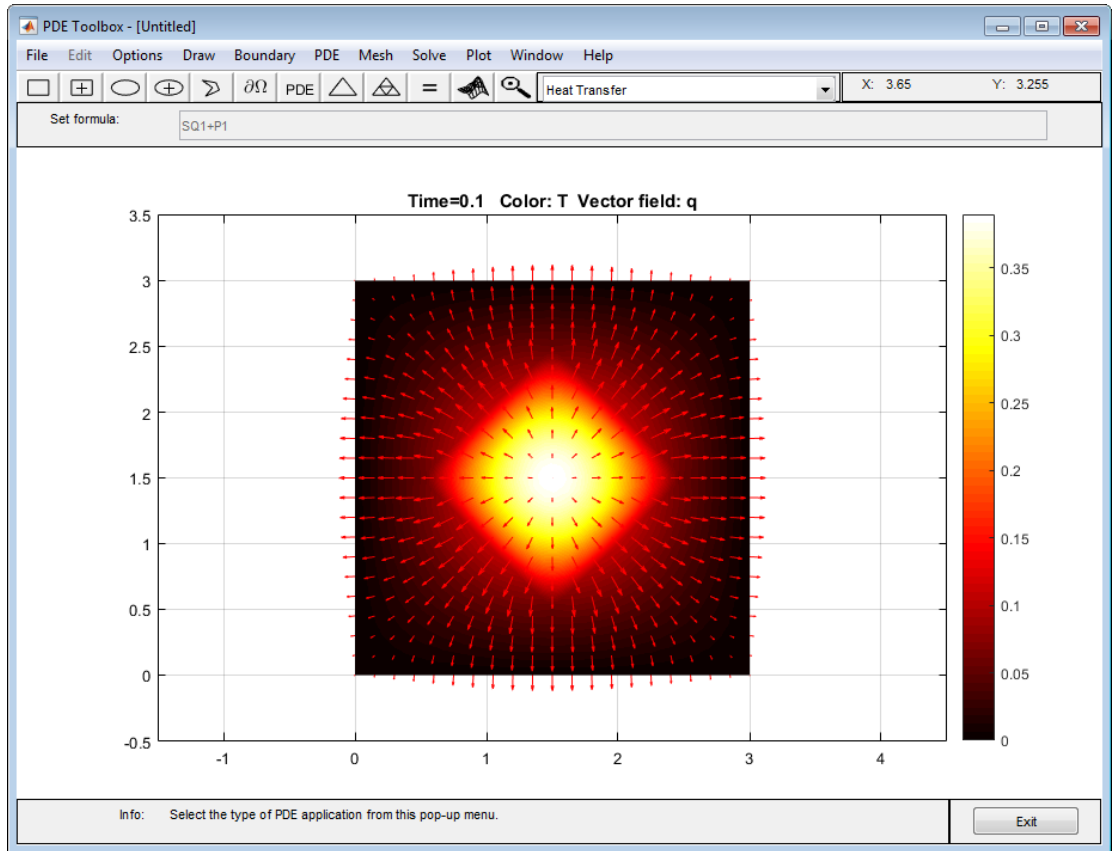
**Temperature and Heat Flux**

# 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

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

**3-107**

where $\epsilon$ is the emissivity of the face and $\sigma$ 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 $\rho$ 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;
model = 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 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(model,g);
```

Plot the geometry and display the edge labels for use in the boundary condition definition.

```
figure;
pdegplot(model,'EdgeLabels','on');
axis([-.1 1.1 -.1 1.1]);
title 'Geometry With Edge Labels Displayed';
```



Geometry With Edge Labels Displayed
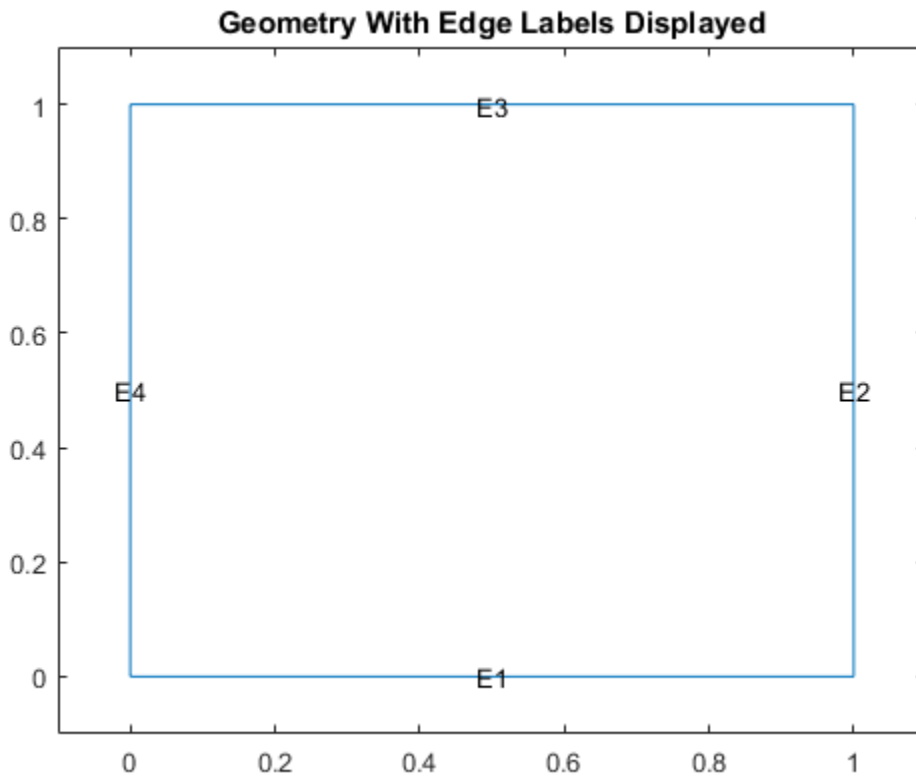
### Definition of PDE Coefficients

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(model,'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(model,'dirichlet','Edge',1,'u',1000);
```

### Initial guess

The initial guess is defined below.

```
setInitialConditions(model,0);
```

### Mesh

Create the triangular mesh on the square with approximately ten elements in each direction.

```
hmax = .1; % element size
msh = generateMesh(model,'Hmax',hmax);
figure;
pdeplot(model);
axis equal
```

```
title 'Plate With Triangular Element Mesh'
xlabel 'X-coordinate, meters'
ylabel 'Y-coordinate, meters'
```
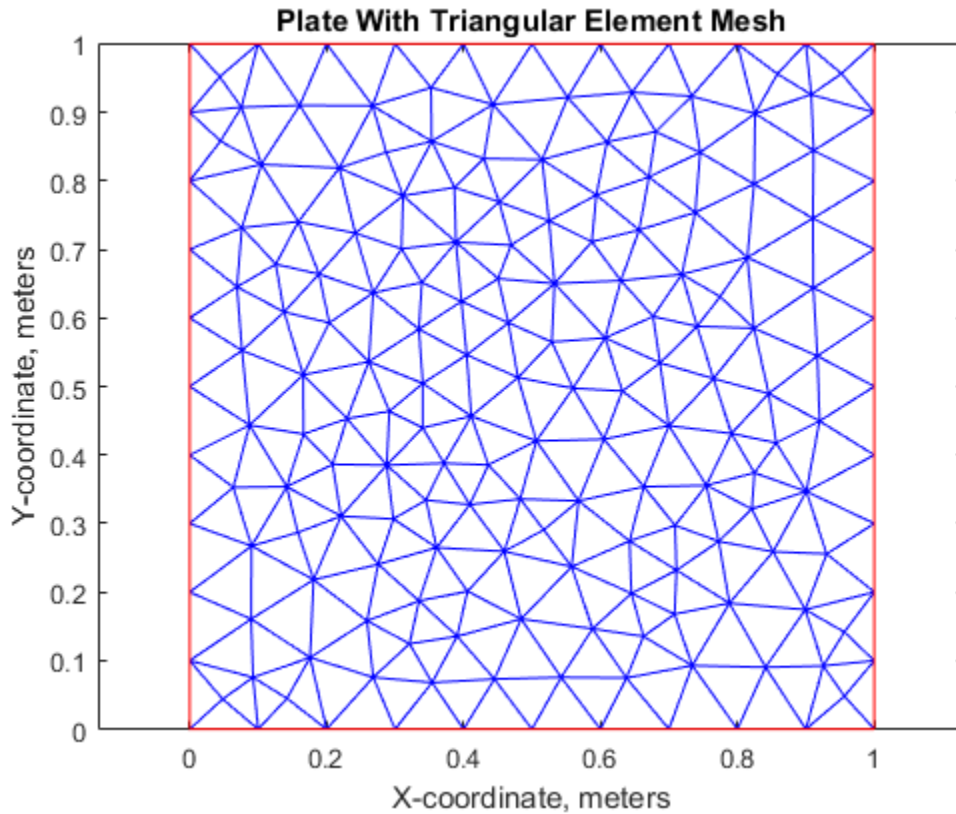


### Steady State Solution

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(model);
u = R.NodalSolution;
figure;
```

```
pdeplot(model,'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 'Y-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
```
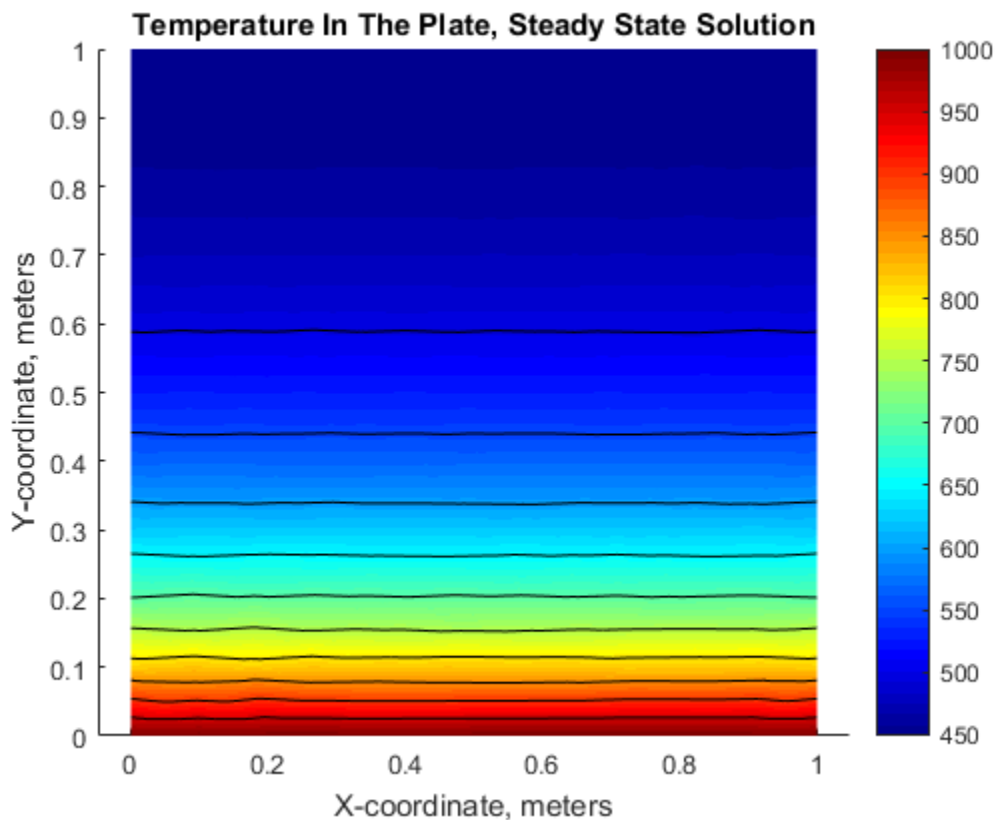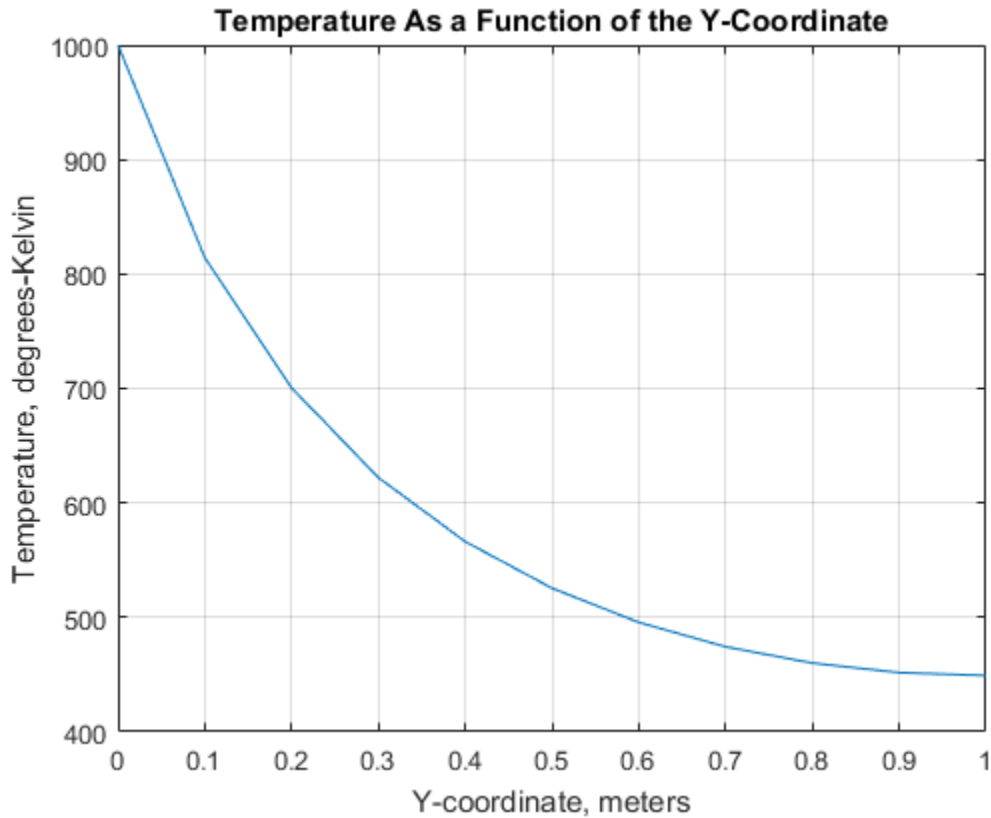
**Transient Solution**

Include the "d" coefficient.

```
specifyCoefficients(model,'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;
```

Set the initial temperature on the bottom edge E1 to the value of the constant BC, 1000 K.

```
setInitialConditions(model,1000,'edge',1);
```

Set solver options.

```
model.SolverOptions.RelativeTolerance = 1.0e-3;
model.SolverOptions.AbsoluteTolerance = 1.0e-4;
```

`solvepde` automatically picks the parabolic solver to obtain the solution.

```
R = solvepde(model,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(model,'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(t=%5.1f secs)=%5.1f degrees-K\n', ...
  tlist(1,end), u(4,end));
```

```
Temperature at the top edge(t=5000.0 secs)=441.3 degrees-K
```

Temperature Along the Top Edge of the Plate as a Function of Time
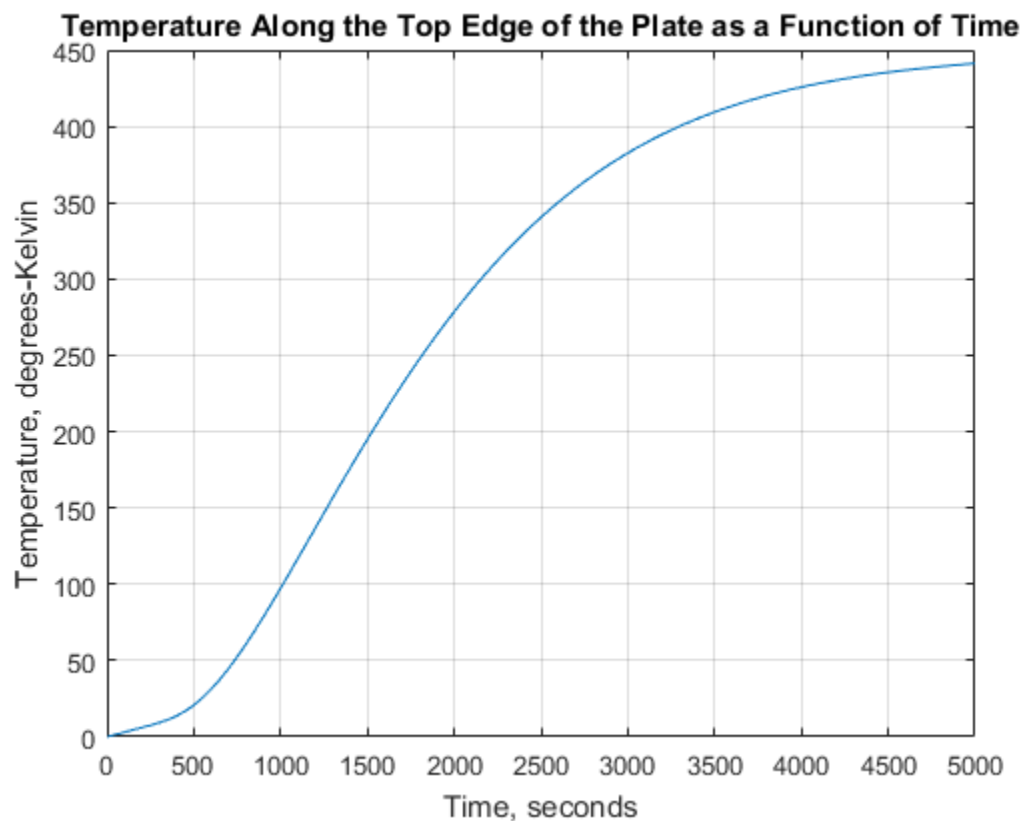
**Temperature In The Plate, Transient Solution( 5000 seconds)**
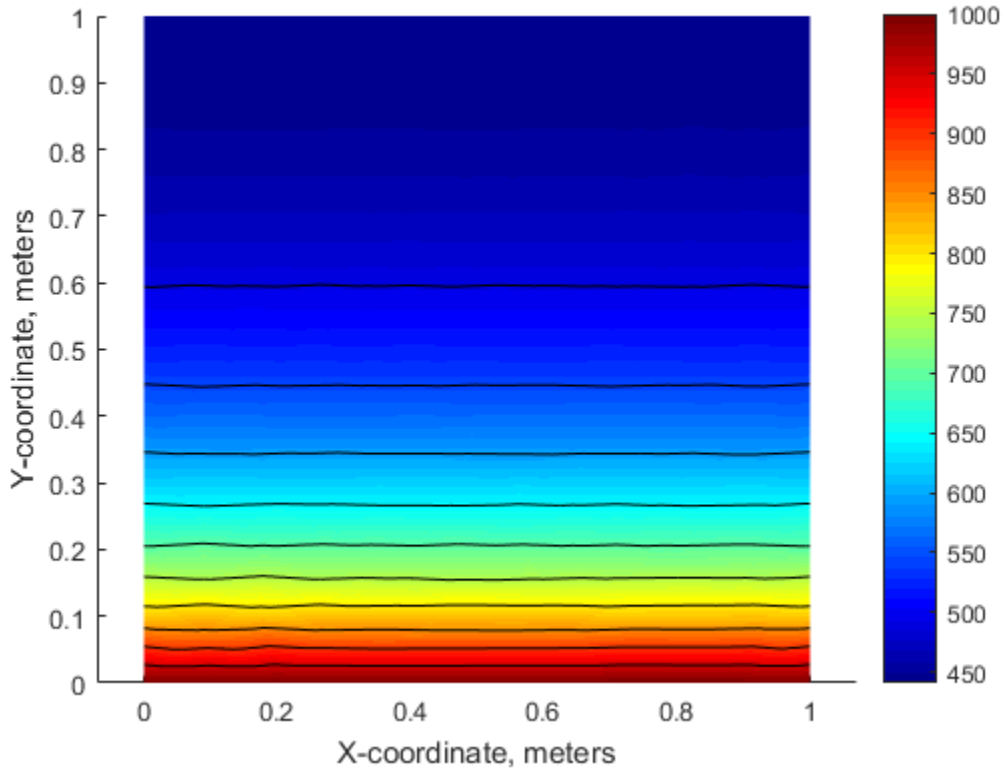


**Summary**

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.

# Solve Poisson's Equation on a Unit Disk: PDE App

This example shows how to solve a simple elliptic PDE in the form of Poisson's equation on a unit disk and evaluate the error of the numeric solution. This example uses the PDE app. For a programmatic workflow, see "Solve Poisson's Equation on a Unit Disk" on page 3-120.

The problem formulation is $-\Delta U = 1$ in $\Omega$, $U = 0$ on $\partial\Omega$, where $\Omega$ is the unit disk. The exact solution is

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

To solve this problem in the PDE app, follow these steps:

1   Open the PDE app by using the `pdetool` command.

2   Display grid lines by selecting **Options** > **Grid**.

3   Align new shapes to the grid lines by selecting **Options** > **Snap**.

4   Draw a circle with the radius 1 and the center at (0,0). To do this, first click the

$\oplus$ button. Then click the origin using the right mouse button and drag to draw a circle. The right mouse button constrains the shape you draw to be a circle rather than an ellipse. If the circle is not a perfect unit circle, double-click it. In the resulting dialog box, specify the exact center location and radius of the circle.

5   Check that the application mode is set to **Generic Scalar**.

6   Specify the boundary conditions. To do this, switch to boundary mode by clicking the

$\partial\Omega$ button or selecting **Boundary** > **Boundary Mode**. Select all boundaries by selecting **Edit** > **Select All**. Then select **Boundary** > **Specify Boundary Conditions** and specify the Dirichlet boundary condition $u = 0$. This boundary condition is the default one (`h = 1, r = 0`), so you do not need to change it.

7   Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Specify `c = 1`, `a = 0`, and `f = 1`.

8   Specify the maximum edge size for the mesh by selecting **Mesh** > **Parameters**. Set the maximum edge size to 0.1.

**9**
Initialize the mesh by selecting **Mesh > Initialize Mesh** or clicking the △ button.

**10** Solve the PDE by selecting **Solve > Solve PDE** or clicking the **=** button on the toolbar. The toolbox assembles the PDE problem, solves it, and plots the solution.
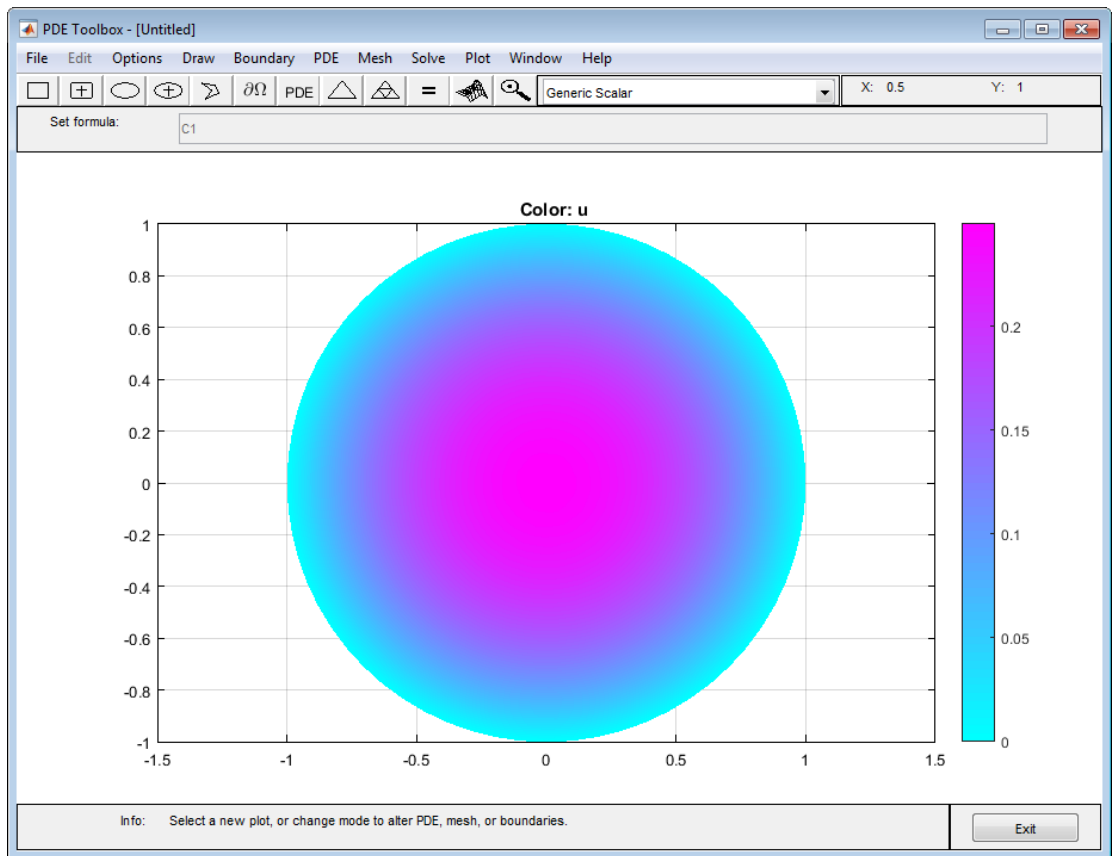


**11** Compare the numerical solution to the exact solution. To do this:

**a** Select **Plot > Parameters**.

**b** In the resulting dialog box, select `user entry` from the **Color** drop-down menu.

**c** Type the MATLAB expression `u-(1-x.^2-y.^2)/4` in the **User entry** field to plot the absolute error in the solution.
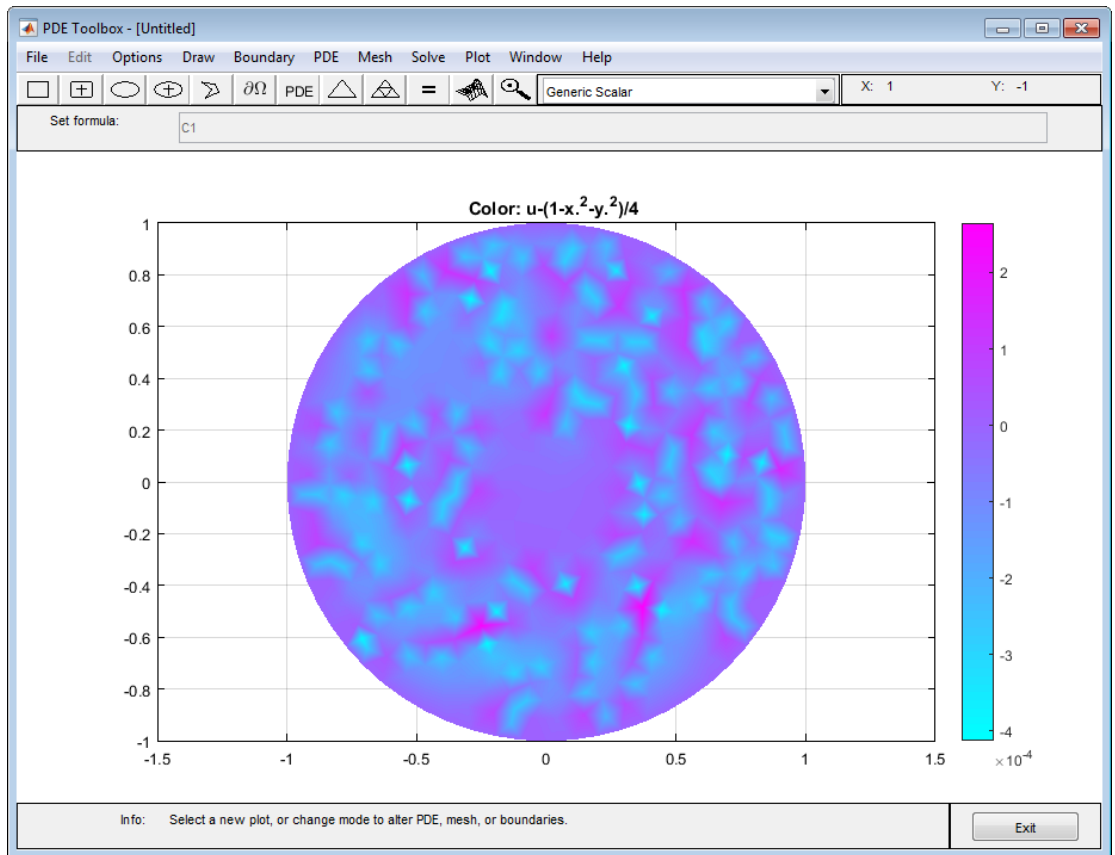
You can export the mesh data and the solution to the MATLAB workspace by selecting **Mesh** > **Export Mesh** and **Solve** > **Export Solution**, respectively.

# Solve Poisson's Equation on a Unit Disk

This example shows how to solve Poisson's equation using the programmatic workflow. For the PDE app solution, see "Solve Poisson's Equation on a Unit Disk: PDE App". The problem formulation is $-\Delta u = 1$ in $\Omega$, $u = 0$ on $\delta\Omega$, where $\Omega$ is the unit disk. The exact solution is

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

The code compares the solution obtained with the PDE Toolbox with the exact solution, and refines the mesh until the solutions are close.

First, create a function that parameterizes the 2-D geometry — in this case, a unit circle. The `circleg.m` file contains a function that returns the coordinates of points on the unit circle's boundary. You can display the file by using the command `type circleg`.

Create the PDE model and include the geometry.

```
model = createpde();
geometryFromEdges(model,@circleg);
```

Specify zero Dirichlet boundary conditions on all edges.

```
applyBoundaryCondition(model,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
```

Specify the coefficients.

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

Create a mesh with target maximum element size 0.1.

```
generateMesh(model,'Hmax',0.1);
```

Solve the PDE and plot the solution:

```
results = solvepde(model);
u = results.NodalSolution;
pdeplot(model,'XYData',u)
title('Numerical Solution');
xlabel('x')
ylabel('y')
```

**Numerical Solution**
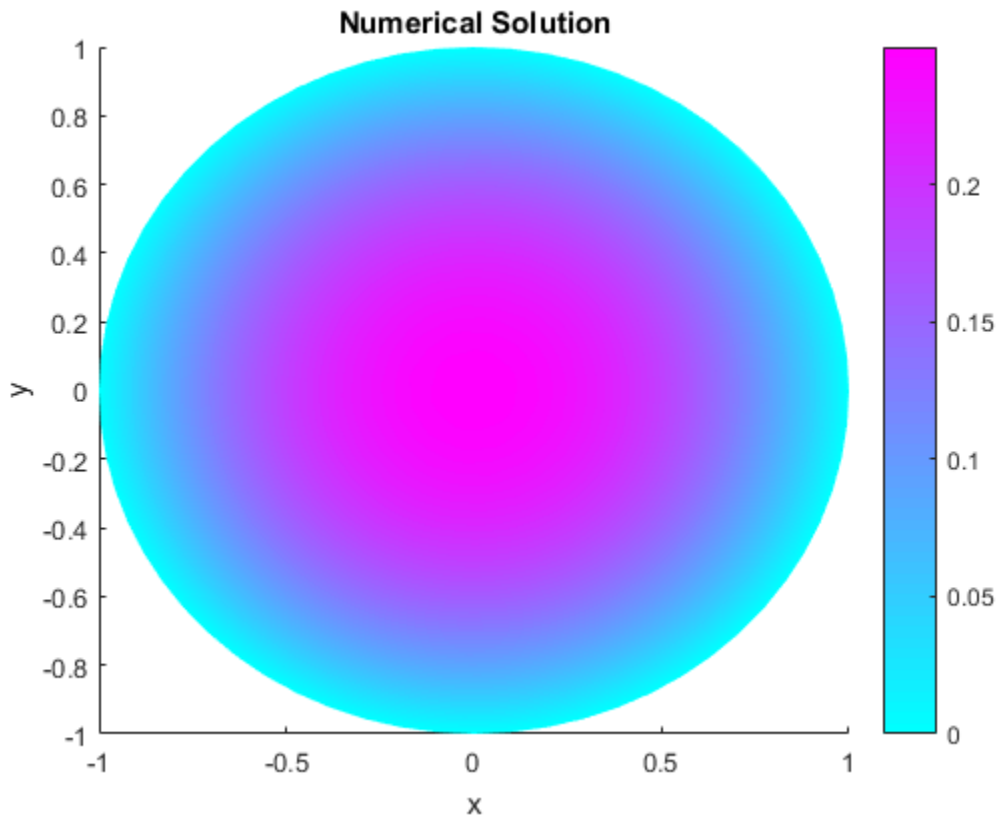


Compare this result with the exact analytical solution and plot the error.

```
p = model.Mesh.Nodes;
exact = (1 - p(1,:).^2 - p(2,:).^2)/4;
pdeplot(model,'XYData',u - exact')
title('Error');
xlabel('x')
ylabel('y')
```
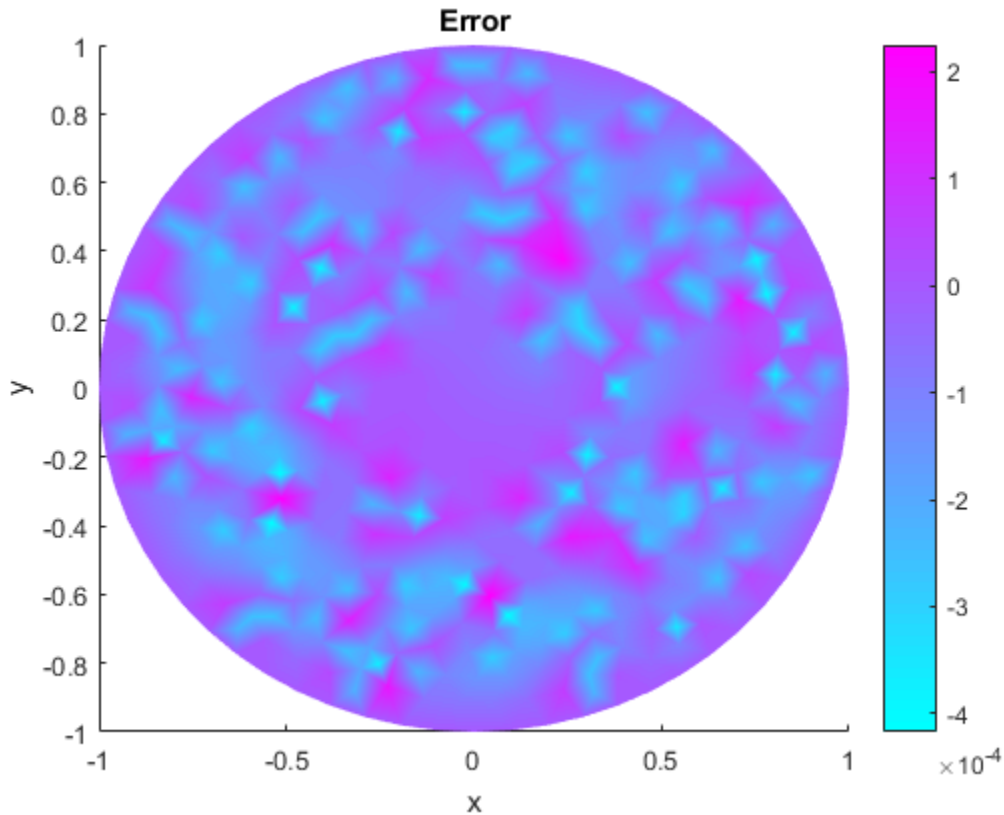
Now use the coarser mesh, refining it in each iteration. Solve the equation using the original coarse and refined meshes, each time comparing the result with the exact solution.

```
hmax = 1;
error = [];
err = 1;
while err > 0.001 % run until error <= 0.001
    generateMesh(model,'Hmax',hmax); % refine mesh
    results = solvepde(model);
    u = results.NodalSolution;
    p = model.Mesh.Nodes;
    exact = (1 - p(1,:).^2 - p(2,:).^2)/4;
    err = norm(u - exact',inf); % compare with exact solution
```

```
    error = [error err]; % keep history of err
    hmax = hmax/2;
end
```

The value of the error decreases in each iteration.

```
plot(error,'rx','MarkerSize',12);
ax = gca;
ax.XTick = 1:numel(error);
title('Error History');
xlabel('Iteration');
ylabel('Norm of Error');
```
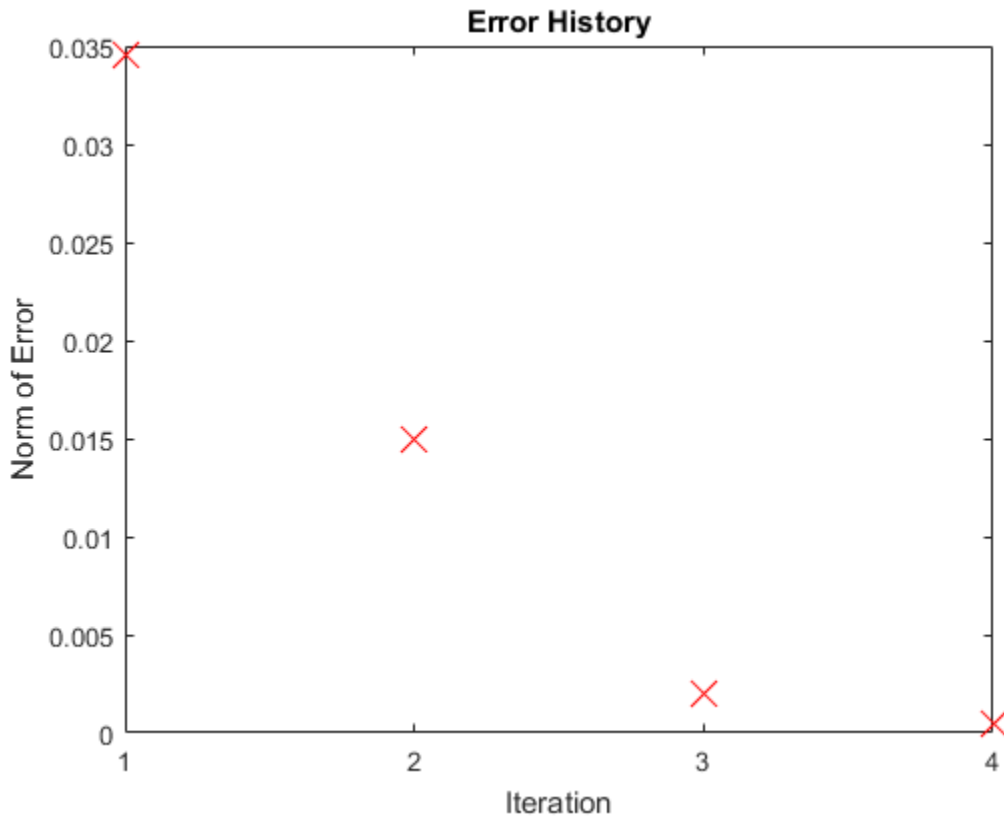
# Scattering Problem

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.
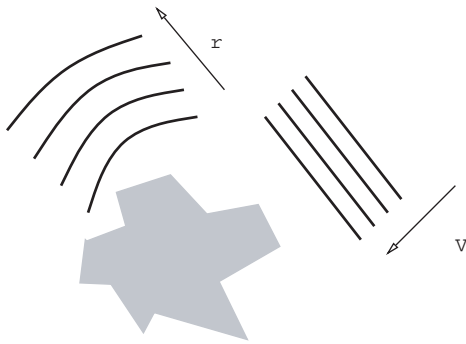


We assume that the medium is homogeneous so that the wave speed is constant, *c*.

---

**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 $\omega$, the frequency $f$, and the wavelength $\lambda$ 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 $\xi$-direction only ($\xi$ 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 $\xi$-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:

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

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.
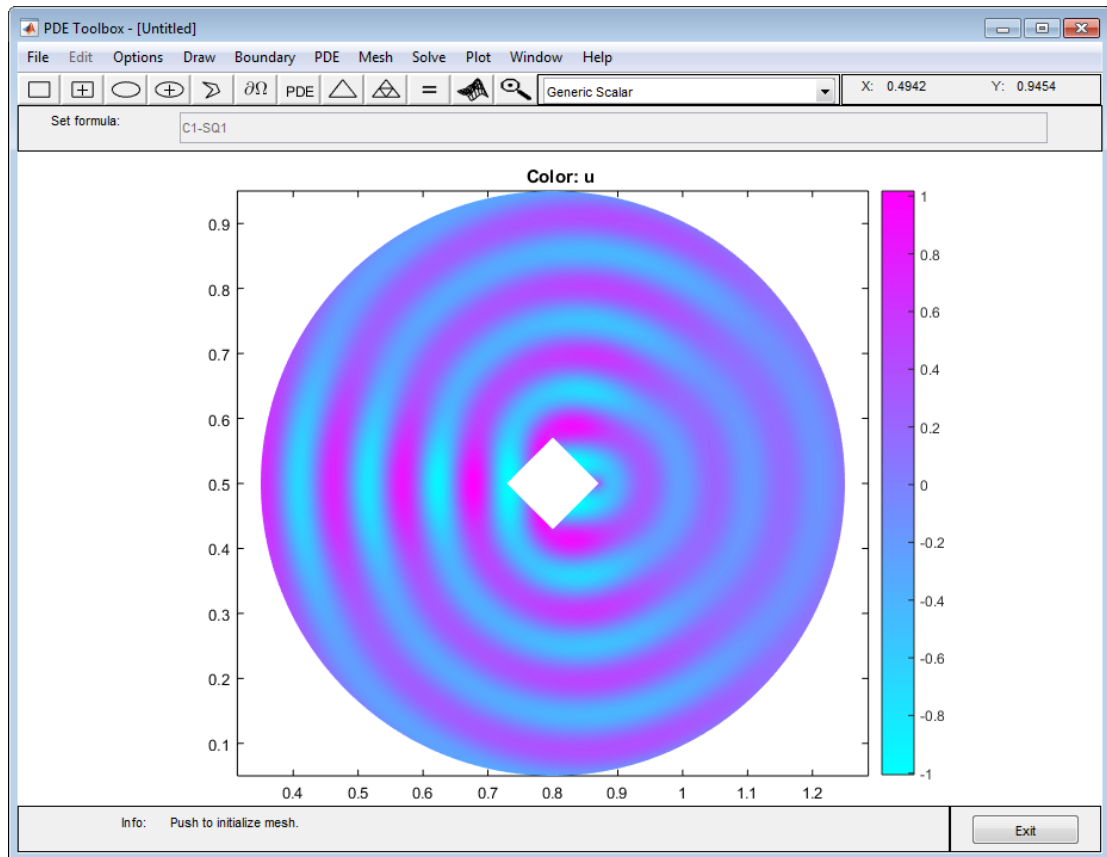


The propagation of the reflected waves is computed as
$\mathrm{Re}(r(x,y)e^{-i\omega t})$,

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 $\Omega = \{(x, y) \mid x^2 + y^2 \leq 1\}$, 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 $\nabla u$, 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**
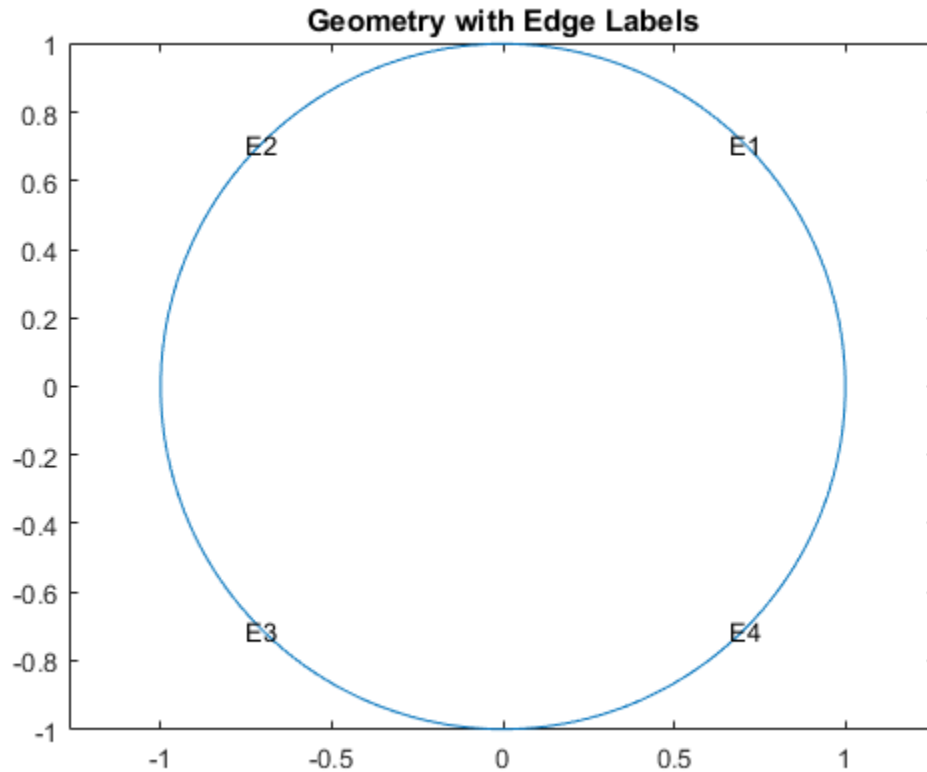


### Specify PDE Coefficients

```
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,'dirichlet',...
                              'Edge',1:model.Geometry.NumEdges,...
```

```
                                      'u',bcMatrix);
```

**Generate Mesh**

```
generateMesh(model,'Hmax',0.1);
figure;
pdemesh(model);
axis equal
```
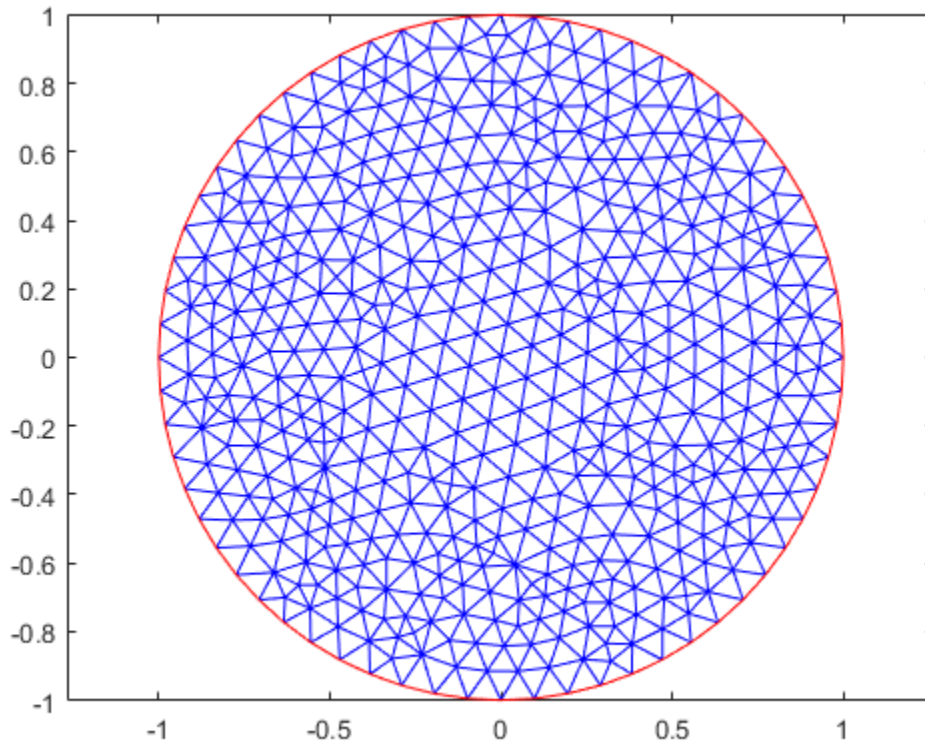


**Solve PDE**

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'
```
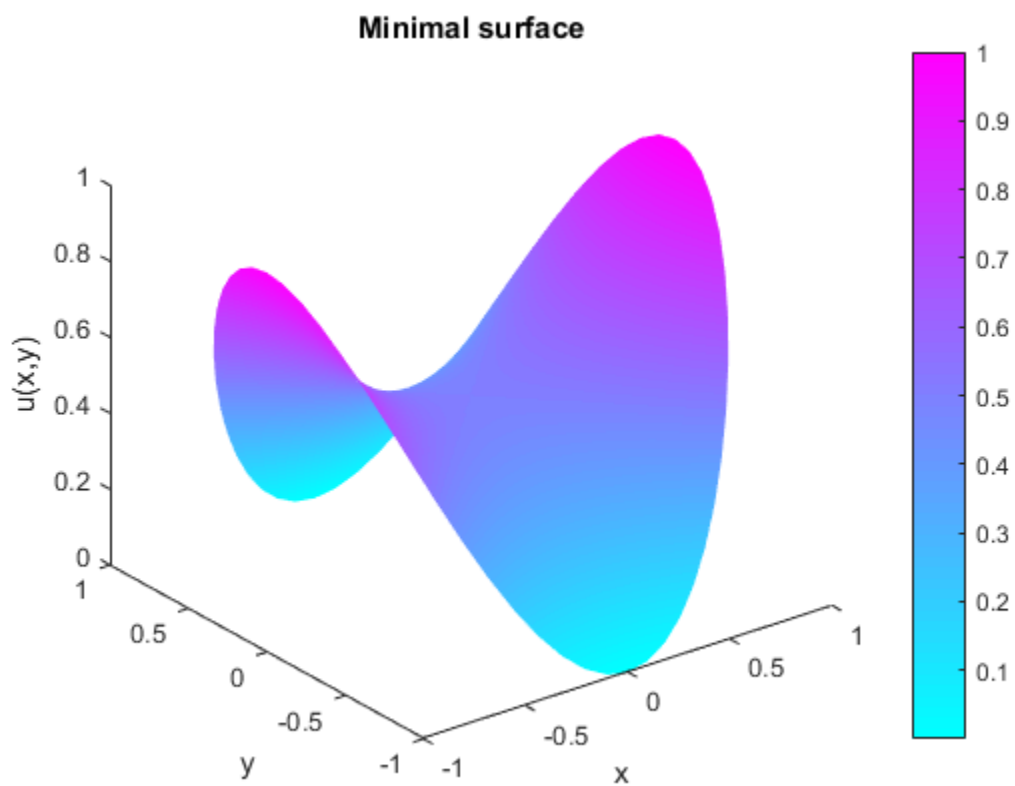
# Domain Decomposition Problem

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 $\Omega$ is the disjoint union of some subdomains $\Omega_1, \Omega_2, \ldots, \Omega_n$. Then you could renumber the nodes of a mesh on $\Omega$ 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:

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

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 $K_j$, $B_j$, 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:

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

$$u_1 = K_1^{-1} \left( f_1 - B_1^T u_c \right)$$

...

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)
```

```
ans =

    1.7397e-04
```
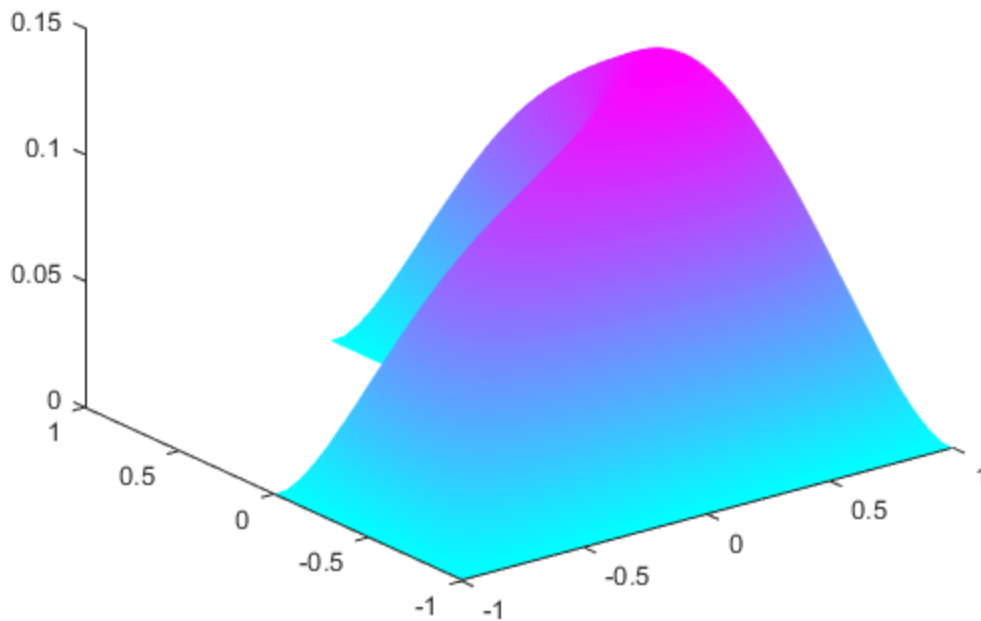


For the entire example, see "Poisson's Equation Using Domain Decomposition".

# Heat Equation for a Block with Cavity: PDE App

This example shows how to solve a heat equation that describes the diffusion of heat in a body. This example uses the PDE app. For programmatic workflow, see "Heat Equation for a Block with Cavity" on page 3-144.

Consider a 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 block, heat flows from the block to the surrounding air at a constant rate, for example -10 $W/m^2$. All the other boundaries are insulated. The temperature in the block at the starting time $t_0 = 0$ is 0 degrees. The goal is to model the heat distribution during the first five seconds.

The PDE governing this problem is a parabolic heat equation. Partial Differential Equation Toolbox solves the generic parabolic PDE of the form

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

The heat equation has the form:

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

To solve this problem in the PDE app, follow these steps:

1  Open the PDE app by using the `pdetool` command.

   pdetool

2  Model the geometry: draw a rectangle with corners (-0.5,-0.8), (0.5,-0.8), (0.5,0.8), and (-0.5,0.8) and a rectangle with corners (-0.05,-0.4), (0.05,-0.4), (0.05,0.4), and (-0.05,0.4). Draw the first rectangle by using the `pderect` function.

   pderect([-0.5 0.5 -0.8 0.8])

3  Display grid lines with extra ticks at `-0.05` and `0.05`. To do this, select **Options** > **Grid Spacing**, clear the **Auto** checkbox, and enter **X-axis extra ticks** at `-0.05` and `0.05`. Then select **Options** > **Grid**.
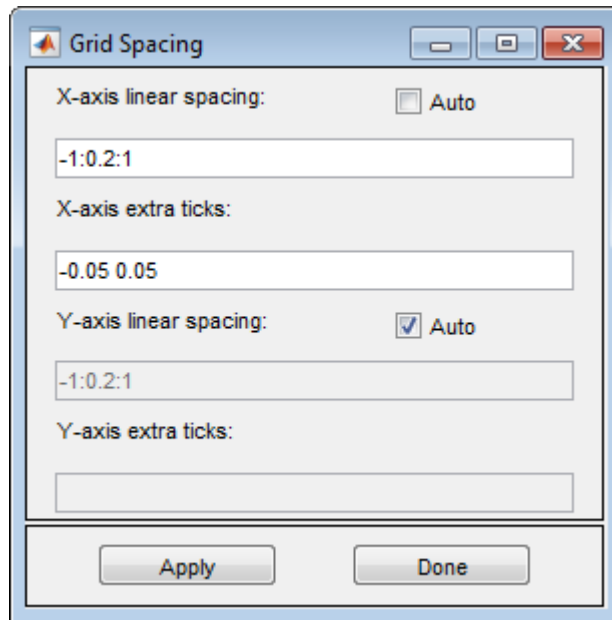
**4** Set the *x*-axis limit to [-0.6 0.6] and *y*-axis limit to [-1 1]. To do this, select **Options** > **Axes Limits** and set the corresponding ranges.

**5** Select **Options** > **Snap** to align any new shape to the grid lines. Then draw the rectangle with corners (-0.05,-0.4), (0.05,-0.4), (0.05,0.4), and (-0.05,0.4)

**6** Model the geometry by entering R1-R2 in the **Set formula** field.

**7** Check that the application mode is set to **Generic Scalar**.

**8** Specify the boundary conditions. To do this, switch to the boundary mode by selecting **Boundary** > **Boundary Mode**. Then select **Boundary** > **Specify Boundary Conditions** and specify the Neumann boundary condition.

- For convenience, first specify the insulating Neumann boundary condition $\partial u/\partial n$ = 0 for all boundaries. To do this, select all boundaries by using **Edit** > **Select All** and specify g = 0, q = 0.

- Specify the Dirichlet boundary condition $u$ = 100 for the left side of the block. To do this, specify h = 1, r = 100.

- Specify the Neumann boundary condition $\partial u/\partial n$ = −10 for the right side of the block. To do this, specify g = -10, q = 0.

**9** Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Heat equation is a parabolic equation, so select the **Parabolic** type of PDE. Specify `c = 1`, `a = 0`, `f = 0`, and `d = 1`.

**10** Initialize the mesh by selecting **Mesh** > **Initialize Mesh**. Refine the mesh by selecting **Mesh** > **Refine Mesh**.

**11** Set the initial value to 0, the solution time to 5 seconds, and compute the solution every 0.5 seconds. To do this, select **Solve** > **Parameters**. In the **Solve Parameters** dialog box, set time to `0:0.5:5`, and $u(t_0)$ to `0`.

**12** Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the `=` button on the toolbar. The app solves the heat equation at 11 different times from 0 to 5 seconds and displays the heat distribution at the end of the time span.

**13** Plot isothermal lines using a contour plot and the heat flux vector field using arrows and change the colormap to `hot`. To do this:

    **a** Select **Plot** > **Parameters**.

    **b** In the resulting dialog box, select the **Color**, **Contour**, and **Arrows** options. Select `-c*grad(u)` from **Arrows** drop-down menu.

    **c** Change the colormap to `hot` by using the corresponding drop-down menu in the same dialog box.
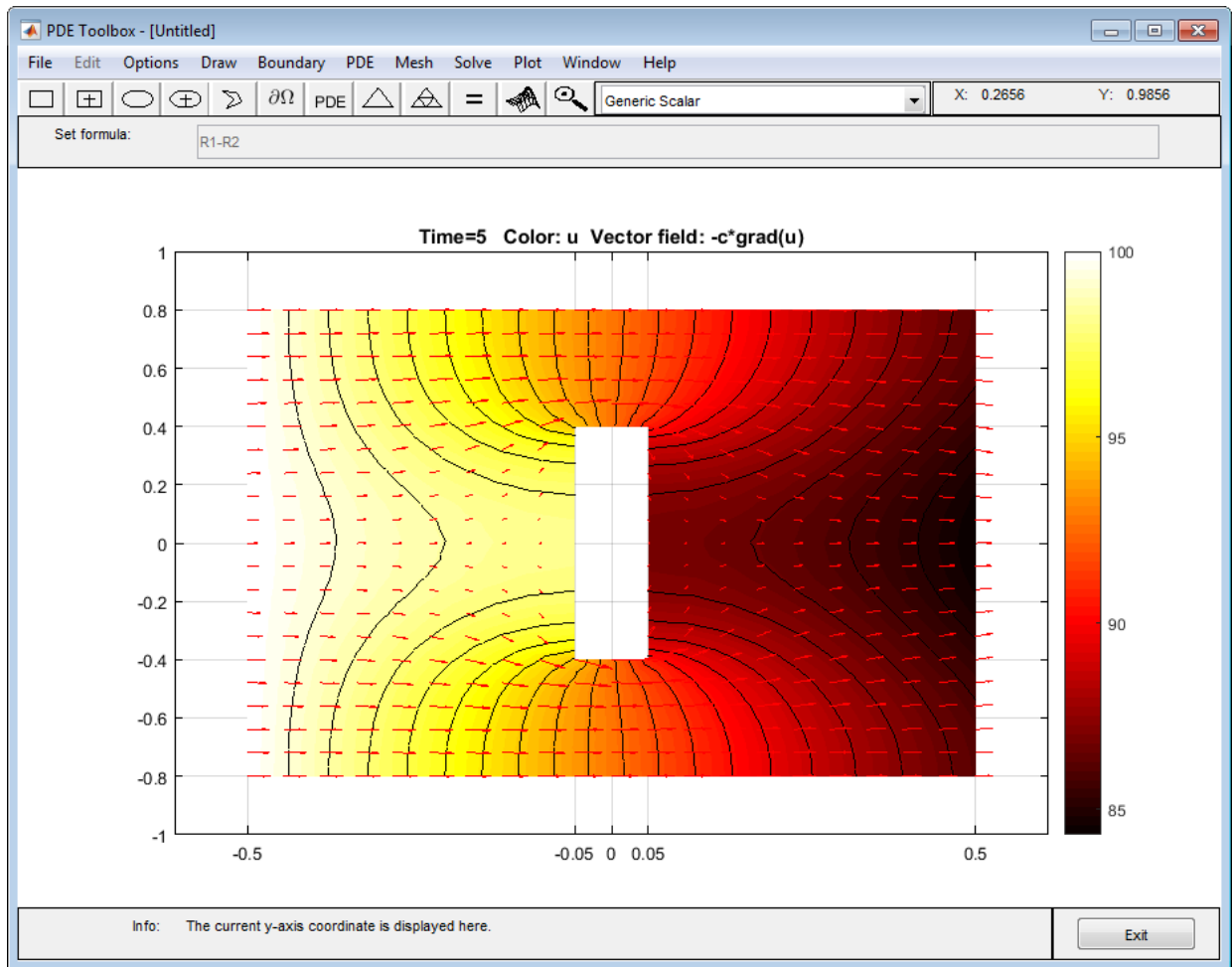
14  Use an animated plot to visualize the dynamic behavior of the temperature. For this, select **Plot** > **Parameters** and then select the **Animation** option.

15  The temperature in the block rises very quickly. To improve the animation and focus on the first second, change the list of times to the MATLAB expression `logspace(-2,0.5,20)`. To do this, select **Solve** > **Parameters**. In the **Solve Parameters** dialog box, set time to `logspace(-2,0.5,20)`.

**16** You can explore the solution by varying the parameters of the model and plotting the results. For example, change the heat capacity coefficient d and the heat flow at the right boundary to see how these parameters affect the heat distribution.

# Heat Equation for a Block with Cavity

This example shows how to solve for the heat distribution in a block with cavity using the programmatic workflow. For the PDE app solution, see "Heat Equation for a Block with Cavity: PDE App".

Consider a 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 block, heat flows from the block to the surrounding air at a constant rate, for example $-10W/m^2$. All the other boundaries are insulated. The temperature in the block at the starting time $t_0 = 0$ is 0 degrees. The goal is to model the heat distribution during the first five seconds.

The `crackg.m` file describes the geometry of the block.

```
thermalmodel = createpde('thermal','transient');
geometryFromEdges(thermalmodel,@crackg);
```

Plot the geometry with edge labels.

```
pdegplot(thermalmodel,'EdgeLabels','on');
ylim([-1,1])
axis equal
```
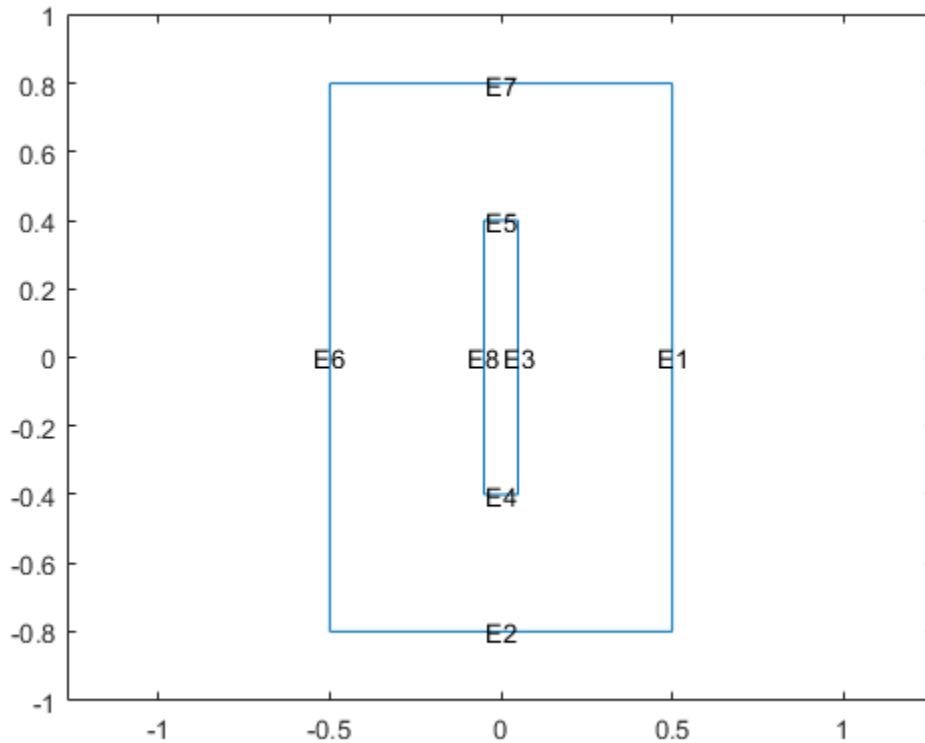
Assume that the block has these thermal properties.

```
thermalProperties(thermalmodel,'ThermalConductivity',1,...
                               'MassDensity',1,...
                               'SpecificHeat',1);
```

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. The toolbox uses the default insulating boundary condition for all other boundaries.

```
thermalBC(thermalmodel,'Edge',6,'Temperature',20);
thermalBC(thermalmodel,'Edge',1,'HeatFlux',-10);
```

Set an initial value of 0 for the temperature.

```
thermalIC(thermalmodel,0);
```

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(thermalmodel);
thermalresults = solve(thermalmodel,tlist);
T = thermalresults.Temperature;
```

Compute the heat flux density. Plot the solution at $t = 5.0$ seconds with isothermal lines using a contour plot, and plot the heat flux vector field using arrows.

```
[qx,qy] = evaluateHeatFlux(thermalresults);
pdeplot(thermalmodel,'XYData',T(:,end),'Contour','on',...
                     'FlowData',[qx(:,end),qy(:,end)],'ColorMap','hot')
```
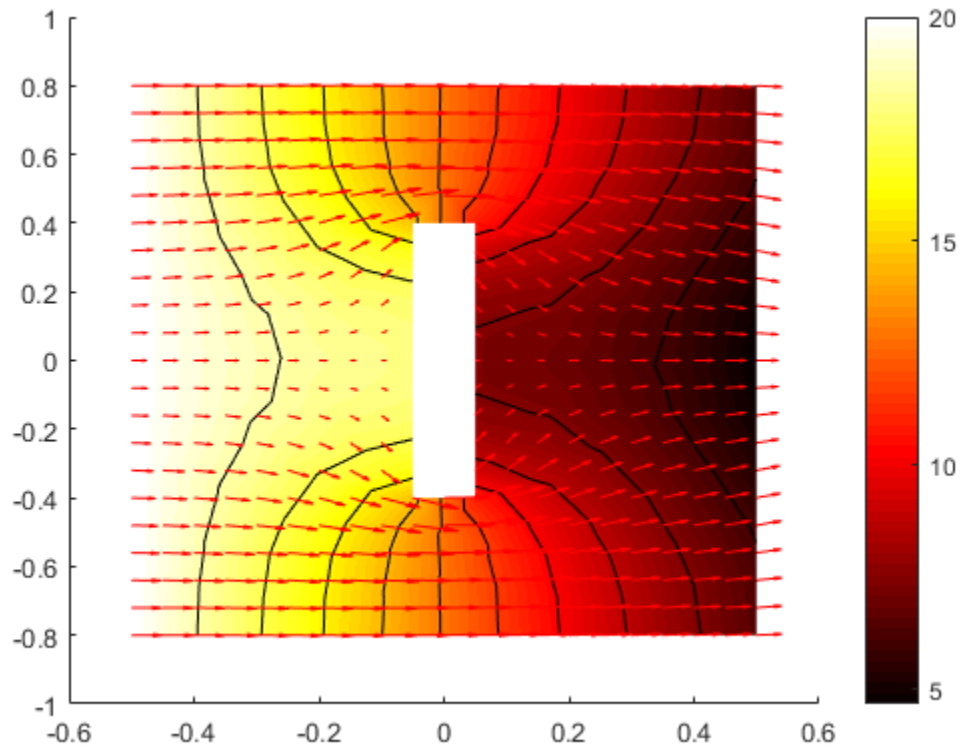
# Heat Distribution in a Circular Cylindrical Rod

This example shows how a 3-D axisymmetric model can be analyzed using a 2-D model. The model geometry, material properties, and boundary conditions must all be symmetric about a single axis for this simplification from 3-D to 2-D to be appropriate. Because of this symmetry, a cylindrical coordinate system is the most convenient form for defining the partial differential equation. However, Partial Differential Equation Toolbox™ expects the equations in a Cartesian system. One of the main goals of this example is to show how to express the PDE defined in a cylindrical system in a Cartesian form that Partial Differential Equation Toolbox™ can handle.

This particular example shows heat transfer in a rod with a circular cross section. There is a heat source at the left end of the rod and a fixed temperature at the right end. The outer surface of the rod exchanges heat with the environment due to convection. In addition, heat is generated within the rod due to radioactive decay.

We would like to calculate the temperature in the rod as a function of time. The parabolic equation describing heat transfer is

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

where $\rho, C, \text{and } k$ are the density, specific heat, and thermal conductivity of the material, respectively, $u$ is the temperature, and $q$ is the heat generated in the rod.

Since the problem is axisymmetric, it is convenient to write this equation in a cylindrical coordinate system.

$$\rho C \frac{\partial u}{\partial t} - \frac{1}{r} \frac{\partial}{\partial r} \left( kr \frac{\partial u}{\partial r} \right) - \frac{1}{r^2} \frac{\partial}{\partial \theta} \left( k \frac{\partial u}{\partial \theta} \right) - \frac{\partial}{\partial z} \left( k \frac{\partial u}{\partial z} \right) = q,$$

where $r, \theta$, and $z$ are the three coordinate variables of the cylindrical system. Because the problem is axisymmetric, $\partial u / \partial \theta = 0$ and after multiplying by $r$ the equation can be rewritten

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

The equation can be converted to the form supported by PDE Toolbox if $r$ is defined as $y$ and $z$ is defined as $x$. Rewriting the above equation gives

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

### Steady State Solution

In transient problems of this type it is often useful to first compute the steady state solution-- the solution to the time-independent, elliptic equation. If the final time in the transient analysis is sufficiently large, the transient solution at the final time should be close to this steady state solution. This provides a valuable check on the accuracy of the transient analysis.

Create a thermal model for steady-state analysis.

```
thermalModelS = createpde('thermal');
```

The 2-D model is a rectangular strip whose y-dimension extends from the axis of symmetry to the outer surface and x-dimension extends over the actual length of the rod (from -1.5 m to 1.5 m). The geometry and mesh for this rectangular section are easily defined by specifying the x and y locations of the four corners as shown below.

```
g = decsg([3 4 -1.5 1.5 1.5 -1.5 0 0 .2 .2]');
```

Convert the geometry and append it to the thermal model.

```
geometryFromEdges(thermalModelS,g);
```

The rod is composed of a material with the following thermal properties.

```
k = 40; % thermal conductivity, W/(m-degree C)
rho = 7800; % density, kg/m^3
cp = 500; % specific heat, W-s/(kg-degree C)
q = 20000; % heat source, W/m^3
```

PDE Toolbox allows definition of the non-constant coefficients as function of spatial coordinates and solution.

```
kFunc = @(region,state) k*region.y;
cFunc = @(region,state) cp*region.y;
qFunc = @(region,state) q*region.y;
```

For a steady-state analysis, specify the thermal conductivity of the material.

```
thermalProperties(thermalModelS,'ThermalConductivity',kFunc);
```

Specify internal heat source.

```
internalHeatSource(thermalModelS,qFunc);
```

When defining boundary conditions below, it is necessary to know the edge numbers for the boundary edges of the geometry. A convenient way to obtain these edge numbers is to plot the geometry using `pdegplot` with option `edgeLabels` set to `'on'`.

```
figure
pdegplot(thermalModelS,'EdgeLabels','on');
axis equal
xlim([-2 2]);
title 'Rod Section Geometry With Edge Labels Displayed';
```



**Rod Section Geometry With Edge Labels Displayed**
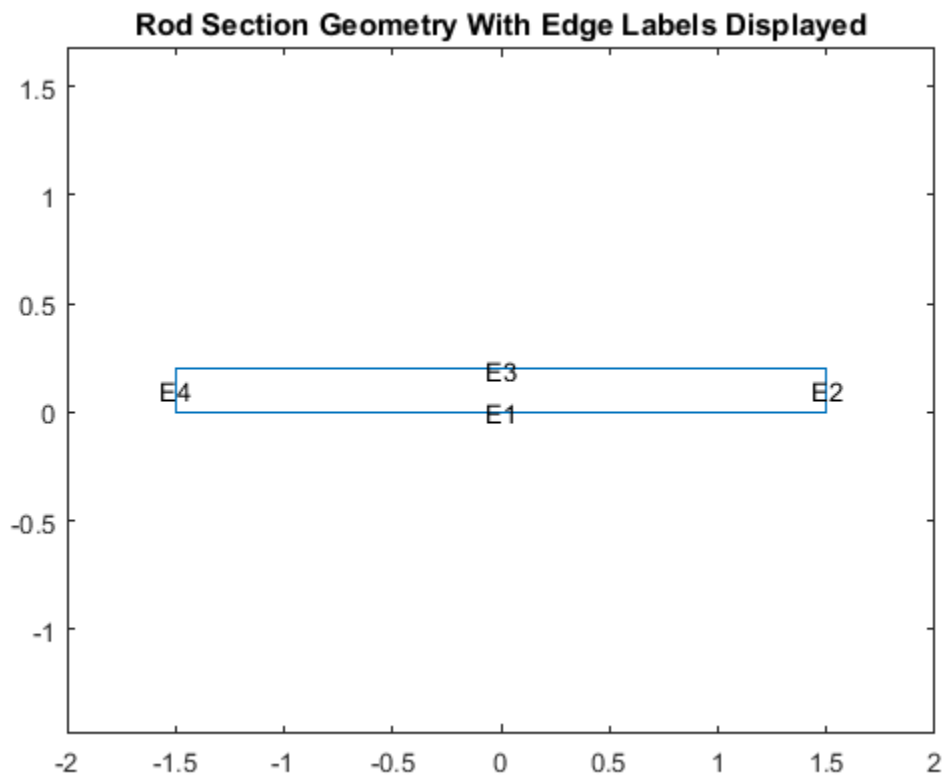
Define the boundary conditions. Edge 1, which is the edge at $y$ equal zero, is along the axis of symmetry so there is no heat transfered in the direction normal to this edge. This boundary is modeled by the default as an insulated boundary. Edge 2 is kept at a

constant temperature T = 100 C. Boundary conditions for the edges 3 and 4 are functions of y.

```
thermalBC(thermalModelS,'Edge',2,'Temperature',100);

outerCC = @(region,~) 50*region.y;
thermalBC(thermalModelS,'Edge',3,...
                        'ConvectionCoefficient',outerCC,...
                        'AmbientTemperature',100);

leftHF = @(region,~) 5000*region.y;
thermalBC(thermalModelS,'Edge',4,'HeatFlux',leftHF);
```

Generate the mesh.

```
generateMesh(thermalModelS,'Hmax',0.1);
figure;
pdeplot(thermalModelS);
axis equal
title 'Rod Section With Triangular Element Mesh'
```

**Rod Section With Triangular Element Mesh**
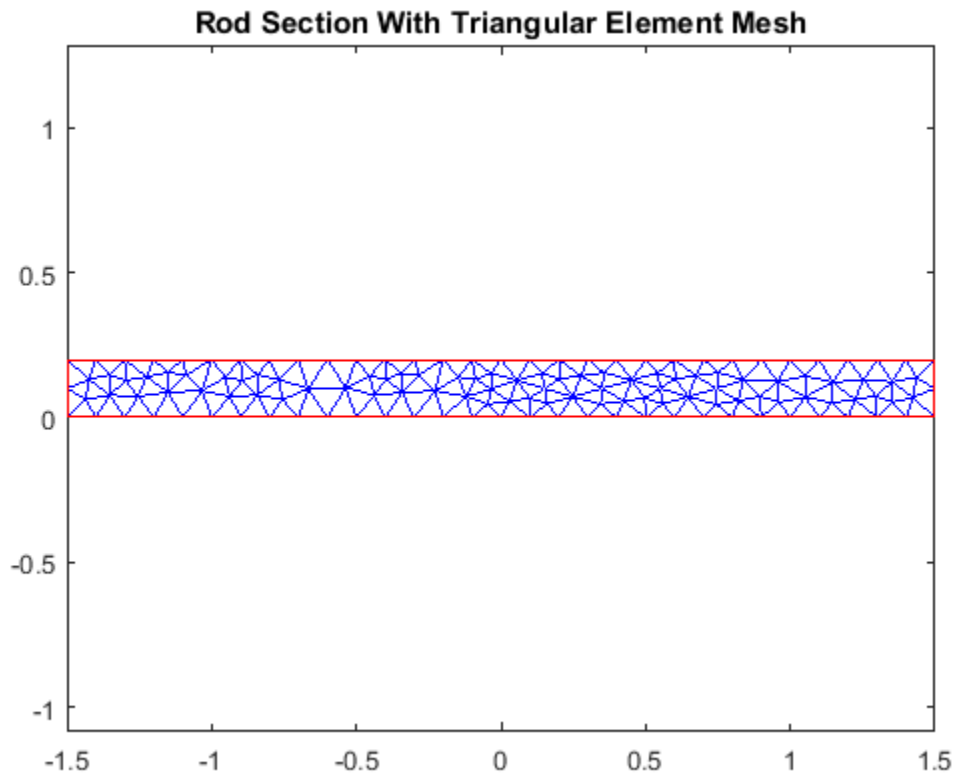


Solve the model and plot the result.

```
result = solve(thermalModelS);
T = result.Temperature;
figure;
pdeplot(thermalModelS,'XYData',T,'Contour','on');
axis equal
title 'Steady State Temperature';
```

**Steady State Temperature**
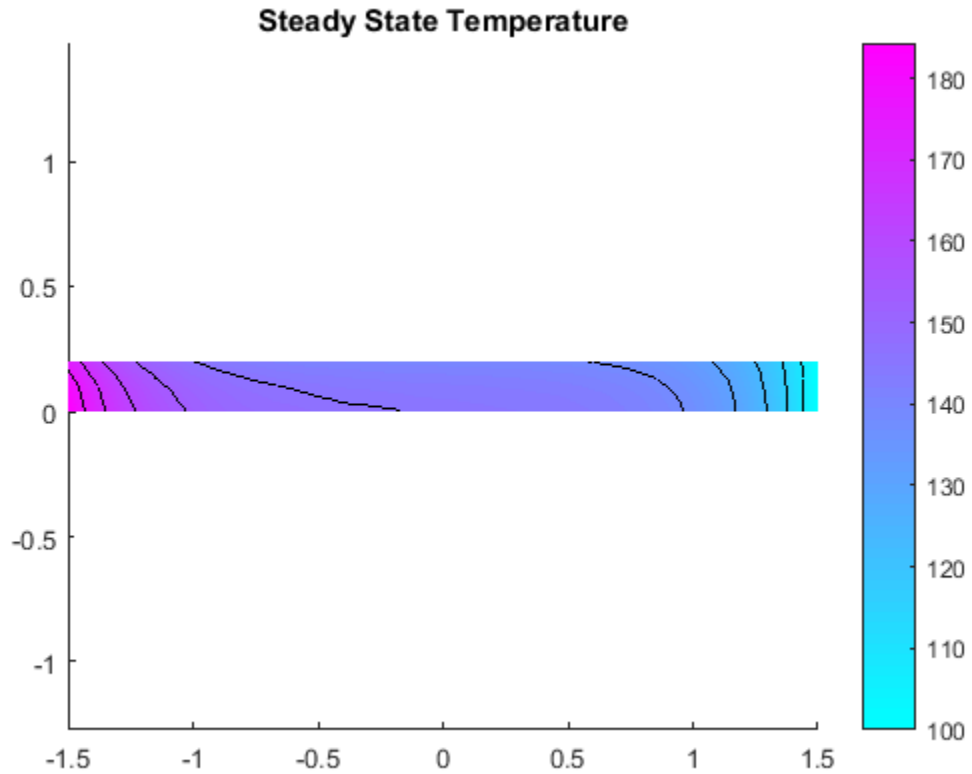
### Transient Solution

Create a thermal model for transient analysis, include the geometry, and mesh.

```
thermalModelT = createpde('thermal','transient');

g = decsg([3 4 -1.5 1.5 1.5 -1.5 0 0 .2 .2]');
geometryFromEdges(thermalModelT,g);

generateMesh(thermalModelT,'Hmax',0.1);
```

For a transient analysis, specify the thermal conductivity, mass density, specific heat of the material.

```
thermalProperties(thermalModelT,'ThermalConductivity',kFunc,...
                                'MassDensity',rho,...
                                'SpecificHeat',cFunc);
```

Specify internal heat source and boundary conditions.

```
internalHeatSource(thermalModelT,qFunc);

thermalBC(thermalModelT,'Edge',2,'Temperature',100);
thermalBC(thermalModelT,'Edge',3,...
                         'ConvectionCoefficient',outerCC,...
                         'AmbientTemperature',100);
thermalBC(thermalModelT,'Edge',4,'HeatFlux',leftHF);
```

Compute the transient solution for solution times from $t = 0$ to $t = 20000$ seconds. Initial temperature in the rod is zero.

```
tfinal = 20000;
tlist = 0:100:tfinal;
thermalIC(thermalModelT,0);
thermalModelT.SolverOptions.ReportStatistics = 'on';

result = solve(thermalModelT,tlist);
T = result.Temperature;

94 successful steps
0 failed attempts
190 function evaluations
1 partial derivatives
19 LU decompositions
189 solutions of linear systems
```

Plot the solution at `t = 20000`.

```
figure;
pdeplot(thermalModelT,'XYData',T(:,end),'Contour','on');
axis equal
title(sprintf('Transient Temperature at Final Time (%g seconds)',tfinal));
```

**Transient Temperature at Final Time (20000 seconds)**
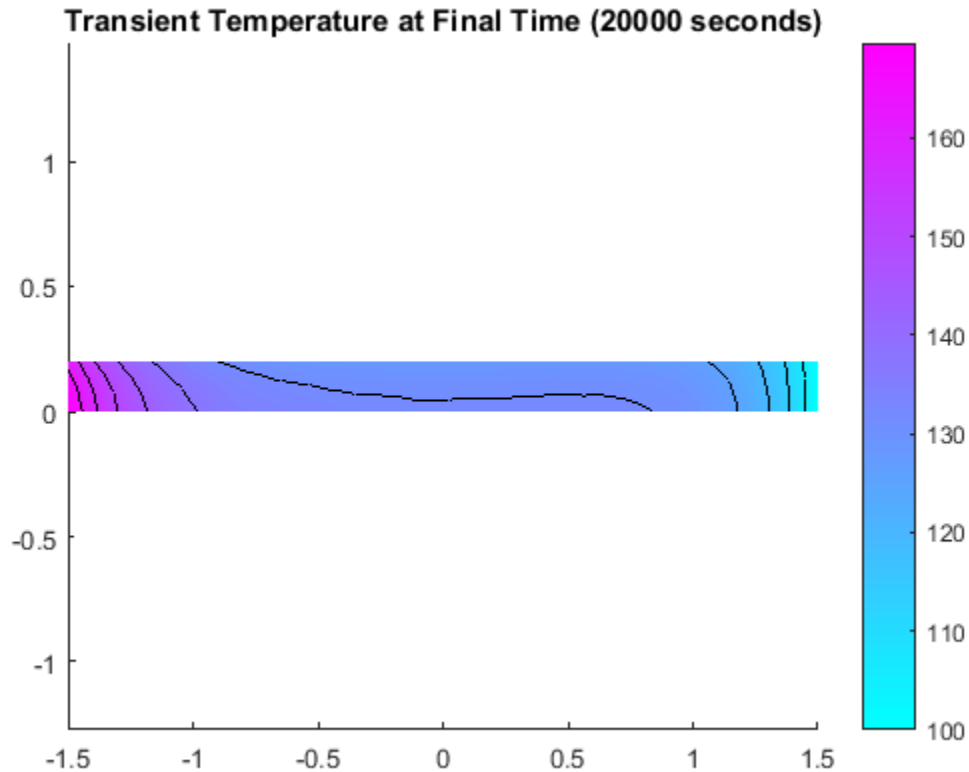


The steady state solution and the transient solution at 20000 seconds are in close agreement. This can be seen by comparing the two figures.

The third figure below shows the temperature at the left end of the rod as a function of time. The outer surface of the rod is exposed to the environment with a constant temperature of 100 degrees-C. Consequently, when the surface temperature of the rod is less than 100, the rod is being heated by the environment and when greater than 100, cooled. When the rod temperature is less than 100 degrees, the outer surface is slightly warmer than the inner axis but when the temperature is around 100 degrees, the outer surface becomes cooler than the interior of the rod.

Find nodes on the left end of the rod and on the center axis and outer surface using their coordinate values.

```
p = thermalModelT.Mesh.Nodes;
nodesLeftEnd  = find(p(1,:) < -1.5+eps);
nodeCenter = nodesLeftEnd(p(2,nodesLeftEnd) < eps);
nodeOuter = nodesLeftEnd(p(2,nodesLeftEnd) > 0.2-eps);

figure;
plot(tlist,T(nodeCenter,:));
hold all
plot(tlist,T(nodeOuter,:),'--');
title 'Temperature at Left End as a Function of Time'
xlabel 'Time, seconds'
ylabel 'Temperature, degrees-C'
grid on;
legend('Center Axis','Outer Surface','Location','SouthEast');
```
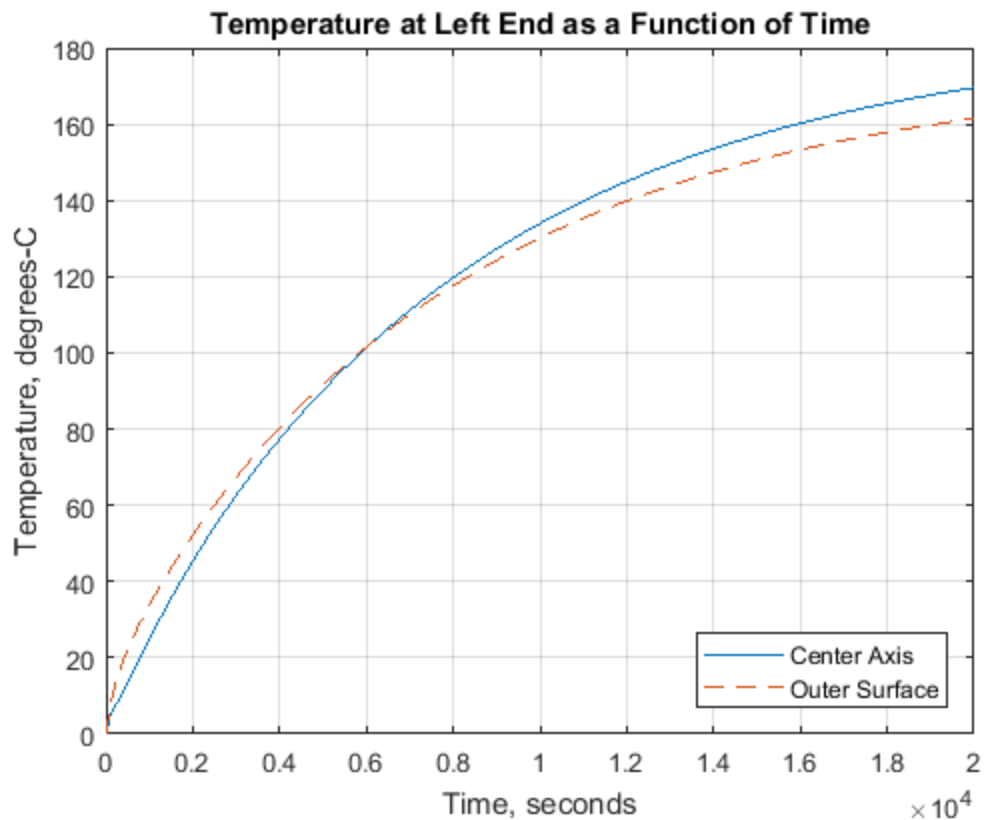
# Heat Distribution in a Circular Cylindrical Rod: PDE App

Solve a 3-D parabolic PDE problem by reducing the problem to 2-D using coordinate transformation. This example uses the PDE app. For the command-line solution, see Heat Distribution in a Circular Cylindrical Rod.

Consider a cylindrical radioactive rod. Heat is continuously added at the left end of the rod, while 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. Assuming that the initial temperature is zero leads to the following equation:

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

Here, $\rho$, $C$, and $k$ are the density, thermal capacity, and thermal conductivity of the material, $u$ is the temperature, and $q$ is the heat generated in the rod.

Since the problem is axisymmetric, it is convenient to write this equation in a cylindrical coordinate system.

$$\rho C \frac{\partial u}{\partial t} - \frac{1}{r} \frac{\partial}{\partial r}\left( kr \frac{\partial u}{\partial r} \right) - \frac{1}{r^2} \frac{\partial}{\partial \theta}\left( k \frac{\partial u}{\partial \theta} \right) - \frac{\partial}{\partial z}\left( k \frac{\partial u}{\partial z} \right) = q$$

Here $r$, $\theta$, and $z$ are the three coordinate variables of the cylindrical system. Because the problem is axisymmetric, $\partial u / \partial \theta = 0$.

This is a cylindrical problem, and Partial Differential Equation Toolbox requires equations to be in Cartesian coordinates. To transform the equation to the Cartesian coordinates, first multiply both sides of the equation by $r$:

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

Then define $r$ as $y$ and $z$ as $x$:

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

For this example, use these parameters:

- Density, $\rho = 7800$ kg/m$^3$
- Thermal capacity, $C = 500$ W·s/kg ºC
- Thermal conductivity, $k = 40$ W/mºC
- Radioactive heat source, $q = 20000$ W/m$^3$
- Temperature at the right end, $T\_right = 100$ ºC
- Heat flux at the left end, $HF\_left = 5000$ W/m$^2$
- Surrounding temperature at the outer boundary, $T\_outer = 100$ ºC
- Heat transfer coefficient, $h\_outer = 50$ W/m$^2$·ºC

To solve this problem in the PDE app, follow these steps:

1 Model the rod as a rectangle with corners in (-1.5,0), (1.5,0), (1.5,0.2), and (-1.5,0.2). Here, the *x*-axis represents the *z* direction, and the *y*-axis represents the *r* direction.

   `pderect([-1.5,1.5,0,0.2])`

2 Specify the boundary conditions. To do this, double-click the boundaries to open the **Boundary Condition** dialog box. The PDE app requires boundary conditions in a particular form. Thus, Neumann boundary conditions must be in the form $\vec{n} \cdot (c \nabla u) + q u = g$, and Dirichlet boundary conditions must be in the form $h u = r$. Also, because both sides of the equation are multiplied by $r = y$, multiply coefficients for the boundary conditions by *y*.

   - For the left end, use the Neumann condition $\vec{n} \cdot (k \nabla u) = HF\_left = 5000$. Specify `g = 5000*y` and `q = 0`.
   - For the right end, use the Dirichlet condition $u = T\_right = 100$. Specify `h = 1` and `r = 100`.
   - For the outer boundary, use the Neumann condition $\vec{n} \cdot (k \nabla u) = h\_outer (T\_outer - u) = 50(100 - u)$. Specify `g = 50*y*100` and `q = 50*y`.

- The cylinder axis $r = 0$ is not a boundary in the original problem, but in the 2-D treatment it has become one. Use the artificial Neumann boundary condition for the axis, $\vec{n} \cdot (k\nabla u) = 0$. Specify `g = 0` and `q = 0`.

**3** Specify the coefficients by selecting **PDE** > **PDE Specification** or click the **PDE** button on the toolbar. Heat equation is a parabolic equation, so select the **Parabolic** type of PDE. Because both sides of the equation are multiplied by $r = y$, multiply the coefficients by $y$ and enter the following values: `c = 40*y`, `a = 0`, `f = 20000*y`, and `d = 7800*500*y`.

**4** Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.

**5** Set the initial value to 0, the solution time to 20000 seconds, and compute the solution every 100 seconds. To do this, select **Solve** > **Parameters**. In the **Solve Parameters** dialog box, set time to `0:100:20000`, and $u(t_0)$ to `0`.

**6** Solve the equation by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.

**7** Plot the solution, using the color and contour plot. To do this, select **Plot** > **Parameters** and choose the color and contour plots in the resulting dialog box.
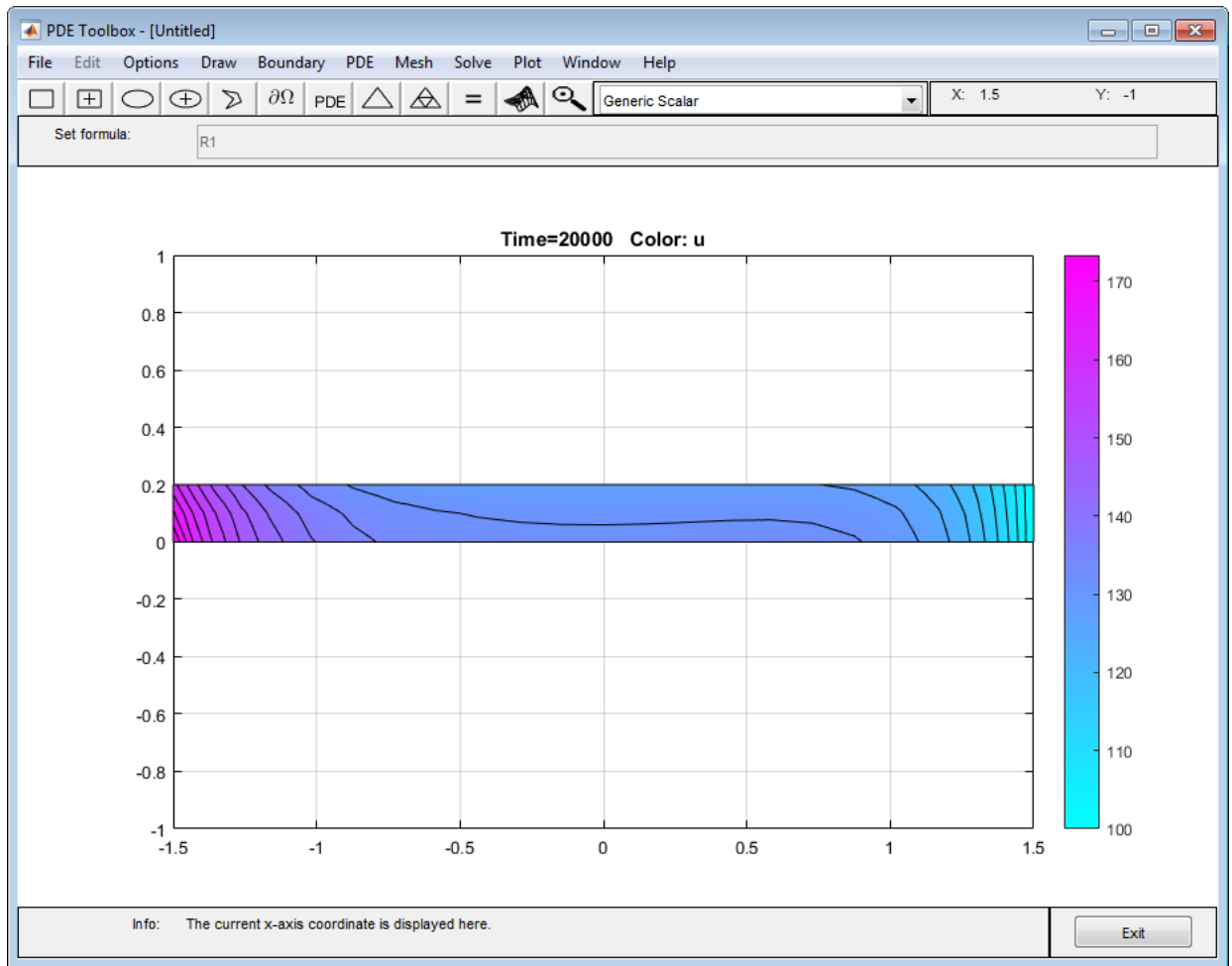
You can explore the solution by varying the parameters of the model and plotting the results. For example, you can:

- Show the solution when $u$ does not depend on time, that is, the steady state solution. To do this, open the PDE Specification dialog box, and change the PDE type to **Elliptic**. The resulting steady state solution is in close agreement with the transient solution at 20000 seconds.

- Show the steady state solution without cooling on the outer boundary: the heat transfer coefficient is zero. To do this, set the Neumann boundary condition at the outer boundary (the top side of the rectangle) to `g = 0` and `q = 0`. The resulting plot shows that the temperature rises to more than 2500 on the left end of the rod.

# 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 $\Delta$ 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))
```
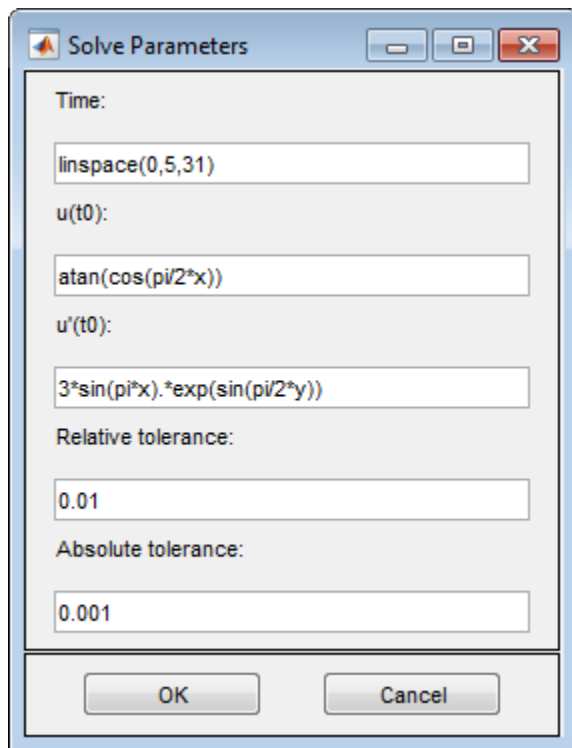


Finally, click the **=** button to compute the solution. The best plot for viewing the waves moving in the $x$ and $y$ directions is an animation of the whole sequence of solutions.

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
```
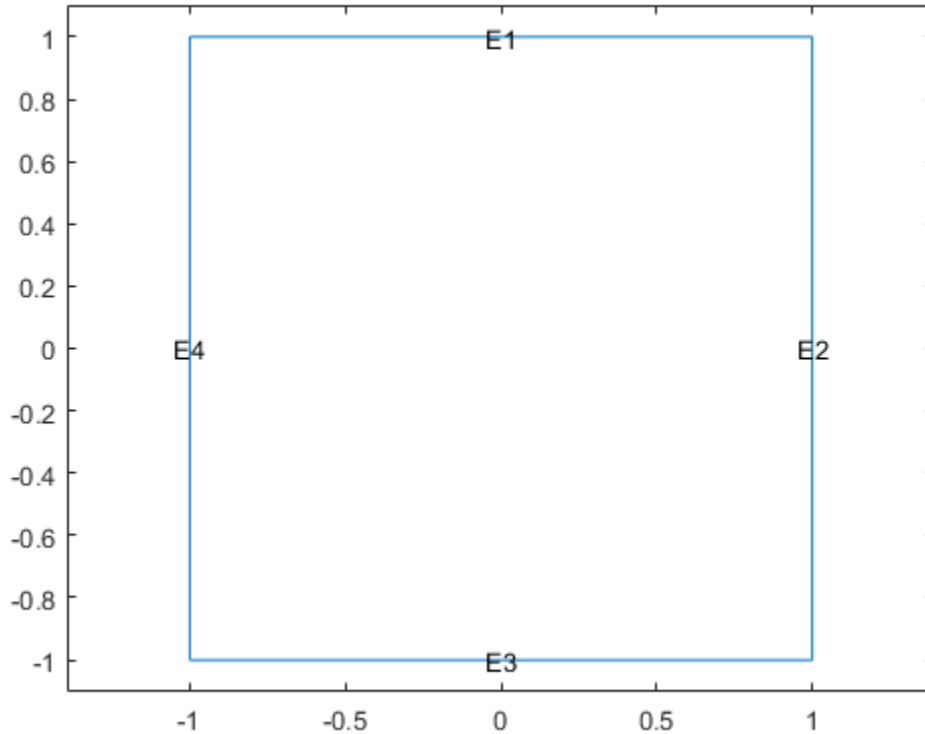
Set the boundary conditions to u = 0 on the left and right boundaries (edges 2 and 4). The default boundary conditions are appropriate for edges 1 and 3, so you do not need to set them.

```
applyBoundaryCondition(model,'dirichlet','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
```
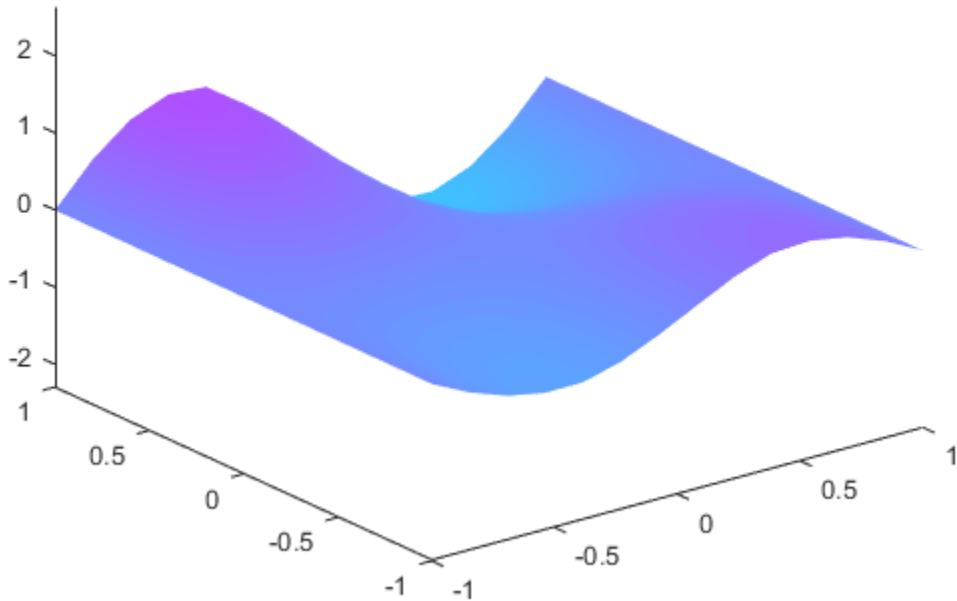
You can review the movie by executing `movie(M,10)`. There is a complete solution of this problem, including animation, in "Wave Equation on a Square Domain". If you have lots of memory, you can try increasing `n`, the number of frames in the movie.

**3-167**

# Eigenvalues and Eigenmodes of the L-Shaped Membrane

This example shows how to calculate eigenvalues and eigenvectors using the programmatic workflow. For the PDE app solution, see "Eigenvalues and Eigenmodes of the L-Shaped Membrane: PDE App".

The eigenvalue problem is $-\Delta u = \lambda u$. This example computes all eigenmodes with eigenvalues smaller than 100.

Create a model and include this geometry. The geometry of the L-shaped membrane is described in the file lshapeg.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
```

Set zero Dirichlet boundary conditions on all edges.

```
applyBoundaryCondition(model,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
```

Specify the coefficients for the problem: d = 1 and c = 1. All other coefficients are 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=  0.03,  New conv eig=  0
Basis= 13,  Time=  0.06,  New conv eig=  0
Basis= 16,  Time=  0.06,  New conv eig=  0
Basis= 19,  Time=  0.06,  New conv eig=  3
Basis= 22,  Time=  0.06,  New conv eig=  3
Basis= 25,  Time=  0.06,  New conv eig=  5
Basis= 28,  Time=  0.06,  New conv eig=  6
Basis= 31,  Time=  0.23,  New conv eig=  8
```

```
               Basis= 34,  Time=    0.23,  New conv eig= 12
               Basis= 37,  Time=    0.23,  New conv eig= 14
               Basis= 40,  Time=    0.34,  New conv eig= 14
               Basis= 43,  Time=    0.47,  New conv eig= 15
               Basis= 46,  Time=    0.47,  New conv eig= 16
               Basis= 49,  Time=    0.72,  New conv eig= 17
               Basis= 52,  Time=    0.81,  New conv eig= 20
End of sweep: Basis= 52,  Time=    0.81,  New conv eig= 20
               Basis= 30,  Time=    1.08,  New conv eig=  0
               Basis= 33,  Time=    1.08,  New conv eig=  0
               Basis= 36,  Time=    1.08,  New conv eig=  0
End of sweep: Basis= 36,  Time=    1.08,  New conv eig=  0
```

There are 19 eigenvalues smaller than 100.

```
length(results.Eigenvalues)
```

```
ans =

    19
```

Plot the first eigenmode and compare it to the MATLAB® `membrane` function. Multiply the PDE solution by -1, so that the plots look similar instead of being inverted.

```
u = results.Eigenvectors;
pdeplot(model,'XYData',-u(:,1),'ZData',-u(:,1));
figure
membrane(1,20,9,9)
```
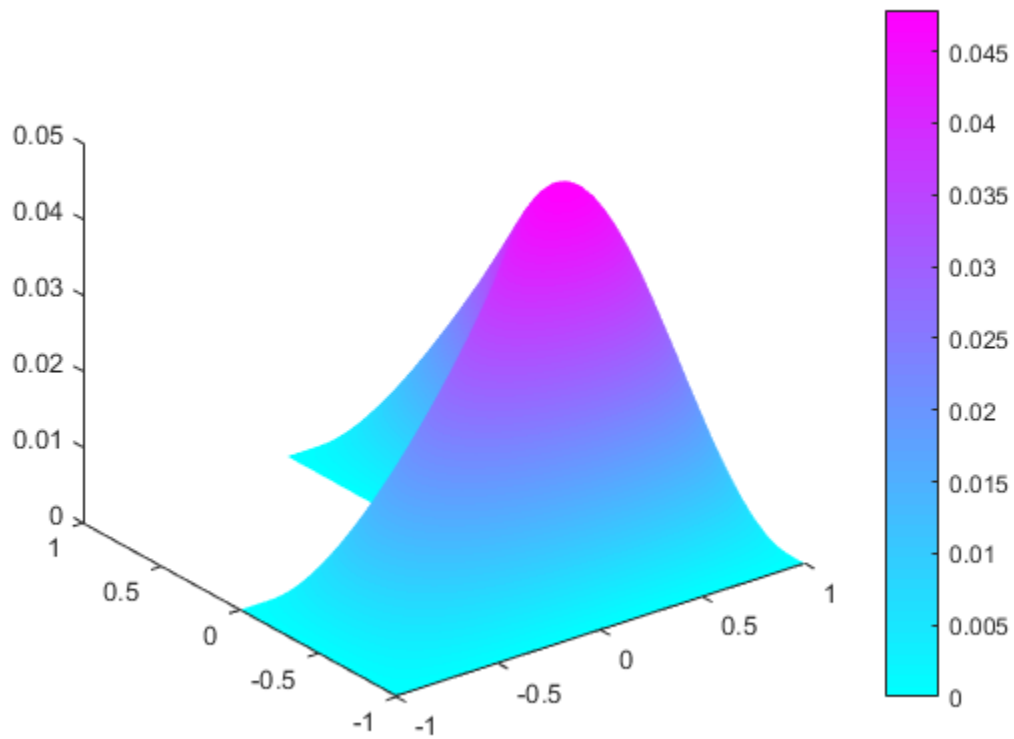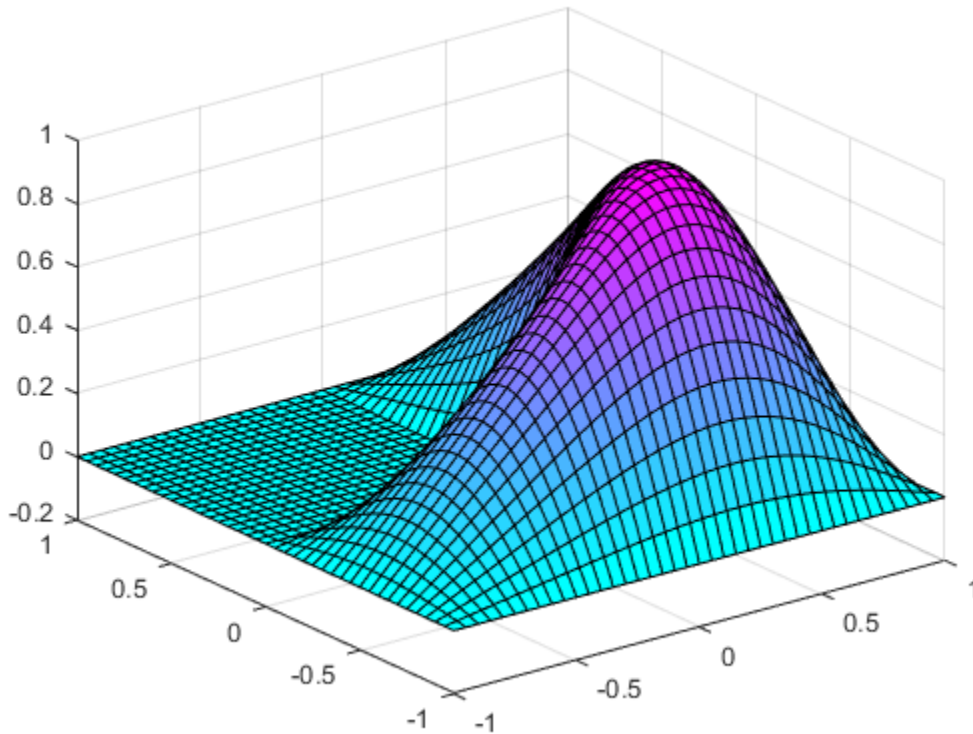
Eigenvectors can be multiplied by any scalar and remain eigenvectors. This explains the difference in scale that you see.

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

```
figure
pdeplot(model,'XYData',u(:,12),'ZData',u(:,12));
figure
membrane(12,20,9,9)
```
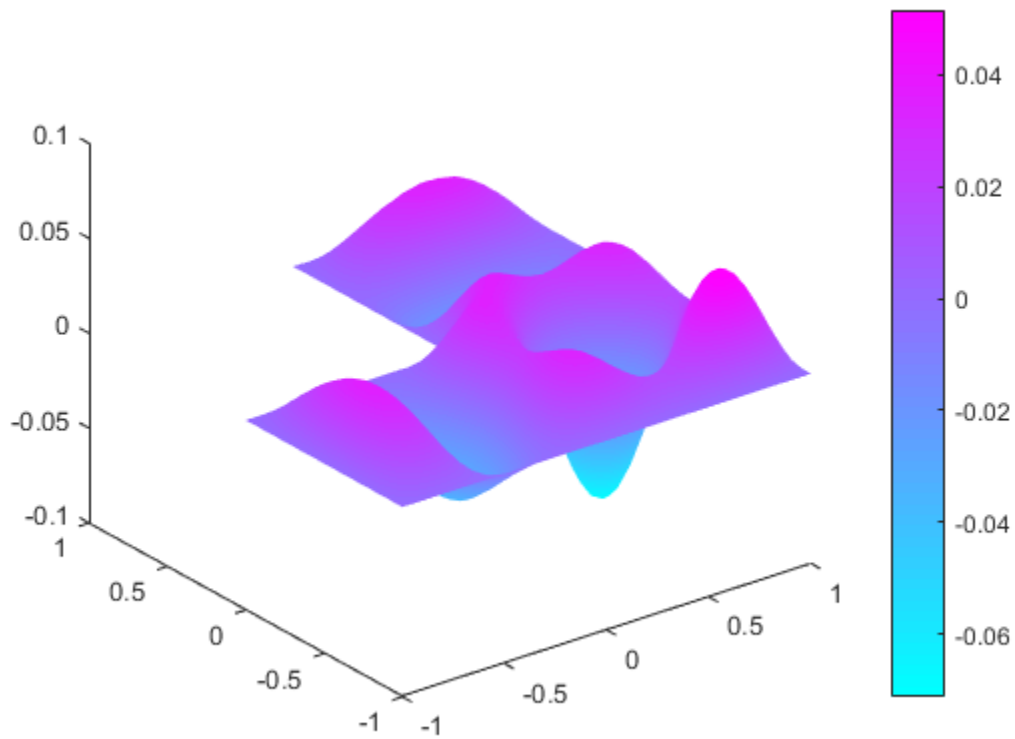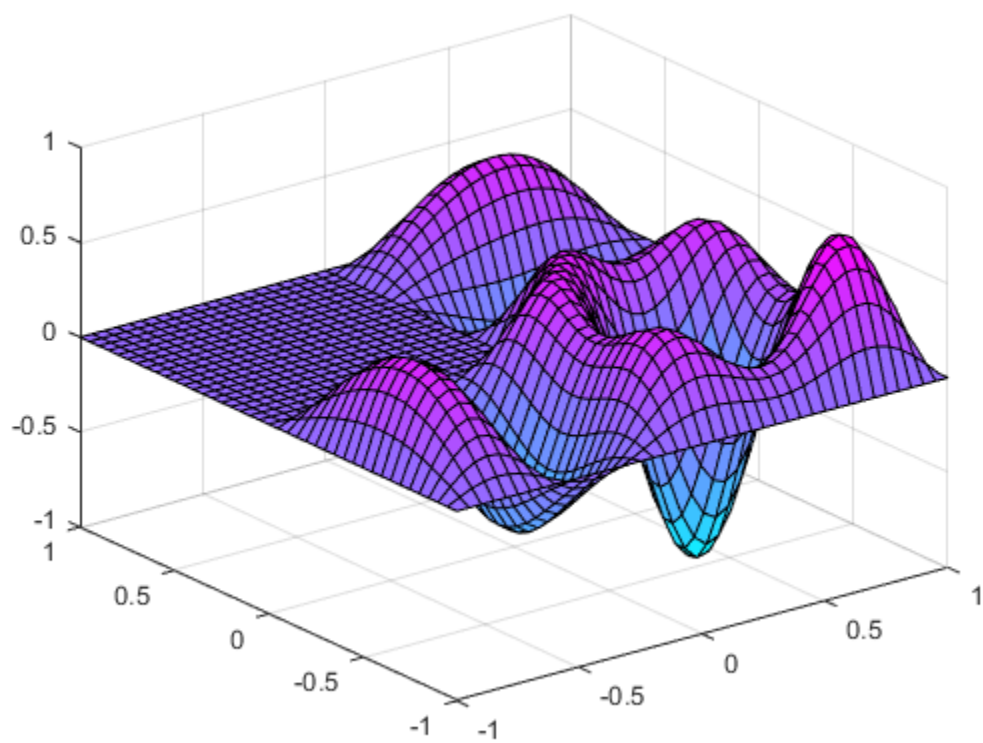
# Eigenvalues and Eigenmodes of the L-Shaped Membrane: PDE App

This example shows how to compute all eigenmodes with eigenvalues smaller than 100 for the eigenmode PDE problem

$-\Delta u = \lambda u$

on the geometry of the L-shaped membrane. The boundary condition is the Dirichlet condition $u = 0$. This example uses the PDE app. For a programmatic workflow, see "Eigenvalues and Eigenmodes of the L-Shaped Membrane" on page 3-168.

To solve this problem in the PDE app, follow these steps:

1  Draw a polygon with the corners (0,0), (–1,0), (–1,–1), (1,–1), (1,1), and (0,1) by using the `pdepoly` function.

    pdepoly([0,-1,-1,1,1,0],[0,0,-1,-1,1,1])

2  Check that the application mode is set to **Generic Scalar**.

3  Use the default Dirichlet boundary condition $u = 0$ for all boundaries. To verify it, switch to boundary mode by selecting **Boundary** > **Boundary Mode**. Use **Edit** > **Select all** to select all boundaries. Select **Boundary** > **Specify Boundary Conditions** and verify that the boundary condition is the Dirichlet condition with `h = 1`, `r = 0`.

4  Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. This is an eigenvalue problem, so select the **Eigenmodes** type of PDE. The general eigenvalue PDE is described by $-\nabla \cdot (c\nabla u) + au = \lambda du$. Thus, for this problem, use the default coefficients `c = 1`, `a = 0`, and `d = 1`.

5  Specify the maximum edge size for the mesh by selecting **Mesh** > **Parameters**. Set the maximum edge size value to 0.05.

6  Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.

7  Specify the eigenvalue range by selecting **Solve** > **Parameters**. In the resulting dialog box, use the default eigenvalue range `[0 100]`.

8  Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar. By default, the app plots the first eigenfunction.
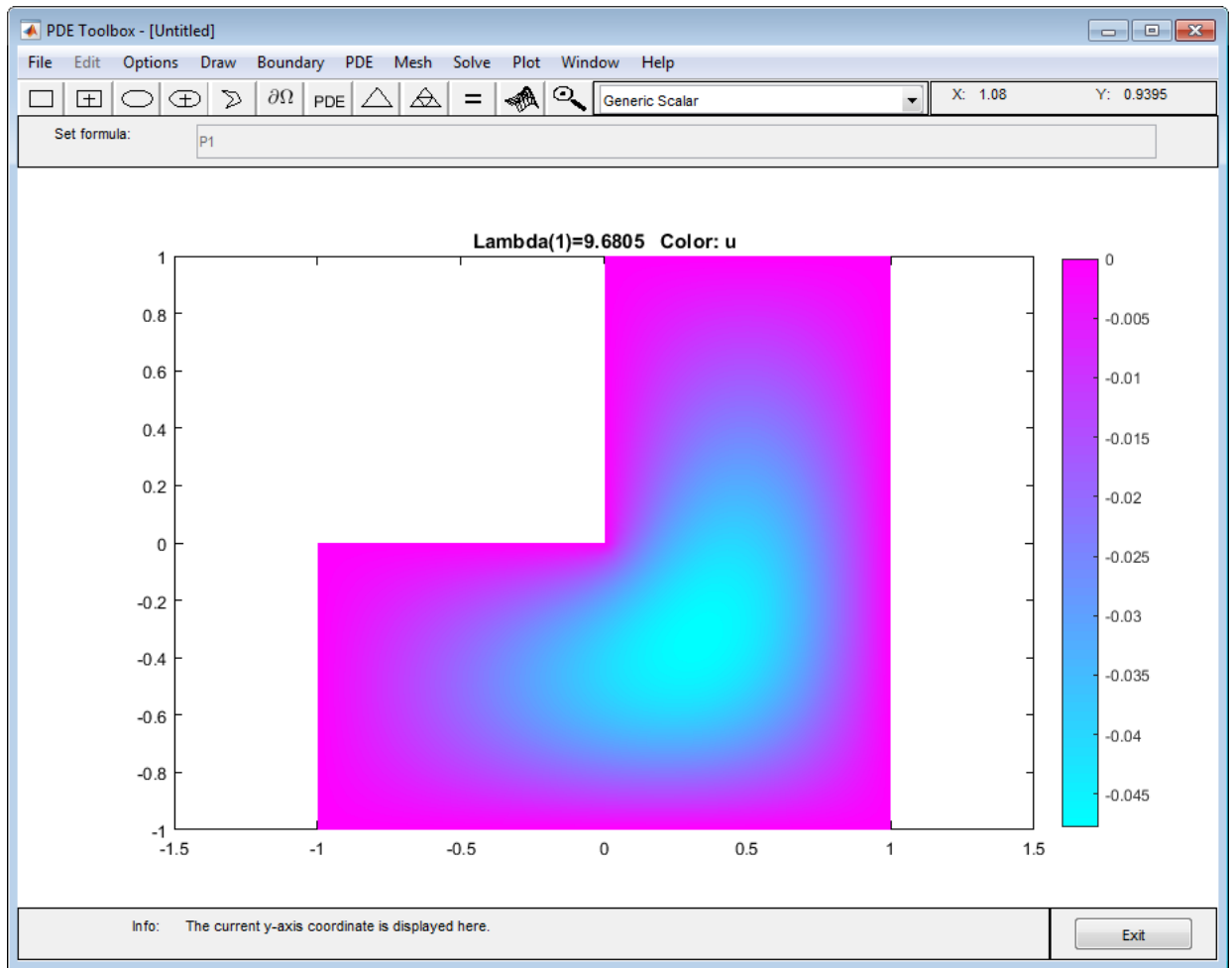
9    Plot other eigenfunctions by selecting **Plot** > **Parameters** and then selecting the corresponding eigenvalue from the drop-down list at the bottom of the dialog box. For example, plot the fifth eigenfunction in the specified range.
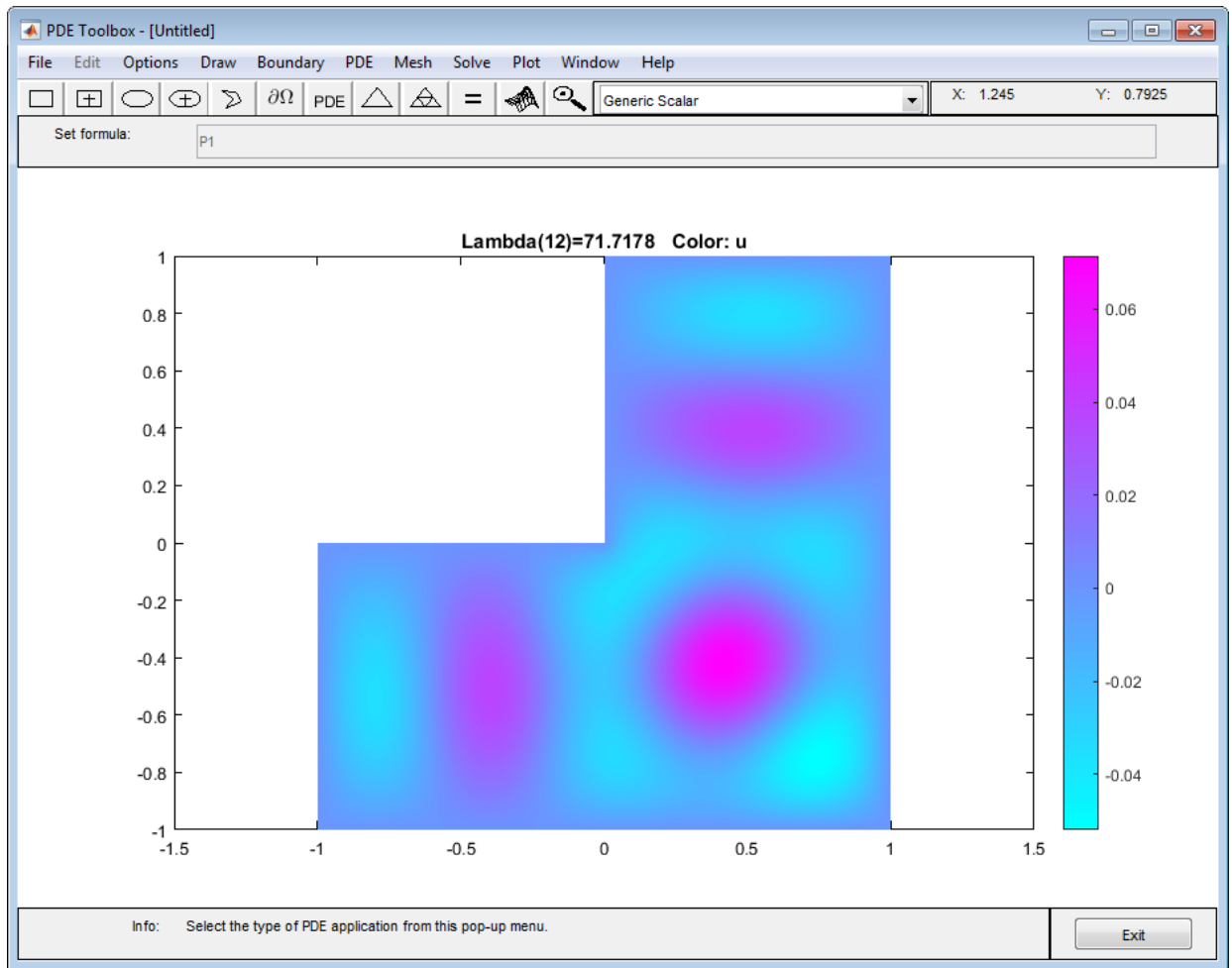
# L-Shaped Membrane with a Rounded Corner

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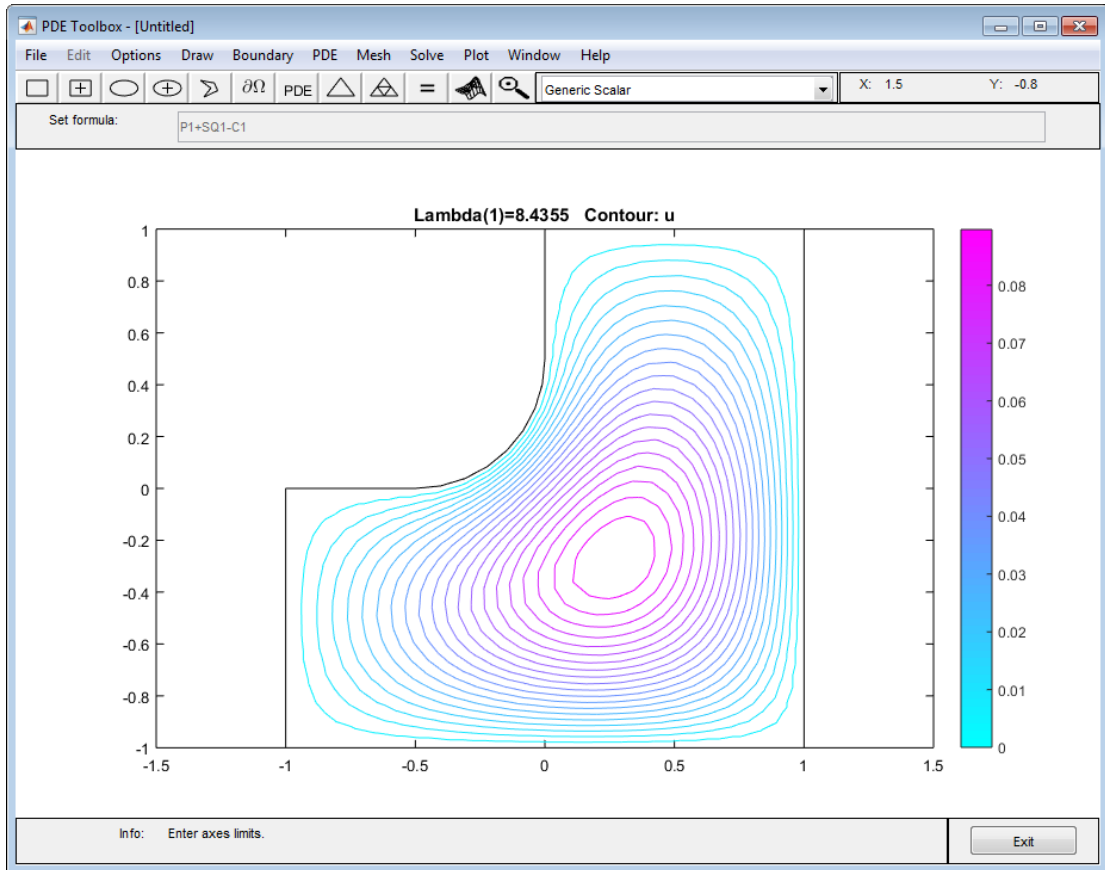
**First Eigenmode for an L-Shaped Membrane with a Rounded Corner**

# Eigenvalues and Eigenmodes of a Square

This example shows how to compute the eigenvalues and eigenmodes of a square domain using the programmatic workflow. For the PDE app solution, see "Eigenvalues and Eigenmodes of a Square: PDE App".

The eigenvalue PDE problem is $-\Delta u = \lambda u$. This example finds the eigenvalues smaller than 10 and the corresponding eigenmodes.

Create a model. Import and plot the geometry. The geometry description file for this problem is called `squareg.m`.

```
model = createpde();
geometryFromEdges(model,@squareg);

pdegplot(model,'EdgeLabels','on')
ylim([-1.5,1.5])
axis equal
```
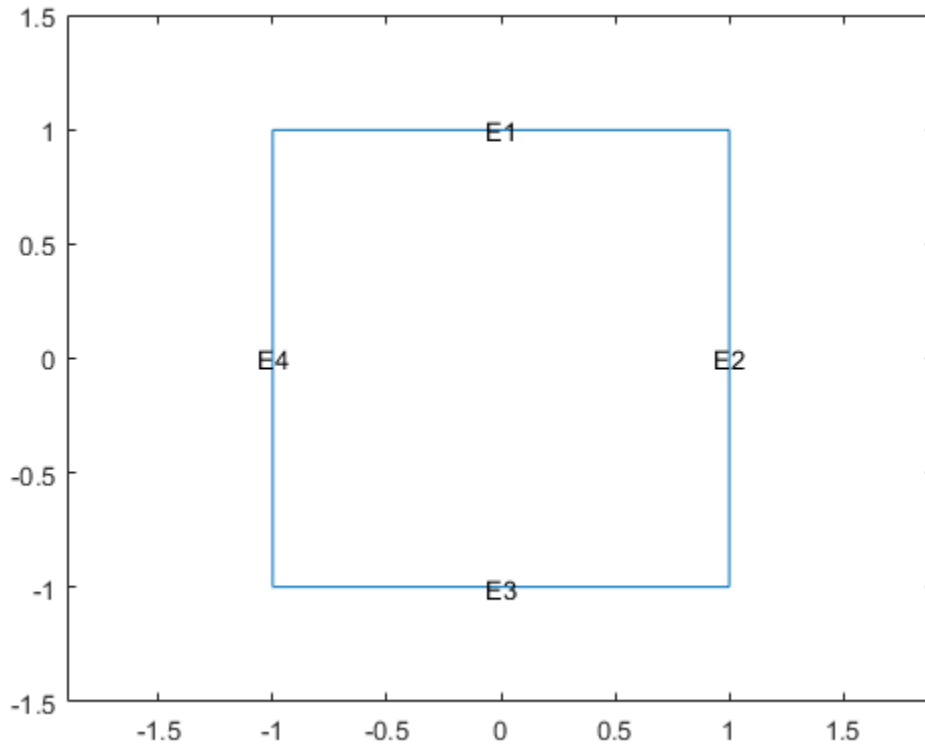
Specify the Dirichlet boundary condition $u = 0$ for the left boundary.

```
applyBoundaryCondition(model,'dirichlet','Edge',4,'u',0);
```

Specify the zero Neumann boundary condition for the upper and lower boundary.

```
applyBoundaryCondition(model,'neumann','Edge',[1,3],'g',0,'q',0);
```

Specify the generalized Neumann condition $\frac{\partial u}{\partial n} - \frac{3}{4}u = 0$ for the right boundary.

```
applyBoundaryCondition(model,'neumann','Edge',2,'g',0,'q',-3/4);
```

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=   0.14,  New conv eig=  0
                Basis= 12,  Time=   0.19,  New conv eig=  1
                Basis= 14,  Time=   0.19,  New conv eig=  1
                Basis= 16,  Time=   0.22,  New conv eig=  1
                Basis= 18,  Time=   0.33,  New conv eig=  2
                Basis= 20,  Time=   0.33,  New conv eig=  2
                Basis= 22,  Time=   0.39,  New conv eig=  4
                Basis= 24,  Time=   0.48,  New conv eig=  7
End of sweep: Basis= 24,  Time=   0.48,  New conv eig=  6
                Basis= 16,  Time=   0.61,  New conv eig=  0
                Basis= 18,  Time=   0.61,  New conv eig=  0
                Basis= 20,  Time=   0.67,  New conv eig=  0
                Basis= 22,  Time=   0.67,  New conv eig=  0
                Basis= 24,  Time=   0.75,  New conv eig=  2
End of sweep: Basis= 24,  Time=   0.75,  New conv eig=  2
                Basis= 18,  Time=   0.84,  New conv eig=  0
End of sweep: Basis= 18,  Time=   0.84,  New conv eig=  0
```

There are six eigenvalues smaller than 10 for this problem.

```
l = results.Eigenvalues
```

```
l =

   -0.4146
    2.0543
    4.8046
    7.2759
    9.4697
```

Plot the first and last eigenfunctions in the specified range.

```
u = results.Eigenvectors;
pdeplot(model,'XYData',u(:,1));
```
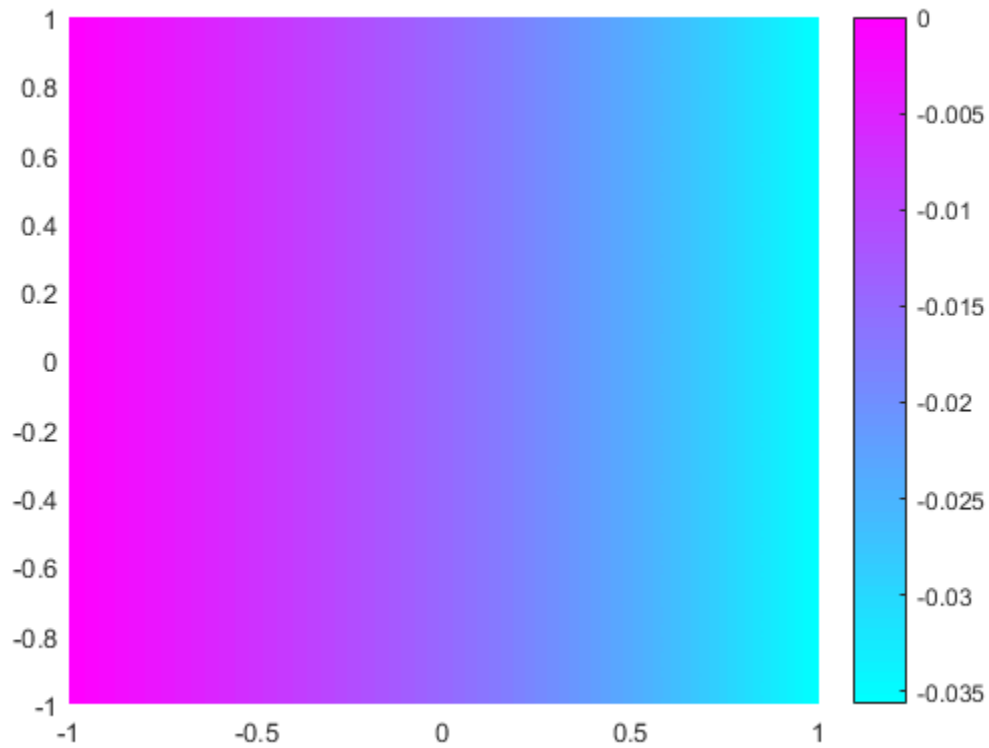
```
pdeplot(model,'XYData',u(:,length(l)));
```
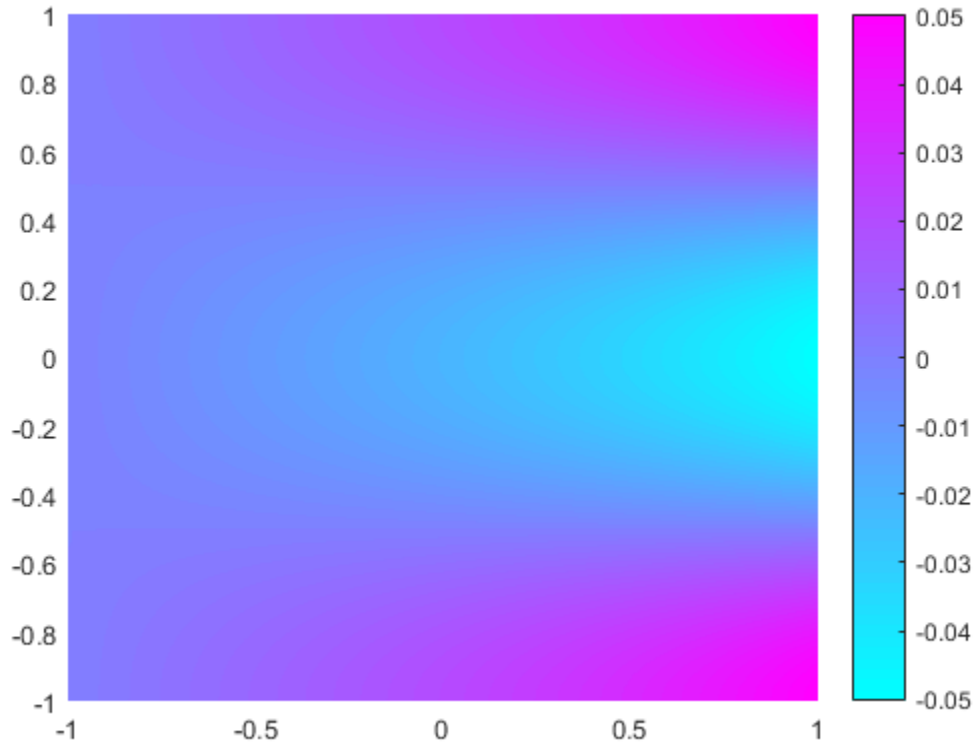
This problem is *separable*, meaning

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

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

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

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(2) - l(1) - pi^2/4
```

```
ans =

    0.0015
```

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 $\pi^2$:

```
l(5) - l(1) - pi^2
```

```
ans =

    0.0147
```

You can explore higher modes by increasing the search range to include eigenvalues greater than 10.

# Eigenvalues and Eigenmodes of a Square: PDE App

This example shows how to compute the eigenvalues and eigenmodes of a square with the corners (-1,-1), (-1,1), (1,1), and (1,-1). This example uses the PDE app. For programmatic workflow, see "Eigenvalues and Eigenmodes of a Square" on page 3-179.

The eigenvalue PDE problem is $-\Delta u = \lambda u$. Find the eigenvalues smaller than 10 and the corresponding eigenmodes.

To solve this problem in the PDE app, follow these steps:

1   Draw a square with the corners (-1,-1), (-1,1), (1,1), and (1,-1) by using the `pderect` function.

    `pderect([-1 1 -1 1])`

2   Check that the application mode is set to **Generic Scalar**.

3   Specify the boundary conditions. To do this, switch to the boundary mode by selecting **Boundary > Boundary Mode**. Double-click the boundary to specify the boundary condition.

    · Specify the Dirichlet condition $u = 0$ for the left boundary. To do this, specify `h = 1, r = 0`.

    · Specify the Neumann condition $\dfrac{\partial u}{\partial n} = 0$ for the upper and lower boundary. To do this, specify `g = 0, q = 0`.

    · Specify the generalized Neumann condition $\dfrac{\partial u}{\partial n} - \dfrac{3}{4}u = 0$ for the right boundary. To do this, specify `g = 0, q = -3/4`.

4   Specify the coefficients by selecting **PDE > PDE Specification** or clicking the **PDE** button on the toolbar. This is a eigenvalue problem, so select the **Eigenmodes** type of PDE. The general eigenvalue PDE is described by $-\nabla \cdot (c\nabla u) + au = \lambda du$. Thus, for this problem, the coefficients are `c = 1, a = 0`, and `d = 1`.

5   Specify the maximum edge size for the mesh by selecting **Mesh > Parameters**. Set the maximum edge size value to 0.05.

6   Initialize the mesh by selecting **Mesh > Initialize Mesh**.

**7** Specify the eigenvalue range by selecting **Solve** > **Parameters**. In the resulting dialog box, enter the eigenvalue range as the MATLAB vector `[-Inf 10]`.

**8** Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar. By default, the app plots the first eigenfunction.
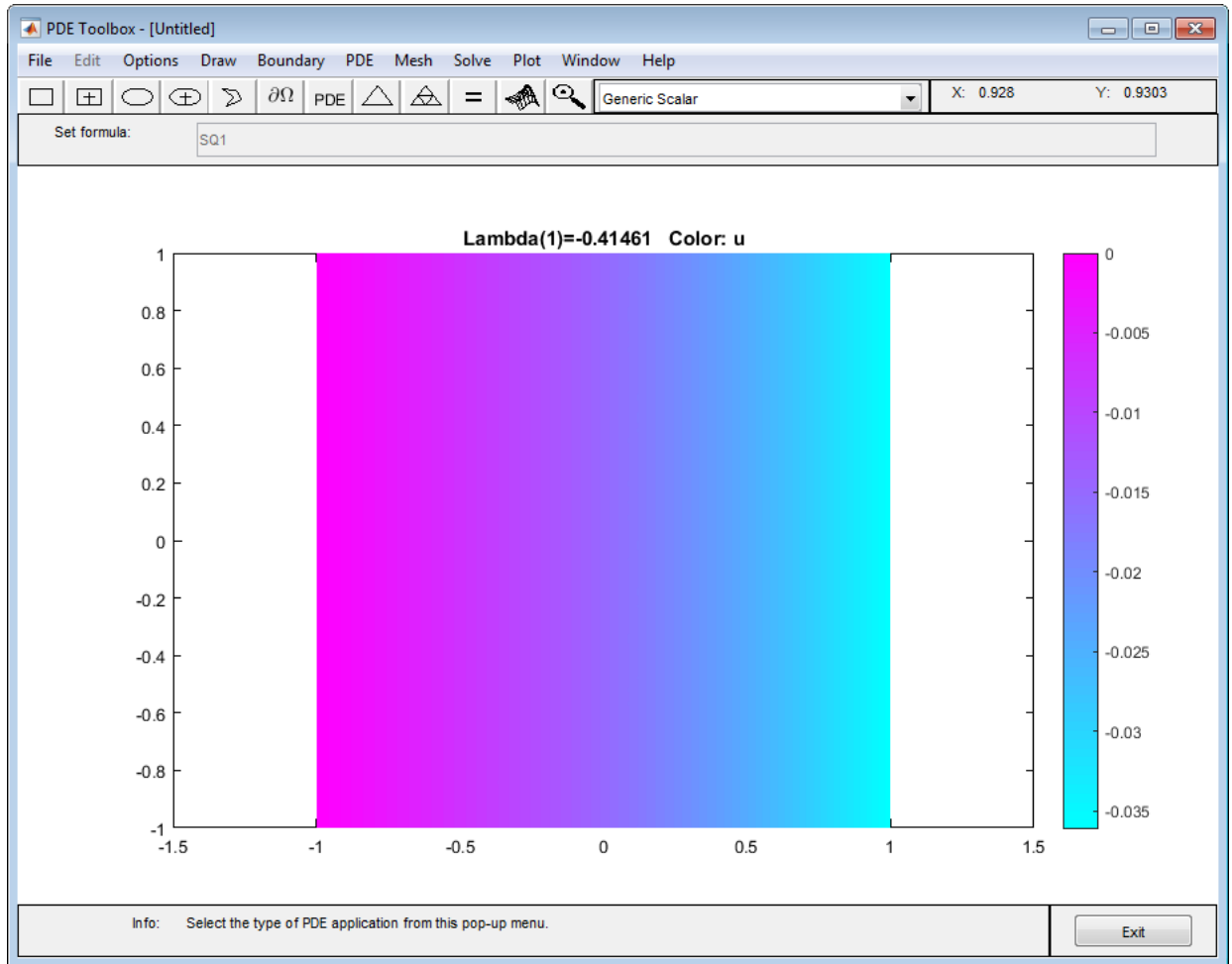


**9** Plot other eigenfunctions by selecting **Plot** > **Parameters** and then selecting the corresponding eigenvalue from the drop-down list at the bottom of the dialog box. For example, plot the last eigenfunction in the specified range.
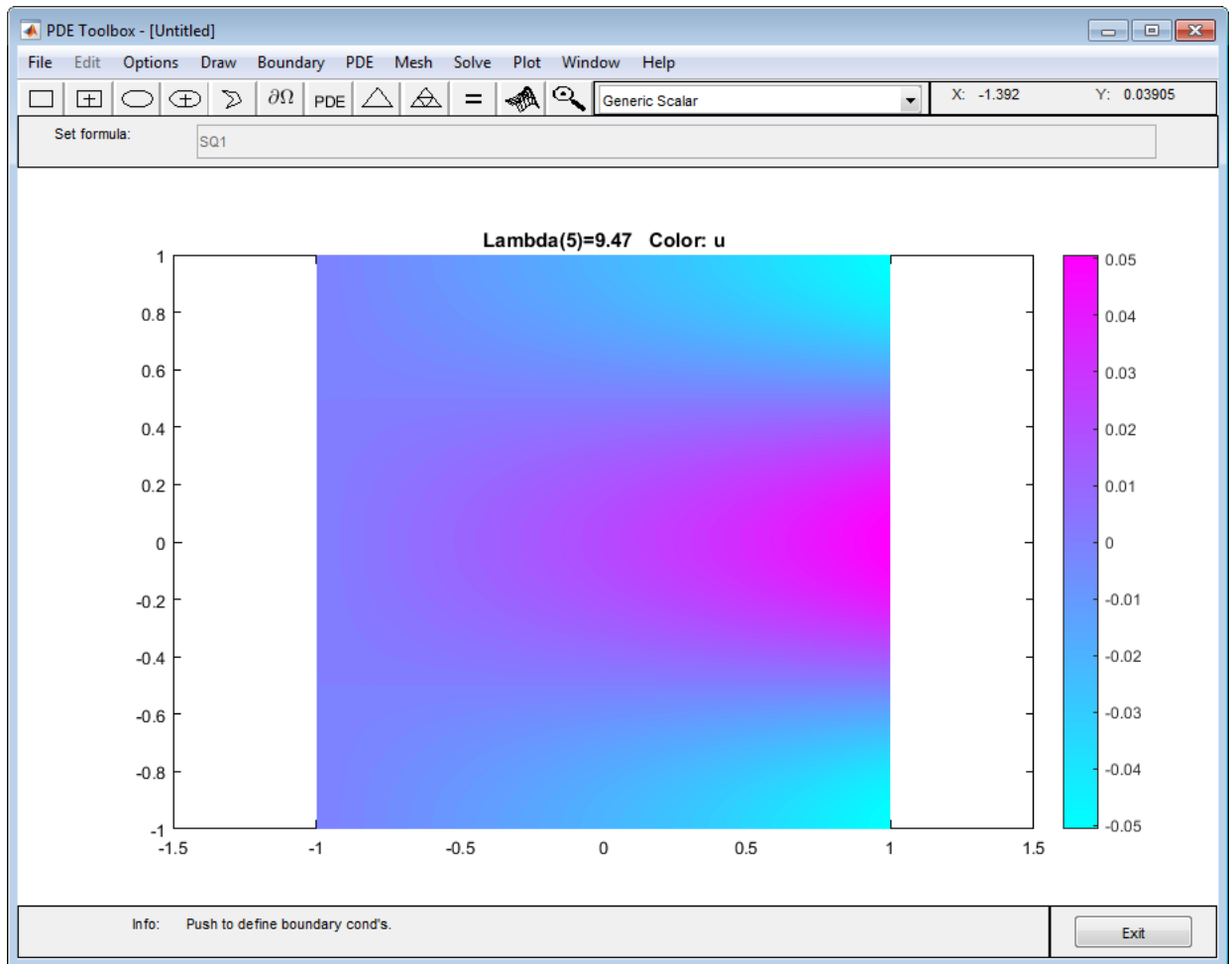
**10** Export the eigenfunctions and eigenvalues to the MATLAB workspace by using the **Solve** > **Export Solution**.

# 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 Model

```
numberOfPDE = 1;
model = createpde(numberOfPDE);
```

### Include Geometry

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(model,g);
```

Plot the geometry and display the edge labels for use in the boundary condition definition.

```
figure;
pdegplot(model,'EdgeLabels','on');
axis equal
title 'Geometry With Edge Labels Displayed';
```

Geometry With Edge Labels Displayed
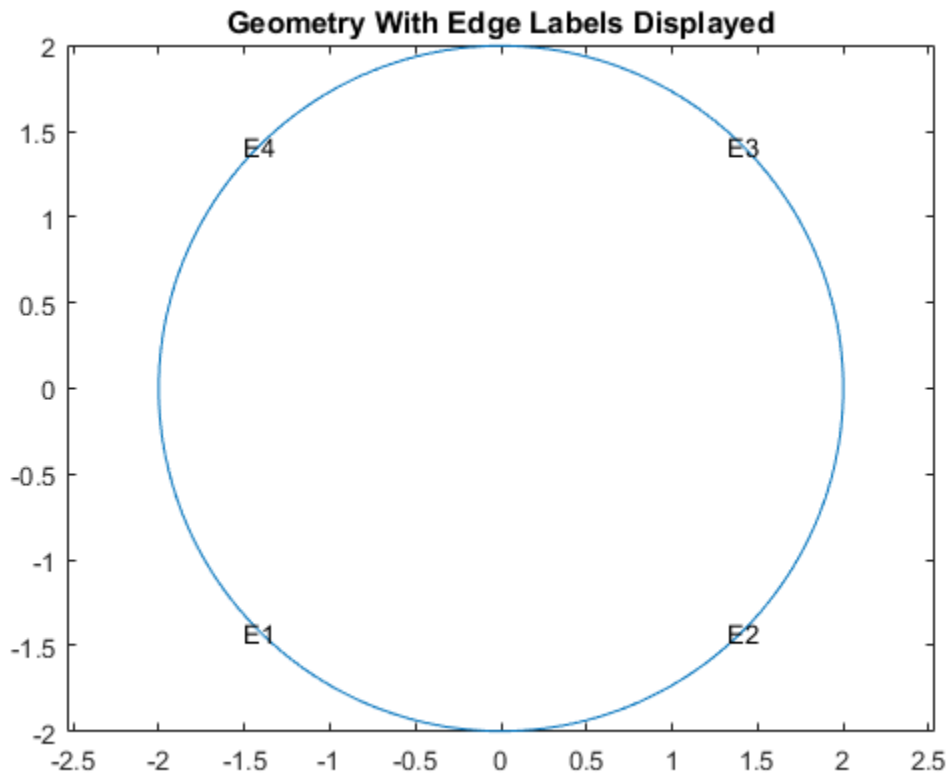
### Define PDE Coefficients and Boundary Conditions

```
c = 1e2;
a = 0;
f = 0;
d = 10;
specifyCoefficients(model,'m',0,'d',d,'c',c,'a',a,'f',f);
```

Solution is zero at all four outer edges of the circle.

```
bOuter = applyBoundaryCondition(model,'dirichlet','Edge',(1:4),'u',0);
```

### Generate Mesh

```
generateMesh(model,'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(model,'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(model,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=   0.06,  New conv eig=  0
End of sweep: Basis= 17,  Time=   0.06,  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));
```

Maximum difference in eigenvalues from solvepdeeig and eigs: 1.563194e-13

As can be seen, both functions calculate the same eigenvalues. For any eigenvalue, the eigenvector can be multiplied by an arbitrary scalar. 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(model,'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(model,'XYData',eigenvectorsPde(:,end),'Contour','on');
title(sprintf('solvepdeeig eigenvector, eigenvalue: %12.4e',eigenvaluesPde(end)));
xlabel('x');
ylabel('y');
```
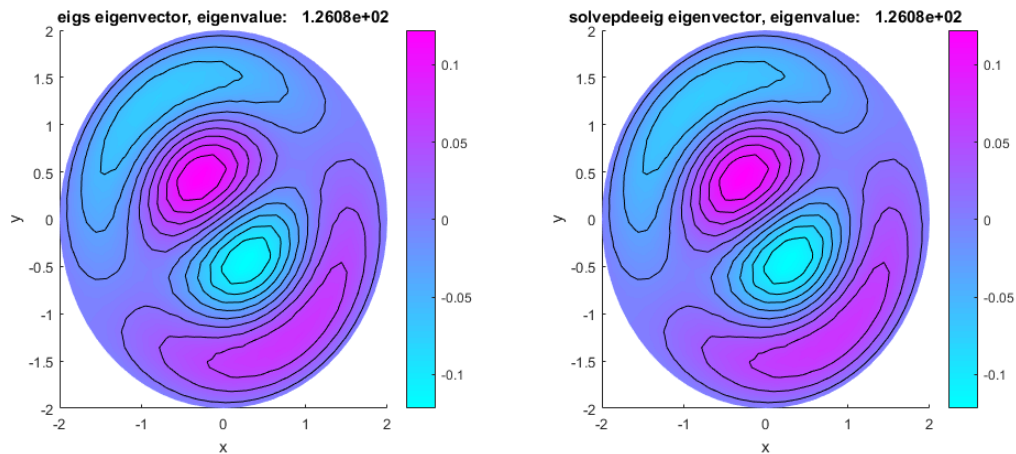
# Solve PDEs Programmatically

---

**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 2-5. The workflow described on this page is an alternative to the legacy workflow described in "Solve Problems Using Legacy PDEModel Objects" on page 2-3.

---

## 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.
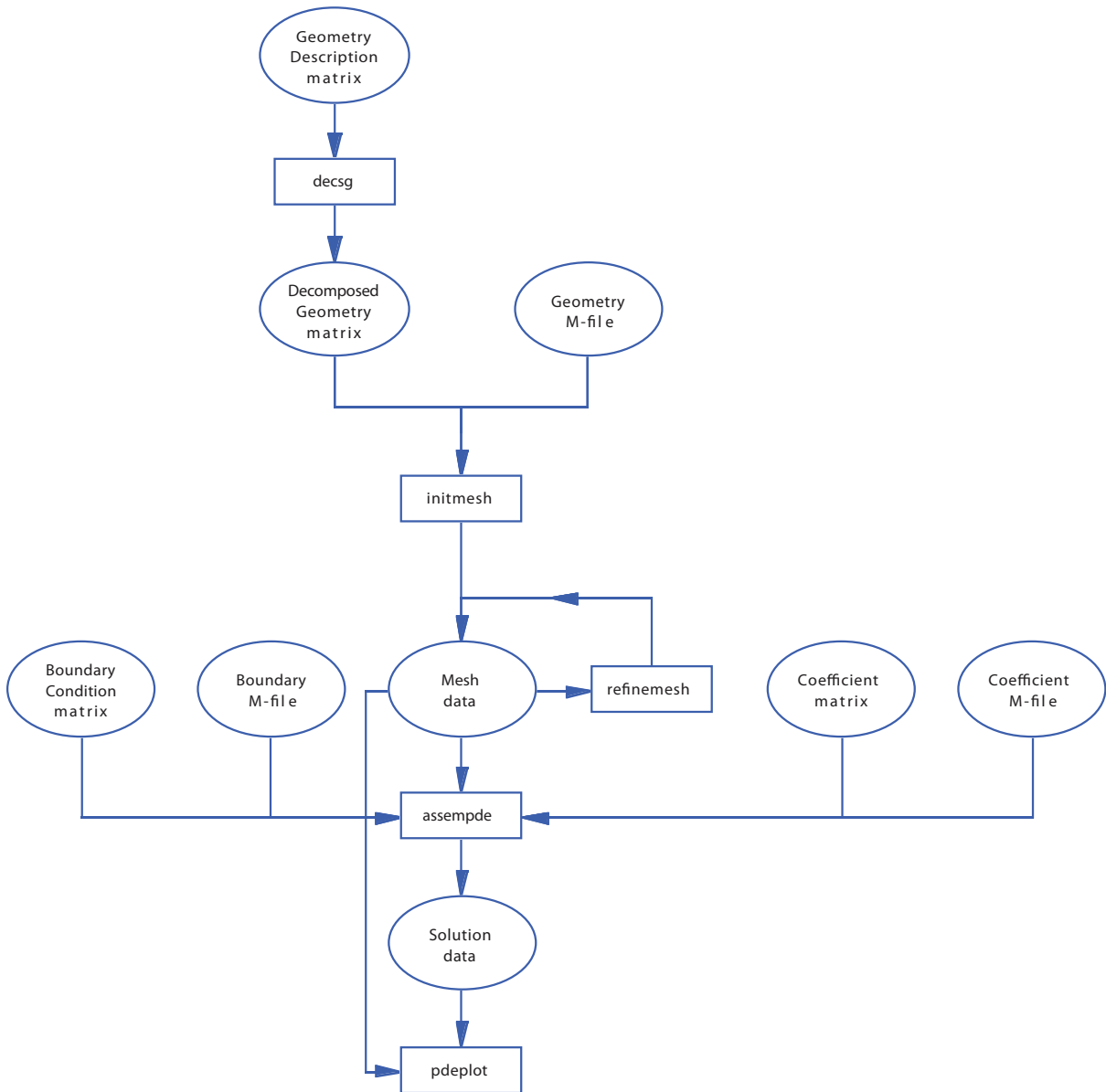
### Constructive Solid Geometry Model

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 2-196.

### 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 $\Omega$

with Dirichlet boundary conditions, where $\Omega$ is a rectangle.

For the fast solution algorithms to work, the mesh on the rectangle must be a *regular mesh*. In this context it means that the first side of the rectangle is divided into $N_1$ segments of length $h_1$, the second into $N_2$ segments of length $h_2$, and $(N_1 + 1)$ by $(N_2 + 1)$ points are introduced on the regular grid thus defined. The triangles are all congruent with sides $h_1$, $h_2$ and a right angle in between.

The Dirichlet boundary conditions are eliminated in the usual way, and the resulting problem for the interior nodes is $Kv = F$. If the interior nodes are numbered from left to right, and then from bottom to top, the $K$ matrix is block tridiagonal. The $N_2 - 1$ diagonal blocks, here called $T$, are themselves tridiagonal $(N_1 - 1)$ by $(N_1 - 1)$ matrices, with $2(h_1/h_2 + h_2/h_1)$ on the diagonal and $-h_2/h_1$ on the subdiagonals. The subdiagonal blocks, here called $I$, are $-h_1/h_2$ times the unit $N_1 - 1$ matrix.

The key to the solution of the problem $Kv = F$ is that the problem $Tw = f$ is possible to solve using the *discrete sine transform*. Let $S$ be the $(N_1 - 1)$ by $(N_1 - 1)$ matrix with $Sij= \sin(\pi ij/N_1)$. Then $S^{-1}TS = \Lambda$, where $\Lambda$ 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 $-\nabla u = 1$ 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,'dirichlet','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')
```
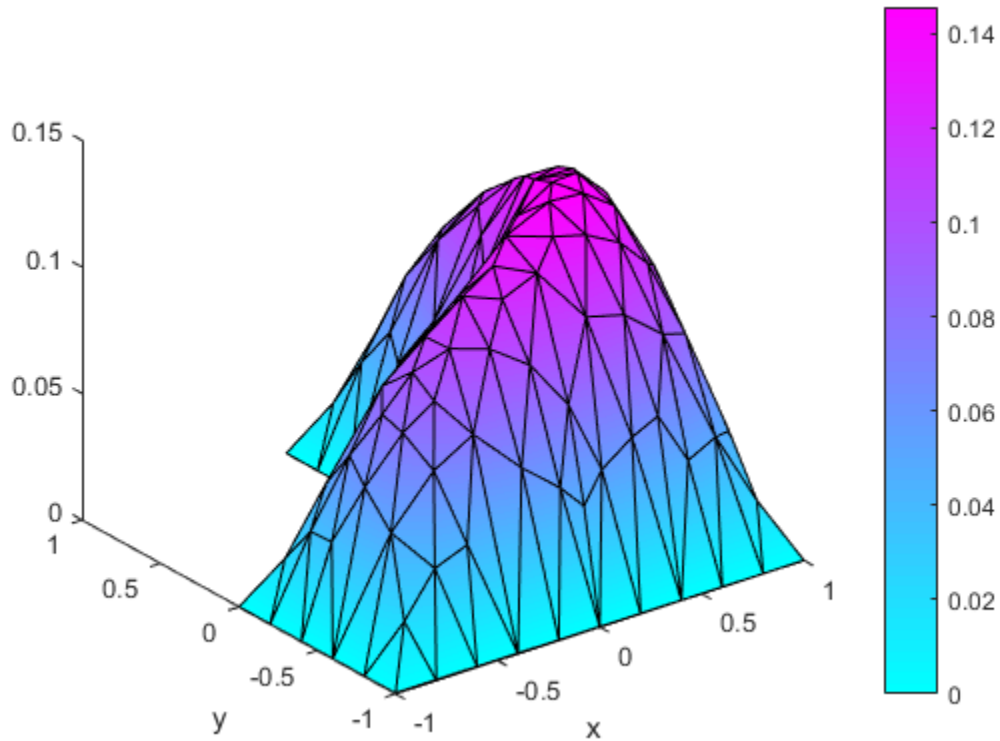
To get a smoother solution surface, specify the maximum size of the mesh triangles by using the `Hmax` argument. Then solve the PDE problem using this new mesh, and plot the solution again.

```
generateMesh(model,'Hmax',0.05);
results = solvepde(model);
u = results.NodalSolution;

pdeplot(model,'XYData',u,'ZData',u,'Mesh','on')
xlabel('x')
ylabel('y')
```
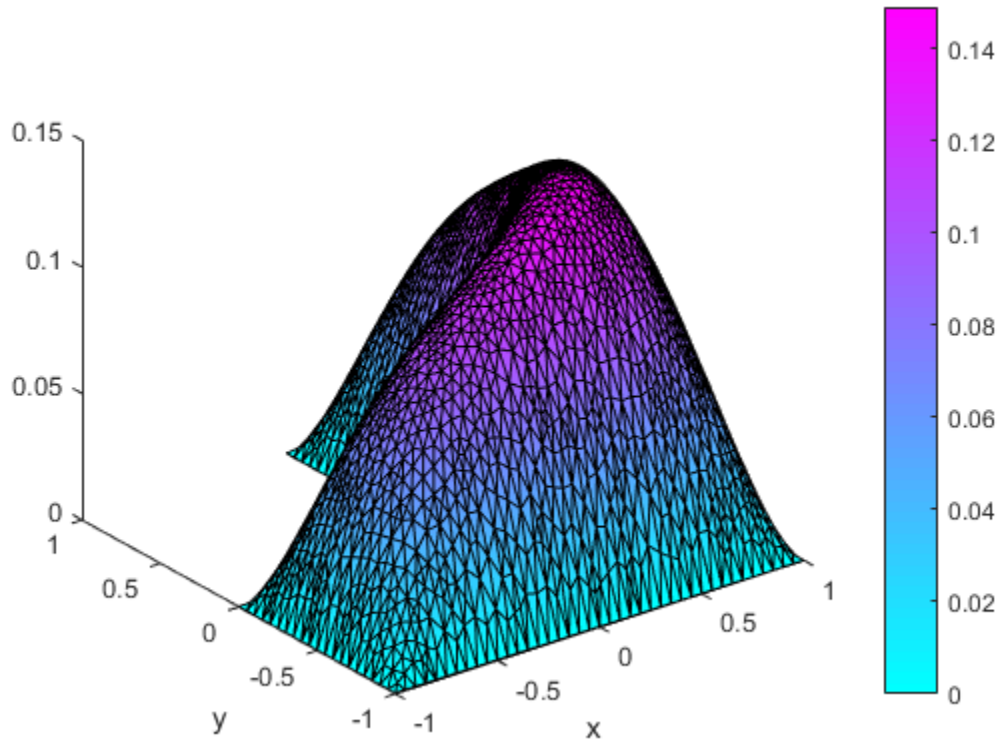
## Interpolate and Plot Solutions and Gradients

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,'dirichlet','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')
```
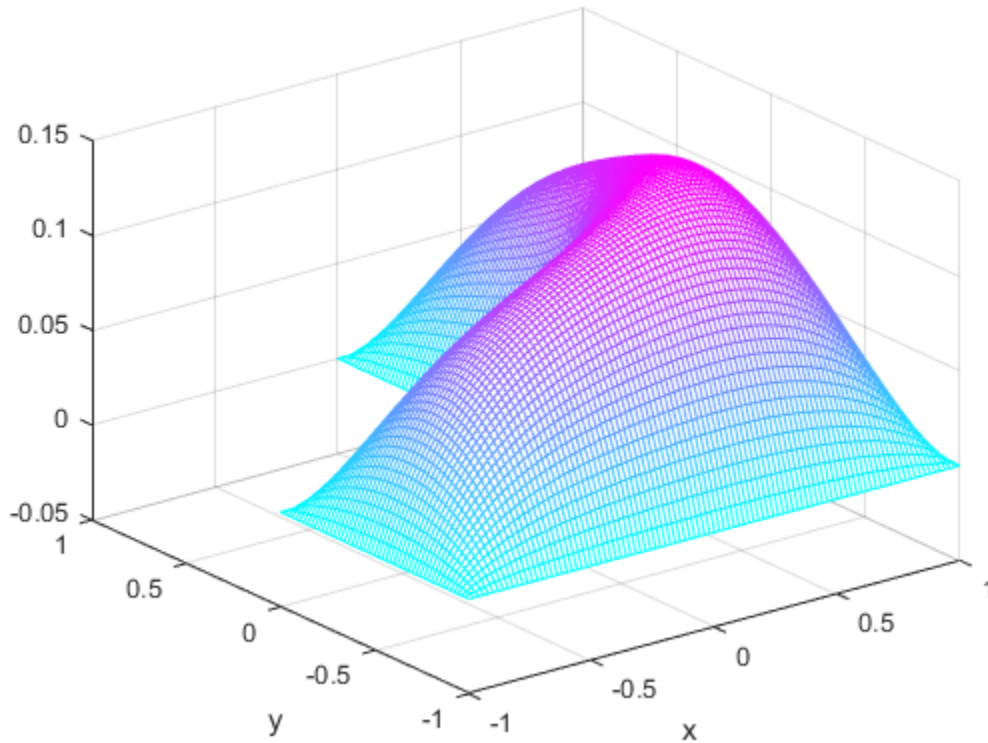
Interpolate gradients of the solution to the grid from -1 to 1 in each direction. Plot the result using `quiver`.

```
[gradx,grady] = evaluateGradient(results,querypoints);
figure
quiver(X(:),Y(:),gradx,grady)
xlabel('x')
ylabel('y')
```
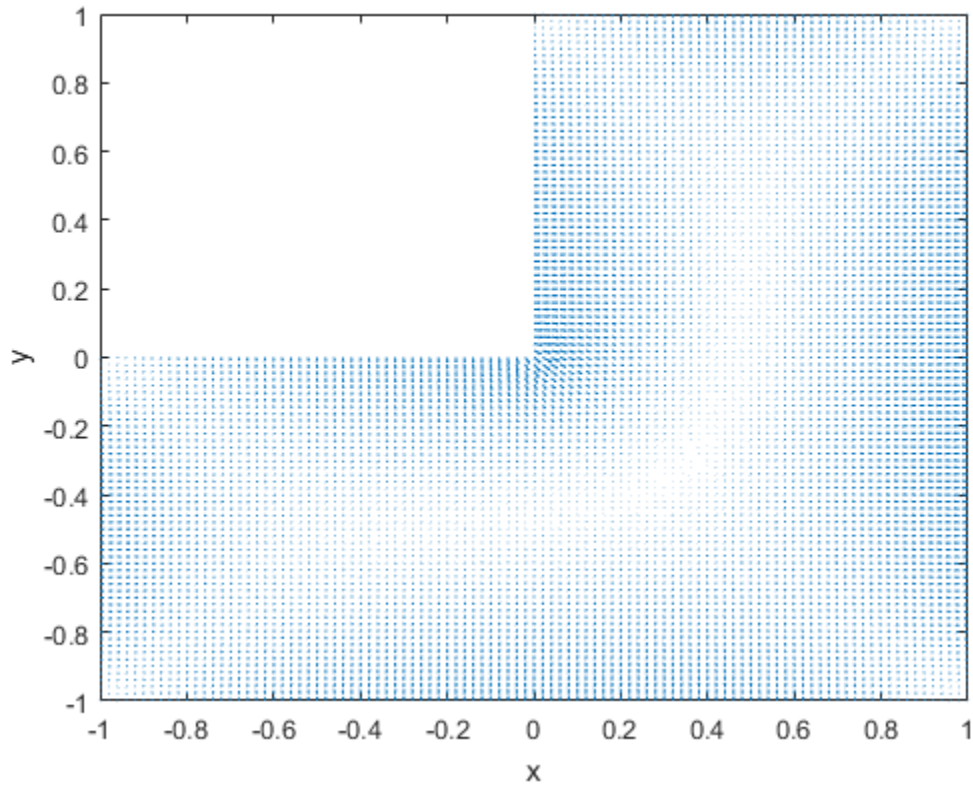
Zoom in to see more details. For example, restrict the range to [-0.2,0.2] in each direction.

```
axis([-0.2 0.2 -0.2 0.2])
```
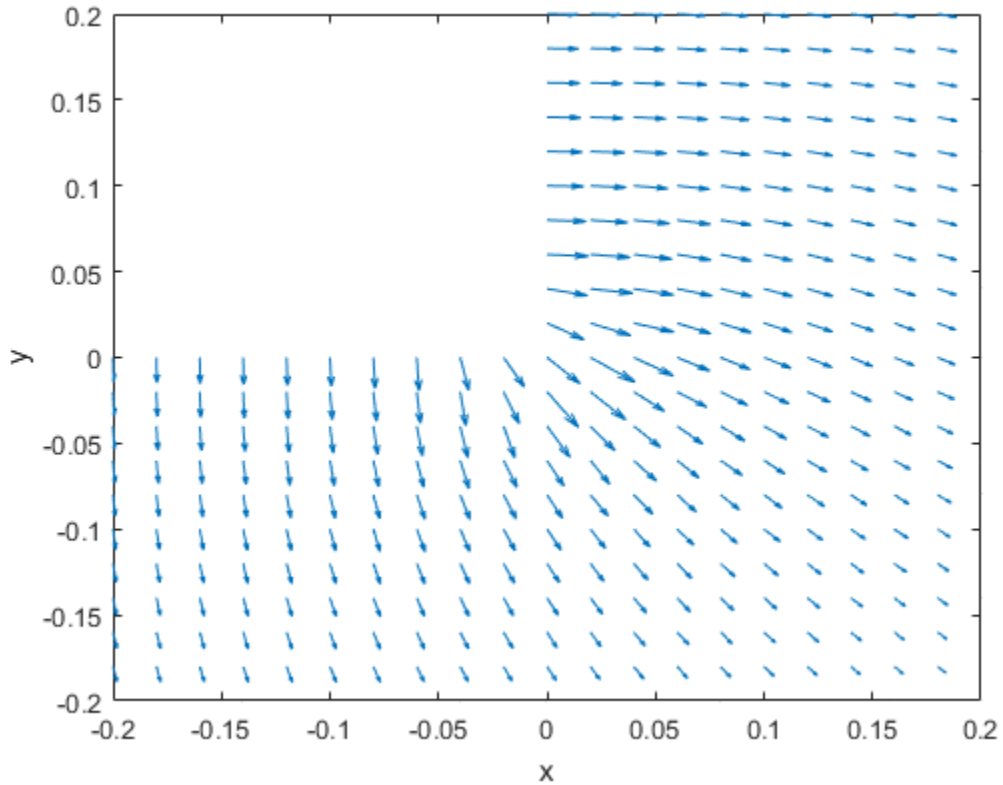
Plot the solution and the gradients on the same range.

```
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])
```
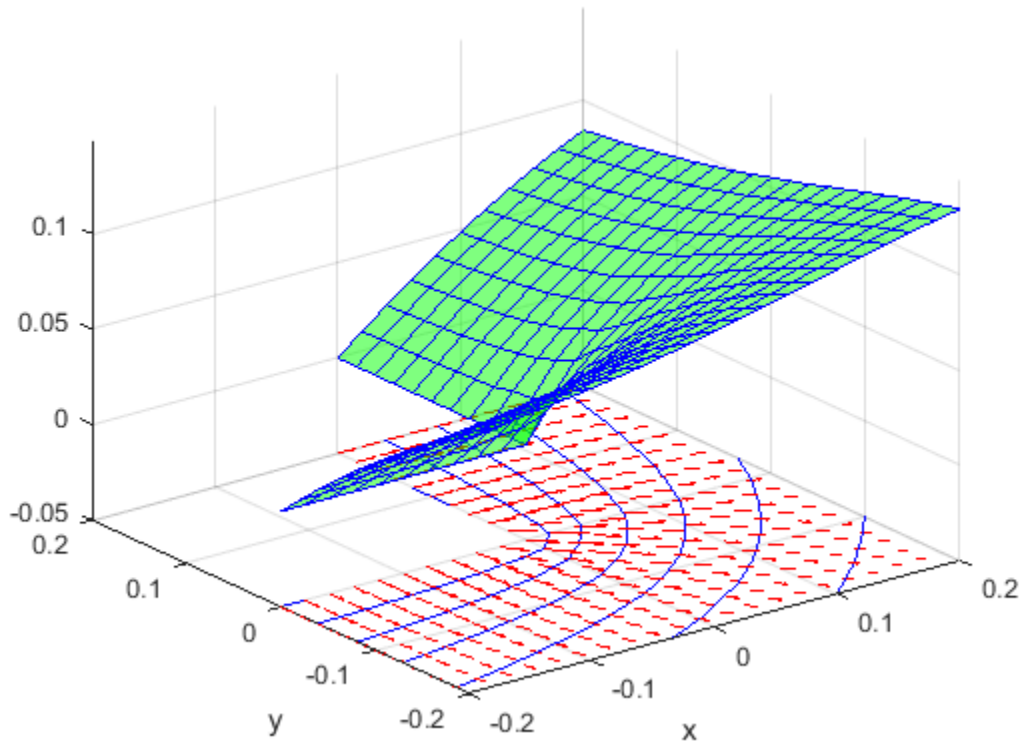


Slice of the solution plot along the line `x = y`.

```
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
```
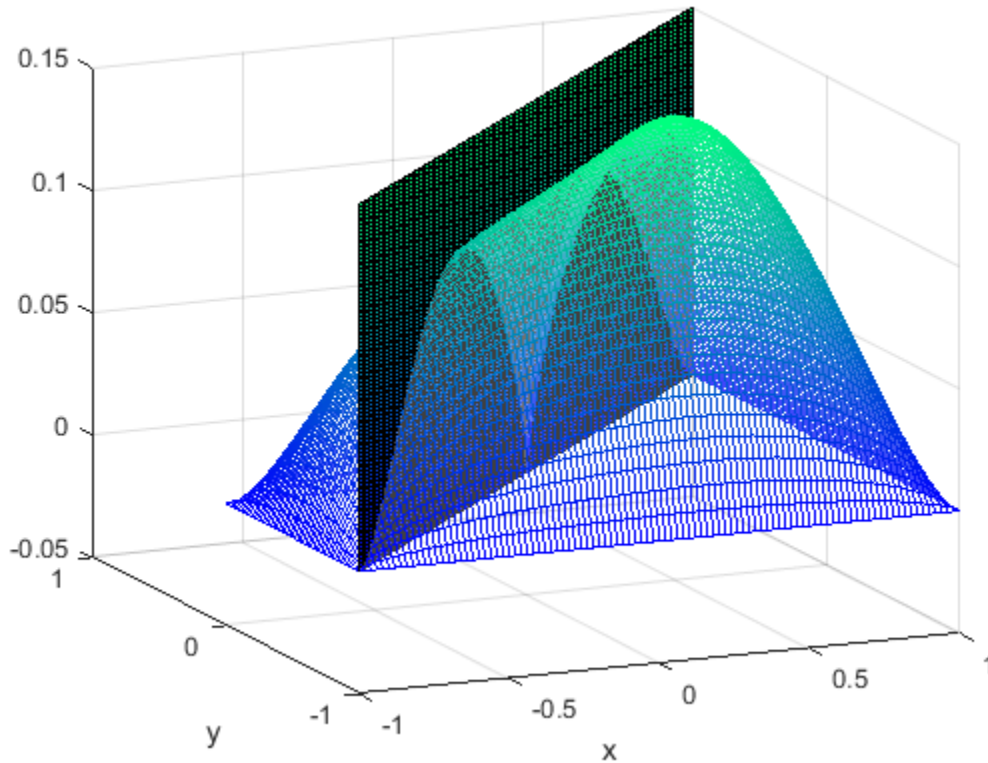


Plot the interpolated solution along the line.

```
figure
xq = v;
yq = v;
uintrp = interpolateSolution(results,xq,yq);

plot3(xq,yq,uintrp)
grid on
```

**3-207**

```
xlabel('x')
ylabel('y')
```
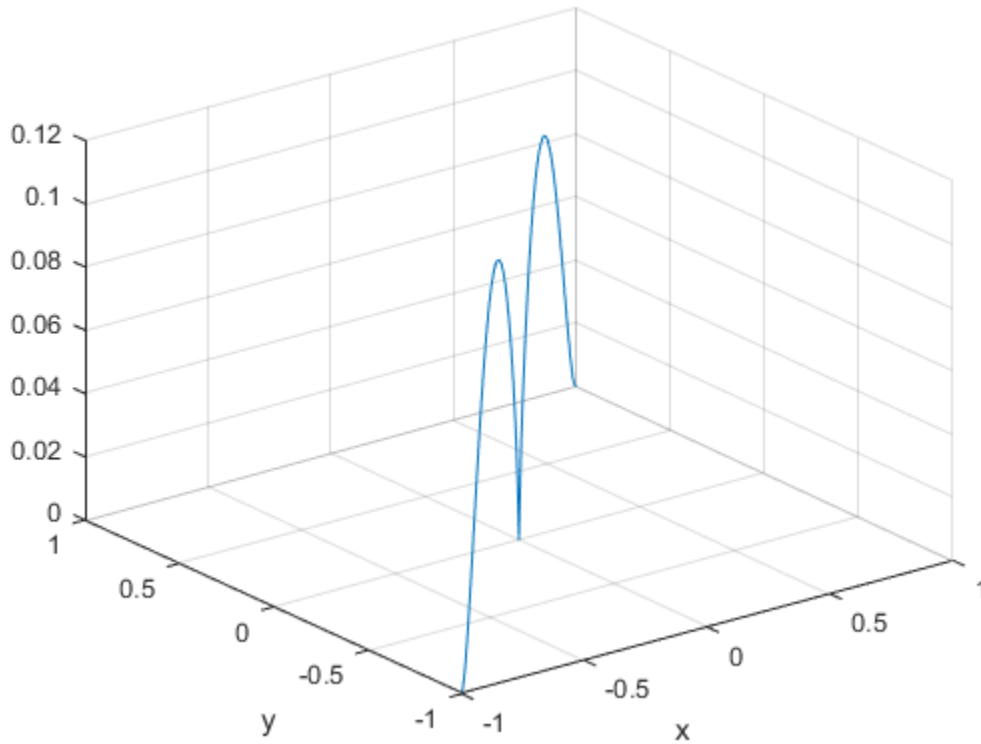


Interpolate gradients of the solution along the same line and add them to the solution plot.

```
[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])
```
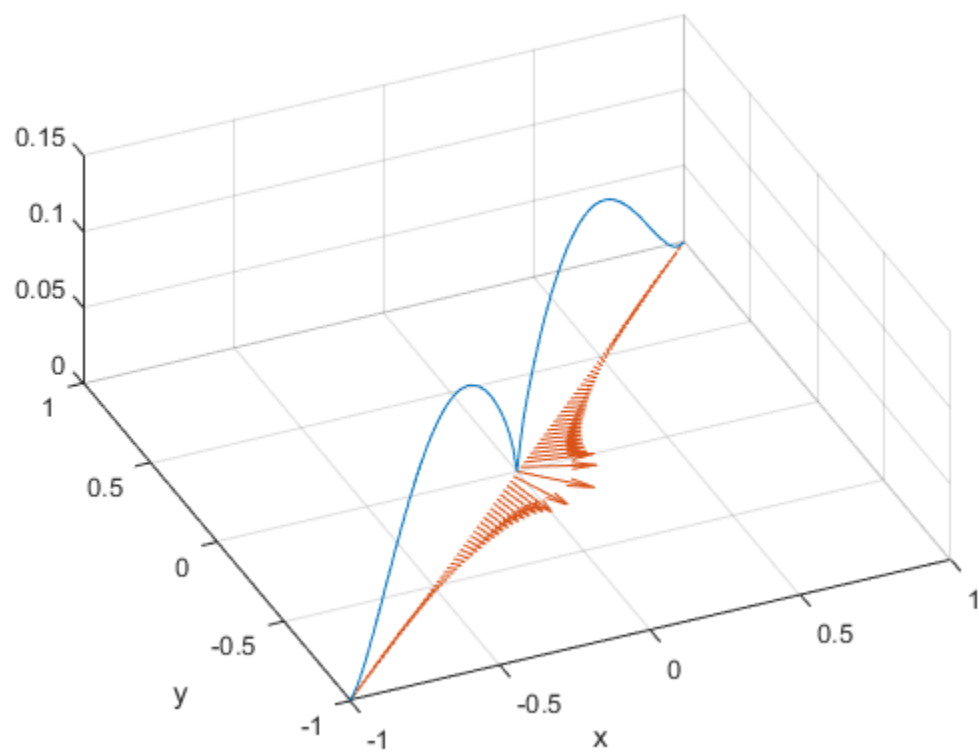
# Plot 3-D Solutions and Their Gradients

---

**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 3-210.

    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 3-214 and "Contour
    Slices Through a 3-D Solution" on page 3-218. 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 3-225.
*   You can use any MATLAB plotting command to create 3-D plots. See "Techniques
    for Visualizing Scalar Volume Data" (MATLAB), "Visualizing Vector Volume Data"
    (MATLAB), and "Graphics" (MATLAB).

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');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
view(-40,24)
```
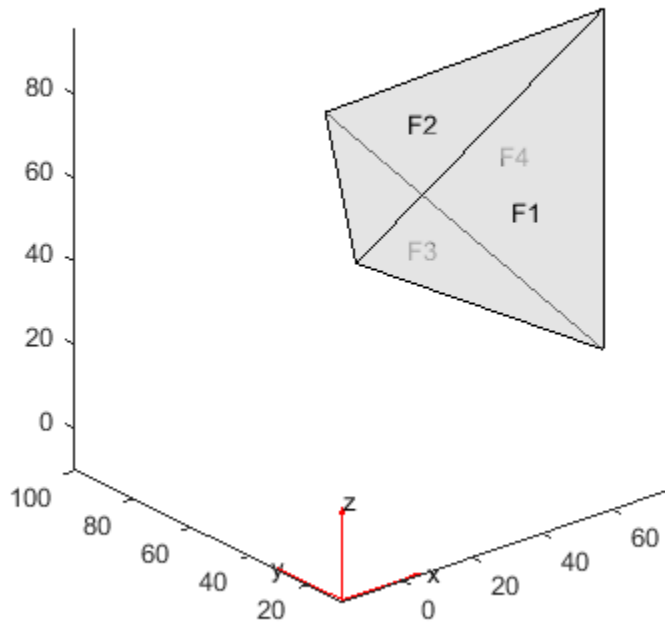


Create a problem with zero Dirichlet boundary conditions on face 4.

```
applyBoundaryCondition(model,'dirichlet','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)
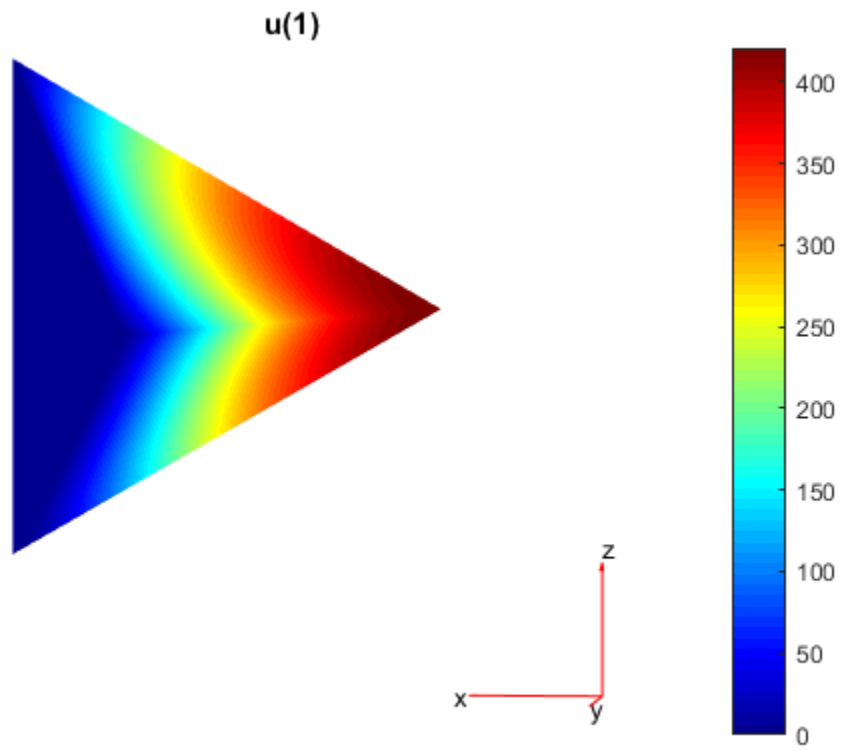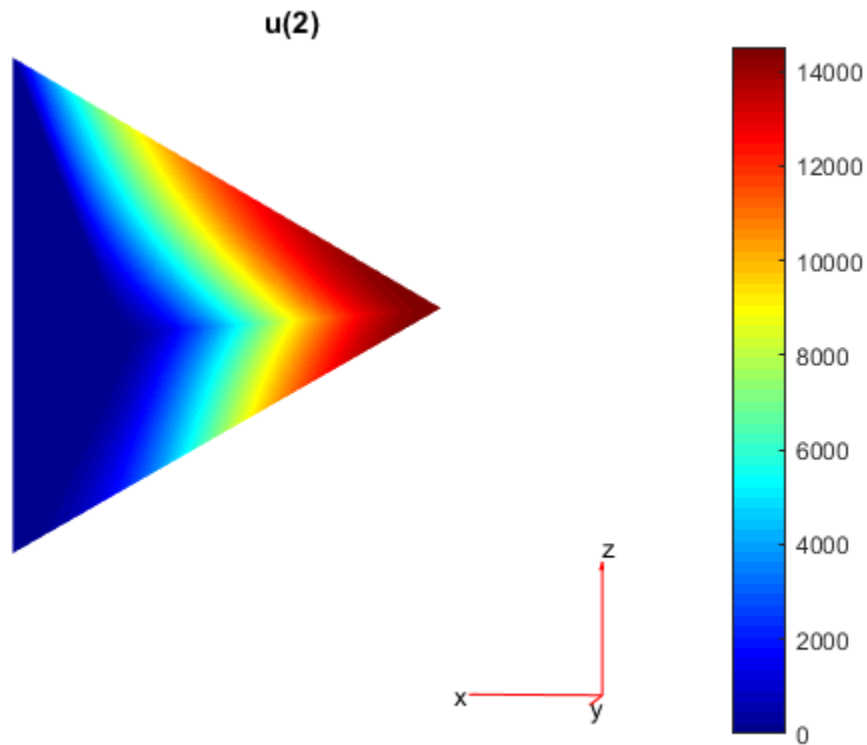
## 2-D Slices Through 3-D Geometry

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".

```
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');
pdegplot(g,'FaceLabels','on','FaceAlpha',0.5)
```
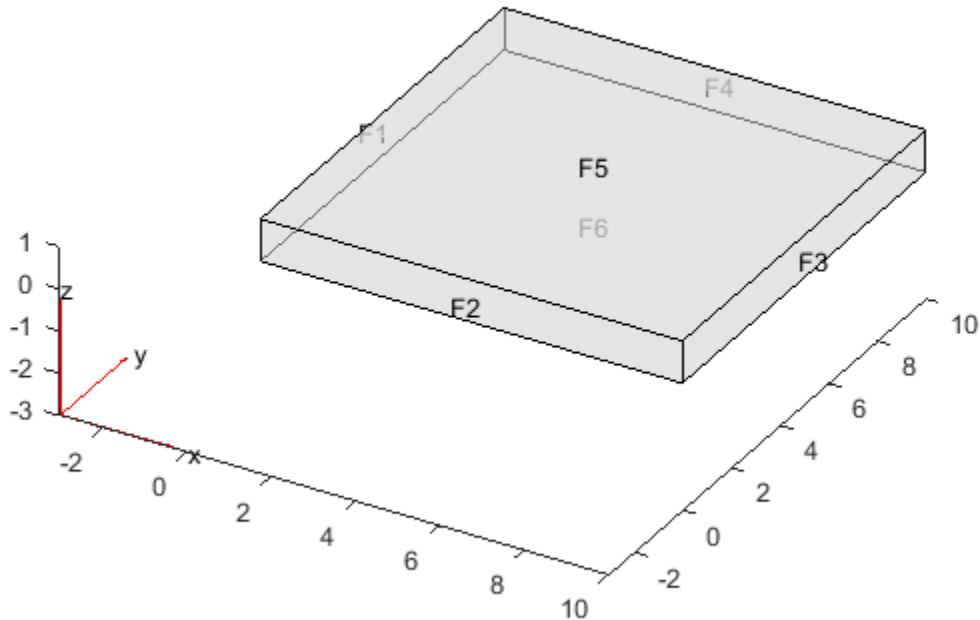
The faces are numbered 1 through 6.

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,'dirichlet','face',1:6,'u',0);
```

Set a zero initial condition.

```
setInitialConditions(model,0);
```

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])
```
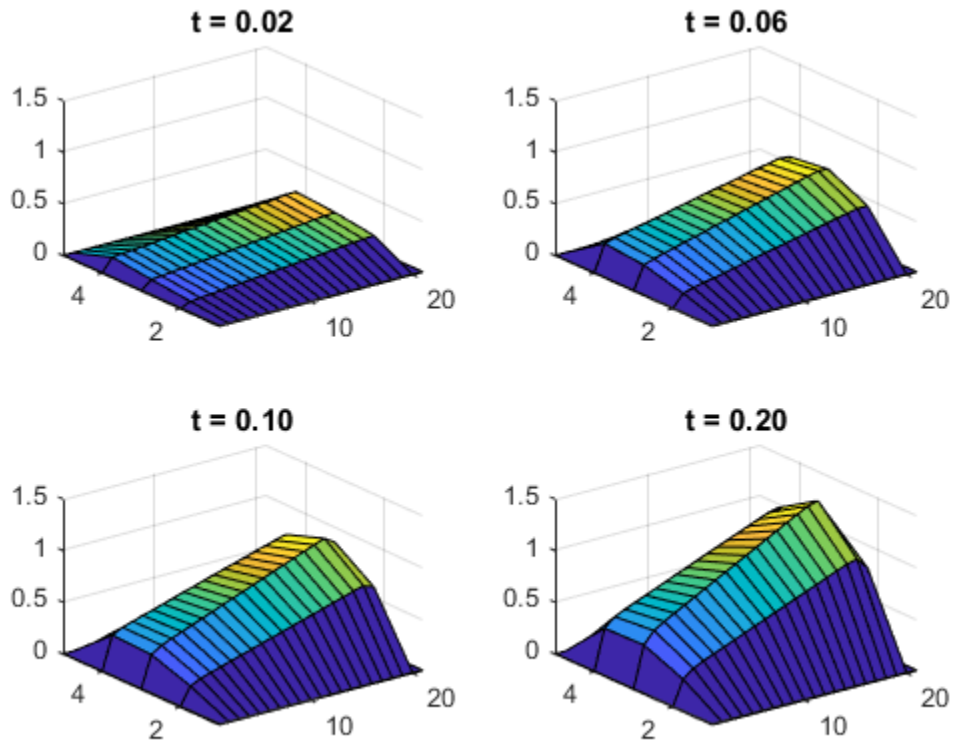
## Contour Slices Through a 3-D Solution

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');
pdegplot(model,'FaceAlpha',0.5)
view(-42,24)
```
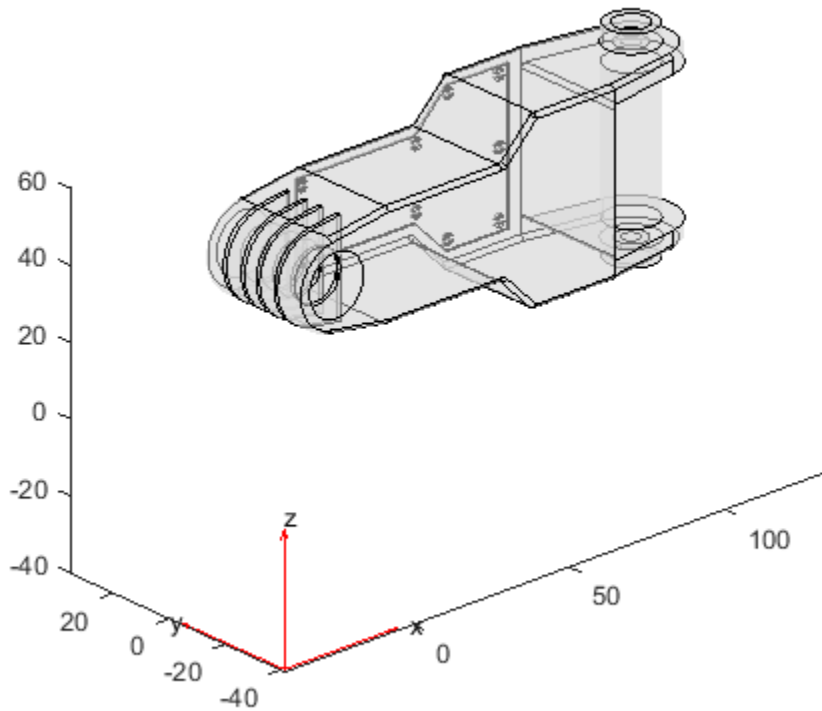
### Specify PDE Coefficients

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,'dirichlet','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
```
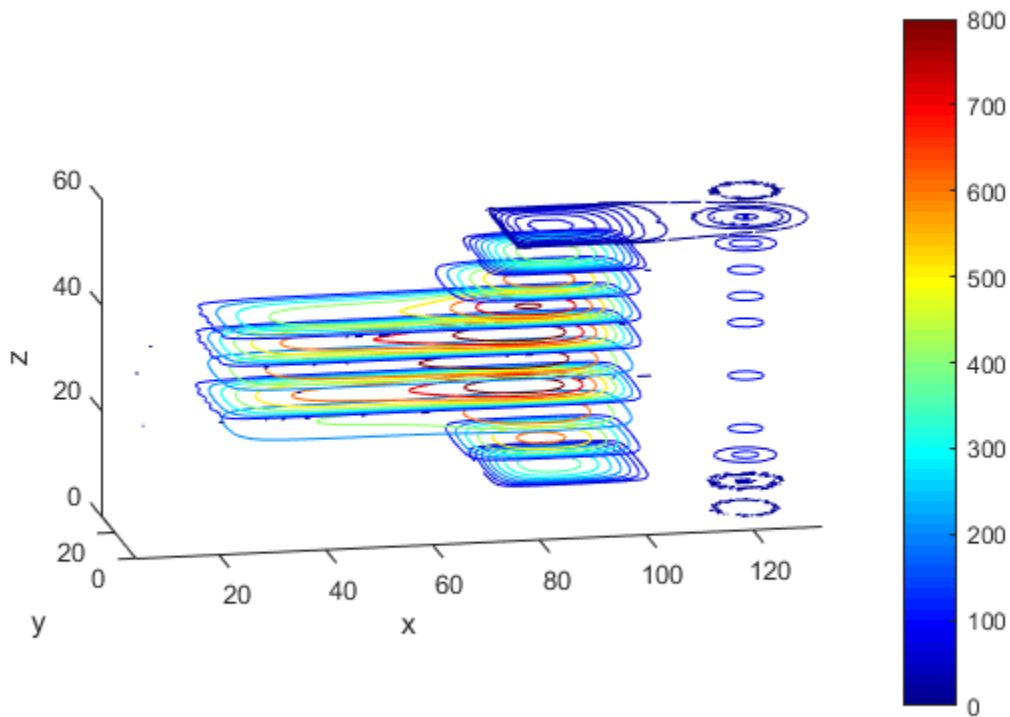
Plot contour slices for various values of $y$.

```
figure
colormap jet
contourslice(X,Y,Z,V,[],1:6:31,[])
xlabel('x')
ylabel('y')
zlabel('z')
colorbar
view(-62,34)
axis equal
```
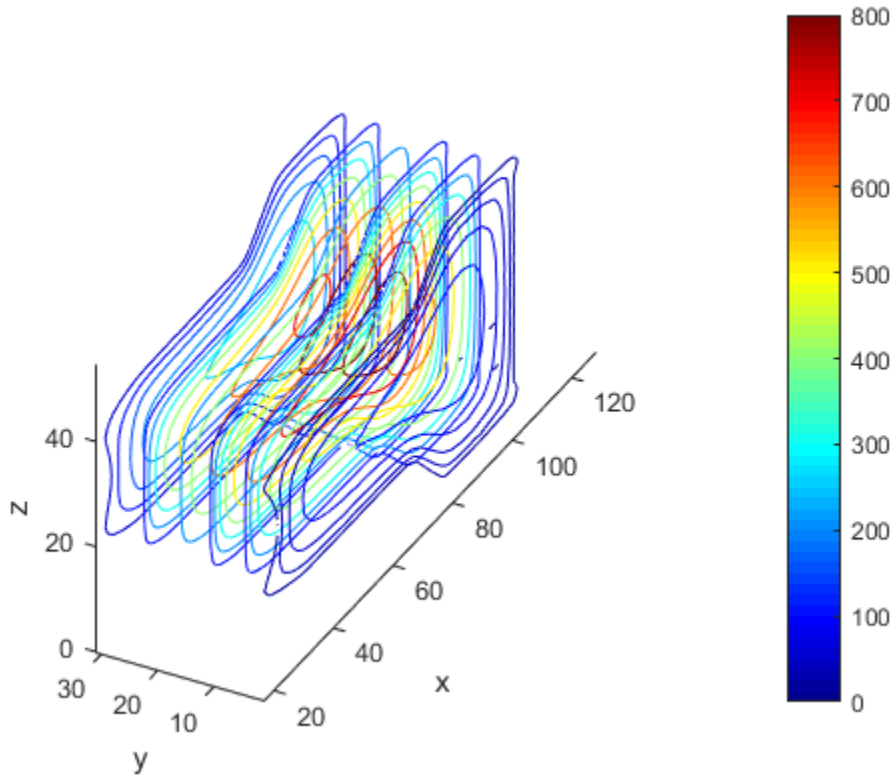
### Save Memory by Evaluating As Needed

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 $0 \le x \le 135$, $0 \le y \le 35$, 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
```
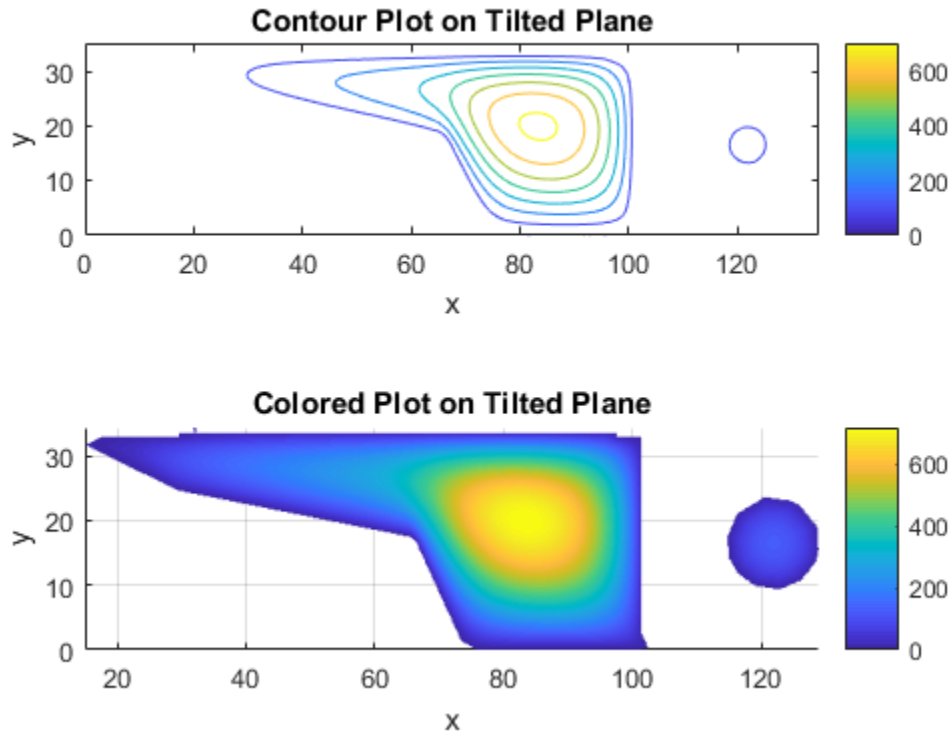
## Plots of Gradients and Streamlines

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');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
view(-42,24)
```
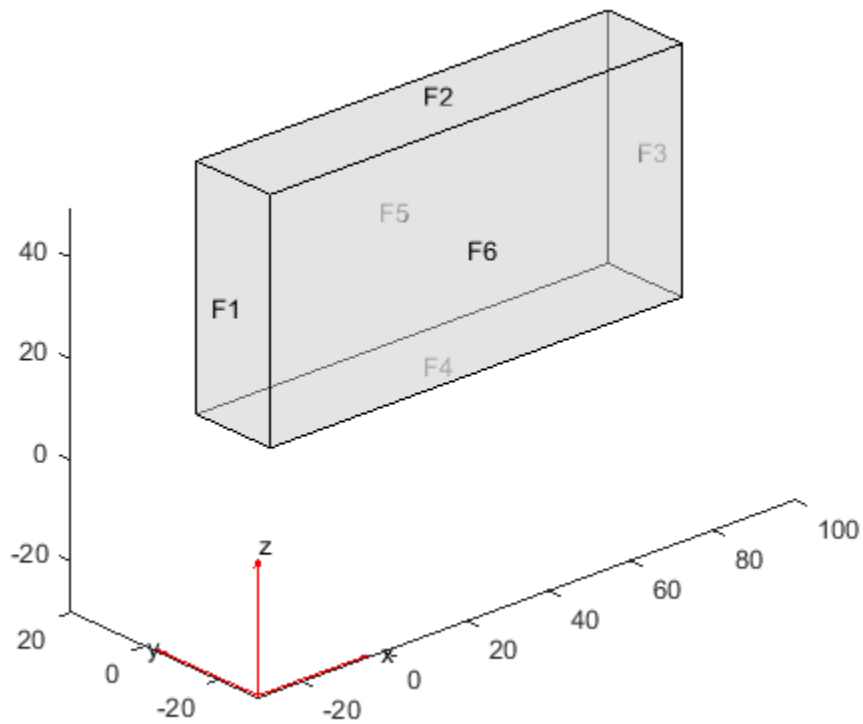


**Set Boundary Conditions**

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,'dirichlet','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
```
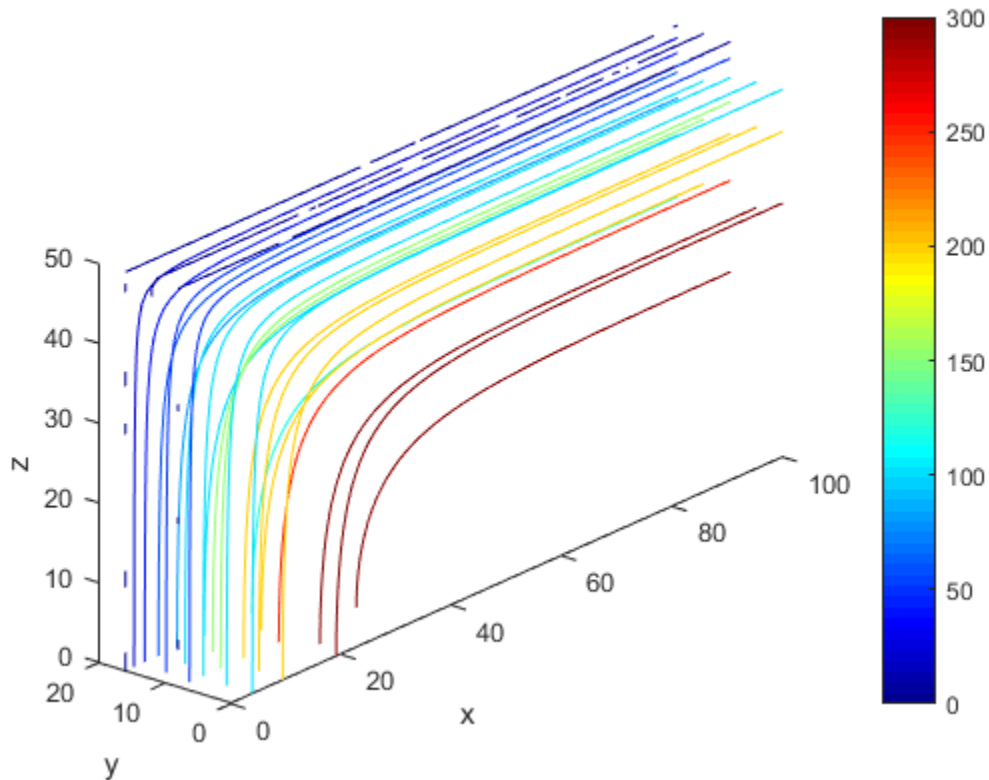


The particle has the largest mean exit time near the point $\left( x,y,z \right) = \left( 100,0,0 \right)$.

### 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')
```
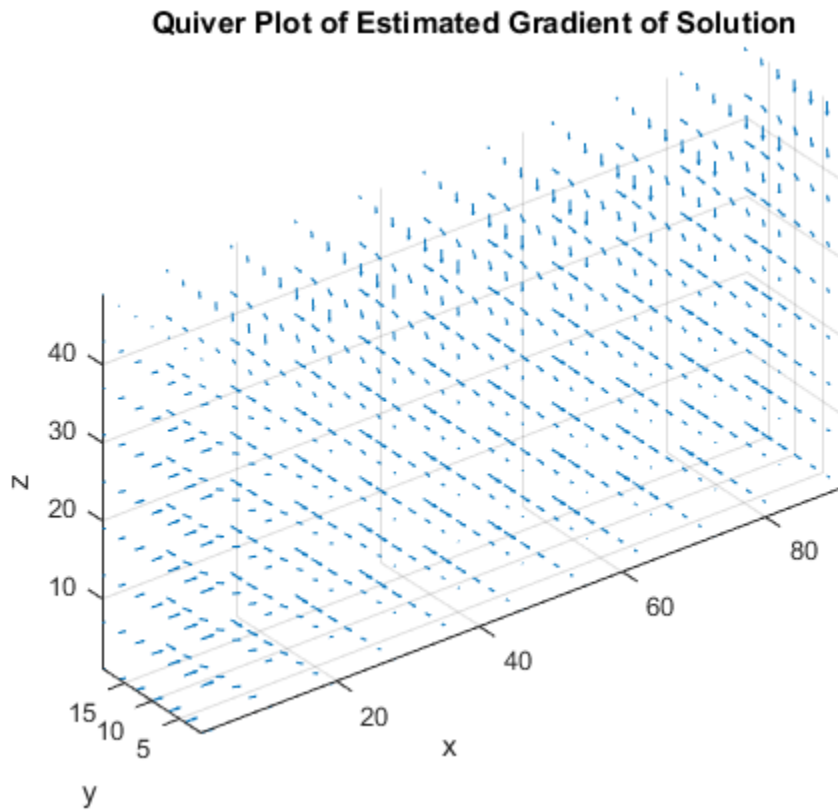
Plot the streamlines of the approximate gradient. Start the streamlines from a sparser set of initial points.

```
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
```
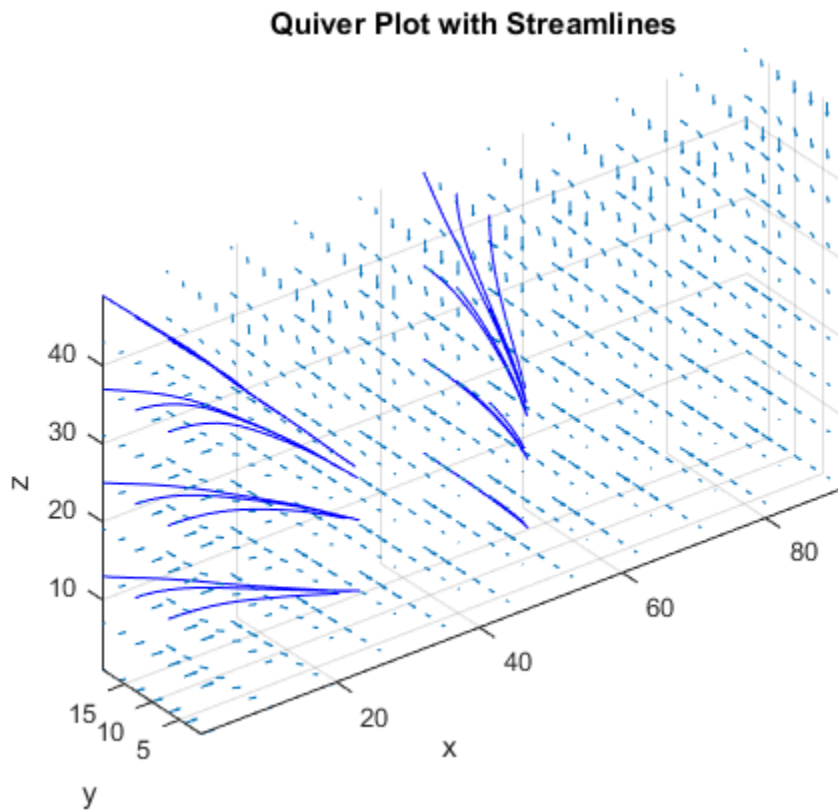


The streamlines show that small values of y and z give larger mean exit times. They also show that the x-coordinate has a significant effect on u when x is small, but when x is

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 2-5

# Dimensions of Solutions, Gradients, and Fluxes

`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`. You also can compute flux of the solution at the mesh nodes and at arbitrary points by using `evaluateCGradient`.

---

**Note:** `solvepde` does not compute components of flux of a PDE solution. To compute flux of the solution at the mesh nodes, use `evaluateCGradient`.

---

`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, its gradient, and flux of the solution 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`, `evaluateGradient`, and `evaluateCGradient` 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`, `evaluateGradient`, and `evaluateCGradient`, 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`, `evaluateGradient`, and `evaluateCGradient`, this is the number of query times for time-dependent problems or query modes for eigenvalue problems.

### Stationary Scalar Problem
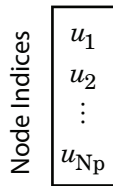
### Time-Dependent or Eigenvalue Scalar Problem
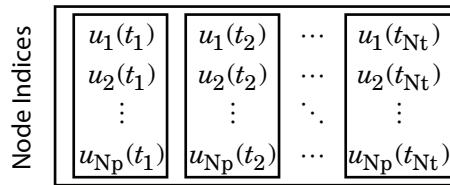


### Stationary System
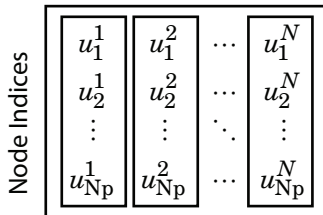
### Time-Dependent or Eigenvalue System
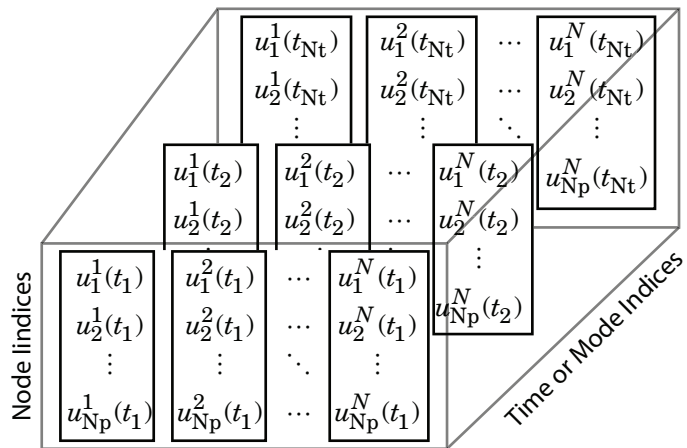


Suppose you have a problem in which:

- `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 compute the solution, its gradient, or flux of the solution at other points ("query points") in the geometry by using `interpolateSolution`, `evaluateGradient`, or `evaluateGradient`, respectively. 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, gradients, and flux of the solution at nodal locations (returned by `solvepde`,`solvepdeeig`, and `evaluateCGradient`) and the dimensions of interpolated solutions and gradients (returned by `interpolateSolution`, `evaluateGradient`, and `evaluateCGradient`).

| Stationary PDE problem | Access solution and components of gradient | Size of `NodalSolution`, `XGradients`, `YGradients`, `ZGradients`, and components of flux at nodal points | Size of solution, components of gradient, and components of flux at query points |
|---|---|---|---|
| Scalar | `result.NodalSolution(iP)`<br><br>`result.XGradients(iP)` | Np-by-1 | Nqp-by-1 |

| Stationary PDE problem | Access solution and components of gradient | Size of `NodalSolution`, `XGradients`, `YGradients`, `ZGradients`, and components of flux at nodal points | Size of solution, components of gradient, and components of flux at query points |
|---|---|---|---|
| | `result.YGradients(iP)` `result.ZGradients(iP)` | | |
| 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`, `ZGradients`, and components of flux at nodal points | Size of solution, components of gradient, and components of flux at query points |
|---|---|---|---|
| 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.

# Open the PDE App

You can open the PDE app using the **Apps** tab or typing the commands in the MATLAB Command Window.

### Use the Apps Tab

1  On the MATLAB Toolstrip, click the **Apps** tab.
2  On the **Apps** tab, click the down arrow at the end of the **Apps** section.
3  Under **Math, Statistics and Optimization**, click the **PDE** button.

### Use Commands

- To open a blank PDE app window, type `pdetool` in the MATLAB Command Window.
- To open the PDE app with a circle already drawn in it, type `pdecirc` in the MATLAB Command Window.
- To open the PDE app with an ellipse already drawn in it, type `pdeellip` in the MATLAB Command Window.
- To open the PDE app with a rectangle already drawn in it, type `pderect` in the MATLAB Command Window.
- To open the PDE app with a polygon already drawn in it, type `pdepoly` in the MATLAB Command Window.

You can use a sequence of drawing commands to create several basic shapes. For example, the following commands create a circle, a rectangle, an ellipse, and a polygon:

```
pderect([-1.5,0,-1,0])
pdecirc(0,0,1)
pdepoly([-1,0,0,1,1,-1],[0,0,1,1,-1,-1])
pdeellip(0,0,1,0.3,pi)
```

If you start using the PDE app and then reopen the app by calling the `pdetool` command or by using the **Apps** tab, you overwrite any existing information in the app.

# Create 2-D Geometry in the PDE App

## Create Basic Shapes

The PDE app lets you draw four basic shapes: a circle, an ellipse, a rectangle, and a polygon. To draw a basic shape, use the **Draw** menu or one of the following buttons on the toolbar. To cut, clear, copy, and paste the solid objects, use the **Edit** menu.

| | |
|---|---|
| ▢ | Draw a rectangle/square starting at a corner.<br><br>Using the left mouse button, drag to create a rectangle. Using the right mouse button (or **Ctrl**+click), drag to create a square. |
| ⊞ | Draw a rectangle/square starting at the center.<br><br>Using the left mouse button, drag to create a rectangle. Using the right mouse button (or **Ctrl**+click), drag to create a square. |
| ◯ | Draw an ellipse/circle starting at the perimeter.<br><br>Using the left mouse button, drag to create an ellipse. Using the right mouse button (or **Ctrl**+click), drag to create a circle. |
| ⊕ | Draw an ellipse/circle starting at the center.<br><br>Using the left mouse button, drag to create an ellipse. Using the right mouse button (or **Ctrl**+click), drag to create a circle. |
| ▷ | Draw a polygon.<br><br>Using the left mouse button, 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. |

Alternatively, you can create a basic shape by typing one of the following commands in the MATLAB Command Window. These commands open the PDE app with the requested shape already drawn in it. If the app is already open, these commands add the requested shape to the app window without deleting any existing shapes.

- pdecirc draws a circle.
- pdeellip draws an ellipse.

- `pderect` draws a rectangle.
- `pdepoly` draws a polygon.

You can use a sequence of drawing commands to create several basic shapes. For example, the following commands create a circle, a rectangle, an ellipse, and a polygon:

```
pderect([-1.5,0,-1,0])
pdecirc(0,0,1)
pdepoly([-1,0,0,1,1,-1],[0,0,1,1,-1,-1])
pdeellip(0,0,1,0.3,pi)
```

## Select Several Shapes

- To select a single shape, click it using the left mouse button.
- To select several shapes and to deselect shapes, use **Shift**+click (or click using the middle mouse button). Note that clicking outside of all shapes, deselects all shapes.
- To select all the intersecting shapes, click the intersection of these shapes.
- To select all shapes, use the **Select All** option from the **Edit** menu.

## Rotate Shapes

To rotate a shape:

1 Select the shapes.
2 Select **Rotate** from the **Draw** menu.
3 In the resulting **Rotate** dialog box, enter the rotation angle in degrees. To rotate counterclockwise, use positive values of rotation angles. To rotate clockwise, use negative values.
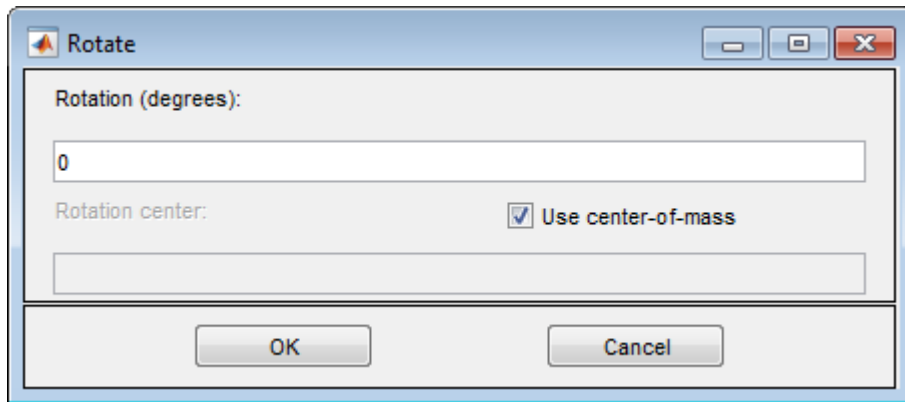
**4** By default, the rotation center is the center-of-mass of the selected shapes. To use a different rotation center, clear the **Use center-of-mass** option and enter a rotation center (*xc*,*yc*) as a 1-by-2 vector, for example, [-0.4 0.3].

## Create Complex Geometries

You can specify complex geometries by overlapping basic shapes. This approach is called Constructive Solid Geometry (CSG). The PDE app lets you combine basic shapes by using their unique names.

The app assigns a unique name to each shape. The names depend on the type of the shape:

- For circles, the default names are C1, C2, C3, and so on.
- For ellipses, the default names are E1, E2, E3, and so on.
- For polygons, the default names are P1, P2, P3, and so on.
- For rectangles, the default names are R1, R2, R3, and so on.
- For squares, the default names are SQ1, SQ2, SQ3, and so on.

To change the name and parameters of a shape, switch to the draw mode by selecting **Draw Mode** from the **Draw** menu. Double-click the shape. The resulting dialog box lets you change the name and parameters of the selected shape. The name cannot contain spaces.
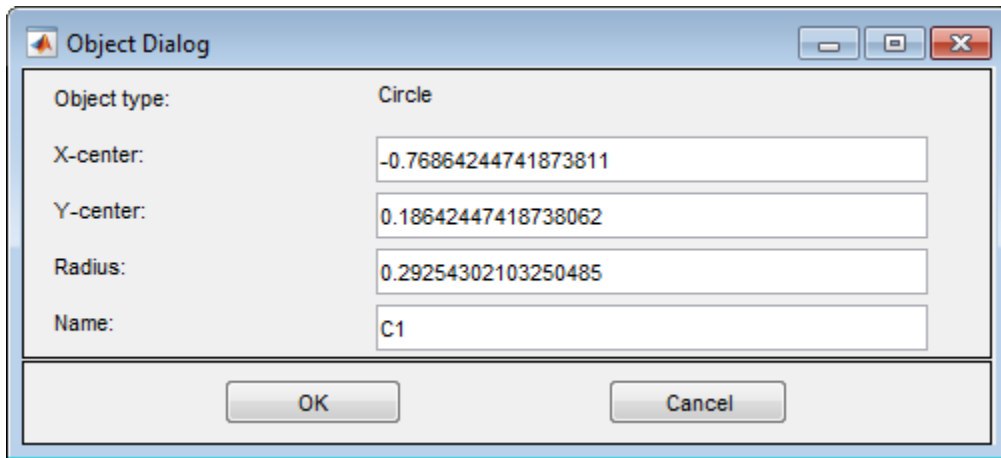
Now you can combine basic shapes to create a complex geometry. To do this, use the **Set formula** field located under the toolbar. Here you can specify a geometry by using the names of basic shapes and the following operators:

- + is the set union operator. For example, SQ1+C2 creates a geometry comprised of all points of the square SQ1 and all points of the circle C2.

- * is the set intersection operator. For example, SQ1*C2 creates a geometry comprised of the points that belong to both the square SQ1 and the circle C2.

- - is the set difference operator. For example, SQ1-C2 creates a geometry comprised of the points of the square SQ1 that do not belong to the circle C2.

The operators + and * have the same precedence. The operator - has a higher precedence. You can control the precedence by using parentheses. The resulting geometrical model (called *decomposed geometry*) is the set of points for which the set formula evaluates to true. By default, it is the union of all basic shapes.

## Adjust Axes Limits and Grid

To adjust axes limits:

- Select **Axes Limits** from the **Options** menu

- Specify the range of the *x*-axis and the *y*-axis as a 1-by-2 vector such as [-10 10]. If you select **Auto**, the app uses automatic scaling for the corresponding axis.
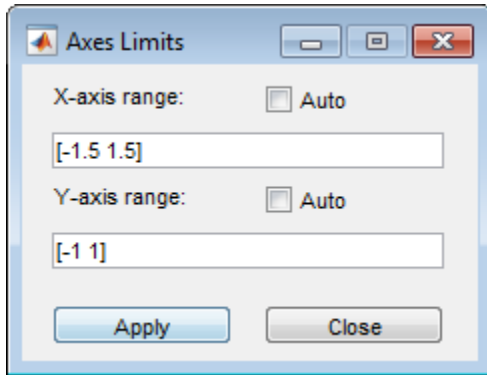
- Click **Apply** to apply the specified axes ranges.
- Click **Close** to close the dialog box.

To add axis grid, the "snap-to-grid" feature, and zoom, use the **Options** menu. To adjust the grid spacing:

- Select **Grid Spacing** from the **Options** menu.
- By default, the app uses automatic linear grid spacing. To enable editing the fields for linear spacing and extra ticks, clear **Auto**.
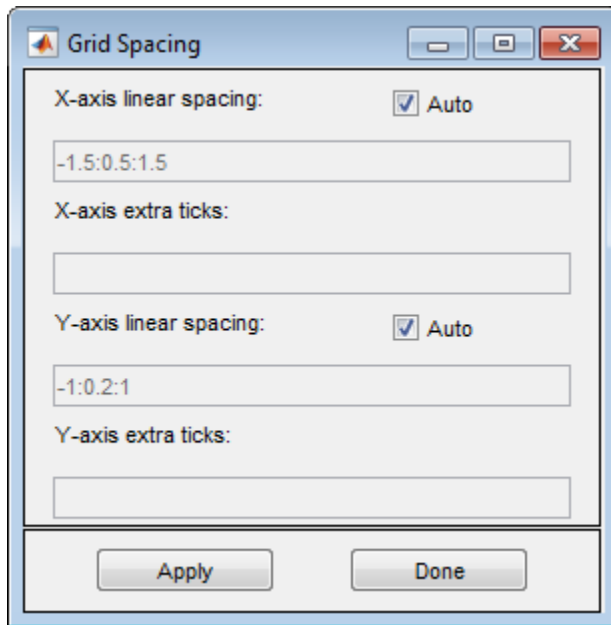
- Specify the grid spacing for the *x*-axis and *y*-axis. For example, change the default linear spacing `-1.5:0.5:1.5` to `-1:0.2:1`.

  You also can add extra ticks to customize the grid and aid in drawing. To separate extra tick entries, use spaces, commas, semicolons, or brackets.
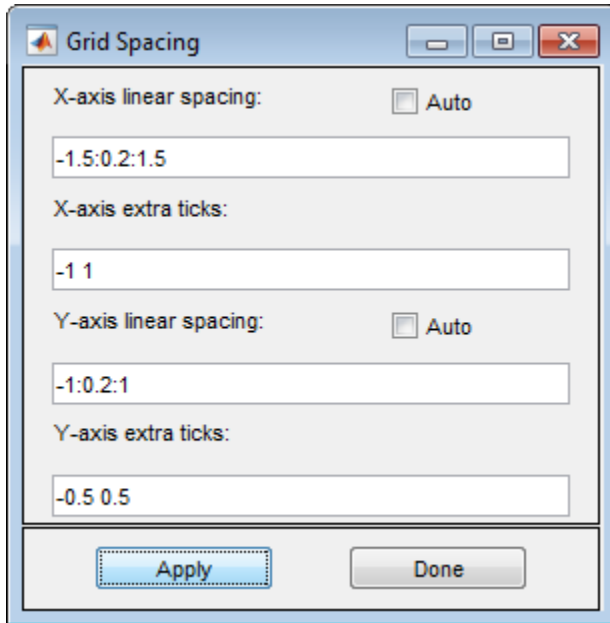
- Click **Apply** to apply the specified grid spacing.
- Click **Done** to close the dialog box.
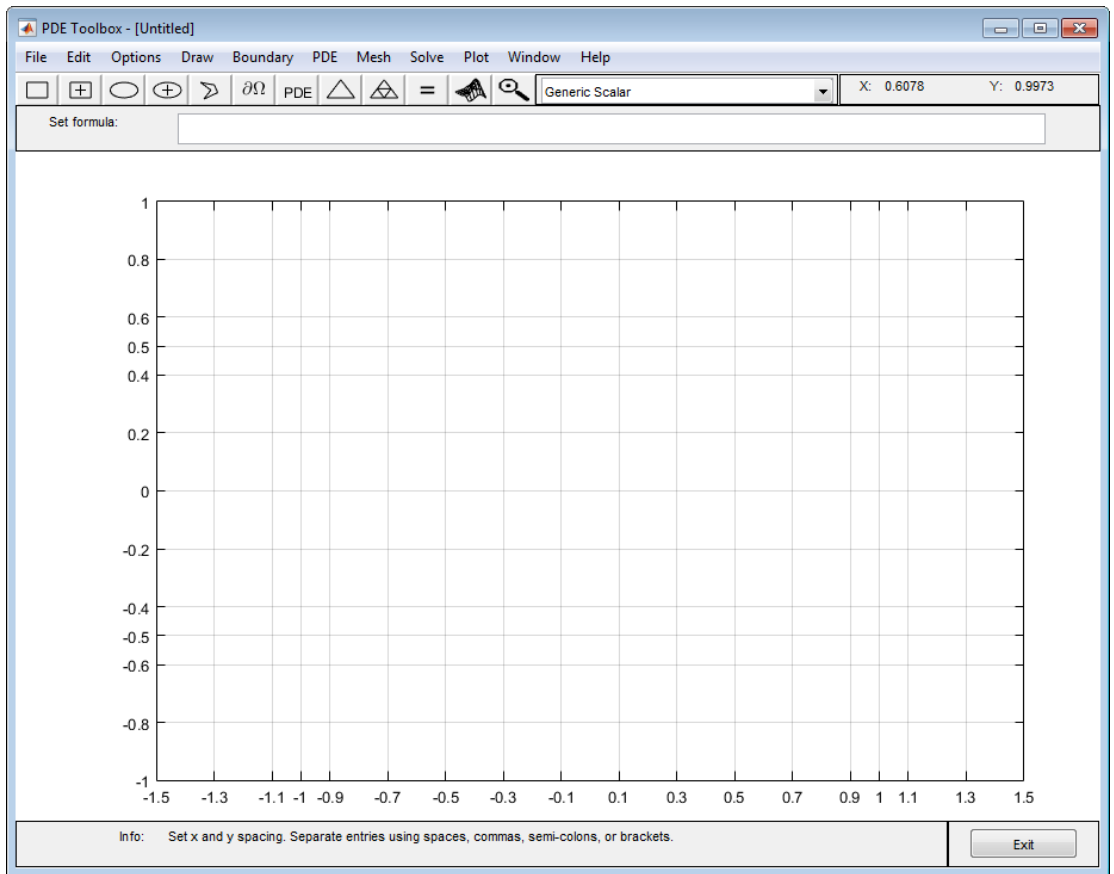
# Rounded Corners Using CSG Modeling

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.
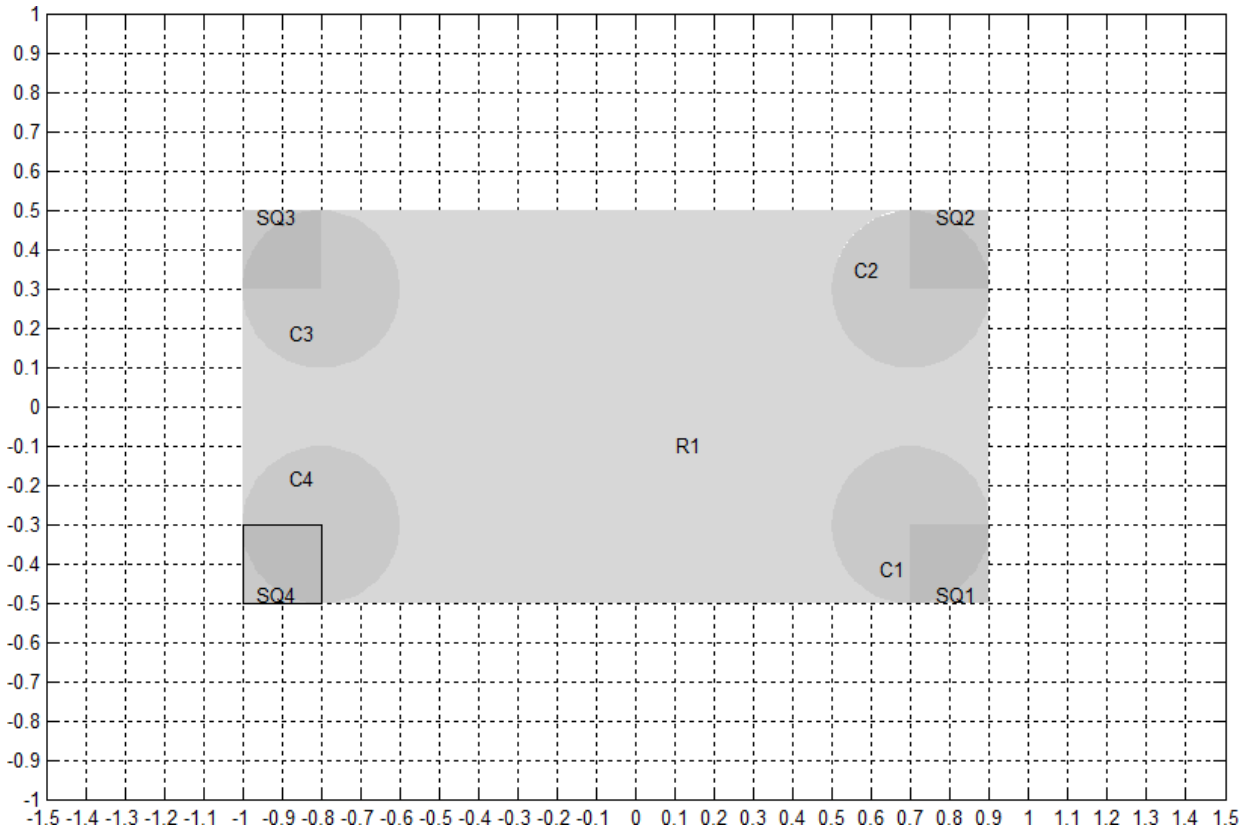
Edit the set formula. To get rounded corners, subtract the small squares from the rectangle and then add the circles. As a set formula, express this as

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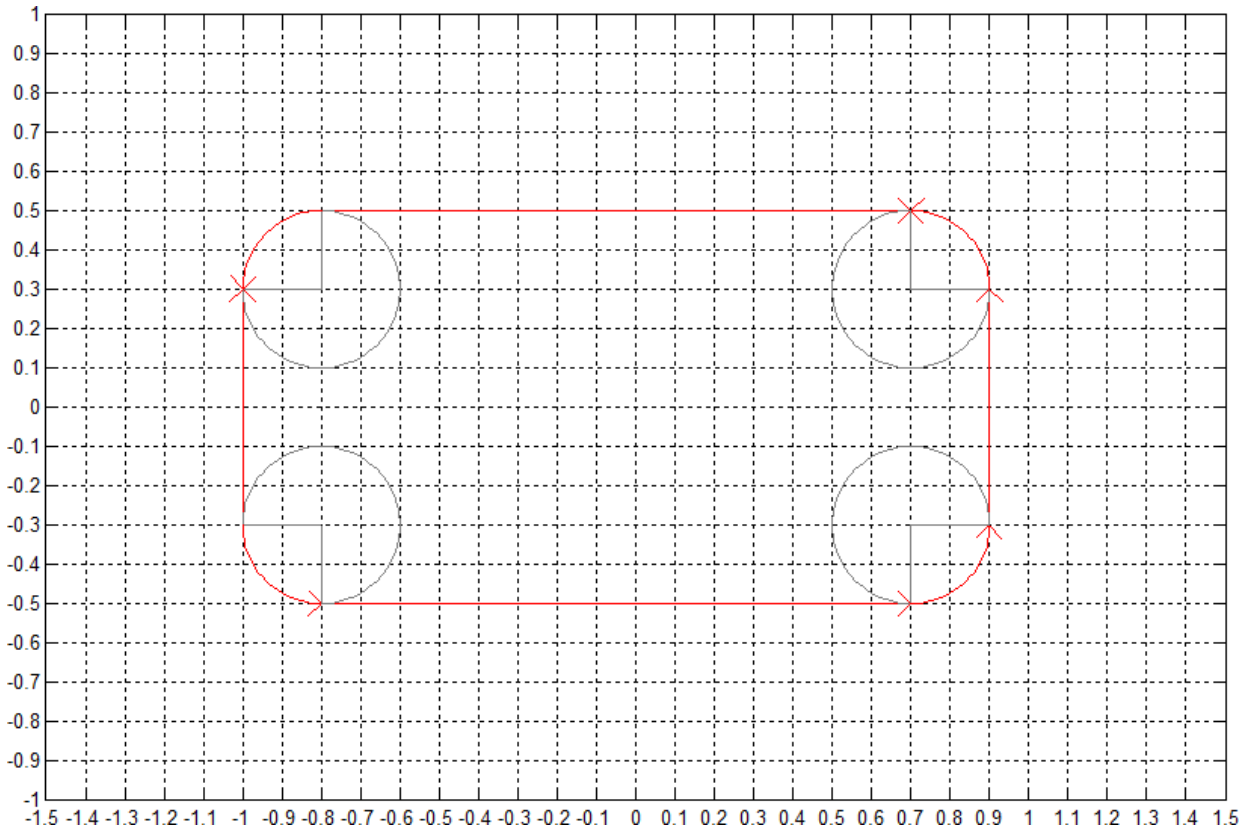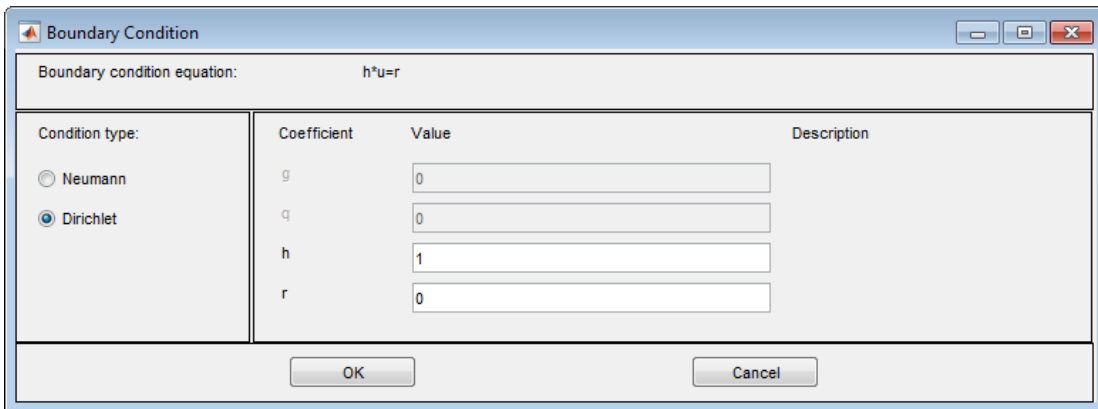
Because of the intersection of the solid objects used in the initial CSG model, a number of subdomain borders remain. They are drawn using gray lines. If this is a model of, e.g., a homogeneous plate, you can remove these borders. Select the **Remove All Subdomain Borders** option from the **Boundary**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

# Specify Boundary Conditions in the PDE App

Select **Boundary Mode** from the **Boundary** menu or click the [∂Ω] button. Then select a boundary or multiple boundaries for which you are specifying the conditions. Note that no if you do not select any boundaries, then the specified conditions apply to all boundaries.

- To select a single boundary, click it using the left mouse button.
- To select several boundaries and to deselect them, use **Shift**+click (or click using the middle mouse button).
- To select all boundaries, use the **Select All** option from the **Edit** menu.

Select **Specify Boundary Conditions** from the **Boundary** menu.



**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

Select **PDE Mode** from the **PDE** menu. Then select a geometrical region or multiple regions for which you are specifying the coefficients. Note that no if you do not select any regions, then the specified coefficients apply to all regions.

- To select a single region, click it using the left mouse button.
- To select several regions and to deselect them, use **Shift**+click (or click using the middle mouse button). Note that clicking outside of all regions, deselects all regions.
- To selects all the intersecting regions, click on the intersection of these regions.
- To select all regions, use the **Select All** option from the **Edit** menu.

Select **PDE Specification** from the **PDE** menu or click the **PDE** button on the toolbar.
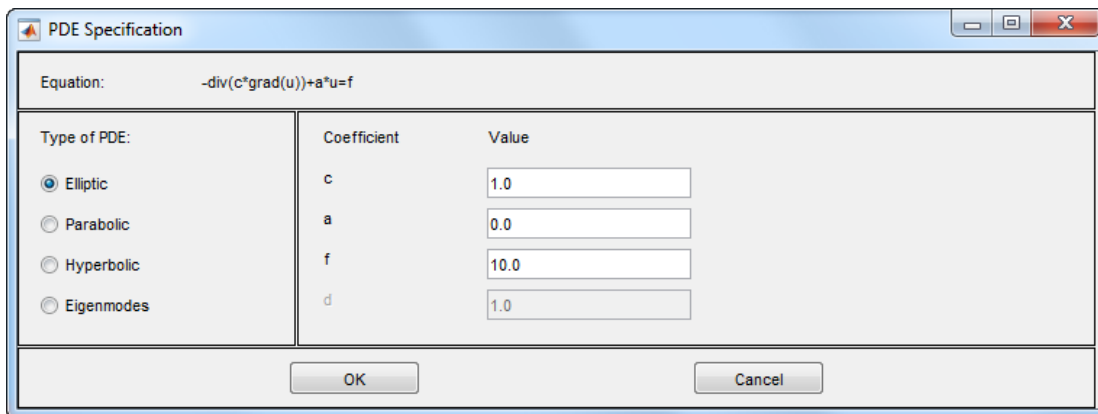


**PDE Specification** opens a dialog box where you enter the type of partial differential equation and the applicable parameters. The dimension of the parameters depends on the dimension of the PDE. The following description applies to scalar PDEs. If you select a nongeneric application mode, application-specific 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 2-58 and "Systems in the PDE App" on page 2-80.

# 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.

Fo 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.

To initialize a triangular mesh, select **Initialize Mesh** from the **Mesh** menu or click the

button. To refine a mesh, select **Refine Mesh** from the **Mesh** menu or click the 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. After you adjust the parameters, solve the PDE by selecting **Solve PDE** from the **Solve** menu or by clicking the ▬ button.

## 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 4-19.

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 $l$p norm. The default is `Inf`, the infinity (maximum) norm.

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

## Parabolic Equations
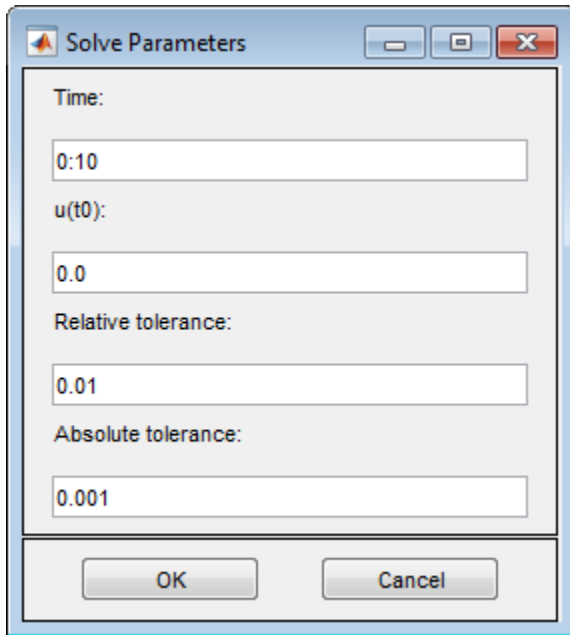


The solve parameters for the parabolic PDEs are:

- **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
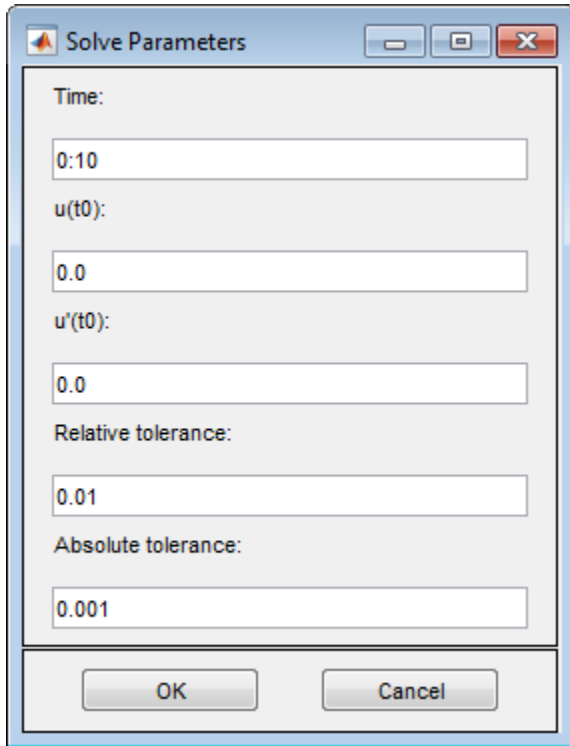


The solve parameters for the hyperbolic PDEs are:

- **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]`
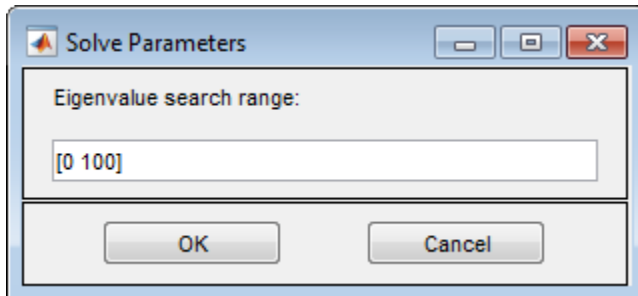
# Plot the Solution in the PDE App

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 or click the ⚛ button.
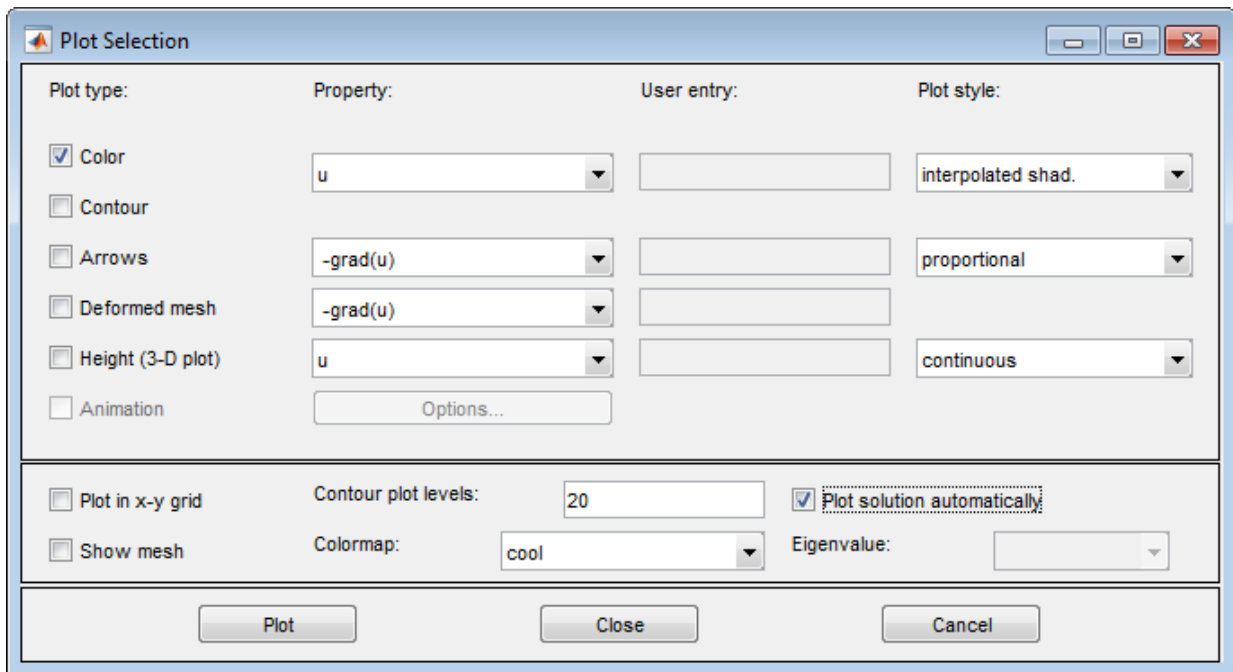


**Parameters** opens a dialog box containing options controlling the plotting and visualization.

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 $\nabla u$, 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.
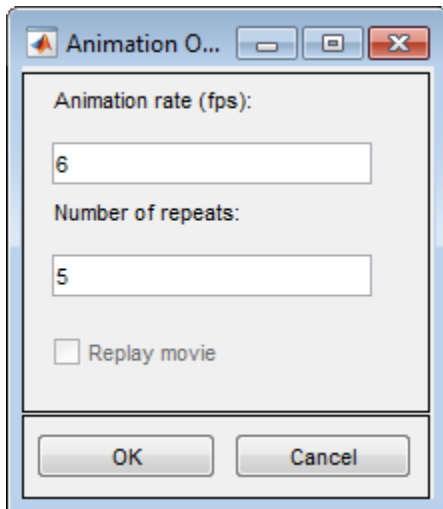


Also, the **Animation** plot type is enabled. In its property field you find an **Options** button. If you press it, an additional dialog box appears. It contains parameters that control the animation:

- **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.

**4-31**
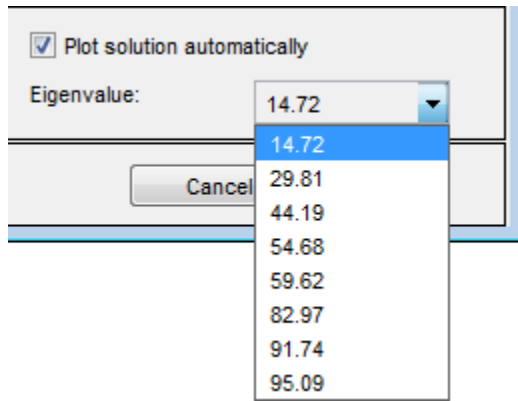
For eigenvalue problems, the bottom right part of the dialog box contains a pop-up menu with all eigenvalues. The plotted solution is the eigenvector associated with the selected eigenvalue. By default, the smallest eigenvalue is selected.

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.

## 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.

**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 $\Omega$, where $\Omega$ is a bounded domain in two or three dimensions. $c$, $a$, $f$, and the unknown solution $u$ are complex functions defined on $\Omega$. $c$ can also be a 2-by-2 matrix function on $\Omega$. 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 $\Omega$ 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 2-36, 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:

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

Replace the original problem with *Find u such that*

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

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 $N_p\to\infty$. Since the differential operator is linear, we demand that the variational equation is satisfied for $N_p$ test-functions $\Phi_i\in V_{N_p}$ that form a basis, i.e.,

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

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

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

Use the following notations:

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

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

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

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

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

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 $\Omega$.

A suitable basis for $V_{N_p}$ in 2-D is the set of "tent" or "hat" functions $\phi_i$. 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 $\phi_i$ 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.
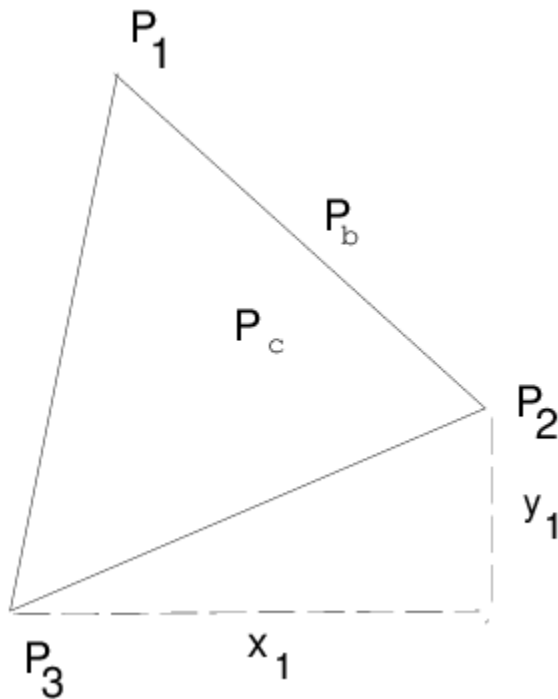


**The Local Triangle P1P2P3**

**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 $\phi_1$, $\phi_2$, and $\phi_3$ 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)

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

If two vertices of the triangle lie on the boundary $\partial\Omega$, they contribute to the line integrals associated to the boundary conditions. If the two boundary points are $P_1$ and $P_2$, then we have

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

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 $(r,s,t)$.
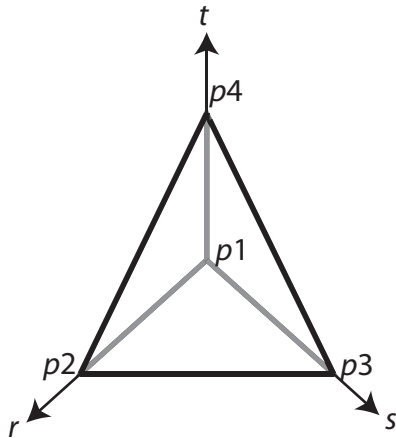


In local coordinates, the point $p1$ is at $(0,0,0)$, $p2$ is at $(1,0,0)$, $p3$ is at $(0,1,0)$, and $p4$ is at $(0,0,1)$.

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.
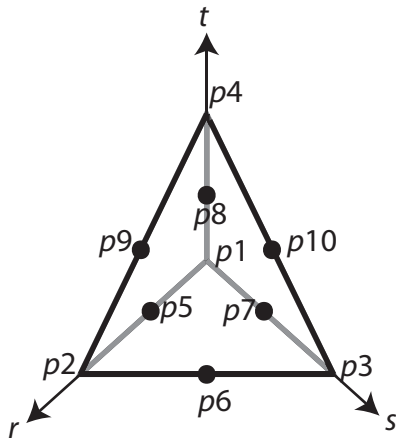
The corresponding basis functions are

$$\phi_1 = 2(1 - r - s - t)^2 - (1 - r - s - t)$$
$$\phi_2 = 2r^2 - r$$
$$\phi_3 = 2s^2 - s$$
$$\phi_4 = 2t^2 - t$$
$$\phi_5 = 4r(1 - r - s - t)$$
$$\phi_6 = 4rs$$
$$\phi_7 = 4s(1 - r - s - t)$$
$$\phi_8 = 4t(1 - r - s - t)$$
$$\phi_9 = 4rt$$
$$\phi_{10} = 4st$$

As in the 2-D case, a 3-D basis function $\phi_i$ takes the value 0 at all nodes $j$, except for node $i$, where it takes the value 1.

## See Also
FEMesh

## More About

- "Elliptic Equations" on page 5-2

# Systems of PDEs

Partial Differential Equation Toolbox software can also handle systems of $N$ partial differential equations over the domain $\Omega$. 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 **h**-matrix is $M$-by-$N$, $M \geq 0$. The generalized Neumann condition contains a source $\mathbf{h}'\mu$, where the Lagrange multipliers $\mu$ 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 **h**-matrix. For the general case, Dirichlet conditions **hu** = **r**

are approximated by adding a term

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

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 $N_p N = N_u$. When Dirichlet boundary conditions fix some of the unknowns, the linear system can be correspondingly reduced. This is easily done by removing rows and columns when $u$ values are given, but here we must treat the case when some linear combinations of the components of $u$ are given, $\mathbf{hu} = \mathbf{r}$. These are collected into $HU = R$ where $H$ is an $M$-by-$N_u$ matrix and $R$ is an $M$-vector.

With the reaction force term the system becomes
$KU + H' \mu = F$
$HU = R.$

The constraints can be solved for $M$ of the $U$-variables, the remaining called $V$, an $N_u - M$ vector. The null space of $H$ is spanned by the columns of $B$, and $U = BV + u_d$ makes $U$ satisfy the Dirichlet conditions. A permutation to block-diagonal form exploits the sparsity of $H$ to speed up the following computation to find $B$ in a numerically stable way. $\mu$ 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. $\rho$ is the density, $C$ is the thermal capacity, $k$ is the thermal conductivity, $h$ is the film coefficient, $u_\infty$ 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 $\Omega$ 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 $\phi_j$, integrating over $\Omega$, 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 $\Omega$

with the original boundary conditions, while $M$ is just the mass matrix of the problem
$-\nabla \cdot (0\nabla u) + du = 0$ in $\Omega$.

When the Dirichlet conditions are time dependent, $F$ contains contributions from time derivatives of $h$ and $\mathbf{r}$. These derivatives are evaluated by finite differences of the user-specified data.

The ODE system is ill conditioned. Explicit time integrators are forced by stability requirements to very short time steps while implicit solvers can be expensive since

they solve an elliptic problem at every time step. The numerical integration of the ODE system is performed by the MATLAB ODE Suite functions, which are efficient for this class of problems. The time step is controlled to satisfy a tolerance on the error, and factorizations of coefficient matrices are performed only when necessary. When coefficients are time dependent, the necessity of reevaluating and refactorizing the matrices each time step may still make the solution time consuming, although `parabolic` reevaluates only that which varies with time. In certain cases a time-dependent Dirichlet matrix $\mathbf{h}(t)$ may cause the error control to fail, even if the problem is mathematically sound and the solution $u(t)$ is smooth. This can happen because the ODE integrator looks only at the reduced solution $v$ with $u = Bv + ud$. As $\mathbf{h}$ changes, the pivoting scheme employed for numerical stability may change the elimination order from one step to the next. This means that $B$, $v$, and $ud$ all change discontinuously, although $u$ itself does not.

# 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 $\Omega$, 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 $\Omega$, and usual boundary conditions. In particular, solutions of the equation $u_{tt} - c\Delta u = 0$ are waves moving with speed $\sqrt{c}$.

Using a given mesh of $\Omega$, 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 $\lambda$ 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 $\lambda$ 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 $\Omega$. 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 Eigenmodes of the L-Shaped Membrane" on page 3-168 or "Eigenvalues and Eigenmodes of a Square" on page 3-179, where actual runs are reported.

First a shift $\mu$ 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 $\mu M$ 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 $\theta_i$ 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 $\mu$.

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 $\theta_i$ and eigenvectors $s_i$ of $H_{j,j}$,

$$H_{j,j}s_i = s_i\theta_i, \quad i = 1,\dots, j$$

Then $y_i = V_j s_i$ is an approximate eigenvector of $A$, and its residual is

$$r_i = Ay_i - y_i\theta_i = AV_j s_i - V_j s_i\theta_i = (AV_j - V_j H_{j,j})s_i = v_{j+1}h_{j+1,j}s_{i,j}$$

This residual has to be small in norm for $\theta_i$ to be a good eigenvalue approximation. The norm of the residual is

$$\|r_i\| = |h_{j+1,j}s_{j,i}|$$

the product of the last subdiagonal element of the Hessenberg matrix and the last element of its eigenvector. It seldom happens that $h_{j+1,j}$ gets particularly small, but after sufficiently many steps $j$ there are always some eigenvectors $s_i$ with small last elements. The long vector $V_{j+1}$ is of unit norm.

It is not necessary to actually compute the eigenvector approximation $y_i$ to get the norm of the residual; we only need to examine the short vectors $s_i$, and flag those with tiny last components as converged. In a typical case $n$ may be 2000, while $j$ seldom exceeds 50, so all computations that involve only matrices and vectors of size $j$ are much cheaper than those involving vectors of length $n$.

This eigenvalue computation and test for convergence is done every few steps $j$, until all approximations to eigenvalues inside the interval [lb,ub] are flagged as converged. When $n$ is much larger than $j$, this is done very often, for smaller $n$ more seldom. When all eigenvalues inside the interval have converged, or when $j$ has reached a prescribed maximum, the converged eigenvectors, or more appropriately *Schur vectors*, are computed and put in the front of the basis $V$.

After this, the Arnoldi algorithm is restarted with a random vector, if all approximations inside the interval are flagged as converged, or else with the best unconverged

approximate eigenvector $y_i$. In each step $j$ of this second Arnoldi run, the vector is made orthogonal to all vectors in $V$ including the converged Schur vectors from the previous runs. This way, the algorithm is applied to a projected matrix, and picks up a second copy of any double eigenvalue there may be in the interval. If anything in the interval converges during this second run, a third is attempted and so on, until no more approximate eigenvalues $\theta_i$ 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 $\phi_i$, integrate on the domain $\Omega$, 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 $\alpha$ 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 $\alpha$

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

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 $\alpha \to 1$ and thus the convergence rate increases. If there is a solution to $\rho(U) = 0$, the scheme ultimately recovers the quadratic convergence rate of the standard Newton iteration.

Closely related to the preceding problem is the choice of the initial guess $U^{(0)}$. By default, the solver sets $U^{(0)}$ and then assembles the FEM matrices $K$ and $F$ and computes
$U^{(1)} = K^{-1}F$

The damped Gauss-Newton iteration is then started with $U^{(1)}$, which should be a better guess than $U^{(0)}$. If the boundary conditions do not depend on the solution $u$, then $U^{(1)}$ satisfies them even if $U^{(0)}$ does not. Furthermore, if the equation is linear, then $U^{(1)}$ is the exact FEM solution and the solver does not enter the Gauss-Newton loop.

There are situations where $U^{(0)} = 0$ makes no sense or convergence is impossible.

In some situations you may already have a good approximation and the nonlinear solver can be started with it, avoiding the slow convergence regime. This idea is used in the adaptive mesh generator. It computes a solution $\tilde{U}$ on a mesh, evaluates the error, and may refine certain triangles. The interpolant of $\tilde{U}$ is a very good starting guess for the solution on the refined mesh.

In general the exact Jacobian

$$J_n = \frac{\partial \rho\left(U^{(n)}\right)}{\partial U}$$

is not available. Approximation of $J_n$ by finite differences in the following way is expensive but feasible. The $i$th 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

$$U^{(n+1)} = U^{(n)} - J_n^{-1}\rho\big(U^{(n)}\big) = U^{(n)} - K^{-1}\big(KU^{(n)} - F\big) = K^{-1}F$$

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.

$$
\begin{aligned}
\frac{\partial (KU)_i}{\partial U_j} &= \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \sum_l \Bigg( \int_\Omega c\big(U + \varepsilon\phi_j\big)\nabla\phi_l\nabla\phi_i \; d\mathbf{x}\big(U_l + \varepsilon\delta_{l,j}\big) \\
&\qquad\qquad - \int_\Omega c(U)\nabla\phi_l\nabla\phi_i \; d\mathbf{x}U_l \Bigg) \\
&= \int_\Omega c(U)\nabla\phi_j\nabla\phi_i \; d\mathbf{x} + \sum_l \int_\Omega \phi_j \frac{\partial c}{\partial u}\nabla\phi_l\nabla\phi_i \; d\mathbf{x}U_l
\end{aligned}
$$

The first integral term is nothing more than $K_{i,j}$.

The second term is "lumped," i.e., replaced by a diagonal matrix that contains the row sums. Since $\Sigma_j\phi_j = 1$, the second term is approximated by

$$\delta_{i,j}\sum_l \int_\Omega \frac{\partial c}{\partial u}\nabla\phi_l\nabla\phi_i \; d\mathbf{x}U_l$$

which is the $i$th 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

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

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. 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 "Boundary Conditions by Writing Functions" on page 2-183.

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 2-196.

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.

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 |
| Ngen | positive integer | 10 | Maximum number of triangle generations |
| Mesh | p1, e1, t1 | initmesh | Initial mesh |
| Tripick | MATLAB function | pdeadworst | Triangle selection method |

| Property | Value | Default | Description |
|---|---|---|---|
| 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 $N_t$ columns, where $N_t$ is the current number of triangles. The number of rows in cc, aa, and ff are exactly the same as the input arguments c, a, and f. errf has one row for each equation in the system. There are two standard triangle selection methods—pdeadworst and pdeadgsc. pdeadworst selects triangles where errf exceeds a fraction (default: 0.5) of the worst value, and pdeadgsc selects triangles using a relative tolerance criterion.

The refinement method is either `longest` or `regular`. For details on the refinement method, see `refinemesh`.

The `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

### Adaptive Mesh Generation and Mesh Refinement

Solve the Laplace equation over a circle sector, with Dirichlet boundary conditions $u = \cos(2/3\text{atan2}(y, x))$ along the arc, and $u = 0$ along the straight lines, and compare to the exact solution. 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)
```
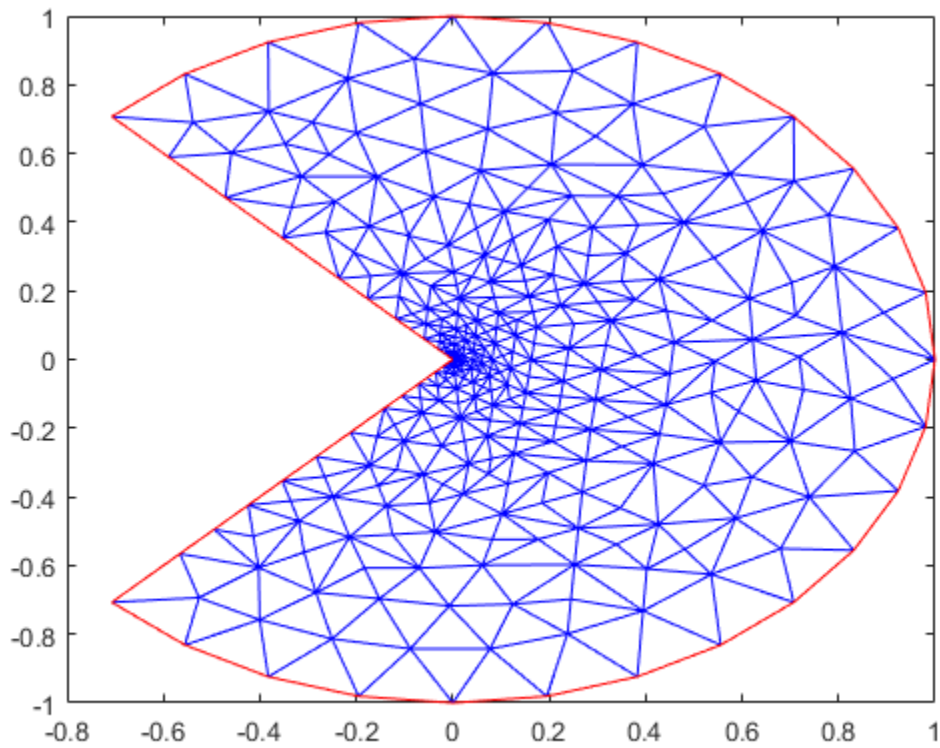
Test how many refinements you have to use with a uniform triangle net:

```
[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)
```

**6-7**
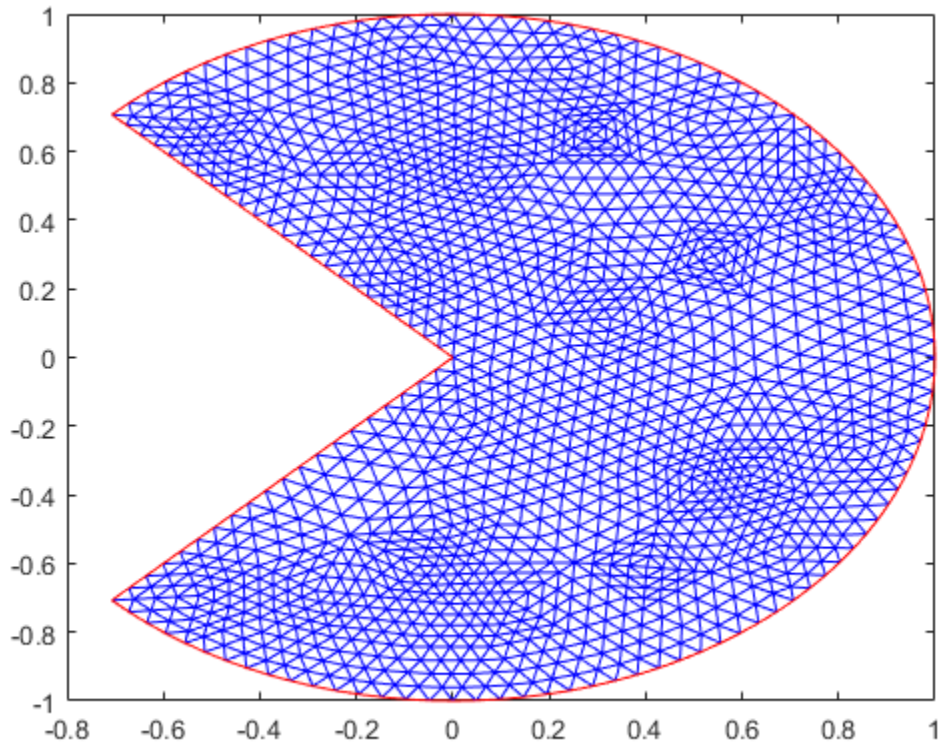
Uniform refinement with 3152 triangles produces an error of 0.0078. This error is over three times as large as that produced by the adaptive method (0.0028) with many fewer triangles (629). For a problem with regular solution, we expect an $O(h^2)$ error, but this solution is singular since $u \approx r^{1/3}$ 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`

**Introduced before R2006a**

# 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

## See Also
geometryFromEdges | PDEModel

## Topics
"Solve Problems Using PDEModel Objects" on page 2-5

**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,'dirichlet',RegionType,RegionID,
Name,Value)
applyBoundaryCondition(model,'neumann',RegionType,RegionID,
Name,Value)
applyBoundaryCondition(model,'mixed',RegionType,RegionID,Name,Value)
bc = applyBoundaryCondition( ___ )
```

## Description

`applyBoundaryCondition(model,'dirichlet',RegionType,RegionID,
Name,Value)` adds a Dirichlet boundary condition to `model`. The boundary condition applies to boundary regions of type `RegionType` with ID numbers in `RegionID`, and with arguments `r`, `h`, `u`, `EquationIndex` specified in the `Name,Value` pairs. For Dirichlet boundary conditions, specify either both arguments `r` and `h`, or the argument `u`. When specifying `u`, you can also use `EquationIndex`.

`applyBoundaryCondition(model,'neumann',RegionType,RegionID,
Name,Value)` adds a Neumann boundary condition to `model`. The boundary condition applies to boundary regions of type `RegionType` with ID numbers in `RegionID`, and with values `g` and `q` specified in the `Name,Value` pairs.

`applyBoundaryCondition(model,'mixed',RegionType,RegionID,Name,Value)` adds an individual boundary condition for each equation in a system of PDEs. The

boundary condition applies to boundary regions of type `RegionType` with ID numbers in `RegionID`, and with values specified in the `Name,Value` pairs. For mixed boundary conditions, you can use `Name,Value` pairs from both Dirichlet and Neumann boundary conditions as needed.

`bc = applyBoundaryCondition( ___ )` returns the boundary condition object.

# Examples

### Dirichlet Boundary Conditions

Create a PDE model and geometry.

```
model = createpde(1);
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
g = decsg(R1);
geometryFromEdges(model,g);
```

View the edge labels.

```
pdegplot(model,'EdgeLabels','on')
xlim([-1.2,1.2])
axis equal
```
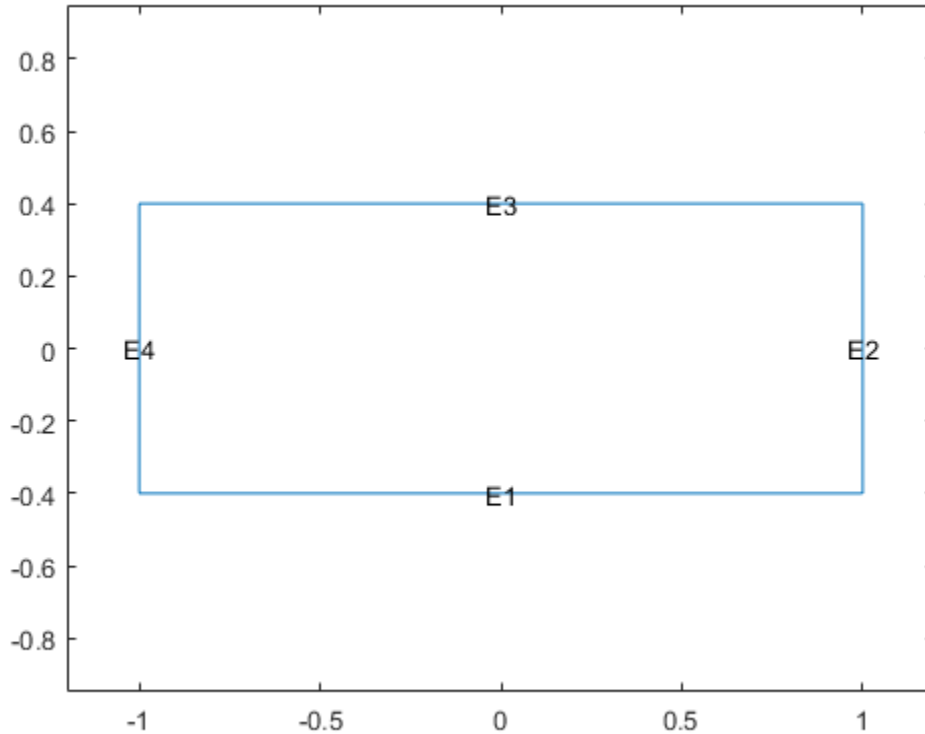
Apply zero Dirichlet condition on the edge 1.

```
applyBoundaryCondition(model,'dirichlet','Edge',1,'u',0);
```

On other edges, apply Dirichlet condition `h*u = r`, where `h = 1` and `r = 1`.

```
applyBoundaryCondition(model,'dirichlet','Edge',2:4,'r',1,'h',1);
```

### Neumann Boundary Conditions

Create a PDE model and geometry.

```
model = createpde(2);
R1 = [3,4,-1,1,1,-1,-.4,-.4,.4,.4]';
```

```
g = decsg(R1);
geometryFromEdges(model,g);
```

View the edge labels.

```
pdegplot(model,'EdgeLabels','on')
xlim([-1.2,1.2])
axis equal
```
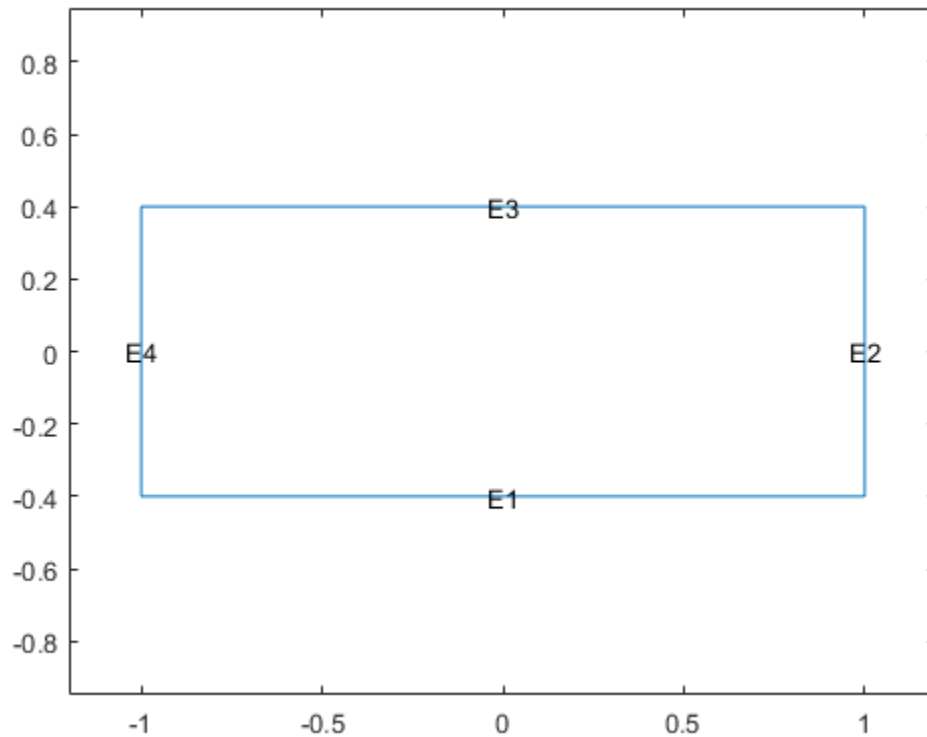


Apply the following Neumann boundary conditions on the edge 4.

```
applyBoundaryCondition(model,'neumann','Edge',4,'g',[0;.123],'q',[0;0;0;0]);
```

### Dirichlet and Neumann Boundary Conditions for Different Boundaries

Apply both types of boundary conditions to a scalar problem. First, create a PDE model and import a simple block geometry.

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

View the face labels.

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
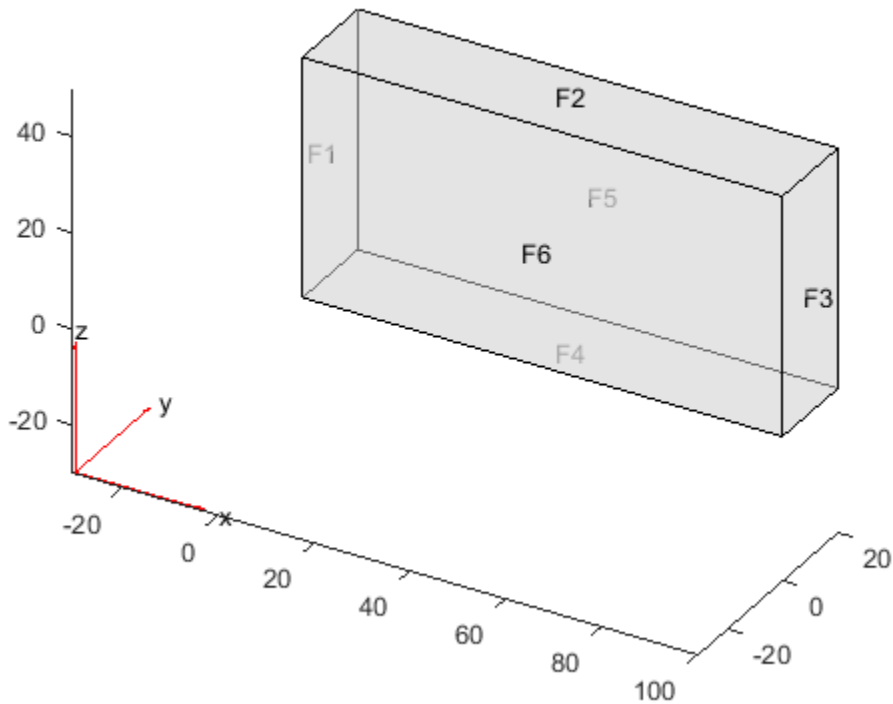
Set zero Dirichlet conditions on the narrow faces, which are labeled 1 through 4.

```
applyBoundaryCondition(model,'dirichlet','Face',1:4,'u',0);
```

Set Neumann boundary conditions with opposite signs on faces 5 and 6.

```
applyBoundaryCondition(model,'neumann','Face',5,'g',1);
applyBoundaryCondition(model,'neumann','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)
```
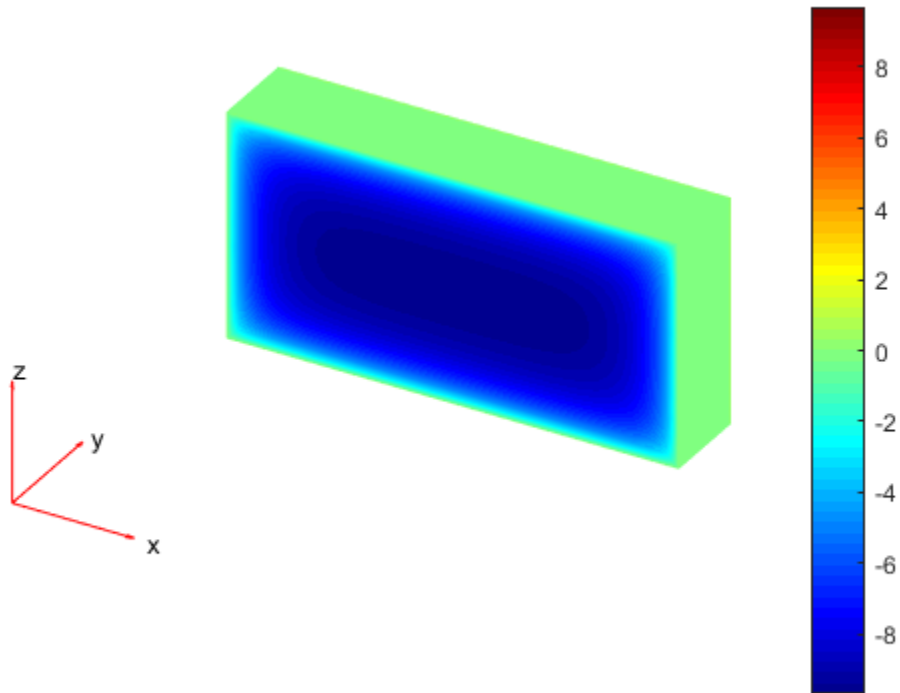
### Individual Boundary Conditions for Equations in a System

Create a PDE model and import a simple block geometry.

```
model = createpde(3);
importGeometry(model,'Block.stl');
```

View the face labels.

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
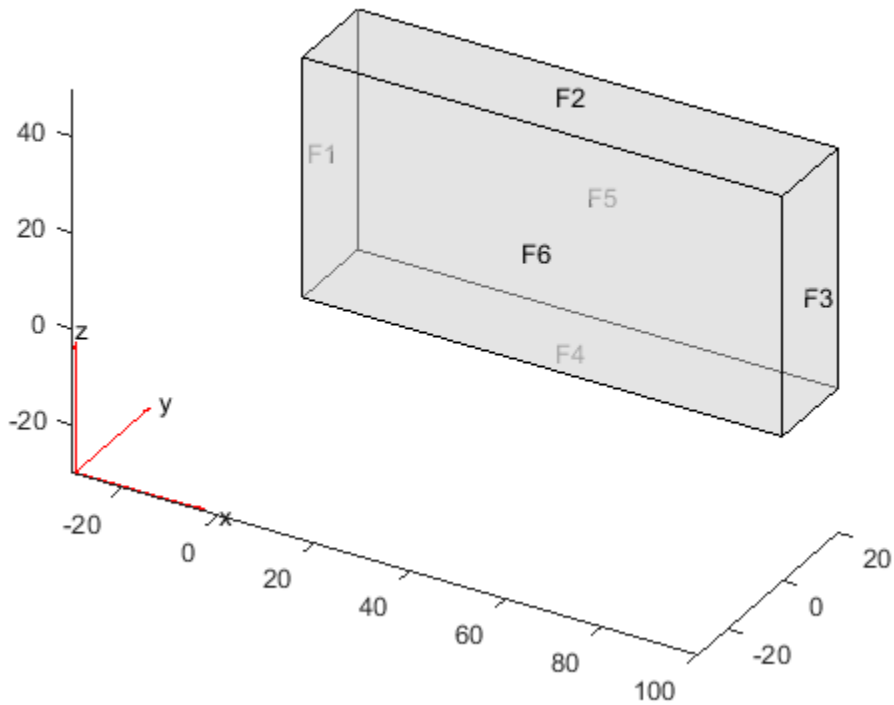
Set zero Dirichlet conditions on faces 1 and 2.

```
applyBoundaryCondition(model,'dirichlet','Face',1:2,'u',[0,0,0]);
```

Set Neumann boundary conditions with opposite signs on faces 4, 5, and 6.

```
applyBoundaryCondition(model,'neumann','Face',4:5,'g',[1;1;1]);
applyBoundaryCondition(model,'neumann','Face',6,'g',[-1;-1;-1]);
```

For face 3, apply generalized Neumann boundary condition for the first equation and Dirichlet boundary conditions for the second and third equations.

```
h = [0 0 0;0 1 0;0 0 1];
r = [0;3;3];
```

```
q = [1 0 0;0 0 0;0 0 0];
g = [10;0;0];
applyBoundaryCondition(model,'mixed','Face',3,'h',h,'r',r,'g',g,'q',q);
```

Solve an elliptic PDE with these boundary conditions, and plot the result.

```
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',[0;0;0]);
generateMesh(model);
results = solvepde(model);
u = results.NodalSolution;
pdeplot3D(model,'ColorMapData',u(:,1))
```
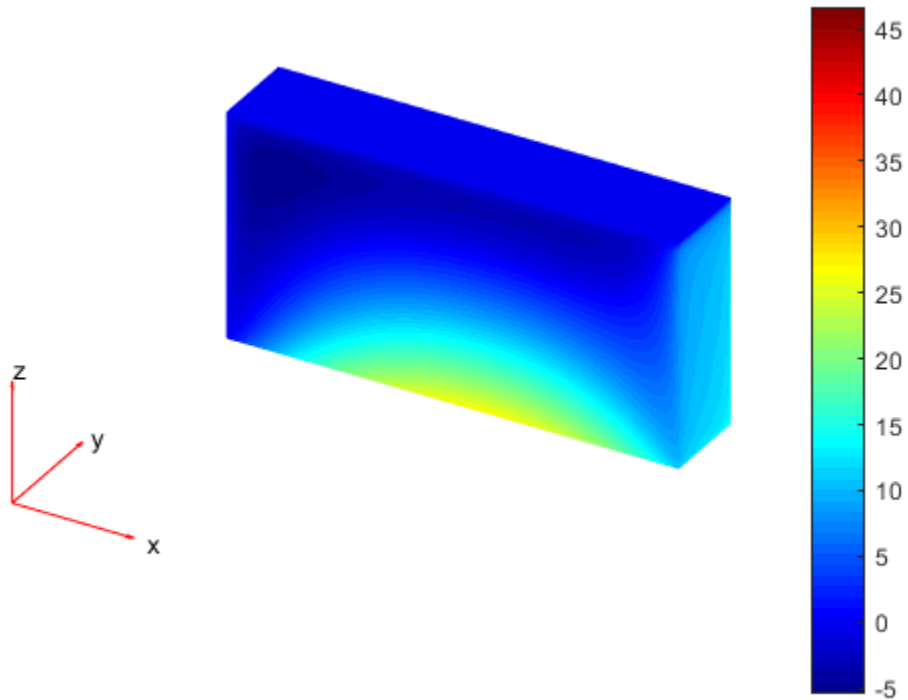
## Input Arguments

**`model`** — **PDE model**
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

**`RegionType`** — **Geometric region type**
'Face' for 3-D geometry | 'Edge' for 2-D geometry

Geometric region type, specified as `'Face'` for 3-D geometry or `'Edge'` for 2-D geometry.

Example: `applyBoundaryCondition(model,'dirichlet','Face',3,'u',0)`

Data Types: `char`

### `RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot` with the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) value set to `'on'`.

Example: `applyBoundaryCondition(model,'dirichlet','Face',3:6,'u',0)`

Data Types: `double`

## Name-Value Pair Arguments

Example: `applyBoundaryCondition(model,'dirichlet','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 a function handle. $N$ is the number of PDEs in the system. For the syntax of the function handle form of `r`, see "Nonconstant Boundary Conditions" on page 2-165.

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, a vector with $N^2$ elements, or a function handle. $N$ is the number of PDEs in the system. For the syntax of the function handle form of `h`, see "Nonconstant Boundary Conditions" on page 2-165.

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 a function handle. $N$ is the number of PDEs in the system. For scalar PDEs, the generalized Neumann condition is n·(c∇u) + qu = g. For the syntax of the function handle form of g, see "Nonconstant Boundary Conditions" on page 2-165.

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, a vector with $N$^2 elements, or a function handle. $N$ is the number of PDEs in the system. For the syntax of the function handle form of q, see "Nonconstant Boundary Conditions" on page 2-165.

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. If u has less than $N$ elements, then you must also use EquationIndex. The u and EquationIndex arguments must have the same length. If u has $N$ elements, then specifying EquationIndex is optional.

For the syntax of the function handle form of u, see "Nonconstant Boundary Conditions" on page 2-165.

Example: applyBoundaryCondition(model,'dirichlet','Face', [2,4,11],'u',0)

Data Types: double
Complex Number Support: Yes

### 'EquationIndex' — Index of the known u components

1:$N$ (default) | vector of integers with entries from 1 to $N$

Index of the known 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,'mixed','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" (MATLAB). For details of this evaluation, see "Nonconstant Boundary
Conditions" on page 2-165.

Example: applyBoundaryCondition(model,'dirichlet','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.

## Tips

· When there are multiple boundary condition assignments to the same geometric
  region, the toolbox uses the last applied setting.

· To avoid assigning boundary conditions to a wrong region, ensure that you are using
  the correct geometric region IDs by plotting and visually inspecting the geometry.

# See Also

## See Also
BoundaryCondition Properties | `findBoundaryConditions` | `PDEModel`

## Topics
"Solve Problems Using PDEModel Objects" on page 2-5

**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');
pdegplot(model,'FaceAlpha',0.5)
```



Create a mesh for the model.

```
generateMesh(model);
```
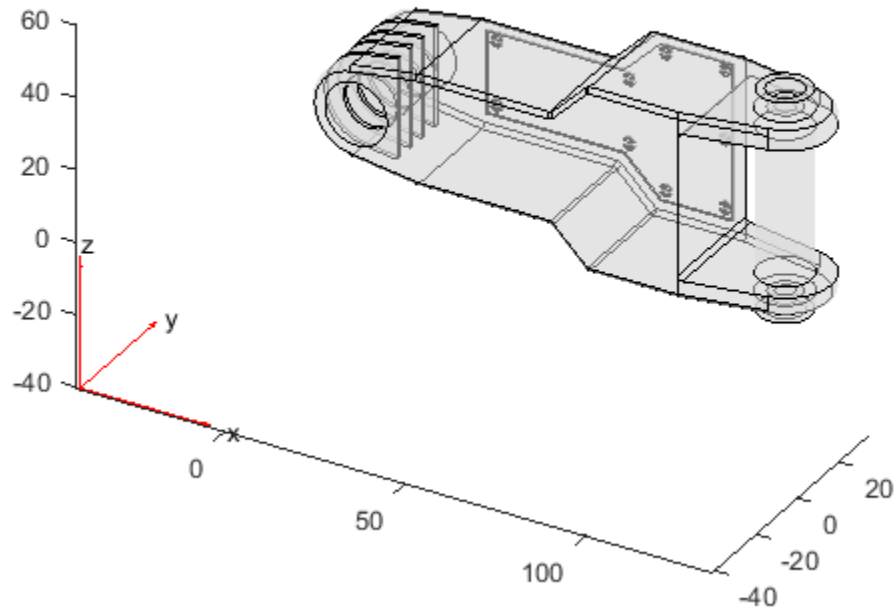
Create the finite element matrics from the mesh and the coefficients.

```
[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 2-110. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-83. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double | char | function_handle`
Complex Number Support: Yes

### p — Mesh points
matrix

Mesh points, specified as a 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 2-196.

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

Mesh triangles, specified as a 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 2-196.

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`

# 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

### 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;
```
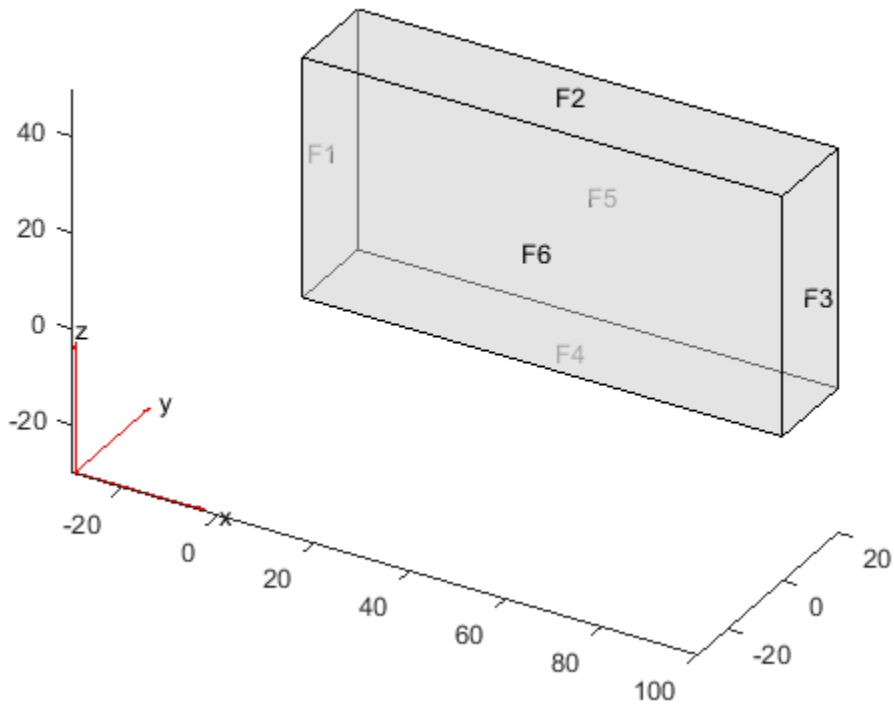
Set zero Dirichlet boundary conditions on the narrow faces (numbered 1 through 4).

```
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 file name.

- 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 2-154.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-183.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

### p — Mesh points
matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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`

### 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 6-37.

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 6-37.

Typically, you use R in a subsequent call to `assempde`.

# 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 6-62 section, and the linear algebra approach detailed in "Systems of PDEs" on page 5-13.

## See Also

### 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 =

  struct with fields:

    K: [150×150 double]
    A: [150×150 double]
    F: [150×1 double]
    Q: [150×150 double]
    G: [150×1 double]
    H: [40×150 double]
    R: [40×1 double]
    M: [150×150 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 =

  struct with fields:

    Kc: [110×110 double]
    Fc: [110×1 double]
     B: [150×110 double]
    ud: [150×1 double]
     M: [110×110 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 6-42.

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 6-42.

## 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.

# See Also

## See Also
solvepde

## Topics
"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

**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 6-62.

[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 6-62.

[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 6-62.

[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 6-62.

[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 6-62.

# 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)
```
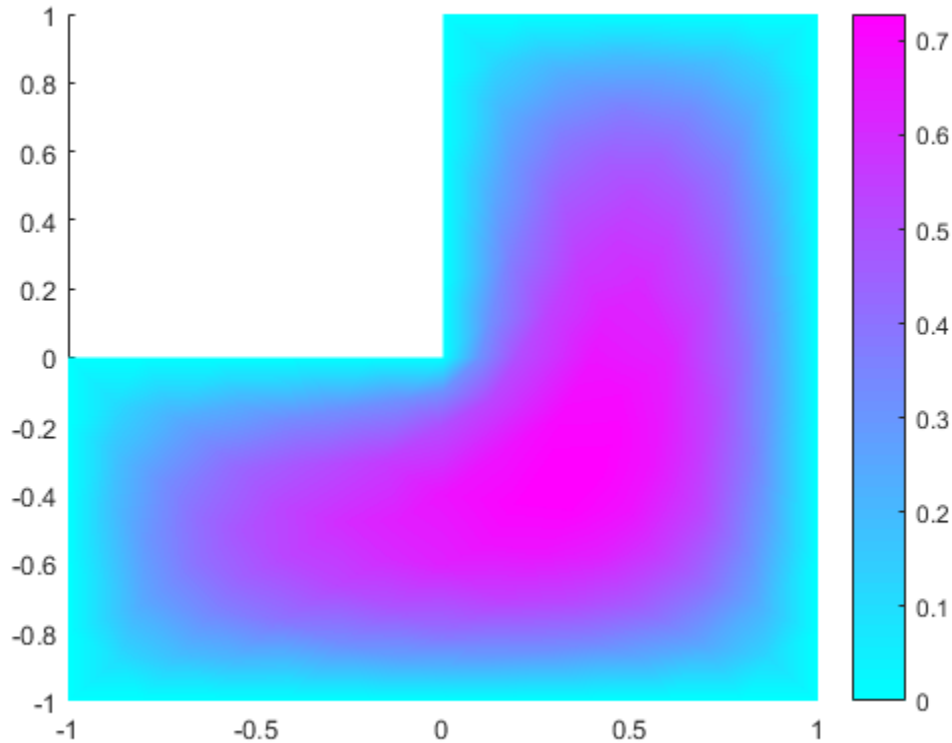
### 3-D Elliptic Problem

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');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
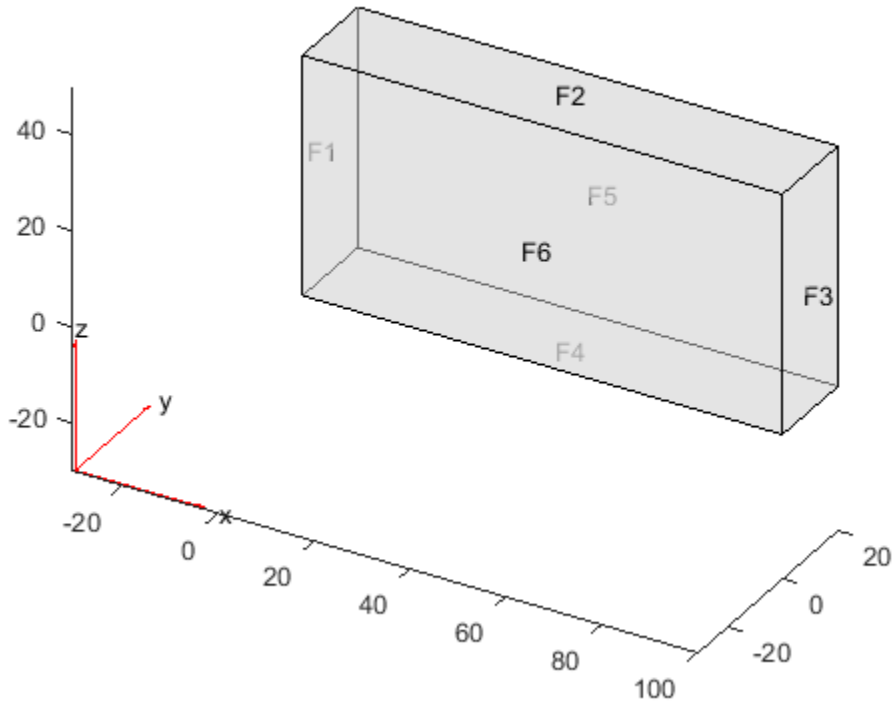
Set zero Dirichlet conditions on faces 1 through 4 (the edges). Set Neumann conditions with g = -1 on face 6 and g = 1 on face 5.

```
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)
```
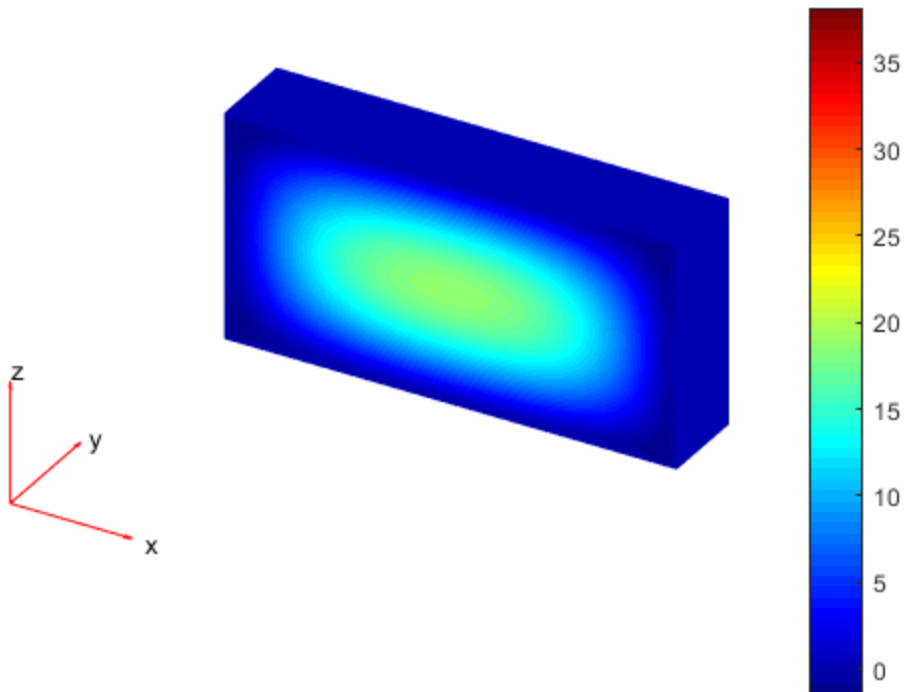


### 2-D PDE Using [p,e,t] Mesh

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)
```
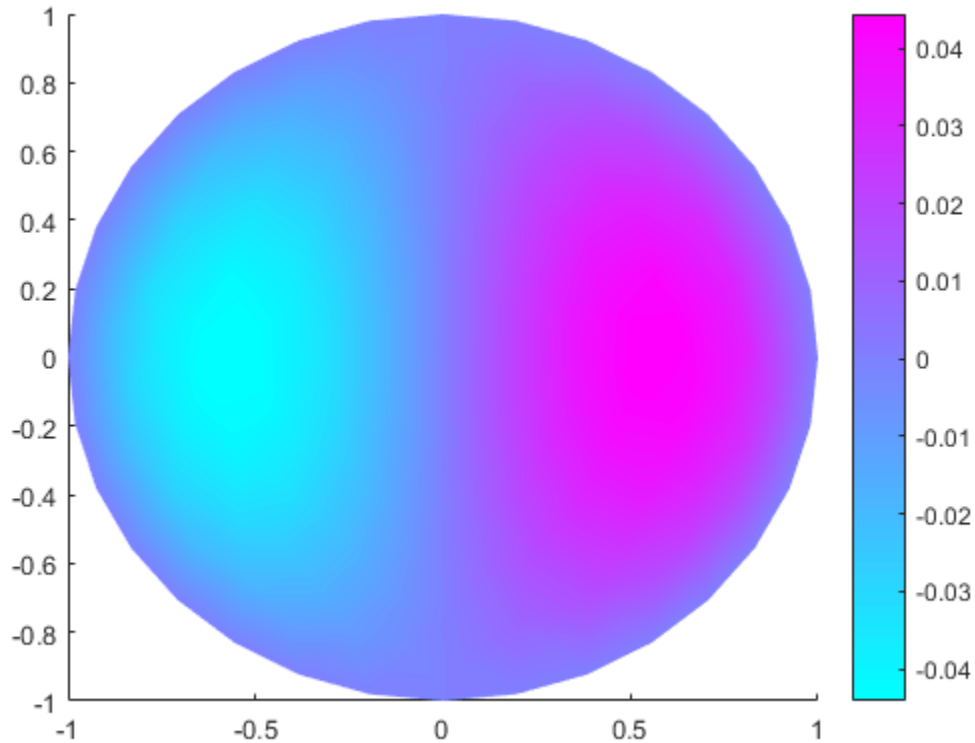
### Finite Element Matrices

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,'dirichlet','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 $f = \log\left(1 + x + \frac{y}{1+z}\right)$.

```
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 the next example.

### 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 $f = \log\left(1 + x + \frac{y}{1+z}\right)$.

```
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 the previous example.

### 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

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

You can specify c in various ways, detailed in "c Coefficient for Systems" on page 2-110. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-83. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 file name.

- 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 2-154.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-183.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

### p — Mesh points

matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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

Mesh triangles, specified as a 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 2-196.

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

### 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 6-62.

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 6-62.

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 3-210.

### 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 6-62.

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 6-62.

`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 6-62.

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 6-62.

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 6-62.

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 6-62.

Typically, you use R in a subsequent call to a solver such as `assempde` or `hyperbolic`.

# Definitions

## 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 6-62 and "Stiff-Spring Approximation" on page 6-62.

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

## 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 the 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

### BCType — Type of boundary condition
`'dirichlet'` | `'neumann'` | `'mixed'`

Boundary type, specified as `'dirichlet'`, `'neumann'`, or `'mixed'`.

Example: `applyBoundaryCondition(model,'dirichlet','Face',3,'u',0)`

Data Types: `char`

### RegionType — Geometric region type
`'Face'` for 3-D geometry | `'Edge'` for 2-D geometry

Geometric region type, specified as `'Face'` for 3-D geometry or `'Edge'` for 2-D geometry.

Example: `applyBoundaryCondition(model,'dirichlet','Face',3,'u',0)`

Data Types: `char`

### RegionID — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot` with the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) value set to `'on'`.

Example: `applyBoundaryCondition(model,'dirichlet','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 a function handle. $N$ is the number of PDEs in the system. For the syntax of the function handle form of `r`, see "Nonconstant Boundary Conditions" on page 2-165.

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, a vector with $N^2$ elements, or a function handle. $N$ is the number of PDEs in the system. For the syntax of the function handle form of `h`, see "Nonconstant Boundary Conditions" on page 2-165.

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 a function handle. $N$ is the number of PDEs in the system. For scalar PDEs, the generalized Neumann condition is `n·(c∇u) + qu = g`. For the syntax of the function handle form of `g`, see "Nonconstant Boundary Conditions" on page 2-165.

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 $n \cdot (c \times \nabla u) + qu = g$, 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. For the syntax of the function handle form of q, see "Nonconstant Boundary Conditions" on page 2-165.

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. If u has less than $N$ elements, then you must also use `EquationIndex`. The u and `EquationIndex` arguments must have the same length. If u has $N$ elements, then specifying `EquationIndex` is optional.

For the syntax of the function handle form of u, see "Nonconstant Boundary Conditions" on page 2-165.

Example: `applyBoundaryCondition(model,'dirichlet','Face', [2,4,11],'u',0)`

Data Types: `double`
Complex Number Support: Yes

### EquationIndex — Index of the known u components
`1:`$N$ (default) | vector of integers with entries from $1$ to $N$

Index of the known 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,'mixed','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" (MATLAB). For details of this evaluation, see "Nonconstant Boundary Conditions" on page 2-165.

Example: `applyBoundaryCondition(model,'dirichlet','Face', [2,4,11],'u',@ucalculator,'Vectorized','on')`

Data Types: `char`

# See Also

## See Also
`applyBoundaryCondition` | `findBoundaryConditions` | `PDEModel`

## Topics
"Specify Boundary Conditions" on page 2-160
"View, Edit, and Delete Boundary Conditions" on page 2-178
"Solve Problems Using PDEModel Objects" on page 2-5

**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

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

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 `'FaceLabels'` 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 2-128.

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 2-128.

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 2-89.

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 2-128.

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 2-86.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

## See Also

### See Also
`findCoefficients` | `specifyCoefficients`

### Topics
"PDE Coefficients"
"Solve Problems Using PDEModel Objects" on page 2-5

**Introduced in R2016a**

# createpde

Create PDE or thermal model

## Compatibility

Creating PDE models is the same for both the recommended and the legacy workflows.

## Syntax

```
model = createpde(N)
thermalmodel = createpde('thermal',AnalysisType)
```

## Description

`model = createpde(N)` returns a PDE model object for a system of N equations. A complete PDE model object contains a description of the problem you want to solve, including the geometry, mesh, boundary conditions, etc.

`thermalmodel = createpde('thermal',AnalysisType)` returns a thermal model for the specified analysis type.

## 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: []
     InitialConditions: []
                  Mesh: []
         SolverOptions: [1×1 PDESolverOptions]
```

### Create a Scalar PDE Model

Create a model for a single (scalar) PDE.

```
model = createpde
```

```
model =

  PDEModel with properties:

          PDESystemSize: 1
        IsTimeDependent: O
               Geometry: []
   EquationCoefficients: []
     BoundaryConditions: []
      InitialConditions: []
                   Mesh: []
          SolverOptions: [1×1 PDESolverOptions]
```

### Create a Steady-State Thermal Model

Create a model for a steady-state thermal problem.

```
thermalmodel = createpde('thermal')
```

```
thermalmodel =

  ThermalModel with properties:

              AnalysisType: 'steadystate'
                  Geometry: []
        MaterialProperties: []
               HeatSources: []
```

```
      StefanBoltzmannConstant: []
           BoundaryConditions: []
            InitialConditions: []
                         Mesh: []
                SolverOptions: [1×1 PDESolverOptions]
```

### Create a Transient Thermal Model

Create a model for a transient thermal problem.

```
thermalmodel = createpde('thermal','transient')
```

```
thermalmodel =

  ThermalModel with properties:

                   AnalysisType: 'transient'
                       Geometry: []
             MaterialProperties: []
                    HeatSources: []
        StefanBoltzmannConstant: []
             BoundaryConditions: []
              InitialConditions: []
                           Mesh: []
                  SolverOptions: [1×1 PDESolverOptions]
```

- "Solve Problems Using PDEModel Objects" on page 2-5
- "Equations You Can Solve Using Recommended Functions" on page 1-6
- "Heat Transfer"

## Input Arguments

### N — Number of equations

1 (default) | positive integer

Number of equations, specified as a positive integer. You do not need to specify N for a model where N = 1.

Example: model = createpde

Example: `model = createpde(3);`

Data Types: `double`

**AnalysisType — Type of thermal analysis**
`'steadystate'` (default) | `'transient'`

Type of thermal analysis, specified as `'steadystate'` or `'transient'`.

- `'steadystate'` creates a steady-state thermal model.
- `'transient'` creates a transient thermal model.

You do not need to specify `AnalysisType` for a steady-state model.

Example: `model = createpde('thermal')`

Example: `model = createpde('thermal','transient');`

Data Types: `double`

## Output Arguments

**model — PDE model container**
`PDEModel` object

PDE model container, returned as a `PDEModel` object.

Example: `model = createpde(2)`

**thermalmodel — Thermal model container**
`ThermalModel` object

Thermal model container, returned as a `ThermalModel` object.

Example: `thermalmodel = createpde('thermal')`

## See Also

### See Also
`PDEModel`

## Topics

**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
pdegplot(model,'FaceLabels','on')
view(30,30)
title('Bracket with Face Labels')
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```

**Bracket with Face Labels**
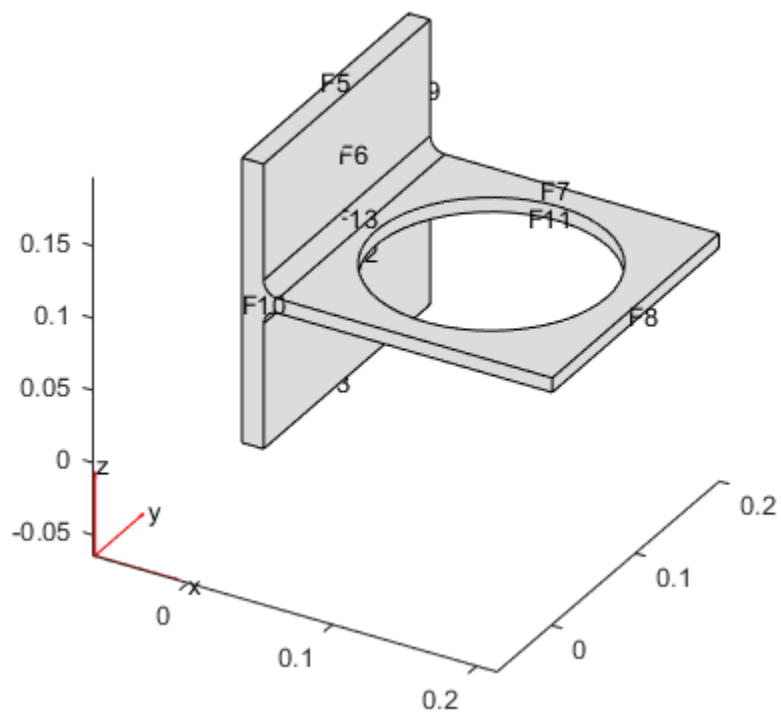
**Bracket with Face Labels, Rear View**



Set boundary conditions: face 3 is immobile, and there is a force in the negative z direction on face 6.

```
applyBoundaryCondition(model,'dirichlet','face',4,'u',[0,0,0]);
applyBoundaryCondition(model,'neumann','face',8,'g',[0,0,-1e4]);
```

Set coefficients that represent the equations of linear elasticity.

```
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: [14036×3 double]
        XGradients: [14036×3 double]
        YGradients: [14036×3 double]
        ZGradients: [14036×3 double]
             Mesh: [1×1 FEMesh]
```

Plot the solution for the z-component, which is component 3.

```
pdeplot3D(model,'ColorMapData',results.NodalSolution(:,3))
```
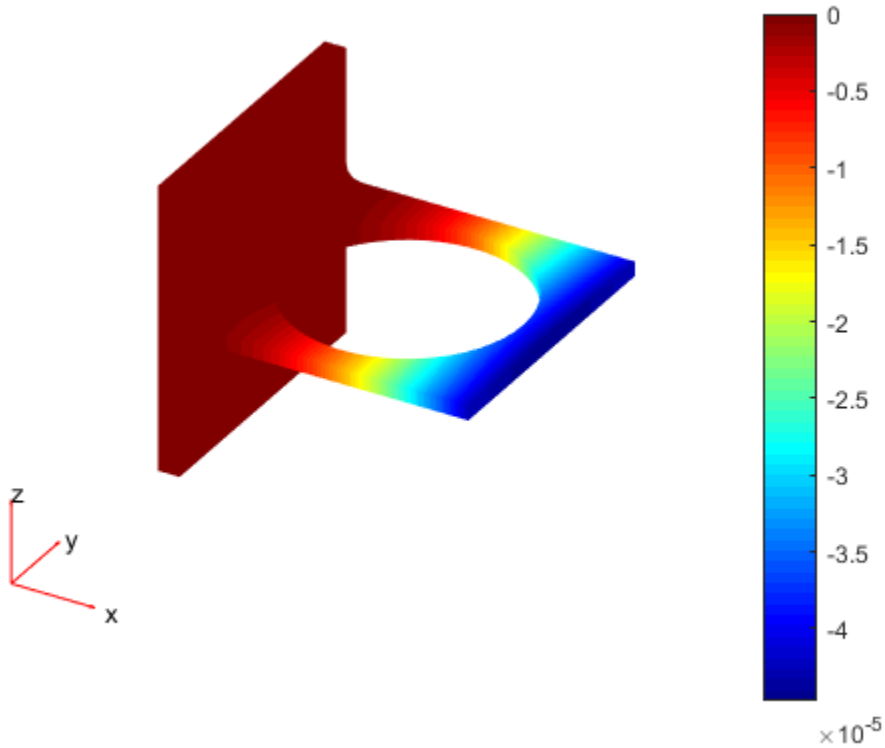
### Results from a Time-Dependent Problem

Obtain a solution from a parabolic problem.

The problem models heat flow in a solid.

```
model = createpde();
importGeometry(model,'Tetrahedron.stl');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
view(45,45)
```
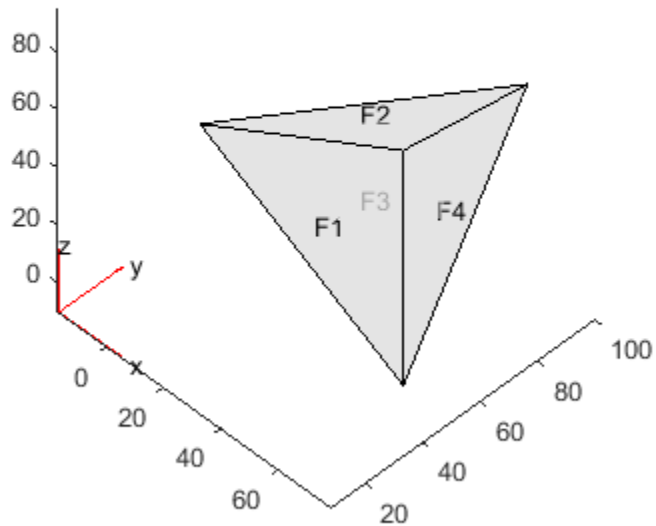
Set the temperature on face 2 to 100. Leave the other boundary conditions at their default values (insulating).

```
applyBoundaryCondition(model,'dirichlet','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))
```
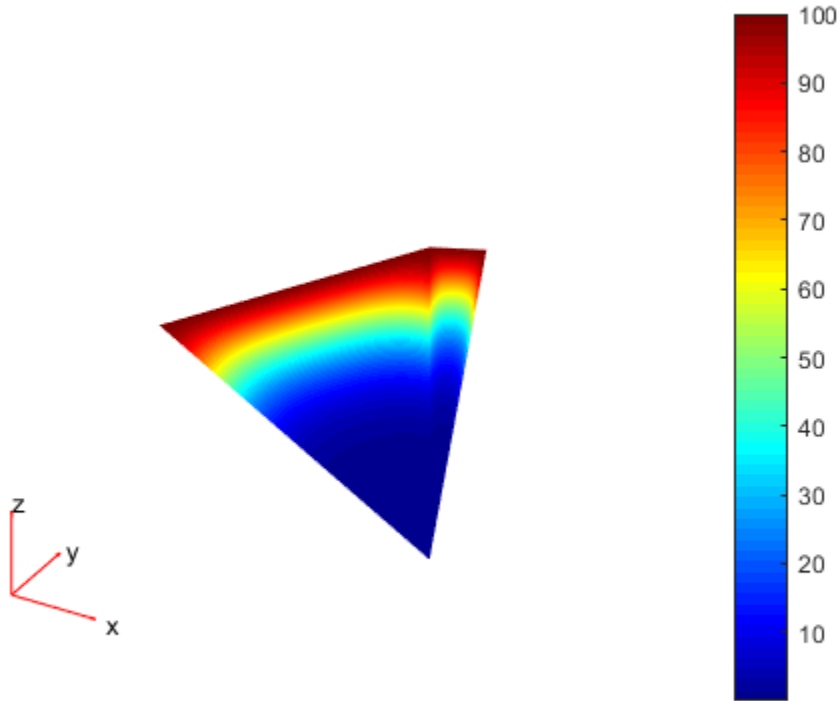
### Results from an Eigenvalue Problem

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,'dirichlet','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=   0.09,  New conv eig=  0
                 Basis= 13,  Time=   0.09,  New conv eig=  0
                 Basis= 16,  Time=   0.16,  New conv eig=  0
                 Basis= 19,  Time=   0.16,  New conv eig=  1
                 Basis= 22,  Time=   0.17,  New conv eig=  2
                 Basis= 25,  Time=   0.17,  New conv eig=  3
                 Basis= 28,  Time=   0.17,  New conv eig=  3
                 Basis= 31,  Time=   0.28,  New conv eig=  5
                 Basis= 34,  Time=   0.39,  New conv eig=  5
                 Basis= 37,  Time=   0.59,  New conv eig=  7
                 Basis= 40,  Time=   0.83,  New conv eig=  7
                 Basis= 43,  Time=   0.97,  New conv eig= 10
                 Basis= 46,  Time=   1.03,  New conv eig= 10
                 Basis= 49,  Time=   1.19,  New conv eig= 11
                 Basis= 52,  Time=   1.19,  New conv eig= 13
                 Basis= 55,  Time=   1.33,  New conv eig= 14
                 Basis= 58,  Time=   1.47,  New conv eig= 14
                 Basis= 61,  Time=   1.61,  New conv eig= 16
                 Basis= 64,  Time=   1.76,  New conv eig= 16
                 Basis= 67,  Time=   1.93,  New conv eig= 18
                 Basis= 70,  Time=   2.06,  New conv eig= 21
End of sweep: Basis= 70,  Time=   2.06,  New conv eig= 21
                 Basis= 31,  Time=   2.18,  New conv eig=  0
                 Basis= 34,  Time=   2.31,  New conv eig=  0
                 Basis= 37,  Time=   2.37,  New conv eig=  0
                 Basis= 40,  Time=   2.42,  New conv eig=  0
End of sweep: Basis= 40,  Time=   2.42,  New conv eig=  0
```

Create an `EigenResults` object from the solution.

```
results = createPDEResults(model,eigenvectors,eigenvalues,'eigen')
```

```
results =

  EigenResults with properties:

    Eigenvectors: [2124×19 double]
```

```
Eigenvalues: [19×1 double]
       Mesh: [1×1 FEMesh]
```

Plot the solution for mode 10.

```
pdeplot(model,'XYData',results.Eigenvectors(:,10))
```
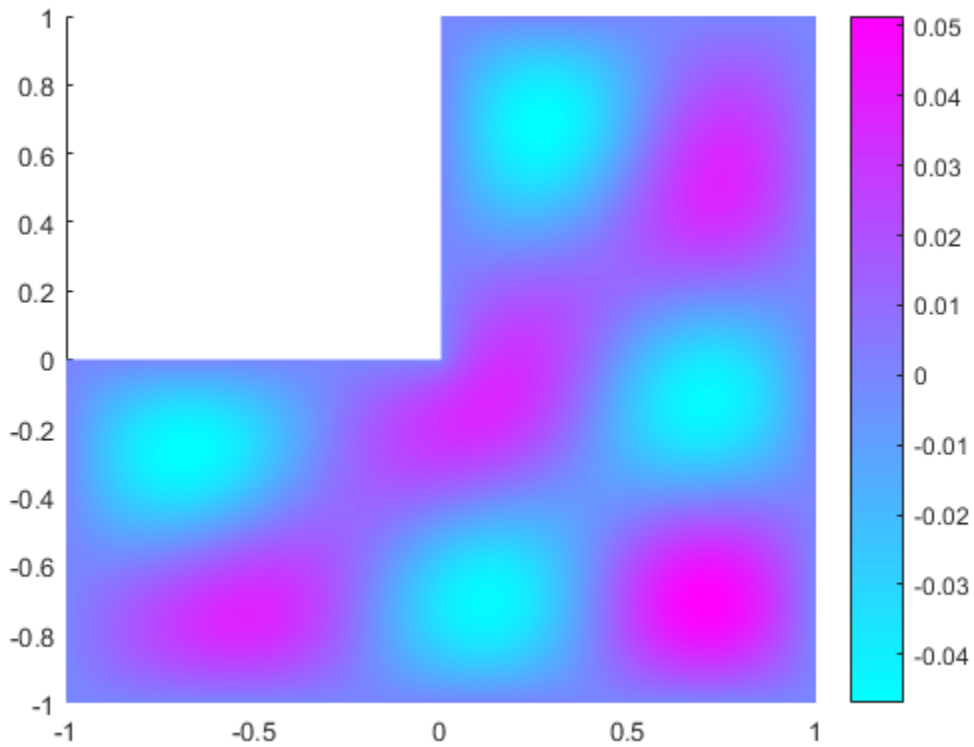


## Input Arguments

**`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)

## Tips

- Dimensions of the returned solutions and gradients are the same as those returned by solvepde and solvepdeeig. For details, see "Dimensions of Solutions, Gradients, and Fluxes" on page 3-232.

## 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.
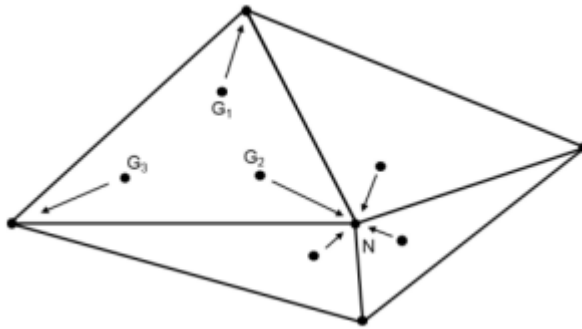
## See Also

### See Also
EigenResults | evaluateGradient | interpolateSolution |
StationaryResults | TimeDependentResults

### Topics
"3-D Linear Elasticity Equations in Toolbox Form" on page 3-72

**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

**Introduced before R2006a**

# 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

**Introduced before R2006a**

# 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( ___ )
```

## Description

`dl = decsg(gd,sf,ns)` decomposes the geometry description matrix `gd` into the geometry matrix `dl` and returns the minimal regions that satisfy the set formula `sf`. The name-space matrix `ns` is a text matrix that relates the columns in `gd` to variable names in `sf`.

Typically, you draw a geometry in the PDE app, then export it to the MATLAB Command Window by selecting **Export Geometry Description, Set Formula, Labels** from the **Draw** menu in the app. The resulting geometry description matrix `gd` represents the CSG model. `decsg` analyzes the model and constructs a set of disjointed minimal regions bounded by boundary segments and border segments. This set of minimal regions constitutes the *decomposed geometry* and allows other Partial Differential Equation Toolbox functions to work with the geometry.

Alternatively, you can use the `decsg` function when creating a geometry without using the app. See "Create CSG Geometry at the Command Line" on page 2-9 for details.

To return all minimal regions (`sf` corresponds to the union of all shapes in `gd`), use the shorter syntax `dl = decsg(gd)`.

[dl,bt] = decsg( ___ ) returns a Boolean table (matrix) that relates the original shapes to the minimal regions. A column in bt corresponds to the column with the same index in gd. A row in bt corresponds to the index of a minimal region. You can use bt to remove boundaries between subdomains.

# Examples

### Decompose Geometry Created in PDE App

Create a 2-D geometry in the PDE app, then export it to the MATLAB workspace and decompose it to minimal regions by using decsg.

Start the PDE app and draw a unit circle and a unit square.

```
pdecirc(0,0,1)
pderect([0 1 0 1])
```

Enter C1-SQ1 in the **Set formula** field.

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.

Decompose the exported geometry into minimal regions. The result is one minimal region with five edge segments: three circle edge segments and two line edge segments.

```
dl = 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
```

View the geometry. Display the edge labels and the subdomain labels.

```
pdegplot(dl,'EdgeLabels','on','SubdomainLabels','on')
axis equal
```
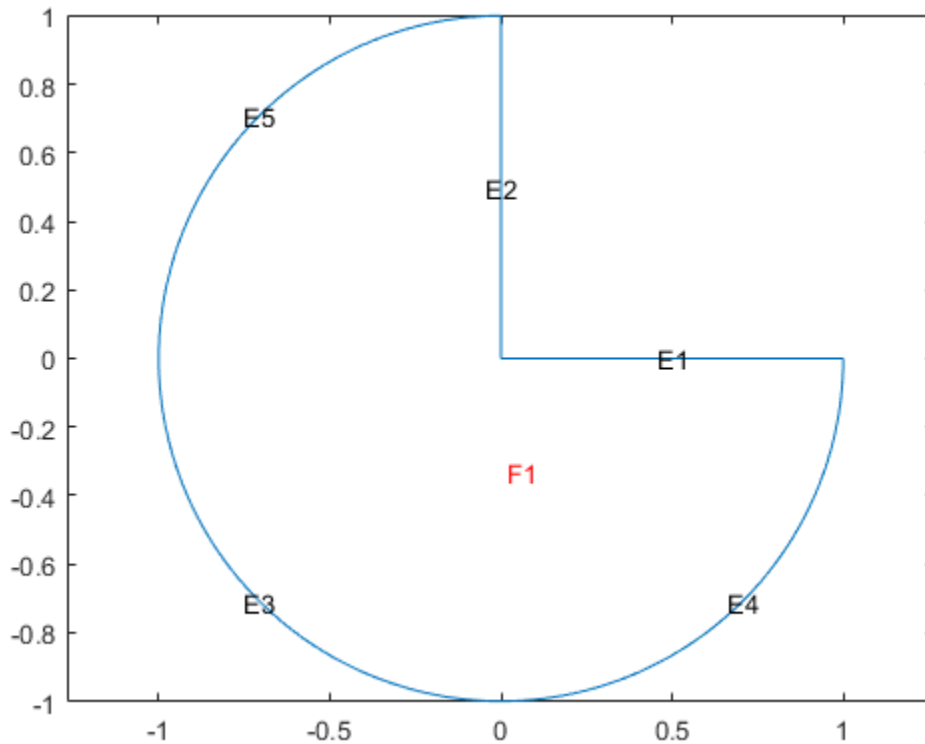


For comparison, decompose the same geometry without specifying the set formula `sf` and the name-space matrix `ns`. This syntax returns the union of all shapes in the geometry `gd`.

```
dl_all = decsg(gd)

dl_all =
    2.0000    2.0000    2.0000    2.0000    1.0000    1.0000    1.0000    1.0000
         0    1.0000    1.0000         0   -1.0000    0.0000    1.0000    0.0000
    1.0000    1.0000         0         0    0.0000    1.0000    0.0000   -1.0000
         0         0    1.0000    1.0000   -0.0000   -1.0000         0    1.0000
         0    1.0000    1.0000         0   -1.0000         0    1.0000   -0.0000
    3.0000    2.0000    2.0000    3.0000    1.0000    1.0000    3.0000    1.0000
```

| 1.0000 | 0 | 0 | 1.0000 | 0 | 0 | 2.0000 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1.0000 | 1.0000 | 1.0000 | 1.0000 |

View the resulting geometry.

```
pdegplot(dl_all,'EdgeLabels','on','SubdomainLabels','on')
axis equal
```
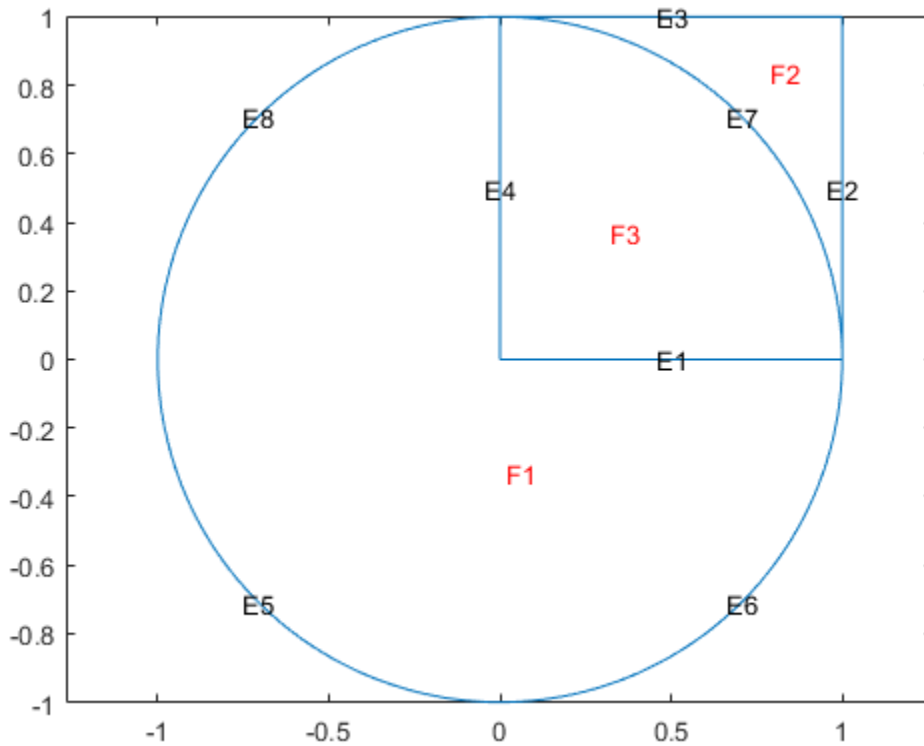


### Remove Boundaries Between Subdomains

Start the PDE app and draw a unit circle and a unit square.

```
pdecirc(0,0,1)
pderect([0 1 0 1])
```

Enter `C1+SQ1` in the **Set formula** field.

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.

Decompose the exported geometry into minimal regions. Because the geometry is a union of all regions, `C1+SQ1`, you can omit the arguments specifying the set formula and name-space matrix when using `decsg`.

```
[dl,bt] = decsg(gd)

dl =
    2.0000    2.0000    2.0000    2.0000    1.0000    1.0000    1.0000    1.0000
         0    1.0000    1.0000         0   -1.0000    0.0000    1.0000    0.0000
    1.0000    1.0000         0         0    0.0000    1.0000    0.0000   -1.0000
         0         0    1.0000    1.0000   -0.0000   -1.0000         0    1.0000
         0    1.0000    1.0000         0   -1.0000         0    1.0000   -0.0000
    3.0000    2.0000    2.0000    3.0000    1.0000    1.0000    3.0000    1.0000
    1.0000         0         0    1.0000         0         0    2.0000         0
         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0
         0         0         0         0    1.0000    1.0000    1.0000    1.0000

bt =
     1     0
     0     1
     1     1
```

View the geometry. Display the edge labels and the subdomain labels.

```
pdegplot(dl,'EdgeLabels','on','SubdomainLabels','on')
axis equal
```
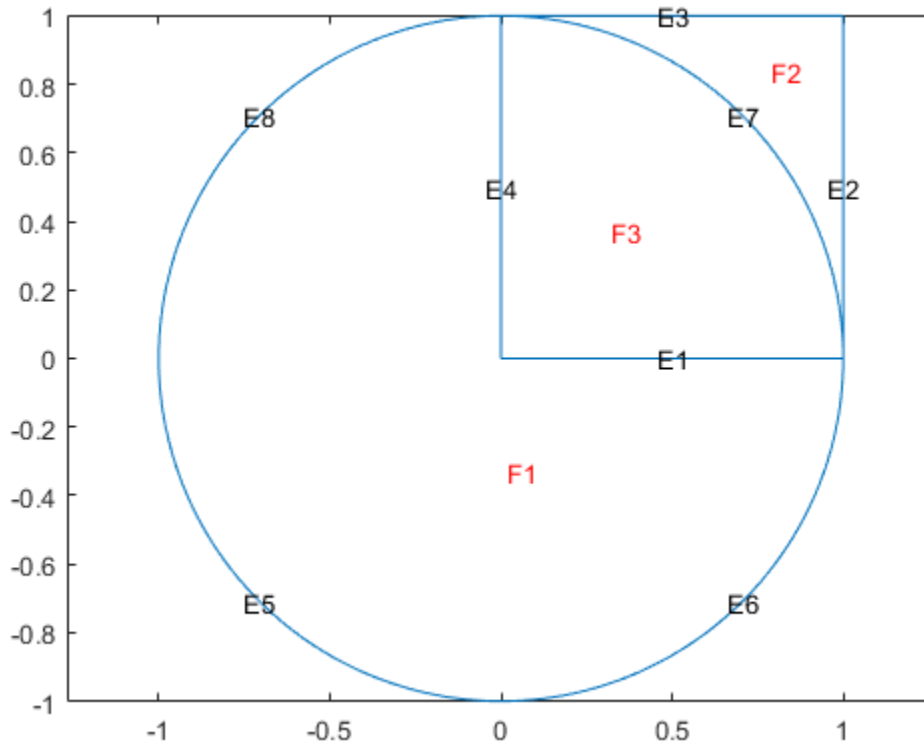
Remove the subdomain boundaries by using the `csgdel` function.

```
[dl2,bt2] = csgdel(dl,bt);
```

View the resulting geometry.

```
figure
pdegplot(dl2,'EdgeLabels','on','SubdomainLabels','on')
axis equal
```
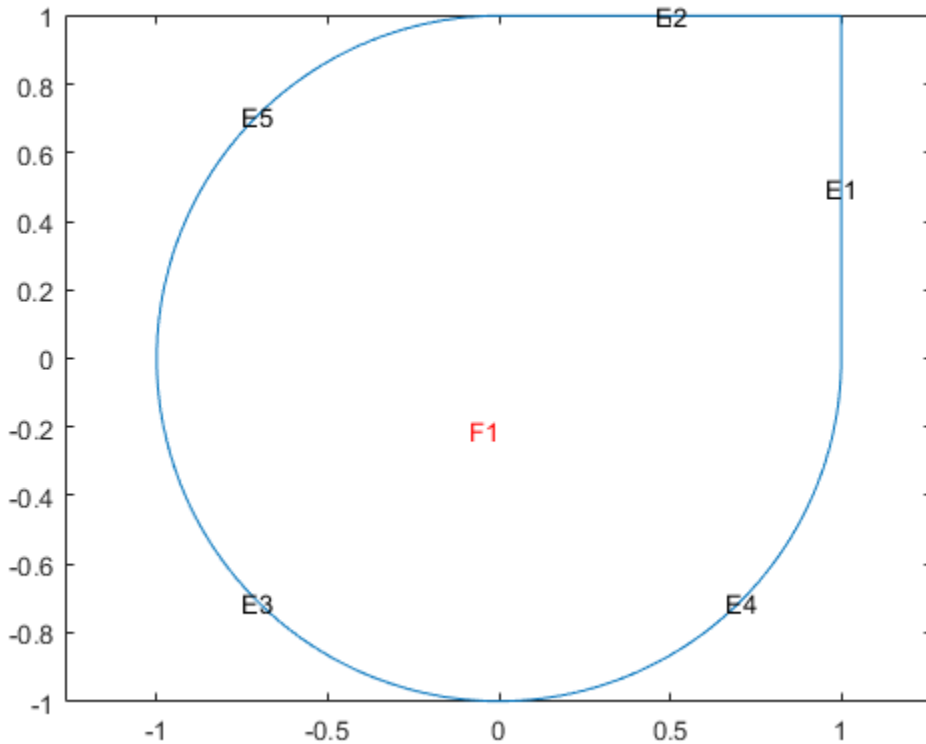
- "Create CSG Geometry at the Command Line" on page 2-9
- "Create 2-D Geometry" on page 2-7

## Input Arguments

### gd — Geometry description matrix
matrix of double-precision numbers

Geometry description matrix, specified as a matrix of double-precision numbers. The number of columns corresponds to the number of shapes used to construct the geometry. Each column in the geometry description matrix corresponds to a shape in the CSG model. The model supports four types of shapes:

- For a circle, the first row contains 1. The second and third rows contain the $x$- and $y$-coordinates of the center. The fourth row contains the radius of the circle.

- For a polygon, the first row contains 2. The second row contains *n*, which is the number of line segments in the boundary of the polygon. The next *n* rows contain the *x*-coordinates of the starting points of the edges, and the *n* rows after that contain the *y*-coordinates of the starting points of the edges.

- For a rectangle, the first row contains 3, and the second row contains 4. The next four rows contain the *x*-coordinates of the starting points of the edges, and the four rows after that contain the *y*-coordinates of the starting points of the edges.

- For an ellipse, the first row contains 4. The second and third rows contain the *x*- and *y*-coordinates of the center. The fourth and fifth rows contain the semiaxes of the ellipse. The sixth row contains the rotational angle of the ellipse, measured in radians.

All shapes in a geometry description matrix have the same number of rows. Rows that are not required for a particular shape are filled with zeros.

When you export geometry from the PDE app by selecting **Export Geometry Description, Set Formula, Labels** from the **Draw** menu in the app, you can use any variable name for the exported geometry description matrix in the MATLAB workspace. The default name is gd.

Data Types: `double`

### sf — Set formula
set of characters, including names of shapes and operators +, *, and -

Set formula, specified as a set of characters. The set of characters includes the names of shapes, such as C1, SQ2, E3, and the operators +, *, and - corresponding to the set operations union, intersection, and set difference, respectively. The operators + and * have the same precedence. The operator - has a higher precedence. You can control the precedence by using parentheses.

When you export geometry from the PDE app by selecting **Export Geometry Description, Set Formula, Labels** from the **Draw** menu in the app, you can use any variable name for the formula in the MATLAB workspace. The default name is sf.

Example: `'(SQ1+C1)-C2'`

Data Types: `char`

### ns — Name-space matrix
matrix of double-precision numbers

Name-space matrix, specified as a matrix of double-precision numbers. The number of columns corresponds to the number of shapes used to construct the geometry. Each column in `ns` contains a sequence of characters padded with spaces. Each character column assigns a name to the corresponding geometric object in `gd`, so you can refer to a specific object in `gd` in the set formula `sf`.

When you export geometry from the PDE app by selecting **Export Geometry Description, Set Formula, Labels** from the **Draw** menu in the app, you can use any variable name for the name-space matrix in the MATLAB workspace. The default name is `ns`.

Data Types: `double`

## Output Arguments

### `dl` — Decomposed geometry matrix
matrix of double-precision numbers

Decomposed geometry matrix, returned as a matrix of double-precision numbers. It contains a representation of the decomposed geometry in terms of disjointed minimal regions constructed by the `decsg` algorithm. Each edge segment of the minimal regions corresponds to a column in `dl`. Edge segments between minimal regions are *border segments*. Outer boundaries are *boundary segments*. In each column, the second and third rows contain the starting and ending *x*-coordinates. The fourth and fifth rows contain the corresponding *y*-coordinates. The sixth and seventh rows contain left and right minimal region labels with respect to the direction induced by the start and end points (counterclockwise direction on circle and ellipse segments). There are three types of possible edge segments in a minimal region:

- For circle edge segments, the first row is 1. The eighth and ninth rows contain the coordinates of the center of the circle. The 10th row contains the radius.
- For line edge segments, the first row is 2.
- For ellipse edge segments, the first row is 4. The eighth and ninth rows contain the coordinates of the center of the ellipse. The 10th and 11th rows contain the semiaxes of the ellipse. The 12th row contains the rotational angle of the ellipse.

All shapes in a decomposed geometry matrix have the same number of rows. Rows that are not required for a particular shape are filled with zeros.

| Row number | Circle edge segment | Line edge segment | Ellipse edge segment |
|---|---|---|---|
| 1 | 1 | 2 | 4 |
| 2 | starting $x$-coordinate | starting $x$-coordinate | starting $x$-coordinate |
| 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 | left minimal region label | left minimal region label | left minimal region label |
| 7 | right minimal region label | right minimal region label | right minimal region label |
| 8 | $x$-coordinate of the center | | $x$-coordinate of the center |
| 9 | $y$-coordinate of the center | | $y$-coordinate of the center |
| 10 | radius of the circle | | $x$-semiaxis before rotation |
| 11 | | | $y$-semiaxis before rotation |
| 12 | | | Angle in radians between $x$-axis and first semiaxis |

Data Types: `double`

### bt — Boolean table relating original shapes to minimal regions
matrix of 1s and 0s

Boolean table relating the original shapes to the minimal regions, returned as a matrix of 1s and 0s.

Data Types: `double`

## Limitations

- In rare cases `decsg` can error or create an invalid geometry because of the limitations of its algorithm. Such issues can occur when two or more edges of a geometry partially overlap, almost coincide, or are almost tangent.

## Tips

- `decsg` does not check the input CSG model for correctness. It assumes 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.

- `decsg` returns `NaN` if it cannot evaluate the set formula `sf`.

## See Also

### See Also
`csgchk` | `csgdel` | `geometryFromEdges` | `pdecirc` | `pdeellip` | `pdepoly` | `pderect` | `pdetool` | `wgeom`

### Topics
"Create CSG Geometry at the Command Line" on page 2-9
"Create 2-D Geometry" on page 2-7

**Introduced before R2006a**

# 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.

You can use the `importGeometry` function or the `geometryFromMesh` function to specify a 3-D geometry for your model.

You also can use the `multicuboid`, `multicylinder`, or `multisphere` functions to create a geometry that you can later assign to your model. These functions let you create multi-cellular geometries for modeling subdomains or single domains of cubic, cylindric, or spheric shapes, respectively. All cells must be of the same type: you cannot combine cuboids, cylinders, and spheres in one geometry. To add a multi-cellilar geometry to a `PDEModel`, assign the `DiscreteGeometry` to the `Geometry` property of the model. For example, create a PDE model and add the following geometry formed by three spheres to the model.

```
model = createpde;
gm = multisphere([1,2,3]);
model.Geometry = gm;
```

## Properties

### `NumCells` — Number of geometry cells
positive integer

Number of geometry cells, returned as a positive integer.

Data Types: `double`

**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

## See Also
geometryFromMesh | importGeometry | multicuboid | multicylinder | multisphere | PDEModel

## Topics
"Solve Problems Using PDEModel Objects" on page 2-5

**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:

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

`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:

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

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.

**Introduced before R2006a**

# EigenResults

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.

## 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`.

## 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, Gradients, and Fluxes" on page 3-232.

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

## 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,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
```

Specify 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=    0.03,   New conv eig=   0
                Basis= 13,   Time=    0.03,   New conv eig=   0
                Basis= 16,   Time=    0.03,   New conv eig=   0
                Basis= 19,   Time=    0.09,   New conv eig=   3
                Basis= 22,   Time=    0.09,   New conv eig=   3
                Basis= 25,   Time=    0.09,   New conv eig=   5
                Basis= 28,   Time=    0.09,   New conv eig=   6
                Basis= 31,   Time=    0.09,   New conv eig=   8
                Basis= 34,   Time=    0.23,   New conv eig=  12
                Basis= 37,   Time=    0.23,   New conv eig=  14
                Basis= 40,   Time=    0.23,   New conv eig=  14
                Basis= 43,   Time=    0.33,   New conv eig=  15
                Basis= 46,   Time=    0.33,   New conv eig=  16
                Basis= 49,   Time=    0.33,   New conv eig=  17
                Basis= 52,   Time=    0.39,   New conv eig=  20
End of sweep: Basis= 52,   Time=    0.39,   New conv eig=  20
                Basis= 30,   Time=    0.51,   New conv eig=   0
                Basis= 33,   Time=    0.51,   New conv eig=   0
                Basis= 36,   Time=    0.51,   New conv eig=   0
End of sweep: Basis= 36,   Time=    0.64,   New conv eig=   0

results =

  EigenResults with properties:

    Eigenvectors: [2124×19 double]
     Eigenvalues: [19×1 double]
             Mesh: [1×1 FEMesh]
```

Plot the solution for mode 10.

```
pdeplot(model,'XYData',results.Eigenvectors(:,10))
```
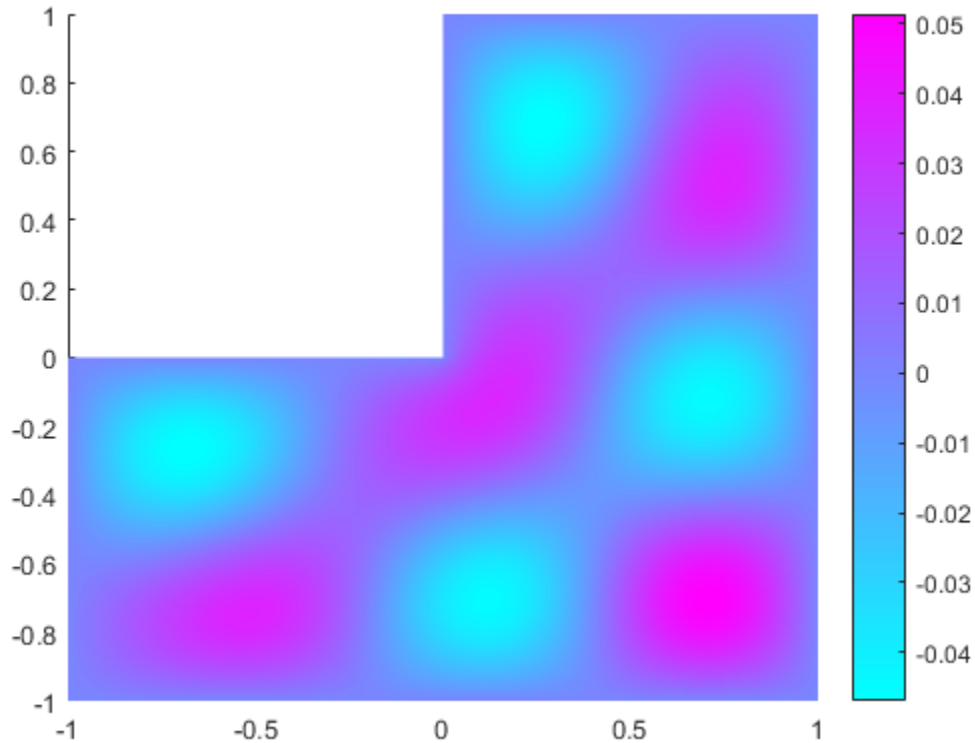
- "Eigenvalues and Eigenmodes of the L-Shaped Membrane" on page 3-168
- "Eigenvalues and Eigenmodes of a Square" on page 3-179

## See Also

### See Also
solvepdeeig | StationaryResults | TimeDependentResults

### Topics
"Eigenvalues and Eigenmodes of the L-Shaped Membrane" on page 3-168

**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)
```
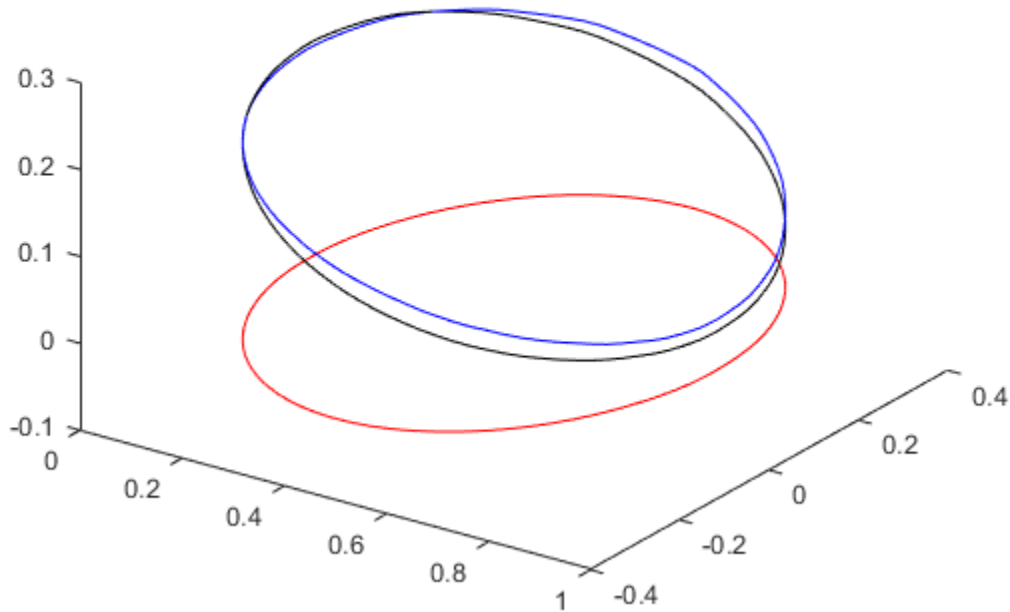
**Interpolate a time-varying solution**

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

### u0ut — 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.

# Definitions

## 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 2-196 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 2-196.

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.

# See Also

## See Also
pdeInterpolant

## Topics
"Mesh Data" on page 2-196

**Introduced in R2014b**

# evaluateCGradient

Evaluate flux of PDE solution

## Compatibility

This page describes the recommended workflow. For the corresponding step in the legacy workflow, see pdecgrad.

## Syntax

```
[cgradx,cgrady] = evaluateCGradient(results,xq,yq)
[cgradx,cgrady,cgradz] = evaluateCGradient(results,xq,yq,zq)
[ ___ ] = evaluateCGradient(results,querypoints)

[ ___ ] = evaluateCGradient( ___ ,iU)

[ ___ ] = evaluateCGradient( ___ ,iT)

[cgradx,cgrady] = evaluateCGradient(results)
[cgradx,cgrady,cgradz] = evaluateCGradient(results)
```

## Description

[cgradx,cgrady] = evaluateCGradient(results,xq,yq) returns the flux of PDE solution for the stationary equation at the 2-D points specified in xq and yq. The flux of the solution is the tensor product of c-coefficient and gradients of the PDE solution, $\mathbf{c} \otimes \nabla \mathbf{u}$.

[cgradx,cgrady,cgradz] = evaluateCGradient(results,xq,yq,zq) returns the flux of PDE solution for the stationary equation at the 3-D points specified in xq, yq, and zq.

[ ___ ] = evaluateCGradient(results,querypoints) returns the flux of PDE solution for the stationary equation at the 2-D or 3-D points specified in querypoints.

[ ___ ] = evaluateCGradient( ___ ,iU) returns the flux of the solution of the PDE system for equation indices (components) iU. When evaluating flux for a system of PDEs, specify iU after the input arguments in any of the previous syntaxes.

The first dimension of cgradx, cgrady, and, in the 3-D case, cgradz corresponds to query points. The second dimension corresponds to equation indices iU.

[ ___ ] = evaluateCGradient( ___ ,iT) returns the flux of PDE solution for the time-dependent equation or system of time-dependent equations at times iT. When evaluating flux for a time-dependent PDE, specify iT after the input arguments in any of the previous syntaxes. For a system of time-dependent PDEs, specify both equation indices (components) iU and time indices iT.

The first dimension of cgradx, cgrady, and, in the 3-D case, cgradz corresponds to query points. For a single time-dependent PDE, the second dimension corresponds to time-steps iT. For a system of time-dependent PDEs, the second dimension corresponds to equation indices iU, and the third dimension corresponds to time-steps iT.

[cgradx,cgrady] = evaluateCGradient(results) returns the flux of PDE solution of a 2-D problem at the nodal points of the triangular mesh. The shape of output arrays, cgradx and cgrady, depends on the number of PDEs for which results is the solution. The first dimension of cgradx and cgrady represents the node indices. For a system of stationary or time-dependent PDEs, the second dimension represents equation indices. For a single time-dependent PDE, the second dimension represents time-steps. The third dimension represents time-step indices for a system of time-dependent PDEs.

[cgradx,cgrady,cgradz] = evaluateCGradient(results) returns the flux of PDE solution of a 3-D problem at the nodal points of the tetrahedral mesh. The first dimension of cgradx, cgrady, and cgradz represents the node indices. The second dimension represents the equation indices. For a system of stationary or time-dependent PDEs, the second dimension represents equation indices. For a single time-dependent PDE, the second dimension represents time-steps. The third dimension represents time-step indices for a system of time-dependent PDEs.

## Examples

### Scalar Elliptic Problem

Solve the problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions. Evaluate the tensor product of c-coefficient and gradients of the solution to a scalar elliptic problem at nodal and arbitrary locations. Plot the results.

Create a PDE model and geometry for this problem.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
pdegplot(model,'FaceLabels','on')
```
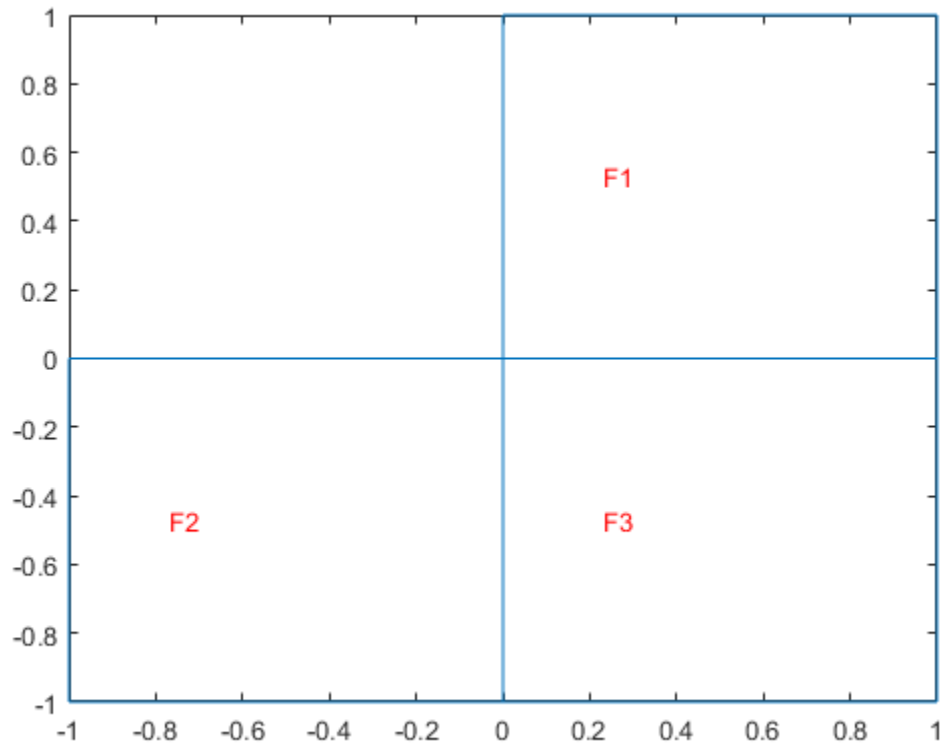


Specify boundary conditions and coefficients.

```
applyBoundaryCondition(model,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);

specifyCoefficients(model,'m',0,'d',0,'c',10,'a',0,'f',1,'Face',1);
specifyCoefficients(model,'m',0,'d',0,'c',5,'a',0,'f',1,'Face',2);
specifyCoefficients(model,'m',0,'d',0,'c',1,'a',0,'f',1,'Face',3);
```

Mesh the geometry and solve the problem.

**6-125**

```
generateMesh(model,'Hmax',0.05);
results = solvepde(model);
u = results.NodalSolution;
```

Compute the flux of the solution and plot the results.

```
[cgradx,cgrady] = evaluateCGradient(results);
```

```
figure
pdeplot(model,'XYData',u,'Contour','on','FlowData',[cgradx,cgrady])
```
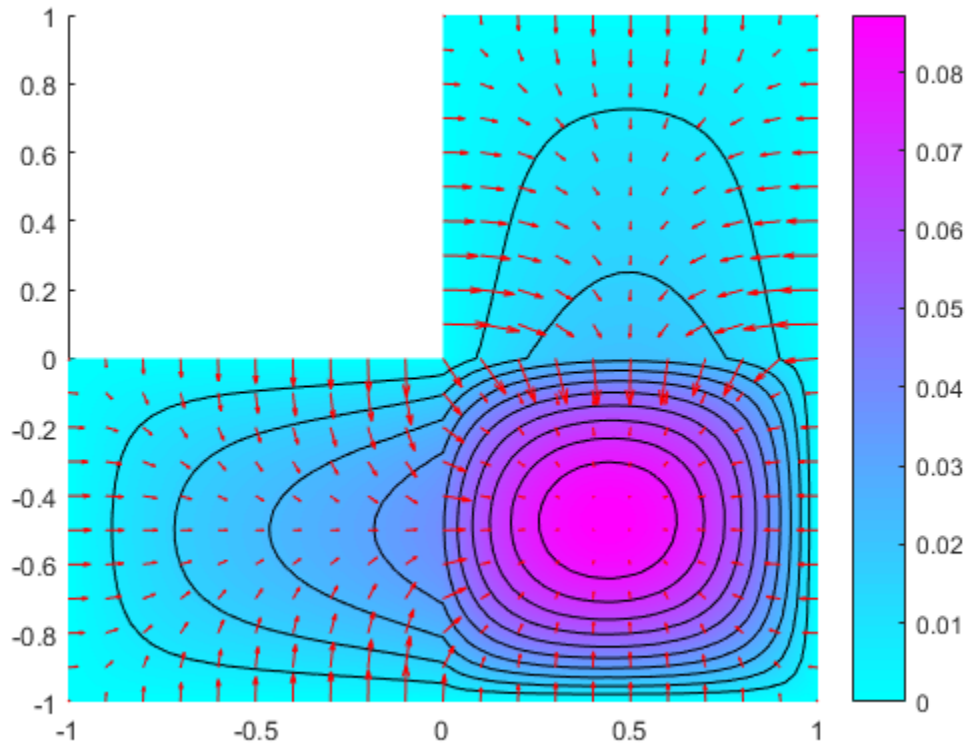


Compute the flux of the solution on the grid from -1 to 1 in each direction using the query points matrix.

```
v = linspace(-1,1,37);
[X,Y] = meshgrid(v);
querypoints = [X(:),Y(:)]';

[cgradxq,cgradyq] = evaluateCGradient(results,querypoints);
```

Alternatively, you can specify the query points as X,Y instead of specifying them as a matrix.

```
[cgradxq,cgradyq] = evaluateCGradient(results,X,Y);
```

Plot the result using the `quiver` plotting function.

```
figure
quiver(X(:),Y(:),cgradxq,cgradyq)
xlabel('x')
ylabel('y')
```
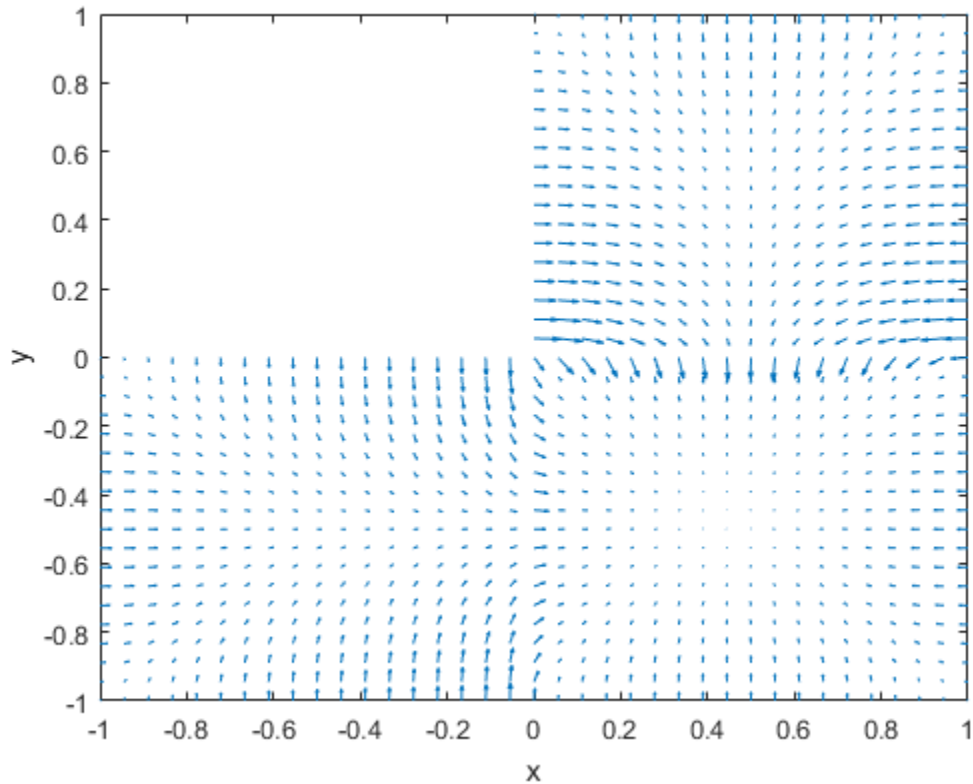
**Stress Components in a Cantilever Beam**

Compute stresses in a cantilever beam subject to shear loading at free end.

Create a PDE model and geometry for this problem.

```
N = 3;
model = createpde(N);
importGeometry(model,'SquareBeam.STL');
pdegplot(model,'FaceLabels','on')
```
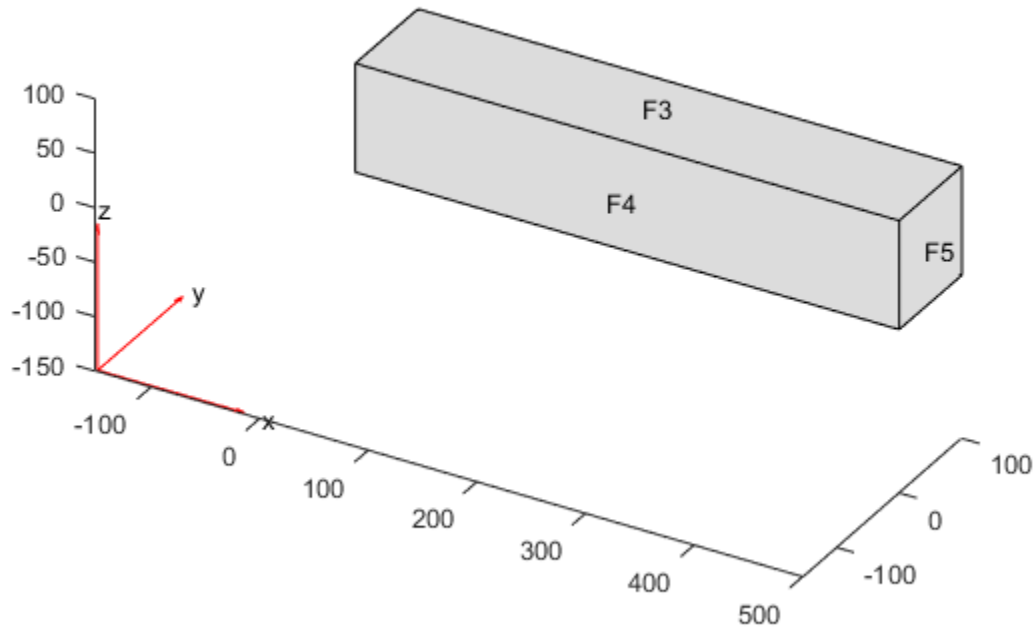
Specify coefficients and apply boundary conditions.

```
E = 2.1e11;
nu = 0.3;
c = elasticityC3D(E, nu);
a = 0;
f = [0;0;0];
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);

applyBoundaryCondition(model,'dirichlet','Face',6,'u',[0 0 0]);
applyBoundaryCondition(model,'neumann','Face',5,'g',[0,0,-3e3]);
```

Mesh the geometry and solve the problem.

```
generateMesh(model,'Hmax',25,'GeometricOrder','quadratic');
```

```
results = solvepde(model);
```

Compute stress, that is, the product of `c`-coefficient and gradients of displacement.

```
[sig_xx,sig_yy,sig_zz] = evaluateCGradient(results);
```

Plot normal component of stress along `x`-direction. The top portion of the beam experiences tension, and the bottom portion experiences compression.

```
figure
pdeplot3D(model,'ColorMapData',sig_xx(:,1))
```
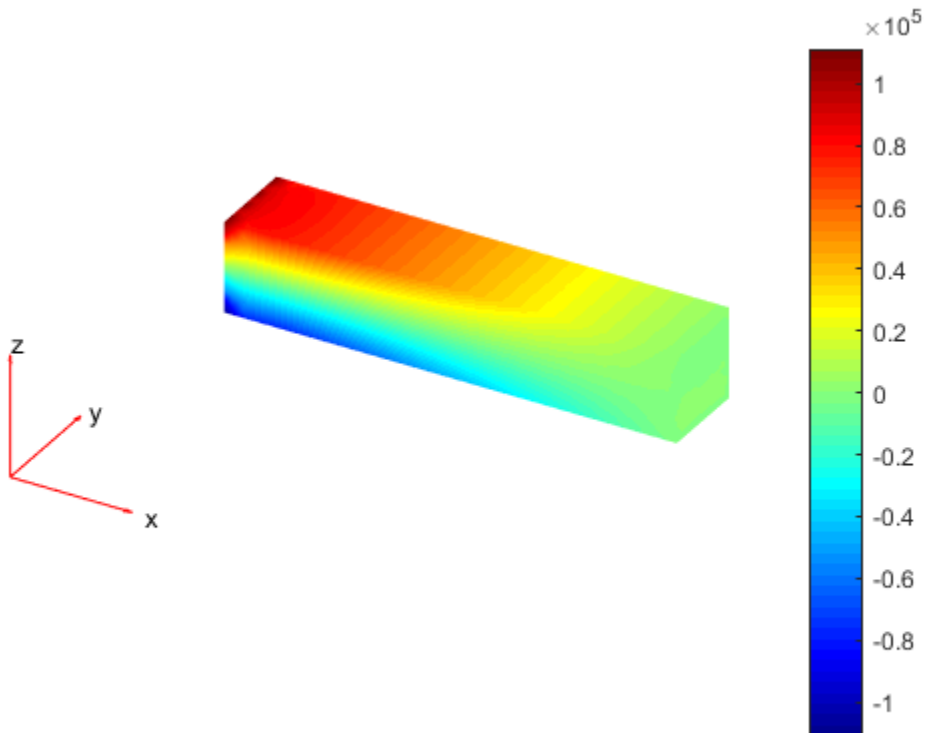


Define a line across the beam from the bottom to the top at mid-span and mid-width. Compute stresses along the line.

```
zg = linspace(0, 100, 10);
xg = 250*ones(size(zg));
yg = 50*ones(size(zg));

[sig_xx,sig_xy,sig_xz] = evaluateCGradient(results,xg,yg,zg,1);
```

Plot the normal stress along x-direction.

```
figure
plot(sig_xx,zg)
grid on
xlabel('\sigma_{xx}')
ylabel('z')
```
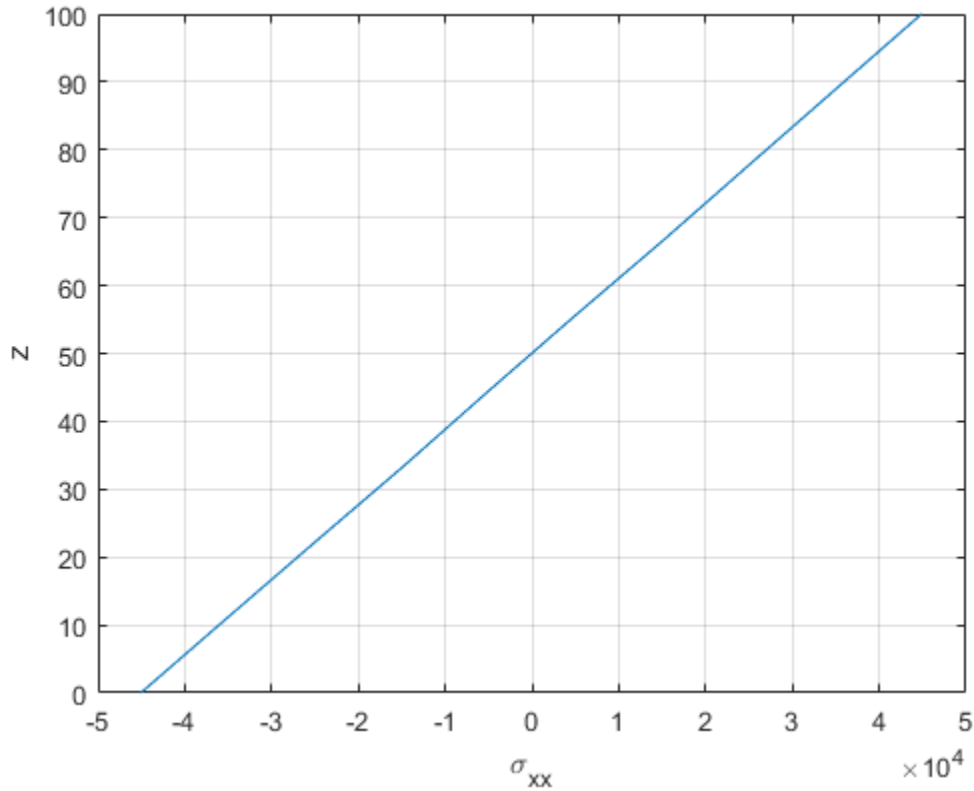
**Stress Components in a Bracket**

Compute stresses in an idealized 3-D mechanical part under an applied load. First, create a PDE model for this problem.

```
N = 3;
model = createpde(N);
```

Import the geometry and plot it.

```
importGeometry(model,'BracketWithHole.stl');
figure
pdegplot(model,'FaceLabels','on')
view(30,30)
```

```
title('Bracket with Face Labels')
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```



**Bracket with Face Labels**
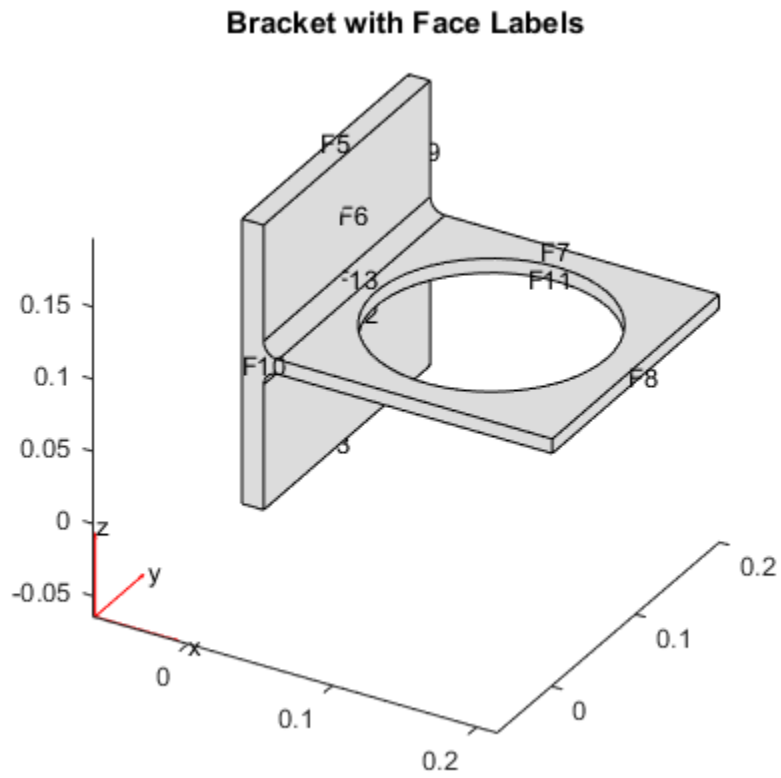
**Bracket with Face Labels, Rear View**
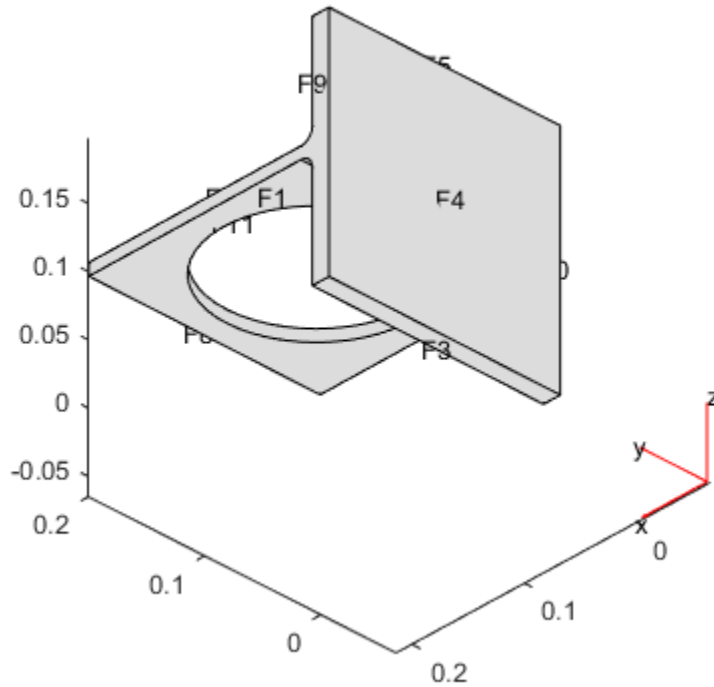


Specify coefficients and apply boundary conditions.

```
E = 200e9; % elastic modulus of steel in Pascals
nu = 0.3; % Poisson's ratio
c = elasticityC3D(E,nu);
a = 0;
f = [0;0;0]; % Assume all body forces are zero
specifyCoefficients(model,'m',0,'d',0,'c',c,'a',a,'f',f);

applyBoundaryCondition(model,'dirichlet','Face',4,'u',[0,0,0]);
distributedLoad = 1e4; % Applied load in Pascals
applyBoundaryCondition(model,'neumann','Face',8,'g',[0,0,-distributedLoad]);
```

Mesh the geometry and solve the problem.

```
bracketThickness = 1e-2; % Thickness of horizontal plate with hole, meters
hmax = bracketThickness; % Maximum element length for a moderately fine mesh
generateMesh(model,'Hmax',hmax,'GeometricOrder','quadratic');

result = solvepde(model);
```

Create a grid. For this grid, compute the stress tensor, which is the product of c-coefficient and gradients of displacement.

```
v = linspace(0,0.2,21);
[xq,yq,zq] = meshgrid(v);

[cgradx,cgrady,cgradz] = evaluateCGradient(result);
```

Extract individual components of stresses.

```
sxx = cgradx(:,1);
sxy = cgradx(:,2);
sxz = cgradx(:,3);

syx = cgrady(:,1);
syy = cgrady(:,2);
syz = cgrady(:,3);

szx = cgradz(:,1);
szy = cgradz(:,2);
szz = cgradz(:,3);
```

Compute von Mises stress.

```
sVonMises = sqrt( 0.5*( (sxx-syy).^2 + (syy -szz).^2 +...
            (szz-sxx).^2) + 3*(sxy.^2 + syz.^2 + szx.^2));
```

Plot von Mises stress. The maximum stress occurs at the weakest section. This section has the least material to support the applied load.

```
pdeplot3D(model,'colormapdata',sVonMises)
```
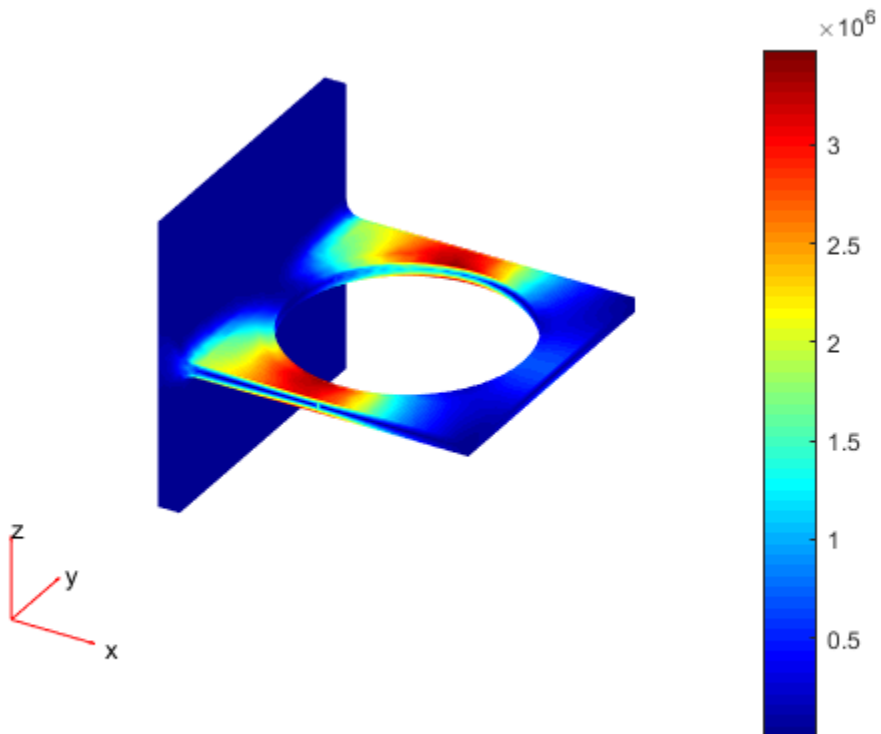
### Heat Transfer Problem on a Square

Solve a 2-D transient heat transfer problem on a square domain and compute heat flow across convective boundary.

Create a PDE model for this problem.

```
numberOfPDE = 1;
model = createpde(numberOfPDE);
```

Create the geometry.

```
g = @squareg;
geometryFromEdges(model,g);
pdegplot(model,'EdgeLabels','on')
```

```
xlim([-1.2,1.2])
ylim([-1.2,1.2])
axis equal
```
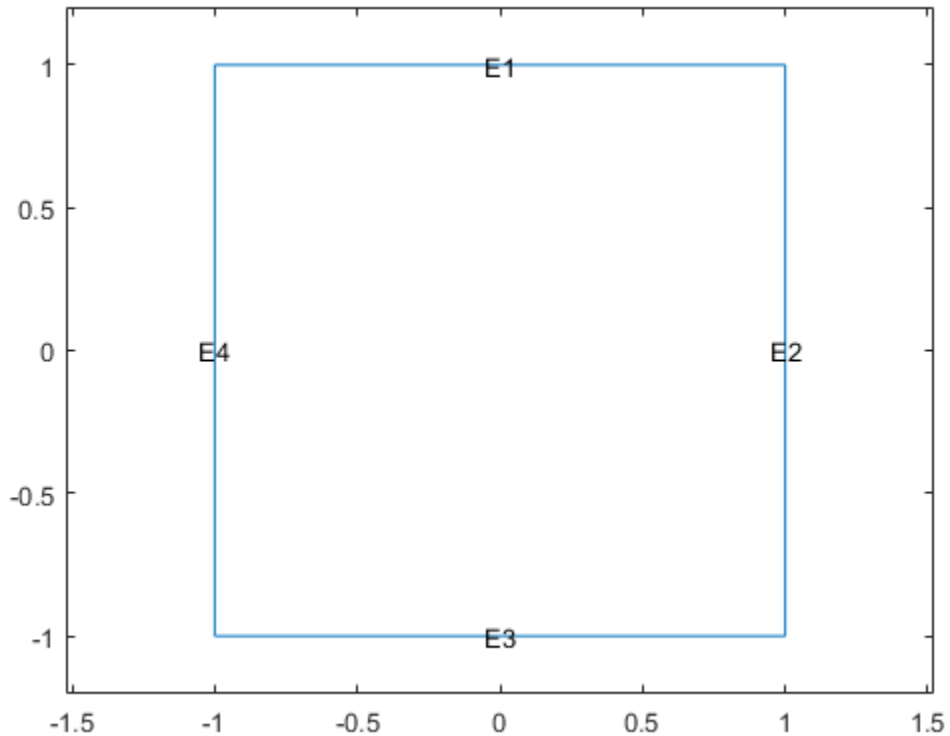


Specify material properties and ambient conditions.

```
rho = 7800;
cp = 500;
k = 100;
Text = 25;
hext = 5000;
```

Specify the coefficients. Apply insulated boundary conditions on three edges and the free convection boundary condition on the right edge.

**6-137**

```
specifyCoefficients(model,'m',0,'d',rho*cp,'c',k,'a',0,'f',0);

applyBoundaryCondition(model,'neumann','Edge', [1,3,4],'q',0,'g',0);
applyBoundaryCondition(model,'neumann','Edge', 2,'q',hext,'g',Text*hext);
```

Set the initial conditions: uniform room temperature across domain and higher temperature on the left edge.

```
setInitialConditions(model,25);
setInitialConditions(model, 100, 'Edge', 4);
```

Generate a mesh and solve the problem using 0:1000:200000 as a vector of times.

```
generateMesh(model);
tlist = 0:1000:200000;
results = solvepde(model,tlist);
```

Define a line at convection boundary to compute heat flux across it.

```
yg = -1:0.1:1;
xg = ones(size(yg));
```

Evaluate the product of c coefficient and spatial gradients at (xg,yg).

```
[qx,qy] = evaluateCGradient(results,xg,yg,1:length(tlist));
```

Spatially integrate gradients to obtain heat flow for each time-step.

```
HeatFlowX(1:length(tlist)) = -trapz(yg,qx(:,1:length(tlist)));
```

Plot convective heat flow over time.

```
figure
plot(tlist,HeatFlowX)
title('Heat flow across convection boundary')
xlabel('Time')
ylabel('Heat flow')
```
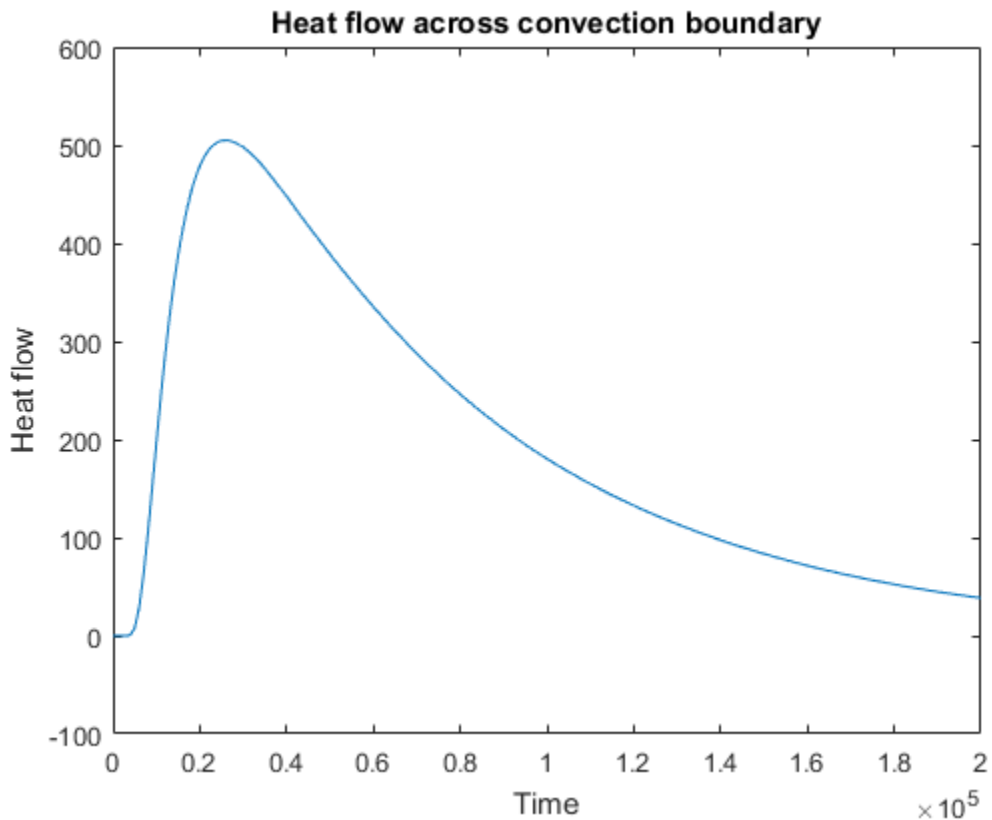
Heat flow across convection boundary

### Heat Transfer Between Two Squares Made of Different Materials

Solve the heat transfer problem for the following 2-D geometry consisting of a square and a diamond made of different materials. Compute the heat flux density and plot it as a vector field.

Create a PDE model for this problem.

```
numberOfPDE = 1;
model = createpde(numberOfPDE);
```

Create a geometry that consists of a square with an embedded diamond.

```
SQ1 = [3; 4; 0; 3; 3; 0; 0; 0; 3; 3];
D1 = [2; 4; 0.5; 1.5; 2.5; 1.5; 1.5; 0.5; 1.5; 2.5];
```

```
gd = [SQ1,D1];
sf = 'SQ1+D1';
ns = char('SQ1','D1');
ns = ns';
dl = decsg(gd,sf,ns);

geometryFromEdges(model,dl);

pdegplot(model,'EdgeLabels','on','FaceLabels','on')
xlim([-1.5,4.5])
ylim([-0.5,3.5])
axis equal
```
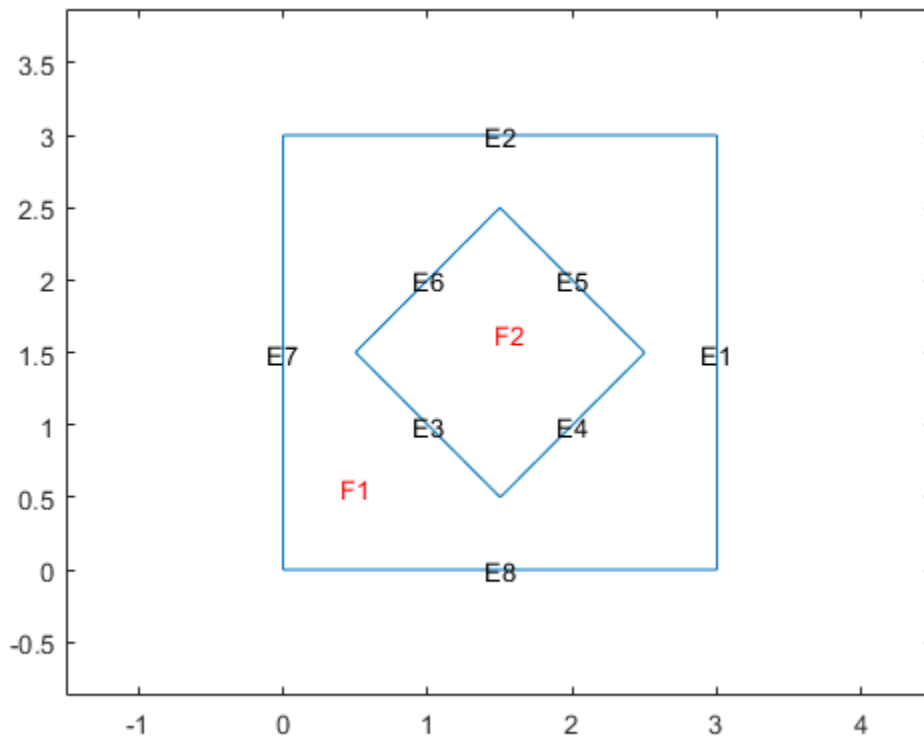


Set parameters for the square region.

```
rho_sq = 2;
C_sq = 0.1;
k_sq = 10;
Q_sq = 0;
h_sq = 0;
```

Set parameters for the diamond region.

```
rho_d = 1;
C_d = 0.1;
k_d = 2;
Q_d = 4;
h_d = 0;
```

Specify the coefficients for both subdomains. Apply the boundary and initial conditions.

```
specifyCoefficients(model,'m',0,'d',rho_sq*C_sq,'c',k_sq,'a',h_sq,'f',Q_sq,'Face',1);
specifyCoefficients(model,'m',0,'d',rho_d*C_d,'c',k_d,'a',h_d,'f',Q_d,'Face',2);

applyBoundaryCondition(model,'dirichlet','Edge',[1,2,7,8],'h',1,'r',0);

setInitialConditions(model,0);
```

Mesh the geometry and solve the problem. To capture the most dynamic part of heat distribution process, solve the problem using `logspace(-2,-1,10)` as a vector of times.

```
generateMesh(model);

tlist = logspace(-2,-1,10);

results = solvepde(model,tlist);
u = results.NodalSolution;
```

Compute the heat flux density. Plot the solution with isothermal lines using a contour plot, and plot the heat flux vector field using arrows. The direction of the heat flow (from higher to lower temperatures) is opposite to the direction of $c \otimes \nabla u$. Therefore, use `-cgradx` and `-cgrady` to show the heat flow.

```
[cgradx,cgrady] = evaluateCGradient(results);

figure
pdeplot(model,'XYData',u(:,10),'Contour','on','FlowData',[-cgradx(:,10),-cgrady(:,10)]
```
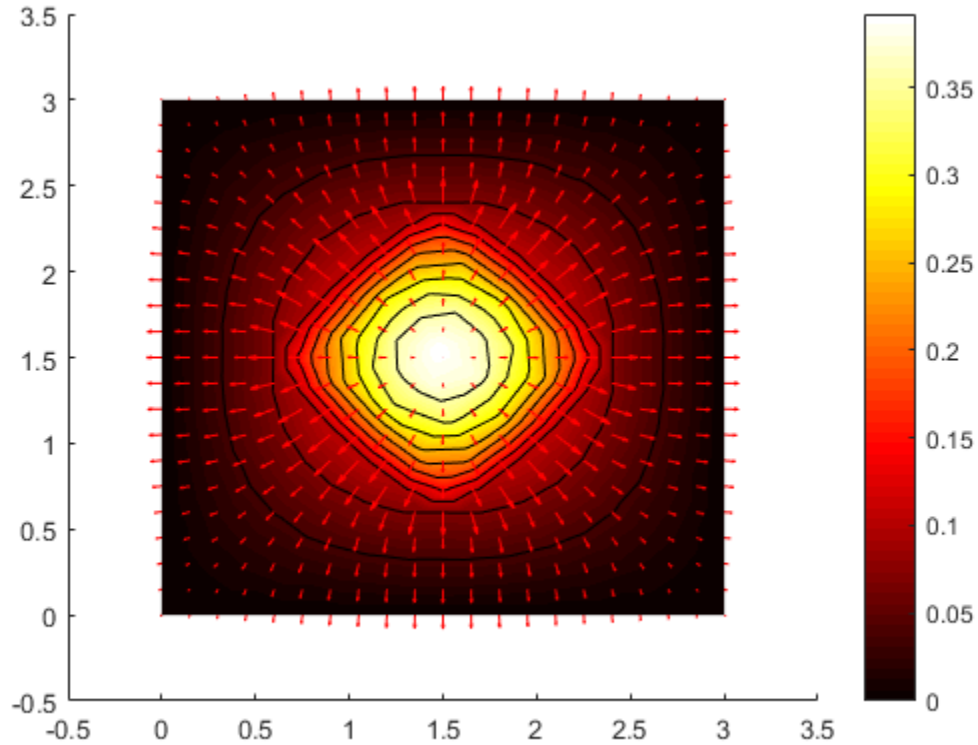
- "Deflection Analysis of a Bracket"
- "Dynamics of a Damped Cantilever Beam"
- "Heat Transfer Between Two Squares Made of Different Materials: PDE App" on page 3-103

## 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. `evaluateCGradient` evaluates the tensor product of `c`-coefficient and gradients of the PDE solution at either 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.

`evaluateCGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. For a single stationary PDE, the result consists of column vectors of the same size. To ensure that the dimensions of the returned `x`-, `y`-, and `z`-components are consistent with the dimensions of the original query points, use `reshape`. For example, use `cgradx = reshape(cgradx,size(xq))`.

For a time-dependent PDE or a system of PDEs, the first dimension of the resulting arrays corresponds to spatial points specified by the column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`.

Data Types: `double`

**yq — *y*-coordinate query points**
real array

*y*-coordinate query points, specified as a real array. `evaluateCGradient` evaluates the tensor product of `c`-coefficient and gradients of the PDE solution at either 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.

`evaluateCGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. For a single stationary PDE, the result consists of column vectors of the same size. To ensure that the dimensions of the returned `x`-, `y`-, and `z`-components are consistent with the dimensions of the original query points, use `reshape`. For example, use `cgrady = reshape(cgrady,size(yq))`.

For a time-dependent PDE or a system of PDEs, the first dimension of the resulting arrays corresponds to spatial points specified by the column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`.

Data Types: `double`

### `zq` — *z*-coordinate query points
real array

*z*-coordinate query points, specified as a real array. `evaluateCGradient` evaluates the tensor product of `c`-coefficient and gradients of the PDE 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.

`evaluateCGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. For a single stationary PDE, the result consists of column vectors of the same size. To ensure that the dimensions of the returned `x`-, `y`-, and `z`-components are consistent with the dimensions of the original query points, use `reshape`. For example, use `cgradz = reshape(cgradz,size(zq))`.

For a time-dependent PDE or a system of PDEs, the first dimension of the resulting arrays corresponds to spatial points specified by the column vectors `xq(:)`, `yq(:)`, and (if present) `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. `evaluateCGradient` evaluates the tensor product of `c`-coefficient and gradients of the PDE 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`

### `iT` — Time indices
vector of positive integers

Time indices, specified as a vector of positive integers. Each entry in `iT` specifies a time index.

Example: `iT = 1:5:21` specifies every fifth time-step up to 21.

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`

# Output Arguments

**`cgradx` — *x*-component of the flux of the PDE solution**
array

*x*-component of the flux of the PDE solution, returned as an array. The first array dimension represents the node index. If `results` is a `StationaryResults` object, the second array dimension represents the equation index for a system of PDEs. If `results` is a `TimeDependentResults` object, the second array dimension represents either the time-step for a single PDE or the equation index for a system of PDEs. The third array dimension represents the time-step index for a system of time-dependent PDEs. For information about the size of `cgradx`, see "Dimensions of Solutions, Gradients, and Fluxes" on page 3-232.

For query points that are outside the geometry, `cgradx = NaN`.

**`cgrady` — *y*-component of the flux of the PDE solution**
array

*y*-component of the flux of the PDE solution, returned as an array. The first array dimension represents the node index. If `results` is a `StationaryResults` object, the second array dimension represents the equation index for a system of PDEs. If `results` is a `TimeDependentResults` object, the second array dimension represents either the time-step for a single PDE or the equation index for a system of PDEs. The third array dimension represents the time-step index for a system of time-dependent PDEs. For information about the size of `cgrady`, see "Dimensions of Solutions, Gradients, and Fluxes" on page 3-232.

For query points that are outside the geometry, `cgrady = NaN`.

**`cgradz` — *z*-component of the flux of the PDE solution**
array

*z*-component of the flux of the PDE solution, returned as an array. The first array dimension represents the node index. If `results` is a `StationaryResults` object, the second array dimension represents the equation index for a system of PDEs. If `results` is a `TimeDependentResults` object, the second array dimension represents either the time-step for a single PDE or the equation index for a system of PDEs. The third array dimension represents the time-step index for a system of time-dependent PDEs. For information about the size of `cgradz`, see "Dimensions of Solutions, Gradients, and Fluxes" on page 3-232.

For query points that are outside the geometry, `cgradz = NaN`.

## Tips

*   While the `results` object contains the solution and its gradient (both calculated at the nodal points of the triangular or tetrahedral mesh), it does not contain the flux of the PDE solution. To compute the flux at the nodal locations, call `evaluateCGradient` without specifying locations. By default, `evaluateCGradient` uses nodal locations.

## See Also

### See Also
`evaluateGradient` | `interpolateSolution` | `PDEModel` | `StationaryResults` | `TimeDependentResults`

### Topics
"Deflection Analysis of a Bracket"
"Dynamics of a Damped Cantilever Beam"
"Heat Transfer Between Two Squares Made of Different Materials: PDE App" on page 3-103

**Introduced in R2016b**

# evaluateGradient

Evaluate gradients of PDE solutions at arbitrary points

## Compatibility

This page describes the recommended workflow. For the corresponding step in the legacy workflow, see `pdegrad`.

## Syntax

```
[gradx,grady] = evaluateGradient(results,xq,yq)
[gradx,grady,gradz] = evaluateGradient(results,xq,yq,zq)
[ ___ ] = evaluateGradient(results,querypoints)

[ ___ ] = evaluateGradient( ___ ,iU)

[ ___ ] = evaluateGradient( ___ ,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 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.

The first dimension of `gradx`, `grady`, and, in 3-D case, `gradz` corresponds to query points. The second dimension corresponds to equation indices `iU`.

[ ___ ] = evaluateGradient( ___ ,iT) returns the interpolated values of the gradients for the time-dependent equation or system of time-dependent equations at times iT. When evaluating gradient for a time-dependent PDE, specify iT after the input arguments in any of the previous syntaxes. For a system of time-dependent equations, specify both time indices iT and equation indices (components) iU.

The first dimension of gradx, grady, and, in 3-D case, gradz corresponds to query points. For a single time-dependent PDE, the second dimension corresponds to time-steps iT. For a system of time-dependent PDEs, the second dimension corresponds to equation indices iU, and the third dimension corresponds to time-steps 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,'dirichlet','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')
```
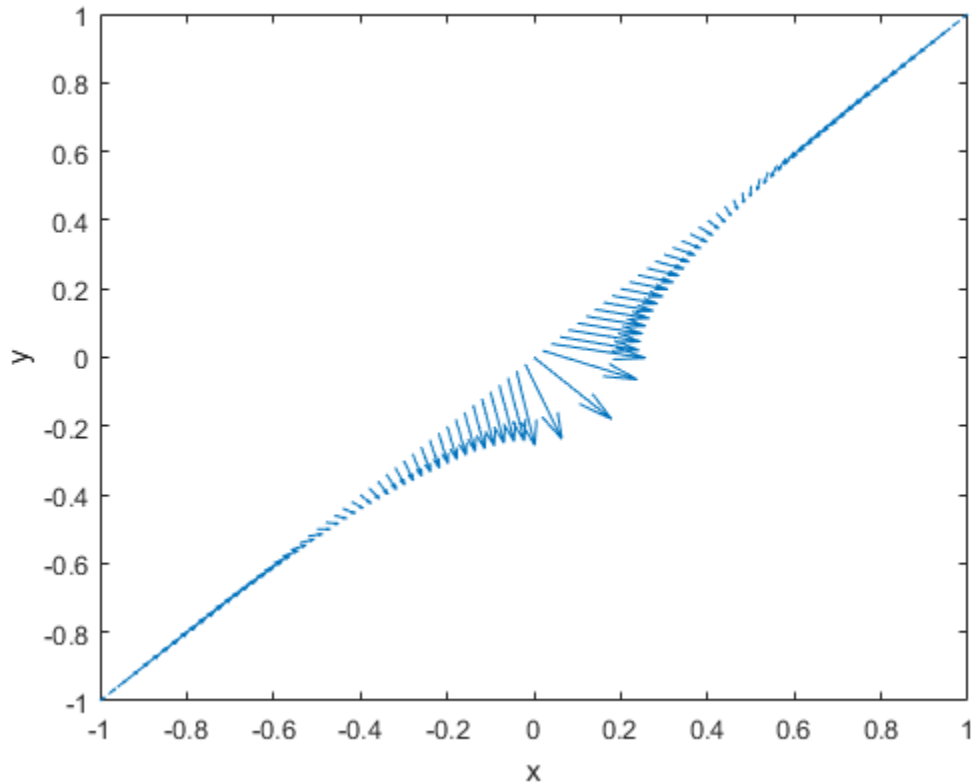


### Evaluate Gradients for Poisson's Equation

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,'dirichlet','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')
```
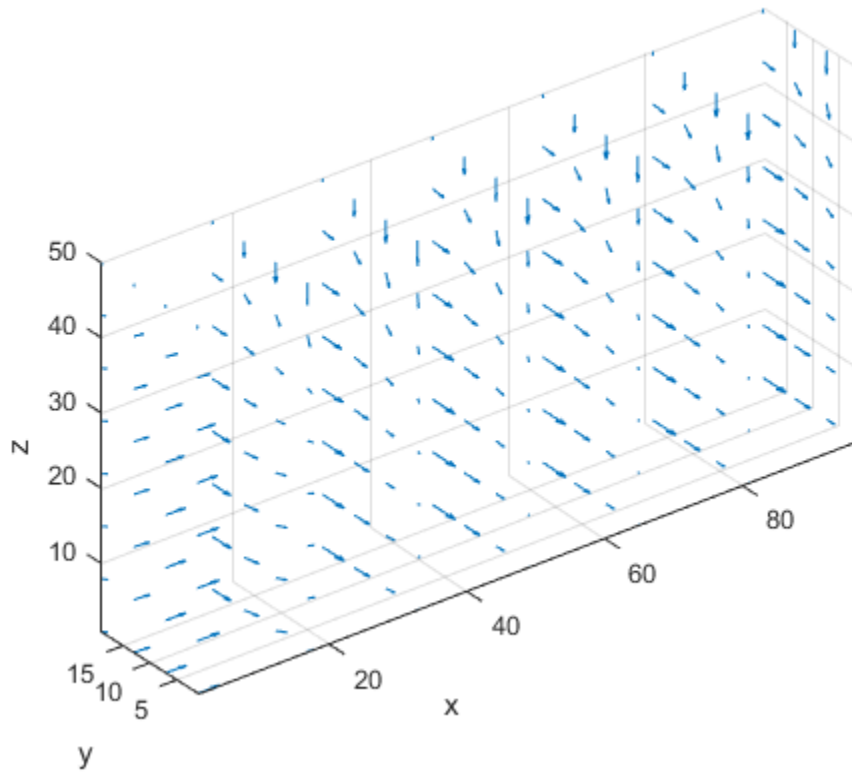
**Evaluate Gradients Using Query Matrix**

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,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
```

**6-151**

```
                              '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')
```
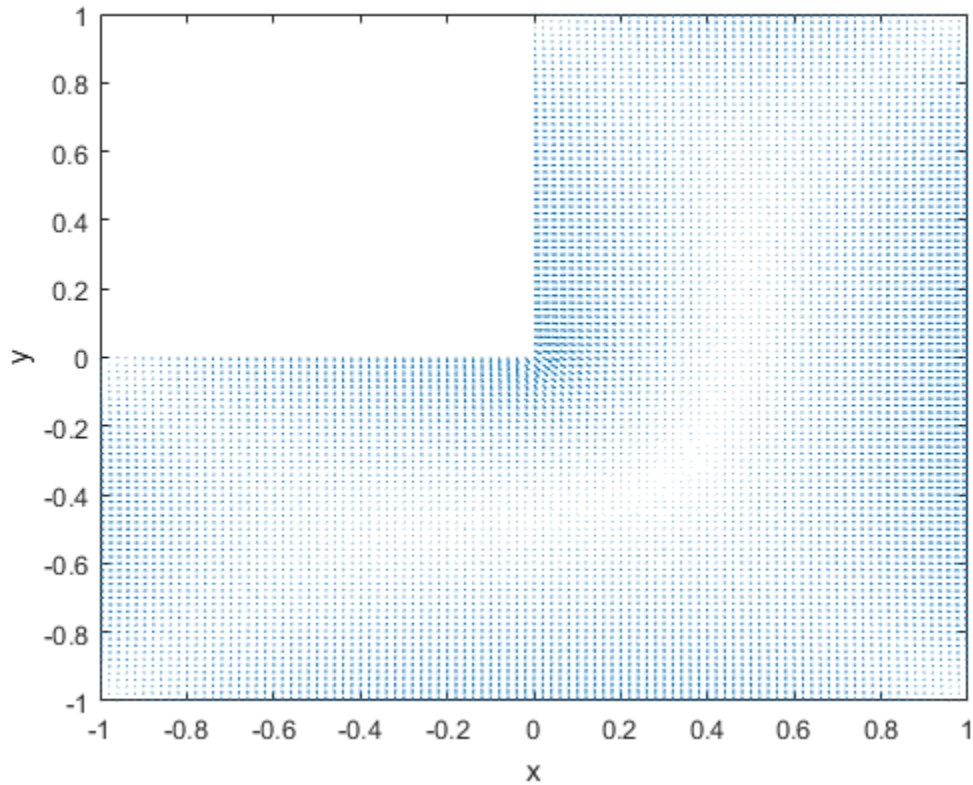
Zoom in on a particular part of the plot to see more details. For example, limit the plotting range to 0.2 in each direction.

```
axis([-0.2 0.2 -0.2 0.2])
```
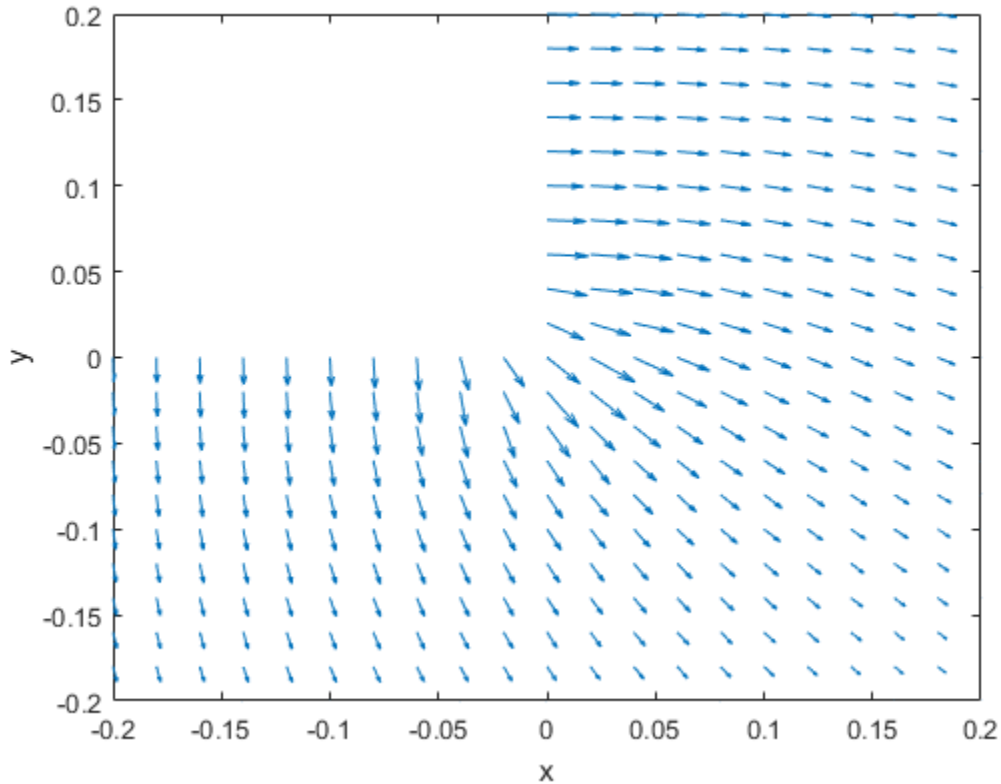
**Evaluate Gradients of Solution of Elliptic System**

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".

```
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])
```
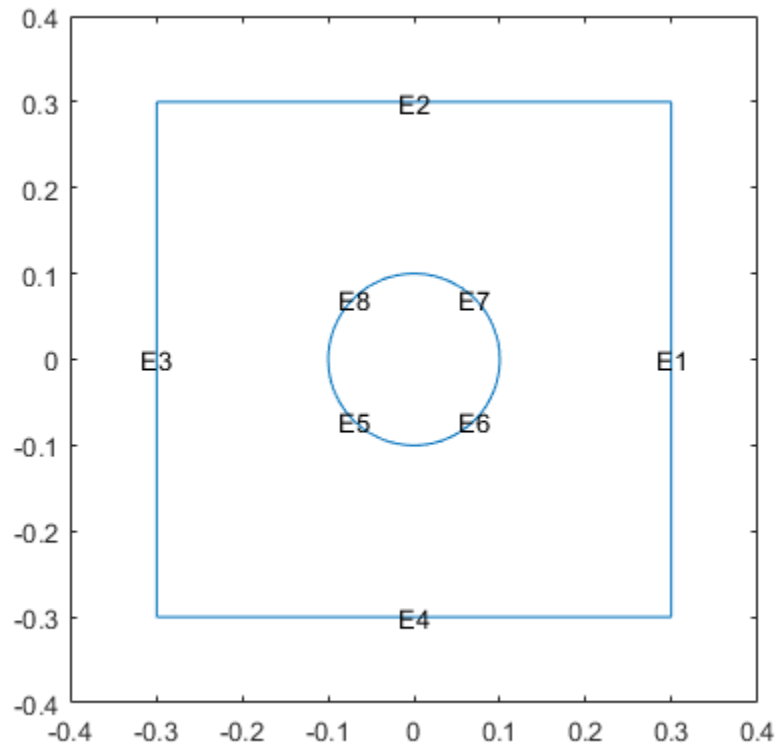


Set the boundary conditions and coefficients.

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

applyBoundaryCondition(model,'dirichlet','Edge',3,'u',[-1,1]);
applyBoundaryCondition(model,'dirichlet','Edge',1,'u',[1,-1]);
applyBoundaryCondition(model,'neumann','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])
```
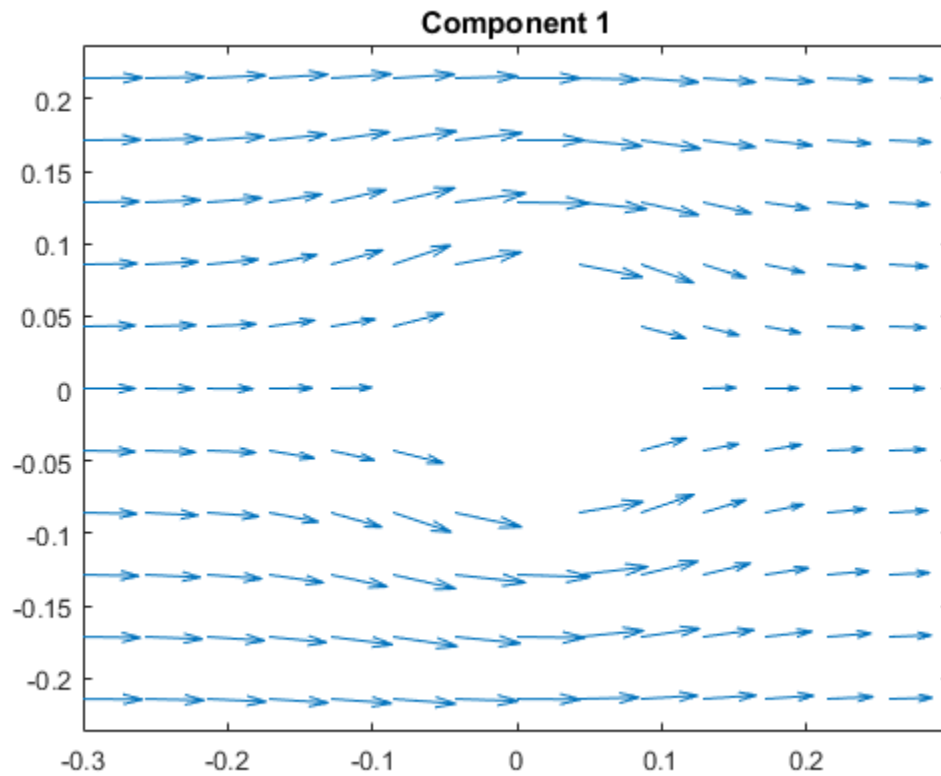
**Component 1**

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])
```
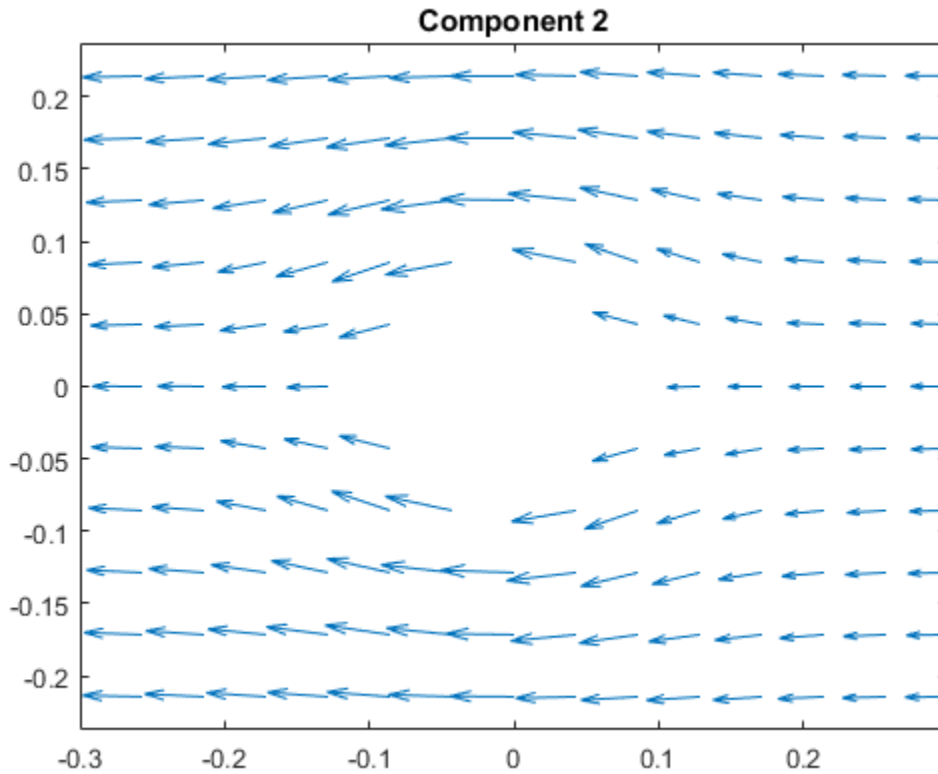
**Component 2**

### Evaluate Gradients of Solution of Hyperbolic System

Solve a system of hyperbolic PDEs and evaluate gradients.

Import slab geometry for a 3-D problem with three solution components. Plot the geometry.

```
model = createpde(3);
importGeometry(model,'Plate10x10x1.stl');
pdegplot(model,'FaceLabels','on','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,'dirichlet','Face',2,'u',[0,0,0]);
applyBoundaryCondition(model,'neumann','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.

```
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`. You might have to wait a few minutes for the solution.

```
tlist = 0:5e-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 following six values from the time interval `tlist`.

```
figure
t = [1:2:11];
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')
ylim([-5e-6, 20e-6])
```



- "Plot 2-D Solutions and Their Gradients" on page 3-199
- "Plot 3-D Solutions and Their Gradients" on page 3-210
- "Dimensions of Solutions, Gradients, and Fluxes" on page 3-232

## 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(:)`. For a single stationary PDE, the result consists of 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))`.

For a time-dependent PDE or a system of PDEs, the first dimension of the resulting arrays corresponds to spatial points specified by the column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`.

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(:)`. For a single stationary PDE, the result consists of 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))`.

For a time-dependent PDE or a system of PDEs, the first dimension of the resulting arrays corresponds to spatial points specified by the column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`.

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(:)`. For a single stationary PDE, the result consists of 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))`.

For a time-dependent PDE or a system of PDEs, the first dimension of the resulting arrays corresponds to spatial points specified by the column vectors `xq(:)`, `yq(:)`, and (if present) `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 indices
vector of positive integers

Time indices, specified as a vector of positive integers. Each entry in `iT` specifies a time index.

Example: `iT = 1:5:21` specifies every fifth time-step 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, Gradients, and Fluxes" on page 3-232.

### `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, Gradients, and Fluxes" on page 3-232.

### `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, Gradients, and Fluxes" on page 3-232.

# 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

## See Also
contour | evaluateCGradient | interpolateSolution | PDEModel | quiver |
quiver3 | StationaryResults | TimeDependentResults

## Topics
"Plot 2-D Solutions and Their Gradients" on page 3-199
"Plot 3-D Solutions and Their Gradients" on page 3-210
"Dimensions of Solutions, Gradients, and Fluxes" on page 3-232

**Introduced in R2016a**

# evaluateHeatFlux

Evaluate heat flux of a thermal solution at nodal or arbitrary spatial locations

## Compatibility

This page describes specific workflow for heat transfer problems. For general workflow, see `evaluateCGradient`.

## Syntax

```
[qx,qy] = evaluateHeatFlux(thermalresults,xq,yq)
[qx,qy,qz] = evaluateHeatFlux(thermalresults,xq,yq,zq)
[ ___ ] = evaluateHeatFlux(thermalresults,querypoints)

[ ___ ] = evaluateHeatFlux( ___ ,iT)

[qx,qy] = evaluateHeatFlux(thermalresults)
[qx,qy,qz] = evaluateHeatFlux(thermalresults)
```

## Description

`[qx,qy] = evaluateHeatFlux(thermalresults,xq,yq)` returns the heat flux for a thermal problem at the 2-D points specified in `xq` and `yq`. This syntax is valid for both the steady-state and transient thermal models.

`[qx,qy,qz] = evaluateHeatFlux(thermalresults,xq,yq,zq)` returns the heat flux for a thermal problem at the 3-D points specified in `xq`, `yq`, and `zq`. This syntax is valid for both the steady-state and transient thermal models.

`[ ___ ] = evaluateHeatFlux(thermalresults,querypoints)` returns the heat flux for a thermal problem at the 2-D or 3-D points specified in `querypoints`. This syntax is valid for both the steady-state and transient thermal models.

`[ ___ ] = evaluateHeatFlux( ___ ,iT)` returns the heat flux for a thermal problem at the times specified in `iT`. You can specify `iT` after the input arguments in any of the previous syntaxes.

The first dimension of qx, qy, and, in the 3-D case, qz corresponds to query points. The second dimension corresponds to time steps iT.

[qx,qy] = evaluateHeatFlux(thermalresults) returns the heat flux for a 2-D problem at the nodal points of the triangular mesh. The first dimension of qx and qy represents the node indices. The second dimension represents time steps.

[qx,qy,qz] = evaluateHeatFlux(thermalresults) returns the heat flux for a 3-D thermal problem at the nodal points of the tetrahedral mesh. The first dimension of qx, qy, and qz represents the node indices. The second dimension represents time steps.

## Examples

### Heat Flux for 2-D Steady-State Thermal Model

For a 2-D steady-state thermal model, evaluate heat flux at the nodal locations and at the points specified by x and y coordinates.

Create a thermal model for steady-state analysis.

```
thermalmodel = createpde('thermal');
```

Create the geometry and include it in the model.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1,'R1',('R1')');
geometryFromEdges(thermalmodel,g);
pdegplot(thermalmodel,'EdgeLabels','on')
xlim([-1.5 1.5])
axis equal
```

Assuming that this geometry represents an iron plate, the thermal conductivity is $79.5 \, W/(mK)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',79.5,'Face',1);
```

Apply a constant temperature of 500 K to the bottom of the plate (edge 3). Also, assume that the top of the plate (edge 1) is insulated, and apply convection on the two sides of the plate (edges 2 and 4).

```
thermalBC(thermalmodel,'Edge',3,'Temperature',500);
thermalBC(thermalmodel,'Edge',1,'HeatFlux',0);
thermalBC(thermalmodel,'Edge',[2 4], ...
                       'ConvectionCoefficient',25, ...
```

```
                         'AmbientTemperature',50);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
results = solve(thermalmodel)


results =

  SteadyStateThermalResults with properties:

    Temperature: [177×1 double]
     XGradients: [177×1 double]
     YGradients: [177×1 double]
     ZGradients: []
           Mesh: [1×1 FEMesh]
```

Evaluate heat flux at the nodal locations.

```
[qx,qy] = evaluateHeatFlux(results);

figure
pdeplot(thermalmodel,'FlowData',[qx qy])
```
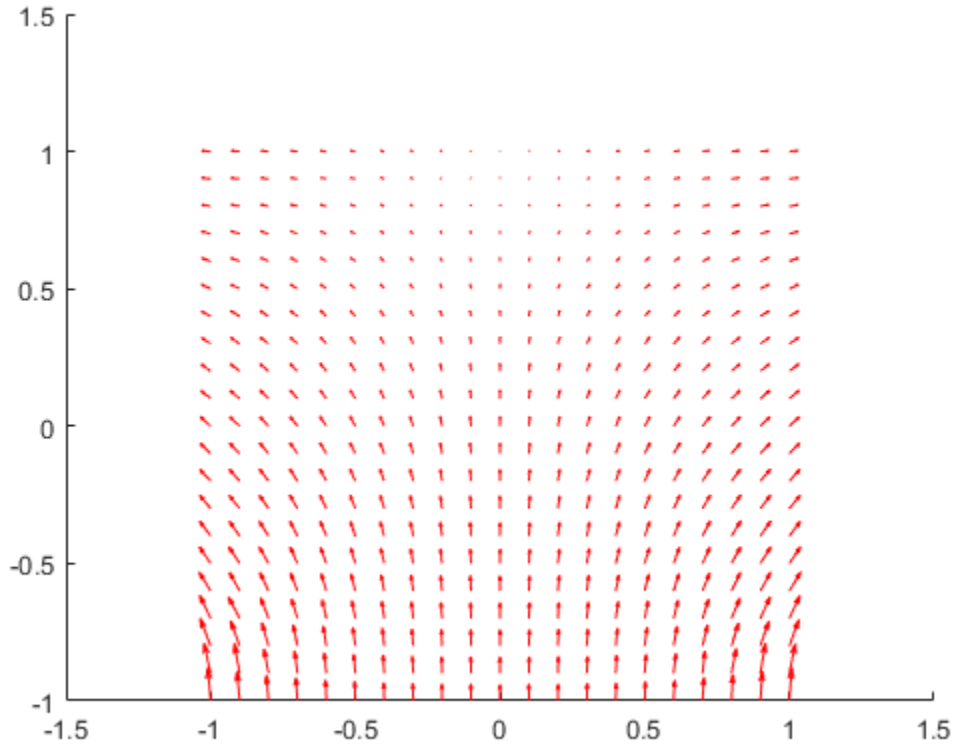
Create a grid specified by x and y coordinates, and evaluate heat flux to the grid.

```
v = linspace(-0.5,0.5,11);
[X,Y] = meshgrid(v);

[qx,qy] = evaluateHeatFlux(results,X,Y);
```

Reshape the qTx and qTy vectors, and plot the resulting heat flux.

```
qx = reshape(qx,size(X));
qy = reshape(qy,size(Y));
figure
quiver(X,Y,qx,qy)
```
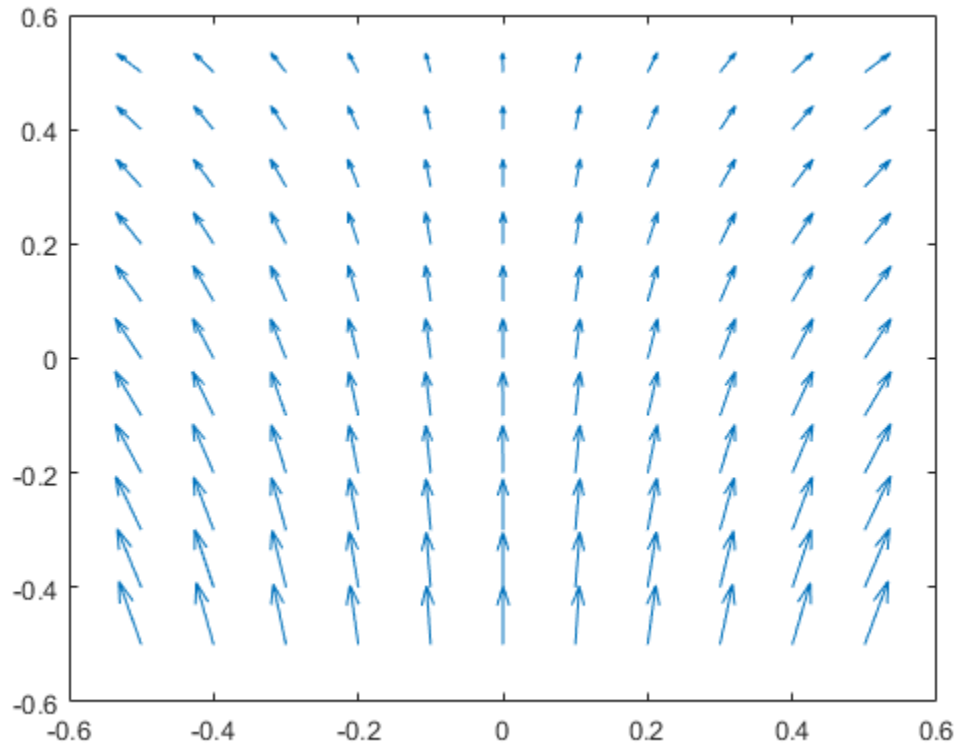
Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:) Y(:)]';
[qx,qy] = evaluateHeatFlux(results,querypoints);

qx = reshape(qx,size(X));
qy = reshape(qy,size(Y));
figure
quiver(X,Y,qx,qy)
```

**6-171**
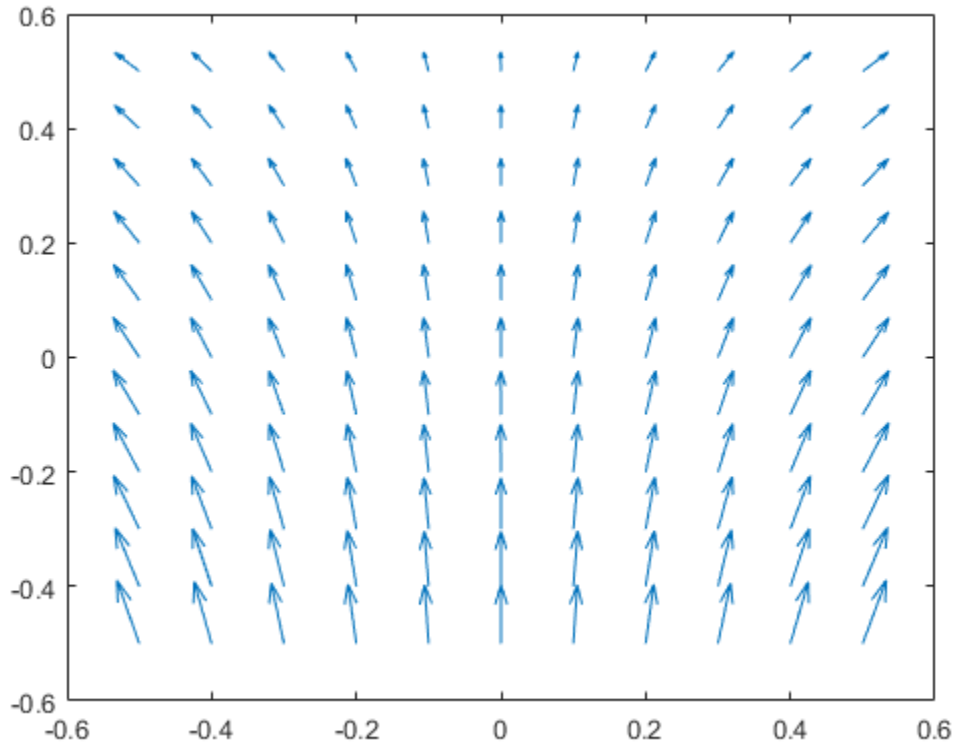
**Heat Flux for 3-D Steady-State Thermal Model**

For a 3-D steady-state thermal model, evaluate heat flux at the nodal locations and at the points specified by x, y, and z coordinates.

Create a thermal model for steady-state analysis.

```
thermalmodel = createpde('thermal');
```

Create the following 3-D geometry and include it in the model.

```
importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabels','on','FaceAlpha',0.5)
title('Copper block, cm')
```

```
axis equal
```

**Copper block, cm**



Assuming that this is a copper block, the thermal conductivity of the block is approximately $4\,W/(cmK)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',4);
```

Apply a constant temperature of 373 K to the left side of the block (face 1) and a constant temperature of 573 K to the right side of the block (face 3).

```
thermalBC(thermalmodel,'Face',1,'Temperature',373);
thermalBC(thermalmodel,'Face',3,'Temperature',573);
```

Apply a heat flux boundary condition to the bottom of the block.

```
thermalBC(thermalmodel,'Face',4,'HeatFlux',-20);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)


thermalresults =

  SteadyStateThermalResults with properties:

    Temperature: [28811×1 double]
     XGradients: [28811×1 double]
     YGradients: [28811×1 double]
     ZGradients: [28811×1 double]
           Mesh: [1×1 FEMesh]
```

Evaluate heat flux at the nodal locations.

```
[qx,qy,qz] = evaluateHeatFlux(thermalresults);

figure
pdeplot3D(thermalmodel,'FlowData',[qx qy qz])
```
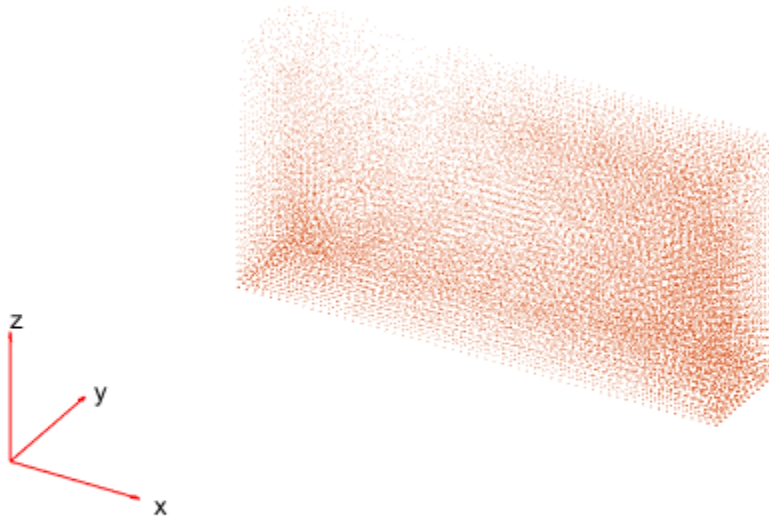
Create a grid specified by x, y, and z coordinates, and evaluate heat flux to the grid.

```
[X,Y,Z] = meshgrid(1:26:100,1:6:20,1:11:50);

[qx,qy,qz] = evaluateHeatFlux(thermalresults,X,Y,Z);
```

Reshape the qx, qy, and qz vectors, and plot the resulting heat flux.

```
qx = reshape(qx,size(X));
qy = reshape(qy,size(Y));
qz = reshape(qz,size(Z));
figure
quiver3(X,Y,Z,qx,qy,qz)
```
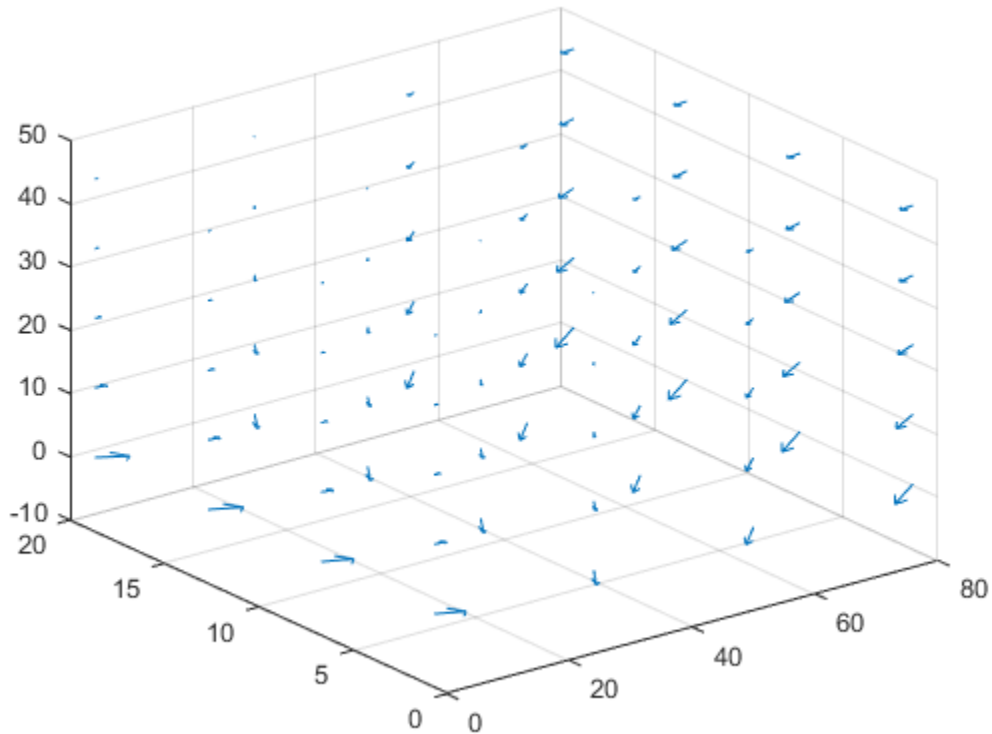
Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:) Y(:) Z(:)]';
[qx,qy,qz] = evaluateHeatFlux(thermalresults,querypoints);

qx = reshape(qx,size(X));
qy = reshape(qy,size(Y));
qz = reshape(qz,size(Z));
figure
quiver3(X,Y,Z,qx,qy,qz)
```
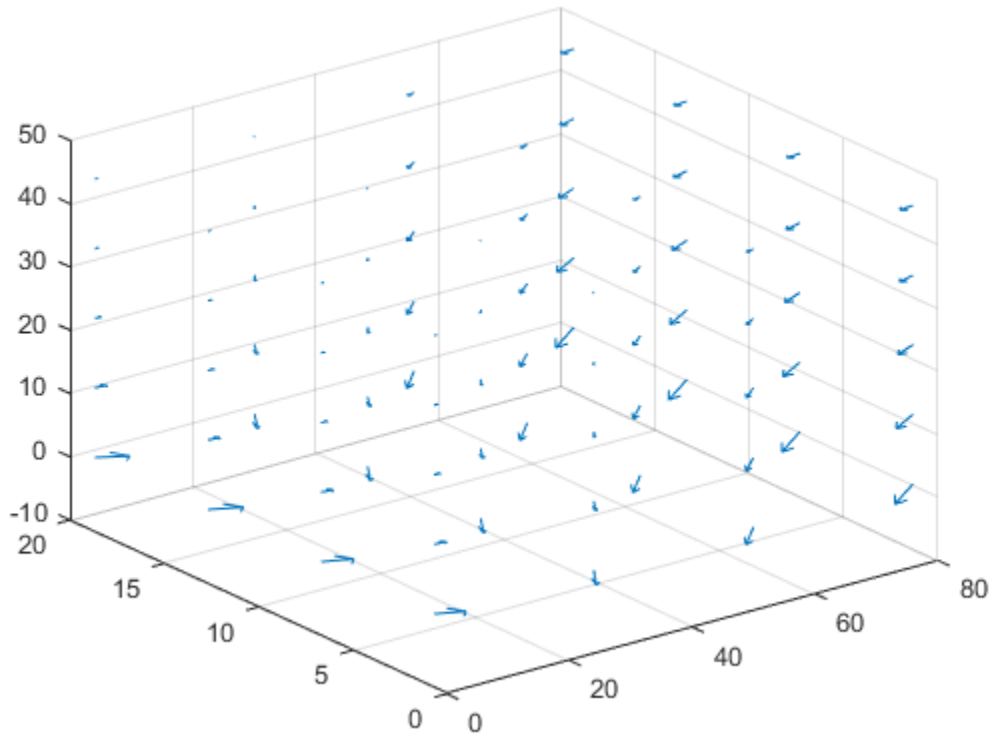
## Heat Flux for Transient Thermal Model on Square

Solve a 2-D transient heat transfer problem on a square domain, and compute heat flow across a convective boundary.

Create a thermal model for this problem.

```
thermalmodel = createpde('thermal','transient');
```

Create the geometry and include it in the model.

```
g = @squareg;
geometryFromEdges(thermalmodel,g);
pdegplot(thermalmodel,'EdgeLabels','on')
xlim([-1.2 1.2])
```

```
ylim([-1.2 1.2])
axis equal
```



Assign the following thermal properties: thermal conductivity is $100\,W/(m°C)$, mass density is $7800\,kg/m^3$, and specific heat is $500\,J/(kg°C)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',100, ...
                               'MassDensity',7800, ...
                               'SpecificHeat',500);
```

Apply insulated boundary conditions on three edges and the free convection boundary condition on the right edge.

```
thermalBC(thermalmodel,'Edge',[1 3 4],'HeatFlux',0);
```

```
thermalBC(thermalmodel,'Edge',2,...
                        'ConvectionCoefficient',5000, ...
                        'AmbientTemperature',25);
```

Set the initial conditions: uniform room temperature across domain and higher temperature on the top edge.

```
thermalIC(thermalmodel,25);
thermalIC(thermalmodel,100,'Edge',1);
```

Generate a mesh and solve the problem using `0:1000:200000` as a vector of times.

```
generateMesh(thermalmodel);
tlist = 0:1000:200000;
thermalresults = solve(thermalmodel,tlist)
```

```
thermalresults =

  TransientThermalResults with properties:

      Temperature: [177×201 double]
    SolutionTimes: [1×201 double]
        XGradients: [177×201 double]
        YGradients: [177×201 double]
        ZGradients: []
             Mesh: [1×1 FEMesh]
```

Create a grid specified by `x` and `y` coordinates, and evaluate heat flux to the grid.

```
v = linspace(-1,1,11);
[X,Y] = meshgrid(v);

[qx,qy] = evaluateHeatFlux(thermalresults,X,Y,1:length(tlist));
```

Reshape `qx` and `qy`, and plot the resulting heat flux for the 25th solution step.

```
tlist(25)

figure
quiver(X(:),Y(:),qx(:,25),qy(:,25));
xlim([-1,1])
axis equal
```

**6-179**

```
ans =

     24000
```
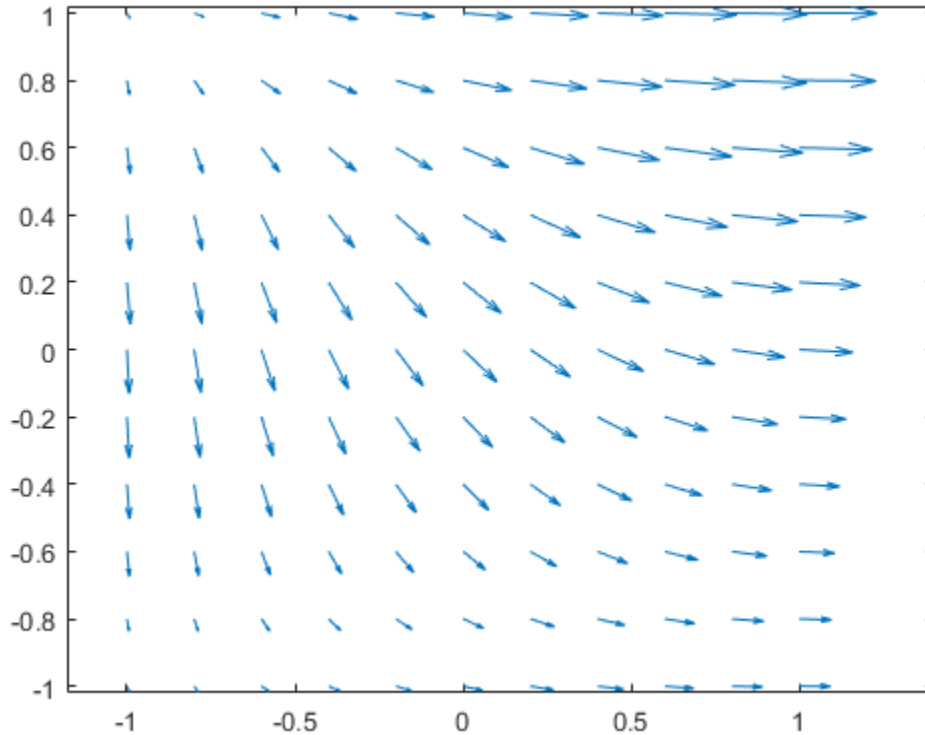


### Heat Flux for Transient Thermal Model on Two Squares Made of Different Materials

Solve the heat transfer problem for the following 2-D geometry consisting of a square and a diamond made of different materials. Compute the heat flux, and plot it as a vector field.

Create a thermal model for transient analysis.

```
thermalmodel = createpde('thermal','transient');
```

Create the geometry and include it in the model.

```
SQ1 = [3; 4; 0; 3; 3; 0; 0; 0; 3; 3];
D1 = [2; 4; 0.5; 1.5; 2.5; 1.5; 1.5; 0.5; 1.5; 2.5];
gd = [SQ1 D1];
sf = 'SQ1+D1';
ns = char('SQ1','D1');
ns = ns';
dl = decsg(gd,sf,ns);
geometryFromEdges(thermalmodel,dl);
pdegplot(thermalmodel,'EdgeLabels','on','FaceLabels','on')
xlim([-1.5 4.5])
ylim([-0.5 3.5])
axis equal
```
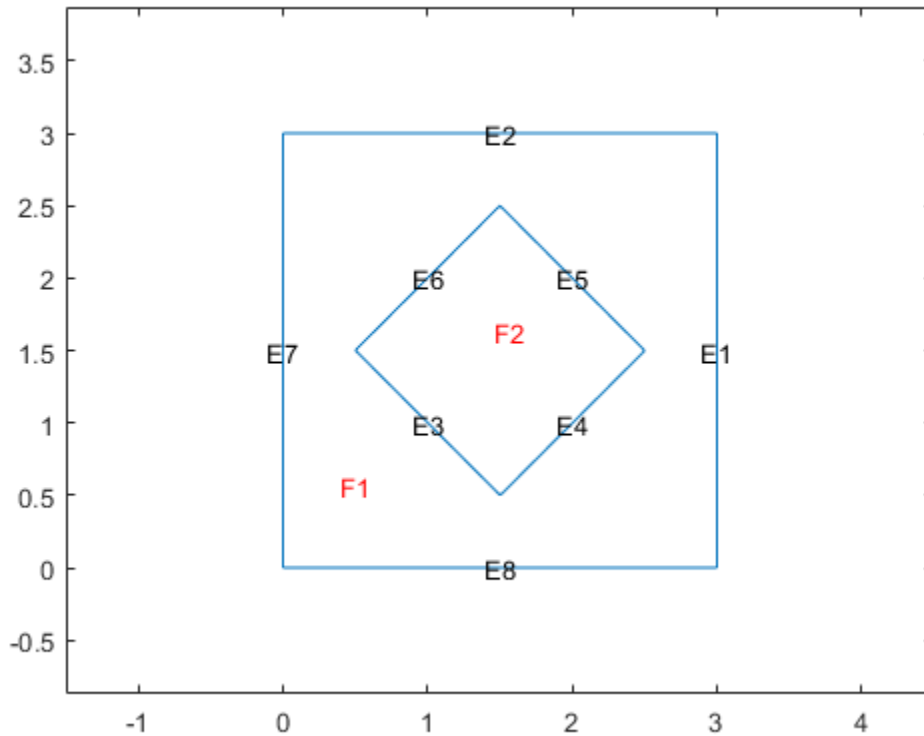
For the square region, assign the following thermal properties: thermal conductivity is $10\,W/(m°C)$, mass density is $2\,kg/m^3$, and specific heat is $0.1\,J/(kg°C)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',10, ...
                               'MassDensity',2, ...
                               'SpecificHeat',0.1, ...
                               'Face',1);
```

For the diamond-shaped region, assign the following thermal properties: thermal conductivity is $2\,W/(m°C)$, mass density is $1\,kg/m^3$, and specific heat is $0.1\,J/(kg°C)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',2, ...
                               'MassDensity',1, ...
                               'SpecificHeat',0.1, ...
                               'Face',2);
```

Assume that the diamond-shaped region is a heat source with the density of $4\,W/m^3$.

```
internalHeatSource(thermalmodel,4,'Face',2);
```

Apply a constant temperature of $0°C$ to the sides of the square plate.

```
thermalBC(thermalmodel,'Temperature',0,'Edge',[1 2 7 8]);
```

Set the initial temperature to $0°C$.

```
thermalIC(thermalmodel,0);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
```

The dynamic for this problem is very fast: the temperature reaches steady state in about 0.1 seconds. To capture the interesting part of the dynamics, set the solution time to `logspace(-2,-1,10)`. This gives 10 logarithmically spaced solution times between 0.01 and 0.1. Solve the equation.

```
tlist = logspace(-2,-1,10);
thermalresults = solve(thermalmodel,tlist);
temp = thermalresults.Temperature;
```

Compute the heat flux density. Plot the solution with isothermal lines using a contour plot, and plot the heat flux vector field using arrows.

```
[qTx,qTy] = evaluateHeatFlux(thermalresults);

figure
pdeplot(thermalmodel,'XYData',temp(:,10),'Contour','on', ...
                     'FlowData',[qTx(:,10) qTy(:,10)], ...
                     'ColorMap','hot')
```



## Input Arguments

**thermalresults — Solution of thermal problem**
SteadyStateThermalResults object | TransientThermalResults object

Solution of a thermal problem, specified as a `SteadyStateThermalResults` object or a `TransientThermalResults` object. Create `thermalresults` using the `solve` function.

Example: `thermalresults = solve(thermalmodel)`

### xq — *x*-coordinate query points
real array

*x*-coordinate query points, specified as a real array. `evaluateHeatFlux` evaluates the heat flux 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.

`evaluateHeatFlux` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns the heat flux in a form of a column vector of the same size. To ensure that the dimensions of the returned solution are consistent with the dimensions of the original query points, use `reshape`. For example, use `qx = reshape(qx,size(xq))`.

Data Types: `double`

### yq — *y*-coordinate query points
real array

*y*-coordinate query points, specified as a real array. `evaluateHeatFlux` evaluates the heat flux 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.

`evaluateHeatFlux` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns the heat flux in a form of 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 `qy = reshape(qy,size(yq))`.

Data Types: `double`

### zq — *z*-coordinate query points
real array

*z*-coordinate query points, specified as a real array. `evaluateHeatFlux` evaluates the heat flux at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. So `xq`, `yq`, and `zq` must have the same number of entries.

evaluateHeatFlux converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns the heat flux in a form of 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 `qz = reshape(qz,size(zq))`.

Data Types: `double`

### `querypoints` — Query points
real matrix

Query points, specified as a real matrix with two rows for 2-D geometry or three rows for 3-D geometry. `evaluateHeatFlux` evaluates the heat flux 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`

### `iT` — Time indices
vector of positive integers

Time indices, specified as a vector of positive integers. Each entry in `iT` specifies a time index.

Example: `iT = 1:5:21` specifies every fifth time-step up to 21.

Data Types: `double`

## Output Arguments

### qx — *x*-component of the heat flux
array

*x*-component of the heat flux, returned as an array. The first array dimension represents the node index. The second array dimension represents the time step.

For query points that are outside the geometry, `qx = NaN`.

### qy — *y*-component of the heat flux
array

*y*-component of the heat flux, returned as an array. The first array dimension represents the node index. The second array dimension represents the time step.

For query points that are outside the geometry, qy = NaN.

### qz — *z*-component of the heat flux
array

*z*-component of the heat flux, returned as an array. The first array dimension represents the node index. The second array dimension represents the time step.

For query points that are outside the geometry, qz = NaN.

# See Also

## See Also
evaluateHeatRate | evaluateTemperatureGradient |
interpolateTemperature | SteadyStateThermalResults | ThermalModel |
TransientThermalResults

**Introduced in R2017a**

# evaluateHeatRate

Evaluate integrated heat flow rate normal to specified boundary

## Syntax

```
Qn = evaluateHeatRate(thermalresults,RegionType,RegionID)
```

## Description

`Qn = evaluateHeatRate(thermalresults,RegionType,RegionID)` returns the integrated heat flow rate normal to the boundary specified by `RegionType` and `RegionID`.

## Examples

### Heat Flow From Face of Block
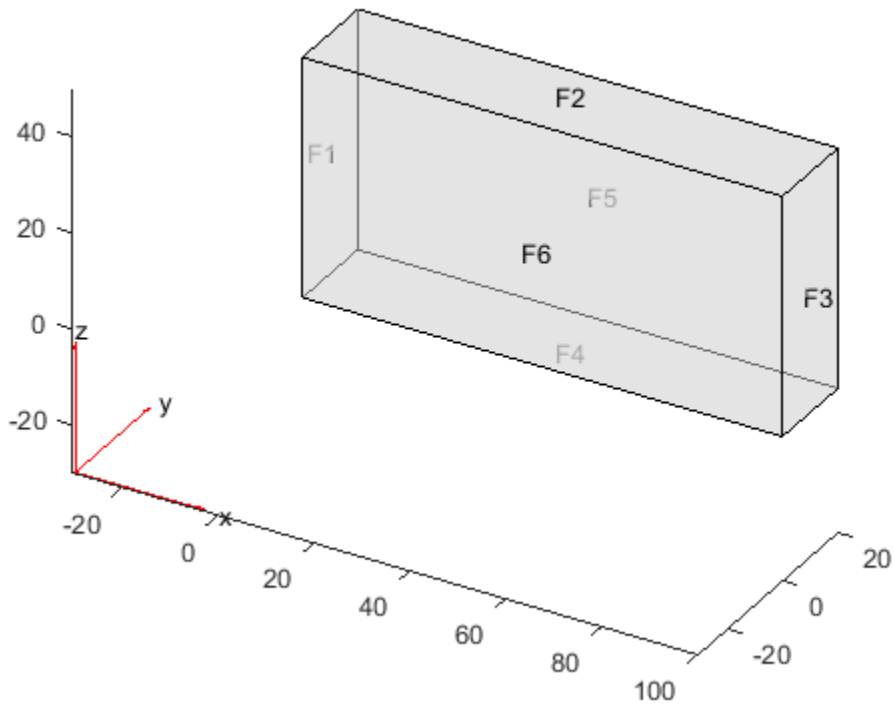
Compute the heat flow rate across a face of the block geometry.

Create a steady-state thermal model.

```
thermalmodel = createpde('thermal','steadystate');
```

Import the block geometry.

```
importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabels','on','FaceAlpha',0.5)
```

Specify the thermal conductivity of the block.

```
thermalProperties(thermalmodel,'ThermalConductivity',80);
```

Apply constant temperatures on the opposite ends of the block. All other faces are insulated by default.

```
thermalBC(thermalmodel,'Face',1,'Temperature',100);
thermalBC(thermalmodel,'Face',3,'Temperature',50);
```

Generate mesh.

```
generateMesh(thermalmodel,'GeometricOrder','linear');
```

Solve the thermal model.

```
thermalresults = solve(thermalmodel);
```

Compute the heat flow rate across face 3 of the block.

```
Qn = evaluateHeatRate(thermalresults,'Face',3)
```

```
Qn =

   4.0000e+04
```

### Convection Cooling of Sphere

Compute the heat flow rate across the surface of the cooling sphere.

Create a thermal model for transient analysis.

```
thermalmodel = createpde('thermal','transient');
```

Create a sphere of radius 1, and assign it to the thermal model.

```
gm = multisphere(1);
thermalmodel.Geometry = gm;
```

Generate mesh.

```
generateMesh(thermalmodel,'GeometricOrder','linear');
```

Specify thermal properties of the sphere.

```
thermalProperties(thermalmodel,'ThermalConductivity',80, ...
                               'SpecificHeat',460, ...
                               'MassDensity',7800);
```

Apply a convection boundary condition on the surface of the sphere.

```
thermalBC(thermalmodel,'Face',1,...
                       'ConvectionCoefficient',500, ...
                       'AmbientTemperature',30);
```

Set the initial temperature.

```
thermalIC(thermalmodel,800);
```

Solve the thermal model.

```
tlist = 0:100:2000;
```

```
result = solve(thermalmodel,tlist);
```

Compute the heat flow rate across the surface of the sphere over time.

```
Qn = evaluateHeatRate(result,'Face',1);
plot(tlist,Qn)
xlabel('Time')
ylabel('Heat Flow Rate')
```
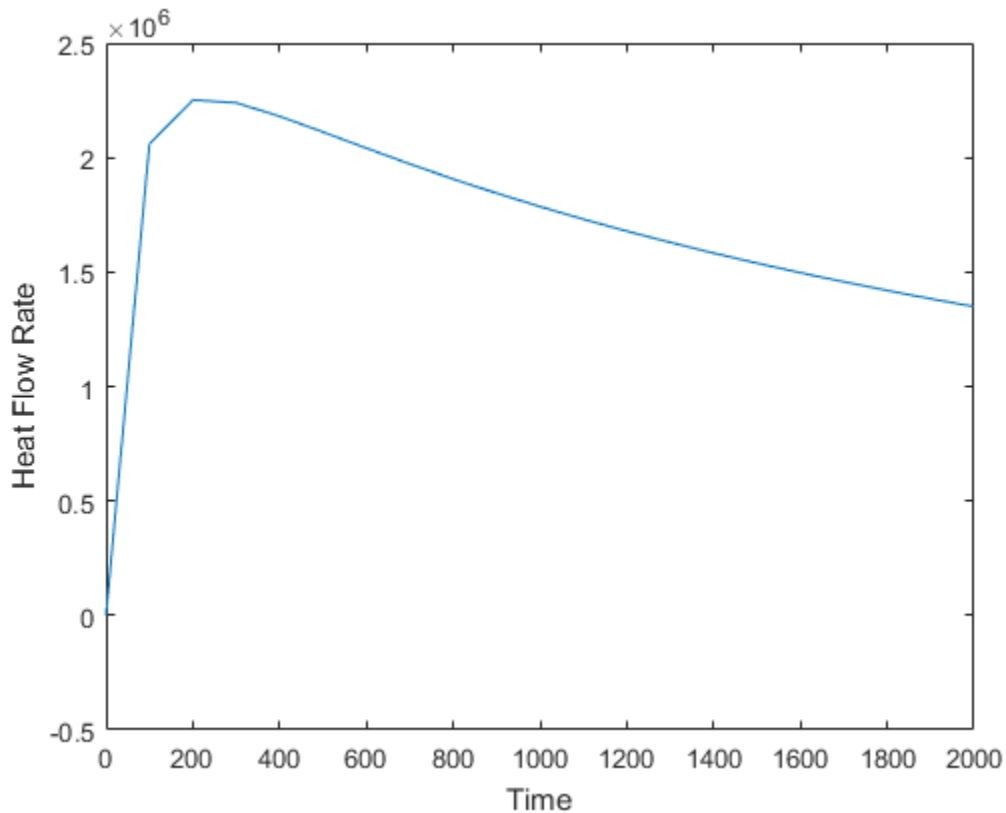


## Input Arguments

**`thermalresults` — Solution of thermal problem**
`SteadyStateThermalResults` object

Solution of a thermal problem, specified as a `SteadyStateThermalResults` object. Create `thermalresults` using the `solve` function.

Example: `thermalresults = solve(thermalmodel)`

**`RegionType` — Geometric region type**
`'Face'` for 3-D geometry | `'Edge'` for 2-D geometry

Geometric region type, specified as `'Face'` for 3-D geometry or `'Edge'` for 2-D geometry.

Example: `Qn = evaluateHeatRate(thermalresults,'Face',3)`

Data Types: `char`

**`RegionID` — Geometric region ID**
positive integer

Geometric region ID, specified as a positive integer. Find the region IDs using the `pdegplot` function with the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) value set to `'on'`.

Example: `Qn = evaluateHeatRate(thermalresults,'Face',3)`

Data Types: `double`

# Output Arguments

**`Qn` — Heat flow rate**
real number | vector of real numbers

Heat flow rate, returned as a real number or, for time-dependent results, a vector of real numbers. This value represents the integrated heat flow rate, measured in energy per unit time, flowing in the direction normal to the boundary. `Qn` is positive if the heat flows out of the domain, and negative if the heat flows into the domain.

## See Also

### See Also
evaluateHeatFlux | evaluateTemperatureGradient |
interpolateTemperature | SteadyStateThermalResults | ThermalModel |
TransientThermalResults

**Introduced in R2017a**

# evaluateTemperatureGradient

Evaluate temperature gradient of a thermal solution at arbitrary spatial locations

## Syntax

```
[gradTx,gradTy] = evaluateTemperatureGradient(thermalresults,xq,yq)
[gradTx,gradTy,gradTz] = evaluateTemperatureGradient(thermalresults,
xq,yq,zq)
[ ___ ] = evaluateTemperatureGradient(thermalresults,querypoints)
[ ___ ] = evaluateTemperatureGradient( ___ ,iT)
```

## Description

`[gradTx,gradTy] = evaluateTemperatureGradient(thermalresults,xq,yq)`
returns the interpolated values of temperature gradients of the thermal model solution
`thermalresults` at the 2-D points specified in `xq` and `yq`. This syntax is valid for both
the steady-state and transient thermal models.

`[gradTx,gradTy,gradTz] = evaluateTemperatureGradient(thermalresults,
xq,yq,zq)` returns the interpolated temperature gradients at the 3-D points specified
in `xq`, `yq`, and `zq`. This syntax is valid for both the steady-state and transient thermal
models.

`[ ___ ] = evaluateTemperatureGradient(thermalresults,querypoints)`
returns the interpolated values of the temperature gradients at the points specified
in `querypoints`. This syntax is valid for both the steady-state and transient thermal
models.

`[ ___ ] = evaluateTemperatureGradient( ___ ,iT)` returns the interpolated values
of the temperature gradients for the time-dependent equation at times `iT`. Specify `iT`
after the input arguments in any of the previous syntaxes.

The first dimension of `gradTx`, `gradTy`, and, in 3-D case, `gradTz` corresponds to query
points. The second dimension corresponds to time-steps `iT`.

# Examples

### Temperature Gradients for 2-D Steady-State Thermal Model

For a 2-D steady-state thermal model, evaluate temperature gradients at the nodal locations and at the points specified by x and y coordinates.

Create a thermal model for steady-state analysis.

```
thermalmodel = createpde('thermal');
```

Create the geometry and include it in the model.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1,'R1',('R1')');
geometryFromEdges(thermalmodel,g);
pdegplot(thermalmodel,'EdgeLabels','on')
xlim([-1.5 1.5])
axis equal
```

Assuming that this geometry represents an iron plate, the thermal conductivity is
$79.5\,W/(mK)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',79.5,'Face',1);
```

Apply a constant temperature of 300 K to the bottom of the plate (edge 3). Also, assume that the top of the plate (edge 1) is insulated, and apply convection on the two sides of the plate (edges 2 and 4).

```
thermalBC(thermalmodel,'Edge',3,'Temperature',300);
thermalBC(thermalmodel,'Edge',1,'HeatFlux',0);
thermalBC(thermalmodel,'Edge',[2 4], ...
                      'ConvectionCoefficient',25, ...
```

```
                        'AmbientTemperature',50);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
results = solve(thermalmodel)


results =

  SteadyStateThermalResults with properties:

    Temperature: [177×1 double]
     XGradients: [177×1 double]
     YGradients: [177×1 double]
     ZGradients: []
           Mesh: [1×1 FEMesh]
```

The solver finds the temperatures and temperature gradients at the nodal locations. To access these values, use `results.Temperature`, `results.XGradients`, and so on. For example, plot the temperature gradients at nodal locations.

```
figure;
pdeplot(thermalmodel,'FlowData',[results.XGradients results.YGradients]);
```
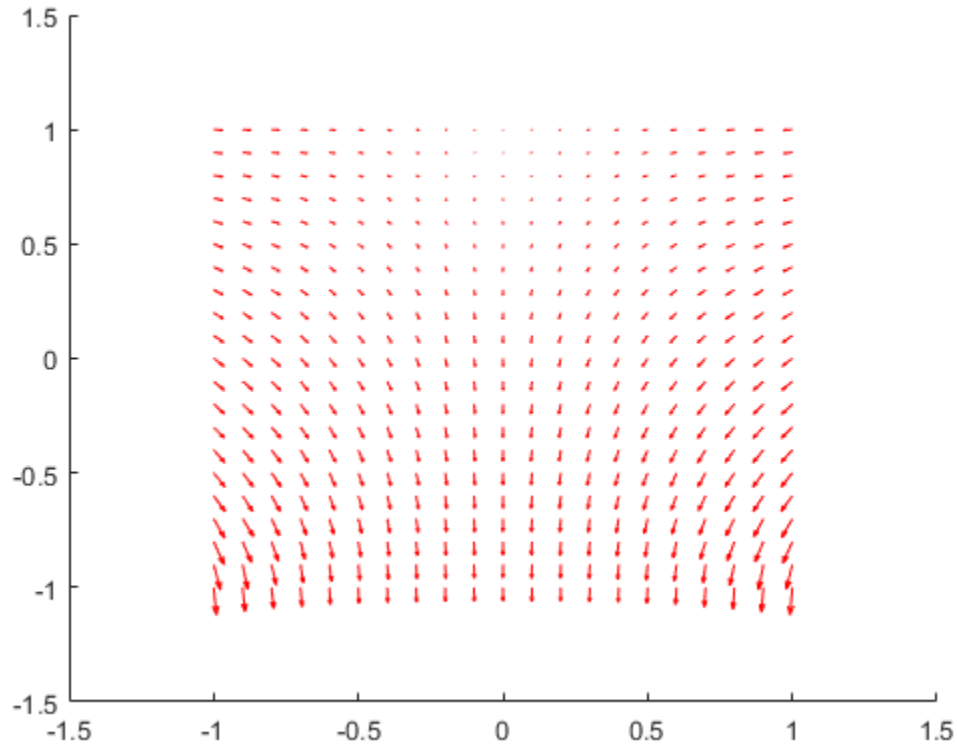
Create a grid specified by x and y coordinates, and evaluate temperature gradients to the grid.

```
v = linspace(-0.5,0.5,11);
[X,Y] = meshgrid(v);

[gradTx,gradTy] = evaluateTemperatureGradient(results,X,Y);
```

Reshape the gradTx and gradTy vectors, and plot the resulting temperature gradients.

```
gradTx = reshape(gradTx,size(X));
gradTy = reshape(gradTy,size(Y));
figure
quiver(X,Y,gradTx,gradTy)
```
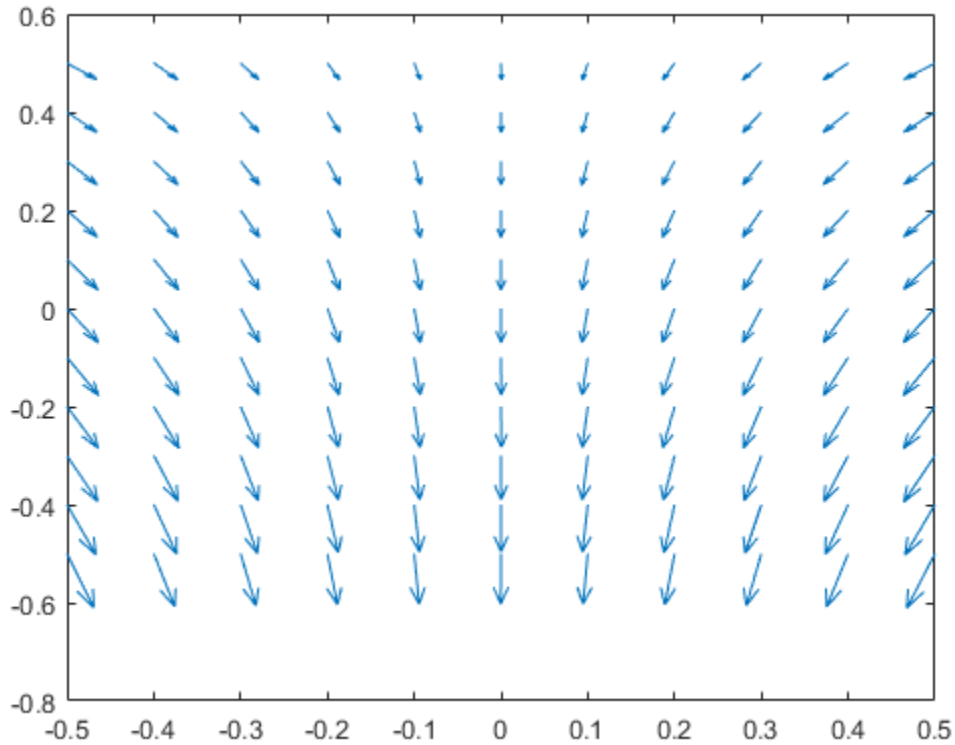
Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:) Y(:)]';
[gradTx,gradTy] = evaluateTemperatureGradient(results,querypoints);

gradTx = reshape(gradTx,size(X));
gradTy = reshape(gradTy,size(Y));
figure
quiver(X,Y,gradTx,gradTy)
```
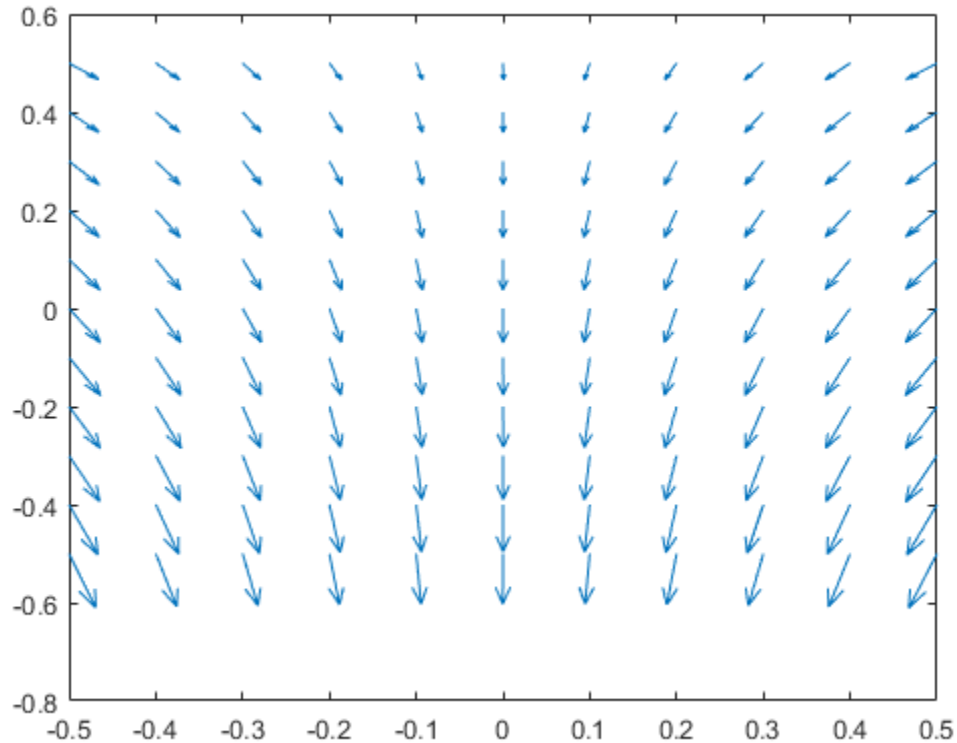
**Temperature Gradients for 3-D Steady-State Thermal Model**

For a 3-D steady-state thermal model, evaluate temperature gradients at the nodal locations and at the points specified by x, y, and z coordinates.

Create a thermal model for steady-state analysis.

```
thermalmodel = createpde('thermal');
```

Create the following 3-D geometry and include it in the model.

```
importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabels','on','FaceAlpha',0.5)
title('Copper block, cm')
```

**6-199**

```
axis equal
```

**Copper block, cm**



Assuming that this is a copper block, the thermal conductivity of the block is approximately $4\,W/(cmK)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',4);
```

Apply a constant temperature of 373 K to the left side of the block (edge 1) and a constant temperature of 573 K to the right side of the block.

```
thermalBC(thermalmodel,'Face',1,'Temperature',373);
thermalBC(thermalmodel,'Face',3,'Temperature',573);
```

Apply a heat flux boundary condition to the bottom of the block.

```
thermalBC(thermalmodel,'Face',4,'HeatFlux',-20);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)


thermalresults =

  SteadyStateThermalResults with properties:

    Temperature: [28811×1 double]
     XGradients: [28811×1 double]
     YGradients: [28811×1 double]
     ZGradients: [28811×1 double]
           Mesh: [1×1 FEMesh]
```

The solver finds the values of temperatures and temperature gradients at the nodal locations. To access these values, use `results.Temperature`, `results.XGradients`, and so on.

Create a grid specified by `x`, `y`, and `z` coordinates, and evaluate temperature gradients to the grid.

```
[X,Y,Z] = meshgrid(1:26:100,1:6:20,1:11:50);

[gradTx,gradTy,gradTz] = evaluateTemperatureGradient(thermalresults,X,Y,Z);
```

Reshape the `gradTx`, `gradTy`, and `gradTz` vectors, and plot the resulting temperature gradients.

```
gradTx = reshape(gradTx,size(X));
gradTy = reshape(gradTy,size(Y));
gradTz = reshape(gradTz,size(Z));

figure
quiver3(X,Y,Z,gradTx,gradTy,gradTz)
axis equal
xlabel('x')
ylabel('y')
zlabel('z')
```
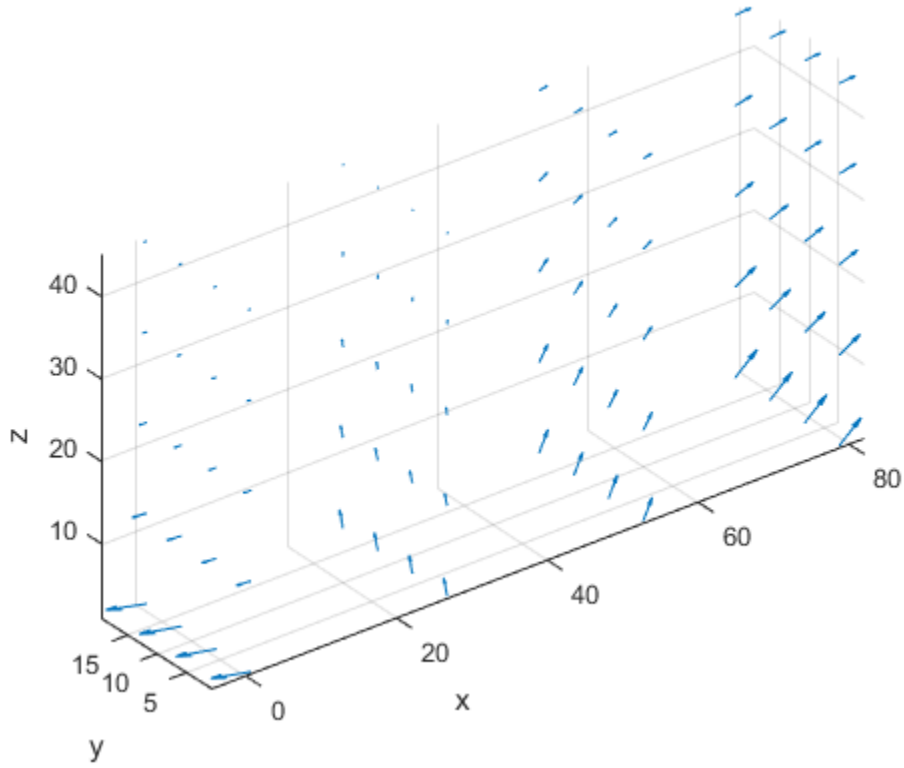
Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:) Y(:) Z(:)]';
[gradTx,gradTy,gradTz] = evaluateTemperatureGradient(thermalresults,querypoints);

gradTx = reshape(gradTx,size(X));
gradTy = reshape(gradTy,size(Y));
gradTz = reshape(gradTz,size(Z));

figure
quiver3(X,Y,Z,gradTx,gradTy,gradTz)
axis equal
xlabel('x')
ylabel('y')
```

```
zlabel('z')
```
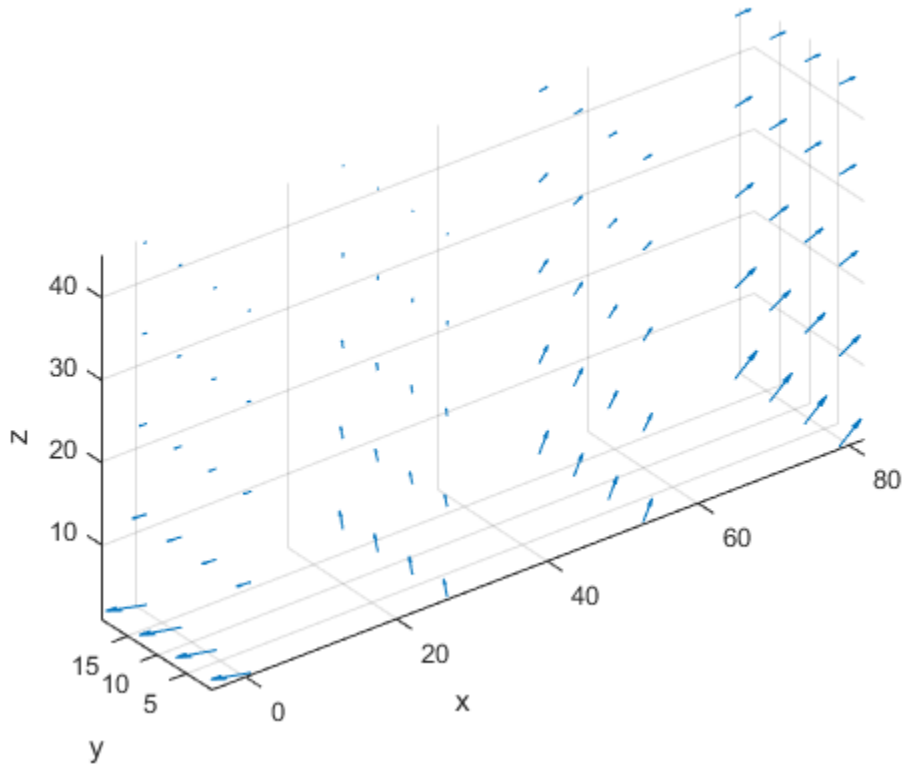


**Temperature Gradients for Transient Thermal Model on Square**

Solve a 2-D transient heat transfer problem on a square domain and compute temperature gradients at the convective boundary.

Create a transient thermal model for this problem.

```
thermalmodel = createpde('thermal','transient');
```

Create the geometry and include it in the model.

```
g = @squareg;
geometryFromEdges(thermalmodel,g);
```

```
pdegplot(thermalmodel,'EdgeLabels','on')
xlim([-1.2 1.2])
ylim([-1.2 1.2])
axis equal
```



Assign the following thermal properties: thermal conductivity is $100\,W/(m°C)$, mass density is $7800\,kg/m^3$, and specific heat is $500\,J/(kg°C)$.

```
thermalProperties(thermalmodel,'ThermalConductivity',100, ...
                               'MassDensity',7800, ...
                               'SpecificHeat',500);
```

Apply insulated boundary conditions on three edges and the free convection boundary condition on the right edge.

```
thermalBC(thermalmodel,'Edge',[1 3 4],'HeatFlux',0);
thermalBC(thermalmodel,'Edge',2, ...
                        'ConvectionCoefficient',5000, ...
                        'AmbientTemperature',25);
```

Set the initial conditions: uniform room temperature across domain and higher temperature on the left edge.

```
thermalIC(thermalmodel,25);
thermalIC(thermalmodel,100,'Edge',4);
```

Generate a mesh and solve the problem using `0:1000:200000` as a vector of times.

```
generateMesh(thermalmodel);
tlist = 0:1000:200000;
thermalresults = solve(thermalmodel,tlist)
```

```
thermalresults =

  TransientThermalResults with properties:

      Temperature: [177×201 double]
    SolutionTimes: [1×201 double]
        XGradients: [177×201 double]
        YGradients: [177×201 double]
        ZGradients: []
              Mesh: [1×1 FEMesh]
```

Define a line at convection boundary and compute temperature gradients across that line.

```
X = -1:0.1:1;
Y = ones(size(X));

[gradTx,gradTy] = evaluateTemperatureGradient(thermalresults,X,Y,1:length(tlist));
```

Plot the interpolated gradient component `gradTx` along the x axis for the following values from the time interval `tlist`.

```
figure
t = [51:50:201];
for i = t
```

```
    p(i) = plot(X,gradTx(:,i),'DisplayName', strcat('t=', num2str(tlist(i))));
    hold on
end
legend(p(t))
xlabel('x')
ylabel('gradTx')
```
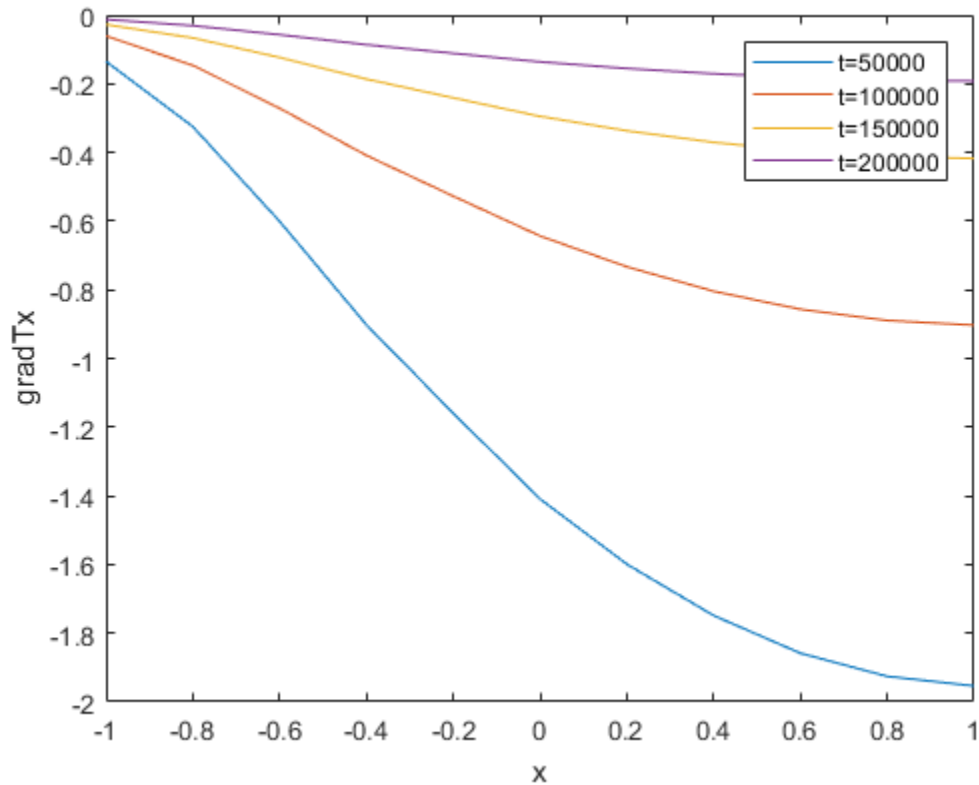


## Input Arguments

**`thermalresults` — Solution of thermal problem**
`SteadyStateThermalResults` object | `TransientThermalResults` object

Solution of a thermal problem, specified as a `SteadyStateThermalResults` object or a `TransientThermalResults` object. Create `thermalresults` using the `solve` function.

Example: `thermalresults = solve(thermalmodel)`

### xq — *x*-coordinate query points
real array

*x*-coordinate query points, specified as a real array. `evaluateTemperatureGradient` evaluates temperature gradient 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.

`evaluateTemperatureGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns the temperature gradient in a form of 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 `gradTx = reshape(gradTx,size(xq))`.

Data Types: `double`

### yq — *y*-coordinate query points
real array

*y*-coordinate query points, specified as a real array. `evaluateTemperatureGradient` evaluates the temperature gradient 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.

`evaluateTemperatureGradient` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns the temperature gradient in a form of 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 `gradTy = reshape(gradTy,size(yq))`.

Data Types: `double`

### zq — *z*-coordinate query points
real array

*z*-coordinate query points, specified as a real array. `evaluateTemperatureGradient` evaluates the temperature gradient at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. So `xq`, `yq`, and `zq` must have the same number of entries.

evaluateTemperatureGradient converts query points to column vectors xq(:), yq(:), and (if present) zq(:). It returns the temperature gradient in a form of 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 gradTz = reshape(gradTz,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. evaluateTemperatureGradient evaluates the temperature gradient 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

### iT — Time indices
vector of positive integers

Time indices, specified as a vector of positive integers. Each entry in iT specifies a time index.

Example: iT = 1:5:21 specifies every fifth time-step up to 21.

Data Types: double

## Output Arguments

### gradTx — *x*-component of the temperature gradient
matrix

*x*-component of the temperature gradient, returned as a matrix. For query points that are outside the geometry, gradTx = NaN.

### gradTy — *y*-component of the temperature gradient
matrix

*y*-component of the temperature gradient, returned as a matrix. For query points that are outside the geometry, gradTy = NaN.

**gradTz — *z*-component of the temperature gradient**
matrix

*z*-component of the temperature gradient, returned as a matrix. For query points that are outside the geometry, `gradTz = NaN`.

# See Also

## See Also
evaluateHeatFlux | evaluateHeatRate | interpolateTemperature | SteadyStateThermalResults | ThermalModel | TransientThermalResults

**Introduced in R2017a**

# 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 2-196.

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 2-196.

Data Types: `double`

# See Also

## See Also
generateMesh | meshToPet | PDEModel

## Topics
"Solve Problems Using PDEModel Objects" on page 2-5
"Finite Element Basis for 3-D" on page 5-10
"Mesh Data" on page 2-196

**Introduced in R2015a**

# findBoundaryConditions

Find boundary condition assignment for a geometric region

## Compatibility

This page describes the recommended workflow.

## Syntax

```
BCregion = findBoundaryConditions(BCs,RegionType,RegionID)
```

## Description

`BCregion = findBoundaryConditions(BCs,RegionType,RegionID)` returns boundary condition `BCregion` assigned to the specified region.

## Examples

### Find Boundary Conditions for Particular Regions

Create a PDE model and import a simple block geometry. Plot the geometry displaying the face labels.

```
model = createpde(3);
importGeometry(model,'Block.stl');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
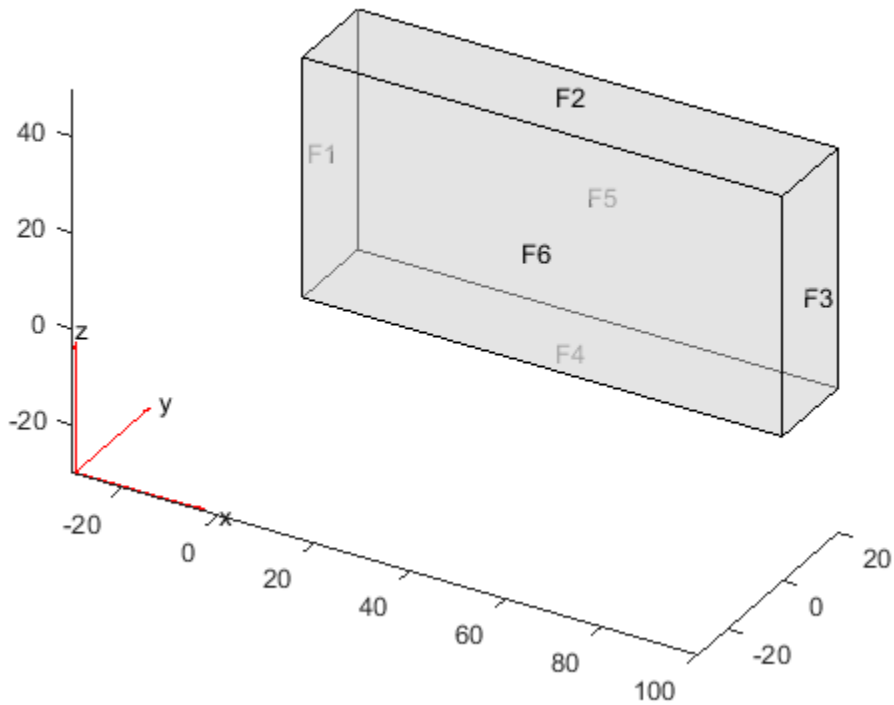
Set zero Dirichlet conditions on faces 1 and 2 for all equations.

```
applyBoundaryCondition(model,'dirichlet','Face',1:2,'u',[0,0,0]);
```

On face 3, set the Neumann boundary condition for equation 1 and Dirichlet boundary condition for equations 2 and 3.

```
h = [0 0 0;0 1 0;0 0 1];
r = [0;3;3];
q = [1 0 0;0 0 0;0 0 0];
g = [10;0;0];
applyBoundaryCondition(model,'mixed','Face',3,'h',h,'r',r,'g',g,'q',q);
```

Set Neumann boundary conditions with opposite signs on faces 5 and 6 for all equations.

```
applyBoundaryCondition(model,'neumann','Face',4:5,'g',[1;1;1]);
applyBoundaryCondition(model,'neumann','Face',6,'g',[-1;-1;-1]);
```

Check the boundary condition specification on face 1.

```
findBoundaryConditions(model.BoundaryConditions,'Face',1)
```

```
ans =

  BoundaryCondition with properties:

          BCType: 'dirichlet'
      RegionType: 'Face'
        RegionID: [1 2]
               r: []
               h: []
               g: []
               q: []
               u: [0 0 0]
   EquationIndex: []
      Vectorized: 'off'
```

Check the boundary condition specification on face 3.

```
findBoundaryConditions(model.BoundaryConditions,'Face',3)
```

```
ans =

  BoundaryCondition with properties:

          BCType: 'mixed'
      RegionType: 'Face'
        RegionID: 3
               r: [3×1 double]
               h: [3×3 double]
               g: [3×1 double]
               q: [3×3 double]
               u: []
   EquationIndex: []
      Vectorized: 'off'
```

Check the boundary condition specification on face 5.

```
findBoundaryConditions(model.BoundaryConditions,'Face',5)


ans =

  BoundaryCondition with properties:

          BCType: 'neumann'
      RegionType: 'Face'
        RegionID: [4 5]
               r: []
               h: []
               g: [3×1 double]
               q: []
               u: []
   EquationIndex: []
      Vectorized: 'off'
```

- "Solve Problems Using PDEModel Objects" on page 2-5

## Input Arguments

### BCs — Boundary conditions of a PDE model
BoundaryConditions property of a PDE model

Boundary conditions of a PDE model, specified as the BoundaryConditions property of PDEModel.

Example: model.BoundaryConditions

### RegionType — Geometric region type
'Face' for 3-D geometry | 'Edge' for 2-D geometry

Geometric region type, specified as 'Face' for 3-D geometry or 'Edge' for 2-D geometry.

Example: applyBoundaryCondition(model,'dirichlet','Face',3,'u',0)

Data Types: char

### RegionID — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot` with the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) value set to `'on'`.

Example: `applyBoundaryCondition(model,'dirichlet','Face',3:6,'u',0)`

Data Types: `double`

## Output Arguments

**`BCregion` — Boundary condition for a particular region**
`BoundaryCondition` object

Boundary condition for a particular region, returned as a `BoundaryCondition` object.

## See Also

### See Also
`applyBoundaryCondition` | BoundaryCondition Properties

### Topics
"Solve Problems Using PDEModel Objects" on page 2-5

**Introduced in R2016b**

# findCoefficients

Locate active PDE coefficients

## Compatibility

This page describes the recommended workflow.

## 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,'FaceLabels','on')
ylim([-1.1,1.1])
axis equal
```
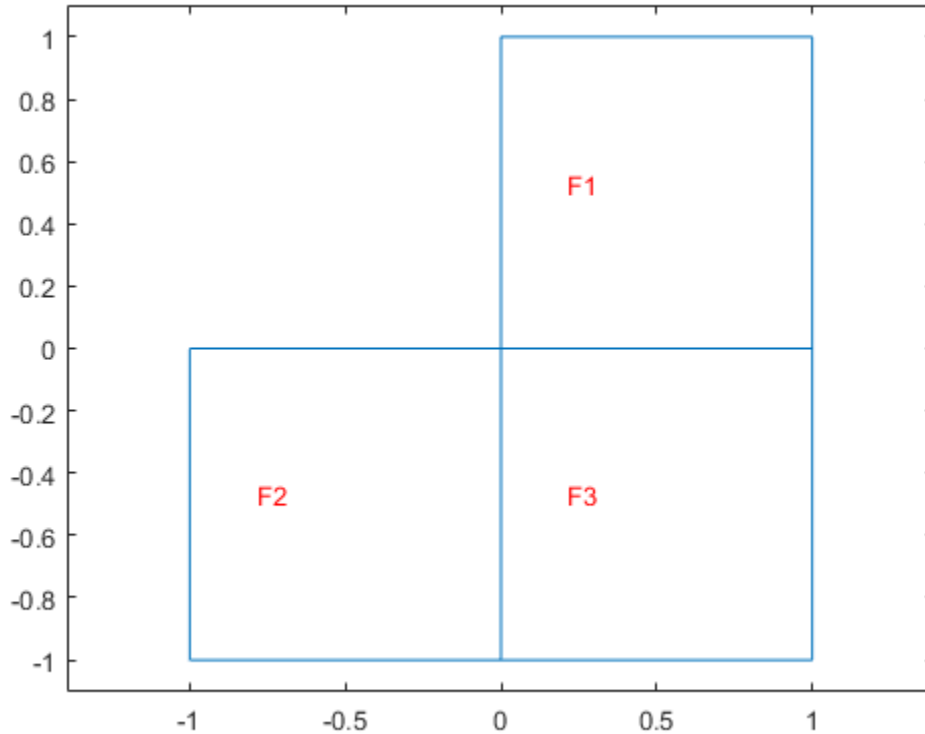
Set coefficients on each pair of regions.

```
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 2-136
- "Solve Problems Using PDEModel Objects" on page 2-5

## 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,'FaceLabels','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.

## See Also

### See Also
CoefficientAssignment Properties | specifyCoefficients

### Topics
"View, Edit, and Delete PDE Coefficients" on page 2-136
"Solve Problems Using PDEModel Objects" on page 2-5
"PDE Coefficients"

**Introduced in R2016a**

# findHeatSource

Find heat source assigned to a geometric region

## Syntax

```
hsa = findHeatSource(thermalmodel.HeatSources,RegionType,RegionID)
```

## Description

`hsa = findHeatSource(thermalmodel.HeatSources,RegionType,RegionID)` returns the heat source value `hsa` assigned to the specified region.

## Examples

### Find Heat Sources for Faces of 2-D Geometry

Create a thermal model that has three faces.

```
thermalmodel = createpde('thermal');
geometryFromEdges(thermalmodel,@lshapeg);
pdegplot(thermalmodel,'FaceLabels','on')
ylim([-1.1 1.1])
axis equal
```

Specify that face 1 generates heat at 10 W/m^3, face 2 generates heat at 20 W/m^3, and face 3 generates heat at 30 W/m^3.

```
internalHeatSource(thermalmodel,10,'Face',1);
internalHeatSource(thermalmodel,20,'Face',2);
internalHeatSource(thermalmodel,30,'Face',3);
```

Check the heat source specification for face 1.

```
hsaFace1 = findHeatSource(thermalmodel.HeatSources,'Face',1)
```

```
hsaFace1 =

  HeatSourceAssignment with properties:
```

```
        RegionType: 'face'
          RegionID: 1
        HeatSource: 10
```

Check the heat source specification for faces 2 and 3.

```
hsa = findHeatSource(thermalmodel.HeatSources,'Face',[2 3]);
hsaFace2 = hsa(1)
hsaFace3 = hsa(2)


hsaFace2 =

  HeatSourceAssignment with properties:

        RegionType: 'face'
          RegionID: 2
        HeatSource: 20


hsaFace3 =

  HeatSourceAssignment with properties:

        RegionType: 'face'
          RegionID: 3
        HeatSource: 30
```

### Find Heat Sources for Cells of 3-D Geometry

Create a geometry that consists of three stacked cylinders and include the geometry in a thermal model.

```
gm = multicylinder(10,[1 2 3],'ZOffset',[0 1 3])
thermalmodel = createpde('thermal');
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,'CellLabels','on','FaceAlpha',0.5)


gm =

  DiscreteGeometry with properties:
```

```
    NumCells: 3
    NumFaces: 7
    NumEdges: 4
 NumVertices: 4
```
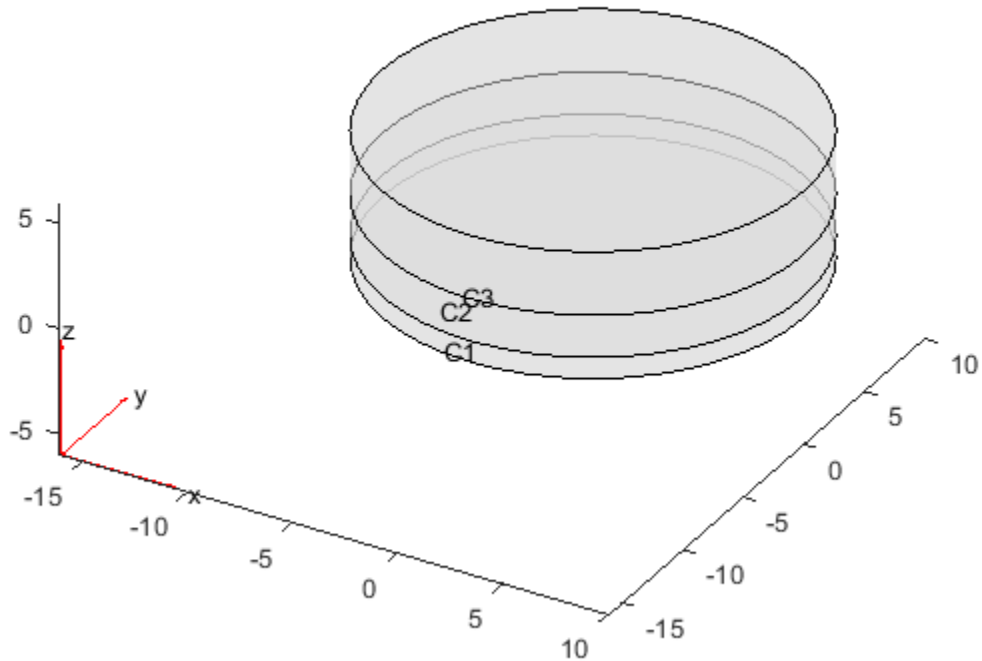


Specify that the cylinder C1 generates heat at $10\ W/m^3$, the cylinder C2 generates heat at $20\ W/m^3$, and the cylinder C3 generates heat at $30\ W/m^3$.

```
internalHeatSource(thermalmodel,10,'Cell',1);
internalHeatSource(thermalmodel,20,'Cell',2);
internalHeatSource(thermalmodel,30,'Cell',3);
```

Check the heat source specification for cell 1.

```
hsaCell1 = findHeatSource(thermalmodel.HeatSources,'Cell',1)
```

```
hsaCell1 =

  HeatSourceAssignment with properties:

    RegionType: 'cell'
      RegionID: 1
    HeatSource: 10
```

Check the heat source specification for cells 2 and 3.

```
hsa = findHeatSource(thermalmodel.HeatSources,'Cell',[2:3]);
hsaCell2 = hsa(1)
hsaCell3 = hsa(2)
```

```
hsaCell2 =

  HeatSourceAssignment with properties:

    RegionType: 'cell'
      RegionID: 2
    HeatSource: 20
```

```
hsaCell3 =

  HeatSourceAssignment with properties:

    RegionType: 'cell'
      RegionID: 3
    HeatSource: 30
```

## Input Arguments

**`thermalmodel.HeatSources` — Internal heat source of the model**
`HeatSources` property of a thermal model

Internal heat source of the model, specified as the `HeatSources` property of a `ThermalModel` object.

### `RegionType` — Geometric region type
`'Face'` | `'Cell'`

Geometric region type, specified as `'Face'` for a 2-D model or `'Cell'` for a 3-D model.

Data Types: `char`

### `RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using the `pdegplot` function, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Data Types: `double`

## Output Arguments

### `hsa` — Heat source assignment
`HeatSourceAssignment` object

Heat source assignment, returned as a `HeatSourceAssignment` object.

## See Also

### See Also
HeatSourceAssignment Properties | `internalHeatSource`

**Introduced in R2017a**

# findInitialConditions

Locate active initial conditions

## Compatibility

This page describes the recommended workflow.

## 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

This example shows find the active initial conditions for a region.

Create a PDE model that has a few subdomains.

```
model = createpde();
geometryFromEdges(model,@lshapeg);
pdegplot(model,'FaceLabels','on')
ylim([-1.1,1.1])
axis equal
```
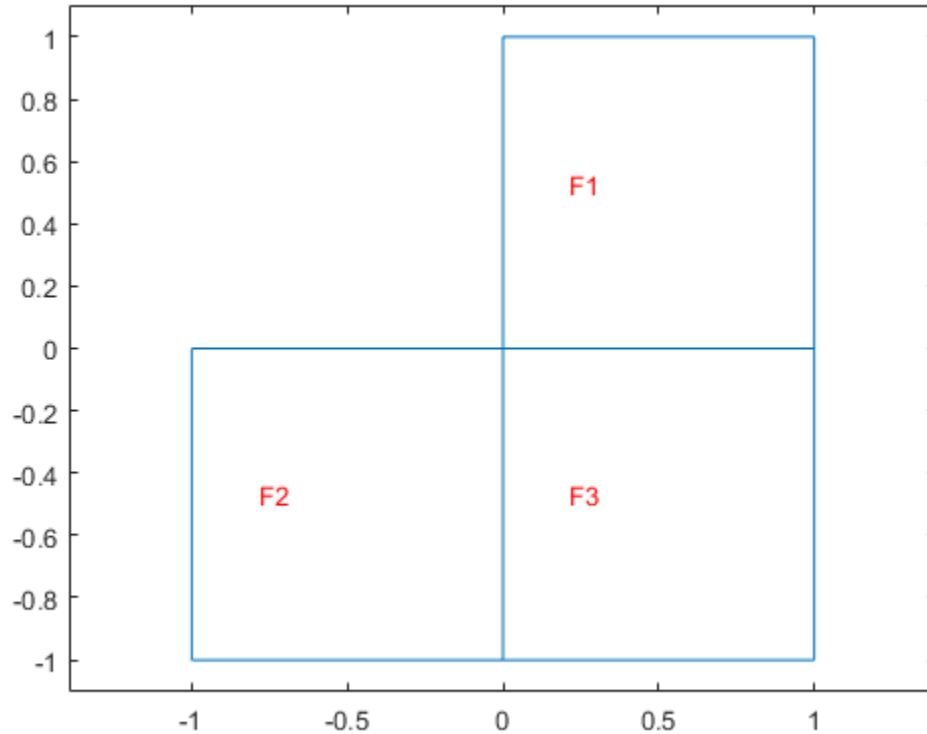
Set initial conditions on each pair of regions.

```
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 2-143
- "Solve Problems Using PDEModel Objects" on page 2-5

## 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,'FaceLabels','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 | NodalInitialConditions object

Initial condition assignment, returned as a `GeometricInitialConditions` or `NodalInitialConditions` object.

## See Also

### See Also
GeometricInitialConditions Properties | `NodalInitialConditions` | `setInitialConditions`

### Topics
"View, Edit, and Delete Initial Conditions" on page 2-143
"Solve Problems Using PDEModel Objects" on page 2-5
"Initial Conditions"

**Introduced in R2016a**

# findThermalBC

Find thermal boundary conditions assigned to a geometric region

## Syntax

```
tbca = findThermalBC(thermalmodel.BoundaryConditions,RegionType,
RegionID)
```

## Description

`tbca = findThermalBC(thermalmodel.BoundaryConditions,RegionType, RegionID)` returns the thermal boundary condition assigned to the specified region.

## Examples

### Find Thermal Boundary Conditions for Edges of 2-D Geometry

Create a thermal model and include a square geometry.

```
thermalmodel = createpde('thermal');
geometryFromEdges(thermalmodel,@squareg);
pdegplot(thermalmodel,'EdgeLabels','on')
ylim([-1.1 1.1])
axis equal
```
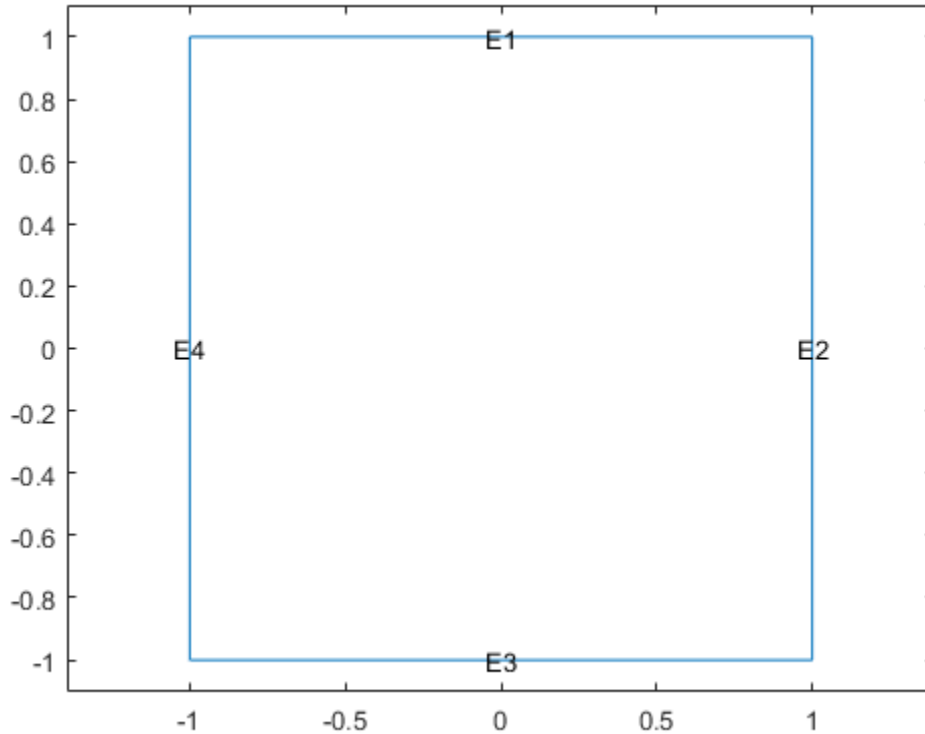
Apply temperature boundary conditions on edges 1 and 3 of the square.

```
thermalBC(thermalmodel,'Edge',[1 3],'Temperature',100);
```

Apply a heat flux boundary condition on edge 4 of the square.

```
thermalBC(thermalmodel,'Edge',4,'HeatFlux',20);
```

Check the boundary condition specification on edge 1.

```
tbcaEdge1 = findThermalBC(thermalmodel.BoundaryConditions,'Edge',1)
```

```
tbcaEdge1 =
```

```
  ThermalBC with properties:

                RegionType: 'Edge'
                  RegionID: [1 3]
               Temperature: 100
                  HeatFlux: []
     ConvectionCoefficient: []
                Emissivity: []
        AmbientTemperature: []
                Vectorized: 'off'
```

Check the boundary condition specifications on edges 3 and 4.

```
tbca = findThermalBC(thermalmodel.BoundaryConditions,'Edge',[3:4]);
tbcaEdge3 = tbca(1)
tbcaEdge4 = tbca(2)


tbcaEdge3 =

  ThermalBC with properties:

                RegionType: 'Edge'
                  RegionID: [1 3]
               Temperature: 100
                  HeatFlux: []
     ConvectionCoefficient: []
                Emissivity: []
        AmbientTemperature: []
                Vectorized: 'off'


tbcaEdge4 =

  ThermalBC with properties:

                RegionType: 'Edge'
                  RegionID: 4
               Temperature: []
                  HeatFlux: 20
     ConvectionCoefficient: []
                Emissivity: []
        AmbientTemperature: []
                Vectorized: 'off'
```

### Find Thermal Boundary Conditions for Faces of 3-D Geometry

Create a thermal model and include a block geometry.

```
thermalmodel = createpde('thermal','transient');
gm = importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabels','on','FaceAlpha',0.5)
```
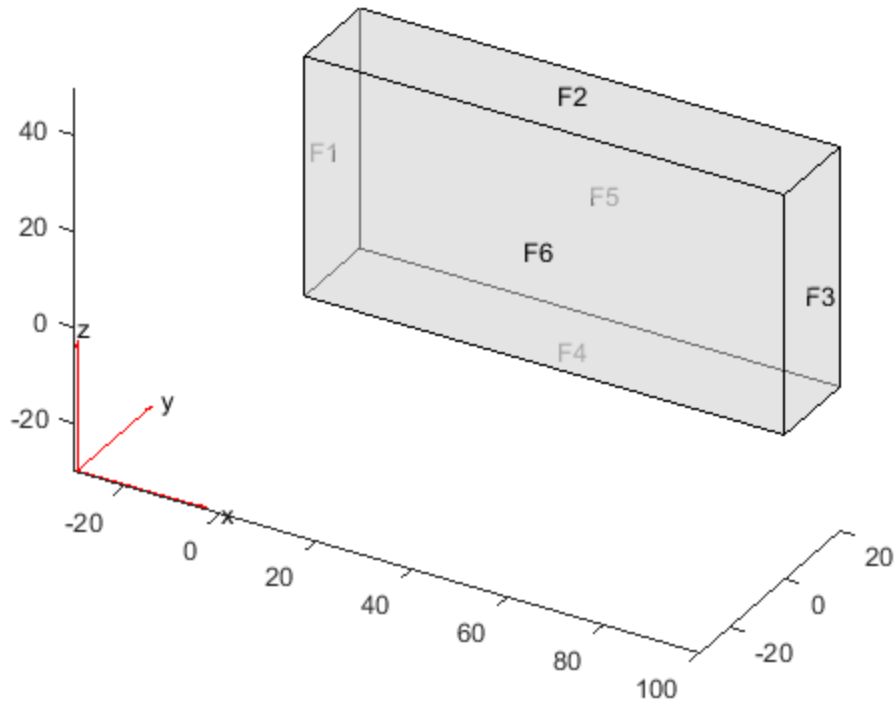


Apply temperature boundary condition on faces 1 and 3 of a block.

```
thermalBC(thermalmodel,'Face',1,'Temperature',100);
thermalBC(thermalmodel,'Face',3,'Temperature',300);
```

Apply convection boundary condition on faces 5 and 6 of a block.

```
thermalBC(thermalmodel,'Face',[5,6],...
                        'ConvectionCoefficient',5,...
                        'AmbientTemperature',27);
```

Check the boundary condition specification on faces 1 and 3.

```
tbca = findThermalBC(thermalmodel.BoundaryConditions,'Face',[1,3]);
tbcaFace1 = tbca(1)
tbcaFace3 = tbca(2)


tbcaFace1 =

  ThermalBC with properties:

                RegionType: 'Face'
                  RegionID: 1
               Temperature: 100
                  HeatFlux: []
     ConvectionCoefficient: []
                 Emissivity: []
         AmbientTemperature: []
                Vectorized: 'off'


tbcaFace3 =

  ThermalBC with properties:

                RegionType: 'Face'
                  RegionID: 3
               Temperature: 300
                  HeatFlux: []
     ConvectionCoefficient: []
                 Emissivity: []
         AmbientTemperature: []
                Vectorized: 'off'
```

Check the boundary condition specifications on faces 5 and 6.

```
tbcaFace5 = findThermalBC(thermalmodel.BoundaryConditions,'Face',5)
```

```
tbcaFace5 =

  ThermalBC with properties:

                RegionType: 'Face'
                  RegionID: [5 6]
               Temperature: []
                  HeatFlux: []
     ConvectionCoefficient: 5
                 Emissivity: []
        AmbientTemperature: 27
                Vectorized: 'off'


tbcaFace6 = findThermalBC(thermalmodel.BoundaryConditions,'Face',6)


tbcaFace6 =

  ThermalBC with properties:

                RegionType: 'Face'
                  RegionID: [5 6]
               Temperature: []
                  HeatFlux: []
     ConvectionCoefficient: 5
                 Emissivity: []
        AmbientTemperature: 27
                Vectorized: 'off'
```

## Input Arguments

**thermalmodel.BoundaryConditions — Boundary conditions of a thermal model**
BoundaryConditions property of a thermal model

Boundary conditions of a thermal model, specified as the BoundaryConditions property of a ThermalModel object.

Example: thermalmodel.BoundaryConditions

**RegionType — Geometric region type**
'Face' | 'Edge'

Geometric region type, specified as `'Face'` for 3-D geometry or `'Edge'` for 2-D geometry.

Data Types: `char`

### RegionID — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot` with the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) value set to `'on'`.

Data Types: `double`

## Output Arguments

### tbca — Thermal boundary condition for a particular region
`ThermalBC` object

Thermal boundary condition for a particular region, returned as a `ThermalBC` object.

## See Also

### See Also
ThermalBC Properties | `thermalBC`

**Introduced in R2017a**

# findThermalIC

Find thermal initial conditions assigned to a geometric region

## Syntax

```
tica = findThermalIC(thermalmodel.InitialConditions,RegionType,
RegionID)
```

## Description

```
tica = findThermalIC(thermalmodel.InitialConditions,RegionType,
RegionID)
```
 returns the thermal initial condition assigned to the specified region.

## Examples

### Find Initial Temperatures for Faces of 2-D Geometry

Create a transient thermal model that has three faces.

```
thermalmodel = createpde('thermal','transient');
geometryFromEdges(thermalmodel,@lshapeg);
pdegplot(thermalmodel,'FaceLabels','on')
ylim([-1.1 1.1])
axis equal
```
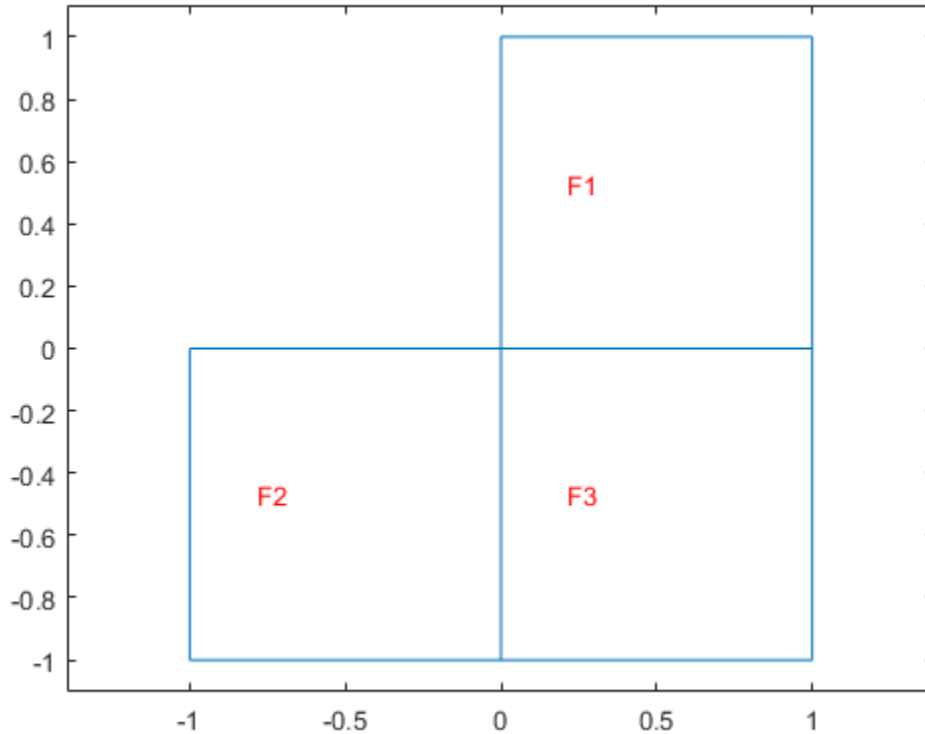
Set initial temperatures for each face.

```
thermalIC(thermalmodel,10,'Face',1);
thermalIC(thermalmodel,20,'Face',2);
thermalIC(thermalmodel,30,'Face',3);
```

Check the initial condition specification for face 1.

```
ticaFace1 = findThermalIC(thermalmodel.InitialConditions,'Face',1)


ticaFace1 =

  GeometricThermalICs with properties:
```

```
          RegionType: 'face'
            RegionID: 1
   InitialTemperature: 10
```

Check the initial temperature specifications for faces 2 and 3.

```
tica = findThermalIC(thermalmodel.InitialConditions,'Face',[2 3]);
ticaFace2 = tica(1)
```

```
ticaFace2 =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: 2
     InitialTemperature: 20
```

```
ticaFace3 = tica(2)
```

```
ticaFace3 =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: 3
     InitialTemperature: 30
```

### Find Initial Temperatures for Cells of 3-D Geometry

Create a geometry that consists of three nested cylinders and include the geometry in a transient thermal model.

```
gm = multicylinder([5 10 15],2)
thermalmodel = createpde('thermal','transient');
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,'CellLabels','on','FaceAlpha',0.5)
```

```
gm =

  DiscreteGeometry with properties:
```

```
        NumCells: 3
        NumFaces: 9
        NumEdges: 6
     NumVertices: 6
```
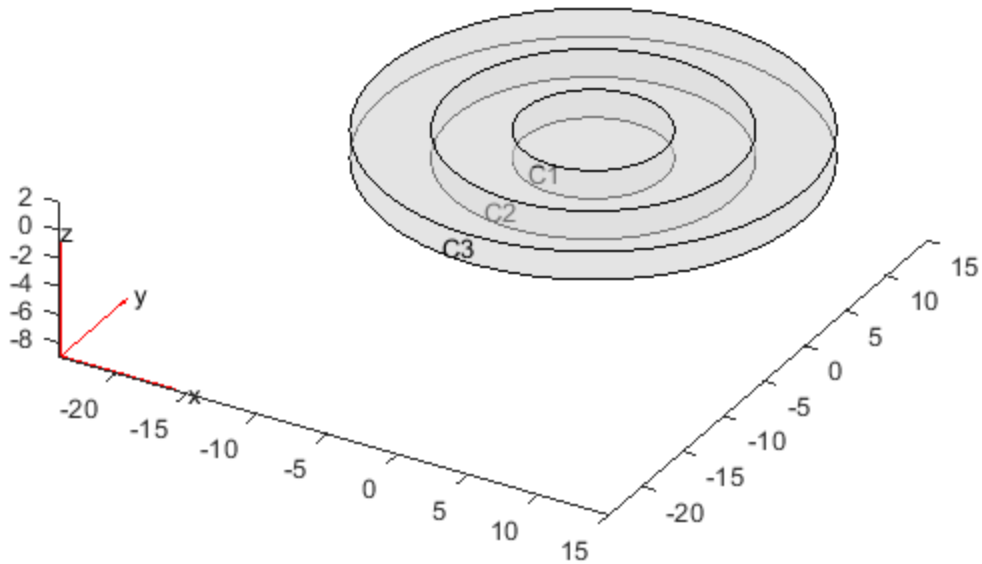


Set initial temperatures for each cell.

```
thermalIC(thermalmodel,0,'Cell',1);
thermalIC(thermalmodel,100,'Cell',2);
thermalIC(thermalmodel,0,'Cell',3);
```

Check the initial condition specification for cell 1.

```
ticaCell1 = findThermalIC(thermalmodel.InitialConditions,'Cell',1)
```

```
ticaCell1 =

  GeometricThermalICs with properties:

             RegionType: 'cell'
               RegionID: 1
      InitialTemperature: 0
```

Check the initial condition specification for cells 2 and 3.

```
tica = findThermalIC(thermalmodel.InitialConditions,'Cell',[2:3]);
ticaCell2 = tica(1)
```

```
ticaCell2 =

  GeometricThermalICs with properties:

             RegionType: 'cell'
               RegionID: 2
      InitialTemperature: 100
```

```
ticaCell3 = tica(2)
```

```
ticaCell3 =

  GeometricThermalICs with properties:

             RegionType: 'cell'
               RegionID: 3
      InitialTemperature: 0
```

### Find Initial Temperature Set by Using Previously Obtained Solution

Create a thermal model and include a square geometry.

```
thermalmodel = createpde('thermal','transient');
gm = @squareg;
geometryFromEdges(thermalmodel,gm);
pdegplot(thermalmodel,'FaceLabels','on')
ylim([-1.1 1.1])
```

```
axis equal
```
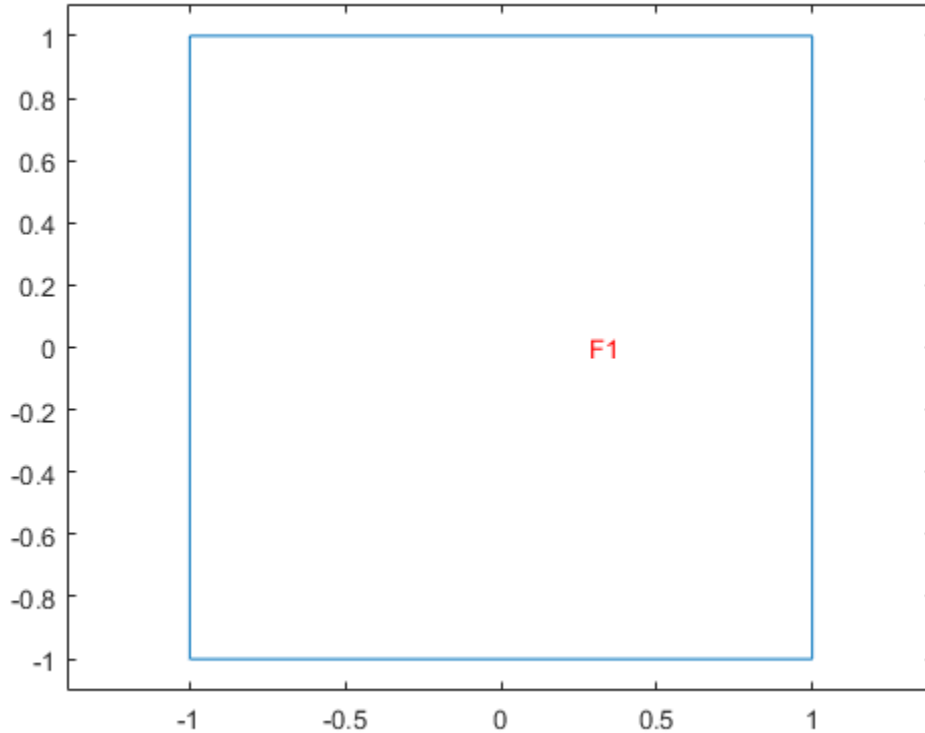


Specify material properties, heat source, set initial and boundary conditions.

```
thermalProperties(thermalmodel,'ThermalConductivity',500,...
                               'MassDensity',200,...
                               'SpecificHeat',100);
internalHeatSource(thermalmodel,2);
thermalBC(thermalmodel,'Edge',[1 3],'Temperature',100);
thermalIC(thermalmodel,0);
```

Generate a mesh and solve the problem.

```
generateMesh(thermalmodel);
tlist = 0:0.5:10;
```

```
result1 = solve(thermalmodel,tlist)


result1 =

  TransientThermalResults with properties:

      Temperature: [177×21 double]
    SolutionTimes: [1×21 double]
       XGradients: [177×21 double]
       YGradients: [177×21 double]
       ZGradients: []
             Mesh: [1×1 FEMesh]
```

Check the currently active initial temperature specification.

```
tica = findThermalIC(thermalmodel.InitialConditions,'Face',1)


tica =

  GeometricThermalICs with properties:

             RegionType: 'face'
               RegionID: 1
     InitialTemperature: 0
```

Now, resume the analysis and solve the problem for times from 10 to 15 seconds. Use the previously obtained solution for 10 seconds as an initial condition. Since 10 seconds is the last element in tlist, you do not need to specify the solution time index. By default, thermalIC uses the last solution index.

```
ic = thermalIC(thermalmodel,result1);
```

Solve the problem

```
tlist = 10:0.5:15;
result2 = solve(thermalmodel,tlist);
```

Check the currently active initial temperature specification.

```
tica = findThermalIC(thermalmodel.InitialConditions,'Face',1)
```

```
tica =

  NodalThermalICs with properties:

    InitialTemperature: [177×1 double]
```

```
pdeplot(thermalmodel,'XYData',tica.InitialTemperature)
```
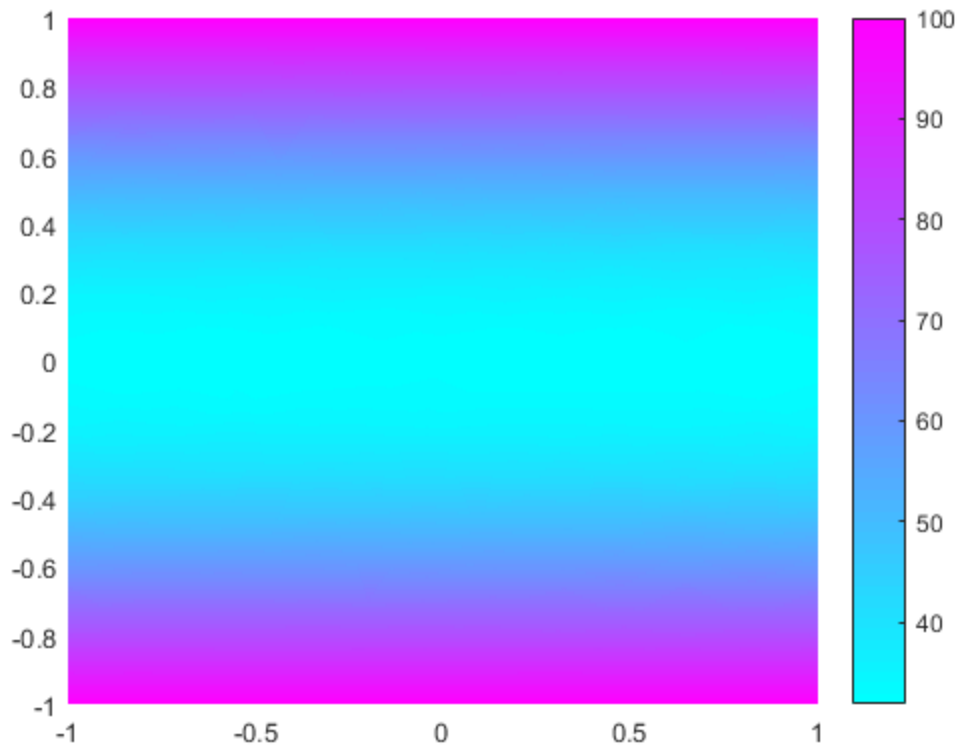


## Input Arguments

**`thermalmodel.InitialConditions`** — Initial conditions of a thermal model
`InitialConditions` property of a thermal model

Initial conditions of a thermal model, specified as the `InitialConditions` property of a `ThermalModel` object.

**`RegionType` — Geometric region type**
`'Edge'` | `'Face'` | `'Vertex'` | `'Cell'` for a 3-D model

Geometric region type, specified as `'Edge'`, `'Face'`, or `'Vertex'` for a 2-D model or 3-D model, or `'Cell'` for a 3-D model.

Data Types: `char`

**`RegionID` — Geometric region ID**
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using the `pdegplot` function with the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) value set to `'on'`.

Data Types: `double`

# Output Arguments

**`tica` — Thermal initial condition for a particular region**
GeometricThermalIC object | NodalThermalIC object

Thermal initial condition for a particular region, returned as a GeometricThermalICs Properties or NodalThermalICs Properties object.

# See Also

## See Also
GeometricThermalICs Properties | NodalThermalICs Properties | `thermalIC`

**Introduced in R2017a**

# findThermalProperties

Find material properties assigned to a geometric region

## Syntax

```
mpa = findThermalProperties(thermalmodel.MaterialProperties,
RegionType,RegionID)
```

## Description

`mpa = findThermalProperties(thermalmodel.MaterialProperties,`
`RegionType,RegionID)` returns material properties `mpa` assigned to the specified region.

## Examples

### Find Thermal Conductivity, Mass Density, and Specific Heat for Faces of 2-D Geometry

Create a transient thermal model that has three faces.

```
thermalmodel = createpde('thermal','transient');
geometryFromEdges(thermalmodel,@lshapeg);
pdegplot(thermalmodel,'FaceLabels','on')
ylim([-1.1,1.1])
axis equal
```

For face 1, specify the following thermal properties:

- Thermal conductivity is 10 W/(m*C)
- Mass density is 1 kg/m^3
- Specific heat is 0.1 J/(kg*C)

```
thermalProperties(thermalmodel,'ThermalConductivity',10,...
                               'MassDensity',1,...
                               'SpecificHeat',0.1,...
                               'Face',1);
```

For face 2, specify the following thermal properties:

- Thermal conductivity is 20 W/(m*C)

- Mass density is 2 kg/m^3

- Specific heat is 0.2 J/(kg*C)

```
thermalProperties(thermalmodel,'ThermalConductivity',20,...
                               'MassDensity',2,...
                               'SpecificHeat',0.2,...
                               'Face',2);
```

For face 1, specify the following thermal properties: thermal conductivity is 30 W/(m*C), mass density is 3 kg/m^3, specific heat is 0.3 J/(kg*C).

- Thermal conductivity is 30 W/(m*C)

- Mass density is 3 kg/m^3

- Specific heat is 0.3 J/(kg*C)

```
thermalProperties(thermalmodel,'ThermalConductivity',30,...
                               'MassDensity',3,...
                               'SpecificHeat',0.3,...
                               'Face',3);
```

Check the material properties specification for face 1.

```
mpaFace1 = findThermalProperties(thermalmodel.MaterialProperties,'Face',1)


mpaFace1 =

  ThermalMaterialAssignment with properties:

              RegionType: 'face'
                RegionID: 1
    ThermalConductivity: 10
             MassDensity: 1
            SpecificHeat: 0.1000
```

Check the heat source specification for faces 2 and 3.

```
mpa = findThermalProperties(thermalmodel.MaterialProperties,'Face',[2,3]);
mpaFace2 = mpa(1)
mpaFace3 = mpa(2)
```

```
mpaFace2 =

  ThermalMaterialAssignment with properties:

             RegionType: 'face'
               RegionID: 2
    ThermalConductivity: 20
            MassDensity: 2
           SpecificHeat: 0.2000


mpaFace3 =

  ThermalMaterialAssignment with properties:

             RegionType: 'face'
               RegionID: 3
    ThermalConductivity: 30
            MassDensity: 3
           SpecificHeat: 0.3000
```

### Find Thermal Conductivity for Cells of 3-D Geometry

Create a geometry that consists of three stacked cylinders and include the geometry in a thermal model.

```
gm = multicylinder(10,[1 2 3],'ZOffset',[0 1 3])
thermalmodel = createpde('thermal');
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,'CellLabels','on','FaceAlpha',0.5)


gm =

  DiscreteGeometry with properties:

        NumCells: 3
        NumFaces: 7
        NumEdges: 4
     NumVertices: 4
```
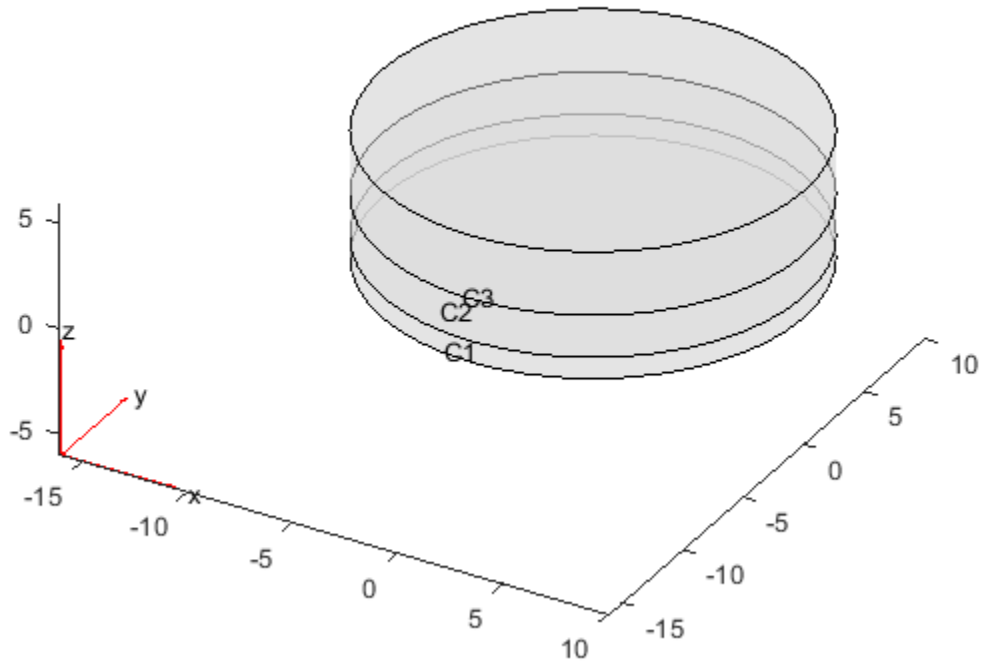
Thermal conductivity of the cylinder `C1` is 10 W/(m*C).

```
thermalProperties(thermalmodel,'ThermalConductivity',10,'Cell',1);
```

Thermal conductivity of the cylinder `C2` is 20 W/(m*C).

```
thermalProperties(thermalmodel,'ThermalConductivity',20,'Cell',2);
```

Thermal conductivity of the cylinder `C3` is 30 W/(m*C).

```
thermalProperties(thermalmodel,'ThermalConductivity',30,'Cell',3);
```

Check the material properties specification for cell 1:

```
mpaCell1 = findThermalProperties(thermalmodel.MaterialProperties,'Cell',1)
```

```
mpaCell1 =

  ThermalMaterialAssignment with properties:

              RegionType: 'cell'
                RegionID: 1
    ThermalConductivity: 10
             MassDensity: []
            SpecificHeat: []
```

Check the heat source specification for cells 2 and 3:

```
mpa = findThermalProperties(thermalmodel.MaterialProperties,'Cell',2:3);
mpaCell2 = mpa(1)
mpaCell3 = mpa(2)
```

```
mpaCell2 =

  ThermalMaterialAssignment with properties:

              RegionType: 'cell'
                RegionID: 2
    ThermalConductivity: 20
             MassDensity: []
            SpecificHeat: []
```

```
mpaCell3 =

  ThermalMaterialAssignment with properties:

              RegionType: 'cell'
                RegionID: 3
    ThermalConductivity: 30
             MassDensity: []
            SpecificHeat: []
```

## Input Arguments

**`thermalmodel.MaterialProperties` — Material properties of the model**
`MaterialProperties` property of a thermal model

Material properties of the model, specified as the `MaterialProperties` property of a thermal model.

Example: `thermalmodel.MaterialProperties`

**`RegionType` — Geometric region type**
`'Face'` for a 2-D model | `'Cell'` for a 3-D model

Geometric region type, specified as `'Face'` or `'Cell'`.

Example:
`findThermalProperties(thermalmodel.MaterialProperties,'Cell',1)`

Data Types: `char`

**`RegionID` — Geometric region ID**
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot`, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Example:
`findThermalProperties(thermalmodel.MaterialProperties,'Face',1:3)`

Data Types: `double`

## Output Arguments

**`mpa` — Material properties assignment**
`ThermalMaterialAssignment` object

Material properties assignment, returned as a `ThermalMaterialAssignment` object.

# See Also

## See Also

ThermalMaterialAssignment Properties | `thermalProperties`

**Introduced in R2017a**

# 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` might return slightly different meshes in different releases. For example, the number of elements in the mesh can change. Avoid writing code that relies on explicitly specified node and element IDs.

`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)
```
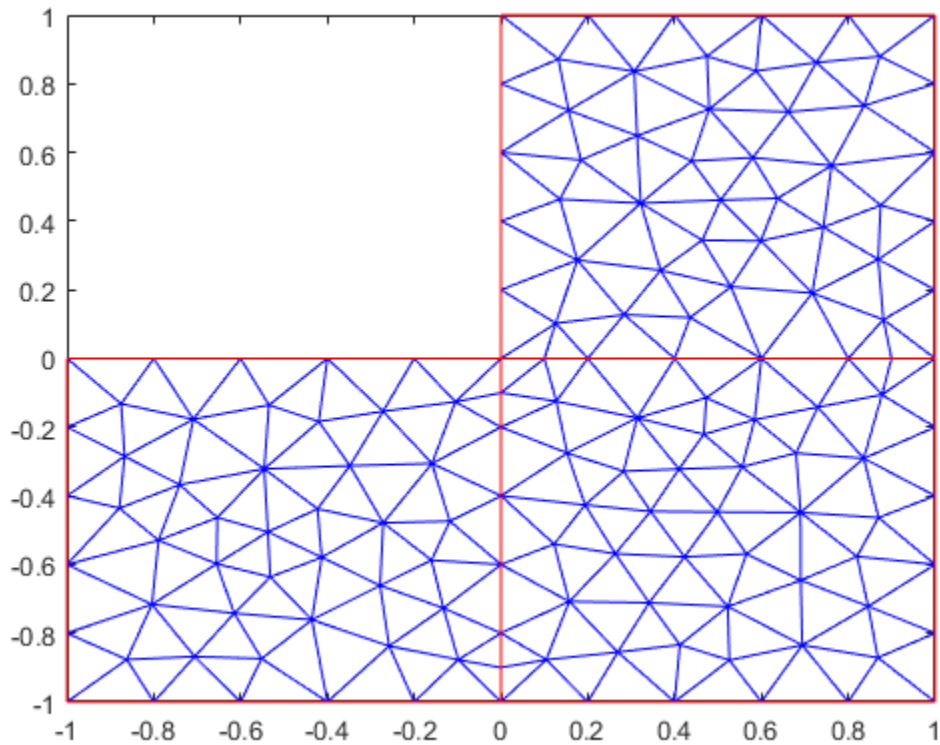


### Generate 3-D Mesh

Create a mesh that is finer than the default.

Create a PDE model and include the `BracketTwoHoles` geometry.

```
model = createpde(1);
importGeometry(model,'BracketTwoHoles.stl');
```

Generate a default mesh for comparison.

```
generateMesh(model)
```

```
ans =

  FEMesh with properties:

             Nodes: [3×21210 double]
          Elements: [10×12937 double]
    MaxElementSize: 7.3485
    MinElementSize: 2.9394
    GeometricOrder: 'quadratic'
```

View the mesh.

```
pdeplot3D(model)
```
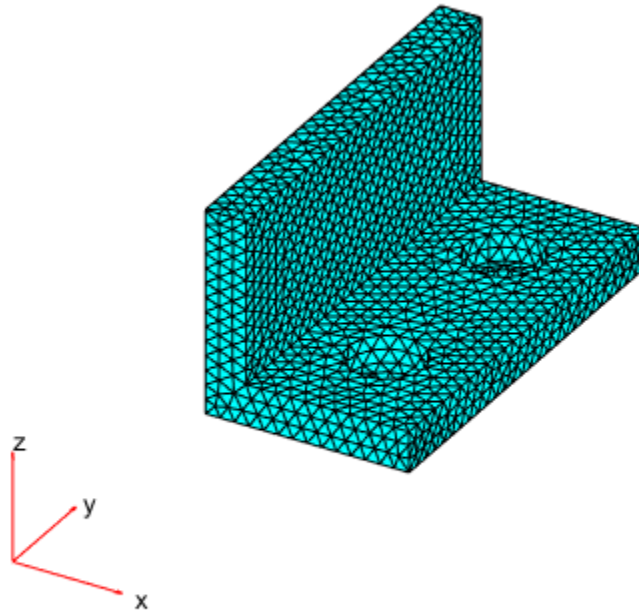
Create a mesh with target maximum element size 5 instead of the default 7.3485.

```
generateMesh(model,'Hmax',5)
```

```
ans =

  FEMesh with properties:

             Nodes: [3×65926 double]
          Elements: [10×43212 double]
     MaxElementSize: 5
     MinElementSize: 2
     GeometricOrder: 'quadratic'
```
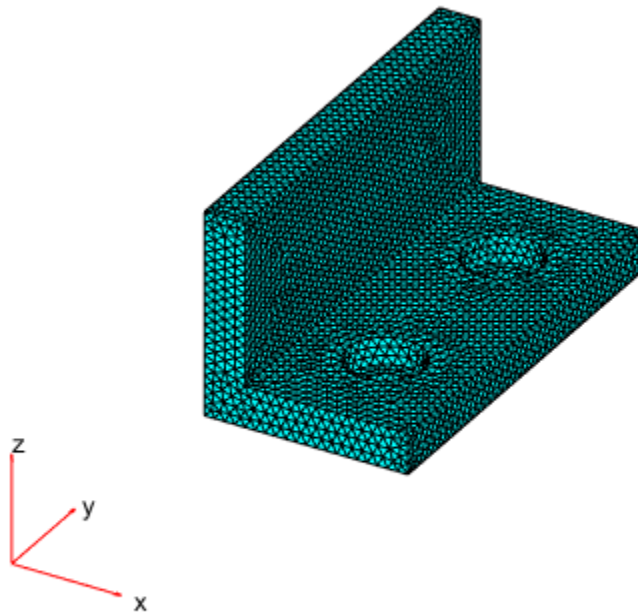
View the mesh.

```
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 `'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 `'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`.

## Definitions

### 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 2-196 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 2-196.

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.

# See Also

## See Also
FEMesh Properties | geometryFromEdges | importGeometry | PDEModel

## Topics
"Solve Problems Using PDEModel Objects" on page 2-5
"Finite Element Basis for 3-D" on page 5-10
"Mesh Data" on page 2-196

**Introduced in R2015a**

# GeometricInitialConditions Properties

Initial conditions over a region or region boundary

## Compatibility

This page describes the recommended workflow.

## Description

A `GeometricInitialConditions` object contains a description of the initial conditions over a geometric region or boundary of the region. 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 `'FacenLabels'` 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

## See Also
`findInitialConditions` | NodalInitialConditions Properties | `setInitialConditions`

## Topics
"Set Initial Conditions" on page 2-140
"View, Edit, and Delete Initial Conditions" on page 2-143
"Solve Problems Using PDEModel Objects" on page 2-5

**Introduced in R2016a**

# GeometricThermalICs Properties

Initial temperature over a region or region boundary

## Description

A `GeometricThermalICs` object contains a description of the initial temperature over a geometric region or a boundary of the region. A `ThermalModel` container has a vector of `GeometricThermalICs` objects in its `InitialConditions.ThermalICAssignments` property.

Set initial conditions for your model using the `thermalIC` function.

## Properties

### `RegionType` — Region type
`'Vertex'` | `'Edge'` | `'Face'` | `'Cell'`

Region type, returned as `'Vertex'`, `'Edge'`, or `'Face'` for a 2-D or 3-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 and setting the `'FaceLabels'` name-value pair to `'on'`.

Data Types: `double`

### `InitialTemperature` — Initial temperature
scalar | vector | function handle

Initial temperature, returned as a scalar, vector, or function handle. For details, see `thermalIC`.

Data Types: `double` | `function_handle`

## See Also

### See Also
findThermalIC | NodalThermalICs Properties | thermalIC

**Introduced in R2017a**

# NodalInitialConditions Properties

Initial conditions at mesh nodes

## Compatibility

This page describes the recommended workflow.

## Description

A `NodalInitialConditions` object contains a description of the initial conditions at mesh nodes. A `PDEModel` container has a vector of `NodalInitialConditions` objects in its `InitialConditions.InitialConditionAssignments` property.

Set initial conditions for your model using the `setInitialConditions` function.

### Properties

**`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

## See Also
findInitialConditions | GeometricInitialConditions Properties |
setInitialConditions

## Topics
"Set Initial Conditions" on page 2-140
"View, Edit, and Delete Initial Conditions" on page 2-143
"Solve Problems Using PDEModel Objects" on page 2-5

**Introduced in R2016b**

# NodalThermalICs Properties

Initial temperature at mesh nodes

## Description

A `NodalThermalICs` object contains a description of the initial temperatures at mesh nodes. A `ThermalModel` container has a vector of `NodalThermalICs` objects in its `InitialConditions.ThermalICAssignments` property.

Set initial conditions for your model using the `thermalIC` function.

## Properties

### `InitialTemperature` — Initial temperature
scalar | vector | function handle

Initial temperature, returned as a scalar, vector, or function handle. For details, see `thermalIC`.

Data Types: `double` | `function_handle`

## See Also

### See Also
`findThermalIC` | GeometricThermalICs Properties | `thermalIC`

**Introduced in R2017a**

# 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])
```
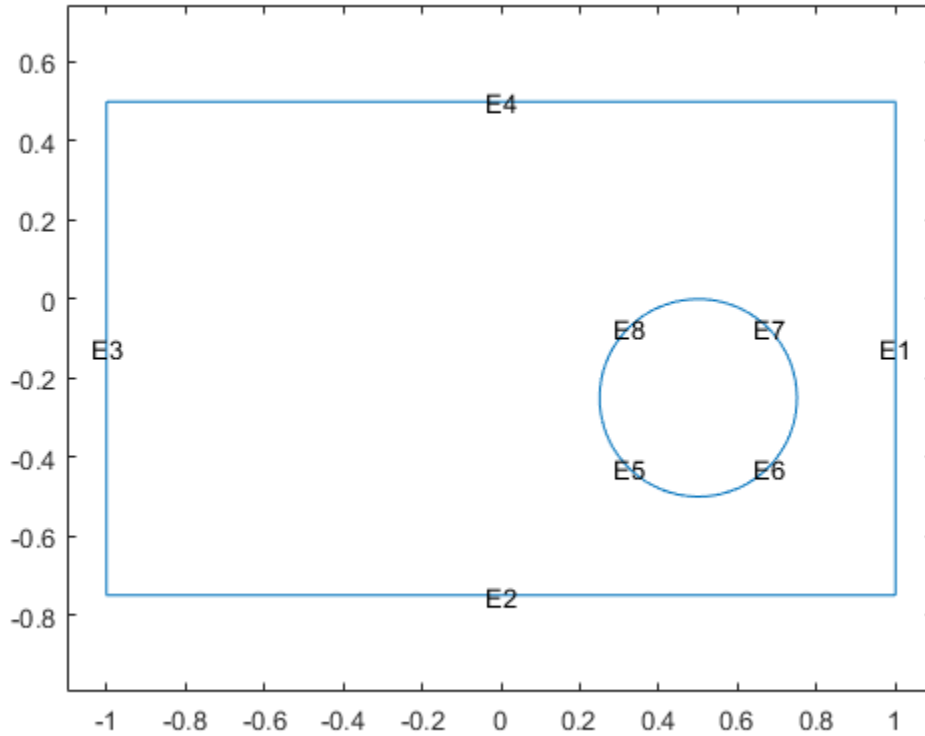
- "Solve PDEs with Constant Boundary Conditions" on page 2-167

## 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 2-7.

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`.

# See Also

## See Also
AnalyticGeometry Properties | `PDEModel`

## Topics
"Solve PDEs with Constant Boundary Conditions" on page 2-167
"Solve Problems Using PDEModel Objects" on page 2-5
"2-D Geometry"

**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.

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
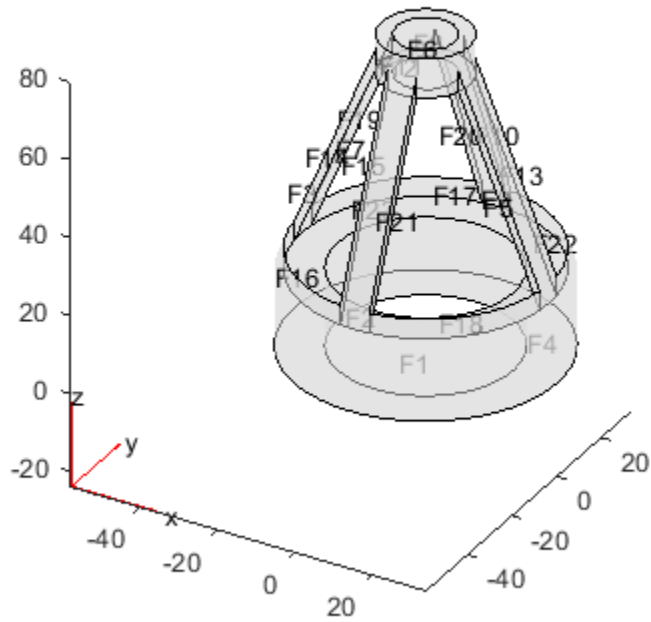
### Geometry from Convex Hull

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.

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
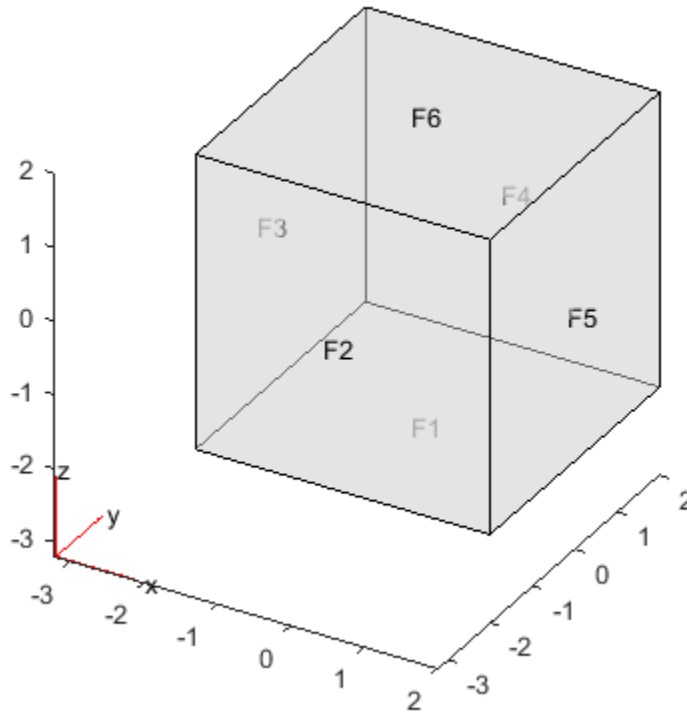
### Geometry from alphaShape

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.

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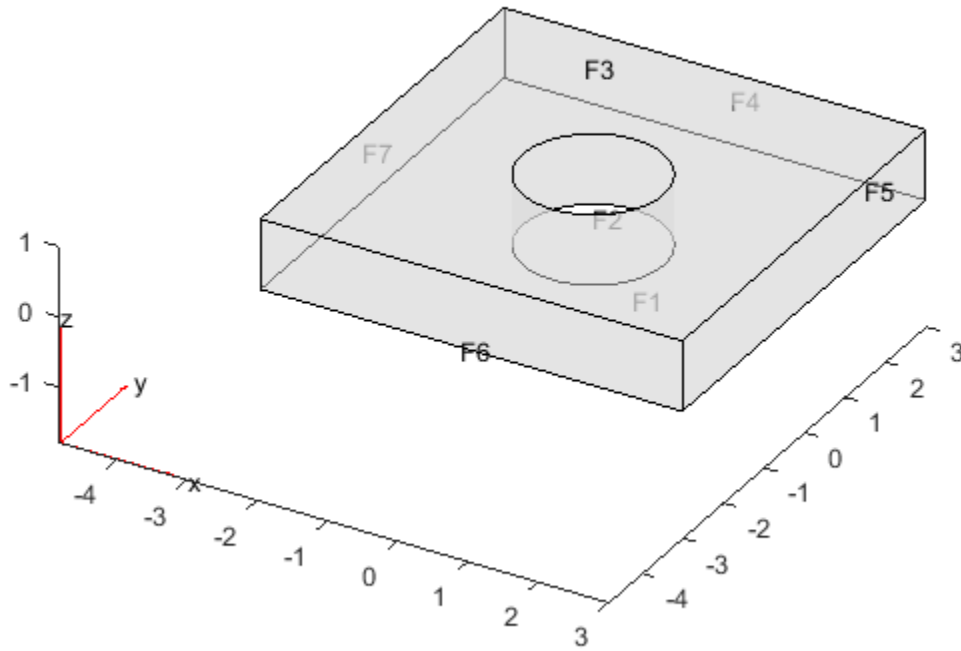
To use the geometry in an analysis, create a volume mesh.

```
generateMesh(model);
```

- "Create and View 3-D Geometry" on page 2-36
- "Solve Problems Using PDEModel Objects" on page 2-5

## 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 2-196.

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

## See Also
alphaShape | DiscreteGeometry Properties | `generateMesh` | `importGeometry`

## Topics
"Create and View 3-D Geometry" on page 2-36
"Solve Problems Using PDEModel Objects" on page 2-5

**Introduced in R2015b**

# HeatSourceAssignment Properties

Heat source assignments

## Description

A `HeatSourceAssignment` object contains a description of the heat sources for a thermal model. A `ThermalModel` container has a vector of `HeatSourceAssignment` objects in its `HeatSources.HeatSourceAssignments` property.

Create heat source assignments for your thermal model using the `internalHeatSource` 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 `'FaceLabels'` name-value pair to `'on'`.

Data Types: `double`

### HeatSource — Heat source value
number | function handle

Heat source value, returned as a number or a function handle. A heat source with a negative value is called a heat sink.

Data Types: `double` | `function_handle`

# See Also

## See Also
findHeatSource | internalHeatSource

**Introduced in R2017a**

# 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

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

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 $\Omega$, or the system PDE problem

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

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

$$\nabla u \cdot \mathbf{n} = 0$$
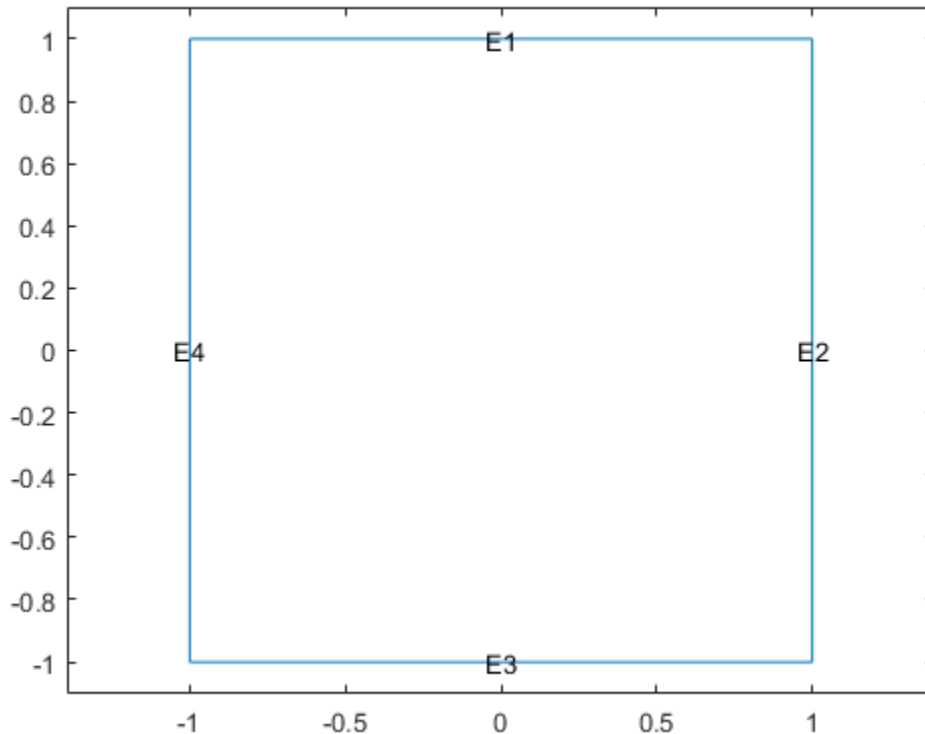
for $y = \pm 1$. (The Neumann boundary condition is the default condition, so the second specification is redundant.)

```
applyBoundaryCondition(model,'dirichlet','Edge',[2,4],'u',0);
applyBoundaryCondition(model,'neumann','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))
```
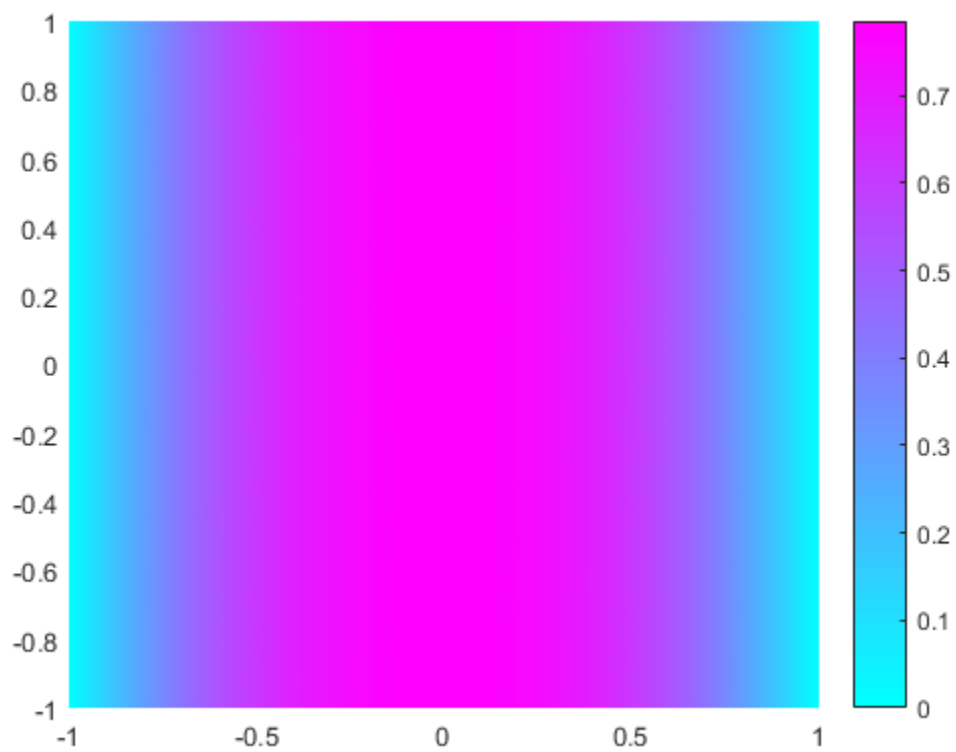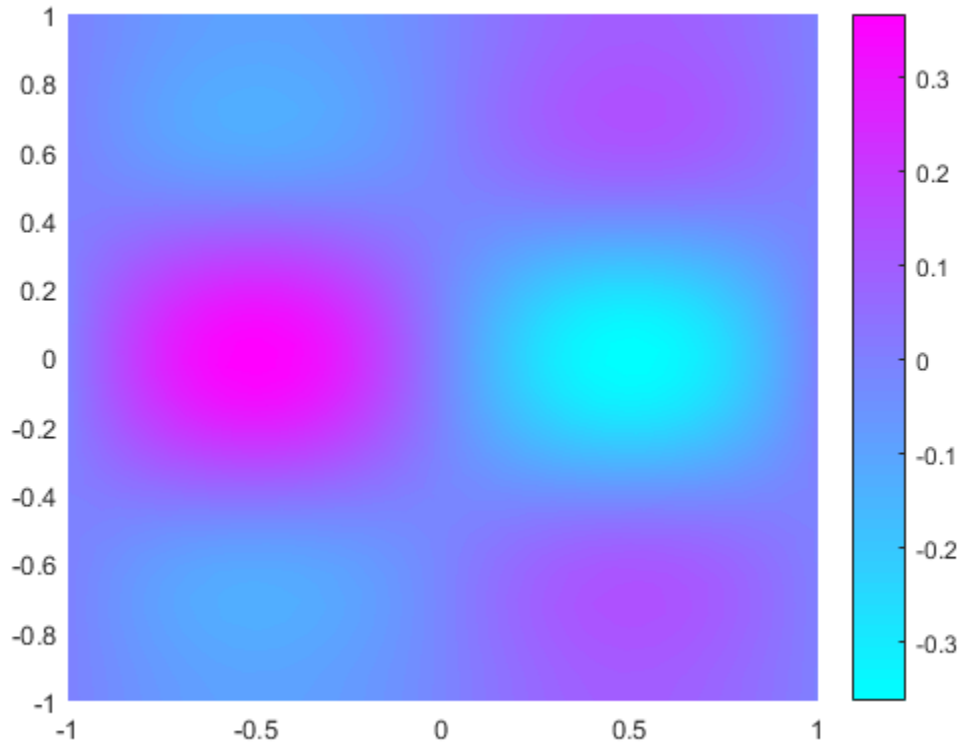
For a version of this example with animation, see "Wave Equation on a Square Domain".

### 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
```
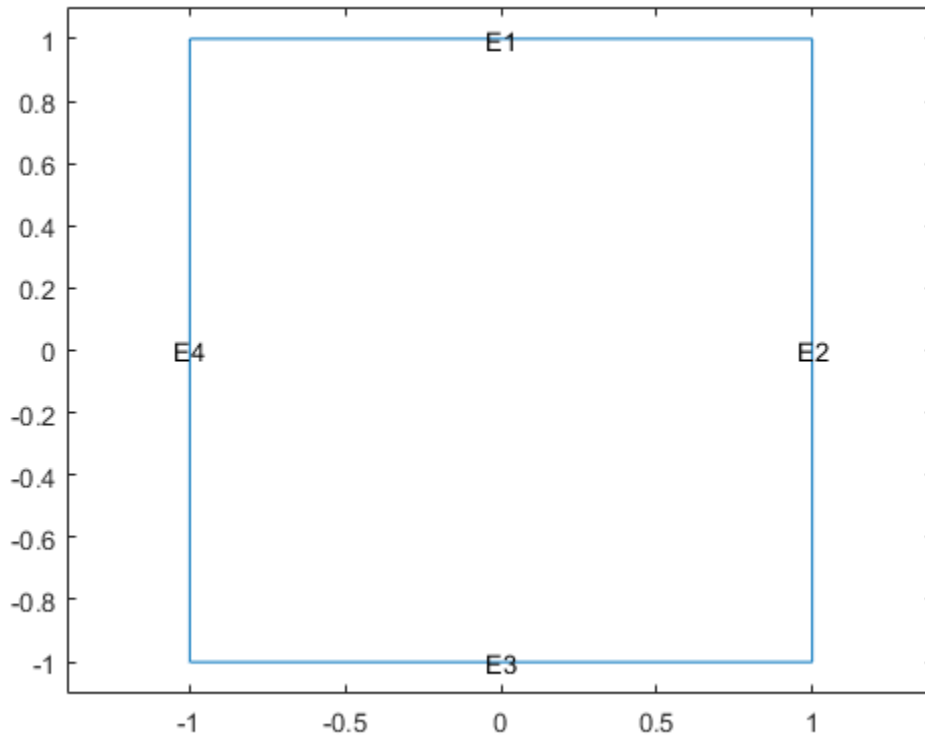


Set Dirichlet boundary conditions $u = 0$ for $x = \pm 1$, and Neumann boundary conditions

$$\nabla u \cdot \mathbf{n} = 0$$

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))
```
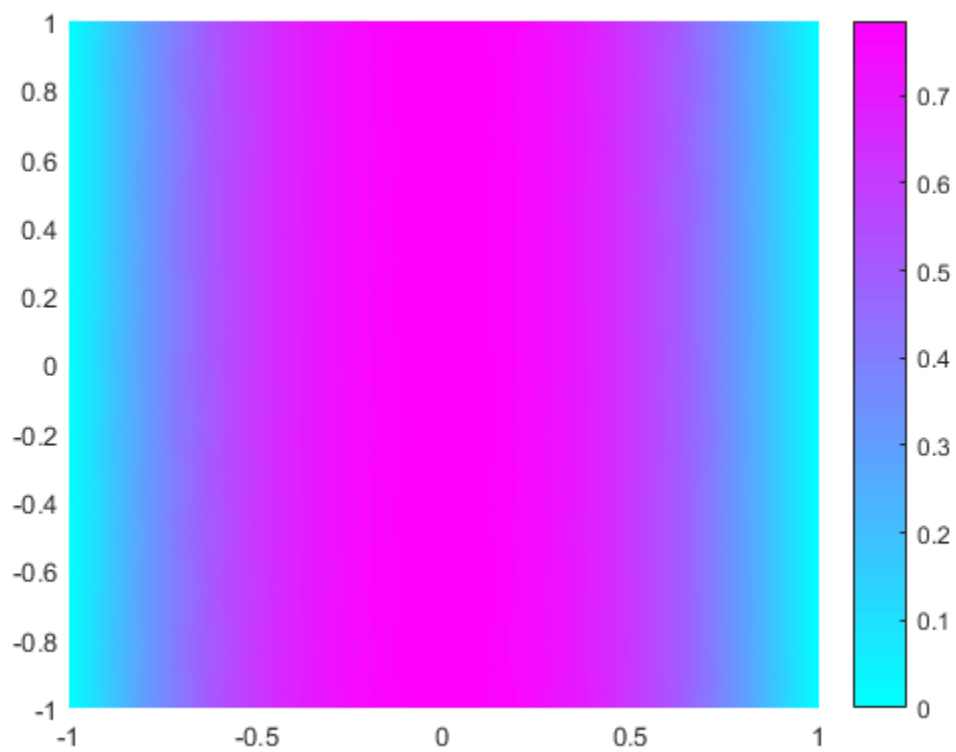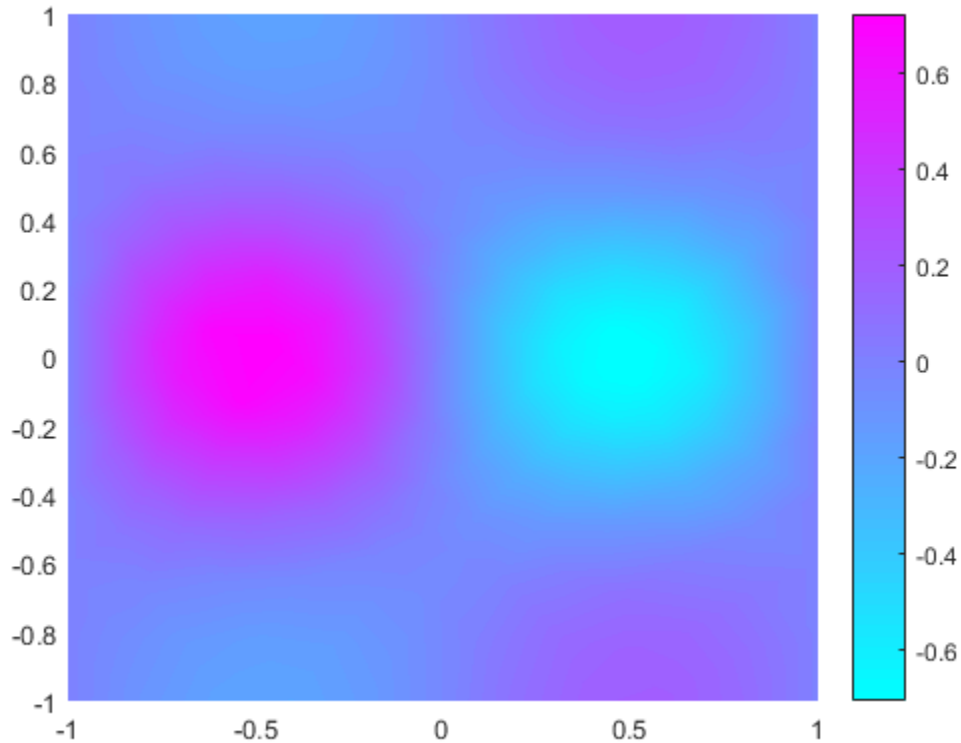
For a version of this example with animation, see "Wave Equation on a Square Domain".

### 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');
figure
pdegplot(model,'FaceLabels','on')
view(30,30)
title('Bracket with Face Labels')
```

```
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```



**Bracket with Face Labels**

**Bracket with Face Labels, Rear View**



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 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',4,'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);
```

```
1483 successful steps
67 failed attempts
2934 function evaluations
1 partial derivatives
277 LU decompositions
2933 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')
```
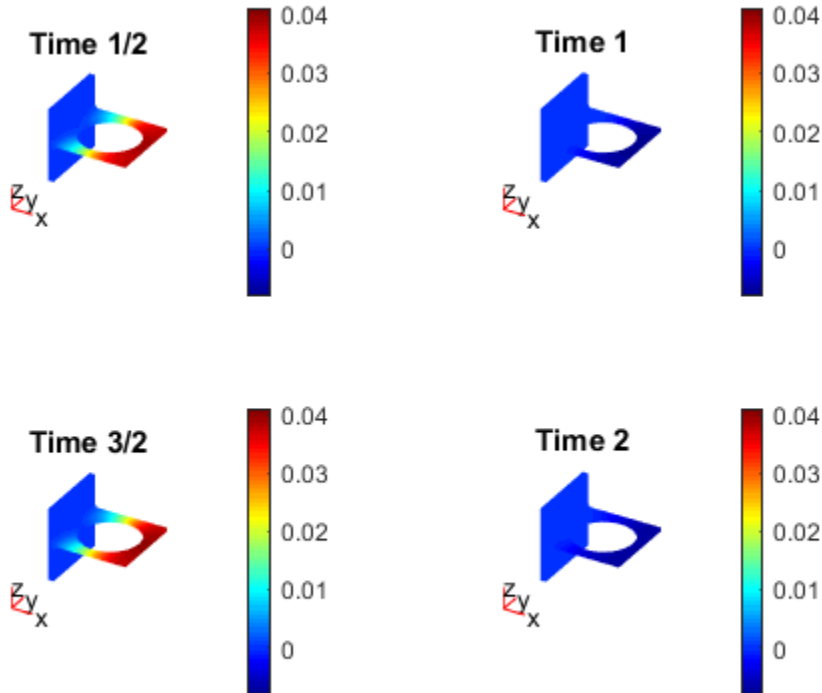


The solution seems to have a frequency of one, because the plots at times 1/2 and 3/2 show maximum values, and those at times 1 and 2 show minimum values.

### 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.

```
model = createpde();
importGeometry(model,'BracketWithHole.stl');
figure
pdegplot(model,'FaceLabels','on')
view(30,30)
title('Bracket with Face Labels')
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```

**Bracket with Face Labels**

**Bracket with Face Labels, Rear View**



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 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',4,'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);

1421 successful steps
72 failed attempts
2806 function evaluations
1 partial derivatives
275 LU decompositions
2805 solutions of linear systems
```

Plot the maximum value at each time. The oscillations damp slightly as time increases.

```
plot(max(u))
xlabel('Time')
ylabel('Maximum value')
title('Maximum of Solution')
```

## 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 2-147.

- 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 2-147.

- 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 2-147.

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 2-110. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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

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

There are a wide variety of ways of specifying a, detailed in "a or d Coefficient for Systems" on page 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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

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

You can specify f in various ways, detailed in "f Coefficient for Systems" on page 2-83. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 file name.

- 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 2-154.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-183.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

### p — Mesh points
matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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

Mesh triangles, specified as a 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 2-196.

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`

### 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 6-62. Typically, `B` is the output of `assempde`.

### `ud` — Dirichlet vector
vector

Dirichlet vector, returned as a vector. See "Algorithms" on page 6-62. 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,O,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 3-210.

## 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.

## See Also

### See Also
solvepde

## Topics

**Introduced before R2006a**

# importGeometry

Import 3-D geometry

## Compatibility

Importing 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')
```
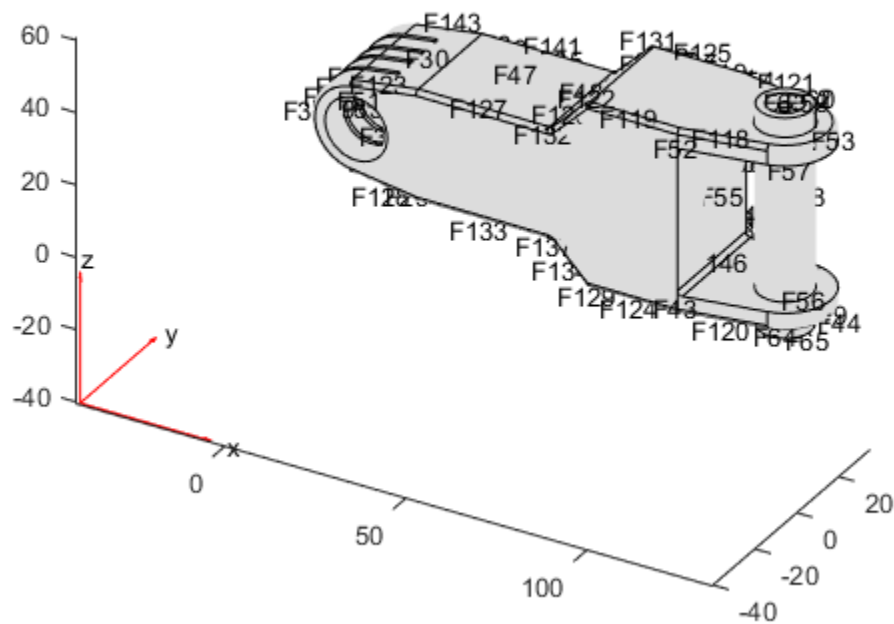


- "Create and View 3-D Geometry" on page 2-36
- "Solve Problems Using PDEModel Objects" on page 2-5

## Input Arguments

### `model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: `model = createpde(1)`

**`geometryfile` — Path to STL file**
vector of characters

Path to STL file, specified as a vector of characters ending 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

## See Also
DiscreteGeometry Properties | `geometryFromMesh` | `pdegplot` | `PDEModel`

## Topics
"Create and View 3-D Geometry" on page 2-36
"Solve Problems Using PDEModel Objects" on page 2-5

**Introduced in R2015a**

# initmesh

Create initial 2-D 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

`[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.
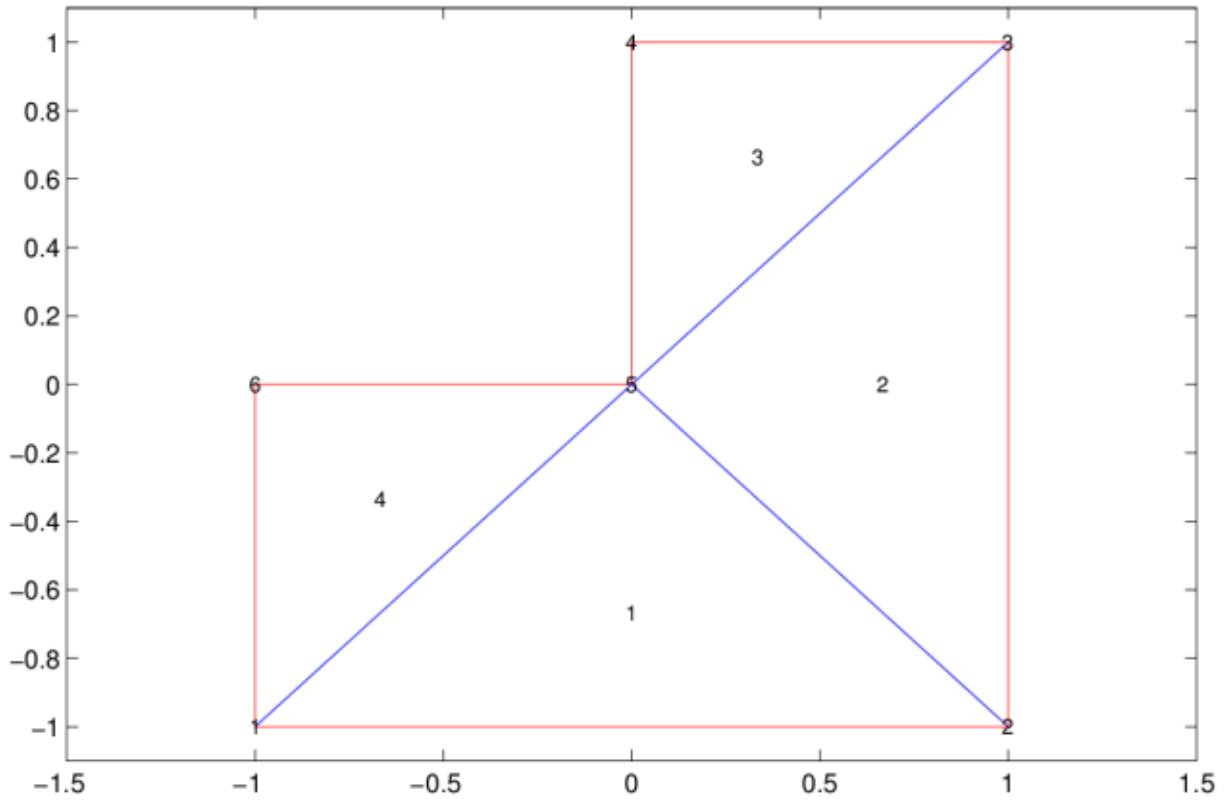
The corresponding mesh data structures can be exported to the main workspace by selecting the **Export Mesh** option from the **Mesh** menu.

```
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
```

## References

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

## See Also

decsg | jigglemesh | refinemesh

## Topics

"Mesh Data" on page 2-196

**Introduced before R2006a**

# internalHeatSource

Specify internal heat source for a thermal model

## Syntax

```
internalHeatSource(thermalmodel,heatSourceValue)
internalHeatSource(thermalmodel,heatSourceValue,RegionType,RegionID)
heatSource = internalHeatSource( ___ )
```

## Description

`internalHeatSource(thermalmodel,heatSourceValue)` specifies an internal heat source for the thermal model. This syntax declares that the entire geometry is a heat source.

---

**Note:** Use `internalHeatSource` for specifying **internal heat generators**, that is, for specifying heat sources that belong to the geometry of the model. To specify a heat influx from an external source, use the `thermalBC` function with the `HeatFlux` parameter.

---

`internalHeatSource(thermalmodel,heatSourceValue,RegionType,RegionID)` specifies geometry regions of type `RegionType` with ID numbers in `RegionID` as heat sources. Always specify `heatSourceValue` first, then specify `RegionType` and `RegionID`.

`heatSource = internalHeatSource( ___ )` returns the heat source object.

## Examples

### Specify Internal Heat Generation on Entire Geometry

Create a transient thermal model.

```
thermalmodel = createpde('thermal','transient');
```

Import the geometry.

```
gm = importGeometry(thermalmodel,'SquareBeam.STL');
```

Set thermal conductivity to 0.2, mass density to 2700e-9, and specific heat to 920.

```
thermalProperties(thermalmodel,'ThermalConductivity',0.2, ...
                               'MassDensity',2700e-9, ...
                               'SpecificHeat',920)
```

```
ans =

  ThermalMaterialAssignment with properties:

               RegionType: 'cell'
                 RegionID: 1
        ThermalConductivity: 0.2000
               MassDensity: 2.7000e-06
               SpecificHeat: 920
```

Specify that the entire geometry generates heat at the rate 2e-4.

```
internalHeatSource(thermalmodel,2e-4)
```

```
ans =

  HeatSourceAssignment with properties:

    RegionType: 'cell'
      RegionID: 1
    HeatSource: 2.0000e-04
```

### Specify a Face of a 2-D Geometry as a Heat Source

Create a steady-state thermal model.

```
thermalModel = createpde('thermal','transient');
```

Create the geometry.

```
SQ1 = [3; 4; 0; 3; 3; 0; 0; 0; 3; 3];
D1 = [2; 4; 0.5; 1.5; 2.5; 1.5; 1.5; 0.5; 1.5; 2.5];
```

```
gd = [SQ1 D1];
sf = 'SQ1+D1';
ns = char('SQ1','D1');
ns = ns';
dl = decsg(gd,sf,ns);

geometryFromEdges(thermalModel,dl);
```

Set thermal conductivity to 50, mass density to 2500, and specific heat to 600.

```
thermalProperties(thermalModel,'ThermalConductivity',50, ...
                               'MassDensity',2500, ...
                               'SpecificHeat',600);
```

Specify that face 1 generates heat at 25.

```
internalHeatSource(thermalModel,25,'Face',1)


ans =

  HeatSourceAssignment with properties:

    RegionType: 'face'
      RegionID: 1
    HeatSource: 25
```

## Input Arguments

**`thermalmodel`** — **Thermal model**
ThermalModel object

Thermal model, specified as a ThermalModel object. The model contains the geometry, mesh, thermal properties of the material, internal heat source, boundary conditions, and initial conditions.

Example: thermalmodel = createpde('thermal')

**`RegionType`** — **Geometric region type**
'Face' | 'Cell'

Geometric region type, specified as 'Face' for a 2-D model or 'Cell' for a 3-D model.

Example: `internalHeatSource(thermalmodel,25,'Cell',1)`

Data Types: `char`

### **RegionID** — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot`, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Example: `internalHeatSource(thermalmodel,25,'Cell',1:3)`

Data Types: `double`

### **heatSourceValue** — Heat source value
number | function handle

Heat source value, specified as a number or a function handle.

Example: `internalHeatSource(thermalmodel,25)`

Data Types: `double` | `function_handle`

## Output Arguments

### **heatSource** — Handle to heat source
object

Handle to heat source, returned as an object. `heatSourceValue` associates the heat source value with the geometric region.

# See Also

## See Also
`thermalBC` | `thermalProperties`

**Introduced in R2017a**

# interpolateSolution

Interpolate PDE solution to arbitrary points

## Compatibility

Interpolating PDE solution to arbitrary points is the same for 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)
```

## Description

`uintrp = interpolateSolution(results,xq,yq)` returns the interpolated values of the solution to the scalar stationary 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 stationary equations for equation indices `iU`.

`uintrp = interpolateSolution( ___ ,iT)` returns the interpolated values of the solution to the time-dependent or eigenvalue equation or system of such equations at

times or modal indices `iT`. For a system of time-dependent or eigenvalue equations, specify both time/modal indices `iT` and equation indices `iU`

## Examples

### Interpolate Scalar Stationary Results

Interpolate the solution to a scalar 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,'dirichlet','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)')
```
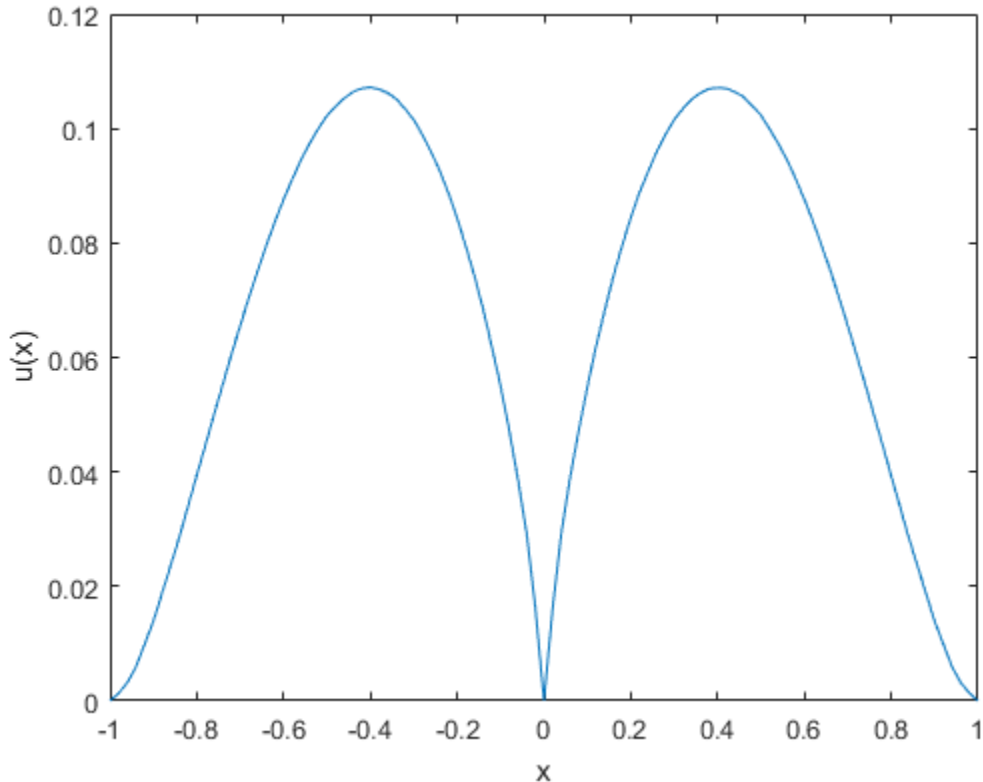
### Interpolate Solution of Poisson's Equation

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,'dirichlet','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
```
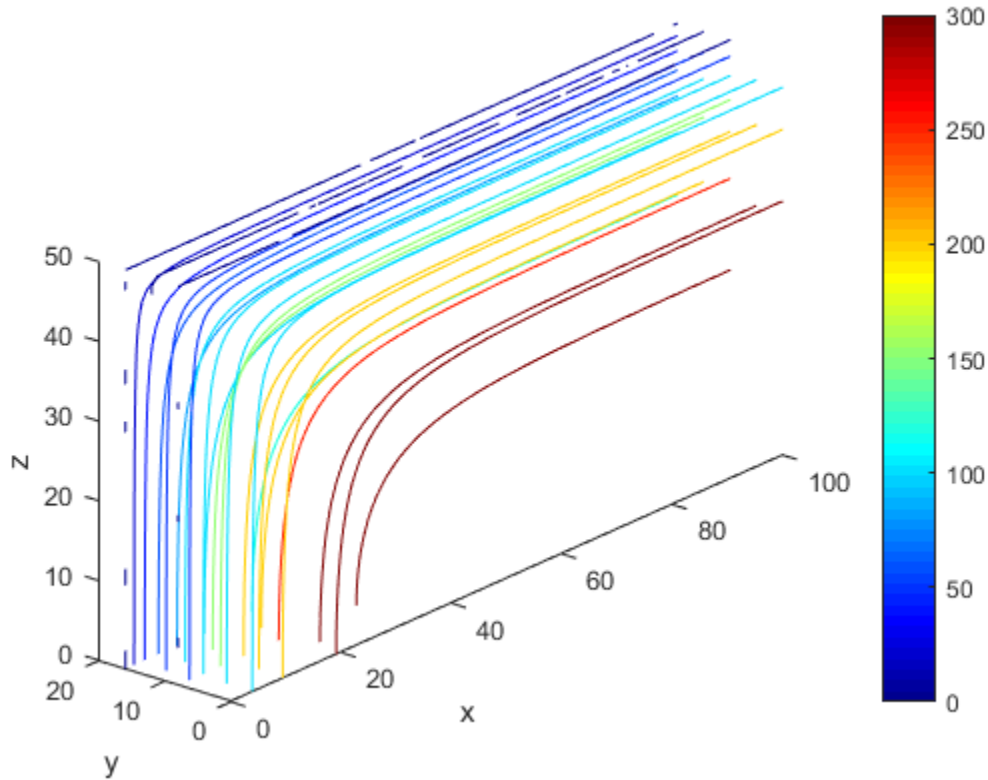
### Interpolate Scalar Stationary Results Using Query Matrix

Solve a scalar stationary 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,'dirichlet','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')
```
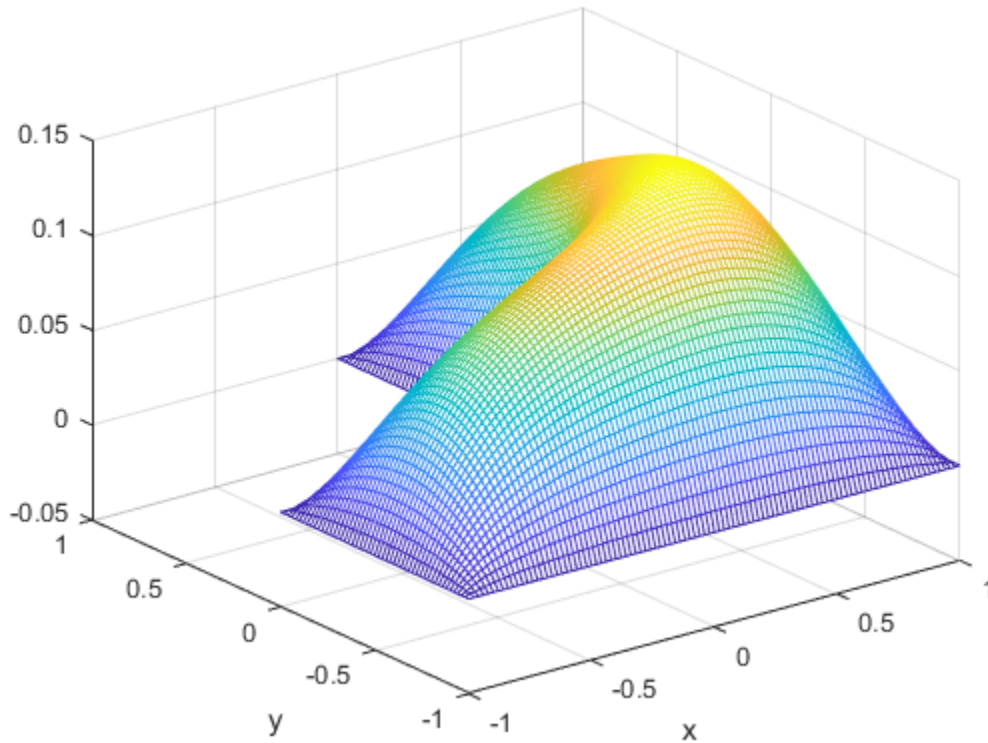
### Interpolate Stationary System

Create the solution to a two-component 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');
```
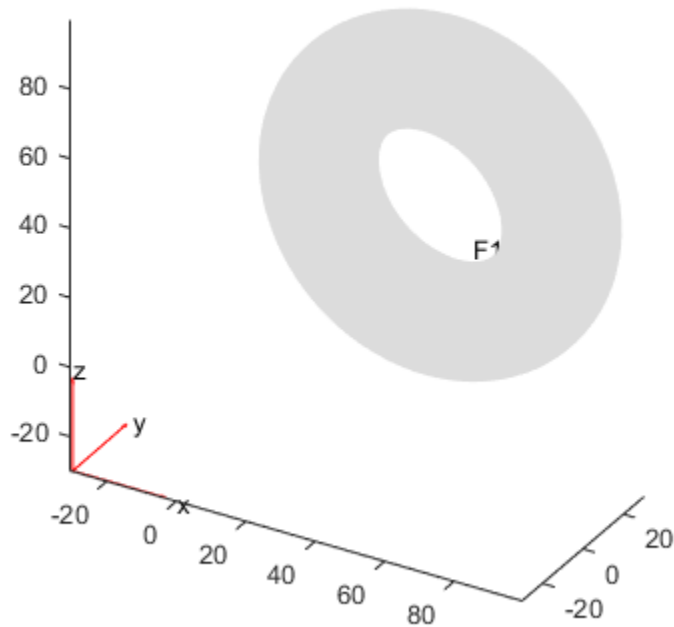
Set boundary conditions.

```
gfun = @(region,state)[0,region.z-40];
applyBoundaryCondition(model,'neumann','Face',1,'g',gfun);
ufun = @(region,state)[region.x-40,0];
applyBoundaryCondition(model,'dirichlet','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**
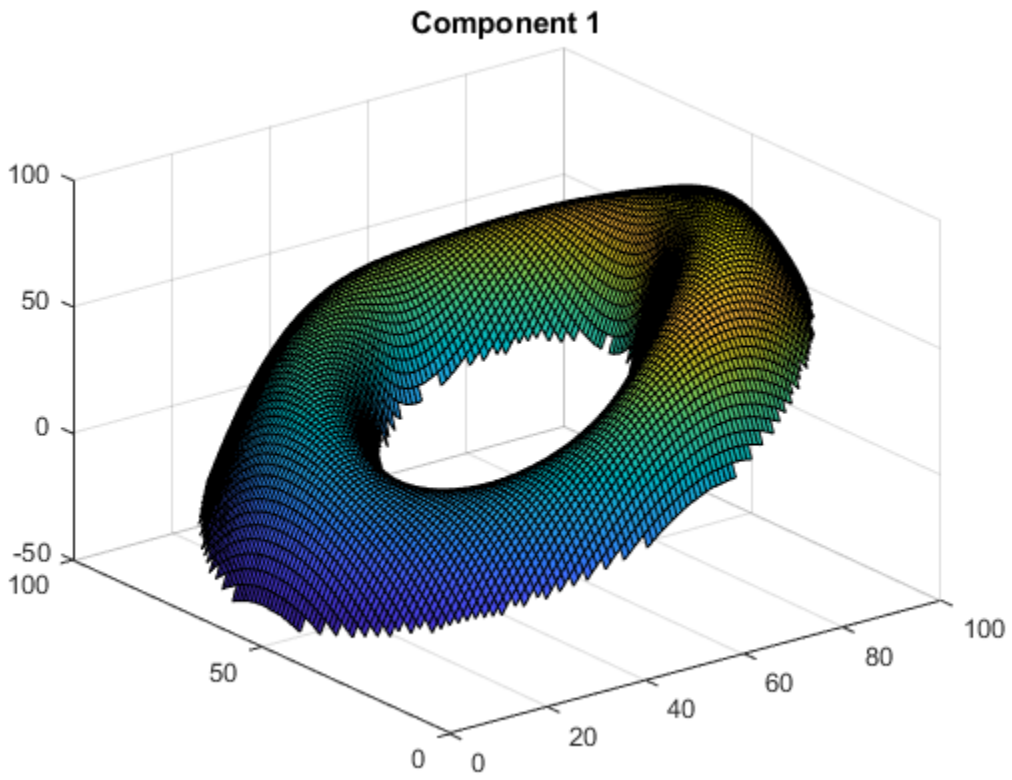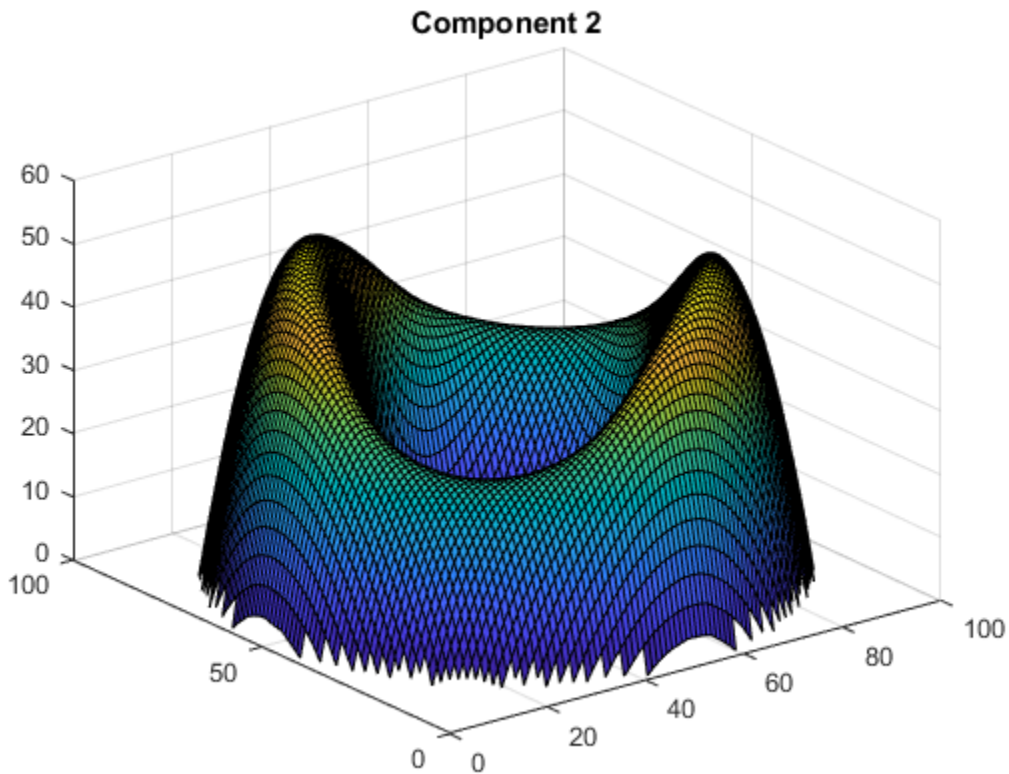
### Interpolate Scalar Eigenvalue Results

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,'dirichlet','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);
```

```
Basis= 10,  Time=  1.15,  New conv eig=  0
Basis= 11,  Time=  1.31,  New conv eig=  0
Basis= 12,  Time=  1.31,  New conv eig=  0
Basis= 13,  Time=  1.33,  New conv eig=  0
Basis= 14,  Time=  1.33,  New conv eig=  0
Basis= 15,  Time=  1.58,  New conv eig=  0
Basis= 16,  Time=  1.58,  New conv eig=  0
Basis= 17,  Time=  1.61,  New conv eig=  0
Basis= 18,  Time=  1.61,  New conv eig=  0
Basis= 19,  Time=  1.83,  New conv eig=  1
Basis= 20,  Time=  1.92,  New conv eig=  1
Basis= 21,  Time=  2.06,  New conv eig=  3
Basis= 22,  Time=  2.12,  New conv eig=  3
Basis= 23,  Time=  2.39,  New conv eig=  3
Basis= 24,  Time=  2.39,  New conv eig=  4
Basis= 25,  Time=  2.45,  New conv eig=  5
Basis= 26,  Time=  2.64,  New conv eig=  6
Basis= 27,  Time=  2.64,  New conv eig=  6
Basis= 28,  Time=  2.68,  New conv eig=  6
Basis= 29,  Time=  2.68,  New conv eig=  6
Basis= 30,  Time=  2.68,  New conv eig=  8
Basis= 31,  Time=  2.70,  New conv eig=  8
Basis= 32,  Time=  2.73,  New conv eig=  9
Basis= 33,  Time=  2.84,  New conv eig= 11
Basis= 34,  Time=  2.84,  New conv eig= 11
Basis= 35,  Time=  2.85,  New conv eig= 14
Basis= 36,  Time=  2.98,  New conv eig= 14
Basis= 37,  Time=  2.98,  New conv eig= 14
Basis= 38,  Time=  3.18,  New conv eig= 14
Basis= 39,  Time=  3.18,  New conv eig= 14
Basis= 40,  Time=  3.20,  New conv eig= 14
Basis= 41,  Time=  3.34,  New conv eig= 14
Basis= 42,  Time=  3.63,  New conv eig= 14
Basis= 43,  Time=  3.76,  New conv eig= 15
Basis= 44,  Time=  3.85,  New conv eig= 16
Basis= 45,  Time=  3.93,  New conv eig= 16
Basis= 46,  Time=  3.93,  New conv eig= 16
Basis= 47,  Time=  3.93,  New conv eig= 16
Basis= 48,  Time=  3.96,  New conv eig= 17
Basis= 49,  Time=  4.07,  New conv eig= 17
```

```
              Basis= 50,  Time=   4.07,  New conv eig= 18
              Basis= 51,  Time=   4.09,  New conv eig= 18
              Basis= 52,  Time=   4.09,  New conv eig= 18
              Basis= 53,  Time=   4.24,  New conv eig= 18
              Basis= 54,  Time=   4.24,  New conv eig= 20
End of sweep: Basis= 54,  Time=   4.24,  New conv eig= 20
              Basis= 30,  Time=   4.85,  New conv eig=  0
              Basis= 31,  Time=   5.02,  New conv eig=  0
              Basis= 32,  Time=   5.05,  New conv eig=  0
              Basis= 33,  Time=   5.05,  New conv eig=  0
              Basis= 34,  Time=   5.21,  New conv eig=  0
End of sweep: Basis= 34,  Time=   5.21,  New conv eig=  0
```

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)
```
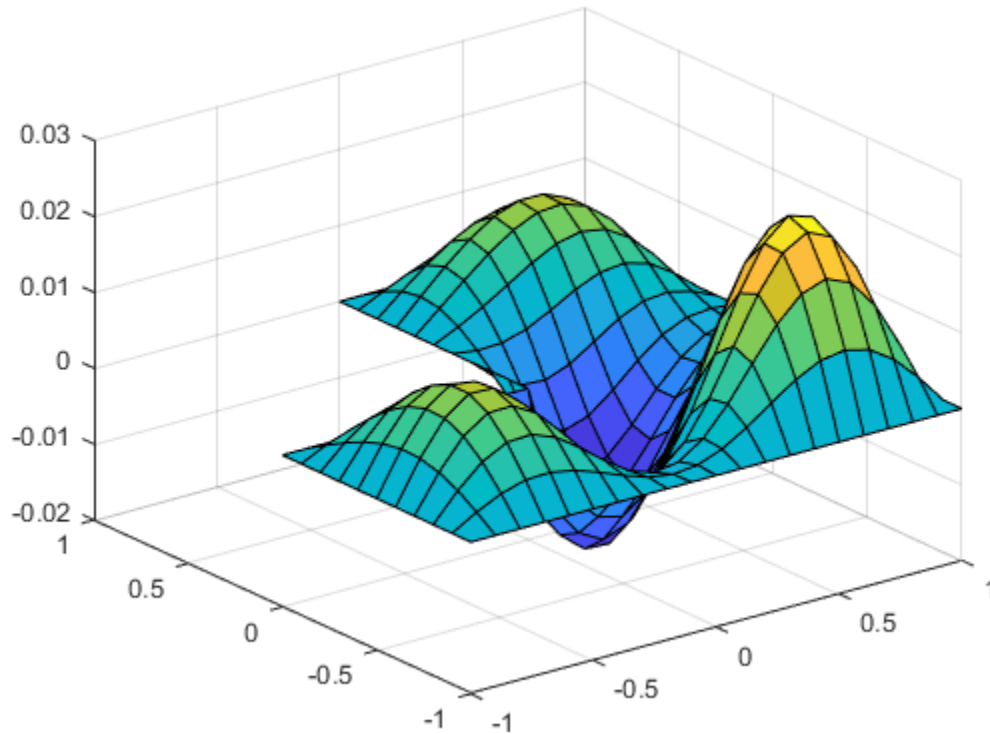
**Interpolate Time-Dependent System**

Solve a system of time-dependent 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');
pdegplot(model,'FaceLabels','on','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,'dirichlet','Face',2,'u',[0,0,0]);
applyBoundaryCondition(model,'neumann','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.

```
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([0,14e-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.

`interpolateSolution` 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 time-dependent 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, Gradients, and Fluxes" on page 3-232.

# See Also

## See Also
evaluateGradient | PDEModel | StationaryResults | TimeDependentResults

## Topics
"Plot 2-D Solutions and Their Gradients" on page 3-199
"Plot 3-D Solutions and Their Gradients" on page 3-210
"Dimensions of Solutions, Gradients, and Fluxes" on page 3-232

**Introduced in R2015b**

# interpolateTemperature

Interpolate temperature in a thermal result at arbitrary spatial locations

## Syntax

```
Tintrp = interpolateTemperature(thermalresults,xq,yq)
Tintrp = interpolateTemperature(thermalresults,xq,yq,zq)
Tintrp = interpolateTemperature(thermalresults,querypoints)
Tintrp = interpolateTemperature( ___ ,iT)
```

## Description

`Tintrp = interpolateTemperature(thermalresults,xq,yq)` returns the interpolated temperature values at the 2-D points specified in `xq` and `yq`. This syntax is valid for both the steady-state and transient thermal models.

`Tintrp = interpolateTemperature(thermalresults,xq,yq,zq)` returns the interpolated temperature values at the 3-D points specified in `xq`, `yq`, and `zq`. This syntax is valid for both the steady-state and transient thermal models.

`Tintrp = interpolateTemperature(thermalresults,querypoints)` returns the interpolated temperature values at the points in `querypoints`. This syntax is valid for both the steady-state and transient thermal models.

`Tintrp = interpolateTemperature( ___ ,iT)` returns the interpolated temperature values for the transient thermal model at times `iT`.

## Examples

### Interpolate Temperatures in 2-D Steady-State Thermal Model

Create a thermal model for steady-state analysis.

```
thermalmodel = createpde('thermal');
```

Create the geometry and include it in the model.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1, 'R1', ('R1')');
geometryFromEdges(thermalmodel,g);
pdegplot(thermalmodel,'EdgeLabels','on')
xlim([-1.5,1.5])
axis equal
```
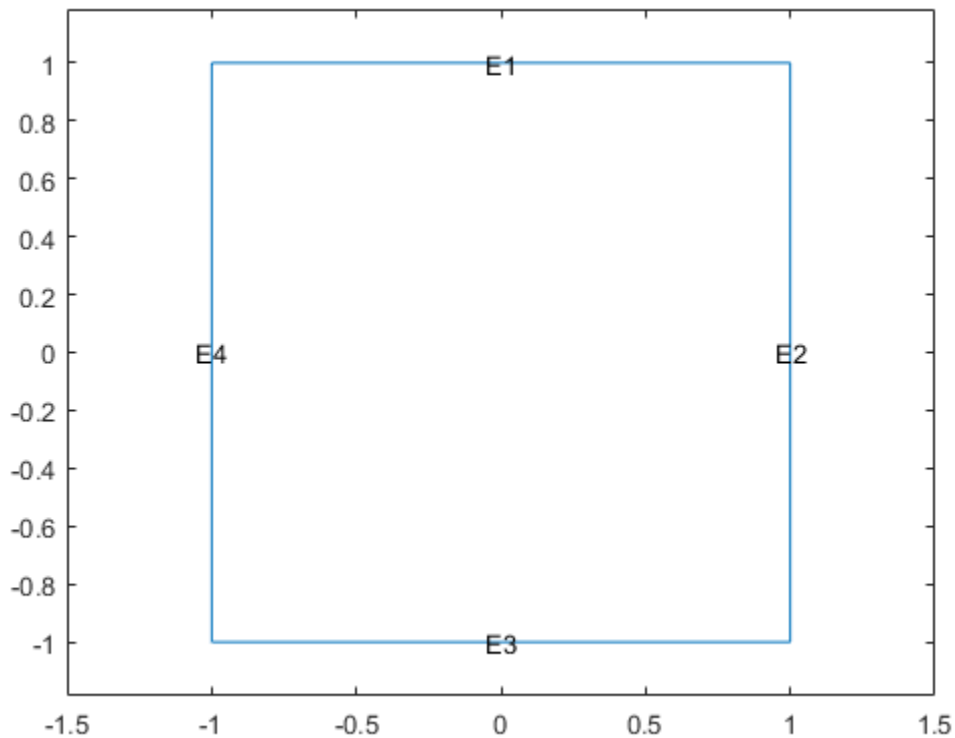


Assuming that this is an iron plate, assign a thermal conductivity of 79.5 W/(m*K). Because this is a steady-state model, you do not need to assign mass density or specific heat values.

```
thermalProperties(thermalmodel,'ThermalConductivity',79.5,'Face',1);
```

Apply a constant temperature of 300 K to the bottom of the plate (edge 3). Also, assume that the top of the plate (edge 1) is insulated, and apply convection on the two sides of the plate (edges 2 and 4).

```
thermalBC(thermalmodel,'Edge',3,'Temperature',300);
thermalBC(thermalmodel,'Edge',1,'HeatFlux',0);
thermalBC(thermalmodel,'Edge',[2,4],...
                       'ConvectionCoefficient',25,...
                       'AmbientTemperature',50);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
results = solve(thermalmodel)


results =

  SteadyStateThermalResults with properties:

    Temperature: [177×1 double]
      XGradients: [177×1 double]
      YGradients: [177×1 double]
      ZGradients: []
            Mesh: [1×1 FEMesh]
```

The solver finds the values of temperatures and temperature gradients at the nodal locations. To access these values, use `results.Temperature`, `results.XGradients`, and so on. For example, plot the temperatures at nodal locations.

```
figure;
pdeplot(thermalmodel,'XYData',results.Temperature,...
                     'Contour','on','ColorMap','hot');
```
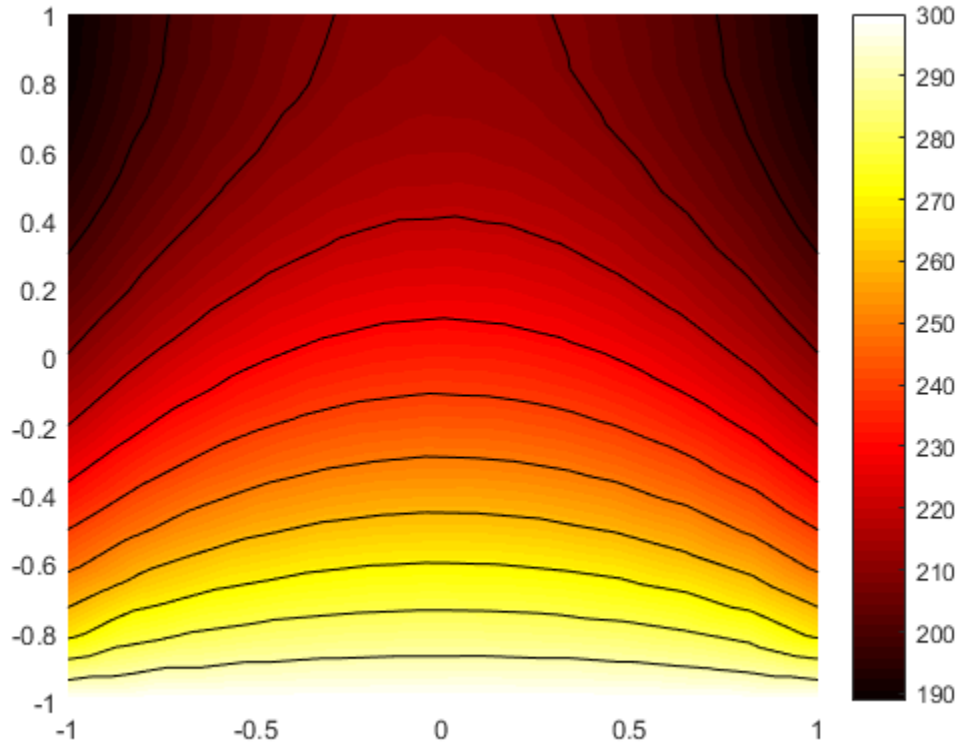
Interpolate the resulting temperatures to a grid covering the central portion of the geometry, for x and y from -0.5 to 0.5.

```
v = linspace(-0.5,0.5,11);
[X,Y] = meshgrid(v);

Tintrp = interpolateTemperature(results,X,Y);
```

Reshape the Tintrp vector and plot the resulting temperatures.

```
Tintrp = reshape(Tintrp,size(X));

figure
contourf(X,Y,Tintrp)
```

```
colormap(hot)
colorbar
```
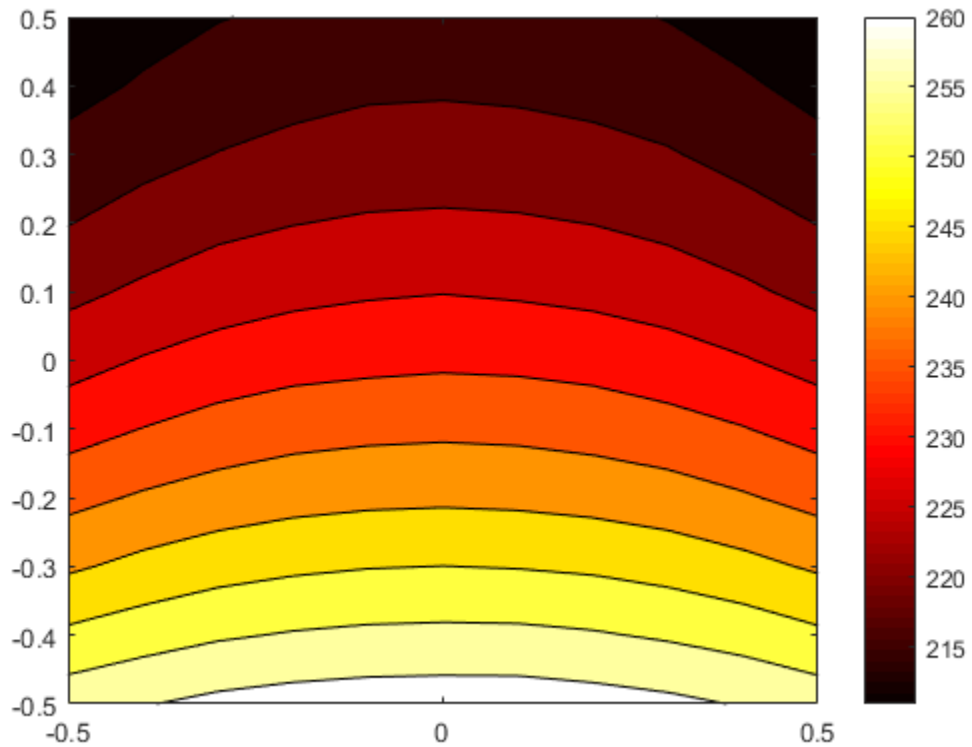


Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Tintrp = interpolateTemperature(results,querypoints);
```

### Interpolate Temperature for a 3-D Steady-State Thermal Model

Create a thermal model for steady-state analysis.

```
thermalmodel = createpde('thermal');
```

Create the following 3-D geometry and include it in the model.

```
importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabels','on','FaceAlpha',0.5)
title('Copper block, cm')
axis equal
```



Copper block, cm

Assuming that this is a copper block, the thermal conductivity of the block is approximately 4 W/(cm*K).

```
thermalProperties(thermalmodel,'ThermalConductivity',4);
```
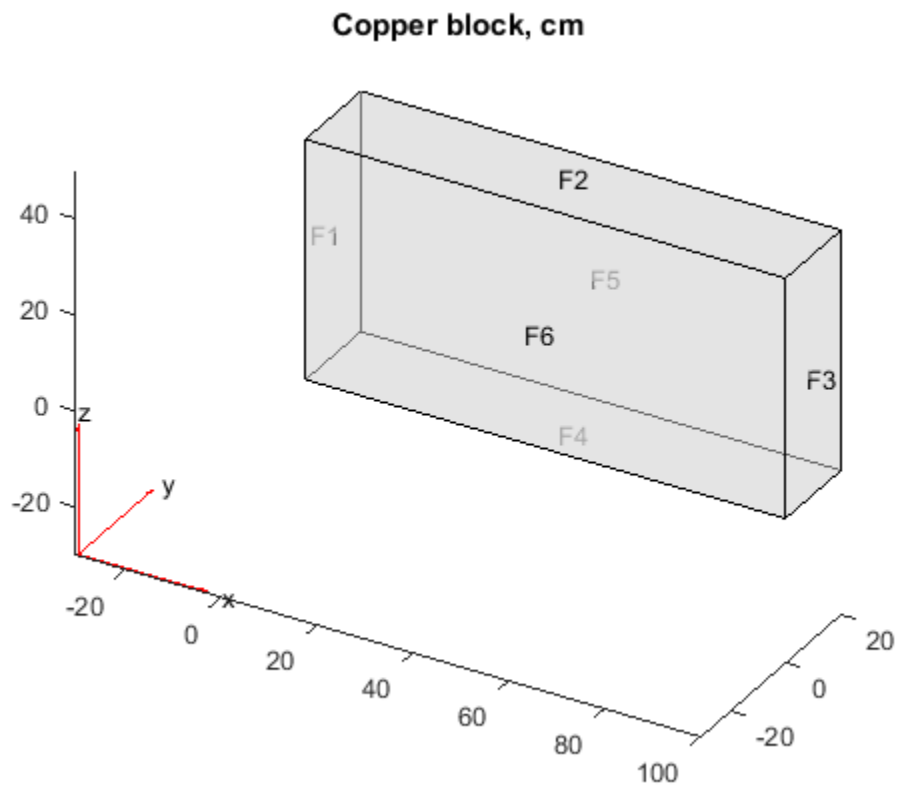
Apply a constant temperature of 373 K to the left side of the block (edge 1) and a constant temperature of 573 K at the right side of the block.

```
thermalBC(thermalmodel,'Face',1,'Temperature',373);
thermalBC(thermalmodel,'Face',3,'Temperature',573);
```

Apply a heat flux boundary condition to the bottom of the block.

```
thermalBC(thermalmodel,'Face',4,'HeatFlux',-20);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)


thermalresults =

  SteadyStateThermalResults with properties:

    Temperature: [28811×1 double]
     XGradients: [28811×1 double]
     YGradients: [28811×1 double]
     ZGradients: [28811×1 double]
           Mesh: [1×1 FEMesh]
```

The solver finds the values of temperatures and temperature gradients at the nodal locations. To access these values, use `results.Temperature`, `results.XGradients`, and so on. For example, plot temperatures at nodal locations.

```
figure;
pdeplot3D(thermalmodel,'ColorMapData',thermalresults.Temperature)
```

Create a grid specified by x, y, and z coordinates and interpolate temperatures to the grid.

```
[X,Y,Z] = meshgrid(1:16:100,1:6:20,1:7:50);
```

```
Tintrp = interpolateTemperature(thermalresults,X,Y,Z);
```

Create a contour slice plot for fixed values of the y coordinate.

```
figure
```

```
Tintrp = reshape(Tintrp,size(X));
```

```
contourslice(X,Y,Z,Tintrp,[],1:6:20,[])
```

```
xlabel('x')
ylabel('y')
zlabel('z')
xlim([1,100])
ylim([1,20])
zlim([1,50])
axis equal
view(-50,22)
colorbar
```
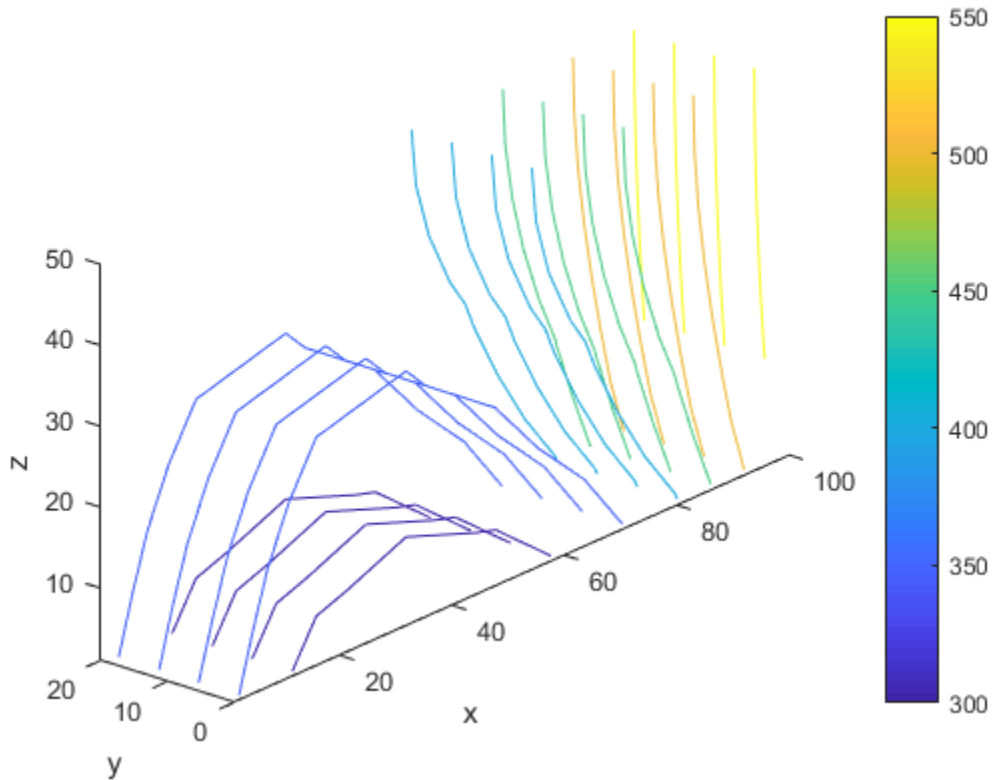


Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:),Z(:)]';
Tintrp = interpolateTemperature(thermalresults,querypoints);
```

Create a contour slice plot for four fixed values of the z coordinate.

```
figure

Tintrp = reshape(Tintrp,size(X));

contourslice(X,Y,Z,Tintrp,[],[],1:7:50)
xlabel('x')
ylabel('y')
zlabel('z')
xlim([1,100])
ylim([1,20])
zlim([1,50])
axis equal
view(-50,22)
colorbar
```
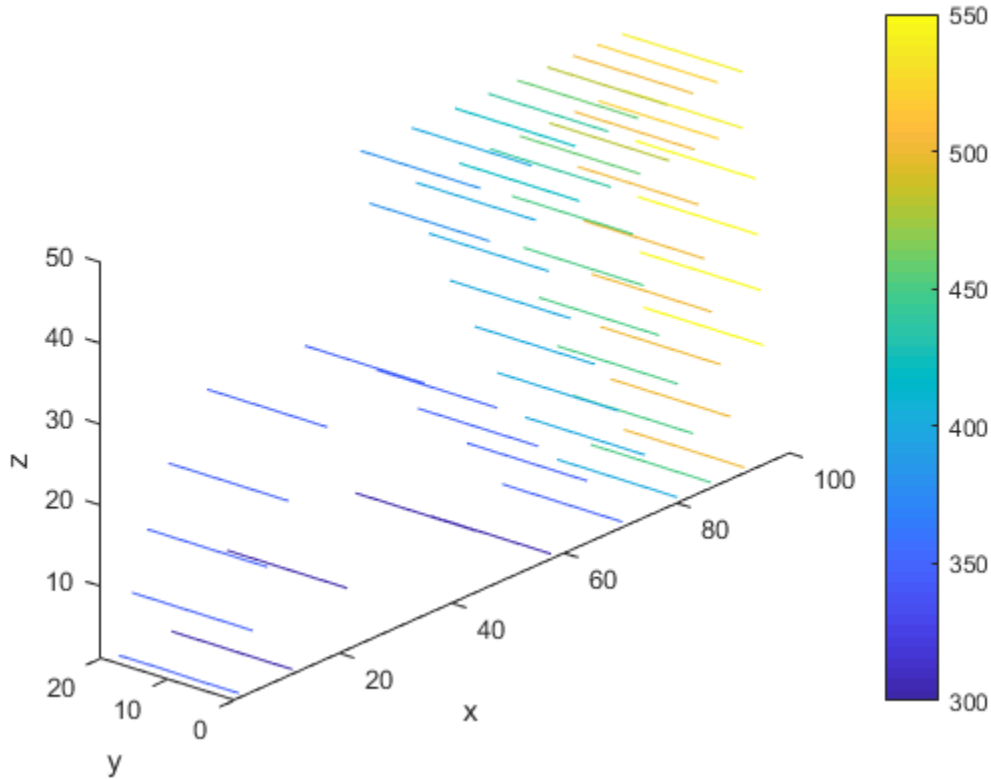
**Temperatures for a Transient Thermal Model on a Square**

Solve a 2-D transient heat transfer problem on a square domain and compute temperatures at the convective boundary.

Create a transient thermal model for this problem.

```
thermalmodel = createpde('thermal','transient');
```

Create the geometry and include it in the model.

```
g = @squareg;
geometryFromEdges(thermalmodel,g);
pdegplot(thermalmodel,'EdgeLabels','on')
```

```
xlim([-1.2,1.2])
ylim([-1.2,1.2])
axis equal
```
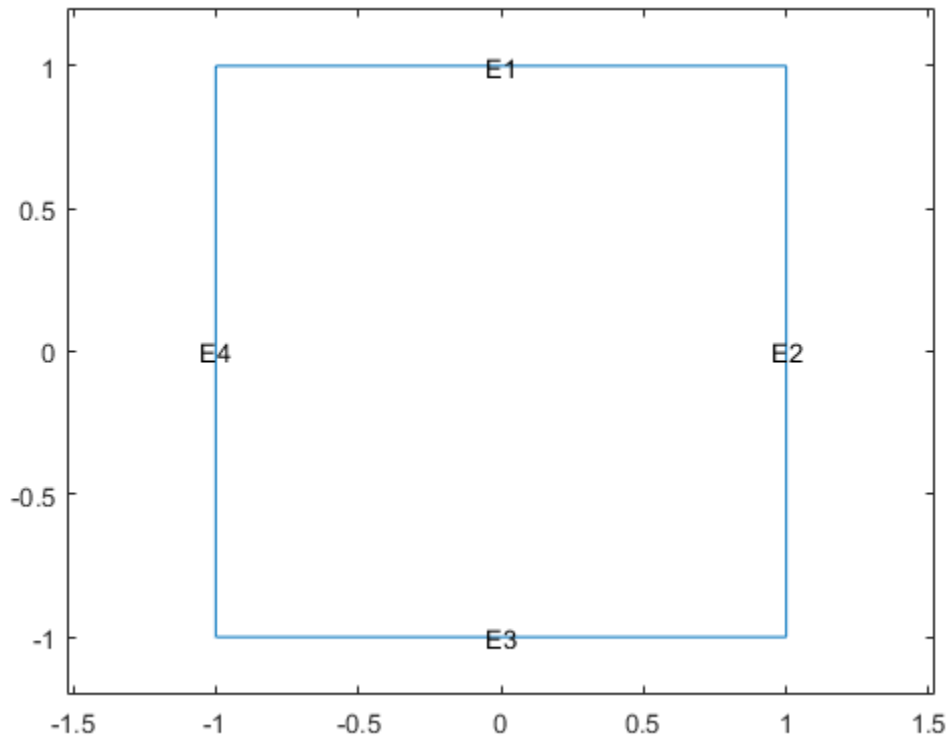


Assign the following thermal properties:

- Thermal conductivity is 100 W/(m*C)
- Mass density is 7800 kg/m^3
- Specific heat is 500 J/(kg*C)

```
thermalProperties(thermalmodel,'ThermalConductivity',100,...
                              'MassDensity',7800,...
```

```
                             'SpecificHeat',500);
```

Apply insulated boundary conditions on three edges and the free convection boundary condition on the right edge.

```
thermalBC(thermalmodel,'Edge',[1,3,4],'HeatFlux',0);
thermalBC(thermalmodel,'Edge',2,...
                       'ConvectionCoefficient',5000,...
                       'AmbientTemperature',25);
```

Set the initial conditions: uniform room temperature across domain and higher temperature on the left edge.

```
thermalIC(thermalmodel,25);
thermalIC(thermalmodel,100,'Edge',4);
```

Generate a mesh and solve the problem using `0:1000:200000` as a vector of times.

```
generateMesh(thermalmodel);
tlist = 0:1000:200000;
thermalresults = solve(thermalmodel,tlist)


thermalresults =

  TransientThermalResults with properties:

      Temperature: [177×201 double]
    SolutionTimes: [1×201 double]
        XGradients: [177×201 double]
        YGradients: [177×201 double]
        ZGradients: []
             Mesh: [1×1 FEMesh]
```

Define a line at convection boundary and compute temperature gradients across that line.

```
X = -1:0.1:1;
Y = ones(size(X));

Tintrp = interpolateTemperature(thermalresults,X,Y,1:length(tlist));
```

Plot the interpolated temperature `Tintrp` along the `x` axis for the following values from the time interval `tlist`.

```
figure
t = [51:50:201];
for i = t
  p(i) = plot(X,Tintrp(:,i),'DisplayName', strcat('t=', num2str(tlist(i))));
  hold on
end
legend(p(t))
xlabel('x')
ylabel('Tintrp')
```
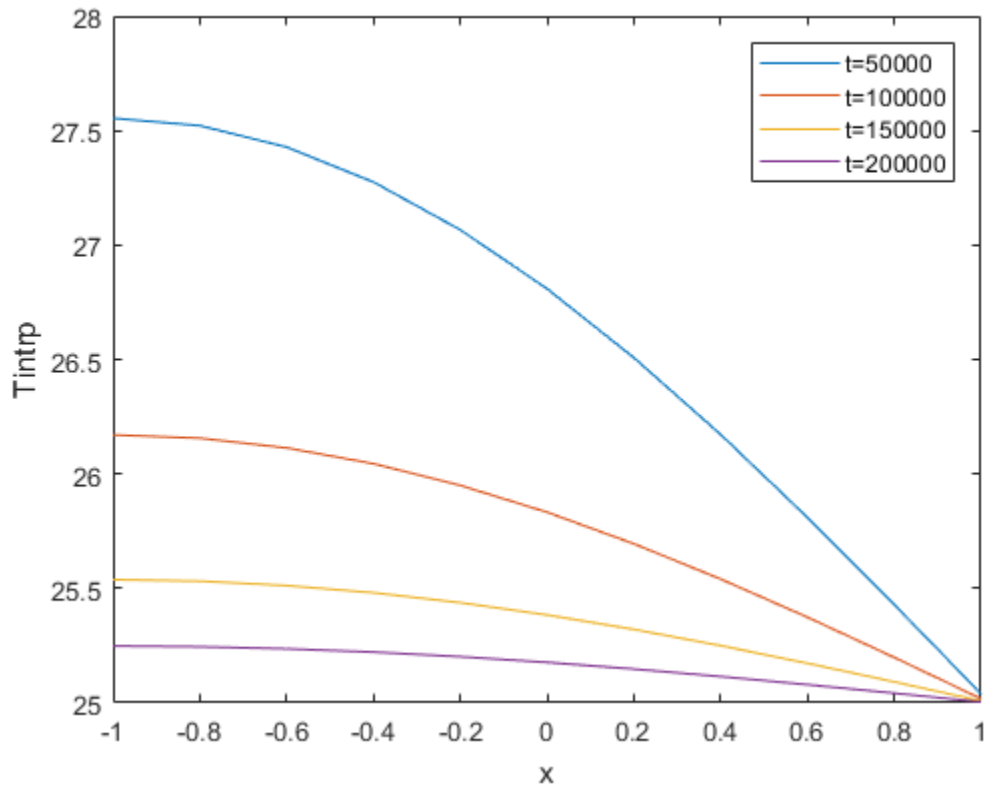


- "Dimensions of Solutions, Gradients, and Fluxes" on page 3-232

# Input Arguments

### `thermalresults` — Solution of thermal problem
SteadyStateThermalResults object | TransientThermalResults object

Solution of thermal problem, specified as a `SteadyStateThermalResults` object or a `TransientThermalResults` object. Create `thermalresults` using `solve`.

Example: `thermalresults = solve(thermalmodel)`

### xq — *x*-coordinate query points
real array

*x*-coordinate query points, specified as a real array. `interpolateTemperature` evaluates temperatures 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.

`interpolateTemperature` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns temperatures in the form of 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 `Txintrp = reshape(gradxTintrp,size(xq))`.

Data Types: `double`

### yq — *y*-coordinate query points
real array

*y*-coordinate query points, specified as a real array. `interpolateTemperature` evaluates temperatures 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, `interpolateTemperature` 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. `interpolateTemperature` evaluates temperatures 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, `interpolateTemperature` 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. `interpolateTemperature` evaluates temperatures 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`

### iT — Time indices
vector of positive integers

Time indices, specified as a vector of positive integers. Each entry in `iT` specifies a time index.

Example: `iT = 1:5:21` specifies every fifth time-step up to 21.

Data Types: `double`

## Output Arguments

### Tintrp — Temperatures at query points
array

Temperatures at query points, returned as an array. For query points that are outside the geometry, `Tintrp` = `NaN`.

## See Also

### See Also
`evaluateHeatFlux` | `evaluateHeatRate` | `evaluateTemperatureGradient` | `SteadyStateThermalResults` | `ThermalModel` | `TransientThermalResults`

## Topics

**Introduced in R2017a**

# jigglemesh

Jiggle internal points of 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 = 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

### Mesh Jiggling

Create a triangular mesh of the L-shaped membrane, first without jiggling, and then jiggle the mesh.

```
[p,e,t] = initmesh('lshapeg','jiggle','off');
q = pdetriq(p,t);
pdeplot(p,e,t,'XYData',q,'ColorBar','on','XYStyle','flat')
```
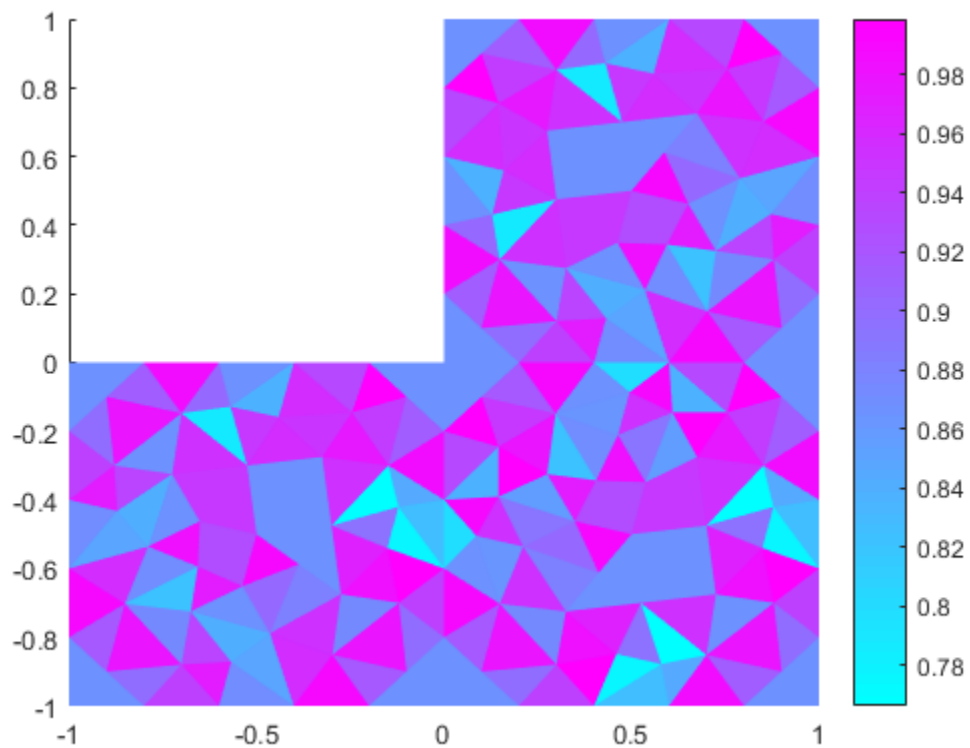
```
p1 = jigglemesh(p,e,t,'opt','mean','iter',inf);
q = pdetriq(p1,t);
pdeplot(p1,e,t,'XYData',q,'ColorBar','on','XYStyle','flat')
```
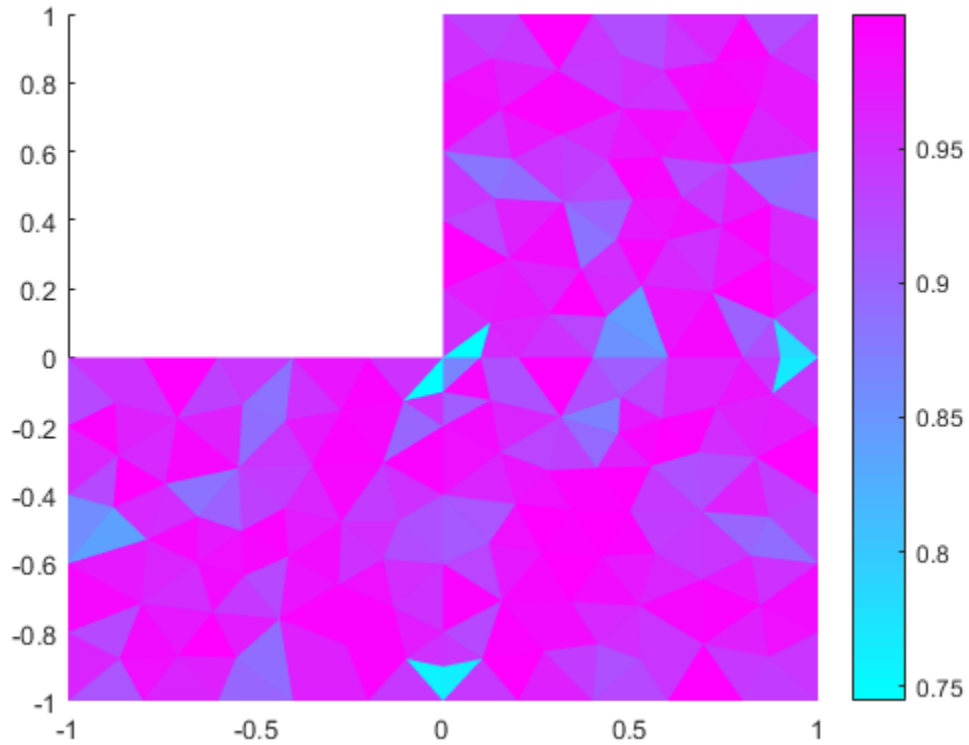
## See Also

`initmesh` | `pdetriq`

## Topics

"Mesh Data" on page 2-196

**Introduced before R2006a**

# meshToPet

`[p,e,t]` representation of `FEMesh` data

## Compatibility

This page describes the legacy workflow. 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
```
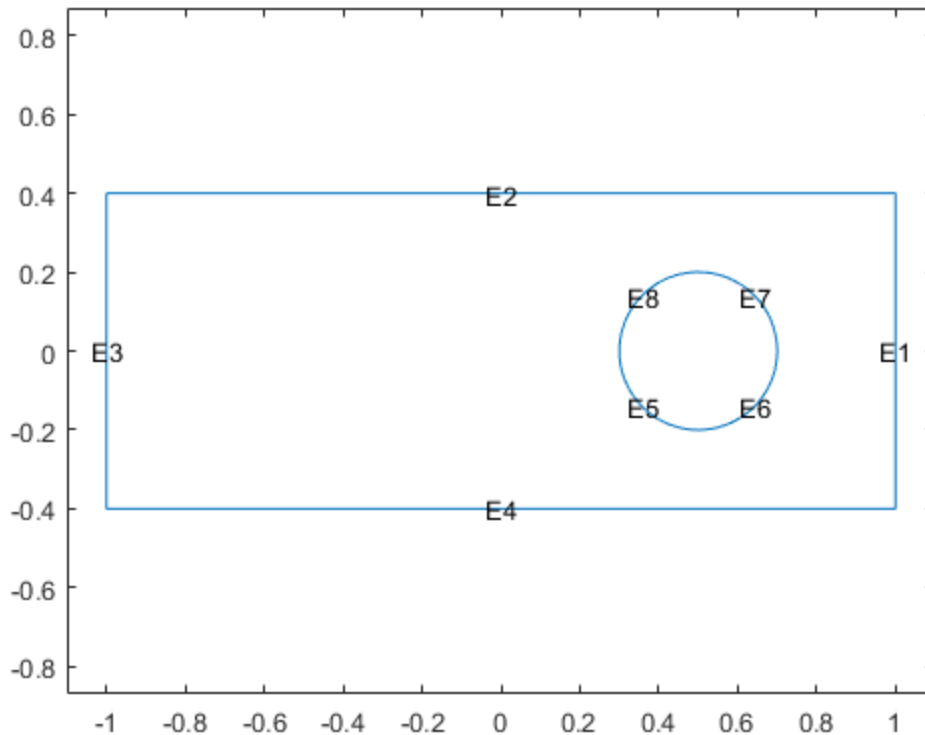


Create a mesh for the geometry. View the mesh.

```
generateMesh(model);
pdemesh(model)
axis equal
```
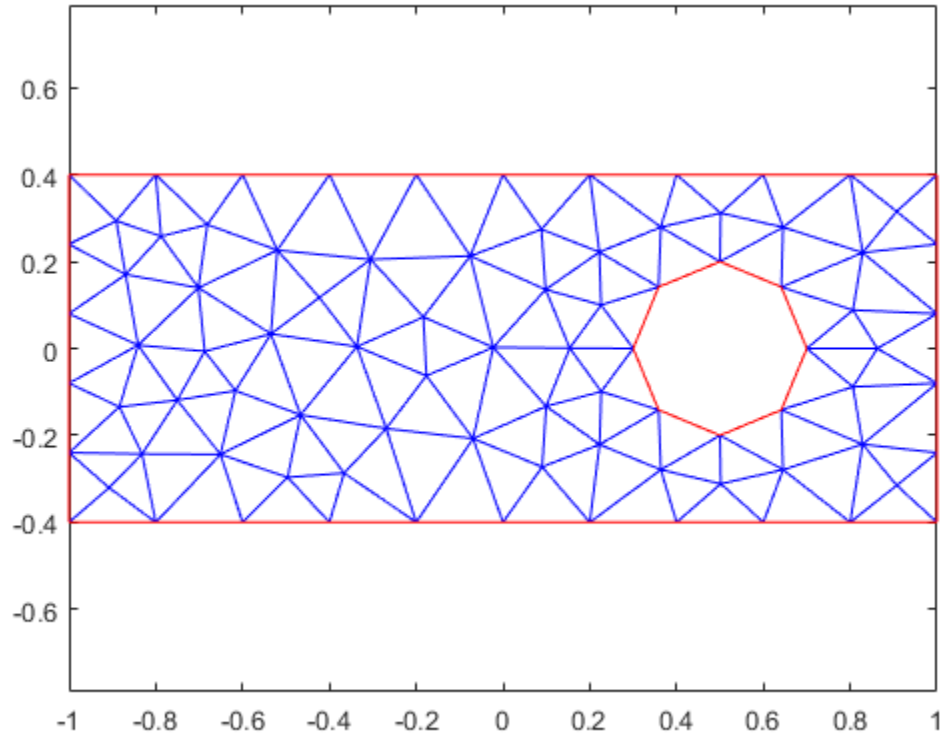
Convert the mesh to [p,e,t] form.

```
[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 2-196.

**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 2-196.

**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 2-196. The last row, t(end,k), contains the subdomain number of the element.

# Tips

- Use meshToPet to obtain the p and t data for interpolation using pdeInterpolant.

# See Also

## See Also
FEMesh Properties | generateMesh

## Topics
"Mesh Data" on page 2-196

**Introduced in R2015a**

# multicuboid

Create geometry formed by several cubic cells

## Compatibility

This page describes the recommended workflow.

## Syntax

```
gm = multicuboid(W,D,H)
gm = multicuboid(W,D,H,Name,Value)
```

## Description

`gm = multicuboid(W,D,H)` creates a geometry by combining several cubic cells.

When creating each cuboid, `multicuboid` uses the following coordinate system.
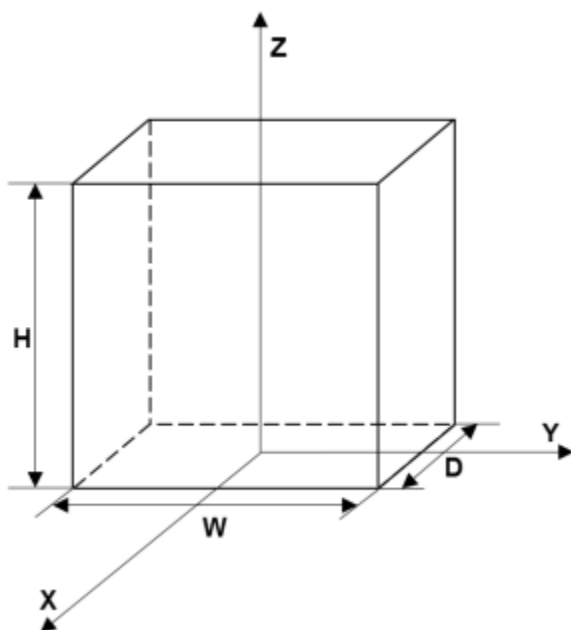
gm = multicuboid(W,D,H,Name,Value) creates a multi-cuboid geometry using one or more Name,Value pair arguments.

## Examples

### Nested Cuboids of the Same Height

Create a geometry that consists of three nested cuboids of the same height and include this geometry in a PDE model.

Create the geometry by using the multicuboid function. The resulting geometry consists of three cells.

```
gm = multicuboid([2 3 5],[4 6 10],3)


gm =

  DiscreteGeometry with properties:
```

```
      NumCells: 3
      NumFaces: 18
      NumEdges: 36
   NumVertices: 24
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: []
      EquationCoefficients: []
       BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: [1×1 DiscreteGeometry]
      EquationCoefficients: []
       BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on','FaceAlpha',0.5)
```



## Stacked Cuboids

Create a geometry that consists of four stacked cuboids and include this geometry in a PDE model.

Create the geometry by using the `multicuboid` function with the `ZOffset` argument. The resulting geometry consists of four cells stacked on top of each other.

```
gm = multicuboid(5,10,[1 2 3 4],'ZOffset',[0 1 3 6])
```

```
gm =
```

```
    DiscreteGeometry with properties:

          NumCells: 4
          NumFaces: 21
          NumEdges: 36
       NumVertices: 20
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: 0
                 Geometry: []
      EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: 0
                 Geometry: [1×1 DiscreteGeometry]
      EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on','FaceAlpha',0.5)
```



## Single Cuboid

Create a geometry that consists of a single cuboid and include this geometry in a PDE model.

Use the `multicuboid` function to create a single cuboid. The resulting geometry consists of one cell.

```
gm = multicuboid(5,10,7)
```

```
gm =

  DiscreteGeometry with properties:

        NumCells: 1
        NumFaces: 6
        NumEdges: 12
     NumVertices: 8
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

             PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: []
      EquationCoefficients: []
        BoundaryConditions: []
          InitialConditions: []
                     Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

             PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: [1×1 DiscreteGeometry]
      EquationCoefficients: []
        BoundaryConditions: []
          InitialConditions: []
                     Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on')
```



## Hollow Cube

Create a hollow cube and include it as a geometry in a PDE model.

Create a hollow cube by using the `multicuboid` function with the `Void` argument. The resulting geometry consists of one cell.

```
gm = multicuboid([6 10],[6 10],10,'Void',[true,false])
```

```
gm =

  DiscreteGeometry with properties:

        NumCells: 1
        NumFaces: 10
        NumEdges: 24
     NumVertices: 16
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: []
     EquationCoefficients: []
       BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: [1×1 DiscreteGeometry]
     EquationCoefficients: []
       BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on','FaceAlpha',0.5)
```



## Input Arguments

### W — Cell width
positive real number | vector of positive real numbers

Cell width, specified as a positive real number or a vector of positive real numbers. If W is a vector, then W(i) specifies the width of the ith cell.

Width W, depth D, and height H can be scalars or vectors of the same length. For a combination of scalar and vector inputs, `multicuboid` replicates the scalar arguments into vectors of the same length.

---

**Note:** All cells in the geometry either must have the same height, or must have both the same width and the same depth.

---

Example: `gm = multicuboid([1 2 3],[2.5 4 5.5],5)`

### D — Cell depth
positive real number | vector of positive real numbers

Cell depth, specified as a positive real number or a vector of positive real numbers. If D is a vector, then `D(i)` specifies the depth of the ith cell.

Width W, depth D, and height H can be scalars or vectors of the same length. For a combination of scalar and vector inputs, `multicuboid` replicates the scalar arguments into vectors of the same length.

---

**Note:** All cells in the geometry either must have the same height, or must have both the same width and the same depth.

---

Example: `gm = multicuboid([1 2 3],[2.5 4 5.5],5)`

### H — Cell height
positive real number | vector of positive real numbers

Cell height, specified as a positive real number or a vector of positive real numbers. If H is a vector, then `H(i)` specifies the height of the ith cell.

Width W, depth D, and height H can be scalars or vectors of the same length. For a combination of scalar and vector inputs, `multicuboid` replicates the scalar arguments into vectors of the same length.

---

**Note:** All cells in the geometry either must have the same height, or must have both the same width and the same depth.

---

Example: gm = multicuboid(4,5,[1 2 3],'ZOffset',[0 1 3])

## 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: gm = multicuboid([1 2],[1 2],[3 3],'Void',[true,false])

### 'ZOffset' — Z offset for each cell
vector of 0 values (default) | vector of real numbers

Z offset for each cell, specified as a vector of real numbers. ZOffset(i) specifies the Z offset of the ith cell. This vector must have the same length as the width vector W, depth vector D, or height vector H.

---

**Note:** The ZOffset argument is valid only if the width and depth are constant for all cells in the geometry.

---

Example: gm = multicuboid(20,30,[10 10],'ZOffset',[0 10])

Data Types: double

### 'Void' — Empty cell indicator
vector of logical false values (default) | vector of logical true or false values

Empty cell indicator, specified as a vector of logical true or false values. This vector must have the same length as the width vector W, depth vector D, or the height vector H.

The value true corresponds to an empty cell. By default, multicuboid assumes that all cells are not empty.

Example: gm = multicuboid([1 2],[1 2],[3 3],'Void',[true,false])

Data Types: double

## Output Arguments

**gm — Geometry object**
`DiscreteGeometry` object

Geometry object, returned as a `DiscreteGeometry` object.

## Limitations

- `multicuboid` lets you create only geometries consisting of stacked or nested cuboids. For nested cuboids, the height must be the same for all cells in the geometry. For stacked cuboids, the width and depth must be the same for all cells in the geometry. Use the `ZOffset` argument to stack the cells on top of each other without overlapping them.

- `multicuboid` does not let you create nested cuboids of the same width and depth. The call `multicuboid(w,d,[h1,h2,...])` is not supported.

## See Also

### See Also
`DiscreteGeometry` | `multicylinder` | `multisphere`

**Introduced in R2017a**

# multicylinder

Create geometry formed by several cylindrical cells

## Compatibility

This page describes the recommended workflow.

## Syntax

```
gm = multicylinder(R,H)
gm = multicylinder(R,H,Name,Value)
```

## Description

`gm = multicylinder(R,H)` creates a geometry by combining several cylindrical cells.

When creating each cylinder, `multicylinder` uses the following coordinate system.
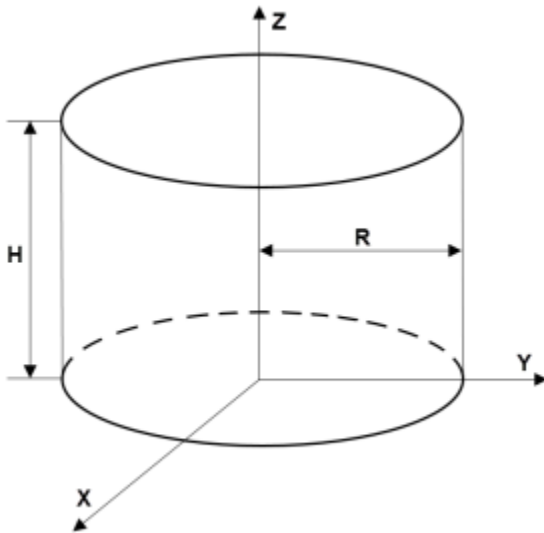
`gm = multicylinder(R,H,Name,Value)` creates a multi-cylinder geometry using one or more `Name,Value` pair arguments.

# Examples

### Nested Cylinders of the Same Height

Create a geometry that consists of three nested cylinders of the same height and include this geometry in a PDE model.

Create the geometry by using the `multicylinder` function. The resulting geometry consists of three cells.

```
gm = multicylinder([5 10 15],2)
```

```
gm =

  DiscreteGeometry with properties:

        NumCells: 3
        NumFaces: 9
        NumEdges: 6
     NumVertices: 6
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: []
      EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
```

```
            SolverOptions: [1×1 PDESolverOptions]
```
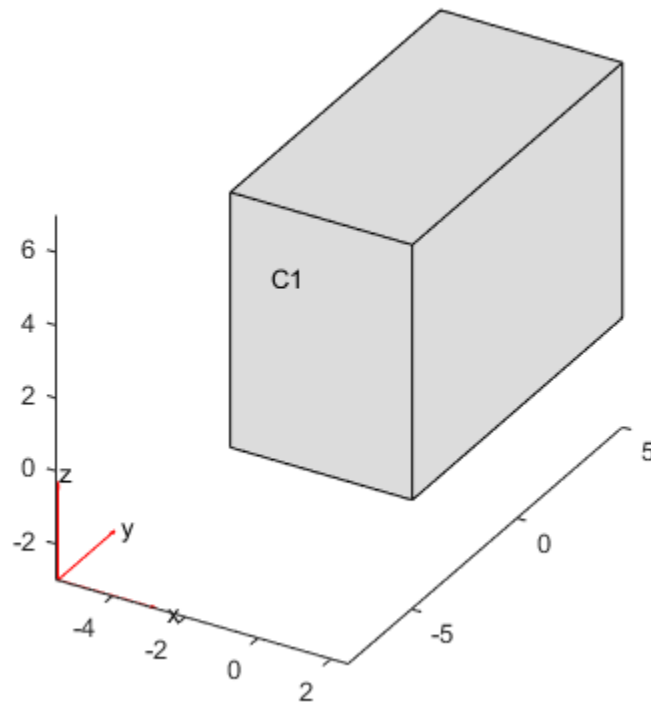
Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
         IsTimeDependent: O
                Geometry: [1×1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on','FaceAlpha',0.5)
```

### Stacked Cylinders

Create a geometry that consists of three stacked cylinders and include this geometry in a PDE model.

Create the geometry by using the `multicylinder` function with the `ZOffset` argument. The resulting geometry consists of four cells stacked on top of each other.

```
gm = multicylinder(10,[1 2 3 4],'ZOffset',[0 1 3 6])
```

```
gm =

  DiscreteGeometry with properties:
```

```
       NumCells: 4
       NumFaces: 9
       NumEdges: 5
    NumVertices: 5
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: []
      EquationCoefficients: []
        BoundaryConditions: []
          InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

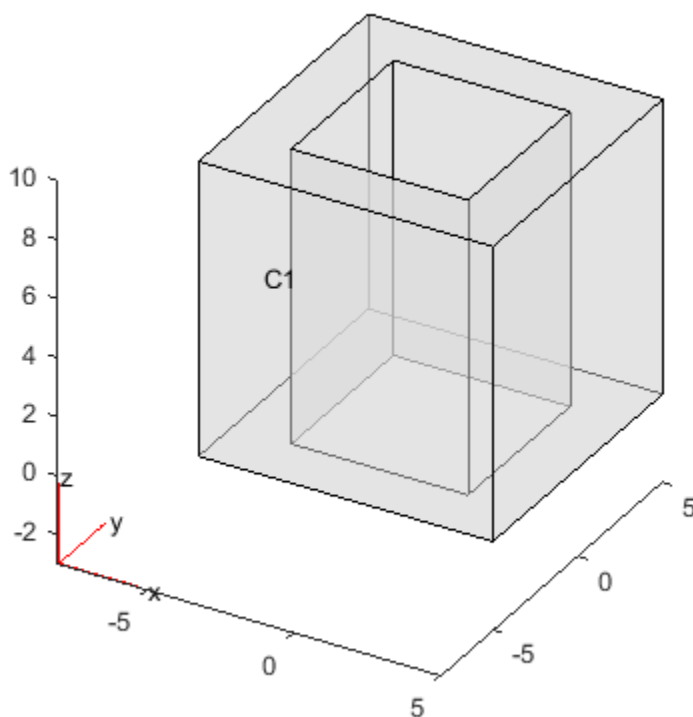Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: [1×1 DiscreteGeometry]
      EquationCoefficients: []
        BoundaryConditions: []
          InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on','FaceAlpha',0.5)
```



### Single Cylinder

Create a geometry that consists of a single cylinder and include this geometry in a PDE model.

Use the `multicylinder` function to create a single cylinder. The resulting geometry consists of one cell.

```
gm = multicylinder(5,10)

gm =
```

```
   DiscreteGeometry with properties:

        NumCells: 1
        NumFaces: 3
        NumEdges: 2
     NumVertices: 2
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
      EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1×1 DiscreteGeometry]
      EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on')
```



### Hollow Cylinder

Create a hollow cylinder and include it as a geometry in a PDE model.

Create a hollow cylinder by using the `multicylinder` function with the `Void` argument. The resulting geometry consists of one cell.

```
gm = multicylinder([9 10],10,'Void',[true,false])
```

```
gm =
```

```
    DiscreteGeometry with properties:

          NumCells: 1
          NumFaces: 4
          NumEdges: 4
       NumVertices: 4
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

             PDESystemSize: 1
           IsTimeDependent: O
                  Geometry: []
       EquationCoefficients: []
         BoundaryConditions: []
          InitialConditions: []
                      Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

             PDESystemSize: 1
           IsTimeDependent: O
                  Geometry: [1×1 DiscreteGeometry]
       EquationCoefficients: []
         BoundaryConditions: []
          InitialConditions: []
                      Mesh: []
              SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on','FaceAlpha',0.5)
```



## Input Arguments

### R — Cell radius
positive real number | vector of positive real numbers

Cell radius, specified as a positive real number or a vector of positive real numbers. If R is a vector, then R(i) specifies the radius of the ith cell.

Radius R and height H can be scalars or vectors of the same length. For a combination of scalar and vector inputs, `multicylinder` replicates the scalar arguments into vectors of the same length.

---

**Note:** Either radius or height must be the same for all cells in the geometry.

---

Example: gm = multicylinder([1 2 3],1,'Zoffset',[0 1 3])

### H — Cell height
positive real number | vector of positive real numbers

Cell height, specified as a positive real number or a vector of positive real numbers. If H is a vector, then `H(i)` specifies the height of the `ith` cell.

Radius R and height H can be scalars or vectors of the same length. For a combination of scalar and vector inputs, `multicylinder` replicates the scalar arguments into vectors of the same length.

---

**Note:** Either radius or height must be the same for all cells in the geometry.

---

Example: gm = multicylinder(1,[1 2 3])

## 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: gm = multicylinder([1 2],1,'Void',[true,false])

### 'ZOffset' — Z-offset for each cell
vector of 0 values (default) | vector of real numbers

Z-offset for each cell, specified as a vector of real numbers. `ZOffset(i)` specifies the Z-offset of the `ith` cell. This vector must have the same length as the radius vector R or height vector H.

---

**Note:** The ZOffset argument is valid only if the radius is the same for all cells in the geometry.

---

Example: gm = multicylinder(20,[10 10],'ZOffset',[0 10])

Data Types: double

### 'Void' — Empty cell indicator
vector of logical false values (default) | vector of logical true or false values

Empty cell indicator, specified as a vector of logical true or false values. This vector must have the same length as the radius vector R or the height vector H.

The value true corresponds to an empty cell. By default, multicylinder assumes that all cells are not empty.

Example: gm = multicylinder([1 2],1,'Void',[true,false])

Data Types: double

## Output Arguments

### gm — Geometry object
DiscreteGeometry object

Geometry object, returned as a DiscreteGeometry object.

## Limitations

- multicylinder lets you create only geometries consisting of stacked or nested cylinders. For nested cylinders, the height must be the same for all cells in the geometry. For stacked cylinders, the radius must be the same for all cells in the geometry. Use the ZOffset argument to stack the cells on top of each over without overlapping them.

- multicylinder does not let you create nested cylinders of the same radius. The call multicylinder(r,[h1,h2,...]) is not supported.

# See Also

## See Also
DiscreteGeometry | multicuboid | multisphere

**Introduced in R2017a**

# multisphere

Create geometry formed by several spherical cells

## Compatibility

This page describes the recommended workflow.

## Syntax

```
gm = multisphere(R)
gm = multisphere(R,Name,Value)
```

## Description

`gm = multisphere(R)` creates a geometry by combining several spherical cells.

When creating each sphere, `multisphere` uses the following coordinate system.
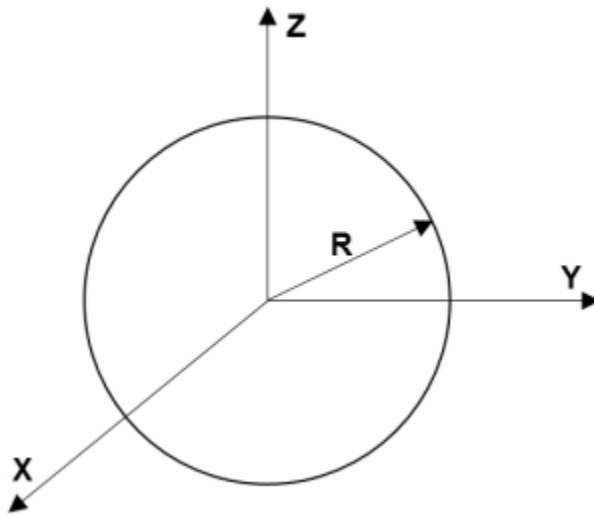
gm = `multisphere(R,Name,Value)` creates a multi-sphere geometry using one or more `Name,Value` pair arguments.

# Examples

### Nested Spheres

Create a geometry that consists of three nested spheres and include this geometry in a PDE model.

Create the geometry by using the `multisphere` function. The resulting geometry consists of three cells.

```
gm = multisphere([5 10 15])


gm =

  DiscreteGeometry with properties:

        NumCells: 3
        NumFaces: 3
        NumEdges: O
     NumVertices: O
```

Create a PDE model.

```
model = createpde


model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: []
       EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
```

```
            SolverOptions: [1×1 PDESolverOptions]
```
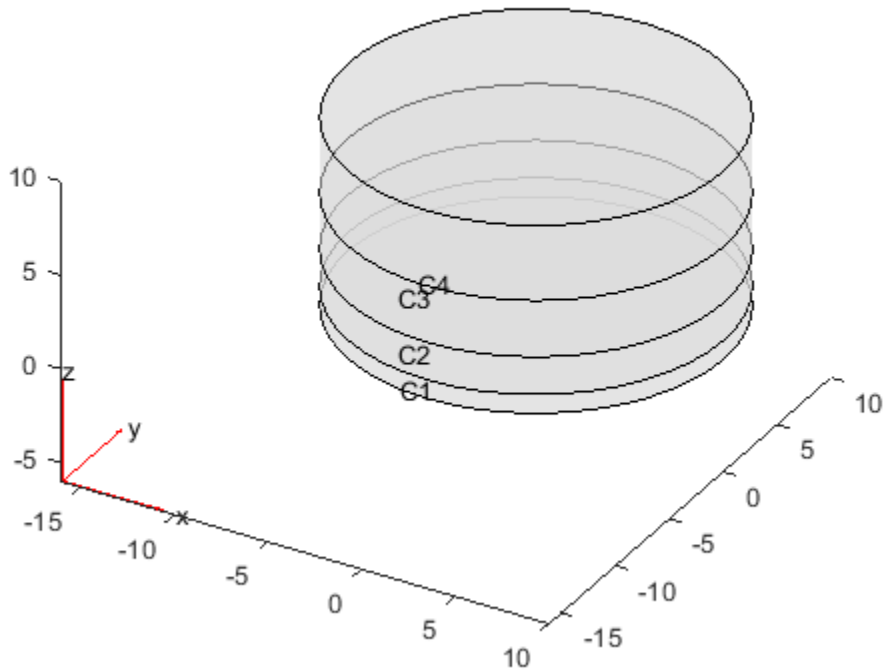
Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
         IsTimeDependent: O
                Geometry: [1×1 DiscreteGeometry]
     EquationCoefficients: []
       BoundaryConditions: []
        InitialConditions: []
                     Mesh: []
            SolverOptions: [1×1 PDESolverOptions]
```
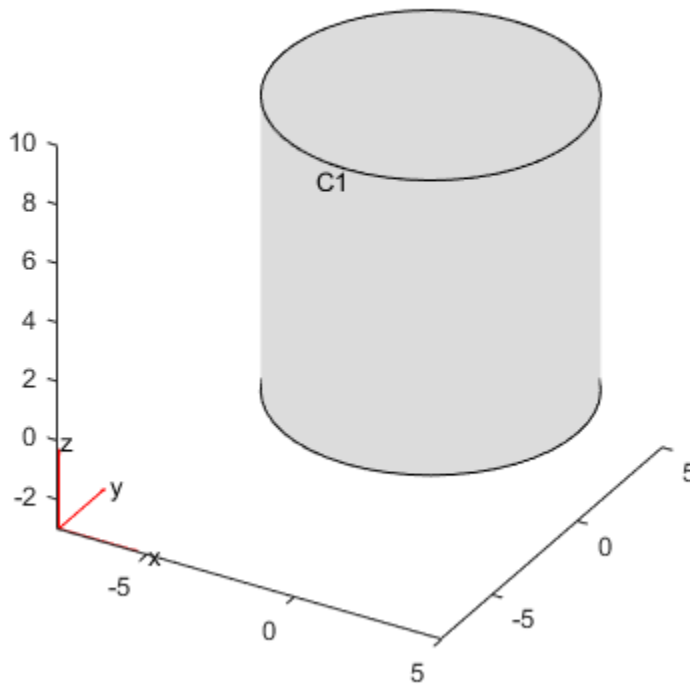
Plot the geometry.

```
pdegplot(model,'CellLabels','on','FaceAlpha',0.2)
```

### Single Sphere

Create a geometry that consists of a single sphere and include this geometry in a PDE model.

Use the `multisphere` function to create a single sphere. The resulting geometry consists of one cell.

```
gm = multisphere(5)
```

```
gm =

  DiscreteGeometry with properties:
```

```
        NumCells: 1
        NumFaces: 1
        NumEdges: O
     NumVertices: O
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: []
       EquationCoefficients: []
         BoundaryConditions: []
          InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

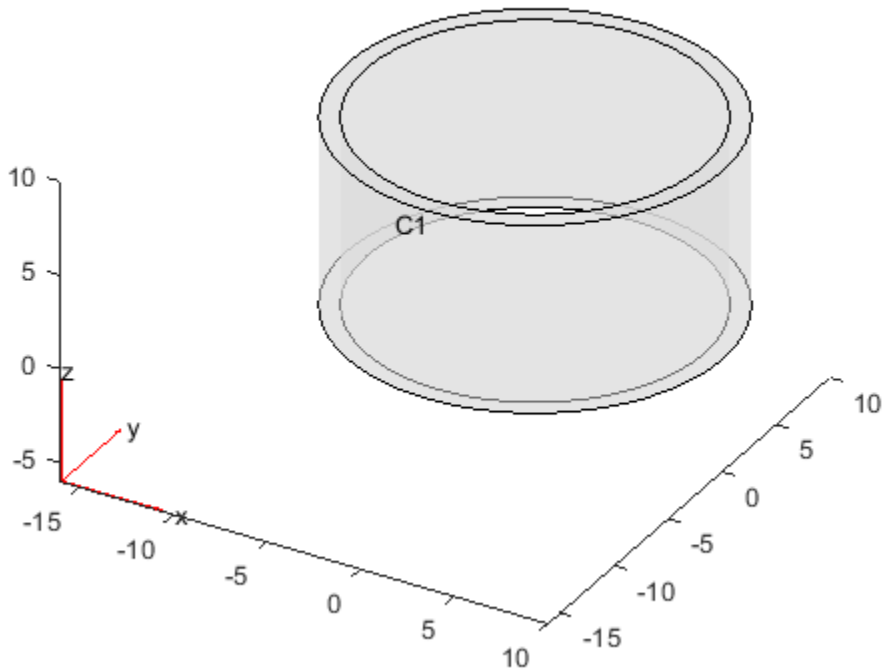```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: O
                 Geometry: [1×1 DiscreteGeometry]
       EquationCoefficients: []
         BoundaryConditions: []
          InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,'CellLabels','on')
```



### Hollow Sphere

Create a hollow sphere and include it as a geometry in a PDE model.

Create a hollow sphere by using the `multisphere` function with the `Void` argument.
The resulting geometry consists of one cell.

```
gm = multisphere([9 10],'Void',[true,false])
```

gm =

```
DiscreteGeometry with properties:

      NumCells: 1
      NumFaces: 2
      NumEdges: 0
   NumVertices: 0
```

Create a PDE model.

```
model = createpde
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: 0
                 Geometry: []
      EquationCoefficients: []
       BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model =

  PDEModel with properties:

            PDESystemSize: 1
          IsTimeDependent: 0
                 Geometry: [1×1 DiscreteGeometry]
      EquationCoefficients: []
       BoundaryConditions: []
         InitialConditions: []
                     Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

- "Heat Conduction in a Spherical Multidomain Geometry with Nonuniform Heat Flux"

# Input Arguments

### R — Cell radius
positive real number | vector of positive real numbers

Cell radius, specified as a positive real number or a vector of positive real numbers. If R is a vector, then R(i) specifies the radius of the ith cell.

Example: gm = multisphere([1,2,3])

## 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: gm = multisphere([1,2],'Void',[true,false])

### 'Void' — Empty cell indicator
vector of logical false values (default) | vector of logical true or false values

Empty cell indicator, specified as a vector of logical true or false values. This vector must have the same length as the radius vector R.

The value true corresponds to an empty cell. By default, multisphere assumes that all cells are not empty.

Example: gm = multisphere([1,2,3],'Void',[false,true,false])

Data Types: double

# Output Arguments

### gm — Geometry object
DiscreteGeometry object

Geometry object, returned as a `DiscreteGeometry` object.

# See Also

### See Also
`DiscreteGeometry` | `multicuboid` | `multicylinder`

### Topics
"Heat Conduction in a Spherical Multidomain Geometry with Nonuniform Heat Flux"

**Introduced in R2017a**

# 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 $\Omega$, or the system PDE problem

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

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 $\Omega$, or the system PDE problem

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

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
```
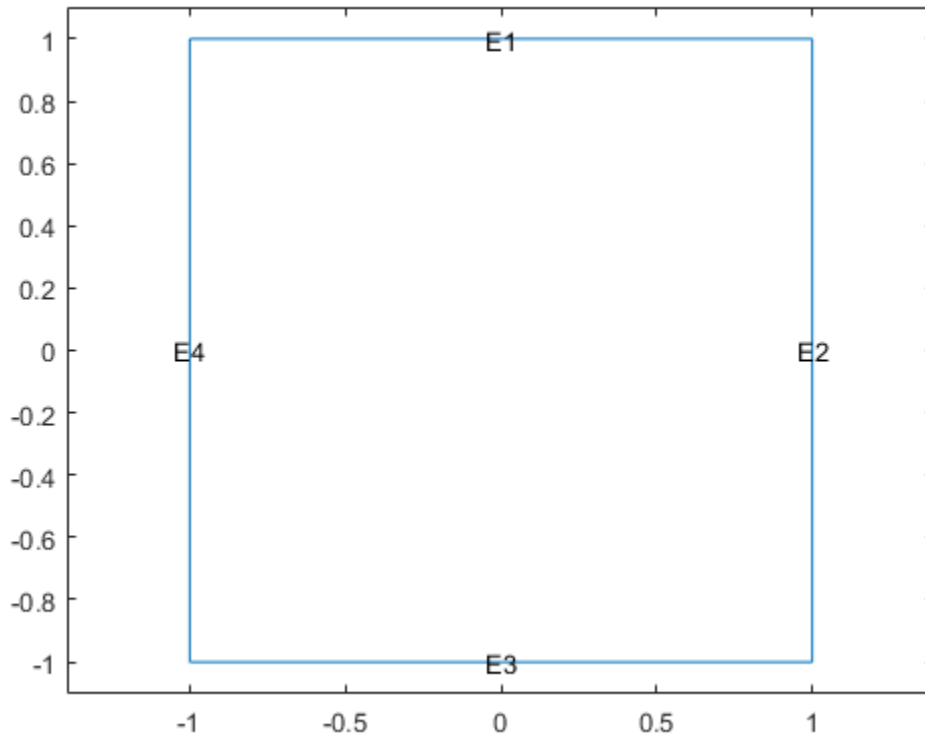


Set Dirichlet boundary conditions $u = 0$ on all edges.

```
applyBoundaryCondition(model,'dirichlet','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 $x^2 + y^2 \le 0.4^2$ 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')
```
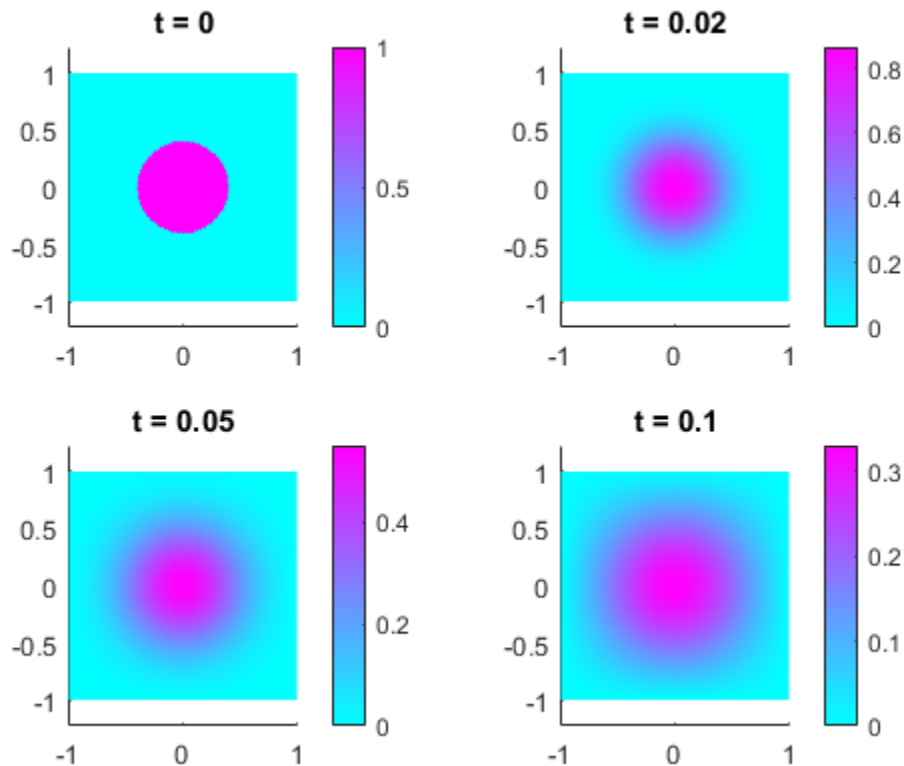


### Parabolic Equation Using Legacy Syntax

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.

**6-411**

```
g = @squareg;
pdegplot(g,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```
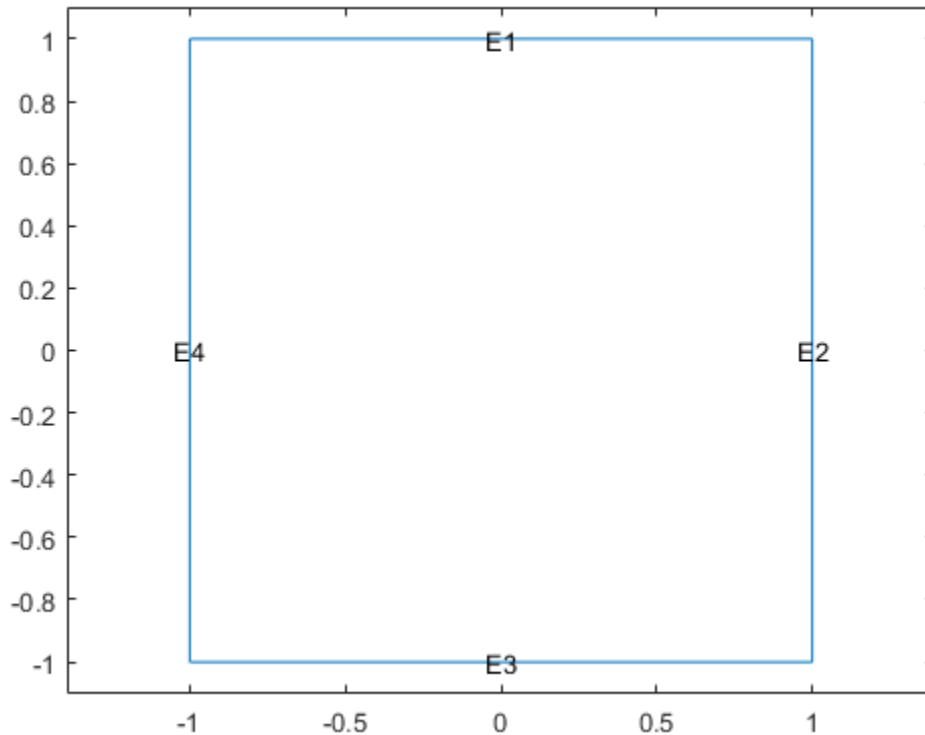


Set Dirichlet boundary conditions $u = 0$ on all edges. The `squareb1` function specifies these boundary conditions.

```
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')
```
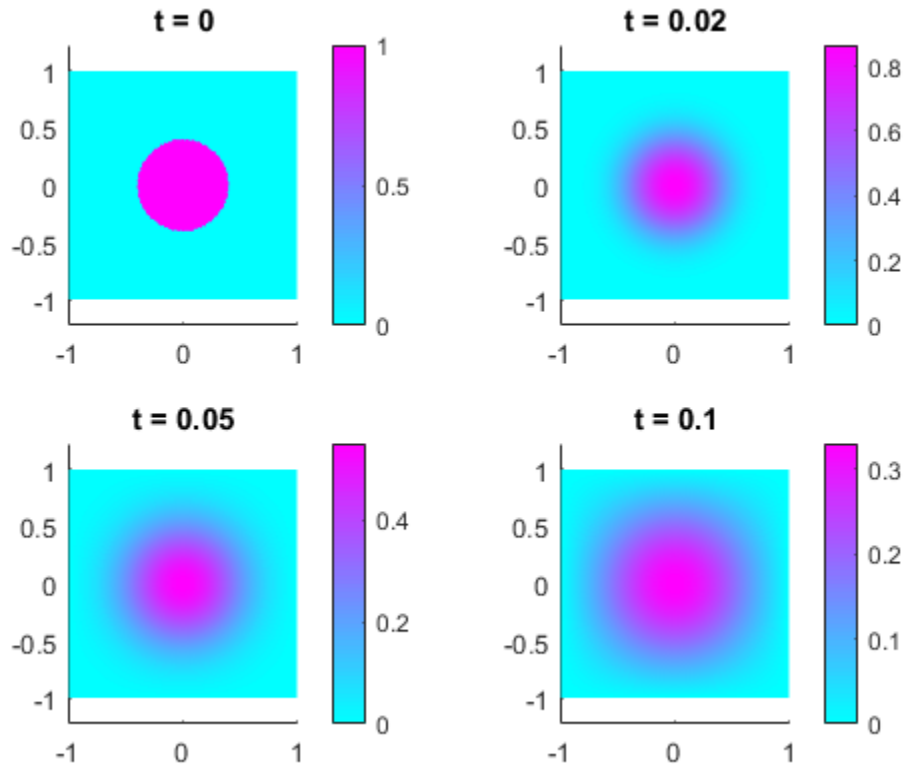


### Parabolic Problem Using Matrix Coefficients

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;
```
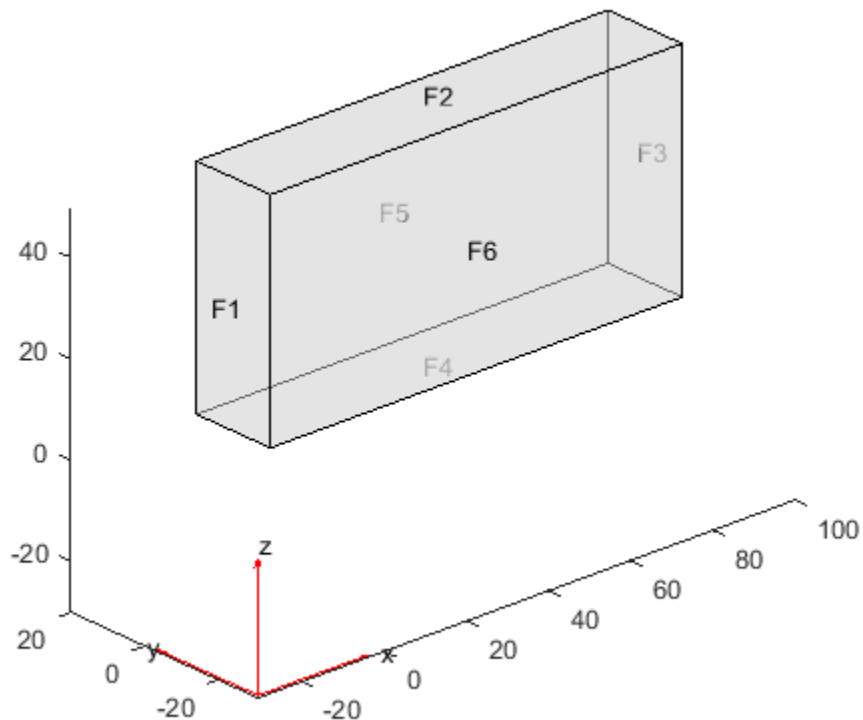
Faces 1, 4, and 6 of the slab are kept at 0 degrees. The other faces are insulated. Include the boundary condition on faces 1, 4, and 6. You do not need to include the boundary condition on the other faces because the default condition is insulated.

```
applyBoundaryCondition(model,'dirichlet','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 2-147.

- 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

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

You can specify c in various ways, detailed in "c Coefficient for Systems" on page 2-110. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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

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

You can specify a in various ways, detailed in "a or d Coefficient for Systems" on page 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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

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

You can specify f in various ways, detailed in "f Coefficient for Systems" on page 2-83. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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

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

You can specify d in various ways, detailed in "a or d Coefficient for Systems" on page 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 file name.

- 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 2-154.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-183.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

### p — Mesh points
matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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

Mesh triangles, specified as a `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 2-196.

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`

### 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 6-62. Typically, `B` is the output of `assempde`.

### ud — Dirichlet vector
vector

Dirichlet vector, returned as a vector. See "Algorithms" on page 6-62. 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,O,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(uO,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 3-210.

# 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.

# See Also

### See Also
solvepde

### Topics
"PDE Problem Setup"

**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 2-196.

`c`,`a`, and `f` are PDE coefficients.

`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

**Introduced before R2006a**

# 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 2-196.

c, a, and f are PDE coefficients.

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

**Introduced before R2006a**

# 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".

**Introduced before R2006a**

# pdeBoundaryConditions class

(To be removed) Boundary conditions

## Compatibility

pdeBoundaryConditions has been removed. 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

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

See "Boundary Conditions for PDE Systems" on page 2-188.

---

**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 "Equations You Can Solve Using Legacy Functions" on page 1-3. For the syntax of the function handle form of $r$, see "Nonconstant Boundary Conditions" on page 2-165.

---

**Note:** You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-431): 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 "Equations You Can Solve Using Legacy Functions" on page 1-3. For the syntax of the function handle form of `h`, see "Nonconstant Boundary Conditions" on page 2-165.

**Note:** You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-431): 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·(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. See "Equations You Can Solve Using Legacy Functions" on page 1-3. For the syntax of the function handle form of `g`, see "Nonconstant Boundary Conditions" on page 2-165.

**Note:** You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-431): 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 $n \cdot (c \times \nabla u)$ `+ qu = g`
`zeros(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 a function handle. N is the number of PDEs in the system. See "Equations You Can Solve Using Legacy Functions" on page 1-3. For the syntax of the function handle form of `q`, see "Nonconstant Boundary Conditions" on page 2-165.

---

**Note:** You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-431): 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 "Nonconstant Boundary Conditions" on page 2-165.

---

**Note:** You can set only one type of boundary condition in a call to `pdeBoundaryConditions` (see "Description" on page 6-431): 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 6-431): 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" (MATLAB). For details, see "Nonconstant Boundary Conditions" on page 2-165.

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·(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. See "Equations You Can Solve Using Legacy Functions" on page 1-3. For details of the function handle syntax, see "Nonconstant Boundary Conditions" on page 2-165.

### 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 "Equations You Can Solve Using Legacy Functions" on page 1-3. For details of the function handle syntax, see "Nonconstant Boundary Conditions" on page 2-165.

### 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·(c×∇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 "Equations You Can Solve Using Legacy Functions" on page 1-3. For details of the function handle syntax, see "Nonconstant Boundary Conditions" on page 2-165.

### 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 "Equations You Can Solve Using Legacy Functions" on page 1-3. For details of the function handle syntax, see "Nonconstant Boundary Conditions" on page 2-165.

### 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 "Equations You Can Solve Using Legacy Functions" on page 1-3. The length of `u` must equal the length of

EquationIndex. For details of the function handle syntax, see "Nonconstant Boundary Conditions" on page 2-165.

### `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 "Nonconstant Boundary Conditions" on page 2-165.

## See Also

### See Also
applyBoundaryCondition | 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 `evaluateCGradient`.

## 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, $\mathbf{c} \otimes \nabla \mathbf{u}$, 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 gradient computed by pdegrad is actually the same everywhere in the triangle interior because pdegrad uses only linear basis functions. The boundaries of triangles are a special case: here the derivatives might be discontinuous. However, the flux $\mathbf{c} \otimes \nabla \mathbf{u}$ can vary inside a triangle because the coefficient c can vary.

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 | evaluateCGradient

**Introduced before R2006a**

# 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

**Introduced before R2006a**

# 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 2-196.

`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

### Contour Plot of the Solution

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)
```
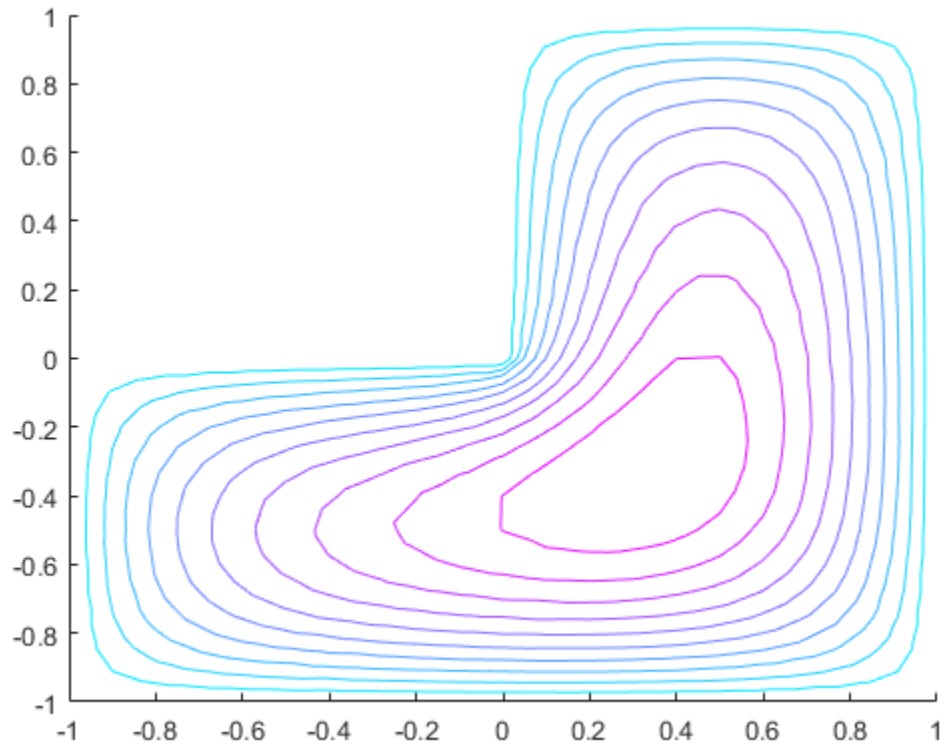
## See Also

pdemesh | pdeplot | pdesurf

**Introduced before R2006a**

# 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

$$-\nabla \cdot (\mathbf{c} \otimes \nabla \mathbf{u}) + \mathbf{au} = \lambda d u \text{ on } \Omega$$

with geometry, boundary conditions, and mesh specified in `model`, a `PDEModel` object. See "Solve Problems Using Legacy PDEModel Objects" on page 2-3.

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($\lambda$) 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
```

```
Basis= 10,  Time=  1.39,  New conv eig=  0
Basis= 11,  Time=  1.39,  New conv eig=  0
Basis= 12,  Time=  1.40,  New conv eig=  0
Basis= 13,  Time=  1.40,  New conv eig=  0
Basis= 14,  Time=  1.59,  New conv eig=  0
Basis= 15,  Time=  1.59,  New conv eig=  1
Basis= 16,  Time=  1.59,  New conv eig=  1
Basis= 17,  Time=  1.59,  New conv eig=  3
Basis= 18,  Time=  1.61,  New conv eig=  4
Basis= 19,  Time=  1.61,  New conv eig=  4
Basis= 20,  Time=  1.70,  New conv eig=  4
Basis= 21,  Time=  1.70,  New conv eig=  4
Basis= 22,  Time=  1.72,  New conv eig=  4
Basis= 23,  Time=  1.72,  New conv eig=  4
Basis= 24,  Time=  1.72,  New conv eig=  5
Basis= 25,  Time=  1.93,  New conv eig=  5
Basis= 26,  Time=  1.93,  New conv eig=  5
Basis= 27,  Time=  1.95,  New conv eig=  6
Basis= 28,  Time=  1.95,  New conv eig=  7
Basis= 29,  Time=  2.04,  New conv eig=  7
```

```
              Basis= 30,  Time=   2.15,  New conv eig=  7
              Basis= 31,  Time=   2.15,  New conv eig=  7
              Basis= 32,  Time=   2.17,  New conv eig=  8
              Basis= 33,  Time=   2.17,  New conv eig=  8
              Basis= 34,  Time=   2.25,  New conv eig=  8
              Basis= 35,  Time=   2.25,  New conv eig=  9
              Basis= 36,  Time=   2.25,  New conv eig=  9
              Basis= 37,  Time=   2.39,  New conv eig=  9
              Basis= 38,  Time=   2.39,  New conv eig=  9
              Basis= 39,  Time=   2.45,  New conv eig=  9
              Basis= 40,  Time=   2.45,  New conv eig=  9
              Basis= 41,  Time=   2.61,  New conv eig=  9
              Basis= 42,  Time=   2.61,  New conv eig= 10
              Basis= 43,  Time=   2.64,  New conv eig= 11
              Basis= 44,  Time=   2.70,  New conv eig= 12
              Basis= 45,  Time=   2.70,  New conv eig= 12
              Basis= 46,  Time=   2.70,  New conv eig= 14
              Basis= 47,  Time=   2.71,  New conv eig= 15
              Basis= 48,  Time=   2.82,  New conv eig= 16
              Basis= 49,  Time=   2.93,  New conv eig= 17
              Basis= 50,  Time=   2.93,  New conv eig= 17
              Basis= 51,  Time=   3.09,  New conv eig= 18
              Basis= 52,  Time=   3.09,  New conv eig= 18
              Basis= 53,  Time=   3.28,  New conv eig= 19
              Basis= 54,  Time=   3.43,  New conv eig= 19
              Basis= 55,  Time=   3.59,  New conv eig= 22
              Basis= 56,  Time=   3.74,  New conv eig= 24
              Basis= 57,  Time=   3.74,  New conv eig= 28
End of sweep: Basis= 57,  Time=   3.74,  New conv eig= 28
              Basis= 38,  Time=   4.54,  New conv eig=  0
              Basis= 39,  Time=   4.71,  New conv eig=  0
              Basis= 40,  Time=   4.71,  New conv eig=  0
              Basis= 41,  Time=   4.79,  New conv eig=  0
              Basis= 42,  Time=   4.79,  New conv eig=  0
End of sweep: Basis= 42,  Time=   4.79,  New conv eig=  0

ans =

    9.6481
```

Display the first eigenmode, and compare it to the built-in membrane plot.

```
pdeplot(model,'XYData',v(:,1),'ZData',v(:,1))
```
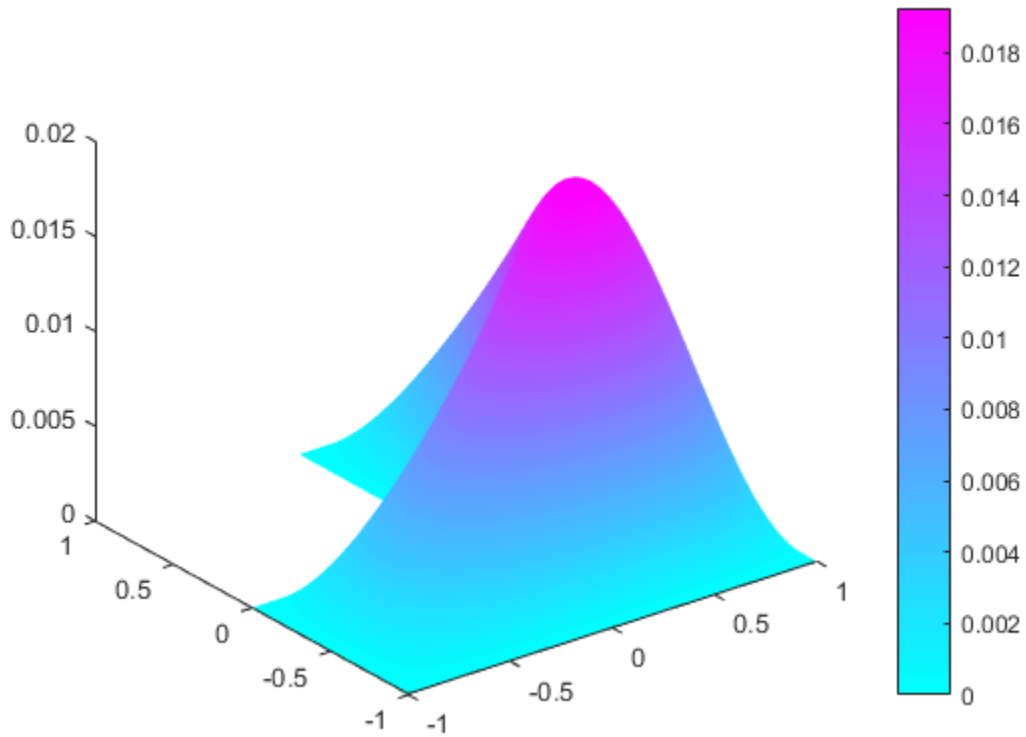
```
figure
membrane(1,20,9,9)        % the MATLAB function
```
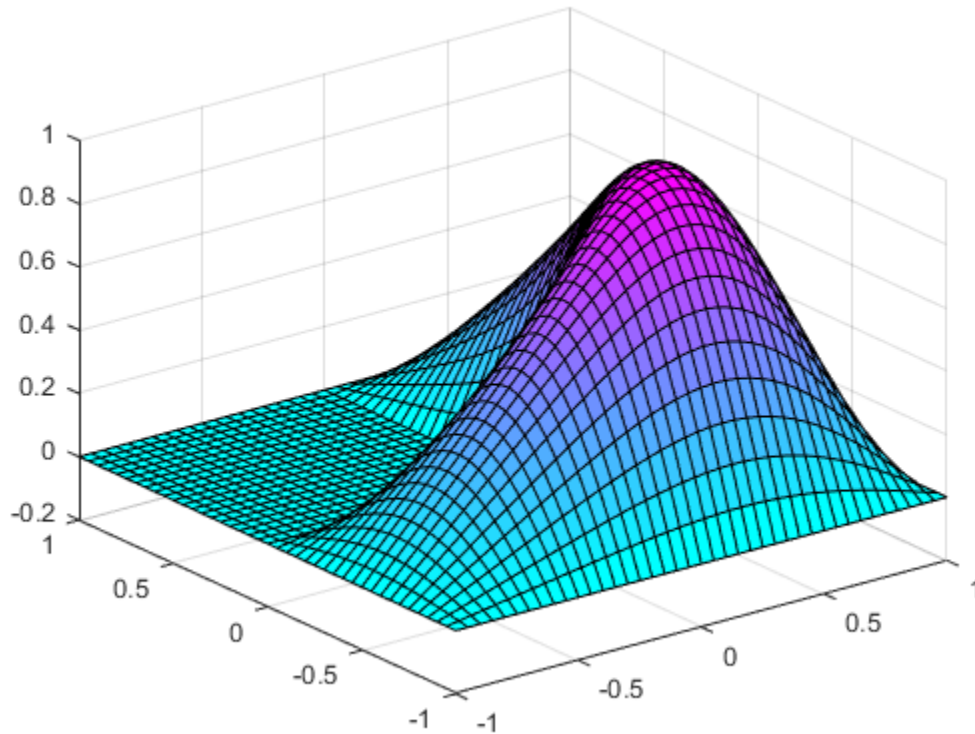
Compute the sixteenth eigenvalue, and plot the sixteenth eigenmode.

```
l(16)                        % sixteenth eigenvalue


ans =

   92.4658


figure
pdeplot(model,'XYData',v(:,16),'ZData',v(:,16))    % sixteenth eigenmode
```
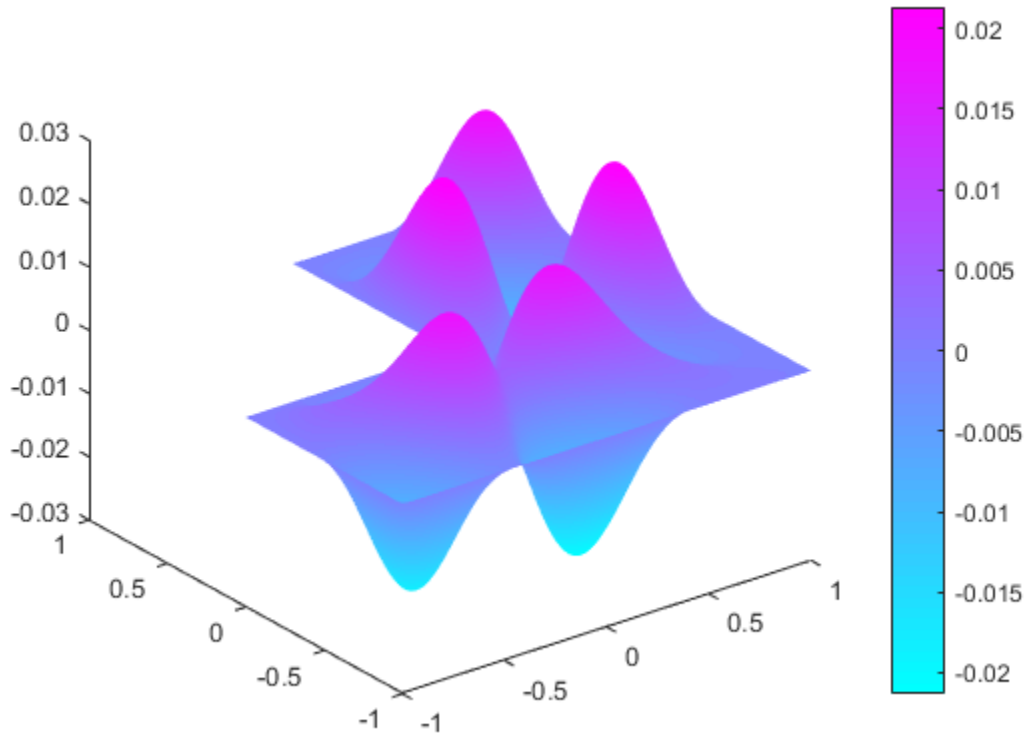
**Eigenvalues and Eigenvectors of the L-Shaped Membrane Using Legacy Syntax**

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".

```
g = @lshapeg;
pdegplot(g,'EdgeLabels','on')
axis equal
ylim([-1.1,1.1])
```
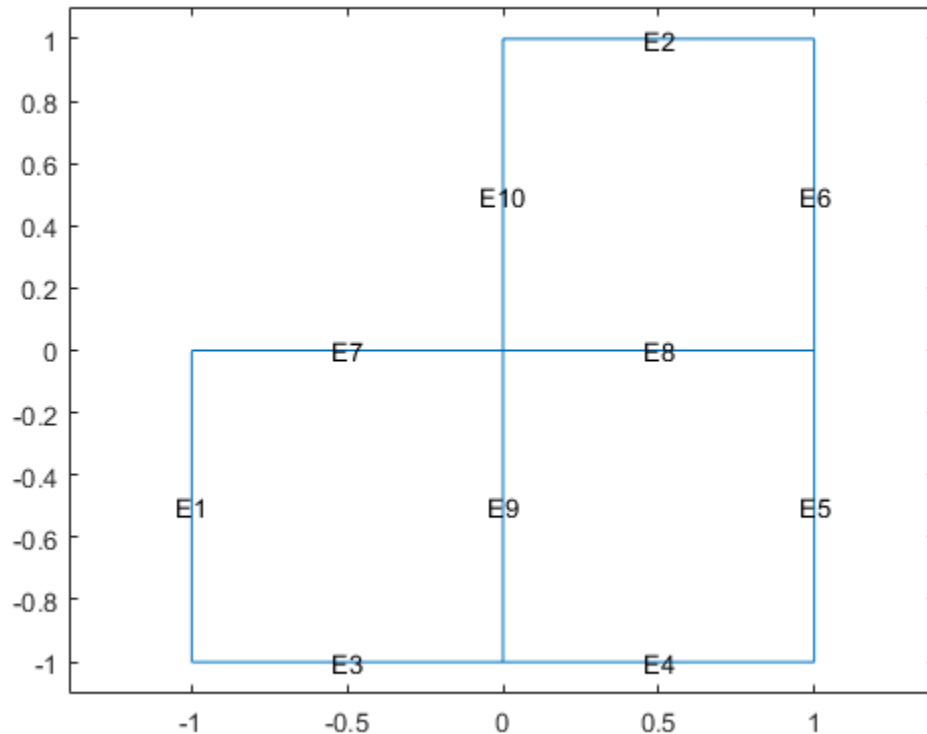
Set zero Dirichlet boundary conditions using the `lshapeb` function. For more information about this syntax, see "Boundary Conditions by Writing Functions".

```
b = @lshapeb;
```

Set coefficients `c = 1`, `a = 0`, and `d = 1`. Collect eigenvalues up to 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);
```

```
Basis= 10,  Time=   1.67,  New conv eig=  0
Basis= 11,  Time=   1.79,  New conv eig=  0
Basis= 12,  Time=   1.79,  New conv eig=  0
Basis= 13,  Time=   1.86,  New conv eig=  0
Basis= 14,  Time=   2.03,  New conv eig=  0
Basis= 15,  Time=   2.03,  New conv eig=  1
Basis= 16,  Time=   2.03,  New conv eig=  1
Basis= 17,  Time=   2.18,  New conv eig=  3
Basis= 18,  Time=   2.18,  New conv eig=  4
Basis= 19,  Time=   2.20,  New conv eig=  4
Basis= 20,  Time=   2.20,  New conv eig=  4
Basis= 21,  Time=   2.37,  New conv eig=  4
Basis= 22,  Time=   2.37,  New conv eig=  4
Basis= 23,  Time=   2.40,  New conv eig=  4
Basis= 24,  Time=   2.40,  New conv eig=  5
Basis= 25,  Time=   2.57,  New conv eig=  5
Basis= 26,  Time=   2.57,  New conv eig=  5
Basis= 27,  Time=   2.57,  New conv eig=  6
Basis= 28,  Time=   2.59,  New conv eig=  7
Basis= 29,  Time=   2.59,  New conv eig=  7
Basis= 30,  Time=   2.76,  New conv eig=  7
Basis= 31,  Time=   2.76,  New conv eig=  7
Basis= 32,  Time=   2.76,  New conv eig=  8
Basis= 33,  Time=   2.76,  New conv eig=  8
Basis= 34,  Time=   2.78,  New conv eig=  8
Basis= 35,  Time=   2.78,  New conv eig=  9
Basis= 36,  Time=   2.90,  New conv eig=  9
Basis= 37,  Time=   2.90,  New conv eig=  9
Basis= 38,  Time=   2.98,  New conv eig=  9
Basis= 39,  Time=   3.04,  New conv eig=  9
Basis= 40,  Time=   3.17,  New conv eig=  9
Basis= 41,  Time=   3.17,  New conv eig=  9
Basis= 42,  Time=   3.18,  New conv eig= 10
Basis= 43,  Time=   3.18,  New conv eig= 11
Basis= 44,  Time=   3.20,  New conv eig= 12
Basis= 45,  Time=   3.39,  New conv eig= 12
Basis= 46,  Time=   3.39,  New conv eig= 14
Basis= 47,  Time=   3.53,  New conv eig= 15
Basis= 48,  Time=   3.71,  New conv eig= 16
Basis= 49,  Time=   3.71,  New conv eig= 17
Basis= 50,  Time=   3.90,  New conv eig= 17
Basis= 51,  Time=   3.90,  New conv eig= 18
```

```
                  Basis= 52,  Time=   3.92,  New conv eig= 18
                  Basis= 53,  Time=   3.92,  New conv eig= 19
                  Basis= 54,  Time=   3.93,  New conv eig= 19
                  Basis= 55,  Time=   3.93,  New conv eig= 22
                  Basis= 56,  Time=   3.95,  New conv eig= 24
                  Basis= 57,  Time=   4.06,  New conv eig= 28
End of sweep: Basis= 57,  Time=   4.06,  New conv eig= 28
                  Basis= 38,  Time=   4.45,  New conv eig=  0
                  Basis= 39,  Time=   4.59,  New conv eig=  0
                  Basis= 40,  Time=   4.59,  New conv eig=  0
                  Basis= 41,  Time=   4.60,  New conv eig=  0
                  Basis= 42,  Time=   4.68,  New conv eig=  0
End of sweep: Basis= 42,  Time=   4.68,  New conv eig=  0
```

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');
figure
pdegplot(model,'FaceLabels','on')
view(30,30)
title('Bracket with Face Labels')
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```

**Bracket with Face Labels**
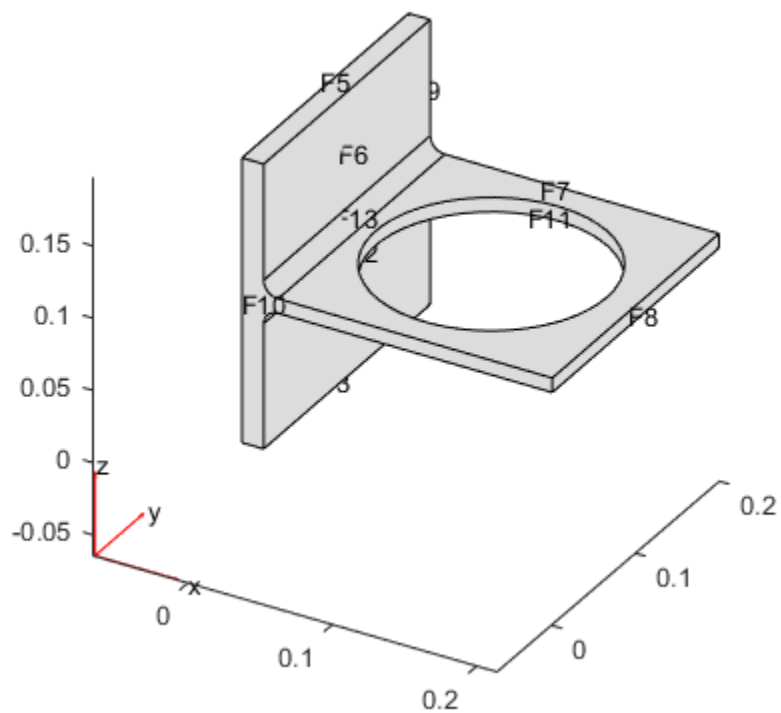
**Bracket with Face Labels, Rear View**
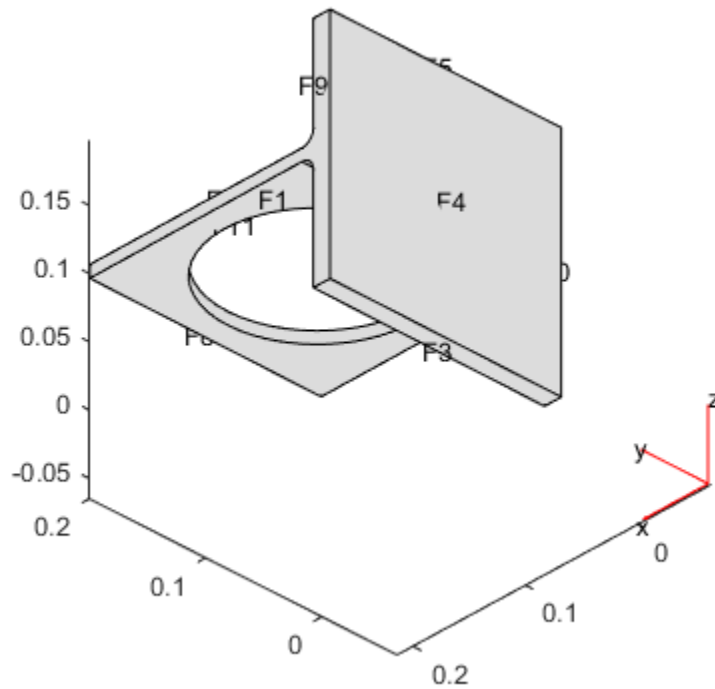
Set coefficients `c = 1`, `a = 0`, and `d = 1`. Collect eigenvalues that are less than 100.

```
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);
```

```
                Basis= 10,  Time=   4.04,  New conv eig=  0
                Basis= 11,  Time=   4.13,  New conv eig=  0
                Basis= 12,  Time=   4.24,  New conv eig=  0
                Basis= 13,  Time=   4.40,  New conv eig=  1
                Basis= 14,  Time=   4.49,  New conv eig=  2
                Basis= 15,  Time=   4.54,  New conv eig=  2
                Basis= 16,  Time=   4.63,  New conv eig=  3
End of sweep: Basis= 16,  Time=   4.63,  New conv eig=  3
                Basis= 13,  Time=   5.51,  New conv eig=  0
End of sweep: Basis= 13,  Time=   5.51,  New conv eig=  0
```

Look at the first two eigenvalues.

```
l([1,2])
```

```
ans =

    0.0000
   42.8170
```

Plot the solution corresponding to eigenvalue 2.

```
pdeplot3D(model,'ColorMapData',v(:,2))
```
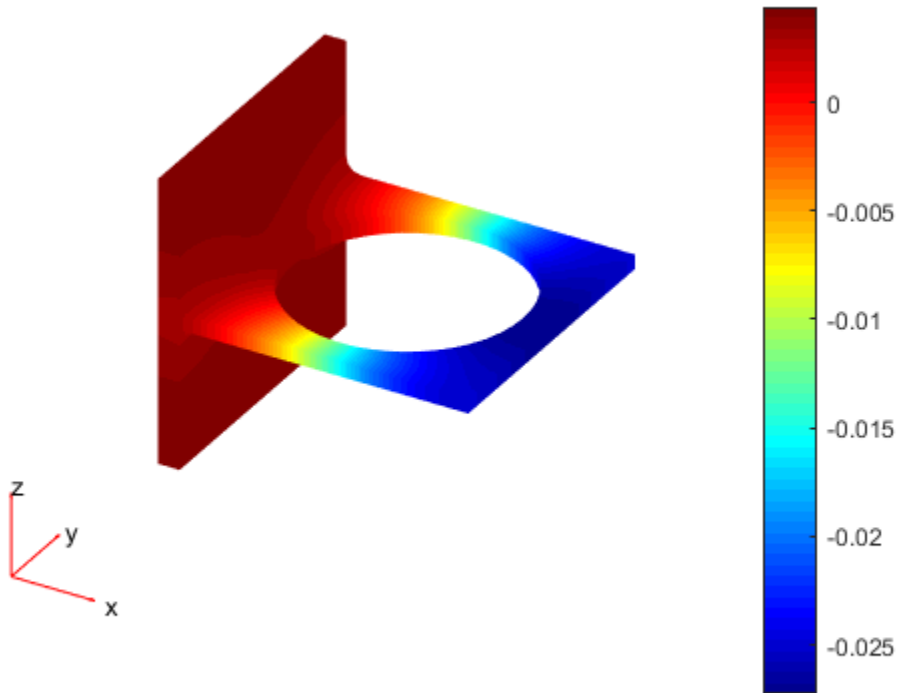
## Input Arguments

**`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 2-110. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 $\lambda$ 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 file name.

- 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 2-154.

- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-183.

Example: `b = 'circleb1'` or equivalently `b = @circleb1`

Data Types: `double | char | function_handle`

### p — Mesh points
matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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

Mesh triangles, specified as a 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 2-196.

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

### 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 6-62. 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,O,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.

## Tips

- The equation coefficients cannot depend on the solution u or its gradient.

## See Also

### See Also
solvepdeeig

### Topics
"Eigenvalue Equations" on page 5-22

**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`

**Introduced before R2006a**

# 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`

**Introduced before R2006a**

# pdeGeometryFromEdges

(To be removed) Create geometry object

## Compatibility

`pdeGeometryFromEdges` has been removed. 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 2-16)

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

## 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)
pdegplot(g,Name,Value)
h = pdegplot( ___ )
```

## Description

pdegplot(g) plots the geometry of a PDE problem, as described in g.

pdegplot(g,Name,Value) plots with additional options specified by one or more Name,Value pair arguments.

h = pdegplot( ___ ) returns handles to the graphics, using any of the previous syntaxes.

## Examples

### Plot 2-D Geometry with and Without Labels

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 -0.6   -0.5  -0.4   -0.5   0.4    0.5   0.6    0.5  -1      0.17;
     1 -0.5   -0.4  -0.5   -0.6   0.5    0.6   0.5    0.4   0.17   1;
     0 -0.25  -0.35 -0.25  -0.15 -0.25  -0.35 -0.25  -0.15  0     -0.74;
     0 -0.35  -0.25 -0.15  -0.25 -0.35  -0.25 -0.15  -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  0.1     0.1   0.1    0.1   0.1    0.1   0.1    0.1   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)
```
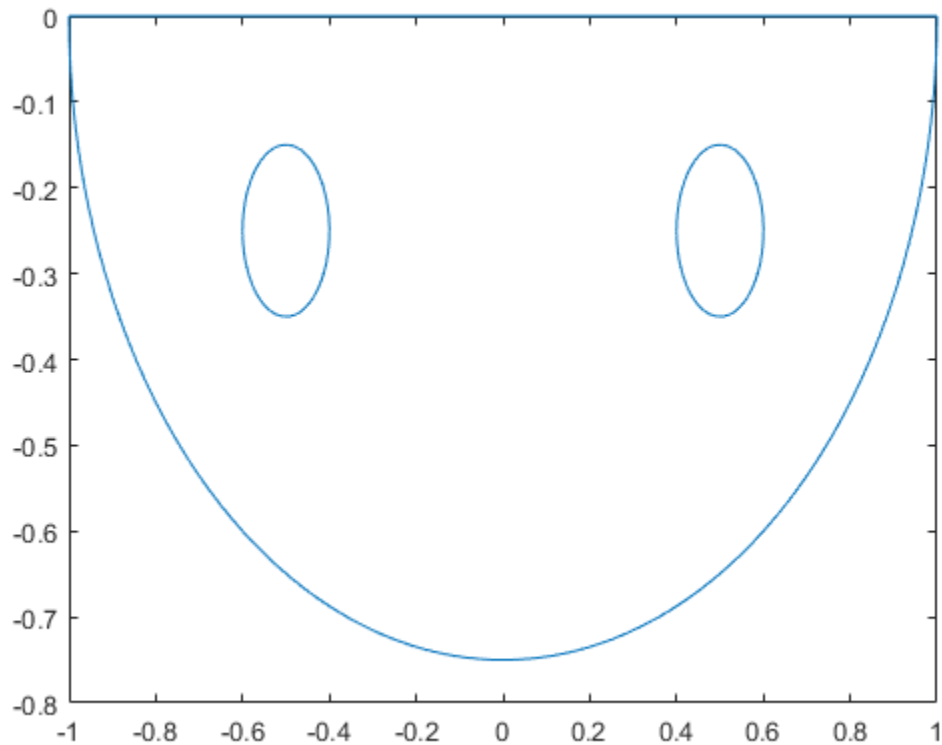


View the vertex labels, edge labels, and the face label. Add space at the top of the plot to see the top edge clearly.

```
pdegplot(g,'VertexLabels','on','EdgeLabels','on','FaceLabels','on')
ylim([-.8,.1])
```
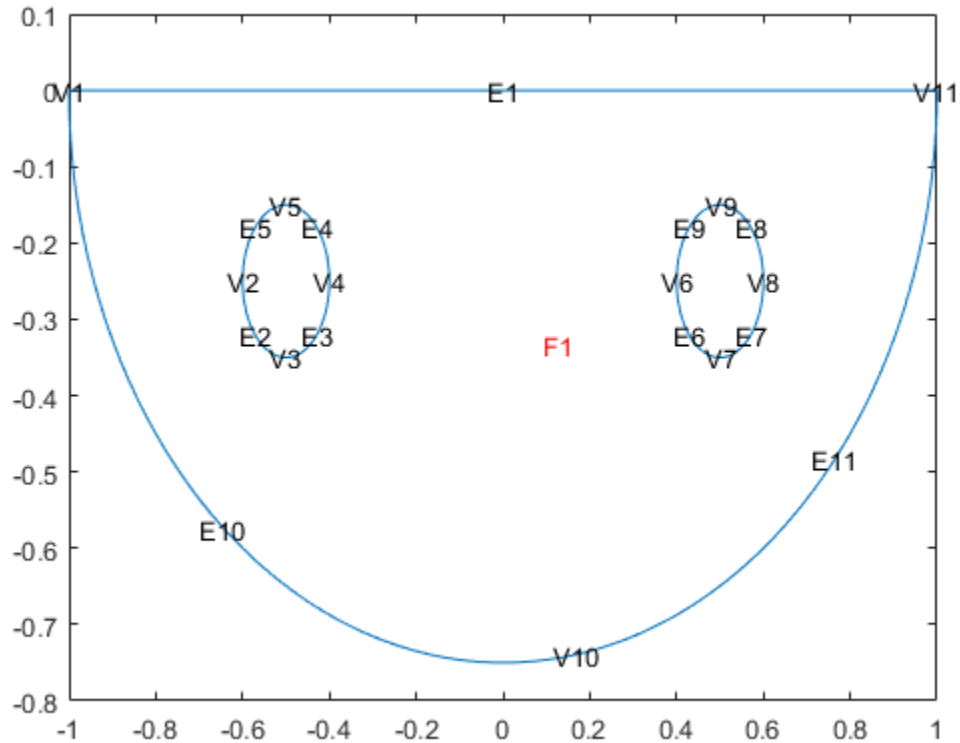
**Plot 3-D Geometry**

Import a 3-D geometry file. Plot the geometry and turn on face labels. To see the labels on all faces of the geometry, set the transparency to 0.5.

```
model = createpde;
importGeometry(model,'BracketWithHole.stl');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
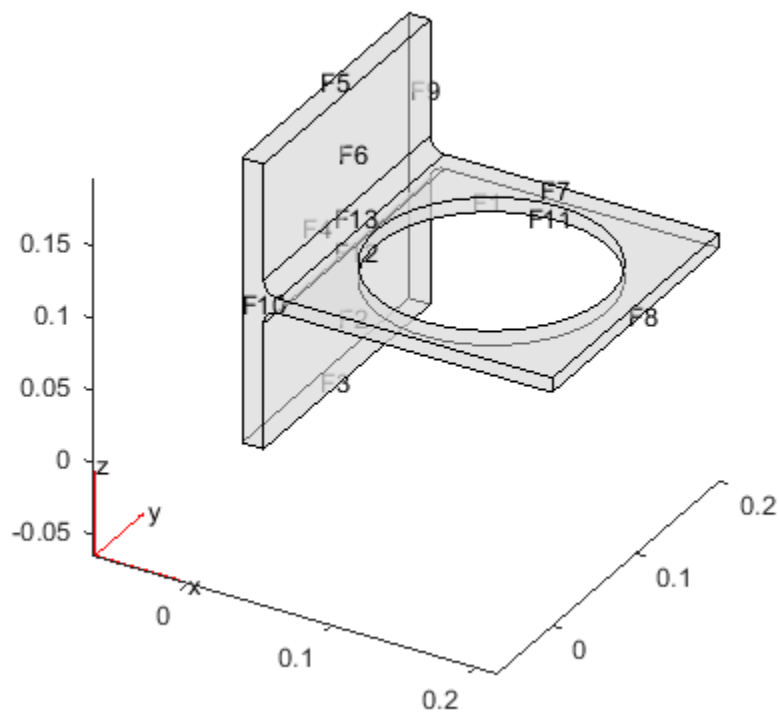
### Plot Multi-Cellular 3-D Geometry

Import a 3-D geometry file. Plot the geometry and turn on cell labels.

```
model = createpde;
importGeometry(model,'DampingMounts.stl');
pdegplot(model,'CellLabels','on')
```
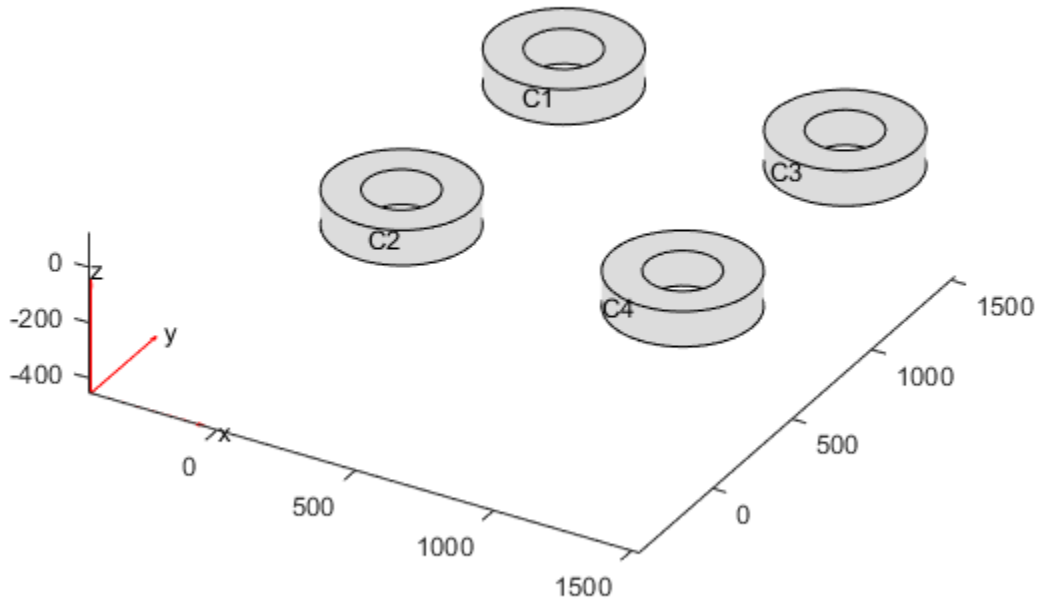
- "Solve PDE with Coefficients in Functional Form" on page 2-66
- "Create Geometry and Remove Face Boundaries" on page 2-12
- "Create and View 3-D Geometry" on page 2-36

## 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 2-14)
- Name of geometry file (see "Create Geometry Using a Geometry Function" on page 2-16)
- Function handle to geometry file (see "Create Geometry Using a Geometry Function" on page 2-16)

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.

The argument SubdomainLabels is not recommended. Use FaceLabels for both 2-D and 3-D geometries instead.

Example: pdegplot(g,'FaceLabels','on')

### 'VertexLabels' — Vertex labels for 2-D or 3-D geometry
'off' (default) | 'on'

Vertex labels for 2-D or 3-D geometry, specified as 'off' or 'on'.

Example: 'VertexLabels','on'

Data Types: char

### 'EdgeLabels' — Boundary edge labels for 2-D or 3-D geometry
'off' (default) | 'on'

Boundary edge labels for 2-D or 3-D geometry, specified as 'off' or 'on'.

Example: 'EdgeLabels','on'

Data Types: char

### 'FaceLabels' — Boundary face labels for 2-D or 3-D geometry
'off' (default) | 'on'

Boundary face labels for 2-D or 3-D geometry, specified as `'off'` or `'on'`.

Example: `'FaceLabels','on'`

Data Types: `char`

### `'CellLabels'` — Cell labels for 3-D geometry
`'off'` (default) | `'on'`

Cell labels for 3-D geometry, specified as `'off'` or `'on'`.

Example: `'CellLabels','on'`

Data Types: `char`

### `'FaceAlpha'` — Surface transparency for 3-D geometry
1 (default) | real number from 0 through 1

Surface transparency for 3-D geometry, specified as a real number from 0 through 1. The default value 1 indicates no transparency. The value 0 indicates complete transparency.

Example: `'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.

## 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 face labels, select **PDE > Show Subdomain Labels**.

# See Also

## See Also
decsg | importGeometry | pdetool | wgeom

## Topics
"Solve PDE with Coefficients in Functional Form" on page 2-66
"Create Geometry and Remove Face Boundaries" on page 2-12
"Create and View 3-D Geometry" on page 2-36
"Create 2-D Geometry" on page 2-7
"Solve Problems Using PDEModel Objects" on page 2-5

**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.

Although `pdegrad` returns the value of the gradient at the center of a triangle, the gradient is actually the same everywhere in the triangle interior. This is because

pdegrad uses only linear basis functions. The boundaries of triangles are a special case: here the derivatives might be discontinuous.

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

**Introduced before R2006a**

# pdeInterpolant

Interpolant for nodal data to selected locations

## Compatibility

A `pdeInterpolant` 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.

## Create Object

```
F = pdeInterpolant(p,t,u)
```

## Description

`F = pdeInterpolant(p,t,u)` returns an interpolant `F` based on the data points `p`, elements `t`, and data values at the points, `u`.

---

**Tip:** Use `meshToPet` to obtain the `p` and `t` data for interpolation using `pdeInterpolant`.

---

## Input Arguments

**p — Data point locations**
matrix with two or three rows

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 2-196.

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).

### t — Triangulation elements
matrix

Triangulation elements, specified as a matrix. For details, see "Mesh Data" on page 2-196.

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
vector | matrix

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.

## Object Functions

evaluate                                        Interpolate data to selected locations

# 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.

## See Also

### See Also
evaluate | tri2grid

### Topics
"Mesh Data" on page 2-196

**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

**Introduced before R2006a**

# 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. `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

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

where $\mathbf{n}_\tau$ is the unit normal of edge $\tau$ and the braced term is the jump in flux across the element edge, where $\alpha$ and $\beta$ 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 2-199.

**Introduced before R2006a**

# pdemdlcv

(Removed) Convert Partial Differential Equation Toolbox 1.0 model files to 1.0.2 format

## Compatibility

pdemdlcv has been removed.

## 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.

**Introduced before R2006a**

# pdemesh

Plot PDE mesh

## Compatibility

Meshing is the same for both the recommended and the legacy workflows.

## Syntax

```
pdemesh(model)
pdemesh(model,u)
pdemesh( ___ ,Name,Value)

pdemesh(p,e,t)
pdemesh(p,e,t,u)
h = pdemesh( ___ )
```

## Description

pdemesh(model) plots the mesh contained in a 2-D or 3-D model object of type
PDEModel.

pdemesh(model,u) plots solution data u as a 3-D plot. This syntax is valid only for 2-D
geometry.

pdemesh( ___ ,Name,Value) plots the mesh or solution data using any of the
arguments in the previous syntaxes and one or more Name,Value pair arguments.

pdemesh(p,e,t) plots the mesh specified by the mesh data p,e,t.

pdemesh(p,e,t,u) plots PDE node or triangle data u using a mesh plot. The function
plots the node data if u is a column vector , and triangle data if u is a row vector.

If you want to have more control over your mesh plot, use pdeplot or pdeplot3D
instead of pdemesh.

`h = pdemesh( ___ )` returns handles to the graphics, using any of the arguments of the previous syntaxes.

# Examples

### Mesh Plot for L-Shaped Membrane

Create a mesh plot and display the node and element labels of the mesh.

Create a PDE model. Include the geometry of the built-in function `lshapeg`. Mesh the geometry.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
generateMesh(model);
```

Plot the mesh.

```
pdemesh(model)
```
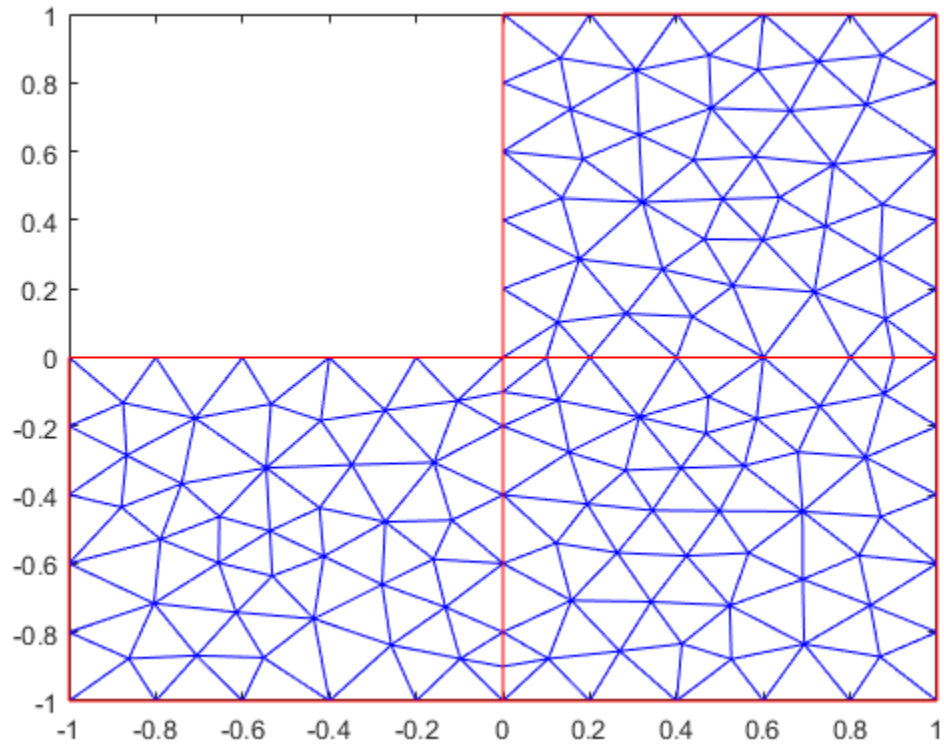
Display node labels.

```
pdemesh(model,'NodeLabels','on')
```
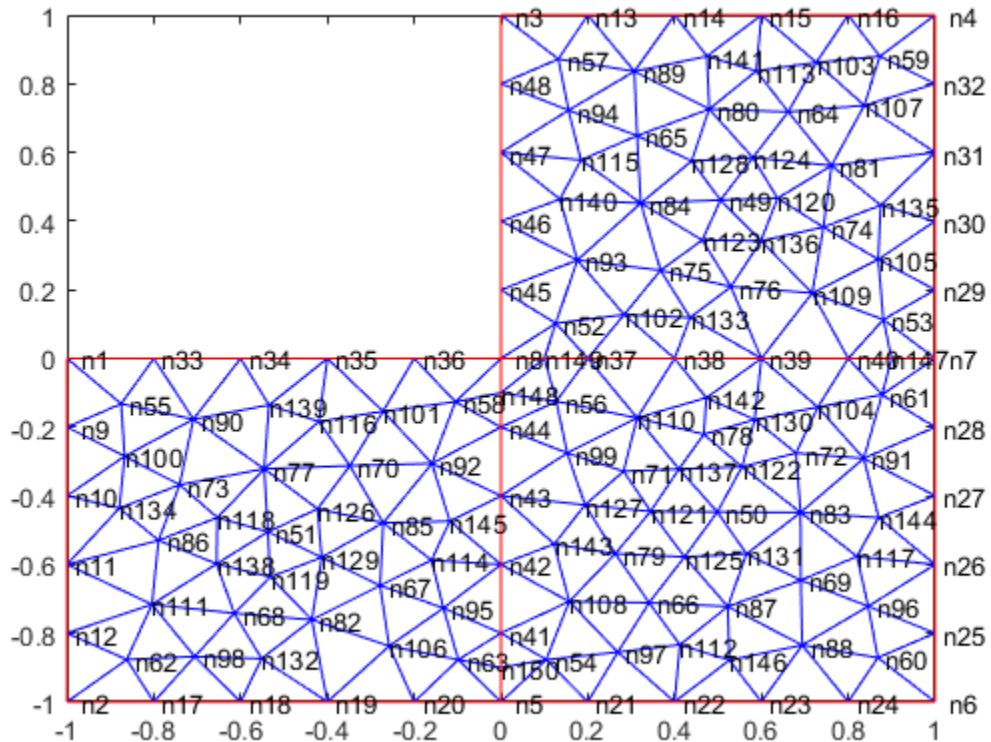
Use `xlim` and `ylim` to zoom in on particular nodes.

```
xlim([-0.4,0.4])
ylim([-0.4,0.4])
```
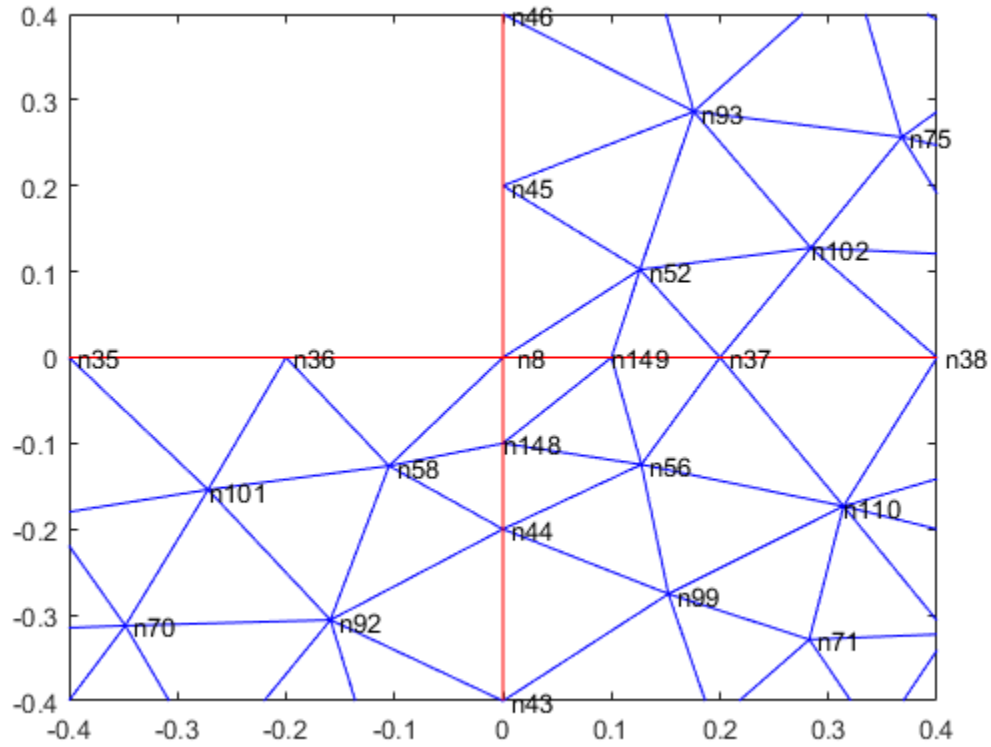
Display element labels.

```
pdemesh(model,'ElementLabels','on')
xlim([-0.4,0.4])
ylim([-0.4,0.4])
```
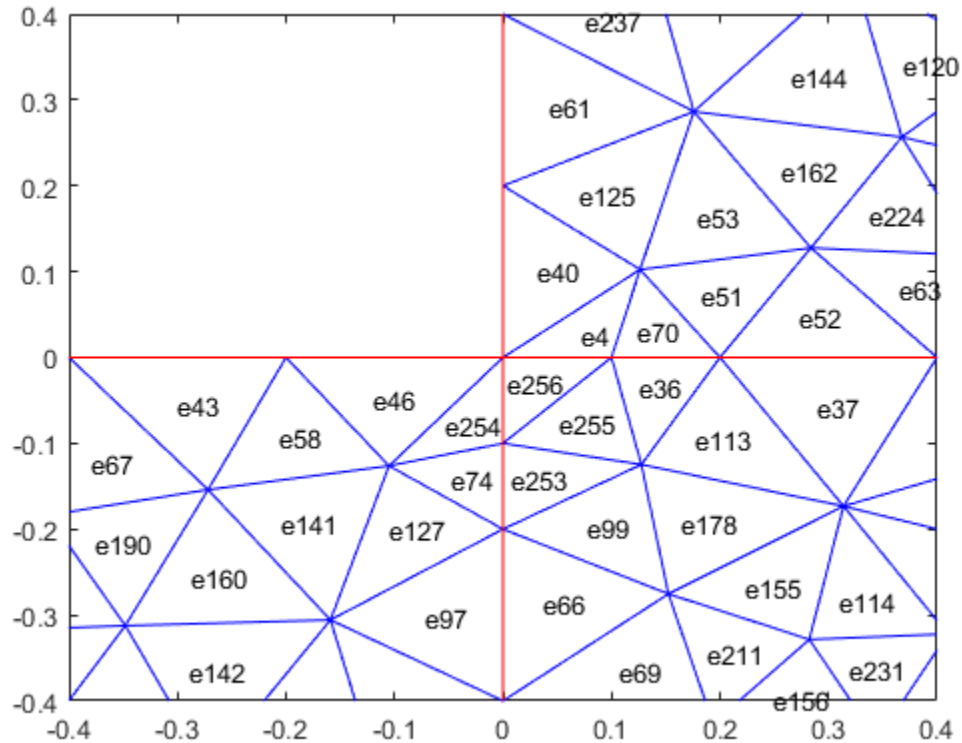
Apply boundary conditions, specify coefficients, and solve the PDE.

```
applyBoundaryCondition(model,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
specifyCoefficients(model,'m',0,...
                          'd',0,...
                          'c',1,...
                          'a',0,...
                          'f',1);
generateMesh(model);
results = solvepde(model)
u = results.NodalSolution;


results =
```

```
StationaryResults with properties:

    NodalSolution: [150×1 double]
       XGradients: [150×1 double]
       YGradients: [150×1 double]
       ZGradients: []
            Mesh: [1×1 FEMesh]
```

Plot the solution at nodal locations by using `pdemesh`.

```
pdemesh(model,u)
```
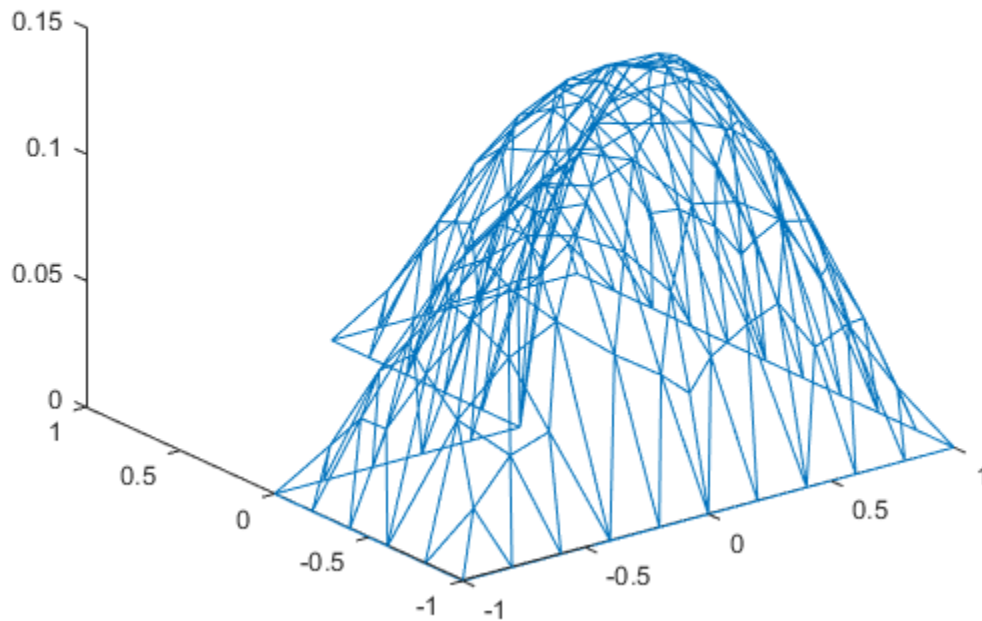
The `pdemesh` function ignores `NodeLabels` and `ElementLabels` when you plot solution data as a 3-D plot.

### Transparency for 3-D Mesh

Create a PDE model, include the geometry and mesh it.

```
model = createpde;
importGeometry(model,'Plate10x10x1.stl');
generateMesh(model,'Hmax',5);
```

Plot the mesh setting the transparency to 0.5.

```
pdemesh(model,'FaceAlpha',0.5)
```
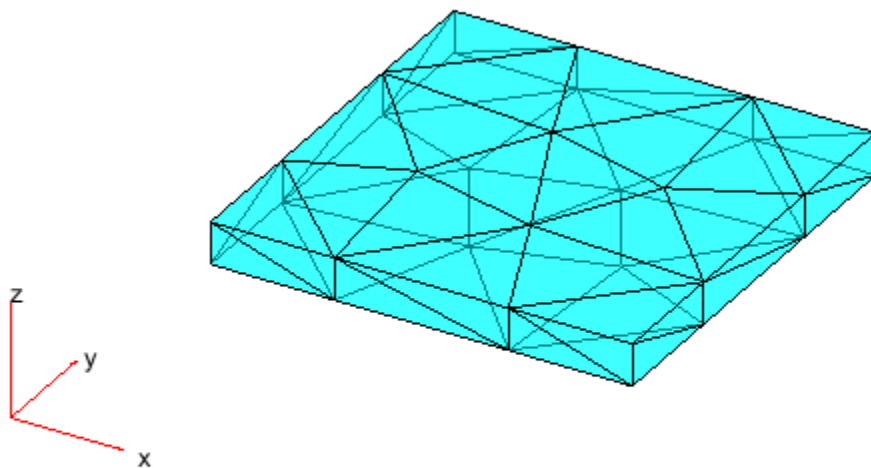
### [p,e,t] Mesh Plot

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)
```
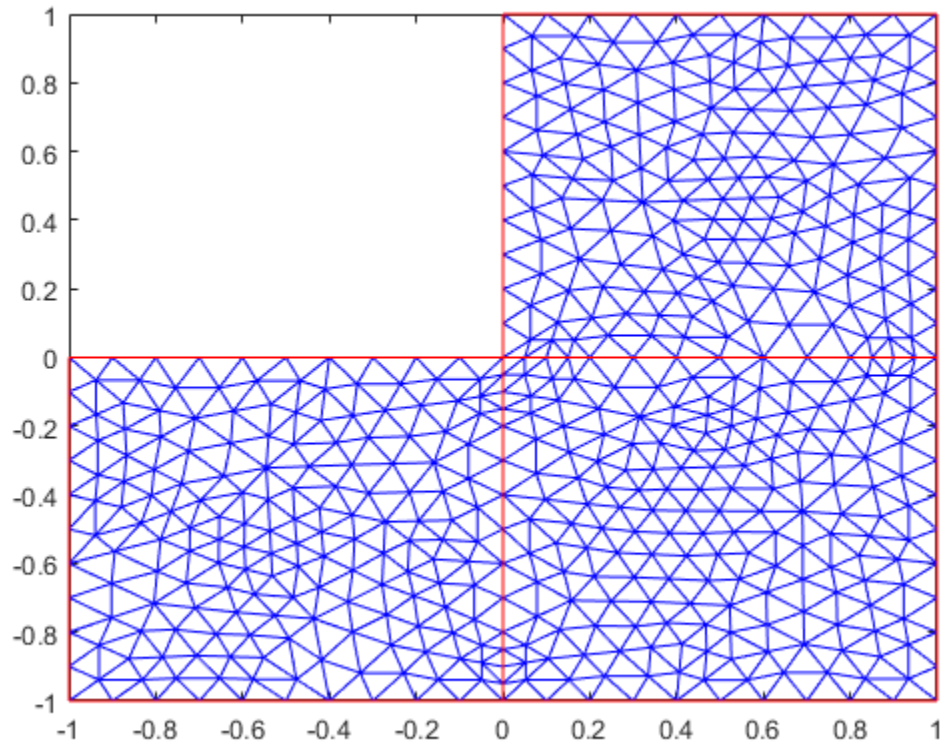
Now solve Poisson's equation $-\Delta u = 1$ over the geometry defined by the L-shaped membrane. Use Dirichlet boundary conditions $u = 0$ on $\delta\Omega$, and plot the result.

```
u = assempde('lshapeb',p,e,t,1,0,1);
pdemesh(p,e,t,u)
```
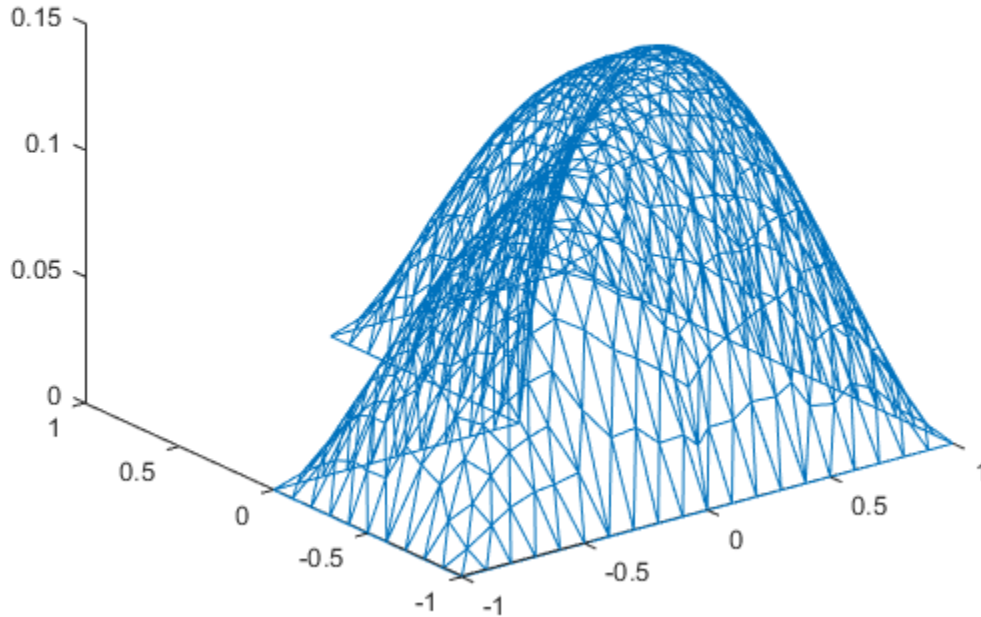
**6-497**

## 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: `results = solvepde(model); u = results.NodalSolution;` or `u = assempde(model,c,a,f);`

### p — Mesh points
matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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

Mesh triangles, specified as a 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 2-196.

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: `pdemesh(model,'NodeLabels','on')`

### `'NodeLabels'` — Node labels
`'off'` (default) | `'on'`

Node labels, specified as `'off'` or `'on'`. Note that

- `pdeplot` ignores `NodeLabels` when you use it with `ZData`.
- `pdemesh` ignores `NodeLabels` when you plot solution data as a 3-D plot.

Example: `'NodeLabels','on'`

Data Types: `char`

### `'ElementLabels'` — Element labels
`'off'` (default) | `'on'`

Element labels, specified as `'off'` or `'on'`. Note that

- `pdeplot` ignores `ElementLabels` when you use it with `ZData`.
- `pdemesh` ignores `ElementLabels` when you plot solution data as a 3-D plot.

Example: `'ElementLabels','on'`

Data Types: `char`

### `'FaceAlpha'` — Surface transparency for 3-D geometry
`1` (default) | real number from `0` through `1`

Surface transparency for 3-D geometry, specified as a real number from `0` through `1`. The default value `1` indicates no transparency. The value `0` indicates complete transparency.

Example: `'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.

## See Also

### See Also
pdegplot | pdeplot | pdeplot3d | pdesurf

### Topics
"Mesh Data" on page 2-196

**Introduced before R2006a**

# PDEModel

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.

## Create Object

`createpde` returns a `PDEModel` container. Initially, the only property that is nonempty is `PDESystemSize`, which is `1` for scalar problems.

## Properties

### `PDESystemSize` — Number of equations
1 (default) | positive integer

Number of equations, *N*, returned as a positive integer. See "Equations You Can Solve Using Recommended Functions" on page 1-6.

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 | `NodalInitialConditions` object

Initial conditions or initial solution, returned as a `GeometricInitialConditions` or `NodalInitialConditions` object.

In case of `GeometricInitialConditions`, for time-dependent problems, you must give one or two 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`.

In case of `NodalInitialConditions`, you use the results of previous analysis to set the initial conditions or initial guess. The geometry and mesh of the previous analysis and current model must be the same.

**`SolverOptions` — Algorithm options for PDE solvers**
PDESolverOptions object

Algorithm options for the PDE solvers, returned as a `PDESolverOptions` object. The properties of `PDESolverOptions` include absolute and relative tolerances for internal ODE solvers, maximum solver iterations, and so on.

# 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 |

# Examples

### Create and Populate a PDE Model

Create and populate a `PDEModel` object.

Create a container for a scalar PDE (N = 1).

```
model = createpde()


model =

  PDEModel with properties:

          PDESystemSize: 1
        IsTimeDependent: O
               Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                   Mesh: []
```

```
        SolverOptions: [1×1 PDESolverOptions]
```

Include a torus geometry, zero Dirichlet boundary conditions, coefficients for Poisson's equation, and the default mesh.

```
importGeometry(model,'Torus.stl');
applyBoundaryCondition(model,'dirichlet','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: [30656×1 double]
       XGradients: [30656×1 double]
       YGradients: [30656×1 double]
       ZGradients: [30656×1 double]
             Mesh: [1×1 FEMesh]
```

• "Solve Problems Using PDEModel Objects" on page 2-5

# See Also

## See Also
applyBoundaryCondition | createpde | generateMesh | geometryFromEdges | geometryFromMesh | importGeometry | pdegplot | pdeplot | pdeplot3D | setInitialConditions | specifyCoefficients

## Topics
"Solve Problems Using PDEModel Objects" on page 2-5

**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)
```
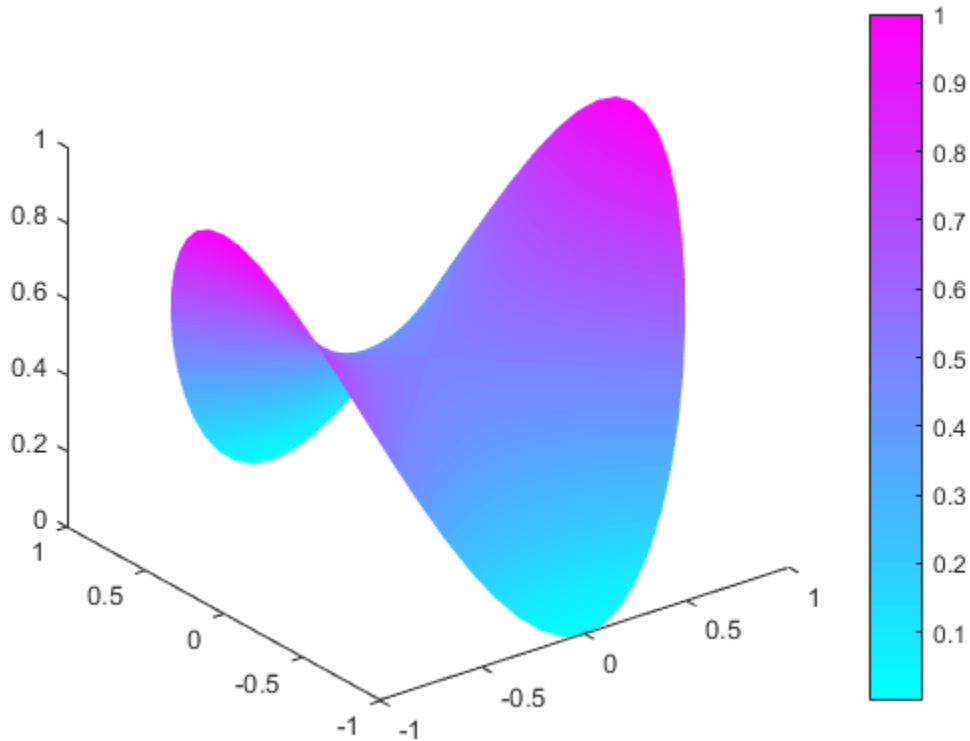
### Minimal Surface Problem Using [p,e,t] Mesh

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
```
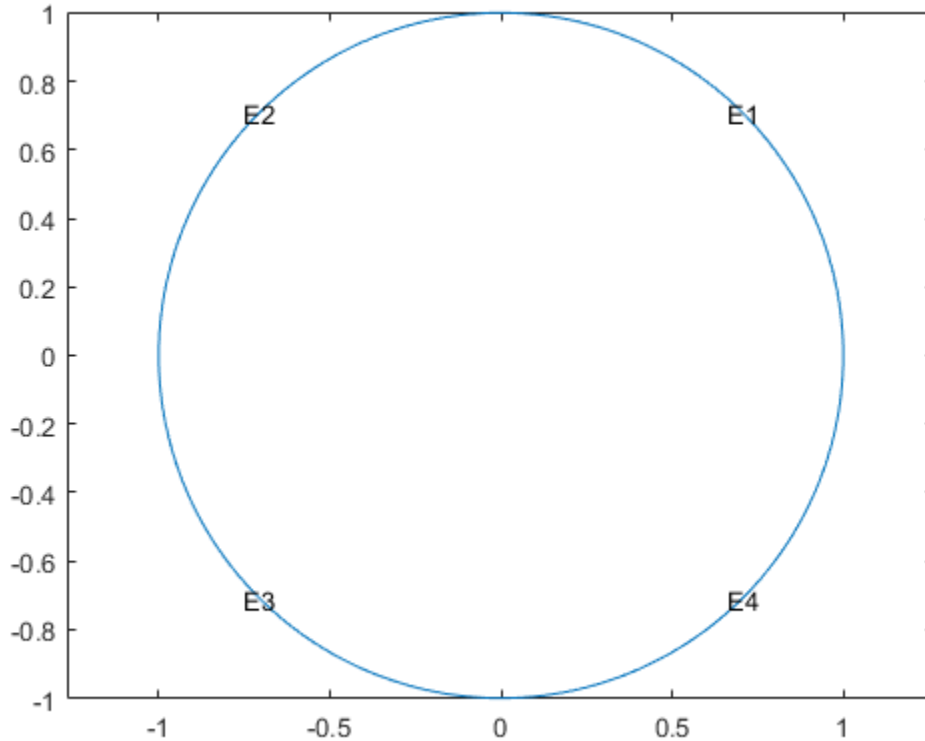
Create Dirichlet boundary conditions with value $x^2$. Create the following file and save it on your Matlab™ path. For details of this approach, see "Boundary Conditions by Writing Functions".

```
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)
```
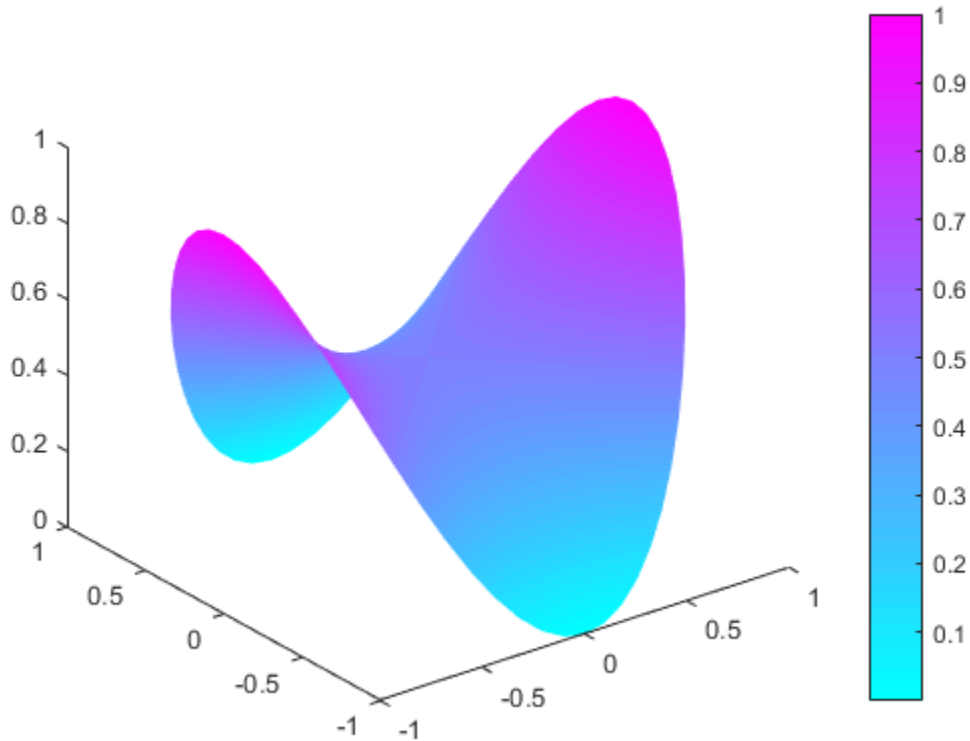
### Nonlinear Problem with 3-D Geometry

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');
figure
pdegplot(model,'FaceLabels','on')
view(30,30)
title('Bracket with Face Labels')
figure
```

```
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```



**Bracket with Face Labels**
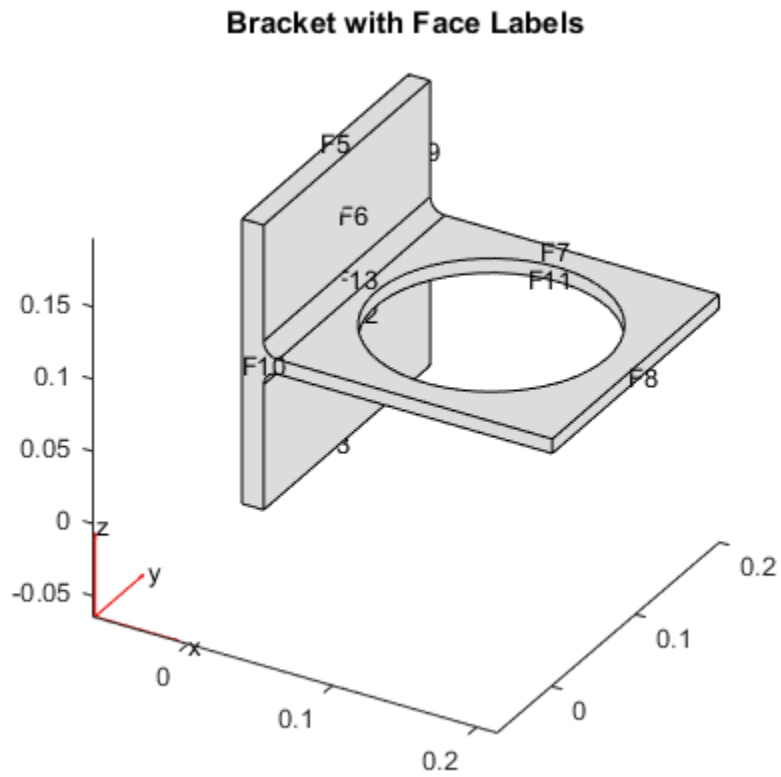
**Bracket with Face Labels, Rear View**
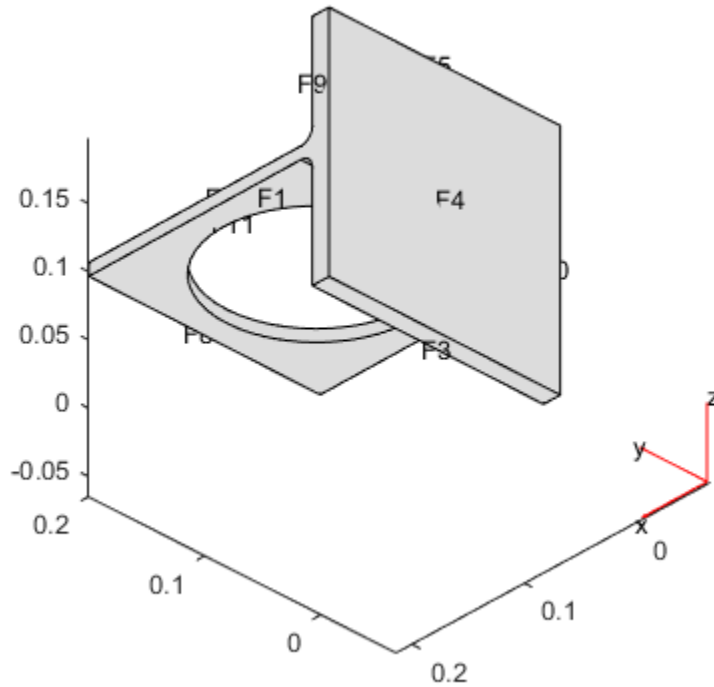


Set a Dirichlet boundary condition with value 1000 on the back face, which is face 4. Set the large faces 1 and 7, and also the circular face 11, to have Neumann boundary conditions with value g = -10. Do not set boundary conditions on the other faces. Those faces default to Neumann boundary conditions with value g = 0.

```
applyBoundaryCondition(model,'Face',4,'u',1000);
applyBoundaryCondition(model,'Face',[1,7,11],'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 4. 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          3.6247e-01
   1          9.9796e-02    1.0000000
   2          2.9715e-02    1.0000000
   3          8.5763e-03    1.0000000
   4          1.8639e-03    1.0000000
   5          1.5245e-04    1.0000000
   6          1.2347e-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 2-110. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-133. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 2-83. See also "Specify Scalar PDE Coefficients in Character Form" on page 2-55, "Specify 2-D Scalar Coefficients in Function Form" on page 2-61, and "Specify 3-D PDE Coefficients in Function Form" on page 2-64.

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 file name.

- 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 2-154.
- A boundary file is a file that you write in the syntax specified in "Boundary Conditions by Writing Functions" on page 2-183.

Example: b = 'circleb1' or equivalently b = @circleb1

Data Types: double | char | function_handle

### p — Mesh points
matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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

Mesh triangles, specified as a 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 2-196.

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:

### `'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 | vector of characters | vector of numbers

Initial solution guess, specified as a scalar, a vector of characters, or a vector of numbers. For details, see "Solve PDEs with Initial Conditions" on page 2-147.

- A scalar specifies a constant initial condition for either a scalar or PDE system.
- For scalar problems, use the same syntax as "Specify Scalar PDE Coefficients in Character Form" on page 2-55.
- For systems of $N$ equations, write a character array with $N$ rows, where each row has the syntax of "Specify Scalar PDE Coefficients in Character Form" on page 2-55.
- 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 2-196.

  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 `'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 3-210.

### 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.

## 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

### See Also
solvepde

**Introduced before R2006a**

# 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(thermalmodel, ___ )

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(thermalmodel, ___ ) plots the mesh specified in thermalmodel or the data (temperature or gradient of temperature) on the thermalmodel mesh. When working with a thermal model, use the same syntaxes as for a generic PDE model.

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, XYData, or ZData 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 graphics, using any of 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)
```
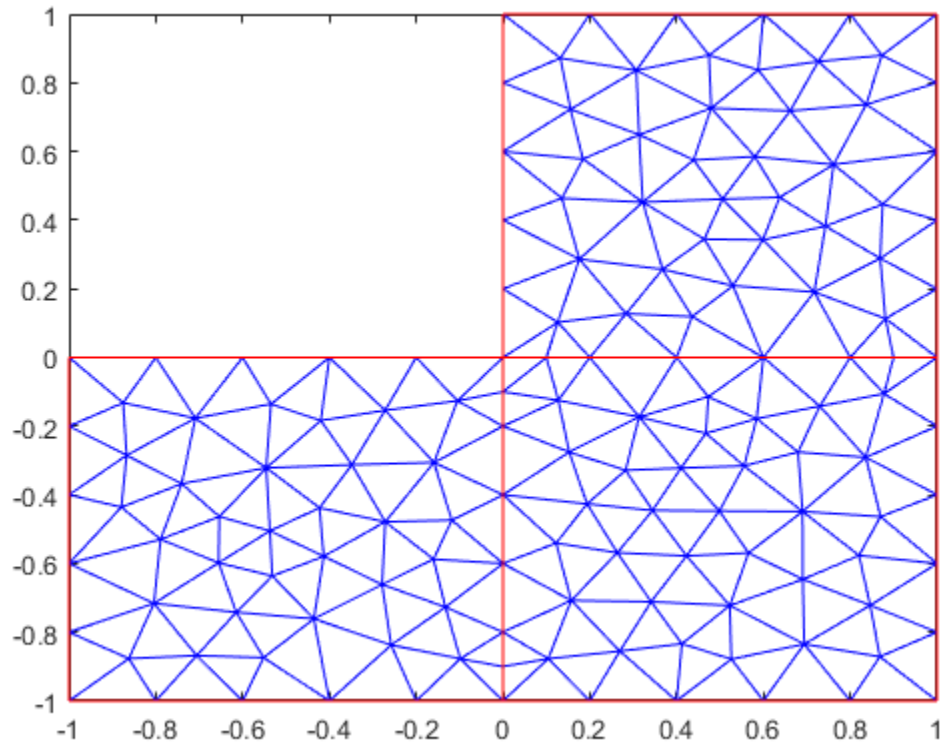
Display the node labels.

```
pdeplot(model,'NodeLabels','on')
```
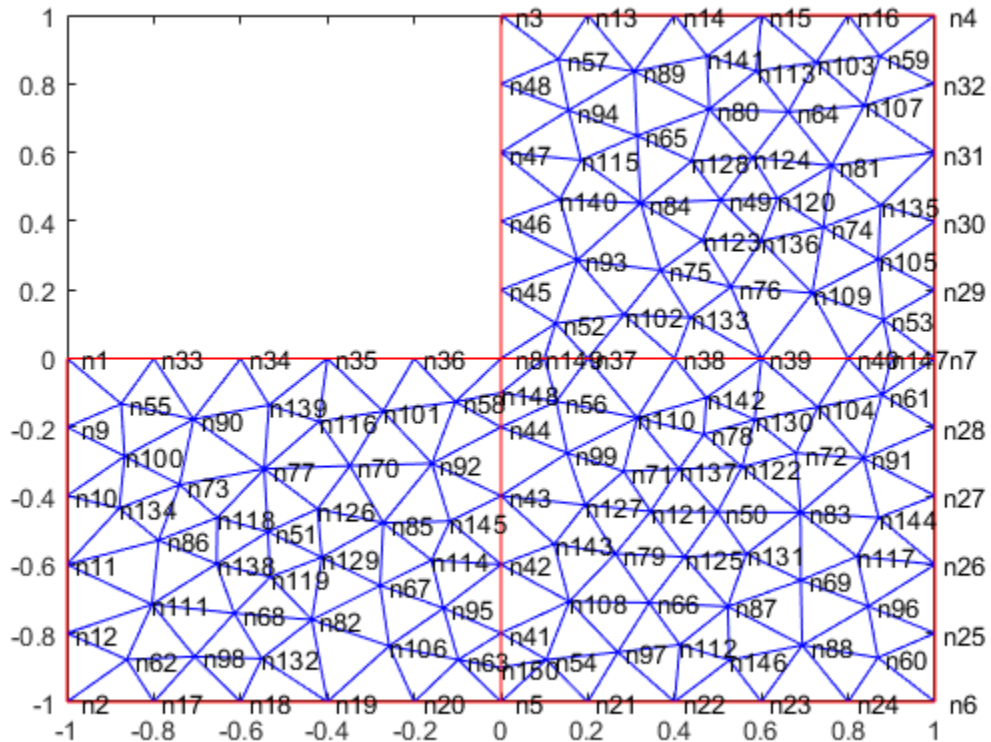
Use xlim and ylim to zoom in on particular nodes.

```
xlim([-0.4,0.4])
ylim([-0.4,0.4])
```

Display the element labels.

```
pdeplot(model,'ElementLabels','on')
xlim([-0.4,0.4])
ylim([-0.4,0.4])
```
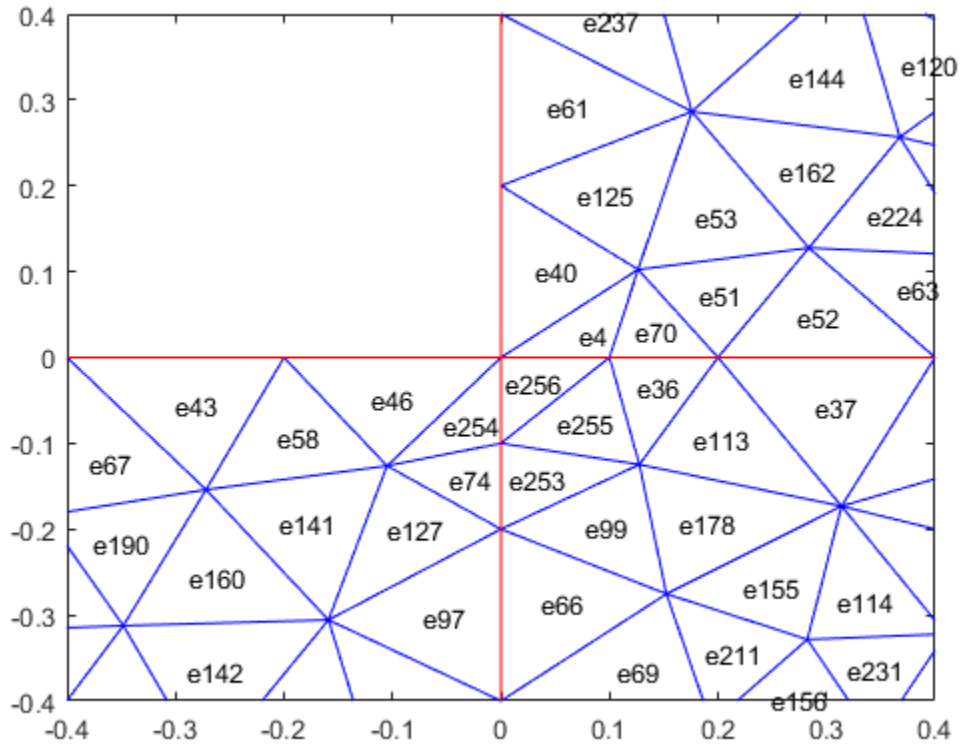
## Solution Plots

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,'dirichlet','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: [150×1 double]
        XGradients: [150×1 double]
        YGradients: [150×1 double]
        ZGradients: []
              Mesh: [1×1 FEMesh]
```

Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

Plot the 2-D solution.

```
pdeplot(model,'XYData',u)
```
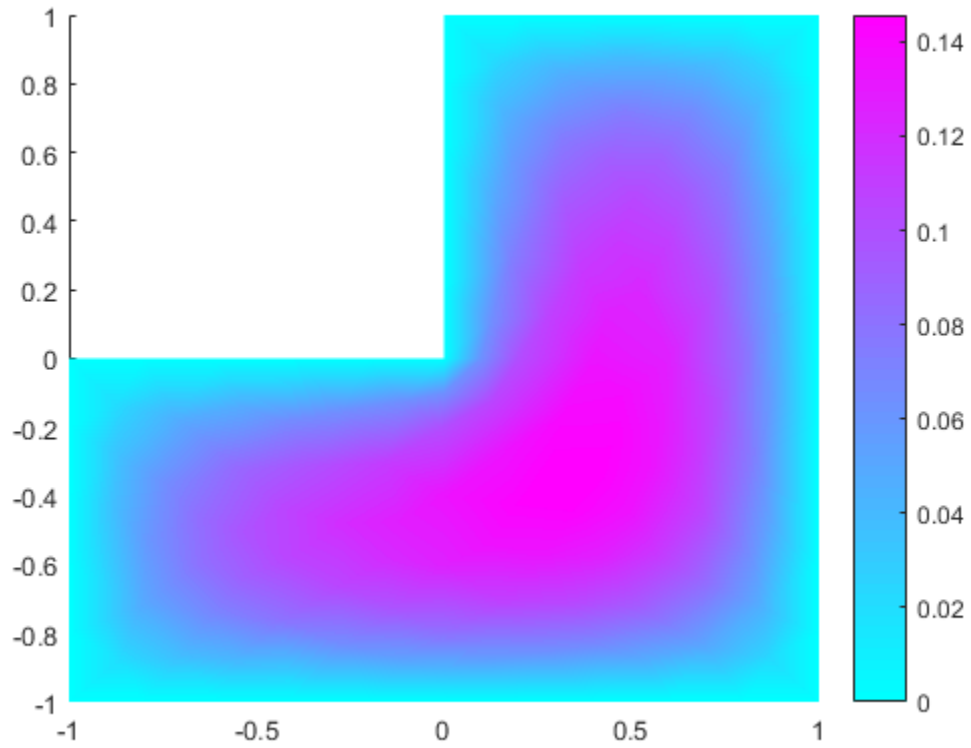
Plot the 3-D solution.

```
pdeplot(model,'XYData',u,'ZData',u)
```
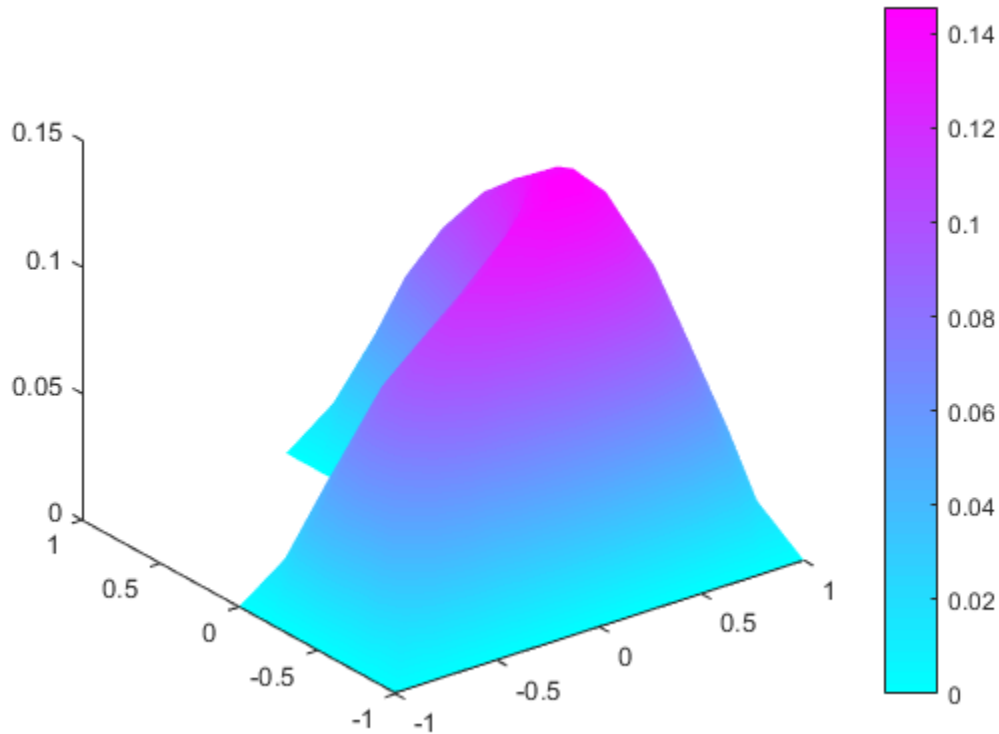
### Solution Quiver Plot

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,'dirichlet','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: [150×1 double]
       XGradients: [150×1 double]
       YGradients: [150×1 double]
       ZGradients: []
             Mesh: [1×1 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])
```
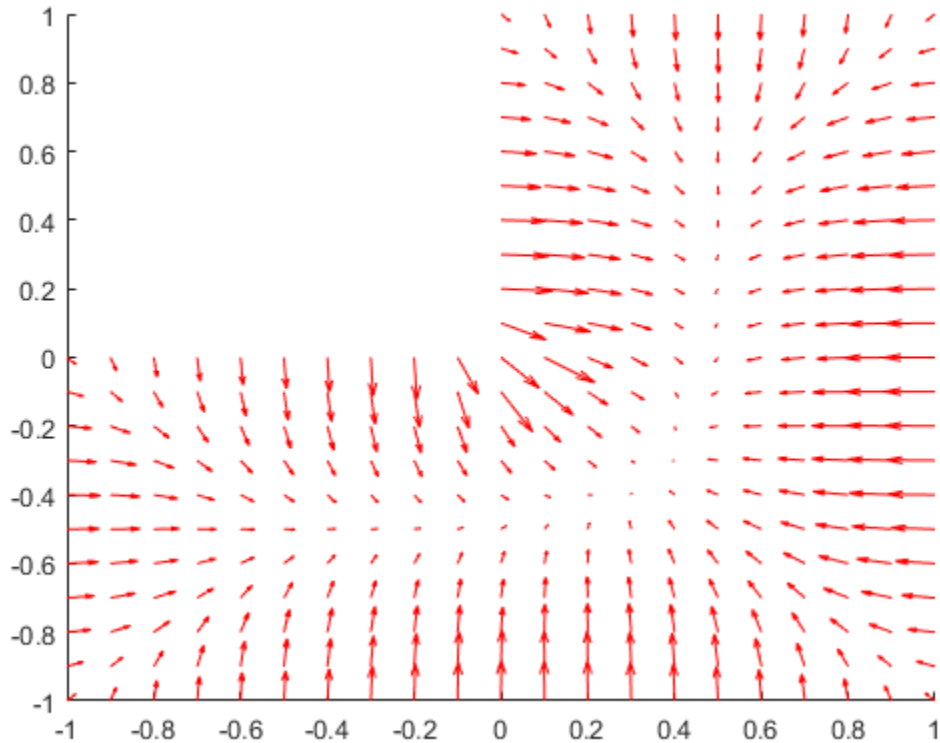
**Composite Plot**

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);
```

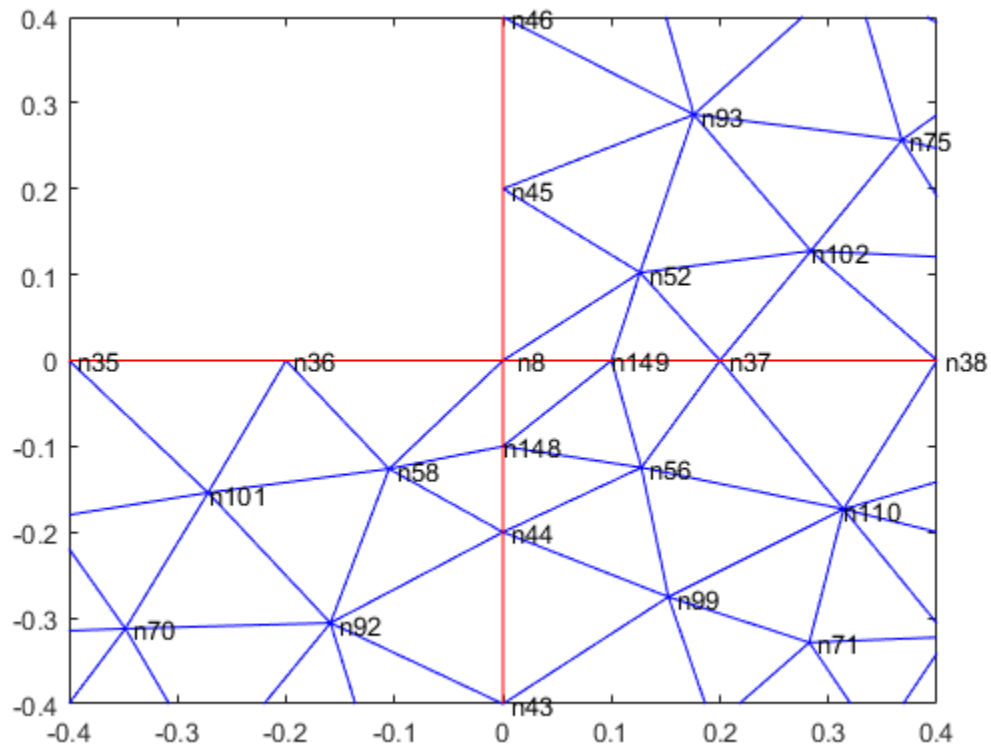Set zero Dirichlet boundary conditions on all edges.

```
applyBoundaryCondition(model,'dirichlet','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: [150×1 double]
        XGradients: [150×1 double]
        YGradients: [150×1 double]
        ZGradients: []
              Mesh: [1×1 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,...
                    'FaceAlpha',0.5,...
                    'FlowData',[ux,uy],...
                    'ColorMap','jet',...
                    'Mesh','on')


h =

  3×1 graphics array:

  Patch
```

```
Quiver
ColorBar
```



### Solution to Transient Thermal Model

Solve a 2-D transient thermal problem.

Create a transient thermal model for this problem.

```
thermalmodel = createpde('thermal','transient');
```

Create the geometry and include it in the model.

```
SQ1 = [3; 4; 0; 3; 3; 0; 0; 0; 3; 3];
D1 = [2; 4; 0.5; 1.5; 2.5; 1.5; 1.5; 0.5; 1.5; 2.5];
gd = [SQ1 D1];
sf = 'SQ1+D1';
ns = char('SQ1','D1');
ns = ns';
dl = decsg(gd,sf,ns);
geometryFromEdges(thermalmodel,dl);
pdegplot(thermalmodel,'EdgeLabels','on','FaceLabels','on')
xlim([-1.5 4.5])
ylim([-0.5 3.5])
axis equal
```



For the square region, assign the following thermal properties:

- Thermal conductivity is $10\,W/(m°C)$

- Mass density is $2\,kg/m^3$

- Specific heat is $0.1\,J/(kg°C)$

```
thermalProperties(thermalmodel,'ThermalConductivity',10, ...
                                'MassDensity',2, ...
                                'SpecificHeat',0.1, ...
                                'Face',1);
```

For the diamond region, assign the following thermal properties:

- Thermal conductivity is $2\,W/(m°C)$

- Mass density is $1\,kg/m^3$

- Specific heat is $0.1\,J/(kg°C)$

```
thermalProperties(thermalmodel,'ThermalConductivity',2, ...
                                'MassDensity',1, ...
                                'SpecificHeat',0.1, ...
                                'Face',2);
```

Assume that the diamond-shaped region is a heat source with the density of $4\,W/m^3$.

```
internalHeatSource(thermalmodel,4,'Face',2);
```

Apply a constant temperature of $0\,°C$ to the sides of the square plate.

```
thermalBC(thermalmodel,'Temperature',0,'Edge',[1 2 7 8]);
```

Set the initial temperature to $0\,°C$.

```
thermalIC(thermalmodel,0);
```

Mesh the geometry.

```
generateMesh(thermalmodel);
```

The dynamics for this problem is very fast--the temperature reaches steady state in about 0.1 seconds. To capture the interesting part of the dynamics, set solution time to

logspace(-2,-1,10). This gives 10 logarithmically spaced solution times between 0.01 and 0.1. Solve the equation.

```
tlist = logspace(-2,-1,10);
thermalresults = solve(thermalmodel,tlist)


thermalresults =

  TransientThermalResults with properties:

      Temperature: [201×10 double]
    SolutionTimes: [1×10 double]
       XGradients: [201×10 double]
       YGradients: [201×10 double]
       ZGradients: []
             Mesh: [1×1 FEMesh]
```

Plot the solution with isothermal lines using a contour plot.

```
T = thermalresults.Temperature;
pdeplot(thermalmodel,'XYData',T(:,10),'Contour','on','ColorMap','hot')
```

### [p,e,t] Mesh and Solution Plots

Plot the `p`,`e`,`t` mesh. Display the solution using 2-D and 3-D colored plots.

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)
```
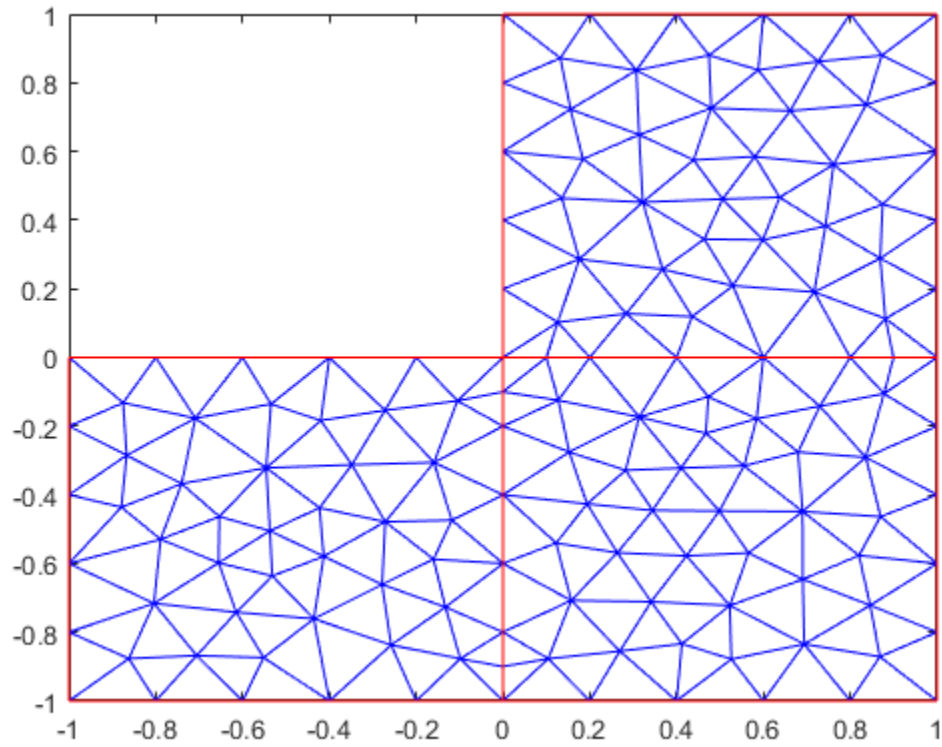
Plot the solution as a 2-D colored plot.

```
pdeplot(p,e,t,'XYData',u)
```
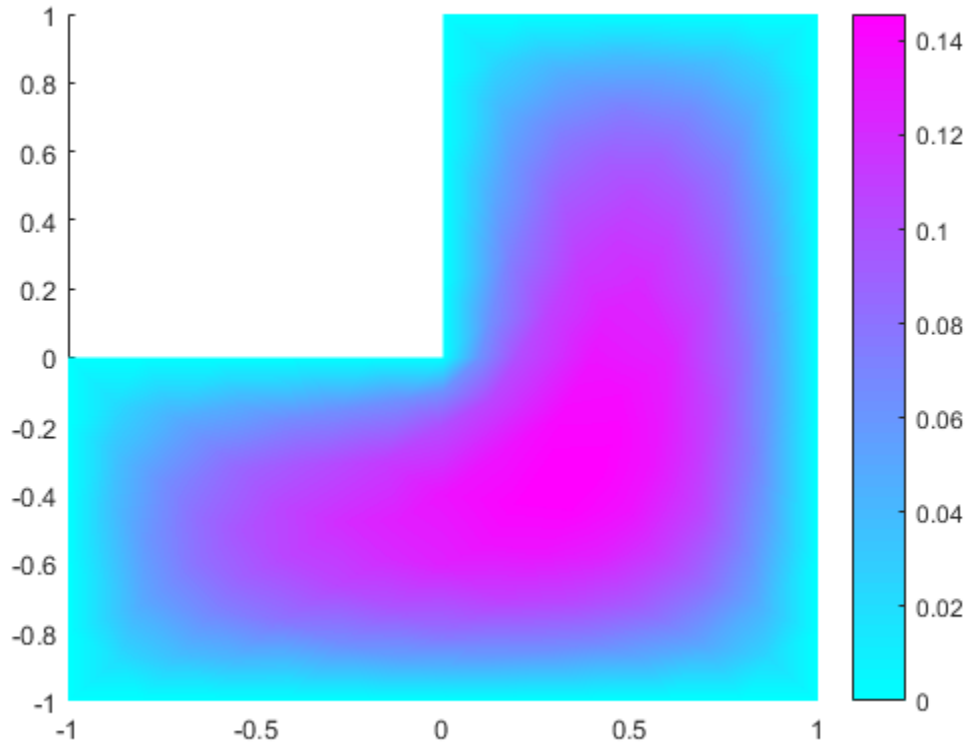
Plot the solution as a 3-D colored plot.

```
pdeplot(p,e,t,'XYData',u,'ZData',u)
```
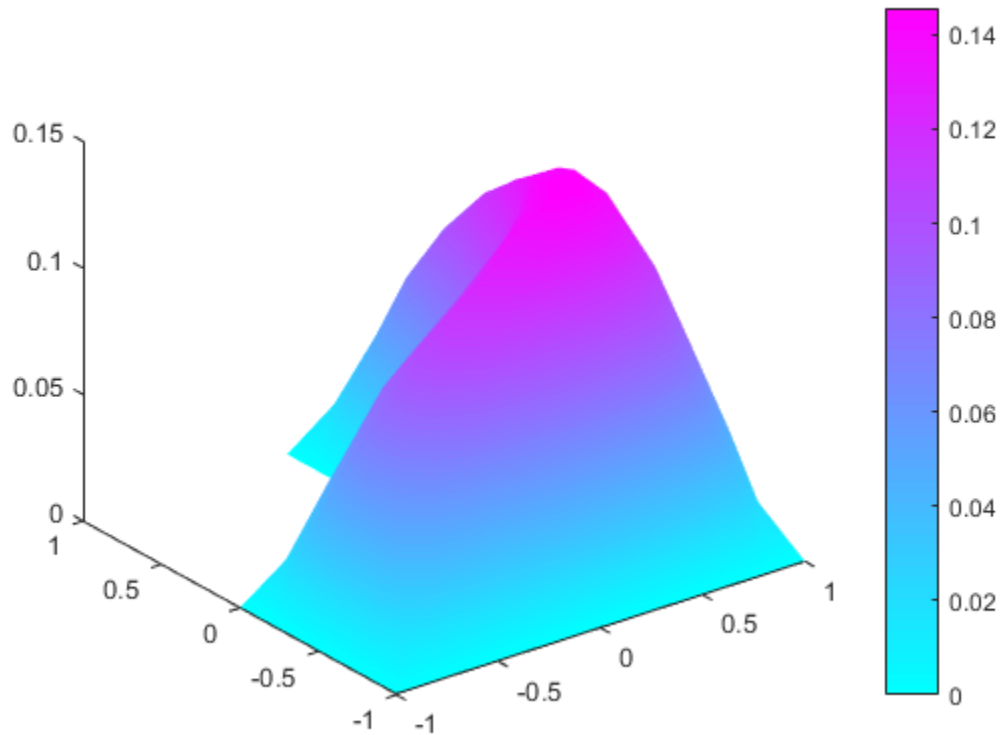
- "Plot 2-D Solutions and Their Gradients" on page 3-199
- "Deflection of a Piezoelectric Actuator" on page 3-15

## Input Arguments

### `model` — PDE model
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

**`thermalmodel` — Thermal model**
ThermalModel object

Thermal model, specified as a `ThermalModel` object. The model contains the geometry, mesh, thermal properties of the material, internal heat source, boundary conditions, and initial conditions.

Example: thermalmodel = createpde('thermal')

**p — Mesh points**
matrix

Mesh points, specified as a 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 2-196.

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

Mesh edges, specified as a 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 2-196.

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

Mesh triangles, specified as a 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 2-196.

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 `'on'` or `'off'`. Here, `'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` value or matrix of such values

Colormap, specified as a value 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 'on' or 'off'. Here,'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 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 `'arrow'` or `'off'`. Here, `'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 a 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 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 `'on'` or `'off'`. Here, `'on'` shows the mesh in the plot.

Example: `'Mesh','on'`

Data Types: `char`

### `'Title'` — Title of plot
set of characters in quotation marks

Title of plot, specified as a set of characters in quotation marks.

Example: `'Title','Solution Plot'`

Data Types: `char`

### `'XYData'` — Colored surface plot data
vector

Colored surface plot data, specified as 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 `'on'` or `'off'`.

---

**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 `'interp'`, `'off'`, or `'flat'`.

- `'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 a matrix. 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 `k`th 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 one of the following.

- `'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. `pdeplot` ignores `ZStyle` when you use it without `ZData`.

Example: `'ZStyle','discontinuous'`

Data Types: `char`

### `'NodeLabels'` — Node labels
`'off'` (default) | `'on'`

Node labels, specified as `'off'` or `'on'`. Note that

- `pdeplot` ignores `NodeLabels` when you use it with `ZData`.
- `pdemesh` ignores `NodeLabels` when you plot solution data as a 3-D plot.

Example: `'NodeLabels','on'`

Data Types: `char`

### `'ElementLabels'` — Element labels
`'off'` (default) | `'on'`

Element labels, specified as `'off'` or `'on'`. Note that

- pdeplot ignores ElementLabels when you use it with ZData.

- pdemesh ignores ElementLabels when you plot solution data as a 3-D plot.

Example: 'ElementLabels','on'

Data Types: char

### **'FaceAlpha'** — Surface transparency for 3-D geometry
1 (default) | real number from 0 through 1

Surface transparency for 3-D geometry, specified as a real number from 0 through 1. The default value 1 indicates no transparency. The value 0 indicates complete transparency.

Example: '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.

# Definitions

## Quiver Plot

Quiver plot is a plot of a vector field. It is 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.

# See Also

## See Also
PDEModel | pdeplot3D

## Topics

**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)

pdeplot3D(thermalmodel, ___ )

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.

pdeplot3D(thermalmodel, ___ ) plots the mesh specified in thermalmodel or the data (temperature or gradient of temperature) on the thermalmodel mesh. When working with a thermal model, use the same syntaxes as for a generic PDE model.

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. First, 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,'dirichlet','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: [28811×1 double]
        XGradients: [28811×1 double]
        YGradients: [28811×1 double]
        ZGradients: [28811×1 double]
             Mesh: [1×1 FEMesh]
```

Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

Plot the solution u on the geometry surface.

```
pdeplot3D(model,'ColorMapData',u)
```

**Solution to Steady-State Thermal Model**

Solve a 3-D steady-state thermal problem.

Create a thermal model for this problem.

```
thermalmodel = createpde('thermal');
```

Import and plot the block geometry.

```
importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabel','on','FaceAlpha',0.5)
axis equal
```

Assign material properties.

```
thermalProperties(thermalmodel,'ThermalConductivity',80);
```

Apply a constant temperature of $100\,^{\circ}C$ to the left side of the block (face 1) and a constant temperature of $300\,^{\circ}C$ at the right side of the block (face 3). All other faces are insulated by default.

```
thermalBC(thermalmodel,'Face',1,'Temperature',100);
thermalBC(thermalmodel,'Face',3,'Temperature',300);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
```

```
thermalresults = solve(thermalmodel)


thermalresults =

  SteadyStateThermalResults with properties:

    Temperature: [28811×1 double]
     XGradients: [28811×1 double]
     YGradients: [28811×1 double]
     ZGradients: [28811×1 double]
           Mesh: [1×1 FEMesh]
```

The solver finds the values of temperatures and temperature gradients at the nodal locations. To access these values, use `thermalresults.Temperature`, `thermalresults.XGradients`, and so on. For example, plot temperatures at nodal locations.

```
pdeplot3D(thermalmodel,'ColorMapData',thermalresults.Temperature)
```

### Plot Mesh Nodes and Elements with Labels

Create a PDE model, include the geometry and mesh it.

```
model = createpde;
importGeometry(model,'Tetrahedron.stl');
generateMesh(model,'Hmax',20,'GeometricOrder','linear');
```

Display the node labels on the surface of a simple mesh.

```
pdeplot3D(model,'NodeLabels','on')
view(101,12)
```
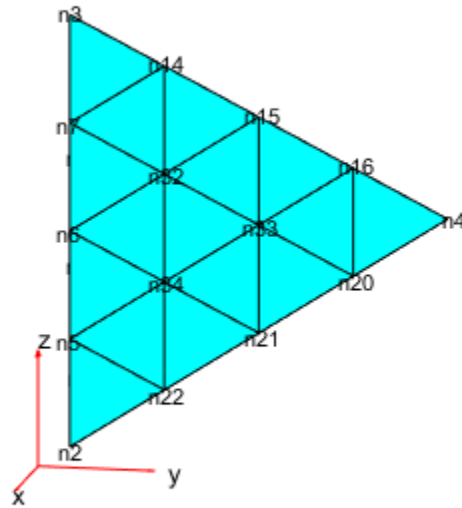
Display the element labels.

```
pdeplot3D(model,'ElementLabels','on')
view(101,12)
```
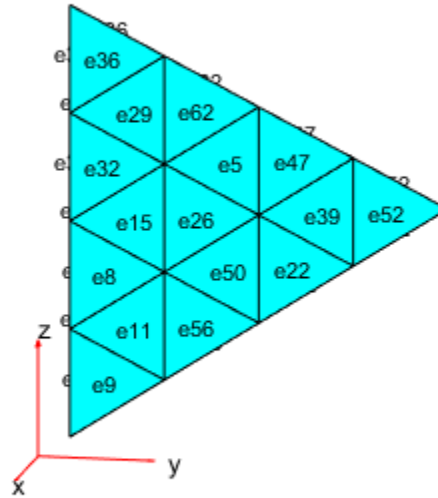
- "Plot 3-D Solutions and Their Gradients" on page 3-210

## Input Arguments

**`model` — PDE model**
PDEModel object

PDE model, specified as a PDEModel object.

Example: model = createpde(1)

**`thermalmodel` — Thermal model**
ThermalModel object

Thermal model, specified as a `ThermalModel` object. The model contains the geometry, mesh, thermal properties of the material, internal heat source, boundary conditions, and initial conditions.

Example: `thermalmodel = createpde('thermal')`

### 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 the `solvepde` function.

Example: `results = solvepde(model); u = results.NodalSolution`

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'`. Note that

- pdeplot ignores `NodeLabels` when you use it with `ZData`.
- pdemesh ignores `NodeLabels` when you plot solution data as a 3-D plot.

Example: `'NodeLabels','on'`

Data Types: `char`

### `'ElementLabels'` — Element labels
`'off'` (default) | `'on'`

Element labels, specified as `'off'` or `'on'`. Note that

- pdeplot ignores `ElementLabels` when you use it with `ZData`.

- pdemesh ignores ElementLabels when you plot solution data as a 3-D plot.

Example: 'ElementLabels','on'

Data Types: char

**'FaceAlpha' — Surface transparency for 3-D geometry**
1 (default) | real number from 0 through 1

Surface transparency for 3-D geometry, specified as a real number from 0 through 1. The default value 1 indicates no transparency. The value 0 indicates complete transparency.

Example: '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.

# See Also

## See Also
PDEModel | pdeplot

## Topics
"Plot 3-D Solutions and Their Gradients" on page 3-210
"Solve Problems Using PDEModel Objects" on page 2-5

**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

**Introduced before R2006a**

# 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

**Introduced before R2006a**

# 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

**Introduced before R2006a**

# 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.

**Introduced before R2006a**

# 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 \| character expression \| {0.3} | Poisson's ratio. Applies to calculating von Mises ('vonmises') effective |

| Property Name | Property Value/Default | Description |
|---|---|---|
| | | 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 character expression in coefficient form: "Specify Scalar PDE Coefficients in Character Form" on page 2-55. |

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 ($\varepsilon_x$)
- 'eyy', the $y$-direction strain ($\varepsilon_y$)
- 'exy', the shear strain ($\gamma_{xy}$)
- 'sxx', the $x$-direction stress ($\sigma_x$)
- 'syy', the $y$-direction stress ($\sigma_y$)
- 'sxy', the shear stress ($\tau_{xy}$)
- 'e1', the first principal strain ($\varepsilon_1$)
- 'e2', the second principal strain ($\varepsilon_2$)

- `'s1'`, the first principal stress ($\sigma_1$)
- `'s2'`, the second principal stress ($\sigma_2$)
- `'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 "Plane Stress and Plane Strain" 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);
```

**Introduced before R2006a**

# PDESolverOptions Properties

Algorithm options for PDE solvers

## Compatibility

This page describes the recommended workflow.

## Description

A `PDESolverOptions` object contains options used by the solvers when solving a PDE problem specified as `PDEModel`. A `PDEModel` object contains a `PDESolverOptions` object in its `SolverOptions` property.

## Properties

### `AbsoluteTolerance` — Absolute tolerance for internal ODE solver
1.0000e-06 (default) | positive real number

Absolute tolerance for internal ODE solver, returned as a positive real number. Absolute tolerance is a threshold below which the value of the solution component is unimportant. This property determines the accuracy when the solution approaches zero.

Example: `model.SolverOptions.AbsoluteTolerance = 5.0000e-06`

Data Types: `double`

### `RelativeTolerance` — Relative tolerance for internal ODE solver
1.0000e-03 (default) | positive real number

Relative tolerance for internal ODE solver, returned as a positive real number. This tolerance is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components, except those smaller than thresholds imposed by `AbsoluteTolerance`. The default value corresponds to 0.1% accuracy.

Example: `model.SolverOptions.RelativeTolerance = 5.0000e-03`

Data Types: `double`

**ResidualTolerance — Acceptable residual tolerance for internal nonlinear solver**
1.0000e-04 (default) | positive real number

Acceptable residual tolerance for internal nonlinear solver, returned as a positive real number. The nonlinear solver iterates until the residual size is less than the value of ResidualTolerance.

Example: model.SolverOptions.ResidualTolerance = 5.0000e-04

Data Types: double

**MaxIterations — Maximal number of Gauss-Newton iterations allowed for the nonlinear solver**
25 (default) | positive real number

Maximal number of Gauss-Newton iterations allowed for the nonlinear solver, returned as a positive integer.

Example: model.SolverOptions.MaxIterations = 30

Data Types: double

**MinStep — Minimum damping of search direction for the nonlinear solver**
1.5259e-05 (default) | positive real number

Minimum damping of search direction for the nonlinear solver, returned as a positive real number.

Example: model.SolverOptions.MinStep = 1.5259e-7

Data Types: double

**ResidualNorm — Residual norm**
Inf (default) | -Inf | positive real number | 'energy'

Residual norm, returned as the p value for $L^p$ norm, or as 'energy'. For the $L^p$-norm, 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: model.SolverOptions.ResidualNorm = 'energy'

Data Types: double | char

**ReportStatistics — Flag that controls the display of internal nonlinear and ODE solver statistics and convergence report during the solution process**
'off' (default) | 'on'

Flag that controls the display of internal nonlinear and ODE solver statistics and convergence report during the solution process, returned as `'off'` or `'on'`.

Example: `model.SolverOptions.ReportStatistics = 'on'`

Data Types: `char`

# See Also

## See Also
`PDEModel` | `solvepde` | `solvepdeeig`

**Introduced in R2016a**

# pdesurf

Shorthand command for surface plot

## Compatibility

This page describes the legacy workflow. 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

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)
```
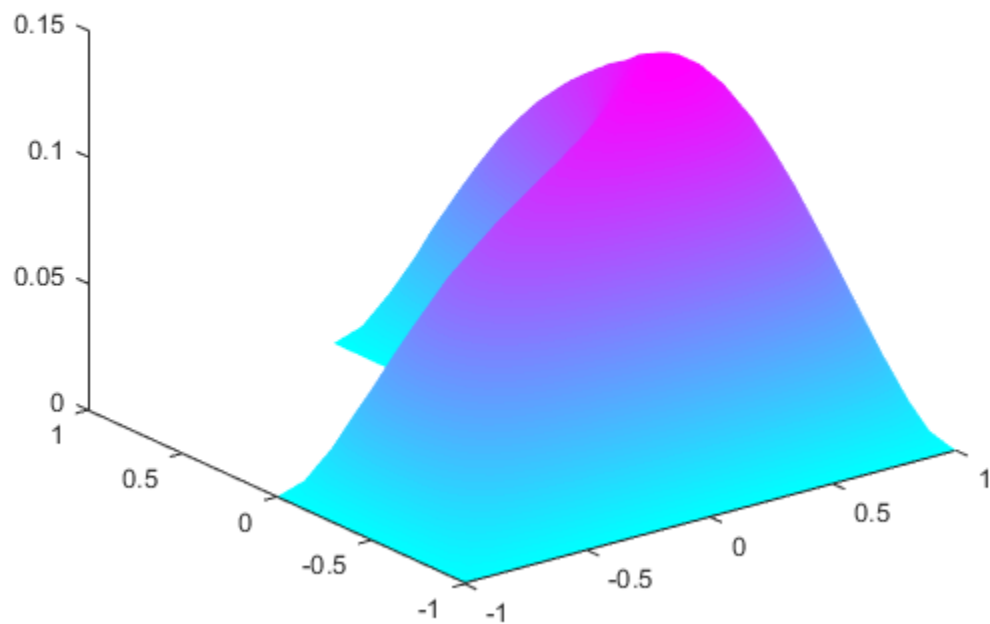
## See Also

pdecont | pdemesh | pdeplot

**Introduced before R2006a**

# PDE

Solve partial differential equations in 2-D regions

## Description

The PDE app provides an interactive interface for solving 2-D geometry problems. Using the app, you can create complex geometries by drawing, overlapping, and rotating basic shapes, such as circles, polygons and so on. The app also includes preset modes for applications, such as electrostatics, magnetostatics, heat transfer, and so on.

When solving a PDE problem in the app, follow these steps:

1 Create a 2-D geometry.
2 Specify boundary conditions.
3 Specify equation coefficients.
4 Generate mesh.
5 Specify parameters for solving a PDE. The set of parameters depends on the type of PDE. For parabolic and hyperbolic PDEs, these parameters include initial conditions.
6 Solve the problem.
7 Specify plotting parameters and plot the results.

You can choose to export data to the MATLAB workspace from any step in the app and continue your work outside the app.

**Note:** The app does not support 3-D geometry problems and systems of more than two PDEs.

## Open the PDE App

· MATLAB Toolstrip: On the **Apps** tab, under **Math, Statistics and Optimization**, click the app icon.
· MATLAB command prompt: Enter `pdetool`.

# Examples

- "Plane Stress and Plane Strain" on page 3-7
- "Conductive Media DC" on page 3-96
- "Heat Equation for a Block with Cavity: PDE App" on page 3-139
- "L-Shaped Membrane with a Rounded Corner" on page 3-177

## Programmatic Use

pdetool opens the PDE app or brings focus to the app if it is already open.

pdecirc(xc,yc,r) opens the PDE app and draws a circle with center in (xc,yc) and radius r.

pdeellip(xc,yc,a,b,phi) opens the PDE app and draws an ellipse with center in (xc,yc) and semiaxes a and b. The rotation of the ellipse (in radians) is phi.

pdepoly(x,y) opens the PDE app and draws a polygon with corner coordinates defined by x and y.

pderect([xmin xmax ymin ymax]) opens the PDE app and draws a rectangle with corner coordinates defined by [xmin xmax ymin ymax].

# See Also

## See Also

**Functions**
pdecirc | pdeellip | pdepoly | pderect | pdetool

**Using Objects**
PDEModel

## Topics
"Plane Stress and Plane Strain" on page 3-7
"Conductive Media DC" on page 3-96
"Heat Equation for a Block with Cavity: PDE App" on page 3-139

"L-Shaped Membrane with a Rounded Corner" on page 3-177

**Introduced before R2006a**

# pdetool

Open PDE app

## Syntax

```
pdetool
```

## Description

`pdetool` opens the PDE app or brings focus to the app if it is already open. The PDE app provides an interactive interface for solving 2-D geometry problems. Using the app, you can create complex geometries by drawing, overlapping, and rotating basic shapes, such as circles, polygons and so on. The app also includes preset modes for applications, such as electrostatics, magnetostatics, heat transfer, and so on. You can choose to export data to the MATLAB workspace from any step in the app and continue your work outside the app.

**Note:** The app does not support 3-D geometry problems and systems of more than two PDEs.

You can save the model, specifications, and results in a file. The file is a MATLAB function that contains the MATLAB commands necessary to create a CSG model, specify boundary conditions and coefficients, create the mesh, solve the PDE, and plot the solution. To save the file or open the existing file use the **File** menu. Editing this file directly is not recommended.

## Examples

### Open the PDE App

Open the PDE app from the MATLAB command prompt. If the PDE app is already open, typing `pdetool` overwrites any existing information in the app.

```
pdetool
```
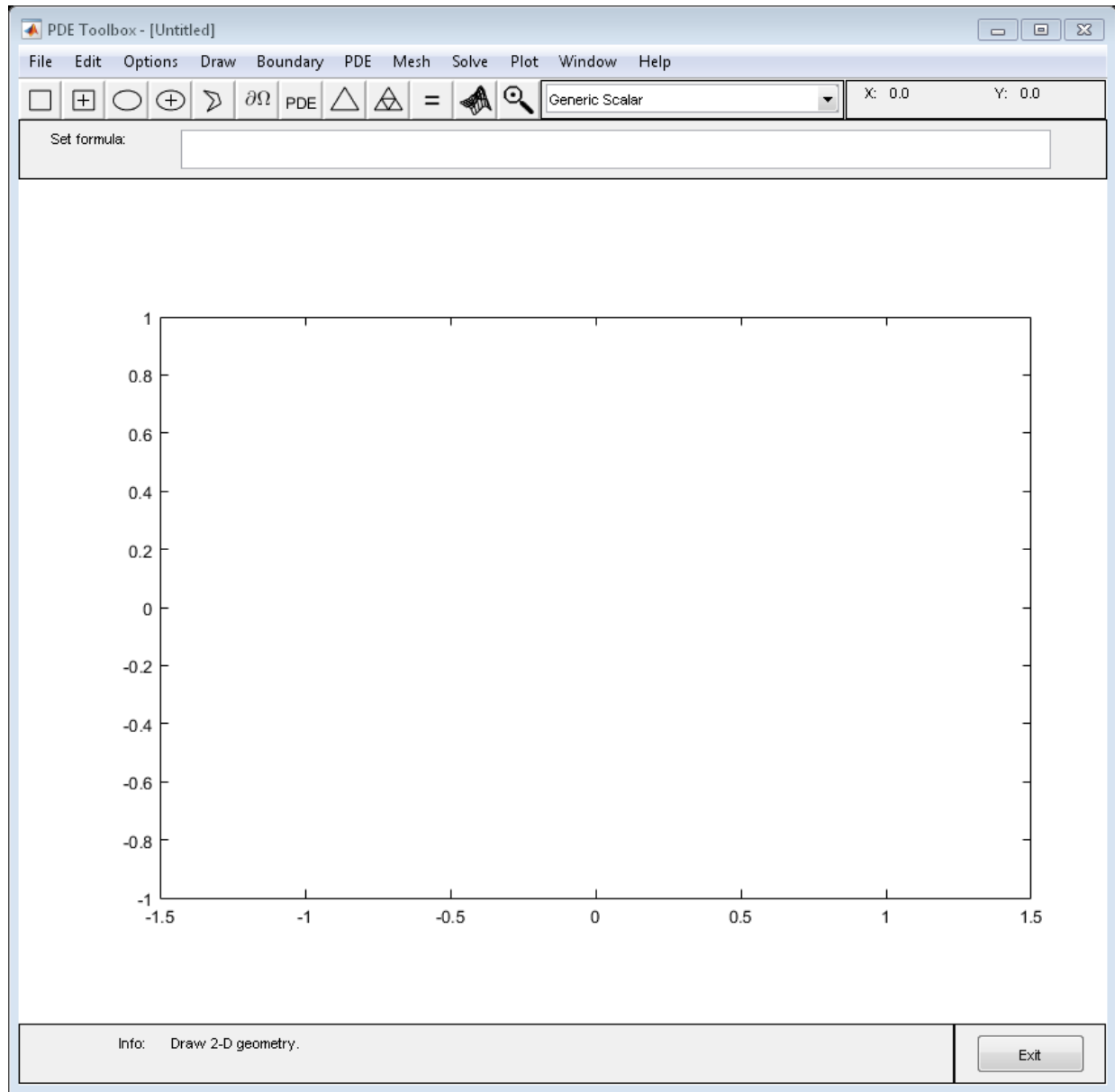
- "Open the PDE App" on page 4-2

## See Also

### See Also
PDE | pdecirc | pdeellip | pdepoly | pderect

### Topics

**Introduced before R2006a**

# 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`.

**Introduced before R2006a**

# 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.

## See Also

**Introduced before R2006a**

# 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 2-5.

## Syntax

```
K = poiasma(n1,n2,h1,h2)
K = poiasma(n1,n2)
K = poiasma(n)
```

## Description

`K = poiasma(n1,n2,h1,h2)` assembles the contributions to the stiffness matrix from boundary points. `n1` and `n2` are the numbers of points in the first and second directions, and `h1` and `h2` are the mesh spacings. `K` is a sparse `n1*n2`-by-`n1*n2` matrix. The point numbering is the canonical numbering for a rectangular mesh.

`K = poiasma(n1,n2)` uses `h1 = h2`.

`K = poiasma(n)` uses `n1 = n2 = n`.

## See Also

**Introduced before R2006a**

# 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 2-5.

## Syntax

```
u = poicalc(f,h1,h2,n1,n2)

u = poicalc(f,h1,h2)

u = poicalc(f)
```

## Description

`u = poicalc(f,h1,h2,n1,n2)` calculates the solution of Poisson's equation for the interior points of an evenly spaced rectangular grid. The columns of `u` contain the solutions corresponding to the columns of the right-hand side `f`. `h1` and `h2` are the spacings in the first and second direction, and `n1` and `n2` are the number of points.

The number of rows in `f` must be `n1*n2`. If `n1` and `n2` are not given, the square root of the number of rows of `f` is assumed. If `h1` and `h2` are not given, they are assumed to be equal.

The ordering of the rows in `u` and `f` is the canonical ordering of interior points, as returned by `poiindex`.

The solution is obtained by sine transforms in the first direction and tridiagonal matrix solution in the second direction. `n1` should be 1 less than a power of 2 for best performance.

## See Also

**Introduced before R2006a**

# 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 2-5.

## 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]`.

**Introduced before R2006a**

# 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 2-5.

## 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.

**Introduced before R2006a**

# 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 2-5.

## 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.

**Introduced before R2006a**

# 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 2-196.

`[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

### Mesh Refinement

Refine the mesh of the L-shaped membrane several times. Plot the mesh for the geometry of the L-shaped membrane.

```
[p,e,t] = initmesh('lshapeg','hmax',inf);
subplot(2,2,1), pdemesh(p,e,t)
[p,e,t] = refinemesh('lshapeg',p,e,t);
subplot(2,2,2), pdemesh(p,e,t)
[p,e,t] = refinemesh('lshapeg',p,e,t);
subplot(2,2,3), pdemesh(p,e,t)
[p,e,t] = refinemesh('lshapeg',p,e,t);
subplot(2,2,4), pdemesh(p,e,t)
subplot
```
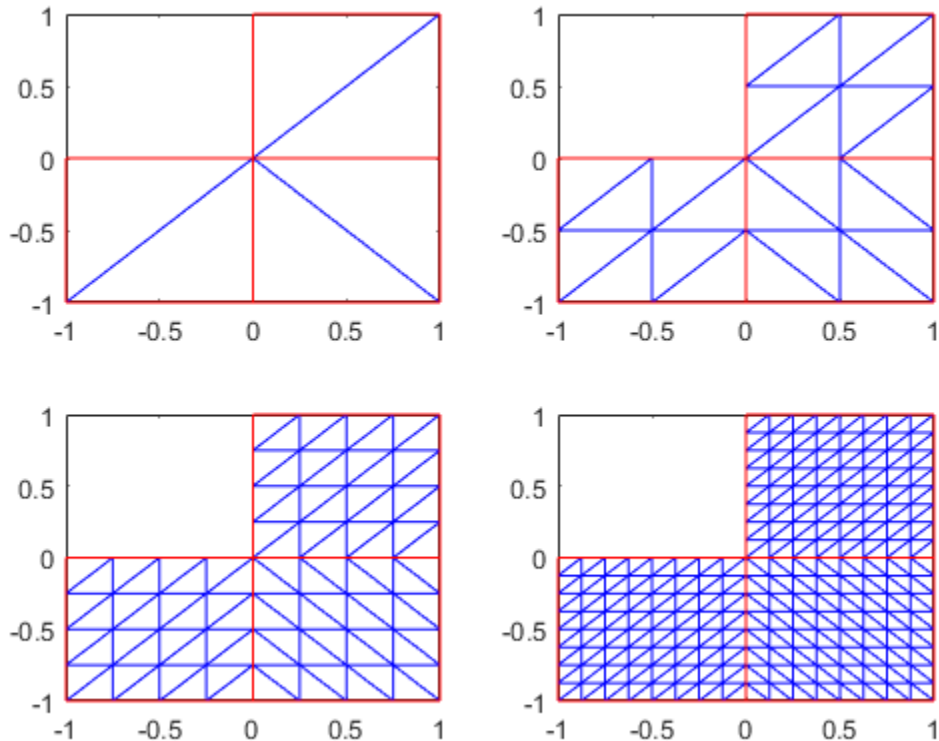
## Algorithms

The algorithm is described by the following steps:

1 Pick the initial set of triangles to be refined.

2 Either divide all edges of the selected triangles in half (regular refinement), or divide the longest edge in half (longest edge refinement).

3 Divide the longest edge of any triangle that has a divided edge.

4 Repeat step 3 until no further edges are divided.

5 Introduce new points of all divided edges, and replace all divided entries in e by two new entries.

6-597

**6**  Form the new triangles. If all three sides are divided, new triangles are formed by joining the side midpoints. If two sides are divided, the midpoint of the longest edge is joined with the opposing corner and with the other midpoint. If only the longest edge is divided, its midpoint is joined with the opposing corner.

## See Also

initmesh | pdeent | pdesdt

## Topics

"Mesh Data" on page 2-196

**Introduced before R2006a**

# 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 2-147.

## Syntax

```
setInitialConditions(model,u0)
setInitialConditions(model,u0,ut0)
setInitialConditions( ___ ,RegionType,RegionID)

setInitialConditions(model,results)
setInitialConditions(model,results,iT)

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.

---

**Note:** Include geometry in `model` before using `setInitialConditions`.

---

`setInitialConditions(model,u0,ut0)` use this syntax for time-dependent problems where a time derivative is second order, such as a hyperbolic problem.

`setInitialConditions( ___ ,RegionType,RegionID)` sets initial conditions on a geometry region using any of the arguments in the previous syntaxes.

`setInitialConditions(model,results)` sets the initial guess for stationary nonlinear problems using the solution `results` from a previous analysis on the same geometry and mesh. The initial derivative for stationary problems is 0.

`setInitialConditions(model,results,iT)` sets the initial conditions for time-dependent problems using the solution `results` corresponding to the solution time index `iT`. If you do not specify the time index `iT`, `setInitialConditions` uses the last solution time in `results`.

`ic = setInitialConditions( ___ )` returns a handle to the initial conditions object.

## Examples

### 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);
```

### Constant Initial Conditions for 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: [5×1 double]
    InitialDerivative: []
```

### 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,'FaceLabels','on')
axis equal
ylim([-1.1,1.1])
```
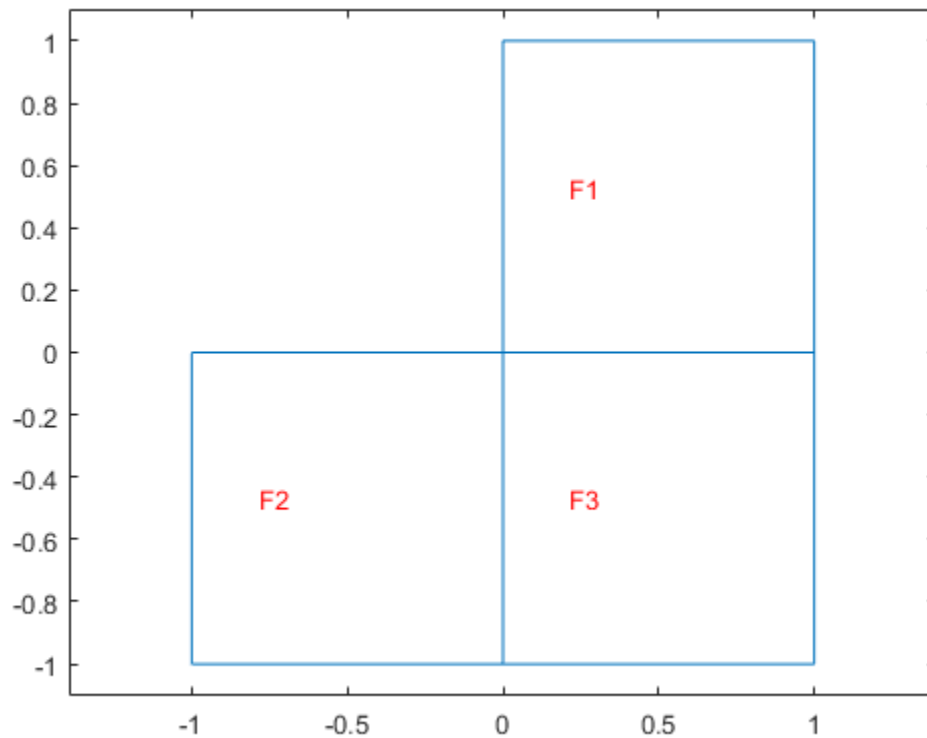
Set subdomain 1 to initial value -1, subdomain 2 to initial value 1, and subdomain 3 to initial value 5.

```
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.

### 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,'FaceLabels','on')
axis equal
ylim([-1.1,1.1])
```
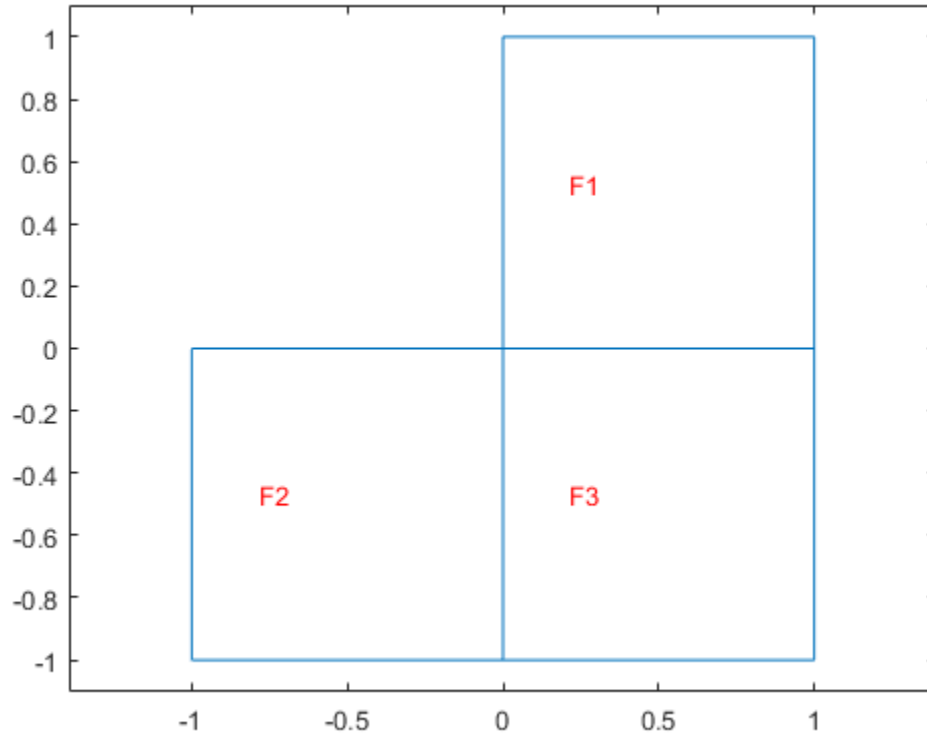
Set the initial conditions to $x^2 + y^2$.

```
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);
```

**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 value to be 0 for all components. Set the initial derivative.

$$ut0 = \begin{bmatrix} 4 + \frac{x}{x^2+y^2+z^2} \\ 5 - \tanh(z) \\ 10\frac{y}{x^2+y^2+z^2} \end{bmatrix}$$

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;
end
```

Set the initial conditions.

```
setInitialConditions(model,0,@ut0fun)


ans =

  GeometricInitialConditions with properties:

          RegionType: 'cell'
```

```
         RegionID: 1
      InitialValue: O
  InitialDerivative: @utOfun
```

### Initial Condition Is Previously Obtained Solution

Set initial conditions using the solution from a previous analysis on the same geometry and mesh.

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','FaceLabels','on')
axis equal
ylim([-0.1,2.1])
```
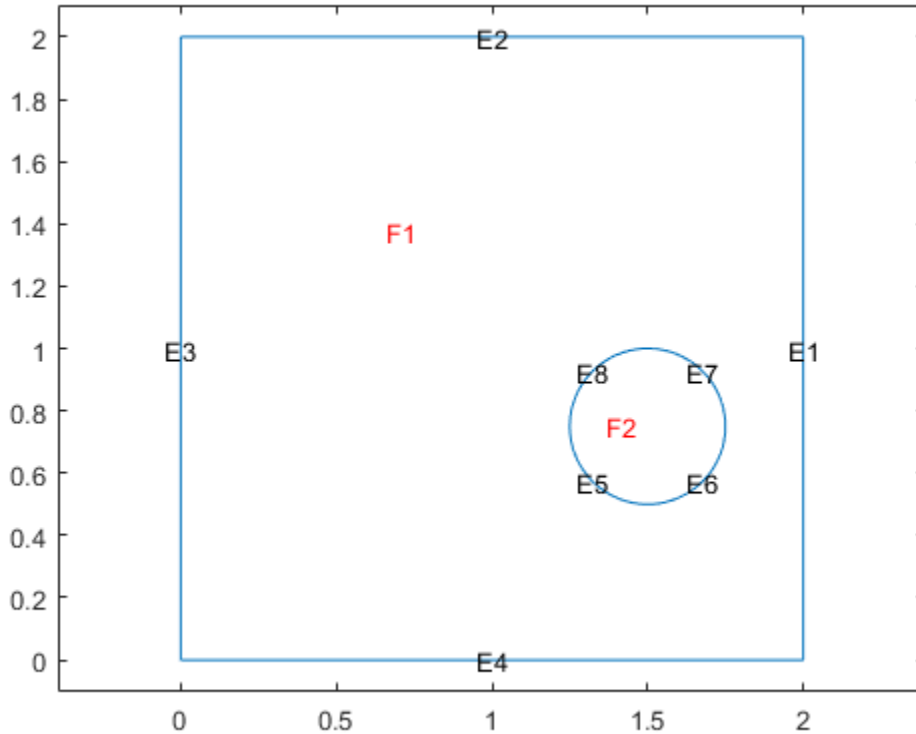
Include the geometry in a PDE model, set boundary and initial conditions, and specify coefficients.

```
model = createpde();
geometryFromEdges(model,dl);

% Set boundary conditions that the upper and left edges are at temperature 10.
applyBoundaryCondition(model,'dirichlet','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);
```

```
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')
```

Now, resume the analysis and solve the problem for times from 1/2 to 1. Use the previously obtained solution for time 1/2 as an initial condition. Since 1/2 is the last element in `tlist`, you do not need to specify the solution time index. By default, `setInitialConditions` uses the last solution index.

```
setInitialConditions(model,results)
```

```
ans =

  NodalInitialConditions with properties:

          InitialValue: [2814×1 double]
     InitialDerivative: []
```

Solve the problem for times 1/2 through 1 in steps of 0.01.

```
tlist1 = 0.5:0.01:1.0;
results1 = solvepde(model,tlist1);
```

Plot the solution for times 0.5, 0.7, 0.9, and 1.

```
sol1 = results1.NodalSolution;

figure

subplot(2,2,1)
pdeplot(model,'XYData',sol1(:,1))
title('Time 0.5')
subplot(2,2,2)
pdeplot(model,'XYData',sol1(:,21))
title('Time 0.7')
subplot(2,2,3)
pdeplot(model,'XYData',sol1(:,41))
title('Time 0.9')
subplot(2,2,4)
pdeplot(model,'XYData',sol1(:,51))
title('Time 1.0')
```
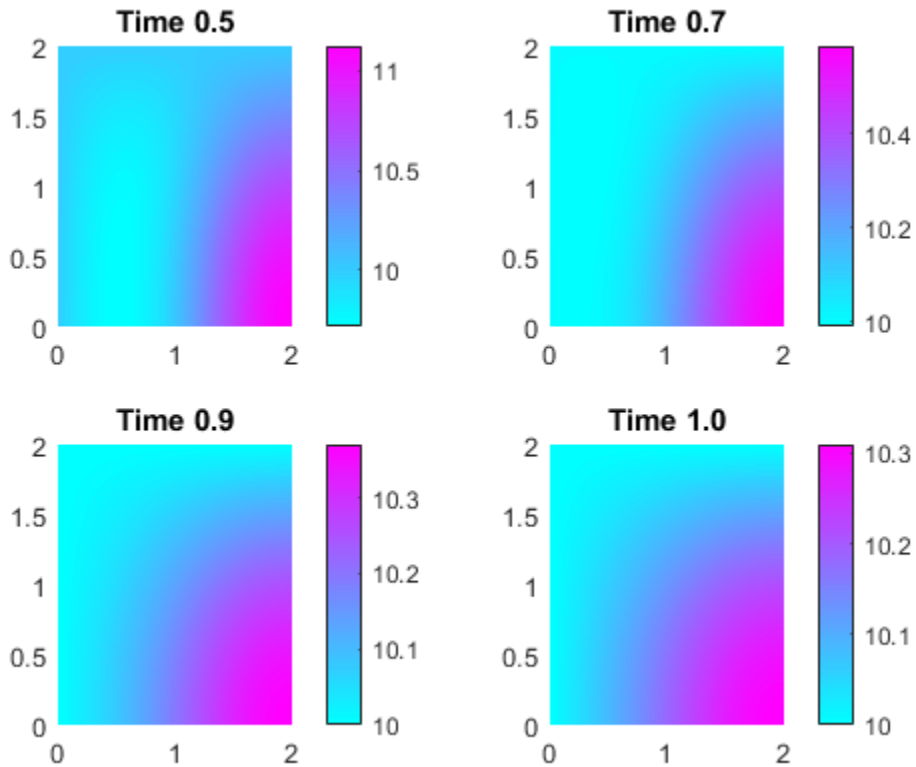
To use the previously obtained solution for a particular solution time instead of the last one, specify the solution time index as a third parameter of `setInitialConditions`. For example, use the solution at time 0.2, which is the 21st element in `tlist`.

```
setInitialConditions(model,results,21)
```

```
ans =

  NodalInitialConditions with properties:

        InitialValue: [2814×1 double]
    InitialDerivative: []
```

Solve the problem for times 0.2 through 1 in steps of 0.01.

```
tlist2 = 0.2:0.01:1.0;
results2 = solvepde(model,tlist2);
```

- "Set Initial Conditions" on page 2-140

## 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 "Equations You Can Solve Using Recommended Functions" on page 1-6.

- Scalar — Use it to represent a constant initial value for all solution components throughout the domain.
- Column vector — Use it to represent a constant initial value for each of the *N* solution components throughout the domain.
- Function handle — Use it to represent 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 "Equations You Can Solve Using Recommended Functions" on page 1-6. You must specify `ut0` when there is a nonzero second-order time-derivative coefficient m.

- Scalar — Use it to represent a constant initial value for all solution components throughout the domain.
- Column vector — Use it to represent a constant initial value for each of the *N* solution components throughout the domain.
- Function handle — Use it to represent 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'` | `'Vertex'` | `'Cell'`

Geometric region type, specified as `'Face'`, `'Edge'`, `'Vertex'`, 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 the same geometric region, solvers use the last applied setting.
- If there are separate assignments to a geometric region and the boundaries of that region, the solvers use the specified assignment on the region and choose the assignment on the boundary as follows. The solvers give an `'Edge'` assignment precedence over a `'Face'` assignment, even if you specify a `'Face'` assignment after an `'Edge'` assignment. The precedence levels are `'Vertex` (highest precedence), `'Edge'`, `'Face'`, `'Cell'` (lowest precedence).

- If there is an assignment made with the `results` object, solvers use that assignment instead of all previous assignments.

Example: `setInitialConditions(model,10,'Face',1:4)`

Data Types: `char`

### RegionID — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot`, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Example: `setInitialConditions(model,10,'Face',1:4)`

Data Types: `double`

### 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)`

### iT — Time index
positive integer

Time index, specified as a positive integer.

Example: `setInitialConditions(model,results,21)`

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 in the case of a geometric assignment or the nodes in the case of a results-based assignment.

## Tips

- To ensure that the model has the correct `TimeDependent` property setting, if possible specify coefficients before setting initial conditions.

- To avoid assigning initial conditions to a wrong region, ensure that you are using the correct geometric region IDs by plotting and visually inspecting the geometry.

## See Also

### See Also
`findInitialConditions` | `pdegplot` | `PDEModel`

### Topics
"Set Initial Conditions" on page 2-140
"Solve Problems Using PDEModel Objects" on page 2-5
"Equations You Can Solve Using Recommended Functions" on page 1-6

**Introduced in R2016a**

# solve

Solve heat transfer problem specified in a ThermalModel

## Syntax

```
thermalresults = solve(thermalmodel)
thermalresults = solve(thermalmodel,tlist)
```

## Description

`thermalresults = solve(thermalmodel)` returns the solution to the steady-state thermal model represented in `thermalmodel`.

`thermalresults = solve(thermalmodel,tlist)` returns the solution to the transient thermal model represented in `thermalmodel` at the times `tlist`.

## Examples

### Solution to Steady-State Thermal Model

Solve a 3-D steady-state thermal problem.

Create a thermal model for this problem.

```
thermalmodel = createpde('thermal');
```

Import and plot the block geometry.

```
importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabel','on','FaceAlpha',0.5)
axis equal
```

Assign material properties.

```
thermalProperties(thermalmodel,'ThermalConductivity',80);
```

Apply a constant temperature of $100\,^{\circ}C$ to the left side of the block (face 1) and a constant temperature of $300\,^{\circ}C$ at the right side of the block (face 3). All other faces are insulated by default.

```
thermalBC(thermalmodel,'Face',1,'Temperature',100);
thermalBC(thermalmodel,'Face',3,'Temperature',300);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
```

```
thermalresults = solve(thermalmodel)


thermalresults =

  SteadyStateThermalResults with properties:

    Temperature: [28811×1 double]
     XGradients: [28811×1 double]
     YGradients: [28811×1 double]
     ZGradients: [28811×1 double]
           Mesh: [1×1 FEMesh]
```

The solver finds the values of temperatures and temperature gradients at the nodal locations. To access these values, use `thermalresults.Temperature`, `thermalresults.XGradients`, and so on. For example, plot temperatures at nodal locations.

```
pdeplot3D(thermalmodel,'ColorMapData',thermalresults.Temperature)
```

### Solution to Transient Thermal Model

Solve a 2-D transient thermal problem.

Create a transient thermal model for this problem.

```
thermalmodel = createpde('thermal','transient');
```

Create the geometry and include it in the model.

```
SQ1 = [3; 4; 0; 3; 3; 0; 0; 0; 3; 3];
D1 = [2; 4; 0.5; 1.5; 2.5; 1.5; 1.5; 0.5; 1.5; 2.5];
gd = [SQ1 D1];
sf = 'SQ1+D1';
```

```
ns = char('SQ1','D1');
ns = ns';
dl = decsg(gd,sf,ns);
geometryFromEdges(thermalmodel,dl);
pdegplot(thermalmodel,'EdgeLabels','on','FaceLabels','on')
xlim([-1.5 4.5])
ylim([-0.5 3.5])
axis equal
```



For the square region, assign the following thermal properties:

- Thermal conductivity is $10\,W/(m^\circ C)$

- Mass density is $2\,kg/m^3$

- Specific heat is $0.1 \, J/(kg°C)$

```
thermalProperties(thermalmodel,'ThermalConductivity',10, ...
                               'MassDensity',2, ...
                               'SpecificHeat',0.1, ...
                               'Face',1);
```

For the diamond region, assign the following thermal properties:

- Thermal conductivity is $2 \, W/(m°C)$

- Mass density is $1 \, kg/m^3$

- Specific heat is $0.1 \, J/(kg°C)$

```
thermalProperties(thermalmodel,'ThermalConductivity',2, ...
                               'MassDensity',1, ...
                               'SpecificHeat',0.1, ...
                               'Face',2);
```

Assume that the diamond-shaped region is a heat source with the density of $4 \, W/m^3$.

```
internalHeatSource(thermalmodel,4,'Face',2);
```

Apply a constant temperature of $0 \, °C$ to the sides of the square plate.

```
thermalBC(thermalmodel,'Temperature',0,'Edge',[1 2 7 8]);
```

Set the initial temperature to $0 \, °C$.

```
thermalIC(thermalmodel,0);
```

Mesh the geometry.

```
generateMesh(thermalmodel);
```

The dynamics for this problem is very fast--the temperature reaches steady state in about 0.1 seconds. To capture the interesting part of the dynamics, set solution time to `logspace(-2,-1,10)`. This gives 10 logarithmically spaced solution times between 0.01 and 0.1. Solve the equation.

```
tlist = logspace(-2,-1,10);
```

```
thermalresults = solve(thermalmodel,tlist)


thermalresults =

  TransientThermalResults with properties:

      Temperature: [201×10 double]
    SolutionTimes: [1×10 double]
       XGradients: [201×10 double]
       YGradients: [201×10 double]
       ZGradients: []
             Mesh: [1×1 FEMesh]
```

Plot the solution with isothermal lines using a contour plot.

```
T = thermalresults.Temperature;
pdeplot(thermalmodel,'XYData',T(:,10),'Contour','on','ColorMap','hot')
```

## Input Arguments

### `thermalmodel` — Thermal model
`ThermalModel` object

Thermal model, specified as a `ThermalModel` object. The model contains the geometry, mesh, thermal properties of the material, internal heat source, boundary conditions, and initial conditions.

Example: `thermalmodel = createpde('thermal')`

**`tlist` — Solution times**
real vector

Solution times, specified as a real vector. `tlist` must be a monotonic vector (increasing or decreasing).

Example: `0:20`

Data Types: `double`

# Output Arguments

**`thermalresults` — PDE results**
`SteadyStateThermalResults` object | `TransientThermalResults` object

Thermal results, returned as a `SteadyStateThermalResults` object or as a `TransientThermalResults` object. The type of `thermalresults` depends on whether `thermalmodel` represents a steady-state problem (`thermalmodel.AnalysisType = 'steadystate'`) or a transient problem (`thermalmodel.AnalysisType = 'transient'`).

# See Also

### See Also
`geometryFromEdges` | `geometryFromMesh` | `importGeometry` | `internalHeatSource` | `thermalBC` | `thermalIC` | `ThermalModel` | `thermalProperties`

**Introduced in R2017a**

# 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
```
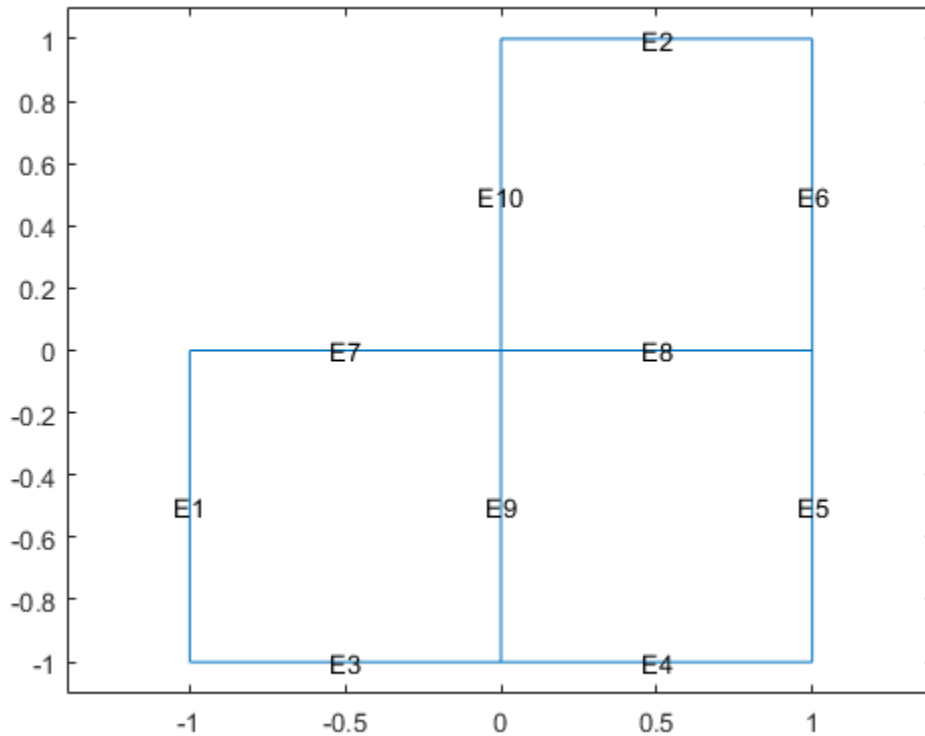


Set zero Dirichlet conditions on all edges.

```
applyBoundaryCondition(model,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
```

Poisson's equation is

$$-\nabla \cdot \nabla u = 1.$$

**6-625**

Toolbox solvers address equations of the form

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

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)
```
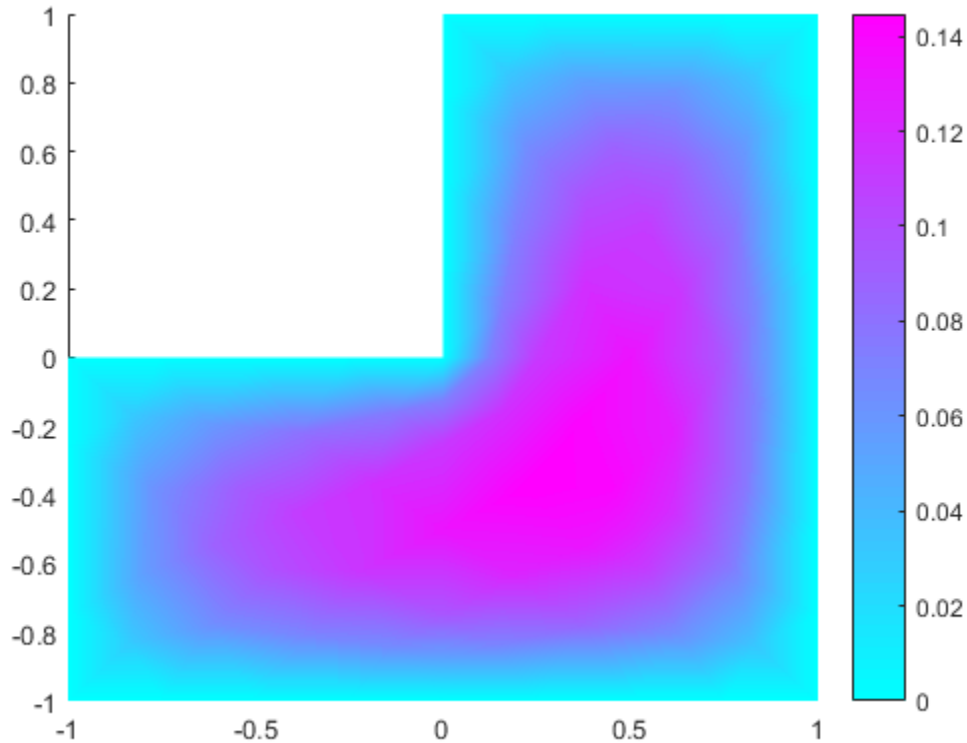
### Solve a Time-Dependent Parabolic Equation with Nonconstant Coefficients

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,'neumann','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))
```
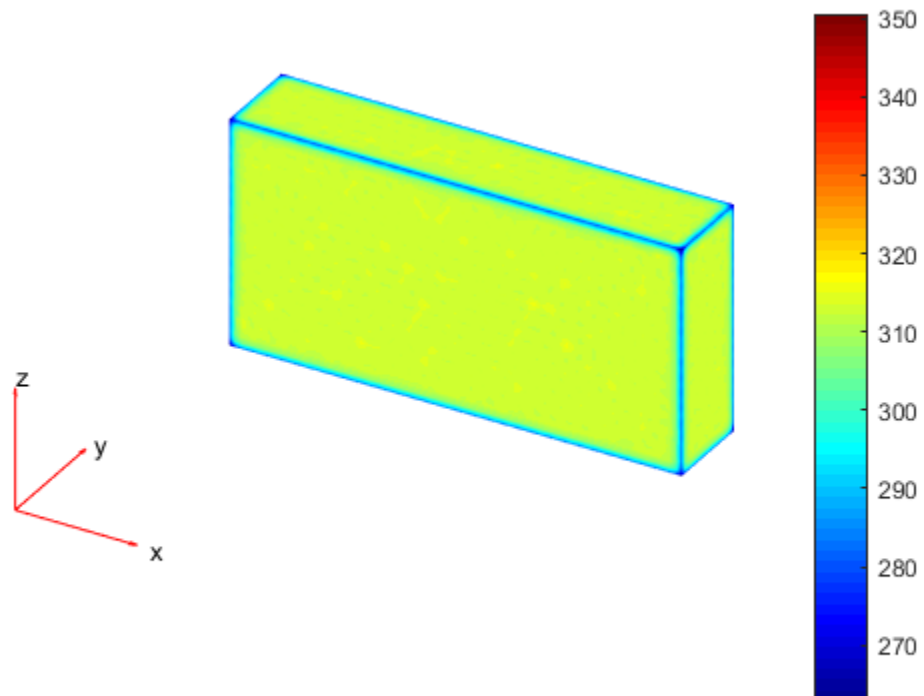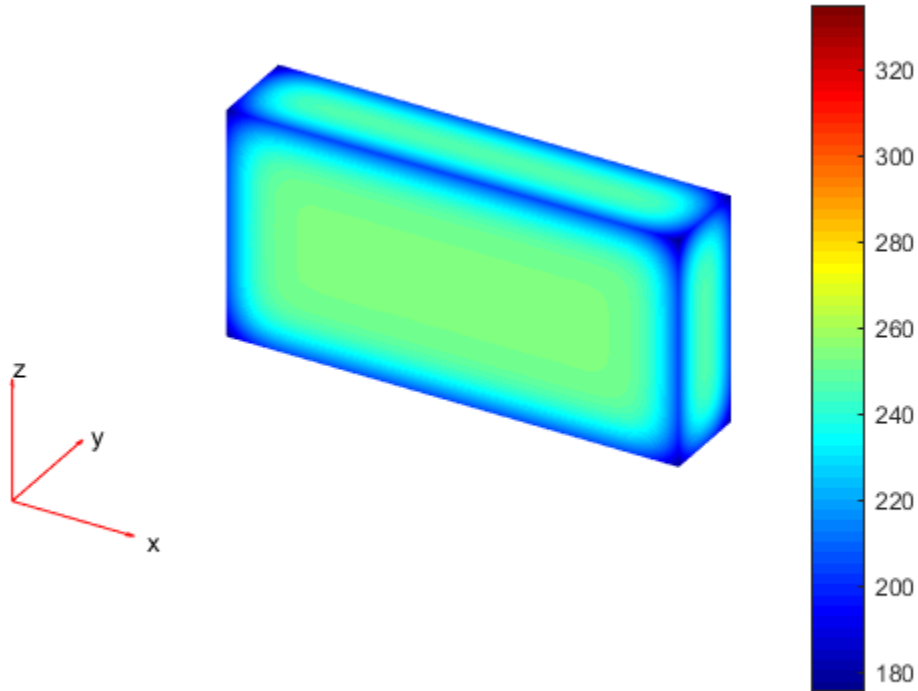
- "Partial Differential Equation Toolbox Examples"
- "Solve Problems Using PDEModel Objects" on page 2-5

## 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

## See Also
`applyBoundaryCondition` | `PDEModel` | `setInitialConditions` | `solvepdeeig` | `specifyCoefficients`

## Topics
"Partial Differential Equation Toolbox Examples"
"Solve Problems Using PDEModel Objects" on page 2-5

**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');
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
```
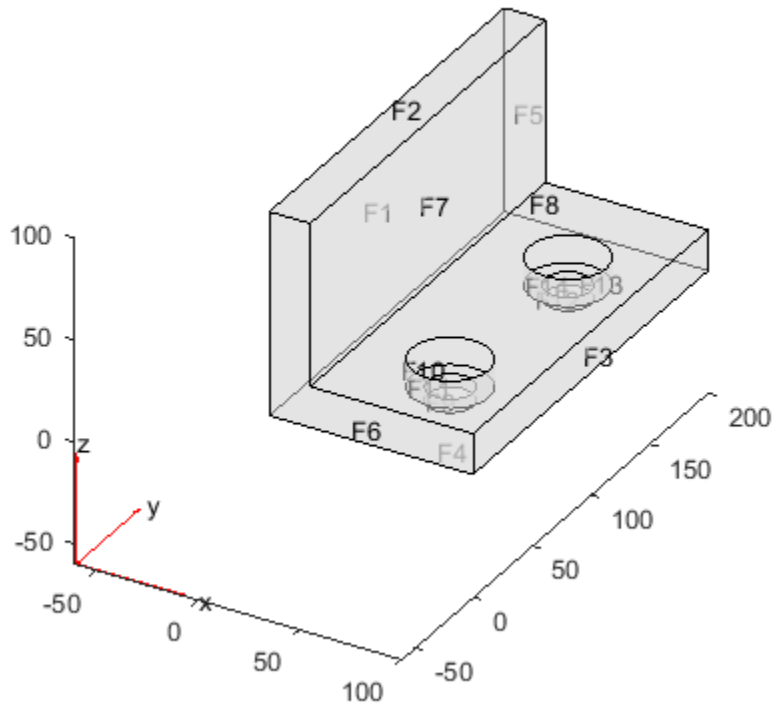
Set F1, the rear face, to have zero deflection.

```
applyBoundaryCondition(model,'dirichlet','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.

```
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);
```

```
                 Basis= 10,  Time=  66.60,  New conv eig=  0
                 Basis= 11,  Time=  66.77,  New conv eig=  0
                 Basis= 12,  Time=  67.03,  New conv eig=  0
                 Basis= 13,  Time=  67.30,  New conv eig=  0
                 Basis= 14,  Time=  67.66,  New conv eig=  0
                 Basis= 15,  Time=  67.94,  New conv eig=  2
                 Basis= 16,  Time=  68.17,  New conv eig=  2
                 Basis= 17,  Time=  68.50,  New conv eig=  2
                 Basis= 18,  Time=  68.77,  New conv eig=  3
                 Basis= 19,  Time=  68.94,  New conv eig=  5
End of sweep: Basis= 19,  Time=  68.94,  New conv eig=  5
                 Basis= 15,  Time=  71.31,  New conv eig=  0
                 Basis= 16,  Time=  71.48,  New conv eig=  0
End of sweep: Basis= 16,  Time=  71.48,  New conv eig=  0
```

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')
```
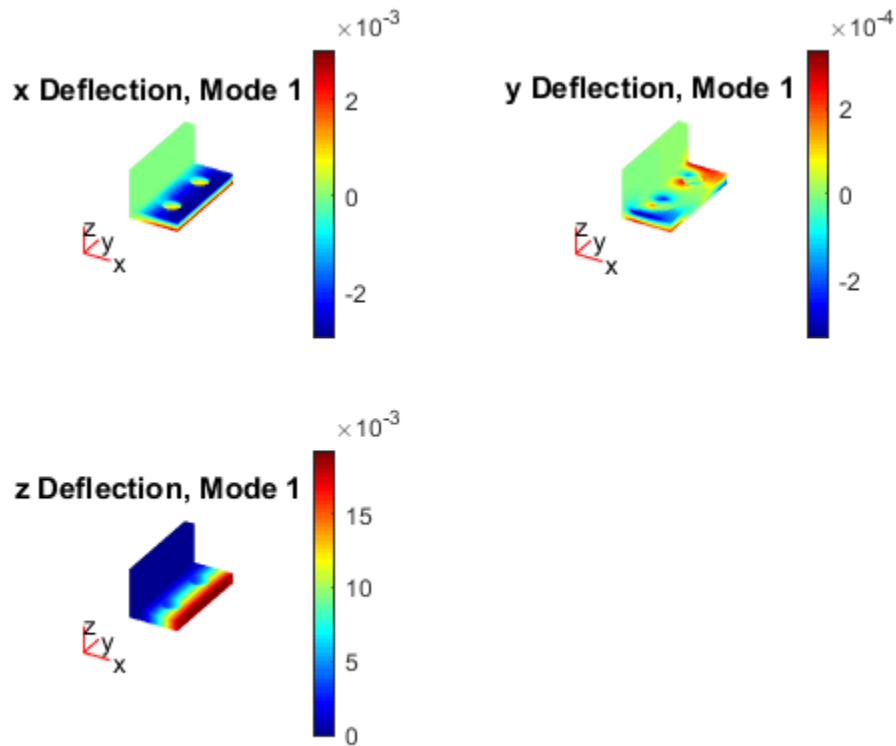


Plot the solution for the highest eigenvalue.

```
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')
```
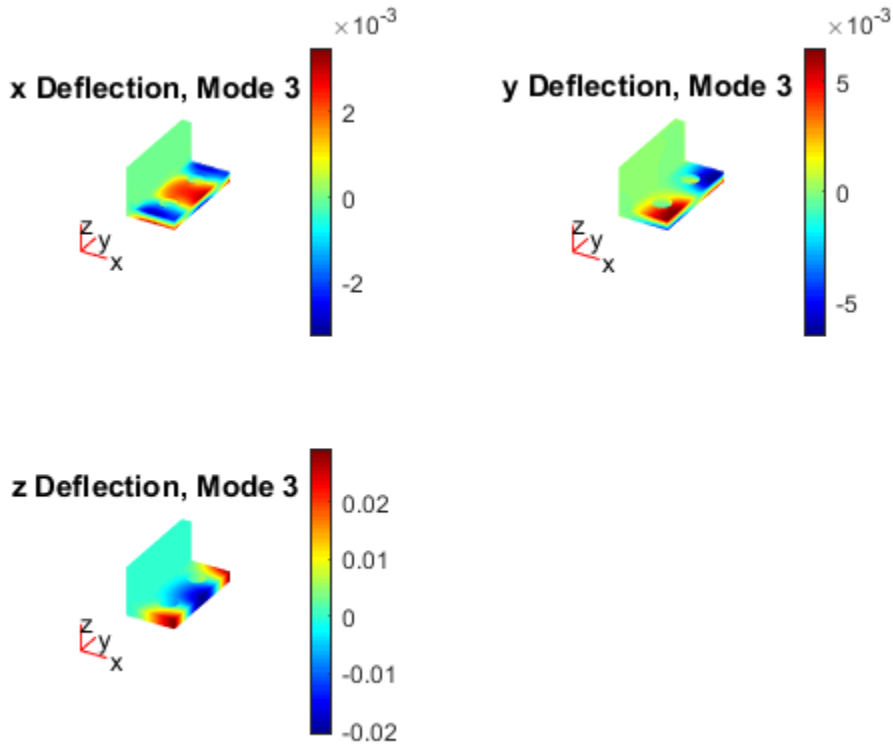
x Deflection, Mode 3

y Deflection, Mode 3

z Deflection, Mode 3

- • "Eigenvalues and Eigenmodes of the L-Shaped Membrane" on page 3-168
- • "Eigenvalues and Eigenmodes of a Square" on page 3-179
- • "Solve Problems Using PDEModel Objects" on page 2-5

## 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.

## Tips

- The equation coefficients cannot depend on the solution `u` or its gradient.

## See Also

### See Also
`applyBoundaryCondition` | `PDEModel` | `solvepde` | `specifyCoefficients`

### Topics
"Eigenvalues and Eigenmodes of the L-Shaped Membrane" on page 3-168
"Eigenvalues and Eigenmodes of a Square" on page 3-179
"Solve Problems Using PDEModel Objects" on page 2-5

**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.

## Syntax

```
specifyCoefficients(model,Name,Value)
specifyCoefficients(model,Name,Value,RegionType,RegionID)
CA = specifyCoefficients( ___ )
```

## 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`. This syntax applies coefficients to the entire geometry.

---

**Note:** Include geometry in `model` before using `specifyCoefficients`.

---

`specifyCoefficients(model,Name,Value,RegionType,RegionID)` assigns coefficients for a specified geometry region.

`CA = specifyCoefficients( ___ )` 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,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
generateMesh(model,'Hmax',0.25);
results = solvepde(model);
```

View the solution.

```
pdeplot(model,'XYData',results.NodalSolution)
```
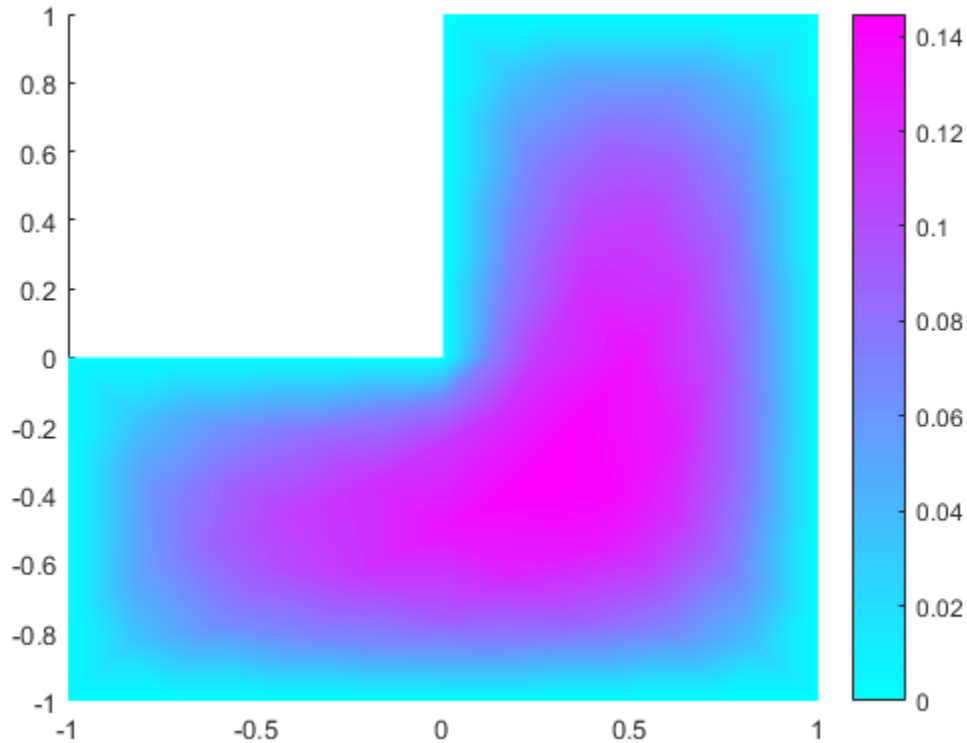
### Coefficient Handle for Nonconstant Coefficients

Specify coefficients for Poisson's equation in 3-D with a nonconstant source term, and obtain the coefficient object.

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,'dirichlet','Face',1,'u',0);
generateMesh(model);
results = solvepde(model);
```

View the solution on the surface.

```
pdeplot3D(model,'ColorMapData',results.NodalSolution)
```
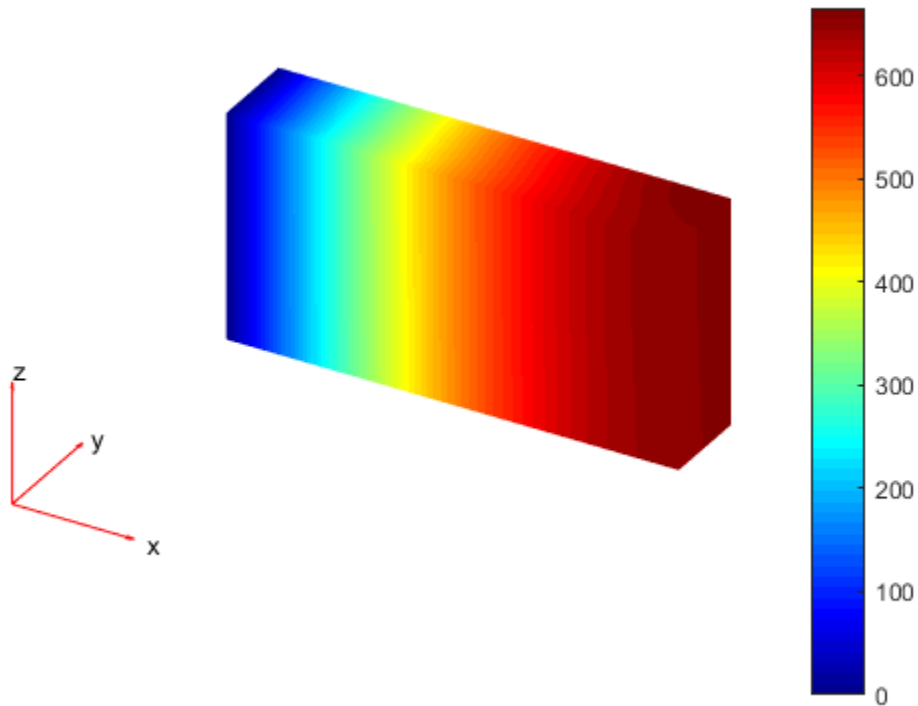
**Specify Coefficients Depending On Subdomain**

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,'FaceLabels','on')
axis equal
ylim([-1.1,1.1])
```
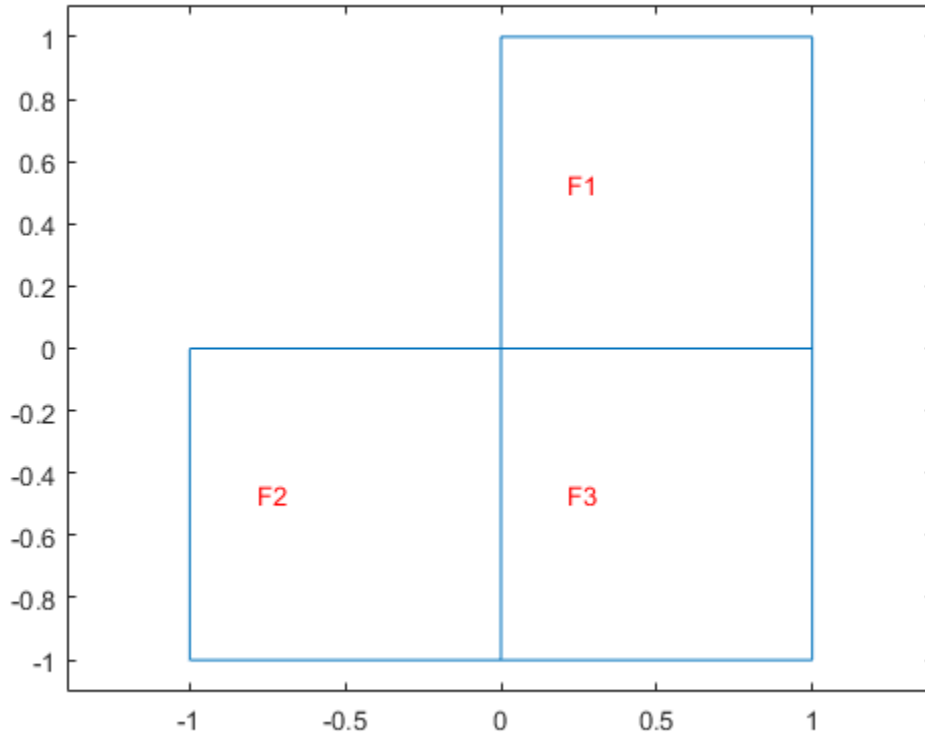
Set the `c` coefficient to 1 in all domains, but the `f` coefficient to 1 in subdomain 1, 5 in subdomain 2, and -8 in subdomain 3. Set all other coefficients to 0.

```
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,'dirichlet','Edge',1:model.Geometry.NumEdges,'u',0);
generateMesh(model,'Hmax',0.25);
results = solvepde(model);
pdeplot(model,'XYData',results.NodalSolution)
```
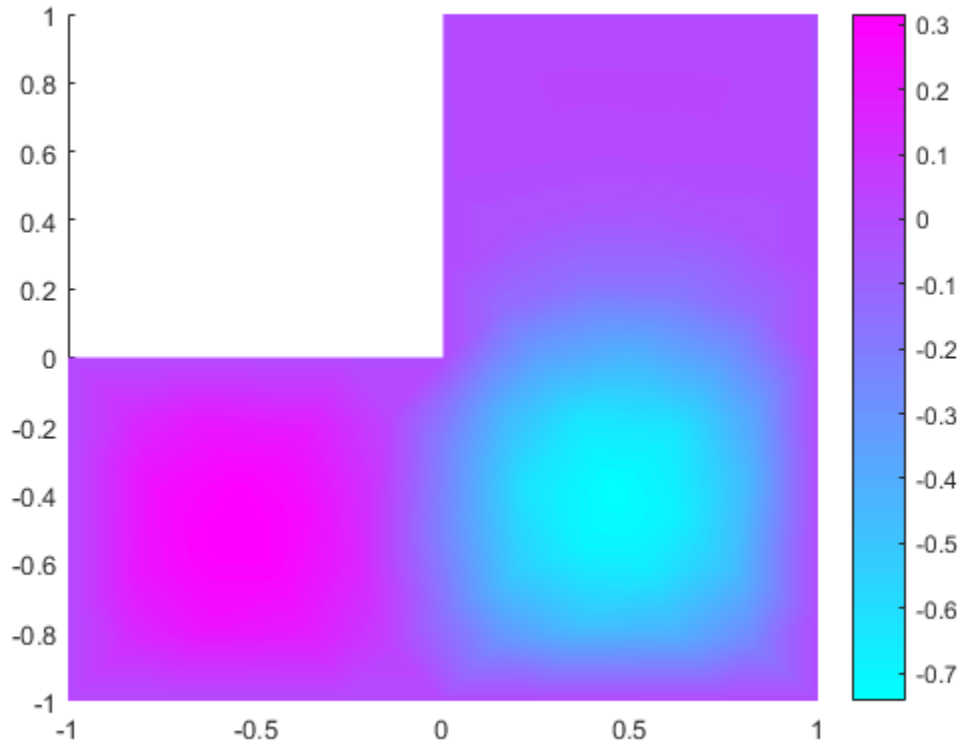
- "PDE Coefficients"
- "Solve Problems Using PDEModel Objects" on page 2-5

## 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 2-128.

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 2-128.

---

**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 6-647.

---

Specify `0` if the term is not part of your problem.

Example:
```
specifyCoefficients('m',O,'d',@dcoef,'c',1,'a',O,'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 2-89.

Example:
```
specifyCoefficients('m',O,'d',O,'c',@ccoef,'a',O,'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 2-128.

Specify `0` if the term is not part of your problem.

Example:
```
specifyCoefficients('m',O,'d',O,'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 2-86.

Specify `0` if the term is not part of your problem.

Example:
```
specifyCoefficients('m',O,'d',O,'c',1,'a',O,'f',@fcoeff,'Face',1:4)
```

Data Types: `double | function_handle`
Complex Number Support: Yes

### `'RegionType'` — Geometric region type

`'Face'` for a 2-D model | `'Cell'` for a 3-D model

Geometric region type, specified as `'Face'` or `'Cell'`.

Example: `specifyCoefficients('m',0,'d',0,'c',1,'a',0,'f',10,'Cell',2)`

Data Types: `char`

### `'RegionID'` — Geometric region ID

vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot`, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Example:
`specifyCoefficients('m',0,'d',0,'c',1,'a',0,'f',10,'Cell',1:3)`

Data Types: `double`

# Output Arguments

### CA — Coefficient assignment

`CoefficientAssignment` object

Coefficient assignment, returned as a `CoefficientAssignment` object.

# Definitions

## 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

### See Also
findCoefficients | PDEModel

### Topics
"PDE Coefficients"
"Solve Problems Using PDEModel Objects" on page 2-5

**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.

## 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

**Introduced before R2006a**

# StationaryResults

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`.

## 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`.

# 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, Gradients, and Fluxes" on page 3-232.

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, Gradients, and Fluxes" on page 3-232.

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, Gradients, and Fluxes" on page 3-232.

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, Gradients, and Fluxes" on page 3-232.

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 |

## 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
pdegplot(model,'FaceLabels','on')
view(30,30)
title('Bracket with Face Labels')
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```

**Bracket with Face Labels**
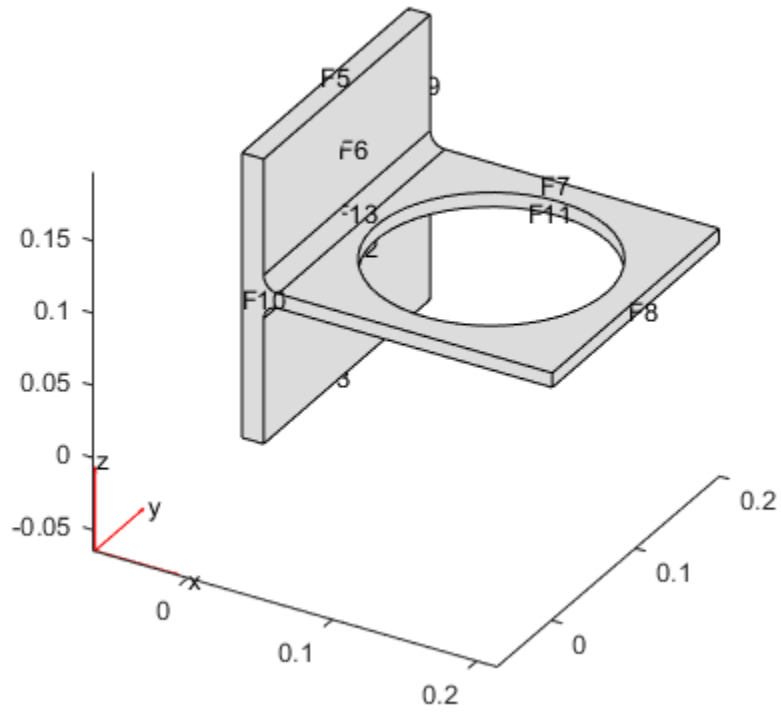
**Bracket with Face Labels, Rear View**
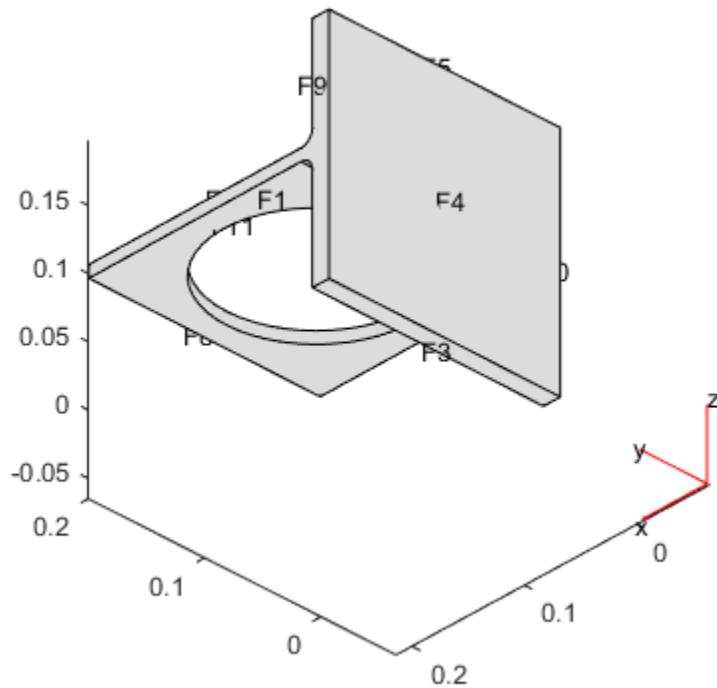


Set boundary conditions such that face 4 is immobile, and face 8 has a force in the negative z direction.

```
applyBoundaryCondition(model,'dirichlet','Face',4,'u',[0,0,0]);
applyBoundaryCondition(model,'neumann','Face',8,'g',[0,0,-1e4]);
```

Set coefficients that represent the equations of linear elasticity. See 3-D Linear Elasticity Equations in Toolbox Form.

```
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: [14036×3 double]
        XGradients: [14036×3 double]
        YGradients: [14036×3 double]
        ZGradients: [14036×3 double]
             Mesh: [1×1 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))
```
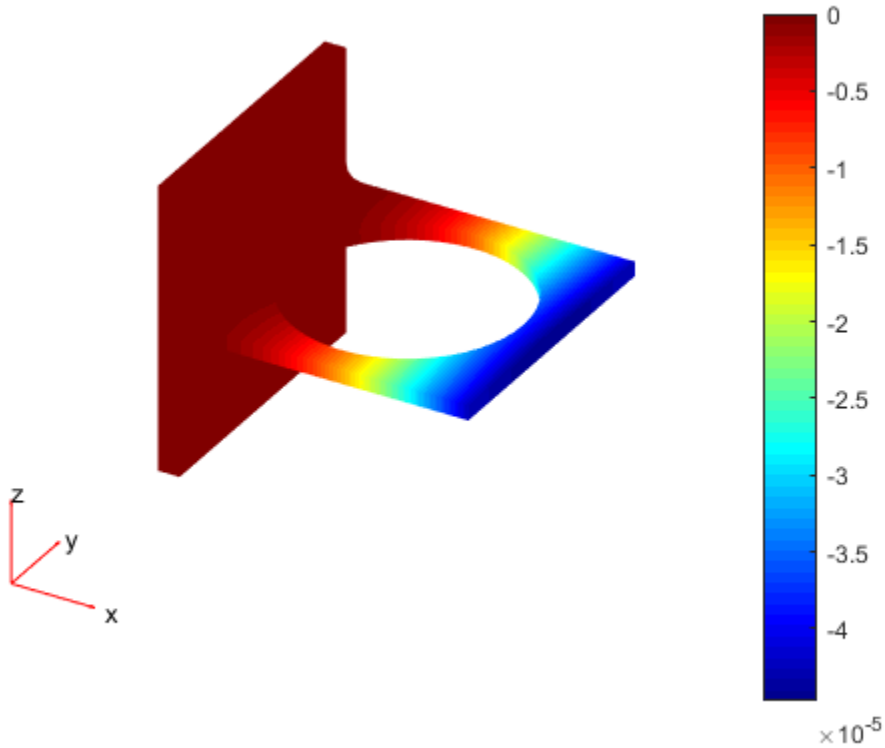
### Results from createPDEResults

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
pdegplot(model,'FaceLabels','on')
view(30,30)
```

```
title('Bracket with Face Labels')
figure
pdegplot(model,'FaceLabels','on')
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```

**Bracket with Face Labels**
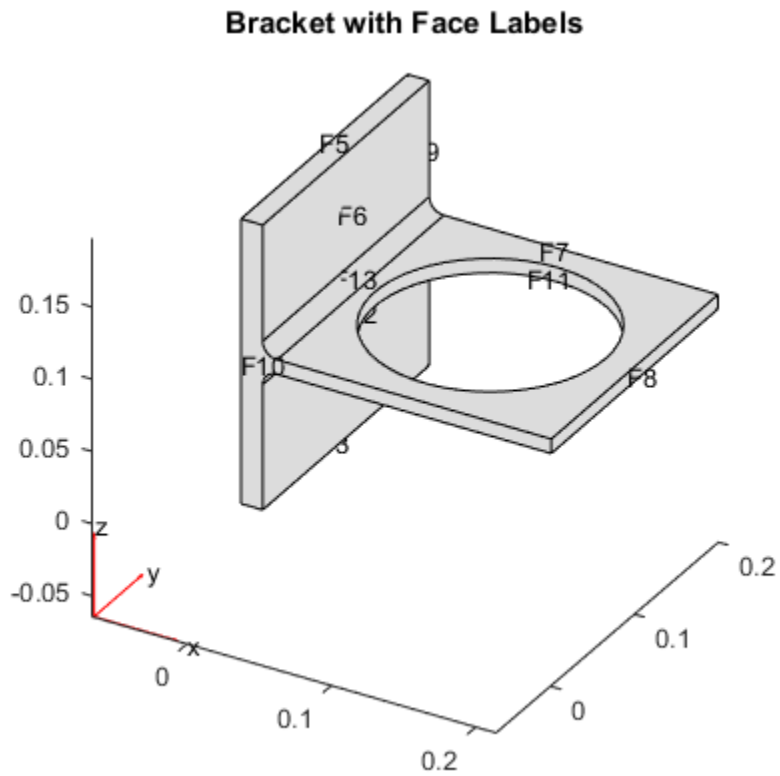
**Bracket with Face Labels, Rear View**
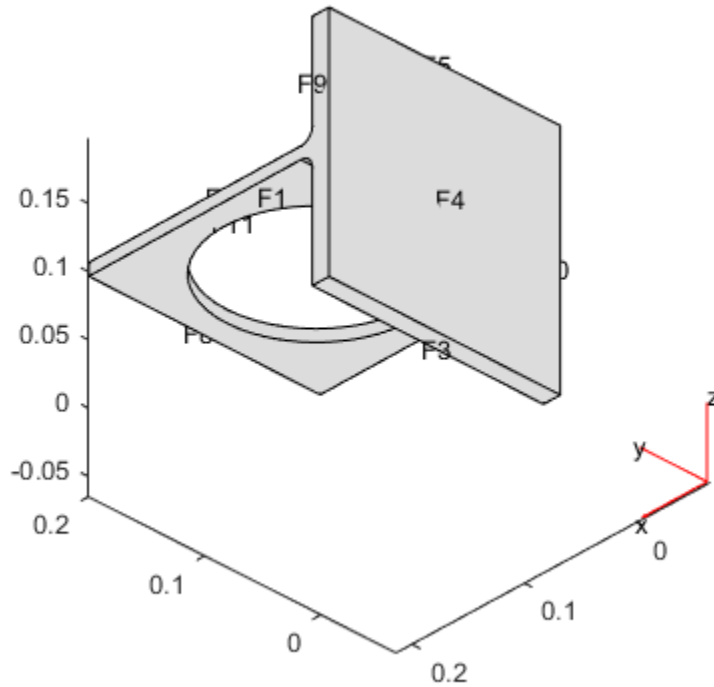


Set boundary conditions such that F4 is immobile, and F8 has a force in the negative z direction.

```
applyBoundaryCondition(model,'dirichlet','Face',4,'u',[0,0,0]);
applyBoundaryCondition(model,'neumann','Face',8,'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.

```
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: [14036×3 double]
        XGradients: [14036×3 double]
        YGradients: [14036×3 double]
        ZGradients: [14036×3 double]
              Mesh: [1×1 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))
```
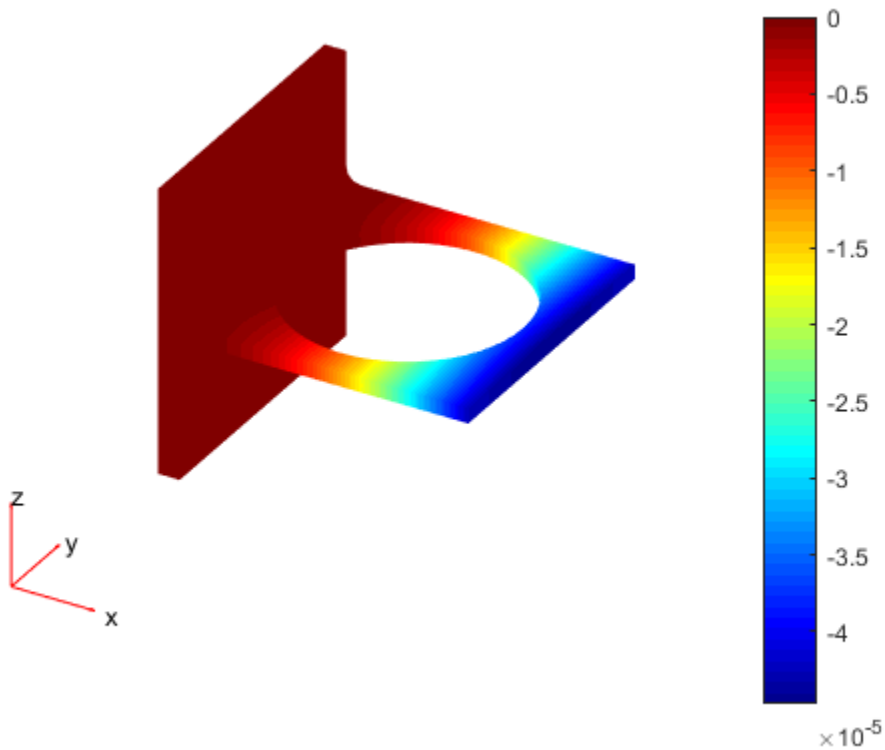
- "Solve Poisson's Equation on a Unit Disk" on page 3-120
- "Minimal Surface Problem on the Unit Disk"
- "Solve Problems Using PDEModel Objects" on page 2-5

## See Also

### See Also
EigenResults | evaluateGradient | interpolateSolution | solvepde |
TimeDependentResults

## Topics

**Introduced in R2016a**

# SteadyStateThermalResults

Contain steady-state thermal solution and derived quantities

## Description

A `SteadyStateThermalResults` object contains the temperature and temperature gradient values in a form convenient for plotting and postprocessing.

The temperature and its gradients are calculated at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. Temperature values at the nodes appear in the `Temperature` property. The three components of the temperature gradient at the nodes appear in the `XGradients`, `YGradients`, and `ZGradients` properties.

To interpolate the temperature or its gradients to a custom grid (for example, specified by `meshgrid`), use `interpolateTemperature` or `evaluateTemperatureGradient`.

To evaluate heat flux of a thermal solution at nodal or arbitrary spatial locations, use `evaluateHeatFlux`. To evaluate integrated heat flow rate normal to specified boundary, use `evaluateHeatRate`.

## Create Object

`solve` returns transient solutions of a thermal problem as a `SteadyStateThermalResults` object.

## Properties

**`Mesh` — Finite element mesh**
`FEMesh` object

Finite element mesh, returned as a `FEMesh` object.

**`Temperature` — Temperature values at nodes**
vector

Temperature values at nodes, returned as a vector.

Data Types: `double`

**XGradients — *x*-component of temperature gradient at nodes**
vector

*x*-component of the temperature gradient at nodes, returned as a vector.

Data Types: `double`

**YGradients — *y*-component of temperature gradient at nodes**
vector

*y*-component of the temperature gradient at nodes, returned as a vector.

Data Types: `double`

**ZGradients — *z*-component of temperature gradient at nodes**
vector

*z*-component of the temperature gradient at nodes, returned as a vector.

Data Types: `double`

## Object Functions

| | |
|---|---|
| evaluateHeatFlux | Evaluate heat flux of a thermal solution at nodal or arbitrary spatial locations |
| evaluateHeatRate | Evaluate integrated heat flow rate normal to specified boundary |
| evaluateTemperatureGradient | Evaluate temperature gradient of a thermal solution at arbitrary spatial locations |
| interpolateTemperature | Interpolate temperature in a thermal result at arbitrary spatial locations |

## Examples

**Solution to Steady-State Thermal Model**

Solve a 3-D steady-state thermal problem.

Create a thermal model for this problem.

```
thermalmodel = createpde('thermal');
```

Import and plot the block geometry.

```matlab
importGeometry(thermalmodel,'Block.stl');
pdegplot(thermalmodel,'FaceLabel','on','FaceAlpha',0.5)
axis equal
```



Assign material properties.

```matlab
thermalProperties(thermalmodel,'ThermalConductivity',80);
```

Apply a constant temperature of $100\,^\circ C$ to the left side of the block (face 1) and a constant temperature of $300\,^\circ C$ at the right side of the block (face 3). All other faces are insulated by default.

```matlab
thermalBC(thermalmodel,'Face',1,'Temperature',100);
thermalBC(thermalmodel,'Face',3,'Temperature',300);
```

Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)


thermalresults =

  SteadyStateThermalResults with properties:

    Temperature: [28811×1 double]
     XGradients: [28811×1 double]
     YGradients: [28811×1 double]
     ZGradients: [28811×1 double]
           Mesh: [1×1 FEMesh]
```

The solver finds the values of temperatures and temperature gradients at the nodal locations. To access these values, use `thermalresults.Temperature`, `thermalresults.XGradients`, and so on. For example, plot temperatures at nodal locations.

```
pdeplot3D(thermalmodel,'ColorMapData',thermalresults.Temperature)
```

## See Also

### See Also
evaluateHeatFlux | evaluateHeatRate | evaluateTemperatureGradient |
interpolateTemperature | TransientThermalResults

**Introduced in R2017a**

# ThermalBC Properties

Boundary condition for thermal model

## Description

A `ThermalBC` object specifies the type of PDE boundary condition on a set of geometry boundaries. A `ThermalModel` object contains a vector of `ThermalBC` objects in its `BoundaryConditions.ThermalBCAssignments` property.

Specify boundary conditions for your model using the `thermalBC` function.

## Properties

### `RegionType` — Geometric region type
`'Face'` for 3-D geometry | `'Edge'` for 2-D geometry

Geometric region type, returned as `'Face'` for 3-D geometry or `'Edge'` for 2-D geometry.

Data Types: `char`

### `RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, returned as a vector of positive integers. Find the region IDs using `pdegplot` with the `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) value set to `'on'`.

Data Types: `double`

### `Temperature` — Temperature boundary condition
number | function handle

Temperature boundary condition, returned as a number or a function handle. Use a function handle to specify spatially or temporally varying temperature.

Data Types: `double` | `function_handle`

### `HeatFlux` — Heat flux boundary condition
number | function handle

Heat flux boundary condition, returned as a number or a function handle. Use a function handle to specify a spatially or temporally varying heat flux or a nonlinear heat flux.

Data Types: `double | function_handle`

**`ConvectionCoefficient` — Coefficient for convection to ambient heat transfer condition**
number | function handle

Convection to ambient boundary condition, returned as a number or a function handle. Use a function handle to specify a spatially or temporally varying convection coefficient or a nonlinear convection coefficient. Specify ambient temperature using the `AmbientTemperature` argument.

Data Types: `double | function_handle`

**`Emissivity` — Radiation emissivity coefficient**
number in the range (0,1)

Radiation emissivity coefficient, returned as a number in the range (0,1). Use a function handle to specify spatially or temporally varying emissivity or nonlinear emissivity. Specify ambient temperature using the `AmbientTemperature` argument and the Stefan-Boltzmann constant using the thermal model properties.

Data Types: `double | function_handle`

**`AmbientTemperature` — Ambient temperature**
number

Ambient temperature, returned as a number. The ambient temperature value is required for specifying convection and radiation boundary conditions.

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" (MATLAB). For details of this evaluation, see "Nonconstant Boundary Conditions" on page 2-165.

Data Types: `char`

# See Also

## See Also
findthermalBC | thermalBC | ThermalModel

**Introduced in R2017a**

# ThermalMaterialAssignment Properties

Material properties assignments

## Description

A `ThermalMaterialAssignment` object contains the description of a thermal model's material properties. A `ThermalModel` container has a vector of `ThermalMaterialAssignment` objects in its `MaterialProperties.MaterialAssignments` property.

Create material properties assignments for your thermal model using the `thermalProperties` 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 `'FaceLabels'` name-value pair to `'on'`.

Data Types: `double`

### `ThermalConductivity` — Thermal conductivity of the material
nonnegative number | function handle

Thermal conductivity of the material, returned as a nonnegative number or a function handle.

Data Types: `double` | `function_handle`

**MassDensity — Mass density of the material**
nonnegative number | function handle

Mass density of the material, returned as a nonnegative number or a function handle.

Data Types: double | function_handle

**SpecificHeat — Specific heat of the material**
nonnegative number | function handle

Specific heat of the material, returned as a nonnegative number or a function handle.

Data Types: double | function_handle

# See Also

## See Also
findThermalProperties | thermalProperties

**Introduced in R2017a**

# ThermalModel

Thermal model container

## Description

A `ThermalModel` object contains information about a heat transfer problem: the geometry, material properties, internal heat sources, temperature on the boundaries, heat fluxes through the boundaries, mesh, and initial conditions.

## Create Object

`createpde` with the first argument `'thermal'` returns a `ThermalModel` container.

## Properties

### `AnalysisType` — Type of thermal analysis
`'steadystate'` | `'transient'`

Type of thermal analysis, returned as `'steadystate'` or `'transient'`.

### `Geometry` — Geometry description
geometry object

Geometry description, returned as a geometry object.

- `AnalyticGeometry` object for 2-D geometry. Create this geometry using the `geometryFromEdges` function.
- `DiscreteGeometry` object for 3-D geometry. Create this geometry using the `importGeometry` function or the `geometryFromMesh` function.

### `MaterialProperties` — Material properties within the domain
object containing material property assignments

Material properties within the domain, returned as an object containing the material property assignments.

### `HeatSources` — Heat source within the domain or subdomain
object containing heat source assignments

Heat source within the domain or subdomain, returned as an object containing heat source assignments.

### `BoundaryConditions` — Boundary conditions applied to the geometry
object containing boundary condition assignments

Boundary conditions applied to the geometry, returned as an object containing the boundary condition assignments.

### `InitialConditions` — Initial temperature or initial guess
object containing the initial temperature assignments within the geometric domain

Initial temperature or initial guess, returned as an object containing the initial temperature assignments within the geometric domain.

### `Mesh` — Mesh for solution
`FEMesh` object

Mesh for solution, returned as a `FEMesh` object. You create the mesh using the `generateMesh` function.

### `StefanBoltzmannConstant` — Constant of proportionality in Stefan-Boltzmann law governing radiation heat transfer
number

Constant of proportionality in Stefan-Boltzmann law governing radiation heat transfer, specified as a number. This value must be consistent with the units of the model. Values of the Stefan-Boltzmann constant in commonly used system of units are:

- SI – 5.670367e-8 W/(m$^2 \cdot$K$^4$)
- CGS – 5.6704e-5 erg/(cm$^2 \cdot$s$\cdot$K$^4$)
- US customary – 1.714e-9 BTU/(hr ft$^2 \cdot$R$^4$)

### `SolverOptions` — Algorithm options for PDE solvers
`PDESolverOptions` object

Algorithm options for the PDE solvers, returned as a `PDESolverOptions` object. The properties of `PDESolverOptions` include absolute and relative tolerances for internal ODE solvers, maximum solver iterations, and so on.

## Object Functions

| | |
|---|---|
| geometryFromEdges | Create 2-D geometry |
| geometryFromMesh | Create 3-D geometry from a triangulated mesh |
| importGeometry | Import 3-D geometry |
| thermalProperties | Assign thermal properties of a material for a thermal model |
| internalHeatSource | Specify internal heat source for a thermal model |
| thermalBC | Specify boundary conditions for a thermal model |
| thermalIC | Set initial conditions or initial guess for a thermal model |
| generateMesh | Create triangular or tetrahedral mesh |
| solve | Solve heat transfer problem specified in a ThermalModel |

## Examples

### Create and Populate a Thermal Model

Create a transient thermal model container.

```
thermalmodel = createpde('thermal','transient')


thermalmodel =

  ThermalModel with properties:

              AnalysisType: 'transient'
                  Geometry: []
        MaterialProperties: []
               HeatSources: []
    StefanBoltzmannConstant: []
        BoundaryConditions: []
         InitialConditions: []
                      Mesh: []
             SolverOptions: [1×1 PDESolverOptions]
```

Create the geometry and include it in the model.

```
g = @squareg;
geometryFromEdges(thermalmodel,g)


ans =

  AnalyticGeometry with properties:

        NumCells: 0
        NumFaces: 1
        NumEdges: 4
     NumVertices: 4
```

Assign material properties.

```
thermalProperties(thermalmodel,'ThermalConductivity',79.5,...
                               'MassDensity',7850,...
                               'SpecificHeat',450,...
                               'Face',1)


ans =

  ThermalMaterialAssignment with properties:

              RegionType: 'face'
                RegionID: 1
    ThermalConductivity: 79.5000
             MassDensity: 7850
            SpecificHeat: 450
```

Specify that the entire geometry generates heat at the rate 25 W/m^3.

```
internalHeatSource(thermalmodel,25)


ans =

  HeatSourceAssignment with properties:

    RegionType: 'face'
      RegionID: 1
```

```
     HeatSource: 25
```

Apply insulated boundary conditions on three edges and the free convection boundary condition on the right edge.

```
thermalBC(thermalmodel,'Edge',[1,3,4],'HeatFlux',0);
thermalBC(thermalmodel,'Edge',2,...
                        'ConvectionCoefficient',5000,...
                        'AmbientTemperature',25)
```

```
ans =

  ThermalBC with properties:

                RegionType: 'Edge'
                  RegionID: 2
               Temperature: []
                  HeatFlux: []
     ConvectionCoefficient: 5000
                Emissivity: []
        AmbientTemperature: 25
                Vectorized: 'off'
```

Set the initial conditions: uniform room temperature across domain and higher temperature on the left edge.

```
thermalIC(thermalmodel,25);
thermalIC(thermalmodel,100,'Edge',4)
```

```
ans =

  GeometricThermalICs with properties:

             RegionType: 'edge'
               RegionID: 4
      InitialTemperature: 100
```

Specify the Stefan-Boltzmann constant.

```
thermalmodel.StefanBoltzmannConstant = 5.670367e-8;
```

Generate mesh.

```
generateMesh(thermalmodel)


ans =

  FEMesh with properties:

              Nodes: [2×177 double]
           Elements: [3×312 double]
      MaxElementSize: 0.2358
      MinElementSize: 0.1040
      GeometricOrder: 'linear'
```

`thermalmodel` now contains the following properties.

```
thermalmodel


thermalmodel =

  ThermalModel with properties:

                  AnalysisType: 'transient'
                      Geometry: [1×1 AnalyticGeometry]
            MaterialProperties: [1×1 MaterialAssignmentRecords]
                   HeatSources: [1×1 HeatSourceAssignmentRecords]
       StefanBoltzmannConstant: 5.6704e-08
            BoundaryConditions: [1×1 ThermalBCRecords]
             InitialConditions: [1×1 ThermalICRecords]
                          Mesh: [1×1 FEMesh]
                 SolverOptions: [1×1 PDESolverOptions]
```

# See Also

## See Also
createpde | generateMesh | geometryFromEdges | geometryFromMesh |
importGeometry | internalHeatSource | pdegplot | pdeplot | pdeplot3D |
thermalBC | thermalIC | thermalProperties

**Introduced in R2017a**

# thermalProperties

Assign thermal properties of a material for a thermal model

## Syntax

```
thermalProperties(thermalmodel,Name,Value)
thermalProperties(thermalmodel,Name,Value,RegionType,RegionID)
mtl = thermalProperties( ___ )
```

## Description

`thermalProperties(thermalmodel,Name,Value)` assigns material properties, such as thermal conductivity, mass density, and specific heat. This syntax sets material properties for the entire geometry.

`thermalProperties(thermalmodel,Name,Value,RegionType,RegionID)` assigns material properties for a specified geometry region.

`mtl = thermalProperties( ___ )` returns the material properties object.

## Examples

### Assign Thermal Conductivity

Assign material properties for a steady-state thermal model.

```
model = createpde('thermal','steadystate');
gm = importGeometry(model,'SquareBeam.STL');
thermalProperties(model,'ThermalConductivity',0.08)
```

```
ans =

  ThermalMaterialAssignment with properties:
```

```
                RegionType: 'cell'
                  RegionID: 1
       ThermalConductivity: 0.0800
               MassDensity: []
               SpecificHeat: []
```

### Assign Thermal Conductivity, Mass Density, and Specific Heat

Assign material properties for transient analysis.

```
thermalmodel = createpde('thermal','transient');
gm = importGeometry(thermalmodel,'SquareBeam.STL');
thermalProperties(thermalmodel,'ThermalConductivity',0.2,...
                               'MassDensity',2.7*10^(-6),...
                               'SpecificHeat',920)
```

```
ans =

  ThermalMaterialAssignment with properties:

                RegionType: 'cell'
                  RegionID: 1
       ThermalConductivity: 0.2000
               MassDensity: 2.7000e-06
               SpecificHeat: 920
```

### Assign Thermal Conductivities for Each Geometric Region

Create a steady-state thermal model.

```
thermalModel = createpde('thermal');
```

Create nested cylinders to model a two-layered insulated pipe section, consisting of inner metal pipe surrounded by insulated material.

```
gm = multicylinder([20,25,35],20,'Void',[1,0,0]);
```

Assign geometry to the thermal model and plot the geometry.

```
thermalModel.Geometry = gm;
pdegplot(thermalModel,'CellLabels','on','FaceAlpha',0.5)
```
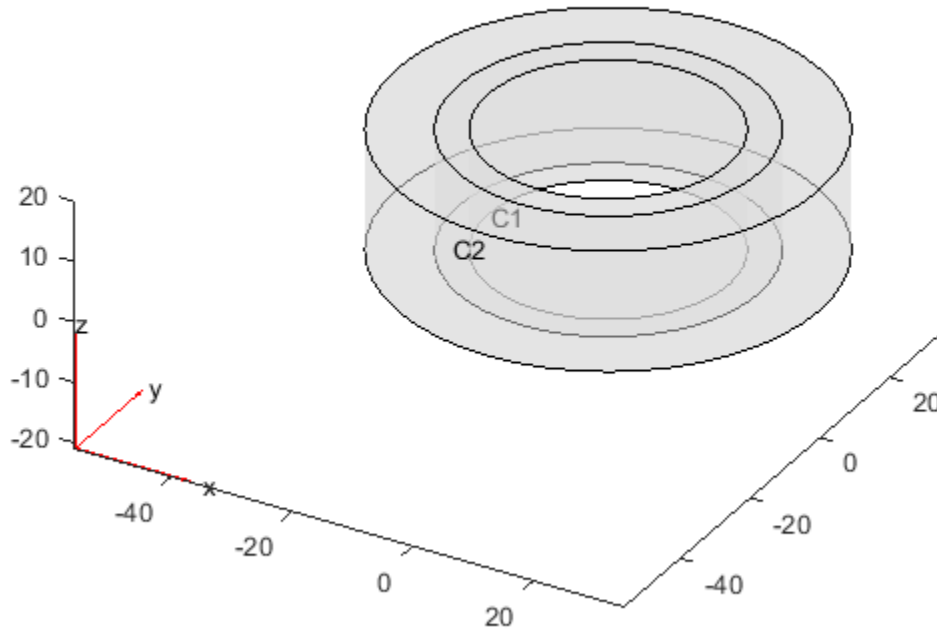
Specify thermal conductivities for metal and insulation.

```
thermalProperties(thermalModel,'Cell',1,'ThermalConductivity',0.4)
thermalProperties(thermalModel,'Cell',2,'ThermalConductivity',0.0015)
```

```
ans =

  ThermalMaterialAssignment with properties:

            RegionType: 'cell'
              RegionID: 1
   ThermalConductivity: 0.4000
           MassDensity: []
```

```
          SpecificHeat: []


ans =

  ThermalMaterialAssignment with properties:

            RegionType: 'cell'
              RegionID: 2
   ThermalConductivity: 0.0015
            MassDensity: []
           SpecificHeat: []
```

* "Heat Conduction in a Spherical Multidomain Geometry with Nonuniform Heat Flux"

## Input Arguments

### **thermalmodel** — Thermal model
ThermalModel object

Thermal model, specified as a `ThermalModel` object. The model contains the geometry, mesh, thermal properties of the material, internal heat source, boundary conditions, and initial conditions.

Example: `thermalmodel = createpde('thermal')`

### **RegionType** — Geometric region type
`'Face'` for a 2-D model | `'Cell'` for a 3-D model

Geometric region type, specified as `'Face'` or `'Cell'`.

Example:
`thermalProperties(thermalmodel,'Cell',1,'ThermalConductivity',100)`

Data Types: char

### **RegionID** — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot`, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Example:
```
thermalProperties(thermalmodel,'Cell',1:3,'ThermalConductivity',100)
```

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:
```
thermalProperties(thermalmodel,'Cell',1,'ThermalConductivity',100,'MassDensity
```

### `'ThermalConductivity'` — Thermal conductivity of the material
positive number | matrix | function handle

Thermal conductivity of the material, specified as a positive number, a matrix, or a function handle. You can specify thermal conductivity for a steady-state or transient model. In case of orthotropic thermal conductivity, use a thermal conductivity matrix.

Example:
```
thermalProperties(thermalmodel,'Cell',1,'ThermalConductivity',100) or
thermalProperties(thermalmodel,'ThermalConductivity',[80;10;80]) for
```
orthotropic thermal conductivity

Data Types: `double` | `function_handle`

### `'MassDensity'` — Mass density of the material
positive number | function handle

Mass density of the material, specified as a positive number or a function handle. Specify this property for a transient thermal conduction analysis model.

Example:
```
thermalProperties(thermalmodel,'Cell',1,'ThermalConductivity',100,'MassDensity
```

Data Types: `double` | `function_handle`

### `'SpecificHeat'` — Specific heat of the material
positive number | function handle

Specific heat of the material, specified as a positive number or a function handle. Specify this property for a transient thermal conduction analysis model.

Example:
`thermalProperties(thermalmodel,'Cell',1,'ThermalConductivity',100,'MassDensity`

Data Types: `double` | `function_handle`

## Output Arguments

**`mtl` — Handle to material properties**
object

Handle to material properties, returned as an object. `mtl` associates material properties with the geometric region.

## See Also

### See Also
`internalHeatSource` | `specifyCoefficients`

### Topics
"Heat Conduction in a Spherical Multidomain Geometry with Nonuniform Heat Flux"

**Introduced in R2017a**

# TimeDependentResults

Contain time-dependent PDE solution and derived quantities

## Compatibility

A `TimeDependentResults` object is used in both the recommended and the legacy workflows. 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`.

## 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.

# 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, Gradients, and Fluxes" on page 3-232.

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, Gradients, and Fluxes" on page 3-232.

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, Gradients, and Fluxes" on page 3-232.

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, Gradients, and Fluxes" on page 3-232.

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

## 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)
rect1 = [3;4;0;2;2;0;0;0;2;2];
% Circle centered at (1.5,0.5)
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','FaceLabels','on')
axis equal
ylim([-0.1,2.1])
```
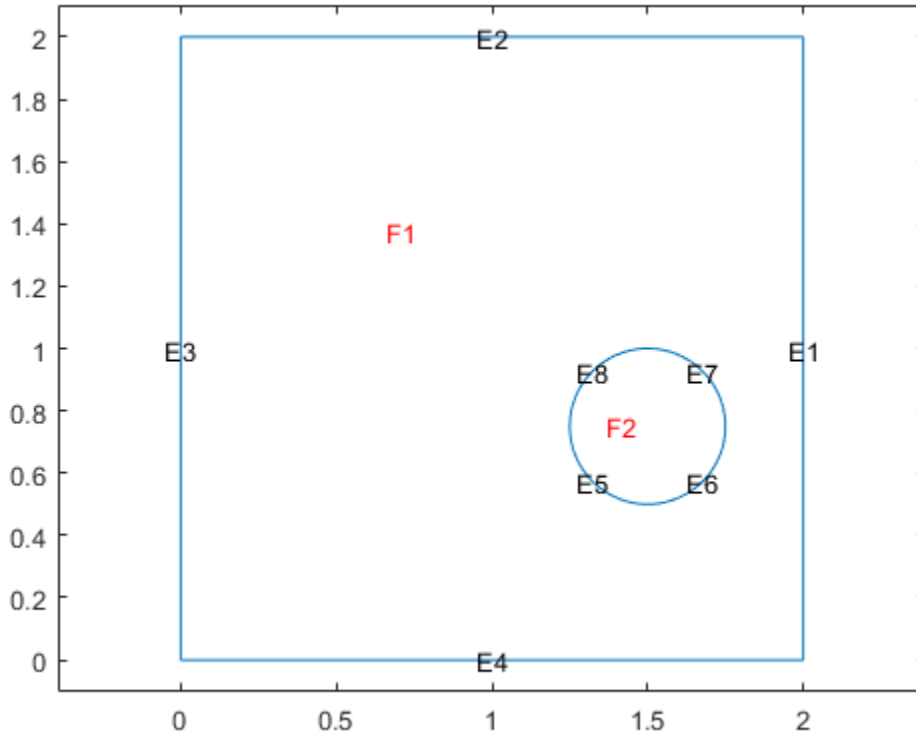
Include the geometry in a PDE model.

```
model = createpde();
geometryFromEdges(model,dl);
```

Set boundary conditions that the upper and left edges are at temperature 10.

```
applyBoundaryCondition(model,'dirichlet','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')
```
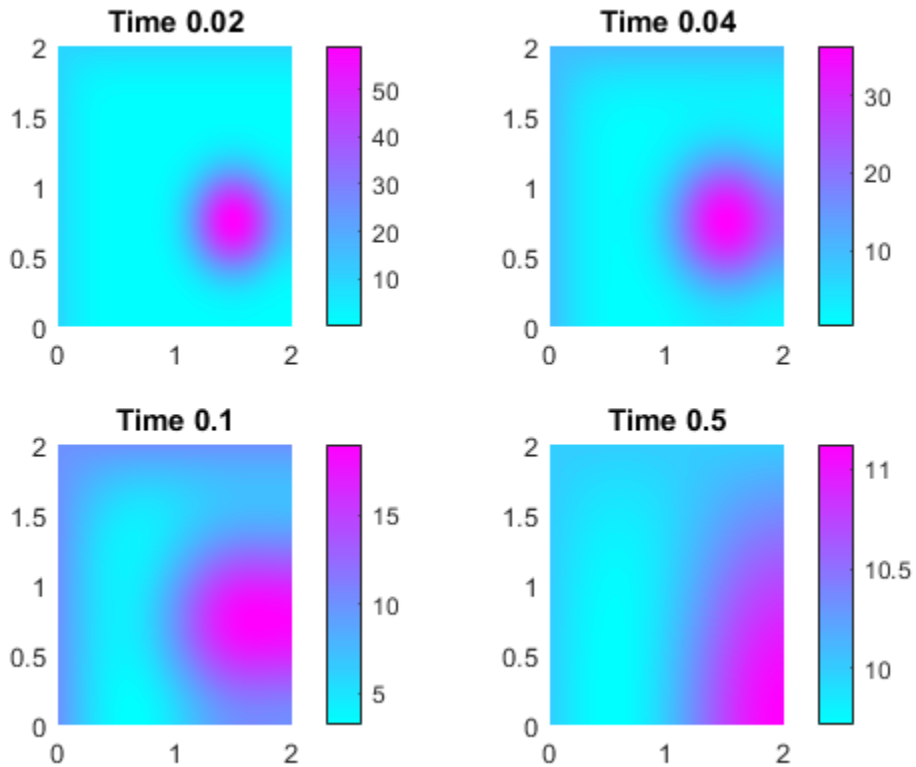
- "Heat Equation for a Block with Cavity" on page 3-144
- "Wave Equation Using Command-Line Functions" on page 3-164
- "Solve Problems Using PDEModel Objects" on page 2-5

# See Also

## See Also
EigenResults | evaluateGradient | interpolateSolution |
StationaryResults

## Topics

**Introduced in R2016a**

# TransientThermalResults

Store transient thermal solution and derived quantities

## Description

A `TransientThermalResults` object contains the temperature and gradients values in a form convenient for plotting and postprocessing.

The temperature and its gradient are calculated at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. Temperature values at the nodes appear in the `Temperature` property. The solution times appear in the `SolutionTimes` property. The three components of the temperature gradient at the nodes appear in the `XGradients`, `YGradients`, and `ZGradients` properties. The array dimensions of `Temperature`, `XGradients`, `YGradients`, and `ZGradients` let you extract solution and gradient values for specified time indices.

To interpolate the temperature or its gradient to a custom grid (for example, specified by `meshgrid`), use `interpolateTemperature` or `evaluateTemperatureGradient`.

To evaluate heat flux of a thermal solution at nodal or arbitrary spatial locations, use `evaluateHeatFlux`. To evaluate integrated heat flow rate normal to specified boundary, use `evaluateHeatRate`.

## Create Object

`solve` returns transient solutions of a thermal problem as a `TransientThermalResults` object.

## Properties

**`Mesh` — Finite element mesh**
FEMesh object

Finite element mesh, returned as a FEMesh object.

**`Temperature` — Temperature values at nodes**
vector | matrix

Temperature values at nodes, returned as a vector or matrix.

### `SolutionTimes` — Solution times
real vector

Solution times, returned as a real vector. `SolutionTimes` is the same as the `tlist` input to `solve`.

Data Types: `double`

### `XGradients` — *x*-component of temperature gradient at nodes
vector | matrix

*x*-component of the temperature gradient at nodes, returned as a vector or matrix.

Data Types: `double`

### `YGradients` — *y*-component of temperature gradient at nodes
vector | matrix

*y*-component of the temperature gradient at nodes, returned as a vector or matrix.

Data Types: `double`

### `ZGradients` — *z*-component of temperature gradient at nodes
vector | matrix

*z*-component of the temperature gradient at nodes, returned as a vector or matrix.

Data Types: `double`

## Object Functions

| | |
|---|---|
| evaluateHeatFlux | Evaluate heat flux of a thermal solution at nodal or arbitrary spatial locations |
| evaluateHeatRate | Evaluate integrated heat flow rate normal to specified boundary |
| evaluateTemperatureGradient | Evaluate temperature gradient of a thermal solution at arbitrary spatial locations |
| interpolateTemperature | Interpolate temperature in a thermal result at arbitrary spatial locations |

# Examples

### Solution to Transient Thermal Model

Solve a 2-D transient thermal problem.

Create a transient thermal model for this problem.

```
thermalmodel = createpde('thermal','transient');
```

Create the geometry and include it in the model.

```
SQ1 = [3; 4; 0; 3; 3; 0; 0; 0; 3; 3];
D1 = [2; 4; 0.5; 1.5; 2.5; 1.5; 1.5; 0.5; 1.5; 2.5];
gd = [SQ1 D1];
sf = 'SQ1+D1';
ns = char('SQ1','D1');
ns = ns';
dl = decsg(gd,sf,ns);
geometryFromEdges(thermalmodel,dl);
pdegplot(thermalmodel,'EdgeLabels','on','FaceLabels','on')
xlim([-1.5 4.5])
ylim([-0.5 3.5])
axis equal
```

For the square region, assign the following thermal properties:

- Thermal conductivity is $10\,W/(m°C)$

- Mass density is $2\,kg/m^3$

- Specific heat is $0.1\,J/(kg°C)$

```
thermalProperties(thermalmodel,'ThermalConductivity',10, ...
                               'MassDensity',2, ...
                               'SpecificHeat',0.1, ...
                               'Face',1);
```

For the diamond region, assign the following thermal properties:

- Thermal conductivity is $2\,W/(m\,{}^\circ C)$

- Mass density is $1\,kg/m^3$

- Specific heat is $0.1\,J/(kg\,{}^\circ C)$

```
thermalProperties(thermalmodel,'ThermalConductivity',2, ...
                               'MassDensity',1, ...
                               'SpecificHeat',0.1, ...
                               'Face',2);
```

Assume that the diamond-shaped region is a heat source with the density of $4\,W/m^3$.

```
internalHeatSource(thermalmodel,4,'Face',2);
```

Apply a constant temperature of $0\,{}^\circ C$ to the sides of the square plate.

```
thermalBC(thermalmodel,'Temperature',0,'Edge',[1 2 7 8]);
```

Set the initial temperature to $0\,{}^\circ C$.

```
thermalIC(thermalmodel,0);
```

Mesh the geometry.

```
generateMesh(thermalmodel);
```

The dynamics for this problem is very fast--the temperature reaches steady state in about 0.1 seconds. To capture the interesting part of the dynamics, set solution time to `logspace(-2,-1,10)`. This gives 10 logarithmically spaced solution times between 0.01 and 0.1. Solve the equation.

```
tlist = logspace(-2,-1,10);
thermalresults = solve(thermalmodel,tlist)


thermalresults =

  TransientThermalResults with properties:

      Temperature: [201×10 double]
    SolutionTimes: [1×10 double]
        XGradients: [201×10 double]
        YGradients: [201×10 double]
        ZGradients: []
             Mesh: [1×1 FEMesh]
```

Plot the solution with isothermal lines using a contour plot.

```
T = thermalresults.Temperature;
pdeplot(thermalmodel,'XYData',T(:,10),'Contour','on','ColorMap','hot')
```



## See Also

### See Also
evaluateHeatFlux | evaluateHeatRate | evaluateTemperatureGradient | interpolateTemperature | SteadyStateThermalResults

**Introduced in R2017a**

# thermalBC

Specify boundary conditions for a thermal model

## Syntax

```
thermalBC(thermalmodel,RegionType,RegionID,Name,Value)
thermalBC = thermalBC( ___ )
```

## Description

`thermalBC(thermalmodel,RegionType,RegionID,Name,Value)` adds a boundary condition to `thermalmodel`. The boundary condition applies to regions of type `RegionType` with ID numbers in `RegionID` and with arguments specified in the `Name,Value` pairs.

---

**Note:** Use `thermalBC` with the `HeatFlux` parameter to specify a heat flux to or from an external source. To specify internal heat generation, that is, heat sources that belong to the geometry of the model, use `internalHeatSource`.

---

`thermalBC = thermalBC( ___ )` returns the thermal boundary condition object.

## Examples

### Specify Temperature on the Boundary

Apply temperature boundary condition on two edges of a square.

```
thermalmodel = createpde('thermal');
geometryFromEdges(thermalmodel,@squareg);
thermalBC(thermalmodel,'Edge',[1,3],'Temperature',100)
```

```
ans =
```

```
    ThermalBC with properties:

                  RegionType: 'Edge'
                    RegionID: [1 3]
                 Temperature: 100
                    HeatFlux: []
       ConvectionCoefficient: []
                   Emissivity: []
          AmbientTemperature: []
                  Vectorized: 'off'
```

### Specify Heat Coming Through the Boundary

Apply heat flux boundary condition on two faces of a block.

```
thermalmodel = createpde('thermal','transient');
gm = importGeometry(thermalmodel,'Block.stl');
thermalBC(thermalmodel,'Face',[1,3],'HeatFlux',20)


ans =

  ThermalBC with properties:

                  RegionType: 'Face'
                    RegionID: [1 3]
                 Temperature: []
                    HeatFlux: 20
       ConvectionCoefficient: []
                   Emissivity: []
          AmbientTemperature: []
                  Vectorized: 'off'
```

### Specify Convection on the Boundary

Apply convection boundary condition on four faces of a block.

```
thermalModel = createpde('thermal','transient');
gm = importGeometry(thermalModel,'Block.stl');
thermalBC(thermalModel,'Face',[2 4 5 6], ...
                       'ConvectionCoefficient',5, ...
```

```
                          'AmbientTemperature',27)


ans =

  ThermalBC with properties:

                RegionType: 'Face'
                  RegionID: [2 4 5 6]
               Temperature: []
                  HeatFlux: []
     ConvectionCoefficient: 5
                Emissivity: []
        AmbientTemperature: 27
                Vectorized: 'off'
```

### Specify Radiation Through the Boundary

Apply radiation boundary condition on four faces of a block.

```
thermalmodel = createpde('thermal','transient');
gm = importGeometry(thermalmodel,'Block.stl');
thermalmodel.StefanBoltzmannConstant = 5.670373E-8;
thermalBC(thermalmodel,'Face',[2,4,5,6],...
                          'Emissivity',0.1,...
                          'AmbientTemperature',300)


ans =

  ThermalBC with properties:

                RegionType: 'Face'
                  RegionID: [2 4 5 6]
               Temperature: []
                  HeatFlux: []
     ConvectionCoefficient: []
                Emissivity: 0.1000
        AmbientTemperature: 300
                Vectorized: 'off'
```

- "Heat Conduction in a Spherical Multidomain Geometry with Nonuniform Heat Flux"

# Input Arguments

### `thermalmodel` — Thermal model
ThermalModel object

Thermal model, specified as a ThermalModel object. The model contains the geometry, mesh, thermal properties of the material, internal heat source, boundary conditions, and initial conditions.

Example: thermalmodel = createpde('thermal')

### `RegionType` — Geometric region type
'Edge' for a 2-D model | 'Face' for a 3-D model

Geometric region type, specified as 'Edge' or 'Face'.

Example: thermalBC(thermalmodel,'Face',1,'Temperature',72)

Data Types: char

### `RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using pdegplot, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Example: thermalBC(thermalmodel,'Edge',2:5,'Temperature',72)

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: thermalBC(thermalmodel,'Face',[1,2],'Temperature',100)

### `'Temperature'` — Temperature boundary condition
number | function handle

Temperature boundary condition, specified as a number or a function handle. Use a function handle to specify spatially or temporally varying temperature.

Example: `thermalBC(thermalmodel,'Face',1,'Temperature',75.2)`

Data Types: `double | function_handle`

### `'HeatFlux'` — Heat flux boundary condition
number | function handle

Heat flux boundary condition, specified as a number or a function handle. Use a function handle to specify spatially or temporally varying heat flux.

Example: `thermalBC(thermalmodel,'Face',[1,3],'HeatFlux',20)`

Data Types: `double | function_handle`

### `'ConvectionCoefficient'` — Coefficient for convection to ambient heat transfer condition
number | function handle

Convection to ambient boundary condition, specified as a number or a function handle. Use a function handle to specify spatially or temporally varying convection coefficient. Specify ambient temperature using the `AmbientTemperature` argument. The value of `ConvectionCoefficient` is positive for heat convection into the ambient environment.

Example: `thermalBC(thermalmodel,'Edge', [2,4],'ConvectionCoefficient',5,'AmbientTemperature',60)`

Data Types: `double | function_handle`

### `'Emissivity'` — Radiation emissivity coefficient
number in the range (0,1)

Radiation emissivity coefficient, specified as a number in the range (0,1). Use a function handle to specify spatially or temporally varying emissivity. Specify ambient temperature using the `AmbientTemperature` argument and the Stefan-Boltzmann constant using the thermal model properties. The value of `Emissivity` is positive for heat radiation into the ambient environment.

Example: `thermalmodel.StefanBoltzmannConstant = 5.670373E-8; thermalBC(thermalmodel,'Edge', [2,4,5,6],'Emissivity',0.1,'AmbientTemperature',300)`

Data Types: `double | function_handle`

### `'AmbientTemperature'` — Ambient temperature
number

Ambient temperature, specified as a number. The ambient temperature value is required for specifying convection and radiation boundary conditions.

Example: `thermalBC(thermalmodel,'Edge', [2,4],'ConvectionCoefficient',5,'AmbientTemperature',60)`

Data Types: `double`

## Output Arguments

**`thermalBC` — Handle to thermal boundary condition**
object

Handle to thermal boundary condition, returned as an object. `thermalBC` associates the thermal boundary condition with the geometric region.

# See Also

## See Also
`applyBoundaryCondition` | `internalHeatSource` | `thermalIC` | `thermalProperties`

## Topics
"Heat Conduction in a Spherical Multidomain Geometry with Nonuniform Heat Flux"

**Introduced in R2017a**

# thermalIC

Set initial conditions or initial guess for a thermal model

## Syntax

```
thermalIC(thermalmodel,T0)
thermalIC(thermalmodel,T0,RegionType,RegionID)
thermalIC(thermalmodel,Tresults)
thermalIC(thermalmodel,Tresults,iT)
thermalIC = thermalIC( ___ )
```

## Description

`thermalIC(thermalmodel,T0)` sets initial temperature or initial guess for temperature to the entire geometry.

`thermalIC(thermalmodel,T0,RegionType,RegionID)` sets initial temperature or initial guess for temperature to a particular geometry region.

`thermalIC(thermalmodel,Tresults)` sets initial temperature or initial guess for temperature using the solution `Tresults` from a previous thermal analysis on the same geometry and mesh. If `Tresults` is obtained by solving a transient thermal problem, `thermalIC` uses the solution `Tresults` for the last time-step.

`thermalIC(thermalmodel,Tresults,iT)` sets initial temperature or initial guess for temperature using the solution `Tresults` for the time-step `iT` from a previous thermal analysis on the same geometry and mesh.

`thermalIC = thermalIC( ___ )`, for any previous syntax, returns a handle to the thermal initial conditions object.

## Examples

### Constant Initial Temperature

Create a thermal model, import geometry, and set the initial temperature to 0 on the entire geometry.

```
thermalModel = createpde('thermal','transient');
geometryFromEdges(thermalModel,@lshapeg);
thermalIC(thermalModel,O)


ans =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: [1 2 3]
    InitialTemperature: O
```
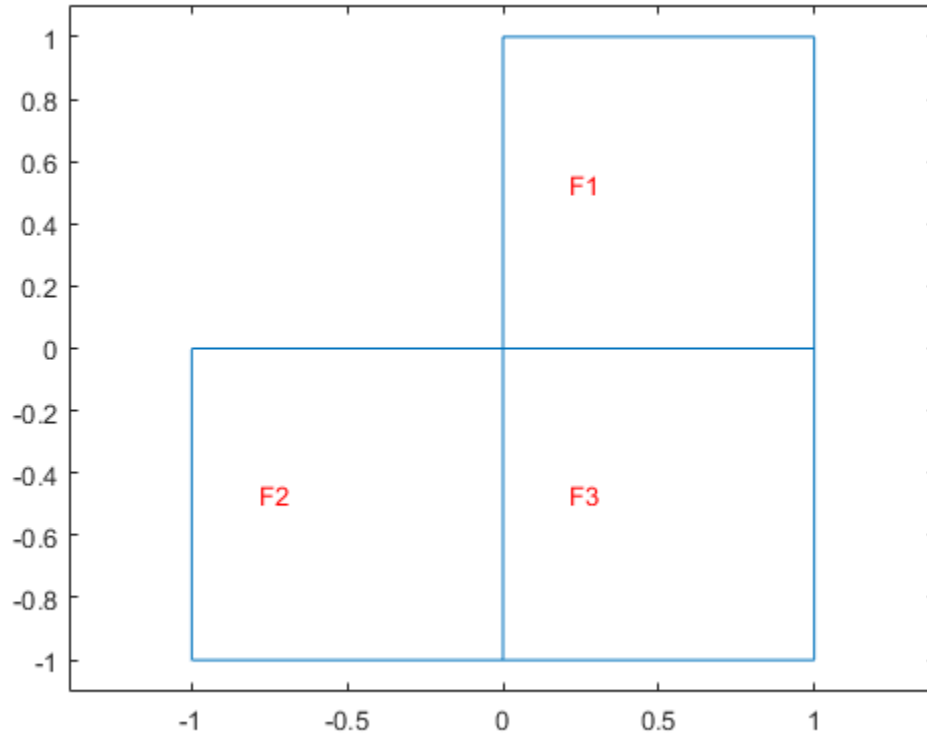
### Different Initial Temperatures on Subdomains

Set different initial conditions on each portion of the L-shaped membrane geometry.

Create a model and include a 2-D geometry.

```
thermalModel = createpde('thermal','transient');
geometryFromEdges(thermalModel,@lshapeg);
pdegplot(thermalModel,'FaceLabels','on')
axis equal
ylim([-1.1 1.1])
```

Set initial conditions.

```
thermalIC(thermalModel,0,'Face',1)
thermalIC(thermalModel,10,'Face',2)
thermalIC(thermalModel,75,'Face',3)
```

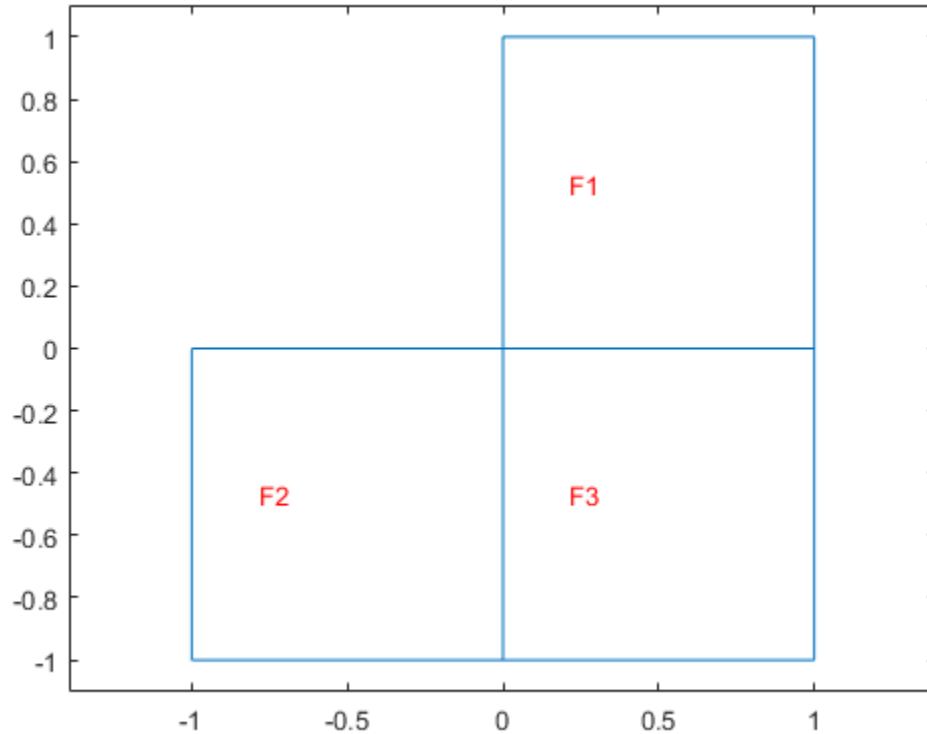```
ans =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: 1
    InitialTemperature: 0
```

```
ans =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: 2
    InitialTemperature: 10


ans =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: 3
    InitialTemperature: 75
```

### Initial Temperature as a Function 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`.

```
thermalModel = createpde('thermal','transient');
geometryFromEdges(thermalModel,@lshapeg);
pdegplot(thermalModel,'FaceLabels','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;
thermalIC(thermalModel,initfun)
```

```
ans =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: [1 2 3]
    InitialTemperature: @(locations)locations.x.^2+locations.y.^2
```

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;
thermalIC(thermalModel,initfun2,'Face',2)
```

```
ans =

  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: 2
    InitialTemperature: @(locations)locations.x.^2-locations.y.^4
```

### Initial Condition as Previously Obtained Solution

Create a thermal model and include a square geometry.

```
thermalmodel = createpde('thermal','transient');
geometryFromEdges(thermalmodel,@squareg);
pdegplot(thermalmodel,'FaceLabels','on')
ylim([-1.1,1.1])
axis equal
```
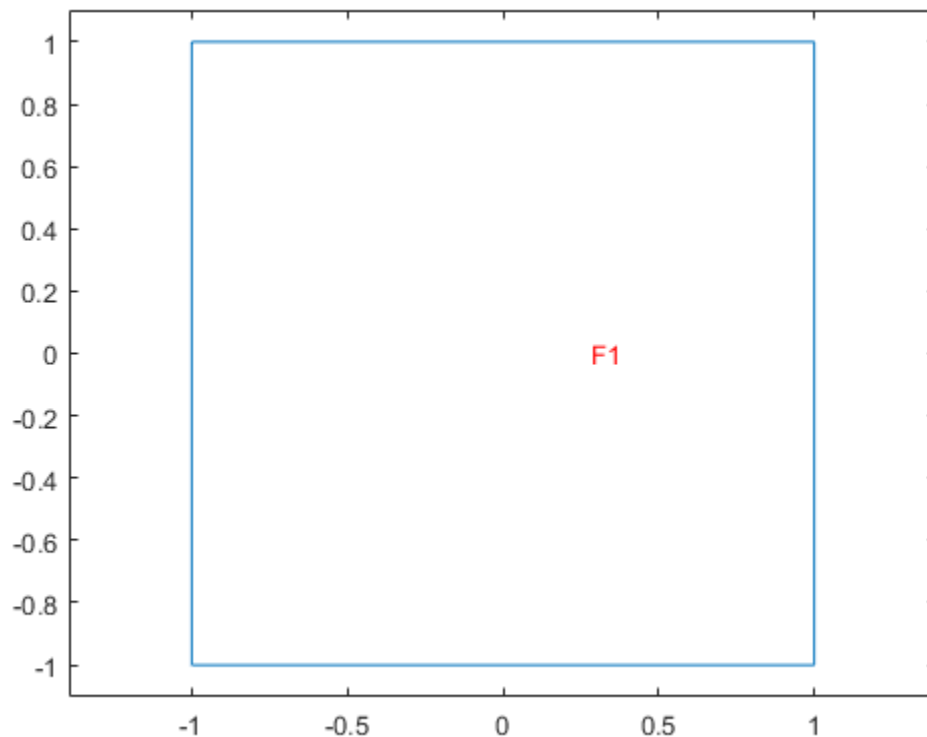
Specify material properties and internal heat source, and set boundary conditions and initial conditions.

```
thermalProperties(thermalmodel,'ThermalConductivity',500,...
                               'MassDensity',200,...
                               'SpecificHeat',100);

internalHeatSource(thermalmodel,2);
thermalBC(thermalmodel,'Edge',[1,3],'Temperature',100);
thermalIC(thermalmodel,0);
```

Generate mesh, solve the problem, and plot the solution.

```
generateMesh(thermalmodel);
```

```
tlist = 0:0.5:10;
result1 = solve(thermalmodel,tlist)
pdeplot(thermalmodel,'XYData',result1.Temperature(:,end))


result1 =

  TransientThermalResults with properties:

      Temperature: [177×21 double]
    SolutionTimes: [1×21 double]
        XGradients: [177×21 double]
        YGradients: [177×21 double]
        ZGradients: []
             Mesh: [1×1 FEMesh]
```
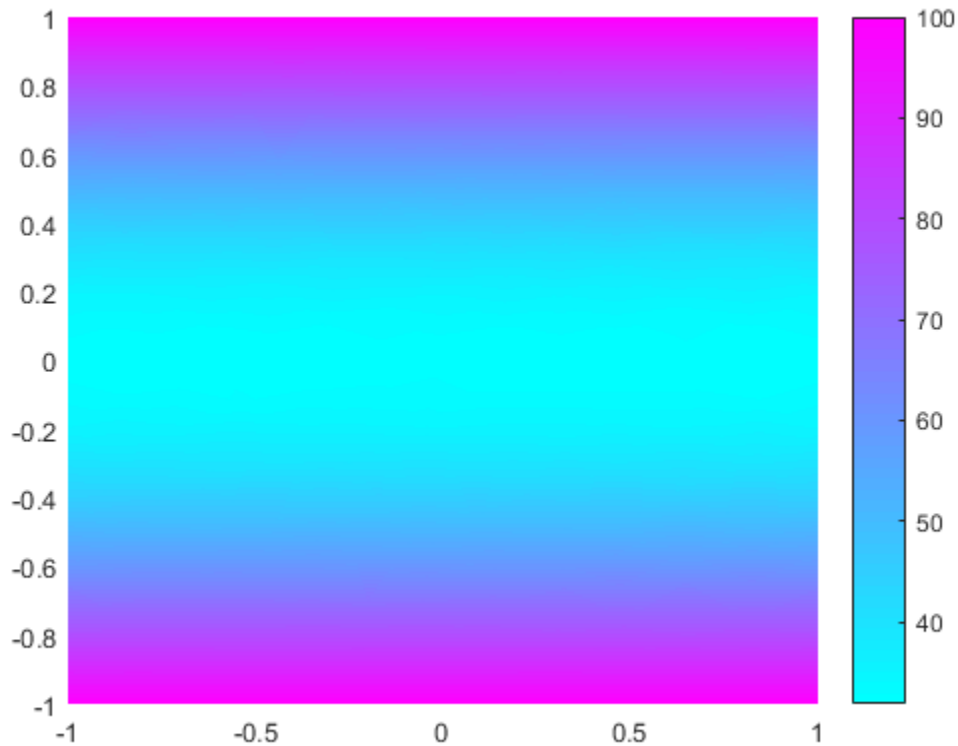
Now, resume the analysis and solve the problem for times from 10 to 15 seconds. Use the previously obtained solution for 10 seconds as an initial condition. Since 10 seconds is the last element in `tlist`, you do not need to specify the solution time index. By default, `thermalIC` uses the last solution index.

```
thermalIC(thermalmodel,result1)
```

```
ans =

  NodalThermalICs with properties:

    InitialTemperature: [177×1 double]
```
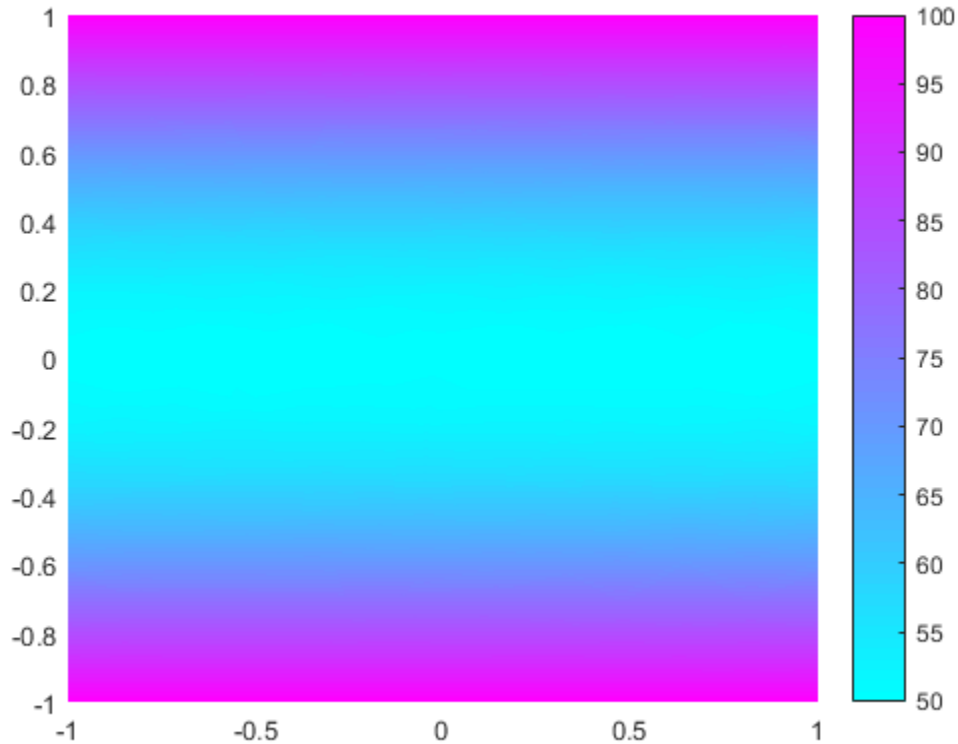
Solve the problem and plot the solution.

```
result2 = solve(thermalmodel,10:0.5:15)
pdeplot(thermalmodel,'XYData',result2.Temperature(:,end))
```

```
result2 =

  TransientThermalResults with properties:

      Temperature: [177×11 double]
    SolutionTimes: [10 10.5000 11 11.5000 12 12.5000 13 13.5000 14 14.5000 15]
       XGradients: [177×11 double]
       YGradients: [177×11 double]
       ZGradients: []
             Mesh: [1×1 FEMesh]
```

To use the previously obtained solution for a particular solution time instead of the last one, specify the solution time index as a third parameter of `thermalIC`. For example, use the solution at time 5 seconds, which is the 11th element in `tlist`.

```
tlist(11)
thermalIC(thermalmodel,result1,11);
```
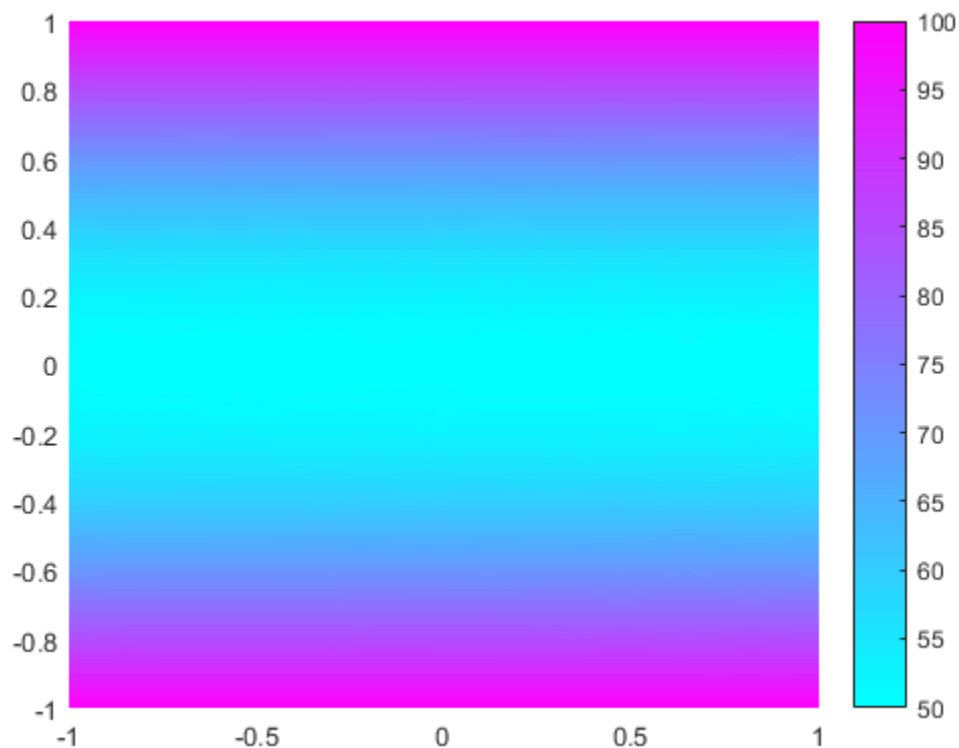
```
ans =

     5
```

```
result2 = solve(thermalmodel,5:0.5:15)
```

```
pdeplot(thermalmodel,'XYData',result2.Temperature(:,end))


result2 =

  TransientThermalResults with properties:

       Temperature: [177×21 double]
     SolutionTimes: [1×21 double]
         XGradients: [177×21 double]
         YGradients: [177×21 double]
         ZGradients: []
              Mesh: [1×1 FEMesh]
```

## Input Arguments

### `thermalmodel` — Thermal model
`ThermalModel` object

Thermal model, specified as a `ThermalModel` object. The model contains the geometry, mesh, thermal properties of the material, internal heat source, boundary conditions, and initial conditions.

Example: `thermalmodel = createpde('thermal')`

### T0 — Initial temperature or initial guess for temperature
number | function handle

Initial temperature or initial guess for temperature, specified as a number or a function handle. Use a function handle to specify spatially varying temperature.

Data Types: `double` | `function_handle`

### RegionType — Geometric region type
`'Vertex'` | `'Edge'` | `'Face'` for a 3-D model only

Geometric region type, specified as `'Vertex'`, `'Edge'`, or `'Face'` for a 3-D model. For a 2-D model, use `'Vertex'` or `'Edge'`.

Example: `thermalIC(thermalmodel,10,'Face',1)`

Data Types: `char`

### RegionID — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs using `pdegplot`, as shown in "Create Geometry and Remove Face Boundaries" on page 2-12 or "Create and View 3-D Geometry" on page 2-36.

Example: `thermalIC(thermalmodel,10,'Edge',2:5)`

Data Types: `double`

### Tresults — Thermal model solution
`ThermalResults` object

Thermal model solution, specified as a `ThermalResults` object. Create `Tresults` by using `solve`.

### iT — Time index
positive integer

Time index, specified as a positive integer.

Example: `thermalIC(thermalmodel,Tresults,21)`

Data Types: `double`

## Output Arguments

**`thermalIC` — Handle to initial condition**
object

Handle to initial condition, returned as an object. `thermalIC` associates the thermal initial condition with the geometric region in the case of a geometric assignment, or the nodes in the case of a results-based assignment.

## See Also

### See Also
internalHeatSource | setInitialConditions | thermalBC | thermalProperties

**Introduced in R2017a**

# 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

**Introduced before R2006a**

# 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 2-183.)

## See Also
decsg | wgeom

**Introduced before R2006a**

# 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 2-14.

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 2-16.

## See Also
decsg

**Introduced before R2006a**