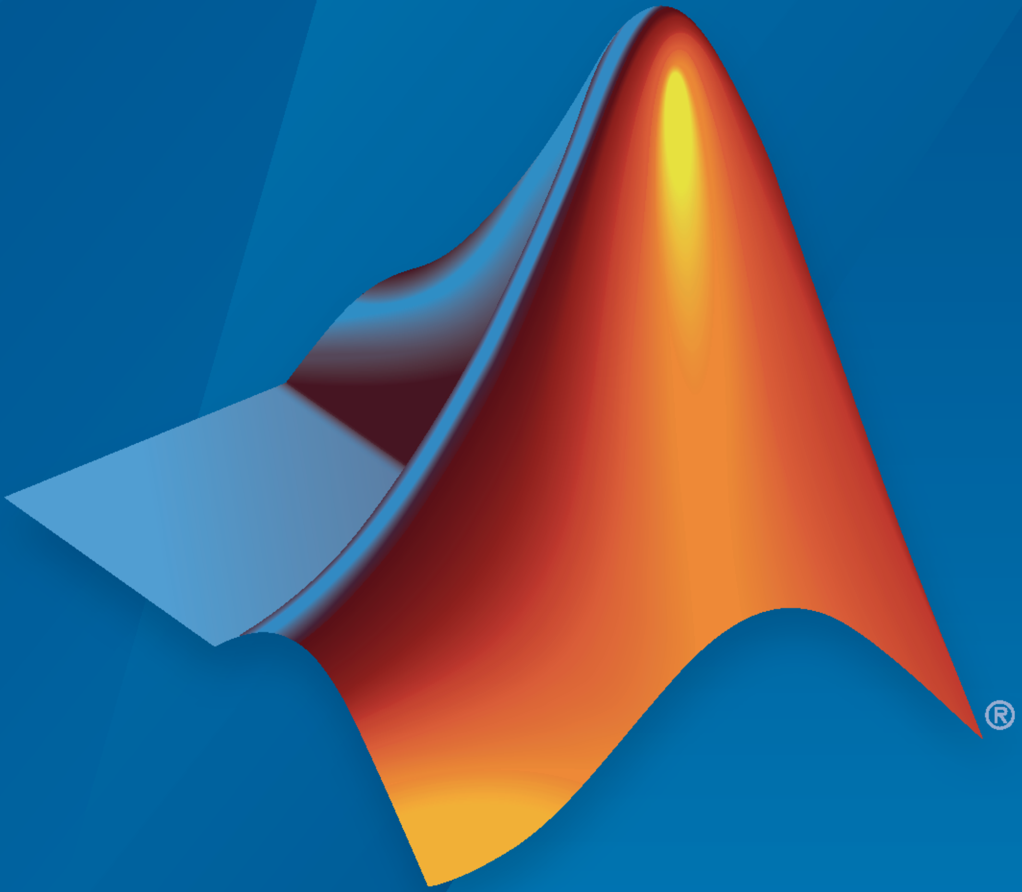
Partial Differential Equation Toolbox™

User's Guide



MATLAB®

R2022b

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Partial Differential Equation Toolbox™ User's Guide

© COPYRIGHT 1995–2022 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

August 1995	First printing	New for Version 1.0
February 1996	Second printing	Revised for Version 1.0.1
July 2002	Online only	Revised for Version 1.0.4 (Release 13)
September 2002	Third printing	Minor Revision for Version 1.0.4
June 2004	Online only	Revised for Version 1.0.5 (Release 14)
October 2004	Online only	Revised for Version 1.0.6 (Release 14SP1)
March 2005	Online only	Revised for Version 1.0.6 (Release 14SP2)
August 2005	Fourth printing	Minor Revision for Version 1.0.6
September 2005	Online only	Revised for Version 1.0.7 (Release 14SP3)
March 2006	Online only	Revised for Version 1.0.8 (Release 2006a)
March 2007	Online only	Revised for Version 1.0.10 (Release 2007a)
September 2007	Online only	Revised for Version 1.0.11 (Release 2007b)
March 2008	Online only	Revised for Version 1.0.12 (Release 2008a)
October 2008	Online only	Revised for Version 1.0.13 (Release 2008b)
March 2009	Online only	Revised for Version 1.0.14 (Release 2009a)
September 2009	Online only	Revised for Version 1.0.15 (Release 2009b)
March 2010	Online only	Revised for Version 1.0.16 (Release 2010a)
September 2010	Online only	Revised for Version 1.0.17 (Release 2010b)
April 2011	Online only	Revised for Version 1.0.18 (Release 2011a)
September 2011	Online only	Revised for Version 1.0.19 (Release 2011b)
March 2012	Online only	Revised for Version 1.0.20 (Release 2012a)
September 2012	Online only	Revised for Version 1.1 (Release 2012b)
March 2013	Online only	Revised for Version 1.2 (Release 2013a)
September 2013	Online only	Revised for Version 1.3 (Release 2013b)
March 2014	Online only	Revised for Version 1.4 (Release 2014a)
October 2014	Online only	Revised for Version 1.5 (Release 2014b)
March 2015	Online only	Revised for Version 2.0 (Release 2015a)
September 2015	Online only	Revised for Version 2.1 (Release 2015b)
March 2016	Online only	Revised for Version 2.2 (Release 2016a)
September 2016	Online only	Revised for Version 2.3 (Release 2016b)
March 2017	Online only	Revised for Version 2.4 (Release 2017a)
September 2017	Online only	Revised for Version 2.5 (Release 2017b)
March 2018	Online only	Revised for Version 3.0 (Release 2018a)
September 2018	Online only	Revised for Version 3.1 (Release 2018b)
March 2019	Online only	Revised for Version 3.2 (Release 2019a)
September 2019	Online only	Revised for Version 3.3 (Release 2019b)
March 2020	Online only	Revised for Version 3.4 (Release 2020a)
September 2020	Online only	Revised for Version 3.5 (Release 2020b)
March 2021	Online only	Revised for Version 3.6 (Release 2021a)
September 2021	Online only	Revised for Version 3.7 (Release 2021b)
March 2022	Online only	Revised for Version 3.8 (Release 2022a)
September 2022	Online only	Revised for Version 3.9 (Release 2022b)

Contents

1 Getting Started

2 Setting Up Your PDE

Solving PDEs

3

4 PDE Modeler App

Functions

5

Getting Started

Partial Differential Equation Toolbox Product Description

Solve partial differential equations using finite element analysis

Partial Differential Equation Toolbox provides functions for solving structural mechanics, heat transfer, and general partial differential equations (PDEs) using finite element analysis.

You can perform linear static analysis to compute deformation, stress, and strain. For modeling structural dynamics and vibration, the toolbox provides a direct time integration solver. You can analyze a component's structural characteristics by performing modal analysis to find natural frequencies and mode shapes. You can model conduction-dominant heat transfer problems to calculate temperature distributions, heat fluxes, and heat flow rates through surfaces. You can also solve standard problems such as diffusion, electrostatics, and magnetostatics, as well as custom PDEs.

Partial Differential Equation Toolbox lets you import 2D and 3D geometries from STL or mesh data. You can automatically generate meshes with triangular and tetrahedral elements. You can solve PDEs by using the finite element method, and postprocess results to explore and analyze them.

Key Features

- Structural analysis, including linear static, dynamic, and modal analysis
- Heat transfer analysis for conduction-dominant problems
- General linear and nonlinear PDEs for stationary, time-dependent, and eigenvalue problems
- 2D and 3D geometry import from STL files and mesh data
- Automatic meshing using triangular and tetrahedral elements with linear or quadratic basis functions
- User-defined functions for specifying PDE coefficients, boundary conditions, and initial conditions
- Plotting and animating results, as well as derived and interpolated values

Equations You Can Solve Using PDE Toolbox

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 **m**, **d**, and **a** are *N*-by-*N* matrices, **f** is an *N*-by-1 vector, and **c** is a 2*N*-by-2*N* tensor (2-D geometry) or a 3*N*-by-3*N* tensor (3-D geometry). For the meaning of $\mathbf{c} \otimes \mathbf{u}$, see “c Coefficient for specifyCoefficients” on page 2-92.

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 **h** multiplying the solution vector **u**, and equaling the vector **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.

See Also

Related Examples

- “Put Equations in Divergence Form” on page 2-87
- “Solve Problems Using PDEModel Objects” on page 2-2
- “f Coefficient for specifyCoefficients” on page 2-90
- “c Coefficient for specifyCoefficients” on page 2-92
- “m, d, or a Coefficient for specifyCoefficients” on page 2-107

Solve 2-D PDEs Using the PDE Modeler App

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

1. Start the PDE Modeler app by using the **Apps** tab or typing `pdeModeler` in the MATLAB® Command Window. For details, see “Open the PDE Modeler 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 “2-D Geometry Creation in PDE Modeler 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 Modeler App” on page 4-12.
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 “Coefficients for Scalar PDEs” on page 4-14 and “Coefficients for Systems of PDEs” on page 4-16.

   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 Modeler App” on page 4-24.
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 Modeler App” on page 4-26.
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 Modeler App” on page 4-31.

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 ∂Ω 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 Modeler app adds the necessary steps automatically.

- If you do not create a geometry, the PDE Modeler app uses an L-shaped geometry with the default boundary conditions.
- If you initialize the mesh while in the draw mode, the PDE Modeler 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 Modeler app first initializes the mesh.
- If you solve the PDE without generating a mesh, the PDE Modeler app initializes a mesh before solving the PDE.
- If you select a plot type and choose to plot the solution, the PDE Modeler app checks if the solution to the current PDE is available. If not, the PDE Modeler 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 Modeler 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.

See Also

Related Examples

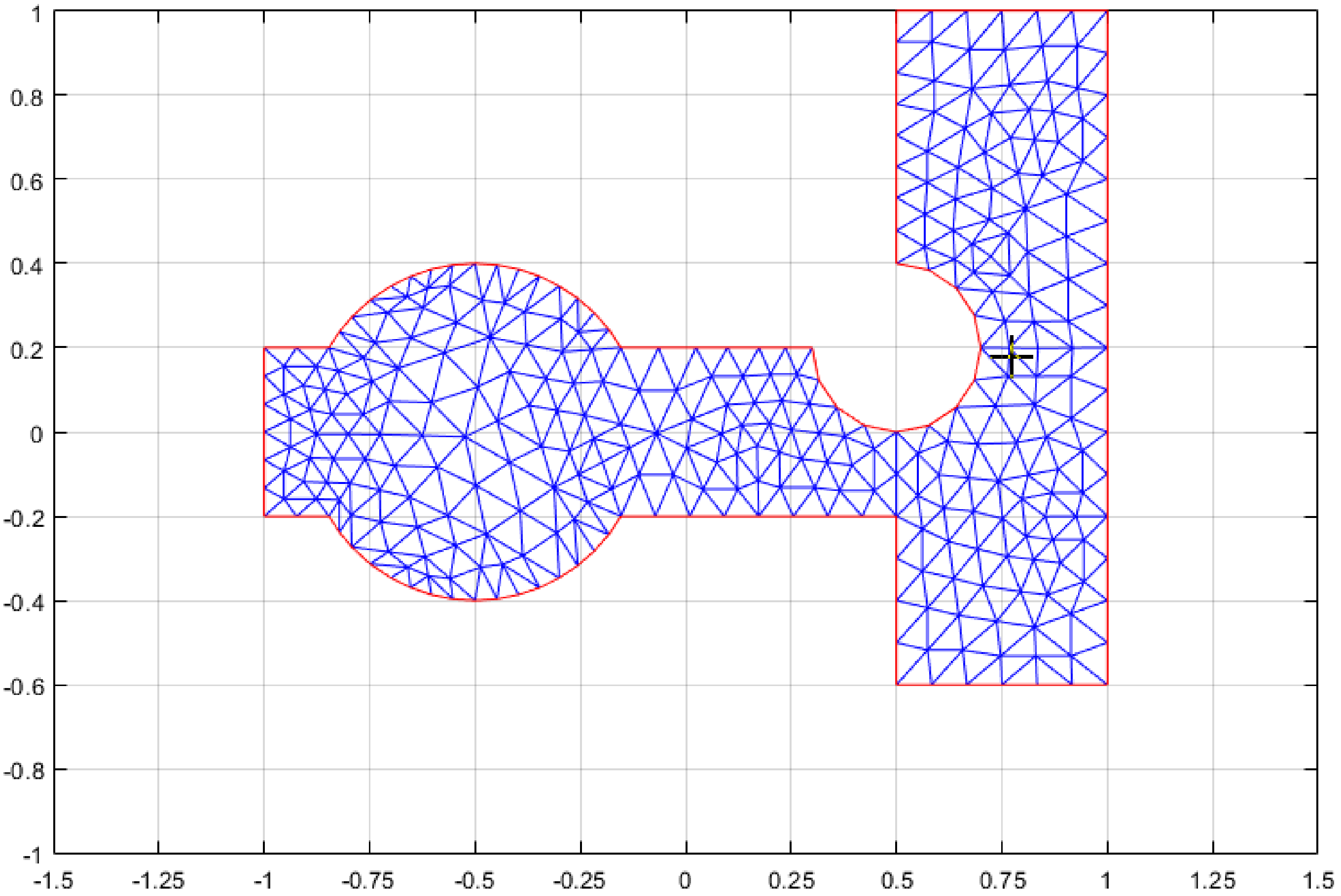
- “Poisson’s Equation with Complex 2-D Geometry: PDE Modeler App” on page 1-7
- “Poisson's Equation on Unit Disk” on page 3-224
- “Current Density Between Two Metallic Conductors: PDE Modeler App” on page 3-192
- “Minimal Surface Problem” on page 3-241

Poisson’s Equation with Complex 2-D Geometry: PDE Modeler App

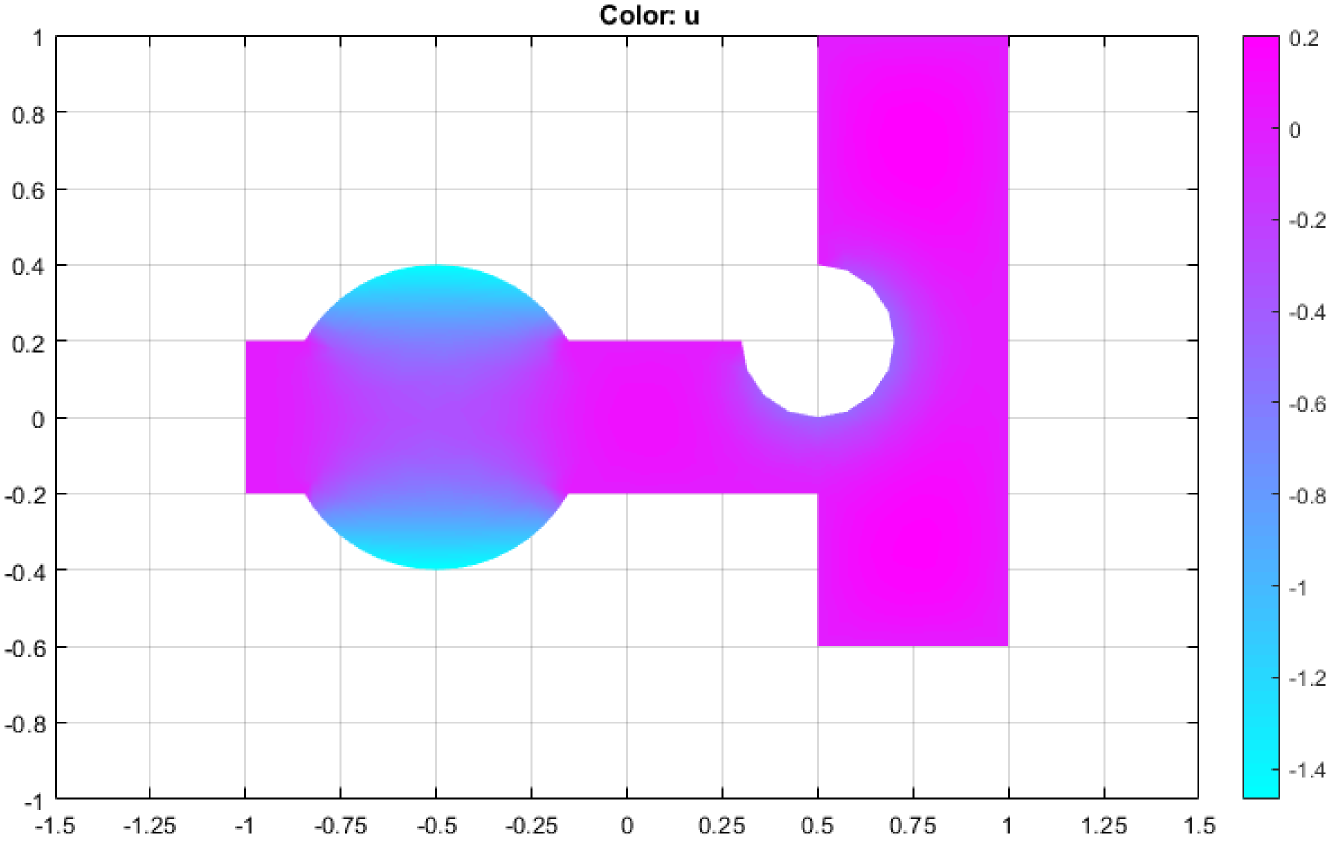
This example shows how to solve the Poisson's equation, $-\Delta u = f$ on a 2-D geometry created as a combination of two rectangles and two circles.

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

1. Open the PDE Modeler app by using the `pdeModeler` command.
2. Display grid lines. To do this, select **Options** > **Grid Spacing** and clear the **Auto** checkbox for the x-axis linear spacing. Enter **X-axis linear spacing** as `-1.5:0.25:1.5`. Then select **Options** > **Grid**.
3. Align new shapes to the grid lines by selecting **Options** > **Snap**.
4. Draw two circles: one with the radius 0.4 and the center at (-0.5,0) and another with the radius 0.2 and the center at (0.5,0.2). To draw a circle, first click the ⊕ button. Then right-click the origin and drag to draw a circle. Right-clicking constrains the shape you draw so that it is a circle rather than an ellipse.
5. Draw two rectangles: one with corners (-1,0.2), (1,0.2), (1,-0.2), and (-1,-0.2) and another with corners (0.5,1), (1,1), (1,-0.6), and (0.5,-0.6). To draw a rectangle, first click the ▭ button. Then click any corner and drag to draw the rectangle.
6. Model the geometry by entering `(R1+C1+R2)-C2` in the **Set formula** field.
7. Save the model to a file by selecting **FileSave As**.
8. Remove the subdomain borders. To do this, switch to the boundary mode by selecting **Boundary** > **Boundary Mode**. Then select **Boundary** > **Remove All Subdomain Borders**.
9. Specify the boundary conditions for all circle arcs. Using **Shift**+click, select these borders. Then select **Boundary** > **Specify Boundary Conditions** and specify the Neumann boundary condition with g = -5 and q = 0. This boundary condition means that the solution has a slope of –5 in the normal direction for these boundary segments.
10. For all other boundaries, keep the default Dirichlet boundary condition: `h = 1`, `r = 0`.
11. Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Specify `c = 1`, `a = 0`, and `f = 10`.
12. Initialize the mesh by selecting **Mesh** > **Initialize Mesh**. Refine the mesh by selecting **Mesh** > **Refine Mesh**.

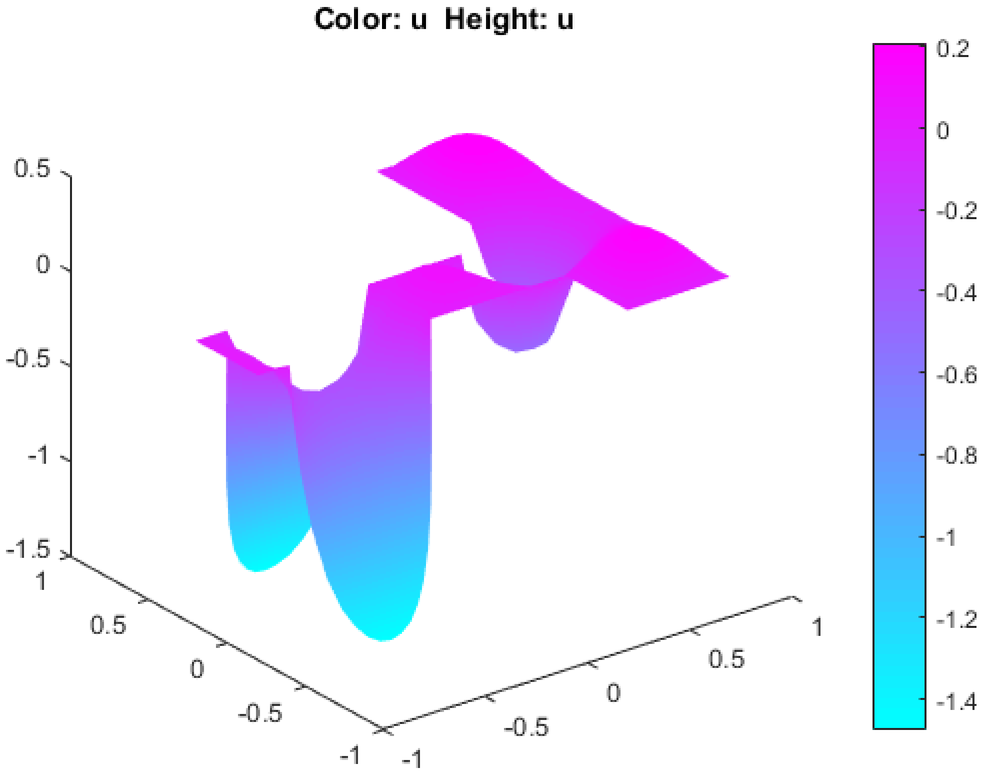


13 Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar. The toolbox assembles the PDE problem, solves it, and plots the solution.



14 Plot the solution as a 3-D plot:

- **a** Select **Plot** > **Parameters**.
- **b** In the resulting dialog box, select **Height (3-D plot)**.
- **c** Click **Plot**.


Color: u Height: u
0.5
0
-0.5
-1
-1.5
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.2
-1.4

Finite Element Method 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 Ω 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 Ω.

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 Ω.

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

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 Ω, 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 Ω into smaller subdomains or elements Ω^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 Ω^e, you can write the discretized weak form of the PDE as:

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

Next, let ϕ_i, with $i = 1, 2, \ldots, N_\mathrm{p}$, 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 ϕ_i, ϕ_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 λ, where λ 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 ∇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`

Deflection Analysis of Bracket

This example shows how to analyze a 3-D mechanical part under an applied load using finite element analysis (FEA) and determine the maximal deflection.

Create Structural Analysis Model

The first step in solving a linear elasticity problem is to create a structural analysis model. This model is a container that holds the geometry, structural material properties, damping parameters, body loads, boundary loads, boundary constraints, superelement interfaces, initial displacement and velocity, and mesh.

```
model = createpde("structural","static-solid");
```

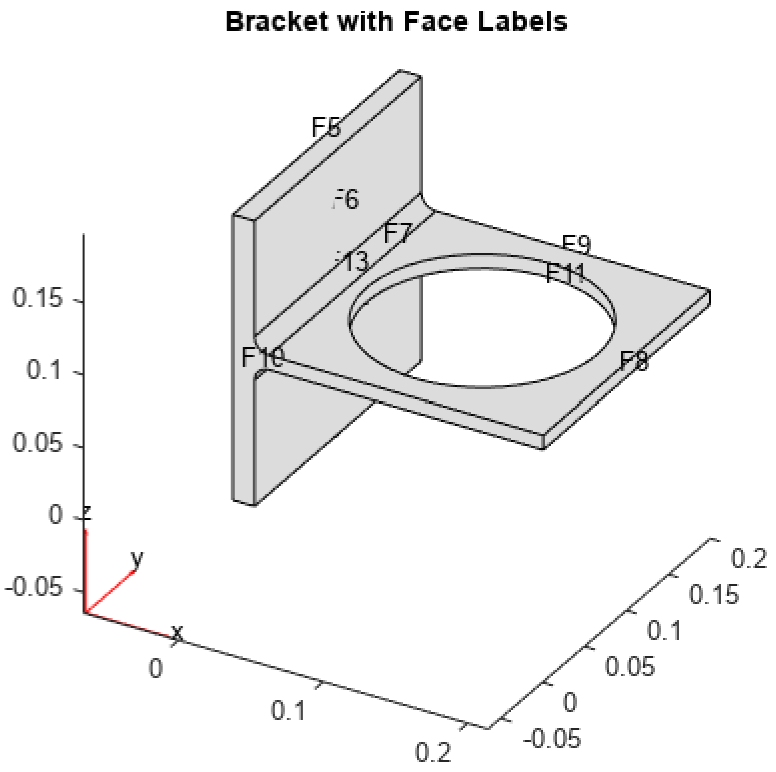
Import Geometry

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.

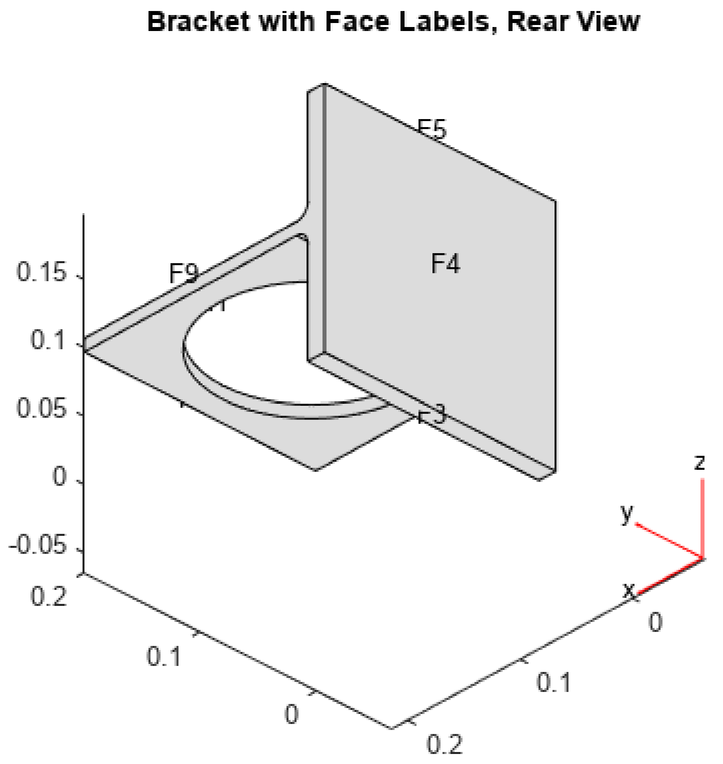
```
importGeometry(model,"BracketWithHole.stl");
```

Plot the geometry, displaying face labels.

```
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, Rear View

Specify Structural Properties of Material

Specify Young's modulus and Poisson's ratio of the material.

```
structuralProperties(model,"YoungsModulus",200e9, ...
                           "PoissonsRatio",0.3);
```

Apply Boundary Conditions and Loads

The problem has two boundary conditions: the back face (face 4) is fixed, and the front face (face 8) has an applied load. All other boundary conditions, by default, are free boundaries.

```
structuralBC(model,"Face",4,"Constraint","fixed");
```

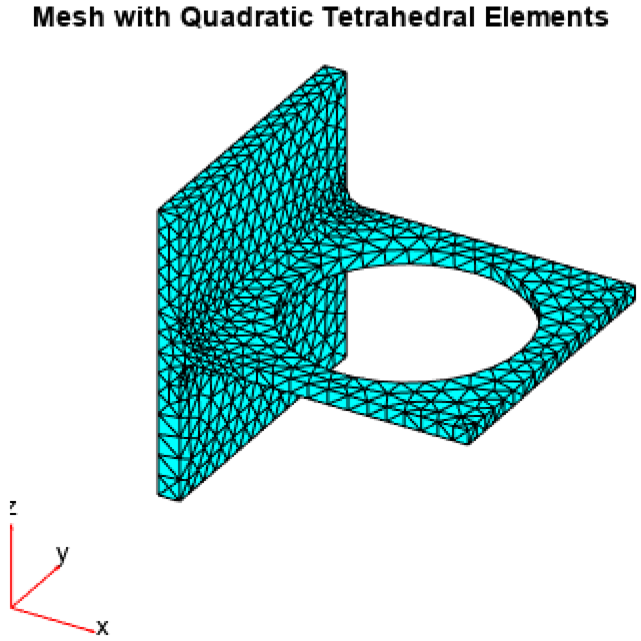
Apply a distributed load in the negative *z*-direction to the front face.

```
structuralBoundaryLoad(model,"Face",8,"SurfaceTraction",[0;0;-1e4]);
```

Generate Mesh

Generate and plot a mesh.

```
generateMesh(model);
figure
pdeplot3D(model)
title("Mesh with Quadratic Tetrahedral Elements");
```



Calculate Solution

Use the `solve` function to calculate the solution.

```
result = solve(model)
```

```
result = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [5993x1 double]
              Mesh: [1x1 FEMesh]
```

Examine Solution

Find the maximal deflection of the bracket in the *z*-direction.

```
minUz = min(result.Displacement.uz);
fprintf("Maximal deflection in the z-direction is %g meters.", minUz)
```

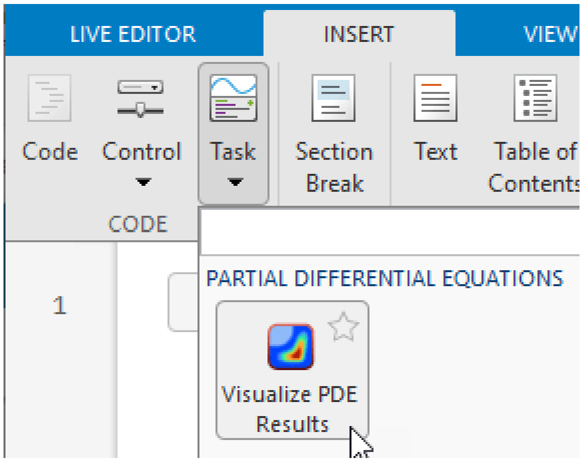
```
Maximal deflection in the z-direction is -4.43075e-05 meters.
```

Plot Results Using Visualize PDE Results Live Editor Task

Visualize the displacement components and the von Mises stress by using the **Visualize PDE Results** Live Editor task. The maximal deflections are in the *z*-direction. Because the bracket and the load are symmetric, the *x*-displacement and *z*-displacement are symmetric, and the *y*-displacement is antisymmetric with respect to the center line.

First, create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

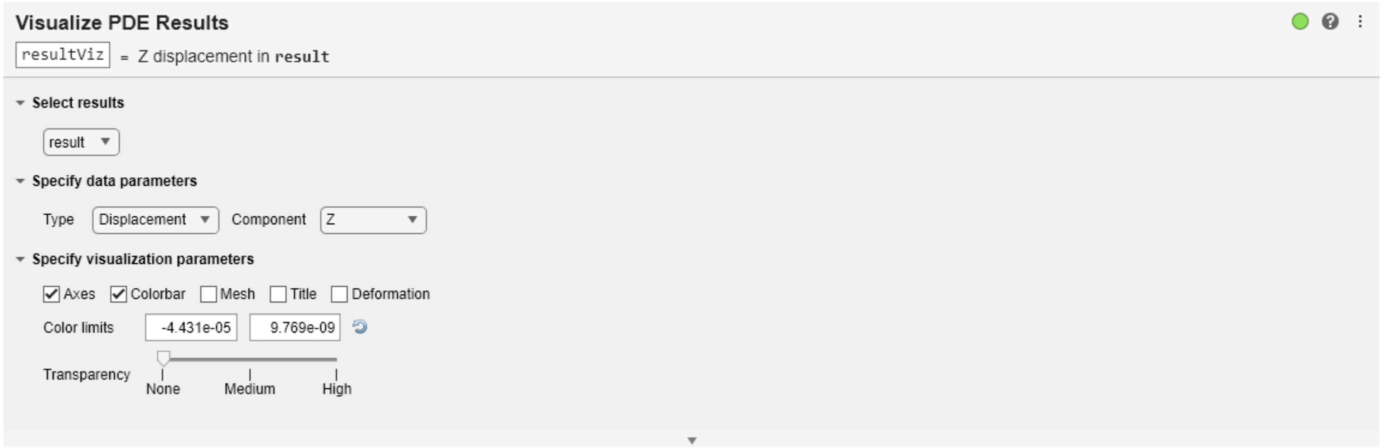
On the **Live Editor** tab, select **Task** > **Visualize PDE Results**. This action inserts the task into your script.



To plot the *z*-displacement, follow these steps. To plot the *x*- and *y*-displacements, follow the same steps, but set **Component** to *X* and *Y*, respectively.

1. In the **Select results** section of the task, select `result` from the drop-down list.
2. In the **Specify data parameters** section of the task, set **Type** to *Displacement* and **Component** to *Z*.
3. In the **Specify visualization parameters** section of the task, clear the **Deformation** check box.

Here, the blue color represents the lowest displacement value, and the red color represents the highest displacement value. The bracket load causes face 8 to dip down, so the maximum *z*-displacement appears blue.



Visualize PDE Results

`resultViz` = Z displacement in `result`

Select results

result

Specify data parameters

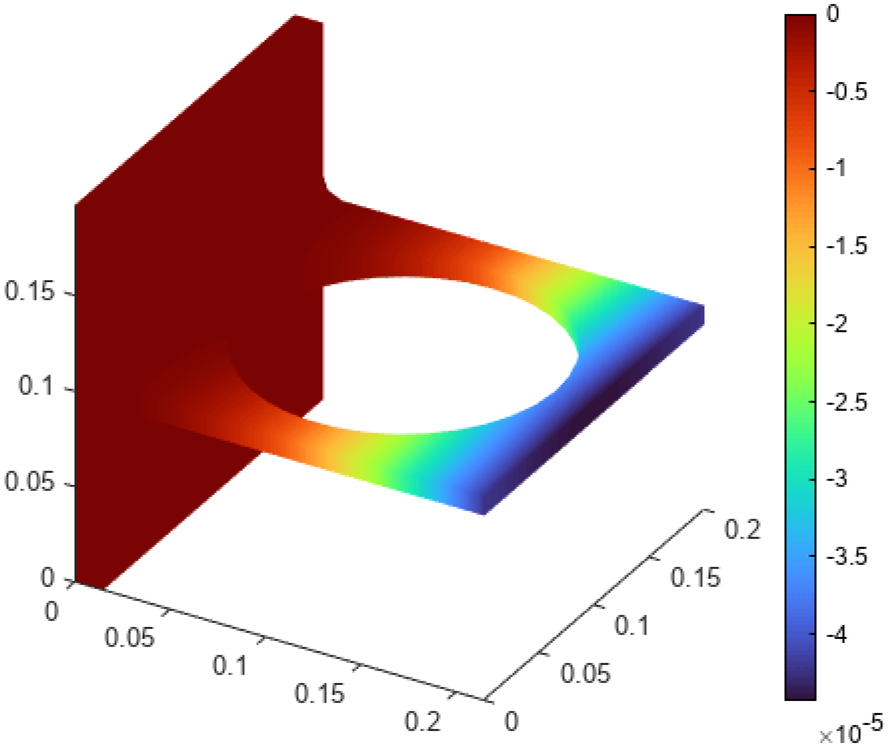
Type: Displacement   Component: Z

Specify visualization parameters

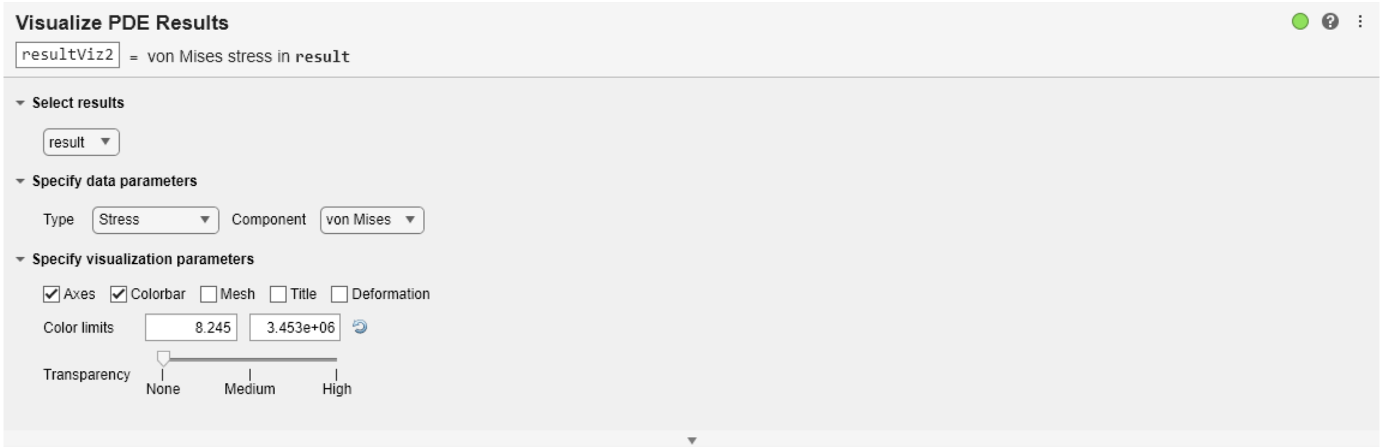
☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☐ Deformation

Color limits: -4.431e-05   9.769e-09

Transparency: None



To plot the von Mises stress, in the **Specify data parameters** section of the task, set **Type** to *Stress* and **Component** to *von Mises*.



Visualize PDE Results

`resultViz2` = von Mises stress in `result`

Select results

result ▾

Specify data parameters

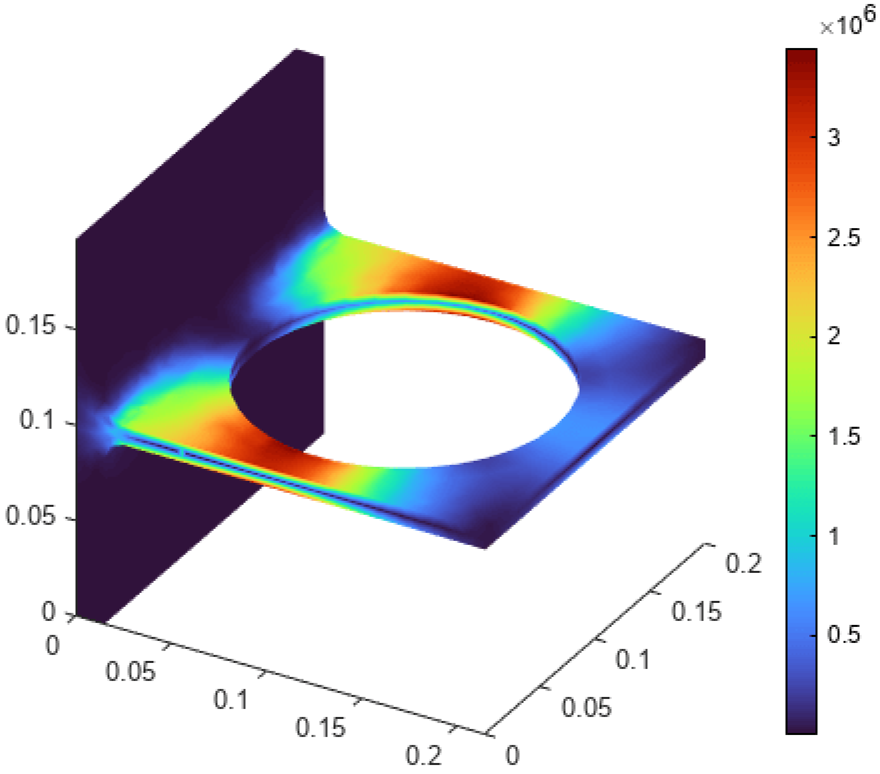
Type: Stress ▾ Component: von Mises ▾

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☐ Deformation

Color limits: 8.245 3.453e+06

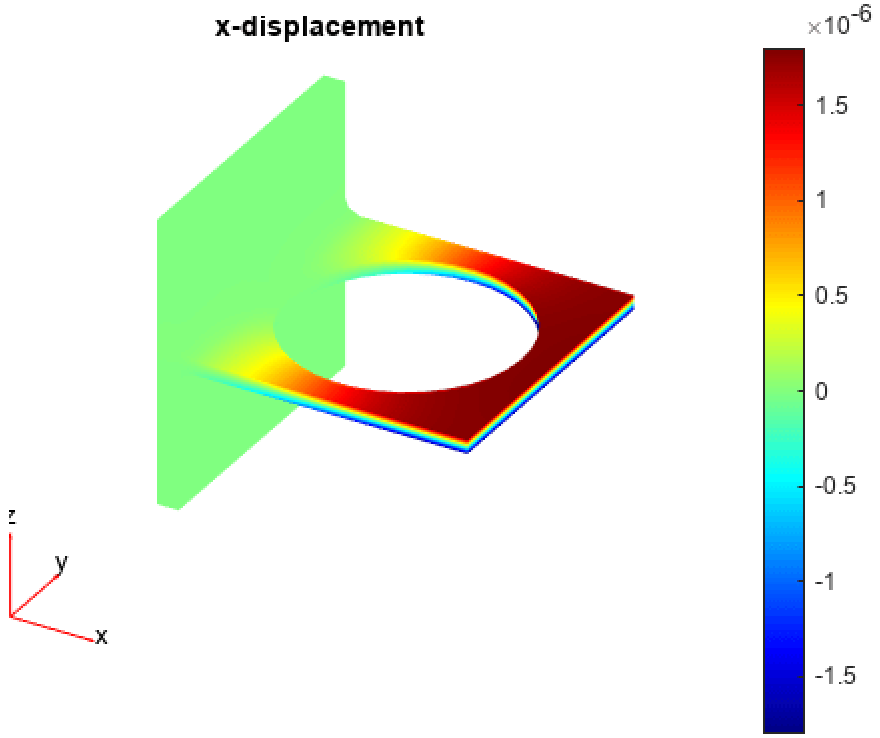
Transparency: None Medium High



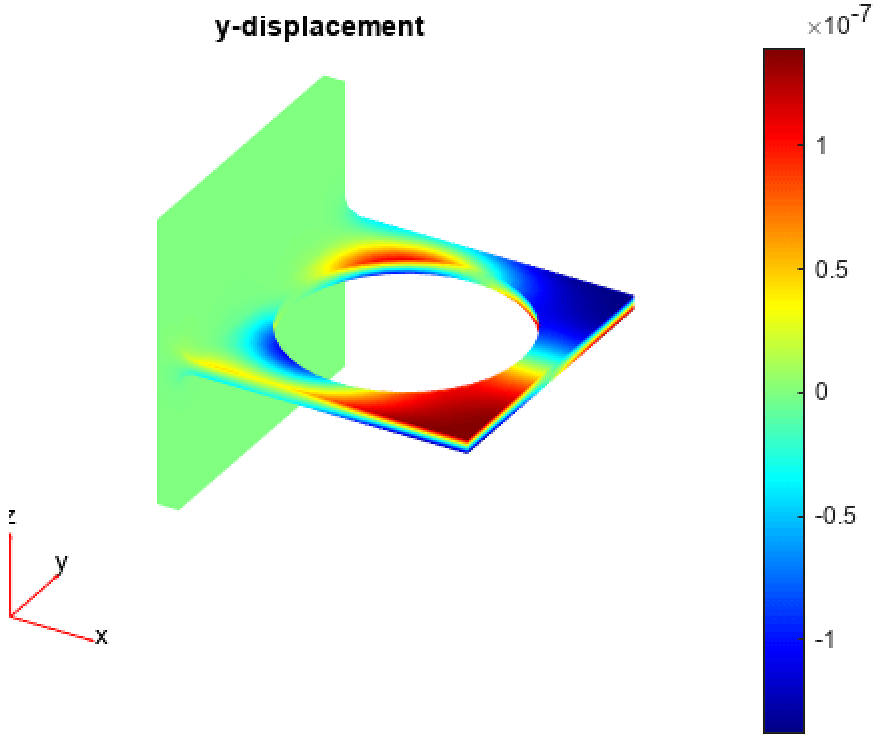
Plot Results at the Command Line

You also can plot the results, such as the displacement components and the von Mises stress, at the MATLAB® command line by using the `pdeplot3D` function.

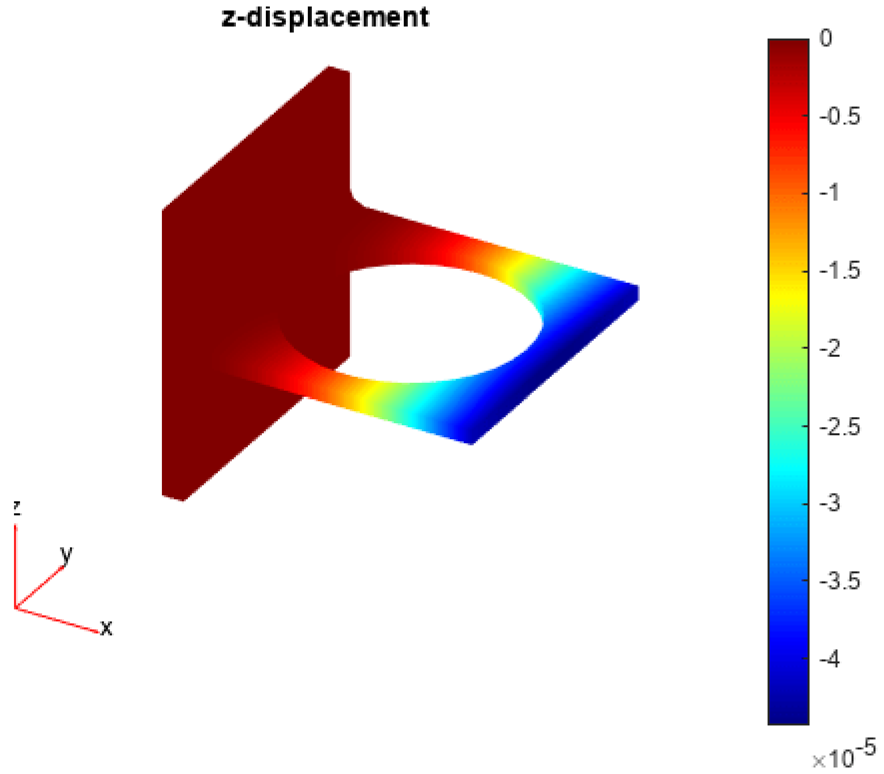
```
figure
pdeplot3D(model,"ColorMapData",result.Displacement.ux)
title("x-displacement")
colormap("jet")
```



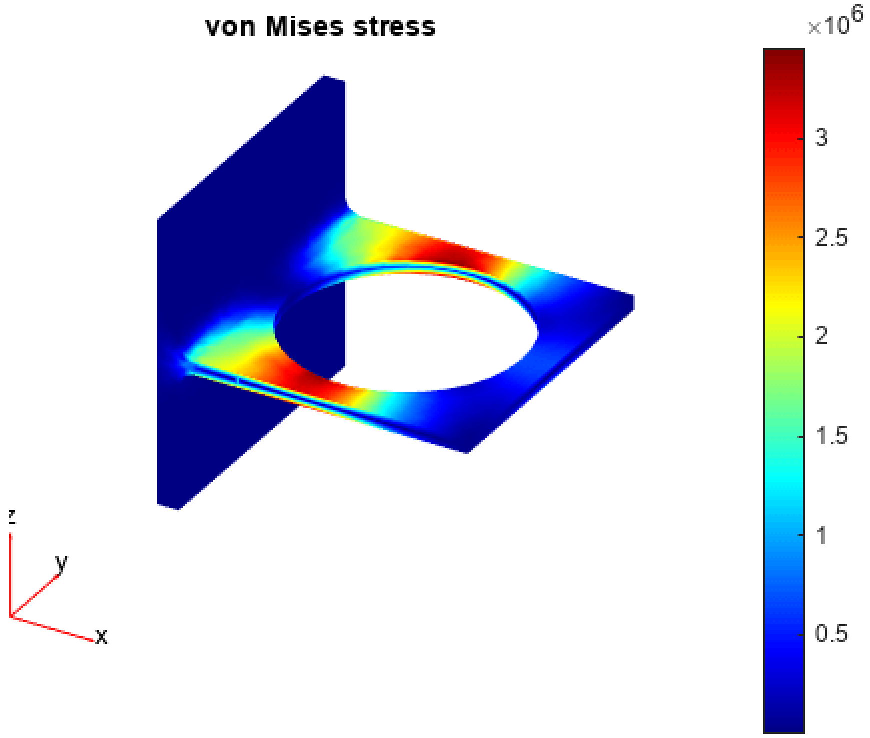
```
figure
pdeplot3D(model,"ColorMapData",result.Displacement.uy)
title("y-displacement")
colormap("jet")
```



```
figure
pdeplot3D(model,"ColorMapData",result.Displacement.uz)
title("z-displacement")
colormap("jet")
```



```
figure
pdeplot3D(model,"ColorMapData",result.VonMisesStress)
title("von Mises stress")
colormap("jet")
```


von Mises stress
×10^6
3
2.5
2
1.5
1
0.5
z
y
x

Heat Transfer in Block with Cavity

This example shows how to solve for the heat distribution in a block with cavity.

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.

Create Thermal Analysis Model

The first step in solving a heat transfer problem is to create a thermal analysis model. This is a container that holds the geometry, thermal material properties, internal heat sources, temperature on the boundaries, heat fluxes through the boundaries, mesh, and initial conditions.

```
thermalmodel = createpde("thermal","transient");
```

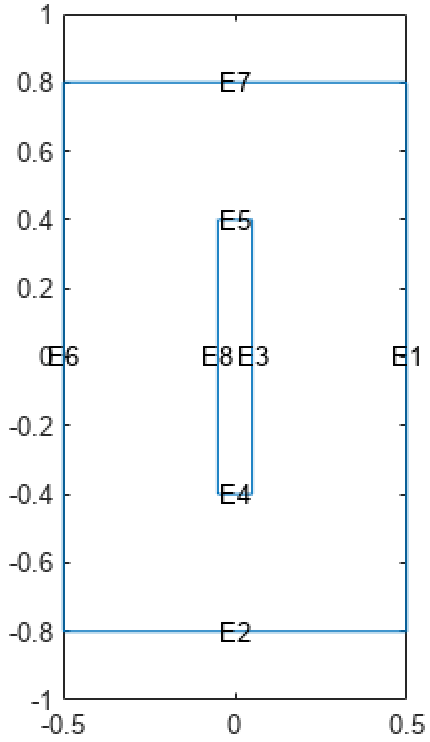
Import Geometry

Add the block geometry to the thermal model by using the `geometryFromEdges` function. The geometry description file for this problem is called `crackg.m`.

```
geometryFromEdges(thermalmodel,@crackg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
ylim([-1,1])
axis equal
```



Specify Thermal Properties of Material

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",1,...
                               "MassDensity",1,...
                               "SpecificHeat",1);
```

Apply Boundary Conditions

Specify the temperature on the left edge as `100`, and constant heat flow to the exterior through the right edge as `-10`. The toolbox uses the default insulating boundary condition for all other boundaries.

```
thermalBC(thermalmodel,"Edge",6,"Temperature",100);
thermalBC(thermalmodel,"Edge",1,"HeatFlux",-10);
```

Set Initial Conditions

Set an initial value of `0` for the temperature.

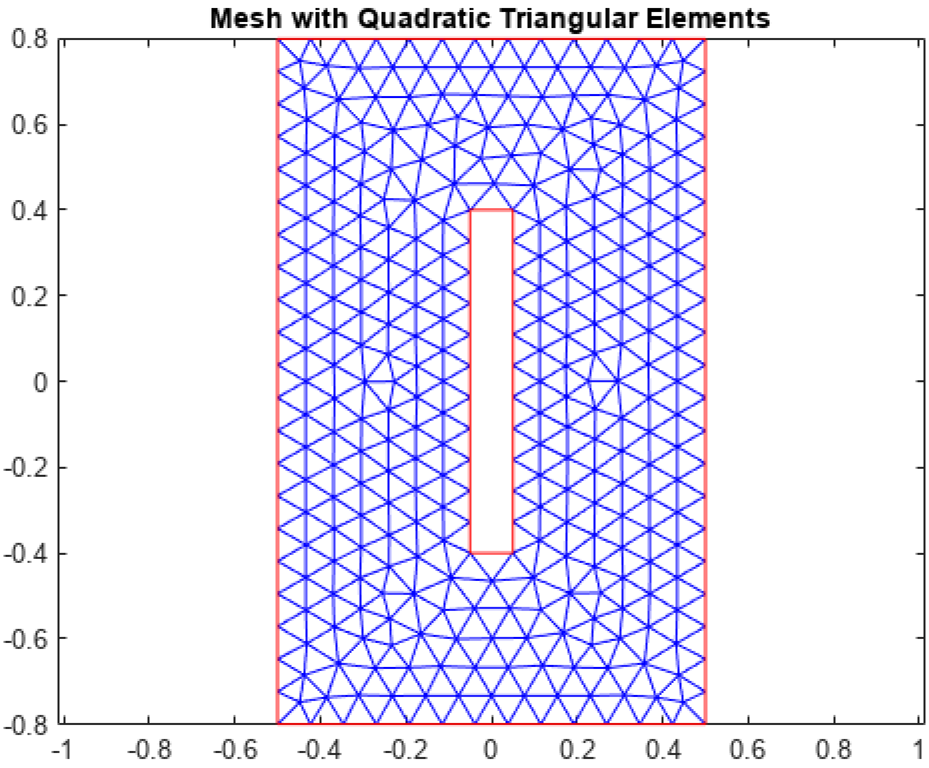
```
thermalIC(thermalmodel,0);
```

Generate Mesh

Create and plot a mesh.

```
generateMesh(thermalmodel);
figure
```

```
pdemesh(thermalmodel)
title("Mesh with Quadratic Triangular Elements")
```



Specify Solution Times

Set solution times to be 0 to 5 seconds in steps of 1/2.

```
tlist = 0:0.5:5;
```

Calculate Solution

Use the `solve` function to calculate the solution.

```
thermalresults = solve(thermalmodel,tlist)
```

```
thermalresults = 
  TransientThermalResults with properties:

      Temperature: [1320x11 double]
    SolutionTimes: [0 0.5000 1 1.5000 2 2.5000 3 3.5000 4 4.5000 5]
       XGradients: [1320x11 double]
       YGradients: [1320x11 double]
       ZGradients: []
             Mesh: [1x1 FEMesh]
```

Evaluate Heat Flux

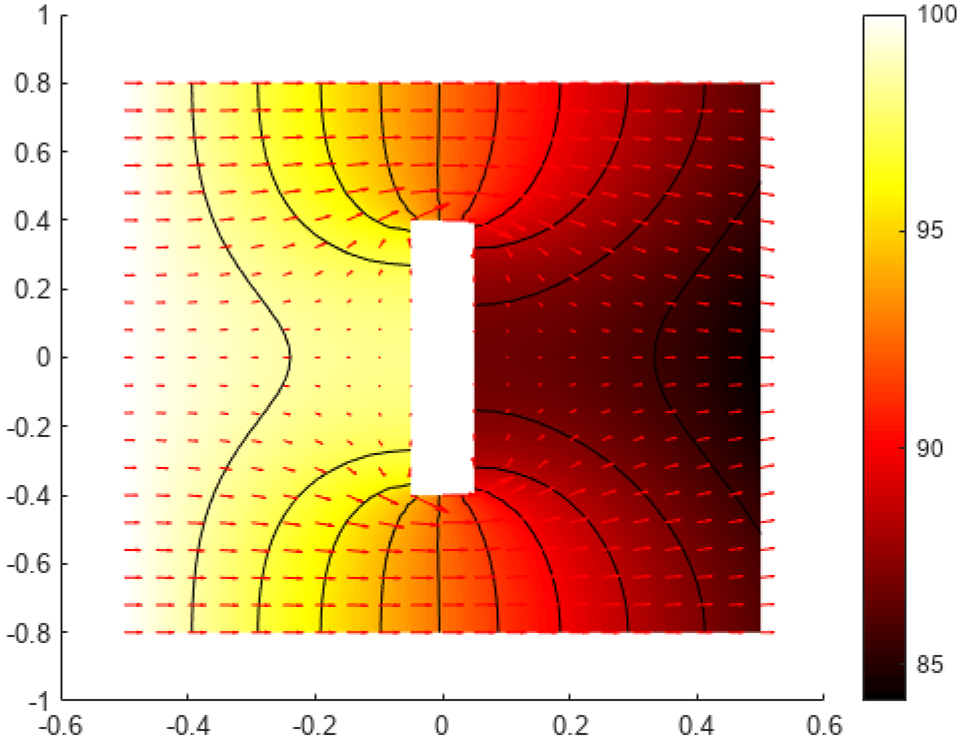
Compute the heat flux density.

```
[qx,qy] = evaluateHeatFlux(thermalresults);
```

Plot Temperature Distribution and Heat Flux

Plot the solution at the final time step, t = 5.0 seconds, with isothermal lines using a contour plot, and plot the heat flux vector field using arrows.

```
pdeplot(thermalmodel,"XYData",thermalresults.Temperature(:,end), ...
                     "Contour","on",...
                     "FlowData",[qx(:,end),qy(:,end)], ...
                     "ColorMap","hot")
```



Setting Up Your PDE

Solve Problems Using PDEModel Objects

1. Put your problem in the correct form for Partial Differential Equation Toolbox solvers. For details, see “Equations You Can Solve Using PDE Toolbox” on page 1-3. If you need to convert your problem to divergence form, see “Put Equations in Divergence Form” on page 2-87.
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 or create the geometry. For details, see “Geometry and Mesh”.

   ```
   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 “STL File Import” on page 2-43.
5. Create the boundary conditions. For details, see “Specify Boundary Conditions” on page 2-129.

   ```
   % "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));
   ```
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 2-90, “c Coefficient for specifyCoefficients” on page 2-92, and “m, d, or a Coefficient for specifyCoefficients” on page 2-107.
7. For time-dependent equations, or optionally for nonlinear stationary equations, create an initial condition. See “Set Initial Conditions” on page 2-114.
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 “Solution and Gradient Plots with pdeplot and pdeplot3D” on page 3-329, “2-D Solution and Gradient Plots with MATLAB Functions” on page 3-338, and “3-D Solution and Gradient Plots with MATLAB Functions” on page 3-344.

See Also

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

Geometry and Mesh Components

This example shows how the toolbox represents geometries and meshes, the components of geometries and meshes, and the relationships between them within a model object.

Geometry

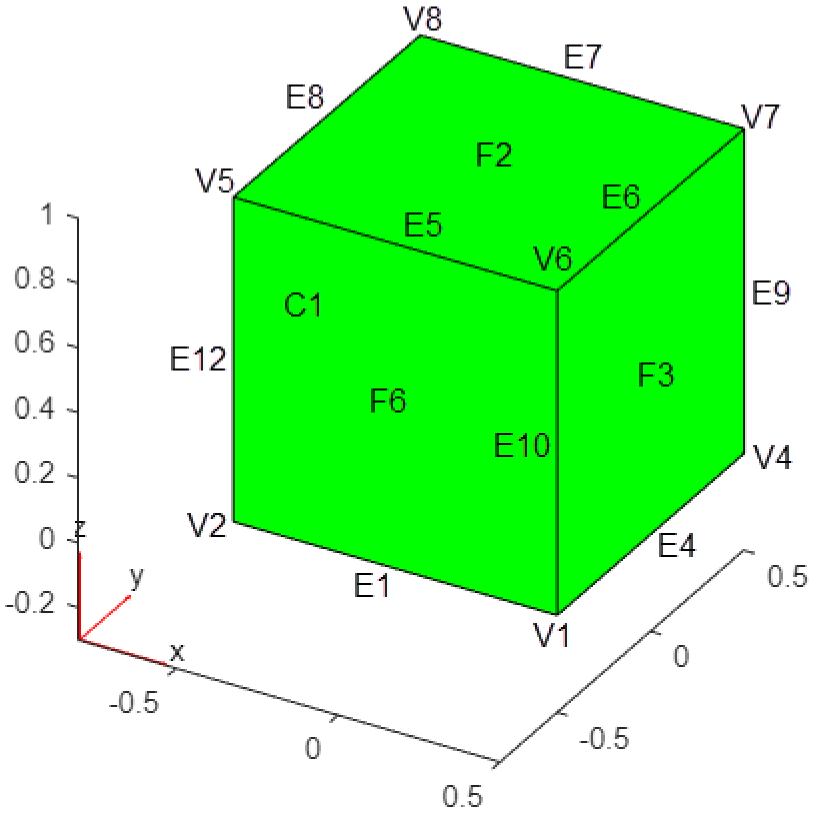
The toolbox supports 2-D and 3-D geometries. Each geometry in the toolbox consists of these components, also called *geometric regions*: vertices, edges, faces, and cells (for a 3-D geometry). Each geometric region has its own label that follows these conventions:

- Vertex labels — Letter `V` and positive integers starting from 1
- Edge labels — Letter `E` and positive integers starting from 1
- Face labels — Letter `F` and positive integers starting from 1
- Cell labels — Letter `C` and positive integers starting from 1

For example, the toolbox represents a unit cube geometry with these geometric regions and labels:

- Eight vertices labeled from `V1` to `V8`
- Twelve edges labeled from `E1` to `E12`
- Six faces labeled from `F1` to `F6`.
- One cell labeled `C1`

Numbering of geometric regions can differ in different releases. Always check that you are assigning parameters of a problem to the intended geometric regions by plotting the geometry and visually inspecting its regions and their labels.



To set up a PDE problem, the toolbox combines a geometry, mesh, PDE coefficients, boundary and initial conditions, and other parameters into a model object. A geometry can exist outside of a model. For example, create a unit sphere geometry.

```
gm1 = multisphere(1)
```

```
gm1 = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 1
       NumEdges: 0
    NumVertices: 0
       Vertices: []
```

You can also import a geometry.

```
gm2 = importGeometry("Block.stl")
```

```
gm2 = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 6
       NumEdges: 12
```

```
    NumVertices: 8
       Vertices: [8x3 double]
```

When a geometry exists within a model, the toolbox stores it in the `Geometry` property of the model object. For example, create a model and assign the unit sphere geometry `gm1` to its `Geometry` property.

```
model1 = createpde;
model1.Geometry = gm1
```

```
model1 = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

You also can import a geometry and assign it to the `Geometry` property of a model in one step by using `importGeometry`.

```
model2 = createpde;
importGeometry(model2,"Block.stl")
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 6
       NumEdges: 12
    NumVertices: 8
       Vertices: [8x3 double]
```

Mesh

A mesh approximates a geometry and consists of elements and nodes. The toolbox uses meshes with triangular elements for 2-D geometries and meshes with tetrahedral elements for 3-D geometries.

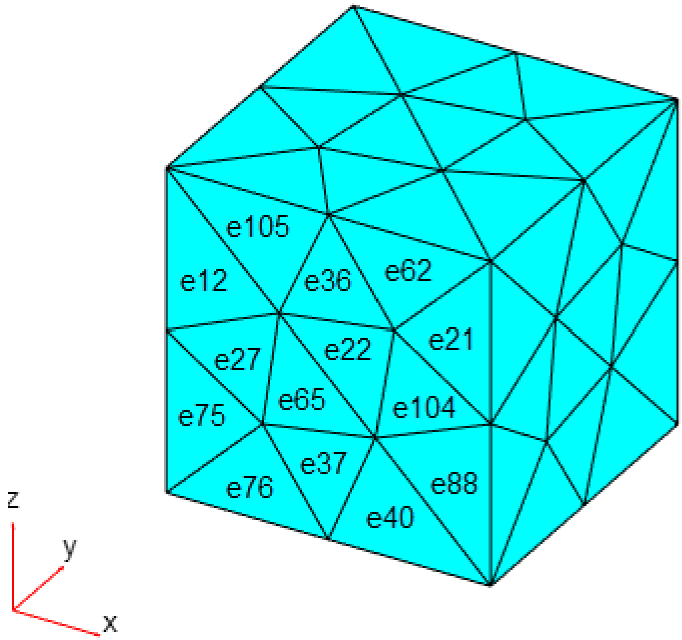
Triangular elements in 2-D meshes are specified by three nodes for linear elements or six nodes for quadratic elements. A triangle representing a linear element has nodes at the corners. A triangle representing a quadratic element has nodes at its corners and edge centers.

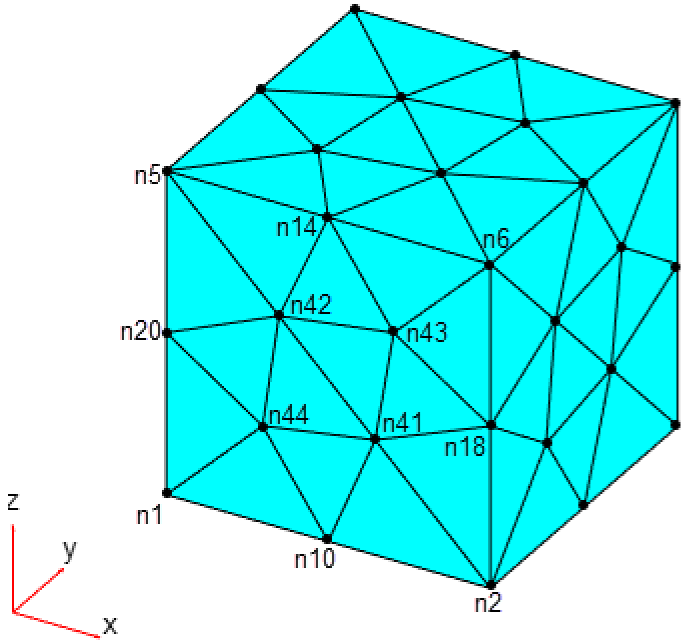
Tetrahedral elements in 3-D meshes are specified by four nodes for linear elements or 10 nodes for quadratic elements. A tetrahedron representing a linear element has nodes at the corners. A tetrahedron representing a quadratic element has nodes at its corners and edge centers.

Each mesh component has its own label that follows these conventions:

- Mesh element labels — Letter `e` and positive integers starting from 1
- Mesh node labels — Letter `n` and positive integers starting from 1

The mesh generator can return slightly different meshes in different releases. For example, the number of elements in the mesh can change. Write code that does not rely on explicitly specified node and element IDs or node and element counts.


e105
e12
e36
e62
e27
e22
e21
e75
e65
e104
e37
e88
e76
e40
z
y
x


n5
n14
n6
n42
n20
n43
n44
n41
n18
n1
n10
n2
z
y
x

Relationship Between Geometry and Mesh

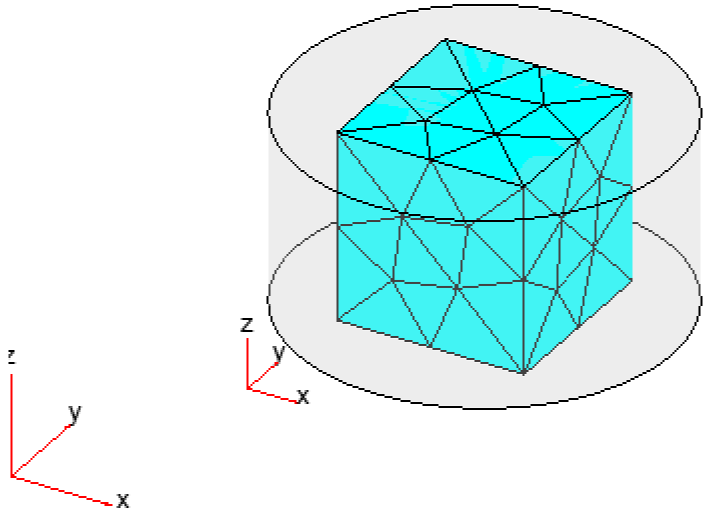
Geometric regions do not describe a mesh or its elements. A geometry can exist outside of a model, while a mesh is always a property of the model.

```
gm = multicuboid(1,1,1);
model = createpde;
model.Geometry = gm;
generateMesh(model,"Hmin",0.5);
model
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: [1x1 FEMesh]
           SolverOptions: [1x1 pde.PDESolverOptions]
```

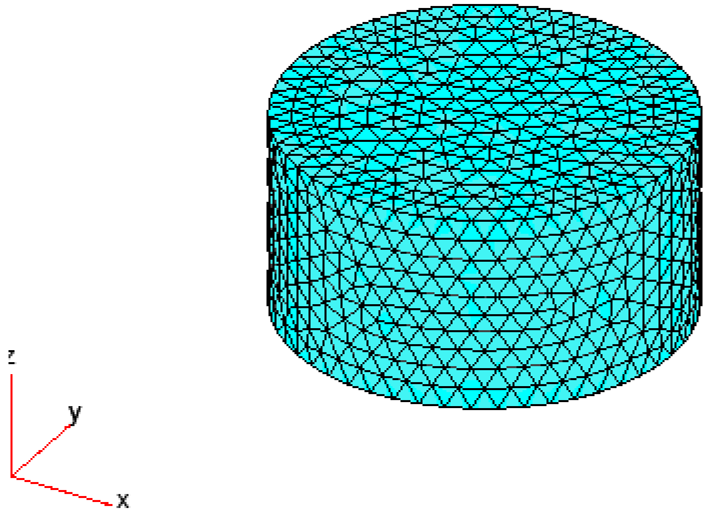
A geometry, even when it is a property of a model, is stored separately from a mesh. The toolbox does not automatically regenerate the mesh when you modify a geometry.

```
new_gm = multicylinder(1,1);
model.Geometry = new_gm;
pdegplot(model,"FaceAlpha",0.3)
hold on
pdemesh(model)
```



You must explicitly update the mesh to correspond to the current cylinder geometry.

```
generateMesh(model);
pdegplot(model,"FaceAlpha",0.3)
hold on
pdemesh(model)
```



Geometry and Mesh Queries

The toolbox enables you to find mesh elements and nodes by their geometric location or proximity to a particular point or node. For example, you can find all elements that belong to a particular face or cell. You also can find all nodes that belong to a particular vertex, edge, face, or cell. For details, see `findElements` and `findNodes`.

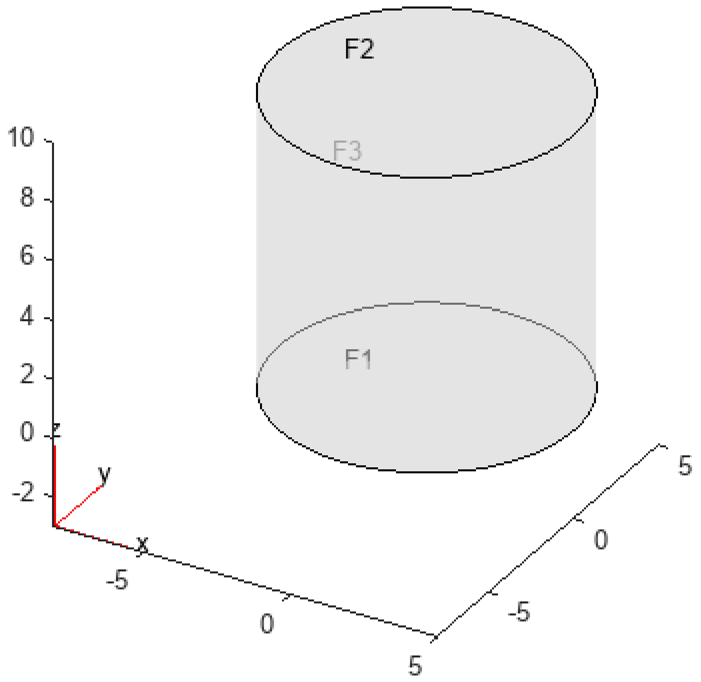
The toolbox also enables you to find edges and faces by their proximity to a particular point or to find only those attached to a particular geometric region:

- `cellEdges` finds edges belonging to boundaries of specified cells.
- `cellFaces` finds faces belonging to specified cells.
- `faceEdges` finds edges belonging to specified faces.
- `facesAttachedToEdges` finds faces attached to specified edges.
- `nearestEdge` finds edges nearest to specified points.
- `nearestFace` finds faces nearest to specified points.

Parameters of a Model on Geometric Regions

The toolbox lets you specify parameters of each problem, such as boundary and initial conditions (including boundary constraints and boundary loads) and PDE coefficients (including properties of materials, internal heat sources, body loads, and electromagnetic sources) on geometric regions. For example, you can apply boundary conditions on the top and bottom faces of this cylinder. First, create the cylinder geometry.

```
gm = multicylinder(5,10);
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.5);
```



Then, create a model, add the geometry to the model, and apply the Dirichlet boundary conditions on the top and bottom faces of the cylinder.

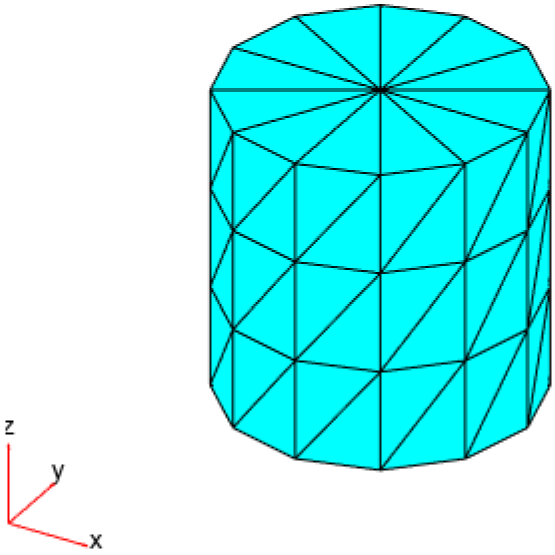
```
model = createpde;
model.Geometry = gm;
applyBoundaryCondition(model,"dirichlet","Face",1:2,"u",0)
```

```
ans = 
  BoundaryCondition with properties:

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

Solvers Use Meshes

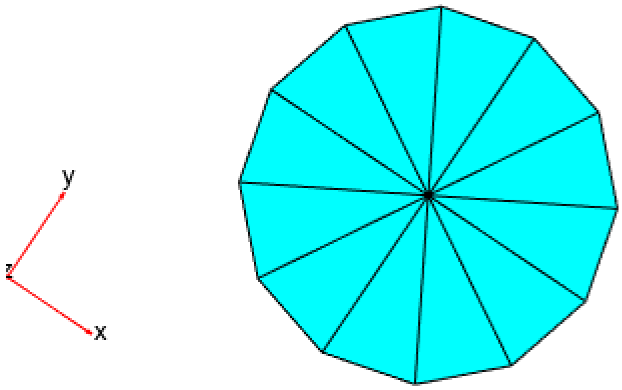
PDE solvers do not work with geometries directly. They work with the corresponding meshes instead. For example, if you generate a coarse mesh, the PDE solver uses the discretized cylinder.

```
generateMesh(model,"Hmin",4);
figure
pdemesh(model)
```



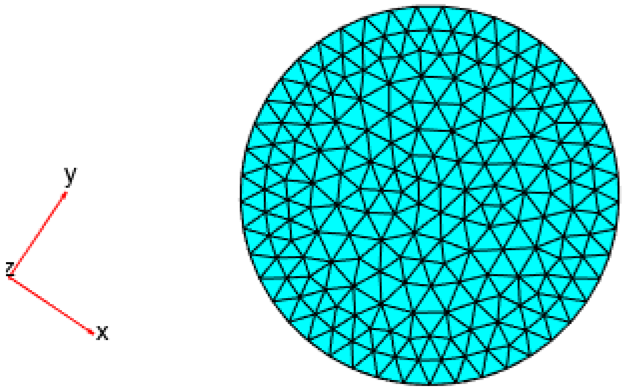
When you solve a problem, the toolbox internally finds all mesh nodes and elements that belong to these geometric regions and applies the specified parameters to those nodes and elements. The discretized top and bottom of the cylinder look like polygons rather than circles.

```
figure
pdemesh(model)
view([34 90])
```



When you refine a mesh for your problem, the toolbox automatically recalculates which nodes and elements belong to particular geometric regions and applies the specified parameters to the new nodes and elements.

```
generateMesh(model);

figure
pdemesh(model)
view([34 90])
```



Although the solvers apply specified parameters to mesh elements and nodes, you cannot explicitly specify these parameters directly on mesh components. All parameters must be specified on geometric regions. This approach prevents unintended assignments that can happen, for example, when you refine a mesh.

2-D Geometry Creation at Command Line

Three Elements of Geometry

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

1. A matrix whose columns describe the basic shapes. When you export geometry from the PDE Modeler app, this matrix has the default name `gd` (geometry description).
2. 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. A set of characters describing the unions, intersections, and set differences of the basic shapes that make the geometry.

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 must not contain any self-intersections.

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

Rectangle with Circular End Cap and Another Circular Excision

Specify a matrix that has a rectangle with a circular end cap and another circular excision.

Create Basic Shapes

First, 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. This function pads the rows as needed so all have the same length. Continuing the example, 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';
```

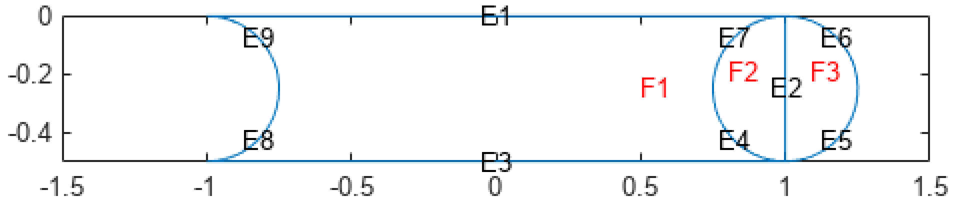
Create Geometry and Remove Face Boundaries

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, combine the basic shapes using the set formula.

```
[dl,bt] = decsg(gd,sf,ns);
```

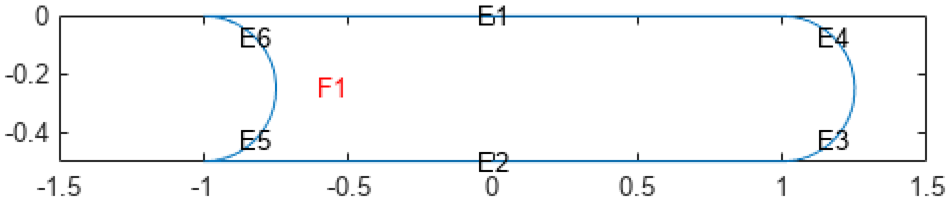
View the geometry with and without boundary removal.

```
pdegplot(dl,"EdgeLabels","on","FaceLabels","on")
xlim([-1.5,1.5])
axis equal
```



Remove the face boundaries.

```
[dl2,bt2] = csgdel(dl,bt);
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 10 rows. The extra three 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
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)

Row	Circle	Line	Ellipse
`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`	Radius	0	Length of first semiaxis
`11`	0	0	Length of second semiaxis
`12`	0	0	Angle in radians between *x* axis and first semiaxis

Parametrized Function for 2-D Geometry Creation

Required Syntax

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 must be proportional to the arc length plus a constant.

You must specify at least two curves for each geometric region. For example, the `'circleg'` geometry function, which is available in Partial Differential Equation 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 data described in this table.

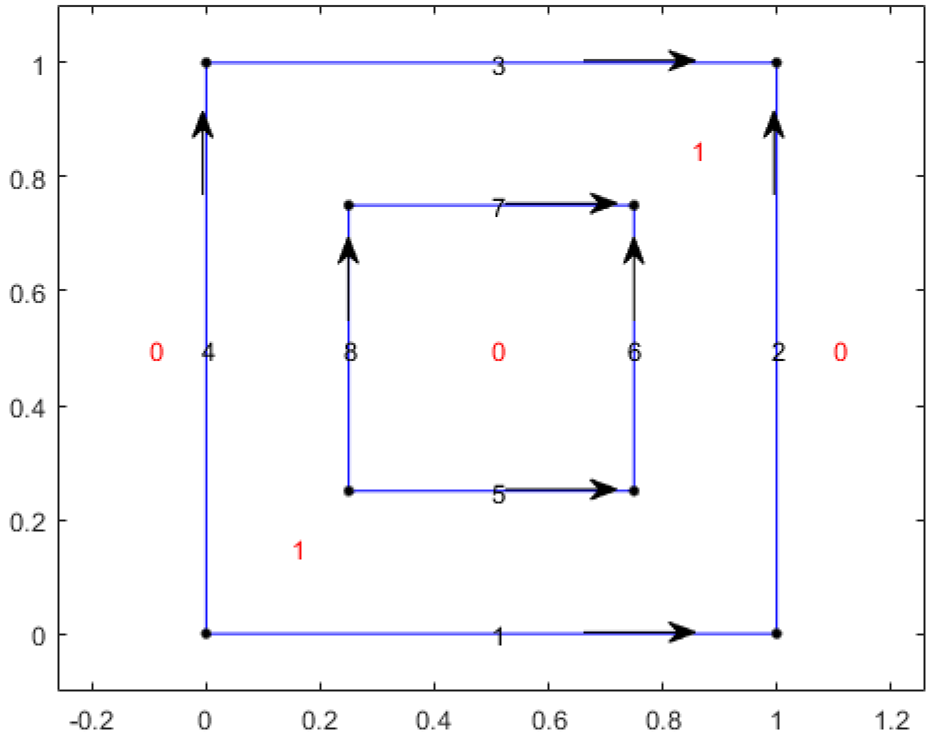
Number of Input Arguments	Returned Data
0 (`ne = pdegeom`)	`ne` is the number of edges in the geometry.
1 (`d = pdegeom(bs)`)	`bs` is a vector of edge segments. Your function returns `d` as a matrix with one column for each edge segment specified in `bs`. The rows of `d` are: **1** Start parameter value **2** End parameter value **3** Left region label, where “left” is with respect to the direction from the start to the end parameter value **4** Right region label A region label is the same as a subdomain number. The region label of the exterior of the geometry is `0`.
2 (`[x,y] = pdegeom(bs,s)`)	`s` is an array of arc lengths, and `bs` is a scalar or an array of the same size as `s` that gives the 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 the parameter value `s`. The `x` and `y` arrays have the same size as `s`.

Relation Between Parametrization and Region Labels

The following figure shows how the direction of parameter increase relates to label numbering. The arrows in the 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.



Geometry Function for a Circle

This example shows how to write a geometry function for creating a circular region. Parametrize a circle with radius 1 centered at the origin `(0,0)`, as follows:

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

A geometry function must have at least two segments. To satisfy this requirement, break up the circle into four segments.

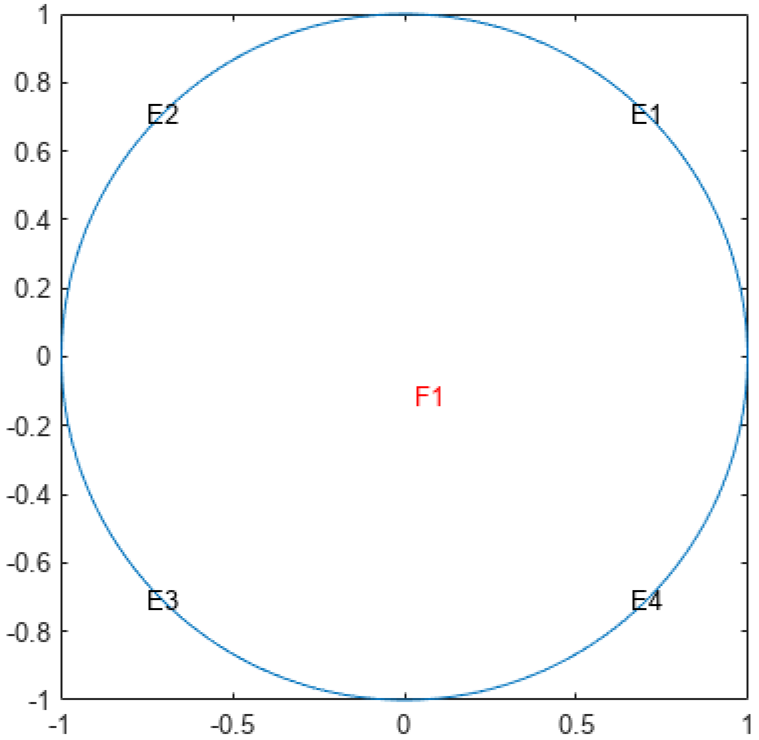
- $0 \le t \le \pi/2$
- $\pi/2 \le t \le \pi$
- $\pi \le t \le 3\pi/2$
- $3\pi/2 \le t \le 2\pi$

Now that you have a parametrization, write the geometry function. Save this function file as `circlefunction.m` on your MATLAB® path. This geometry is simple to create because the parametrization does not change depending on the segment number.

```
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
```

Plot the geometry displaying the edge numbers and the face label.

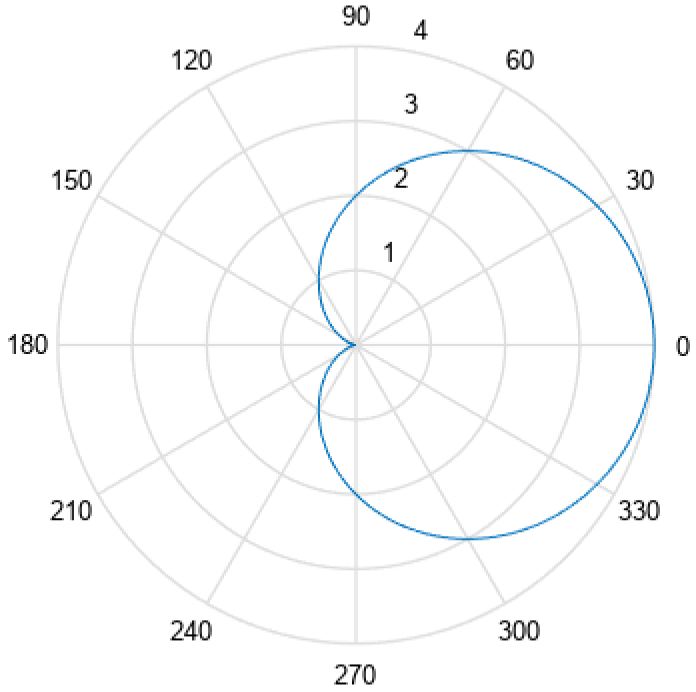
```
pdegplot(@circlefunction,"EdgeLabels","on","FaceLabels","on")
axis equal
```



Arc Length Calculations for a Geometry Function

This example shows how to create a cardioid geometry using four distinct techniques. The techniques are ways to parametrize your geometry using arc length calculations. The cardioid satisfies the equation $r = 2(1 + \cos(\Phi))$.

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



r = 2 (1+cos(Φ))

The following are the four ways to parametrize the cardioid as a function of the 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 requiring you to choose an arbitrary polygon.
- Use an analytic calculation of the arc length. This approach is the best when it applies, but there are many cases where it does not apply.
- Use a parametrization that is not proportional to the arc length plus a constant. This approach is the simplest, but can yield a distorted mesh that does not give the most accurate solution to your PDE problem.

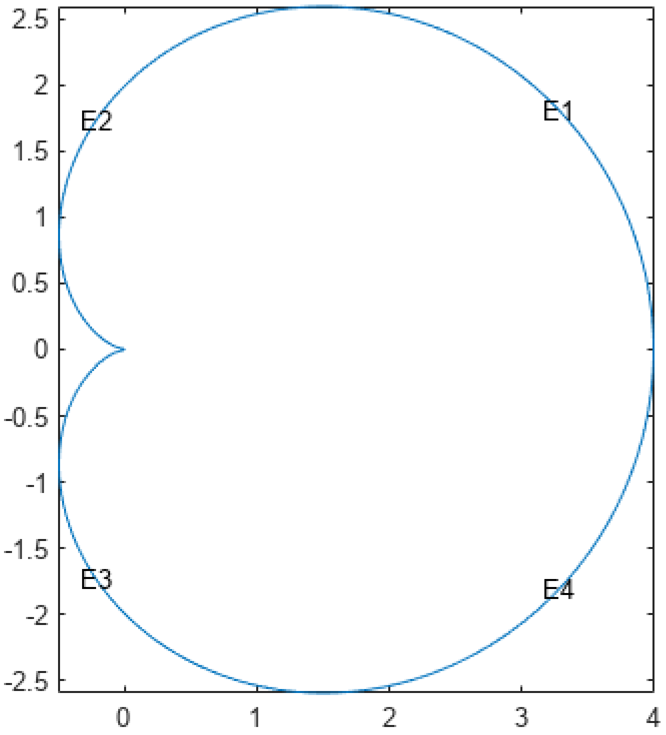
Polygonal Approximation

The finite element method uses a triangular mesh to approximate the solution to a PDE numerically. You can avoid 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. Write the following 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 arc length $s(t)$ is

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

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

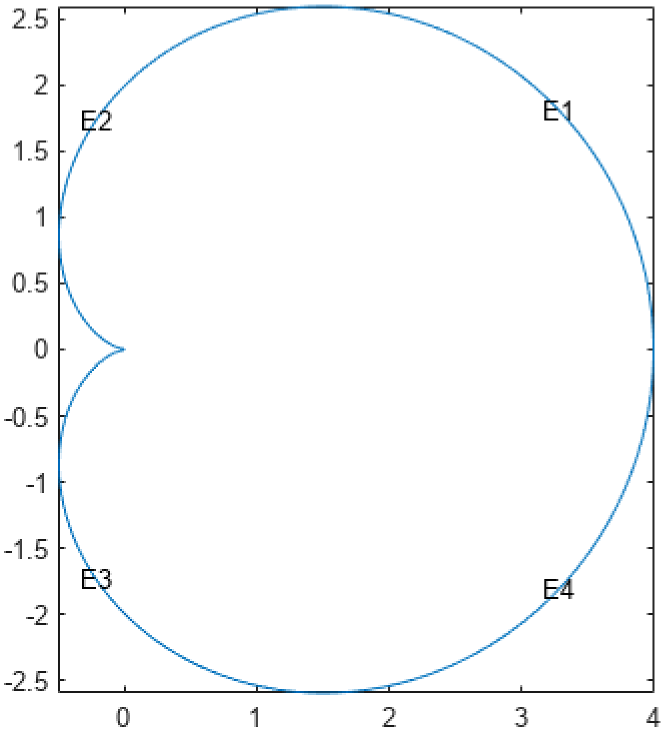
Write the following geometry function for the cardioid example.

```
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 cardioid
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 displaying the edge labels.

```
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

You also 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. For example, find an analytic expression for the arc length by 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 =
```

$$8\sin\left(\frac{t}{2}\right)$$

In terms of the arc length `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 of the cardioid, `t = pi + 2*asin((16-s)/8)`. Furthermore, you can express `x` and `y` in terms of `s` by these analytic calculations.

```
syms s real
th = 2*asin(s/8);
```

```
r = 2*(1 + cos(th));
r = expand(r)
```

```
r =
```

$$4-\frac{s^2}{16}$$

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

```
x =
```

$$\frac{s^4}{512}-\frac{3\,s^2}{16}+4$$

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

```
y =
```

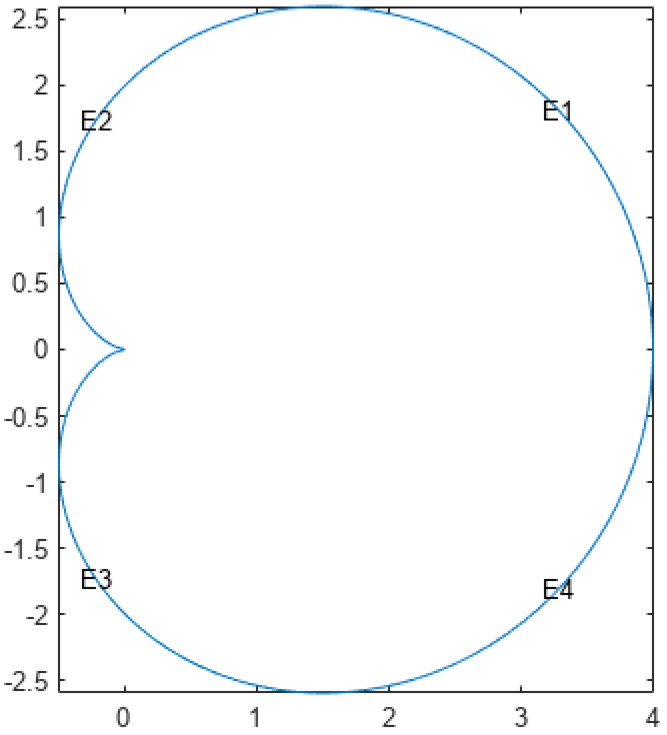
$$\frac{s\left(64-s^2\right)^{3/2}}{512}$$

Now that you have analytic expressions for `x` and `y` in terms of the arc length `s`, 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   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 cardioid
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 displaying the edge labels.

```
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

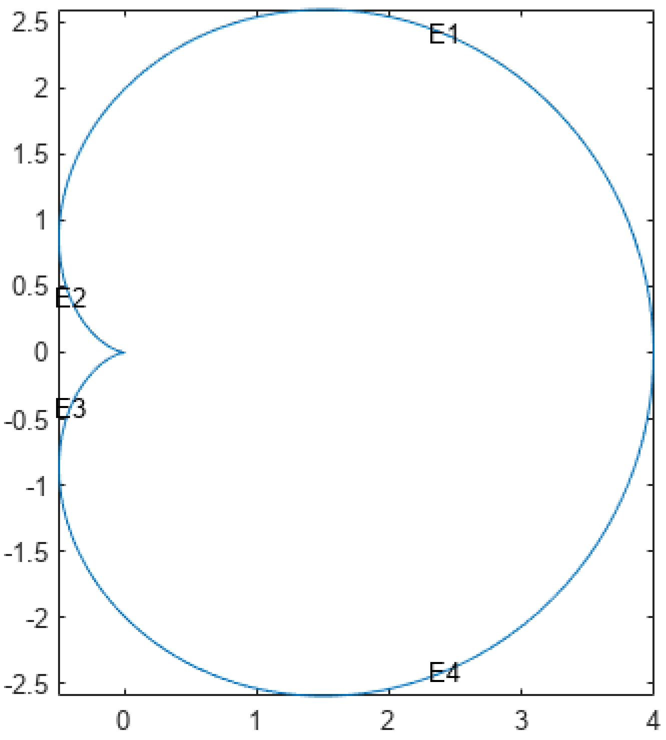
You also can write a geometry function where the parameter is not proportional to the arc length. This approach can yield a distorted mesh.

```
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 displaying the edge labels.

```
pdegplot("cardioid4","EdgeLabels","on")
axis equal
```

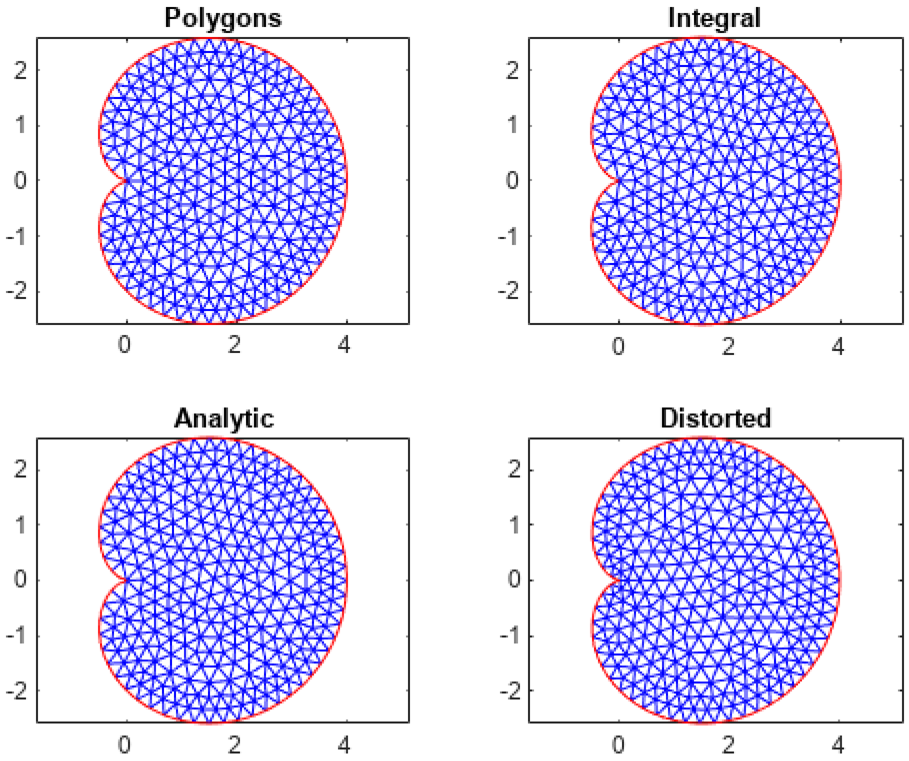


The labels are not evenly spaced on the edges because the parameter is not proportional to the arc length.

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

```
subplot(2,2,1)
model = createpde;
geometryFromEdges(model,@cardioid1);
generateMesh(model);
pdeplot(model)
title("Polygons")
axis equal

subplot(2,2,2)
model = createpde;
geometryFromEdges(model,@cardioid2);
generateMesh(model);
pdeplot(model)
```

```
title("Integral")
axis equal

subplot(2,2,3)
model = createpde;
geometryFromEdges(model,@cardioid3);
generateMesh(model);
pdeplot(model)
title("Analytic")
axis equal

subplot(2,2,4)
model = createpde;
geometryFromEdges(model,@cardioid4);
generateMesh(model);
pdeplot(model)
title("Distorted")
axis equal
```



The distorted mesh looks a bit less regular than the other meshes. It has some very narrow triangles near the cusp of the cardioid. Nevertheless, all of the meshes appear to be usable.

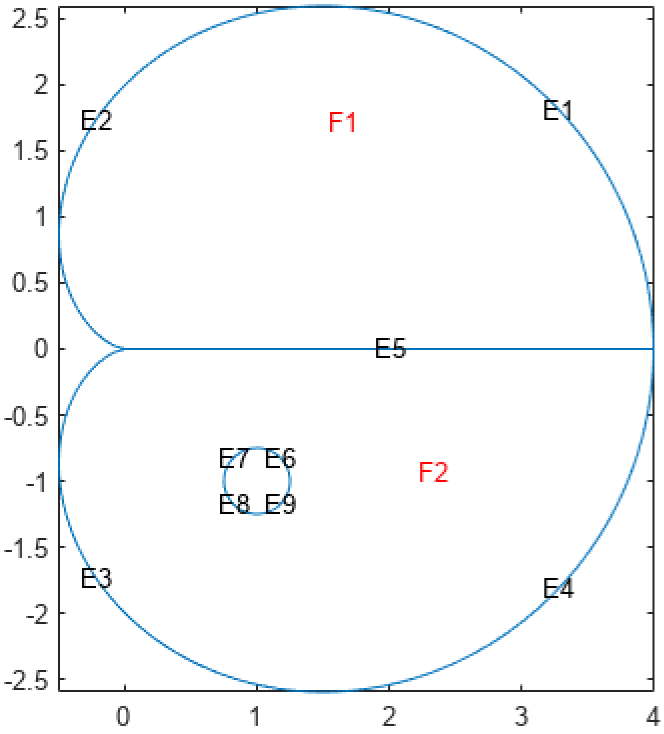
Geometry Function Example with Subdomains and a Hole

This example shows how to create a geometry file for a region with subdomains and a hole. It uses the "Analytic Arc Length" section of the "Arc Length Calculations for a Geometry Function" example 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. The following 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
       % Region 1 to the left in
       % the upper half, 2 in the lower
       1   1   2   2
       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 % Empty to the left
        2      2      2        2]; % Region 2 to the right
    % 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 cardioid
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", ...
                 "FaceLabels","on")
axis equal
```



Nested Function for Geometry with Additional Parameters

This example shows how to include additional parameters into a function for creating a 2-D geometry.

When a 2-D geometry function requires additional parameters, you cannot use a standard anonymous function approach because geometry functions return a varying number of arguments. Instead, you can use global variables or nested functions. In most cases, the recommended approach is to use nested functions.

The example solves a Poisson's equation with zero Dirichlet boundary conditions on all boundaries. The geometry is a cardioid with an elliptic hole that has a center at (1,-1) and variable semiaxes. To set up and solve the PDE model with this geometry, use a nested function. Here, the parent function accepts the lengths of the semiaxes, `rr` and `ss`, as input parameters. The reason to nest `cardioidWithEllipseGeom` within `cardioidWithEllipseModel` is that nested functions share the workspace of their parent functions. Therefore, the `cardioidWithEllipseGeom` function can access the values of `rr` and `ss` that you pass to `cardioidWithEllipseModel`.

```
function cardioidWithEllipseModel(rr,ss)
```

```
if (rr > 0) & (ss > 0)
  model = createpde();
```

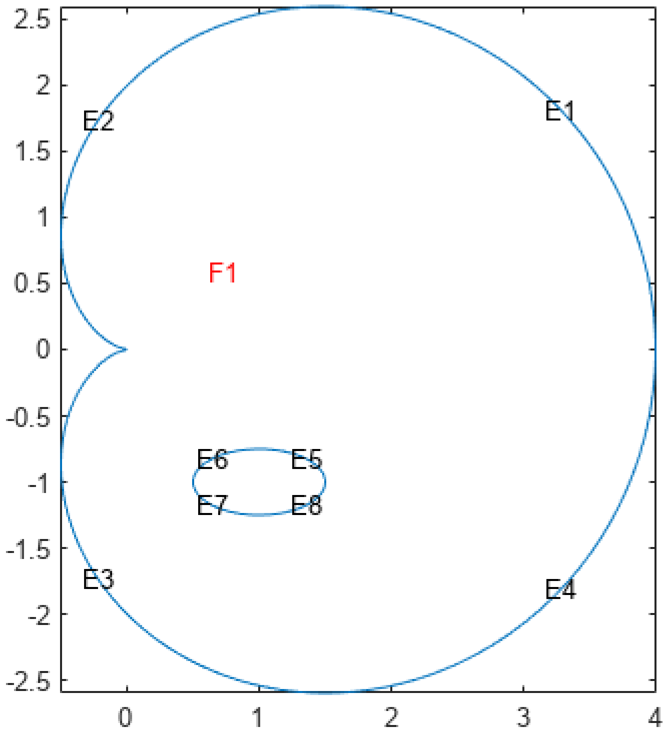
```
    geometryFromEdges(model,@cardioidWithEllipseGeom);
    pdegplot(model,"EdgeLabels","on","FaceLabels","on")
    axis equal

    applyBoundaryCondition(model,"dirichlet","Edge",1:8,"u",0);
    specifyCoefficients(model,"m",0,"d",0,"c",1,"a",0,"f",1);

    generateMesh(model);
    u = solvepde(model);
    figure
    pdeplot(model,"XYData",u.NodalSolution)
    axis equal

  else
      display("Semiaxes values must be positive numbers.")
  end

  function [x,y] = cardioidWithEllipseGeom(bs,s)

  if nargin == 0
   x = 8; % eight segments in boundary
   return
  end

  if nargin == 1
      % Cardioid
      dlc = [ 0   4   8  12
              4   8  12  16
              1   1   1   1
              0   0   0   0];
      % Ellipse
      dle = [0      pi/2   pi       3*pi/2
             pi/2   pi     3*pi/2   2*pi
             0      0      0        0
             1      1      1        1];
      % Combine the edge matrices
      dl = [dlc,dle];
      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 cardioid
  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 > 4); % Inner ellipse center (1,-1) axes rr and ss
  x(cbs) = 1 + rr*cos(s(cbs));
  y(cbs) = -1 + ss*sin(s(cbs));

  end
```

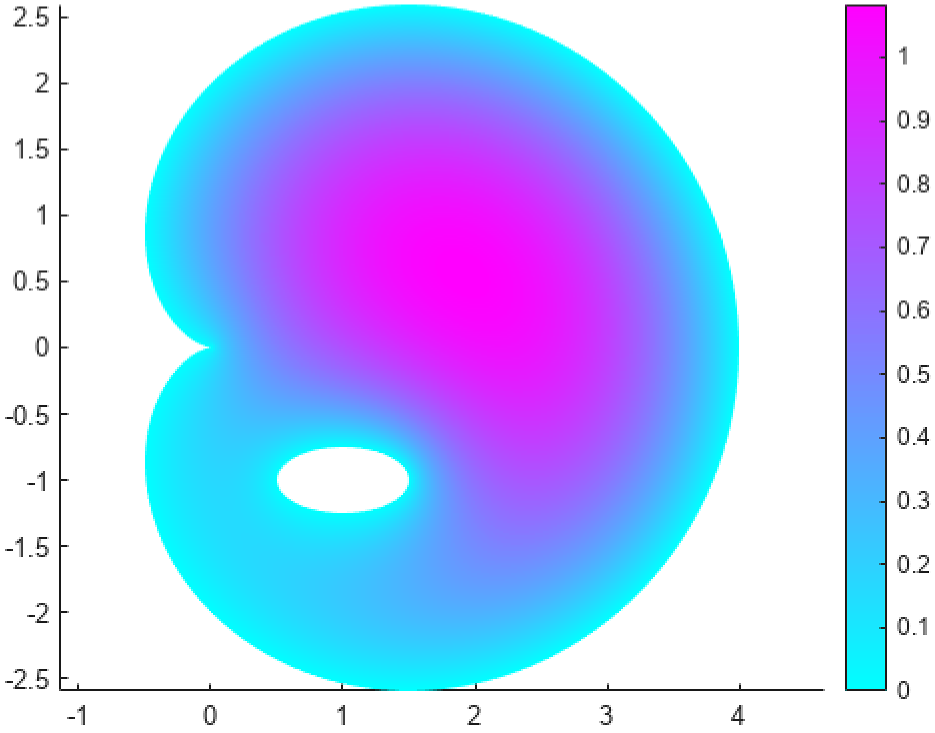
```
end
```

When calling `cardioidWithEllipseModel`, ensure that the semiaxes values are small enough, so that the elliptic hole appears entirely within the outer cardioid. Otherwise, the geometry becomes invalid.

For example, call the function for the ellipse with the major semiaxis `rr = 0.5` and the minor semiaxis `ss = 0.25`. This function call returns the following geometry and the solution.

```
cardioidWithEllipseModel(0.5,0.25)
```




2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-1
0
1
2
3
4
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

Geometry from polyshape

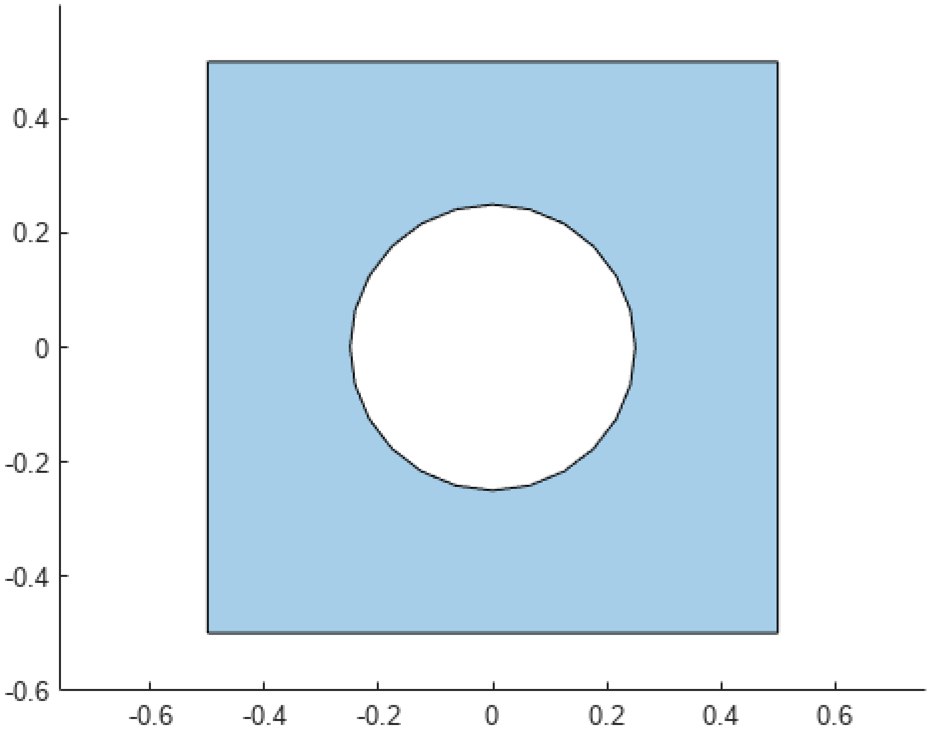
This example shows how to create a polygonal geometry using the MATLAB® `polyshape` function. Then use the triangulated representation of the geometry as an input mesh for the `geometryFromMesh` function.

Create and plot a `polyshape` object of a square with a hole.

```
t = pi/12:pi/12:2*pi;
pgon = polyshape({[-0.5 -0.5 0.5 0.5], 0.25*cos(t)}, ...
                 {[0.5 -0.5 -0.5 0.5], 0.25*sin(t)})
```

```
pgon = 
  polyshape with properties:

      Vertices: [29x2 double]
    NumRegions: 1
      NumHoles: 1
```

```
plot(pgon)
axis equal
```



Create a triangulation representation of this object.

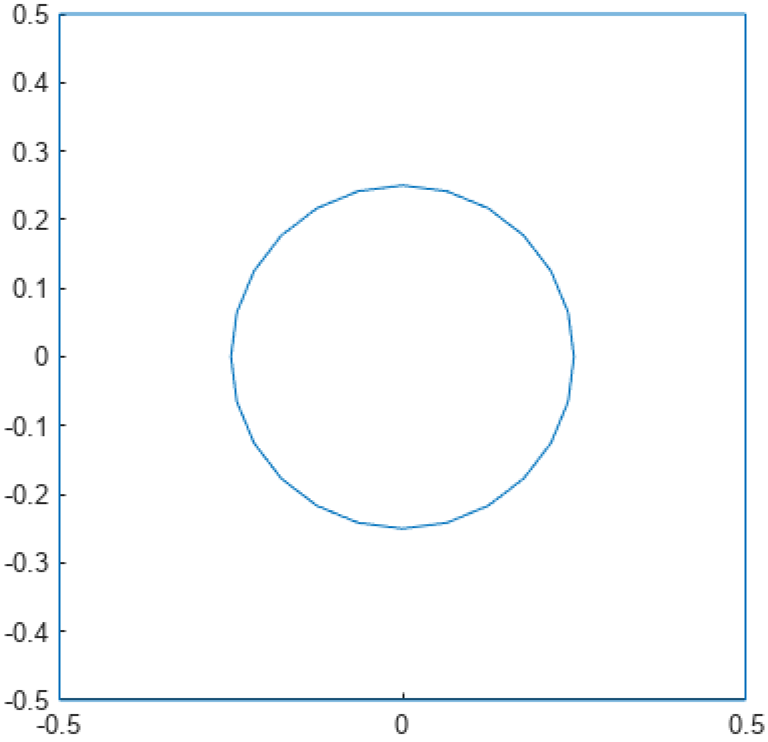
```
tr = triangulation(pgon);
```

Create a PDE model.

```
model = createpde;
```

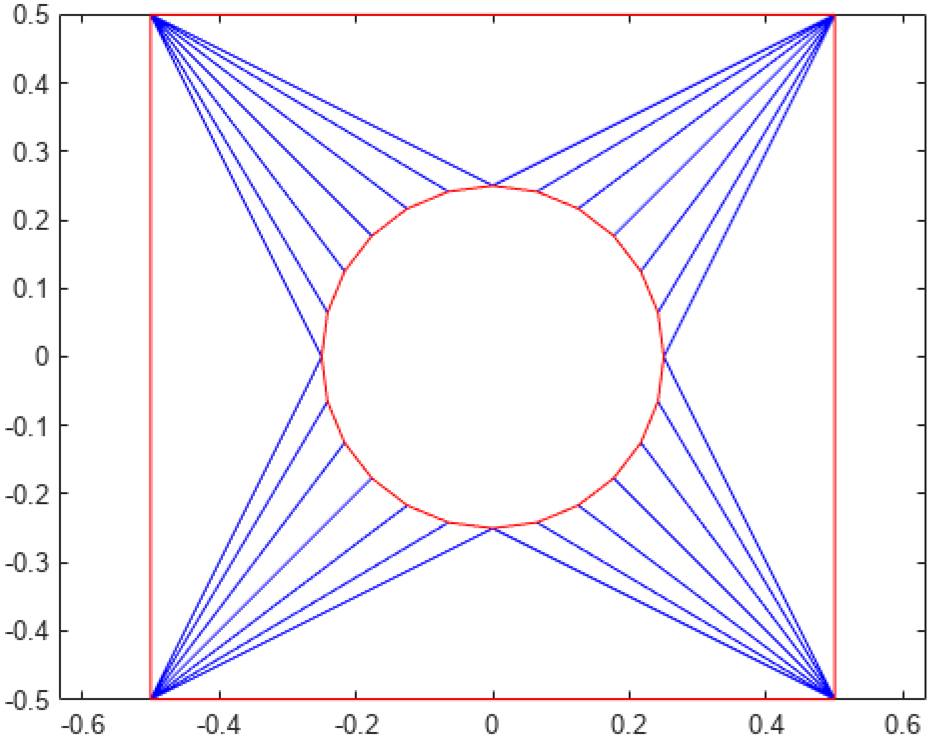
With the triangulation data as a mesh, use the `geometryFromMesh` function to create a geometry. Plot the geometry.

```
tnodes = tr.Points';
telements = tr.ConnectivityList';

geometryFromMesh(model,tnodes,telements);
pdegplot(model)
```



Plot the mesh.

```
figure
pdemesh(model)
```



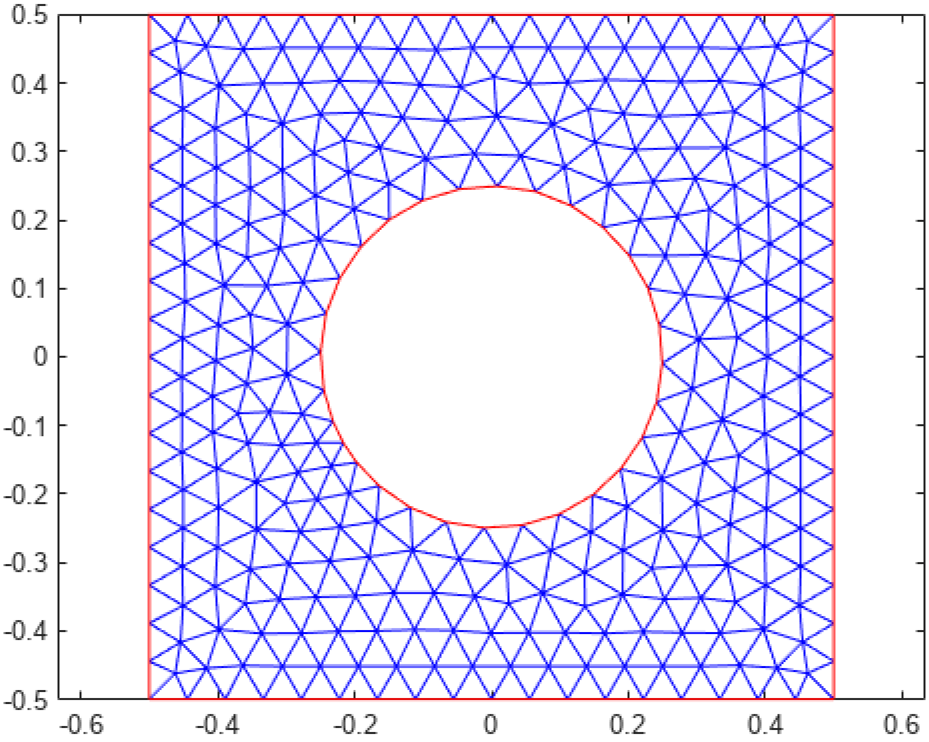
Because the triangulation data resulted in a low-quality mesh, generate a new finer mesh for further analysis.

```
generateMesh(model)
```

```
ans = 
  FEMesh with properties:

             Nodes: [2x1259 double]
          Elements: [6x579 double]
    MaxElementSize: 0.0566
    MinElementSize: 0.0283
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

Plot the mesh.

```
figure
pdemesh(model)
```

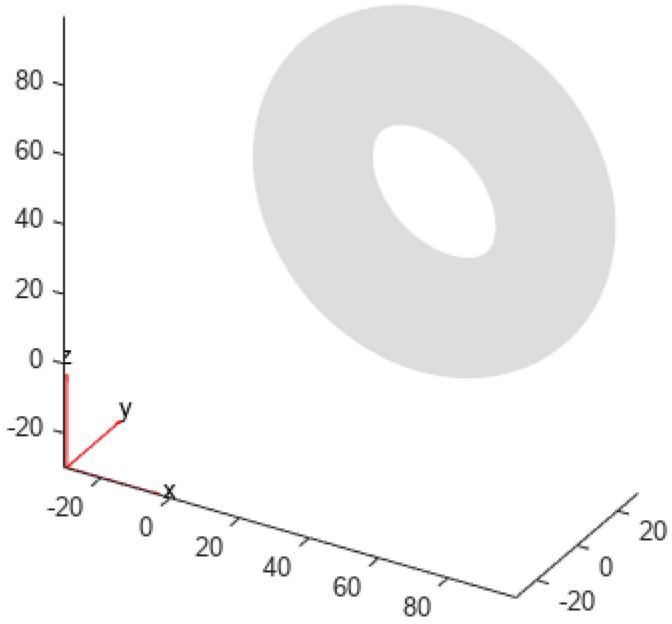

0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.4
-0.2
0
0.2
0.4
0.6

STL File Import

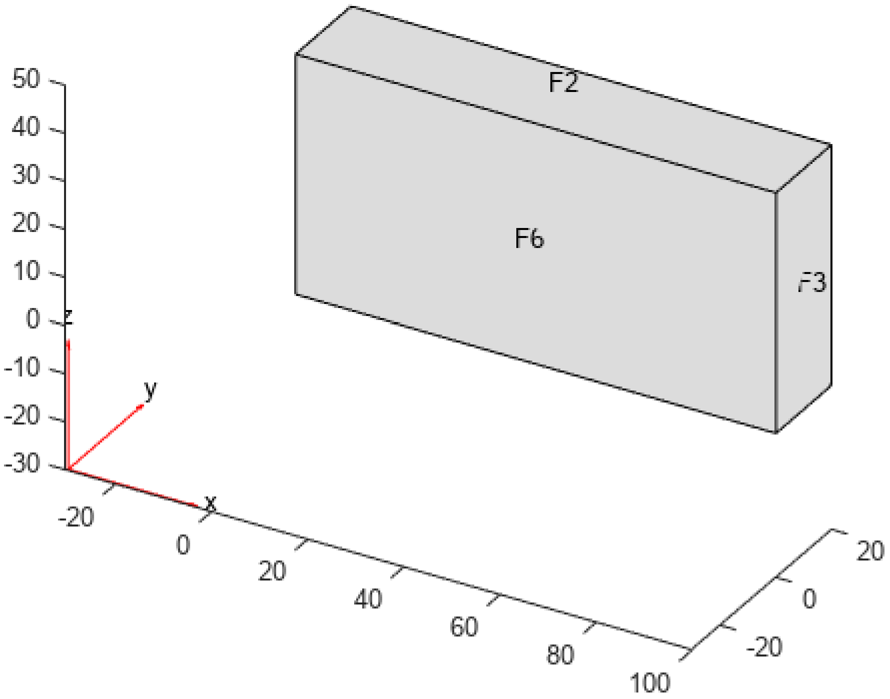
This example shows how to import a geometry from an STL file, and then plot the geometry. Generally, you create the STL file by exporting from a CAD system, such as SolidWorks®. For best results, export a fine (not coarse) STL file in binary (not ASCII) format. After importing, view the geometry using 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™.

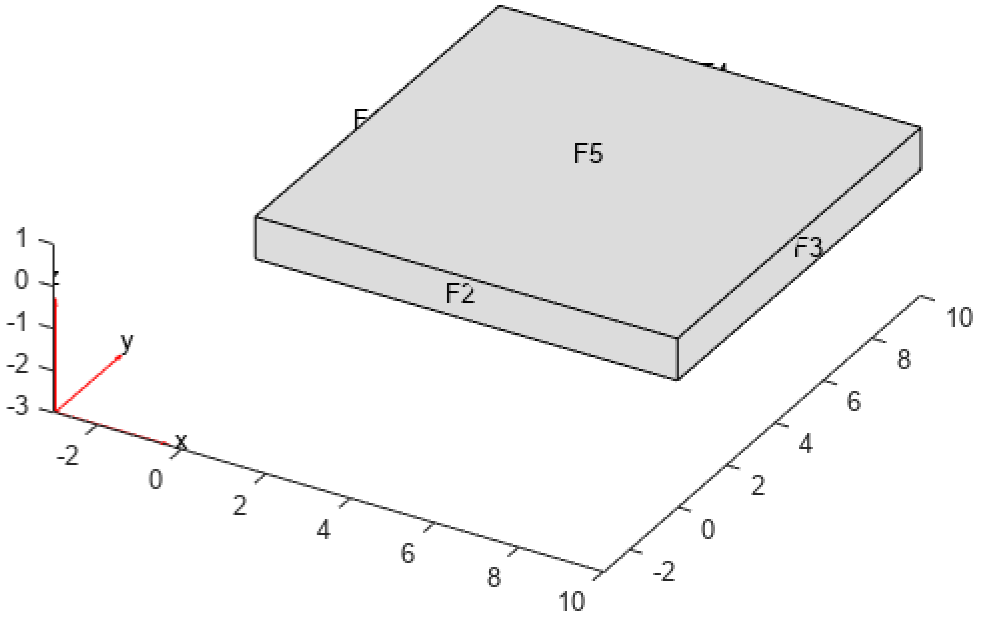
```
figure
gm = importGeometry("Torus.stl");
pdegplot(gm)
```



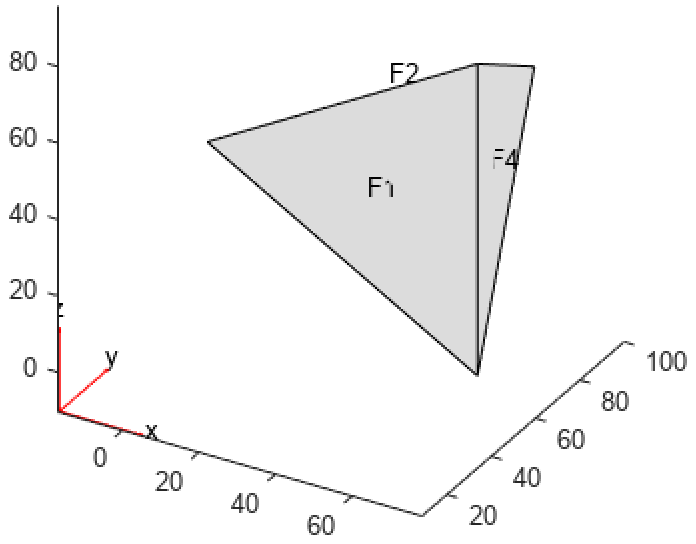
```
figure
gm = importGeometry("Block.stl");
pdegplot(gm,"FaceLabels","on")
```



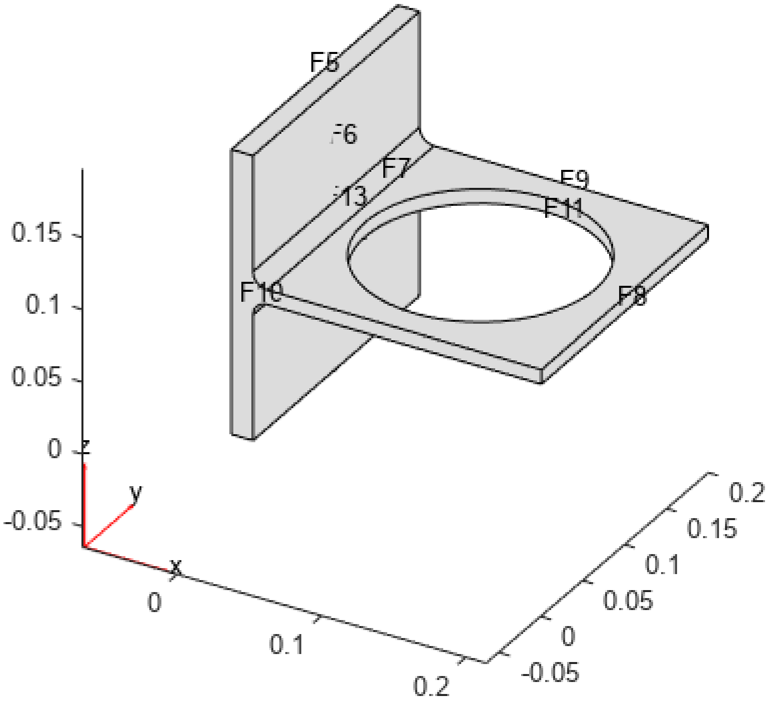
```
figure
gm = importGeometry("Plate10x10x1.stl");
pdegplot(gm,"FaceLabels","on")
```



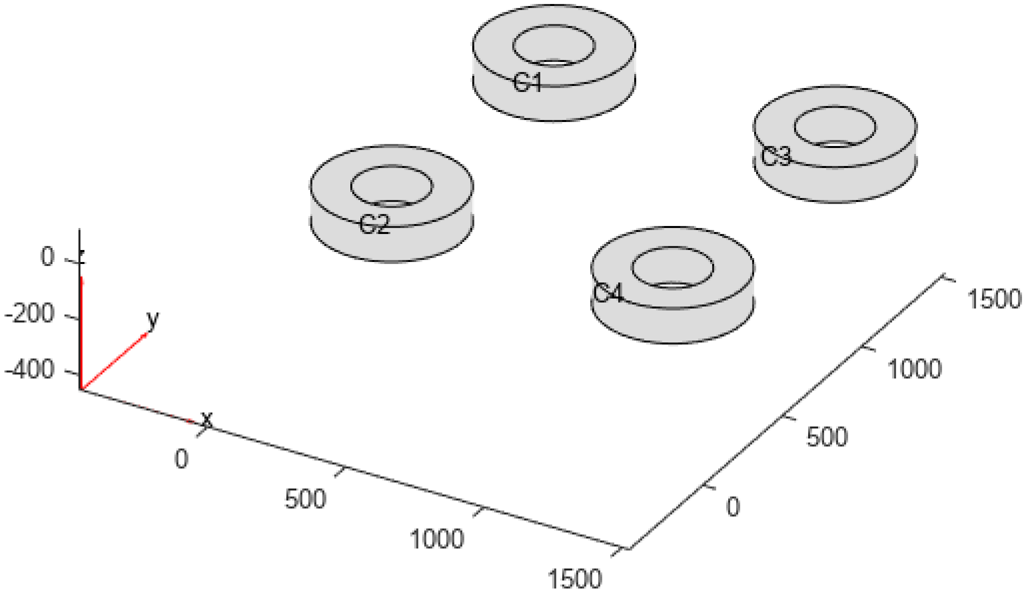
```
figure
gm = importGeometry("Tetrahedron.stl");
pdegplot(gm,"FaceLabels","on")
```



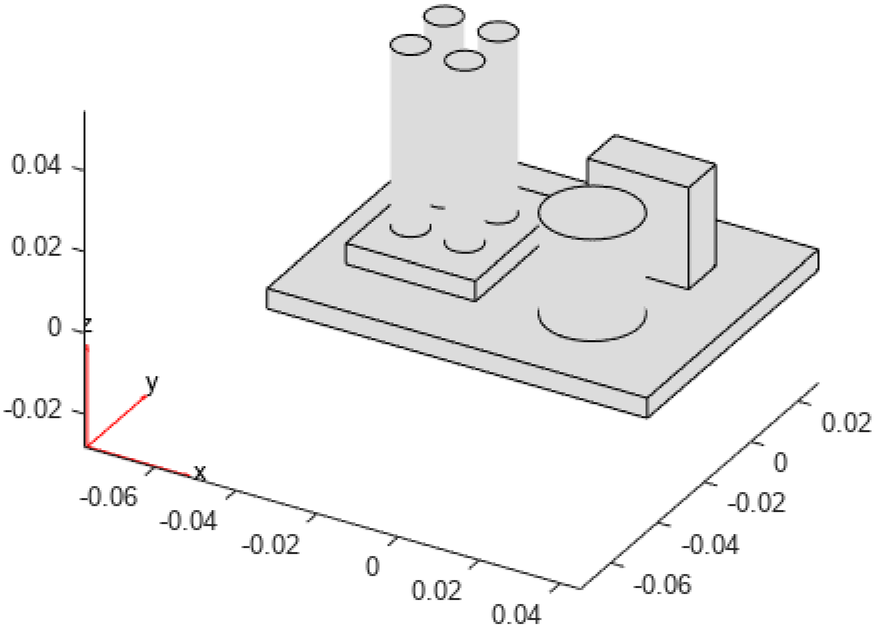
```
figure
gm = importGeometry("BracketWithHole.stl");
pdegplot(gm,"FaceLabels","on")
```



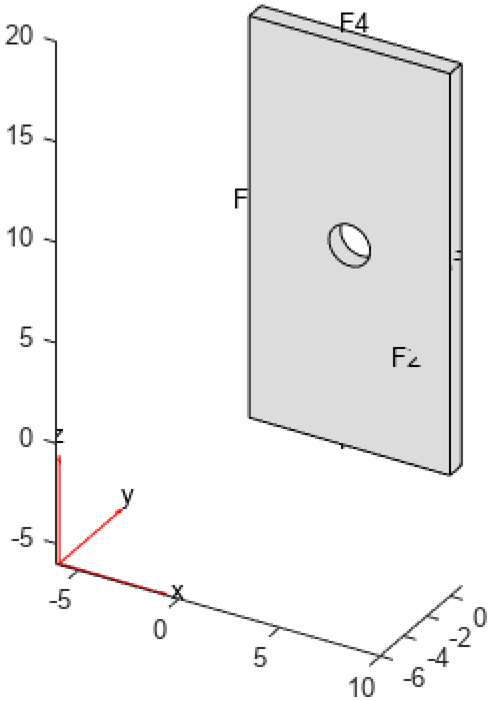
```
figure
gm = importGeometry("DampingMounts.stl");
pdegplot(gm,"CellLabels","on")
```



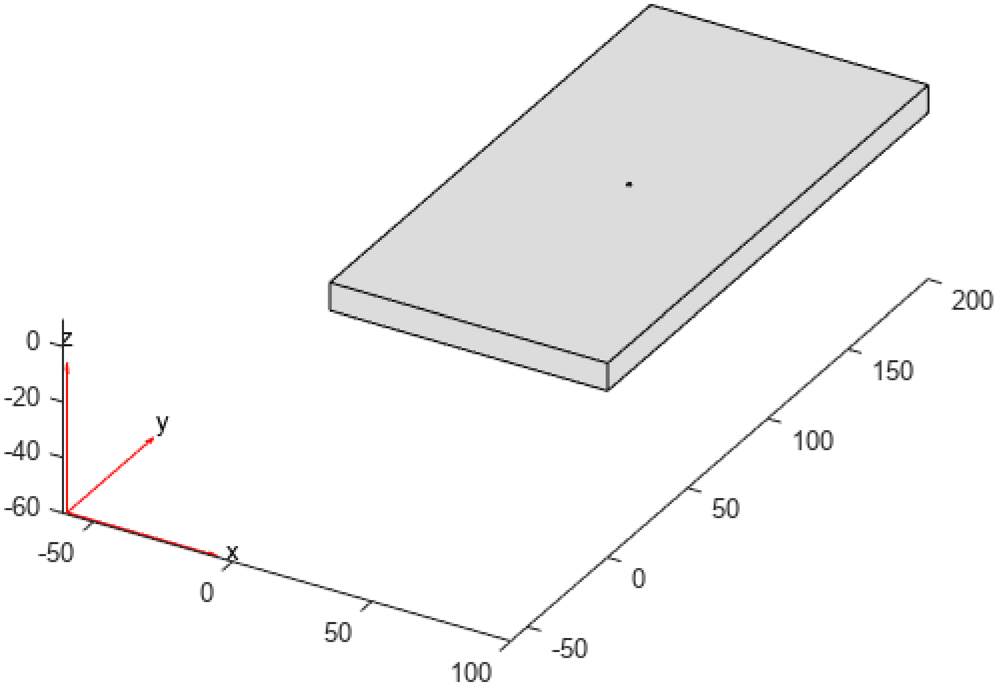
```
figure
gm = importGeometry("MotherboardFragment1.stl");
pdegplot(gm)
```



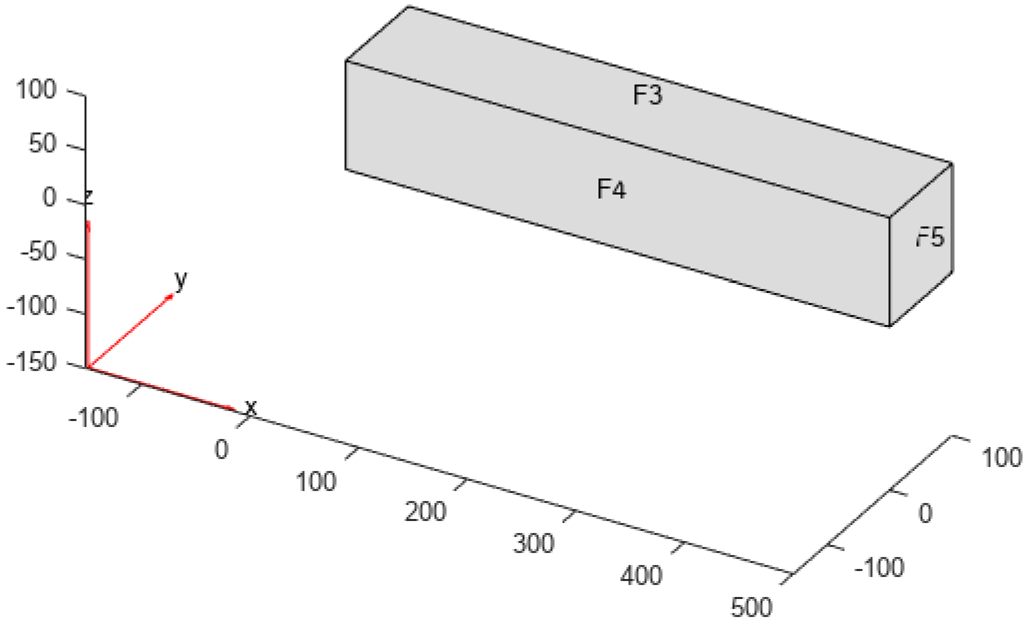
```
figure
gm = importGeometry("PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on")
```



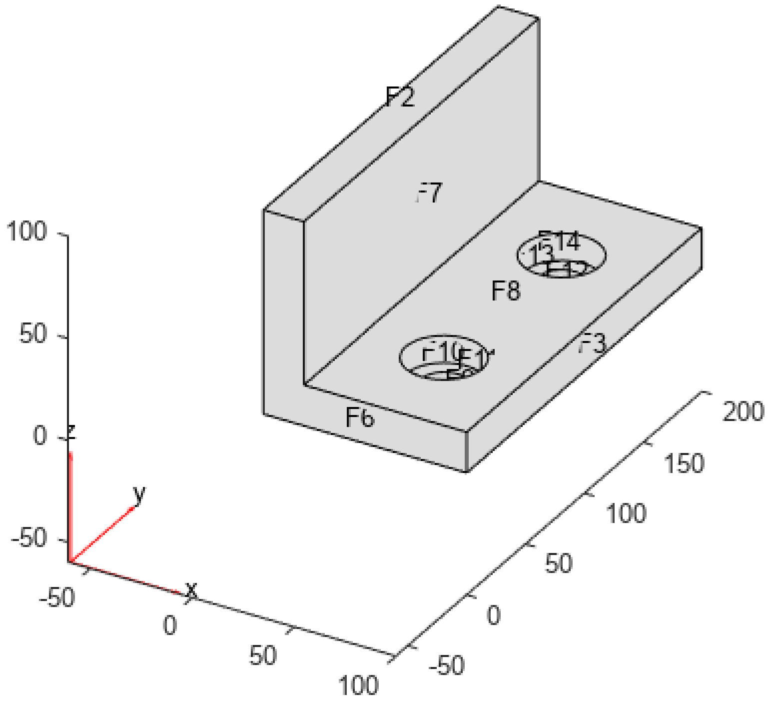
```
figure
gm = importGeometry("PlateSquareHoleSolid.stl");
pdegplot(gm)
```



```
figure
gm = importGeometry("SquareBeam.stl");
pdegplot(gm,"FaceLabels","on")
```

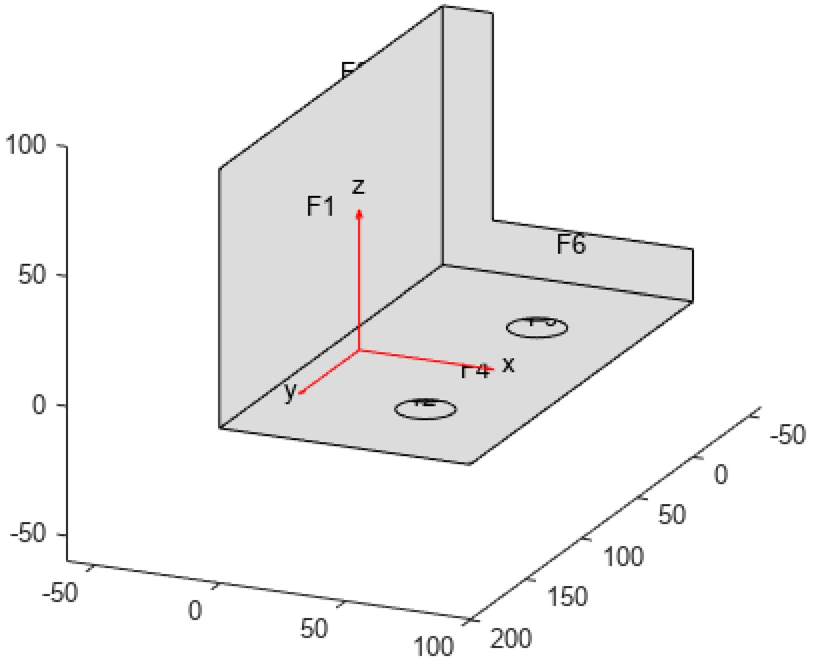


```
figure
gm = importGeometry("BracketTwoHoles.stl");
pdegplot(gm,"FaceLabels","on")
```

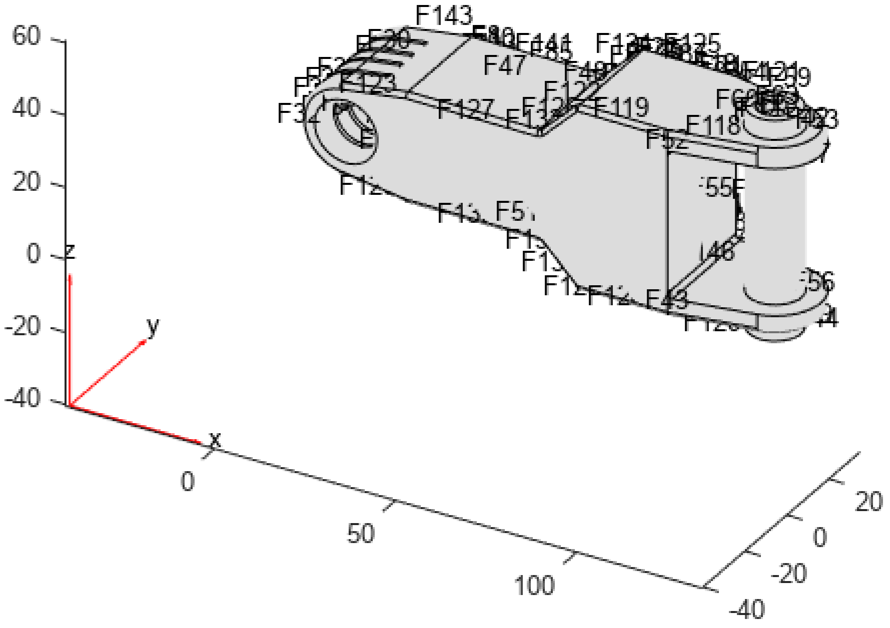


To see hidden portions of the geometry, rotate the figure using **Rotate 3D** button ⟲ or the `view` function. You can rotate the angle bracket to obtain the following view.

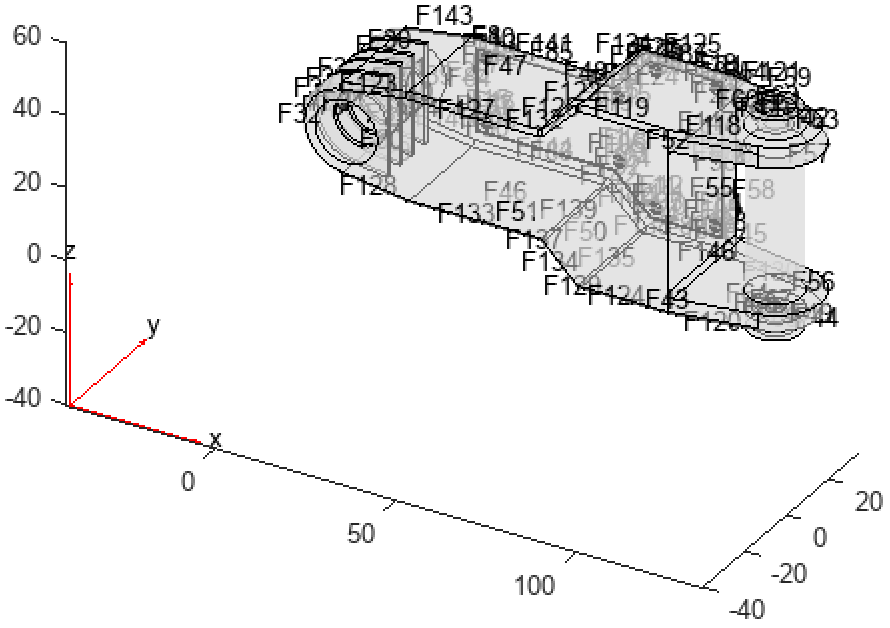
```
figure
pdegplot(gm,"FaceLabels","on")
view([-24 -19])
```



```
figure
gm = importGeometry("ForearmLink.stl");
pdegplot(gm,"FaceLabels","on");
```

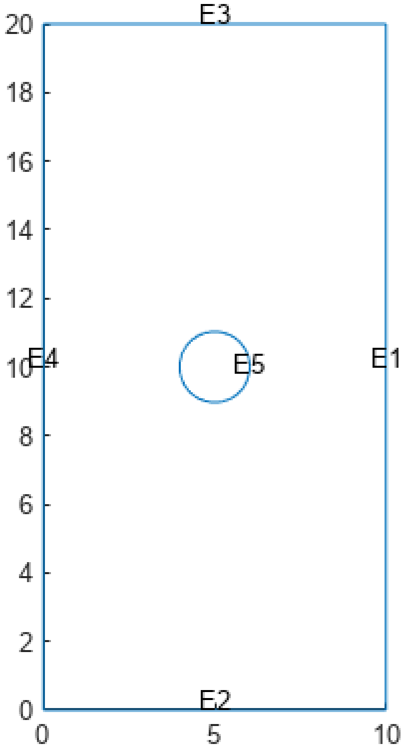


```
figure
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.5)
```

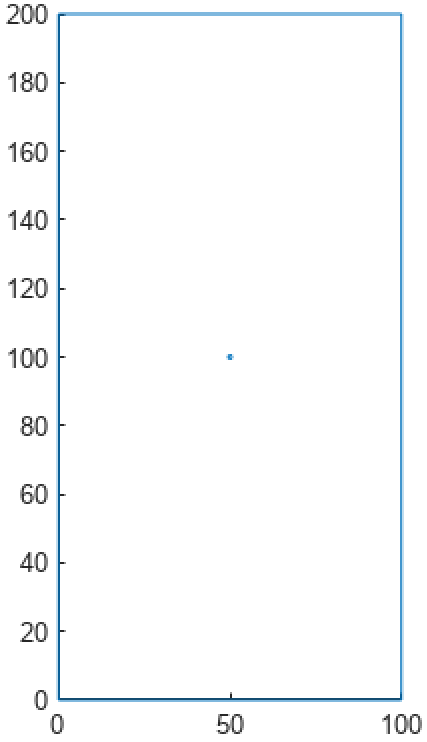


When you import a planar STL geometry, the toolbox converts it to a 2-D geometry by mapping it to the X-Y plane.

```
figure
gm = importGeometry("PlateHolePlanar.stl");
pdegplot(gm,"EdgeLabels","on")
```



```
figure
gm = importGeometry("PlateSquareHolePlanar.stl");
pdegplot(gm);
```



See Also

Related Examples

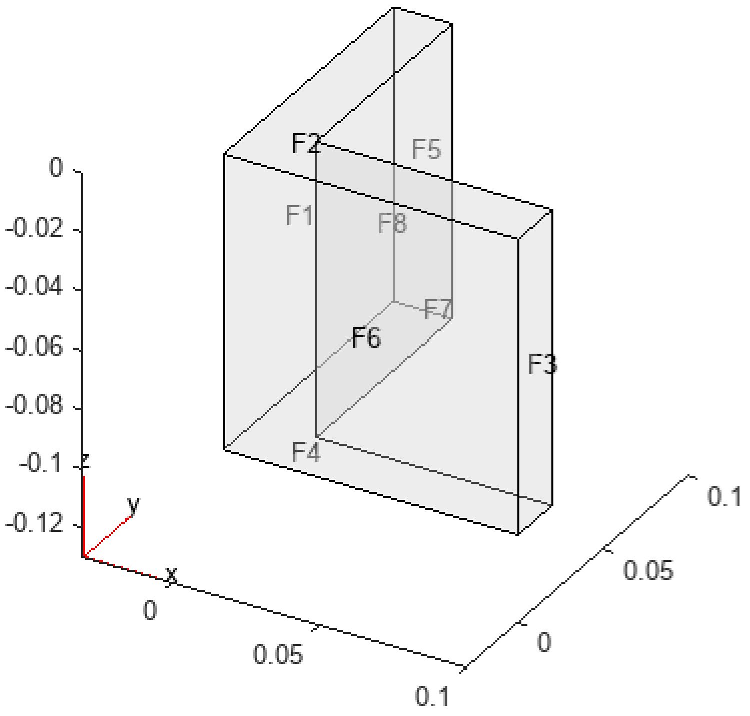
- "STEP File Import" on page 2-59

STEP File Import

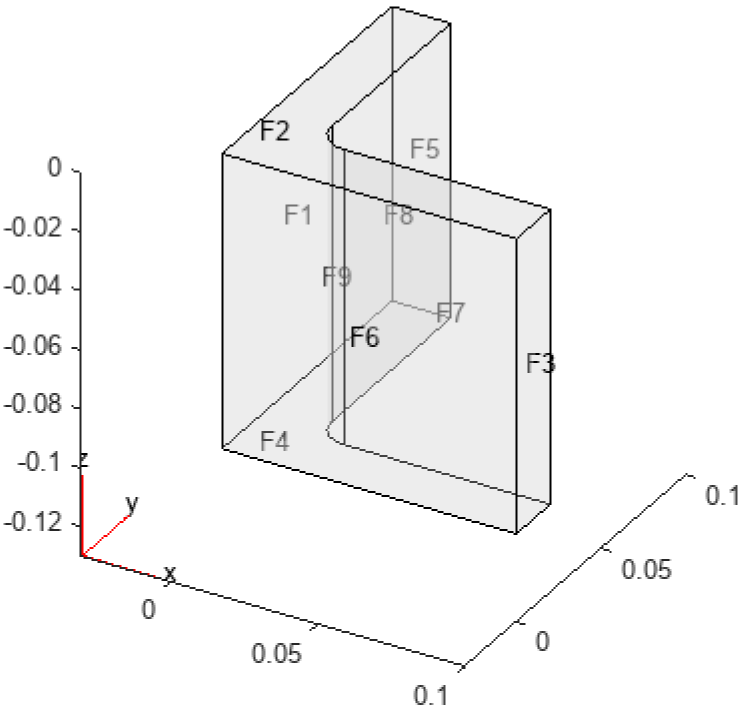
This example shows how to import a geometry from a STEP file and then plot the geometry. After importing, view the geometry using the `pdegplot` function.

Import and view the geometry examples from the STEP files included with Partial Differential Equation Toolbox™. To see the face IDs, set the `FaceLabels` name-value argument to `"on"`. To see the labels on all faces of the geometry, set the transparency to 0.3.

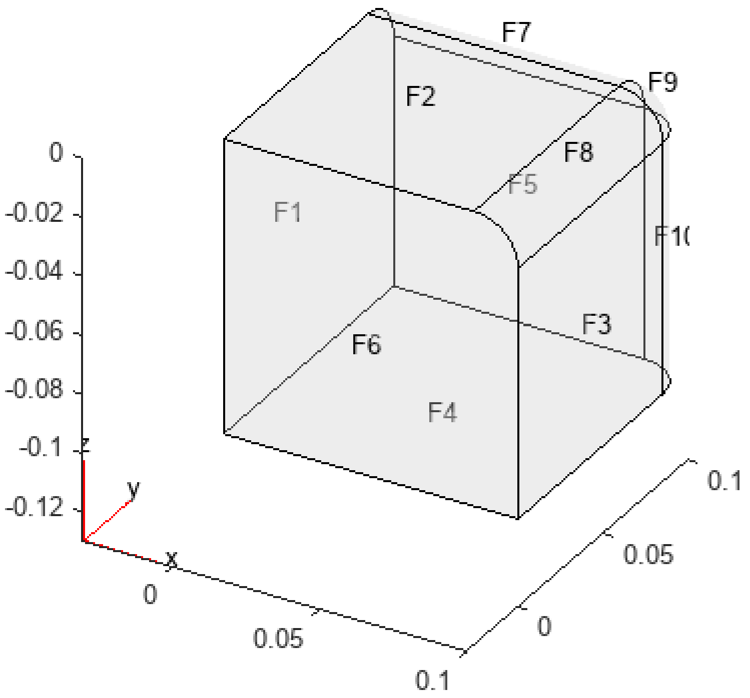
```
figure
gm = importGeometry("AngleBlock.step");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



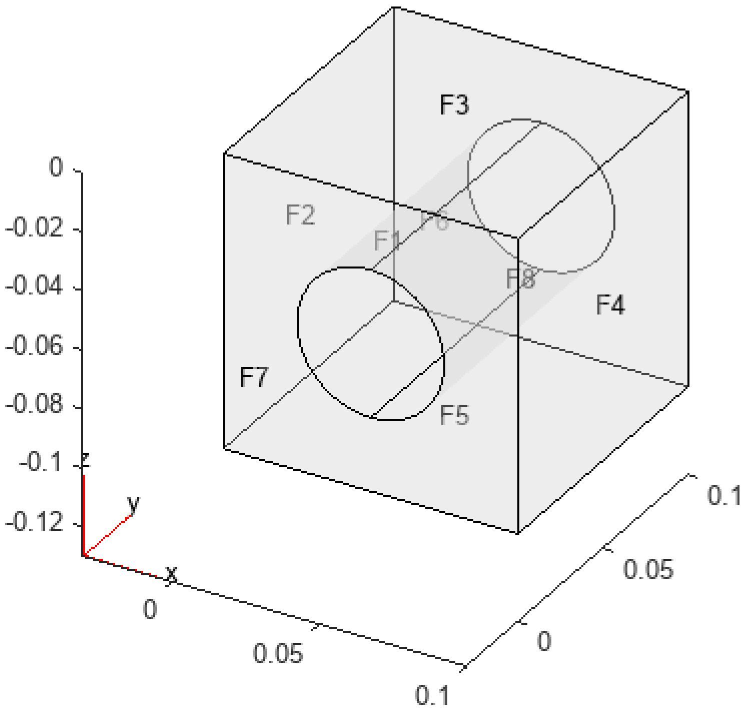
```
figure
gm = importGeometry("AngleBlockBlendR10.step");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



```
figure
gm = importGeometry("BlockBlendR15.step");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



```
figure
gm = importGeometry("BlockWithHole.step");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



See Also

Related Examples

- “STL File Import” on page 2-43

Geometry from Triangulated Mesh

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)
```

```
ans = 1×2

        4969           4
```

```
size(X)
```

```
ans = 1×2

        1456           3
```

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

Create data matrices of the appropriate sizes.

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

Create a PDE model and import the mesh.

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

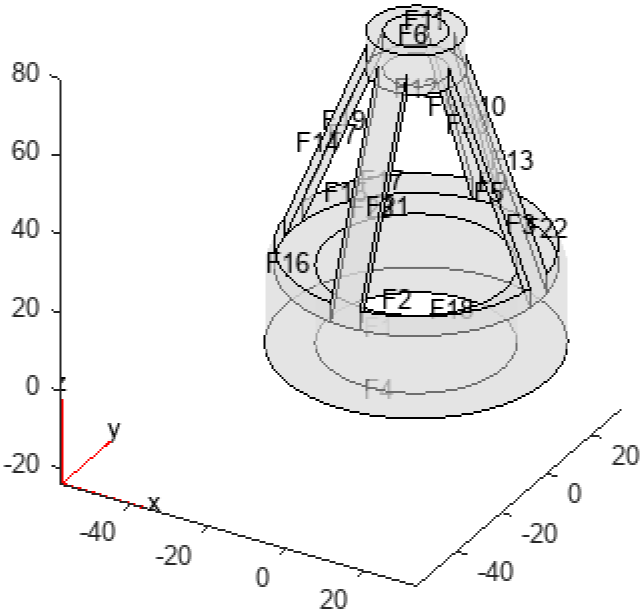
The model contains the imported mesh.

```
model.Mesh
```

```
ans = 
  FEMesh with properties:

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

View the geometry and face numbers.

```
pdegplot(model,"FaceLabels","on","FaceAlpha",0.5)
```



2-D Multidomain Geometry

Create a 2-D multidomain geometry from a mesh.

Load information about nodes, elements, and element-to-domain correspondence into your workspace. The file `MultidomainMesh2D` ships with your software.

```
load MultidomainMesh2D
```

Create a PDE model.

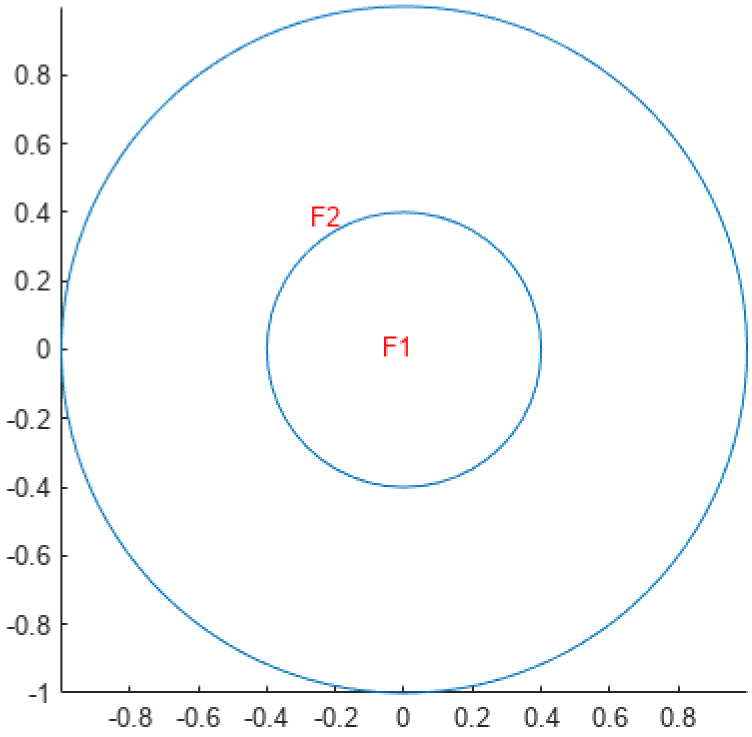
```
model = createpde;
```

Import the mesh into the model.

```
geometryFromMesh(model,nodes,elements,ElementIdToRegionId);
```

View the geometry and face numbers.

```
pdegplot(model,"FaceLabels","on")
```


F2
F1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.6 0.8

Geometry from alphaShape

Create a 3-D geometry using the MATLAB® `alphaShape` function. First, create an `alphaShape` object of a block with a cylindrical hole. Then 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`.

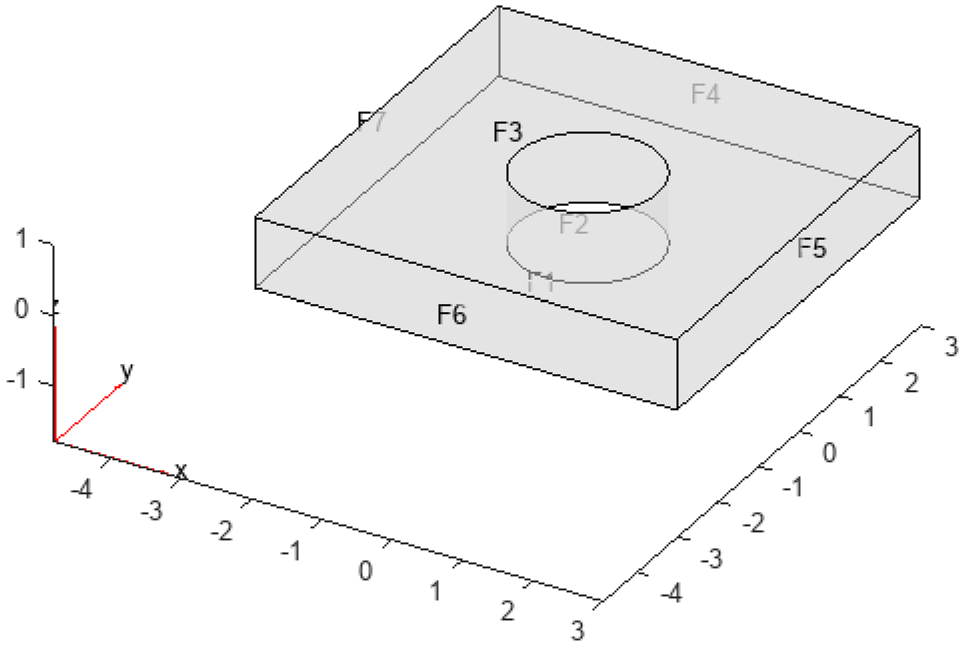
```
nodes = nodes';
elements = elements';
```

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);
```

Cuboids, Cylinders, and Spheres

This example shows how to create 3-D geometries formed by one or more cubic, cylindrical, and spherical cells by using the `multicuboid`, `multicylinder`, and `multisphere` functions, respectively. With these functions, you can create stacked or nested geometries. You also can create geometries where some cells are empty; for example, hollow cylinders, cubes, or spheres.

All cells in a geometry must be of the same type: either cuboids, or cylinders, or spheres. These functions do not combine cells of different types in one geometry.

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: 0
    NumVertices: 0
       Vertices: []
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

Include the geometry in the model.

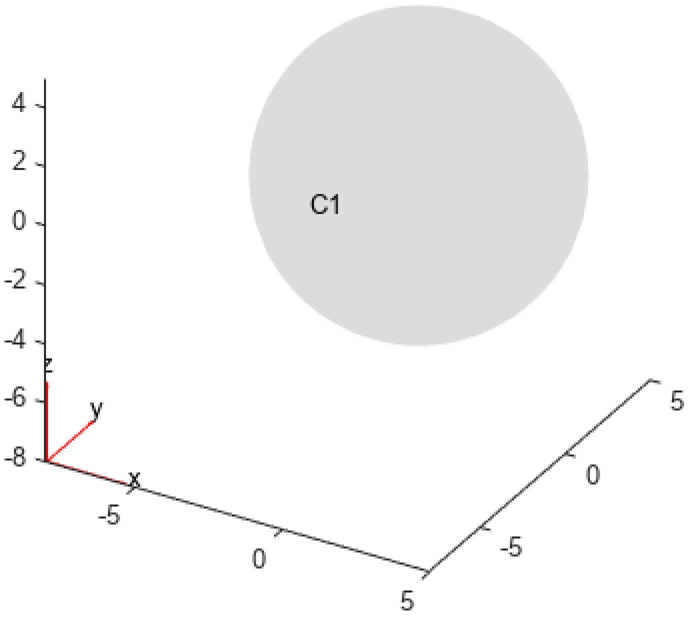
```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
```

```
        InitialConditions: []
                     Mesh: []
            SolverOptions: [1x1 pde.PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,"CellLabels","on")
```



Nested Cuboids of 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
       Vertices: [24x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

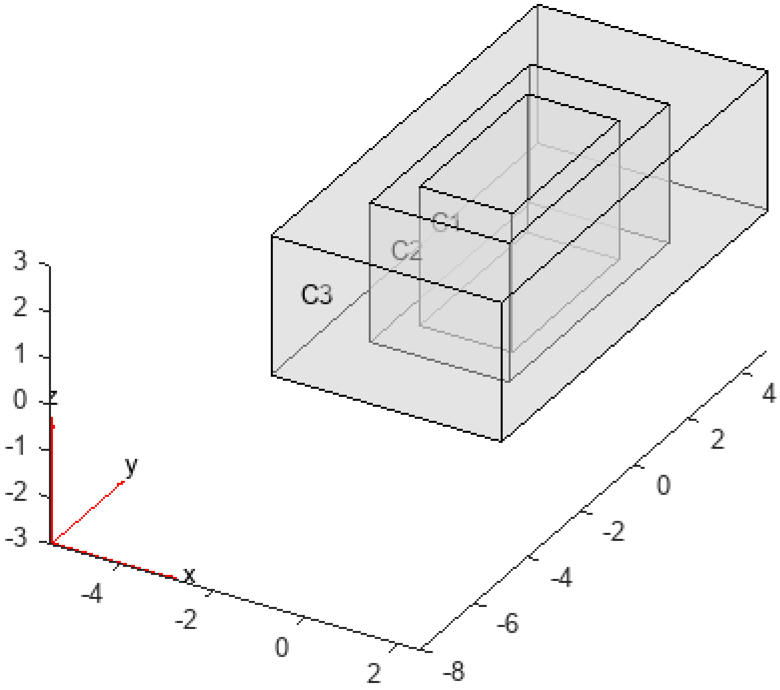
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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
       Vertices: [5x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
```

```
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

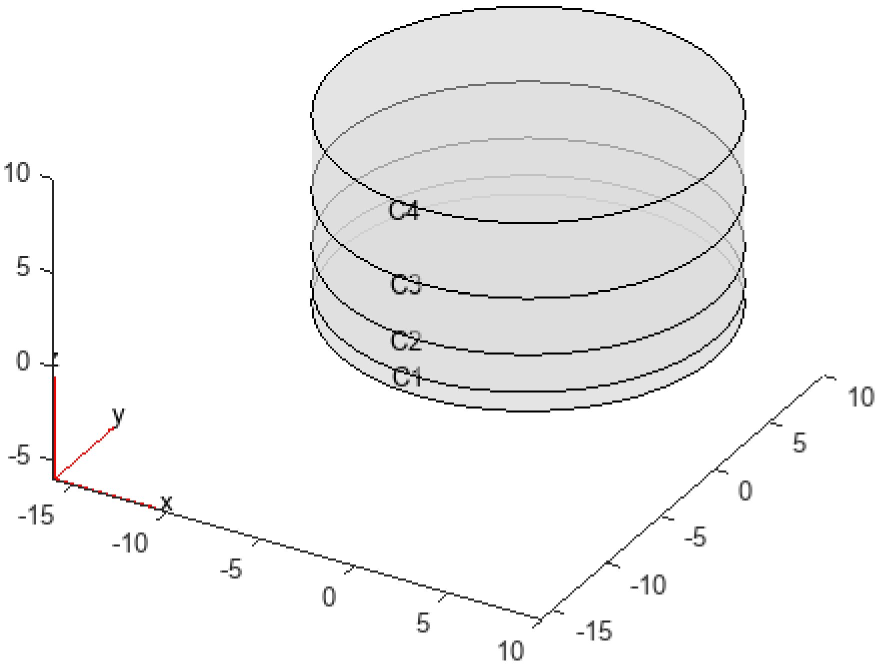
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,"CellLabels","on","FaceAlpha",0.5)
```



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
       Vertices: [4x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

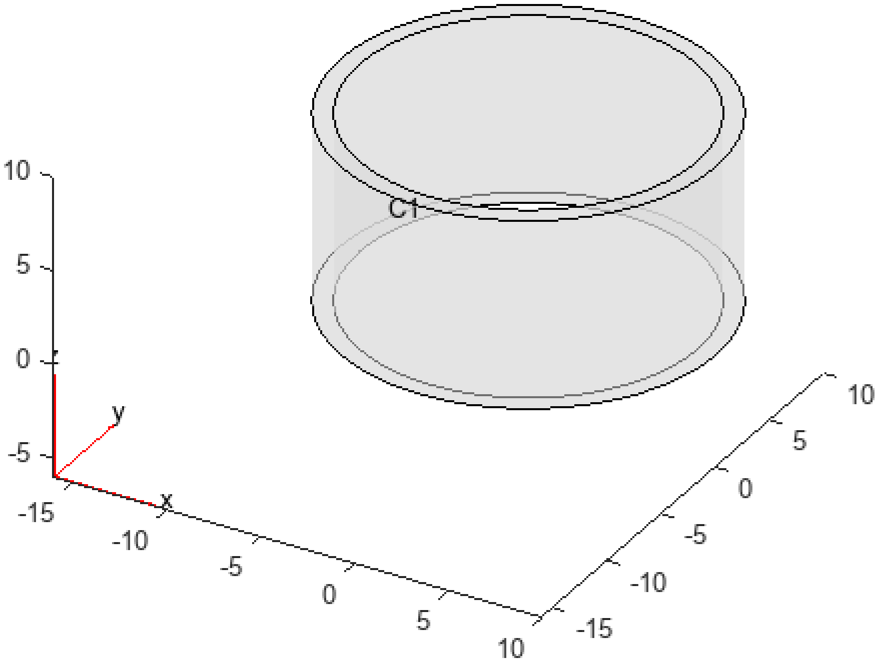
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

Plot the geometry.

```
pdegplot(model,"CellLabels","on","FaceAlpha",0.5)
```


C1
10
5
0
-5
z
y
x
-15
-10
-5
0
5
10
-15
-10
-5
0
5
10

Sphere in Cube

This example shows how to create a nested multidomain geometry consisting of a unit sphere and a cube. The first part of the example creates a cube with a spherical cavity by using `alphaShape`. The second part creates a solid sphere using tetrahedral elements, and then combines all tetrahedral elements to obtain a solid sphere embedded in a cube.

Cube with Spherical Cavity

First, create a geometry consisting of a cube with a spherical cavity. This geometry has one cell.

Create a 3-D rectangular mesh grid.

```
[xg, yg, zg] = meshgrid(-2:0.25:2);
Pcube = [xg(:) yg(:), zg(:)];
```

Extract the grid points located outside of the unit spherical region.

```
Pcavitycube = Pcube(vecnorm(Pcube') > 1,:);
```

Create points on the unit sphere.

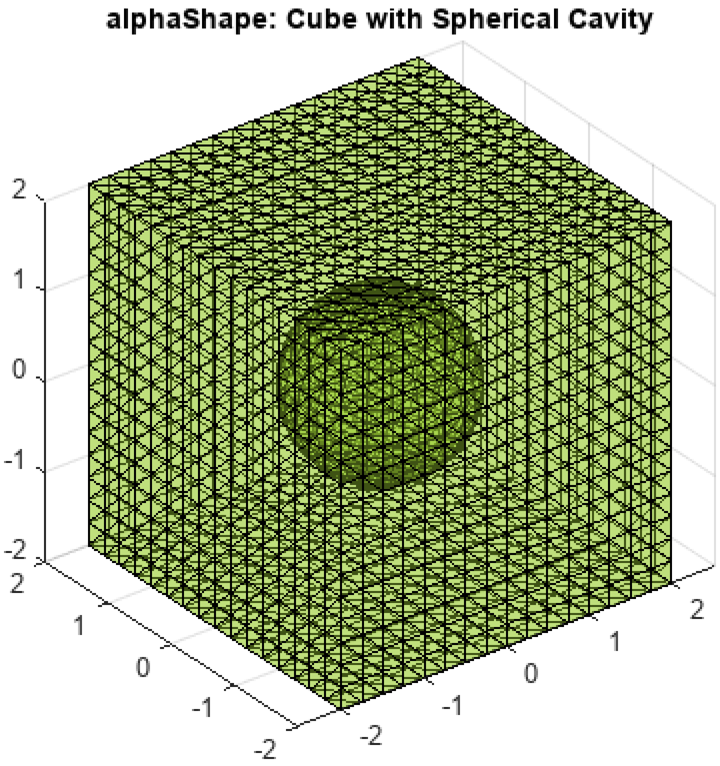
```
[x1,y1,z1] = sphere(24);
Psphere = [x1(:) y1(:) z1(:)];
Psphere = unique(Psphere,"rows");
```

Combine the coordinates of the rectangular grid (without the points inside the sphere) and the surface coordinates of the unit sphere.

```
Pcombined = [Pcavitycube;Psphere];
```

Create an `alphaShape` object representing the cube with the spherical cavity.

```
shpCubeWithSphericalCavity = alphaShape(Pcombined(:,1), ...
                                        Pcombined(:,2), ...
                                        Pcombined(:,3));

figure
plot(shpCubeWithSphericalCavity,"FaceAlpha",0.4)
title("alphaShape: Cube with Spherical Cavity")
```



Recover the triangulation that defines the domain of the `alphaShape` object.

```
[tri,loc] = alphaTriangulation(shpCubeWithSphericalCavity);
```

Create a PDE model.

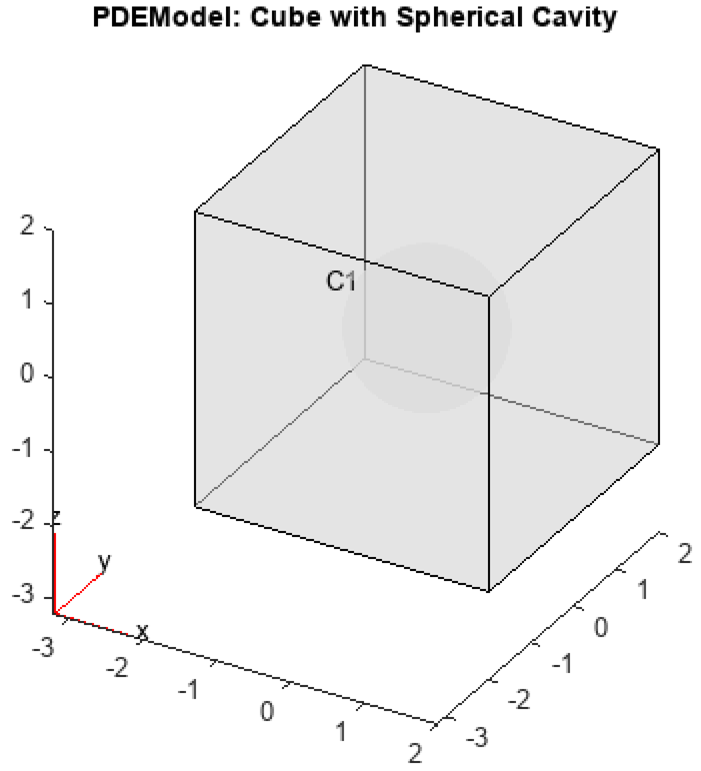
```
modelCube = createpde;
```

Create a geometry from the mesh and import the geometry and the mesh into the model.

```
[gCube,mshCube] = geometryFromMesh(modelCube,loc',tri');
```

Plot the resulting geometry.

```
figure
pdegplot(modelCube,"FaceAlpha",0.5,"CellLabels","on")
title("PDEModel: Cube with Spherical Cavity")
```



Solid Sphere Nested in Cube

Create tetrahedral elements to form a solid sphere by using the spherical shell and adding a new node at the center. First, obtain the spherical shell by extracting facets of the spherical boundary.

```
sphereFacets = boundaryFacets(mshCube,"Face",3);
sphereNodes = findNodes(mshCube,"region","Face",3);
```

Add a new node at the center.

```
newNodeID = size(mshCube.Nodes,2) + 1;
```

Construct the tetrahedral elements by using each of the three nodes on the spherical boundary facets and the new node at the origin.

```
sphereTets =  [sphereFacets; newNodeID*ones(1,size(sphereFacets,2))];
```

Create a model that combines the cube with the spherical cavity and a sphere.

```
model = createpde;
```

Create a vector that maps all `mshCube` elements to cell 1, and all elements of the solid sphere to cell 2.

```
e2c = [ones(1,size(mshCube.Elements,2)), 2*ones(1,size(sphereTets,2))];
```

Add a new node at the center `[0;0;0]` to the nodes of the cube with the cavity.

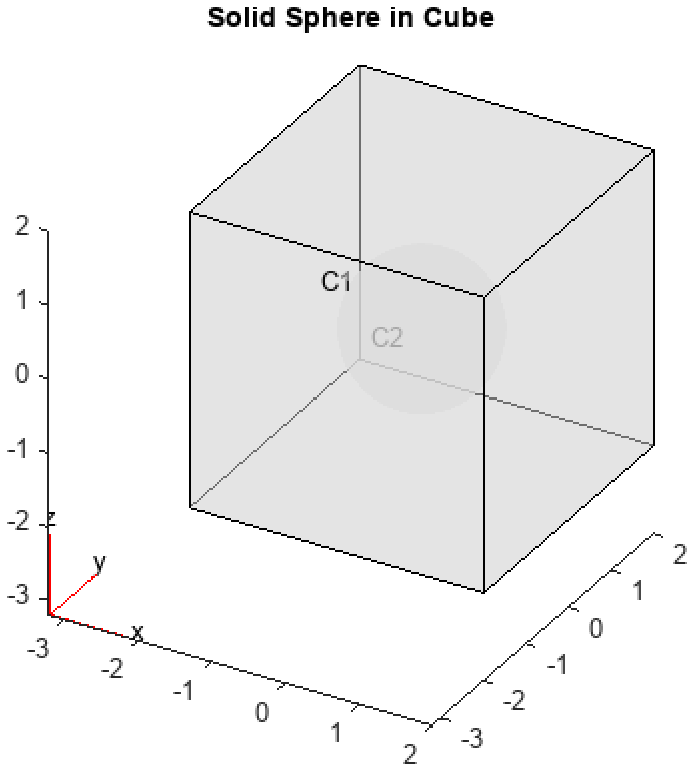
```
combinedNodes = [mshCube.Nodes,[0;0;0]];
```

Combine the element connectivity matrices.

```
combinedElements = [mshCube.Elements,sphereTets];
```

Create a two-cell geometry from the mesh.

```
[g,msh] = geometryFromMesh(model,combinedNodes,combinedElements,e2c);

figure
pdegplot(model,"FaceAlpha",0.5,"CellLabels","on")
title("Solid Sphere in Cube")
```



3-D Multidomain Geometry from 2-D Geometry

This example shows how to create a 3-D multidomain geometry by extruding a 2-D geometry imported from STL data. The original 2-D geometry represents a cooled turbine blade section defined by a 2-D profile.

Before extruding the geometry, this example modifies the original 2-D profile as follows:

- Translates the geometry to move the tip to the origin
- Aligns the chord with the *x*-axis
- Changes the dimensions from inches to millimeters

First, create a PDE model.

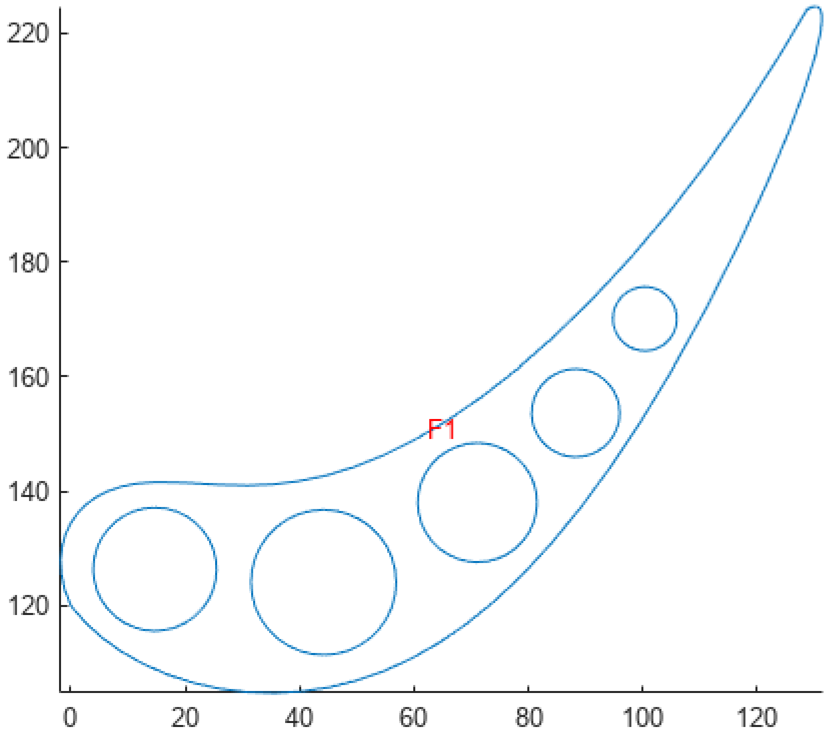
```
model = createpde;
```

Import the geometry into the model.

```
g = importGeometry(model,"CooledBlade2D.STL");
```

Plot the geometry with the face labels.

```
figure
pdegplot(model,"FaceLabels","on")
```



Translate the geometry to align the tip of the blade with the origin.

```
tip = [1.802091,-127.98192215];
translate(g,tip);
```

Rotate the geometry to align the chord with the *x*-axis.

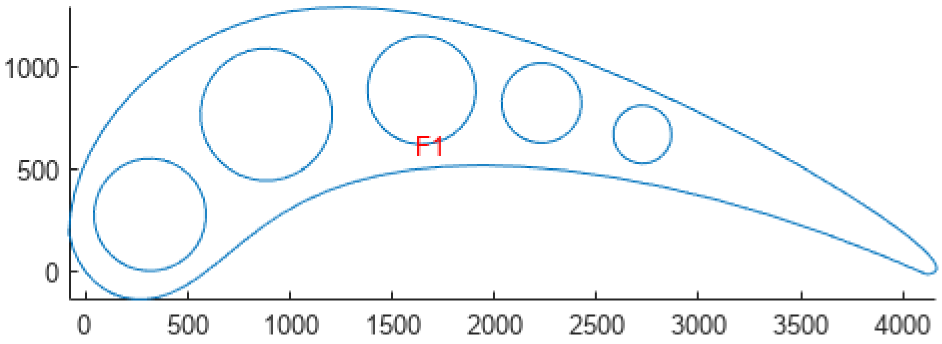
```
angle = -36.26005;
rotate(g,angle);
```

Scale the geometry to convert from inches to millimeters.

```
scale(g,[25.4 -25.4]);
```

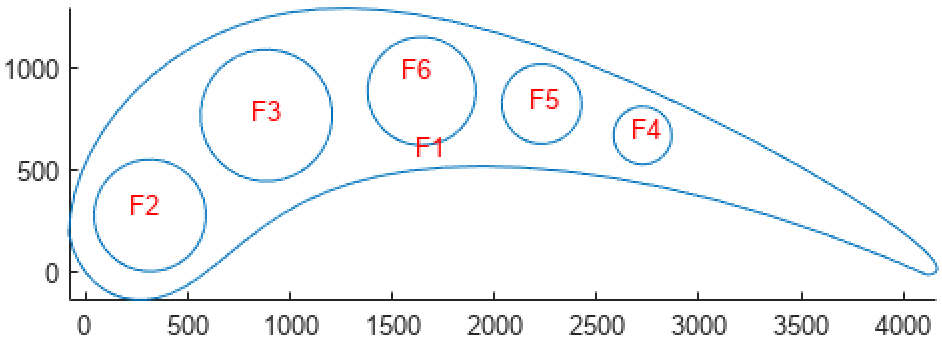
Plot the resulting geometry with the face labels.

```
figure
pdegplot(model,"FaceLabels","on")
```



Fill the void regions with faces and plot the resulting geometry.

```
g = addFace(g,{3, 4, 5, 6, 7});

figure
pdegplot(model,"FaceLabels","on")
```

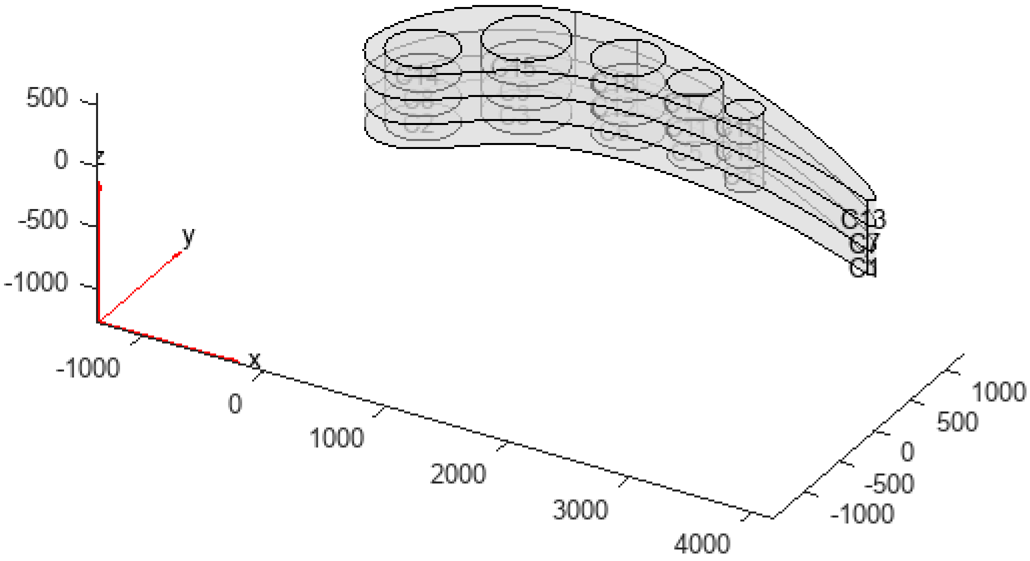


Extrude the geometry to create a stacked multilayer 3-D model of the blade. The thickness of each layer is 200 mm.

```
g = extrude(g,[200 200 200]);
```

Plot the geometry with the cell labels.

```
figure
pdegplot(model,"CellLabels","on","FaceAlpha",0.5)
```


500
0
-500
-1000
z
y
x
-1000
0
1000
2000
3000
4000
1000
500
0
-500
-1000
C14
C8
C2
C15
C9
C3
C13
C7
C1

Multidomain Geometry Reconstructed from Mesh

This example shows how to split a single-domain block geometry into two domains. The first part of the example generates a mesh and divides the mesh elements into two groups. The second part of the example creates a two-domain geometry based on this division.

Generate Mesh and Split Its Elements into Two Groups

Create a PDE model.

```
modelSingleDomain = createpde;
```

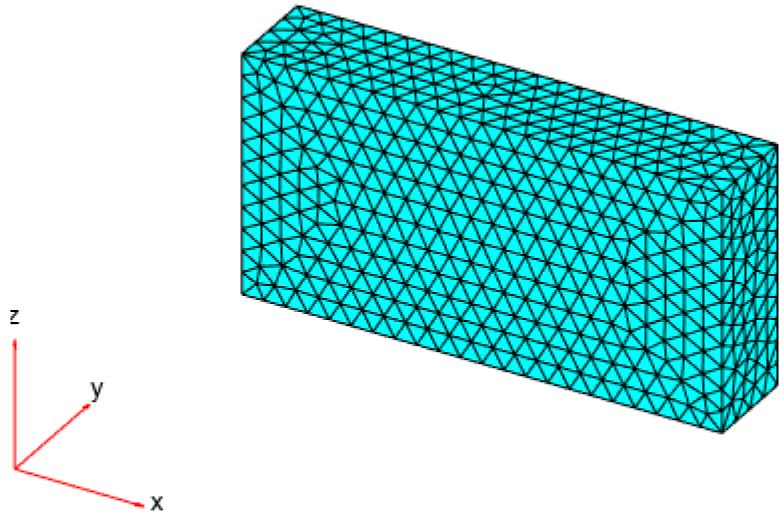
Import the geometry.

```
importGeometry(modelSingleDomain,"Block.stl");
```

Generate and plot a mesh.

```
msh = generateMesh(modelSingleDomain);
```

```
figure
pdemesh(modelSingleDomain)
```



Obtain the nodes and elements of the mesh.

```
nodes = msh.Nodes;
elements = msh.Elements;
```

Find the *x*-coordinates of the geometric centers of all elements of the mesh. First, create an array of the same size as `elements` that contains the *x*-coordinates of the nodes forming the mesh elements. Each column of this vector contains the *x*-coordinates of 10 nodes that form an element.

```
elemXCoords = reshape(nodes(1,elements),10,[]);
```

Compute the mean of each column of this array to get a vector of the *x*-coordinates of the element geometric centers.

```
elemXCoordsGeometricCenter = mean(elemXCoords);
```

Assume that all elements have the same region ID and create a matrix `ElementIdToRegionId`.

```
ElementIdToRegionId = ones(1,size(elements,2));
```

Find IDs of all elements for which the *x*-coordinate of the geometric center exceeds 60.

```
idx = elemXCoordsGeometricCenter > 60;
```

For the elements with centers located beyond $x = 60$, change the region IDs to 2.

```
ElementIdToRegionId(idx) = 2;
```

Create Geometry with Two Cells

Create a new PDE model.

```
modelTwoDomain = createpde;
```

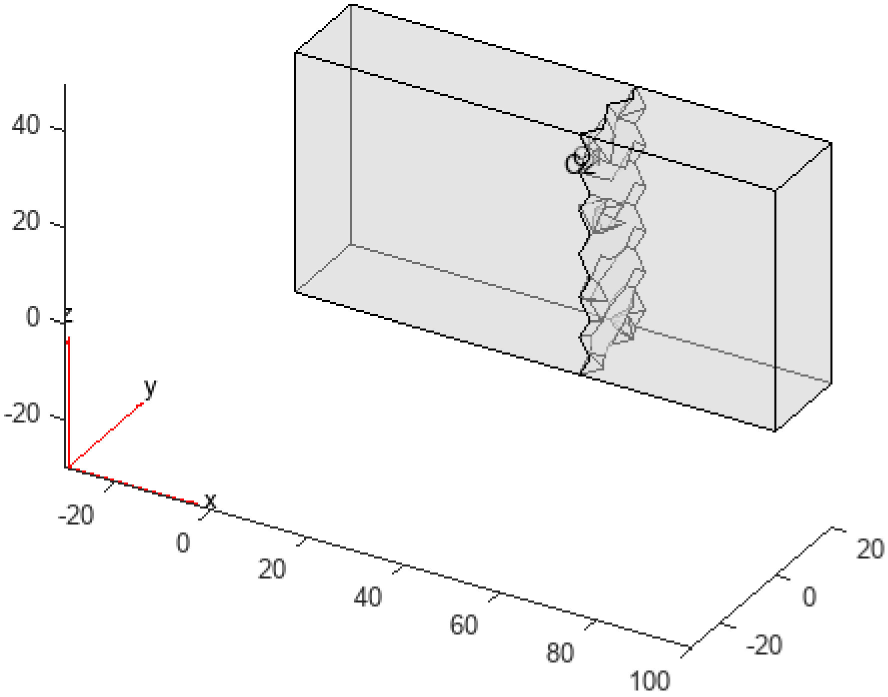
Using `geometryFromMesh`, import the mesh. Assign the elements to two cells based on their IDs.

```
geometryFromMesh(modelTwoDomain,nodes,elements,ElementIdToRegionId)
```

```
ans =
  DiscreteGeometry with properties:

       NumCells: 2
       NumFaces: 56
       NumEdges: 121
    NumVertices: 68
       Vertices: [68x3 double]
```

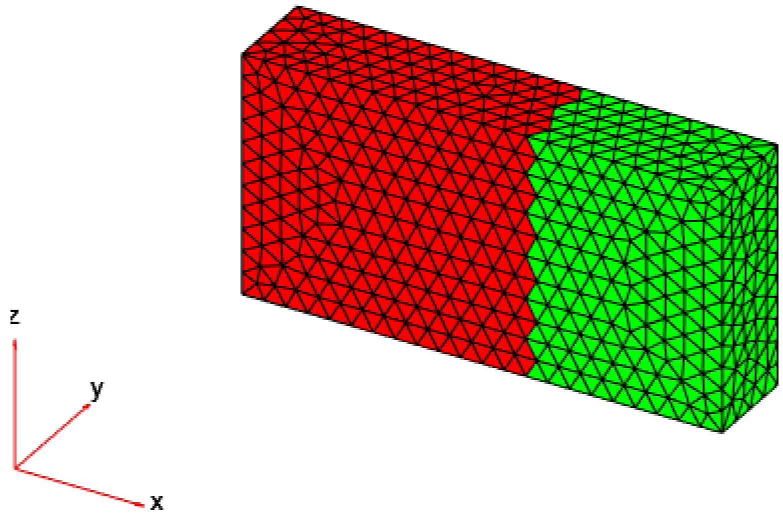
Plot the geometry, displaying the cell labels.

```
pdegplot(modelTwoDomain,"CellLabels","on","FaceAlpha",0.5)
```



Highlight the elements from cell 1 in red and the elements from cell 2 in green.

```
elementIDsCell1 = findElements(modelTwoDomain.Mesh,"region","Cell",1);
elementIDsCell2 = findElements(modelTwoDomain.Mesh,"region","Cell",2);

figure
pdemesh(modelTwoDomain.Mesh.Nodes, ...
        modelTwoDomain.Mesh.Elements(:,elementIDsCell1), ...
        "FaceColor","red")
hold on
pdemesh(modelTwoDomain.Mesh.Nodes, ...
        modelTwoDomain.Mesh.Elements(:,elementIDsCell2), ...
        "FaceColor","green")
```



When you divide mesh elements into groups and then create a multidomain geometry based on this division, the mesh might be invalid for the multidomain geometry. For example, elements in a cell might be touching by only a node or an edge instead of sharing a face. In this case, `geometryFromMesh` throws an error saying that neighboring elements in the mesh are not properly connected.

Put Equations in Divergence Form

In this section...

Coefficient Matching for Divergence Form

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

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

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

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

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

In order to transform this expanded equation into the required form, 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

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

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

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

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

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

so

$$c_2 = 2xy.$$

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

so

$$c_3 = xy$$

This completes the conversion of the equation to the divergence form

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

Boundary Conditions Can Affect the c Coefficient

The `c` coefficient appears in the generalized Neumann condition

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

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

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

$$\begin{aligned} \nabla \cdot (c\nabla u) &= \nabla \cdot \left(\begin{pmatrix} c_1 & c_3 \\ c_2 & c_4 \end{pmatrix}\begin{pmatrix} u_x \\ u_y \end{pmatrix}\right) \\ &= \frac{\partial}{\partial x}\left(c_1u_x + c_3u_y\right) + \frac{\partial}{\partial y}\left(c_2u_x + c_4u_y\right) \\ &= c_1u_{xx} + c_4u_{yy} + (c_2 + c_3)u_{xy} \end{aligned}$$

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.

Coefficient Conversion with Symbolic Math Toolbox

You can transform a partial differential equation into the required form by using Symbolic Math Toolbox™. The toolbox offers these two functions to help with the conversion:

- `pdeCoefficients` converts a PDE into the required form and extracts the coefficients into a structure of double-precision numbers and function handles, which can be used by `specifyCoefficients`. The `pdeCoefficients` function also can return a structure of symbolic expressions, in which case you need to convert these expressions to double format before passing them to `specifyCoefficients`.
- `pdeCoefficientsToDouble` converts symbolic PDE coefficients to double format.

“Solve Partial Differential Equation of Nonlinear Heat Transfer” (Symbolic Math Toolbox) shows how the Symbolic Math Toolbox functions can help you convert a PDE to the required form. “Nonlinear Heat Transfer in Thin Plate” on page 3-210 shows the same example without the use of Symbolic Math Toolbox.

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,

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

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

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

See Also

Related Examples

- “Equations You Can Solve Using PDE Toolbox” on page 1-3
- “Solve Problems Using PDEModel Objects” on page 2-2
- “Solve Partial Differential Equation of Nonlinear Heat Transfer” (Symbolic Math Toolbox)

f Coefficient for specifyCoefficients

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 PDE Toolbox" on page 1-3. 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

  ```
  fcoeff = fcoeffunction(location,state)
  ```

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

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

  `solvepde` or `solvepdeeig` compute and populate the data in the `location` and `state` structure arrays and pass this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments. To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

  ```
  fcoeff = ...
  @(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
  specifyCoefficients(model,"f",fcoeff,...
  ```

  - `location` is a structure with these fields:
    - `location.x`
    - `location.y`
    - `location.z`
    - `location.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 location 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 location structure. The solution and gradient estimates are N-by-Nr matrices. The `state.time` field is a scalar representing time for time-dependent models.

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

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

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

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

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

This represents the coefficient function

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

See Also

Related Examples

- "Put Equations in Divergence Form" on page 2-87
- "Solve Problems Using PDEModel Objects" on page 2-2
- "m, d, or a Coefficient for specifyCoefficients" on page 2-107
- "c Coefficient for specifyCoefficients" on page 2-92

c Coefficient for specifyCoefficients

In this section...

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 2-92. N is the number of equations, see "Equations You Can Solve Using PDE Toolbox" on page 1-3.

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(location,state)
  ```

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

  - $N1$ is the length of the vector representing the `c` coefficient. There are several possible values of $N1$, detailed in "Some c Vectors Can Be Short" on page 2-94. For 2-D geometry, $1 \le N1 \le 4N^2$, and for 3-D geometry, $1 \le N1 \le 9N^2$.
  - Nr is the number of points in the location that the solver passes. Nr is equal to the length of the `location.x` or any other `location` 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

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

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 3*N*-by-3*N* matrix, where the matrix is logically an *N*-by-*N* matrix of 3-by-3 blocks.

$$
\left(\begin{matrix}
c(1,1,1,1) & c(1,1,1,2) & c(1,1,1,3) & c(1,2,1,1) & c(1,2,1,2) & c(1,2,1,3) & \cdots & c(1,N,1,1) & c(1,N,1,2) & c(1,N,1,3)\\
c(1,1,2,1) & c(1,1,2,2) & c(1,1,2,3) & c(1,2,2,1) & c(1,2,2,2) & c(1,2,2,3) & \cdots & c(1,N,2,1) & c(1,N,2,2) & c(1,N,2,3)\\
c(1,1,3,1) & c(1,1,3,2) & c(1,1,3,3) & c(1,2,3,1) & c(1,2,3,2) & c(1,2,3,3) & \cdots & c(1,N,3,1) & c(1,N,3,2) & c(1,N,3,3)\\
c(2,1,1,1) & c(2,1,1,2) & c(2,1,1,3) & c(2,2,1,1) & c(2,2,1,2) & c(2,2,1,3) & \cdots & c(2,N,1,1) & c(2,N,1,2) & c(2,N,1,3)\\
c(2,1,2,1) & c(2,1,2,2) & c(2,1,2,3) & c(2,2,2,1) & c(2,2,2,2) & c(2,2,2,3) & \cdots & c(2,N,2,1) & c(2,N,2,2) & c(2,N,2,3)\\
c(2,1,3,1) & c(2,1,3,2) & c(2,1,3,3) & c(2,2,3,1) & c(2,2,3,2) & c(2,2,3,3) & \cdots & c(2,N,3,1) & c(2,N,3,2) & c(2,N,3,3)\\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots\\
c(N,1,1,1) & c(N,1,1,2) & c(N,1,1,3) & c(N,2,1,1) & c(N,2,1,2) & c(N,2,1,3) & \cdots & c(N,N,1,1) & c(N,N,1,2) & c(N,N,1,3)\\
c(N,1,2,1) & c(N,1,2,2) & c(N,1,2,3) & c(N,2,2,1) & c(N,2,2,2) & c(N,2,2,3) & \cdots & c(N,N,2,1) & c(N,N,2,2) & c(N,N,2,3)\\
c(N,1,3,1) & c(N,1,3,2) & c(N,1,3,3) & c(N,2,3,1) & c(N,2,3,2) & c(N,2,3,3) & \cdots & c(N,N,3,1) & c(N,N,3,2) & c(N,N,3,3)
\end{matrix}\right.
$$

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,

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

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

Some c Vectors Can Be Short

Often, your tensor **c** has structure, such as symmetric or block diagonal. In many cases, you can represent **c** using a smaller vector than one with $4N^2$ components for 2-D or $9N^2$ components for 3-D. The following sections give the possibilities.

- “2-D Systems” on page 2-94
- “3-D Systems” on page 2-98

2-D Systems

- “Scalar c, 2-D Systems” on page 2-95
- “Two-Element Column Vector c, 2-D Systems” on page 2-95
- “Three-Element Column Vector c, 2-D Systems” on page 2-95
- “Four-Element Column Vector c, 2-D Systems” on page 2-95
- “N-Element Column Vector c, 2-D Systems” on page 2-96
- “2N-Element Column Vector c, 2-D Systems” on page 2-96
- “3N-Element Column Vector c, 2-D Systems” on page 2-97

- “4N-Element Column Vector c, 2-D Systems” on page 2-97
- “2N(2N+1)/2-Element Column Vector c, 2-D Systems” on page 2-98
- “4N2-Element Column Vector c, 2-D Systems” on page 2-98

Scalar c, 2-D Systems

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

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

Two-Element Column Vector c, 2-D Systems

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

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

Three-Element Column Vector c, 2-D Systems

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

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

Four-Element Column Vector c, 2-D Systems

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

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

N-Element Column Vector c, 2-D Systems

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

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

Caution If N = 2, 3, or 4, the 2-, 3-, or 4-element column vector form takes precedence over the N-element form. For example, if N = 3, and you have a c matrix of the form

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

you cannot use the N-element form of c. Instead, you must use the 2N-element form. If you give c as the vector `[c1;c2;c3]`, the software interprets `c` as a 3-element form:

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

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

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

The software interprets a 2N-element column vector c as a diagonal matrix.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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" on page 2-99
- "Three-Element Column Vector c, 3-D Systems" on page 2-99
- "Six-Element Column Vector c, 3-D Systems" on page 2-99
- "Nine-Element Column Vector c, 3-D Systems" on page 2-100

- “N-Element Column Vector c, 3-D Systems” on page 2-100
- “3N-Element Column Vector c, 3-D Systems” on page 2-101
- “6N-Element Column Vector c, 3-D Systems” on page 2-103
- “9N-Element Column Vector c, 3-D Systems” on page 2-103
- “3N(3N+1)/2-Element Column Vector c, 3-D Systems” on page 2-103
- “9N2-Element Column Vector c, 3-D Systems” on page 2-104

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 6*N*-element column vector *c* as a symmetric block diagonal matrix. In the following diagram, • means the entry is symmetric.

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

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

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

The software interprets a 9*N*-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 (9*i* + 3*l* + *k* – 12) of the vector `c`.

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

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

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

Coefficient *c*(*i*,*j*,*k*,*l*), for *i* < *j*, is in row (9(*j*–1)(*j*–2)/2 + 6(*j*–1) + 9*i* + 3*l* + *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.

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

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

Functional Form

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

```
ccoeff = ccoeffunction(location,state)
```

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

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

`solvepde` or `solvepdeeig` compute and populate the data in the `location` and `state` structure arrays and pass this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments. To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
ccoeff = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
specifyCoefficients(model,"c",ccoeff,...
```

- `location` is a structure with these fields:
  - `location.x`
  - `location.y`
  - `location.z`
  - `location.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 location 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 location structure. The solution and gradient estimates are N-by-Nr matrices. The `state.time` field is a scalar representing time for time-dependent models.

Your function must return a matrix of size $N1$-by-Nr, where:

- $N1$ is the number of coefficients you pass to the solver. There are several possible values of $N1$, detailed in "Some c Vectors Can Be Short" on page 2-94. For 2-D geometry, $1 \le N1 \le 4N^2$, and for 3-D geometry, $1 \le N1 \le 9N^2$.
- Nr is the number of points in the location that the solver passes. Nr is equal to the length of the `location.x` or any other `location` field. The function should evaluate `c` at these points.

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(location,state)

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

See Also

Related Examples

- “Put Equations in Divergence Form” on page 2-87
- “Solve Problems Using PDEModel Objects” on page 2-2
- “f Coefficient for specifyCoefficients” on page 2-90
- “m, d, or a Coefficient for specifyCoefficients” on page 2-107

m, d, or a Coefficient for specifyCoefficients

In this section...

Coefficients m, d, or a

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

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

or in the eigenvalue system

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

or

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

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

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

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

For nonconstant coefficient matrices, see "Nonconstant m, d, or a" on page 2-108.

Note The `d` coefficient takes a special matrix form when `m` is nonzero. See "d Coefficient When m is Nonzero" on page 5-1205.

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 2-107
- "N-Element Column Vector m, d, or a" on page 2-108
- "N(N+1)/2-Element Column Vector m, d, or a" on page 2-108
- "N2-Element Column Vector m, d, or a" on page 2-108

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`.

N2-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

```
dcoeff = dcoeffunction(location,state)
```

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

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

`solvepde` or `solvepdeeig` compute and populate the data in the `location` and `state` structure arrays and pass this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments. To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
dcoeff = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
specifyCoefficients(model,"d",dcoeff,...
```

- `location` is a structure with these fields:
  - `location.x`
  - `location.y`
  - `location.z`
  - `location.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 location 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 location structure. The solution and gradient estimates are *N*-by-*Nr* matrices. The `state.time` field is a scalar representing time for time-dependent models.

Your function must return a matrix of size *N*1-by-*Nr*, where:

- *N*1 is the length of the vector representing the coefficient. There are several possible values of *N*1, detailed in "Short m, d, or a vectors" on page 2-107. $1 \le N1 \le N^2$.
- *Nr* is the number of points in the location that the solver passes. *Nr* is equal to the length of the `location.x` or any other `location` field. The function should evaluate **m**, **d**, or **a** at these points.

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

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

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

This `d` is a symmetric matrix. So it is natural to represent `d` as a "N(N+1)/2-Element Column Vector m, d, or a" on page 2-108:

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

For that form, the following function is appropriate.

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

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

See Also

Related Examples

- “Put Equations in Divergence Form” on page 2-87
- “Solve Problems Using PDEModel Objects” on page 2-2
- “f Coefficient for specifyCoefficients” on page 2-90
- “c Coefficient for specifyCoefficients” on page 2-92

View, Edit, and Delete PDE Coefficients

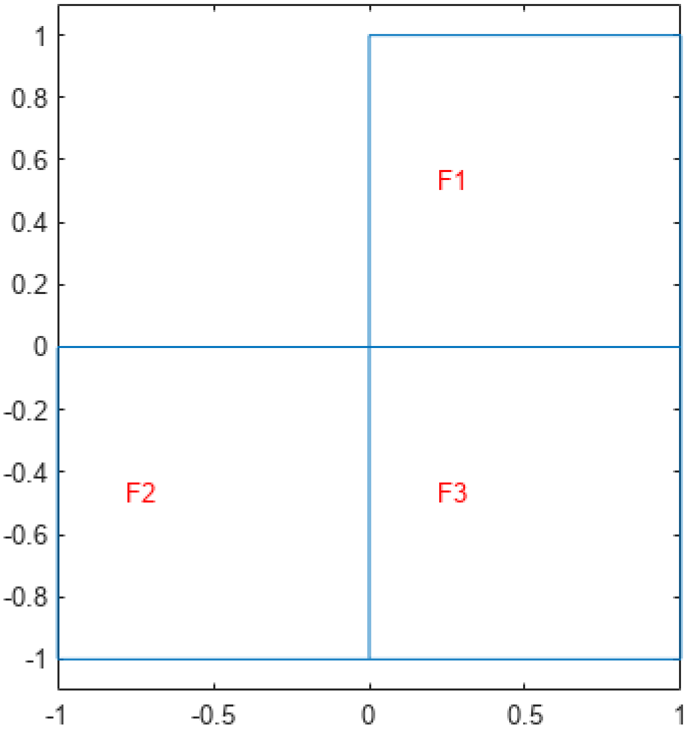
View Coefficients

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

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(location)
```

`solvepde` computes and populates the data in the `location` structure array and passes this data to your function. You can define your function so that its output depends on this data. You can use any name instead of `location`. To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` argument. For example:

```
u0 = @(location) initfunWithAdditionalArgs(location,arg1,arg2...)
setInitialConditions(model,u0)
```

`location` is a structure array with fields `location.x`, `location.y`, and, for 3-D problems, `location.z`. Your function must return a matrix `u0` of size N-by-`M`, where N is the number of equations in your PDE and `M = length(location.x)`. The fields in `location` 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(location)

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

```
function utinit = ut0fun(location)

M = length(location.x);
utinit = zeros(2,M);
utinit(2,:) = sin(location.x.*location.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)
```

See Also

Related Examples

- “Solve Problems Using PDEModel Objects” on page 2-2
- “Wave Equation on Square Domain” on page 3-298
- “Inhomogeneous Heat Equation on Square Domain” on page 3-277
- “Heat Distribution in Circular Cylindrical Rod” on page 3-281
- “Heat Transfer Problem with Temperature-Dependent Properties” on page 3-262
- “Dynamic Analysis of Clamped Beam” on page 3-28

Nonlinear System with Cross-Coupling Between Components

This example shows how to solve a nonlinear PDE system of two equations with cross-coupling between the two components. The system is a Schnakenberg system

$$\frac{\partial u_1}{\partial t} - D_1 \Delta u_1 = \kappa\left(a - u_1 + {u_1}^2 u_2\right)$$
$$\frac{\partial u_2}{\partial t} - D_2 \Delta u_2 = \kappa\left(b - {u_1}^2 u_2\right)$$

with the steady-state solution $u_{1S} = a + b$ and $u_{2S} = \frac{b}{(a+b)^2}$. The initial conditions are a small perturbation of the steady-state solution.

Solution for First Time Span

First, create a PDE model for a system of two equations.

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

Create a cubic geometry and assign it to the model.

```
gm = multicuboid(1,1,1);
model.Geometry = gm;
```

Generate the mesh using the linear geometric order to save memory.

```
generateMesh(model,"GeometricOrder","linear");
```

Define the parameters of the system.

```
D1 = 0.05;
D2 = 1;
kappa = 100;
a = 0.2;
b = 0.8;
```

Based on these parameters, specify the PDE coefficients in the toolbox format.

```
d = [1;1];
c = [D1;D2];
f = @(region,state) [kappa*(a - state.u(1,:) + ...
                            state.u(1,:).^2.*state.u(2,:));
                     kappa*(b - state.u(1,:).^2.*state.u(2,:))
                    ];
specifyCoefficients(model,"m",0,"d",d,"c",c,"a",0,"f",f);
```

Set the initial conditions. The first component is a small perturbation of the steady-state solution $u_{1S} = a + b$. The second component is the steady-state solution $u_{2S} = \frac{b}{(a+b)^2}$.

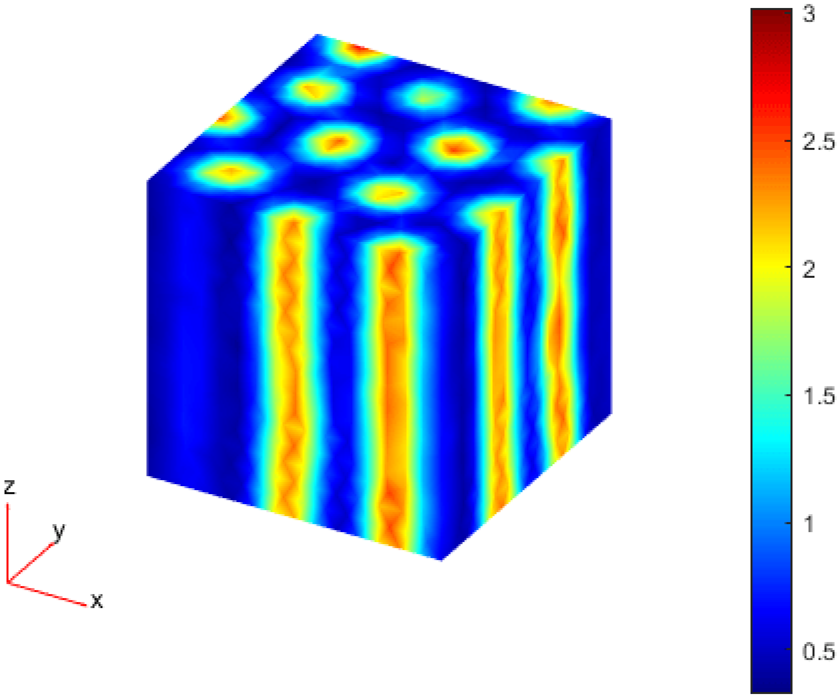
```
icFcn = @(region) [a + b + 10^(-3)*exp(-100*((region.x - 1/3).^2 ...
                   + (region.y - 1/2).^2)); ...
                   (b/(a + b)^2)*ones(size(region.x))];

setInitialConditions(model,icFcn);
```

Solve the system for times 0 through 2 seconds.

```
tlist = linspace(0,2,10);
results = solvepde(model,tlist);
```

Plot the first component of the solution at the last time step.

```
pdeplot3D(model,"ColorMapData",results.NodalSolution(:,1,end));
```



Initial Condition for Second Time Span Based on Previous Solution

Now, resume the analysis and solve the problem for times from 2 to 5 seconds. Reduce the magnitude of the previously obtained solution for time 2 seconds to 10% of the original value.

```
u2 = results.NodalSolution(:,:,end);
newResults = createPDEResults(model,u2(:)*0.1);
```

Use `newResults` as the initial condition for further analysis.

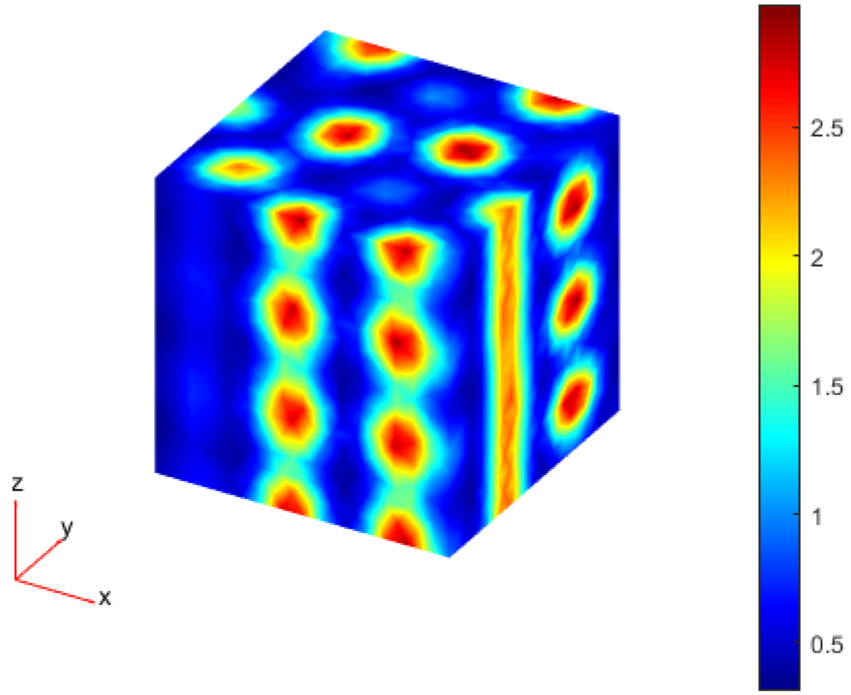
```
setInitialConditions(model,newResults);
```

Solve the system for times 2 through 5 seconds.

```
tlist = linspace(2,5,10);
results25 = solvepde(model,tlist);
```

Plot the first component of the solution at the last time step.

```
figure
pdeplot3D(model,"ColorMapData",results25.NodalSolution(:,1,end));
```



Alternatively, you can write a function that uses the results returned by the solver and computes the initial conditions based on the results of the previous analysis.

```
NewIC = @(location) computeNewIC(results)
```

```
NewIC = function_handle with value:
    @(location)computeNewIC(results)
```

Remove the previous initial conditions.

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

Set the initial conditions using the function `NewIC`.

```
setInitialConditions(model,NewIC)
```

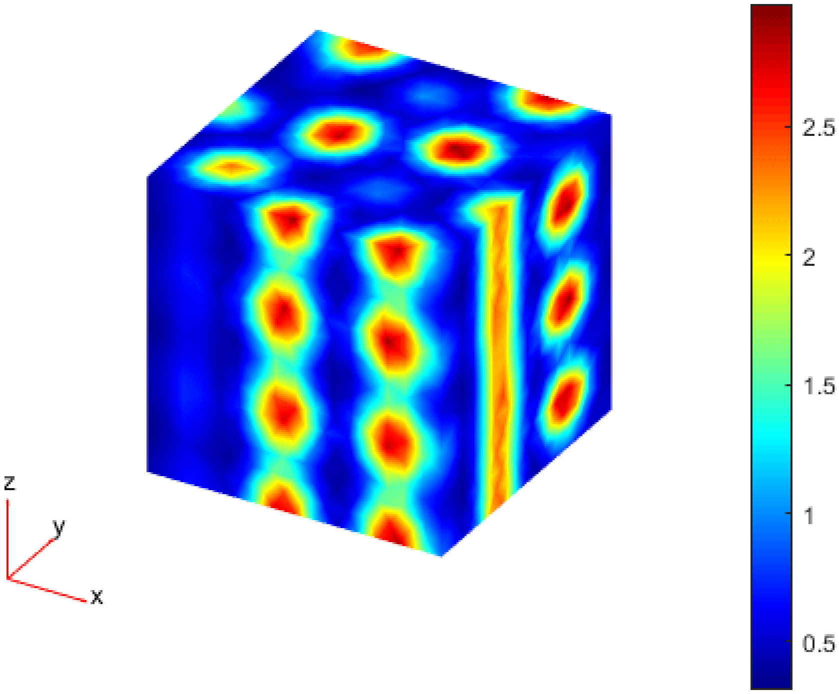
```
ans = 
  GeometricInitialConditions with properties:

           RegionType: 'cell'
             RegionID: 1
         InitialValue: @(location)computeNewIC(results)
    InitialDerivative: []
```

Solve the system for times 2 through 5 seconds.

```
results25f = solvepde(model,tlist);
```

Plot the first component of the solution at the last time step.

```
figure
pdeplot3D(model,"ColorMapData",results25f.NodalSolution(:,1,end));
```



Function Computing Initial Conditions

```
function newU0 = computeNewIC(resultsObject)
newU0 = 0.1*resultsObject.NodalSolution(:,:,end).';
end
```

Set Initial Condition for Model with Fine Mesh Using Solution Obtained with Coarser Mesh

Set initial conditions for a model with a fine mesh by using the coarse-mesh solution from a previous analysis.

Create a PDE model and include the geometry of the built-in function `squareg`.

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

Specify the coefficients, apply boundary conditions, and set initial conditions.

```
specifyCoefficients(model,"m",0,"d",1,"c",5,"a",0,"f",0.1);
applyBoundaryCondition(model,"dirichlet","Edge",1,"u",1);
setInitialConditions(model,10);
```

Generate a comparatively coarse mesh with the target maximum element edge length of 0.1.

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

Solve the model for the entire time span of 0 through 0.02 seconds.

```
tlist = linspace(0,2E-2,20);
Rtotal = solvepde(model,tlist);
```

Interpolate the solution at the origin for the entire time span.

```
singleSpanSol = Rtotal.interpolateSolution(0,0,1:numel(tlist));
```

Now solve the model for the first half of the time span. You will use this solution as an initial condition when solving the model with a finer mesh for the second half of the time span.

```
tlist1 = linspace(0,1E-2,10);
R1 = solvepde(model,tlist1);
```

Create an interpolant to interpolate the initial condition.

```
x = model.Mesh.Nodes(1,:)';
y = model.Mesh.Nodes(2,:)';
interpolant = scatteredInterpolant(x,y,R1.NodalSolution(:,end));
```

Generate a finer mesh by setting the target maximum element edge length to 0.05.

```
generateMesh(model,"Hmax",0.05);
```

Use the coarse mesh model results as the initial condition for the model with the finer mesh. For the definition of the `icFcn` function, see Initial Conditions Function on page 2-122.

```
setInitialConditions(model,@(region) icFcn(region,interpolant));
```

Solve the model for the second half of the time span.

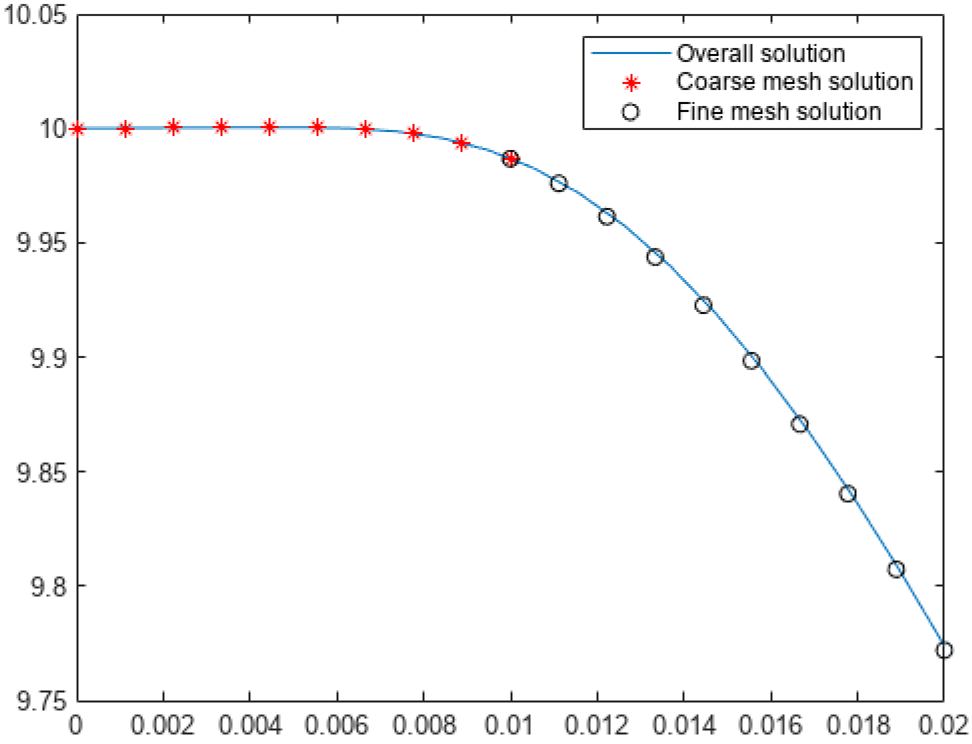
```
tlist2 = linspace(1E-2,2E-2,10);
R2 = solvepde(model,tlist2);
```

Interpolate the solutions at the origin for the first and the second halves of the time span.

```
multispanSol1 = R1.interpolateSolution(0,0,1:numel(tlist1));
multispanSol2 = R2.interpolateSolution(0,0,1:numel(tlist2));
```

Plot all three solutions at the origin.

```
figure
plot(tlist,singleSpanSol)
hold on
plot(tlist1, multispanSol1,"r*")
plot(tlist2, multispanSol2,"ko")
legend("Overall solution","Coarse mesh solution", "Fine mesh solution")
```



Initial Conditions Function

```
function u0 = icFcn(region,interpolant)
u0 = interpolant(region.x',region.y');
end
```

View, Edit, and Delete Initial Conditions

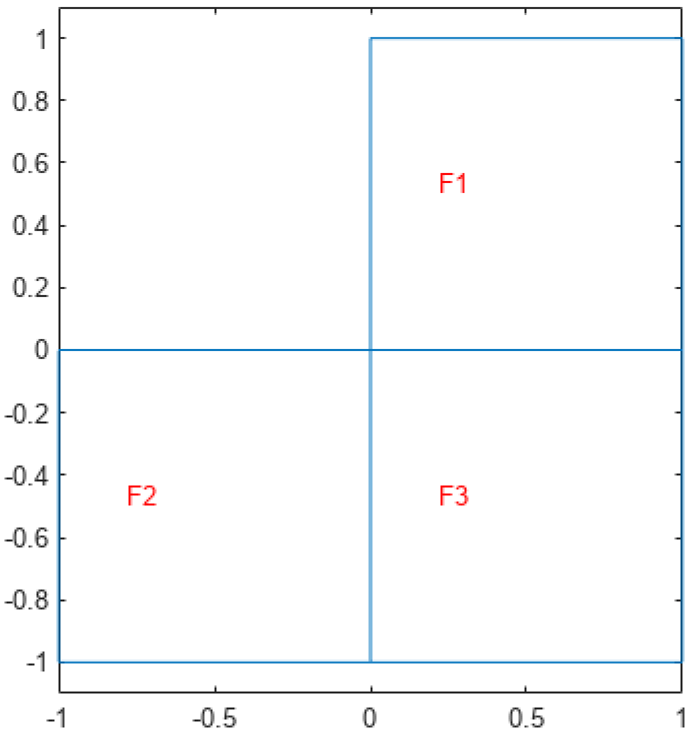
View Initial Conditions

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 = @ut0fun;
```

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`.

No Boundary Conditions Between Subdomains

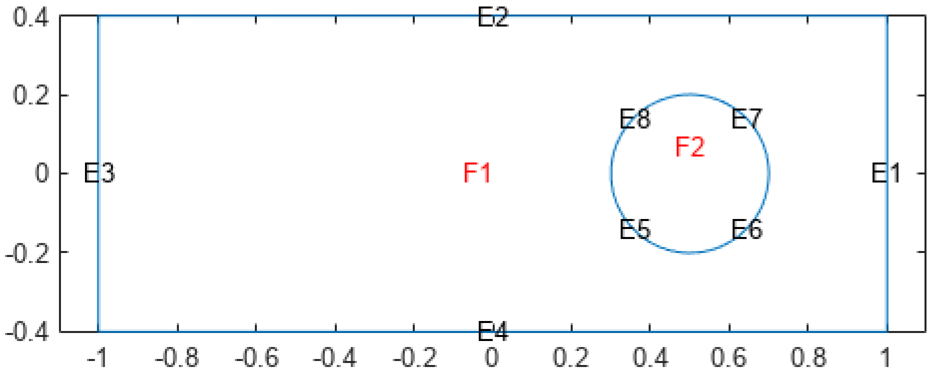
There are two types of boundaries:

- 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 Basics” on page 1-11. In the weak formulation you do not specify boundary conditions between subdomains, even if coefficients are discontinuous between subdomains. So the toolbox does not support defining boundary conditions on subdomain boundaries.

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

```
% 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", ...
            "FaceLabels","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

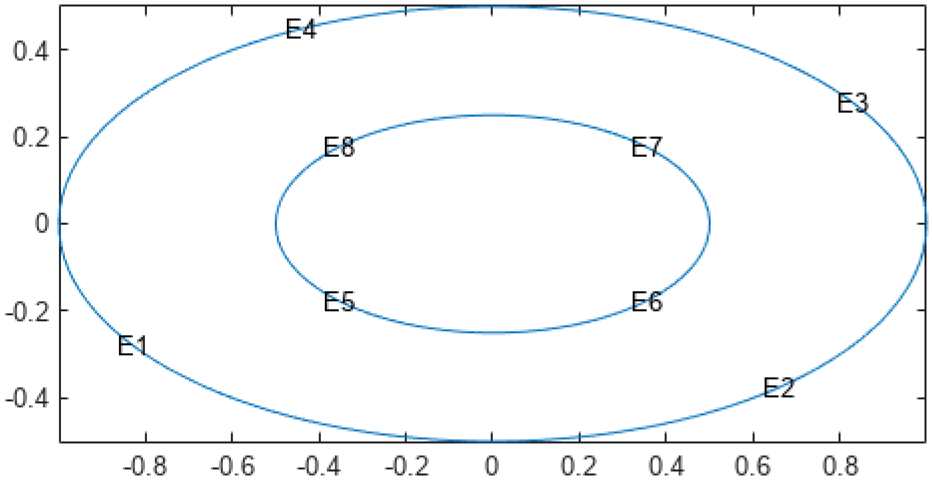
You can see the edge labels by using the `pdegplot` function with the `EdgeLabels` name-value pair set to "on":

```
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 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")
```



Specify Boundary Conditions

Before you create boundary conditions, you need to create a `PDEModel` container. For details, see "Solve Problems Using PDEModel Objects" on page 2-2. 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 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"`.

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 2-132.

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:
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.

```
% 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 $\mathbf{hu} = \mathbf{r}$, where $\mathbf{h}$ is a matrix, $\mathbf{u}$ is the solution vector, and $\mathbf{r}$ 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.

```
% 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.

```
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 form of the N-by-N matrix. For details about the linear indexing form, see "Array Indexing". 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

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

$\overrightarrow{n}$ 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

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

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 form of the *N*-by-*N* matrix. For details about the linear indexing form, see “Array Indexing”. 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(location,state)
```

`solvepde` or `solvepdeeig` compute and populate the data in the `location` and `state` structure arrays and pass this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments.

- `location` — A structure containing the following fields. If you pass a name-value pair to `applyBoundaryCondition` with `Vectorized` set to `"on"`, then `location` 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.

  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.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 `location` structure.

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points
  - `location.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. If there is no `"EquationIndex"` argument, then $N1 = N$.
- `"r"` or `"g"` — *N*-by-*M* matrix, where each column corresponds to one evaluation point, and *M* is the number of evaluation points.
- `"h"` or `"q"` — N^2-by-*M* matrix, where each column corresponds to one evaluation point via linear indexing of the underlying *N*-by-*N* matrix, and *M* is the number of evaluation points. Alternatively, an *N*-by-*N*-by-*M* array, where each evaluation point is an *N*-by-*N* matrix. For details about linear indexing, see "Array Indexing".

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 “Specify Nonconstant Boundary Conditions” on page 2-139.

Additional Arguments in Functions for Nonconstant Boundary Conditions

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
uVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
applyBoundaryCondition(model,"mixed", ...
                             "Edge",[3,4], ...
                             "u",uVal, ...
                             "EquationIndex",[2,3]);
```

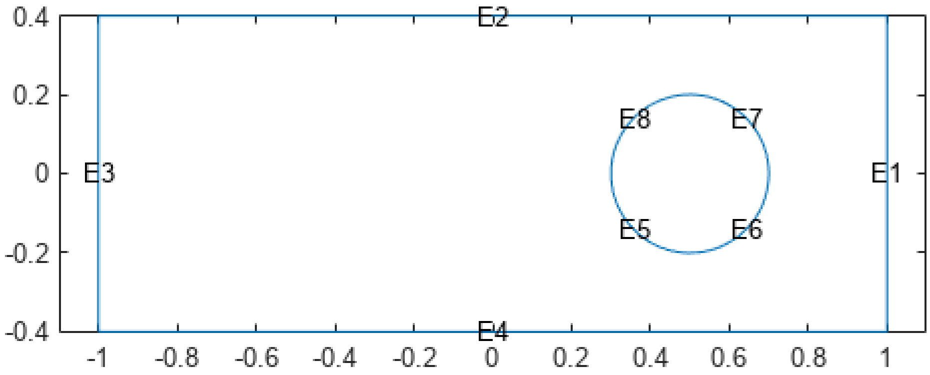
Solve PDEs with Constant Boundary Conditions

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

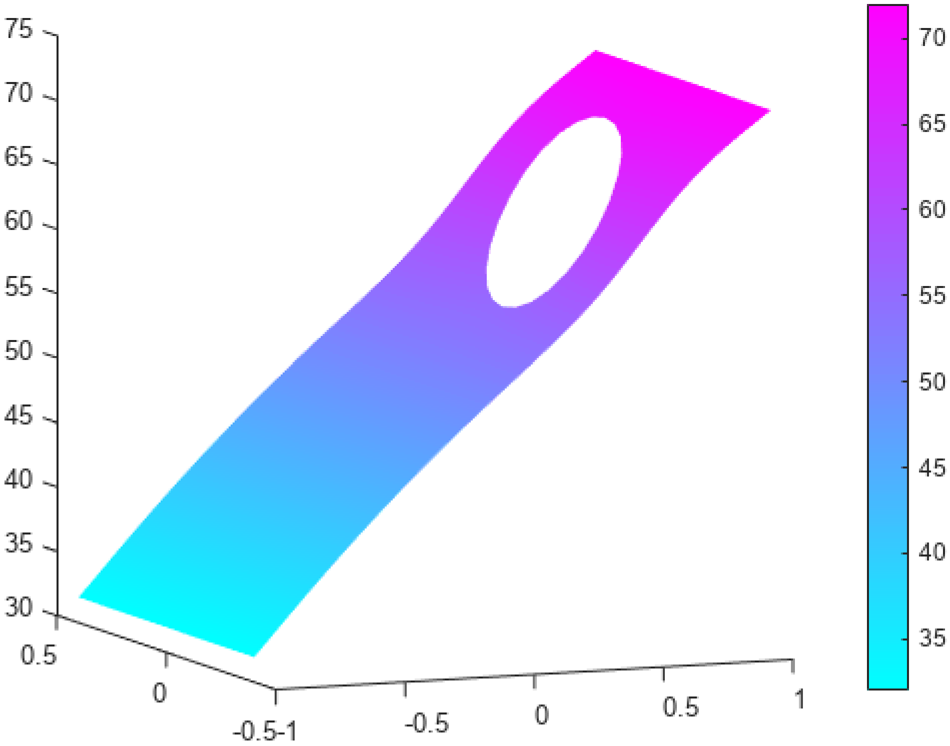
Suppose that edge 3 has Dirichlet conditions with value 32, edge 1 has Dirichlet conditions with value 72, and all other edges have Neumann boundary conditions with `q = 0`, `g = -1`.

```
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 the system has `N = 2`.

- Edge 3 has Dirichlet conditions with values [32,72].
- Edge 1 has Dirichlet conditions with values [72,32].
- Edge 4 has a Dirichlet condition for the first component with value 52, and has a Neumann condition for the second component with `q = 0`, `g = -1`.
- Edge 2 has Neumann boundary conditions with `q = [1,2;3,4]` and `g = [5,-6]`.
- The circular edges (edges 5 through 8) have `q = 0` and `g = 0`.

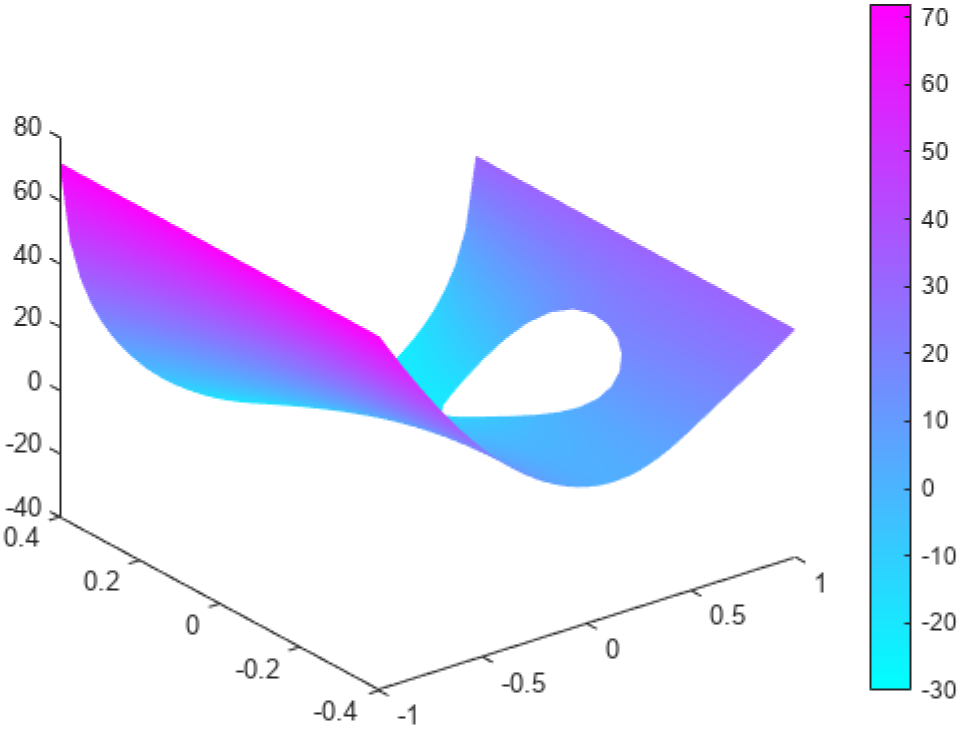
```
model = createpde(2);
geometryFromEdges(model,g);

applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",3,"u",[32,72]);
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1,"u",[72,32]);
applyBoundaryCondition(model,"mixed", ...
                             "Edge",4,"u",52, ...
                             "EquationIndex",1,"g",[0,-1]);
Q2 = [1,2;3,4];
G2 = [5,-6];
applyBoundaryCondition(model,"neumann", ...
                             "Edge",2, ...
                             "q",Q2,"g",G2);
```

```
% The next step is optional,
% because it sets "g" to its default value
applyBoundaryCondition(model,"neumann", ...
                             "Edge",5:8,"g",[0,0]);
```

This completes the boundary condition specification.

Solve an elliptic PDE with these boundary conditions using `c = 1`, `a = 0`, and `f = [10;-10]`. Because the shorter rectangular side has length 0.8, to ensure that the mesh is not too coarse choose a maximum mesh size `Hmax = 0.1`.

```
specifyCoefficients(model,"m",0,"d",0,"c",1, ...
                          "a",0,"f", [10;-10]);
generateMesh(model,"Hmax",0.1);
results = solvepde(model);
u = results.NodalSolution;
pdeplot(model,"XYData",u(:,2),"ZData",u(:,2))
```



See Also

More About

- "Specify Boundary Conditions" on page 2-129
- "Specify Nonconstant Boundary Conditions" on page 2-139

Specify Nonconstant Boundary Conditions

When solving PDEs with nonconstant boundary conditions, specify these conditions by using function handles. This example shows how to write functions to represent nonconstant boundary conditions for PDE problems.

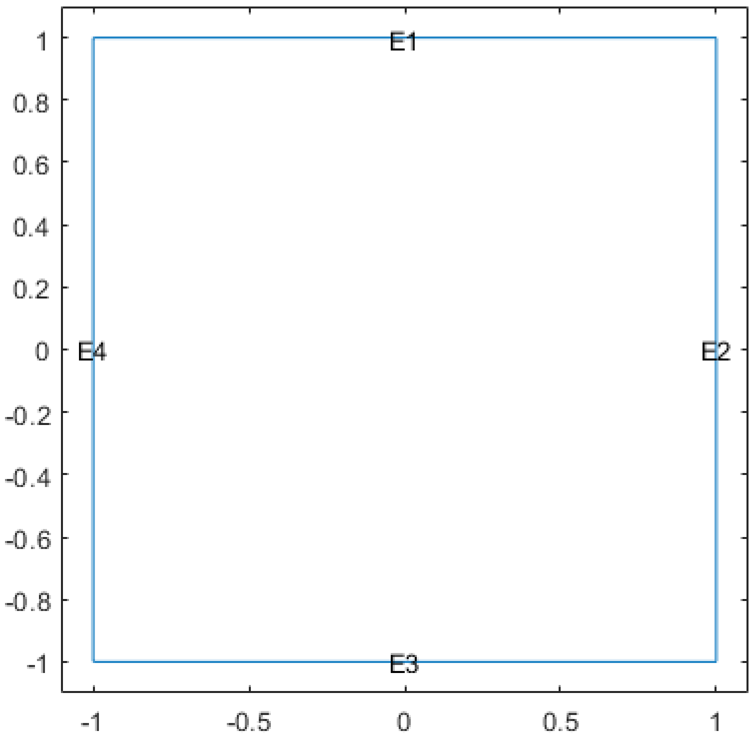
Geometry and Mesh

Create a model.

```
model = createpde;
```

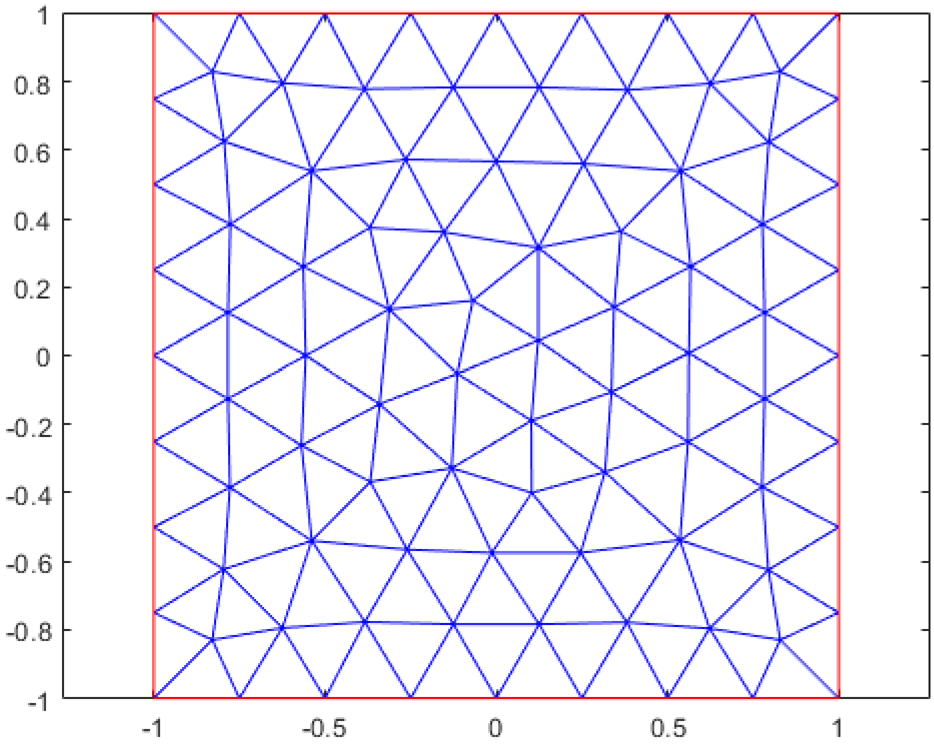
Include a unit square geometry in the model and plot the geometry.

```
geometryFromEdges(model,@squareg);
pdegplot(model,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Generate a mesh with a maximum edge length of 0.25. Plot the mesh.

```
generateMesh(model,"Hmax",0.25);
figure
pdemesh(model)
```



Scalar PDE Problem with Nonconstant Boundary Conditions

Write functions to represent the nonconstant boundary conditions on edges 1 and 3. Each function must accept two input arguments, `location` and `state`. The solvers automatically compute and populate the data in the `location` and `state` structure arrays and pass this data to your function.

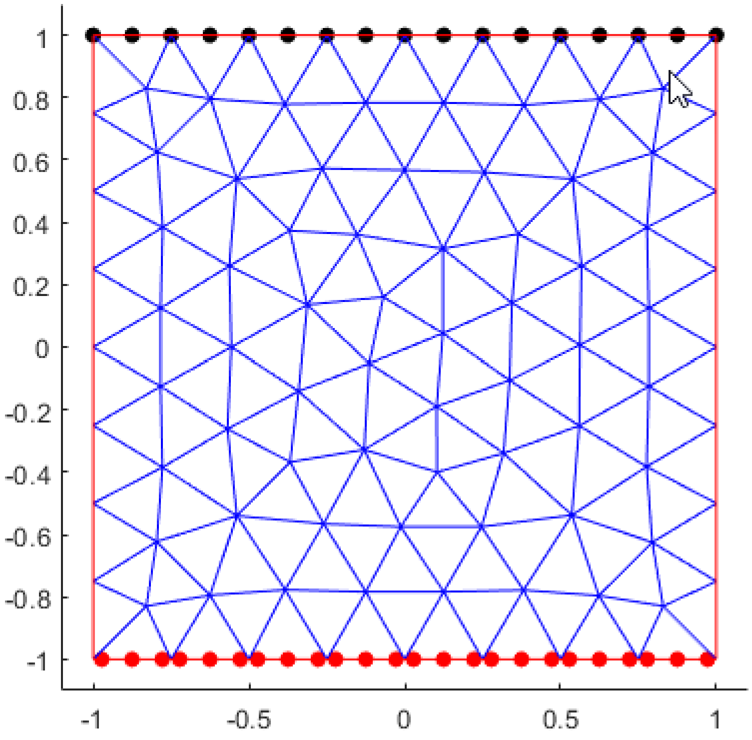
Write a function that returns the value $u(x, y) = 52 + 20x$ to represent the Dirichlet boundary condition for edge 1.

```
function bc = bcfuncD(location,state)
    bc = 52 + 20*location.x;
    scatter(location.x,location.y,"filled","black");
    hold on
end
```

Write a function that returns the value $u(x, y) = \cos\left(x^2\right)$ to represent the Neumann boundary condition for edge 3.

```
function bc = bcfuncN(location,state)
    bc = cos(location.x).^2;
    scatter(location.x,location.y,"filled","red");
    hold on
end
```

The scatter plot command in each of these functions enables you to visualize the `location` data used by the toolbox. For Dirichlet boundary conditions, the `location` data (black markers on edge 1) corresponds with the mesh nodes. Each element of a quadratic mesh has nodes at its corners and edge centers. For Neumann boundary conditions, the `location` data (red markers on edge 3) corresponds with the quadrature integration points.



Specify the boundary condition for edges 1 and 3 using the functions that you wrote.

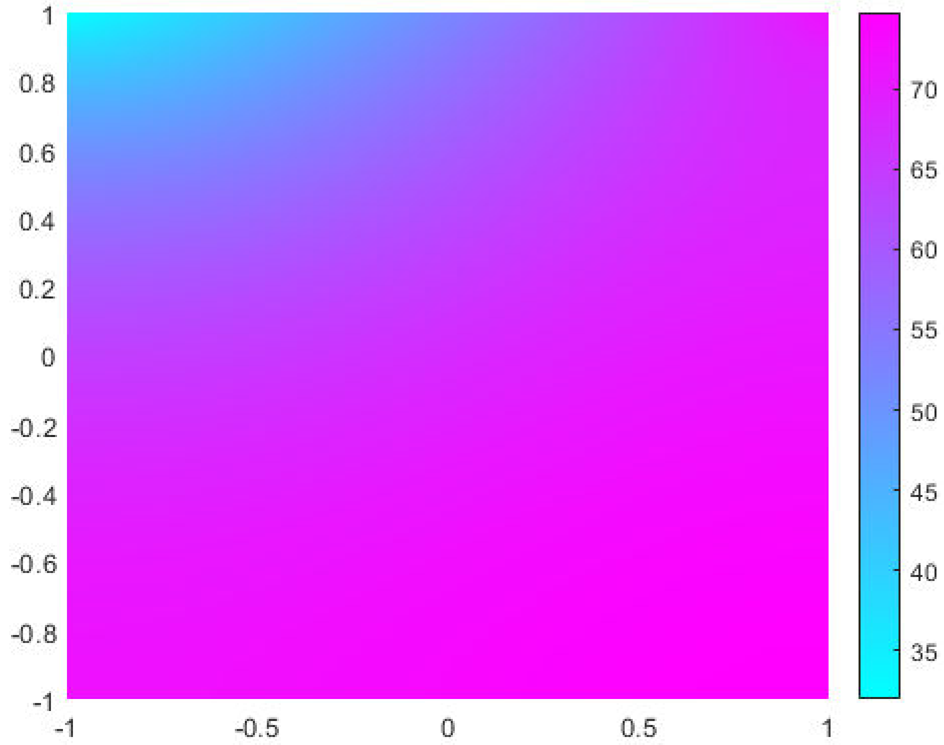
```
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1, ...
                             "u",@bcfuncD);
applyBoundaryCondition(model,"neumann", ...
                             "Edge",3, ...
                             "g",@bcfuncN);
```

Specify the PDE coefficients.

```
specifyCoefficients(model,"m",0,"d",0,"c",1, ...
                          "a",0,"f",10);
```

Solve the equation and plot the solution.

```
results = solvepde(model);

figure
pdeplot(model,"XYData",results.NodalSolution)
```



Anonymous Functions for Nonconstant Boundary Conditions

If the dependency of a boundary condition on coordinates, time, or solution is simple, you can use an anonymous function to represent the nonconstant boundary condition. Thus, you can implement the linear interpolation shown earlier in this example as the `bcfuncD` function, as this anonymous function.

```
bcfuncD = @(location,state)52 + 20*location.x;
```

Specify the boundary condition for edge 1.

```
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1, ...
                             "u",bcfuncD);
```

Additional Arguments

If a function that represents a nonconstant boundary condition requires more arguments than `location` and `state`, follow these steps:

1. Write a function that takes the `location` and `state` arguments and the additional arguments.
2. Wrap that function with an anonymous function that takes only the `location` and `state` arguments.

For example, define boundary conditions on edge 1 as $u(x, y) = 52a^2 + 20bx + c$. First, write the function that takes the arguments `a`, `b`, and `c` in addition to the `location` and `state` arguments.

```
function bc = bcfunc_abc(location,state,a,b,c)
    bc = 52*a^2 + 20*b*location.x + c;
end
```

Because a function defining nonconstant boundary conditions must have exactly two arguments, wrap the `bcfunc_abc` function with an anonymous function.

```
bcfunc_add_args = @(location,state) bcfunc_abc(location,state,1,2,3);
```

Now you can use `bcfunc_add_args` to specify a boundary condition for edge 1.

```
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1, ...
                             "u",bcfunc_add_args);
```

System of PDEs

Create a model for a system of two equations.

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

Use the same unit square geometry that you used for the scalar PDE problem.

```
geometryFromEdges(model,@squareg);
```

The first component on edge 1 satisfies the equation $u_1(x, y) = 52 + 20x + 10\sin\left(\pi x^3\right)$. The second component on edge 1 satisfies the equation $u_1(x, y) = 52 - 20x - 10\sin\left(\pi x^3\right)$.

Write a function file `myufun.m` that incorporates these equations.

```
function bcMatrix = myufun(location,state)
bcMatrix = [52 + 20*location.x + 10*sin(pi*(location.x.^3));
            52 - 20*location.x - 10*sin(pi*(location.x.^3))];
end
```

Specify the boundary condition for edge 1 using the `myufun` function.

```
applyBoundaryCondition(model,"dirichlet","Edge",1, ...
                                         "u",@myufun);
```

Specify the PDE coefficients.

```
specifyCoefficients(model,"m",0,"d",0,"c",1, ...
                          "a",0,"f",[10;-10]);
```

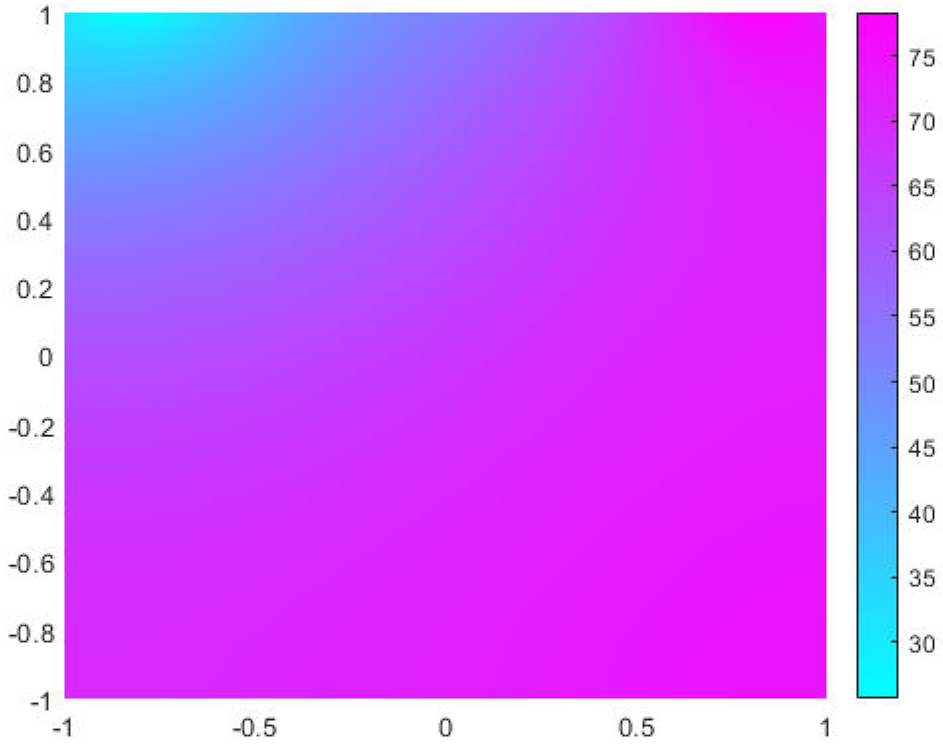
Generate a mesh.

```
generateMesh(model);
```

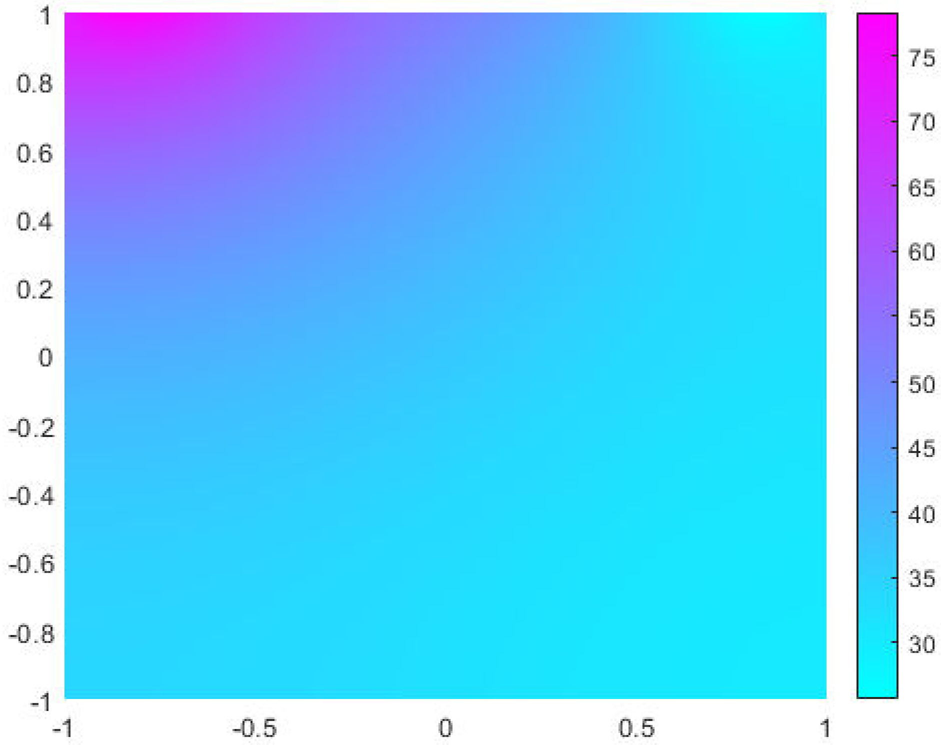
Solve the system and plot the solution.

```
results = solvepde(model);
u = results.NodalSolution;
```

```
figure
pdeplot(model,"XYData",u(:,1))
```



```
figure
pdeplot(model,"XYData",u(:,2))
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.5
0
0.5
1
75
70
65
60
55
50
45
40
35
30

Specify Nonconstant PDE Coefficients

When solving PDEs with nonconstant coefficients, specify these coefficients by using function handles. This example shows how to write functions to represent nonconstant coefficients for PDE problems.

Geometry and Mesh

Create a model.

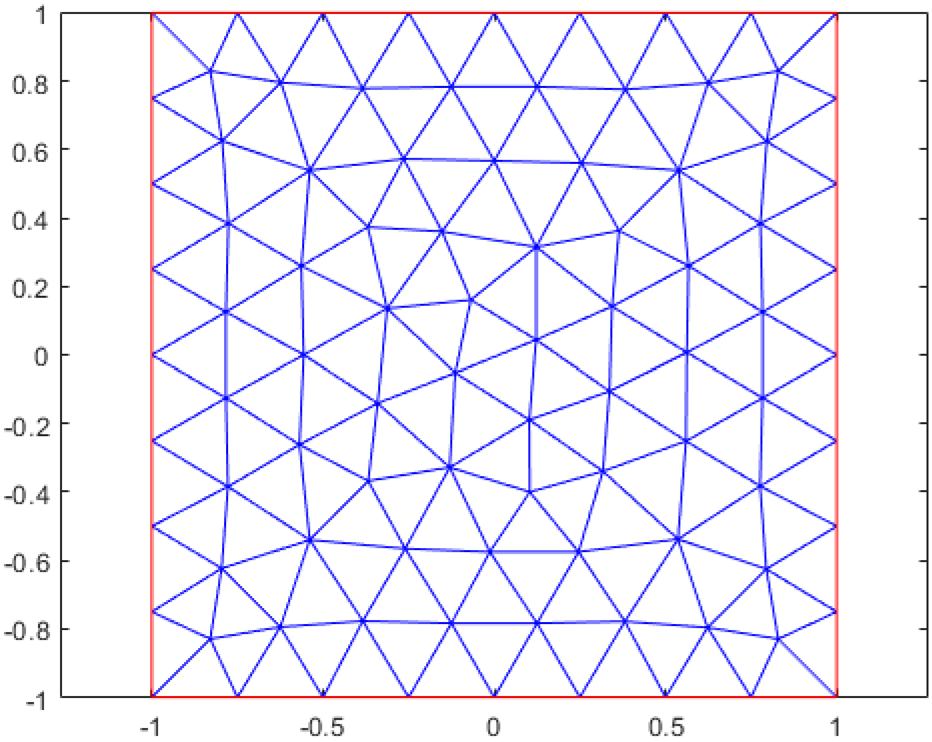
```
model = createpde;
```

Include a unit square geometry in the model.

```
geometryFromEdges(model,@squareg);
```

Generate a mesh with a maximum edge length of 0.25. Plot the mesh.

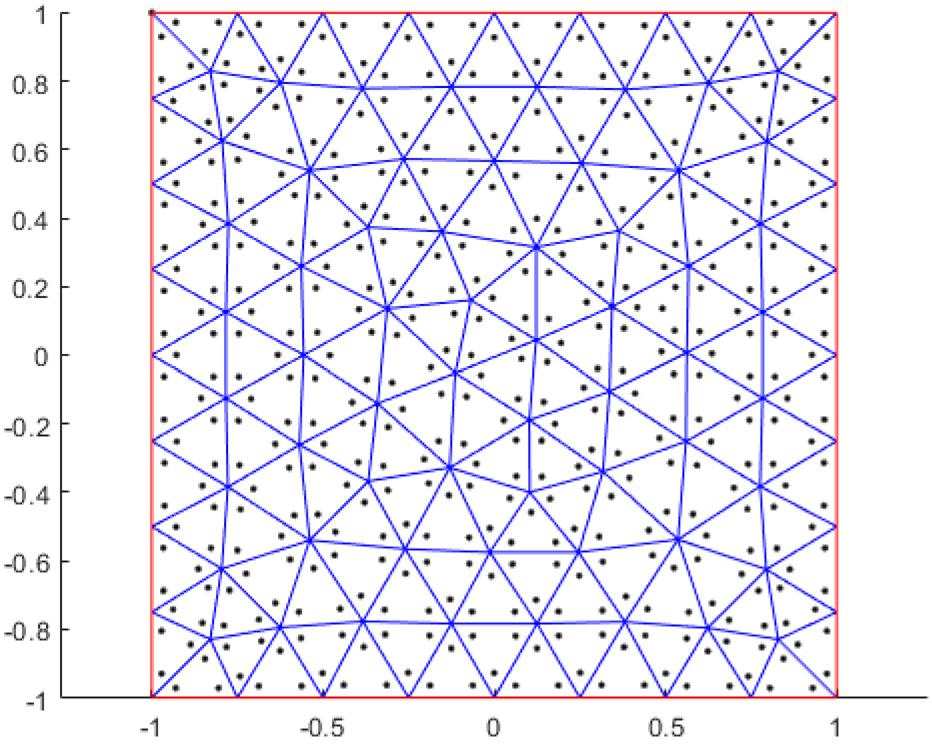
```
generateMesh(model,"Hmax",0.25);
pdemesh(model)
```



Function for Nonconstant Coefficient f

Write a function that returns the value $f(x, y) = x^2\sin(y)$ for the nonconstant coefficient f. The function must accept two input arguments, `location` and `state`. The solvers automatically compute and populate the data in the `location` and `state` structure arrays and pass this data to your function. To visualize the `location` data used by the toolbox, add the scatter plot command to your function.

```
function fcoeff = fcoefffunc(location,state)
    fcoeff = location.x.^2.*sin(location.y);
    scatter(location.x,location.y,".","black");
    hold on
end
```



Specify the PDE coefficients using the function that you wrote for the *f* coefficient.

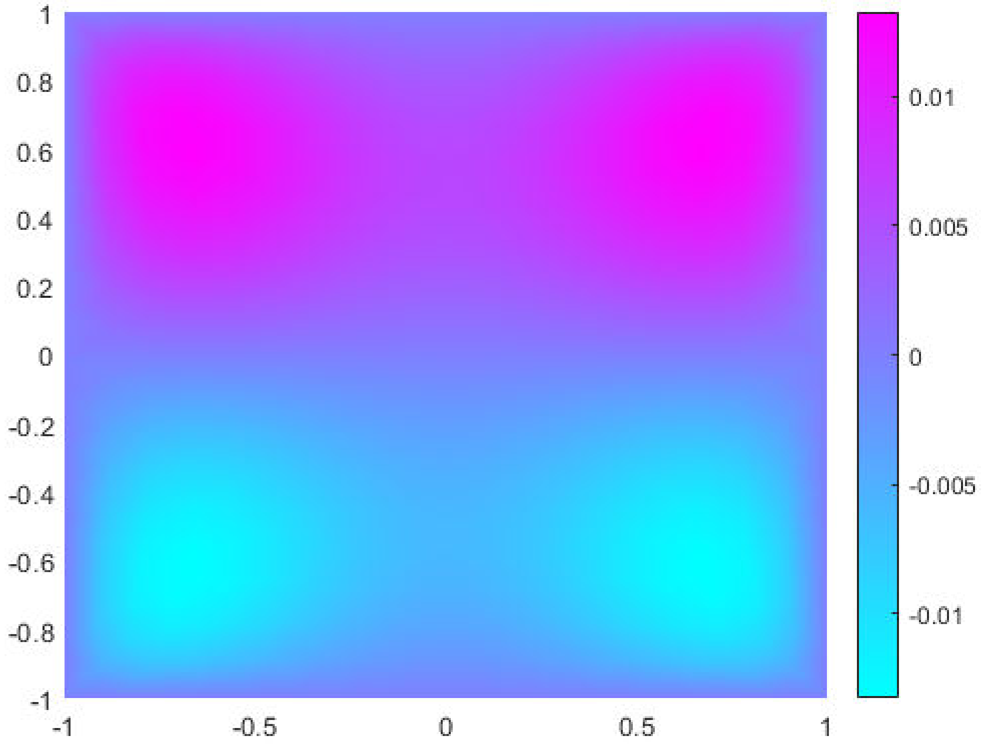
```
specifyCoefficients(model,"m",0, ...
                          "d",0, ...
                          "c",1, ...
                          "a",0, ...
                          "f",@fcoefffunc);
```

Apply the Dirichlet boundary condition $u = 0$ for all edges of the square.

```
applyBoundaryCondition(model,"dirichlet","Edge",1:4,"u",0);
```

Solve the equation and plot the solution.

```
results = solvepde(model);
figure
pdeplot(model,"XYData",results.NodalSolution)
```



Anonymous Function for a PDE Coefficient

If the dependency of a coefficient on coordinates, time, or solution is simple, you can use an anonymous function to represent the nonconstant PDE coefficient. Thus, you can implement the dependency shown earlier in this example as the `fcoefffunc` function, as this anonymous function.

```
f = @(location,state)location.x.^2.*sin(location.y);
```

Specify the PDE coefficients.

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

Additional Arguments

If a function that represents a nonconstant PDE coefficient requires more arguments than `location` and `state`, follow these steps:

1. Write a function that takes the `location` and `state` arguments and the additional arguments.
2. Wrap that function with an anonymous function that takes only the `location` and `state` arguments.

For example, define the coefficient *f* as $f(x, y) = ax^2\sin(by) + c$. First, write the function that takes the arguments `a`, `b`, and `c` in addition to the `location` and `state` arguments.

```
function fcoeff = fcoefffunc_abc(location,state,a,b,c)
    fcoeff = a*location.x.^2.*sin(b*location.y) + c;
end
```

Because functions defining nonconstant coefficients must have exactly two arguments, wrap the `fcoefffunc_abc` function with an anonymous function.

```
fcoefffunc_add_args = ...
@(location,state) fcoefffunc_abc(location,state,1,2,3);
```

Now you can use `fcoefffunc_add_args` to specify the coefficient *f*.

```
specifyCoefficients(model,"m",0, ...
                          "d",0, ...
                          "c",1, ...
                          "a",0, ...
                          "f",fcoefffunc_add_args);
```

View, Edit, and Delete Boundary Conditions

View Boundary Conditions

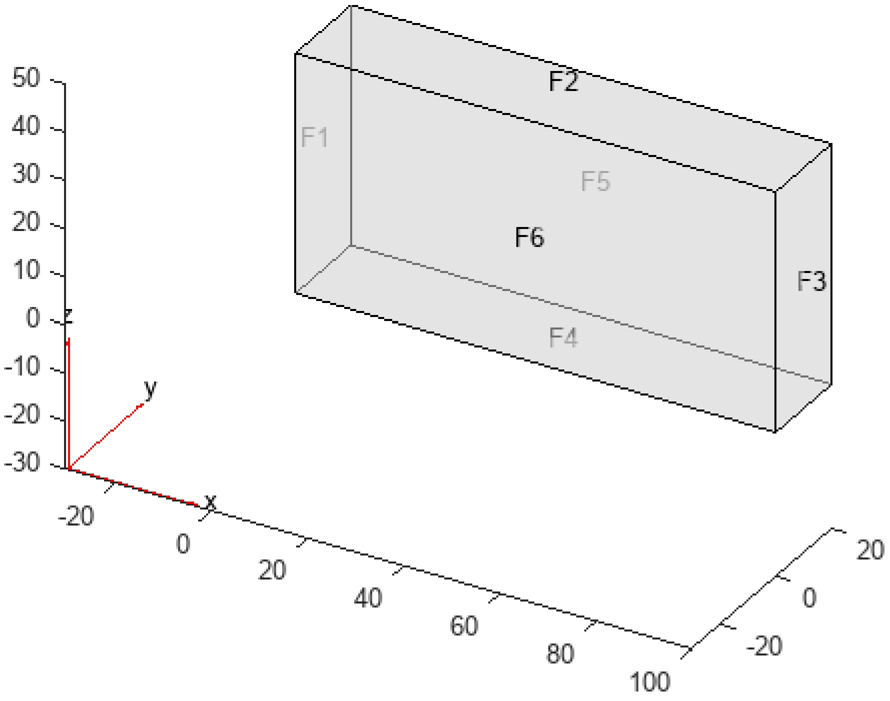
A PDE model stores boundary conditions in its `BoundaryConditions` property. To obtain the boundary conditions stored in the PDE model called `model`, use this syntax:

```
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: [3x1 double]
                h: [3x3 double]
                g: [3x1 double]
                q: [3x3 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.

See Also

Related Examples

- “Specify Boundary Conditions” on page 2-129

Generate Mesh

The `generateMesh` function creates a triangular mesh for a 2-D geometry and a tetrahedral mesh for a 3-D geometry. By default, the mesh generator uses internal algorithms to choose suitable sizing parameters for a particular geometry. You also can use additional arguments to specify the following parameters explicitly:

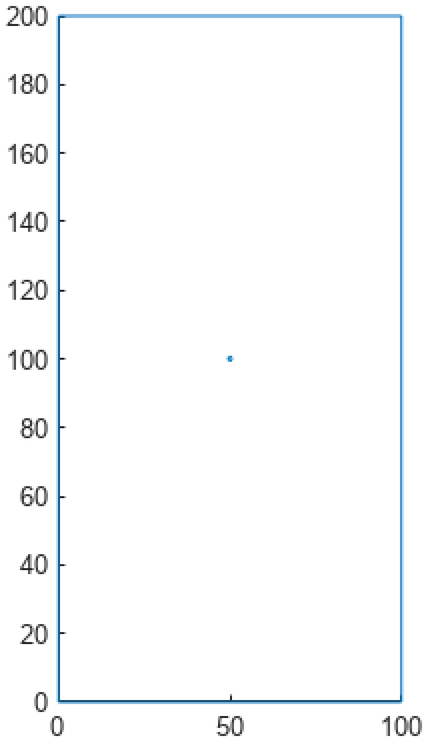
- Target maximum mesh edge length, which is an approximate upper bound on the mesh edge lengths. Note that occasionally, some elements can have edges longer than this parameter.
- Target minimum mesh edge length, which is an approximate lower bound on the mesh edge lengths. Note that occasionally, some elements can have edges shorter than this parameter.
- Mesh growth rate, which is the rate at which the mesh size increases away from the small parts of the geometry. The value must be between 1 and 2. This ratio corresponds to the edge length of two successive elements. The default value is 1.5, that is, the mesh size increases by 50%.
- Quadratic or linear geometric order. A quadratic element has nodes at its corners and edge centers, while a linear element has nodes only at its corners.

Create a PDE model.

```
model = createpde;
```

Include and plot the following geometry.

```
importGeometry(model,"PlateSquareHolePlanar.stl");
pdegplot(model)
```



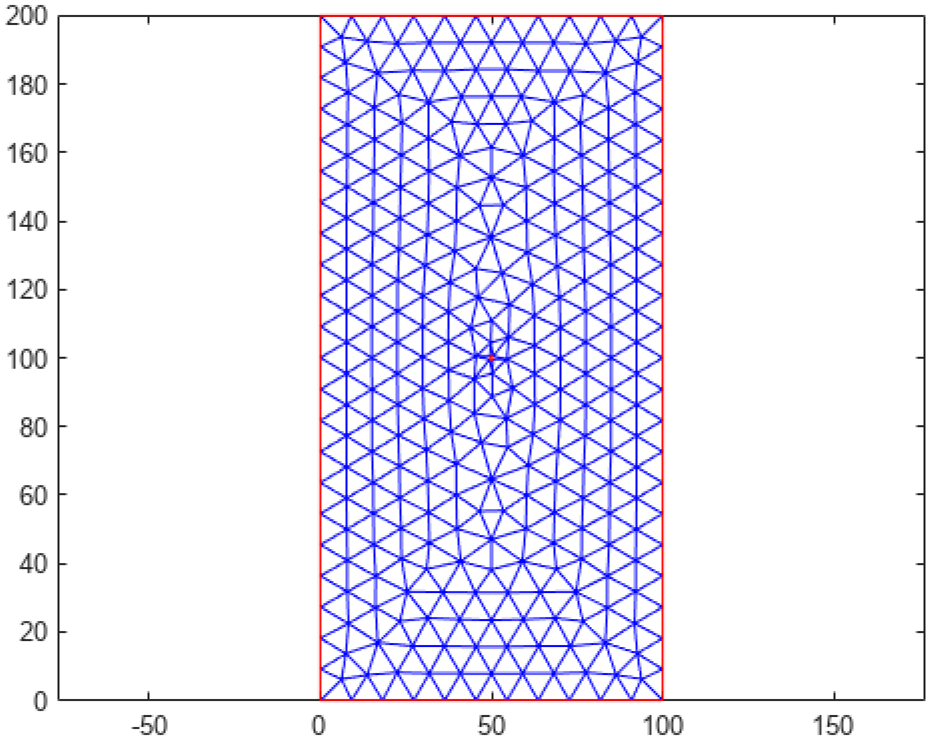
Generate a default mesh. For this geometry, the default target maximum and minimum mesh edge lengths are 8.9443 and 4.4721, respectively.

```
mesh_default = generateMesh(model)
```

```
mesh_default = 
  FEMesh with properties:

             Nodes: [2x1218 double]
          Elements: [6x574 double]
    MaxElementSize: 8.9443
    MinElementSize: 4.4721
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

View the mesh.

```
figure
pdemesh(mesh_default)
```



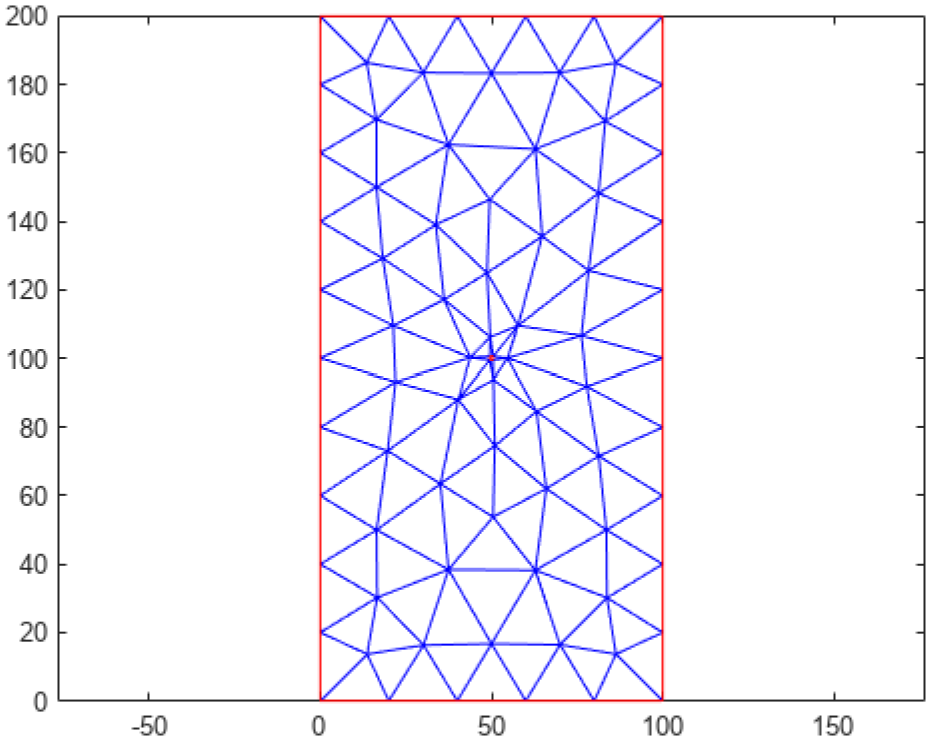
For comparison, create a mesh with the target maximum element edge length of 20.

```
mesh_Hmax = generateMesh(model,"Hmax",20)
```

```
mesh_Hmax = 
  FEMesh with properties:
```

```
            Nodes: [2x286 double]
         Elements: [6x126 double]
   MaxElementSize: 20
   MinElementSize: 10
    MeshGradation: 1.5000
   GeometricOrder: 'quadratic'
```

```
figure
pdemesh(mesh_Hmax)
```

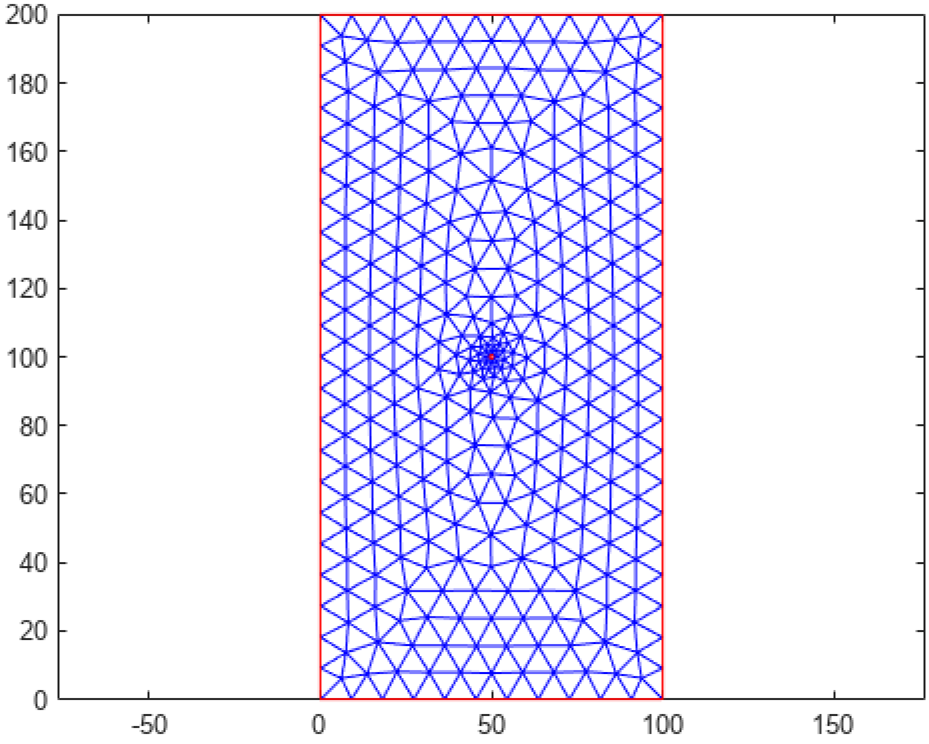


Now create a mesh with the target minimum element edge length of 0.5.

```
mesh_Hmin = generateMesh(model,"Hmin",0.5)
```

```
mesh_Hmin = 
  FEMesh with properties:

            Nodes: [2x1378 double]
         Elements: [6x654 double]
   MaxElementSize: 8.9443
   MinElementSize: 0.5000
    MeshGradation: 1.5000
   GeometricOrder: 'quadratic'
```

```
figure
pdemesh(mesh_Hmin)
```



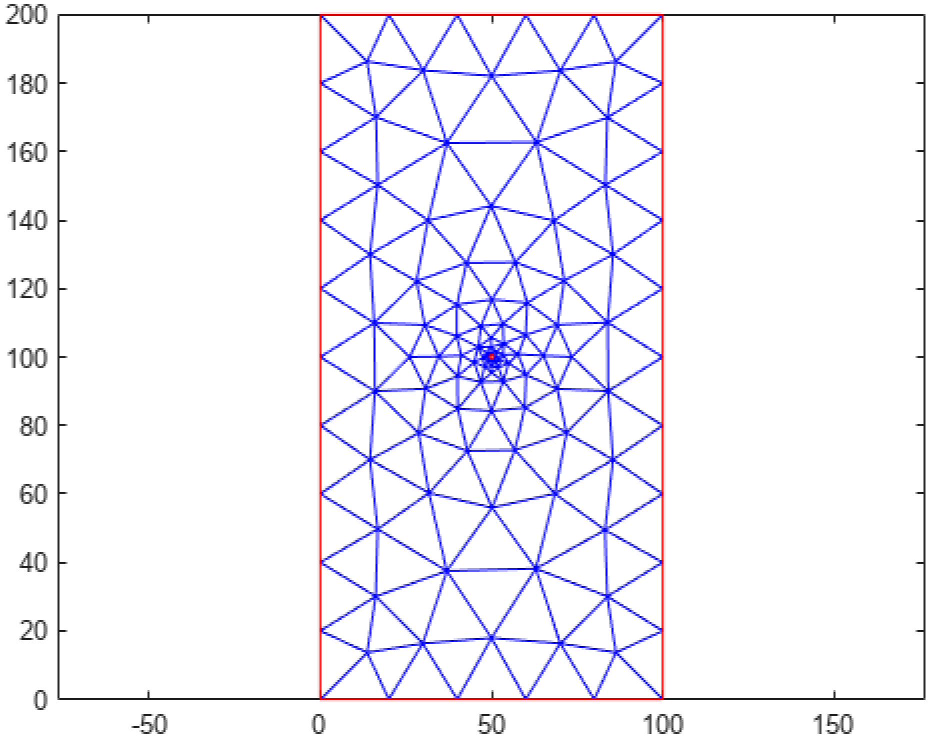
Create a mesh, specifying both the maximum and minimum element edge lengths instead of using the default values.

```
mesh_HminHmax = generateMesh(model,"Hmax",20, ...
                                   "Hmin",0.5)
```

```
mesh_HminHmax = 
  FEMesh with properties:

             Nodes: [2x458 double]
          Elements: [6x212 double]
    MaxElementSize: 20
    MinElementSize: 0.5000
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

View the mesh.

```
figure
pdemesh(mesh_HminHmax)
```

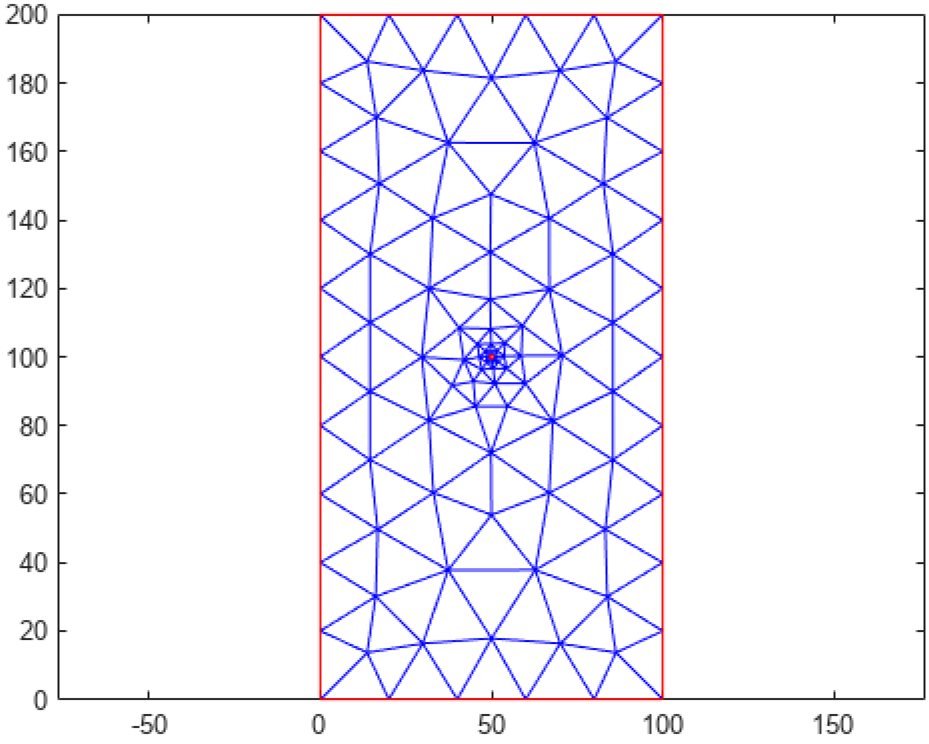


Create a mesh with the same maximum and minimum element edge lengths, but with the growth rate of 1.9 instead of the default value of 1.5.

```
mesh_Hgrad = generateMesh(model,"Hmax",20, ...
                                "Hmin",0.5, ...
                                "Hgrad",1.9)
```

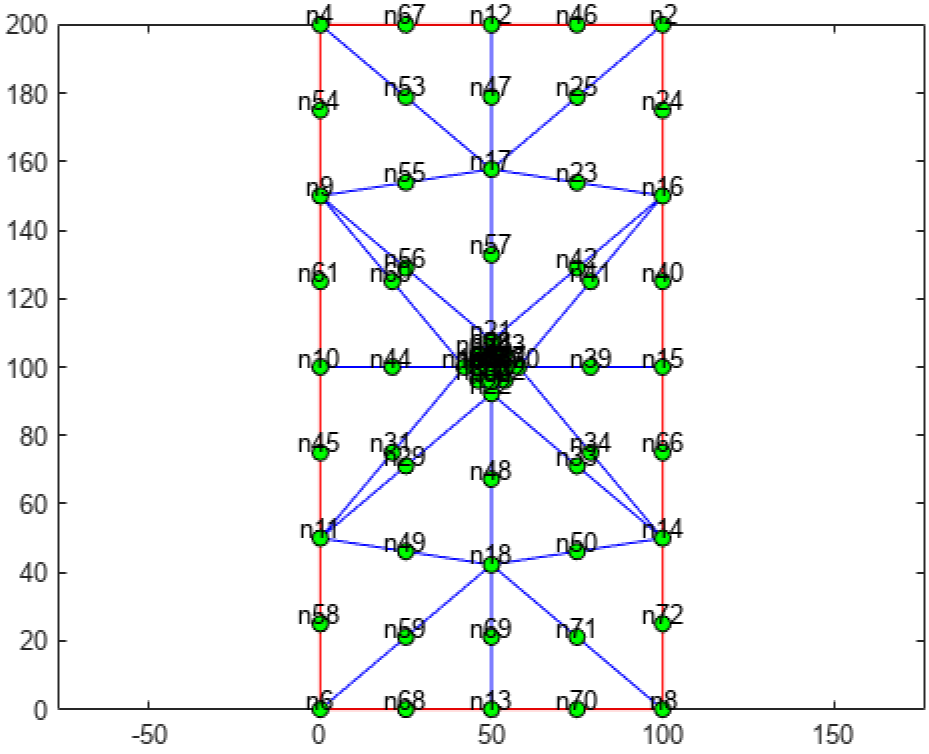
```
mesh_Hgrad = 
  FEMesh with properties:

             Nodes: [2x390 double]
          Elements: [6x178 double]
    MaxElementSize: 20
    MinElementSize: 0.5000
     MeshGradation: 1.9000
    GeometricOrder: 'quadratic'
```

```
figure
pdemesh(mesh_Hgrad)
```



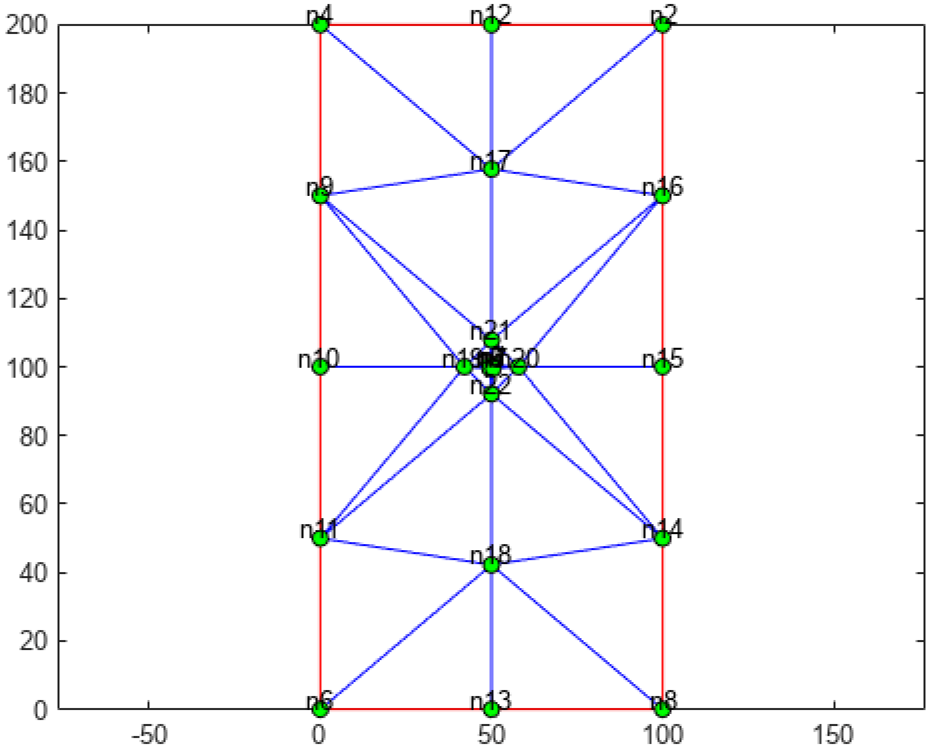
You also can choose the geometric order of the mesh. The toolbox can generate meshes made up of quadratic or linear elements. By default, it uses quadratic meshes, which have nodes at both the edge centers and corner nodes.

```
mesh_quadratic = generateMesh(model,"Hmax",50);
figure
pdemesh(mesh_quadratic,"NodeLabels","on")
hold on
plot(mesh_quadratic.Nodes(1,:), ...
     mesh_quadratic.Nodes(2,:), ...
     "ok","MarkerFaceColor","g")
```



To save memory or solve a 2-D problem using a legacy solver, override the default quadratic geometric order. Legacy PDE solvers require linear triangular meshes for 2-D geometries.

```
mesh_linear = generateMesh(model, ...
                           "Hmax",50, ...
                           "GeometricOrder","linear");
figure
pdemesh(mesh_linear,"NodeLabels","on")
hold on
plot(mesh_linear.Nodes(1,:), ...
     mesh_linear.Nodes(2,:), ...
     "ok","MarkerFaceColor","g")
```


n4
n12
n2
n17
n9
n16
n21
n10
n19
n20
n15
n22
n11
n14
n18
n6
n13
n8
200
180
160
140
120
100
80
60
40
20
0
-50
0
50
100
150

Find Mesh Elements and Nodes by Location

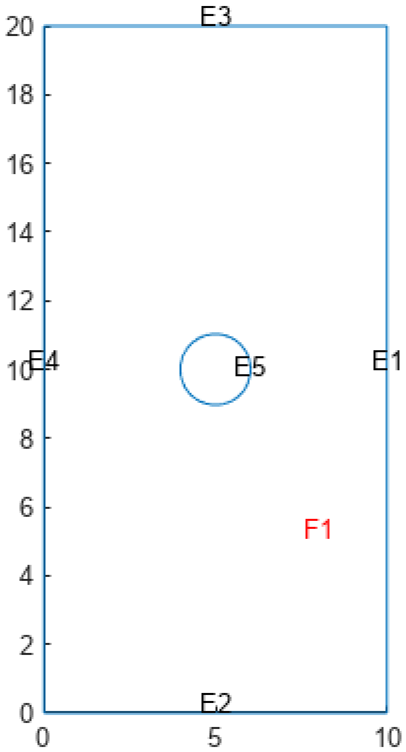
Partial Differential Equation Toolbox™ allows you to find mesh elements and nodes by their geometric location or proximity to a particular point or node. This example works with a group of elements and nodes located within the specified bounding disk.

Create a steady-state thermal model.

```
thermalmodel = createpde("thermal","steadystate");
```

Import and plot the geometry.

```
importGeometry(thermalmodel,"PlateHolePlanar.stl");
pdegplot(thermalmodel,"FaceLabels","on", ...
                      "EdgeLabels","on")
```



Assign the thermal conductivity of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",1);
```

Apply a constant temperature of $20^\circ C$ to the left edge and a constant temperature of $-10^\circ C$to the right edge. All other edges are insulated by default.

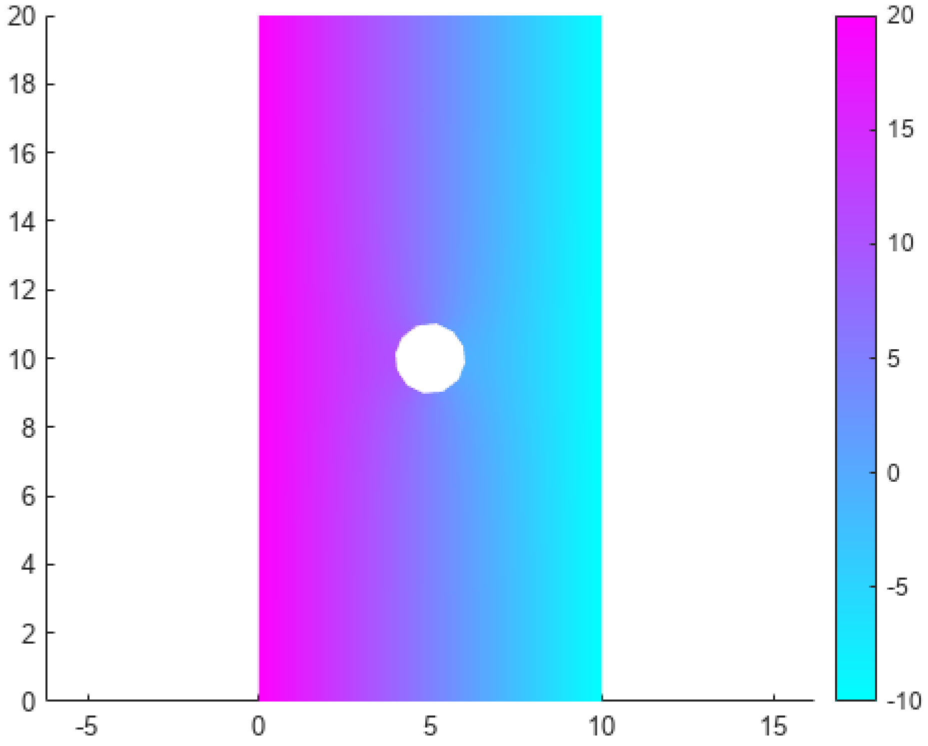
```
thermalBC(thermalmodel,"Edge",4,"Temperature",20);
thermalBC(thermalmodel,"Edge",1,"Temperature",-10);
```

Generate a mesh and solve the problem. For this example, use a linear mesh to better see the nodes on the mesh plots. Additional nodes on a quadratic mesh make it difficult to see the plots in this example clearly.

```
mesh = generateMesh(thermalmodel, ...
                    "GeometricOrder","linear");
thermalresults = solve(thermalmodel);
```

The solver finds the temperatures and temperature gradients at all nodal locations. Plot the temperatures.

```
pdeplot(thermalmodel,"XYData",thermalresults.Temperature)
axis equal
```

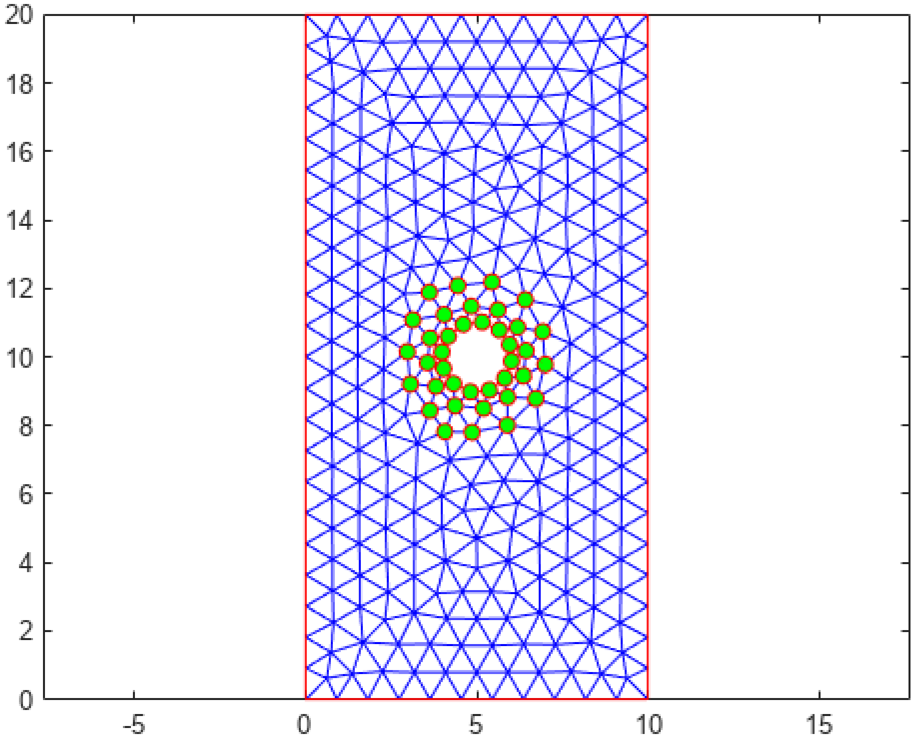


Suppose you need to analyze the results around the center hole more closely. First, find the nodes and elements located next to the hole by using the `findNodes` and `findElements` functions. For example, find nodes and elements located within the radius of 2.5 from the center [5 10].

```
Nr = findNodes(mesh,"radius",[5 10],2.5);
Er = findElements(mesh,"radius",[5 10],2.5);
```

Highlight the nodes within this radius on the mesh plot using a green marker.

```
figure
pdemesh(thermalmodel)
hold on
plot(mesh.Nodes(1,Nr),mesh.Nodes(2,Nr), ...
     "or","MarkerFaceColor","g")
```



Find the minimal and maximal temperatures within the specified radius.

```
[Temps_disk] = thermalresults.Temperature(Nr);
[T_min,index_min] = min(Temps_disk);
[T_max,index_max] = max(Temps_disk);
T_min
```

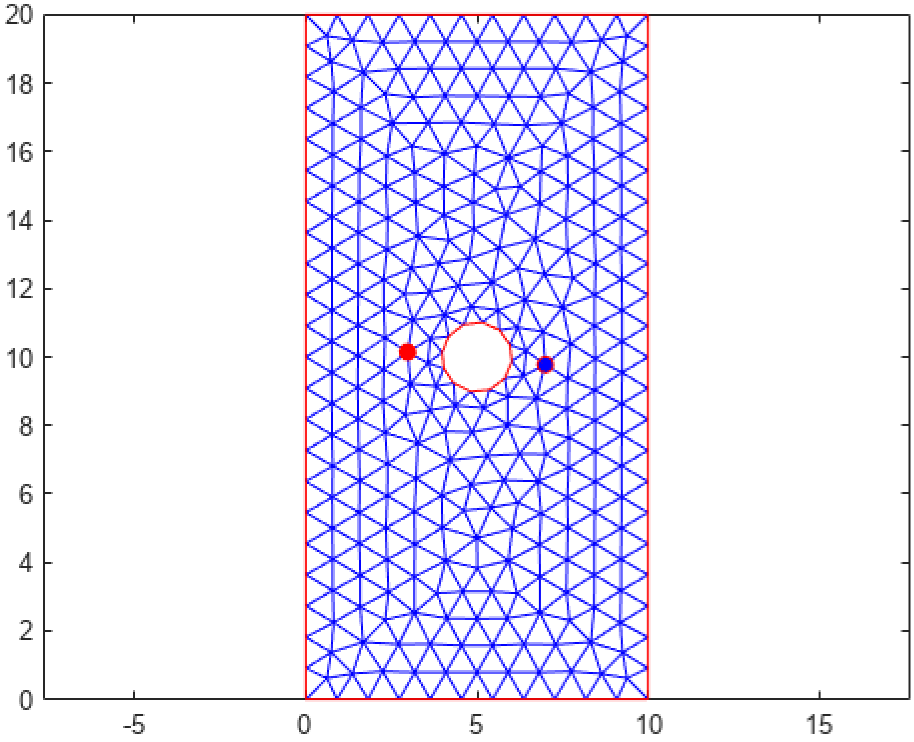
```
T_min = -2.1698
```

```
T_max
```

```
T_max = 12.2420
```

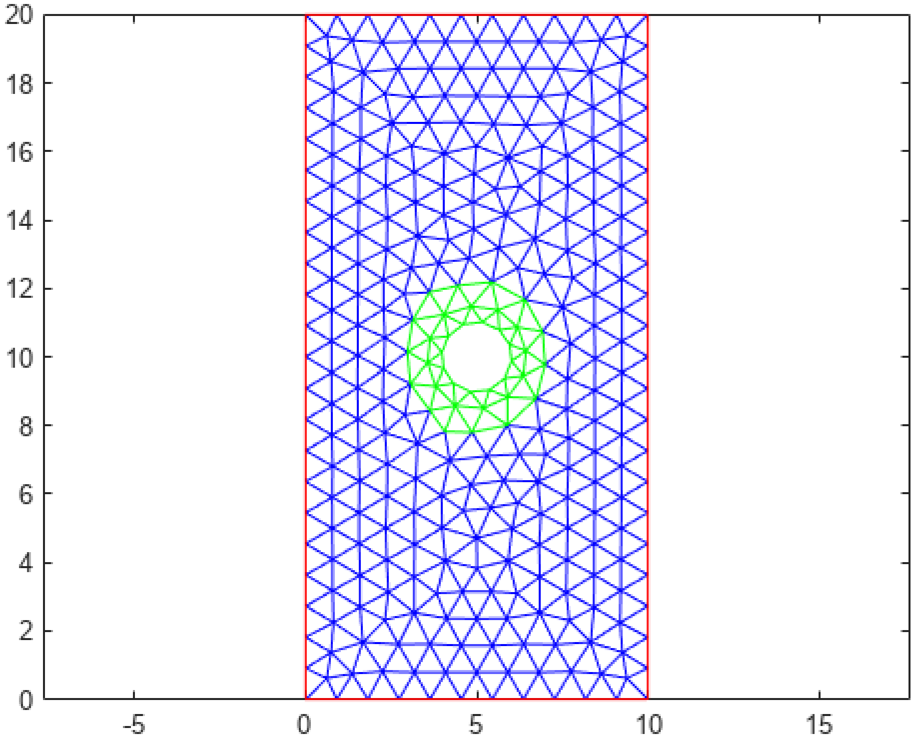
Find the IDs of the nodes corresponding to the minimal and maximal temperatures. Plot these nodes on the mesh plot.

```
nodeIDmin = Nr(index_min);
nodeIDmax = Nr(index_max);

figure
pdemesh(thermalmodel)
hold on
plot(mesh.Nodes(1,nodeIDmin), ...
     mesh.Nodes(2,nodeIDmin), ...
     "or","MarkerFaceColor","b")
plot(mesh.Nodes(1,nodeIDmax), ...
     mesh.Nodes(2,nodeIDmax), ...
     "or","MarkerFaceColor","r")
```



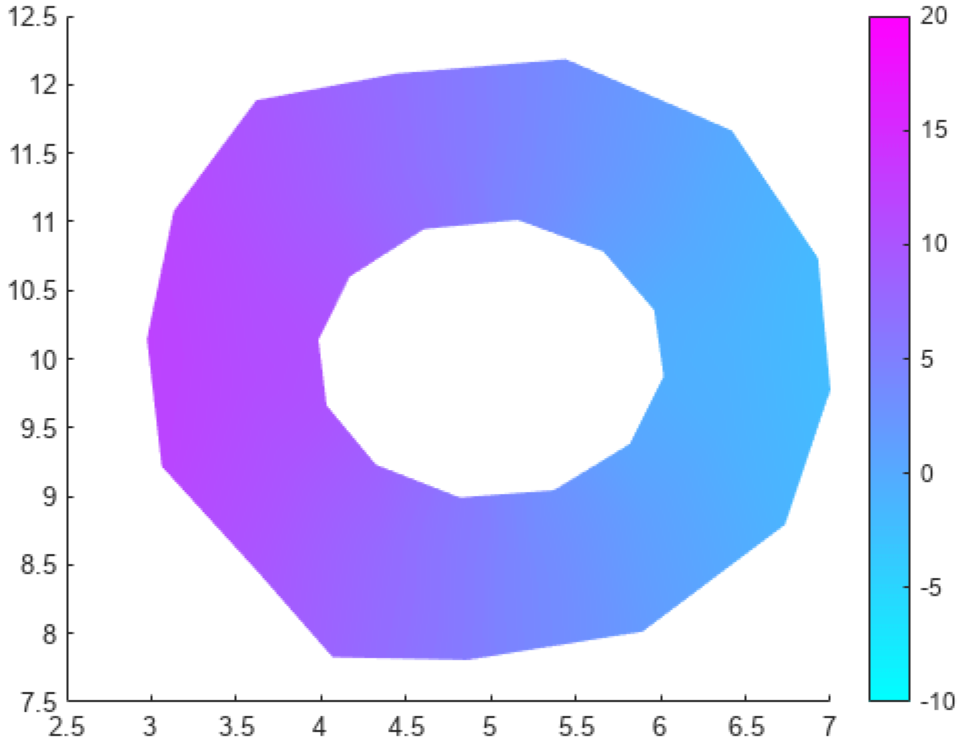
Now highlight the elements within the specified radius on the mesh plot using a green marker.

```
figure
pdemesh(thermalmodel)
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,Er), ...
        "EdgeColor","green")
```



Show the solution for only these elements.

```
figure
pdeplot(mesh.Nodes,mesh.Elements(:,Er), ...
        "XYData",thermalresults.Temperature)
```


12.5
12
11.5
11
10.5
10
9.5
9
8.5
8
7.5
2.5
3
3.5
4
4.5
5
5.5
6
6.5
7
20
15
10
5
0
-5
-10

Assess Quality of Mesh Elements

Partial Differential Equation Toolbox™ uses the finite element method to solve PDE problems. This method discretizes a geometric domain into a collection of simple shapes that make up a mesh. The quality of the mesh is crucial for obtaining an accurate approximation of a solution.

Typically, PDE solvers work best with meshes made up of elements that have an equilateral shape. Such meshes are ideal. In reality, creating an ideal mesh for most 2-D and 3-D geometries is impossible because geometries have tiny or narrow regions and sharp angles. For such regions, a mesh generator creates meshes with some elements that are much smaller than the rest of mesh elements or have drastically different side lengths.

As mesh elements become distorted, numeric approximations of a solution typically become less accurate. Refining a mesh using smaller elements produces better shaped elements and, therefore, more accurate results. However, it also can be computationally expensive.

Checking if the mesh is of good quality before running an analysis is a good practice, especially for simulations that take a long time. The toolbox provides the `meshQuality` function for this task.

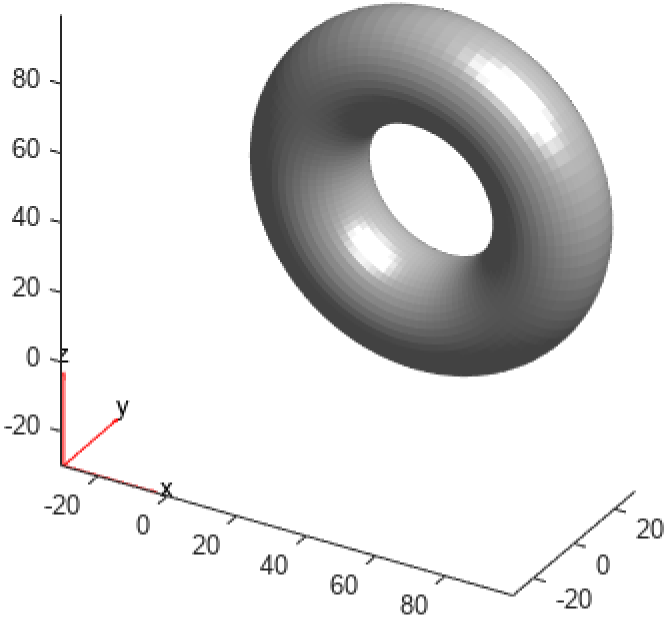
`meshQuality` evaluates the shape quality of mesh elements and returns numbers from 0 to 1 for each mesh element. The value 1 corresponds to the optimal shape of the element. By default, the `meshQuality` function combines several criteria when evaluating the shape quality. In addition to the default metric, you can use the `aspect-ratio` metric, which is based solely on the ratio of the minimum dimension of an element to its maximum dimension.

Create a PDE model.

```
model = createpde;
```

Include and plot the torus geometry.

```
importGeometry(model,"Torus.stl");
pdegplot(model)
camlight right
```



Generate a coarse mesh.

```
mesh = generateMesh(model,"Hmax",10);
```

Evaluate the shape quality of all mesh elements.

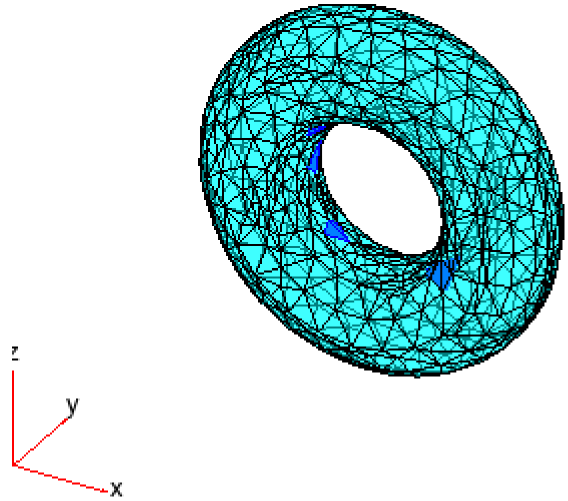
```
Q = meshQuality(mesh);
```

Find the elements with quality values less than 0.3.

```
elemIDs = find(Q < 0.3);
```

Highlight these elements in blue on the mesh plot.

```
figure
pdemesh(mesh,"FaceAlpha",0.5)
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,elemIDs), ...
        "FaceColor","blue","EdgeColor","blue")
```



Determine how much of the total mesh volume belongs to elements with quality values less than 0.3. Return the result as a percentage.

```
mv03_percent = volume(mesh,elemIDs)/volume(mesh)*100
```

```
mv03_percent = 0.0198
```

Evaluate the shape quality of the mesh elements by using the ratio of minimal to maximal dimension for each element.

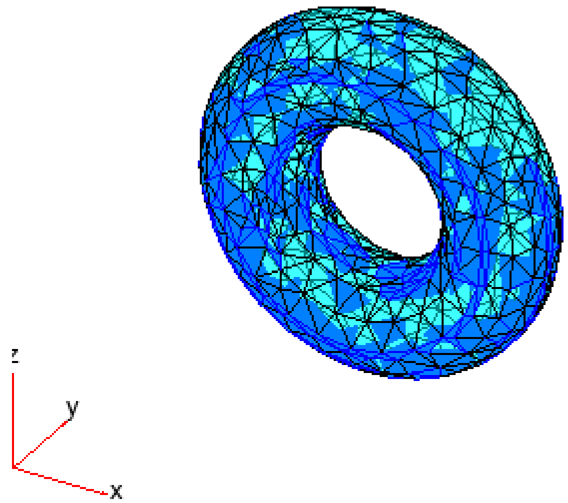
```
Q = meshQuality(mesh,"aspect-ratio");
```

Find the elements with quality values less than 0.3.

```
elemIDs = find(Q < 0.3);
```

Highlight these elements in blue on the mesh plot.

```
figure
pdemesh(mesh,"FaceAlpha",0.5)
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,elemIDs), ...
        "FaceColor","blue","EdgeColor","blue")
```


z
y
x

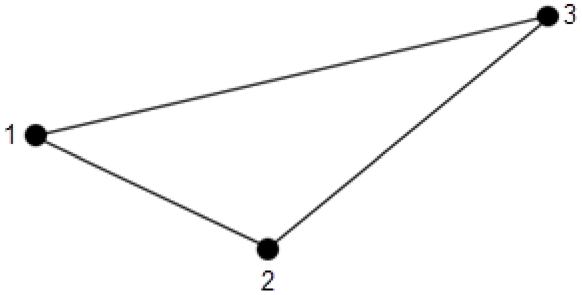
Mesh Data as [p,e,t] Triples

Partial Differential Equation Toolbox uses meshes with triangular elements for 2-D geometries and meshes with tetrahedral elements for 3-D geometries. Earlier versions of Partial Differential Equation Toolbox use meshes in the form of a `[p,e,t]` triple. The matrices `p`, `e`, and `t` represent the points (nodes), elements, and triangles or tetrahedra of a mesh, respectively. Later versions of the toolbox support the `[p,e,t]` meshes for compatibility reasons.

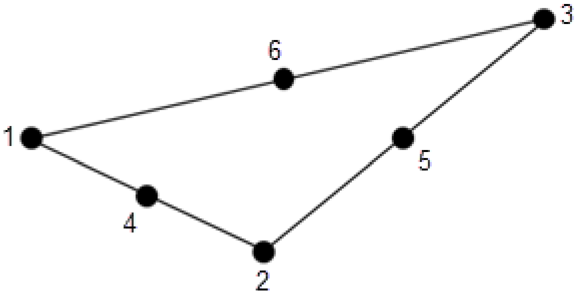
Note New features might not be compatible with the legacy workflow. For description of the mesh data in the recommended workflow, see “Mesh Data” on page 2-175.

The mesh data for a 2-D mesh has these components:

- `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. The mesh edges in `e` and the edges of the geometry have a one-to-one correspondence. The `e` matrix represents the discrete edges of the geometry in the same manner as the `t` matrix represents the discrete faces. Each column in the `e` matrix represents one edge.
  - `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 the 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 by using the command `pdegplot(geom,'EdgeLabels','on')`.
  - `e(6,k)` is the subdomain number on the left side of the edge. The direction along the edge is given by increasing parameter values. The subdomain `0` is the exterior of the geometry.
  - `e(7,k)` is the subdomain number on the right side of the edge.
- `t` (triangles) is a `4`-by-`Nt` matrix of triangles or a `7`-by-`Nt` matrix of triangles, depending on whether you call `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.



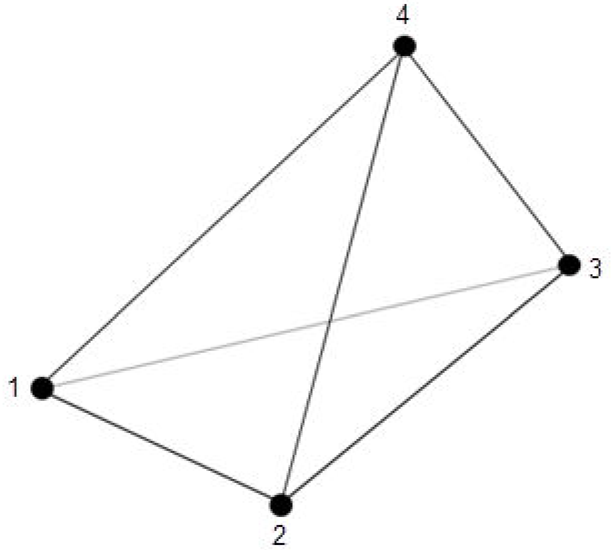
2-D linear element
showing node numbering



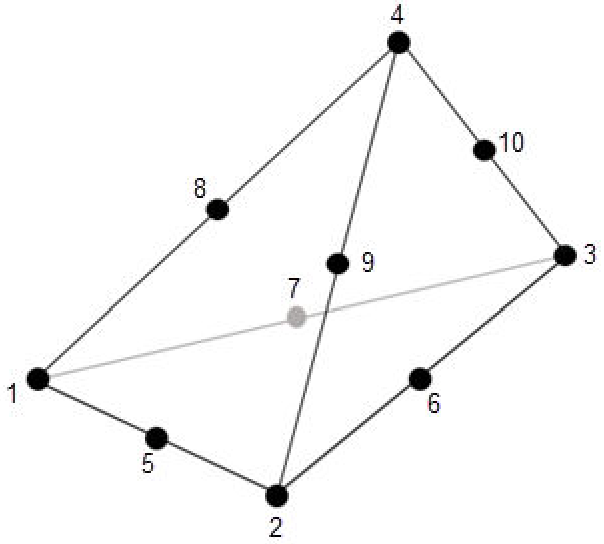
2-D quadratic element
showing node numbering

The mesh data for a 3-D mesh has these components:

- `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 with the geometry boundaries. 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 call `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. Tetrahedron points are ordered as shown.



3-D linear element
showing node numbering



3-D quadratic element
showing node numbering

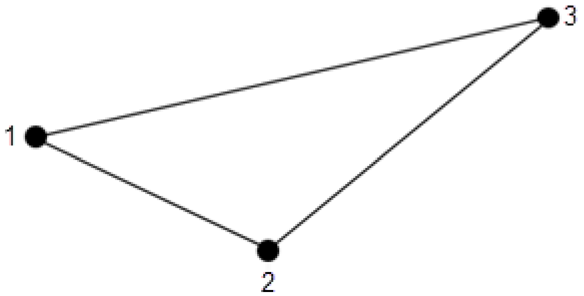
You can create a `[p,e,t]` mesh by using one of these approaches:

- Use the `initmesh` function to create a 2-D `[p,e,t]` mesh.
- Use the `generateMesh` function to create a 2-D or 3-D mesh as a `FEMesh` object. Then use the `meshToPet` function to convert the mesh to a `[p,e,t]` mesh.

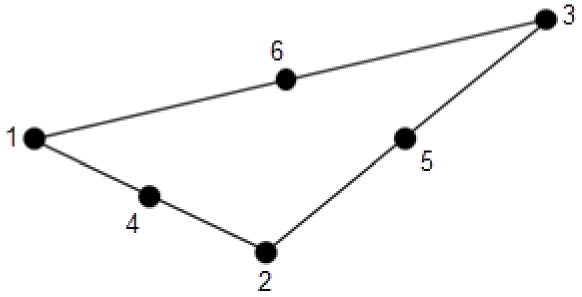
Mesh Data

Partial Differential Equation Toolbox uses meshes with triangular elements for 2-D geometries and meshes with tetrahedral elements for 3-D geometries. In both cases, it uses the quadratic geometric order by default, and provides the option to switch to the linear geometric order. A mesh always consists of elements of the same order. The toolbox does not support mixed meshes.

triangular elements in 2-D meshes are specified by three nodes for linear elements or six nodes for quadratic elements. A triangle representing a linear element has nodes at the corners. A triangle representing a quadratic element has nodes at its corners and edge centers.

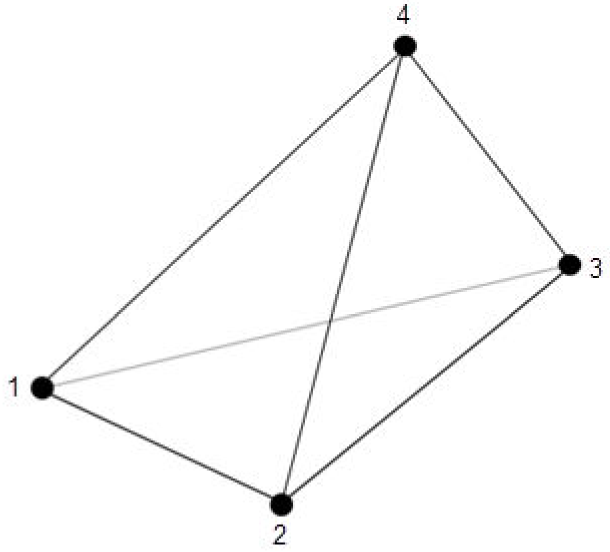


2-D linear element
showing node numbering

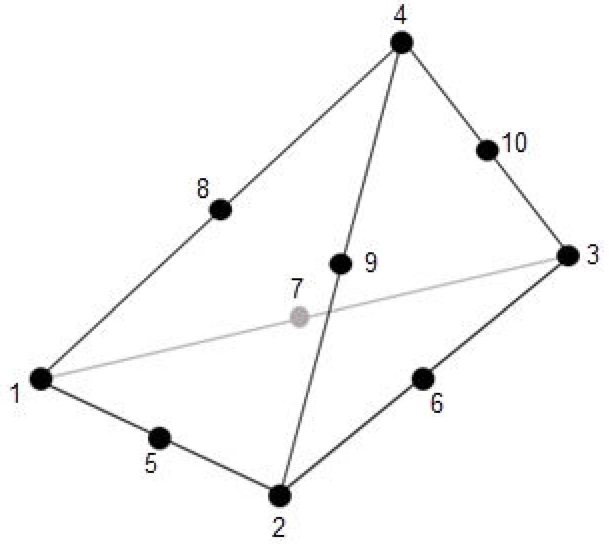


2-D quadratic element
showing node numbering

Tetrahedral elements in 3-D meshes are specified by four nodes for linear elements or 10 nodes for quadratic elements. A tetrahedron representing a linear element has nodes at the corners. A tetrahedron representing a quadratic element has nodes at its corners and edge centers.



3-D linear element
showing node numbering



3-D quadratic element
showing node numbering

The center nodes in quadratic meshes are always added at half-distance between corners. For geometries with curved surfaces and edges, center nodes might not appear on the edge or surface itself.

The model container object stores the parameters of the PDE model. The toolbox offers several types of model container objects, each for a particular application area. For example, for linear elasticity problems, the model container is a `StructuralModel` object, and for heat transfer problems, the model container is a `ThermalModel` object. For general PDE problems, the toolbox uses the `PDEModel` object.

The `Mesh` property of the model container object stores mesh data. The `Mesh` property contains a `FEMesh` object. FEMesh include information on the nodes and elements of the mesh, mesh growth rate, and target minimum and maximum element size. The properties also indicate whether the mesh is linear or quadratic. You can specify these mesh parameters when creating a mesh.

To generate a mesh for your PDE model, use the `generateMesh` function.

By default, `generateMesh` uses the quadratic geometric order, which typically produces more accurate results than the linear geometric order. To switch to the linear geometric order, call the mesh generator and set the `GeometricOrder` name-value pair to `'linear'`.

Solving PDEs

von Mises Effective Stress and Displacements: PDE Modeler App

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 Modeler 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:

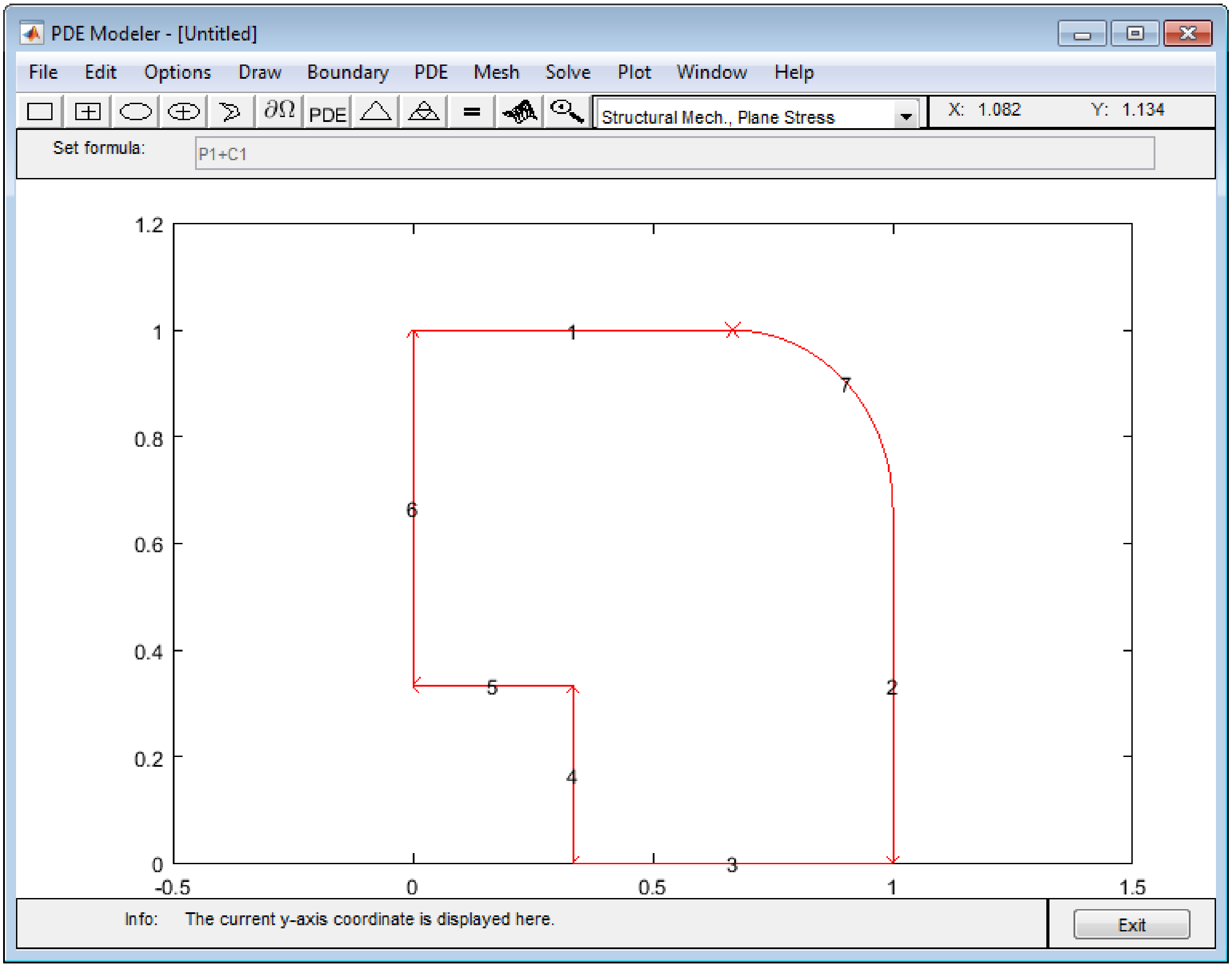
- Dimensions 1 m-by-1 m-by 0.001 m;
- Inset is 1/3-by-1/3 m
- The rounded cut runs from (2/3, 1) to (1, 2/3)
- Young's modulus: $196 \cdot 10^3$ (MN/m^2)
- Poisson's ratio: 0.31.

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 Modeler 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**.

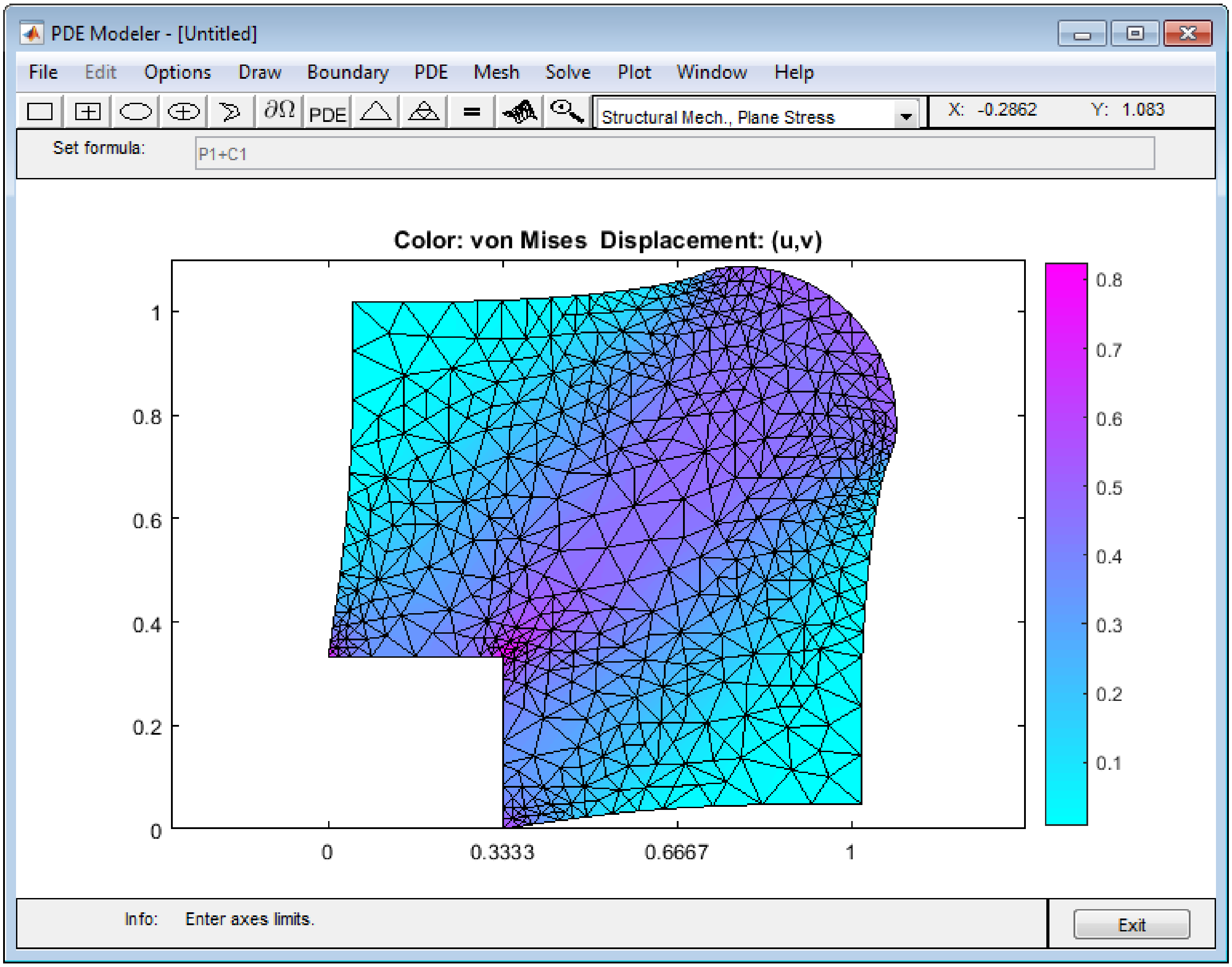
10 Refining the mesh in areas where the gradient of the solution (the stress) is large. To do this, select **Solve** > **Parameters**. In the resulting dialog box, select **Adaptive mode**. Use the default adaptation options: the **Worst triangles** triangle selection method with the **Worst triangle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



Set formula: P1+C1

Info: Enter axes limits.

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.

Clamped Square Isotropic Plate with Uniform Pressure Load

This example shows how to calculate the deflection of a structural plate under a pressure loading.

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

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

where D is the bending stiffness of the plate given by

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

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

Partial Differential Equation Toolbox™ cannot directly solve this fourth-order plate equation. Convert the fourth-order equation to these two second-order partial differential equations, where v is the new dependent variable.

$$\nabla^2 w = v$$

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

You cannot directly specify boundary conditions for both w and w' in this second-order system. Instead, specify that w' is 0, and define v' so that w also equals 0 on the boundary. To specify these conditions, use stiff "springs" distributed along the boundary. The springs apply a transverse shear force to the plate edge. Define the shear force along the boundary due to these springs as $n \cdot D\nabla v = -kw$, where n is the normal to the boundary, and k is the stiffness of the springs. This expression is a generalized Neumann boundary condition supported by the toolbox. The value of k must be large enough so that w is approximately 0 at all points on the boundary. It also must be small enough to avoid numerical errors due to an ill-conditioned stiffness matrix.

The toolbox uses the dependent variables u_1 and u_2 instead of w and v. Rewrite the two second-order partial differential equations using variables u_1 and u_2:

$$-\nabla^2 u_1 + u_2 = 0$$

$$-D\nabla^2 u_2 = p$$

Create a PDE model for a system of two equations.

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

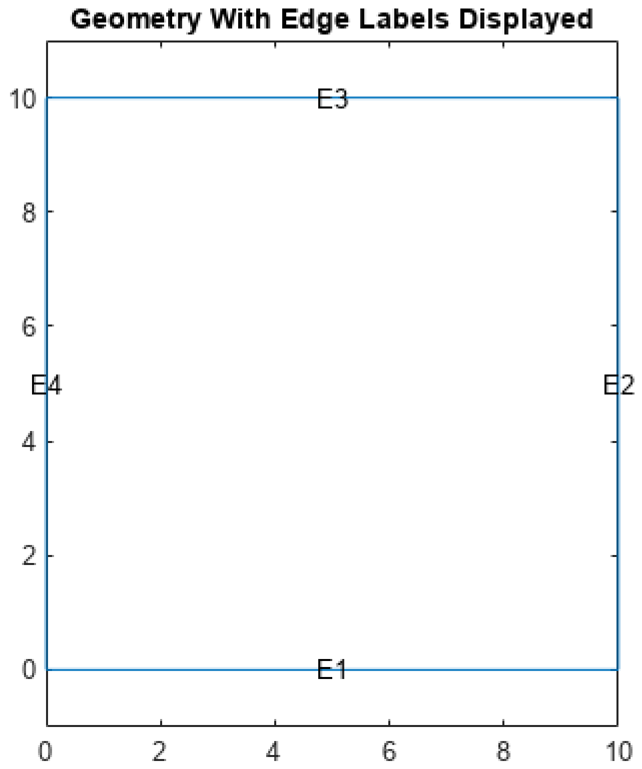
Create a square geometry and include it in the model.

```
len = 10;
gdm = [3 4 0 len len 0 0 0 len len]';
```

```
g = decsg(gdm,'S1',('S1')');
geometryFromEdges(model,g);
```

Plot the geometry with the edge labels.

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



PDE coefficients must be specified using the format required by the toolbox. For details, see

- “c Coefficient for specifyCoefficients” on page 2-92
- “m, d, or a Coefficient for specifyCoefficients” on page 2-107
- “f Coefficient for specifyCoefficients” on page 2-90

The c coefficient in this example is a tensor, which can be represented as a 2-by-2 matrix of 2-by-2 blocks:

$$\left[\begin{array}{cc|cc} c(1) & c(2) & \cdot & \cdot \\ \cdot & c(3) & \cdot & \cdot \\ \hline \cdot & \cdot & c(4) & c(5) \\ \cdot & \cdot & \cdot & c(6) \end{array}\right]$$

This matrix is further flattened into a column vector of six elements. The entries in the full 2-by-2 matrix (defining the coefficient a) and the 2-by-1 vector (defining the coefficient f) follow directly from the definition of the two-equation system.

```
E = 1.0e6; % Modulus of elasticity
nu = 0.3; % Poisson's ratio
thick = 0.1; % Plate thickness
pres = 2; % External pressure

D = E*thick^3/(12*(1 - nu^2));

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);
```

To define boundary conditions, first specify spring stiffness.

```
k = 1e7;
```

Define distributed springs on all four edges.

```
bOuter = applyBoundaryCondition(model,"neumann","Edge",(1:4),...
                                     "g",[0 0],"q",[0 0; k 0]);
```

Generate a mesh.

```
generateMesh(model);
```

Solve the model.

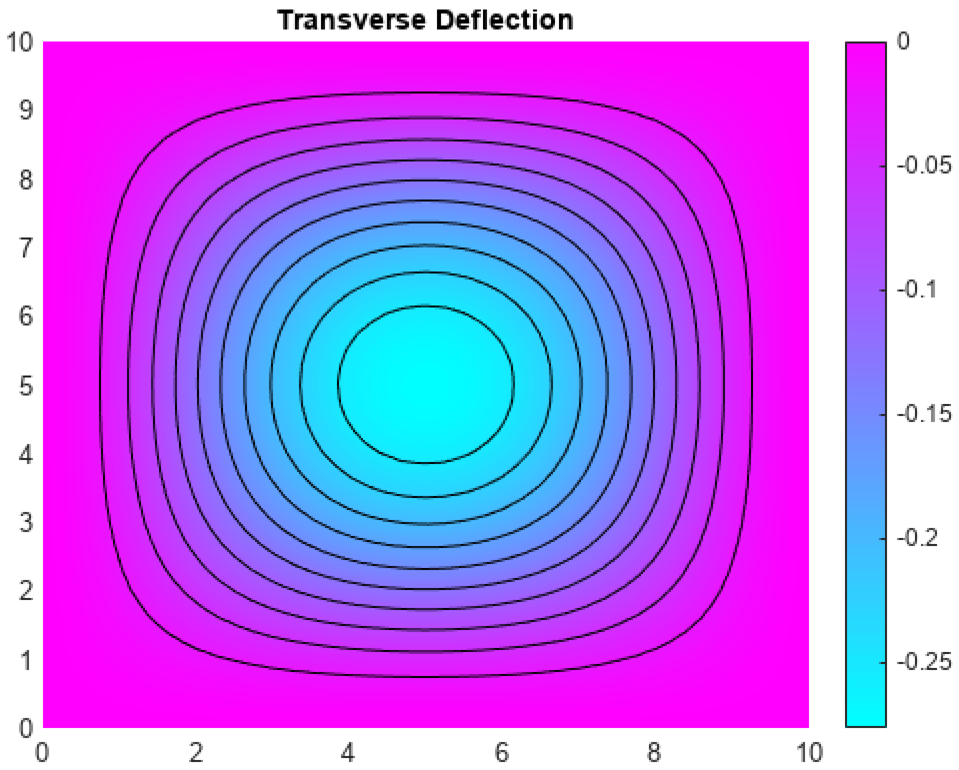
```
res = solvepde(model);
```

Access the solution at the nodal locations.

```
u = res.NodalSolution;
```

Plot the transverse deflection.

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



Find the transverse deflection at the plate center.

```
numNodes = size(model.Mesh.Nodes,2);
wMax = min(u(1:numNodes,1))
```

```
wMax = -0.2763
```

Compare the result with the deflection at the plate center computed analytically.

```
wMax = -.0138*pres*len^4/(E*thick^3)
```

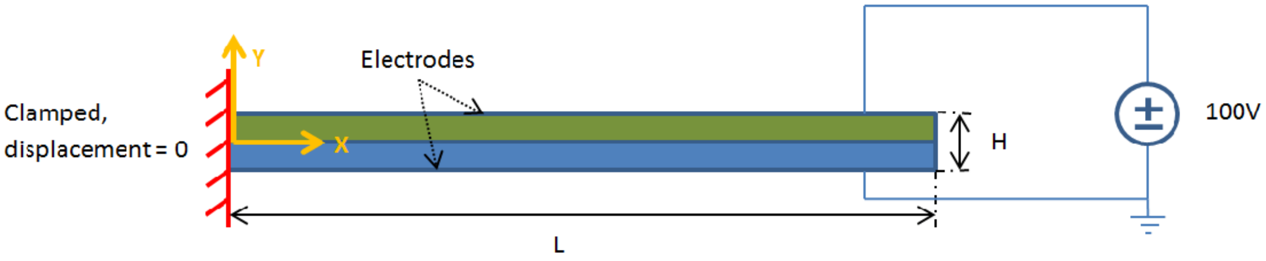
```
wMax = -0.2760
```

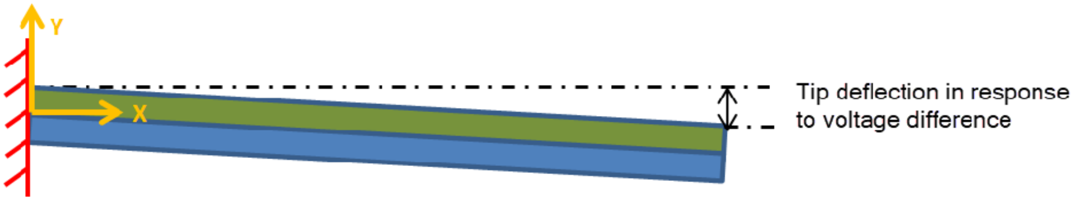
Deflection of Piezoelectric Actuator

This example shows how to solve a coupled elasticity-electrostatics problem.

Piezoelectric materials deform under an applied voltage. Conversely, deforming a piezoelectric material produces a voltage. Therefore, analysis of a piezoelectric part requires the solution of a set of coupled partial differential equations with deflections and electrical potential as dependent variables.

In this example, the model is a two-layer cantilever beam, with both layers made of the same polyvinylidene fluoride (PVDF) material. The polarization direction points down (negative *y*-direction) in the top layer and points up in the bottom layer. The typical length to thickness ratio is 100. When you apply a voltage between the lower and upper surfaces of the beam, the beam deflects in the *y*-direction because one layer shortens and the other layer lengthens.





The equilibrium equations describe the elastic behavior of the solid:

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

Here, σ is the stress tensor, and f is the body force vector. Gauss's Law describes the electrostatic behavior of the solid:

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

D is the electric displacement, and ρ is the distributed free charge. Combine these two PDE systems into this single system:

$$-\nabla \cdot \begin{Bmatrix} \sigma \\ D \end{Bmatrix} = \begin{Bmatrix} f \\ -\rho \end{Bmatrix}$$

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

The constitutive equations for the material define the stress tensor and electric displacement vector in terms of the strain tensor and electric field. For a 2-D analysis of an orthotropic piezoelectric material under plane stress conditions, you can write these equations as

$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \\ D_1 \\ D_2 \end{Bmatrix} = \begin{bmatrix} C_{11} & C_{12} & & e_{11} & e_{31} \\ C_{12} & C_{22} & & e_{13} & e_{33} \\ & & G_{12} & e_{14} & e_{34} \\ e_{11} & e_{13} & e_{14} & -\mathscr{E}_1 & \\ e_{31} & e_{33} & e_{34} & & -\mathscr{E}_2 \end{bmatrix} \begin{Bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \gamma_{12} \\ -E_1 \\ -E_2 \end{Bmatrix}$$

C_{ij} are the elastic coefficients, $\mathscr{E}_i$ are the electrical permittivities, and e_{ij} are the piezoelectric stress coefficients. The piezoelectric stress coefficients in these equations conform to conventional notation in piezoelectric materials where the z-direction (the third direction) aligns with the "poled" direction of the material. For the 2-D analysis, align the "poled" direction with the y-axis. Write the strain vector in terms of the x-displacement u and y-displacement v:

$$\begin{Bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \gamma_{12} \end{Bmatrix} = \begin{Bmatrix} \dfrac{\partial u}{\partial x} \\ \dfrac{\partial v}{\partial y} \\ \dfrac{\partial u}{\partial y} + \dfrac{\partial v}{\partial x} \end{Bmatrix}$$

Write the electric field in terms of the electrical potential ϕ:

$$\begin{Bmatrix} E_1 \\ E_2 \end{Bmatrix} = -\begin{Bmatrix} \dfrac{\partial \phi}{\partial x} \\ \dfrac{\partial \phi}{\partial y} \end{Bmatrix}$$

You can substitute the strain-displacement equations and electric field equations into the constitutive equations and get a system of equations for the stresses and electrical displacements in terms of displacement and electrical potential derivatives. Substituting the resulting equations into the PDE system equations yields a system of equations that involve the divergence of the displacement and electrical potential derivatives. As the next step, arrange these equations to match the form required by the toolbox.

Partial Differential Equation Toolbox™ requires a system of elliptic equations to be expressed in a vector form:

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

or in a 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 repeated indices imply summation. For the 2-D piezoelectric system in this example, the system vector $\mathbf{u}$ is

$$\mathbf{u} = \begin{Bmatrix} u \\ v \\ \phi \end{Bmatrix}$$

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

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

For details on specifying the coefficients in the format required by the toolbox, see:

- “c Coefficient for specifyCoefficients” on page 2-92
- “m, d, or a Coefficient for specifyCoefficients” on page 2-107
- “f Coefficient for specifyCoefficients” on page 2-90

The c coefficient in this example is a tensor. You can represent it as a 3-by-3 matrix of 2-by-2 blocks:

$$\left[\begin{array}{cc|cc|cc} c(1) & c(3) & c(13) & c(15) & c(25) & c(27) \\ c(2) & c(4) & c(14) & c(16) & c(26) & c(28) \\ \hline \mathrm{c}\ (5) & c(7) & c(17) & c(19) & c(29) & c(31) \\ c(6) & c(8) & \mathrm{c}\ (18) & c(20) & c(30) & c(32) \\ \hline \mathrm{c}\ (9) & c(11) & \mathrm{c}\ (21) & c(23) & c(33) & c(35) \\ \mathrm{c}\ (10) & c(12) & \mathrm{c}\ (22) & c(24) & c(34) & c(36) \end{array} \right]$$

To map terms of constitutive equations to the form required by the toolbox, write the c tensor and the solution gradient in this form:

$$\left[\begin{array}{cc|cc|cc} c_{1111} & c_{1112} & c_{1211} & c_{1212} & c_{1311} & c_{1312} \\ c_{1121} & c_{1122} & c_{1221} & c_{1222} & c_{1321} & c_{1322} \\ \hline c_{2111} & c_{2112} & c_{2211} & c_{2212} & c_{2311} & c_{2312} \\ c_{2121} & c_{2122} & c_{2221} & c_{2222} & c_{2321} & c_{2322} \\ \hline c_{3111} & c_{3112} & c_{3211} & c_{3212} & c_{3311} & c_{3312} \\ c_{3121} & c_{3122} & c_{3221} & c_{3222} & c_{3321} & c_{3322} \end{array} \right] \left\{ \begin{array}{c} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \\ \hline \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial y} \\ \hline \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{array} \right\}$$

From this equation, you can map the traditional constitutive coefficients to the form required for the c matrix. The minus sign in the equations for the electric field is incorporated into the c matrix to match the toolbox's convention.

$$\left[\begin{array}{cc|cc|cc} C_{11} & \cdot & \cdot & C_{12} & e_{11} & e_{31} \\ \cdot & G_{12} & G_{12} & \cdot & e_{14} & e_{34} \\ \hline \cdot & G_{12} & G_{12} & \cdot & e_{14} & e_{34} \\ C_{21} & \cdot & \cdot & C_{22} & e_{13} & e_{33} \\ \hline e_{11} & e_{14} & e_{14} & e_{13} & -\mathscr{E}_1 & \cdot \\ e_{31} & e_{34} & e_{34} & e_{33} & \cdot & -\mathscr{E}_2 \end{array}\right] \left\{\begin{array}{c} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \\ \hline \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial y} \\ \hline \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{array}\right\}$$

Beam Geometry

Create a PDE model. The equations have three components: two components due to linear elasticity and one component due to electrostatics. Therefore, the model must have three equations.

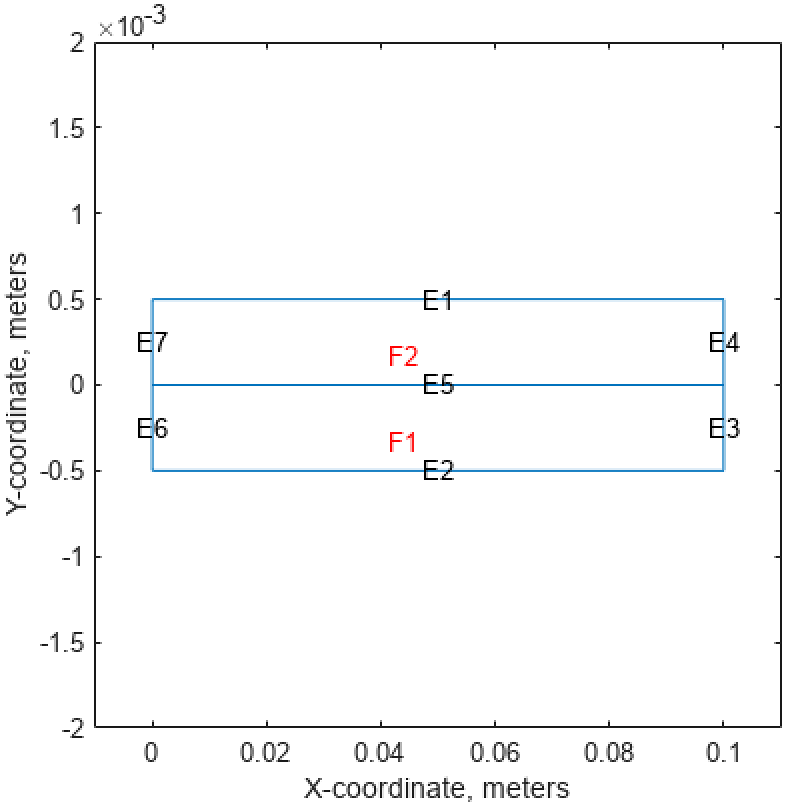
```
model = createpde(3);
```

Create the geometry and include it in the model.

```
L = 100e-3; % Beam length in meters
H = 1e-3; % Overall height of the beam
H2 = H/2; % Height of each layer in meters

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']');

geometryFromEdges(model,g);
```

Plot the geometry with the face and edge labels.

```
figure
pdegplot(model,"EdgeLabels","on", ...
               "FaceLabels","on")
xlabel("X-coordinate, meters")
ylabel("Y-coordinate, meters")
axis([-.1*L,1.1*L,-4*H2,4*H2])
axis square
```



Material Properties

Specify the material properties of the beam layers. The material in both layers 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;
```

Specify relative electrical permittivity of the material at constant stress.

```
relPermittivity = 12;
```

Specify the 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];
```

Specify the piezoelectric stress coefficients.

```
pzeE = c2d*pzeD;
D_const_stress = [relPermittivity 0;
                  0 relPermittivity]*permittivityFreeSpace;
```

Convert the dielectric matrix from constant stress to constant strain.

```
D_const_strain = D_const_stress - pzeD'*pzeE;
```

The parameters of the elastic equations are of the order of 10^9 while the electric parameters are of the order of 10^{-11}. To avoid constructing an ill-conditioned matrix, rescale the last equation so that the coefficients are larger. Note that before any post-processing involving the c coefficient (for example, when you evaluate a flux of PDE solution), you must revert the scaling changes to the c matrix.

```
cond_scaling = 1e5;
```

You can view the 36 coefficients as a 3-by-3 matrix of 2-by-2 blocks.

```
c11 = [c2d(1,1) c2d(1,3) c2d(3,1) c2d(3,3)];
c12 = [c2d(1,3) c2d(1,2); c2d(3,3) c2d(2,3)];
c21 = c12';

c22 = [c2d(3,3) c2d(2,3) c2d(3,2) c2d(2,2)];
c13 = [pzeE(1,1) pzeE(1,2); pzeE(3,1) pzeE(3,2)];
c31 = cond_scaling*c13';
c23 = [pzeE(3,1) pzeE(3,2); pzeE(2,1) pzeE(2,2)];
c32 = cond_scaling*c23';

c33 = cond_scaling*[D_const_strain(1,1)
                    D_const_strain(2,1)
                    D_const_strain(1,2)
                    D_const_strain(2,2)];
ctop = [c11(:); c21(:); -c31(:);
        c12(:); c22(:); -c32(:);
       -c13(:); -c23(:); -c33(:)];
cbot = [c11(:); c21(:); c31(:);
        c12(:); c22(:); c32(:);
        c13(:); c23(:); -c33(:)];
```

If you problem includes a current source for the third equation, scale the f coefficient: `f = [0 0 cond_scaling*value_f]'`. Otherwise, specify it as follows.

```
f = [0 0 0]';
```

Specify coefficients.

```
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 Conditions

Set the voltage (solution component 3) on the top of the beam (edge 1) to 100 volts.

```
voltTop = applyBoundaryCondition(model,"mixed", ...
                                       "Edge",1,...
                                       "u",100,...
                                       "EquationIndex",3);
```

Specify that the bottom of the beam (edge 2) is grounded by setting the voltage to 0.

```
voltBot = applyBoundaryCondition(model,"mixed", ...
                                       "Edge",2,...
```

```
                                                   "u",0,...
                                                   "EquationIndex",3);
```

Specify that the left side (edges 6 and 7) is clamped by setting the *x*- and *y*-displacements (solution components 1 and 2) to 0.

```
clampLeft = applyBoundaryCondition(model,"mixed", ...
                                         "Edge",6:7,...
                                         "u",[0 0],...
                                         "EquationIndex",1:2);
```

The stress and charge on the right side of the beam are zero. Accordingly, use the default boundary condition for edges 3 and 4.

Finite Element and Analytical Solutions

Generate a mesh and solve the model.

```
msh = generateMesh(model,"Hmax",5e-4);
result = solvepde(model)
```

```
result = 
  StationaryResults with properties:

    NodalSolution: [3605x3 double]
       XGradients: [3605x3 double]
       YGradients: [3605x3 double]
       ZGradients: [0x3 double]
             Mesh: [1x1 FEMesh]
```

Access the solution at the nodal locations. The first column contains the *x*-deflection. The second column contains the *y*-deflection. The third column contains the electrical potential.

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

Find the minimum *y*-deflection.

```
feTipDeflection = min(rs(:,2));
fprintf("Finite element tip deflection is: %12.4e\n",feTipDeflection);
```

```
Finite element tip deflection is:  -3.2900e-05
```

Compare this result with the known analytical solution.

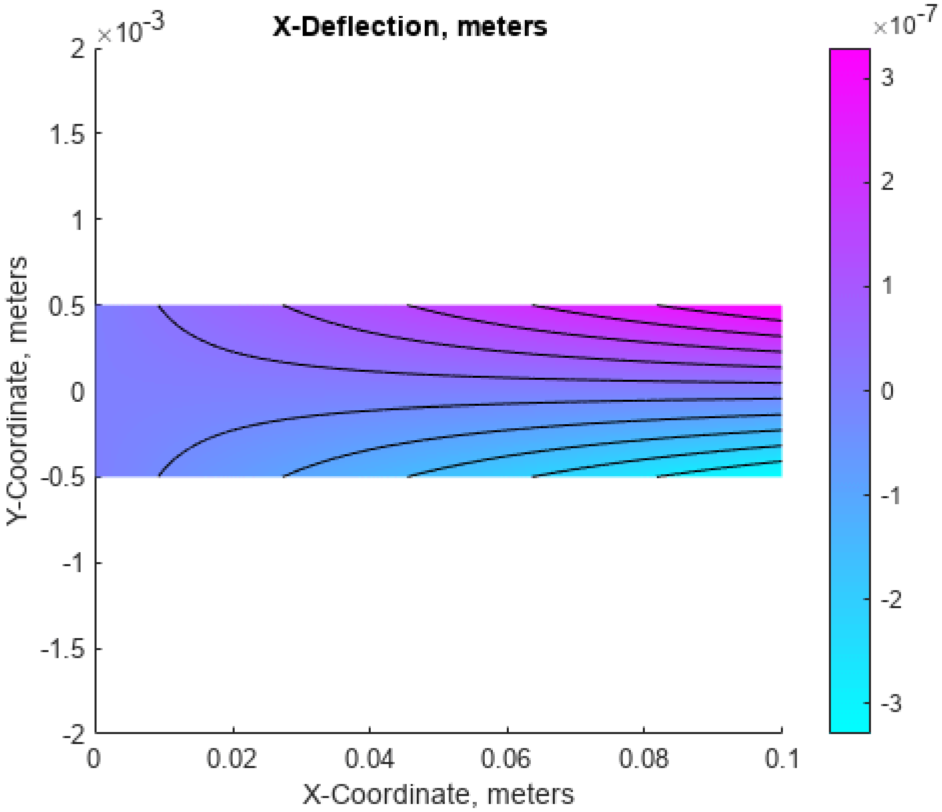
```
tipDeflection = -3*d31*100*L^2/(8*H2^2);
fprintf("Analytical tip deflection is: %12.4e\n",tipDeflection);
```

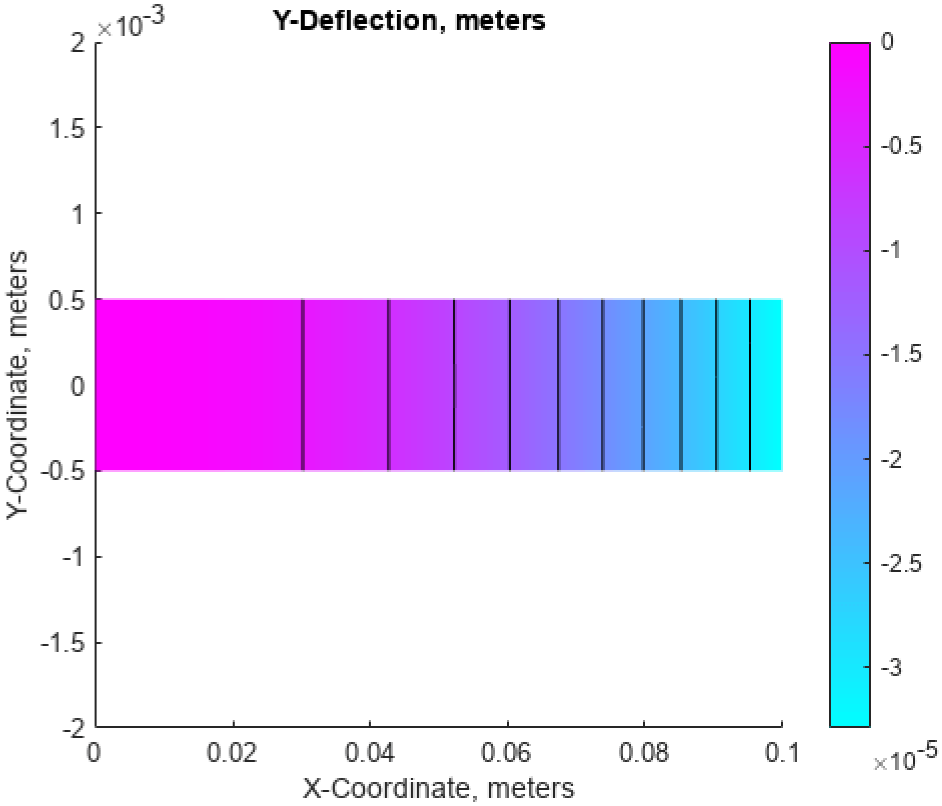
```
Analytical tip deflection is:  -3.3000e-05
```

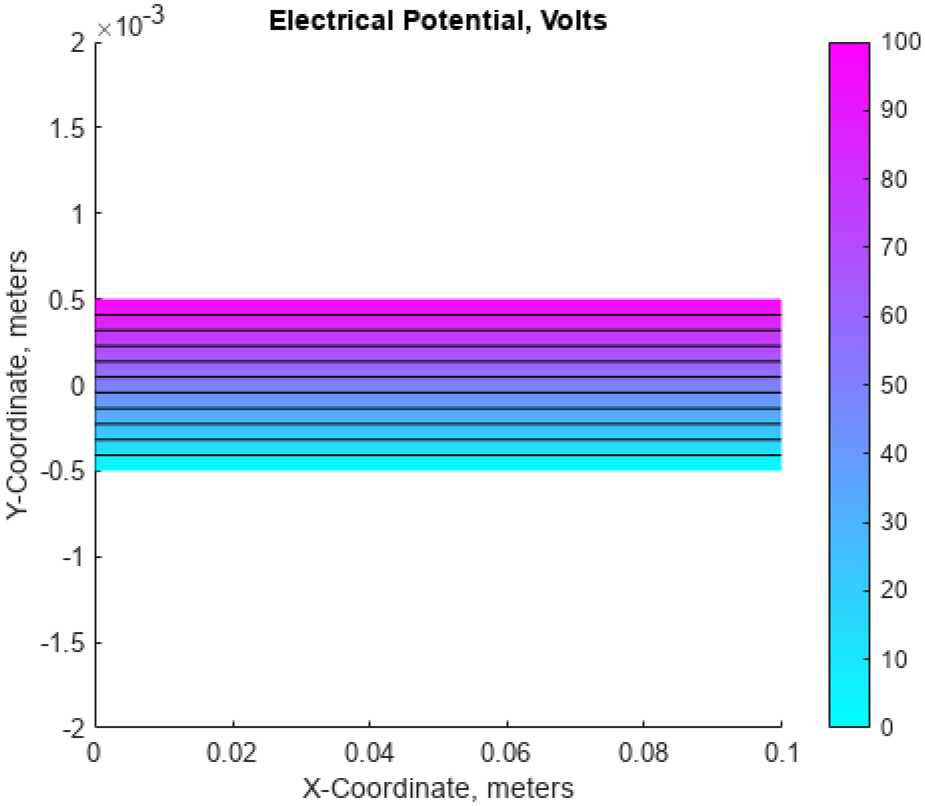
Plot the deflection components and the electrical potential.

```
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")
    axis square
end
```




Y-Deflection, meters
×10^-3
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
Y-Coordinate, meters
0
0.02
0.04
0.06
0.08
0.1
X-Coordinate, meters
0
-0.5
-1
-1.5
-2
-2.5
-3
×10^-5



References

1 Hwang, Woo-Seok, and Hyun Chul Park. "Finite Element Modeling of Piezoelectric Sensors and Actuators." *AIAA Journal* 31, no.5 (May 1993): 930-937.

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

Dynamics of Damped Cantilever Beam

This example shows how to include damping in the transient analysis of a simple cantilever beam.

The damping model is basic viscous damping distributed uniformly through the volume of the beam. The beam is deformed by applying an external load at the tip of the beam and then released at time $t = 0$. This example does not use any additional loading, so the displacement of the beam decreases as a function of time due to the damping. The example uses plane-stress modal, static, and transient analysis models in its three-step workflow:

1. Perform modal analysis to compute the fundamental frequency of the beam and to speed up computations for the transient analysis.
2. Find the static solution of the beam with a vertical load at the tip to use as an initial condition for a transient model.
3. Perform the transient analysis with and without damping.

Damping is typically expressed as a percentage of critical damping, ξ, for a selected vibration frequency. This example uses $\xi = 0.03$, which is three percent of critical damping.

The 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.

Modal Analysis

Create a modal analysis model for a plane-stress problem.

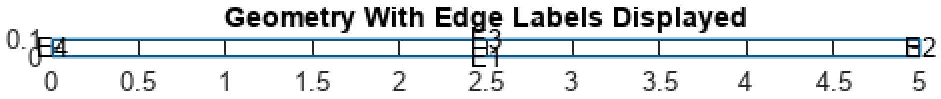
```
modelM = createpde("structural","modal-planestress");
```

Create the geometry and include it in the model. Suppose, the beam is 5 inches long and 0.1 inches thick.

```
width = 5;
height = 0.1;

gdm = [3;4;0;width;width;0;0;0;height;height];
g = decsg(gdm,'S1',('S1')');
geometryFromEdges(modelM,g);
```

Plot the geometry with the edge labels.

```
figure;
pdegplot(modelM,"EdgeLabels","on");
axis equal
title("Geometry With Edge Labels Displayed")
```



Define a maximum element size so that there are five elements through the beam thickness. Generate a mesh.

```
hmax = height/5;
msh = generateMesh(modelM,"Hmax",hmax);
```

Specify Young's modulus, Poisson's ratio, and the mass density of steel.

```
E = 3.0e7;
nu = 0.3;
rho = 0.3/386;
structuralProperties(modelM,"YoungsModulus",E, ...
                            "PoissonsRatio",nu, ...
                            "MassDensity",rho);
```

Specify that the left edge of the beam is a fixed boundary.

```
structuralBC(modelM,"Edge",4,"Constraint","fixed");
```

Solve the problem for the frequency range from `0` to `1e5`. The recommended approach is to use a value that is slightly smaller than the expected lowest frequency. Thus, use `-0.1` instead of `0`.

```
res = solve(modelM,"FrequencyRange",[-0.1,1e5]')
```

```
res = 
  ModalStructuralResults with properties:

    NaturalFrequencies: [8x1 double]
```

```
            ModeShapes: [1x1 FEStruct]
                  Mesh: [1x1 FEMesh]
```

By default, the solver returns circular frequencies.

```
modeID = 1:numel(res.NaturalFrequencies);
```

Express the resulting frequencies in Hz by dividing them by 2π. Display the frequencies in a table.

```
tmodalResults = table(modeID.',res.NaturalFrequencies/(2*pi));
tmodalResults.Properties.VariableNames = {'Mode','Frequency'};
disp(tmodalResults)
```

```
    Mode    Frequency
    ____    _________

     1       126.94  
     2       794.05  
     3       2216.8  
     4       4325.3  
     5       7110.7  
     6       9825.9  
     7        10551  
     8        14623  
```

Compute the analytical fundamental frequency (Hz) using the beam theory.

```
I = height^3/12;
freqAnalytical = 3.516*sqrt(E*I/(width^4*rho*height))/(2*pi)
```

```
freqAnalytical = 126.9498
```

Compare the analytical result with the numerical result.

```
freqNumerical = res.NaturalFrequencies(1)/(2*pi)
```

```
freqNumerical = 126.9416
```

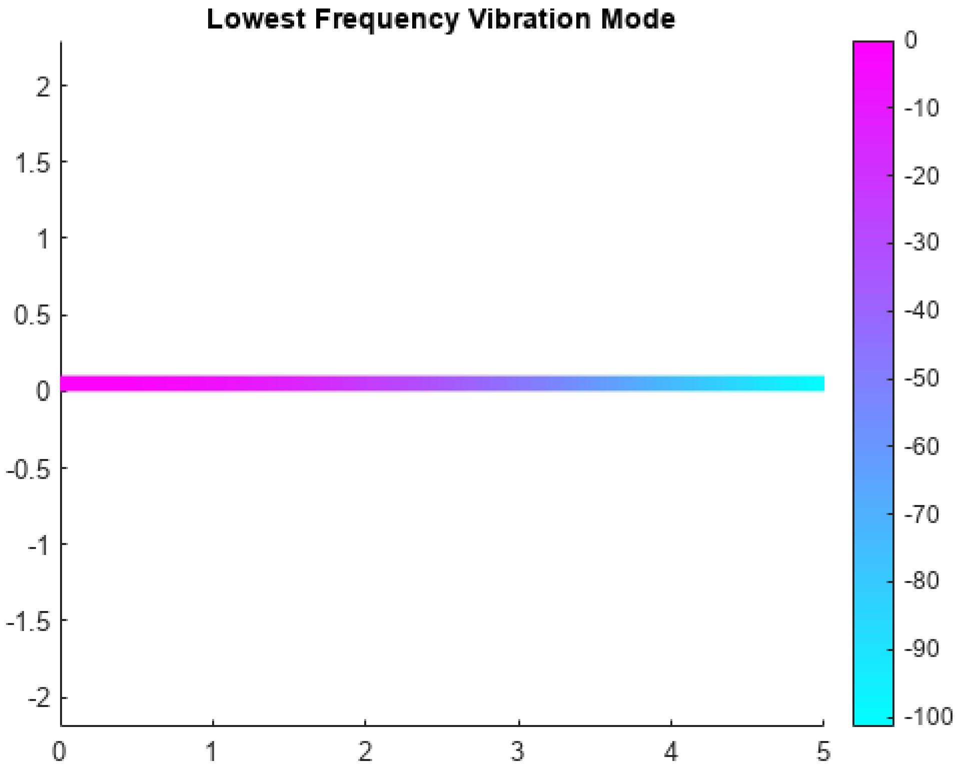
Compute the period corresponding to the lowest vibration mode.

```
longestPeriod = 1/freqNumerical
```

```
longestPeriod = 0.0079
```

Plot the *y*-component of the solution for the lowest beam frequency.

```
figure;
pdeplot(modelM,"XYData",res.ModeShapes.uy(:,1))
title("Lowest Frequency Vibration Mode")
axis equal
```



Initial Displacement from Static Solution

The beam is deformed by applying an external load at its tip and then released at time *t* = 0. Find the initial condition for the transient analysis by using the static solution of the beam with a vertical load at the tip.

Create a static plane-stress model.

```
modelS = createpde("structural","static-planestress");
```

Use the same geometry and mesh that you used for the modal analysis.

```
geometryFromEdges(modelS,g);
modelS.Mesh = msh;
```

Specify the same values for Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelS,"YoungsModulus",E, ...
                            "PoissonsRatio",nu, ...
                            "MassDensity",rho);
```

Specify the same constraint on the left end of the beam.

```
structuralBC(modelS,"Edge",4,"Constraint","fixed");
```

Apply the static vertical load on the right side of the beam.

```
structuralBoundaryLoad(modelS,"Edge",2,"SurfaceTraction",[0;1]);
```

Solve the static model. The resulting static solution serves as an initial condition for transient analysis.

```
Rstatic = solve(modelS);
```

Transient Analysis

Perform the transient analysis of the cantilever beam with and without damping. Use the modal superposition method to speed up computations.

Create a transient plane-stress model.

```
modelT = createpde("structural","transient-planestress");
```

Use the same geometry and mesh that you used for the modal analysis.

```
geometryFromEdges(modelT,g);
modelT.Mesh = msh;
```

Specify the same values for Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelT,"YoungsModulus",E, ...
                            "PoissonsRatio",nu, ...
                            "MassDensity",rho);
```

Specify the same constraint on the left end of the beam.

```
structuralBC(modelT,"Edge",4,"Constraint","fixed");
```

Specify the initial condition by using the static solution.

```
structuralIC(modelT,Rstatic)
```

```
ans = 
  NodalStructuralICs with properties:

    InitialDisplacement: [6511x2 double]
        InitialVelocity: [6511x2 double]
```

Solve the undamped transient model for three full periods corresponding to the lowest vibration mode.

```
tlist = 0:longestPeriod/100:3*longestPeriod;
resT = solve(modelT,tlist,"ModalResults",res);
```

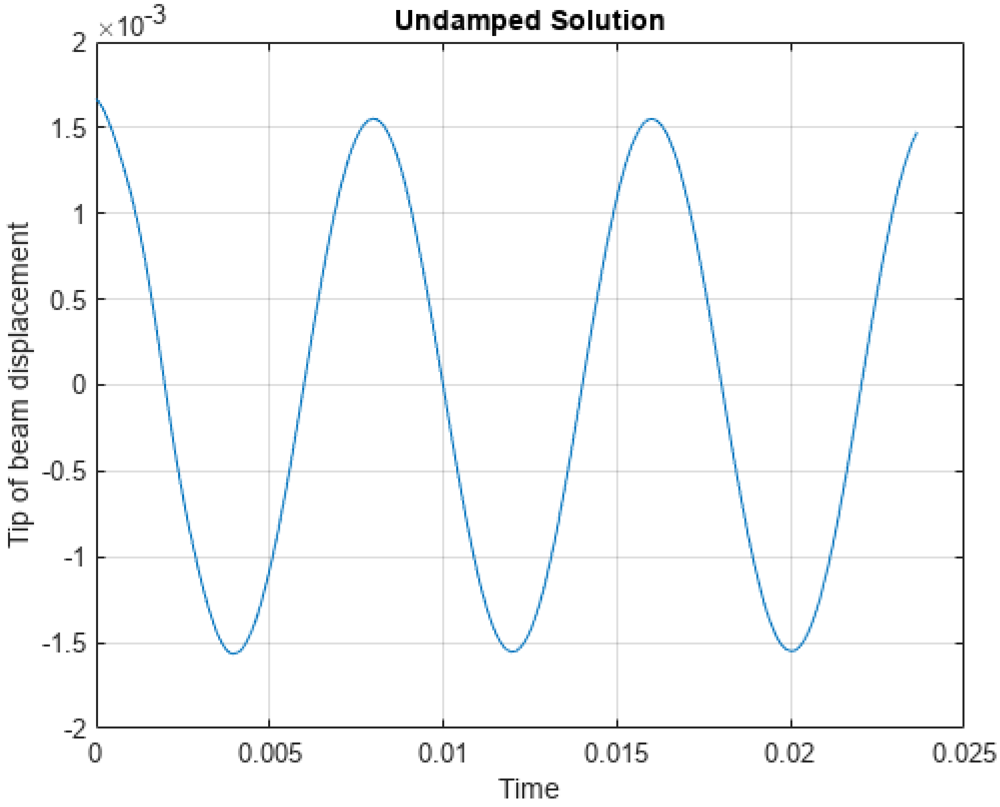
Interpolate the displacement at the tip of the beam.

```
intrpUt = interpolateDisplacement(resT,[5;0.05]);
```

The displacement at the tip is a sinusoidal function of time with amplitude equal to the initial *y*-displacement. This result agrees with the solution to the simple spring-mass system.

```
plot(resT.SolutionTimes,intrpUt.uy)
grid on
title("Undamped Solution")
```

```
xlabel("Time")
ylabel("Tip of beam displacement")
```



Now solve the model with damping equal to 3% of critical damping.

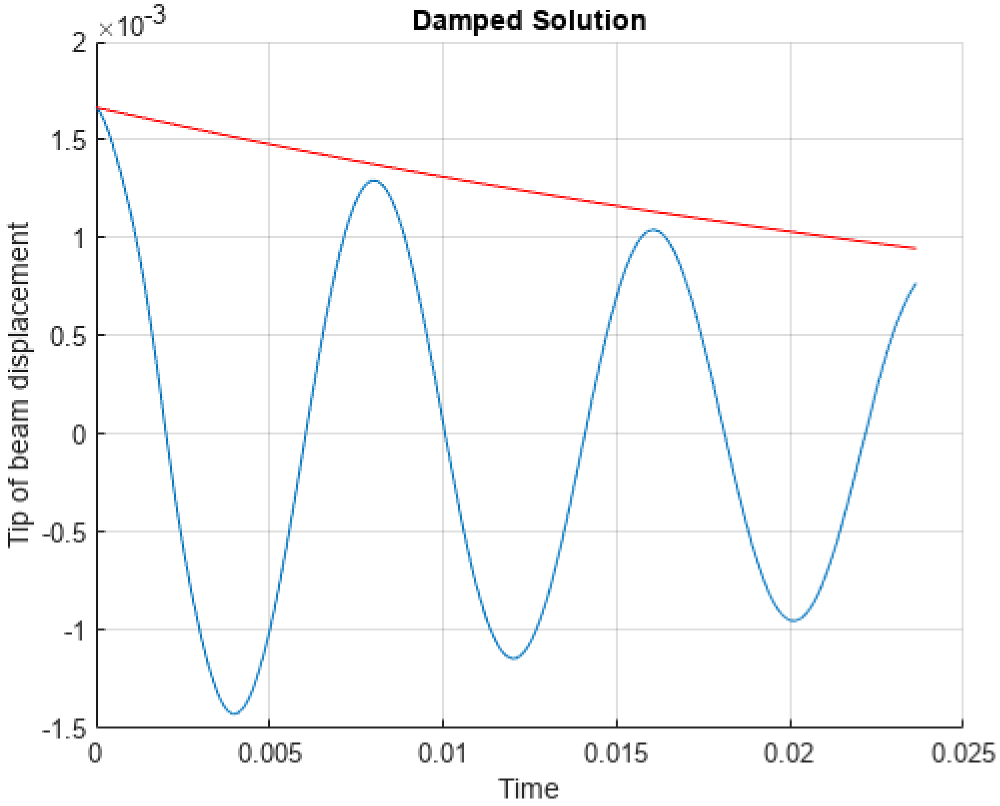
```
zeta = 0.03;
omega = 2*pi*freqNumerical;
structuralDamping(modelT,"Zeta",zeta);
resT = solve(modelT,tlist,"ModalResults",res);
```

Interpolate the displacement at the tip of the beam.

```
intrpUt = interpolateDisplacement(resT,[5;0.05]);
```

The *y*-displacement at the tip is a sinusoidal function of time with amplitude exponentially decreasing with time.

```
figure
hold on
plot(resT.SolutionTimes,intrpUt.uy)
plot(tlist,intrpUt.uy(1)*exp(-zeta*omega*tlist),"Color","r")
grid on
title("Damped Solution")
xlabel("Time")
ylabel("Tip of beam displacement")
```


Damped Solution
×10^-3
2
1.5
1
0.5
0
-0.5
-1
-1.5
Tip of beam displacement
0
0.005
0.01
0.015
0.02
0.025
Time

Dynamic Analysis of Clamped Beam

This example shows how to analyze the dynamic behavior of a beam under a uniform pressure load and clamped at both ends.

This example uses the Imperial system of units. If you replace them with values specified in the metric system, ensure that you specify all values using the same system.

In this example, the pressure load is suddenly applied at time equal to zero. The pressure magnitude is high enough to produce deflections on the same order as the beam thickness. Accurate prediction of this type of behavior requires geometrically nonlinear elasticity equations. This example solves the clamped beam elasticity problem using both linear and nonlinear formulations of elasticity equations.

One approach to handling the large deflections is to consider the elasticity equations in the deformed position. However, the toolbox uses the equations based on the original geometry. Therefore, you must use a Lagrangian formulation of nonlinear elasticity where stresses, strains, and coordinates refer to the original geometry. The Lagrangian formulation of the equilibrium equations is

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

where ρ 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. You also can write this equation in the 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 this formulation accounts for large deflections, it assumes that the strains remain small, so that linear elastic constitutive relations apply. For the 2-D plane stress case, you can write the constitutive relations in matrix form:

$$\begin{Bmatrix} S_{11} \\ S_{22} \\ S_{12} \end{Bmatrix} = \begin{bmatrix} C_{11} & C_{12} & \\ C_{12} & C_{22} & \\ & & 2G_{12} \end{bmatrix} \begin{Bmatrix} E_{11} \\ E_{22} \\ E_{12} \end{Bmatrix}$$

E_{ij} is the Green-Lagrange strain tensor:

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

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 the Young's modulus, and ν is the Poisson's ratio. For more details about the Lagrangian formulation for nonlinear elasticity, see [1] on page 3-32.

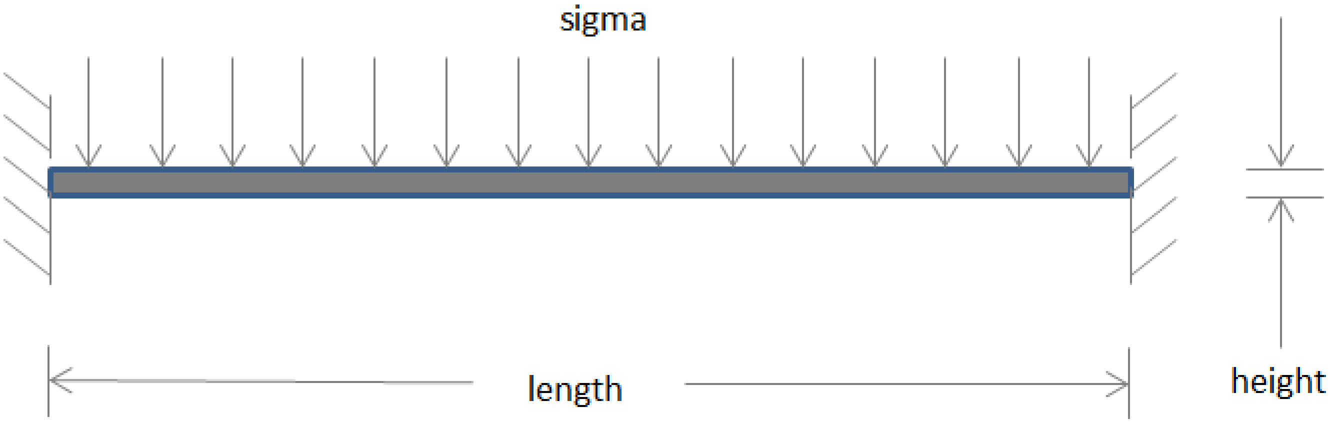
These equations completely define the geometrically nonlinear plane stress problem. This example uses Symbolic Math Toolbox™ to define the c coefficient in the form required by Partial Differential Equation Toolbox™. The c coefficient is a function `cCoefficientLagrangePlaneStress`. You can use it with any geometric nonlinear plane stress analysis of a model made from an isotropic material. You can find it under `matlab/R20XXx/examples/pde/main`.

Linear Solution

Create a PDE model for a system of two equations.

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

Create the following beam geometry.



Specify the length and thickness of the beam.

```
blength = 5; % Beam length, in
height = 0.1; % Thickness of the beam, in
```

Because the beam geometry and loading are symmetric about the beam center, you can simplify the model by considering only the right half of the beam.

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

Create the edges of the rectangle representing the beam.

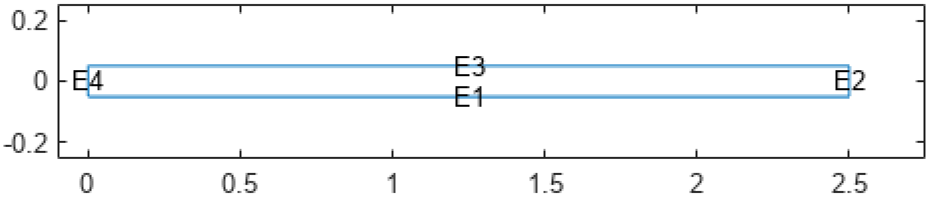
```
rect = [3 4 0 l2 l2 0 -h2 -h2  h2 h2]';
g = decsg(rect,'R1',('R1')');
```

Create the geometry from the edges and include it in the model.

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

Plot the geometry with the edge labels.

```
figure
pdegplot(g,"EdgeLabels","on")
axis([-.1 1.1*l2 -5*h2 5*h2])
```



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 = 0.3; % Poisson's ratio of the material
rho = 0.3/386; % 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. From the symmetry condition, the *x*-displacement equals zero at the left edge.

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

Because the beam is clamped, the *x*- and *y*-displacements equal zero along the 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 zero initial displacements and velocities.

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

Generate a mesh.

```
generateMesh(model);
```

Solve the model.

```
tlist = linspace(0,3e-3,100);
result = solvepde(model,tlist);
```

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

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

Nonlinear Solution

Specify the coefficients for the nonlinear case. The `cCoefficientLagrangePlaneStress` function takes the isotropic material properties and location and state structures, and returns a c-matrix for a nonlinear plane stress analysis. It assumes that strains are small, that is, E and ν are independent of the solution.

```
c  = @(location,state)cCoefficientLagrangePlaneStress(E,gnu, ...
                                                 location,state);
specifyCoefficients(model,"m",rho,"d",0,"c", c,"a",0,"f",f);
```

Solve the model.

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

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

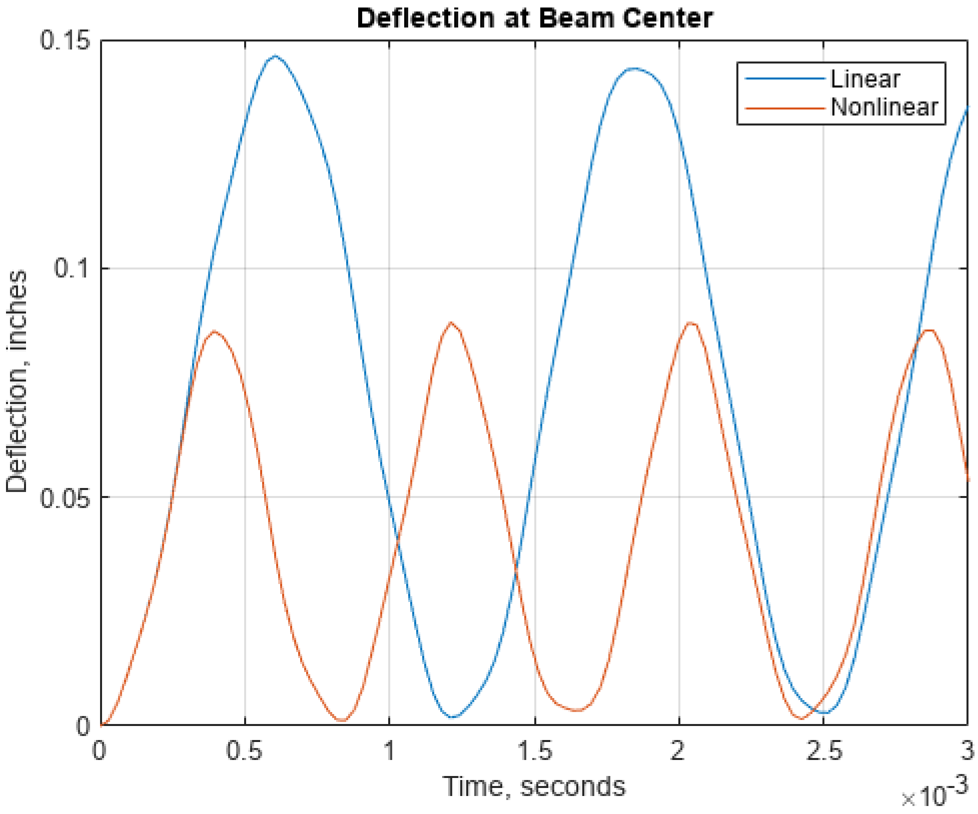
```
u4NonLinear = interpolateSolution(result,xc,yc,2,1:length(tlist));
```

Solution Plots

Plot the *y*-deflection at the center of the beam as a function of time. The nonlinear analysis yields substantially smaller displacements than the linear analysis. This "stress stiffening" effect also results 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 Series in Engineering of the Physical Sciences. Englewood Cliffs, NJ: Prentice-Hall, 1969.

Reduced-Order Modeling Technique for Beam with Point Load

This example shows how to eliminate degrees of freedom (DoFs) that are not on the boundaries of interest by using the Craig-Bampton reduced-order modeling technique. The example also uses the smaller dimension superelement to analyze the dynamics of the system. For comparison, the example also performs a direct transient analysis on the original structure.

Create a structural model for transient analysis.

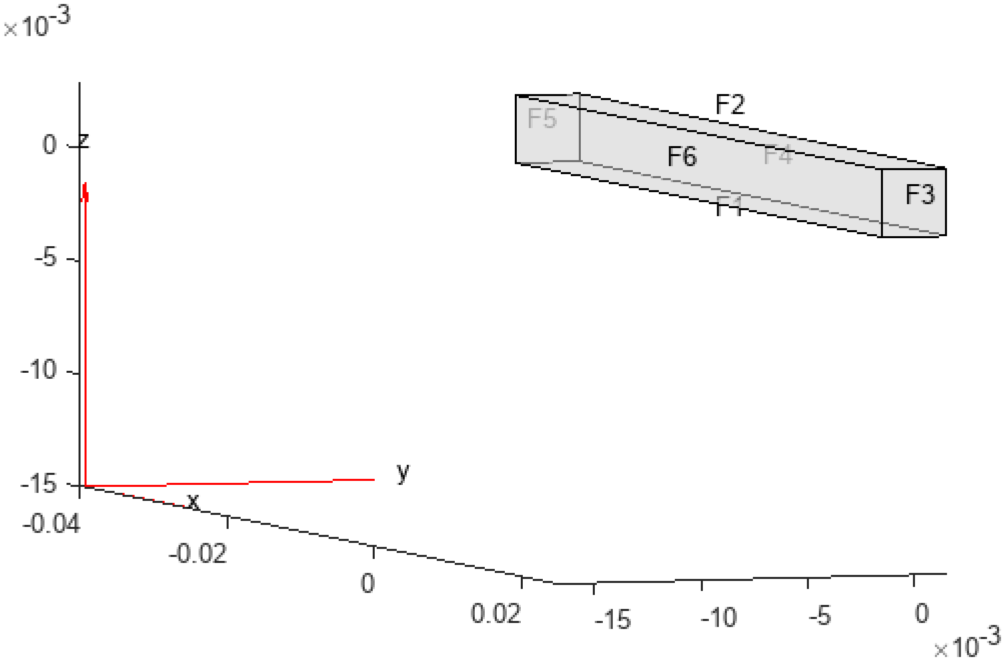
```
modelT = createpde("structural","transient-solid");
```

Create a square cross-section beam geometry and include it in the model.

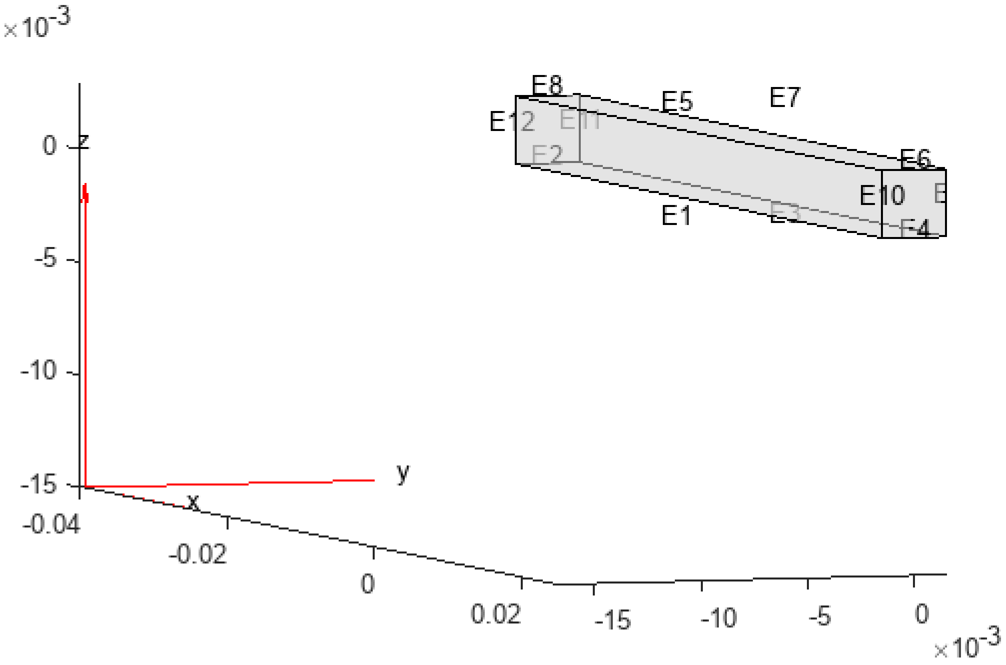
```
gm = multicuboid(0.05,0.003,0.003);
modelT.Geometry = gm;
```

Plot the geometry, displaying face and edge labels.

```
figure
pdegplot(modelT,"FaceLabels","on","FaceAlpha",0.5)
view([71 4])
```



```
figure
pdegplot(modelT,"EdgeLabels","on","FaceAlpha",0.5)
view([71 4])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelT,"YoungsModulus",210E9, ...
                            "PoissonsRatio",0.3, ...
                            "MassDensity",7800);
```

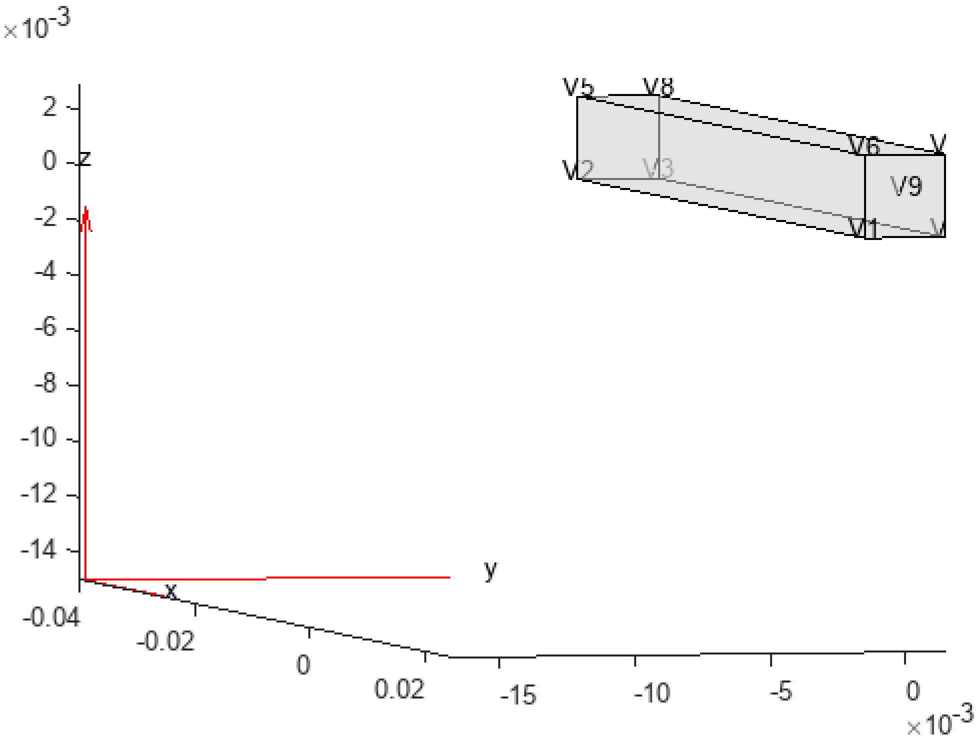
Fix one end of the beam.

```
structuralBC(modelT,"Edge",[2 8 11 12],"Constraint","fixed");
```

Add a vertex at the center of face 3.

```
loadedVertex = addVertex(gm,"Coordinates",[0.025 0.0 0.0015]);
```

```
figure
pdegplot(modelT,"VertexLabels","on","FaceAlpha",0.5)
view([78 2.5])
```



Generate a mesh.

```
generateMesh(modelT);
```

Apply a sinusoidal concentrated force in the *z*-direction on the new vertex.

```
structuralBoundaryLoad(modelT,"Vertex",loadedVertex, ...
                              "Force",[0;0;10],"Frequency",6000);
```

Specify zero initial conditions.

```
structuralIC(modelT,"Velocity",[0 0 0],"Displacement",[0 0 0]);
```

Solve the model.

```
tlist = 0:0.00005:3E-3;
RT = solve(modelT,tlist);
```

Define superelement interfaces using the fixed and loaded boundaries. In this case, the reduced order model retains the degrees of freedom (DoFs) on the fixed face and the loaded vertex while condensing all other DoFs in favor of modal DoFs. For better performance, use the set of edges bounding face 5 instead of using the entire face.

```
structuralSEInterface(modelT,"Edge",[2 8 11 12]);
structuralSEInterface(modelT,"Vertex",loadedVertex);
```

Reduce the structure, retaining all fixed interface modes up to `5e5`.

```
rom = reduce(modelT,"FrequencyRange",[-0.1,5e5]);
```

Next, use the reduced order model to simulate the transient dynamics. Use the `ode15s` function directly to integrate the reduced system ODE. Working with the reduced model requires indexing into the reduced system matrices `rom.K` and `rom.M`. First, construct mappings of indices of `K` and `M` to loaded and fixed DoFs by using the data available in `rom`.

DoFs correspond to translational displacements. If the number of mesh points in a model is `Nn`, then the toolbox assigns the IDs to the DoFs as follows: the first `1` to `Nn` are *x*-displacements, `Nn+1` to `2*Nn` are *y*-displacements, and `2Nn+1` to `3*Nn` are *z*-displacements. The reduced model object `rom` contains these IDs for the retained DoFs in `rom.RetainedDoF`.

Create a function that returns DoF IDs given node IDs and the number of nodes.

```
getDoF = @(x,numNodes) [x(:); x(:) + numNodes; x(:) + 2*numNodes];
```

Knowing the DoF IDs for the given node IDs, use the `intersect` function to find the required indices.

```
numNodes = size(rom.Mesh.Nodes,2);

loadedNode = findNodes(rom.Mesh,"region","Vertex",loadedVertex);
loadDoFs = getDoF(loadedNode,numNodes);
[~,loadNodeROMIds,~] = intersect(rom.RetainedDoF,loadDoFs);
```

In the reduced matrices `rom.K` and `rom.M`, generalized modal DoFs appear after the retained DoFs.

```
fixedIntModeIds = (numel(rom.RetainedDoF) + 1:size(rom.K,1))';
```

Because fixed-end DoFs are not a part of the ODE system, the indices for the ODE DoFs in reduced matrices are as follows.

```
odeDoFs = [loadNodeROMIds;fixedIntModeIds];
```

The relevant components of `rom.K` and `rom.M` for time integration are:

```
Kconstrained = rom.K(odeDoFs,odeDoFs);
Mconstrained = rom.M(odeDoFs,odeDoFs);
numODE = numel(odeDoFs);
```

Now you have a second-order system of ODEs. To use `ode15s`, convert this into a system of first-order ODEs by applying linearization. Such a first-order system is twice the size of the second-order system.

```
Mode = [eye(numODE,numODE),   zeros(numODE,numODE); ...
        zeros(numODE,numODE), Mconstrained];
Kode = [zeros(numODE,numODE), -eye(numODE,numODE); ...
        Kconstrained,         zeros(numODE,numODE)];
Fode = zeros(2*numODE,1);
```

The specified concentrated force load in the full system is along the *z*-direction, which is the third DoF in the ODE system. Accounting for the linearization to obtain the first-order system gives the loaded ODE DoF.

```
loadODEDoF = numODE + 3;
```

Specify the mass matrix and the Jacobian for the ODE solver.

```
odeoptions = odeset;
odeoptions = odeset(odeoptions,"Jacobian",-Kode);
odeoptions = odeset(odeoptions,"Mass",Mode);
```

Specify zero initial conditions.

```
u0 = zeros(2*numODE,1);
```

Solve the reduced system by using ode15s and the helper function `CMSODEf`, which is defined at the end of this example.

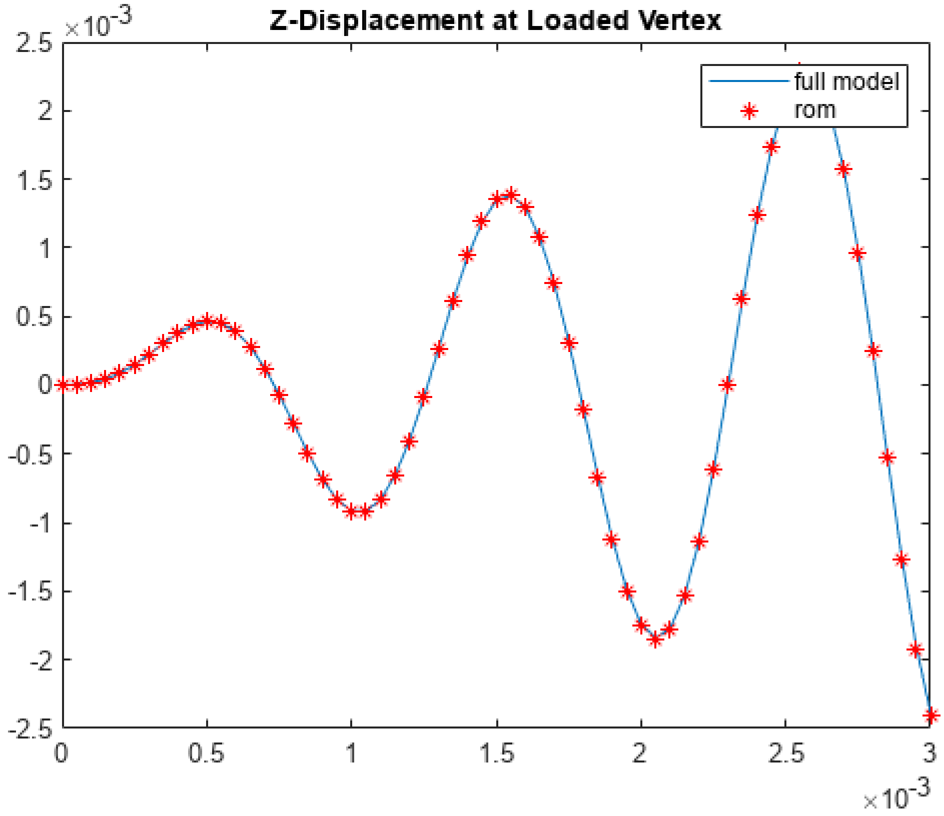
```
sol = ode15s(@(t,y) CMSODEf(t,y,Kode,Fode,loadODEDoF), ...
                              tlist,u0,odeoptions);
```

Compute the values of the ODE variable and the time derivatives.

```
[displ,vel] = deval(sol,tlist);
```

Plot the *z*-displacement at the loaded vertex and compare it to the third DoF in the solution of the reduced ODE system.

```
figure
plot(tlist,RT.Displacement.uz(loadedVertex,:))
hold on
plot(tlist,displ(3,:),"r*")
title("Z-Displacement at Loaded Vertex")
legend("full model","rom")
```



Knowing the solution in terms of the interface DoFs and modal DoFs, you can reconstruct the solution for the full model. The `reconstructSolution` function requires the displacement, velocity, and acceleration at all DoFs in `rom`. Construct the complete solution vector, including the zero values at the fixed DoFs.

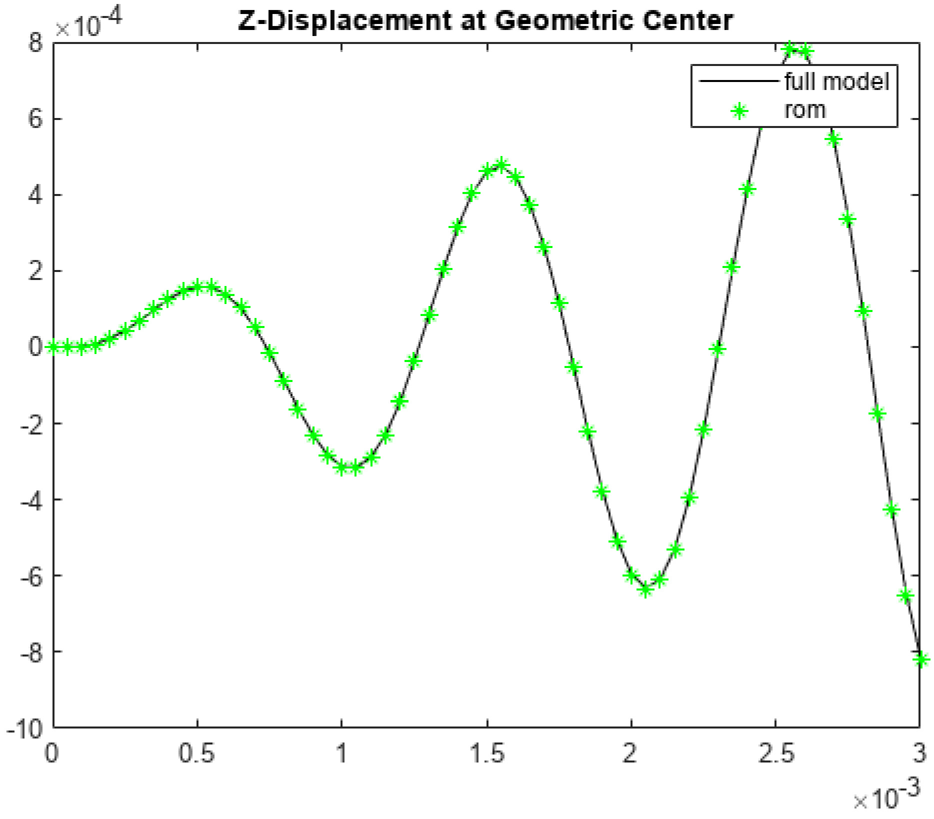
```
u = zeros(size(rom.K,1),numel(tlist));
ut = zeros(size(rom.K,1),numel(tlist));
utt = zeros(size(rom.K,1),numel(tlist));
u(odeDoFs,:) = displ(1:numODE,:);
ut(odeDoFs,:) = vel(1:numODE,:);
utt(odeDoFs,:) = vel(numODE+1:2*numODE,:);
```

Construct a transient results object using this solution.

```
RTrom = reconstructSolution(rom,u,ut,utt,tlist);
```

For comparison, compute the displacement in the interior at the center of the beam using the full and reconstructed solutions.

```
coordCenter = [0;0;0];
iDispRT = interpolateDisplacement(RT, coordCenter);
iDispRTrom = interpolateDisplacement(RTrom, coordCenter);
figure
plot(tlist,iDispRT.uz,"k")
hold on
plot(tlist,iDispRTrom.uz,"g*")
title("Z-Displacement at Geometric Center")
legend("full model","rom")
```



ODE Helper Function

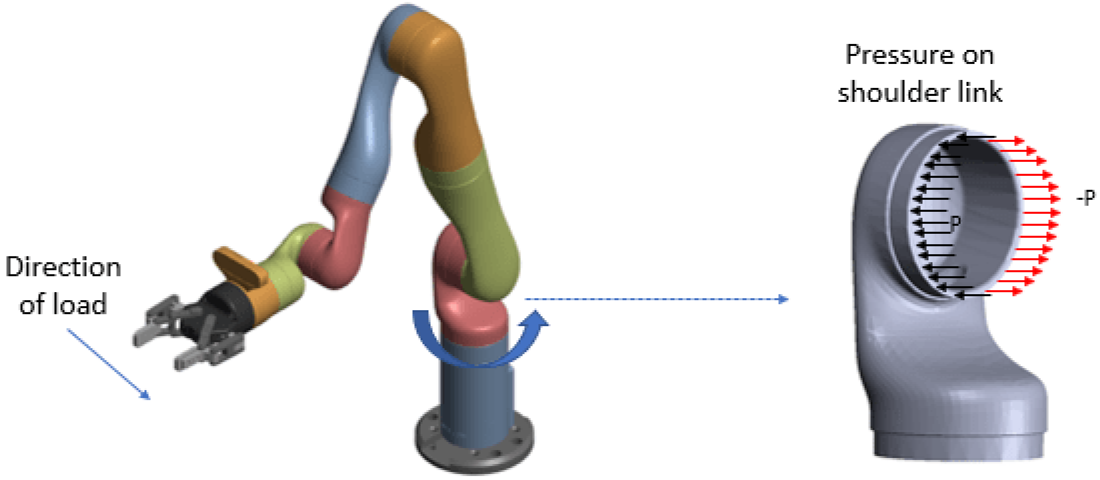
```
function f = CMSODEf(t,u,Kode,Fode,loadedVertex)
Fode(loadedVertex) = 10*sin(6000*t);
f = -Kode*u +Fode;
end
```

Modal and Frequency Response Analysis for Single Part of Kinova Gen3 Robotic Arm

This example shows how to analyze the shoulder link of a Kinova® Gen3 Ultra lightweight robotic arm for possible deformation under pressure.

Robotic arms perform precise manipulations in a wide variety of applications from factory automation to medical surgery. Typically, robotic arms consist of several links connected in a serial chain, with a base attached to a tabletop or the ground and an end-effector attached at the tip. These links must be structurally strong to avoid any vibrations when the rotors are moving with a load on them.

Loads at the tips of a robotic arm cause pressure on the joints of each link. The direction of pressure depends on the direction of the load.



This example computes deformations of the shoulder link under applied pressure by performing a modal analysis and frequency response analysis simulation. You can find the helper function `animateSixLinkModes.m` and the geometry file `Gen3Shoulder.stl` under `matlab/R20XXx/examples/pde/main`.

Modal Analysis

Assuming that one end of the robotic arm is fixed, find the natural frequencies and mode shapes.

Create a structural model for modal analysis.

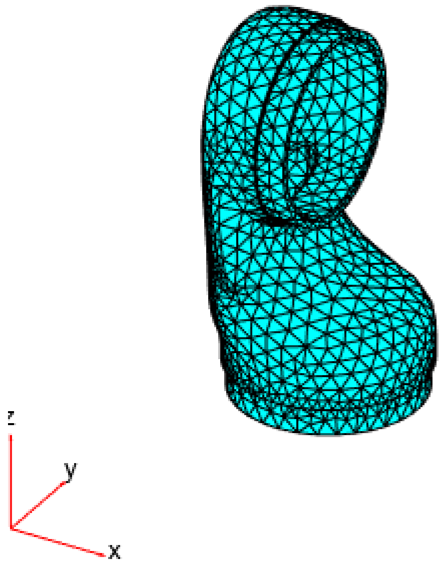
```
model = createpde("structural","modal-solid");
```

To perform unconstrained modal analysis of a structure, you must specify the geometry, mesh, and material properties. First, import the geometry of the shoulder part of the robotic arm.

```
importGeometry(model,"Gen3Shoulder.stl");
```

Generate a mesh.

```
generateMesh(model);
pdemesh(model)
```

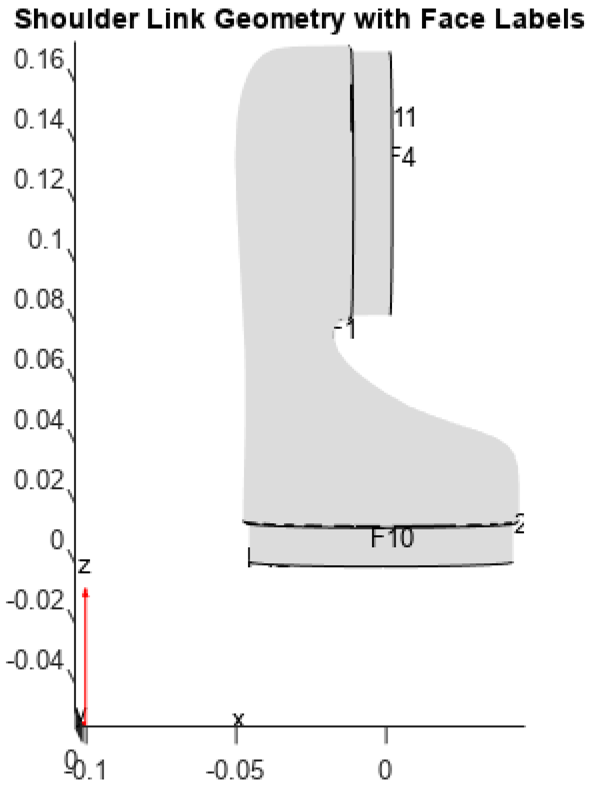


Specify Young's modulus, Poisson's ratio, and the mass density of the material in consistent units. Typically, the material used for the link is carbon fiber reinforced plastic. Assume that the material is homogeneous.

```
E = 1.5e11;
nu = 0.3;
rho = 2000;
structuralProperties(model,"YoungsModulus",E, ...
                           "PoissonsRatio",nu, ...
                           "MassDensity",rho);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face labels.

```
pdegplot(model,"FaceLabels","on")
view([-1 2])
title("Shoulder Link Geometry with Face Labels")
```



The shoulder link is fixed on one end (face 3) and connected to a moving link on the other end (face 4). Apply the fixed boundary condition on face 3.

```
structuralBC(model,"Face",3,"Constraint","fixed");
```

Solve the model for a chosen frequency range. Specify the lower frequency limit below zero so that all modes with frequencies near zero, if any, appear in the solution.

```
RF = solve(model,"FrequencyRange",[-1,10000]*2*pi);
```

By default, the solver returns circular frequencies.

```
modeID = 1:numel(RF.NaturalFrequencies);
```

Express the resulting frequencies in Hz by dividing them by 2π. Display the frequencies in a table.

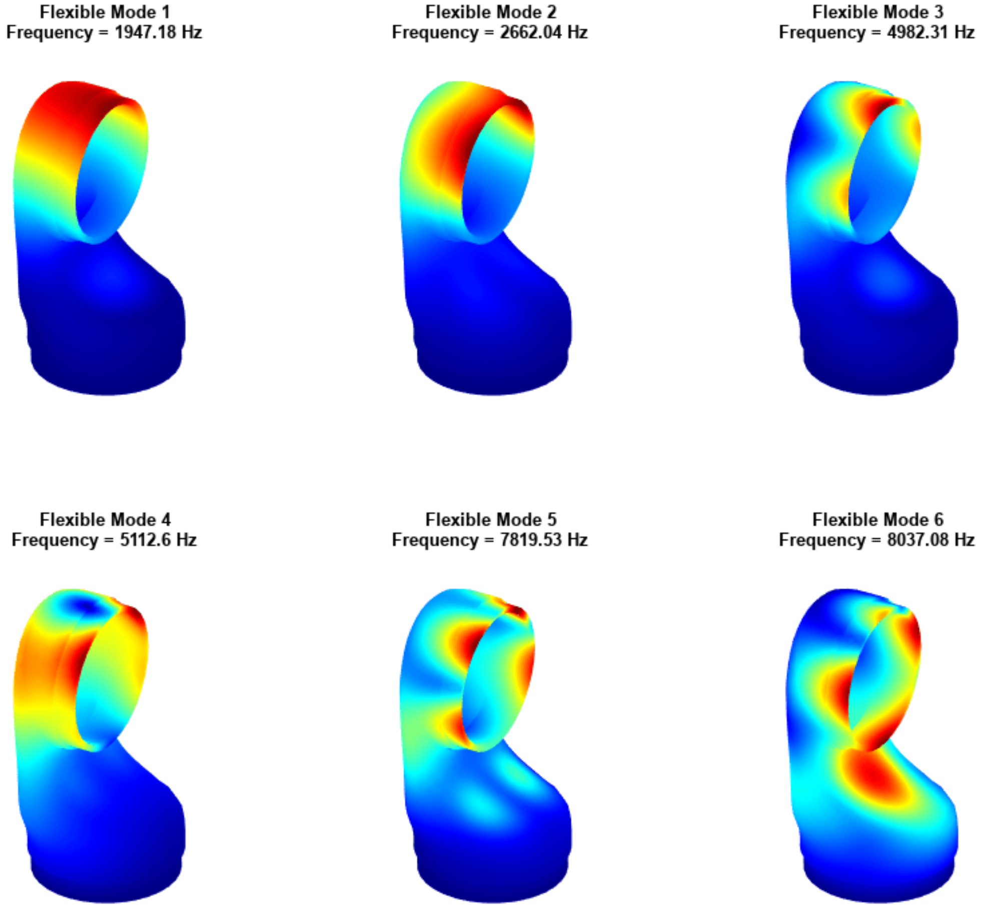
```
tmodalResults = table(modeID.',RF.NaturalFrequencies/2/pi);
tmodalResults.Properties.VariableNames = {'Mode','Frequency'};
disp(tmodalResults);
```

```
    Mode    Frequency
    ____    _________

     1       1947.2  
     2         2662  
     3       4982.3  
     4       5112.6  
     5       7819.5  
```

```
    6       8037.1
    7         9361
```

The best way to visualize the mode shapes is to animate the harmonic motion at their respective frequencies. The `animateSixLinkModes` function animates the first six modes. The resulting plot shows the areas of dominant deformation under load.

```
frames = animateSixLinkModes(RF);
```

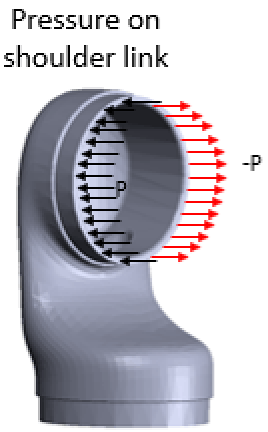


To play the animation, use this command:

```
movie(figure("units","normalized","outerposition",[0 0 1 1]),frames,5,30)
```

Frequency Response Analysis

Simulate the dynamics of the shoulder under pressure loading on a face, assuming that the attached link applies an equal and opposite amount of pressure on the halves of the face. Analyze the frequency response and deformation of a point in the face.



First, create a structural model for the frequency response analysis.

```
fmodel = createpde("structural","frequency-solid");
```

Import the same geometry for the shoulder part that you used for the modal analysis.

```
importGeometry(fmodel,"Gen3Shoulder.stl");
```

Generate a mesh.

```
mesh = generateMesh(fmodel);
```

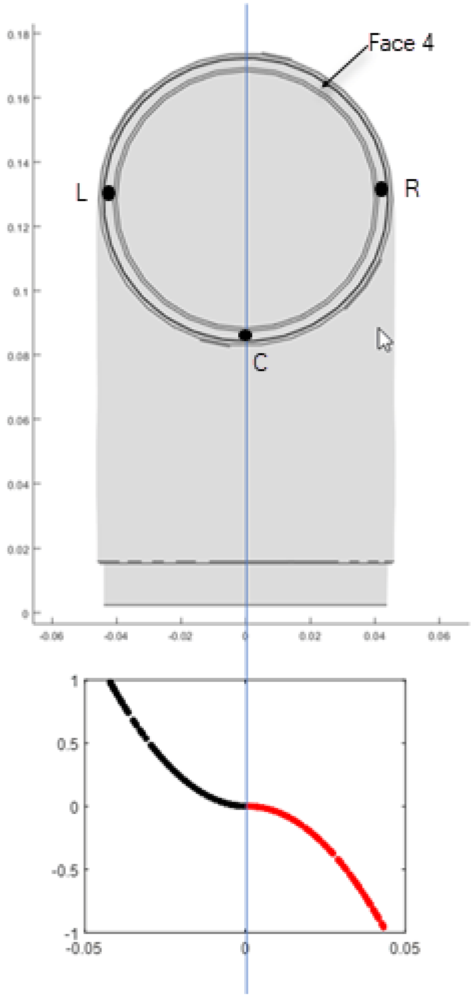
Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(fmodel,"YoungsModulus",E, ...
                            "PoissonsRatio",nu, ...
                            "MassDensity",rho);
```

The shoulder link is fixed on one end (face 3) and connected to a moving link on the other end (face 4). Apply the fixed boundary condition on face 3.

```
structuralBC(fmodel,"Face",3,"Constraint","fixed");
```

Estimate the pressure that the moving link applies on face 4 when the arm carries a load. This figure shows two halves of face 4 divided at the center along the *y*-coordinate.



Use the `pressFcnFR` function to apply the boundary load on face 4. This function applies a push and a twist pressure signal. The push pressure component is uniform. The twist pressure component applies positive pressure on the left side and negative pressure on the right side of the face. For the definition of the `pressFcnFR` function, see Pressure Function on page 3-50. This function does not have an explicit dependency on frequency. Therefore, in the frequency domain, this pressure load acts across all frequencies of the solution.

```
structuralBoundaryLoad(fmodel, ...
                       "Face",4, ...
                       "Pressure", ...
                        @(region,state)pressFcnFR(region,state), ...
                       "Vectorized","on");
```

Define the frequency list for the solution as 0 to 3500 Hz with 200 steps.

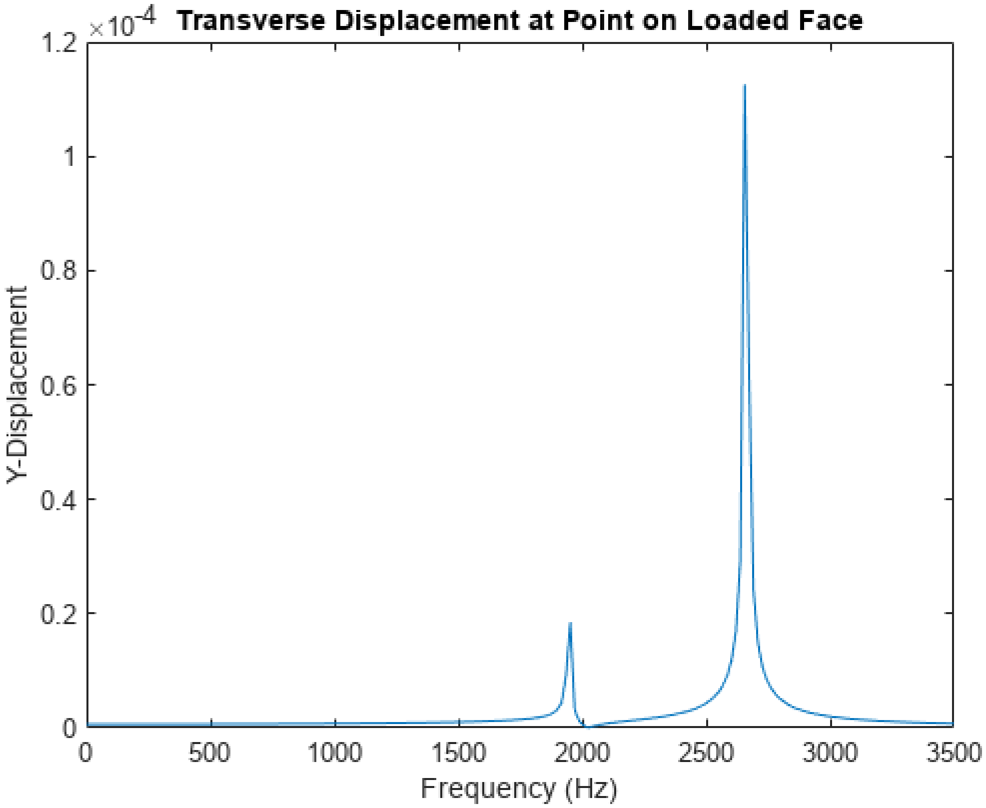
```
flist = linspace(0,3500,200)*2*pi;
```

Solve the model using the modal frequency response solver by specifying the modal results object RF as one of the inputs.

```
R = solve(fmodel,flist,"ModalResults",RF);
```

Plot the frequency response at a point on the loaded face. A point on face 4 located at maximum negative pressure loading is `(0.003; 0.0436; 0.1307)`. Interpolate the displacement to this point and plot the result.

```
queryPoint  = [0.003; 0.0436; 0.1307];
queryPointDisp = interpolateDisplacement(R,queryPoint);

figure
plot(R.SolutionFrequencies/2/pi,abs(queryPointDisp.uy))
title("Transverse Displacement at Point on Loaded Face")
xlabel("Frequency (Hz)")
ylabel("Y-Displacement")
xlim([0.0000 3500])
```



The peak of the response occurs near 2700 Hz, which is close to the second mode of vibration. A smaller response also occurs at the first mode close to 1950 Hz.

Find the peak response frequency index by using the `max` function with two output arguments. The second output argument provides the index of the peak frequency.

```
[M,I] = max(abs(queryPointDisp.uy))
```

```
M = 1.1256e-04
```

```
I = 152
```

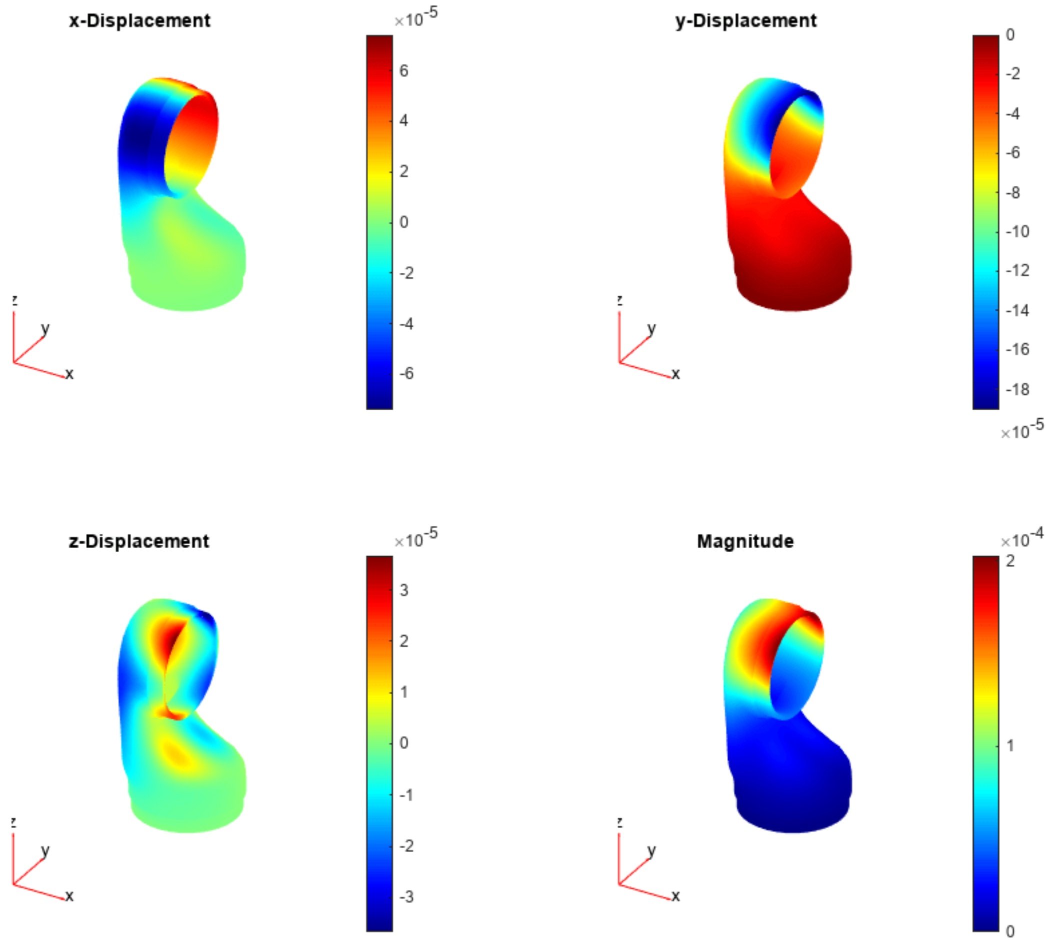
Find the peak response frequency value in Hz.

```
R.SolutionFrequencies(152)/2/pi
```

```
ans = 2.6558e+03
```

Plot the deformation at the peak response frequency. The applied load is such that it predominantly excites the opening mode and the bending mode of the shoulder.

```
RD = struct();
RD.ux = R.Displacement.ux(:,I);
RD.uy = R.Displacement.uy(:,I);
RD.uz = R.Displacement.uz(:,I);

figure("units","normalized","outerposition",[0 0 1 1]);

subplot(2,2,1)
pdeplot3D(fmodel,"ColorMapData",R.Displacement.ux(:,I), ...
                 "Deformation",RD,"DeformationScaleFactor",1);
title("x-Displacement")

subplot(2,2,2)
pdeplot3D(fmodel,"ColorMapData",R.Displacement.uy(:,I), ...
                 "Deformation",RD,"DeformationScaleFactor",1);
title("y-Displacement")

subplot(2,2,3)
pdeplot3D(fmodel,"ColorMapData",R.Displacement.uz(:,I), ...
                 "Deformation",RD,"DeformationScaleFactor",1);
title("z-Displacement")

subplot(2,2,4)
pdeplot3D(fmodel,"ColorMapData",R.Displacement.Magnitude(:,I), ...
                 "Deformation",RD,"DeformationScaleFactor",1);
title("Magnitude")
```

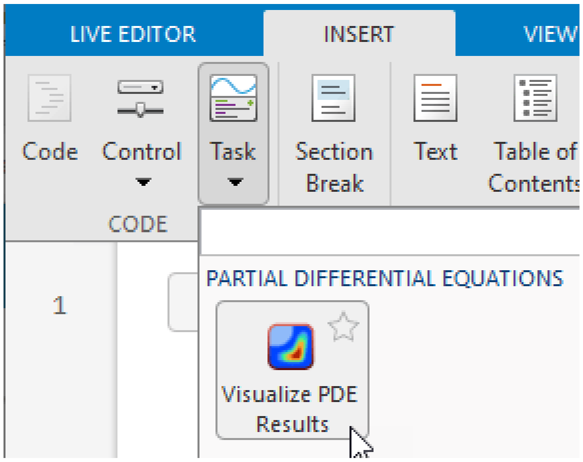


Clear figure for future plots.

```
clf
```

You also can plot the same results by using the **Visualize PDE Results** Live Editor task. First, create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

On the **Live Editor** tab, select **Task** > **Visualize PDE Results**. This action inserts the task into your script.



Plot the components and the magnitude of the displacement at the peak response frequency. To plot the *x*-displacement, follow these steps. To plot the *y*- and *z*-displacements and the magnitude, follow the same steps, but set **Component** to *Y*, *Z*, and *Magnitude*, respectively.

1. In the **Select results** section of the task, select R from the drop-down menu.
2. In the **Specify data parameters** section of the task, set **Type** to *Displacement*, **Component** to *X*, and **Frequency** to *2655.7789 Hz*.
3. In the **Specify visualization parameters** section of the task, clear the **Deformation** check box.

Visualize PDE Results

`resultViz` = X displacement of frequency 2655.7789 Hz in R

Select results

R

Specify data parameters

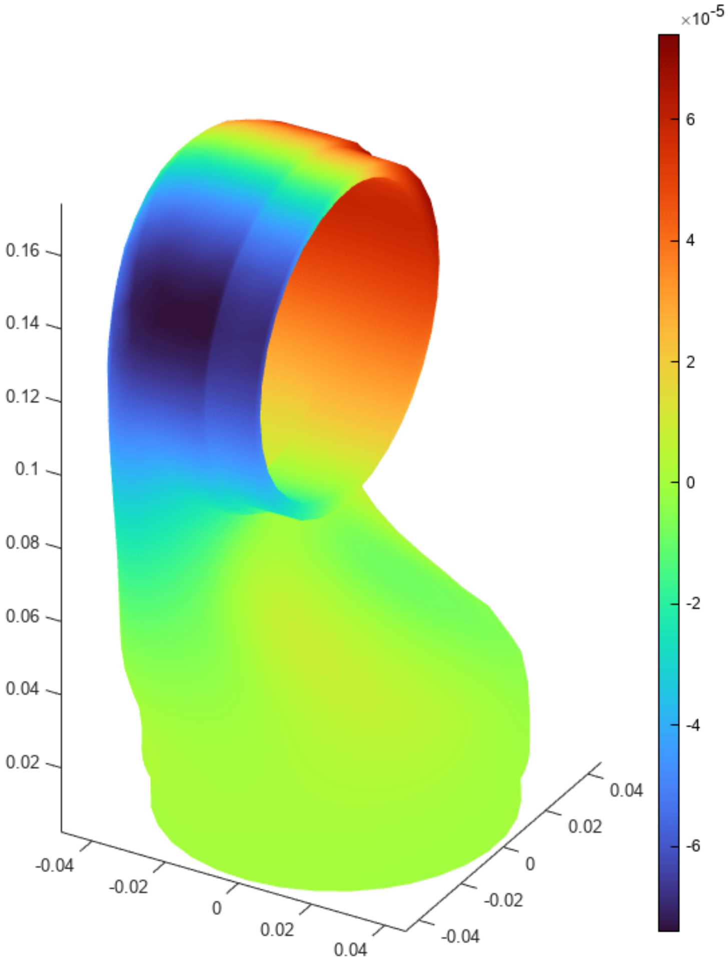
Type: Displacement
Component: X
Frequency: 2655.7789 Hz
Phase: 0 (0 to 2π)
☐ Animate

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☐ Deformation
Color limits: -7.414e-05 7.414e-05
Transparency: None Medium High

```
% Data to
meshData = R.Mesh;
nodalData = R.Displacement.ux(:,152);
deformationData = [R.Displacement.ux(:,152) ...
    R.Displacement.uy(:,152) ...
    R.Displacement.uz(:,152)];
phaseData = cospi(0) + 1i*sinpi(0);

% Create PDE result visualization
resultViz = pdeviz(meshData,nodalData*phaseData, ...
    "DeformationData",deformationData*phaseData, ...
```

```
    "DeformationScaleFactor",0, ...
    "ColorLimits",[-7.414e-05 7.414e-05]);
```



```
% Clear temporary variables
clearvars meshData nodalData deformationData phaseData
```

Pressure Function

Define a pressure function, `pressFcnFR`, to calculate a push and a twist pressure signal. The push pressure component is uniform. The twist pressure component applies positive pressure on the left side and negative pressure on the right side of the face. The value of the twist pressure loading increases in a parabolic distribution from the minimum at point `C` to the positive peak at `L` and to the negative peak at `R`. The twist pressure factor for the parabolic distribution obtained in `pressFcnFR` is multiplied with a sinusoidal function with a magnitude of 0.1 MPa. The uniform push pressure value is 10 kPa.

```
function p = pressFcnFR(region,~)

meanY = mean(region.y);
absMaxY = max(abs(region.y));
scaleFactor = zeros(size(region.y));

% Find IDs of the points on the left
% and right halves of the face
% using y-coordinate values.
leftHalfIdx = region.y <= meanY;
rightHalfIdx = region.y >= meanY;

% Define a parabolic scale factor
% for each half of the face.
scaleFactor(leftHalfIdx) = ...
    ((region.y(leftHalfIdx) - meanY)/absMaxY).^2;
scaleFactor(rightHalfIdx) = ...
   -((region.y(rightHalfIdx) - meanY)/absMaxY).^2;

p = 10E3 + 0.1E6*scaleFactor;

end
```

Thermal Stress Analysis of Jet Engine Turbine Blade

This example shows how to compute the thermal stress and deformation of a turbine blade in its steady-state operating condition. The blade has interior cooling ducts. The cool air flowing through the ducts maintains the temperature of the blade within the limit for its material. This feature is common in modern blades.

A turbine is a component of the jet engine. It is responsible for extracting energy from the high-temperature and high-pressure gas produced in the combustion chamber and transforming it into rotational motion to produce thrust. The turbine is a radial array of blades typically made of nickel alloys. These alloys resist the extremely high temperatures of the gases. At such temperatures, the material expands significantly, producing mechanical stress in the joints and significant deformations of several millimeters. To avoid mechanical failure and friction between the tip of the blade and the turbine casing, the blade design must account for the stress and deformations.

The example shows a three-step workflow:

1. Perform structural analysis accounting only for pressure of the surrounding gases while ignoring thermal effects.
2. Compute the thermal stress while ignoring the pressure.
3. Combine the pressure and thermal stress.

Pressure Loading

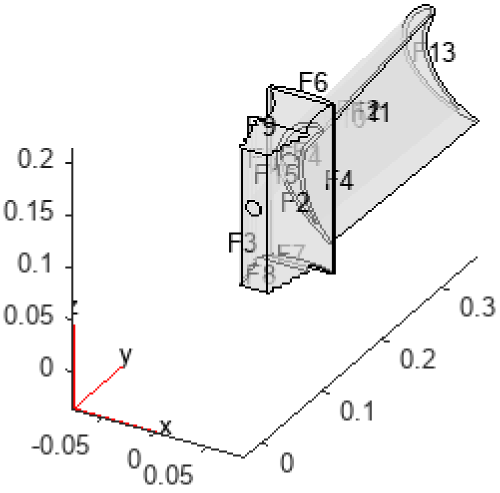
The blade experiences high pressure from the surrounding gases. Compute the stress caused only by this pressure.

First, create a static structural model.

```
smodel = createpde("structural","static-solid");
```

Import and plot the geometry, displaying face labels.

```
importGeometry(smodel,"Blade.stl");
figure
pdegplot(smodel,"FaceLabels","on","FaceAlpha",0.5)
```



Generate a mesh with the maximum element size 0.01.

```
msh = generateMesh(smodel,"Hmax",0.01);
```

Specify Young's modulus, Poisson's ratio, and the coefficient of thermal expansion for nickel-based alloy (NIMONIC 90).

```
E = 227E9; % in Pa
CTE = 12.7E-6; % in 1/K
nu = 0.27;

structuralProperties(smodel,"YoungsModulus",E, ...
                            "PoissonsRatio",nu, ...
                            "CTE",CTE);
```

Specify that the face of the root that is in contact with other metal is fixed.

```
structuralBC(smodel,"Face",3,"Constraint","fixed");
```

Specify the pressure load on the pressure and suction sides of the blade. This pressure is due to the high-pressure gas surrounding these sides of the blade.

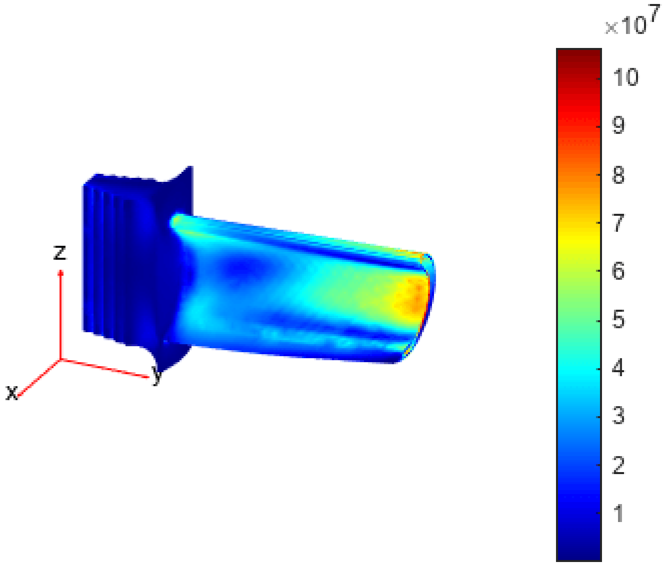
```
p1 = 5e5; %in Pa
p2 = 4.5e5; %in Pa

structuralBoundaryLoad(smodel,"Face",11,"Pressure",p1); % Pressure side
structuralBoundaryLoad(smodel,"Face",10,"Pressure",p2);  % Suction side
```

Solve the structural problem.

```
Rs = solve(smodel);
```

Plot the von Mises stress and the displacement. Specify a deformation scale factor of 100 to better visualize the deformation.

```
figure
pdeplot3D(smodel,"ColorMapData",Rs.VonMisesStress, ...
                 "Deformation",Rs.Displacement, ...
                 "DeformationScaleFactor",100)
view([116,25]);
```



The maximum stress is around 100 Mpa, which is significantly below the elastic limit.

Thermal Stress

Determine the temperature distribution and compute the stress and deformation due to thermal expansion only. This part of the example ignores the pressure.

First, create a thermal model for steady-state thermal analysis.

```
tmodel = createpde("thermal","steadystate");
```

Import the same geometry and use the same mesh as for the structural analysis.

```
importGeometry(tmodel,"Blade.stl");
tmodel.Mesh = msh;
```

Assuming that the blade is made of nickel-based alloy (NIMONIC 90), specify the thermal conductivity.

```
kapp = 11.5; % in W/m/K
thermalProperties(tmodel,"ThermalConductivity",kapp);
```

Convective heat transfer between the surrounding fluid and the faces of the blade defines the boundary conditions for this problem. The convection coefficient is greater where the gas velocity is higher. Also, the gas temperature is different around different faces. The temperature of the interior cooling air is $150^{\circ}C$, while the temperature on the pressure and suction sides is $1000^{\circ}C$.

```
% Interior cooling
thermalBC(tmodel,"Face",[15 12 14], ...
```

```
                 "ConvectionCoefficient",30, ...
                 "AmbientTemperature",150);
% Pressure side
thermalBC(tmodel,"Face",11, ...
                 "ConvectionCoefficient",50, ...
                 "AmbientTemperature",1000);
% Suction side
thermalBC(tmodel,"Face",10, ...
                 "ConvectionCoefficient",40, ...
                 "AmbientTemperature",1000);
% Tip
thermalBC(tmodel,"Face",13, ...
                 "ConvectionCoefficient",20, ...
                 "AmbientTemperature",1000);
% Base (exposed to hot gases)
thermalBC(tmodel,"Face",1, ...
                 "ConvectionCoefficient",40, ...
                 "AmbientTemperature",800);
% Root in contact with hot gases
thermalBC(tmodel,"Face",[6 9 8 2 7], ...
                 "ConvectionCoefficient",15, ...
                 "AmbientTemperature",400);
```

The boundary condition for the faces of the root in contact with other metal is a thermal contact that can be modeled as convection with a very large coefficient (around 1000 $W/\left(m^2K\right)$ for metal-metal contact).

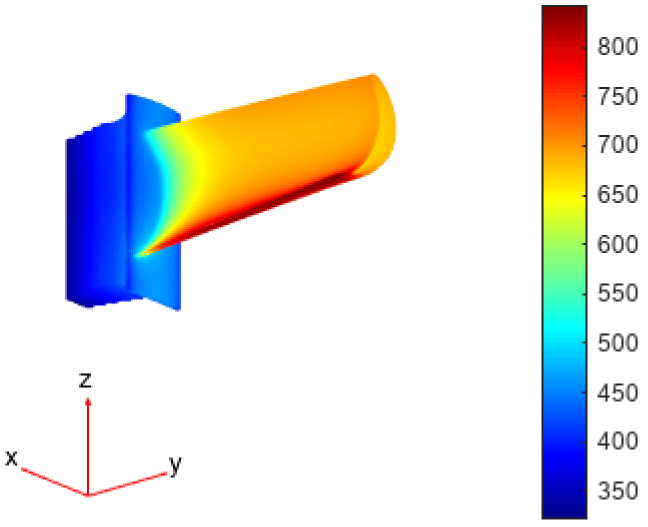
```
% Root in contact with metal
thermalBC(tmodel,"Face",[3 4 5], ...
                 "ConvectionCoefficient",1000, ...
                 "AmbientTemperature",300);
```

Solve the thermal model.

```
Rt = solve(tmodel);
```

Plot the temperature distribution. The temperature between the tip and the root ranges from around $820^{\circ}C$ to $330^{\circ}C$. The exterior gas temperature is $1000^{\circ}C$. The interior cooling is efficient: it significantly lowers the temperature.

```
figure
pdeplot3D(tmodel,"ColorMapData",Rt.Temperature)
view([130,-20]);
```



Now, create a static structural model to compute the stress and deformation due to thermal expansion.

```
tsmodel = createpde("structural","static-solid");
```

Import the same geometry, and use the same mesh and structural properties of the material as for the structural analysis.

```
importGeometry(tsmodel,"Blade.stl");
tsmodel.Mesh = msh;
structuralProperties(tsmodel,"YoungsModulus",E, ...
                             "PoissonsRatio",nu, ...
                             "CTE",CTE);
```

Specify the reference temperature.

```
tsmodel.ReferenceTemperature = 300; %in degrees C
structuralBodyLoad(tsmodel,"Temperature",Rt);
```

Specify the boundary condition.

```
structuralBC(tsmodel,"Face",3,"Constraint","fixed");
```

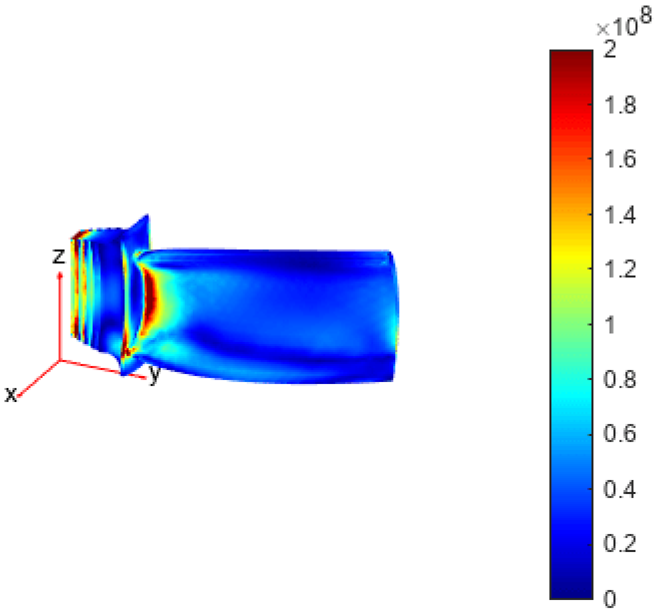
Solve the thermal stress problem.

```
Rts = solve(tsmodel);
```

Plot the von Mises stress and the displacement. Specify a deformation scale factor of 100 to better visualize the deformation. The stress concentrates in the constrained root because it cannot freely expand, and also in the junction between the blade and the root.

```
figure("units","normalized","outerposition",[0 0 1 1]);
pdeplot3D(tsmodel,"ColorMapData",Rts.VonMisesStress, ...
                  "Deformation",Rts.Displacement, ...
                  "DeformationScaleFactor",100)
```

```
caxis([0, 200e6])
view([116,25]);
```



Evaluate the displacement at the tip. In the design of the cover, this displacement must be taken into account to avoid friction between the cover and the blade.

```
max(Rts.Displacement.Magnitude)
```

```
ans = 0.0015
```

Combined Pressure Loading and Thermal Stress

Compute the stress and deformations caused by the combination of thermal and pressure effects.

Use the same model as for the thermal stress analysis. Add the pressure load on the pressure and suction sides of the blade. This pressure is due to the high-pressure gas surrounding these sides of the blade.

```
% Pressure side
structuralBoundaryLoad(tsmodel,"Face",11, ...
                               "Pressure",p1);
% Suction side
structuralBoundaryLoad(tsmodel,"Face",10, ...
                               "Pressure",p2);
```

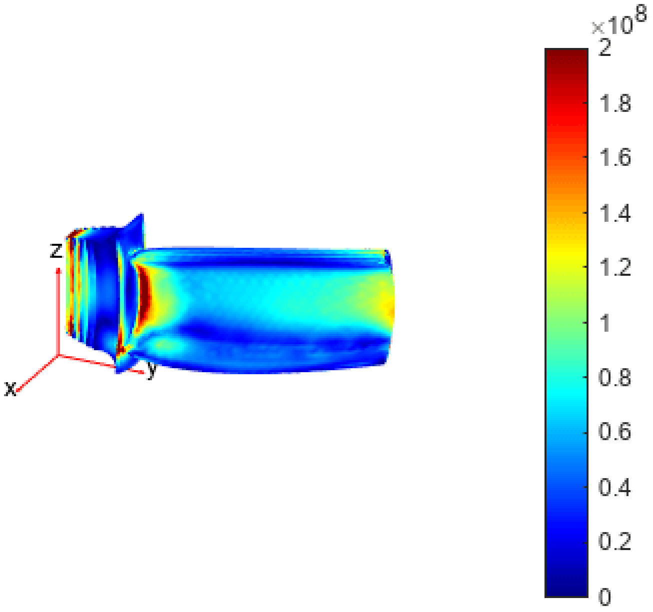
Solve the model.

```
Rc = solve(tsmodel);
```

Plot the von Mises stress and the displacement. Specify a deformation scale factor of 100 to better visualize the deformation.

```
figure("units","normalized","outerposition",[0 0 1 1]);
pdeplot3D(tsmodel,"ColorMapData",Rc.VonMisesStress, ...
```

```
                  "Deformation",Rc.Displacement, ...
                  "DeformationScaleFactor",100)
caxis([0, 200e6])
view([116,25]);
```



Evaluate the maximum stress and maximum displacement. The displacement is almost the same as for the thermal stress analysis, while the maximum stress, 854 MPa, is significantly higher.

```
max(Rc.VonMisesStress)
```

```
ans = 9.8378e+08
```

```
max(Rc.Displacement.Magnitude)
```

```
ans = 0.0015
```

Finite Element Analysis of Electrostatically Actuated MEMS Device

This example shows a simple approach to the coupled electromechanical finite element analysis of an electrostatically actuated micro-electromechanical (MEMS) device. For simplicity, this example uses the relaxation-based algorithm rather than the Newton method to couple the electrostatic and mechanical domains.

MEMS Devices

MEMS devices typically consist of movable thin beams or electrodes with a high aspect ratio that are suspended over a fixed electrode.

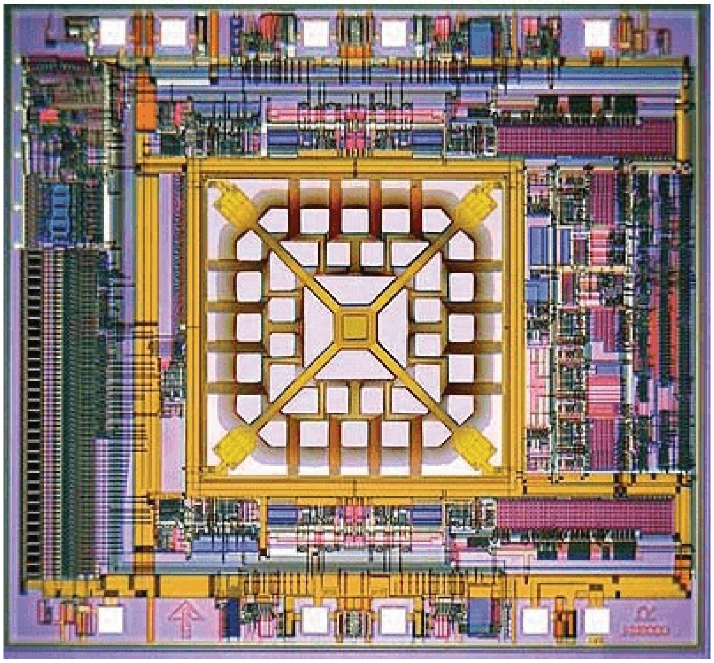


Figure 1. MEMS-based accelerometer. Image courtesy MEMSIC Inc.

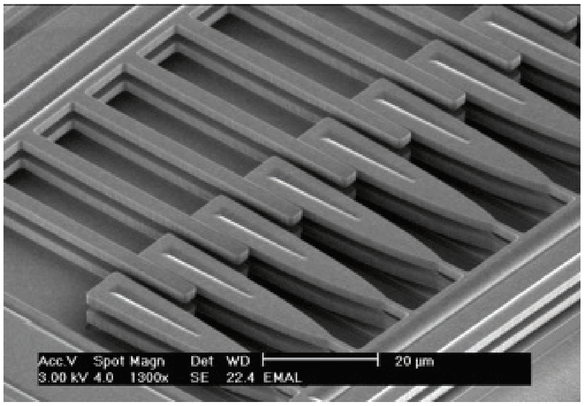


Figure 2. Electrostatic comb drive. Image courtesy Compliant Mechanisms Research Group, Brigham Young University.

Actuation, switching, and other signal and information processing functions can use the electrode deformation caused by the application of voltage between the movable and fixed electrodes. FEM provides a convenient tool for characterizing the inner workings of MEMS devices, and can predict temperatures, stresses, dynamic response characteristics, and possible failure mechanisms. One of the most common MEMS switches is the series of cantilever beams suspended over a ground electrode.

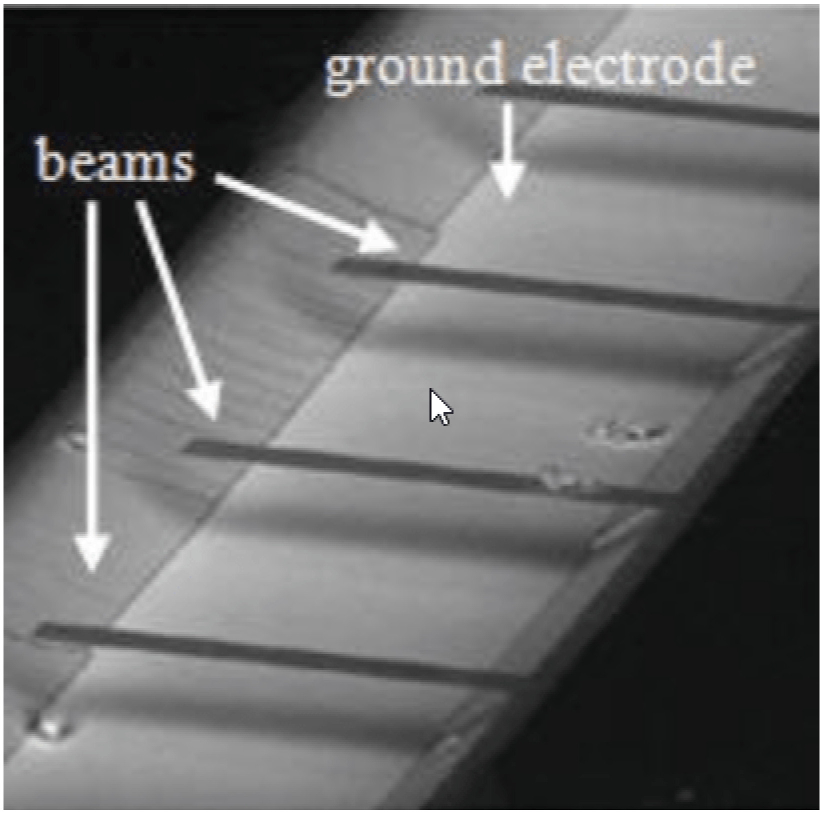


Figure 3. MEMS cantilever switch. Image courtesy Advanced Diamond Technologies.

This example uses the following geometry to model a MEMS switch. The top electrode is 150 µm in length and 2 µm in thickness. Young's modulus E is 170 GPa, and Poisson's ratio υ is 0.34. The bottom electrode is 50 µm in length and 2 µm in thickness, and is located 100 µm from the leftmost end of the top electrode. The gap between the top and bottom electrodes is 2 µm.

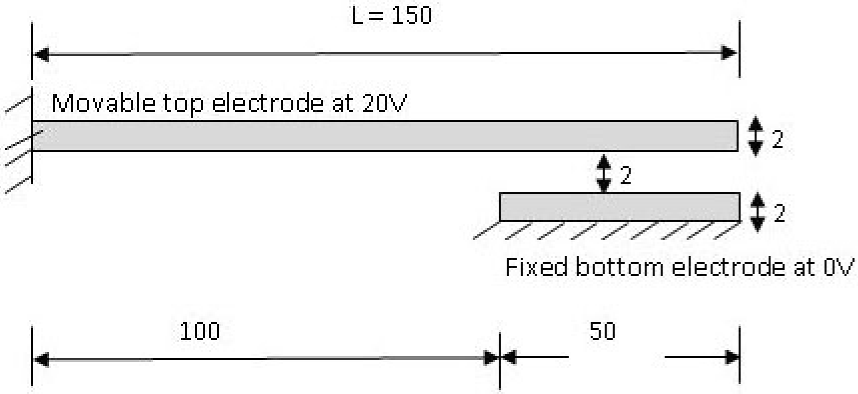


Figure 4. Cantilever switch modeled geometry.

A voltage applied between the top electrode and the ground plane induces electrostatic charges on the surface of the conductors, which in turn leads to electrostatic forces acting normal to the surface of the conductors. Because the ground plane is fixed, the electrostatic forces deform only the top electrode. When the beam deforms, the charge redistributes on the surface of the conductors. The resultant electrostatic forces and the deformation of the beam also change. This process continues until the system reaches a state of equilibrium.

Approach for Coupled Electromechanical Analysis

For simplicity, this example uses the relaxation-based algorithm rather than the Newton method to couple the electrostatic and mechanical domains. The example follows these steps:

1. Solve the electrostatic FEA problem in the nondeformed geometry with the constant potential $V0$ on the movable electrode.

2. Compute the load and boundary conditions for the mechanical solution by using the calculated values of the charge density along the movable electrode. The electrostatic pressure on the movable electrode is given by

$$P = \frac{1}{2\epsilon}|D|^2,$$

where $|D|$ is the magnitude of the electric flux density and ϵ is the electric permittivity next to the movable electrode.

3. Compute the deformation of the movable electrode by solving the mechanical FEA problem.

4. Update the charge density along the movable electrode by using the calculated displacement of the movable electrode,

$$|D_{\mathrm{def}}(x)| \approx |D_0(x)|\frac{G}{G - v(x)},$$

where $|D_{\mathrm{def}}(x)|$ is the magnitude of the electric flux density in the deformed electrode, $|D_0(x)|$ is the magnitude of the electric flux density in the undeformed electrode, G is the distance between the movable and fixed electrodes in the absence of actuation, and $v(x)$ is the displacement of the movable electrode at position x along its axis.

5. Repeat steps 2–4 until the electrode deformation values in the last two iterations converge.

Electrostatic Analysis

In the electrostatic analysis part of this example, you compute the electric potential around the electrodes.

First, create the cantilever switch geometry by using the constructive solid geometry (CSG) modeling approach. The geometry for electrostatic analysis consists of three rectangles represented by a matrix. Each column of the matrix describes a basic shape.

```
rect_domain = [3 4 1.75e-4 1.75e-4 -1.75e-4 -1.75e-4 ...
                   -1.7e-5 1.3e-5 1.3e-5 -1.7e-5]';
rect_movable = [3 4 7.5e-5 7.5e-5 -7.5e-5 -7.5e-5 ...
                     2.0e-6 4.0e-6 4.0e-6 2.0e-6]';
rect_fixed = [3 4 7.5e-5 7.5e-5 2.5e-5 2.5e-5 -2.0e-6 0 0 -2.0e-6]';
gd = [rect_domain,rect_movable,rect_fixed];
```

Create a name for each basic shape. Specify the names as a matrix whose columns contain the names of the corresponding columns in the basic shape matrix.

```
ns = char('rect_domain','rect_movable','rect_fixed');
ns = ns';
```

Create a formula describing the unions and intersections of the basic shapes.

```
sf = 'rect_domain-(rect_movable+rect_fixed)';
```

Create the geometry by using the `decsg` function.

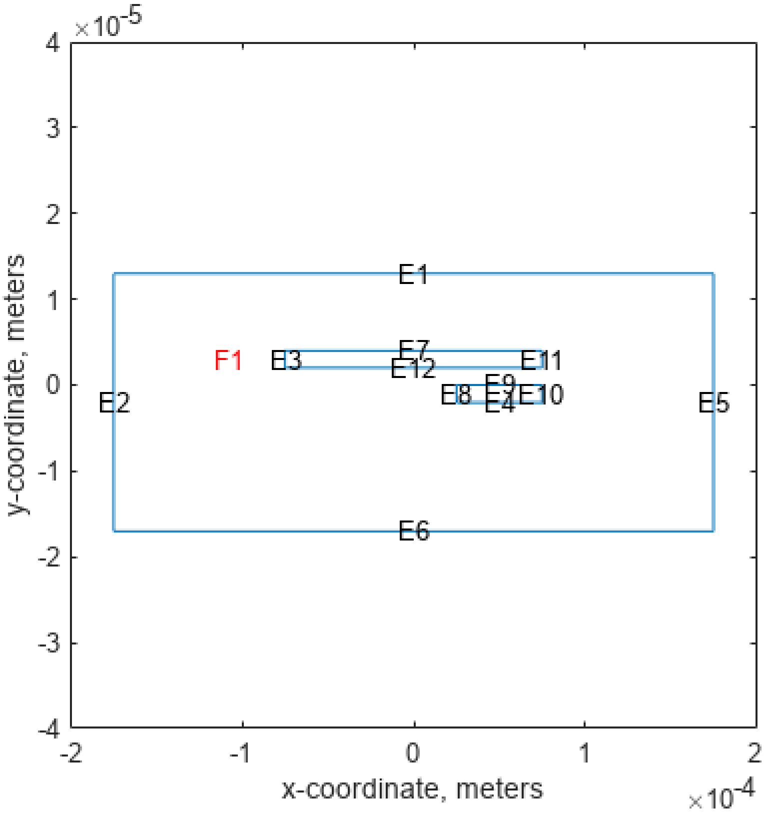
```
dl = decsg(gd,sf,ns);
```

Create a PDE model and include the geometry in the model.

```
model = createpde;
geometryFromEdges(model,dl);
```

Plot the geometry.

```
pdegplot(model,"EdgeLabels","on","FaceLabels","on")
xlabel("x-coordinate, meters")
ylabel("y-coordinate, meters")
axis([-2e-4,2e-4,-4e-5,4e-5])
axis square
```



The edge numbers in this geometry are:

- Movable electrode: E3, E7, E11, E12
- Fixed electrode: E4, E8, E9, E10
- Domain boundary: E1, E2, E5, E6

Set constant potential values of 20 V to the movable electrode and 0 V to the fixed electrode and domain boundary.

```
V0 = 0;
V1 = 20;
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",[4,8,9,10],"u",V0);
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",[1,2,5,6],"u",V0);
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",[3,7,11,12],"u",V1);
```

The PDE governing this problem is the Poisson equation,

$$-\nabla \cdot (\epsilon \nabla V) = \rho,$$

where ϵ is the coefficient of permittivity and ρ is the charge density. 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,

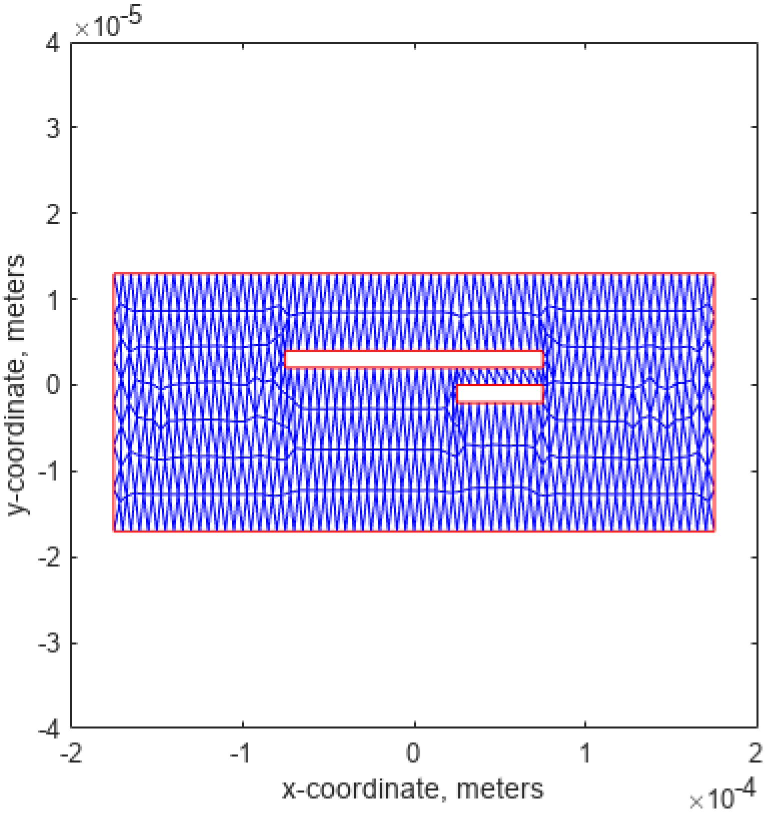
$$\Delta V = 0.$$

Specify the coefficients.

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

Generate a relatively fine mesh.

```
hmax = 5e-6;
generateMesh(model,"Hmax",hmax);
pdeplot(model)
xlabel("x-coordinate, meters")
ylabel("y-coordinate, meters")
axis([-2e-4, 2e-4,-4e-5, 4e-5])
axis square
```

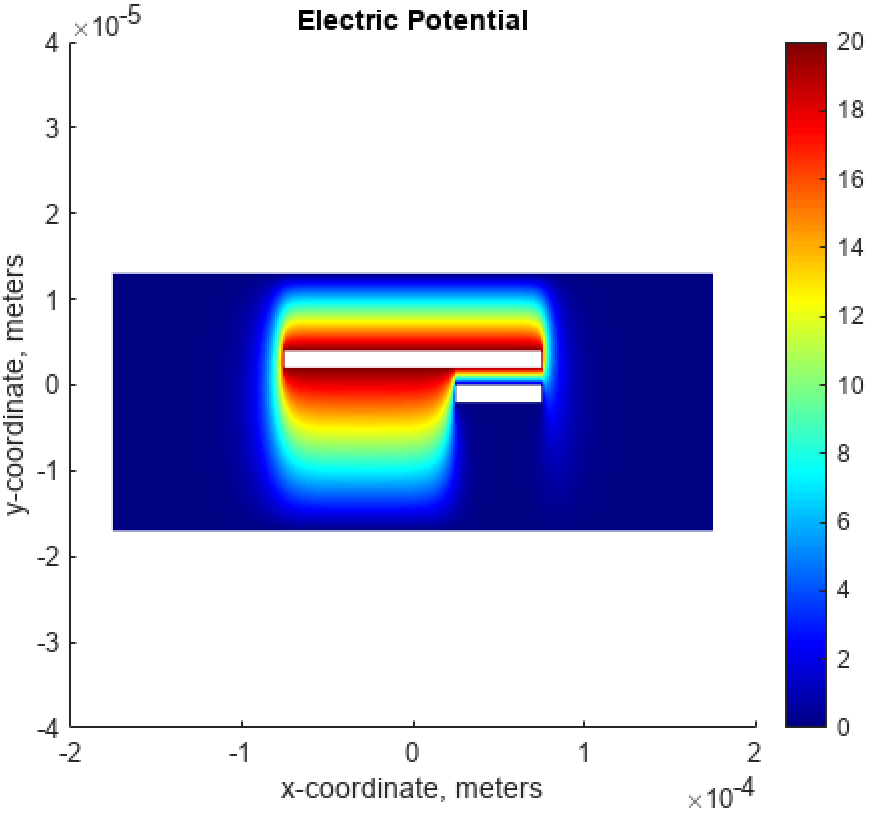


Solve the model.

```
results = solvepde(model);
```

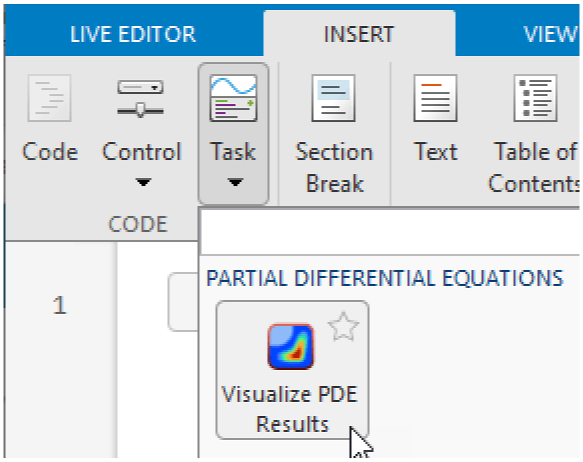
Plot the electric potential in the exterior domain.

```
u = results.NodalSolution;
figure
pdeplot(model,"XYData",results.NodalSolution, ...
              "ColorMap","jet");

title("Electric Potential");
xlabel("x-coordinate, meters")
ylabel("y-coordinate, meters")
axis([-2e-4,2e-4,-4e-5,4e-5])
axis square
```



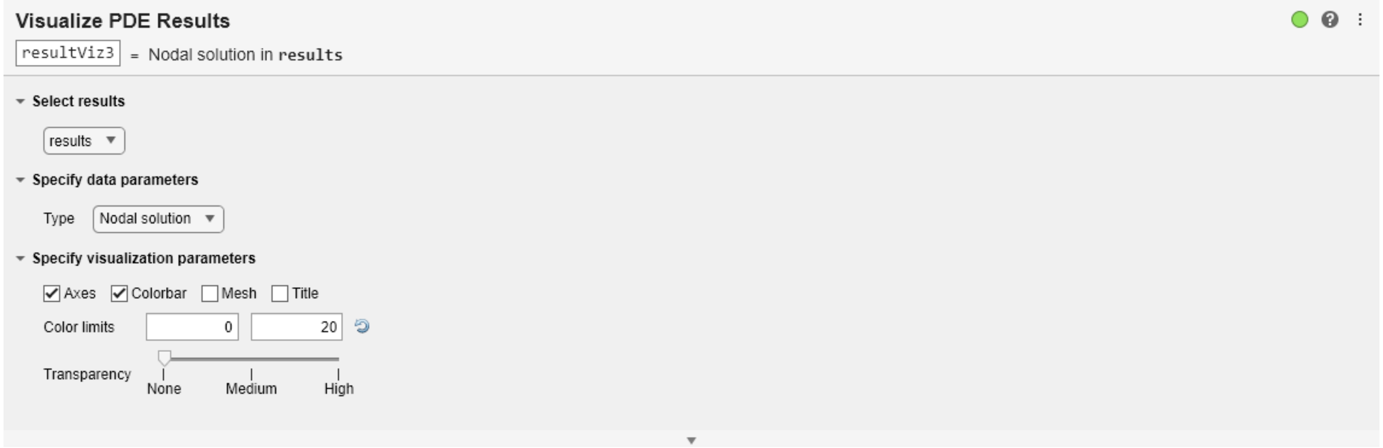
You also can plot the electric potential in the exterior domain by using the **Visualize PDE Results** Live Editor task. First, create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

On the **Live Editor** tab, select **Task > Visualize PDE Results**. This action inserts the task into your script.



To plot the electric potential, follow these steps.

1. In the **Select results** section of the task, select `results` from the drop-down list.
2. In the **Specify data parameters** section of the task, set **Type** to *Nodal solution*.



Visualize PDE Results

`resultViz3` = Nodal solution in `results`

Select results

results

Specify data parameters

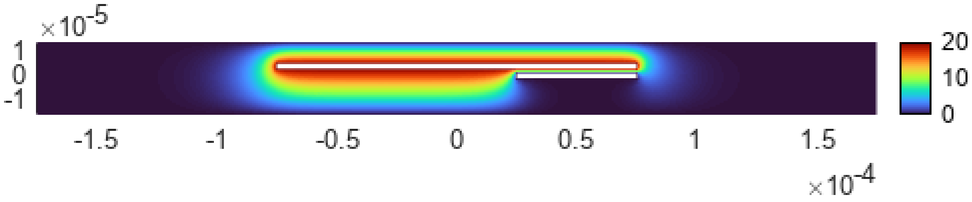
Type: Nodal solution

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title

Color limits: 0 20

Transparency: None Medium High



Mechanical Analysis

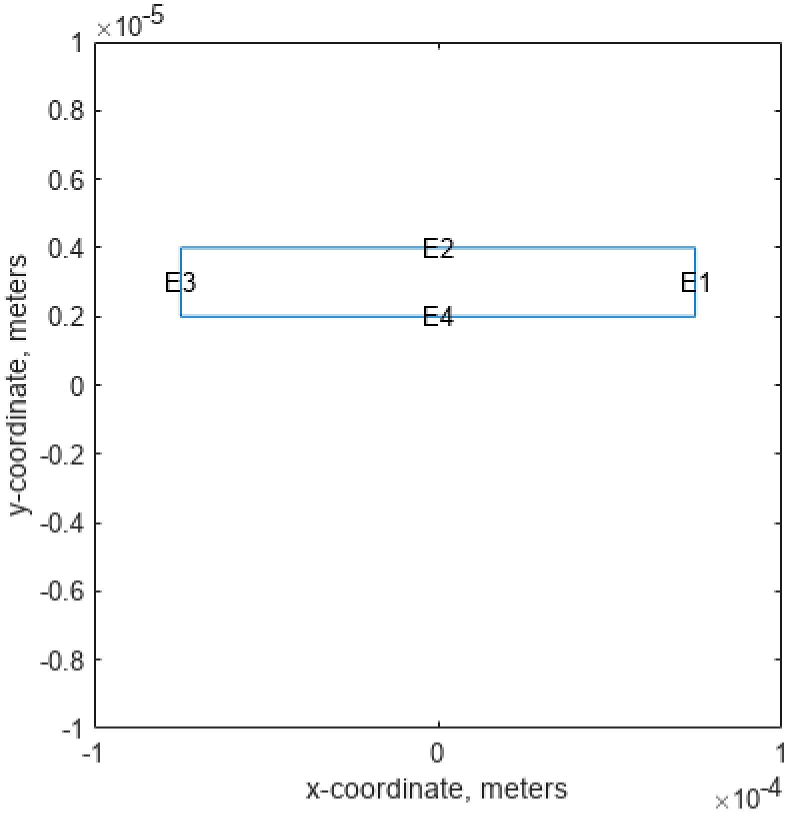
In the mechanical analysis part of this example, you compute the deformation of the movable electrode.

Create a structural model.

```
structuralmodel = createpde("structural","static-planestress");
```

Create the movable electrode geometry and include it in the model. Plot the geometry.

```
dl = decsg(rect_movable);
geometryFromEdges(structuralmodel,dl);
pdegplot(structuralmodel,"EdgeLabels","on")
xlabel("x-coordinate, meters")
ylabel("y-coordinate, meters")
axis([-1e-4,1e-4,-1e-5,1e-5])
axis square
```



Specify the structural properties: Young's modulus *E* is 170 GPa and Poisson's ratio *ν* is 0.34.

```
structuralProperties(structuralmodel,"YoungsModulus",170e9, ...
                                     "PoissonsRatio",0.34);
```

Specify the pressure as a boundary load on the edges. The pressure tends to draw the conductor into the field regardless of the sign of the surface charge. For the definition of the `CalculateElectrostaticPressure` function, see Electrostatic Pressure Function on page 3-71.

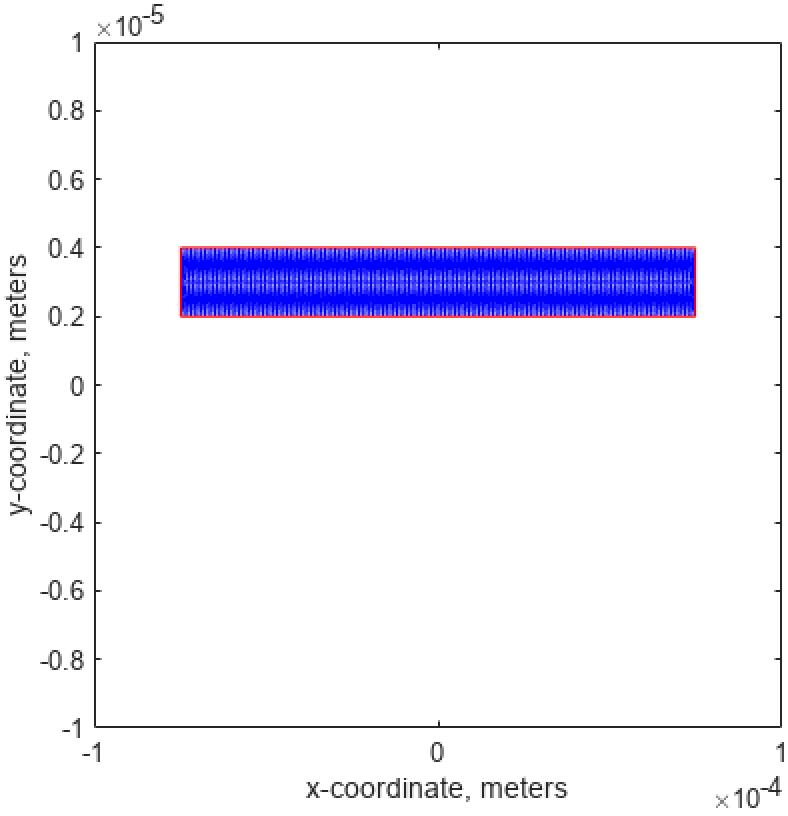
```
pressureFcn = @(location,state) - ...
              CalculateElectrostaticPressure(results,[],location);
structuralBoundaryLoad(structuralmodel,"Edge",[1,2,4], ...
                                       "Pressure",pressureFcn, ...
                                       "Vectorized","on");
```

Specify that the movable electrode is fixed at edge 3.

```
structuralBC(structuralmodel,"Edge",3,"Constraint","fixed");
```

Generate a mesh.

```
hmax = 1e-6;
generateMesh(structuralmodel,"Hmax",hmax);
pdeplot(structuralmodel);
xlabel("x-coordinate, meters")
ylabel("y-coordinate, meters")
axis([-1e-4, 1e-4,-1e-5, 1e-5])
axis square
```

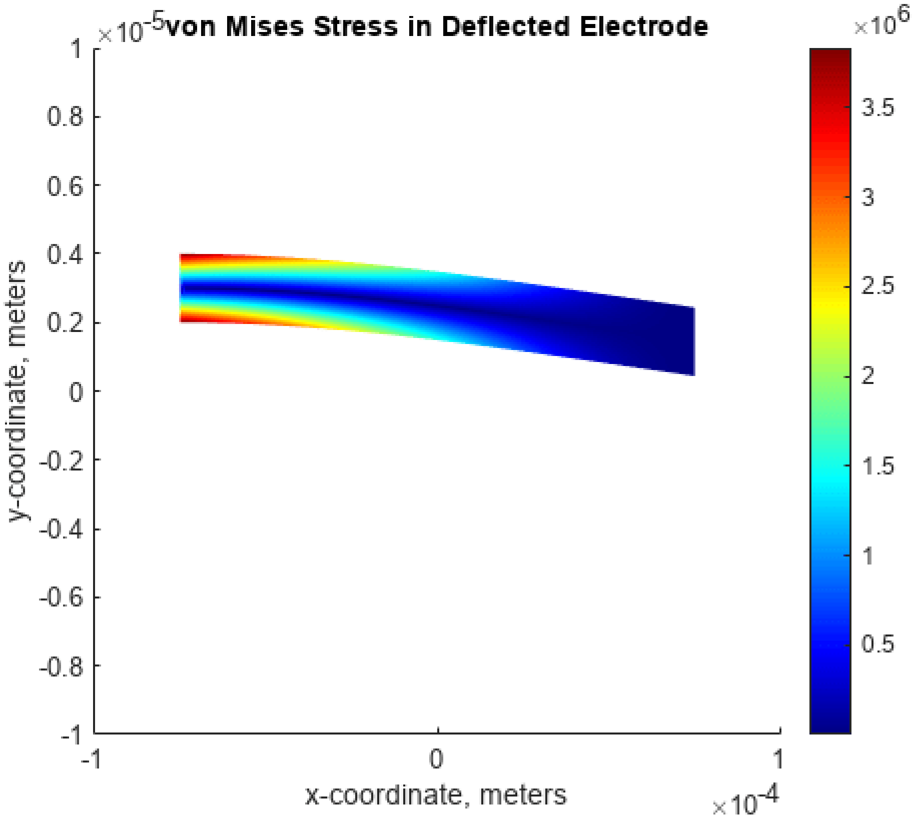


Solve the equations.

```
R = solve(structuralmodel);
```

Plot the displacement for the movable electrode.

```
pdeplot(structuralmodel,"XYData",R.VonMisesStress, ...
                        "Deformation",R.Displacement, ...
                        "DeformationScaleFactor",10, ...
                        "ColorMap","jet");

title("von Mises Stress in Deflected Electrode")
xlabel("x-coordinate, meters")
ylabel("y-coordinate, meters")
axis([-1e-4,1e-4,-1e-5,1e-5])
axis square
```



Find the maximal displacement.

```
maxdisp = max(abs(R.Displacement.uy));
fprintf('Finite element maximal tip deflection is: %12.4e\n', ...
         maxdisp);
```

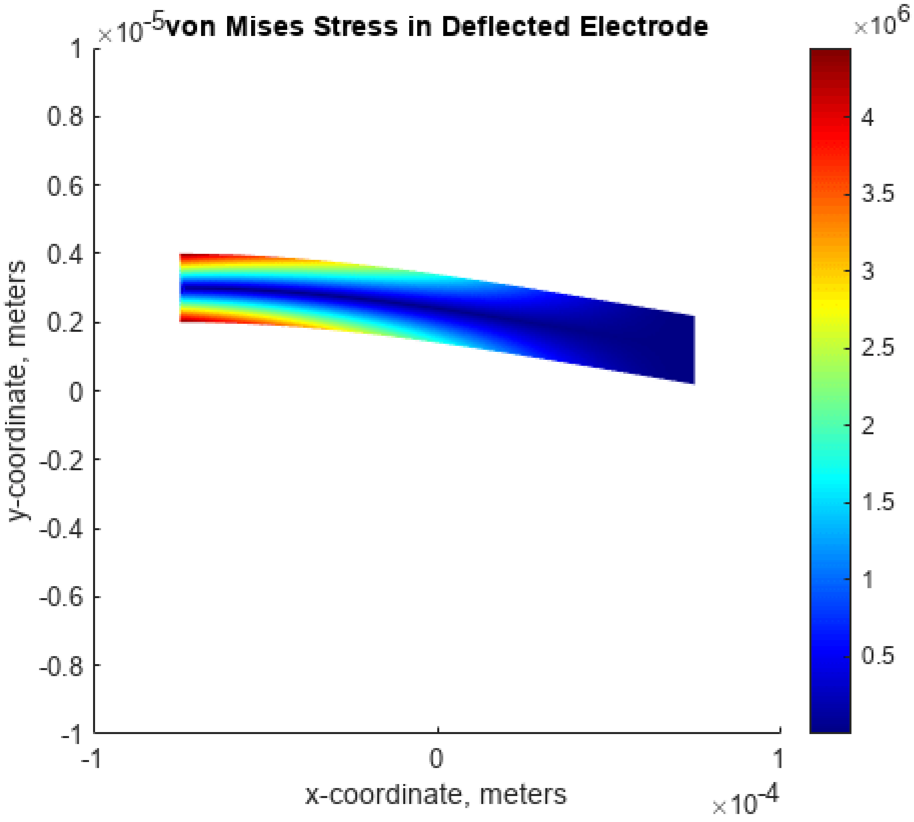
```
Finite element maximal tip deflection is:   1.5630e-07
```

Repeatedly update the charge density along the movable electrode and solve the model until the electrode deformation values converge.

```
olddisp = 0;
while abs((maxdisp-olddisp)/maxdisp) > 1e-10
% Impose boundary conditions
  pressureFcn = @(location,state) - ...
                CalculateElectrostaticPressure(results,R,location);
  bl = structuralBoundaryLoad(structuralmodel, ...
                              "Edge",[1,2,4], ...
                              "Pressure",pressureFcn, ...
                              "Vectorized","on");
% Solve the equations
    R = solve(structuralmodel);
    olddisp = maxdisp;
    maxdisp = max(abs(R.Displacement.uy));
    delete(bl)
end
```

Plot the displacement.

```
pdeplot(structuralmodel,"XYData",R.VonMisesStress, ...
                        "Deformation",R.Displacement, ...
                        "DeformationScaleFactor",10, ...
                        "ColorMap","jet");

title("von Mises Stress in Deflected Electrode")
xlabel("x-coordinate, meters")
ylabel("y-coordinate, meters")
axis([-1e-4,1e-4,-1e-5,1e-5])
axis square
```



Find the maximal displacement.

```
maxdisp = max(abs(R.Displacement.uy));
fprintf('Finite element maximal tip deflection is: %12.4e\n', maxdisp);
```

```
Finite element maximal tip deflection is:   1.8162e-07
```

References

[1] Sumant, P. S., N. R. Aluru, and A. C. Cangellaris. “A Methodology for Fast Finite Element Modeling of Electrostatically Actuated MEMS.” *International Journal for Numerical Methods in Engineering*. Vol 77, Number 13, 2009, 1789-1808.

Electrostatic Pressure Function

The electrostatic pressure on the movable electrode is given by

$$P = \frac{1}{2\epsilon}|D|^2,$$

where $|D| = \epsilon|E|$ is the magnitude of the electric flux density, ϵ is the electric permittivity next to the movable electrode, and $|E|$ is the magnitude of the electric field. The electric field E is the gradient of the electric potential V:

$$E = -\nabla V.$$

Solve the mechanical FEA to compute the deformation of the movable electrode. Using the calculated displacement of the movable electrode, update the charge density along the movable electrode.

$$|D_{\mathrm{def}}(x)| \approx |D_0(x)|\frac{G}{G - v(x)},$$

where $|D_{\mathrm{def}}(x)|$ is the magnitude of the electric flux density in the deformed electrode, $|D_0(x)|$ is the magnitude of the electric flux density in the undeformed electrode, G is the distance between the movable and fixed electrodes in the absence of actuation, and $v(x)$is the displacement of the movable electrode at position x along its axis. Initially, the movable electrode is undeformed, $v(x) = 0$, and therefore, $|D_{\mathrm{def}}(x)| \approx |D_0(x)|$.

```
function ePressure = ...
 CalculateElectrostaticPressure(elecResults,structResults,location)
% Function to compute electrostatic pressure.
% structuralBoundaryLoad is used to specify
% the pressure load on the movable electrode.
% Inputs:
% elecResults: Electrostatic FEA results
% structResults: Mechanical FEA results (optional)
% location: The x,y coordinate
% where pressure is obtained
%
% Output:
% ePressure: Electrostatic pressure at location
%
% location.x : The x-coordinate of the points
% location.y : The y-coordinate of the points
xq = location.x;
yq = location.y;

% Compute the magnitude of the electric field
% from the potential difference.
[gradx,grady] = evaluateGradient(elecResults,xq,yq);
absE = sqrt(gradx.^2 + grady.^2);

% The permittivity of vacuum is 8.854*10^-12 farad/meter.
epsilon0 = 8.854e-12;

% Compute the magnitude of the electric flux density.
absD0 = epsilon0*absE;
absD = absD0;

% If structResults (deformation) is available,
% update the charge density along the movable electrode.
if ~isempty(structResults)
 % Displacement of the movable electrode at position x
 intrpDisp = interpolateDisplacement(structResults,xq,yq);
```

```
 vdisp = abs(intrpDisp.uy);
 G = 2e-6; % Gap 2 micron
 absD = absD0.*G./(G-vdisp);
end

% Compute the electrostatic pressure.
ePressure = absD.^2/(2*epsilon0);

end
```

Deflection Analysis of Bracket

This example shows how to analyze a 3-D mechanical part under an applied load using finite element analysis (FEA) and determine the maximal deflection.

Create Structural Analysis Model

The first step in solving a linear elasticity problem is to create a structural analysis model. This model is a container that holds the geometry, structural material properties, damping parameters, body loads, boundary loads, boundary constraints, superelement interfaces, initial displacement and velocity, and mesh.

```
model = createpde("structural","static-solid");
```

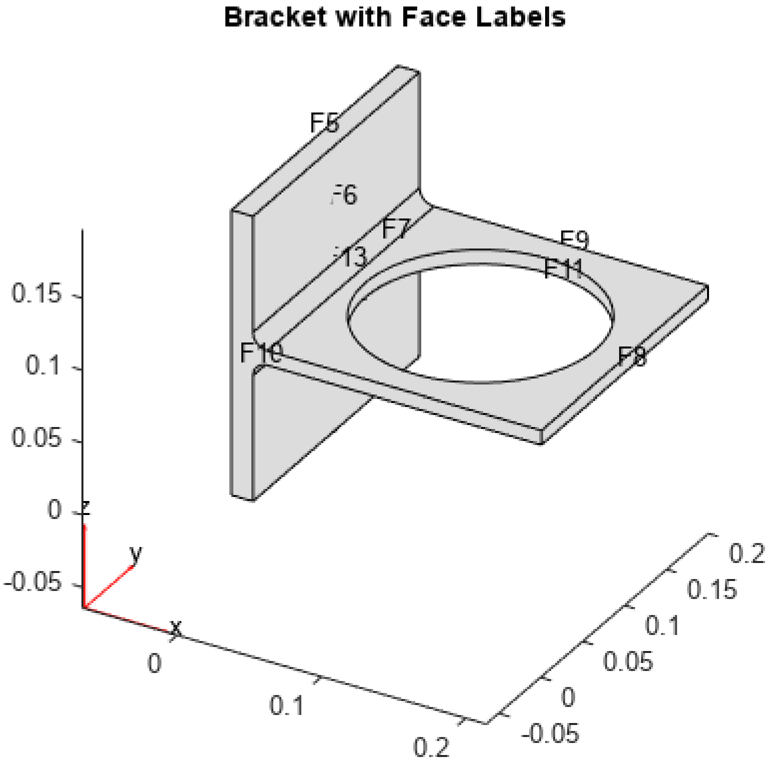
Import Geometry

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.

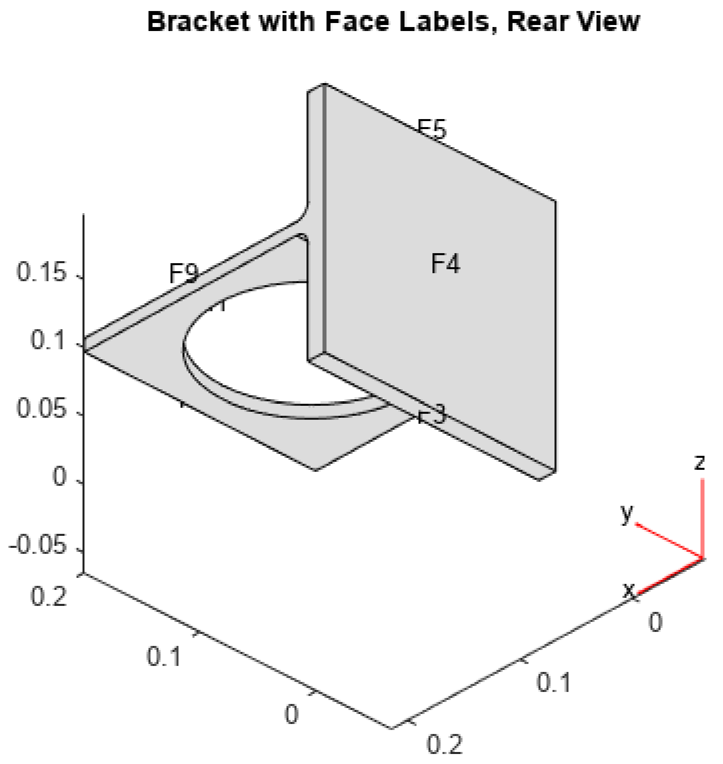
```
importGeometry(model,"BracketWithHole.stl");
```

Plot the geometry, displaying face labels.

```
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")
```



Specify Structural Properties of Material

Specify Young's modulus and Poisson's ratio of the material.

```
structuralProperties(model,"YoungsModulus",200e9, ...
                           "PoissonsRatio",0.3);
```

Apply Boundary Conditions and Loads

The problem has two boundary conditions: the back face (face 4) is fixed, and the front face (face 8) has an applied load. All other boundary conditions, by default, are free boundaries.

```
structuralBC(model,"Face",4,"Constraint","fixed");
```

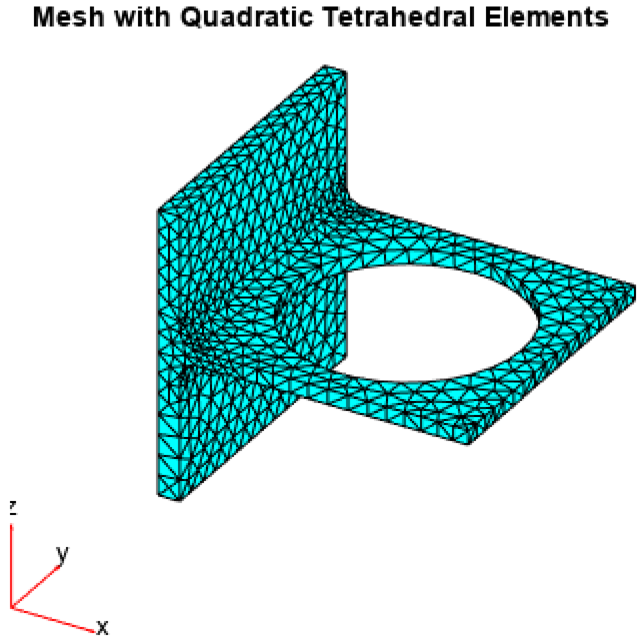
Apply a distributed load in the negative *z*-direction to the front face.

```
structuralBoundaryLoad(model,"Face",8,"SurfaceTraction",[0;0;-1e4]);
```

Generate Mesh

Generate and plot a mesh.

```
generateMesh(model);
figure
pdeplot3D(model)
title("Mesh with Quadratic Tetrahedral Elements");
```



Calculate Solution

Use the `solve` function to calculate the solution.

```
result = solve(model)
```

```
result = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [5993x1 double]
              Mesh: [1x1 FEMesh]
```

Examine Solution

Find the maximal deflection of the bracket in the *z*-direction.

```
minUz = min(result.Displacement.uz);
fprintf("Maximal deflection in the z-direction is %g meters.", minUz)
```

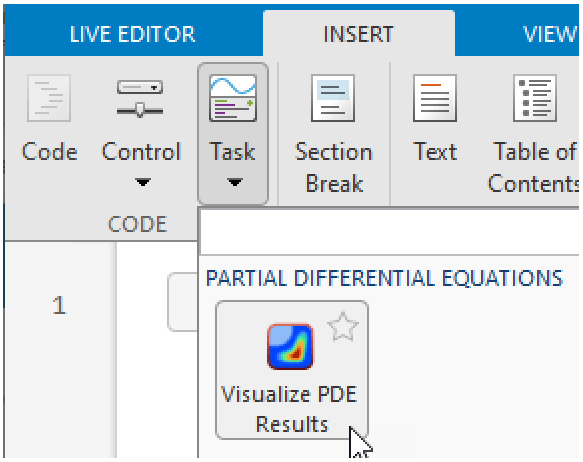
```
Maximal deflection in the z-direction is -4.43075e-05 meters.
```

Plot Results Using Visualize PDE Results Live Editor Task

Visualize the displacement components and the von Mises stress by using the **Visualize PDE Results** Live Editor task. The maximal deflections are in the *z*-direction. Because the bracket and the load are symmetric, the *x*-displacement and *z*-displacement are symmetric, and the *y*-displacement is antisymmetric with respect to the center line.

First, create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

On the **Live Editor** tab, select **Task** > **Visualize PDE Results**. This action inserts the task into your script.



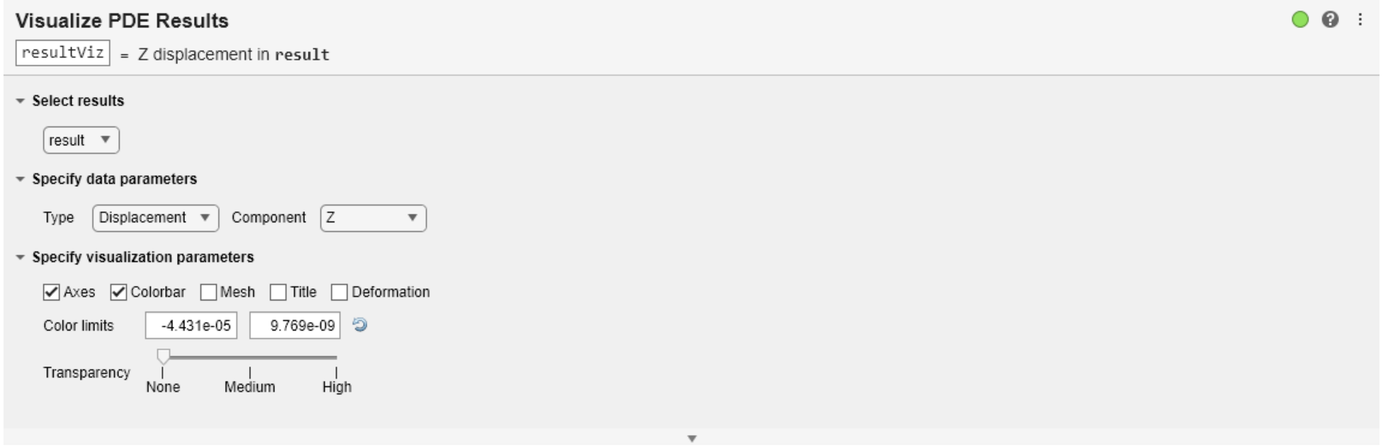
To plot the *z*-displacement, follow these steps. To plot the *x*- and *y*-displacements, follow the same steps, but set **Component** to *X* and *Y*, respectively.

1 In the **Select results** section of the task, select `result` from the drop-down list.

2 In the **Specify data parameters** section of the task, set **Type** to *Displacement* and **Component** to *Z*.

3 In the **Specify visualization parameters** section of the task, clear the **Deformation** check box.

Here, the blue color represents the lowest displacement value, and the red color represents the highest displacement value. The bracket load causes face 8 to dip down, so the maximum *z*-displacement appears blue.



Visualize PDE Results

`resultViz` = Z displacement in `result`

▾ Select results

result ▾

▾ Specify data parameters

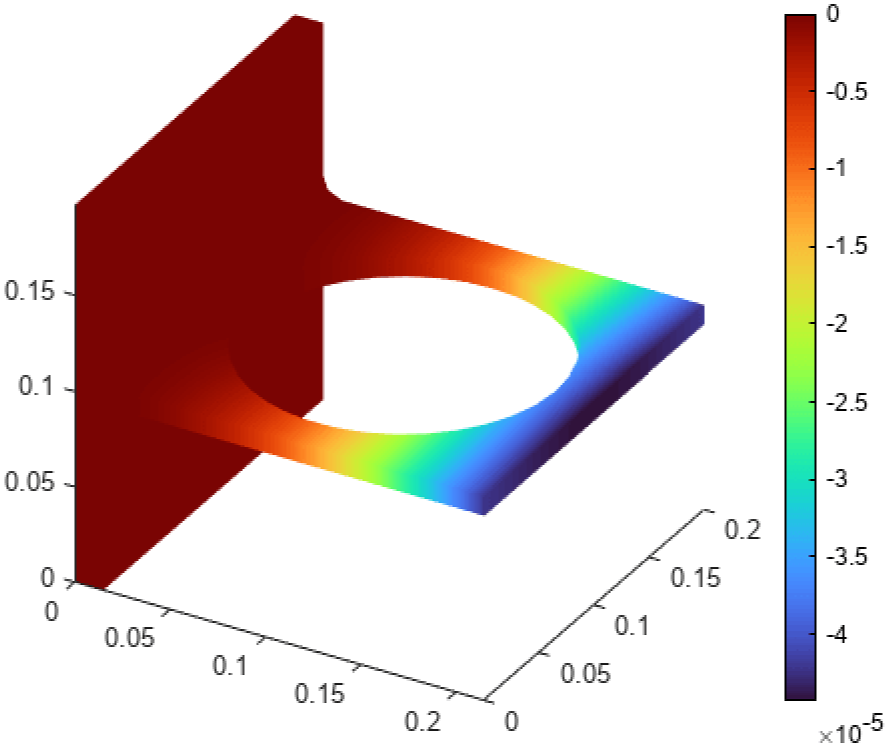
Type: Displacement ▾ Component: Z ▾

▾ Specify visualization parameters

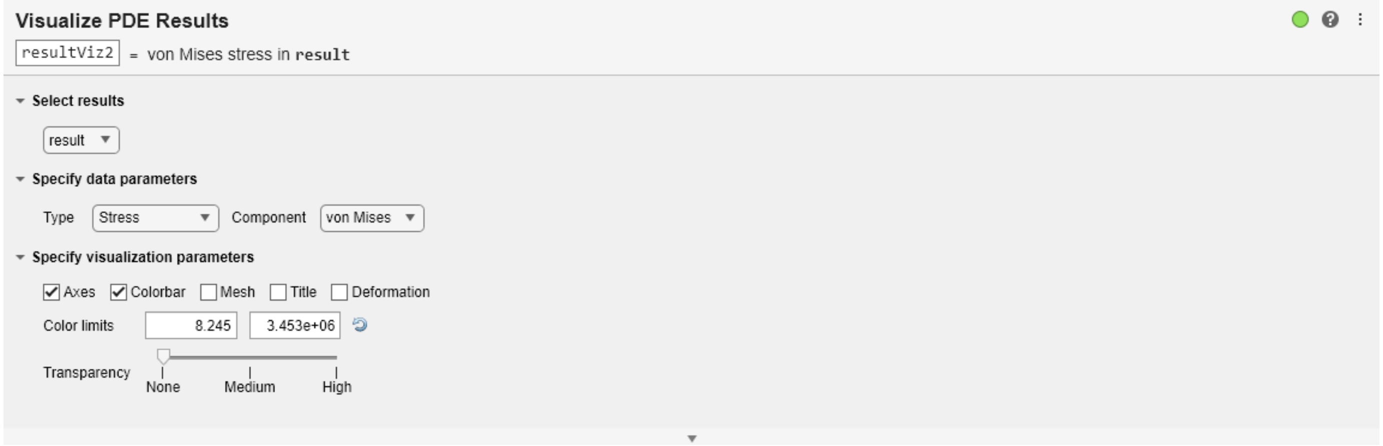
☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☐ Deformation

Color limits: -4.431e-05 9.769e-09

Transparency: None Medium High



To plot the von Mises stress, in the **Specify data parameters** section of the task, set **Type** to *Stress* and **Component** to *von Mises*.



Visualize PDE Results

`resultViz2` = von Mises stress in **result**

Select results

result

Specify data parameters

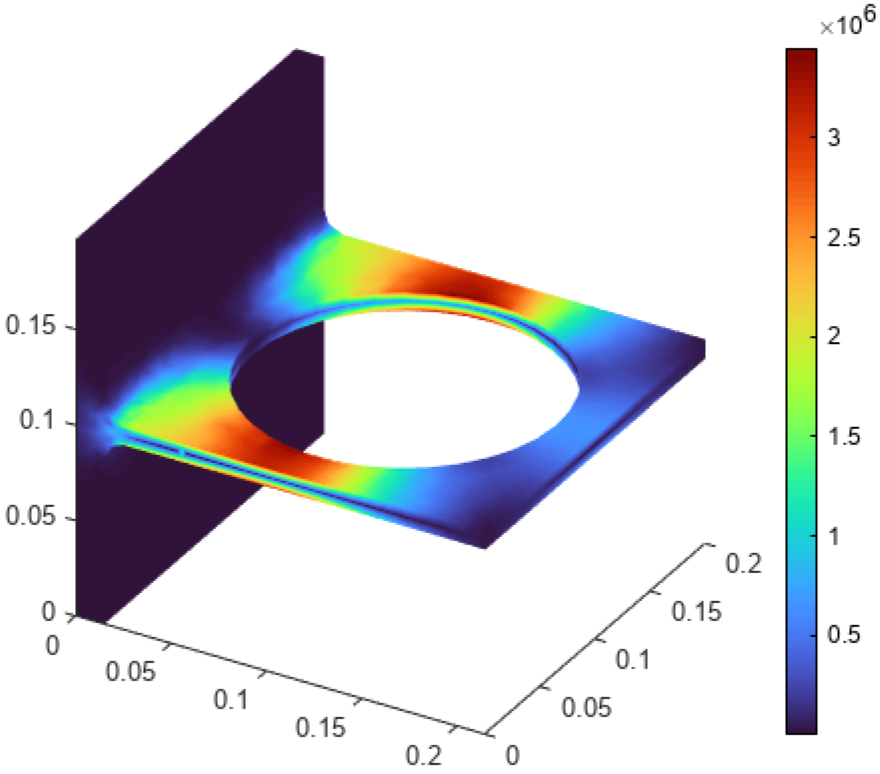
Type: Stress Component: von Mises

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☐ Deformation

Color limits: 8.245 3.453e+06

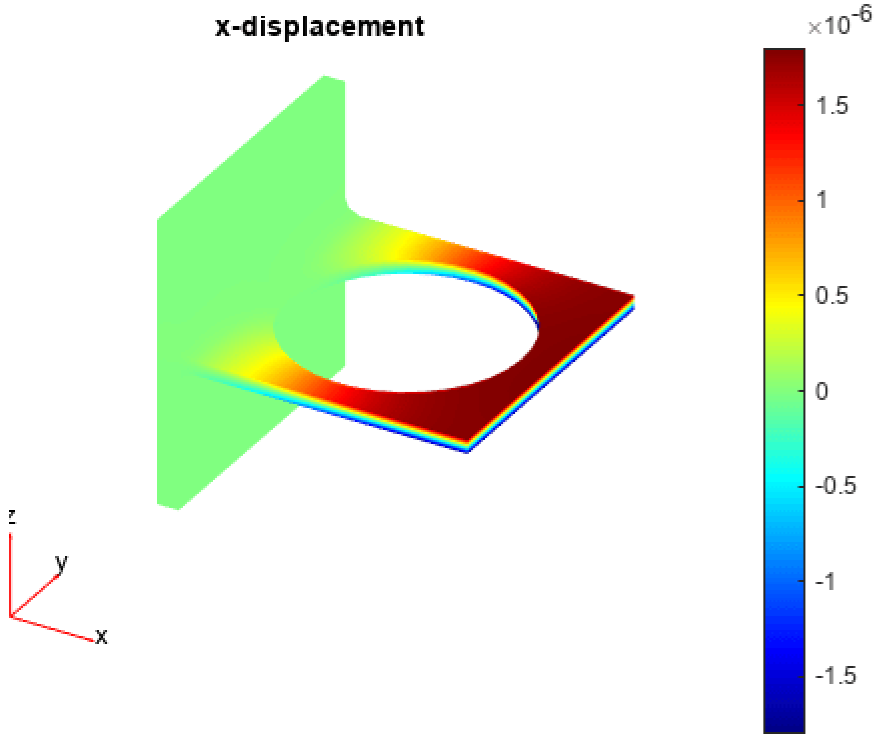
Transparency: None (Medium, High)



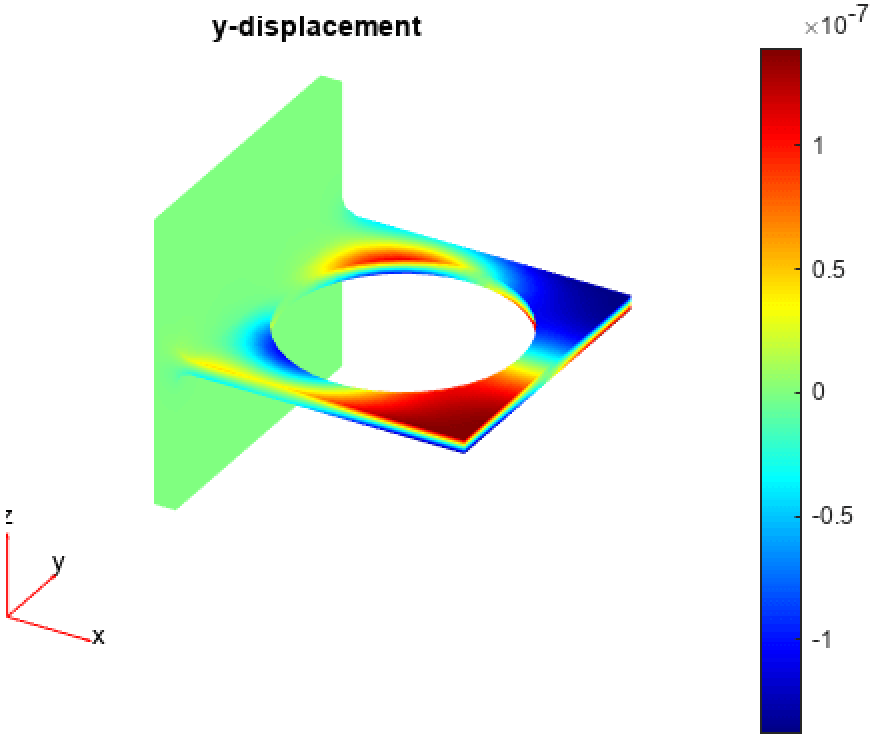
Plot Results at the Command Line

You also can plot the results, such as the displacement components and the von Mises stress, at the MATLAB® command line by using the `pdeplot3D` function.

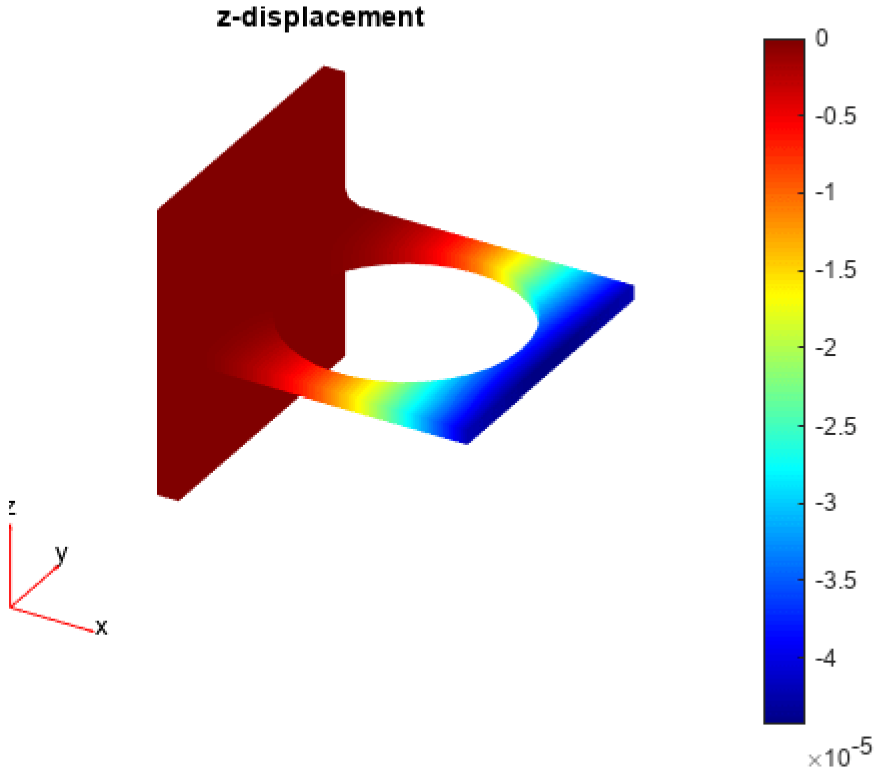
```
figure
pdeplot3D(model,"ColorMapData",result.Displacement.ux)
title("x-displacement")
colormap("jet")
```



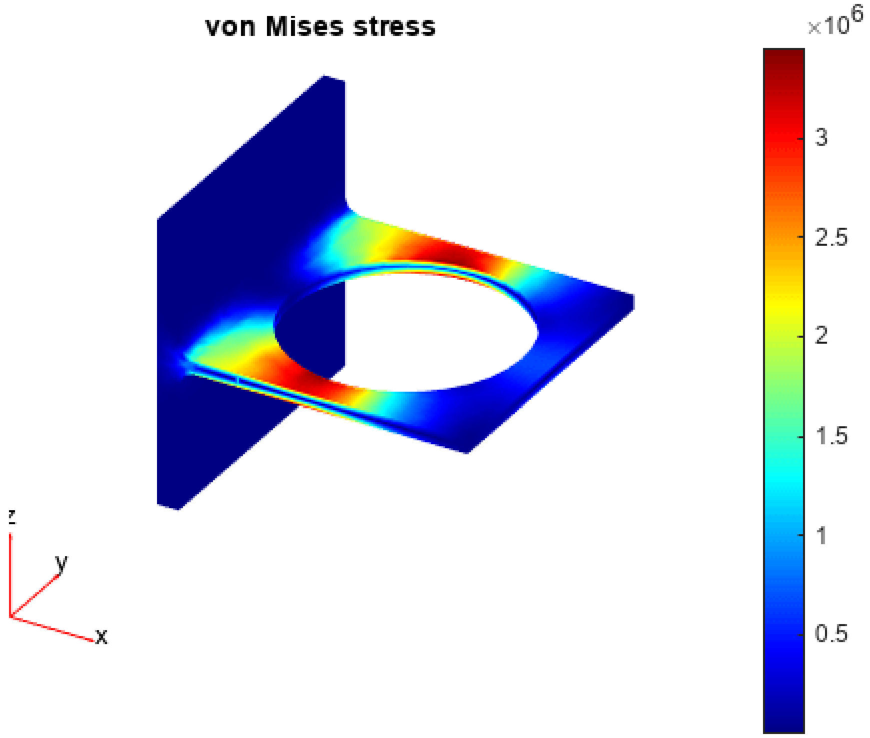
```
figure
pdeplot3D(model,"ColorMapData",result.Displacement.uy)
title("y-displacement")
colormap("jet")
```



```
figure
pdeplot3D(model,"ColorMapData",result.Displacement.uz)
title("z-displacement")
colormap("jet")
```



```
figure
pdeplot3D(model,"ColorMapData",result.VonMisesStress)
title("von Mises stress")
colormap("jet")
```


von Mises stress
×10^6
3
2.5
2
1.5
1
0.5
z
y
x

Vibration of 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 (See Reference).

First, create a structural model container for your 3-D modal analysis problem. This is a container that holds the geometry, properties of the material, body loads, boundary loads, boundary constraints, and mesh.

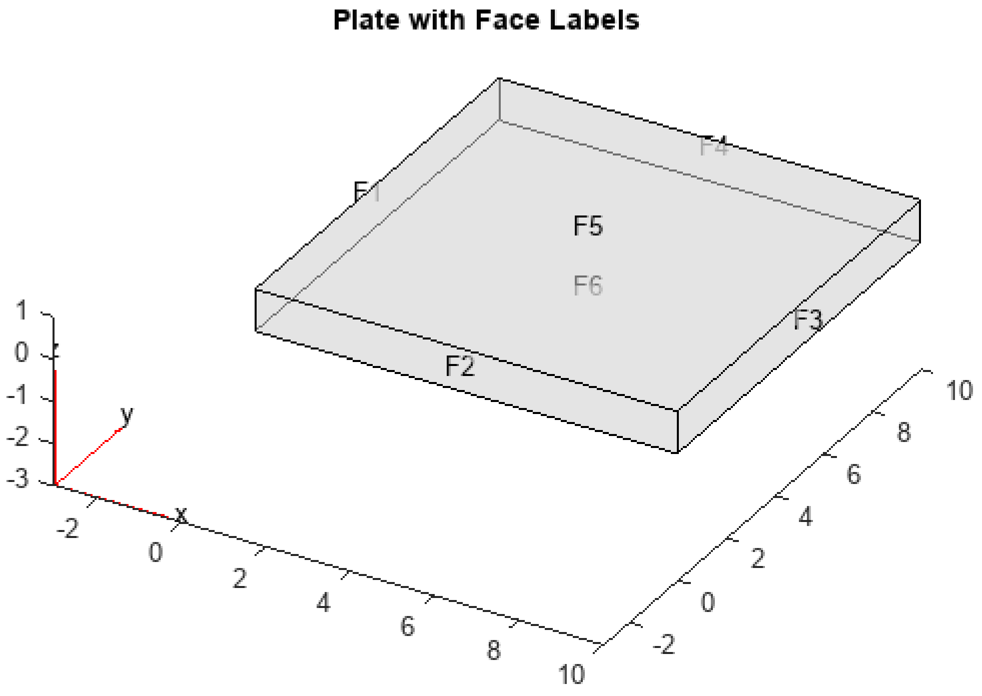
```
model = createpde("structural","modal-solid");
```

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 need the face labels when defining the boundary conditions.

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



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

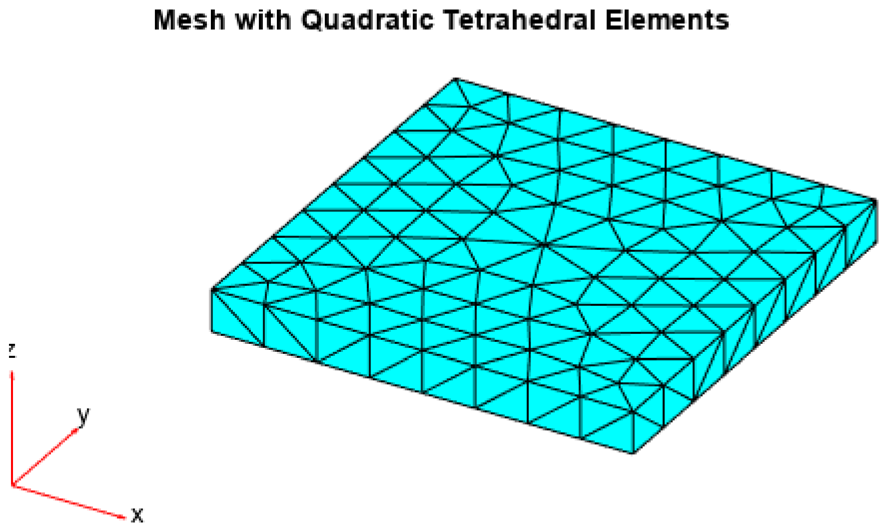
```
structuralProperties(model,"YoungsModulus",200e9, ...
                           "PoissonsRatio",0.3, ...
                           "MassDensity",8000);
```

In this example, the only boundary condition is the zero *z*-displacement on the four edge faces. These edge faces have labels 1 through 4.

```
structuralBC(model,"Face",1:4,"ZDisplacement",0);
```

Create and plot a mesh. Specify the target minimum edge length so that there is one row of elements per plate thickness.

```
generateMesh(model,"Hmin",1.3);
figure
pdeplot3D(model);
title("Mesh with Quadratic Tetrahedral Elements");
```


Mesh with Quadratic Tetrahedral Elements

For comparison with the published values, load the reference frequencies in Hz.

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

Solve the problem for the specified frequency range. Define the upper limit as slightly larger than the highest reference frequency and the lower limit as slightly smaller than the lowest reference frequency.

```
maxFreq = 1.1*refFreqHz(end)*2*pi;
result = solve(model,"FrequencyRange",[-0.1 maxFreq]);
```

Calculate frequencies in Hz.

```
freqHz = result.NaturalFrequencies/(2*pi);
```

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

```
tfreqHz = table(refFreqHz.',freqHz(1:10));
tfreqHz.Properties.VariableNames = {'Reference','Computed'};
disp(tfreqHz);
```

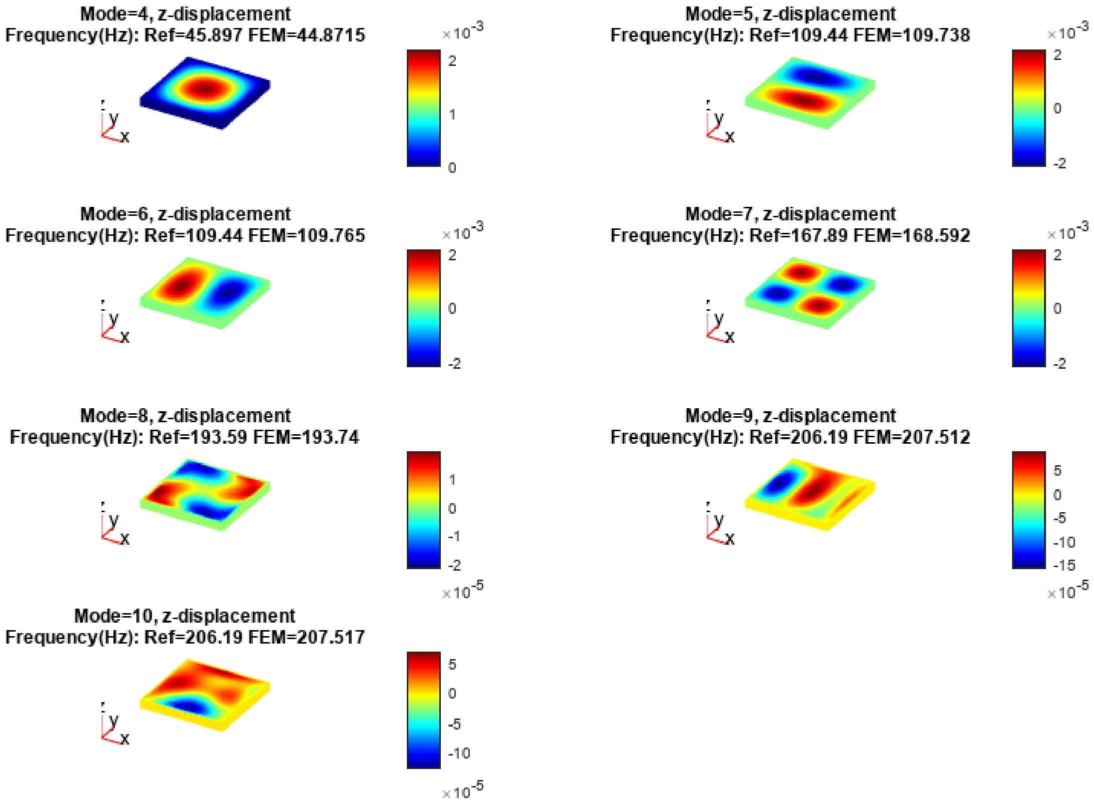
```
    Reference     Computed 
    _________    __________

         0                0
         0        3.878e-05
         0       4.0126e-05
```

```
    45.897           44.871
    109.44           109.74
    109.44           109.77
    167.89           168.59
    193.59           193.74
    206.19           207.51
    206.19           207.52
```

You see good agreement between the computed and published frequencies.

Plot the third component (*z*-component) of the solution for the seven lowest nonzero-frequency modes.

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



Reference

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

Structural Dynamics of Tuning Fork

Perform modal and transient analysis of a tuning fork.

A tuning fork is a U-shaped beam. When struck on one of its prongs or tines, it vibrates at its fundamental (first) frequency and produces an audible sound.

The first flexible mode of a tuning fork is characterized by symmetric vibration of the tines: they move towards and away from each other simultaneously, balancing the forces at the base where they intersect. The fundamental mode of vibration does not produce any bending effect on the handle attached at the intersection of tines. The lack of bending at the base enables easy handling of tuning fork without influencing its dynamics.

Transverse vibration of the tines causes the handle to vibrate axially at the fundamental frequency. This axial vibration can be used to amplify the audible sound by bringing the end of the handle in contact with a larger surface area, like a metal table top. The next higher mode with symmetric mode shape is about 6.25 times the fundamental frequency. Therefore, a properly excited tuning fork tends to vibrate with a dominant frequency corresponding to fundamental frequency, producing a pure audible tone. This example simulates these aspects of the tuning fork dynamics by performing a modal analysis and a transient dynamics simulation.

You can find the helper functions `animateSixTuningForkModes` and `tuningForkFFT` and the geometry file `TuningFork.stl` under `matlab/R20XXx/examples/pde/main`.

Modal Analysis of Tuning Fork

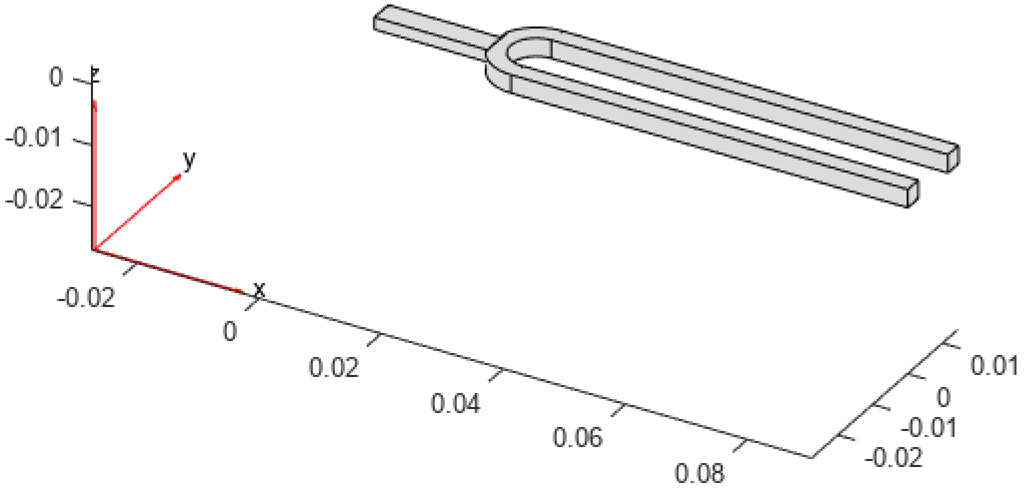
Find natural frequencies and mode shapes for the fundamental mode of a tuning fork and the next several modes. Show the lack of bending effect on the fork handle at the fundamental frequency.

First, create a structural model for modal analysis of a solid tuning fork.

```
model = createpde("structural","modal-solid");
```

To perform unconstrained modal analysis of a structure, it is enough to specify geometry, mesh, and material properties. First, import and plot the tuning fork geometry.

```
importGeometry(model,"TuningFork.stl");
figure
pdegplot(model)
```



Specify Young's modulus, Poisson's ratio, and the mass density to model linear elastic material behavior. Specify all physical properties in consistent units.

```
E = 210E9;
nu = 0.3;
rho = 8000;
structuralProperties(model,"YoungsModulus",E, ...
                           "PoissonsRatio",nu, ...
                           "MassDensity",rho);
```

Generate a mesh.

```
generateMesh(model,"Hmax",0.001);
```

Solve the model for a chosen frequency range. Specify the lower frequency limit below zero so that all modes with frequencies near zero appear in the solution.

```
RF = solve(model,"FrequencyRange",[-1,4000]*2*pi);
```

By default, the solver returns circular frequencies.

```
modeID = 1:numel(RF.NaturalFrequencies);
```

Express the resulting frequencies in Hz by dividing them by 2π. Display the frequencies in a table.

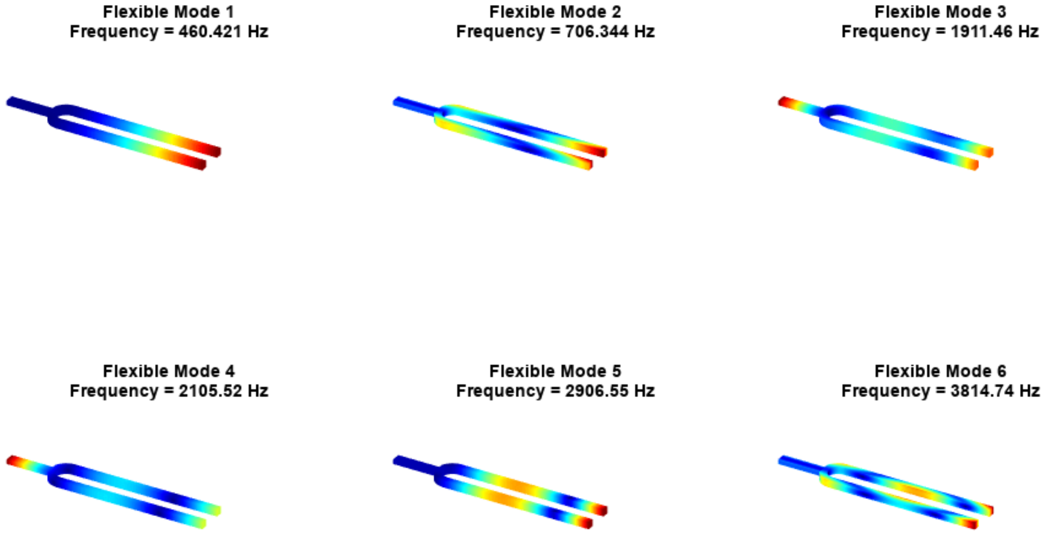
```
tmodalResults = table(modeID.',RF.NaturalFrequencies/2/pi);
tmodalResults.Properties.VariableNames = {'Mode','Frequency'};
disp(tmodalResults);
```

```
    Mode    Frequency
    ____    _________

      1     0.0090573
      2      0.007208
      3     0.0058295
      4     0.0054257
      5     0.0049971
      6     0.0084112
      7        460.42
      8        706.34
      9        1911.5
     10        2105.5
     11        2906.5
     12        3814.7
```

Because there are no boundary constraints in this example, modal results include the rigid body modes. The first six near-zero frequencies indicate the six rigid body modes of a 3-D solid body. The first flexible mode is the seventh mode with a frequency around 460 Hz.

The best way to visualize mode shapes is to animate the harmonic motion at their respective frequencies. The `animateSixTuningForkModes` function animates the six flexible modes, which are modes 7 through 12 in the modal results `RF`.

```
frames  = animateSixTuningForkModes(RF);
```



To play the animation, use the following command:

```
movie(figure("units","normalized","outerposition",[0 0 1 1]),frames,5,30)
```

In the first mode, two oscillating tines of the tuning fork balance out transverse forces at the handle. The next mode with this effect is the fifth flexible mode with the frequency 2906.5 Hz. This frequency is about 6.25 times greater than the fundamental frequency 460 Hz.

Transient Analysis of Tuning Fork

Simulate the dynamics of a tuning fork being gently and quickly struck on one of its tines. Analyze vibration of tines over time and axial vibration of the handle.

First, create a structural transient analysis model.

```
tmodel = createpde("structural","transient-solid");
```

Import the same tuning fork geometry you used for the modal analysis.

```
importGeometry(tmodel,"TuningFork.stl");
```

Generate a mesh.

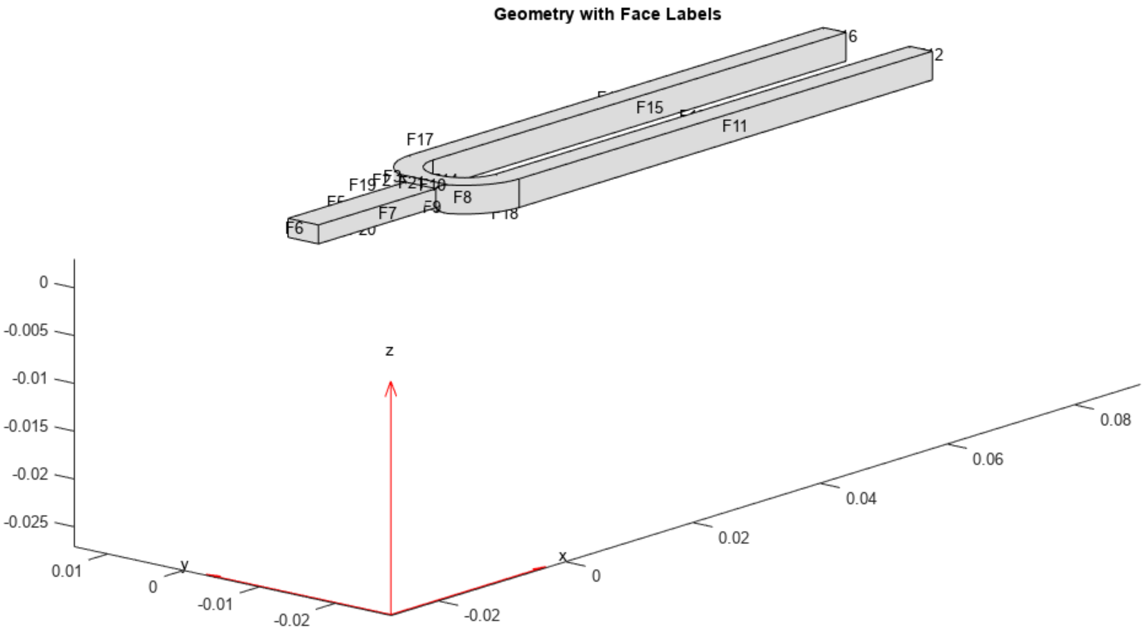
```
mesh = generateMesh(tmodel,"Hmax",0.005);
```

Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(tmodel,"YoungsModulus",E, ...
                            "PoissonsRatio",nu, ...
                            "MassDensity",rho);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face labels.

```
figure("units","normalized","outerposition",[0 0 1 1])
pdegplot(tmodel,"FaceLabels","on")
view(-50,15)
title("Geometry with Face Labels")
```



Impose sufficient boundary constraints to prevent rigid body motion under applied loading. Typically, you hold a tuning fork by hand or mount it on a table. A simplified approximation to this boundary condition is fixing a region near the intersection of tines and the handle (faces 21 and 22).

```
structuralBC(tmodel,"Face",[21,22],"Constraint","fixed");
```

Approximate an impulse loading on a face of a tine by applying a pressure load for a very small fraction of the time period of the fundamental mode. By using this very short pressure pulse, you ensure that only the fundamental mode of a tuning fork is excited. To evaluate the time period `T` of the fundamental mode, use the results of modal analysis.

```
T = 2*pi/RF.NaturalFrequencies(7);
```

Specify the pressure loading on a tine as a short rectangular pressure pulse.

```
structuralBoundaryLoad(tmodel,"Face",11,"Pressure",5E6,"EndTime",T/300);
```

Apply zero displacement and velocity as initial conditions.

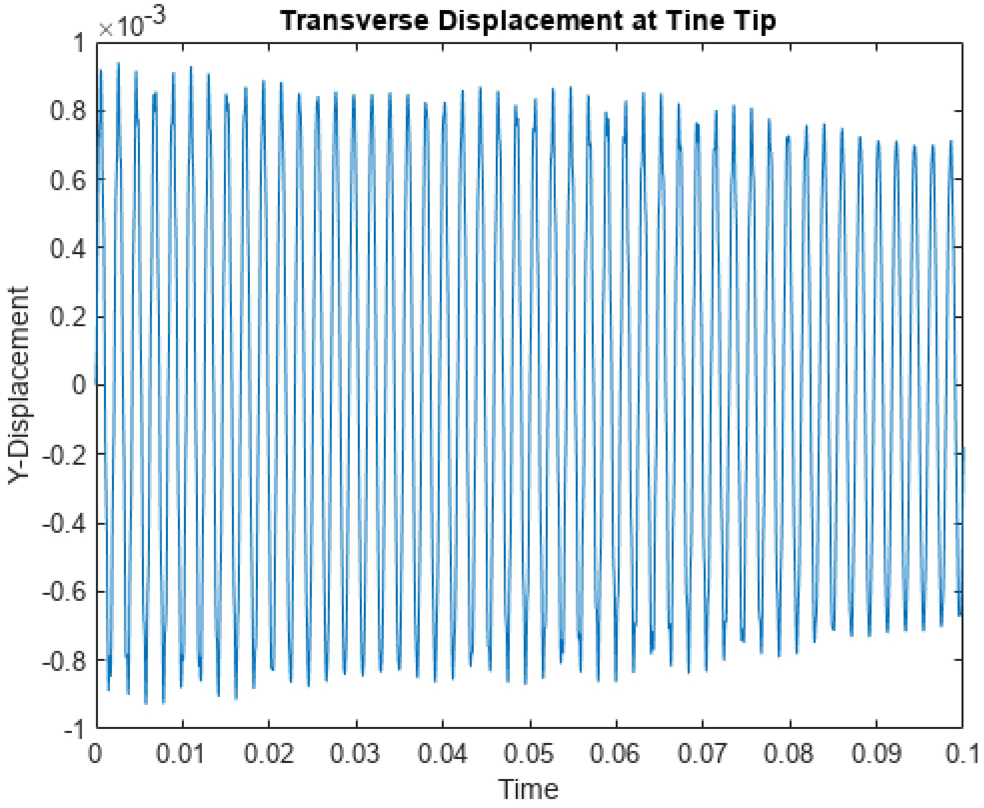
```
structuralIC(tmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the transient model for 50 periods of the fundamental mode. Sample the dynamics 60 times per period of the fundamental mode.

```
ncycle = 50;
samplingFrequency = 60/T;
tlist = linspace(0,ncycle*T,ncycle*T*samplingFrequency);
R = solve(tmodel,tlist);
```

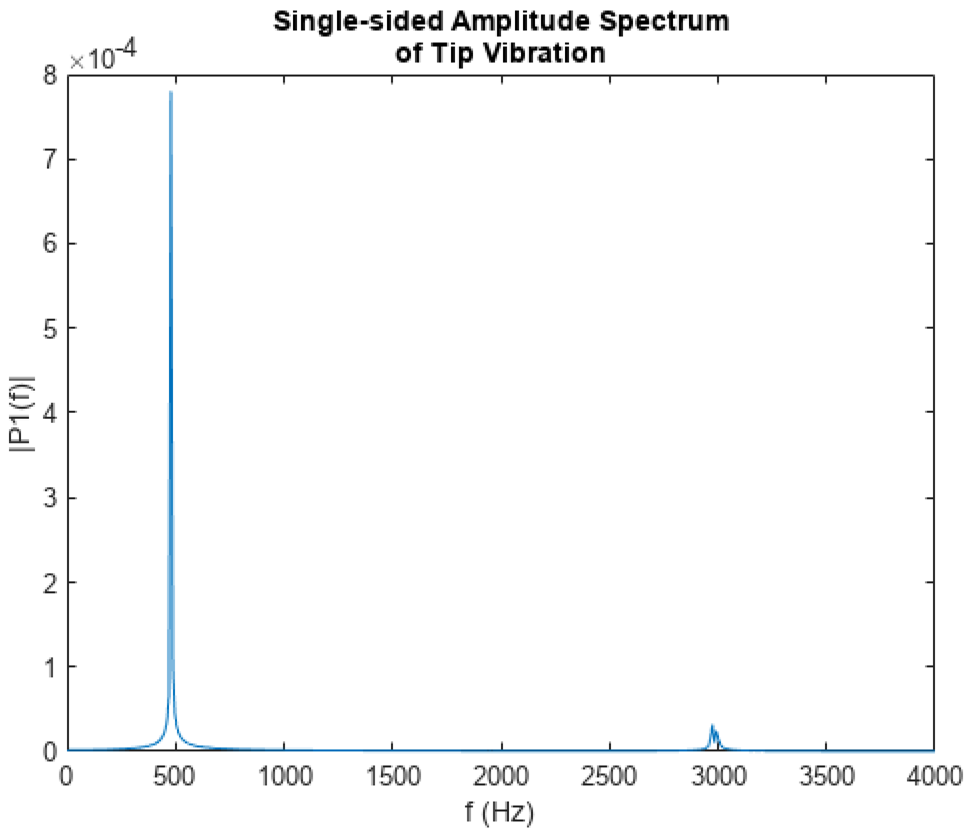
Plot the time-series of the vibration of the tine tip, which is face 12. Find nodes on the tip face and plot the y-component of the displacement over time, using one of these nodes.

```
excitedTineTipNodes = findNodes(mesh,"region","Face",12);
tipDisp = R.Displacement.uy(excitedTineTipNodes(1),:);

figure
plot(R.SolutionTimes,tipDisp)
title("Transverse Displacement at Tine Tip")
xlim([0,0.1])
xlabel("Time")
ylabel("Y-Displacement")
```



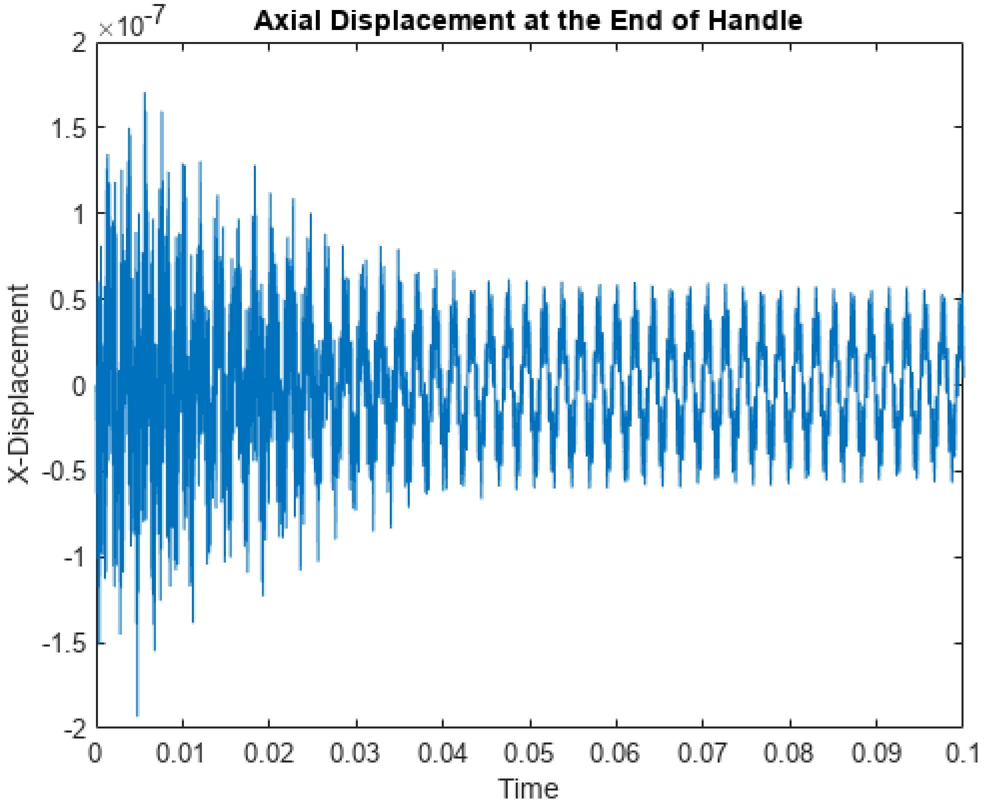
Perform fast Fourier transform (FFT) on the tip displacement time-series to see that the vibration frequency of the tuning fork is close to its fundamental frequency. A small deviation from the fundamental frequency computed in an unconstrained modal analysis appears because of constraints imposed in the transient analysis.

```
[fTip,PTip] = tuningForkFFT(tipDisp,samplingFrequency);
figure
plot(fTip,PTip)
title({'Single-sided Amplitude Spectrum', 'of Tip Vibration'})
xlabel("f (Hz)")
ylabel("|P1(f)|")
xlim([0,4000])
```



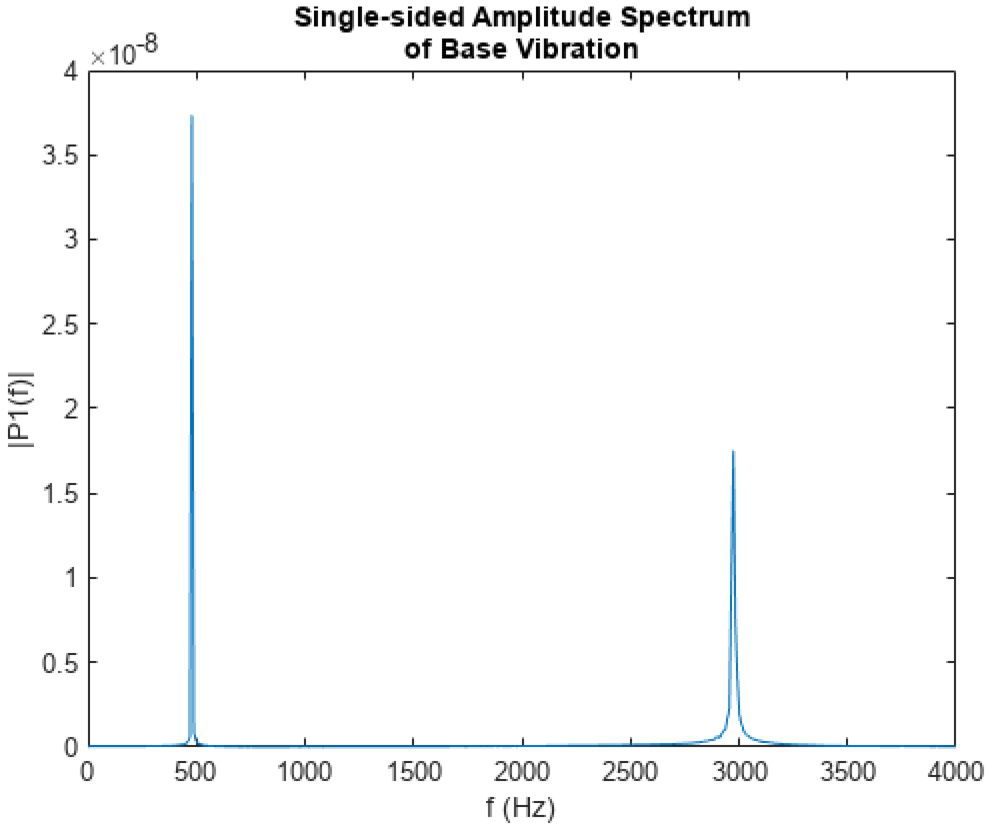
Transverse vibration of tines causes the handle to vibrate axially with the same frequency. To observe this vibration, plot the axial displacement time-series of the end face of the handle.

```
baseNodes = tmodel.Mesh.findNodes("region","Face",6);
baseDisp = R.Displacement.ux(baseNodes(1),:);
figure
plot(R.SolutionTimes,baseDisp)
title("Axial Displacement at the End of Handle")
xlim([0,0.1])
ylabel("X-Displacement")
xlabel("Time")
```



Perform an FFT of the time-series of the axial vibration of the handle. This vibration frequency is also close to its fundamental frequency.

```
[fBase,PBase] = tuningForkFFT(baseDisp,samplingFrequency);
figure
plot(fBase,PBase)
title({'Single-sided Amplitude Spectrum', 'of Base Vibration'})
xlabel("f (Hz)")
ylabel("|P1(f)|")
xlim([0,4000])
```


Single-sided Amplitude Spectrum
of Base Vibration
×10^-8
4
3.5
3
2.5
2
1.5
1
0.5
0
|P1(f)|
0
500
1000
1500
2000
2500
3000
3500
4000
f (Hz)

Modal Superposition Method for Structural Dynamics Problem

This example shows how to solve a structural dynamics problem by using modal analysis results. Solve for the transient response at the center of a 3-D beam under a harmonic load on one of its corners. Compare the direct integration results with the results obtained by modal superposition.

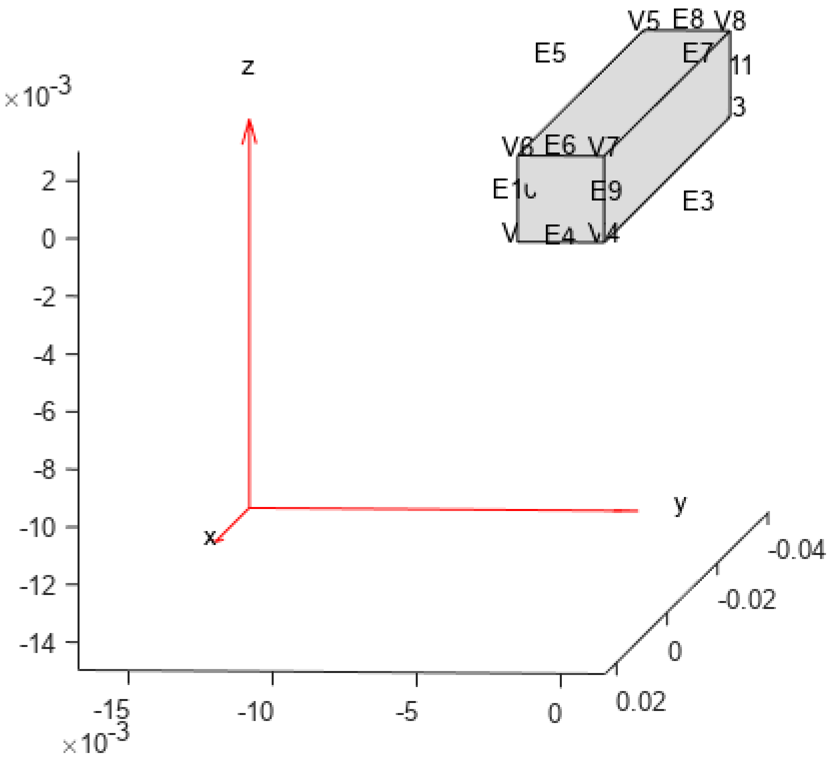
Modal Analysis

Create a modal analysis model for a 3-D problem.

```
modelM = createpde("structural","modal-solid");
```

Create the geometry and include it in the model. Plot the geometry and display the edge and vertex labels.

```
gm = multicuboid(0.05,0.003,0.003);
modelM.Geometry=gm;
pdegplot(modelM,"EdgeLabels","on","VertexLabels","on");
view([95 5])
```



Generate a mesh.

```
msh = generateMesh(modelM);
```

Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelM,"YoungsModulus",210E9, ...
                            "PoissonsRatio",0.3, ...
                            "MassDensity",7800);
```

Specify minimal constraints on one end of the beam to prevent rigid body modes. For example, specify that edge 4 and vertex 7 are fixed boundaries.

```
structuralBC(modelM,"Edge",4,"Constraint","fixed");
structuralBC(modelM,"Vertex",7,"Constraint","fixed");
```

Solve the problem for the frequency range from 0 to 500000. The recommended approach is to use a value that is slightly smaller than the expected lowest frequency. Thus, use -0.1 instead of 0.

```
Rm = solve(modelM,"FrequencyRange",[-0.1,500000]);
```

Transient Analysis

Create a transient analysis model for a 3-D problem.

```
modelD = createpde("structural","transient-solid");
```

Use the same geometry and mesh as for the modal analysis.

```
modelD.Geometry = gm;
modelD.Mesh = msh;
```

Specify the same values for Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelD,"YoungsModulus",210E9, ...
                            "PoissonsRatio",0.3, ...
                            "MassDensity",7800);
```

Specify the same minimal constraints on one end of the beam to prevent rigid body modes.

```
structuralBC(modelD,"Edge",4,"Constraint","fixed");
structuralBC(modelD,"Vertex",7,"Constraint","fixed");
```

Apply a sinusoidal force on the corner opposite the constrained edge and vertex.

```
structuralBoundaryLoad(modelD,"Vertex",5, ...
                              "Force",[0,0,10], ...
                              "Frequency",7600);
```

Specify the zero initial displacement and velocity.

```
structuralIC(modelD,"Velocity",[0;0;0],"Displacement",[0;0;0]);
```

Specify the relative and absolute tolerances for the solver.

```
modelD.SolverOptions.RelativeTolerance = 1E-5;
modelD.SolverOptions.AbsoluteTolerance = 1E-9;
```

Solve the model using the default direct integration method.

```
tlist = linspace(0,0.004,120);
Rd = solve(modelD,tlist);
```

Now, solve the model using the modal results.

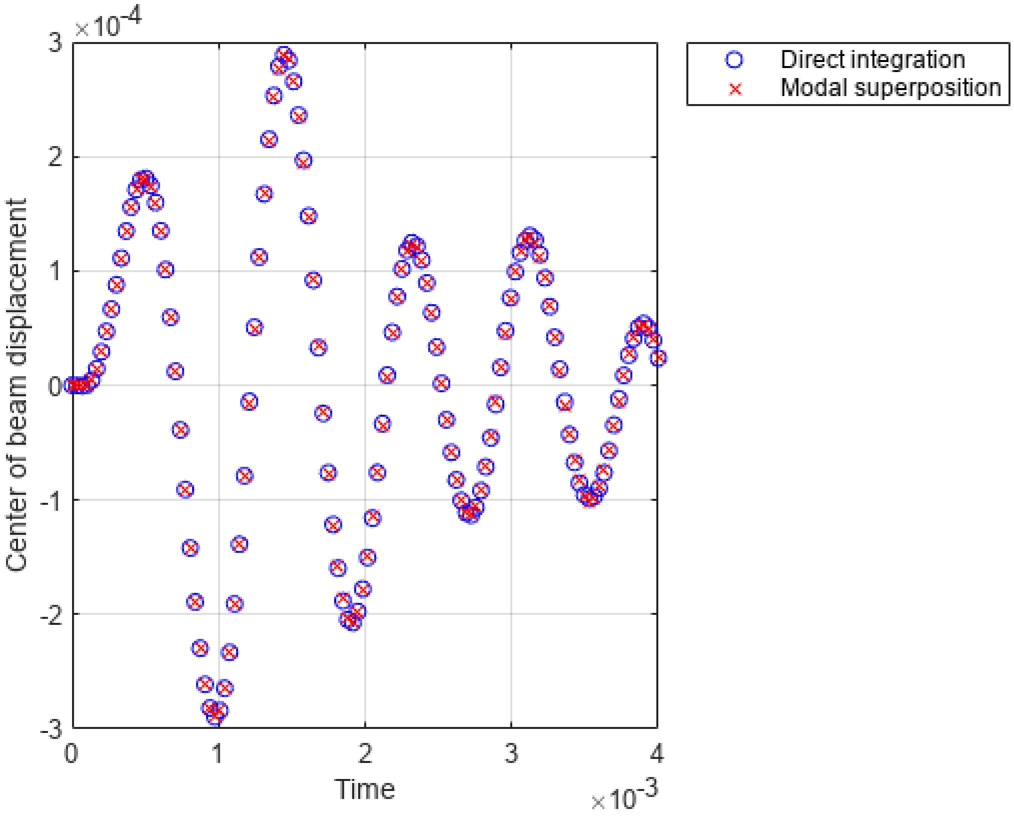
```
tlist = linspace(0,0.004,120);
Rdm = solve(modelD,tlist,"ModalResults",Rm);
```

Interpolate the displacement at the center of the beam.

```
intrpUd = interpolateDisplacement(Rd,0,0,0.0015);
intrpUdm = interpolateDisplacement(Rdm,0,0,0.0015);
```

Compare the direct integration results with the results obtained by modal superposition.

```
plot(Rd.SolutionTimes,intrpUd.uz,"bo")
hold on
plot(Rdm.SolutionTimes,intrpUdm.uz,"rx")
grid on
legend("Direct integration", "Modal superposition")
xlabel("Time");
ylabel("Center of beam displacement")
```



Stress Concentration in Plate with Circular Hole

Perform a 2-D plane-stress elasticity analysis.

A thin rectangular plate under a uniaxial tension has a uniform stress distribution. Introducing a circular hole in the plate disturbs the uniform stress distribution near the hole, resulting in a significantly higher than average stress. Such a thin plate, subject to in-plane loading, can be analyzed as a 2-D plane-stress elasticity problem. In theory, if the plate is infinite, then the stress near the hole is three times higher than the average stress. For a rectangular plate of finite width, the stress concentration factor is a function of the ratio of hole diameter to the plate width. This example approximates the stress concentration factor using a plate of a finite width.

Create Structural Model and Include Geometry

Create a structural model for static plane-stress analysis.

```
model = createpde("structural","static-planestress");
```

The plate must be sufficiently long, so that the applied loads and boundary conditions are far from the circular hole. This condition ensures that a state of uniform tension prevails in the far field and, therefore, approximates an infinitely long plate. In this example the length of the plate is four times greater than its width. Specify the following geometric parameters of the problem.

```
radius = 20.0;
width = 50.0;
totalLength = 4*width;
```

Define the geometry description matrix (GDM) for the rectangle and circle.

```
R1 = [3 4 -totalLength  totalLength ...
           totalLength -totalLength ...
          -width -width width width]';
C1 = [1 0 0 radius 0 0 0 0 0 0]';
```

Define the combined GDM, name-space matrix, and set formula to construct decomposed geometry using `decsg`.

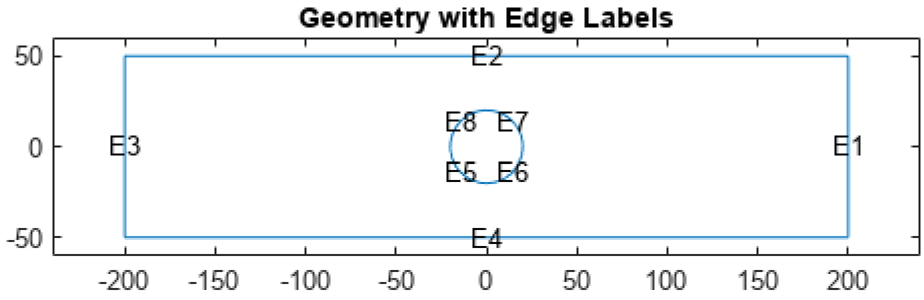
```
gdm = [R1 C1];
ns = char('R1','C1');
g = decsg(gdm,'R1 - C1',ns');
```

Create the geometry and include it into the structural model.

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

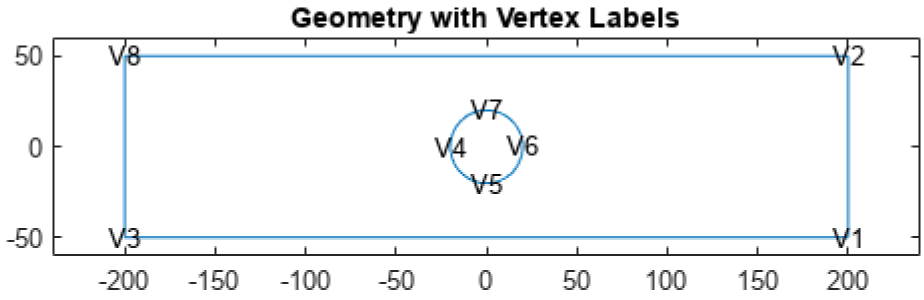
Plot the geometry displaying edge labels.

```
figure
pdegplot(model,"EdgeLabel","on");
axis([-1.2*totalLength 1.2*totalLength -1.2*width 1.2*width])
title("Geometry with Edge Labels")
```



Plot the geometry displaying vertex labels.

```
figure
pdegplot(model,"VertexLabels","on");
axis([-1.2*totalLength 1.2*totalLength -1.2*width 1.2*width])
title("Geometry with Vertex Labels")
```



Specify Model Parameters

Specify Young's modulus and Poisson's ratio to model linear elastic material behavior. Remember to specify physical properties in consistent units.

```
structuralProperties(model,"YoungsModulus",200E3,"PoissonsRatio",0.25);
```

Restrain all rigid-body motions of the plate by specifying sufficient constraints. For static analysis, the constraints must also resist the motion induced by applied load.

Set the `x`-component of displacement on the left edge (edge 3) to zero to resist the applied load. Set the `y`-component of displacement at the bottom left corner (vertex 3) to zero to restraint the rigid body motion.

```
structuralBC(model,"Edge",3,"XDisplacement",0);
structuralBC(model,"Vertex",3,"YDisplacement",0);
```

Apply the surface traction with a non-zero `x`-component on the right edge of the plate.

```
structuralBoundaryLoad(model,"Edge",1,"SurfaceTraction",[100;0]);
```

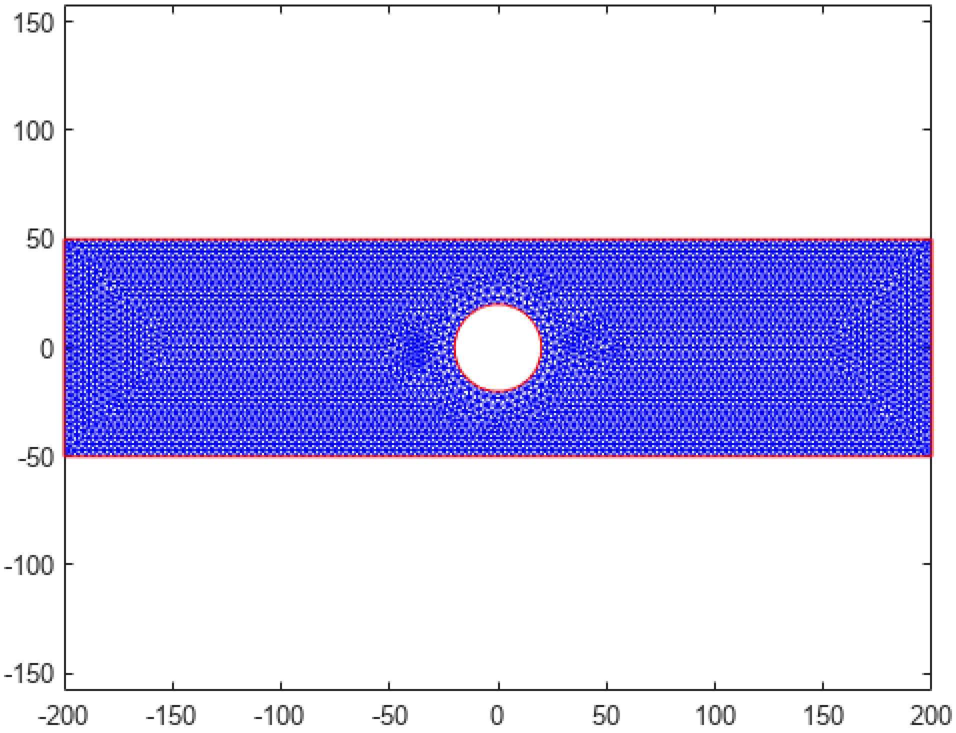
Generate Mesh and Solve

To capture the gradation in solution accurately, use a fine mesh. Generate the mesh, using `Hmax` to control the mesh size.

```
generateMesh(model,"Hmax",radius/6);
```

Plot the mesh.

```
figure
pdemesh(model)
```



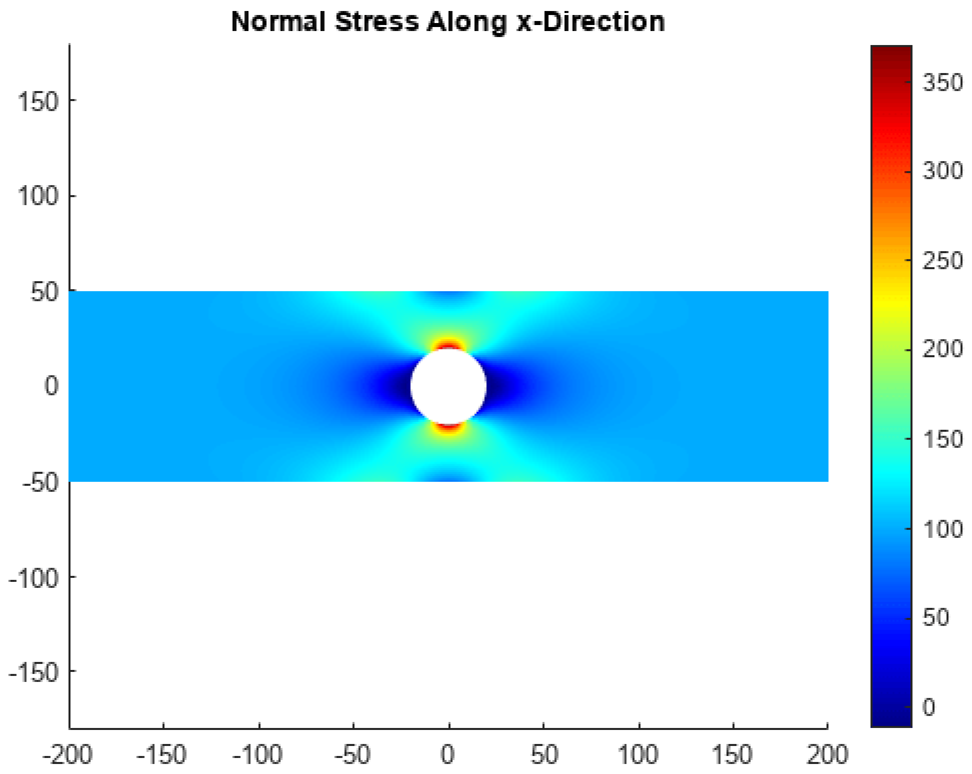
Solve the plane-stress elasticity model.

```
R = solve(model);
```

Plot Stress Contours

Plot the x-component of the normal stress distribution. The stress is equal to applied tension far away from the circular boundary. The maximum value of stress occurs near the circular boundary.

```
figure
pdeplot(model,"XYData",R.Stress.sxx,"ColorMap","jet")
axis equal
title("Normal Stress Along x-Direction")
```



Interpolate Stress

To see the details of the stress variation near the circular boundary, first define a set of points on the boundary.

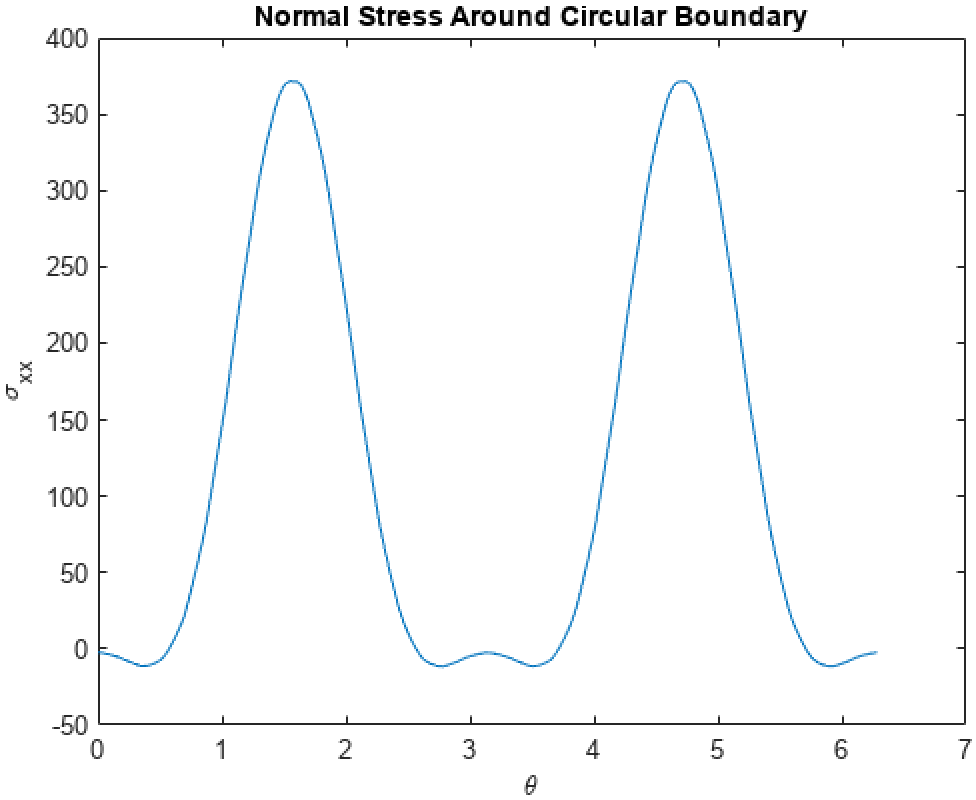
```
thetaHole = linspace(0,2*pi,200);
xr = radius*cos(thetaHole);
yr = radius*sin(thetaHole);
CircleCoordinates = [xr;yr];
```

Then interpolate stress values at these points by using `interpolateStress`. This function returns a structure array with its fields containing interpolated stress values.

```
stressHole = interpolateStress(R,CircleCoordinates);
```

Plot the normal direction stress versus angular position of the interpolation points.

```
figure
plot(thetaHole,stressHole.sxx)
xlabel("\theta")
ylabel("\sigma_{xx}")
title("Normal Stress Around Circular Boundary")
```



Solve the Same Problem Using Symmetric Model

The plate with a hole model has two axes of symmetry. Therefore, you can model a quarter of the geometry. The following model solves a quadrant of the full model with appropriate boundary conditions.

Create a structural model for the static plane-stress analysis.

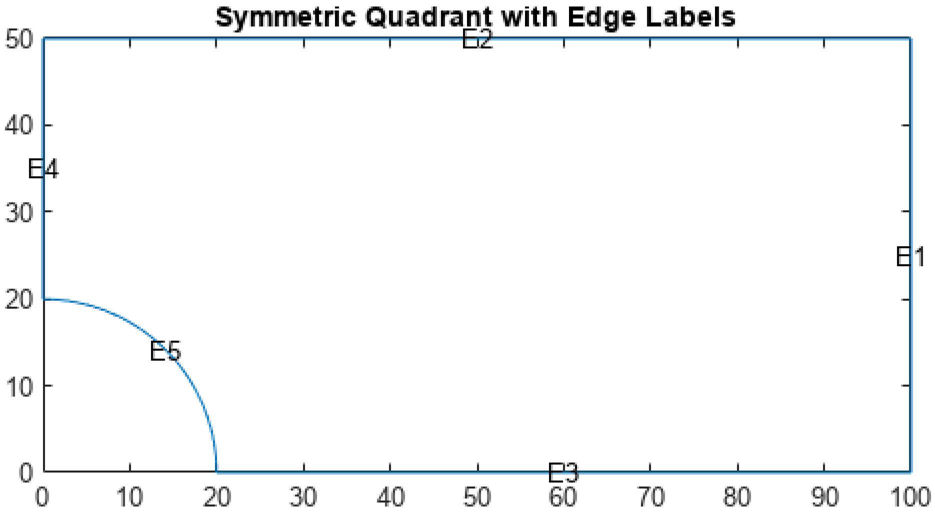
```
symModel = createpde("structural","static-planestress");
```

Create the geometry that represents one quadrant of the original model. You do not need to create additional edges to constrain the model properly.

```
R1 = [3 4 0 totalLength/2 totalLength/2 ...
      0 0 0 width width]';
C1 = [1 0 0 radius 0 0 0 0 0 0]';
gm = [R1 C1];
sf = 'R1-C1';
ns = char('R1','C1');
g = decsg(gm,sf,ns');
geometryFromEdges(symModel,g);
```

Plot the geometry displaying the edge labels.

```
figure
pdegplot(symModel,"EdgeLabel","on");
axis equal
title("Symmetric Quadrant with Edge Labels")
```



Specify structural properties of the material.

```
structuralProperties(symModel,"YoungsModulus",200E3, ...
                              "PoissonsRatio",0.25);
```

Apply symmetric constraints on the edges 3 and 4.

```
structuralBC(symModel,"Edge",[3 4],"Constraint","symmetric");
```

Apply surface traction on the edge 1.

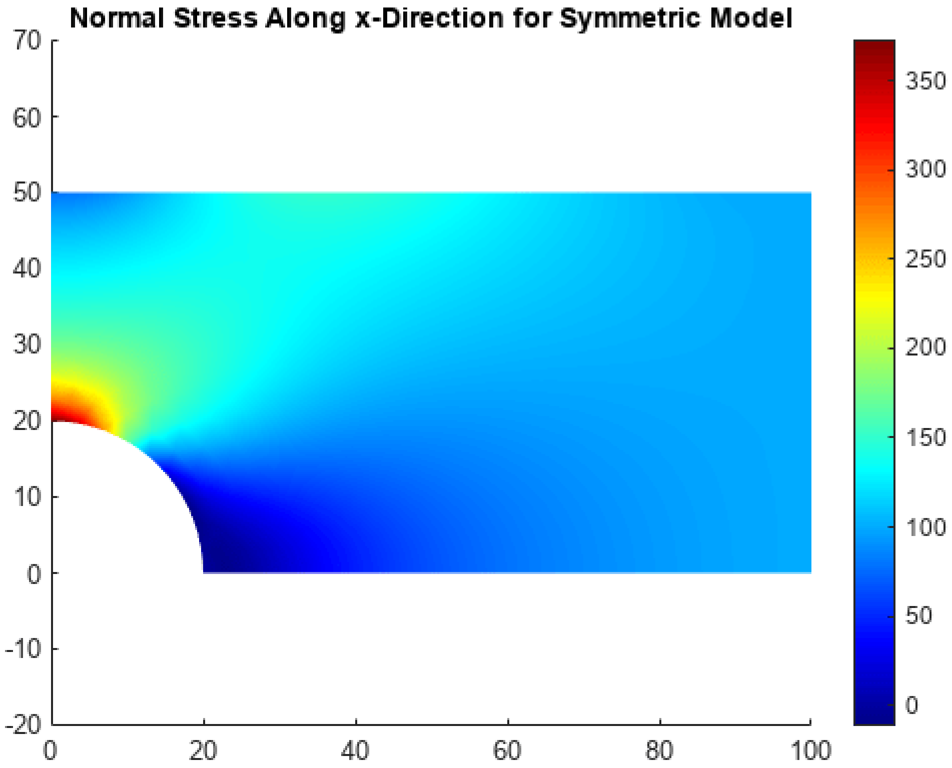
```
structuralBoundaryLoad(symModel,"Edge",1,"SurfaceTraction",[100;0]);
```

Generate mesh and solve the symmetric plane-stress model.

```
generateMesh(symModel,"Hmax",radius/6);
Rsym = solve(symModel);
```

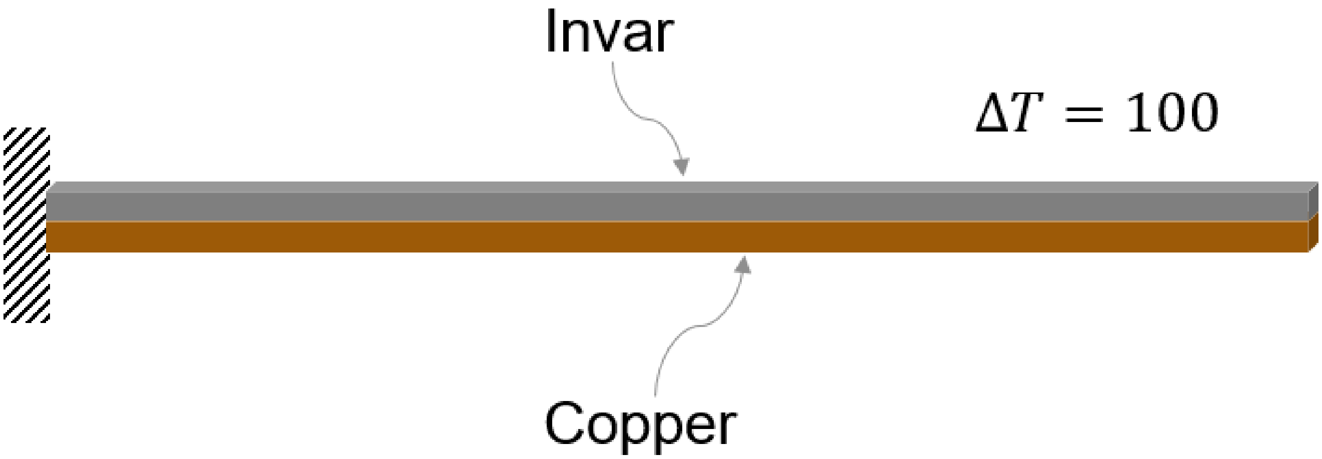
Plot the `x`-component of the normal stress distribution. The results are identical to the first quadrant of the full model.

```
figure
pdeplot(symModel,"XYData",Rsym.Stress.sxx,"ColorMap","jet");
axis equal
title("Normal Stress Along x-Direction for Symmetric Model")
```


Normal Stress Along x-Direction for Symmetric Model
70
60
50
40
30
20
10
0
-10
-20
0
20
40
60
80
100
350
300
250
200
150
100
50
0

Thermal Deflection of Bimetallic Beam

This example shows how to solve a coupled thermo-elasticity problem. Thermal expansion or contraction in mechanical components and structures occurs due to temperature changes in the operating environment. Thermal stress is a secondary manifestation: the structure experiences stresses when structural constraints prevent free thermal expansion or contraction of the component. Deflection of a bimetallic beam is a common physics experiment. A typical bimetallic beam consists of two materials bonded together. The coefficients of thermal expansion (CTE) of these materials are significantly different.



This example finds the deflection of a bimetallic beam using a structural finite-element model. The example compares this deflection to the analytic solution based on beam theory approximation.

Create a static structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Create a beam geometry with these dimensions.

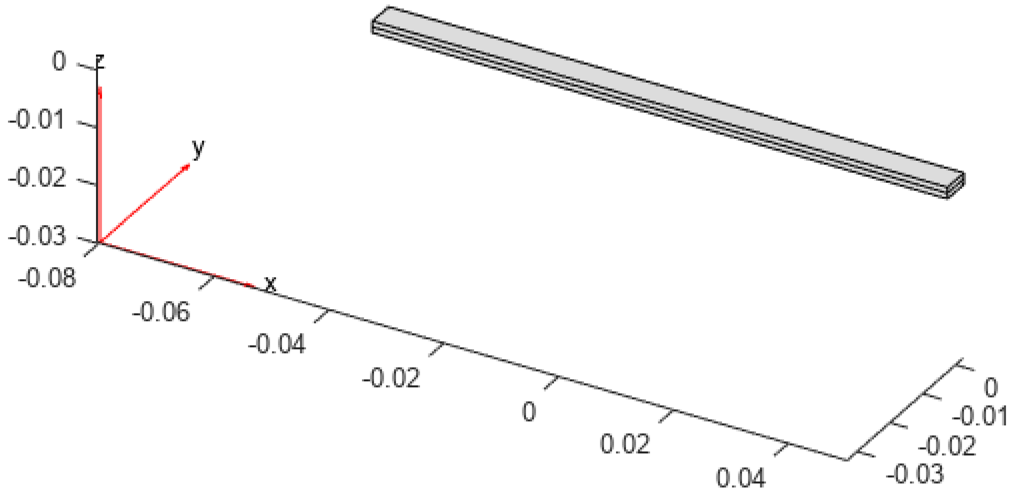
```
L = 0.1; % m
W = 5e-3; % m
H = 1e-3; % m
gm = multicuboid(L,W,[H,H],"Zoffset",[0,H]);
```

Include the geometry in the structural model.

```
structuralmodel.Geometry = gm;
```

Plot the geometry.

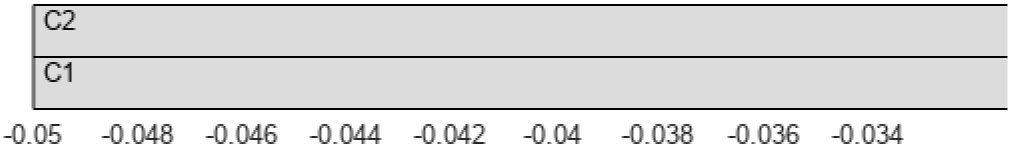
```
figure
pdegplot(structuralmodel)
```



Identify the cell labels of the cells for which you want to specify material properties.

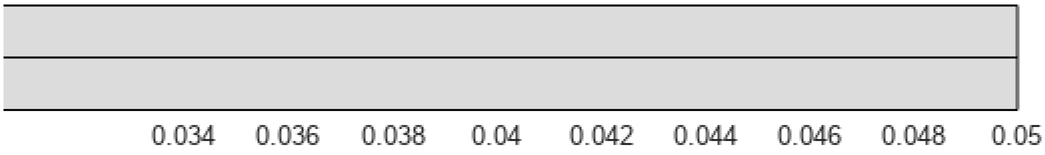
First, display the cell label for the bottom cell. To see the cell label clearly, zoom into the left end of the beam and rotate the geometry.

```
figure
pdegplot(structuralmodel,"CellLabels","on")
axis([-L/2 -L/3 -W/2 W/2 0 2*H])
view([0 0])
zticks([])
```



Now, display the cell label for the top cell. To see the cell label clearly, zoom into the right end of the beam and rotate the geometry.

```
figure
pdegplot(structuralmodel,"CellLabels","on")
axis([L/3 L/2 -W/2 W/2 0 2*H])
view([0 0])
zticks([])
```



Specify Young's modulus, Poisson's ratio, and the linear coefficient of thermal expansion to model linear elastic material behavior. To maintain unit consistency, specify all physical properties in SI units.

Assign the material properties of copper to the bottom cell.

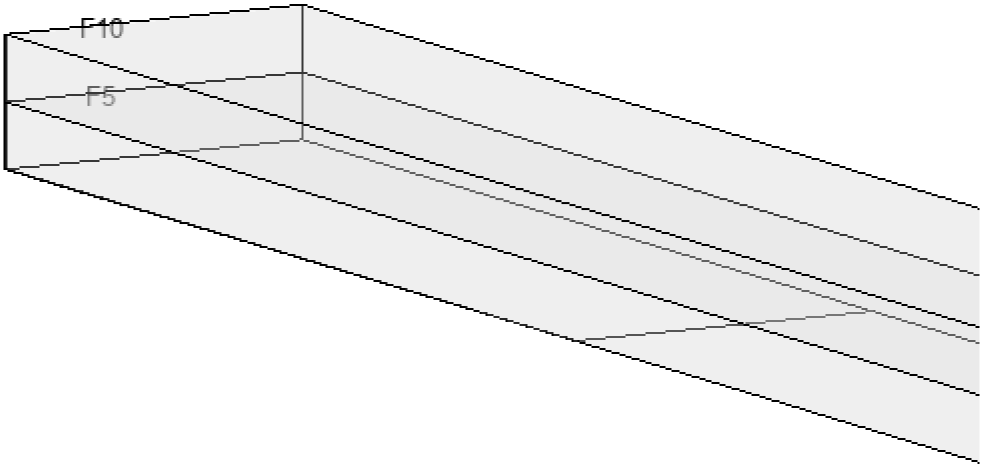
```
Ec = 137e9; % N/m^2
nuc = 0.28;
CTEc = 20.00e-6; % m/m-C
structuralProperties(structuralmodel,"Cell",1, ...
                                     "YoungsModulus",Ec, ...
                                     "PoissonsRatio",nuc, ...
                                     "CTE",CTEc);
```

Assign the material properties of invar to the top cell.

```
Ei = 130e9; % N/m^2
nui = 0.354;
CTEi = 1.2e-6; % m/m-C
structuralProperties(structuralmodel,"Cell",2, ...
                                     "YoungsModulus",Ei, ...
                                     "PoissonsRatio",nui, ...
                                     "CTE",CTEi);
```

For this example, assume that the left end of the beam is fixed. To impose this boundary condition, display the face labels on the left end of the beam.

```
figure
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.25)
axis([-L/2 -L/3 -W/2 W/2 0 2*H])
view([60 10])
xticks([])
yticks([])
zticks([])
```



Apply a fixed boundary condition on faces 5 and 10.

```
structuralBC(structuralmodel,"Face",[5,10],"Constraint","fixed");
```

Apply the temperature change as a thermal load. Use a reference temperature of 25 degrees Celsius and an operating temperature of 125 degrees Celsius. Thus, the temperature change for this model is 100 degrees Celsius.

```
structuralBodyLoad(structuralmodel,"Temperature",125);
structuralmodel.ReferenceTemperature = 25;
```

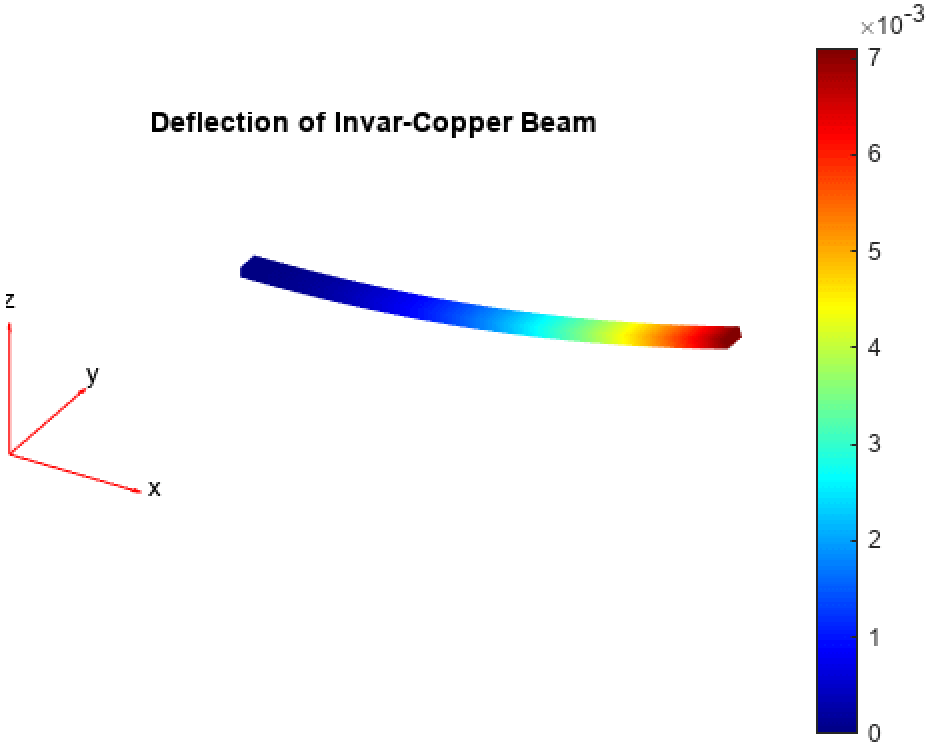
Generate a mesh and solve the model.

```
generateMesh(structuralmodel,"Hmax",H/2);
R = solve(structuralmodel);
```

Plot the deflected shape of the bimetallic beam with the magnitude of displacement as the colormap data.

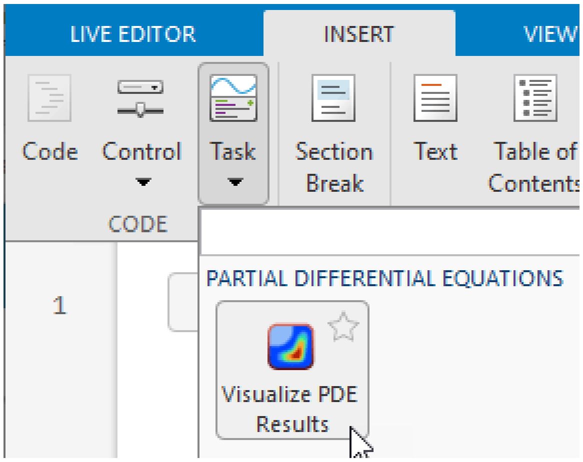
```
figure
pdeplot3D(structuralmodel,"ColorMapData",R.Displacement.Magnitude, ...
```

```
                          "Deformation",R.Displacement, ...
                          "DeformationScaleFactor",2)
title("Deflection of Invar-Copper Beam")
```



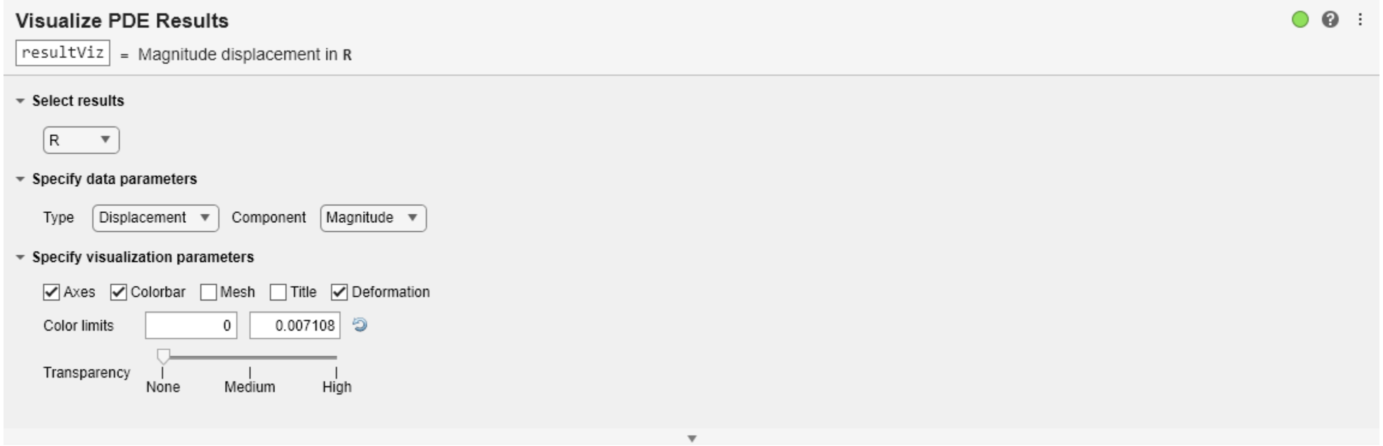
You also can plot the deflected shape of the bimetallic beam with the magnitude of displacement as the colormap data by using the **Visualize PDE Results** Live Editor task. First, create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

On the **Live Editor** tab, select **Task** > **Visualize PDE Results**. This action inserts the task into your script.



To plot the magnitude of displacement, follow these steps.

1. In the **Select results** section of the task, select R from the drop-down list.
2. In the **Specify data parameters** section of the task, set **Type** to *Displacement* and **Component** to *Magnitude*.



Visualize PDE Results

`resultViz` = Magnitude displacement in R

Select results

R

Specify data parameters

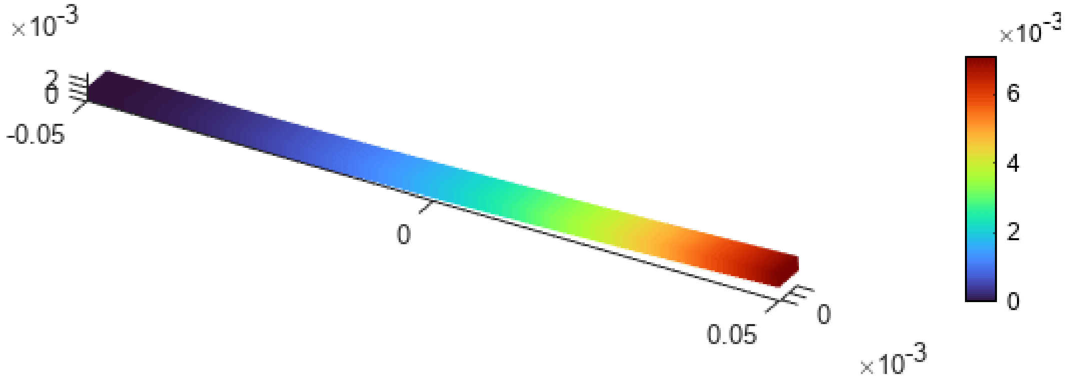
Type Displacement Component Magnitude

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☑ Deformation

Color limits 0 0.007108

Transparency None Medium High



Compute the deflection analytically, based on beam theory. The deflection of the beam is $\delta = \frac{6\Delta T(\alpha_c - \alpha_i)L^2}{K_1}$, where $K_1 = 14 + \frac{E_c}{E_i} + \frac{E_i}{E_c}$, ΔT is the temperature difference, α_c and α_i are the coefficients of thermal expansion of copper and invar, E_c and E_i are Young's modulus of copper and invar, and L is the length of the beam.

```
K1 = 14 + (Ec/Ei)+ (Ei/Ec);
deflectionAnalytical = 3*(CTEc - CTEi)*100*2*H*L^2/(H^2*K1);
```

Compare the analytical results and the results obtained in this example. The results are comparable because of the large aspect ratio.

```
PDEToobox_Deflection = max(R.Displacement.uz);
percentError = 100*(PDEToobox_Deflection - ...
                    deflectionAnalytical)/PDEToobox_Deflection;

bimetallicResults = table(PDEToobox_Deflection, ...
                          deflectionAnalytical,percentError);
bimetallicResults.Properties.VariableNames = {'PDEToolbox', ...
                                              'Analytical', ...
                                              'PercentageError'};
disp(bimetallicResults)
```

```
    PDEToolbox    Analytical    PercentageError
    __________    __________    _______________

    0.0071061     0.0070488         0.80608    
```

Axisymmetric Thermal and Structural Analysis of Disc Brake

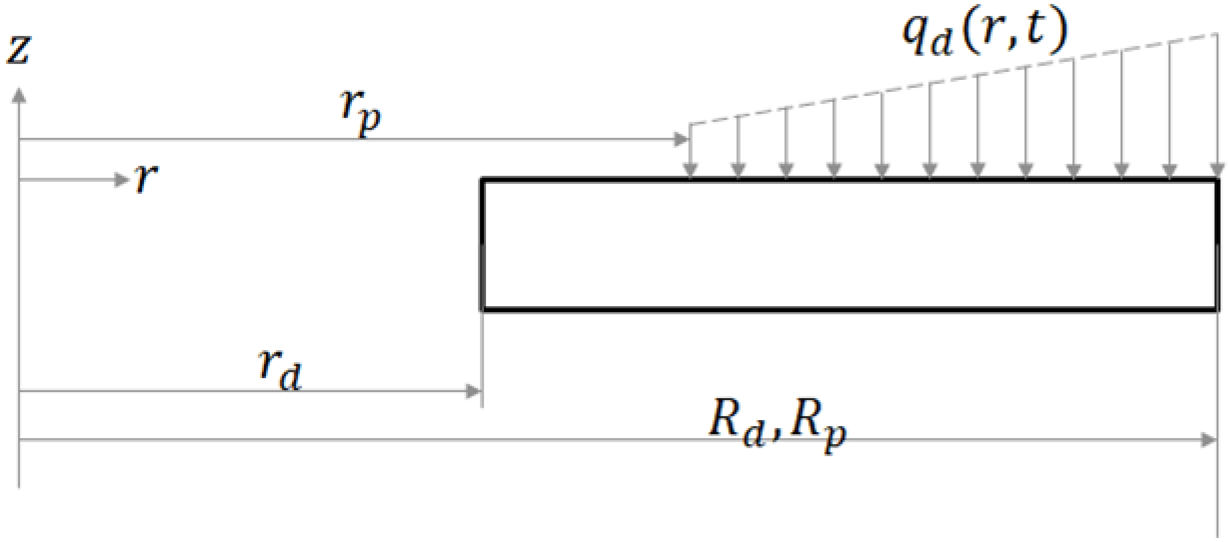
This example shows a quasistatic axisymmetric thermal stress analysis workflow by reproducing the results of the simplified disc brake model discussed in [1] on page 3-129. Disc brakes absorb mechanical energy through friction and transform it into thermal energy, which then dissipates. The example uses a simplified model of a disc brake in a single braking process from a constant initial angular speed to a standstill. The workflow has two steps:

1. Transient thermal analysis to compute the temperature distribution in the disc using the heat flux from brake pads
2. Quasistatic structural analysis to compute thermal stresses at several solution times using previously obtained temperature distribution to specify thermal loads

The resulting plots show the temperature distribution, radial stress, hoop stress, and von Mises stress for the corresponding solution times.

Disc Brake Properties and Geometry

Based on the assumptions used in [1] on page 3-129, the example reduces the analysis domain to a rectangular region corresponding to the axisymmetric section of the annular disc. Because of the geometric and load symmetry of the disc, the example models only half the thickness of the disc and the effect of one pad. In the following figure, the left edge corresponds to the inner radius of the disc r_d. The right edge corresponds to the outer radius of the disc R_d and also coincides with the outer radius of the pad R_p. The disc experiences pressure from the pad, which generates the heat flux. Instead of modeling the pad explicitly, include its effect in the thermal analysis by specifying this heat flux as a boundary condition from the inner radius of the pad r_p to the outer radius of the pad R_p.



Thermal Analysis: Compute Temperature Distribution

Create a transient axisymmetric thermal model.

```
modelT = createpde("thermal","transient-axisymmetric");
```

Create a geometry with two adjacent rectangles. The top edge of the longer rectangle (on the right) represents the disc-pad contact region.

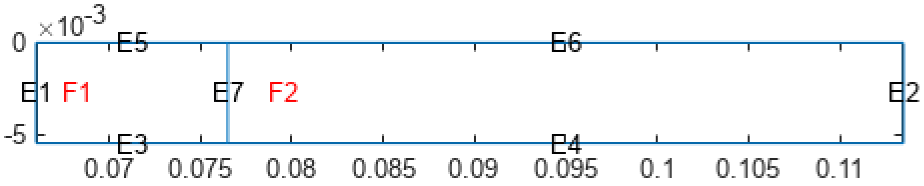
```
R1 = [3,4, [  66,  76.5,  76.5,   66, -5.5, -5.5, 0, 0]/1000]';
R2 = [3,4, [76.5, 113.5, 113.5, 76.5, -5.5, -5.5, 0, 0]/1000]';

gdm = [R1 R2];
ns = char('R1','R2');
g = decsg(gdm,'R1 + R2',ns');
```

Assign the geometry to the thermal model.

```
geometryFromEdges(modelT,g);
```

Plot the geometry with the edge and face labels.

```
figure
pdegplot(modelT,"EdgeLabels","on","FaceLabels","on")
```



Generate a mesh. To match the mesh used in [1] on page 3-129, use the linear geometric order instead of the default quadratic order.

```
generateMesh(modelT,"Hmax",0.5E-04,"GeometricOrder","linear");
```

Specify the thermal material properties of the disc.

```
alphad = 1.44E-5; % Diffusivity of disc
Kd = 51;
```

```
rhod = 7100;
cpd = Kd/rhod/alphad;
thermalProperties(modelT,"ThermalConductivity",Kd, ...
                         "MassDensity",rhod, ...
                         "SpecificHeat",cpd);
```

Specify the heat flux boundary condition to account for the pad region. For the definition of the `qFcn` function, see Heat Flux Function on page 3-128.

```
thermalBC(modelT,"Edge",6,"HeatFlux",@qFcn);
```

Set the initial temperature.

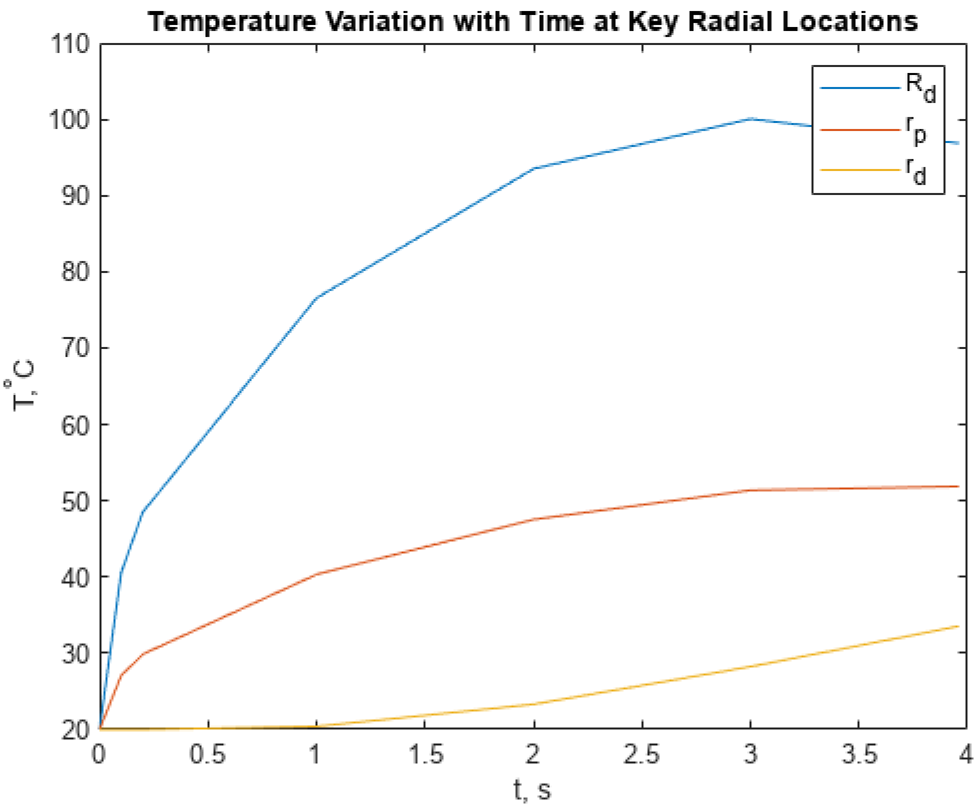
```
thermalIC(modelT,20);
```

Solve the model for the times used in [1] on page 3-129.

```
tlist = [0 0.1 0.2 1.0 2.0 3.0 3.96];
Rt = solve(modelT,tlist);
```

Plot the temperature variation with time at three key radial locations. The resulting plot is comparable to the plot obtained in [1] on page 3-129.

```
iTRd = interpolateTemperature(Rt,[0.1135;0],1:numel(Rt.SolutionTimes));
iTrp = interpolateTemperature(Rt,[0.0765;0],1:numel(Rt.SolutionTimes));
iTrd = interpolateTemperature(Rt,[0.066;0],1:numel(Rt.SolutionTimes));

figure
plot(tlist,iTRd)
hold on
plot(tlist,iTrp)
plot(tlist,iTrd)
title("Temperature Variation with Time at Key Radial Locations")
legend("R_d","r_p","r_d")
xlabel("t, s")
ylabel("T,^{\circ}C")
```



Structural Analysis: Compute Thermal Stress

Create an axisymmetric static structural analysis model.

```
model = createpde("structural","static-axisymmetric");
```

Assign the geometry and mesh used for the thermal model.

```
model.Geometry = modelT.Geometry;
model.Mesh = modelT.Mesh;
```

Specify the structural properties of the disc.

```
structuralProperties(model,"YoungsModulus",99.97E9, ...
                           "PoissonsRatio",0.29, ...
                           "CTE",1.08E-5);
```

Constrain the model to prevent rigid motion.

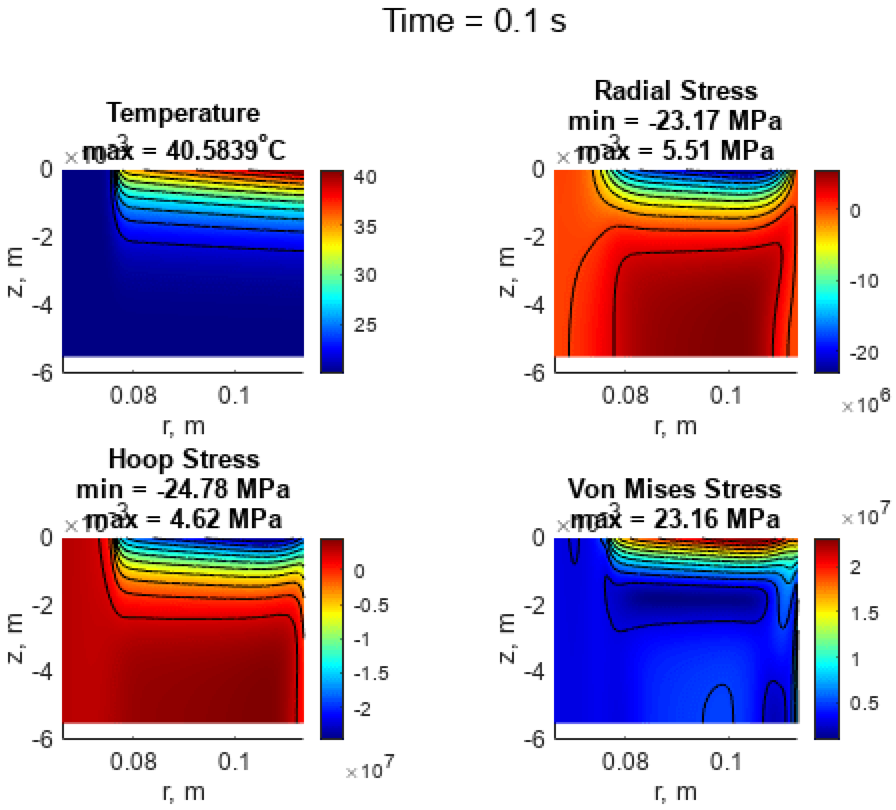
```
structuralBC(model,"Edge",[3,4],"ZDisplacement",0);
```

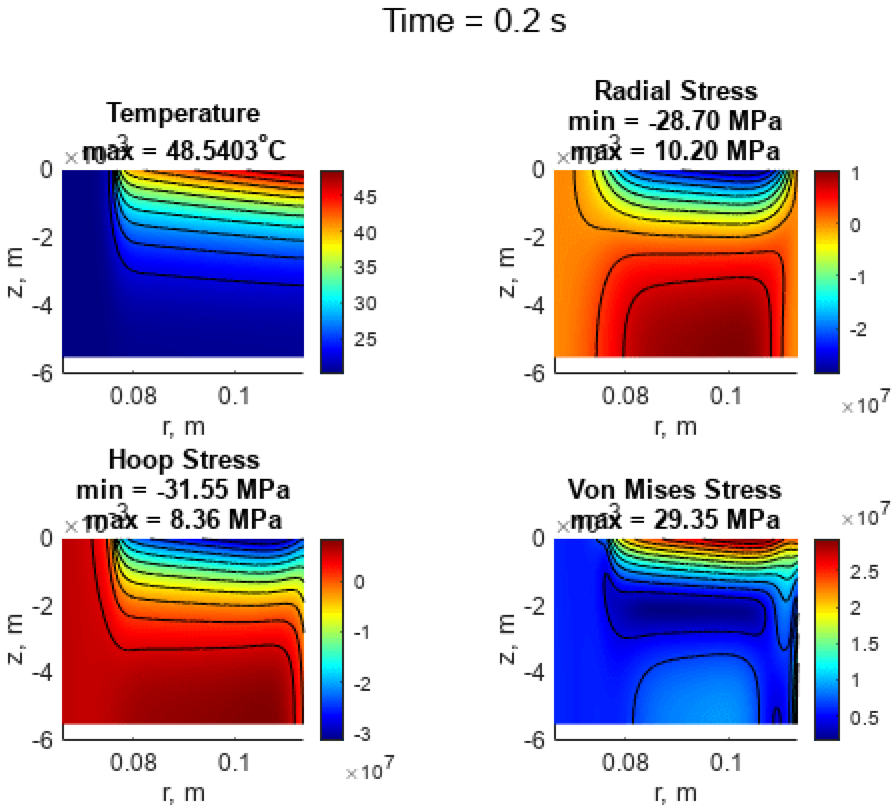
Specify the reference temperature that corresponds to the state of zero thermal stress of the model.

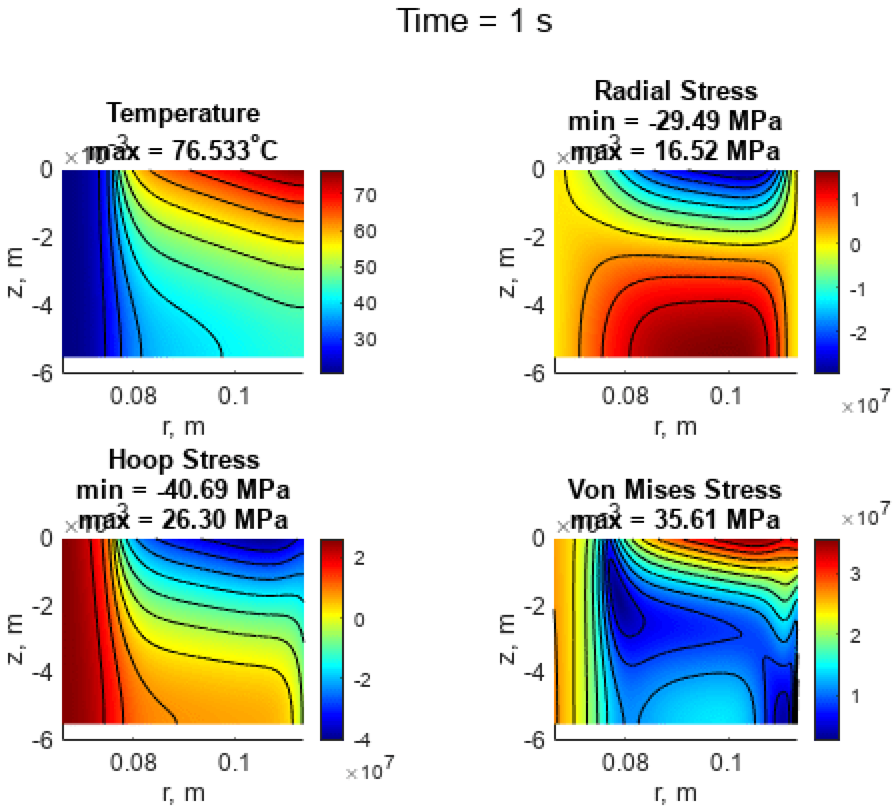
```
model.ReferenceTemperature = 20;
```

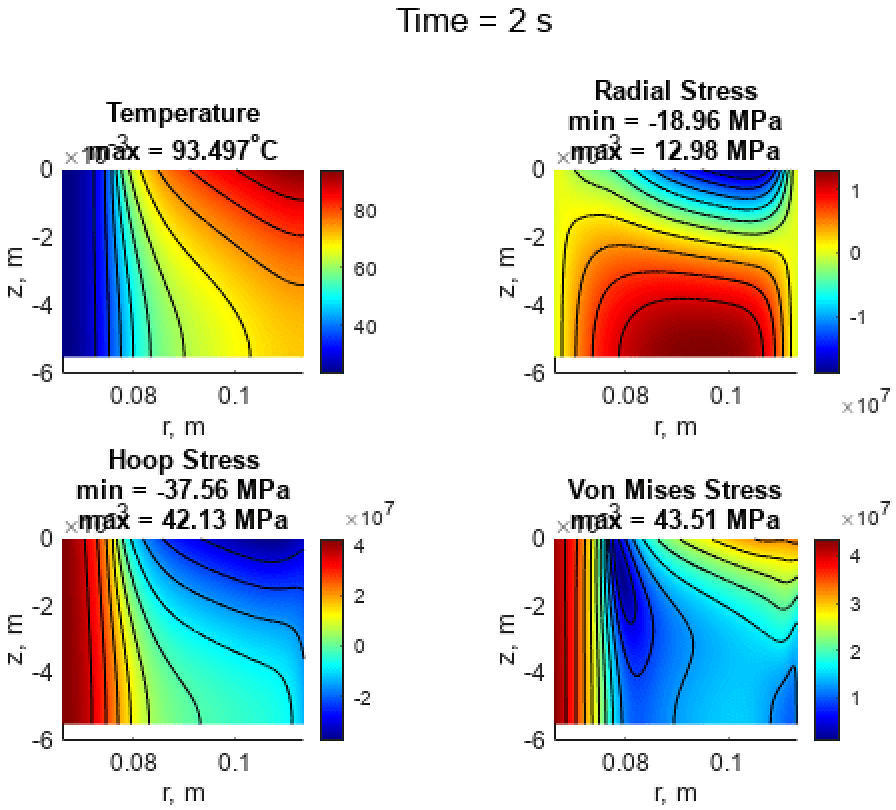
Specify the thermal load by using the transient thermal results `Rt`. The solution times are the same as in the thermal model analysis. For each solution time, solve the corresponding static structural analysis problem and plot the temperature distribution, radial stress, hoop stress, and von Mises stress. For the definition of the `plotResults` function, see Plot Results Function on page 3-129. The results are comparable to figure 5 from [1] on page 3-129.

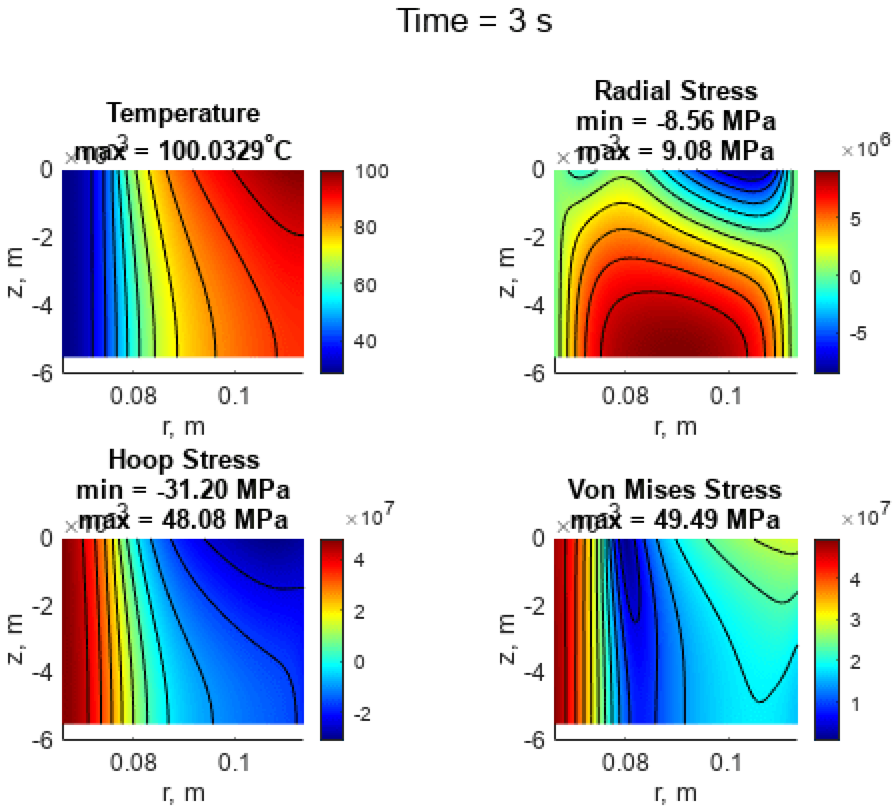
```
for n = 2:numel(Rt.SolutionTimes)
structuralBodyLoad(model,"Temperature",Rt,"TimeStep",n);
R = solve(model);
plotResults(model,R,modelT,Rt,n);
end
```

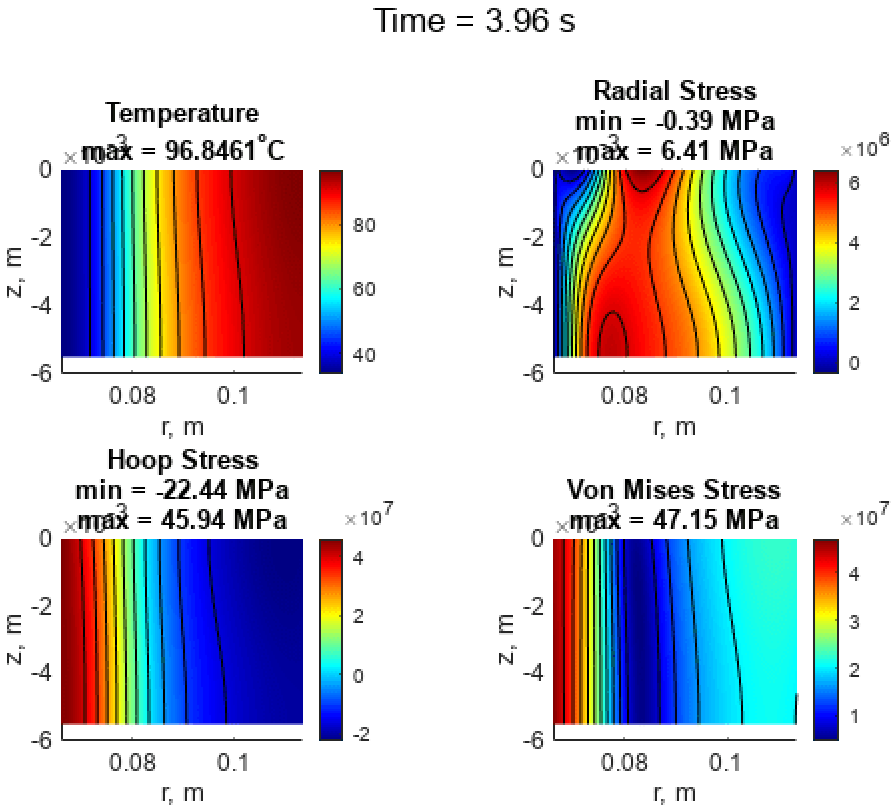



Time = 0.2 s
Temperature
max = 48.5403°C
×10^-3
0
-2
-4
-6
z, m
0.08
0.1
r, m
45
40
35
30
25
Radial Stress
min = -28.70 MPa
max = 10.20 MPa
1
0
-1
-2
×10^7
Hoop Stress
min = -31.55 MPa
max = 8.36 MPa
0
-1
-2
-3
×10^7
Von Mises Stress
max = 29.35 MPa
×10^7
2.5
2
1.5
1
0.5


Time = 1 s
Temperature
max = 76.533°C
×10^-3
0
-2
-4
-6
z, m
0.08
0.1
r, m
70
60
50
40
30
Radial Stress
min = -29.49 MPa
max = 16.52 MPa
×10^-3
0
-2
-4
-6
z, m
0.08
0.1
r, m
1
0
-1
-2
×10^7
Hoop Stress
min = -40.69 MPa
max = 26.30 MPa
×10^-3
0
-2
-4
-6
z, m
0.08
0.1
r, m
2
0
-2
-4
×10^7
Von Mises Stress
max = 35.61 MPa
×10^-3
0
-2
-4
-6
z, m
0.08
0.1
r, m
×10^7
3
2
1


Time = 2 s
Temperature
max = 93.497°C
×10-3
0
-2
-4
-6
z, m
0.08
0.1
r, m
80
60
40
Radial Stress
min = -18.96 MPa
max = 12.98 MPa
×10-3
z, m
r, m
1
0
-1
×10^7
Hoop Stress
min = -37.56 MPa
max = 42.13 MPa
×10-3
×10^7
z, m
r, m
4
2
0
-2
Von Mises Stress
max = 43.51 MPa
×10-3
×10^7
z, m
r, m
4
3
2
1


Time = 3 s
Temperature
max = 100.0329°C
×10^-3
100
80
60
40
Radial Stress
min = -8.56 MPa
max = 9.08 MPa
×10^6
5
0
-5
Hoop Stress
min = -31.20 MPa
max = 48.08 MPa
×10^7
4
2
0
-2
Von Mises Stress
max = 49.49 MPa
×10^7
4
3
2
1
0
-2
-4
-6
z, m
0.08
0.1
r, m



Heat Flux Function

This helper function computes the transient value of the heat flux from the pad to the disc. It uses the empirical formula from [1] on page 3-129.

```
function q = qFcn(r,s)
alphad = 1.44E-5; % Diffusivity of disc
Kd = 51; % Conductivity of disc
rhod = 7100; % Density of disc
cpd = Kd/rhod/alphad; % Specific heat capacity of disc

alphap = 1.46E-5; % Diffusivity of pad
Kp = 34.3; % Conductivity of pad
rhop = 4700; % Density of pad
cpp = Kp/rhop/alphap; % Specific heat capacity of pad

f = 0.5; % Coefficient of friction
omega0 = 88.464; % Initial angular velocity
ts = 3.96; % Stopping time
p0 = 1.47E6*(64.5/360); % Pressure only spans 64.5 deg occupied by pad

omegat = omega0*(1 - s.time/ts); % Angular speed over time

eta = sqrt(Kd*rhod*cpd)/(sqrt(Kd*rhod*cpd) + sqrt(Kp*rhop*cpp));
q = (eta)*f*omegat*r.r*p0;
end
```

Plot Results Function

This helper function plots the temperature distribution, radial stress, hoop stress, and von Mises stress.

```
function plotResults(model,R,modelT,Rt,tID)
figure
subplot(2,2,1)
pdeplot(modelT,"XYData",Rt.Temperature(:,tID), ...
               "ColorMap","jet","Contour","on")
title({'Temperature'; ...
      ['max = ' num2str(max(Rt.Temperature(:,tID))) '^{\circ}C']})
xlabel("r, m")
ylabel("z, m")

subplot(2,2,2)
pdeplot(model,"XYData",R.Stress.srr, ...
              "ColorMap","jet","Contour","on")
title({'Radial Stress'; ...
       ['min = ' num2str(min(R.Stress.srr)/1E6,'%3.2f') ' MPa']; ...
       ['max = ' num2str(max(R.Stress.srr)/1E6,'%3.2f') ' MPa']})
xlabel("r, m")
ylabel("z, m")

subplot(2,2,3)
pdeplot(model,"XYData",R.Stress.sh, ...
              "ColorMap","jet","Contour","on")
title({'Hoop Stress'; ...
      ['min = ' num2str(min(R.Stress.sh)/1E6,'%3.2f') ' MPa']; ...
      ['max = ' num2str(max(R.Stress.sh)/1E6,'%3.2f') ' MPa']})
xlabel("r, m")
ylabel("z, m")

subplot(2,2,4)
pdeplot(model,"XYData",R.VonMisesStress, ...
              "ColorMap","jet","Contour","on")
title({'Von Mises Stress'; ...
      ['max = ' num2str(max(R.VonMisesStress)/1E6,'%3.2f') ' MPa']})
xlabel("r, m")
ylabel("z, m")

sgtitle(['Time = ' num2str(Rt.SolutionTimes(tID)) ' s'])
end
```

References

[1] Adamowicz, Adam. "Axisymmetric FE Model to Analysis of Thermal Stresses in a Brake Disc." *Journal of Theoretical and Applied Mechanics* 53, issue 2 (April 2015): 357–370. https://doi.org/10.15632/jtam-pl.53.2.357.

Electrostatic Potential in Air-Filled Frame

This example shows how to find the electrostatic potential in an air-filled annular quadrilateral frame.

The PDE governing this problem is the Poisson equation

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

Here, ρ is the space charge density, and ε is the absolute dielectric permittivity of the material. The toolbox uses the relative permittivity of the material ε_r, such that $\varepsilon = \varepsilon_r\varepsilon_0$, where ε_0 is the absolute permittivity of the vacuum. The relative permittivity for air is 1.00059. Note that the permittivity of the air does not affect the result in this example as long as the coefficient is constant.

Assuming that there is no charge in the domain, the Poisson equation simplifies to the Laplace equation: $\Delta V = 0$. For this example, use the following boundary conditions:

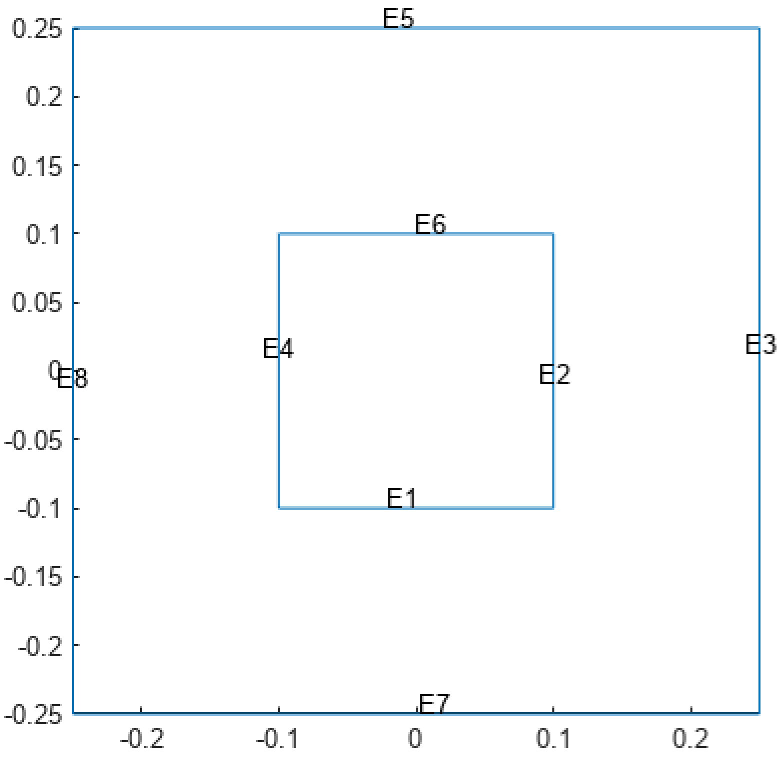
- The electrostatic potential at the inner boundary is $1000V$.
- The electrostatic potential at the outer boundary is $0V$.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot a geometry of a simple frame.

```
importGeometry(emagmodel,"Frame.STL");
pdegplot(emagmodel,"EdgeLabels","on")
```



Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1.00059);
```

Specify the electrostatic potential at the inner boundary.

```
electromagneticBC(emagmodel,"Voltage",1000,"Edge",[1 2 4 6]);
```

Specify the electrostatic potential at the outer boundary.

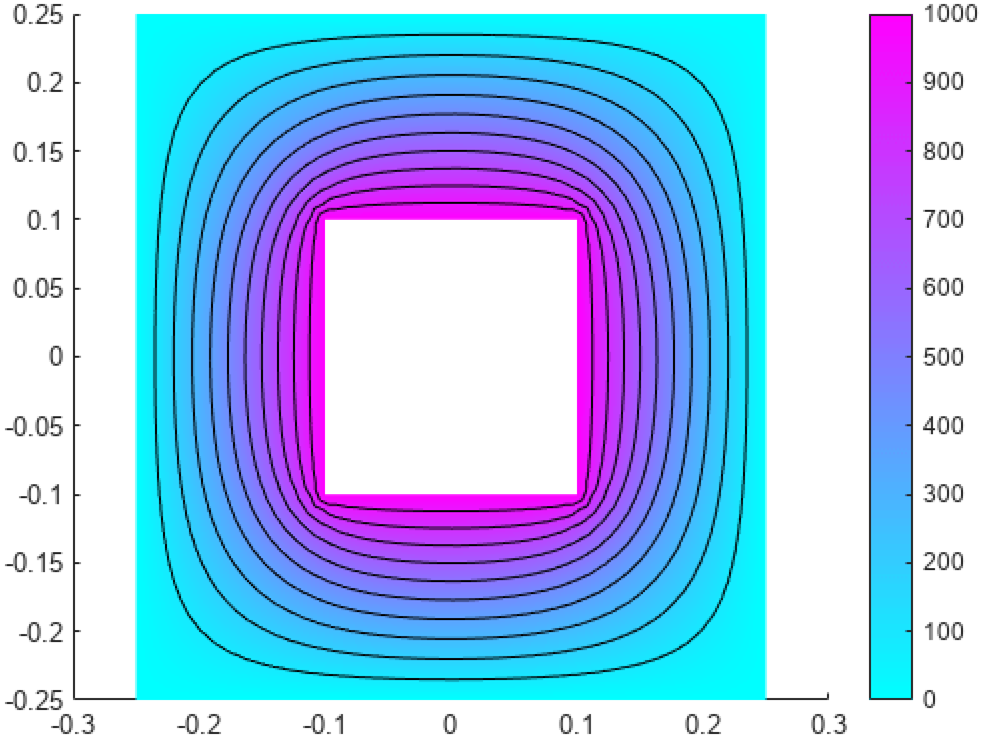
```
electromagneticBC(emagmodel,"Voltage",0,"Edge",[3 5 7 8]);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model. Plot the electric potential using the `Contour` parameter to display equipotential lines.

```
R = solve(emagmodel);
u = R.ElectricPotential;
pdeplot(emagmodel,"XYData",u,"Contour","on")
```


0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
-0.25
-0.3
-0.2
-0.1
0
0.1
0.2
0.3
1000
900
800
700
600
500
400
300
200
100
0

Electrostatic Potential in Air-Filled Frame: PDE Modeler App

Find the electrostatic potential in an air-filled annular quadrilateral frame using the PDE Modeler 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 Modeler app uses the relative permittivity $\varepsilon_r = \varepsilon/\varepsilon_0$, where ε_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 Modeler app, follow these steps:

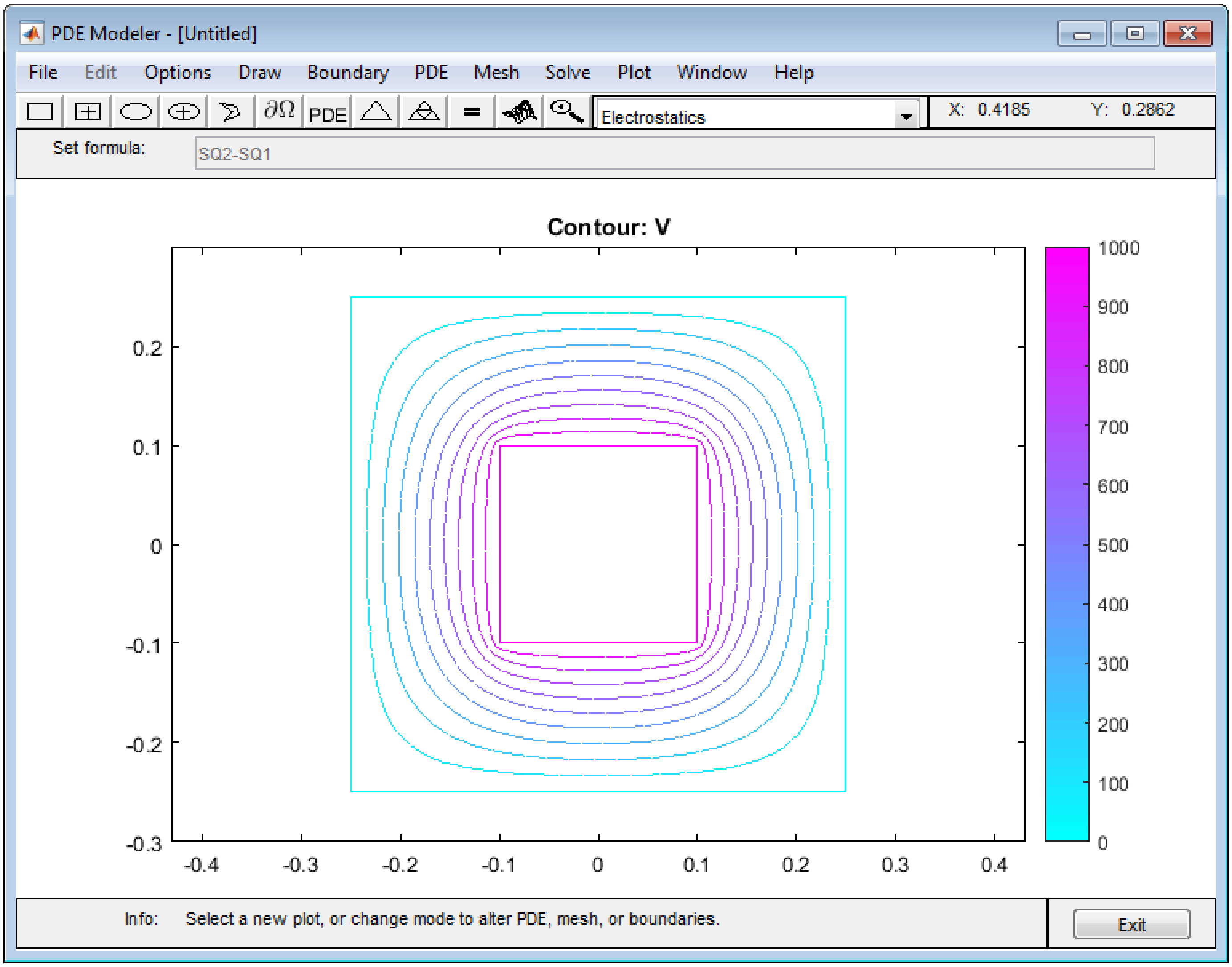
1. Draw the following two squares.

   ```
   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.



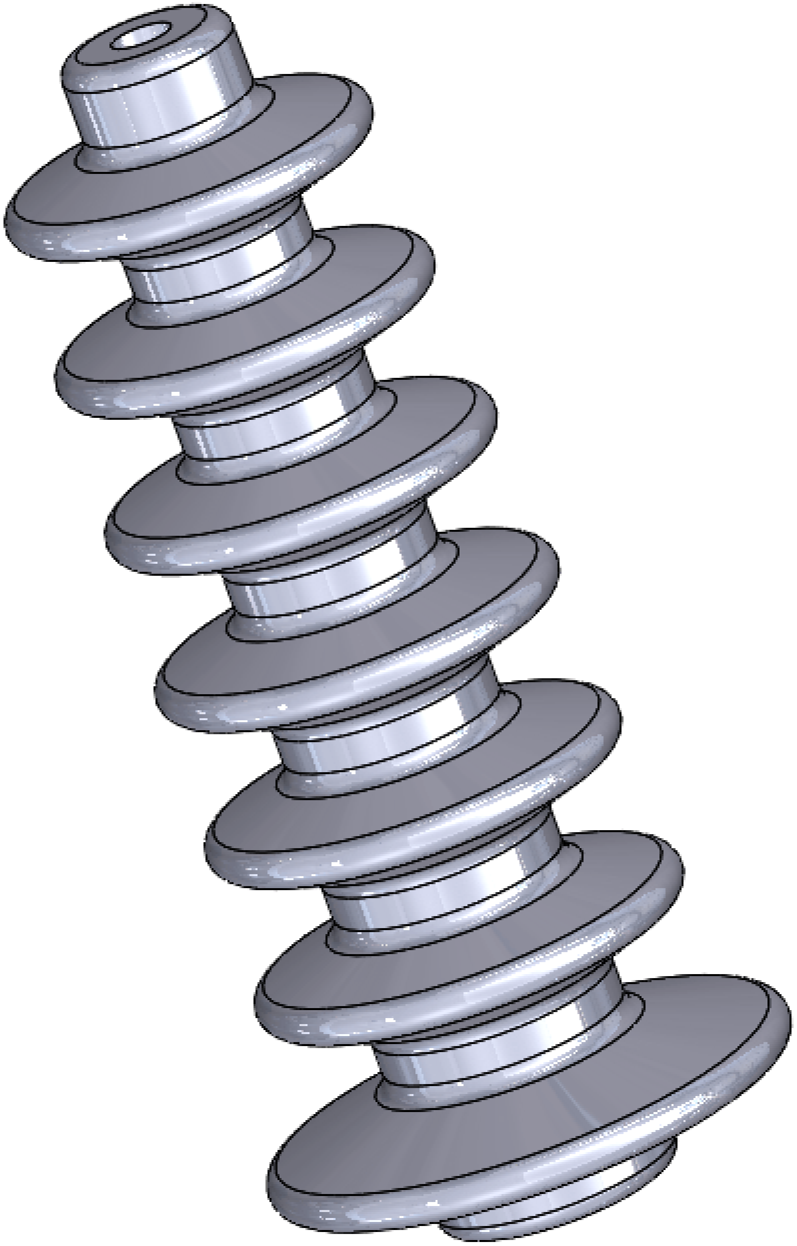
X: 0.4185 Y: 0.2862

Set formula: SQ2-SQ1

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

Electrostatic Analysis of Transformer Bushing Insulator

This example shows how to compute the electric field intensity in a bushing insulator of a transformer. Bushing insulators must withstand large electric fields due to the potential difference between the ground and the high-voltage conductor. This example uses a 3-D electrostatic model to compute the voltage distribution and electric field intensity in the bushing.

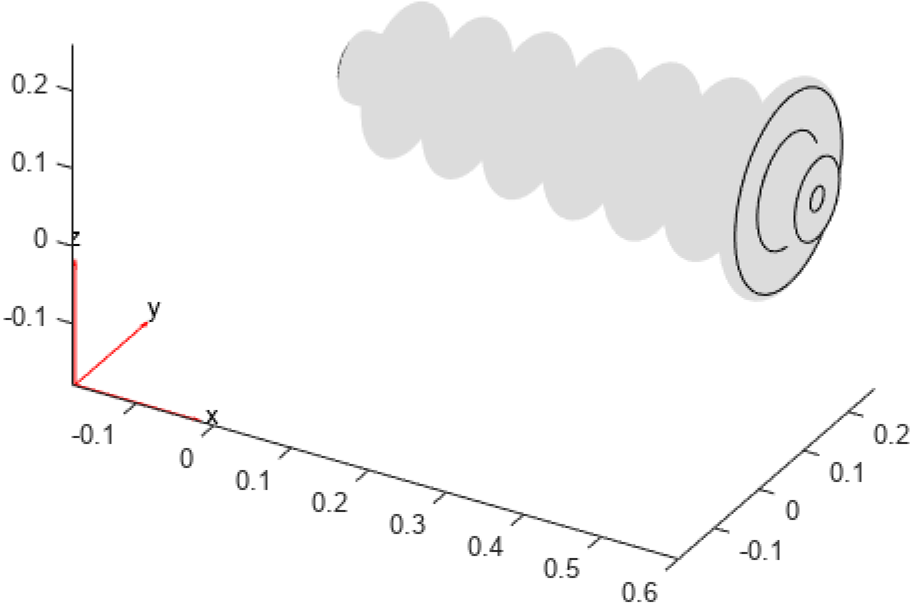


Create an electromagnetic model for electrostatic analysis.

```
model = createpde("electromagnetic","electrostatic");
```

Import and plot the bushing geometry.

```
gmBushing = importGeometry("TransformerBushing.stl");
pdegplot(gmBushing)
```

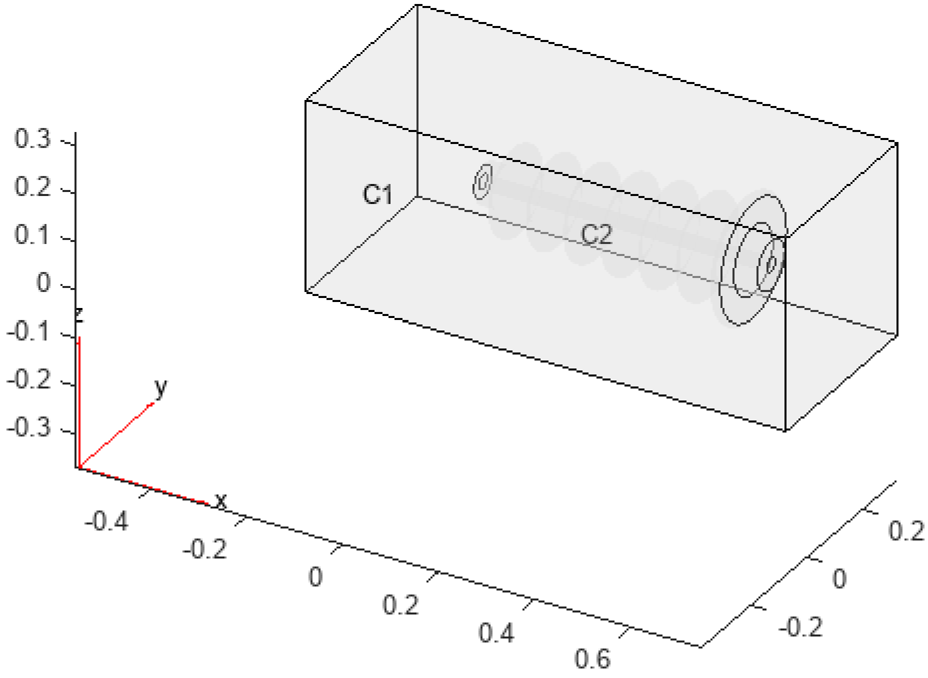


Model the surrounding air as a cuboid, and position the cuboid to contain the bushing at its center.

```
gmAir = multicuboid(1,0.4,0.4);
gmAir.translate([0.25,0.125,-0.07]);
gmModel = addCell(gmAir,gmBushing);
```

Plot the resulting geometry with the cell labels.

```
pdegplot(gmModel,"CellLabels","on","FaceAlpha",0.25)
```



Include the geometry in the model.

```
model.Geometry = gmModel;
```

Specify the vacuum permittivity value in the SI system of units.

```
model.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the air.

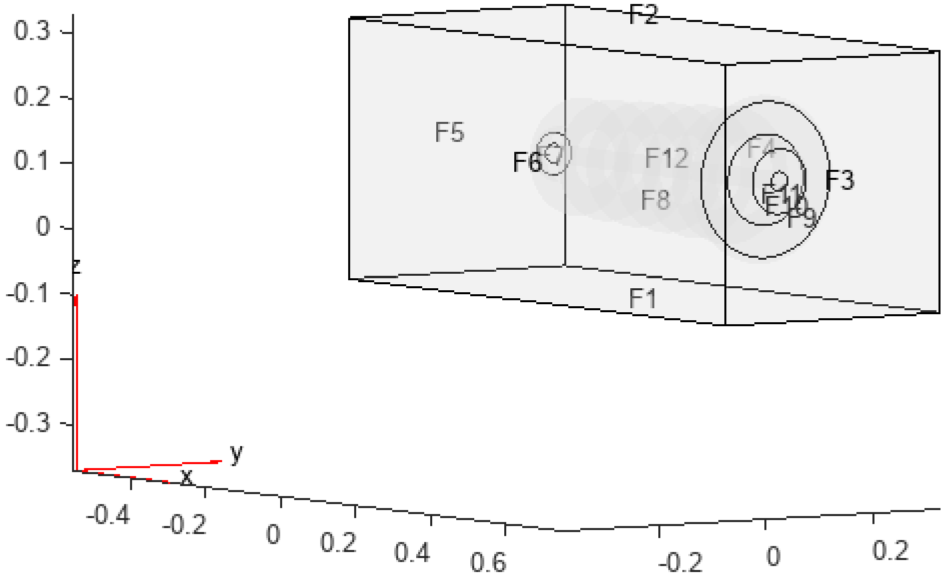
```
electromagneticProperties(model,"Cell",1,"RelativePermittivity",1);
```

Specify the relative permittivity of the bushing insulator.

```
electromagneticProperties(model,"Cell",2,"RelativePermittivity",5);
```

Before specifying boundary conditions, identify the face IDs by plotting the geometry with the face labels. To see the IDs more clearly, rotate the geometry.

```
pdegplot(gmModel,"FaceLabels","on","FaceAlpha",0.2)
view([55 5])
```



Specify the voltage boundary condition on the inner walls of the bushing exposed to conductor.

```
electromagneticBC(model,"Face",12,"Voltage",10E3);
```

Specify the grounding boundary condition on the surface in contact with the oil tank.

```
electromagneticBC(model,"Face",9,"Voltage",0);
```

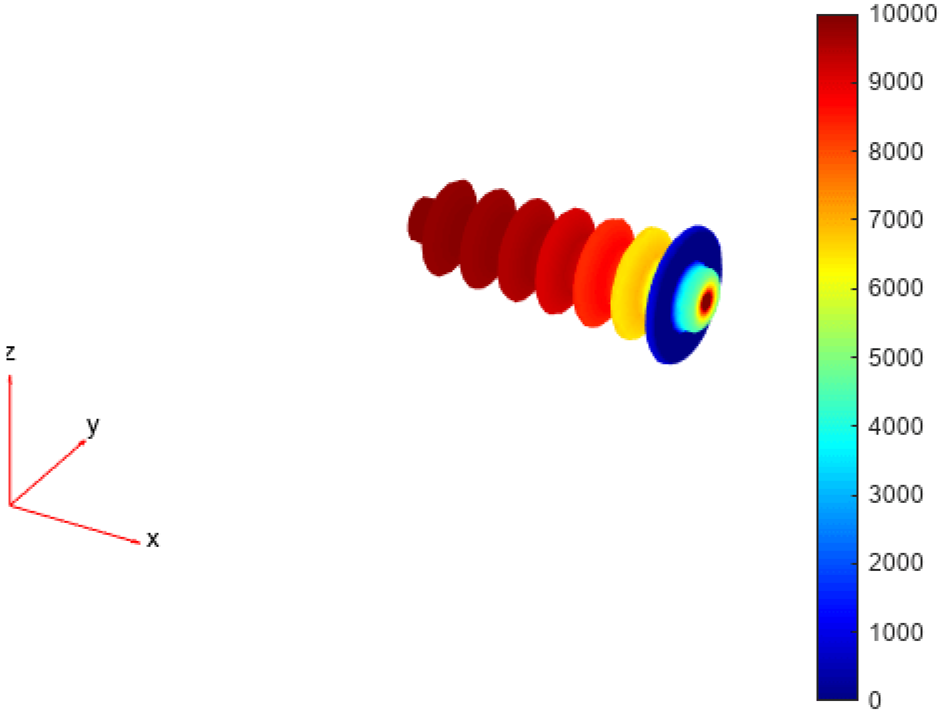
Generate a mesh and solve the model.

```
generateMesh(model);
R = solve(model)
```

```
R = 
  ElectrostaticResults with properties:

      ElectricPotential: [41034x1 double]
          ElectricField: [1x1 FEStruct]
    ElectricFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

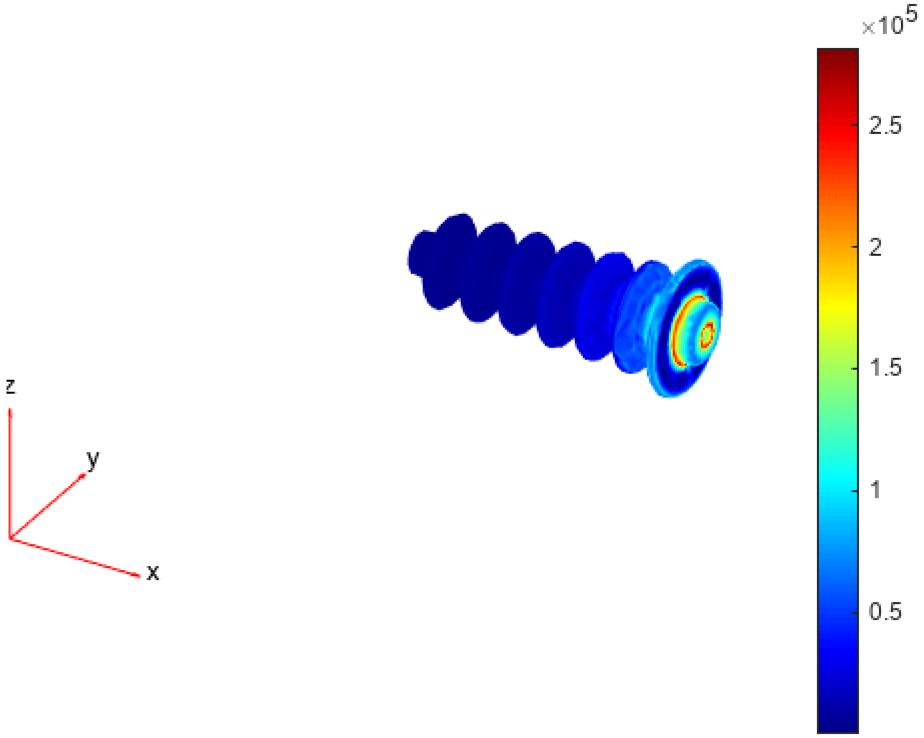
Plot the voltage distribution in the bushing.

```
elemsBushing = findElements(model.Mesh,"Region","Cell",2);
pdeplot3D(model.Mesh.Nodes, ...
          model.Mesh.Elements(:,elemsBushing), ...
          "ColorMapData",R.ElectricPotential);
```



Plot the magnitude of the electric field intensity in the bushing.

```
Emag = sqrt(R.ElectricField.Ex.^2 + ...
            R.ElectricField.Ey.^2 + ...
            R.ElectricField.Ez.^2);
pdeplot3D(model.Mesh.Nodes, ...
          model.Mesh.Elements(:,elemsBushing), ...
          "ColorMapData",Emag);
```


×10^5
2.5
2
1.5
1
0.5
z
y
x

Magnetic Flux Density in H-Shaped Magnet

This example shows how to solve a 2-D magnetostatic model for a ferromagnetic frame with an H-shaped cavity. This setup generates a uniform magnetic field due to the presence of two coils.

Create a geometry that consists of a rectangular frame with an H-shaped cavity, four rectangles representing the two coils, and a unit square representing the air domain around the magnet. Specify all dimensions in millimeters, and use the value `convfactor = 1000` to convert the dimensions to meters.

```
convfactor = 1000;
```

Create the H-shaped geometry to model the cavity.

```
xCoordsCavity = [-425 -125 -125 125 125 425 425 ...
               125 125 -125 -125 -425]/convfactor;
yCoordsCavity = [-400 -400 -100 -100 -400 -400 ...
               400 400 100 100 400 400]/convfactor;
RH = [2;12;xCoordsCavity';yCoordsCavity'];
```

Create the geometry to model the rectangular ferromagnetic frame.

```
RS = [3;4;[-525;525;525;-525;-500;-500;500;500]/convfactor];
zeroPad  = zeros(numel(RH)-numel(RS),1);
RS = [RS;zeroPad];
```

Create the geometries to model the coils.

```
RC1 = [3;4;[150;250;250;150;120;120;350;350]/convfactor;
                                                zeroPad];
RC2 = [3;4;[-150;-250;-250;-150;120;120;350;350]/convfactor;
                                                zeroPad];
RC3 = [3;4;[150;250;250;150;-120;-120;-350;-350]/convfactor;
                                                zeroPad];
RC4 = [3;4;[-150;-250;-250;-150;-120;-120;-350;-350]/convfactor;
                                                zeroPad];
```

Create the geometry to model the air domain around the magnet.

```
RD = [3;4;[-1000;1000;1000;-1000;-1000; ...
         -1000;1000;1000]/convfactor;zeroPad];
```

Combine the shapes into one matrix.

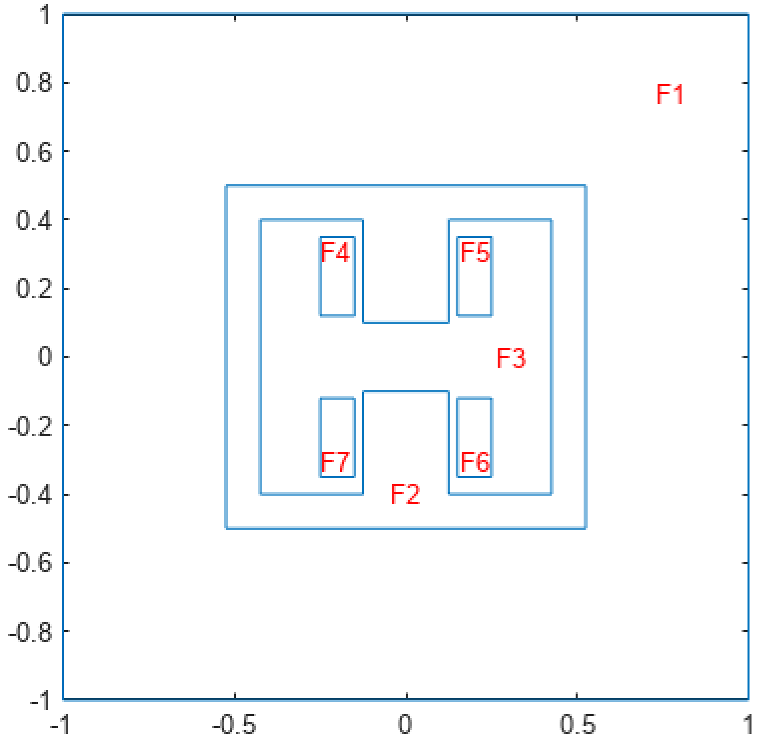
```
gd = [RS,RH,RC1,RC2,RC3,RC4,RD];
```

Create a set formula and create the geometry.

```
ns = char('RS','RH','RC1','RC2','RC3','RC4','RD');
g = decsg(gd,'(RS+RH+RC1+RC2+RC3+RC4)+RD',ns');
```

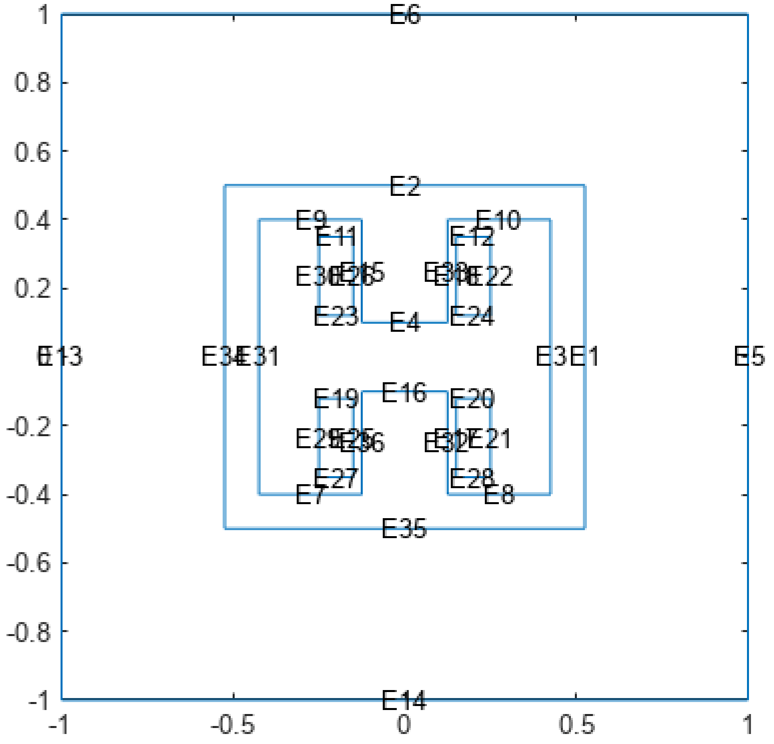
Plot the geometry with the face labels.

```
figure
pdegplot(g,"FaceLabels","on")
```



Plot the geometry with the edge labels.

```
figure
pdegplot(g,"EdgeLabels","on")
```

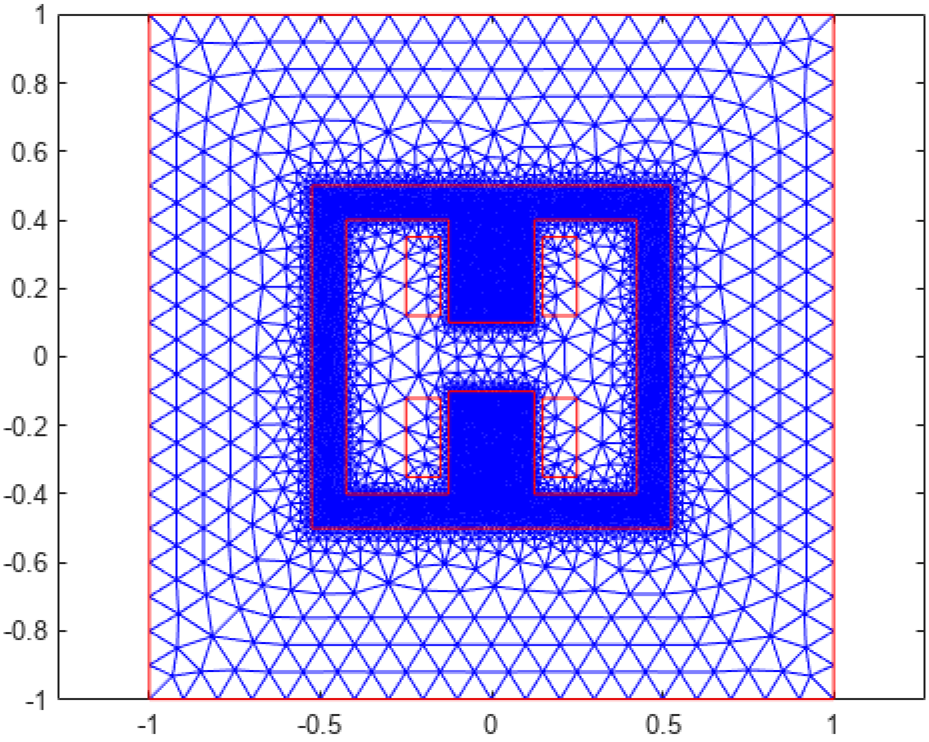


Create a magnetostatic model and include the geometry in the model.

```
model = createpde("electromagnetic","magnetostatic");
geometryFromEdges(model,g);
```

Generate a mesh with fine refinement in the ferromagnetic frame.

```
generateMesh(model,"Hface",{2,0.01},"Hmax",0.1,"Hgrad",2);
figure
pdemesh(model)
```



Specify the vacuum permeability value in the SI system of units.

```
model.VacuumPermeability = 1.2566370614E-6;
```

Specify a relative permeability of 1 for all domains.

```
electromagneticProperties(model,"RelativePermeability",1);
```

Now specify the large relative permeability of the ferromagnetic frame.

```
electromagneticProperties(model,"RelativePermeability",10000,"Face",2);
```

Specify the current density values on the upper and lower coils.

```
electromagneticSource(model,"CurrentDensity",1E6,"Face",[5,6]);
electromagneticSource(model,"CurrentDensity",-1E6,"Face",[4,7]);
```

Specify that the magnetic potential on the outer surface of the air domain is 0.

```
electromagneticBC(model,"Edge",[5,6,13,14],"MagneticPotential",0);
```

Solve the model.

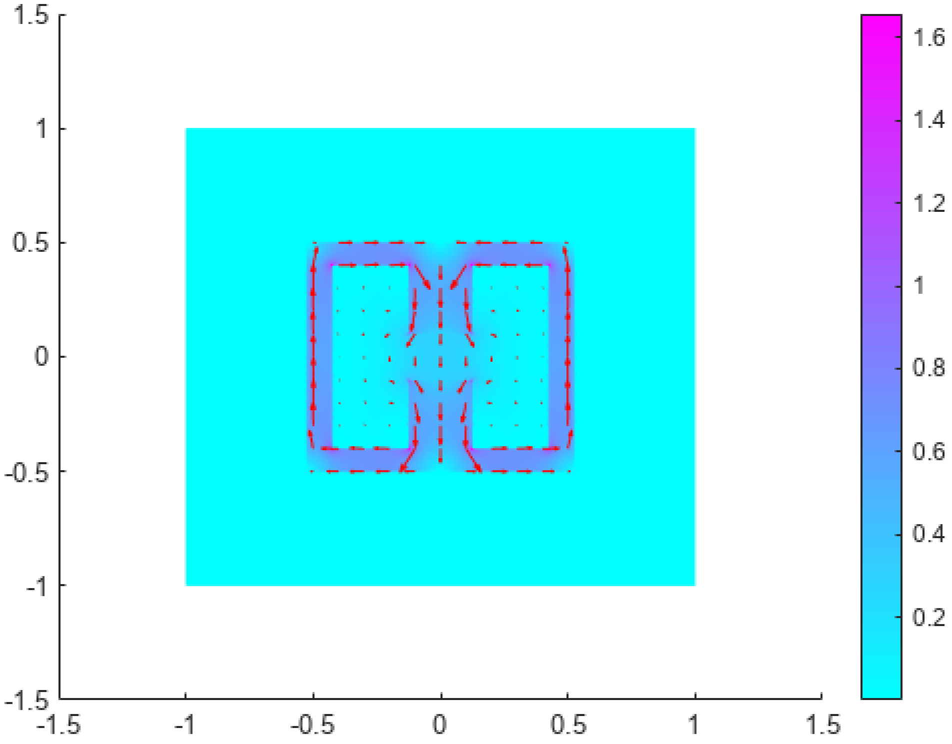
```
R = solve(model)
```

```
R = 
  MagnetostaticResults with properties:
```

```
      MagneticPotential: [26381x1 double]
          MagneticField: [1x1 FEStruct]
    MagneticFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

Plot the magnitude of the flux density.

```
Bmag = sqrt(R.MagneticFluxDensity.Bx.^2 + ...
            R.MagneticFluxDensity.By.^2);

pdeplot(model,"XYData",Bmag, ...
              "FlowData",[R.MagneticFluxDensity.Bx ...
                          R.MagneticFluxDensity.By])
```

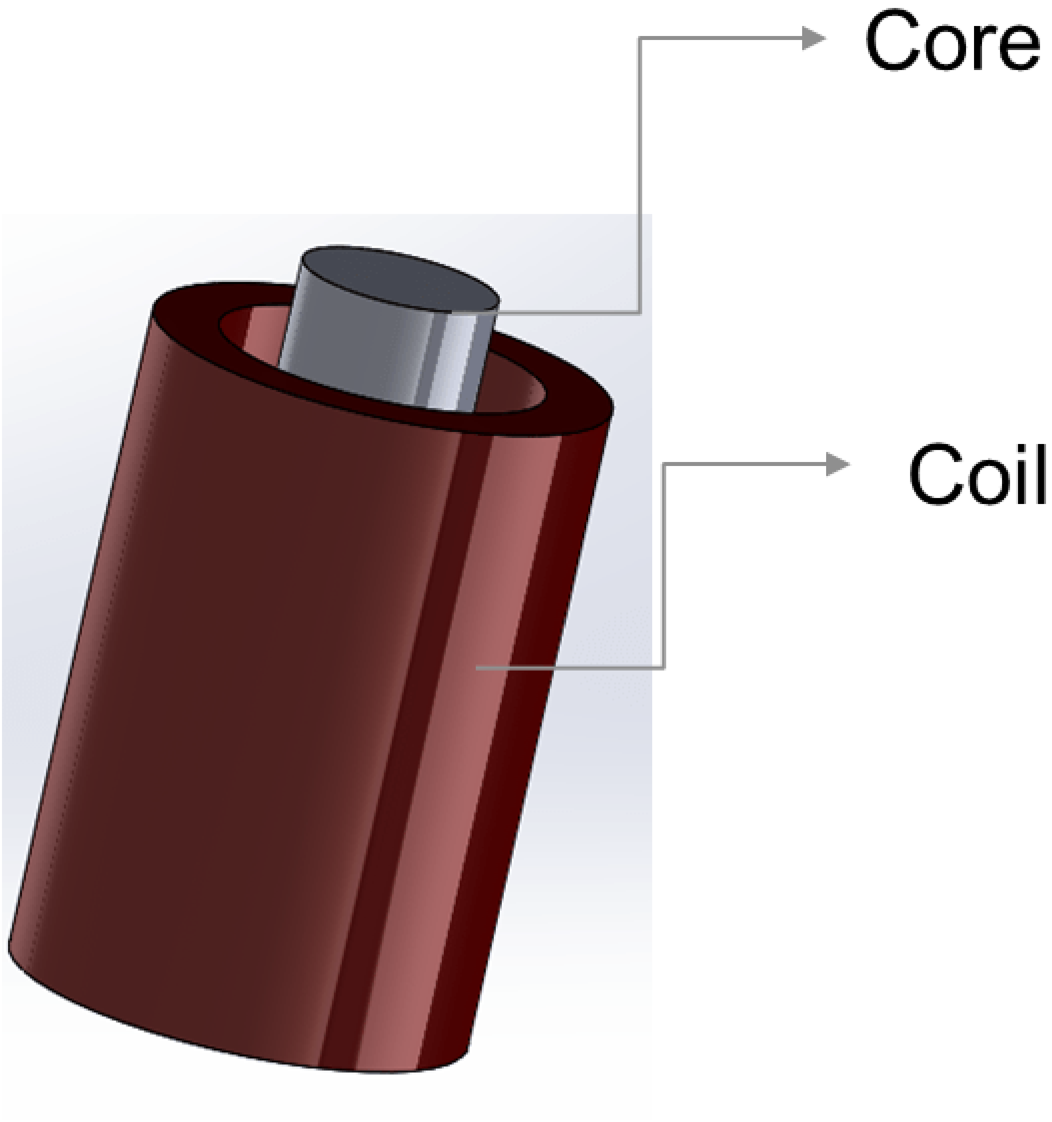


References

[1] Kozlowski, A., R. Rygal, and S. Zurek. "Large DC electromagnet for semi-industrial thermomagnetic processing of nanocrystalline ribbon." *IEEE Transactions on Magnetics* 50, issue 4 (April 2014): 1-4. https://ieeexplore.ieee.org/document/6798057.

Magnetic Flux Density in Electromagnet

This example shows how to solve a 3-D magnetostatic problem for a solenoid with a finite length iron core. Using a ferromagnetic core with high permeability, such as an iron core, inside a solenoid increases magnetic field and flux density. In this example, you find the magnetic flux density for a geometry consisting of a coil with a finite length core in a cylindrical air domain.



The first part of the example solves the magnetostatic problem using a 3-D model. The second part solves the same problem using an axisymmetric 2-D model to speed up computations.

3-D Model of Coil with Core

Create geometries consisting of three cylinders: a solid circular cylinder models the core, an annular circular cylinder models the coil, and a larger circular cylinder models the air around the coil.

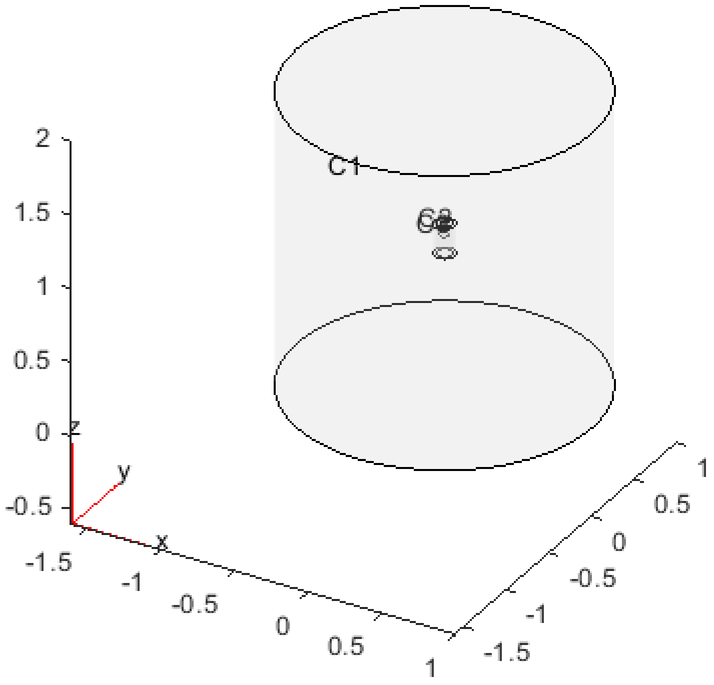
```
coreGm = multicylinder(0.03,0.1);
coilGm = multicylinder([0.05 0.07],0.2,"Void",[1 0]);
airGm = multicylinder(1,2);
```

Position the core and coil so that the finite length core is located near the top of coil.

```
coreGm = translate(coreGm,[0 0 1.025]);
coilGm = translate(coilGm,[0 0 0.9]);
```

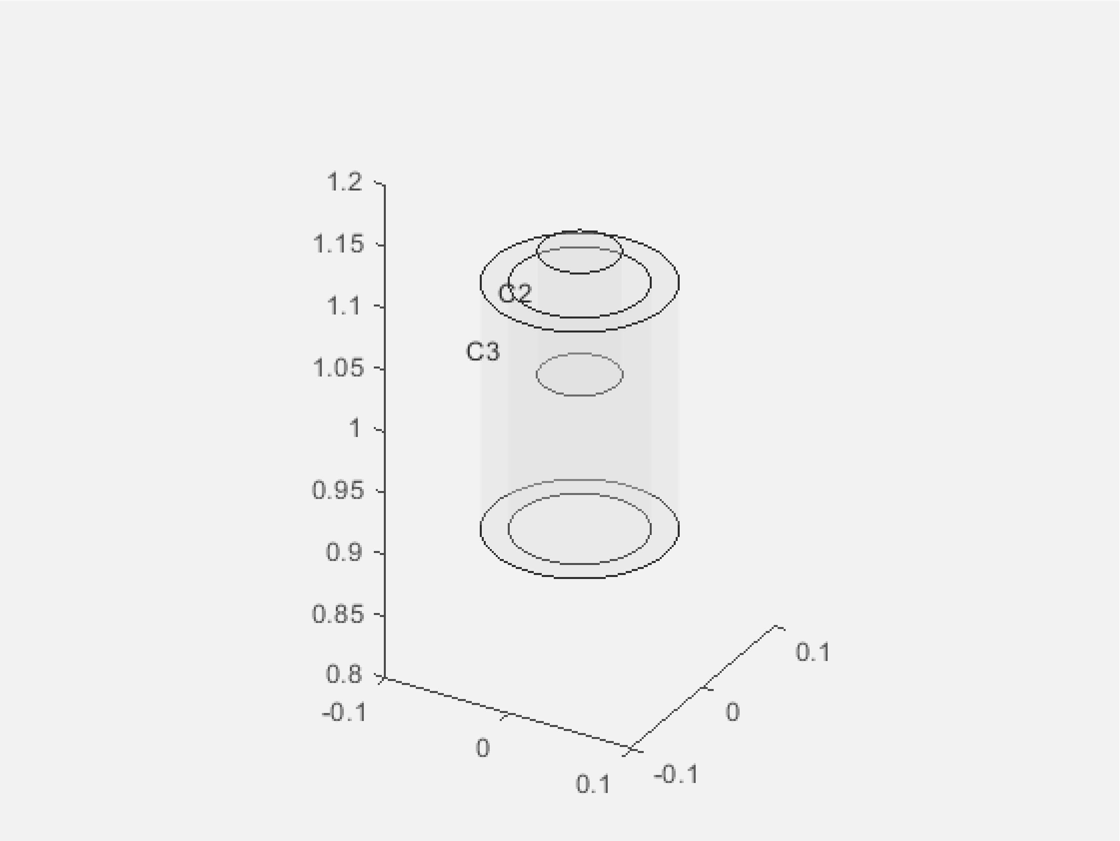
Combine the geometries and plot the result.

```
gm = addCell(airGm,coreGm);
gm = addCell(gm,coilGm);
pdegplot(gm,"FaceAlpha",0.2,"CellLabels","on")
```



Zoom in to see the cell labels on the core and coil.

```
figure
pdegplot(gm,"FaceAlpha",0.2,"CellLabels","on")
axis([-0.1 0.1 -0.1 0.1 0.8 1.2])
```



Create an electromagnetic model and assign air geometry to the model.

```
model3D = createpde("electromagnetic","magnetostatic");
model3D.Geometry = gm;
```

Specify the vacuum permeability value in the SI system of units.

```
model3D.VacuumPermeability = 1.2566370614E-6;
```

Specify a relative permeability of 1 for all domains.

```
electromagneticProperties(model3D,"RelativePermeability",1);
```

Now specify the large relative permeability of the core.

```
electromagneticProperties(model3D,"RelativePermeability",10000, ...
                                  "Cell",2);
```

Assign an excitation current using a function that defines counterclockwise current density in the coil.

```
electromagneticSource(model3D,"CurrentDensity",@windingCurrent3D, ...
                              "Cell",3);
```

Specify that the magnetic potential on the outer surface of the air domain is 0.

```
electromagneticBC(model3D,"MagneticPotential",[0;0;0],"Face",1:3);
```

Generate a mesh where only the core and coil regions are well refined and the air domain is relatively coarse to limit the size of the problem.

```
internalFaces = cellFaces(model3D.Geometry,2:3);
generateMesh(model3D,"Hface",{internalFaces,0.007});
```

Solve the model.

```
R = solve(model3D)
```

```
R = 
  MagnetostaticResults with properties:

      MagneticPotential: [1×1 FEStruct]
          MagneticField: [1×1 FEStruct]
    MagneticFluxDensity: [1×1 FEStruct]
                   Mesh: [1×1 FEMesh]
```

Find the magnitude of the flux density.

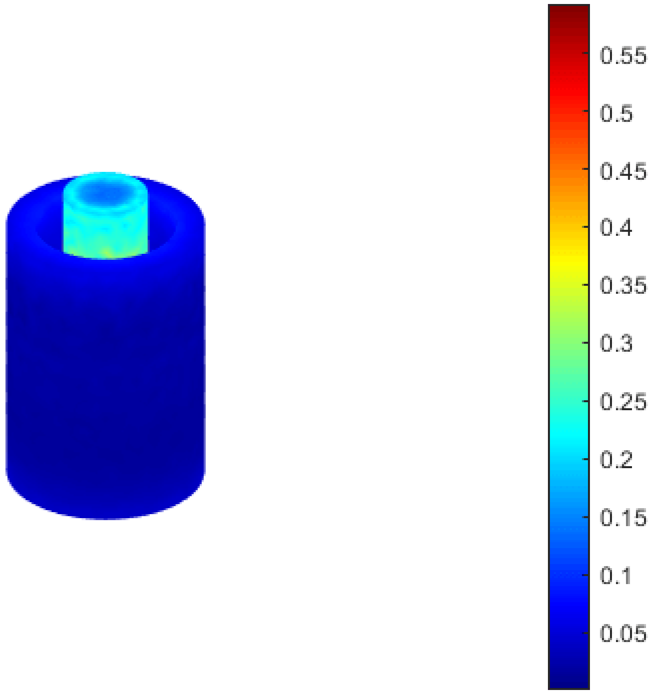
```
Bmag = sqrt(R.MagneticFluxDensity.Bx.^2 + ...
            R.MagneticFluxDensity.By.^2 + ...
            R.MagneticFluxDensity.Bz.^2);
```

Find the mesh elements belonging to the core and the coil.

```
coreAndCoilElem = findElements(model3D.Mesh,"region","Cell",[2 3]);
```

Plot the magnitude of the flux density on the core and coil.

```
pdeplot3D(model3D.Mesh.Nodes, ...
          model3D.Mesh.Elements(:,coreAndCoilElem), ...
          "ColorMapData",Bmag)
axis([-0.1 0.1 -0.1 0.1 0.8 1.2])
```

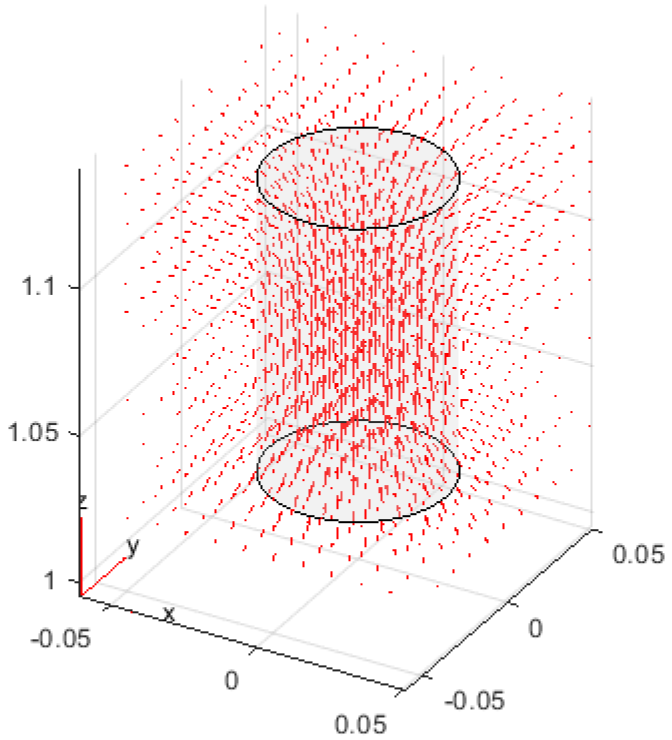


Interpolate the flux to a grid covering the portion of the geometry near the core.

```
x = -0.05:0.01:0.05;
z = 1.02:0.01:1.14;
y = x;
[X,Y,Z] = meshgrid(x,y,z);
intrpBcore = R.interpolateMagneticFlux(X,Y,Z);
```

Reshape `intrpBcore.Bx`, `intrpBcore.By`, and `intrpBcore.Bz` and plot the magnetic flux density as a vector plot.

```
Bx = reshape(intrpBcore.Bx,size(X));
By = reshape(intrpBcore.By,size(Y));
Bz = reshape(intrpBcore.Bz,size(Z));

quiver3(X,Y,Z,Bx,By,Bz,"Color","r")
hold on
pdegplot(coreGm,"FaceAlpha",0.2);
```



2-D Axisymmetric Model of Coil with Core

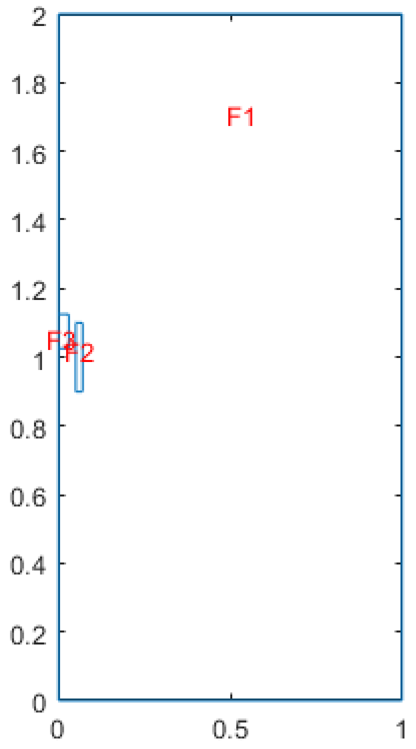
Now, simplify this 3-D problem to 2-D using the symmetry around the axis of rotation.

First, create the geometry. The axisymmetric section consists of two small rectangular regions (the core and coil) located within a large rectangular region (air).

```
R1 = [3,4,0.0,1,1,0.0,0,0,2,2]';
R2 = [3,4,0,0.03,0.03,0,1.025,1.025,1.125,1.125]';
R3 = [3,4,0.05,0.07,0.07,0.05,0.90,0.90,1.10,1.10]';
ns = char('R1','R2','R3');
sf = 'R1+R2+R3';
gdm = [R1, R2, R3];
g = decsg(gdm,sf,ns');
```

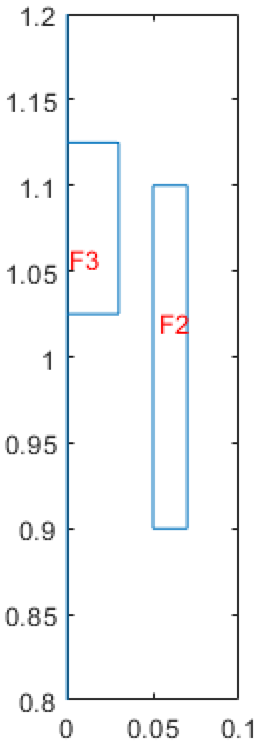
Plot the geometry with the face labels.

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



Zoom in to see the face labels on the core and coil.

```
figure
pdegplot(g,'FaceLabels','on')
axis([0 0.1 0.8 1.2])
```



Create an electromagnetic model for axisymmetric magnetostatic analysis and assign the geometry.

```
model2D = createpde('electromagnetic','magnetostatic-axisymmetric');
geometryFromEdges(model2D,g);
```

Specify the vacuum permeability value in the SI system of units.

```
model2D.VacuumPermeability = 1.2566370614E-6;
```

Specify a relative permeability of 1 for all domains.

```
electromagneticProperties(model2D,'RelativePermeability',1);
```

Now specify the large relative permeability of the core.

```
electromagneticProperties(model2D,'RelativePermeability',10000, ...
                                  'Face',3);
```

Specify the current density in the coil. For an axisymmetric model, use the constant current value.

```
electromagneticSource(model2D,'CurrentDensity',5E6,'Face',2);
```

Assign zero magnetic potential on the outer edges of the air domain as the boundary condition.

```
electromagneticBC(model2D,'MagneticPotential',0,'Edge',[2 8]);
```

Generate a mesh.

```
generateMesh(model2D,'Hmin',0.0004,'Hgrad',2,'Hmax',0.008);
```

Solve the model.

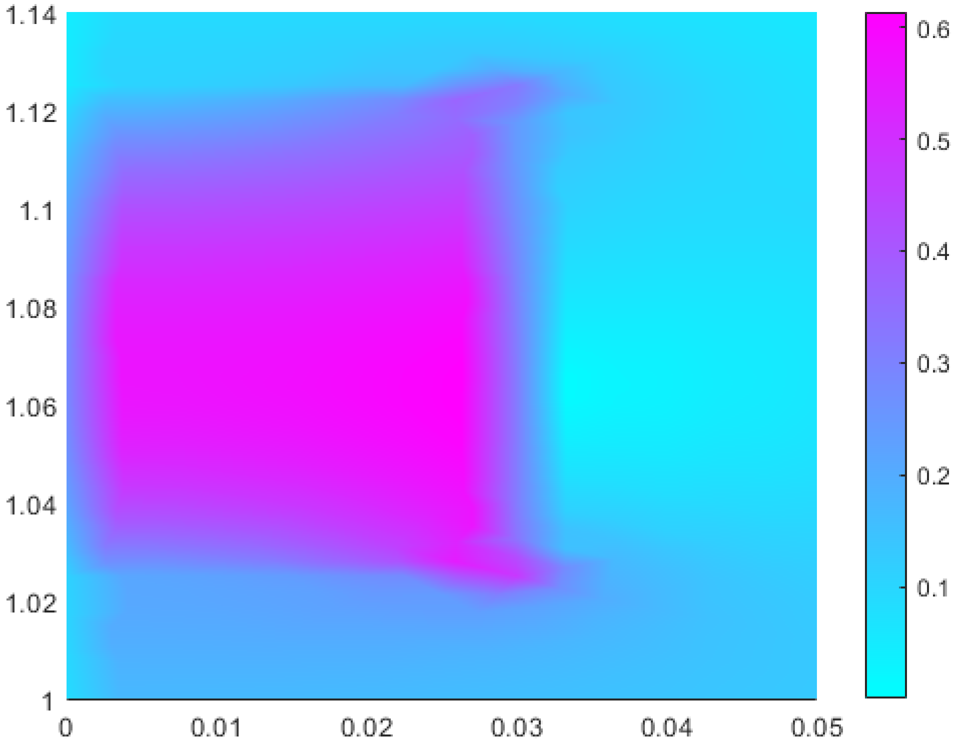
```
R = solve(model2D);
```

Find the magnitude of the flux density.

```
Bmag = sqrt(R.MagneticFluxDensity.Bx.^2 + ...
            R.MagneticFluxDensity.By.^2);
```

Plot the magnitude of the flux density on the core and coil.

```
pdeplot(model2D,'XYData',Bmag)
xlim([0,0.05]);
ylim([1.0,1.14])
```



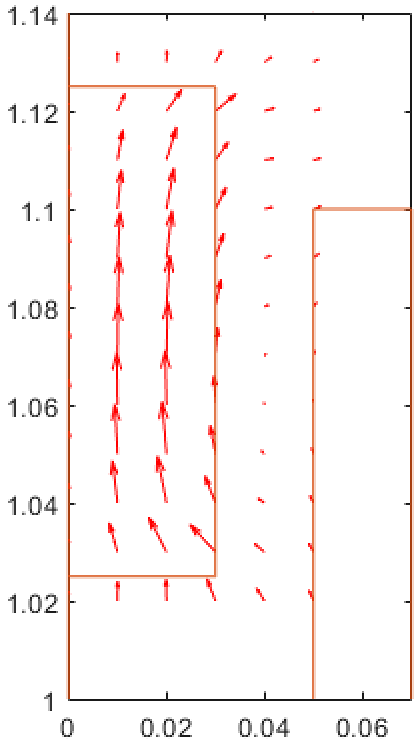
Interpolate the flux to a grid covering the portion of the geometry near the core.

```
x = 0:0.01:0.05;
y = 1.02:0.01:1.14;
[X,Y] = meshgrid(x,y);
intrpBcore = R.interpolateMagneticFlux(X,Y);
```

Reshape `intrpBcore.Bx` and `intrpBcore.By` and plot the magnetic flux density as a vector plot.

```
Bx = reshape(intrpBcore.Bx,size(X));
By = reshape(intrpBcore.By,size(Y));

quiver(X,Y,Bx,By,'Color','r')
```

```
hold on
pdegplot(model2D);
xlim([0,0.07]);
ylim([1.0,1.14])
```



Function Defining Current Density in Coil for 3-D Model

```
function f3D = windingCurrent3D(region,~)
[TH,~,~] = cart2pol(region.x,region.y,region.z);
f3D = -5E6*[sin(TH); -cos(TH); zeros(size(TH))];
end
```

References

[1] Thierry Lubin, Kévin Berger, Abderrezak Rezzoug. "Inductance and Force Calculation for Axisymmetric Coil Systems Including an Iron Core of Finite Length." *Progress In Electromagnetics Research B, EMW Publishing* 41 (2012): 377-396. https://hal.archives-ouvertes.fr/hal-00711310.

Linear Elasticity Equations

Summary of the Equations of Linear Elasticity

The stiffness matrix of linear elastic isotropic material contains two parameters:

- E, Young's modulus (elastic modulus)
- ν, Poisson's ratio

Define the following quantities.

σ = stress
f = body force
ε = strain
u = displacement

The equilibrium equation is

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

The linearized, small-displacement strain-displacement relationship is

$$\varepsilon = \frac{1}{2}\left(\nabla u + \nabla u^T\right)$$

The balance of angular momentum states that stress is symmetric:

$$\sigma_{ij} = \sigma_{ji}$$

The Voigt notation for the constitutive equation of the linear isotropic model is

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2\nu \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \varepsilon_{23} \\ \varepsilon_{13} \\ \varepsilon_{12} \end{bmatrix}$$

The expanded form uses all the entries in σ and ε takes symmetry into account.

$$\begin{bmatrix}\sigma_{11}\\ \sigma_{12}\\ \sigma_{13}\\ \sigma_{21}\\ \sigma_{22}\\ \sigma_{23}\\ \sigma_{31}\\ \sigma_{32}\\ \sigma_{33}\end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)}\begin{bmatrix}1-\nu & 0 & 0 & 0 & \nu & 0 & 0 & 0 & \nu\\ \bullet & 1-2\nu & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ \bullet & \bullet & 1-2\nu & 0 & 0 & 0 & 0 & 0 & 0\\ \bullet & \bullet & \bullet & 1-2\nu & 0 & 0 & 0 & 0 & 0\\ \bullet & \bullet & \bullet & \bullet & 1-\nu & 0 & 0 & 0 & \nu\\ \bullet & \bullet & \bullet & \bullet & \bullet & 1-2\nu & 0 & 0 & 0\\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-2\nu & 0 & 0\\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-2\nu & 0\\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & 1-\nu\end{bmatrix}\begin{bmatrix}\varepsilon_{11}\\ \varepsilon_{12}\\ \varepsilon_{13}\\ \varepsilon_{21}\\ \varepsilon_{22}\\ \varepsilon_{23}\\ \varepsilon_{31}\\ \varepsilon_{32}\\ \varepsilon_{33}\end{bmatrix} \tag{3-1}$$

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

3D Linear Elasticity Problem

The toolbox form for the equation is

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

But the equations in the summary do not have ∇u alone, it appears together with its transpose:

$$\varepsilon = \frac{1}{2}\left(\nabla u + \nabla u^T\right)$$

It is a straightforward exercise to convert this equation for strain ε to ∇u. In column vector form,

$$\nabla u = \begin{bmatrix}\partial u_x/\partial x\\ \partial u_x/\partial y\\ \partial u_x/\partial z\\ \partial u_y/\partial x\\ \partial u_y/\partial y\\ \partial u_y/\partial z\\ \partial u_z/\partial x\\ \partial u_z/\partial y\\ \partial u_z/\partial z\end{bmatrix}$$

Therefore, you can write the strain-displacement equation as

$$\varepsilon = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \nabla u \equiv A \nabla u$$

where A stands for the displayed matrix. So rewriting “Equation 3-1”, and recalling that • means an entry is symmetric, you can write the stiffness tensor as

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

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

Make the definitions

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

and the equation becomes

$$\sigma = \begin{bmatrix} 2\mu+\lambda & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & \lambda \\ 0 & \mu & 0 & \mu & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mu & 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & \mu & 0 & \mu & 0 & 0 & 0 & 0 & 0 \\ \lambda & 0 & 0 & 0 & 2\mu+\lambda & 0 & 0 & 0 & \lambda \\ 0 & 0 & 0 & 0 & 0 & \mu & 0 & \mu & 0 \\ 0 & 0 & \mu & 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu & 0 & \mu & 0 \\ \lambda & 0 & 0 & 0 & \lambda & 0 & 0 & 0 & 2\mu+\lambda \end{bmatrix} \nabla u \equiv c\nabla u$$

If you are solving a 3-D linear elasticity problem by using `PDEModel` instead of `StructuralModel`, use the `elasticityC3D(E,nu)` function (included in your software) to obtain the `c` coefficient. This function uses the linearized, small-displacement assumption for an isotropic material. For examples that use this function, see `StationaryResults`.

Plane Stress

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. For plane stress, $\sigma_{13} = \sigma_{23} = \sigma_{31} = \sigma_{32} = \sigma_{33} = 0$. Assuming isotropic conditions, the Hooke's law for plane stress gives the following strain-stress relation:

$$\begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ 2\varepsilon_{12} \end{bmatrix} = \frac{1}{E}\begin{bmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & 2+2\nu \end{bmatrix}\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix}$$

Inverting this equation, obtain the stress-strain relation:

$$\begin{pmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{pmatrix} = \frac{E}{1-\nu^2}\begin{pmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{pmatrix}\begin{pmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ 2\varepsilon_{12} \end{pmatrix}$$

Convert the equation for strain ε to ∇u.

$$\varepsilon = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \nabla u \equiv A\nabla u$$

Now you can rewrite the stiffness matrix as

$$\begin{bmatrix}\sigma_{11}\\ \sigma_{12}\\ \sigma_{21}\\ \sigma_{22}\end{bmatrix} = \begin{bmatrix}\dfrac{E}{1-\nu^2} & 0 & 0 & \dfrac{E\nu}{1-\nu^2}\\ 0 & \dfrac{E}{2(1+\nu)} & \dfrac{E}{2(1+\nu)} & 0\\ 0 & \dfrac{E}{2(1+\nu)} & \dfrac{E}{2(1+\nu)} & 0\\ \dfrac{E\nu}{1-\nu^2} & 0 & 0 & \dfrac{E}{1-\nu^2}\end{bmatrix}\nabla u = \begin{bmatrix}\dfrac{2\mu(\mu+\lambda)}{2\mu+\lambda} & 0 & 0 & \dfrac{2\lambda\mu}{2\mu+\lambda}\\ 0 & \mu & \mu & 0\\ 0 & \mu & \mu & 0\\ \dfrac{2\lambda\mu}{2\mu+\lambda} & 0 & 0 & \dfrac{2\mu(\mu+\lambda)}{2\mu+\lambda}\end{bmatrix}\nabla u$$

Plane Strain

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.

For plane strain, $\varepsilon_{13} = \varepsilon_{23} = \varepsilon_{31} = \varepsilon_{32} = \varepsilon_{33} = 0$. Assuming isotropic conditions, the stress-strain relation can be written as follows:

$$\begin{pmatrix}\sigma_{11}\\ \sigma_{22}\\ \sigma_{12}\end{pmatrix} = \frac{E}{(1+\nu)(1-2\nu)}\begin{pmatrix}1-\nu & \nu & 0\\ \nu & 1-\nu & 0\\ 0 & 0 & \dfrac{1-2\nu}{2}\end{pmatrix}\begin{pmatrix}\varepsilon_{11}\\ \varepsilon_{22}\\ 2\varepsilon_{12}\end{pmatrix}$$

Convert the equation for strain ε to ∇u.

$$\varepsilon = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \frac{1}{2} & \frac{1}{2} & 0\\ 0 & \frac{1}{2} & \frac{1}{2} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\nabla u \equiv A\nabla u$$

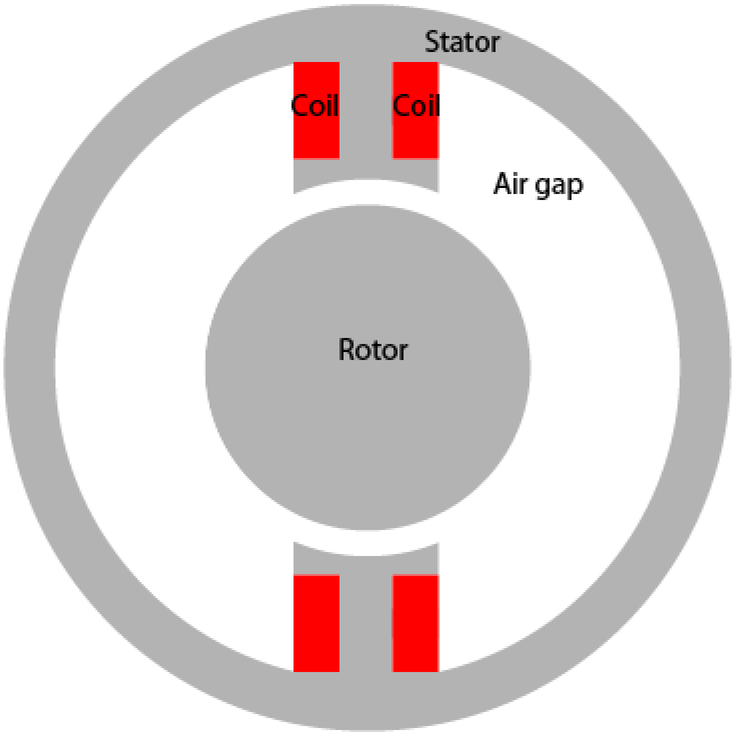
Now you can rewrite the stiffness matrix as

$$\begin{bmatrix}\sigma_{11}\\ \sigma_{12}\\ \sigma_{21}\\ \sigma_{22}\end{bmatrix} = \begin{bmatrix}\dfrac{E(1-\nu)}{(1+\nu)(1-2\nu)} & 0 & 0 & \dfrac{E\nu}{(1+\nu)(1-2\nu)}\\ 0 & \dfrac{E}{2(1+\nu)} & \dfrac{E}{2(1+\nu)} & 0\\ 0 & \dfrac{E}{2(1+\nu)} & \dfrac{E}{2(1+\nu)} & 0\\ \dfrac{E\nu}{(1+\nu)(1-2\nu)} & 0 & 0 & \dfrac{E(1-\nu)}{(1+\nu)(1-2\nu)}\end{bmatrix}\nabla u = \begin{bmatrix}2\mu+\lambda & 0 & 0 & \lambda\\ 0 & \mu & \mu & 0\\ 0 & \mu & \mu & 0\\ \lambda & 0 & 0 & 2\mu+\lambda\end{bmatrix}\nabla u$$

Magnetic Field in Two-Pole Electric Motor

Find the static magnetic field induced by the stator windings in a two-pole electric motor. Assuming that the motor is long and the 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, made of transformer steel
- The air gap between the stator and the rotor
- The armature copper coil carrying the DC current



The magnetic permeability of air and of copper are both close to the magnetic permeability of a vacuum, $\mu = \mu_0$. The magnetic permeability of the stator and the rotor is $\mu = 5000\mu_0$. 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 the y-axis and antisymmetric with respect to the x-axis. Therefore, you can limit the domain to $x \geq 0$, $y \geq 0$, with the default boundary condition

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

on the x-axis and the boundary condition $A = 0$ on the y-axis. Because the field outside the motor is negligible, you can use the boundary condition $A = 0$ on the exterior boundary.

First, create the geometry in the PDE Modeler app. The geometry of this electric motor is a union of five circles and two rectangles. To draw the geometry, enter the following commands in the MATLAB Command Window:

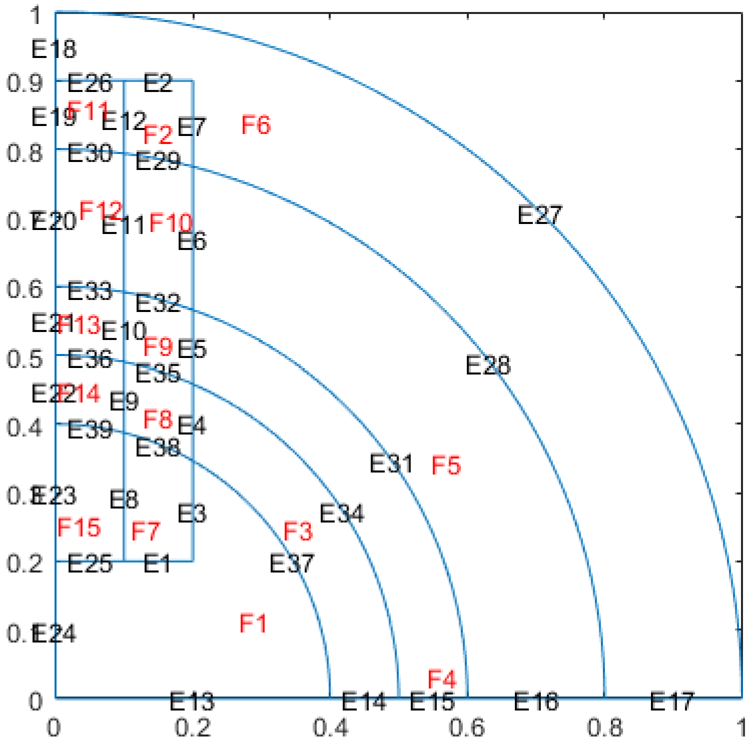
```
pdecirc(0,0,1,'C1')
pdecirc(0,0,0.8,'C2')
pdecirc(0,0,0.6,'C3')
pdecirc(0,0,0.5,'C4')
pdecirc(0,0,0.4,'C5')
pderect([-0.2 0.2 0.2 0.9],'R1')
pderect([-0.1 0.1 0.2 0.9],'R2')
pderect([0 1 0 1],'SQ1')
```

Reduce the geometry to the first quadrant by intersecting it with a square. To do this, enter `(C1+C2+C3+C4+C5+R1+R2)*SQ1` in the **Set formula** field.

From the PDE Modeler app, export the geometry description matrix, set formula, and name-space matrix to the MATLAB workspace by selecting **Export Geometry Description, Set Formula, Labels...** from the **Draw** menu.

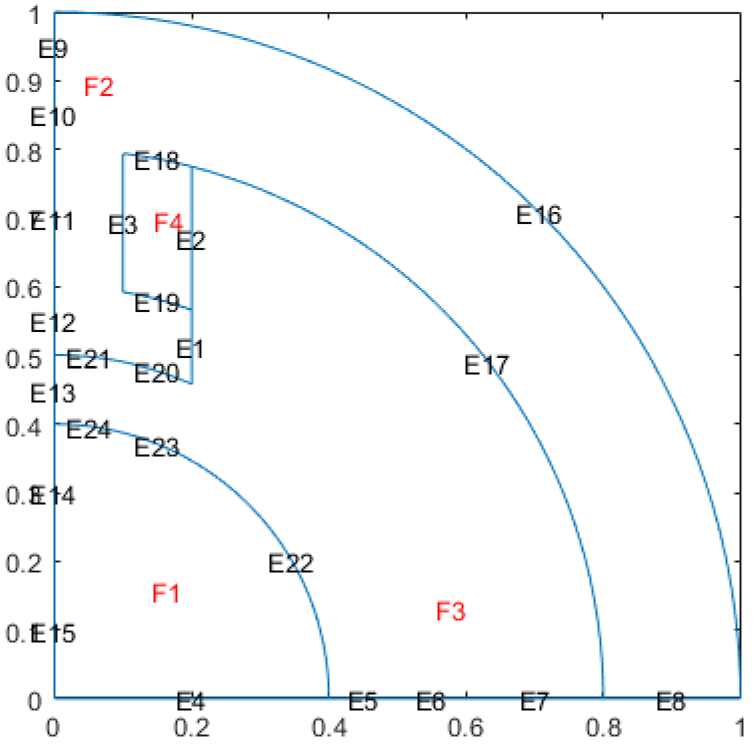
In the MATLAB Command Window, use the `decsg` function to decompose the exported geometry into minimal regions. This command creates an `AnalyticGeometry` object `d1`. Plot the geometry `d1`.

```
[d1,bt1] = decsg(gd,sf,ns);
pdegplot(d1,"EdgeLabels","on","FaceLabels","on")
```



Remove unnecessary edges using the `csgdel` function. Specify the edges to delete as a vector of edge IDs. Plot the resulting geometry.

```
[d2,bt2] = csgdel(d1,bt1,[1 3 8 25 7 2 12 26 30 33 4 9 34 10 31]);
pdegplot(d2,"EdgeLabels","on","FaceLabels","on")
```



Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Include the geometry in the model.

```
geometryFromEdges(emagmodel,d2);
```

Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the air gap and copper coil, which correspond to the faces 3 and 4 of the geometry.

```
electromagneticProperties(emagmodel,"RelativePermeability",1, ...
                                    "Face",[3 4]);
```

Specify the relative permeability of the stator and the rotor, which correspond to the faces 1 and 2 of the geometry.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000, ...
                                    "Face",[1 2]);
```

Specify the current density in the coil.

```
electromagneticSource(emagmodel,"CurrentDensity",10,"Face",4);
```

Apply the zero magnetic potential condition to all boundaries, except the edges along the *x*-axis. The edges along the *x*-axis retain the default boundary condition.

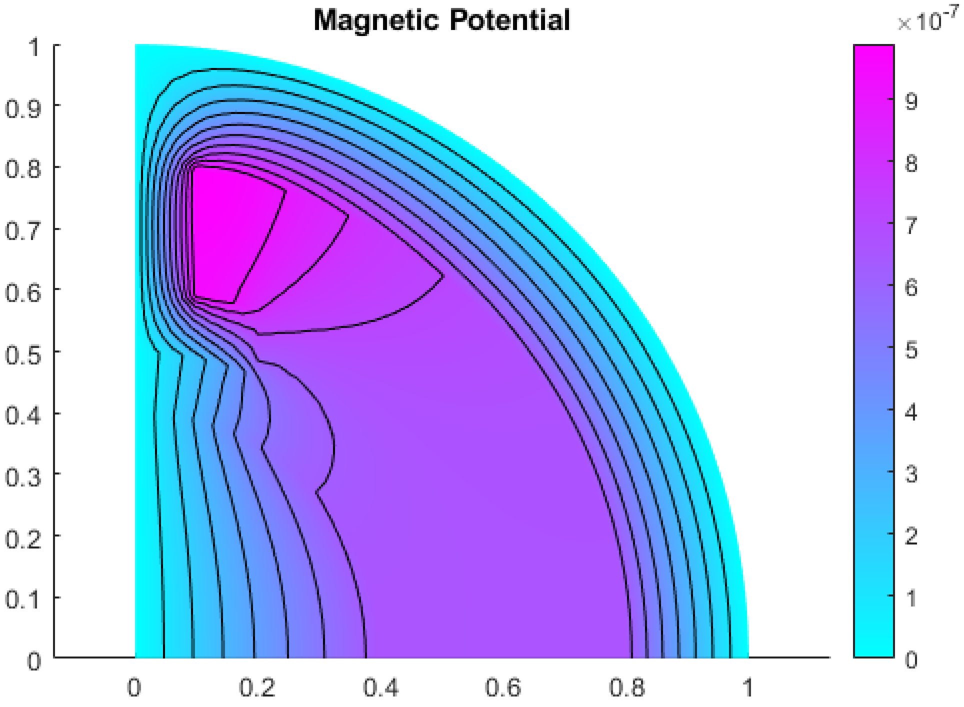
```
electromagneticBC(emagmodel,"MagneticPotential",0,...
                  "Edge",[16 9 10 11 12 13 14 15]);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

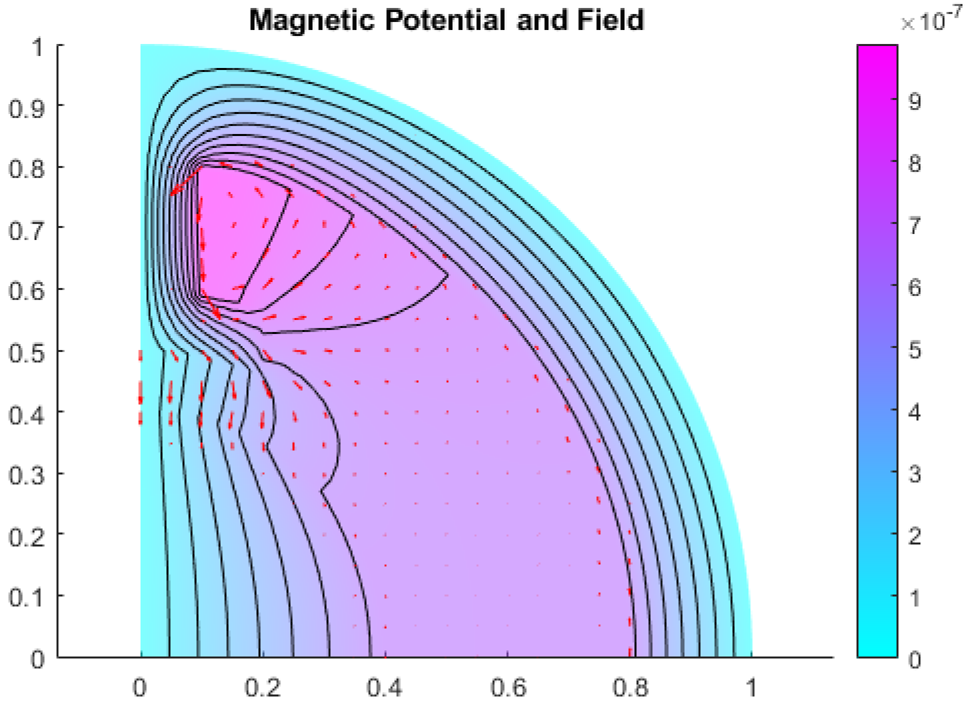
Solve the model and plot the magnetic potential. Use the `Contour` parameter to display equipotential lines.

```
R = solve(emagmodel);
figure
pdeplot(emagmodel,"XYData",R.MagneticPotential,"Contour","on")
title "Magnetic Potential"
```



Add the magnetic field data to the plot. Use the `FaceAlpha` parameter to make the quiver plot for magnetic field more visible.

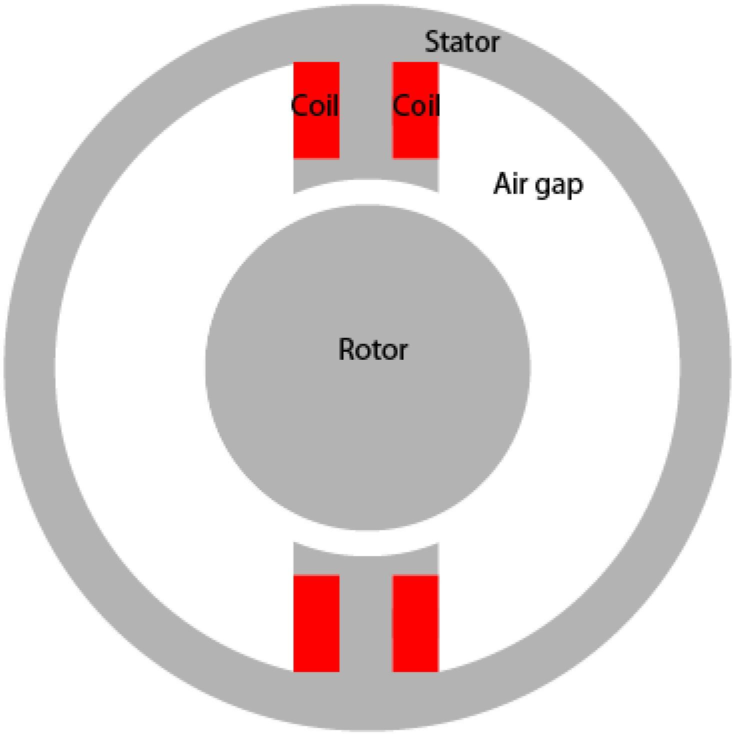
```
figure
pdeplot(emagmodel,"XYData",R.MagneticPotential, ...
                  "FlowData",[R.MagneticField.Hx, ...
                              R.MagneticField.Hy], ...
                  "Contour","on", ...
                  "FaceAlpha",0.5)
title "Magnetic Potential and Field"
```


Magnetic Potential and Field
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1
×10^-7
9
8
7
6
5
4
3
2
1
0

Magnetic Field in Two-Pole Electric Motor: PDE Modeler App

Find the static magnetic field induced by the stator windings in a two-pole electric motor. The example uses the PDE Modeler 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, μ 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

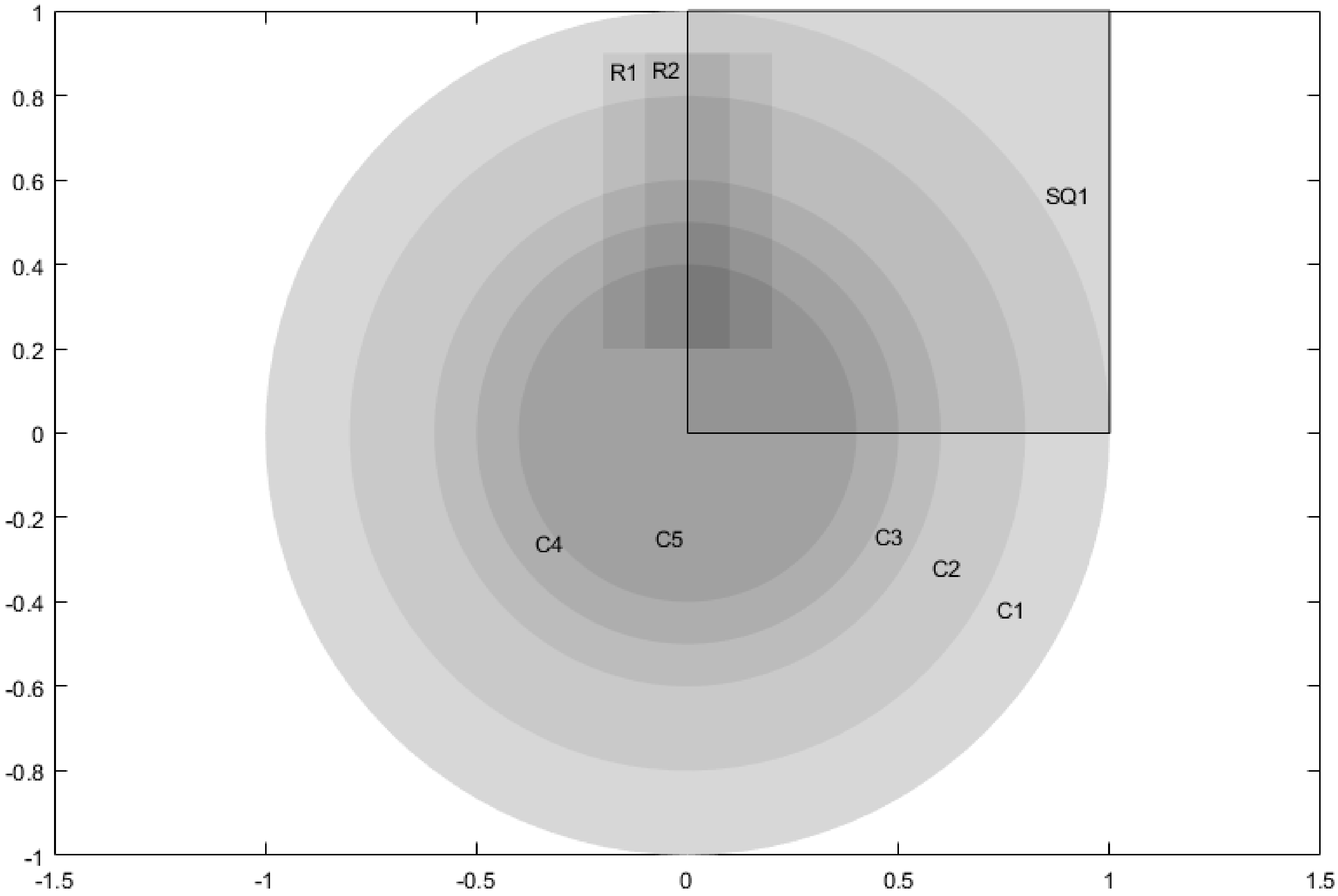
$$\mathbf{n} \cdot \left(\frac{1}{\mu}\nabla A\right) = 0$$

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 Modeler app, follow these steps:

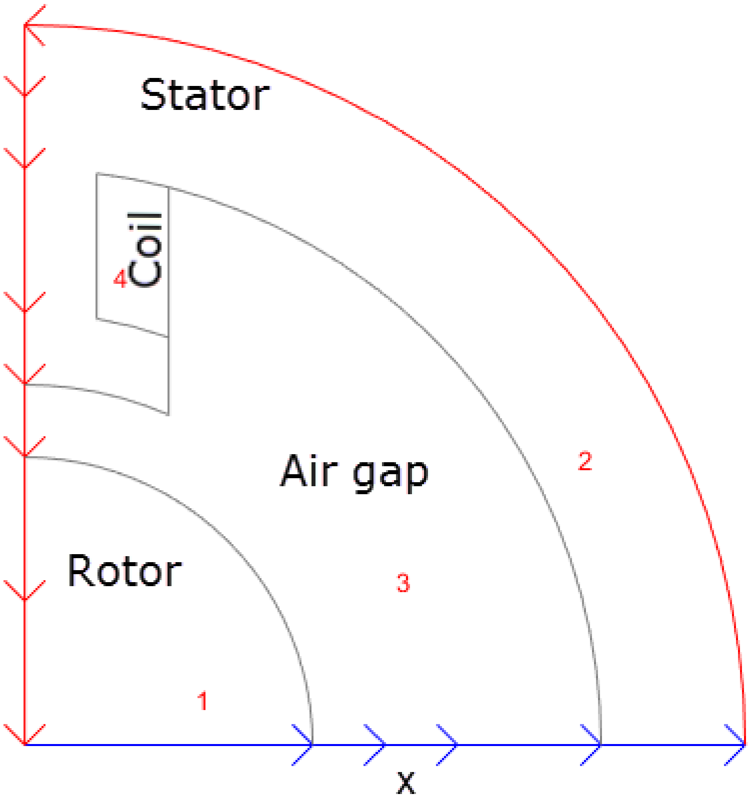
1. Set the *x*-axis limits to `[-1.5 1.5]` and the *y*-axis limits to `[-1 1]`. To do this, select **Options > Axes Limits** and set the corresponding ranges.
2. Set the application mode to **Magnetostatics**.
3. Create the geometry. The geometry of this electric motor is complex. The model is a union 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 Modeler 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: μ = `4*pi*10^(-7)` H/m, `J = 10` A/m^2.
- Stator and rotor: μ = `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: μ = `4*pi*10^(-7)` H/m, `J = 0`.

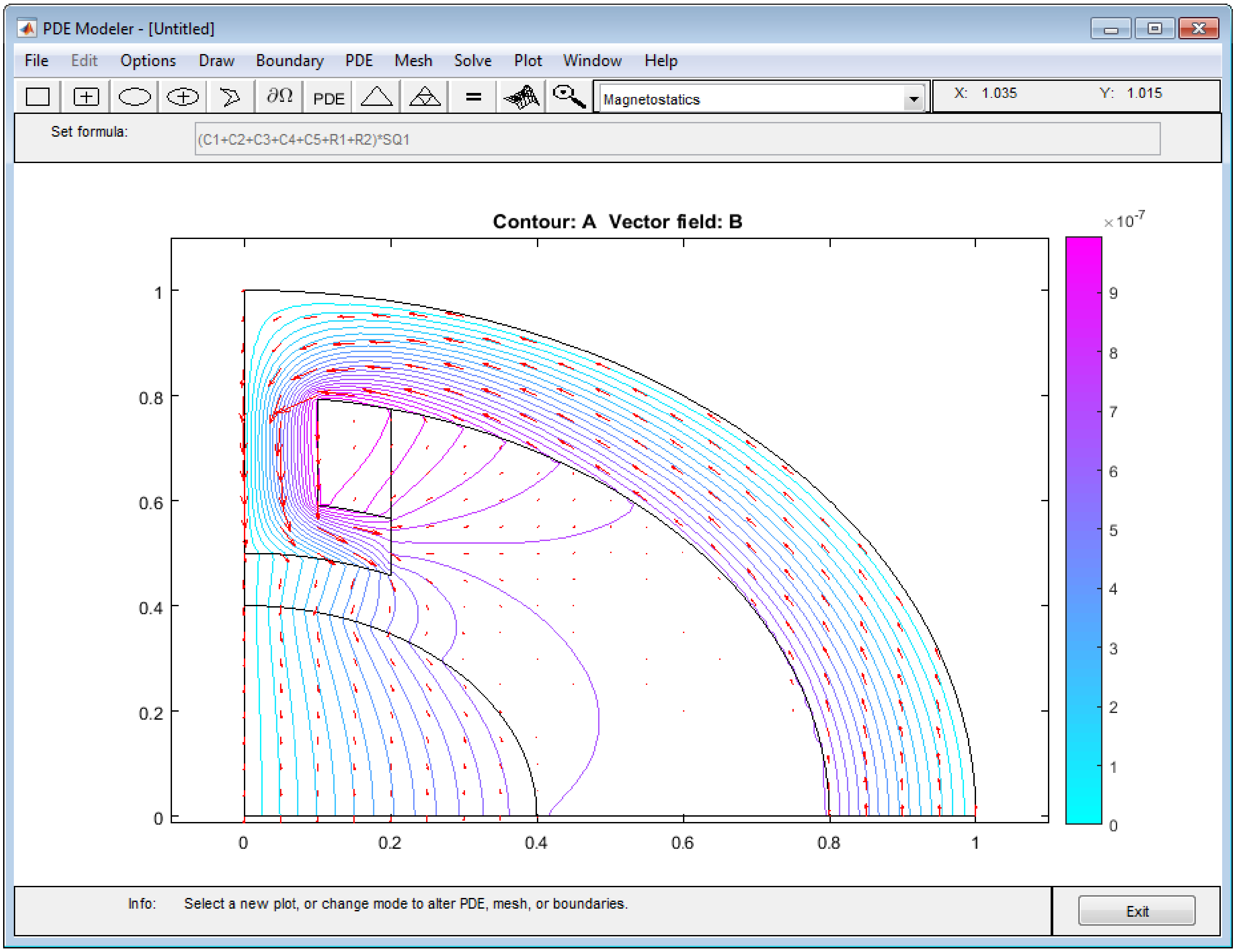
8 Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.

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.



Magnetostatics

Set formula: (C1+C2+C3+C4+C5+R1+R2)*SQ1

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

Helmholtz Equation on Disk with Square Hole

This example shows how to solve a Helmholtz equation using the general `PDEModel` container and the `solvepde` function. For the electromagnetic workflow that uses `ElectromagneticModel` and familiar domain-specific language, see “Scattering Problem” on page 3-232.

Solve a simple scattering problem, where you compute the waves reflected by a square object illuminated by incident waves that are coming from the left. For this problem, assume an infinite horizontal membrane subjected to small vertical displacements U. The membrane is fixed at the object boundary. The medium is homogeneous, and the phase velocity (propagation speed) of a wave, α, is constant. The wave equation is

$$\frac{\partial^2 U}{\partial t^2} - \alpha^2 \vartriangle U = 0$$

The solution U is the sum of the incident wave V and the reflected wave R:

$$U = V + R$$

When the illumination is harmonic in time, you can compute the field by solving a single steady problem. Assume that the incident wave is a plane wave traveling in the -x direction:

$$V(x, y, t) = e^{i(-kx - \omega t)} = e^{-ikx} \cdot e^{-i\omega t}$$

The reflected wave can be decomposed into spatial and time components:

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

Now you can rewrite the wave equation as the Helmholtz equation for the spatial component of the reflected wave with the wave number $k = \omega/\alpha$:

$$-\Delta r - k^2 r = 0$$

The Dirichlet boundary condition for the boundary of the object is $U = 0$, or in terms of the incident and reflected waves, $R = -V$. For the time-harmonic solution and the incident wave traveling in the -x direction, you can write this boundary condition as follows:

$$r(x, y) = \; -e^{-ikx}$$

The reflected wave R travels outward from the object. The condition at the outer computational boundary must allow waves to pass without reflection. Such conditions are usually called nonreflecting. As $\left|\vec{x}\right|$ approaches infinity, R approximately satisfies the one-way wave equation

$$\frac{\partial R}{\partial t} + \alpha \vec{\xi} \cdot \nabla R = 0$$

This equation considers only the waves moving in the positive ξ-direction. Here, ξ is the radial distance from the object. With the time-harmonic solution, this equation turns into the generalized Neumann boundary condition

$$\vec{\xi} \cdot \nabla r - ikr = 0$$

To solve the scattering problem using the programmatic workflow, first create a PDE model with a single dependent variable.

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

Specify the variables that define the problem:

- `g`: A geometry specification function. For more information, see the documentation section “Parametrized Function for 2-D Geometry Creation” on page 2-22 and the code for `scatterg.m`.
- `k`, `c`, `a`, `f`: The coefficients and inhomogeneous term.

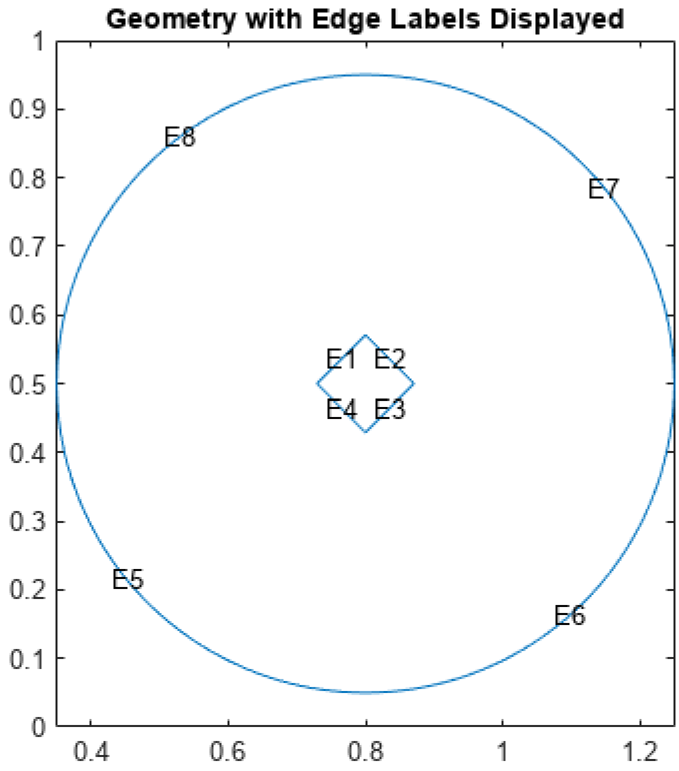
```
g = @scatterg;
k = 60;
c = 1;
a = -k^2;
f = 0;
```

Convert the geometry and append it to the 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")
ylim([0,1])
```



Apply the boundary conditions.

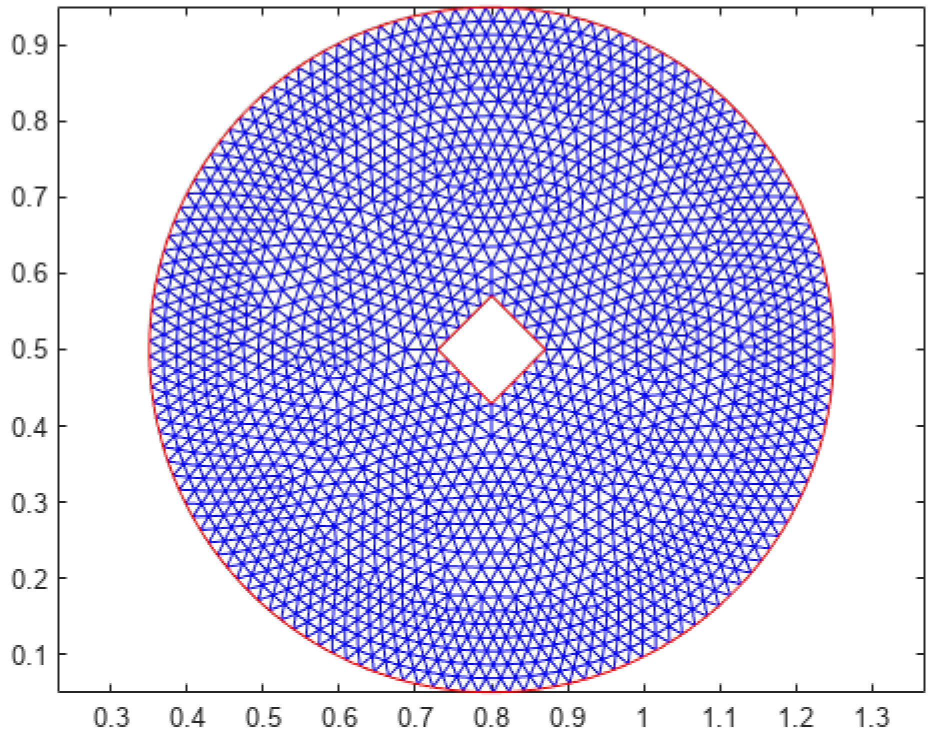
```
bOuter = applyBoundaryCondition(model,"neumann","Edge",(5:8), ...
                                      "g",0,"q",-60i);
innerBCFunc = @(loc,state)-exp(-1i*k*loc.x);
bInner = applyBoundaryCondition(model,"dirichlet","Edge",(1:4), ...
                                      "u",innerBCFunc);
```

Specify the coefficients.

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

Generate a mesh.

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

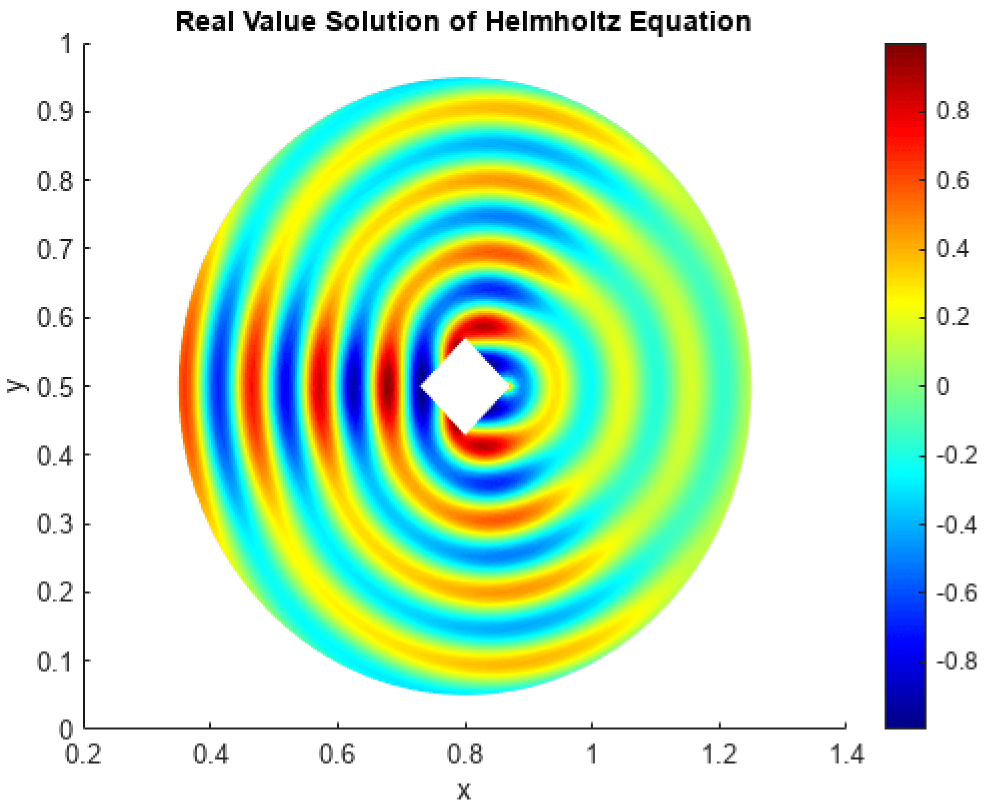


Solve for the complex amplitude. The real part of vector `u` stores an approximation to a real value solution of the Helmholtz equation.

```
result = solvepde(model);
u = result.NodalSolution;
```

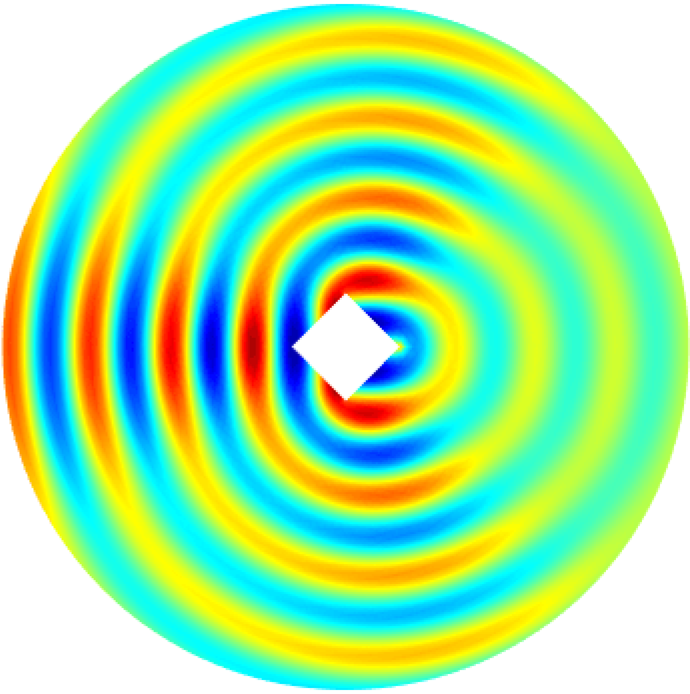
Plot the solution.

```
figure
pdeplot(model,"XYData",real(u),"Mesh","off");
colormap(jet)
xlabel("x")
ylabel("y")
title("Real Value Solution of Helmholtz Equation")
```



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

```
figure
m = 10;
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");
    colormap(jet)
    caxis([-maxu maxu]);
    axis tight
    ax = gca;
    ax.DataAspectRatio = [1 1 1];
    axis off
    M(j) = getframe;
end
```



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

Electrostatics and Magnetostatics

Maxwell's equations describe electrodynamics as:

$$\begin{aligned}\nabla\cdot\mathbf{D} &= \rho,\\ \nabla\cdot\mathbf{B} &= 0,\\ \nabla\times\mathbf{E} &= -\frac{\partial\mathbf{B}}{\partial t},\\ \nabla\times\mathbf{H} &= \mathbf{J}+\frac{\partial\mathbf{D}}{\partial t}.\end{aligned}$$

Here, **E** and **H** are the electric and magnetic field intensities, **D** and **B** are the electric and magnetic flux densities, and ρ and **J** are the electric charge and current densities.

Electrostatics

For electrostatic problems, Maxwell's equations simplify to this form:

$$\begin{aligned}&\nabla\cdot\mathbf{D} = \nabla\cdot(\varepsilon\mathbf{E}) = \rho,\\ &\nabla\times\mathbf{E} = 0,\end{aligned}$$

where ε is the electrical permittivity of the material.

Because the electric field **E** is the gradient of the electric potential V, $\mathbf{E} = -\nabla V$., the first equation yields this PDE:

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

For electrostatic problems, Dirichlet boundary conditions specify the electric potential V on the boundary.

Magnetostatics

For magnetostatic problems, Maxwell's equations simplify to this form:

$$\begin{aligned}&\nabla\cdot\mathbf{B} = 0,\\ &\nabla\times\mathbf{H} = \mathbf{J}+\frac{\partial(\varepsilon\mathbf{E})}{\partial t} = \mathbf{J}.\end{aligned}$$

Because $\nabla\cdot\mathbf{B} = 0$, there exists a magnetic vector potential **A**, such that $\mathbf{B} = \nabla\times\mathbf{A}$. For non-ferromagnetic materials, $\mathbf{B} = \mu\mathbf{H}$, where μ is the magnetic permeability of the material. Therefore,

$$\begin{aligned}&\mathbf{H} = \mu^{-1}\nabla\times\mathbf{A},\\ &\nabla\times\left(\mu^{-1}\nabla\times\mathbf{A}\right) = \mathbf{J}.\end{aligned}$$

Using the identity

$$\nabla\times(\nabla\times\mathbf{A}) = \nabla(\nabla\cdot\mathbf{A}) - \nabla^2\mathbf{A}$$

and the Coulomb gauge $\nabla\cdot\mathbf{A} = 0$, simplify the equation for **A** in terms of **J** to this PDE:

$$-\nabla^2\mathbf{A} = -\nabla\cdot\nabla\mathbf{A} = \mu\mathbf{J}.$$

For magnetostatic problems, Dirichlet boundary conditions specify the magnetic potential **A** on the boundary.

Magnetostatics with Permanent Magnets

In the case of a permanent magnet, the constitutive relation between **B** and **H** includes the magnetization **M**:

$$\mathbf{B} = \mu\mathbf{H} + \mu_0\mathbf{M}\,.$$

Here, $\mu = \mu_0\mu_r$, where μ_r is the relative magnetic permeability of the material, and μ_0 is the vacuum permeability.

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

$$\mathbf{H} = \frac{1}{\mu_0\mu_r}\mathbf{B} - \frac{1}{\mu_r}\mathbf{M},$$

$$\nabla \times \mathbf{H} = \nabla \times \left(\frac{1}{\mu_0\mu_r} \times \mathbf{A} - \frac{1}{\mu_r}\mathbf{M}\right) = \mathbf{J}\,.$$

The equation for **A** in terms of the current density **J** and magnetization **M** is

$$\nabla \times \left(\frac{1}{\mu_r\mu_0}\nabla \times \mathbf{A}\right) = \mathbf{J} + \nabla \times \left(\frac{1}{\mu_r}\mathbf{M}\right).$$

DC Conduction

Direct current electrical conduction problems, such as electrolysis and computation of resistances of grounding plates, involve a steady current passing through a conductive medium. The current density **J** is related to the electric field **E** as follows:

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

where σ is the electric conductivity.

The electric field **E** is the gradient of the electric potential V:

$$\mathbf{E} = -\nabla V.$$

Combining this definition with the homogeneous continuity equation

$$\nabla \cdot \mathbf{J} = -\frac{\partial\rho}{\partial t} = 0,$$

where ρ is the current density, yields this equation:

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

For DC conduction problems, Dirichlet boundary conditions specify the electric potential V on the boundary. The Neumann boundary conditions specify the surface current density, which is the value of the normal component of the current density ($\mathbf{n} \cdot (\sigma \nabla V)$) on a face for a 3-D geometry or an edge for a 2-D geometry.

Harmonic Electromagnetics

Maxwell's equations describe electrodynamics as:

$$\begin{aligned} \varepsilon \nabla \cdot \mathbf{E} &= \rho, \\ \nabla \cdot \mathbf{H} &= 0, \\ \nabla \times \mathbf{E} &= -\mu \frac{\partial \mathbf{H}}{\partial t}, \\ \nabla \times \mathbf{H} &= \mathbf{J} + \varepsilon \frac{\partial \mathbf{E}}{\partial t}. \end{aligned}$$

Here, **E** and **H** are the electric and magnetic fields, ε and μ are the electrical permittivity and magnetic permeability of the material, and ρ and **J** are the electric charge and current densities.

The time-harmonic electric and magnetic fields can be represented using these formulas:

$$\mathbf{E} = \widehat{\mathbf{E}} e^{i\omega t},$$

$$\mathbf{H} = \widehat{\mathbf{H}} e^{i\omega t}.$$

Accounting for the electric conductivity of the material and the applied current separately, you can represent the total electric current density as the sum of the current density $\sigma\mathbf{E}$ due to the electric field and the current density of the applied current: $\mathbf{J} = \sigma\mathbf{E} + \mathbf{J_a}$. Here, σ is the conductivity of the material. For a time-harmonic problem, the applied current can be defined as:

$$\mathbf{J_a} = \widehat{\mathbf{J}} e^{i\omega t}.$$

Maxwell’s equations for the electric field yield this equation:

$$-\nabla \times \left(\mu^{-1} \nabla \times \mathbf{E}\right) = \varepsilon \frac{\partial^2 \mathbf{E}}{\partial t^2} + \sigma \frac{\partial \mathbf{E}}{\partial t} + \frac{\partial \mathbf{J_a}}{\partial t}.$$

For the time-harmonic electric field and applied current, the derivative $\frac{\partial}{\partial t} = i\omega$, and the resulting equation is:

$$\nabla \times \left(\mu^{-1} \nabla \times \widehat{\mathbf{E}}\right) + \left(i\sigma\omega - \varepsilon\omega^2\right)\widehat{\mathbf{E}} = -i\omega\widehat{\mathbf{J}}.$$

Given an incident electric field $\mathbf{E}_\mathrm{i}$ and a scattered electric field $\mathbf{E}_\mathrm{s}$, you can compute the total electric field **E**. Due to linearity, it suffices to solve the equation for the scattered field with the boundary condition for the scattered field along the scattering object determined by

$$\mathbf{n} \times \mathbf{E_i} = -\mathbf{n} \times \mathbf{E_s}.$$

For the time-harmonic magnetic field and applied current, Maxwell's equations can be simplified under the assumption of zero conductivity to this form:

$$\nabla \times \left(\varepsilon^{-1} \nabla \times \widehat{\mathbf{H}}\right) - \mu\omega^2 \widehat{\mathbf{H}} = \nabla \times \left(\varepsilon^{-1} \widehat{\mathbf{J}}\right).$$

For the time-harmonic magnetic field, it suffices to solve the equation for the scattered field with the boundary condition for the scattered field along the scattering object determined by

$$\mathbf{n} \times \mathbf{H_i} = -\mathbf{n} \times \mathbf{H_s}.$$

Here, $\mathbf{H}_\mathrm{i}$ is an incident magnetic field, and $\mathbf{H}_\mathrm{s}$ is a scattered magnetic field.

Current Density Between Two Metallic Conductors

This example shows how to find the electric potential and the components of the current density between two circular metallic conductors. Two metallic conductors are placed on a brine-soaked blotting paper that serves as a plane, thin conductor. The physical model for this problem is DC conduction. The boundary conditions are:

- The electric potential $V = 1$ on the left circular conductor
- The electric potential $V = –1$ on the right circular conductor
- No surface current on the outer boundaries of the plane conductor

First, create a geometry consisting of a rectangle and two circles. Start by defining a rectangle and two circles.

```
R1 = [3;4
     -1.2;-1.2;1.2;1.2
     -0.6;0.6;0.6;-0.6];
C1 = [1;-0.6;0;0.3];
C2 = [1;0.6;0;0.3];
```

Append extra zeros to the circles so they have the same number of rows as the rectangle.

```
C1 = [C1;zeros(length(R1) - length(C1),1)];
C2 = [C2;zeros(length(R1) - length(C2),1)];
```

Combine the shapes into one matrix.

```
gd = [R1,C1,C2];
```

Create names for the rectangle and the circles, and specify the formula to create the geometry.

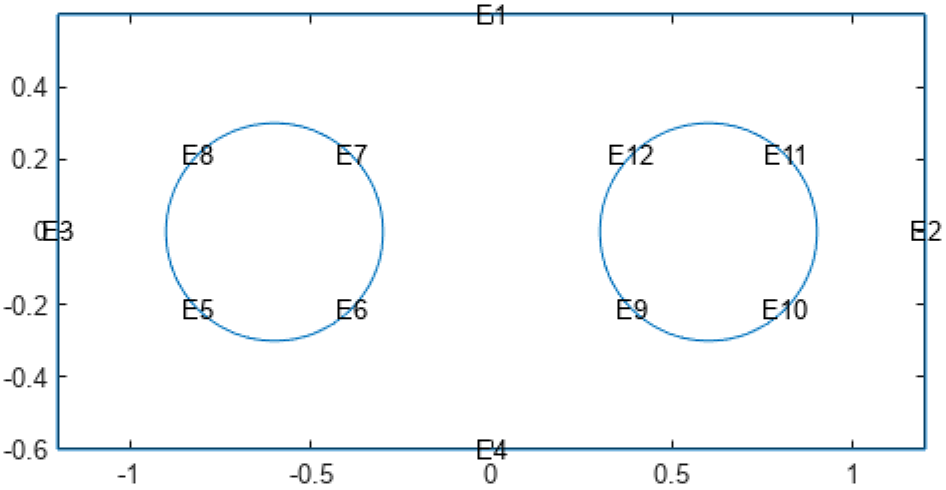
```
ns = char('R1','C1','C2');
ns = ns';
sf = 'R1 - (C1 + C2)';
g = decsg(gd,sf,ns);
```

Create an electromagnetic model for DC conduction analysis.

```
model = createpde("electromagnetic","conduction");
```

Include the geometry in the model, and plot it with the edge labels.

```
geometryFromEdges(model,g);
pdegplot(model,"EdgeLabels","on")
```



Specify the conductivity of the material as $\sigma = 1$.

```
electromagneticProperties(model,"Conductivity",1);
```

Specify the electric potential values on the left and right circular conductors.

```
electromagneticBC(model,"Edge",5:8,"Voltage",1);
electromagneticBC(model,"Edge",9:12,"Voltage",-1);
```

Specify the zero surface current density on the outer boundaries.

```
electromagneticBC(model,"Edge",1:4,"SurfaceCurrentDensity",0);
```

Generate the mesh.

```
generateMesh(model);
```

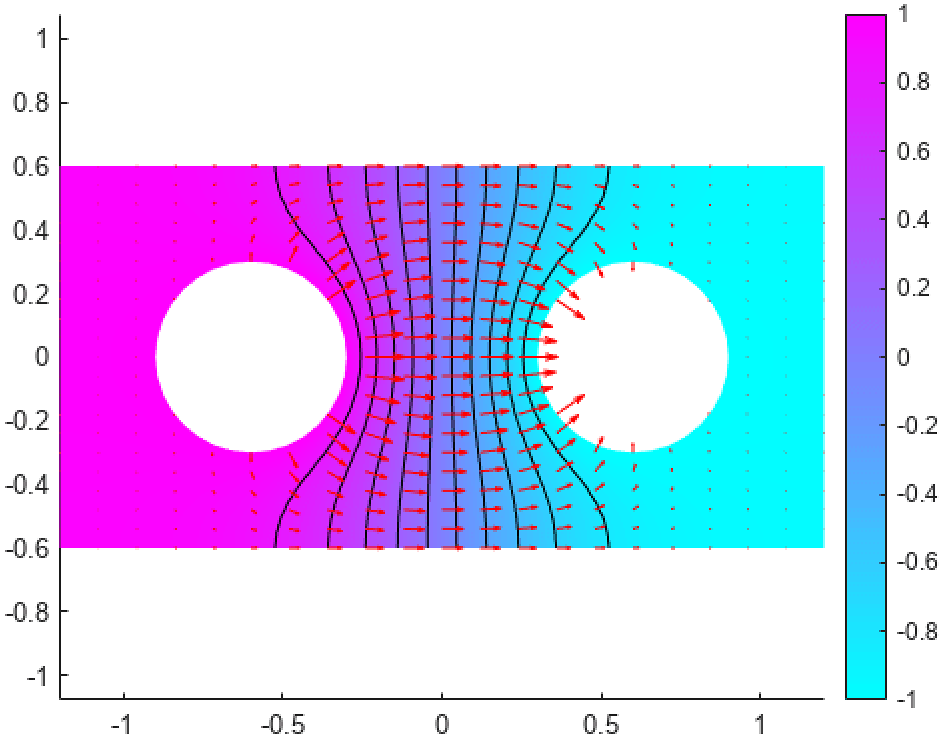
Solve the model.

```
R = solve(model)
```

```
R = 
  ConductionResults with properties:

    ElectricPotential: [989x1 double]
        ElectricField: [1x1 FEStruct]
       CurrentDensity: [1x1 FEStruct]
                 Mesh: [1x1 FEMesh]
```

Plot the resulting electric potential and current density, and display the equipotential lines. The current flows from the conductor with a positive potential to the conductor with a negative potential. The conductivity σ is isotropic, and the equipotential lines are orthogonal to the current lines.

```
pdeplot(model,"XYData",R.ElectricPotential, ...
              "Contour","on", ...
              "FlowData",[R.CurrentDensity.Jx,R.CurrentDensity.Jy])
axis equal
```



Skin Effect in Copper Wire with Circular Cross Section: PDE Modeler App

This example shows the *skin effect* when a wire with a circular cross section carries AC current. In a solid conductor, such as the wire, AC current travels near the surface of a wire and avoids the area close to the center of the wire. This effect is called the skin effect. The example uses the PDE Modeler app.

The Helmholtz equation

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

describes the propagation of plane electromagnetic waves in imperfect dielectrics and good conductors ($\sigma \gg \omega\varepsilon$). The coefficient of dielectricity is $\varepsilon = 8.8*10^{-12}$ F/m. The conductivity of copper is $\sigma = 57 * 10^6$ S/m. The magnetic permeability of copper is close to the magnetic permeability of a vacuum, $\mu = 4\pi*10^{-7}$ H/m. The $\omega^2\varepsilon$-term is negligible at the line frequency (50 Hz).

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

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

Here,

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

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.

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

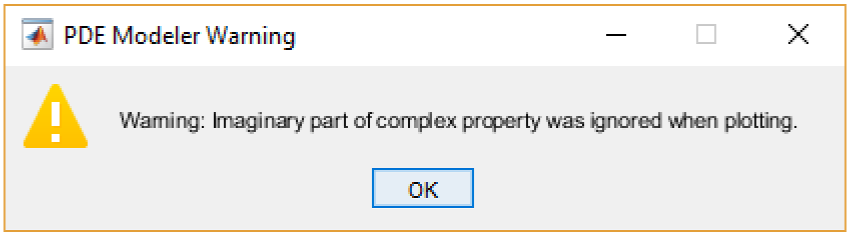
1. Draw a circle with a radius of 0.1. The circle represents a cross section of the conductor.

   ```
   pdecirc(0,0.05,0.1)
   ```
2. Set the x-axis limit to `[-0.2 0.2]` and the y-axis limit to `[-0.1 0.2]`. To do this, select **Options** > **Axes Limits** and set the corresponding ranges. Then select **Options** > **Axes Equal**.
3. Set the application mode to **AC Power Electromagnetics**.
4. Specify the Dirichlet boundary condition $E = J_S/\sigma = 1/\sigma$ for the boundary of the circle. To do this:
   - a. Switch to the boundary mode by selecting **Boundary** > **Boundary Mode**.
   - b. Select all boundaries by using **Edit** > **Select All**.
   - c. Select **Boundary** > **Specify Boundary Conditions**.
   - d. Specify `h = 1` and `r = 1/57E6`.
5. Specify the PDE coefficients. To do this, switch to the PDE mode by selecting **PDE** > **PDE Mode**. Then select **PDE** > **PDE Specification** or click the **PDE** button on the toolbar. Specify the following values:
   - Angular frequency `omega = 2*pi*50`

- Magnetic permeability `mu = 4*pi*1E-7`
- Conductivity `sigma = 57E6`
- Coefficient of dielectricity `epsilon = 8.8E-12`

6. Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.
7. Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.

   The solution of the AC power electromagnetics equation is complex. When plotting the solution, you get a warning message.

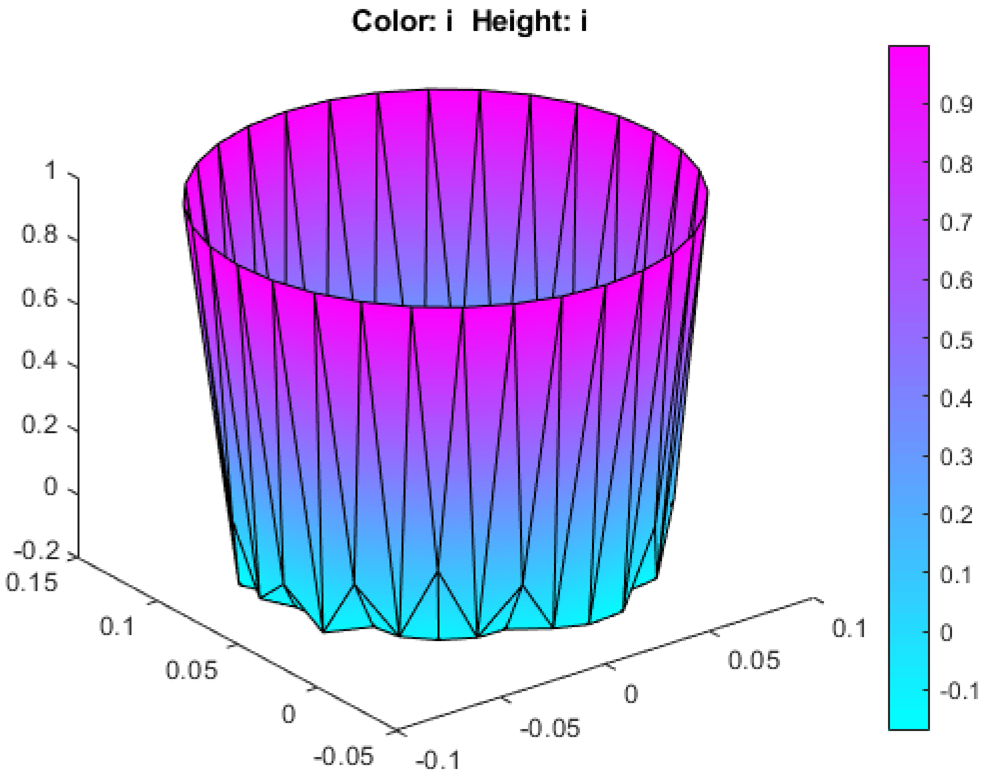


PDE Modeler Warning

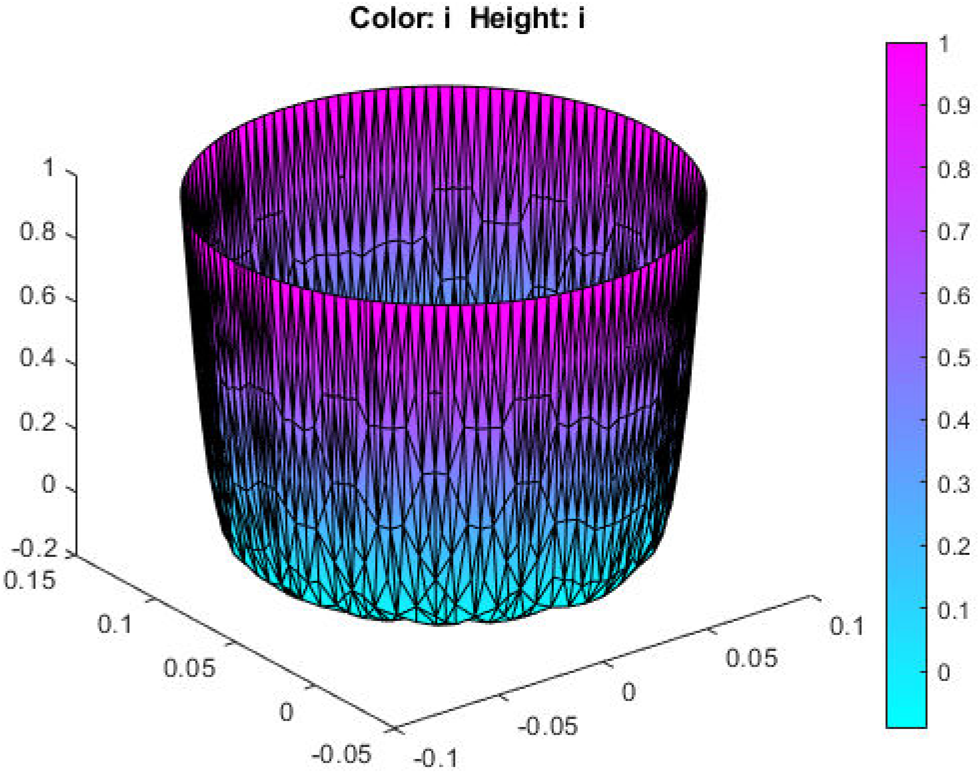
Warning: Imaginary part of complex property was ignored when plotting.

8. Plot the current density as a 3-D plot. To do this:
   - **a** Select **Plot** > **Parameters**.
   - **b** Select the **Color** and **Height(3-D plot)** options.
   - **c** Select `current density` from the **Property** drop-down menu for both the **Color** and **Height(3-D plot)** options.
   - **d** Select **Show Mesh** to observe the mesh.

   Due to the skin effect, the current density at the surface of the conductor is much higher than in the conductor's interior.

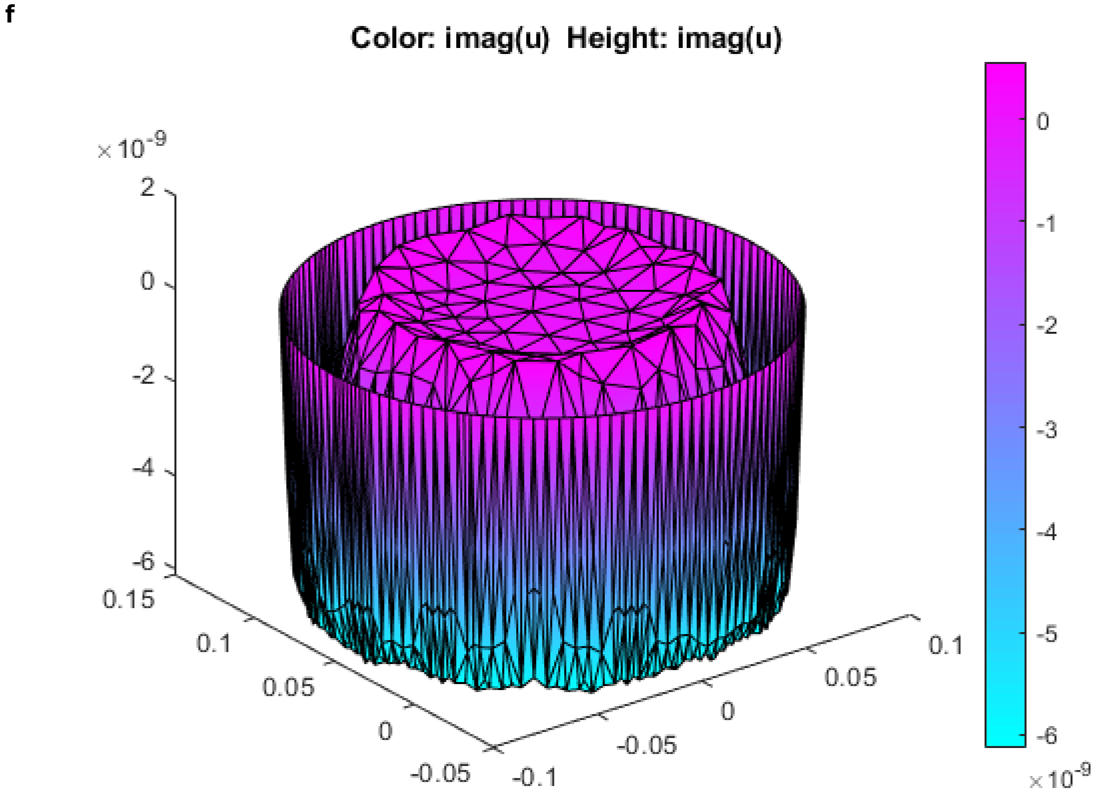


9. Improve the accuracy of the solution close to the surface by using adaptive mesh refinement. To do this:
   - **a** Select **Solve** > **Parameters**.
   - **b** In the resulting dialog box, select **Adaptive mode**.
   - **c** Set the maximum numbers of triangles to `Inf`.
   - **d** Set the maximum numbers of refinements to `1`.
   - **e** Select the `Worst triangles` selection method.
10. Recompute the solution five times. Each time, the adaptive solver refines the area with the largest errors. The number of triangles is printed at the command line.
11. Plot the current density as a 3-D plot.



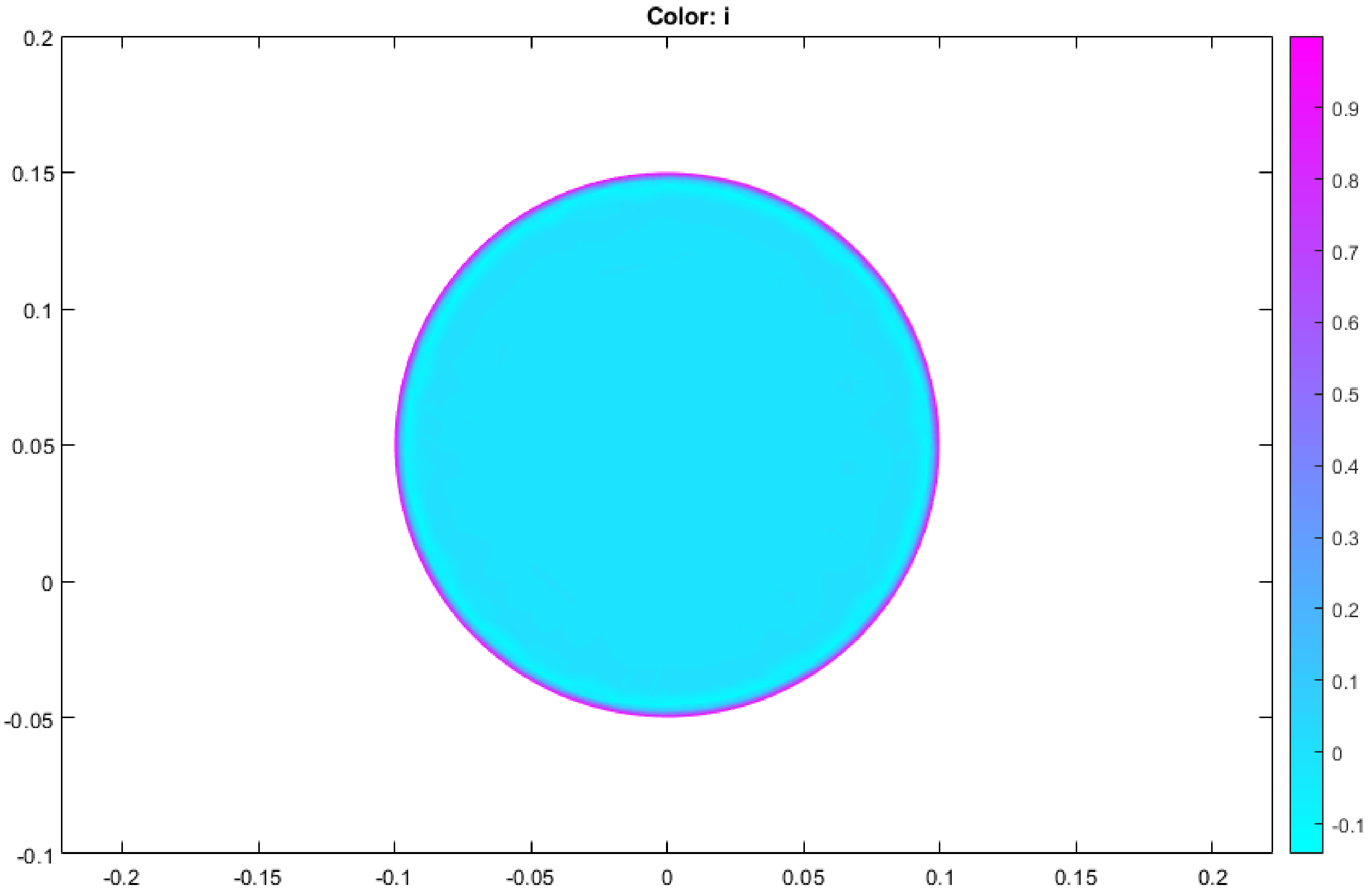
12 These plots show the real part of the solution, but the solution vector is the full complex solution. Plot the imaginary part of the solution. To do this:

- **a** Select **Plot** > **Parameters**.
- **b** Select the **Color** and **Height(3-D plot)** options.
- **c** Select `user entry` from the **Property** drop-down menu for both **Color** and **Height(3-D plot)** options.
- **d** Type `imag(u)` in the corresponding **User entry** fields.
- **e** Select **Show Mesh** to observe the mesh.

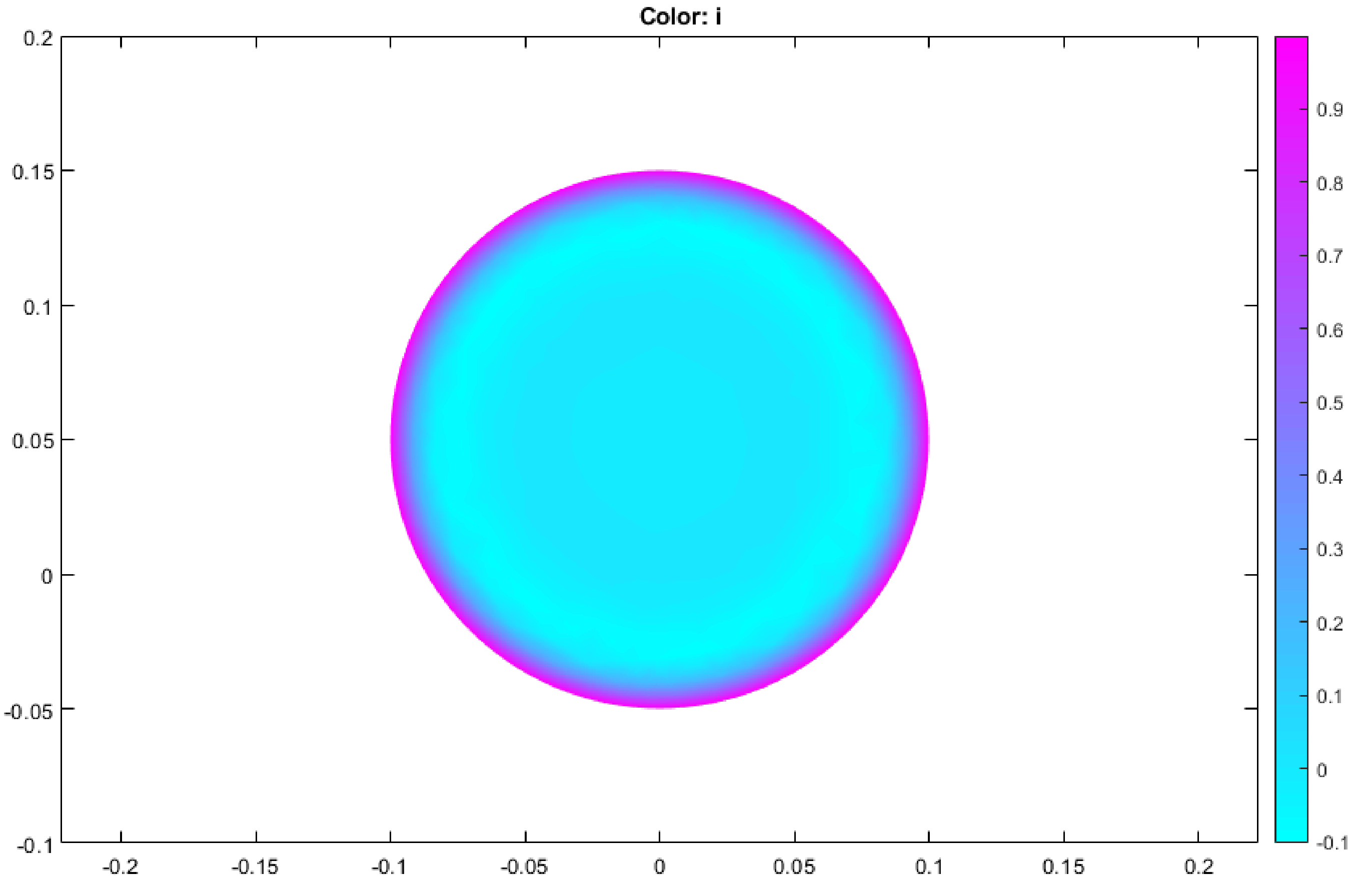


13 Observe that the skin effect depends on the frequency of the alternating current. When you increase or decrease the frequency, the skin "depth" increases or decreases, respectively. At high frequencies, only a thin layer on the surface of the wire conducts the current. At very low frequencies (approaching DC conditions), almost the entire cross section area of the wire conducts the current.

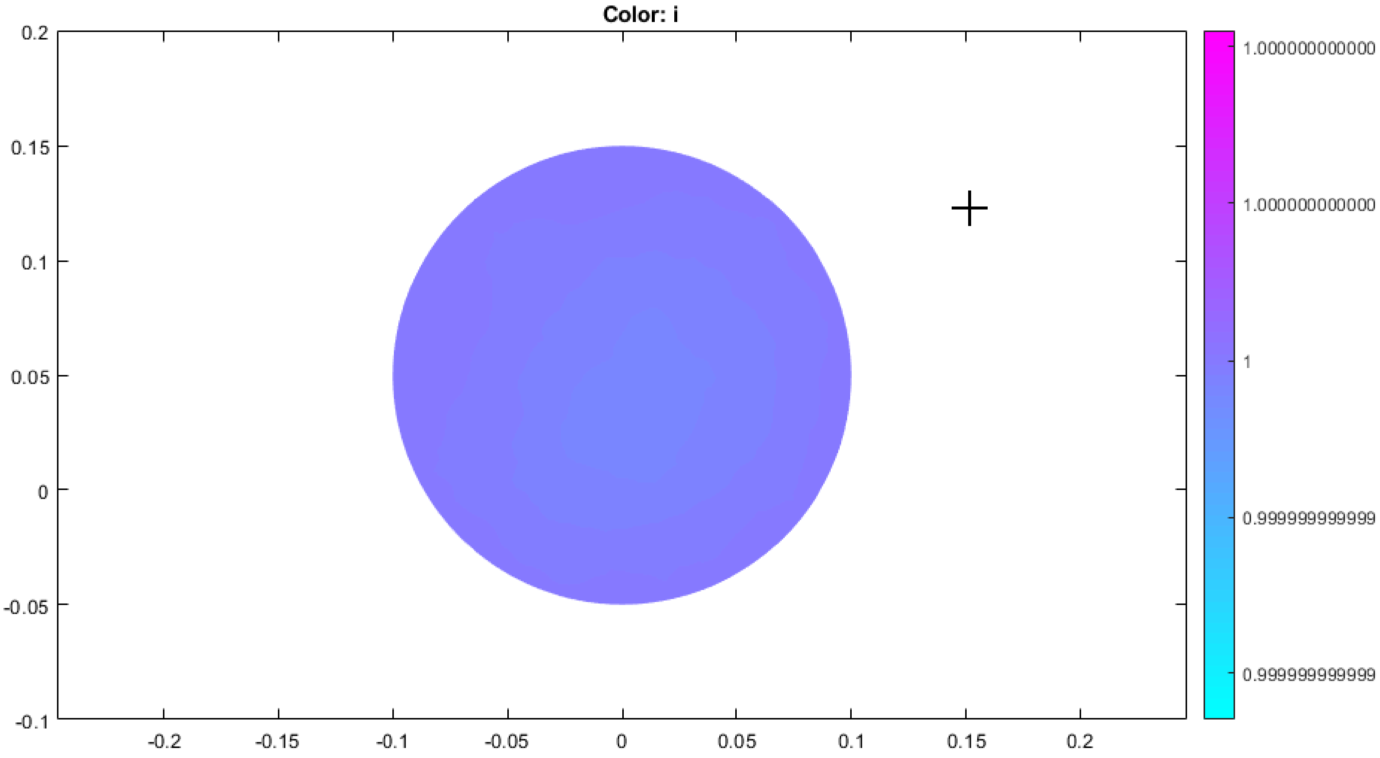
Find the solution for the angular frequencies `omega = 2*pi*1000`, `omega = 2*pi*50`, and `omega = 1E-6`. Plot the real parts of the solutions in 2-D.



Current density for omega = 2*pi*1000



Current density for omega = 2*pi*50



Current density for omega = 1E-6

Current Density Between Two Metallic Conductors: PDE Modeler App

Two circular metallic conductors are placed on a brine-soaked blotting paper which serves as a plane, thin conductor. The physical model for this problem consists of the Laplace equation

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

for the electric potential V and these 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

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

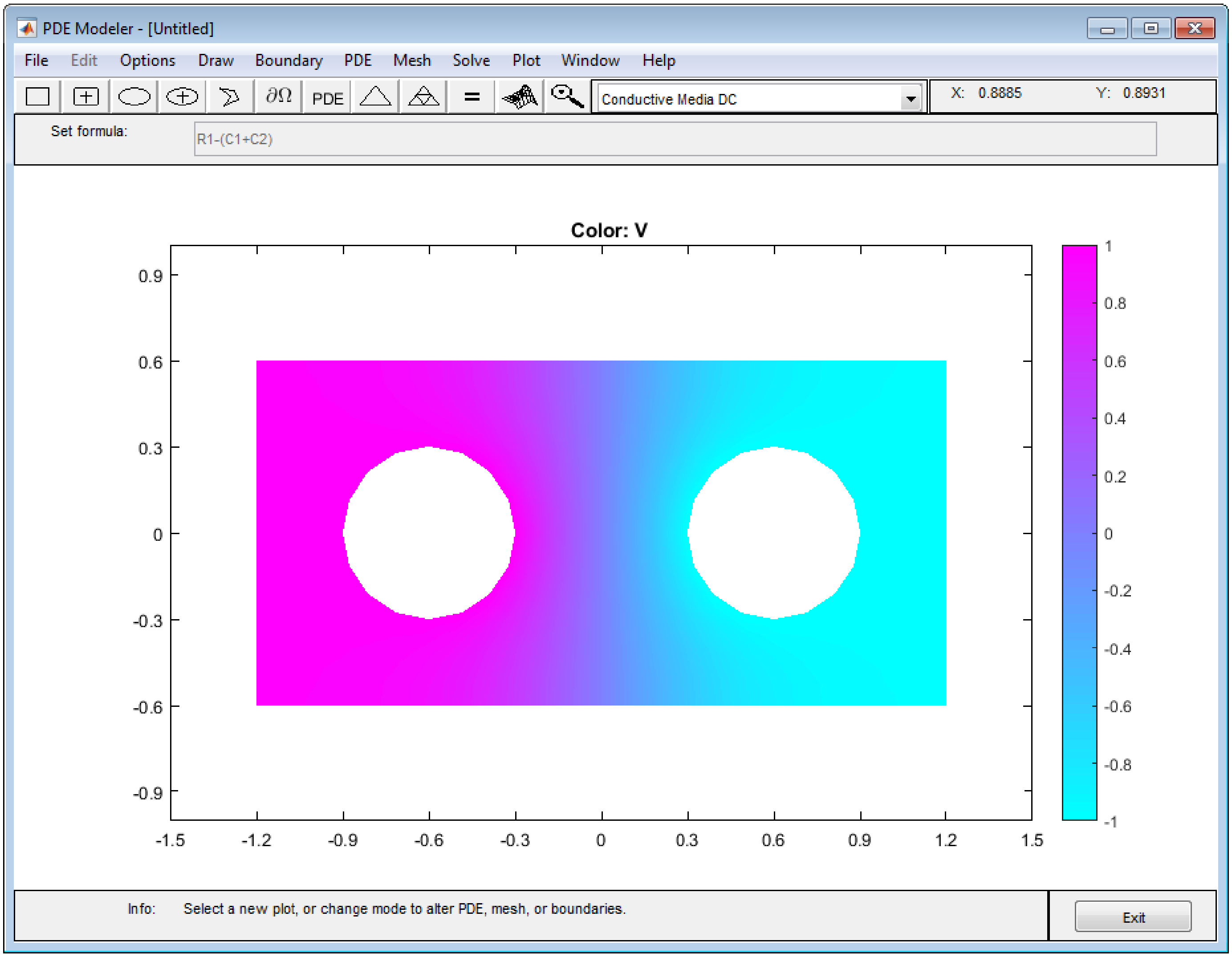
The conductivity is $\sigma = 1$.

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

1. Model the geometry: draw the rectangle with corners at (-1.2,-0.6), (1.2,-0.6), (1.2,0.6), and (-1.2,0.6), and two circles with a radius of 0.3 and centers at (-0.6,0) and (0.6,0). The rectangle represents the blotting paper, and the circles represent the conductors.

   ```
   pderect([-1.2 1.2 -0.6 0.6])
   pdecirc(-0.6,0,0.3)
   pdecirc(0.6,0,0.3)
   ```
2. Model the geometry by entering `R1-(C1+C2)` in the **Set formula** field.
3. Set the application mode to **Conductive Media DC**.
4. 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 rectangle, use the Neumann boundary condition with `g = 0` and `q = 0`.
   - For the left circle, use the Dirichlet boundary condition with `h = 1` and `r = 1`.
   - For the right circle, use the Dirichlet boundary condition with `h = 1` and `r = -1`.
5. Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Specify `sigma = 1` and `q = 0`.
6. Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.
7. Refine the mesh by selecting **Mesh** > **Refine Mesh**.
8. Improve the triangle quality by selecting **Mesh** > **Jiggle Mesh**.
9. Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar. The resulting potential is zero along the *y*-axis, which, for this problem, is a vertical line of antisymmetry.



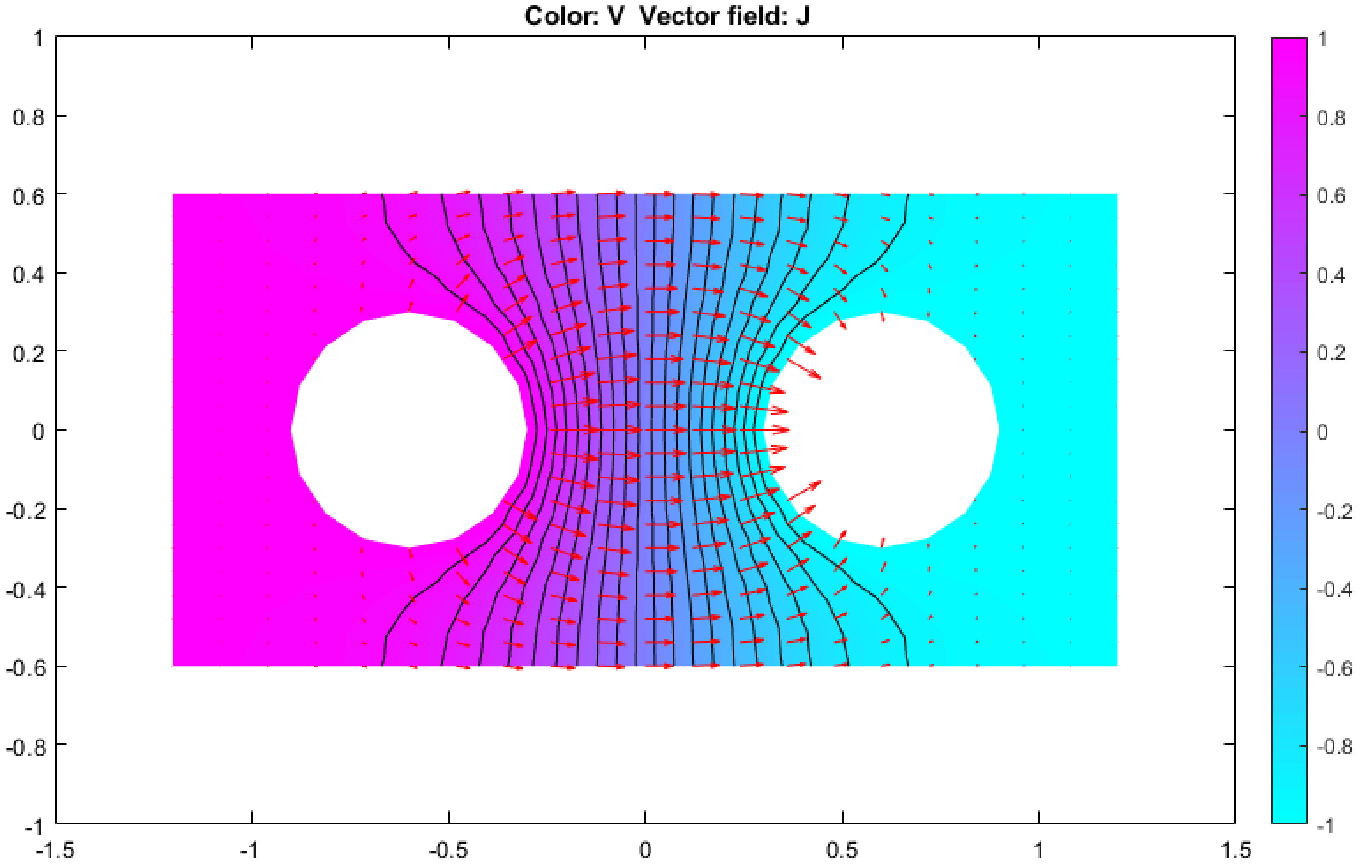
Set formula: R1-(C1+C2)

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

10 Plot the current density **J**. To do this:

- **a** Select **Plot > Parameters**.
- **b** In the resulting dialog box, select the **Color**, **Contour**, and **Arrows** options.
- **c** Set the **Arrows** value to `current density`.

The current flows, as expected, from the conductor with a positive potential to the conductor with a negative potential. The conductivity σ is isotropic, and the equipotential lines are orthogonal to the current lines.


Color: V Vector field: J
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.5
-1
-0.5
0
0.5
1
1.5
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1

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

Solve the following heat transfer problem with different material parameters. This example uses the PDE Modeler app. For the command-line solutions see “Heat Transfer Between Two Squares Made of Different Materials” on page 5-285.

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:

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

where ρ 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_{ext} is the external temperature.

To solve this problem in the PDE Modeler 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`
   - Heat source, `Q = 0`
   - Convective heat transfer coefficient, `h = 0`
   - External temperature, `Text = 0`

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

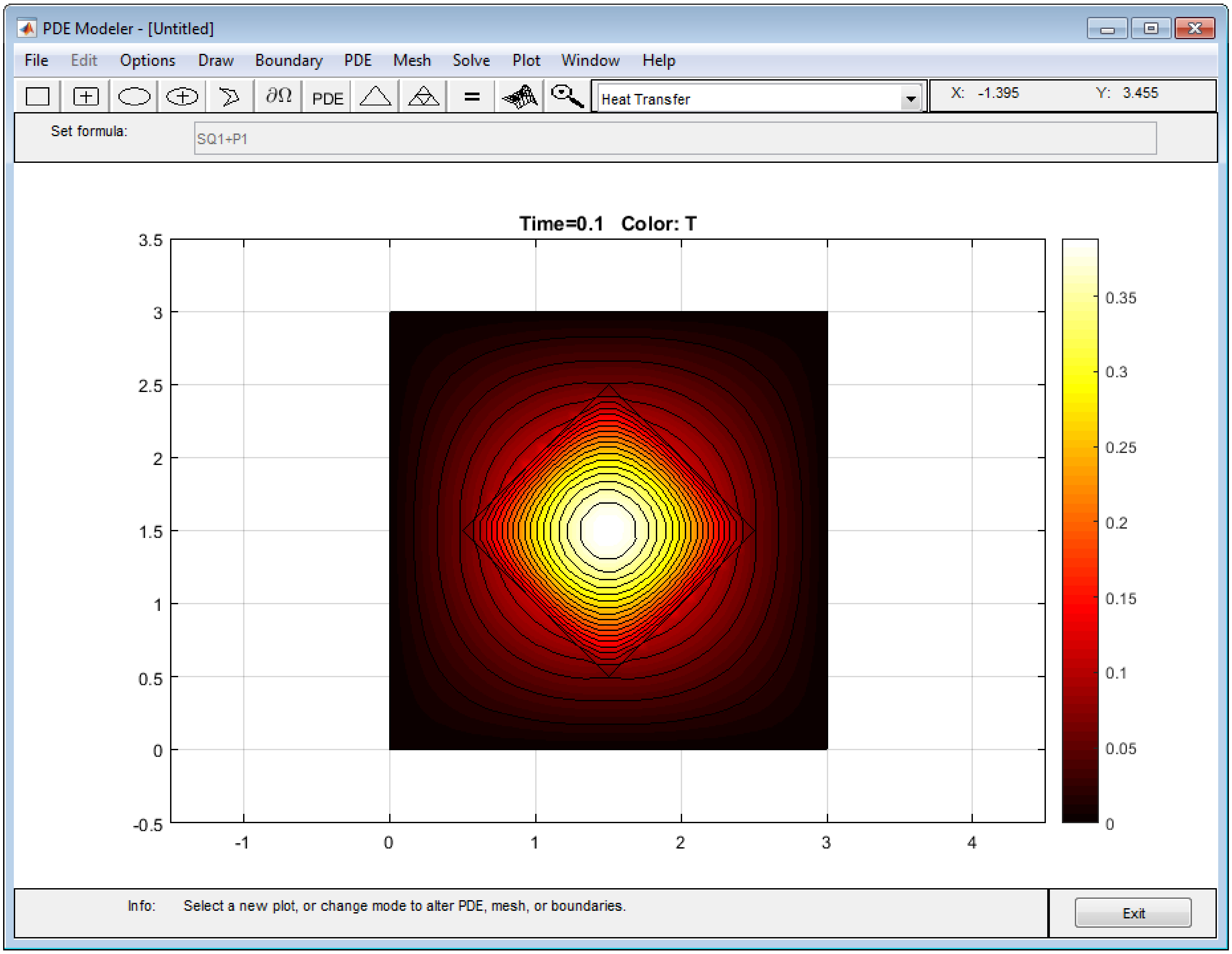
   - Density, `pho = 1`
   - Heat capacity, `C = 0.1`
   - Coefficient of heat conduction, `k = 2`
   - Heat source, `Q = 4`
   - Convective heat transfer coefficient, `h = 0`
   - External temperature, `Text = 0`

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.

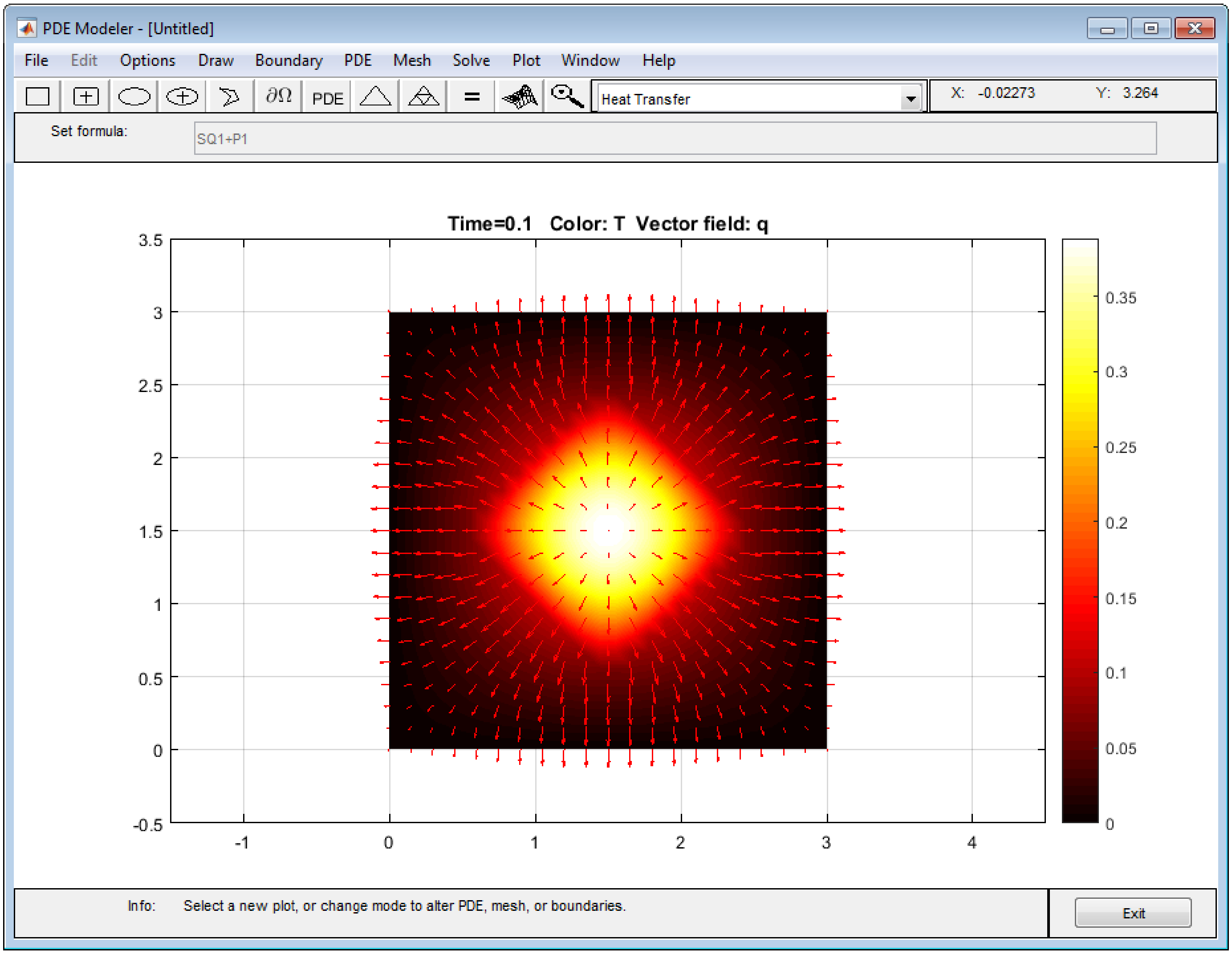
   a. Select **Plot** > **Parameters**.
   b. In the resulting dialog box, deselect the **Animation**, and **Height (3-D plot)**, and **Plot in x-y grid** options.
   c. Change the colormap to `hot` by using the corresponding drop-down menu in the same dialog box.
   d. To obtain the first plot, select 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.



Set formula: SQ1+P1

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

Isothermal Lines



Heat Transfer

X: -0.02273 Y: 3.264

Set formula: SQ1+P1

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

Temperature and Heat Flux

Temperature Distribution in Heat Sink

This example shows how to create a simple 3-D heat sink geometry and analyze heat transfer on the heat sink. The process has three steps.

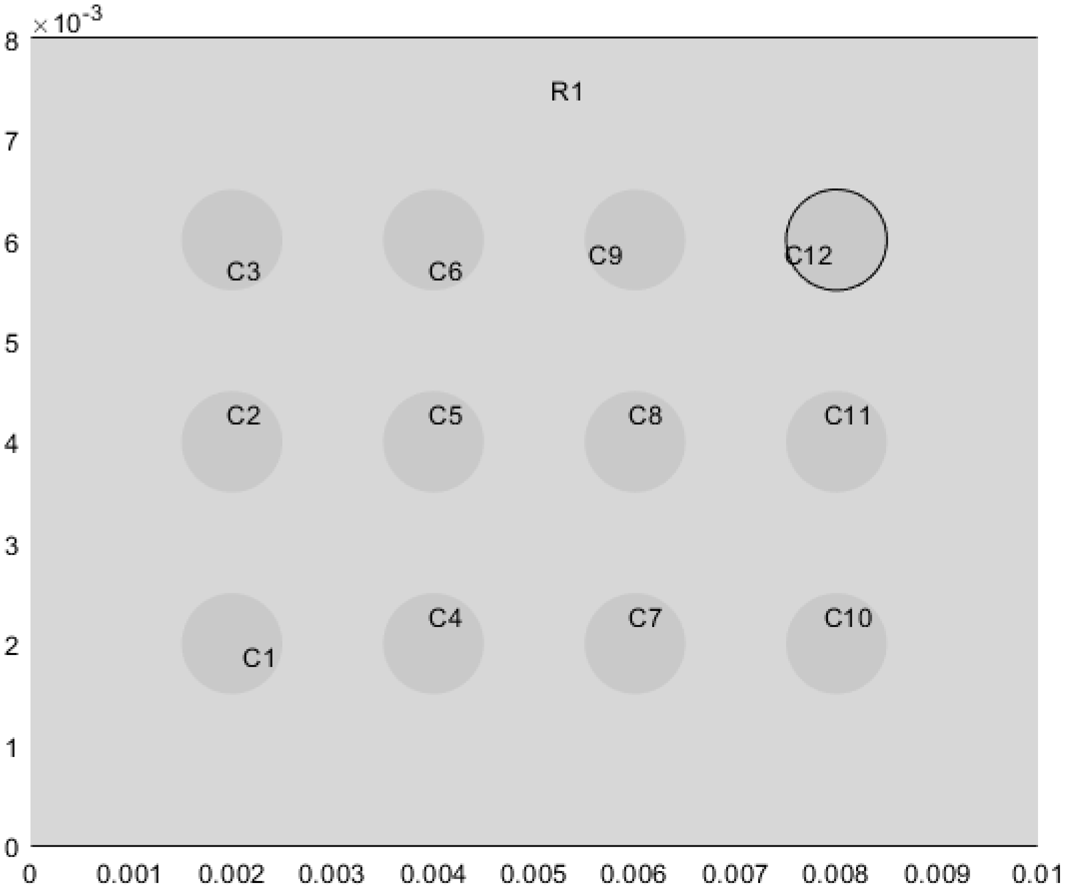
1. “Create 2-D Geometry in PDE Modeler App” on page 3-199.
2. “Extrude 2-D Geometry into 3-D Geometry of Heat Sink” on page 3-200.
3. “Perform Thermal Analysis” on page 3-203.

Create 2-D Geometry in PDE Modeler App

Create a geometry in the PDE Modeler app. First, open the PDE Modeler app with a geometry consisting of a rectangle and 12 circles.

```
pderect([0 0.01 0 0.008])
for i = 0.002:0.002:0.008
   for j = 0.002:0.002:0.006
       pdecirc(i,j,0.0005)
   end
end
```

Adjust the axes limits by selecting **Options** > **Axes Limits**. Select **Auto** to use automatic scaling for both axes.

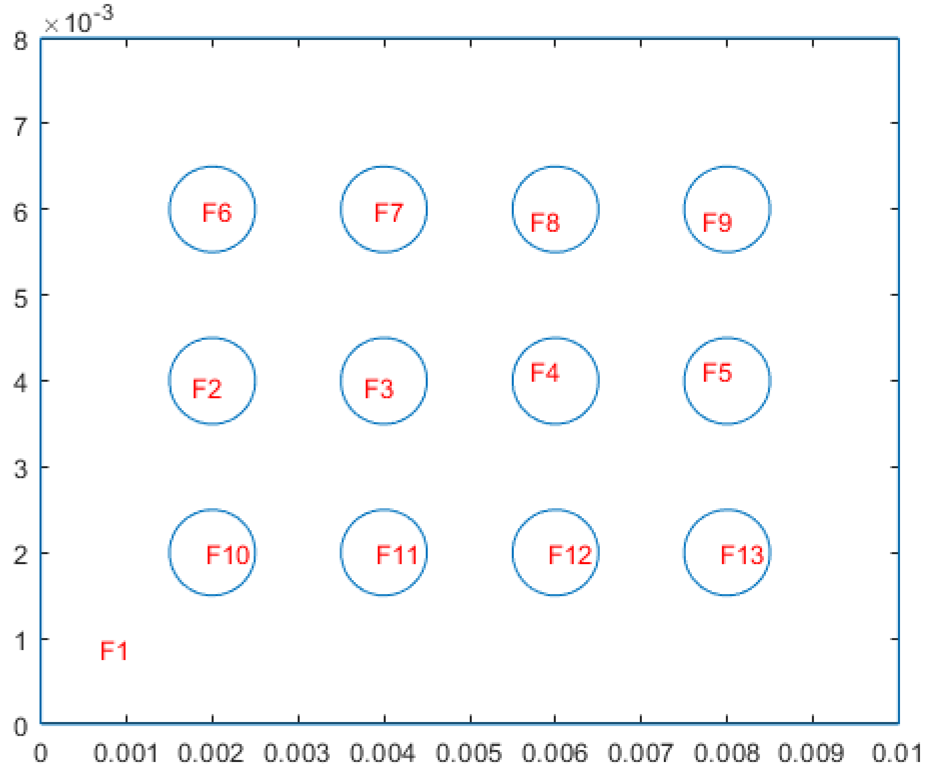


Export the geometry description matrix, set formula, and name-space matrix into the MATLAB workspace by selecting **Draw** > **Export Geometry Description, Set Formula, Labels**. This data lets you reconstruct the geometry in the workspace.

Extrude 2-D Geometry into 3-D Geometry of Heat Sink

In the MATLAB Command Window, use the `decsg` function to decompose the exported geometry into minimal regions. Plot the result.

```
g = decsg(gd,sf,ns);
pdegplot(g,"FaceLabels","on")
```



Create a thermal model for transient analysis.

```
model = createpde("thermal","transient");
```

Create a 2-D geometry from the decomposed geometry matrix and assign the geometry to the thermal model.

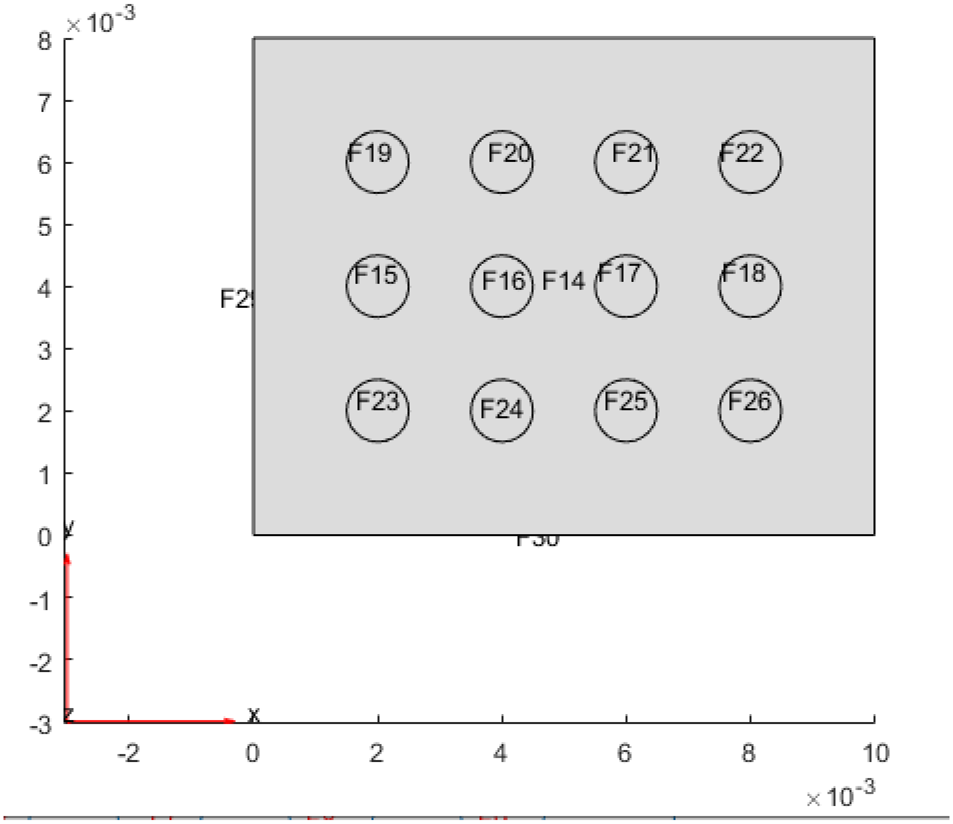
```
g = geometryFromEdges(model,g);
```

Extrude the 2-D geometry along the *z*-axis by 0.0005 units.

```
g = extrude(g,0.0005);
```

Plot the extruded geometry so that you can see the face labels on the top.

```
figure
pdegplot(g,"FaceLabels","on")
view([0 90])
```

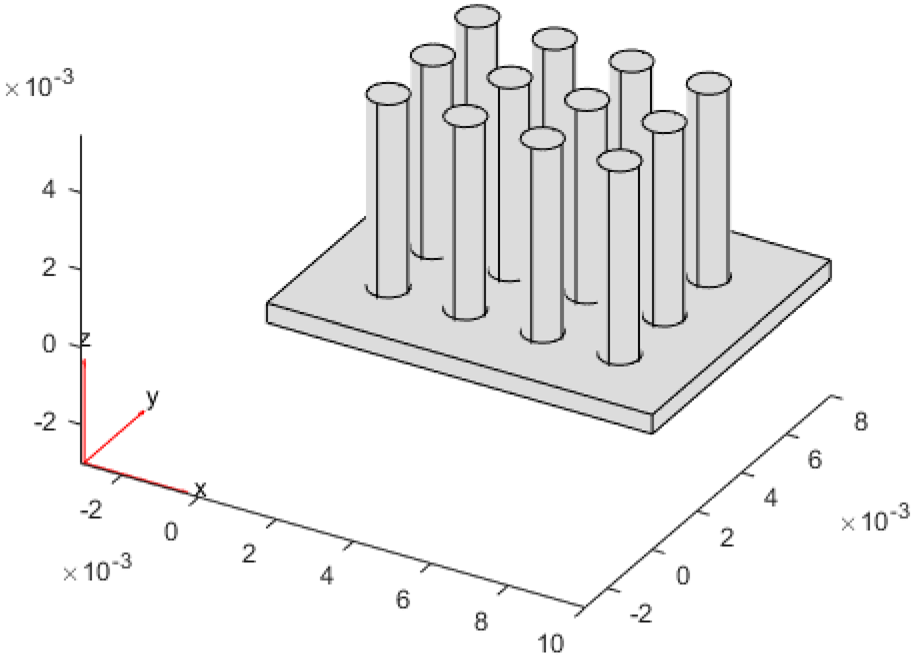


Extrude the circular faces (faces with IDs from 15 to 26) along the *z*-axis by 0.005 more units. These faces form the fins of the heat sink.

```
g = extrude(g,[15:26],0.005);
```

Assign the modified geometry to the thermal model and plot the geometry.

```
model.Geometry = g;
figure
pdegplot(g)
```



Perform Thermal Analysis

Assuming that the heat sink is made of copper, specify the thermal conductivity, mass density, and specific heat.

```
thermalProperties(model,"ThermalConductivity",400, ...
                        "MassDensity",8960, ...
                        "SpecificHeat",386);
```

Specify the Stefan-Boltzmann constant.

```
model.StefanBoltzmannConstant = 5.670367e-8;
```

Apply the temperature boundary condition on the bottom surface of the heat sink, which consists of 13 faces.

```
thermalBC(model,"Face",1:13,"Temperature",1000);
```

Specify the convection and radiation parameters on all other surfaces of the heat sink.

```
thermalBC(model,"Face",14:g.NumFaces, ...
                "ConvectionCoefficient",5, ...
                "AmbientTemperature",300, ...
                "Emissivity",0.8);
```

Set the initial temperature of all the surfaces to the ambient temperature.

```
thermalIC(model,300);
```

Generate a mesh.

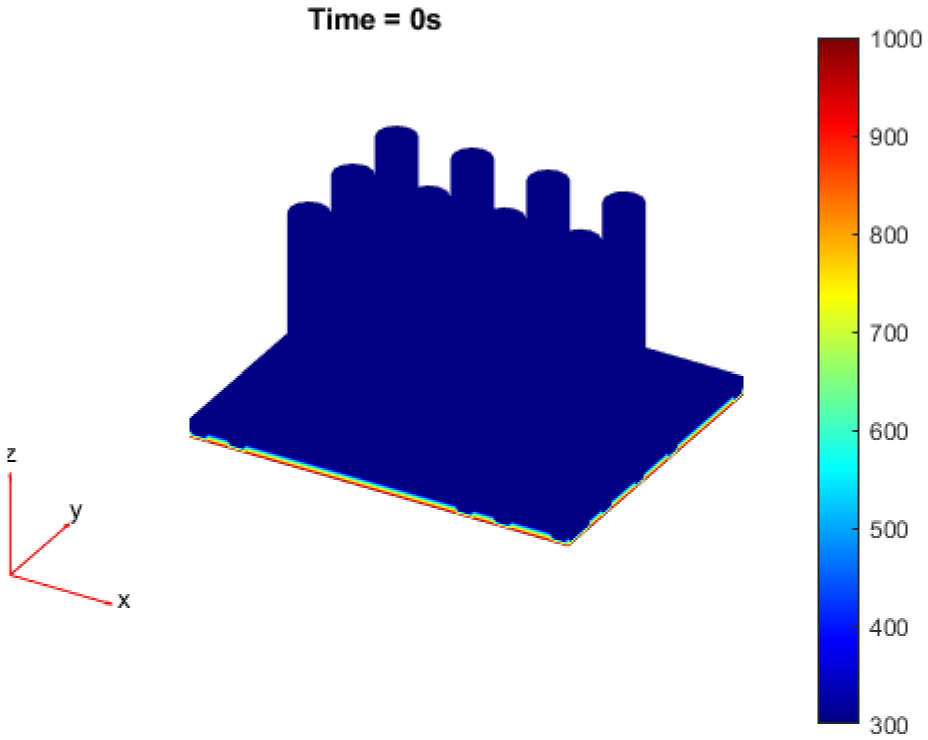
```
generateMesh(model);
```

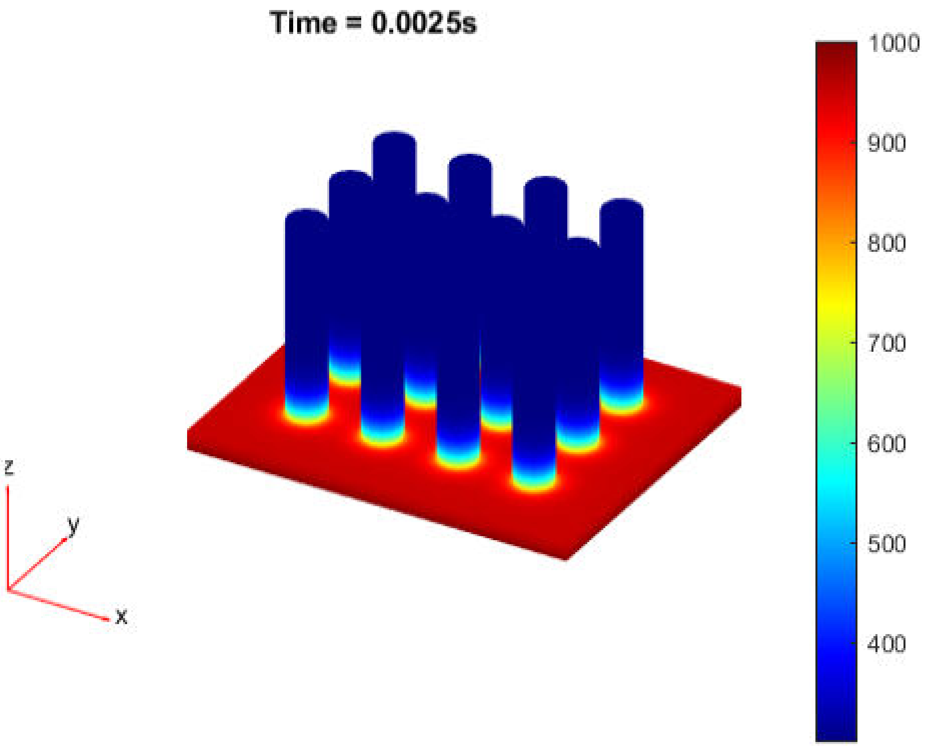
Solve the transient thermal problem for times between 0 and 0.0075 s with a time step of 0.0025 s.

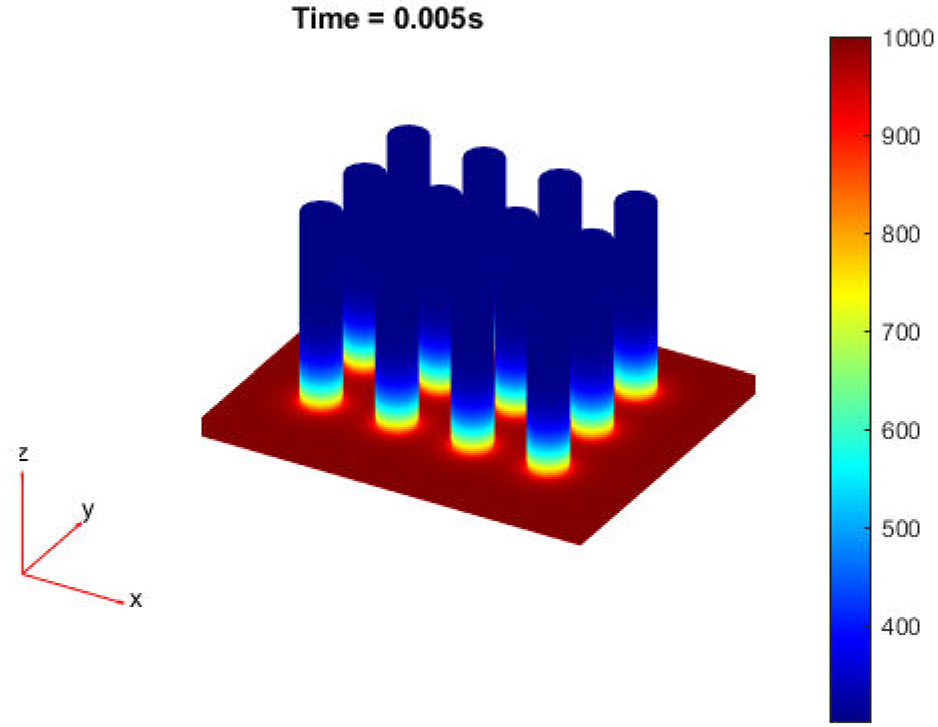
```
results = solve(model,0:0.0025:0.0075);
```

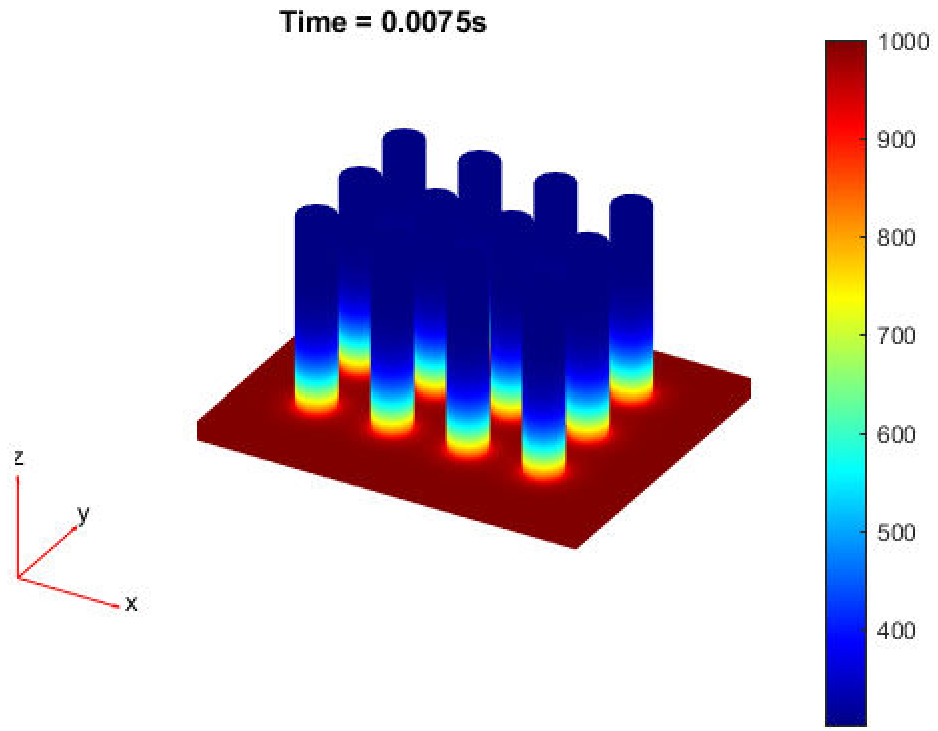
Plot the temperature distribution for each time step.

```
for i = 1:length(results.SolutionTimes)
  figure
  pdeplot3D(model,"ColorMapData",results.Temperature(:,i))
  title({['Time = ' num2str(results.SolutionTimes(i)) 's']})
end
```



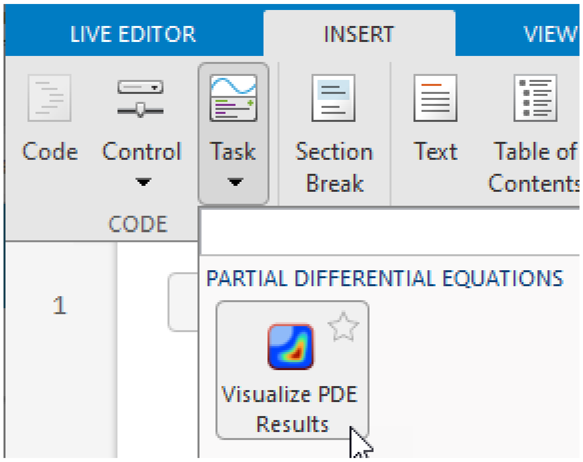

Time = 0.0025s
1000
900
800
700
600
500
400
z
y
x


Time = 0.005s
1000
900
800
700
600
500
400
z
y
x



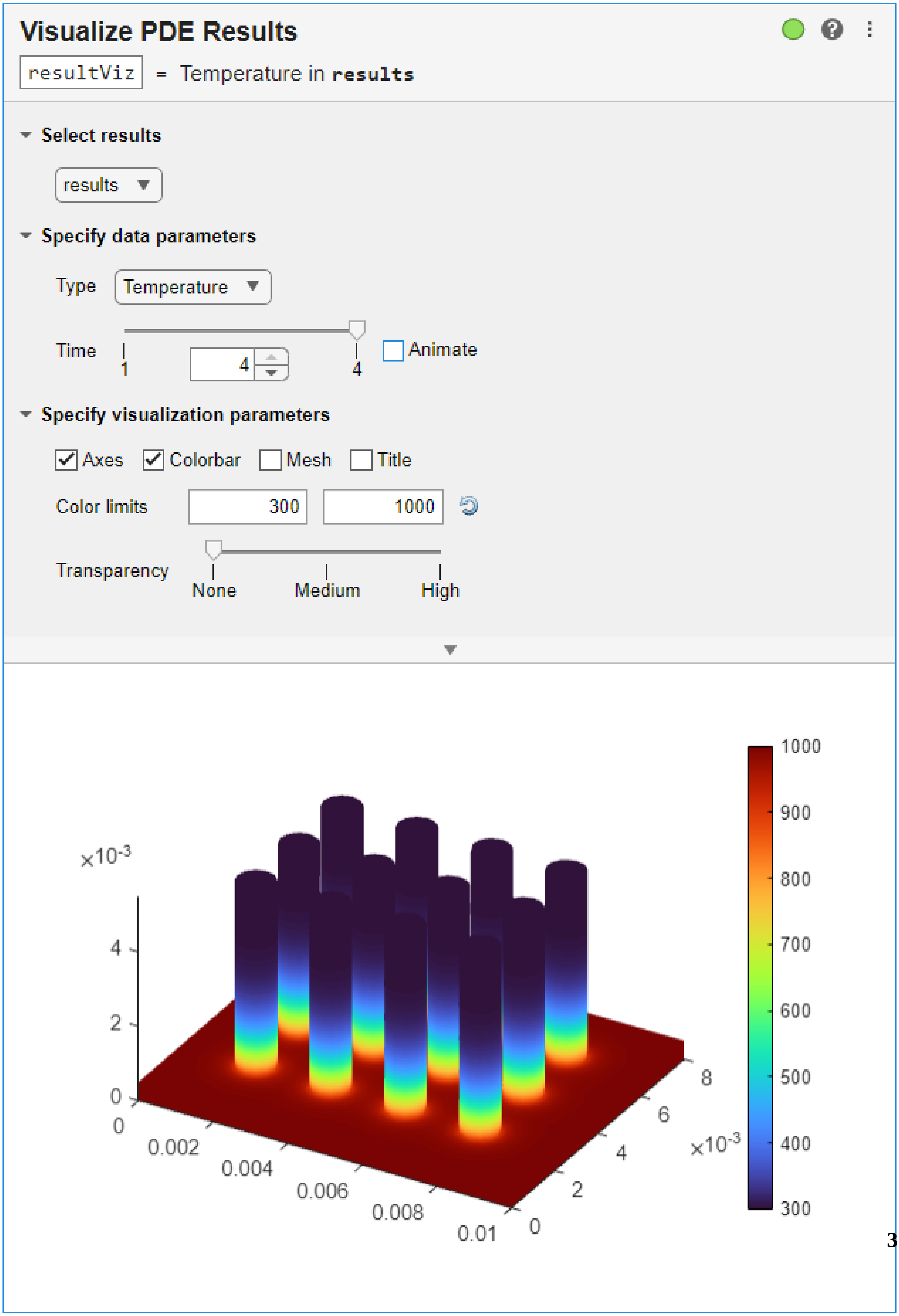
You also can plot the same results by using the **Visualize PDE Results** Live Editor task. First, create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

On the **Live Editor** tab, select **Task** > **Visualize PDE Results**. This action inserts the task into your script.



To plot the temperature distribution for each time step, follow these steps.

1. In the **Select results** section of the task, select `results` from the drop-down list.
2. In the **Specify data parameters** section of the task, set **Type** to *Temperature*, and set the time steps to 1, 2, 3, and 4. These time steps correspond to the four time values that you used to solve the problem. You also can animate the solution by selecting **Animate**.



Visualize PDE Results

`resultViz` = Temperature in **`results`**

▾ **Select results**

results ▼

▾ **Specify data parameters**

Type: Temperature ▼

Time: 4 (1 to 4) ☐ Animate

▾ **Specify visualization parameters**

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title

Color limits: 300 1000

Transparency: None | Medium | High

Nonlinear Heat Transfer in Thin Plate

This example shows how to perform a heat transfer analysis of a thin plate.

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

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

Heat Transfer Equations for the Plate

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

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

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

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

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

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

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

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

The PDE describing the temperature in this thin plate is

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

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

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

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

Problem Setup

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);
```

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.

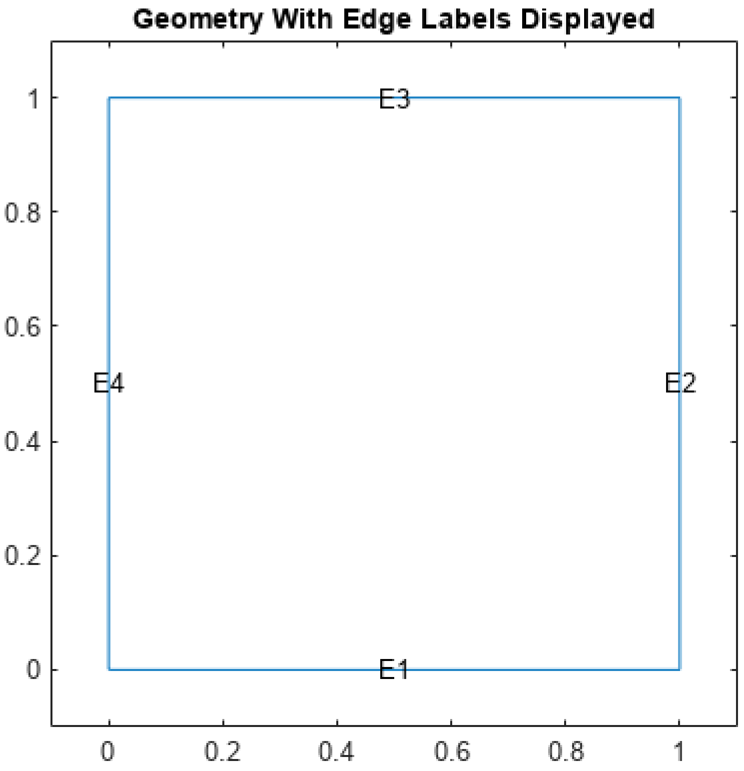
```
gdm = [3 4 0 width width 0 0 0 height height]';
g = decsg(gdm, 'S1', ('S1')');
```

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")
```



Specify the 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);
```

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

Apply the boundary conditions. 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,

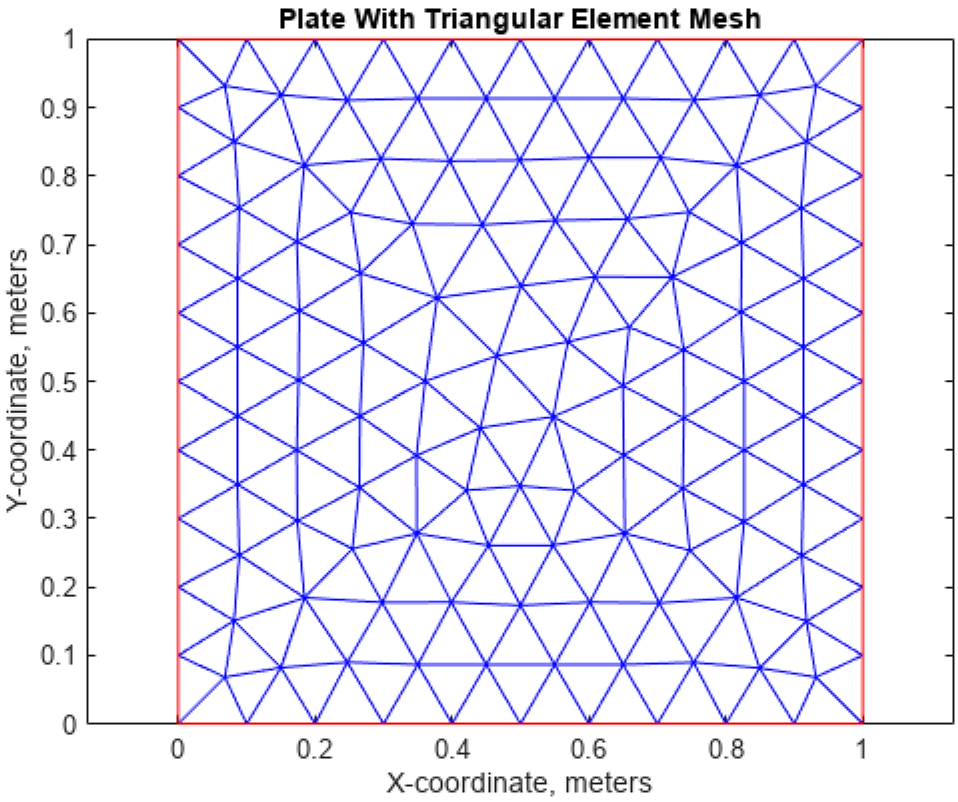
```
applyBoundaryCondition(model,"dirichlet","Edge",1,"u",1000);
```

Specify the initial guess.

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

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

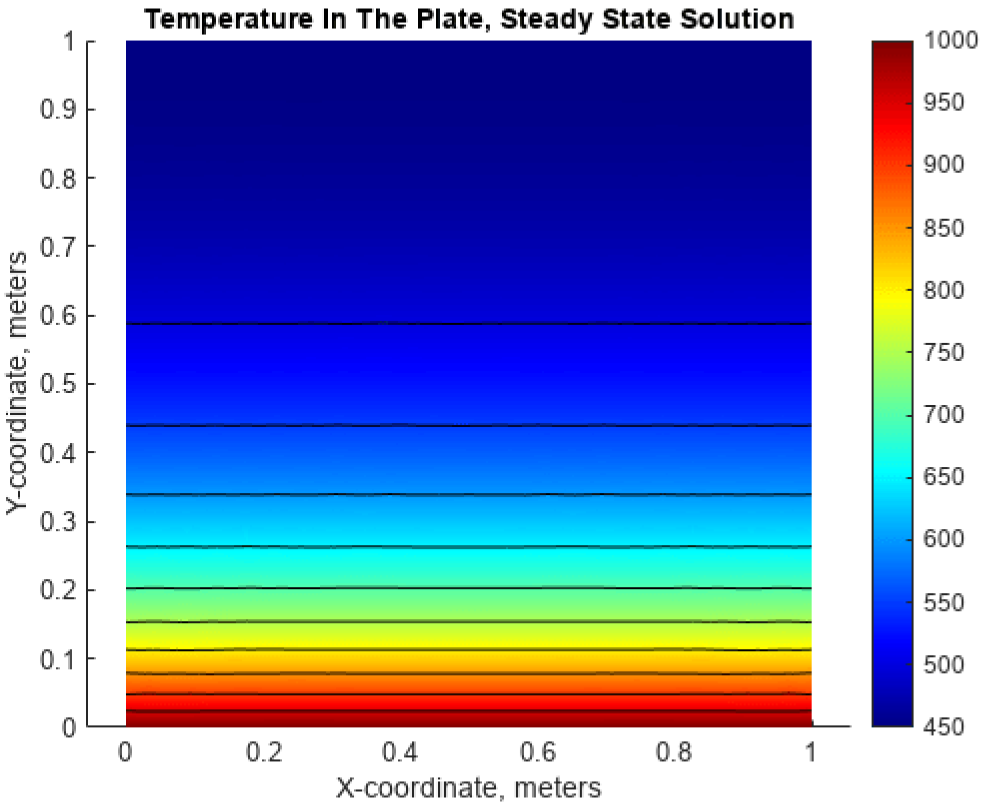
```
hmax = .1; % element size
msh = generateMesh(model,"Hmax",hmax);
figure;
pdeplot(model);
axis equal
title("Plate With Triangular Element Mesh")
xlabel("X-coordinate, meters")
ylabel("Y-coordinate, meters")
```



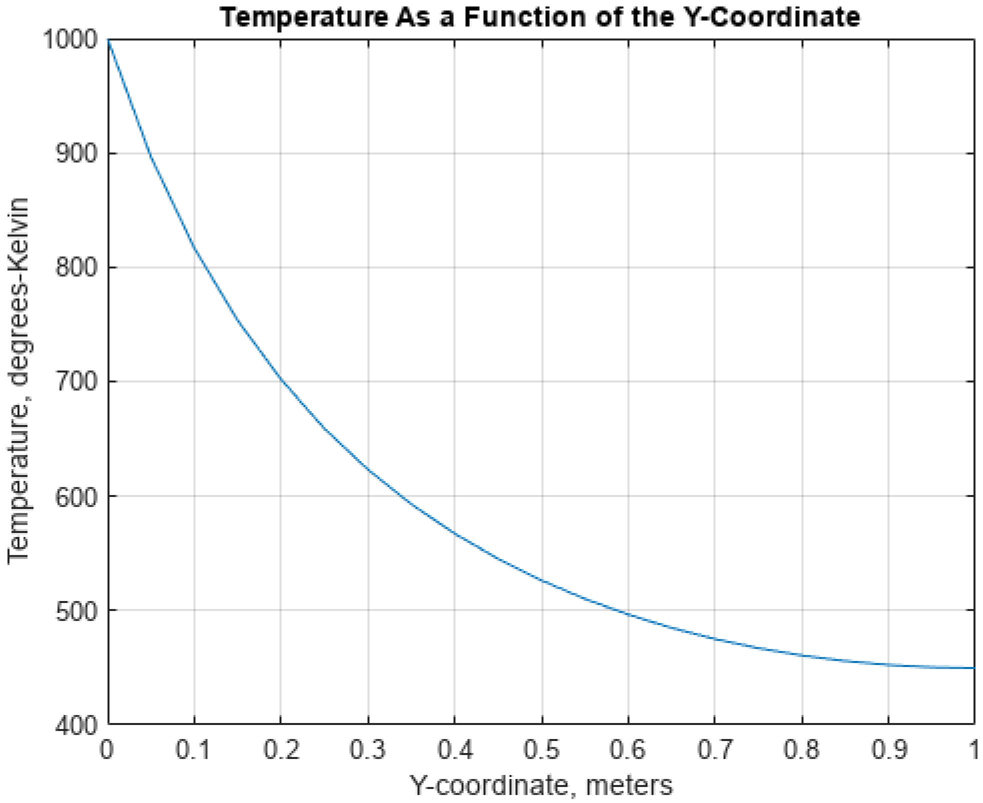
Steady State Solution

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 = 449.8 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.

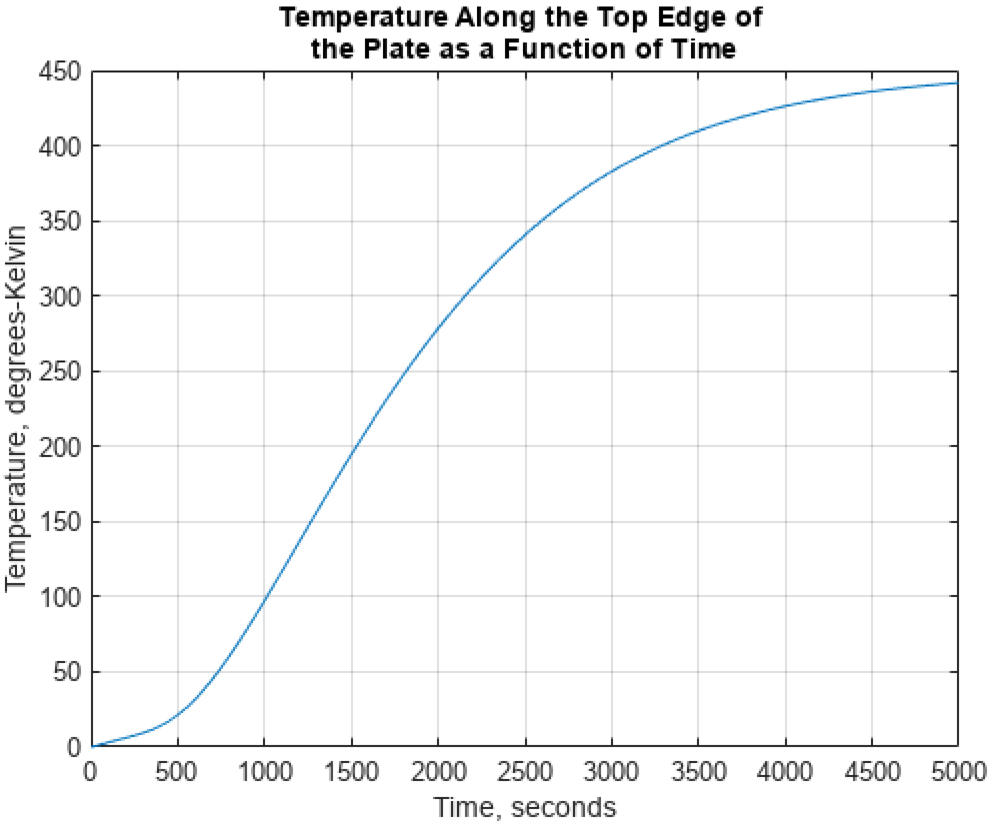
```
setInitialConditions(model,1000,"Edge",1);
```

Set the following solver options.

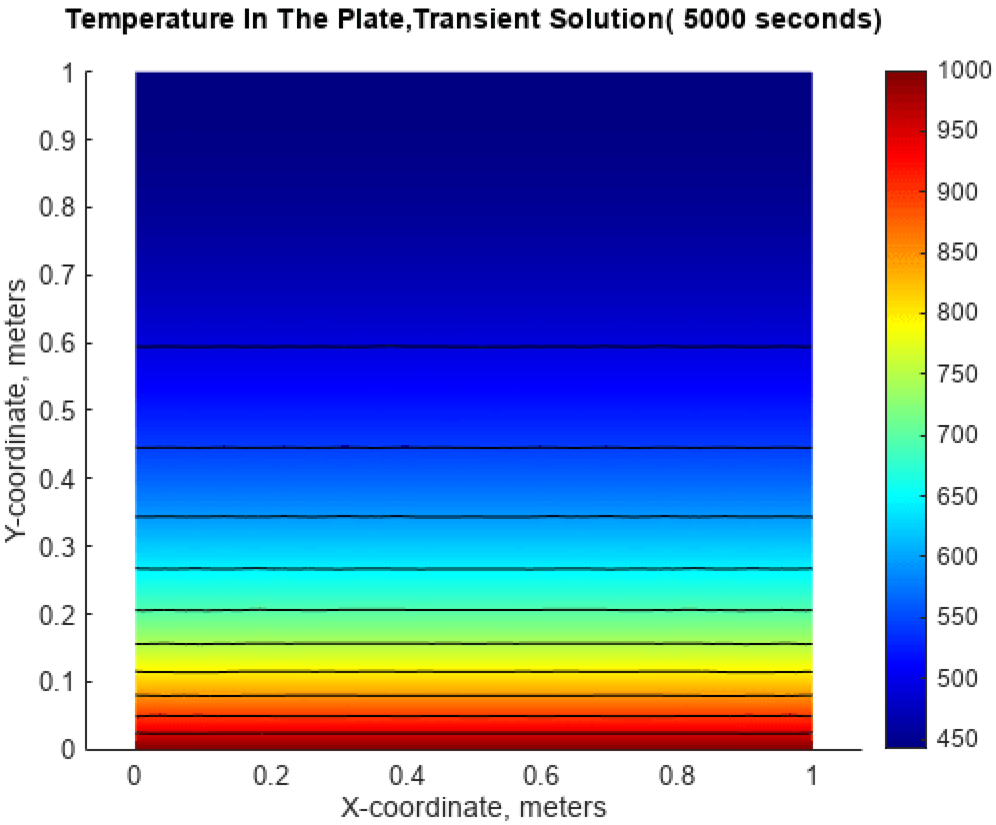
```
model.SolverOptions.RelativeTolerance = 1.0e-3;
model.SolverOptions.AbsoluteTolerance = 1.0e-4;
```

Solve the problem by using `solvepde`. The solver 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) = 441.8 degrees-K
```

Summary

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.

Poisson's Equation on Unit Disk: PDE Modeler App

This example shows how to solve the Poisson's equation on a unit disk and evaluate the numeric solution error.

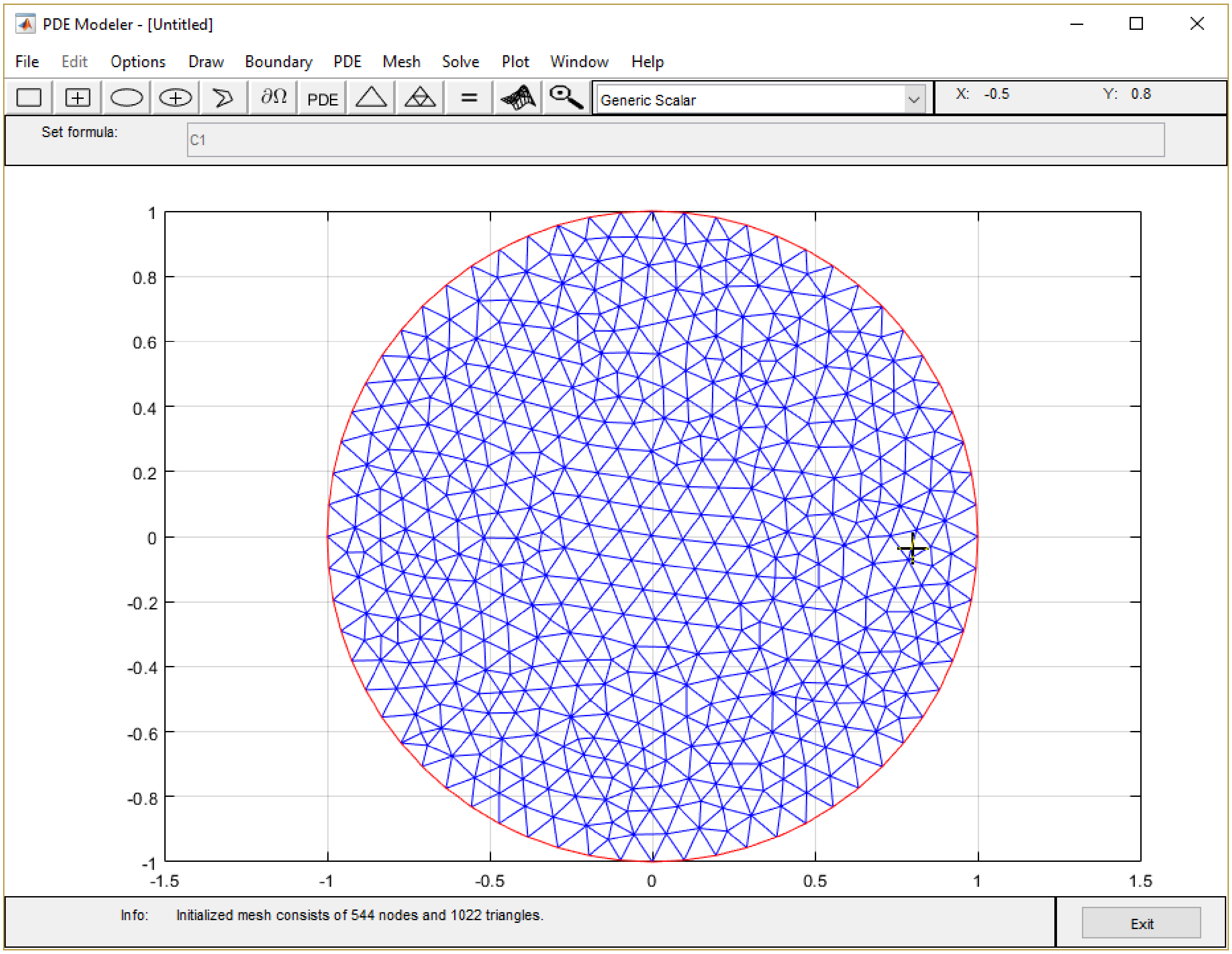
This example uses the PDE Modeler app. For a programmatic workflow, see “Poisson's Equation on Unit Disk” on page 3-224. Because the app and the programmatic workflow use different meshers, they yield slightly different results.

The problem formulation is $-\Delta u = 1$ in Ω, $u = 0$ on $\partial\Omega$, where Ω 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 Modeler app, follow these steps:

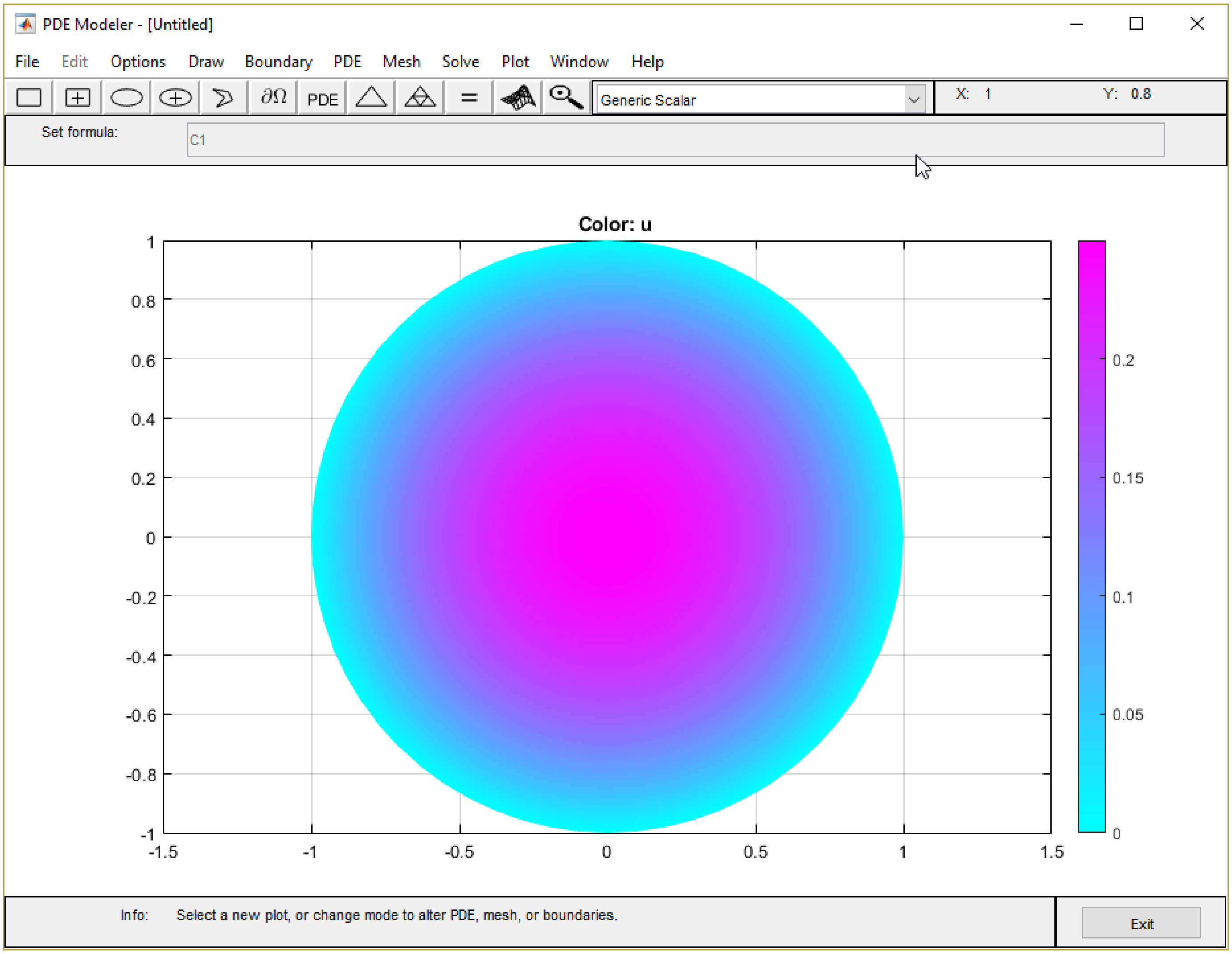
1. Open the PDE Modeler app by using the `pdeModeler` command.
2. Display grid lines by selecting **Options** > **Grid**.
3. Align new shapes to the grid lines by selecting **Options** > **Snap**.
4. Draw a circle with the radius 1 and the center at (0,0). To do this, first click the ⊕ button. Then right-click the origin and drag to draw a circle. Right-clicking constrains the shape you draw so that it is 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 ∂Ω 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 (`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.



Set formula: C1

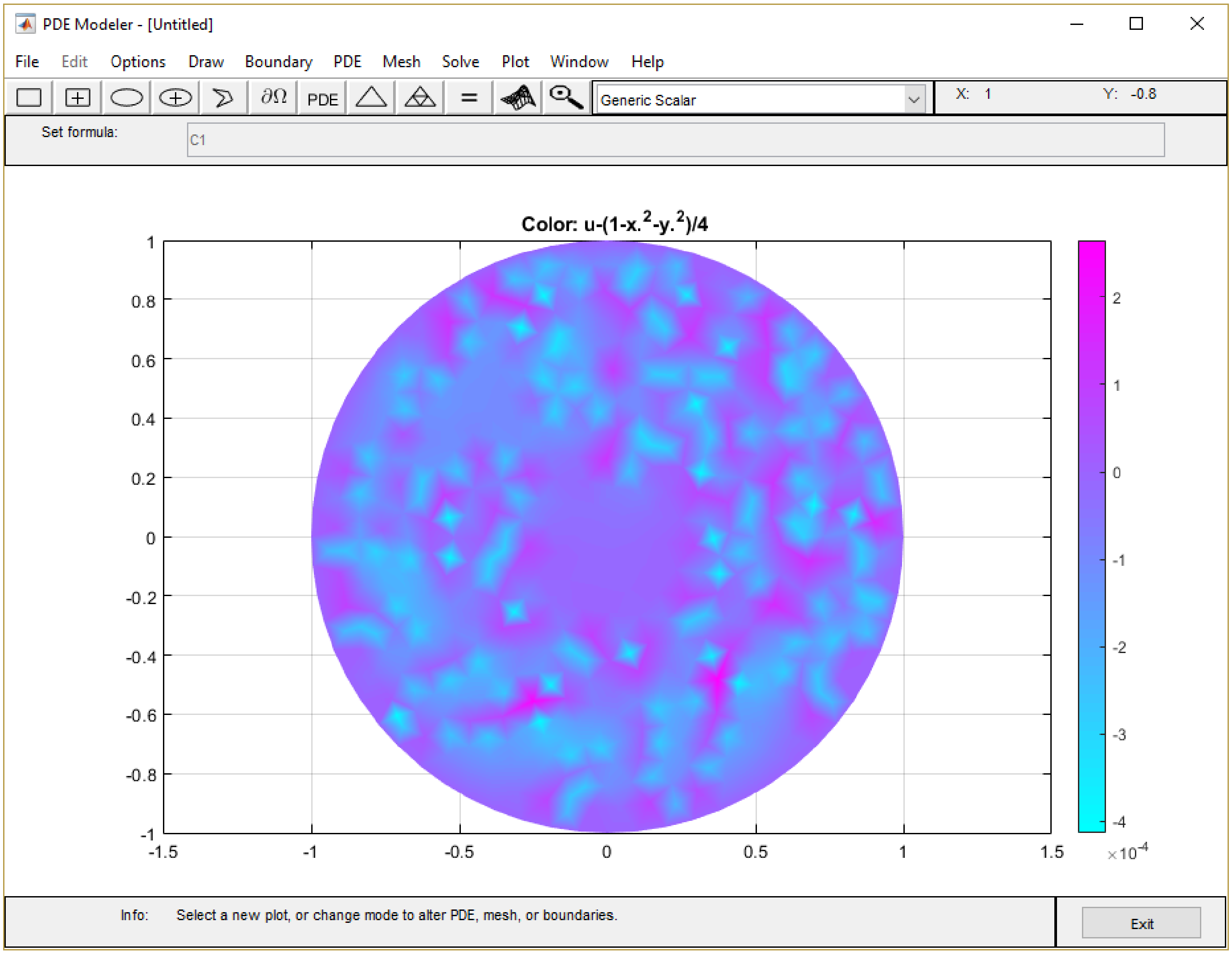
Info: Initialized mesh consists of 544 nodes and 1022 triangles.

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:

- **a** Select **Plot** > **Parameters**.
- **b** In the resulting dialog box, select `user entry` from the **Color** drop-down menu.
- **c** Plot the absolute error in the solution by typing the MATLAB expression `u-(1-x.^2-y.^2)/4` in the **User entry** field.

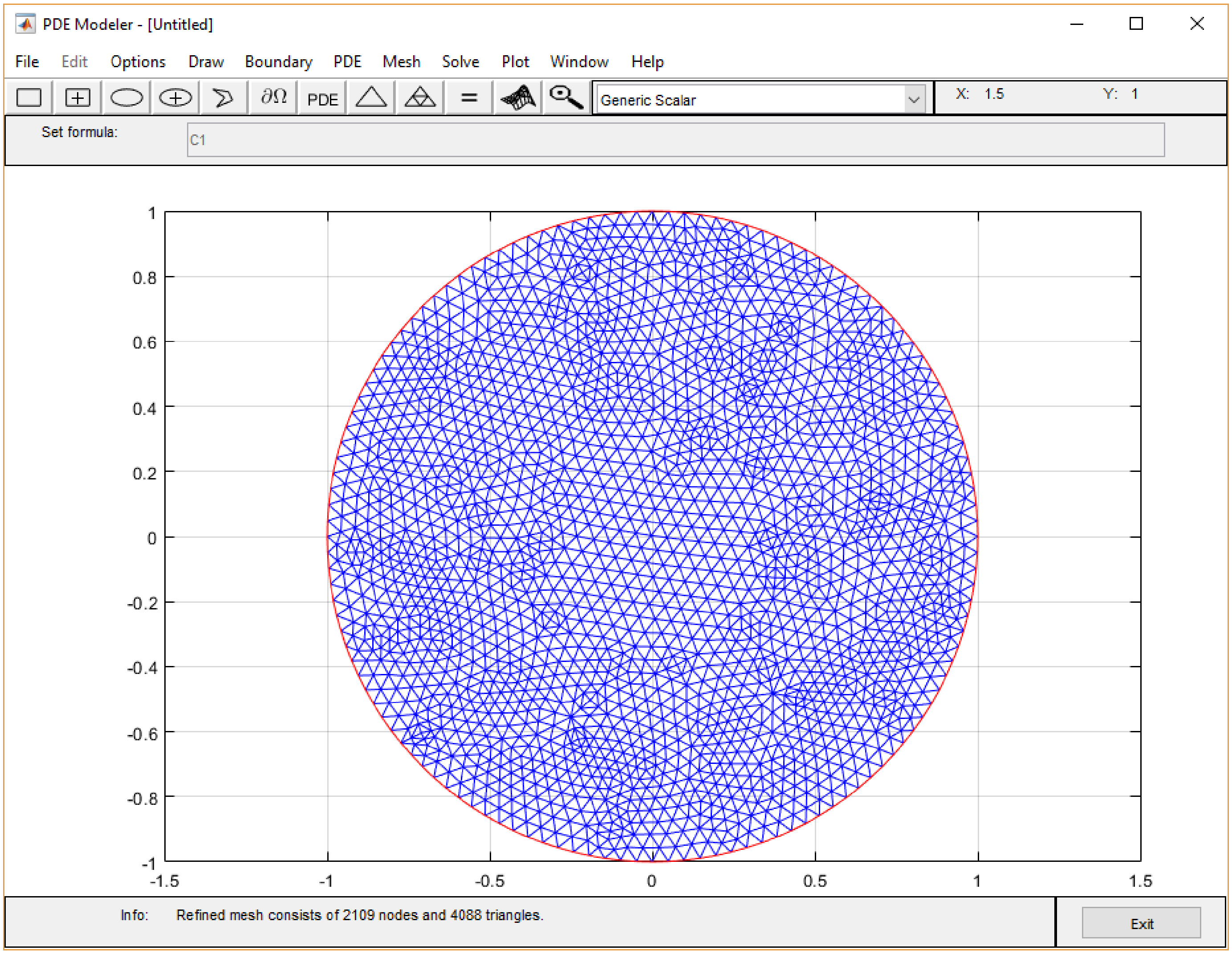


X: 1 Y: -0.8

Set formula: C1

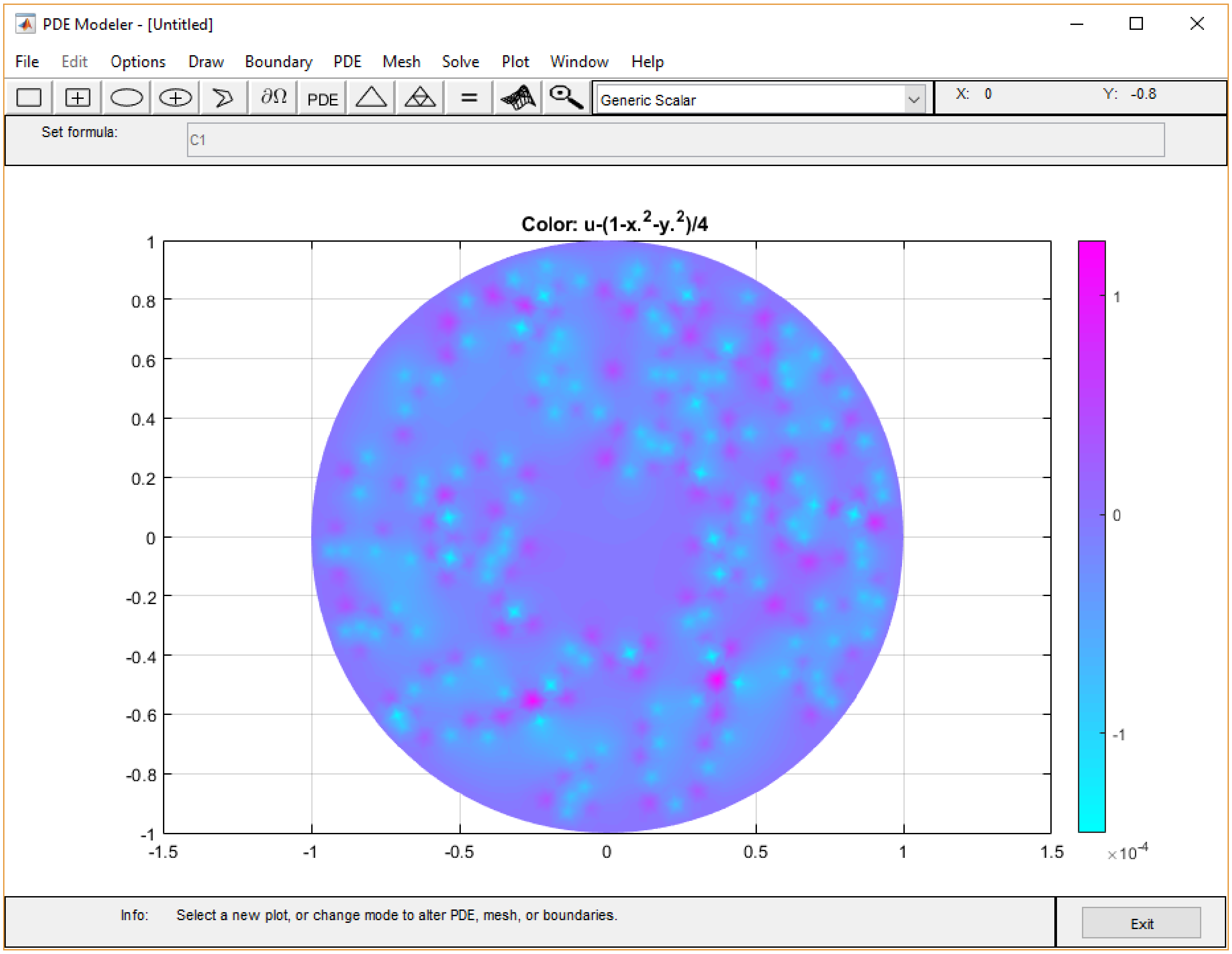
Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

12 Refine the mesh by selecting **Mesh** > **Refine Mesh** or clicking the △ button.



Refined mesh consists of 2109 nodes and 4088 triangles.

13 Compare the numerical solution to the exact solution for the refined mesh. Plot the absolute error.



Set formula: C1

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

14 Export the mesh data and the solution to the MATLAB workspace by selecting **Mesh** > **Export Mesh** and **Solve** > **Export Solution**, respectively.

Poisson's Equation on Unit Disk

This example shows how to numerically solve a Poisson's equation, compare the numerical solution with the exact solution, and refine the mesh until the solutions are close.

The Poisson equation on a unit disk with zero Dirichlet boundary condition can be written as $-\Delta u = 1$ in Ω, $u = 0$ on $\delta\Omega$, where Ω is the unit disk. The exact solution is

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

For most PDEs, the exact solution is not known. However, the Poisson's equation on a unit disk has a known, exact solution that you can use to see how the error decreases as you refine the mesh.

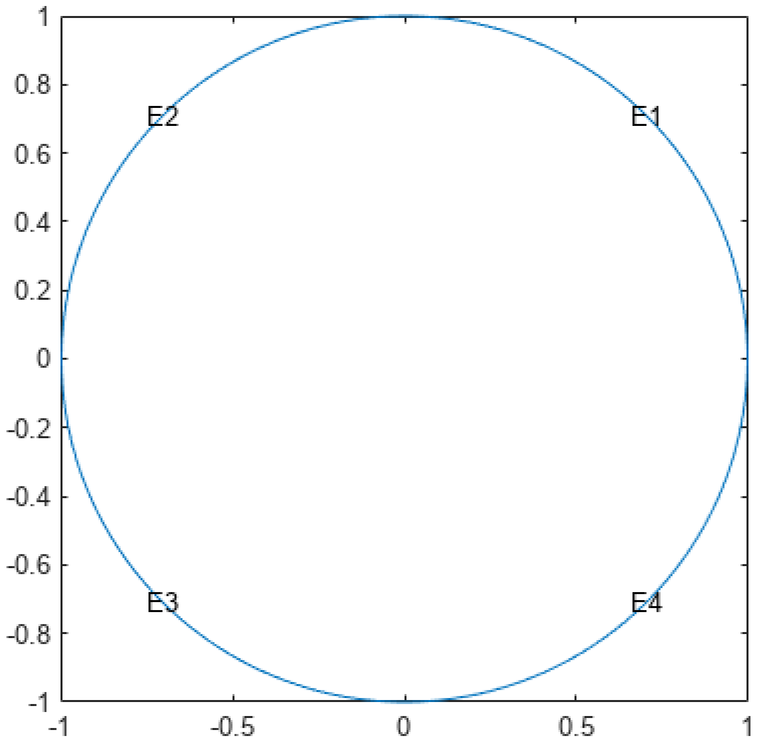
Problem Definition

Create the PDE model and include the geometry.

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

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

```
figure
pdegplot(model,"EdgeLabels","on");
axis equal
```



Specify zero Dirichlet boundary conditions on all edges.

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

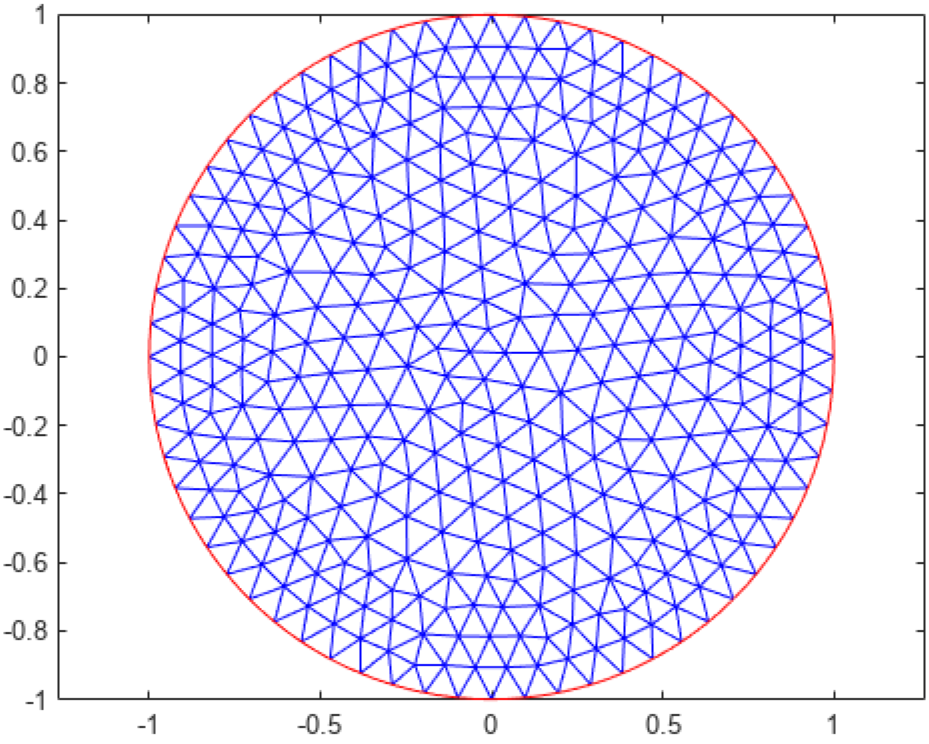
Specify the coefficients.

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

Solution and Error with a Coarse Mesh

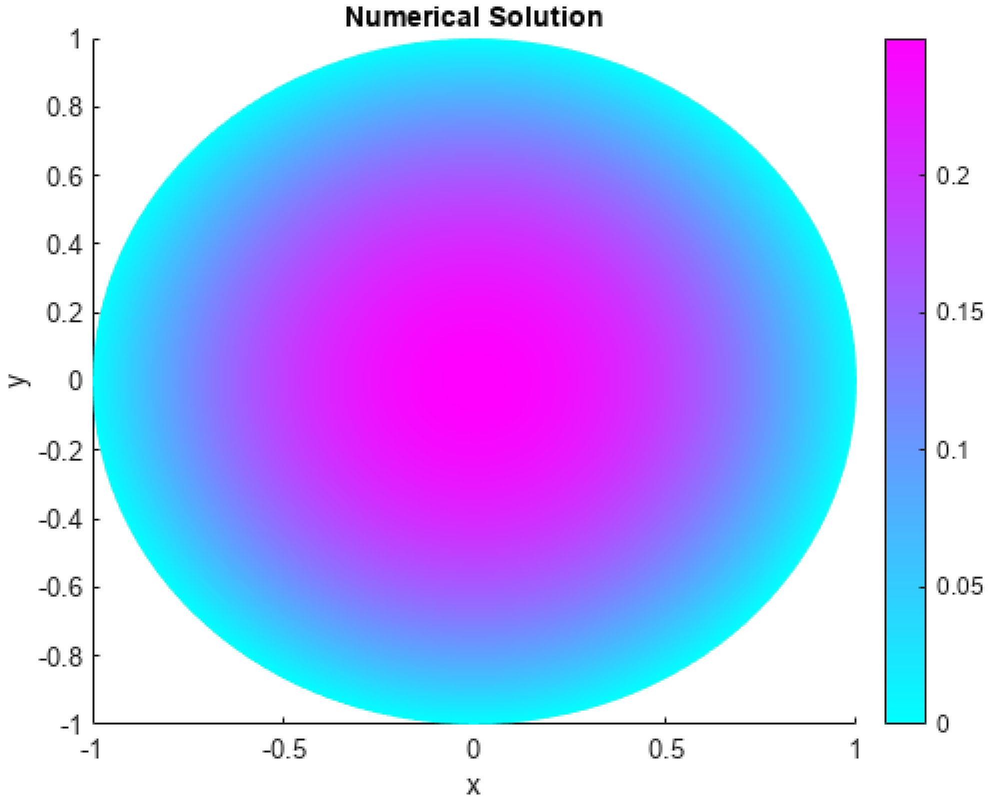
Create a mesh with target maximum element size 0.1.

```
hmax = 0.1;
generateMesh(model,"Hmax",hmax);
figure
pdemesh(model);
axis equal
```



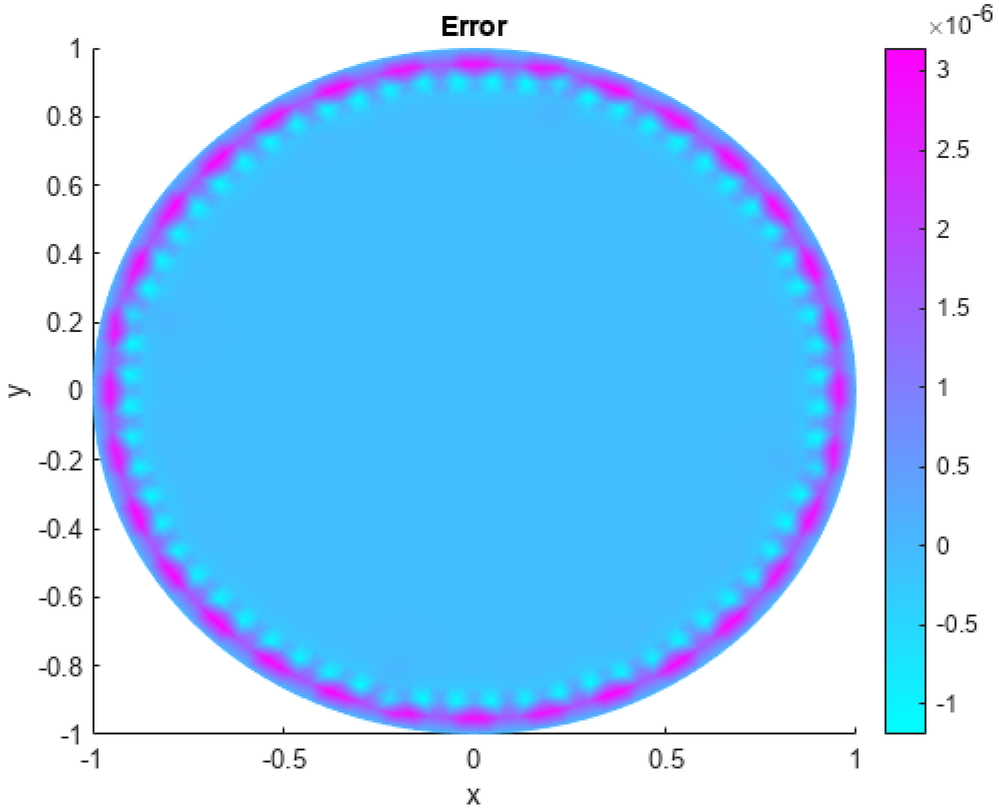
Solve the PDE and plot the solution.

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



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")
```



Solutions and Errors with Refined Meshes

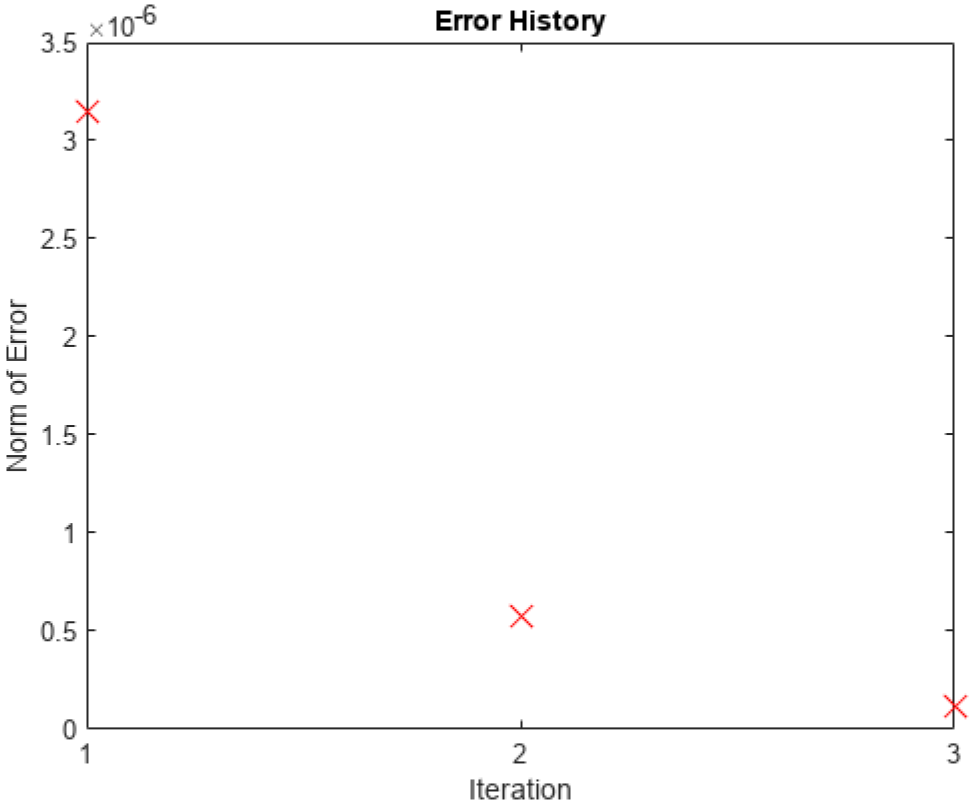
Solve the equation while refining the mesh in each iteration and comparing the result with the exact solution. Each refinement halves the `Hmax` value. Refine the mesh until the infinity norm of the error vector is less than $5 \cdot 10^{-7}$.

```
hmax = 0.1;
error = [];
err = 1;
while err > 5e-7 % run until error <= 5e-7
    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
```

Plot the infinity norm of the error vector for each iteration. The value of the error decreases in each iteration.

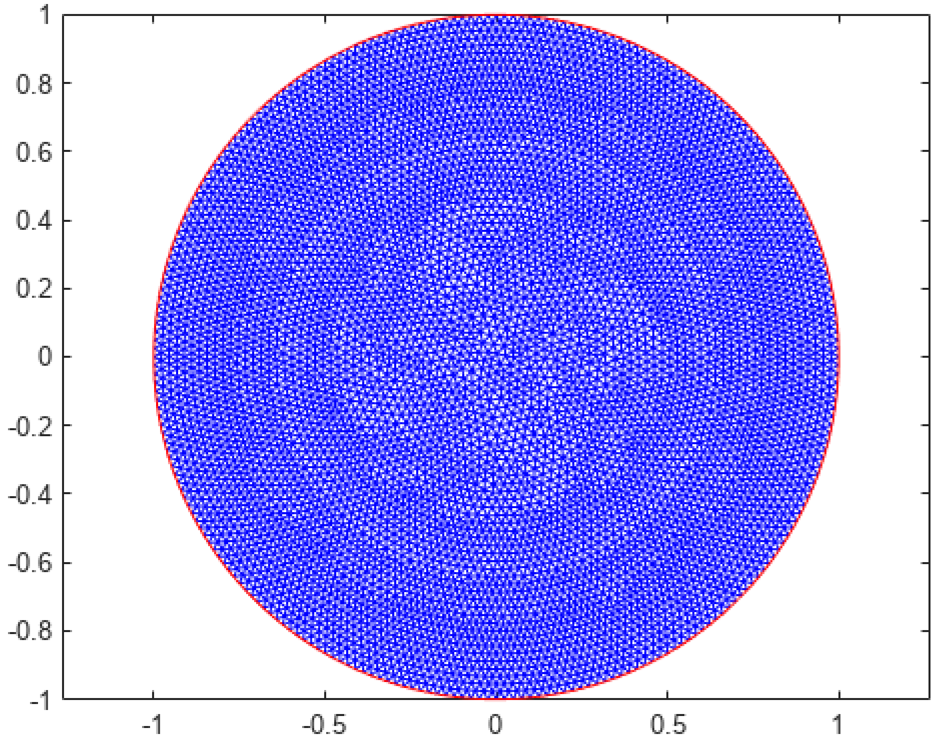
```
plot(error,"rx","MarkerSize",12);
ax = gca;
ax.XTick = 1:numel(error);
title("Error History");
```

```
xlabel("Iteration");
ylabel("Norm of Error");
```

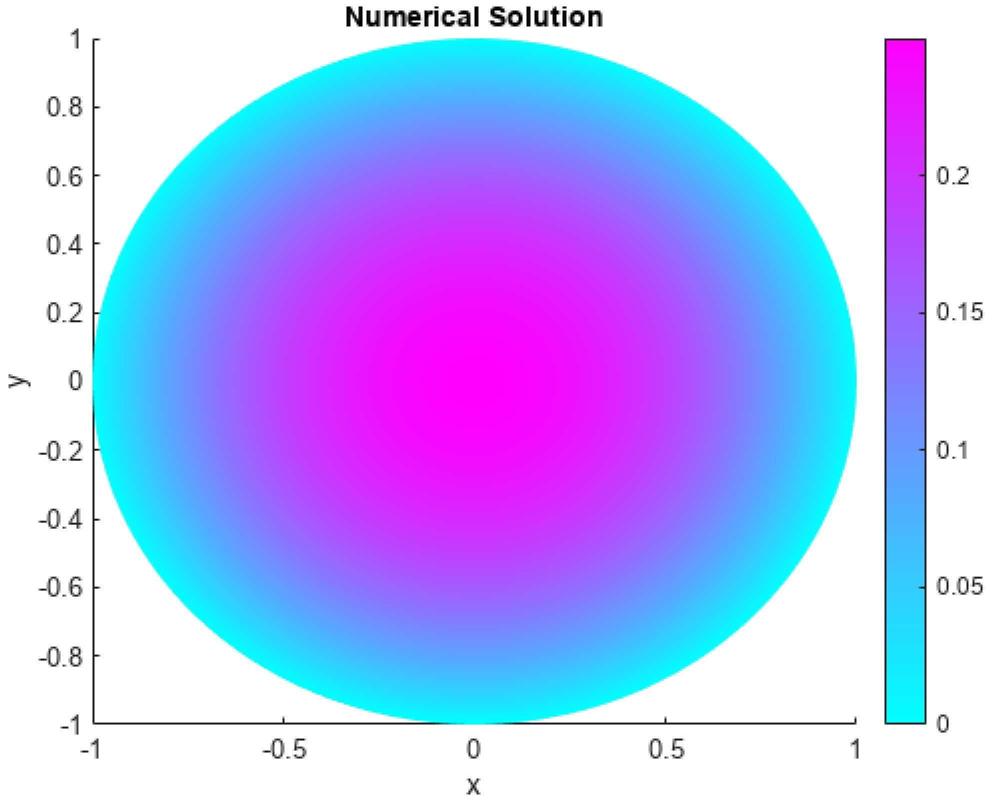


Plot the final mesh and its corresponding solution.

```
figure
pdemesh(model);
axis equal
```

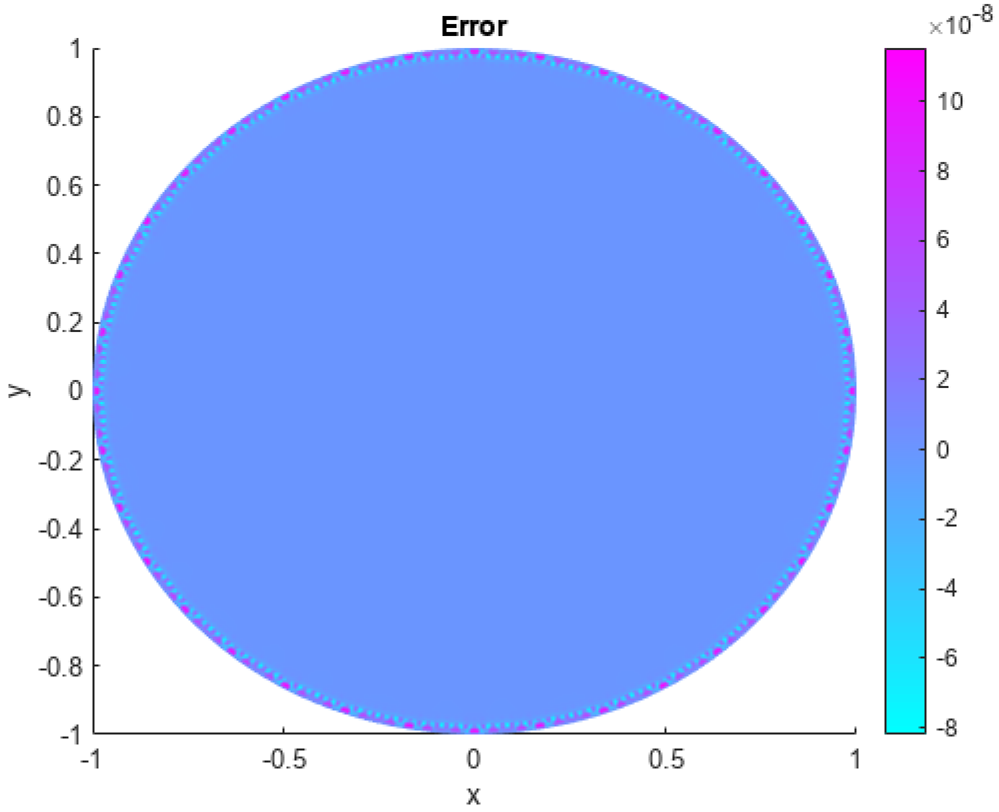


```
figure
pdeplot(model,"XYData",u)
title("Numerical Solution");
xlabel("x")
ylabel("y")
```



Compare the 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")
```


Error
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
y
-1
-0.5
0
0.5
1
x
×10^-8
10
8
6
4
2
0
-2
-4
-6
-8

Scattering Problem

Solve a simple scattering problem, where you compute the waves reflected by an object illuminated by incident waves. This example shows how to solve a scattering problem using the electromagnetic workflow. For the general PDE workflow, see “Helmholtz Equation on Disk with Square Hole” on page 3-170.

For this problem, assume that the domain is an infinite horizontal membrane is subjected to small vertical displacements. The membrane is fixed at the object boundary. The medium is homogeneous, and the phase velocity (propagation speed) of a wave, α, is constant. For this problem, assume α=1. In this case, the frequency wave number equals the frequency, $k = \omega/\alpha = \omega$.

To solve the scattering problem, first create an electromagnetic model for harmonic analysis.

```
emagmodel = createpde("electromagnetic","harmonic");
```

Specify the frequency as 4π.

```
omega = 4*pi;
```

Represent the square surface with a diamond-shaped hole. Define a diamond in a square, place them in one matrix, and create a set formula that subtracts the diamond from the square.

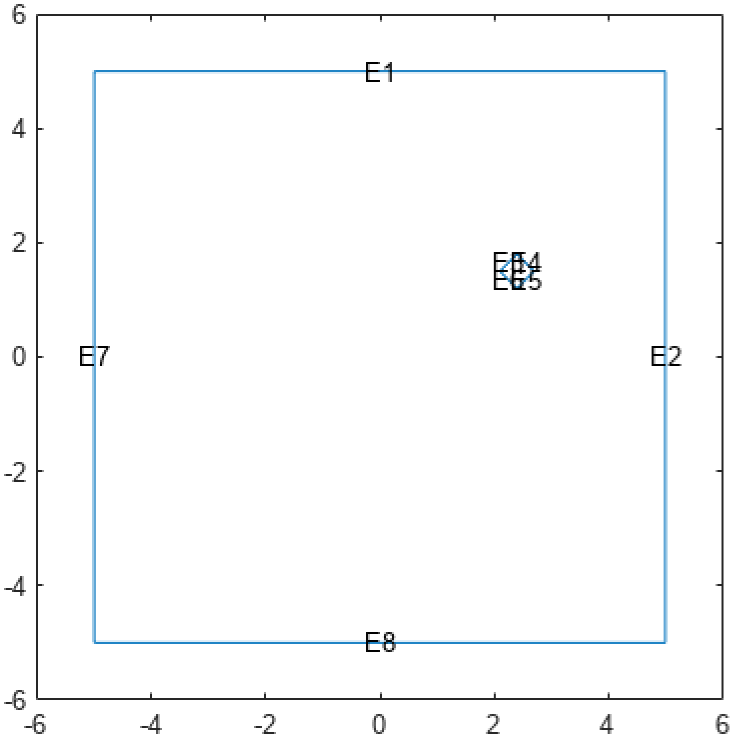
```
square = [3; 4; -5; -5; 5; 5; -5; 5; 5; -5];
diamond = [2; 4; 2.1; 2.4; 2.7; 2.4; 1.5; 1.8; 1.5; 1.2];
gd = [square,diamond];
ns = char('square','diamond')';
sf = 'square - diamond';
```

Create the geometry.

```
g = decsg(gd,sf,ns);
geometryFromEdges(emagmodel,g);
```

Include the geometry in the model and plot it with the edge labels.

```
figure;
pdegplot(emagmodel,"EdgeLabels","on");
xlim([-6,6])
ylim([-6,6])
```



Specify the vacuum permittivity and permeability values as 1.

```
emagmodel.VacuumPermittivity = 1;
emagmodel.VacuumPermeability = 1;
```

Specify the relative permittivity, relative permeability, and conductivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1, ...
                                    "RelativePermeability",1, ...
                                    "Conductivity",0);
```

Apply the absorbing boundary condition on the edges of the square. Specify the thickness and attenuation rate for the absorbing region by using the `Thickness`, `Exponent`, and `Scaling` arguments.

```
electromagneticBC(emagmodel,"Edge",[1 2 7 8], ...
                            "FarField","absorbing", ...
                            "Thickness",2, ...
                            "Exponent",4, ...
                            "Scaling",1);
```

Apply the boundary condition on the diamond edges.

```
innerBCFunc = @(location,~) [-exp(-1i*omega*location.x); ...
                            zeros(1,length(location.x))];
bInner = electromagneticBC(emagmodel,"Edge",[3 4 5 6], ...
                                     "ElectricField",innerBCFunc);
```

Generate a mesh.

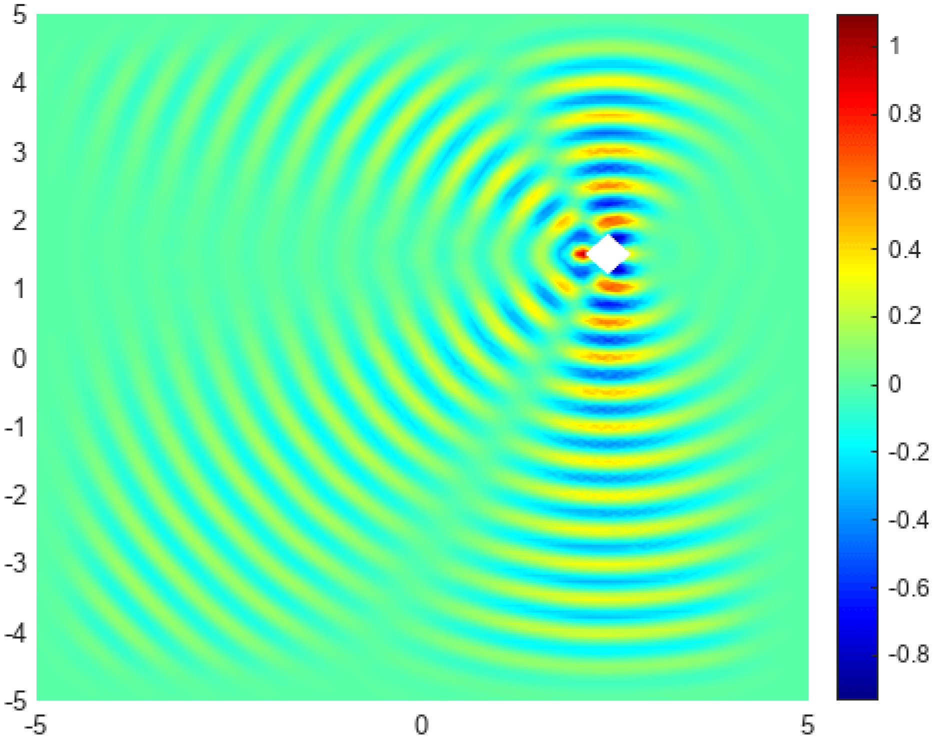
```
generateMesh(emagmodel,"Hmax",0.1);
```

Solve the harmonic analysis model for the frequency $\omega = 4\pi$.

```
result = solve(emagmodel,"Frequency",omega);
```

Plot the real part of the `x`-component of the resulting electric field.

```
u = result.ElectricField;

figure
pdeplot(emagmodel,"XYData",real(u.Ex),"Mesh","off");
colormap(jet)
```



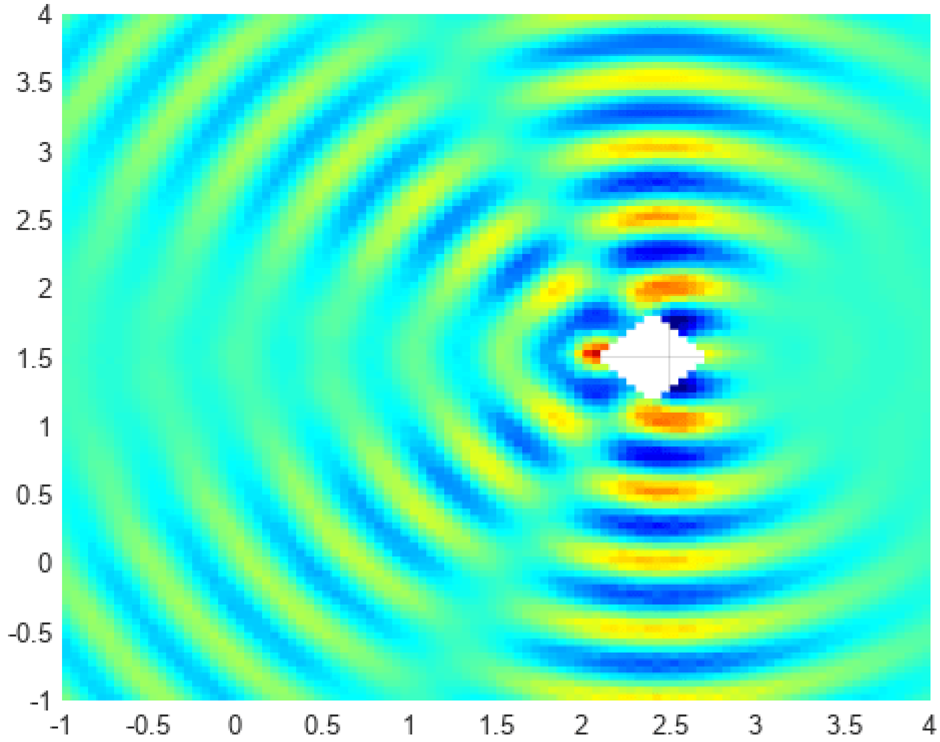
Interpolate the resulting electric field to a grid covering the portion of the geometry, for x and y from -1 to 4.

```
v = linspace(-1,4,101);
[X,Y] = meshgrid(v);
Eintrp = interpolateHarmonicField(result,X,Y);
```

Reshape `Eintrp.Ex` and plot the x-component of the resulting electric field.

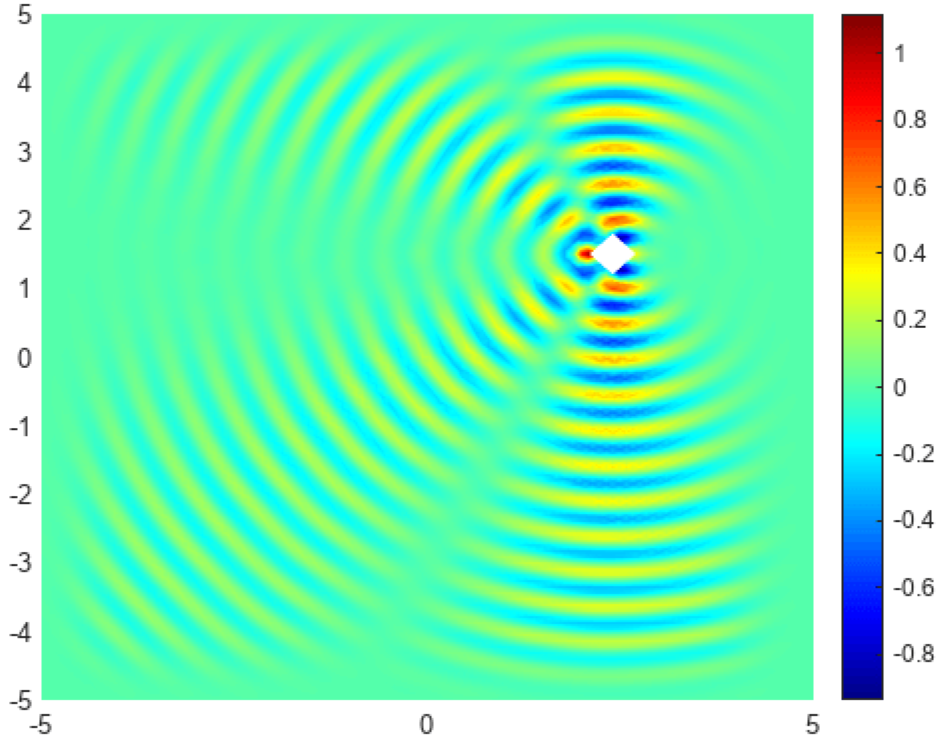
```
EintrpX = reshape(Eintrp.ElectricField.Ex,size(X));

figure
```

```
surf(X,Y,real(EintrpX),"LineStyle","none");
view(0,90)
colormap(jet)
```



Using the solutions for a vector of frequencies, create an animation showing the corresponding solution to the time-dependent wave equation.

```
result = solve(emagmodel,"Frequency",omega/10:omega);
figure
for m = 1:length(omega/10:omega)
    u = result.ElectricField;
    pdeplot(emagmodel,"XYData",real(u.Ex(:,m)),"Mesh","off");
    colormap(jet)
    M(m) = getframe;
end
```



To play the animation, use the `movie` function. For example, to play the animation five times in a loop with 3 frames per second, use the `movie(M,5,3)` command.

Scattering Problem: PDE Modeler App

This example shows how to solve a simple scattering problem, where you compute the waves reflected by a square object illuminated by incident waves that are coming from the left. This example uses the PDE Modeler app. For the programmatic workflow, see “Scattering Problem” on page 3-232.

For this problem, assume an infinite horizontal membrane subjected to small vertical displacements U. The membrane is fixed at the object boundary. The medium is homogeneous, and the phase velocity (propagation speed) of a wave, α, is constant. The wave equation is

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

The solution U is the sum of the incident wave V and the reflected wave R:

$$U = V + R$$

When the illumination is harmonic in time, you can compute the field by solving a single steady problem. Assume that the incident wave is a plane wave traveling in the –x direction:

$$V(x, y, t) = e^{i(-kx - \omega t)} = e^{-ikx}e^{-i\omega t}$$

The reflected wave can be decomposed into spatial and time components:

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

Now you can rewrite the wave equation as the Helmholtz equation for the spatial component of the reflected wave with the wave number $k = \omega/\alpha$:

$$-\Delta r - k^2 r = 0$$

The Dirichlet boundary condition for the boundary of the object is $U = 0$, or in terms of the incident and reflected waves, $R = -V$. For the time-harmonic solution and the incident wave traveling in the –x direction, you can write this boundary condition as follows:

$$r(x, y) = -e^{-ikx}$$

The reflected wave R travels outward from the object. The condition at the outer computational boundary must allow waves to pass without reflection. Such conditions are usually called nonreflecting. As $\left|\vec{x}\right|$ approaches infinity, R approximately satisfies the one-way wave equation

$$\frac{\partial R}{\partial t} + \alpha \vec{\xi} \cdot \nabla R = 0$$

This equation considers only the waves moving in the positive ξ-direction. Here, ξ is the radial distance from the object. With the time-harmonic solution, this equation turns into the generalized Neumann boundary condition

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

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

1. Open the PDE Modeler app by using the `pdeModeler` command.

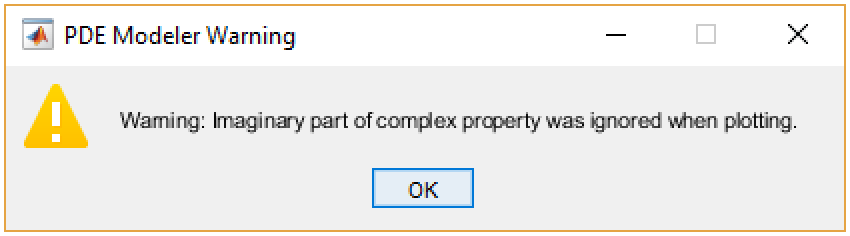
2. Set the *x*-axis limit to `[0.1 1.5]` and the *y*-axis limit to `[0 1]`. To do this, select **Options** > **Axes Limits** and set the corresponding ranges.
3. Display grid lines. To do this:
   - a. Select **Options** > **Grid Spacing** and clear the **Auto** checkboxes.
   - b. Enter **X-axis linear spacing** as `0.1:0.05:1.5` and **Y-axis linear spacing** as `0:0.05:1`.
   - c. Select **Options** > **Grid**.
4. Align new shapes to the grid lines by selecting **Options** > **Snap**.
5. Draw a square with sides of length 0.1 and a center in `[0.8 0.5]`. To do this, first click the ⊞ button. Then right-click the origin and drag to draw a square. Right-clicking constrains the shape you draw so that it is a square rather than a rectangle. If the square is not a perfect square, double-click it. In the resulting dialog box, specify the exact location of the bottom left corner and the side length.
6. Rotate the square by 45 degrees. To do this, select **Draw** > **Rotate...** and enter 45 in the resulting dialog box. The rotated square represents the illuminated object.
7. Draw a circle with a radius of 0.45 and a center in `[0.8 0.5]`. To do this, first click the ⊕ button. Then right-click the origin and drag to draw a circle. Right-clicking constrains the shape you draw so that it is 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.
8. Model the geometry by entering `C1-SQ1` in the **Set formula** field.
9. Check that the application mode is set to `Generic Scalar`.
10. 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 perimeter of the circle, the boundary condition is the Neumann boundary condition with `q = -ik`, where the wave number `k = 60` corresponds to a wavelength of about 0.1 units. Enter `g = 0` and `q = -60*i`.
    - For the perimeter of the square, the boundary condition is the Dirichlet boundary condition:

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

    In this problem, because the reflected wave travels in the –*x* direction, the boundary condition is $r = -e^{-ikx}$. Enter `h = 1` and `r = -exp(-i*60*x)`.
11. Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. The Helmholtz equation is a wave equation, but in Partial Differential Equation Toolbox you can treat it as an elliptic equation with `a = -k`^{2}. Specify `c = 1`, `a = -3600`, and `f = 0`.
12. Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.

    For sufficient accuracy, you need about 10 finite elements per wavelength. The outer boundary must be located a few object diameters away from the object itself. Refine the mesh by selecting **Mesh** > **Refine Mesh**. Refine the mesh two more times to achieve the required resolution.
13. Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.

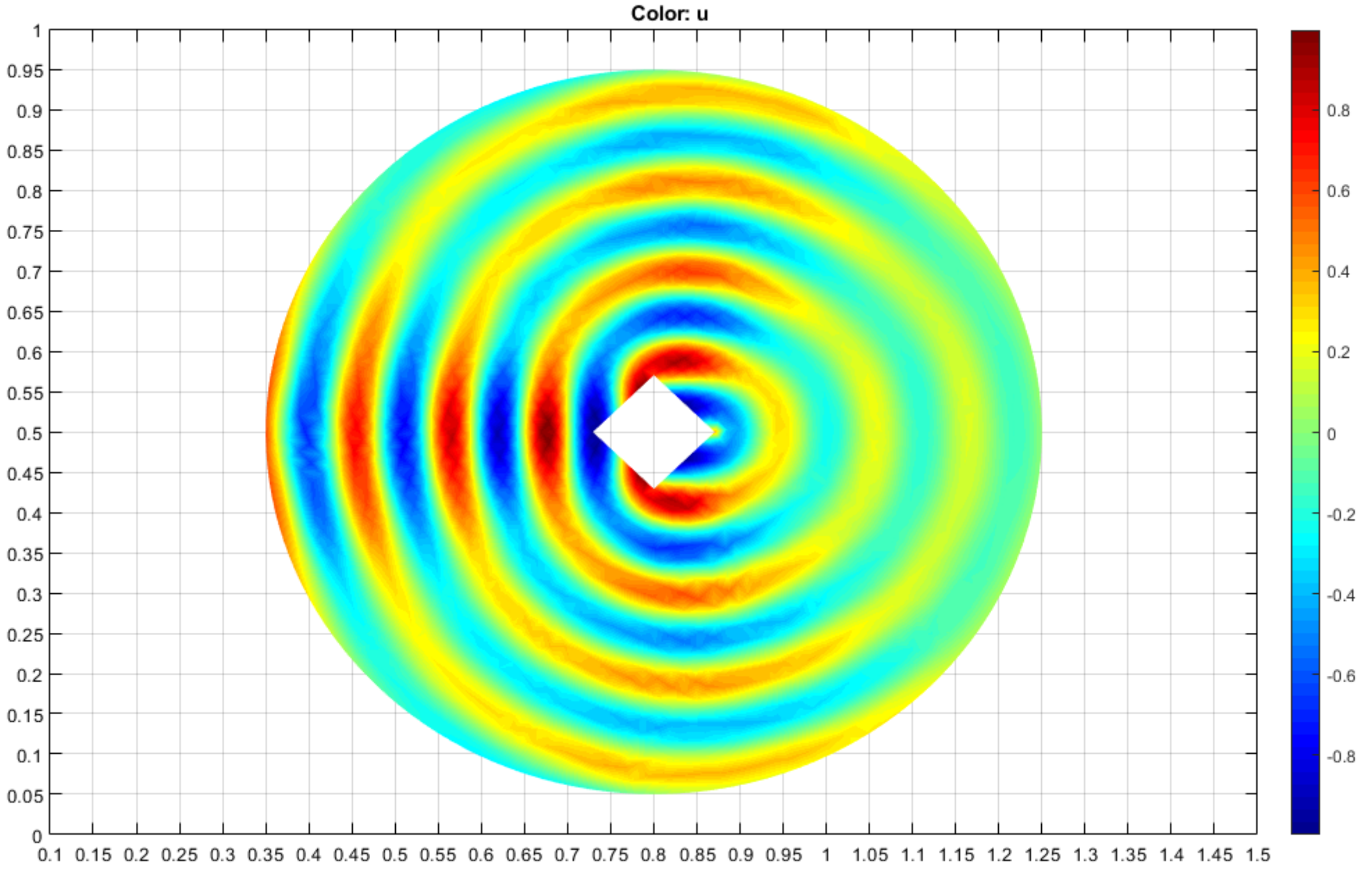
    The solution is complex. When plotting the solution, you get a warning message.



PDE Modeler Warning

Warning: Imaginary part of complex property was ignored when plotting.

14 Plot the reflected waves. Change the colormap to `jet` by selecting **Plot** > **Parameters** and then selecting `jet` from the **Colormap** drop-down menu.



15 Animate the solution for the time-dependent wave equation. To do this:

a Export the mesh data and the solution to the MATLAB workspace by selecting **Mesh** > **Export Mesh** and **Solve** > **Export Solution**, respectively.

b Enter the following commands in the MATLAB Command Window.

```
figure
maxu = max(abs(u));
m = 10;
for j = 1:m,
   uu = real(exp(-j*2*pi/10*sqrt(-1))*u);
   pdeplot(p,e,t,'XYData',uu,'ColorBar','off','Mesh','off');
   colormap(jet)
```

```
      caxis([-maxu maxu]);
      axis tight
      ax = gca;
      ax.DataAspectRatio = [1 1 1];
      axis off
      M(:,j) = getframe;
  end
movie(M);
```

Minimal Surface Problem

This example shows how to solve the minimal surface equation

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

on the unit disk $\Omega=\left\{(x,y)\middle|x^2+y^2\le 1\right\}$, with $u(x,y)=x^2$ on the boundary $\partial\Omega$. An elliptic equation in the toolbox form is

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

Therefore, for the minimal surface problem, the coefficients are:

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

Because the coefficient c is a function of the solution u, the minimal surface problem is a nonlinear elliptic problem.

To solve the minimal surface problem using the programmatic workflow, first create a PDE model with a single dependent variable.

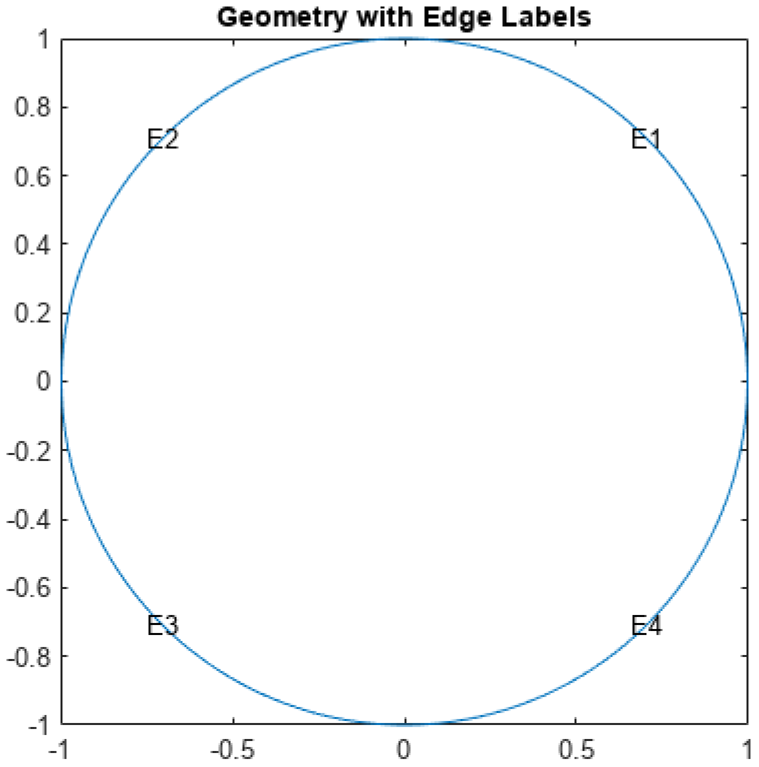
```
model = createpde;
```

Create the geometry and include it in the model. The `circleg` function represents this geometry.

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

Plot the geometry with the edge labels.

```
pdegplot(model,"EdgeLabels","on")
axis equal
title("Geometry with Edge Labels")
```



Specify the coefficients.

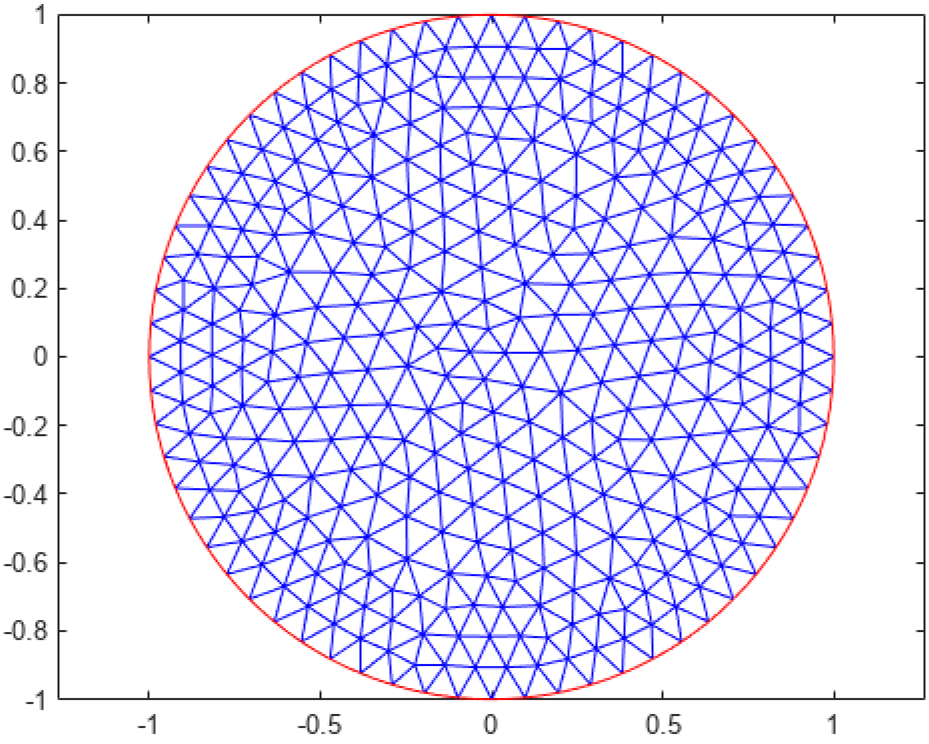
```
a = 0;
f = 0;
cCoef = @(region,state) 1./sqrt(1+state.ux.^2 + state.uy.^2);
specifyCoefficients(model,"m",0,"d",0,"c",cCoef,"a",a,"f",f);
```

Specify the boundary conditions using the function $u(x, y) = x^2$.

```
bcMatrix = @(region,~)region.x.^2;
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1:model.Geometry.NumEdges, ...
                             "u",bcMatrix);
```

Generate and plot a mesh.

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



Clear figure for future plots.

```
clf
```

Solve the problem by using the `solvepde` function. Because the problem is nonlinear, `solvepde` invokes a nonlinear solver. Observe the solver progress by setting the `SolverOptions.ReportStatistics` property of the model to `'on'`.

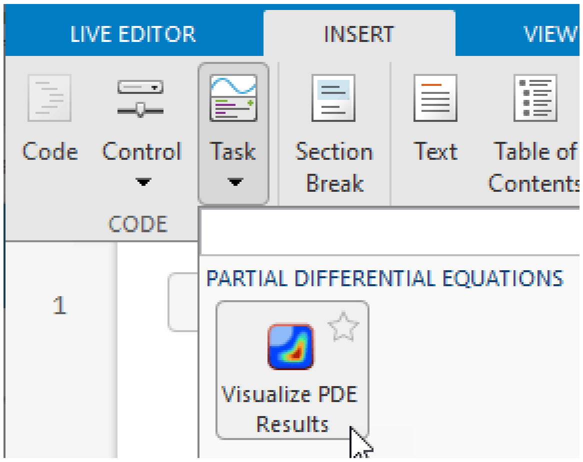
```
model.SolverOptions.ReportStatistics = 'on';
result = solvepde(model);
```

```
Iteration     Residual     Step size  Jacobian: Full
   0          1.8540e-02
   1          2.8715e-04   1.0000000
   2          1.2145e-06   1.0000000
```

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

Plot the solution by using the **Visualize PDE Results** Live Editor task. First, create a new live script by clicking the **New Live Script** button in the **File** section on the **Home** tab.

On the **Live Editor** tab, select **Task** > **Visualize PDE Results**. This action inserts the task into your script.

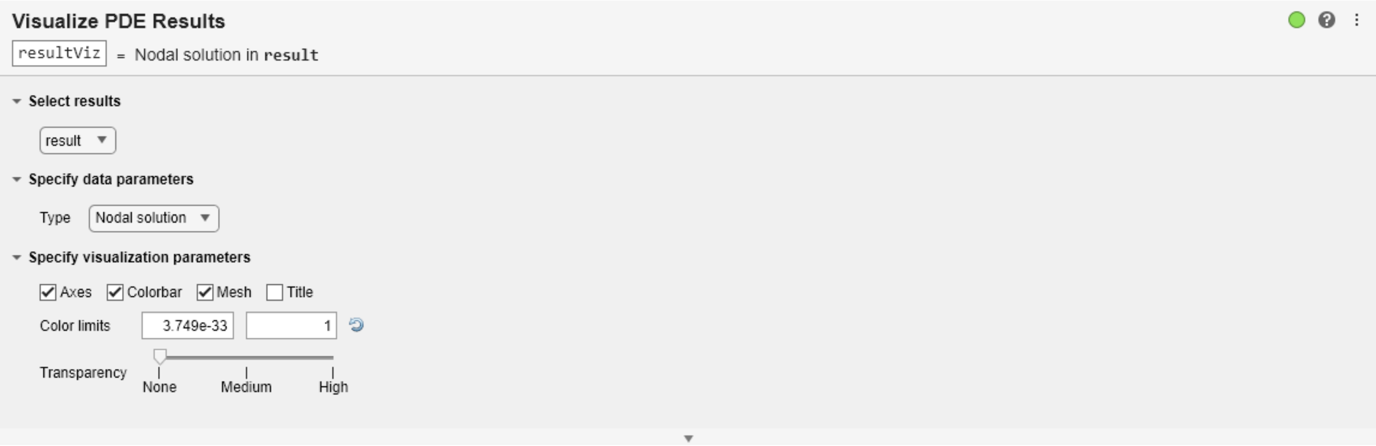


To plot the solution, follow these steps.

1 In the **Select results** section of the task, select `result` from the drop-down list.

2 In the **Specify data parameters** section of the task, set **Type** to *Nodal solution*.

3 In the **Specify visualization parameters** section of the task, select the **Mesh** check box.



Visualize PDE Results

`resultViz` = Nodal solution in `result`

Select results

result

Specify data parameters

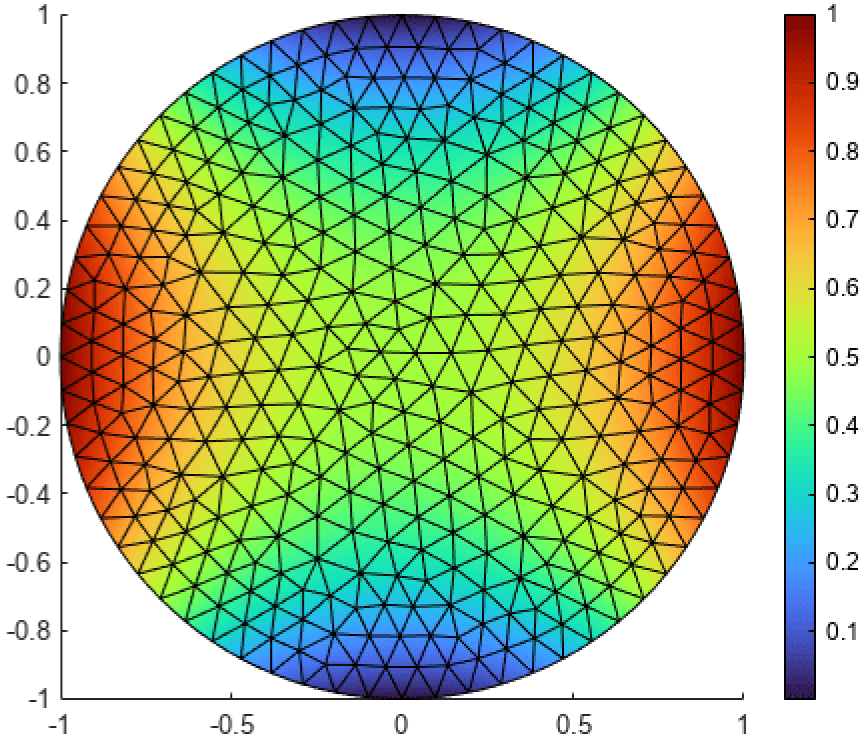
Type Nodal solution

Specify visualization parameters

☑ Axes ☑ Colorbar ☑ Mesh ☐ Title

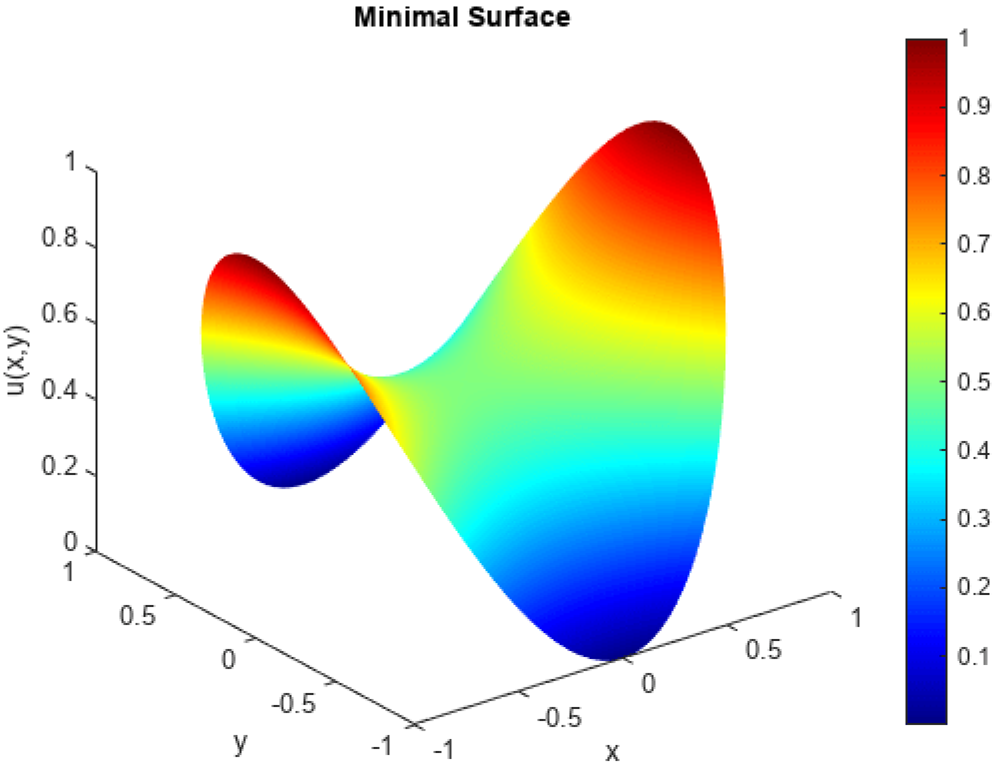
Color limits 3.749e-33 1

Transparency None Medium High



You also can plot the solution at the MATLAB® command line by using the `pdeplot` function. For example, plot the solution as a 3-D plot, using the solution values for plot heights.

```
figure;
pdeplot(model,"XYData",u,"ZData",u);
xlabel x
ylabel y
zlabel u(x,y)
title("Minimal Surface")
colormap jet
```


Minimal Surface
u(x,y)
1
0.8
0.6
0.4
0.2
0
y
1
0.5
0
-0.5
-1
x
-1
-0.5
0
0.5
1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

Minimal Surface Problem: PDE Modeler App

This example shows how to solve the minimal surface equation

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

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

This example uses the PDE Modeler app. For the programmatic workflow, see “Minimal Surface Problem” on page 3-241.

An elliptic equation in the toolbox form is

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

Therefore, for the minimal surface problem the coefficients are as follows:

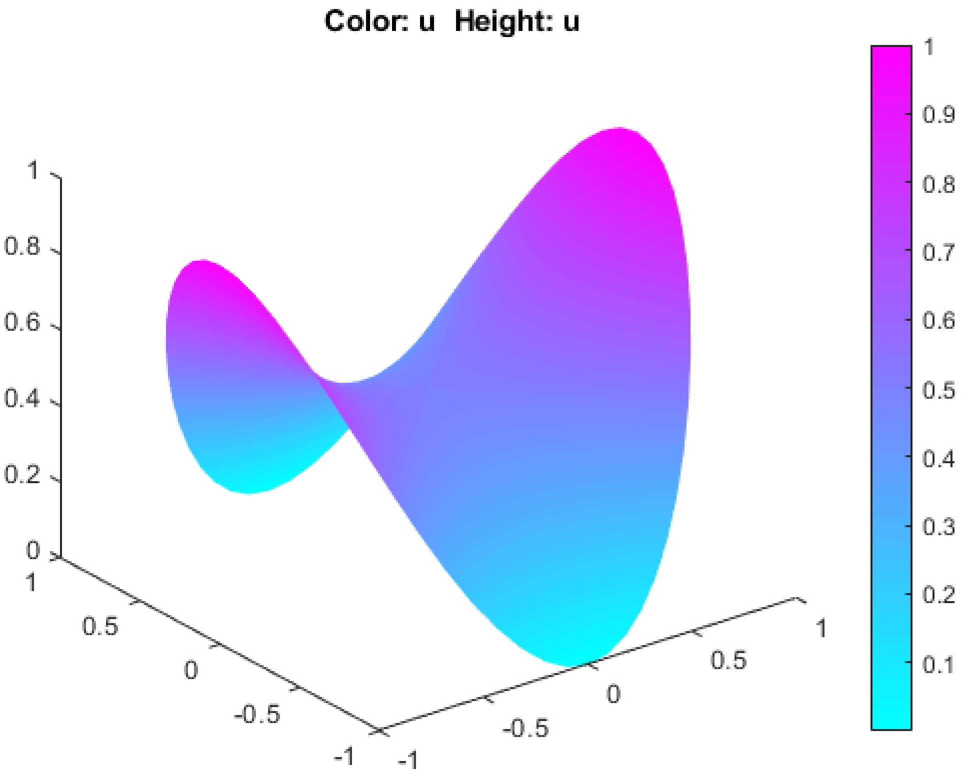
$$c=\frac{1}{\sqrt{1+|\nabla u|^2}},\ a=0,\ f=0$$

Because the coefficient c is a function of the solution u, the minimal surface problem is a nonlinear elliptic problem.

To solve the minimal surface problem in the PDE Modeler app, follow these steps:

1. Model the surface as a unit circle.

   ```
   pdecirc([0 0 1])
   ```
2. Check that the application mode is set to `Generic Scalar`.
3. Specify the boundary conditions. To do this:
   - **a** Switch to boundary mode by clicking the ∂Ω button or selecting **Boundary** > **Boundary Mode**.
   - **b** Select all boundaries by selecting **Edit** > **Select All**.
   - **c** Select **Boundary** > **Specify Boundary Conditions**.
   - **d** Specify the Dirichlet boundary condition $u = x^2$. To do this, specify `h = 1`, `r = x.^2`.
4. Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Specify `c = 1./sqrt(1+ux.^2+uy.^2)`, `a = 0`, and `f = 0`.
5. Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.
6. Refine the mesh by selecting **Mesh** > **Refine Mesh**.
7. Choose the nonlinear solver. To do this, select **Solve** > **Parameters** and check **Use nonlinear solver**. Set the tolerance parameter to `0.001`.
8. Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar.
9. Plot the solution in 3-D. To do this, select **PlotParameters**. In the resulting dialog box, select **Height (3-D plot)**.


Color: u Height: u
1
0.8
0.6
0.4
0.2
0
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

Poisson's Equation with Point Source and Adaptive Mesh Refinement

This example shows how to solve a Poisson's equation with a delta-function point source on the unit disk using the `adaptmesh` function.

Specifically, solve the Poisson's equation

$$-\Delta u = \delta(x, y)$$

on the unit disk with zero Dirichlet boundary conditions. The exact solution expressed in polar coordinates is

$$u(r, \theta) = \frac{\log(r)}{2\pi},$$

which is singular at the origin.

By using adaptive mesh refinement, Partial Equation Toolbox™ can accurately find the solution everywhere away from the origin.

The following variables define the problem:

- `c`, `a`: The coefficients of the PDE.
- `f`: A function that captures a point source at the origin. It returns 1/area for the triangle containing the origin and 0 for other triangles.

```
c = 1;
a = 0;
f = @circlef;
```

Create a PDE Model with a single dependent variable.

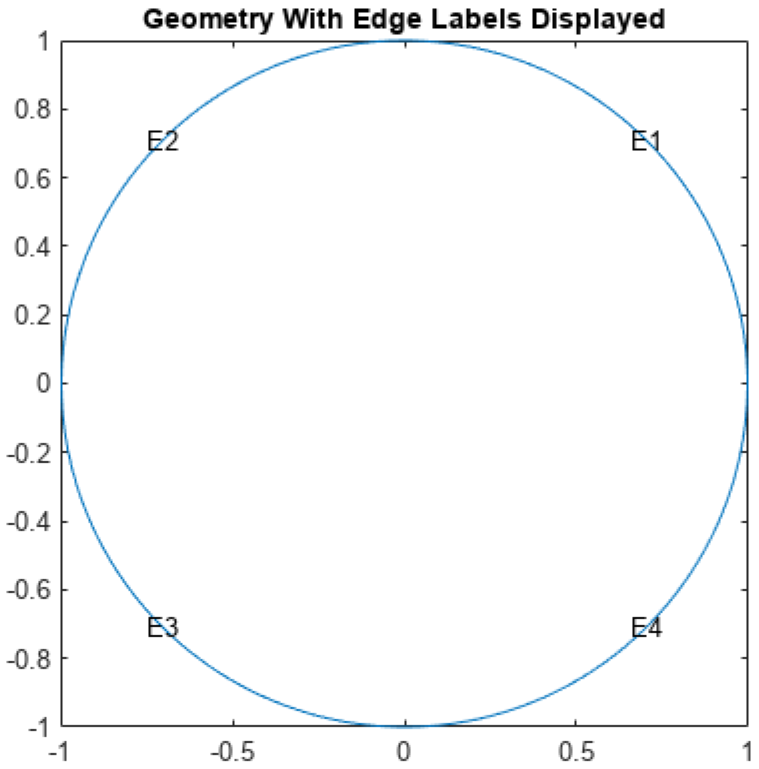
```
numberOfPDE = 1;
model = createpde(numberOfPDE);
```

Create a geometry and include it in the model.

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

Plot the geometry and display the edge labels.

```
figure;
pdegplot(model,"EdgeLabels","on");
axis equal
title("Geometry With Edge Labels Displayed")
```



Specify the zero solution at all four outer edges of the circle.

```
applyBoundaryCondition(model,"dirichlet","Edge",(1:4),"u",0);
```

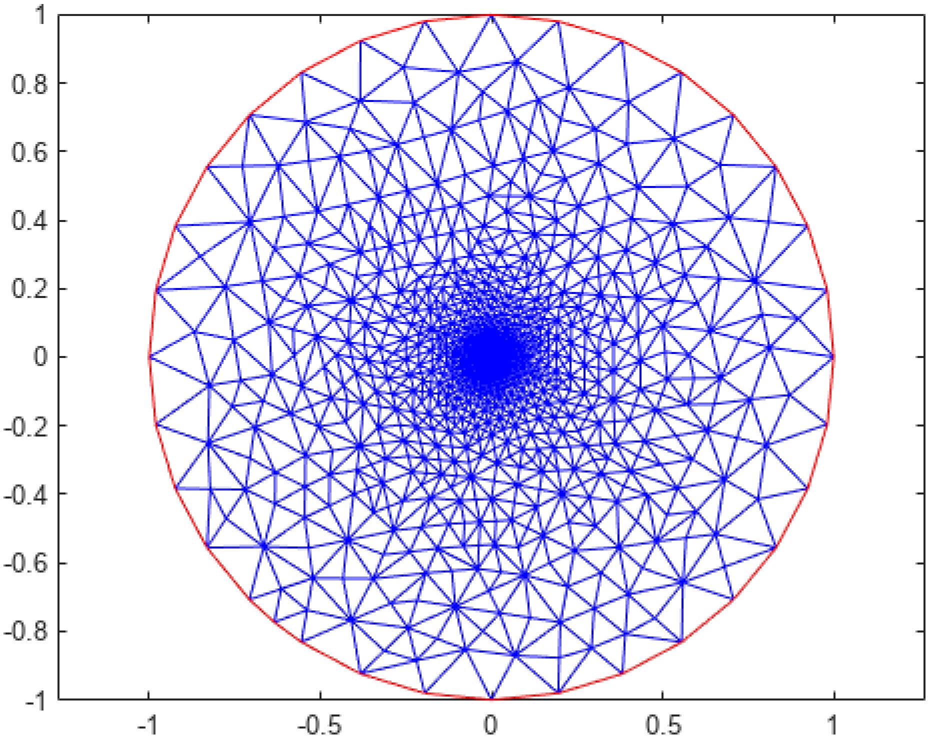
`adaptmesh` solves elliptic PDEs using the adaptive mesh generation. The `tripick` parameter lets you specify a function that returns which triangles will be refined in the next iteration. `circlepick` returns triangles with computed error estimates greater a given tolerance. The tolerance is provided to `circlepick` using the "par" parameter.

```
[u,p,e,t] = adaptmesh(g,model,c,a,f,"tripick", ...
                                    "circlepick", ...
                                    "maxt",2000, ...
                                    "par",1e-3);
```

```
Number of triangles: 258
Number of triangles: 515
Number of triangles: 747
Number of triangles: 1003
Number of triangles: 1243
Number of triangles: 1481
Number of triangles: 1705
Number of triangles: 1943
Number of triangles: 2155

Maximum number of triangles obtained.
```

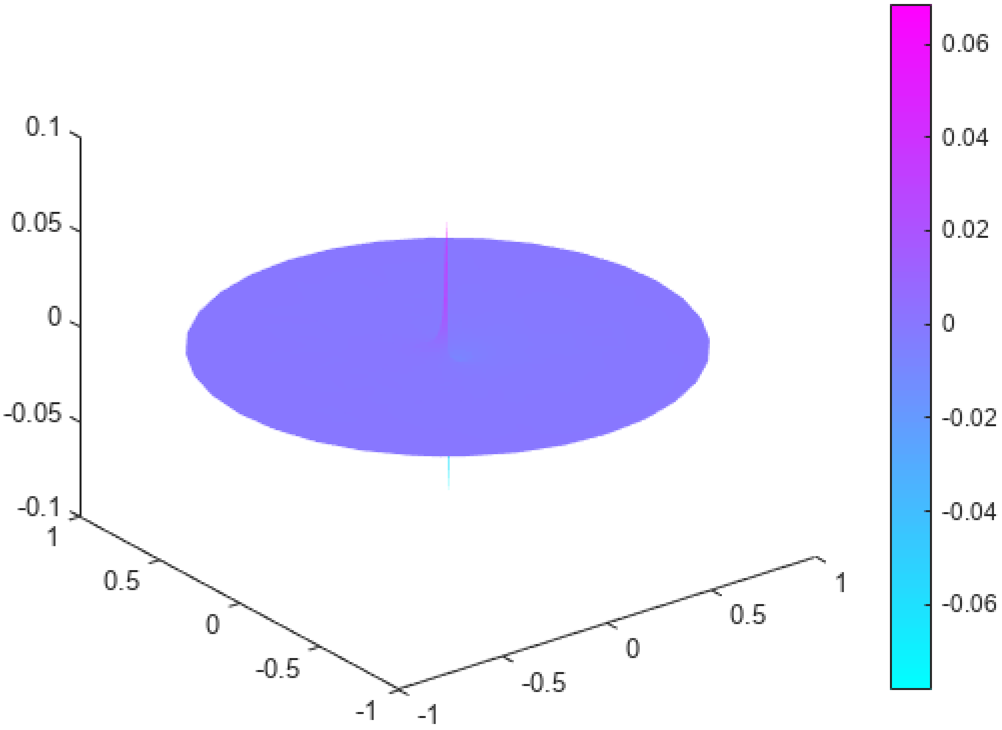
Plot the finest mesh.

```
figure;
pdemesh(p,e,t);
axis equal
```



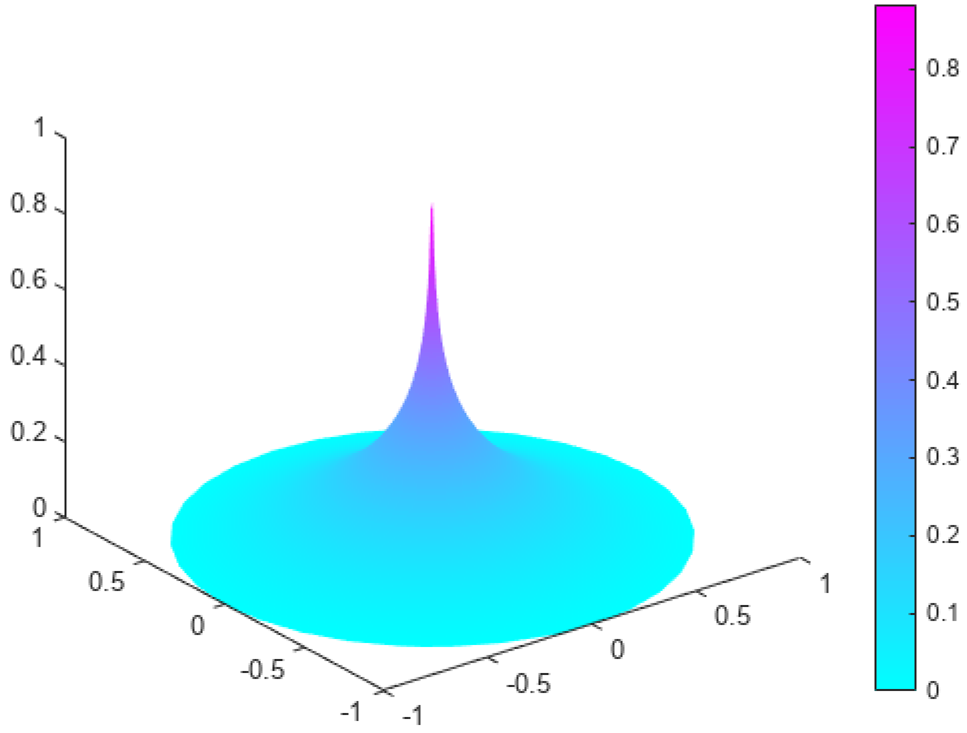
Plot the error values.

```
x = p(1,:)';
y = p(2,:)';
r = sqrt(x.^2+y.^2);
uu = -log(r)/2/pi;
figure;
pdeplot(p,e,t,"XYData",u-uu,"ZData",u-uu,"Mesh","off");
```



Plot the FEM solution on the finest mesh.

```
figure;
pdeplot(p,e,t,"XYData",u,"ZData",u,"Mesh","off");
```


1
0.8
0.6
0.4
0.2
0
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

Heat Transfer in Block with Cavity: PDE Modeler App

This example shows how to solve a heat equation that describes the diffusion of heat in a body. This example uses the PDE Modeler app. For programmatic workflow, see “Heat Transfer in Block with Cavity” on page 3-258.

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:

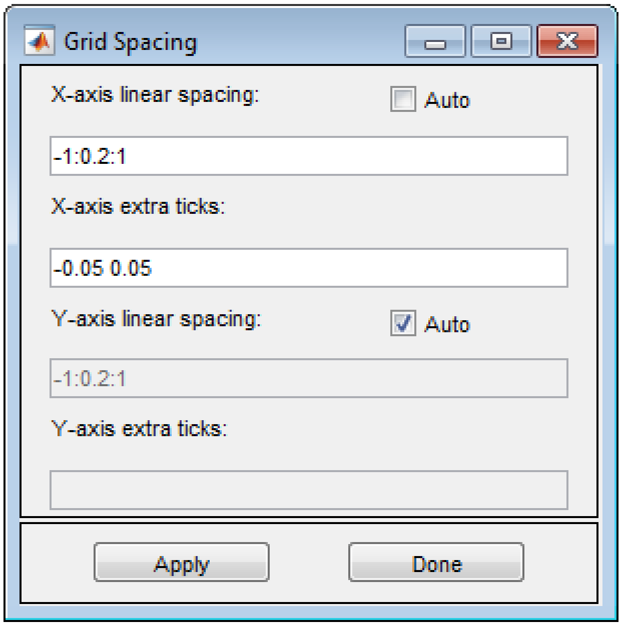
$$d\frac{\partial u}{\partial t} - \Delta u = 0$$

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

1. Open the PDE Modeler app by using the `pdeModeler` command.

   ```
   pdeModeler
   ```
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**.



X-axis linear spacing: ☐ Auto

-1:0.2:1

X-axis extra ticks:

-0.05 0.05

Y-axis linear spacing: ☑ Auto

-1:0.2:1

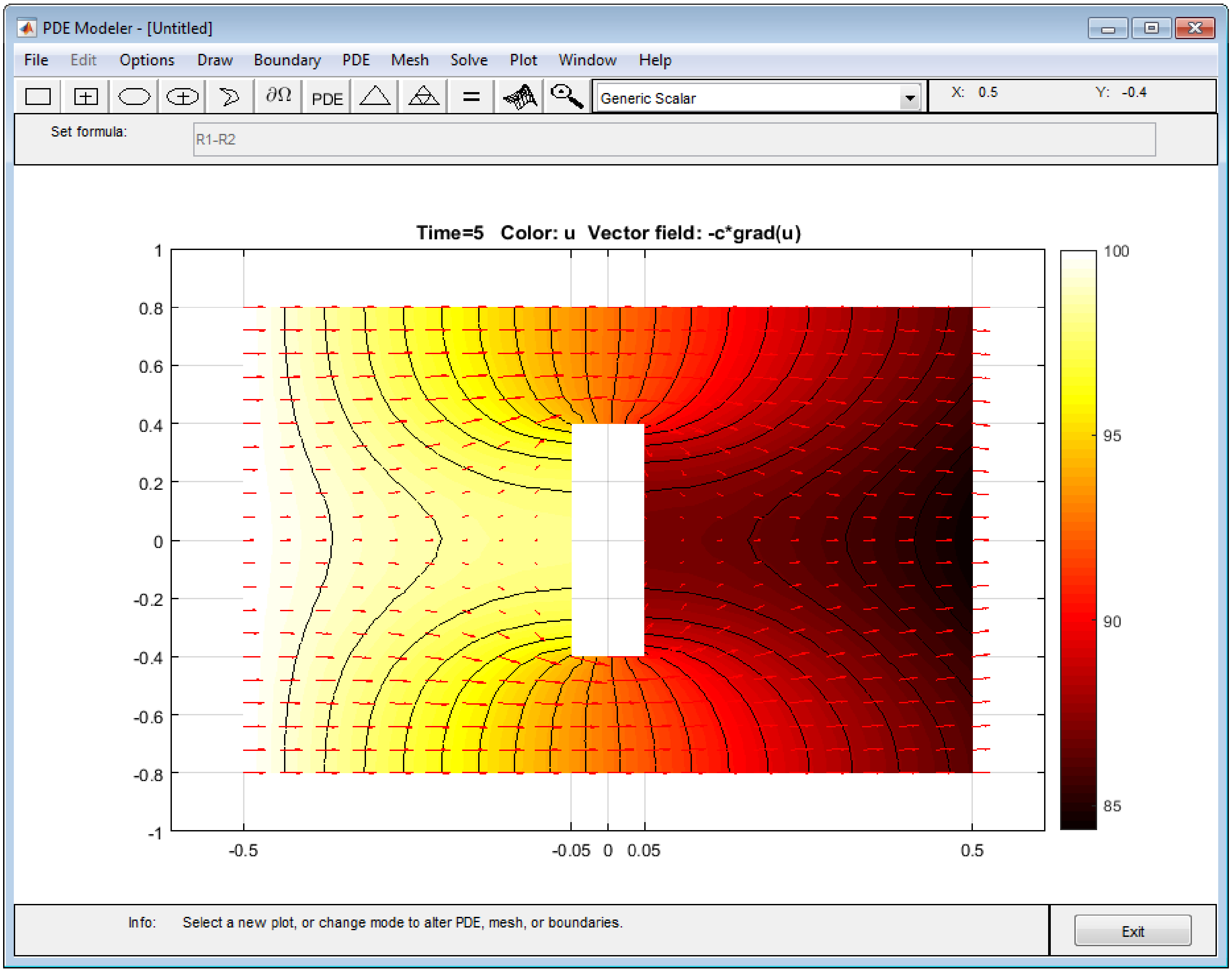
Y-axis extra ticks:

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.



Generic Scalar

X: 0.5 Y: -0.4

Set formula: R1-R2

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

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 Transfer in Block with Cavity

This example shows how to solve for the heat distribution in a block with cavity.

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.

Create Thermal Analysis Model

The first step in solving a heat transfer problem is to create a thermal analysis model. This is a container that holds the geometry, thermal material properties, internal heat sources, temperature on the boundaries, heat fluxes through the boundaries, mesh, and initial conditions.

```
thermalmodel = createpde("thermal","transient");
```

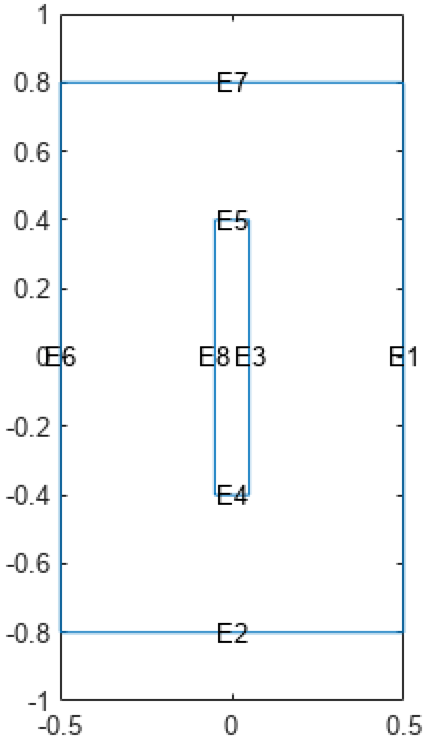
Import Geometry

Add the block geometry to the thermal model by using the `geometryFromEdges` function. The geometry description file for this problem is called `crackg.m`.

```
geometryFromEdges(thermalmodel,@crackg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
ylim([-1,1])
axis equal
```



Specify Thermal Properties of Material

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",1,...
                               "MassDensity",1,...
                               "SpecificHeat",1);
```

Apply Boundary Conditions

Specify the temperature on the left edge as `100`, and constant heat flow to the exterior through the right edge as `-10`. The toolbox uses the default insulating boundary condition for all other boundaries.

```
thermalBC(thermalmodel,"Edge",6,"Temperature",100);
thermalBC(thermalmodel,"Edge",1,"HeatFlux",-10);
```

Set Initial Conditions

Set an initial value of `0` for the temperature.

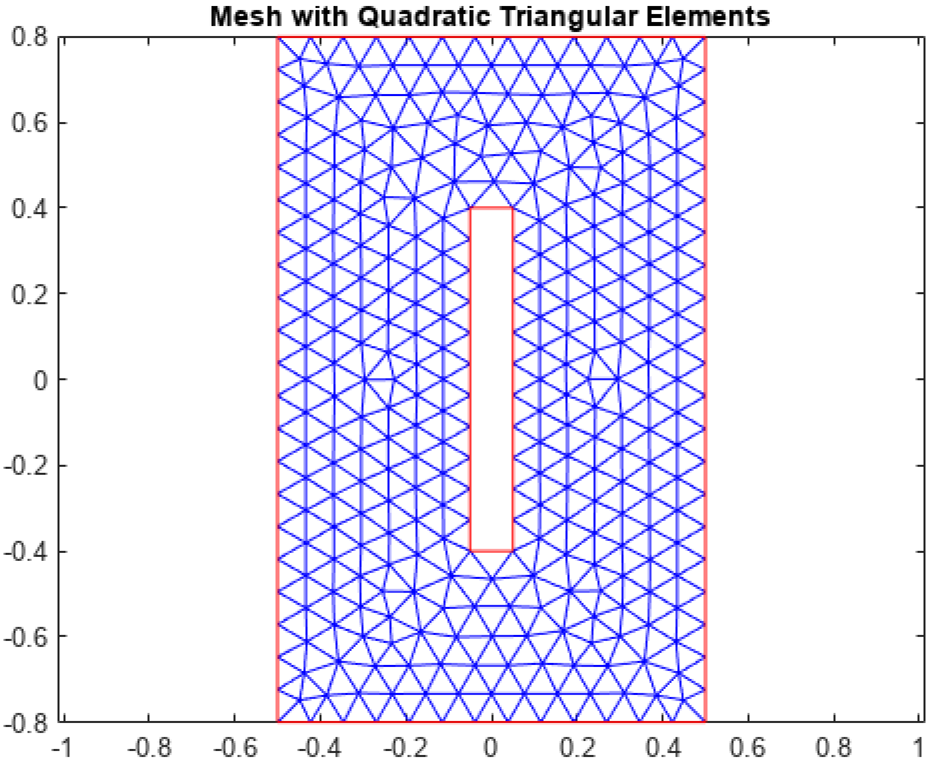
```
thermalIC(thermalmodel,0);
```

Generate Mesh

Create and plot a mesh.

```
generateMesh(thermalmodel);
figure
```

```
pdemesh(thermalmodel)
title("Mesh with Quadratic Triangular Elements")
```



Specify Solution Times

Set solution times to be 0 to 5 seconds in steps of 1/2.

```
tlist = 0:0.5:5;
```

Calculate Solution

Use the `solve` function to calculate the solution.

```
thermalresults = solve(thermalmodel,tlist)
```

```
thermalresults = 
  TransientThermalResults with properties:

      Temperature: [1320x11 double]
    SolutionTimes: [0 0.5000 1 1.5000 2 2.5000 3 3.5000 4 4.5000 5]
       XGradients: [1320x11 double]
       YGradients: [1320x11 double]
       ZGradients: []
             Mesh: [1x1 FEMesh]
```

Evaluate Heat Flux

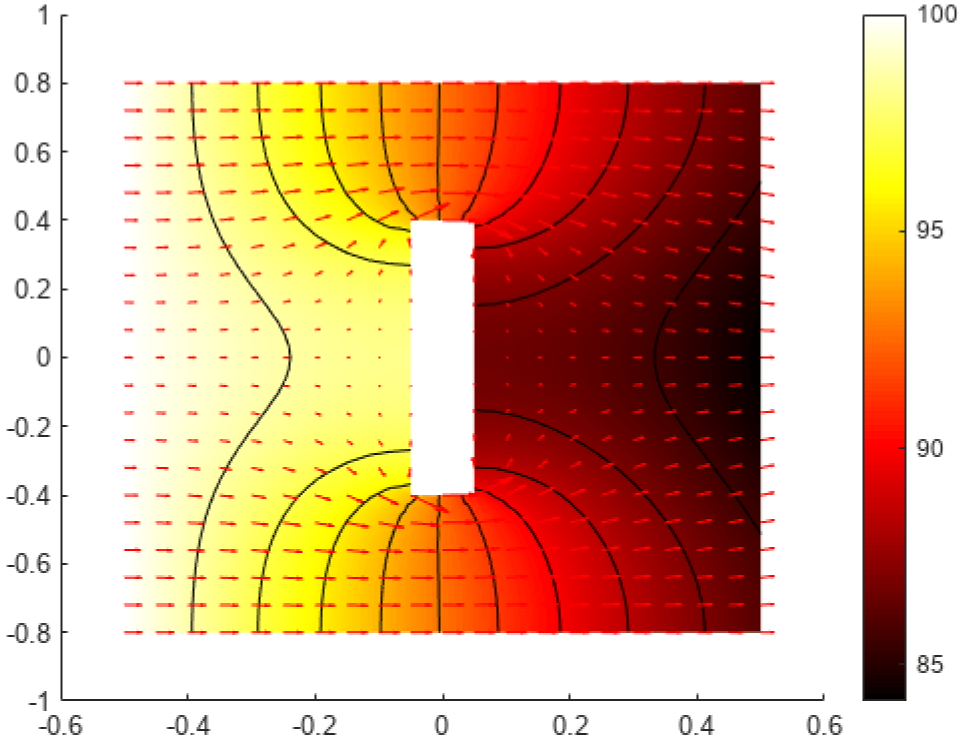
Compute the heat flux density.

```
[qx,qy] = evaluateHeatFlux(thermalresults);
```

Plot Temperature Distribution and Heat Flux

Plot the solution at the final time step, t = 5.0 seconds, with isothermal lines using a contour plot, and plot the heat flux vector field using arrows.

```
pdeplot(thermalmodel,"XYData",thermalresults.Temperature(:,end), ...
                     "Contour","on",...
                     "FlowData",[qx(:,end),qy(:,end)], ...
                     "ColorMap","hot")
```



Heat Transfer Problem with Temperature-Dependent Properties

This example shows how to solve the heat equation with a temperature-dependent thermal conductivity. The example shows an idealized thermal analysis of a rectangular block with a rectangular cavity in the center.

The partial differential equation for transient conduction heat transfer is:

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

where T is the temperature, ρ is the material density, C_p is the specific heat, and k is the thermal conductivity. f is the heat generated inside the body which is zero in this example.

Steady-State Solution: Constant Thermal Conductivity

Create a steady-state thermal model.

```
thermalmodelS = createpde("thermal","steadystate");
```

Create a 2-D geometry by drawing one rectangle the size of the block and a second rectangle the size of the slot.

```
r1 = [3 4 -.5 .5 .5 -.5  -.8 -.8 .8 .8];
r2 = [3 4 -.05 .05 .05 -.05  -.4 -.4 .4 .4];
gdm = [r1; r2]';
```

Subtract the second rectangle from the first to create the block with a slot.

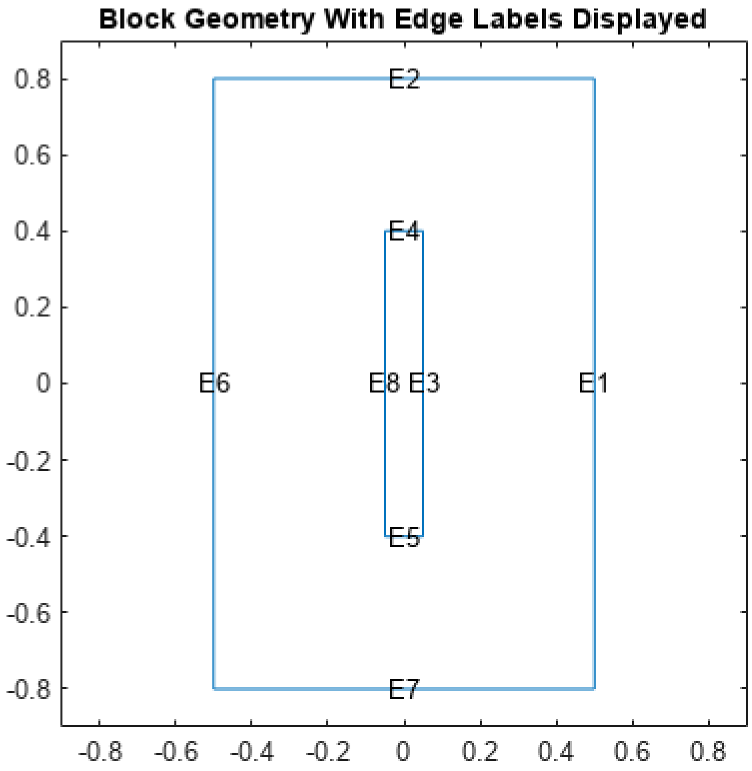
```
g = decsg(gdm,'R1-R2',['R1'; 'R2']');
```

Convert the `decsg` format into a geometry object. Include the geometry in the model.

```
geometryFromEdges(thermalmodelS,g);
```

Plot the geometry with edge labels displayed. The edge labels will be used below in the function for defining boundary conditions.

```
figure
pdegplot(thermalmodelS,"EdgeLabels","on");
axis([-.9 .9 -.9 .9]);
title("Block Geometry With Edge Labels Displayed")
```



Set the temperature on the left edge to 100 degrees. On the right edge, there is a prescribed heat flux out of the block. The top and bottom edges and the edges inside the cavity are all insulated, that is, no heat is transferred across these edges.

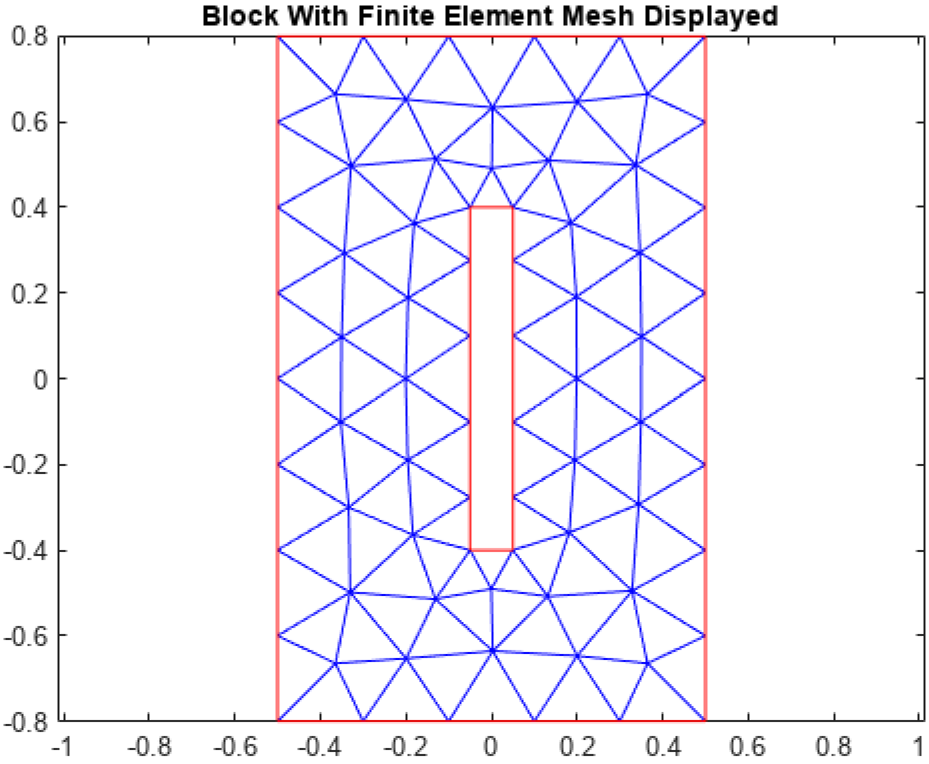
```
thermalBC(thermalmodelS,"Edge",6,"Temperature",100);
thermalBC(thermalmodelS,"Edge",1,"HeatFlux",-10);
```

Specify the thermal conductivity of the material. First, consider the constant thermal conductivity, for example, equal one. Later, consider a case where the thermal conductivity is a function of temperature.

```
thermalProperties(thermalmodelS,"ThermalConductivity",1);
```

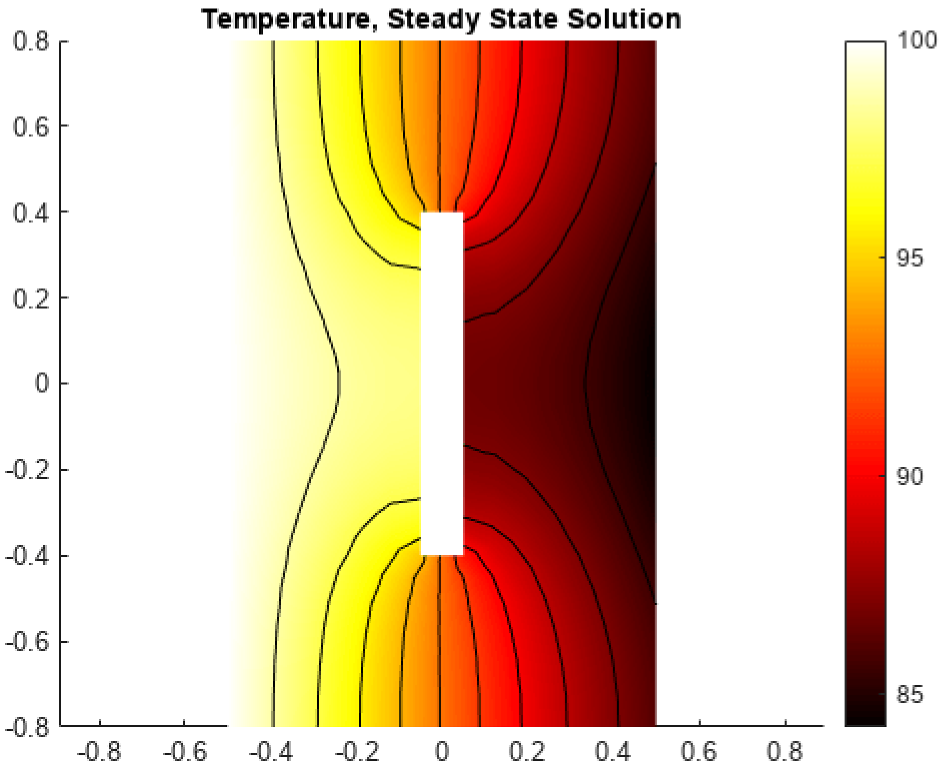
Create a mesh with elements no larger than 0.2.

```
generateMesh(thermalmodelS,"Hmax",0.2);
figure
pdeplot(thermalmodelS);
axis equal
title("Block With Finite Element Mesh Displayed")
```



Calculate the steady-state solution.

```
R = solve(thermalmodelS);
T = R.Temperature;
figure
pdeplot(thermalmodelS,"XYData",T,"Contour","on","ColorMap","hot");
axis equal
title("Temperature, Steady State Solution")
```



Transient Solution: Constant Thermal Conductivity

Create a transient thermal model and include the geometry.

```
thermalmodelT = createpde("thermal","transient");

r1 = [3 4 -.5 .5 .5 -.5  -.8 -.8 .8 .8];
r2 = [3 4 -.05 .05 .05 -.05  -.4 -.4 .4 .4];
gdm = [r1; r2]';
g = decsg(gdm,'R1-R2',['R1'; 'R2']');
geometryFromEdges(thermalmodelT,g);
```

Specify thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodelT,"ThermalConductivity",1,...
                                "MassDensity",1,...
                                "SpecificHeat",1);
```

Define boundary conditions. In the transient cases, the temperature on the left edge is zero at time=0 and ramps to 100 degrees over .5 seconds. You can find the helper function `transientBCHeatedBlock` under `matlab/R20XXx/examples/pde/main`.

```
thermalBC(thermalmodelT,"Edge",6,"Temperature",@transientBCHeatedBlock);
```

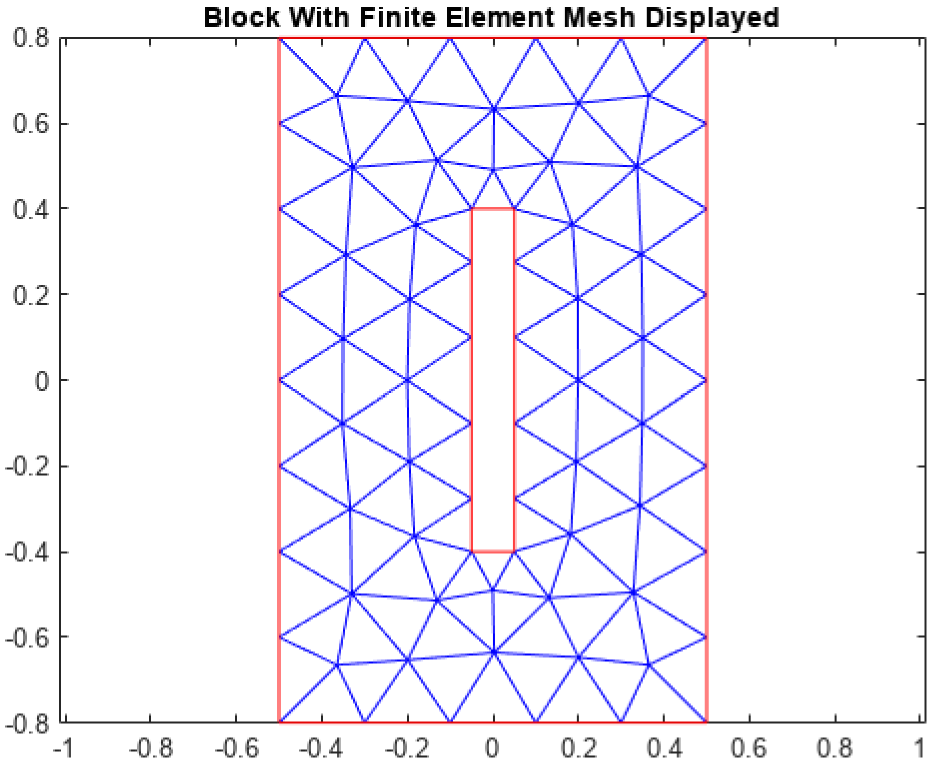
On the right edge, there is a prescribed heat flux out of the block.

```
thermalBC(thermalmodelT,"Edge",1,"HeatFlux",-10);
```

The top and bottom edges as well as the edges inside the cavity are all insulated, that is no heat is transferred across these edges.

Create a mesh with elements no larger than 0.2.

```
msh = generateMesh(thermalmodelT,"Hmax",0.2);
figure
pdeplot(thermalmodelT);
axis equal
title("Block With Finite Element Mesh Displayed")
```



Calculate the transient solution. Perform a transient analysis from zero to five seconds. The toolbox saves the solution every .1 seconds so that plots of the results as functions of time can be created.

```
tlist = 0:.1:5;
thermalIC(thermalmodelT,0);
R = solve(thermalmodelT,tlist);
T = R.Temperature;
```

Two plots are useful in understanding the results from this transient analysis. The first is a plot of the temperature at the final time. The second is a plot of the temperature at a specific point in the block, in this case near the center of the right edge, as a function of time. To identify a node near the center of the right edge, it is convenient to define this short utility function.

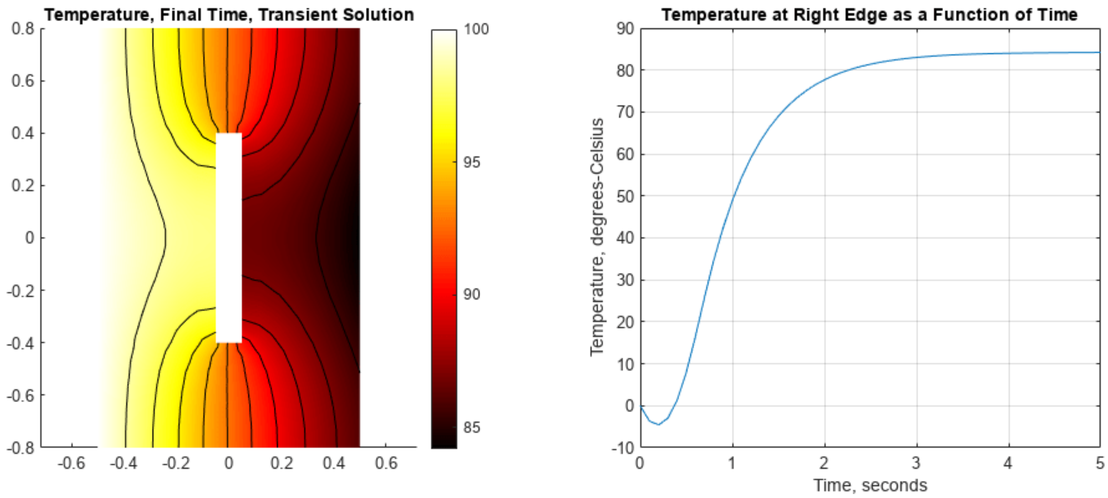
```
getClosestNode = @(p,x,y) min((p(1,:) - x).^2 + (p(2,:) - y).^2);
```

Call this function to get a node near the center of the right edge.

```
[~,nid] = getClosestNode( msh.Nodes, .5, 0 );
```

The two plots are shown side-by-side in the figure below. The temperature distribution at this time is very similar to that obtained from the steady-state solution above. At the right edge, for times less than about one-half second, the temperature is less than zero. This is because heat is leaving the block faster than it is arriving from the left edge. At times greater than about three seconds, the temperature has essentially reached steady-state.

```
h = figure;
h.Position = [1 1 2 1].*h.Position;
subplot(1,2,1);
axis equal
pdeplot(thermalmodelT,"XYData",T(:,end),"Contour","on", ...
                                        "ColorMap","hot");
axis equal
title("Temperature, Final Time, Transient Solution")
subplot(1,2,2);
axis equal
plot(tlist, T(nid,:));
grid on
title("Temperature at Right Edge as a Function of Time")
xlabel("Time, seconds")
ylabel("Temperature, degrees-Celsius")
```



Steady State Solution: Temperature-Dependent Thermal Conductivity

It is not uncommon for material properties to be functions of the dependent variables. For example, assume that the thermal conductivity is a simple linear function of temperature:

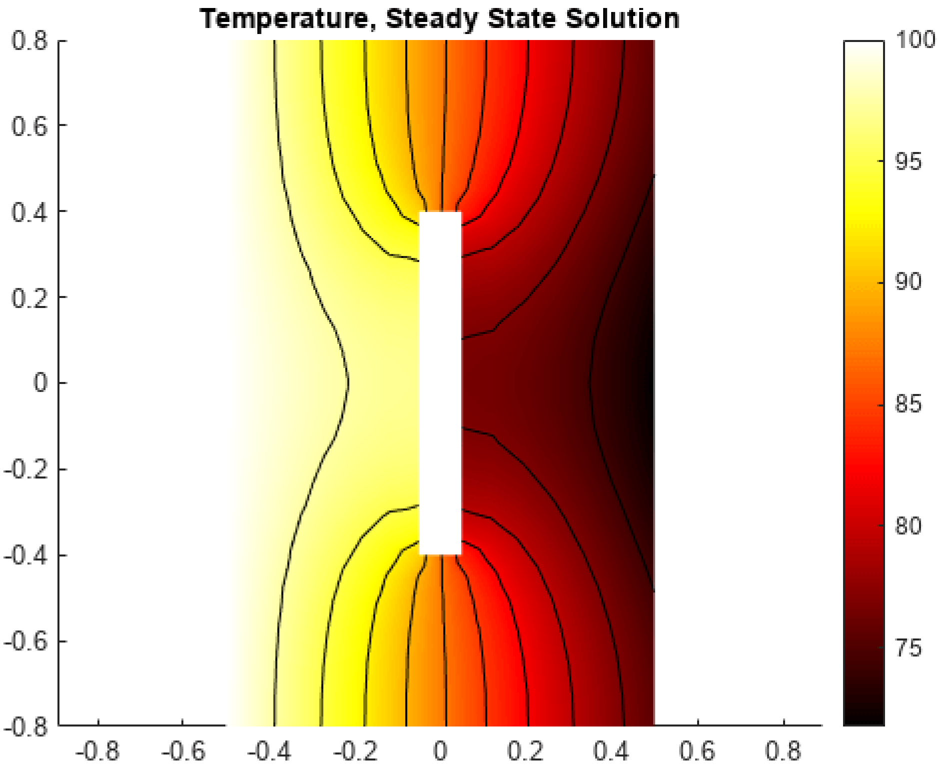
```
k = @(~,state) 0.3+0.003*state.u;
```

In this case, the variable u is the temperature. For this example, assume that the density and specific heat are not functions of temperature.

```
thermalProperties(thermalmodelS,"ThermalConductivity",k);
```

Calculate the steady-state solution. Compared with the constant-conductivity case, the temperature on the right-hand edge is lower. This is due to the lower conductivity in regions with lower temperature.

```
R = solve(thermalmodelS);
T = R.Temperature;
figure
pdeplot(thermalmodelS,"XYData",T,"Contour","on","ColorMap","hot");
axis equal
title("Temperature, Steady State Solution")
```



Transient Solution: Temperature-Dependent Thermal Conductivity

Now perform a transient analysis with the temperature-dependent conductivity.

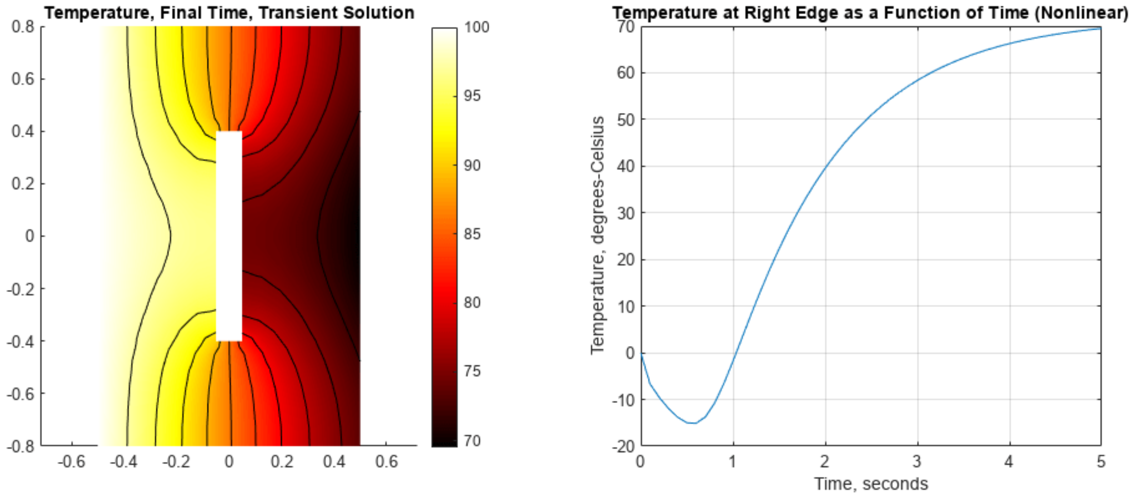
```
thermalProperties(thermalmodelT,"ThermalConductivity",k,...
                                "MassDensity",1,...
                                "SpecificHeat",1);
```

Use the same timespan `tlist = 0:.1:5` as for the linear case.

```
thermalIC(thermalmodelT,0);
R = solve(thermalmodelT,tlist);
T = R.Temperature;
```

Plot the temperature at the final time step and the temperature at the right edge as a function of time. The plot of temperature at the final time step is only slightly different from the comparable plot from the linear analysis: temperature at the right edge is slightly lower than the linear case. The plot of temperature as a function of time is considerably different from the linear case. Because of the lower conductivity at lower temperatures, the heat takes longer to reach the right edge of the block. In the linear case, the temperature is essentially constant at around three seconds but for this nonlinear case, the temperature curve is just beginning to flatten at five seconds.

```
h = figure;
h.Position = [1 1 2 1].*h.Position;
subplot(1,2,1);
axis equal
pdeplot(thermalmodelT,"XYData",T(:,end),"Contour","on", ...
                                        "ColorMap","hot");
axis equal
title("Temperature, Final Time, Transient Solution")
subplot(1,2,2);
axis equal
plot(tlist(1:size(T,2)), T(nid,:));
grid on
title("Temperature at Right Edge as a Function of Time (Nonlinear)")
xlabel("Time, seconds")
ylabel("Temperature, degrees-Celsius")
```



Heat Conduction in Multidomain Geometry with Nonuniform Heat Flux

This example shows how to perform a 3-D transient heat conduction analysis of a hollow sphere made of three different layers of material.

The sphere is subject to a nonuniform external heat flux.

The physical properties and geometry of this problem are described in Singh, Jain, and Rizwan-uddin (see Reference), which also has an analytical solution for this problem. The inner face of the sphere has a temperature of zero at all times. The outer hemisphere with positive y value has a nonuniform heat flux defined by

$$q_{\text{outer}} = \theta^2(\pi - \theta)^2\phi^2(\pi - \phi)^2$$

$$0 \le \theta \le \pi, 0 \le \phi \le \pi\,.$$

θ and ϕ are azimuthal and elevation angles of points in the sphere. Initially, the temperature at all points in the sphere is zero.

Create a thermal model for transient thermal analysis.

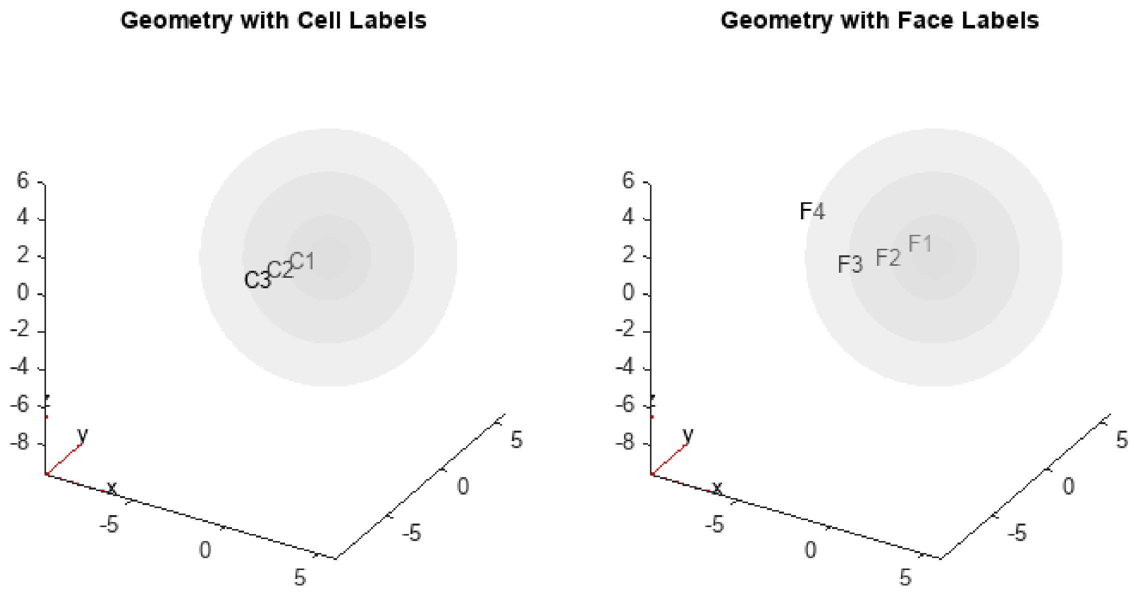
```
thermalmodel = createpde("thermal","transient");
```

Create a multilayered sphere using the `multisphere` function. Assign the resulting geometry to the thermal model. The sphere has three layers of material with a hollow inner core.

```
gm = multisphere([1,2,4,6],"Void",[true,false,false,false]);
thermalmodel.Geometry = gm;
```

Plot the geometry and show the cell labels and face labels. Use a `FaceAlpha` of 0.25 so that labels of the interior layers are visible.

```
figure("Position",[10,10,800,400]);
subplot(1,2,1)
pdegplot(thermalmodel,"FaceAlpha",0.25,"CellLabel","on")
title("Geometry with Cell Labels")
subplot(1,2,2)
pdegplot(thermalmodel,"FaceAlpha",0.25,"FaceLabel","on")
title("Geometry with Face Labels")
```



Generate a mesh for the geometry. Choose a mesh size that is coarse enough to speed the solution, but fine enough to represent the geometry reasonably accurately.

```
generateMesh(thermalmodel,"Hmax",1);
```

Specify the thermal conductivity, mass density, and specific heat for each layer of the sphere. The material properties are dimensionless values, not given by realistic material properties.

```
thermalProperties(thermalmodel,"Cell",1,"ThermalConductivity",1, ...
                                        "MassDensity",1, ...
                                        "SpecificHeat",1);
thermalProperties(thermalmodel,"Cell",2,"ThermalConductivity",2, ...
                                        "MassDensity",1, ...
                                        "SpecificHeat",0.5);
thermalProperties(thermalmodel,"Cell",3,"ThermalConductivity",4, ...
                                        "MassDensity",1, ...
                                        "SpecificHeat",4/9);
```

Specify boundary conditions. The innermost face has a temperature of zero at all times.

```
thermalBC(thermalmodel,"Face",1,"Temperature",0);
```

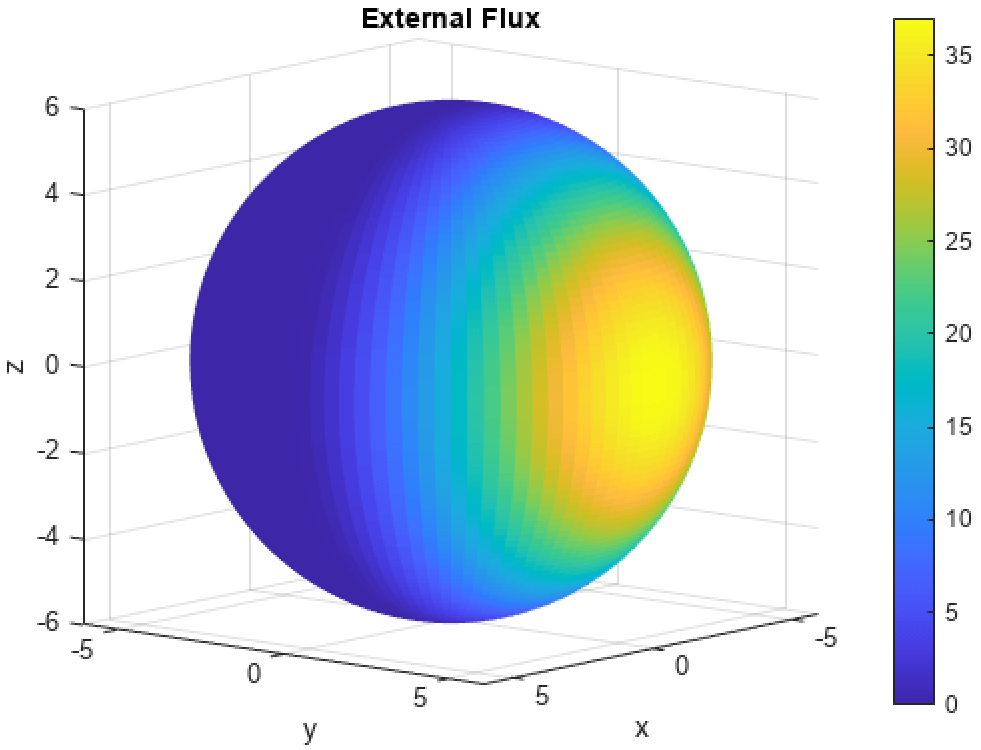
The outer surface of the sphere has an external heat flux. Use the functional form of thermal boundary conditions to define the heat flux.

```
function Qflux = externalHeatFlux(region,~)

[phi,theta,~] = cart2sph(region.x,region.y,region.z);

theta = pi/2 - theta; % transform to 0 <= theta <= pi

ids = phi > 0;
```

```
Qflux = zeros(size(region.x));

Qflux(ids) = theta(ids).^2.*(pi - theta(ids)).^2.*phi(ids).^2.*(pi -
phi(ids)).^2;

end
```

Plot the flux on the surface.

```
[phi,theta,r] = meshgrid(linspace(0,2*pi),linspace(-pi/2,pi/2),6);
[x,y,z] = sph2cart(phi,theta,r);
region.x = x;
region.y = y;
region.z = z;
flux = externalHeatFlux(region,[]);
figure
surf(x,y,z,flux,"LineStyle","none")
axis equal
view(130,10)
colorbar
xlabel("x")
ylabel("y")
zlabel("z")
title("External Flux")
```



Include this boundary condition in the model.

```
thermalBC(thermalmodel,"Face",4, ...
                       "HeatFlux",@externalHeatFlux, ...
                       "Vectorized","on");
```

Define the initial temperature to be zero at all points.

```
thermalIC(thermalmodel,0);
```

Define a time-step vector and solve the transient thermal problem.

```
tlist = [0,2,5:5:50];
R = solve(thermalmodel,tlist);
```

To plot contours at several times, with the contour levels being the same for all plots, determine the range of temperatures in the solution. The minimum temperature is zero because it is the boundary condition on the inner face of the sphere.

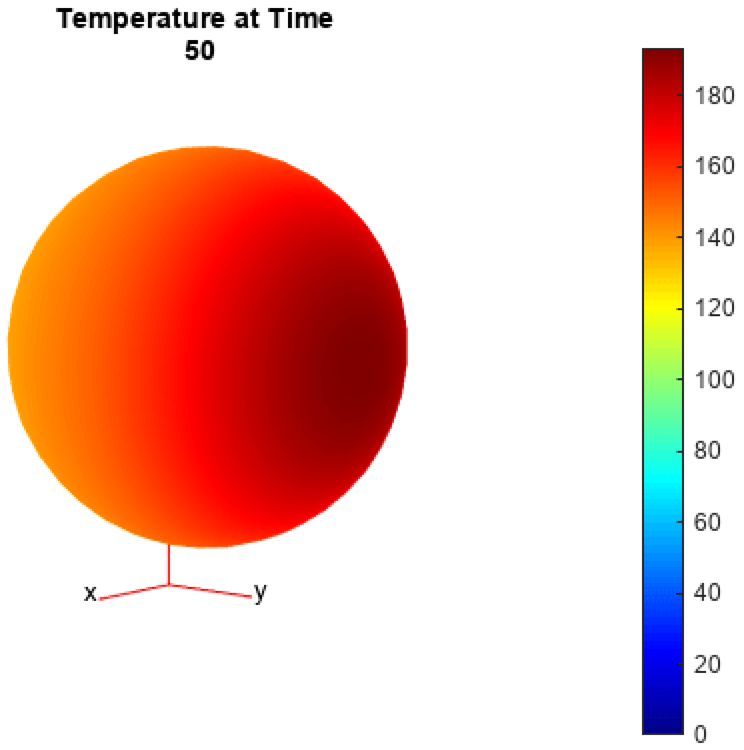
```
Tmin = 0;
```

Find the maximum temperature from the final time-step solution.

```
Tmax = max(R.Temperature(:,end));
```

Plot contours in the range `Tmin` to `Tmax` at the times in `tlist`.

```
h = figure;
for i = 1:numel(tlist)
    pdeplot3D(thermalmodel,"ColorMapData",R.Temperature(:,i))
    caxis([Tmin,Tmax])
    view(130,10)
    title(["Temperature at Time " num2str(tlist(i))]);
    M(i) = getframe;

end
```



To see a movie of the contours when running this example on your computer, execute the following line:

```
movie(M,2)
```

Visualize the temperature contours on the cross-section. First, define a rectangular grid of points on the $y - z$ plane where $x = 0$.

```
[YG,ZG] = meshgrid(linspace(-6,6,100),linspace(-6,6,100));
XG = zeros(size(YG));
```

Interpolate the temperature at the grid points. Perform interpolation for several time steps to observe the evolution of the temperature contours.

```
tIndex = [2,3,5,7,9,11];
varNames = {'Time_index','Time_step'};
index_step = table(tIndex.',tlist(tIndex).','VariableNames',varNames);
disp(index_step);
```

```
    Time_index    Time_step
    __________    _________

         2             2   
         3             5   
         5            15   
         7            25   
         9            35   
        11            45   
```

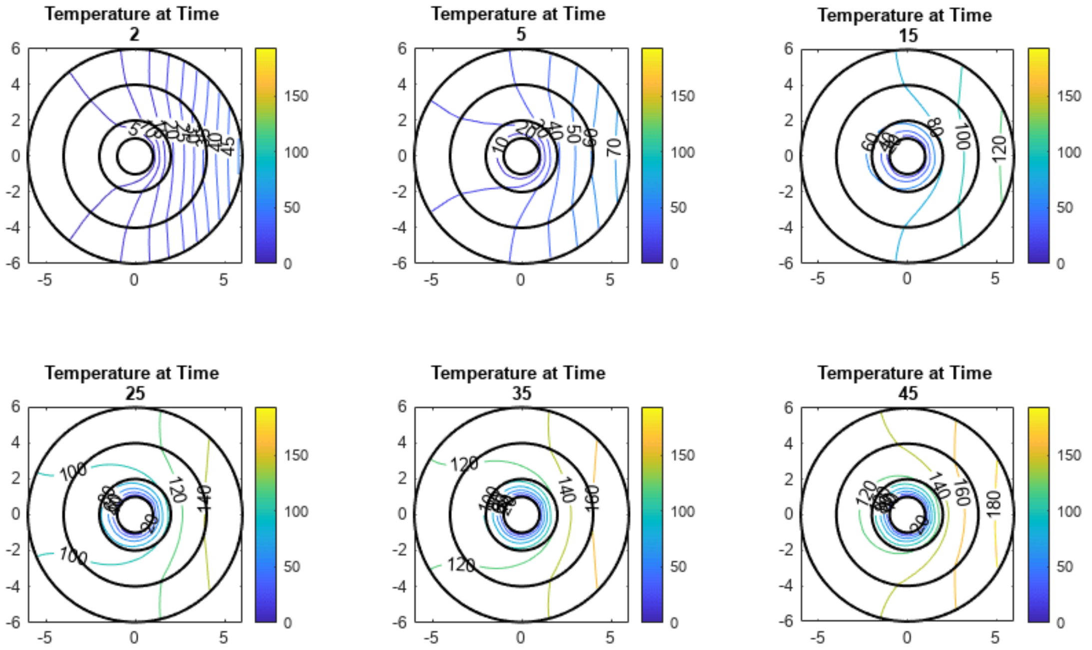
```
TG = interpolateTemperature(R,XG,YG,ZG,tIndex);
```

Define the geometric spherical layers on the cross-section.

```
t = linspace(0,2*pi);
ylayer1 = cos(t); zlayer1 = sin(t);
ylayer2 = 2*cos(t); zlayer2 = 2*sin(t);
ylayer3 = 4*cos(t); zlayer3 = 4*sin(t);
ylayer4 = 6*cos(t); zlayer4 = 6*sin(t);
```

Plot the contours in the range `Tmin` to `Tmax` for the time steps corresponding to the time indices `tIndex`.

```
figure("Position",[10,10,1000,550]);
for i = 1:numel(tIndex)
    subplot(2,3,i)
    contour(YG,ZG,reshape(TG(:,i),size(YG)),"ShowText","on")
    colorbar
    title(["Temperature at Time " num2str(tlist(tIndex(i)))]);
    hold on
    caxis([Tmin,Tmax])
    axis equal
    % Plot boundaries of spherical layers for reference.
    plot(ylayer1,zlayer1,"k","LineWidth",1.5)
    plot(ylayer2,zlayer2,"k","LineWidth",1.5)
    plot(ylayer3,zlayer3,"k","LineWidth",1.5)
    plot(ylayer4,zlayer4,"k","LineWidth",1.5)
end
```



Reference

[1] Singh, Suneet, P. K. Jain, and Rizwan-uddin. "Analytical Solution for Three-Dimensional, Unsteady Heat Conduction in a Multilayer Sphere." ASME. J. Heat Transfer. 138(10), 2016, pp. 101301-101301-11.

Inhomogeneous Heat Equation on Square Domain

This example shows how to solve the heat equation with a source term.

The basic heat equation with a unit source term is

$$\frac{\partial u}{\partial t} - \Delta u = 1$$

This equation is solved on a square domain with a discontinuous initial condition and zero temperatures on the boundaries.

Create a transient thermal model.

```
thermalmodel = createpde("thermal","transient");
```

Create a square geometry centered at `x = 0` and `y = 0` with sides of length 2. Include a circle of radius 0.4 concentric with the square.

```
R1 = [3;4;-1;1;1;-1;-1;-1;1;1];
C1 = [1;0;0;0.4];
C1 = [C1;zeros(length(R1) - length(C1),1)];
gd = [R1,C1];
sf = 'R1+C1';
ns = char('R1','C1')';
g = decsg(gd,sf,ns);
```

Append the geometry to the model.

```
geometryFromEdges(thermalmodel,g);
```

Specify thermal properties of the material.

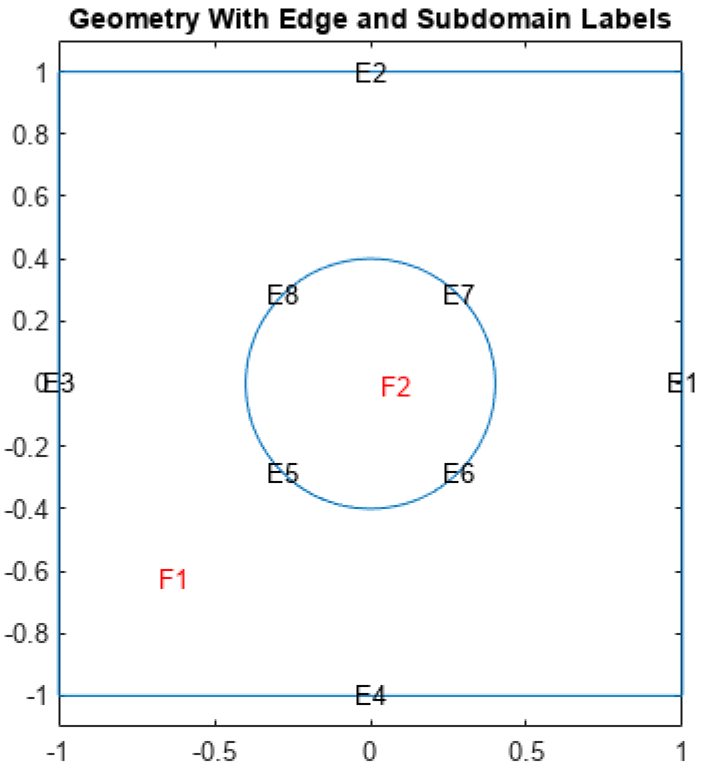
```
thermalProperties(thermalmodel,"ThermalConductivity",1,...
                               "MassDensity",1,...
                               "SpecificHeat",1);
```

Specify internal heat source.

```
internalHeatSource(thermalmodel,1);
```

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

```
figure
pdegplot(thermalmodel,"EdgeLabels","on","FaceLabels","on")
axis([-1.1 1.1 -1.1 1.1]);
axis equal
title("Geometry With Edge and Subdomain Labels")
```



Set zero temperatures on all four outer edges of the square.

```
thermalBC(thermalmodel,"Edge",1:4,"Temperature",0);
```

The discontinuous initial value is 1 inside the circle and zero outside. Specify zero initial temperature everywhere.

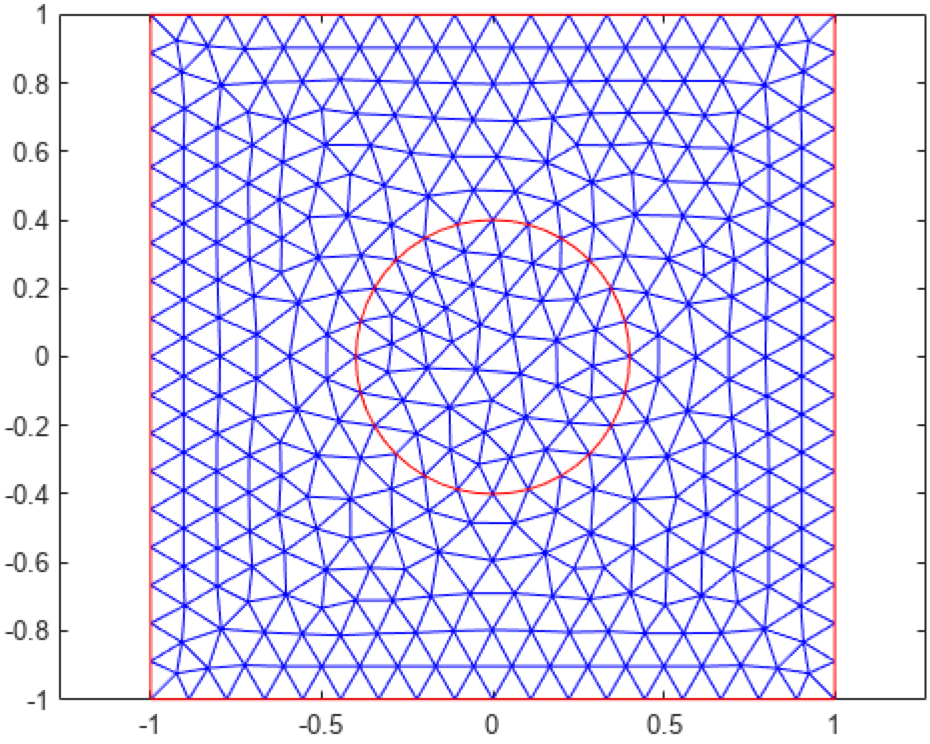
```
thermalIC(thermalmodel,0);
```

Specify non-zero initial temperature inside the circle (Face 2).

```
thermalIC(thermalmodel,1,"Face",2);
```

Generate and plot a mesh.

```
msh = generateMesh(thermalmodel);
figure;
pdemesh(thermalmodel);
axis equal
```



Find the solution at 20 points in time between 0 and 0.1.

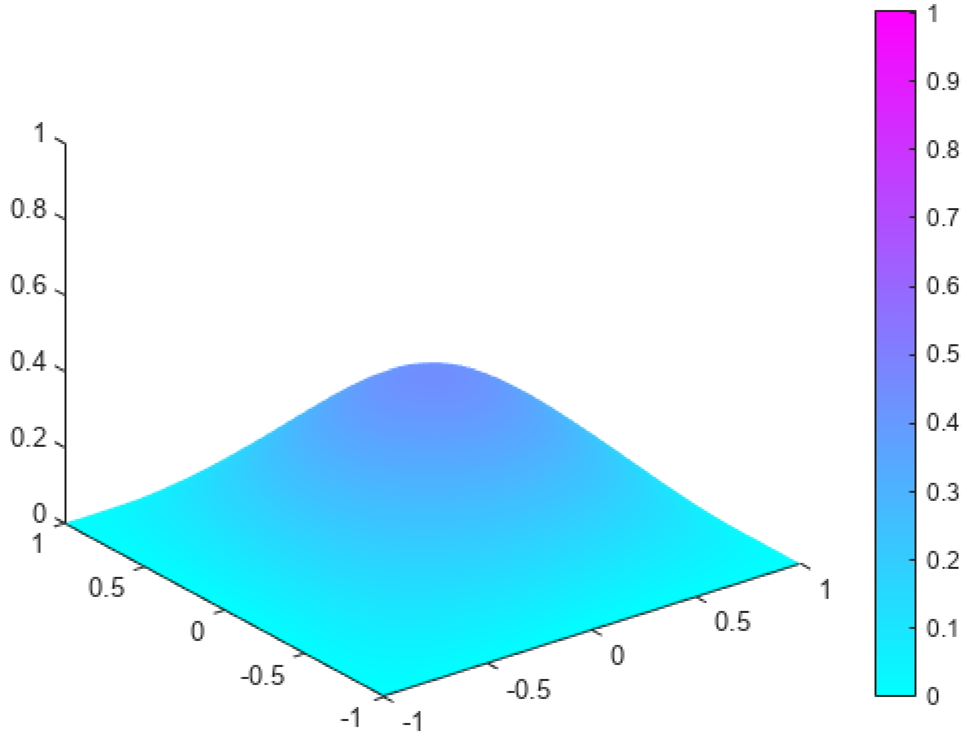
```
nframes = 20;
tlist = linspace(0,0.1,nframes);

thermalmodel.SolverOptions.ReportStatistics ='on';
result = solve(thermalmodel,tlist);
```

```
99 successful steps
0 failed attempts
200 function evaluations
1 partial derivatives
20 LU decompositions
199 solutions of linear systems
```

```
T = result.Temperature;
```

Plot the solution.

```
figure
Tmax = max(max(T));
Tmin = min(min(T));
for j = 1:nframes
    pdeplot(thermalmodel,"XYData",T(:,j),"ZData",T(:,j));
    caxis([Tmin Tmax]);
    axis([-1 1 -1 1 0 1]);
    Mv(j) = getframe;
end
```



To play the animation, use the `movie(Mv,1)` command.

Heat Distribution in Circular Cylindrical Rod

This example shows how to simplify a 3-D axisymmetric thermal problem to a 2-D problem using the symmetry around the axis of rotation of the body.

This example analyzes heat transfer in a rod with a circular cross section. There is a heat source at the bottom of the rod and a fixed temperature at the top. The outer surface of the rod exchanges heat with the environment because of convection. In addition, the rod itself generates heat because of radioactive decay. The goal is to find the temperature in the rod as a function of time.

The model geometry, material properties, and boundary conditions must all be symmetric about the axis of rotation. The toolbox assumes that the axis of rotation is the vertical axis passing through r = 0.

Steady-State Solution

First, compute the steady-state solution. If the final time in the transient analysis is sufficiently large, the transient solution at the final time must be close to the steady state solution. By comparing these two results, you can check the accuracy of the transient analysis.

Create a steady-state thermal model for solving an axisymmetric problem.

```
thermalmodel = createpde("thermal","steadystate-axisymmetric");
```

The 2-D model is a rectangular strip whose *x*-dimension extends from the axis of symmetry to the outer surface and *y*-dimension extends over the actual length of the rod (from -1.5 m to 1.5 m). Create the geometry by specifying the coordinates of its four corners.

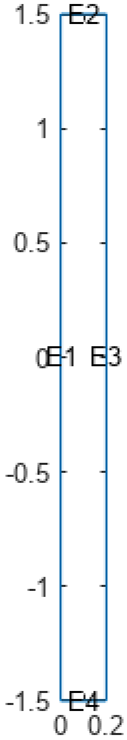
```
g = decsg([3 4 0 0 .2 .2 -1.5 1.5 1.5 -1.5]');
```

Include the geometry in the model.

```
geometryFromEdges(thermalmodel,g);
```

Plot the geometry with the edge labels.

```
figure
pdegplot(thermalmodel,"EdgeLabels","on")
axis equal
```



The rod is composed of a material with these thermal properties.

```
k = 40; % Thermal conductivity, W/(m*C)
rho = 7800; % Density, kg/m^3
cp = 500; % Specific heat, W*s/(kg*C)
q = 20000; % Heat source, W/m^3
```

For a steady-state analysis, specify the thermal conductivity of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",k);
```

Specify the internal heat source.

```
internalHeatSource(thermalmodel,q);
```

Define the boundary conditions. There is no heat transferred in the direction normal to the axis of symmetry (edge 1). You do not need to change the default boundary condition for this edge. Edge 2 is kept at a constant temperature *T* = 100 °C.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Specify the convection boundary condition on the outer boundary (edge 3). The surrounding temperature at the outer boundary is 100 °C, and the heat transfer coefficient is 50 $\mathrm{W/(m \cdot {}^{\circ}C)}$.

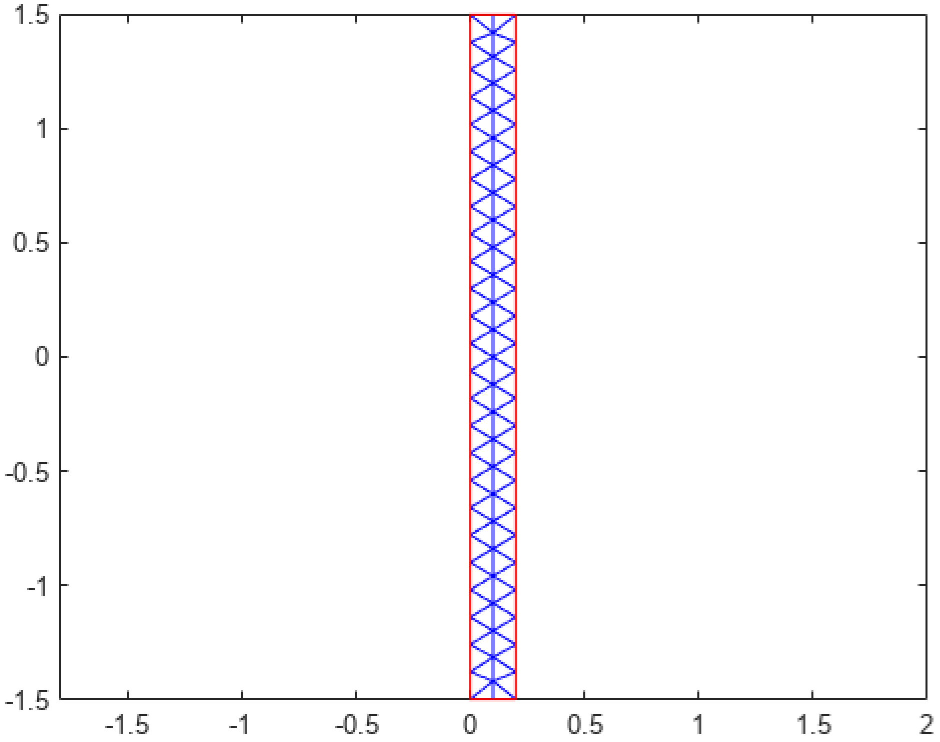
```
thermalBC(thermalmodel,"Edge",3,...
                       "ConvectionCoefficient",50,...
                       "AmbientTemperature",100);
```

The heat flux at the bottom of the rod (edge 4) is 5000 W/m^2.

```
thermalBC(thermalmodel,"Edge",4,"HeatFlux",5000);
```

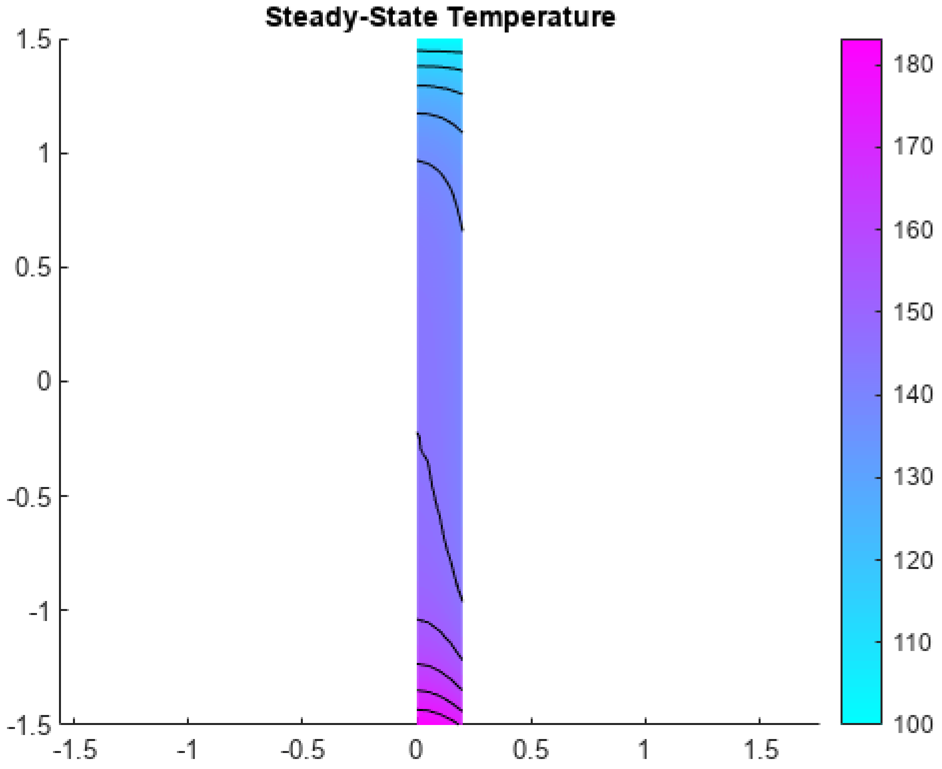
Generate the mesh.

```
msh = generateMesh(thermalmodel);
figure
pdeplot(thermalmodel)
axis equal
```



Solve the model and plot the result.

```
result = solve(thermalmodel);
T = result.Temperature;
figure
pdeplot(thermalmodel,"XYData",T,"Contour","on")
axis equal
title("Steady-State Temperature")
```



Transient Solution

Switch the analysis type of the model to transient-axisymmetric.

```
thermalmodel.AnalysisType = "transient-axisymmetric";
```

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",k,...
                               "MassDensity",rho,...
                               "SpecificHeat",cp);
```

Specify that the Initial temperature in the rod is 0 °C.

```
thermalIC(thermalmodel,0);
```

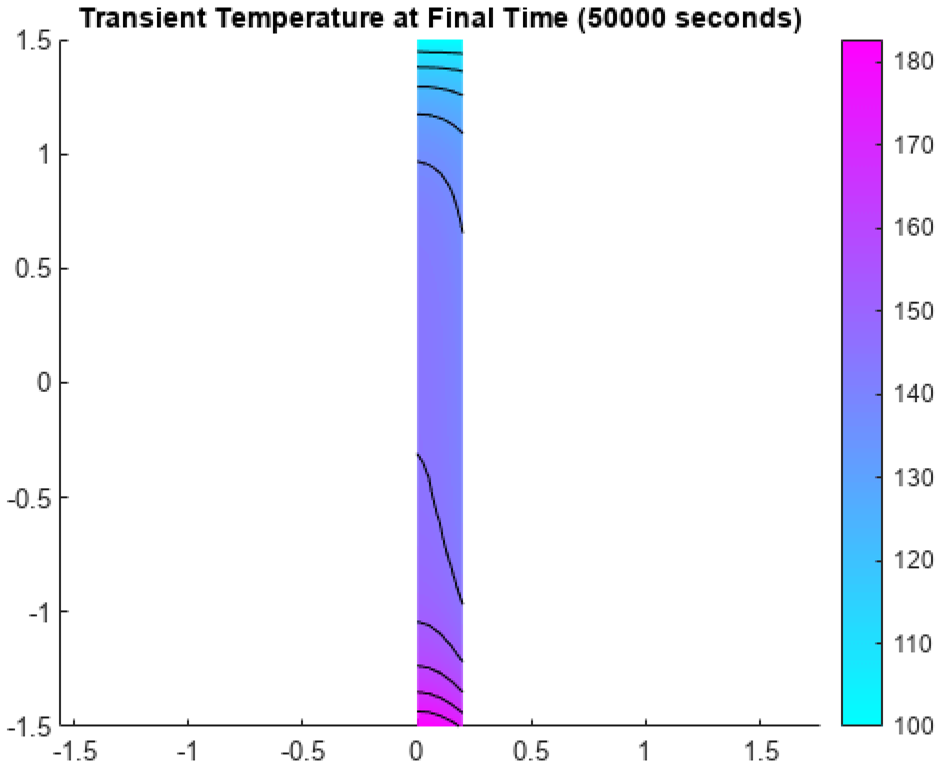
Compute the transient solution for solution times from t = 0 to t = 50000 seconds.

```
tfinal = 50000;
tlist = 0:100:tfinal;
result = solve(thermalmodel,tlist);
```

Plot the temperature distribution at t = 50000 seconds.

```
T = result.Temperature;

figure
pdeplot(thermalmodel,"XYData",T(:,end),"Contour","on")
```

```
axis equal
title(sprintf(['Transient Temperature' ...
               ' at Final Time (%g seconds)'],tfinal))
```

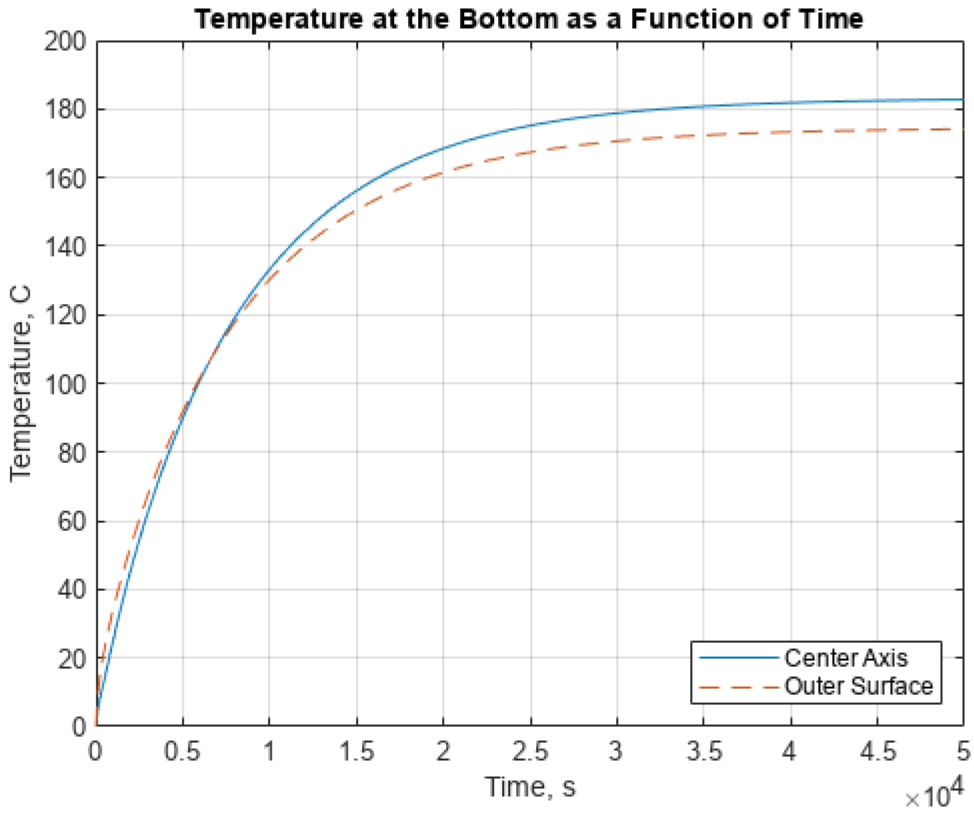


Find the temperature at the bottom surface of the rod: first, at the center axis and then on the outer surface.

```
Tcenter = interpolateTemperature(result,[0.0;-1.5],1:numel(tlist));
Touter = interpolateTemperature(result,[0.2;-1.5],1:numel(tlist));
```

Plot 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 °C. When the surface temperature of the rod is less than 100 °C, the environment heats the rod. The outer surface is slightly warmer than the inner axis. When the surface temperature is greater than 100 °C, the environment cools the rod. The outer surface becomes cooler than the interior of the rod.

```
figure
plot(tlist,Tcenter)
hold on
plot(tlist,Touter,"--")
title("Temperature at the Bottom as a Function of Time")
xlabel("Time, s")
ylabel("Temperature, C")
grid on
legend("Center Axis","Outer Surface","Location","SouthEast")
```


Temperature at the Bottom as a Function of Time
Temperature, C
Time, s
×10^4
0
20
40
60
80
100
120
140
160
180
200
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Center Axis
Outer Surface

Thermal Analysis of Disc Brake

This example analyses the temperature distribution of a disc brake. Disc brakes absorb the translational mechanical energy through friction and transform it into the thermal energy, which then dissipates. The transient thermal analysis of a disc brake is critical because the friction and braking performance decreases at high temperatures. Therefore, disc brakes must not exceed a given temperature limit during operation.

This example simulates the disc behavior in two steps:

- Perform a highly detailed simulation of the brake pad moving around the disc. Because the computational cost is high, this part of the example only simulates one half revolution (25 ms).
- Simulate full braking when the car goes from 100 km/h to 0 km/h in 2.75 seconds, and then remains stopped for 2.25 more seconds in order to allow the heat in the disc to dissipate.

The example uses a vehicle model in Simscape™ Driveline™ to obtain the time dependency of the dissipated power.

Point Heat Source Moving Around the Disc

Simulate a circular brake pad moving around the disc. This detailed simulation over a short timescale models the heat source as a point moving around the disc.

First, create a thermal transient model.

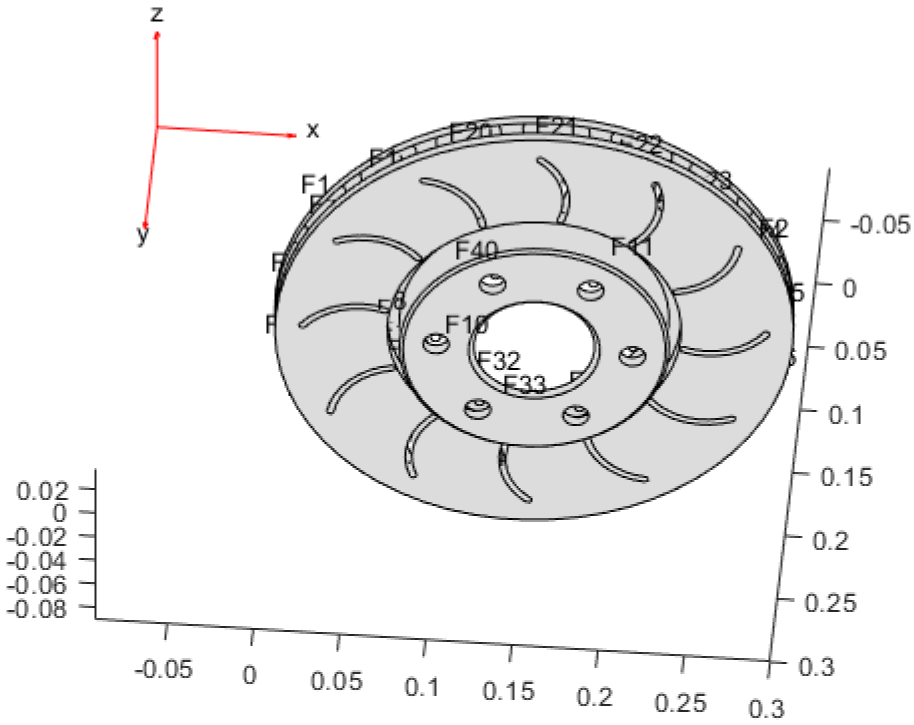
```
model = createpde("thermal","transient");
```

Import the disc geometry.

```
importGeometry(model,"brake_disc.stl");
```

Plot the geometry with the face labels.

```
figure
pdegplot(model,"FaceLabels","on");
view([-5 -47])
```

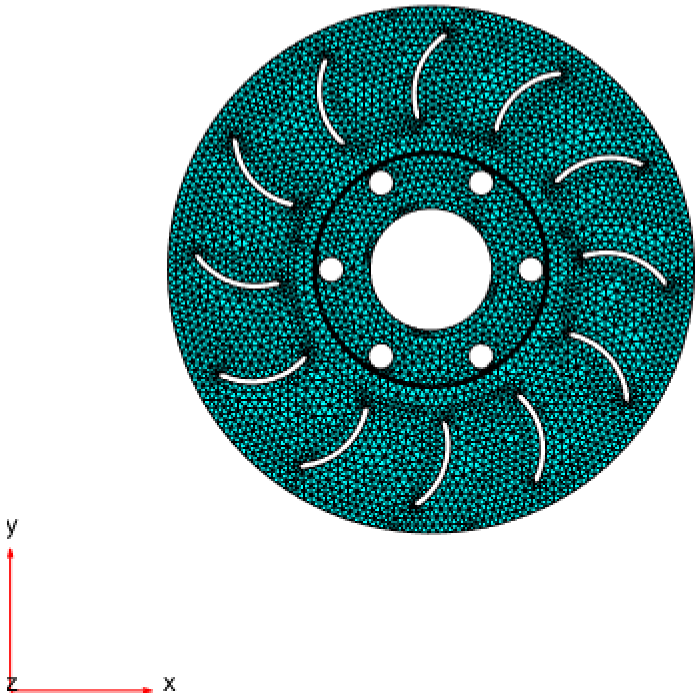


Generate a fine mesh with a small target maximum element edge length. The resulting mesh has more than 130000 nodes (degrees of freedom).

```
generateMesh(model,"Hmax",0.005);
```

Plot the mesh.

```
figure
pdemesh(model)
view([0,90])
```



Specify the thermal properties of the material.

```
thermalProperties(model,"ThermalConductivity",100, ...
                        "MassDensity",8000, ...
                        "SpecificHeat",500);
```

Specify the boundary conditions. All the faces are exposed to air, so there will be free convection.

```
thermalBC(model,"Face",1:model.Geometry.NumFaces, ...
                "ConvectionCoefficient",10, ...
                "AmbientTemperature",30);
```

Model the moving heat source by using a function handle that defines the thermal load as a function of space and time. For the definition of the `movingHeatSource` function, see the Heat Source Functions section at the bottom of this page.

```
thermalBC(model,"Face",11,"HeatFlux",@movingHeatSource);
thermalBC(model,"Face",4,"HeatFlux",@movingHeatSource);
```

Specify the initial temperature.

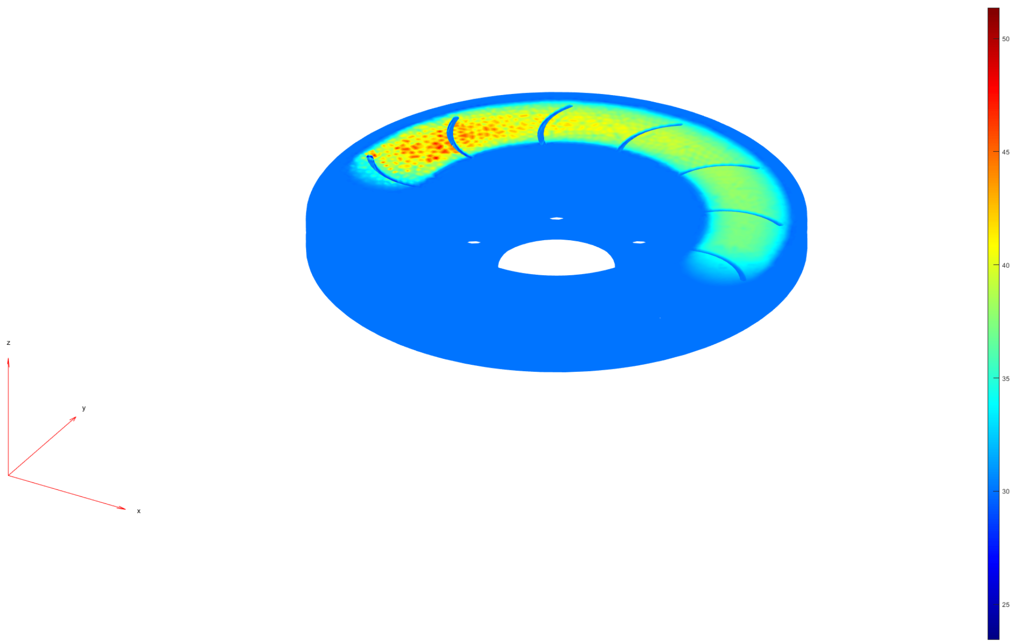
```
thermalIC(model,30);
```

Solve the model for the time steps from 0 to 25 ms.

```
tlist = linspace(0,0.025,100); % Half revolution
R1 = solve(model,tlist);
```

Plot the temperature distribution at 25 ms.

```
figure("units","normalized","outerposition",[0 0 1 1])
pdeplot3D(model,"ColorMapData",R1.Temperature(:,end))
```



The `animation` function visualizes the solution for all time steps. To play the animation, use this command:

```
animation(model,R1)
```

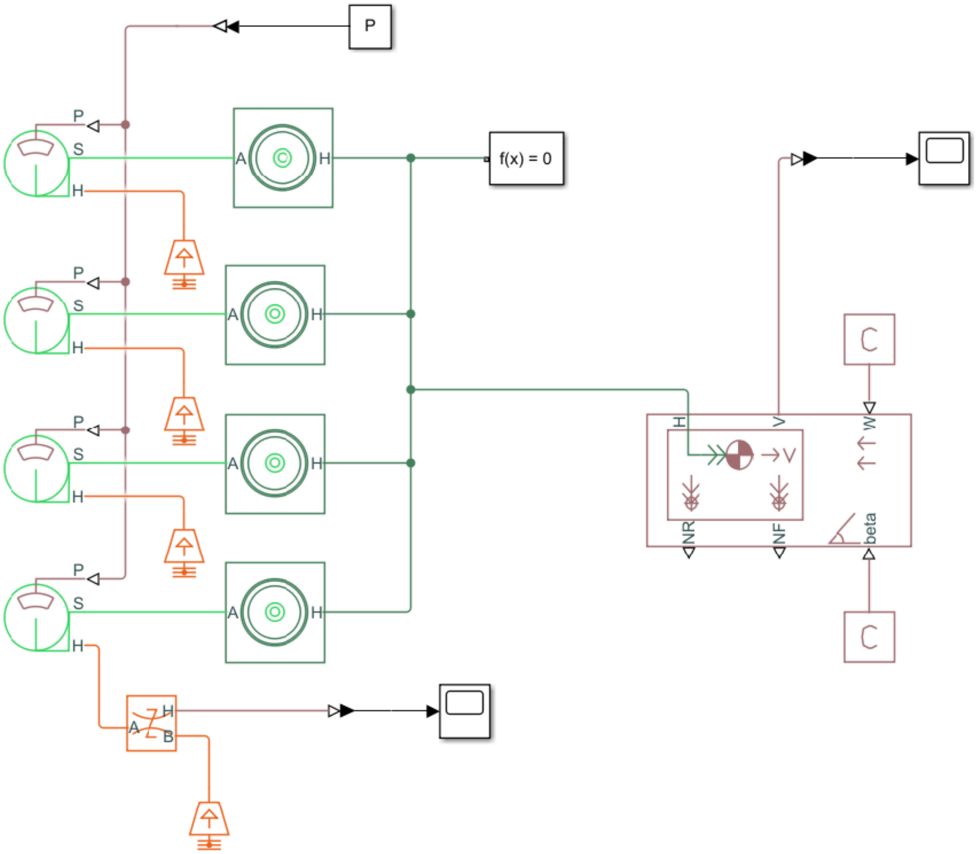
Because the heat diffusion time is much longer than the period of a revolution, you can simplify the heat source for the long simulation.

Static Ring Heat Source

Now find the disc temperatures for a longer period of time. Because the heat does not have time to diffuse during a revolution, it is reasonable to approximate the heat source with a static heat source in the shape of the path of the brake pad.

Compute the heat flow applied to the disc as a function of time. To do this, use a Simscape Driveline™ model of a four-wheeled, 2000 kg vehicle that brakes from 100 km/h to 0 km/h in approximately 2.75 s.

```
driveline_model = "DrivelineVehicle_isothermal";
open_system(driveline_model);
```



```
M = 2000; % kg
V0 = 27.8; % m/s, which is around 100 km/h
P = 277; % bar

simOut = sim(driveline_model);

heatFlow = simOut.simlog.Heat_Flow_Rate_Sensor.Q.series.values;
tout = simOut.tout;
```

Obtain the time-dependent heat flow by using the results from the Simscape Driveline model.

```
drvln = struct();
drvln.tout = tout;
drvln.heatFlow = heatFlow;
```

Generate a mesh.

```
generateMesh(model);
```

Specify the boundary condition as a function handle. For the definition of the `ringHeatSource` function, see the Heat Source Functions section at the bottom of this page.

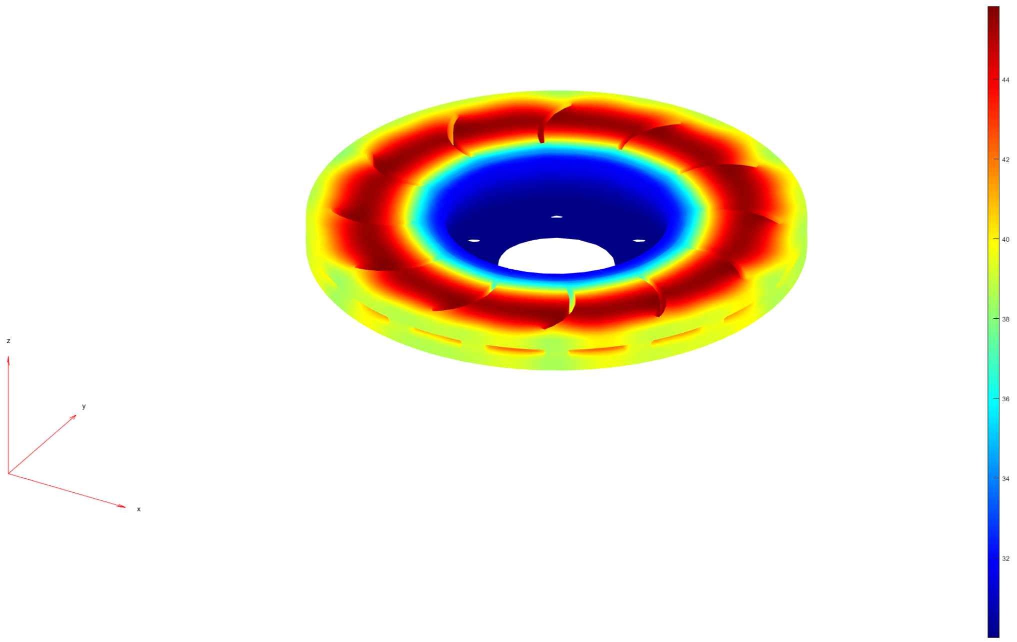
```
thermalBC(model,"Face",11, ...
                "HeatFlux",@(r,s)ringHeatSource(r,s,drvln));
thermalBC(model,"Face",4, ...
                "HeatFlux",@(r,s)ringHeatSource(r,s,drvln));
```

Solve the model for times from 0 to 5 seconds.

```
tlist = linspace(0,5,250);
R2 = solve(model,tlist);
```

Plot the temperature distribution at the final time step *t* = 5 seconds.

```
figure("units","normalized","outerposition",[0 0 1 1])
pdeplot3D(model,"ColorMapData",R2.Temperature(:,end))
```



The `animation` function visualizes the solution for all time steps. To play the animation, use the following command:

```
animation(model,R2)
```

Find the maximum temperature of the disc. The maximum temperature is low enough to ensure that the braking pad performs as expected.

```
Tmax = max(max(R2.Temperature))
```

```
Tmax = 52.2895
```

Heat Source Functions for Moving and Static Heat Sources

```
function F = movingHeatSource(region,state)

% Parameters ---------

R = 0.115; % Distance from center of disc to center of braking pad
r = 0.025; % Radius of braking pad
```

```
xc = 0.15; % x-coordinate of center of disc
yc = 0.15; % y-coordinate of center of disc

T = 0.05; % Period of 1 revolution of disc

power = 35000; % Braking power in watts
Tambient = 30; % Ambient temperature (for convection)
h = 10; % Convection heat transfer coefficient in W/m^2*K
%--------------------

theta = 2*pi/T*state.time;

xs = xc + R*cos(theta);
ys = yc + R*sin(theta);

x = region.x;
y = region.y;

F = h*(Tambient - state.u); % Convection


if isnan(state.time)
    F = nan(1,numel(x));
end

idx = (x - xs).^2 + (y - ys).^2 <= r^2;

F(1,idx) = 0.5*power/(pi*r.^2);  % 0.5 because there are 2 faces

end

function F = ringHeatSource(region,state,driveline)

% Parameters ---------

R = 0.115; % Distance from center of disc to center of braking pad
r = 0.025; % Radius of braking pad
xc = 0.15; % x-coordinate of center of disc
yc = 0.15; % y-coordinate of center of disc

% Braking power in watts
power = interp1(driveline.tout,driveline.heatFlow,state.time);
Tambient = 30; % Ambient temperature (for convection)
h = 10; % Convection heat transfer coefficient in W/m^2*K
Tf = 2.5; % Time in seconds
%--------------------


x = region.x;
y = region.y;

F = h*(Tambient - state.u); % Convection

if isnan(state.time)
    F = nan(1,numel(x));
end
```

```
if state.time < Tf
    rad = sqrt((x - xc).^2 + (y - yc).^2);

    idx = rad >= R-r & rad <= R+r;

    area = pi*( (R+r)^2 - (R-r)^2 );
    F(1,idx) = 0.5*power/area;  % 0.5 because there are 2 faces
end

end
```

Heat Distribution in Circular Cylindrical Rod: PDE Modeler App

Solve a 3-D parabolic PDE problem by reducing the problem to 2-D using coordinate transformation. This example uses the PDE Modeler app. For the command-line solution, see “Heat Distribution in Circular Cylindrical Rod” on page 3-281.

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, ρ, 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, θ, 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:

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

Then define r as y and z as x:

$$\rho yC\frac{\partial u}{\partial t} - \nabla \cdot (ky\nabla u) = qy$$

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

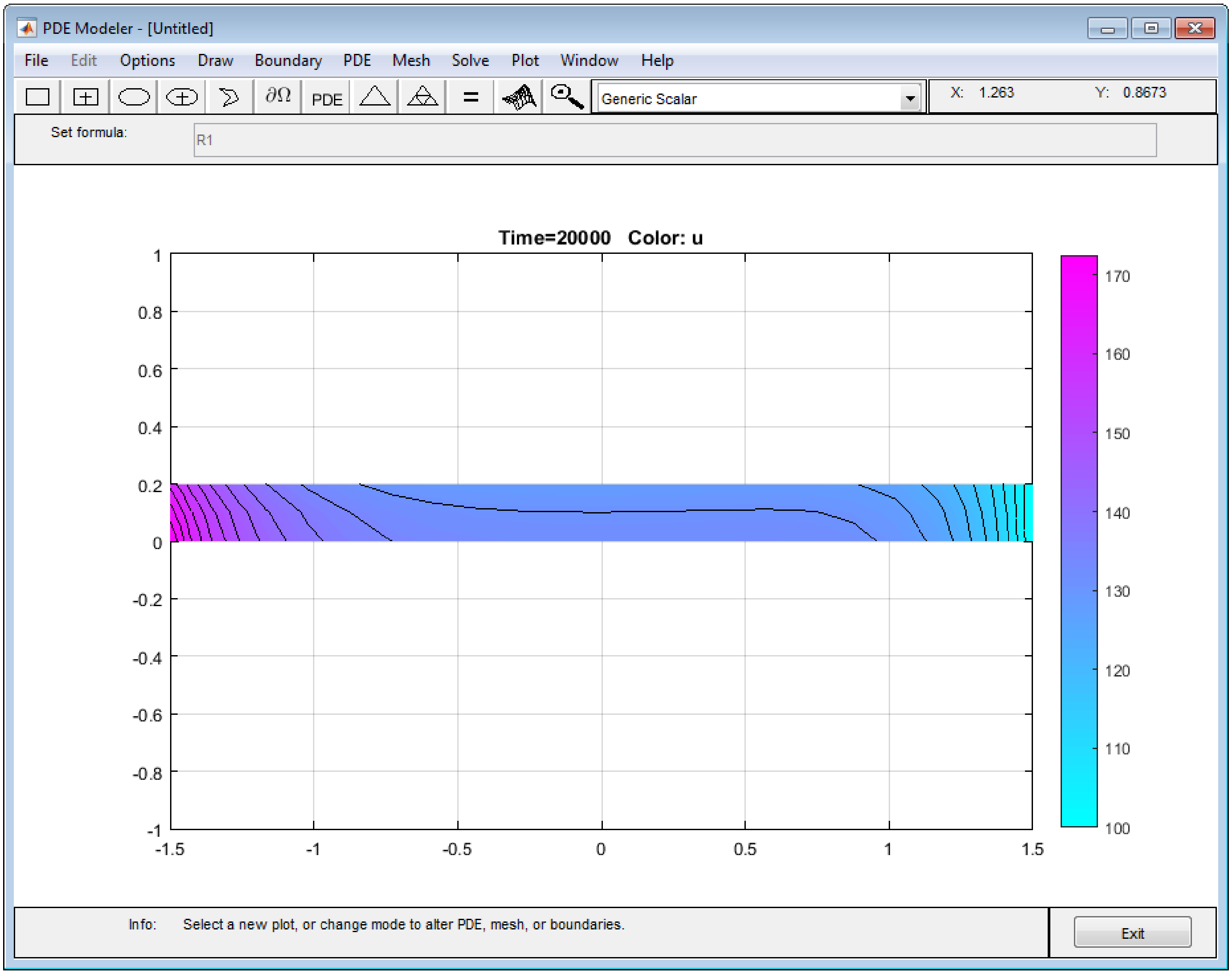
To solve this problem in the PDE Modeler app, follow these steps:

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 Modeler app requires boundary conditions in a particular form. Thus, Neumann boundary conditions must be in the form $\vec{n} \cdot (c\nabla u) + qu = g$, and Dirichlet boundary conditions must be in the form $hu = r$. 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.



X: 1.263 Y: 0.8673

Set formula: R1

Info: Select a new plot, or change mode to alter PDE, mesh, or boundaries.

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 on Square Domain

This example shows how to solve the wave equation using the `solvepde` function.

The standard second-order wave equation is

$$\frac{\partial^2 u}{\partial t^2} - \nabla \cdot \nabla u = 0\,.$$

To express this in toolbox form, note that the `solvepde` function solves problems of the form

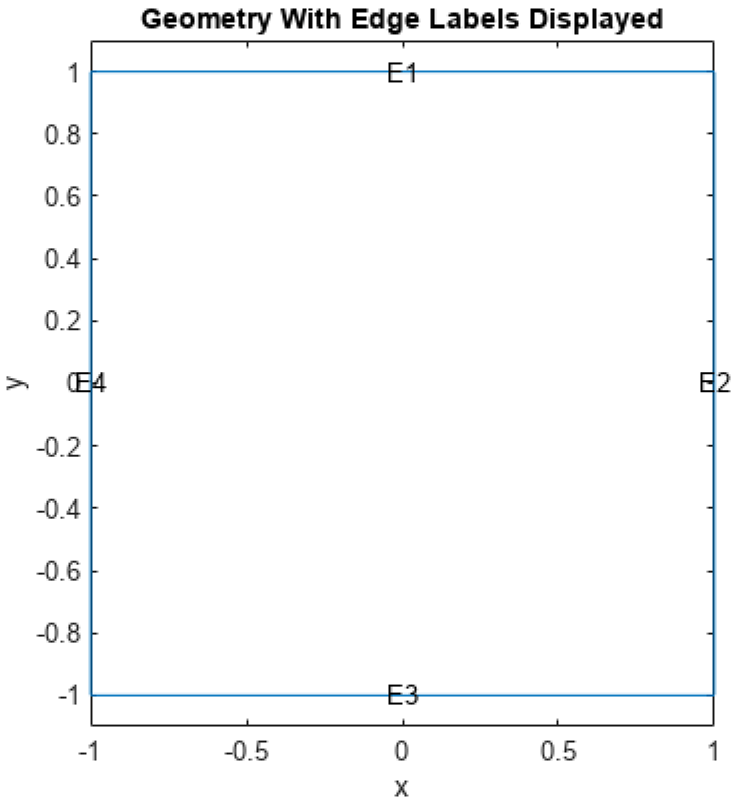
$$m\frac{\partial^2 u}{\partial t^2} - \nabla \cdot (c\nabla u) + au = f\,.$$

So the standard wave equation has coefficients $m = 1$, $c = 1$, $a = 0$, and $f = 0$.

```
c = 1;
a = 0;
f = 0;
m = 1;
```

Solve the problem on a square domain. The `squareg` function describes this geometry. Create a `model` object and include the geometry. Plot the geometry and view the edge labels.

```
numberOfPDE = 1;
model = createpde(numberOfPDE);
geometryFromEdges(model,@squareg);
pdegplot(model,"EdgeLabels","on");
ylim([-1.1 1.1]);
axis equal
title("Geometry With Edge Labels Displayed")
xlabel("x")
ylabel("y")
```



Specify PDE coefficients.

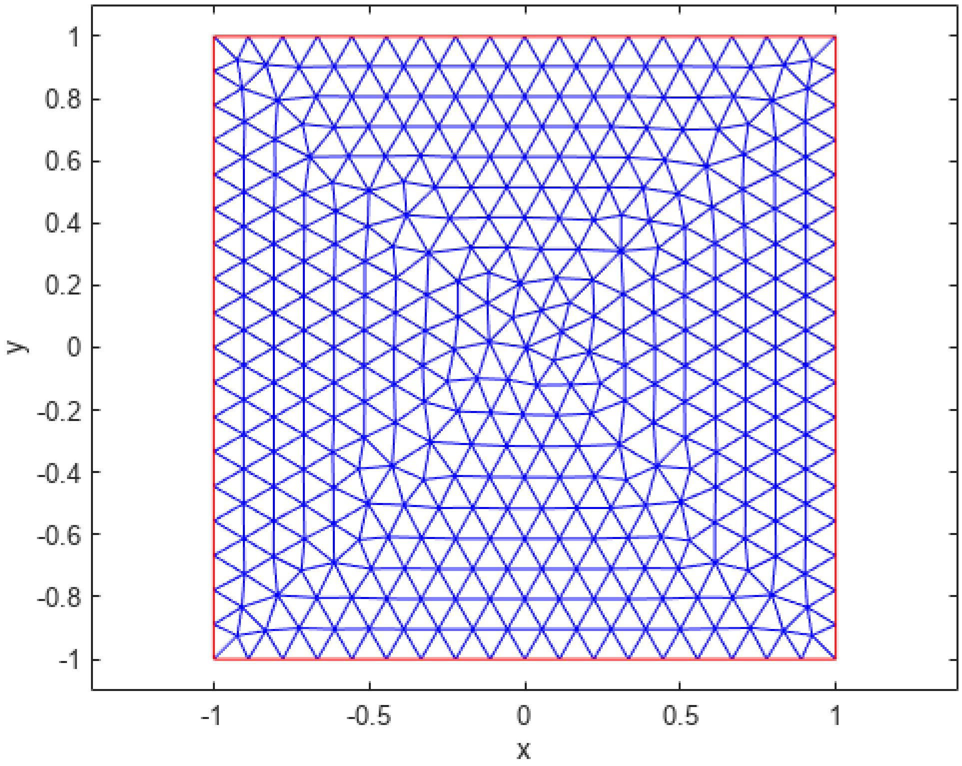
```
specifyCoefficients(model,"m",m,"d",0,"c",c,"a",a,"f",f);
```

Set zero Dirichlet boundary conditions on the left (edge 4) and right (edge 2) and zero Neumann boundary conditions on the top (edge 1) and bottom (edge 3).

```
applyBoundaryCondition(model,"dirichlet","Edge",[2,4],"u",0);
applyBoundaryCondition(model,"neumann","Edge",([1 3]),"g",0);
```

Create and view a finite element mesh for the problem.

```
generateMesh(model);
figure
pdemesh(model);
ylim([-1.1 1.1]);
axis equal
xlabel x
ylabel y
```



Set the following initial conditions:

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

```
u0 = @(location) atan(cos(pi/2*location.x));
ut0 = @(location) 3*sin(pi*location.x).*exp(sin(pi/2*location.y));
setInitialConditions(model,u0,ut0);
```

This choice avoids putting energy into the higher vibration modes and permits a reasonable time step size.

Specify the solution times as 31 equally-spaced points in time from 0 to 5.

```
n = 31;
tlist = linspace(0,5,n);
```

Set the `SolverOptions.ReportStatistics` of `model` to `'on'`.

```
model.SolverOptions.ReportStatistics ='on';
result = solvepde(model,tlist);
```

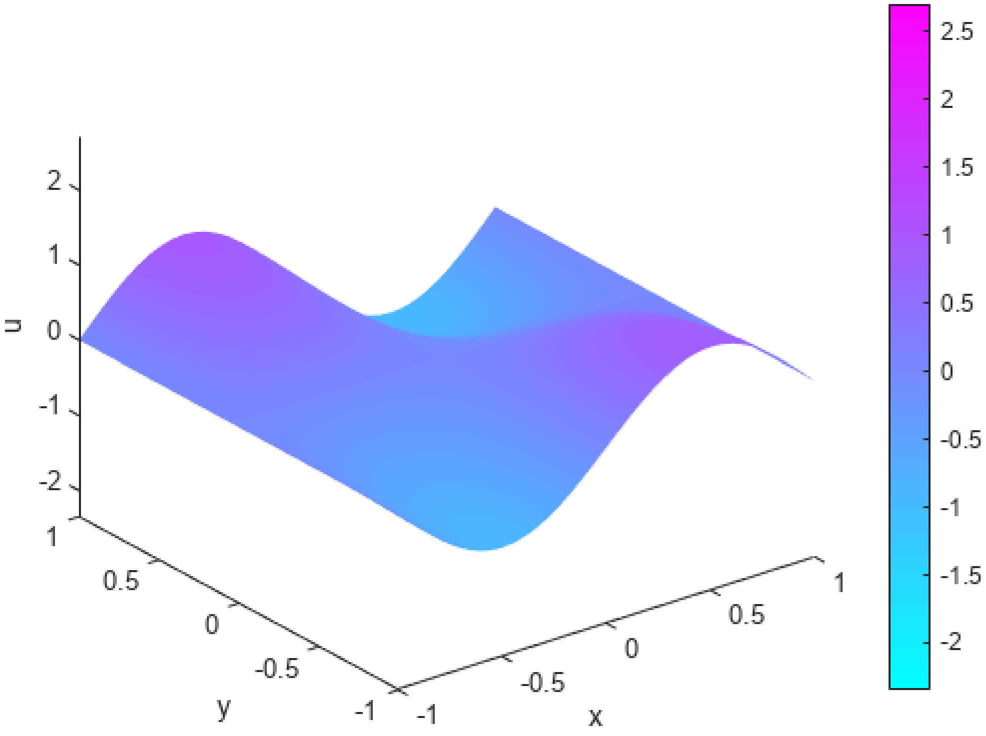
```
458 successful steps
37 failed attempts
989 function evaluations
```

```
1 partial derivatives
114 LU decompositions
988 solutions of linear systems
```

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

Create an animation to visualize the solution for all time steps. Keep a fixed vertical scale by first calculating the maximum and minimum values of `u` over all times, and scale all plots to use those *z*-axis limits.

```
figure
umax = max(max(u));
umin = min(min(u));
for i = 1:n
    pdeplot(model,"XYData",u(:,i),"ZData",u(:,i), ...
                  "ZStyle","continuous","Mesh","off");
    axis([-1 1 -1 1 umin umax]);
    caxis([umin umax]);
    xlabel x
    ylabel y
    zlabel u
    M(i) = getframe;
end
```



To play the animation, use the `movie(M)` command.

Wave Equation on Square Domain: PDE Modeler App

This example shows how to solve a wave equation for transverse vibrations of a membrane on a square. The membrane is fixed at the left and right sides, and is free at the upper and lower sides. This example uses the PDE Modeler app. For a programmatic workflow, see “Wave Equation on Square Domain” on page 3-298.

A wave equation is a hyperbolic PDE:

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

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

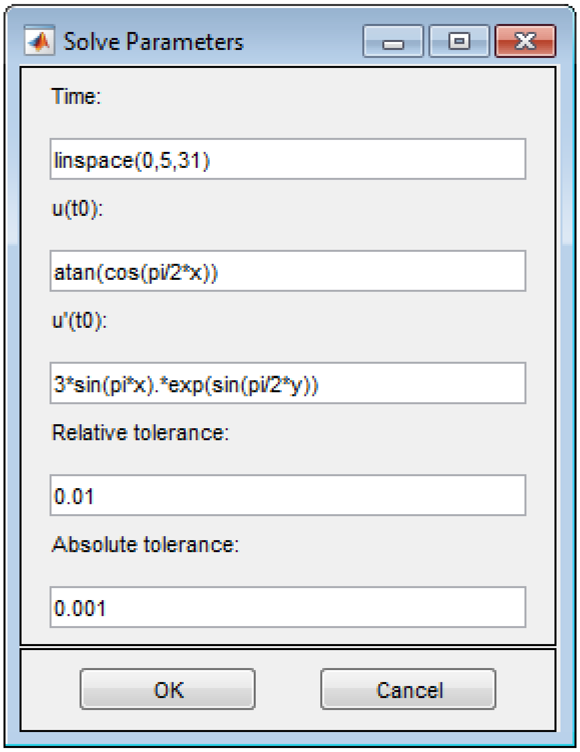
1. Open the PDE Modeler app by using the `pdeModeler` command.
2. Display grid lines by selecting **Options** > **Grid**.
3. Align new shapes to the grid lines by selecting **Options** > **Snap**.
4. Draw a square with the corners at (-1,-1), (-1,1), (1,1), and (1,-1). To do this, first click the ▭ button. Then click one of the corners using the right mouse button and drag to draw a square. The right mouse button constrains the shape you draw to be a square rather than a rectangle.

   You also can use the `pderect` function:

   ```
   pderect([-1 1 -1 1])
   ```
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 ∂Ω button or selecting **Boundary** > **Boundary Mode**. Select the left and right boundaries. 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.

   For the bottom and top boundaries, set the Neumann boundary condition $\partial u/\partial n = 0$. To do this, set `g = 0`, `q = 0`.
7. Specify the coefficients by selecting **PDEPDE Specification** or clicking the **PDE** button on the toolbar. Select the **Hyperbolic** type of PDE, and specify `c = 1`, `a = 0`, `f = 0`, and `d = 1`.
8. Initialize the mesh by selecting **Mesh** > **Initialize Mesh**. Refine the mesh by selecting **Mesh** > **Refine Mesh**.
9. Set the solution times. To do this, select **Solve** > **Parameters**. Create linearly spaced time vector from 0 to 5 seconds by setting the solution time to `linspace(0,5,31)`.
10. In the same dialog box, specify initial conditions for the wave equation. For a well-behaved solution, the initial values must match the boundary conditions. If the initial time is $t = 0$, then the following initial values that satisfy the boundary conditions: `atan(cos(pi/2*x))` for `u(0)` and `3*sin(pi*x).*exp(sin(pi/2*y))` for $\partial u/\partial t$,

    The inverse tangent function and exponential function introduce more modes into the solution.



Time:

linspace(0,5,31)

u(t0):

atan(cos(pi/2*x))

u'(t0):

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

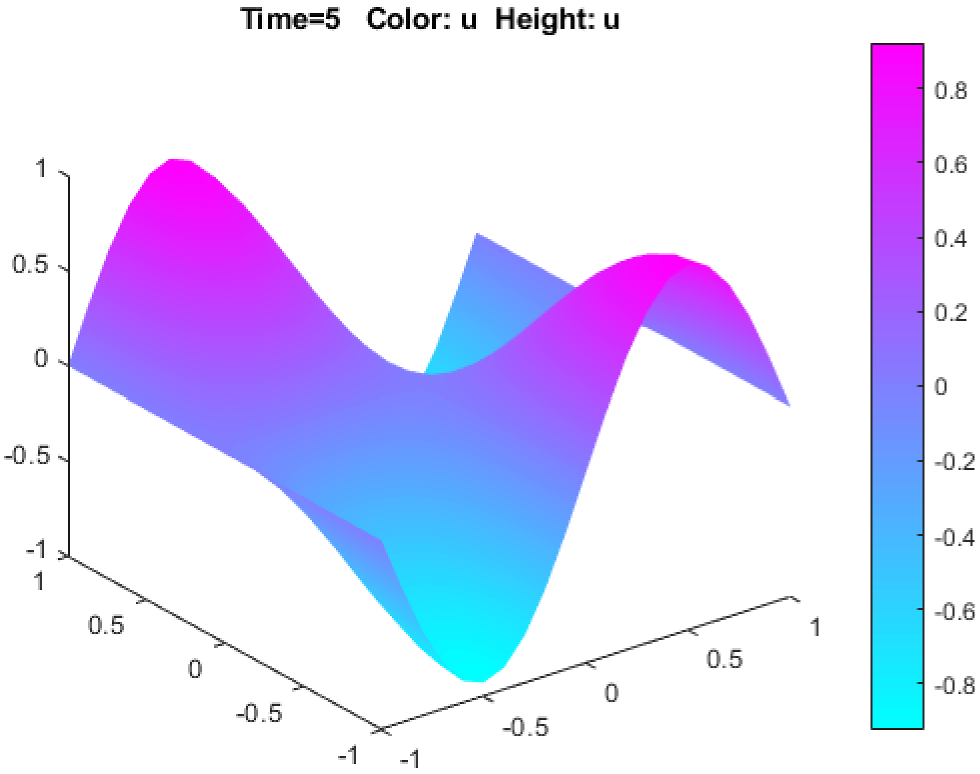
Relative tolerance:

0.01

Absolute tolerance:

0.001

11. Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar. The app solves the heat equation at times from 0 to 5 seconds and displays the result at the end of the time span.
12. Visualize the solution as a 3-D static and animated plots. To do this:
    - **a** Select **Plot** > **Parameters**.
    - **b** In the resulting dialog box, select the **Color** and **Height (3-D plot)** options.
    - **c** To visualize the dynamic behavior of the wave, select **Animation** in the same dialog box. If the animation progress is too slow, select the **Plot in x-y grid** option. An *x*-*y* grid can speed up the animation process significantly.


Time=5 Color: u Height: u
1
0.5
0
-0.5
-1
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8

Eigenvalues and Eigenmodes of L-Shaped Membrane

This example shows how to calculate eigenvalues and eigenvectors. 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=   0.19,  New conv eig=  0
              Basis= 11,  Time=   0.27,  New conv eig=  0
              Basis= 12,  Time=   0.30,  New conv eig=  0
              Basis= 13,  Time=   0.30,  New conv eig=  0
              Basis= 14,  Time=   0.33,  New conv eig=  0
              Basis= 15,  Time=   0.33,  New conv eig=  0
              Basis= 16,  Time=   0.36,  New conv eig=  0
              Basis= 17,  Time=   0.42,  New conv eig=  1
              Basis= 18,  Time=   0.42,  New conv eig=  1
              Basis= 19,  Time=   0.48,  New conv eig=  1
              Basis= 20,  Time=   0.48,  New conv eig=  1
              Basis= 21,  Time=   0.48,  New conv eig=  1
              Basis= 22,  Time=   0.62,  New conv eig=  2
              Basis= 23,  Time=   0.72,  New conv eig=  3
              Basis= 24,  Time=   0.84,  New conv eig=  3
              Basis= 25,  Time=   0.89,  New conv eig=  3
              Basis= 26,  Time=   0.95,  New conv eig=  3
              Basis= 27,  Time=   1.27,  New conv eig=  4
              Basis= 28,  Time=   1.34,  New conv eig=  4
              Basis= 29,  Time=   1.42,  New conv eig=  4
              Basis= 30,  Time=   1.44,  New conv eig=  4
              Basis= 31,  Time=   1.48,  New conv eig=  5
              Basis= 32,  Time=   1.56,  New conv eig=  5
              Basis= 33,  Time=   1.64,  New conv eig=  5
              Basis= 34,  Time=   1.69,  New conv eig=  5
              Basis= 35,  Time=   1.72,  New conv eig=  6
              Basis= 36,  Time=   1.78,  New conv eig=  6
```

```
                Basis= 37,  Time=   1.78,  New conv eig=  7
                Basis= 38,  Time=   1.86,  New conv eig=  7
                Basis= 39,  Time=   1.97,  New conv eig=  7
                Basis= 40,  Time=   2.03,  New conv eig=  7
                Basis= 41,  Time=   2.08,  New conv eig=  7
                Basis= 42,  Time=   2.08,  New conv eig=  7
                Basis= 43,  Time=   2.08,  New conv eig=  8
                Basis= 44,  Time=   2.14,  New conv eig=  8
                Basis= 45,  Time=   2.14,  New conv eig=  8
                Basis= 46,  Time=   2.28,  New conv eig= 11
                Basis= 47,  Time=   2.28,  New conv eig= 11
                Basis= 48,  Time=   2.30,  New conv eig= 11
                Basis= 49,  Time=   2.34,  New conv eig= 11
                Basis= 50,  Time=   2.56,  New conv eig= 12
                Basis= 51,  Time=   2.64,  New conv eig= 12
                Basis= 52,  Time=   2.72,  New conv eig= 12
                Basis= 53,  Time=   2.80,  New conv eig= 13
                Basis= 54,  Time=   2.83,  New conv eig= 14
                Basis= 55,  Time=   3.12,  New conv eig= 14
                Basis= 56,  Time=   3.31,  New conv eig= 14
                Basis= 57,  Time=   3.36,  New conv eig= 14
                Basis= 58,  Time=   3.47,  New conv eig= 14
                Basis= 59,  Time=   3.64,  New conv eig= 15
                Basis= 60,  Time=   3.86,  New conv eig= 16
                Basis= 61,  Time=   3.92,  New conv eig= 16
                Basis= 62,  Time=   3.97,  New conv eig= 16
                Basis= 63,  Time=   4.00,  New conv eig= 16
                Basis= 64,  Time=   4.09,  New conv eig= 16
                Basis= 65,  Time=   4.17,  New conv eig= 16
                Basis= 66,  Time=   4.33,  New conv eig= 18
                Basis= 67,  Time=   4.47,  New conv eig= 18
                Basis= 68,  Time=   4.61,  New conv eig= 18
                Basis= 69,  Time=   4.69,  New conv eig= 18
                Basis= 70,  Time=   4.77,  New conv eig= 18
                Basis= 71,  Time=   4.84,  New conv eig= 19
                Basis= 72,  Time=   5.02,  New conv eig= 20
                Basis= 73,  Time=   5.19,  New conv eig= 21
End of sweep: Basis= 73,  Time=   5.23,  New conv eig= 21
                Basis= 31,  Time=   5.59,  New conv eig=  0
                Basis= 32,  Time=   5.62,  New conv eig=  0
                Basis= 33,  Time=   5.83,  New conv eig=  0
                Basis= 34,  Time=   5.83,  New conv eig=  0
                Basis= 35,  Time=   5.89,  New conv eig=  0
                Basis= 36,  Time=   5.95,  New conv eig=  0
                Basis= 37,  Time=   5.95,  New conv eig=  0
End of sweep: Basis= 37,  Time=   5.95,  New conv eig=  0
```

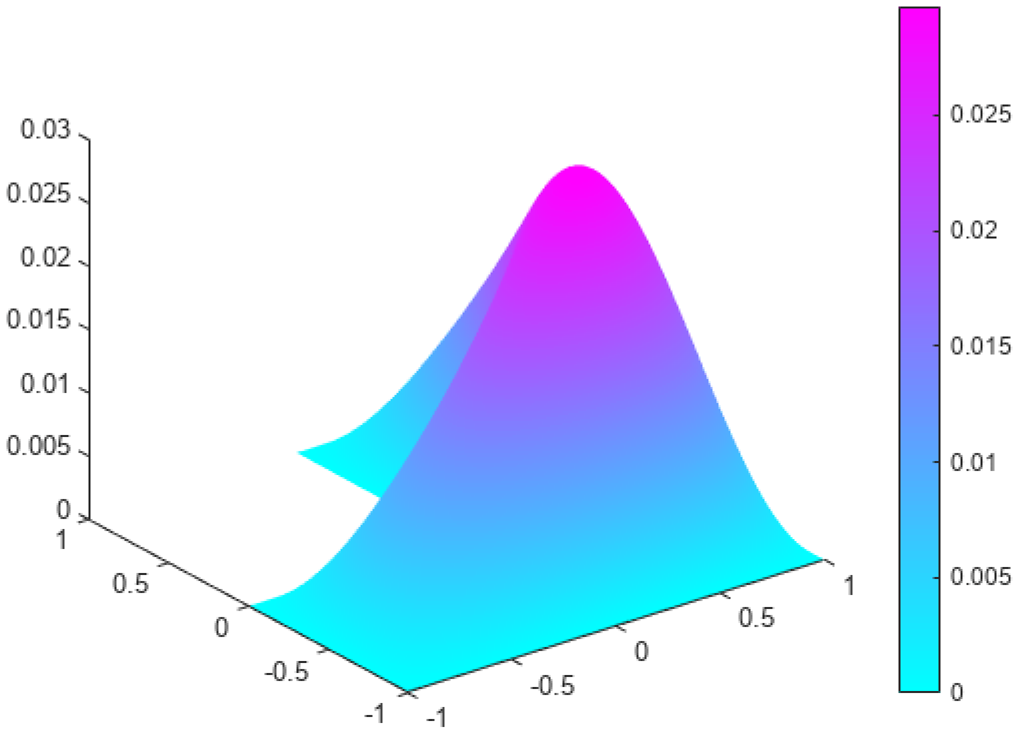
There are 19 eigenvalues smaller than 100.

```
length(results.Eigenvalues)
```

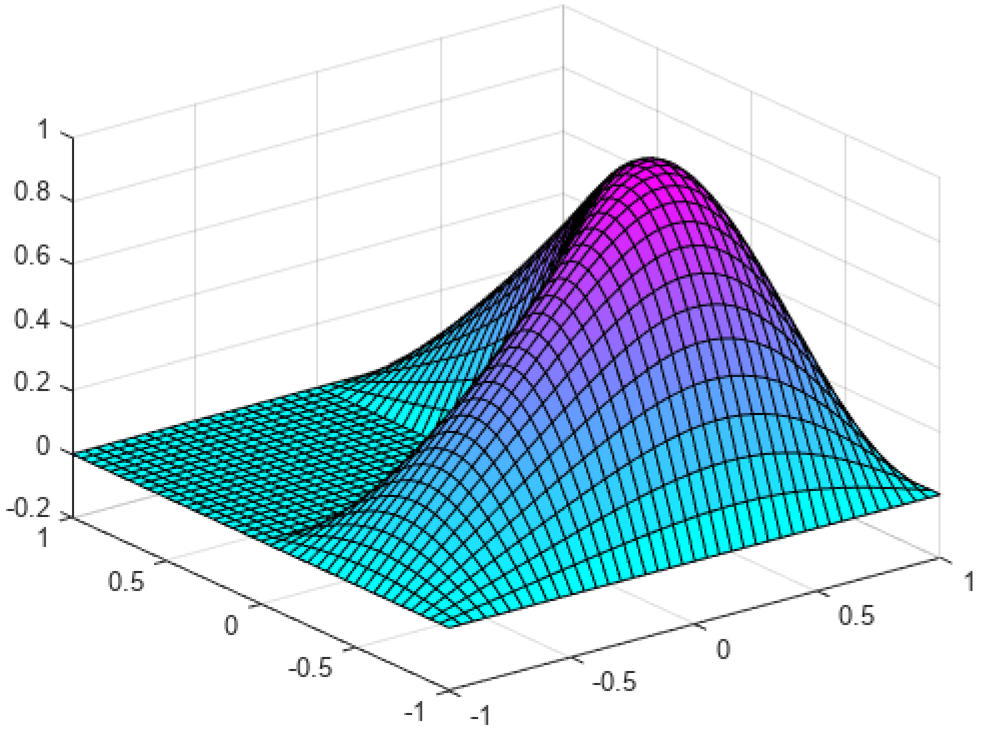
```
ans = 19
```

Plot the first eigenmode and compare it to the MATLAB's `membrane` function.

```
u = results.Eigenvectors;
pdeplot(model,"XYData",u(:,1),"ZData",u(:,1));
```



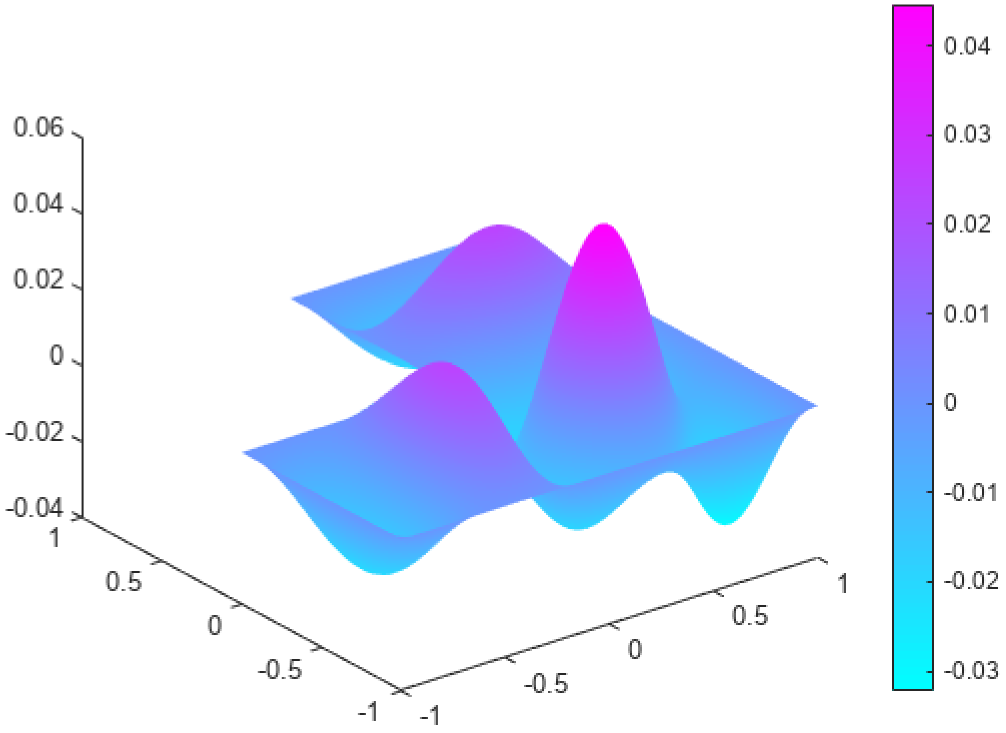
```
figure
membrane(1,20,9,9)
```



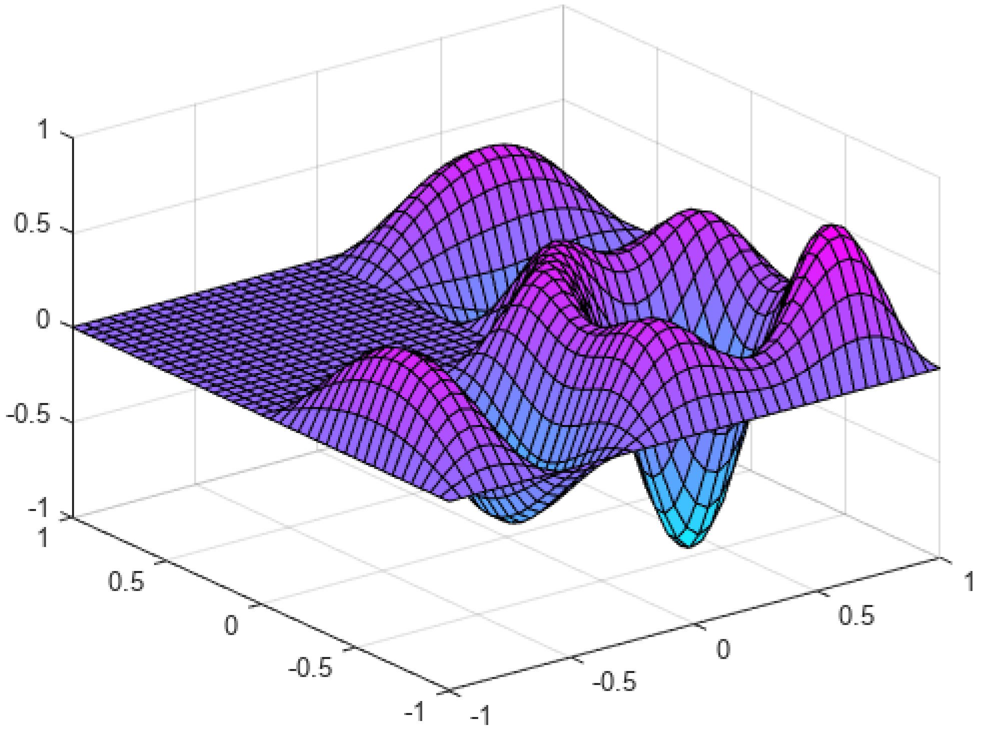
Eigenvectors can be multiplied by any scalar and remain eigenvectors. This explains the difference in scale that you see.

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

```
figure
pdeplot(model,"XYData",u(:,12),"ZData",u(:,12));
```



```
figure
membrane(12,20,9,9)
```


1
0.5
0
-0.5
-1
1
0.5
0
-0.5
-1
-1
-0.5
0
0.5
1

Eigenvalues and Eigenmodes of L-Shaped Membrane: PDE Modeler App

This example shows how to compute all eigenmodes with eigenvalues smaller than 100 for the eigenmode PDE problem

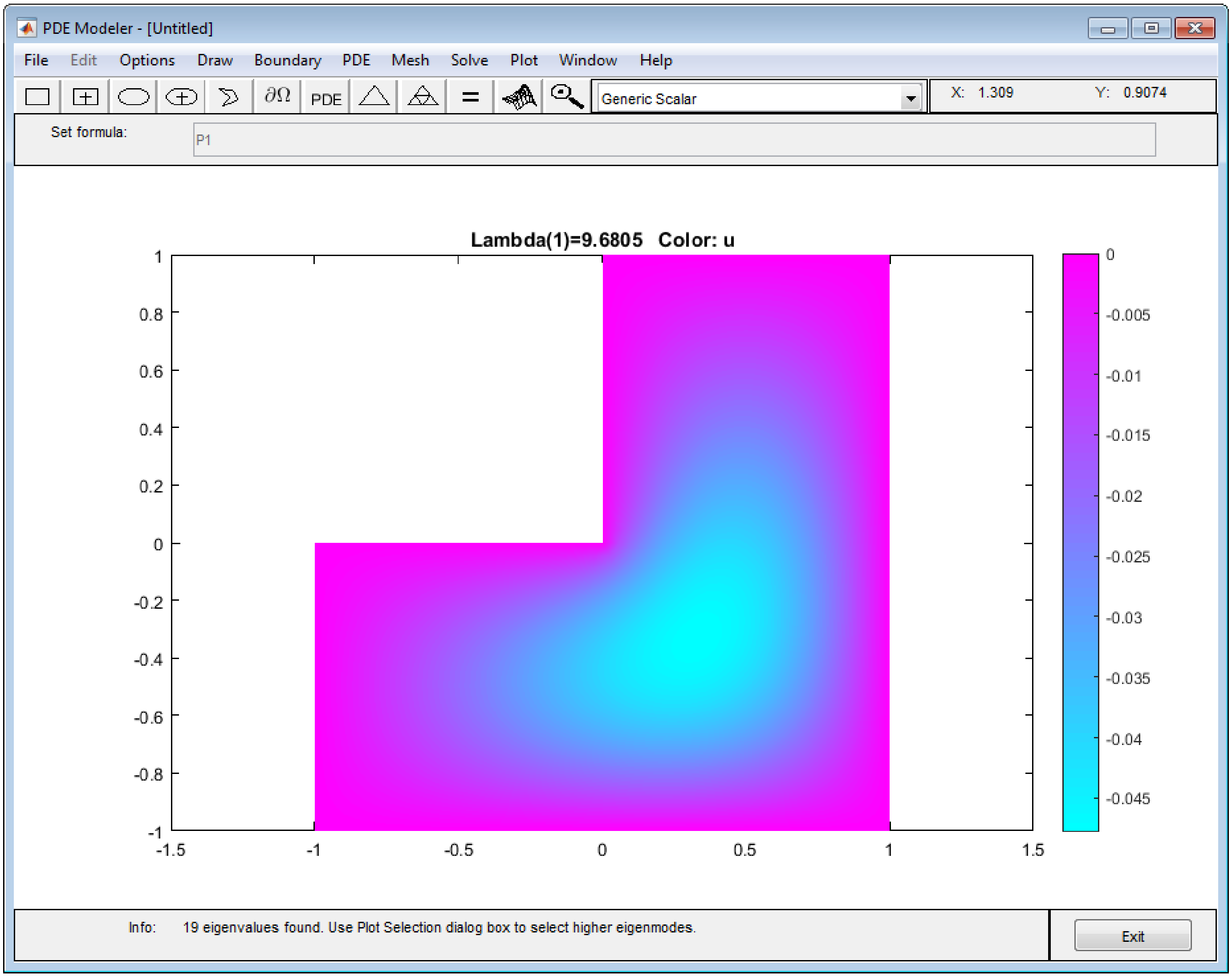
$$-\Delta u = \lambda u$$

on the geometry of the L-shaped membrane. The boundary condition is the Dirichlet condition $u = 0$. This example uses the PDE Modeler app. For a programmatic workflow, see “Eigenvalues and Eigenmodes of L-Shaped Membrane” on page 3-305.

To solve this problem in the PDE Modeler 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.

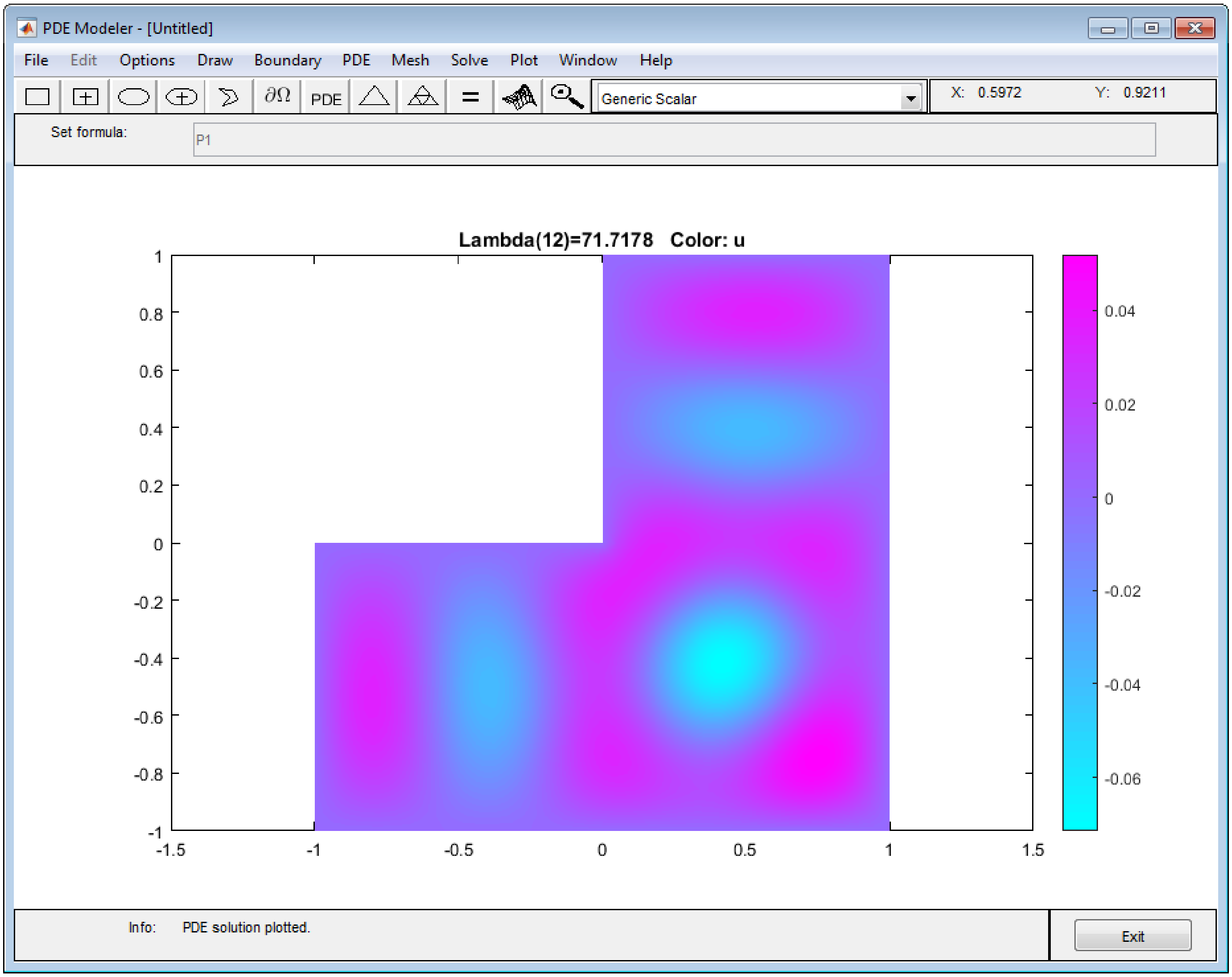
   ```
   pdepoly([0,-1,-1,1,1,0],[0,0,-1,-1,1,1])
   ```
2. Check that the application mode is set to **Generic Scalar**.
3. Use the default Dirichlet boundary condition $u = 0$ for all boundaries. To verify it, switch to boundary mode by selecting **Boundary** > **Boundary Mode**. Use **Edit** > **Select 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.



Set formula: P1

Info: 19 eigenvalues found. Use Plot Selection dialog box to select higher eigenmodes.

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.



Generic Scalar

X: 0.5972 Y: 0.9211

Set formula: P1

Lambda(12)=71.7178 Color: u

Info: PDE solution plotted.

L-Shaped Membrane with Rounded Corner: PDE Modeler App

This example shows how to compute all eigenvalues smaller than 100 and their corresponding eigenvectors. Consider the eigenvalue problem

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

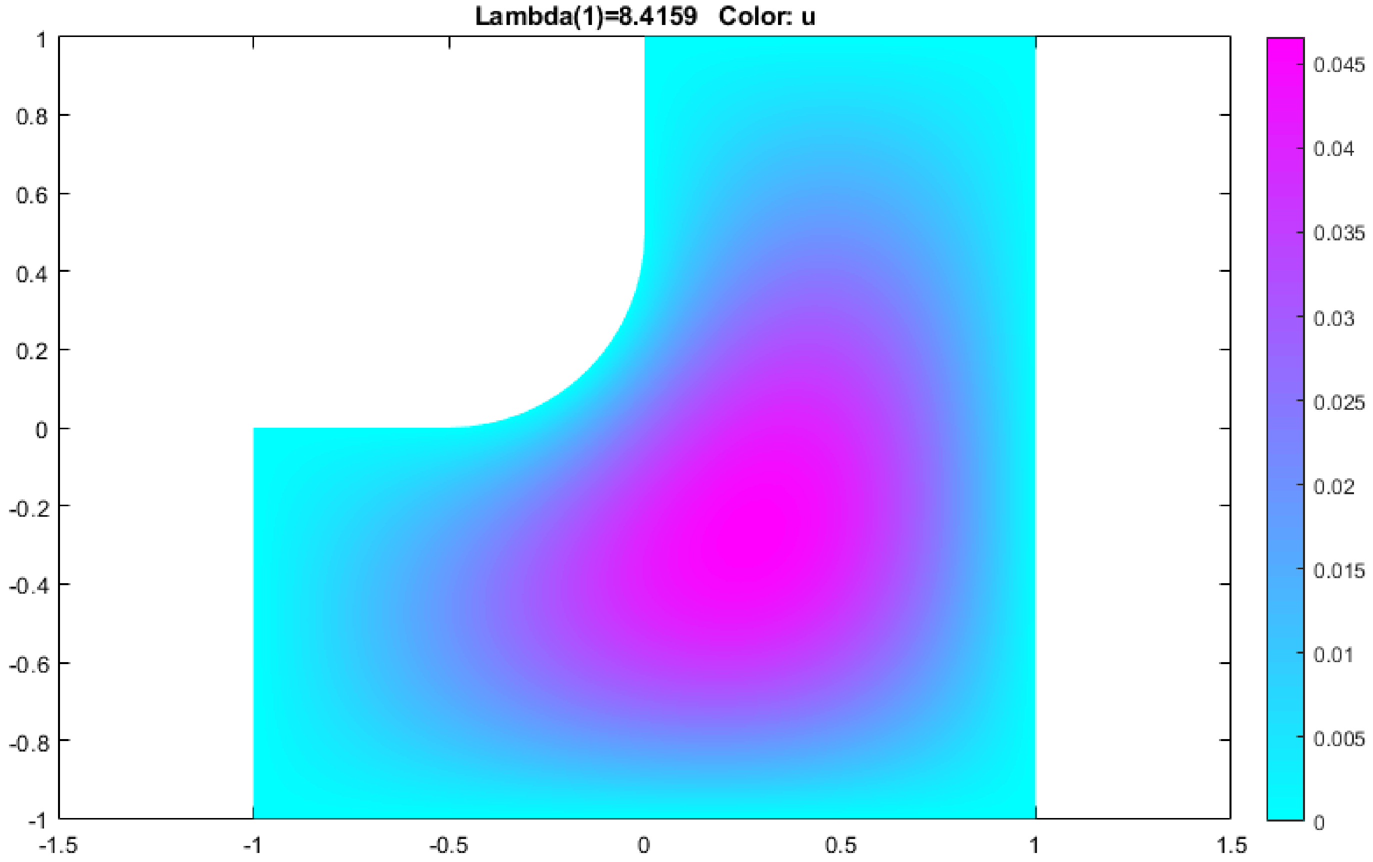
on an L-shaped membrane with a rounded inner corner. The boundary condition is the Dirichlet condition $u = 0$.

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

1. Draw an L-shaped membrane as a polygon with the corners (0,0), (–1,0), (–1,–1), (1,–1), (1,1), and (0,1) by using the `pdepoly` function.

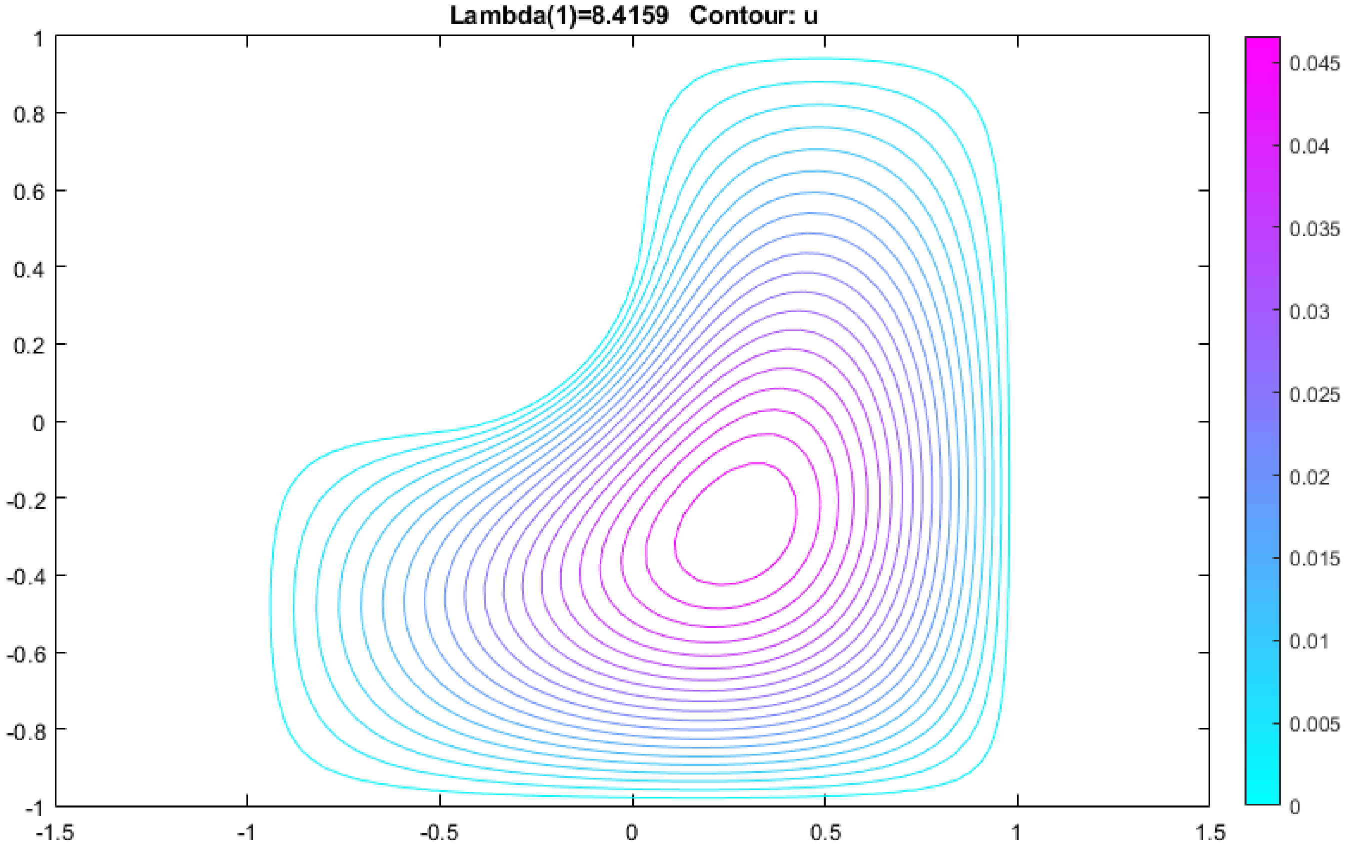
   ```
   pdepoly([0 -1 -1 1 1 0],[0 0 -1 -1 1 1])
   ```
2. Draw a circle and a square as follows.

   ```
   pdecirc(-0.5,0.5,0.5,'C1')
   pderect([-0.5 0 0.5 0],'SQ1')
   ```
3. Model the geometry with the rounded corner by entering `P1+SQ1-C1` in the **Set formula** field.
4. Check that the application mode is set to **Generic Scalar**.
5. Remove unnecessary subdomain borders by selecting **Boundary** > **Remove All Subdomain Borders**.
6. Use the default Dirichlet boundary condition $u = 0$ for all boundaries. To check the boundary condition, 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`.
7. 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** as the 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`.
8. Specify the maximum edge size for the mesh by selecting **Mesh** > **Parameters**. Set the maximum edge size value to 0.05.
9. Initialize the mesh by selecting **Mesh** > **Initialize Mesh**.
10. Specify the eigenvalue range by selecting **Solve** > **Parameters**. In the resulting dialog box, use the default eigenvalue range `[0 100]`.
11. Solve the PDE by selecting **Solve** > **Solve PDE** or clicking the **=** button on the toolbar. By default, the app plots the first eigenfunction as a color plot.



12 Plot the same eigenfunction as a contour plot. To do this:

- **a** Select **Plot** > **Parameters**.
- **b** Clear the **Color** option and select the **Contour** option.


Lambda(1)=8.4159 Contour: u
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1.5
-1
-0.5
0
0.5
1
1.5
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0

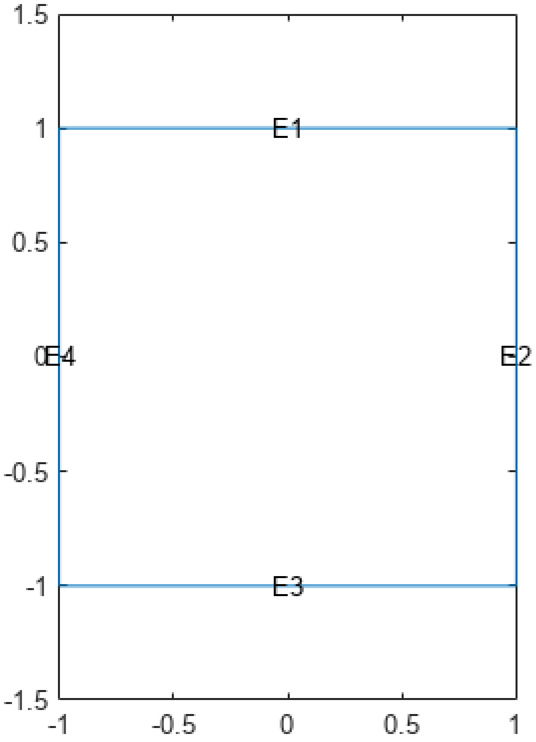
Eigenvalues and Eigenmodes of Square

This example shows how to compute the eigenvalues and eigenmodes of a square domain.

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=   0.98,  New conv eig=  0
              Basis= 11,  Time=   1.06,  New conv eig=  0
              Basis= 12,  Time=   1.06,  New conv eig=  1
              Basis= 13,  Time=   1.08,  New conv eig=  1
              Basis= 14,  Time=   1.17,  New conv eig=  1
              Basis= 15,  Time=   1.20,  New conv eig=  1
              Basis= 16,  Time=   1.22,  New conv eig=  1
              Basis= 17,  Time=   1.34,  New conv eig=  1
              Basis= 18,  Time=   1.45,  New conv eig=  2
              Basis= 19,  Time=   1.47,  New conv eig=  2
              Basis= 20,  Time=   1.61,  New conv eig=  2
              Basis= 21,  Time=   1.75,  New conv eig=  3
              Basis= 22,  Time=   1.77,  New conv eig=  3
              Basis= 23,  Time=   1.86,  New conv eig=  4
              Basis= 24,  Time=   1.92,  New conv eig=  6
End of sweep: Basis= 24,  Time=   1.97,  New conv eig=  3
              Basis= 13,  Time=   2.27,  New conv eig=  0
              Basis= 14,  Time=   2.38,  New conv eig=  0
              Basis= 15,  Time=   2.42,  New conv eig=  0
              Basis= 16,  Time=   2.44,  New conv eig=  0
              Basis= 17,  Time=   2.47,  New conv eig=  0
              Basis= 18,  Time=   2.53,  New conv eig=  0
              Basis= 19,  Time=   2.53,  New conv eig=  0
              Basis= 20,  Time=   2.61,  New conv eig=  0
              Basis= 21,  Time=   2.64,  New conv eig=  0
              Basis= 22,  Time=   2.66,  New conv eig=  1
              Basis= 23,  Time=   2.70,  New conv eig=  2
End of sweep: Basis= 23,  Time=   2.72,  New conv eig=  0
              Basis= 13,  Time=   3.03,  New conv eig=  1
              Basis= 14,  Time=   3.06,  New conv eig=  1
              Basis= 15,  Time=   3.11,  New conv eig=  1
              Basis= 16,  Time=   3.11,  New conv eig=  3
              Basis= 17,  Time=   3.20,  New conv eig=  3
              Basis= 18,  Time=   3.25,  New conv eig=  3
              Basis= 19,  Time=   3.28,  New conv eig=  3
              Basis= 20,  Time=   3.28,  New conv eig=  3
              Basis= 21,  Time=   3.38,  New conv eig=  4
End of sweep: Basis= 21,  Time=   3.38,  New conv eig=  4
              Basis= 17,  Time=   3.73,  New conv eig=  0
End of sweep: Basis= 17,  Time=   3.81,  New conv eig=  0
```

There are six eigenvalues smaller than 10 for this problem.

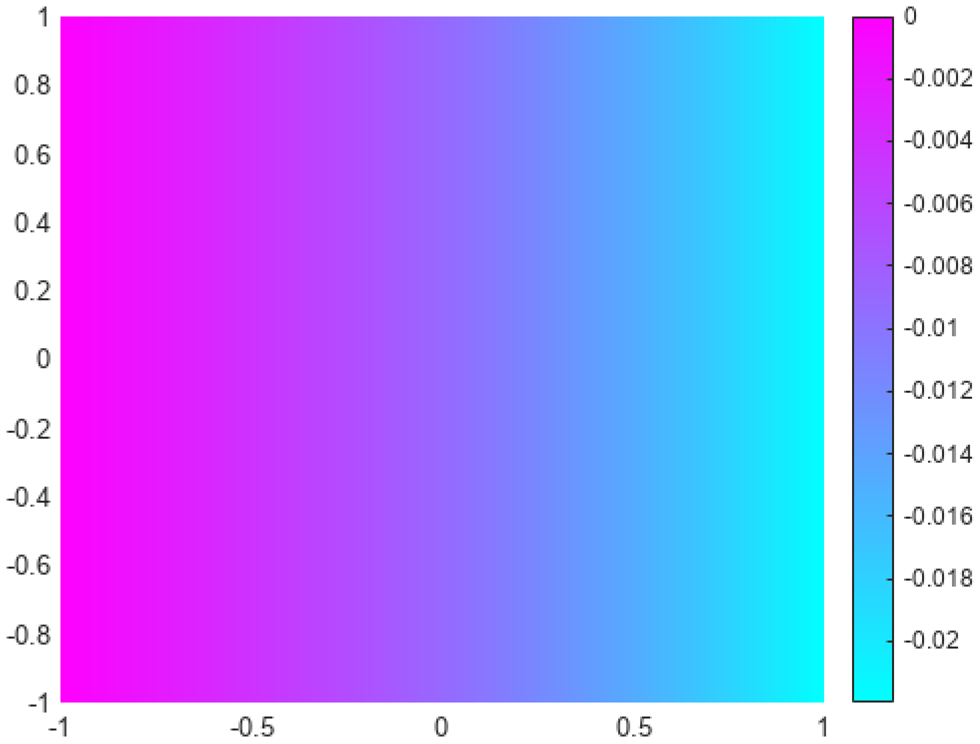
```
l = results.Eigenvalues
```

```
l = 5×1
```

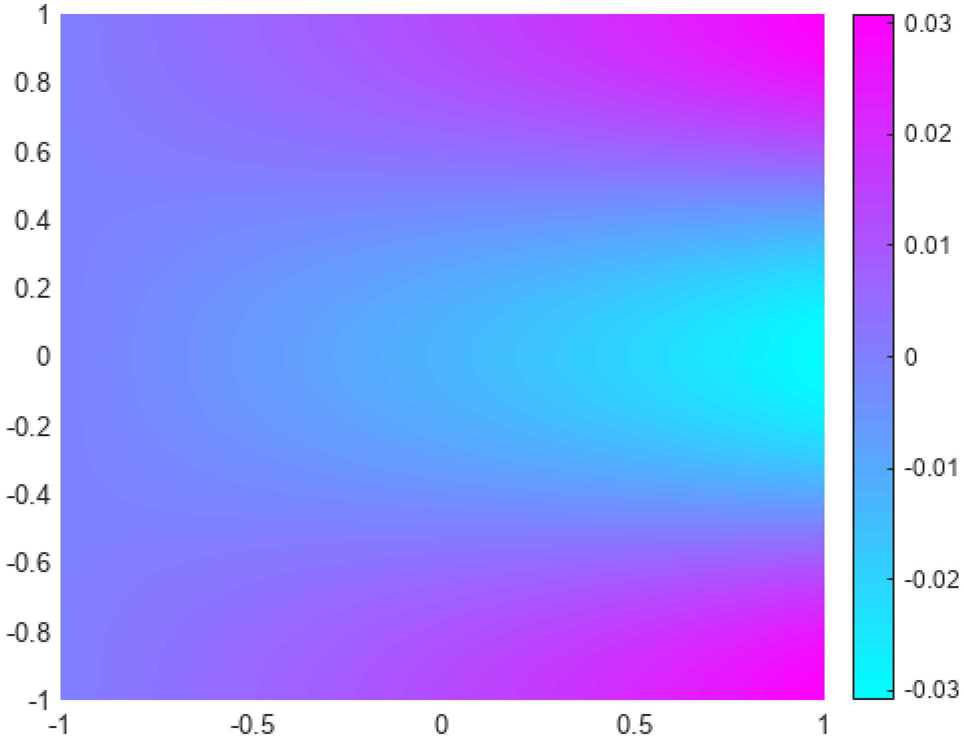
```
   -0.4146
    2.0528
    4.8019
    7.2693
    9.4550
```

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

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

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 = 1.6754e-07
```

Likewise, the fifth eigenmode is made up of the first eigenmode in the x direction and the third eigenmode in the y direction. As expected, `l(5)-l(1)` is approximately equal to π^2:

```
l(5) - l(1) - pi^2
```

```
ans = 6.2135e-06
```

You can explore higher modes by increasing the search range to include eigenvalues greater than 10.

Eigenvalues and Eigenmodes of Square: PDE Modeler App

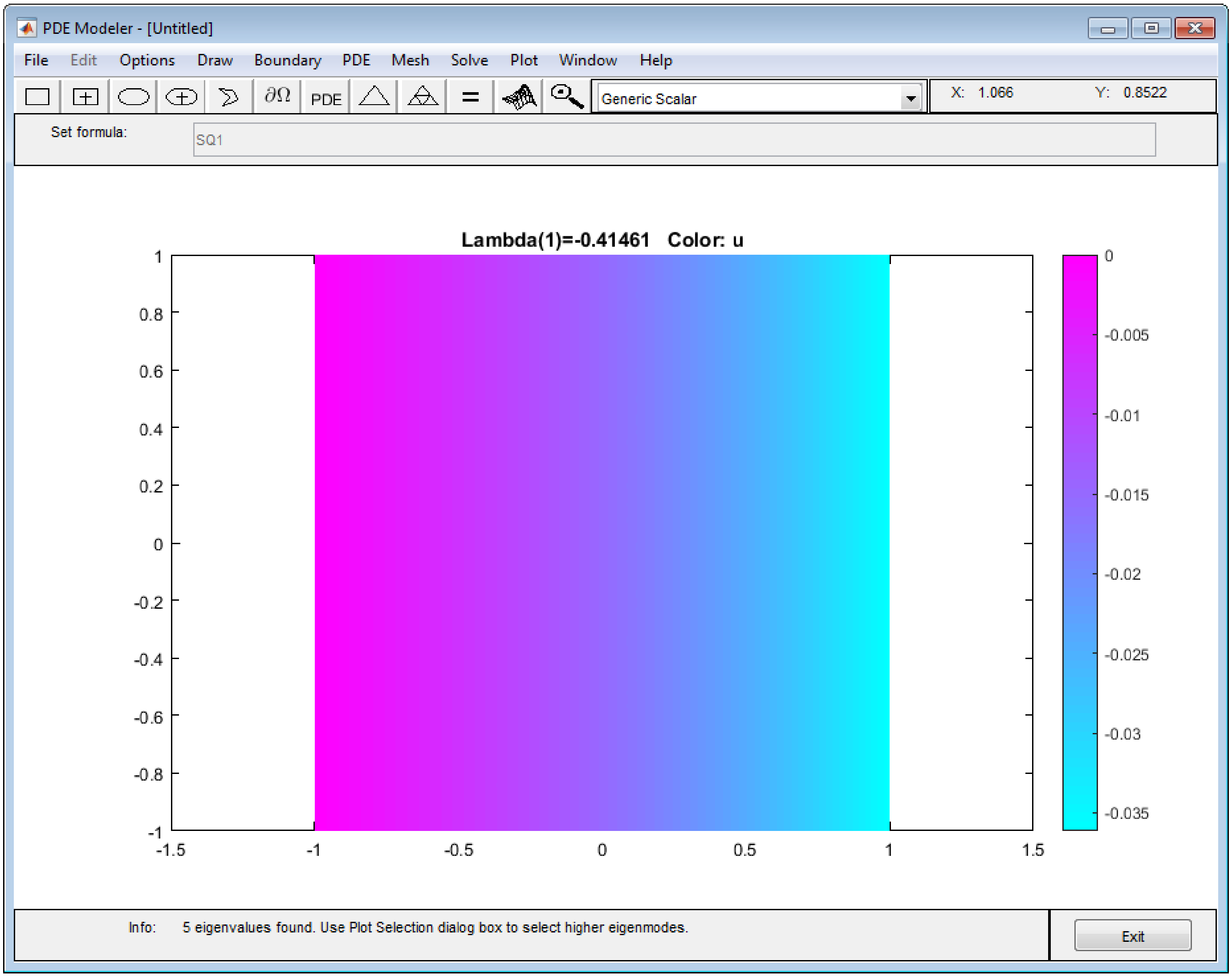
This example shows how to compute the eigenvalues and eigenmodes of a square with the corners (-1,-1), (-1,1), (1,1), and (1,-1). This example uses the PDE Modeler app. For programmatic workflow, see “Eigenvalues and Eigenmodes of Square” on page 3-317.

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 Modeler 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.

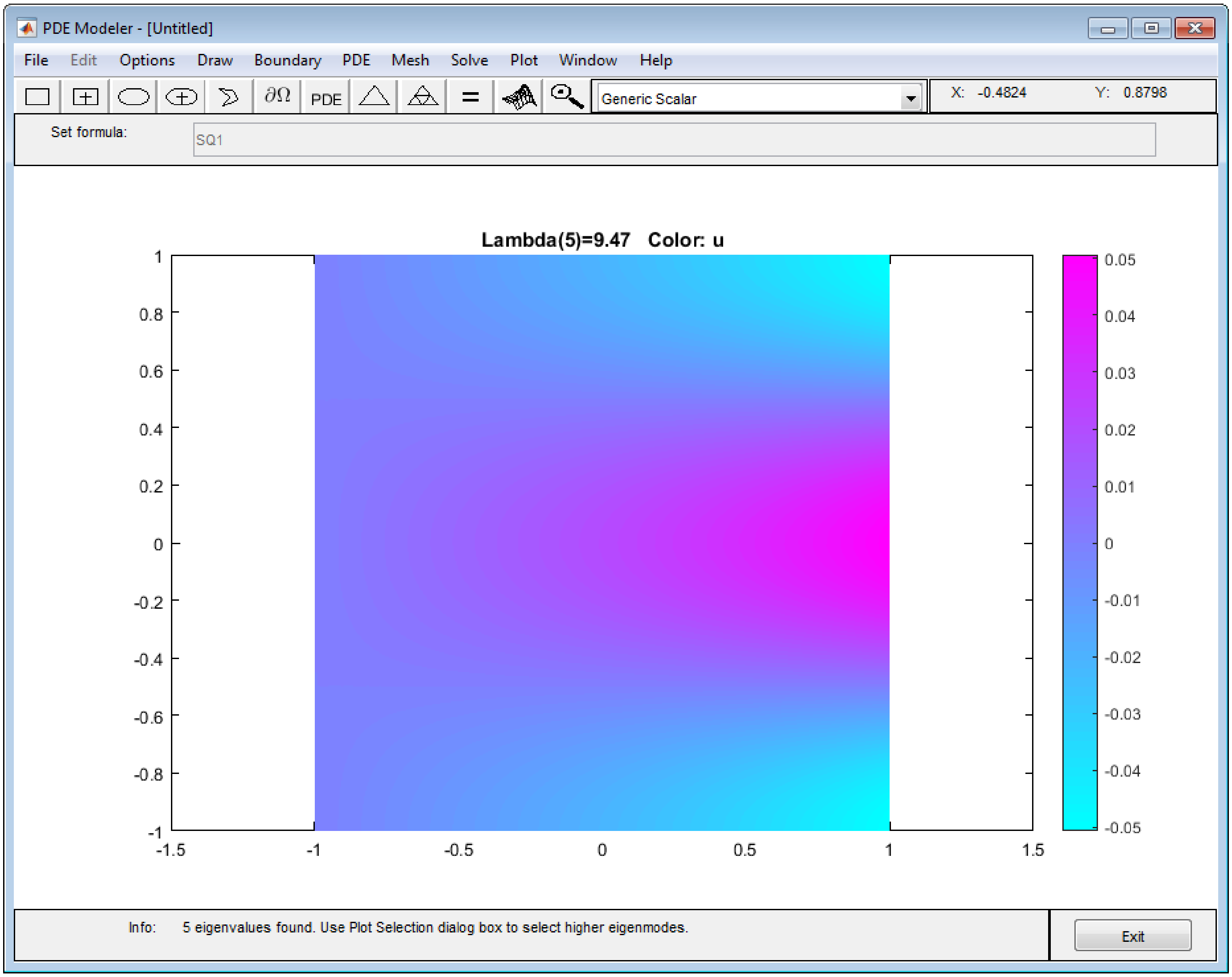
   ```
   pderect([-1 1 -1 1])
   ```
2. Check that the application mode is set to **Generic Scalar**.
3. Specify the boundary conditions. To do this, switch to the boundary mode by selecting **Boundary** > **Boundary Mode**. Double-click the boundary to specify the boundary condition.
   - Specify the Dirichlet condition $u = 0$ for the left boundary. To do this, specify `h = 1`, `r = 0`.
   - Specify the Neumann condition $\frac{\partial u}{\partial n} = 0$ for the upper and lower boundary. To do this, specify `g = 0`, `q = 0`.
   - Specify the generalized Neumann condition $\frac{\partial u}{\partial n} - \frac{3}{4}u = 0$ 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.



Set formula: SQ1

Info: 5 eigenvalues found. Use Plot Selection dialog box to select higher eigenmodes.

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.



Generic Scalar

X: -0.4824 Y: 0.8798

Set formula: SQ1

Info: 5 eigenvalues found. Use Plot Selection dialog box to select higher eigenmodes.

10 Export the eigenfunctions and eigenvalues to the MATLAB workspace by using the **Solve** > **Export Solution**.

Vibration of Circular Membrane

This example shows how to calculate the vibration modes of a circular membrane.

The calculation of vibration modes requires the solution of the eigenvalue partial differential equation. This example compares the solution obtained by using the `solvepdeeig` solver from Partial Differential Toolbox™ and the `eigs` solver from MATLAB®. Eigenvalues calculated by `solvepdeeig` and `eigs` are practically identical, but in some cases one solver is more convenient than the other. For example, `eigs` is more convenient when calculating a specified number of eigenvalues in the vicinity of a particular target value. While `solvepdeeig` requires that a lower and upper bound surrounding this target, `eigs` requires only the target eigenvalue and the desired number of eigenvalues.

Create a PDE model.

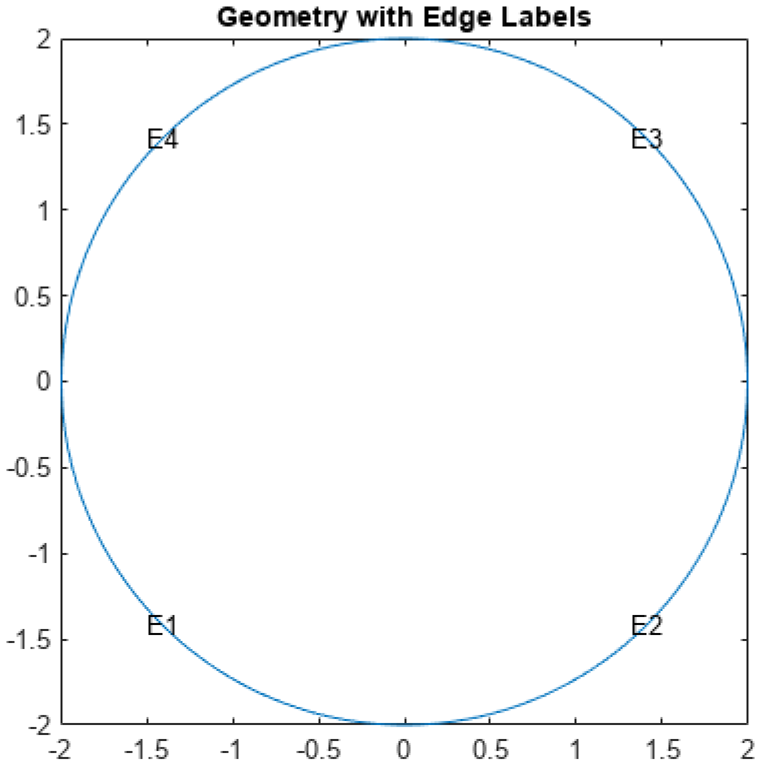
```
model = createpde;
```

Create the circle geometry and include it in the model.

```
radius = 2;
g = decsg([1 0 0 radius]','C1',('C1')');

geometryFromEdges(model,g);
```

Plot the geometry with the edge labels.

```
figure
pdegplot(model,"EdgeLabels","on")
axis equal
title("Geometry with Edge Labels")
```



Specify the coefficients.

```
c = 1e2;
a = 0;
f = 0;
d = 10;
specifyCoefficients(model,"m",0,"d",d,"c",c,"a",a,"f",f);
```

Specify that the solution is zero at all four outer edges of the circle.

```
bOuter = applyBoundaryCondition(model,"dirichlet","Edge",(1:4),"u",0);
```

Generate a mesh.

```
generateMesh(model,"Hmax",0.2);
```

Use `assembleFEMatrices` to calculate the global finite element mass and stiffness matrices with boundary conditions imposed using the nullspace approach.

```
FEMatrices = assembleFEMatrices(model,"nullspace");
K = FEMatrices.Kc;
B = FEMatrices.B;
M = FEMatrices.M;
```

Solve the eigenvalue problem by using the `eigs` function.

```
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);
```

Add the constraint values to get the full eigenvector.

```
eigenvectorsEigs = B*eigenvectorsEigs;
```

Now, solve the same eigenvalue problem using `solvepdeeig`. Set the range for `solvepdeeig` to be slightly larger than the range from `eigs`.

```
r = [min(eigenvaluesEigs)*0.99 max(eigenvaluesEigs)*1.01];
result = solvepdeeig(model,r);
```

```
              Basis= 10,  Time=   0.17,  New conv eig=  1
              Basis= 13,  Time=   0.19,  New conv eig=  1
              Basis= 16,  Time=   0.20,  New conv eig=  1
              Basis= 19,  Time=   0.30,  New conv eig=  2
              Basis= 22,  Time=   0.42,  New conv eig=  3
              Basis= 25,  Time=   0.45,  New conv eig=  4
              Basis= 28,  Time=   0.52,  New conv eig=  8
              Basis= 31,  Time=   0.53,  New conv eig= 10
End of sweep: Basis= 31,  Time=   0.53,  New conv eig= 10
              Basis= 20,  Time=   0.64,  New conv eig=  0
End of sweep: Basis= 20,  Time=   0.64,  New conv eig=  0
```

```
eigenvectorsPde = result.Eigenvectors;
eigenvaluesPde = result.Eigenvalues;
```

Compare the solutions.

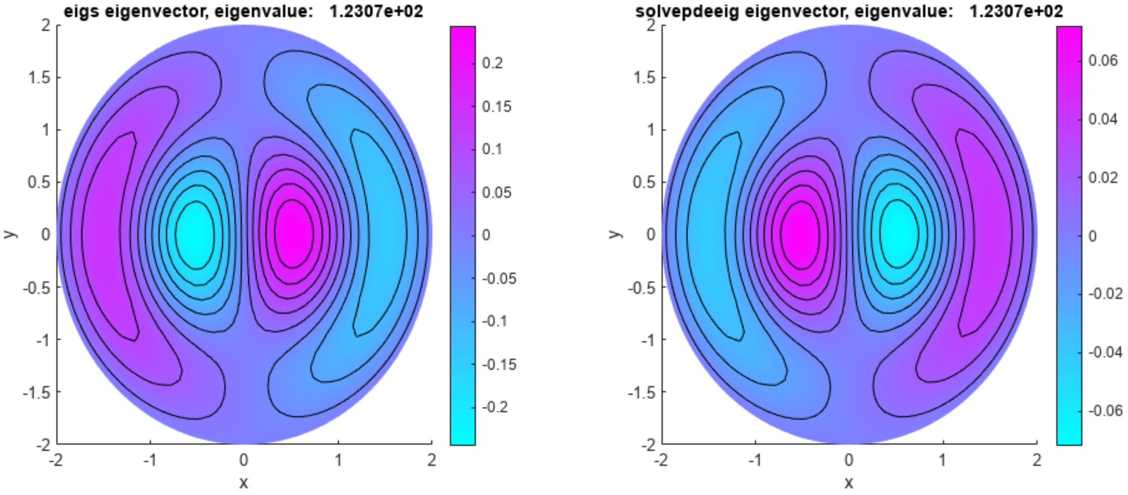
```
eigenValueDiff = sort(eigenvaluesPde) - sort(eigenvaluesEigs);
fprintf(['Max difference in eigenvalues' ...
         ' from solvepdeeig and eigs: %e\n'], ...
  norm(eigenValueDiff,inf));
```

```
Max difference in eigenvalues from solvepdeeig and eigs: 2.131628e-13
```

Both functions calculate the same eigenvalues. For any eigenvalue, you can multiply the eigenvector by an arbitrary scalar. The `eigs` and `solvepdeeig` functions might choose a different arbitrary scalar for normalizing their eigenvectors.

```
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")
```



Solution and Gradient Plots with pdeplot and pdeplot3D

2-D Solution and Gradient Plots

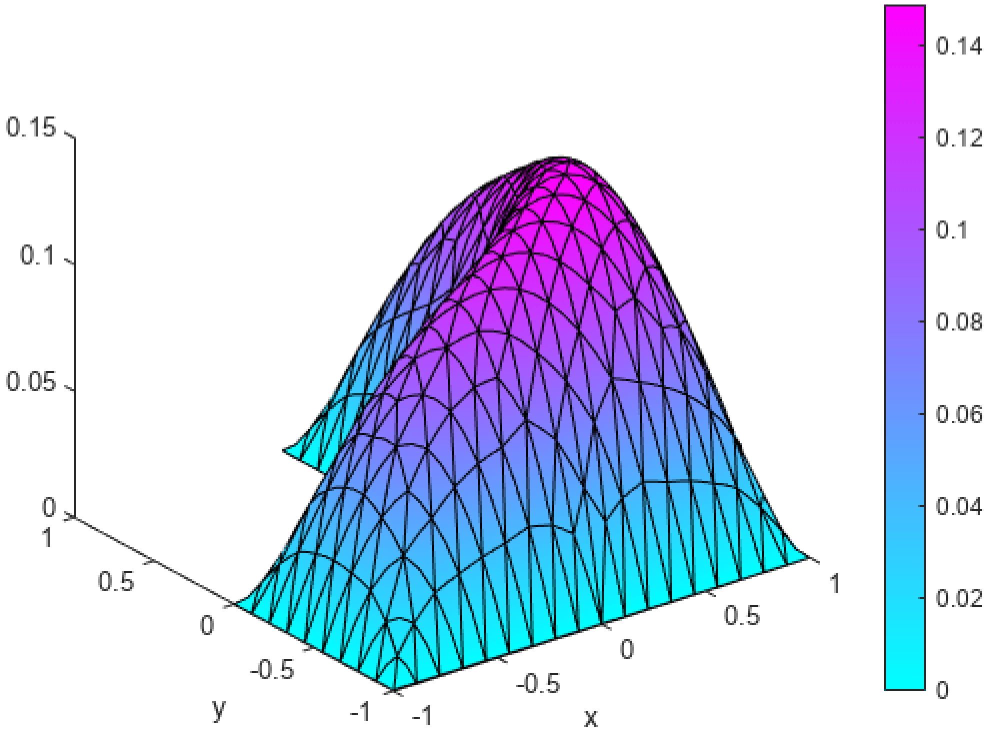
To visualize a 2-D scalar PDE solution, you can use the `pdeplot` function. This function lets you plot the solution without explicitly interpolating the solution. For example, solve the scalar elliptic problem $-\nabla u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions and plot 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);
```

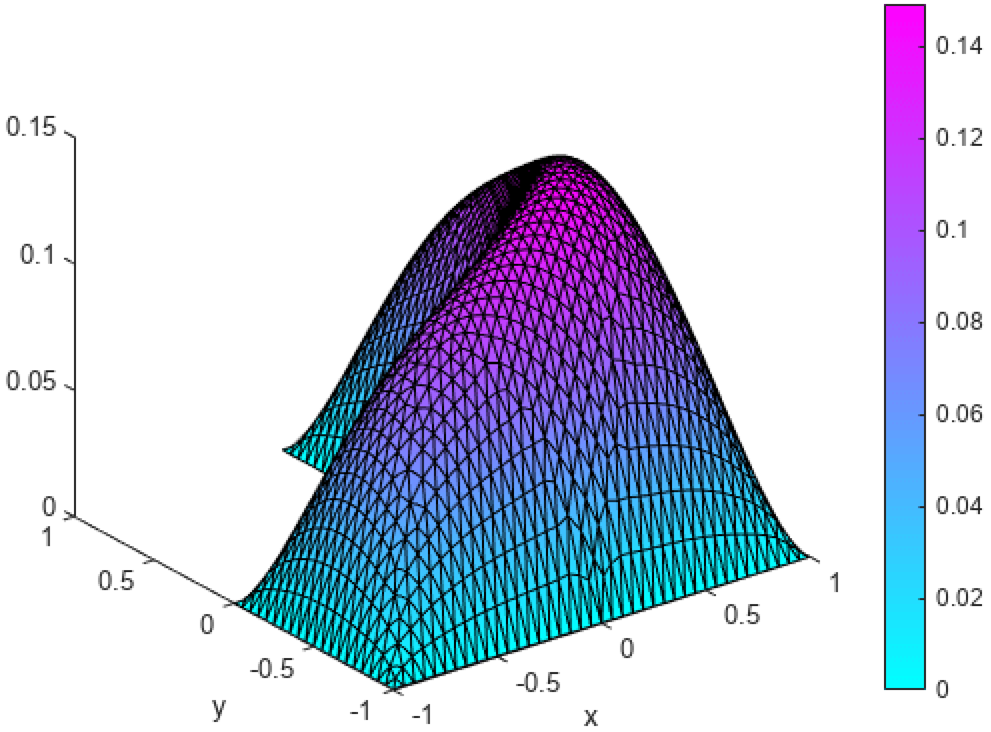
Use `pdeplot` to plot the solution.

```
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")
```

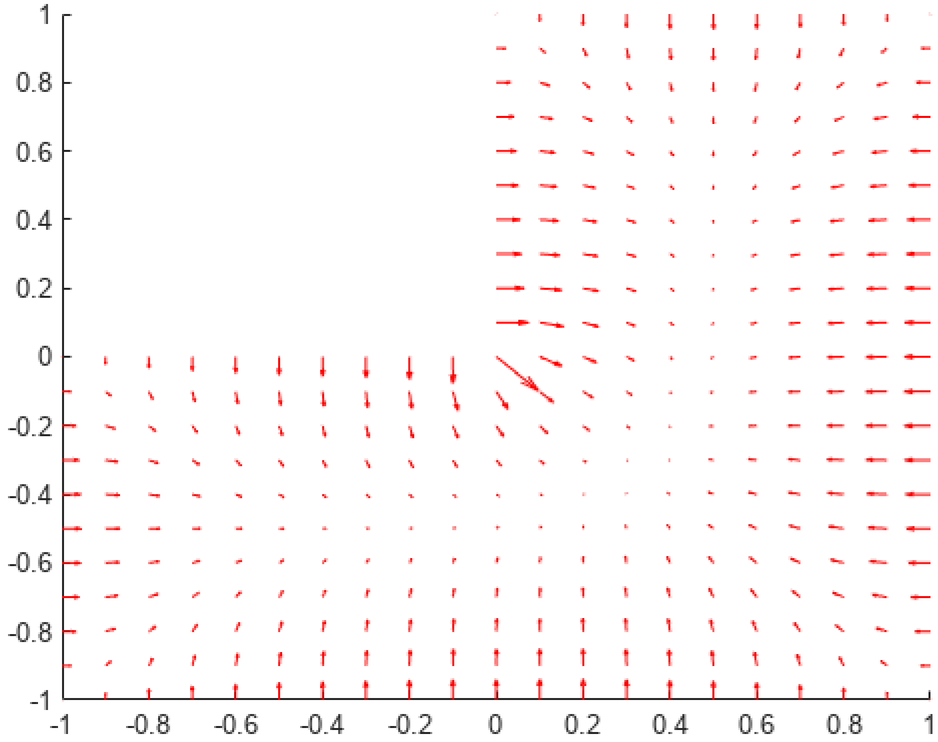


Access the gradient of the solution at the nodal locations.

```
ux = results.XGradients;
uy = results.YGradients;
```

Plot the gradient as a quiver plot.

```
pdeplot(model,"FlowData",[ux,uy])
```

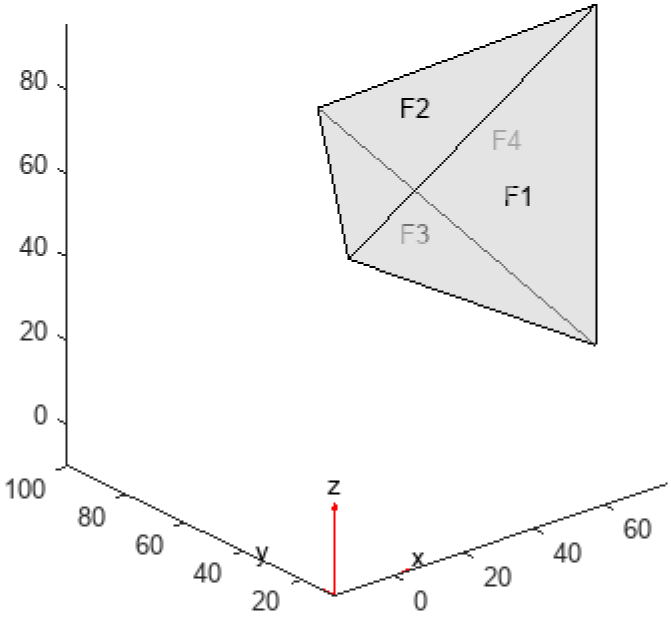


3-D Surface and Gradient Plots

Obtain a surface plot of a solution with 3-D geometry and N > 1.

First, 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.

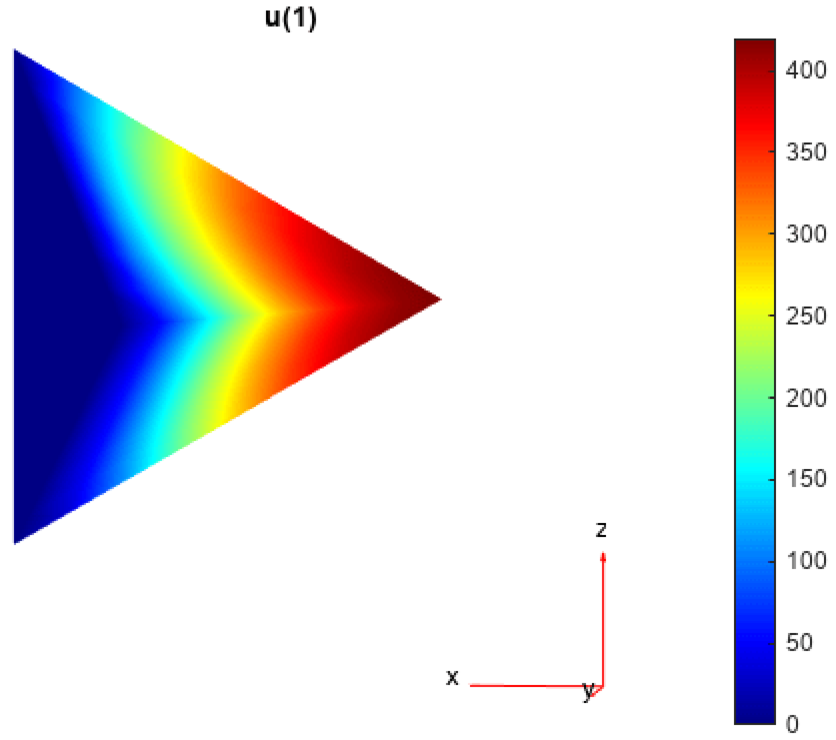
```
generateMesh(model,"Hmax",20);
```

Solve the problem.

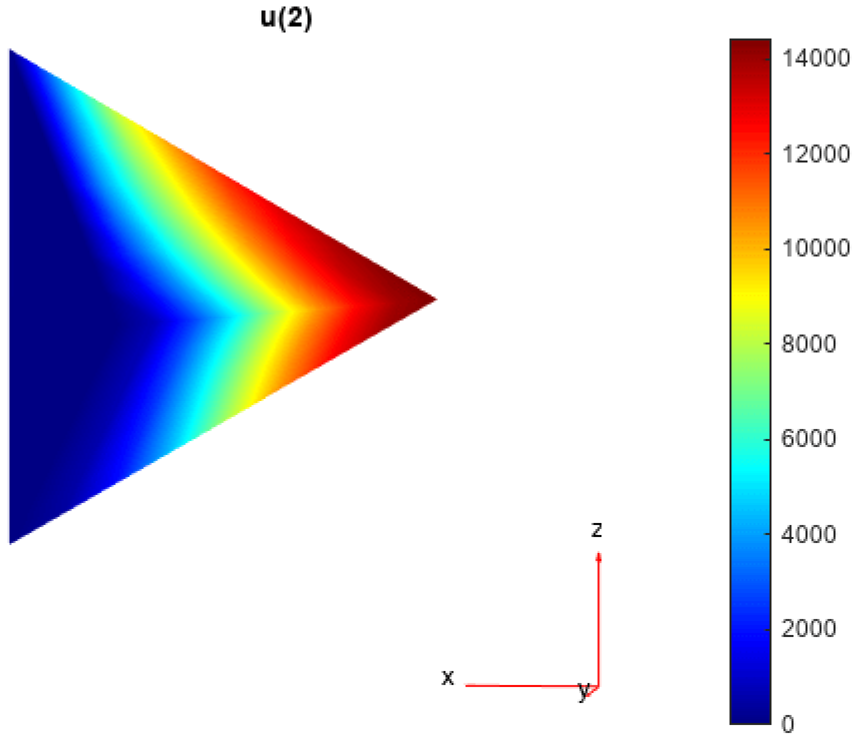
```
results = solvepde(model);
u = results.NodalSolution;
```

Plot the two components of the solution.

```
pdeplot3D(model,"ColorMapData",u(:,1))
view(-175,4)
title("u(1)")
```

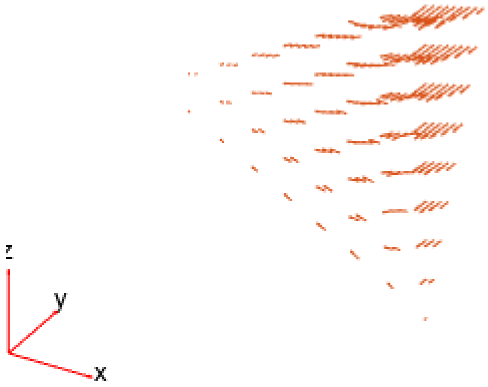


```
figure
pdeplot3D(model,"ColorMapData",u(:,2))
view(-175,4)
title("u(2)")
```

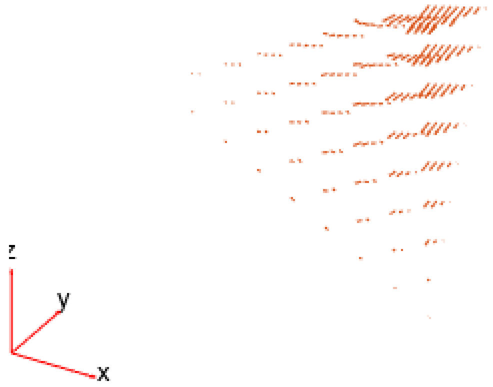


Compute the flux of the solution and plot the results for both components.

```
[cgradx,cgrady,cgradz] = evaluateCGradient(results);
figure
pdeplot3D(model,"FlowData",[cgradx(:,1) cgrady(:,1) cgradz(:,1)])
```



```
figure
pdeplot3D(model,"FlowData",[cgradx(:,2) cgrady(:,2) cgradz(:,2)])
```


z
y
x

2-D Solution and Gradient Plots with MATLAB Functions

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.

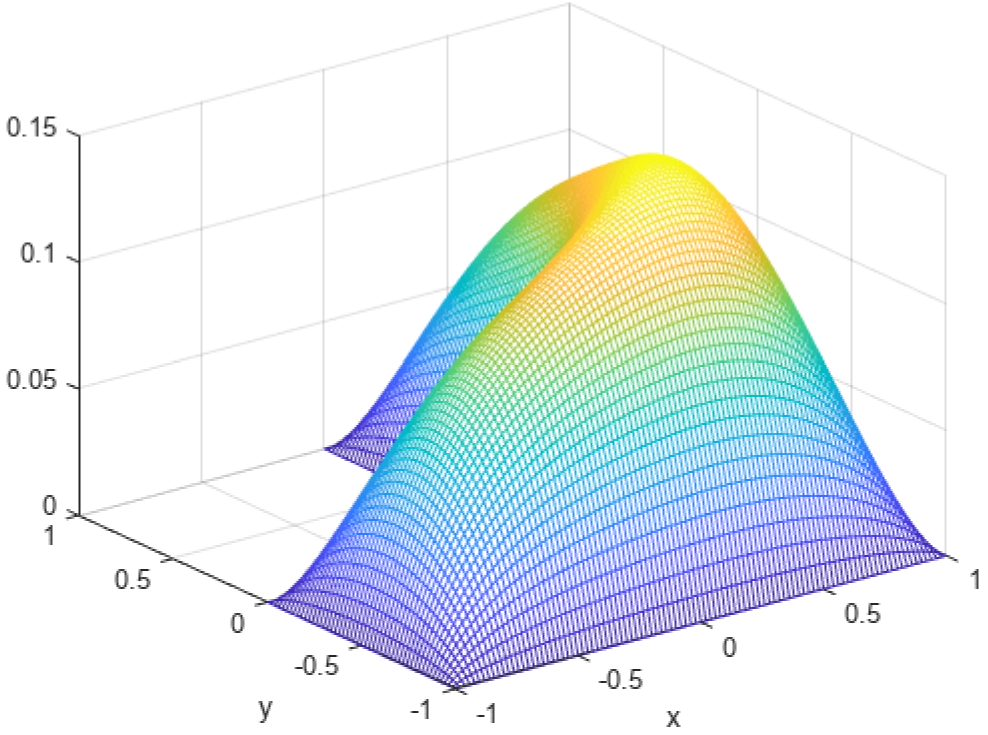
```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1:model.Geometry.NumEdges, ...
                             "u",0);
c = 1;
a = 0;
f = 1;
specifyCoefficients(model,"m",0,"d",0,"c",c,"a",a,"f",f);
generateMesh(model,"Hmax",0.05);
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")
```

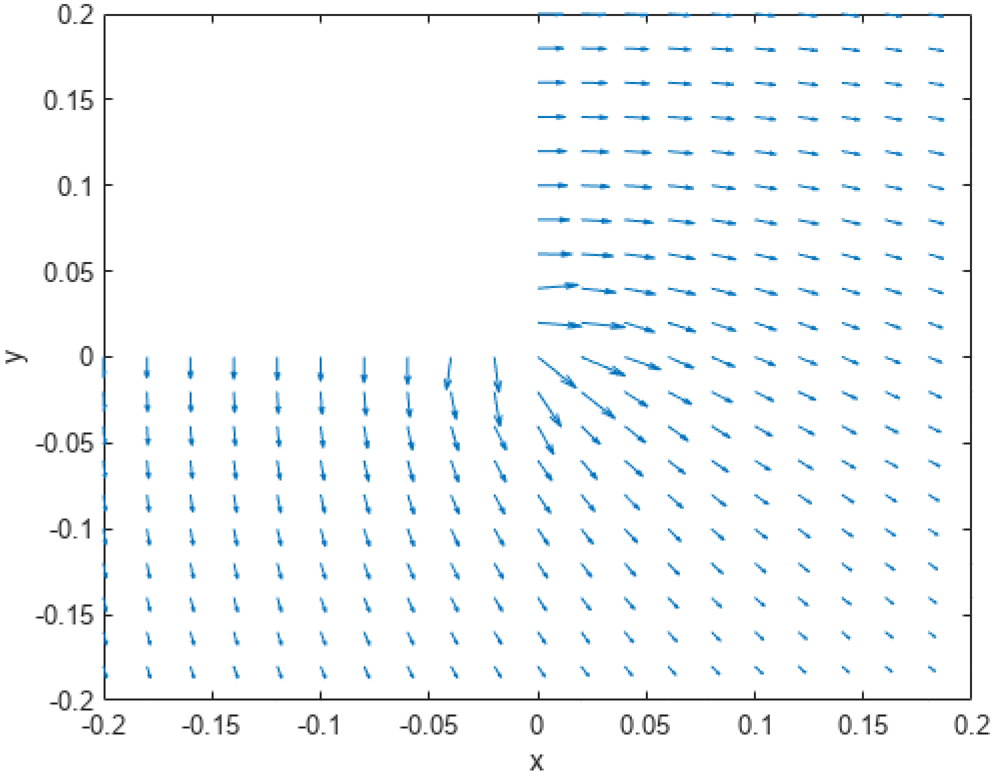


Interpolate gradients of the solution to the grid from -1 to 1 in each direction. Plot the result using `quiver`.

```
[gradx,grady] = evaluateGradient(results,querypoints);
figure
quiver(X(:),Y(:),gradx,grady)
xlabel("x")
ylabel("y")
```

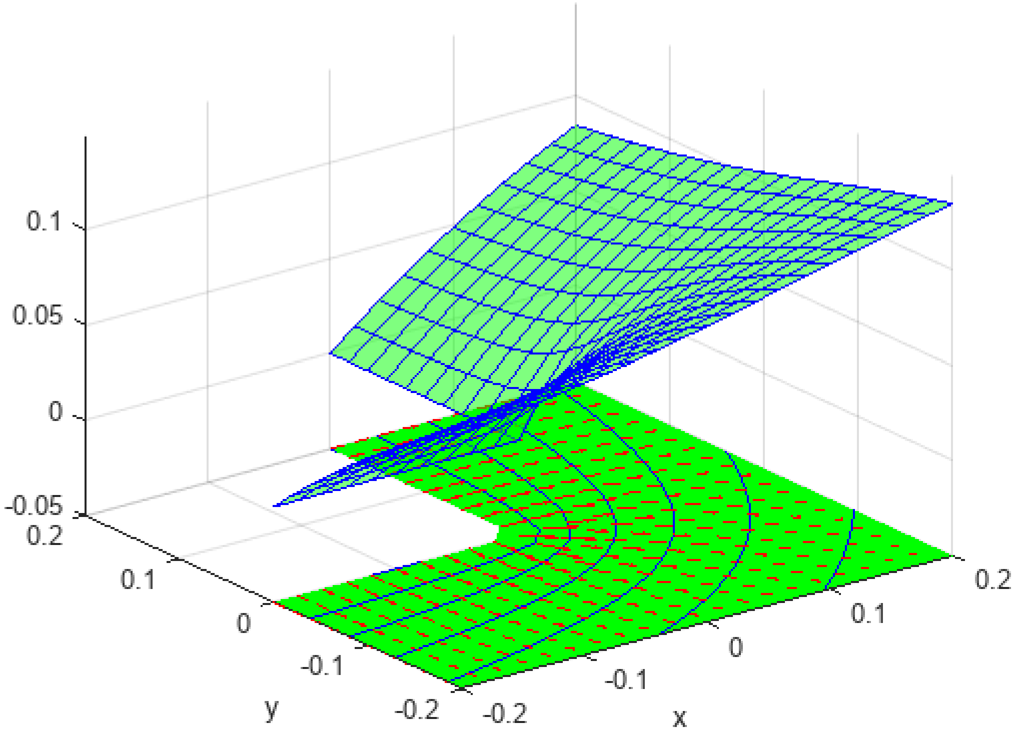
Zoom in to see more details. For example, restrict the range to `[-0.2,0.2]` in each direction.

```
axis([-0.2 0.2 -0.2 0.2])
```



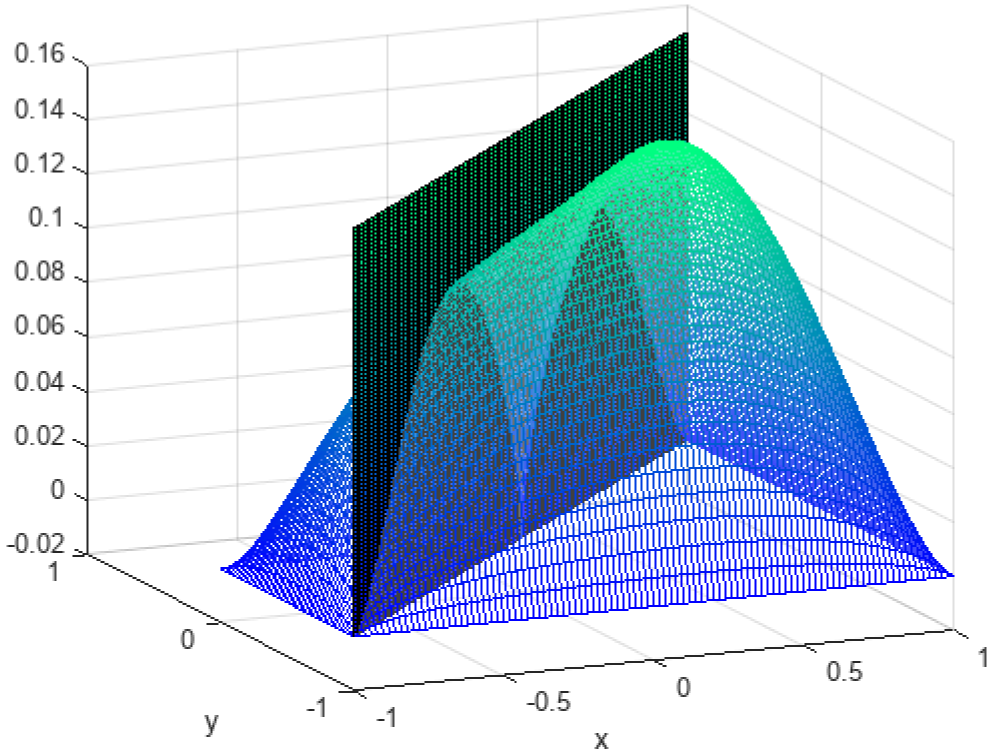
Plot the solution and the gradients on the same range.

```
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
```

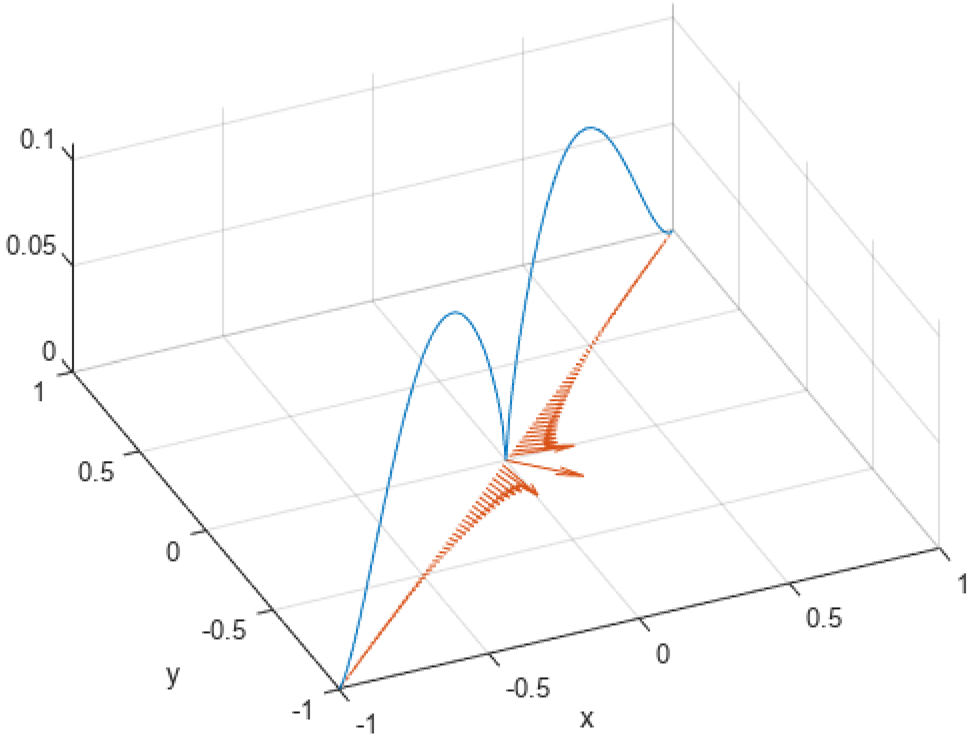


Plot the interpolated solution along the line.

```
figure
xq = v;
yq = v;
uintrp = interpolateSolution(results,xq,yq);

plot3(xq,yq,uintrp)
grid on
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])
```


0.1
0.05
0
1
0.5
0
-0.5
y
-1
-1
-0.5
0
0.5
1
x

3-D Solution and Gradient Plots with MATLAB Functions

Types of 3-D Solution Plots Available in MATLAB

In addition to surface and gradient plots available with the PDE plotting functions, you can use MATLAB graphics capabilities to create more types of plots for your 3-D model.

- 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 3-344 and “Contour Slices Through 3-D Solution” on page 3-347. 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 3-351.
- You can use any MATLAB plotting command to create 3-D plots. See “Techniques for Visualizing Scalar Volume Data” and “Visualizing Vector Volume Data”.

2-D Slices Through 3-D Geometry

This example shows how to obtain plots from 2-D slices through a 3-D geometry.

The problem is

$$\frac{\partial u}{\partial t} - \Delta u = f$$

on a 3-D slab with dimensions 10-by-10-by-1, where $u = 0$ at time `t = 0`, boundary conditions are Dirichlet, and

$$f(x,y,z) = 1 + y + 10z^2$$

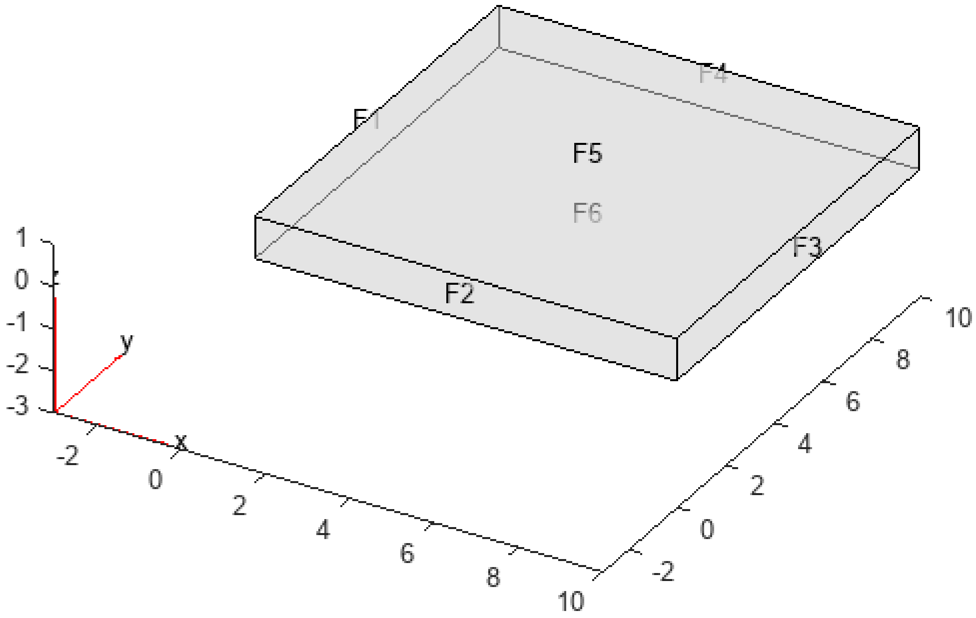
Set Up and Solve the PDE

Define a function for the nonlinear `f` coefficient in the syntax as given in “f Coefficient for specifyCoefficients” on page 2-90.

```
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.

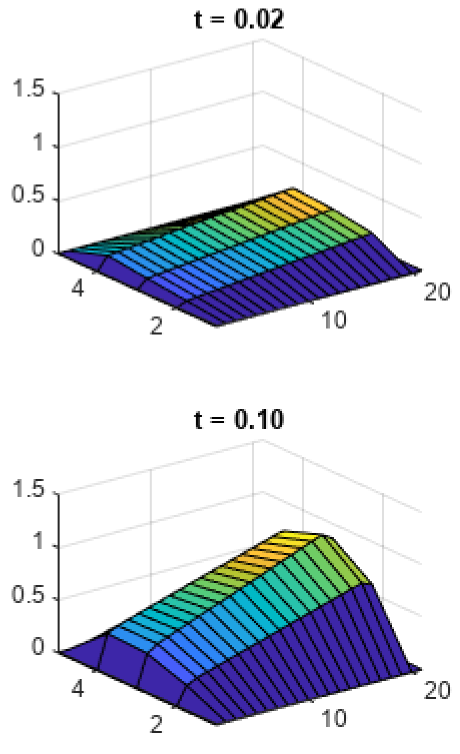
```
yy = 0:0.5:10;
zz = 0:0.25:1;
[YY,ZZ] = meshgrid(yy,zz);
XX = 5*ones(size(YY));
uintrp = interpolateSolution(results,XX,YY,ZZ,1:length(tlist));
```

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 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

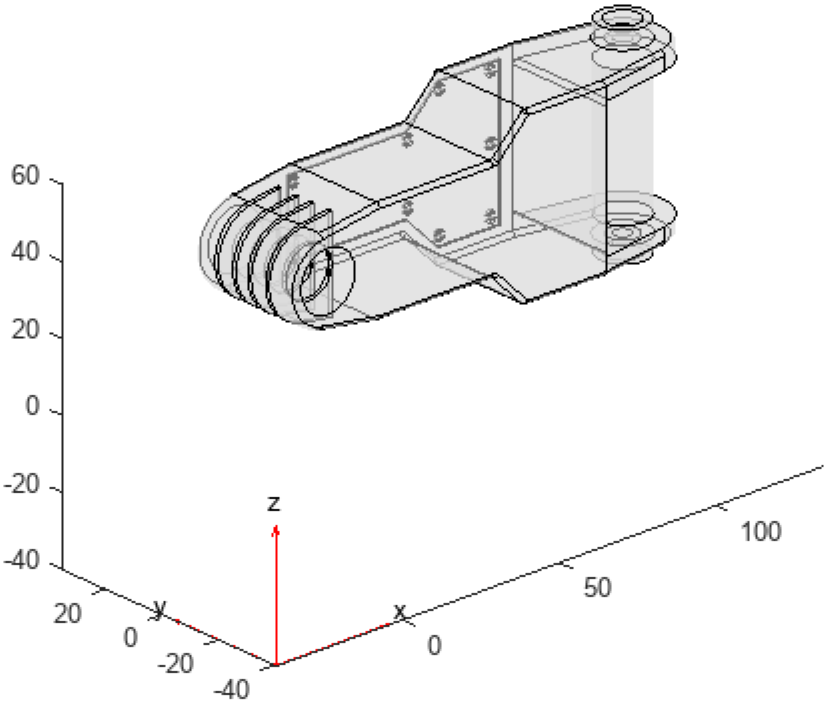
$$-\nabla \cdot \nabla (cu) + au = f .$$

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.

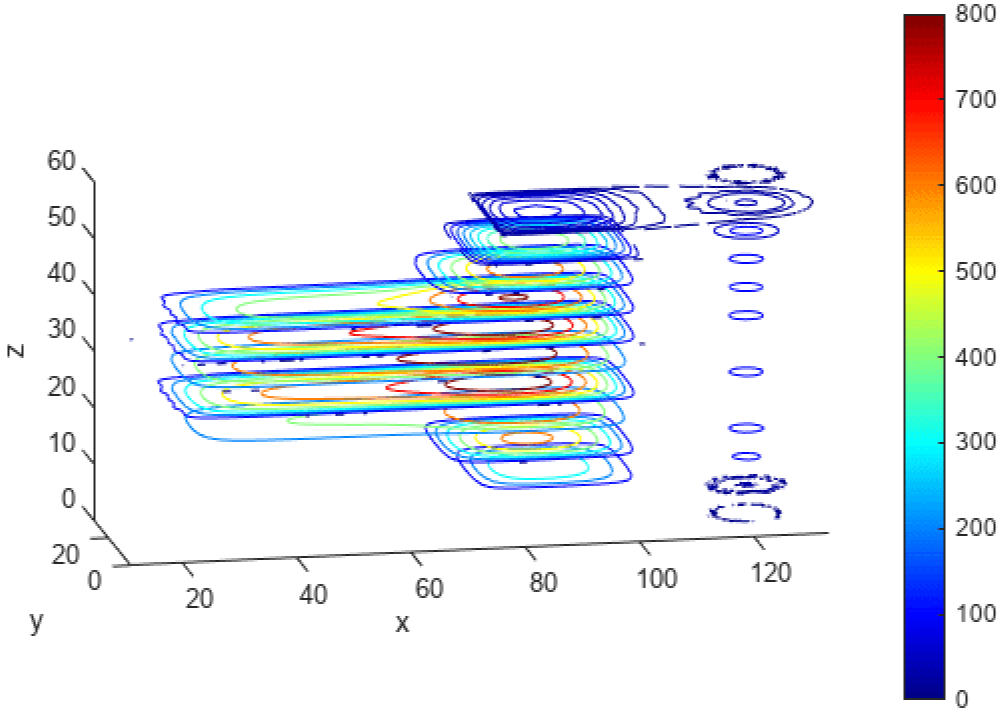
```
[X,Y,Z] = meshgrid(0:135,0:35,0:61);
```

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*.

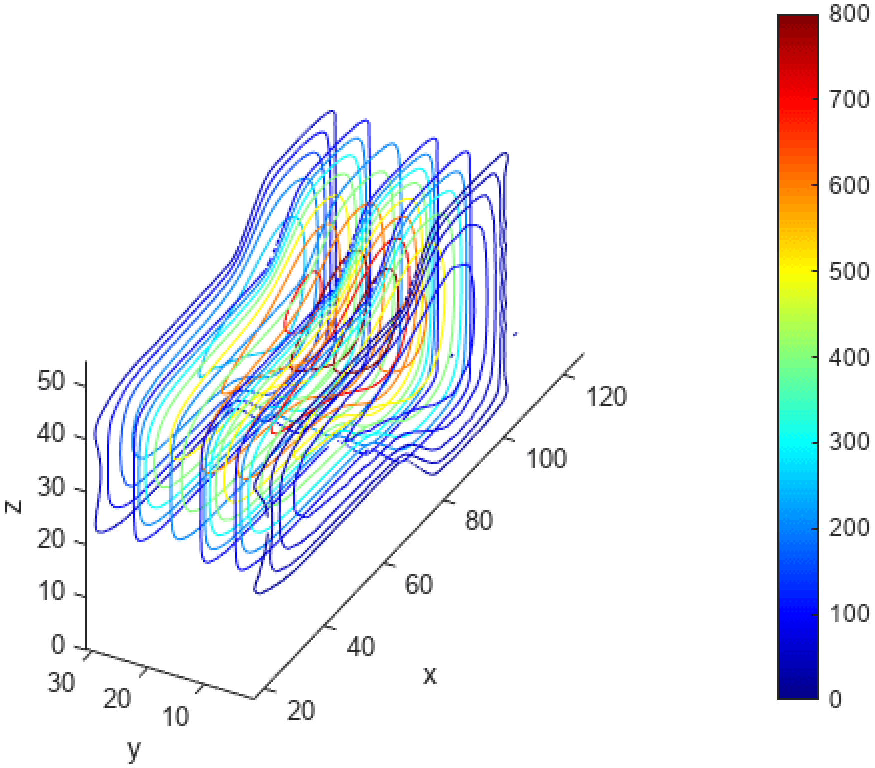
```
figure
colormap jet
contourslice(X,Y,Z,V,[],[],0:5:60)
xlabel("x")
ylabel("y")
zlabel("z")
colorbar
view(-11,14)
axis equal
```



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
```



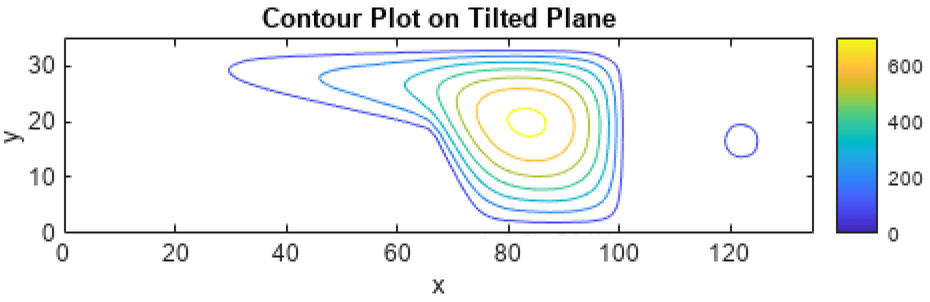
Save Memory by Evaluating As Needed

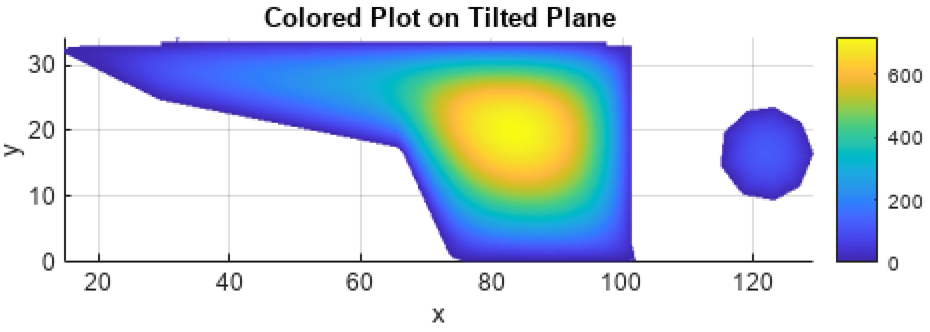
For large problems you can run out of memory when creating a fine 3-D grid. Furthermore, it can be time-consuming to evaluate the solution on a full grid. To save memory and time, evaluate only at the points you plot. You can also use this technique to interpolate to tilted grids, or to other surfaces.

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
```





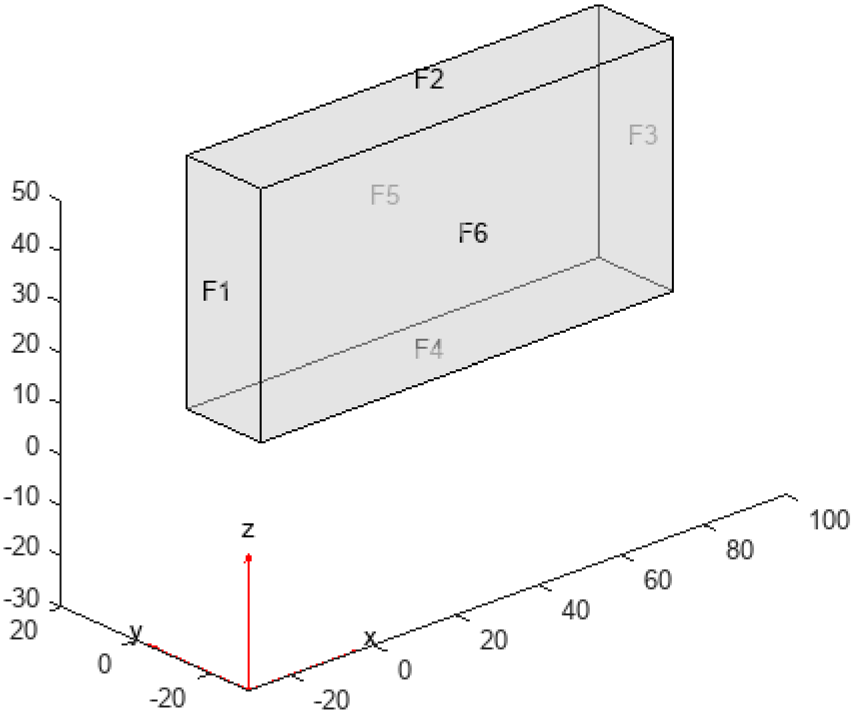
Plots of Gradients and Streamlines

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);
```

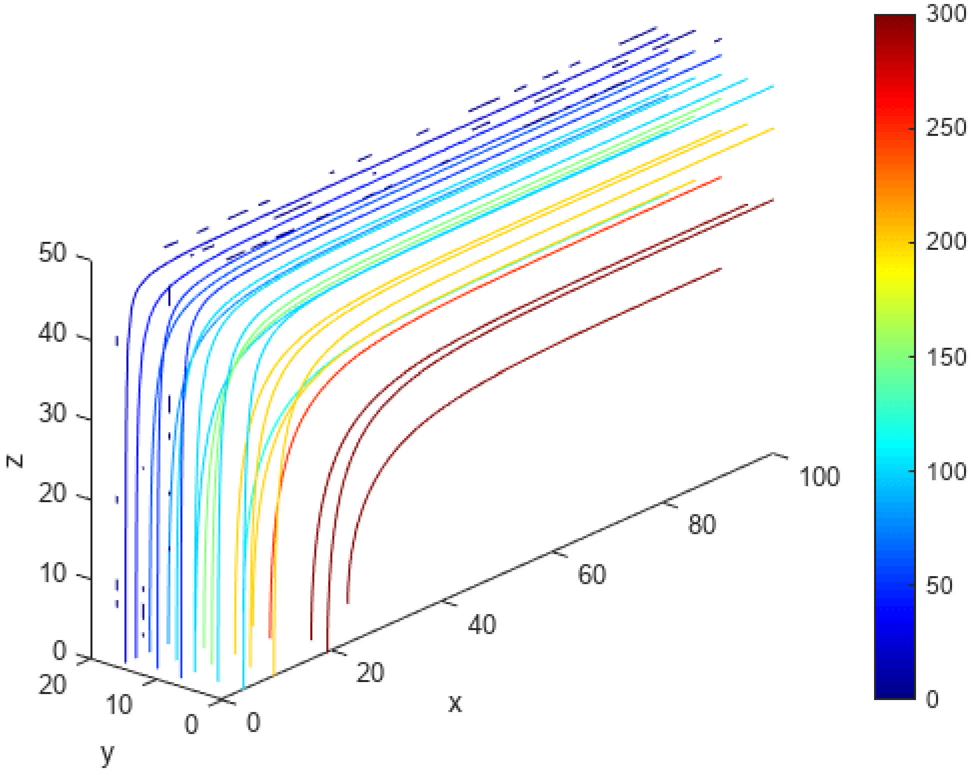
Examine the Solution in a Contour Slice Plot

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 $(x, y, z) = (100, 0, 0)$.

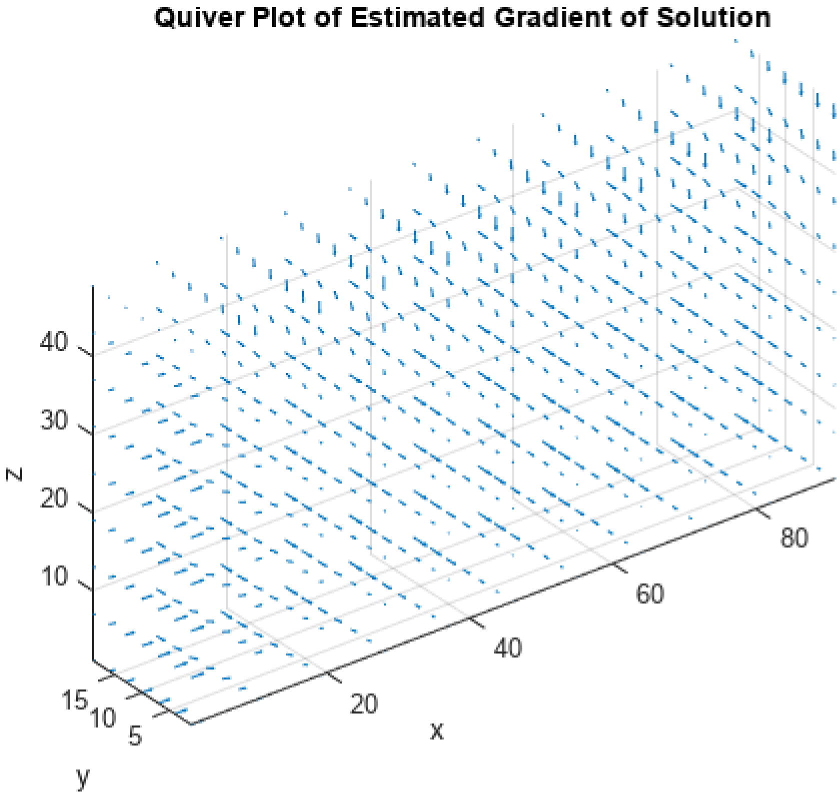
Use Gradients for Quiver and Streamline Plots

Examine the solution in more detail by evaluating the gradient of the solution. Use a rather coarse mesh so that you can see the details on the quiver and streamline plots.

```
[X,Y,Z] = meshgrid(1:9:99,1:3:20,1:6:50);
[gradx,grady,gradz] = evaluateGradient(results,X,Y,Z);
```

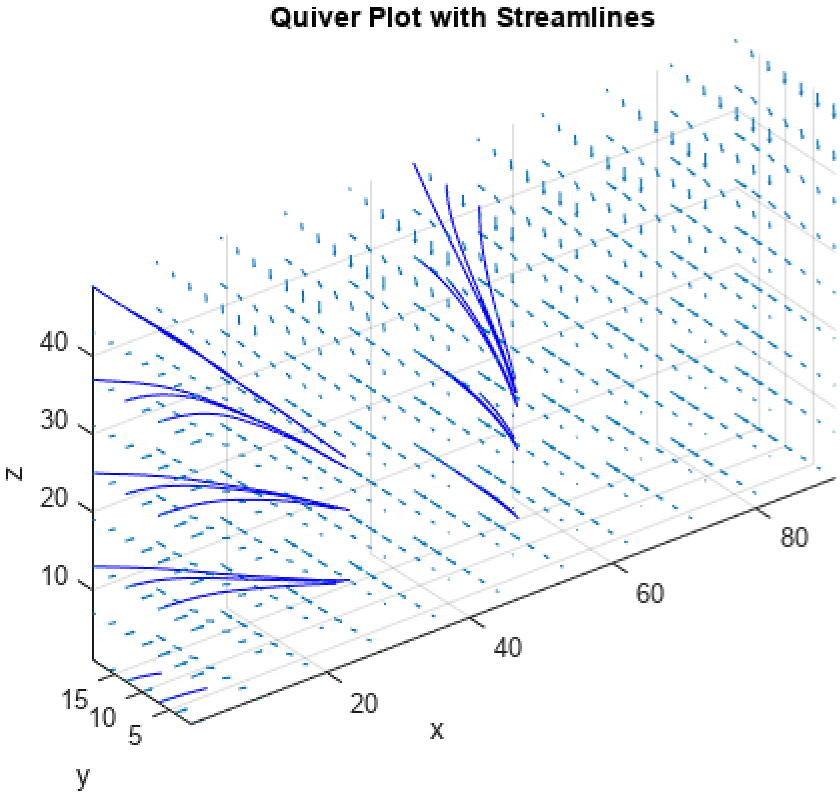
Plot the gradient vectors. First reshape the approximate gradients to the shape of the mesh.

```
gradx = reshape(gradx,size(X));
grady = reshape(grady,size(Y));
gradz = reshape(gradz,size(Z));

figure
quiver3(X,Y,Z,gradx,grady,gradz)
axis equal
xlabel("x")
ylabel("y")
zlabel("z")
title("Quiver Plot of Estimated Gradient of Solution")
```



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.

Dimensions of Solutions, Gradients, and Fluxes

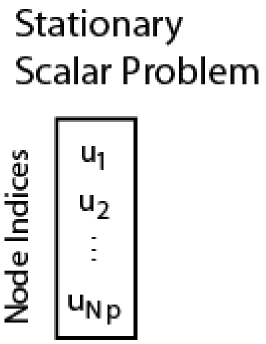
`solvepde` returns a `StationaryResults` or `TimeDependentResults` object whose properties contain the solution and its gradient at the mesh nodes. You can interpolate the solution and its gradient to other points in the geometry by using `interpolateSolution` and `evaluateGradient`. You also can compute flux of the solution at the mesh nodes and at arbitrary points by using `evaluateCGradient`.

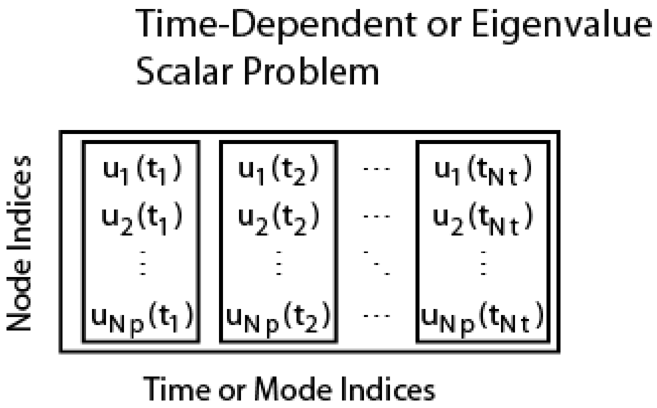
Note `solvepde` does not compute components of flux of a PDE solution. To compute flux of the solution at the mesh nodes, use `evaluateCGradient`.

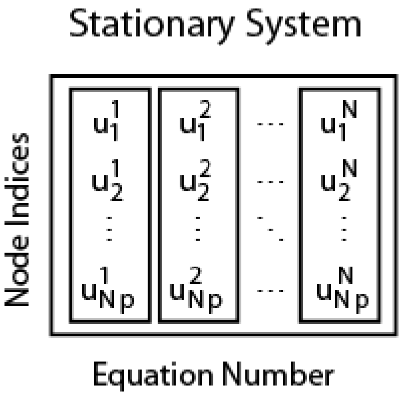
`solvepdeeig` returns an `EigenResults` object whose properties contain the solution eigenvectors calculated at the mesh nodes. You can interpolate the solution to other points by using `interpolateSolution`.

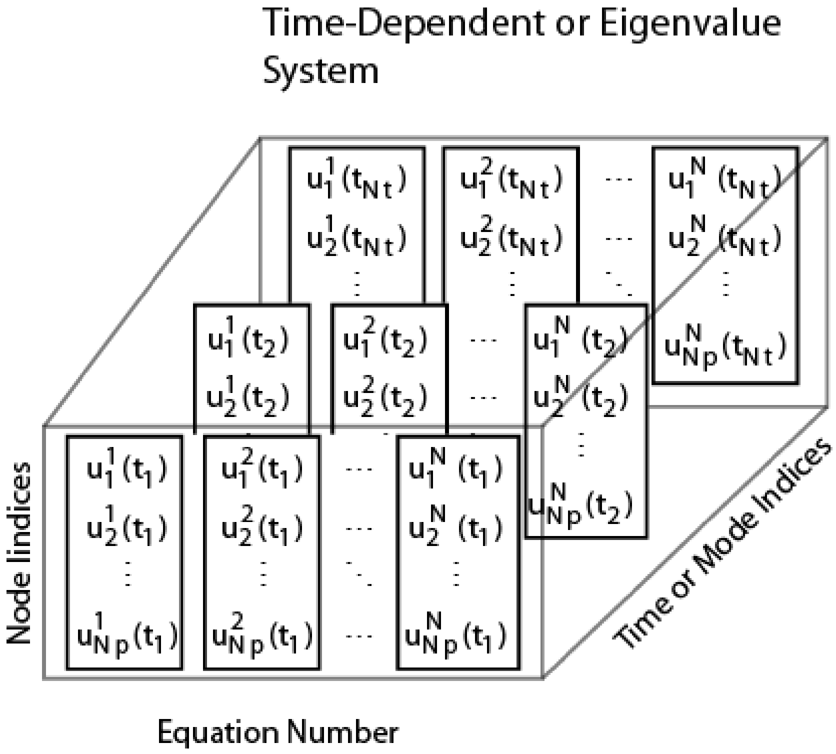
The dimensions of the solution, its gradient, and flux of the solution depend on:

- 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.









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 `evaluateCGradient`, 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)`
	`result.XGradients(iP)`
	`result.YGradients(iP)`
	`result.ZGradients(iP)`
		Np-by-1	Nqp-by-1

System, N > 1	`result.NodalSolution(iP,iN)`
	`result.XGradients(iP,iN)`
	`result.YGradients(iP,iN)`
	`result.ZGradients(iP,iN)`
		Np-by-N	Nqp-by-N

Time-dependent PDE problem	**Access solution and components of gradient**	**Size of `NodalSolution`, `XGradients`, `YGradients`, `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`

See Also

`solvepde` | `solvepdeeig` | `interpolateSolution` | `evaluateGradient` | `StationaryResults` | `TimeDependentResults` | `EigenResults`

4

PDE Modeler App

You open the PDE Modeler app by entering `pdeModeler` at the command line. The main components of the PDE Modeler app are the menus, the dialog boxes, and the toolbar.

Open the PDE Modeler App

You can open the PDE Modeler 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 Modeler app window, type `pdeModeler` in the MATLAB Command Window.
- To open the PDE Modeler app with a circle already drawn in it, type `pdecirc` in the MATLAB Command Window.
- To open the PDE Modeler app with an ellipse already drawn in it, type `pdeellip` in the MATLAB Command Window.
- To open the PDE Modeler app with a rectangle already drawn in it, type `pderect` in the MATLAB Command Window.
- To open the PDE Modeler 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)
```

2-D Geometry Creation in PDE Modeler App

Create Basic Shapes

The PDE Modeler 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. 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. 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. 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. 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. 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:

- `pdecirc` draws a circle.
- `pdeellip` draws an ellipse.
- `pderect` draws a rectangle.
- `pdepoly` draws a polygon.

These commands open the PDE Modeler 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.

You can use a sequence of drawing commands to create several basic shapes. For example, these 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)
```

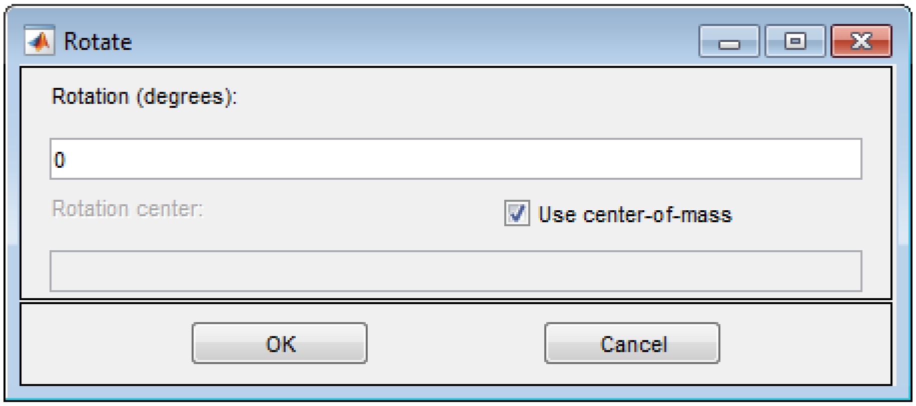
Select Several Shapes

- 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). 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.



Rotation (degrees): 0

Rotation center:

☑ Use center-of-mass

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]`.

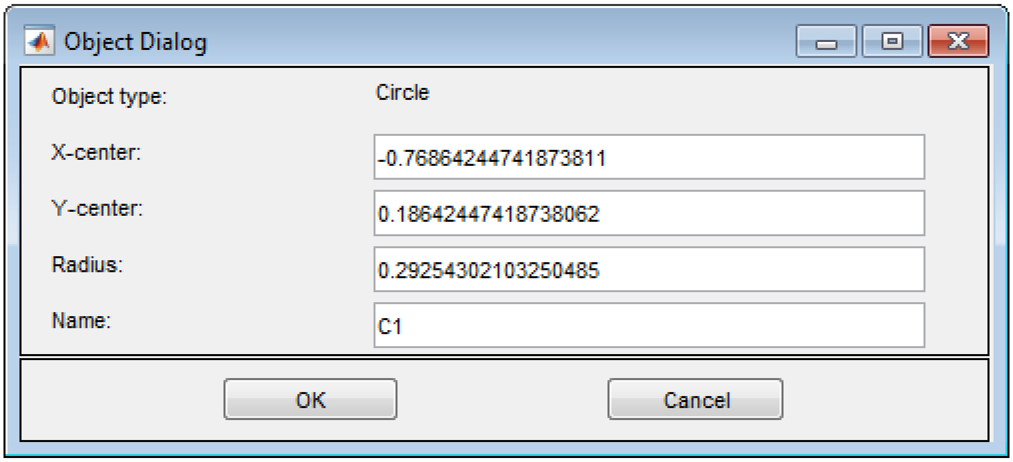
Create Complex Geometries

You can specify complex geometries by overlapping basic shapes. This approach is called Constructive Solid Geometry (CSG). The PDE Modeler 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, first switch to the draw mode and then double-click the shape. (Select **Draw Mode** from the **Draw** menu to switch to the draw mode.) The resulting dialog box lets you change the name and parameters of the selected shape. The name cannot contain spaces.



Object type: Circle

X-center: -0.76864244741873811

Y-center: 0.18642447418738062

Radius: 0.29254302103250485

Name: C1

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.

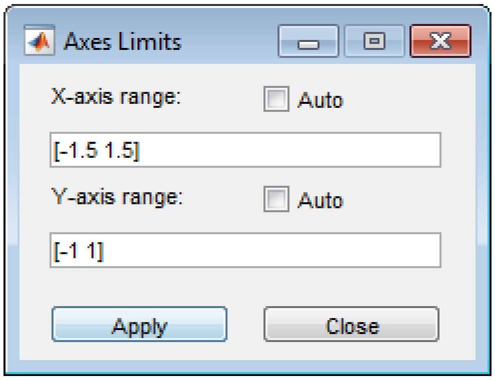
  For example, `SQ1-C2` creates a geometry comprised of the points of the square `SQ1` that do not belong to the circle `C2`.

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.



X-axis range: ☐ Auto

[-1.5 1.5]

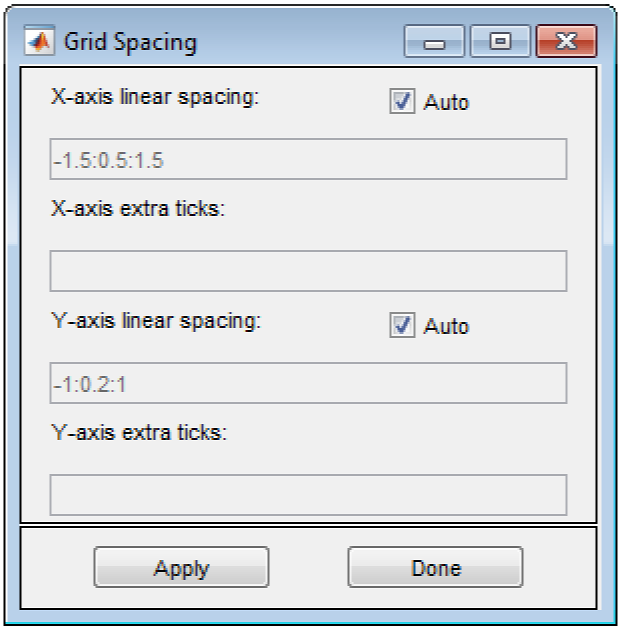
Y-axis range: ☐ Auto

[-1 1]

- Apply the specified axes ranges by clicking **Apply**.
- Close the dialog box by clicking **Close**.

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**.



X-axis linear spacing: ☑ Auto

-1.5:0.5:1.5

X-axis extra ticks:

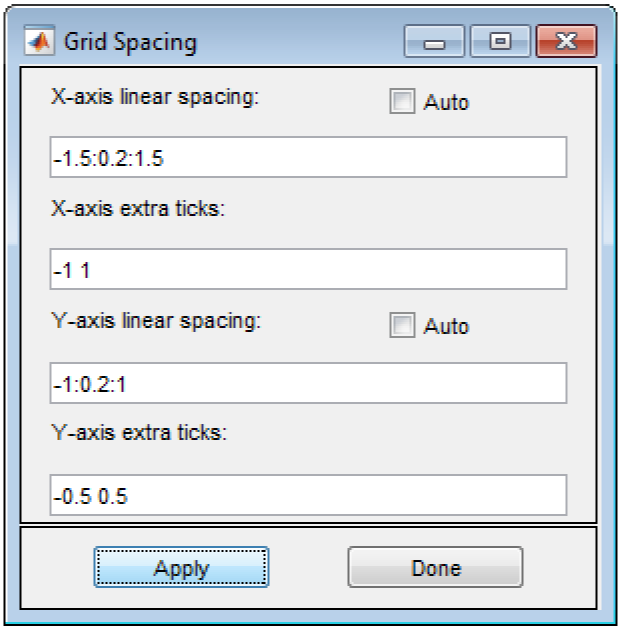
Y-axis linear spacing: ☑ Auto

-1:0.2:1

Y-axis extra ticks:

- 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.



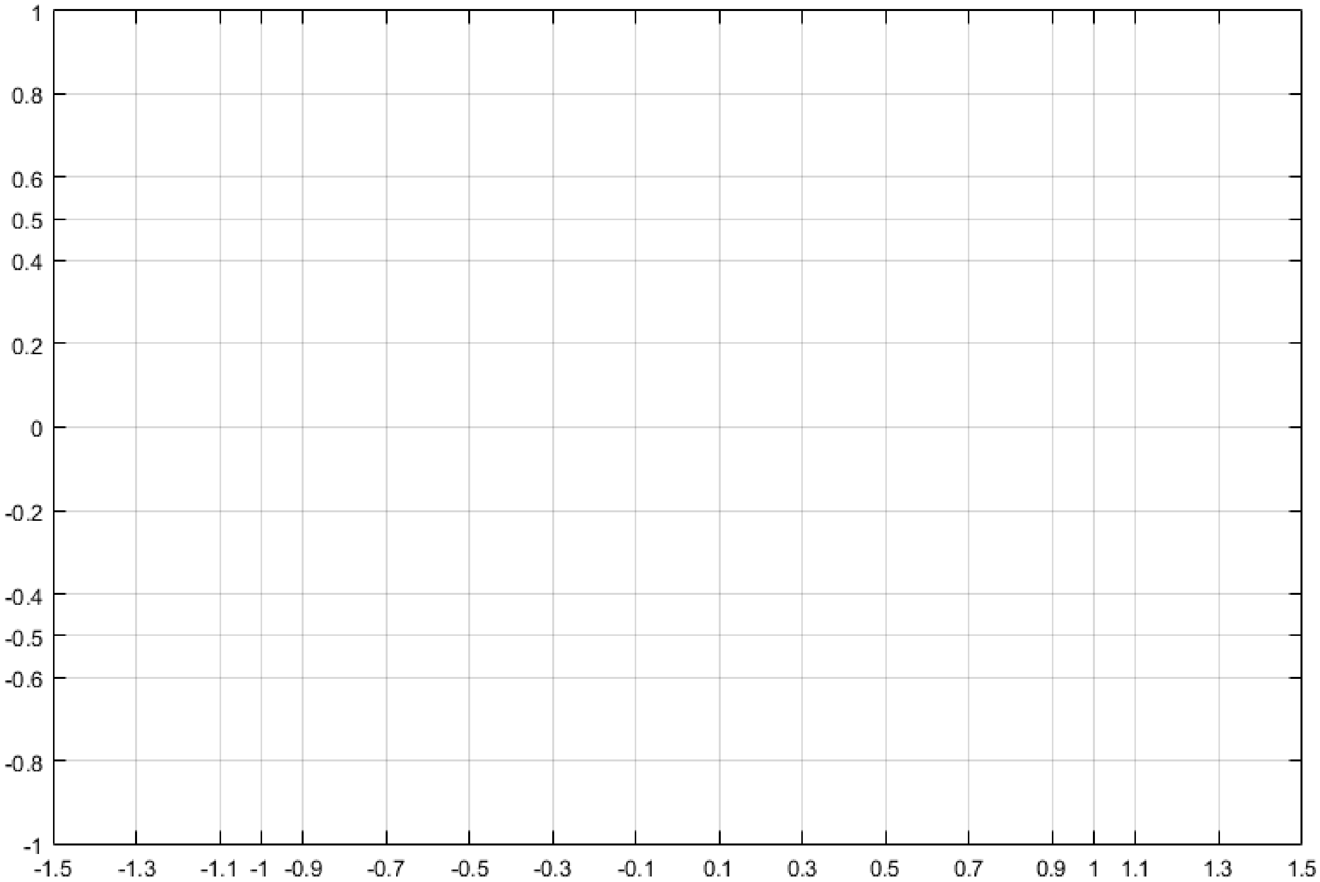
X-axis linear spacing: ☐ Auto
-1.5:0.2:1.5

X-axis extra ticks:
-1 1

Y-axis linear spacing: ☐ Auto
-1:0.2:1

Y-axis extra ticks:
-0.5 0.5

- Apply the specified grid spacing by clicking **Apply**.
- Close the dialog box by clicking **Done**.

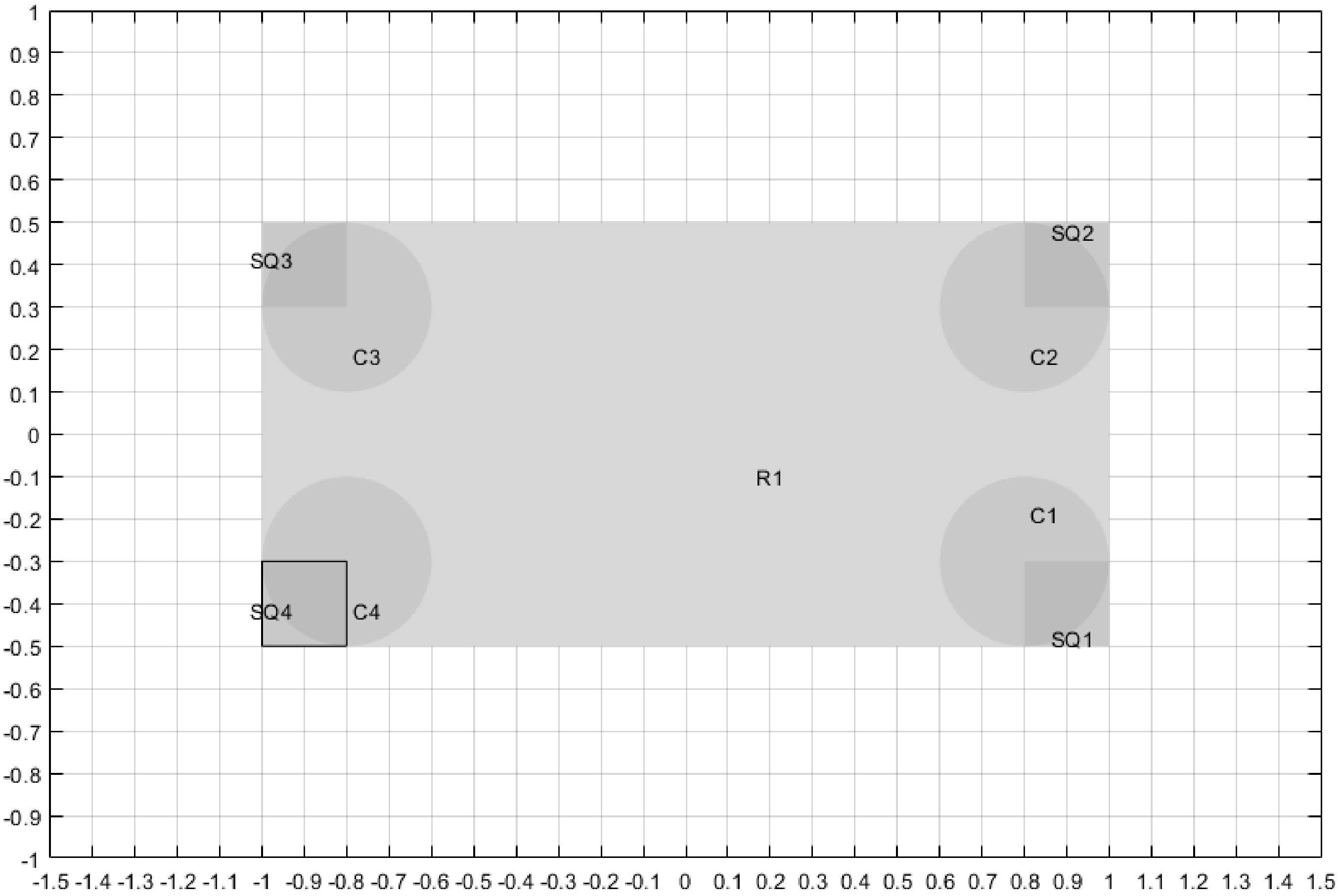


Create Geometry with Rounded Corners

1. Open the PDE Modeler app by using the `pdeModeler` command.
2. Display grid lines by selecting **Options** > **Grid**.
3. Align new shapes to the grid lines by selecting **Options** > **Snap**.
4. Set the grid spacing for *x*-axis to `-1.5:0.1:1.5` and for *y*-axis to `-1:0.1:1`. To do this, select **Options** > **Grid Spacing**, clear the **Auto** checkboxes, and set the corresponding ranges.
5. Draw a rectangle with the width 2, the height 1, and the top left corner at (–1,0.5). To do this, first click the ▭ button. Then click the point (–1,0.5) and drag to draw a rectangle.

   To edit the parameters of the rectangle, double-click it. In the resulting dialog box, specify the exact parameters.
6. Draw four circles with the radius 0.2 and the centers at (–0.8,–0.3), (–0.8,0.3), (0.8,–0.3), and (0.8,0.3).To do this, first click the ⊕ button. Then click the center of a circle 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, then double-click it. In the resulting dialog box, specify the exact center location and radius of the circle.

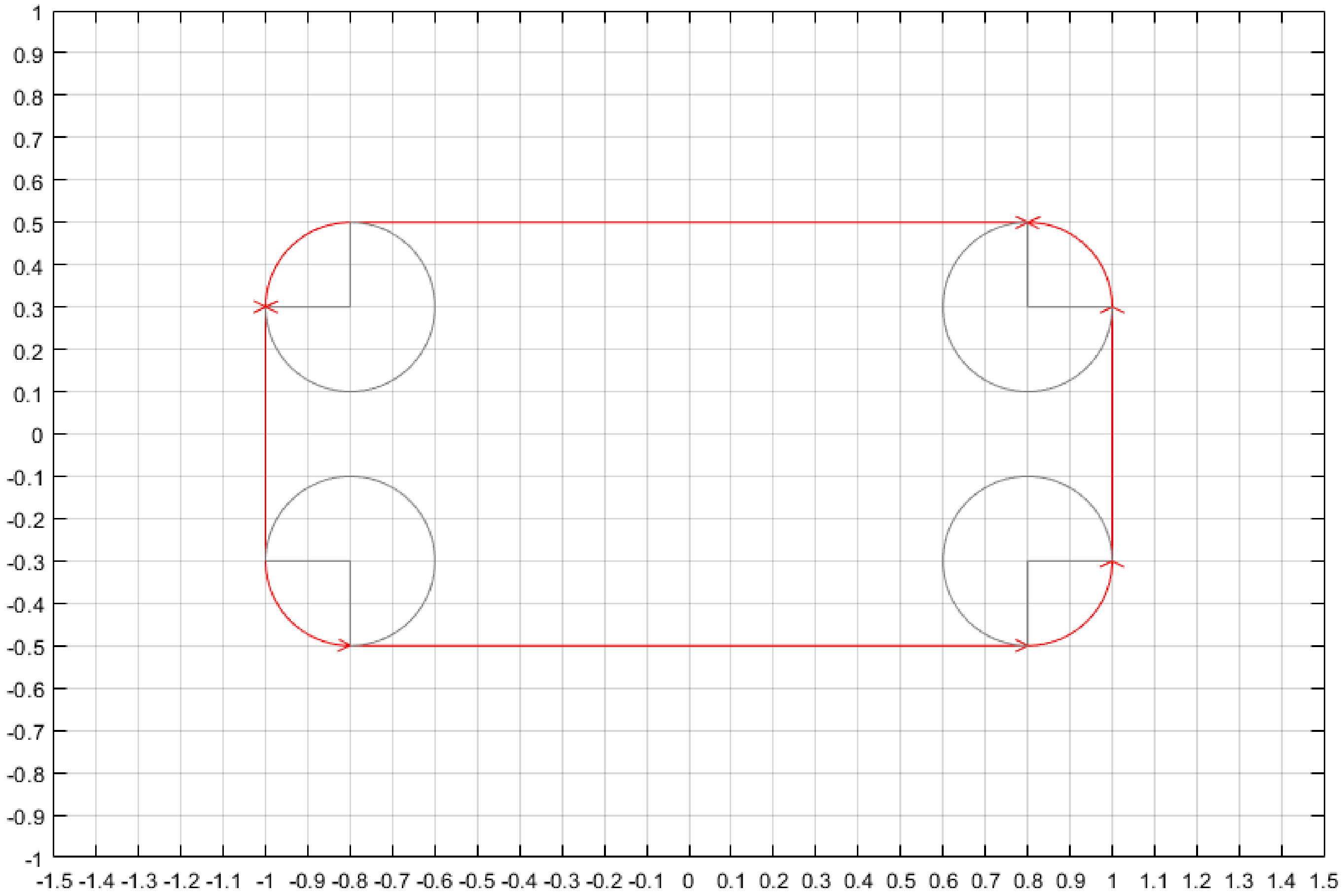
7 Add four squares with the side 0.2, one in each corner.



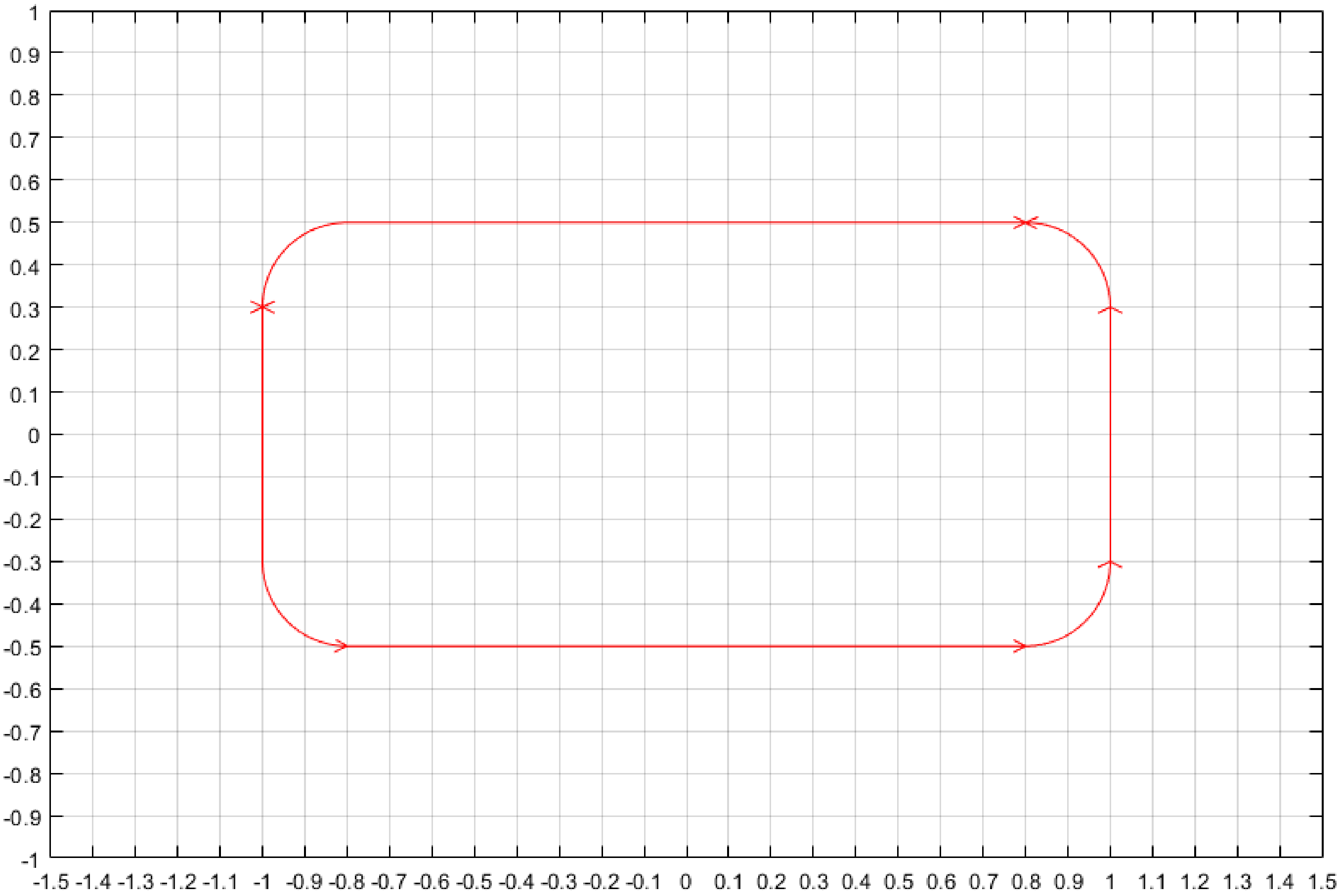
8 Model the geometry with rounded corners by subtracting the small squares from the rectangle, and then adding the circles. To do this, enter the following formula in the **Set formula** field.

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

9 Switch to the boundary mode by clicking the ∂Ω button or selecting **Boundary** > **Boundary Mode**. The CSG model is now decomposed using the set formula, and you get a rectangle with rounded corners.



10 Because of the intersection of the solid objects used in the initial CSG model, a number of subdomain borders remain. They appear as gray lines. To remove these borders, select **Boundary > Remove All Subdomain Borders**.

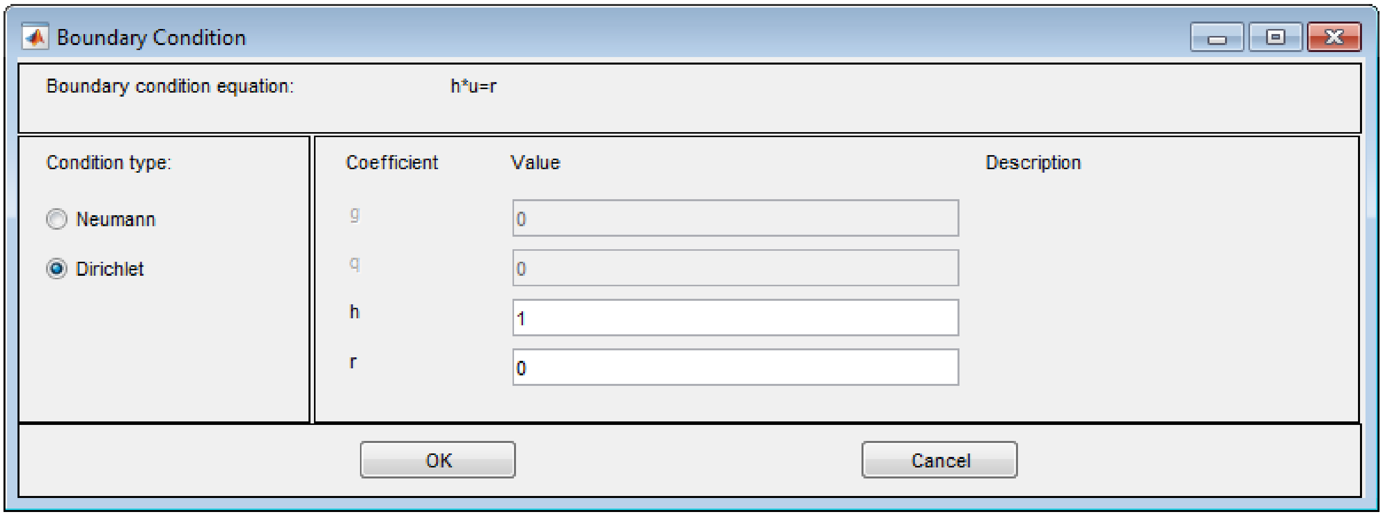

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
-1
-1.5 -1.4 -1.3 -1.2 -1.1 -1 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 1.2 1.3 1.4 1.5

Specify Boundary Conditions in the PDE Modeler 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.



Boundary condition equation: h*u=r

Condition type:
○ Neumann
● Dirichlet

Coefficient	Value	Description
g	0	
q	0	
h	1	
r	0	

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:

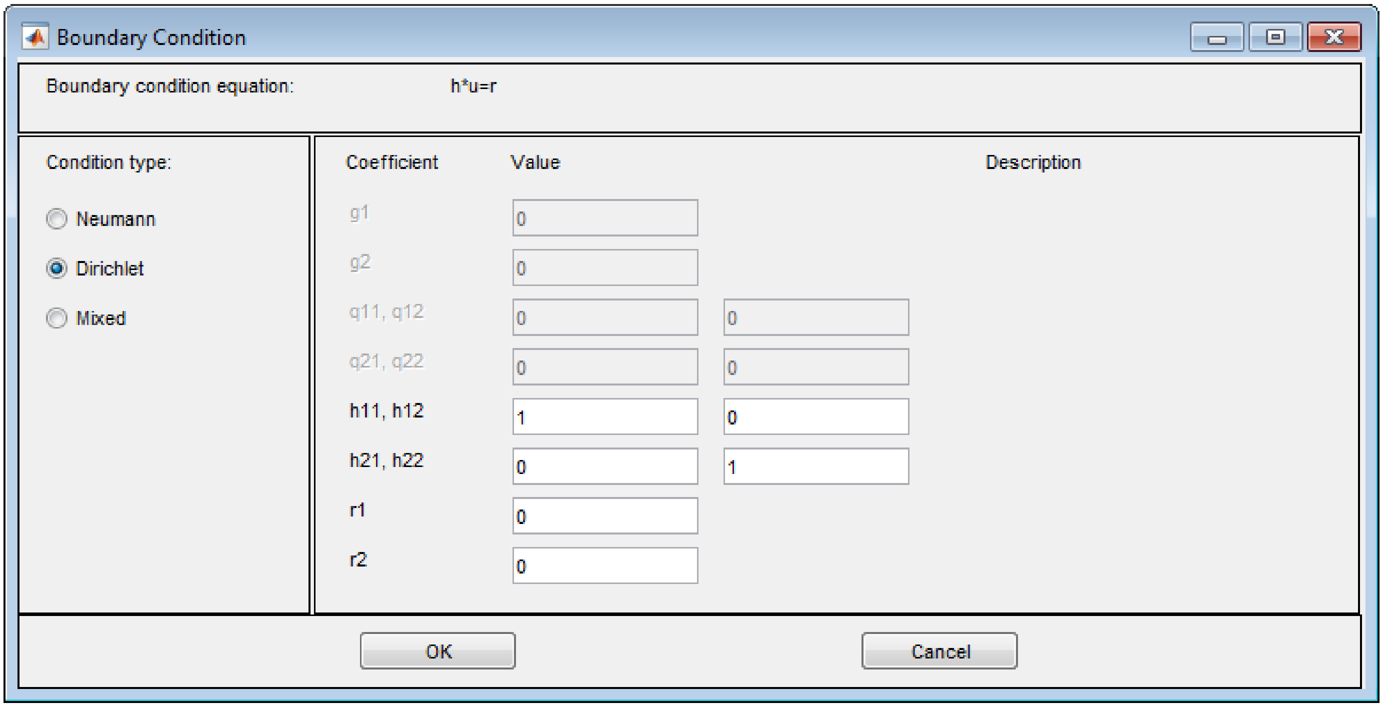
- Generalized Neumann conditions, where the boundary condition is determined by the coefficients `q` and `g` according to the following equation:

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

  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**).



Boundary condition equation: h*u=r

Condition type:

- ○ Neumann
- ● Dirichlet
- ○ Mixed

Coefficient	Value		Description
g1	0		
g2	0		
q11, q12	0	0	
q21, q22	0	0	
h11, h12	1	0	
h21, h22	0	1	
r1	0		
r2	0		

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.

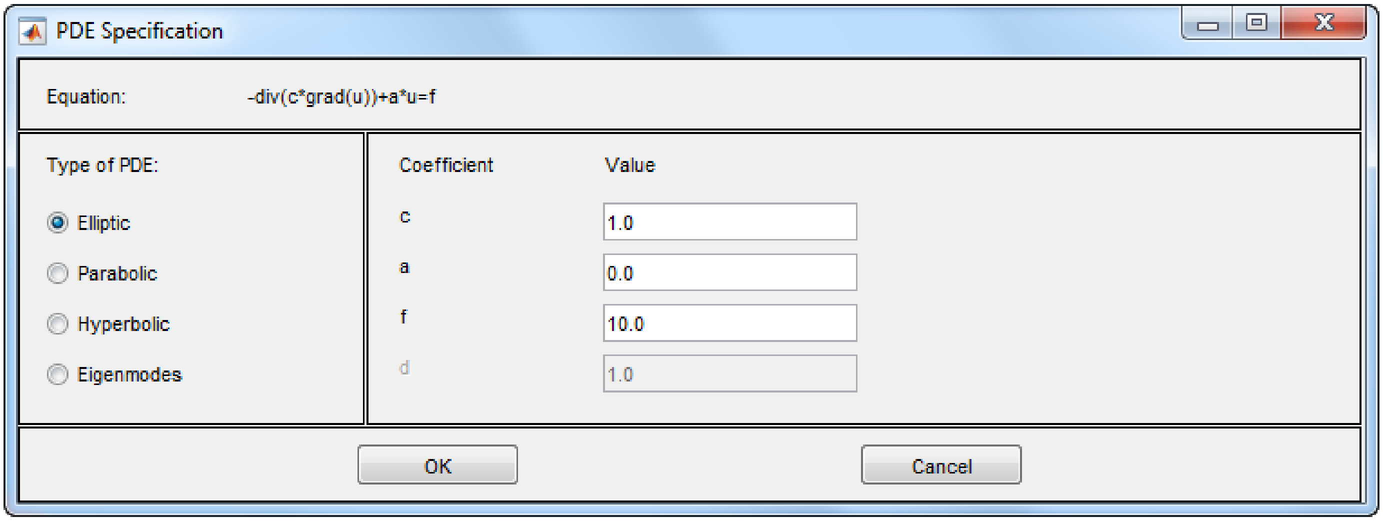
Examples: `(100-80*s).*nx`, and `cos(x.^2)`

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

Specify Coefficients in PDE Modeler App

Coefficients for Scalar PDEs

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



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

Type of PDE:

- ● Elliptic
- ○ Parabolic
- ○ Hyperbolic
- ○ Eigenmodes

Coefficient	Value
c	1.0
a	0.0
f	10.0
d	1.0

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**.

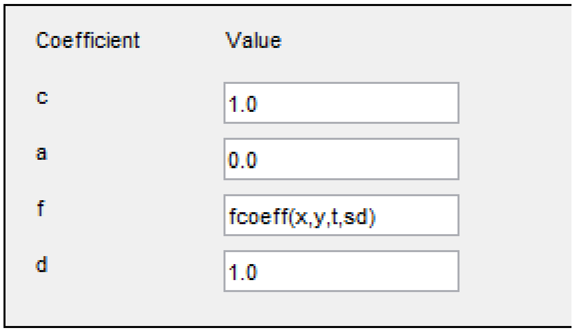
Note

- 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.



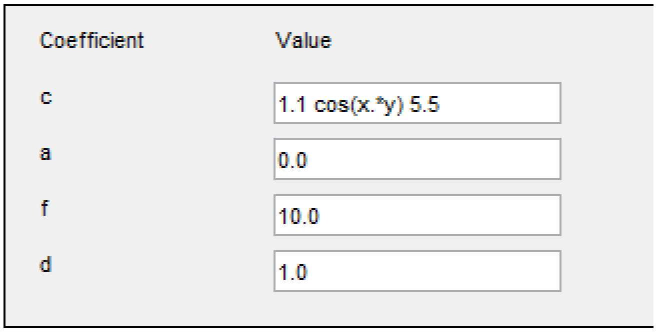
c	1.0
a	0.0
f	fcoeff(x,y,t,sd)
d	1.0

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
```
(4-1)



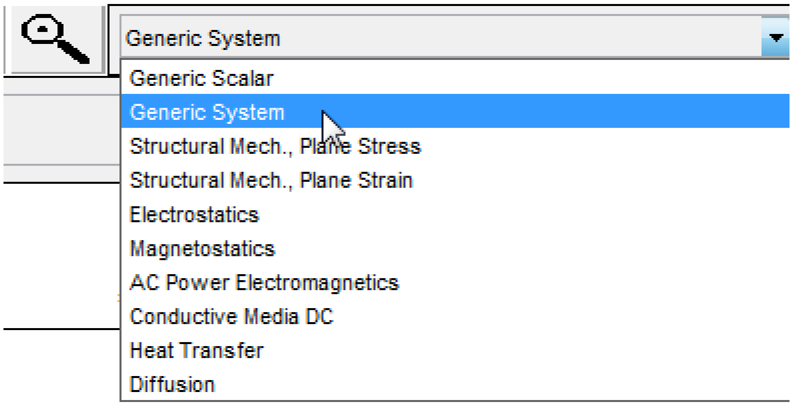
c	1.1 cos(x.*y) 5.5
a	0.0
f	10.0
d	1.0

This corresponds to coefficients for the parabolic equation

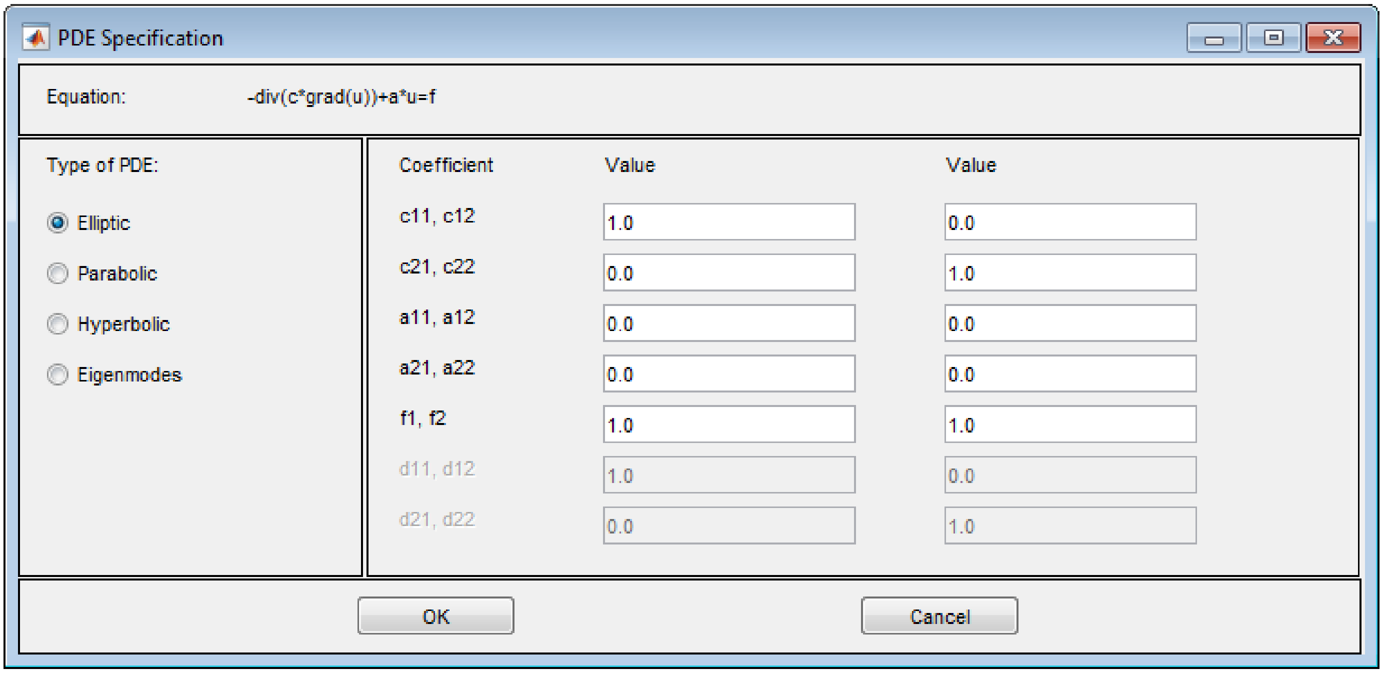
$$\frac{\partial u}{\partial t} - \nabla \cdot \left(\begin{pmatrix} 1.1 & \cos(xy) \\ \cos(xy) & 5.5 \end{pmatrix} \nabla u \right) = 10.$$

Coefficients for Systems of PDEs

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



Then select **PDE > PDE Specification**.



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

Type of PDE:

● Elliptic
○ Parabolic
○ Hyperbolic
○ Eigenmodes

Coefficient	Value	Value
c11, c12	1.0	0.0
c21, c22	0.0	1.0
a11, a12	0.0	0.0
a21, a22	0.0	0.0
f1, f2	1.0	1.0
d11, d12	1.0	0.0
d21, d22	0.0	1.0

Enter character expressions for coefficients using the form in “Coefficients for Scalar PDEs” on page 4-14, with additional options for nonlinear equations. The additional options are:

- Represent the `i`th component of the solution `u` using `'u(i)'` for `i` = 1 or 2.

- Similarly, represent the `i`th 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 Modeler app. Represent your problem coefficients at the command line.

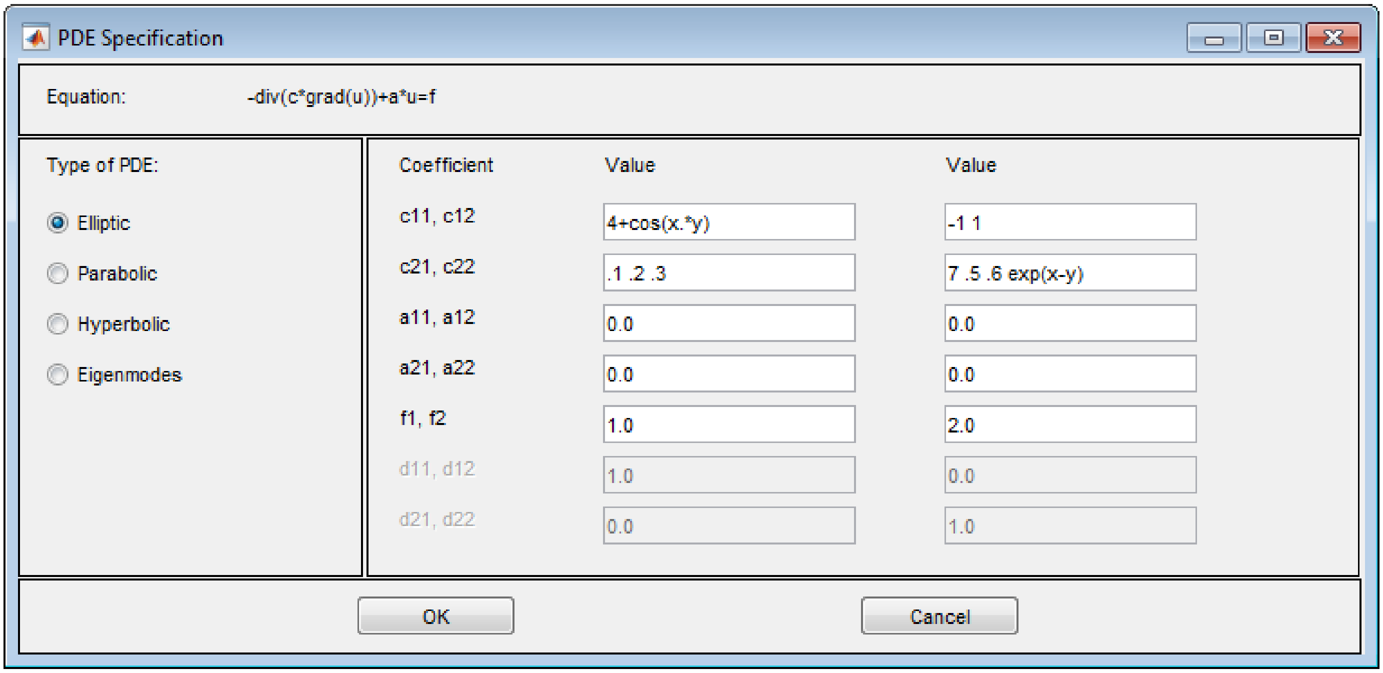
You can enter scalars into the `c` matrix, corresponding to these equations:

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

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)



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

Type of PDE:

- ● Elliptic
- ○ Parabolic
- ○ Hyperbolic
- ○ Eigenmodes

Coefficient	Value	Value
c11, c12	4+cos(x.*y)	-1 1
c21, c22	.1 .2 .3	7 .5 .6 exp(x-y)
a11, a12	0.0	0.0
a21, a22	0.0	0.0
f1, f2	1.0	2.0
d11, d12	1.0	0.0
d21, d22	0.0	1.0

These expressions correspond to the equations

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

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

Coefficients That Depend on Time and Space

This example shows how to enter time- and coordinate-dependent coefficients in the PDE Modeler app.

Solve the parabolic PDE,

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

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:

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

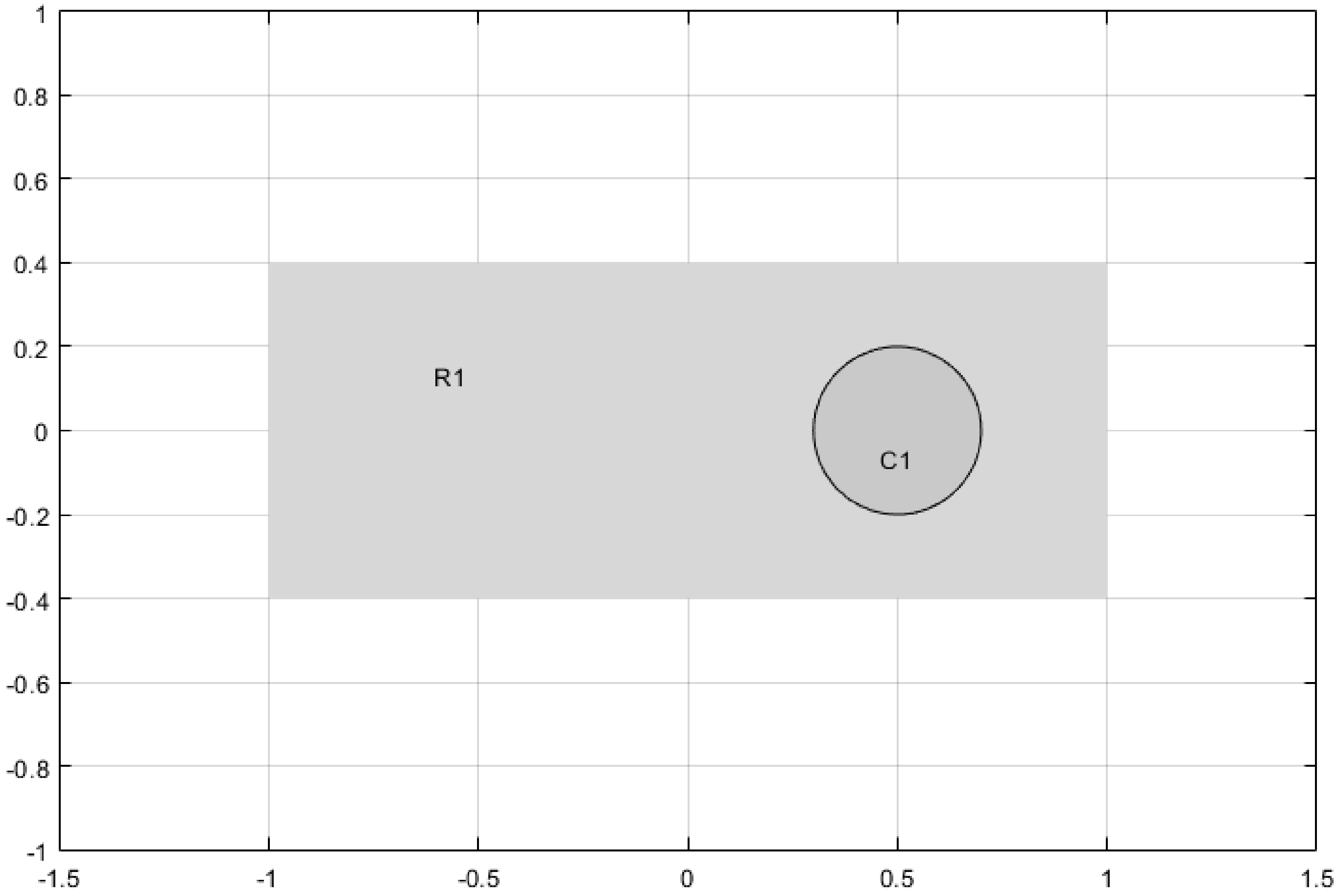
- $c = 1 + .x^2 + y^2$

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

1. Write the 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;
   ```

2. Open the PDE Modeler app by using the `pdeModeler` command.
3. Display grid lines by selecting **Options** > **Grid**.
4. Align new shapes to the grid lines by selecting **Options** > **Snap**.
5. Draw a rectangle with the corners at (-1,-0.4), (-1,0.4), (1,0.4), and (1,-0.4). To do this, first click the ▭ button. Then click one of the corners using the left mouse button and drag to draw a rectangle.
6. Draw a circle with the radius 0.2 and the center at (0.5,0). To do this, first click the ⊕ button. Then right-click the origin and drag to draw a circle. Right-clicking constrains the shape you draw so that it is 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.
7. Model the geometry by entering `R1-C1` in the **Set formula** field.



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

9 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 rectangle, use the Dirichlet boundary condition with `h = 1` and `r = t*(x-y)`.
- For the circle, use the Neumann boundary condition with `g = x.^2+y.^2` and `q = 1`.

10 Specify the coefficients by selecting **PDE** > **PDE Specification** or clicking the **PDE** button on the toolbar. Select the **Parabolic** type of PDE. Specify `c = 1+x.^2+y.^2`, `a = 0`, `f = framp(t)`, and `d = 5`.

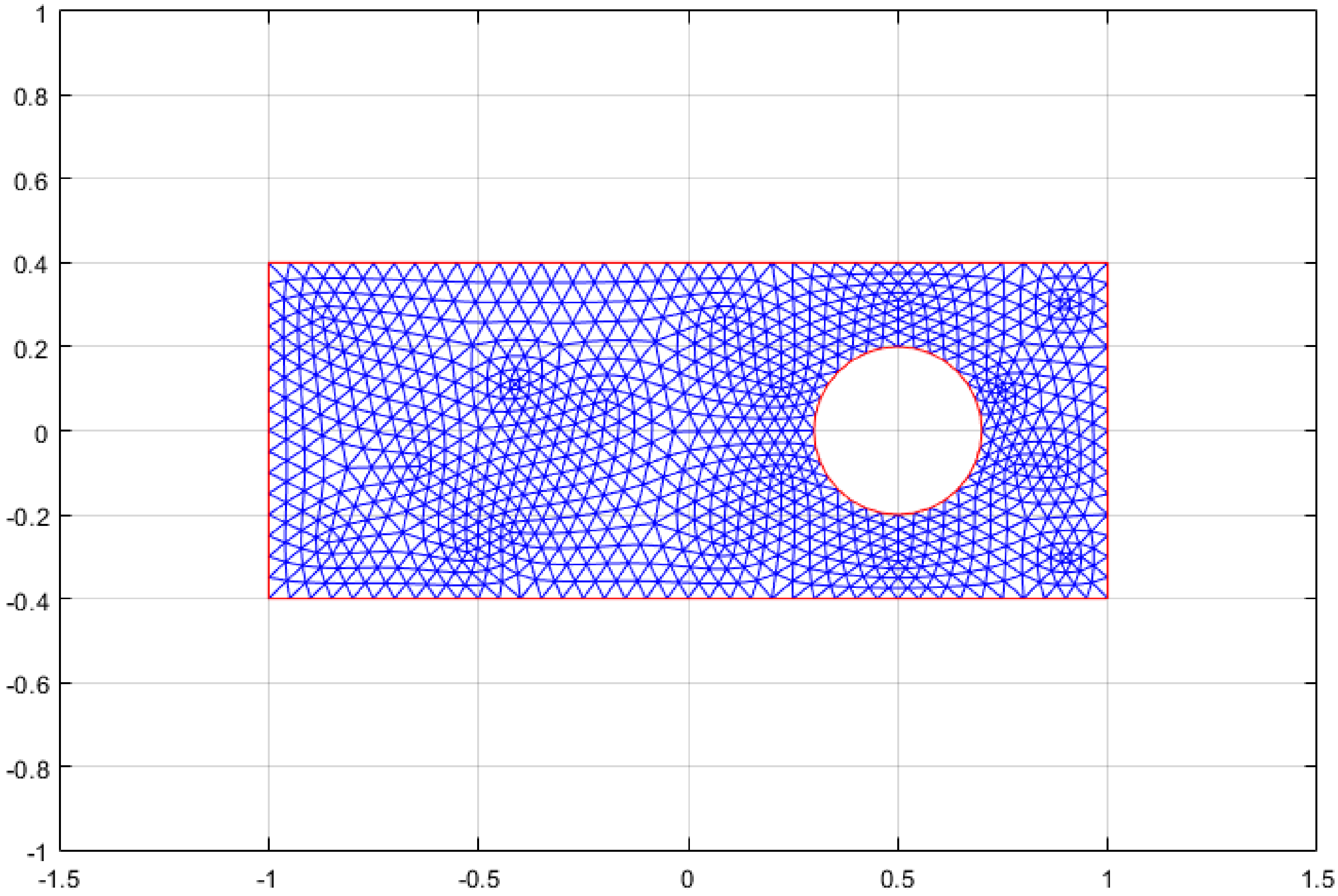
Note Do not include quotes or spaces when you specify your coefficients the PDE Modeler app. The parser interprets all inputs as vectors of characters. It can misinterpret a space as a vector separator, as when a MATLAB vector uses a space to separate elements of a vector.

---

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

12 Refine the mesh twice by selecting **Mesh** > **Refine Mesh**.

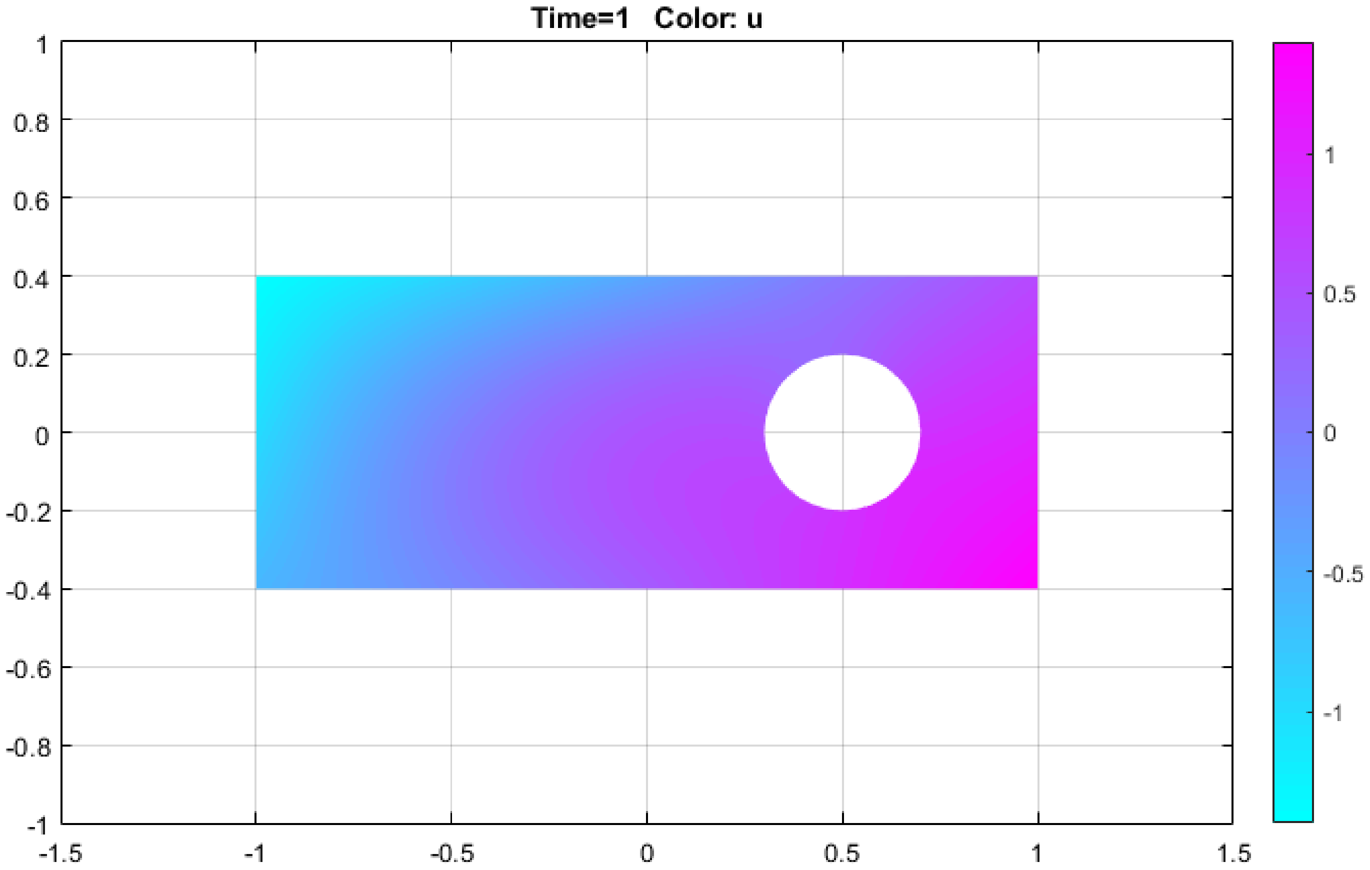
13 Improve the triangle quality by selecting **Mesh** > **Jiggle Mesh**.



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

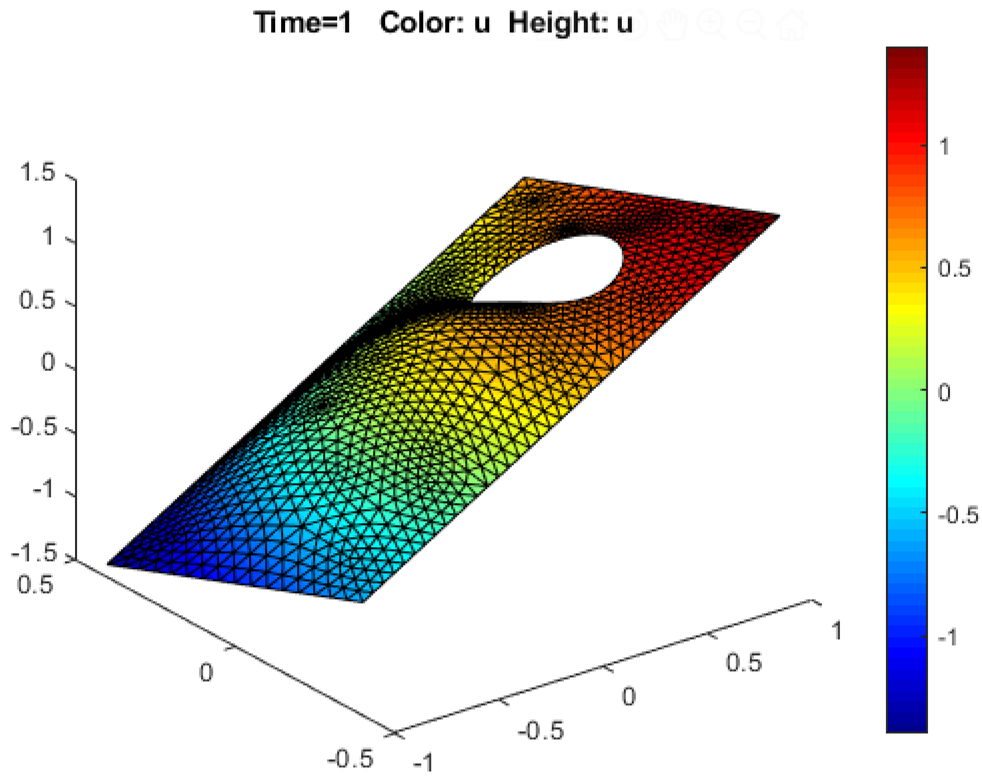
In the resulting dialog box, set the time to `linspace(0,1,50)` and the initial value $u(t_0)$ to `0`.

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



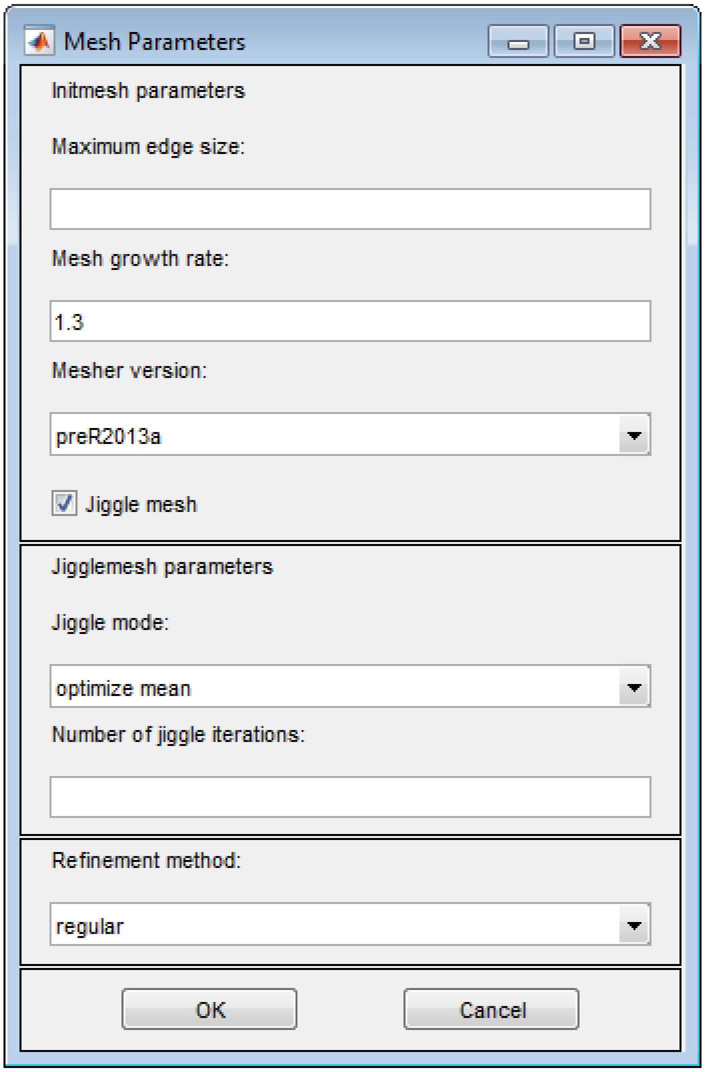
16 Visualize the solution as a 3-D static plot. To do this:

- **a** Select **Plot** > **Parameters**.
- **b** In the resulting dialog box, select the **Color** and **Height (3-D plot)** options.
- **c** Select the **Show mesh** option.
- **d** Change the colormap to `jet` by using the corresponding drop-down menu in the same dialog box.


Time=1 Color: u Height: u
1.5
1
0.5
0
-0.5
-1
-1.5
0.5
0
-0.5
-1
-0.5
0
0.5
1
1
0.5
0
-0.5
-1

Specify Mesh Parameters in the PDE Modeler App

Select **Parameters** from the **Mesh** menu to open the following dialog box containing mesh generation parameters.



Maximum edge size:

Mesh growth rate: 1.3

Mesher version: preR2013a

☑ Jiggle mesh

Jiggle mode: optimize mean

Number of jiggle iterations:

Refinement method: regular

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.

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

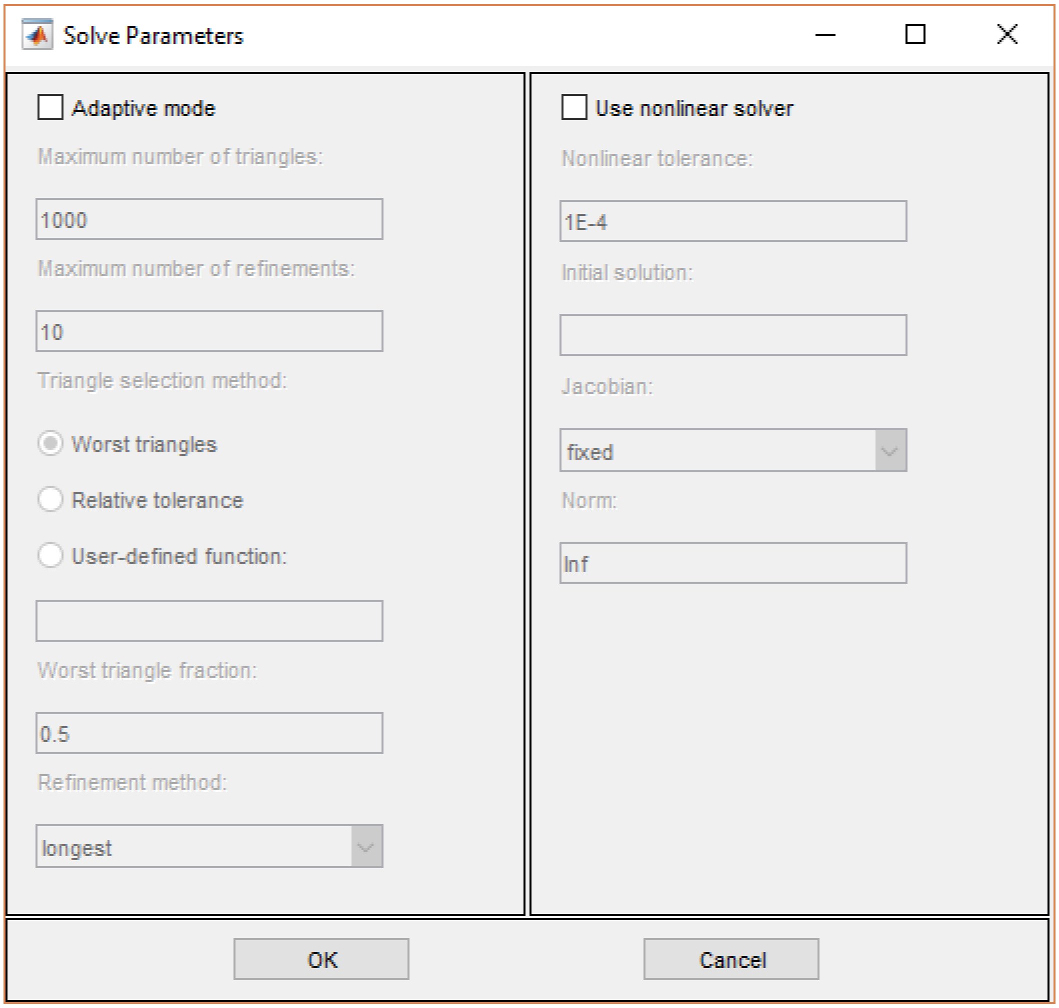
To initialize a triangular mesh, select **Initialize Mesh** from the **Mesh** menu or click the △ button.

To refine a mesh, select **Refine Mesh** from the **Mesh** menu or click the ⟁ button.

Adjust Solve Parameters in the PDE Modeler 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



☐ Adaptive mode

Maximum number of triangles: 1000

Maximum number of refinements: 10

Triangle selection method:

● Worst triangles

○ Relative tolerance

○ User-defined function:

Worst triangle fraction: 0.5

Refinement method: longest

☐ Use nonlinear solver

Nonlinear tolerance: 1E-4

Initial solution:

Jacobian: fixed

Norm: Inf

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” on page 3-249 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 Modeler App” on page 4-24.

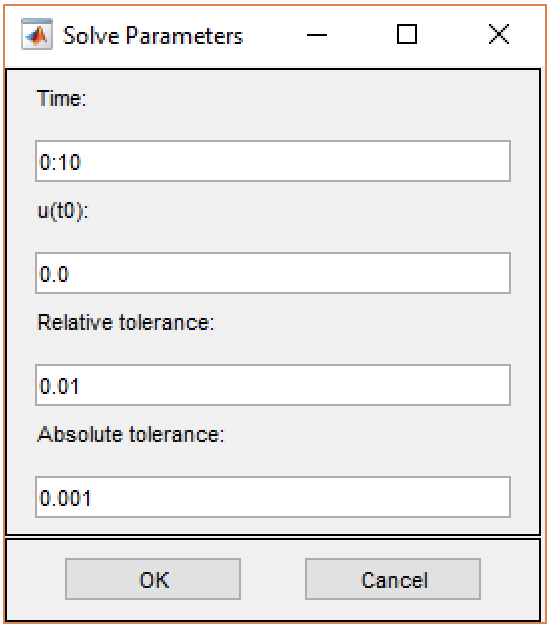
If the problem is nonlinear, i.e., parameters in the PDE are directly dependent on the solution `u`, a nonlinear solver must be used. The following parameters are used:

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

  Examples: `1`, and `exp(x.*y)`. Optional parameter, defaults to zero.
- **Jacobian**. Jacobian approximation method: `fixed` (the default), a fixed point iteration, `lumped`, a “lumped” (diagonal) approximation, or `full`, the full Jacobian.
- **Norm**. The type of norm used for computing the residual. Enter as `energy` for an energy norm, or as a real scalar p to give the lp norm. The default is `Inf`, the infinity (maximum) norm.

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

Parabolic Equations



Time: 0:10

u(t0): 0.0

Relative tolerance: 0.01

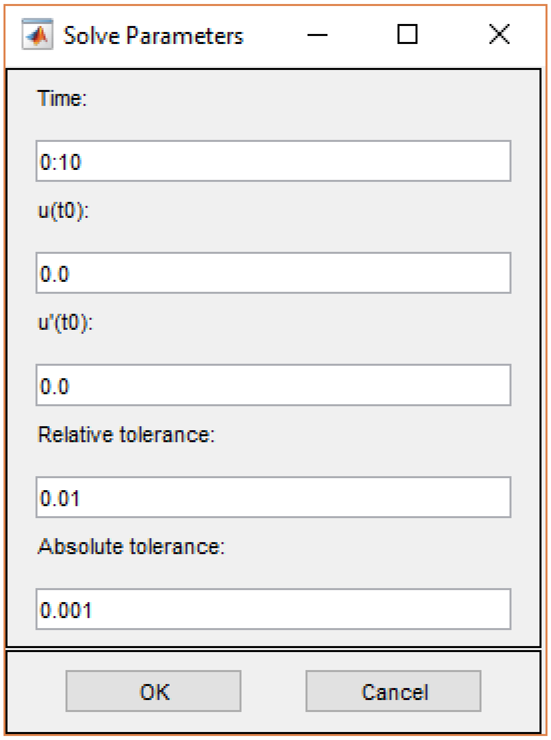
Absolute tolerance: 0.001

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



Time:
0:10

u(t0):
0.0

u'(t0):
0.0

Relative tolerance:
0.01

Absolute tolerance:
0.001

The solve parameters for the hyperbolic PDEs are:

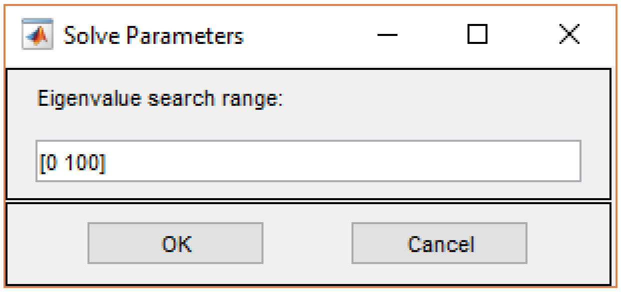
- **Time**. A MATLAB vector of times at which a solution to the hyperbolic PDE should be generated. The relevant time span is dependent on the dynamics of the problem.

  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]`

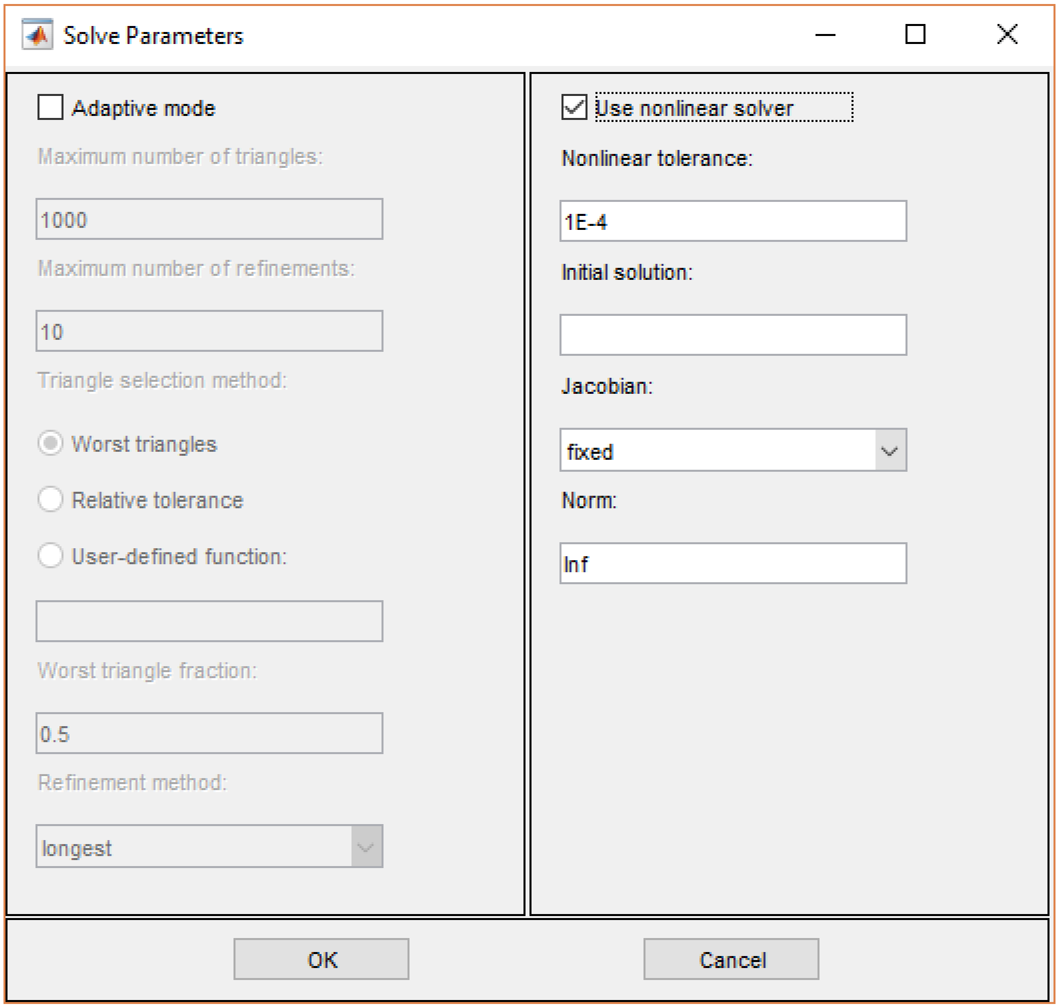


Eigenvalue search range:

[0 100]

Nonlinear Equations

Before solving a nonlinear elliptic PDE in the PDE Modeler app, select **SolveParameters**. Then select **Use nonlinear solver** and click **OK**.



☐ Adaptive mode

Maximum number of triangles: 1000

Maximum number of refinements: 10

Triangle selection method:

● Worst triangles

○ Relative tolerance

○ User-defined function:

Worst triangle fraction: 0.5

Refinement method: longest

☑ Use nonlinear solver

Nonlinear tolerance: 1E-4

Initial solution:

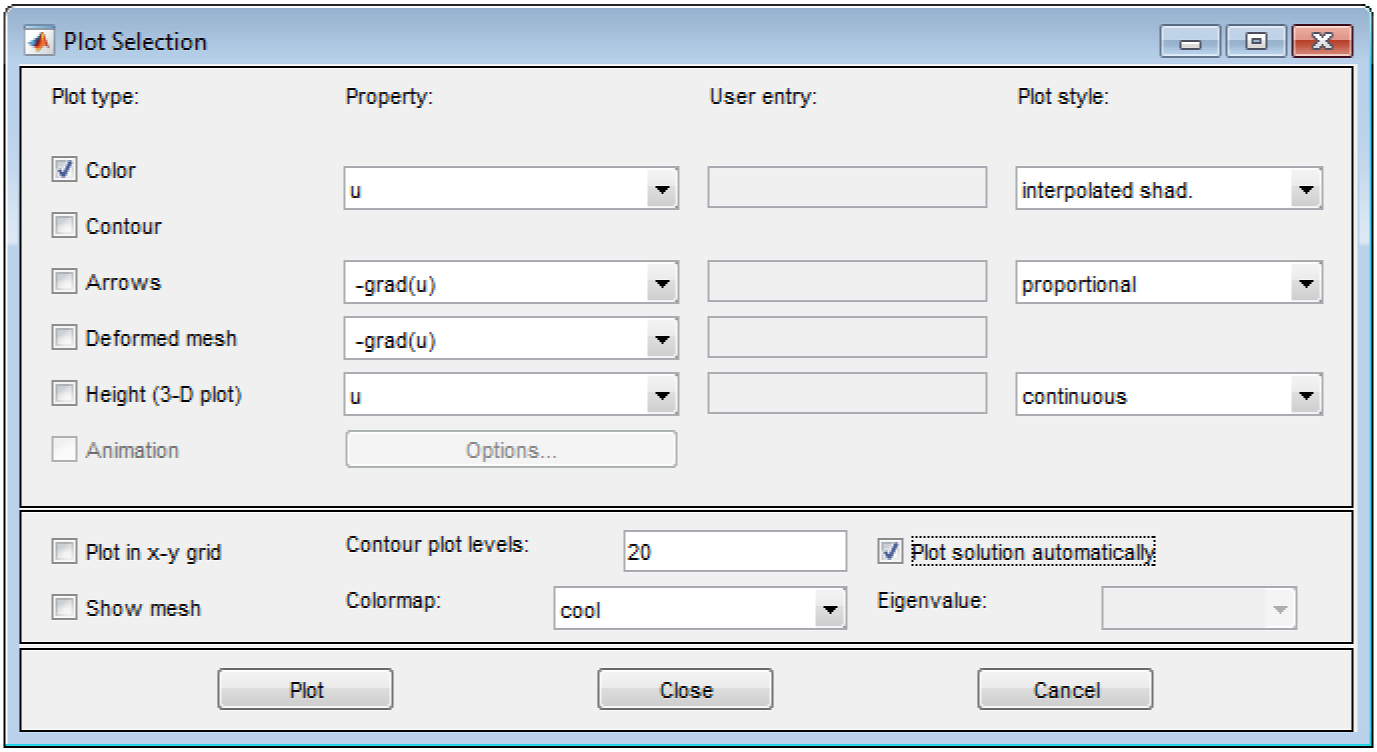
Jacobian: fixed

Norm: Inf

Plot the Solution in the PDE Modeler 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.



Plot type:	Property:	User entry:	Plot style:
☑ Color	u		interpolated shad.
☐ Contour			
☐ Arrows	-grad(u)		proportional
☐ Deformed mesh	-grad(u)		
☐ Height (3-D plot)	u		continuous
☐ Animation	Options...		

☐ Plot in x-y grid
☐ Show mesh
Contour plot levels: 20
Colormap: cool
☑ Plot solution automatically
Eigenvalue:

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

The upper part of the dialog box contains four columns:

- **Plot type** (far left) contains a row of six different plot types, which can be used for visualization:
  - **Color**. Visualization of a scalar property using colored surface objects.
  - **Contour**. Visualization of a scalar property using colored contour lines. The contour lines can also enhance the color visualization when both plot types (**Color** and **Contour**) are checked. The contour lines are then drawn in black.
  - **Arrows**. Visualization of a vector property using arrows.
  - **Deformed mesh**. Visualization of a vector property by deforming the mesh using the vector property. The deformation is automatically scaled to 10% of the problem domain. This plot type is primarily intended for visualizing *x*- and *y*-displacements (*u* and *v*) for problems in structural mechanics. If no other plot type is selected, the deformed triangular mesh is displayed.
  - **Height (3-D plot)**. Visualization of a scalar property using height (*z*-axis) in a 3-D plot. 3-D plots are plotted in separate figure windows. If the **Color** and **Contour** plot types are not used,

  the 3-D plot is simply a mesh plot. You can visualize another scalar property simultaneously using **Color** and/or **Contour**, which results in a 3-D surface or contour plot.

  - **Animation**. Animation of time-dependent solutions to parabolic and hyperbolic problems. If you select this option, the solution is recorded and then animated in a separate figure window using the MATLAB `movie` function.

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

- **Property** contains four pop-up menus containing lists of properties that are available for plotting using the corresponding plot type. From the first pop-up menu you control the property that is visualized using color and/or contour lines. The second and third pop-up menus contain vector valued properties for visualization using arrows and deformed mesh, respectively. From the fourth pop-up menu, finally, you control which scalar property to visualize using *z*-height in a 3-D plot. The lists of properties are dependent on the current application mode. For the generic scalar mode, you can select the following scalar properties:

  - **u**. The solution itself.
  - **abs(grad(u))**. The absolute value of ∇u, evaluated at the center of each triangle.
  - **abs(c*grad(u))**. The absolute value of $c \cdot \nabla u$, evaluated at the center of each triangle.
  - **user entry**. A MATLAB expression returning a vector of data defined on the nodes or the triangles of the current triangular mesh. The solution `u`, its derivatives `ux` and `uy`, the *x* and *y* components of $c \cdot \nabla u$, `cux` and `cuy`, and `x` and `y` are all available in the local workspace. You enter the expression into the edit box to the right of the **Property** pop-up menu in the **User entry** column.

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

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

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

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

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

The variables available in the local workspace for a user entered expression are the same for all scalar and system modes (the solution is always referred to as *u* and, in the system case, `v`).

- **User entry** contains four edit fields where you can enter your own expression, if you select the user entry property from the corresponding pop-up menu to the left of the edit fields. If the user entry property is not selected, the corresponding edit field is disabled.

- **Plot style** contains three pop-up menus from which you can control the plot style for the color, arrow, and height plot types respectively. The available plot styles for color surface plots are

  - **Interpolated shading**. A surface plot using the selected colormap and interpolated shading, i.e., each triangular area is colored using a linear, interpolated shading (the default).
  - **Flat shading**. A surface plot using the selected colormap and flat shading, i.e., each triangular area is colored using a constant color.

  You can use two different arrow plot styles:

  - **Proportional**. The length of the arrow corresponds to the magnitude of the property that you visualize (the default).
  - **Normalized**. The lengths of all arrows are normalized, i.e., all arrows have the same length. This is useful when you are interested in the direction of the vector field. The direction is clearly visible even in areas where the magnitude of the field is very small.

  For height (3-D plots), the available plot styles are:

  - **Continuous**. Produces a “smooth” continuous plot by interpolating data from triangle midpoints to the mesh nodes (the default).
  - **Discontinuous**. Produces a discontinuous plot where data and z-height are constant on each triangle.

A total of three properties of the solution—two scalar properties and one vector field—can be visualized simultaneously. If the **Height (3-D plot)** option is turned off, the solution plot is a 2-D plot and is plotted in the main axes of the PDE Modeler 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

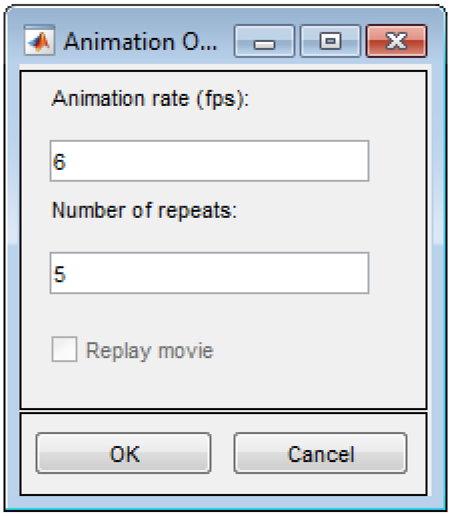
In the middle of the dialog box are a number of additional plot control options:

- **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 color maps: `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.



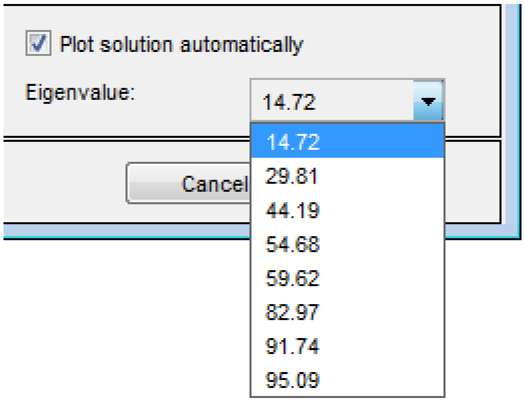
Animation rate (fps): 6

Number of repeats: 5

☐ Replay movie

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.



☑ Plot solution automatically

Eigenvalue: 14.72

For eigenvalue problems, the bottom right part of the dialog box contains a drop-down 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

Functions

Visualize PDE Results

Create and explore visualizations of PDE results in the Live Editor

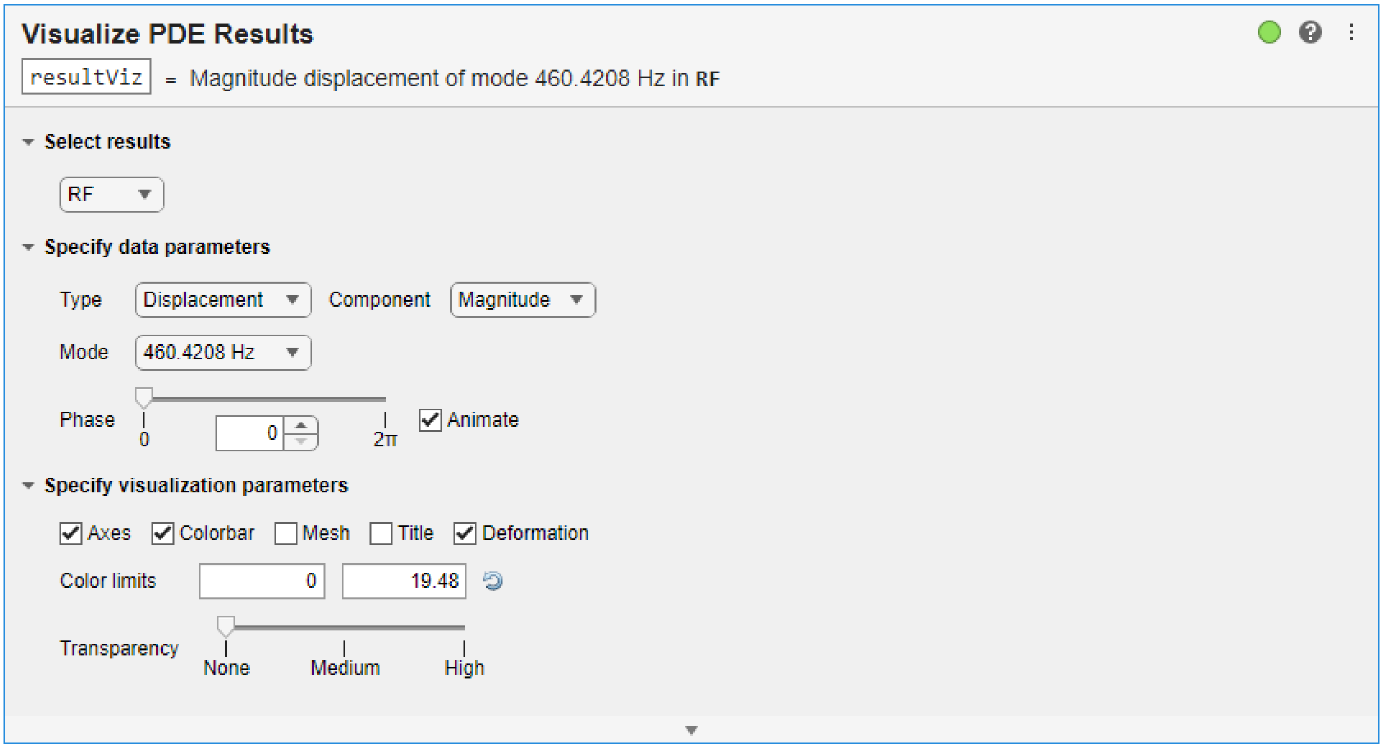
Description

The **Visualize PDE Results** task enables you to plot and inspect results of structural, thermal, electromagnetic, or general PDE analysis using interactive controls. The task automatically generates MATLAB code for your live script.

Using this task, you can:

- Select which solution data to display.
- Quickly access plots for different frequencies, modes, time steps, and phases.
- Show animated solutions.
- Add a mesh to a solution plot.
- Adjust color limits for a colormap.
- Adjust surface transparency.

For general information about Live Editor tasks, see “Add Interactive Tasks to a Live Script”.



Visualize PDE Results

`resultViz` = Magnitude displacement of mode 460.4208 Hz in **RF**

Select results

RF

Specify data parameters

Type: Displacement

Component: Magnitude

Mode: 460.4208 Hz

Phase: 0 (0 to 2π) ☑ Animate

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☑ Deformation

Color limits: 0 19.48

Transparency: None (None, Medium, High)

Open the Task

To add the **Visualize PDE Results** task to a live script in the MATLAB Editor:

- On the **Live Editor** tab, select **Task** > **Visualize PDE Results**.
- In a code block in the script, type a relevant keyword, such as `pde`. Select **Visualize PDE Results** from the suggested command completions.

Examples

Visualization of Modal Analysis Results in Live Editor

Use the Visualize PDE Results task in the Live Editor to explore structural modal analysis results of a tuning fork. Open this example to see a preconfigured script containing the **Visualize PDE Results** task.

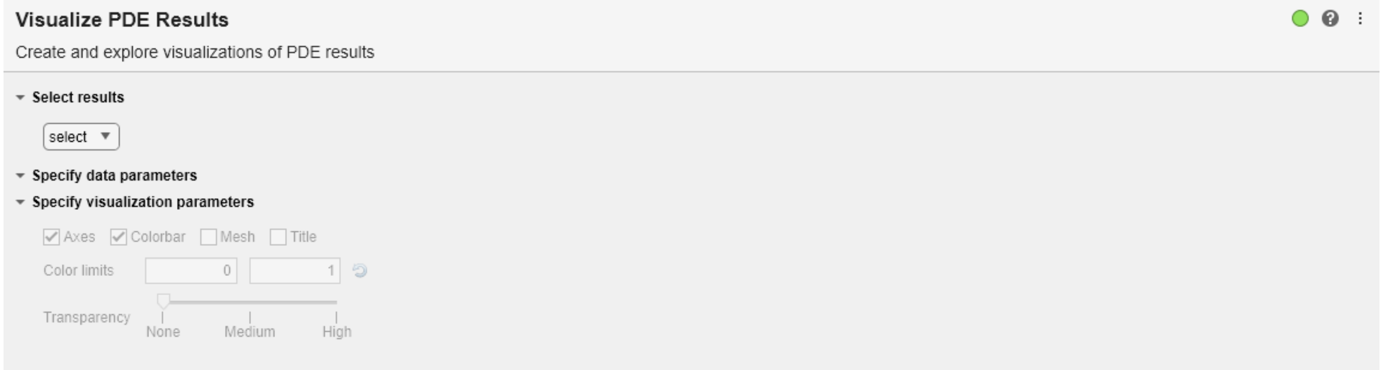
Find natural frequencies and mode shapes for the fundamental mode of a tuning fork and the next several modes. To perform unconstrained modal analysis of a structure, you can specify just the geometry, mesh, and material properties.

```
model = createpde("structural","modal-solid");
importGeometry(model,"TuningFork.stl");
structuralProperties(model,"YoungsModulus",210e9, ...
                           "PoissonsRatio",0.3, ...
                           "MassDensity",8000);
generateMesh(model,"Hmax",0.001);
```

Solve the model for a chosen frequency range.

```
RF = solve(model,"FrequencyRange",[-1,4000]*2*pi);
```

To visualize mode shapes, open the **Visualize PDE Results** Live Editor task. On the **Live Editor** tab, select **Task** > **Visualize PDE Results**. This action inserts the task into your script.



Visualize PDE Results

Create and explore visualizations of PDE results

Select results

select ▼

Specify data parameters

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title

Color limits 0 1

Transparency None Medium High

In the **Select results** section of the task, select RF from the drop-down list. Because there are no boundary constraints in this example, modal results include the rigid body modes. The first six near-zero frequencies indicate the six rigid body modes of the 3-D tuning fork. The first flexible mode is the seventh mode with a frequency of about 460 Hz.

Visualize PDE Results

`resultViz` = Magnitude displacement of mode 460.4208 Hz in **RF**

Select results

RF

Specify data parameters

Type: Displacement Component: Magnitude

Mode: 460.4208 Hz

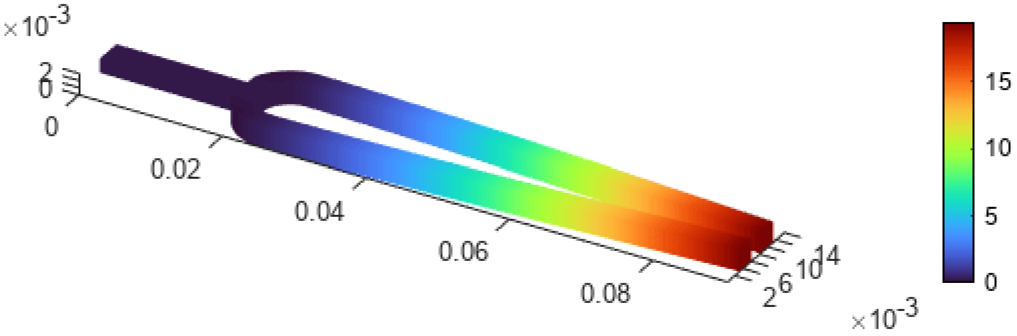
Phase: 0 (0 to 2π) ☐ Animate

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☑ Deformation

Color limits: 0 19.48

Transparency: None Medium High



The best way to visualize mode shapes is to animate the harmonic motion at their respective frequencies. Select **Animate** to see the harmonic motion at a frequency of about 460 Hz.

Visualize PDE Results

`resultViz2` = Magnitude displacement of mode 460.4208 Hz in **RF**

Select results

RF

Specify data parameters

Type: Displacement

Component: Magnitude

Mode: 460.4208 Hz

Phase: 0 (0 to 2π)

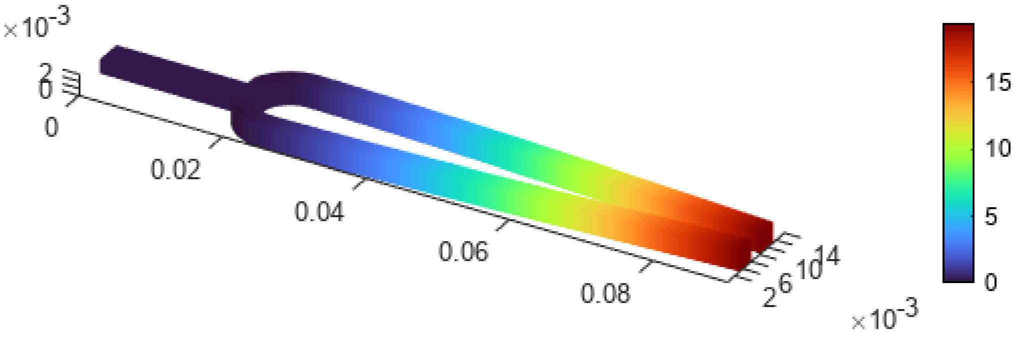
☑ Animate

Specify visualization parameters

☑ Axes ☑ Colorbar ☐ Mesh ☐ Title ☑ Deformation

Color limits: 0 to 19.48

Transparency: None (None, Medium, High)



- “Deflection Analysis of Bracket” on page 3-74
- “Thermal Deflection of Bimetallic Beam” on page 3-110
- “Modal and Frequency Response Analysis for Single Part of Kinova Gen3 Robotic Arm” on page 3-40
- “Finite Element Analysis of Electrostatically Actuated MEMS Device” on page 3-59

- "Temperature Distribution in Heat Sink" on page 3-199
- "Minimal Surface Problem" on page 3-241

Parameters

`Select results` — Results to plot

structural results | thermal results | electromagnetic results | general PDE results

Choose the results to plot from the structural, thermal, electromagnetic, and general PDE results available in the MATLAB workspace. You can select only one solution at a time. The solution must be specified as one of these objects:

- `StaticStructuralResults`, `TransientStructuralResults`, `ModalStructuralResults`, or `FrequencyStructuralResults` for a structural model
- `SteadyStateThermalResults`, `TransientThermalResults`, or `ModalThermalResults` for a thermal model
- `ElectrostaticResults`, `MagnetostaticResults`, `HarmonicResults`, or `ConductionResults` for an electromagnetic model
- `StationaryResults`, `TimeDependentResults`, or `EigenResults` for a general PDE model

Specify data parameters

`Type` — Part of solution to plot

`Displacement` | `Temperature` | `Electric field` | ...

Choose the part of the solution to plot, such as a displacement for a structural problem, temperature for a thermal problem, or electric field for an electrostatic problem.

`Component` — Components to plot

`Magnitude` (default) | X | Y | Z | ...

Choose to plot the *x*-, *y*-, or *z*-component of a solution such as displacement, stress, temperature gradient, electric field, or magnetic field.

`Mode` — Mode to plot

first mode (default) | number

Choose the number corresponding to the solution mode to plot.

`Frequency` — Solution frequency to plot

lowest frequency (default) | number

Choose the number corresponding to the solution frequency to plot.

`Decay rate` — Eigenvalue of thermal model

first eigenvalue (default) | number

Choose the number corresponding to the eigenvalue of a thermal model.

`Equation` — Equation to plot

1 (default) | positive integer

Choose the number corresponding to the equation to plot.

`Eigenvalue` — Eigenvalue of general PDE model
first eigenvalue (default) | number

Choose the number corresponding to the eigenvalue of a PDE model.

`Time` — Time to plot results for
solution times

Choose the time to plot results for. The slider values correspond to the times that you used when solving the problem.

`Phase` — Phase to plot results for
values are from 0 to 2π

Choose the phase to plot results for. The slider values are from 0 to 2π.

`Animate` — Animate plot
off (default) | on

Select this option to play the animation.

Specify visualization parameters

`Axes` — Show coordinate axes
on (default) | off

Select this option to show the coordinate axes.

`Colorbar` — Show color bar
on (default) | off

Select this option to show the color bar.

`Mesh` — Show mesh
off (default) | on

Select this option to show the mesh.

`Title` — Show plot title
off (default) | on

Select this option to show an autogenerated plot title.

`Deformation` — Deformed or undeformed shape for structural analysis models
on (default) | off

Switch between deformed and undeformed shapes for a 3-D structural model. In an undeformed shape, center nodes in quadratic meshes are always added at the halfway point between corners. When you plot a deformed shape, the center nodes can move away from the edge centers.

`Color limits` — Set colormap limits
numbers

Set the colormap limits. The default mapping is useful in most cases, but you can perform the mapping over any range you choose, even if the range you choose is different than the range of your data. Choosing a different mapping range allows you to:

- See where your data is at or beyond the limits of that range.
- See where your data lies within that range.

For details, see “Control Colormap Limits”.

`Transparency` — Surface transparency
`None` | transparency in the range from `None` to `High`

Adjust the surface transparency by moving the slider between `None`, `Medium`, and `High`.

Version History

Introduced in R2022b

See Also

`pdeplot` | `pdeplot3D` | `pdegplot` | `pdemesh` | `pdeviz`

Topics

“Deflection Analysis of Bracket” on page 3-74
“Thermal Deflection of Bimetallic Beam” on page 3-110
“Modal and Frequency Response Analysis for Single Part of Kinova Gen3 Robotic Arm” on page 3-40
“Finite Element Analysis of Electrostatically Actuated MEMS Device” on page 3-59
“Temperature Distribution in Heat Sink” on page 3-199
“Minimal Surface Problem” on page 3-241

adaptmesh

Package: `pde`

Create adaptive 2-D mesh and solve PDE

Note 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,Name,Value)
```

Description

`[u,p,e,t] = adaptmesh(g,b,c,a,f)` generates an adaptive `[p,e,t]` mesh and returns the solution `u` for an elliptic 2-D PDE problem

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

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

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

with the problem geometry and boundary conditions given by `g` and `b`. The mesh is described by the `p`, `e`, and `t` matrices.

Upon termination, the function issues one of these messages:

- `Adaption completed`. (This means that the `Tripick` function returned zero triangles to refine.)
- `Maximum number of triangles obtained`.
- `Maximum number of refinement passes obtained`.

`[u,p,e,t] = adaptmesh(g,b,c,a,f,Name,Value)` performs adaptive mesh generation and PDE solution for elliptic 2-D PDE problems using one or more `Name,Value` pair arguments.

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 the resulting solution to the exact solution. Set the options so that `adaptmesh` refines the triangles using the worst error criterion until it obtains a mesh with at least 500 triangles.

```
c45 = cos(pi/4);
L1 = [2 -c45 0  c45 0 1 0 0 0 0]';
```

```
L2 = [2 -c45 0 -c45 0 1 0 0 0 0]';
C1 = [1 -c45  c45 -c45 -c45 1 0 0 0 1]';
C2 = [1  c45  c45 -c45  c45 1 0 0 0 1]';
C3 = [1  c45 -c45  c45  c45 1 0 0 0 1]';
g = [L1 L2 C1 C2 C3];

[u,p,e,t] = adaptmesh(g,"cirsb",1,0,0,"Maxt",500,...
                     "Tripick","pdeadworst","Ngen",Inf);
```

```
Number of triangles: 204
Number of triangles: 208
Number of triangles: 217
Number of triangles: 230
Number of triangles: 265
Number of triangles: 274
Number of triangles: 332
Number of triangles: 347
Number of triangles: 460
Number of triangles: 477
Number of triangles: 699

Maximum number of triangles obtained.
```

Find the maximal absolute error.

```
x = p(1,:); y = p(2,:);
exact = ((x.^2 + y.^2).^(1/3).*cos(2/3*atan2(y,x)))';
max(abs(u - exact))
```

```
ans = 0.0028
```

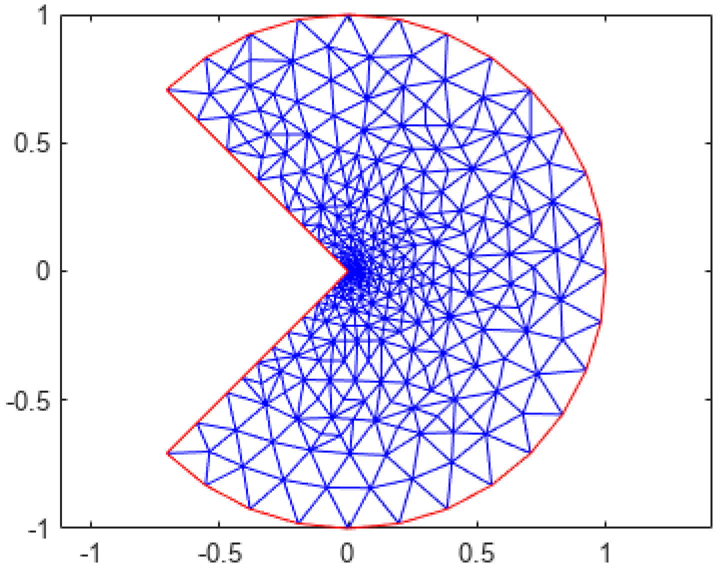
Find the number of triangles.

```
size(t,2)
```

```
ans = 699
```

Plot the mesh.

```
pdemesh(p,e,t)
```



Test how many refinements you need with a uniform triangle mesh.

```
[p,e,t] = initmesh(g);
[p,e,t] = refinemesh(g,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 = 0.0116
```

Find the number of triangles in this case.

```
size(t,2)
```

```
ans = 816
```

Refine the mesh one more time. The maximal absolute error for uniform meshing is still greater than for adaptive meshing.

```
[p,e,t] = refinemesh(g,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 = 0.0075
```

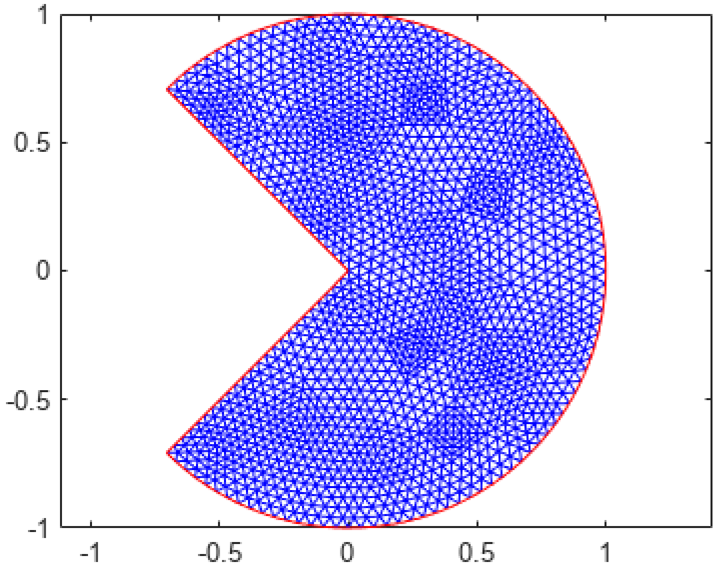
Find the number of triangles in this case.

```
size(t,2)
```

```
ans = 3264
```

Plot the mesh.

```
pdemesh(p,e,t)
```



Uniform refinement with more triangles produces a larger error. Typically, a problem with regular solution has an $O(h^2)$ error. However, this solution is singular since $u \approx r^{1/3}$ at the origin.

Input Arguments

g — Geometry description

decomposed geometry matrix | geometry function | handle to geometry function

Geometry description, specified as a decomposed geometry matrix, a geometry function, or a handle to the geometry function. For details about a decomposed geometry matrix, see `decsg`. For details about a geometry function, see "Parametrized Function for 2-D Geometry Creation" on page 2-22.

A geometry function must return the same result for the same input arguments in every function call. Thus, it must not contain functions and expressions designed to return a variety of results, such as random number generators.

Data Types: `double` | `char` | `string` | `function_handle`

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. Typically, you export a boundary matrix from the PDE Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `function_handle`

c — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `c` represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

The coefficients `c`, `a`, and `f` can depend on the solution `u` if you use the nonlinear solver by setting the value of `"Nonlin"` to `"on"`. The coefficients cannot be functions of the time `t`.

Example: `"cosh(x+y.^2)"`

Data Types: `double` | `char` | `string` | `function_handle`

a — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `a` represents the *a* coefficient in the scalar PDE

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

or in the system of PDEs

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

The coefficients `c`, `a`, and `f` can depend on the solution `u` if you use the nonlinear solver by setting the value of `"Nonlin"` to `"on"`. The coefficients cannot be functions of the time `t`.

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`

f — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `f` represents the *f* coefficient in the scalar PDE

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

or in the system of PDEs

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

The coefficients `c`, `a`, and `f` can depend on the solution `u` if you use the nonlinear solver by setting the value of `"Nonlin"` to `"on"`. The coefficients cannot be function of the time `t`.

Example: `char("sin(x)";"cos(y)";"tan(z)")`

Data Types: `double` | `char` | `string` | `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[u,p,e,t] = adaptmesh(g,"cirsb",1,0,0,"Maxt",500,"Tripick","pdeadworst","Ngen",Inf)`

`Maxt` — Maximum number of new triangles

`Inf` (default) | positive integer

Maximum number of new triangles, specified as the comma-separated pair consisting of `"Maxt"` and a positive integer.

Data Types: `double`

`Ngen` — Maximum number of triangle generations

10 (default) | positive integer

Maximum number of triangle generations, specified as the comma-separated pair consisting of `"Ngen"` and a positive integer.

Data Types: `double`

`Mesh` — Initial mesh

mesh generated by `initmesh` (default) | `[p,e,t]` mesh

Initial mesh, specified as the comma-separated pair consisting of `"Mesh"` and a mesh specified by `[p,e,t]` triples. By default, the function uses the mesh generated by the `initmesh` function.

Data Types: `double`

`Tripick` — Triangle selection method

indices of triangles returned by `pdeadworst` (default) | MATLAB function

Triangle selection method, specified as the comma-separated pair consisting of `"Tripick"` and a MATLAB function. By default, the function uses the indices of triangles returned by the `pdeadworst` function.

Given the error estimate computed by the function `pdejmps`, the triangle selection method identifies 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 the triangle midpoints.
- `u` is the current solution.
- `errf` is the computed error estimate.
- `Par` is the optional argument of `adaptmesh`.

The matrices `cc`, `aa`, `ff`, and `errf` all have Nt columns, where Nt is the current number of triangles. The numbers 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. The two standard triangle selection methods are `pdeadworst` and `pdeadgsc`.

- `pdeadworst` identifies triangles where `errf` exceeds a fraction of the worst value. The default fraction value is 0.5. You can change it by specifying the `Par` argument value when calling `adaptmesh`.
- `pdeadgsc` selects triangles using a relative tolerance criterion.

Data Types: `double`

`Par` — Function parameter for triangle selection method
0.5 (default) | number

Function parameter for the triangle selection method, specified as the comma-separated pair consisting of `"Par"` and a number between 0 and 1. The `pdeadworst` triangle selection method uses it as its `wlevel` argument. The `pdeadgsc` triangle selection method uses it as its `tol` argument.

Data Types: `double`

`Rmethod` — Triangle refinement method
`"longest"` (default) | `"regular"`

Triangle refinement method, specified as the comma-separated pair consisting of `"Rmethod"` and either `"longest"` or `"regular"`. For details on the refinement method, see `refinemesh`.

Data Types: `char` | `string`

`Nonlin` — Toggle to use a nonlinear solver
`"off"` (default) | `"on"`

Toggle to use a nonlinear solver, specified as the comma-separated pair consisting of `"Nonlin"` and `"on"` or `"off"`.

Data Types: `char` | `string`

`Toln` — Nonlinear tolerance
`1e-4` (default) | positive number

Nonlinear tolerance, specified as the comma-separated pair consisting of `"Toln"` and a positive number. The function passes this argument to `pdenonlin`, which iterates until the magnitude of the residual is less than `Toln`.

Data Types: `double`

`Init` — Nonlinear initial value
0 (default) | scalar | vector of characters | vector of numbers

Nonlinear initial value, specified as the comma-separated pair consisting of `"Init"` and a scalar, a vector of characters, or a vector of numbers. The function passes this argument to `pdenonlin`, which uses it as an initial guess in its `"U0"` argument.

Data Types: `double`

`Jac` — Nonlinear Jacobian calculation
`"fixed"` (default) | `"lumped"` | `"full"`

Nonlinear Jacobian calculation, specified as the comma-separated pair consisting of `"Jac"` and either `"fixed"`, `"lumped"`, or `"full"`. The function passes this argument to `pdenonlin`, which uses it as an initial guess in its `"Jacobian"` argument.

Data Types: `char` | `string`

`Norm` — Nonlinear solver residual norm

`Inf` (default) | p value for L^p norm

Nonlinear solver residual norm, specified as the comma-separated pair consisting of `"Norm"` and `p` value for 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)`. The function passes this argument to `pdenonlin`, which uses it as an initial guess in its `"Norm"` argument.

Data Types: `double` | `char` | `string`

`MesherVersion` — Algorithm for generating initial mesh

`"preR2013a"` (default) | `"R2013a"`

Algorithm for generating initial mesh, specified as the comma-separated pair consisting of `"MesherVersion"` and either `"R2013a"` or `"preR2013a"`. The `"R2013a"` algorithm runs faster, and can triangulate more geometries than the `"preR2013a"` algorithm. Both algorithms use Delaunay triangulation.

Data Types: `char` | `string`

Output Arguments

`u` — PDE solution

vector

PDE solution, returned as a vector.

- If the PDE is scalar, meaning that it has 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, the next `Np` elements represent the solution of equation 2, and so on.

`p` — Mesh points

2-by-`Np` matrix

Mesh points, returned as a 2-by-`Np` matrix. `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)` and the y-coordinate of point `k` in `p(2,k)`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

`e` — Mesh edges

7-by-`Ne` matrix

Mesh edges, returned as a 7-by-`Ne` matrix. `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 as [p,e,t] Triples” on page 2-172.

`t` — Mesh elements

4-by-`Nt` matrix

Mesh elements, returned as a 4-by-`Nt` matrix. `Nt` is the number of triangles in the mesh.

The `t(i,k)`, with `i` ranging from 1 through `end-1`, contain indices to the corner points of element `k`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172. The last row, `t(end,k)`, contains the subdomain numbers of the elements.

Algorithms

Mesh Refinement for Improving Solution Accuracy

Partial Differential Equation Toolbox provides the `refinemesh` function for global, uniform mesh refinement for 2-D geometries. It divides each triangle into four similar triangles by creating new corners at the mid-sides, 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 can be obtained by extrapolation.

The solutions of equations often have geometric features such as 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. In such cases, it is more efficient to refine the mesh selectively. The selection that is based on estimates of errors in the computed solutions is called *adaptive mesh refinement*.

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 stops34 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 (computes an estimate of the element error contribution), the mesh refiner (selects and subdivides elements), and the termination criteria.

Error Estimate for FEM Solution

The adaptation is a feedback process. It is easily applied to a larger range of problems than those for which its design was tailored. Estimates, selection criteria, and so on must be optimal for giving the most accurate solution at fixed cost or lowest computational effort for a given accuracy. Such results have been proven only for model problems, but generally, the equidistribution heuristic has been found nearly optimal. Element sizes must be chosen so 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 bounds might not exist, while the refinement scheme is still well defined and works.

The error indicator function used in the software is an elementwise estimate of the contribution, based on [1] and [2]. For Poisson's equation $-\Delta u = f$ on Ω, the following error estimate for the FEM-solution u_h holds in the L_2-norm $\| \cdot \|$:

$$\| \nabla (u - u_h) \| \leq \alpha \| h f \| + \beta D_h(u_h)$$

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 the edge τ, h_τ is the length of the edge τ, and the sum runs over E_i, the set of all interior edges of the triangulation. The coefficients α and β are independent of the triangulation. This bound is turned into an elementwise error indicator function $E(K)$ for the element K by summing the contributions from its edges.

The general form of the error indicator function for the elliptic equation

$$-\nabla \cdot (c\nabla u) + au = f \tag{5-1}$$

is

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

where $\mathbf{n}_\tau$ is the unit normal of the edge τ and the braced term is the jump in flux across the element edge. The L_2 norm is computed over the element K. The `pdejmps` function computes this error indicator.

Mesh Refinement Functions

Partial Differential Equation Toolbox mesh refinement is geared to elliptic problems. For reasons of accuracy and ill-conditioning, such problems 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 is used in [3], 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.

There are two selection criteria. 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 specified dimensionless tolerance. The comparison with the tolerance is properly scaled with respect to domain, solution size, and so on.

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 might never vanish, regardless of element size, making equidistribution impossible.

Version History

Introduced before R2006a

Performance and robustness enhancements in meshing algorithm

Performance change in R2013a

`adaptmesh` provides an enhancement option for increased meshing speed and robustness. Choose the enhanced algorithm by setting the `MesherVersion` name-value pair to `'R2013a'`. The default `MesherVersion` value of `'preR2013a'` gives the same mesh as previous toolbox versions.

The enhancement is available in`pdeModeler` from the **Mesh > Parameters > Mesher version** menu.

References

[1] Johnson, C. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Lund, Sweden: Studentlitteratur, 1987.

[2] Johnson, C., and K. Eriksson. *Adaptive Finite Element Methods for Parabolic Problems I: A Linear Model Problem*. SIAM J. Numer. Anal, 28, (1991), pp. 43–77.

[3] Rosenberg, I.G., and F. Stenger. *A lower bound on the angles of triangles constructed by bisecting the longest side*. Mathematics of Computation. Vol 29, Number 10, 1975, pp 390–395.

See Also

`initmesh` | `refinemesh`

Topics

“Mesh Data as [p,e,t] Triples” on page 2-172

addCell

Combine two geometries by adding one inside a cell of another

Syntax

`g3 = addCell(g1,g2)`

Description

`g3 = addCell(g1,g2)` creates nonempty cells inside `g1` using all cells of `g2`. All cells of the geometry `g2` must be contained inside one cell of the geometry `g1`. Ensure that the geometries do not have enclosed cavities and do not intersect one another.

The combined geometry contains cells from both geometries. The cells from `g1` retain their original IDs, while the cells from `g2` are numbered starting with `N+1`, where `N` is the number of cells in `g1`.

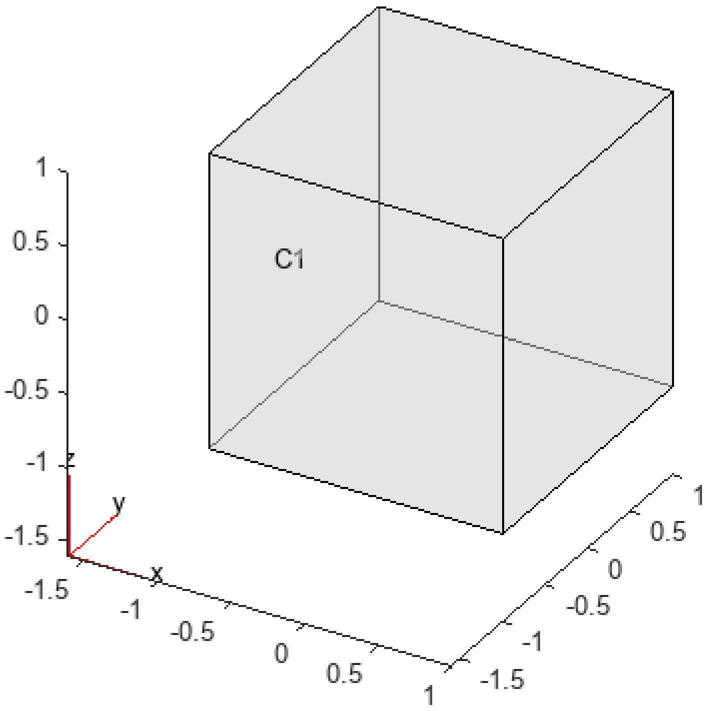
Note Added cells modify a geometry, but they do not modify the corresponding mesh. After modifying a geometry, always call `generateMesh` to ensure a proper mesh association with the new geometry.

Examples

Combine Two Geometries

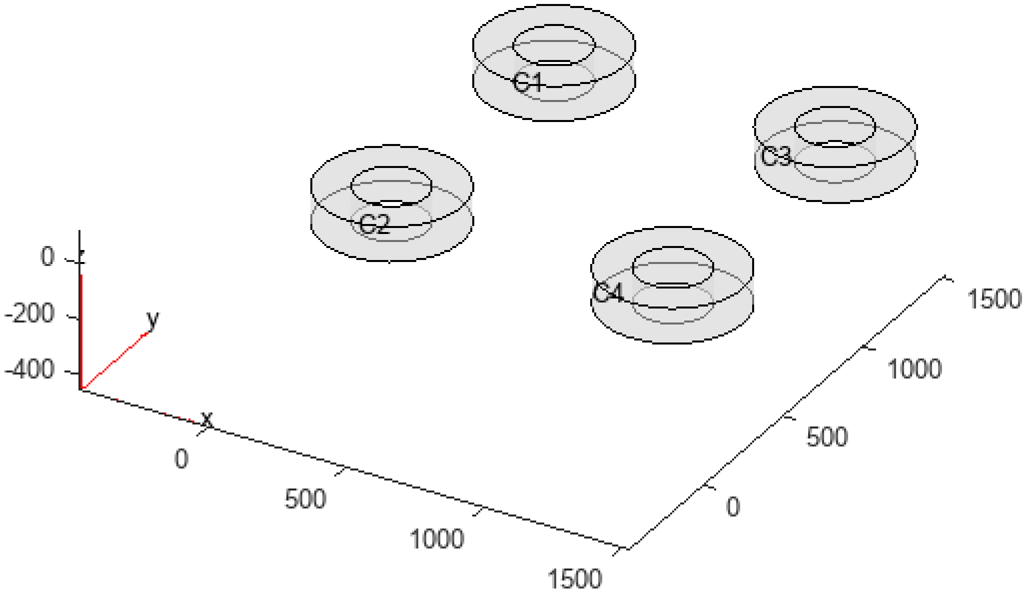
Create and plot a geometry.

```
g1 = multicuboid(2,2,2,"Zoffset",-1);
pdegplot(g1,"CellLabels","on","FaceAlpha",0.5)
```



Import and plot another geometry.

```
g2 = importGeometry("DampingMounts.stl");
pdegplot(g2,"CellLabels","on","FaceAlpha",0.5)
```

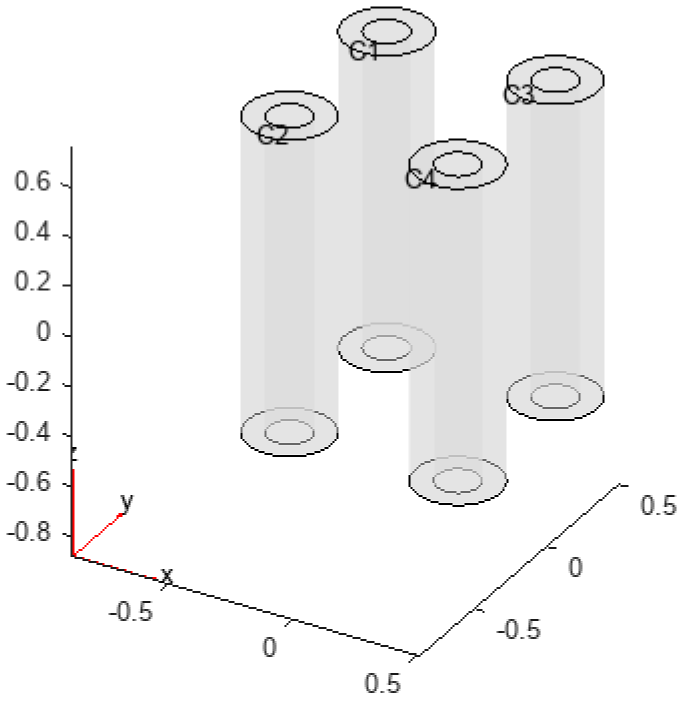


Scale and move the second geometry to fit entirely within the cube `g1`.

```
g2 = scale(g2,[1/1500 1/1500 1/100]);
g2 = translate(g2,[-0.5 -0.5 -0.5]);
```

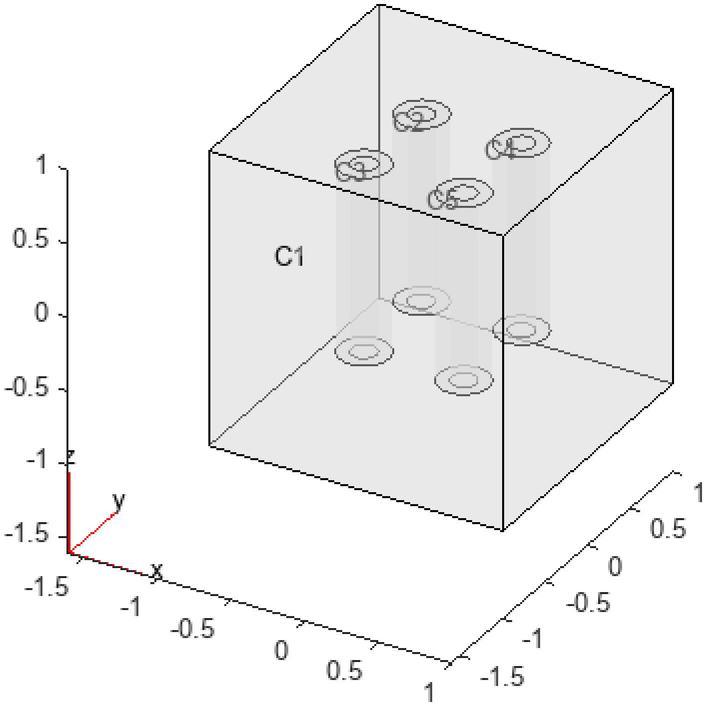
Plot the result.

```
pdegplot(g2,"CellLabels","on","FaceAlpha",0.5)
```



Combine the geometries and plot the result. The combined geometry `g3` contains cells from both geometries. The cell from `g1` keeps its ID, and the cells from `g2` are now C2, C3, C4, and C5.

```
g3 = addCell(g1,g2);
pdegplot(g3,"CellLabels","on","FaceAlpha",0.4)
```



Input Arguments

g1 — 3-D geometry

`DiscreteGeometry` object

3-D geometry, specified as a `DiscreteGeometry` object. For more information, see DiscreteGeometry.

g2 — 3-D geometry

`DiscreteGeometry` object

3-D geometry, specified as a `DiscreteGeometry` object. For more information, see DiscreteGeometry.

Output Arguments

g3 — Resulting 3-D geometry

`DiscreteGeometry` object

Resulting 3-D geometry, returned as a `DiscreteGeometry` object. For more information, see DiscreteGeometry.

Version History

Introduced in R2021a

See Also

`addFace` | `addVertex` | `addVoid` | DiscreteGeometry

addFace

Package: `pde`

Fill void regions in 2-D and split cells in 3-D geometry

Syntax

```
h = addFace(g,edges)
[h,FaceID] = addFace(g,edges)
```

Description

`h = addFace(g,edges)` adds a new face to the geometry `g`. The specified edges must form a closed contour. For a 2-D geometry, adding a new face lets you fill voids in the geometry. For a 3-D geometry, adding a new face lets you split one cell into multiple cells.

You can add several new faces simultaneously by specifying their contours in a cell array. Each contour in the cell array must be unique.

Note New faces modify a geometry, but they do not modify the mesh. After modifying a geometry, always call `generateMesh` to ensure a proper mesh association with the new geometry.

`[h,FaceID] = addFace(g,edges)` also returns a row vector containing IDs of the added faces.

Examples

Fill Void Region in 2-D Geometry

Add a face to a 2-D geometry to fill an internal void.

Create a PDE model.

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

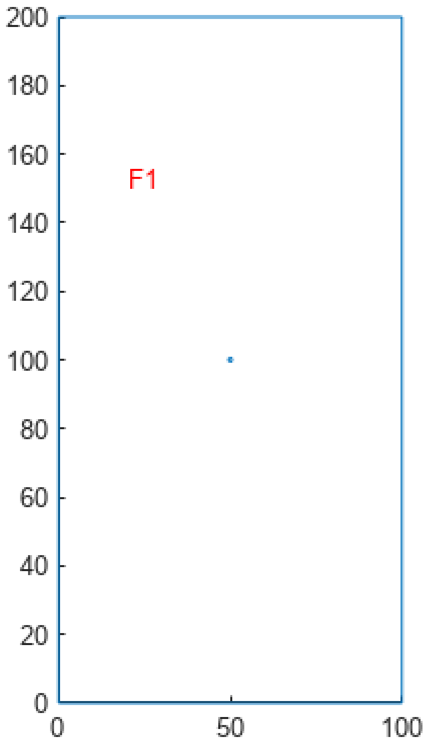
Import the geometry. This geometry has one face.

```
gm = importGeometry(model,"PlateSquareHolePlanar.stl")
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 0
       NumFaces: 1
       NumEdges: 8
    NumVertices: 8
       Vertices: [8x3 double]
```

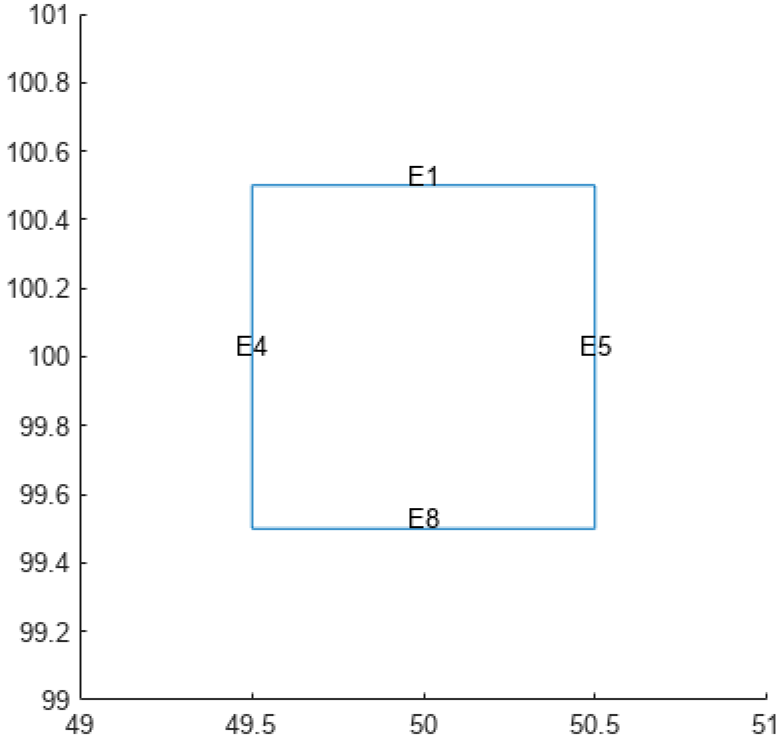
Plot the geometry and display the face labels.

```
pdegplot(gm,"FaceLabels","on")
```



Zoom in and display the edge labels of the small hole at the center.

```
figure
pdegplot(gm,"EdgeLabels","on")
axis([49 51 99 101])
```



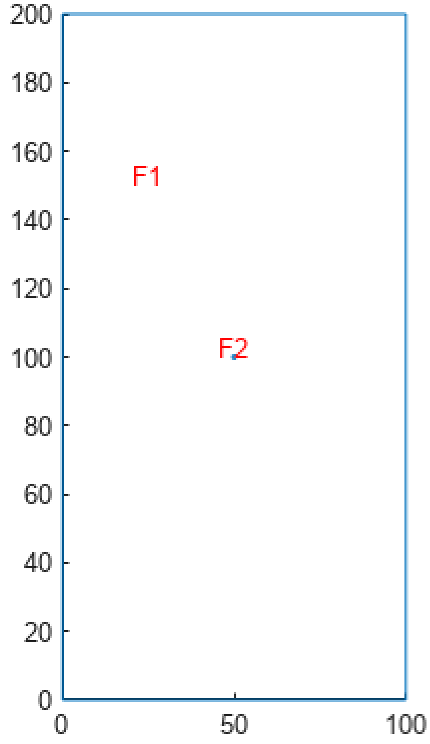
Fill the hole by adding a face. The number of faces in the geometry changes to 2.

```
gm = addFace(gm,[1 8 4 5])
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 0
       NumFaces: 2
       NumEdges: 8
    NumVertices: 8
       Vertices: [8x3 double]
```

Plot the modified geometry and display the face labels.

```
pdegplot(gm,"FaceLabels","on")
```



Split Cells in 3-D Geometry

Add a face in a 3-D geometry to split a cell into two cells.

Create a PDE model.

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

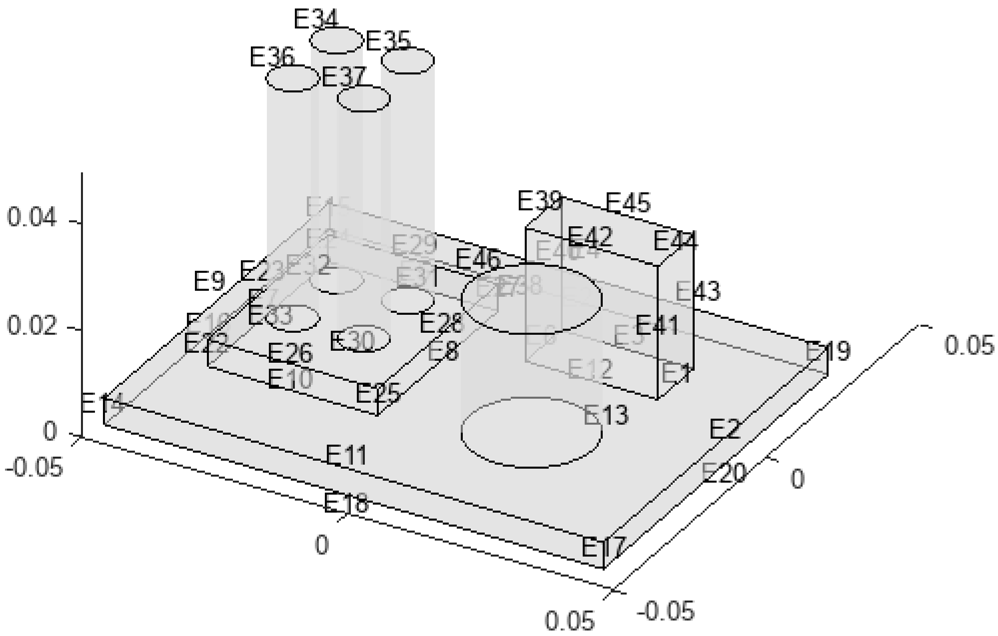
Import the geometry. The geometry consists of one cell.

```
gm = importGeometry(model,"MotherboardFragment1.stl")
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 26
       NumEdges: 46
    NumVertices: 34
       Vertices: [34x3 double]
```

Plot the geometry and display the edge labels. Zoom in on the corresponding part of the geometry to see the edge labels there more clearly.

```
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.5)

xlim([-0.05 0.05])
ylim([-0.05 0.05])
zlim([0 0.05])
```



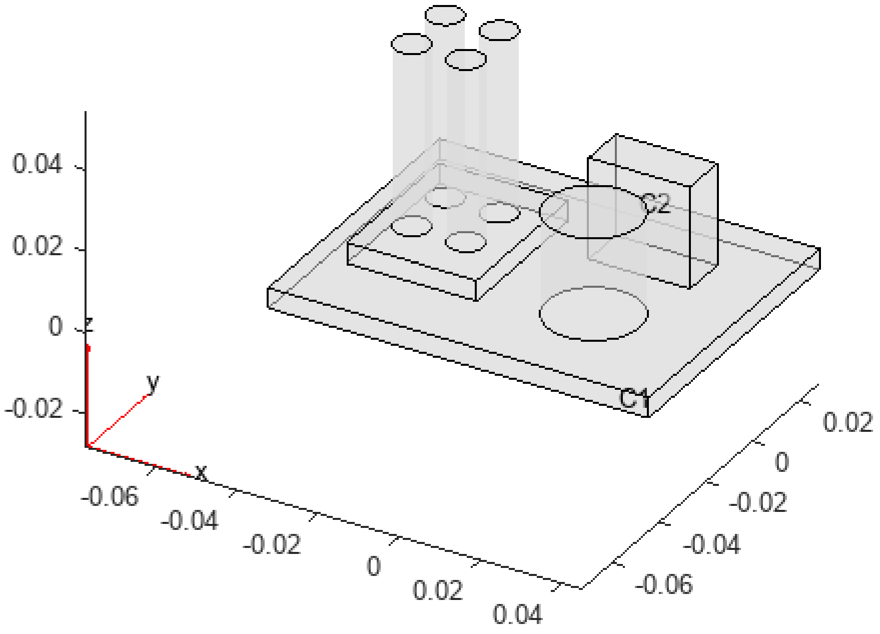
Split the cuboid on the right side into a separate cell. For this, add a face bounded by edges 1, 3, 6, and 12.

```
[gm,ID] = addFace(gm,[1 3 6 12])
```

```
gm =
  DiscreteGeometry with properties:

       NumCells: 2
       NumFaces: 27
       NumEdges: 46
    NumVertices: 34
       Vertices: [34x3 double]


ID = 27
```

Plot the modified geometry and display the cell labels.

```
pdegplot(gm,"CellLabels","on","FaceAlpha",0.5)
```



Now split the cuboid on the left side of the board and all cylinders into separate cells by adding a face at the bottom of each shape. To see edge labels more clearly, zoom and rotate the plot. Use a cell array to add several new faces simultaneously.

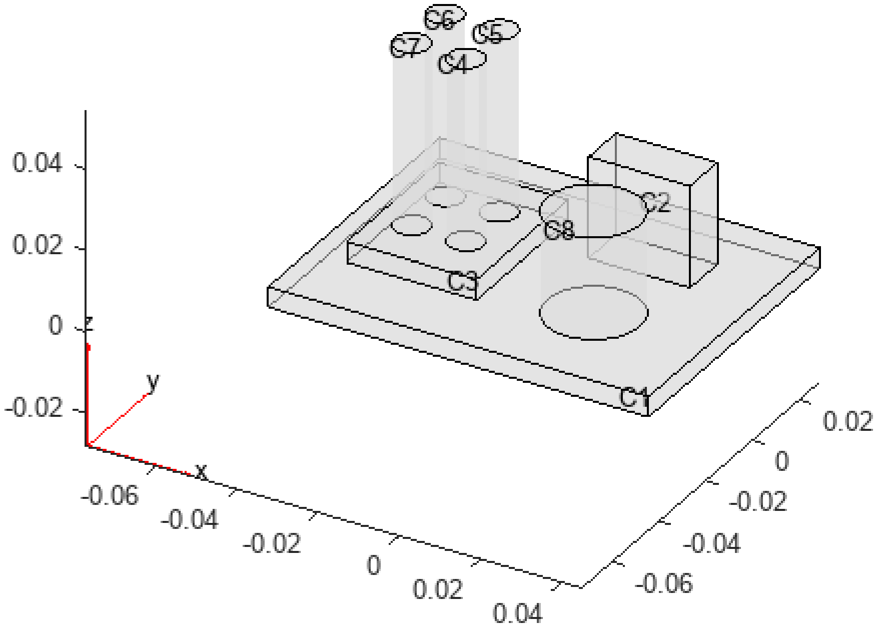
```
[gm,IDs] = addFace(gm,{[5 7 8 10], ...
                        30, ...
                        31, ...
                        32, ...
                        33, ...
                        13})
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 8
       NumFaces: 33
       NumEdges: 46
    NumVertices: 34
       Vertices: [34x3 double]


IDs = 6×1

    28
    29
    30
    31
    32
```

```
    33
```

Plot the modified geometry and display the cell labels.

```
pdegplot(gm,"CellLabels","on","FaceAlpha",0.5)
```



Input Arguments

g — Geometry
`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

edges — Edges forming unique closed flat contour
vector of positive integers | cell array of vectors of positive integers

Edges forming a unique closed flat contour, specified as a vector of positive integers or a cell array of such vectors. You can specify edges within a vector in any order.

When you use a cell array to add several new faces, each contour in the cell array must be unique.

Example: `addFace(g,[1 3 4 7])`

Output Arguments

`h` — Resulting geometry
handle

Resulting geometry, returned as a handle.

`FaceID` — Face ID
positive number | row vector of positive numbers

Face ID, returned as a positive number or a row vector of positive numbers. Each number represents a face ID. When you add a new face to a geometry with N faces, the ID of the added face is $N + 1$.

Tips

- `addFace` errors when the specified contour defines an already existing face.
- `addFace` always modifies the original geometry `g`.

Version History

Introduced in R2020a

Face addition for analytic geometries

`addFace` now lets you fill void regions in 2-D `AnalyticGeometry` objects.

See Also

DiscreteGeometry Properties | AnalyticGeometry Properties | `addVertex` | `pdegplot` | `importGeometry` | `geometryFromMesh` | `generateMesh` | `structuralBoundaryLoad` | `structuralBC`

addVertex

Package: `pde`

Add a vertex on a geometry boundary

Syntax

```
VertexID = addVertex(g,"Coordinates",Coords)
```

Description

`VertexID = addVertex(g,"Coordinates",Coords)` adds a new isolated vertex at the point with coordinates `Coords` to a boundary of the geometry `g`. To add several vertices simultaneously, specify `Coords` as an `N`-by-2 matrix for a 2-D geometry or an `N`-by-3 matrix for a 3-D geometry. Here, `N` is the number of new points.

If a point with the specified coordinates is slightly offset (within an internally specified tolerance) from a geometry boundary, `addVertex` approximates it to a point on the boundary. If a vertex already exists at the specified location, `addVertex` returns the ID of the existing vertex instead of creating one.

Examples

Add Vertices on Edge of Block

Use `addVertex` to add a single vertex and multiple vertices on a side of a block geometry.

Create a PDE model.

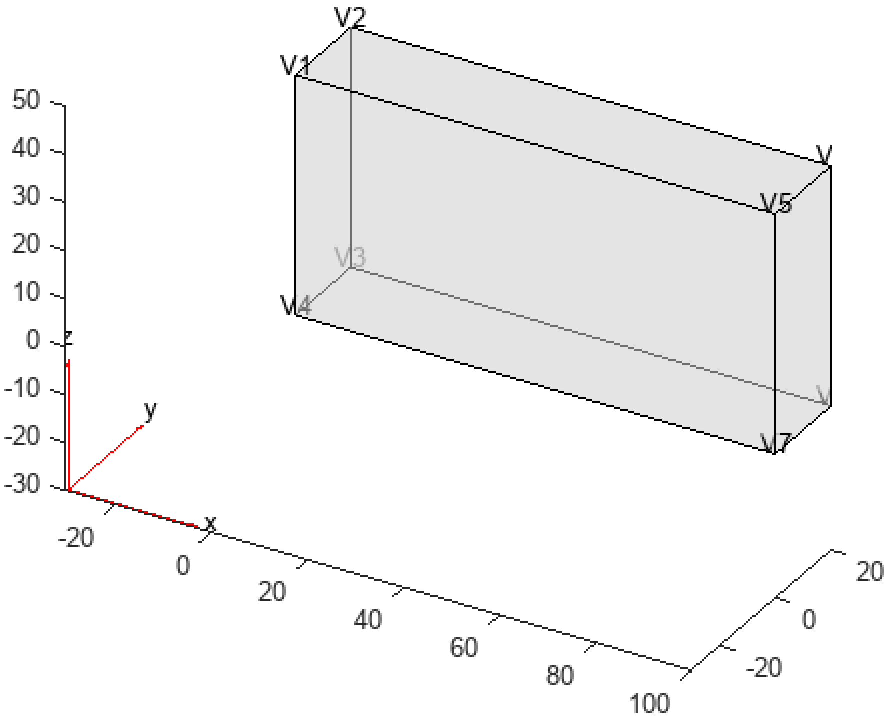
```
model = createpde();
```

Import the geometry.

```
g = importGeometry(model,"Block.stl");
```

Plot the geometry and display the vertex labels.

```
pdegplot(g, "VertexLabels", "on","FaceAlpha",0.5)
```



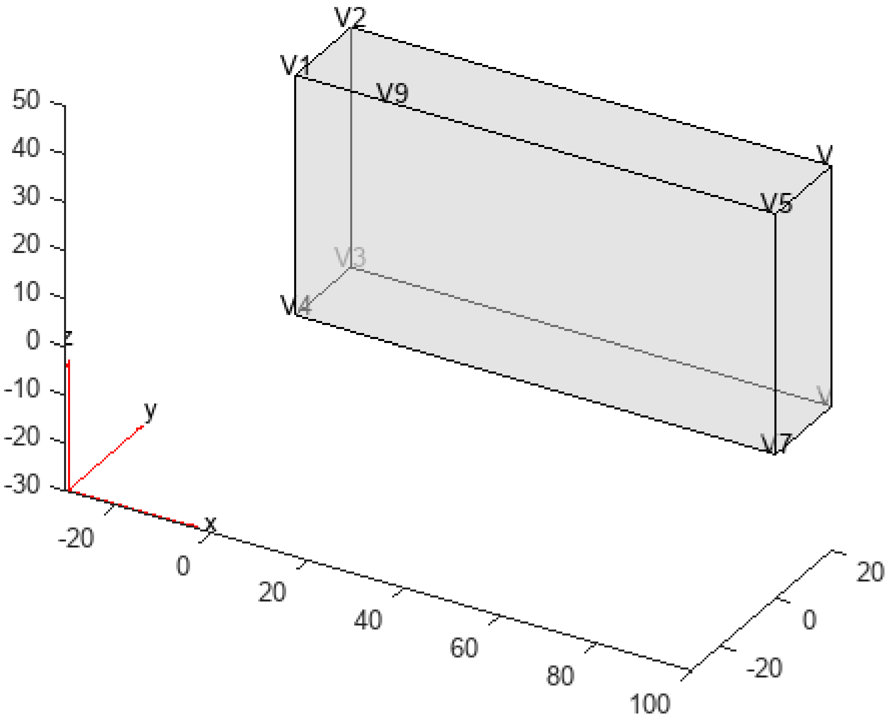
Add a vertex on the edge of a block.

```
VertexID = addVertex(g,"Coordinates",[20 0 50])
```

```
VertexID = 9
```

Plot the geometry and display the vertex labels.

```
pdegplot(g, "VertexLabels", "on","FaceAlpha",0.5)
```



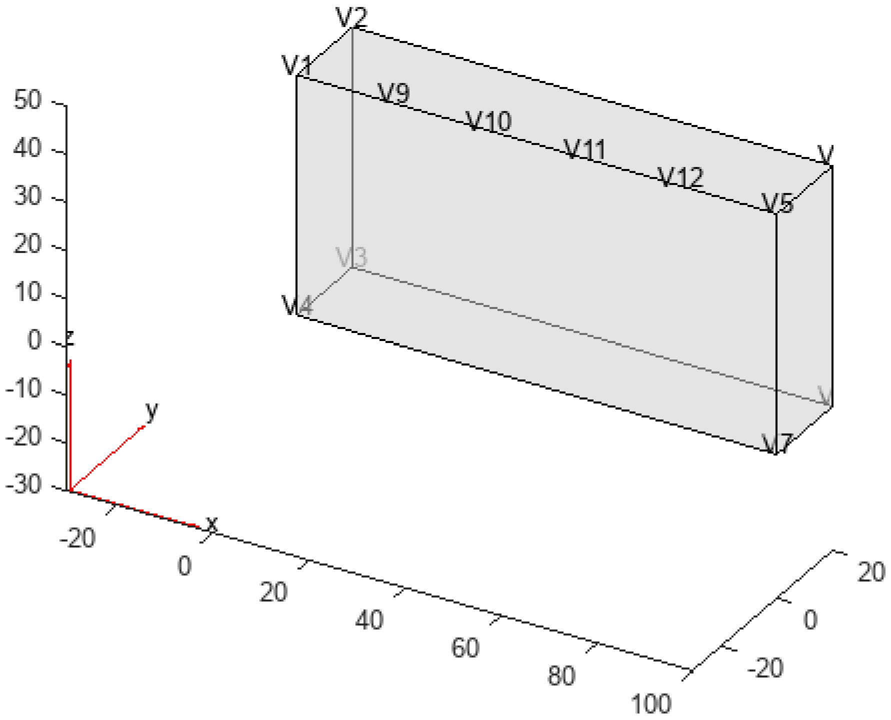
Add three more vertices on the same edge of the block.

```
V = ([40 0 50; 60 0 50; 80 0 50]);
VertexIDs = addVertex(g,"Coordinates",V)
```

```
VertexIDs = 3×1

    10
    11
    12
```

Plot the geometry and display the vertex labels.

```
pdegplot(g, "VertexLabels", "on","FaceAlpha",0.5)
```



Add a vertex at the corner of the block. Since there is already a vertex at the corner, `addVertex` does not create a new vertex, but returns the ID of the existing vertex.

```
VertexID = addVertex(g,"Coordinates",[100 0 50])
```

```
VertexID = 5
```

Input Arguments

g — Geometry
`DiscreteGeometry` object

Geometry, specified as a `DiscreteGeometry` object.

`Coords` — Coordinates of new vertex
N-by-2 numeric matrix | N-by-3 numeric matrix

Coordinates of a new vertex, specified as an `N`-by-2 or `N`-by-3 numeric matrix for a 2-D or 3-D geometry, respectively. Here, `N` is the number of new vertices.

Example: `"Coordinates",[0;0;1]`

Data Types: `double`

Output Arguments

`VertexID` — Vertex ID

row vector

Vertex ID, returned as a row vector of positive numbers. Each number represents a vertex ID. When you add a new vertex to a geometry with `N` vertices, the ID of the added vertex is `N + 1`. If a vertex already exists at the specified location, `addVertex` returns the ID of the existing vertex.

Limitations

- `addVertex` does not work with `AnalyticGeometry` objects. See AnalyticGeometry.

Version History

Introduced in R2019b

See Also

DiscreteGeometry Properties | `addFace` | `pdegplot` | `importGeometry` | `geometryFromMesh` | `generateMesh` | `structuralBoundaryLoad` | `structuralBC`

addVoid

Create void regions inside 3-D geometry

Syntax

```
g3 = addVoid(g1,g2)
```

Description

`g3 = addVoid(g1,g2)` creates void regions inside `g1` using all cells of `g2`. All cells of the geometry `g2` must be contained inside one cell of the geometry `g1`. Ensure that the geometries do not have enclosed cavities and do not intersect one another.

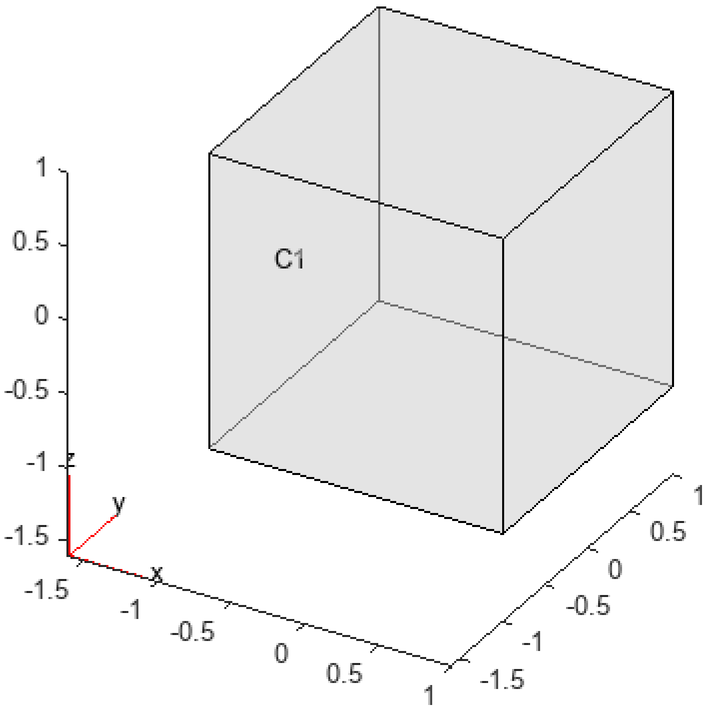
Note Added void regions modify a geometry, but they do not modify the corresponding mesh. After modifying a geometry, always call `generateMesh` to ensure a proper mesh association with the new geometry.

Examples

Add Void Regions Inside Cube

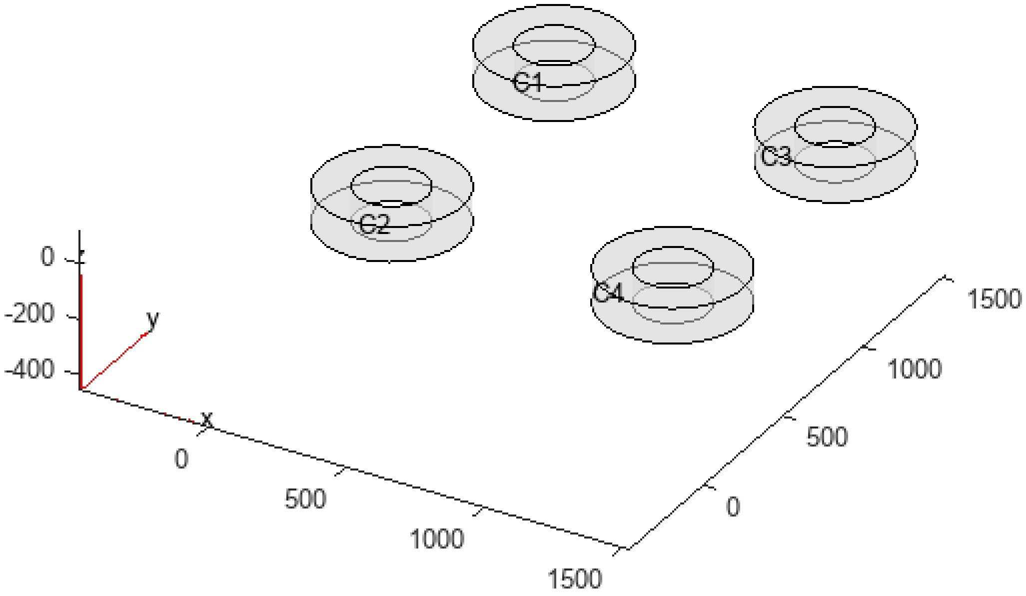
Create and plot a geometry.

```
g1 = multicuboid(2,2,2,"Zoffset",-1);
pdegplot(g1,"CellLabels","on","FaceAlpha",0.5)
```



Import and plot another geometry.

```
g2 = importGeometry("DampingMounts.stl");
pdegplot(g2,"CellLabels","on","FaceAlpha",0.5)
```

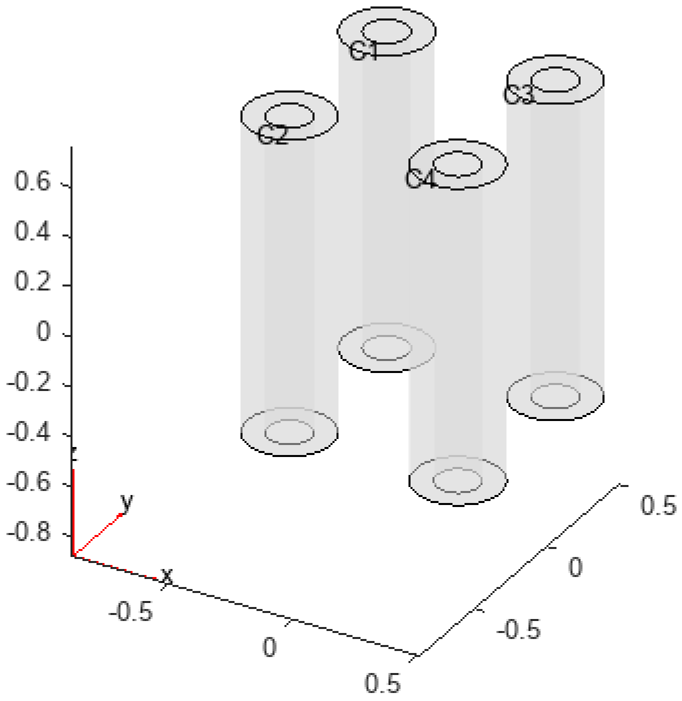


Scale and move the second geometry to fit entirely within the cube `g1`.

```
g2 = scale(g2,[1/1500 1/1500 1/100]);
g2 = translate(g2,[-0.5 -0.5 -0.5]);
```

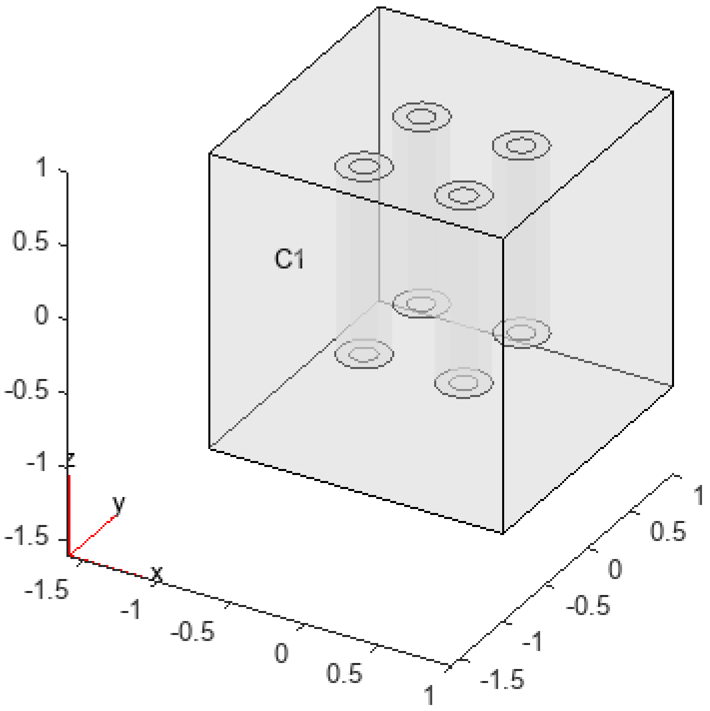
Plot the result.

```
pdegplot(g2,"CellLabels","on","FaceAlpha",0.5)
```



Create void regions inside the cube using the cells of the geometry `g2`. Plot the result.

```
g3 = addVoid(g1,g2);
pdegplot(g3,"CellLabels","on","FaceAlpha",0.4)
```



Input Arguments

g1 — 3-D geometry
`DiscreteGeometry` object

3-D geometry, specified as a `DiscreteGeometry` object. For more information, see DiscreteGeometry.

g2 — 3-D geometry
`DiscreteGeometry` object

3-D geometry, specified as a `DiscreteGeometry` object. For more information, see DiscreteGeometry.

Output Arguments

g3 — Resulting 3-D geometry
`DiscreteGeometry` object

Resulting 3-D geometry, returned as a `DiscreteGeometry` object. For more information, see DiscreteGeometry.

Version History

Introduced in R2021a

See Also

`addCell` | `addFace` | `addVertex` | DiscreteGeometry

AnalyticGeometry Properties

Analytic 2-D geometry description

Description

`AnalyticGeometry` describes a 2-D geometry in the form of an analytic geometry object. `PDEModel`, `StructuralModel`, and `ThermalModel` objects have a `Geometry` property, which can be an `AnalyticGeometry` or `DiscreteGeometry` object.

Add a 2-D analytic geometry to your model by using `decsg` to create the geometry and `geometryFromEdges` to attach it to the model.

Properties

Properties

`NumEdges` — Number of geometry edges
positive integer

Number of geometry edges, specified as a positive integer.

Data Types: `double`

`NumFaces` — Number of geometry faces
positive integer

Number of geometry faces, specified 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, specified as a positive integer.

Data Types: `double`

`Vertices` — Coordinates of geometry vertices
`N`-by-2 numeric matrix

Coordinates of geometry vertices, specified as an `N`-by-2 numeric matrix where `N` is the number of vertices.

Data Types: `double`

Version History

Introduced in R2015a

See Also

`geometryFromEdges` | `decsg` | `PDEModel` | `StructuralModel` | `ThermalModel` | DiscreteGeometry Properties

Topics

“Solve Problems Using PDEModel Objects” on page 2-2

applyBoundaryCondition

Package: `pde`

Add boundary condition to `PDEModel` container

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

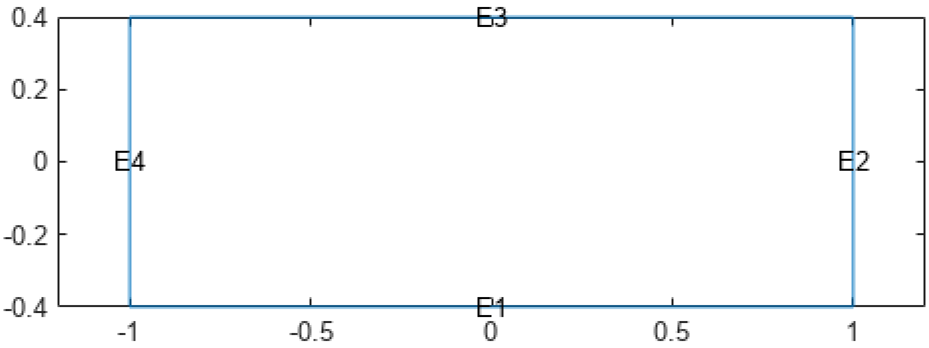
Dirichlet Boundary Conditions

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);
```

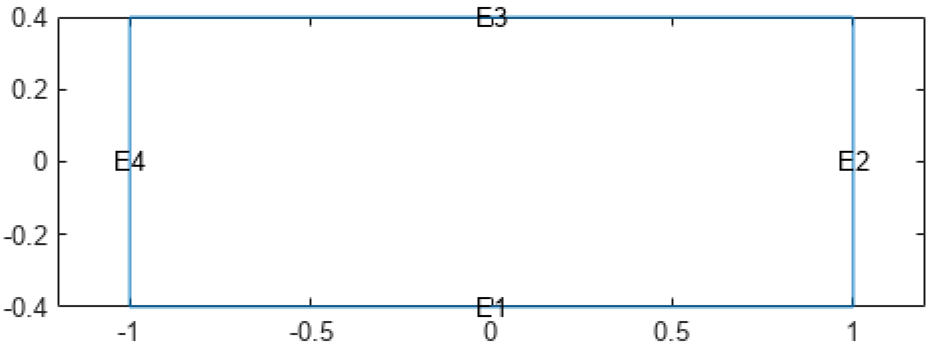
Neumann Boundary Conditions

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]);
```

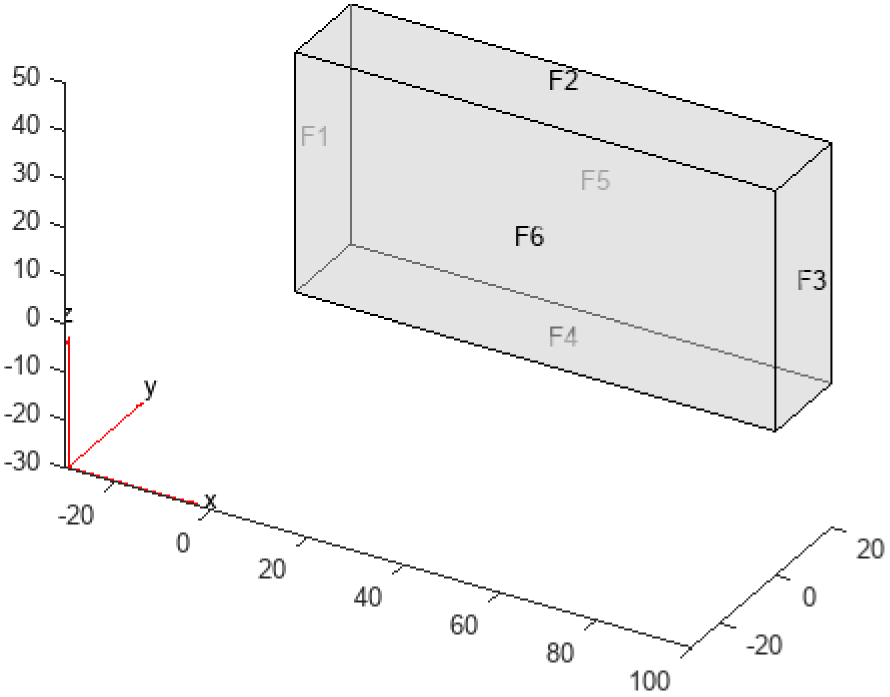
Dirichlet and Neumann Boundary Conditions for Different Boundaries

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.

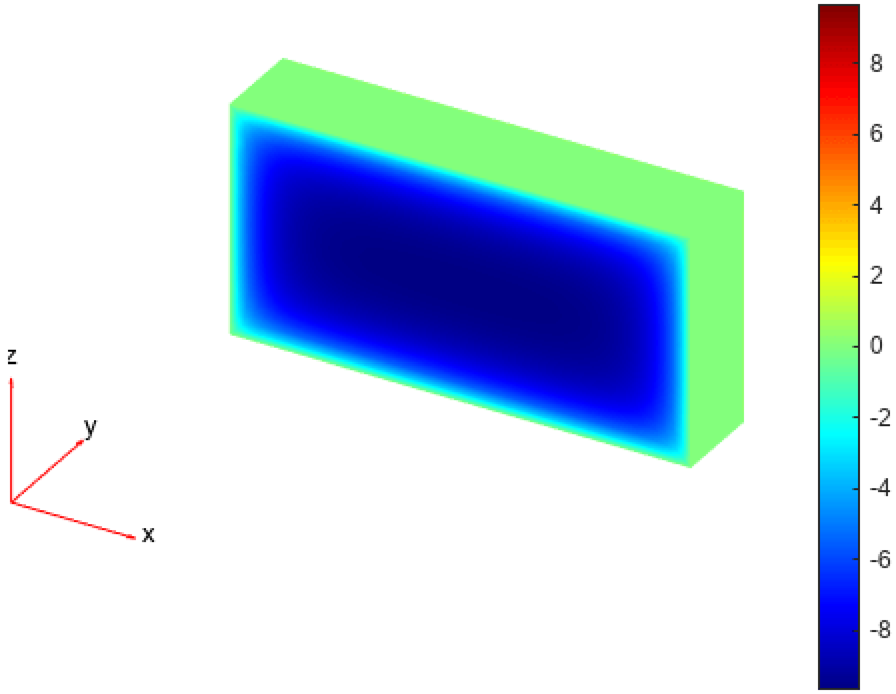
```
applyBoundaryCondition(model,"dirichlet", ...
                             "Face",1:4,"u",0);
```

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)
```



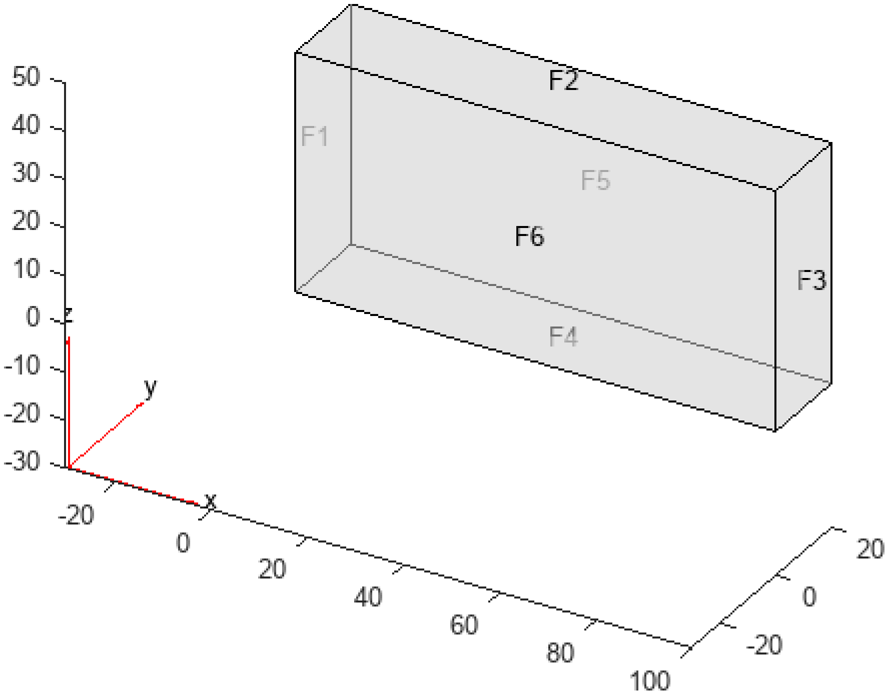
Individual Boundary Conditions for Equations in a System

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]);
```

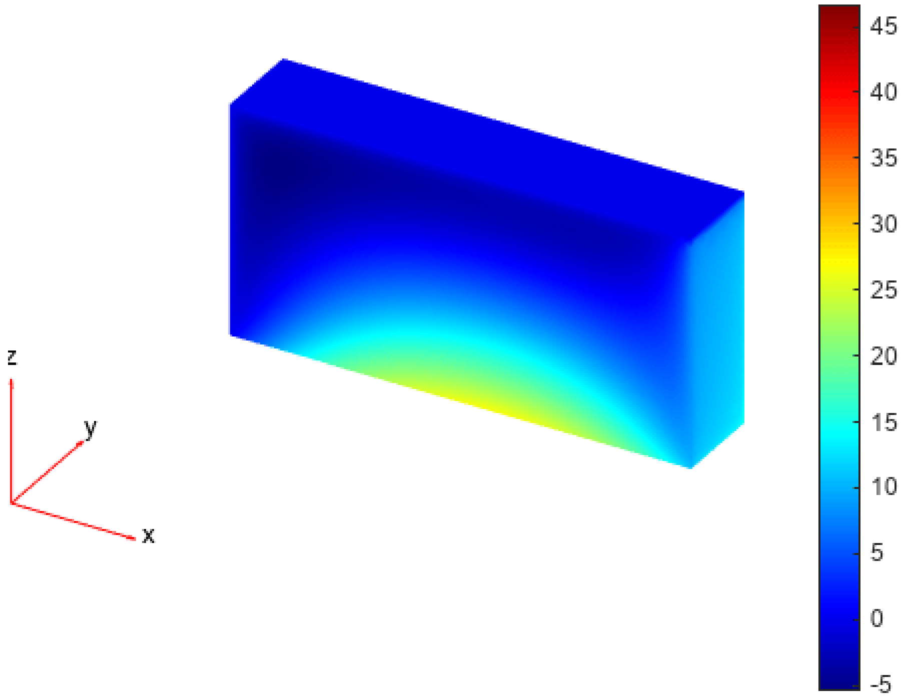
For face 3, apply generalized Neumann boundary condition for the first equation and Dirichlet boundary conditions for the second and third equations.

```
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`

`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` | `string`

`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 2-132.

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 2-132.

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 2-132.

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 2-132.

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 2-132.

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.

When using `EquationIndex` to specify Dirichlet boundary conditions for a subset of components, use the `mixed` argument instead of `dirichlet`. The remaining components satisfy the default Neumann boundary condition with the zero values for `"g"` and `"q"`.

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". For details of this evaluation, see "Nonconstant Boundary Conditions" on page 2-132.

Example: `applyBoundaryCondition(model,"dirichlet","Face",[2,4,11],"u",@ucalculator,"Vectorized","on")`

Data Types: `char` | `string`

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.

- 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"`.

Version History

Introduced in R2015a

See Also

`findBoundaryConditions` | BoundaryCondition | `PDEModel`

Topics

"Specify Boundary Conditions" on page 2-129
"Solve PDEs with Constant Boundary Conditions" on page 2-135
"Specify Nonconstant Boundary Conditions" on page 2-139
"No Boundary Conditions Between Subdomains" on page 2-126
"Identify Boundary Labels" on page 2-128
"Solve Problems Using PDEModel Objects" on page 2-2

area

Package: `pde`

Area of 2-D mesh elements

Syntax

```
A = area(mesh)
[A,AE] = area(mesh)
A = area(mesh,elements)
```

Description

`A = area(mesh)` returns the area `A` of the entire mesh.

`[A,AE] = area(mesh)` also returns a row vector `AE` containing areas of each individual element of the mesh.

`A = area(mesh,elements)` returns the combined area of the specified elements of the mesh.

Examples

Area of Entire 2-D Mesh

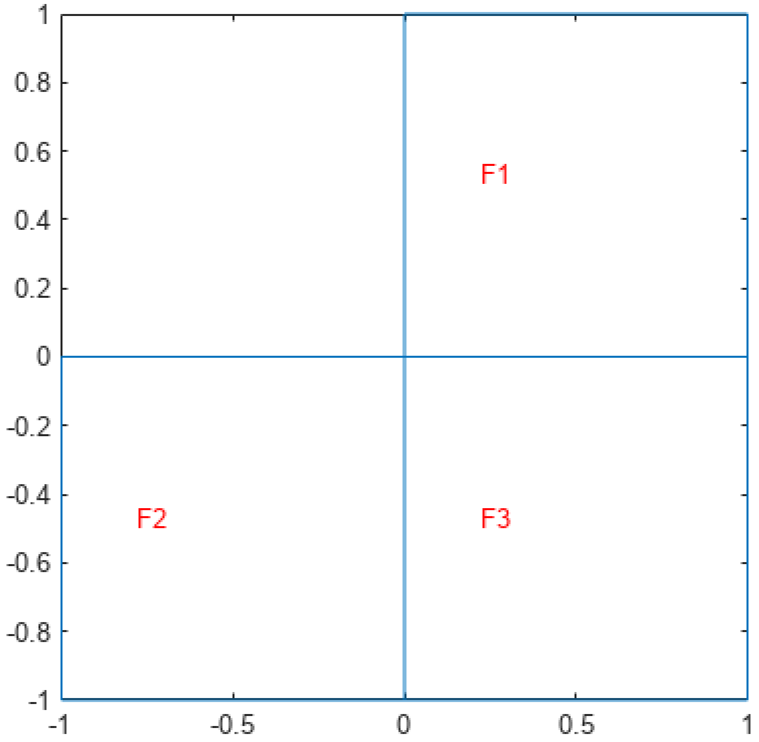
Generate a 2-D mesh and find its area.

Create a PDE model.

```
model = createpde;
```

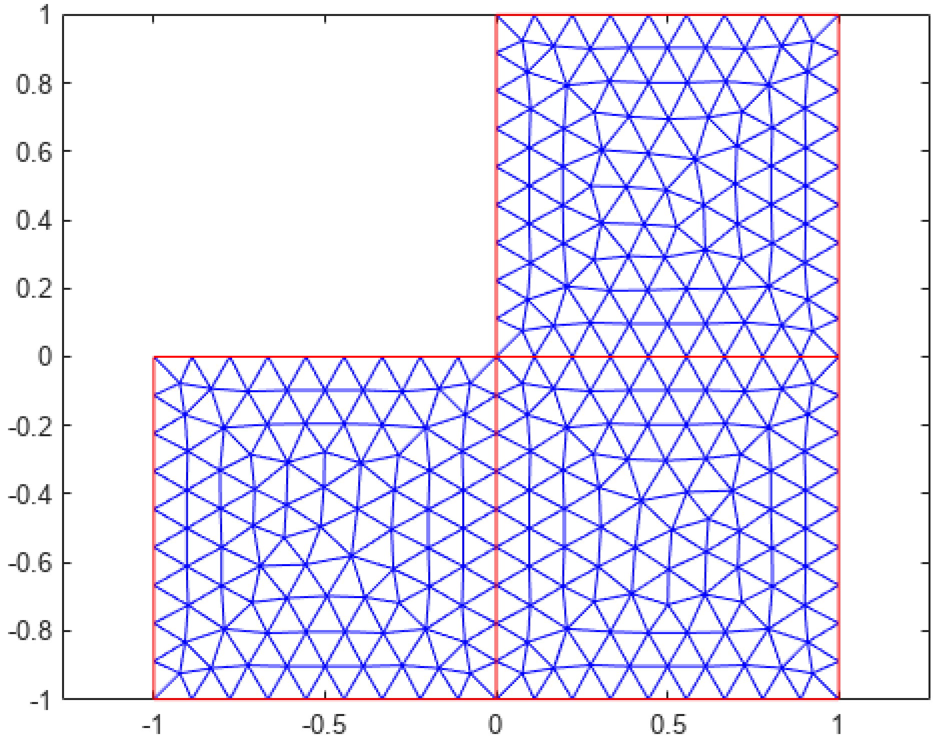
Include the geometry of the built-in function `lshapeg`. Plot the geometry.

```
geometryFromEdges(model,@lshapeg);
pdegplot(model,"FaceLabels","on")
```



Generate a mesh and plot it.

```
mesh = generateMesh(model);
figure
pdemesh(model)
```



Compute the area of the entire mesh.

```
ma = area(mesh)
```

```
ma = 3.0000
```

Area of Individual Elements of 2-D Mesh

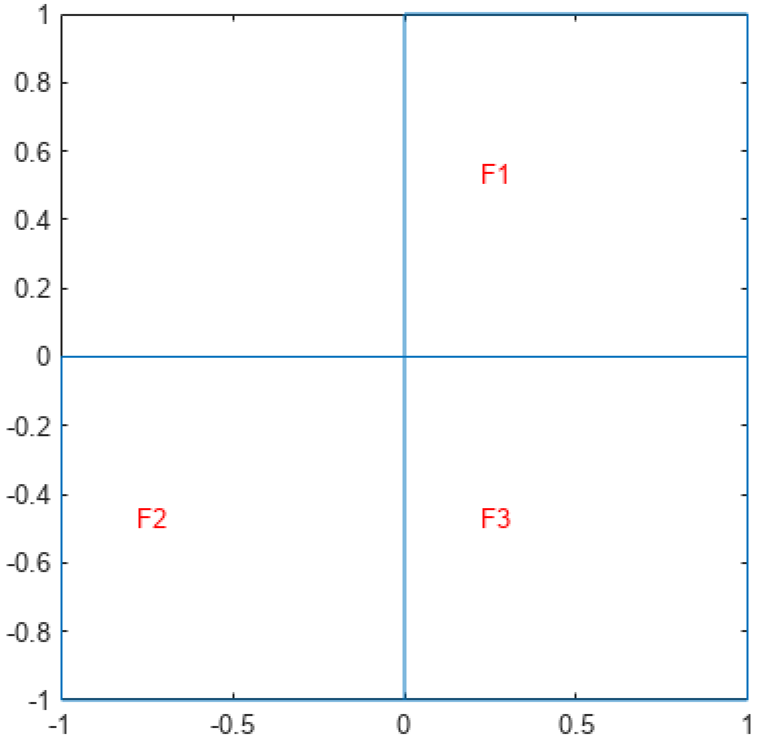
Generate a 2-D mesh and find the area of each element.

Create a PDE model.

```
model = createpde;
```

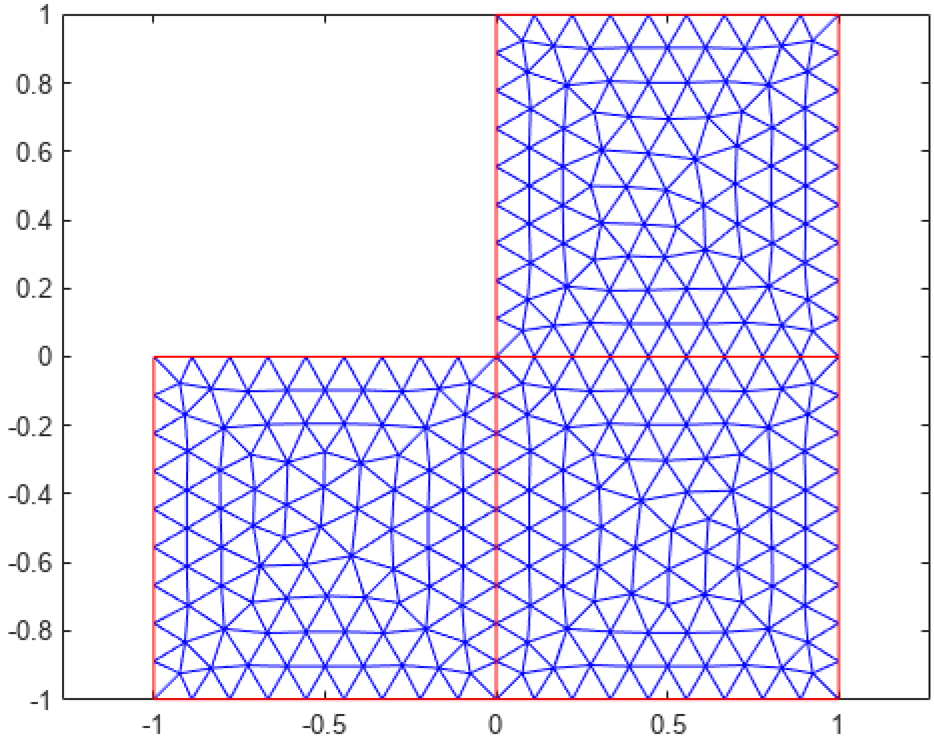
Include the geometry of the built-in function `lshapeg`. Plot the geometry.

```
geometryFromEdges(model,@lshapeg);
pdegplot(model,"FaceLabels","on")
```



Generate a mesh and plot it.

```
mesh = generateMesh(model);
figure
pdemesh(model)
```



Compute the area of the entire mesh and the area of each individual element of the mesh. Display the areas of the first 5 elements.

```
[ma,mi] = area(mesh);
mi(1:5)
```

```
ans = 1×5

    0.0047    0.0054    0.0053    0.0048    0.0061
```

Total Area of Group of Elements

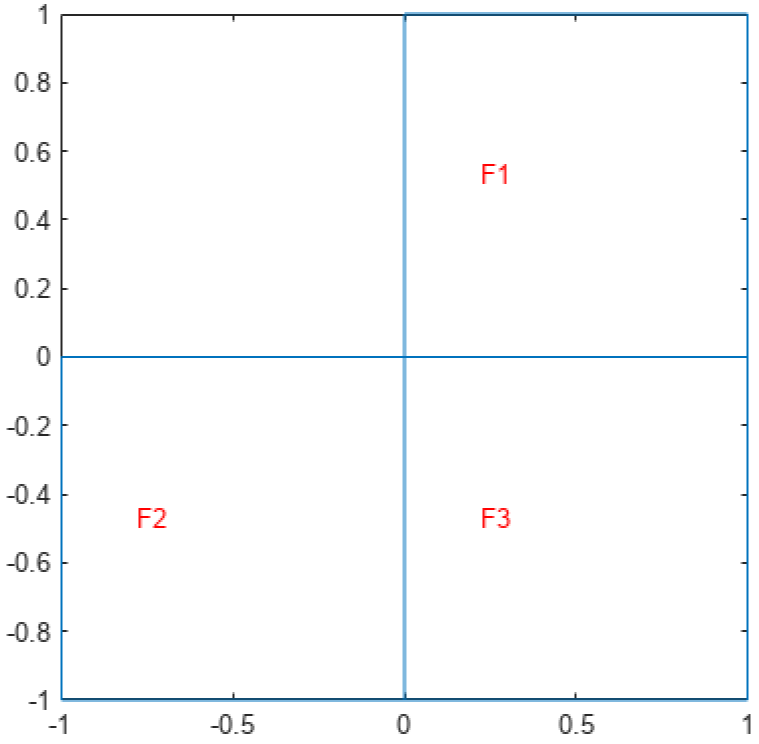
Find the combined area of the elements associated with a particular face of a 2-D mesh.

Create a PDE model.

```
model = createpde;
```

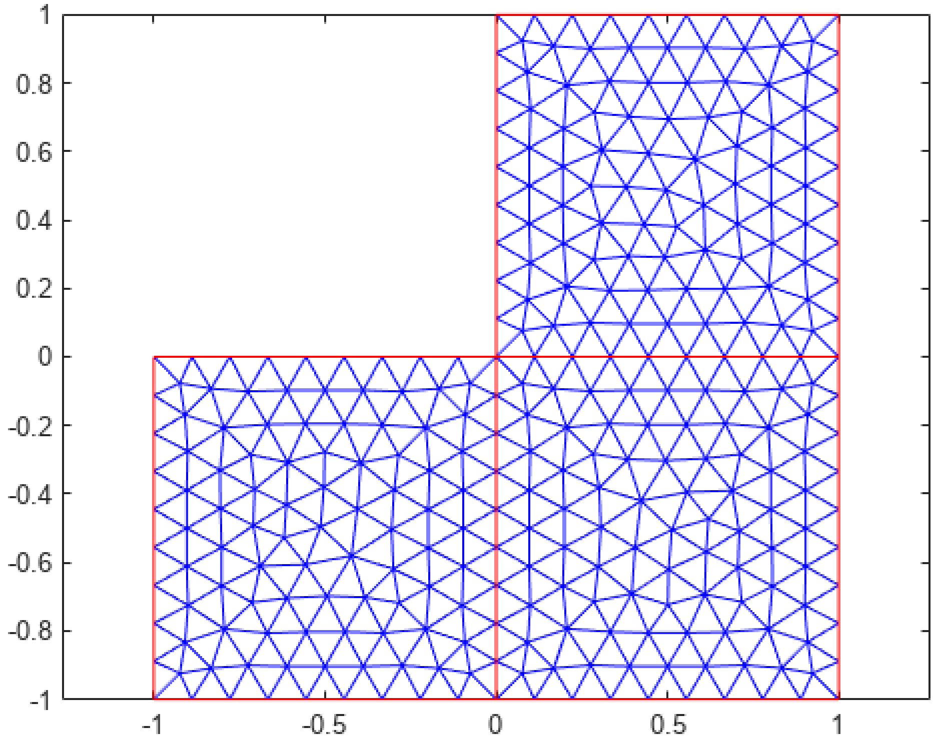
Include the geometry of the built-in function `lshapeg`. Plot the geometry.

```
geometryFromEdges(model,@lshapeg);
pdegplot(model,"FaceLabels","on")
```



Generate a mesh and plot it.

```
mesh = generateMesh(model);
figure
pdemesh(model)
```



Find the elements associated with face 1 and compute the total area of these elements.

```
Ef1 = findElements(mesh,"region","Face",1);
maf1 = area(mesh,Ef1)
```

```
maf1 = 1.0000
```

Find how much of the total mesh area belongs to these elements. Return the result as a percentage.

```
maf1_percent = maf1/area(mesh)*100
```

```
maf1_percent = 33.3333
```

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`

`elements` — Element IDs

positive integer | matrix of positive integers

Element IDs, specified as a positive integer or a matrix of positive integers.

Example: `[10 68 81 97 113 130 136 164]`

Output Arguments

`A` — Area

positive number

Area of the entire mesh or the combined area of the specified elements of the mesh, returned as a positive number.

`AE` — Areas of individual elements

row vector of positive numbers

Areas of individual elements, returned as a row vector of positive numbers.

Version History

Introduced in R2018a

See Also

`volume` | `findElements` | `findNodes` | `meshQuality` | FEMesh Properties

Topics

"Finite Element Method Basics" on page 1-11

assema

(Not recommended) Assemble area integral contributions

Note `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

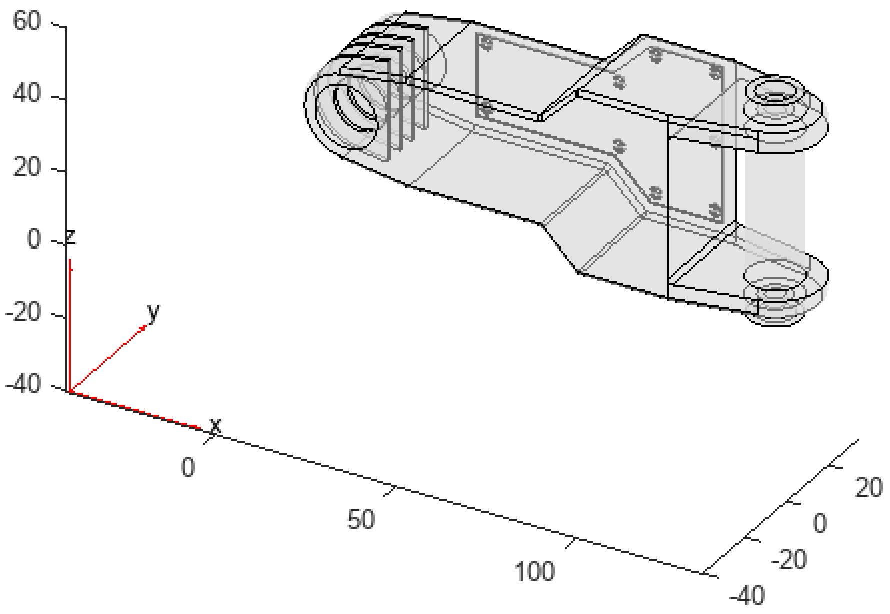
$$-\nabla \cdot (c\nabla u) + au = f\,.$$

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 matrices 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))])
```

```
Fraction of nonzero entries in K is 0.0010942
```

```
disp(['Number of nonzero entries in M is ',num2str(nnz(M))])
```

```
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))])
```

```
Fraction of nonzero entries in K is 0.042844
```

```
disp(['Number of nonzero entries in M is ',num2str(nnz(M))])
```

```
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`

`c` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `c` represents the c coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

`a` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `a` represents the a coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

`f` — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `f` represents the *f* coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `string` | `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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 5-101.

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 5-101.

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 5-101.

Typically, you use `F` in a subsequent call to `assempde`.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`assema` is not recommended. Use `assembleFEMatrices` instead. There are no plans to remove `assema`.

See Also

`assembleFEMatrices`

assemb

(Not recommended) Assemble boundary condition contributions

Note `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

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

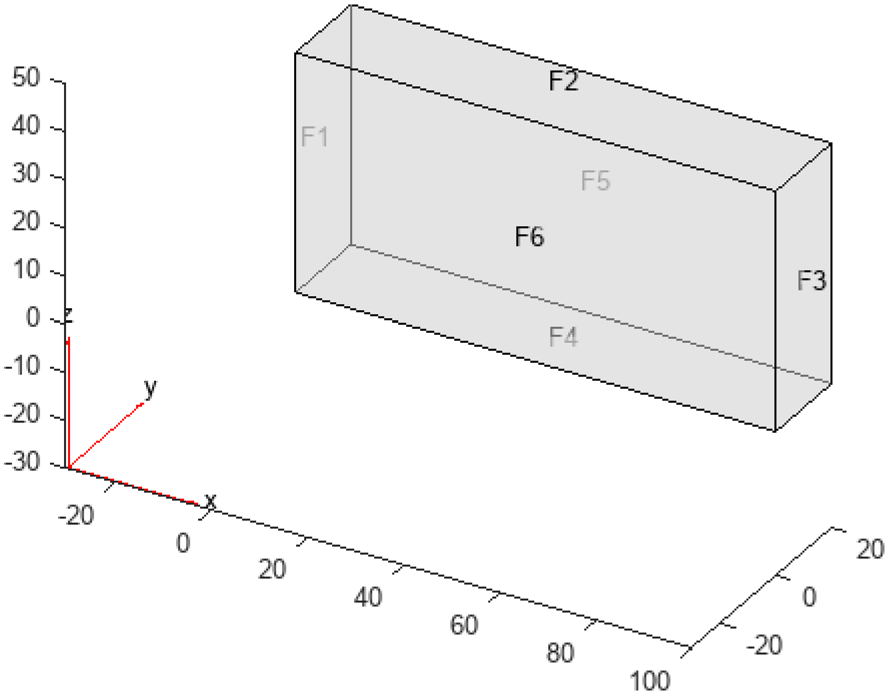
So, represent Poisson's equation in toolbox syntax by setting `c` = 1, `a` = 0, and `f` = 1.

```
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))])
```

```
Fraction of nonzero entries in H is 7.8796e-05
```

```
disp(['Number of nonzero entries in Q is ',num2str(nnz(Q))])
```

```
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`

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 Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 5-101.

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 5-101.

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 5-74.

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 5-74.

Typically, you use `R` in a subsequent call to `assempde`.

Algorithms

As explained in “Elliptic Equations” on page 5-101, 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` “More About” on page 5-101 section, and the linear algebra approach detailed in “Systems of PDEs” on page 5-107.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`assemb` is not recommended. Use `assembleFEMatrices` instead. There are no plans to remove `assemb`.

See Also

`assembleFEMatrices`

assembleFEMatrices

Assemble finite element matrices

Syntax

```
FEM = assembleFEMatrices(model)
FEM = assembleFEMatrices(model,matrices)
FEM = assembleFEMatrices(model,bcmethod)
FEM = assembleFEMatrices( ___ ,state)
```

Description

`FEM = assembleFEMatrices(model)` returns a structural array containing all finite element matrices for a PDE problem specified as a `model`.

`FEM = assembleFEMatrices(model,matrices)` returns a structural array containing only the specified finite element matrices.

`FEM = assembleFEMatrices(model,bcmethod)` assembles finite element matrices and imposes boundary conditions using the method specified by `bcmethod`.

`FEM = assembleFEMatrices( ___ ,state)` assembles finite element matrices using the input time or solution specified in the `state` structure array. The function uses the `time` field of the structure for time-dependent models and the solution field `u` for nonlinear models. You can use this argument with any of the previous syntaxes.

Examples

Finite Element Matrices for 2-D Problem

Create a PDE model for the Poisson equation on an 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: [401x401 double]
    A: [401x401 double]
    F: [401x1 double]
    Q: [401x401 double]
    G: [401x1 double]
```

```
    H: [80x401 double]
    R: [80x1 double]
    M: [401x401 double]
```

Specified Set of Finite Element Matrices

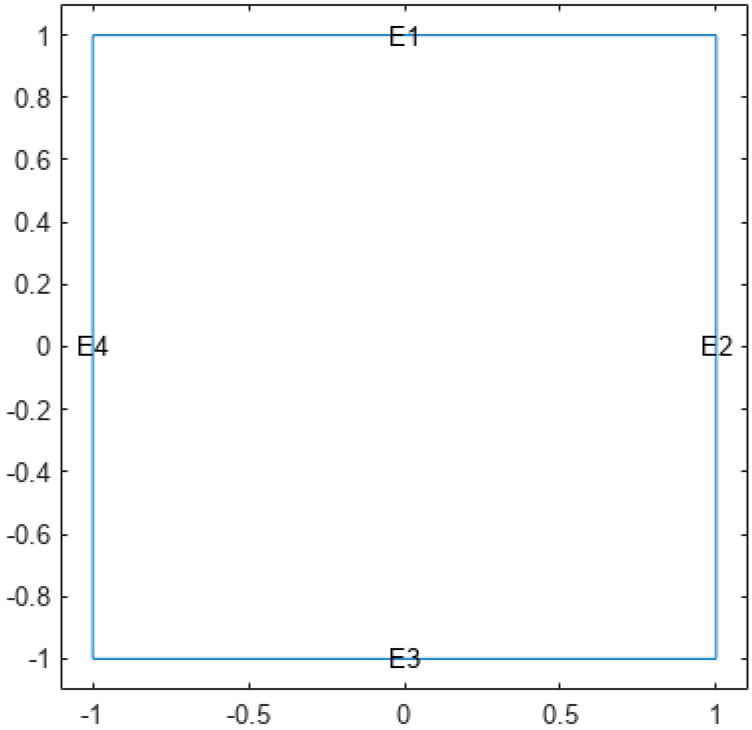
Make computations faster by specifying which finite element matrices to assemble.

Create a transient thermal model and include the geometry of the built-in function `squareg`.

```
thermalmodel = createpde("thermal","steadystate");
geometryFromEdges(thermalmodel,@squareg);
```

Plot the geometry with the edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Specify the thermal conductivity of the material and the internal heat source.

```
thermalProperties(thermalmodel,"ThermalConductivity",0.2);
internalHeatSource(thermalmodel,10);
```

Set the boundary conditions.

```
thermalBC(thermalmodel,"Edge",[1,3],"Temperature",100);
```

Generate a mesh.

```
generateMesh(thermalmodel);
```

Assemble the stiffness and mass matrices.

```
FEM_KM = assembleFEMatrices(thermalmodel,"KM")
```

```
FEM_KM = struct with fields:
    K: [1541x1541 double]
    M: [1541x1541 double]
```

Now, assemble the finite element matrices M, K, A, and F.

```
FEM_MKAF = assembleFEMatrices(thermalmodel,"MKAF")
```

```
FEM_MKAF = struct with fields:
    M: [1541x1541 double]
    K: [1541x1541 double]
    A: [1541x1541 double]
    F: [1541x1 double]
```

The four matrices M, K, A, and F correspond to discretized versions of the PDE coefficients m, c, a, and f. These four matrices also represent the domain of the finite-element model of the PDE. Instead of specifying them explicitly, you can use the `domain` argument.

```
FEMd = assembleFEMatrices(thermalmodel,"domain")
```

```
FEMd = struct with fields:
    M: [1541x1541 double]
    K: [1541x1541 double]
    A: [1541x1541 double]
    F: [1541x1 double]
```

The four matrices Q, G, H, and R, correspond to discretized versions of q, g, h, and r in the Neumann and Dirichlet boundary condition specification. These four matrices also represent the boundary of the finite-element model of the PDE. Use the `boundary` argument to assemble only these matrices.

```
FEMb = assembleFEMatrices(thermalmodel,"boundary")
```

```
FEMb = struct with fields:
    H: [74x1541 double]
    R: [74x1 double]
    G: [1541x1 double]
    Q: [1541x1541 double]
```

Finite Element Matrices with `nullspace` and `stiff-spring` Methods

Create a PDE model for the Poisson equation on an 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.

```
generateMesh(model,"Hmax",0.2);
```

Obtain the finite element matrices after imposing the boundary condition using the null-space approach. This approach eliminates the Dirichlet degrees of freedom and provides a reduced system of equations.

```
FEMn = assembleFEMatrices(model,"nullspace")
```

```
FEMn = struct with fields:
    Kc: [321x321 double]
    Fc: [321x1 double]
     B: [401x321 double]
    ud: [401x1 double]
     M: [321x321 double]
```

Obtain the solution to the PDE using the `nullspace` finite element matrices.

```
un = FEMn.B*(FEMn.Kc\FEMn.Fc) + FEMn.ud;
```

Compare this result to the solution given by `solvepde`. The two solutions are identical.

```
u1 = solvepde(model);
norm(un - u1.NodalSolution)
```

```
ans = 0
```

Obtain the finite element matrices after imposing the boundary condition using the stiff-spring approach. This approach retains the Dirichlet degrees of freedom, but imposes a large penalty on them.

```
FEMs = assembleFEMatrices(model,"stiff-spring")
```

```
FEMs = struct with fields:
    Ks: [401x401 double]
    Fs: [401x1 double]
     M: [401x401 double]
```

Obtain the solution to the PDE using the stiff-spring finite element matrices. This technique gives a less accurate solution.

```
us = FEMs.Ks\FEMs.Fs;
norm(us - u1.NodalSolution)
```

```
ans = 0.0098
```

Finite Element Matrices for Time-Dependent Problem

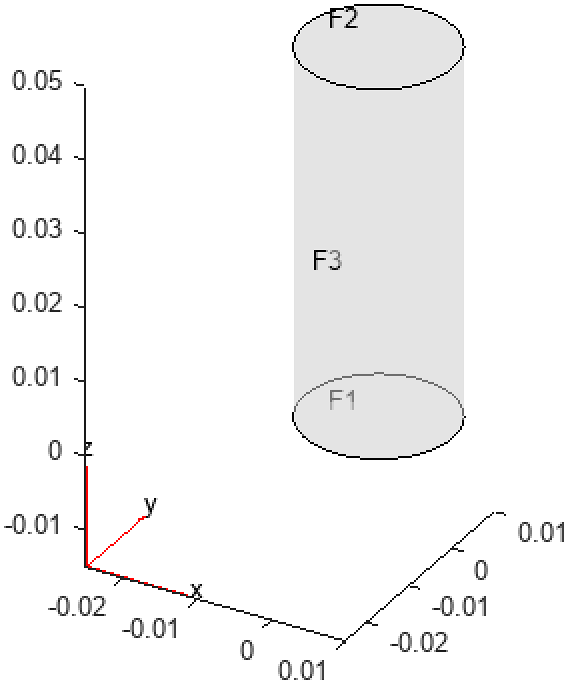
Assemble finite element matrices for the first and last time steps of a transient structural problem.

Create a transient structural model for solving a solid (3-D) problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder(0.01,0.05);
addVertex(gm,"Coordinates",[0,0,0.05]);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",201E9, ...
                                             "PoissonsRatio",0.3, ...
                                             "MassDensity",7800);
```

Specify that the bottom of the cylinder is a fixed boundary.

```
structuralBC(structuralmodel,"Face",1,"Constraint","fixed");
```

Specify the harmonic pressure on the top of the cylinder.

```
structuralBoundaryLoad(structuralmodel,"Face",2,...
                                       "Pressure",5E7, ...
                                       "Frequency",50);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0], ...
                             "Velocity",[0;0;0]);
```

Generate a linear mesh.

```
generateMesh(structuralmodel,"GeometricOrder","linear");
tlist = linspace(0,1,300);
```

Assemble the finite element matrices for the initial time step.

```
state.time = tlist(1);
FEM_domain = assembleFEMatrices(structuralmodel,state)
```

```
FEM_domain = struct with fields:
    K: [6609x6609 double]
    A: [6609x6609 double]
    F: [6609x1 double]
    Q: [6609x6609 double]
    G: [6609x1 double]
    H: [252x6609 double]
    R: [252x1 double]
    M: [6609x6609 double]
```

Pressure applied at the top of the cylinder is the only time-dependent quantity in the model. To model the dynamics of the system, assemble the boundary-load finite element matrix G for the initial, intermediate, and final time steps.

```
state.time = tlist(1);
FEM_boundary_init = assembleFEMatrices(structuralmodel,"G",state)
```

```
FEM_boundary_init = struct with fields:
    G: [6609x1 double]
```

```
state.time = tlist(floor(length(tlist)/2));
FEM_boundary_half = assembleFEMatrices(structuralmodel,"G",state)
```

```
FEM_boundary_half = struct with fields:
    G: [6609x1 double]
```

```
state.time = tlist(end);
FEM_boundary_final = assembleFEMatrices(structuralmodel,"G",state)
```

```
FEM_boundary_final = struct with fields:
    G: [6609x1 double]
```

Finite Element Matrices for Nonlinear Problem

Assemble finite element matrices for a heat transfer problem with temperature-dependent thermal conductivity.

Create a steady-state thermal model.

```
thermalmodelS = createpde("thermal","steadystate");
```

Create a 2-D geometry by drawing one rectangle the size of the block and a second rectangle the size of the slot.

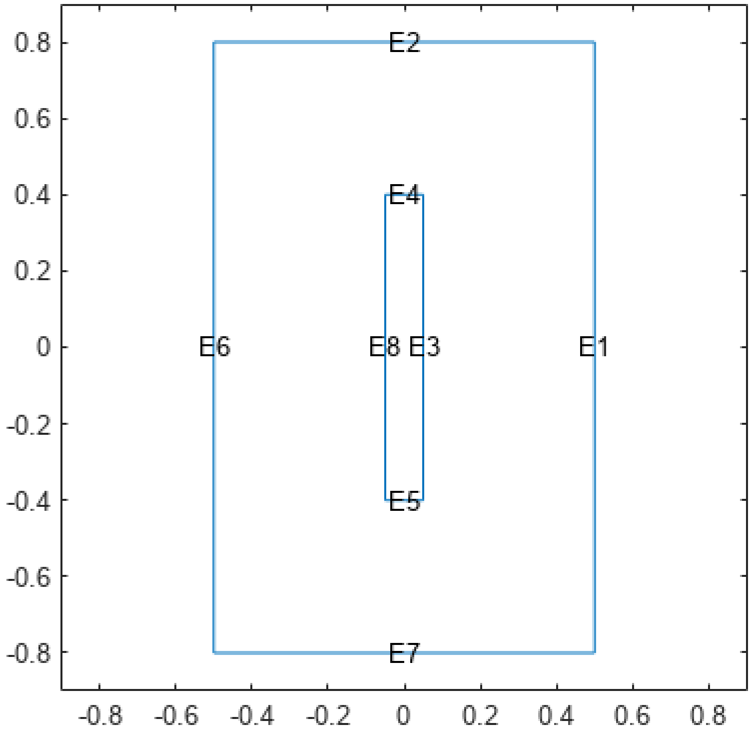
```
r1 = [3 4 -.5 .5 .5 -.5  -.8 -.8 .8 .8];
r2 = [3 4 -.05 .05 .05 -.05  -.4 -.4 .4 .4];
gdm = [r1; r2]';
```

Subtract the second rectangle from the first to create the block with a slot.

```
g = decsg(gdm,'R1-R2',['R1'; 'R2']');
```

Convert the `decsg` format into a geometry object. Include the geometry in the model and plot the geometry.

```
geometryFromEdges(thermalmodelS,g);
figure
pdegplot(thermalmodelS,"EdgeLabels","on");
axis([-.9 .9 -.9 .9]);
```



Set the temperature on the left edge to 100 degrees. Set the heat flux out of the block on the right edge to -10. The top and bottom edges and the edges inside the cavity are all insulated: there is no heat transfer across these edges.

```
thermalBC(thermalmodelS,"Edge",6,"Temperature",100);
thermalBC(thermalmodelS,"Edge",1,"HeatFlux",-10);
```

Specify the thermal conductivity of the material as a simple linear function of temperature `u`.

```
k = @(~,state) 0.7+0.003*state.u;
thermalProperties(thermalmodelS,"ThermalConductivity",k);
```

Generate a mesh.

```
generateMesh(thermalmodelS);
```

Calculate the steady-state solution.

```
Rnonlin = solve(thermalmodelS);
```

Because the thermal conductivity is nonlinear (depends on the temperature), compute the system matrices corresponding to the converged temperature. Assign the temperature distribution to the `u` field of the `state` structure array. Because the `u` field must contain a row vector, transpose the temperature distribution.

```
state.u = Rnonlin.Temperature.';
```

Assemble finite element matrices using the temperature distribution at the nodal points.

```
FEM = assembleFEMatrices(thermalmodelS,"nullspace",state)
```

```
FEM = struct with fields:
    Kc: [1277x1277 double]
    Fc: [1277x1 double]
     B: [1320x1277 double]
    ud: [1320x1 double]
     M: [1277x1277 double]
```

Compute the solution using the system matrices to verify that they yield the same temperature as `Rnonlin`.

```
u = FEM.B*(FEM.Kc\FEM.Fc) + FEM.ud;
```

Compare this result to the solution given by `solve`.

```
norm(u - Rnonlin.Temperature)
```

```
ans = 1.7735e-04
```

Input Arguments

`model` — Model object

`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectroMagneticModel` object

Model object, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectroMagneticModel` object.

`assembleFEMatrices` does not support assembling FE matrices for 3-D magnetostatic analysis models.

Example: `model = createpde(1)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","electrostatic")`

`bcmethod` — Method for including boundary conditions

`"none"` (default) | `"nullspace"` | `"stiff-spring"`

Method for including boundary conditions, specified as `"none"`, `"nullspace"`, or `"stiff-spring"`. For more information, see “Algorithms” on page 5-84.

Example: `FEM = assembleFEMatrices(model,"nullspace")`

Data Types: `char` | `string`

`matrices` — Matrices to assemble

matrix identifiers | `"boundary"` | `"domain"`

Matrices to assemble, specified as:

- Matrix identifiers, such as `"F"`, `"MKF"`, `"K"`, and so on — Assemble the corresponding matrices. Each uppercase letter represents one matrix: `K`, `A`, `F`, `Q`, `G`, `H`, `R`, `M`, and `T`. You can combine several letters into one character vector or string, such as `"MKF"`.
- `"boundary"` — Assemble all matrices related to geometry boundaries.
- `"domain"` — Assemble all domain-related matrices.

Example: `FEM = assembleFEMatrices(model,"KAF")`

Data Types: `char` | `string`

`state` — Time for time-dependent models and solution for nonlinear models

structure array

Time for time-dependent models and solution for nonlinear models, specified in a structure array. The array fields represent the following values:

- `state.time` contains a nonnegative number specifying the time value for time-dependent models.
- `state.u` contains a solution matrix of size *N*-by-*Np* that can be used to assemble matrices in a nonlinear problem setup, where coefficients are functions of `state.u`. Here, *N* is the number of equations in the system, and *Np* is the number of nodes in the mesh.

Example: `state.time = tlist(end); FEM = assembleFEMatrices(model,"boundary",state)`

Output Arguments

`FEM` — Finite element matrices

structural array

Finite element matrices, returned as a structural array. Use the `bcmethod` and `matrices` arguments to specify which finite element matrices you want to assemble.

The fields in the structural array depend on `bcmethod`:

- If the value is `"none"`, then the fields are `K`, `A`, `F`, `Q`, `G`, `H`, `R`, and `M`.
- If the value is `"nullspace"`, then the fields are `Kc`, `Fc`, `B`, `ud`, and `M`.
- If the value is `"stiff-spring"`, then the fields are `Ks`, `Fs`, and `M`.

The fields in the structural array also depend on `matrices`:

- If the value is `boundary`, then the fields are all matrices related to geometry boundaries.
- If the value is `domain`, then the fields are all domain-related matrices.
- If the value is a matrix identifier or identifiers, such as `"F"`, `"MKF"`, `"K"`, and so on, then the fields are the corresponding matrices.

For more information, see “Algorithms” on page 5-84.

Algorithms

Partial Differential Equation Toolbox solves 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 equations of the form

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

or

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

with the Dirichlet boundary conditions, $\mathbf{hu} = \mathbf{r}$, and Neumann boundary conditions, $\mathbf{n}\cdot(\mathbf{c}\otimes\nabla\mathbf{u}) + \mathbf{qu} = \mathbf{g}$.

`assembleFEMatrices` returns the following full finite element matrices and vectors that represent the corresponding PDE problem:

- `K` is the stiffness matrix, the integral of the discretized version of the `c` coefficient.
- `M` is the mass matrix, the integral of the discretized version of the `m` or `d` coefficients. `M` is nonzero for time-dependent and eigenvalue problems.
- `A` is the integral of the discretized version of the `a` coefficient.
- `F` is the integral of the discretized version of the `f` coefficient. For thermal, electromagnetic, and structural problems, `F` is a source or body load vector.
- `Q` is the integral of the discretized version of the `q` term in a Neumann boundary condition.

- `G` is the integral of the discretized version of the `g` term in a Neumann boundary condition. For structural problems, `G` is a boundary load vector.
- The `H` and `R` matrices come directly from the Dirichlet conditions and the mesh.

Imposing Dirichlet Boundary Conditions

The `"nullspace"` technique eliminates Dirichlet conditions from the problem using a linear algebra approach. It generates the combined finite-element matrices `Kc`, `Fc`, `B`, and vector `ud` corresponding to the reduced system `Kc*u = Fc`, where `Kc = B'*(K + A + Q)*B`, and `Fc = B'*(F + G)`. The `B` matrix spans the null space of the columns of `H` (the Dirichlet condition matrix representing `h*ud = r`). The `R` vector represents the Dirichlet conditions in `H*ud = R`. The `ud` vector has the size of the solution vector. Its elements are zeros everywhere except at Dirichlet degrees-of-freedom (DoFs) locations where they contain the prescribed values.

From the `"nullspace"` matrices, you can compute the solution `u` as

`u = B*(Kc\Fc) + ud`.

If you assembled a particular set of matrices, for example `G` and `M`, you can impose the boundary conditions on `G` and `M` as follows. First, compute the nullspace of columns of `H`.

```
[B,Or] = pdenullorth(H);
ud = Or*((H*Or\R)); % Vector with known value of the constraint DoF.
```

Then use the `B` matrix as follows. To eliminate Dirichlet degrees of freedom from the load vector `G`, use:

```
GwithBC = B'*G
```

To eliminate Dirichlet degrees of freedom from mass matrix, use:

```
M = B'*M*B
```

You can eliminate Dirichlet degrees of freedom from other vectors and matrices using the same technique.

The `"stiff-spring"` technique converts Dirichlet boundary conditions to Neumann boundary conditions using a stiff-spring approximation. It returns a matrix `Ks` and a vector `Fs` that together represent a different type of combined finite element matrices. The approximate solution is `u = Ks\Fs`. Compared to the `"nullspace"` technique, the `"stiff-spring"` technique generates matrices more quickly, but generally gives less accurate solutions.

Note Internally, the toolbox uses the `"nullspace"` approach to impose Dirichlet boundary conditions while computing the solution using `solvepde` and `solve`.

Degrees of Freedom (DoFs)

If the number of nodes in a model is `NumNodes`, and the number of equations is `N`, then the length of column vectors `u` and `ud` is `N*NumNodes`. The toolbox assigns the IDs to the degrees of freedom in `u` and `ud`:

- Entries from 1 to `NumNodes` correspond to the first equation.
- Entries from `NumNodes+1` to `2*NumNodes` correspond to the second equation.

- Entries from `2*NumNodes+1` to `3*NumNodes` correspond to the third equation.

The same approach applies to all other entries, up to `N*NumNodes`.

For example, in a 3-D structural model, the length of a solution vector `u` is `3*NumNodes`. The first `NumNodes` entries correspond to the `x`-displacement at each node, the next `NumNodes` entries correspond to the `y`-displacement, and the next `NumNodes` entries correspond to the `z`-displacement.

Thermal, Structural, and Electromagnetic Analysis

In thermal analysis, the `m` and `a` coefficients are zeros. The thermal conductivity maps to the `c` coefficient. The product of the mass density and the specific heat maps to the `d` coefficient. The internal heat source maps to the `f` coefficient. The temperature on a boundary corresponds to the Dirichlet boundary condition term `r` with `h = 1`. Various forms of boundary heat flux, such as the heat flux itself, emissivity, and convection coefficient, map to the Neumann boundary condition terms `q` and `g`.

In structural analysis, the `a` coefficient is zero. Young's modulus and Poisson's ratio map to the `c` coefficient. The mass density maps to the `m` coefficient. The body loads map to the `f` coefficient. Displacements, constraints, and components of displacement along the axes map to the Dirichlet boundary condition terms `h` and `r`. Boundary loads, such as pressure, surface tractions, and translational stiffnesses, correspond to the Neumann boundary condition terms `q` and `g`. When you specify the damping model by using the Rayleigh damping parameters `Alpha` and `Beta`, the discretized damping matrix `C` is computed by using the mass matrix `M` and the stiffness matrix `K` as `C = Alpha*M + Beta*K`. Hysteretic (structural) damping contributes to the stiffness matrix `K`, which becomes complex.

In electrostatic and magnetostatic analyses, the `m`, `a`, and `d` coefficients are zeros. The relative permittivity and relative permeability map to the `c` coefficient. The charge density and current density map to the `f` coefficient. The voltage and magnetic potential on a boundary correspond to the Dirichlet boundary condition term `r` with `h = 1`.

Note Assembling FE matrices does not work for harmonic analysis and 3-D magnetostatic analysis.

Version History

Introduced in R2016a

FE matrices for time-dependent and nonlinear models

The function now can assemble matrices using input time or solution for a time-dependent or nonlinear model, respectively. It also can assemble a subset of matrices, such as updated load only or stiffness only, to save computation time.

FE matrices for thermal and structural models

The function accepts thermal and structural models with time- and solution-independent coefficients.

See Also

`PDEModel` | `ThermalModel` | `StructuralModel` | `ElectromagneticModel` | `solvepde` | `solve`

Topics

“Finite Element Method Basics” on page 1-11

“Equations You Can Solve Using PDE Toolbox” on page 1-3

assempde

(Not recommended) Assemble finite element matrices and solve elliptic PDE

Note `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 5-101.

`[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 5-101.

`[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 5-101.

`[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 5-101.

`[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 5-101.

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.

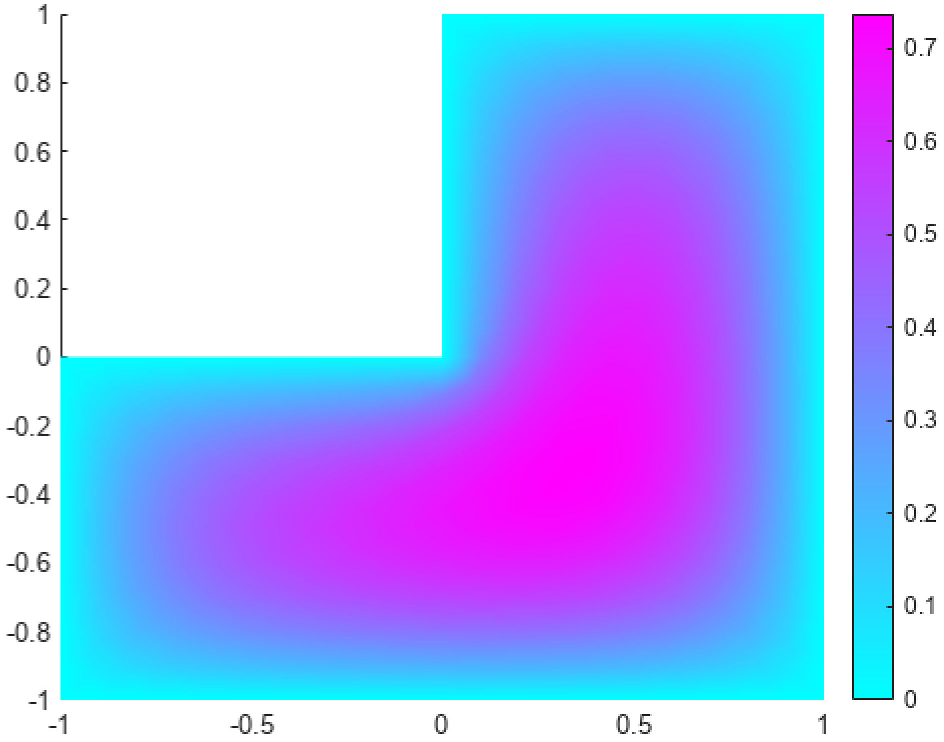
```
generateMesh(model,'GeometricOrder','linear');
```

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)
```

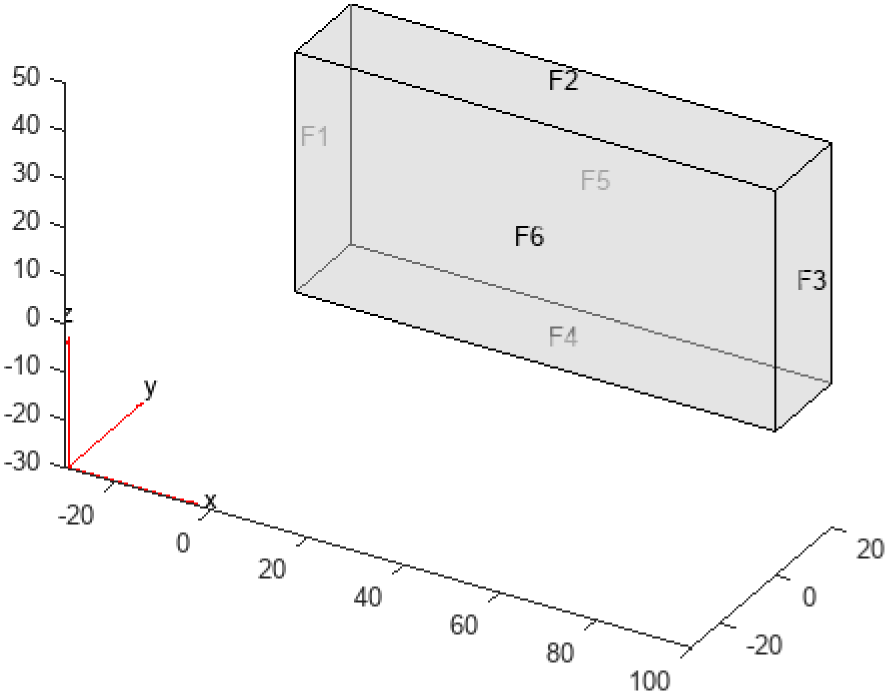


3-D Elliptic Problem

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`.

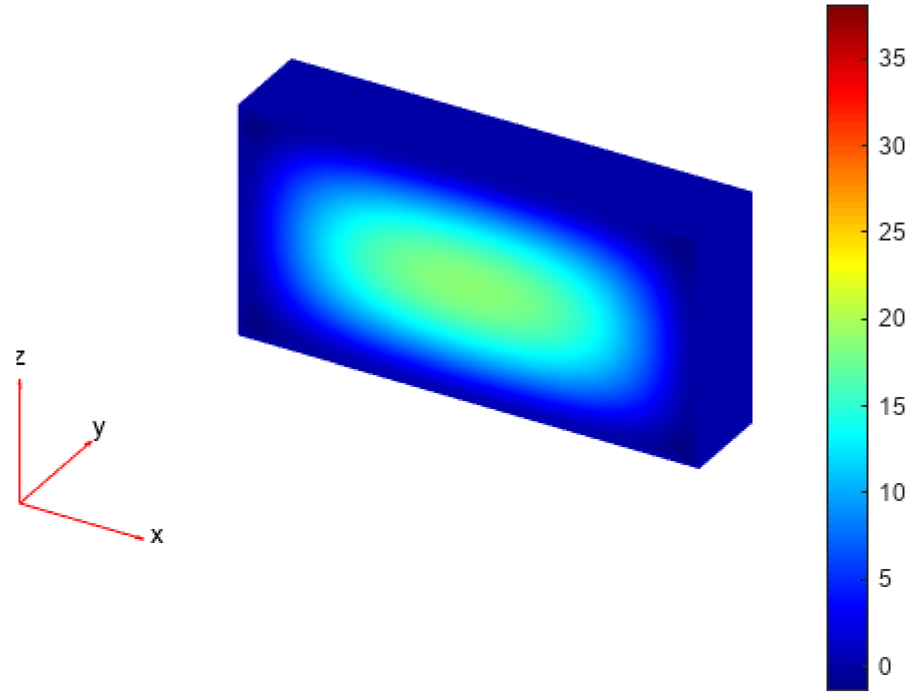
```
c = 1;
a = 0;
f = 0.1;
```

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.

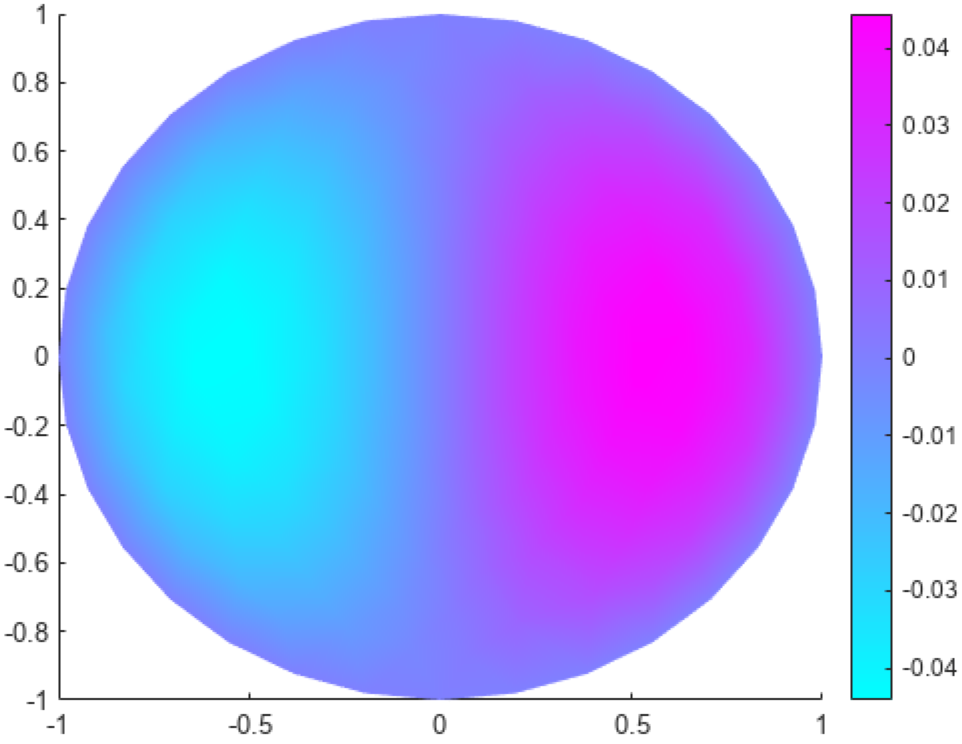
```
[p,e,t] = initmesh(g);
```

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 0.002006
```

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`

`c` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `c` represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

`a` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `a` represents the *a* coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

f — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `f` represents the f coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `string` | `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 Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 5-101.

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 5-101.

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 5-101.

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 5-101.

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 5-101.

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 5-101.

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 5-101.

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 `i`th column of `model.Mesh.Nodes` or the `i`th 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 "3-D Solution and Gradient Plots with MATLAB Functions" on page 3-344.

Kc — Stiffness matrix
sparse matrix

Stiffness matrix, returned as a sparse matrix. See "Elliptic Equations" on page 5-101.

`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 5-101.

```
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 5-101.

```
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 5-101.

```
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 5-101.

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 5-101.

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 5-101.

`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 5-101.

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 5-101.

`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 5-101.

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 5-101.

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 5-101.

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 5-101.

Typically, you use `R` in a subsequent call to a solver such as `assempde` or `hyperbolic`.

More About

Reduced Linear System

This form of the finite element matrices eliminates Dirichlet conditions from the problem using a linear algebra approach. The finite element matrices reduce to the solution `u = B*(Kc\Fc) + ud`, where `B` spans the null space of the columns of `H` (the Dirichlet condition matrix representing $hu = r$). `R` is the Dirichlet condition vector for `Hu = R`. `ud` is the vector of boundary condition solutions for the Dirichlet conditions. `u1` = `Kc\Fc` returns the solution on the non-Dirichlet points.

See “Systems of PDEs” on page 5-107 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 5-101. For details of the stiff-spring approximation, see “Systems of PDEs” on page 5-107.

Algorithms

Elliptic Equations

Partial Differential Equation Toolbox solves equations of the form

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

When the m and d coefficients are 0, this reduces to

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

which the documentation calls an elliptic equation, whether or not the equation is elliptic in the mathematical sense. The equation holds in Ω, where Ω is a bounded domain in two or three dimensions. c, a, f, and the unknown solution u are complex functions defined on Ω. c can also be a 2-by-2 matrix function on Ω. The boundary conditions specify a combination of u and its normal derivative on the boundary:

- *Dirichlet:* $hu = r$ on the boundary $\partial\Omega$.
- *Generalized Neumann:* $\vec{n} \cdot (c\nabla u) + qu = g$ on $\partial\Omega$.
- *Mixed:* Only applicable to *systems*. A combination of Dirichlet and generalized Neumann.

$\vec{n}$ is the outward unit normal. g, q, h, and r are functions defined on $\partial\Omega$.

Our nomenclature deviates slightly from the tradition for potential theory, where a Neumann condition usually refers to the case $q = 0$ and our Neumann would be called a mixed condition. In some contexts, the generalized Neumann boundary conditions is also referred to as the *Robin boundary conditions*. In variational calculus, Dirichlet conditions are also called essential boundary conditions and restrict the trial space. Neumann conditions are also called natural conditions and arise as necessary conditions for a solution. The variational form of the Partial Differential Equation Toolbox equation with Neumann conditions is given below.

The approximate solution to the elliptic PDE is found in three steps:

1. Describe the geometry of the domain Ω and the boundary conditions. For 2-D geometry, create geometry using the PDE Modeler app or through MATLAB files. For 3-D geometry, import the geometry in STL file format.
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 5-107.

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

$$\int_{\Omega}(-(\nabla \cdot c\nabla u)v + auv)dx = \int_{\Omega} fv\, dx$$

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

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

The boundary integral can be replaced by the boundary condition:

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

Replace the original problem with *Find u such that*

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

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.,

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

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:

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

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

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

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

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

and rewrite the system in the form

$$(K + M + Q)U = F + G. \tag{5-2}$$

K, M, and Q are N_p-by-N_p matrices, and F and G are N_p-vectors. K, M, and F are produced by `assema`, while Q, G are produced by `assemb`. When it is not necessary to distinguish K, M, and Q or F and G, we collapse the notations to $KU = F$, which form the output of `assempde`.

When the problem is *self-adjoint* and *elliptic* in the usual mathematical sense, the matrix $K + M + Q$ becomes symmetric and positive definite. Many common problems have these characteristics, most notably those that can also be formulated as minimization problems. For the case of a scalar equation, K, M, and Q are obviously symmetric. If $c(x) \geq \delta > 0$, $a(x) \geq 0$ and $q(x) \geq 0$ with $q(x) > 0$ on some part of $\partial\Omega$, then, if $U \neq 0$.

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

$U^T(K + M + Q)U$ is the *energy norm*. There are many choices of the test-function spaces. The software uses continuous functions that are linear on each element of a 2-D mesh, and are linear or quadratic on elements of a 3-D mesh. Piecewise linearity guarantees that the integrals defining the stiffness matrix K exist. Projection onto V_{N_p} is nothing more than linear interpolation, and the evaluation of the solution inside an element is done just in terms of the nodal values. If the mesh is uniformly refined, V_{N_p} approximates the set of smooth functions on Ω.

A suitable basis for V_{N_p} in 2-D is the set of “tent” or “hat” functions ϕ_i. These are linear on each element and take the value 0 at all nodes x_j except for x_i. For the definition of basis functions for 3-D geometry, see “Finite Element Basis for 3-D” on page 5-109. Requesting $\phi_i(x_i) = 1$ yields the very pleasant property

$$u(x_i) = \sum_{j=1}^{N_p} U_j\phi_j(x_i) = U_i$$

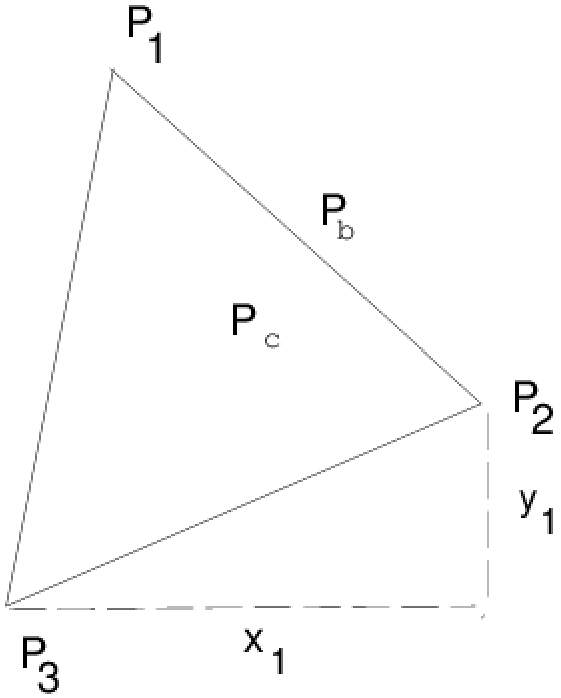
That is, by solving the FEM system we obtain the nodal values of the approximate solution. The basis function ϕ_i vanishes on all the elements that do not contain the node x_i. The immediate consequence is that the integrals appearing in $K_{i,j}$, $M_{i,j}$, $Q_{i,j}$, F_i and G_i only need to be computed on the elements that contain the node x_i. Secondly, it means that $K_{i,j}$ and$M_{i,j}$ are zero unless x_i and x_j are vertices of the same element and thus K and M are very sparse matrices. Their sparse structure depends on the ordering of the indices of the mesh points.

The integrals in the FEM matrices are computed by adding the contributions from each element to the corresponding entries (i.e., only if the corresponding mesh point is a vertex of the element). This process is commonly called *assembling*, hence the name of the function `assempde`.

The assembling routines scan the elements of the mesh. For each element they compute the so-called local matrices and add their components to the correct positions in the sparse matrices or vectors.

The discussion now specializes to triangular meshes in 2-D. The local 3-by-3 matrices contain the integrals evaluated only on the current triangle. The coefficients are assumed constant on the triangle and they are evaluated only in the triangle barycenter. The integrals are computed using the midpoint rule. This approximation is optimal since it has the same order of accuracy as the piecewise linear interpolation.

Consider a triangle given by the nodes P_1, P_2, and P_3 as in the following figure.



The Local Triangle P1P2P3

Note The local 3-by-3 matrices contain the integrals evaluated only on the current triangle. The coefficients are assumed constant on the triangle and they are evaluated only in the triangle barycenter.

The simplest computations are for the local mass matrix *m*:

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

where P_c is the center of mass of $\Delta\, P_1P_2P_3$, i.e.,

$$P_c = \frac{P_1 + P_2 + P_3}{3}$$

The contribution to the right side F is just

$$f_i = f(P_c)\frac{\text{area}(\Delta P_1 P_2 P_3)}{3}$$

For the local stiffness matrix we have to evaluate the gradients of the basis functions that do not vanish on $P_1P_2P_3$. Since the basis functions are linear on the triangle $P_1P_2P_3$, the gradients are constants. Denote the basis functions ϕ_1, ϕ_2, and ϕ_3 such that $\phi(P_i) = 1$. If $P_2 - P_3 = [x_1,y_1]^T$ then we have that

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

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

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

If two vertices of the triangle lie on the boundary $\partial\Omega$, they contribute to the line integrals associated to the boundary conditions. If the two boundary points are P_1 and P_2, then we have

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

and

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

where P_b is the midpoint of P_1P_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.

To summarize, `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 5-101 and “Stiff-Spring Approximation” on page 5-101.

Systems of PDEs

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

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

the parabolic system

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

the hyperbolic system

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

and the eigenvalue system

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

where **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

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

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,

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

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

$$
\begin{aligned}
&\sum_{j=1}^{N}\left(\cos(\alpha)c_{i,j,1,1}\frac{\partial}{\partial x}+\cos(\alpha)c_{i,j,1,2}\frac{\partial}{\partial y}+\cos(\alpha)c_{i,j,1,3}\frac{\partial}{\partial z}\right)u_j\\
&+\sum_{j=1}^{N}\left(\cos(\beta)c_{i,j,2,1}\frac{\partial}{\partial x}+\cos(\beta)c_{i,j,2,2}\frac{\partial}{\partial y}+\cos(\beta)c_{i,j,2,3}\frac{\partial}{\partial z}\right)u_j\\
&+\sum_{j=1}^{N}\left(\cos(\gamma)c_{i,j,3,1}\frac{\partial}{\partial x}+\cos(\gamma)c_{i,j,3,2}\frac{\partial}{\partial y}+\cos(\gamma)c_{i,j,3,3}\frac{\partial}{\partial z}\right)u_j
\end{aligned}
$$

where the outward normal to the boundary is

$$\mathbf{n} = (\cos(\alpha), \cos(\beta), \cos(\gamma))$$

There are M Dirichlet conditions and the **h**-matrix is M-by-N, $M \geq 0$. The generalized Neumann condition contains a source $\mathbf{h}'\mu$, where the Lagrange multipliers μ 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 5-101 for the scalar case, which is equivalent to a diagonal **h**-matrix. For the general case, Dirichlet conditions

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

are approximated by adding a term

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

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

When this number is increased, **hu** = **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 **h**, and is free of the ill-conditioning, but is more involved computationally. Assume that there are N_p nodes in the mesh. Then the number of unknowns is $N_pN = N_u$. When Dirichlet boundary conditions fix some of the unknowns, the linear system can be correspondingly reduced. This is easily done by removing rows and columns when u values are given, but here we must treat the case when some linear combinations of the components of u are given, **hu** = **r**. These are collected into $HU = R$ where H is an M-by-N_u matrix and R is an M-vector.

With the reaction force term the system becomes

$$KU + H'\mu = F$$

$$HU = R.$$

The constraints can be solved for M of the U-variables, the remaining called V, an $N_u - M$ vector. The null space of H is spanned by the columns of B, and $U = BV + u_d$ makes U satisfy the Dirichlet conditions. A permutation to block-diagonal form exploits the sparsity of H to speed up the following computation to find B in a numerically stable way. μ can be eliminated by pre-multiplying by B' since, by the construction, $HB = 0$ or $B'H' = 0$. The reduced system becomes

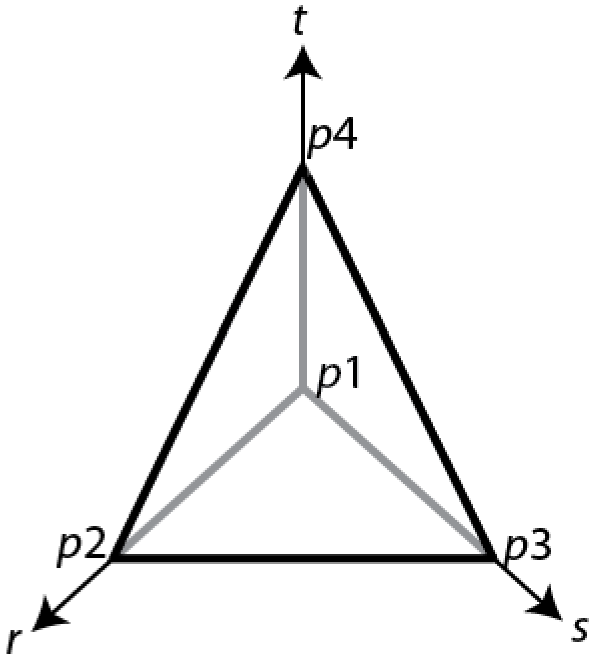
$$B'KBV = B'F - B'Ku_d$$

which is symmetric and positive definite if K is.

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 5-101. 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).

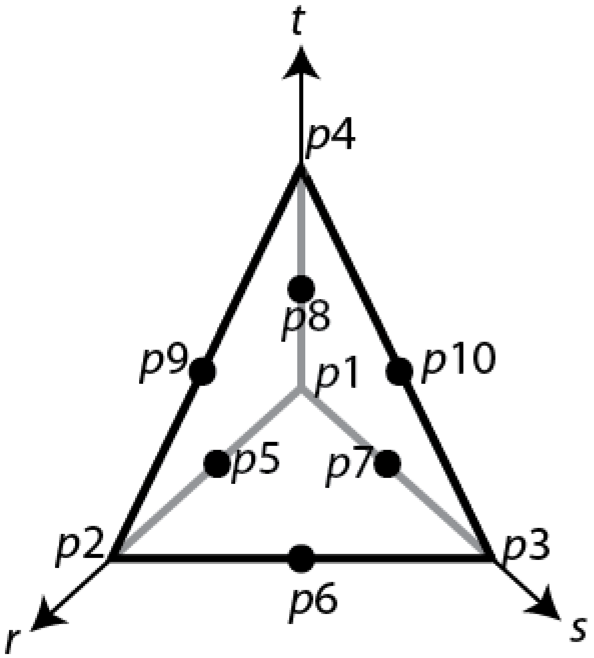


In local coordinates, the point $p1$ is at (0,0,0), $p2$ is at (1,0,0), $p3$ is at (0,1,0), and $p4$ is at (0,0,1).

For a linear tetrahedron, the basis functions are

$$\begin{aligned}\phi_1 &= 1 - r - s - t\\ \phi_2 &= r\\ \phi_3 &= s\\ \phi_4 &= t\end{aligned}$$

For a quadratic tetrahedron, there are additional nodes at the edge midpoints.



The corresponding basis functions are

$$\phi_1 = 2(1-r-s-t)^2 - (1-r-s-t)$$
$$\phi_2 = 2r^2 - r$$
$$\phi_3 = 2s^2 - s$$
$$\phi_4 = 2t^2 - t$$
$$\phi_5 = 4r(1-r-s-t)$$
$$\phi_6 = 4rs$$
$$\phi_7 = 4s(1-r-s-t)$$
$$\phi_8 = 4t(1-r-s-t)$$
$$\phi_9 = 4rt$$
$$\phi_{10} = 4st$$

As in the 2-D case, a 3-D basis function ϕ_i takes the value 0 at all nodes j, except for node i, where it takes the value 1.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`assempde` is not recommended. Use `solvepde` instead. There are no plans to remove `assempde`.

See Also

`assembleFEMatrices` | `solvepde`

cellEdges

Find edges belonging to boundaries of specified cells

Syntax

```
EdgeID = cellEdges(g,RegionID)
EdgeID = cellEdges(g,RegionID,FilterType)
```

Description

`EdgeID = cellEdges(g,RegionID)` finds edges belonging to the boundaries of the cells with ID numbers listed in `RegionID`.

`EdgeID = cellEdges(g,RegionID,FilterType)` returns internal, external, or all edges belonging to the boundaries of the cells with ID numbers listed in `RegionID`.

Examples

Edges Belonging to Specified Cells

Find edges belonging to the boundaries of the two middle cylinders in a geometry consisting of four stacked cylinders.

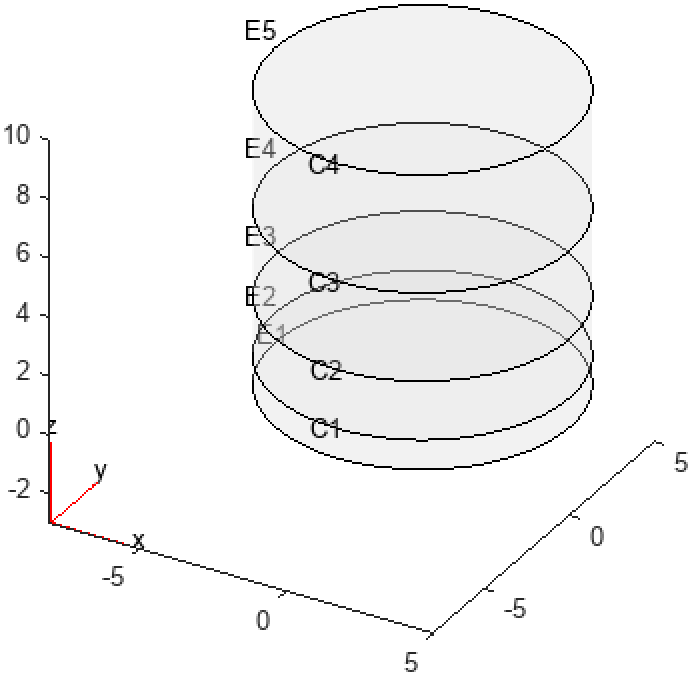
Create a geometry that consists of four stacked cylinders.

```
gm = multicylinder(5,[1 2 3 4],"ZOffset",[0 1 3 6])
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 4
       NumFaces: 9
       NumEdges: 5
    NumVertices: 5
       Vertices: [5x3 double]
```

Plot the geometry with the cell and edge labels.

```
pdegplot(gm,"CellLabels","on","EdgeLabels","on","FaceAlpha",0.2)
```



Find edges belonging to the boundaries of cells 2 and 3.

```
edgeIDs = cellEdges(gm,[2 3])
```

```
edgeIDs = 1×3

     2     3     4
```

Cell Edges Belonging to Internal and External Faces

Find edges belonging to the boundaries of the outer cuboid in a geometry consisting of two nested cuboids.

Create a geometry that consists of two nested cuboids of the same height.

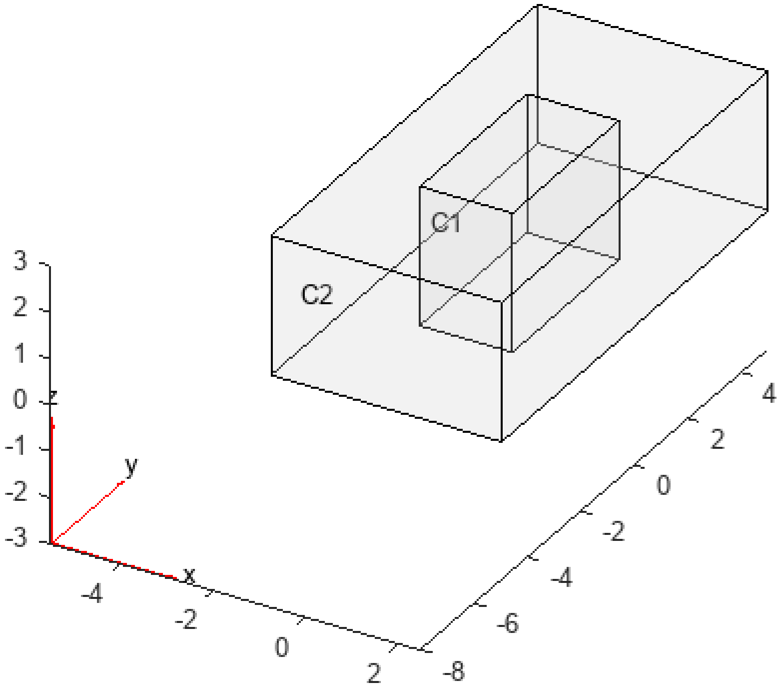
```
gm = multicuboid([2 5],[4 10],3)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 2
       NumFaces: 12
       NumEdges: 24
    NumVertices: 16
```

```
        Vertices: [16x3 double]
```

Plot the geometry with the cell labels.

```
pdegplot(gm,"CellLabels","on","FaceAlpha",0.2)
```



Find all edges belonging to the boundaries of the outer cell. Show the first 10 edges.

```
edgeIDs = cellEdges(gm,2);
edgeIDs(1:10)
```

```
ans = 1×10

     1     2     3     4     5     6     7     8     9    10
```

From all edges belonging to the boundaries of the outer cell, return the edges belonging to only the internal faces. Internal faces are faces shared between multiple cells.

```
edgeIDs_int = cellEdges(gm,2,"internal")
```

```
edgeIDs_int = 1×4

     9    10    11    12
```

From all edges belonging to the boundaries of the outer cell, return the edges belonging to the external faces. Show the first 10 edges.

```
edgeIDs_ext = cellEdges(gm,2,"external");
edgeIDs_ext(1:10)

ans = 1×10

     1     2     3     4     5     6     7     8    13    14
```

Input Arguments

g — 3-D geometry

`DiscreteGeometry` object

3-D geometry, specified as a `DiscreteGeometry` object.

`RegionID` — Cell ID

positive number | vector of positive numbers

Cell ID, specified as a positive number or a vector of positive numbers. Each number represents a cell ID.

`FilterType` — Type of edges to return

`"all"` (default) | `"internal"` | `"external"`

Type of edges to return, specified as `"internal"`, `"external"`, or `"all"`. Depending on this argument, `cellEdges` returns these types of faces:

- `"internal"` — Edges belonging to only internal faces. Internal faces are faces shared between multiple cells.
- `"external"` — Edges belonging to only external faces. External faces are faces not shared between multiple cells.
- `"all"` — All edges belonging to the specified cells.

Output Arguments

`EdgeID` — IDs of edges belonging to boundaries of specified cells

positive number | vector of positive numbers

IDs of edges belonging to boundaries of specified cells, returned as a positive number or a vector of positive numbers.

Version History

Introduced in R2021a

See Also

`cellFaces` | `faceEdges` | `facesAttachedToEdges` | `nearestEdge` | `nearestFace` | DiscreteGeometry Properties | AnalyticGeometry Properties

cellFaces

Find faces belonging to specified cells

Syntax

```
FaceID = cellFaces(g,RegionID)
FaceID = cellFaces(g,RegionID,FilterType)
```

Description

`FaceID = cellFaces(g,RegionID)` finds faces belonging to the cells with ID numbers listed in `RegionID`.

`FaceID = cellFaces(g,RegionID,FilterType)` returns internal, external, or all faces belonging to the cells with ID numbers listed in `RegionID`.

Examples

Faces Belonging to Specified Cells

Find faces belonging to two cuboids in a geometry consisting of four stacked cuboids.

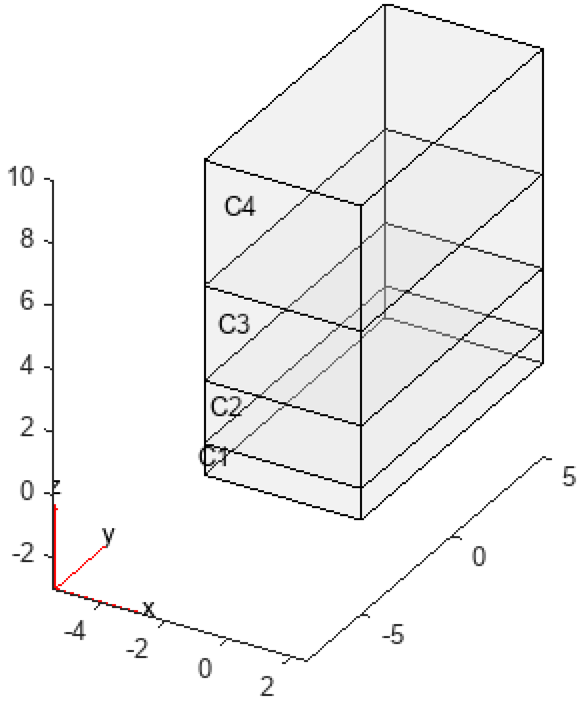
Create a geometry that consists of four stacked cuboids.

```
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
       Vertices: [20x3 double]
```

Plot the geometry with the cell labels.

```
pdegplot(gm,"CellLabels","on","FaceAlpha",0.2)
```



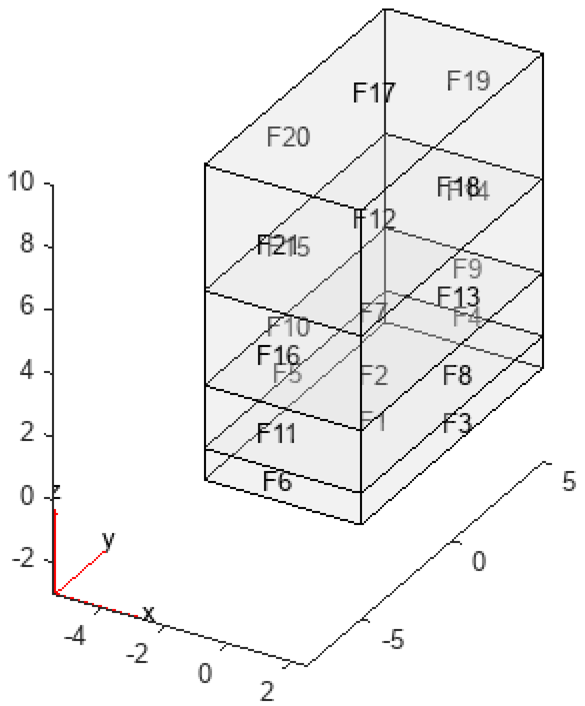
Find faces belonging to cells 1 and 3.

```
faceIDs = cellFaces(gm,[1 3])
```

```
faceIDs = 1×12

     1     2     3     4     5     6     7    12    13    14    15    16
```

Plot the geometry with the face labels.

```
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.2)
```



Internal and External Faces Belonging to Specified Cells

Find faces belonging to the outer cuboid in a geometry consisting of two nested cuboids.

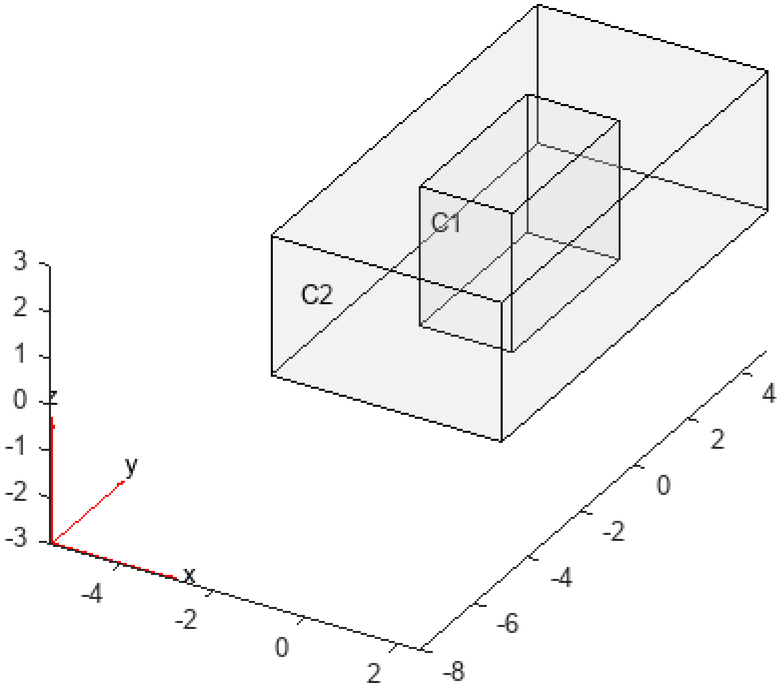
Create a geometry that consists of two nested cuboids of the same height.

```
gm = multicuboid([2 5],[4 10],3)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 2
       NumFaces: 12
       NumEdges: 24
    NumVertices: 16
       Vertices: [16x3 double]
```

Plot the geometry with the cell labels.

```
pdegplot(gm,"CellLabels","on","FaceAlpha",0.2)
```



Find all faces belonging to the outer cell.

```
faceIDs = cellFaces(gm,2)
```

```
faceIDs = 1×10

     3     4     5     6     7     8     9    10    11    12
```

Find only the internal faces belonging to the outer cell. Internal faces are faces shared between multiple cells.

```
faceIDs_int = cellFaces(gm,2,"internal")
```

```
faceIDs_int = 1×4

     3     4     5     6
```

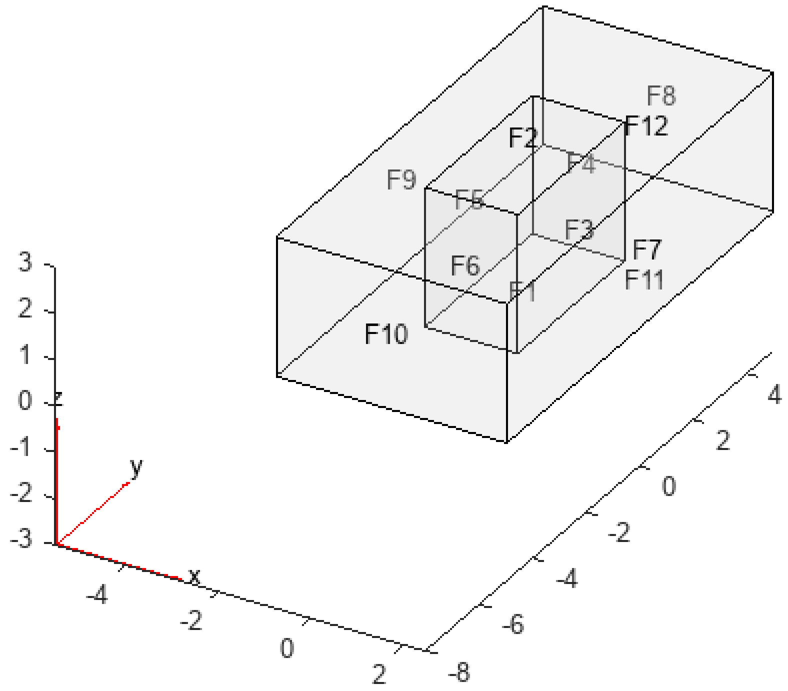
Find only the external faces belonging to the outer cell.

```
faceIDs_ext = cellFaces(gm,2,"external")
```

```
faceIDs_ext = 1×6

     7     8     9    10    11    12
```

Plot the geometry with the face labels.

```
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.2)
```



Input Arguments

g — 3-D geometry

`DiscreteGeometry` object

3-D geometry, specified as a `DiscreteGeometry` object.

`RegionID` — Cell ID

positive number | vector of positive numbers

Cell ID, specified as a positive number or a vector of positive numbers. Each number represents a cell ID.

`FilterType` — Type of faces to return

`"all"` (default) | `"internal"` | `"external"`

Type of faces to return, specified as `"internal"`, `"external"`, or `"all"`. Depending on this argument, `cellFaces` returns these types of faces:

- `"internal"` — Internal faces, that is, faces shared between multiple cells.
- `"external"` — External faces, that is, faces not shared between multiple cells.
- `"all"` — All faces belonging to the specified cells.

Output Arguments

`FaceID` — IDs of faces belonging to specified cells
positive number | vector of positive numbers

IDs of faces belonging to the specified cells, returned as a positive number or a vector of positive numbers.

Version History

Introduced in R2021a

See Also

`cellEdges` | `faceEdges` | `facesAttachedToEdges` | `nearestEdge` | `nearestFace` | DiscreteGeometry Properties | AnalyticGeometry Properties

BodyLoadAssignment Properties

Body load assignments

Description

A `BodyLoadAssignment` object contains a description of the body loads for a structural analysis model. A `StructuralModel` container has a vector of `BodyLoadAssignment` objects in its `BodyLoads.BodyLoadAssignments` property.

To create body load assignments for your structural analysis model, use the `structuralBodyLoad` function.

Properties

Properties of BodyLoadAssignment

`RegionType` — Region type
`'Face'` | `'Cell'`

Region type, specified as `'Face'` for a 2-D region or `'Cell'` for a 3-D region.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. To determine which ID corresponds to which portion of the geometry, use the `pdegplot` function, setting `'FaceLabels'` to `'on'`.

Data Types: `double`

`GravitationalAcceleration` — Acceleration due to gravity
numeric vector

Acceleration due to gravity, specified as a numeric vector. This property must be specified in units consistent with those of the geometry and material properties.

Example: `structuralBodyLoad(structuralmodel,'GravitationalAcceleration',[0,0,-9.8])`

Data Types: `double`

`AngularVelocity` — Angular velocity for axisymmetric model
positive number

Angular velocity for an axisymmetric model, specified as a positive number. This property must be specified in units consistent with those of the geometry and material properties.

Example: `structuralBodyLoad(structuralmodel,'AngularVelocity',2.3)`

Data Types: `double`

`Temperature` — Thermal load
real number | `StaticThermalResults` object | `TransientThermalResults` object

Thermal load, specified as a real number, a `StaticThermalResults` object, or a `TransientThermalResults` object. This property must be specified in units consistent with those of the geometry and material properties.

Example: `structuralBodyLoad(structuralmodel,'Temperature',300)`

Data Types: `double`

`TimeStep` — Time index for thermal load
positive integer

Time index for thermal load, specified as a positive integer.

Example:
`structuralBodyLoad(structuralmodel,'Temperature',Tresults,'TimeStep',21)`

Data Types: `double`

`Label` — Label for use with `linearizeInput`
character vector | string

Label for use with `linearizeInput`, specified as a character vector or a string.

Data Types: `char` | `string`

Version History

Introduced in R2017b

See Also

`findBodyLoad` | `structuralBodyLoad`

BoundaryCondition Properties

Boundary condition for PDE model

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

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` | `string`

RegionID — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by 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 2-132.

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 2-132.

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 2-132.

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 2-132.

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 2-132.

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”. For details of this evaluation, see “Nonconstant Boundary Conditions” on page 2-132.

Example: `applyBoundaryCondition(model,'dirichlet','Face',[2,4,11],'u',@ucalculator,'Vectorized','on')`

Data Types: `char`

Version History

Introduced in R2015a

See Also

`applyBoundaryCondition` | `findBoundaryConditions` | `PDEModel`

Topics

“Specify Boundary Conditions” on page 2-129
“View, Edit, and Delete Boundary Conditions” on page 2-150
“Solve Problems Using PDEModel Objects” on page 2-2

CoefficientAssignment Properties

Coefficient assignments

Description

A `CoefficientAssignment` object contains a description of the PDE coefficients. A `PDEModel` container has a vector of `CoefficientAssignment` objects in its `EquationCoefficients.CoefficientAssignments` property.

Coefficients are the *m*, *d*, *c*, *a*, and *f* variables in the PDE

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

or the eigenvalue problem

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

or

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

Create coefficients for your model using the `specifyCoefficients` function.

Properties

Properties

`RegionType` — Region type
`'face'` | `'cell'`

Region type, specified as `'face'` for a 2-D region, or `'cell'` for a 3-D region.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified 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, specified 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 2-107.

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 of the `d` coefficient specification, see “m, d, or a Coefficient for specifyCoefficients” on page 2-107.

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 of the `c` coefficient specification, see “c Coefficient for specifyCoefficients” on page 2-92.

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 of the `a` coefficient specification, see “m, d, or a Coefficient for specifyCoefficients” on page 2-107.

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 of the `f` coefficient specification, see “f Coefficient for specifyCoefficients” on page 2-90.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

Version History

Introduced in R2016a

See Also

`findCoefficients` | `specifyCoefficients`

Topics

“Solve Problems Using PDEModel Objects” on page 2-2

createpde

Create model

Syntax

```
structuralmodel = createpde("structural",StructuralAnalysisType)
thermalmodel = createpde("thermal",ThermalAnalysisType)
emagmodel = createpde("electromagnetic",ElectromagneticAnalysisType)
model = createpde(N)
model = createpde
```

Description

`structuralmodel = createpde("structural",StructuralAnalysisType)` returns a structural analysis model for the specified analysis type. This model lets you solve small-strain linear elasticity problems.

`thermalmodel = createpde("thermal",ThermalAnalysisType)` returns a thermal analysis model for the specified analysis type.

`emagmodel = createpde("electromagnetic",ElectromagneticAnalysisType)` returns an electromagnetic analysis model for the specified analysis type.

`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, and boundary conditions.

`model = createpde` returns a PDE model object for one equation (a scalar PDE). This syntax is equivalent to `model = createpde(1)` and `model = createpde()`.

Examples

Create Structural Model

Create a static structural model for solving a solid (3-D) problem.

```
staticStructural = createpde("structural","static-solid")
```

```
staticStructural = 
  StructuralModel with properties:

              AnalysisType: "static-solid"
                  Geometry: []
        MaterialProperties: []
                 BodyLoads: []
        BoundaryConditions: []
      ReferenceTemperature: []
    SuperelementInterfaces: []
                      Mesh: []
```

```
             SolverOptions: [1x1 pde.PDESolverOptions]
```

Create a transient structural model for solving a plane-stress (2-D) problem.

```
transientStructural = createpde("structural","transient-planestress")
```

```
transientStructural = 
  StructuralModel with properties:

              AnalysisType: "transient-planestress"
                  Geometry: []
        MaterialProperties: []
                 BodyLoads: []
        BoundaryConditions: []
             DampingModels: []
         InitialConditions: []
    SuperelementInterfaces: []
                      Mesh: []
             SolverOptions: [1x1 pde.PDESolverOptions]
```

Create a structural model for modal analysis of a plane-strain (2-D) problem.

```
modalStructural = createpde("structural","modal-planestrain")
```

```
modalStructural = 
  StructuralModel with properties:

              AnalysisType: "modal-planestrain"
                  Geometry: []
        MaterialProperties: []
        BoundaryConditions: []
    SuperelementInterfaces: []
                      Mesh: []
             SolverOptions: [1x1 pde.PDESolverOptions]
```

Create a structural model for frequency response analysis of an axisymmetric problem. An axisymmetric model simplifies a 3-D problem to a 2-D problem using symmetry around the axis of rotation.

```
frStructural = createpde("structural","frequency-axisymmetric")
```

```
frStructural = 
  StructuralModel with properties:

              AnalysisType: "frequency-axisymmetric"
                  Geometry: []
        MaterialProperties: []
                 BodyLoads: []
        BoundaryConditions: []
             DampingModels: []
    SuperelementInterfaces: []
                      Mesh: []
             SolverOptions: [1x1 pde.PDESolverOptions]
```

Create Thermal Model

Create a model for steady-state thermal analysis.

```
thermalmodel = createpde("thermal","steadystate")
```

```
thermalmodel = 
  ThermalModel with properties:

               AnalysisType: "steadystate"
                   Geometry: []
         MaterialProperties: []
                HeatSources: []
    StefanBoltzmannConstant: []
         BoundaryConditions: []
          InitialConditions: []
                       Mesh: []
              SolverOptions: [1x1 pde.PDESolverOptions]
```

Create a model for transient thermal analysis.

```
thermalmodel = createpde("thermal","transient")
```

```
thermalmodel = 
  ThermalModel with properties:

               AnalysisType: "transient"
                   Geometry: []
         MaterialProperties: []
                HeatSources: []
    StefanBoltzmannConstant: []
         BoundaryConditions: []
          InitialConditions: []
                       Mesh: []
              SolverOptions: [1x1 pde.PDESolverOptions]
```

Create a model for modal thermal analysis.

```
thermalmodel = createpde("thermal","modal")
```

```
thermalmodel = 
  ThermalModel with properties:

               AnalysisType: "modal"
                   Geometry: []
         MaterialProperties: []
                HeatSources: []
    StefanBoltzmannConstant: []
         BoundaryConditions: []
          InitialConditions: []
                       Mesh: []
              SolverOptions: [1x1 pde.PDESolverOptions]
```

Create a transient thermal model for solving an axisymmetric problem. An axisymmetric model simplifies a 3-D problem to a 2-D problem using symmetry around the axis of rotation.

```
thermalmodel = createpde("thermal","transient-axisymmetric")
```

```
thermalmodel = 
  ThermalModel with properties:

               AnalysisType: "transient-axisymmetric"
                   Geometry: []
         MaterialProperties: []
                HeatSources: []
    StefanBoltzmannConstant: []
         BoundaryConditions: []
          InitialConditions: []
                       Mesh: []
              SolverOptions: [1x1 pde.PDESolverOptions]
```

Create Electromagnetic Model

Create a model for electrostatic analysis.

```
emagE = createpde("electromagnetic","electrostatic")
```

```
emagE = 
  ElectromagneticModel with properties:

          AnalysisType: "electrostatic"
              Geometry: []
    MaterialProperties: []
               Sources: []
    BoundaryConditions: []
    VacuumPermittivity: []
                  Mesh: []
```

Create an axisymmetric model for magnetostatic analysis. An axisymmetric model simplifies a 3-D problem to a 2-D problem using symmetry around the axis of rotation.

```
emagMA = createpde("electromagnetic","magnetostatic-axisymmetric")
```

```
emagMA = 
  ElectromagneticModel with properties:

          AnalysisType: "magnetostatic-axisymmetric"
              Geometry: []
    MaterialProperties: []
               Sources: []
    BoundaryConditions: []
    VacuumPermeability: []
                  Mesh: []
```

Create a model for harmonic analysis.

```
emagH = createpde("electromagnetic","harmonic")
```

```
emagH = 
  ElectromagneticModel with properties:

          AnalysisType: "harmonic"
              Geometry: []
    MaterialProperties: []
               Sources: []
    BoundaryConditions: []
    VacuumPermittivity: []
    VacuumPermeability: []
                  Mesh: []
             FieldType: "electric"
```

Create a model for DC conduction analysis.

```
emagDC = createpde("electromagnetic","conduction")
```

```
emagDC = 
  ElectromagneticModel with properties:

          AnalysisType: "conduction"
              Geometry: []
    MaterialProperties: []
    BoundaryConditions: []
                  Mesh: []
```

Create General PDE Model

Create a model for a general linear or nonlinear single (scalar) PDE.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

Create a PDE model for a system of three equations.

```
model = createpde(3)
```

```
model = 
  PDEModel with properties:
```

```
          PDESystemSize: 3
        IsTimeDependent: 0
               Geometry: []
   EquationCoefficients: []
     BoundaryConditions: []
      InitialConditions: []
                   Mesh: []
          SolverOptions: [1x1 pde.PDESolverOptions]
```

Input Arguments

`StructuralAnalysisType` — Type of structural analysis

`"static-solid"` | `"static-planestress"` | `"static-planestrain"` | `"static-axisymmetric"` | `"transient-solid"` | `"transient-planestress"` | `"transient-planestrain"` | `"transient-axisymmetric"` | `"modal-solid"` | `"modal-planestress"` | `"modal-planestrain"` | `"modal-axisymmetric"` | `"frequency-solid"` | `"frequency-planestress"` | `"frequency-planestrain"` | `"frequency-axisymmetric"`

Type of structural analysis, specified as one of the following values.

For static analysis, use these values:

- `"static-solid"` — Creates a structural model for static analysis of a solid (3-D) problem.
- `"static-planestress"` — Creates a structural model for static analysis of a plane-stress problem.
- `"static-planestrain"` — Creates a structural model for static analysis of a plane-strain problem.
- `"static-axisymmetric"` — Creates an axisymmetric (2-D) structural model for static analysis.

For transient analysis, use these values:

- `"transient-solid"` — Creates a structural model for transient analysis of a solid (3-D) problem.
- `"transient-planestress"` — Creates a structural model for transient analysis of a plane-stress problem.
- `"transient-planestrain"` — Creates a structural model for transient analysis of a plane-strain problem.
- `"transient-axisymmetric"` — Creates an axisymmetric (2-D) structural model for transient analysis.

For modal analysis, use these values:

- `"modal-solid"` — Creates a structural model for modal analysis of a solid (3-D) problem.
- `"modal-planestress"` — Creates a structural model for modal analysis of a plane-stress problem.
- `"modal-planestrain"` — Creates a structural model for modal analysis of a plane-strain problem.
- `"modal-axisymmetric"` — Creates an axisymmetric (2-D) structural model for modal analysis.

For frequency response analysis, use these values:

- `"frequency-solid"` — Creates a structural model for frequency response analysis of a solid (3-D) problem.
- `"frequency-planestress"` — Creates a structural model for frequency response analysis of a plane-stress problem.
- `"frequency-planestrain"` — Creates a structural model for frequency response analysis of a plane-strain problem.
- `"frequency-axisymmetric"` — Creates an axisymmetric (2-D) structural model for frequency response analysis.

For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through *r* = 0.

Example: `model = createpde("structural","static-solid")`

Data Types: `char` | `string`

`ThermalAnalysisType` — Type of thermal analysis
`"steadystate"` | `"steadystate-axisymmetric"` | `"transient"` | `"transient-axisymmetric"` | `"modal"` | `"modal-axisymmetric"`

Type of thermal analysis, specified as one of these values:

- `"steadystate"` — Creates a steady-state thermal model. If you do not specify `ThermalAnalysisType` for a thermal model, `createpde` creates a steady-state model.
- `"steadystate-axisymmetric"` — Creates an axisymmetric (2-D) thermal model for steady-state analysis.
- `"transient"` — Creates a transient thermal model.
- `"transient-axisymmetric"` — Creates an axisymmetric (2-D) thermal model for transient analysis.
- `"modal"` — Creates a thermal model for modal analysis. Modal solutions enable you to speed up transient thermal analysis by using the reduced-order modeling (ROM) technique.
- `"modal-axisymmetric"` creates an axisymmetric (2-D) thermal model for modal analysis.

For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through *r* = 0.

Example: `model = createpde("thermal","transient")`

Data Types: `char` | `string`

`ElectromagneticAnalysisType` — Type of electromagnetic analysis
`"electrostatic"` | `"magnetostatic"` | `"harmonic"` | `"conduction"` | `"electrostatic-axisymmetric"` | `"magnetostatic-axisymmetric"` | `"harmonic-axisymmetric"`

Type of electromagnetic analysis, specified as one of these values:

- `"electrostatic"` — Creates a model for electrostatic analysis.
- `"magnetostatic"` — Creates a model for magnetostatic analysis.
- `"harmonic"` creates a model for harmonic electromagnetic analysis.
- `"conduction"` — Creates a model for DC conduction analysis.
- `"electrostatic-axisymmetric"` — Creates an axisymmetric (2-D) model for electrostatic analysis.

- `"magnetostatic-axisymmetric"` — Creates an axisymmetric (2-D) model for magnetostatic analysis.
- `"harmonic-axisymmetric"` — Creates an axisymmetric (2-D) model for harmonic electromagnetic analysis.

For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through $r = 0$.

Example: `model = createpde("electromagnetic","electrostatic")`

Data Types: `char` | `string`

`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`

Output Arguments

`structuralmodel` — Structural model
`StructuralModel` object

Structural model, returned as a `StructuralModel` object.

Example: `structuralmodel = createpde("structural","static-solid")`

`thermalmodel` — Thermal model
`ThermalModel` object

Thermal model, returned as a `ThermalModel` object.

Example: `thermalmodel = createpde("thermal","transient")`

`emagmodel` — Electromagnetic model
`ElectromagneticModel` object

Electromagnetic model, returned as an `ElectromagneticModel` object.

Example: `emagmodel = createpde("electromagnetic","magnetostatic")`

`model` — PDE model
`PDEModel` object

PDE model, returned as a `PDEModel` object.

Example: `model = createpde(2)`

Version History

Introduced in R2015a

Electromagnetic analysis

`createpde` now can create a model for electromagnetic analysis.

Axisymmetric analysis

`createpde` now can create a model for axisymmetric thermal and structural analyses. Axisymmetric analysis simplifies 3-D structural and thermal problems to 2-D using their symmetry around the axis of rotation.

Structural analysis

`createpde` now can create a model for structural analysis.

Thermal analysis

`createpde` now can create a model for thermal analysis.

See Also

`PDEModel` | `ThermalModel` | `StructuralModel` | `ElectromagneticModel`

Topics

"Structural Mechanics"
"Heat Transfer"
"Electromagnetics"
"Solve Problems Using PDEModel Objects" on page 2-2
"Equations You Can Solve Using PDE Toolbox" on page 1-3

createPDEResults

Create solution object

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 `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

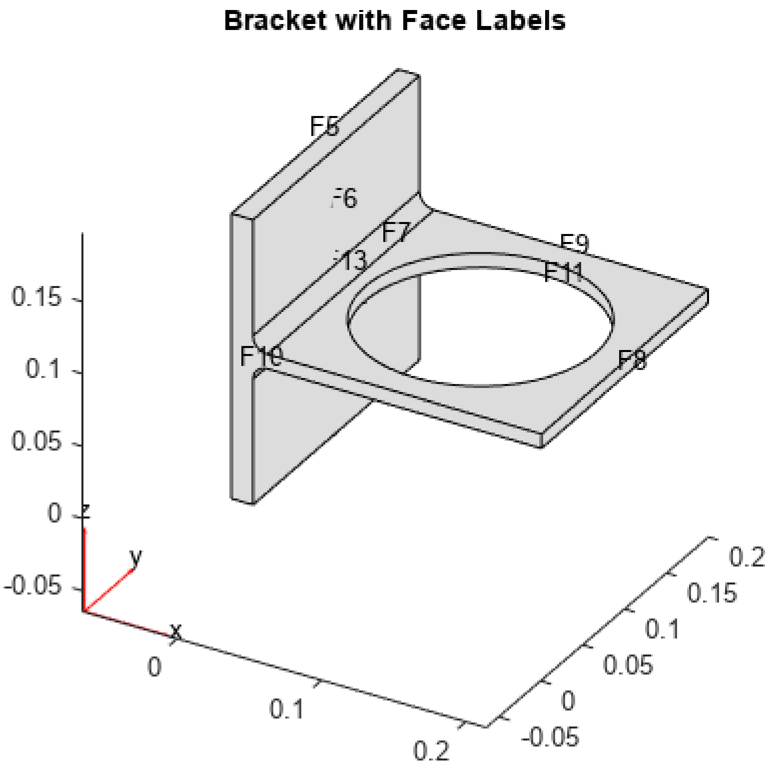
Results From an Elliptic Problem

Create a `StationaryResults` object from the solution to an elliptic system.

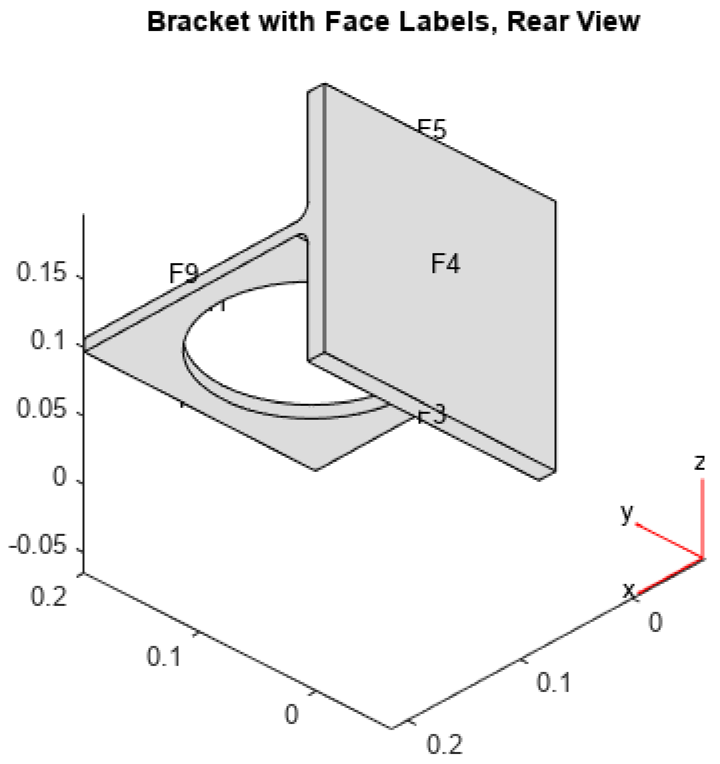
Create a PDE model for a system of three equations. Import the geometry of a bracket and plot the face labels.

```
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")
```



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.

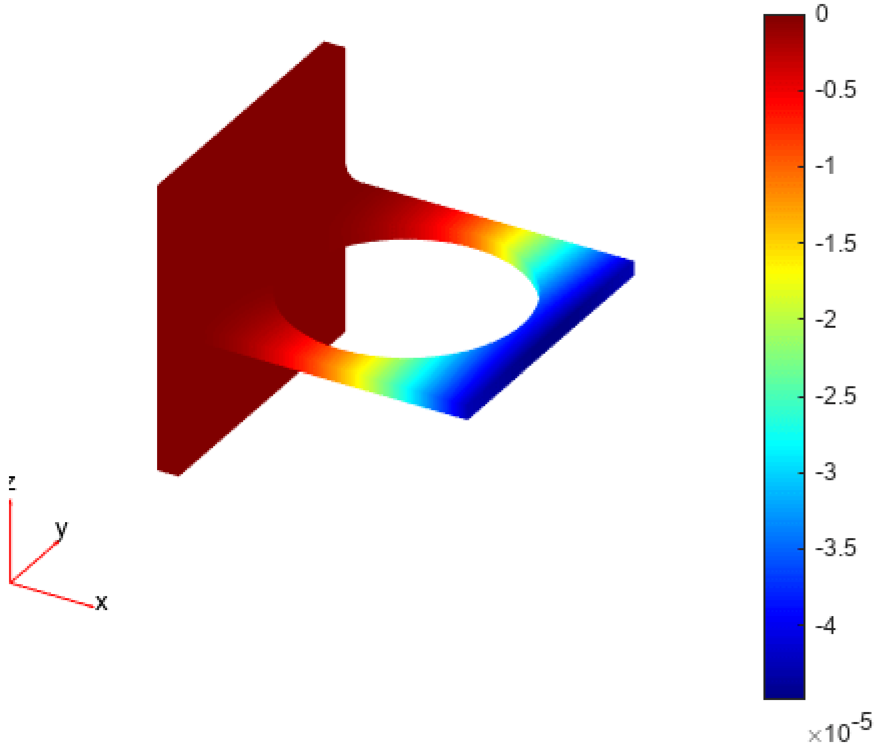
```
results = createPDEResults(model,u)
```

```
results = 
  StationaryResults with properties:

    NodalSolution: [13911x3 double]
       XGradients: [13911x3 double]
       YGradients: [13911x3 double]
```

```
        ZGradients: [13911x3 double]
              Mesh: [1x1 FEMesh]
```

Plot the solution for the z-component, which is component 3.

```
pdeplot3D(model,"ColorMapData",results.NodalSolution(:,3))
```

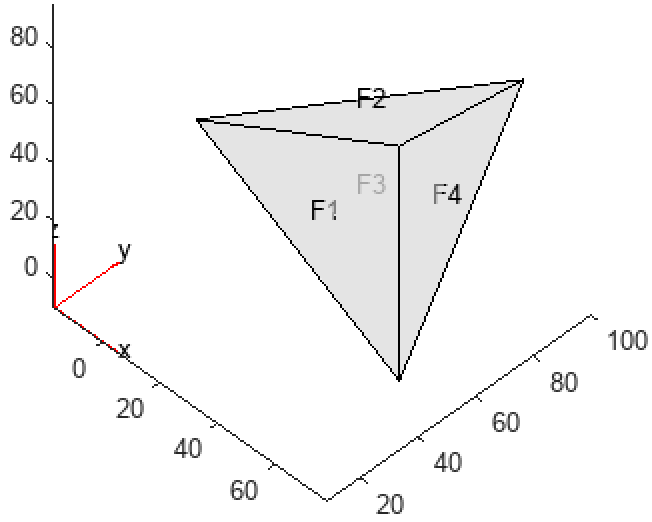


Results from a Time-Dependent Problem

Obtain a solution from a parabolic problem.

The problem models heat flow in a solid.

```
model = createpde();
importGeometry(model,"Tetrahedron.stl");
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);
```

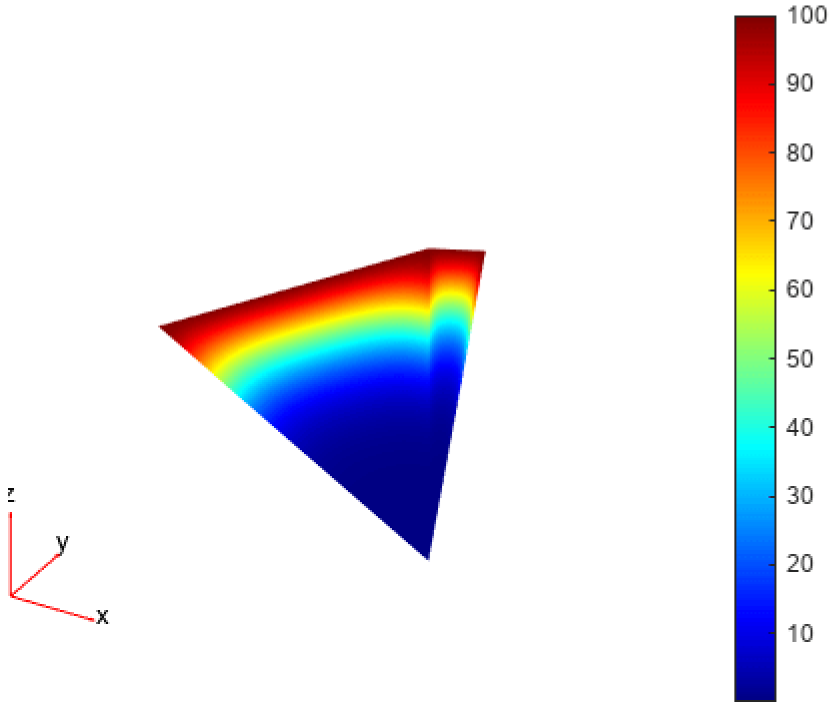
```
168 successful steps
0 failed attempts
329 function evaluations
1 partial derivatives
28 LU decompositions
328 solutions of linear systems
```

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,"GeometricOrder","linear");
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=   0.20,  New conv eig=  0
              Basis= 14,  Time=   0.27,  New conv eig=  0
              Basis= 18,  Time=   0.33,  New conv eig=  1
              Basis= 22,  Time=   0.39,  New conv eig=  2
              Basis= 26,  Time=   0.58,  New conv eig=  3
              Basis= 30,  Time=   0.70,  New conv eig=  5
              Basis= 34,  Time=   0.81,  New conv eig=  5
              Basis= 38,  Time=   0.81,  New conv eig=  7
              Basis= 42,  Time=   0.89,  New conv eig=  8
              Basis= 46,  Time=   1.12,  New conv eig= 11
              Basis= 50,  Time=   1.33,  New conv eig= 12
              Basis= 54,  Time=   1.58,  New conv eig= 14
              Basis= 58,  Time=   1.72,  New conv eig= 14
              Basis= 62,  Time=   1.89,  New conv eig= 16
              Basis= 66,  Time=   2.16,  New conv eig= 18
End of sweep: Basis= 66,  Time=   2.19,  New conv eig= 17
              Basis= 27,  Time=   2.33,  New conv eig=  0
              Basis= 31,  Time=   2.34,  New conv eig=  0
              Basis= 35,  Time=   2.41,  New conv eig=  0
End of sweep: Basis= 35,  Time=   2.53,  New conv eig=  0
```

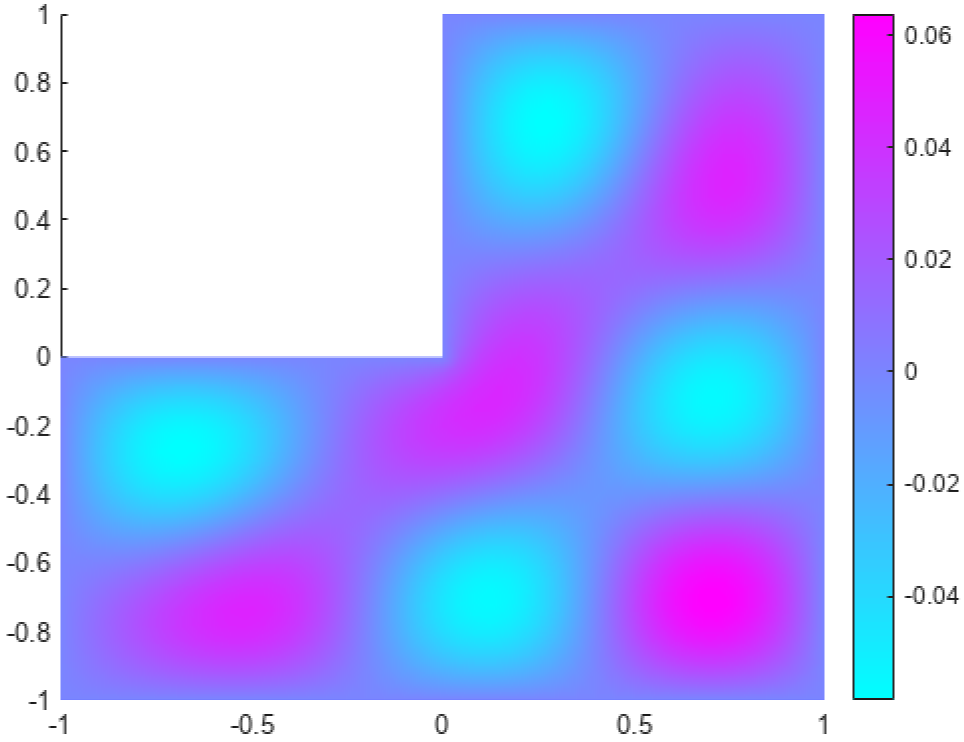
Create an `EigenResults` object from the solution.

```
results = createPDEResults(model,eigenvectors,eigenvalues,"eigen")
```

```
results = 
  EigenResults with properties:

    Eigenvectors: [1440x17 double]
     Eigenvalues: [17x1 double]
            Mesh: [1x1 FEMesh]
```

Plot the solution for mode 10.

```
pdeplot(model,"XYData",results.Eigenvectors(:,10))
```



Input Arguments

`model` — PDE model

`PDEModel` object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde`

`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)`

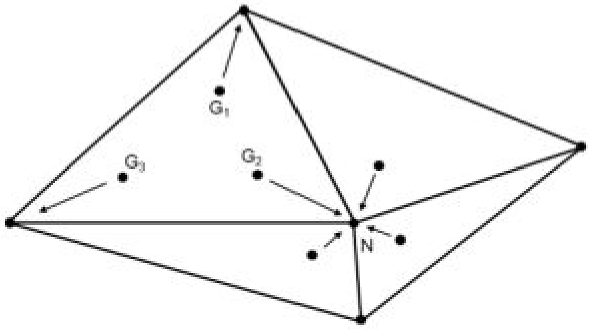
Tips

- 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 3-356.

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.



Version History

Introduced in R2015b

No longer creates an object of type `PDEResults`

Behavior change in future release

`createPDEResults` no longer creates an object of type `PDEResults`. The syntax of `createPDEResults` has changed to accommodate creating the new result types for time-dependent and eigenvalue problems.

- To create the `TimeDependentResults` object for a time-dependent problem, use the syntax `createPDEResults(pdem,u,utimes,'time-dependent')`, where `utimes` is a vector of solution times.
- To create the `EigenResults` object for an eigenvalue problem, use the syntax `createPDEResults(pdem,eigenvectors,eigenvalues,'eigen')`.

`EigenResults` has different property names than `PDEResults`. Update any eigenvalue scripts that use `PDEResults` property names.

See Also

`interpolateSolution` | `evaluateGradient` | `StationaryResults` | `TimeDependentResults` | `EigenResults`

Topics

"Linear Elasticity Equations" on page 3-156

csgdel

Delete boundaries between subdomains

Syntax

```
[dl1,bt1] = csgdel(dl,bt,bl)
[dl1,bt1] = csgdel(dl,bt)
```

Description

`[dl1,bt1] = csgdel(dl,bt,bl)` deletes the boundaries between subdomains specified in `bl`. If deleting the boundaries in `bl` makes the decomposed geometry matrix inconsistent, then `csgdel` deletes additional border segments (edge segments between subdomains) to preserve consistency.

Deleting boundaries typically changes the edge IDs of the remaining boundaries.

`csgdel` does not delete boundary segments (outer boundaries).

`[dl1,bt1] = csgdel(dl,bt)` deletes all boundaries between subdomains.

Examples

Delete Edges to Merge Faces of 2-D Geometry

Delete edges in a 2-D geometry created in the PDE Modeler app and exported to the MATLAB workspace.

Create a geometry in the PDE Modeler app by entering the following commands in the MATLAB Command Window:

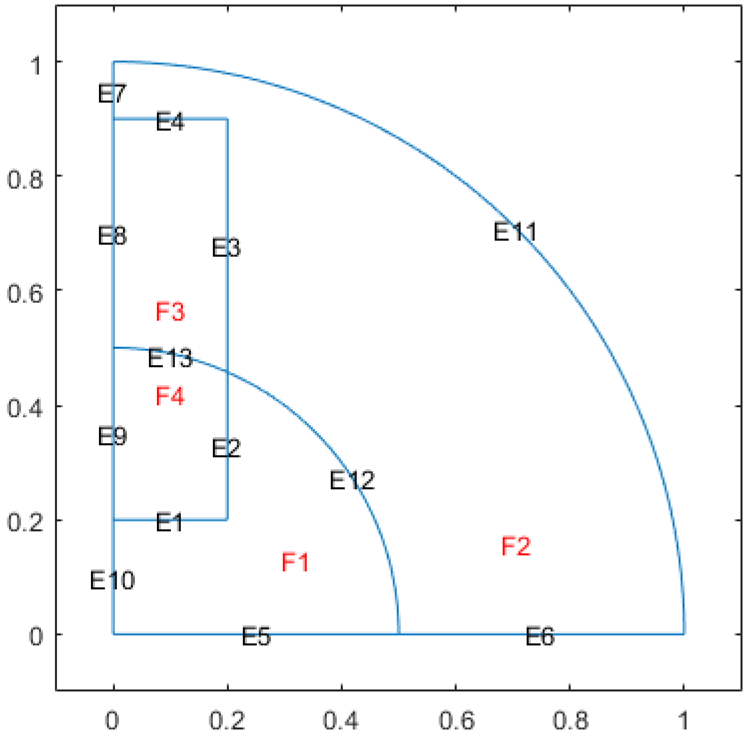
```
pdecirc(0,0,1,"C1")
pdecirc(0,0,0.5,"C2")
pderect([-0.2 0.2 0.2 0.9],"R1")
pderect([0 1 0 1],"SQ1")
```

Reduce the geometry to the first quadrant by intersecting it with a square. To do this, enter `(C1+C2+R1)*SQ1` in the **Set formula** field.

From the PDE Modeler app, export the geometry description matrix, set formula, and name-space matrix to the MATLAB workspace by selecting **Export Geometry Description, Set Formula, Labels** from the **Draw** menu.

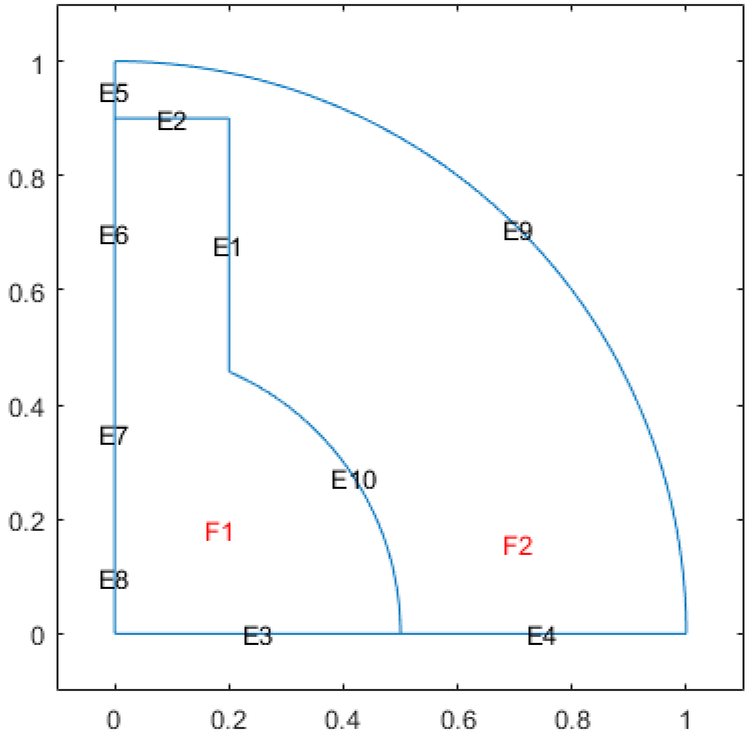
In the MATLAB Command Window, use the `decsg` function to decompose the exported geometry into minimal regions. This creates an `AnalyticGeometry` object `dl`. Plot `dl`.

```
[dl,bt] = decsg(gd,sf,ns);
pdegplot(dl,"EdgeLabels","on","FaceLabels","on")
xlim([-0.1 1.1])
ylim([-0.1 1.1])
```



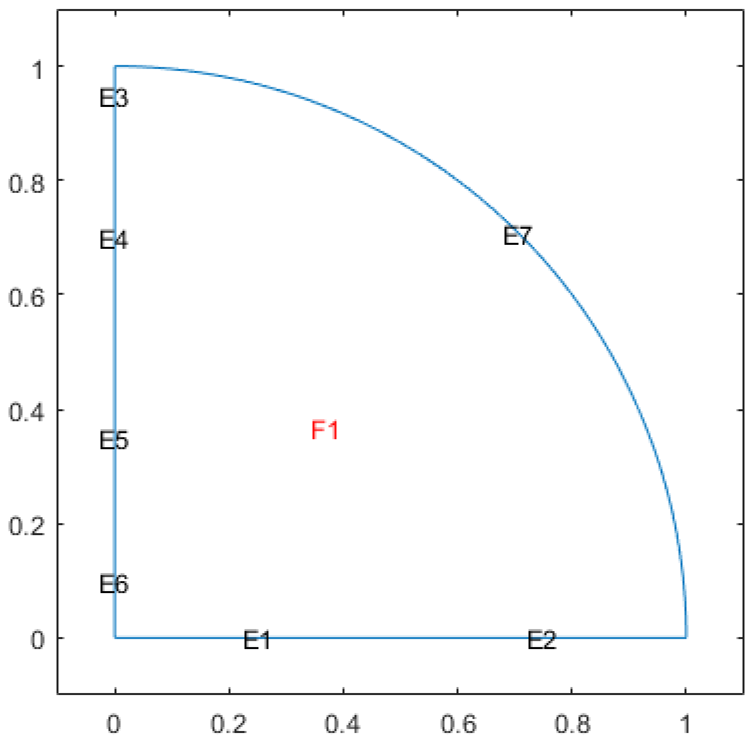
Remove edges 1, 2, and 13 using the `csgdel` function. Specify the edges to delete as a vector of edge IDs. Plot the resulting geometry.

```
[dl1,bt1] = csgdel(dl,bt,[1 2 13]);
pdegplot(dl1,"EdgeLabels","on","FaceLabels","on")
xlim([-0.1 1.1])
ylim([-0.1 1.1])
```



Now remove all boundaries between subdomains and plot the resulting geometry.

```
[dl1,bt1] = csgdel(dl,bt);
pdegplot(dl1,"EdgeLabels","on","FaceLabels","on")
xlim([-0.1 1.1])
ylim([-0.1 1.1])
```



Input 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 (subdomains) 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 columns 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`

`bl` — Boundaries to delete
positive integer | vector of positive integers

Boundaries to delete, specified as a positive integer or a vector of positive integers. Each integer represents a boundary ID.

Data Types: `double`

Output Arguments

`dl1` — Modified decomposed geometry matrix
matrix of double-precision numbers

Modified decomposed geometry matrix, returned as a matrix of double-precision numbers.

Data Types: `double`

`bt1` — Boolean table relating remaining original shapes to minimal regions
matrix of 1s and 0s

Boolean table relating the remaining original shapes to the minimal regions, returned as a matrix of 1s and 0s.

Data Types: `double`

Version History

Introduced before R2006a

See Also

`decsg`

decsg

Decompose constructive solid 2-D geometry into minimal regions

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 Modeler 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 “2-D Geometry Creation at Command Line” on page 2-16 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 Modeler App

Create a 2-D geometry in the PDE Modeler app, then export it to the MATLAB workspace and decompose it to minimal regions by using `decsg`.

Start the PDE Modeler 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.

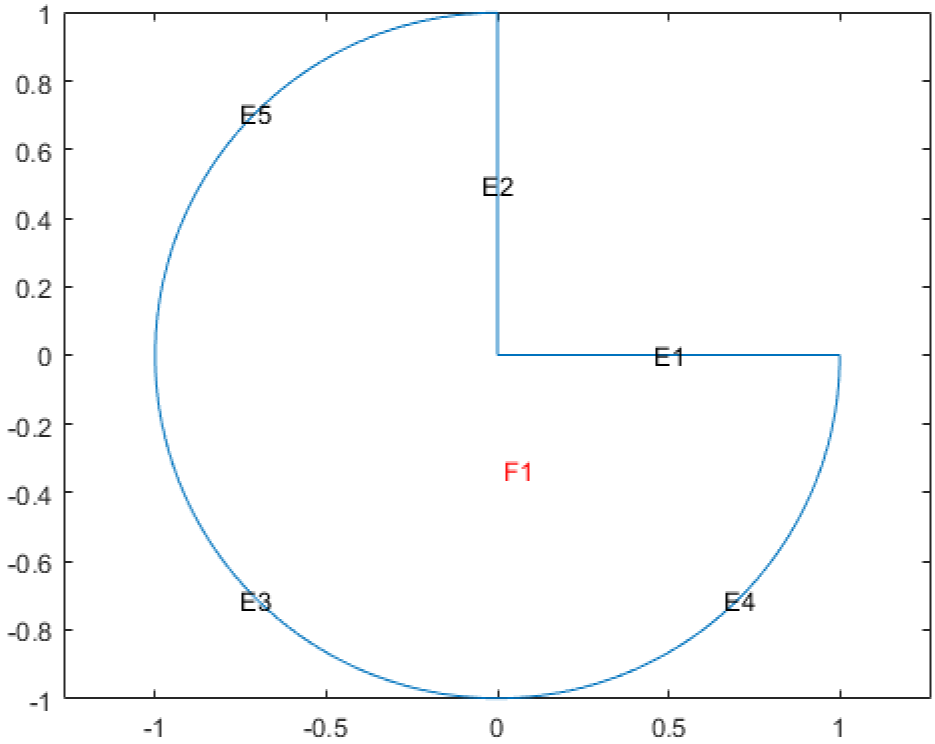
Decompose the exported geometry into minimal regions. The result is one minimal region with five edge segments: three circle edge segments and two line edge segments.

```
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 face labels.

```
pdegplot(dl,"EdgeLabels","on","FaceLabels","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`.

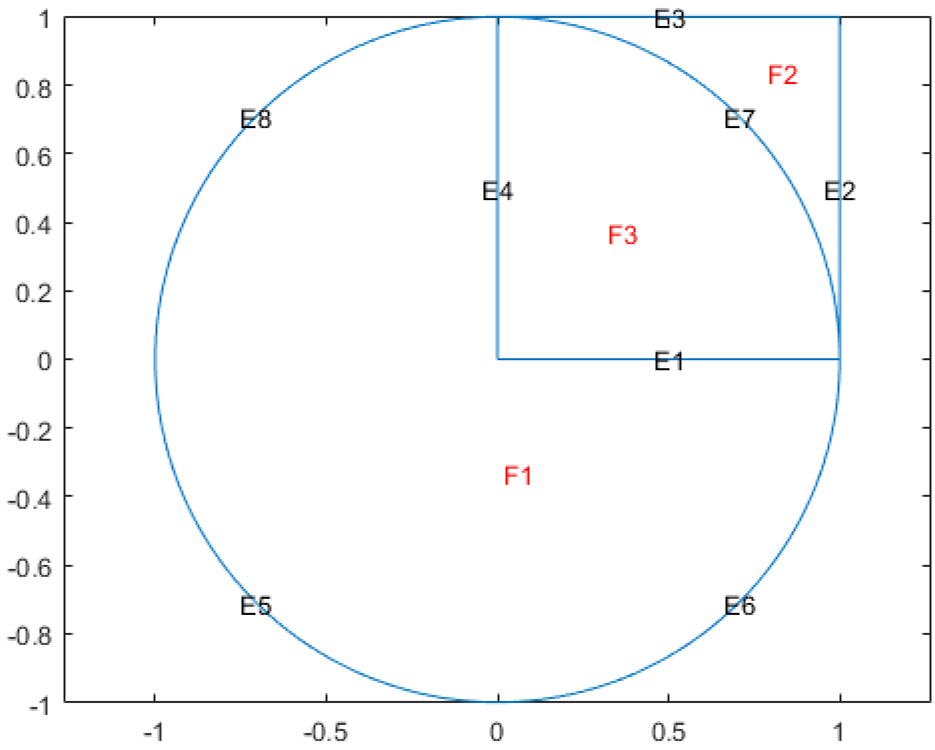
```
dl_all = decsg(gd)
```

```
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","FaceLabels","on")
axis equal
```



Remove Boundaries Between Subdomains

Start the PDE Modeler 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.

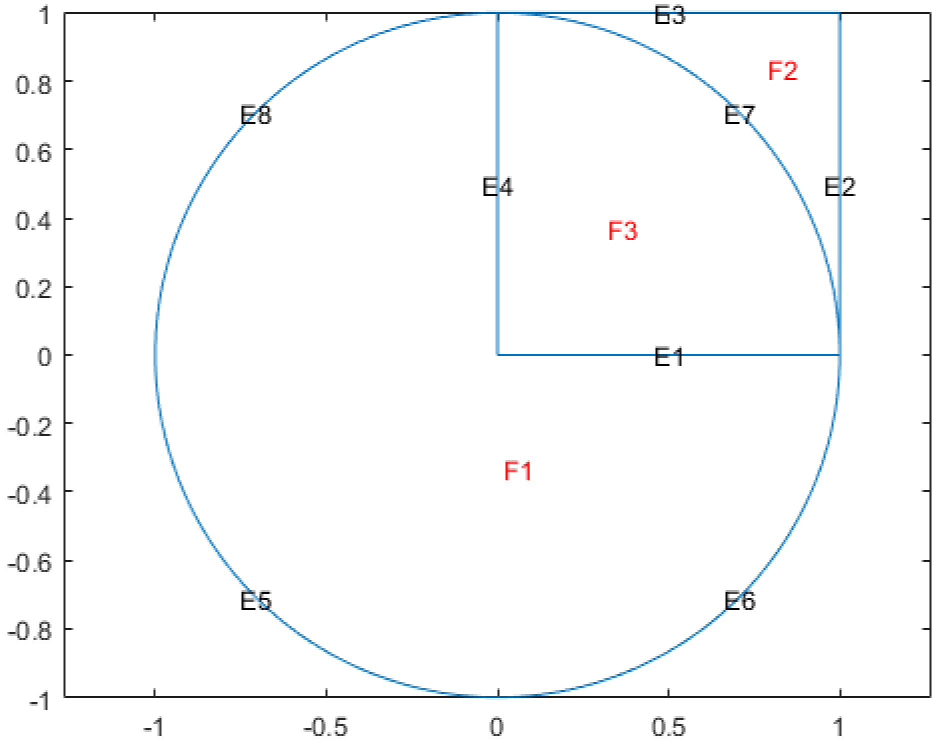
Decompose the exported geometry into minimal regions. Because the geometry is a union of all regions, `C1+SQ1`, you can omit the arguments specifying the set formula and name-space matrix when using `decsg`.

```
[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 face labels.

```
pdegplot(dl,"EdgeLabels","on","FaceLabels","on")
axis equal
```

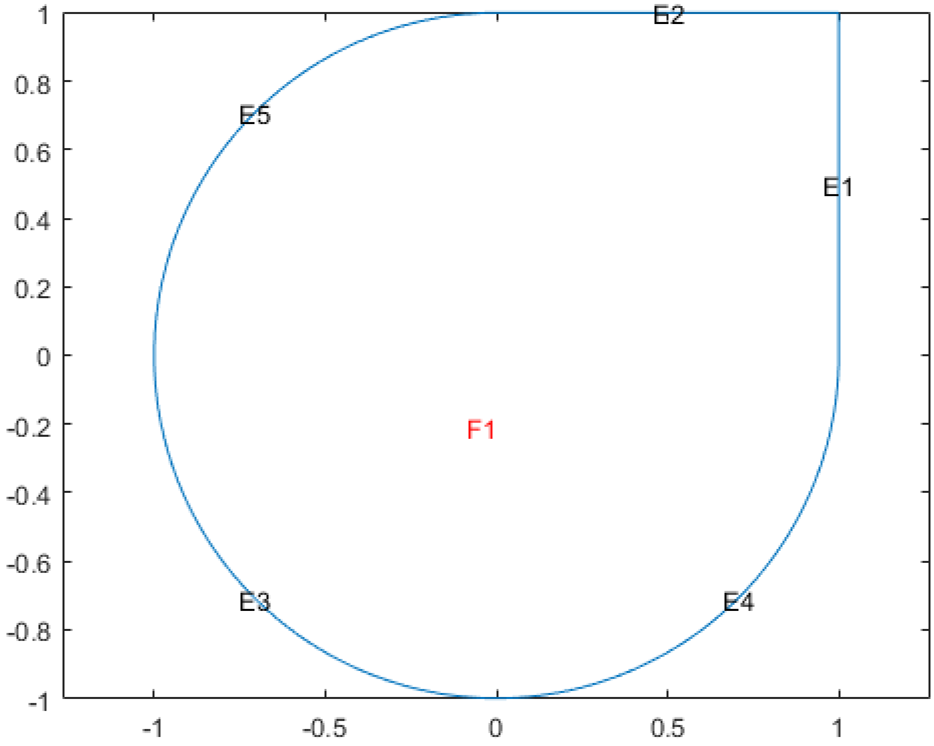


Remove the subdomain boundaries by using the `csgdel` function.

```
[dl2,bt2] = csgdel(dl,bt);
```

View the resulting geometry.

```
figure
pdegplot(dl2,"EdgeLabels","on","FaceLabels","on")
axis equal
```



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 Modeler 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

character vector | string scalar

Set formula, specified as a character vector or a string including 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 Modeler 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` | `string`

`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 Modeler 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.
- `decsg` returns `NaN` if it cannot evaluate the set formula `sf`.

Version History

Introduced before R2006a

See Also

`geometryFromEdges` | `csgdel` | `wgeom` | `pdecirc` | `pdeellip` | `pdepoly` | `pderect` | **PDE Modeler**

Topics

"2-D Geometry Creation at Command Line" on page 2-16

DiscreteGeometry Properties

Discrete 2-D or 3-D geometry description

Description

`DiscreteGeometry` describes a 2-D or 3-D geometry in the form of a discrete geometry object. `PDEModel`, `StructuralModel`, and `ThermalModel` objects have a `Geometry` property, which can be an `AnalyticGeometry` or `DiscreteGeometry` object.

Create a discrete geometry for your model by using one of the following approaches:

- Use `importGeometry` to import a 2-D or 3-D geometry from an STL file or a 3-D geometry from a STEP file and attach it to the model.
- Use `geometryFromMesh` to reconstruct a 2-D or 3-D geometry from mesh and attach it to the model.
- Use `multicuboid`, `multicylinder`, or `multisphere` to create a 3-D geometry. Then assign the resulting geometry 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;
  ```
- Use `extrude` to create a 3-D geometry by vertically extruding a 2-D geometry.

Properties

Properties

`NumCells` — Number of geometry cells
nonnegative integer

Number of geometry cells, specified as a nonnegative integer.

Data Types: `double`

`NumEdges` — Number of geometry edges
nonnegative integer

Number of geometry edges, specified as a nonnegative integer.

Data Types: `double`

`NumFaces` — Number of geometry faces
positive integer

Number of geometry faces, specified as a positive integer.

Data Types: `double`

`NumVertices` — Number of geometry vertices
nonnegative integer

Number of geometry vertices, specified as a nonnegative integer.

Data Types: `double`

`Vertices` — Coordinates of geometry vertices
`N`-by-2 numeric matrix | `N`-by-3 numeric matrix

Coordinates of geometry vertices, specified as an `N`-by-2 or `N`-by-3 numeric matrix for a 2-D or 3-D geometry, respectively. Here, `N` is the number of vertices.

Data Types: `double`

Version History

Introduced in R2015a

See Also

`addFace` | `addVertex` | `geometryFromMesh` | `importGeometry` | `multicuboid` | `multicylinder` | `multisphere` | `PDEModel` | `StructuralModel` | `ThermalModel` | AnalyticGeometry Properties

Topics

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

dst

(Not recommended) Discrete sine transform

Note `dst` is not recommended.

Syntax

```
y = dst(x)
y = dst(x,n)
```

Description

`y = dst(x)` computes the discrete sine transform of `x` according to the equation

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

If `x` is a matrix, then `dst` applies to each column. For best performance, the number of rows in `x` must be $2^m - 1$, where m is an integer.

`y = dst(x,n)` truncates the vector `x` or pads it with trailing zeros to length `n` before computing the transform. If `x` is a matrix, then `dst` truncates or pads each column.

Examples

Discrete Sine Transform

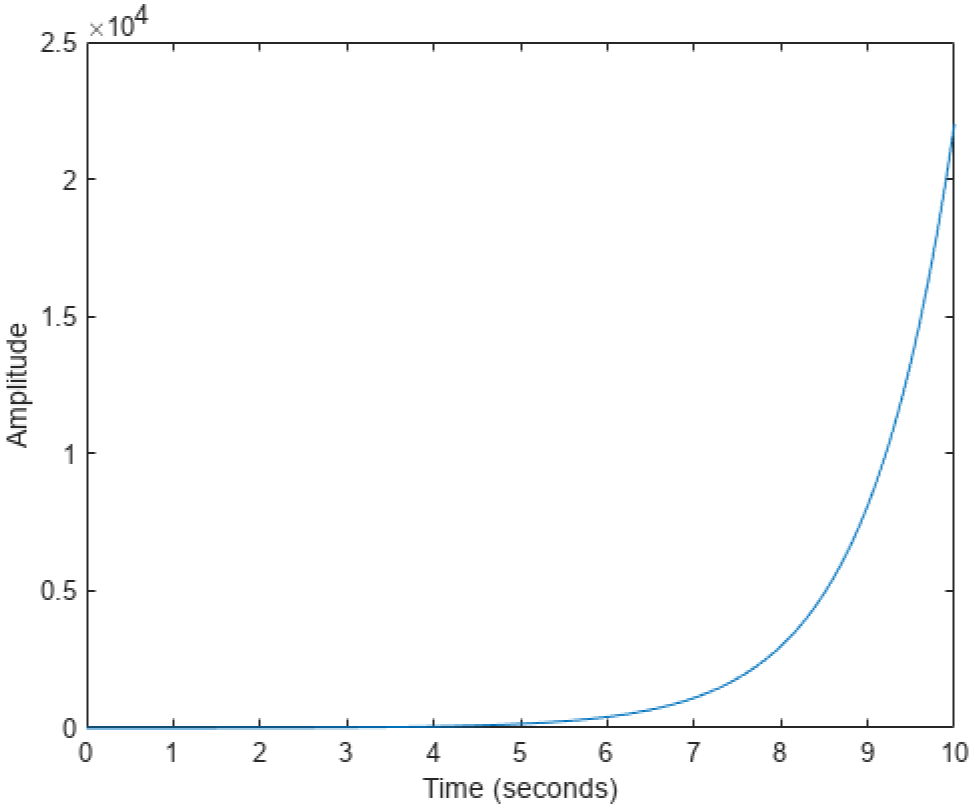
Find the discrete sine transform of the exponential by using `dst`, and then invert the result by using `idst`.

Create a time vector sampled in increments of 0.1 second over a period of 10 seconds.

```
Ts = 0.1;
t = 0:Ts:10;
```

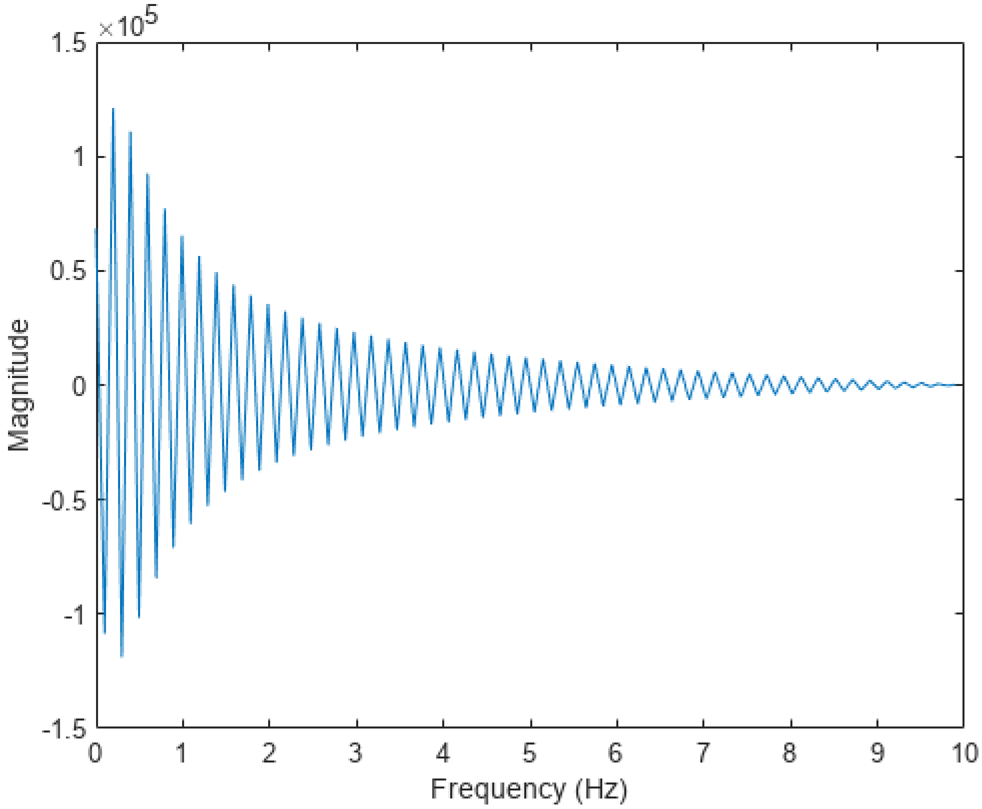
Compute and plot the exponential signal.

```
x = exp(t);
plot(t,x)
xlabel('Time (seconds)')
ylabel('Amplitude')
```



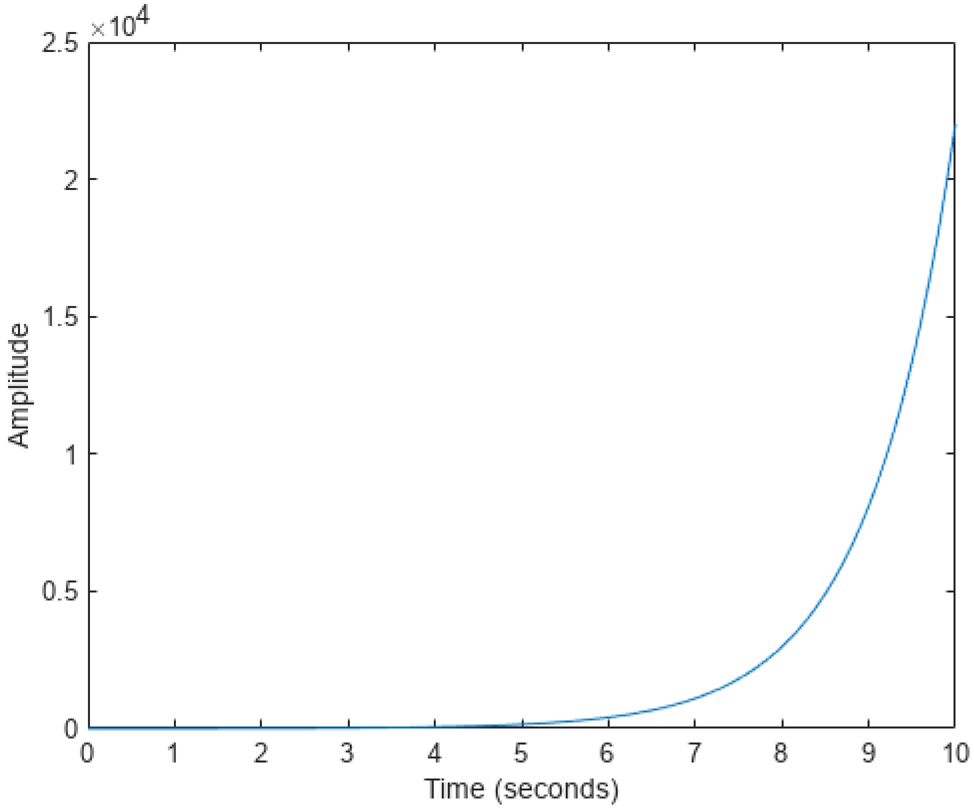
Compute the discrete sign transform of the signal, and create the vector `f` that corresponds to the sampling of the signal in frequency space.

```
y = dst(x);
fs = 1/Ts;
f = (0:length(y)-1)*fs/length(y);
plot(f,y)
xlabel('Frequency (Hz)')
ylabel('Magnitude')
```



Compute the inverse discrete sine transform of `y`, and plot the result.

```
z = idst(y);
figure
plot(t,z)
xlabel('Time (seconds)')
ylabel('Amplitude')
```



Input Arguments

x — Input array

vector | matrix

Input array, specified as a vector or a matrix. If `x` is a matrix, then `dst` applies to each column.

Data Types: `double`

n — Transform length

nonnegative integer

Transform length, specified as a nonnegative integer. If `x` is a vector, then `dst` truncates it or pads it with trailing zeros, so that the resulting vector has `n` elements. If `x` is a matrix, then `dst` truncates or pads each column.

Data Types: `double`

Output Arguments

y — Discrete sine transform coefficients

row vector | matrix

Discrete sine transform coefficients, returned as a vector or matrix of the same size as `x`.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`dst` is not recommended. There are no plans to remove `dst`.

See Also

`idst`

idst

(Not recommended) Invert discrete sine transform

Note `idst` is not recommended.

Syntax

```
x = idst(y)
x = idst(y,n)
```

Description

`x = idst(y)` computes the inverse discrete sine transform of `y` according to the equation

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

If `y` is a matrix, `idst` applies to each column. For best performance, the number of rows in `y` must be $2^m - 1$, where m is an integer.

`x = idst(y,n)` truncates the vector `y` or pads it with zeros to length `n` before computing the transform.

Examples

Discrete Sine Transform

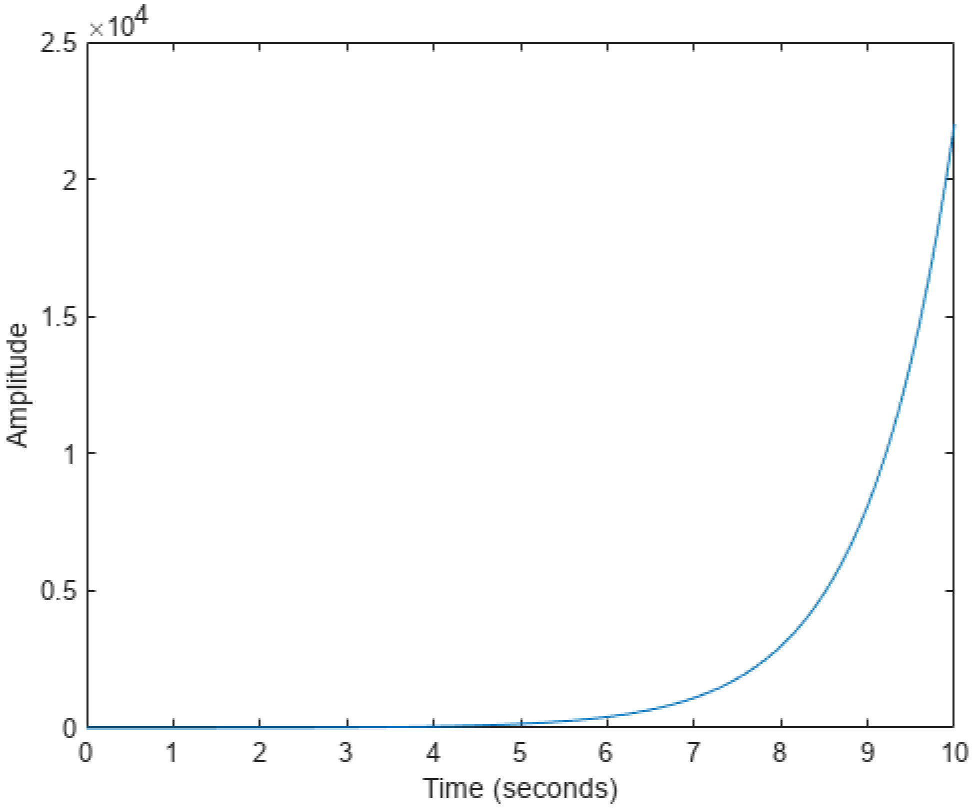
Find the discrete sine transform of the exponential by using `dst`, and then invert the result by using `idst`.

Create a time vector sampled in increments of 0.1 second over a period of 10 seconds.

```
Ts = 0.1;
t = 0:Ts:10;
```

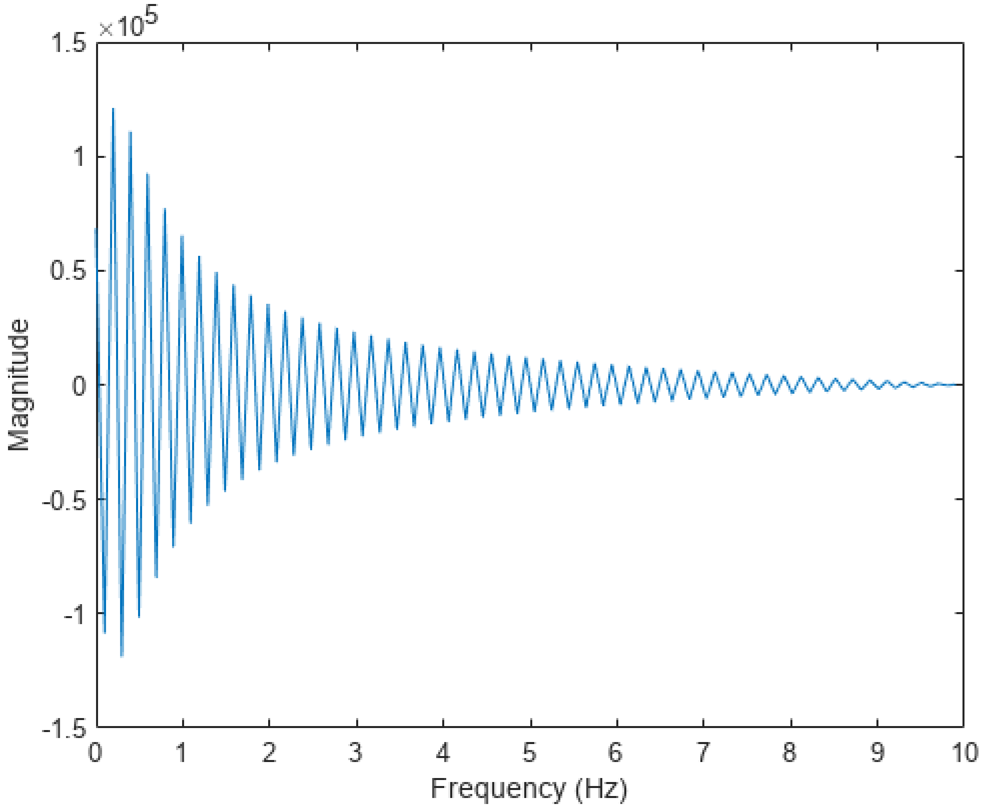
Compute and plot the exponential signal.

```
x = exp(t);
plot(t,x)
xlabel('Time (seconds)')
ylabel('Amplitude')
```



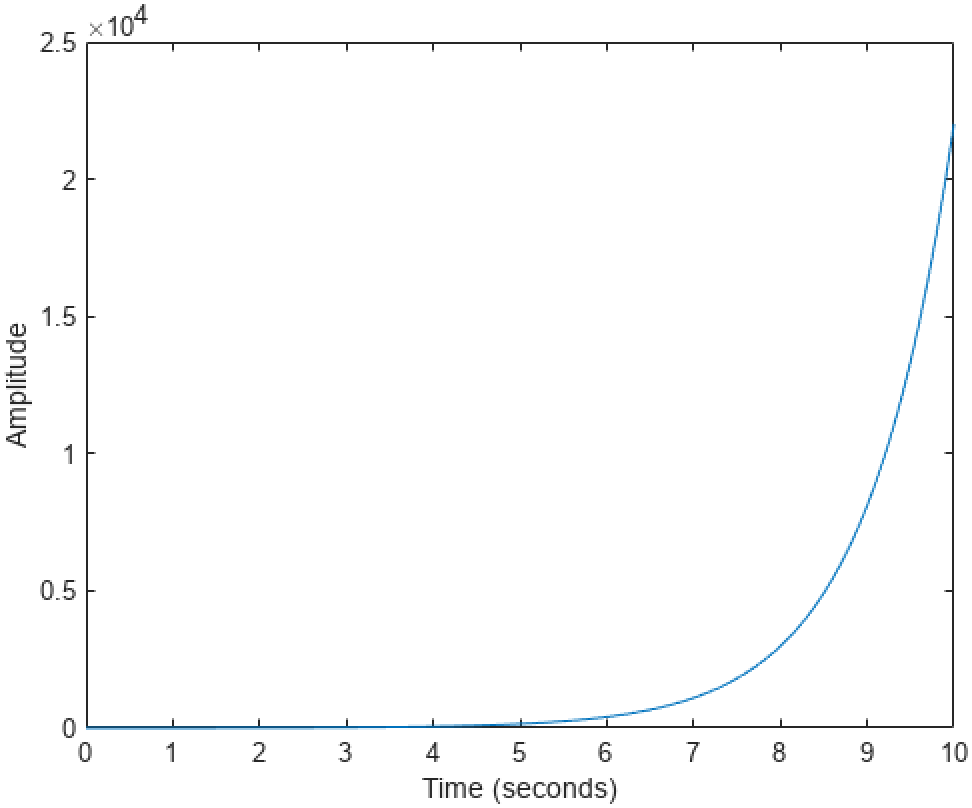
Compute the discrete sign transform of the signal, and create the vector `f` that corresponds to the sampling of the signal in frequency space.

```
y = dst(x);
fs = 1/Ts;
f = (0:length(y)-1)*fs/length(y);
plot(f,y)
xlabel('Frequency (Hz)')
ylabel('Magnitude')
```



Compute the inverse discrete sine transform of `y`, and plot the result.

```
z = idst(y);
figure
plot(t,z)
xlabel('Time (seconds)')
ylabel('Amplitude')
```



Input Arguments

y — Input array

vector | matrix

Input array, specified as a vector or a matrix. If `y` is a matrix, then `idst` applies to each column.

Data Types: `double`

n — Transform length

nonnegative integer

Transform length, specified as a nonnegative integer. If `y` is a vector, then `idst` truncates it or pads it with trailing zeros, so that the resulting vector has `n` elements. If `y` is a matrix, then `idst` truncates or pads each column.

Data Types: `double`

Output Arguments

x — Inverse discrete sine transform coefficients

vector | matrix

Inverse discrete sine transform coefficients, returned as a vector or matrix of the same size as `y`.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`idst` is not recommended. There are no plans to remove `idst`.

See Also

`dst`

EigenResults

PDE eigenvalue solution and derived quantities

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.

Creation

There are several ways to create an `EigenResults` object:

- Solve an eigenvalue problem using the `solvepdeeig` function. This function returns a PDE eigenvalue solution as an `EigenResults` object. This is the recommended approach.
- Solve an eigenvalue problem using the `pdeeig` function. Then use the `createPDEResults` function to obtain an `EigenResults` object from a PDE eigenvalue solution returned by `pdeeig`. Note that `pdeeig` is a legacy function. It is not recommended for solving eigenvalue problems.

Properties

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as a FEMesh object.

`Eigenvectors` — Solution eigenvectors
matrix | 3-D array

This property is read-only.

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 3-356.

Data Types: `double`

`Eigenvalues` — Solution eigenvalues
vector

This property is read-only.

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`

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.

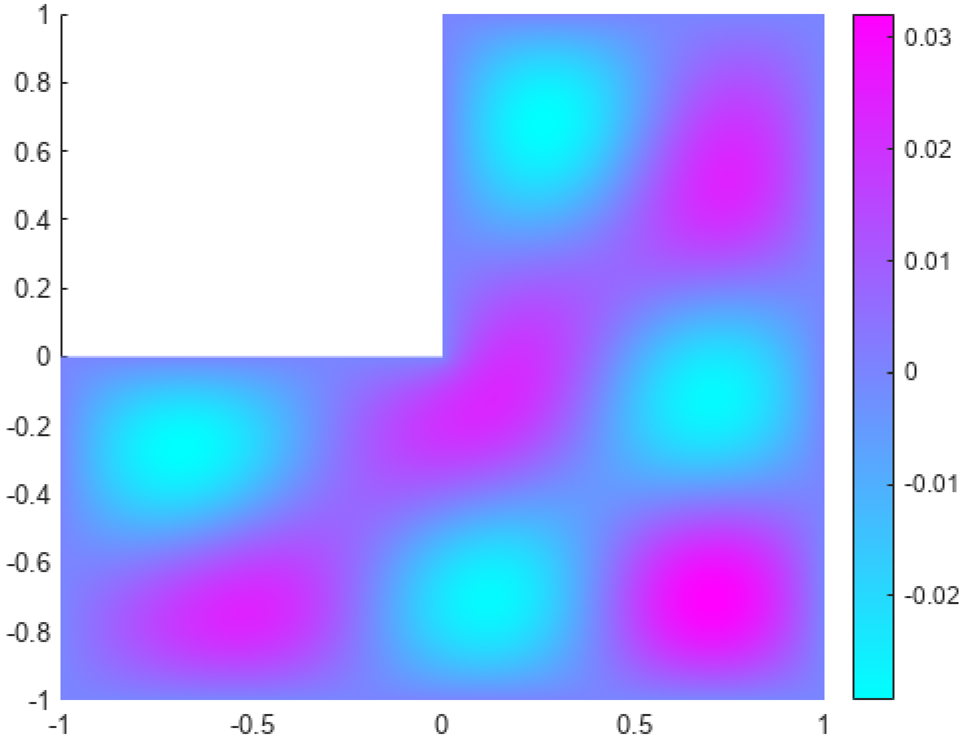
```
generateMesh(model,'Hmax',0.05);
ev = [0,100];
results = solvepdeeig(model,ev)
```

```
              Basis= 10,  Time=   0.16,  New conv eig=  0
              Basis= 11,  Time=   0.17,  New conv eig=  0
              Basis= 12,  Time=   0.19,  New conv eig=  0
              Basis= 13,  Time=   0.19,  New conv eig=  0
              Basis= 14,  Time=   0.19,  New conv eig=  0
              Basis= 15,  Time=   0.20,  New conv eig=  0
              Basis= 16,  Time=   0.20,  New conv eig=  0
              Basis= 17,  Time=   0.20,  New conv eig=  1
              Basis= 18,  Time=   0.20,  New conv eig=  1
              Basis= 19,  Time=   0.38,  New conv eig=  1
              Basis= 20,  Time=   0.38,  New conv eig=  1
              Basis= 21,  Time=   0.38,  New conv eig=  1
              Basis= 22,  Time=   0.39,  New conv eig=  2
              Basis= 23,  Time=   0.39,  New conv eig=  3
              Basis= 24,  Time=   0.39,  New conv eig=  3
              Basis= 25,  Time=   0.70,  New conv eig=  3
              Basis= 26,  Time=   0.70,  New conv eig=  3
              Basis= 27,  Time=   0.70,  New conv eig=  4
              Basis= 28,  Time=   0.70,  New conv eig=  4
              Basis= 29,  Time=   1.02,  New conv eig=  4
              Basis= 30,  Time=   1.02,  New conv eig=  4
              Basis= 31,  Time=   1.02,  New conv eig=  5
              Basis= 32,  Time=   1.02,  New conv eig=  5
              Basis= 33,  Time=   1.05,  New conv eig=  5
              Basis= 34,  Time=   1.05,  New conv eig=  5
```

```
              Basis= 35,  Time=   1.11,  New conv eig=  6
              Basis= 36,  Time=   1.11,  New conv eig=  6
              Basis= 37,  Time=   1.11,  New conv eig=  7
              Basis= 38,  Time=   1.42,  New conv eig=  7
              Basis= 39,  Time=   1.42,  New conv eig=  7
              Basis= 40,  Time=   1.42,  New conv eig=  7
              Basis= 41,  Time=   1.42,  New conv eig=  7
              Basis= 42,  Time=   1.44,  New conv eig=  7
              Basis= 43,  Time=   1.44,  New conv eig=  8
              Basis= 44,  Time=   1.44,  New conv eig=  8
              Basis= 45,  Time=   1.83,  New conv eig=  8
              Basis= 46,  Time=   1.83,  New conv eig= 11
              Basis= 47,  Time=   1.83,  New conv eig= 11
              Basis= 48,  Time=   1.83,  New conv eig= 11
              Basis= 49,  Time=   2.16,  New conv eig= 11
              Basis= 50,  Time=   2.16,  New conv eig= 12
              Basis= 51,  Time=   2.19,  New conv eig= 12
              Basis= 52,  Time=   2.19,  New conv eig= 12
              Basis= 53,  Time=   2.19,  New conv eig= 13
              Basis= 54,  Time=   2.22,  New conv eig= 14
              Basis= 55,  Time=   2.22,  New conv eig= 14
              Basis= 56,  Time=   2.45,  New conv eig= 14
              Basis= 57,  Time=   2.55,  New conv eig= 14
              Basis= 58,  Time=   2.55,  New conv eig= 14
              Basis= 59,  Time=   2.56,  New conv eig= 15
              Basis= 60,  Time=   2.56,  New conv eig= 16
              Basis= 61,  Time=   2.77,  New conv eig= 16
              Basis= 62,  Time=   2.77,  New conv eig= 16
              Basis= 63,  Time=   2.77,  New conv eig= 16
              Basis= 64,  Time=   2.78,  New conv eig= 16
              Basis= 65,  Time=   2.78,  New conv eig= 16
              Basis= 66,  Time=   2.94,  New conv eig= 18
              Basis= 67,  Time=   3.12,  New conv eig= 18
              Basis= 68,  Time=   3.30,  New conv eig= 18
              Basis= 69,  Time=   3.41,  New conv eig= 18
              Basis= 70,  Time=   3.45,  New conv eig= 18
              Basis= 71,  Time=   3.45,  New conv eig= 19
              Basis= 72,  Time=   3.50,  New conv eig= 20
              Basis= 73,  Time=   3.52,  New conv eig= 21
End of sweep: Basis= 73,  Time=   3.52,  New conv eig= 21
              Basis= 31,  Time=   3.81,  New conv eig=  0
              Basis= 32,  Time=   4.11,  New conv eig=  0
              Basis= 33,  Time=   4.11,  New conv eig=  0
              Basis= 34,  Time=   4.11,  New conv eig=  0
              Basis= 35,  Time=   4.11,  New conv eig=  0
              Basis= 36,  Time=   4.12,  New conv eig=  0
              Basis= 37,  Time=   4.12,  New conv eig=  0
End of sweep: Basis= 37,  Time=   4.12,  New conv eig=  0

results = 
  EigenResults with properties:

    Eigenvectors: [5597x19 double]
     Eigenvalues: [19x1 double]
            Mesh: [1x1 FEMesh]
```

Plot the solution for mode 10.

```
pdeplot(model,'XYData',results.Eigenvectors(:,10))
```



Version History

Introduced in R2016a

See Also

`solvepdeeig` | `StationaryResults` | `TimeDependentResults`

Topics

"Eigenvalues and Eigenmodes of L-Shaped Membrane" on page 3-305
"Eigenvalues and Eigenmodes of Square" on page 3-317
"Solve Problems Using PDEModel Objects" on page 2-2

ElectromagneticModel

Electromagnetic model object

Description

An `ElectromagneticModel` object contains information about an electromagnetic analysis problem: the geometry, material properties, electromagnetic sources, boundary conditions, and mesh.

Creation

To create a `ElectromagneticModel` object, use the `createpde` function and specify `"electromagnetic"` as its first argument .

Properties

`AnalysisType` — Type of electromagnetic analysis
`'electrostatic'` | `'magnetostatic'` | `'harmonic'` | `'conduction'` | `'electrostatic-axisymmetric'` | `'magnetostatic-axisymmetric'` | `'harmonic-axisymmetric'`

Type of electromagnetic analysis, specified as `'electrostatic'`, `'magnetostatic'`, `'harmonic'`, `'conduction'`, `'electrostatic-axisymmetric'`, `'magnetostatic-axisymmetric'`, or `'harmonic-axisymmetric'`.

To change an electromagnetic analysis type, assign a new type to `model.AnalysisType`. Ensure that all other properties of the model are consistent with the new analysis type.

`Geometry` — Geometry description
`AnalyticGeometry` | `DiscreteGeometry`

Geometry description, specified as an `AnalyticGeometry` or `DiscreteGeometry` object.

`MaterialProperties` — Material properties within domain
object containing material property assignments

Material properties within the domain, specified as an object containing the material property assignments.

`Sources` — Electromagnetic sources within the domain or subdomain
object containing heat source assignments

Electromagnetic source within the domain or subdomain, specified as an object containing electromagnetic source assignments.

`BoundaryConditions` — Boundary conditions applied to geometry
object containing boundary condition assignments

Boundary conditions applied to the geometry, specified as an object containing the boundary condition assignments.

`VacuumPermittivity` — Permittivity of vacuum for entire model

number

Permittivity of vacuum for the entire model, specified as a number. This value must be consistent with the units of the model. If the model parameters are in the SI system of units, then the permittivity of vacuum must be 8.8541878128E-12.

`VacuumPermeability` — Permeability of vacuum for entire model

number

Permeability of vacuum for the entire model, specified as a number. This value must be consistent with the units of the model. If the model parameters are in the SI system of units, then the permeability of vacuum must be 1.2566370614E-6.

`Mesh` — Mesh for solution

`FEMesh` object

Mesh for the solution, specified as a `FEMesh` object. See FEMesh. You create the mesh using the `generateMesh` function. For a 3-D magnetostatic model, the mesh must be linear.

`SolverOptions` — Algorithm options for PDE solvers

`PDESolverOptions` object

Algorithm options for the PDE solvers, specified as a `PDESolverOptions` object. The properties of a `PDESolverOptions` object include absolute and relative tolerances for internal ODE solvers, maximum solver iterations, and so on. For details, see PDESolverOptions.

`FieldType` — Type of field for harmonic analysis

`'electric'` (default) | `'magnetic'`

Type of field for a harmonic analysis, specified as `'electric'` or `'magnetic'`.

Object Functions

geometryFromEdges	Create 2-D geometry from decomposed geometry matrix
geometryFromMesh	Create 2-D or 3-D geometry from mesh
importGeometry	Import geometry from STL or STEP file
generateMesh	Create triangular or tetrahedral mesh
electromagneticProperties	Assign properties of material for electromagnetic model
electromagneticSource	Specify current density, charge density, and magnetization for electromagnetic model
electromagneticBC	Apply boundary conditions to electromagnetic model
solve	Solve heat transfer, structural analysis, or electromagnetic analysis problem

Examples

Create Electromagnetic Model

Create a model for electrostatic analysis.

```
emagE = createpde("electromagnetic","electrostatic")
```

```
emagE = 
  ElectromagneticModel with properties:

          AnalysisType: "electrostatic"
              Geometry: []
    MaterialProperties: []
               Sources: []
    BoundaryConditions: []
    VacuumPermittivity: []
                  Mesh: []
```

Create an axisymmetric model for magnetostatic analysis. An axisymmetric model simplifies a 3-D problem to a 2-D problem using symmetry around the axis of rotation.

```
emagMA = createpde("electromagnetic","magnetostatic-axisymmetric")
```

```
emagMA = 
  ElectromagneticModel with properties:

          AnalysisType: "magnetostatic-axisymmetric"
              Geometry: []
    MaterialProperties: []
               Sources: []
    BoundaryConditions: []
    VacuumPermeability: []
                  Mesh: []
```

Create a model for harmonic analysis.

```
emagH = createpde("electromagnetic","harmonic")
```

```
emagH = 
  ElectromagneticModel with properties:

          AnalysisType: "harmonic"
              Geometry: []
    MaterialProperties: []
               Sources: []
    BoundaryConditions: []
    VacuumPermittivity: []
    VacuumPermeability: []
                  Mesh: []
             FieldType: "electric"
```

Create a model for DC conduction analysis.

```
emagDC = createpde("electromagnetic","conduction")
```

```
emagDC = 
  ElectromagneticModel with properties:

          AnalysisType: "conduction"
              Geometry: []
    MaterialProperties: []
    BoundaryConditions: []
```

```
                    Mesh: []
```

Version History

Introduced in R2021a

DC conduction analysis and permanent magnets

The programmatic workflow for a DC conduction analysis enables you to analyze stationary current distribution in conductors due to applied voltage.

You can also specify magnetization using the `electromagneticSource` function to account for materials generating their own magnetic fields in a magnetostatic analysis workflow.

Harmonic analysis

The programmatic workflow for a harmonic electromagnetic analysis enables you to set up, solve, and analyze time-harmonic Maxwell's equations (the Helmholtz equation).

Electrostatic and magnetostatic analysis for 3-D models

The programmatic workflow for electrostatic and magnetostatic analyses now enables you to set up, solve, and analyze 3-D problems in addition to previously supported 2-D problems.

See Also

`electromagneticProperties` | `electromagneticSource` | `electromagneticBC` | `solve`

electromagneticBC

Package: `pde`

Apply boundary conditions to electromagnetic model

Syntax

```
electromagneticBC(emagmodel,RegionType,RegionID,"Voltage",V)
electromagneticBC(emagmodel,RegionType,RegionID,"MagneticPotential",A)
electromagneticBC(emagmodel,RegionType,RegionID,"SurfaceCurrentDensity",K)
```

```
electromagneticBC(emagmodel,RegionType,RegionID,"ElectricField",E)
electromagneticBC(emagmodel,RegionType,RegionID,"MagneticField",H)
electromagneticBC(emagmodel,RegionType,
RegionID,"FarField","absorbing","Thickness",h)
electromagneticBC(emagmodel,RegionType,
RegionID,"FarField","absorbing","Thickness",h,"Exponent",e,"Scaling",s)
```

```
electromagneticBC( ___ ,"Vectorized","on")
emagBC = electromagneticBC( ___ )
```

Description

`electromagneticBC(emagmodel,RegionType,RegionID,"Voltage",V)` adds a voltage boundary condition to `emagmodel`. The boundary condition applies to regions of type `RegionType` with ID numbers in `RegionID`. The solver uses a voltage boundary condition for an electrostatic analysis.

`electromagneticBC(emagmodel,RegionType,RegionID,"MagneticPotential",A)` adds a magnetic potential boundary condition to `emagmodel`. The solver uses a magnetic potential boundary condition for a magnetostatic analysis.

`electromagneticBC(emagmodel,RegionType,RegionID,"SurfaceCurrentDensity",K)` adds a surface current density boundary condition to `emagmodel`. The solver uses a surface current density boundary condition for a DC conduction analysis.

`electromagneticBC(emagmodel,RegionType,RegionID,"ElectricField",E)` adds an electric field boundary condition to `emagmodel`. The solver uses an electric field boundary condition for a harmonic analysis with the electric field type.

`electromagneticBC(emagmodel,RegionType,RegionID,"MagneticField",H)` adds a magnetic field boundary condition to `emagmodel`. The solver uses a magnetic field boundary condition for a harmonic analysis with the magnetic field type.

`electromagneticBC(emagmodel,RegionType,RegionID,"FarField","absorbing","Thickness",h)` adds an absorbing boundary condition to `emagmodel` and specifies the thickness of the absorbing region. The solver uses an absorbing boundary condition for a harmonic analysis.

`electromagneticBC(emagmodel,RegionType,RegionID,"FarField","absorbing","Thickness",h,"Exponent",e,"Scaling",s)` specifies the rate of attenuation of the waves entering the absorbing region. You can specify `e`, `s`, or both.

`electromagneticBC(___,"Vectorized","on")` uses vectorized function evaluation when you pass a function handle as an argument. If your function handle computes in a vectorized fashion, then using this argument saves time. For details on this evaluation, see “More About” on page 5-193 and “Vectorization”.

Use this syntax with any of the input argument combinations in the previous syntaxes.

`emagBC = electromagneticBC(___)` returns the electromagnetic boundary condition object.

Examples

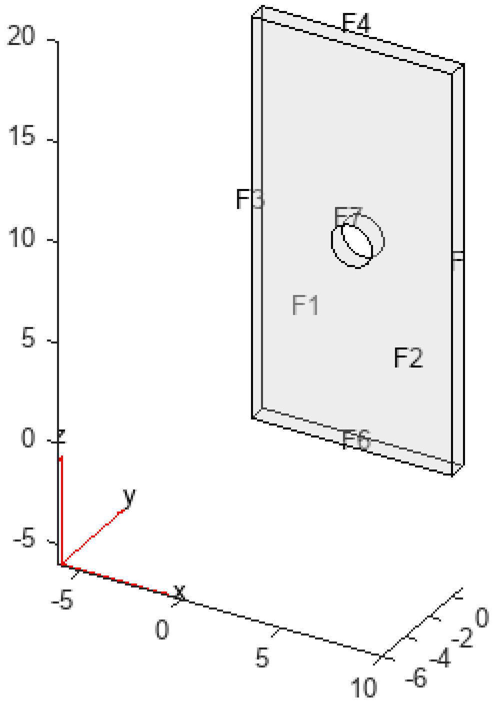
Specify Voltage on Boundaries

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Apply the voltage boundary condition on the side faces of the geometry.

```
bc1 = electromagneticBC(emagmodel,"Voltage",0,"Face",3:6)
```

```
bc1 = 
  ElectromagneticBCAssignment with properties:

      RegionID: [3 4 5 6]
    RegionType: 'Face'
    Vectorized: 'off'
       Voltage: 0
```

Apply the voltage boundary condition on the face bordering the hole.

```
bc2 = electromagneticBC(emagmodel,"Voltage",1000,"Face",7)
```

```
bc2 = 
  ElectromagneticBCAssignment with properties:

      RegionID: 7
    RegionType: 'Face'
    Vectorized: 'off'
       Voltage: 1000
```

Specify Magnetic Potential on Boundary

Apply a magnetic potential boundary condition on the boundary of a circle.

```
emagmodel = createpde("electromagnetic","magnetostatic");
geometryFromEdges(emagmodel,@circleg);
electromagneticBC(emagmodel,"Edge",1,"MagneticPotential",0)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

             RegionID: 1
           RegionType: 'Edge'
           Vectorized: 'off'
    MagneticPotential: 0
```

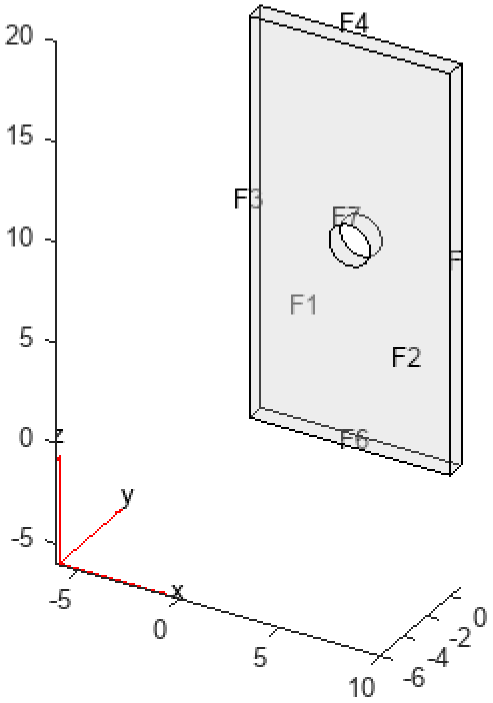
Specify Surface Current Density on Boundaries

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Apply the surface current density boundary condition on the front and back faces of the geometry.

```
electromagneticBC(emagmodel,"SurfaceCurrentDensity",100,"Face",[1 2])
```

```
ans = 
  ElectromagneticBCAssignment with properties:

               RegionID: [1 2]
             RegionType: 'Face'
             Vectorized: 'off'
                Voltage: []
    SurfaceCurrentDensity: 100
```

Specify Nonconstant Voltage on Boundary

Use a function handle to specify a boundary condition that depends on the coordinates.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Create a unit circle geometry and include it in the model.

```
geometryFromEdges(emagmodel,@circleg);
```

Specify the voltage on the boundary using the function $V(x, y) = x^2$.

```
bc = @(location,~)location.x.^2;
electromagneticBC(emagmodel,"Edge",1:emagmodel.Geometry.NumEdges, ...
                            "Voltage",bc)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

      RegionID: [1 2 3 4]
    RegionType: 'Edge'
    Vectorized: 'off'
       Voltage: @(location,~)location.x.^2
```

Specify Boundary Conditions for Harmonic Analysis

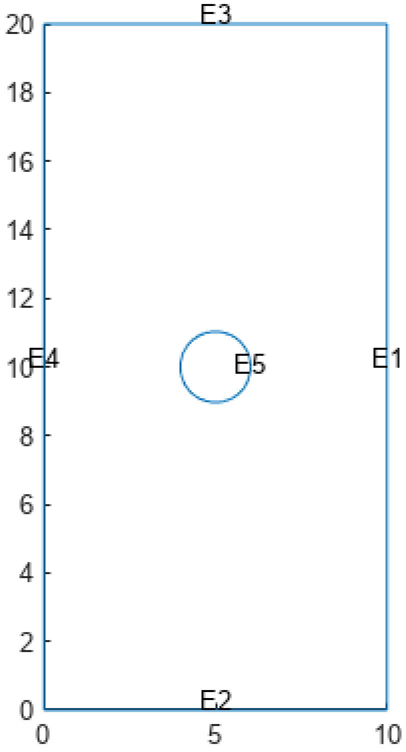
Specify an absorbing boundary condition and an electric field on a boundary for harmonic analysis.

Create an electromagnetic model for harmonic analysis.

```
emagmodel = createpde("electromagnetic","harmonic");
```

Import and plot a 2-D geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(gm,"EdgeLabels","on")
```



Specify the electric field on the circular edge.

```
electromagneticBC(emagmodel,"Edge",5,"ElectricField",[10 0])
```

```
ans = 
  ElectromagneticBCAssignment with properties:

         RegionID: 5
       RegionType: 'Edge'
       Vectorized: 'off'
    ElectricField: [10 0]
    MagneticField: []

   Far Field
         FarField: []
        Thickness: []
          Scaling: 5
         Exponent: 4
```

Specify absorbing regions with the thickness 2 on the edges of the rectangle. Use the default attenuation rate for the absorbing regions.

```
electromagneticBC(emagmodel,"Edge",1:4, ...
                            "FarField","absorbing", ...
                            "Thickness",2)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

         RegionID: [1 2 3 4]
       RegionType: 'Edge'
       Vectorized: 'off'
    ElectricField: [2x1 double]
    MagneticField: [2x1 double]

   Far Field
         FarField: "absorbing"
        Thickness: 2
          Scaling: 5
         Exponent: 4
```

Now specify the attenuation rate for the absorbing regions by using the `Exponent` and `Scaling` arguments.

```
electromagneticBC(emagmodel,"Edge",1:4, ...
                            "FarField","absorbing", ...
                            "Thickness",2, ...
                            "Exponent",3, ...
                            "Scaling",100)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

         RegionID: [1 2 3 4]
       RegionType: 'Edge'
       Vectorized: 'off'
    ElectricField: [2x1 double]
    MagneticField: [2x1 double]

   Far Field
         FarField: "absorbing"
        Thickness: 2
          Scaling: 100
         Exponent: 3
```

Specify Magnetic Field on Boundary

Apply a magnetic field on the boundary of a square for harmonic analysis.

Create an electromagnetic model for harmonic analysis.

```
emagmodel = createpde("electromagnetic","harmonic")
```

```
emagmodel = 
  ElectromagneticModel with properties:

          AnalysisType: "harmonic"
              Geometry: []
    MaterialProperties: []
               Sources: []
```

```
    BoundaryConditions: []
    VacuumPermittivity: []
    VacuumPermeability: []
                  Mesh: []
             FieldType: "electric"
```

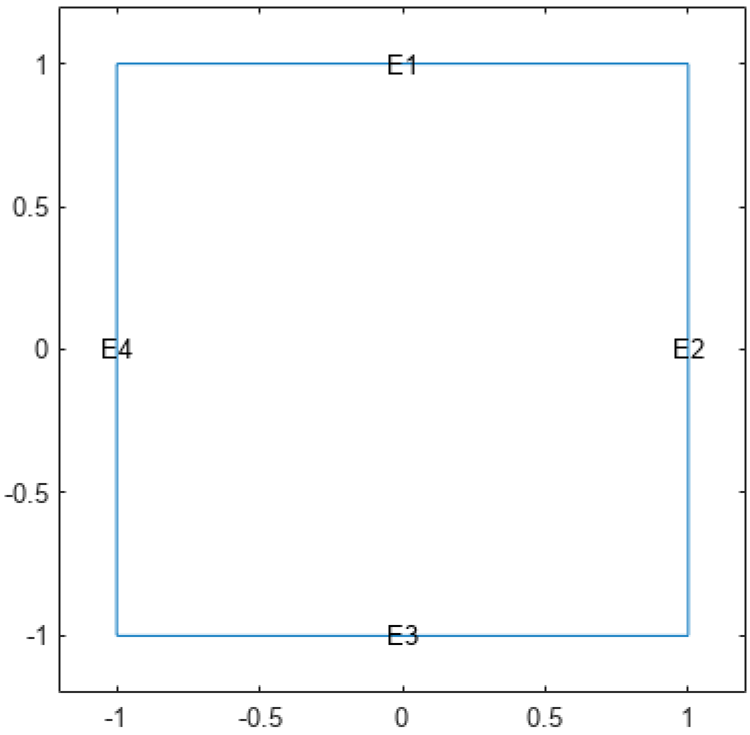
Change the field type from the default `electric` to `magnetic`.

```
emagmodel.FieldType = "magnetic"
```

```
emagmodel = 
  ElectromagneticModel with properties:

          AnalysisType: "harmonic"
              Geometry: []
    MaterialProperties: []
               Sources: []
    BoundaryConditions: []
    VacuumPermittivity: []
    VacuumPermeability: []
                  Mesh: []
             FieldType: "magnetic"
```

Include a square geometry in the model. Plot the geometry with the edge labels.

```
geometryFromEdges(emagmodel,@squareg);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-1.2 1.2])
ylim([-1.2 1.2])
```



Specify a magnetic field on the edges of the square.

```
electromagneticBC(emagmodel,"Edge",1:4,"MagneticField",[10 10])
```

```
ans = 
  ElectromagneticBCAssignment with properties:

         RegionID: [1 2 3 4]
       RegionType: 'Edge'
       Vectorized: 'off'
    ElectricField: []
    MagneticField: [10 10]

   Far Field
         FarField: []
        Thickness: []
          Scaling: 5
         Exponent: 4
```

Input Arguments

`emagmodel` — Electromagnetic model
`ElectromagneticModel` object

Electromagnetic model, specified as an `ElectromagneticModel` object. The model contains a geometry, a mesh, electromagnetic properties of the material, the electromagnetic sources, and the boundary conditions.

`RegionType` — Geometric region type
`"Edge"` for a 2-D model | `"Face"` for a 3-D model

Geometric region type, specified as `"Edge"` for a 2-D model or `"Face"` for a 3-D model.

Example: `electromagneticBC(emagmodel,"Edge",1,"Voltage",100)`

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. Find the edge or face IDs by using `pdegplot` with the `EdgeLabels` or `FaceLabels` name-value argument set to `"on"`.

Data Types: `double`

`V` — Voltage
real number | function handle

Voltage, specified as a real number or a function handle. Use a function handle to specify a voltage that depends on the coordinates. For details, see “More About” on page 5-193.

The solver uses a voltage boundary condition for an electrostatic analysis.

Data Types: `double` | `function_handle`

`A` — Magnetic potential
real number | column vector | function handle

Magnetic potential, specified as a real number, a column vector of three elements for a 3-D model, or a function handle. Use a function handle to specify a magnetic potential that depends on the coordinates. For details, see “More About” on page 5-193.

The solver uses a magnetic potential boundary condition for a magnetostatic analysis.

Data Types: `double` | `function_handle`

`E` — Electric field
column vector | function handle

Electric field, specified as a column vector of two elements for a 2-D model, a vector of three elements for a 3-D model, or a function handle. Use a function handle to specify an electric field that depends on the coordinates. For details, see “More About” on page 5-193.

The solver uses an electric field boundary condition for a harmonic analysis with the electric field type.

Data Types: `double` | `function_handle`

`K` — Surface current density
real number | function handle

Surface current density in the direction normal to the boundary, specified as a real number or a function handle. The solver uses a surface current density boundary condition for a DC conduction analysis. Use a function handle to specify a surface current density that depends on the coordinates. For details, see “More About” on page 5-193.

Data Types: `double`

`H` — Magnetic field
column vector | function handle

Magnetic field, specified as a column vector of two elements for a 2-D model, a column vector of three elements for a 3-D model, or a function handle. Use a function handle to specify a magnetic field that depends on the coordinates. For details, see “More About” on page 5-193.

The solver uses a magnetic field boundary condition for a harmonic analysis with the magnetic field type.

Data Types: `double` | `function_handle`

`h` — Width of far field absorbing region
nonnegative number

Width of the far field absorbing region, specified as a nonnegative number. The solver uses an absorbing boundary condition for a harmonic analysis.

Data Types: `double`

`e` — Exponent defining attenuation rate
4 (default) | nonnegative number

Exponent defining the attenuation rate of the waves entering the absorbing region, specified as a nonnegative number. The solver uses an absorbing boundary condition for a harmonic analysis.

Data Types: `double`

`s` — Scaling parameter defining attenuation rate
5 (default) | nonnegative number

Scaling parameter defining the attenuation rate of the waves entering the absorbing region, specified as a nonnegative number. The solver uses an absorbing boundary condition for a harmonic analysis.

Data Types: `double`

Output Arguments

`emagBC` — Handle to electromagnetic boundary condition
`ElectromagneticBCAssignment` object

Handle to the electromagnetic boundary condition, returned as an `ElectromagneticBCAssignment` object. For more information, see ElectromagneticBCAssignment Properties.

More About

Specifying Nonconstant Parameters of Electromagnetic Model

In Partial Differential Equation Toolbox, use a function handle to specify these electromagnetic parameters when they depend on the coordinates and, for a harmonic analysis, on the frequency:

- Relative permittivity of the material
- Relative permeability of the material
- Conductivity of the material
- Charge density as source (can depend on space only)
- Current density as source (can depend on space only)
- Magnetization (can depend on space only)
- Voltage on the boundary (can depend on space only)
- Magnetic potential on the boundary (can depend on space only)
- Electric field on the boundary (can depend on space only)
- Magnetic field on the boundary (can depend on space only)
- Surface current density on the boundary (can depend on space only)

For example, use function handles to specify the relative permittivity, charge density, and voltage on the boundary for `emagmodel`.

```
electromagneticProperties(emagmodel, ...
                          "RelativePermittivity", ...
                          @myfunPermittivity)
electromagneticSource(emagmodel, ...
                      "ChargeDensity",@myfunCharge, ...
                      "Face",2)
electromagneticBC(emagmodel, ...
                  "Voltage",@myfunBC, ...
                  "Edge",2)
```

The function must be of the form:

```
function emagVal = myfun(location,state)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names in place of `location` and `state`.

If you call `electromagneticBC` with `Vectorized` set to `"on"`, then `location` can contain several evaluation points. If you do not set `Vectorized` or set `Vectorized` to `"off"`, then the solver passes just one evaluation point in each call.

- `location` — A structure array containing these fields:
  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes this data in the `location` structure:

  - `location.nx` — The *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — The *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, the *z*-component of the normal vector at the evaluation point or points

  - `location.nr` — For an axisymmetric geometry, the *r*-component of the normal vector at the evaluation point or points
- `state` — A structure array containing this field for a harmonic electromagnetic problem:
  - `state.frequency` - Frequency at evaluation points

Relative permittivity, relative permeability, and conductivity get this data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `state.frequency` for a harmonic analysis
- Subdomain ID

Charge density, current density, magnetization, surface current density on the boundary, and electric or magnetic field on the boundary get this data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

Voltage or magnetic potential on the boundary get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`

When you solve an electrostatic, magnetostatic, or DC conduction problem, the output returned by the function handle must be of the following size. Here, `Np = numel(location.x)` is the number of points.

- `1`-by-`Np` if a function specifies the nonconstant relative permittivity, relative permeability, and charge density. For the charge density, the output can also be `Np`-by-`1`.
- `1`-by-`Np` for a 2-D model and `3`-by-`Np` for a 3-D model if a function specifies the nonconstant current density and magnetic potential on the boundary. For the current density, the output can also be `Np`-by-`1` or `Np`-by-`3`.
- `2`-by-`Np` for a 2-D model and `3`-by-`Np` for a 3-D model if a function specifies the nonconstant magnetization or surface current density on the boundary.

When you solve a harmonic problem, the output returned by the function handle must be of the following size. Here, `Np = numel(location.x)` is the number of points.

- `1`-by-`Np` if a function specifies the nonconstant relative permittivity, relative permeability, and conductivity.
- `2`-by-`Np` for a 2-D problem and `3`-by-`Np` for a 3-D problem if a function specifies the nonconstant electric or magnetic field.
- `2`-by-`Np` or `Np`-by-`2` for a 2-D problem and `3`-by-`Np` or `Np`-by-`3` for a 3-D problem if a function specifies the nonconstant current density and the field type is electric.
- `1`-by-`Np` or `Np`-by-`1` for a 2-D problem and `3`-by-`Np` or `Np`-by-`3` for a 3-D problem if a function specifies the nonconstant current density and the field type is magnetic.

If relative permittivity, relative permeability, or conductivity for a harmonic analysis depends on the frequency, ensure that your function returns a matrix of `NaN` values of the correct size when `state.frequency` is `NaN`. Solvers check whether a problem is nonlinear or time dependent by passing `NaN` state values and looking for returned `NaN` values.

Additional Arguments in Functions for Nonconstant Electromagnetic Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
emagVal = @(location,state) myfunWithAdditionalArgs(location,arg1,arg2,...)
electromagneticBC(model,"Edge",3,"Voltage",emagVal)
```

Version History

Introduced in R2021a

Surface current density

Surface current density boundary condition can be specified for a DC conduction analysis.

Boundary conditions for harmonic analysis

Electric and magnetic field on boundaries, and absorbing boundary conditions can be specified for harmonic analysis.

Support for 3-D electrostatic and magnetostatic problems

Voltage and magnetic potential can be specified on faces for 3-D electrostatic and magnetostatic problems.

See Also

`ElectromagneticModel` | ElectromagneticBCAssignment Properties | `createpde` | `electromagneticSource` | `solve` | `assembleFEMatrices` | `electromagneticProperties`

ElectromagneticBCAssignment Properties

Electromagnetic boundary condition assignments

Description

An `ElectromagneticBCAssignment` object specifies the type of PDE boundary condition on a set of geometry boundaries. An `ElectromagneticModel` object contains an array of `ElectromagneticBCAssignment` objects in its `BoundaryConditions.BCAssignments` property.

Create boundary condition assignments for your electromagnetic model using the `electromagneticBC` function.

Properties

Properties

`ElectricField` — Electric field boundary condition
column vector | function handle

Electric field boundary condition, specified as a column vector of two elements for a 2-D model, a column vector of three elements for a 3-D model, or a function handle. Use a function handle to specify an electric field that depends on the coordinates.

The solver uses an electric field boundary condition for a harmonic analysis with the electric field type.

Data Types: `double` | `function_handle`

`MagneticField` — Magnetic field boundary condition
column vector | function handle

Magnetic field boundary condition, specified as a column vector of two elements for a 2-D model, a column vector of three elements for a 3-D model, or a function handle. Use a function handle to specify a magnetic field that depends on the coordinates.

The solver uses a magnetic field boundary condition for a harmonic analysis with the magnetic field type.

Data Types: `double` | `function_handle`

`MagneticPotential` — Magnetic potential boundary condition
real number | column vector | function handle

Magnetic potential boundary condition, specified as a real number for a 2-D model, a column vector of three elements for a 3-D model, or a function handle. Use a function handle to specify a magnetic potential that depends on the coordinates.

The solver uses a magnetic potential boundary condition for a magnetostatic analysis.

Data Types: `double` | `function_handle`

`SurfaceCurrentDensity` — Surface current density boundary condition

real number

Surface current density boundary condition, specified as a real number. The solver uses a surface current density boundary condition for a DC conduction analysis.

Data Types: `double`

`RegionID` — Region ID

vector of positive integers

Region ID, specified as a vector of positive integers. Find the edge or face IDs by using `pdegplot` with the `EdgeLabels` or `FaceLabels` name-value argument set to `"on"`.

Data Types: `double`

`RegionType` — Geometric region type

`"Edge"` for a 2-D model | `"Face"` for a 3-D model

Geometric region type, specified as `"Edge"` for a 2-D model or `"Face"` for a 3-D model.

Data Types: `char` | `string`

`Vectorized` — Vectorized function evaluation

`"off"` (default) | `"on"`

Vectorized function evaluation, specified as `"off"` or `"on"`. This property applies when you pass a function handle as an argument. To save time in the function handle evaluation, specify `"on"` if your function handle computes in a vectorized fashion. See “Vectorization”. For details on vectorized function evaluation, see “Nonconstant Boundary Conditions” on page 2-132.

Data Types: `char` | `string`

`Voltage` — Voltage boundary condition

real number | function handle

Voltage boundary condition, specified as a real number or a function handle. Use a function handle to specify a voltage that depends on the coordinates.

The solver uses a voltage boundary condition for an electrostatic analysis.

Data Types: `double` | `function_handle`

`FarField` — Absorbing boundary condition

`"absorbing"`

Absorbing boundary condition, specified as `"absorbing"`. The solver uses an absorbing boundary condition for a harmonic analysis.

Data Types: `char`

`Thickness` — Width of far field absorbing region

nonnegative number

Width of the far field absorbing region, specified as a nonnegative number. The solver uses an absorbing boundary condition for a harmonic analysis.

Data Types: `double`

`Scaling` — Scaling parameter defining attenuation rate
5 (default) | nonnegative number

Scaling parameter defining the attenuation rate of the waves entering the absorbing region, specified as a nonnegative number. The solver uses an absorbing boundary condition for a harmonic analysis.

Data Types: `double`

`Exponent` — Exponent defining attenuation rate
4 (default) | nonnegative number

Exponent defining the attenuation rate of the waves entering the absorbing region, specified as a nonnegative number. The solver uses an absorbing boundary condition for a harmonic analysis.

Data Types: `double`

Version History

Introduced in R2021a

See Also

`ElectromagneticModel` | `electromagneticBC`

electromagneticProperties

Package: pde

Assign properties of material for electromagnetic model

Syntax

```
electromagneticProperties(emagmodel,"RelativePermittivity",epsilon)
electromagneticProperties(emagmodel,"RelativePermeability",mu)
electromagneticProperties(emagmodel,"Conductivity",sigma)
electromagneticProperties(emagmodel,"RelativePermittivity",
epsilon,"RelativePermeability",mu,"Conductivity",sigma)
electromagneticProperties( ___ ,RegionType,RegionID)
mtl = electromagneticProperties( ___ )
```

Description

`electromagneticProperties(emagmodel,"RelativePermittivity",epsilon)` assigns the relative permittivity `epsilon` to the entire geometry. Specify the permittivity of vacuum using the electromagnetic model properties. The solver uses a relative permittivity for electrostatic and harmonic analyses.

For a nonconstant material, specify `epsilon` as a function handle.

`electromagneticProperties(emagmodel,"RelativePermeability",mu)` assigns the relative permeability to the entire geometry. Specify the permeability of vacuum using the electromagnetic model properties. The solver uses a relative permeability for magnetostatic and harmonic analyses.

For a nonconstant material, specify `mu` as a function handle.

`electromagneticProperties(emagmodel,"Conductivity",sigma)` assigns the conductivity to the entire geometry. The solver uses a conductivity for DC conduction and harmonic analyses.

For a nonconstant material, specify `sigma` as a function handle.

`electromagneticProperties(emagmodel,"RelativePermittivity", epsilon,"RelativePermeability",mu,"Conductivity",sigma)` assigns the relative permittivity, relative permeability, and conductivity to the entire geometry. Specify the permittivity and permeability of vacuum using the electromagnetic model properties. The solver requires all three parameters for a harmonic analysis.

For a nonconstant material, specify `epsilon`, `mu`, and `sigma` as function handles.

`electromagneticProperties( ___ ,RegionType,RegionID)` assigns the material properties to specified faces of a 2-D geometry or cells of a 3-D geometry. Use this syntax with any of the input argument combinations in the previous syntaxes.

`mtl = electromagneticProperties( ___ )` returns the material properties object.

Examples

Specify Relative Permittivity

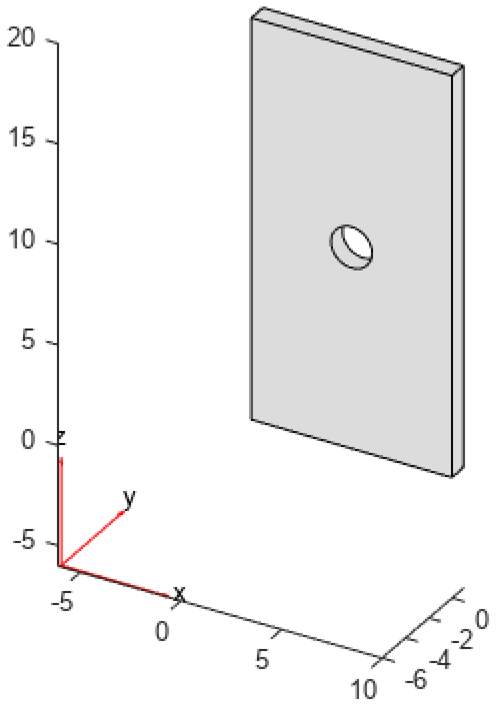
Specify relative permittivity for an electrostatic analysis.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot a geometry of a plate with a hole in its center.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm)
```



Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
mtl = electromagneticProperties(emagmodel,"RelativePermittivity",2.25)
```

```
mtl = 
  ElectromagneticMaterialAssignment with properties:

              RegionType: 'Cell'
```

```
               RegionID: 1
    RelativePermittivity: 2.2500
    RelativePermeability: []
            Conductivity: []
```

Specify Relative Permeability

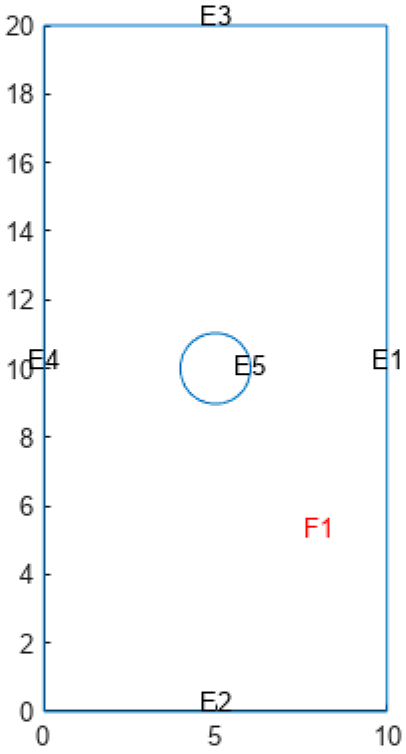
Specify relative permeability for a magnetostatic analysis.

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Import and plot a 2-D geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(gm,"EdgeLabels","on","FaceLabels","on")
```



Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the material.

```
mtl = electromagneticProperties(emagmodel,"RelativePermeability",5000)
```

```
mtl = 
  ElectromagneticMaterialAssignment with properties:

              RegionType: 'Face'
                RegionID: 1
    RelativePermittivity: []
    RelativePermeability: 5000
            Conductivity: []
```

Specify Conductivity

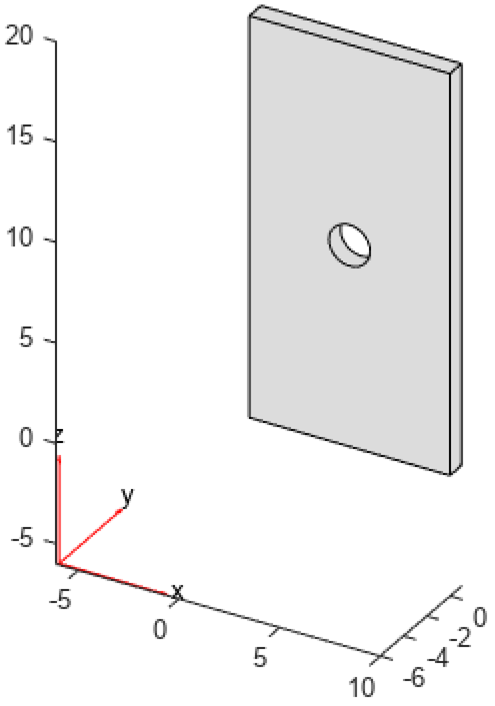
Specify a conductivity for DC conduction analysis.

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole in its center.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm)
```



Specify the conductivity of the material.

```
electromagneticProperties(emagmodel,"Conductivity",3.7e7)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

               RegionType: 'Cell'
                 RegionID: 1
    RelativePermittivity: []
    RelativePermeability: []
            Conductivity: 37000000
```

Specify Material Properties for Harmonic Analysis

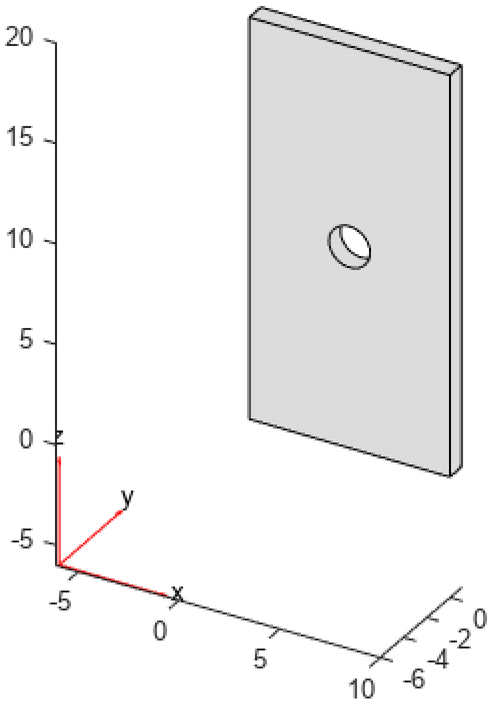
Specify relative permittivity, relative permeability, and conductivity of a material for harmonic analysis.

Create an electromagnetic model for harmonic analysis.

```
emagmodel = createpde("electromagnetic","harmonic");
```

Import and plot a geometry of a plate with a hole in its center.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm)
```



Specify the vacuum permittivity and permeability values in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity, relative permeability, and conductivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",2.25, ...
                                    "RelativePermeability",5000, ...
                                    "Conductivity",1)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

              RegionType: 'Cell'
                RegionID: 1
    RelativePermittivity: 2.2500
    RelativePermeability: 5000
            Conductivity: 1
```

Specify Relative Permittivity for Each Face

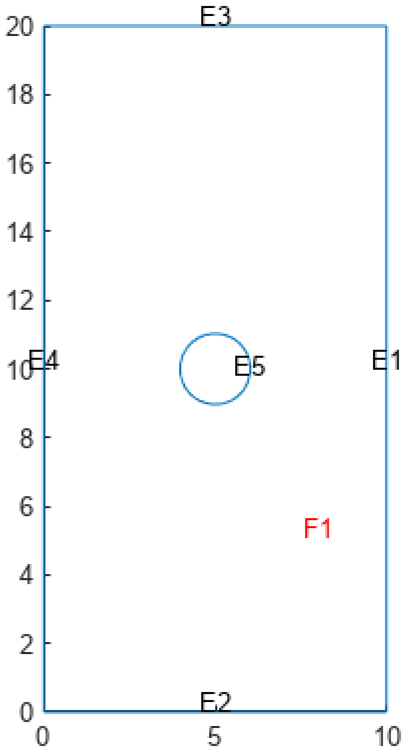
Specify relative permittivity for individual faces in an electrostatic model.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

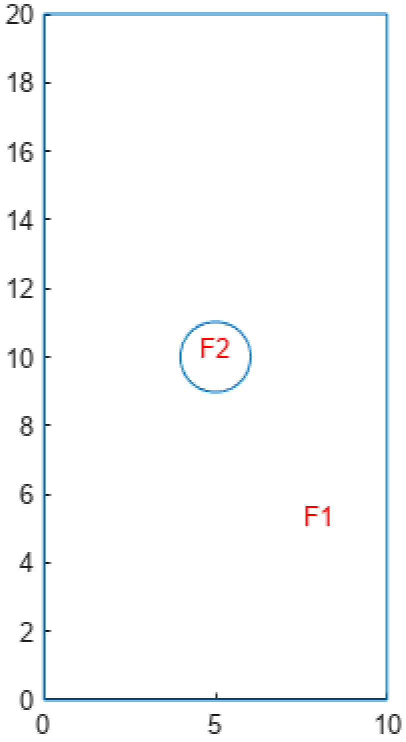
Create a 2-D geometry with two faces. First, import and plot a 2-D geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(gm,"EdgeLabels","on","FaceLabels","on")
```



Then, fill the hole by adding a face and plot the resulting geometry.

```
gm = addFace(gm,5);
pdegplot(gm,"FaceLabels","on")
```



Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify relative permittivities separately for faces 1 and 2.

```
electromagneticProperties(emagmodel,"RelativePermittivity",2.25, ...
                                    "Face",1)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

               RegionType: 'Face'
                 RegionID: 1
    RelativePermittivity: 2.2500
    RelativePermeability: []
             Conductivity: []
```

```
electromagneticProperties(emagmodel,"RelativePermittivity",1, ...
                                    "Face",2)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

               RegionType: 'Face'
                 RegionID: 2
    RelativePermittivity: 1
```

```
    RelativePermeability: []
            Conductivity: []
```

Specify Nonconstant Relative Permittivity

Use a function handle to specify a relative permittivity that depends on the spatial coordinates.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Create a square geometry and include it in the model.

```
geometryFromEdges(emagmodel,@squareg);
```

Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material as a function of the x-coordinate, $\varepsilon = \sqrt{1 + x^2}$.

```
perm = @(location,~)sqrt(1 + location.x.^2);
electromagneticProperties(emagmodel,"RelativePermittivity",perm)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

              RegionType: 'Face'
                RegionID: 1
    RelativePermittivity: @(location,~)sqrt(1+location.x.^2)
    RelativePermeability: []
            Conductivity: []
```

Input Arguments

`emagmodel` — Electromagnetic model
`ElectromagneticModel` object

Electromagnetic model, specified as an `ElectromagneticModel` object. The model contains a geometry, a mesh, the electromagnetic properties of the material, the electromagnetic sources, and the boundary conditions.

`epsilon` — Relative permittivity
positive number | complex number | function handle

Relative permittivity, specified as a positive or complex number or a function handle.

- Use a positive number to specify a relative permittivity for an electrostatic analysis.
- Use a complex number to specify a relative permittivity for a harmonic electromagnetic analysis.
- Use a function handle to specify a relative permittivity that depends on the coordinates and, for a harmonic analysis, on the frequency.

For details, see “More About” on page 5-210.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`mu` — Relative permeability

positive number | complex number | function handle

Relative permeability, specified as a positive or complex number or a function handle.

- Use a positive number to specify a relative permeability for a magnetostatic analysis.
- Use a complex number to specify a relative permeability for a harmonic electromagnetic analysis.
- Use a function handle to specify a relative permeability that depends on the coordinates and, for a harmonic analysis, on the frequency.

For details, see “More About” on page 5-210.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`sigma` — Conductivity

nonnegative number | function handle

Conductivity, specified as a nonnegative number or a function handle. Use a function handle to specify a conductivity that depends on the coordinates and, for a harmonic analysis, on the frequency. For details, see “More About” on page 5-210.

Data Types: `double` | `function_handle`

`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 geometry or `"Cell"` for a 3-D geometry.

Data Types: `char` | `string`

`RegionID` — Region ID

vector of positive integers

Region ID, specified as a vector of positive integers. Find the face or cell IDs by using `pdegplot` with the `"FaceLabels"` or `"CellLabels"` name-value argument set to `"on"`.

Example:
`electromagneticProperties(emagmodel,"RelativePermeability",5000,"Face",1:3)`

Data Types: `double`

Output Arguments

`mtl` — Handle to material properties

`ElectromagneticMaterialAssignment` object

Handle to material properties, returned as an `ElectromagneticMaterialAssignment` object. For more information, see ElectromagneticMaterialAssignment Properties.

`mtl` associates material properties with the geometric faces.

More About

Specifying Nonconstant Parameters of Electromagnetic Model

In Partial Differential Equation Toolbox, use a function handle to specify these electromagnetic parameters when they depend on the coordinates and, for a harmonic analysis, on the frequency:

- Relative permittivity of the material
- Relative permeability of the material
- Conductivity of the material
- Charge density as source (can depend on space only)
- Current density as source (can depend on space only)
- Magnetization (can depend on space only)
- Voltage on the boundary (can depend on space only)
- Magnetic potential on the boundary (can depend on space only)
- Electric field on the boundary (can depend on space only)
- Magnetic field on the boundary (can depend on space only)
- Surface current density on the boundary (can depend on space only)

For example, use function handles to specify the relative permittivity, charge density, and voltage on the boundary for `emagmodel`.

```
electromagneticProperties(emagmodel, ...
                          "RelativePermittivity", ...
                          @myfunPermittivity)
electromagneticSource(emagmodel, ...
                      "ChargeDensity",@myfunCharge, ...
                      "Face",2)
electromagneticBC(emagmodel, ...
                  "Voltage",@myfunBC, ...
                  "Edge",2)
```

The function must be of the form:

```
function emagVal = myfun(location,state)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names in place of `location` and `state`.

If you call `electromagneticBC` with `Vectorized` set to `"on"`, then `location` can contain several evaluation points. If you do not set `Vectorized` or set `Vectorized` to `"off"`, then the solver passes just one evaluation point in each call.

- `location` — A structure array containing these fields:
  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes this data in the `location` structure:

  - `location.nx` — The *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — The *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, the *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, the *r*-component of the normal vector at the evaluation point or points
- `state` — A structure array containing this field for a harmonic electromagnetic problem:
  - `state.frequency` - Frequency at evaluation points

Relative permittivity, relative permeability, and conductivity get this data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `state.frequency` for a harmonic analysis
- Subdomain ID

Charge density, current density, magnetization, surface current density on the boundary, and electric or magnetic field on the boundary get this data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

Voltage or magnetic potential on the boundary get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`

When you solve an electrostatic, magnetostatic, or DC conduction problem, the output returned by the function handle must be of the following size. Here, `Np = numel(location.x)` is the number of points.

- `1`-by-`Np` if a function specifies the nonconstant relative permittivity, relative permeability, and charge density. For the charge density, the output can also be `Np`-by-`1`.
- `1`-by-`Np` for a 2-D model and `3`-by-`Np` for a 3-D model if a function specifies the nonconstant current density and magnetic potential on the boundary. For the current density, the output can also be `Np`-by-`1` or `Np`-by-`3`.
- `2`-by-`Np` for a 2-D model and `3`-by-`Np` for a 3-D model if a function specifies the nonconstant magnetization or surface current density on the boundary.

When you solve a harmonic problem, the output returned by the function handle must be of the following size. Here, `Np = numel(location.x)` is the number of points.

- `1`-by-`Np` if a function specifies the nonconstant relative permittivity, relative permeability, and conductivity.
- `2`-by-`Np` for a 2-D problem and `3`-by-`Np` for a 3-D problem if a function specifies the nonconstant electric or magnetic field.
- `2`-by-`Np` or `Np`-by-`2` for a 2-D problem and `3`-by-`Np` or `Np`-by-`3` for a 3-D problem if a function specifies the nonconstant current density and the field type is electric.

- `1`-by-`Np` or `Np`-by-`1` for a 2-D problem and `3`-by-`Np` or `Np`-by-`3` for a 3-D problem if a function specifies the nonconstant current density and the field type is magnetic.

If relative permittivity, relative permeability, or conductivity for a harmonic analysis depends on the frequency, ensure that your function returns a matrix of `NaN` values of the correct size when `state.frequency` is `NaN`. Solvers check whether a problem is nonlinear or time dependent by passing `NaN` state values and looking for returned `NaN` values.

Additional Arguments in Functions for Nonconstant Electromagnetic Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
emagVal = @(location,state) myfunWithAdditionalArgs(location,arg1,arg2,...)
electromagneticBC(model,"Edge",3,"Voltage",emagVal)
```

Version History

Introduced in R2021a

Material properties for harmonic analysis

Material properties now include conductivity. For harmonic analysis, relative permittivity, relative permeability, and conductivity can depend on frequency. Also for harmonic analysis, relative permittivity and permeability can be a complex value.

See Also

`ElectromagneticModel` | ElectromagneticMaterialAssignment Properties | `createpde` | `electromagneticSource` | `electromagneticBC` | `solve` | `assembleFEMatrices`

ElectromagneticMaterialAssignment Properties

Electromagnetic material properties assignments

Description

An `ElectromagneticMaterialAssignment` object describes the material properties of an electromagnetic model. An `ElectromagneticModel` object contains a vector of `ElectromagneticMaterialAssignment` objects in its `MaterialProperties.MaterialAssignments` property.

Create material property assignments for your electromagnetic model using the `electromagneticProperties` function.

Properties

Properties

`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 geometry or `"Cell"` for a 3-D geometry.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. Find the face or cell IDs by using `pdegplot` with the `"FaceLabels"` or `"CellLabels"` name-value argument set to `"on"`.

Data Types: `double`

`RelativePermittivity` — Relative permittivity of material
positive number | complex number | function handle

Relative permittivity of the material, specified as a positive or complex number or a function handle.

- A positive number specifies a relative permittivity for an electrostatic analysis.
- A complex number specifies a relative permittivity for a harmonic electromagnetic analysis.
- A function handle specifies a relative permittivity that depends on the coordinates or, for a harmonic analysis, on the frequency.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`RelativePermeability` — Relative permeability of material
positive number | complex number | function handle

Relative permeability of the material, specified as a positive or complex number or a function handle.

- A positive number specifies a relative permeability for a magnetostatic analysis.
- A complex number specifies a relative permeability for a harmonic electromagnetic analysis.
- A function handle specifies a relative permeability that depends on the coordinates or, for a harmonic analysis, on the frequency.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`Conductivity` — Conductivity of material
nonnegative number | function handle

Conductivity of the material, specified as a nonnegative number or a function handle. A function handle specifies a conductivity that depends on the coordinates or, for a harmonic analysis, on the frequency.

Data Types: `double` | `function_handle`

Tips

- When there are multiple assignments to the same face, the toolbox uses the last applied setting.
- To avoid assigning material properties to the wrong region, check that you are using the correct face IDs and cell IDs by plotting and visually inspecting the geometry. Use `pdegplot` with the `"FaceLabels"` or `"CellLabels"` name-value argument set to `"on"`.

Version History

Introduced in R2021a

See Also

`ElectromagneticModel` | `electromagneticProperties` | `pdegplot`

electromagneticSource

Package: pde

Specify current density, charge density, and magnetization for electromagnetic model

Syntax

```
electromagneticSource(emagmodel,"ChargeDensity",rho)
electromagneticSource(emagmodel,"CurrentDensity",J)
electromagneticSource(emagmodel,"Magnetization",M)
electromagneticSource( ___ ,RegionType,RegionID)
emagSource = electromagneticSource( ___ )
```

Description

`electromagneticSource(emagmodel,"ChargeDensity",rho)` specifies the charge density. The solver uses a charge density for an electrostatic analysis.

`electromagneticSource(emagmodel,"CurrentDensity",J)` specifies the current density. The solver uses a current density for magnetostatic or harmonic (time-harmonic) analyses.

For a 3-D magnetostatic analysis, you can specify current density by using the DC conduction results. See `ConductionResults`. The toolbox does not support conduction results as a source of current density for a 2-D magnetostatic analysis, in which case current density must be a scalar or a function handle returning a scalar that represents out-of-plane current.

`electromagneticSource(emagmodel,"Magnetization",M)` specifies the magnetization. The solver uses a magnetization to model permanent magnets in a magnetostatic workflow.

`electromagneticSource( ___ ,RegionType,RegionID)` specifies the charge or current density for the specified geometry region. Use this syntax with any of the input argument combinations in the previous syntaxes.

`emagSource = electromagneticSource( ___ )` returns the electromagnetic source object.

Examples

Specify Charge Density on Entire Geometry

Specify charge density on the entire geometry for an electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
importGeometry(emagmodel,"PlateHoleSolid.stl");
electromagneticSource(emagmodel,"ChargeDensity",10)
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

        RegionType: 'Cell'
          RegionID: 1
```

```
  ChargeDensity: 10
 CurrentDensity: []
  Magnetization: []
```

Specify Current Density on Entire Geometry

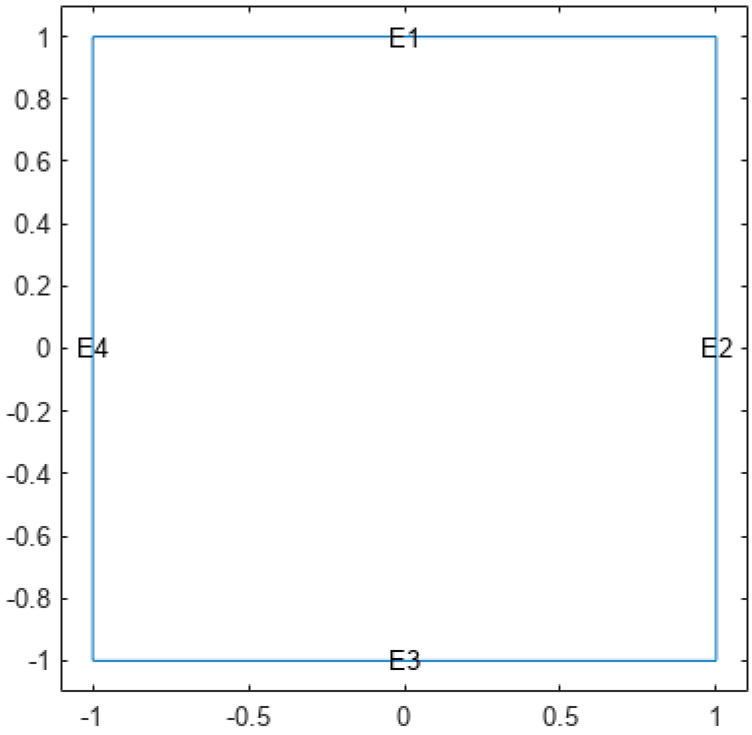
Specify current density on the entire geometry for harmonic analysis.

Create an electromagnetic model for harmonic analysis.

```
model = createpde("electromagnetic","harmonic");
```

Include a square geometry in the model. Plot the geometry with the edge labels.

```
geometryFromEdges(model,@squareg);
pdegplot(model,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Specify current density on the entire geometry. For a 2-D harmonic analysis model with the electric field type, the current density must be a column vector of two elements. When solving the model, the toolbox multiplies the specified current density value by `-i` and by frequency.

```
electromagneticSource(model,"CurrentDensity",[1;0])
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

        RegionType: 'Face'
          RegionID: 1
     ChargeDensity: []
    CurrentDensity: [2x1 double]
     Magnetization: []
```

Specify Charge Density on Each Face

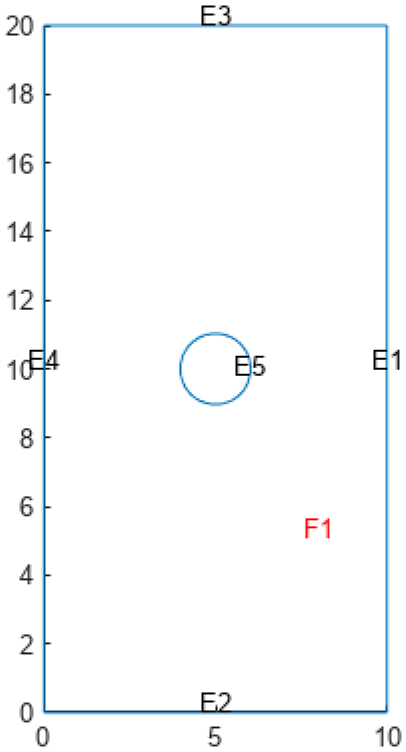
Specify charge density on individual faces in electrostatic analysis.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

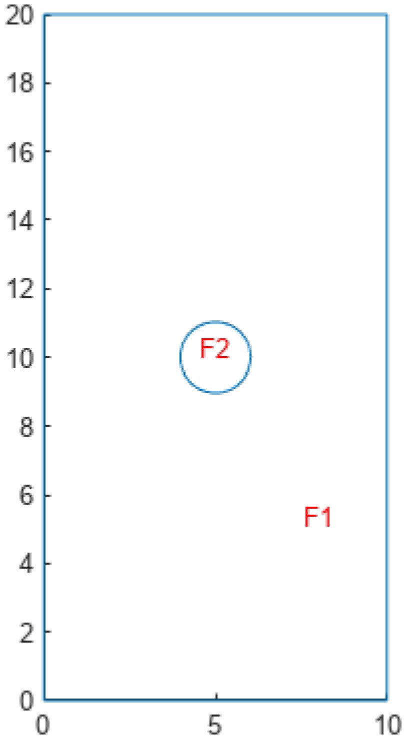
Create a 2-D geometry with two faces. First, import and plot a 2-D geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(gm,"EdgeLabels","on","FaceLabels","on")
```



Then, fill the hole by adding a face and plot the resulting geometry.

```
gm = addFace(gm,5);
pdegplot(gm,"FaceLabels","on")
```



Specify charge density values separately for faces 1 and 2.

```
sc1 = electromagneticSource(emagmodel,"Face",1,"ChargeDensity",0.3)
```

```
sc1 = 
  ElectromagneticSourceAssignment with properties:

         RegionType: 'Face'
           RegionID: 1
      ChargeDensity: 0.3000
     CurrentDensity: []
      Magnetization: []
```

```
sc2 = electromagneticSource(emagmodel,"Face",2,"ChargeDensity",0.28)
```

```
sc2 = 
  ElectromagneticSourceAssignment with properties:

         RegionType: 'Face'
           RegionID: 2
      ChargeDensity: 0.2800
     CurrentDensity: []
      Magnetization: []
```

Specify Nonconstant Charge Density

Use a function handle to specify a charge density that depends on the coordinates.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Create a unit circle geometry and include it in the model.

```
geometryFromEdges(emagmodel,@circleg);
```

Specify the charge density as a function of the x- and y-coordinates, $\rho = 0.3\sqrt{x^2 + y^2}$.

```
rho = @(location,~)0.3.*sqrt(location.x.^2 + location.y.^2);
electromagneticSource(emagmodel,"ChargeDensity",rho)
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

        RegionType: 'Face'
          RegionID: 1
     ChargeDensity: @(location,~)0.3.*sqrt(location.x.^2+location.y.^2)
    CurrentDensity: []
     Magnetization: []
```

Use DC Conduction Solution as Current Density

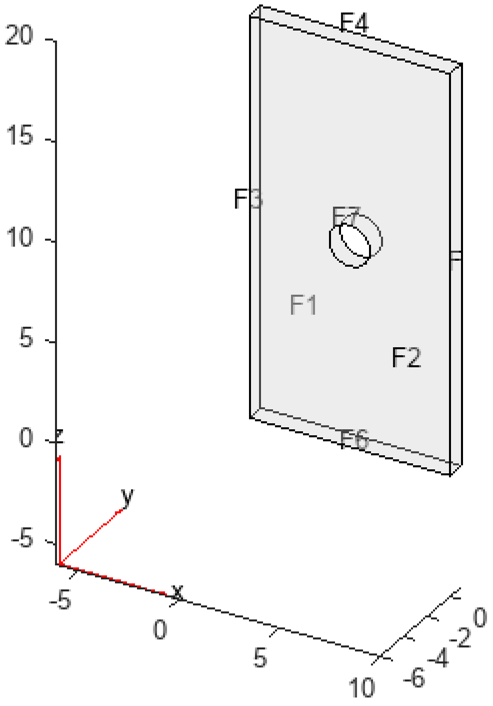
Use a solution obtained by performing DC conduction analysis to specify current density for a magnetostatic model.

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the conductivity of the material.

```
electromagneticProperties(emagmodel,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the left, right, and back faces of the plate.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",[1 3 5]);
```

Specify the surface current density on the front face of the geometry and on the face bordering the hole.

```
electromagneticBC(emagmodel,"SurfaceCurrentDensity",100,"Face",[2 7]);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

```
R = solve(emagmodel);
```

Change the analysis type of the model to magnetostatic.

```
emagmodel.AnalysisType = "magnetostatic";
```

This model already has a quadratic mesh that you generated for the DC conduction analysis. For a 3-D magnetostatic model, the mesh must be linear. Generate a new linear mesh. The `generateMesh` function creates a linear mesh by default if the model is 3-D and magnetostatic.

```
generateMesh(emagmodel);
```

Specify the current density for the entire geometry using the DC conduction solution.

```
electromagneticSource(emagmodel,"CurrentDensity",R)
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

        RegionType: 'Cell'
          RegionID: 1
     ChargeDensity: []
    CurrentDensity: [1x1 pde.ConductionResults]
     Magnetization: []
```

Specify Magnetization

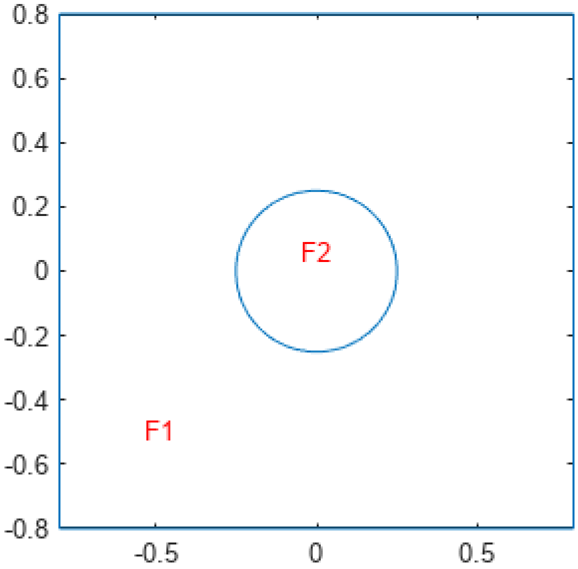
Specify magnetization on a face in a magnetostatic analysis.

Create a unit square geometry with a circle in its center. The circle represents a permanent magnet.

```
L = 0.8;
r = 0.25;
sq = [3 4 -L L L -L -L -L L L]';
circ = [1 0 0 r 0 0 0 0 0 0]';
gd = [sq,circ];
sf = "sq + circ";
ns = char('sq','circ');
ns = ns';
g = decsg(gd,sf,ns);
```

Plot the geometry with the face labels.

```
pdegplot(g,"FaceLabels","on")
```



Create a magnetostatic model and include the geometry in the model.

```
emagmodel = createpde("electromagnetic","magnetostatic");
geometryFromEdges(emagmodel,g);
```

Specify the magnetization magnitude.

```
M = 1;
```

To make the circle represent a permanent magnet, specify the uniform magnetization in the positive *x*-direction.

```
electromagneticSource(emagmodel,"Face",2,"Magnetization",M*[1;0])
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

         RegionType: 'Face'
           RegionID: 2
      ChargeDensity: []
     CurrentDensity: []
      Magnetization: [2×1 double]
```

Input Arguments

emagmodel — Electromagnetic model
`ElectromagneticModel` object

Electromagnetic model, specified as an `ElectromagneticModel` object. The model contains a geometry, a mesh, the electromagnetic properties of the material, the electromagnetic sources, and the boundary conditions.

`rho` — Charge density

real number | function handle

Charge density, specified as a real number or a function handle. Use a function handle to specify a charge density that depends on the coordinates. For details, see “More About” on page 5-224.

Data Types: `double` | `function_handle`

`J` — Current density

real number | column vector | function handle | `ConductionResults` object

Current density, specified as a real number, a column vector, a function handle, or a `ConductionResults` object. Use a function handle to specify a current density that depends on the coordinates.

For magnetostatic analysis, the current density must be

- A real number or a function handle for a 2-D model. The toolbox does not support conduction results as a source of current density for a 2-D magnetostatic analysis.
- A column vector of three elements, a `ConductionResults` object, or a function handle for a 3-D model.

For harmonic analysis with the electric field type, the toolbox multiplies the specified current density by `-i` and by frequency. The current density must be

- A column vector of two elements or a function handle for a 2-D model.
- A column vector of three elements or a function handle for a 3-D model.

For harmonic analysis with the magnetic field type, the toolbox uses the curl of the specified current density. The current density must be

- A scalar or a function handle for a 2-D model.
- A column vector of three elements or a function handle for a 3-D model.

For details, see “More About” on page 5-224.

Data Types: `double` | `function_handle`

`M` — Magnetization

column vector | function handle

Magnetization, specified as a column vector of two elements for a 2-D model, a column vector of three elements for a 3-D model, or a function handle. Use a function handle to specify a magnetization that depends on the coordinates. For details, see “More About” on page 5-224.

Data Types: `double` | `function_handle`

`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.

Data Types: `char` | `string`

`RegionID` — Region ID

vector of positive integers

Region ID, specified as a vector of positive integers. Find the face or cell IDs by using `pdegplot` with the `FaceLabels` or `CellLabels` name-value argument set to `"on"`.

Example: `electromagneticSource(emagmodel,"CurrentDensity",10,"Face",1:3)`

Data Types: `double`

Output Arguments

`emagSource` — Handle to electromagnetic source
`ElectromagneticSourceAssignment` object

Handle to the electromagnetic source, returned as an `ElectromagneticSourceAssignment` object. For more information, see ElectromagneticSourceAssignment Properties.

More About

Specifying Nonconstant Parameters of Electromagnetic Model

In Partial Differential Equation Toolbox, use a function handle to specify these electromagnetic parameters when they depend on the coordinates and, for a harmonic analysis, on the frequency:

- Relative permittivity of the material
- Relative permeability of the material
- Conductivity of the material
- Charge density as source (can depend on space only)
- Current density as source (can depend on space only)
- Magnetization (can depend on space only)
- Voltage on the boundary (can depend on space only)
- Magnetic potential on the boundary (can depend on space only)
- Electric field on the boundary (can depend on space only)
- Magnetic field on the boundary (can depend on space only)
- Surface current density on the boundary (can depend on space only)

For example, use function handles to specify the relative permittivity, charge density, and voltage on the boundary for `emagmodel`.

```
electromagneticProperties(emagmodel, ...
                          "RelativePermittivity", ...
                          @myfunPermittivity)
electromagneticSource(emagmodel, ...
                      "ChargeDensity",@myfunCharge, ...
                      "Face",2)
electromagneticBC(emagmodel, ...
                  "Voltage",@myfunBC, ...
                  "Edge",2)
```

The function must be of the form:

```
function emagVal = myfun(location,state)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names in place of `location` and `state`.

If you call `electromagneticBC` with `Vectorized` set to `"on"`, then `location` can contain several evaluation points. If you do not set `Vectorized` or set `Vectorized` to `"off"`, then the solver passes just one evaluation point in each call.

- `location` — A structure array containing these fields:
  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes this data in the `location` structure:

  - `location.nx` — The *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — The *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, the *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, the *r*-component of the normal vector at the evaluation point or points
- `state` — A structure array containing this field for a harmonic electromagnetic problem:
  - `state.frequency` - Frequency at evaluation points

Relative permittivity, relative permeability, and conductivity get this data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `state.frequency` for a harmonic analysis
- Subdomain ID

Charge density, current density, magnetization, surface current density on the boundary, and electric or magnetic field on the boundary get this data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

Voltage or magnetic potential on the boundary get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`

When you solve an electrostatic, magnetostatic, or DC conduction problem, the output returned by the function handle must be of the following size. Here, `Np = numel(location.x)` is the number of points.

- `1`-by-`Np` if a function specifies the nonconstant relative permittivity, relative permeability, and charge density. For the charge density, the output can also be `Np`-by-`1`.

- `1`-by-`Np` for a 2-D model and `3`-by-`Np` for a 3-D model if a function specifies the nonconstant current density and magnetic potential on the boundary. For the current density, the output can also be `Np`-by-`1` or `Np`-by-`3`.
- `2`-by-`Np` for a 2-D model and `3`-by-`Np` for a 3-D model if a function specifies the nonconstant magnetization or surface current density on the boundary.

When you solve a harmonic problem, the output returned by the function handle must be of the following size. Here, `Np = numel(location.x)` is the number of points.

- `1`-by-`Np` if a function specifies the nonconstant relative permittivity, relative permeability, and conductivity.
- `2`-by-`Np` for a 2-D problem and `3`-by-`Np` for a 3-D problem if a function specifies the nonconstant electric or magnetic field.
- `2`-by-`Np` or `Np`-by-`2` for a 2-D problem and `3`-by-`Np` or `Np`-by-`3` for a 3-D problem if a function specifies the nonconstant current density and the field type is electric.
- `1`-by-`Np` or `Np`-by-`1` for a 2-D problem and `3`-by-`Np` or `Np`-by-`3` for a 3-D problem if a function specifies the nonconstant current density and the field type is magnetic.

If relative permittivity, relative permeability, or conductivity for a harmonic analysis depends on the frequency, ensure that your function returns a matrix of `NaN` values of the correct size when `state.frequency` is `NaN`. Solvers check whether a problem is nonlinear or time dependent by passing `NaN` state values and looking for returned `NaN` values.

Additional Arguments in Functions for Nonconstant Electromagnetic Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
emagVal = @(location,state) myfunWithAdditionalArgs(location,arg1,arg2,...)
electromagneticBC(model,"Edge",3,"Voltage",emagVal)
```

Version History

Introduced in R2021a

Magnetization value for permanent magnets

Magnetization can be specified to account for materials generating their own magnetic fields in a magnetostatic analysis workflow.

See Also

`ElectromagneticModel` | `ConductionResults` | ElectromagneticSourceAssignment Properties | `createpde` | `electromagneticProperties` | `electromagneticBC` | `solve` | `assembleFEMatrices`

ElectromagneticSourceAssignment Properties

Electromagnetic source assignments

Description

An `ElectromagneticSourceAssignment` object describes the source of an electromagnetic model. An `ElectromagneticModel` object contains a vector of `ElectromagneticSourceAssignment` objects in its `Sources.SourceAssignments` property.

Create electromagnetic source assignments for your model using the `electromagneticSource` function.

Properties

Properties

`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.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. Find the face or cell IDs by using `pdegplot` with the `FaceLabels` or `CellLabels` name-value argument set to `"on"`.

Data Types: `double`

`ChargeDensity` — Charge density
real number | function handle

Charge density, specified as a real number or a function handle. Use a function handle to specify a charge density that depends on the coordinates.

Data Types: `double` | `function_handle`

`CurrentDensity` — Current density
real number | column vector | function handle | `ConductionResults` object

Current density, specified as a real number, a column vector, a function handle, or a `ConductionResults` object. Use a function handle to specify a current density that depends on the coordinates.

For magnetostatic analysis, the current density must be a real number for a 2-D model, a column vector of three elements for a 3-D model, a function handle for a 2-D or 3-D model, or a `ConductionResults` object for a 3-D model.

For harmonic analysis with the electric field type, the current density must be a column vector of two elements for a 2-D model, a column vector of three elements for a 3-D model, or a function handle for a 2-D or 3-D model. The toolbox multiplies the specified current density value by `-i` and by frequency.

For harmonic analysis with the magnetic field type, the current density must be a scalar for a 2-D model, a column vector of three elements for a 3-D model, or a function handle for a 2-D or 3-D model. The toolbox uses the curl of the specified current density.

Data Types: `double` | `function_handle`

`Magnetization` — Magnetization
column vector | function handle

Magnetization, specified as a column vector of two elements for a 2-D model, a column vector of three elements for a 3-D model, or a function handle. Use a function handle to specify a magnetization that depends on the coordinates.

Data Types: `double` | `function_handle`

Version History

Introduced in R2021a

See Also

`electromagneticSource` | `ElectromagneticModel`

ElectrostaticResults

Electrostatic solution and derived quantities

Description

An `ElectrostaticResults` object contains the electric potential, electric field, and electric flux density values in a form convenient for plotting and postprocessing.

The electric potential, electric field, and electric flux density are calculated at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. Electric potential values at the nodes appear in the `ElectricPotential` property. Electric field values at the nodes appear in the `ElectricField` property. Electric flux density at the nodes appear in the `ElectricFluxDensity` property.

To interpolate the electric potential, electric field, and electric flux density to a custom grid, such as the one specified by `meshgrid`, use the `interpolateElectricPotential`, `interpolateElectricField`, and `interpolateElectricFlux` functions.

Creation

Solve an electrostatic problem using the `solve` function. This function returns a solution as an `ElectrostaticResults` object.

Properties

`ElectricPotential` — Electric potential values at nodes
vector

This property is read-only.

Electric potential values at nodes, returned as a vector.

Data Types: `double`

`ElectricField` — Electric field values at nodes
`FEStruct` object

This property is read-only.

Electric field values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of the electric field at nodes.

`ElectricFluxDensity` — Electric flux density values at nodes
`FEStruct` object

This property is read-only.

Electric flux density values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of electric flux density at nodes.

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as an FEMesh object. For details, see FEMesh.

Object Functions

interpolateElectricPotential	Interpolate electric potential in electrostatic or DC conduction result at arbitrary spatial locations
interpolateElectricField	Interpolate electric field in electrostatic or DC conduction result at arbitrary spatial locations
interpolateElectricFlux	Interpolate electric flux density in electrostatic result at arbitrary spatial locations

Examples

Solution to 2-D Electrostatic Analysis Model

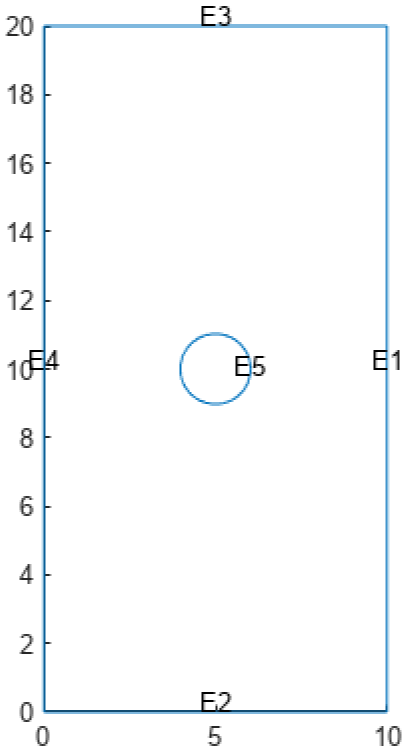
Solve an electromagnetic problem and find the electric potential and field distribution for a 2-D geometry representing a plate with a hole.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(emagmodel,"EdgeLabels","on")
```



Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Apply the voltage boundary conditions on the edges framing the rectangle and the circle.

```
electromagneticBC(emagmodel,"Voltage",0,"Edge",1:4);
electromagneticBC(emagmodel,"Voltage",1000,"Edge",5);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

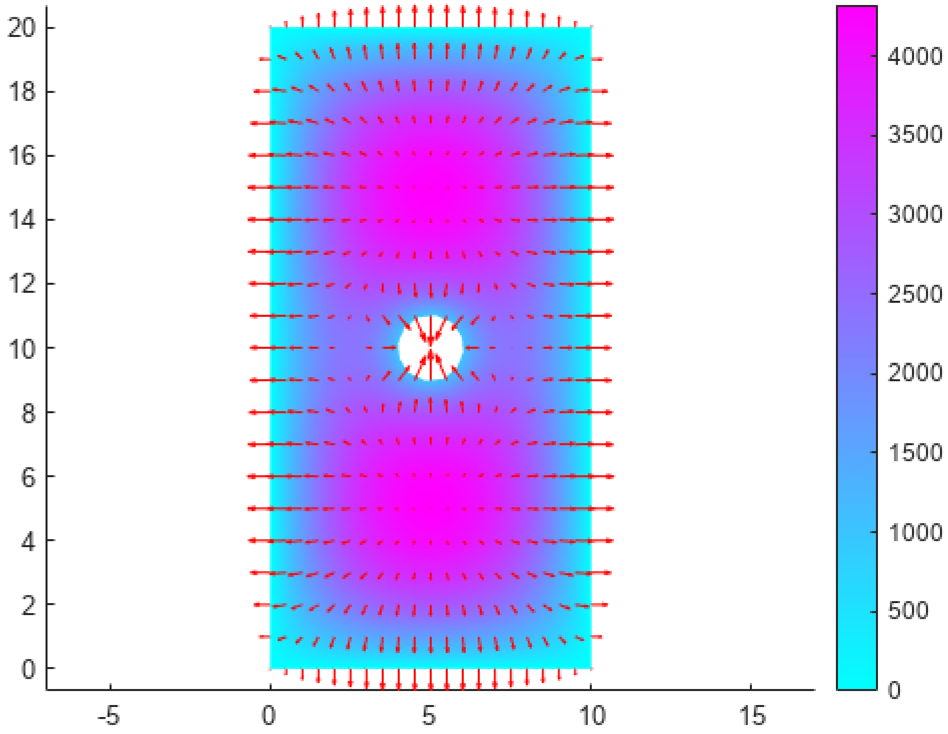
```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:
```

```
        ElectricPotential: [1218x1 double]
            ElectricField: [1x1 FEStruct]
      ElectricFluxDensity: [1x1 FEStruct]
                     Mesh: [1x1 FEMesh]
```

Plot the electric potential and field.

```
pdeplot(emagmodel,"XYData",R.ElectricPotential, ...
                  "FlowData",[R.ElectricField.Ex ...
                              R.ElectricField.Ey])
axis equal
```



Solution to 3-D Electrostatic Analysis Model

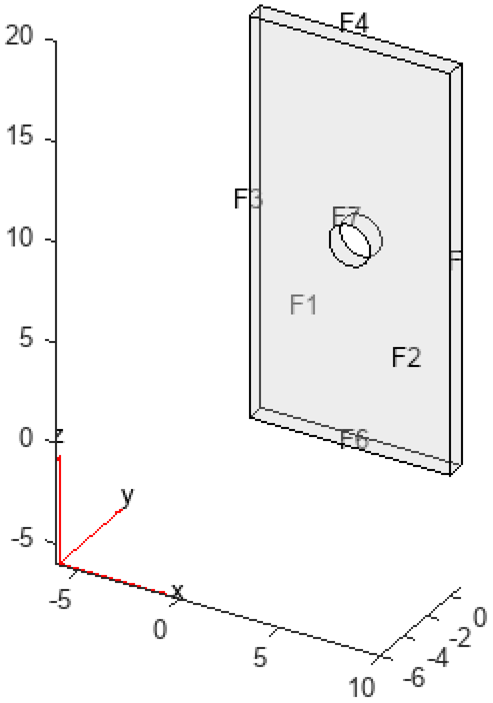
Solve an electromagnetic problem and find the electric potential and field distribution for a 3-D geometry representing a plate with a hole.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Apply the voltage boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
electromagneticBC(emagmodel,"Voltage",1000,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

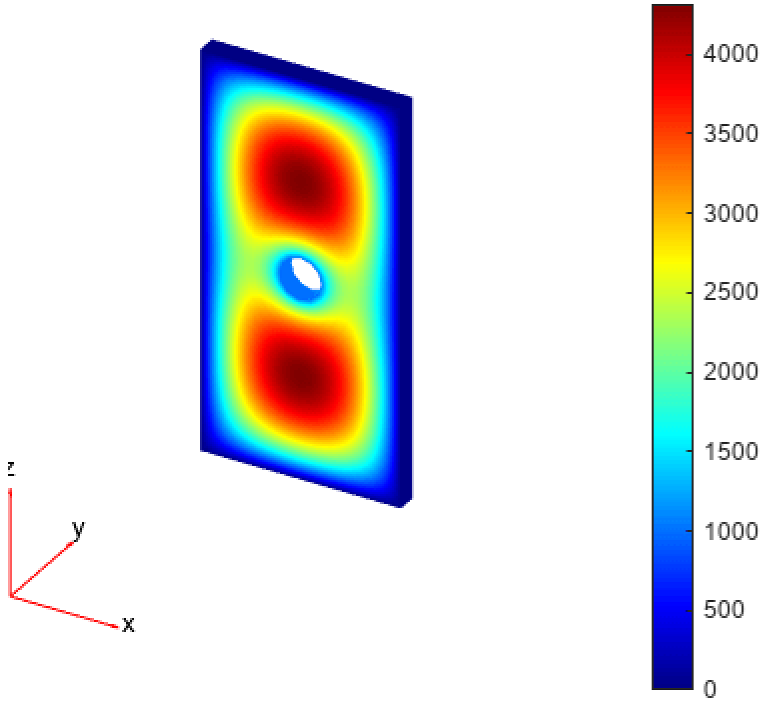
```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:
```

```
        ElectricPotential: [4359x1 double]
            ElectricField: [1x1 FEStruct]
      ElectricFluxDensity: [1x1 FEStruct]
                     Mesh: [1x1 FEMesh]
```

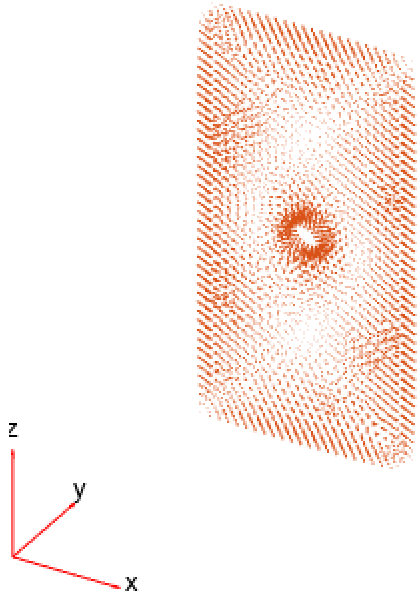
Plot the electric potential.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.ElectricPotential)
```



Plot the electric field.

```
pdeplot3D(emagmodel,"FlowData",[R.ElectricField.Ex ...
                                R.ElectricField.Ey ...
                                R.ElectricField.Ez])
```



Version History

Introduced in R2021a

See Also

`ElectromagneticModel` | `MagnetostaticResults` | `solve`

MagnetostaticResults

Magnetostatic solution and derived quantities

Description

A `MagnetostaticResults` object contains the magnetic potential, magnetic field, magnetic flux density, and mesh values in a form convenient for plotting and postprocessing.

The magnetic potential, magnetic field, and magnetic flux density are calculated at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. Magnetic potential values at the nodes appear in the `MagneticPotential` property. Magnetic field values at the nodes appear in the `MagneticField` property. Magnetic flux density values at the nodes appear in the `MagneticFluxDensity` property.

To interpolate the magnetic potential, magnetic field, and magnetic flux density to a custom grid, such as the one specified by `meshgrid`, use the `interpolateMagneticPotential`, `interpolateMagneticField`, and `interpolateMagneticFlux` functions.

Creation

Solve a magnetostatic problem using the `solve` function. This function returns a solution as a `MagnetostaticResults` object.

Properties

`MagneticPotential` — Magnetic potential values at nodes
vector | `FEStruct` object

This property is read-only.

Magnetic potential values at nodes, returned as a vector for a 2-D problem or an `FEStruct` object for a 3-D problem. The properties of this object contain the components of the magnetic potential at nodes.

`MagneticField` — Magnetic field values at nodes
`FEStruct` object

This property is read-only.

Magnetic field values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of the magnetic field at nodes.

`MagneticFluxDensity` — Magnetic flux density values at nodes
`FEStruct` object

This property is read-only.

Magnetic flux density values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of the magnetic flux density at nodes.

`Mesh` — Finite element mesh

`FEMesh` object

This property is read-only.

Finite element mesh, returned as an `FEMesh` object. For details, see FEMesh. For a 3-D model, the mesh must be linear.

Object Functions

interpolateMagneticPotential	Interpolate magnetic potential in magnetostatic result at arbitrary spatial locations
interpolateMagneticField	Interpolate magnetic field in magnetostatic result at arbitrary spatial locations
interpolateMagneticFlux	Interpolate magnetic flux density in magnetostatic result at arbitrary spatial locations

Examples

Solution to 2-D Magnetostatic Analysis Model

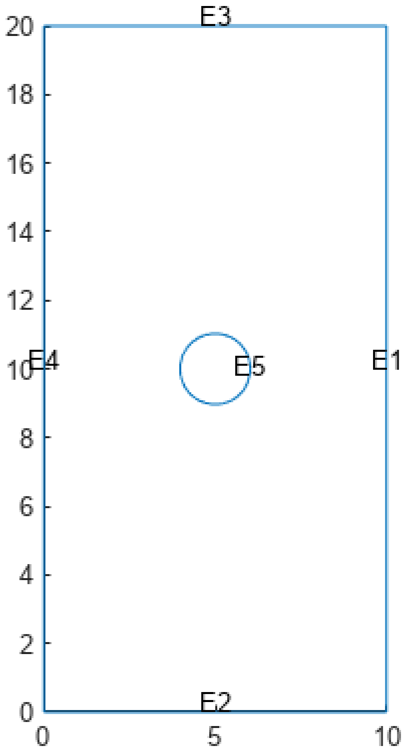
Solve a 2-D electromagnetic problem on a geometry representing a plate with a hole in its center. Plot the resulting magnetic potential and field distribution.

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(emagmodel,"EdgeLabels","on")
```



Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the edges framing the rectangle and the circle.

```
electromagneticBC(emagmodel,"MagneticPotential",0,"Edge",1:4);
electromagneticBC(emagmodel,"MagneticPotential",0.01,"Edge",5);
```

Specify the current density for the entire geometry.

```
electromagneticSource(emagmodel,"CurrentDensity",0.5);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

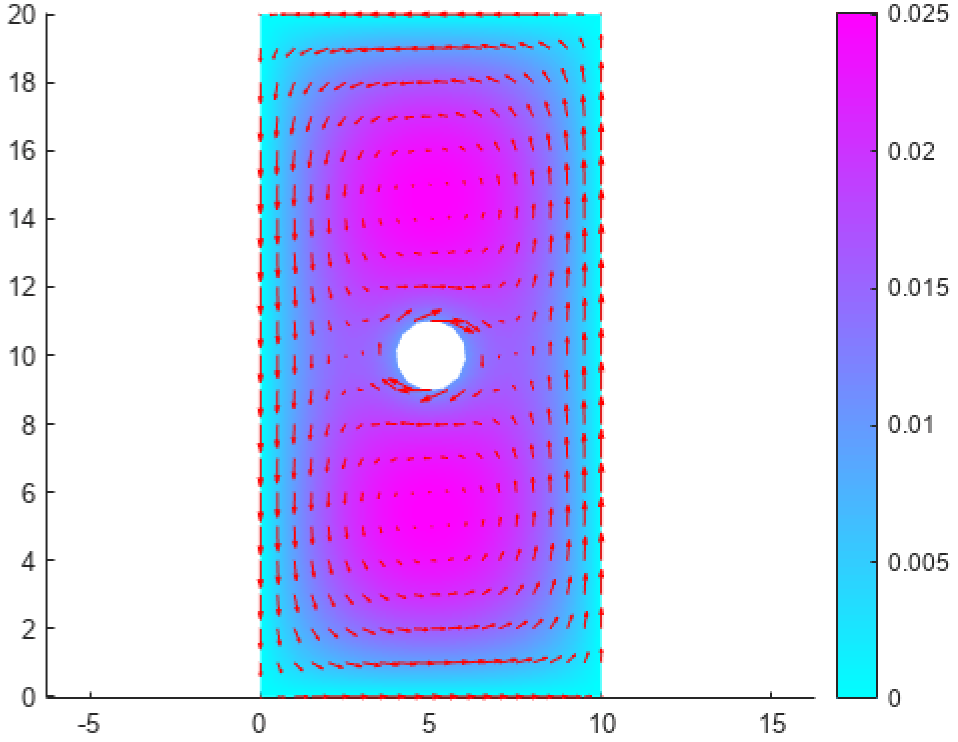
```
R = solve(emagmodel)
```

```
R = 
  MagnetostaticResults with properties:
```

```
      MagneticPotential: [1218x1 double]
          MagneticField: [1x1 FEStruct]
    MagneticFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

Plot the magnetic potential and field.

```
pdeplot(emagmodel,"XYData",R.MagneticPotential, ...
                  "FlowData",[R.MagneticField.Hx ...
                              R.MagneticField.Hy])
axis equal
```



Solution to 3-D Magnetostatic Analysis Model

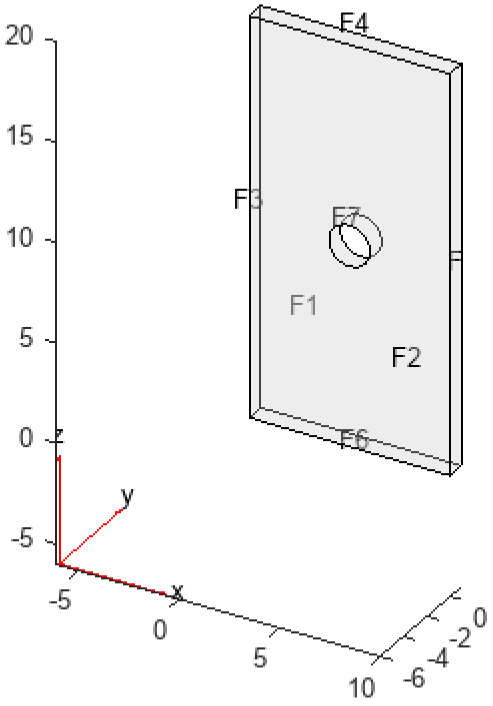
Solve a 3-D electromagnetic problem on a geometry representing a plate with a hole in its center. Plot the resulting magnetic potential and field distribution.

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Specify the current density for the entire geometry.

```
electromagneticSource(emagmodel,"CurrentDensity",[0;0;0.5]);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

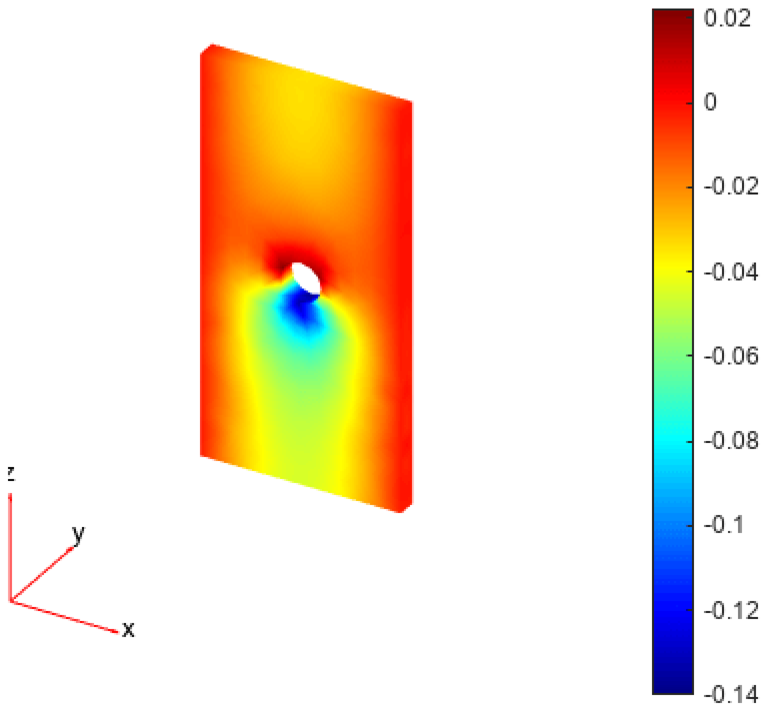
```
R = solve(emagmodel)
```

```
R = 
  MagnetostaticResults with properties:
```

```
    MagneticPotential: [1x1 FEStruct]
        MagneticField: [1x1 FEStruct]
  MagneticFluxDensity: [1x1 FEStruct]
                 Mesh: [1x1 FEMesh]
```

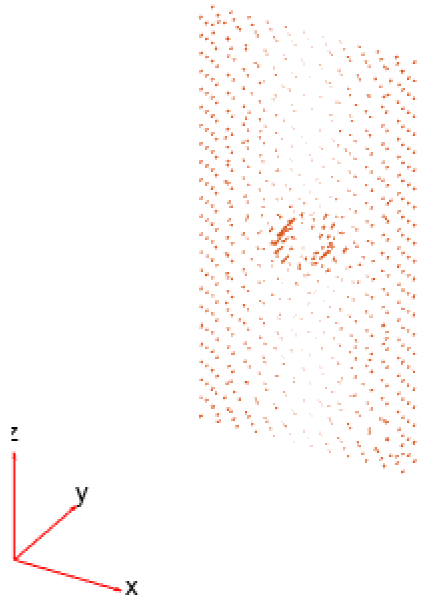
Plot the z-component of the magnetic potential.

```
pdeplot3D(emagmodel,"ColormapData",R.MagneticPotential.Az)
```



Plot the magnetic field.

```
pdeplot3D(emagmodel,"FlowData",[R.MagneticField.Hx ...
                                R.MagneticField.Hy ...
                                R.MagneticField.Hz])
```



DC Conduction Solution as Current Density for Magnetostatic Model

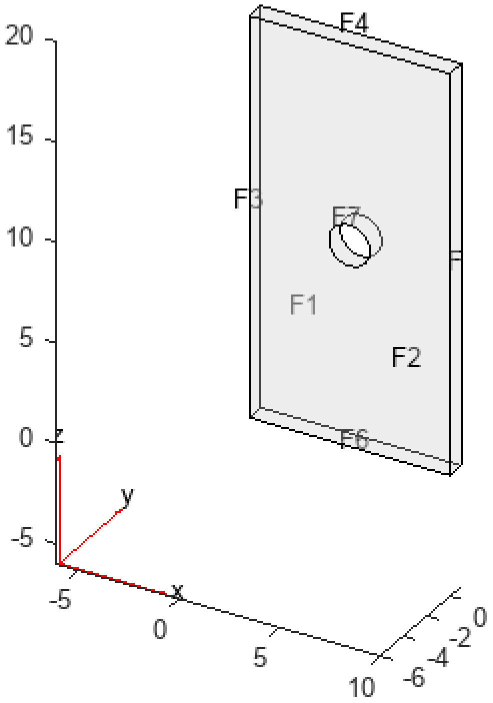
Use a solution obtained by performing a DC conduction analysis to specify current density for a magnetostatic model.

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the conductivity of the material.

```
electromagneticProperties(emagmodel,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the left, right, top, and bottom faces of the plate.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
```

Specify the surface current density on the face bordering the hole.

```
electromagneticBC(emagmodel,"SurfaceCurrentDensity",100,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

```
R = solve(emagmodel);
```

Change the analysis type of the model to magnetostatic.

```
emagmodel.AnalysisType = "magnetostatic";
```

This model already has a quadratic mesh that you generated for the DC conduction analysis. For a 3-D magnetostatic model, the mesh must be linear. Generate a new linear mesh. The `generateMesh` function creates a linear mesh by default if the model is 3-D and magnetostatic.

```
generateMesh(emagmodel);
```

Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Specify the current density for the entire geometry using the DC conduction solution.

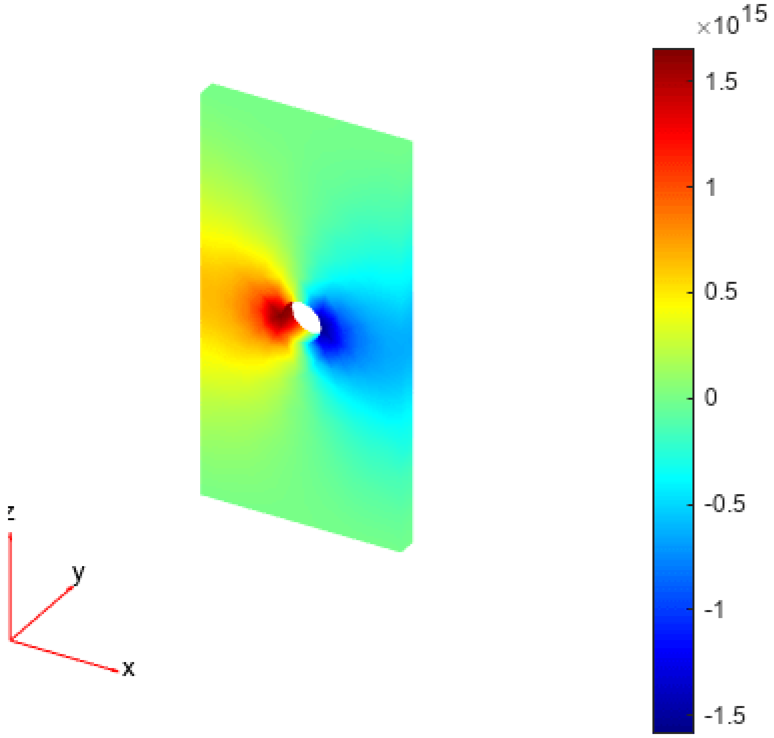
```
electromagneticSource(emagmodel,"CurrentDensity",R);
```

Solve the model.

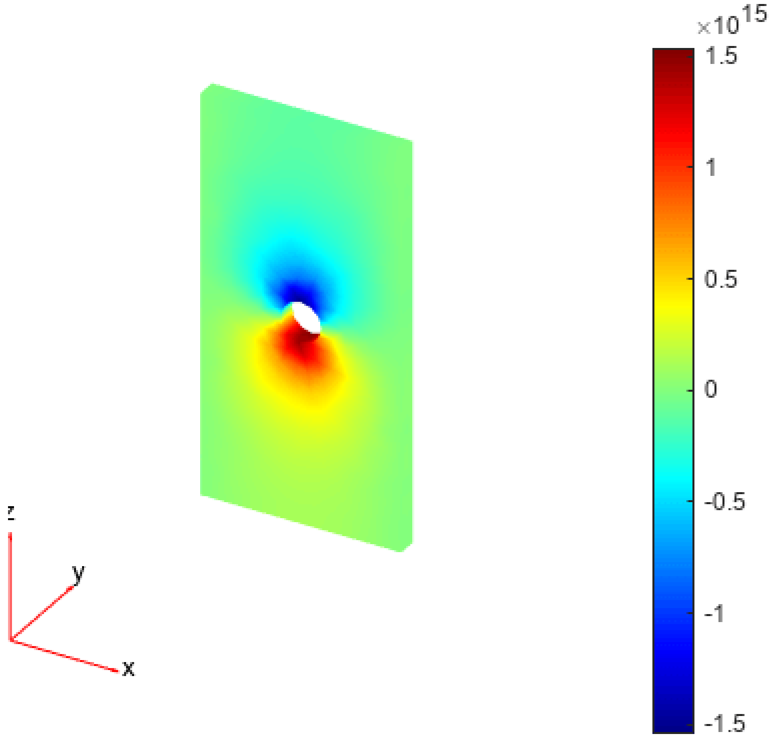
```
Rmagnetostatic = solve(emagmodel);
```

Plot the *x*- and *z*-components of the magnetic potential.

```
pdeplot3D(emagmodel,"ColormapData",Rmagnetostatic.MagneticPotential.Ax)
```



```
pdeplot3D(emagmodel,"ColormapData",Rmagnetostatic.MagneticPotential.Az)
```



Solution to 2-D Magnetostatic Model with Permanent Magnet

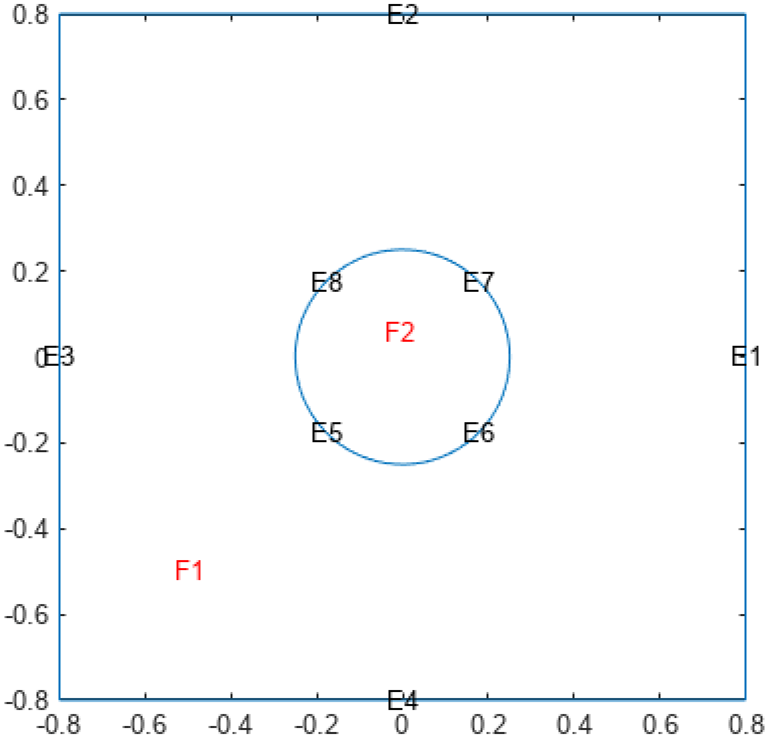
Solve a magnetostatic model of a copper square with a permanent neodymium magnet in its center.

Create the unit square geometry with a circle in its center.

```
L = 0.8;
r = 0.25;
sq = [3 4 -L L L -L -L -L L L]';
circ = [1 0 0 r 0 0 0 0 0 0]';
gd = [sq,circ];
sf = "sq + circ";
ns = char('sq','circ');
ns = ns';
g = decsg(gd,sf,ns);
```

Plot the geometry with the face and edge labels.

```
pdegplot(g,"FaceLabels","on","EdgeLabels","on")
```



Create a magnetostatic model and include the geometry in the model.

```
emagmodel = createpde("electromagnetic","magnetostatic");
geometryFromEdges(emagmodel,g);
```

Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the copper for the square.

```
electromagneticProperties(emagmodel,"Face",1, ...
                                    "RelativePermeability",1);
```

Specify the relative permeability of the neodymium for the circle.

```
electromagneticProperties(emagmodel,"Face",2, ...
                                    "RelativePermeability",1.05);
```

Specify the magnetization magnitude for the neodymium magnet.

```
M = 1e6;
```

Specify magnetization on the circular face in the positive *x*-direction. Magnetization for a 2-D model is a column vector of two elements.

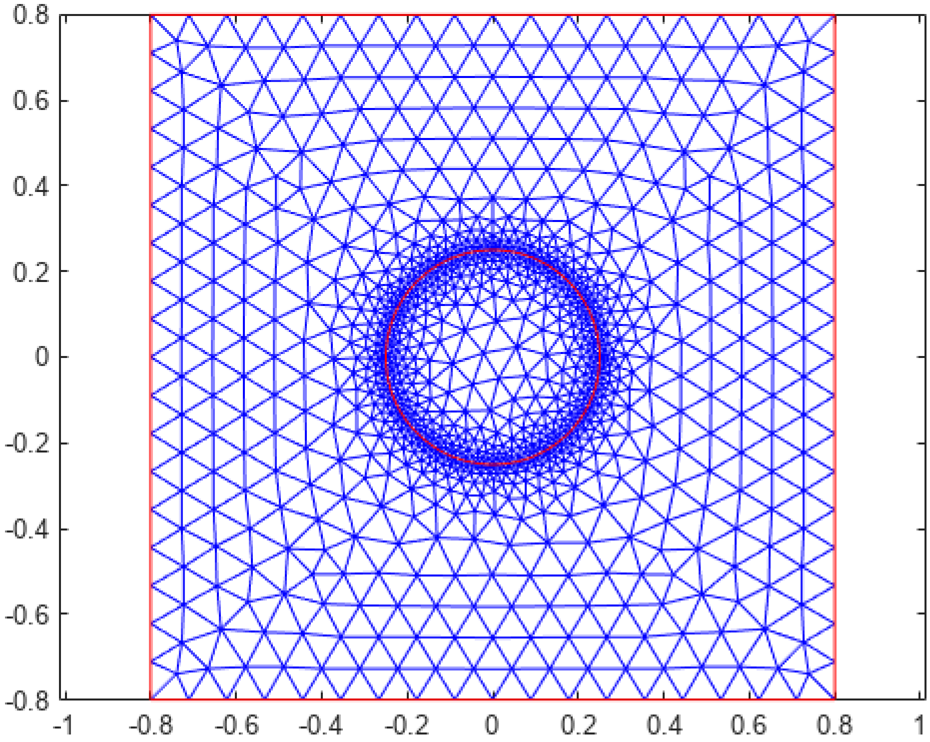
```
dir = [1;0];
electromagneticSource(emagmodel,"Face",2,"Magnetization",M*dir);
```

Apply the magnetic potential boundary conditions on the edges framing the square.

```
electromagneticBC(emagmodel,"Edge",1:4,"MagneticPotential",0);
```

Generate the mesh with finer meshing near the edges of the circle.

```
generateMesh(emagmodel,"Hedge",{5:8,0.007});
figure
pdemesh(emagmodel)
```

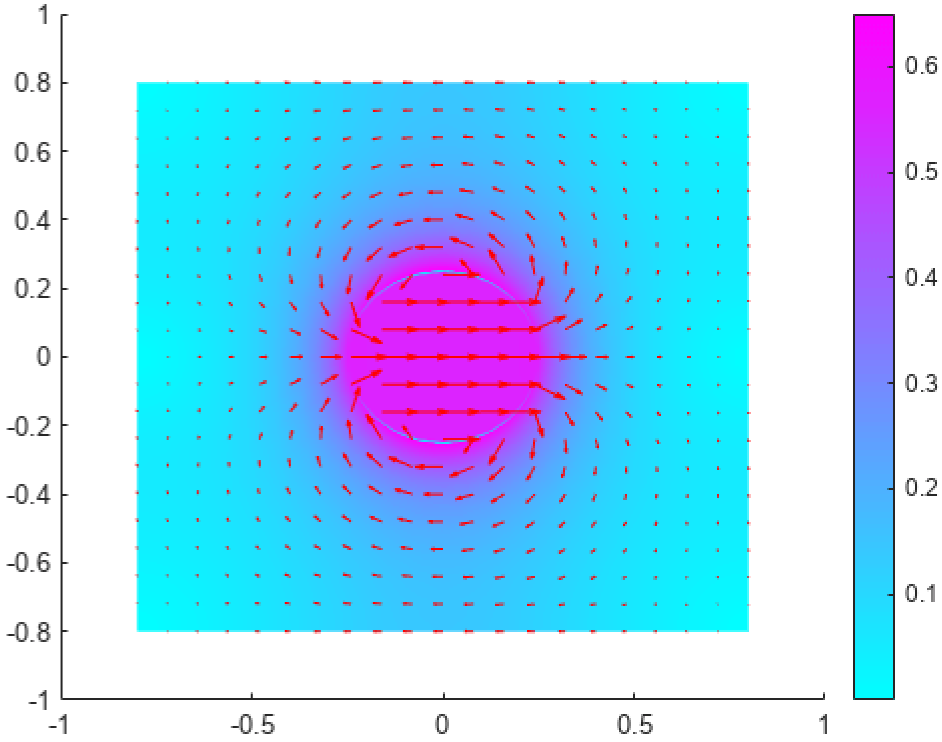


Solve the model, and find the resulting magnetic fields B and H. Here, $B = \mu H + \mu_0 M$, where μ is the absolute magnetic permeability of the material, μ_0 is the vacuum permeability, and M is the magnetization.

```
R = solve(emagmodel);
Bmag = sqrt(R.MagneticFluxDensity.Bx.^2 + R.MagneticFluxDensity.By.^2);
Hmag = sqrt(R.MagneticField.Hx.^2 + R.MagneticField.Hy.^2);
```

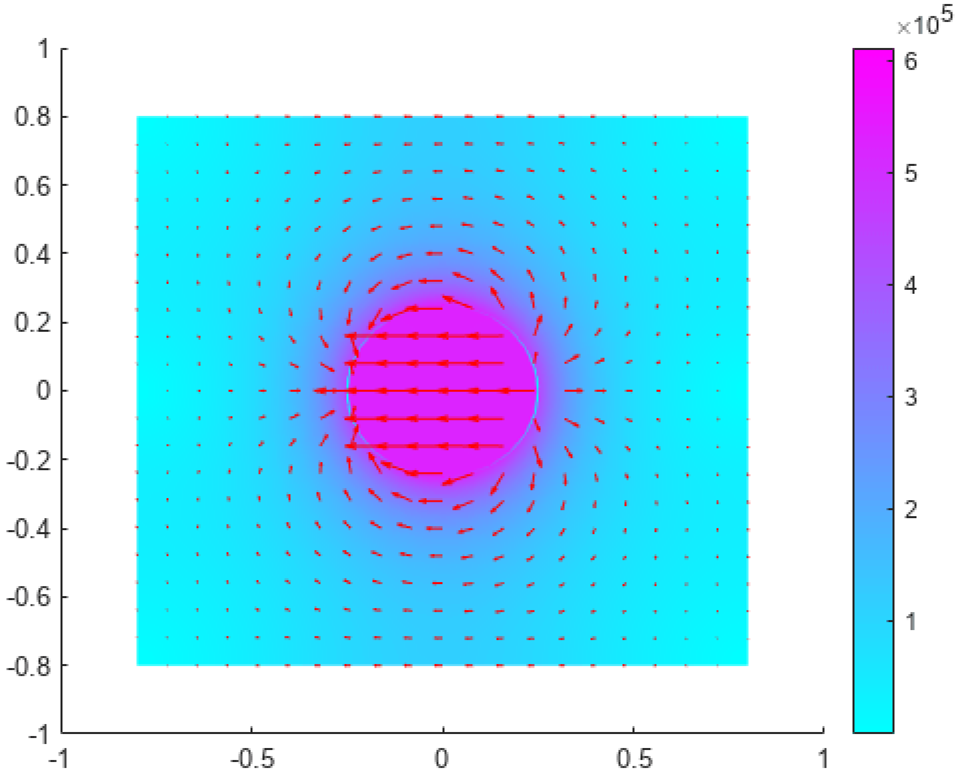
Plot the magnetic field B.

```
figure
pdeplot(emagmodel,"XYData",Bmag, ...
                  "FlowData",[R.MagneticFluxDensity.Bx ...
                              R.MagneticFluxDensity.By])
```



Plot the magnetic field *H*.

```
figure
pdeplot(emagmodel,"XYData",Hmag, ...
                  "FlowData",[R.MagneticField.Hx R.MagneticField.Hy])
```



Version History

Introduced in R2021a

See Also

`ElectromagneticModel` | `ElectrostaticResults` | `ConductionResults` | `solve`

HarmonicResults

Harmonic electromagnetic solution

Description

A `HarmonicResults` object contains the electric or magnetic field, frequency, and mesh values in a form convenient for plotting and postprocessing.

The electric or magnetic field values are calculated at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. Electric field values at the nodes appear in the `ElectricField` property. Magnetic field values at the nodes appear in the `MagneticField` property.

To interpolate the electric or magnetic field to a custom grid, such as the one specified by `meshgrid`, use the `interpolateHarmonicField` function.

Creation

Solve a harmonic electromagnetic analysis problem using the `solve` function. This function returns a solution as a `HarmonicResults` object.

Properties

`ElectricField` — Electric field values at nodes
`FEStruct` object

This property is read-only.

Electric field values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of the electric field at nodes.

`MagneticField` — Magnetic field values at nodes
`FEStruct` object

This property is read-only.

Magnetic field values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of the magnetic field at nodes.

`Frequency` — Solution frequencies
vector

This property is read-only.

Solution frequencies, returned as a vector.

Data Types: `double`

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as an `FEMesh` object. For details, see FEMesh.

Object Functions

interpolateHarmonicField	Interpolate electric or magnetic field in harmonic result at arbitrary spatial locations

Examples

Solution to 2-D Harmonic Electromagnetic Model

For an electromagnetic harmonic analysis problem, find the *x*- and *y*-components of the electric field. Solve the problem on a domain consisting of a square with a circular hole.

Create an electromagnetic model for harmonic analysis.

```
emagmodel = createpde("electromagnetic","harmonic");
```

Define a circle in a square, place them in one matrix, and create a set formula that subtracts the circle from the square.

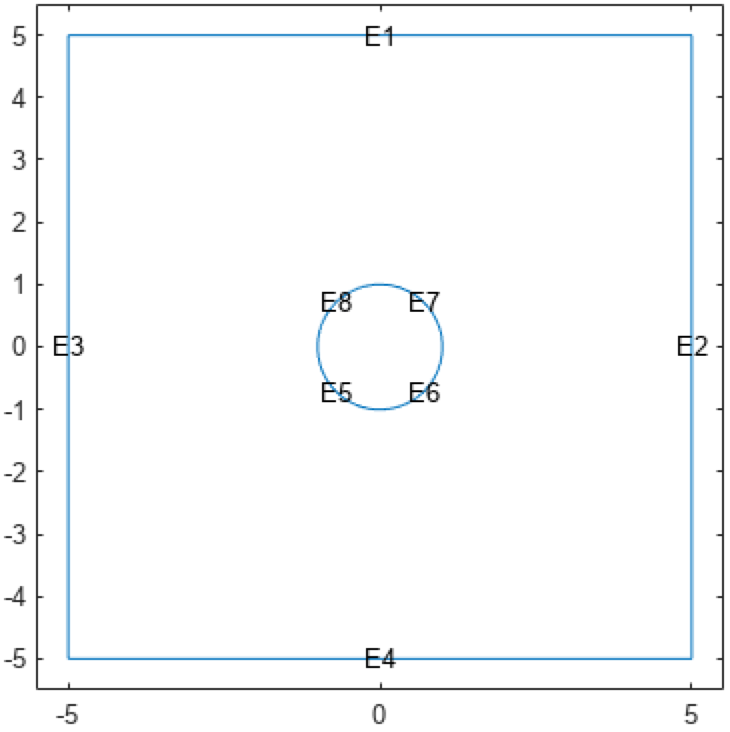
```
SQ = [3,4,-5,-5,5,5,-5,5,5,-5]';
C = [1,0,0,1]';
C = [C;zeros(length(SQ) - length(C),1)];
gm = [SQ,C];
sf = 'SQ-C';
```

Create the geometry.

```
ns = char('SQ','C');
ns = ns';
g = decsg(gm,sf,ns);
```

Include the geometry in the model and plot the geometry with the edge labels.

```
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-5.5 5.5])
ylim([-5.5 5.5])
```



Specify the vacuum permittivity and permeability values as 1.

```
emagmodel.VacuumPermittivity = 1;
emagmodel.VacuumPermeability = 1;
```

Specify the relative permittivity, relative permeability, and conductivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1, ...
                                    "RelativePermeability",1, ...
                                    "Conductivity",0);
```

Apply the absorbing boundary condition with a thickness of 2 on the edges of the square. Use the default attenuation rate for the absorbing region.

```
electromagneticBC(emagmodel,"Edge",1:4, ...
                            "FarField","absorbing", ...
                            "Thickness",2);
```

Specify an electric field on the edges of the hole.

```
E = @(location,state) [1;0]*exp(-1i*2*pi*location.y);
electromagneticBC(emagmodel,"Edge",5:8,"ElectricField",E);
```

Generate a mesh.

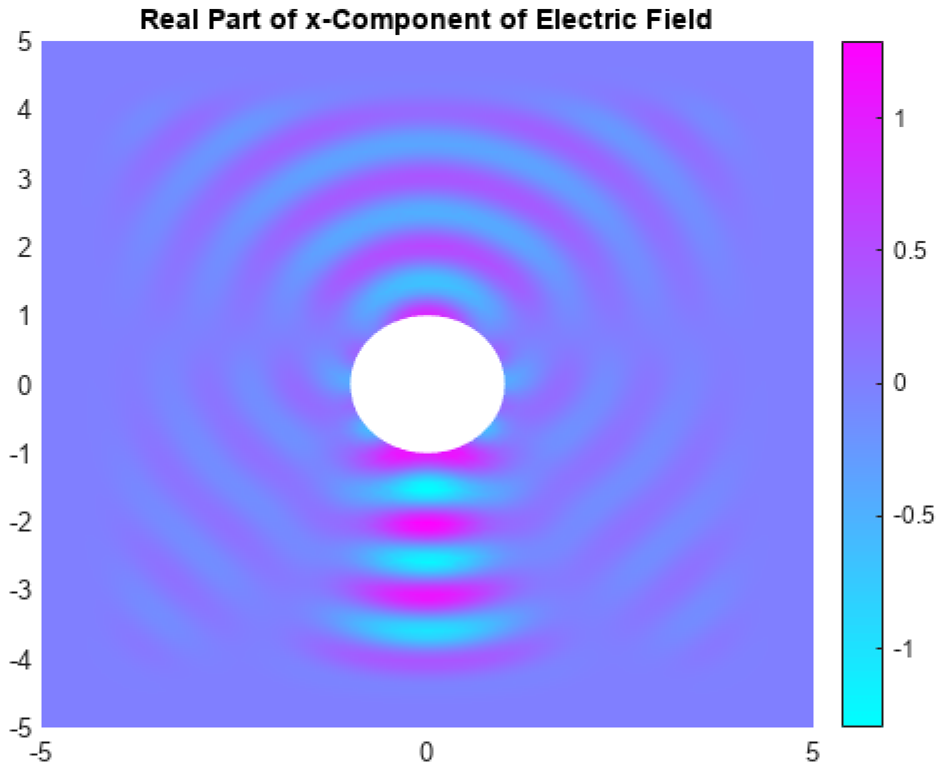
```
generateMesh(emagmodel,"Hmax",1/2^3);
```

Solve the model for a frequency of 2π.

```
result = solve(emagmodel,"Frequency",2*pi);
```

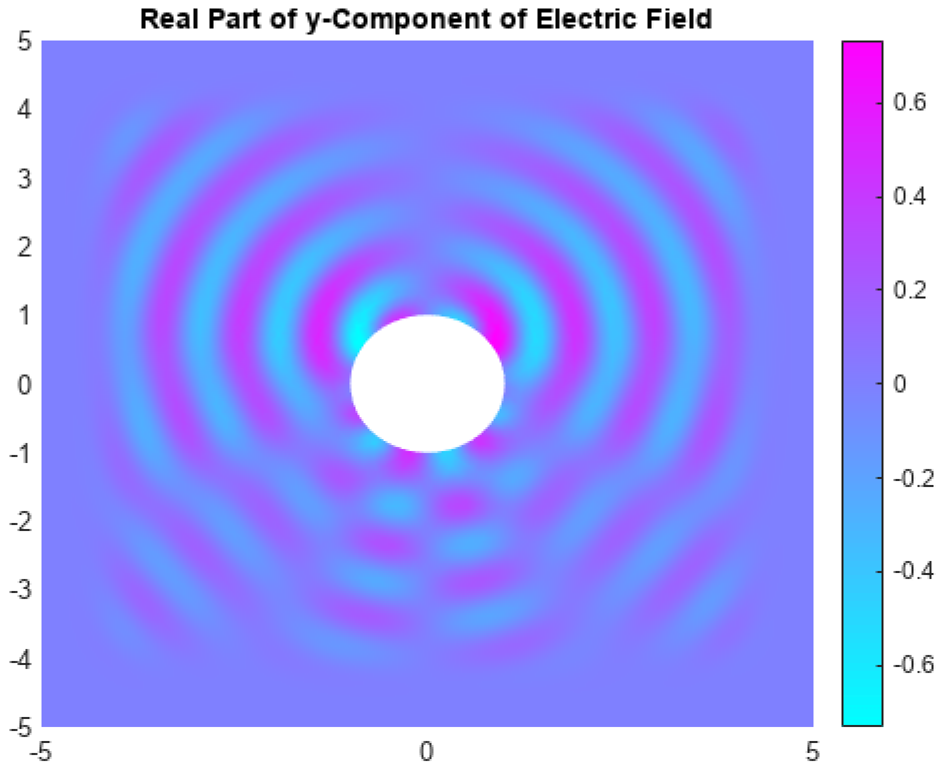
Plot the real part of the *x*-component of the resulting electric field.

```
figure
pdeplot(emagmodel,"XYData",real(result.ElectricField.Ex));
title("Real Part of x-Component of Electric Field")
```



Plot the real part of the *y*-component of the resulting electric field.

```
figure
pdeplot(emagmodel,"XYData",real(result.ElectricField.Ey));
title("Real Part of y-Component of Electric Field")
```



Version History

Introduced in R2022a

See Also

`interpolateHarmonicField` | `solve` | `ElectromagneticModel`

ConductionResults

DC conduction solution

Description

A `ConductionResults` object contains the electric potential, electric field, current density, and mesh values in a form convenient for plotting and postprocessing.

The electric potential, electric field, and current density values are calculated at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. Electric potential values at the nodes appear in the `ElectricPotential` property. Electric field values at the nodes appear in the `ElectricField` property. Current density values at the nodes appear in the `CurrentDensity` property.

To interpolate the electric potential, electric field, and current density to a custom grid, such as the one specified by `meshgrid`, use the `interpolateElectricPotential`, `interpolateElectricField`, and `interpolateCurrentDensity` functions.

Creation

Solve a DC conduction problem using the `solve` function. This function returns a solution as a `ConductionResults` object.

Properties

`ElectricPotential` — Electric potential values at nodes
vector

This property is read-only.

Electric potential values at nodes, returned as a vector.

Data Types: `double`

`ElectricField` — Electric field values at nodes
`FEStruct` object

This property is read-only.

Electric field values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of the electric field at nodes.

`CurrentDensity` — Current density values at nodes
`FEStruct` object

This property is read-only.

Electric flux density values at nodes, returned as an `FEStruct` object. The properties of this object contain the components of the electric flux density at nodes.

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as an `FEMesh` object. For details, see FEMesh.

Object Functions

interpolateElectricPotential	Interpolate electric potential in electrostatic or DC conduction result at arbitrary spatial locations
interpolateElectricField	Interpolate electric field in electrostatic or DC conduction result at arbitrary spatial locations
interpolateCurrentDensity	Interpolate current density in DC conduction result at arbitrary spatial locations

Examples

Solution to 3-D DC Conduction Model

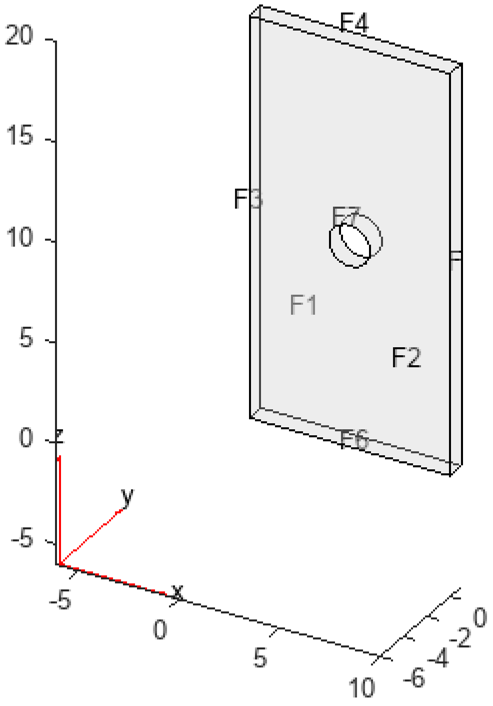
Solve a DC conduction problem on a geometry representing a 3-D plate with a hole in its center. Plot the electric potential and the components of the current density.

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the conductivity of the material.

```
electromagneticProperties(emagmodel,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the left, right, top, and bottom faces of the plate.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
```

Specify the surface current density on the face bordering the hole.

```
electromagneticBC(emagmodel,"SurfaceCurrentDensity",100,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

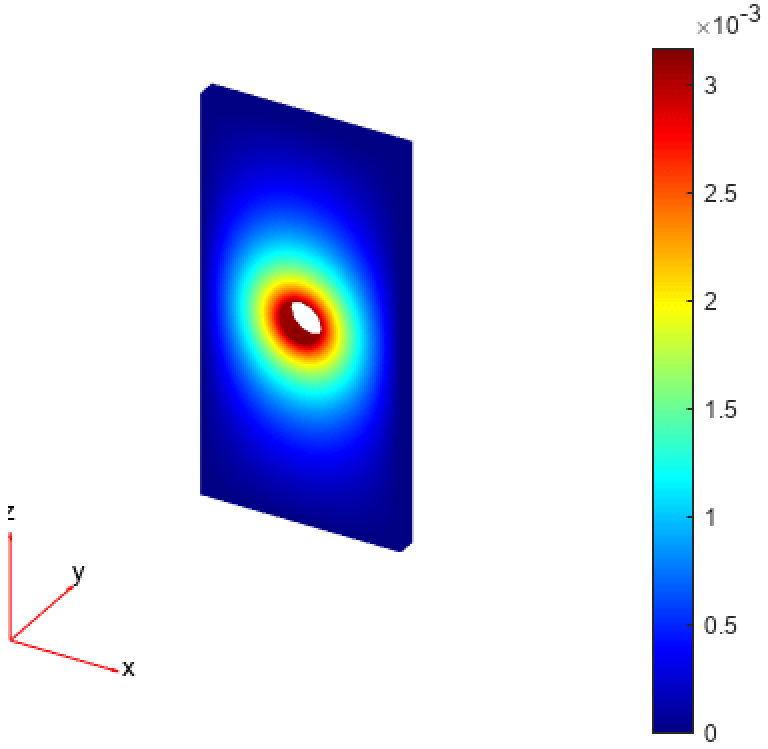
Solve the model.

```
R = solve(emagmodel)
```

```
R = 
  ConductionResults with properties:

    ElectricPotential: [4359x1 double]
        ElectricField: [1x1 FEStruct]
       CurrentDensity: [1x1 FEStruct]
                 Mesh: [1x1 FEMesh]
```

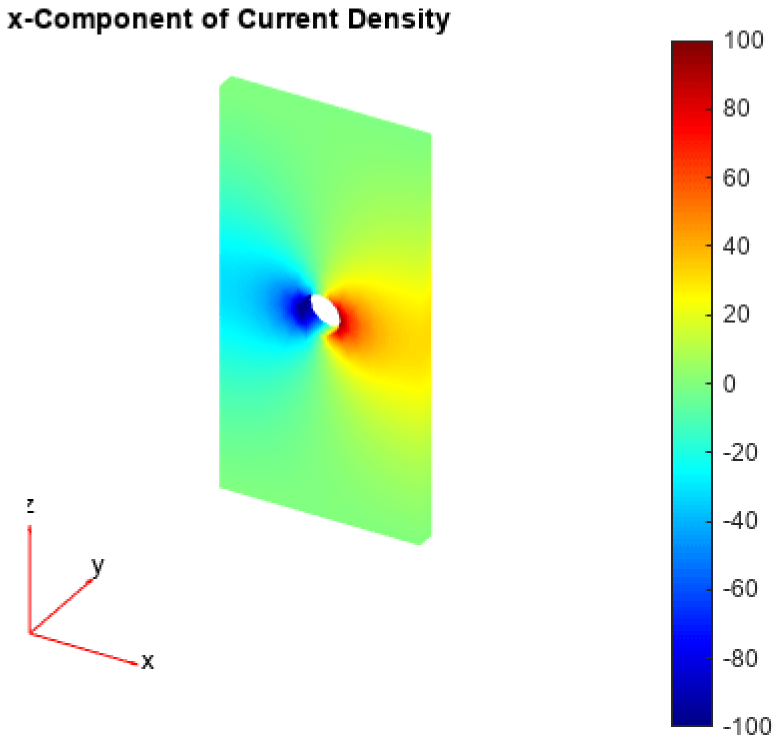
Plot the electric potential.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.ElectricPotential)
```



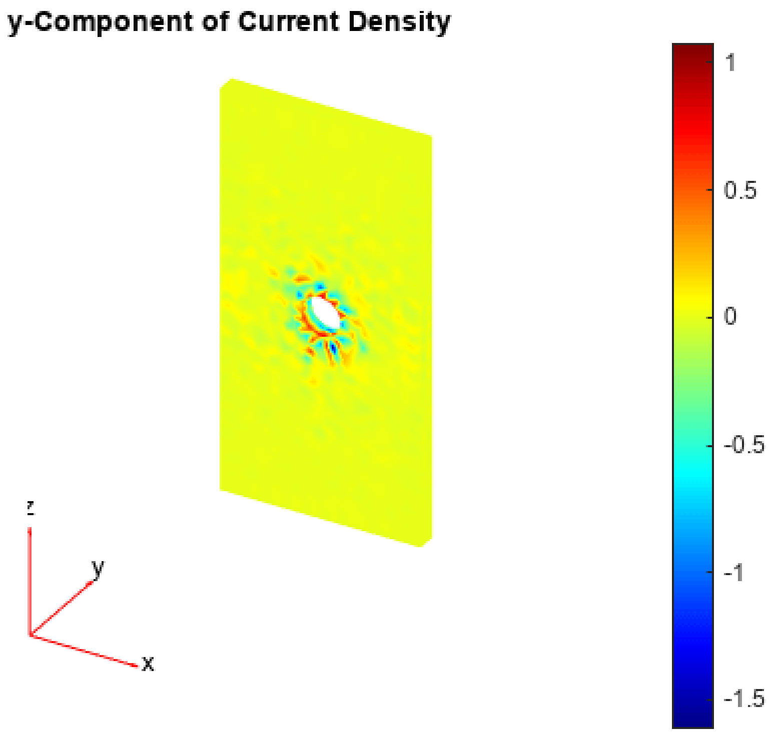
Plot the *x*-component of the current density.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.CurrentDensity.Jx)
title("x-Component of Current Density")
```



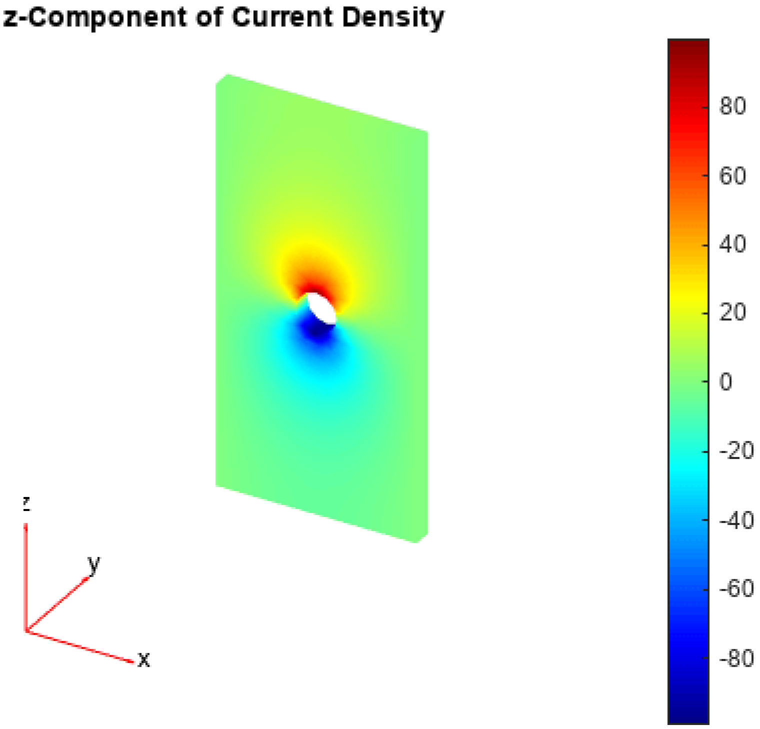
Plot the *y*-component of the current density.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.CurrentDensity.Jy)
title("y-Component of Current Density")
```



Plot the *z*-component of the current density.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.CurrentDensity.Jz)
title("z-Component of Current Density")
```



DC Conduction Solution as Current Density for Magnetostatic Model

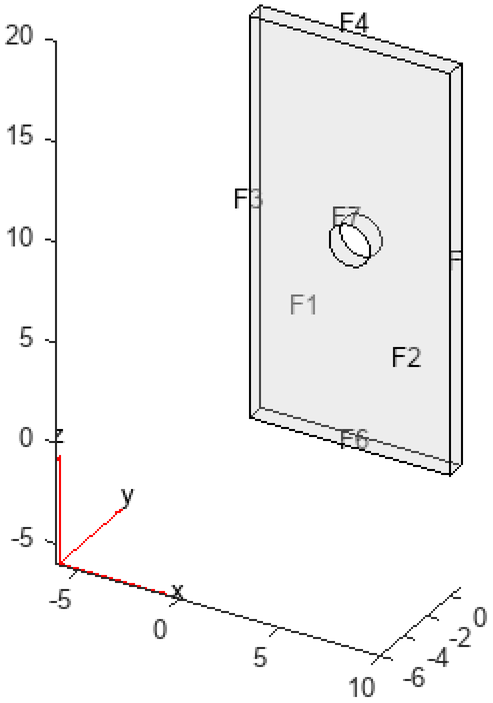
Use a solution obtained by performing a DC conduction analysis to specify current density for a magnetostatic model.

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the conductivity of the material.

```
electromagneticProperties(emagmodel,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the left, right, top, and bottom faces of the plate.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
```

Specify the surface current density on the face bordering the hole.

```
electromagneticBC(emagmodel,"SurfaceCurrentDensity",100,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

```
R = solve(emagmodel);
```

Change the analysis type of the model to magnetostatic.

```
emagmodel.AnalysisType = "magnetostatic";
```

This model already has a quadratic mesh that you generated for the DC conduction analysis. For a 3-D magnetostatic model, the mesh must be linear. Generate a new linear mesh. The `generateMesh` function creates a linear mesh by default if the model is 3-D and magnetostatic.

```
generateMesh(emagmodel);
```

Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Specify the current density for the entire geometry using the DC conduction solution.

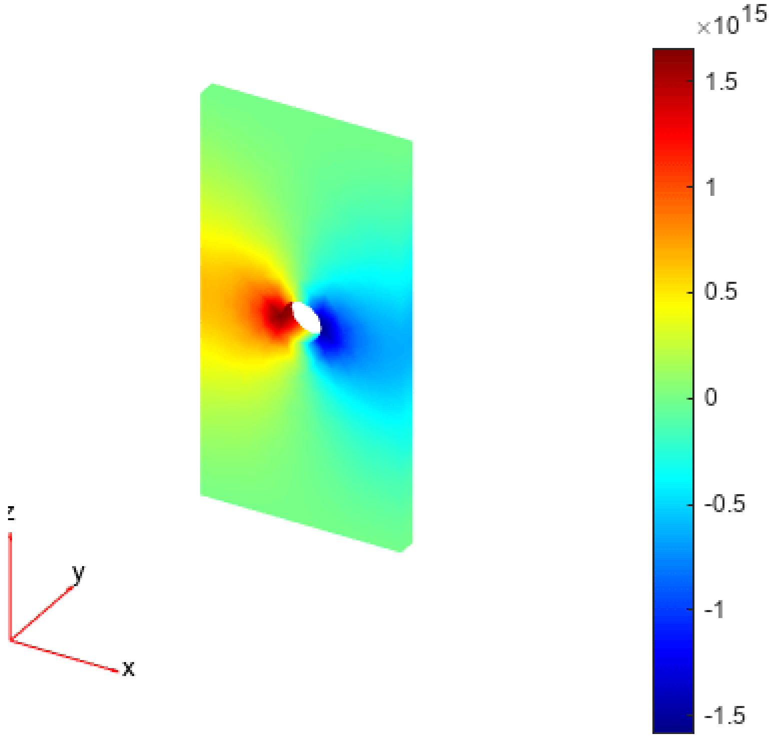
```
electromagneticSource(emagmodel,"CurrentDensity",R);
```

Solve the model.

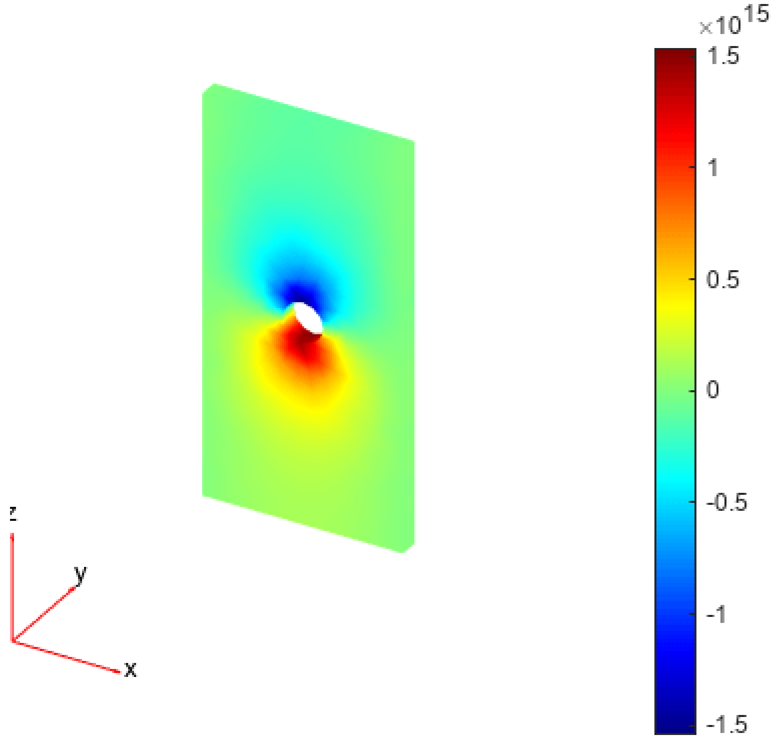
```
Rmagnetostatic = solve(emagmodel);
```

Plot the *x*- and *z*-components of the magnetic potential.

```
pdeplot3D(emagmodel,"ColormapData",Rmagnetostatic.MagneticPotential.Ax)
```



```
pdeplot3D(emagmodel,"ColormapData",Rmagnetostatic.MagneticPotential.Az)
```



Version History

Introduced in R2022b

See Also

`ElectromagneticModel` | `solve` | `interpolateElectricPotential` | `interpolateElectricField` | `interpolateCurrentDensity`

evaluate

Package: `pde`

Interpolate data to selected locations

Note 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 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
model = createpde;
gm = geometryFromEdges(model,g);
% Zero Dirichlet conditions
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",(1:gm.NumEdges), ...
                             "u",0);
[p,e,t] = initmesh(gm);
c = 1;
```

```
a = 0;
f = 1;
u = assempde(model,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
model = createpde;
gm = geometryFromEdges(model,g);
% Zero Dirichlet conditions
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",(1:gm.NumEdges), ...
                             "u",0);
[p,e,t] = initmesh(gm);
c = 1;
a = 0;
f = 1;
u = assempde(model,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
model = createpde(3);
gm = geometryFromEdges(model,g);
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",(1:gm.NumEdges), ...
                             "u",zeros(3,1));
[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(model,p,e,t,c,a,f); % solve the PDE
```

Construct an interpolant for the solution.

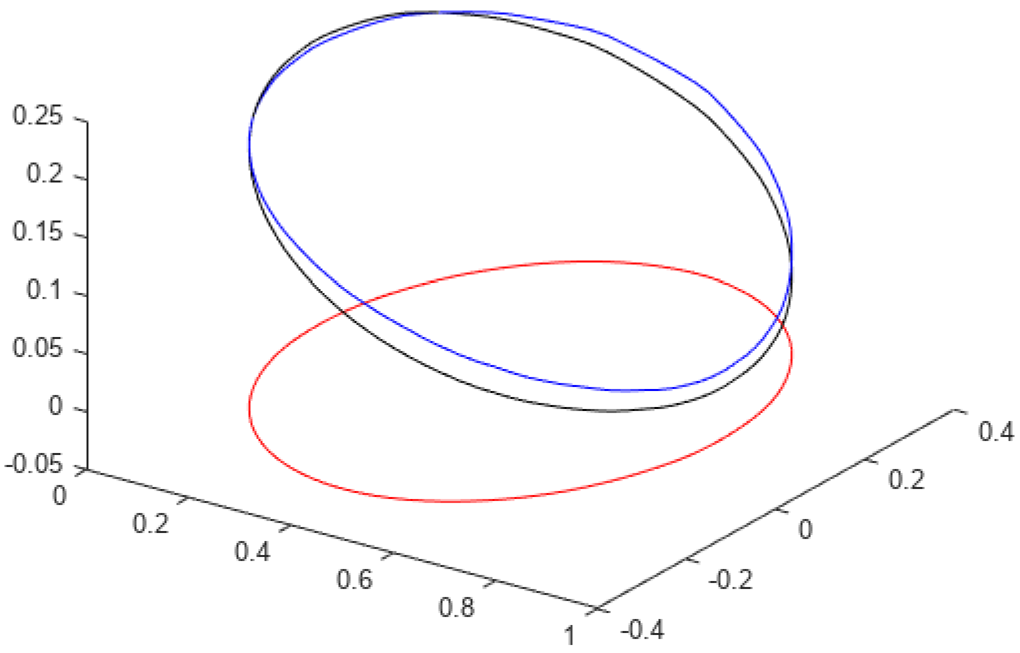
```
F = pdeInterpolant(p,t,u); % create the interpolant
```

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
model = createpde;
gm = geometryFromEdges(model,g);
% Zero Dirichlet conditions
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",(1:gm.NumEdges), ...
                             "u",0);
[p,e,t] = initmesh(gm);
c = 1;
a = 0;
```

```
f = 1;
d = 1;
tlist = 0:1/4:1;
u = parabolic(0,tlist,model,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 = 1×5

         0    0.1809    0.2278    0.2388    0.2413
```

The solution starts at 0 at time 0, as it should. It grows to about 1/4 at time 1.

Interpolate to a Grid

This example shows how to interpolate an elliptic solution to a grid.

Define and Solve the Problem

Use the built-in geometry functions to create an L-shaped region with zero Dirichlet boundary conditions. Solve an elliptic PDE with coefficients $c = 1$, $a = 0$, $f = 1$, with zero Dirichlet boundary conditions.

```
[p,e,t] = initmesh("lshapeg"); % Predefined geometry
u = assempde("lshapeb",p,e,t,1,0,1); % Predefined boundary condition
```

Create an Interpolant

Create an interpolant for the solution.

```
F = pdeInterpolant(p,t,u);
```

Create a Grid for the Solution

```
xgrid = -1:0.1:1;
ygrid = -1:0.2:1;
[X,Y] = meshgrid(xgrid,ygrid);
```

The resulting grid has some points that are outside the L-shaped region.

Evaluate the Solution On the Grid

```
uout = evaluate(F,X,Y);
```

The interpolated solution `uout` is a column vector. You can reshape it to match the size of `X` or `Y`. This gives a matrix, like the output of the `tri2grid` function.

```
Z = reshape(uout,size(X));
```

Input Arguments

F — Interpolant
output of `pdeInterpolant`

Interpolant, specified as the output of `pdeInterpolant`.

Example: `F = pdeInterpolant(p,t,u)`

`pOut` — Query points
matrix with two or three rows

Query points, specified as a matrix with two or three rows. The first row represents the `x` component of the query points, the second row represents the `y` component, and, for 3-D geometry, the third row represents the `z` component. `evaluate` computes the interpolant at each column of `pOut`. In other words, `evaluate` interpolates at the points `pOut(:,k)`.

Example: `pOut = [-1.5,0,1;`
`1,1,2.2]`

Data Types: `double`

x — Query point component
vector or array

Query point component, specified as a vector or array. `evaluate` interpolates at either 2-D points `[x(k),y(k)]` or at 3-D points `[x(k),y(k),z(k)]`. The `x` and `y`, and `z` arrays must contain the same number of entries.

`evaluate` transforms query point components to the linear index representation, such as `x(:)`.

Example: `x = -1:0.2:3`

Data Types: `double`

y — Query point component
vector or array

Query point component, specified as a vector or array. `evaluate` interpolates at either 2-D points `[x(k),y(k)]` or at 3-D points `[x(k),y(k),z(k)]`. The `x` and `y`, and `z` arrays must contain the same number of entries.

`evaluate` transforms query point components to the linear index representation, such as `y(:)`.

Example: `y = -1:0.2:3`

Data Types: `double`

`z` — Query point component
vector or array

Query point component, specified as a vector or array. `evaluate` interpolates at either 2-D points `[x(k),y(k)]` or at 3-D points `[x(k),y(k),z(k)]`. The `x` and `y`, and `z` arrays must contain the same number of entries.

`evaluate` transforms query point components to the linear index representation, such as `z(:)`.

Example: `z = -1:0.2:3`

Data Types: `double`

Output Arguments

`uOut` — Interpolated values
array

Interpolated values, returned as an array. `uOut` has the same number of columns as the data `u` used in creating `F`. If `u` depends on time, `uOut` contains a column for each time step. For time-independent `u`, `uOut` has one column.

The number of rows in `uOut` is the number of equations in the PDE system, `N`, times the number of query points, `pOut`. The first `pOut` rows correspond to equation 1, the next `pOut` rows correspond to equation 2, and so on.

If a query point is outside the mesh, `evaluate` returns `NaN` for that point.

More About

Element

An element is a basic unit in the finite-element method.

For 2-D problems, an element is a triangle 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 details, see “Mesh Data” on page 2-175.

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.

Version History

Introduced in R2014b

See Also

`pdeInterpolant`

Topics

"Mesh Data" on page 2-175

evaluateCGradient

Package: `pde`

Evaluate flux of PDE solution

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.

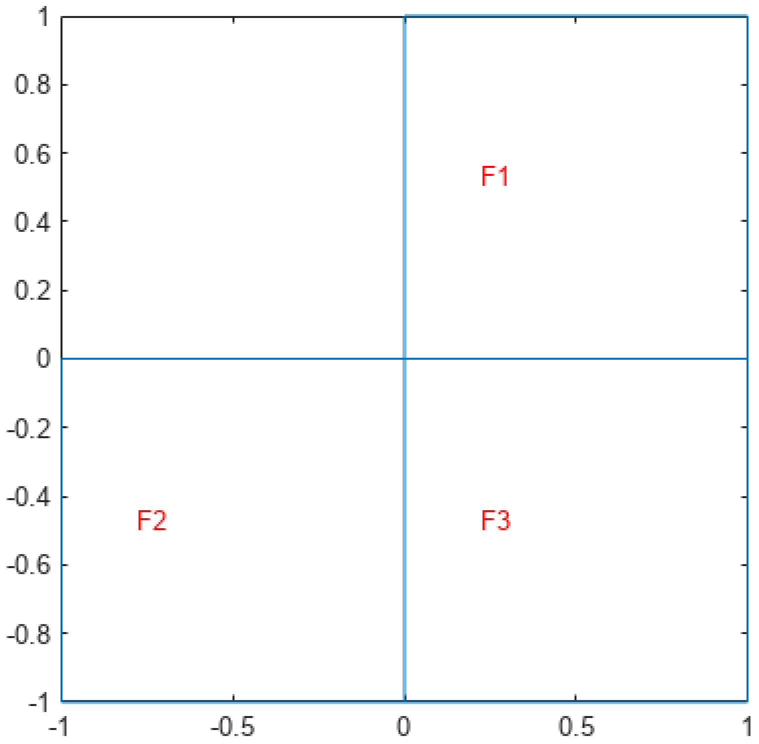
Examples

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.

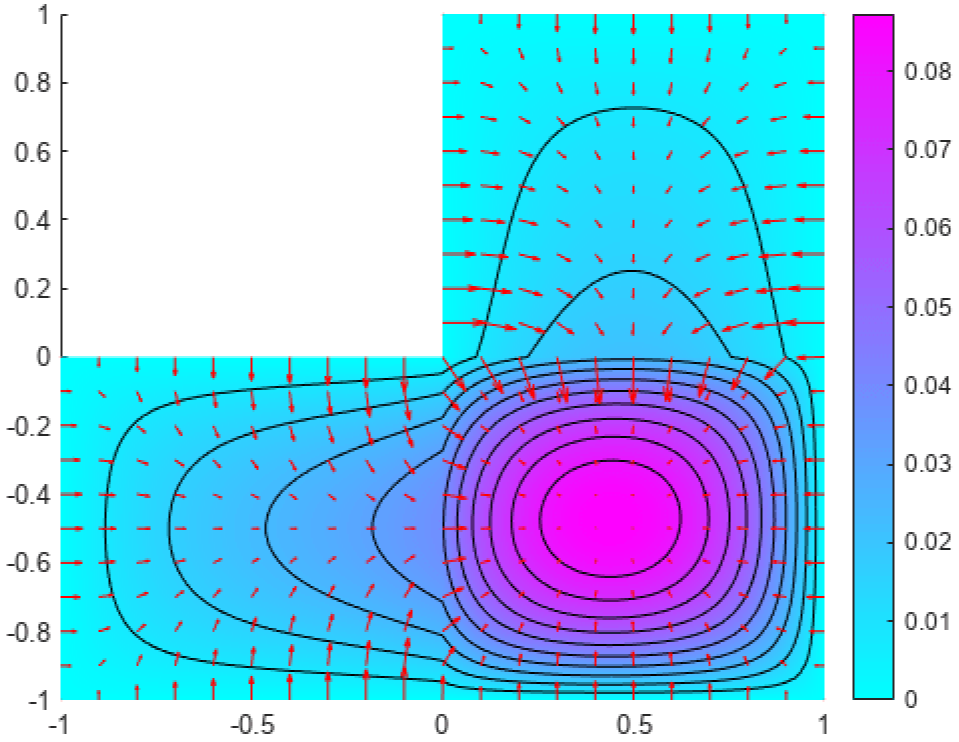
```
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1:model.Geometry.NumEdges, ...
                             "u",0);

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.

```
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.

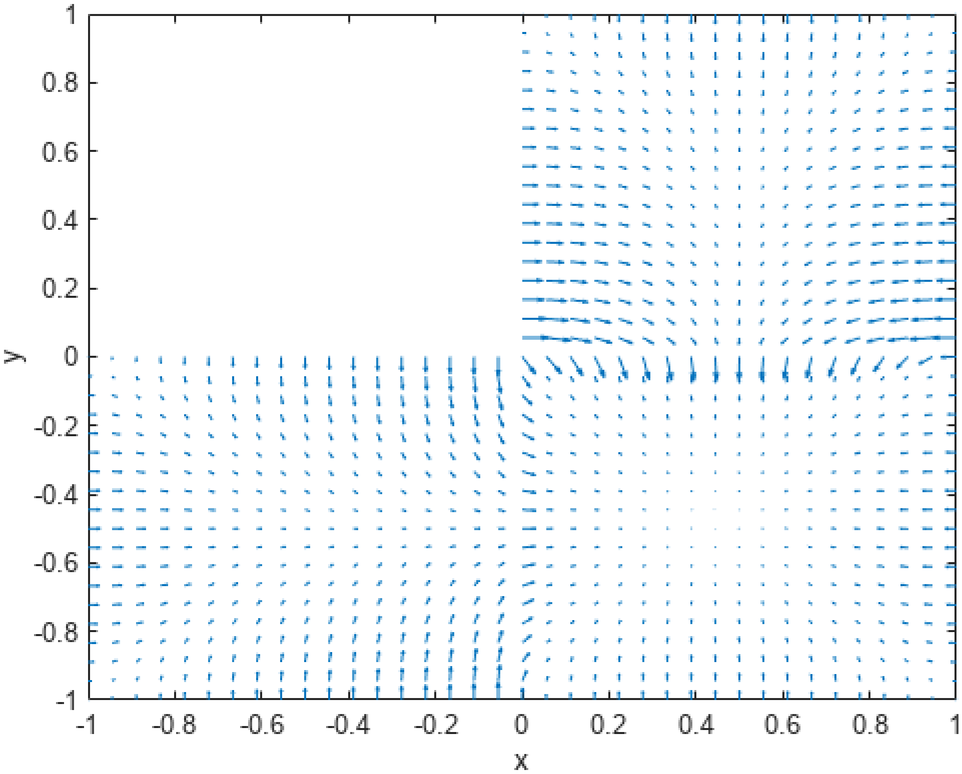
```
v = linspace(-1,1,37);
[X,Y] = meshgrid(v);
querypoints = [X(:),Y(:)]';

[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")
```



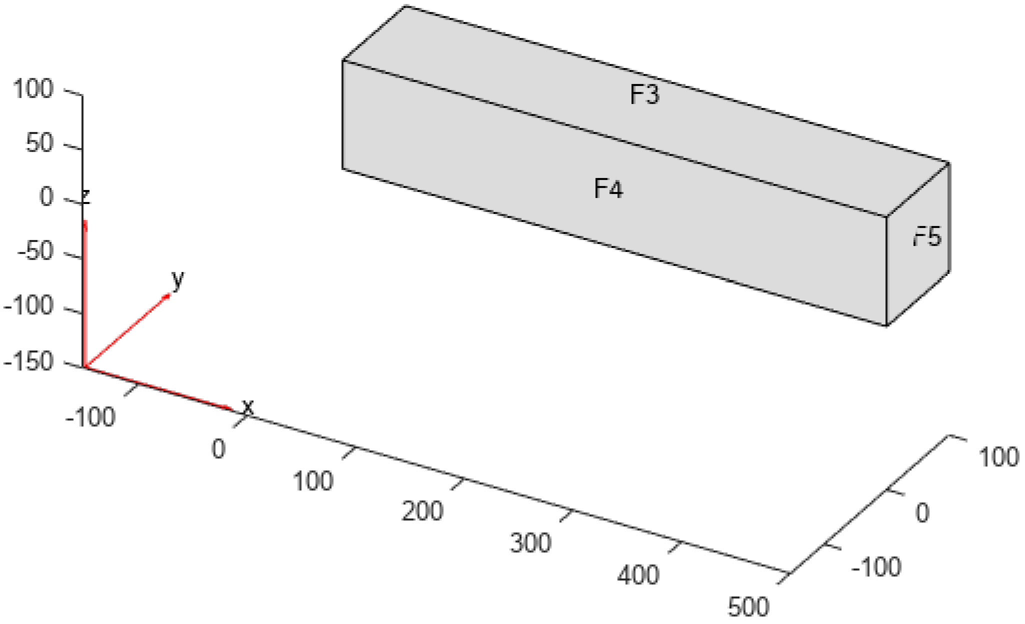
Stress Components in a Cantilever Beam

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.

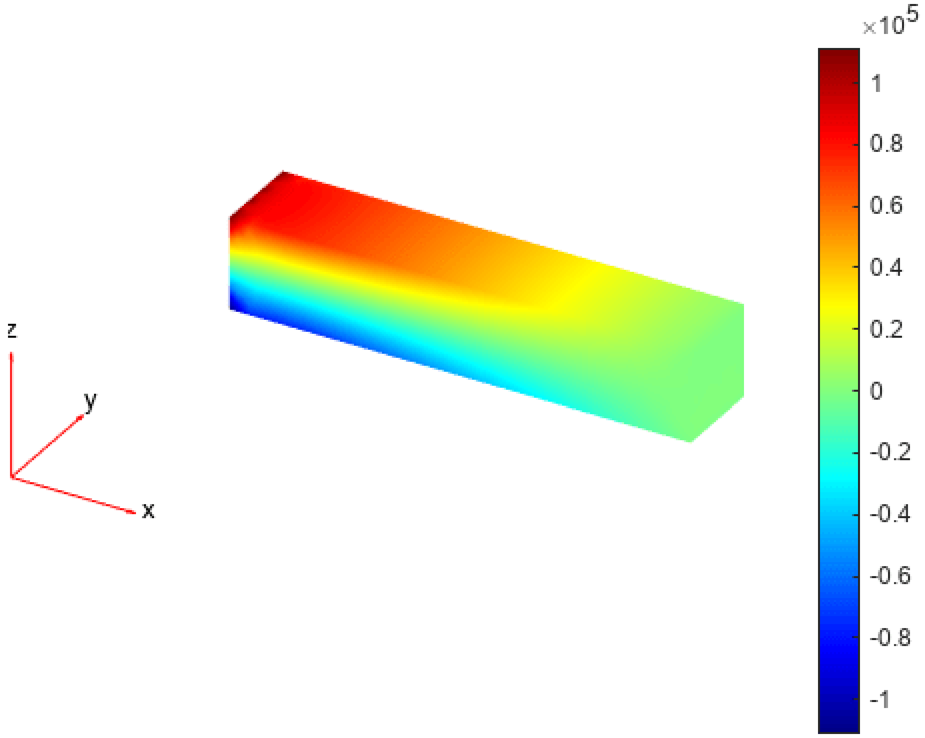
```
generateMesh(model,"Hmax",25,"GeometricOrder","quadratic");
results = solvepde(model);
```

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))
```

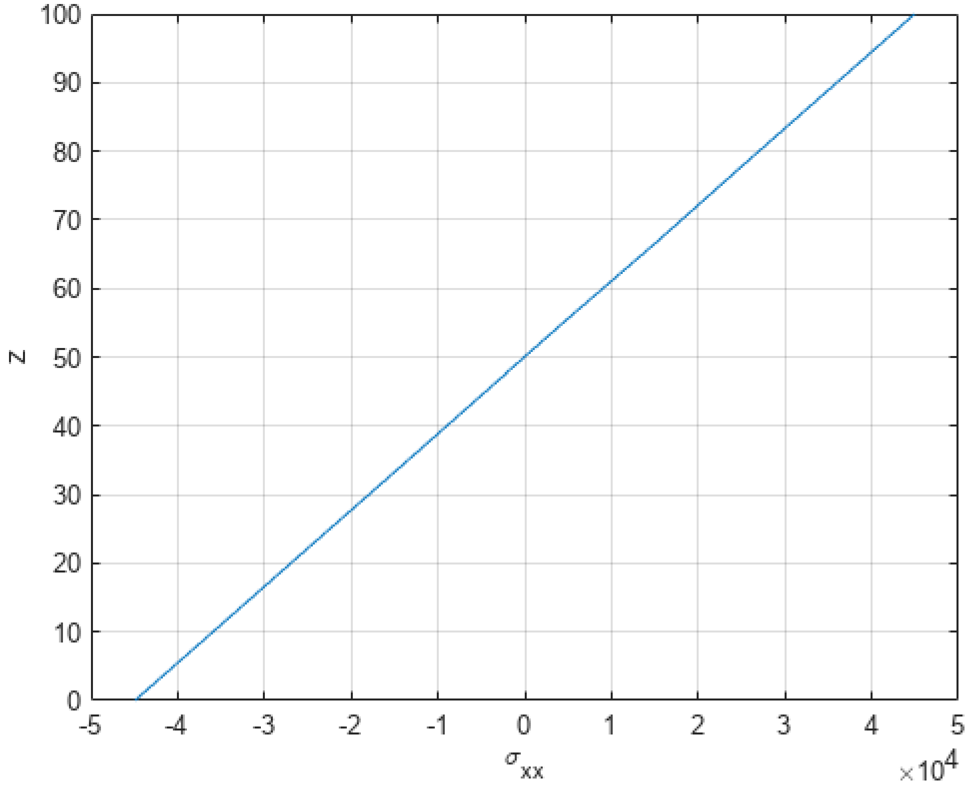


Define a line across the beam from the bottom to the top at mid-span and mid-width. Compute stresses along the line.

```
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")
```



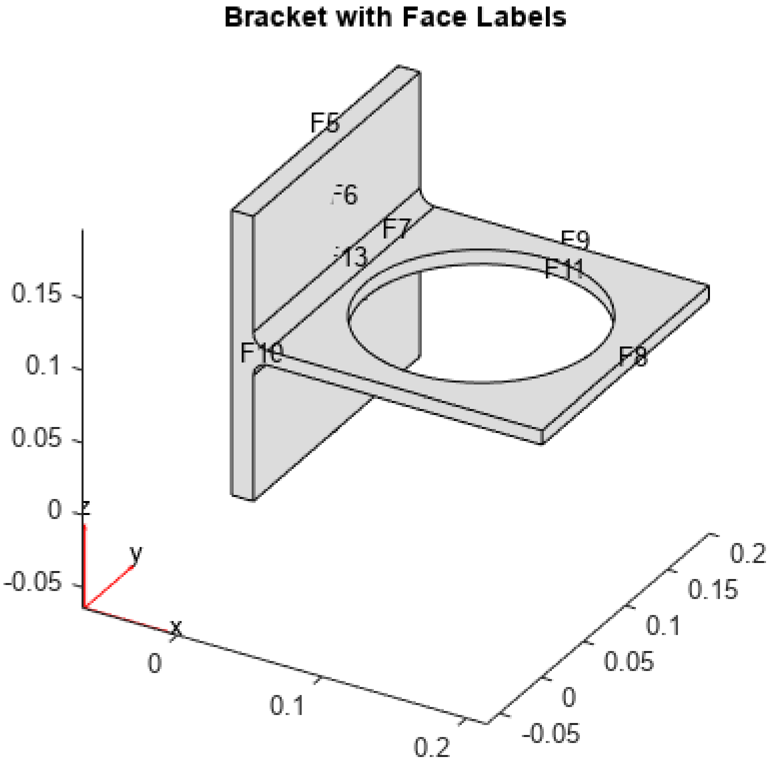
Stress Components in a Bracket

Compute stresses in an idealized 3-D mechanical part under an applied load. First, create a PDE model for this problem.

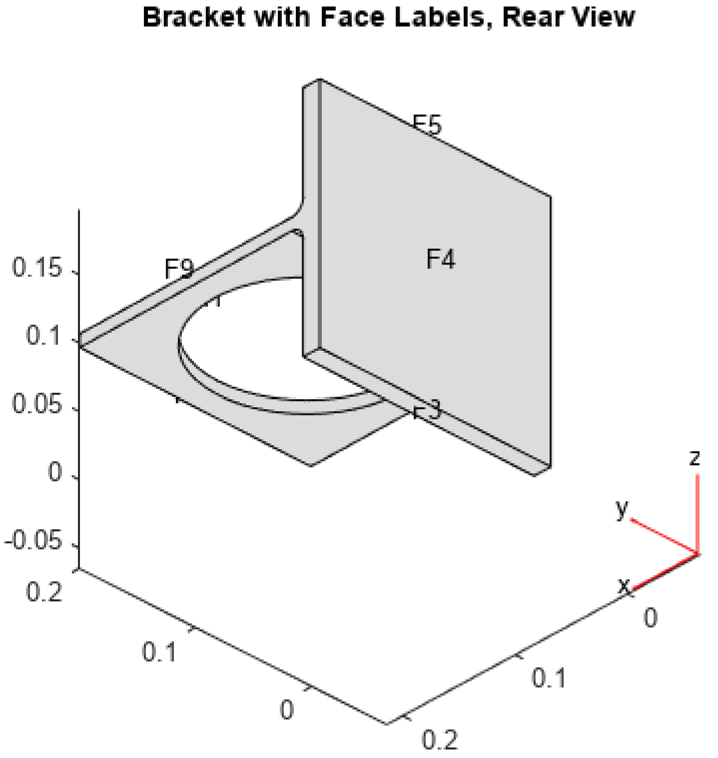
```
N = 3;
model = createpde(N);
```

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")
```



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.

```
% Thickness of horizontal plate with hole, meters
bracketThickness = 1e-2;
% Maximum element length for a moderately fine mesh
hmax = bracketThickness;
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.

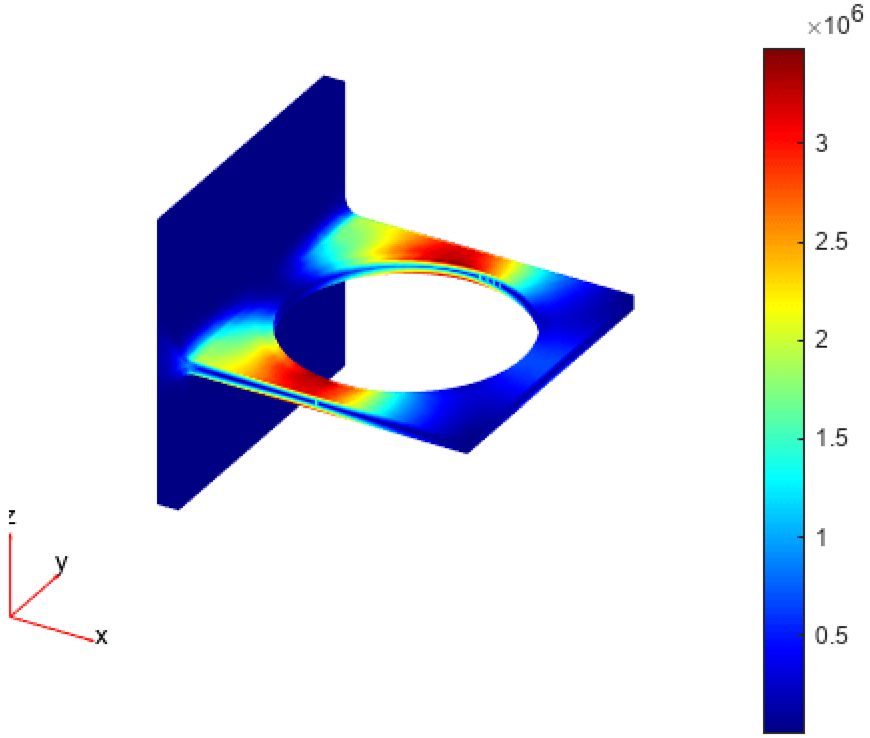
```
sxx = cgradx(:,1);
sxy = cgradx(:,2);
sxz = cgradx(:,3);

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

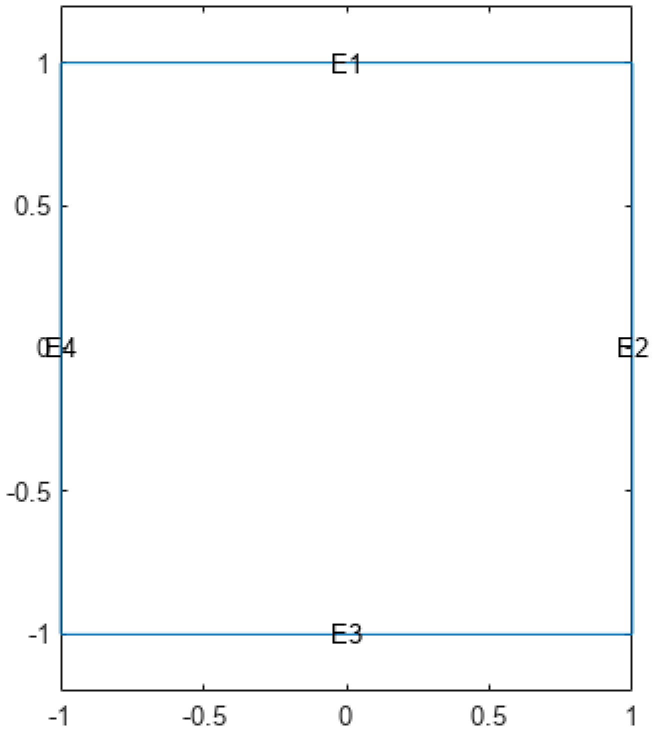
Solve a 2-D transient heat transfer problem on a square domain and compute heat flow across convective boundary.

Create a PDE model for this problem.

```
model = createpde;
```

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.

```
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)`.

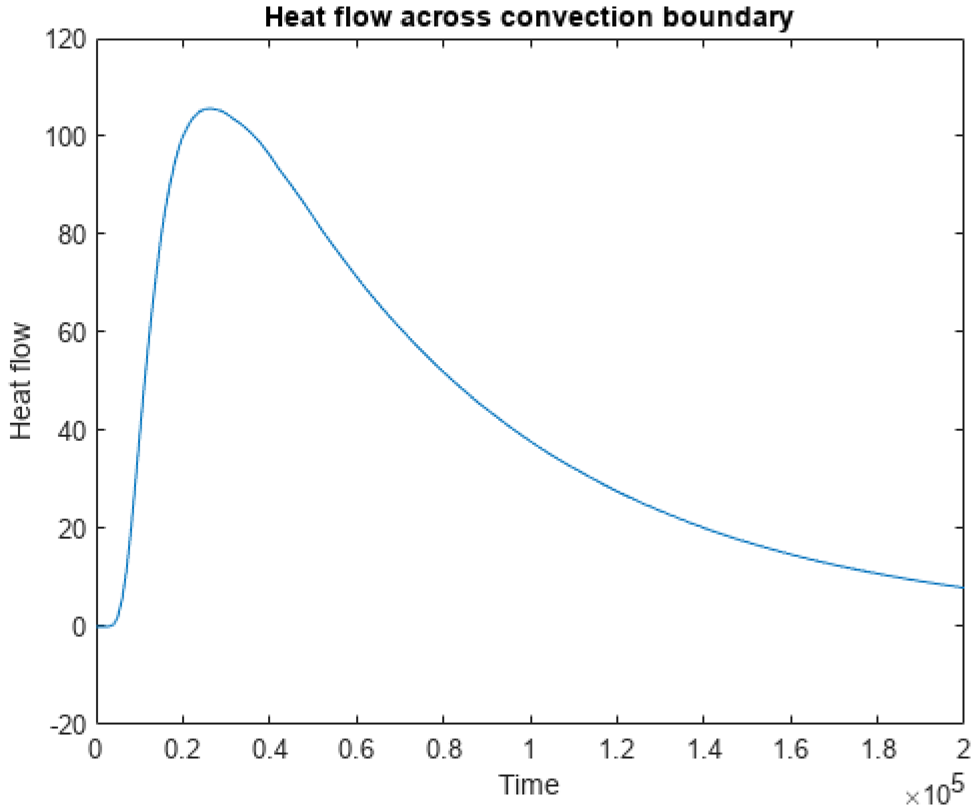
```
[qx,qy] = evaluateCGradient(results,xg,yg,1:length(tlist));
```

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 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.

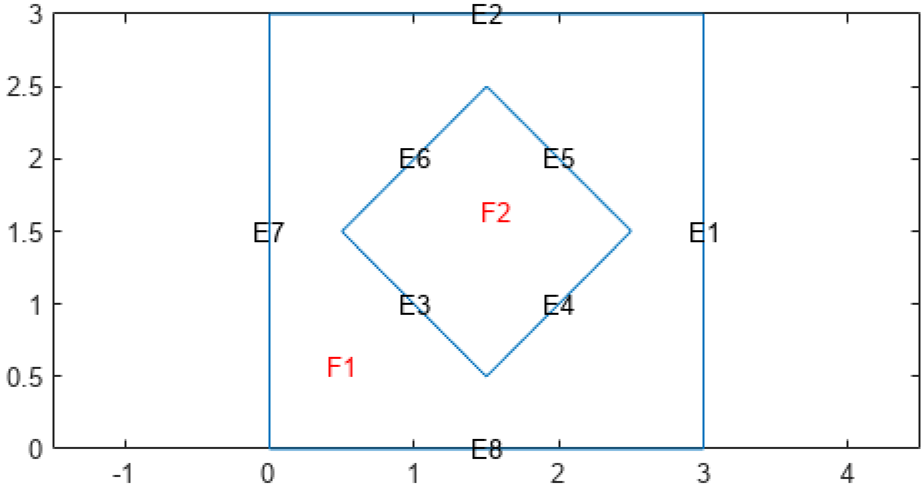
Create a PDE model for this problem.

```
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);
```

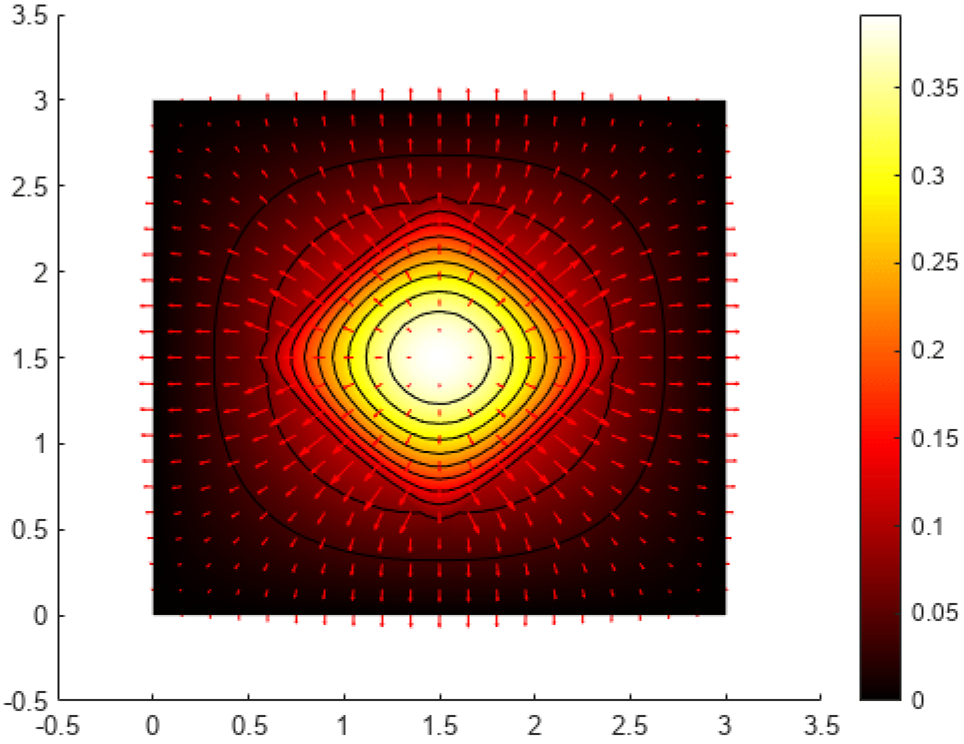
```
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",[1,2,7,8], ...
                             "h",1,"r",0);

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)], ...
              "ColorMap","hot")
```



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 3-356.

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 3-356.

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 3-356.

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.

Version History

Introduced in R2016b

See Also

`PDEModel` | `StationaryResults` | `TimeDependentResults` | `evaluateGradient` | `interpolateSolution`

Topics

“Deflection Analysis of Bracket” on page 3-74
“Dynamics of Damped Cantilever Beam” on page 3-21
“Heat Transfer Between Two Squares Made of Different Materials: PDE Modeler App” on page 3-195

evaluateGradient

Package: `pde`

Evaluate gradients of PDE solutions at arbitrary points

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

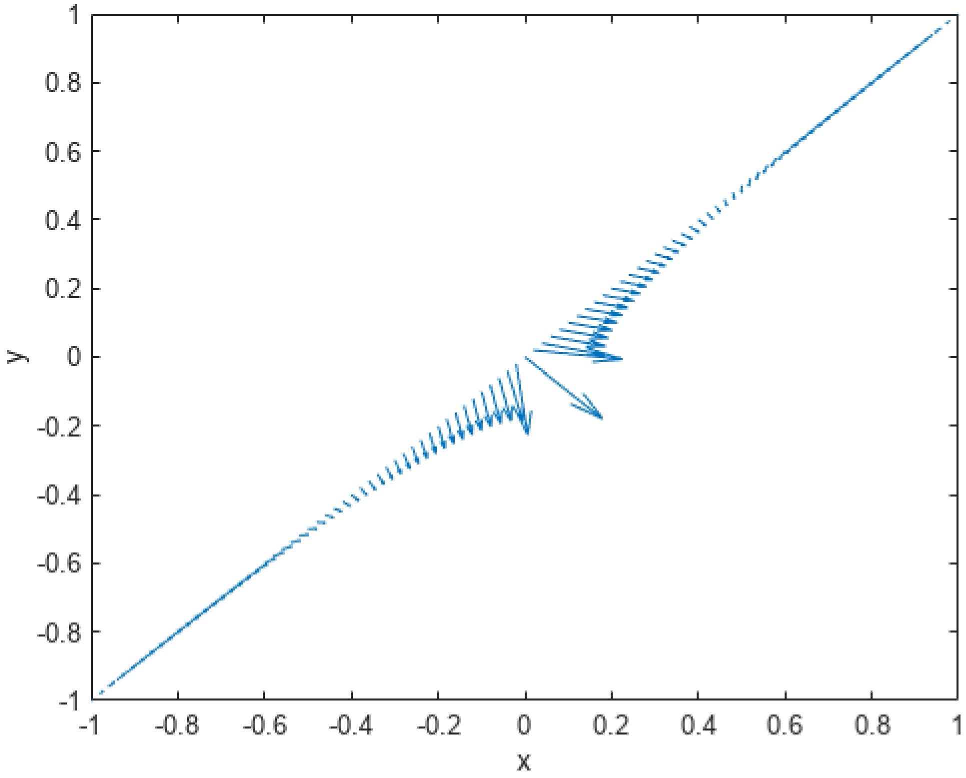
Evaluate gradients of the solution to a scalar elliptic problem along a line. Plot the results.

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.

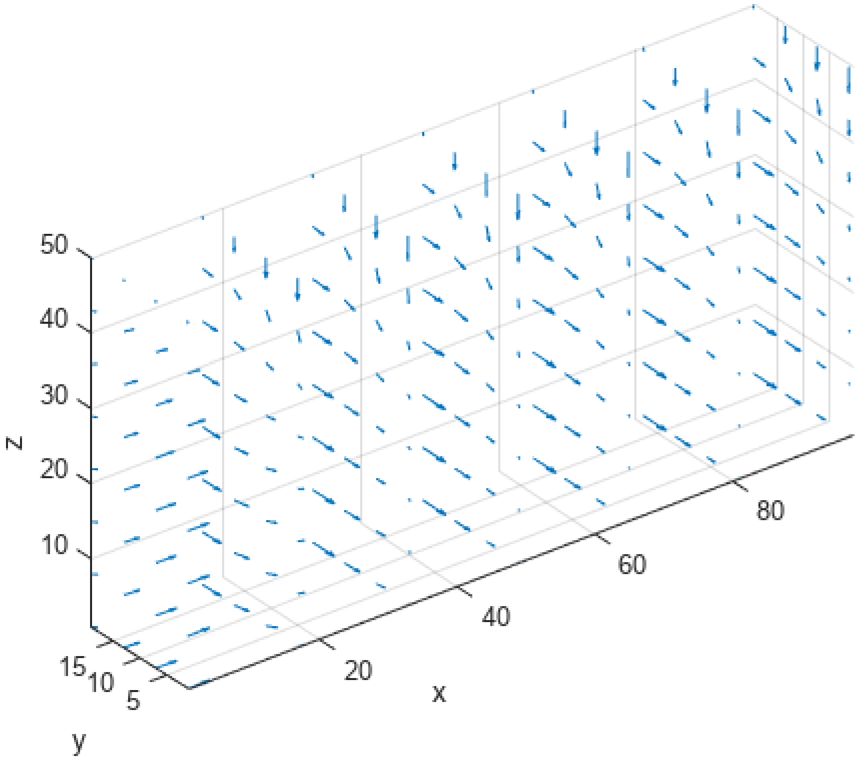
```
model = createpde;
importGeometry(model,"Block.stl");
applyBoundaryCondition(model,"dirichlet", ...
                             "Face",[1,2,5], ...
                             "u",0);
specifyCoefficients(model,"m",0,...
                          "d",0,...
                          "c",1,...
                          "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.

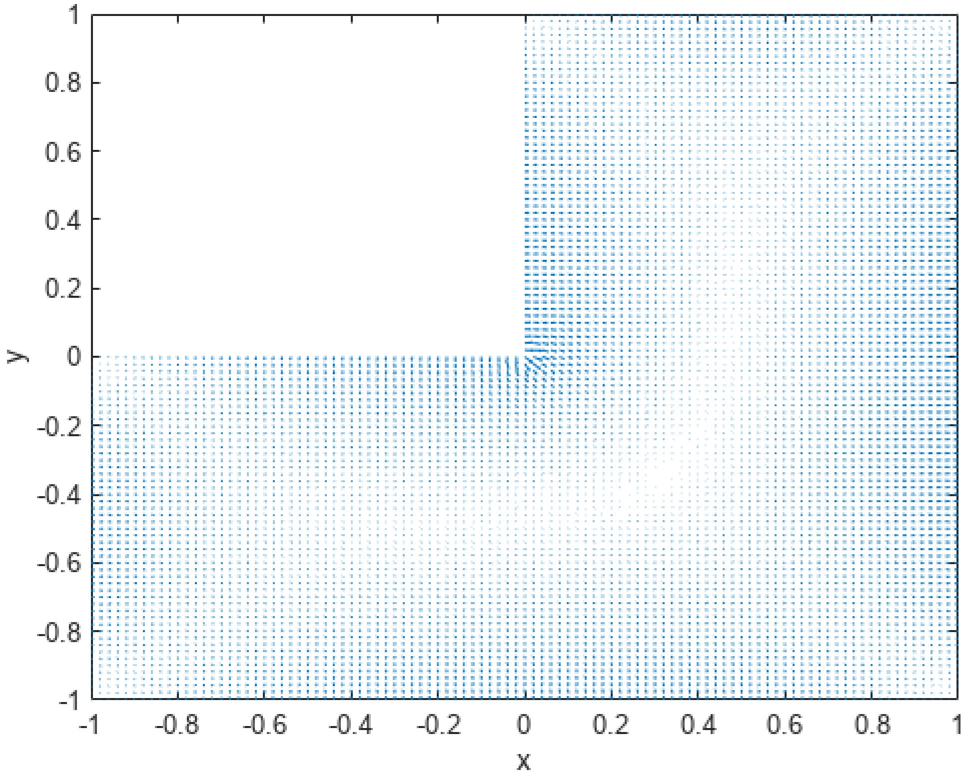
Create the solution to the problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1:model.Geometry.NumEdges, ...
                             "u",0);
```

```
specifyCoefficients(model,"m",0,...
                          "d",0,...
                          "c",1,...
                          "a",0,...
                          "f",1);
generateMesh(model,"Hmax",0.05);
results = solvepde(model);
```

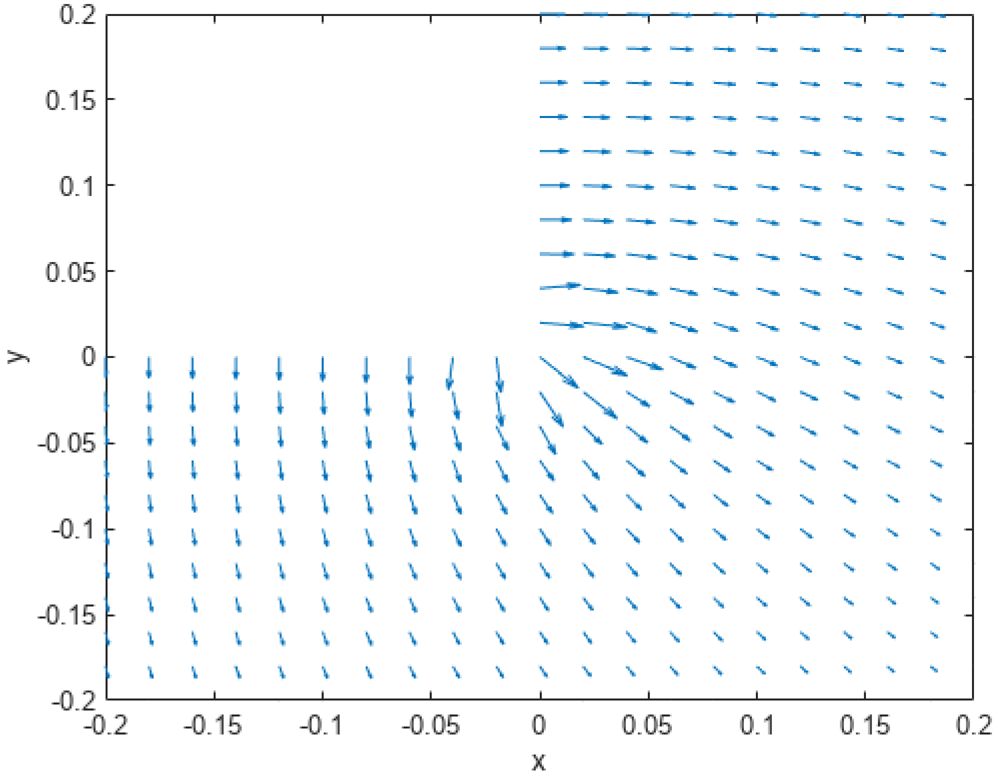
Interpolate gradients of the solution to the grid from -1 to 1 in each direction. Plot the result using the `quiver` plotting function.

```
v = linspace(-1,1,101);
[X,Y] = meshgrid(v);
querypoints = [X(:),Y(:)]';

[gradx,grady] = evaluateGradient(results,querypoints);
quiver(X(:),Y(:),gradx,grady)
xlabel("x")
ylabel("y")
```



Zoom in on a particular part of the plot to see more details. For example, limit the plotting range to 0.2 in each direction.

```
axis([-0.2 0.2 -0.2 0.2])
```



Evaluate Gradients of Solution of Elliptic System

Evaluate gradients of the solution to a two-component elliptic system and plot the results.

Create a PDE model for two components.

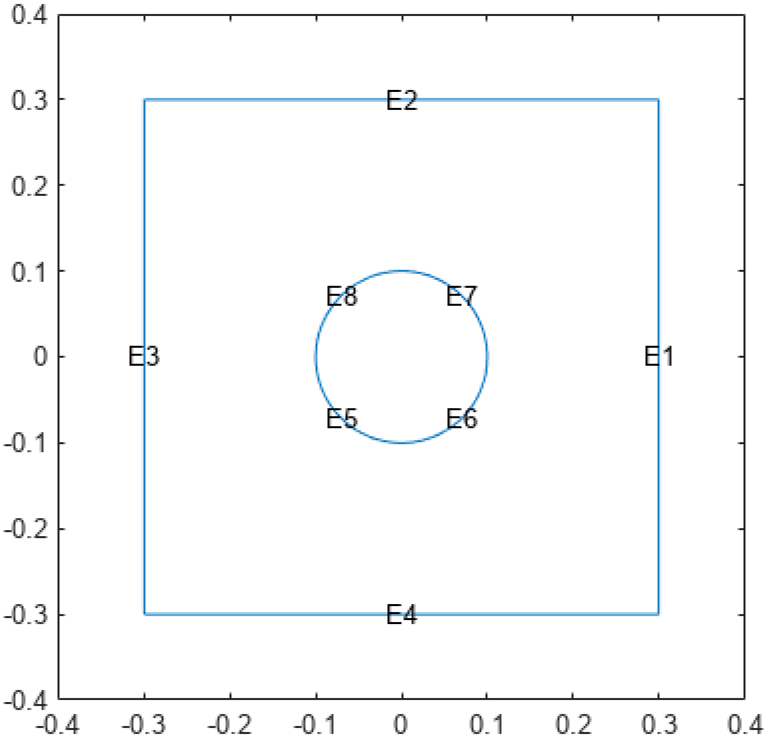
```
model = createpde(2);
```

Create the 2-D geometry as a rectangle with a circular hole in its center. For details about creating the geometry, see the example in “Solve PDEs with Constant Boundary Conditions” on page 2-135.

```
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.

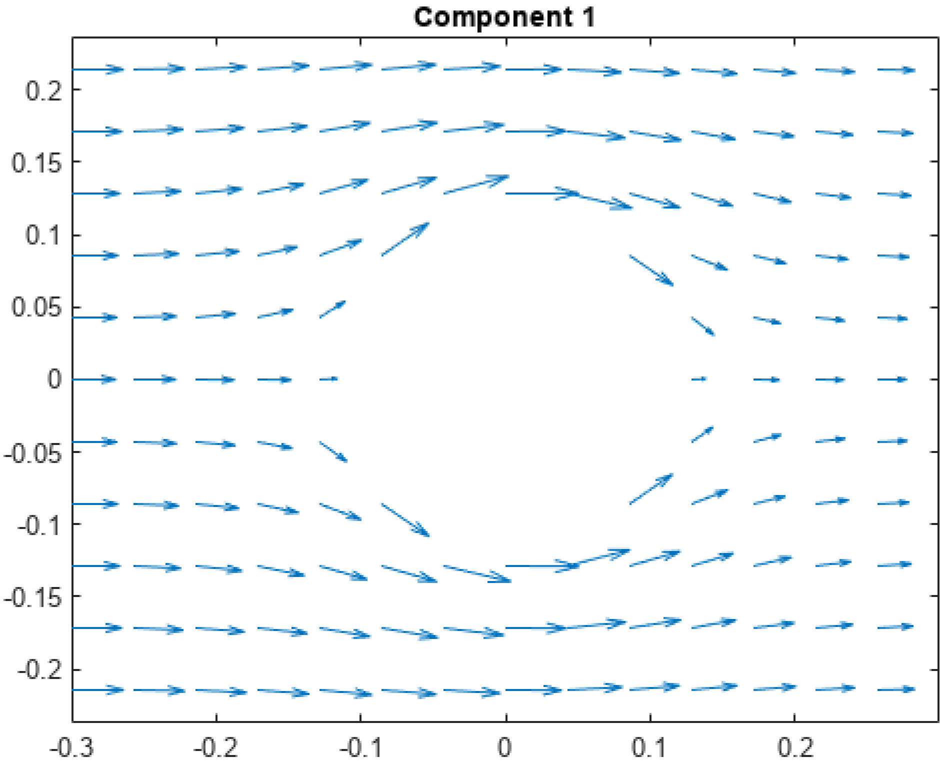
```
generateMesh(model,"Hmax",0.1);
results = solvepde(model);
```

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]);
```

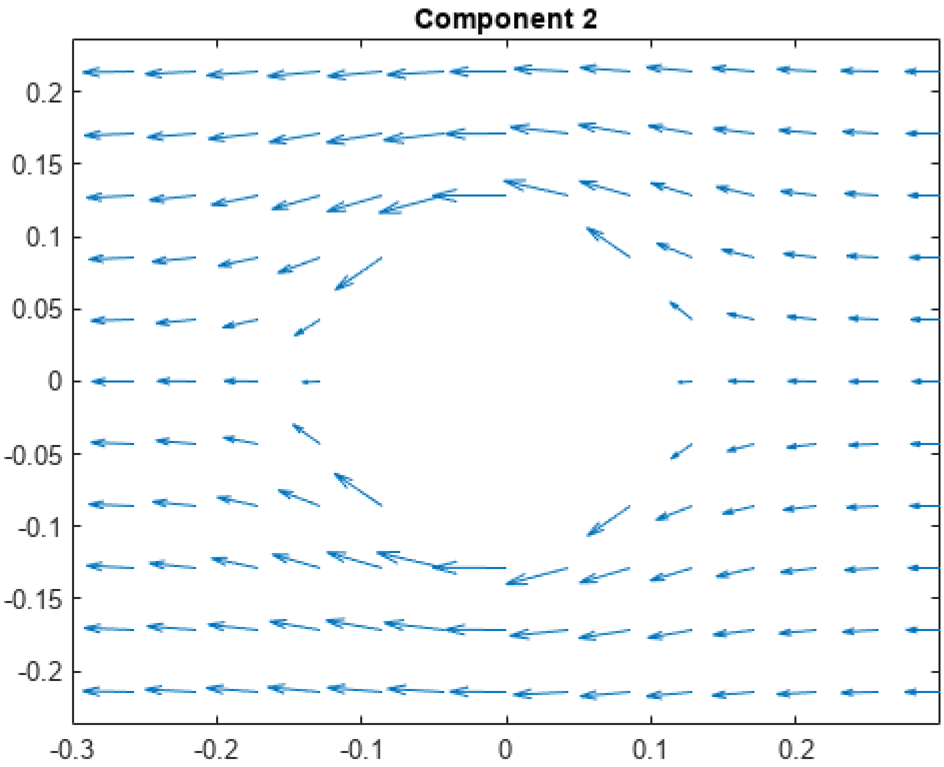
Plot the gradients for the first component.

```
figure
gradx1 = gradx(:,1);
grady1 = grady(:,1);
quiver(X(:),Y(:),gradx1,grady1)
title("Component 1")
axis equal
xlim([-0.3,0.3])
```



Plot the gradients for the second component.

```
figure
gradx2 = gradx(:,2);
grady2 = grady(:,2);
quiver(X(:),Y(:),gradx2,grady2)
title("Component 2")
axis equal
xlim([-0.3,0.3])
```

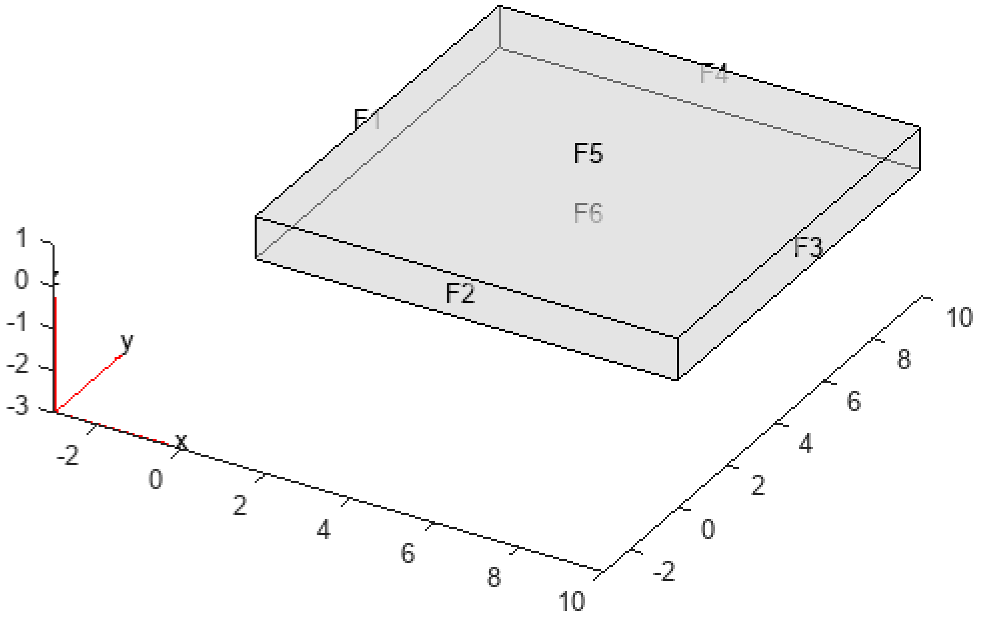


Evaluate Gradients of Solution of Hyperbolic System

Solve a system of hyperbolic PDEs and 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 “Linear Elasticity Equations” on page 3-156.

```
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.

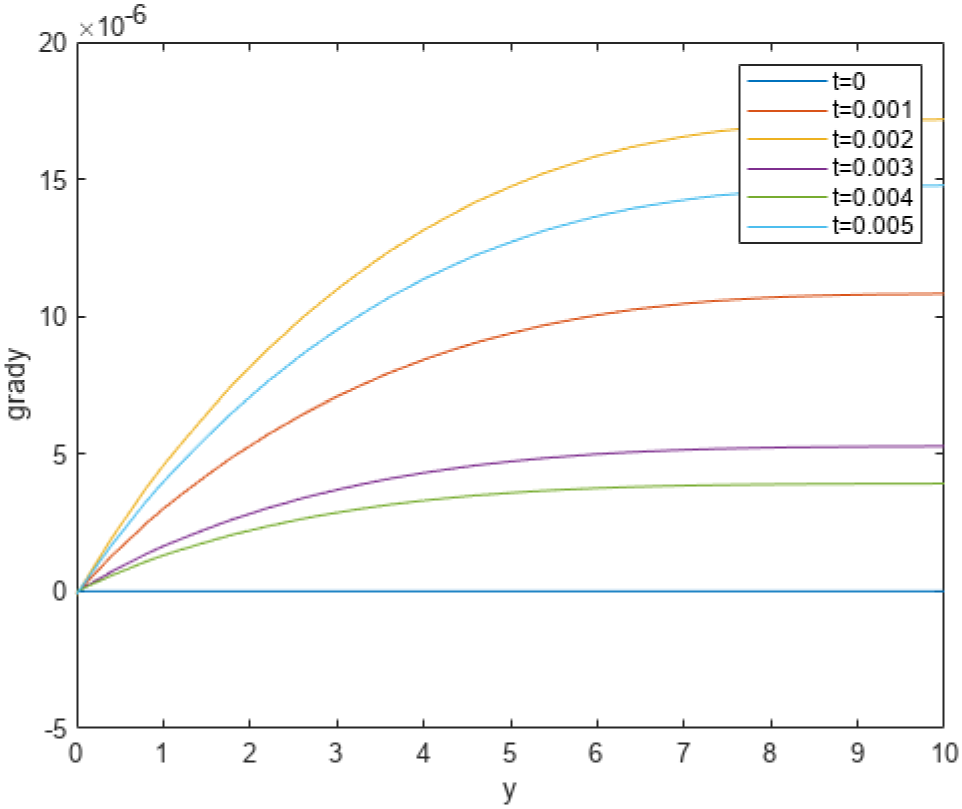
```
yy = 0:0.2:10;
zz = 0.5*ones(size(yy));
xx = 10*zz;
component = 3;
[gradx,grady,gradz] = evaluateGradient(results,xx,yy,zz, ...
                                 component,1:length(tlist));
```

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])
```



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 3-356.

`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 3-356.

`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 3-356.

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`.

Version History

Introduced in R2016a

See Also

`PDEModel` | `StationaryResults` | `TimeDependentResults` | `interpolateSolution` | `evaluateCGradient` | `quiver` | `quiver3` | `contour`

Topics

“Solution and Gradient Plots with pdeplot and pdeplot3D” on page 3-329
“3-D Solution and Gradient Plots with MATLAB Functions” on page 3-344

"Dimensions of Solutions, Gradients, and Fluxes" on page 3-356

evaluateHeatFlux

Package: pde

Evaluate heat flux of a thermal solution at nodal or arbitrary spatial locations

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

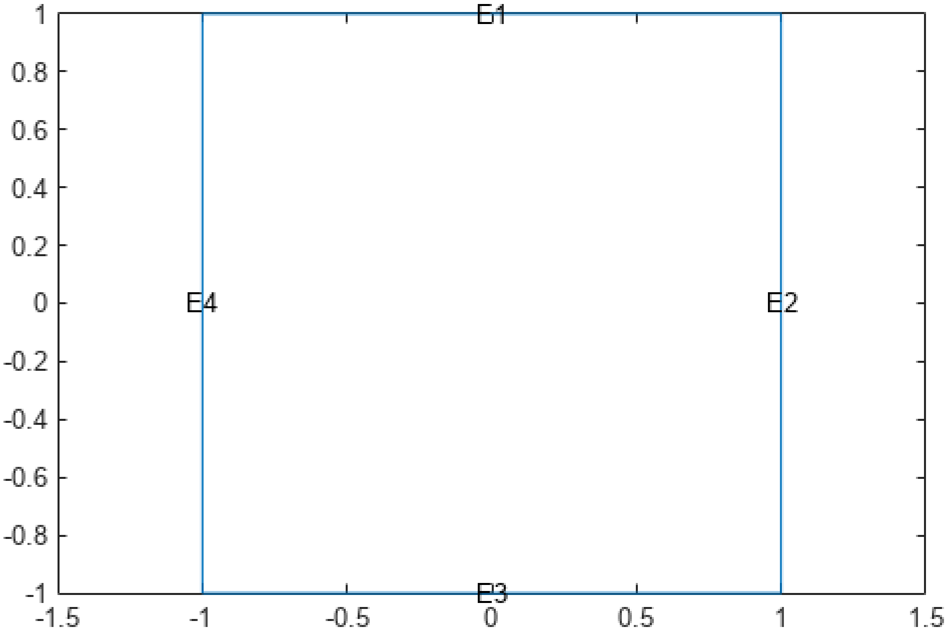
For a 2-D steady-state thermal model, evaluate heat flux at the nodal locations and at the points specified by `x` and `y` coordinates.

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);
```

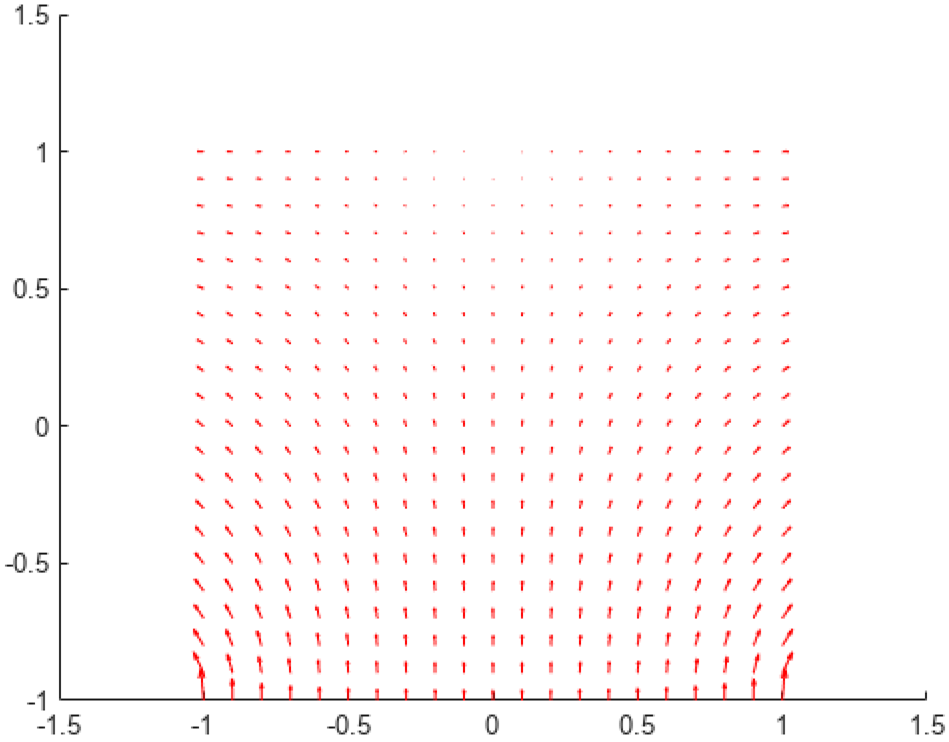
Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
results = solve(thermalmodel)
```

```
results = 
  SteadyStateThermalResults with properties:

    Temperature: [1541x1 double]
     XGradients: [1541x1 double]
     YGradients: [1541x1 double]
     ZGradients: []
           Mesh: [1x1 FEMesh]
```

Evaluate heat flux at the nodal locations.

```
[qx,qy] = evaluateHeatFlux(results);

figure
pdeplot(thermalmodel,"FlowData",[qx qy])
```



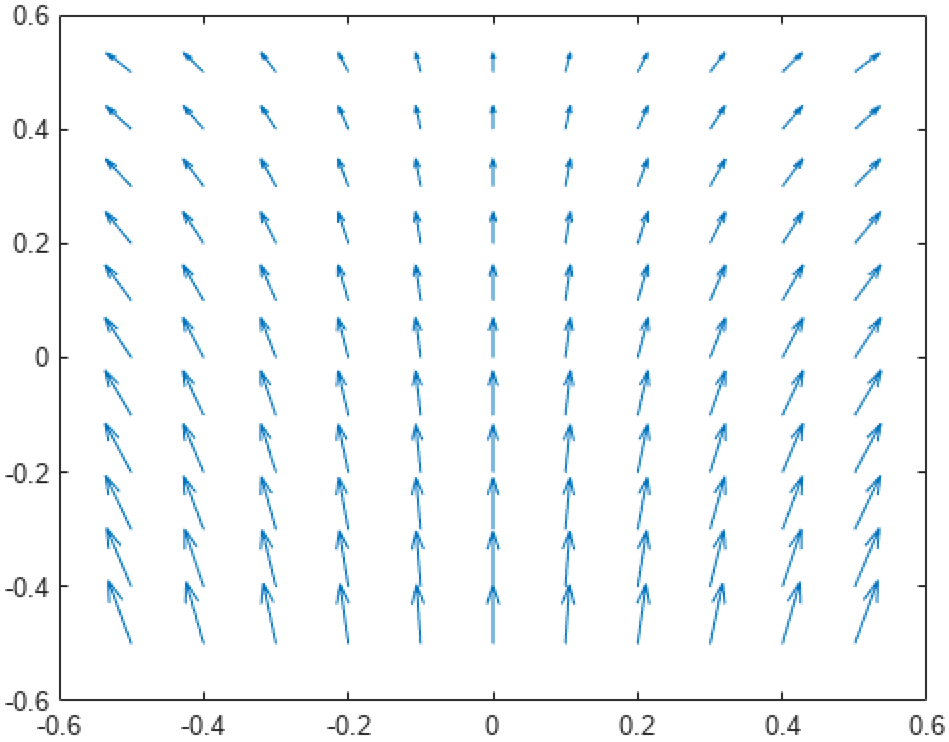
Create a grid specified by `x` and `y` coordinates, and evaluate heat flux to the grid.

```
v = linspace(-0.5,0.5,11);
[X,Y] = meshgrid(v);
```

```
[qx,qy] = evaluateHeatFlux(results,X,Y);
```

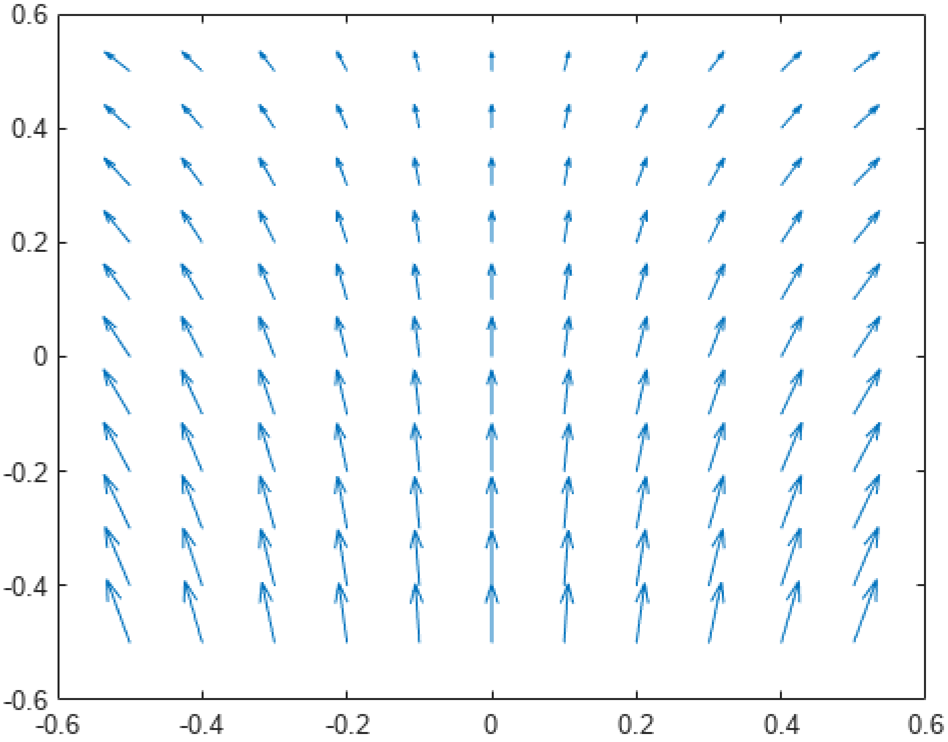
Reshape the `qTx` and `qTy` vectors, and plot the resulting heat flux.

```
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)
```



Heat Flux for 3-D Steady-State Thermal Model

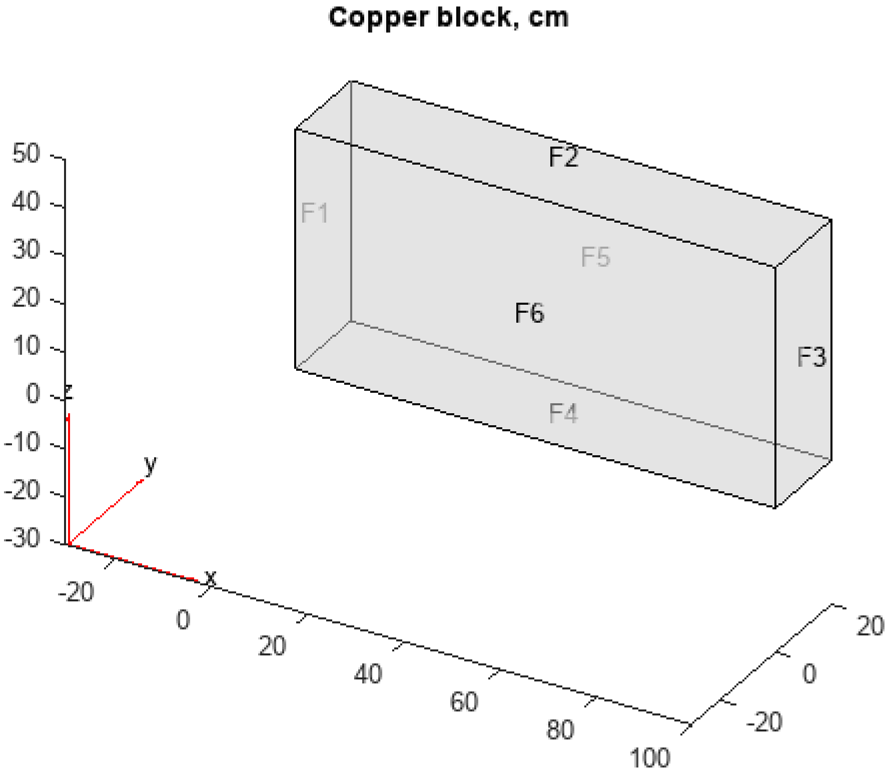
For a 3-D steady-state thermal model, evaluate heat flux at the nodal locations and at the points specified by `x`, `y`, and `z` coordinates.

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
```



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.

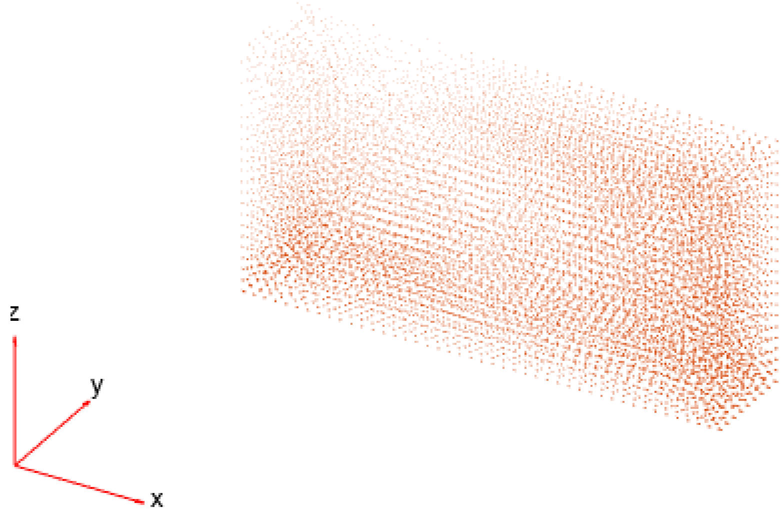
```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)
```

```
thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [12691x1 double]
     XGradients: [12691x1 double]
     YGradients: [12691x1 double]
     ZGradients: [12691x1 double]
```

```
            Mesh: [1x1 FEMesh]
```

Evaluate heat flux at the nodal locations.

```
[qx,qy,qz] = evaluateHeatFlux(thermalresults);

figure
pdeplot3D(thermalmodel,"FlowData",[qx qy qz])
```

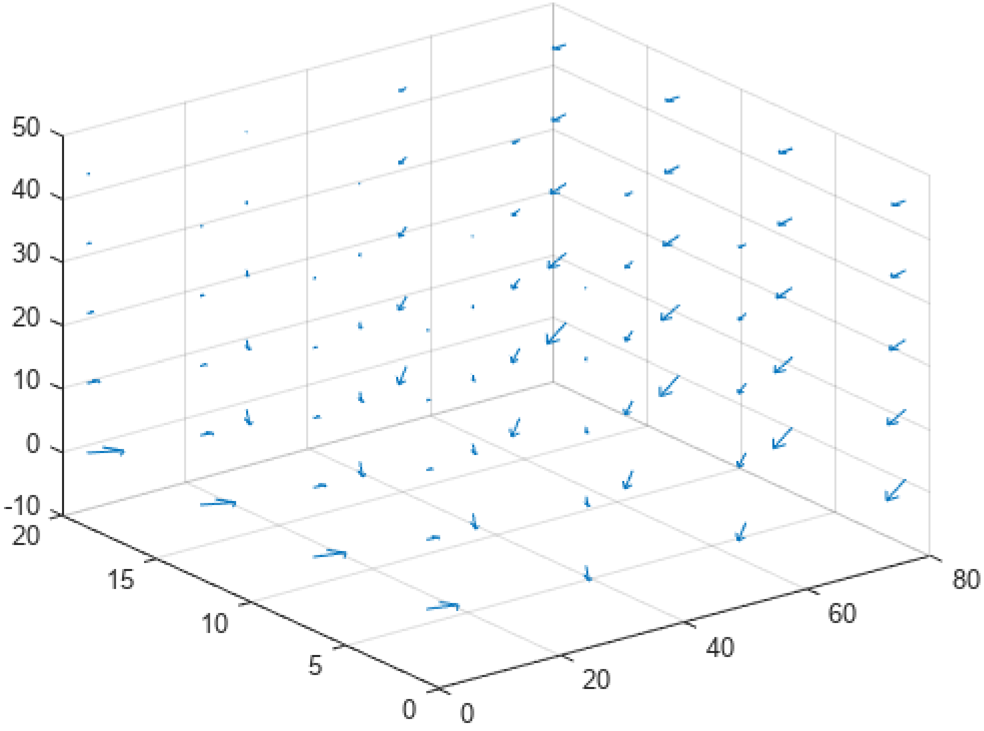


Create a grid specified by `x`, `y`, and `z` coordinates, and evaluate heat flux to the grid.

```
[X,Y,Z] = meshgrid(1:26:100,1:6:20,1:11:50);

[qx,qy,qz] = evaluateHeatFlux(thermalresults,X,Y,Z);
```

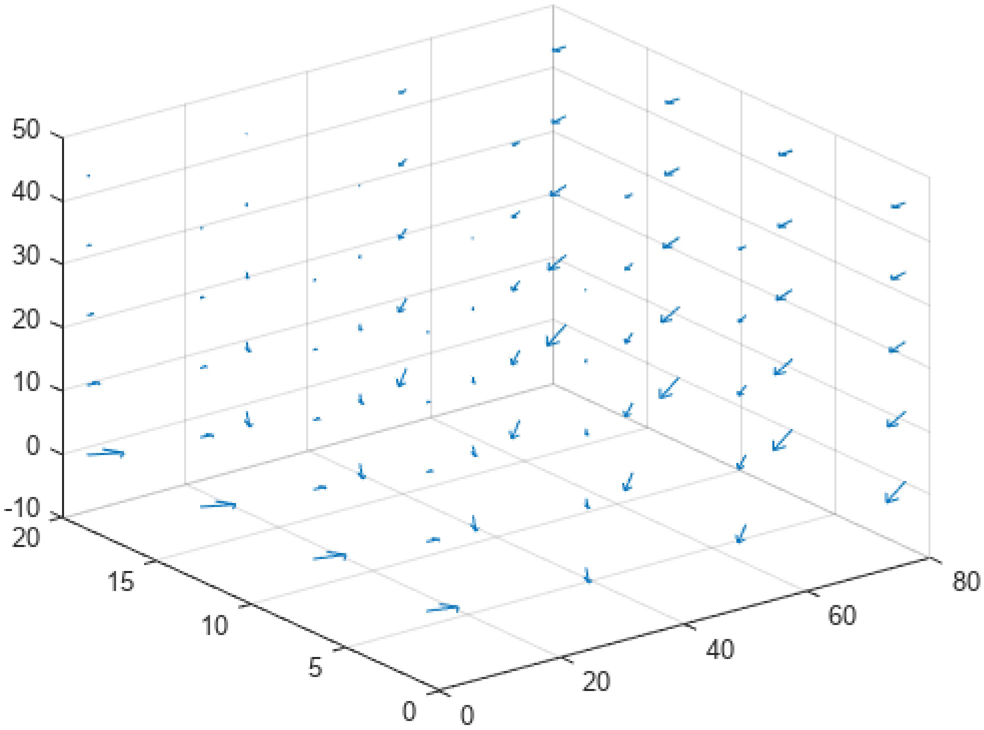
Reshape the `qx`, `qy`, and `qz` vectors, and plot the resulting heat flux.

```
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

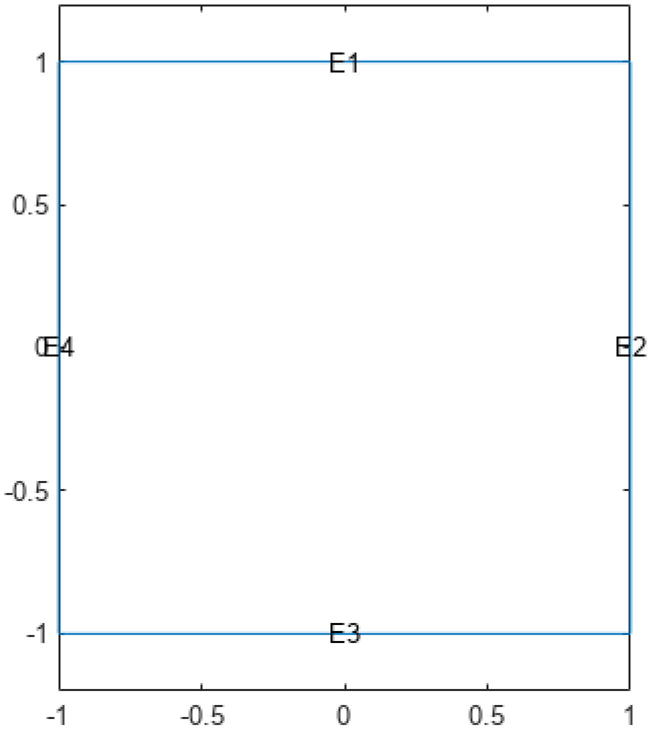
Solve a 2-D transient heat transfer problem on a square domain, and compute heat flow across a convective boundary.

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 $W/(m^{\circ}C)$, mass density is 7800 kg/m^3, and specific heat is 500 $J/(kg^{\circ}C)$.

```
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);
```

Create a grid specified by `x` and `y` coordinates, and evaluate heat flux to the grid.

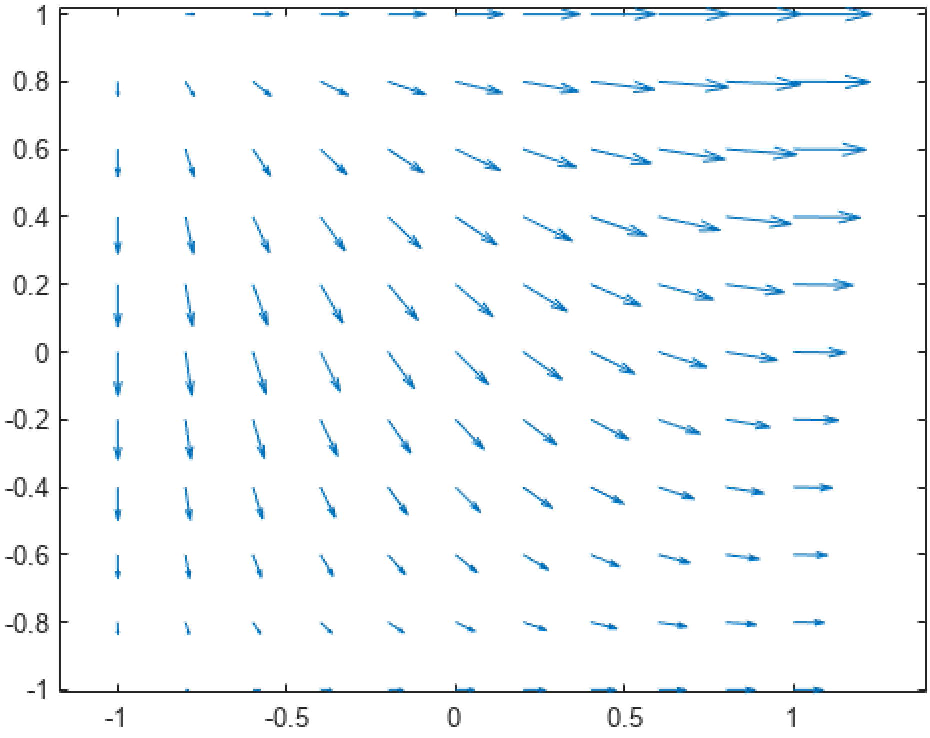
```
v = linspace(-1,1,11);
[X,Y] = meshgrid(v);

[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)
```

```
ans = 24000
```

```
figure
quiver(X(:),Y(:),qx(:,25),qy(:,25));
xlim([-1,1])
axis equal
```



Heat Flux for Transient Thermal Model on Two Squares Made of Different Materials

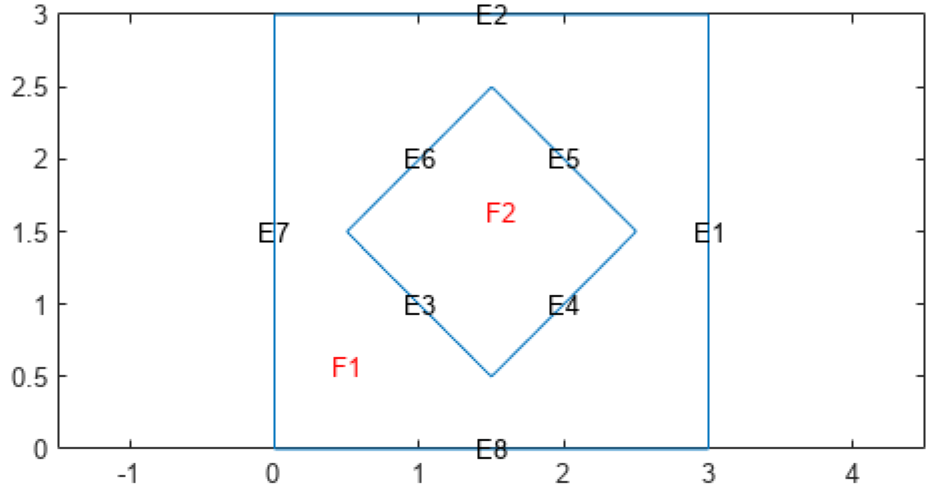
Solve the heat transfer problem for the following 2-D geometry consisting of a square and a diamond made of different materials. Compute the heat flux, and plot it as a vector field.

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 $10\,W/(m^{\circ}C)$, mass density is $2\,kg/m^3$, and specific heat is $0.1\,J/(kg^{\circ}C)$.

```
thermalProperties(thermalmodel,"ThermalConductivity",10, ...
                               "MassDensity",2, ...
                               "SpecificHeat",0.1, ...
                               "Face",1);
```

For the diamond-shaped region, assign the following thermal properties: thermal conductivity is $2\,W/(m^{\circ}C)$, mass density is $1\,kg/m^3$, and 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 and solve the problem.

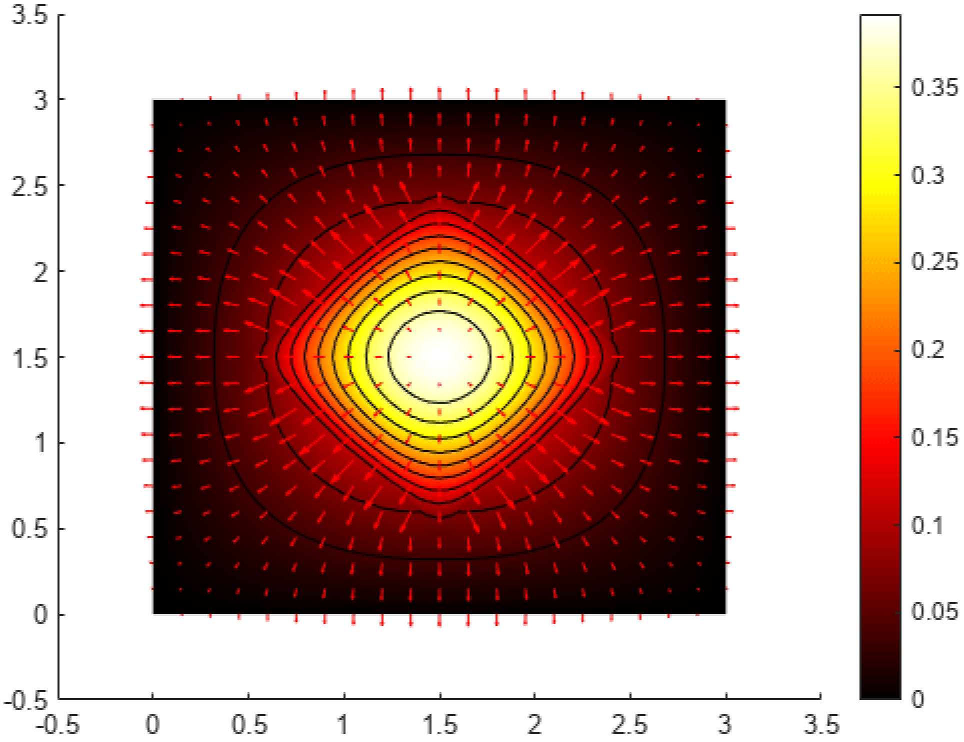
```
generateMesh(thermalmodel);
```

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`.

Version History

Introduced in R2017a

See Also

`ThermalModel` | `SteadyStateThermalResults` | `TransientThermalResults` | `evaluateHeatRate` | `evaluateTemperatureGradient` | `interpolateTemperature`

evaluateHeatRate

Package: `pde`

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

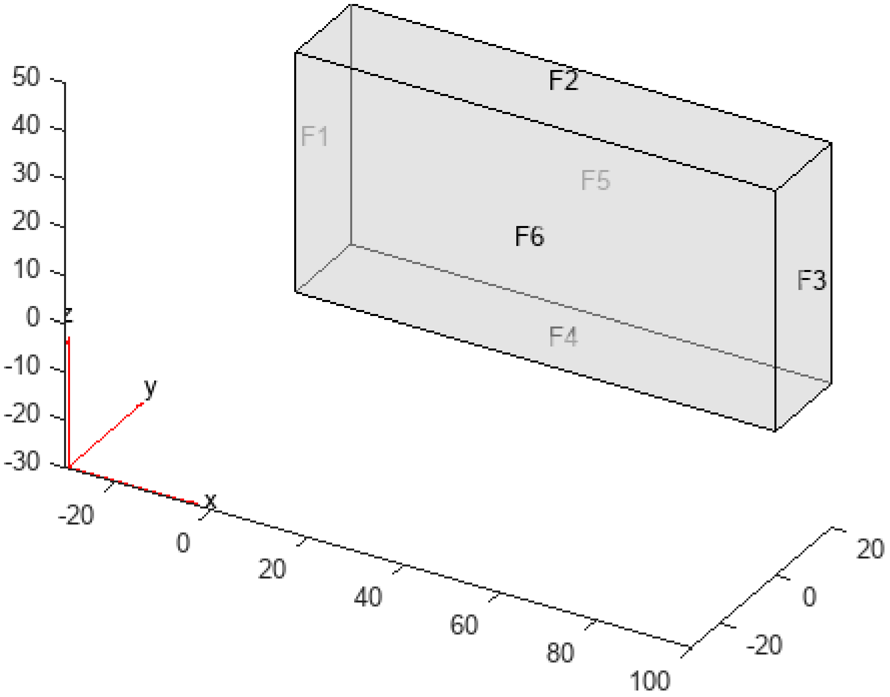
Compute the heat flow rate across a face of the block geometry.

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.

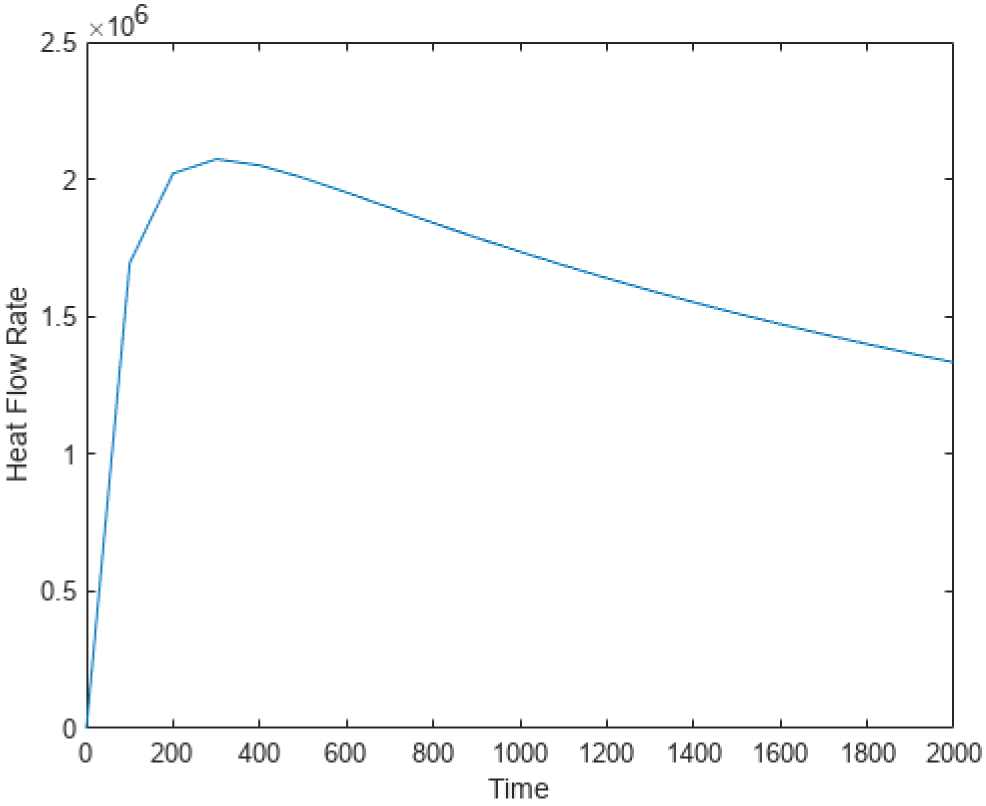
```
thermalIC(thermalmodel,800);
```

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` | `string`

`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.

Version History

Introduced in R2017a

See Also

`ThermalModel` | `SteadyStateThermalResults` | `TransientThermalResults` | `evaluateHeatFlux` | `evaluateTemperatureGradient` | `interpolateTemperature`

evaluatePrincipalStrain

Package: `pde`

Evaluate principal strain at nodal locations

Syntax

`pStrain = evaluatePrincipalStrain(structuralresults)`

Description

`pStrain = evaluatePrincipalStrain(structuralresults)` evaluates principal strain at nodal locations using strain values from `structuralresults`. For transient and frequency response structural models, `evaluatePrincipalStrain` evaluates principal strain for all time- or frequency-steps, respectively.

Examples

Octahedral Shear Strain for Bimetallic Cable Under Tension

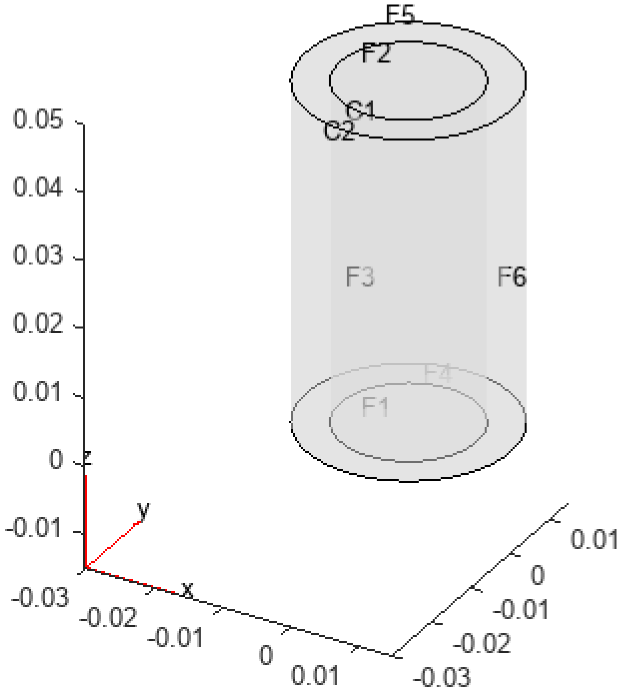
Solve a static structural model representing a bimetallic cable under tension, and compute octahedral shear strain.

Create a structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01,0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                             "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                             "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

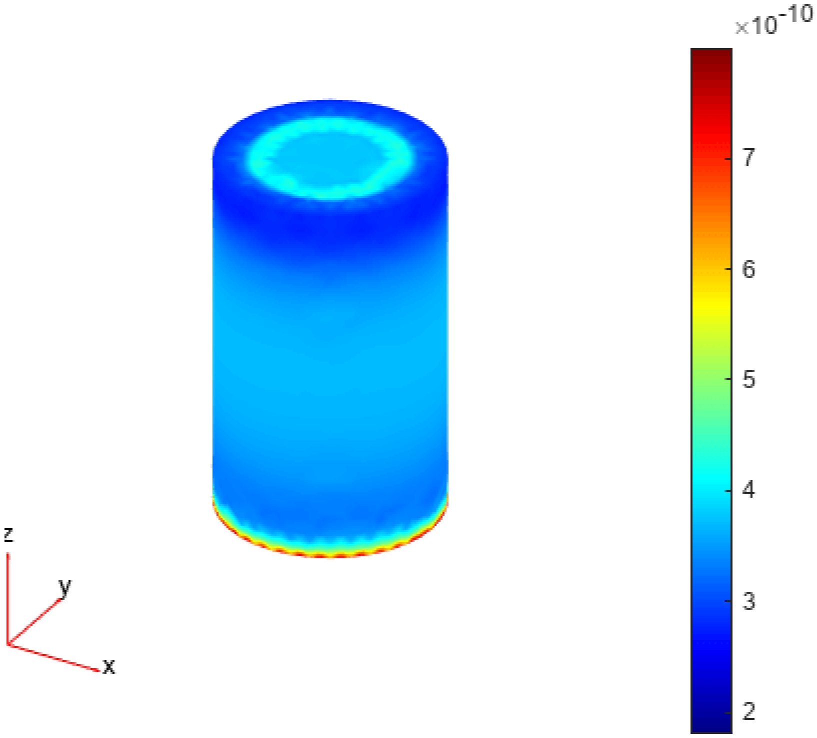
```
            Mesh: [1x1 FEMesh]
```

Evaluate the principal strain at nodal locations.

```
pStrain = evaluatePrincipalStrain(structuralresults);
```

Use the principal strain to evaluate the first and second invariant of strain.

```
I1 = pStrain.e1 + pStrain.e2 + pStrain.e3;
I2 = pStrain.e1.*pStrain.e2 + ...
     pStrain.e2.*pStrain.e3 + ...
     pStrain.e3.*pStrain.e1;
tauOct = sqrt(2*(I1.^2 -3*I2))/3;
pdeplot3D(structuralmodel,"ColorMapData",tauOct)
```



Principal Strain for 3-D Structural Dynamic Problem

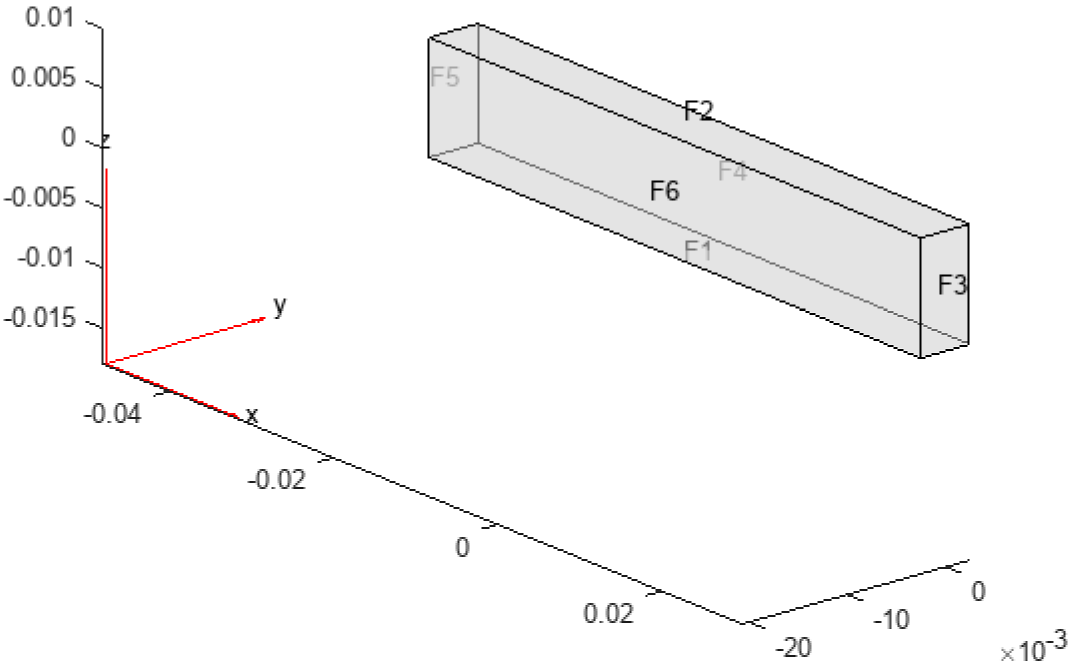
Evaluate the principal strain and octahedral shear strain in a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3,...
                             "YDisplacement",1E-4,...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Evaluate the principal strain in the beam.

```
pStrain = evaluatePrincipalStrain(structuralresults);
```

Use the principal strain to evaluate the first and second invariants.

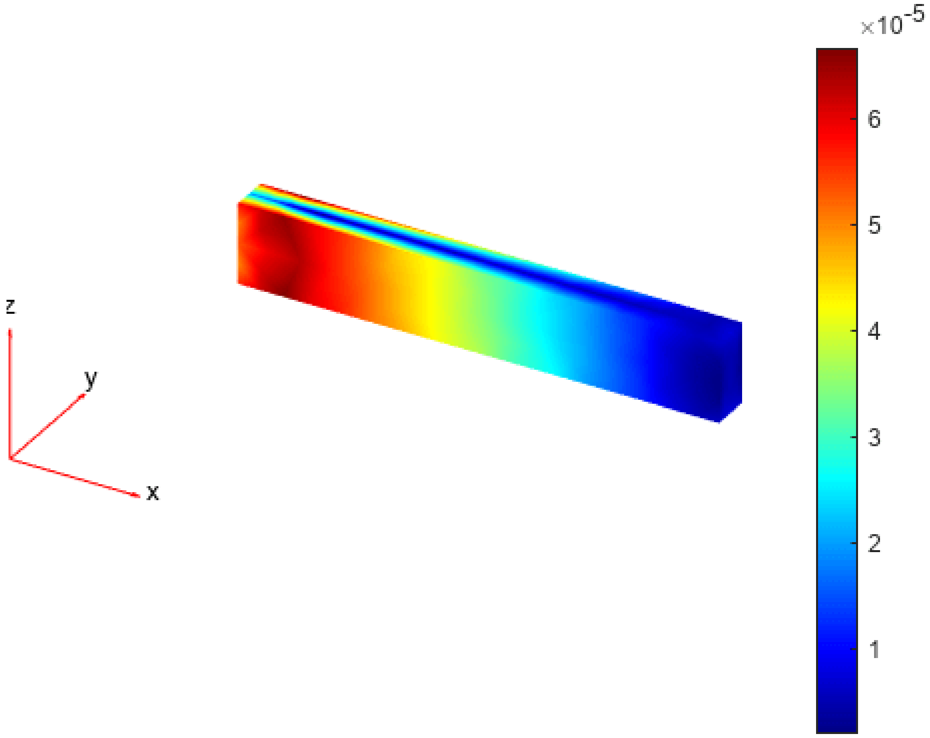
```
I1 = pStrain.e1 + pStrain.e2 + pStrain.e3;
I2 = pStrain.e1.*pStrain.e2 + ...
     pStrain.e2.*pStrain.e3 + ...
     pStrain.e3.*pStrain.e1;
```

Use the stress invariants to compute the octahedral shear strain.

```
tauOct = sqrt(2*(I1.^2 -3*I2))/3;
```

Plot the results.

```
figure
pdeplot3D(structuralmodel,"ColorMapData",tauOct(:,end))
```



Input Arguments

`structuralresults` — Solution of structural analysis problem
`StaticStructuralResults` object | `TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the structural analysis problem, specified as a `StaticStructuralResults`, `TransientStructuralResults`, or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel)`

Output Arguments

`pStrain` — Principal strain at nodal locations
structure array

Principal strain at the nodal locations, returned as a structure array.

Version History

Introduced in R2017b

Support for frequency response structural problems

For frequency response structural models, `evaluatePrincipalStrain` evaluates principal strain for all frequency-steps.

Support for transient structural problems

For transient structural models, `evaluatePrincipalStrain` evaluates principal strain for all time-steps.

See Also

`StructuralModel` | `StaticStructuralResults` | `interpolateDisplacement` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress`

evaluatePrincipalStress

Package: `pde`

Evaluate principal stress at nodal locations

Syntax

```
pStress = evaluatePrincipalStress(structuralresults)
```

Description

`pStress = evaluatePrincipalStress(structuralresults)` evaluates principal stress at nodal locations using stress values from `structuralresults`. For transient and frequency response structural models, `evaluatePrincipalStress` evaluates principal stress for all time- and frequency-steps, respectively.

Examples

Octahedral Shear Stress for Bimetallic Cable Under Tension

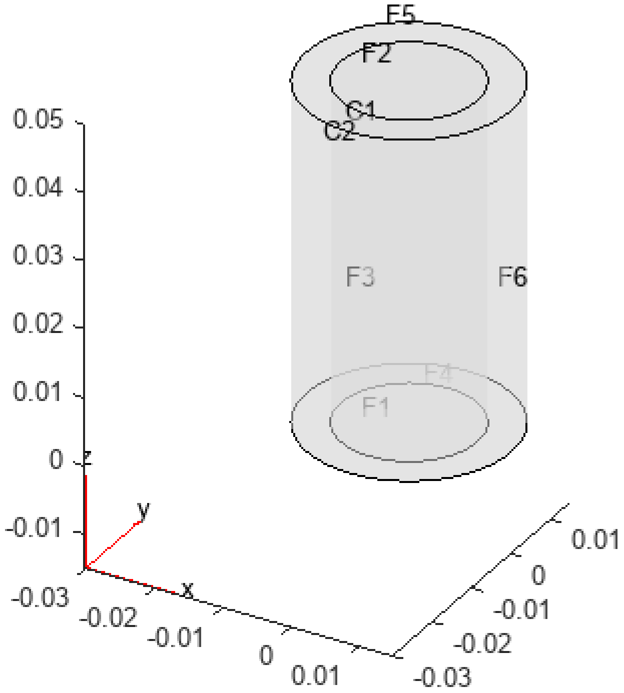
Solve a static structural model representing a bimetallic cable under tension, and compute octahedral shear stress.

Create a structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01,0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

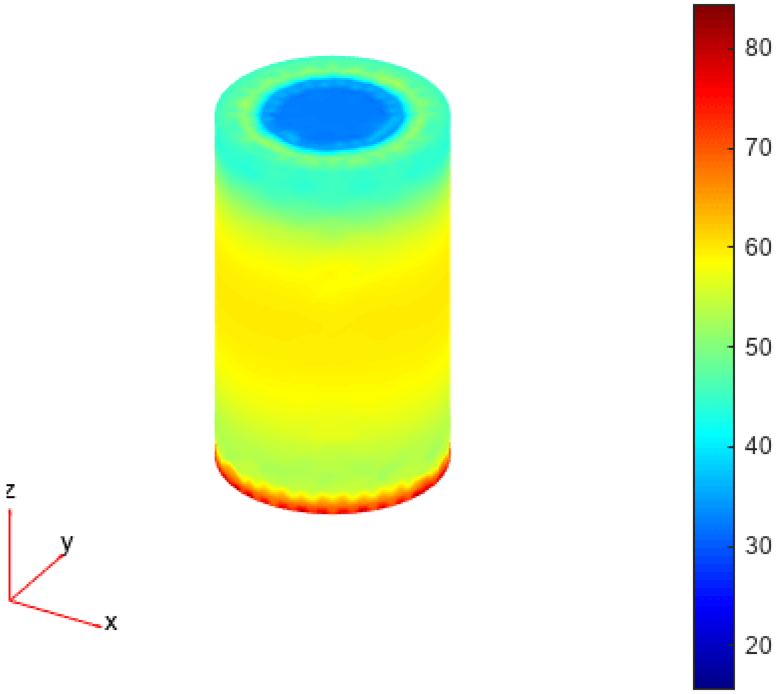
```
            Mesh: [1x1 FEMesh]
```

Evaluate the principal stress at nodal locations.

```
pStress = evaluatePrincipalStress(structuralresults);
```

Use the principal stress to evaluate the first and second invariant of stress.

```
I1 = pStress.s1 + pStress.s2 + pStress.s3;
I2 = pStress.s1.*pStress.s2 + ...
     pStress.s2.*pStress.s3 + ...
     pStress.s3.*pStress.s1;
tauOct = sqrt(2*(I1.^2 -3*I2))/3;
pdeplot3D(structuralmodel,"ColorMapData",tauOct)
```



Principal Stress for 3-D Structural Dynamic Problem

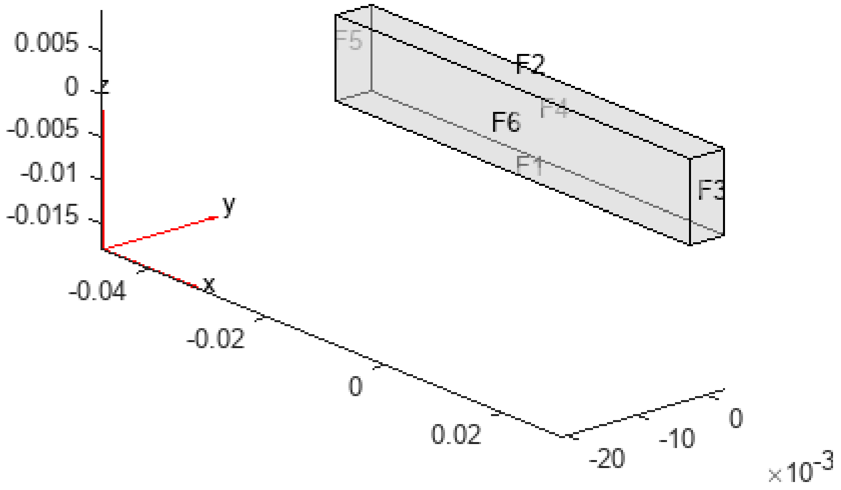
Evaluate the principal stress and octahedral shear stress in a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3,...
                             "YDisplacement",1E-4,...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Evaluate the principal stress in the beam.

```
pStress = evaluatePrincipalStress(structuralresults);
```

Use the principal stress to evaluate the first and second invariants.

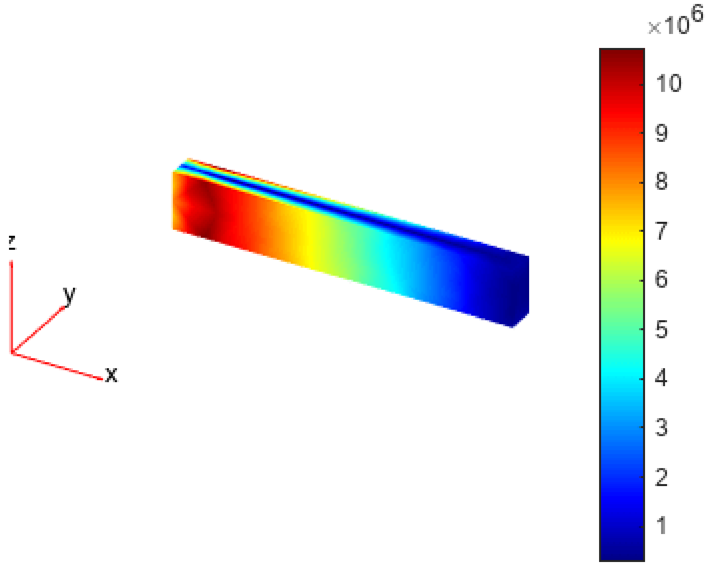
```
I1 = pStress.s1 + pStress.s2 + pStress.s3;
I2 = pStress.s1.*pStress.s2 + ...
     pStress.s2.*pStress.s3 + ...
     pStress.s3.*pStress.s1;
```

Use the stress invariants to compute the octahedral shear stress.

```
tauOct = sqrt(2*(I1.^2 -3*I2))/3;
```

Plot the results.

```
figure
pdeplot3D(structuralmodel,"ColorMapData",tauOct(:,end))
```



Input Arguments

`structuralresults` — Solution of structural analysis problem
`StaticStructuralResults` object | `TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the structural analysis problem, specified as a `StaticStructuralResults`, `TransientStructuralResults`, or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel)`

Output Arguments

`pStress` — Principal stress at nodal locations
structure array

Principal stress at the nodal locations, returned as a structure array.

Version History

Introduced in R2017b

Support for frequency response structural problems

For frequency response structural models, `evaluatePrincipalStress` evaluates principal stress for all frequency-steps.

Support for transient structural problems

For transient structural models, `evaluatePrincipalStress` evaluates principal stress for all time-steps.

See Also

`StructuralModel` | `StaticStructuralResults` | `interpolateDisplacement` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStrain`

evaluateReaction

Package: `pde`

Evaluate reaction forces on boundary

Syntax

`F = evaluateReaction(structuralresults,RegionType,RegionID)`

Description

`F = evaluateReaction(structuralresults,RegionType,RegionID)` evaluates reaction forces on the boundary specified by `RegionType` and `RegionID`. The function uses the global Cartesian coordinate system. For transient and frequency response structural models, `evaluateReaction` evaluates reaction forces for all time- and frequency-steps, respectively.

Examples

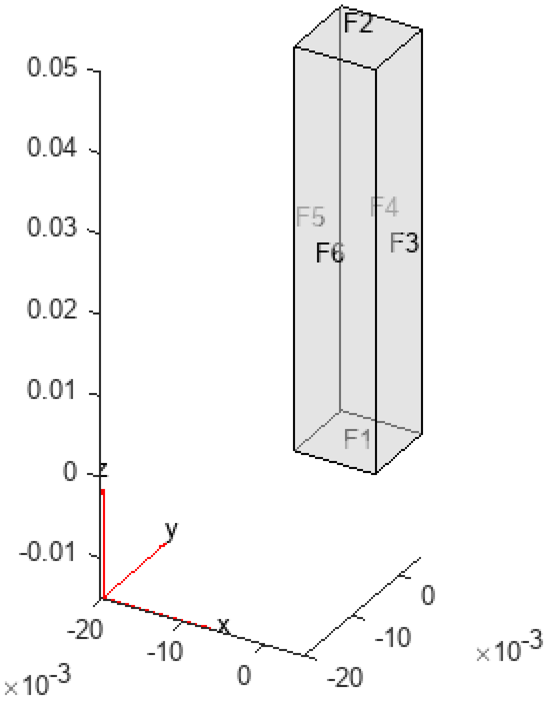
Reaction Forces on Restrained End of Prismatic Bar

Create a static structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Create a cuboid geometry and include it in the model. Plot the geometry.

```
structuralmodel.Geometry = multicuboid(0.01,0.01,0.05);
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5);
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3);
```

Fix one end of the bar and apply pressure to the opposite end.

```
structuralBC(structuralmodel,"Face",1,"Constraint","fixed")
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 1
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: "fixed"
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
```

```
           SurfaceTraction: []
                  Pressure: []
    TranslationalStiffness: []
                     Label: []
```

```
structuralBoundaryLoad(structuralmodel,"Face",2,"Pressure",100)
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 2
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
           SurfaceTraction: []
                  Pressure: 100
    TranslationalStiffness: []
                     Label: []
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel,"Hmax",0.003);
structuralresults = solve(structuralmodel);
```

Compute the reaction forces on the fixed end.

```
reaction = evaluateReaction(structuralresults,"Face",1)
```

```
reaction = struct with fields:
    Fx: -1.3620e-06
    Fy: 2.2303e-06
    Fz: 0.0103
```

Reaction Forces for 3-D Structural Dynamic Problem

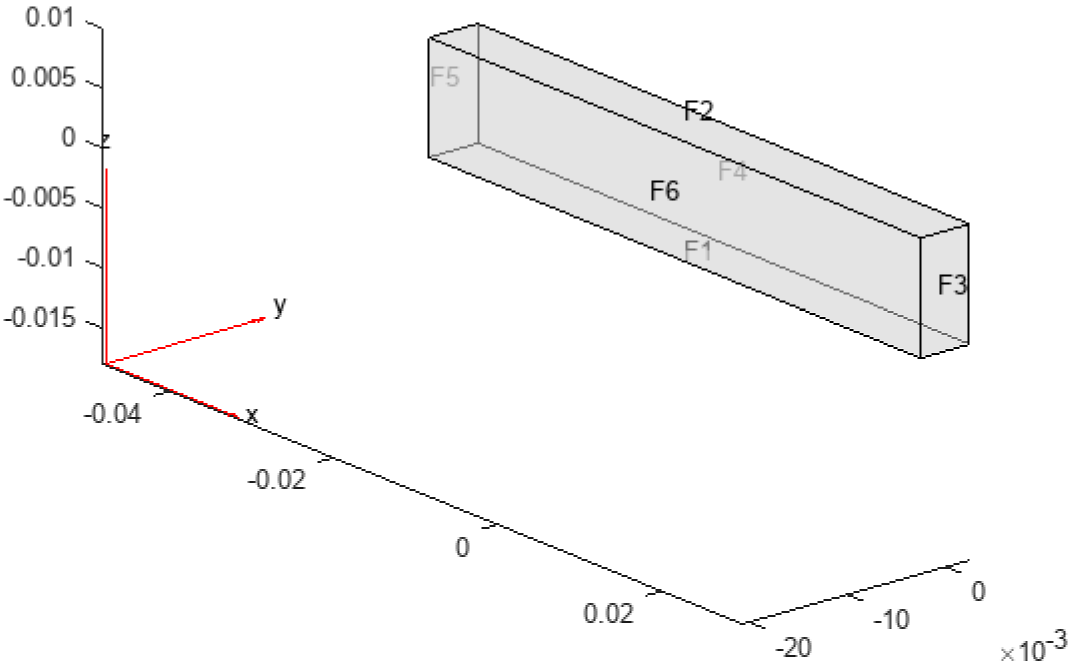
Evaluate the reaction forces at the fixed end of a beam subject to harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Compute the reaction forces on the fixed end.

```
reaction = evaluateReaction(structuralresults,"Face",5)
```

```
reaction = struct with fields:
    Fx: [101x1 double]
    Fy: [101x1 double]
    Fz: [101x1 double]
```

Input Arguments

`structuralresults` — Solution of structural analysis problem

`StaticStructuralResults` object | `TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the structural analysis problem, specified as a `StaticStructuralResults`, `TransientStructuralResults`, or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel)`

`RegionType` — Geometric region type

`"Edge"` for a 2-D model | `"Face"` for a 3-D model

Geometric region type, specified as `"Edge"` for a 2-D model or `"Face"` for a 3-D model.

Example: `evaluateReaction(structuralresults,"Face",2)`

Data Types: `char` | `string`

`RegionID` — Geometric region ID

vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `evaluateReaction(structuralresults,"Face",2)`

Data Types: `double`

Output Arguments

F — Reaction forces

structure array

Reaction forces, returned as a structure array. The array fields represent the integrated reaction forces and surface traction vector, which are computed by using the state of stress on the boundary and the outward normal.

Version History

Introduced in R2017b

Support for frequency response structural problems

For frequency response structural models, `evaluateReaction` evaluates reaction forces for all frequency-steps.

Support for transient structural problems

For transient structural models, `evaluateReaction` evaluates reaction forces for all time-steps.

See Also

`StructuralModel` | `StaticStructuralResults` | `interpolateDisplacement` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

evaluateStrain

Package: `pde`

Evaluate strain for dynamic structural analysis problem

Syntax

`nodalStrain = evaluateStrain(structuralresults)`

Description

`nodalStrain = evaluateStrain(structuralresults)` evaluates strain at nodal locations for all time- or frequency-steps.

Examples

Strain for 3-D Structural Dynamic Problem

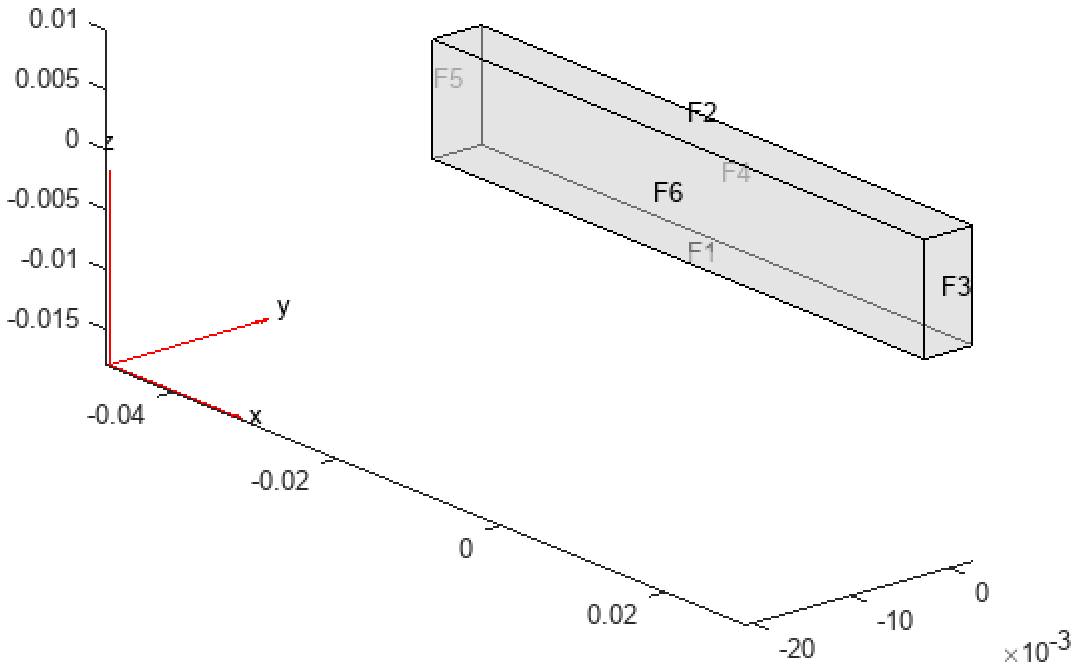
Evaluate the strain in a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0,0,0],"Velocity",[0,0,0]);
```

Solve the model.

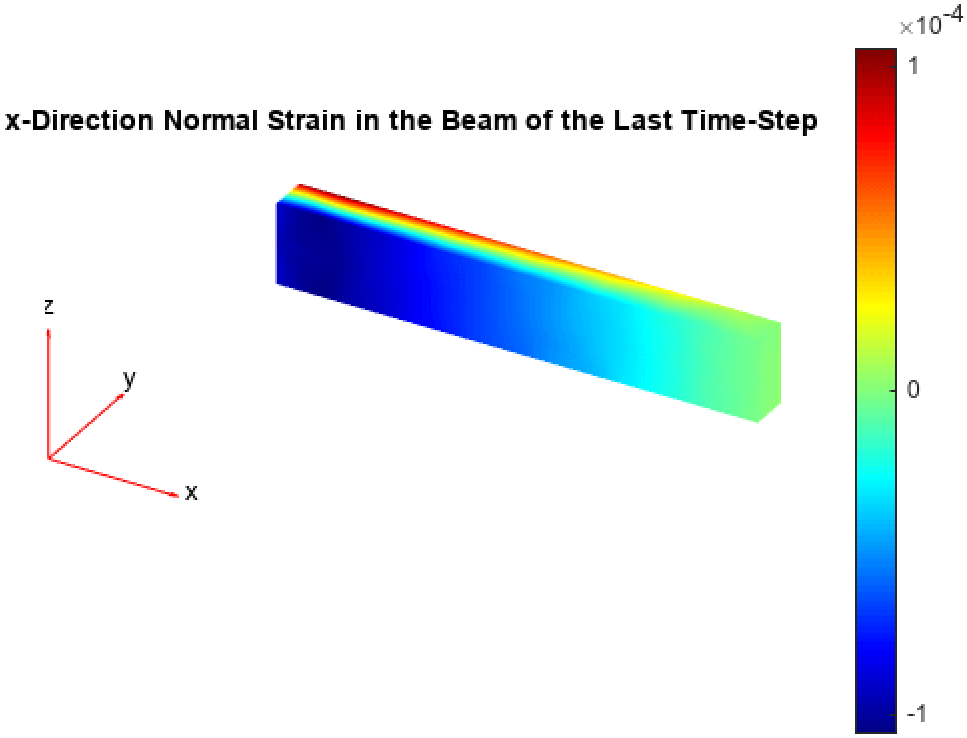
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Evaluate the strain in the beam.

```
strain = evaluateStrain(structuralresults);
```

Plot the normal strain along `x`-direction for the last time-step.

```
figure
pdeplot3D(structuralmodel,"ColorMapData",strain.exx(:,end))
title("x-Direction Normal Strain in the Beam of the Last Time-Step")
```



Input Arguments

`structuralresults` — Solution of dynamic structural analysis problem

`TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of a dynamic structural analysis problem, specified as a `TransientStructuralResults` or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel,tlist)`

Output Arguments

`nodalStrain` — Strain at nodes

`FEStruct` object

Strain at the nodes, returned as an `FEStruct` object with the properties representing the components of strain tensor at nodal locations. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2018a

See Also

`StructuralModel` | `TransientStructuralResults` | `interpolateDisplacement` | `interpolateVelocity` | `interpolateAcceleration` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateStress` | `evaluateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

evaluateStress

Package: `pde`

Evaluate stress for dynamic structural analysis problem

Syntax

`nodalStress = evaluateStress(structuralresults)`

Description

`nodalStress = evaluateStress(structuralresults)` evaluates stress at nodal locations for all time- or frequency-steps.

Examples

Stress for 3-D Structural Dynamic Problem

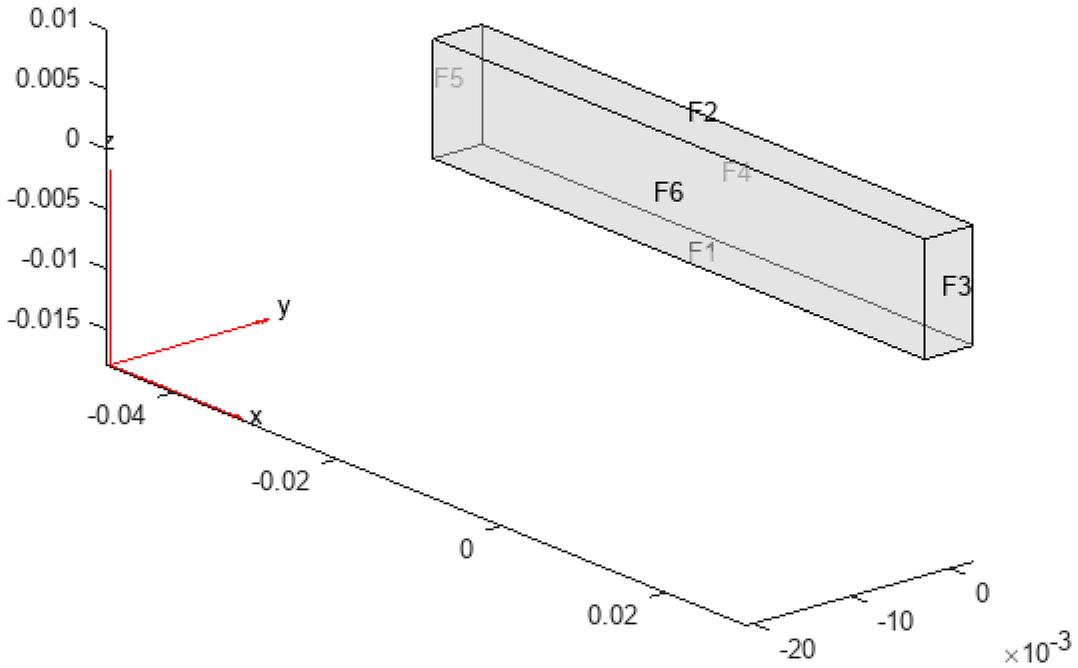
Evaluate the stress in a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0,0,0],"Velocity",[0,0,0]);
```

Solve the model.

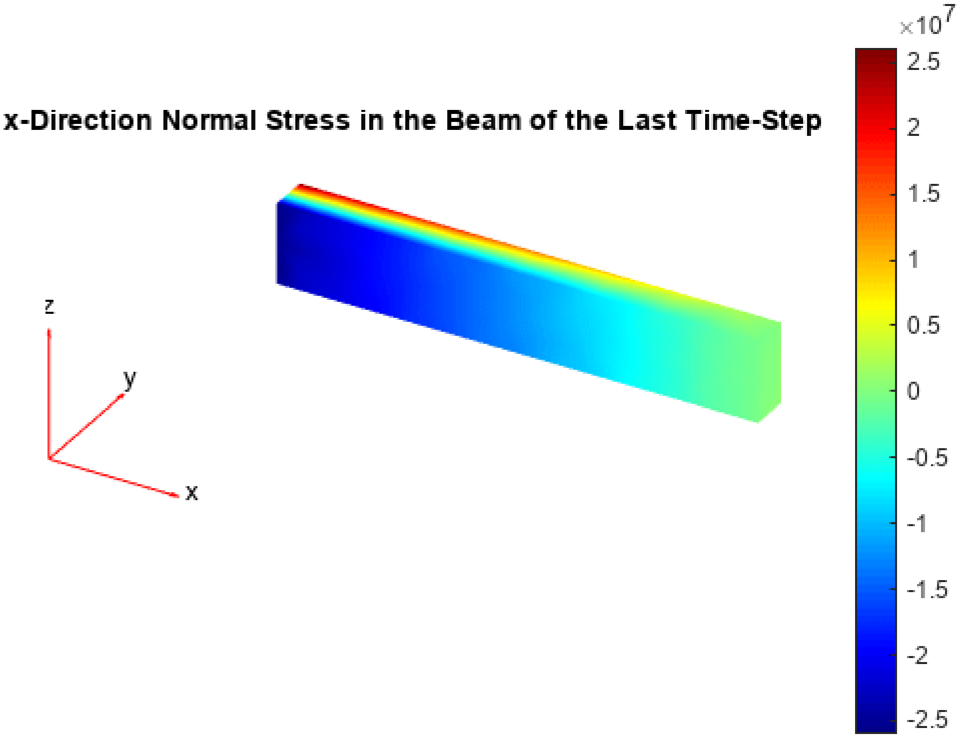
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Evaluate stress in the beam.

```
stress = evaluateStress(structuralresults);
```

Plot the normal stress along x-direction for the last time-step.

```
figure
pdeplot3D(structuralmodel,"ColorMapData",stress.sxx(:,end))
title("x-Direction Normal Stress in the Beam of the Last Time-Step")
```



Input Arguments

`structuralresults` — Solution of dynamic structural analysis problem

`TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of a dynamic structural analysis problem, specified as a `TransientStructuralResults` or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel,tlist)`

Output Arguments

`nodalStress` — Stress at nodes

`FEStruct` object

Stress at the nodes, returned as an `FEStruct` object with the properties representing the components of a stress tensor at nodal locations. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2018a

See Also

`StructuralModel` | `TransientStructuralResults` | `interpolateDisplacement` | `interpolateVelocity` | `interpolateAcceleration` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateStrain` | `evaluateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

evaluateTemperatureGradient

Package: `pde`

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.

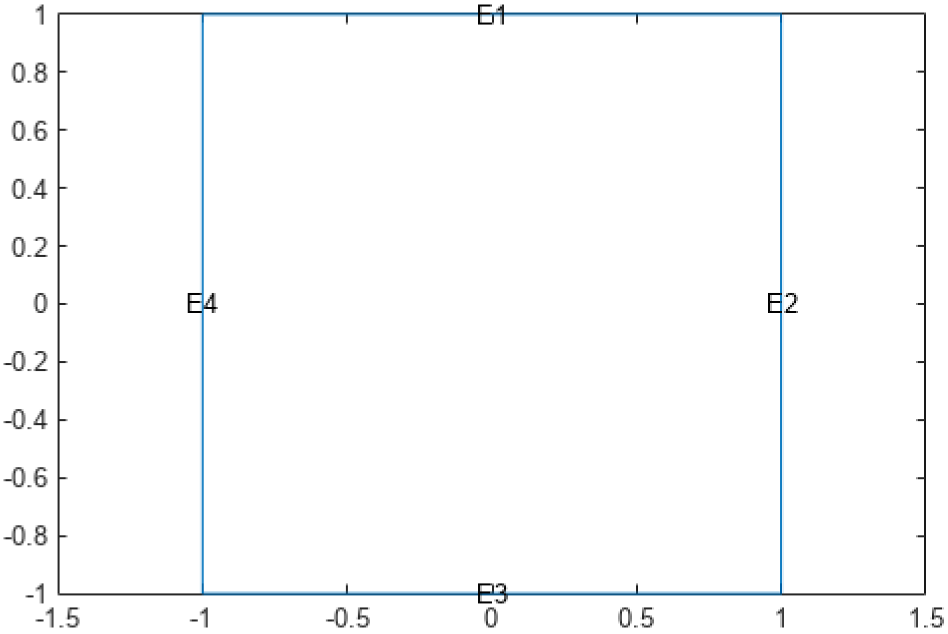
Create a thermal model for steady-state analysis.

```
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.

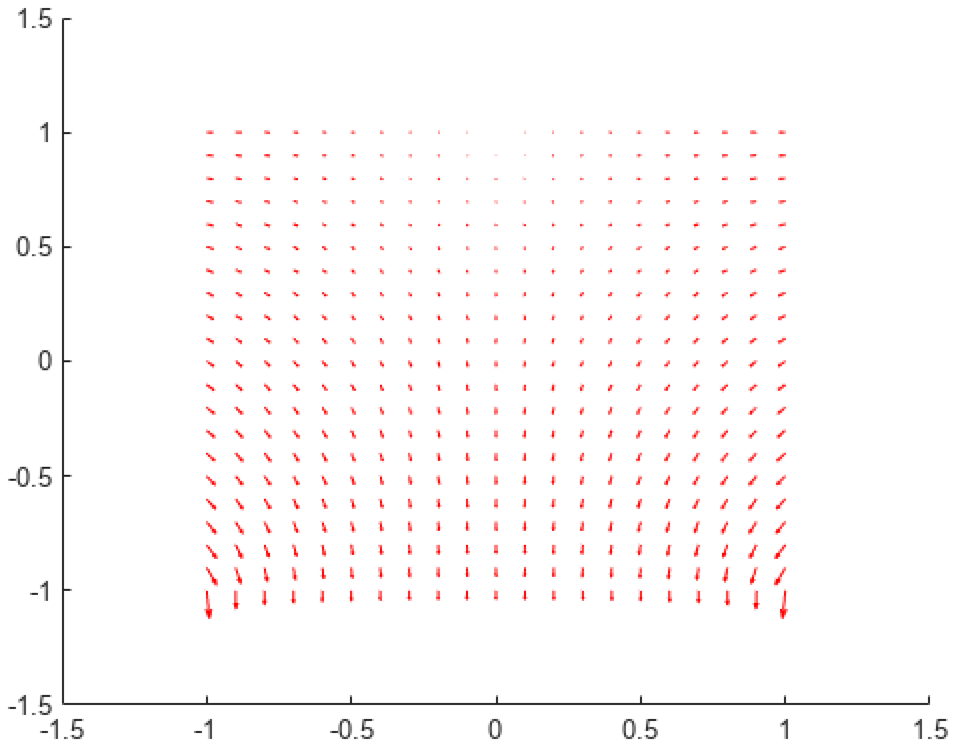
```
generateMesh(thermalmodel);
results = solve(thermalmodel)
```

```
results = 
  SteadyStateThermalResults with properties:

    Temperature: [1541x1 double]
     XGradients: [1541x1 double]
```

```
    YGradients: [1541x1 double]
    ZGradients: []
          Mesh: [1x1 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]);
```

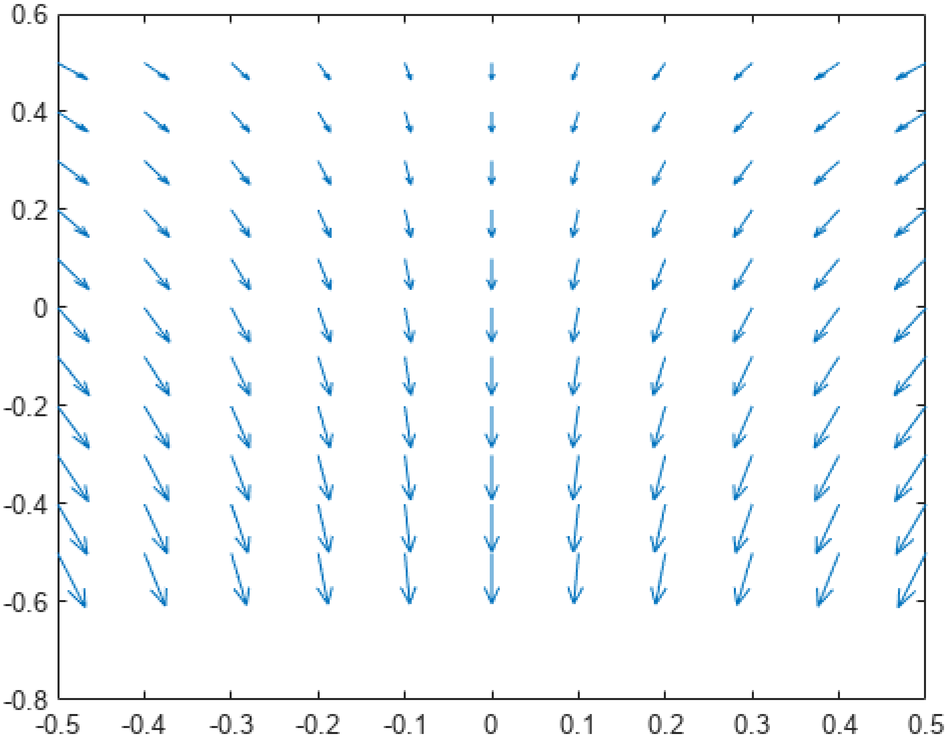


Create a grid specified by `x` and `y` coordinates, and evaluate temperature gradients to the grid.

```
v = linspace(-0.5,0.5,11);
[X,Y] = meshgrid(v);

[gradTx,gradTy] = ...
evaluateTemperatureGradient(results,X,Y);
```

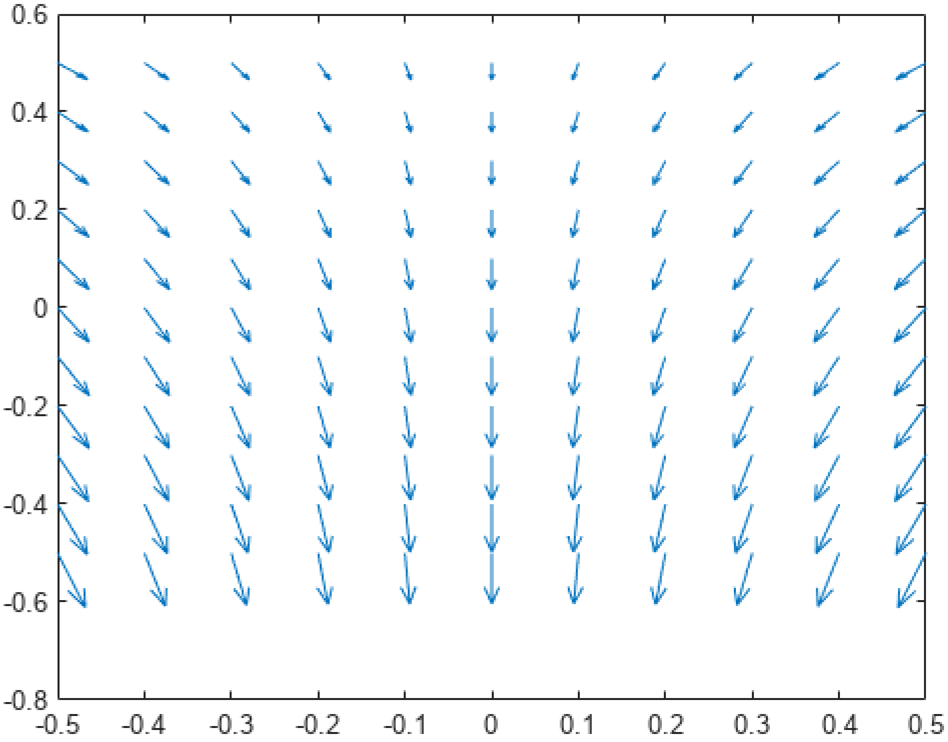
Reshape the `gradTx` and `gradTy` vectors, and plot the resulting temperature gradients.

```
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

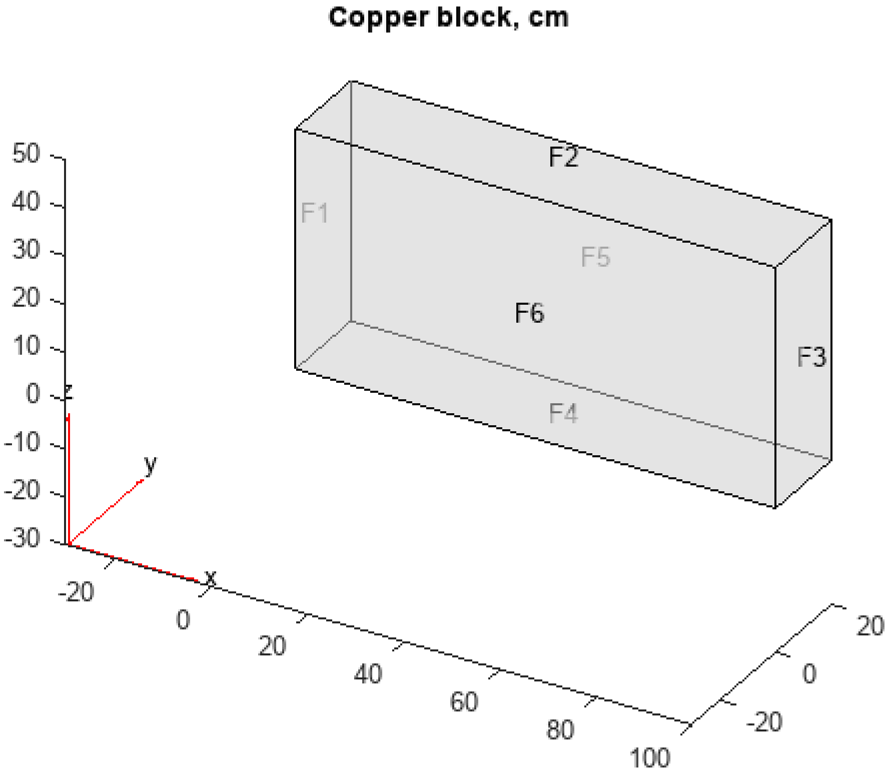
For a 3-D steady-state thermal model, evaluate temperature gradients at the nodal locations and at the points specified by `x`, `y`, and `z` coordinates.

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")
axis equal
```



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: [12691x1 double]
     XGradients: [12691x1 double]
     YGradients: [12691x1 double]
     ZGradients: [12691x1 double]
```

```
            Mesh: [1x1 FEMesh]
```

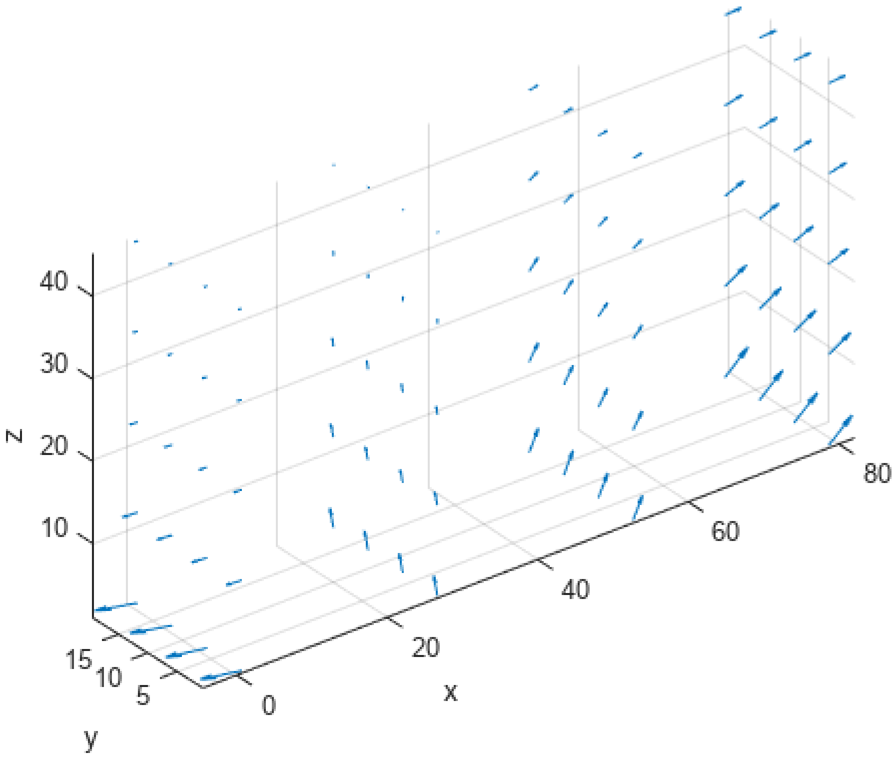
The solver finds the values of temperatures and temperature gradients at the nodal locations. To access these values, use `results.Temperature`, `results.XGradients`, and so on.

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);
```

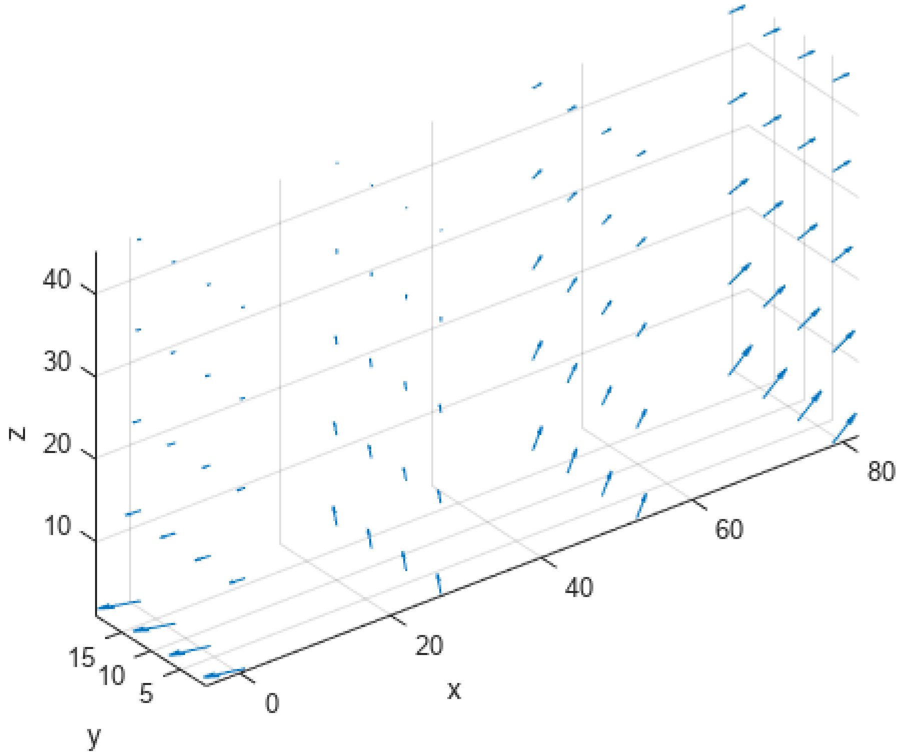
Reshape the `gradTx`, `gradTy`, and `gradTz` vectors, and plot the resulting temperature gradients.

```
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

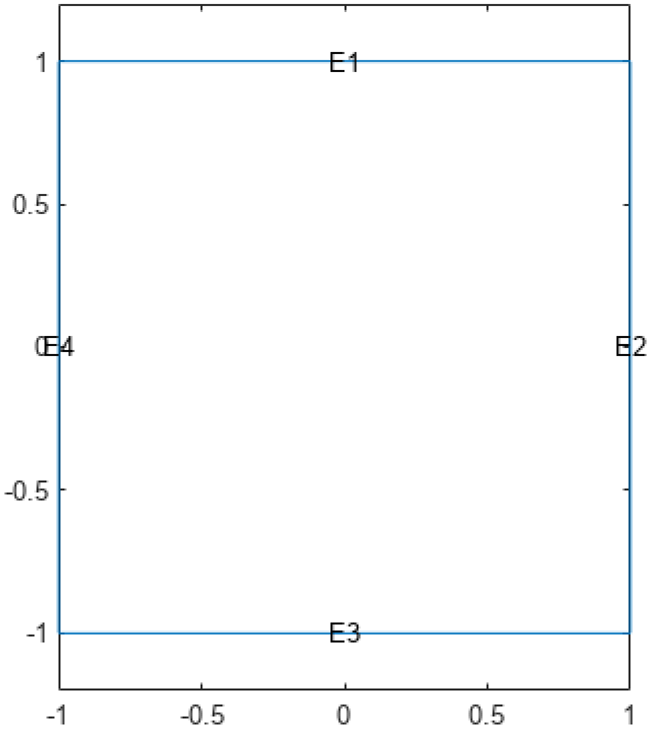
Solve a 2-D transient heat transfer problem on a square domain and compute temperature gradients at the convective boundary.

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 $W/(m^{\circ}C)$, mass density is 7800 kg/m^3, and specific heat is 500 $J/(kg^{\circ}C)$.

```
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.

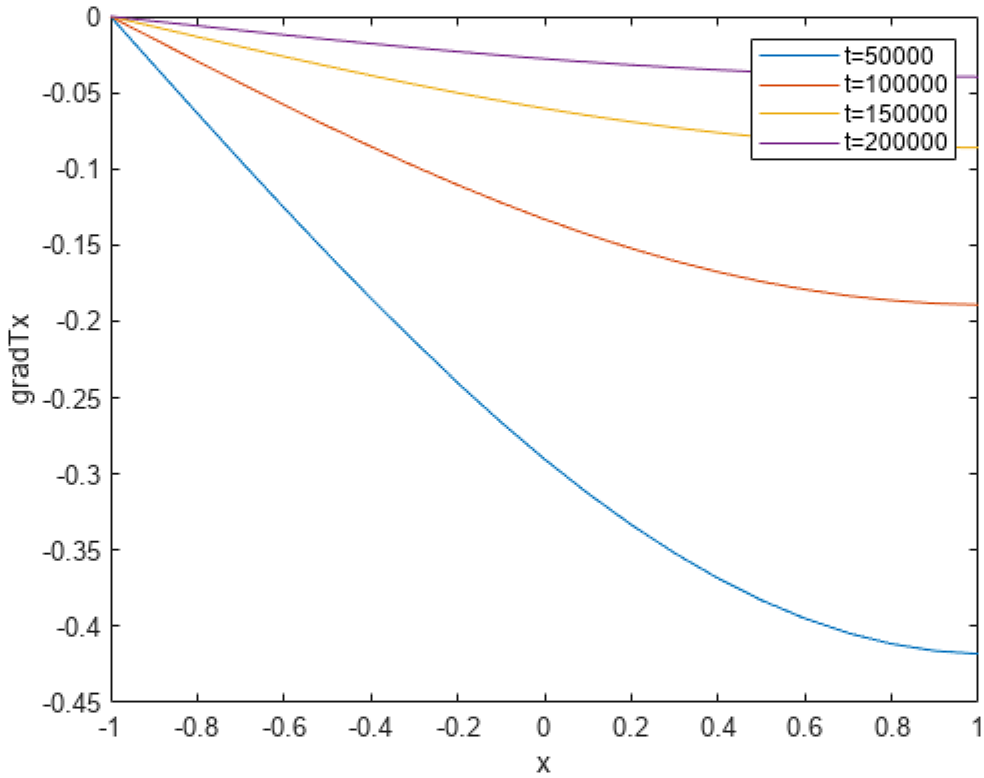
```
generateMesh(thermalmodel);
tlist = 0:1000:200000;
thermalresults = solve(thermalmodel,tlist);
```

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`.

Version History

Introduced in R2017a

See Also

`ThermalModel` | `SteadyStateThermalResults` | `TransientThermalResults` | `evaluateHeatFlux` | `evaluateHeatRate` | `interpolateTemperature`

evaluateVonMisesStress

Package: `pde`

Evaluate von Mises stress for dynamic structural analysis problem

Syntax

```
vmStress = evaluateVonMisesStress(structuralresults)
```

Description

`vmStress = evaluateVonMisesStress(structuralresults)` evaluates von Mises stress at nodal locations for all time- or frequency-steps.

Examples

von Mises Stress for 3-D Structural Dynamic Problem

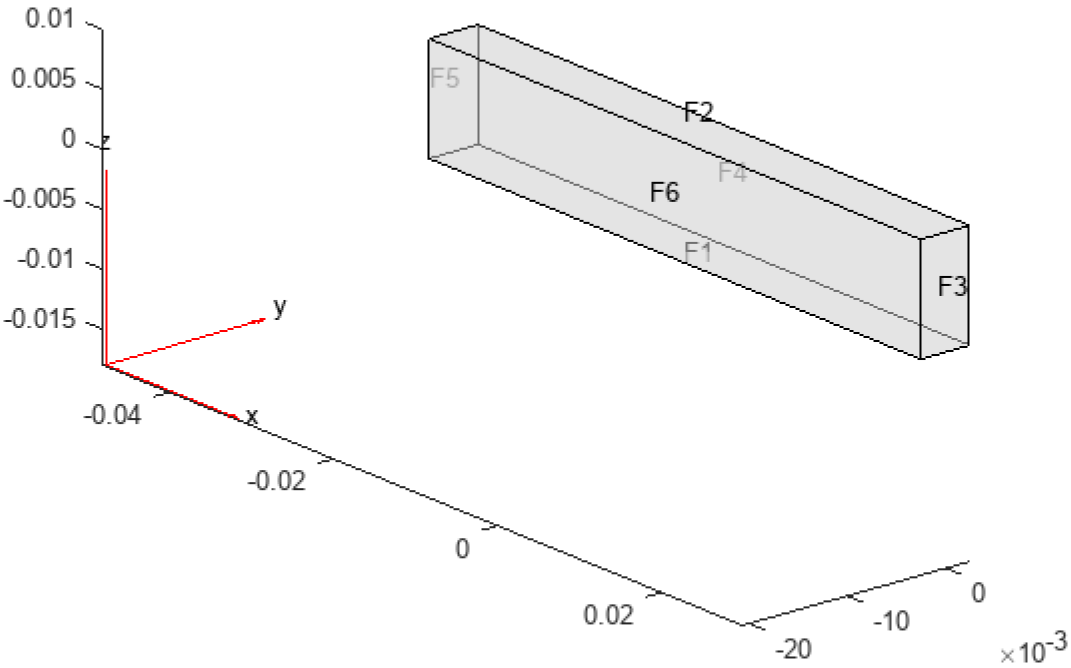
Evaluate the von Mises stress in a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

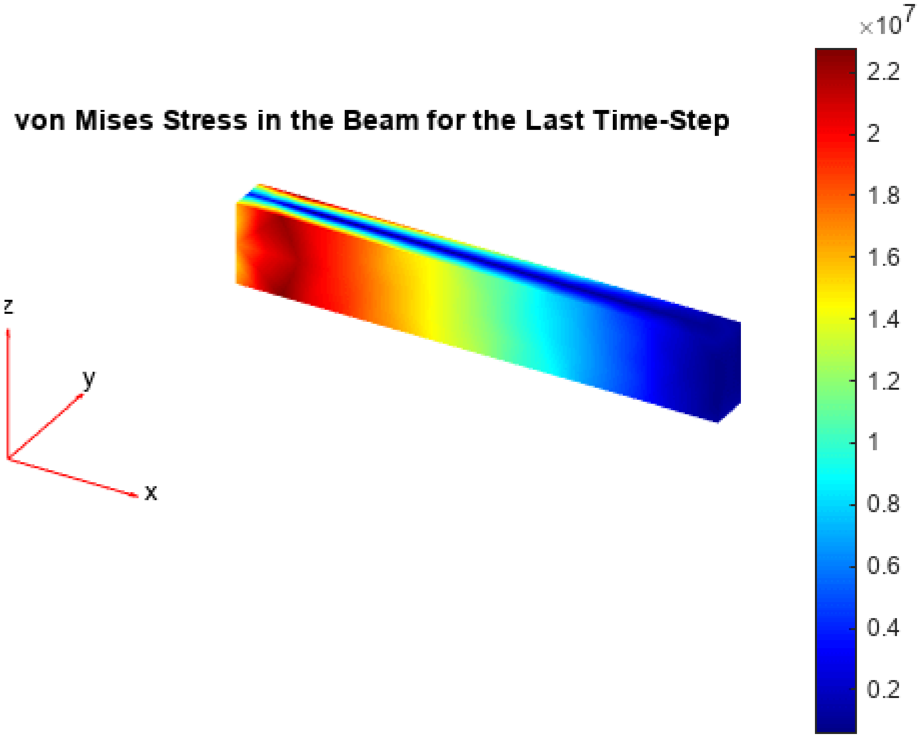
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Evaluate the von Mises stress in the beam.

```
vmStress = evaluateVonMisesStress(structuralresults);
```

Plot the von Mises stress for the last time-step.

```
figure
pdeplot3D(structuralmodel,"ColorMapData",vmStress(:,end))
title("von Mises Stress in the Beam for the Last Time-Step")
```



Input Arguments

`structuralresults` — Solution of dynamic structural analysis problem
`TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of a dynamic structural analysis problem, specified as a `TransientStructuralResults` or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel,tlist)`

Output Arguments

`vmStress` — Von Mises Stress at nodes
matrix

Von Mises Stress at the nodes, returned as a matrix. The rows of the matrix contain the values of von Mises stress at nodal locations, while the columns correspond to the time or frequency steps.

Version History

Introduced in R2018a

See Also

`StructuralModel` | `TransientStructuralResults` | `interpolateDisplacement` | `interpolateVelocity` | `interpolateAcceleration` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateStrain` | `evaluateStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

FEMesh Properties

Mesh object

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

Properties

Nodes — Mesh nodes
matrix

Mesh nodes, specified 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 2-175.

Data Types: `double`

`Elements` — Mesh elements
matrix

Mesh elements, specified 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, specified 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, specified as a positive real number. The minimum mesh element size is the length of the shortest edge in the mesh. The `Hmin` name-value pair passed to the `generateMesh` function sets the target minimum size the at the time it creates the mesh. `generateMesh` can occasionally create a mesh with some elements that are smaller than `MinElementSize`.

Data Types: `double`

`MeshGradation` — Mesh growth rate
1.5 (default) | scalar strictly between 1 and 2

Mesh growth rate, specified as a scalar strictly between 1 and 2.

Data Types: `double`

`GeometricOrder` — Element polynomial order
`'linear'` | `'quadratic'`

Element polynomial order, specified as `'linear'` or `'quadratic'`. See `Elements` or “Mesh Data” on page 2-175.

Data Types: `char`

Version History

Introduced in R2015a

See Also

`generateMesh` | `meshToPet` | `PDEModel` | `findElements` | `findNodes` | `meshQuality` | `area` | `volume`

Topics

“Solve Problems Using PDEModel Objects” on page 2-2
“Finite Element Method Basics” on page 1-11
“Mesh Data” on page 2-175

extrude

Package: `pde`

Vertically extrude 2-D geometry or specified faces of 3-D geometry

Syntax

```
extrude(g,height)
extrude(g,FaceID,height)
h = extrude( ___ )
```

Description

`extrude(g,height)` creates a 3-D discrete geometry by extruding a 2-D geometry along the *z*-axis by the value of `height`. You can create a stacked multilayered 3-D discrete geometry by specifying `height` as a vector of thicknesses of the layers.

`extrude(g,FaceID,height)` extrudes specified faces of a 3-D geometry along the direction normal to the faces. Here, `FaceID` specifies which faces to extrude. You can extrude faces into multiple layers by specifying `height` as a vector of thicknesses of the layers.

All of the specified faces must be flat and have the same orientation. The extruded volumes must not intersect with each other or with the existing geometry.

`h = extrude( ___ )` returns a handle `h`. If the original geometry is a `DiscreteGeometry` object, then the function modifies the original geometry, and `h` a handle to the modified `DiscreteGeometry` object. If the original geometry is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` object. In this case, the original geometry remains unchanged.

Examples

Single Layer Extrusion

Create a 3-D geometry by extruding a 2-D geometry along the *z*-axis.

Create a PDE model.

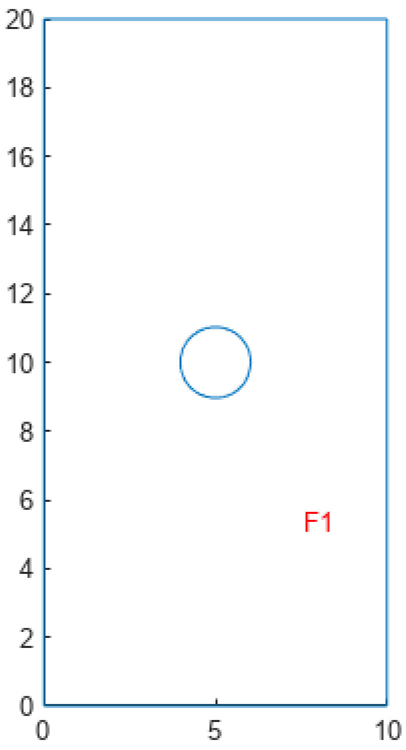
```
model = createpde;
```

Import a 2-D geometry.

```
g = importGeometry(model,"PlateHolePlanar.stl");
```

Plot the geometry and display the face labels.

```
pdegplot(g,"FaceLabels","on")
```



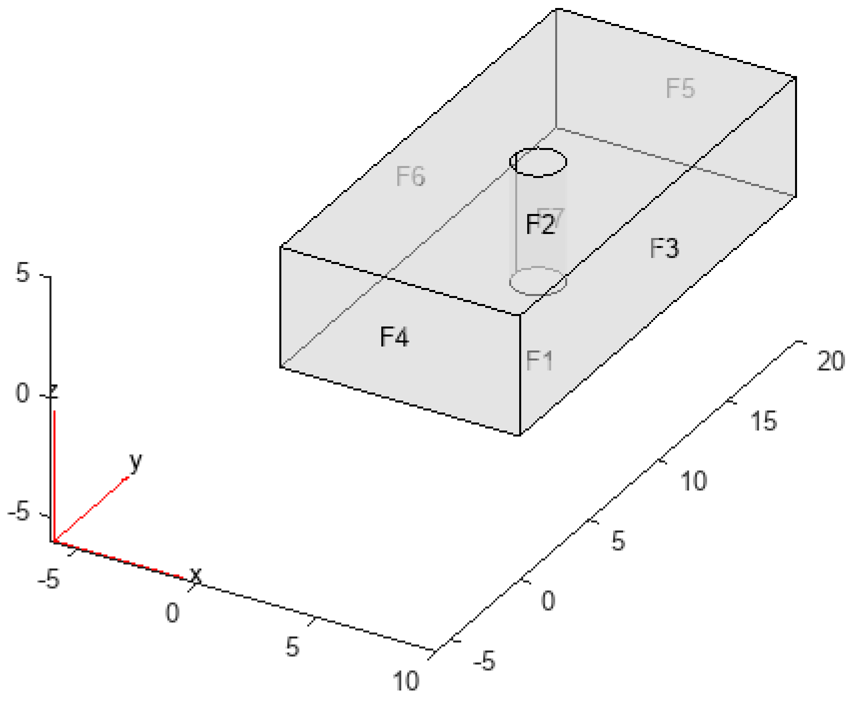
Create a 3-D geometry by extruding the 2-D geometry along the *z*-axis by 5 units.

```
extrude(g,5)
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 7
       NumEdges: 15
    NumVertices: 10
       Vertices: [10x3 double]
```

Plot the new geometry and display the face labels.

```
pdegplot(g,"FaceLabels","on","FaceAlpha",0.5)
```



Multiple Layer Extrusion

Create a stacked multilayered 3-D geometry by extruding a 2-D geometry along the *z*-axis.

Create a PDE model.

```
model = createpde;
```

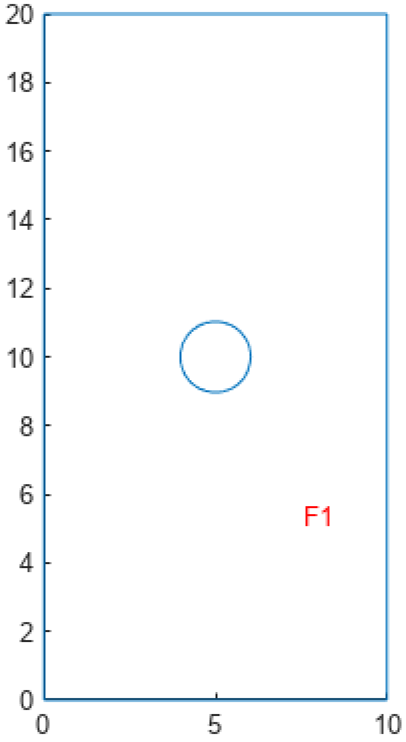
Import a geometry.

```
g = importGeometry(model,"PlateHolePlanar.stl")
```

```
g = 
  DiscreteGeometry with properties:

       NumCells: 0
       NumFaces: 1
       NumEdges: 5
    NumVertices: 5
       Vertices: [5x3 double]
```

Plot the geometry and display the face labels.

```
pdegplot(g,"FaceLabels","on")
```



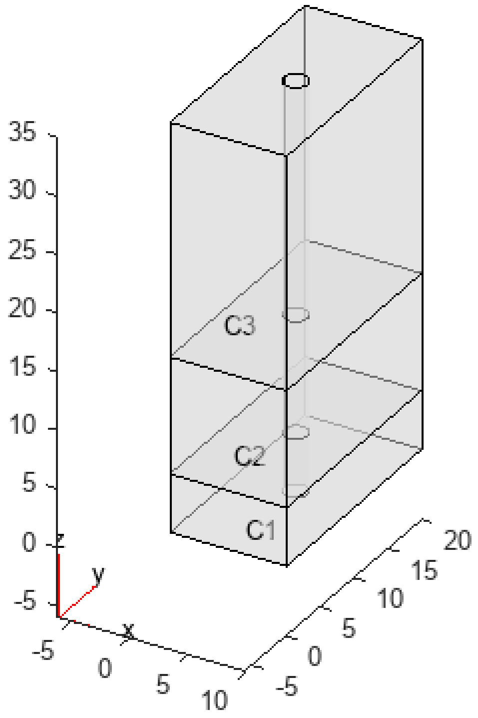
Create a 3-D geometry consisting of three blocks with holes stacked on top of each other. The heights of the blocks are 5, 10, and 20 units.

```
extrude(g,[5,10,20])
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 3
       NumFaces: 19
       NumEdges: 35
    NumVertices: 20
       Vertices: [20x3 double]
```

Plot the new geometry and display the cell labels.

```
pdegplot(g,"CellLabels","on","FaceAlpha",0.5)
```



Geometry with an Added Vertex

Extrude a 2-D geometry that has a vertex added by the `addVertex` function. The layers of the extruded geometry all have a corresponding vertex, but there are no edges between these vertices.

Create a PDE model.

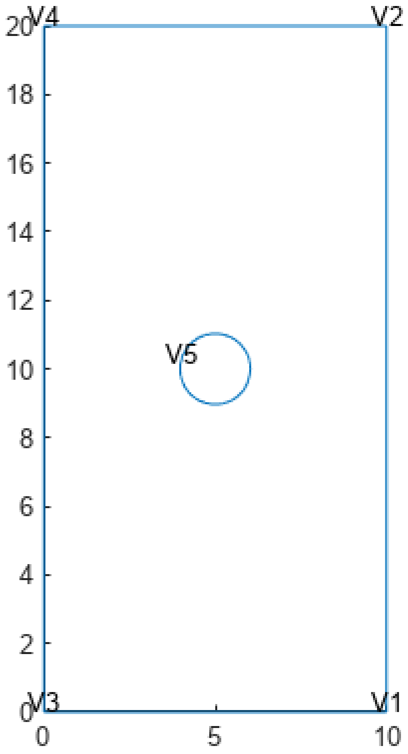
```
model = createpde;
```

Import a geometry.

```
g = importGeometry(model,"PlateHolePlanar.stl");
```

Plot the geometry and display the vertex labels.

```
pdegplot(g,"VertexLabels","on")
```

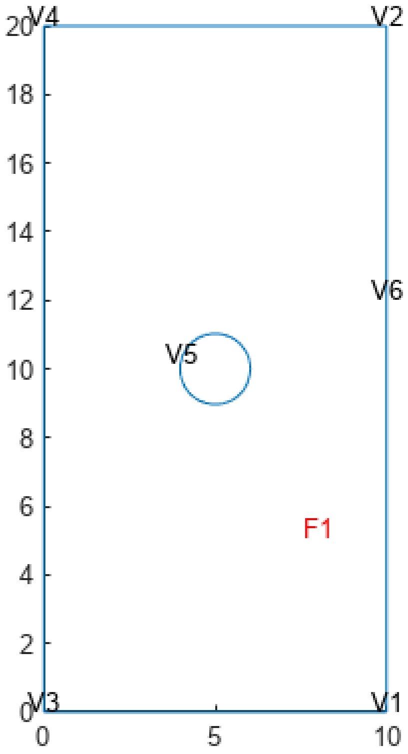


Add a new vertex on the right edge.

```
addVertex(g,"Coordinates",[10 12]);
```

Plot the new geometry and display the vertex labels.

```
pdegplot(g,"FaceLabels","on","VertexLabels","on")
```



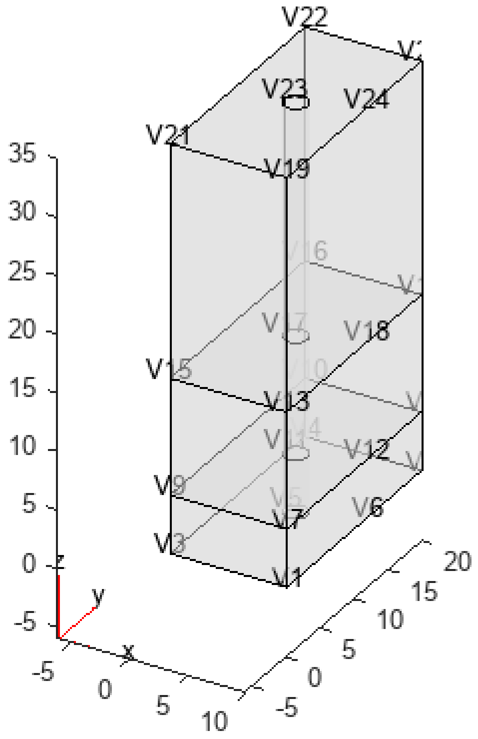
Create a 3-D geometry consisting of three blocks with holes stacked on top of each other. The heights of the blocks are 5, 10, and 20 units.

```
extrude(g,[5,10,20])
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 3
       NumFaces: 19
       NumEdges: 35
    NumVertices: 24
       Vertices: [24x3 double]
```

Plot the new geometry and display the vertex labels. The `extrude` function replicates the added vertex `V6` into three new vertices: `V12`, `V18`, and `V24`. It does not create edges between these vertices.

```
pdegplot(g,"VertexLabels","on","FaceAlpha",0.5)
```



Geometry with an Added Face

Extrude a 2-D geometry that has a face added by the `addFace` function.

Create a PDE model.

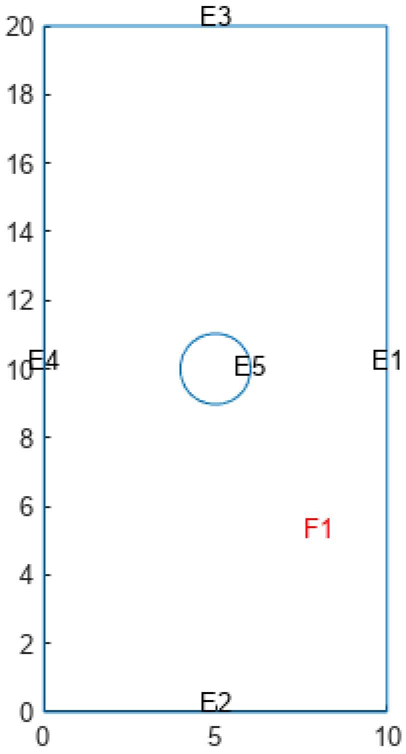
```
model = createpde;
```

Import a geometry.

```
g = importGeometry(model,"PlateHolePlanar.stl");
```

Plot the geometry and display the face and edge labels.

```
pdegplot(g,"FaceLabels","on","EdgeLabels","on")
```



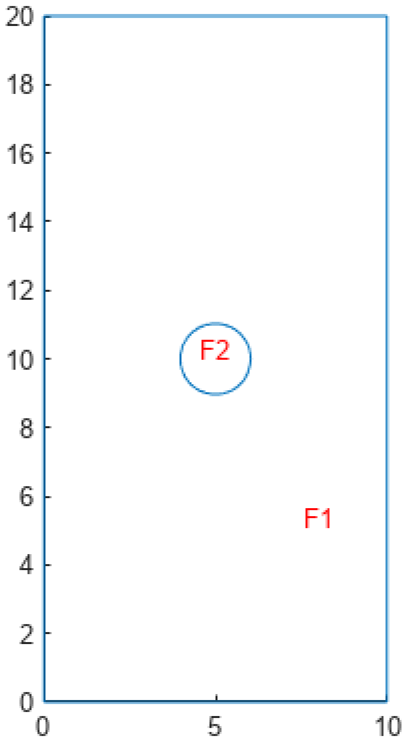
Fill the hole in the center by adding a face.

```
addFace(g,5)
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 0
       NumFaces: 2
       NumEdges: 5
    NumVertices: 5
       Vertices: [5x3 double]
```

Plot the modified geometry.

```
pdegplot(g,"FaceLabels","on")
```



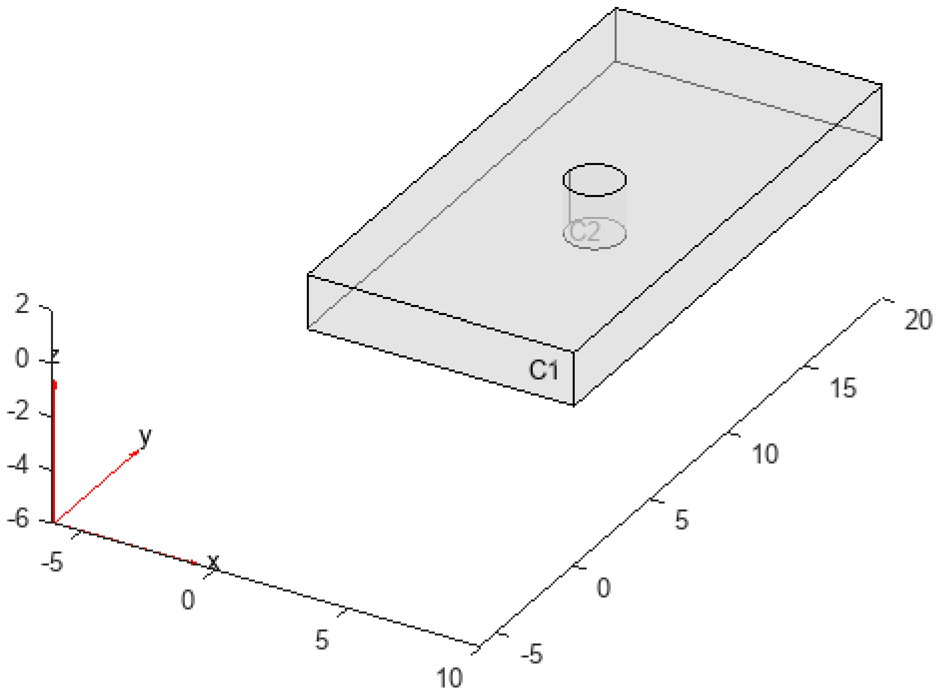
Create a 3-D geometry by extruding the 2-D geometry along the *z*-axis by 2 units.

```
extrude(g,2)
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 2
       NumFaces: 9
       NumEdges: 15
    NumVertices: 10
       Vertices: [10x3 double]
```

Plot the new geometry and display the cell labels.

```
pdegplot(g,"CellLabels","on","FaceAlpha",0.5)
```

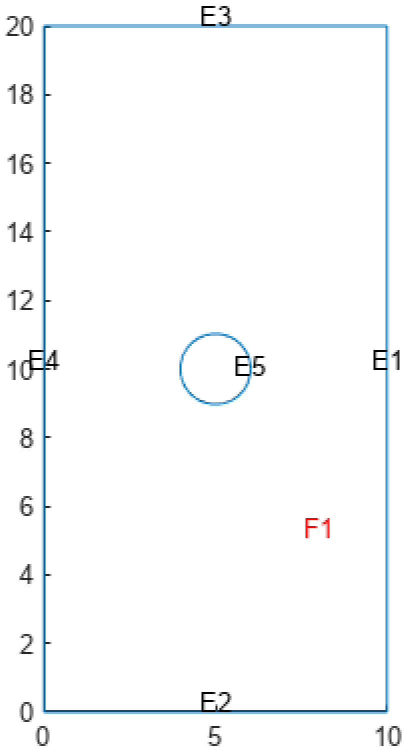


Faces of 3-D Geometry

Extrude specified faces of a 3-D geometry.

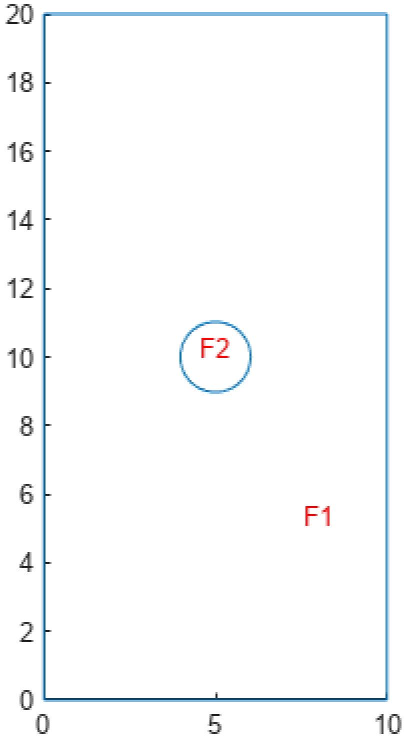
Import the geometry and plot it with the face and edge labels.

```
g = importGeometry("PlateHolePlanar.stl");
pdegplot(g,"FaceLabels","on","EdgeLabels","on")
```



Fill the hole in the center by adding a face. Plot the modified geometry.

```
addFace(g,5);
pdegplot(g,"FaceLabels","on")
```

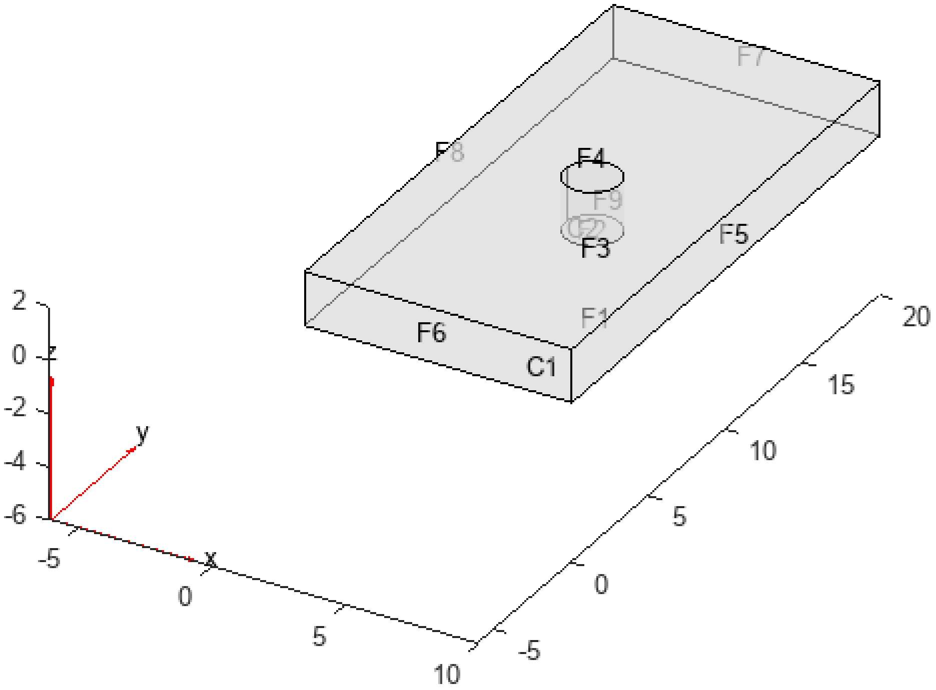


Create a 3-D geometry by extruding the 2-D geometry along the *z*-axis by 2 units.

```
extrude(g,2);
```

Plot the new geometry with the cell and face labels.

```
pdegplot(g,"CellLabels","on","Facelabels","on","FaceAlpha",0.5)
```

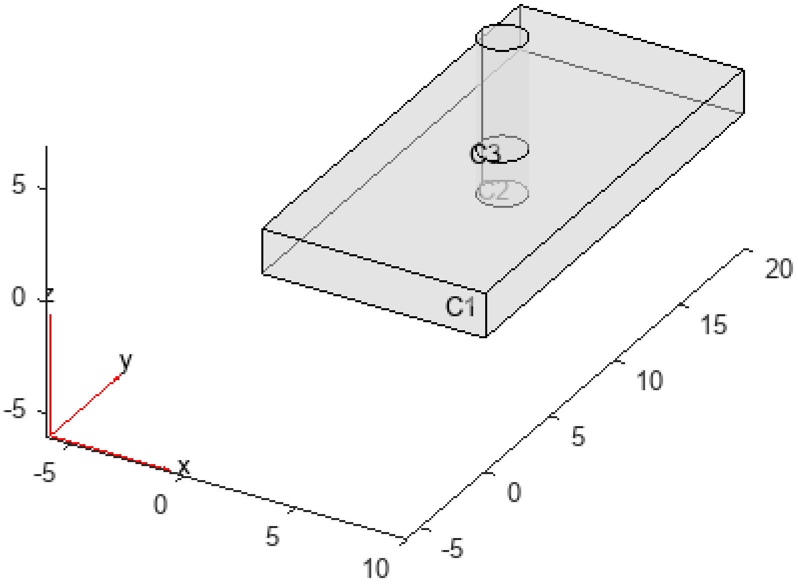


Now, extrude the center face of the geometry by 5 units.

```
extrude(g,4,5);
```

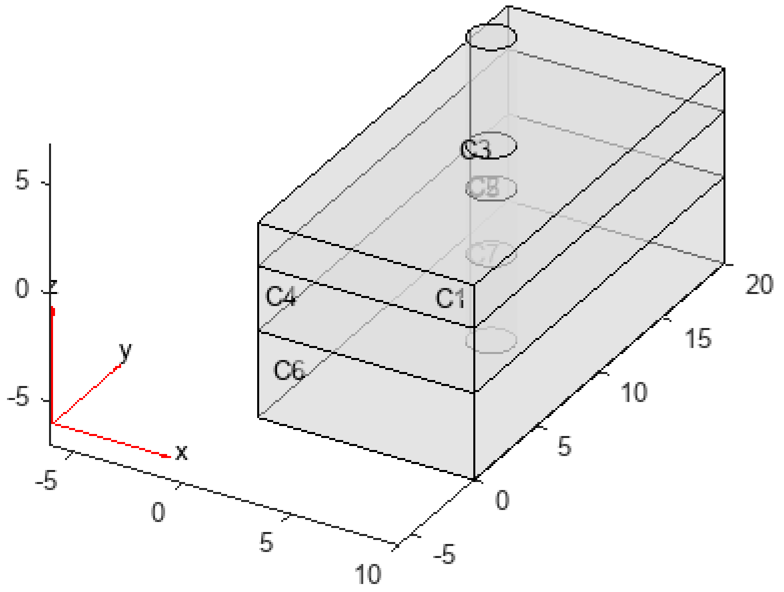
Plot the resulting geometry with the cell labels.

```
pdegplot(g,"CellLabels","on","FaceAlpha",0.5)
```



Now, call `extrude` again, and this time specify a vector of heights. The function extrudes all specified faces by each of the specified heights, which creates multiple layers.

```
extrude(g,[1 2],[3 4]);
pdegplot(g,"CellLabels","on","FaceAlpha",0.5)
```



Input Arguments

g — Geometry

`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

height — Cell heights

positive real number | vector of positive real numbers

Cell heights, specified as a positive real number or a vector of positive real numbers.

If `height` is a vector and `g` is a 2-D geometry, then `height(i)` specifies the height of the `i`th layer of a multilayered (stacked) 3-D geometry. Each layer constitutes a new cell.

If `g` is a 3-D geometry, the function extrudes all specified faces into several layers, with `height(i)` specifying the height of the `i`th layer.

Example: `extrude(g,5.5)`

FaceID — Faces to extrude in 3-D geometry

positive real number | vector of positive real numbers

Faces to extrude in 3-D geometry, specified as a positive real number or a vector of positive real numbers. If `height` is a vector, then the function extrudes all specified faces into several layers, same as it does for 2-D geometries.

Output Arguments

`h` — Resulting geometry
handle

Resulting geometry, returned as a handle. If the original geometry `g` is a `DiscreteGeometry` object, then `h` is a handle to the modified `DiscreteGeometry` object `g`. If `g` is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` object. In this case, the original geometry `g` remains unchanged.

Tips

- `extrude` modifies a geometry, but it does not modify the corresponding mesh. After modifying a geometry, regenerate the mesh to ensure a proper mesh association with the new geometry.
- If a 2-D geometry has new vertices added by using the `addVertex` function, `extrude` replicates the new vertices on each new layer of the extruded 3-D geometry, but it does not connect these vertices by edges.
- If `g` is an `AnalyticGeometry` object, and you want to replace it with the extruded discrete 3-D geometry, assign the output to the original geometry, for example, `g = extrude(g,20)`.

Version History

Introduced in R2020b

Extrude specified faces of a 3-D geometry

In addition to extruding 2-D geometries into 3-D geometries, the function now lets you extrude specified faces of a 3-D geometry. The function extrudes specified faces along the direction normal to the faces.

See Also

DiscreteGeometry Properties | AnalyticGeometry Properties | `addVertex` | `addFace` | `pdegplot` | `importGeometry` | `generateMesh` | `multicuboid` | `multicylinder` | `multisphere`

faceEdges

Find edges belonging to specified faces

Syntax

```
EdgeID = faceEdges(g,RegionID)
EdgeID = faceEdges(g,RegionID,FilterType)
```

Description

`EdgeID = faceEdges(g,RegionID)` finds edges belonging to the faces with ID numbers listed in `RegionID`.

`EdgeID = faceEdges(g,RegionID,FilterType)` returns internal, external, or all edges belonging to the faces with ID numbers listed in `RegionID`. This syntax is valid for 3-D geometries only.

Examples

Edges Belonging to Specified Faces of 3-D Geometry

Find edges belonging to the top and bottom faces of a block.

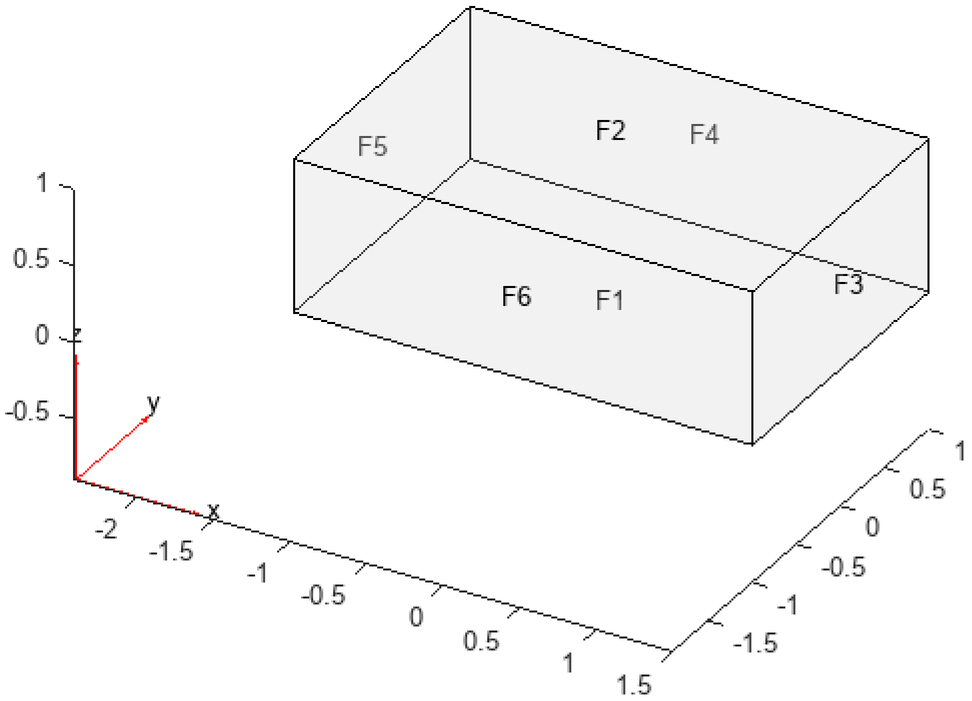
Create a block geometry.

```
gm = multicuboid(3,2,1)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 6
       NumEdges: 12
    NumVertices: 8
       Vertices: [8x3 double]
```

Plot the geometry with the face labels.

```
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.2)
```



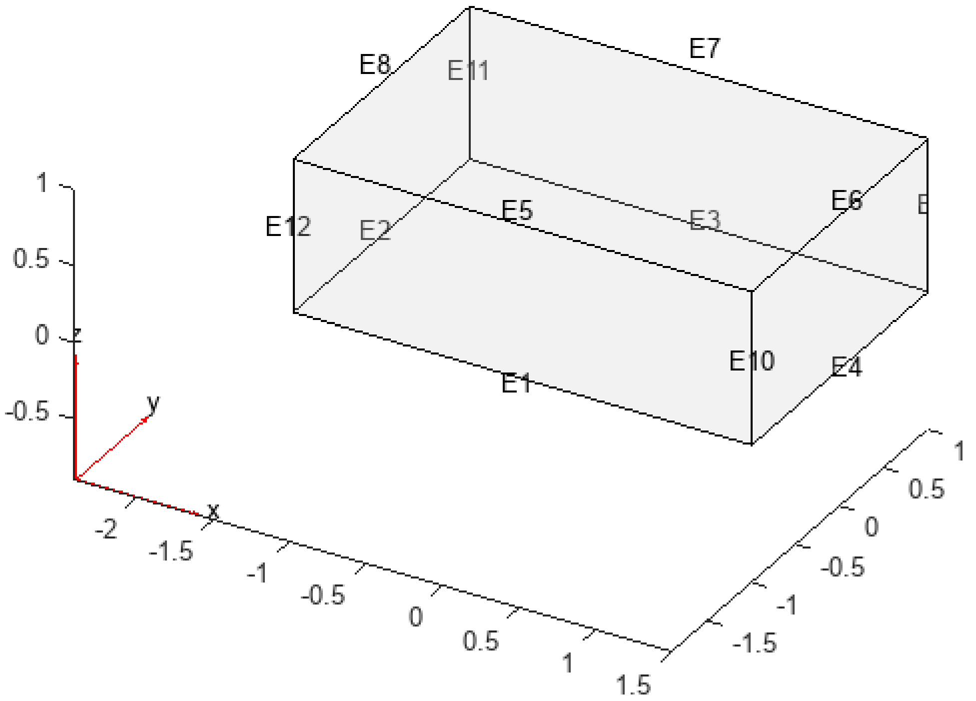
Find edges belonging to faces 1 and 2.

```
edgeIDs = faceEdges(gm,[1 2])
```

```
edgeIDs = 1×8

     1     2     3     4     5     6     7     8
```

Plot the geometry with the edge labels.

```
figure
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.2)
```



Edges Belonging to Specified Faces of 2-D Geometry

Find edges belonging to two faces of the L-shaped membrane.

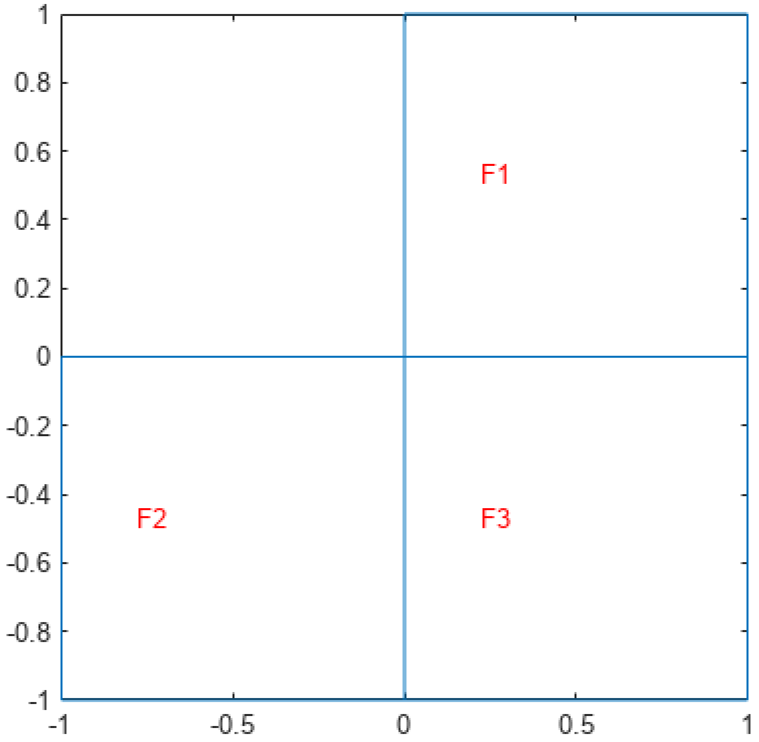
Create a model and include this geometry. The geometry of the L-shaped membrane is described in the file `lshapeg`.

```
model = createpde();
gm = geometryFromEdges(model,@lshapeg)
```

```
gm = 
  AnalyticGeometry with properties:

       NumCells: 0
       NumFaces: 3
       NumEdges: 10
    NumVertices: 8
       Vertices: [8x2 double]
```

Plot the geometry with the face labels.

```
pdegplot(gm,"FaceLabels","on")
```



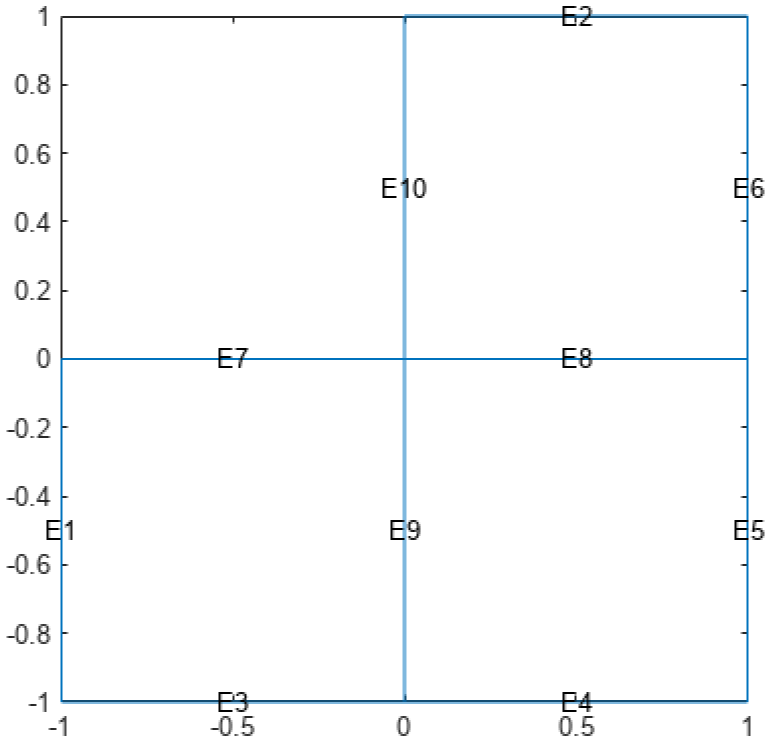
Find edges belonging to faces 1 and 2.

```
edgeIDs = faceEdges(gm,[1 2])
```

```
edgeIDs = 1×8

     1     2     3     6     7     8     9    10
```

Plot the geometry with the edge labels.

```
figure
pdegplot(gm,"EdgeLabels","on")
```



Edges Belonging to Internal and External Faces

Find edges belonging to the side face of the inner cuboid in a geometry consisting of two nested cuboids.

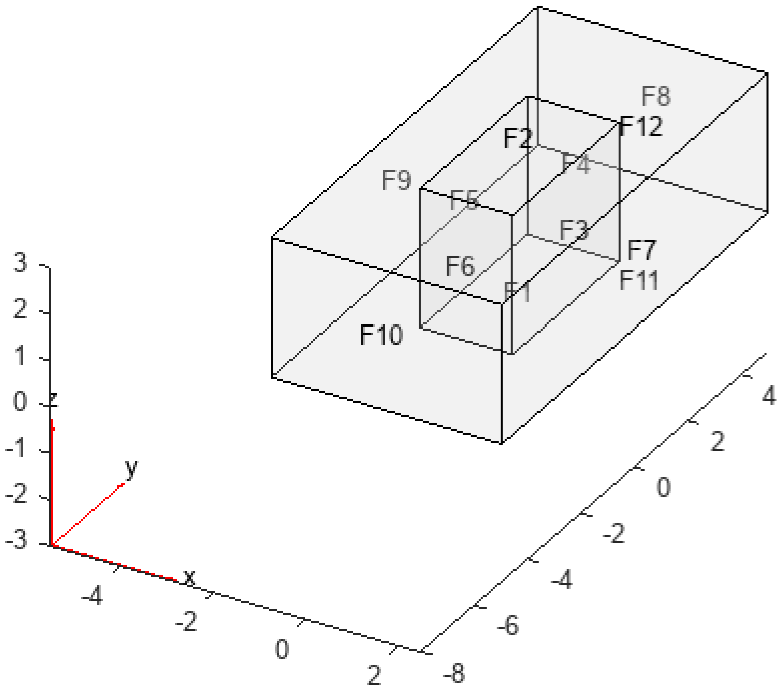
Create a geometry that consists of two nested cuboids of the same height.

```
gm = multicuboid([2 5],[4 10],3)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 2
       NumFaces: 12
       NumEdges: 24
    NumVertices: 16
       Vertices: [16x3 double]
```

Plot the geometry with the face labels.

```
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.2)
```



Find all edges belonging to the side face of the inner cuboid.

```
edgeIDs = faceEdges(gm,6)
```

```
edgeIDs = 1×4

     1     5    10    12
```

From all edges belonging to that face, return the edges belonging to only the internal faces. Internal faces are faces shared between multiple cells.

```
edgeIDs = faceEdges(gm,6,"internal")
```

```
edgeIDs = 1×2

    10    12
```

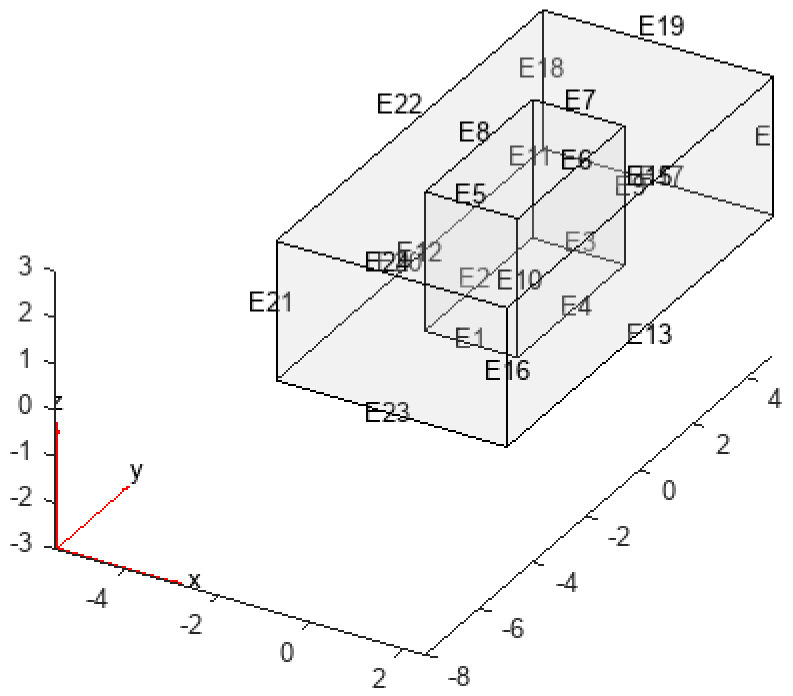
From all edges belonging to that face, return the edges belonging to the external faces.

```
edgeIDs = faceEdges(gm,6,"external")
```

```
edgeIDs = 1×2

     1     5
```

Plot the geometry with the edge labels.

```
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.2)
```



Input Arguments

g — Geometry

DiscreteGeometry object | AnalyticGeometry object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

RegionID — Face ID

positive number | vector of positive numbers

Face ID, specified as a positive number or a vector of positive numbers. Each number represents a face ID.

FilterType — Type of edges to return

`"all"` (default) | `"internal"` | `"external"`

Type of edges to return, specified as `"internal"`, `"external"`, or `"all"`. Depending on this argument, `faceEdges` returns these types of faces for a 3-D geometry:

- `"internal"` — Edges belonging to only internal faces. Internal faces are faces shared between multiple cells.
- `"external"` — Edges belonging to only external faces. External faces are faces not shared between multiple cells.

- `"all"` — All edges belonging to the specified cells.

Output Arguments

`EdgeID` — IDs of edges belonging to specified faces
positive number | vector of positive numbers

IDs of edges belonging to the specified faces, returned as a positive number or a vector of positive numbers.

Version History

Introduced in R2021a

See Also

`cellEdges` | `cellFaces` | `facesAttachedToEdges` | `nearestEdge` | `nearestFace` | DiscreteGeometry Properties | AnalyticGeometry Properties

facesAttachedToEdges

Find faces attached to specified edges

Syntax

```
FaceID = facesAttachedToEdges(g,RegionID)
FaceID = facesAttachedToEdges(g,RegionID,FilterType)
```

Description

`FaceID = facesAttachedToEdges(g,RegionID)` finds faces attached to the edges with ID numbers listed in `RegionID`.

`FaceID = facesAttachedToEdges(g,RegionID,FilterType)` returns internal, external, or all faces attached to the edges with ID numbers listed in `RegionID`. This syntax is valid for 3-D geometries only.

Examples

Faces Attached to Specified Edges of 3-D Geometry

Find faces attached to particular edges of a block.

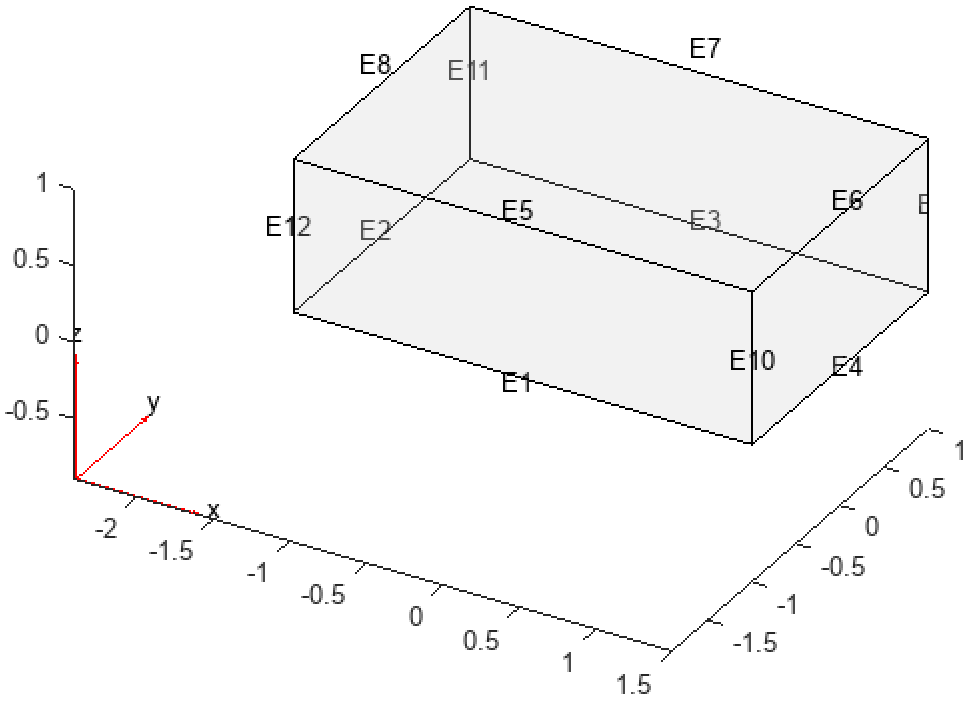
Create a block geometry.

```
gm = multicuboid(3,2,1)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 6
       NumEdges: 12
    NumVertices: 8
       Vertices: [8x3 double]
```

Plot the geometry with the edge labels.

```
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.2)
```



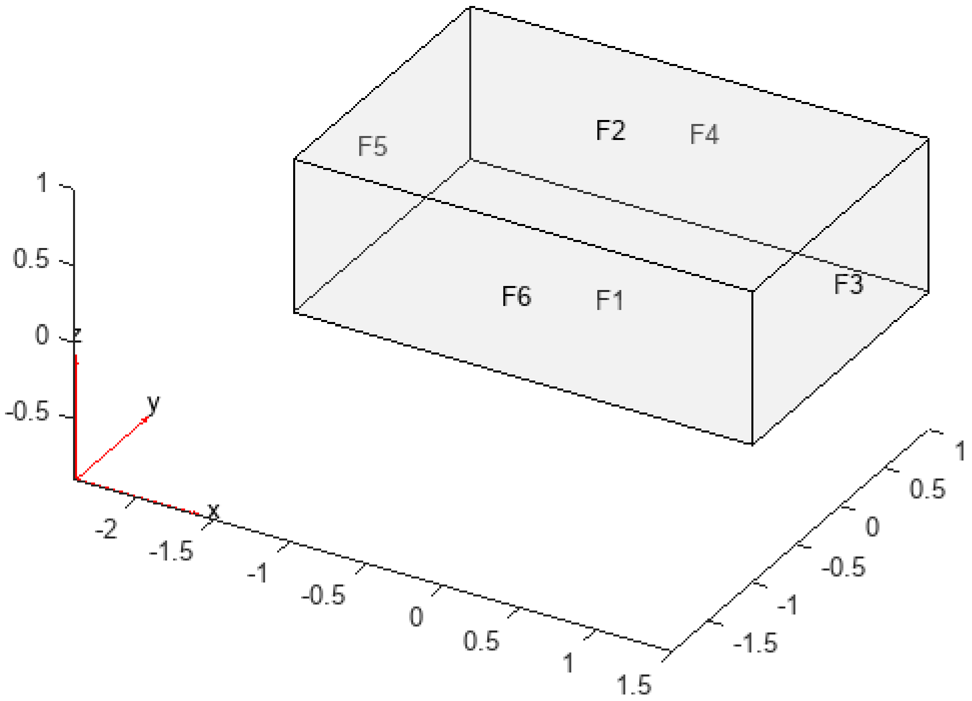
Find faces attached to edges 1, 2, and 5.

```
faceIDs = facesAttachedToEdges(gm,[1 2 5])
```

```
faceIDs = 1×4

     1     2     5     6
```

Plot the geometry with the face labels.

```
figure
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.2)
```



Faces Attached to Specified Edges of 2-D Geometry

Find faces attached to particular edges of the L-shaped membrane.

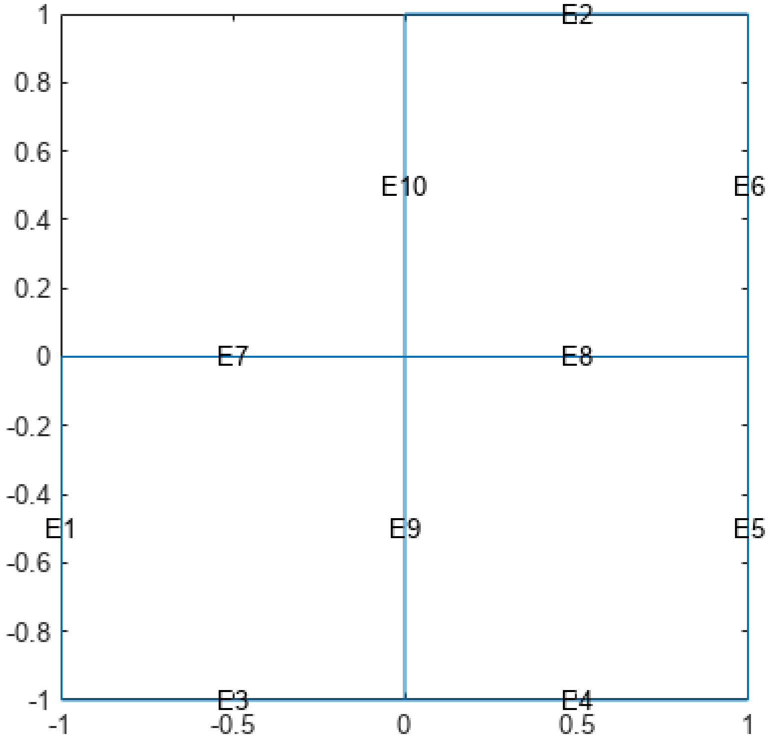
Create a model and include this geometry. The geometry of the L-shaped membrane is described in the file `lshapeg`.

```
model = createpde();
gm = geometryFromEdges(model,@lshapeg)
```

```
gm = 
  AnalyticGeometry with properties:

       NumCells: 0
       NumFaces: 3
       NumEdges: 10
    NumVertices: 8
       Vertices: [8x2 double]
```

Plot the geometry with the edge labels.

```
pdegplot(gm,"EdgeLabels","on")
```



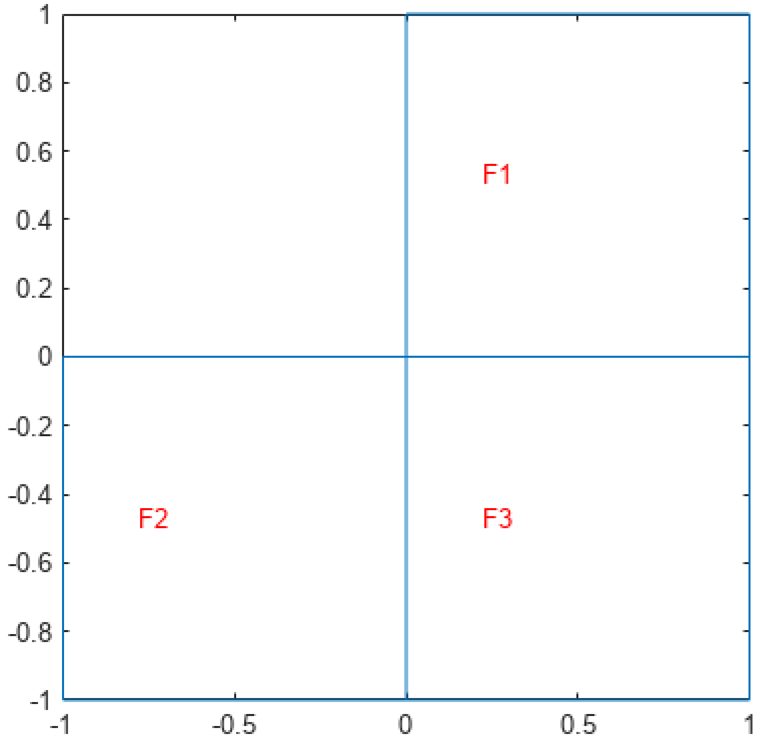
Find faces attached to edges 7 and 10.

```
faceIDs = facesAttachedToEdges(gm,[7 10])
```

```
faceIDs = 1×2

     1     2
```

Plot the geometry with the face labels.

```
figure
pdegplot(gm,"FaceLabels","on")
```



Internal and External Faces Attached to Specified Edges

Find internal and external faces attached to the edges of the inner cuboid in a geometry consisting of two nested cuboids.

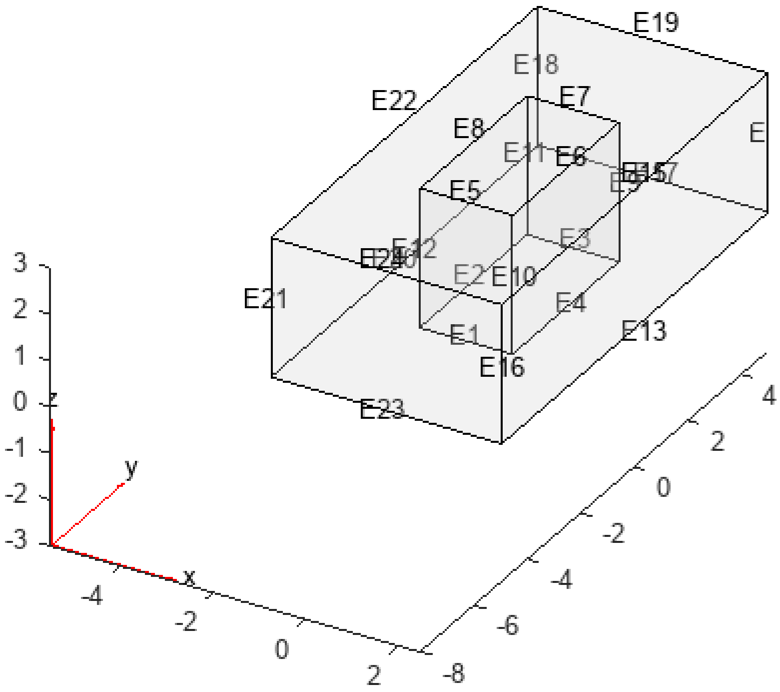
Create a geometry that consists of two nested cuboids of the same height.

```
gm = multicuboid([2 5],[4 10],3)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 2
       NumFaces: 12
       NumEdges: 24
    NumVertices: 16
       Vertices: [16x3 double]
```

Plot the geometry with the edge labels.

```
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.2)
```



Find all faces attached to the top edges of the inner cuboid.

```
facesAttachedToEdges(gm,[5:8])
```

```
ans = 1×6

     2     3     4     5     6    12
```

Find only the internal faces attached to the top edges of the inner cuboid. Internal faces are faces shared between multiple cells.

```
facesAttachedToEdges(gm,[5:8],"internal")
```

```
ans = 1×4

     3     4     5     6
```

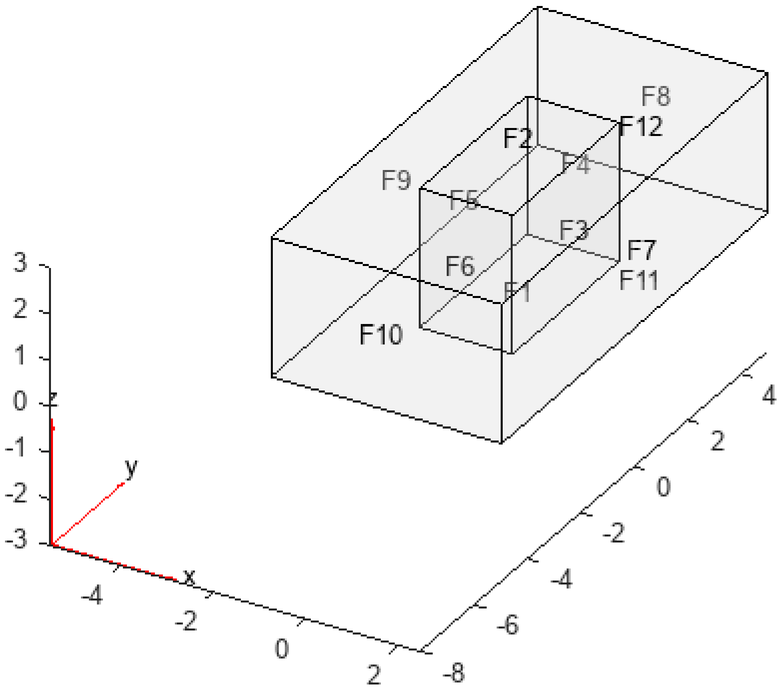
Find only the external faces attached to the top edges of the inner cuboid.

```
facesAttachedToEdges(gm,[5:8],"external")
```

```
ans = 1×2

     2    12
```

Plot the geometry with the face labels.

```
figure
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.2)
```



Input Arguments

g — Geometry

`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

`RegionID` — Edge ID

positive number | vector of positive numbers

Edge ID, specified as a positive number or a vector of positive numbers. Each number represents an edge ID.

`FilterType` — Type of faces to return

`"all"` (default) | `"internal"` | `"external"`

Type of faces to return, specified as `"internal"`, `"external"`, or `"all"`. Depending on this argument, `facesAttachedToEdges` returns these types of faces for a 3-D geometry:

- `"internal"` — Internal faces, that is, faces shared between multiple cells.
- `"external"` — External faces, that is, faces not shared between multiple cells.

- `"all"` — All faces attached to the specified cells.

Output Arguments

`FaceID` — IDs of faces attached to specified edges
positive number | vector of positive numbers

IDs of faces attached to the specified edges, returned as a positive number or a vector of positive numbers.

Version History

Introduced in R2021a

See Also

`cellEdges` | `cellFaces` | `faceEdges` | `nearestEdge` | `nearestFace` | DiscreteGeometry Properties | AnalyticGeometry Properties

findBodyLoad

Package: `pde`

Find body load assigned to geometric region

Syntax

```
bl = findBodyLoad(structuralmodel.BodyLoads,RegionType,RegionID)
```

Description

`bl = findBodyLoad(structuralmodel.BodyLoads,RegionType,RegionID)` returns the body load assigned to a geometric region of the structural model. A body load must use units consistent with the geometry and other model attributes.

Examples

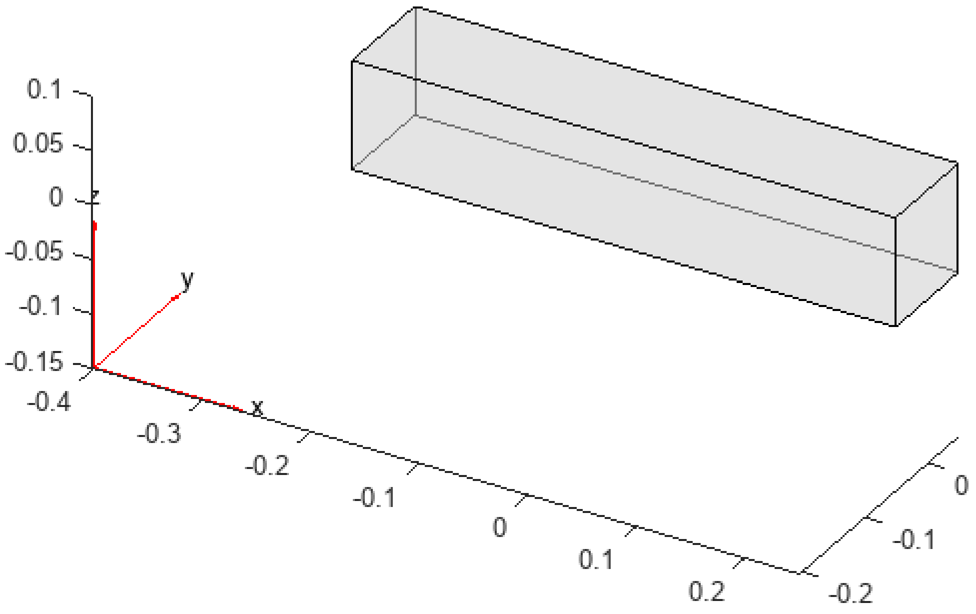
Find Body Load

Create a structural model.

```
structuralModel = createpde("structural","static-solid");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.1,0.1);
structuralModel.Geometry = gm;
pdegplot(structuralModel,"FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density. Notice that the mass density value is required for modeling gravitational effects.

```
structuralProperties(structuralModel,"YoungsModulus",210E3, ...
                                     "PoissonsRatio",0.3,...
                                     "MassDensity",2.7E-6);
```

Specify the gravity load on the beam.

```
structuralBodyLoad(structuralModel, ...
                  "GravitationalAcceleration",[0;0;-9.8]);
```

Check the body load specification for cell 1.

```
findBodyLoad(structuralModel.BodyLoads,"Cell",1)
```

```
ans =
  BodyLoadAssignment with properties:

                   RegionType: 'Cell'
                     RegionID: 1
    GravitationalAcceleration: [3x1 double]
              AngularVelocity: []
                  Temperature: []
                     TimeStep: []
                        Label: []
```

Input Arguments

`structuralmodel.BodyLoads` — Body loads

`BodyLoads` property of `StructuralModel` object

Body loads of the model, specified as a `BodyLoads` property of a `StructuralModel` object.

`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: `findBodyLoad(structuralmodel.BodyLoads,"Cell",1)`

Data Types: `char` | `string`

`RegionID` — Geometric region ID

vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `findBodyLoad(structuralmodel.BodyLoads,"Cell",1)`

Data Types: `double`

Output Arguments

`bl` — Body load assignment

`BodyLoadAssignment` object

Body load assignment, returned as a `BodyLoadAssignment` object. For details, see BodyLoadAssignment Properties.

Version History

Introduced in R2017b

See Also

`structuralBodyLoad`

findBoundaryConditions

Package: `pde`

Find boundary condition assignment for a geometric region

Syntax

`BCregion = findBoundaryConditions(BCs,RegionType,RegionID)`

Description

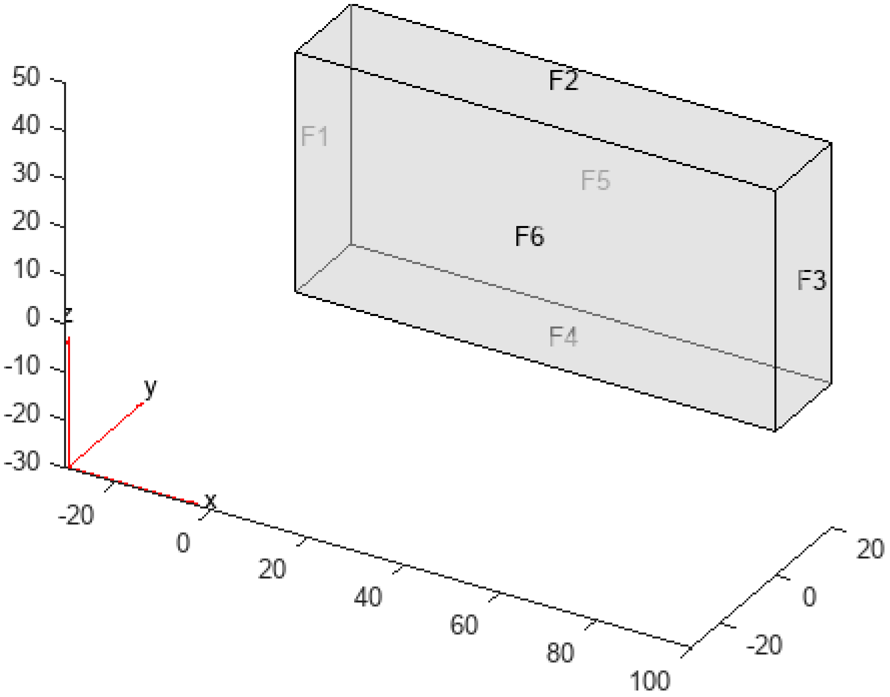
`BCregion = findBoundaryConditions(BCs,RegionType,RegionID)` returns boundary condition `BCregion` assigned to the specified region.

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: [3x1 double]
                h: [3x3 double]
                g: [3x1 double]
                q: [3x3 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: [3x1 double]
                q: []
                u: []
    EquationIndex: []
       Vectorized: 'off'
```

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: `findBoundaryConditions(model.BoundaryConditions,"Face",3)`

Data Types: `char` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot` with the `"FaceLabels"` (3-D) or `"EdgeLabels"` (2-D) value set to `"on"`.

Example: `findBoundaryConditions(model.BoundaryConditions,"Face",3)`

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.

Version History

Introduced in R2016b

See Also

`applyBoundaryCondition` | BoundaryCondition

Topics
“Solve Problems Using PDEModel Objects” on page 2-2

findCoefficients

Package: `pde`

Locate active PDE coefficients

Syntax

`CA = findCoefficients(coeffs,RegionType,RegionID)`

Description

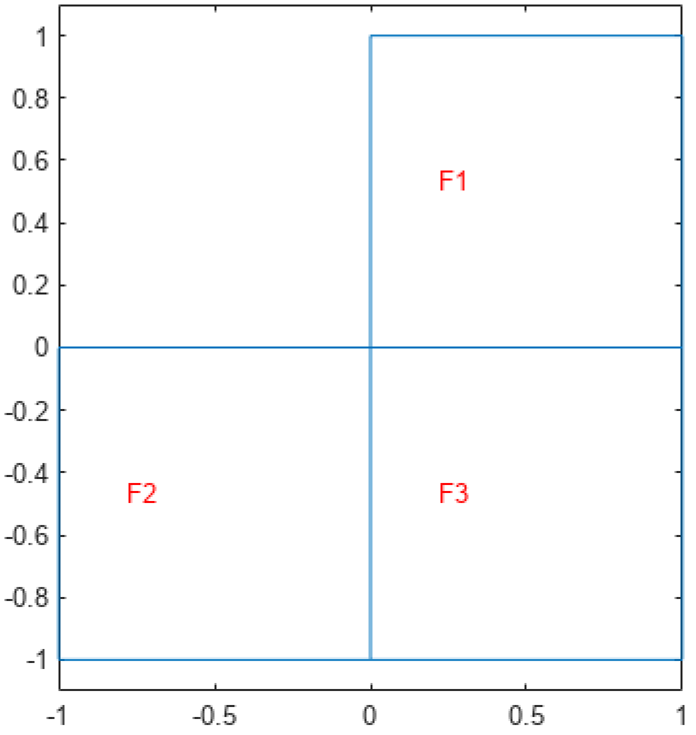
`CA = findCoefficients(coeffs,RegionType,RegionID)` returns the active coefficient assignment `CA` for the coefficients in the specified region.

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
```

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` | `string`

`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.

Version History

Introduced in R2016a

See Also

CoefficientAssignment | `specifyCoefficients`

Topics

"View, Edit, and Delete PDE Coefficients" on page 2-111
"Solve Problems Using PDEModel Objects" on page 2-2

findElectromagneticBC

Package: `pde`

Find electromagnetic boundary conditions assigned to geometric region

Syntax

```
emBC = findElectromagneticBC(emagmodel.BoundaryConditions,RegionType,
RegionID)
```

Description

`emBC = findElectromagneticBC(emagmodel.BoundaryConditions,RegionType, RegionID)` returns the boundary conditions assigned to the specified region of the specified model.

Examples

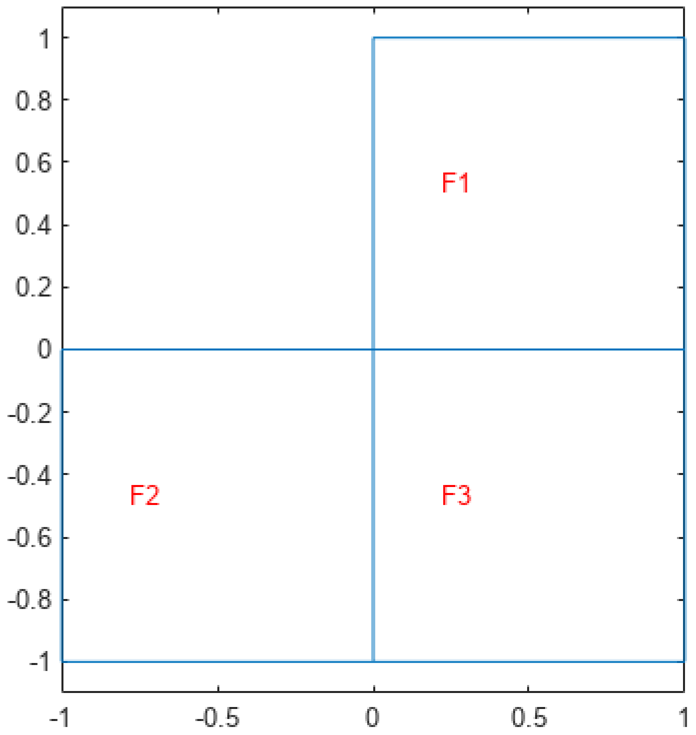
Find Electromagnetic Boundary Conditions for Edges of 2-D Geometry

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Include an L-shaped membrane geometry in the model and plot it with the face labels.

```
geometryFromEdges(emagmodel,@lshapeg);
pdegplot(emagmodel,"FaceLabels","on")
ylim([-1.1 1.1])
axis equal
```



Assign magnetic potential values to edges 1 and 2.

```
electromagneticBC(emagmodel,"Edge",1,"MagneticPotential",1);
electromagneticBC(emagmodel,"Edge",2,"MagneticPotential",0);
```

Check the boundary condition specifications for edges 1 and 2.

```
emBC = findElectromagneticBC(emagmodel.BoundaryConditions,"Edge",1:2);
emBC(1)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

             RegionID: 1
           RegionType: 'Edge'
           Vectorized: 'off'
    MagneticPotential: 1
```

```
emBC(2)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

             RegionID: 2
           RegionType: 'Edge'
           Vectorized: 'off'
    MagneticPotential: 0
```

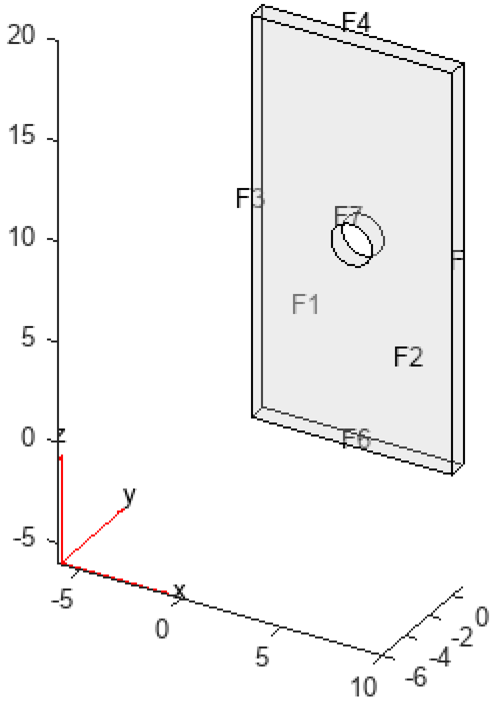
Find Electromagnetic Boundary Conditions for Faces of 3-D Geometry

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Apply the voltage boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
electromagneticBC(emagmodel,"Voltage",1000,"Face",7);
```

Check the boundary condition specifications for faces 4, 5, and 7.

```
emBC = findElectromagneticBC(emagmodel.BoundaryConditions, ...
                             "Face",[4 5 7]);
emBC(1)
```

```
ans = 
  ElectromagneticBCAssignment with properties:
```

```
      RegionID: [3 4 5 6]
    RegionType: 'Face'
    Vectorized: 'off'
       Voltage: 0
```

```
emBC(2)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

      RegionID: [3 4 5 6]
    RegionType: 'Face'
    Vectorized: 'off'
       Voltage: 0
```

```
emBC(3)
```

```
ans = 
  ElectromagneticBCAssignment with properties:

      RegionID: 7
    RegionType: 'Face'
    Vectorized: 'off'
       Voltage: 1000
```

Input Arguments

`emagmodel.BoundaryConditions` — Boundary conditions of electromagnetic model
`BoundaryConditions` property

Boundary conditions of an electromagnetic model, specified as the `BoundaryConditions` property of the model.

Example: `findElectromagneticBC(emagmodel.BoundaryConditions,"Edge",1)`

RegionType — Geometric region type
`"Edge"` for a 2-D model | `"Face"` for a 3-D model

Geometric region type, specified as `"Edge"` for a 2-D model or `"Face"` for a 3-D model.

Data Types: `char` | `string`

RegionID — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. Find the edge or face IDs by using `pdegplot` with the `"EdgeLabels"` or `"FaceLabels"` name-value argument set to `"on"`.

Data Types: `double`

Output Arguments

emBC — Electromagnetic boundary condition assignment
`ElectromagneticBCAssignment` object

Electromagnetic boundary condition assignment, returned as an `ElectromagneticBCAssignment` object. For more information, see ElectromagneticBCAssignment Properties.

Version History

Introduced in R2021a

See Also

`ElectromagneticModel` | `electromagneticBC` | ElectromagneticBCAssignment Properties

findElectromagneticProperties

Package: `pde`

Find electromagnetic material properties assigned to geometric region

Syntax

```
emProperties = findElectromagneticProperties(emagmodel.MaterialProperties,
RegionType,RegionID)
```

Description

`emProperties = findElectromagneticProperties(emagmodel.MaterialProperties, RegionType,RegionID)` returns the electromagnetic material properties assigned to the specified region of the specified model.

Examples

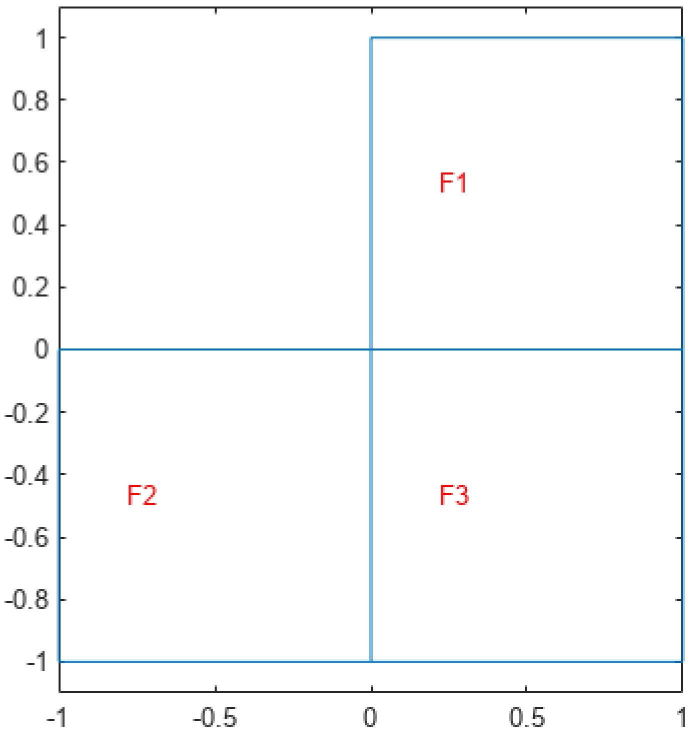
Find Relative Permittivity for Faces of 2-D Geometry

Create an electromagnetic model for an electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Include the L-shaped membrane geometry in the model and plot it with the face labels.

```
geometryFromEdges(emagmodel,@lshapeg);
pdegplot(emagmodel,"FaceLabels","on")
ylim([-1.1 1.1])
axis equal
```



Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify a different value of relative permittivity for each face.

```
electromagneticProperties(emagmodel,"RelativePermittivity",2.5, ...
                                    "Face",1);
electromagneticProperties(emagmodel,"RelativePermittivity",2.25, ...
                                    "Face",2);
electromagneticProperties(emagmodel,"RelativePermittivity",1, ...
                                    "Face",3);
```

Check the electromagnetic material properties specification for each face.

```
findElectromagneticProperties(emagmodel.MaterialProperties,"Face",1)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

               RegionType: 'Face'
                 RegionID: 1
    RelativePermittivity: 2.5000
    RelativePermeability: []
            Conductivity: []
```

```
findElectromagneticProperties(emagmodel.MaterialProperties,"Face",2)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

               RegionType: 'Face'
                 RegionID: 2
    RelativePermittivity: 2.2500
    RelativePermeability: []
            Conductivity: []
```

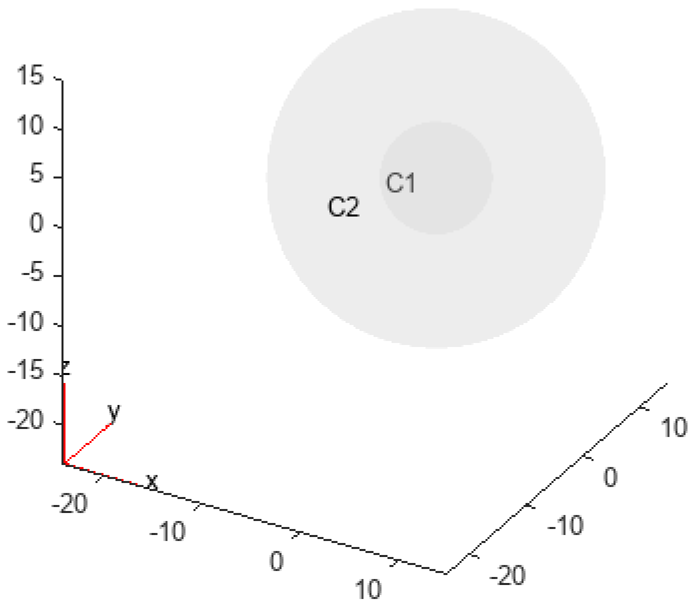
```
findElectromagneticProperties(emagmodel.MaterialProperties,"Face",3)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

               RegionType: 'Face'
                 RegionID: 3
    RelativePermittivity: 1
    RelativePermeability: []
            Conductivity: []
```

Find Relative Permeability for Cells of 3-D Geometry

Create and plot a geometry consisting of two nested spheres.

```
gm = multisphere([5 15]);
pdegplot(gm,"CellLabels","on","FaceAlpha",0.3)
```



Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Include the geometry in the model.

```
emagmodel.Geometry = gm;
```

Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify a different value of relative permittivity for each cell.

```
electromagneticProperties(emagmodel,"RelativePermittivity",2.5, ...
                                    "Cell",1);
electromagneticProperties(emagmodel,"RelativePermittivity",2.25, ...
                                    "Cell",2);
```

Check the electromagnetic material properties specification for each cell.

```
findElectromagneticProperties(emagmodel.MaterialProperties,"Cell",1)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

              RegionType: 'Cell'
                RegionID: 1
```

```
    RelativePermittivity: 2.5000
    RelativePermeability: []
            Conductivity: []
```

```
findElectromagneticProperties(emagmodel.MaterialProperties,"Cell",2)
```

```
ans = 
  ElectromagneticMaterialAssignment with properties:

              RegionType: 'Cell'
                RegionID: 2
    RelativePermittivity: 2.2500
    RelativePermeability: []
            Conductivity: []
```

Input Arguments

`emagmodel.MaterialProperties` — Material properties of electromagnetic model

`MaterialProperties` property

Material properties of an electromagnetic model, specified as the `MaterialProperties` property of the model.

Example: `emagmodel.MaterialProperties`

`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 geometry or `"Cell"` for a 3-D geometry.

Data Types: `char` | `string`

`RegionID` — Region ID

vector of positive integers

Region ID, specified as a vector of positive integers. Find the face or cell IDs by using `pdegplot` with the `"FaceLabels"` or `"CellLabels"` name-value argument set to `"on"`.

Data Types: `double`

Output Arguments

`emProperties` — Material properties assignment

`ElectromagneticMaterialAssignment` object

Material properties assignment, returned as an `ElectromagneticMaterialAssignment` object. For more information, see ElectromagneticMaterialAssignment Properties.

Version History

Introduced in R2021a

See Also

`electromagneticProperties` | `ElectromagneticModel` | ElectromagneticMaterialAssignment Properties

findElectromagneticSource

Package: `pde`

Find electromagnetic source assigned to geometric region

Syntax

`emSource = findElectromagneticSource(emagmodel.Sources,RegionType,RegionID)`

Description

`emSource = findElectromagneticSource(emagmodel.Sources,RegionType,RegionID)` returns the change or current density `emSource` assigned to the specified region of the specified model.

Examples

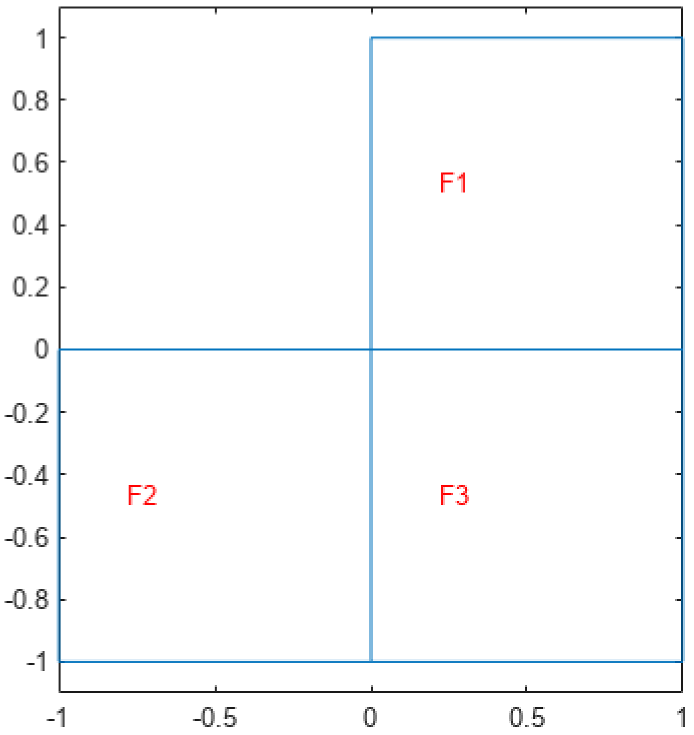
Find Current Density for Faces of 2-D Geometry

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Include the L-shaped membrane geometry in the model and plot it with the face labels.

```
geometryFromEdges(emagmodel,@lshapeg);
pdegplot(emagmodel,"FaceLabels","on")
ylim([-1.1 1.1])
axis equal
```



Specify a different current density for each face.

```
electromagneticSource(emagmodel,"Face",1,"CurrentDensity",10);
electromagneticSource(emagmodel,"Face",2,"CurrentDensity",20);
electromagneticSource(emagmodel,"Face",3,"CurrentDensity",30);
```

Check the electromagnetic source specification for each face.

```
findElectromagneticSource(emagmodel.Sources,"Face",1)
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

         RegionType: 'Face'
           RegionID: 1
      ChargeDensity: []
     CurrentDensity: 10
      Magnetization: []
```

```
findElectromagneticSource(emagmodel.Sources,"Face",2)
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

         RegionType: 'Face'
           RegionID: 2
      ChargeDensity: []
```

```
    CurrentDensity: 20
     Magnetization: []
```

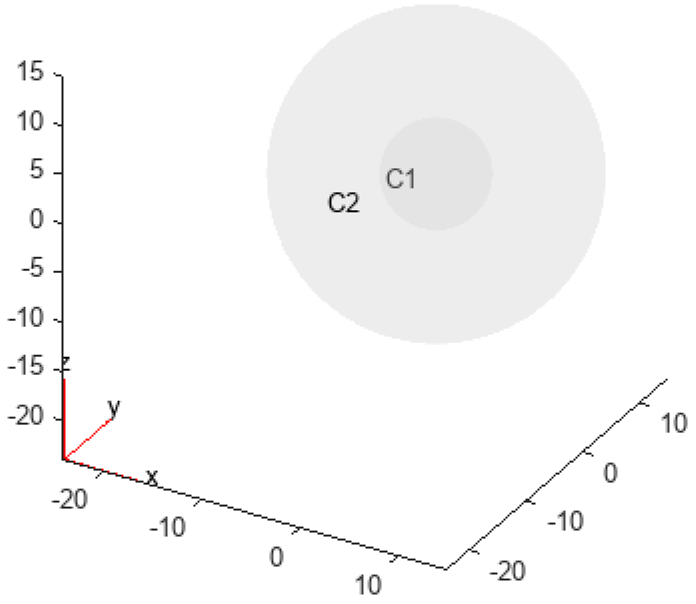
```
findElectromagneticSource(emagmodel.Sources,"Face",3)
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

        RegionType: 'Face'
          RegionID: 3
     ChargeDensity: []
    CurrentDensity: 30
     Magnetization: []
```

Find Charge Density for Cells of 3-D Geometry

Create and plot a geometry consisting of two nested spheres.

```
gm = multisphere([5 15]);
pdegplot(gm,"CellLabels","on","FaceAlpha",0.3)
```



Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Include the geometry in the model.

```
emagmodel.Geometry = gm;
```

Specify the charge density for the inner sphere.

```
electromagneticSource(emagmodel,"Cell",1,"ChargeDensity",10);
```

Check the electromagnetic source specification for each cell.

```
findElectromagneticSource(emagmodel.Sources,"Cell",1)
```

```
ans = 
  ElectromagneticSourceAssignment with properties:

        RegionType: 'Cell'
          RegionID: 1
     ChargeDensity: 10
    CurrentDensity: []
     Magnetization: []
```

```
findElectromagneticSource(emagmodel.Sources,"Cell",2)
```

```
ans = 

  0x1 ElectromagneticSourceAssignment array with properties:

    RegionType
    RegionID
    ChargeDensity
    CurrentDensity
    Magnetization
```

Input Arguments

`emagmodel.Sources` — Source in electromagnetic model

`Sources` property

Source in an electromagnetic model, specified as the `Sources` property of the model.

Example: `findElectromagneticSource(emagmodel.Sources,"Face",1)`

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.

Data Types: `char` | `string`

RegionID — Region ID

vector of positive integers

Region ID, specified as a vector of positive integers. Find the face or cell IDs by using `pdegplot` with the `"FaceLabels"` or `"CellLabels"` name-value argument set to `"on"`.

Data Types: `double`

Output Arguments

`emSource` — Electromagnetic source assignment
`ElectromagneticSourceAssignment` object

Electromagnetic source assignment, returned as an `ElectromagneticSourceAssignment` object. For more information, see ElectromagneticSourceAssignment Properties.

Version History

Introduced in R2021a

See Also

`ElectromagneticModel` | ElectromagneticSourceAssignment Properties | `electromagneticSource`

findElements

Package: pde

Find mesh elements in specified region

Syntax

```
elemIDs = findElements(mesh,"region",RegionType,RegionID)
elemIDs = findElements(mesh,"box",xlim,ylim)
elemIDs = findElements(mesh,"box",xlim,ylim,zlim)
elemIDs = findElements(mesh,"radius",center,radius)
elemIDs = findElements(mesh,"attached",nodeID)
```

Description

`elemIDs = findElements(mesh,"region",RegionType,RegionID)` returns the IDs of the mesh elements that belong to the specified geometric region.

`elemIDs = findElements(mesh,"box",xlim,ylim)` returns the IDs of the mesh elements within a bounding box specified by `xlim` and `ylim`. Use this syntax for 2-D meshes.

`elemIDs = findElements(mesh,"box",xlim,ylim,zlim)` returns the IDs of the mesh elements located within a bounding box specified by `xlim`, `ylim`, and `zlim`. Use this syntax for 3-D meshes.

`elemIDs = findElements(mesh,"radius",center,radius)` returns the IDs of mesh elements located within a circle (for 2-D meshes) or sphere (for 3-D meshes) specified by `center` and `radius`.

`elemIDs = findElements(mesh,"attached",nodeID)` returns the IDs of the mesh elements attached to the specified node. Here, `nodeID` is the ID of a corner node. This syntax ignores the IDs of the nodes located in the middle of element edges.

For multiple nodes, specify `nodeID` as a vector.

Examples

Elements Associated with Particular Face

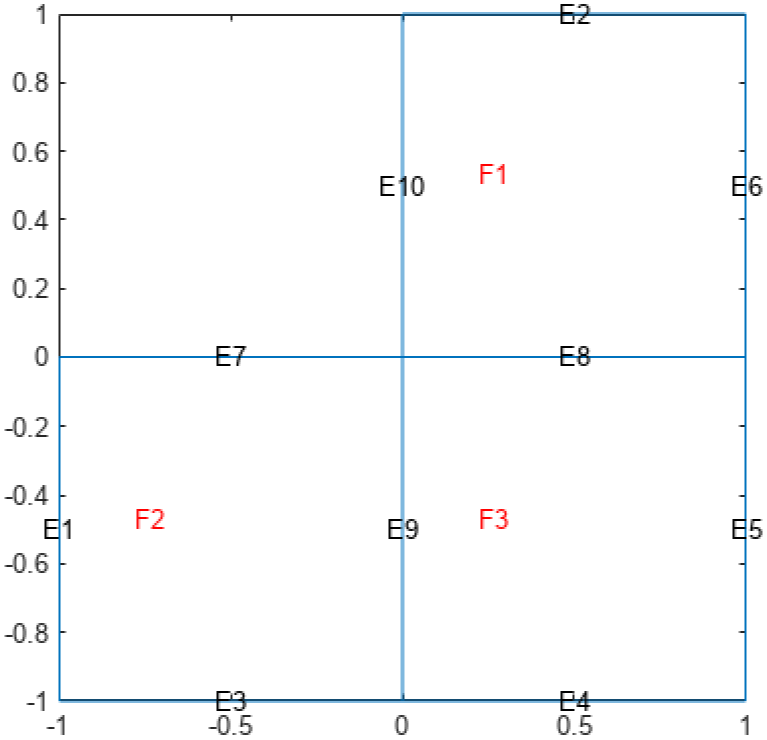
Find the elements associated with a geometric region.

Create a PDE model.

```
model = createpde;
```

Include the geometry of the built-in function `lshapeg`. Plot the geometry.

```
geometryFromEdges(model,@lshapeg);
pdegplot(model,"FaceLabels","on","EdgeLabels","on")
```



Generate a mesh.

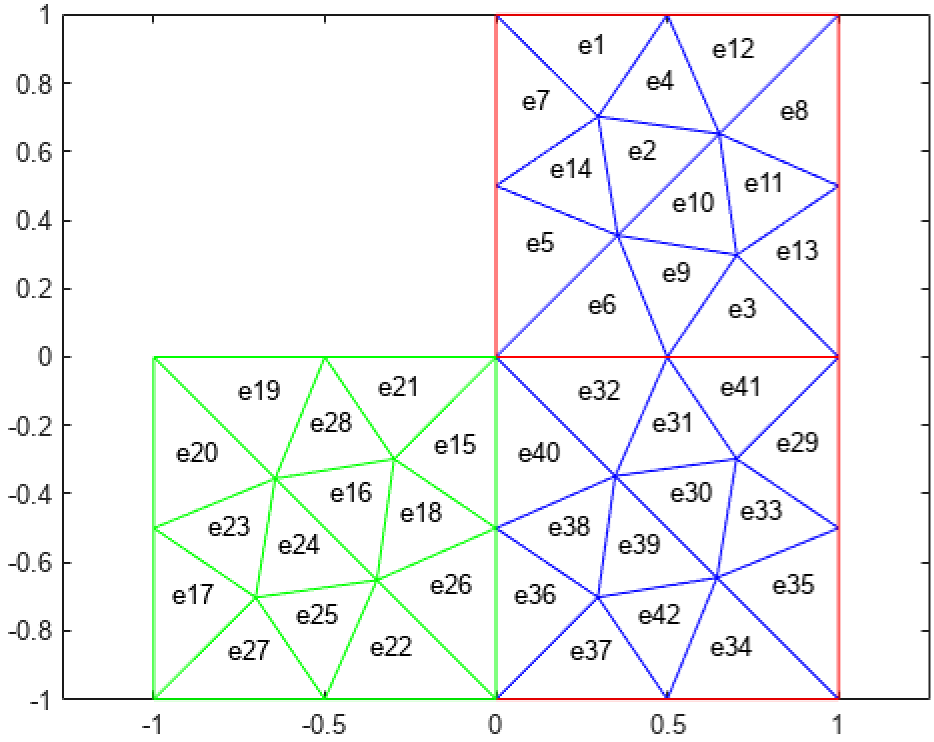
```
mesh = generateMesh(model,"Hmax",0.5);
```

Find the elements associated with face 2.

```
Ef2 = findElements(mesh,"region","Face",2);
```

Highlight these elements in green on the mesh plot.

```
figure
pdemesh(mesh,"ElementLabels","on")
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,Ef2),"EdgeColor","green")
```



Elements Within Bounding Box

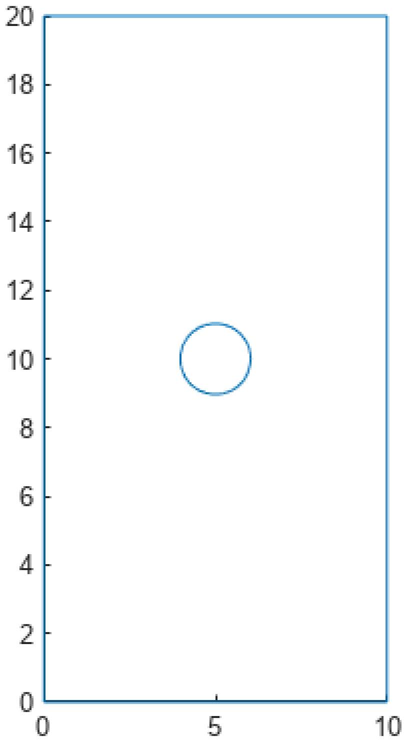
Find the elements located within a specified box.

Create a PDE model.

```
model = createpde;
```

Import and plot the geometry.

```
importGeometry(model,"PlateHolePlanar.stl");
pdegplot(model)
```



Generate a mesh.

```
mesh = generateMesh(model,"Hmax",2,"Hmin",0.4)
```

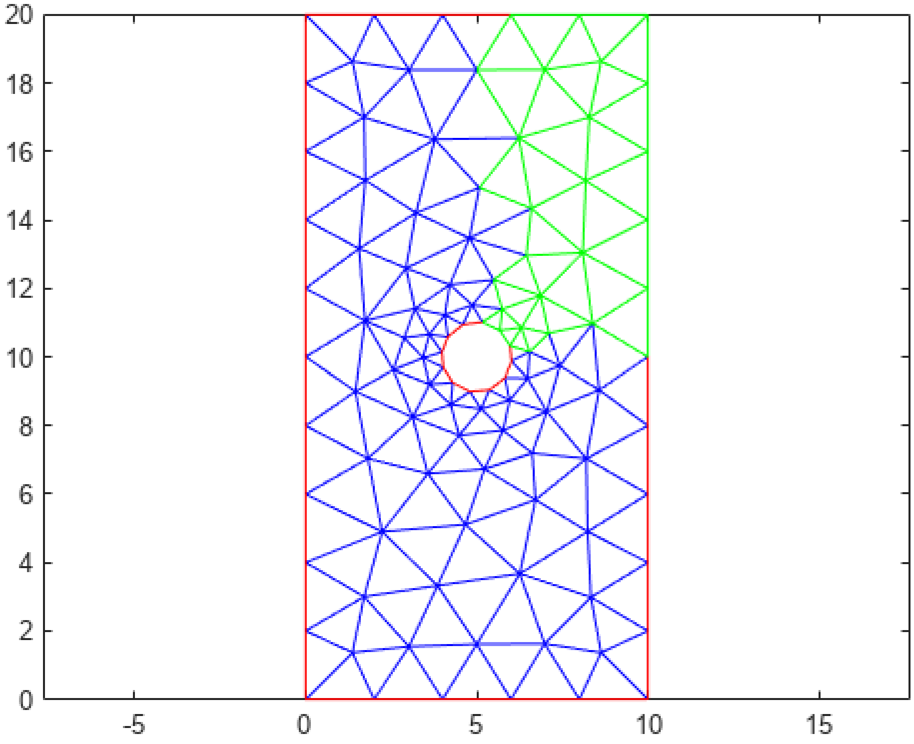
```
mesh = 
  FEMesh with properties:

             Nodes: [2x386 double]
          Elements: [6x172 double]
    MaxElementSize: 2
    MinElementSize: 0.4000
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

Find the elements located within the following box.

```
Eb = findElements(mesh,"box",[5 10],[10 20]);
```

Highlight these elements in green on the mesh plot.

```
figure
pdemesh(model)
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,Eb),"EdgeColor","green")
```



Elements Within Bounding Disk

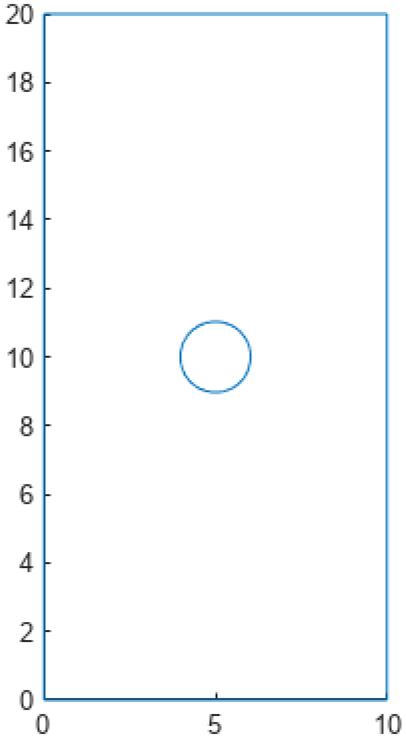
Find the elements located within a specified disk.

Create a PDE model.

```
model = createpde;
```

Import and plot the geometry.

```
importGeometry(model,"PlateHolePlanar.stl");
pdegplot(model)
```



Generate a mesh.

```
mesh = generateMesh(model,"Hmax",2,"Hmin",0.4,"GeometricOrder","linear")
```

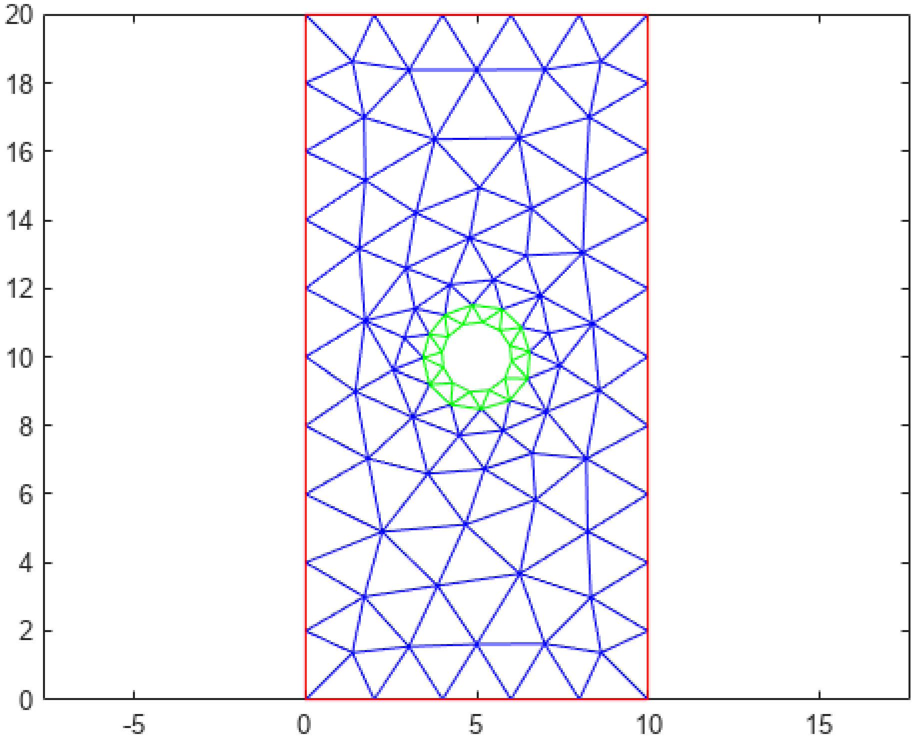
```
mesh = 
  FEMesh with properties:

             Nodes: [2x107 double]
          Elements: [3x172 double]
    MaxElementSize: 2
    MinElementSize: 0.4000
     MeshGradation: 1.5000
    GeometricOrder: 'linear'
```

Find the elements located within radius 2 from the center [5,10].

```
Er = findElements(mesh,"radius",[5 10],2);
```

Highlight these elements in green on the mesh plot.

```
figure
pdemesh(model)
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,Er),"EdgeColor","green")
```



Elements Attached to Specified Nodes

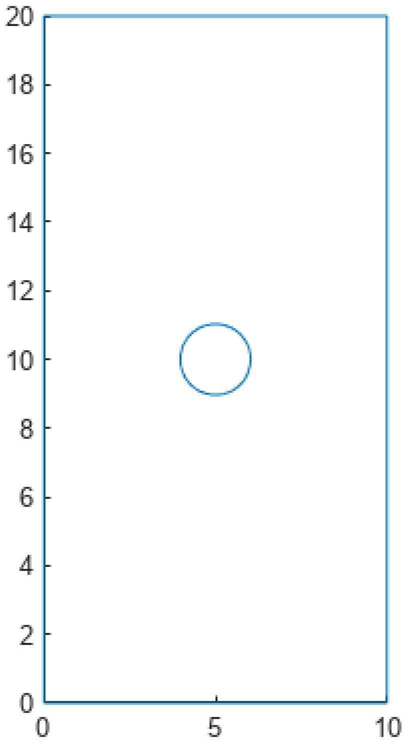
Find the elements attached to a specified corner node.

Create a PDE model.

```
model = createpde;
```

Import and plot the geometry.

```
importGeometry(model,"PlateHolePlanar.stl");
pdegplot(model)
```



Generate a linear triangular mesh by setting the geometric order value to `linear`. This mesh contains only corner nodes.

```
mesh = generateMesh(model,"Hmax",2,"Hmin",0.4, ...
                          "GeometricOrder","linear");
```

Find the node closest to the point [15;10].

```
N_ID = findNodes(mesh,"nearest",[15;10])
```

```
N_ID = 10
```

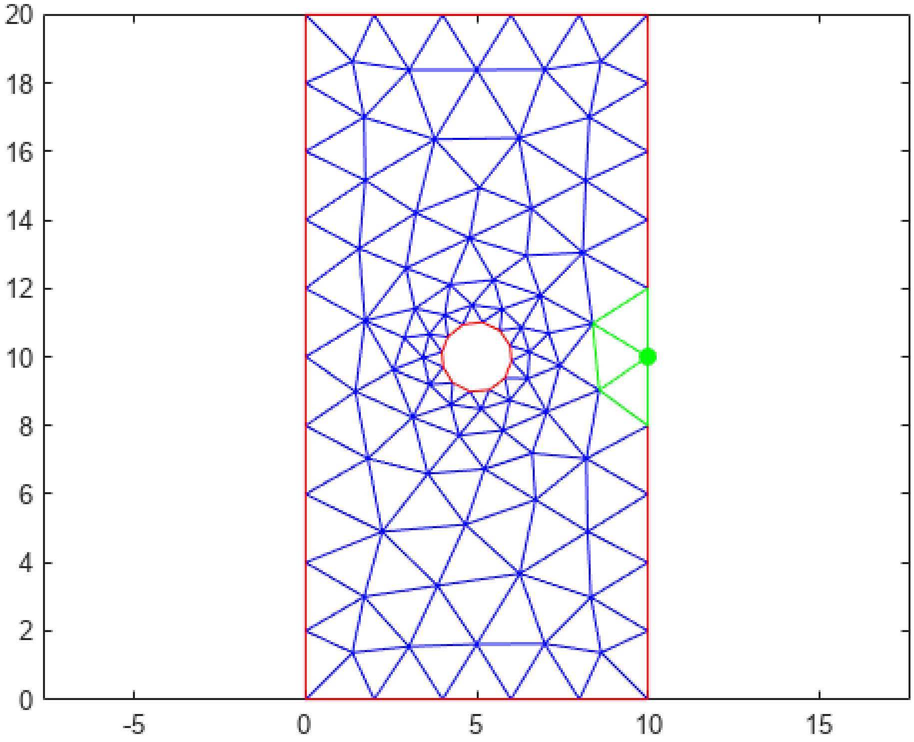
Find the elements attached to this node.

```
En = findElements(mesh,"attached",N_ID)
```

```
En = 1×3

    95    97    98
```

Highlight the node and the elements in green on the mesh plot.

```
figure
pdemesh(model)
hold on
plot(mesh.Nodes(1,N_ID),mesh.Nodes(2,N_ID),"or","Color","g", ...
                                          "MarkerFaceColor","g")
pdemesh(mesh.Nodes,mesh.Elements(:,En),"EdgeColor","green")
```



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`

`RegionType` — Geometric region type

`"Cell"` for a 3-D model | `"Face"` for a 2-D model

Geometric region type, specified as `"Cell"` or `"Face"`.

Example: `findElements(mesh,"region","Face",1:3)`

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 by using `pdegplot`.

Example: `findElements(mesh,"region","Face",1:3)`

Data Types: `double`

`xlim` — *x*-limits of bounding box

two-element row vector

x-limits of the bounding box, specified as a two-element row vector. The first element of `xlim` is the lower *x*-bound, and the second element is the upper *x*-bound.

Example: `findElements(mesh,"box",[5 10],[10 20])`

Data Types: `double`

`ylim` — *y*-limits of bounding box

two-element row vector

y-limits of the bounding box, specified as a two-element row vector. The first element of `ylim` is the lower *y*-bound, and the second element is the upper *y*-bound.

Example: `findElements(mesh,"box",[5 10],[10 20])`

Data Types: `double`

`zlim` — *z*-limits of bounding box

two-element row vector

z-limits of the bounding box, specified as a two-element row vector. The first element of `zlim` is the lower *z*-bound, and the second element is the upper *z*-bound. You can specify `zlim` only for 3-D meshes.

Example: `findElements(mesh,"box",[5 10],[10 20],[1 2])`

Data Types: `double`

`center` — Center of bounding circle or sphere

two-element row vector for a 2-D mesh | three-element row vector for a 3-D mesh

Center of the bounding circle or sphere, specified as a two-element row vector for a 2-D mesh or three-element row vector for a 3-D mesh. The elements of these vectors contain the coordinates of the center of a circle or a sphere.

Example: `findElements(mesh,"radius",[0 0 0],0.5)`

Data Types: `double`

`radius` — Radius of bounding circle or sphere

positive number

Radius of the bounding circle or sphere, specified as a positive number.

Example: `findElements(mesh,"radius",[0 0 0],1)`

Data Types: `double`

`nodeID` — ID of corner node of element

positive integer | vector of positive integers

ID of the corner node of the element, specified as a positive integer or a vector of positive integers. The `findElements` function can find an ID of the element by the ID of the corner node of the element. The function ignores IDs of the nodes located in the middle of element edges. For multiple nodes, specify `nodeID` as a vector.

Example: `findElements(mesh,"attached",[7 8 21])`

Data Types: `double`

Output Arguments

`elemIDs` — Element IDs
positive integer | row vector of positive integers

Element IDs, returned as a positive integer or a row vector of positive integers.

Version History

Introduced in R2018a

See Also

`findNodes` | `meshQuality` | `area` | `volume` | FEMesh Properties

Topics

“Finite Element Method Basics” on page 1-11

findHeatSource

Package: pde

Find heat source assigned to a geometric region

Syntax

```
hsa = findHeatSource(thermalmodel.HeatSources,RegionType,RegionID)
```

Description

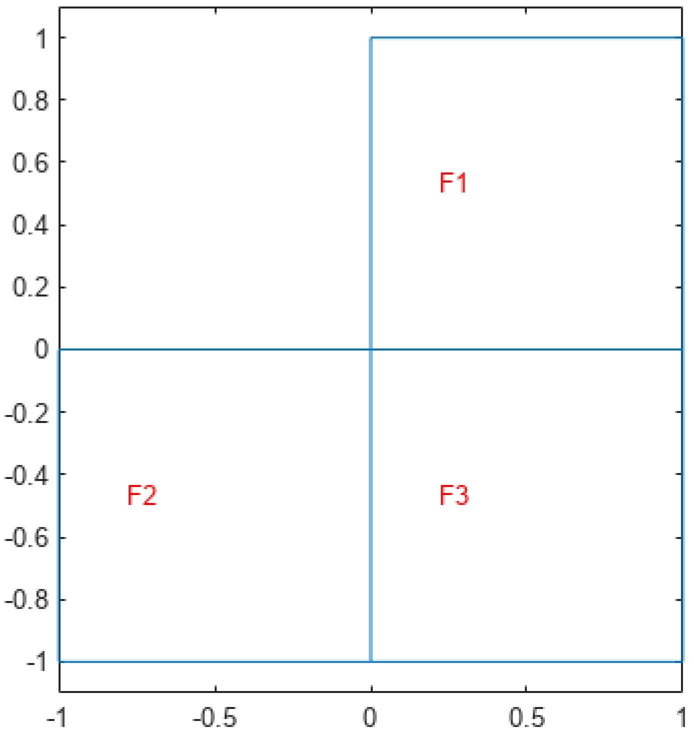
`hsa = findHeatSource(thermalmodel.HeatSources,RegionType,RegionID)` returns the heat source value `hsa` assigned to the specified region.

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
         Label: []
```

Check the heat source specification for faces 2 and 3.

```
hsa = findHeatSource(thermalmodel.HeatSources,"Face",[2 3]);
hsaFace2 = hsa(1)
```

```
hsaFace2 = 
  HeatSourceAssignment with properties:
```

```
    RegionType: 'face'
      RegionID: 2
    HeatSource: 20
         Label: []
```

```
hsaFace3 = hsa(2)
```

```
hsaFace3 = 
  HeatSourceAssignment with properties:

    RegionType: 'face'
      RegionID: 3
    HeatSource: 30
         Label: []
```

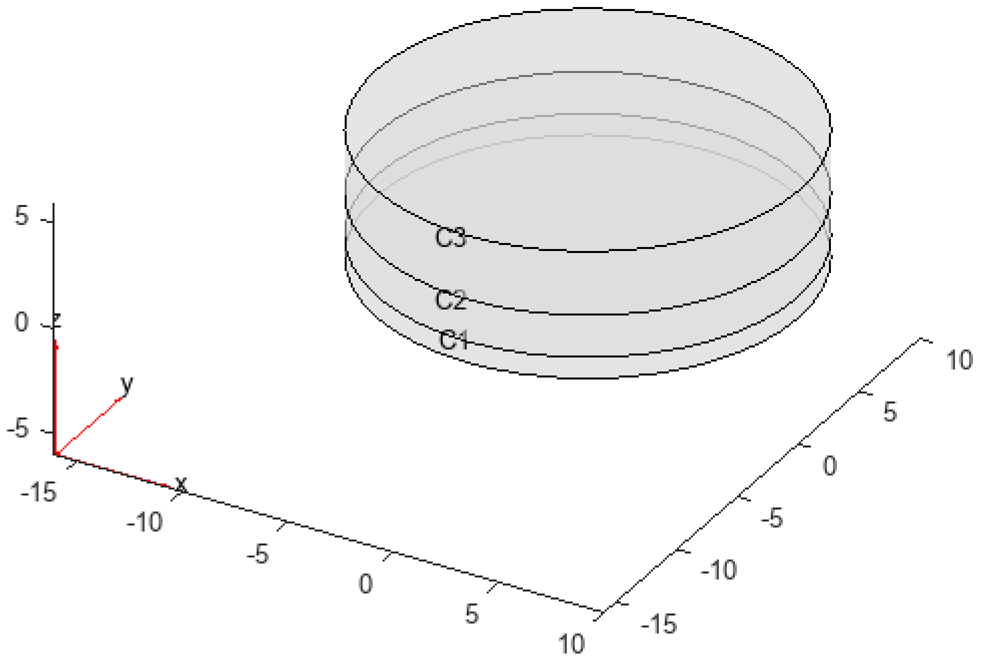
Find Heat Sources for Cells of 3-D Geometry

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])
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 3
       NumFaces: 7
       NumEdges: 4
    NumVertices: 4
       Vertices: [4x3 double]
```

```
thermalmodel = createpde("thermal");
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,"CellLabels","on","FaceAlpha",0.5)
```



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
         Label: []
```

Check the heat source specification for cells 2 and 3.

```
hsa = findHeatSource(thermalmodel.HeatSources,"Cell",2:3);
hsaCell2 = hsa(1)
```

```
hsaCell2 = 
  HeatSourceAssignment with properties:
```

```
    RegionType: 'cell'
      RegionID: 2
    HeatSource: 20
         Label: []
```

```
hsaCell3 = hsa(2)
```

```
hsaCell3 = 
  HeatSourceAssignment with properties:

    RegionType: 'cell'
      RegionID: 3
    HeatSource: 30
         Label: []
```

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` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using the `pdegplot` function.

Data Types: `double`

Output Arguments

`hsa` — Heat source assignment
`HeatSourceAssignment` object

Heat source assignment, returned as a HeatSourceAssignment object.

Version History

Introduced in R2017a

See Also

HeatSourceAssignment | `internalHeatSource`

findInitialConditions

Package: `pde`

Locate active initial conditions

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.

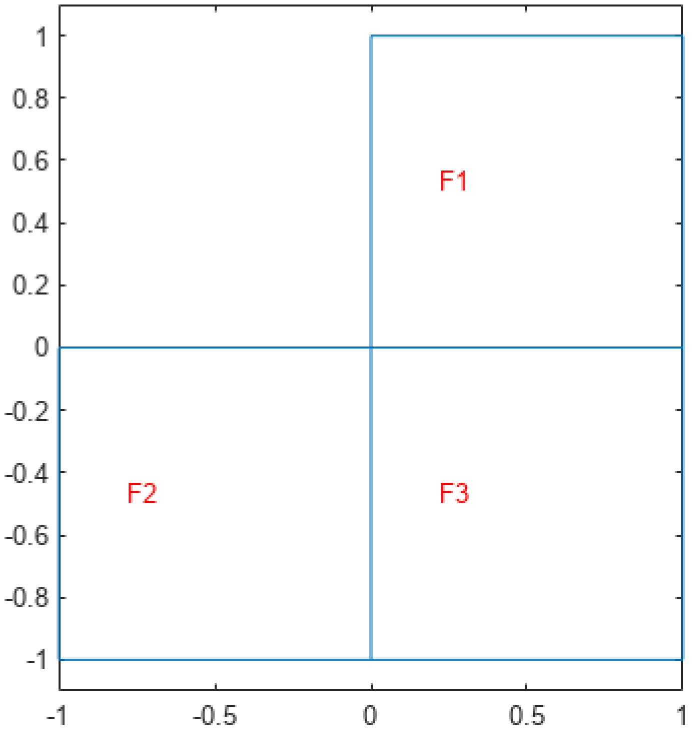
Examples

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: []
```

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` | `string`

`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.

Version History

Introduced in R2016a

See Also

GeometricInitialConditions | NodalInitialConditions | `setInitialConditions`

Topics

"View, Edit, and Delete Initial Conditions" on page 2-123
"Solve Problems Using PDEModel Objects" on page 2-2

findNodes

Package: `pde`

Find mesh nodes in specified region

Syntax

```
nodes = findNodes(mesh,"region",RegionType,RegionID)
nodes = findNodes(mesh,"box",xlim,ylim)
nodes = findNodes(mesh,"box",xlim,ylim,zlim)
nodes = findNodes(mesh,"radius",center,radius)
nodes = findNodes(mesh,"nearest",point)
```

Description

`nodes = findNodes(mesh,"region",RegionType,RegionID)` returns the IDs of the mesh nodes that belong to the specified geometric region.

`nodes = findNodes(mesh,"box",xlim,ylim)` returns the IDs of the mesh nodes within a bounding box specified by `xlim` and `ylim`. Use this syntax for 2-D meshes.

`nodes = findNodes(mesh,"box",xlim,ylim,zlim)` returns the IDs of the mesh nodes located within a bounding box specified by `xlim`, `ylim`, and `zlim`. Use this syntax for 3-D meshes.

`nodes = findNodes(mesh,"radius",center,radius)` returns the IDs of mesh nodes located within a circle (for 2-D meshes) or sphere (for 3-D meshes) specified by `center` and `radius`.

`nodes = findNodes(mesh,"nearest",point)` returns the IDs of mesh nodes closest to a query point or multiple query points with Cartesian coordinates specified by `point`.

Examples

Nodes Associated with Particular Edges and Faces

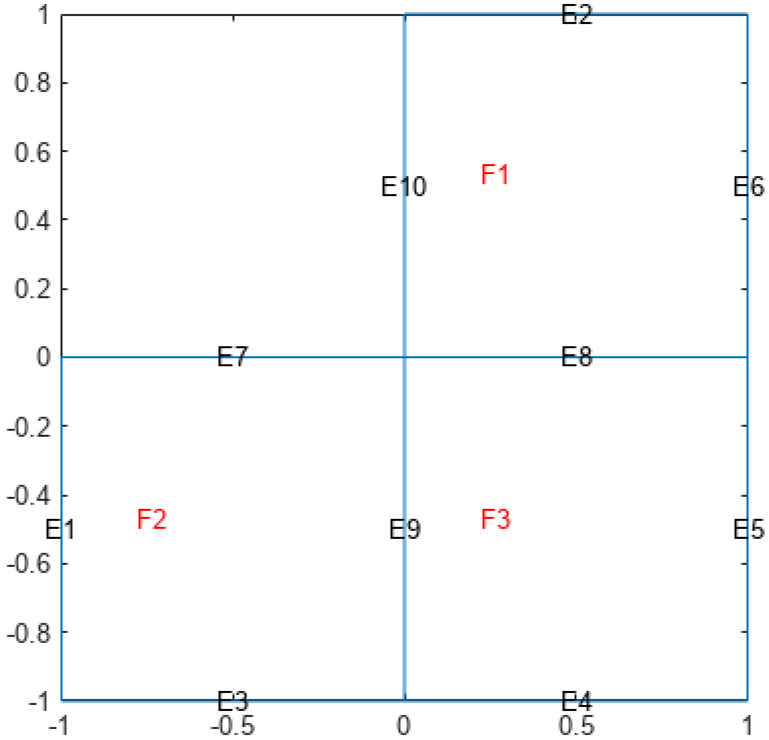
Find the nodes associated with a geometric region.

Create a PDE model.

```
model = createpde;
```

Include the geometry of the built-in function `lshapeg`. Plot the geometry.

```
geometryFromEdges(model,@lshapeg);
pdegplot(model,"FaceLabels","on","EdgeLabels","on")
```



Generate a mesh.

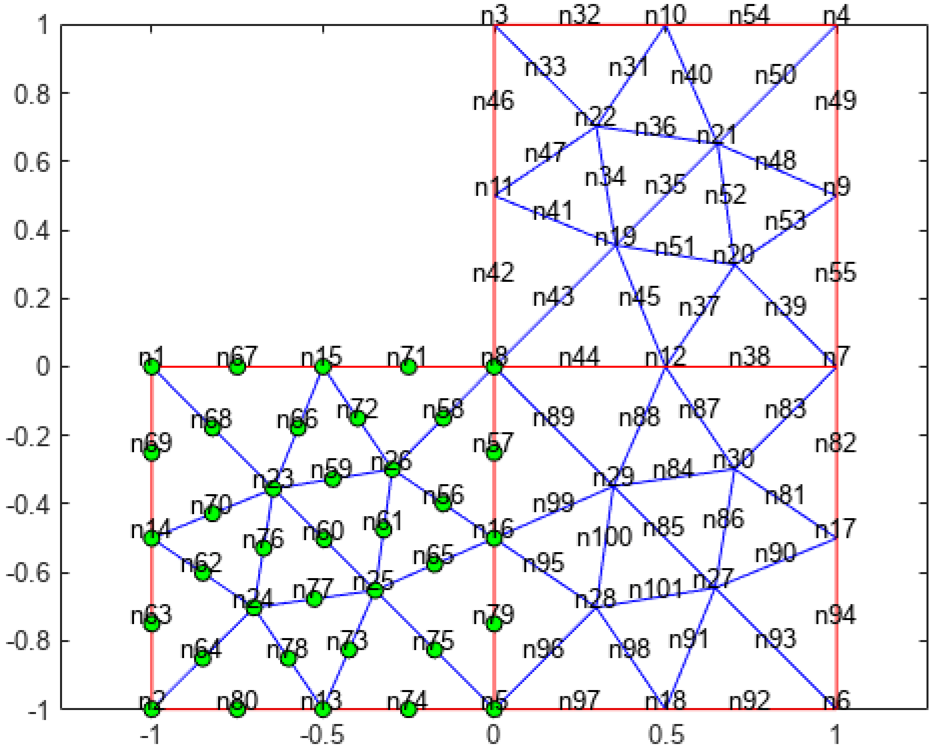
```
mesh = generateMesh(model,"Hmax",0.5);
```

Find the nodes associated with face 2.

```
Nf2 = findNodes(mesh,"region","Face",2);
```

Highlight these nodes in green on the mesh plot.

```
figure
pdemesh(model,"NodeLabels","on")
hold on
plot(mesh.Nodes(1,Nf2),mesh.Nodes(2,Nf2),"ok","MarkerFaceColor","g")
```

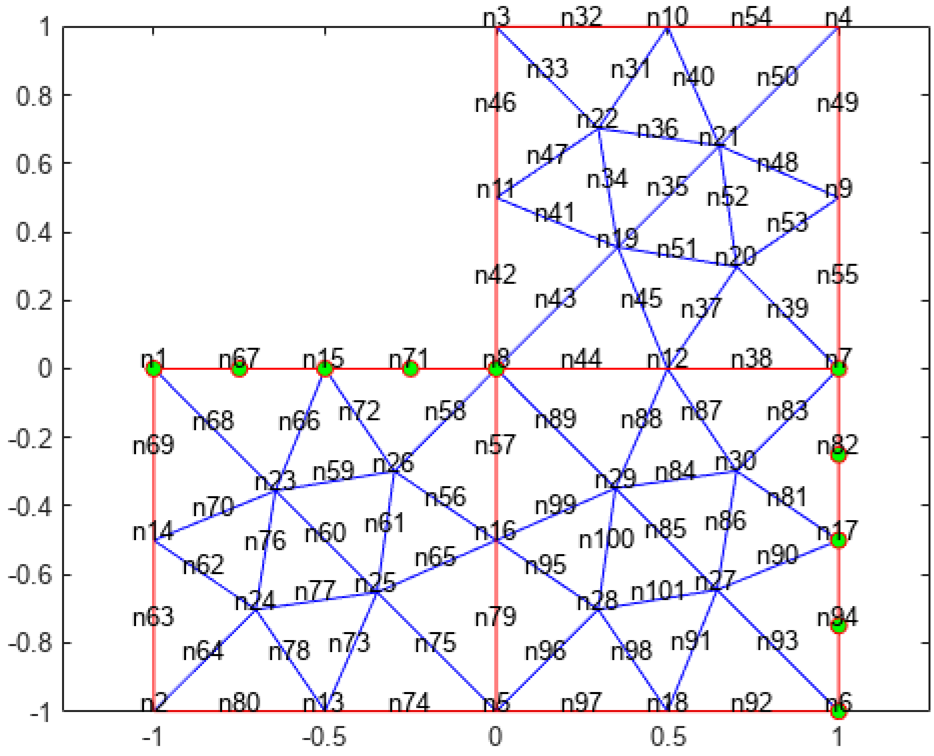


Find the nodes associated with edges 5 and 7.

```
Ne57 = findNodes(mesh,"region","Edge",[5 7]);
```

Highlight these nodes in green on the mesh plot.

```
figure
pdemesh(model,"NodeLabels","on")
hold on
plot(mesh.Nodes(1,Ne57),mesh.Nodes(2,Ne57),"or","MarkerFaceColor","g")
```



Nodes Within Bounding Box

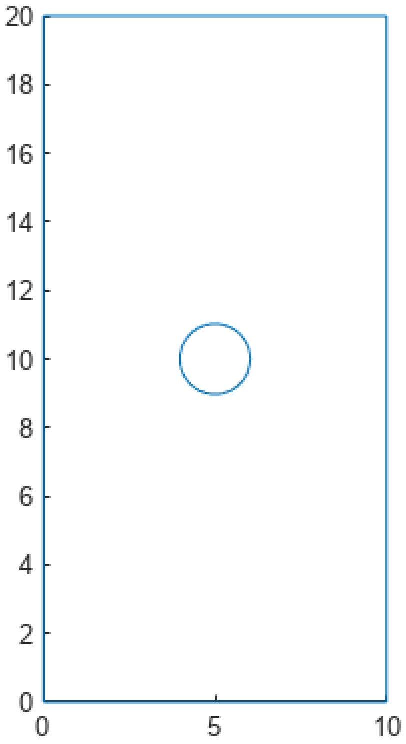
Find the nodes located within a specified box.

Create a PDE model.

```
model = createpde;
```

Import and plot the geometry.

```
importGeometry(model,"PlateHolePlanar.stl");
pdegplot(model)
```



Generate a mesh.

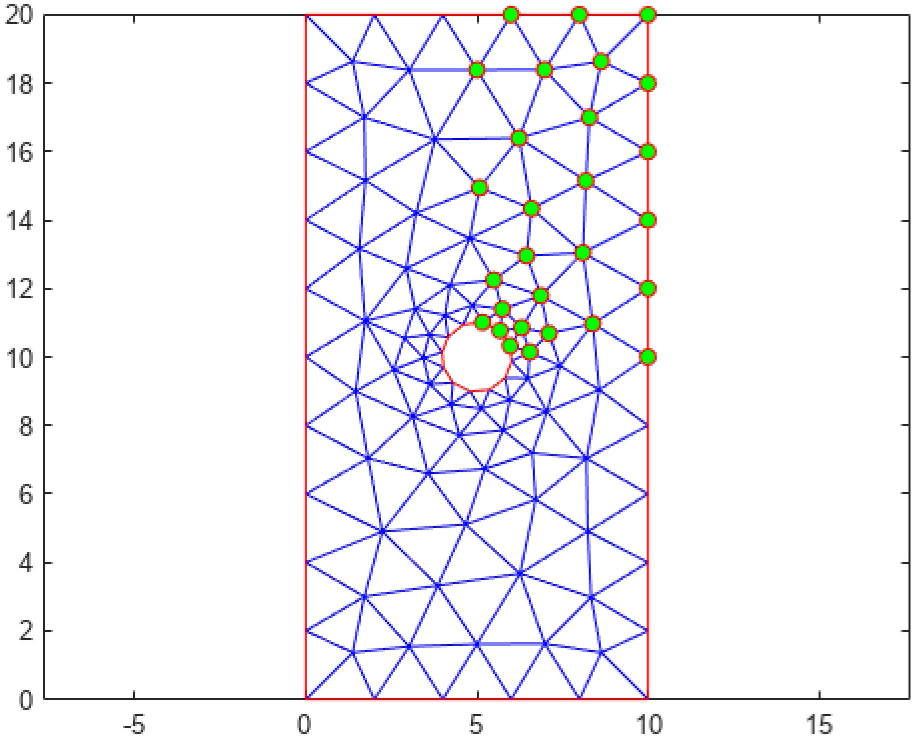
```
mesh = generateMesh(model,"Hmax",2,"Hmin",0.4, ...
                          "GeometricOrder","linear");
```

Find the nodes located within the following box.

```
Nb = findNodes(mesh,"box",[5 10],[10 20]);
```

Highlight these nodes in green on the mesh plot.

```
figure
pdemesh(model)
hold on
plot(mesh.Nodes(1,Nb),mesh.Nodes(2,Nb),"or","MarkerFaceColor","g")
```



Nodes Within Bounding Disk

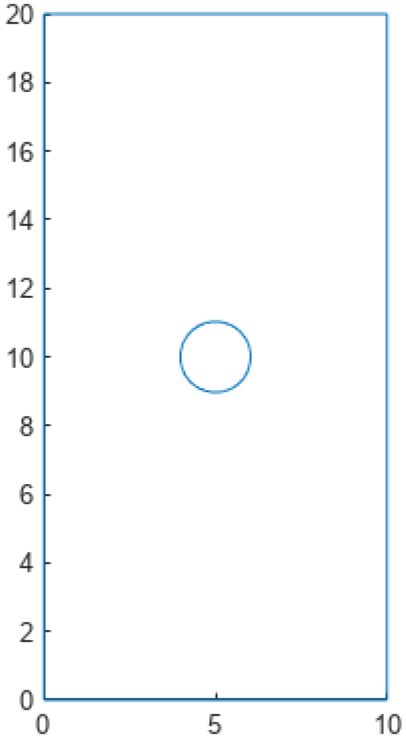
Find the nodes located within a specified disk.

Create a PDE model.

```
model = createpde;
```

Import and plot the geometry.

```
importGeometry(model,"PlateHolePlanar.stl");
pdegplot(model)
```



Generate a mesh.

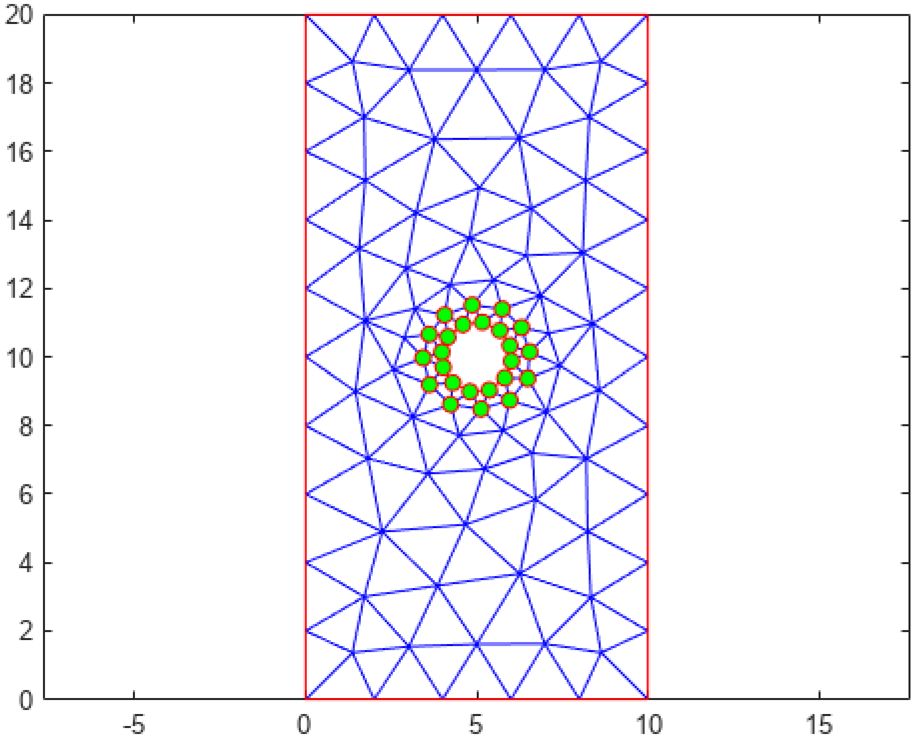
```
mesh = generateMesh(model,"Hmax",2,"Hmin",0.4, ...
                          "GeometricOrder","linear");
```

Find the nodes located within radius 2 from the center [5 10].

```
Nb = findNodes(mesh,"radius",[5 10],2);
```

Highlight these nodes in green on the mesh plot.

```
figure
pdemesh(model)
hold on
plot(mesh.Nodes(1,Nb),mesh.Nodes(2,Nb),"or","MarkerFaceColor","g")
```



Nodes Closest to Specified Points

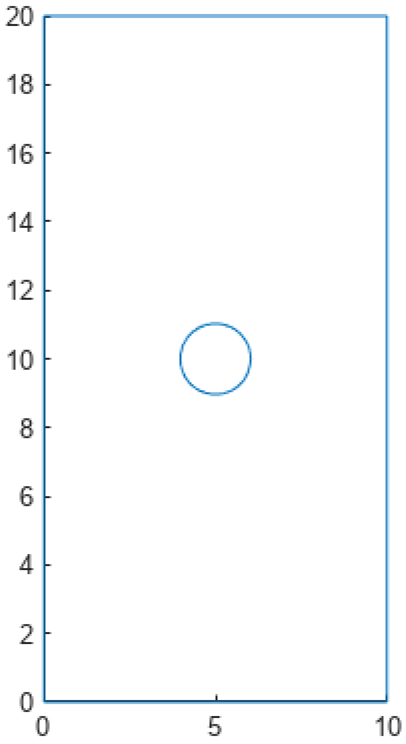
Find the node closest to a specified point and highlight it on the mesh plot.

Create a PDE model.

```
model = createpde;
```

Import and plot the geometry.

```
importGeometry(model,"PlateHolePlanar.stl");
pdegplot(model)
```



Generate a mesh.

```
mesh = generateMesh(model,"Hmax",2,"Hmin",0.4);
```

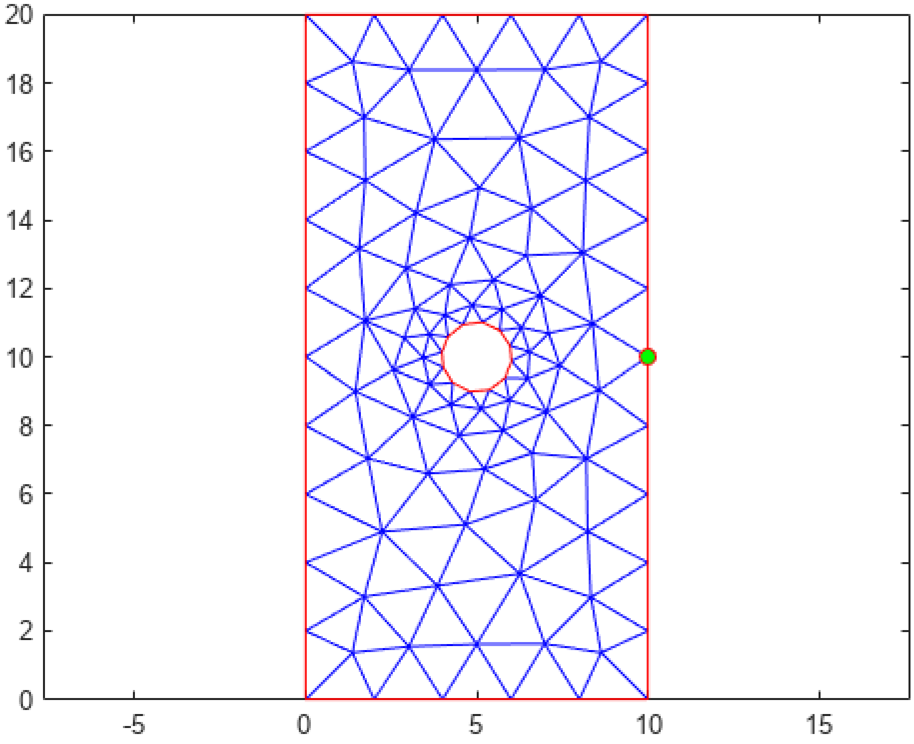
Find the node closest to the point [15;10].

```
N_ID = findNodes(mesh,"nearest",[15;10])
```

```
N_ID = 10
```

Highlight this node in green on the mesh plot.

```
figure
pdemesh(model)
hold on
plot(mesh.Nodes(1,N_ID),mesh.Nodes(2,N_ID),"or","MarkerFaceColor","g")
```



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`

`RegionType` — Geometric region type

`"Cell"` | `"Face"` | `"Edge"` | `"Vertex"`

Geometric region type, specified as `"Cell"`, `"Face"`, `"Edge"`, or `"Vertex"`.

Example: `findNodes(mesh,"region","Face",1:3)`

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 by using `pdegplot`.

Example: `findNodes(mesh,"region","Face",1:3)`

Data Types: `double`

`xlim` — *x*-limits of bounding box

two-element row vector

x-limits of the bounding box, specified as a two-element row vector. The first element of `xlim` is the lower *x*-bound, and the second element is the upper *x*-bound.

Example: `findNodes(mesh,"box",[5 10],[10 20])`

Data Types: `double`

`ylim` — *y*-limits of bounding box

two-element row vector

y-limits of the bounding box, specified as a two-element row vector. The first element of `ylim` is the lower *y*-bound, and the second element is the upper *y*-bound.

Example: `findNodes(mesh,"box",[5 10],[10 20])`

Data Types: `double`

`zlim` — *z*-limits of bounding box

two-element row vector

z-limits of the bounding box, specified as a two-element row vector. The first element of `zlim` is the lower *z*-bound, and the second element is the upper *z*-bound. You can specify `zlim` only for 3-D meshes.

Example: `findNodes(mesh,"box",[5 10],[10 20],[1 2])`

Data Types: `double`

`center` — Center of bounding circle or sphere

two-element row vector for a 2-D mesh | three-element row vector for a 3-D mesh

Center of the bounding circle or sphere, specified as a two-element row vector for a 2-D mesh or three-element row vector for a 3-D mesh. The elements of these vectors contain the coordinates of the center of a circle or a sphere.

Example: `findNodes(mesh,"radius",[0 0 0],0.5)`

Data Types: `double`

`radius` — Radius of bounding circle or sphere

positive number

Radius of the bounding circle or sphere, specified as a positive number.

Example: `findNodes(mesh,"radius",[0 0 0],0.5)`

Data Types: `double`

`point` — Cartesian coordinates of query points

2-by-N or 3-by-N matrix

Cartesian coordinates of query points, specified as a 2-by-*N* or 3-by-*N* matrix. These matrices contain the coordinates of the query points. Here, *N* is the number of query points.

Example: `findNodes(mesh,"nearest",[15 10.5 1; 12 10 1.2])`

Data Types: `double`

Output Arguments

nodes — Node IDs
positive integer | row vector of positive integers

Node IDs, returned as a positive integer or a row vector of positive integers.

Version History

Introduced in R2018a

See Also

`findElements` | `meshQuality` | `area` | `volume` | FEMesh Properties

Topics
“Finite Element Method Basics” on page 1-11

findStructuralBC

Package: `pde`

Find structural boundary conditions and boundary loads assigned to geometric region

Syntax

```
sbca = findStructuralBC(structuralmodel.BoundaryConditions,RegionType,
RegionID)
```

Description

`sbca = findStructuralBC(structuralmodel.BoundaryConditions,RegionType, RegionID)` returns the structural boundary conditions and boundary loads assigned to the region specified by `RegionType` and `RegionID`. The function returns structural boundary conditions assigned by `structuralBC` and boundary loads assigned by `structuralBoundaryLoad`.

Examples

Find Structural Boundary Conditions

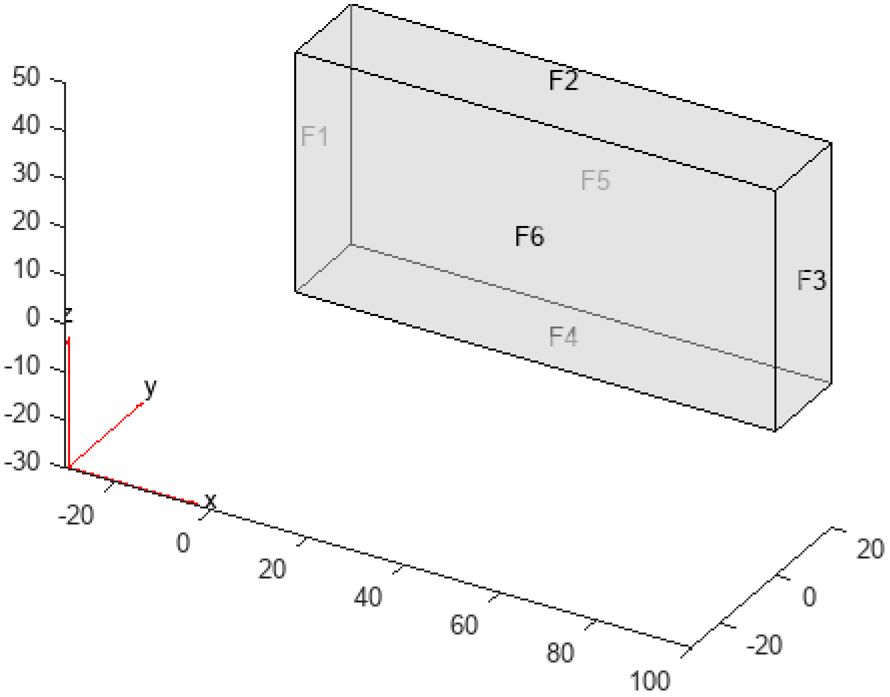
Find the structural boundary conditions for the faces of a 3-D geometry.

Create a structural model and include a block geometry.

```
structuralmodel = createpde("structural","static-solid");
```

Include the block geometry in the model and plot the geometry.

```
importGeometry(structuralmodel,"Block.stl");
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify the surface traction on face 1 of the block.

```
structuralBoundaryLoad(structuralmodel,"Face",1, ...
                                       "SurfaceTraction", ...
                                       [100;10;300]);
```

Specify the pressure on face 3 of the block.

```
structuralBoundaryLoad(structuralmodel,"Face",3,"Pressure",300);
```

Apply free constraint on faces 5 and 6 of the block.

```
structuralBC(structuralmodel,"Face",[5,6],"Constraint","free");
```

Check the boundary condition specification for faces 1 and 3.

```
sbca = findStructuralBC(structuralmodel.BoundaryConditions, ...
                                                "Face",[1,3]);
```

```
sbcaFace1 = sbca(1)
```

```
sbcaFace1 = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 1
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
```

```
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
           SurfaceTraction: [3x1 double]
                  Pressure: []
    TranslationalStiffness: []
                     Label: []
```

```
sbcaFace3 = sbca(2)
```

```
sbcaFace3 = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 3
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
           SurfaceTraction: []
                  Pressure: 300
    TranslationalStiffness: []
                     Label: []
```

Check the boundary condition specification for faces 5 and 6.

```
sbca = findStructuralBC(structuralmodel.BoundaryConditions, ...
                                                 "Face",[5,6]);
sbcaFace5 = sbca(1)
```

```
sbcaFace5 = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: [5 6]
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
```

```
            XDisplacement: []
            YDisplacement: []
            ZDisplacement: []
               Constraint: "free"
                   Radius: []
                Reference: []
                    Label: []

   Boundary Loads
                    Force: []
           SurfaceTraction: []
                 Pressure: []
    TranslationalStiffness: []
                    Label: []
```

```
sbcaFace6 = sbca(2)
```

```
sbcaFace6 = 
  StructuralBC with properties:

               RegionType: 'Face'
                 RegionID: [5 6]
               Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
             Displacement: []
            XDisplacement: []
            YDisplacement: []
            ZDisplacement: []
               Constraint: "free"
                   Radius: []
                Reference: []
                    Label: []

   Boundary Loads
                    Force: []
           SurfaceTraction: []
                 Pressure: []
    TranslationalStiffness: []
                    Label: []
```

Input Arguments

`structuralmodel.BoundaryConditions` — Structural boundary conditions

`BoundaryConditions` property of `StructuralModel` object

Structural boundary conditions of the model, specified as the `BoundaryConditions` property of a `StructuralModel` object.

RegionType — Geometric region type

`"Edge"` for a 2-D model | `"Face"` for a 3-D model

Geometric region type, specified as `"Edge"` for a 2-D model or `"Face"` for a 3-D model.

Example: `findStructuralBC(structuralmodel.BoundaryConditions,"Edge",1)`

Data Types: `char` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `findStructuralBC(structuralmodel.BoundaryConditions,"Face",1:3)`

Data Types: `double`

Output Arguments

`sbca` — Structural boundary conditions and boundary loads assignment
`StructuralBC` object

Structural boundary conditions and boundary loads assignment, returned as a `StructuralBC` object. For details, see StructuralBC Properties.

Version History

Introduced in R2017b

See Also

`structuralBC` | `structuralBoundaryLoad`

findStructuralIC

Package: `pde`

Find initial displacement and velocity assigned to geometric region

Syntax

`sica = findStructuralIC(structuralmodel.InitialConditions,RegionType, RegionID)`

Description

`sica = findStructuralIC(structuralmodel.InitialConditions,RegionType, RegionID)` returns the initial displacement and velocity assigned to the specified region.

Examples

Find Initial Conditions for Cells of 3-D Geometry

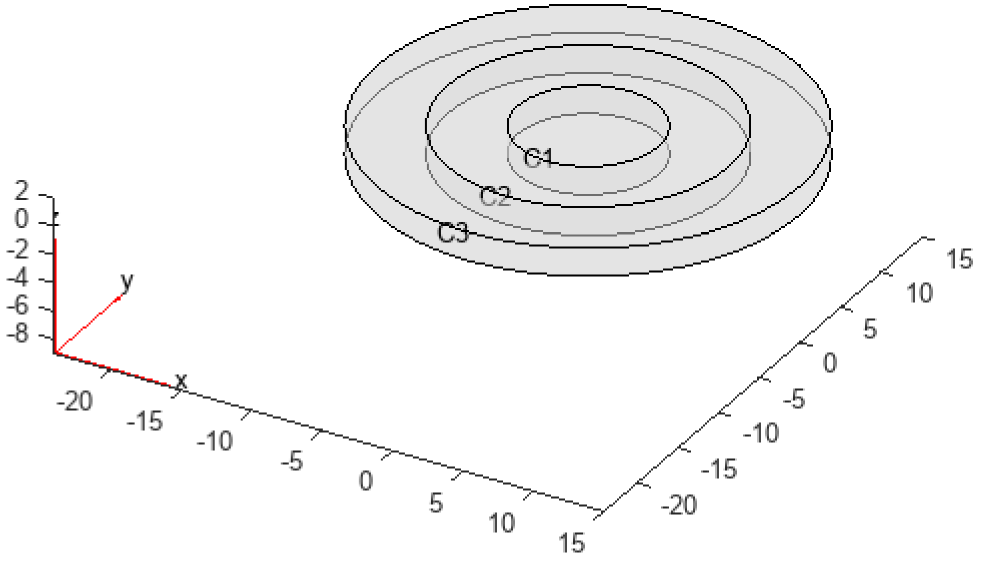
Find the initial displacement and velocity assigned to the cells of a 3-D geometry.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry consisting of the three nested cylinders and include it in the model. Plot the geometry.

```
gm = multicylinder([5 10 15],2);
structuralmodel = createpde("structural","transient-solid");
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"CellLabels","on","FaceAlpha",0.5)
```



Set the initial conditions for each cell. When you specify only the initial velocity or initial displacement, `structuralIC` assumes that the omitted parameter is zero.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],...
                             "Velocity",[0;0;0], ...
                             "Cell",1);
structuralIC(structuralmodel,"Displacement",[0;0.1;0], ...
                             "Cell",2);
structuralIC(structuralmodel,"Velocity",[0;0.2;0], ...
                             "Cell",3);
```

Check the initial condition specification for cell 1.

```
SICACell1 = findStructuralIC(structuralmodel.InitialConditions,"Cell",1)
```

```
SICACell1 = 
  GeometricStructuralICs with properties:

             RegionType: 'Cell'
               RegionID: 1
    InitialDisplacement: [3x1 double]
        InitialVelocity: [3x1 double]
```

```
SICACell1.InitialDisplacement
```

```
ans = 3×1
```

```
     0
     0
     0
```

```
SICACell1.InitialVelocity
```

```
ans = 3×1

     0
     0
     0
```

Find Initial Displacement Set as Previously Obtained Static Solution

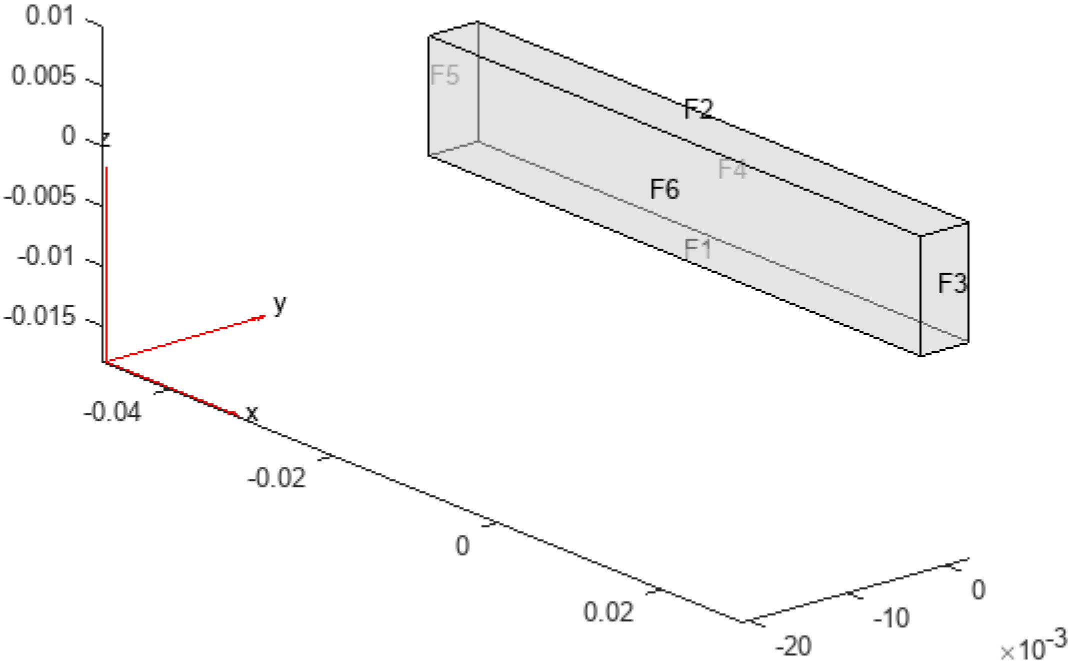
Use a static solution as an initial condition for a dynamic structural model. Check and plot the initial displacement.

Create a static model.

```
staticmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
staticmodel.Geometry = gm;
pdegplot(staticmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(staticmodel,"YoungsModulus",210E9, ...
                                 "PoissonsRatio",0.3,...
                                 "MassDensity",7800);
```

Apply the boundary condition and static load.

```
structuralBC(staticmodel,"Face",5,"Constraint","fixed");
structuralBoundaryLoad(staticmodel,"Face",3, ...
                                   "SurfaceTraction",[0;1E6;0]);
generateMesh(staticmodel,"Hmax",0.02);
Rstatic = solve(staticmodel);
```

Create a dynamic model and assign geometry.

```
dynamicmodel = createpde("structural","transient-solid");
gm = multicuboid(0.06,0.005,0.01);
dynamicmodel.Geometry = gm;
```

Apply the boundary condition.

```
structuralBC(dynamicmodel,"Face",5,"Constraint","fixed");
```

Specify the initial condition using the static solution.

```
generateMesh(dynamicmodel,"Hmax",0.02);
structuralIC(dynamicmodel,Rstatic)
```

```
ans =
  NodalStructuralICs with properties:

    InitialDisplacement: [113x3 double]
        InitialVelocity: [113x3 double]
```

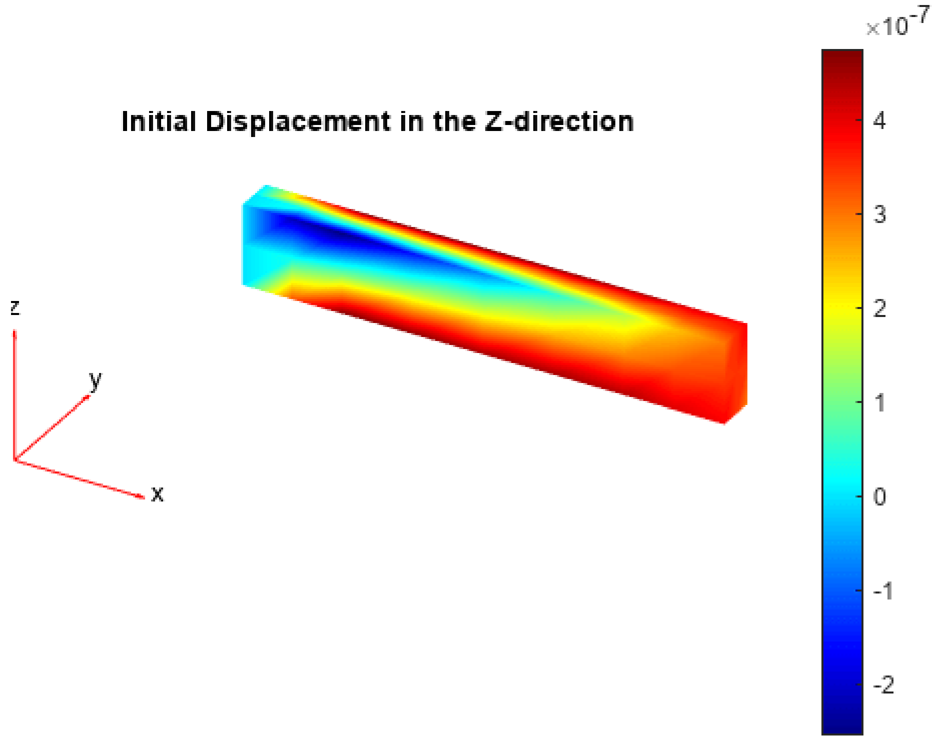
Check the initial condition specification for `dynamicmodel`.

```
sica = findStructuralIC(dynamicmodel.InitialConditions,"Cell",1)
```

```
sica =
  NodalStructuralICs with properties:

    InitialDisplacement: [113x3 double]
        InitialVelocity: [113x3 double]
```

Plot the z-component of the initial displacement.

```
pdeplot3D(dynamicmodel,"ColorMapData",sica.InitialDisplacement(:,3))
title("Initial Displacement in the Z-direction")
```



Input Arguments

`structuralmodel.InitialConditions` — Initial conditions
`InitialConditions` property of a `StructuralModel` object

Initial conditions of a transient structural model, specified as the `InitialConditions` property of a `StructuralModel` object.

`RegionType` — Geometric region type
`"Face"` | `"Edge"` | `"Vertex"` | `"Cell"` for a 3-D model

Geometric region type, specified as `"Face"`, `"Edge"`, 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 by using `pdegplot`.

Data Types: `double`

Output Arguments

`sica` — Structural initial condition assignment
`GeometricStructuralICs` object | `NodalStructuralICs` object

Structural initial condition for a particular region, returned as a `GeometricStructuralICs` or `NodalStructuralICs` object. For details, see GeometricStructuralICs Properties and NodalStructuralICs Properties.

Version History

Introduced in R2018a

See Also

`structuralIC` | GeometricStructuralICs Properties | NodalStructuralICs Properties | `StructuralModel`

findStructuralProperties

Package: `pde`

Find structural material properties assigned to geometric region

Syntax

`smpa = findStructuralProperties(structuralmodel.MaterialProperties, RegionType,RegionID)`

Description

`smpa = findStructuralProperties(structuralmodel.MaterialProperties, RegionType,RegionID)` returns the structural material properties assigned to the specified region. Structural properties include Young's modulus, Poisson's ratio, the mass density, the coefficient of thermal expansion, and the hysteretic damping parameter of the material.

Examples

Find Young's Modulus and Poisson's Ratio

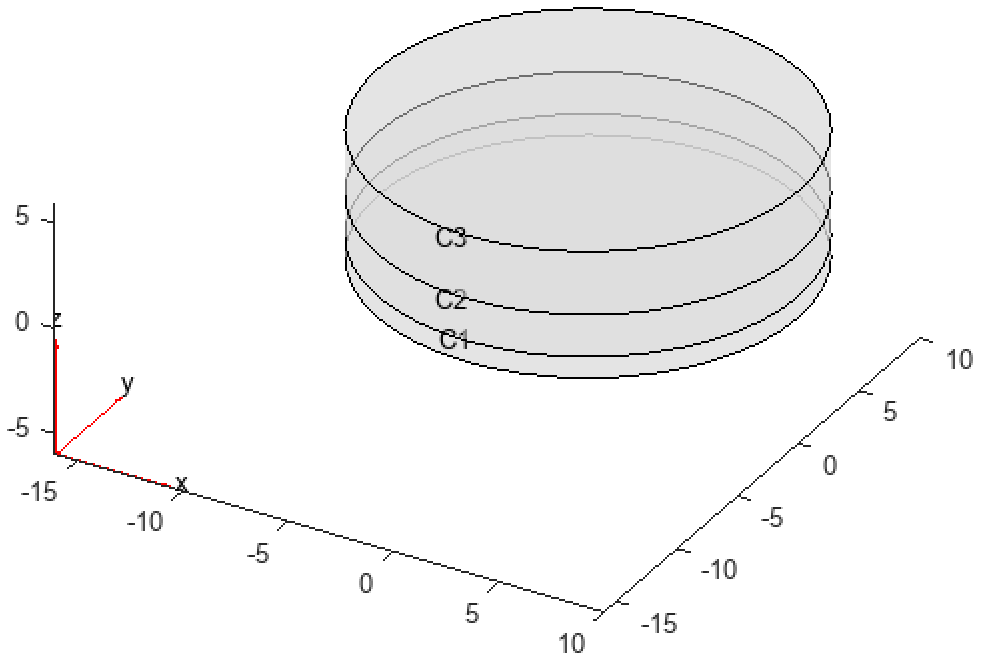
Find Young's modulus and Poisson's ratio for cells of a 3-D geometry.

Create a structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry consisting of three stacked cylinders and include it in the model. Plot the geometry.

```
gm = multicylinder(10,[1 2 3],"ZOffset",[0 1 3]);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"CellLabels","on","FaceAlpha",0.5)
```



Assign different values of Young's modulus and Poisson's ratio to each cell.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",200E9, ...
                                              "PoissonsRatio",0.3);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
structuralProperties(structuralmodel,"Cell",3,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.35);
```

Check the structural properties specification for cell 1.

```
mC1 = findStructuralProperties(structuralmodel.MaterialProperties, ...
                                                         "Cell",1)
```

```
mC1 = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 1
        YoungsModulus: 2.0000e+11
        PoissonsRatio: 0.3000
          MassDensity: []
                  CTE: []
    HystereticDamping: []
```

Check the structural properties specification for cells 2 and 3.

```
mC23 = findStructuralProperties(structuralmodel.MaterialProperties, ...
                                                     "Cell",[2,3]);
mC2 = mC23(1)
```

```
mC2 = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 2
        YoungsModulus: 2.1000e+11
        PoissonsRatio: 0.3000
          MassDensity: []
                  CTE: []
    HystereticDamping: []
```

```
mC3 = mC23(2)
```

```
mC3 = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 3
        YoungsModulus: 1.1000e+11
        PoissonsRatio: 0.3500
          MassDensity: []
                  CTE: []
    HystereticDamping: []
```

Input Arguments

`structuralmodel.MaterialProperties` — Material properties

`MaterialProperties` property of `StructuralModel` object

Material properties of the model, specified as the `MaterialProperties` property of a `StructuralModel` object.

Example: `structuralmodel.MaterialProperties`

`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: `findStructuralProperties(structuralmodel.MaterialProperties,"Cell",1)`

Data Types: `char` | `string`

`RegionID` — Geometric region ID

vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example:
`findStructuralProperties(structuralmodel.MaterialProperties,"Face",1:3)`

Data Types: `double`

Output Arguments

`smpa` — Material properties assignment
`StructuralMaterialAssignment` object

Material properties assignment, returned as a `StructuralMaterialAssignment` object. For details, see StructuralMaterialAssignment Properties.

Version History

Introduced in R2017b

See Also

`structuralProperties` | StructuralMaterialAssignment Properties

findThermalBC

Package: `pde`

Find thermal boundary conditions assigned to a geometric region

Syntax

```
tbca = findThermalBC(thermalmodel.BoundaryConditions,RegionType,RegionID)
```

Description

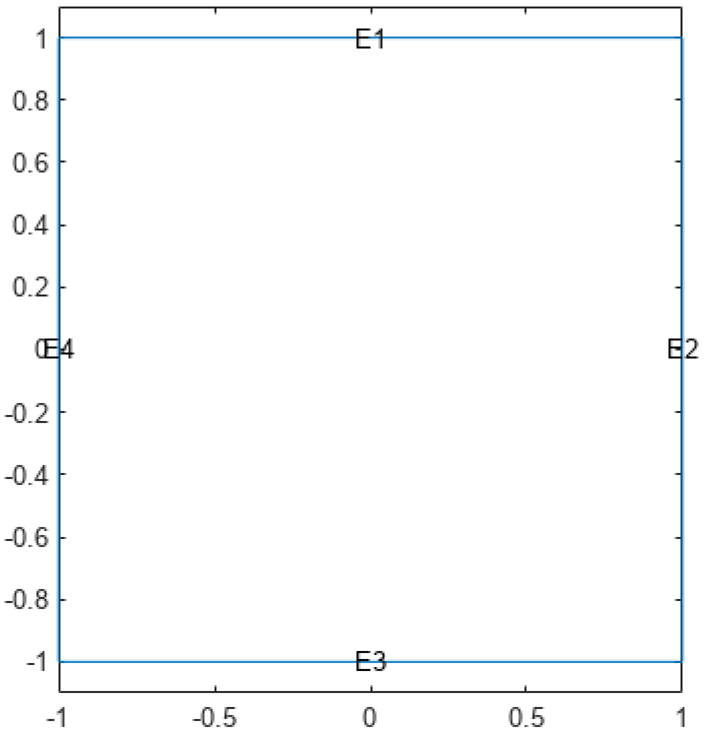
`tbca = findThermalBC(thermalmodel.BoundaryConditions,RegionType,RegionID)` returns the thermal boundary condition assigned to the specified region.

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'
                    Label: []
```

Check the boundary condition specifications on edges 3 and 4.

```
tbca = findThermalBC(thermalmodel.BoundaryConditions,"Edge",3:4);
tbcaEdge3 = tbca(1)
```

```
tbcaEdge3 = 
  ThermalBC with properties:

               RegionType: 'Edge'
                 RegionID: [1 3]
              Temperature: 100
                 HeatFlux: []
    ConvectionCoefficient: []
               Emissivity: []
       AmbientTemperature: []
               Vectorized: 'off'
                    Label: []
```

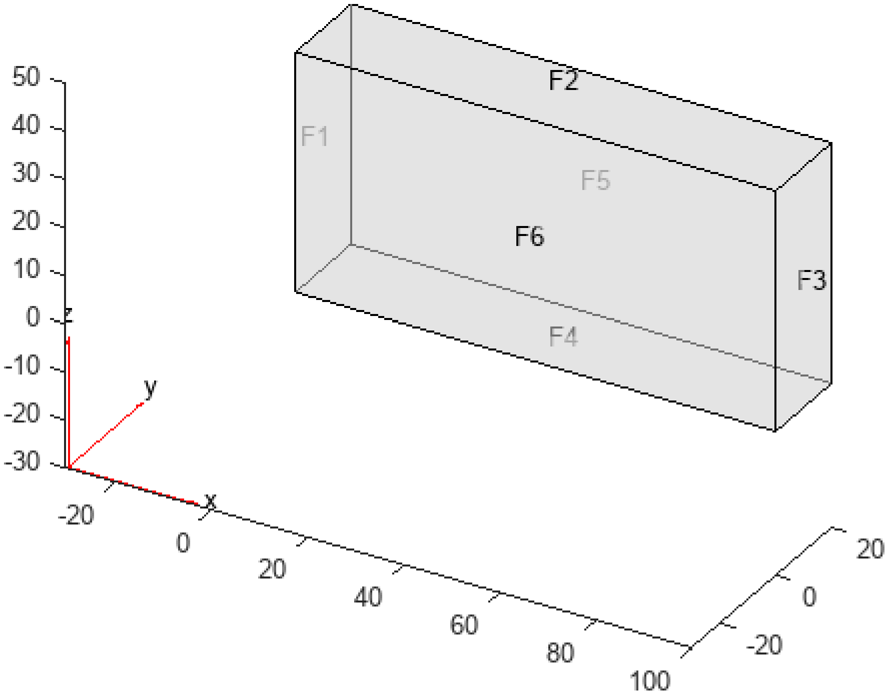
```
tbcaEdge4 = tbca(2)
```

```
tbcaEdge4 = 
  ThermalBC with properties:

               RegionType: 'Edge'
                 RegionID: 4
              Temperature: []
                 HeatFlux: 20
    ConvectionCoefficient: []
               Emissivity: []
       AmbientTemperature: []
               Vectorized: 'off'
                    Label: []
```

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)
```

```
tbcaFace1 =
  ThermalBC with properties:

               RegionType: 'Face'
                 RegionID: 1
              Temperature: 100
                 HeatFlux: []
    ConvectionCoefficient: []
               Emissivity: []
       AmbientTemperature: []
               Vectorized: 'off'
```

```
                    Label: []
```

```
tbcaFace3 = tbca(2)
```

```
tbcaFace3 = 
  ThermalBC with properties:

               RegionType: 'Face'
                 RegionID: 3
              Temperature: 300
                 HeatFlux: []
    ConvectionCoefficient: []
               Emissivity: []
       AmbientTemperature: []
               Vectorized: 'off'
                    Label: []
```

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'
                    Label: []
```

```
tbcaFace6 = findThermalBC(thermalmodel.BoundaryConditions,"Face",6)
```

```
tbcaFace6 = 
  ThermalBC with properties:

               RegionType: 'Face'
                 RegionID: [5 6]
              Temperature: []
                 HeatFlux: []
    ConvectionCoefficient: 5
               Emissivity: []
       AmbientTemperature: 27
               Vectorized: 'off'
                    Label: []
```

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` | `string`

`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.

Version History

Introduced in R2017a

See Also

`thermalBC` | ThermalBC

findThermalIC

Package: pde

Find thermal initial conditions assigned to a geometric region

Syntax

`tica = findThermalIC(thermalmodel.InitialConditions,RegionType,RegionID)`

Description

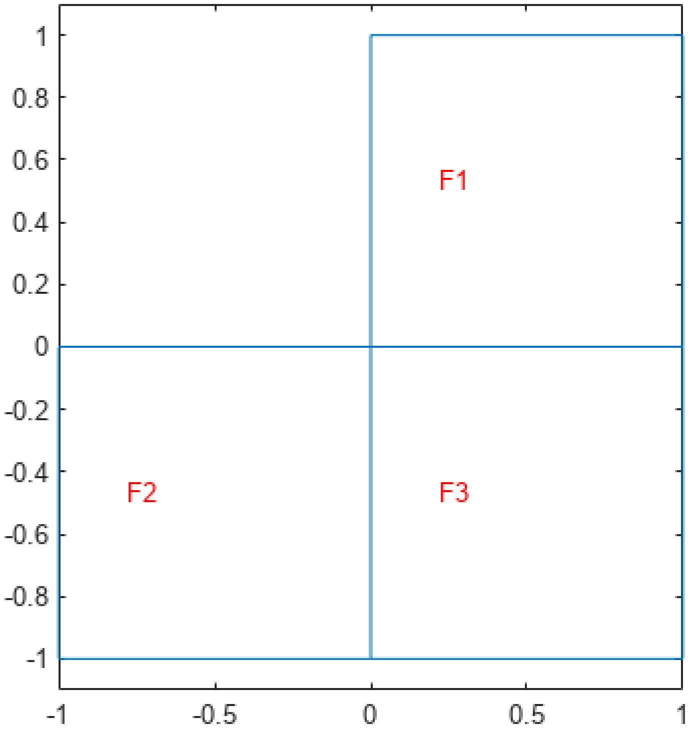
`tica = findThermalIC(thermalmodel.InitialConditions,RegionType,RegionID)` returns the thermal initial condition assigned to the specified region.

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
```

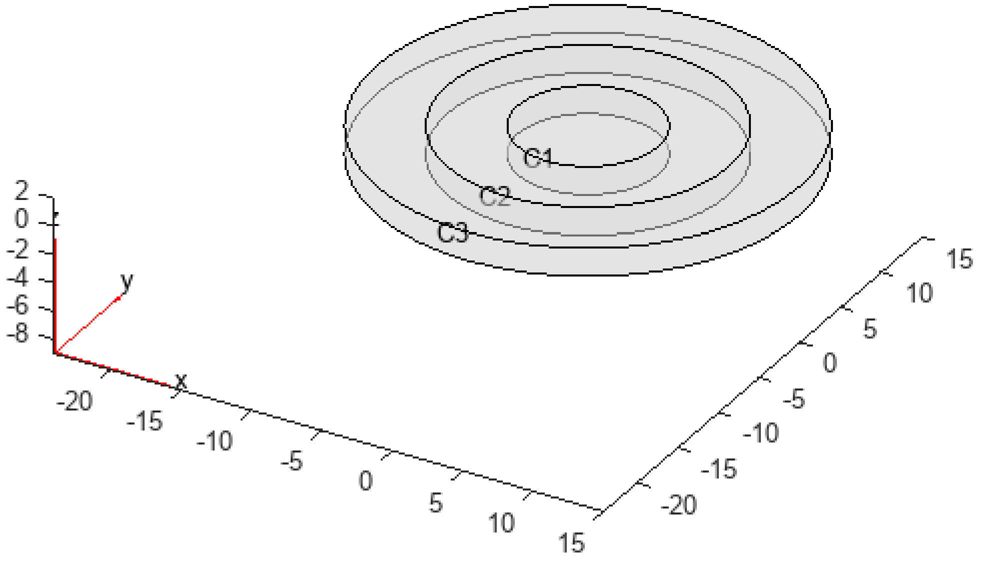
Find Initial Temperatures for Cells of 3-D Geometry

Create a geometry that consists of three nested cylinders and include the geometry in a transient thermal model.

```
gm = multicylinder([5 10 15],2)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 3
       NumFaces: 9
       NumEdges: 6
    NumVertices: 6
       Vertices: [6x3 double]
```

```
thermalmodel = createpde("thermal","transient");
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,"CellLabels","on","FaceAlpha",0.5)
```



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
```

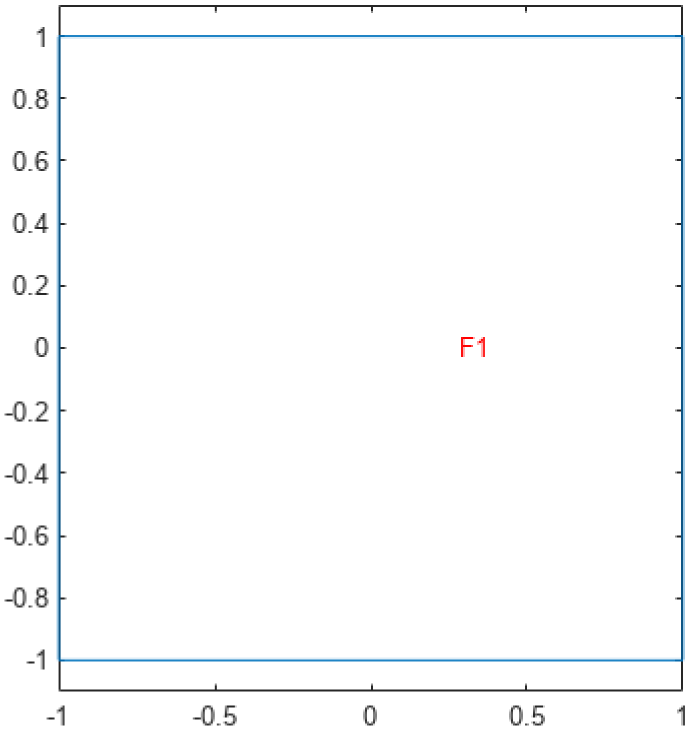
```
ticaCell3 = tica(2)
```

```
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: [1541x21 double]
    SolutionTimes: [0 0.5000 1 1.5000 2 2.5000 3 3.5000 4 4.5000 5 ... ]
       XGradients: [1541x21 double]
       YGradients: [1541x21 double]
       ZGradients: []
             Mesh: [1x1 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

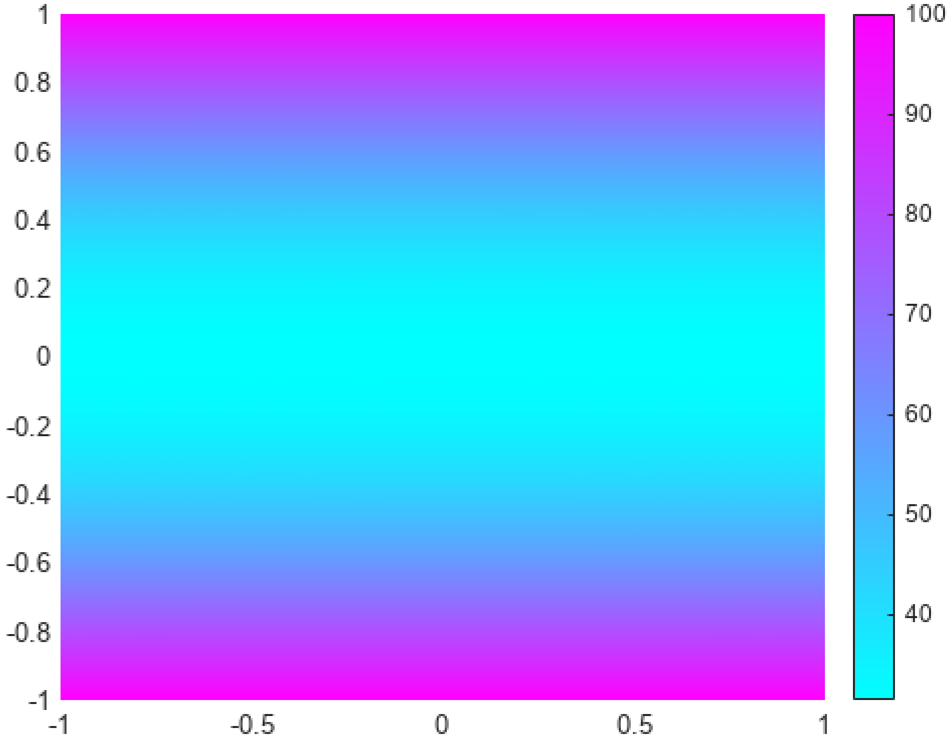
```
tlist = 10:0.5:15;
result2 = solve(thermalmodel,tlist);
```

Check the currently active initial temperature specification.

```
tica = findThermalIC(thermalmodel.InitialConditions,"Face",1)
```

```
tica = 
  NodalThermalICs with properties:

    InitialTemperature: [1541x1 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` | `string`

`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
`GeometricThermalICs` object | `NodalThermalICs` object

Thermal initial condition for a particular region, returned as a GeometricThermalICs or NodalThermalICs object.

Version History

Introduced in R2017a

See Also

GeometricThermalICs | NodalThermalICs | `thermalIC`

findThermalProperties

Package: pde

Find thermal material properties assigned to a geometric region

Syntax

`tmpa = findThermalProperties(thermalmodel.MaterialProperties,RegionType,RegionID)`

Description

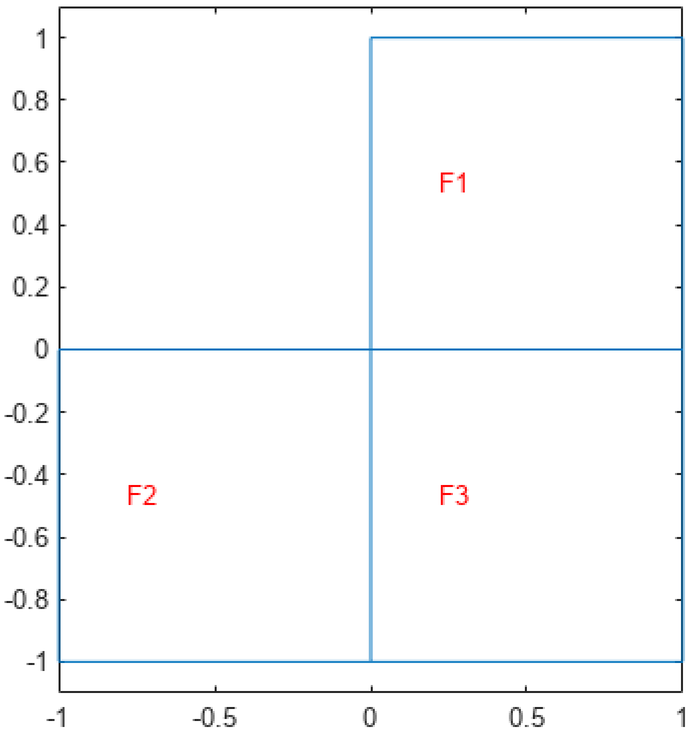
`tmpa = findThermalProperties(thermalmodel.MaterialProperties,RegionType,RegionID)` returns thermal material properties `tmpa` assigned to the specified region.

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 3, 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)

```
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)
```

```
mpaFace2 = 
  ThermalMaterialAssignment with properties:

             RegionType: 'face'
               RegionID: 2
    ThermalConductivity: 20
            MassDensity: 2
           SpecificHeat: 0.2000
```

```
mpaFace3 = mpa(2)
```

```
mpaFace3 = 
  ThermalMaterialAssignment with properties:

             RegionType: 'face'
               RegionID: 3
    ThermalConductivity: 30
            MassDensity: 3
           SpecificHeat: 0.3000
```

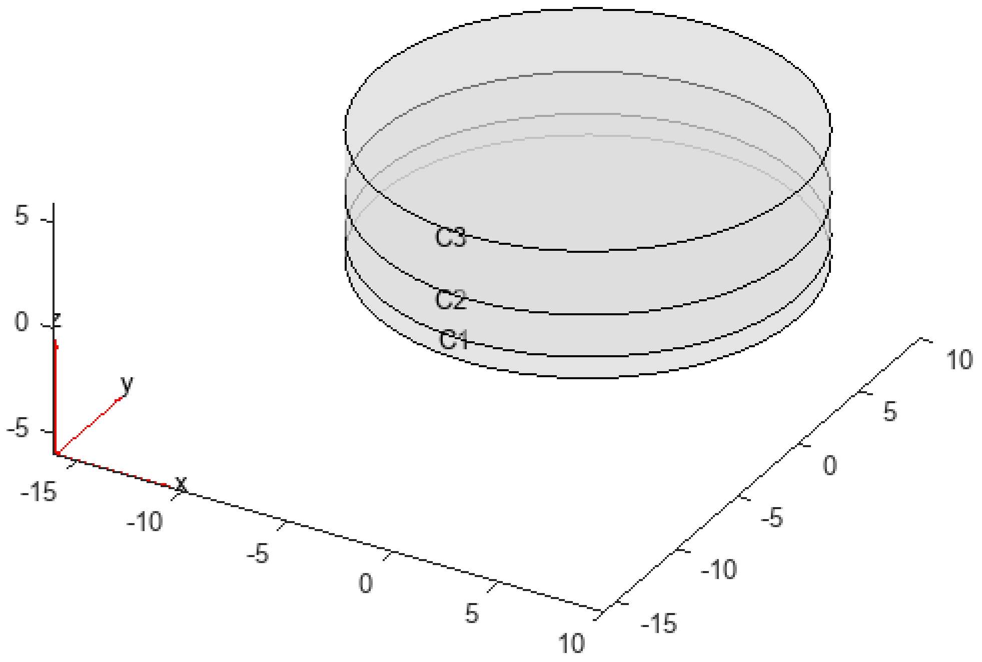
Find Thermal Conductivity for Cells of 3-D Geometry

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])
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 3
       NumFaces: 7
       NumEdges: 4
    NumVertices: 4
       Vertices: [4x3 double]
```

```
thermalmodel = createpde("thermal");
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,"CellLabels","on","FaceAlpha",0.5)
```



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)
```

```
mpaCell2 = 
  ThermalMaterialAssignment with properties:

             RegionType: 'cell'
               RegionID: 2
    ThermalConductivity: 20
            MassDensity: []
           SpecificHeat: []
```

```
mpaCell3 = mpa(2)
```

```
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` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `findThermalProperties(thermalmodel.MaterialProperties,"Face",1:3)`

Data Types: `double`

Output Arguments

`tmpa` — Material properties assignment
`ThermalMaterialAssignment` object

Material properties assignment, returned as a `ThermalMaterialAssignment` object. See ThermalMaterialAssignment.

Version History

Introduced in R2017a

See Also

ThermalMaterialAssignment | `thermalProperties` | `ThermalModel`

generateMesh

Package: `pde`

Create triangular or tetrahedral mesh

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 a geometry. For details about creating a geometry and including it in a model, see “Geometry and Mesh” and the geometry functions listed there.

`generateMesh(model,Name,Value)` modifies the mesh creation according to the `Name,Value` arguments.

`mesh = generateMesh( ___ )` also 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.

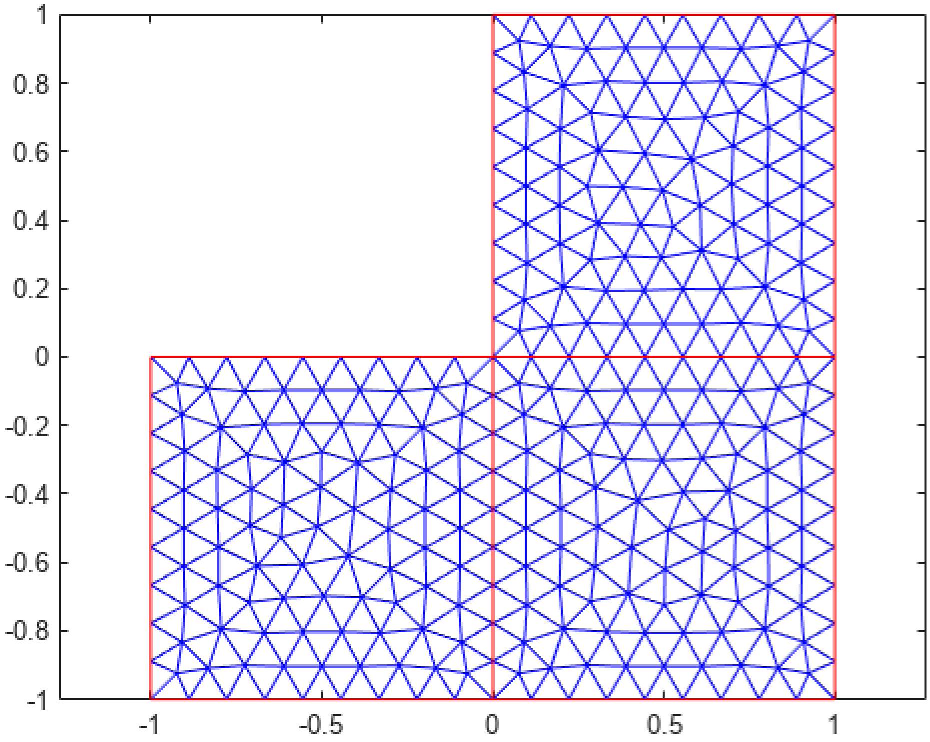
```
model = createpde(1);
geometryFromEdges(model,@lshapeg);
```

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");
```

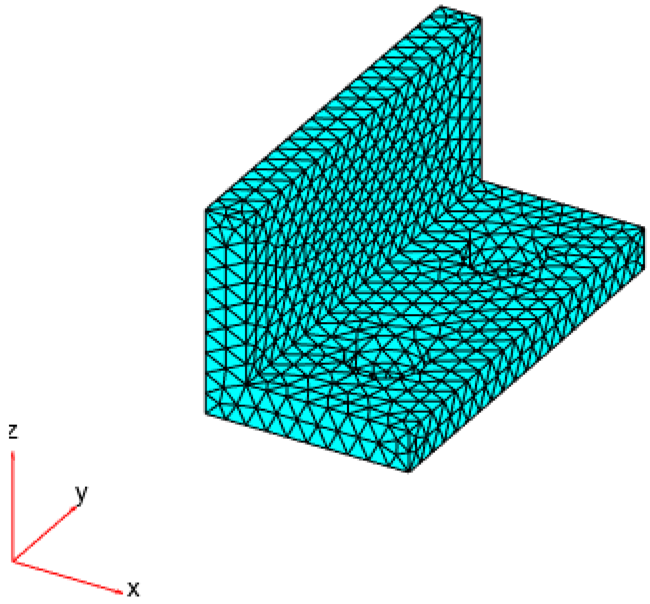
Generate a default mesh for comparison.

```
generateMesh(model)
```

```
ans = 
  FEMesh with properties:

             Nodes: [3x10003 double]
          Elements: [10x5774 double]
    MaxElementSize: 9.7980
    MinElementSize: 4.8990
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

View the mesh.

```
pdeplot3D(model)
```



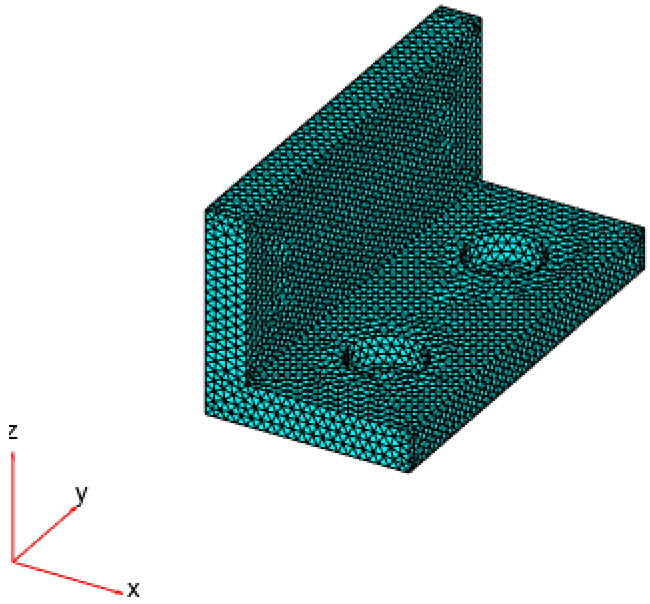
Create a mesh with target maximum element size 5 instead of the default 7.3485.

```
generateMesh(model,"Hmax",5)
```

```
ans = 
  FEMesh with properties:

             Nodes: [3x66936 double]
          Elements: [10x44059 double]
    MaxElementSize: 5
    MinElementSize: 2.5000
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

View the mesh.

```
pdeplot3D(model)
```



Refine Mesh on Specified Edges and Vertices

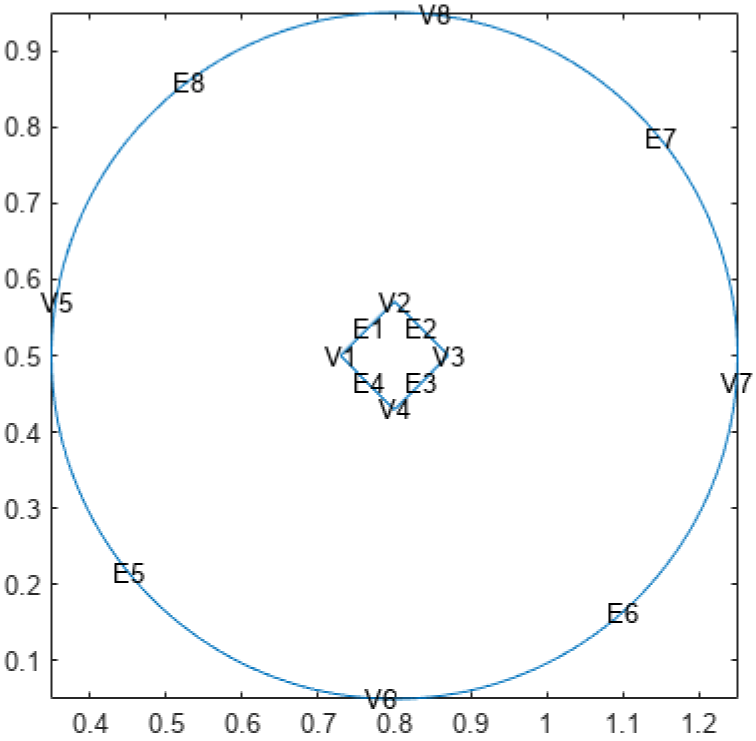
Generate a 2-D mesh with finer spots around the specified edges and vertices.

Create a model.

```
model = createpde;
```

Create and plot a 2-D geometry representing a circle with a diamond-shaped hole in its center.

```
g = geometryFromEdges(model,@scatterg);
pdegplot(g,"VertexLabels","on","EdgeLabels","on")
```



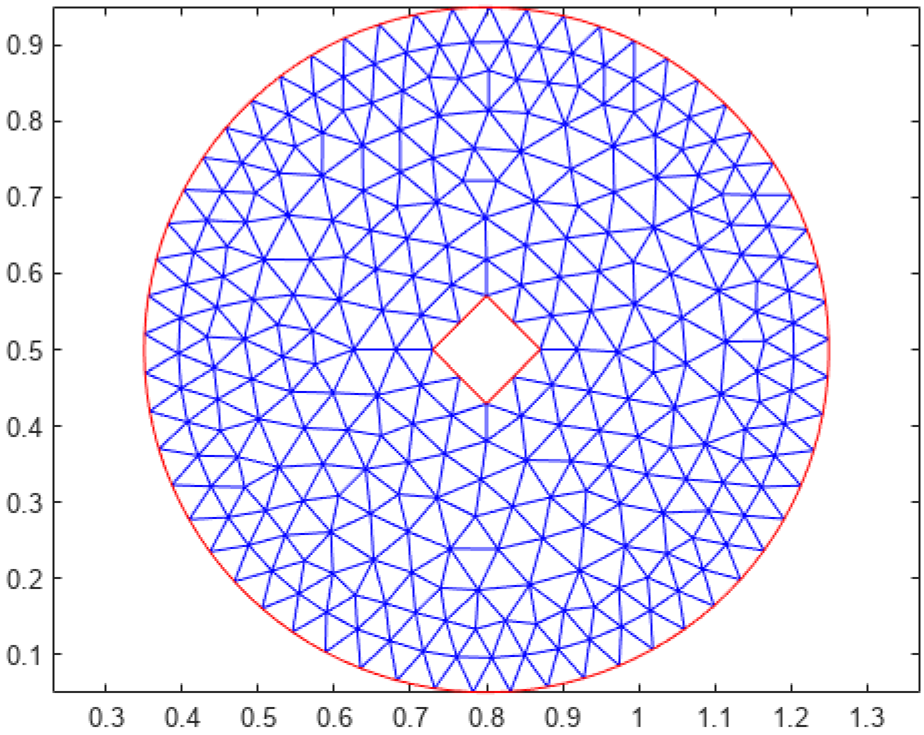
Generate a mesh for this geometry using the default mesh parameters.

```
m1 = generateMesh(model)
```

```
m1 = 
  FEMesh with properties:

             Nodes: [2x1159 double]
          Elements: [6x547 double]
    MaxElementSize: 0.0509
    MinElementSize: 0.0254
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

Plot the resulting mesh.

```
pdeplot(m1)
```



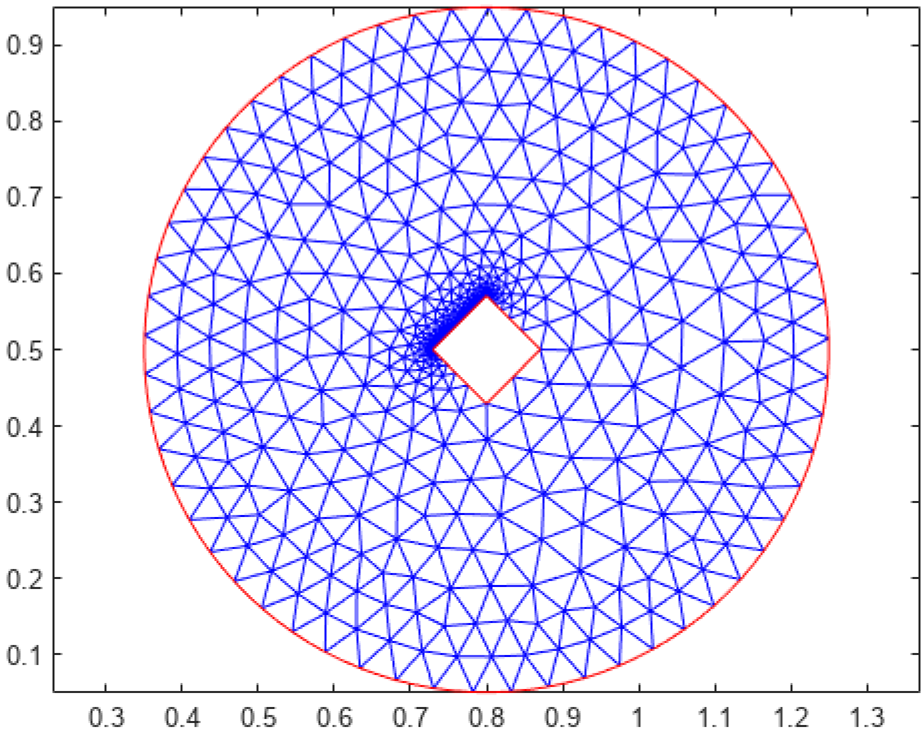
Generate a mesh with the target size on edge 1, which is smaller than the target minimum element size, `MinElementSize`, of the default mesh.

```
m2 = generateMesh(model,"Hedge",{1,0.001})
```

```
m2 = 
  FEMesh with properties:

             Nodes: [2x2631 double]
          Elements: [6x1241 double]
    MaxElementSize: 0.0509
    MinElementSize: 0.0254
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

Plot the resulting mesh.

```
pdeplot(m2)
```



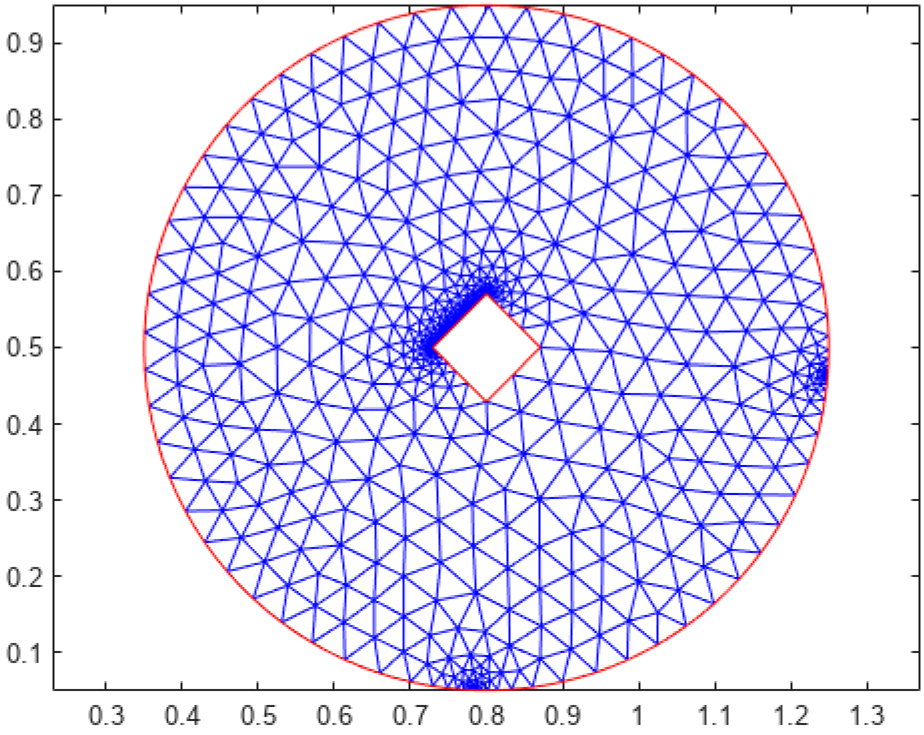
Generate a mesh specifying the target sizes for edge 1 and vertices 6 and 7.

```
m3 = generateMesh(model,"Hedge",{1,0.001},"Hvertex",{[6 7],0.002})
```

```
m3 = 
  FEMesh with properties:

             Nodes: [2x2903 double]
          Elements: [6x1365 double]
    MaxElementSize: 0.0509
    MinElementSize: 0.0254
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

Plot the resulting mesh.

```
pdeplot(m3)
```



Input Arguments

`model` — Model container

`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectromagneticModel` object

Model container, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectromagneticModel` object.

Example: `model = createpde(3)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","electrostatic")`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `generateMesh(model,"Hmax",0.25);`

`GeometricOrder` — Element geometric order
`"quadratic"` (default) | `"linear"`

Element geometric order, specified as `"linear"` or `"quadratic"`.

A triangle or tetrahedron representing a linear element has nodes at the corners. A triangle or tetrahedron representing a quadratic element has nodes at its corners and edge centers. The center nodes in quadratic meshes are always added at half-distance between corners. For geometries with curved surfaces and edges, center nodes might not appear on the edge or surface itself.

In general, `"quadratic"` elements produce more accurate solutions. Override the default `"quadratic"` only to solve a 3-D magnetostatic problem, to save memory, or to solve a 2-D problem using a legacy solver. Legacy PDE solvers use linear triangular mesh for 2-D geometries.

Example: `generateMesh(model,"GeometricOrder","linear");`

Data Types: `char` | `string`

`Hgrad` — Mesh growth rate
`1.5` (default) | number greater than or equal to 1 and less than or equal to 2

Mesh growth rate, specified as a number greater than or equal to 1 and less than or equal to 2.

Example: `generateMesh(model,"Hgrad",1.3);`

Data Types: `double`

`Hmax` — Target maximum mesh edge length
positive number

Target maximum mesh edge length, specified as a positive number.

`Hmax` is an approximate upper bound on the mesh edge lengths. Occasionally, `generateMesh` can create a mesh with some elements that exceed `Hmax`.

`generateMesh` estimates the default value of `Hmax` from overall dimensions of the geometry.

Small `Hmax` values let you create finer meshes, but mesh generation can take a very long time in this case. You can interrupt mesh generation by using **Ctrl+C**. Note that `generateMesh` can take additional time to respond to the interrupt.

Example: `generateMesh(model,"Hmax",0.25);`

Data Types: `double`

`Hmin` — Target minimum mesh edge length
nonnegative number

Target minimum mesh edge length, specified as a nonnegative number.

`Hmin` is an approximate lower bound on the mesh edge lengths. Occasionally, `generateMesh` can create a mesh with some elements that are smaller than `Hmin`.

`generateMesh` estimates the default value of `Hmin` from overall dimensions of the geometry.

Example: `generateMesh(model,"Hmin",0.05);`

Data Types: `double`

`Hface` — Target size on selected faces
cell array

Target size on selected faces, specified as a cell array containing an even number of elements. Odd-indexed elements are positive integers or vectors of positive integers specifying face IDs. Even-indexed elements are positive numbers specifying the target size for the corresponding faces.

Example: `generateMesh(model,"Hmax",0.25,"Hface",{[1 2],0.1,[3 4 5],0.05})`

Data Types: `double`

`Hedge` — Target size around selected edges
cell array

Target size around selected edges, specified as a cell array containing an even number of elements. Odd-indexed elements are positive integers or vectors of positive integers specifying edge IDs. Even-indexed elements are positive numbers specifying the target sizes for the corresponding edges.

Example: `generateMesh(model,"Hmax",0.25,"Hedge",{[1 2],0.01,3,0.05})`

Data Types: `double`

`Hvertex` — Target size around selected vertices
cell array

Target size around selected vertices, specified as a cell array containing an even number of elements. Odd-indexed elements are positive integers or vectors of positive integers specifying vertex IDs. Even-indexed elements are positive numbers specifying the target sizes for the corresponding vertices.

Example: `generateMesh(model,"Hmax",0.25,"Hvertex",{1,0.02})`

Data Types: `double`

Output Arguments

`mesh` — Mesh description
`FEMesh` object

Mesh description, returned as an FEMesh object. `mesh` is the same as `model.Mesh`.

More About

Element

An element is a basic unit in the finite-element method.

For 2-D problems, an element is a triangle 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 details, see “Mesh Data” on page 2-175.

Tips

- `generateMesh` can 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 or node and element counts.
- `generateMesh` uses the following set of rules when you specify local element sizes with `Hface`, `Hedge`, or `Hvertex`. These rules are valid for both the default and custom values of `Hmin` and `Hmax`.
  - If you specify local sizes for regions near each other, `generateMesh` uses the minimum size. For example, if you specify size 1 on an edge and size 0.5 on one of its vertices, the function gradually reduces the element sizes in the proximity of that vertex.
  - If you specify local sizes smaller than `Hmin`, `generateMesh` ignores `Hmin` in those localities.
  - If you specify local sizes larger than `Hmax`, `generateMesh` ignores the specified local sizes. `Hmax` is not exceeded anywhere in the mesh.

Version History

Introduced in R2015a

Local mesh refinement

You can now specify local target mesh size around geometric vertices, edges, and faces. The function generates a mesh with the element sizes around the specified regions as close to the target sizes as possible, and gradually blends the mesh between regions with different element sizes.

Improved mesh generation

The mesh generator now uses new mesh generation algorithm for 2-D geometries. It also lets you specify mesh growth rate by using the `Hgrad` argument.

Resulting meshes can differ from meshes generated in previous releases. For example, meshes generated with the default size controls can have fewer elements than before. Also, `generateMesh` creates quadratic meshes for 2-D problems by default. In previous releases, the default mesh for 2-D geometries is a linear mesh.

`Jiggle`, `JiggleIter`, and `MesherVersion` arguments are ignored

`generateMesh` ignores the `Jiggle`, `JiggleIter`, and `MesherVersion` arguments. The mesher produces good quality meshes without jiggling the nodes.

Quadratic elements for 2-D meshes

Not recommended starting in R2016a

Generate a quadratic 2-D mesh by setting `GeometricOrder` to `'quadratic'`.

See Also

FEMesh | `geometryFromEdges` | `importGeometry` | `PDEModel`

Topics

“Solve Problems Using PDEModel Objects” on page 2-2
“Finite Element Method Basics” on page 1-11
“Mesh Data” on page 2-175
“Mesh Data as [p,e,t] Triples” on page 2-172

GeometricInitialConditions Properties

Initial conditions over a region or region boundary

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

Properties

`RegionType` — Region type
`'face'` | `'cell'`

Region type, specified as `'face'` for a 2-D region, or `'cell'` for a 3-D region.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified 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`

`InitialValue` — Initial value
scalar | vector | function handle

Initial value, specified 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, specified as a scalar, vector, or function handle. For details, see `setInitialConditions`.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

Version History

Introduced in R2016a

See Also

`findInitialConditions` | `setInitialConditions` | NodalInitialConditions

Topics

"Set Initial Conditions" on page 2-114
"View, Edit, and Delete Initial Conditions" on page 2-123
"Solve Problems Using PDEModel Objects" on page 2-2

GeometricStructuralICs Properties

Initial displacement and velocity over a region

Description

A `GeometricStructuralICs` object contains a description of the initial displacement and velocity over a geometric region for a transient structural model. A `StructuralModel` container has a vector of `GeometricStructuralICs` objects in its `InitialConditions.StructuralICAssignments` property.

To set initial conditions for your structural model, use the `structuralIC` function.

Properties

Properties

`RegionType` — Geometric region type
`'Face'` | `'Edge'` | `'Vertex'` | `'Cell'` for a 3-D model

Geometric region type, specified as `'Face'`, `'Edge'`, or `'Vertex'` for a 2-D model or 3-D model, or `'Cell'` for a 3-D model.

Data Types: `char` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Data Types: `double`

`InitialDisplacement` — Initial displacement
numeric vector | function handle

Initial displacement, specified as a numeric vector or function handle. For details, see `structuralIC`.

Data Types: `double` | `function_handle`

`InitialVelocity` — Initial velocity
numeric vector | function handle

Initial velocity, specified as a numeric vector or function handle. For details, see `structuralIC`.

Data Types: `double` | `function_handle`

Version History

Introduced in R2018a

See Also

`structuralIC` | `findStructuralIC` | NodalStructuralICs Properties

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

Properties

`RegionType` — Region type
`'Vertex'` | `'Edge'` | `'Face'` | `'Cell'`

Region type, specified as `'Vertex'`, `'Edge'`, or `'Face'` for a 2-D or 3-D region, or `'Cell'` for a 3-D region.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified 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, specified as a scalar, vector, or function handle. For details, see `thermalIC`.

Data Types: `double` | `function_handle`

Version History
Introduced in R2017a

See Also
`thermalIC` | `findThermalIC` | NodalThermalICs

NodalInitialConditions Properties

Initial conditions at mesh nodes

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

Properties

`InitialValue` — Initial value
scalar | vector | function handle

Initial value, specified 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, specified as a scalar, vector, or function handle. For details, see `setInitialConditions`.

Data Types: `double` | `function_handle`
Complex Number Support: Yes

Version History

Introduced in R2016b

See Also

`findInitialConditions` | `setInitialConditions` | GeometricInitialConditions

Topics
“Set Initial Conditions” on page 2-114
“View, Edit, and Delete Initial Conditions” on page 2-123
“Solve Problems Using PDEModel Objects” on page 2-2

NodalStructuralICs Properties

Initial displacement and velocity at mesh nodes

Description

A `NodalStructuralICs` object contains a description of the initial displacement and velocity at mesh nodes. A `StructuralModel` container has a vector of `GeometricStructuralICs` objects in its `InitialConditions.StructuralICAssignments` property.

To set initial conditions for your structural model, use the `structuralIC` function.

Properties

Properties

`InitialDisplacement` — Initial displacement
numeric vector | function handle

Initial displacement, specified as a numeric vector or function handle. For details, see `structuralIC`.

Data Types: `double` | `function_handle`

`InitialVelocity` — Initial velocity
numeric vector | function handle

Initial velocity, specified as a numeric vector or function handle. For details, see `structuralIC`.

Data Types: `double` | `function_handle`

Version History

Introduced in R2018a

See Also

`structuralIC` | `findStructuralIC` | GeometricStructuralICs Properties

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

Properties

`InitialTemperature` — Initial temperature
scalar | vector | function handle

Initial temperature, specified as a scalar, vector, or function handle. For details, see `thermalIC`.

Data Types: `double` | `function_handle`

Version History

Introduced in R2017a

See Also

`thermalIC` | `findThermalIC` | GeometricThermalICs

geometryFromEdges

Package: `pde`

Create 2-D geometry from decomposed geometry matrix

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.

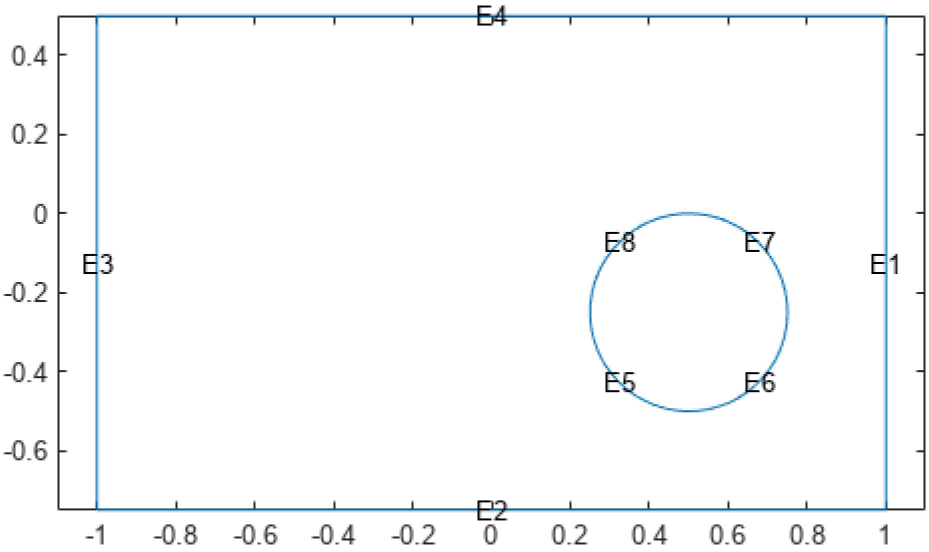
```
R1 = [3,4,-1,1,1,-1,0.5,0.5,-0.75,-0.75]';
C1 = [1,0.5,-0.25,0.25]';
C1 = [C1;zeros(length(R1) - length(C1),1)];
gm = [R1,C1];
sf = 'R1-C1';
```

Create the geometry.

```
ns = char('R1','C1');
ns = ns';
g = decsg(gm,sf,ns);
```

Include the geometry in the model and plot it.

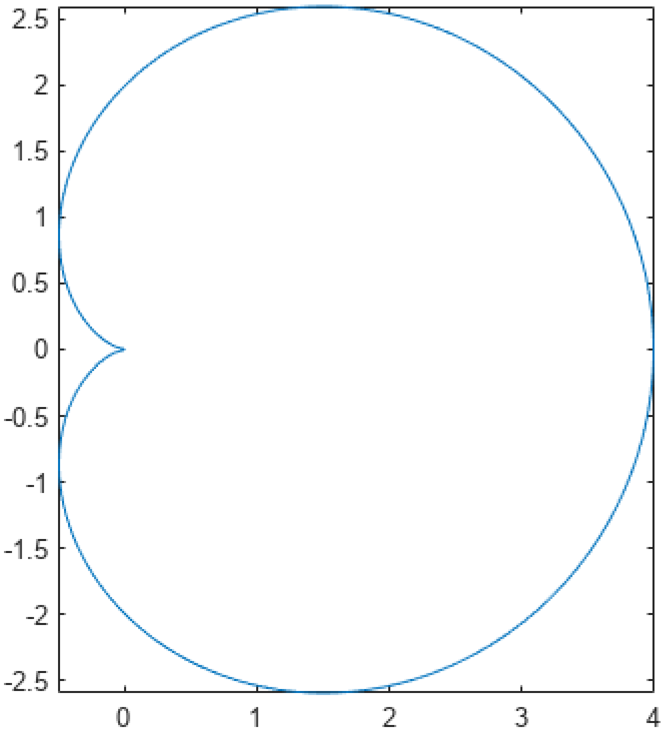
```
geometryFromEdges(model,g);
pdegplot(model,"EdgeLabels","on")
axis equal
xlim([-1.1,1.1])
```



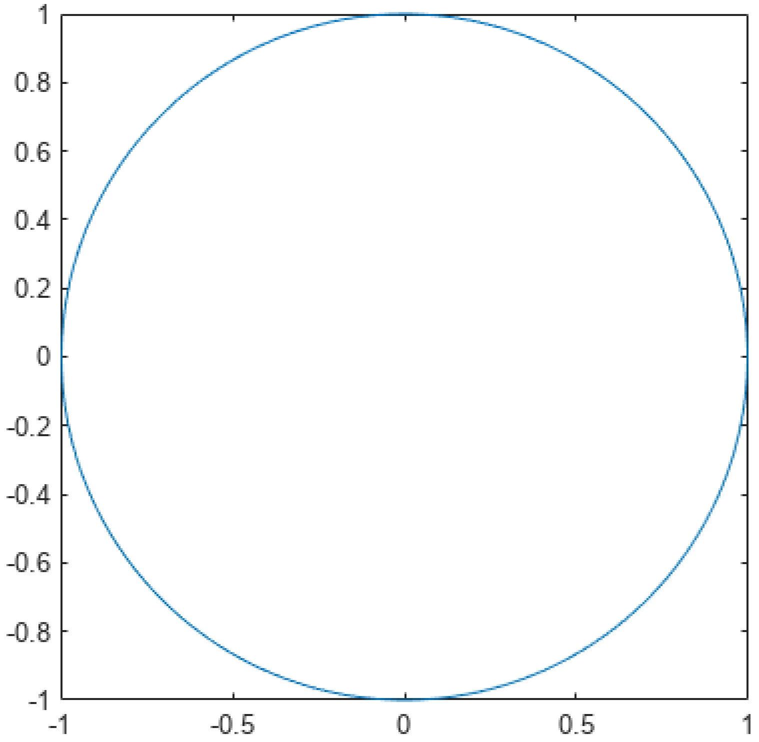
Predefined Geometry Functions

The toolbox provides the several geometry functions. Specify them by using the following function handles.

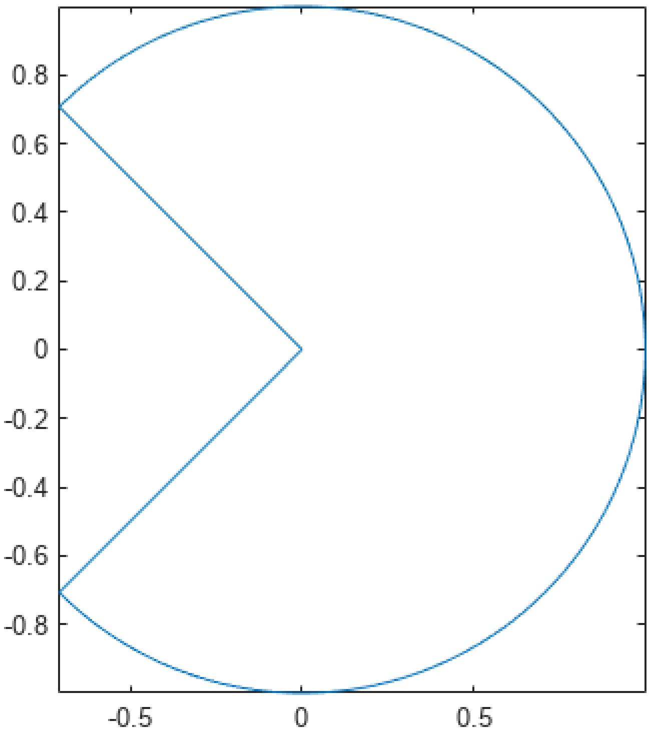
```
model = createpde;
g = geometryFromEdges(model,@cardg);
pdegplot(model)
```



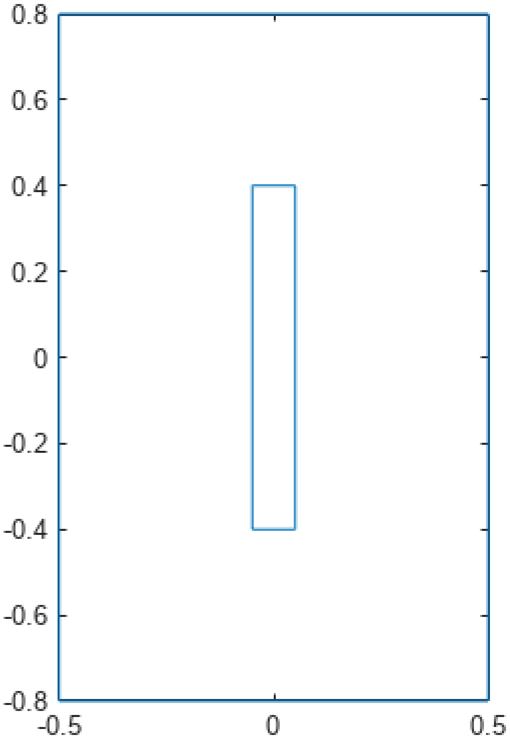
```
clear model
model = createpde;
g = geometryFromEdges(model,@circleg);
pdegplot(model)
```



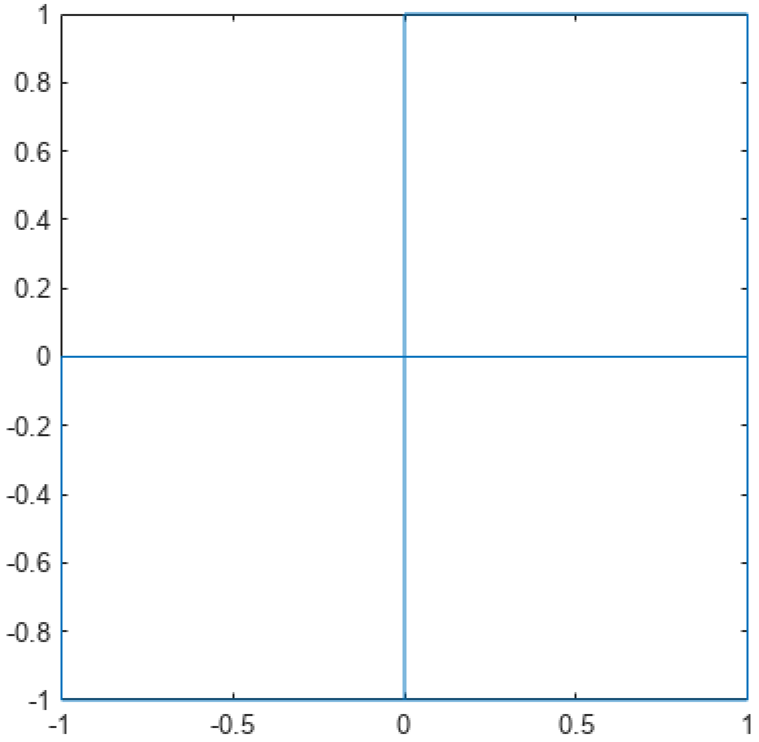
```
clear model
model = createpde;
g = geometryFromEdges(model,@cirsg);
pdegplot(model)
```



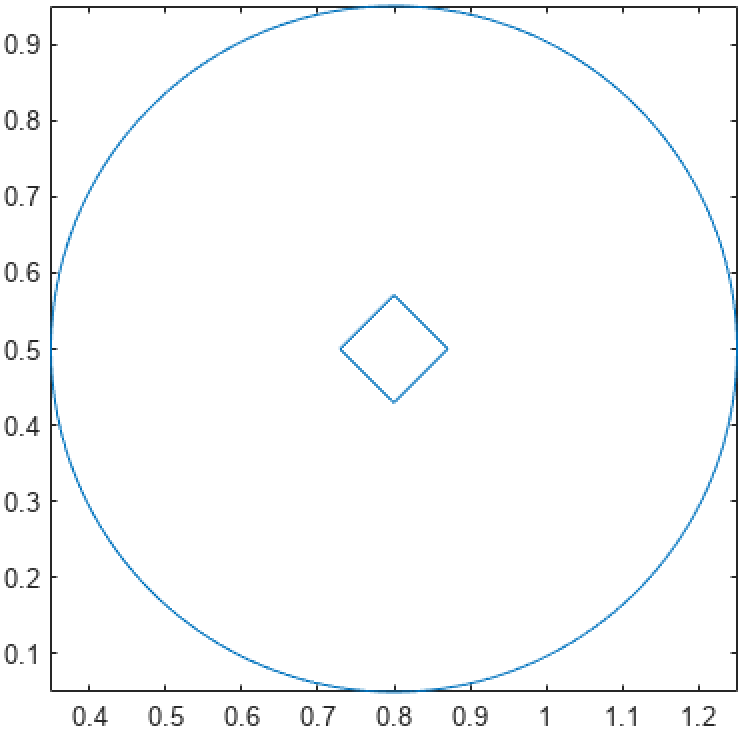
```
clear model
model = createpde;
g = geometryFromEdges(model,@crackg);
pdegplot(model)
```



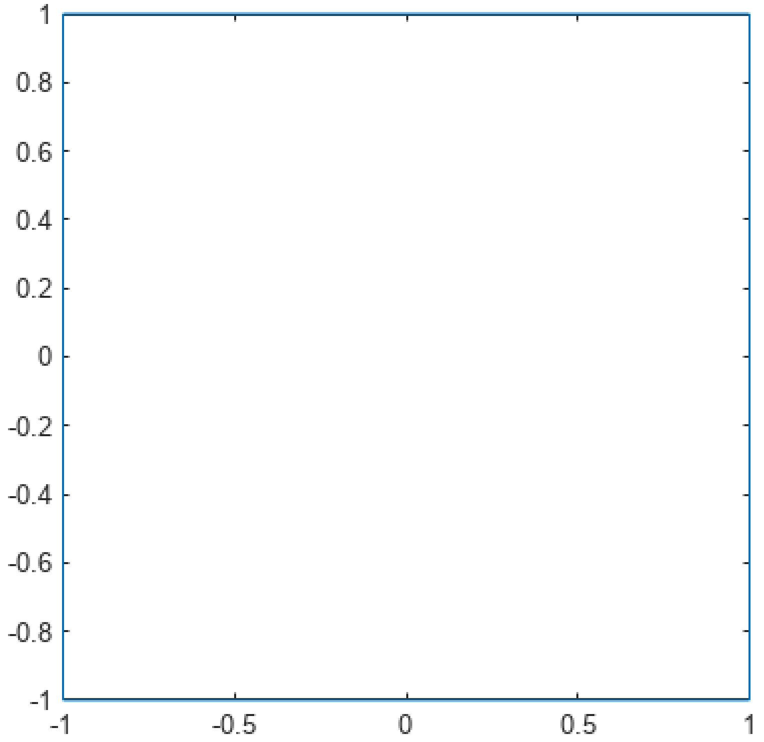
```
clear model
model = createpde;
g = geometryFromEdges(model,@lshapeg);
pdegplot(model)
```



```
clear model
model = createpde;
g = geometryFromEdges(model,@scatterg);
pdegplot(model)
```



```
clear model
model = createpde;
g = geometryFromEdges(model,@squareg);
pdegplot(model)
```



Input Arguments

`model` — Model container

`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectromagneticModel` object

Model container, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectromagneticModel` object.

Example: `model = createpde(3)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","electrostatic")`

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 about a decomposed geometry matrix, see `decsg`.

A geometry function must return the same result for the same input arguments in every function call. Thus, it must not contain functions and expressions designed to return a variety of results, such as random number generators.

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`.

Version History

Introduced in R2015a

See Also

AnalyticGeometry | `PDEModel`

Topics

“Solve PDEs with Constant Boundary Conditions” on page 2-135
“Solve Problems Using PDEModel Objects” on page 2-2

geometryFromMesh

Package: `pde`

Create 2-D or 3-D geometry from mesh

Syntax

```
geometryFromMesh(model,nodes,elements)
geometryFromMesh(model,nodes,elements,ElementIDToRegionID)
[G,mesh] = geometryFromMesh(model,nodes,elements)
```

Description

`geometryFromMesh(model,nodes,elements)` creates geometry within `model`. For planar and volume triangulated meshes, this function also incorporates `nodes` in the `model.Mesh.Nodes` property and `elements` in the `model.Mesh.Elements` property. To replace the imported mesh with a mesh having a different target element size, use `generateMesh`.

If `elements` represents a surface triangular mesh that bounds a closed volume, then `geometryFromMesh` creates the geometry, but does not incorporate the mesh into the corresponding properties of the model. To generate a mesh in this case, use `generateMesh`.

`geometryFromMesh(model,nodes,elements,ElementIDToRegionID)` creates a multidomain geometry. Here, `ElementIDToRegionID` specifies the subdomain IDs for each element of the mesh.

`[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`.

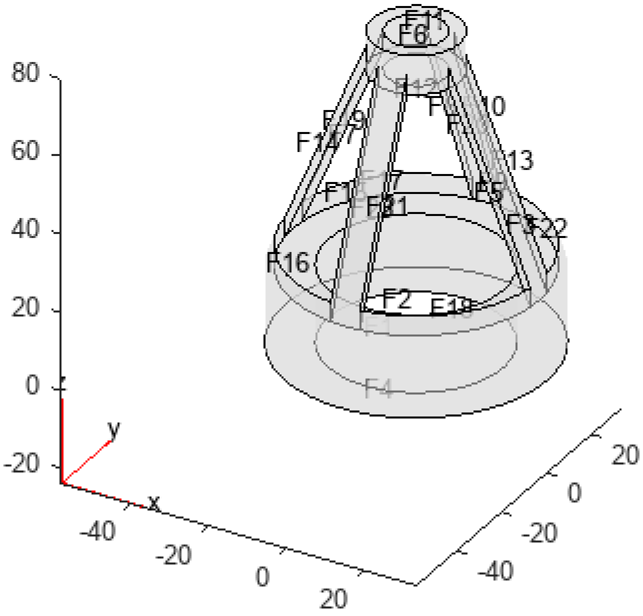
```
load tetmesh
nodes = X';
elements = tet';
```

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`.

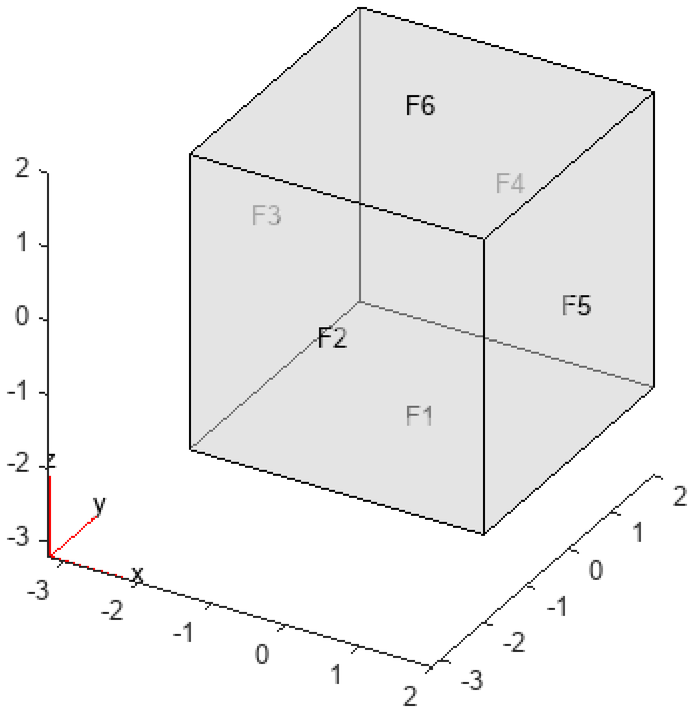
```
nodes = [x';y';z'];
elements = K';
```

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 a 3-D geometry using the MATLAB® `alphaShape` function. First, create an `alphaShape` object of a block with a cylindrical hole. Then 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`.

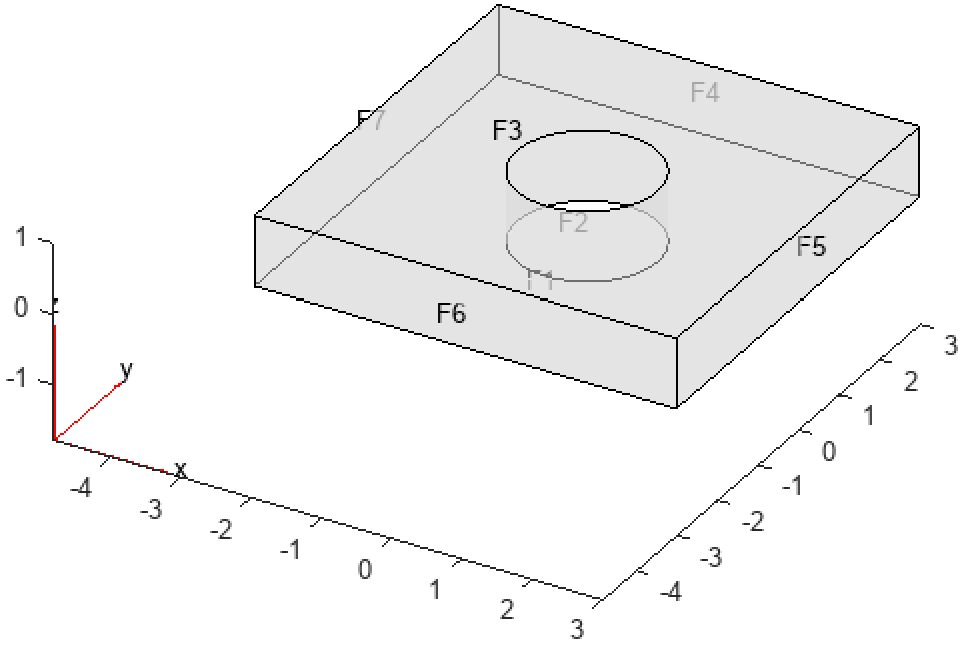
```
nodes = nodes';
elements = elements';
```

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);
```

2-D Multidomain Geometry

Create a 2-D multidomain geometry from a mesh.

Load information about nodes, elements, and element-to-domain correspondence into your workspace. The file `MultidomainMesh2D` ships with your software.

```
load MultidomainMesh2D
```

Create a PDE model.

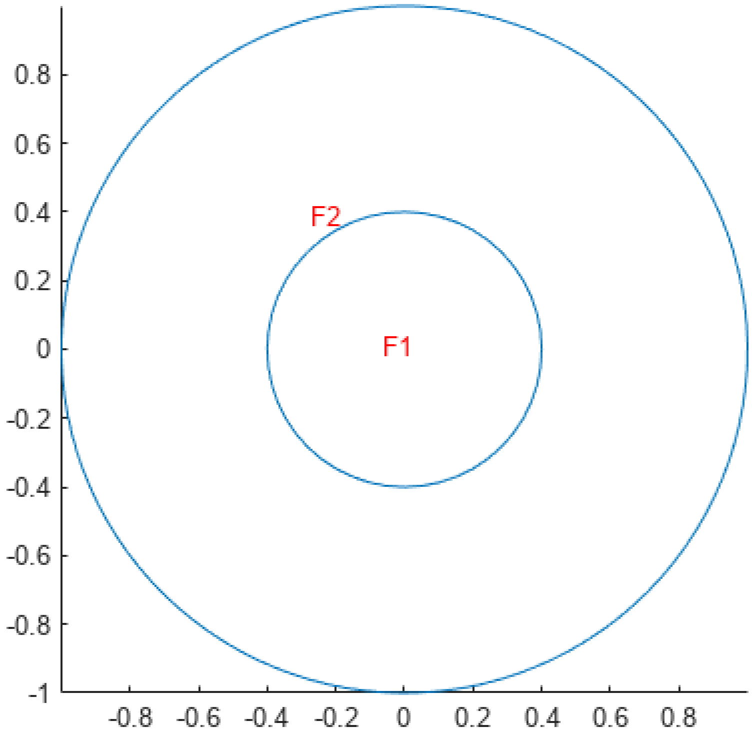
```
model = createpde;
```

Import the mesh into the model.

```
geometryFromMesh(model,nodes,elements,ElementIdToRegionId);
```

View the geometry and face numbers.

```
pdegplot(model,"FaceLabels","on")
```



3-D Multidomain Geometry

Create a 3-D multidomain geometry from a mesh.

Load information about nodes, elements, and element-to-domain correspondence into your workspace. The file `MultidomainMesh3D` ships with your software.

```
load MultidomainMesh3D
```

Create a PDE model.

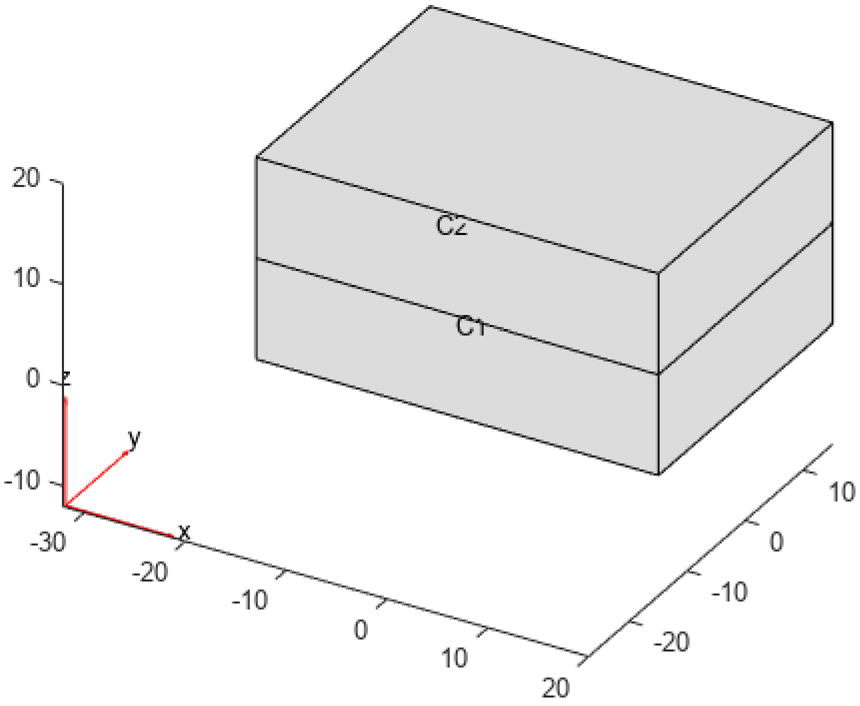
```
model = createpde;
```

Import the mesh into the model.

```
geometryFromMesh(model,nodes,elements,ElementIdToRegionId);
```

View the geometry and cell numbers.

```
pdegplot(model,"CellLabels","on")
```



Input Arguments

`model` — Model container

`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectromagneticModel` object

Model container, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectromagneticModel` object.

Example: `model = createpde(3)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","electrostatic")`

nodes — Mesh nodes

matrix of real numbers

Mesh nodes, specified as a matrix of real numbers. The matrix size is 2-by-`Nnodes` for a 2-D case and 3-by-`Nnodes` for a 3-D case. `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 | 6-by-`Nelements` integer matrix | 10-by-`Nelements` integer matrix

Mesh elements, specified as an integer matrix with 3, 4, 6, or 10 rows, and `Nelements` columns, where `Nelements` is the number of elements in the mesh.

- Linear planar mesh or linear 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 planar mesh or quadratic mesh on the geometry surface has size 6-by-`Nelements`. Each column of `elements` contains the indices of the triangle corner nodes and edge centers for a surface element. In this case, the resulting geometry does not contain a full mesh. Create the mesh using the `generateMesh` function.
- 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 details on node numbering for linear and quadratic elements, see “Mesh Data” on page 2-175.

Data Types: `double`

`ElementIDToRegionID` — Domain information for each element

vector of positive integers

Domain information for each mesh element, specified as a vector of positive integers. Each element is an ID of a geometric region for an element of the mesh. The length of this vector equals the number of elements in the mesh.

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` is a 3-by-`Nelements` matrix representing a surface mesh, then `mesh` is `[]`. In this case, create a mesh for the geometry using the `generateMesh` function.
- If `elements` is a matrix with more than three rows representing a volume mesh, 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.

Version History

Introduced in R2015b

Multidomain geometry from nodes and elements

The function now lets you create a multidomain geometry by specifying the subdomain ID for each element of the mesh.

See Also

`alphaShape` | DiscreteGeometry | `generateMesh` | `importGeometry`

Topics
“STL File Import” on page 2-43
“Solve Problems Using PDEModel Objects” on page 2-2

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

Properties

`RegionType` — Region type
`'Face'` | `'Cell'`

Region type, specified as `'Face'` for a 2-D region, or `'Cell'` for a 3-D region.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified 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, specified as a number or a function handle. A heat source with a negative value is called a heat sink.

Data Types: `double` | `function_handle`

`Label` — Label for use with `linearizeInput`
character vector | string

Label for use with `linearizeInput`, specified as a character vector or a string.

Data Types: `char` | `string`

Version History

Introduced in R2017a

See Also

`findHeatSource` | `internalHeatSource`

hyperbolic

(Not recommended) Solve hyperbolic PDE problem

Note `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

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.

`u = hyperbolic(u0,ut0,tlist,model,c,a,f,d)` produces the solution to the FEM formulation of the scalar PDE problem

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

on a 2-D or 3-D region Ω, or the system PDE problem

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

with geometry, mesh, and boundary conditions specified in `model`, with initial value `u0` and initial derivative with respect to time `ut0`. The variables *c*, *a*, *f*, and *d* in the equation correspond to the function coefficients `c`, `a`, `f`, and `d` respectively.

`u = hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d)` solves the problem using boundary conditions `b` and finite element mesh specified in `[p,e,t]`.

`u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M)` solves the problem based on finite element matrices that encode the equation, mesh, and boundary conditions.

`u = hyperbolic(___,rtol)` and `u = hyperbolic(___,rtol,atol)` modify the solution process by passing to the ODE solver a relative tolerance `rtol`, and optionally an absolute tolerance `atol`.

`u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M,___,'DampingMatrix',D)` modifies the problem to include a damping matrix `D`.

`u = hyperbolic(___,'Stats','off')` turns off the display of internal ODE solver statistics during the solution process.

Examples

Hyperbolic Equation

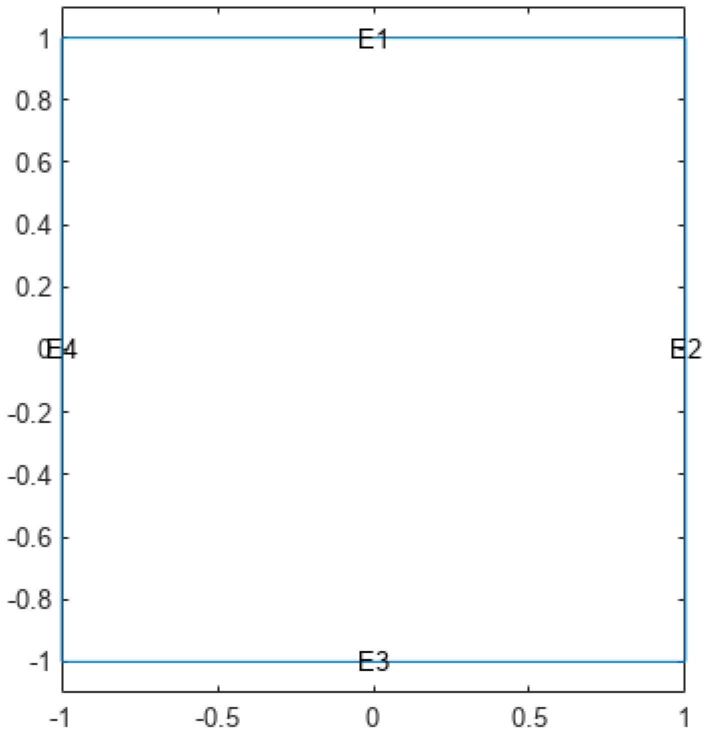
Solve the wave equation

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

on the square domain specified by `squareg`.

Create a PDE model and import the geometry.

```
model = createpde;
geometryFromEdges(model,@squareg);
pdegplot(model,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```



Set Dirichlet boundary conditions $u = 0$ for $x = \pm 1$, and Neumann boundary conditions

$$\nabla u \cdot \mathbf{n} = \mathbf{0}$$

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;
```

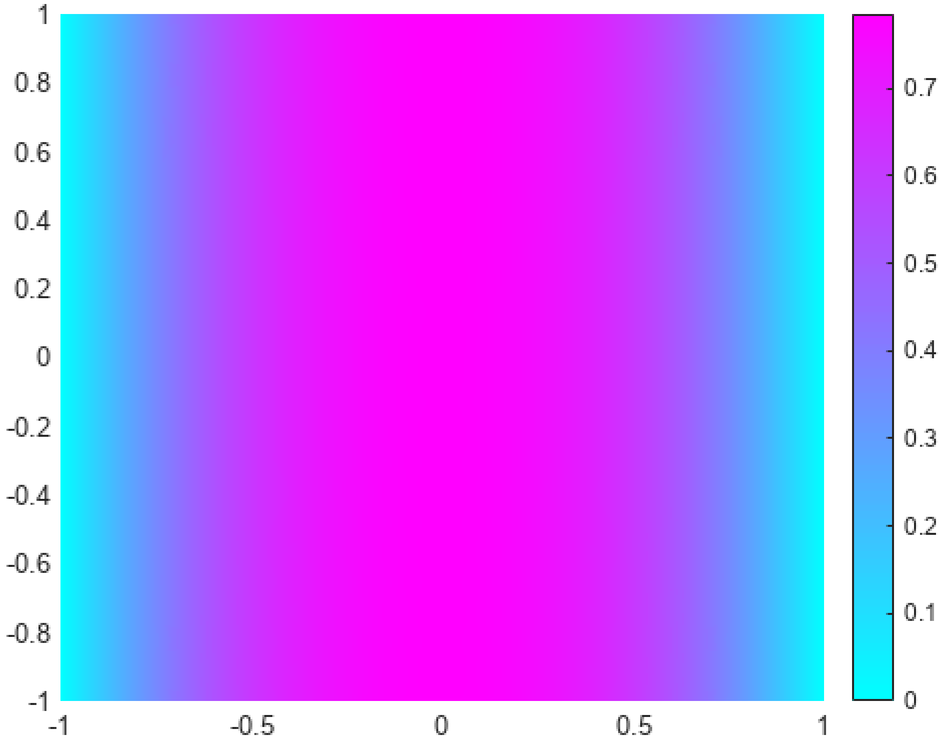
Generate a mesh and solve the PDE.

```
generateMesh(model,'GeometricOrder','linear','Hmax',0.1);
u1 = hyperbolic(u0,ut0,tlist,model,c,a,f,d);
```

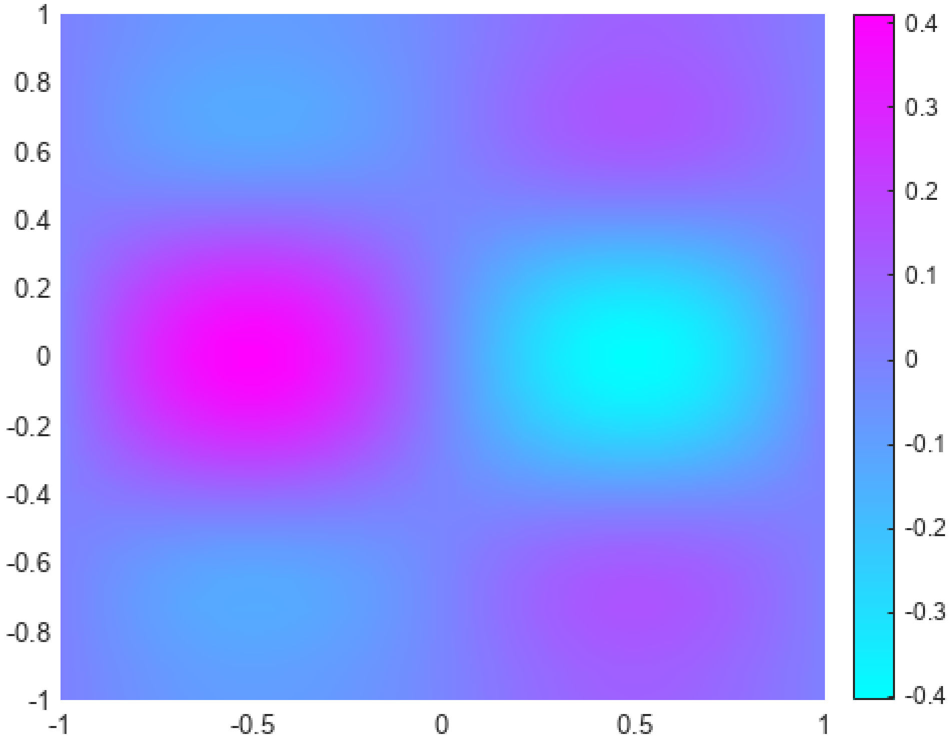
```
462 successful steps
51 failed attempts
1028 function evaluations
1 partial derivatives
135 LU decompositions
1027 solutions of linear systems
```

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 Square Domain” on page 3-298.

Hyperbolic Equation using Legacy Syntax

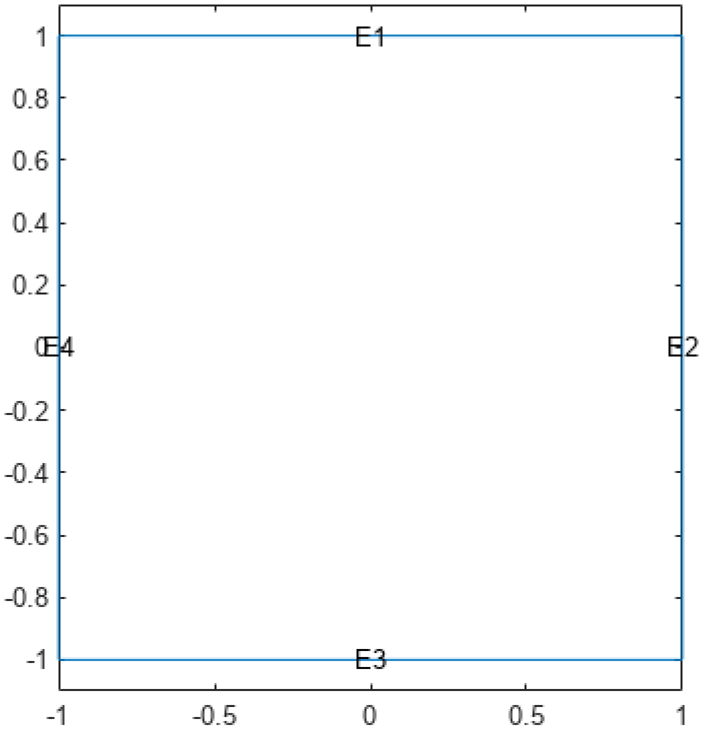
Solve the wave equation

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

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

$$\nabla u \cdot \mathbf{n} = \mathbf{0}$$

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;
```

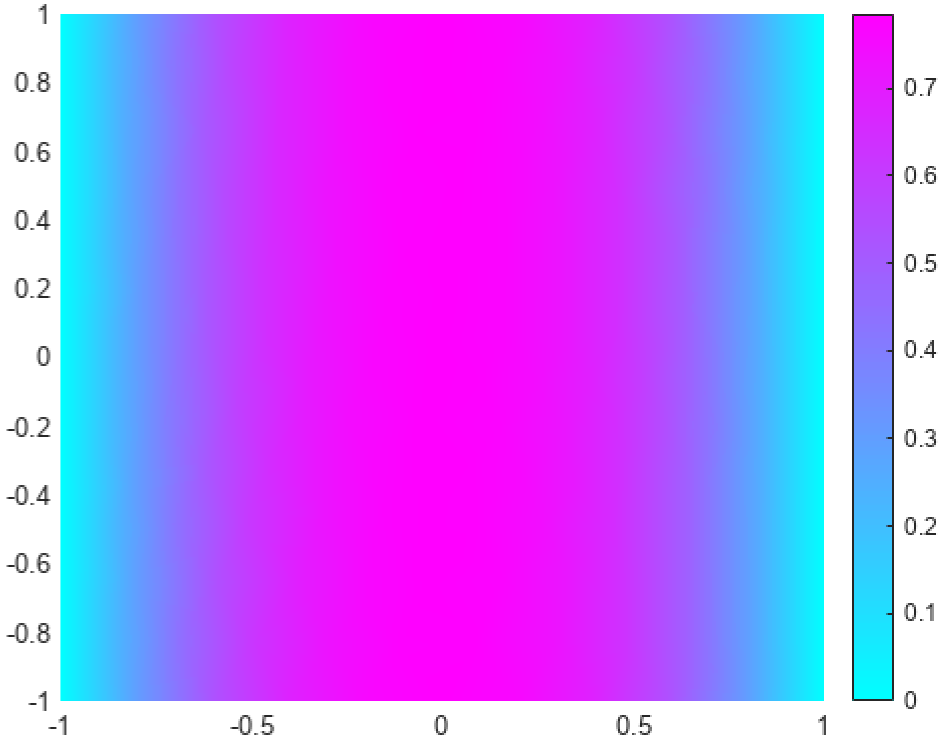
Create a mesh and solve the PDE.

```
[p,e,t] = initmesh(g);
u = hyperbolic(u0,ut0,tlist,b,p,e,t,c,a,f,d);
```

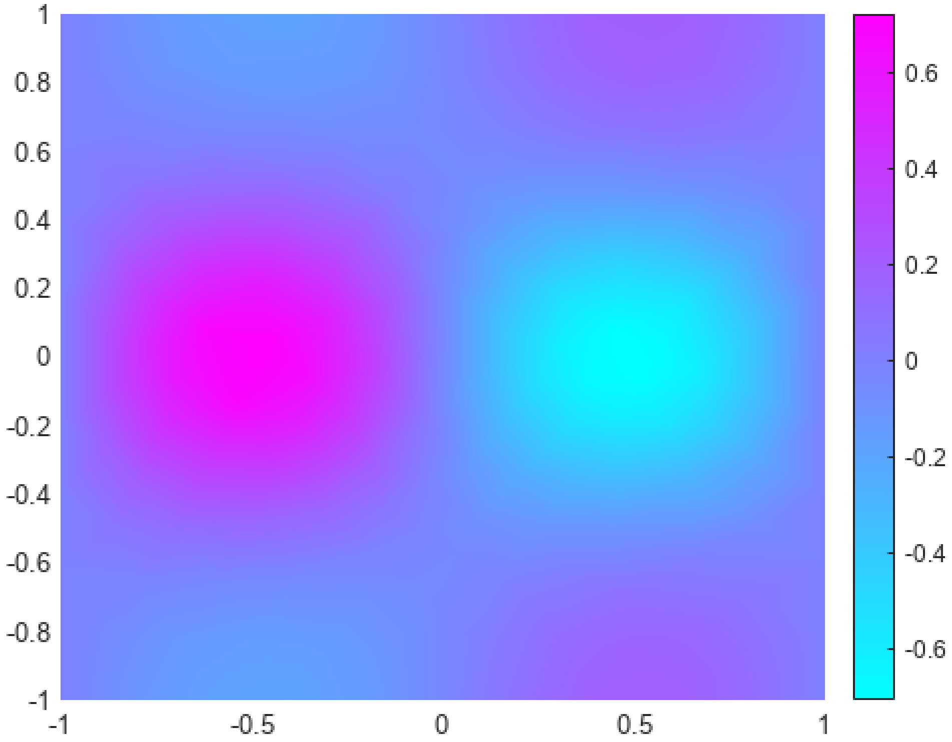
```
462 successful steps
70 failed attempts
1066 function evaluations
1 partial derivatives
156 LU decompositions
1065 solutions of linear systems
```

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))
```



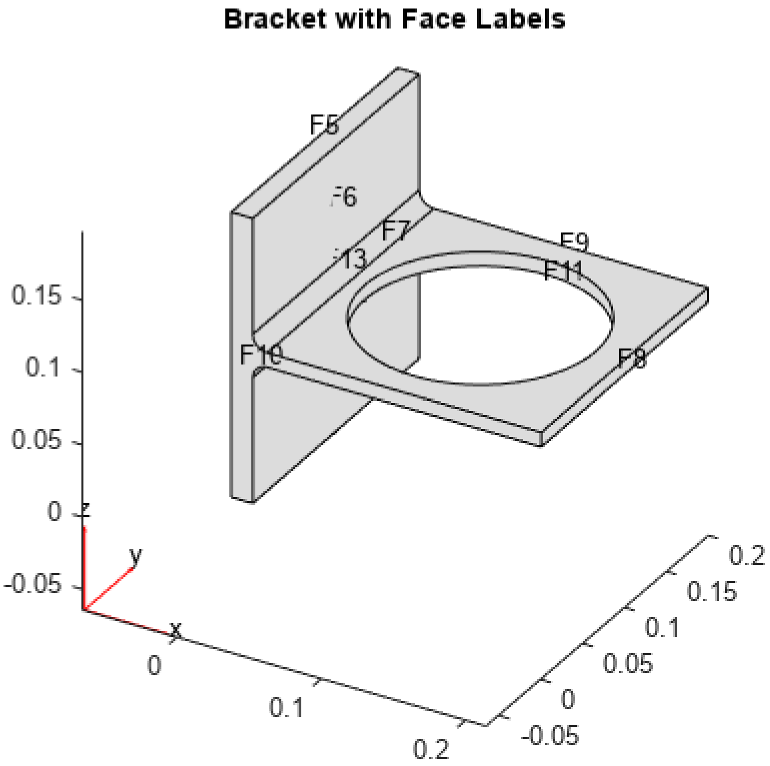
For a version of this example with animation, see “Wave Equation on Square Domain” on page 3-298.

Hyperbolic Solution Using Finite Element Matrices

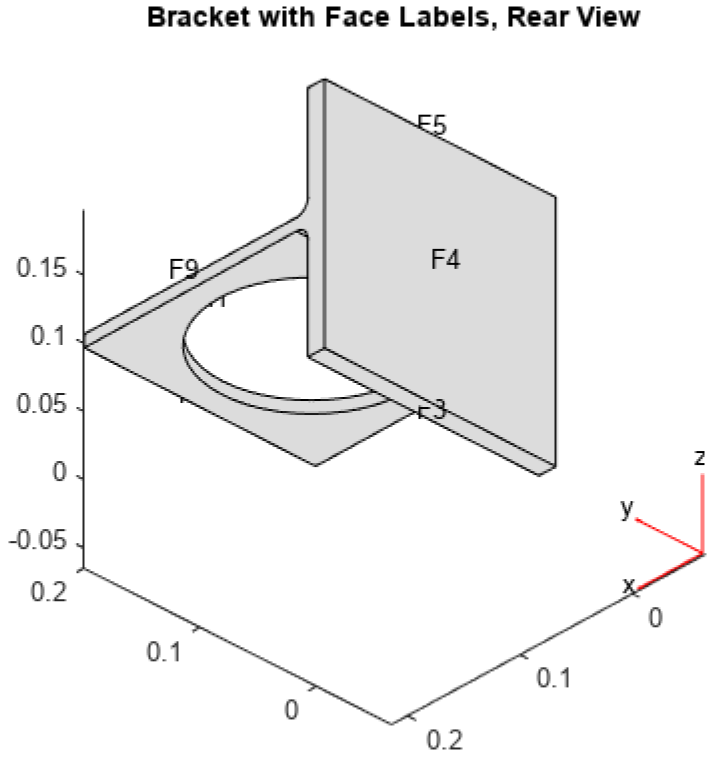
Solve a hyperbolic problem using finite element matrices.

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')
```



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);
```

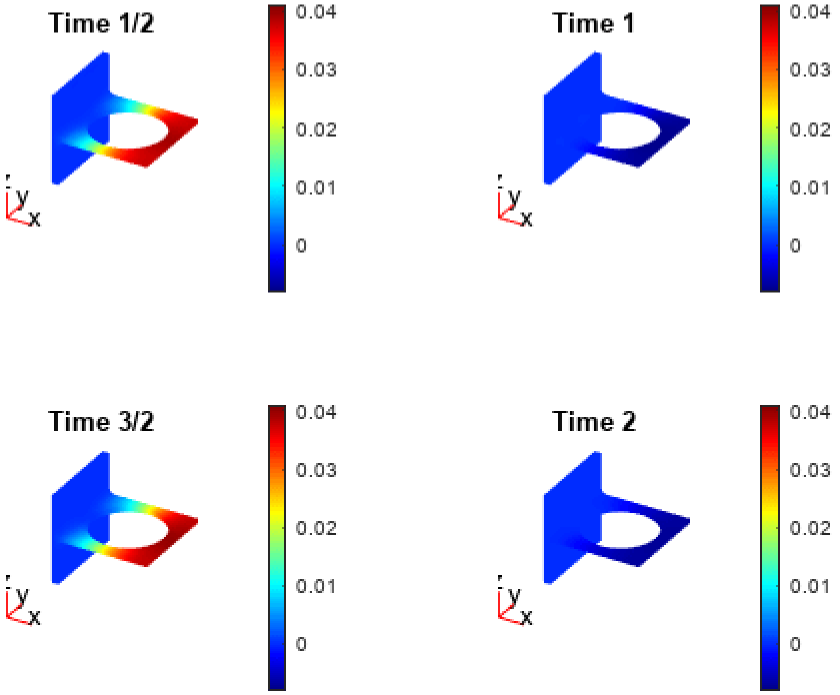
Solve the PDE for times from 0 to 2.

```
tlist = linspace(0,5,50);
u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M);
```

```
1493 successful steps
70 failed attempts
2970 function evaluations
1 partial derivatives
276 LU decompositions
2969 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')
```



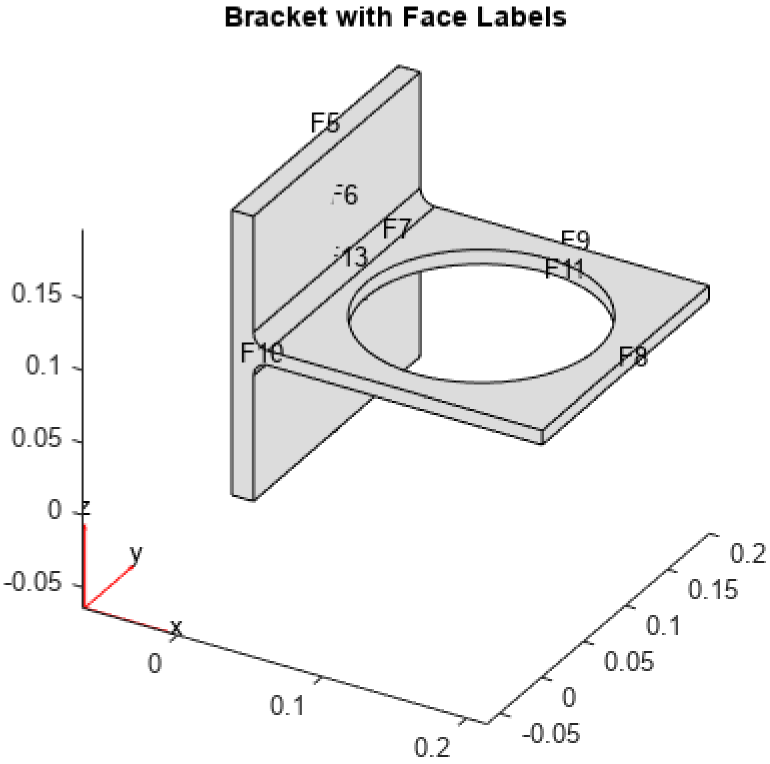
The solution seems to have a frequency of one, because the plots at times 1/2 and 3/2 show maximum values, and those at times 1 and 2 show minimum values.

Hyperbolic Equation with Damping

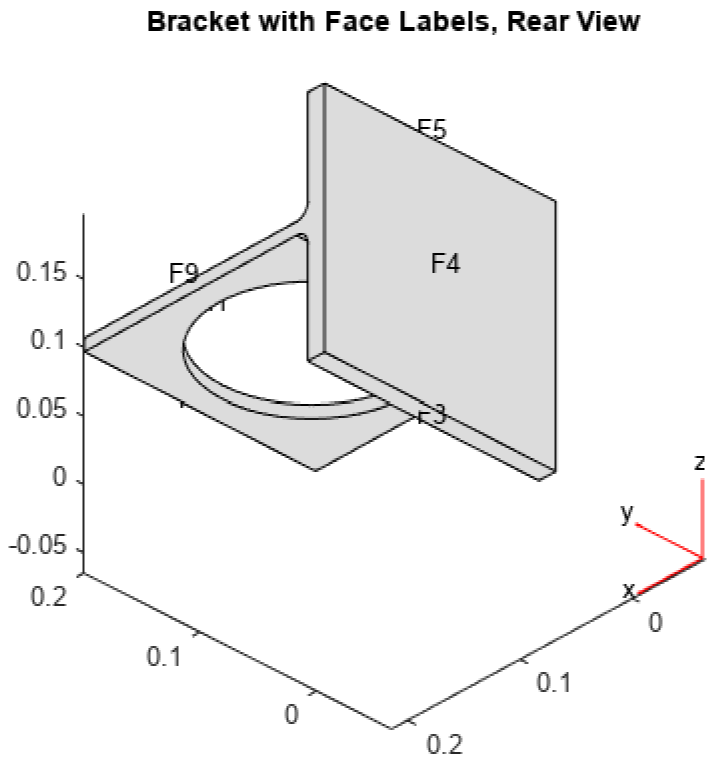
Solve a hyperbolic problem that includes damping. You must use the finite element matrix form to use damping.

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')
```



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;
```

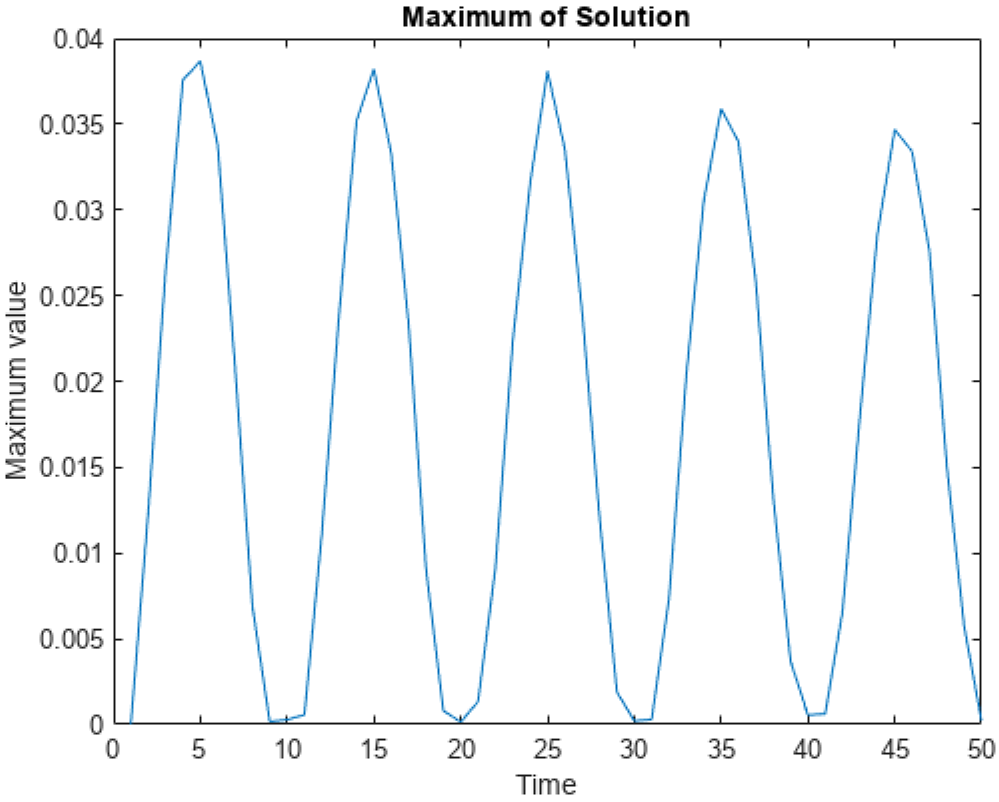
Solve the PDE for times from 0 to 2.

```
tlist = linspace(0,5,50);
u = hyperbolic(u0,ut0,tlist,Kc,Fc,B,ud,M,'DampingMatrix',Damping);
```

```
1441 successful steps
70 failed attempts
2828 function evaluations
1 partial derivatives
288 LU decompositions
2827 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 | character vector | character array | string scalar | string vector

Initial condition, specified as a scalar, vector of nodal values, character vector, character array, string scalar, or string vector. 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`.

- 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` | `string`
Complex Number Support: Yes

`ut0` — Initial derivative

vector | character vector | character array | string scalar | string vector

Initial derivative, specified as a vector, character vector, character array, string scalar, or string vector. 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`.

- 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)')
```

Example: `p(1,:).^2+5*cos(p(2,:).*p(1,:))`

Data Types: `double` | `char` | `string`
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`

c — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

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

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

a — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

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

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

f — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

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

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

d — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

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

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `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 Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples" on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 5-101. Typically, `Kc` is the output of `assempde`.

`Fc` — Load vector

vector

Load vector, specified as a vector. See "Elliptic Equations" on page 5-101. Typically, `Fc` is the output of `assempde`.

`B` — Dirichlet nullspace

sparse matrix

Dirichlet nullspace, returned as a sparse matrix. See "Algorithms" on page 5-101. Typically, `B` is the output of `assempde`.

`ud` — Dirichlet vector

vector

Dirichlet vector, returned as a vector. See "Algorithms" on page 5-101. 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 5-101.

To obtain the input matrices for `pdeeig`, `hyperbolic` or `parabolic`, run both `assema` and `assempde`:

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Note Create the `M` matrix using `assema` with `d`, not `a`, as the argument before `f`.

Data Types: `double`
Complex Number Support: Yes

`rtol` — Relative tolerance for ODE solver

`1e-3` (default) | positive real

Relative tolerance for ODE solver, specified as a positive real.

Example: `2e-4`

Data Types: `double`

`atol` — Absolute tolerance for ODE solver

`1e-6` (default) | positive real

Absolute tolerance for ODE solver, specified as a positive real.

Example: `2e-7`

Data Types: `double`

`D` — Damping matrix

matrix

Damping matrix, specified as a matrix. `D` has the same size as the stiffness matrix `Kc` or the mass matrix `M`. When you include `D`, `hyperbolic` solves the following ODE for the variable v:

$$B^T MB\frac{d^2 v}{dt^2} + B^T DB\frac{dv}{dt} + Kv = F$$

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 Damped Cantilever Beam” on page 3-21.

Example: `alpha*M + beta*K`

Data Types: `double`
Complex Number Support: Yes

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 “3-D Solution and Gradient Plots with MATLAB Functions” on page 3-344.

Algorithms

Hyperbolic Equations

Partial Differential Equation Toolbox solves equations of the form

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

When the d coefficient is 0, but m is not, the documentation calls this a hyperbolic equation, whether or not it is mathematically of the hyperbolic form.

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

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

for $\mathbf{x}$ in Ω, where $\mathbf{x}$ represents a 2-D or 3-D point, with the initial conditions

$$\begin{aligned} u(\mathbf{x}, 0) &= u_0(\mathbf{x}) \\ \frac{\partial u}{\partial t}(\mathbf{x}, 0) &= v_0(\mathbf{x}) \end{aligned}$$

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

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

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

with the initial conditions

$$\begin{aligned} U_i(0) &= u_0(\mathbf{x}_i) \ \ \forall i \\ \frac{d}{dt}U_i(0) &= v_0(\mathbf{x}_i) \ \ \forall i \end{aligned}$$

after we eliminate the unknowns fixed by Dirichlet boundary conditions. As before, the stiffness matrix K and the mass matrix M are assembled with the aid of the function `assempde` from the problems

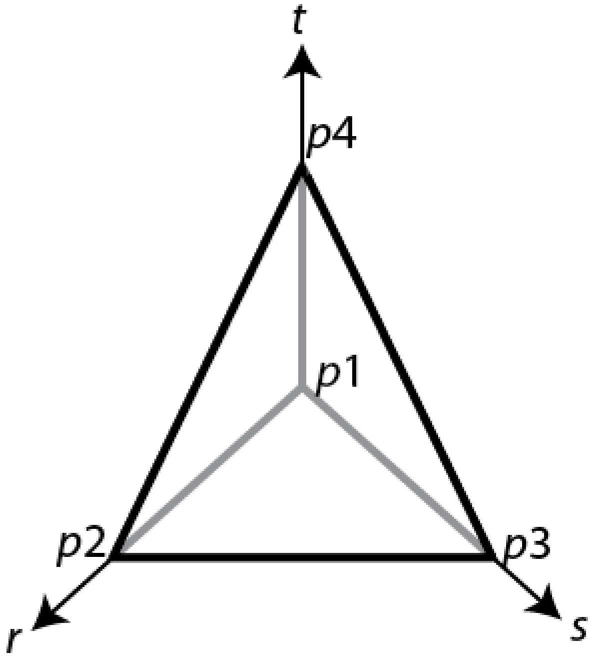
$$-\nabla \cdot (c\nabla u) + au = f \text{ and } -\nabla \cdot (0\nabla u) + mu = 0. \tag{5-3}$$

`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.

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 5-101. 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*).

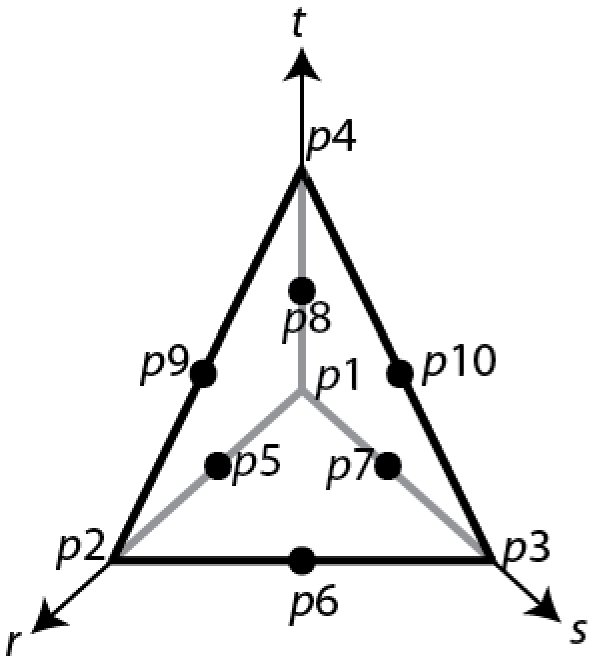


In local coordinates, the point *p*1 is at (0,0,0), *p*2 is at (1,0,0), *p*3 is at (0,1,0), and *p*4 is at (0,0,1).

For a linear tetrahedron, the basis functions are

$$\begin{aligned}\phi_1 &= 1 - r - s - t\\ \phi_2 &= r\\ \phi_3 &= s\\ \phi_4 &= t\end{aligned}$$

For a quadratic tetrahedron, there are additional nodes at the edge midpoints.



The corresponding basis functions are

$$\phi_1 = 2(1 - r - s - t)^2 - (1 - r - s - t)$$
$$\phi_2 = 2r^2 - r$$
$$\phi_3 = 2s^2 - s$$
$$\phi_4 = 2t^2 - t$$
$$\phi_5 = 4r(1 - r - s - t)$$
$$\phi_6 = 4rs$$
$$\phi_7 = 4s(1 - r - s - t)$$
$$\phi_8 = 4t(1 - r - s - t)$$
$$\phi_9 = 4rt$$
$$\phi_{10} = 4st$$

As in the 2-D case, a 3-D basis function ϕ_i takes the value 0 at all nodes j, except for node i, where it takes the value 1.

Systems of PDEs

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

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

the parabolic system

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

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

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

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

The elements c_{ijkl}, a_{ij}, d_{ij}, and f_i of $\mathbf{c}$, $\mathbf{a}$, $\mathbf{d}$, and $\mathbf{f}$ are stored row-wise in the MATLAB matrices `c`, `a`, `d`, and `f`. The case of identity, diagonal, and symmetric matrices are handled as special cases. For the tensor c_{ijkl} this applies both to the indices i and j, and to the indices k and l.

Partial Differential Equation Toolbox software does not check the ellipticity of the problem, and it is quite possible to define a system that is *not* elliptic in the mathematical sense. The preceding procedure that describes the scalar case is applied to each component of the system, yielding a symmetric positive definite system of equations whenever the differential system possesses these characteristics.

The boundary conditions now in general are *mixed*, i.e., for each point on the boundary a combination of Dirichlet and generalized Neumann conditions,

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

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

$$\mathbf{n} = (\cos(\alpha), \cos(\beta), \cos(\gamma))$$

There are M Dirichlet conditions and the **h**-matrix is M-by-N, $M \geq 0$. The generalized Neumann condition contains a source $\mathbf{h}'\mu$, where the Lagrange multipliers μ 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 5-101 for the scalar case, which is equivalent to a diagonal **h**-matrix. For the general case, Dirichlet conditions

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

are approximated by adding a term

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

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 **h**, and is free of the ill-conditioning, but is more involved computationally. Assume that there are N_p nodes in the mesh. Then the number of unknowns is $N_pN = N_u$. When Dirichlet boundary conditions fix some of the unknowns, the linear system can be correspondingly reduced. This is easily done by removing rows and columns when u values are given, but here we must treat the case when some linear combinations of the components of u are given, $\mathbf{hu} = \mathbf{r}$. These are collected into $HU = R$ where H is an M-by-N_u matrix and R is an M-vector.

With the reaction force term the system becomes

$KU + H'\mu = F$

$HU = R.$

The constraints can be solved for M of the U-variables, the remaining called V, an $N_u - M$ vector. The null space of H is spanned by the columns of B, and $U = BV + u_d$ makes U satisfy the Dirichlet conditions. A permutation to block-diagonal form exploits the sparsity of H to speed up the following computation to find B in a numerically stable way. μ can be eliminated by pre-multiplying 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.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`hyperbolic` is not recommended. Use `solvepde` instead. There are no plans to remove `hyperbolic`.

Coefficients of hyperbolic PDEs as functions of the solution and its gradient

You can now solve hyperbolic equations whose coefficients depend on the solution u or on the gradient of u.

See Also

`solvepde`

importGeometry

Package: `pde`

Import geometry from STL or STEP file

Syntax

```
gm = importGeometry(geometryfile)
gm = importGeometry(model,geometryfile)

importGeometry(model,___)
___ = importGeometry(___,"MaxRelativeDeviation",d)
```

Description

`gm = importGeometry(geometryfile)` creates a geometry object from the specified STL or STEP geometry file. A geometry imported from an STL file can be 3-D or planar. A geometry imported from a STEP file must be 3-D.

`gm = importGeometry(model,geometryfile)` also includes the geometry in the `model` container.

`importGeometry(model,___)` creates a geometry object from the specified STL or STEP geometry file and includes the geometry in the `model` container.

`___ = importGeometry(___,"MaxRelativeDeviation",d)` and lets you control the accuracy of the geometry import from a STEP file by specifying the relative sag. Use this syntax with any of the argument combinations from the previous syntaxes.

Examples

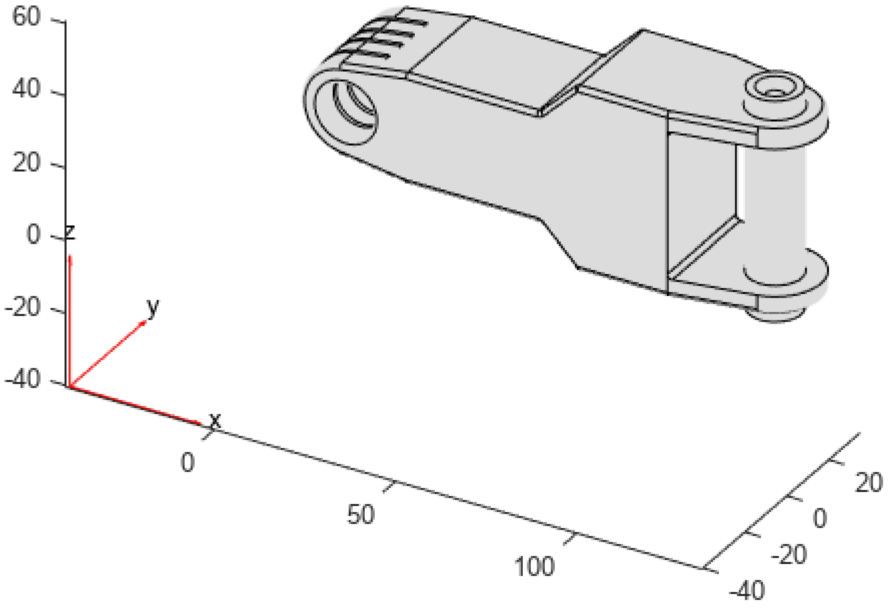
Import 3-D Geometry from STL File Without Creating Model

Create a geometry object from an STL geometry file.

```
gm = importGeometry("ForearmLink.stl");
```

Plot the geometry.

```
pdegplot(gm)
```



Import Planar Geometry from STL File into Model

Import a planar STL geometry and include it in a PDE model. When importing a planar geometry, `importGeometry` converts it to a 2-D geometry by mapping it to the *xy*-plane.

Create a `PDEModel` container.

```
model = createpde;
```

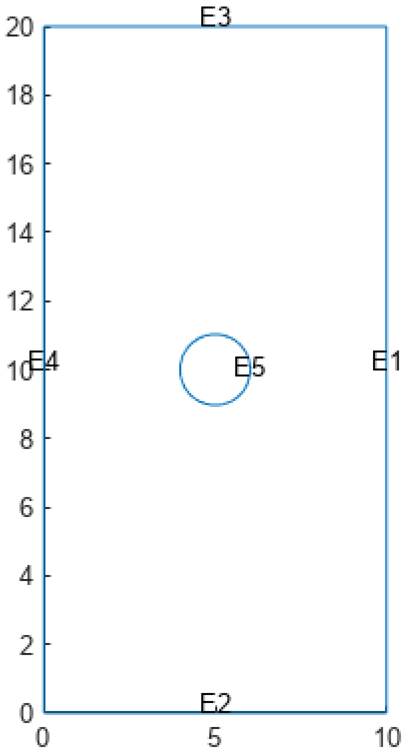
Import a geometry into the container.

```
importGeometry(model,"PlateHolePlanar.stl")
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 0
       NumFaces: 1
       NumEdges: 5
    NumVertices: 5
       Vertices: [5x3 double]
```

Plot the geometry with the edge labels.

```
pdegplot(model,"EdgeLabels","on")
```



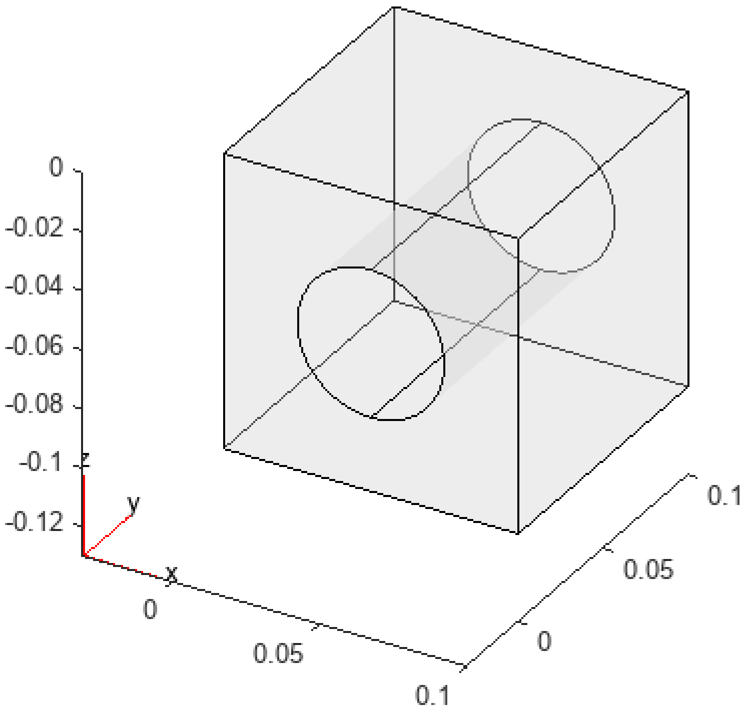
Import 3-D Geometry from STEP File

Create a geometry object from a STEP geometry file.

```
gm = importGeometry("BlockWithHole.step");
```

Plot the geometry.

```
pdegplot(gm,"FaceAlpha",0.3)
```

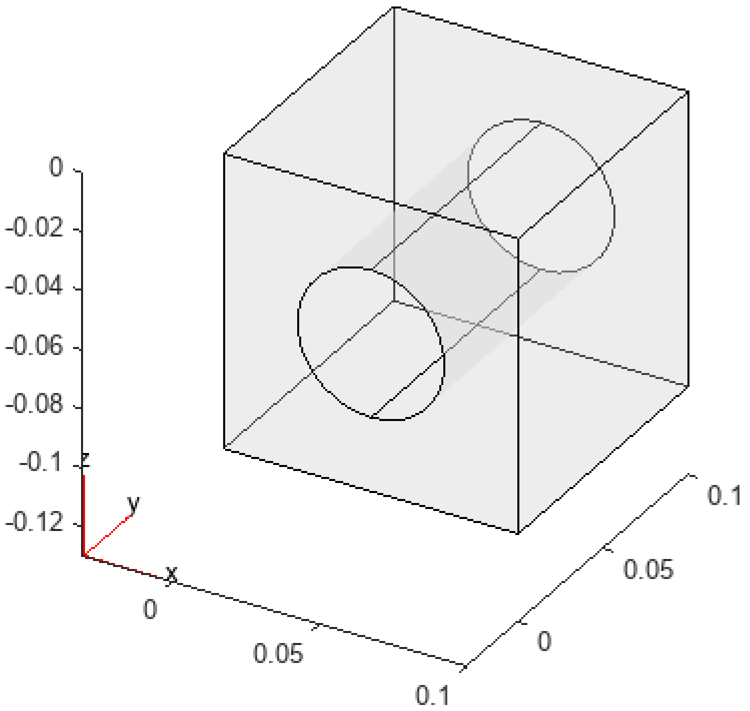


Now import the same geometry while specifying the relative sag. You can use this parameter to control the accuracy of the geometry import.

```
gm = importGeometry("BlockWithHole.step","MaxRelativeDeviation",10);
```

Plot the geometry.

```
figure
pdegplot(gm,"FaceAlpha",0.3)
```



Input Arguments

`model` — Model container
`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectromagneticModel` object

Model container, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectromagneticModel` object.

Example: `model = createpde(3)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","electrostatic")`

`geometryfile` — Path to STL or STEP file
string scalar | character vector

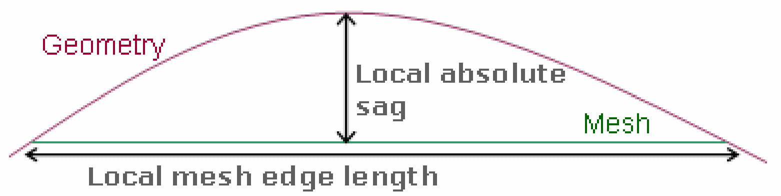
Path to STL or STEP file, specified as a string scalar or a character vector ending with the file extension `".stl"`, `".stp"`, or `".step"`. Use can also use the uppercase extensions `".STL"`, `".STP"` or `".STEP"`, or any combinations of uppercase and lowercase letters in these extensions.

Example: `"../geometries/Carburetor.stl"`

Data Types: `string` | `char`

d — Relative sag
1 (default) | number in the range [0.1, 10]

Relative sag for importing a STEP geometry, specified as a number in the range [0.1, 10]. A relative sag is the ratio between the local absolute sag and the local mesh edge length. The absolute sag is the maximal gap between the mesh and the geometry.



Example: `gm = importGeometry("AngleBlock.step","MaxRelativeDeviation",5)`

Data Types: `double`

Output Arguments

gm — Geometry
`DiscreteGeometry` object

Geometry, returned as a `DiscreteGeometry` object. See DiscreteGeometry for details.

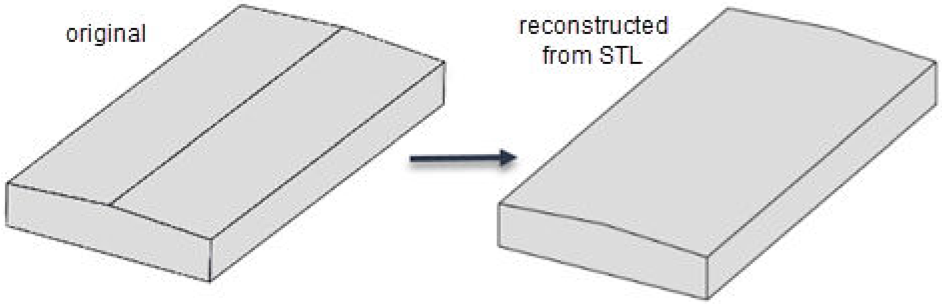
Limitations

- `importGeometry` does not allow you to import a multidomain 2-D or 3-D geometry where subdomains have any common points. If the subdomains of the geometry have common points, the toolbox still treats these subdomains as disconnected, without any common interface between them. Each subdomain has its own mesh.

  Because of this limitation, you cannot import nested 3-D geometries directly. As a workaround, you can import a mesh and then create a multidomain geometry from the mesh by using the `geometryFromMesh` function. See “Multidomain Geometry Reconstructed from Mesh” on page 2-83.

Tips

- The STL format approximates the boundary of a CAD geometry by using a collection of triangles, and the `importGeometry` function reconstructs the faces and edges from this data. Reconstruction from STL data is not precise and can result in a loss of edges and, therefore, the merging of adjacent faces. Typically, lost edges are the edges between two adjacent faces meeting at a small angle, or smooth edges bounding blend surfaces. Usually, the loss of such edges does not affect the analysis workflow.



- Because STL geometries are only approximations of the original CAD geometries, the areas and volumes of the STL and CAD geometries can differ.

Version History

Introduced in R2015a

Import from STEP files

You can now import a geometry from a STEP file. When importing from a STEP file, you can control the accuracy of the geometry import by specifying the relative sag value.

Import without creating a model

You can now create a geometry object from an STL geometry file without attaching the imported geometry to any model. The new syntax is `g = importGeometry('geometryfile.stl')`.

Import and mesh planar STL geometries

You can now import a planar STL geometry and convert it to a 2-D geometry by mapping it to the *xy*-plane.

Improved quality of STL import

When importing an STL geometry for a 3-D problem, the function can now recognize and reconstruct more geometric vertices, edges, and faces of the original CAD geometry in some instances. In these cases, the resulting geometry is a closer match to the original CAD geometry.

Detailed geometries can now contain more faces and edges than in previous releases. As a result, in rare instances, the new faces and edges can cause renumbering of the existing ones. If your code imports an STL geometry, visually check the geometry to ensure that you are assigning boundary and initial conditions to the intended regions.

See Also

DiscreteGeometry | `geometryFromMesh` | `pdegplot` | `PDEModel` | `StructuralModel` | `ThermalModel` | `ElectromagneticModel`

Topics

"STL File Import" on page 2-43
"STEP File Import" on page 2-59
"Multidomain Geometry Reconstructed from Mesh" on page 2-83
"Sphere in Cube" on page 2-75
"Geometry from Triangulated Mesh" on page 2-63

initmesh

Package: `pde`

Create initial 2-D mesh

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 `generateMesh`.

Syntax

```
[p,e,t] = initmesh(g)
[p,e,t] = initmesh(g,Name,Value)
```

Description

`[p,e,t] = initmesh(g)` generates a triangular mesh for a 2-D geometry. The function uses a Delaunay triangulation algorithm.

`[p,e,t] = initmesh(g,Name,Value)` generates a 2-D mesh using one or more `Name,Value` pair arguments.

Examples

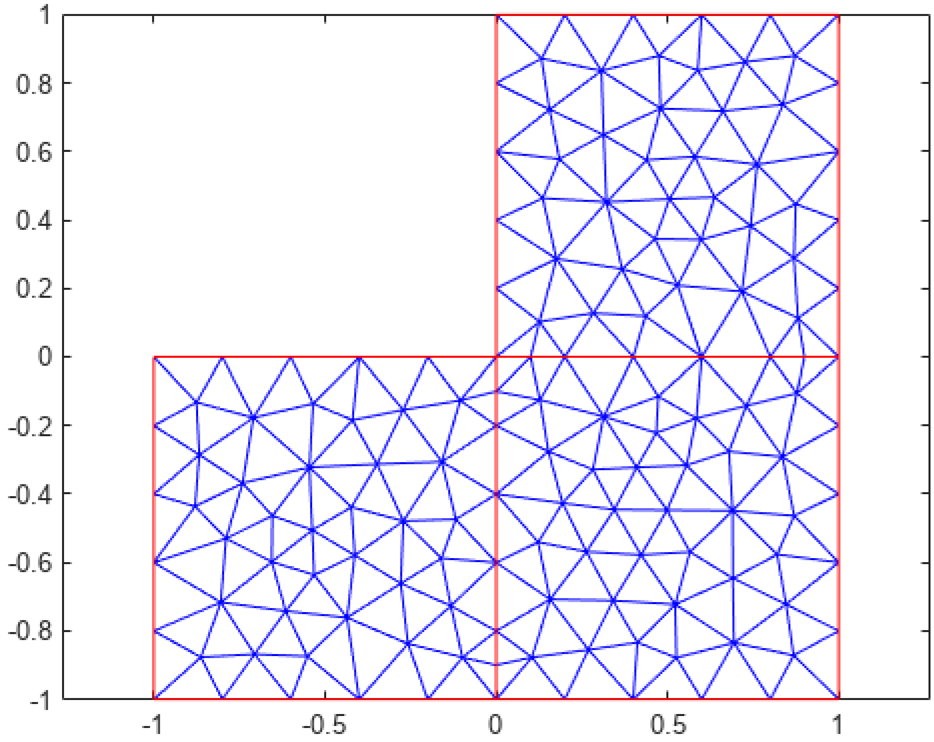
Initial Mesh for L-shaped Membrane

Generate a triangular mesh of the L-shaped membrane.

```
[p,e,t] = initmesh("lshapeg");
```

Plot the mesh.

```
pdemesh(p,e,t)
```



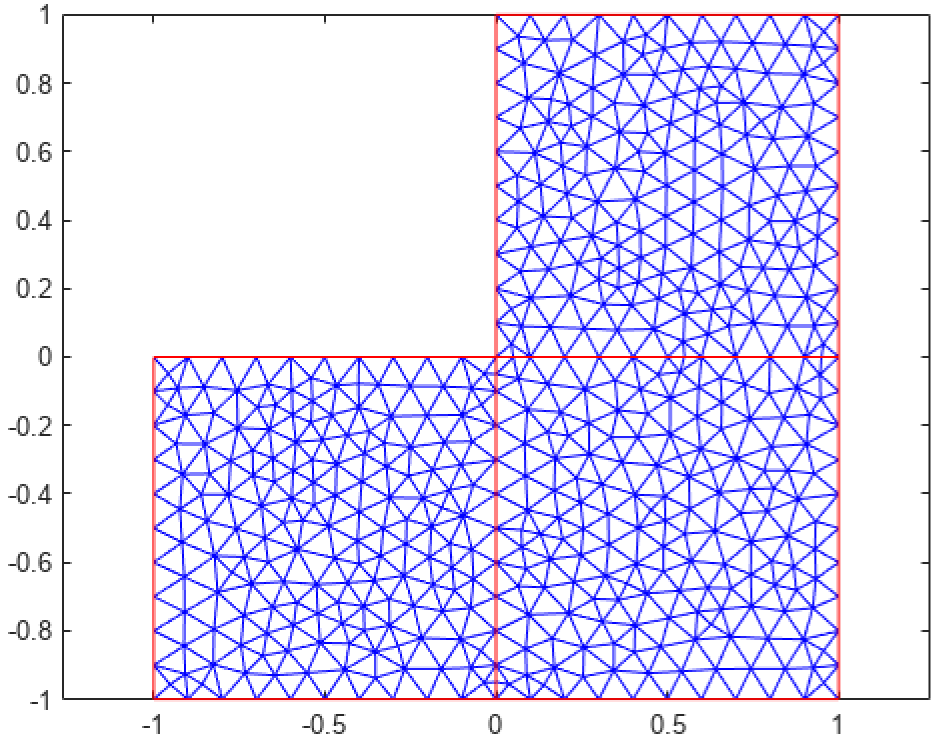
Maximum Mesh Edge Length

Generate a triangular mesh of the L-shaped membrane with the target maximum mesh edge length of 0.1.

```
[p,e,t] = initmesh("lshapeg","Hmax",0.1);
```

Plot the mesh.

```
pdemesh(p,e,t)
```



Input Arguments

g — Geometry description

decomposed geometry matrix | geometry function | handle to geometry function

Geometry description, specified as a decomposed geometry matrix, a geometry function, or a handle to the geometry function. For details about a decomposed geometry matrix, see `decsg`. For details about a geometry function, see "Parametrized Function for 2-D Geometry Creation" on page 2-22.

A geometry function must return the same result for the same input arguments in every function call. Thus, it must not contain functions and expressions designed to return a variety of results, such as random number generators.

Data Types: `double` | `char` | `string` | `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[p,e,t] = initmesh("lshapeg","Hmax",0.1);`

`Hmax` — Target maximum mesh edge length

positive real number

Target maximum mesh edge length, specified as the comma-separated pair consisting of `"Hmax"` and a positive real number.

`Hmax` is an approximate upper bound on the mesh edge lengths. `initmesh` estimates the default value of `Hmax` from overall dimensions of the geometry.

Small `Hmax` values let you create finer meshes, but mesh generation can take a very long time in this case. You can interrupt mesh generation by using **Ctrl+C**. Note that `initmesh` can take additional time to respond to the interrupt.

Example: `[p,e,t] = initmesh(g,"Hmax",0.25);`

Data Types: `double`

`Hgrad` — Mesh growth rate

`1.3` (default) | number strictly greater than 1 and less than 2

Mesh growth rate, specified as the comma-separated pair consisting of `"Hgrad"` and a number strictly greater than 1 and less than 2.

Example: `[p,e,t] = initmesh(g,"Hgrad",1.5);`

Data Types: `double`

`Box` — Toggle to preserve bounding box

`"off"` (default) | `"on"`

Toggle to preserve bounding box, specified as the comma-separated pair consisting of `"Box"` and `"on"` or `"off"`. By turning on `"Box"` you can get a good idea of how the mesh generation algorithm works within the bounding box.

Example: `[p,e,t] = initmesh(g,"Box","on");`

Data Types: `char` | `string`

`Init` — Toggle to use edge triangulation

`"off"` (default) | `"on"`

Toggle to use edge triangulation, specified as the comma-separated pair consisting of `"Init"` and `"on"` or `"off"`. By turning on `Init` you can see the initial triangulation of the boundaries. For example, use these commands to determine the subdomain number `n` of the point `xy`.

```
[p,e,t] = initmesh(g,"Hmax",Inf,"Init","on");
[uxy,tn,a2,a3] = tri2grid(p,t,zeros(size(p,2)),x,y);
n = t(4,tn);
```

If the point is outside the geometry, `tn` is `NaN`, and the command `n = t(4,tn)` results in a failure.

Data Types: `char` | `string`

`Jiggle` — Toggle to call `jigglemesh` after creating the mesh

`"mean"` (default) | `"minimum"` | `"on"` | `"off"`

Toggle to call `jigglemesh` after creating the mesh, specified as the comma-separated pair consisting of `"Jiggle"` and `"mean"`, `"minimum"``"on"`, or `"off"`.

- `"mean"` — call `jigglemesh` with the argument `"Opt"` set to `"mean"`.
- `"minimum"` — call `jigglemesh` with the argument `"Opt"` set to `"minimum"`.
- `"on"` — call `jigglemesh` with the argument `"Opt"` set to `"off"`.
- `"off"` — do not call `jigglemesh`.

Example: `[p,e,t] = initmesh(g,"Jiggle","minimum");`

Data Types: `char` | `string`

`JiggleIter` — Maximum number of iterations for `jigglemesh`
`10` (default) | positive integer

Maximum number of iterations for `jigglemesh`, specified as the comma-separated pair consisting of `"JiggleIter"` and a positive integer.

Example: `[p,e,t] = initmesh(g,"Jiggle","on","JiggleIter",50);`

Data Types: `double`

`MesherVersion` — Algorithm for generating initial mesh
`"preR2013a"` (default) | `"R2013a"`

Algorithm for generating initial mesh, specified as the comma-separated pair consisting of `"MesherVersion"` and either `"R2013a"` or `"preR2013a"`. The `"R2013a"` algorithm runs faster, and can triangulate more geometries than the `"preR2013a"` algorithm. Both algorithms use Delaunay triangulation.

Data Types: `char` | `string`

Output Arguments

p — Mesh points
2-by-`Np` matrix

Mesh points, returned as a 2-by-`Np` matrix. `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)` and the *y*-coordinate of point `k` in `p(2,k)`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

e — Mesh edges
7-by-`Ne` matrix

Mesh edges, returned as a 7-by-`Ne` matrix, where `Ne` is the number of boundary 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 as [p,e,t] Triples” on page 2-172.

t — Mesh elements
4-by-`Nt` matrix

Mesh elements, returned as a 4-by-`Nt` matrix. `Nt` is the number of triangles in the mesh.

The `t(i,k)`, with `i` ranging from 1 through `end - 1`, contain indices to the corner points of element `k`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172. The last row, `t(end,k)`, contains the subdomain number of the element.

Version History

Introduced before R2006a

Performance and robustness enhancements in meshing algorithm

Performance change in R2013a

`initmesh` provides an enhancement option for increased meshing speed and robustness. Choose the enhanced algorithm by setting the `MesherVersion` name-value pair to `'R2013a'`. The default `MesherVersion` value of `'preR2013a'` gives the same mesh as previous toolbox versions.

The enhancement is available in`pdeModeler` from the **Mesh > Parameters > Mesher version** menu.

References

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

See Also

`decsg` | `jigglemesh` | `refinemesh` | `adaptmesh`

Topics

"Mesh Data as [p,e,t] Triples" on page 2-172

internalHeatSource

Package: `pde`

Specify internal heat source for a thermal model

Syntax

```
internalHeatSource(thermalmodel,heatSourceValue)
internalHeatSource(thermalmodel,heatSourceValue,RegionType,RegionID)
internalHeatSource( ___ ,"Label",labeltext)
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`.

`internalHeatSource( ___ ,"Label",labeltext)` adds a label for the internal heat source to be used by the `linearizeInput` function. This function lets you pass internal heat sources to the `linearize` function that extracts sparse linear models for use with Control System Toolbox™.

`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
         Label: []
```

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
         Label: []
```

Specify Nonconstant Internal Heat Source

Use a function handle to specify an internal heat source that depends on coordinates.

Create a thermal model for transient analysis and include the geometry. The geometry is a rod with a circular cross section. The 2-D model is a rectangular strip whose *y*-dimension extends from the axis of symmetry to the outer surface, and whose *x*-dimension extends over the actual length of the rod.

```
thermalmodel = createpde("thermal","transient");
g = decsg([3 4 -1.5 1.5 1.5 -1.5 0 0 .2 .2]');
geometryFromEdges(thermalmodel,g);
```

The heat is generated within the rod due to the radioactive decay. Therefore, the entire geometry is an internal nonlinear heat source and can be represented by a function of the *y*-coordinate, for example, $q = 2000y$.

```
q = @(location,state)2000*location.y;
```

Specify the internal heat source for the transient model.

```
internalHeatSource(thermalmodel,q)
```

```
ans = 
  HeatSourceAssignment with properties:

    RegionType: 'face'
      RegionID: 1
    HeatSource: @(location,state)2000*location.y
         Label: []
```

Specify Time-Dependent Internal Heat Source

Use a function handle to specify an internal heat source that depends on time.

Create a thermal model for transient analysis and include the geometry. The geometry is a rectangular strip.

```
thermalmodel = createpde("thermal","transient");
g = decsg([3 4 -1.5 1.5 1.5 -1.5 0 0 .2 .2]');
geometryFromEdges(thermalmodel,g);
```

Specify the thermal properties of the rod.

```
thermalProperties(thermalmodel,"ThermalConductivity",40,...
                               "MassDensity",7800,...
                               "SpecificHeat",500);
```

Specify the boundary conditions and initial temperature.

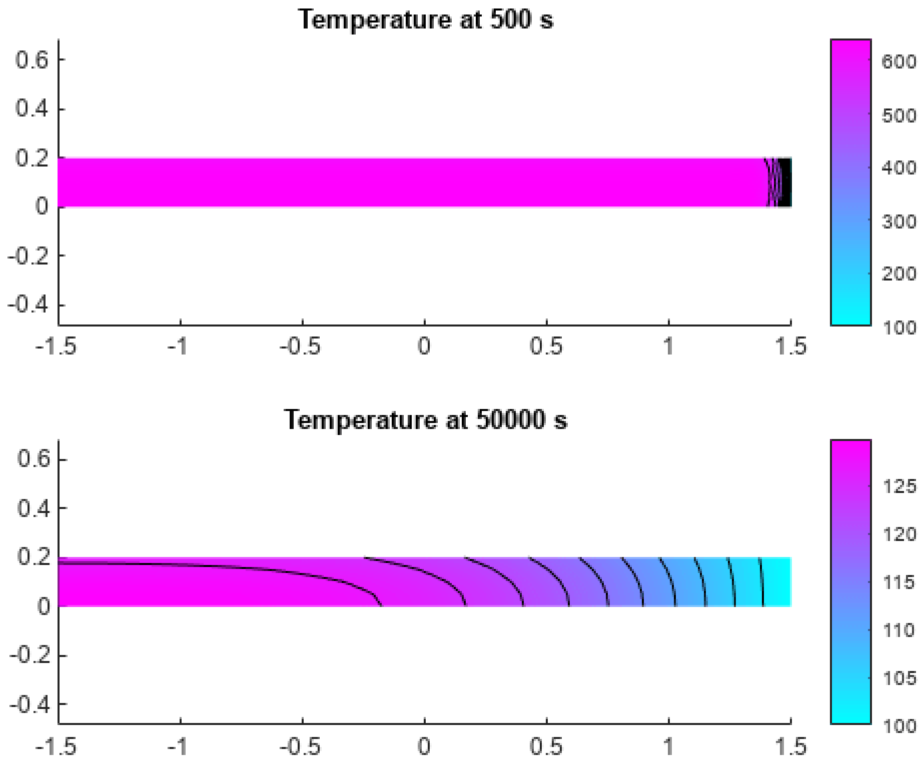
```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
thermalBC(thermalmodel,"Edge",3,...
                       "ConvectionCoefficient",50,...
                       "AmbientTemperature",100);
thermalIC(thermalmodel,0);
```

Specify that the entire geometry generates heat at the rate 20000*t* during the first 500 seconds, and then the heat source turns off. For details, see Time-Dependent Heat Source Function on page 5-581.

```
internalHeatSource(thermalmodel,@heatSource);
```

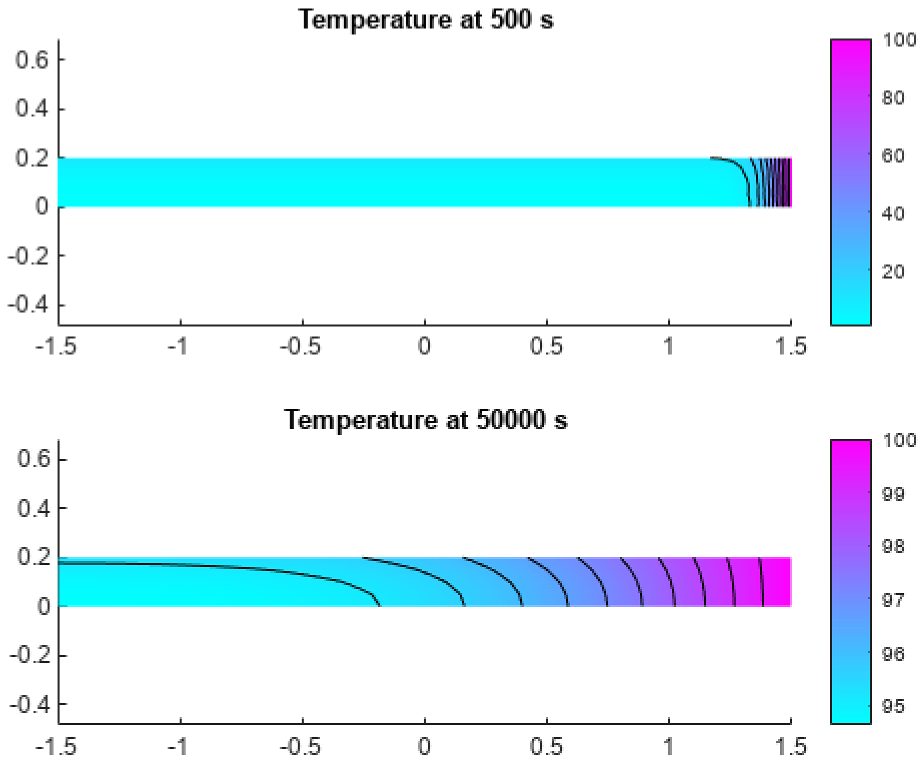
Generate the mesh, solve the model using the solution times from 0 to 50000 seconds, and plot the results.

```
generateMesh(thermalmodel);

tfinal = 50000;
tlist = 0:100:tfinal;
result = solve(thermalmodel,tlist);
T = result.Temperature;

figure
subplot(2,1,1)
pdeplot(thermalmodel,"XYData",T(:,6),"Contour","on")
axis equal
title(sprintf("Temperature at %g s",tlist(6)))
subplot(2,1,2)
pdeplot(thermalmodel,"XYData",T(:,end),"Contour","on")
axis equal
title(sprintf("Temperature at %g s",tfinal))
```



Always ensure that your function returns a matrix of `NaN` of the correct size when `state.time` is `NaN`. The solver properly recognizes a time-dependent problem by passing `NaN` state values and looking for returned `NaN` values. Without this condition, the solver might fail or return incorrect results.

```
internalHeatSource(thermalmodel,@heatSourceInvalid);

result = solve(thermalmodel,tlist);
T = result.Temperature;

figure
subplot(2,1,1)
pdeplot(thermalmodel,"XYData",T(:,6),"Contour","on")
axis equal
title(sprintf("Temperature at %g s",tlist(6)))
subplot(2,1,2)
pdeplot(thermalmodel,"XYData",T(:,end),"Contour","on")
axis equal
title(sprintf("Temperature at %g s",tfinal))
```



Time-Dependent Heat Source Function

```
function Q = heatSource(location,state)
  Q = zeros(1,numel(location.x));
if(isnan(state.time))
% Returning a NaN when time=NaN tells
% the solver that the heat source is a function of time.
  Q(1,:) = NaN;
  return
end
if state.time < 500
  Q(1,:) = 20000*state.time;
end
end

function Q = heatSourceInvalid(location,state) % No checks for NaN
  Q = zeros(1,numel(location.x));
if state.time < 500
  Q(1,:) = 20000*state.time;
end
end
```

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","steadystate")`

`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` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

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. Use a function handle to specify the internal heat source that depends on space, time, or temperature. For details, see “More About” on page 5-582.

Example: `internalHeatSource(thermalmodel,25)`

Data Types: `double` | `function_handle`

`labeltext` — Label for internal heat source
character vector | string

Label for the internal heat source, specified as a character vector or a string.

Data Types: `char` | `string`

Output Arguments

`heatSource` — Handle to heat source
`HeatSourceAssignment` object

Handle to heat source, returned as a `HeatSourceAssignment` object. See HeatSourceAssignment Properties.

`heatSourceValue` associates the heat source value with the geometric region.

More About

Specifying Nonconstant Parameters of a Thermal Model

Use a function handle to specify these thermal parameters when they depend on space, temperature, and time:

- Thermal conductivity of the material
- Mass density of the material
- Specific heat of the material
- Internal heat source
- Temperature on the boundary
- Heat flux through the boundary
- Convection coefficient on the boundary
- Radiation emissivity coefficient on the boundary
- Initial temperature (can depend on space only)

For example, use function handles to specify the thermal conductivity, internal heat source, convection coefficient, and initial temperature for this model.

```
thermalProperties(model,"ThermalConductivity", ...
                        @myfunConductivity)
internalHeatSource(model,"Face",2,@myfunHeatSource)
thermalBC(model,"Edge",[3,4], ...
                 "ConvectionCoefficient",@myfunBC, ...
                 "AmbientTemperature",27)
thermalIC(model,@myfunIC)
```

For all parameters, except the initial temperature, the function must be of the form:

```
function thermalVal = myfun(location,state)
```

For the initial temperature the function must be of the form:

```
function thermalVal = myfun(location)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments (or one argument if the function specifies the initial temperature).

- `location` — A structure containing these fields:

  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes these data in the `location` structure:

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, *r*-component of the normal vector at the evaluation point or points
- `state` — A structure containing these fields for transient or nonlinear problems:

  - `state.u` — Temperatures at the corresponding points of the location structure
  - `state.ux` — Estimates of the *x*-component of temperature gradients at the corresponding points of the location structure
  - `state.uy` — Estimates of the *y*-component of temperature gradients at the corresponding points of the location structure
  - `state.uz` — For a 3-D or an axisymmetric geometry, estimates of the *z*-component of temperature gradients at the corresponding points of the location structure
  - `state.ur` — For an axisymmetric geometry, estimates of the *r*-component of temperature gradients at the corresponding points of the location structure
  - `state.time` — Time at evaluation points

Thermal material properties (thermal conductivity, mass density, and specific heat) and internal heat source get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID
- `state.u`, `state.ux`, `state.uy`, `state.uz`, `state.r`, `state.time`

Boundary conditions (temperature on the boundary, heat flux, convection coefficient, and radiation emissivity coefficient) get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`
- `state.u`, `state.time`

Initial temperature gets the following data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

For all thermal parameters, except for thermal conductivity, your function must return a row vector `thermalVal` with the number of columns equal to the number of evaluation points, for example, `M = length(location.y)`.

For thermal conductivity, your function must return a matrix `thermalVal` with number of rows equal to 1, `Ndim`, `Ndim*(Ndim+1)/2`, or `Ndim*Ndim`, where `Ndim` is 2 for 2-D problems and 3 for 3-D problems. The number of columns must equal the number of evaluation points, for example, `M = length(location.y)`. For details about dimensions of the matrix, see “c Coefficient for specifyCoefficients” on page 2-92.

If properties depend on the time or temperature, 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 time dependent by passing `NaN` state values and looking for returned `NaN` values.

Additional Arguments in Functions for Nonconstant Thermal Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
thermalVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
```

```
thermalBC(model,"Edge",3,"Temperature",thermalVal)

thermalVal = @(location) myfunWithAdditionalArgs(location,arg1,arg2...)
thermalIC(model,thermalVal)
```

Version History

Introduced in R2017a

Label to extract sparse linear models for use with Control System Toolbox

Now you can add a label for the internal heat source to be used by the `linearizeInput` function. This function lets you pass internal heat sources to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

See Also

`thermalBC` | `thermalProperties` | HeatSourceAssignment Properties

interpolateAcceleration

Package: `pde`

Interpolate acceleration at arbitrary spatial locations for all time or frequency steps for dynamic structural model

Syntax

```
intrpAccel = interpolateAcceleration(structuralresults,xq,yq)
intrpAccel = interpolateAcceleration(structuralresults,xq,yq,zq)
intrpAccel = interpolateAcceleration(structuralresults,querypoints)
```

Description

`intrpAccel = interpolateAcceleration(structuralresults,xq,yq)` returns the interpolated acceleration values at the 2-D points specified in `xq` and `yq` for all time or frequency steps.

`intrpAccel = interpolateAcceleration(structuralresults,xq,yq,zq)` uses the 3-D points specified in `xq`, `yq`, and `zq`.

`intrpAccel = interpolateAcceleration(structuralresults,querypoints)` uses the points specified in `querypoints`.

Examples

Interpolate Acceleration for 3-D Structural Dynamic Problem

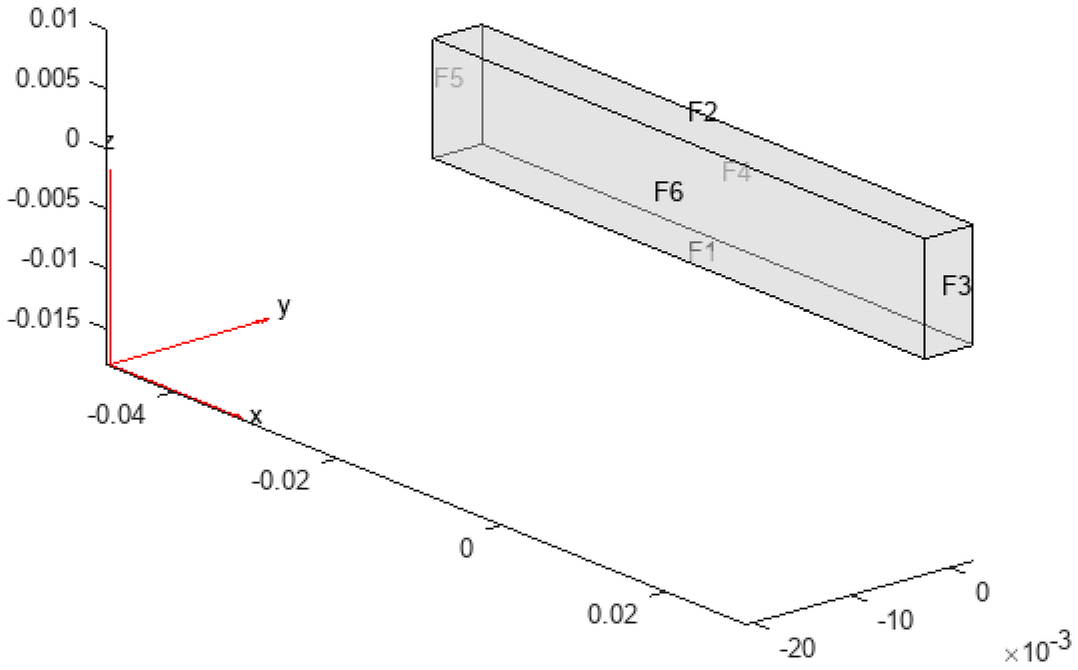
Interpolate acceleration at the geometric center of a beam under a harmonic excitation

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the `y`-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3,...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0], ...
                             "Velocity",[0;0;0]);
```

Solve the model.

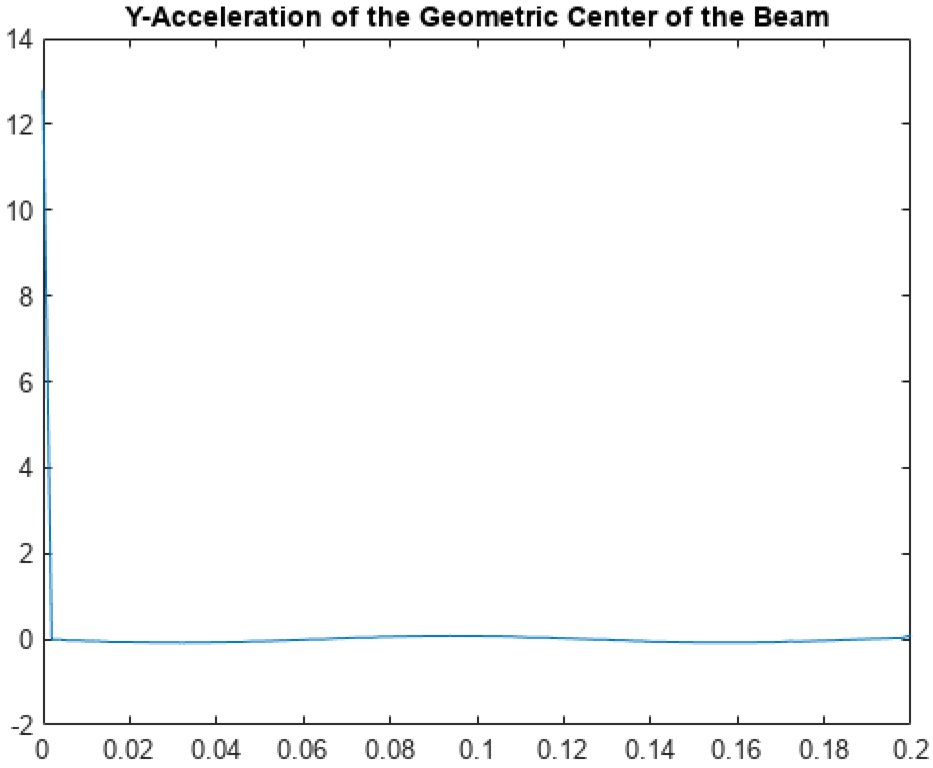
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Interpolate acceleration at the geometric center of the beam.

```
coordsMidSpan = [0;0;0.005];
intrpAccel = interpolateAcceleration(structuralresults,coordsMidSpan);
```

Plot the y-component of acceleration of the geometric center of the beam.

```
figure
plot(structuralresults.SolutionTimes,intrpAccel.ay)
title("Y-Acceleration of the Geometric Center of the Beam")
```



Input Arguments

`structuralresults` — Solution of dynamic structural analysis problem

`TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the dynamic structural analysis problem, specified as a `TransientStructuralResults` or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel,tlist)`

`xq` — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateAcceleration` evaluates accelerations at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateAcceleration` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The function returns accelerations as an `FEStruct` object with the properties containing vectors of the same size as these column vectors. To ensure that the dimensions of the returned solution are consistent with the dimensions of the original query points, use the `reshape` function. For example, use `intrpAccel = reshape(intrpAccel.ux,size(xq))`.

Data Types: `double`

`yq` — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateAcceleration` evaluates accelerations at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries. Internally, `interpolateAcceleration` converts the 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. `interpolateAcceleration` evaluates accelerations at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries. Internally, `interpolateAcceleration` converts the 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. `interpolateAcceleration` evaluates accelerations 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`

Output Arguments

`intrpAccel` — Accelerations at query points
`FEStruct` object

Accelerations at the query points, returned as an `FEStruct` object with the properties representing spatial components of acceleration at the query points. For query points that are outside the geometry, `intrpAccel` returns `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2018a

See Also

`StructuralModel` | `TransientStructuralResults` | `interpolateDisplacement` | `interpolateVelocity` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateStress` | `evaluateStrain` | `evaluateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

interpolateDisplacement

Package: `pde`

Interpolate displacement at arbitrary spatial locations

Syntax

```
intrpDisp = interpolateDisplacement(structuralresults,xq,yq)
intrpDisp = interpolateDisplacement(structuralresults,xq,yq,zq)
intrpDisp = interpolateDisplacement(structuralresults,querypoints)
```

Description

`intrpDisp = interpolateDisplacement(structuralresults,xq,yq)` returns the interpolated displacement values at the 2-D points specified in `xq` and `yq`. For transient and frequency response structural models, `interpolateDisplacement` returns the interpolated displacement values for all time- or frequency-steps, respectively.

`intrpDisp = interpolateDisplacement(structuralresults,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`intrpDisp = interpolateDisplacement(structuralresults,querypoints)` uses points specified in `querypoints`.

Examples

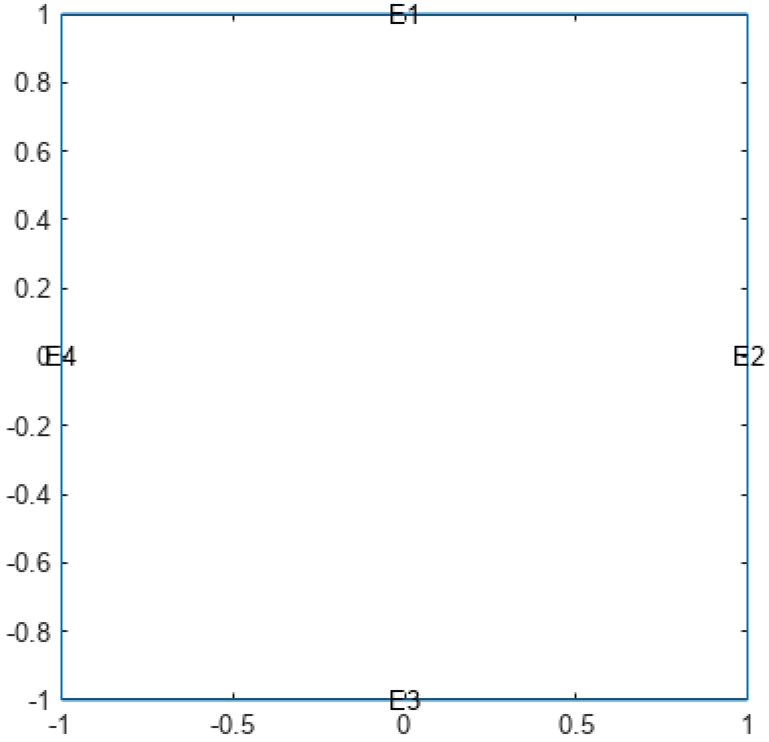
Interpolate Displacement for Plane-Strain Problem

Create a structural analysis model for a plane-strain problem.

```
structuralmodel = createpde("structural","static-planestrain");
```

Include the square geometry in the model. Plot the geometry.

```
geometryFromEdges(structuralmodel,@squareg);
pdegplot(structuralmodel,"EdgeLabels","on")
axis equal
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"PoissonsRatio",0.3, ...
                                     "YoungsModulus",210E3);
```

Specify the *x*-component of the enforced displacement for edge 1.

```
structuralBC(structuralmodel,"XDisplacement",0.001,"Edge",1);
```

Specify that edge 3 is a fixed boundary.

```
structuralBC(structuralmodel,"Constraint","fixed","Edge",3);
```

Generate a mesh and solve the problem.

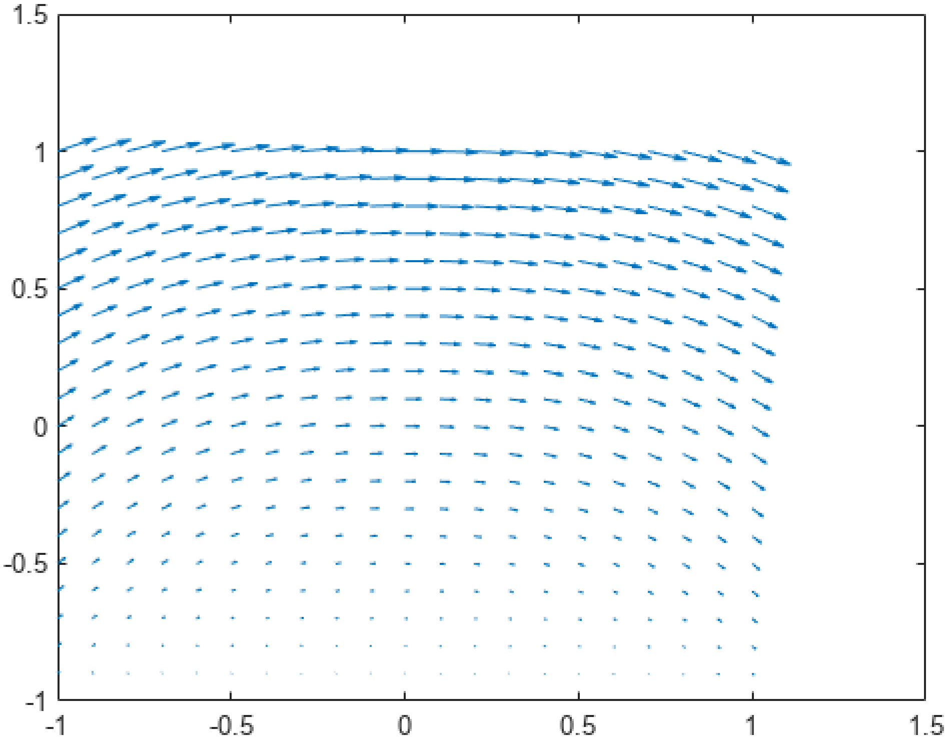
```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel);
```

Create a grid and interpolate the *x*- and *y*-components of the displacement to the grid.

```
v = linspace(-1,1,21);
[X,Y] = meshgrid(v);
intrpDisp = interpolateDisplacement(structuralresults,X,Y);
```

Reshape the displacement components to the shape of the grid. Plot the displacement.

```
ux = reshape(intrpDisp.ux,size(X));
uy = reshape(intrpDisp.uy,size(Y));
quiver(X,Y,ux,uy)
```



Interpolate Displacement for 3-D Static Structural Analysis Problem

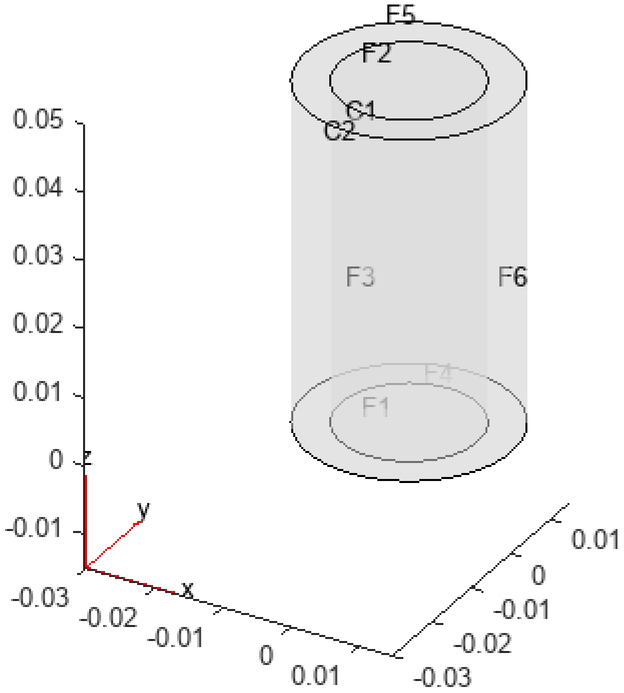
Solve a static structural model representing a bimetallic cable under tension, and interpolate the displacement on a cross-section of the cable.

Create a static structural model for solving a solid (3-D) problem.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01,0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                             "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                             "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

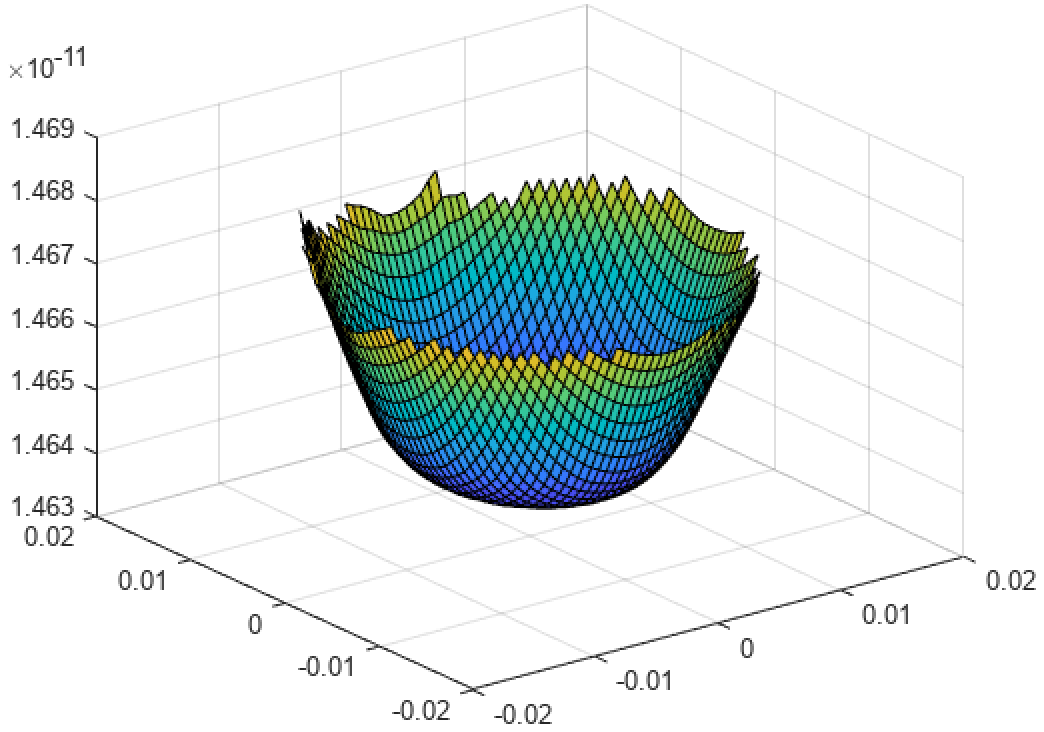
```
            Mesh: [1x1 FEMesh]
```

Define coordinates of a midspan cross-section of the cable.

```
[X,Y] = meshgrid(linspace(-0.015,0.015,50));
Z = ones(size(X))*0.025;
```

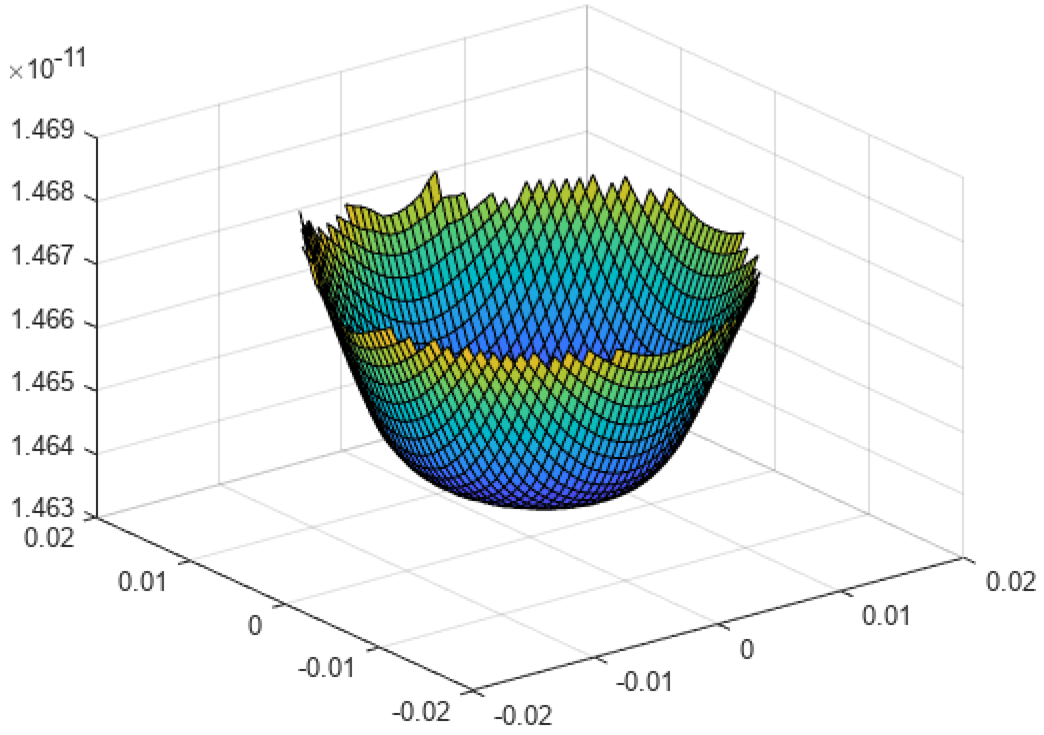
Interpolate the displacement and plot the result.

```
intrpDisp = interpolateDisplacement(structuralresults,X,Y,Z);
surf(X,Y,reshape(intrpDisp.uz,size(X)))
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:),Z(:)]';
intrpDisp = interpolateDisplacement(structuralresults,querypoints);
surf(X,Y,reshape(intrpDisp.uz,size(X)))
```



Interpolate Displacement for Transient Structural Analysis Problem

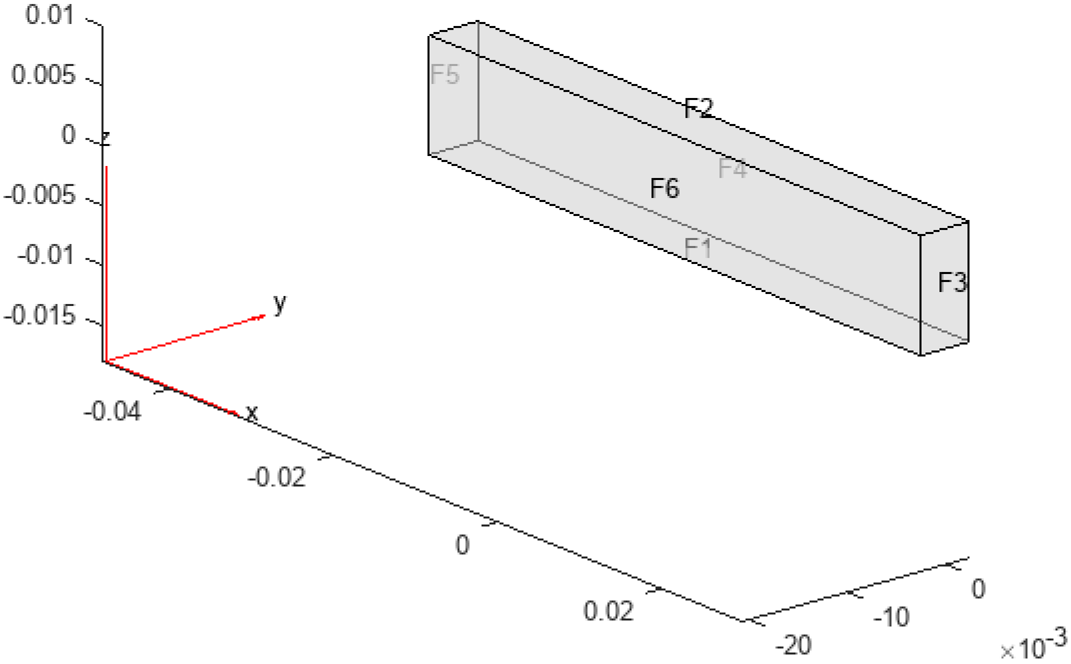
Interpolate the displacement at the geometric center of a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the `y`-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

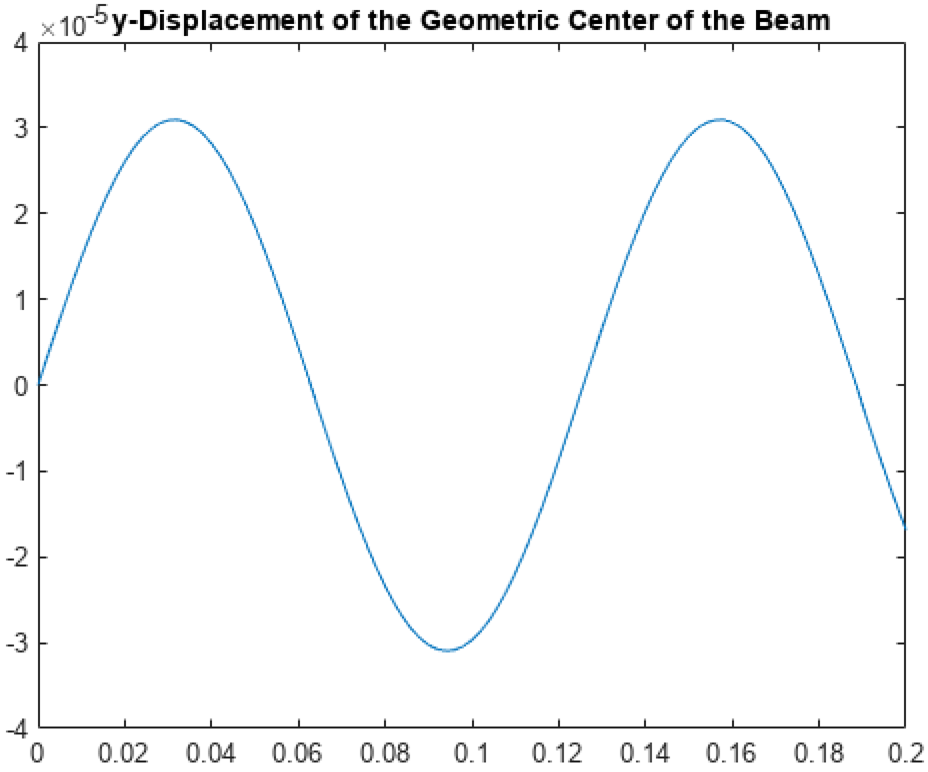
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Interpolate the displacement at the geometric center of the beam.

```
coordsMidSpan = [0;0;0.005];
intrpDisp = interpolateDisplacement(structuralresults,coordsMidSpan);
```

Plot the `y`-component of displacement of the geometric center of the beam.

```
figure
plot(structuralresults.SolutionTimes,intrpDisp.uy)
title("y-Displacement of the Geometric Center of the Beam")
```



Input Arguments

`structuralresults` — Solution of structural analysis problem

`StaticStructuralResults` object | `TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the structural analysis problem, specified as a `StaticStructuralResults`, `TransientStructuralResults`, or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function. For `TransientStructuralResults` and `FrequencyStructuralResults` objects, `interpolateDisplacement` returns the interpolated displacement values for all time- and frequency-steps, respectively.

Example: `structuralresults = solve(structuralmodel)`

xq — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateDisplacement` evaluates the displacements at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateDisplacement` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The function returns displacements as an `FEStruct` object with the properties containing vectors of the same size as these column vectors. To ensure that the dimensions of the returned solution are consistent with the dimensions of the original query points, use the `reshape` function. For example, use `intrpDisp = reshape(intrpDisp.ux,size(xq))`.

Data Types: `double`

yq — *y*-coordinate query points

real array

y-coordinate query points, specified as a real array. `interpolateDisplacement` evaluates the displacements at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries. Internally, `interpolateDisplacement` 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. `interpolateDisplacement` evaluates the displacements at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries. Internally, `interpolateDisplacement` 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. `interpolateDisplacement` evaluates the displacements 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`

Output Arguments

`intrpDisp` — Displacements at query points

`FEStruct` object

Displacements at the query points, returned as an `FEStruct` object with the properties representing spatial components of displacement at the query points. For query points that are outside the geometry, `intrpDisp` returns `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2017b

Support for frequency response structural problems

For frequency response structural models, `interpolateDisplacement` interpolates displacement for all frequency-steps.

Support for transient structural problems

For transient structural models, `interpolateDisplacement` interpolates displacement for all time-steps.

See Also

`StructuralModel` | `StaticStructuralResults` | `TransientStructuralResults` | `interpolateVelocity` | `interpolateAcceleration` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateStress` | `evaluateStrain` | `evaluateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

interpolateElectricField

Package: `pde`

Interpolate electric field in electrostatic or DC conduction result at arbitrary spatial locations

Syntax

```
Eintrp = interpolateElectricField(results,xq,yq)
Eintrp = interpolateElectricField(results,xq,yq,zq)
Eintrp = interpolateElectricField(results,querypoints)
```

Description

`Eintrp = interpolateElectricField(results,xq,yq)` returns the interpolated electric field values at the 2-D points specified in `xq` and `yq`.

`Eintrp = interpolateElectricField(results,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`Eintrp = interpolateElectricField(results,querypoints)` returns the interpolated electric field values at the points specified in `querypoints`.

Examples

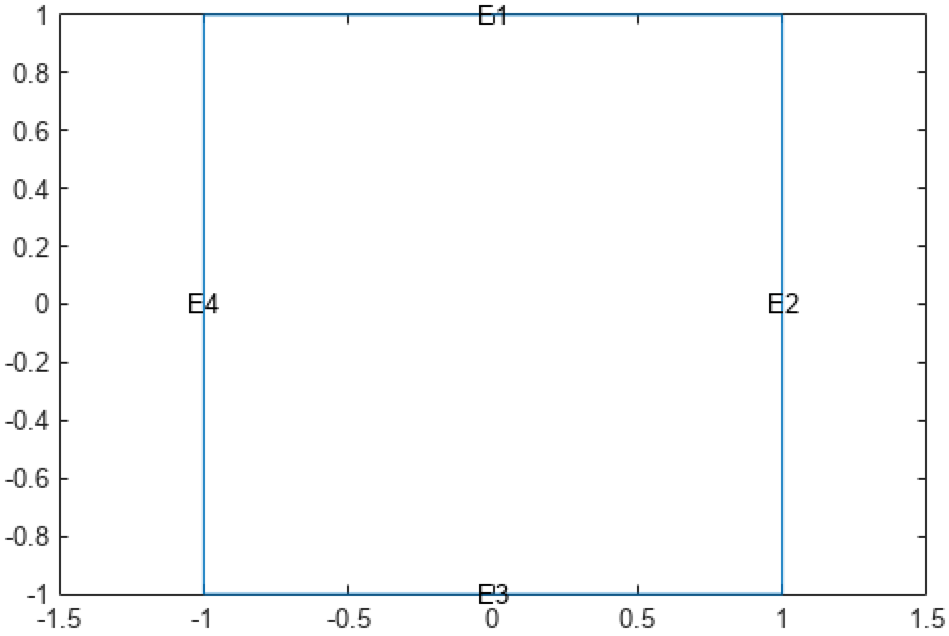
Interpolate Electric Field in 2-D Electrostatic Analysis

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Create a square geometry and include it in the model. Plot the geometry with the edge labels.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1,'R1',('R1')');
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-1.5 1.5])
axis equal
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Apply the voltage boundary conditions on the edges of the square.

```
electromagneticBC(emagmodel,"Voltage",0,"Edge",[1 3]);
electromagneticBC(emagmodel,"Voltage",1000,"Edge",[2 4]);
```

Specify the charge density for the entire geometry.

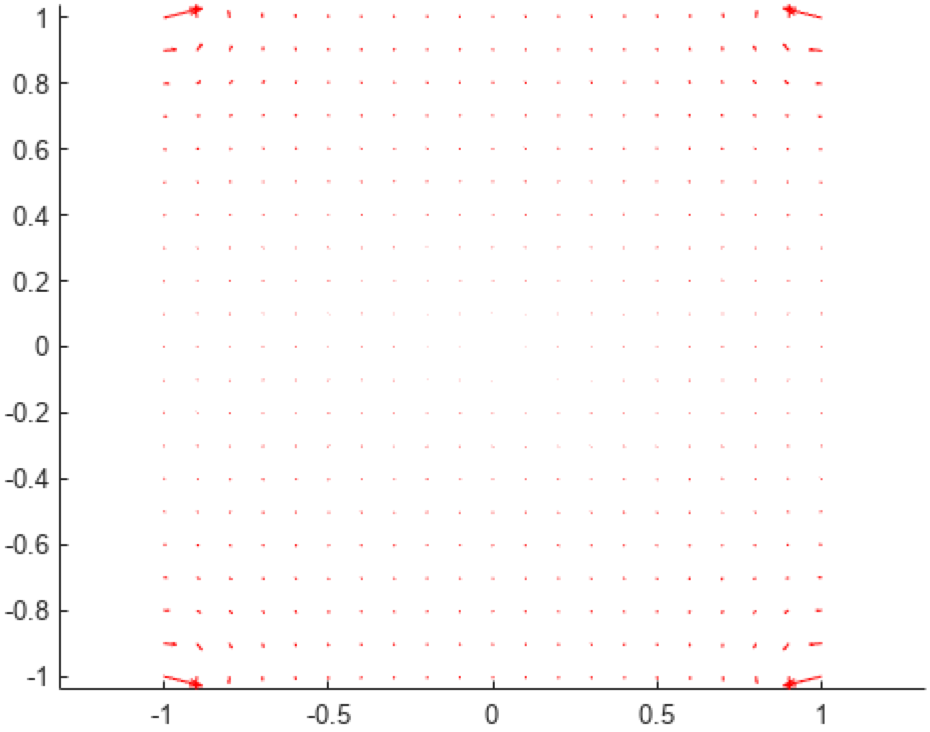
```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model and plot the electric field.

```
R = solve(emagmodel);
pdeplot(emagmodel,"FlowData",[R.ElectricField.Ex ...
                              R.ElectricField.Ey])
axis equal
```



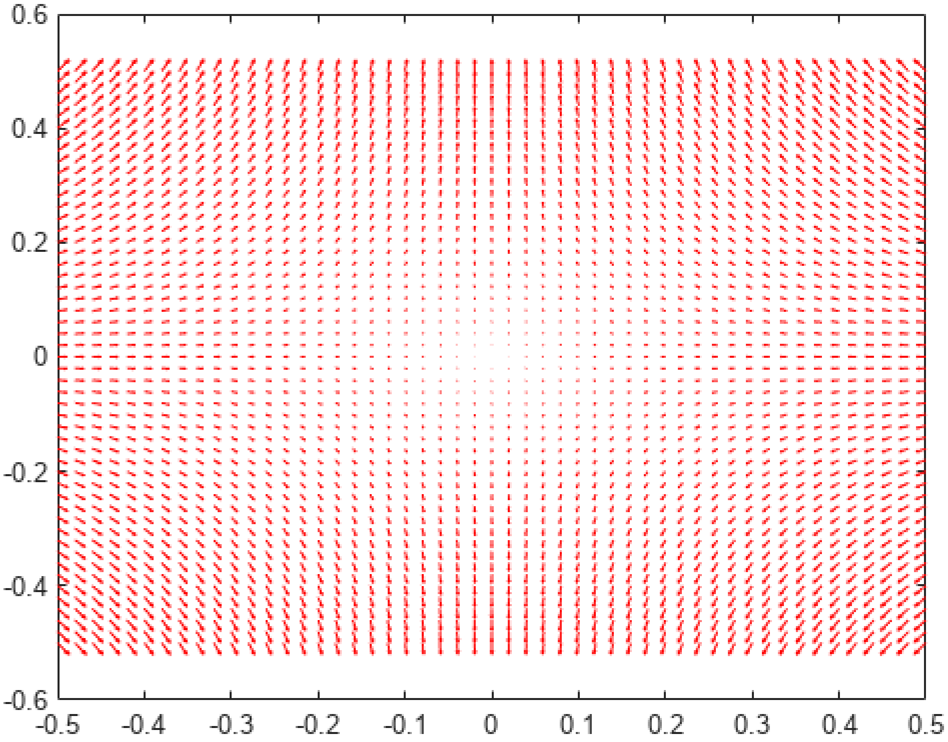
Interpolate the resulting electric field to a grid covering the central portion of the geometry, for `x` and `y` from `-0.5` to `0.5`.

```
v = linspace(-0.5,0.5,51);
[X,Y] = meshgrid(v);

Eintrp = interpolateElectricField(R,X,Y)
```

```
Eintrp = 
  FEStruct with properties:

    Ex: [2601x1 double]
    Ey: [2601x1 double]
```

Reshape `Eintrp.Ex` and `Eintrp.Ey` and plot the resulting electric field.

```
EintrpX = reshape(Eintrp.Ex,size(X));
EintrpY = reshape(Eintrp.Ey,size(Y));

figure
quiver(X,Y,EintrpX,EintrpY,"Color","red")
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Eintrp = interpolateElectricField(R,querypoints);
```

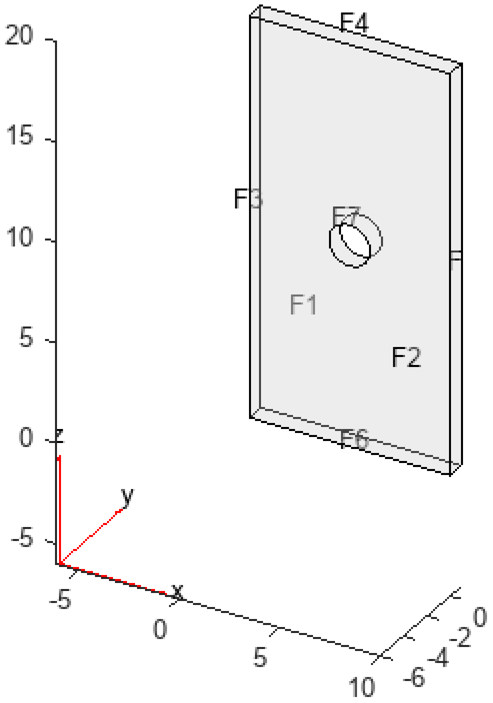
Interpolate Electric Field in 3-D Electrostatic Analysis

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Apply the voltage boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
electromagneticBC(emagmodel,"Voltage",1000,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

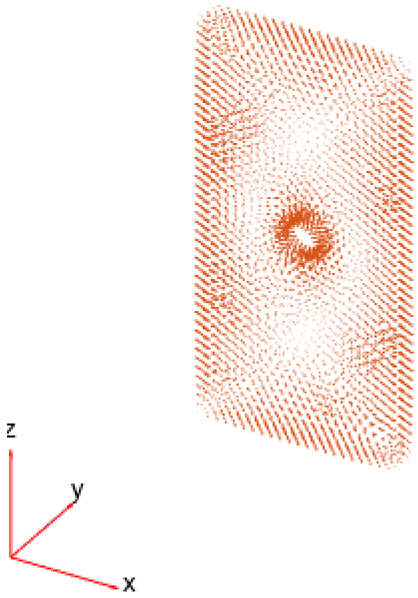
```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:
```

```
        ElectricPotential: [4359x1 double]
            ElectricField: [1x1 FEStruct]
      ElectricFluxDensity: [1x1 FEStruct]
                     Mesh: [1x1 FEMesh]
```

Plot the electric field.

```
pdeplot3D(emagmodel,"FlowData",[R.ElectricField.Ex ...
                                R.ElectricField.Ey ...
                                R.ElectricField.Ez])
```



Interpolate the resulting electric field to a grid covering the central portion of the geometry, for `x`, `y`, and `z`.

```
x = linspace(3,7,7);
y = linspace(0,1,7);
z = linspace(8,12,7);
[X,Y,Z] = meshgrid(x,y,z);

Eintrp = interpolateElectricField(R,X,Y,Z)
```

```
Eintrp = 
  FEStruct with properties:

    Ex: [343x1 double]
    Ey: [343x1 double]
```

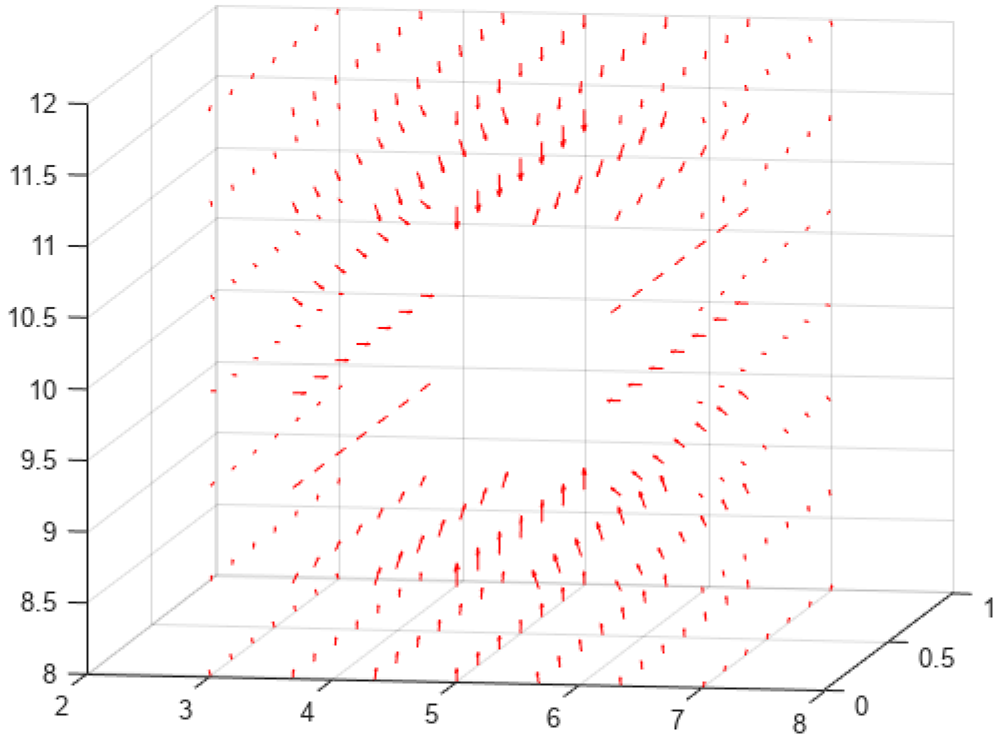
```
    Ez: [343x1 double]
```

Reshape `Eintrp.Ex`, `Eintrp.Ey`, and `Eintrp.Ez`.

```
EintrpX = reshape(Eintrp.Ex,size(X));
EintrpY = reshape(Eintrp.Ey,size(Y));
EintrpZ = reshape(Eintrp.Ez,size(Z));
```

Plot the resulting electric field.

```
figure
quiver3(X,Y,Z,EintrpX,EintrpY,EintrpZ,"Color","red")
view([10 10])
```



Input Arguments

`results` — Solution of electrostatic or DC conduction problem
`ElectrostaticResults` object | `ConductionResults` object

Solution of an electrostatic or DC conduction problem, specified as an `ElectrostaticResults` or `ConductionResults` object. Create `results` using the `solve` function.

Example: `results = solve(emagmodel)`

xq — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `interpolateElectricField` evaluates the electric field at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateElectricField` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric field values as 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 `EintrpX = reshape(Eintrp.Ex,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

`yq` — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateElectricField` evaluates the electric field at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateElectricField` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric field values as 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 `EintrpY = reshape(Eintrp.Ey,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

`zq` — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateElectricField` evaluates the electric field at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateElectricField` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns electric field values as 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 `EintrpZ = reshape(Eintrp.Ez,size(zq))`.

Example: `zq = [1 1 0 1.5]`

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. `interpolateElectricField` evaluates the electric field at the coordinate points `querypoints(:,i)` for every `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`Eintrp` — Electric field at query points

`FEStruct` object

Electric field at query points, returned as an `FEStruct` object with the properties representing the spatial components of the electric field at the query points. For query points that are outside the geometry, `Eintrp.Ex(i)`, `Eintrp.Ey(i)`, and `Eintrp.Ez(i)` are `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2021a

Electric field in DC conduction results

The function now interpolates electric field in DC conduction results in addition to electrostatic results.

See Also

`solve` | `interpolateElectricFlux` | `interpolateElectricPotential` | `interpolateCurrentDensity` | `ElectromagneticModel` | `ElectrostaticResults` | `ConductionResults`

interpolateElectricFlux

Package: `pde`

Interpolate electric flux density in electrostatic result at arbitrary spatial locations

Syntax

```
Dintrp = interpolateElectricFlux(electrostaticresults,xq,yq)
Dintrp = interpolateElectricFlux(electrostaticresults,xq,yq,zq)
Dintrp = interpolateElectricFlux(electrostaticresults,querypoints)
```

Description

`Dintrp = interpolateElectricFlux(electrostaticresults,xq,yq)` returns the interpolated electric flux density at the 2-D points specified in `xq` and `yq`.

`Dintrp = interpolateElectricFlux(electrostaticresults,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`Dintrp = interpolateElectricFlux(electrostaticresults,querypoints)` returns the interpolated electric flux density at the points specified in `querypoints`.

Examples

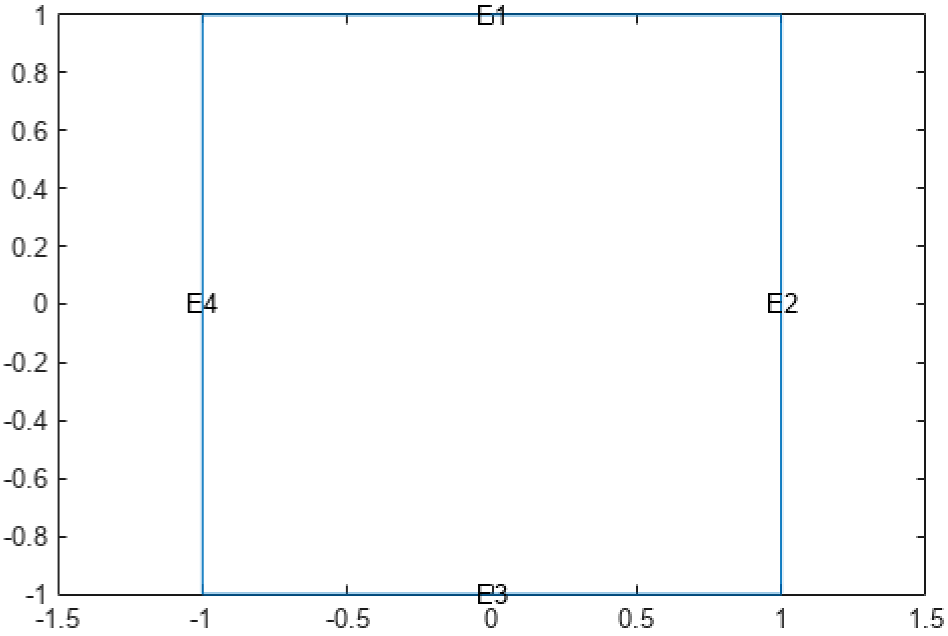
Interpolate Electric Flux Density in 2-D Electrostatic Analysis

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Create a square geometry and include it in the model. Plot the geometry with the edge labels.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1, 'R1', ('R1')');
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-1.5 1.5])
axis equal
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Apply the voltage boundary conditions on the edges of the square.

```
electromagneticBC(emagmodel,"Voltage",0,"Edge",[1 3]);
electromagneticBC(emagmodel,"Voltage",1000,"Edge",[2 4]);
```

Specify the charge density for the entire geometry.

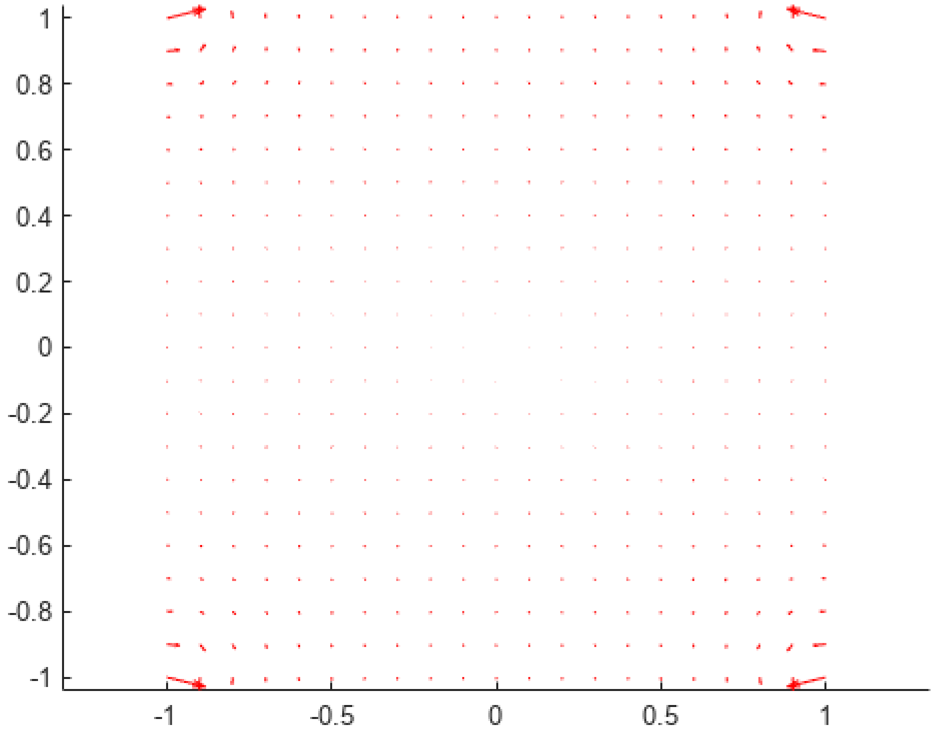
```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model and plot the electric flux density.

```
R = solve(emagmodel);
pdeplot(emagmodel,"FlowData",[R.ElectricFluxDensity.Dx ...
                              R.ElectricFluxDensity.Dy])
axis equal
```



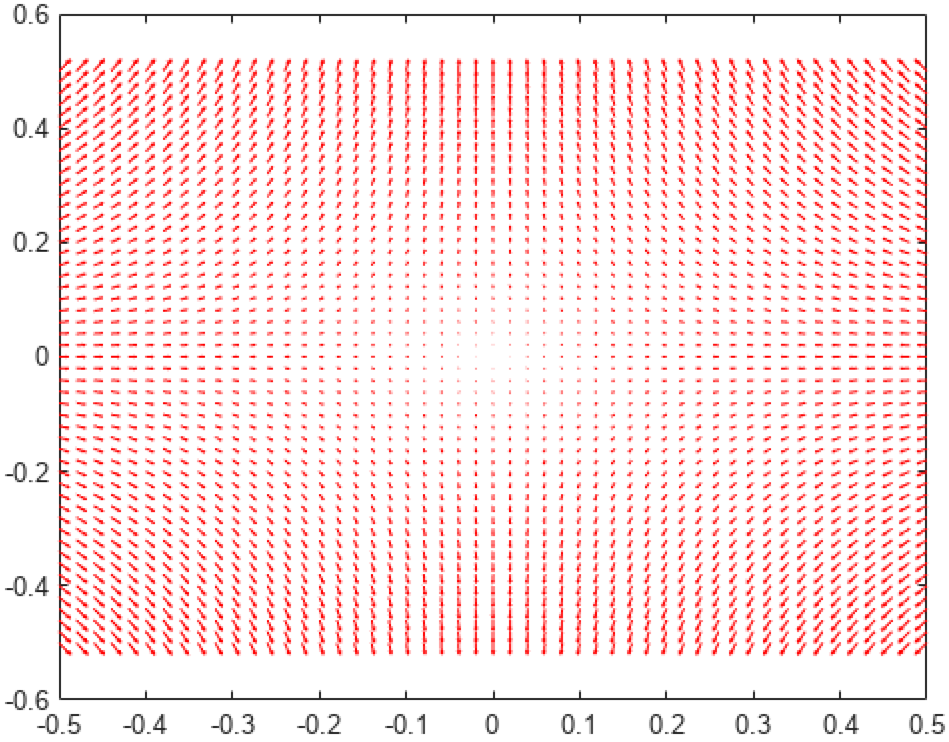
Interpolate the resulting electric flux density to a grid covering the central portion of the geometry, for `x` and `y` from `-0.5` to `0.5`.

```
v = linspace(-0.5,0.5,51);
[X,Y] = meshgrid(v);

Dintrp = interpolateElectricFlux(R,X,Y)
```

```
Dintrp = 
  FEStruct with properties:

    Dx: [2601x1 double]
    Dy: [2601x1 double]
```

Reshape `Dintrp.Dx` and `Dintrp.Dy` and plot the resulting electric flux density.

```
DintrpX = reshape(Dintrp.Dx,size(X));
DintrpY = reshape(Dintrp.Dy,size(Y));

figure
quiver(X,Y,DintrpX,DintrpY,"Color","red")
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Dintrp = interpolateElectricFlux(R,querypoints);
```

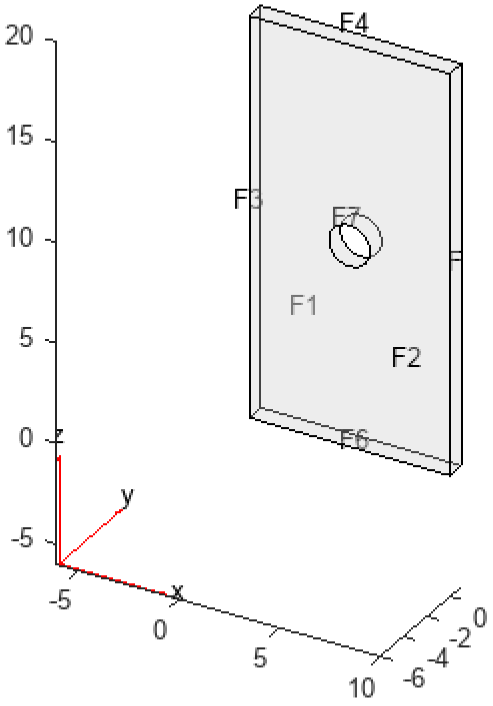
Interpolate Electric Flux Density in 3-D Electrostatic Analysis

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Apply the voltage boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
electromagneticBC(emagmodel,"Voltage",1000,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

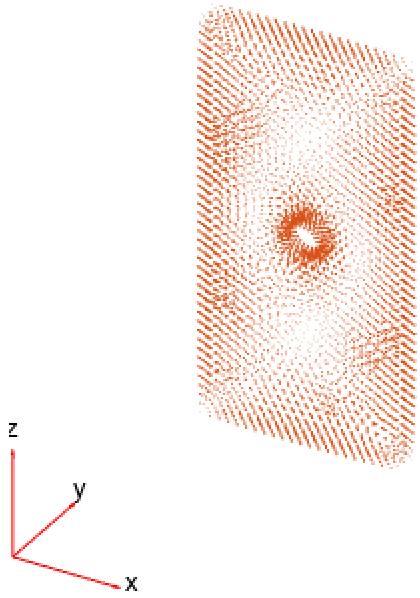
```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:
```

```
      ElectricPotential: [4359x1 double]
          ElectricField: [1x1 FEStruct]
    ElectricFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

Plot the electric flux density.

```
pdeplot3D(emagmodel,"FlowData",[R.ElectricFluxDensity.Dx ...
                                R.ElectricFluxDensity.Dy ...
                                R.ElectricFluxDensity.Dz])
```



Interpolate the resulting electric flux density to a grid covering the central portion of the geometry, for `x`, `y`, and `z`.

```
x = linspace(3,7,7);
y = linspace(0,1,7);
z = linspace(8,12,7);
[X,Y,Z] = meshgrid(x,y,z);

Dintrp = interpolateElectricFlux(R,X,Y,Z)
```

```
Dintrp = 
  FEStruct with properties:

    Dx: [343x1 double]
    Dy: [343x1 double]
```

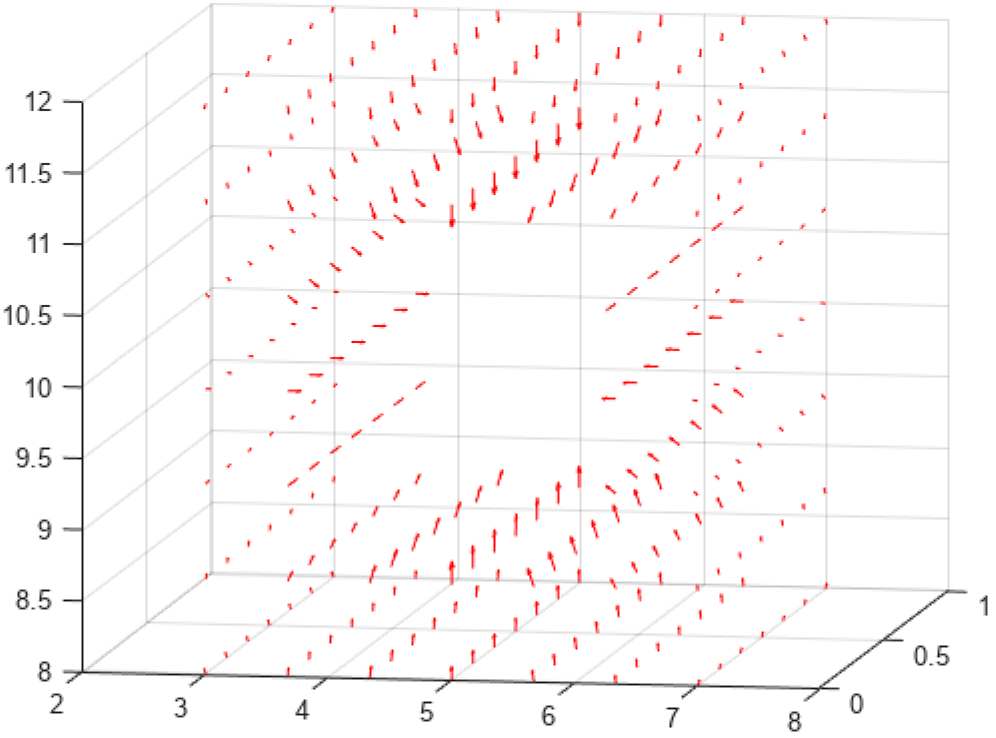
```
    Dz: [343x1 double]
```

Reshape `Dintrp.Dx`, `Dintrp.Dy`, and `Dintrp.Dz`.

```
DintrpX = reshape(Dintrp.Dx,size(X));
DintrpY = reshape(Dintrp.Dy,size(Y));
DintrpZ = reshape(Dintrp.Dz,size(Z));
```

Plot the resulting electric flux density.

```
figure
quiver3(X,Y,Z,DintrpX,DintrpY,DintrpZ,"Color","red")
view([10 10])
```



Input Arguments

`electrostaticresults` — Solution of electrostatic problem
`ElectrostaticResults` object

Solution of thermal problem, specified as an `ElectrostaticResults` object. Create `electrostaticresults` using the `solve` function.

Example: `electrostaticresults = solve(emagmodel)`

`xq` — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `interpolateElectricFlux` evaluates the electric flux density at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateElectricFlux` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric flux density as 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 `DintrpX = reshape(Dintrp.Dx,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateElectricFlux` evaluates the electric flux density at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateElectricFlux` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric flux density as 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 `DintrpY = reshape(Dintrp.Dy,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateElectricFlux` evaluates the electric flux density at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateElectricFlux` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns electric flux density values as 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 `DintrpZ = reshape(Dintrp.Dz,size(zq))`.

Example: `zq = [1 1 0 1.5]`

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. `interpolateElectricFlux` evaluates the electric flux density at the coordinate points `querypoints(:,i)` for every `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`Dintrp` — Electric flux density at query points

`FEStruct`

Electric flux density at query points, returned as an `FEStruct` object with the properties representing the spatial components of the electric flux density at the query points. For query points that are outside the geometry, `Dintrp.Dx(i)`, `Dintrp.Dy(i)`, and `Dintrp.Dz(i)` are `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2021a

See Also

`solve` | `interpolateElectricField` | `interpolateElectricPotential` | `ElectromagneticModel` | `ElectrostaticResults`

interpolateCurrentDensity

Package: `pde`

Interpolate current density in DC conduction result at arbitrary spatial locations

Syntax

```
Jintrp = interpolateCurrentDensity(results,xq,yq)
Jintrp = interpolateCurrentDensity(results,xq,yq,zq)
Jintrp = interpolateCurrentDensity(results,querypoints)
```

Description

`Jintrp = interpolateCurrentDensity(results,xq,yq)` returns the interpolated current density values at the 2-D points specified in `xq` and `yq`.

`Jintrp = interpolateCurrentDensity(results,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`Jintrp = interpolateCurrentDensity(results,querypoints)` returns the interpolated current density values at the points specified in `querypoints`.

Examples

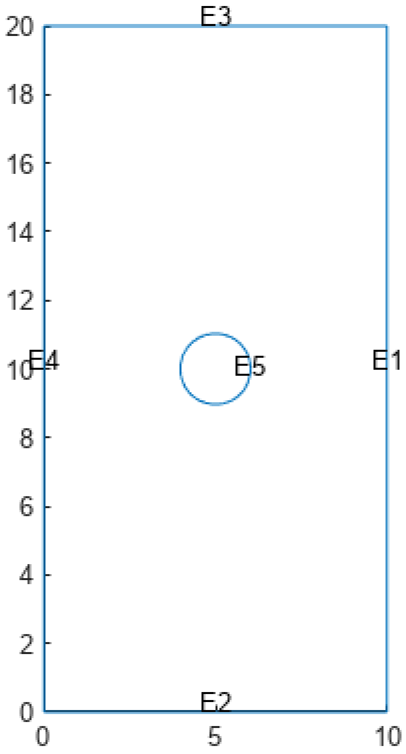
Interpolate Current Density in 2-D DC Conduction Analysis

Create an electromagnetic model for DC conduction analysis.

```
model = createpde("electromagnetic","conduction");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(model,"PlateHolePlanar.stl");
figure
pdegplot(model,"EdgeLabels","on");
```



Specify the conductivity of the material.

```
electromagneticProperties(model,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the top and bottom edges of the plate.

```
electromagneticBC(model,"Voltage",100,"Edge",3);
electromagneticBC(model,"Voltage",200,"Edge",2);
```

Specify the surface current density on the edge representing the hole.

```
electromagneticBC(model,"SurfaceCurrentDensity",200000,"Edge",5);
```

Generate the mesh.

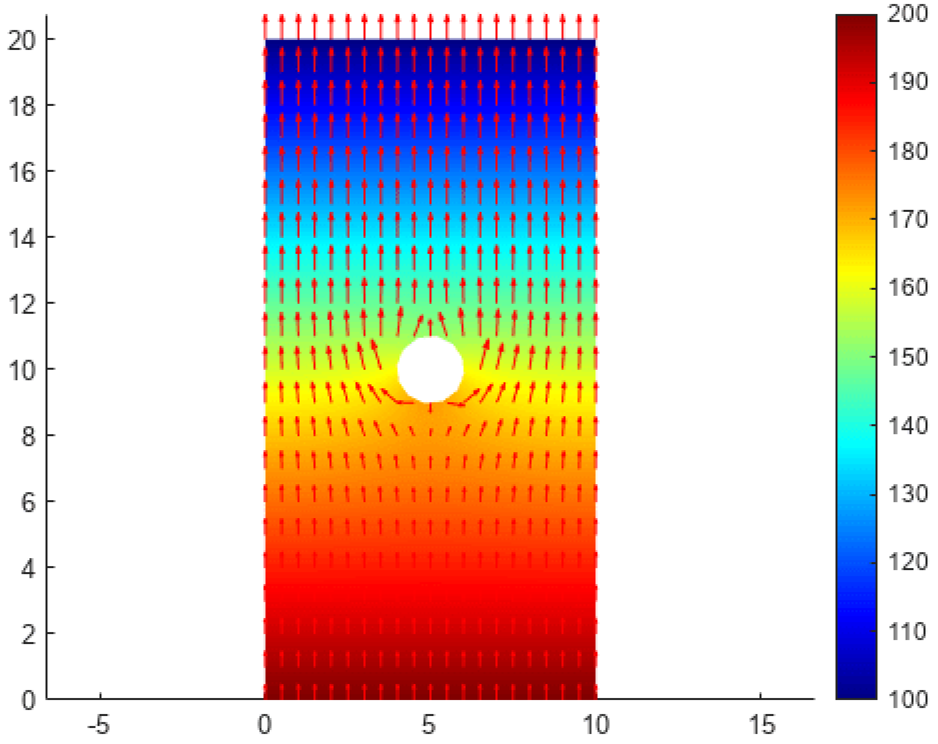
```
generateMesh(model);
```

Solve the model.

```
R = solve(model);
```

Plot the electric potential and current density.

```
figure
pdeplot(model,"XYData",R.ElectricPotential,"ColorMap","jet", ...
              "FlowData",[R.CurrentDensity.Jx R.CurrentDensity.Jy])
axis equal
```



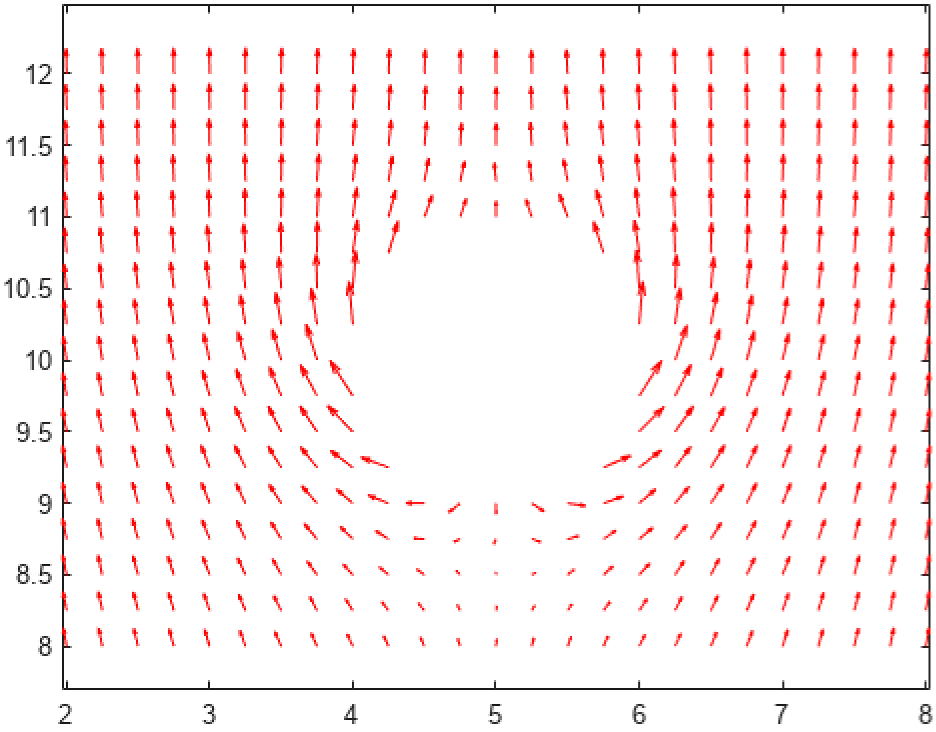
Interpolate the resulting current density to a grid covering the central portion of the geometry.

```
[X,Y] = meshgrid(2:0.25:8,8:0.25:12);
Jintrp = interpolateCurrentDensity(R,X,Y)
```

```
Jintrp = 
  FEStruct with properties:

    Jx: [425x1 double]
    Jy: [425x1 double]
```

Reshape `Jintrp.Jx` and `Jintrp.Jy`, and plot the resulting current density.

```
JintrpX = reshape(Jintrp.Jx,size(X));
JintrpY = reshape(Jintrp.Jy,size(Y));
quiver(X,Y,JintrpX,JintrpY,"Color","red")
axis equal
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Jintrp = interpolateCurrentDensity(R,querypoints);
```

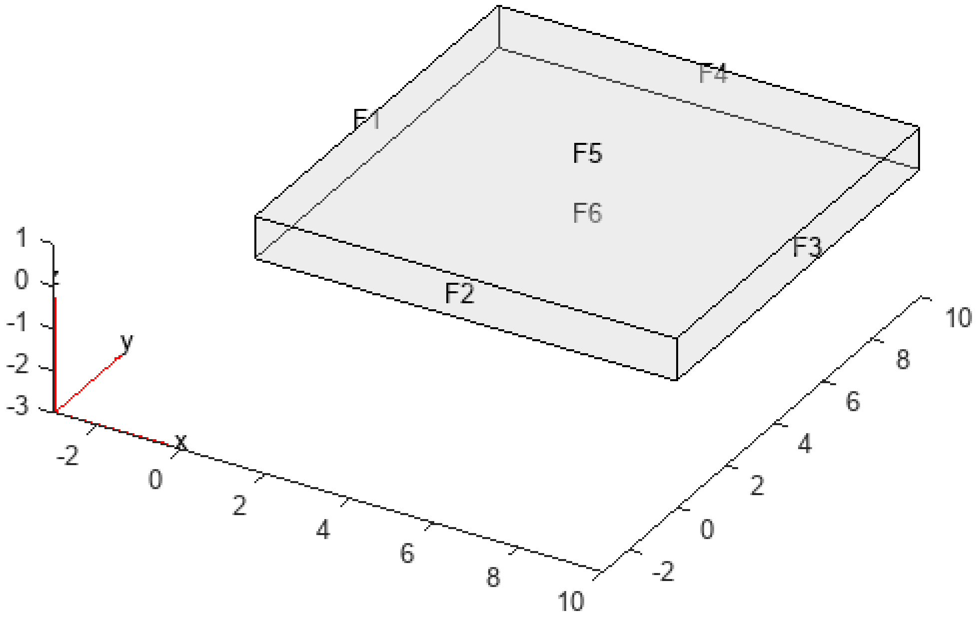
Interpolate Current Density in 3-D DC Conduction Analysis

Create an electromagnetic model for DC conduction analysis.

```
model = createpde("electromagnetic","conduction");
```

Import and plot the geometry representing a 10-by-10-by-1 solid plate.

```
g = importGeometry(model,"Plate10x10x1.stl");
pdegplot(model,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the conductivity of the material.

```
electromagneticProperties(model,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the two faces of the plate.

```
electromagneticBC(model,"Voltage",0,"Face",[1 3]);
```

Specify the surface current density on the top of the plate.

```
electromagneticBC(model,"SurfaceCurrentDensity",100,"Face",5);
```

Generate the mesh.

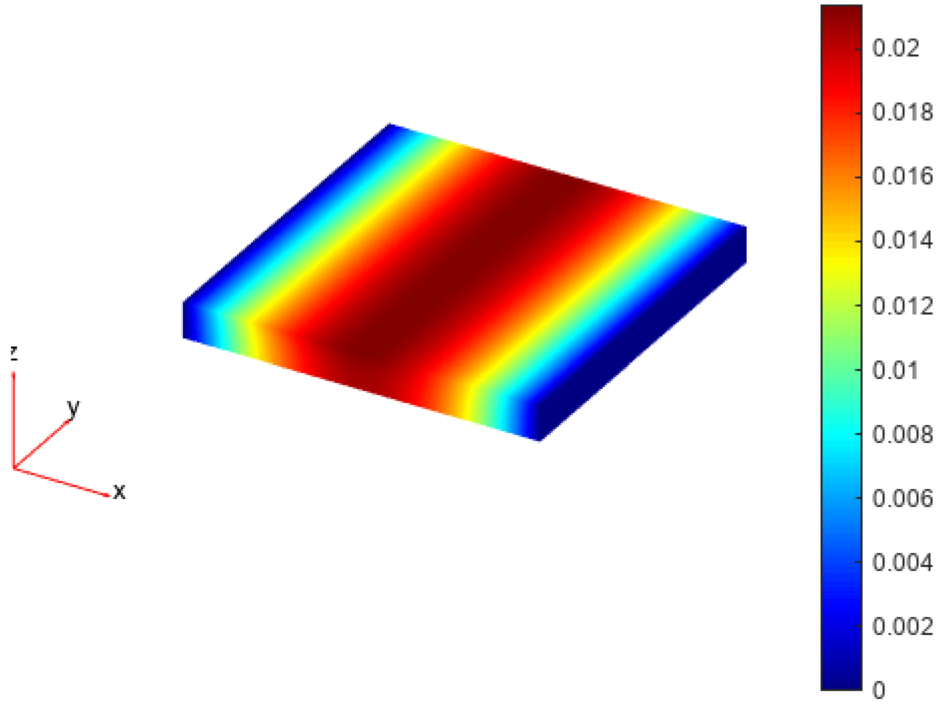
```
generateMesh(model);
```

Solve the model.

```
R = solve(model);
```

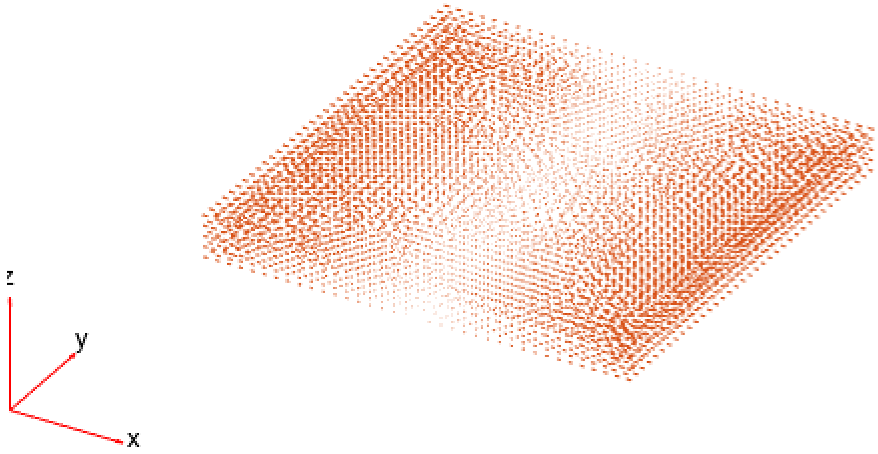
Plot the electric potential.

```
figure
pdeplot3D(model,"ColorMapData",R.ElectricPotential)
```



Plot the current density.

```
figure
pdeplot3D(model,"FlowData",[R.CurrentDensity.Jx, ...
                             R.CurrentDensity.Jy, ...
                             R.CurrentDensity.Jz])
```



Interpolate the resulting current density to a coarser grid.

```
[X,Y,Z] = meshgrid(0:10,0:10,0:0.5:1);
Jintrp = interpolateCurrentDensity(R,X,Y,Z)
```

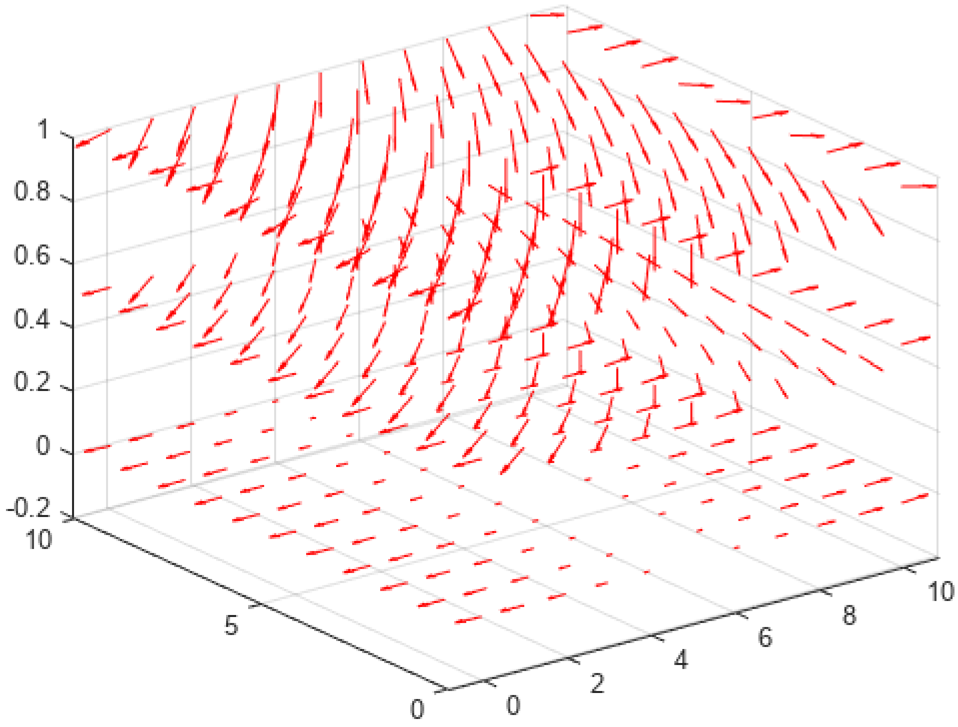
```
Jintrp = 
  FEStruct with properties:

    Jx: [363x1 double]
    Jy: [363x1 double]
    Jz: [363x1 double]
```

Reshape `Jintrp.Jx`, `Jintrp.Jy`, and `Jintrp.Jz`.

```
JintrpX = reshape(Jintrp.Jx,size(X));
JintrpY = reshape(Jintrp.Jy,size(Y));
JintrpZ = reshape(Jintrp.Jz,size(Z));
```

Plot the resulting current density.

```
figure
quiver3(X,Y,Z,JintrpX,JintrpY,JintrpZ,"Color","red")
```



Input Arguments

`results` — Solution of DC conduction problem

`ConductionResults` object

Solution of a DC conduction problem, specified as a `ConductionResults` object. Create `results` using the `solve` function.

Example: `results = solve(emagmodel)`

xq — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateCurrentDensity` evaluates the current density at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateCurrentDensity` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns current density values as 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 `Jintrp = reshape(Jintrp,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateCurrentDensity` evaluates the current density at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateCurrentDensity` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns current density values as 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 `Jintrp = reshape(Jintrp,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateCurrentDensity` evaluates the current density at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateCurrentDensity` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns current density values as 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 `Jintrp = reshape(Jintrp,size(zq))`.

Example: `zq = [1 1 0 1.5]`

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. `interpolateCurrentDensity` evaluates the current density at the coordinate points `querypoints(:,i)` for every `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`Jintrp` — Current density at query points
`FEStruct` object

Current density at query points, returned as an `FEStruct` object with the properties representing the spatial components of the current density at the query points. For query points that are outside the geometry, `Jintrp.Jx(i)`, `Jintrp.Jy(i)`, and `Jintrp.Jz(i)` are `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2022b

See Also

`solve` | `interpolateElectricPotential` | `interpolateElectricField` | `ElectromagneticModel` | `ConductionResults`

interpolateHarmonicField

Package: `pde`

Interpolate electric or magnetic field in harmonic result at arbitrary spatial locations

Syntax

```
EHintrp = interpolateHarmonicField(harmonicresults,xq,yq)
EHintrp = interpolateHarmonicField(harmonicresults,xq,yq,zq)
EHintrp = interpolateHarmonicField(harmonicresults,querypoints)
```

Description

`EHintrp = interpolateHarmonicField(harmonicresults,xq,yq)` returns the interpolated electric or magnetic field values at the 2-D points specified in `xq` and `yq`.

`EHintrp = interpolateHarmonicField(harmonicresults,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`EHintrp = interpolateHarmonicField(harmonicresults,querypoints)` returns the interpolated electric or magnetic field values at the points specified in `querypoints`.

Examples

Interpolate Electric Field in 2-D Harmonic Analysis

Solve a simple scattering problem and interpolate the x-component of the resulting electric field. A scattering problem computes the waves reflected by a square object illuminated by incident waves.

Create an electromagnetic model for harmonic analysis.

```
emagmodel = createpde("electromagnetic","harmonic");
```

Specify the wave number $k = \omega/\alpha$ as 4π.

```
k = 4*pi;
```

Represent the square surface with a diamond-shaped hole. Define a diamond in a square, place them in one matrix, and create a set formula that subtracts the diamond from the square.

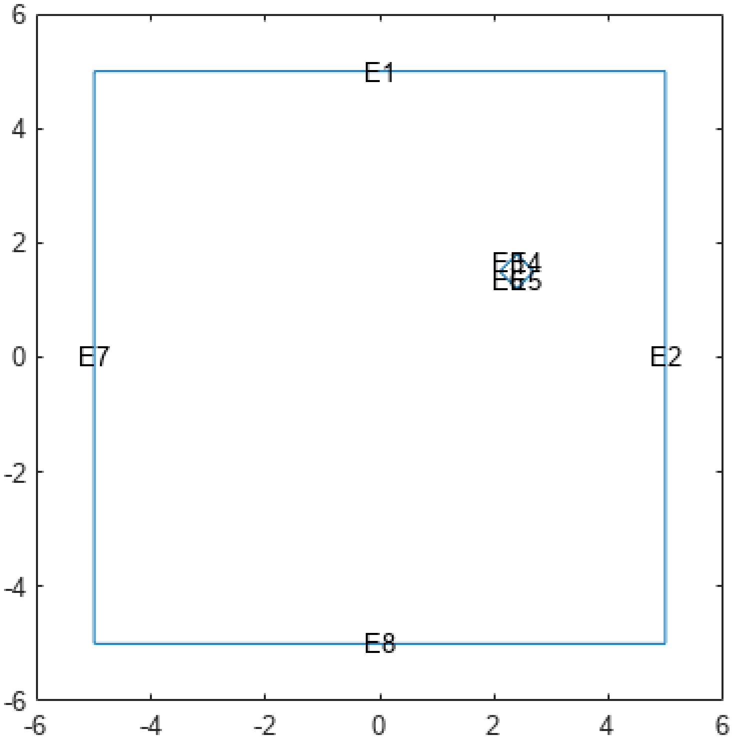
```
square = [3; 4; -5; -5; 5; 5; -5; 5; 5; -5];
diamond = [2; 4; 2.1; 2.4; 2.7; 2.4; 1.5; 1.8; 1.5; 1.2];
gd = [square,diamond];
ns = char('square','diamond')';
sf = 'square - diamond';
```

Create the geometry.

```
g = decsg(gd,sf,ns);
geometryFromEdges(emagmodel,g);
```

Include the geometry in the model and plot it with the edge labels.

```
figure;
pdegplot(emagmodel,"EdgeLabels","on");
xlim([-6,6])
ylim([-6,6])
```



Specify the vacuum permittivity and permeability values as 1.

```
emagmodel.VacuumPermittivity = 1;
emagmodel.VacuumPermeability = 1;
```

Specify the relative permittivity, relative permeability, and conductivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1, ...
                                    "RelativePermeability",1, ...
                                    "Conductivity",0);
```

Apply the absorbing boundary condition on the edges of the square. Specify the thickness and attenuation rate for the absorbing region by using the `Thickness`, `Exponent`, and `Scaling` arguments.

```
electromagneticBC(emagmodel,"Edge",[1 2 7 8], ...
                            "FarField","absorbing", ...
                            "Thickness",2, ...
                            "Exponent",4, ...
                            "Scaling",1);
```

Apply the boundary condition on the edges of the diamond.

```
innerBCFunc = @(location,~) [-exp(-1i*k*location.x);
                             zeros(1,length(location.x))];
bInner = electromagneticBC(emagmodel,"Edge",[3 4 5 6], ...
                                      "ElectricField",innerBCFunc);
```

Generate a mesh.

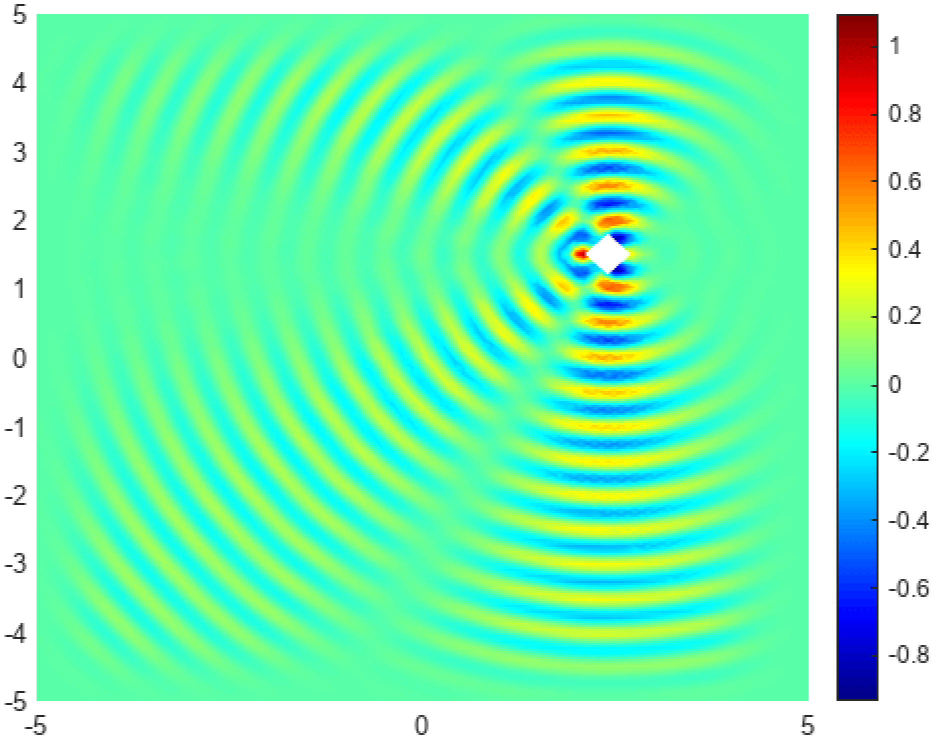
```
generateMesh(emagmodel,"Hmax",0.1);
```

Solve the harmonic analysis model for the frequency `k` = 4π.

```
result = solve(emagmodel,"Frequency",k);
```

Plot the real part of the *x*-component of the resulting electric field.

```
u = result.ElectricField;

figure
pdeplot(emagmodel,"XYData",real(u.Ex),"Mesh","off");
colormap(jet)
```

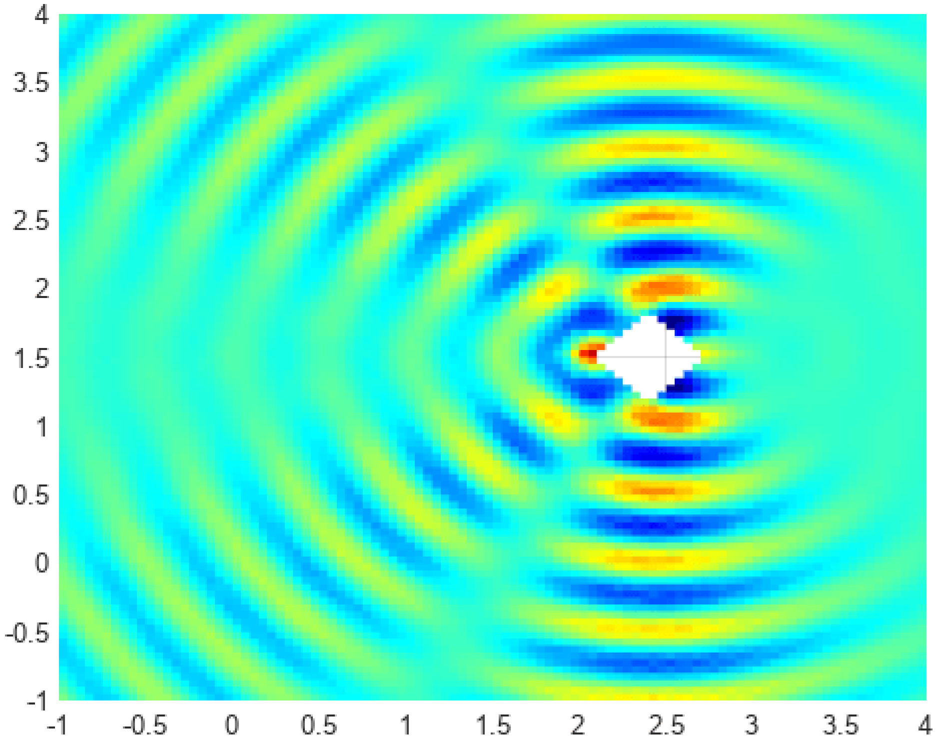


Interpolate the resulting electric field to a grid covering the portion of the geometry, for *x* and *y* from -1 to 4.

```
v = linspace(-1,4,101);
[X,Y] = meshgrid(v);
Eintrp = interpolateHarmonicField(result,X,Y);
```

Reshape `Eintrp.Ex` and plot the *x*-component of the resulting electric field.

```
EintrpX = reshape(Eintrp.ElectricField.Ex,size(X));

figure
surf(X,Y,real(EintrpX),"LineStyle","none");
view(0,90)
colormap(jet)
```



Interpolate Magnetic Field in 3-D Harmonic Analysis

Interpolate the *x*-component of the magnetic field in a harmonic analysis of a 3-D model.

Create an electromagnetic model for harmonic analysis.

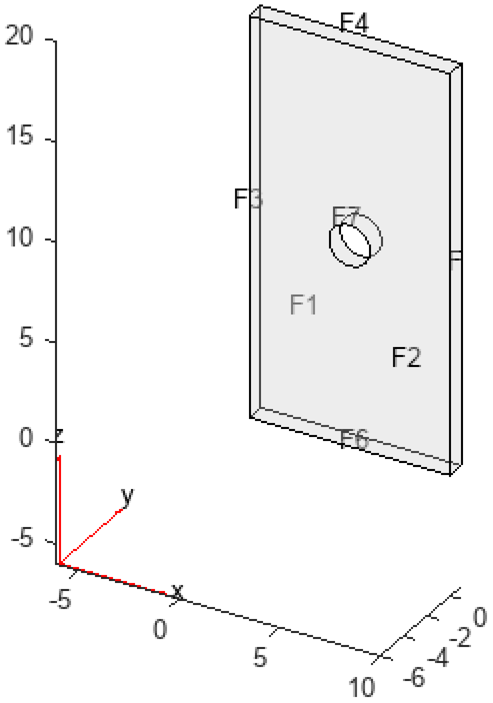
```
emagmodel = createpde("electromagnetic","harmonic");
```

By default, the field type is electric. Use the `FieldType` property of the model to change the field type to magnetic.

```
emagmodel.FieldType = "magnetic";
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permittivity and permeability values in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permittivity, relative permeability, and conductivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1, ...
                                    "RelativePermeability",6, ...
                                    "Conductivity",60);
```

Specify the current density for the entire geometry. For harmonic analysis with the magnetic field type, the toolbox uses the curl of the specified current density.

```
electromagneticSource(emagmodel,"CurrentDensity",[1;1;1]);
```

Apply the absorbing boundary condition with a thickness of 0.1 on the side faces.

```
electromagneticBC(emagmodel,"Face",3:6, ...
                            "FarField","absorbing", ...
                            "Thickness",0.1);
```

Specify the magnetic field on the face bordering the round hole in the center of the geometry.

```
electromagneticBC(emagmodel,"Face",7,"MagneticField",[1000;0;0]);
```

Generate a mesh.

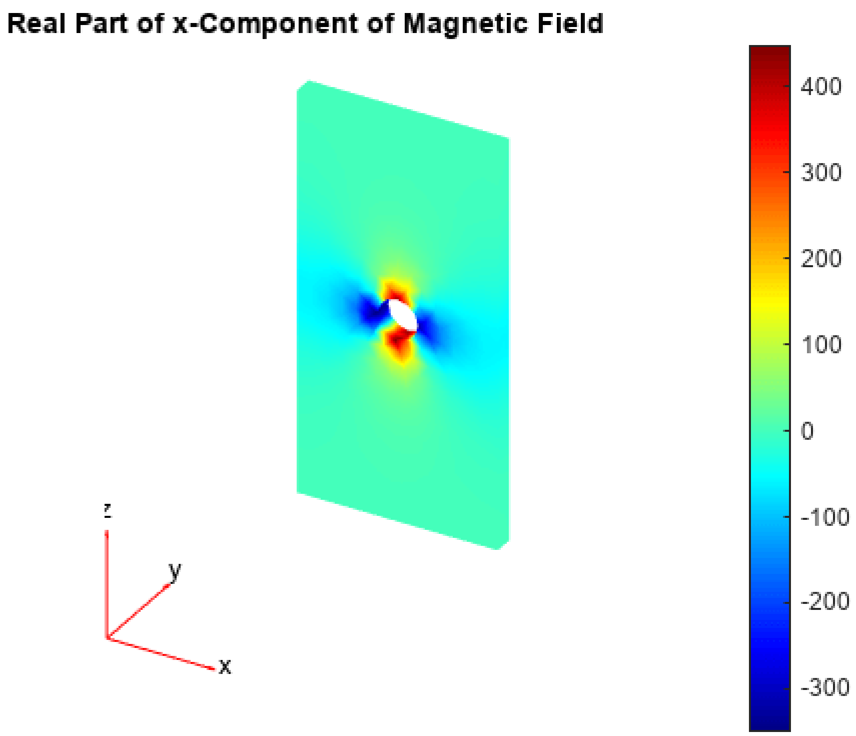
```
generateMesh(emagmodel);
```

Solve the model for a frequency of 50.

```
result = solve(emagmodel,"Frequency",50);
```

Plot the real part of the *x*-component of the resulting magnetic field.

```
u = result.MagneticField;
figure
pdeplot3D(emagmodel,"ColorMapData",real(u.Hx));
colormap jet
xlabel 'x'
ylabel 'y'
title("Real Part of x-Component of Magnetic Field")
```



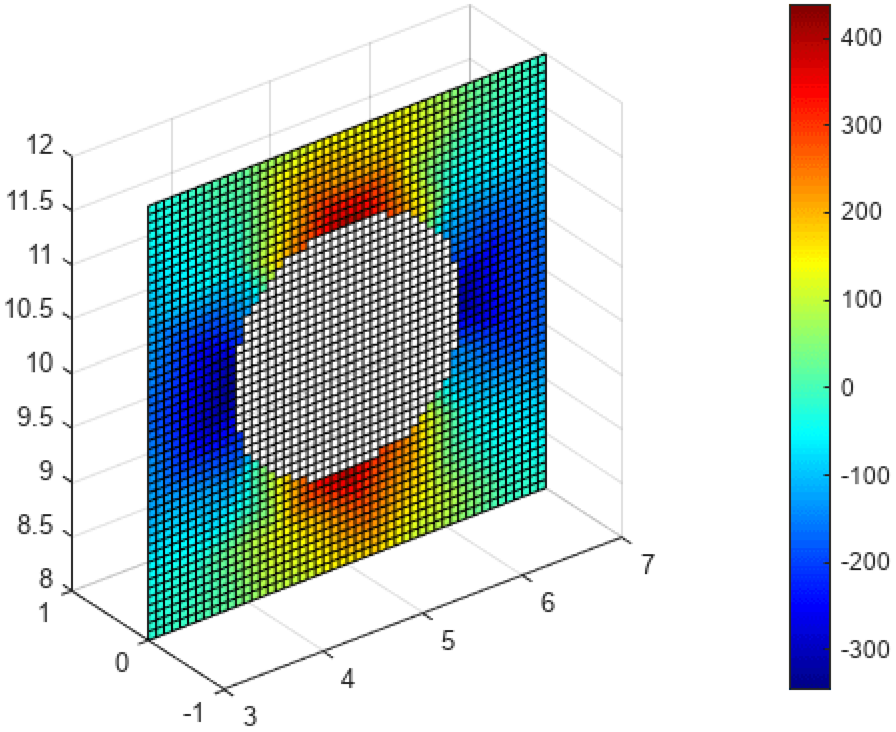
Interpolate the resulting magnetic field to a grid covering the central portion of the geometry, for *x*, *y*, and *z*.

```
x = linspace(3,7,51);
y = linspace(0,1,51);
z = linspace(8,12,51);
[X,Y,Z] = meshgrid(x,y,z);

Hintrp = interpolateHarmonicField(result,X,Y,Z)
```

```
Hintrp = struct with fields:
    MagneticField: [1x1 FEStruct]
```

Reshape `Hintrp.Hx` and plot the *x*-component of the resulting magnetic field as a slice plot for *y* = 0.

```
HintrpX = reshape(Hintrp.MagneticField.Hx,size(X));

figure
slice(X,Y,Z,real(HintrpX),[],0,[],'cubic')
axis equal
colorbar
colormap(jet)
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:),Z(:)]';
Hintrp = interpolateHarmonicField(result,querypoints)
```

```
Hintrp = struct with fields:
    MagneticField: [1x1 FEStruct]
```

Input Arguments

`harmonicresults` — Solution of harmonic electromagnetic problem
`HarmonicResults` object

Solution of a harmonic electromagnetic problem, specified as a `HarmonicResults` object. Create `harmonicresults` using the `solve` function.

Example: `harmonicresults = solve(emagmodel,"Frequency",omega)`

xq — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `interpolateHarmonicField` evaluates the electric or magnetic field at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every index `i`.Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateHarmonicField` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric or magnetic field values as 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 `EHintrpX = reshape(EHintrp.Ex,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateHarmonicField` evaluates the electric or magnetic field at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every index `i`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateHarmonicField` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric or magnetic field values as 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 `EHintrpY = reshape(EHintrp.Ey,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateHarmonicField` evaluates the electric or magnetic field at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every index `i`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateHarmonicField` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns electric or magnetic field values as 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 `EHintrpZ = reshape(EHintrp.Ez,size(zq))`.

Example: `zq = [1 1 0 1.5]`

Data Types: `double`

`querypoints` — Query points
real matrix

Query points, specified as a real matrix with either two rows for a 2-D geometry or three rows for a 3-D geometry. `interpolateHarmonicField` evaluates the electric or magnetic field at the coordinate points `querypoints(:,i)` for every index `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`EHintrp` — Electric or magnetic field at query points

`FEStruct` object

Electric or magnetic field at query points, returned as an `FEStruct` object with the properties representing the spatial components of the electric or magnetic field at the query points. For query points that are outside the geometry, `EHintrp.Ex(i)`, `EHintrp.Ey(i)`, `EHintrp.Ez(i)`, `EHintrp.Hx(i)`, `EHintrp.Hy(i)`, and `EHintrp.Hz(i)` are `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2022a

See Also

`solve` | `ElectromagneticModel` | `HarmonicResults`

interpolateElectricPotential

Package: `pde`

Interpolate electric potential in electrostatic or DC conduction result at arbitrary spatial locations

Syntax

```
Vintrp = interpolateElectricPotential(results,xq,yq)
Vintrp = interpolateElectricPotential(results,xq,yq,zq)
Vintrp = interpolateElectricPotential(results,querypoints)
```

Description

`Vintrp = interpolateElectricPotential(results,xq,yq)` returns the interpolated electric potential values at the 2-D points specified in `xq` and `yq`.

`Vintrp = interpolateElectricPotential(results,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`Vintrp = interpolateElectricPotential(results,querypoints)` returns the interpolated electric potential values at the points specified in `querypoints`.

Examples

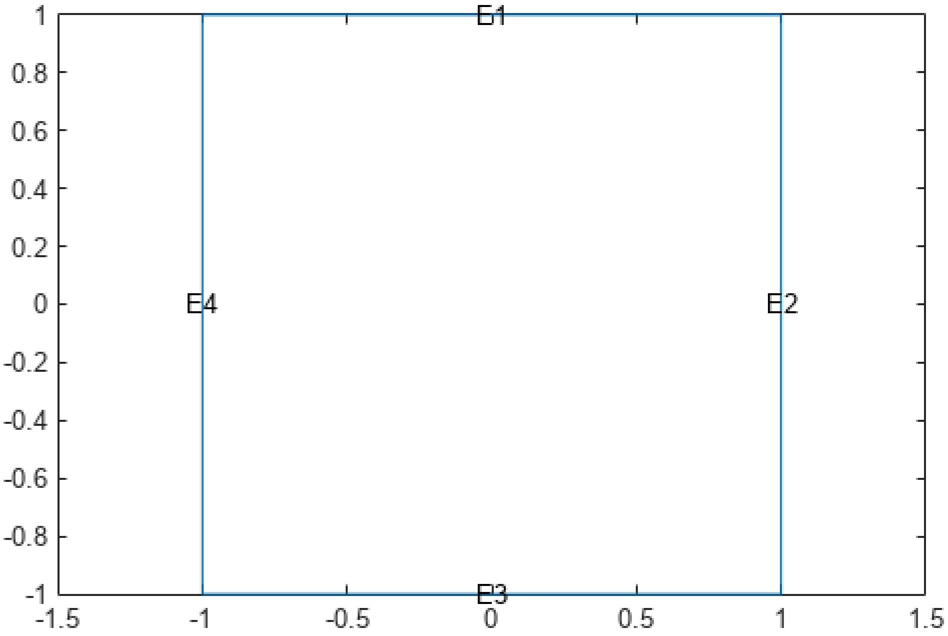
Interpolate Electric Potential in 2-D Electrostatic Analysis

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Create a square geometry and include it in the model. Plot the geometry with the edge labels.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1, 'R1', ('R1')');
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-1.5 1.5])
axis equal
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Apply the voltage boundary conditions on the edges of the square.

```
electromagneticBC(emagmodel,"Voltage",0,"Edge",[1 3]);
electromagneticBC(emagmodel,"Voltage",1000,"Edge",[2 4]);
```

Specify the charge density for the entire geometry.

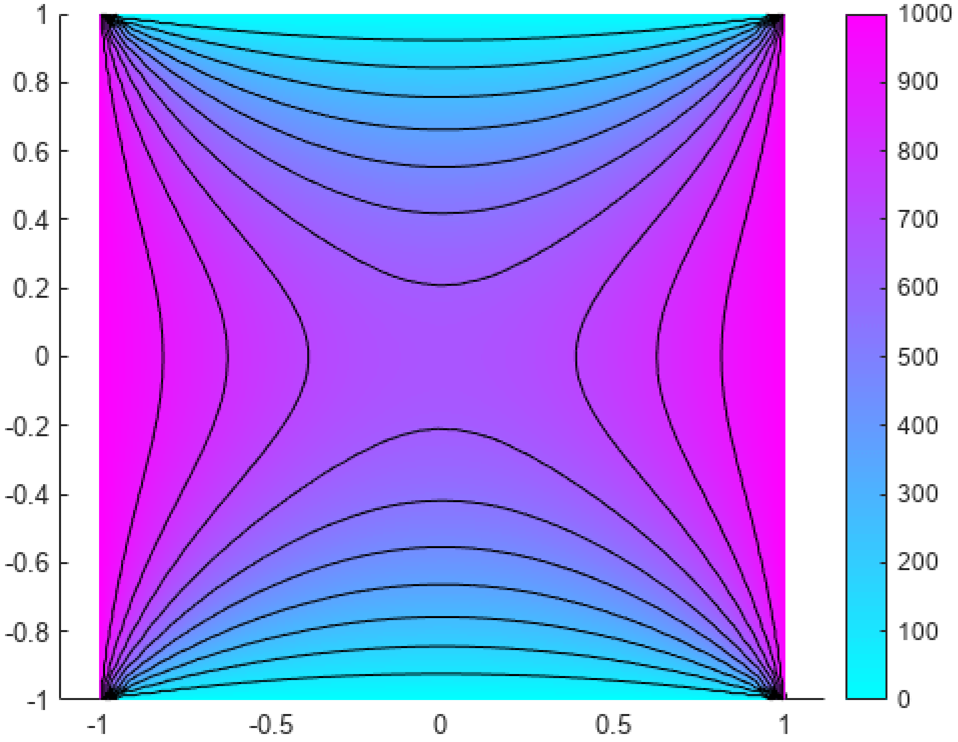
```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model and plot the electric potential.

```
R = solve(emagmodel);
pdeplot(emagmodel,"XYData",R.ElectricPotential, ...
                  "Contour","on")
axis equal
```



Interpolate the resulting electric potential to a grid covering the central portion of the geometry, for `x` and `y` from `-0.5` to `0.5`.

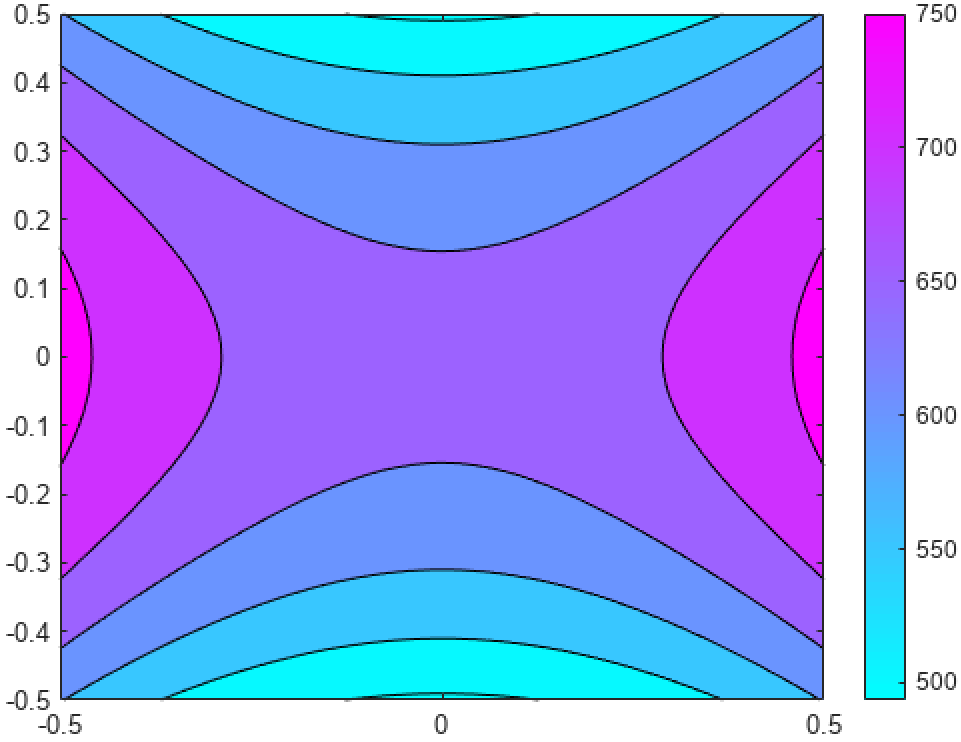
```
v = linspace(-0.5,0.5,51);
[X,Y] = meshgrid(v);

Vintrp = interpolateElectricPotential(R,X,Y)
```

```
Vintrp = 2601×1

  602.2959
  616.0208
  629.0498
  641.4049
  653.0828
  664.0757
  674.4209
  684.1432
  693.2704
  701.8026
      ⋮
```

Reshape `Vintrp` and plot the resulting electric potential.

```
Vintrp = reshape(Vintrp,size(X));

figure
```

```
contourf(X,Y,Vintrp)
colormap(cool)
colorbar
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Vintrp = interpolateElectricPotential(R,querypoints);
```

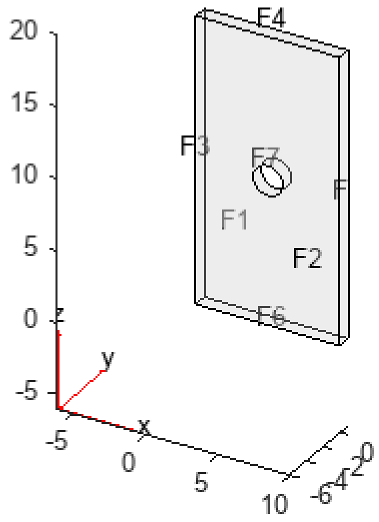
Interpolate Electric Potential in 3-D Electrostatic Analysis

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Apply the voltage boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
electromagneticBC(emagmodel,"Voltage",1000,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

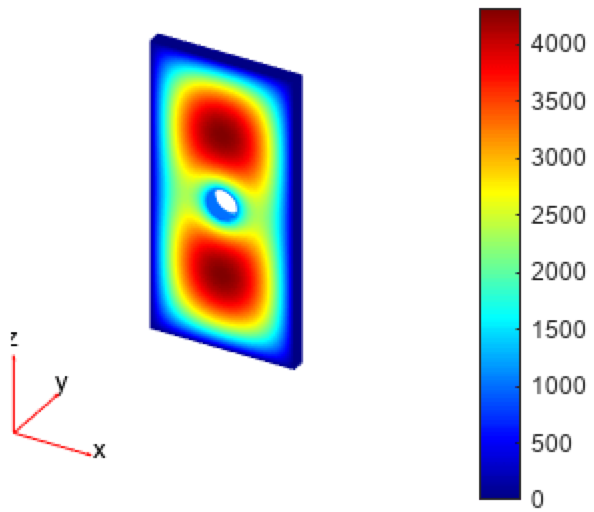
Solve the model.

```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:

      ElectricPotential: [4359x1 double]
          ElectricField: [1x1 FEStruct]
    ElectricFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

Plot the electric potential.

```
pdeplot3D(emagmodel,"ColorMapData",R.ElectricPotential)
```



Interpolate the resulting electric potential to a grid covering the entire geometry, for `x`, `y`, and `z`.

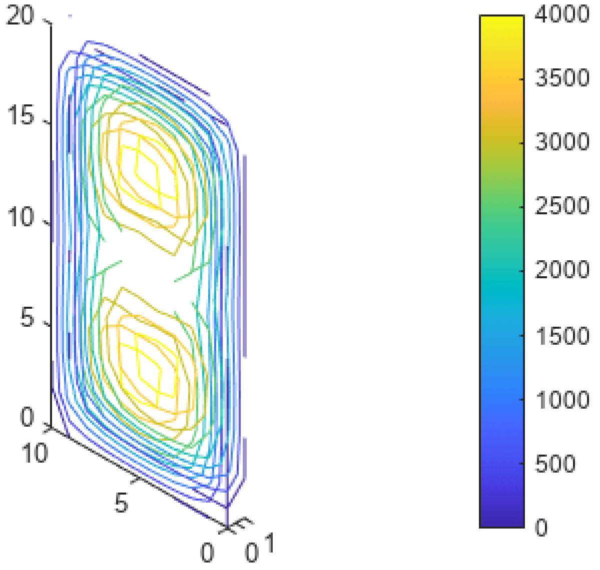
```
x = linspace(0,10,11);
y = linspace(0,1,5);
z = linspace(0,20,11);
[X,Y,Z] = meshgrid(x,y,z);

Vintrp = interpolateElectricPotential(R,X,Y,Z);
```

Reshape `Vintrp`.

```
Vintrp = reshape(Vintrp,size(X));
```

Plot the resulting electric potential as a contour slice plot for two values of the *y*-coordinate.

```
figure
contourslice(X,Y,Z,Vintrp,[],[0 1],[])
view([10,10,-10])
axis equal
colorbar
```



Input Arguments

`results` — Solution of electrostatic or DC conduction problem

`ElectrostaticResults` object | `ConductionResults` object

Solution of an electrostatic or DC conduction problem, specified as an `ElectrostaticResults` or `ConductionResults` object. Create `results` using the `solve` function.

Example: `results = solve(emagmodel)`

xq — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateElectricPotential` evaluates the electric potential at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateElectricPotential` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric potential values as 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 `Vintrp = reshape(Vintrp,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

yq — *y*-coordinate query points

real array

y-coordinate query points, specified as a real array. `interpolateElectricPotential` evaluates the electric potential at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateElectricPotential` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns electric potential values as 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 `Vintrp = reshape(Vintrp,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateElectricPotential` evaluates the electric potential at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateElectricPotential` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns electric potential values as 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 `Vintrp = reshape(Vintrp,size(zq))`.

Example: `zq = [1 1 0 1.5]`

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. `interpolateElectricPotential` evaluates the electric potential at the coordinate points `querypoints(:,i)` for every `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`Vintrp` — Electric potential at query points
vector

Electric potential at query points, returned as a vector. For query points that are outside the geometry, `Vintrp(i)` = `NaN`.

Version History

Introduced in R2021a

Electric potential in DC conduction results

The function now interpolates electric potential in DC conduction results in addition to electrostatic results.

See Also

`solve` | `interpolateElectricField` | `interpolateElectricFlux` | `interpolateCurrentDensity` | `ElectromagneticModel` | `ElectrostaticResults` | `ConductionResults`

interpolateMagneticField

Package: `pde`

Interpolate magnetic field in magnetostatic result at arbitrary spatial locations

Syntax

```
Hintrp = interpolateMagneticField(magnetostaticresults,xq,yq)
Hintrp = interpolateMagneticField(magnetostaticresults,xq,yq,zq)
Hintrp = interpolateMagneticField(magnetostaticresults,querypoints)
```

Description

`Hintrp = interpolateMagneticField(magnetostaticresults,xq,yq)` returns the interpolated magnetic field values at the 2-D points specified in `xq` and `yq`.

`Hintrp = interpolateMagneticField(magnetostaticresults,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`Hintrp = interpolateMagneticField(magnetostaticresults,querypoints)` returns the interpolated magnetic field values at the points specified in `querypoints`.

Examples

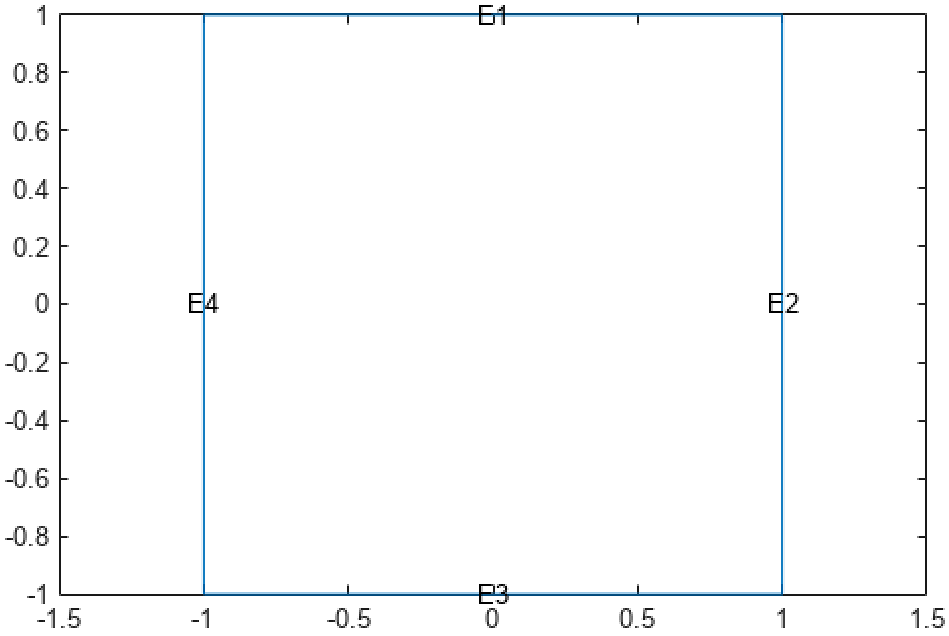
Interpolate Magnetic Field in 2-D Magnetostatic Analysis

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Create a square geometry and include it in the model. Plot the geometry with the edge labels.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1,'R1',('R1')');
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-1.5 1.5])
axis equal
```



Specify the vacuum permeability in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the boundaries of the square.

```
electromagneticBC(emagmodel,"MagneticPotential",0,"Edge",[1 3]);
electromagneticBC(emagmodel,"MagneticPotential",0.01,"Edge",[2 4]);
```

Specify the current density for the entire geometry.

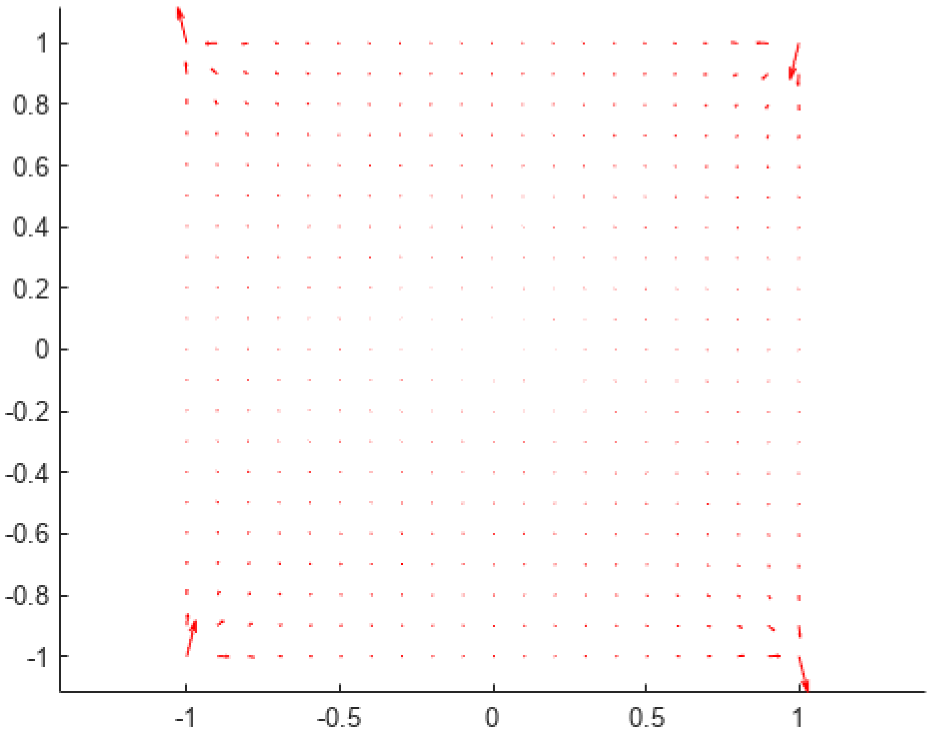
```
electromagneticSource(emagmodel,"CurrentDensity",0.5);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model and plot the magnetic field.

```
R = solve(emagmodel);
pdeplot(emagmodel,"FlowData",[R.MagneticField.Hx ...
                              R.MagneticField.Hy])
axis equal
```



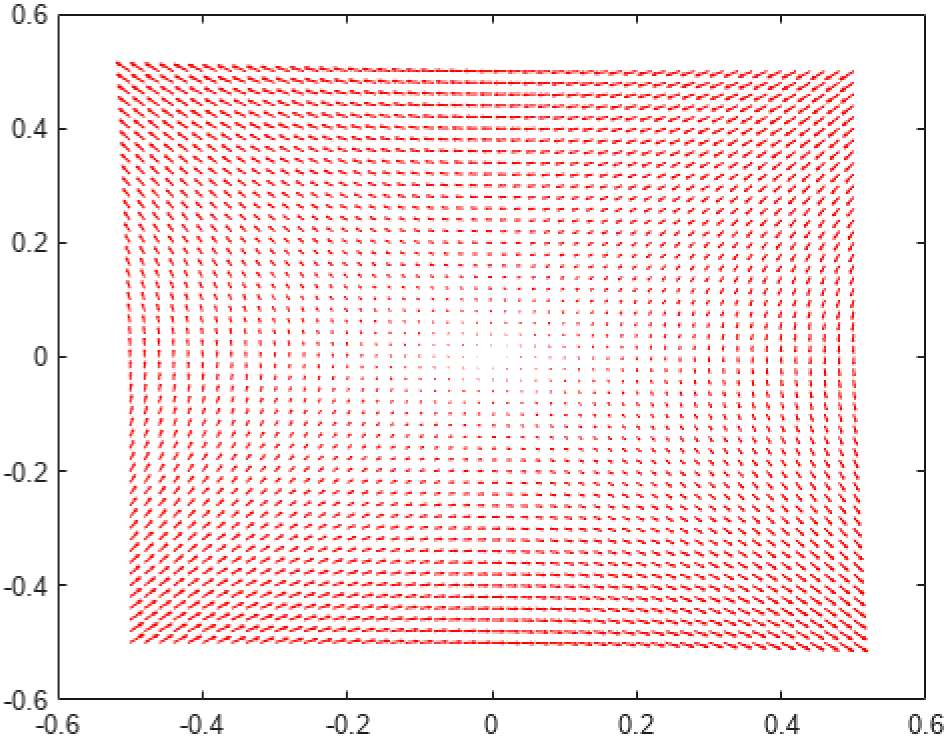
Interpolate the resulting magnetic field to a grid covering the central portion of the geometry, for `x` and `y` from `-0.5` to `0.5`.

```
v = linspace(-0.5,0.5,51);
[X,Y] = meshgrid(v);

Hintrp = interpolateMagneticField(R,X,Y)
```

```
Hintrp = 
  FEStruct with properties:

    Hx: [2601x1 double]
    Hy: [2601x1 double]
```

Reshape `Hintrp.Hx` and `Hintrp.Hy` and plot the resulting electric field.

```
HintrpX = reshape(Hintrp.Hx,size(X));
HintrpY = reshape(Hintrp.Hy,size(Y));

figure
quiver(X,Y,HintrpX,HintrpY,"Color","red")
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Hintrp = interpolateMagneticField(R,querypoints);
```

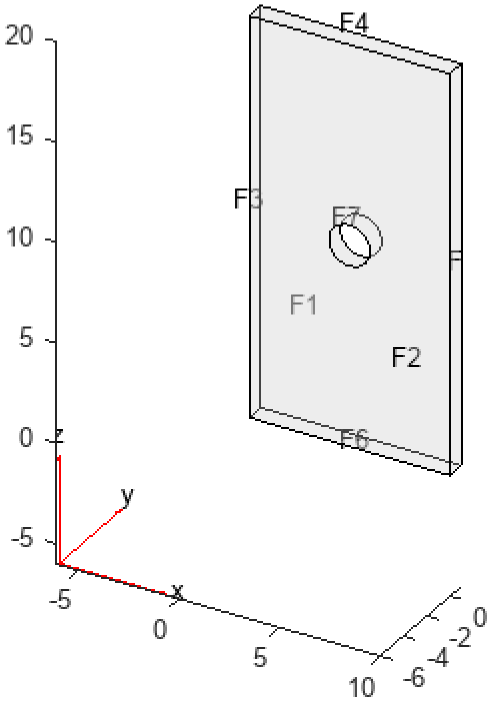
Interpolate Magnetic Field in 3-D Magnetostatic Analysis

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Specify the current density for the entire geometry.

```
electromagneticSource(emagmodel,"CurrentDensity",[0;0;0.5]);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Generate the linear mesh.

```
generateMesh(emagmodel,"GeometricOrder","linear");
```

Solve the model.

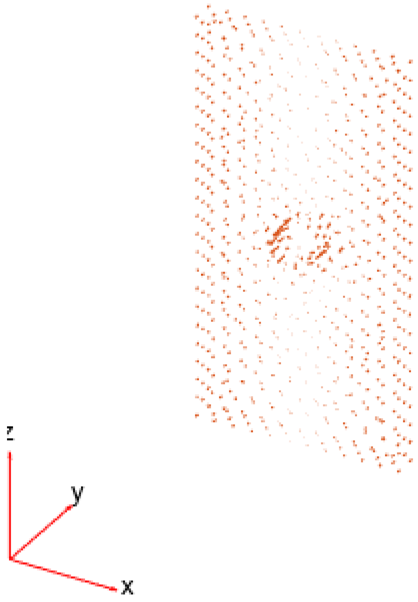
```
R = solve(emagmodel)
```

```
R = 
  MagnetostaticResults with properties:
```

```
       MagneticPotential: [1x1 FEStruct]
           MagneticField: [1x1 FEStruct]
     MagneticFluxDensity: [1x1 FEStruct]
                    Mesh: [1x1 FEMesh]
```

Plot the magnetic field density.

```
pdeplot3D(emagmodel,"FlowData",[R.MagneticField.Hx ...
                                R.MagneticField.Hy ...
                                R.MagneticField.Hz])
```



Interpolate the resulting magnetic field to a grid covering the central portion of the geometry, for `x`, `y`, and `z`.

```
x = linspace(3,7,5);
y = linspace(0,1,5);
z = linspace(8,12,5);
[X,Y,Z] = meshgrid(x,y,z);
Hintrp = interpolateMagneticField(R,X,Y,Z)
```

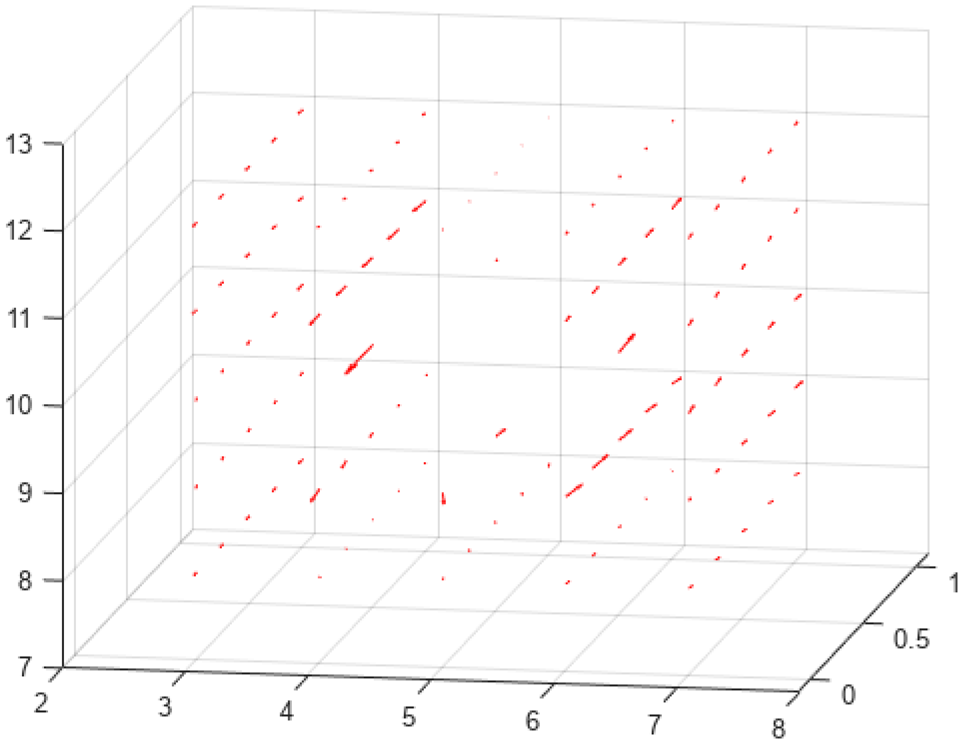
```
Hintrp = 
  FEStruct with properties:

    Hx: [125x1 double]
    Hy: [125x1 double]
    Hz: [125x1 double]
```

Reshape `Hintrp.Hx`, `Hintrp.Hy`, and `Hintrp.Hz`.

```
HintrpX = reshape(Hintrp.Hx,size(X));
HintrpY = reshape(Hintrp.Hy,size(Y));
HintrpZ = reshape(Hintrp.Hz,size(Z));
```

Plot the resulting magnetic field.

```
figure
quiver3(X,Y,Z,HintrpX,HintrpY,HintrpZ,"Color","red")
view([30 10])

view([10 15])
```



Input Arguments

`magnetostaticresults` — Solution of magnetostatic problem
`MagnetostaticResults` object

Solution of a magnetostatic problem, specified as a `MagnetostaticResults` object. Create `magnetostaticresults` using the `solve` function.

Example: `magnetostaticresults = solve(emagmodel)`

`xq` — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `interpolateMagneticField` evaluates the magnetic field at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateMagneticField` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns magnetic field values as 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 `HintrpX = reshape(Hintrp.Hx,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateMagneticField` evaluates the magnetic field at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateMagneticField` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns magnetic field values as 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 `HintrpY = reshape(Hintrp.Hy,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateMagneticField` evaluates the magnetic field at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateMagneticField` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns magnetic field values as 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 `HintrpZ = reshape(Hintrp.Hz,size(zq))`.

Example: `zq = [1 1 0 1.5]`

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. `interpolateMagneticField` evaluates magnetic field at the coordinate points `querypoints(:,i)` for every `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`Hintrp` — Magnetic field at query points

`FEStruct`

Magnetic field at query points, returned as an `FEStruct` object with the properties representing the spatial components of the magnetic field at the query points. For query points that are outside the geometry, `Hintrp.Hx(i)`, `Hintrp.Hy(i)`, and `Hintrp.Hz(i)` are `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2021a

See Also

`solve` | `interpolateMagneticFlux` | `interpolateMagneticPotential` | `ElectromagneticModel` | `MagnetostaticResults`

interpolateMagneticFlux

Package: `pde`

Interpolate magnetic flux density in magnetostatic result at arbitrary spatial locations

Syntax

```
Bintrp = interpolateMagneticFlux(magnetostaticresults,xq,yq)
Bintrp = interpolateMagneticFlux(magnetostaticresults,xq,yq,zq)
Bintrp = interpolateMagneticFlux(magnetostaticresults,querypoints)
```

Description

`Bintrp = interpolateMagneticFlux(magnetostaticresults,xq,yq)` returns the interpolated magnetic flux density at the 2-D points specified in `xq` and `yq`.

`Bintrp = interpolateMagneticFlux(magnetostaticresults,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`Bintrp = interpolateMagneticFlux(magnetostaticresults,querypoints)` returns the interpolated magnetic flux density at the points specified in `querypoints`.

Examples

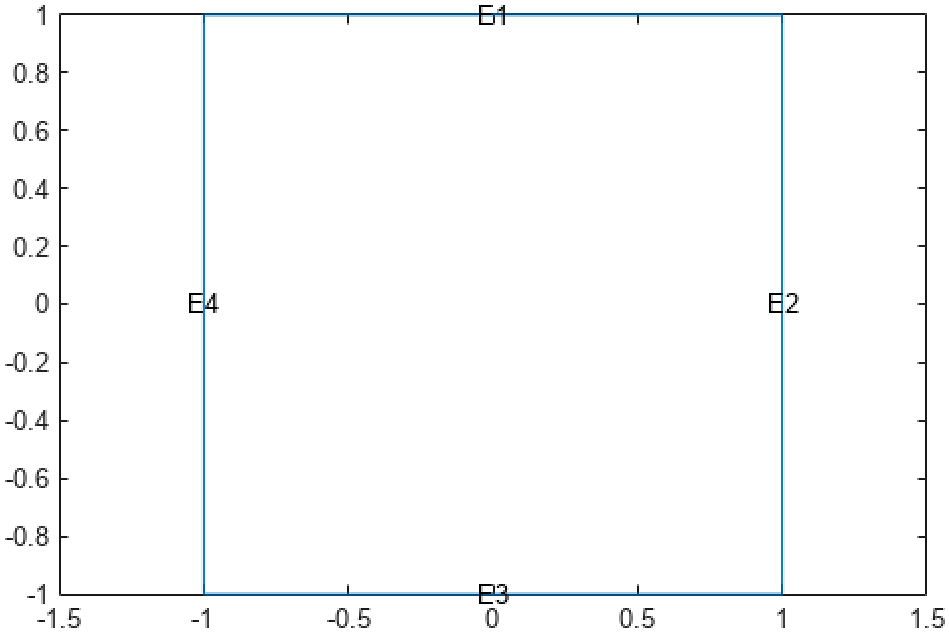
Interpolate Magnetic Flux Density in 2-D Magnetostatic Analysis

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Create a square geometry and include it in the model. Plot the geometry with the edge labels.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1,'R1',('R1')');
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-1.5 1.5])
axis equal
```



Specify the vacuum permeability in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the boundaries of the square.

```
electromagneticBC(emagmodel,"MagneticPotential",0,"Edge",[1 3]);
electromagneticBC(emagmodel,"MagneticPotential",0.01,"Edge",[2 4]);
```

Specify the current density for the entire geometry.

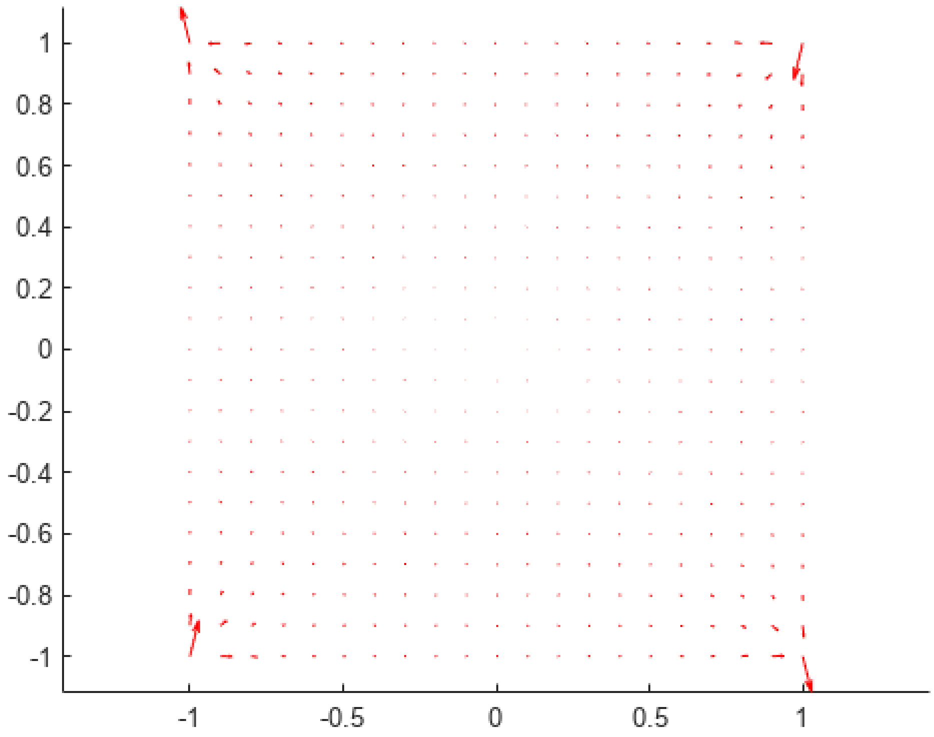
```
electromagneticSource(emagmodel,"CurrentDensity",0.5);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model and plot the magnetic flux density.

```
R = solve(emagmodel);
pdeplot(emagmodel,"FlowData",[R.MagneticFluxDensity.Bx ...
                              R.MagneticFluxDensity.By])
axis equal
```



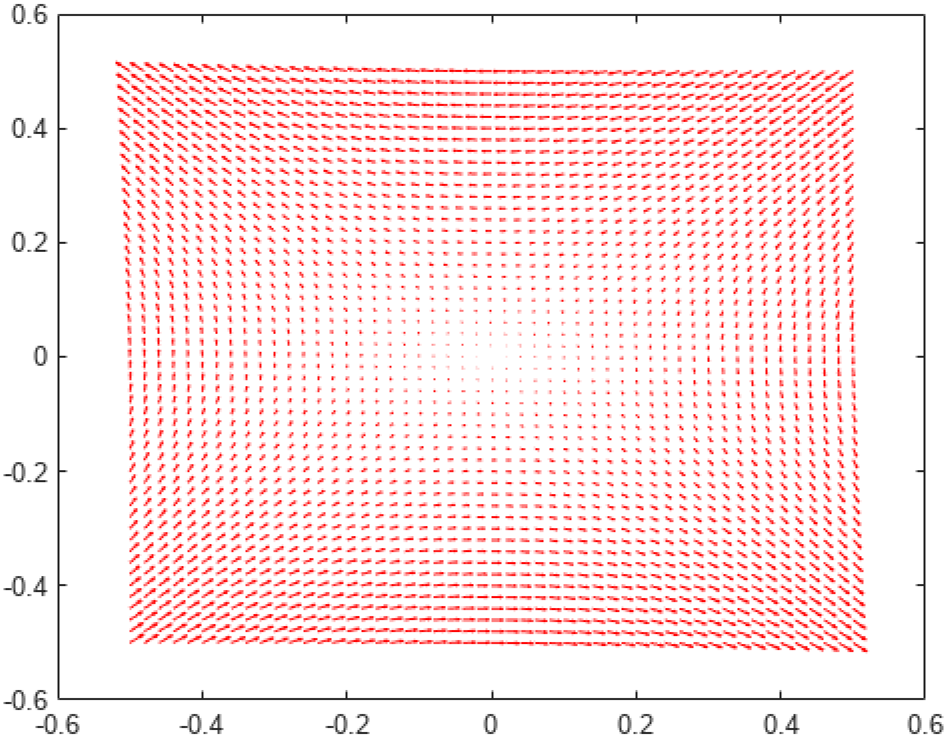
Interpolate the resulting electric flux density to a grid covering the central portion of the geometry, for `x` and `y` from `-0.5` to `0.5`.

```
v = linspace(-0.5,0.5,51);
[X,Y] = meshgrid(v);
Bintrp = interpolateMagneticFlux(R,X,Y)
```

```
Bintrp = 
  FEStruct with properties:

    Bx: [2601x1 double]
    By: [2601x1 double]
```

Reshape `Bintrp.Bx` and `Bintrp.By` and plot the resulting magnetic flux density.

```
BintrpX = reshape(Bintrp.Bx,size(X));
BintrpY = reshape(Bintrp.By,size(Y));
figure
quiver(X,Y,BintrpX,BintrpY,"Color","red")
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Bintrp = interpolateMagneticFlux(R,querypoints);
```

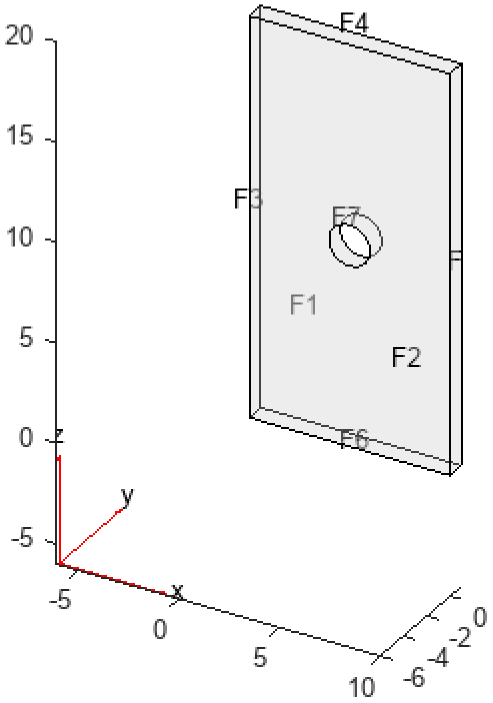
Interpolate Magnetic Flux Density in 3-D Magnetostatic Analysis

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Specify the current density for the entire geometry.

```
electromagneticSource(emagmodel,"CurrentDensity",[0;0;0.5]);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Generate a mesh.

```
generateMesh(emagmodel);
```

Solve the model.

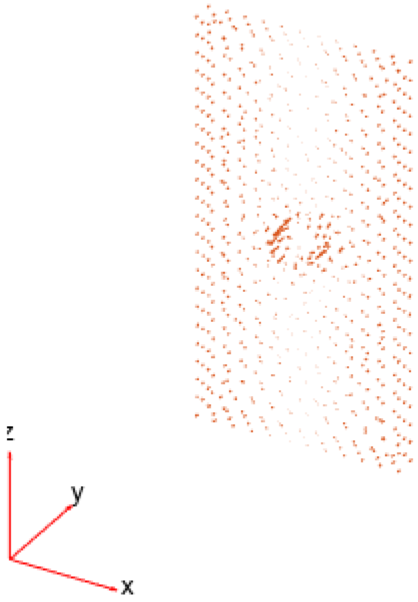
```
R = solve(emagmodel)
```

```
R = 
  MagnetostaticResults with properties:
```

```
      MagneticPotential: [1x1 FEStruct]
          MagneticField: [1x1 FEStruct]
    MagneticFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

Plot the magnetic flux density.

```
pdeplot3D(emagmodel,"FlowData",[R.MagneticFluxDensity.Bx ...
                                R.MagneticFluxDensity.By ...
                                R.MagneticFluxDensity.Bz])
```



Interpolate the resulting magnetic flux density to a grid covering the central portion of the geometry, for `x`, `y`, and `z`.

```
x = linspace(3,7,5);
y = linspace(0,1,5);
z = linspace(8,12,5);
[X,Y,Z] = meshgrid(x,y,z);
Bintrp = interpolateMagneticFlux(R,X,Y,Z)
```

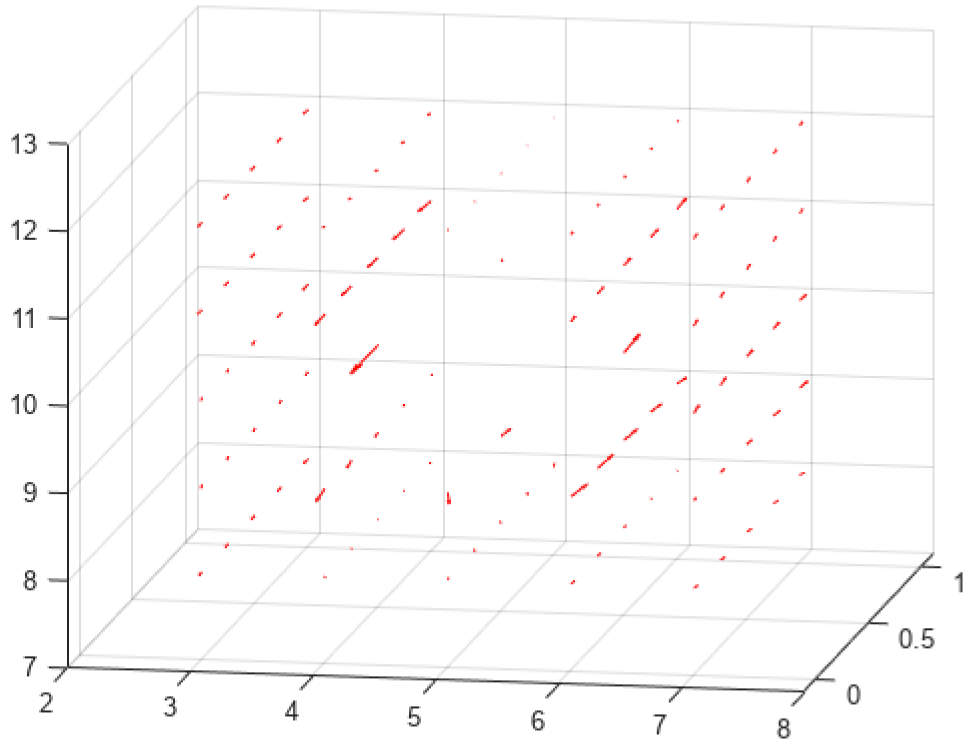
```
Bintrp = 
  FEStruct with properties:

    Bx: [125x1 double]
    By: [125x1 double]
    Bz: [125x1 double]
```

Reshape `Bintrp.Bx`, `Bintrp.By`, and `Bintrp.Bz`.

```
BintrpX = reshape(Bintrp.Bx,size(X));
BintrpY = reshape(Bintrp.By,size(Y));
BintrpZ = reshape(Bintrp.Bz,size(Z));
```

Plot the resulting magnetic flux density.

```
figure
quiver3(X,Y,Z,BintrpX,BintrpY,BintrpZ,"Color","red")
view([30 10])

view([10 15])
```



Input Arguments

`magnetostaticresults` — Solution of magnetostatic problem

`MagnetostaticResults` object

Solution of a magnetostatic problem, specified as a `MagnetostaticResults` object. Create `magnetostaticresults` using the `solve` function.

Example: `magnetostaticresults = solve(emagmodel)`

`xq` — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateMagneticFlux` evaluates the magnetic flux density at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateMagneticFlux` converts the query points to column vectors `xq(:)` and `yq(:)`. It returns magnetic flux density as 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 `BintrpX = reshape(Bintrp.Bx,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateMagneticFlux` evaluates the magnetic flux density at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateMagneticFlux` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns magnetic flux density as 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 `BintrpY = reshape(Bintrp.By,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateMagneticFlux` evaluates the magnetic flux density at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateMagneticFlux` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns magnetic flux density values as 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 `BintrpZ = reshape(Bintrp.Bz,size(zq))`.

Example: `zq = [1 1 0 1.5]`

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. `interpolateMagneticFlux` evaluates the magnetic flux density at the coordinate points `querypoints(:,i)` for every `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`Bintrp` — Magnetic flux density at query points
`FEStruct`

Magnetic flux density at query points, returned as an `FEStruct` object with the properties representing the spatial components of the magnetic flux density at the query points. For query points that are outside the geometry, `Bintrp.Bx(i)`, `Bintrp.By(i)`, and `Bintrp.Bz(i)` are `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2021a

See Also

`solve` | `interpolateMagneticField` | `interpolateMagneticPotential` | `ElectromagneticModel` | `MagnetostaticResults`

interpolateMagneticPotential

Package: `pde`

Interpolate magnetic potential in magnetostatic result at arbitrary spatial locations

Syntax

```
Aintrp = interpolateMagneticPotential(magnetostaticresults,xq,yq)
Aintrp = interpolateMagneticPotential(magnetostaticresults,xq,yq,zq)
Aintrp = interpolateMagneticPotential(magnetostaticresults,querypoints)
```

Description

`Aintrp = interpolateMagneticPotential(magnetostaticresults,xq,yq)` returns the interpolated magnetic potential values at the 2-D points specified in `xq` and `yq`. .

`Aintrp = interpolateMagneticPotential(magnetostaticresults,xq,yq,zq)` uses 3-D points specified in `xq`, `yq`, and `zq`.

`Aintrp = interpolateMagneticPotential(magnetostaticresults,querypoints)` returns the interpolated magnetic potential values at the points specified in `querypoints`.

Examples

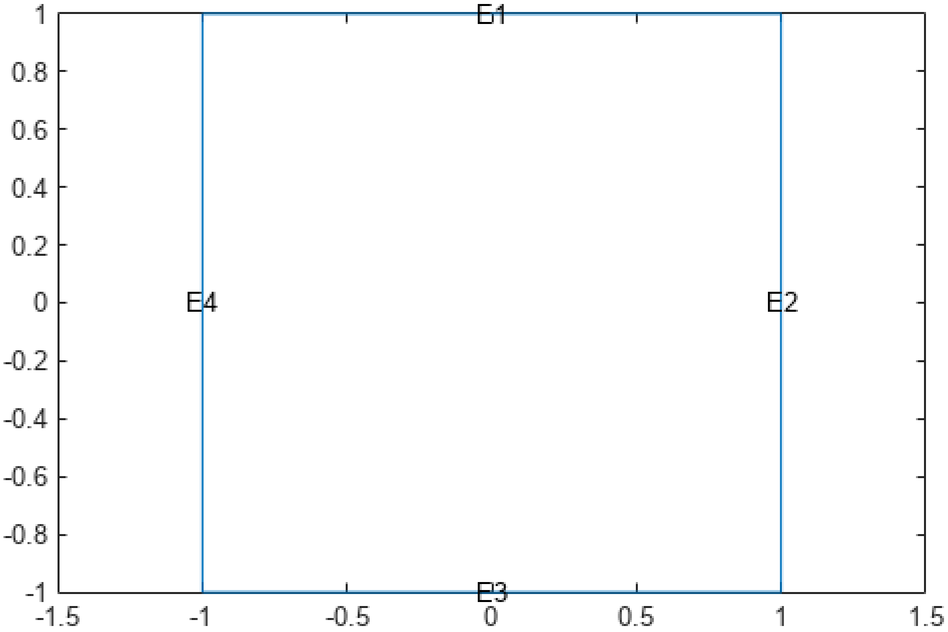
Interpolate Magnetic Potential in 2-D Magnetostatic Analysis

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Create a square geometry and include it in the model. Plot the geometry with the edge labels.

```
R1 = [3,4,-1,1,1,-1,1,1,-1,-1]';
g = decsg(R1,'R1',('R1')');
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-1.5 1.5])
axis equal
```



Specify the vacuum permeability in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the boundaries of the square.

```
electromagneticBC(emagmodel,"MagneticPotential",0,"Edge",[1 3]);
electromagneticBC(emagmodel,"MagneticPotential",0.01,"Edge",[2 4]);
```

Specify the current density for the entire geometry.

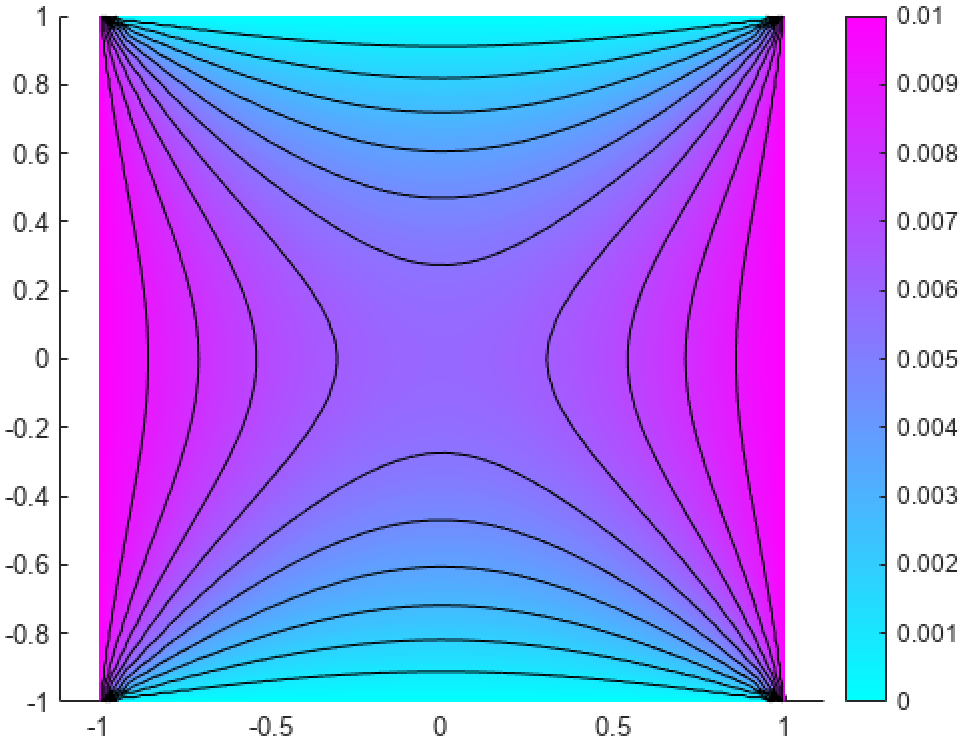
```
electromagneticSource(emagmodel,"CurrentDensity",0.5);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model and plot the magnetic potential.

```
R = solve(emagmodel);
pdeplot(emagmodel,"XYData",R.MagneticPotential, ...
                  "Contour","on")
axis equal
```



Interpolate the resulting magnetic potential to a grid covering the central portion of the geometry, for `x` and `y` from `-0.5` to `0.5`.

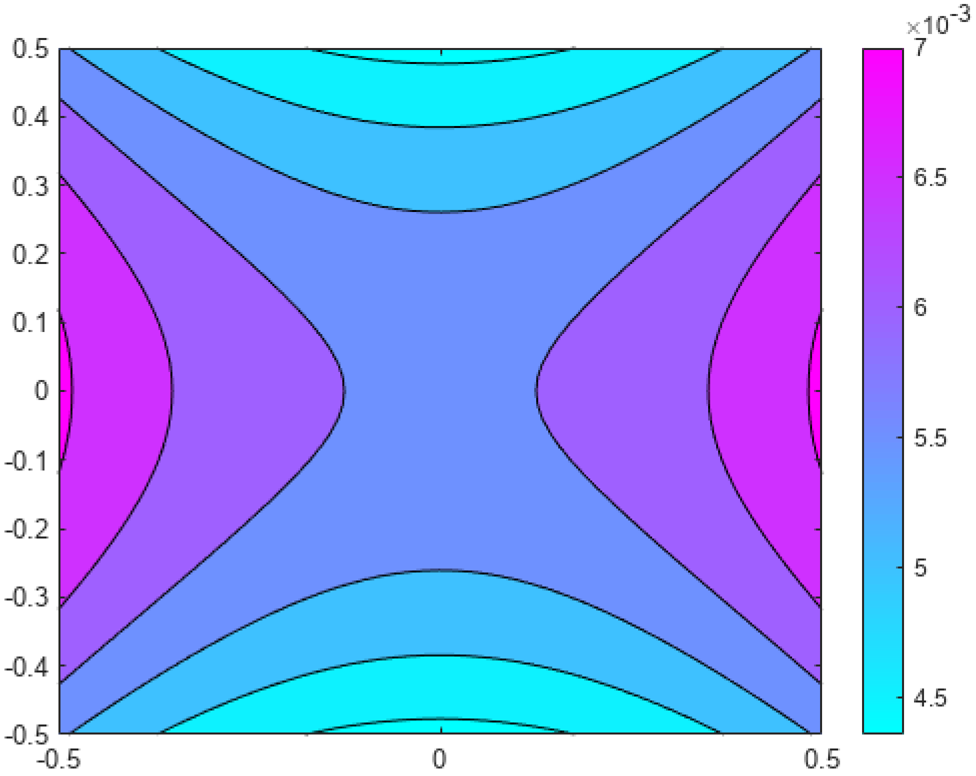
```
v = linspace(-0.5,0.5,51);
[X,Y] = meshgrid(v);
Aintrp = interpolateMagneticPotential(R,X,Y)
```

```
Aintrp = 2601×1

    0.0056
    0.0057
    0.0058
    0.0059
    0.0060
    0.0061
    0.0062
    0.0063
    0.0064
    0.0065
      ⋮
```

Reshape `Aintrp` and plot the resulting magnetic potential.

```
Aintrp = reshape(Aintrp,size(X));
figure
contourf(X,Y,Aintrp)
```

```
colormap(cool)
colorbar
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:)]';
Aintrp = interpolateMagneticPotential(R,querypoints);
```

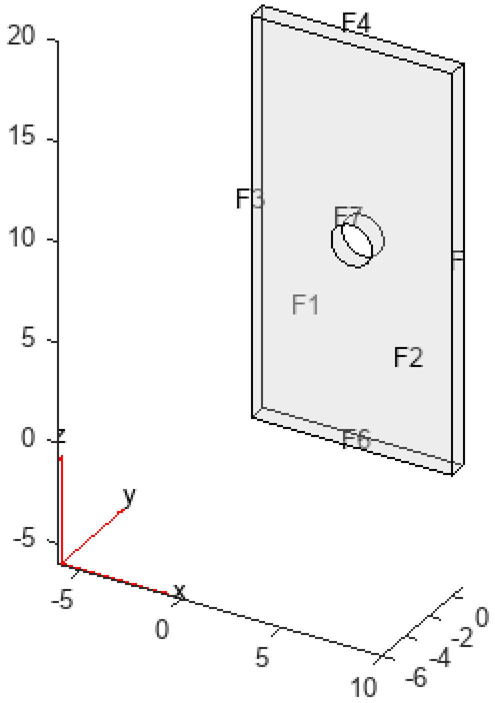
Interpolate Magnetic Potential in 3-D Magnetostatic Analysis

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614E-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Specify the current density for the entire geometry.

```
electromagneticSource(emagmodel,"CurrentDensity",[0;0;0.5]);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Generate the linear mesh.

```
generateMesh(emagmodel);
```

Solve the model.

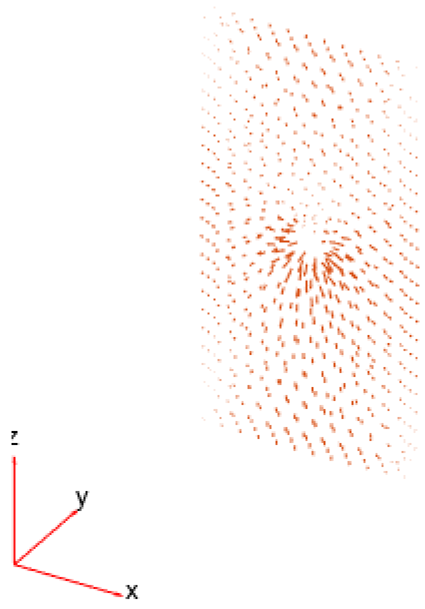
```
R = solve(emagmodel)
```

```
R = 
  MagnetostaticResults with properties:
```

```
      MagneticPotential: [1x1 FEStruct]
          MagneticField: [1x1 FEStruct]
    MagneticFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

Plot the magnetic potential.

```
pdeplot3D(emagmodel,"FlowData",[R.MagneticPotential.Ax ...
                                R.MagneticPotential.Ay ...
                                R.MagneticPotential.Az])
```



Interpolate the resulting magnetic potential to a grid covering the entire geometry, for `x`, `y`, and `z`.

```
x = linspace(0,10,11);
y = linspace(0,1,5);
z = linspace(0,20,11);
[X,Y,Z] = meshgrid(x,y,z);
Aintrp = interpolateMagneticPotential(R,X,Y,Z)
```

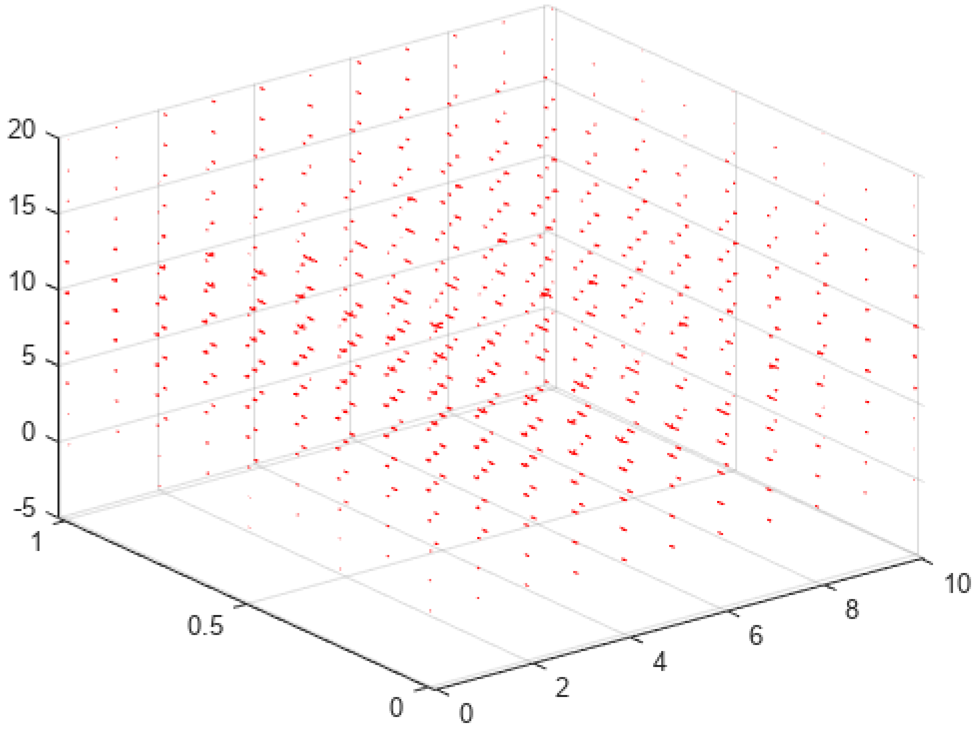
```
Aintrp = 
  FEStruct with properties:

    Ax: [605x1 double]
    Ay: [605x1 double]
    Az: [605x1 double]
```

Reshape `Aintrp.Ax`, `Aintrp.Ay`, and `Aintrp.Az` to match the shape of the input grid.

```
AintrpX = reshape(Aintrp.Ax,size(X));
AintrpY = reshape(Aintrp.Ay,size(Y));
AintrpZ = reshape(Aintrp.Az,size(Z));
```

Plot the resulting magnetic potential.

```
figure
quiver3(X,Y,Z,AintrpX,AintrpY,AintrpZ,"Color","red")
```



Input Arguments

`magnetostaticresults` — Solution of magnetostatic problem

`MagnetostaticResults` object

Solution of a magnetostatic problem, specified as a `MagnetostaticResults` object. Create `magnetostaticresults` using the `solve` function.

Example: `magnetostaticresults = solve(emagmodel)`

`xq` — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateMagneticPotential` evaluates the magnetic potential at the 2-D coordinate points `[xq(i) yq(i)]` or at the 3-D coordinate points `[xq(i) yq(i) zq(i)]` for every `i`. Because of this, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateMagneticPotential` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns magnetic potential values as 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 `Aintrp = reshape(Aintrp,size(xq))`.

Example: `xq = [0.5 0.5 0.75 0.75]`

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateMagneticPotential` evaluates the magnetic potential at the coordinate points `[xq(i),yq(i)]` for every `i`. Because of this, `xq` and `yq` must have the same number of entries.

`interpolateMagneticPotential` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns magnetic potential values as 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 `Aintrp = reshape(Aintrp,size(yq))`.

Example: `yq = [1 2 0 0.5]`

Data Types: `double`

zq — *z*-coordinate query points
real array

z-coordinate query points, specified as a real array. `interpolateMagneticPotential` evaluates the magnetic potential at the 3-D coordinate points `[xq(i) yq(i) zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries.

`interpolateMagneticPotential` converts the query points to column vectors `xq(:)`, `yq(:)`, and `zq(:)`. It returns magnetic potential values as 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 `Aintrp = reshape(Aintrp,size(zq))`.

Example: `zq = [1 1 0 1.5]`

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. `interpolateMagneticPotential` evaluates the magnetic potential at the coordinate points `querypoints(:,i)` for every `i`, so each column of `querypoints` contains exactly one 2-D or 3-D query point.

Example: For a 2-D geometry, `querypoints = [0.5 0.5 0.75 0.75; 1 2 0 0.5]`

Data Types: `double`

Output Arguments

`Aintrp` — Magnetic potential at query points
vector | `FEStruct` object

Magnetic potential at query points, returned as a vector for a 2-D problem or an `FEStruct` object for a 3-D problem. The properties of `FEStruct` contain the components of the magnetic potential at query points. For query points `i` that are outside the geometry, `Aintrp(i)`, `Aintrp.Ax(i)`, `Aintrp.Ay(i)`, and `Aintrp.Az(i)` are `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2021a

See Also

`solve` | `interpolateMagneticField` | `interpolateMagneticFlux` | `ElectromagneticModel` | `MagnetostaticResults`

interpolateSolution

Package: `pde`

Interpolate PDE solution to arbitrary points

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.

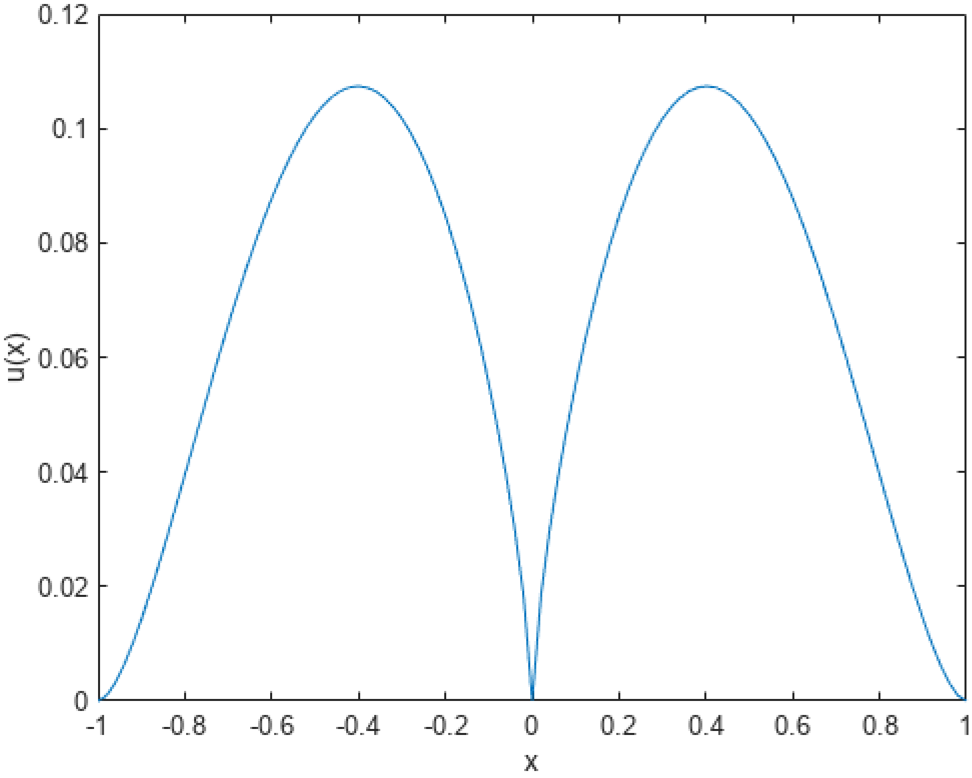
Create the solution to the problem $-\Delta u = 1$ on the L-shaped membrane with zero Dirichlet boundary conditions.

```
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.

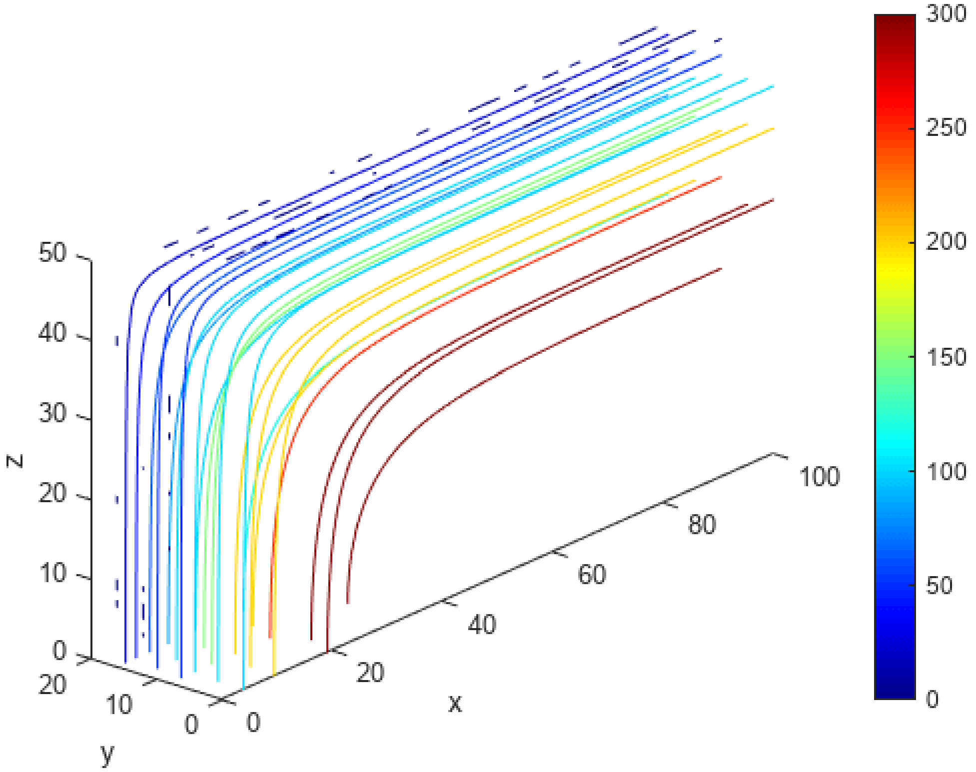
```
model = createpde;
importGeometry(model,"Block.stl");
applyBoundaryCondition(model,"dirichlet","Face",[1,2,5],"u",0);
specifyCoefficients(model,"m",0,...
                          "d",0,...
                          "c",1,...
                          "a",0,...
                          "f",2);
generateMesh(model);
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.

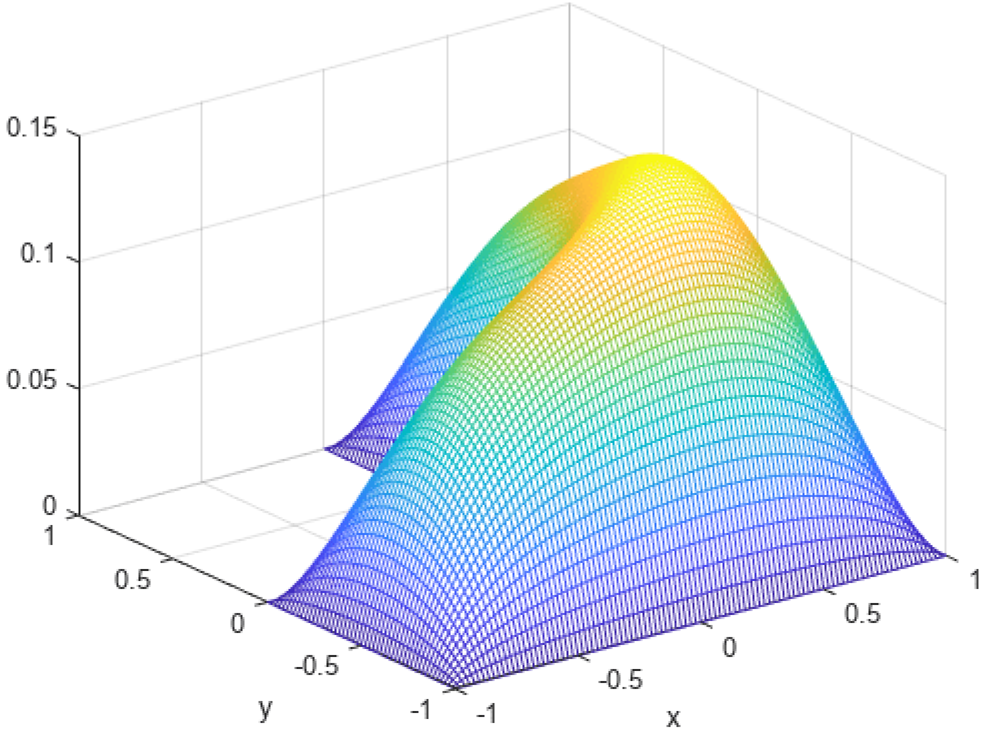
```
model = createpde;
geometryFromEdges(model,@lshapeg);
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1:model.Geometry.NumEdges, ...
                             "u",0);
specifyCoefficients(model,"m",0,"d",0,"c",1,"a",0,"f",1);
generateMesh(model,"Hmax",0.05);
results = 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")
```

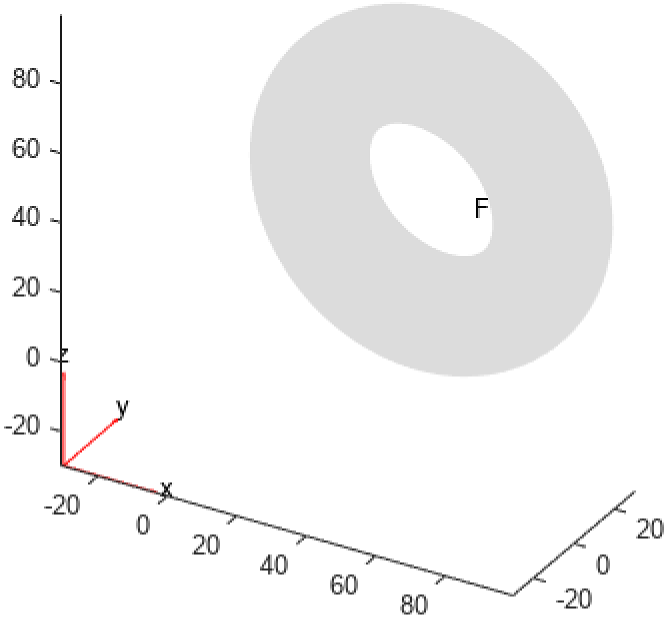


Interpolate Stationary System

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.

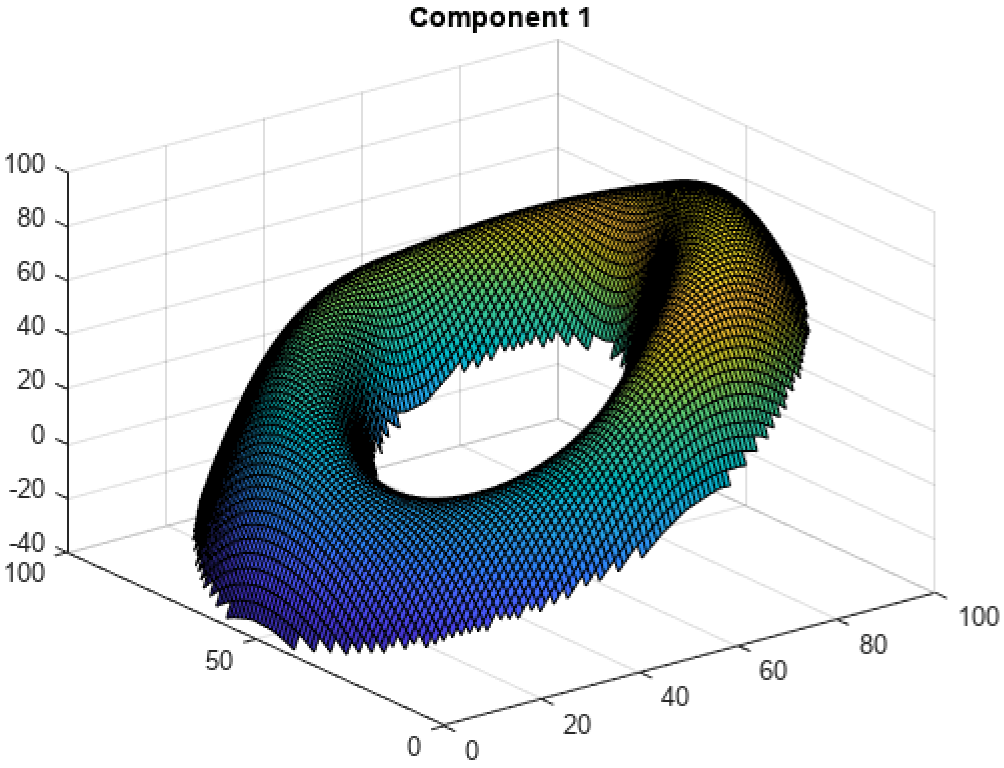
```
generateMesh(model);
results = solvepde(model);
```

Interpolate the results on a plane that slices the torus for each of the two components.

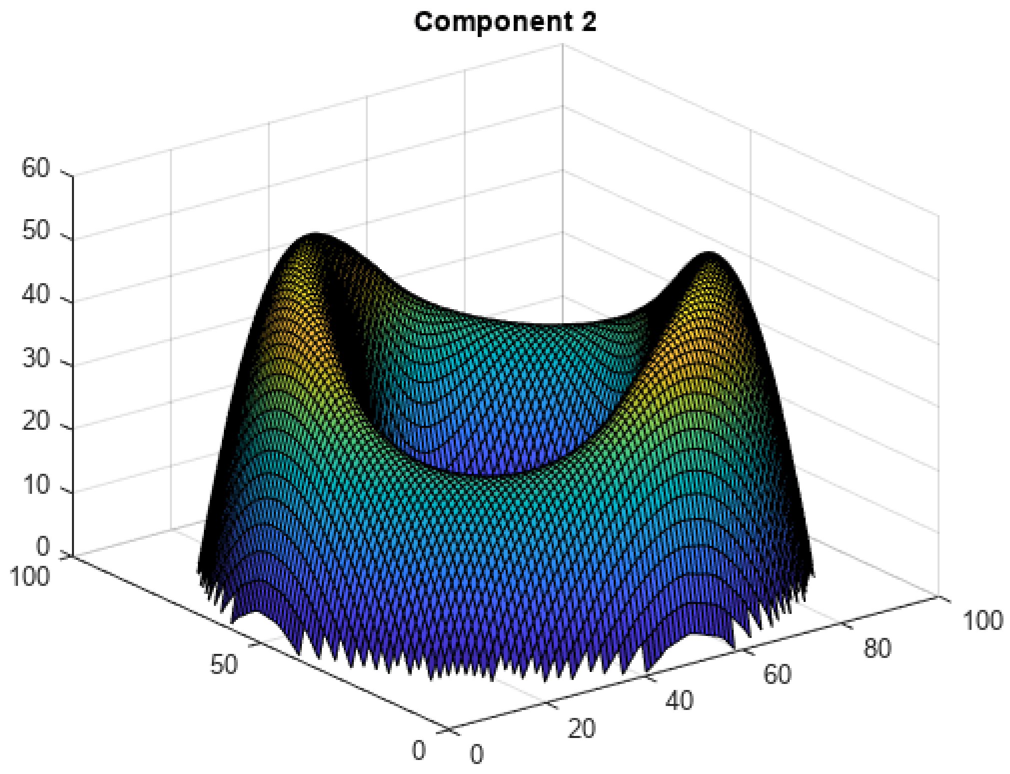
```
[X,Z] = meshgrid(0:100);
Y = 15*ones(size(X));
uintrp = interpolateSolution(results,X,Y,Z,[1,2]);
```

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")
```



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);
```

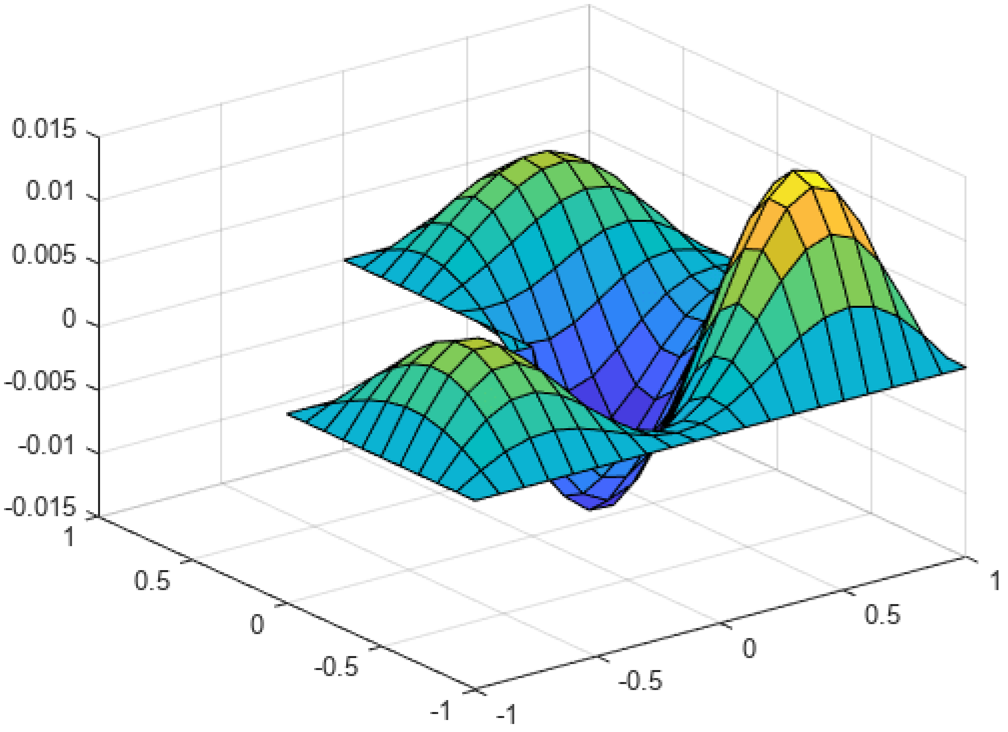
```
              Basis= 10,  Time=   1.73,  New conv eig=  0
              Basis= 11,  Time=   1.95,  New conv eig=  0
              Basis= 12,  Time=   2.03,  New conv eig=  0
              Basis= 13,  Time=   2.09,  New conv eig=  0
              Basis= 14,  Time=   2.12,  New conv eig=  0
```

```
Basis= 15,  Time=   2.19,  New conv eig=  0
Basis= 16,  Time=   2.36,  New conv eig=  0
Basis= 17,  Time=   2.42,  New conv eig=  0
Basis= 18,  Time=   2.47,  New conv eig=  1
Basis= 19,  Time=   2.52,  New conv eig=  1
Basis= 20,  Time=   2.53,  New conv eig=  1
Basis= 21,  Time=   2.62,  New conv eig=  2
Basis= 22,  Time=   2.81,  New conv eig=  2
Basis= 23,  Time=   2.92,  New conv eig=  3
Basis= 24,  Time=   2.94,  New conv eig=  3
Basis= 25,  Time=   3.19,  New conv eig=  3
Basis= 26,  Time=   3.20,  New conv eig=  3
Basis= 27,  Time=   3.33,  New conv eig=  3
Basis= 28,  Time=   3.41,  New conv eig=  3
Basis= 29,  Time=   3.47,  New conv eig=  5
Basis= 30,  Time=   3.53,  New conv eig=  5
Basis= 31,  Time=   3.58,  New conv eig=  5
Basis= 32,  Time=   3.66,  New conv eig=  5
Basis= 33,  Time=   3.73,  New conv eig=  5
Basis= 34,  Time=   3.75,  New conv eig=  5
Basis= 35,  Time=   3.88,  New conv eig=  5
Basis= 36,  Time=   3.98,  New conv eig=  6
Basis= 37,  Time=   4.17,  New conv eig=  6
Basis= 38,  Time=   4.25,  New conv eig=  7
Basis= 39,  Time=   4.38,  New conv eig=  7
Basis= 40,  Time=   4.39,  New conv eig=  7
Basis= 41,  Time=   4.50,  New conv eig=  7
Basis= 42,  Time=   4.66,  New conv eig=  7
Basis= 43,  Time=   4.73,  New conv eig=  7
Basis= 44,  Time=   4.81,  New conv eig=  7
Basis= 45,  Time=   4.88,  New conv eig=  7
Basis= 46,  Time=   4.98,  New conv eig=  8
Basis= 47,  Time=   5.08,  New conv eig=  9
Basis= 48,  Time=   5.20,  New conv eig= 10
Basis= 49,  Time=   5.33,  New conv eig= 12
Basis= 50,  Time=   5.41,  New conv eig= 12
Basis= 51,  Time=   5.52,  New conv eig= 13
Basis= 52,  Time=   5.61,  New conv eig= 14
Basis= 53,  Time=   5.73,  New conv eig= 14
Basis= 54,  Time=   5.91,  New conv eig= 14
Basis= 55,  Time=   6.05,  New conv eig= 14
Basis= 56,  Time=   6.12,  New conv eig= 14
Basis= 57,  Time=   6.22,  New conv eig= 14
Basis= 58,  Time=   6.28,  New conv eig= 14
Basis= 59,  Time=   6.36,  New conv eig= 14
Basis= 60,  Time=   6.55,  New conv eig= 14
Basis= 61,  Time=   6.66,  New conv eig= 16
Basis= 62,  Time=   6.69,  New conv eig= 16
Basis= 63,  Time=   6.92,  New conv eig= 16
Basis= 64,  Time=   7.02,  New conv eig= 16
Basis= 65,  Time=   7.11,  New conv eig= 16
Basis= 66,  Time=   7.17,  New conv eig= 17
Basis= 67,  Time=   7.38,  New conv eig= 18
Basis= 68,  Time=   7.42,  New conv eig= 18
Basis= 69,  Time=   7.47,  New conv eig= 20
Basis= 70,  Time=   7.55,  New conv eig= 20
Basis= 71,  Time=   7.59,  New conv eig= 20
Basis= 72,  Time=   7.69,  New conv eig= 20
```

```
               Basis= 73,  Time=   7.73,  New conv eig= 20
               Basis= 74,  Time=   7.94,  New conv eig= 20
               Basis= 75,  Time=   8.08,  New conv eig= 21
               Basis= 76,  Time=   8.23,  New conv eig= 21
               Basis= 77,  Time=   8.41,  New conv eig= 22
End of sweep: Basis= 77,  Time=   8.44,  New conv eig= 22
               Basis= 32,  Time=   9.19,  New conv eig=  0
               Basis= 33,  Time=   9.31,  New conv eig=  0
               Basis= 34,  Time=   9.38,  New conv eig=  0
               Basis= 35,  Time=   9.44,  New conv eig=  0
               Basis= 36,  Time=   9.47,  New conv eig=  0
               Basis= 37,  Time=   9.53,  New conv eig=  0
               Basis= 38,  Time=   9.64,  New conv eig=  0
               Basis= 39,  Time=   9.73,  New conv eig=  0
               Basis= 40,  Time=   9.83,  New conv eig=  0
               Basis= 41,  Time=   9.84,  New conv eig=  0
               Basis= 42,  Time=   9.88,  New conv eig=  0
End of sweep: Basis= 42,  Time=   9.89,  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)
```

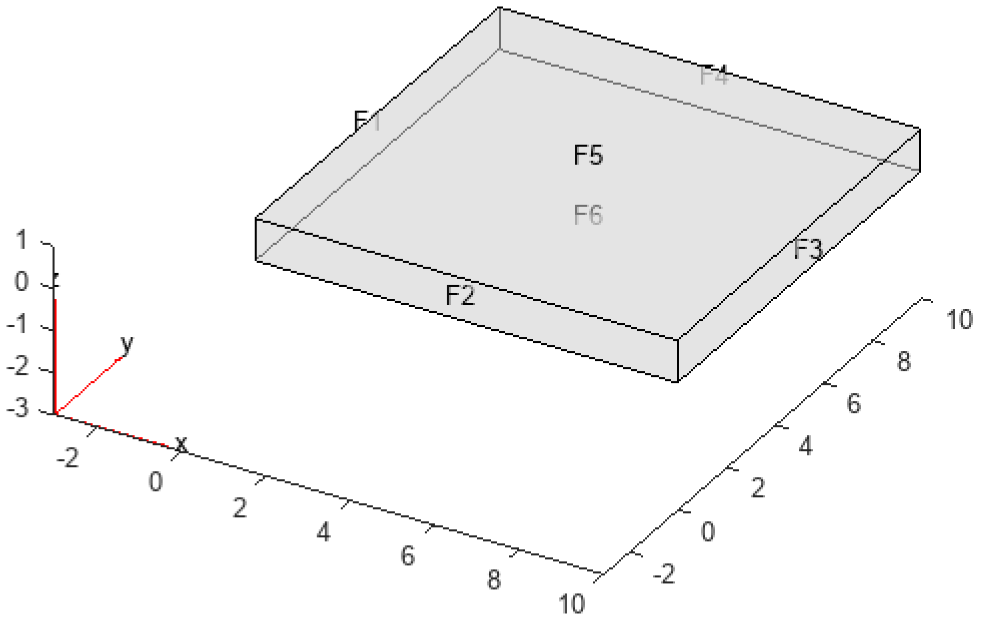


Interpolate Time-Dependent System

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 “Linear Elasticity Equations” on page 3-156.

```
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.

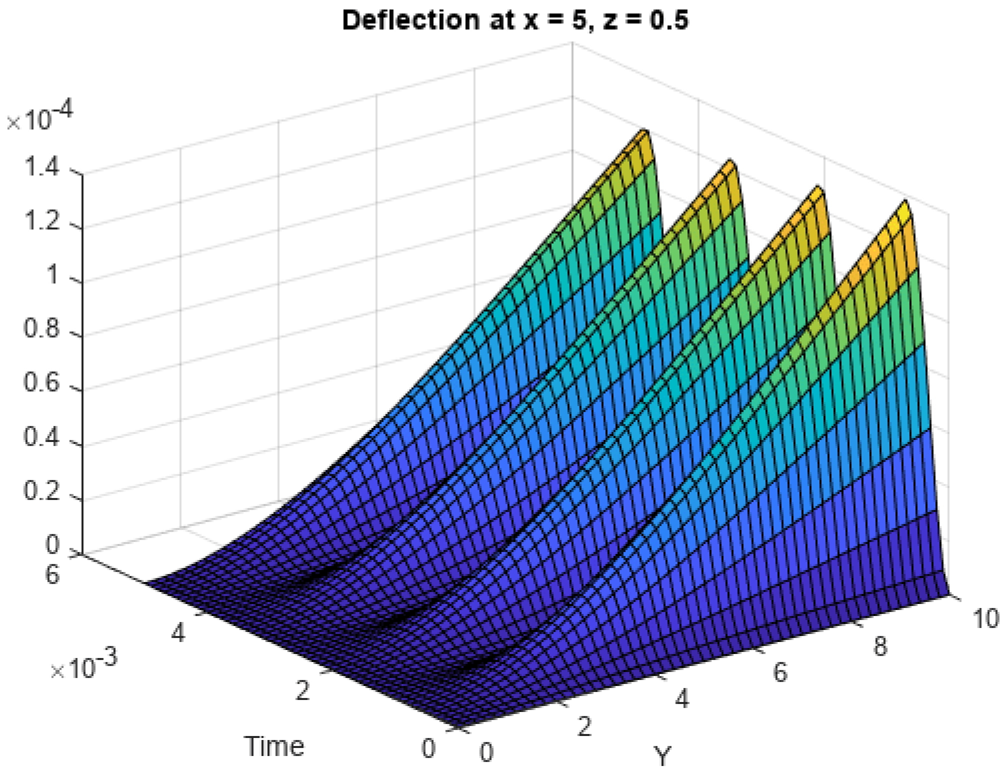
```
yy = 0:0.2:10;
zz = 0.5*ones(size(yy));
xx = 10*zz;
component = 3;
uintrp = interpolateSolution(results,xx,yy,zz, ...
                   component,1:length(tlist));
```

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])
```



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 3-356.

Version History

Introduced in R2015b

See Also

`PDEModel` | `StationaryResults` | `TimeDependentResults` | `evaluateGradient`

Topics

"Solution and Gradient Plots with pdeplot and pdeplot3D" on page 3-329
"3-D Solution and Gradient Plots with MATLAB Functions" on page 3-344
"Dimensions of Solutions, Gradients, and Fluxes" on page 3-356

interpolateStrain

Package: `pde`

Interpolate strain at arbitrary spatial locations

Syntax

```
intrpStrain = interpolateStrain(structuralresults,xq,yq)
intrpStrain = interpolateStrain(structuralresults,xq,yq,zq)
intrpStrain = interpolateStrain(structuralresults,querypoints)
```

Description

`intrpStrain = interpolateStrain(structuralresults,xq,yq)` returns the interpolated strain values at the 2-D points specified in `xq` and `yq`. For transient and frequency-response structural models, `interpolateStrain` interpolates strain for all time- or frequency-steps, respectively.

`intrpStrain = interpolateStrain(structuralresults,xq,yq,zq)` uses the 3-D points specified in `xq`, `yq`, and `zq`.

`intrpStrain = interpolateStrain(structuralresults,querypoints)` uses the points specified in `querypoints`.

Examples

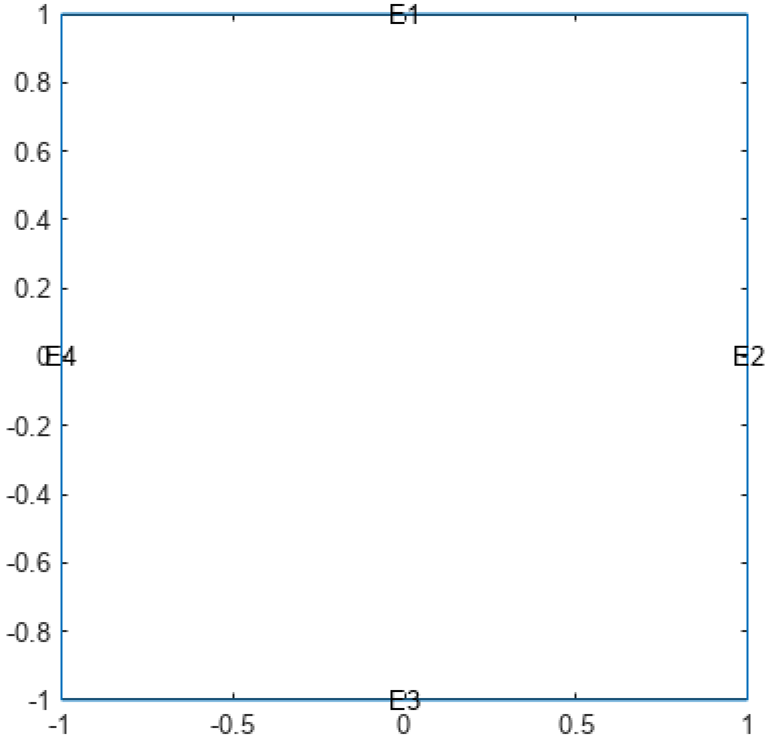
Interpolate Strain for Plane-Strain Problem

Create a structural analysis model for a plane-strain problem.

```
structuralmodel = createpde("structural","static-planestrain");
```

Include the square geometry in the model. Plot the geometry.

```
geometryFromEdges(structuralmodel,@squareg);
pdegplot(structuralmodel,"EdgeLabels","on")
axis equal
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"PoissonsRatio",0.3, ...
                                     "YoungsModulus",210E3);
```

Specify the *x*-component of the enforced displacement for edge 1.

```
structuralBC(structuralmodel,"XDisplacement",0.001,"Edge",1);
```

Specify that edge 3 is a fixed boundary.

```
structuralBC(structuralmodel,"Constraint","fixed","Edge",3);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel);
```

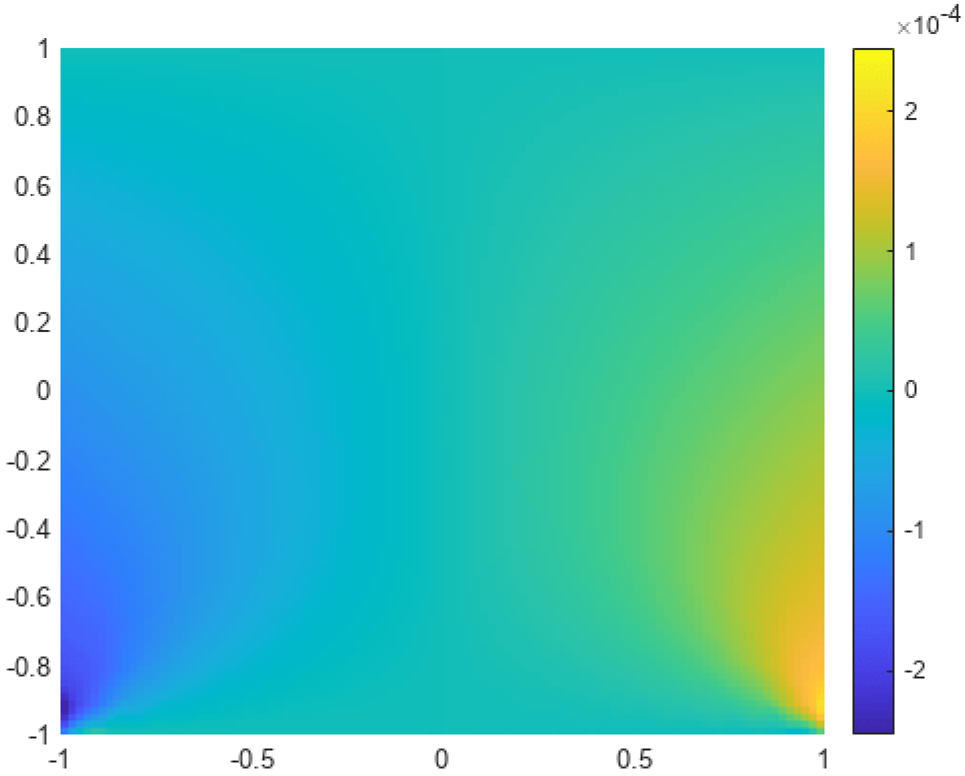
Create a grid and interpolate the *x*- and *y*-components of the normal strain to the grid.

```
v = linspace(-1,1,101);
[X,Y] = meshgrid(v);
intrpStrain = interpolateStrain(structuralresults,X,Y);
```

Reshape the *x*-component of the normal strain to the shape of the grid and plot it.

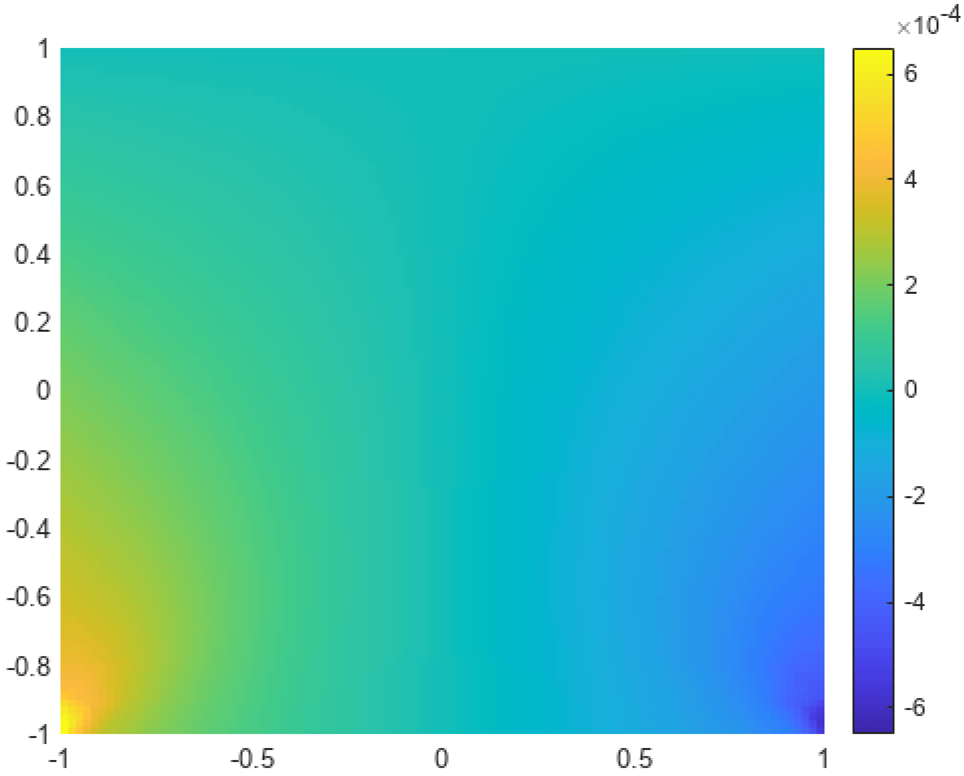
```
exx = reshape(intrpStrain.exx,size(X));
px = pcolor(X,Y,exx);
```

```
px.EdgeColor="none";
colorbar
```



Reshape the *y*-component of the normal strain to the shape of the grid and plot it.

```
eyy = reshape(intrpStrain.eyy,size(Y));
figure
py = pcolor(X,Y,eyy);
py.EdgeColor="none";
colorbar
```



Interpolate Strain for 3-D Static Structural Analysis Problem

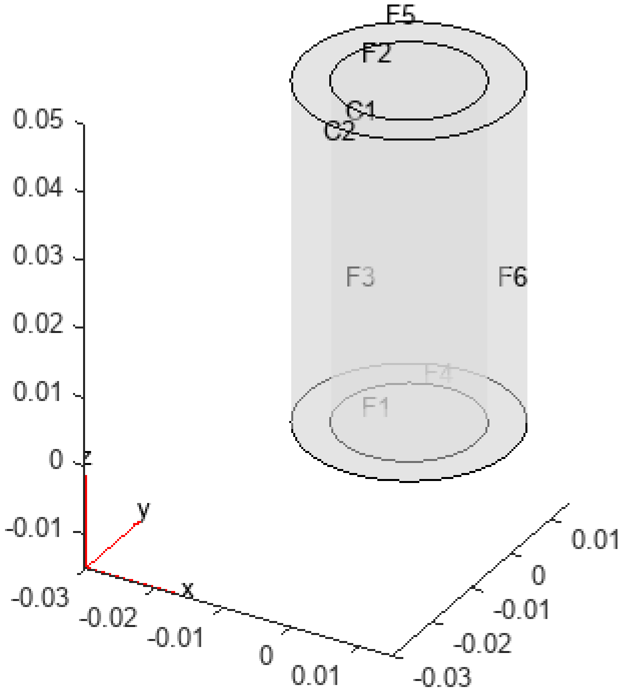
Solve a static structural model representing a bimetallic cable under tension, and interpolate strain on a cross-section of the cable.

Create a static structural model for solving a solid (3-D) problem.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01,0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

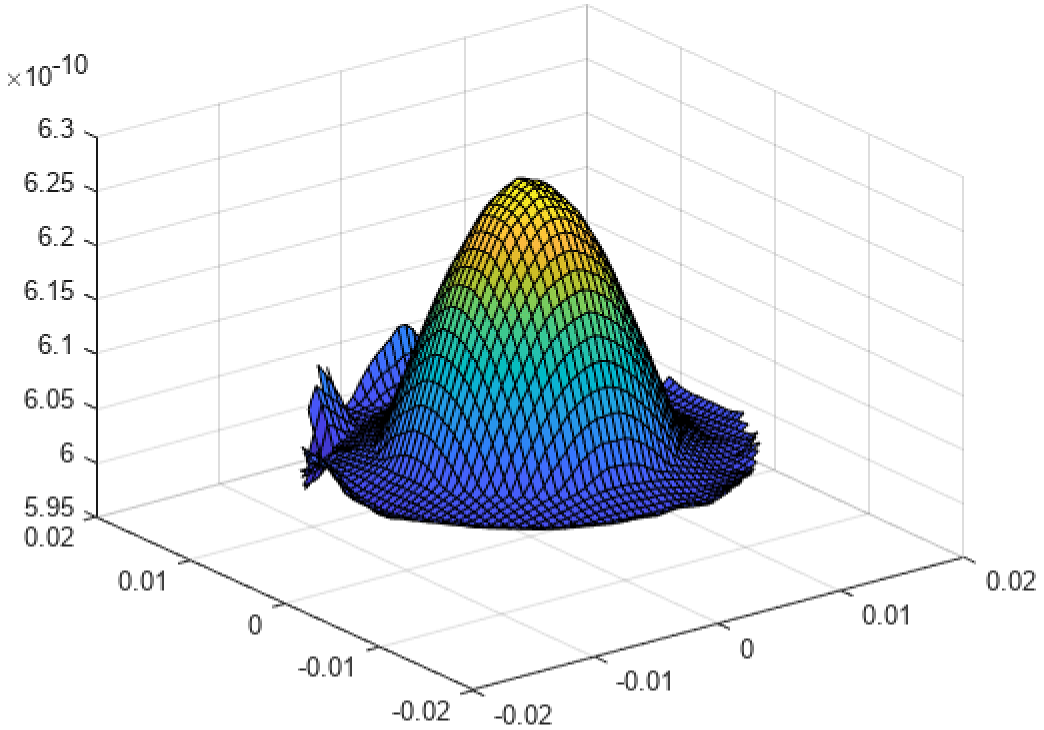
```
            Mesh: [1x1 FEMesh]
```

Define the coordinates of a midspan cross-section of the cable.

```
[X,Y] = meshgrid(linspace(-0.015,0.015,50));
Z = ones(size(X))*0.025;
```

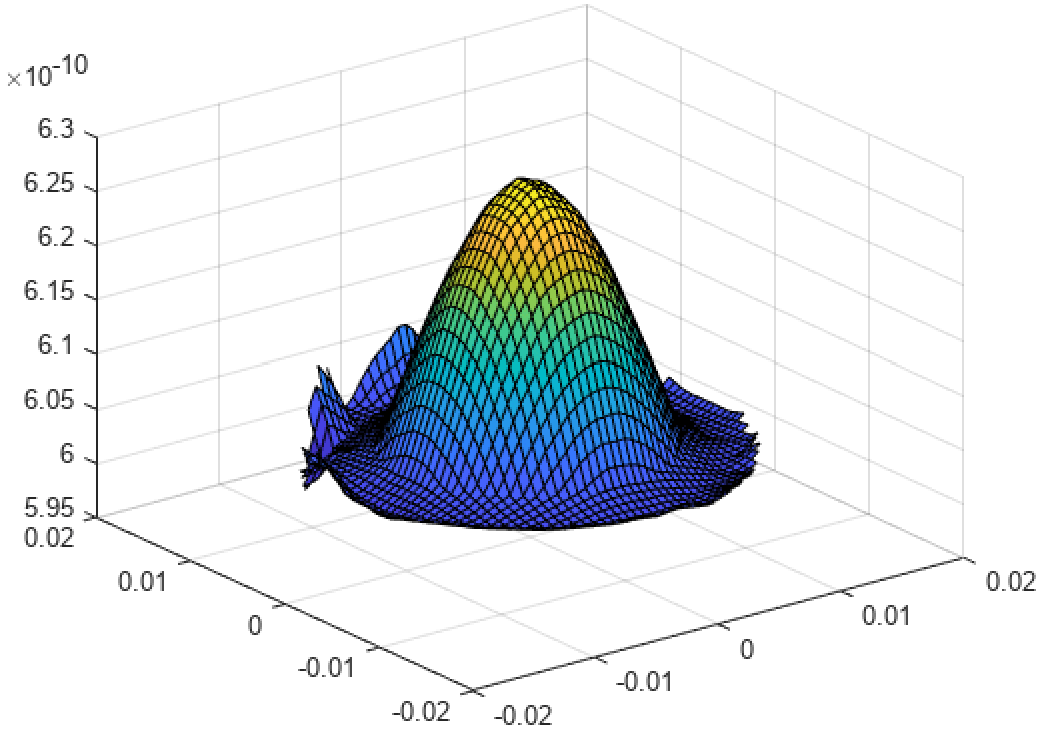
Interpolate the strain and plot the result.

```
intrpStrain = interpolateStrain(structuralresults,X,Y,Z);
surf(X,Y,reshape(intrpStrain.ezz,size(X)))
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:),Z(:)]';
intrpStrain = interpolateStrain(structuralresults,querypoints);
surf(X,Y,reshape(intrpStrain.ezz,size(X)))
```



Interpolate Strain for 3-D Structural Dynamic Problem

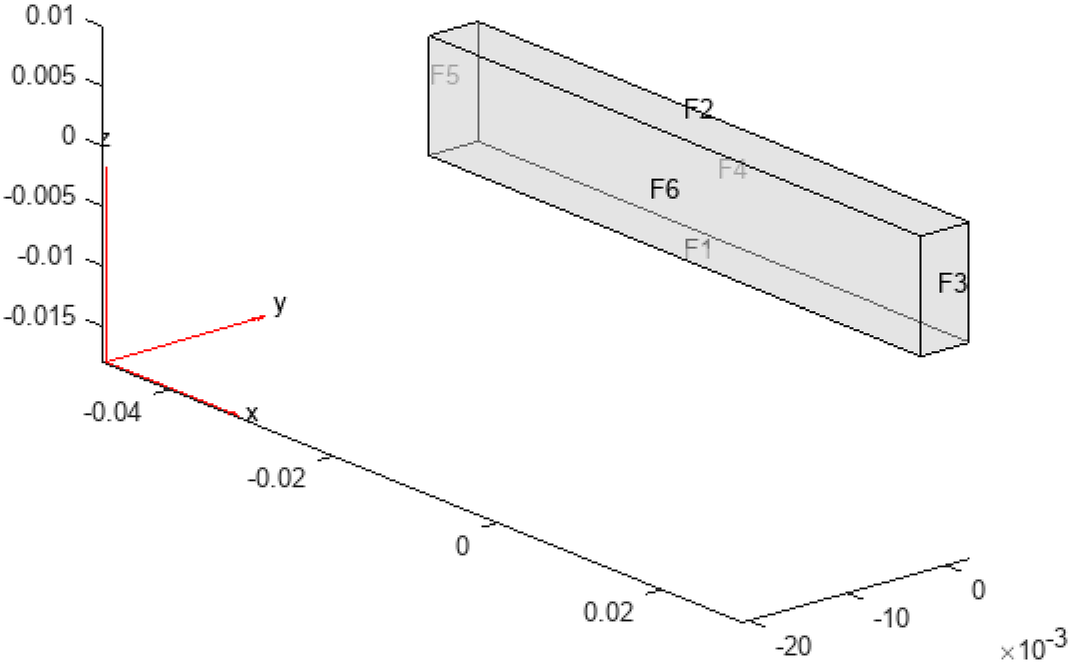
Interpolate the strain at the geometric center of a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the `y`-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0], ...
                             "Velocity",[0;0;0]);
```

Solve the model.

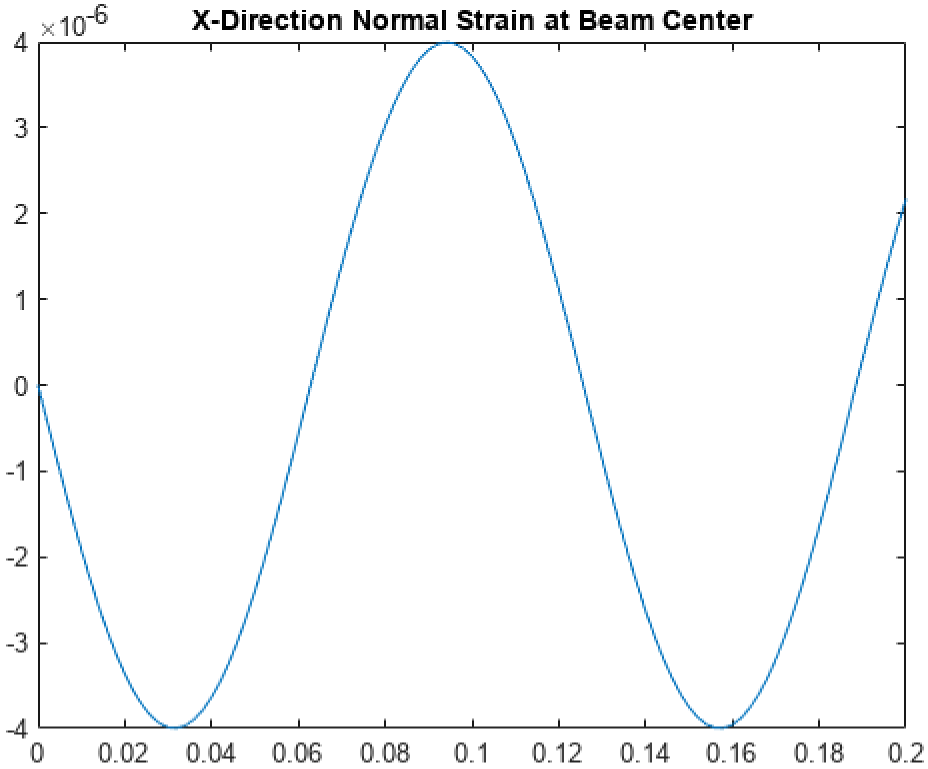
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Interpolate the strain at the geometric center of the beam.

```
coordsMidSpan = [0;0;0.005];
intrpStrain = interpolateStrain(structuralresults,coordsMidSpan);
```

Plot the normal strain at the geometric center of the beam.

```
figure
plot(structuralresults.SolutionTimes,intrpStrain.exx)
title("X-Direction Normal Strain at Beam Center")
```



Input Arguments

`structuralresults` — Solution of structural analysis problem
`StaticStructuralResults` object | `TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the structural analysis problem, specified as a `StaticStructuralResults`, `TransientStructuralResults`, or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel)`

xq — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `interpolateStrain` evaluates the strains at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateStrain` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The function returns strains as an `FEStruct` object with the properties containing vectors of the same size as these column vectors. To ensure that the dimensions of the returned solution are consistent with the dimensions of the original query points, use the `reshape` function. For example, use `intrpStrain = reshape(intrpStrain.exx,size(xq))`.

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateStrain` evaluates the strains at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries. Internally, `interpolateStrain` converts the 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. `interpolateStrain` evaluates the strains at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries. Internally, `interpolateStrain` converts the 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. `interpolateStrain` evaluates the strains 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`

Output Arguments

`intrpStrain` — Strains at query points
`FEStruct` object

Strains at the query points, returned as an `FEStruct` object with the properties representing spatial components of strain at the query points. For query points that are outside the geometry, `intrpStrain` returns `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2017b

Support for frequency response structural problems

For frequency response structural models, `interpolateStrain` interpolates strain for all frequency-steps.

Support for transient structural problems

For transient structural models, `interpolateStrain` interpolates strain for all time-steps.

See Also

`StructuralModel` | `StaticStructuralResults` | `interpolateDisplacement` | `interpolateStress` | `interpolateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

interpolateStress

Package: `pde`

Interpolate stress at arbitrary spatial locations

Syntax

```
intrpStress = interpolateStress(structuralresults,xq,yq)
intrpStress = interpolateStress(structuralresults,xq,yq,zq)
intrpStress = interpolateStress(structuralresults,querypoints)
```

Description

`intrpStress = interpolateStress(structuralresults,xq,yq)` returns the interpolated stress values at the 2-D points specified in `xq` and `yq`. For transient and frequency-response structural models, `interpolateStress` interpolates stress for all time- or frequency-steps, respectively.

`intrpStress = interpolateStress(structuralresults,xq,yq,zq)` uses the 3-D points specified in `xq`, `yq`, and `zq`.

`intrpStress = interpolateStress(structuralresults,querypoints)` uses the points specified in `querypoints`.

Examples

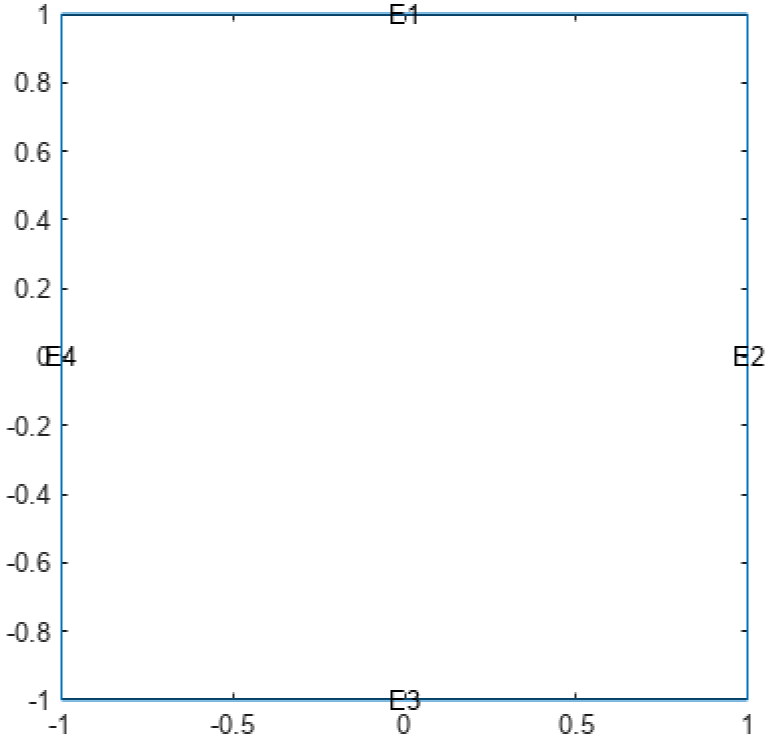
Interpolate Stress for Plane-Strain Problem

Create a structural analysis model for a plane-strain problem.

```
structuralmodel = createpde("structural","static-planestrain");
```

Include the square geometry in the model. Plot the geometry.

```
geometryFromEdges(structuralmodel,@squareg);
pdegplot(structuralmodel,"EdgeLabels","on")
axis equal
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"PoissonsRatio",0.3, ...
                                     "YoungsModulus",210E3);
```

Specify the *x*-component of the enforced displacement for edge 1.

```
structuralBC(structuralmodel,"XDisplacement",0.001,"Edge",1);
```

Specify that edge 3 is a fixed boundary.

```
structuralBC(structuralmodel,"Constraint","fixed","Edge",3);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel);
```

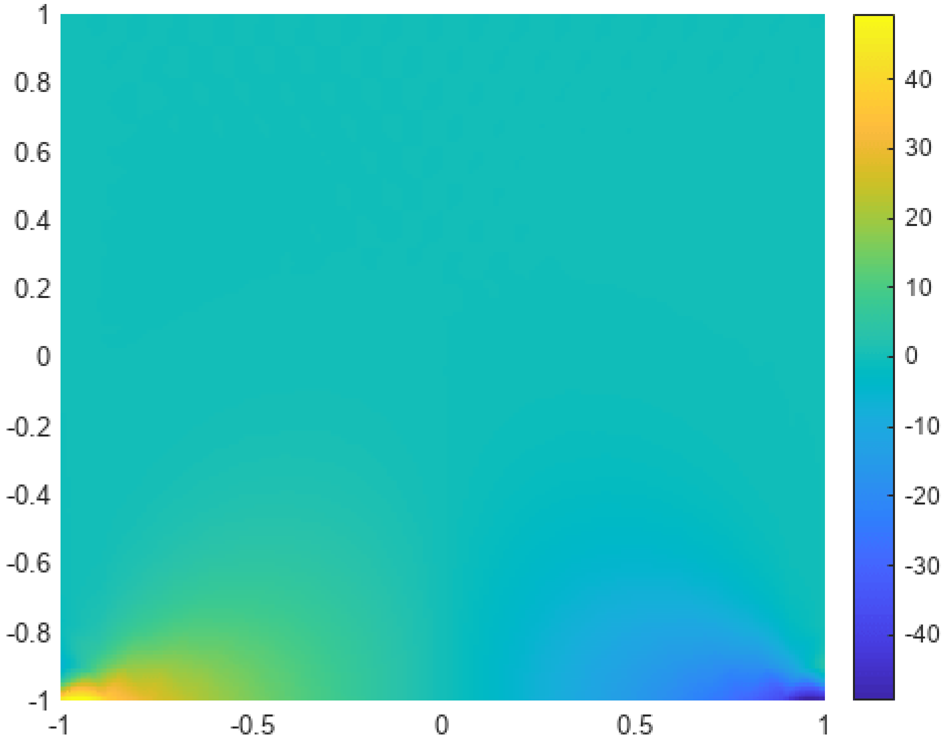
Create a grid and interpolate the *x*- and *y*-components of the normal stress to the grid.

```
v = linspace(-1,1,151);
[X,Y] = meshgrid(v);
intrpStress = interpolateStress(structuralresults,X,Y);
```

Reshape the *x*-component of the normal stress to the shape of the grid and plot it.

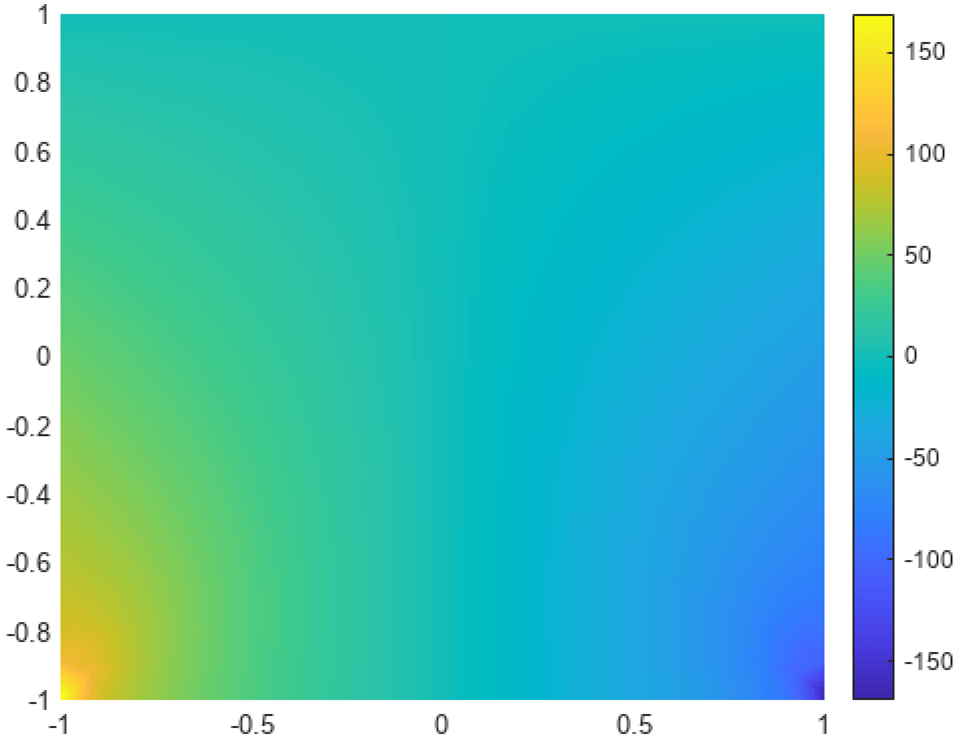
```
sxx = reshape(intrpStress.sxx,size(X));
px = pcolor(X,Y,sxx);
```

```
px.EdgeColor="none";
colorbar
```



Reshape the *y*-component of the normal stress to the shape of the grid and plot it.

```
syy = reshape(intrpStress.syy,size(Y));
figure
py = pcolor(X,Y,syy);
py.EdgeColor="none";
colorbar
```



Interpolate Stress for 3-D Static Structural Analysis Problem

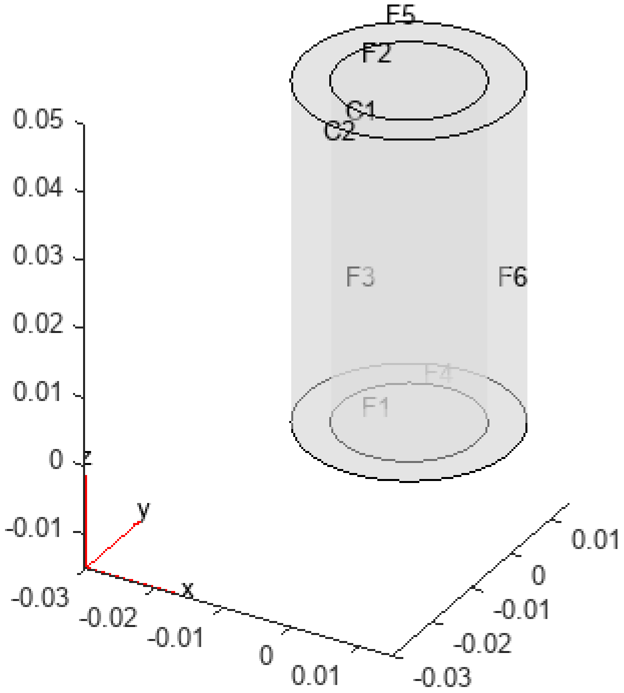
Solve a static structural model representing a bimetallic cable under tension, and interpolate stress on a cross-section of the cable.

Create a static structural model for solving a solid (3-D) problem.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01,0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                             "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                             "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

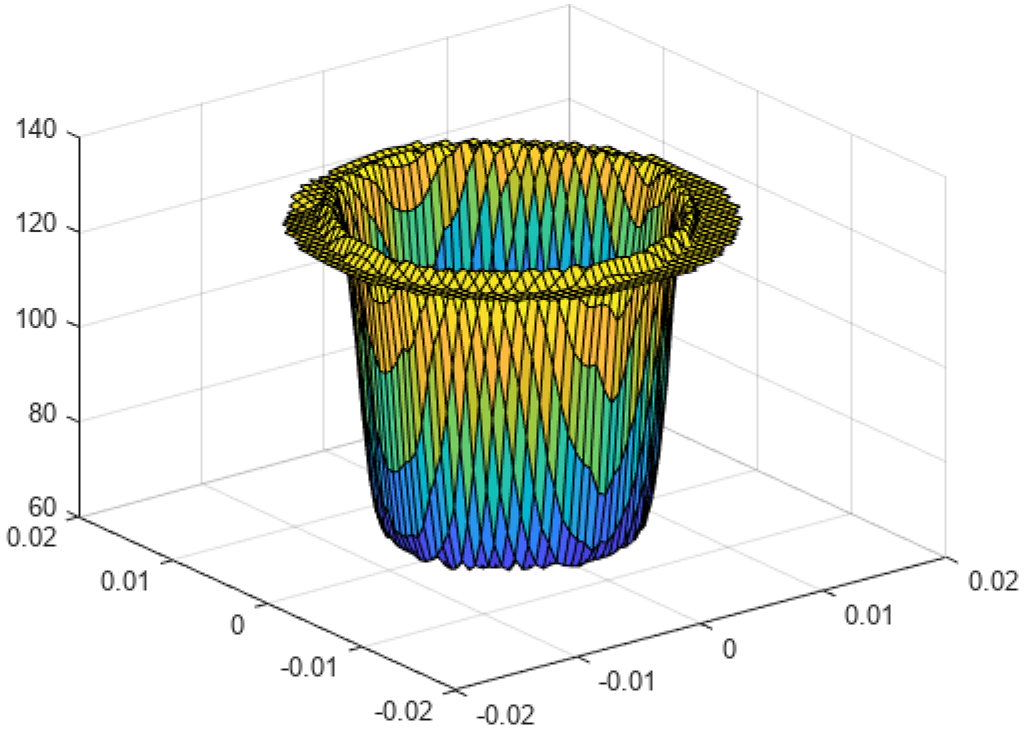
```
            Mesh: [1x1 FEMesh]
```

Define coordinates of a midspan cross-section of the cable.

```
[X,Y] = meshgrid(linspace(-0.015,0.015,50));
Z = ones(size(X))*0.025;
```

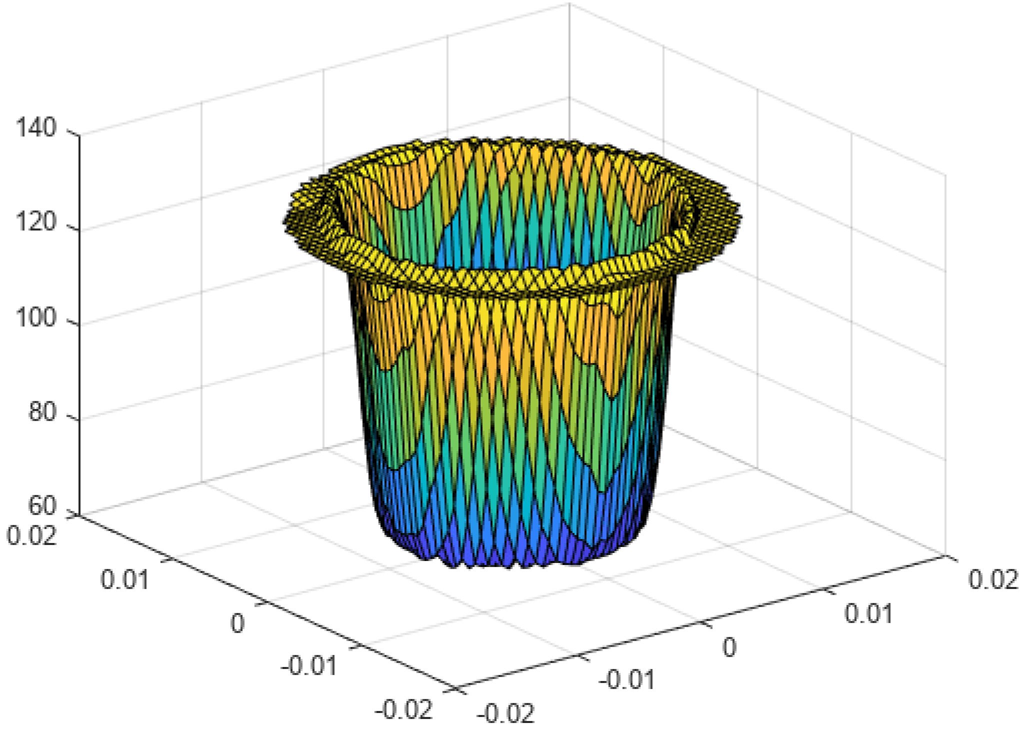
Interpolate the stress and plot the result.

```
intrpStress = interpolateStress(structuralresults,X,Y,Z);
surf(X,Y,reshape(intrpStress.szz,size(X)))
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:),Z(:)]';
intrpStress = interpolateStress(structuralresults,querypoints);
surf(X,Y,reshape(intrpStress.szz,size(X)))
```



Interpolate Stress for 3-D Structural Dynamic Problem

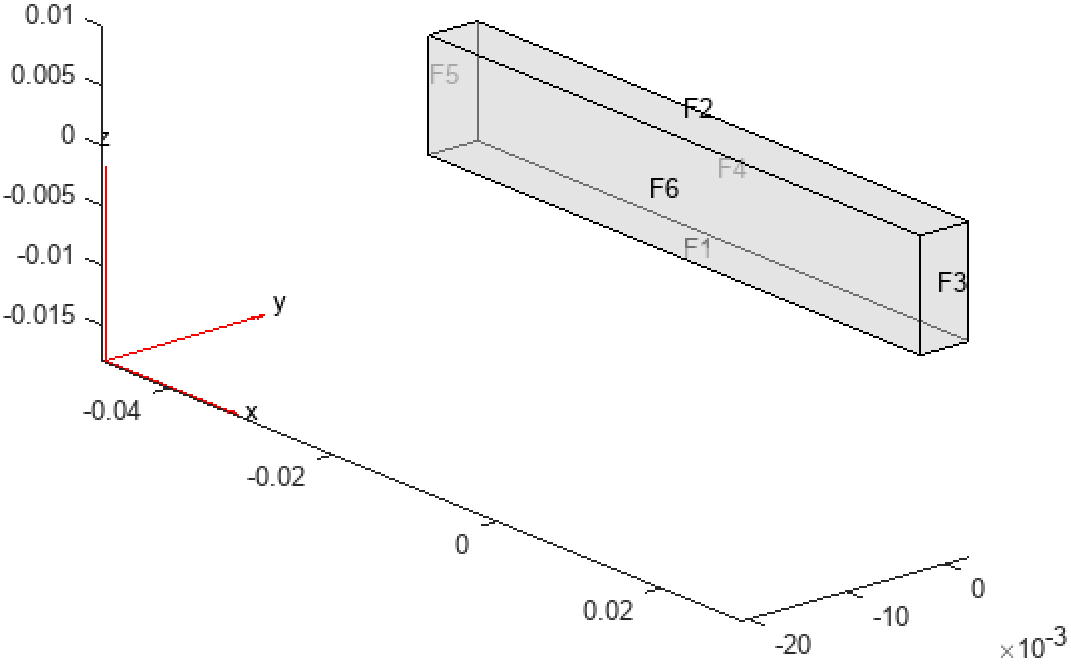
Interpolate the stress at the geometric center of a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the `y`-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0], ...
                             "Velocity",[0;0;0]);
```

Solve the model.

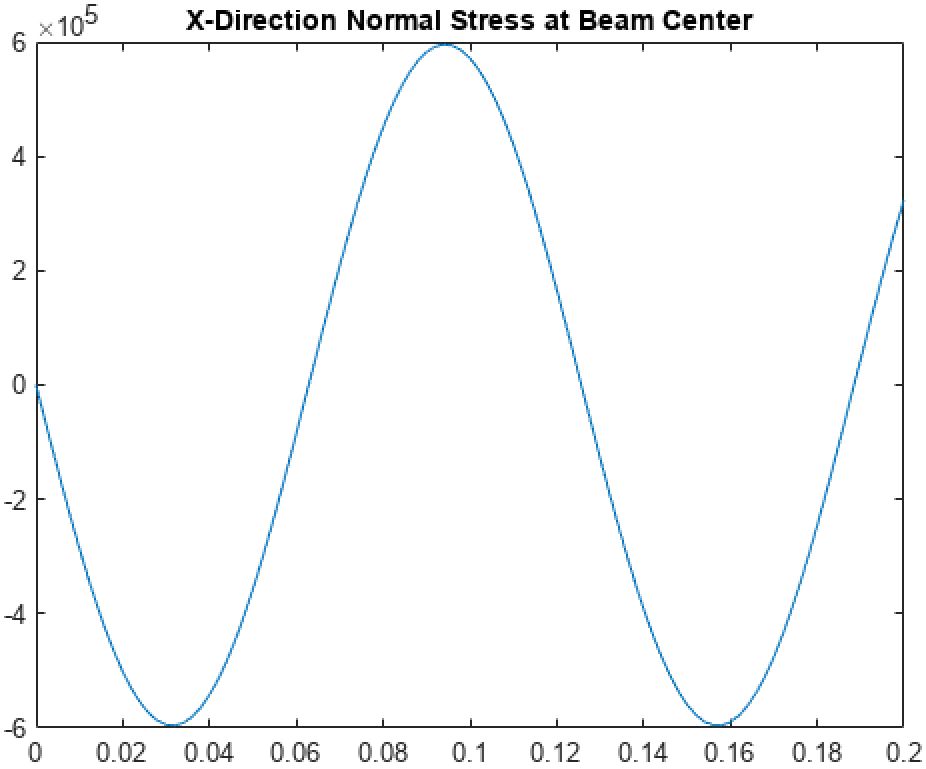
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Interpolate the stress at the geometric center of the beam.

```
coordsMidSpan = [0;0;0.005];
intrpStress = interpolateStress(structuralresults,coordsMidSpan);
```

Plot the normal stress at the geometric center of the beam.

```
figure
plot(structuralresults.SolutionTimes,intrpStress.sxx)
title("X-Direction Normal Stress at Beam Center")
```



Input Arguments

`structuralresults` — Solution of structural analysis problem

`StaticStructuralResults` object | `TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the structural analysis problem, specified as a `StaticStructuralResults`, `TransientStructuralResults`, or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel)`

xq — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateStress` evaluates the stresses at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateStress` converts the query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns stresses as an `FEStruct` object with the properties containing vectors of the same size as these column vectors. To ensure that the dimensions of the returned solution are consistent with the dimensions of the original query points, use the `reshape` function. For example, use `intrpStress = reshape(intrpStress.sxx,size(xq))`.

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateStress` evaluates the stresses at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries. Internally, `interpolateStress` converts the 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. `interpolateStress` evaluates the stresses at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries. Internally, `interpolateStress` converts the 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. `interpolateStress` evaluates stresses 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`

Output Arguments

`intrpStress` — Stresses at query points
`FEStruct` object

Stresses at the query points, returned as an `FEStruct` object with the properties representing spatial components of stress at the query points. For query points that are outside the geometry, `intrpStress` returns `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2017b

Support for frequency response structural problems

For frequency response structural models, `interpolateStress` interpolates stress for all frequency-steps.

Support for transient structural problems

For transient structural models, `interpolateStress` interpolates stress for all time-steps.

See Also

`StructuralModel` | `StaticStructuralResults` | `interpolateDisplacement` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

interpolateTemperature

Package: `pde`

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

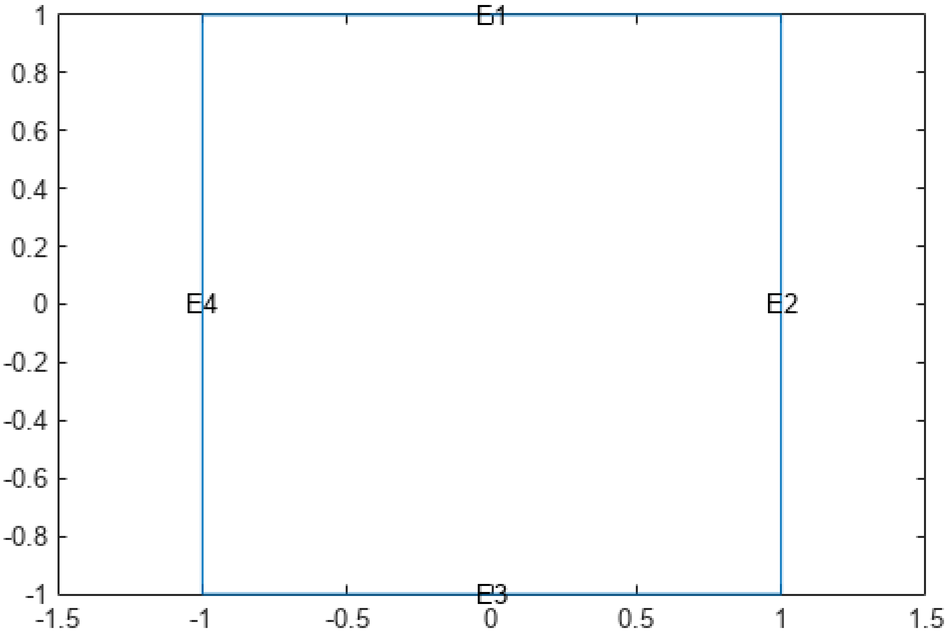
Interpolate Temperatures in 2-D Steady-State Thermal Model

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);
```

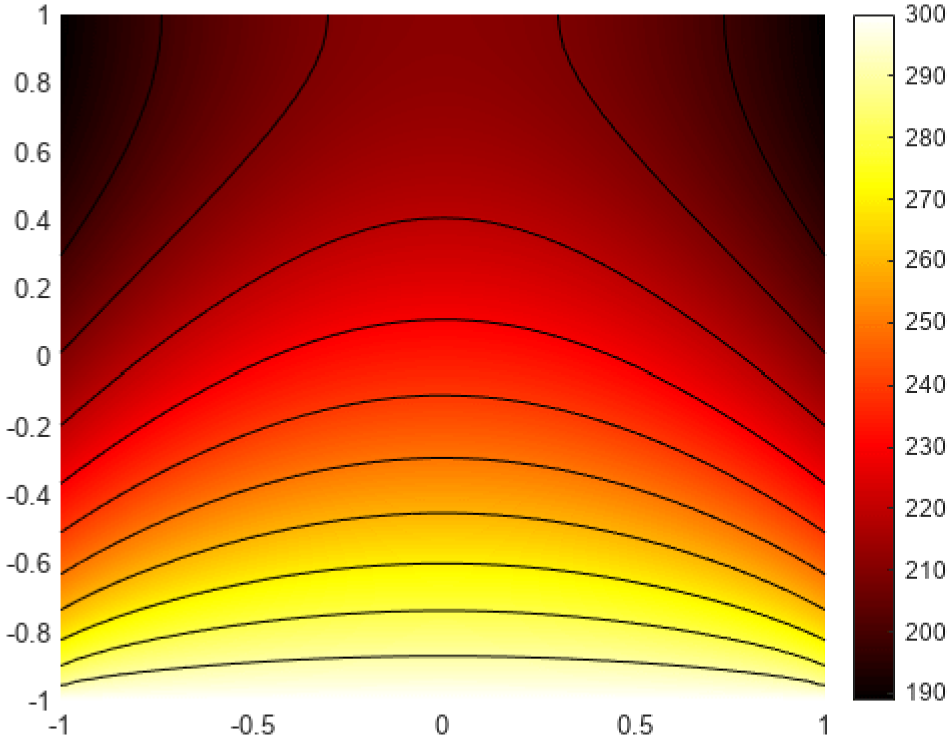
Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
results = solve(thermalmodel)
```

```
results = 
  SteadyStateThermalResults with properties:

    Temperature: [1541x1 double]
     XGradients: [1541x1 double]
     YGradients: [1541x1 double]
     ZGradients: []
           Mesh: [1x1 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");
```

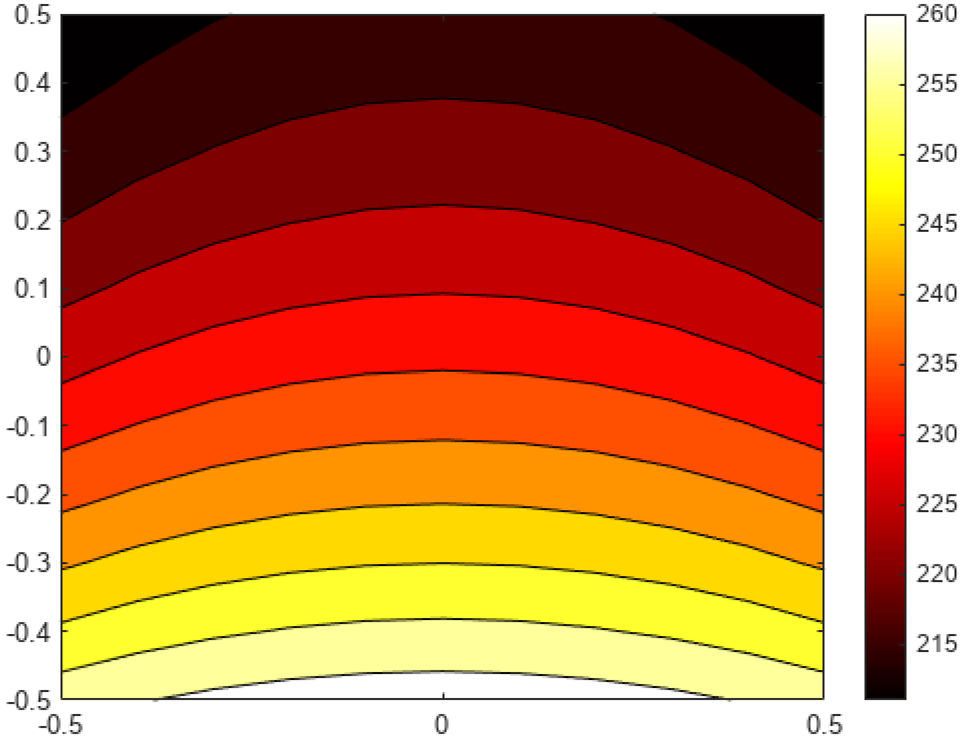


Interpolate the resulting temperatures to a grid covering the central portion of the geometry, for x and y from `-0.5` to `0.5`.

```
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);
```

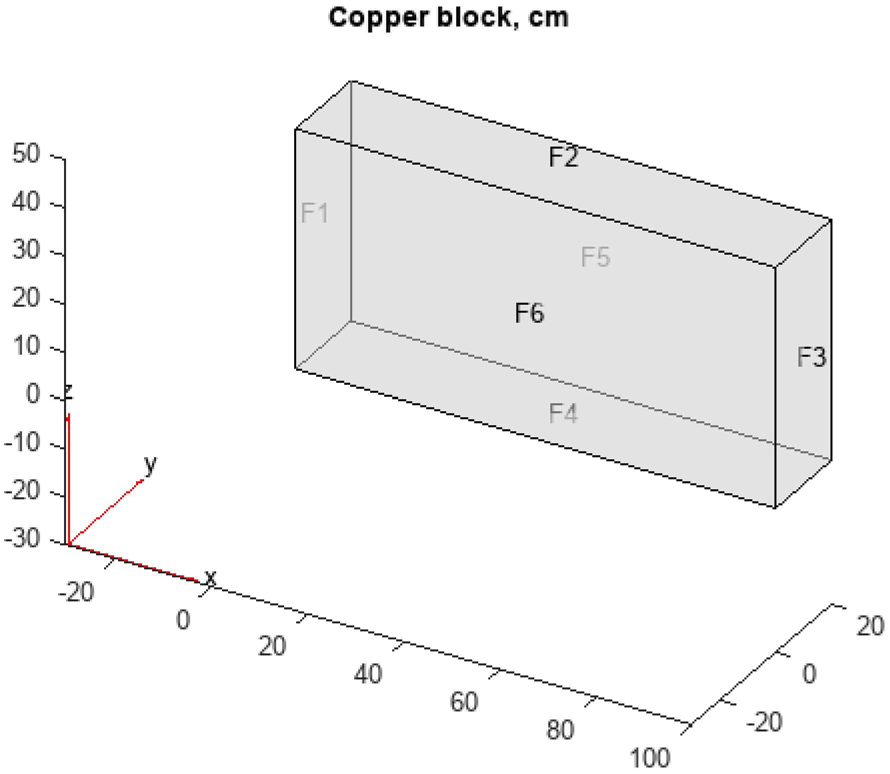
Interpolate Temperature for a 3-D Steady-State Thermal Model

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
```



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.

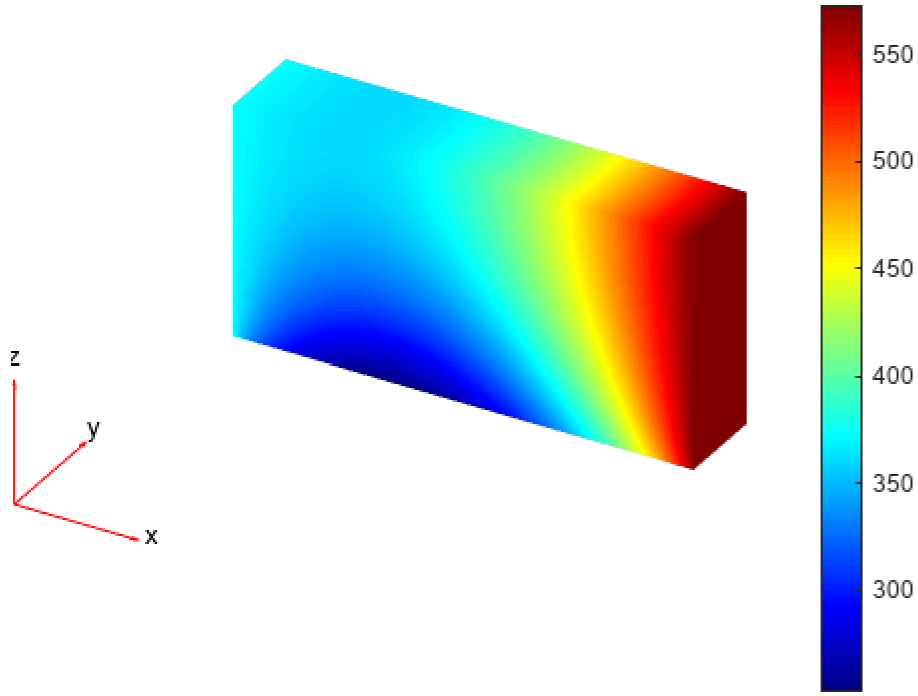
```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)
```

```
thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [12691x1 double]
     XGradients: [12691x1 double]
     YGradients: [12691x1 double]
     ZGradients: [12691x1 double]
```

```
            Mesh: [1x1 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)
```



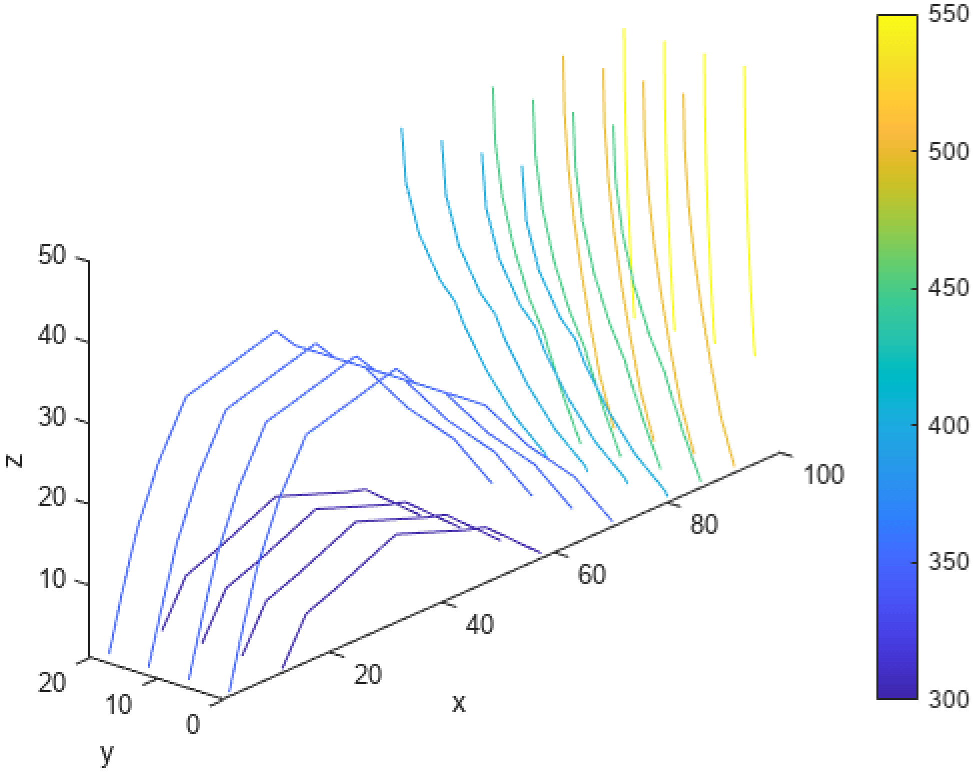
Create a grid specified by `x`, `y`, and `z` coordinates and interpolate temperatures to the grid.

```
[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
```

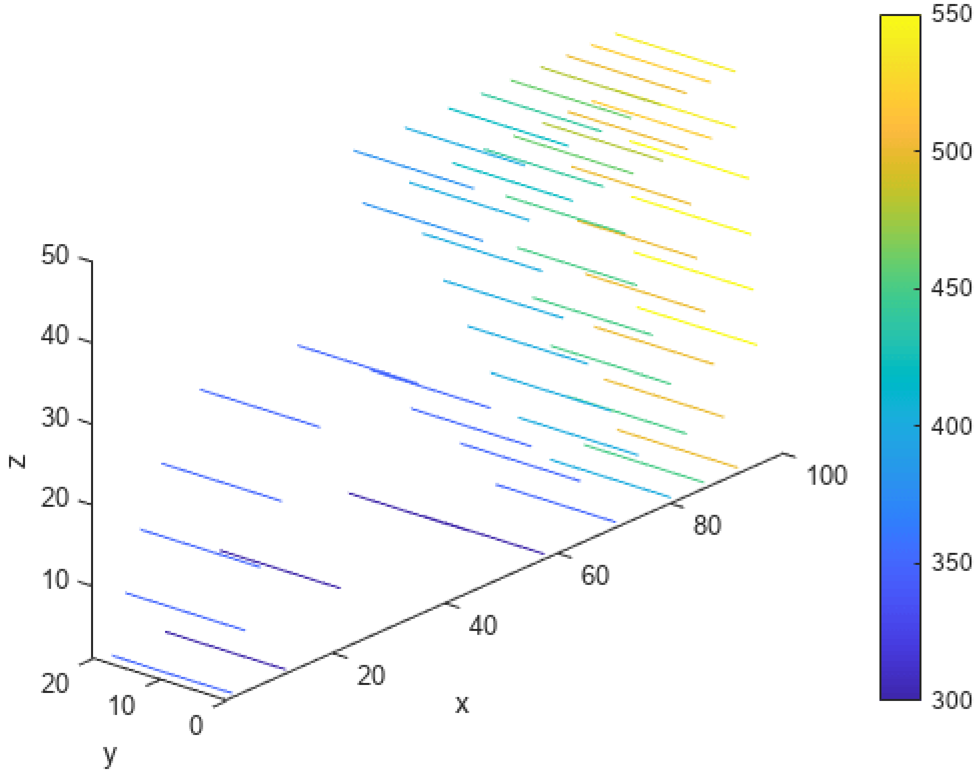


Alternatively, you can specify the grid by using a matrix of query points.

```
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

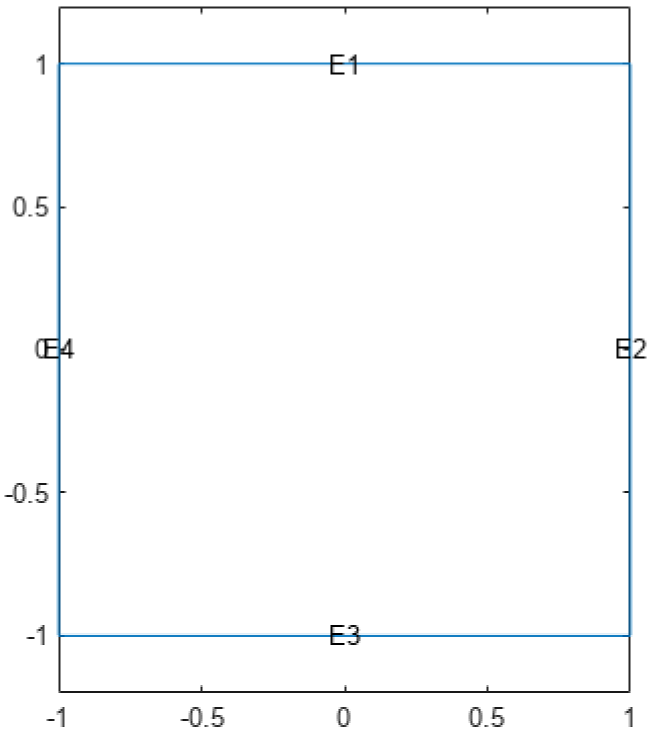
Solve a 2-D transient heat transfer problem on a square domain and compute temperatures at the convective boundary.

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.

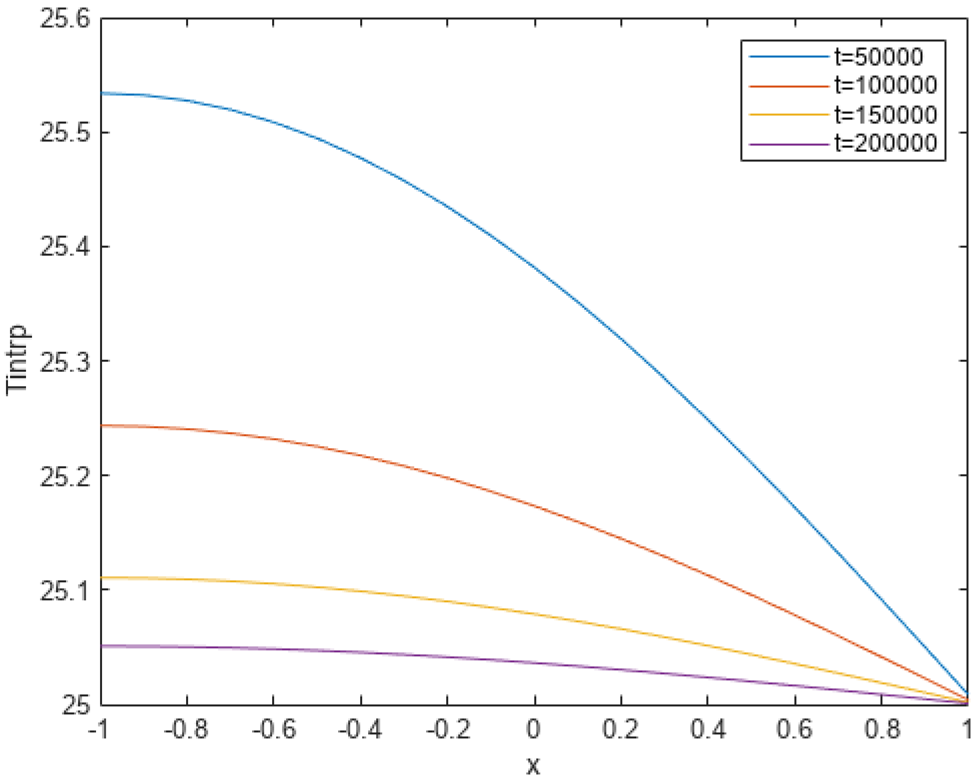
```
generateMesh(thermalmodel);
tlist = 0:1000:200000;
thermalresults = solve(thermalmodel,tlist);
```

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")
```



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 `Tintrp = reshape(Tintrp,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`.

Version History

Introduced in R2017a

See Also

`ThermalModel` | `SteadyStateThermalResults` | `TransientThermalResults` | `evaluateHeatFlux` | `evaluateHeatRate` | `evaluateTemperatureGradient`

Topics

“Dimensions of Solutions, Gradients, and Fluxes” on page 3-356

interpolateVelocity

Package: `pde`

Interpolate velocity at arbitrary spatial locations for all time or frequency steps for dynamic structural model

Syntax

```
intrpVel = interpolateVelocity(structuralresults,xq,yq)
intrpVel = interpolateVelocity(structuralresults,xq,yq,zq)
intrpVel = interpolateVelocity(structuralresults,querypoints)
```

Description

`intrpVel = interpolateVelocity(structuralresults,xq,yq)` returns the interpolated velocity values at the 2-D points specified in `xq` and `yq` for all time or frequency steps.

`intrpVel = interpolateVelocity(structuralresults,xq,yq,zq)` uses the 3-D points specified in `xq`, `yq`, and `zq`.

`intrpVel = interpolateVelocity(structuralresults,querypoints)` uses the points specified in `querypoints`.

Examples

Interpolate Velocity for 3-D Structural Dynamic Problem

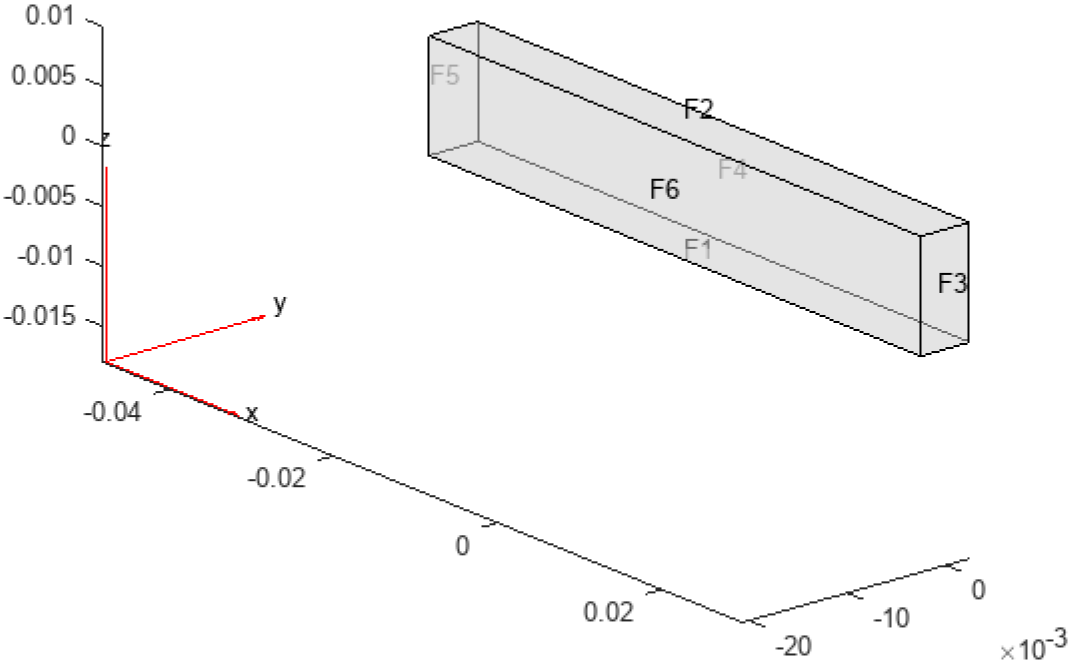
Interpolate velocity at the geometric center of a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the y-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0], ...
                             "Velocity",[0;0;0]);
```

Solve the model.

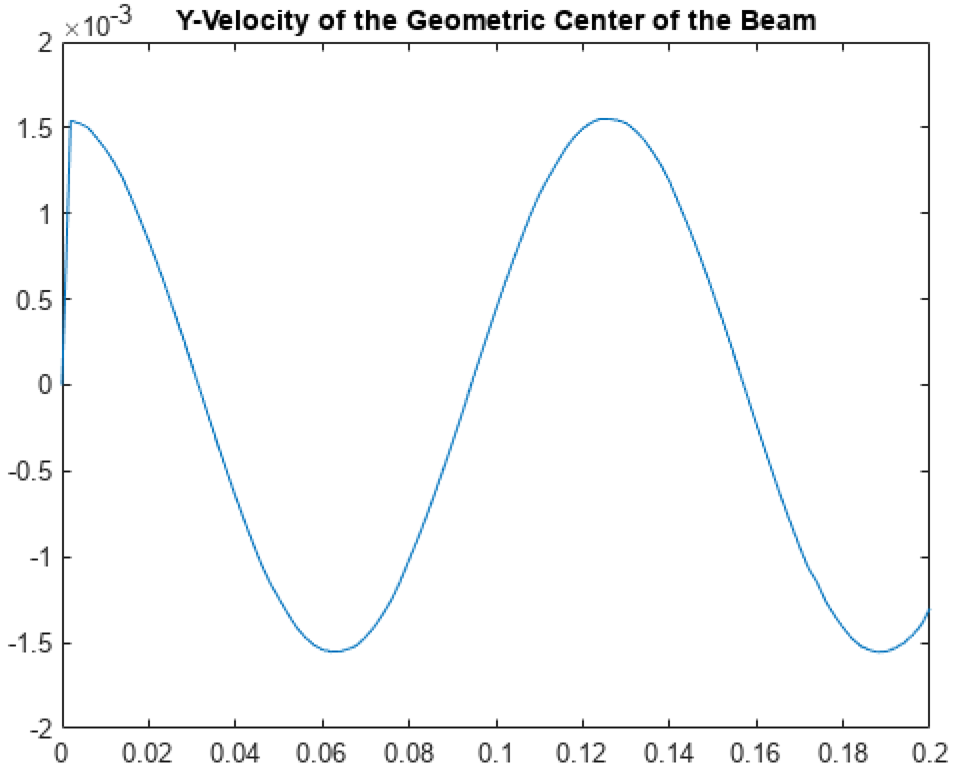
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Interpolate velocity at the geometric center of the beam.

```
coordsMidSpan = [0;0;0.005];
intrpVel = interpolateVelocity(structuralresults,coordsMidSpan);
```

Plot the `y`-component of velocity of the geometric center of the beam.

```
figure
plot(structuralresults.SolutionTimes,intrpVel.vy)
title("Y-Velocity of the Geometric Center of the Beam")
```



Input Arguments

`structuralresults` — Solution of dynamic structural analysis problem
`TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the dynamic structural analysis problem, specified as a `TransientStructuralResults` or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel,tlist)`

xq — *x*-coordinate query points
real array

x-coordinate query points, specified as a real array. `interpolateVelocity` evaluates velocities at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateVelocity` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. It returns velocities as an `FEStruct` object with the properties containing vectors of the same size as these column vectors. To ensure that the dimensions of the returned solution are consistent with the dimensions of the original query points, use the `reshape` function. For example, use `intrpVel = reshape(intrpVel.ux,size(xq))`.

Data Types: `double`

`yq` — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateVelocity` evaluates velocities at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries. Internally, `interpolateVelocity` 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. `interpolateVelocity` evaluates velocities at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries. Internally, `interpolateVelocity` 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. `interpolateVelocity` evaluates velocities 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`

Output Arguments

`intrpVel` — Velocities at query points
`FEStruct` object

Velocities at the query points, returned as an `FEStruct` object with the properties representing spatial components of velocity at the query points. For query points that are outside the geometry, `intrpVel` returns `NaN`. Properties of an `FEStruct` object are read-only.

Version History

Introduced in R2018a

See Also

`StructuralModel` | `TransientStructuralResults` | `interpolateDisplacement` | `interpolateAcceleration` | `interpolateStress` | `interpolateStrain` | `interpolateVonMisesStress` | `evaluateStress` | `evaluateStrain` | `evaluateVonMisesStress` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

interpolateVonMisesStress

Package: `pde`

Interpolate von Mises stress at arbitrary spatial locations

Syntax

```
intrpVMStress = interpolateVonMisesStress(structuralresults,xq,yq)
intrpVMStress = interpolateVonMisesStress(structuralresults,xq,yq,zq)
intrpVMStress = interpolateVonMisesStress(structuralresults,querypoints)
```

Description

`intrpVMStress = interpolateVonMisesStress(structuralresults,xq,yq)` returns the interpolated von Mises stress values at the 2-D points specified in `xq` and `yq`. For transient and frequency-response structural models, `interpolateVonMisesStress` interpolates von Mises stress for all time- or frequency-steps, respectively.

`intrpVMStress = interpolateVonMisesStress(structuralresults,xq,yq,zq)` uses the 3-D points specified in `xq`, `yq`, and `zq`.

`intrpVMStress = interpolateVonMisesStress(structuralresults,querypoints)` uses the points specified in `querypoints`.

Examples

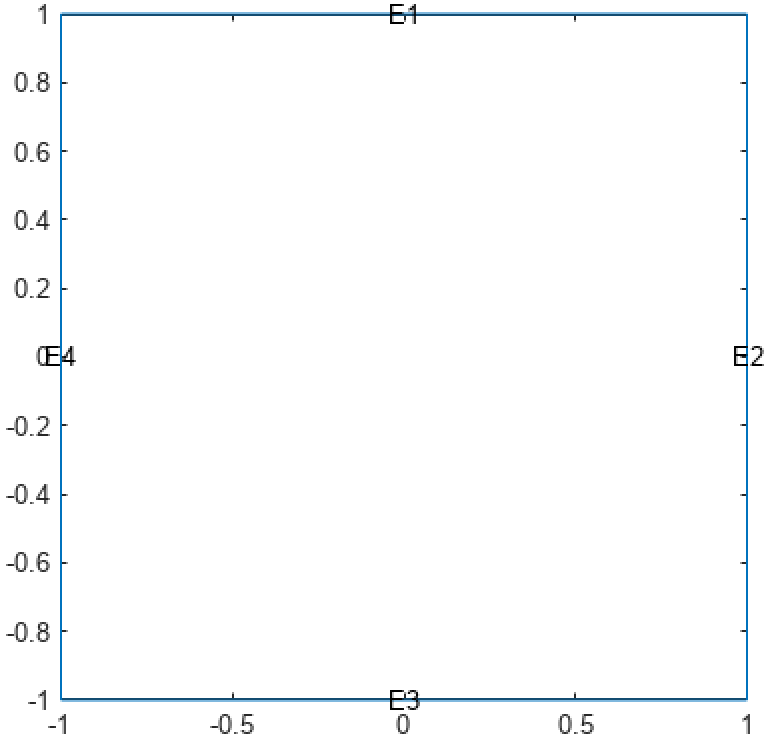
Interpolate von Mises Stress for Plane-Strain Problem

Create a structural analysis model for a plane-strain problem.

```
structuralmodel = createpde("structural","static-planestrain");
```

Include the square geometry in the model. Plot the geometry.

```
geometryFromEdges(structuralmodel,@squareg);
pdegplot(structuralmodel,"EdgeLabels","on")
axis equal
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"PoissonsRatio",0.3, ...
                                     "YoungsModulus",210E3);
```

Specify the x-component of the enforced displacement for edge 1.

```
structuralBC(structuralmodel,"XDisplacement",0.001,"Edge",1);
```

Specify that edge 3 is a fixed boundary.

```
structuralBC(structuralmodel,"Constraint","fixed","Edge",3);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel);
```

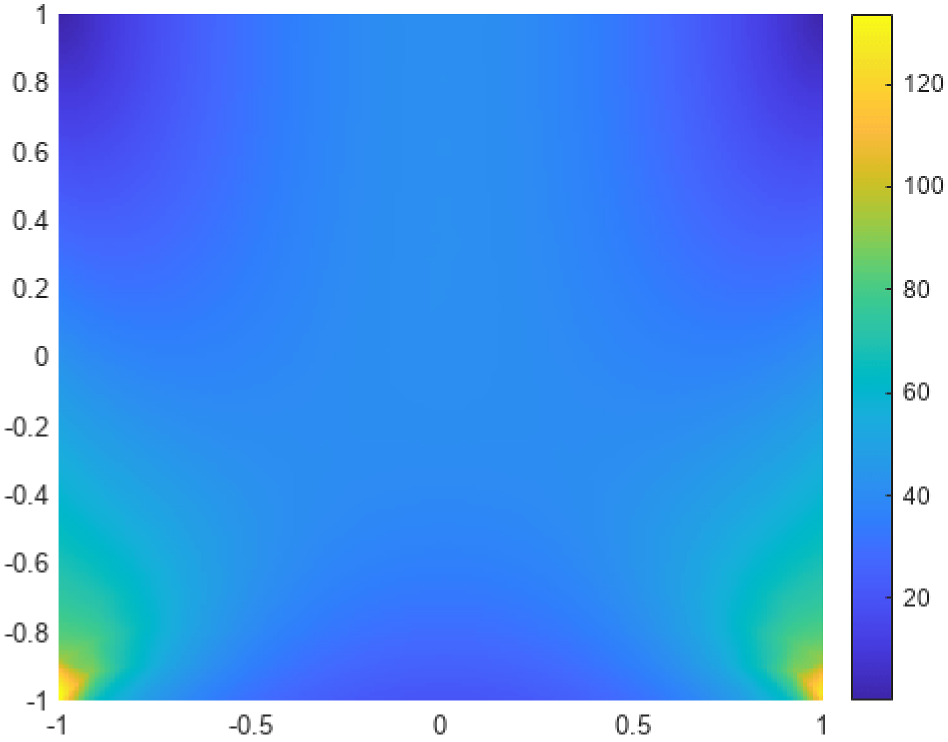
Create a grid and interpolate the von Mises stress to the grid.

```
v = linspace(-1,1,151);
[X,Y] = meshgrid(v);
intrpVMStress = interpolateVonMisesStress(structuralresults,X,Y);
```

Reshape the von Mises stress to the shape of the grid and plot it.

```
VMStress = reshape(intrpVMStress,size(X));
p = pcolor(X,Y,VMStress);
```

```
p.EdgeColor="none";
colorbar
```



Interpolate Von Mises Stress for 3-D Static Structural Analysis Problem

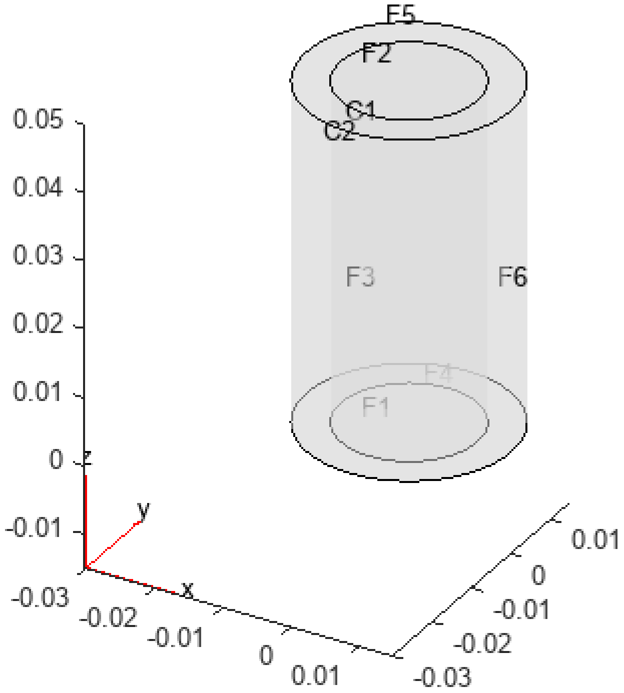
Solve a static structural model representing a bimetallic cable under tension, and interpolate the von Mises stress on a cross-section of the cable.

Create a static structural model for solving a solid (3-D) problem.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01,0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

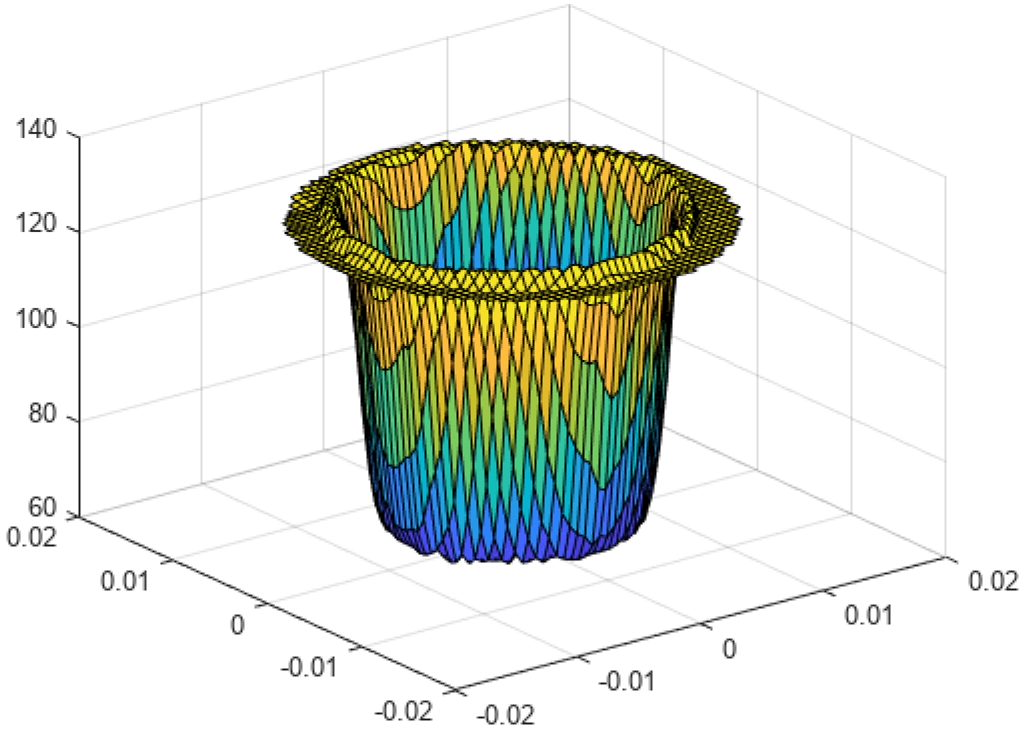
```
              Mesh: [1x1 FEMesh]
```

Define the coordinates of a midspan cross-section of the cable.

```
[X,Y] = meshgrid(linspace(-0.015,0.015,50));
Z = ones(size(X))*0.025;
```

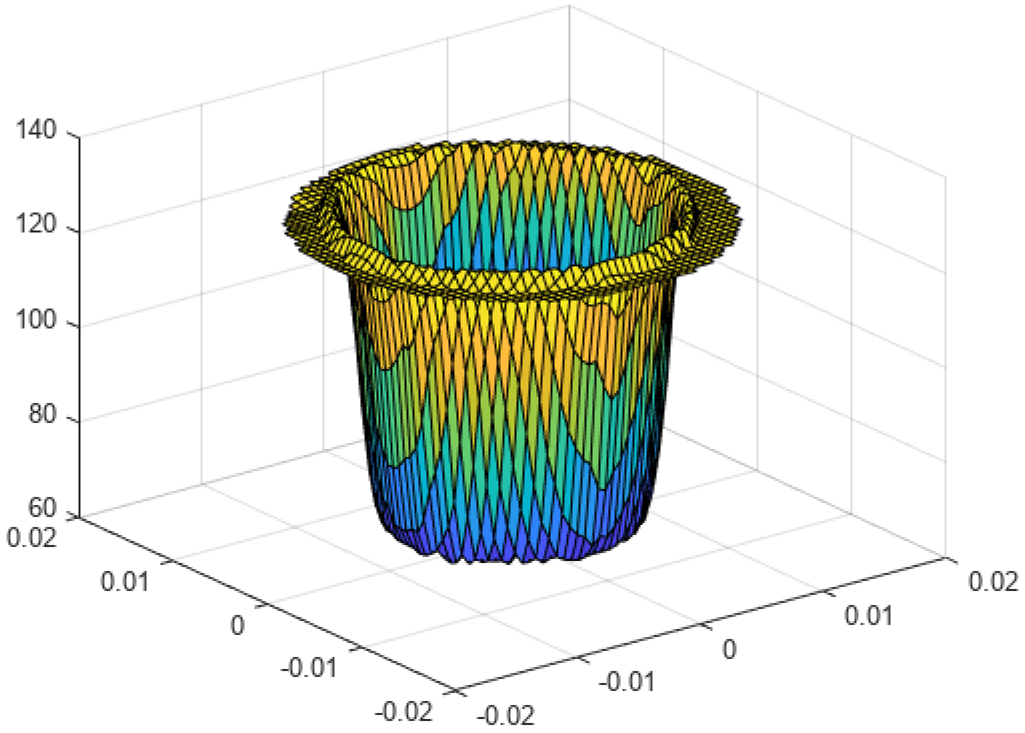
Interpolate the von Mises stress and plot the result.

```
IntrpVMStress = interpolateVonMisesStress(structuralresults,X,Y,Z);
surf(X,Y,reshape(IntrpVMStress,size(X)))
```



Alternatively, you can specify the grid by using a matrix of query points.

```
querypoints = [X(:),Y(:),Z(:)]';
IntrpVMStress = ...
    interpolateVonMisesStress(structuralresults,querypoints);
surf(X,Y,reshape(IntrpVMStress,size(X)))
```



Interpolate von Mises Stress for 3-D Structural Dynamic Problem

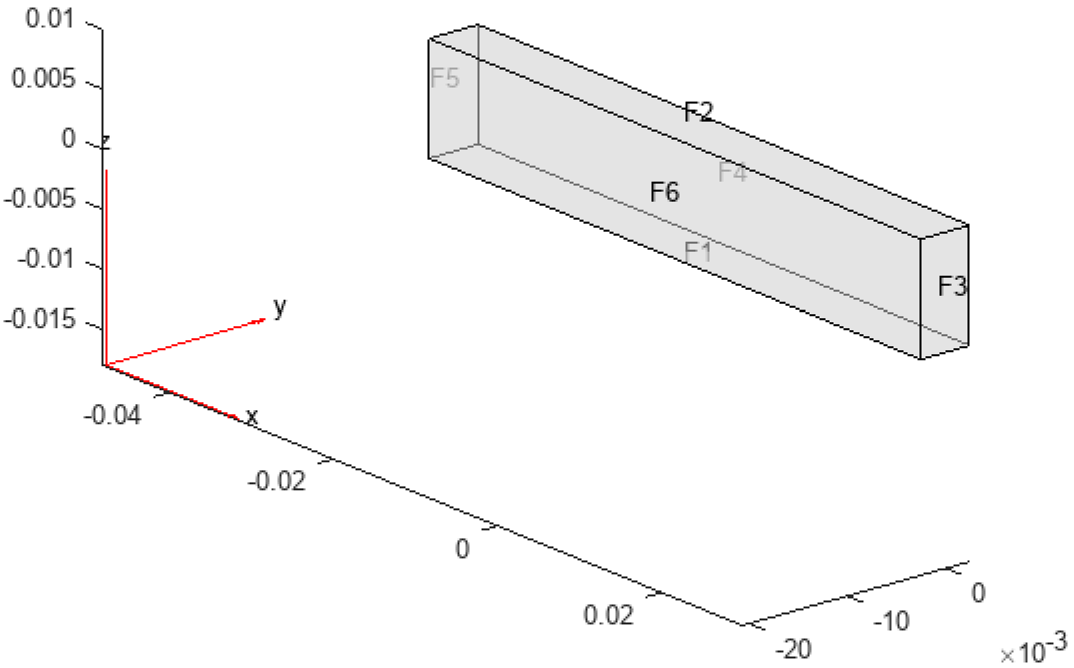
Interpolate the von Mises stress at the geometric center of a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the `y`-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0], ...
                             "Velocity",[0;0;0]);
```

Solve the model.

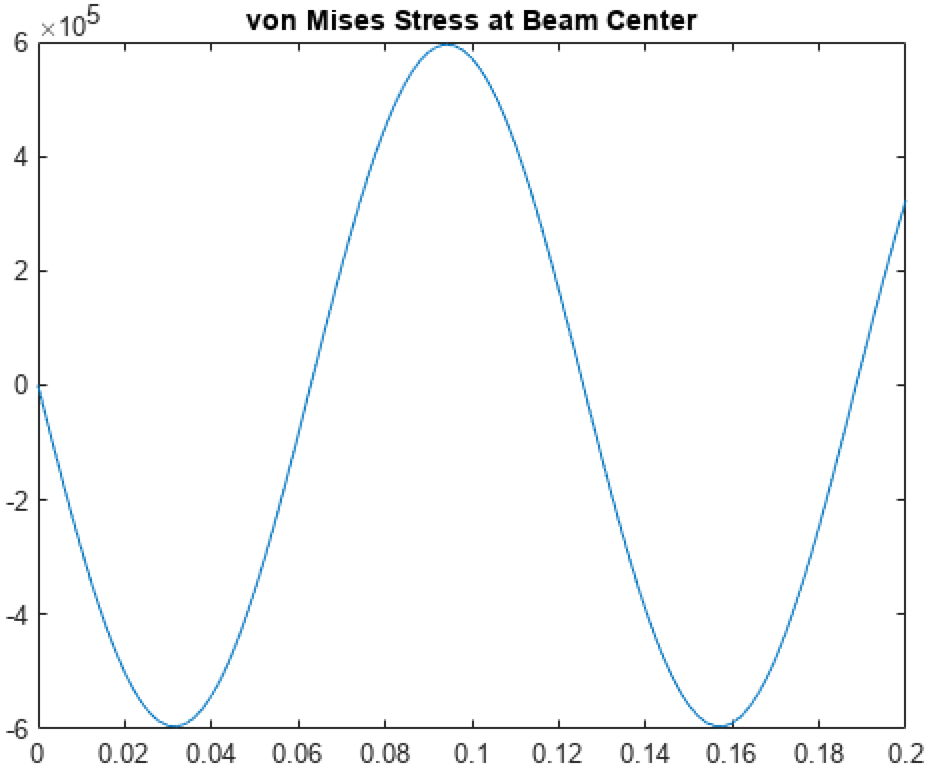
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Interpolate the von Mises stress at the geometric center of the beam.

```
coordsMidSpan = [0;0;0.005];
intrpStress = interpolateStress(structuralresults,coordsMidSpan);
```

Plot the von Mises stress at the geometric center of the beam.

```
figure
plot(structuralresults.SolutionTimes,intrpStress.sxx)
title("von Mises Stress at Beam Center")
```



Input Arguments

`structuralresults` — Solution of structural analysis problem

`StaticStructuralResults` object | `TransientStructuralResults` object | `FrequencyStructuralResults` object

Solution of the structural analysis problem, specified as a `StaticStructuralResults`, `TransientStructuralResults`, or `FrequencyStructuralResults` object. Create `structuralresults` by using the `solve` function.

Example: `structuralresults = solve(structuralmodel)`

`xq` — *x*-coordinate query points

real array

x-coordinate query points, specified as a real array. `interpolateVonMisesStress` evaluates the von Mises stress at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries.

`interpolateVonMisesStress` converts query points to column vectors `xq(:)`, `yq(:)`, and (if present) `zq(:)`. The function returns von Mises stress as a column vector of the same size as the query point column vectors. To ensure that the dimensions of the returned solution are consistent with the dimensions of the original query points, use the `reshape` function. For example, use `intrpVMStress = reshape(intrpVMStress,size(xq))`.

Data Types: `double`

yq — *y*-coordinate query points
real array

y-coordinate query points, specified as a real array. `interpolateVonMisesStress` evaluates the von Mises stress at the 2-D coordinate points `[xq(i),yq(i)]` or at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and (if present) `zq` must have the same number of entries. Internally, `interpolateVonMisesStress` converts the 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. `interpolateVonMisesStress` evaluates the von Mises stress at the 3-D coordinate points `[xq(i),yq(i),zq(i)]`. Therefore, `xq`, `yq`, and `zq` must have the same number of entries. Internally, `interpolateVonMisesStress` converts the 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. `interpolateVonMisesStress` evaluates the von Mises stress 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`

Output Arguments

`intrpVMStress` — von Mises stress at query points
column vector

von Mises stress at the query points, returned as a column vector.

For query points that are outside the geometry, `intrpVMStress` = `NaN`.

Version History

Introduced in R2017b

Support for frequency response structural problems

For frequency response structural models, `interpolateVonMisesStress` interpolates von Mises stress for all frequency-steps.

Support for transient structural problems

For transient structural models, `interpolateVonMisesStress` interpolates von Mises stress for all time-steps.

See Also

`StructuralModel` | `StaticStructuralResults` | `interpolateDisplacement` | `interpolateStress` | `interpolateStrain` | `evaluateReaction` | `evaluatePrincipalStress` | `evaluatePrincipalStrain`

jigglemesh

Package: `pde`

(Not recommended) Jiggle internal points of triangular mesh

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 `generateMesh`.

Syntax

```
p1 = jigglemesh(p,e,t)
p1 = jigglemesh(p,e,t,Name,Value)
```

Description

`p1 = jigglemesh(p,e,t)` jiggles the triangular mesh by adjusting the node point positions. Typically, the quality of the mesh increases after jiggling.

`p1 = jigglemesh(p,e,t,Name,Value)` jiggles the mesh using one or more `Name,Value` arguments.

Examples

Jiggle Mesh

Create a triangular mesh of the square geometry by using `initmesh`. To avoid jiggling, call `initimesh` with the `Jiggle` value set to `off`.

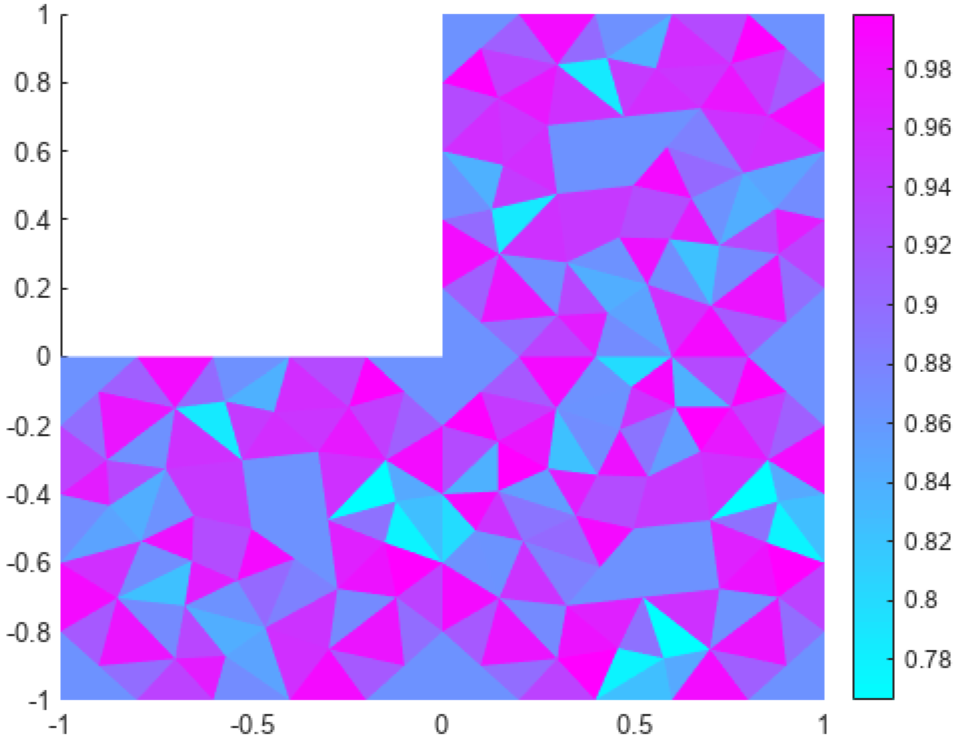
```
[p,e,t] = initmesh("lshapeg","Jiggle","off");
```

Evaluate quality of the mesh elements using the `pdetriq` function.

```
q = pdetriq(p,t);
```

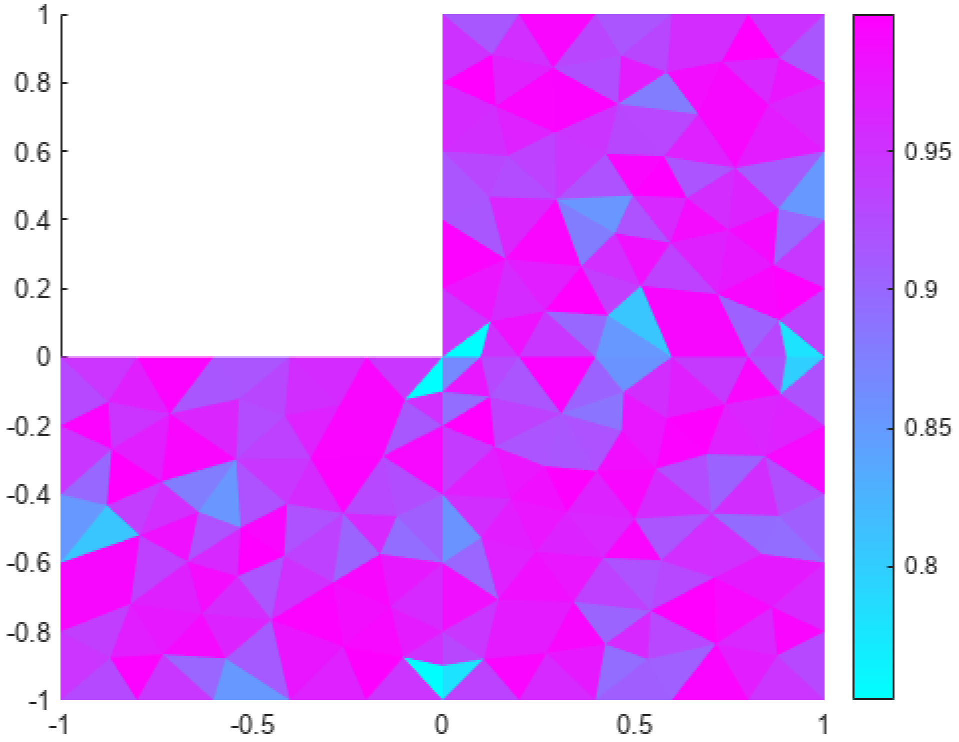
Plot the mesh.

```
pdeplot(p,e,t,"XYData",q,"ColorBar","on","XYStyle","flat")
```



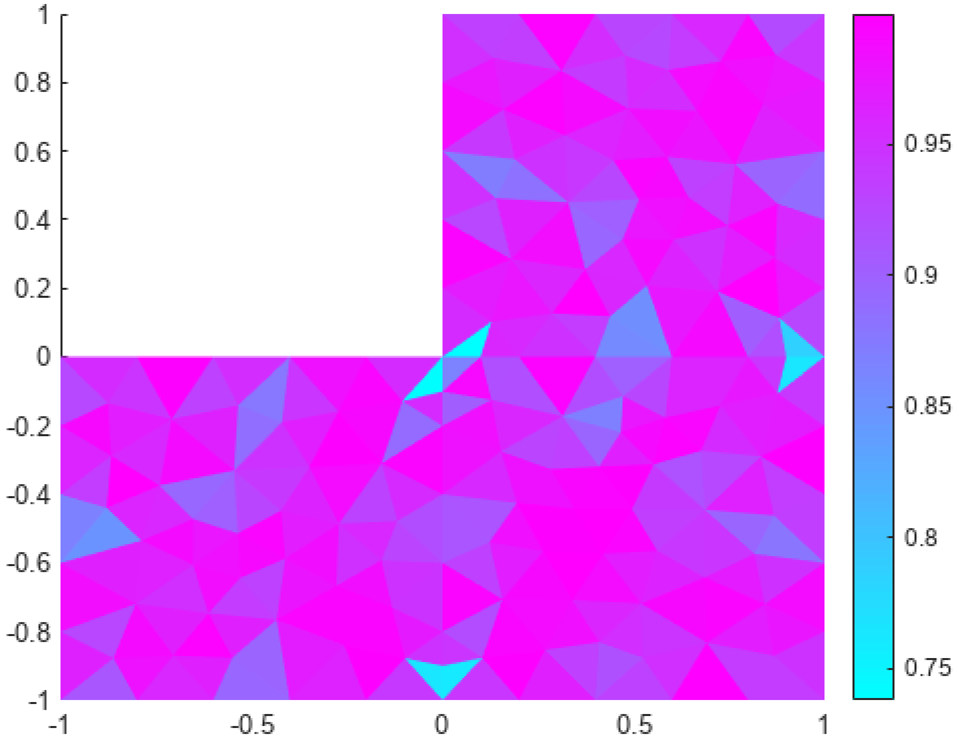
Jiggle the mesh using the default parameter values. Plot the result.

```
p1 = jigglemesh(p,e,t);
q = pdetriq(p1,t);
pdeplot(p1,e,t,"XYData",q,"ColorBar","on","XYStyle","flat")
```



Now jiggle the original mesh again, this time using 50000 iterations.

```
p2 = jigglemesh(p,e,t,"Opt","off","Iter",5e4);
q = pdetriq(p2,t);
pdeplot(p2,e,t,"XYData",q,"ColorBar","on","XYStyle","flat")
```



Input Arguments

p — Mesh points
2-by-`Np` matrix

Mesh points, specified as a 2-by-`Np` matrix. `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)` and the *y*-coordinate of point `k` in `p(2,k)`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

e — Mesh edges
7-by-`Ne` matrix

Mesh edges, specified as a 7-by-`Ne` matrix, where `Ne` is the number of edges in the mesh. An edge is a pair of points in `p` containing a boundary between subdomains, or containing an outer boundary. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

t — Mesh elements
4-by-`Nt` matrix

Mesh elements, specified as a 4-by-`Nt` matrix. `Nt` is the number of triangles in the mesh.

The `t(i,k)`, with `i` ranging from 1 through `end - 1`, contain indices to the corner points of element `k`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172. The last row, `t(end,k)`, contains the subdomain number of the element.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `p1 = jigglemesh(p,e,t,"Iter",Inf)`

`Opt` — Optimization method

`"mean"` (default) | `"minimum"` | `"off"`

Optimization method, specified as the comma-separated pair consisting of `"Opt"` and `"mean"`, `"minimum"`, or `"off"`.

Jiggling a mesh moves each mesh point not located on an edge segment towards the center of mass of the polygon formed by the adjacent triangles. The optimization method controls how many times `jigglemesh` repeats this process:

- If `Opt` is `"off"`, `jigglemesh` repeats this process `Iter` times. The default value of `Iter` in this case is 1.
- If `Opt` is `"mean"`, `jigglemesh` repeats this process until the mean triangle quality stops increasing significantly or until the maximum number of iterations is reached. The default value of `Iter` in this case is 20.
- If `Opt` is `"minimum"`, `jigglemesh` repeats this process until the minimum triangle quality stops increasing significantly or until the maximum number of iterations is reached. The default value of `Iter` in this case is 20.

Example: `p1 = jigglemesh(p,e,t,"Opt","off","Iter",1000);`

Data Types: `char` | `string`

`Iter` — Maximum number of iterations

1 or 20 (default) | positive integer

Maximum number of iterations, specified as the comma-separated pair consisting of `"Iter"` and a positive number. The default value depends on the `Opt` argument value. If `Opt` is set to `"mean"` (default) or `"minimum"`, the default maximum number of iterations is 20. If `Opt` is set to `"off"`, the default maximum number of iterations is 1.

Example: `p1 = jigglemesh(p,e,t,"Opt","off","Iter",1000);`

Data Types: `double`

Output Arguments

p1 — Modified mesh points

2-by-`Np` matrix

Modified mesh points, returned as a 2-by-`Np` matrix. `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)` and the *y*-coordinate of point `k` in `p(2,k)`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

Version History

Introduced before R2006a

See Also

`initmesh` | `pdetriq`

Topics

“Mesh Data as [p,e,t] Triples” on page 2-172

linearize

Package: `pde`

Linearize structural or thermal model

Syntax

```
sys = linearize(model)
mx = linearize(model,"OutputType","matrices")
```

Description

`sys = linearize(model)` extracts a sparse linear model for use with Control System Toolbox. For a structural analysis model, `linearize` extracts a `mechss` model. For a thermal analysis model, it extracts a `sparss` model. For transient models, `linearize` uses time 0.

Use `linearizeInput` to specify the inputs of the linear model that correspond to external forcing, such as loads or internal heat sources. The toolbox treats the value of each selected constraint, load, or source as a constant, and the value becomes one input channel in the linearized model. The remaining boundary conditions are set to zero for linearization purposes, regardless of their value in the structural or thermal model. Ensure that you label all nonzero boundary conditions and pass them as inputs using `linearizeInput`.

Use `linearizeOutput` to specify the outputs of the linear model in terms of regions of the geometry, such as cells (for 3-D geometries only), faces, edges, or vertices. This includes all degrees of freedom (DoFs) in the specified region as output values. For structural models, you can also specify which of the *x*, *y*, and *z* degrees of freedom to include as outputs.

Use `sys.InputName` and `sys.OutputGroup` to locate the inputs and outputs of `sys` that correspond to a particular boundary condition or to a selected region.

`mx = linearize(model,"OutputType","matrices")` returns the finite element matrices `A`, `B`, `C`, `D`, `E` or `M`, `K`, `B`, `F` used to construct the `mechss` and `sparss` models in the previous syntax.

Examples

Extract `sparss` Model and Finite Element Matrices

Linearize a model for thermal analysis and return finite element matrices.

Create a transient thermal model.

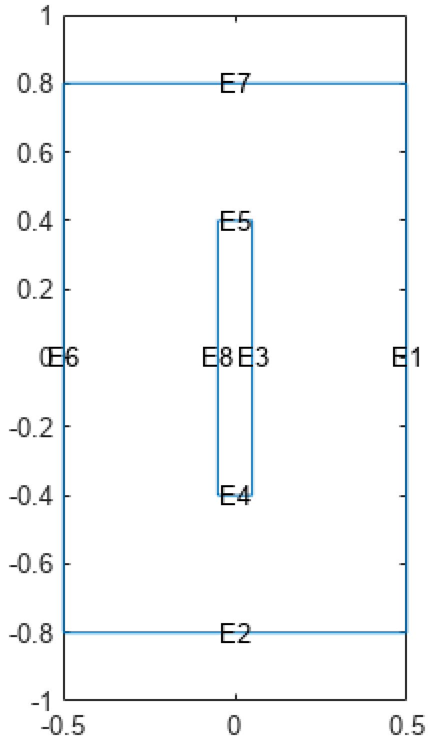
```
thermalmodel = createpde("thermal","transient");
```

Add the block geometry to the thermal model by using the `geometryFromEdges` function. The geometry description file for this problem is called `crackg.m`.

```
geometryFromEdges(thermalmodel,@crackg);
```

Plot the geometry with the edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
ylim([-1,1])
axis equal
```



Generate a mesh.

```
generateMesh(thermalmodel);
```

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",1, ...
                               "MassDensity",1, ...
                               "SpecificHeat",1);
```

Specify the temperature on the left edge as `100`, and constant heat flow to the exterior through the right edge as `-10`. Add a unique label to each boundary condition.

```
thermalBC(thermalmodel,"Edge",6,"Temperature",100,"Label","TempBC");
thermalBC(thermalmodel,"Edge",1,"HeatFlux",-10,"Label","FluxBC");
```

Specify that the entire geometry generates heat and add a unique label to this assignment.

```
internalHeatSource(thermalmodel,25,"Label","HeatSource");
```

Set an initial value of 0 for the temperature.

```
thermalIC(thermalmodel,0);
```

Specify the inputs of the linearized model by calling the `linearizeInput` function with the previously defined labels for the boundary conditions and the internal heat source. Add one label per function call.

```
linearizeInput(thermalmodel,"HeatSource");
linearizeInput(thermalmodel,"TempBC");
linearizeInput(thermalmodel,"FluxBC");
```

Specify the outputs of the linearized model by calling the `linearizeOutput` function to set the regions of interest for measuring temperature. Specify one region per function call. For example, specify that the output is the temperature value at all nodes on edge 2.

```
linearizeOutput(thermalmodel,"Edge",2);
```

Measure the temperature on edge 2.

```
sys = linearize(thermalmodel)
```

```
Sparse continuous-time state-space model with 27 outputs, 3 inputs, and 1363 states.

Use "spy" and "showStateInfo" to inspect model structure.
Type "properties('sparss')" for a list of model properties.
Type "help sparssOptions" for available solver options for this model.
```

In the linearized model, use `sys.InputName` to check that the inputs to `sys` are the heat source, the temperature on edge 6, and the heat flux on edge 1.

```
sys.InputName
```

```
ans = 3x1 cell
    {'HeatSource'}
    {'TempBC'    }
    {'FluxBC'    }
```

In the linearized model, use `sys.OutputGroup` to locate the sections associated with each coordinate.

```
sys.OutputGroup
```

```
ans = struct with fields:
    Edge2: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 ... ]
```

If you do not have Control System Toolbox™, you can access the finite element matrices `A`, `B`, `C`, and `E` as follows.

```
mx = linearize(thermalmodel,"OutputType","matrices")
```

```
mx = struct with fields:
    A: [1363x1363 double]
    B: [1363x3 double]
    C: [27x1363 double]
    E: [1363x1363 double]
```

Extract `mechss` Model and Finite Element Matrices

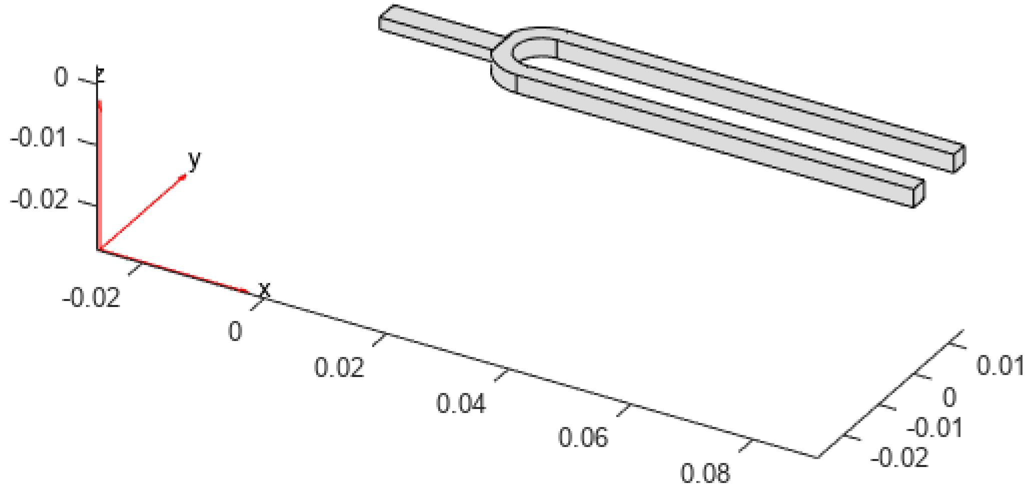
Linearize a structural model and return finite element matrices.

Create a structural transient analysis model.

```
structuralmodel = createpde("structural","transient-solid");
```

Import and plot the tuning fork geometry.

```
importGeometry(structuralmodel,"TuningFork.stl");
pdegplot(structuralmodel)
```



Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.005);
```

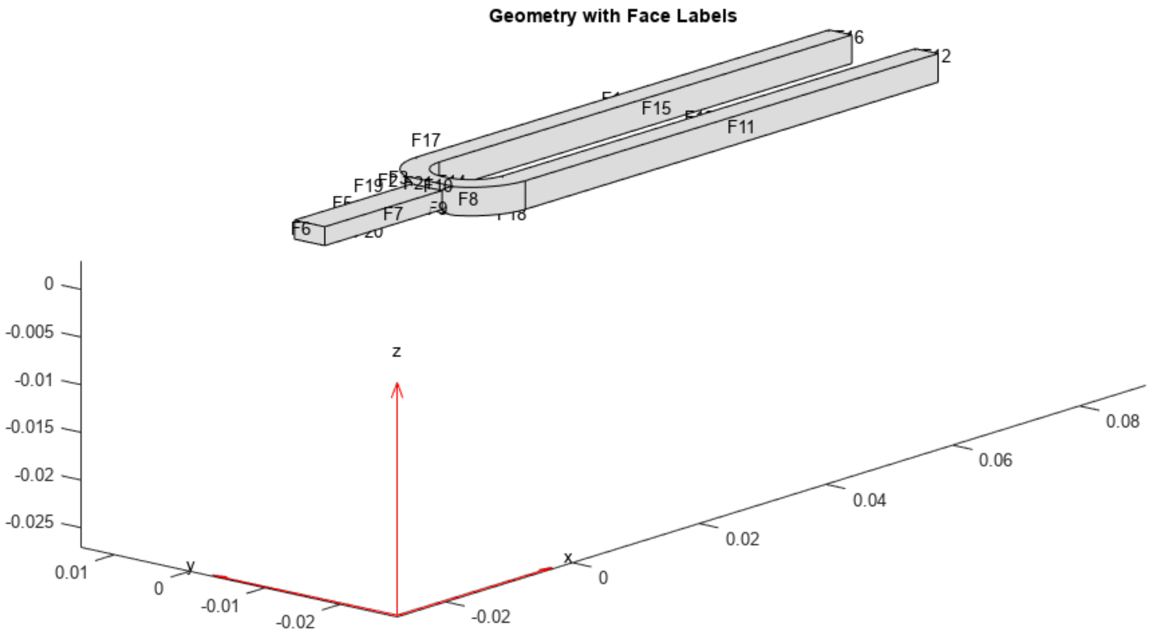
Specify Young's modulus, Poisson's ratio, and the mass density to model linear elastic material behavior. Specify all physical properties in consistent units.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",8000);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face labels.

```
figure("units","normalized","outerposition",[0 0 1 1])
pdegplot(structuralmodel,"FaceLabels","on")
```

```
view(-50,15)
title("Geometry with Face Labels")
```



Impose sufficient boundary constraints to prevent rigid body motion under applied loading. Typically, you hold a tuning fork by hand or mount it on a table. A simplified approximation to this boundary condition is fixing a region near the intersection of tines and the handle (faces 21 and 22).

```
structuralBC(structuralmodel,"Face",[21,22],"Constraint","fixed");
```

Specify the pressure loading on a tine as a short rectangular pressure pulse.

```
structuralBoundaryLoad(structuralmodel,"Face",11,"Pressure",5E6, ...
                                       "EndTime",1e-3,"Label","Pressure");
```

Specify acceleration due to gravity as a body load.

```
structuralBodyLoad(structuralmodel,"GravitationalAcceleration",[0 0 -1], ...
                                   "Label","Gravity");
```

Create inputs for gravity and the pressure pulse on tuning fork.

```
linearizeInput(structuralmodel,"Gravity");
linearizeInput(structuralmodel,"Pressure");
```

Measure the *y*-displacement of face 12 and *x*-displacement of face 6.

```
linearizeOutput(structuralmodel,"Face",12,"Component","y");
linearizeOutput(structuralmodel,"Face",6,"Component","x");
```

Obtain a `mechss` model of the tuning fork.

```
sys = linearize(structuralmodel)
```

```
Sparse continuous-time second-order model with 26 outputs, 4 inputs, and 3240 degrees of freedom

Use "spy" and "showStateInfo" to inspect model structure.
Type "properties('mechss')" for a list of model properties.
Type "help mechssOptions" for available solver options for this model.
```

In the linearized model, use `sys.InputName` to check that the inputs to `sys` are the gravity body load and the pressure pulse on a tine. The gravity body load produces three inputs because it has *x*-, *y*-, and *z*-components.

```
sys.InputName
```

```
ans = 4x1 cell
    {'Gravity_x'}
    {'Gravity_y'}
    {'Gravity_z'}
    {'Pressure' }
```

In the linearized model, use `sys.OutputGroup` to locate the sections associated with each coordinate.

```
sys.OutputGroup
```

```
ans = struct with fields:
    Face12_y: [1 2 3 4 5 6 7 8 9 10 11 12 13]
     Face6_x: [14 15 16 17 18 19 20 21 22 23 24 25 26]
```

If you do not have Control System Toolbox™, you can access the finite element matrices `M`, `K`, `B`, and `F` as follows.

```
mx = linearize(structuralmodel,"OutputType","matrices")
```

```
mx = struct with fields:
    M: [3240x3240 double]
    K: [3240x3240 double]
    B: [3240x4 double]
    F: [26x3240 double]
```

Input Arguments

`model` — Structural or thermal model

`StructuralModel` object | `ThermalModel` object

Structural or thermal model, specified as a `StructuralModel` object or a `ThermalModel` object. The `linearize` function does not support nonlinear thermal analysis.

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Output Arguments

`sys` — Sparse linear models for use with Control System Toolbox

`mechss` model object | `sparss` model object

Sparse linear models for use with Control System Toolbox, returned as a `mechss` or `sparss` model object.

`mx` — Finite element matrices

structure array

Finite element matrices `A`, `B`, `C`, `D`, and `E` or `M`, `K`, `B`, and `F`, returned as a structure array.

Version History

Introduced in R2021b

See Also

`linearizeInput` | `linearizeOutput`

linearizeInput

Package: `pde`

Specify inputs to linearized model

Syntax

```
linearizeInput(model,labeltext)
input = linearizeInput(model,labeltext)
```

Description

`linearizeInput(model,labeltext)` adds inputs for the boundary condition, constraint, load, or source with the label `labeltext`. In the linearized model, the input value $u = 1$ corresponds to a unit boundary condition acting on the entire region specified by `labeltext`. In other words, simulating the linearized model with the input value $u(t) = 25$ is equivalent to setting the boundary condition value to 25 in the thermal or structural model in Partial Differential Equation Toolbox. For more information, see “Algorithms” on page 5-757.

For a structural analysis model, the following boundary conditions, constraints, and loads can become inputs of the linearized model:

- A structural boundary constraint. Use the `structuralBC` function with the `Constraint` argument.
- A displacement or a displacement component on the boundary. Use the `structuralBC` function with the `Displacement`, `XDisplacement`, `YDisplacement`, or `ZDisplacement` argument.
- A structural boundary load. Use the `structuralBoundaryLoad` function with the `Pressure`, `Force`, or `SurfaceTraction` argument.
- A structural body load. Use the `structuralBodyLoad` function with the `GravitationalAcceleration` argument.

The boundary conditions, loads, or constraints with *x*-, *y*-, and *z*- components produce one input channel per component.

For a thermal analysis model, the following boundary conditions and sources can become inputs of the linearized model:

- A temperature or heat flux on the boundary. Use the `thermalBC` function with the `Temperature` or `HeatFlux` argument.
- An internal heat source. Use the `internalHeatSource` function.

Each selected condition or source produces a single scalar input in the linearized model.

To make a condition, constraint, load, or source available as a linearization input, always label it upon creation. For example, specify an internal heat source for a thermal model as follows:

```
internalHeatSource(thermalmodel,25,"Label","HeatSource");
```

The remaining boundary conditions are set to zero for linearization purposes, regardless of their value in the structural or thermal model. Ensure that you label all nonzero boundary conditions and pass them as inputs using `linearizeInput`.

Use `linearizeInput` and `linearizeOutput` together with the `linearize` function to extract sparse linear models from structural and thermal models.

`input = linearizeInput(model,labeltext)` returns a structure array `input` with the linearization input description.

Examples

Thermal Boundary Conditions and Internal Heat Source as Inputs for `linearize` Function

Use labels to pass the parameters of a 2-D thermal analysis model to the `linearize` function. This function extracts sparse linear models for use with Control System Toolbox™.

Create a transient thermal model.

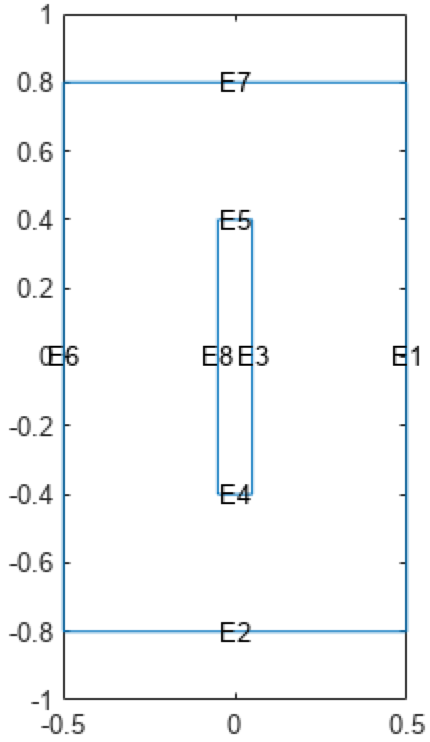
```
thermalmodel = createpde("thermal","transient");
```

Add the block geometry to the thermal model by using the `geometryFromEdges` function. The geometry description file for this problem is called `crackg.m`.

```
geometryFromEdges(thermalmodel,@crackg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
ylim([-1,1])
axis equal
```



Generate a mesh.

```
generateMesh(thermalmodel);
```

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",1, ...
                               "MassDensity",1, ...
                               "SpecificHeat",1);
```

Specify the temperature on the left edge as `100`, and constant heat flow to the exterior through the right edge as `-10`. Add a unique label to each boundary condition.

```
thermalBC(thermalmodel,"Edge",6,"Temperature",100,"Label","TempBC");
thermalBC(thermalmodel,"Edge",1,"HeatFlux",-10,"Label","FluxBC");
```

Specify that the entire geometry generates heat and add a unique label to this assignment.

```
internalHeatSource(thermalmodel,25,"Label","HeatSource");
```

Set an initial value of `0` for the temperature.

```
thermalIC(thermalmodel,0);
```

Call the `linearizeInput` function with the previously defined labels for the boundary conditions and the internal heat source to set the inputs for the `linearize` function. Add one label per function call.

```
linearizeInput(thermalmodel,"HeatSource");
linearizeInput(thermalmodel,"TempBC");
linearizeInput(thermalmodel,"FluxBC");
```

The `LinearizeInputs` property of `thermalmodel` stores the inputs.

```
thermalmodel.LinearizeInputs
```

```
ans=1×3 struct array with fields:
    RegionType
    RegionID
    Label
```

Pressure and Gravity as Inputs for `linearize` Function

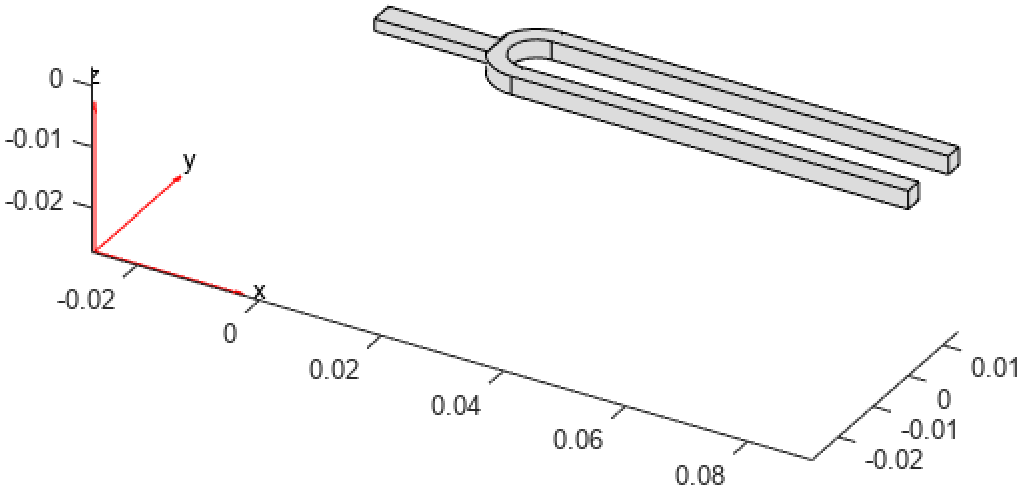
Create inputs for gravity and a short pressure pulse on tuning fork.

Create a structural transient analysis model.

```
structuralmodel = createpde("structural","transient-solid");
```

Import and plot the tuning fork geometry.

```
importGeometry(structuralmodel,"TuningFork.stl");
pdegplot(structuralmodel)
```



Generate a mesh.

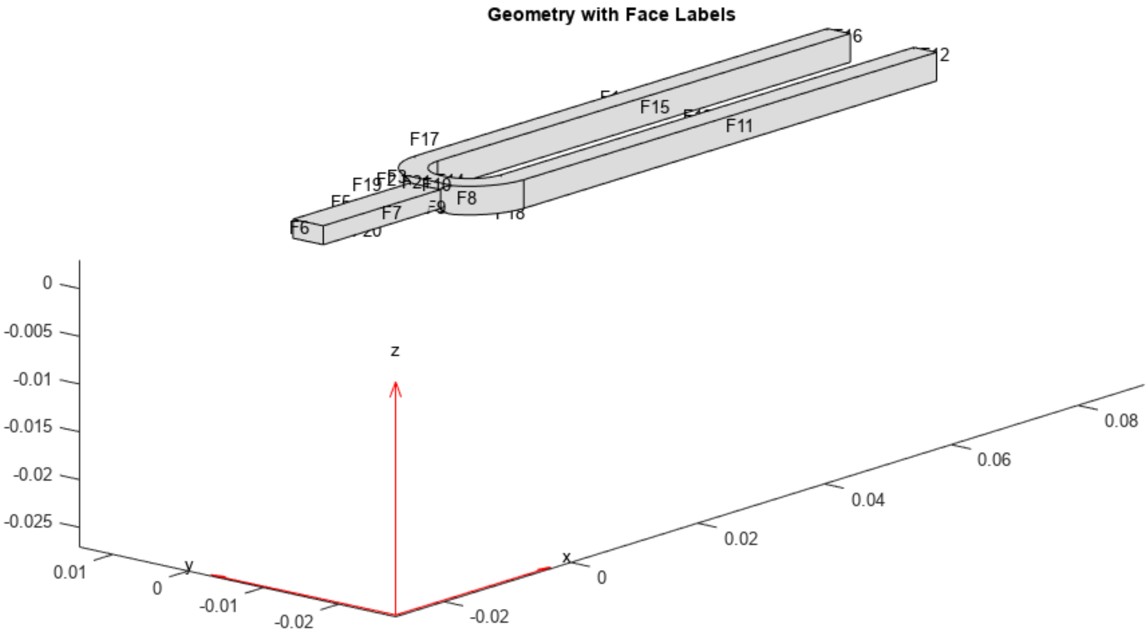
```
generateMesh(structuralmodel,"Hmax",0.005);
```

Specify Young's modulus, Poisson's ratio, and the mass density to model linear elastic material behavior. Specify all physical properties in consistent units.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",8000);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face labels.

```
figure("units","normalized","outerposition",[0 0 1 1])
pdegplot(structuralmodel,"FaceLabels","on")
view(-50,15)
title("Geometry with Face Labels")
```



Impose sufficient boundary constraints to prevent rigid body motion under applied loading. Typically, you hold a tuning fork by hand or mount it on a table. As a simplified approximation of this boundary condition, fix a region near the intersection of the tines and the handle (faces 21 and 22).

```
structuralBC(structuralmodel,"Face",[21,22],"Constraint","fixed");
```

Specify the pressure loading on a tine as a short rectangular pressure pulse.

```
structuralBoundaryLoad(structuralmodel,"Face",11, ...
                                       "Pressure",5E6, ...
                                       "EndTime",1e-3, ...
                                       "Label","Pressure");
```

Specify the acceleration due to gravity as a body load.

```
structuralBodyLoad(structuralmodel, ...
                   "GravitationalAcceleration",[0 0 -1], ...
                                     "Label","Gravity");
```

Create inputs for gravity and the pressure pulse on the tuning fork.

```
linearizeInput(structuralmodel,"Gravity");
linearizeInput(structuralmodel,"Pressure");
```

The `LinearizeInputs` property of `structuralmodel` stores the inputs.

```
structuralmodel.LinearizeInputs
```

```
ans=1×2 struct array with fields:
    RegionType
    RegionID
    Label
```

Input Arguments

`model` — Structural or linear thermal model

`StructuralModel` object | `ThermalModel` object

Structural or linear thermal model, specified as a `StructuralModel` object or a `ThermalModel` object. The `linearize` function does not support nonlinear thermal analysis.

`labeltext` — Label for boundary condition

character vector | string

Label for boundary condition, specified as a character vector or a string.

Data Types: `char` | `string`

Output Arguments

`input` — Linearization input description

structure array

Linearization input description, returned as a structure array.

Algorithms

The `linearize` function constructs a linear model whose inputs are a subset of the boundary conditions, loads, or sources applied to the thermal or structural model in Partial Differential Equation Toolbox and whose outputs are the resulting values at the selected DoFs. For example, if you designate the heat source

```
internalHeatSource(model,25,"Face",2,"Label","heatSource")
```

as a linearization input

```
linearizeInput(model,"heatSource")
```

and designate the temperatures on face `X` as linearization outputs

```
linearizeOutput(model,"Face",X)
```

then the response of the linearized model to the constant input $u(t) = 25$ (the heat source value in the thermal model) matches the Partial Differential Equation Toolbox simulation results for face `X`.

```
tlist = 1:10;
u = repmat(25,size(tlist));
ysp = lsim(linsys,uLoad,tlist);
```

Note that loads and boundary conditions not included as linearization inputs are assumed to be zero in the linearized model regardless of their values in the structural or thermal model in Partial Differential Equation Toolbox. Simulation results can differ in this case.

Version History

Introduced in R2021b

See Also

`linearize` | `linearizeOutput` | `structuralBodyLoad` | `structuralBoundaryLoad` | `structuralBC` | `internalHeatSource` | `thermalBC`

linearizeOutput

Package: `pde`

Specify outputs of linearized model

Syntax

```
linearizeOutput(model,RegionType,RegionID)
linearizeOutput(model,RegionType,RegionID,"Component",xyz)
output = linearizeOutput( ___ )
```

Description

`linearizeOutput(model,RegionType,RegionID)` adds all degrees of freedom (DoFs) associated with the region defined by `RegionType` and `RegionID` to the output vector of the linearized model. For 3-D structural models, `linearizeOutput` adds all *x*-coordinates first, then all *y*-coordinates, then all *z*-coordinates. In the linearized model `sys`, use `sys.OutputGroup` to locate the sections associated with each coordinate.

Use `linearizeInput` and `linearizeOutput` together with the `linearize` function to extract sparse linear models from structural and thermal models.

`linearizeOutput(model,RegionType,RegionID,"Component",xyz)` specifies which of the coordinates to include.

`output = linearizeOutput( ___ )` returns a structure array `output` with the linearization output description. Use this syntax with any of the previous arguments.

Examples

Regions for Extracting Sparse Linear Models

Specify the regions of a 2-D thermal model for which `linearize` extracts sparse linear models used in Control System Toolbox™.

Create a transient thermal model.

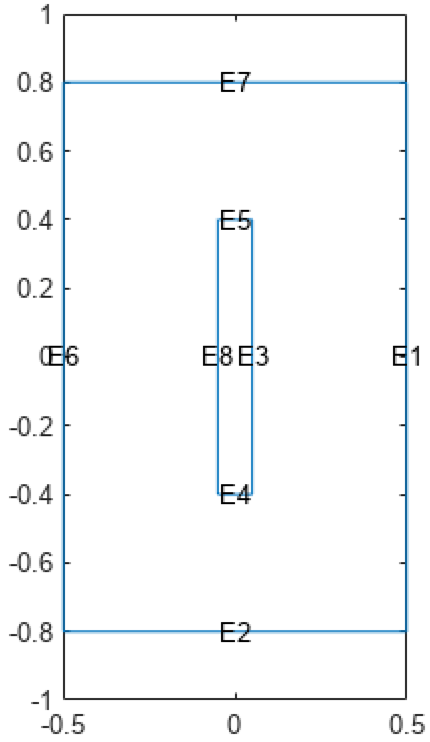
```
thermalmodel = createpde("thermal","transient");
```

Add the block geometry to the thermal model by using the `geometryFromEdges` function. The geometry description file for this problem is called `crackg.m`.

```
geometryFromEdges(thermalmodel,@crackg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
ylim([-1,1])
axis equal
```



Generate a mesh.

```
generateMesh(thermalmodel);
```

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",1, ...
                               "MassDensity",1, ...
                               "SpecificHeat",1);
```

Specify the temperature on the left edge as `100`, and constant heat flow to the exterior through the right edge as `-10`. Add a unique label to each boundary condition.

```
thermalBC(thermalmodel,"Edge",6,"Temperature",100,"Label","TempBC");
thermalBC(thermalmodel,"Edge",1,"HeatFlux",-10,"Label","FluxBC");
```

Specify that the entire geometry generates heat and add a unique label to this assignment.

```
internalHeatSource(thermalmodel,25,"Label","HeatSource");
```

Set an initial value of `0` for the temperature.

```
thermalIC(thermalmodel,0);
```

Call the `linearizeInput` function with the previously defined labels for the boundary conditions and the internal heat source to set the inputs for the `linearize` function. Add one label per function call.

```
linearizeInput(thermalmodel,"HeatSource");
linearizeInput(thermalmodel,"TempBC");
linearizeInput(thermalmodel,"FluxBC");
```

Call the `linearizeOutput` function to specify the regions for which you want `linearize` to extract sparse linear models. Specify one region per function call.

```
linearizeOutput(thermalmodel,"Edge",2)
```

```
ans = struct with fields:
    RegionType: 'Edge'
      RegionID: 2
```

Components of Displacement as Outputs for Linearized Model

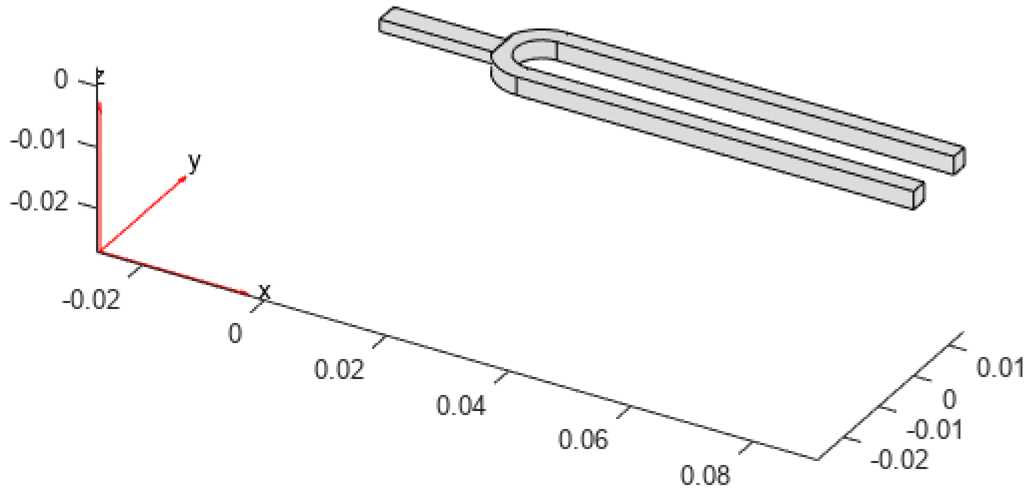
Specify which of the *x*-, *y*-, and *z*- coordinates to include in a linearized model.

Create a structural transient analysis model.

```
structuralmodel = createpde("structural","transient-solid");
```

Import and plot the tuning fork geometry.

```
importGeometry(structuralmodel,"TuningFork.stl");
pdegplot(structuralmodel)
```



Generate a mesh.

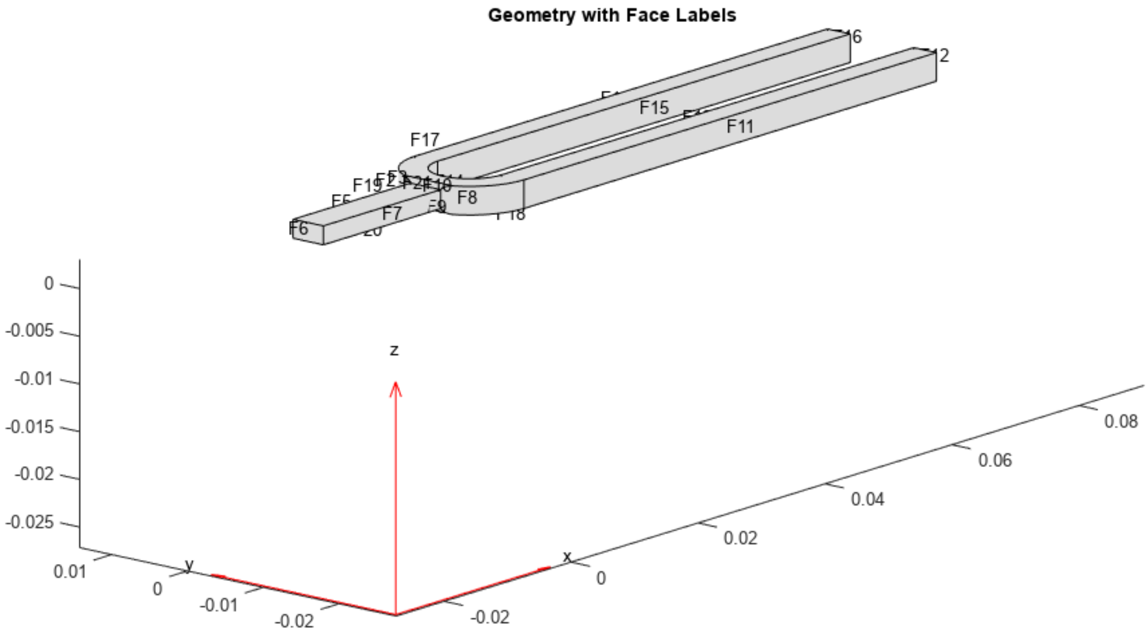
```
generateMesh(structuralmodel,"Hmax",0.005);
```

Specify Young's modulus, Poisson's ratio, and the mass density to model linear elastic material behavior. Specify all physical properties in consistent units.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",8000);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face labels.

```
figure("units","normalized","outerposition",[0 0 1 1])
pdegplot(structuralmodel,"FaceLabels","on")
view(-50,15)
title("Geometry with Face Labels")
```



Impose sufficient boundary constraints to prevent rigid body motion under applied loading. Typically, you hold a tuning fork by hand or mount it on a table. As a simplified approximation of this boundary condition, fix a region near the intersection of the tines and the handle (faces 21 and 22).

```
structuralBC(structuralmodel,"Face",[21,22],"Constraint","fixed");
```

Specify the pressure loading on a tine as a short rectangular pressure pulse.

```
structuralBoundaryLoad(structuralmodel,"Face",11,...
                                       "Pressure",5E6, ...
                                       "EndTime",1e-3, ...
                                       "Label","Pressure");
```

Specify acceleration due to gravity as a body load.

```
structuralBodyLoad(structuralmodel, ...
                    "GravitationalAcceleration",[0 0 -1], ...
                                      "Label","Gravity");
```

Create inputs for gravity and the pressure pulse on tuning fork.

```
linearizeInput(structuralmodel,"Gravity");
linearizeInput(structuralmodel,"Pressure");
```

Measure the *y*-displacement of face 12 and *x*-displacement of face 6.

```
linearizeOutput(structuralmodel,"Face",12,"Component","y")
```

```
ans = struct with fields:
    RegionType: 'Face'
      RegionID: 12
     Component: 'y'
```

```
linearizeOutput(structuralmodel,"Face",6,"Component","x")
```

```
ans = struct with fields:
    RegionType: 'Face'
      RegionID: 6
     Component: 'x'
```

Input Arguments

`model` — Structural or linear thermal model

`StructuralModel` object | `ThermalModel` object

Structural or linear thermal model, specified as a `StructuralModel` object or a `ThermalModel` object. The `linearize` function does not support nonlinear thermal analysis.

RegionType — Geometric region type

`"Cell"` | `"Face"` | `"Edge"` | `"Vertex"`

Geometric region type, specified as `"Cell"` (for a 3-D model only), `"Face"`, `"Edge"`, or `"Vertex"`.

Data Types: `char`

`RegionID` — Geometric region ID
positive integer

Geometric region ID, specified as a positive integer. Find the region IDs by using `pdegplot` with the `"CellLabels"`, `"FaceLabels"`, `"EdgeLabels"`, or `"VertexLabels"` value set to `"on"`.

Data Types: `double`

`xyz` — Coordinates to include
character vector | string

Coordinates to include, specified as a character vector or a string of *x*-, *y*-, and *z*-coordinates to include.

Example: `linearizeOutput(pdemodel,"Face",10,"Component","xz")` selects the *x* and *z* DoFs for face 10

Data Types: `char` | `string`

Output Arguments

`output` — Linearization output description
structure array

Linearization output description, returned as a structure array.

Version History

Introduced in R2021b

See Also

`linearize` | `linearizeInput`

meshQuality

Package: `pde`

Evaluate shape quality of mesh elements

Syntax

```
Q = meshQuality(mesh)
Q = meshQuality(mesh,elemIDs)
Q = meshQuality(___,"aspect-ratio")
```

Description

`Q = meshQuality(mesh)` returns a row vector of numbers from 0 through 1 representing shape quality of all elements of the mesh. Here, 1 corresponds to the optimal shape of the element.

`Q = meshQuality(mesh,elemIDs)` returns the shape quality of the specified elements.

`Q = meshQuality(___,"aspect-ratio")` determines the shape quality by using the ratio of minimal to maximal dimensions of an element. The quality values are numbers from 0 through 1, where 1 corresponds to the optimal shape of the element. Specify `"aspect-ratio"` after any of the previous syntaxes.

Examples

Element Quality of 3-D Mesh

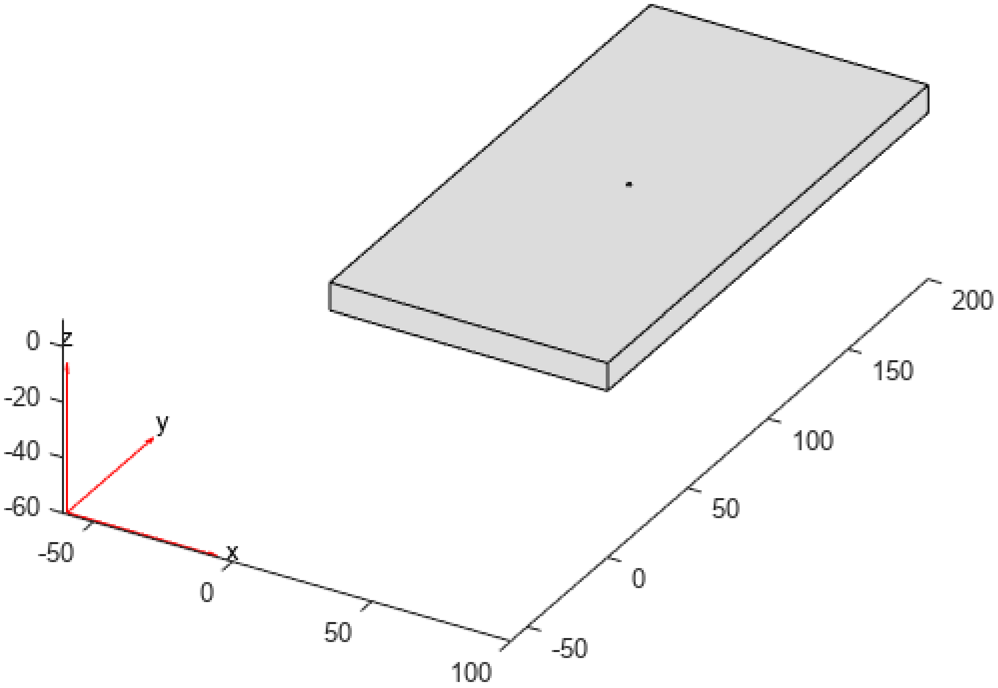
Evaluate the shape quality of the elements of a 3-D mesh.

Create a PDE model.

```
model = createpde;
```

Include and plot the following geometry.

```
importGeometry(model,"PlateSquareHoleSolid.stl");
pdegplot(model)
```

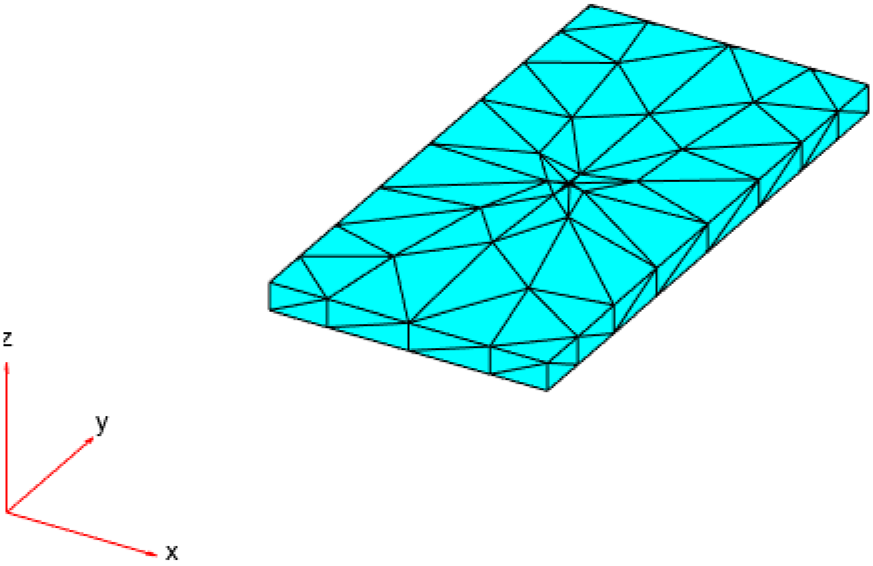


Create and plot a coarse mesh.

```
mesh = generateMesh(model,"Hmax",35)
```

```
mesh = 
  FEMesh with properties:

             Nodes: [3x487 double]
          Elements: [10x213 double]
    MaxElementSize: 35
    MinElementSize: 17.5000
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

```
pdemesh(model)
```



Evaluate the shape quality of all mesh elements. Display the first five values.

```
Q = meshQuality(mesh);
Q(1:5)
```

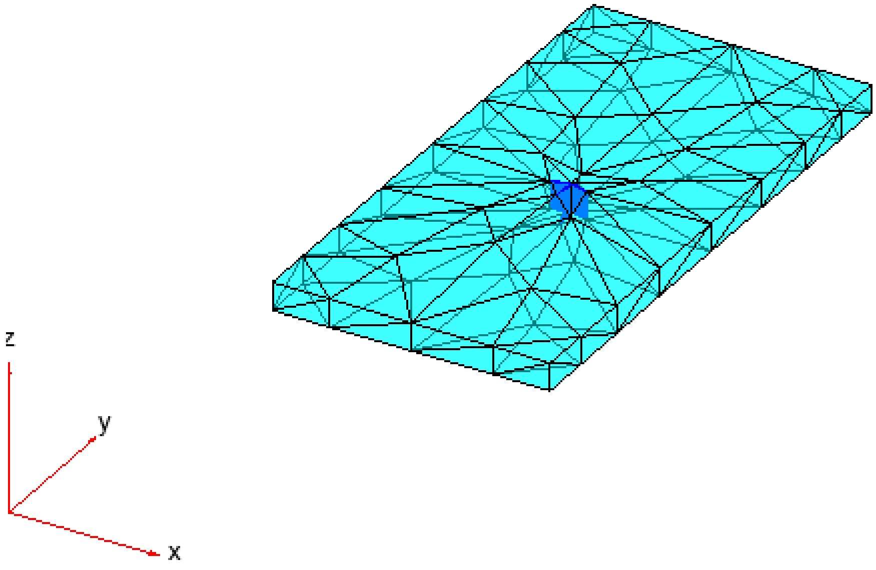
```
ans = 1×5

    0.3079    0.2917    0.6189    0.6688    0.5571
```

Find the elements with the quality values less than 0.2.

```
elemIDs = find(Q < 0.2);
```

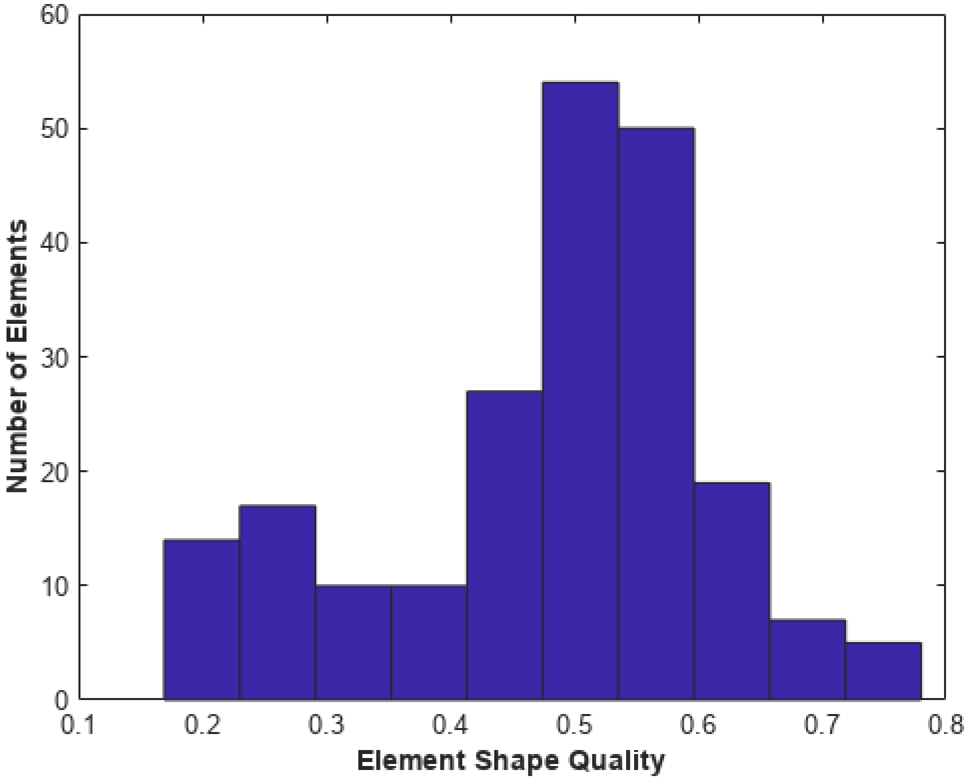
Highlight these elements in blue on the mesh plot.

```
pdemesh(mesh,"FaceAlpha",0.5)
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,elemIDs), ...
                  "FaceColor","blue", ...
                  "EdgeColor","blue")
```



Plot the element quality in a histogram.

```
figure
hist(Q)
xlabel("Element Shape Quality","fontweight","b")
ylabel("Number of Elements","fontweight","b")
```



Find the worst quality value.

```
Qworst = min(Q)
```

```
Qworst = 0.1691
```

Find the corresponding element IDs.

```
elemIDs = find(Q==Qworst)
```

```
elemIDs = 1×2

    10   136
```

Element Quality of 2-D Mesh

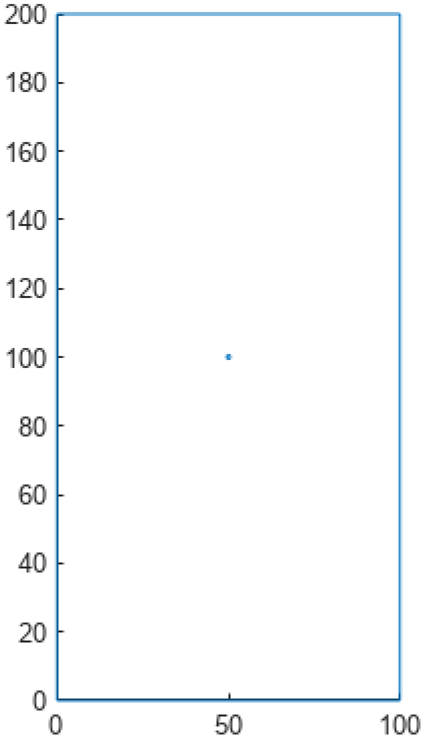
Evaluate the shape quality of the elements of a 2-D mesh.

Create a PDE model.

```
model = createpde;
```

Include and plot the following geometry.

```
importGeometry(model,"PlateSquareHolePlanar.stl");
pdegplot(model)
```

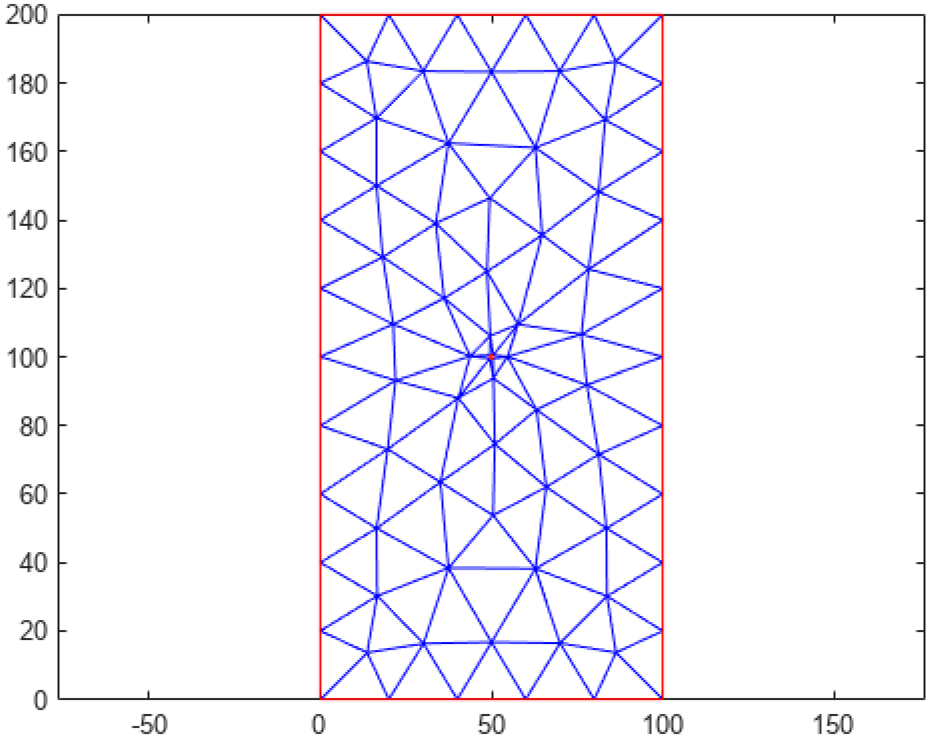


Create and plot a coarse mesh.

```
mesh = generateMesh(model,"Hmax",20)
```

```
mesh = 
  FEMesh with properties:

             Nodes: [2x286 double]
          Elements: [6x126 double]
    MaxElementSize: 20
    MinElementSize: 10
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

```
pdemesh(model)
```



Find the IDs of the elements within a box enclosing the center of the plate.

```
elemIDs = findElements(mesh,"box",[25,75],[80,120]);
```

Evaluate the shape quality of these elements. Display the result as a column vector.

```
Q = meshQuality(mesh,elemIDs);
Q.'
```

```
ans = 12×1

    0.2980
    0.8253
    0.2994
    0.6581
    0.7838
    0.6104
    0.3992
    0.6921
    0.2948
    0.5726
      ⋮
```

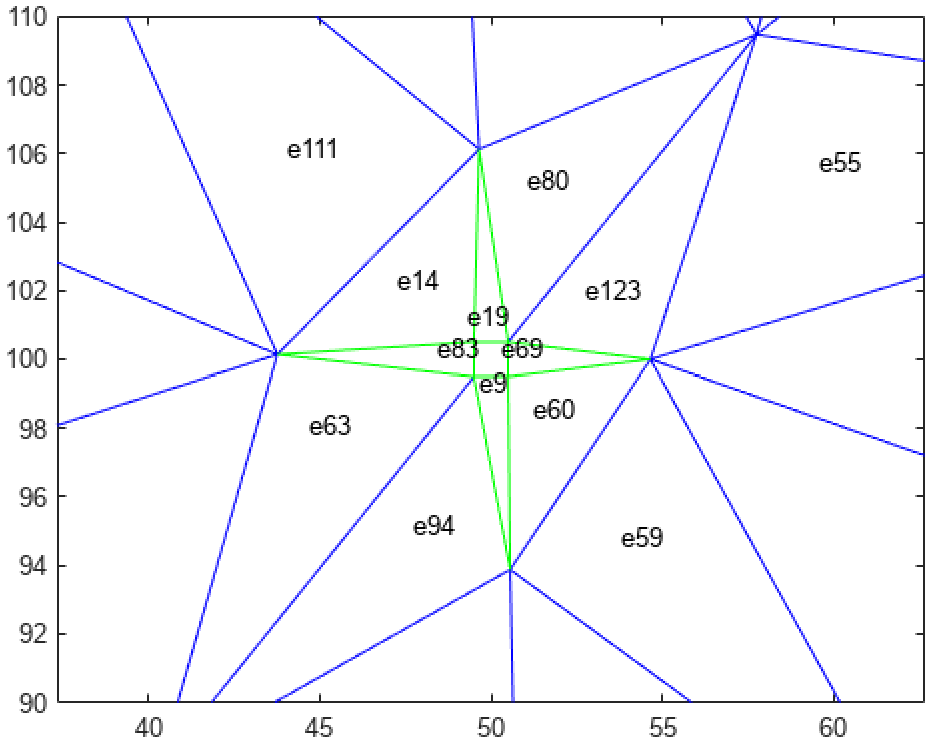
Find the elements with the quality values less than 0.4.

```
elemIDs04 = elemIDs(Q < 0.4)
```

```
elemIDs04 = 1×4

     9    19    69    83
```

Highlight these elements in green on the mesh plot. Zoom in to see the details.

```
pdemesh(mesh,"ElementLabels","on")
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,elemIDs04),"EdgeColor","green")
zoom(10)
```



Element Quality Determined by Aspect Ratio

Determine the shape quality of mesh elements by using the ratios of minimal to maximal dimensions.

Create a PDE model and include the L-shaped geometry.

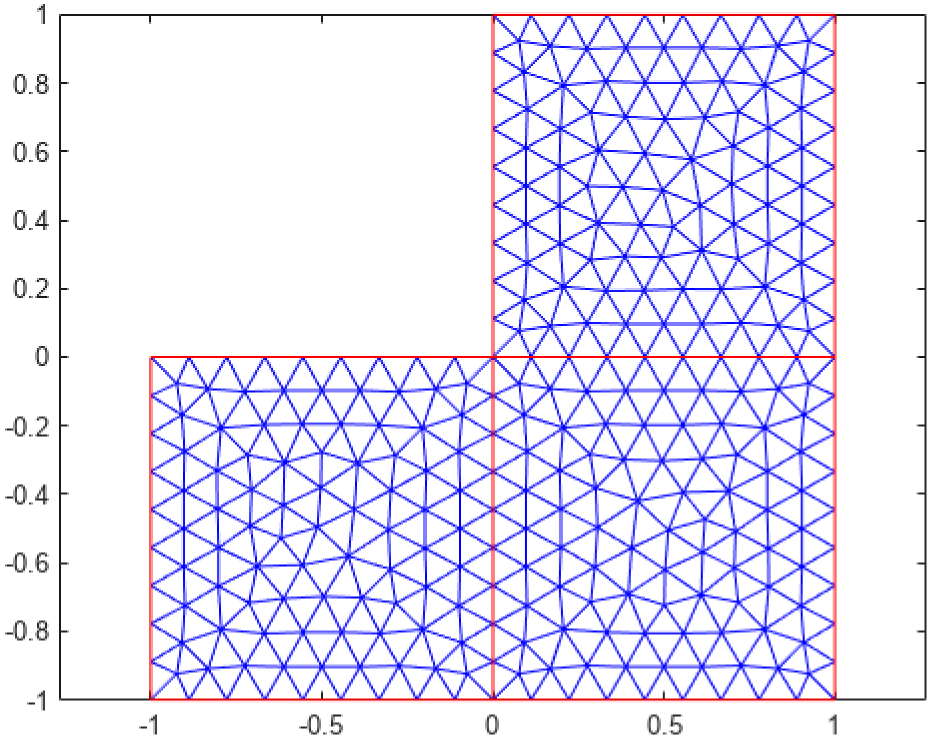
```
model = createpde(1);
geometryFromEdges(model,@lshapeg);
```

Generate the default mesh for the geometry.

```
mesh = generateMesh(model);
```

View the mesh.

```
pdeplot(model)
```



Evaluate the shape quality of mesh elements by using the minimal to maximal dimensions ratio. Display the first five values.

```
Q = meshQuality(mesh,"aspect-ratio");
Q(1:5)
```

```
ans = 1×5

    0.8339    0.7655    0.7755    0.8301    0.8969
```

Evaluate the shape quality of mesh elements by using the default setting. Display the first five values.

```
Q = meshQuality(mesh);
Q(1:5)
```

```
ans = 1×5

    0.9837    0.9605    0.9654    0.9829    0.9913
```

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`

`elemIDs` — Element IDs
positive integer | matrix of positive integers

Element IDs, specified as a positive integer or a matrix of positive integers.

Example: `[10 68 81 97 113 130 136 164]`

Output Arguments

`Q` — Shape quality of mesh elements
row vector of numbers from 0 through 1

Shape quality of mesh elements, returned as a row vector of numbers from 0 through 1. The value 0 corresponds to a deflated element with zero area or volume. The value 1 corresponds to an element of optimal shape.

Example: [0.9150 0.7787 0.9417 0.2744 0.9843 0.9181]

Data Types: `double`

Version History

Introduced in R2018a

References

[1] Knupp, Patrick M. "Matrix Norms & the Condition Number: A General Framework to Improve Mesh Quality via Node-Movement." In Proceedings, 8th International Meshing Roundtable. Lake Tahoe, CA, October 1999: 13-22.

See Also

`findElements` | `findNodes` | `area` | `volume` | FEMesh Properties

Topics

“Finite Element Method Basics” on page 1-11

meshToPet

Package: `pde`

`[p,e,t]` representation of `FEMesh` data

Note 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.

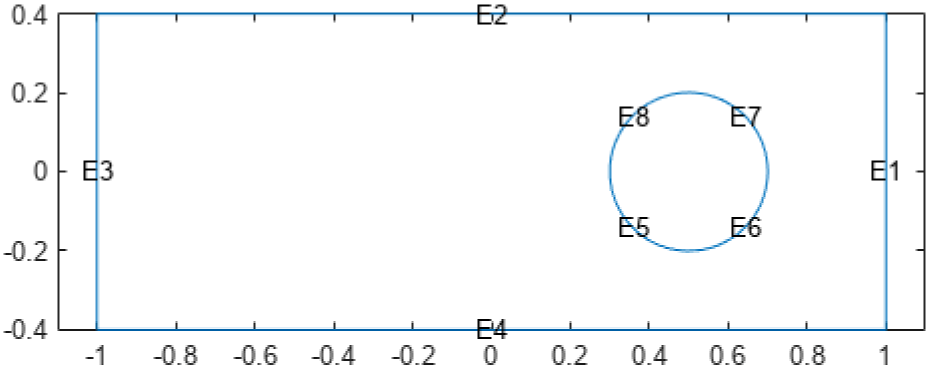
Examples

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.

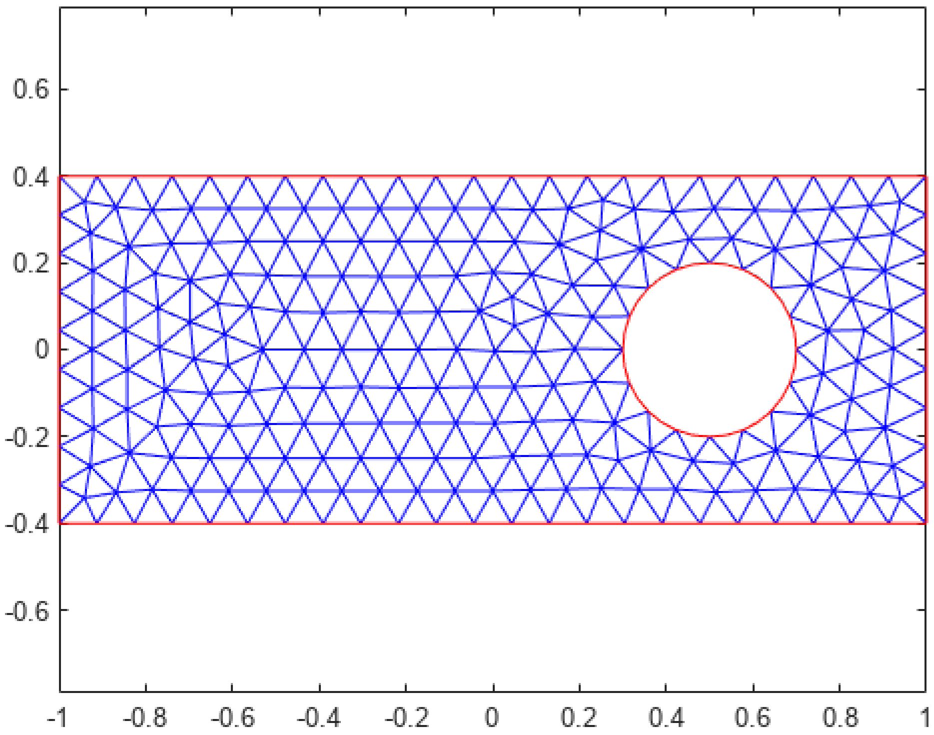
Create a 2-D PDE geometry and incorporate it into a model object. View the geometry.

```
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 = 1×2

     2   956
```

```
size(e)
```

```
ans = 1×2

     7   160
```

```
size(t)
```

```
ans = 1×2

     7   438
```

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 2-175.

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 2-175.

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 2-175. 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`.

Version History

Introduced in R2015a

See Also

FEMesh | `generateMesh`

Topics

“Mesh Data” on page 2-175

multicuboid

Create geometry formed by several cubic cells

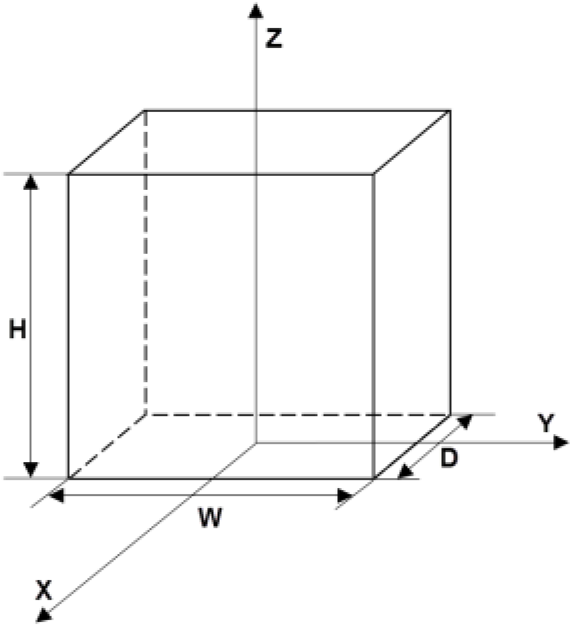
Syntax

```
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 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
       Vertices: [24x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

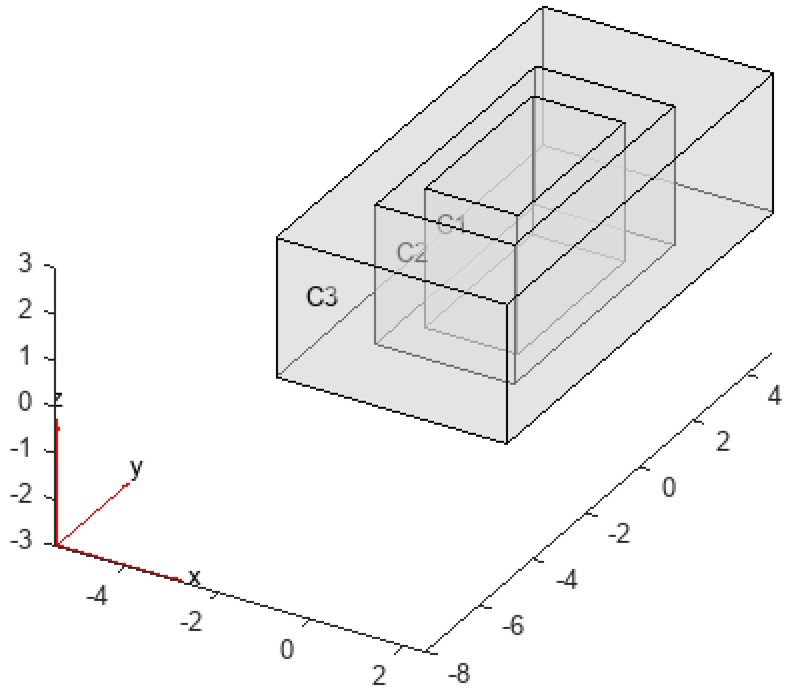
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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
       Vertices: [20x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

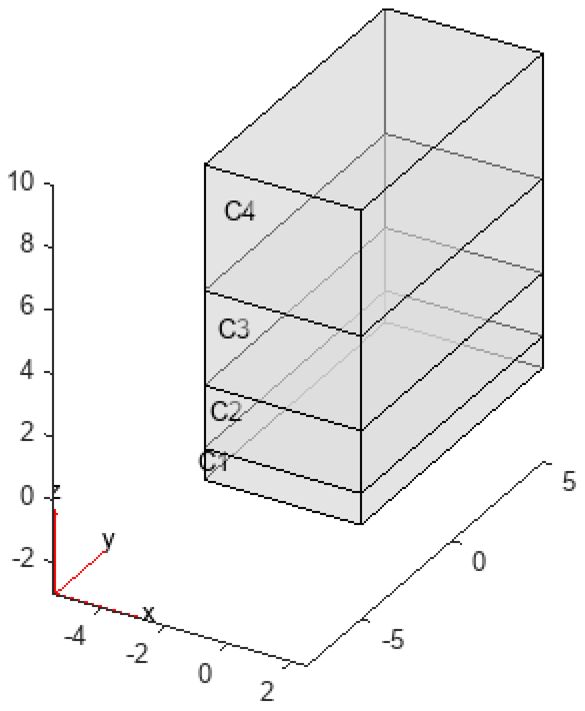
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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
       Vertices: [8x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

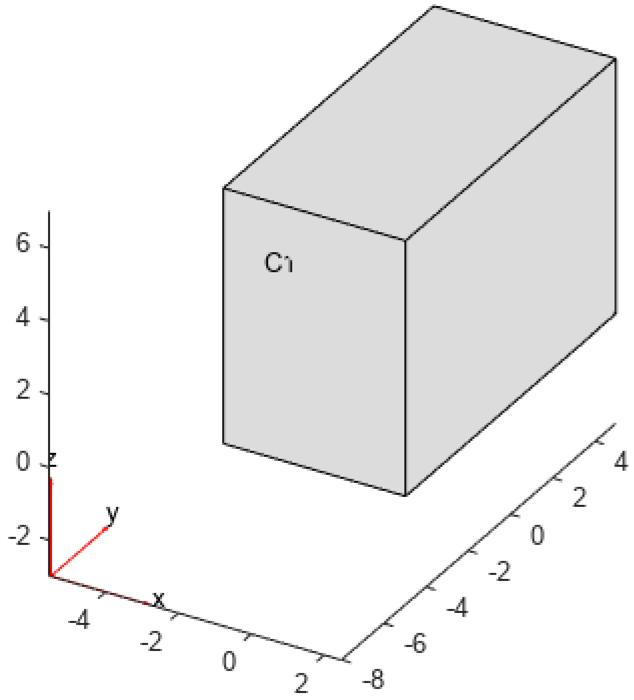
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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
       Vertices: [16x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

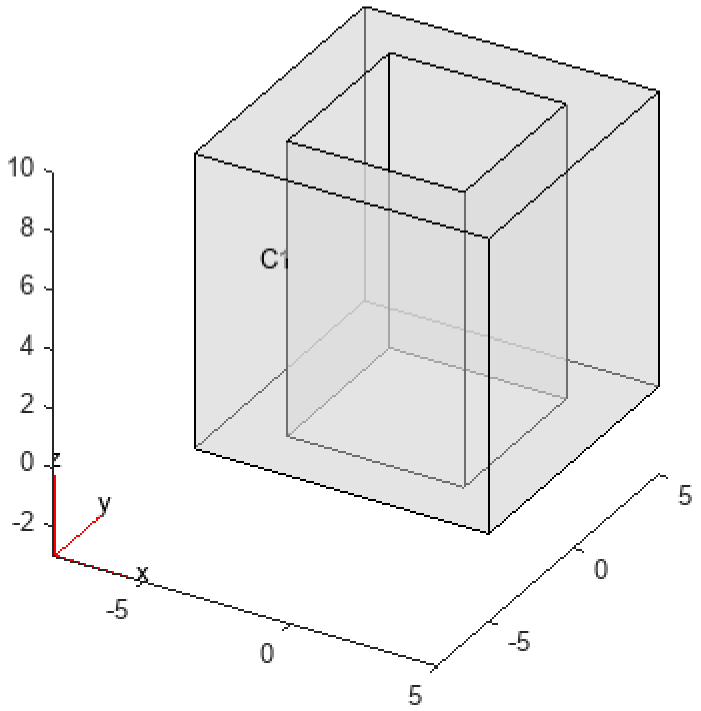
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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 `i`th 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 `i`th 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 `i`th 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 `i`th 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.

Version History

Introduced in R2017a

See Also

`multicylinder` | `multisphere` | DiscreteGeometry

multicylinder

Create geometry formed by several cylindrical cells

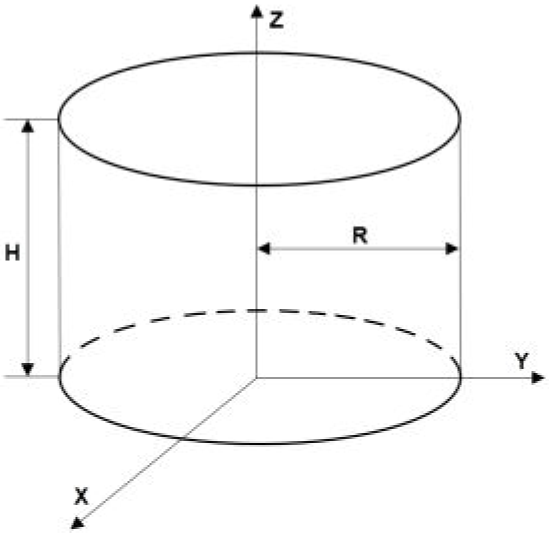
Syntax

```
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 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
       Vertices: [6x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

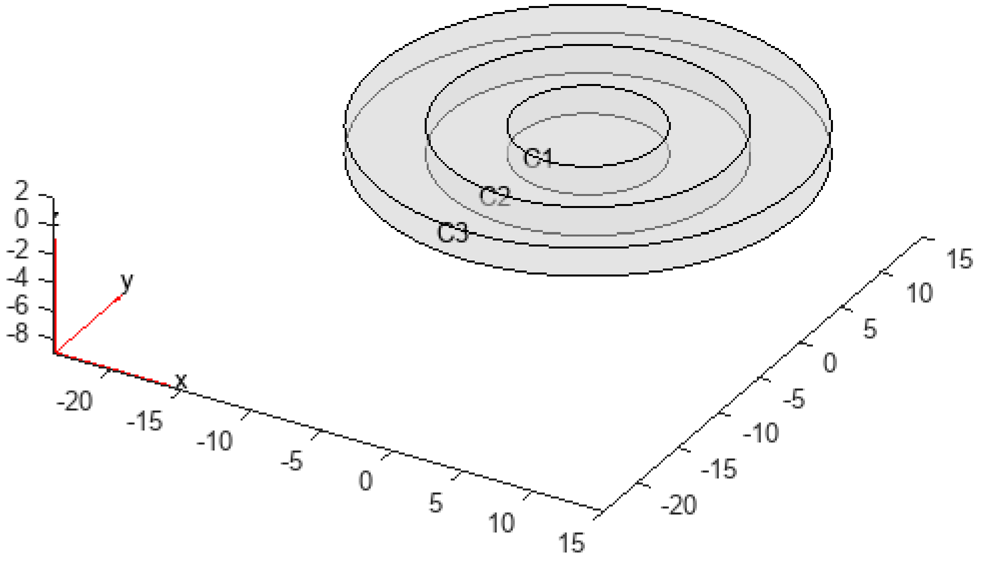
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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
       Vertices: [5x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
             PDESystemSize: 1
           IsTimeDependent: 0
                  Geometry: []
      EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                      Mesh: []
             SolverOptions: [1x1 pde.PDESolverOptions]
```

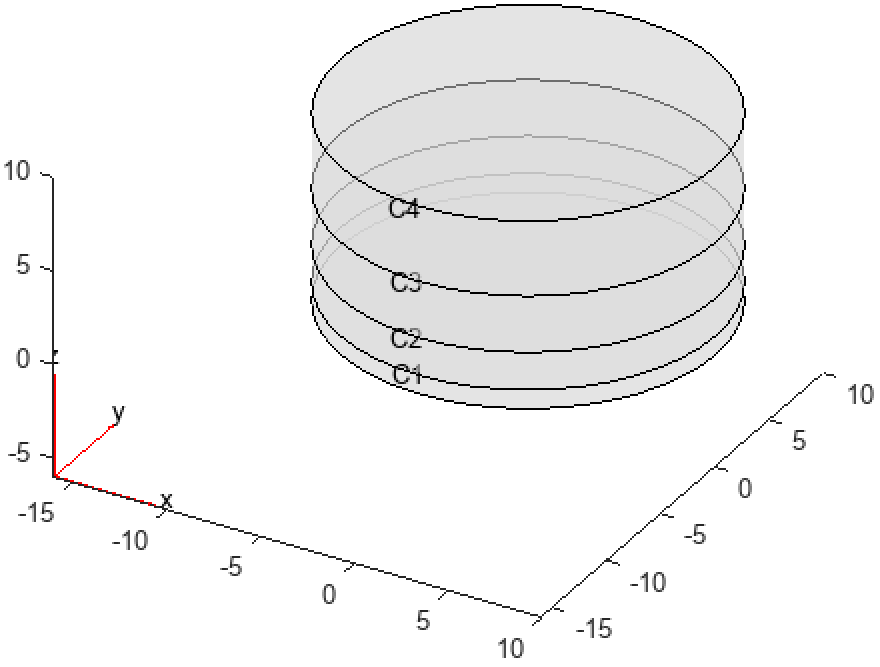
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

             PDESystemSize: 1
           IsTimeDependent: 0
                  Geometry: [1x1 DiscreteGeometry]
      EquationCoefficients: []
        BoundaryConditions: []
         InitialConditions: []
                      Mesh: []
             SolverOptions: [1x1 pde.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
       Vertices: [2x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

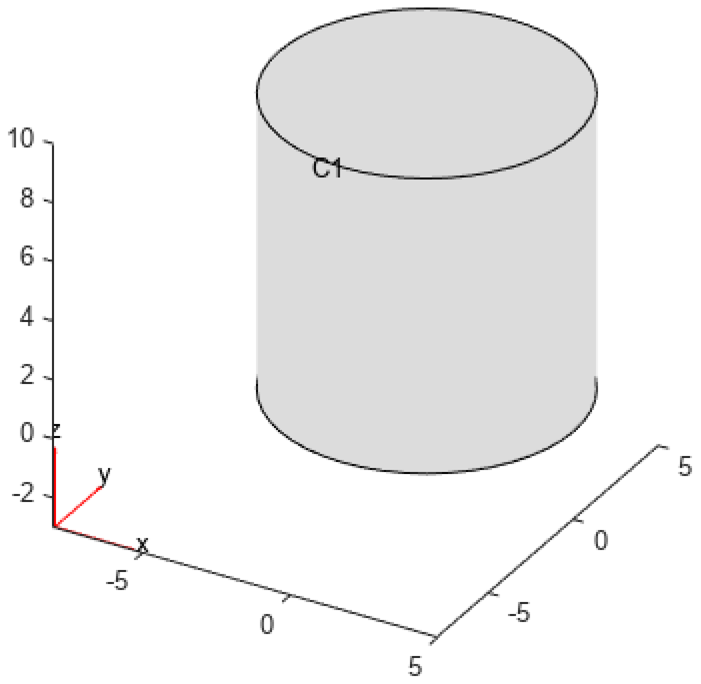
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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
       Vertices: [4x3 double]
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

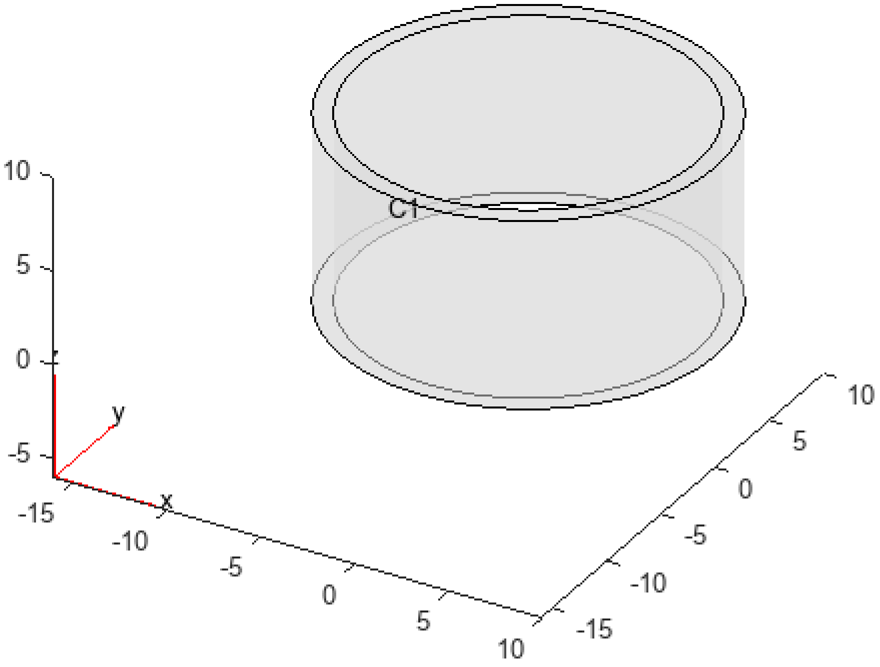
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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 `i`th 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)`

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 `i`th 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],"Zoffset",[0 1 3])`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 `i`th 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.

Tip A cylinder has one cell, three faces, and two edges. Also, since every edge has a start and an end vertex, a cylinder has vertices. Both edges are circles, their start and end vertices coincide. Thus, a cylinder has two vertices - one for each edge.

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.

Version History

Introduced in R2017a

See Also

`multicuboid` | `multisphere` | DiscreteGeometry

multisphere

Create geometry formed by several spherical cells

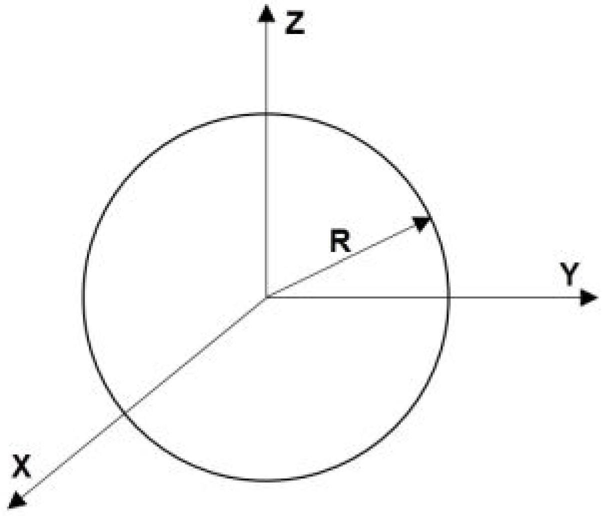
Syntax

```
gm = multisphere(R)
gm = multisphere(R,"Void",eci)
```

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,"Void",eci)` creates a multi-sphere geometry with empty cells.

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: 0
    NumVertices: 0
       Vertices: []
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

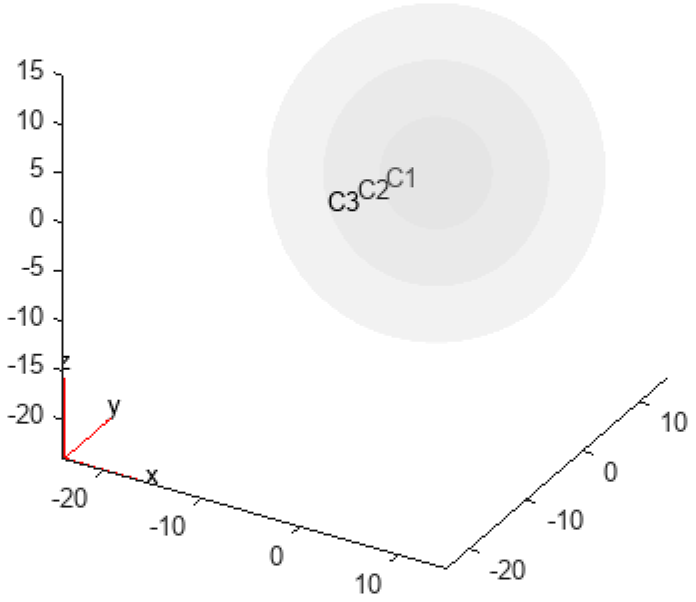
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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: 0
    NumVertices: 0
       Vertices: []
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

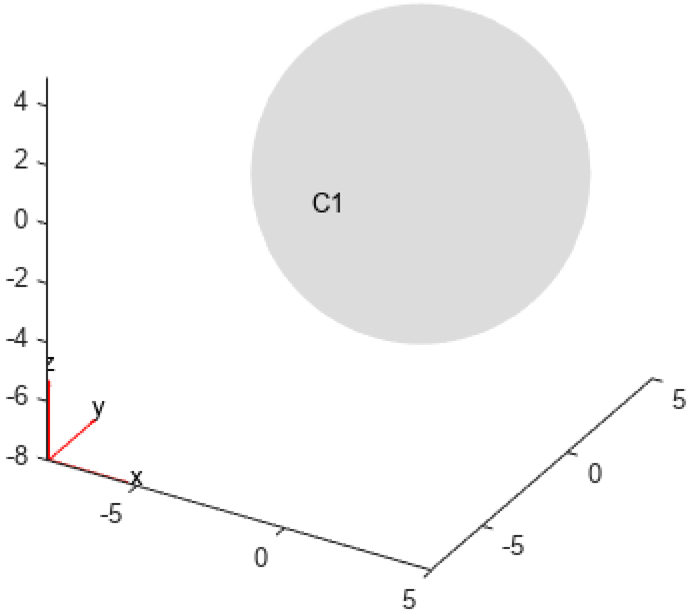
Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.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
       Vertices: []
```

Create a PDE model.

```
model = createpde
```

```
model = 
  PDEModel with properties:
```

```
           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: []
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

Include the geometry in the model.

```
model.Geometry = gm
```

```
model = 
  PDEModel with properties:

           PDESystemSize: 1
         IsTimeDependent: 0
                Geometry: [1x1 DiscreteGeometry]
    EquationCoefficients: []
      BoundaryConditions: []
       InitialConditions: []
                    Mesh: []
           SolverOptions: [1x1 pde.PDESolverOptions]
```

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 `i`th cell.

Example: `gm = multisphere([1,2,3])`

`eci` — Empty cell indicator

vector of logical `true` or `false` values

Empty cell indicator, specified as a vector of logical `true` and `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])`

Output Arguments

`gm` — Geometry object

`DiscreteGeometry` object

Geometry object, returned as a DiscreteGeometry object.

Version History

Introduced in R2017a

See Also

`multicuboid` | `multicylinder` | DiscreteGeometry

Topics

“Heat Conduction in Multidomain Geometry with Nonuniform Heat Flux” on page 3-270

nearestEdge

Find edges nearest to specified point

Syntax

`EdgeID = nearestEdge(g,Coords)`

Description

`EdgeID = nearestEdge(g,Coords)` finds edges nearest to the point with the coordinates `Coords`.

Examples

Edges of 3-D Geometry Closest to Specified Points

Find edges of a block nearest to the specified points.

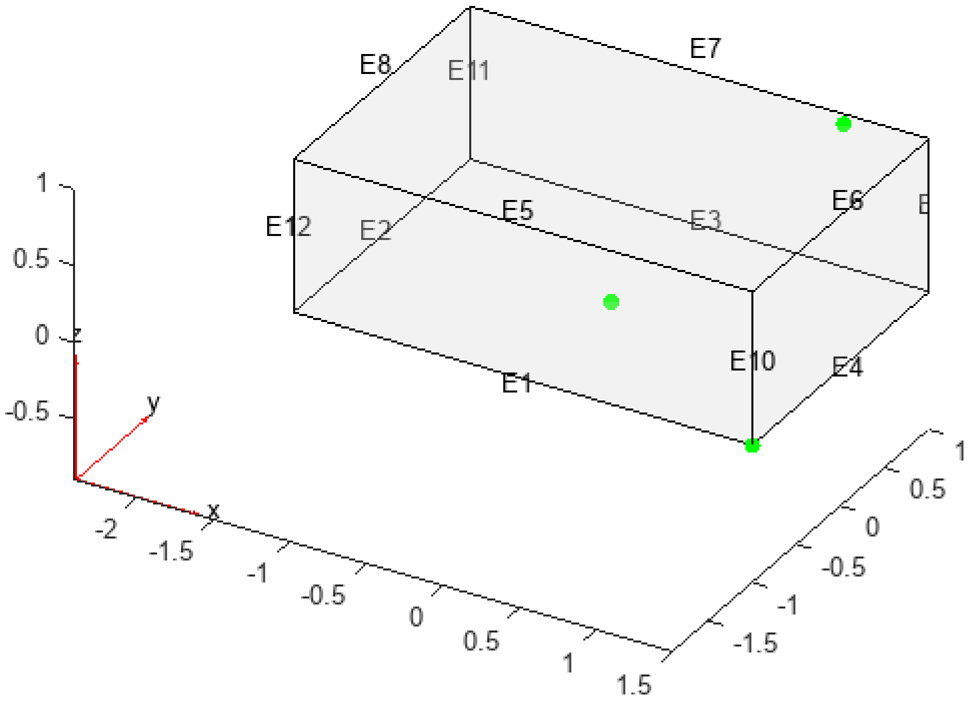
Create a block geometry.

```
gm = multicuboid(3,2,1)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 6
       NumEdges: 12
    NumVertices: 8
       Vertices: [8x3 double]
```

Plot the geometry with the edge labels. Add the points with the coordinates (0 0 0), (1 0.9 1), and (1.5 -1 0) to the plot.

```
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.2)
hold on
scatter3([0 1 1.5],[0 0.9 -1],[0 1 0],"filled","MarkerFaceColor","g")
```



Find edges closest to the points with the coordinates (0 0 0), (1 0.9 1), and (1.5 -1 0). If several edges are equally close (within the tolerance) to the point, `nearestEdge` returns the ID of one of the edges.

```
edgeIDs = nearestEdge(gm,[0 0 0; 1 0.9 1; 1.5 -1 0])
```

```
edgeIDs = 1×3

     1     7     1
```

Edges of 2-D Geometry Closest to Specified Points

Find edges of the L-shaped membrane nearest to the specified points.

Create a model and include this geometry. The geometry of the L-shaped membrane is described in the file `lshapeg`.

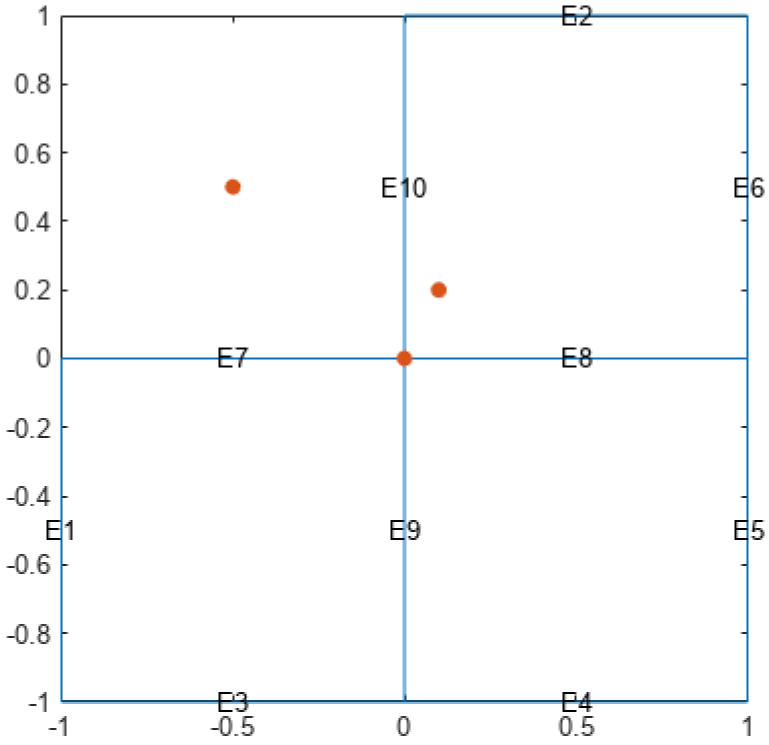
```
model = createpde();
gm = geometryFromEdges(model,@lshapeg)
```

```
gm = 
  AnalyticGeometry with properties:

       NumCells: 0
       NumFaces: 3
```

```
       NumEdges: 10
    NumVertices: 8
       Vertices: [8x2 double]
```

Plot the geometry with the edge labels. Add the points with the coordinates (0 0), (0.1 0.2), and (-0.5 0.5) to the plot.

```
pdegplot(gm,"EdgeLabels","on")
hold on
scatter([0 0.1 -0.5],[0 0.2 0.5],"filled")
```



Find edges closest to the points with the coordinates (0 0), (0.1 0.2), and (-0.5 0.5). If several edges are equally close (within the tolerance) to the point, `nearestEdge` returns the ID of one of the edges.

```
edgeIDs = nearestEdge(gm,[0 0; 0.1 0.2; -0.5 0.5])
```

```
edgeIDs = 1×3

     9    10    10
```

Input Arguments

g — Geometry
`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

`Coords` — Edge ID
2-element vector | 3-element vector

Coordinates of the point, specified as a vector of 2 or 3 elements for a 2-D or 3-D geometry, respectively.

Data Types: `double`

Output Arguments

`EdgeID` — IDs of edges nearest to specified point
positive number | vector of positive numbers

IDs of edges nearest to the specified point, returned as a positive number or a vector of positive numbers.

Version History

Introduced in R2021a

See Also

`cellEdges` | `cellFaces` | `faceEdges` | `facesAttachedToEdges` | `nearestFace` | DiscreteGeometry Properties | AnalyticGeometry Properties

nearestFace

Find faces nearest to specified point

Syntax

`FaceID = nearestFace(g,Coords)`

Description

`FaceID = nearestFace(g,Coords)` finds faces nearest to the point with the coordinates `Coords`.

Examples

Faces of 3-D Geometry Closest to Specified Points

Find faces of a block nearest to the specified points.

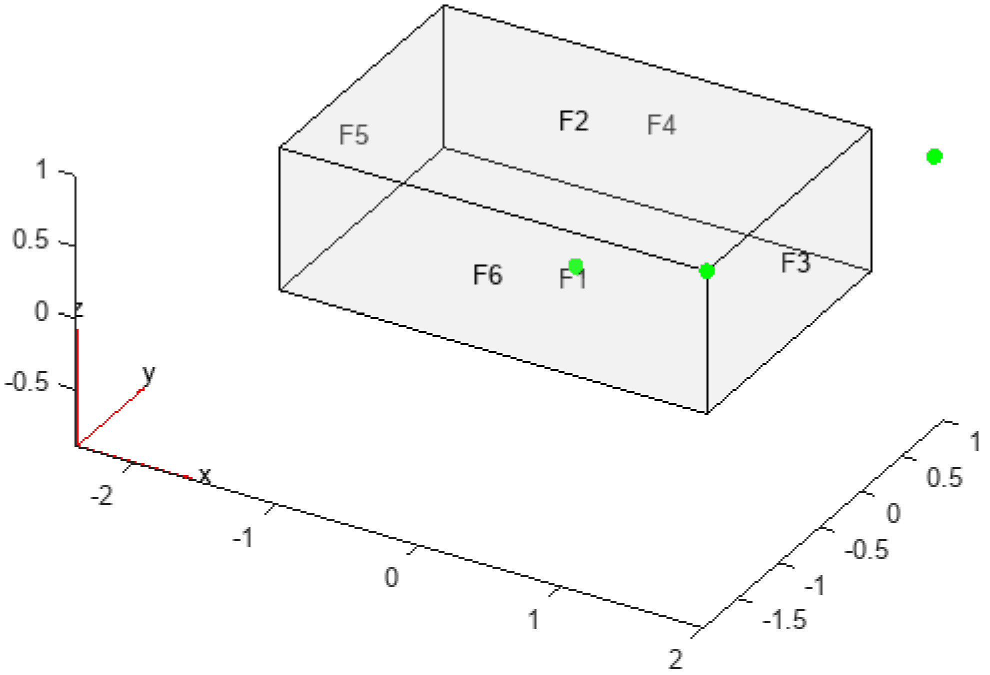
Create a block geometry.

```
gm = multicuboid(3,2,1)
```

```
gm = 
  DiscreteGeometry with properties:

       NumCells: 1
       NumFaces: 6
       NumEdges: 12
    NumVertices: 8
       Vertices: [8x3 double]
```

Plot the geometry with the face labels. Add the points with the coordinates (0 0 0.1), (2 0.9 1), and (1.5 -1 1) to the plot.

```
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.2)
hold on
scatter3([0 2 1.5],[0 0.9 -1],[0.1 1 1],"filled","MarkerFaceColor","g")
```



Find faces closest to the points with the coordinates (0 0 0.1), (2 0.9 1), and (1.5 -1 1). If several faces are equally close (within the tolerance) to the point, `nearestFace` returns the ID of one of the faces.

```
faceIDs = nearestFace(gm,[0 0 0; 2 0.9 1; 1.5 -1 1])
```

```
faceIDs = 1×3

     1     3     2
```

Faces of 2-D Geometry Closest to Specified Points

Find faces of the L-shaped membrane nearest to the specified points.

Create a model and include this geometry. The geometry of the L-shaped membrane is described in the file `lshapeg`.

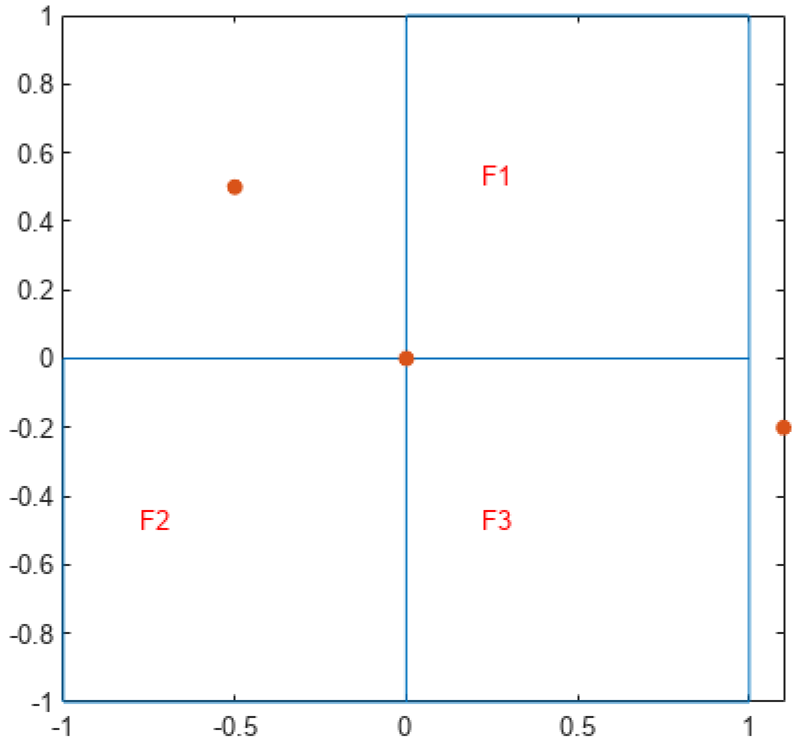
```
model = createpde();
gm = geometryFromEdges(model,@lshapeg)
```

```
gm = 
  AnalyticGeometry with properties:

       NumCells: 0
       NumFaces: 3
```

```
       NumEdges: 10
    NumVertices: 8
       Vertices: [8x2 double]
```

Plot the geometry with the face labels. Add the points with the coordinates (0 0), (1.1 -0.2), and (-0.5 0.5) to the plot.

```
pdegplot(gm,"FaceLabels","on")
hold on
scatter([0 1.1 -0.5],[0 -0.2 0.5],"filled")
```



Find faces closest to the points with the coordinates (0 0), (1.1 -0.2), and (-0.5 0.5). If several faces are equally close (within the tolerance) to the point, `nearestFace` returns the ID of one of the faces.

```
faceIDs = nearestFace(gm,[0 0; 1.1 -0.2; -0.5 0.5])
```

```
faceIDs = 1×3

     2     3     1
```

Input Arguments

g — Geometry

`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

`Coords` — Edge ID
2-element vector | 3-element vector

Coordinates of the point, specified as a vector of 2 or 3 elements for a 2-D or 3-D geometry, respectively.

Data Types: `double`

Output Arguments

`FaceID` — IDs of faces nearest to specified point
positive number | vector of positive numbers

IDs of faces nearest to the specified point, returned as a positive number or a vector of positive numbers.

Version History

Introduced in R2021a

See Also

`cellEdges` | `cellFaces` | `faceEdges` | `facesAttachedToEdges` | `nearestEdge` | `nearestFace` | DiscreteGeometry Properties | AnalyticGeometry Properties

parabolic

(Not recommended) Solve parabolic PDE problem

Note `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

Parabolic equation solver

Solves PDE problems of the type

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

on a 2-D or 3-D region Ω, or the system PDE problem

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

The variables *c*, *a*, *f*, and *d* can depend on position, time, and the solution *u* and its gradient.

`u = parabolic(u0,tlist,model,c,a,f,d)` produces the solution to the FEM formulation of the scalar PDE problem

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

on a 2-D or 3-D region Ω, or the system PDE problem

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

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

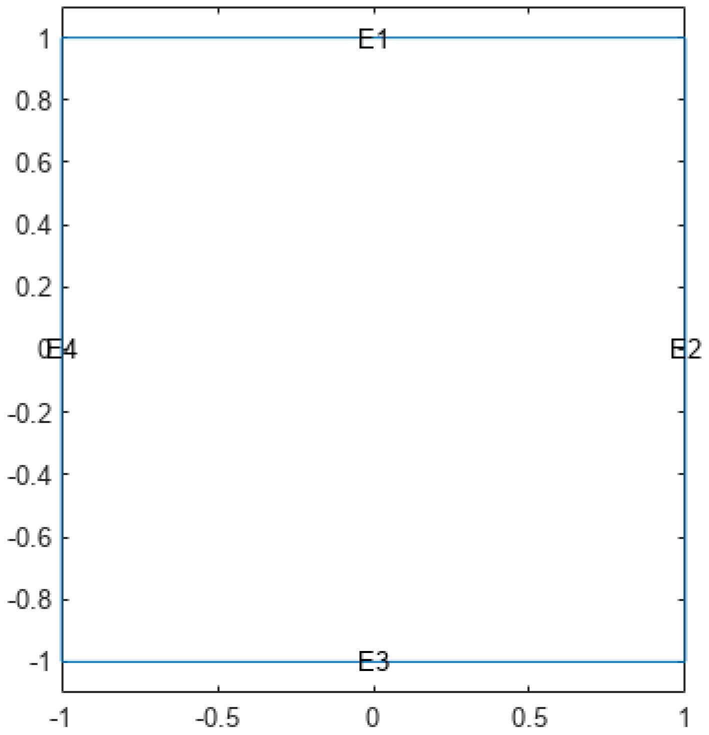
Solve the parabolic equation

$$\frac{\partial u}{\partial t} = \Delta u$$

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,'GeometricOrder','linear');
```

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;
```

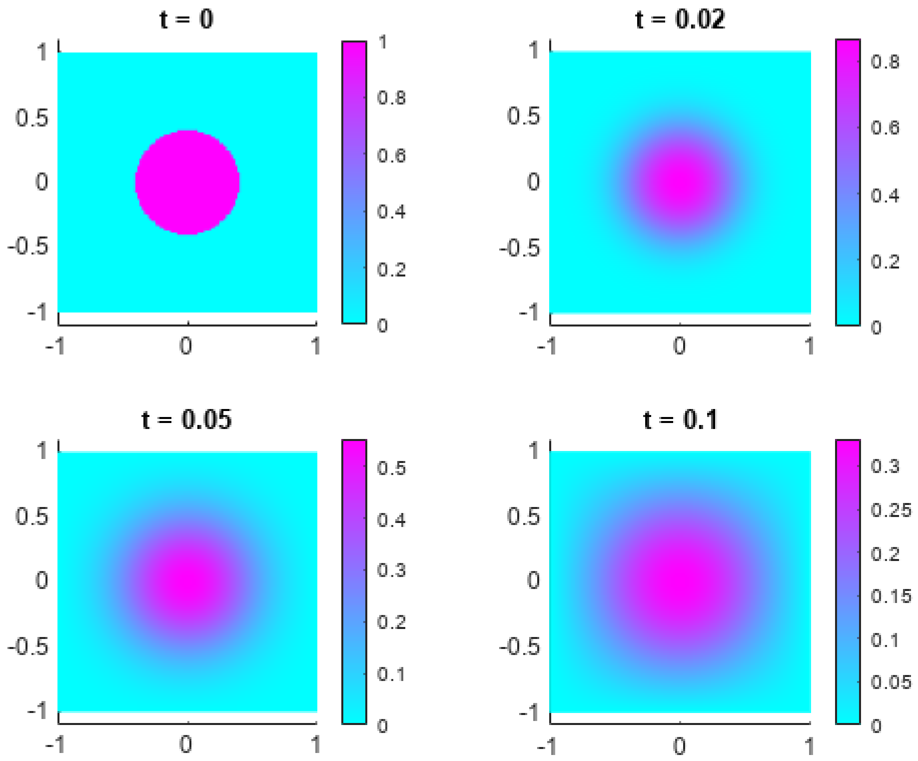
Solve the PDE.

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

```
133 successful steps
0 failed attempts
268 function evaluations
1 partial derivatives
26 LU decompositions
267 solutions of linear systems
```

Plot the initial condition, the solution at the final time, and two intermediate solutions.

```
figure
subplot(2,2,1)
pdeplot(model,'XYData',u(:,1));
axis equal
title('t = 0')
subplot(2,2,2)
pdeplot(model,'XYData',u(:,5))
axis equal
title('t = 0.02')
subplot(2,2,3)
pdeplot(model,'XYData',u(:,11))
axis equal
title('t = 0.05')
subplot(2,2,4)
pdeplot(model,'XYData',u(:,end))
axis equal
title('t = 0.1')
```



Parabolic Equation Using Legacy Syntax

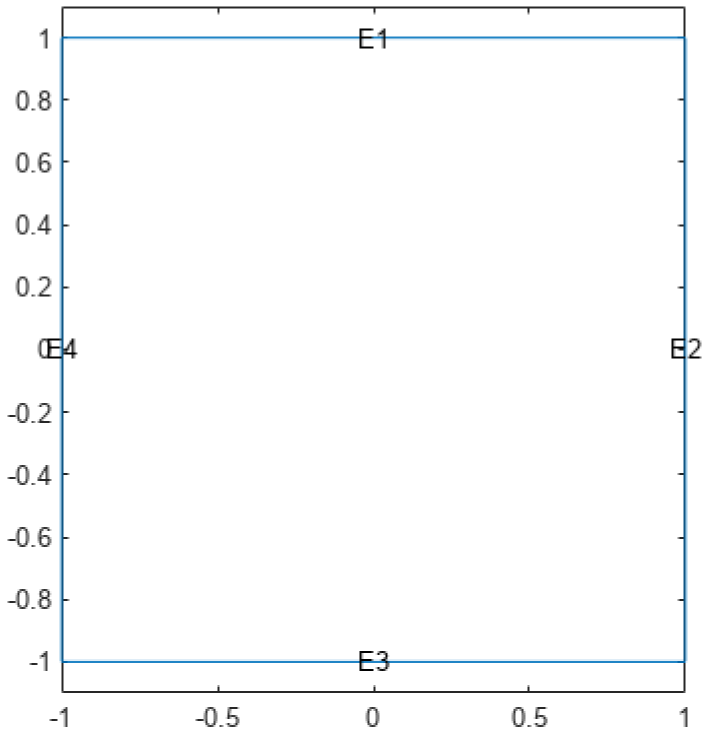
Solve the parabolic equation

$$\frac{\partial u}{\partial t} = \Delta u$$

on the square domain specified by `squareg`, using a geometry function to specify the geometry, a boundary function to specify the boundary conditions, and using `initmesh` to create the finite element mesh.

Specify the geometry as `@squareg` and plot the geometry.

```
g = @squareg;
pdegplot(g,'EdgeLabels','on')
ylim([-1.1,1.1])
axis equal
```



Set Dirichlet boundary conditions $u = 0$ on all edges. The `squareb1` function specifies these boundary conditions.

```
b = @squareb1;
```

Generate a relatively fine mesh.

```
[p,e,t] = initmesh(g,'Hmax',0.02);
```

Set the initial condition to have $u(0) = 1$ on the disk $x^2 + y^2 \le 0.4^2$ 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;
```

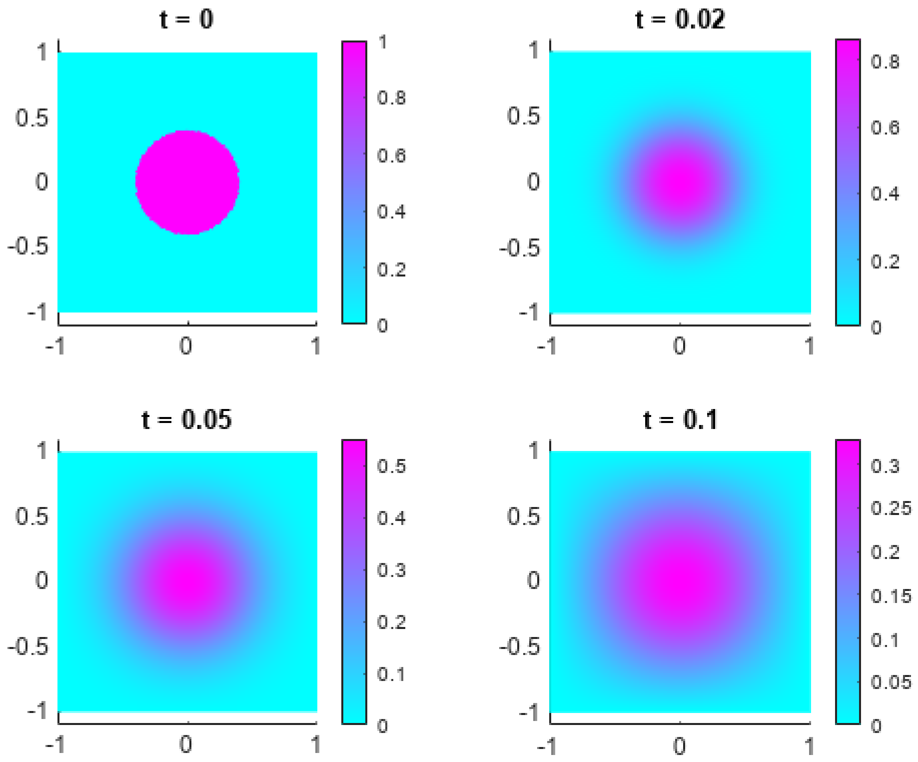
Solve the PDE.

```
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')
```

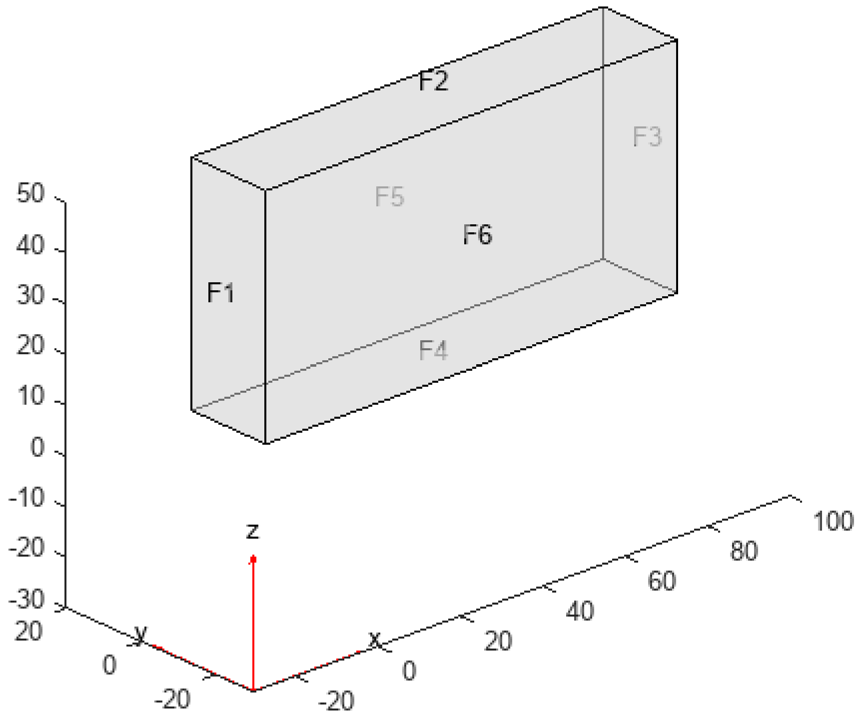


Parabolic Problem Using Matrix Coefficients

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.

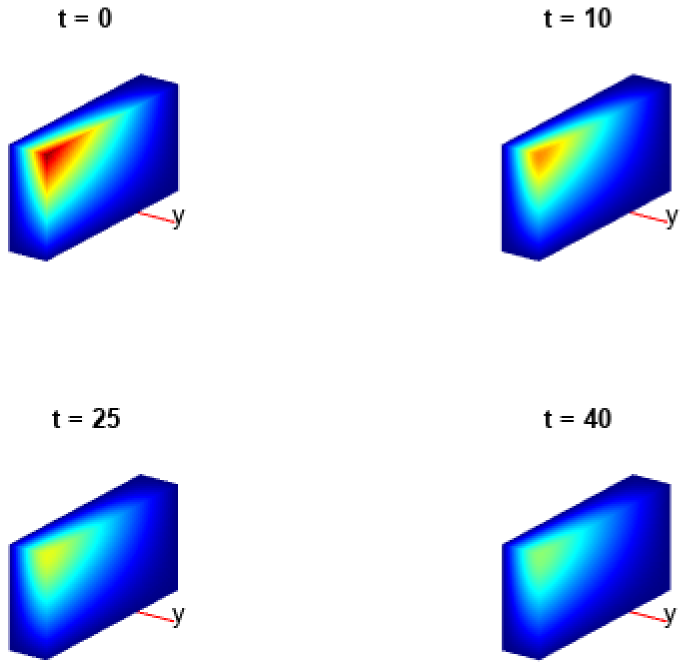
```
tlist = linspace(0,40,41);
u = parabolic(u0,tlist,Kc,Fc,B,ud,M);
```

```
35 successful steps
0 failed attempts
72 function evaluations
1 partial derivatives
11 LU decompositions
71 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 | character vector | character array | string scalar | string vector

Initial condition, specified as a scalar, vector of nodal values, character vector, character array, string scalar, or string vector. 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`.

- 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` | `string`
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`

`c` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `c` represents the c coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

`a` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `a` represents the a coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

`f` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

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

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

`d` — PDE coefficient

scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

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

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `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 Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples" on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples" on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 5-101. Typically, `Kc` is the output of `assempde`.

Fc — Load vector

vector

Load vector, specified as a vector. See "Elliptic Equations" on page 5-101. Typically, `Fc` is the output of `assempde`.

B — Dirichlet nullspace

sparse matrix

Dirichlet nullspace, returned as a sparse matrix. See "Algorithms" on page 5-101. Typically, `B` is the output of `assempde`.

ud — Dirichlet vector

vector

Dirichlet vector, returned as a vector. See "Algorithms" on page 5-101. 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 5-101.

To obtain the input matrices for `pdeeig`, `hyperbolic` or `parabolic`, run both `assema` and `assempde`:

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Note Create the `M` matrix using `assema` with `d`, not `a`, as the argument before `f`.

Data Types: `double`
Complex Number Support: Yes

`rtol` — Relative tolerance for ODE solver

`1e-3` (default) | positive real

Relative tolerance for ODE solver, specified as a positive real.

Example: `2e-4`

Data Types: `double`

`atol` — Absolute tolerance for ODE solver

`1e-6` (default) | positive real

Absolute tolerance for ODE solver, specified as a positive real.

Example: `2e-7`

Data Types: `double`

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 “3-D Solution and Gradient Plots with MATLAB Functions” on page 3-344.

Algorithms

Reducing Parabolic Equations to Elliptic Equations

`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.

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 $\mathbf{x}\epsilon\Omega$

where $\mathbf{x}$ represents a 2-D or 3-D point and there are boundary conditions of the same kind as for the elliptic equation on $\partial\Omega$.

The heat equation reads

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

in the presence of distributed heat loss to the surroundings. ρ is the density, C is the thermal capacity, k is the thermal conductivity, h is the film coefficient, u_∞ is the ambient temperature, and f is the heat source.

For time-independent coefficients, the steady-state solution of the equation is the solution to the standard elliptic equation

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

Assuming a mesh on Ω and $t \geq 0$, expand the solution to the PDE (as a function of $\mathbf{x}$) in the Finite Element Method basis:

$$u(\mathbf{x}, t) = \sum_i U_i(t)\phi_i(\mathbf{x})$$

Plugging the expansion into the PDE, multiplying with a test function ϕ_j, integrating over Ω, and applying Green's formula and the boundary conditions yield

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

In matrix notation, we have to solve the *linear, large* and *sparse* ODE system

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

This method is traditionally called *method of lines* semidiscretization.

Solving the ODE with the initial value

$$U_i(0) = u_0(\mathbf{x}_i)$$

yields the solution to the PDE at each node $\mathbf{x}_i$ and time t. Note that K and F are the stiffness matrix and the right-hand side of the elliptic problem

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

with the original boundary conditions, while M is just the mass matrix of the problem

$$-\nabla \cdot (0\nabla u) + du = 0 \text{ in } \Omega.$$

When the Dirichlet conditions are time dependent, F contains contributions from time derivatives of h and $\mathbf{r}$. These derivatives are evaluated by finite differences of the user-specified data.

The ODE system is ill conditioned. Explicit time integrators are forced by stability requirements to very short time steps while implicit solvers can be expensive since they solve an elliptic problem at every time step. The numerical integration of the ODE system is performed by the MATLAB ODE Suite functions, which are efficient for this class of problems. The time step is controlled to satisfy a tolerance on the error, and factorizations of coefficient matrices are performed only when necessary. When coefficients are time dependent, the necessity of reevaluating and refactorizing the matrices each time step may still make the solution time consuming, although `parabolic` reevaluates only that which varies with time. In certain cases a time-dependent Dirichlet matrix $\mathbf{h}(t)$ may cause the error control to fail, even if the problem is mathematically sound and the solution $u(t)$ is smooth. This can happen because the ODE integrator looks only at the reduced solution v with $u = Bv + ud$. As $\mathbf{h}$ changes, the pivoting scheme employed for numerical stability may change the elimination order from one step to the next. This means that B, v, and ud all change discontinuously, although u itself does not.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`parabolic` is not recommended. Use `solvepde` instead. There are no plans to remove `parabolic`.

Coefficients of parabolic PDEs as functions of the solution and its gradient

You can now solve parabolic equations whose coefficients depend on the solution u or on the gradient of u.

See Also

`solvepde`

pdearcl

Represent arc lengths as parametrized curve

Syntax

`pp = pdearcl(p,xy,s,s0,s1)`

Description

`pp = pdearcl(p,xy,s,s0,s1)` returns parameter values for a parametrized curve corresponding to a given set of arc length values.

The arc length values `s`,`s0`, and `s1` can be an affine transformation of the arc length.

Examples

Polygonal Approximation

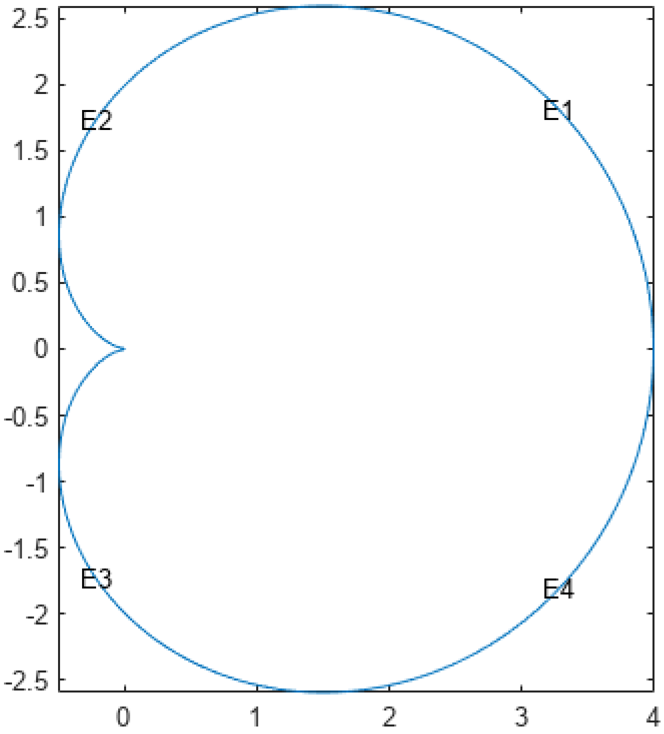
Create a cardioid geometry by using the `pdearcl` function with a polygonal approximation to the geometry. The finite element method uses a triangular mesh to approximate the solution to a PDE numerically. You can avoid loss in accuracy by taking a sufficiently fine polygonal approximation to the geometry. The `pdearcl` function maps between parametrization and arc length in a form well suited to a geometry function. Write this 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
```



Input Arguments

p — Parameter values corresponding to points on curve
row vector

Parameter values corresponding to the points `xy` on the curve, specified as a monotone row vector.

Data Types: `double`

xy — Points on curve
2-row matrix

Points on the curve, specified as a 2-row matrix. Each column specifies the coordinates of a point on the curve.

Data Types: `double`

`s` — Arc length values
vector

Arc length values, specified as a vector.

Data Types: `double`

`s0` — Arc length value for first point
real number

Arc length value for the first point, specified as a real number.

Data Types: `double`

`s1` — Arc length value for last point
real number

Arc length value for the last point, specified as a real number.

Data Types: `double`

Output Arguments

`pp` — Parameter values corresponding to arc length values
vector

Parameter values corresponding to the arc length values `s`, returned as a vector.

Version History

Introduced before R2006a

See Also

`pdegplot`

pdecgrad

(Not recommended) Flux of PDE solution

Note `pdecgrad` is not recommended. Use `evaluateCGradient` instead.

Syntax

```
[cgxu,cgyu] = pdecgrad(p,t,u,c)
[cgxu,cgyu] = pdecgrad(p,t,u,c,time)
[cgxu,cgyu] = pdecgrad( ___ ,FaceID)
```

Description

`[cgxu,cgyu] = pdecgrad(p,t,u,c)` computes the flux of the solution $\mathbf{c} \otimes \nabla \mathbf{u}$ evaluated at the center of each mesh triangle.

The gradient is the same everywhere in the triangle interior because `pdecgrad` uses only linear basis functions. However, the flux can vary inside a triangle because the coefficient `c` can vary.

`[cgxu,cgyu] = pdecgrad(p,t,u,c,time)` uses `time` for parabolic and hyperbolic problems if `c` is time-dependent.

`[cgxu,cgyu] = pdecgrad( ___ ,FaceID)` uses the arguments from the previous syntaxes and restricts the computation to the faces listed in `FaceID`.

Examples

Flux of PDE solution

Create a `[p,e,t]` mesh on the L-shaped membrane.

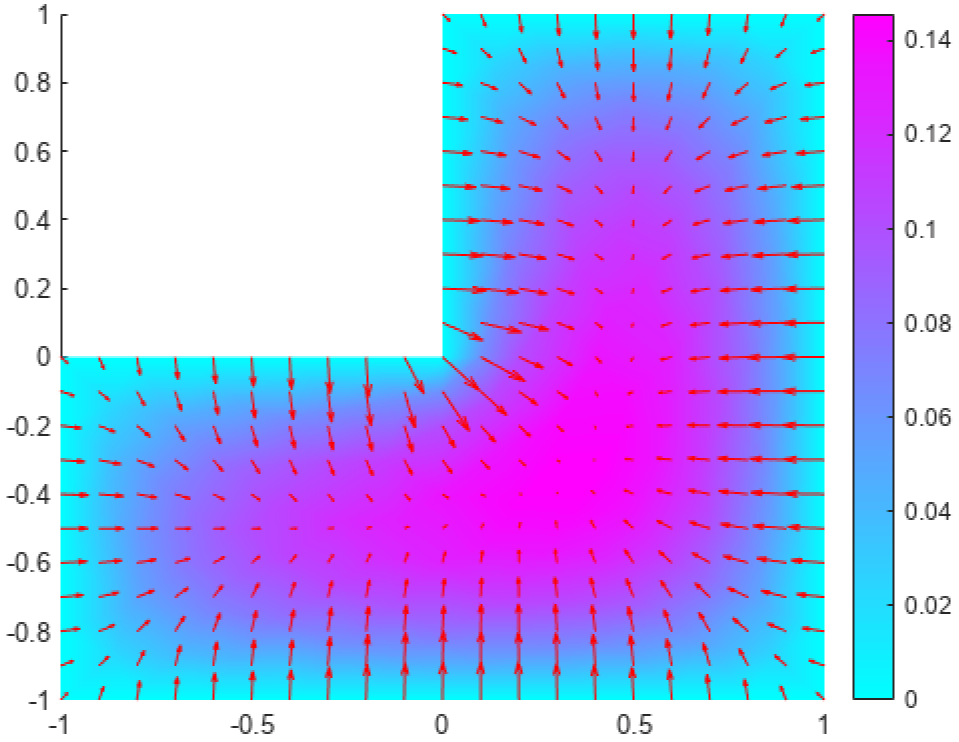
```
[p,e,t] = initmesh('lshapeg');
```

Solve the equation using the Dirichlet boundary condition $u = 0$ on $\partial\Omega$.

```
c = 1;
a = 0;
f = 1;
u = assempde('lshapeb',p,e,t,c,a,f);
```

Compute the flux of the solution and plot the results.

```
[cgradx,cgrady] = pdecgrad(p,t,c,u);
pdeplot(p,e,t,'XYData',u,'FlowData',[cgradx;cgrady])
```



Input Arguments

p — Mesh nodes

matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh elements

matrix

Mesh elements, specified as a 4-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

u — Data at nodes

column vector

Data at nodes, specified as a column vector.

For a PDE system of *N* equations and a mesh with `Np` node points, the first `Np` values of `u` describe the first component, the following `Np` values of `u` describe the second component, and so on.

Data Types: `double`

`c` — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `c` represents the c coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

`time` — Time for parabolic and hyperbolic problems
nonnegative number

Time for parabolic and hyperbolic problems with a time-dependent coefficient `c`, specified as a nonnegative number.

Data Types: `double`

`FaceID` — Face IDs
vector of integers

Face IDs, specified as a vector of integers.

Data Types: `double`

Output Arguments

`cgxu` — *x*-component of flux of `u` evaluated at the center of each triangle
row vector | matrix

x-component of the flux of `u` evaluated at the center of each triangle, returned as a row vector for a scalar PDE or a matrix for a system of PDEs. The number of elements in a row vector or columns in a matrix corresponds to the number `Nt` of mesh triangles. For a PDE system of N equations, each row i from 1 to N contains $\sum_{j=1}^{N} c_{ij11}\frac{\partial u_j}{\partial x}+c_{ij12}\frac{\partial u_j}{\partial y}$.

`cgyu` — *y*-component of flux of `u` evaluated at the center of each triangle
row vector | matrix

y-component of the flux of `u` evaluated at the center of each triangle, returned as a row vector for a scalar PDE or a matrix for a system of PDEs. The number of elements in a row vector or columns in a matrix corresponds to the number `Nt` of mesh triangles. For a PDE system of N equations, each row i from 1 to N contains $\sum_{j=1}^{N} c_{ij21}\frac{\partial u_j}{\partial x}+c_{ij22}\frac{\partial u_j}{\partial y}$.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2018a

`pdecgrad` is not recommended. Use `evaluateCGradient` instead. There are no plans to remove `pdecgrad`.

See Also

`evaluateCGradient` | `pdegrad`

pdecirc

Package: `pde`

Draw circle in PDE Modeler app

Syntax

```
pdecirc(xc,yc,R)
pdecirc(xc,yc,R,label)
```

Description

`pdecirc(xc,yc,R)` draws a circle with the center at (`xc`,`yc`) and the radius `R`. The `pdecirc` command opens the PDE Modeler app with the specified circle already drawn in it. If the app is already open, `pdecirc` adds the specified circle to the app window without deleting any existing shapes.

`pdecirc` updates the state of the geometry description matrix inside the PDE Modeler app to include the circle. You can export the geometry description matrix from the PDE Modeler app to the MATLAB Workspace by selecting **DrawExport Geometry Description, Set Formula, Labels...**. For details on the format of the geometry description matrix, see `decsg`.

`pdecirc(xc,yc,R,label)` assigns a name to the circle. Otherwise, `pdecirc` uses a default name, such as `C1`, `C2`, and so on.

Examples

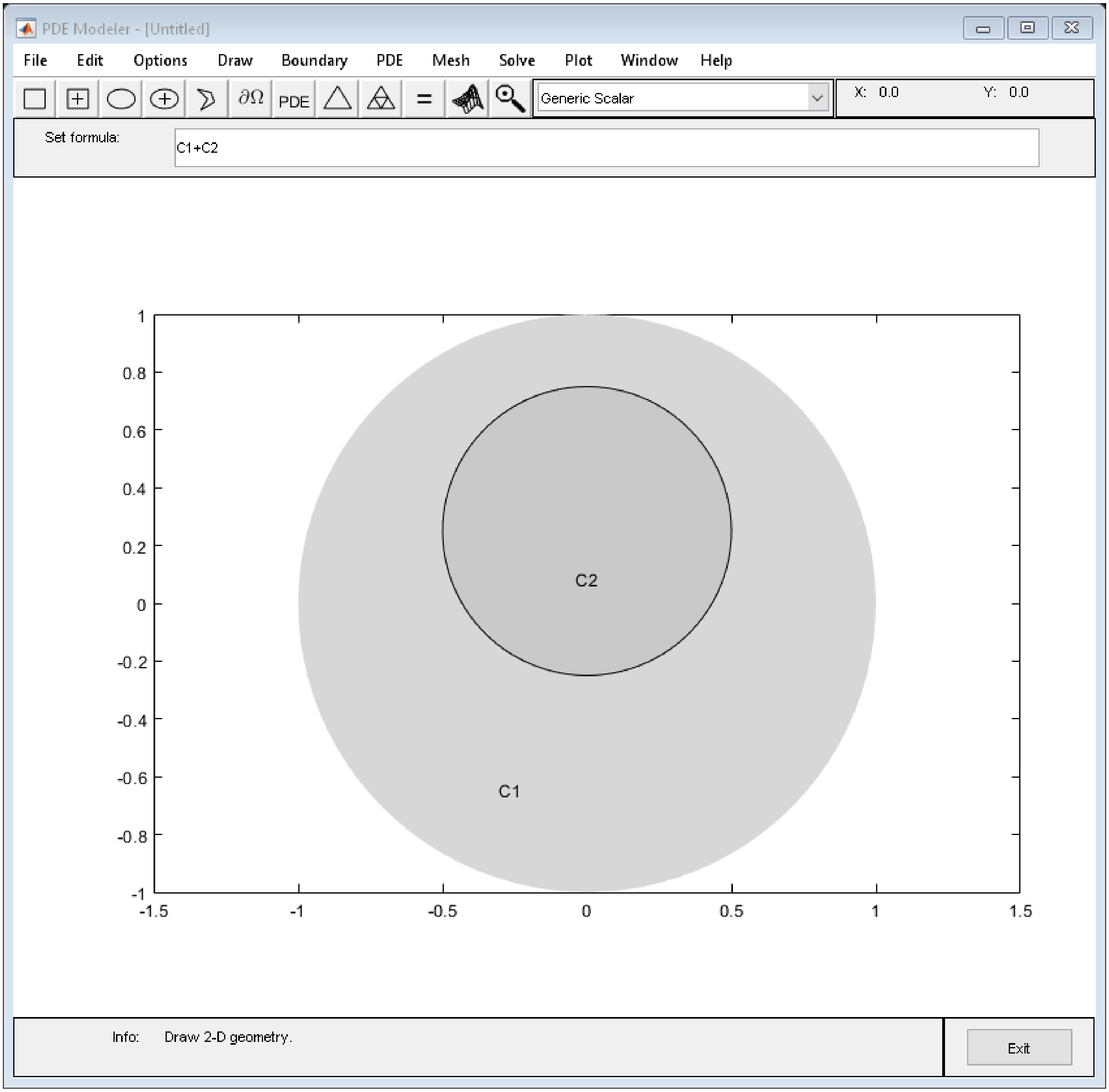
Draw Circle in PDE Modeler App

Open the PDE Modeler app window containing a circle with the center at (0,0) and the radius 1.

```
pdecirc(0,0,1)
```

Call the `pdecirc` command again to draw a circle with the center at (0,0.25) and the radius 0.5. The `pdecirc` command adds the second circle to the app window without deleting the first.

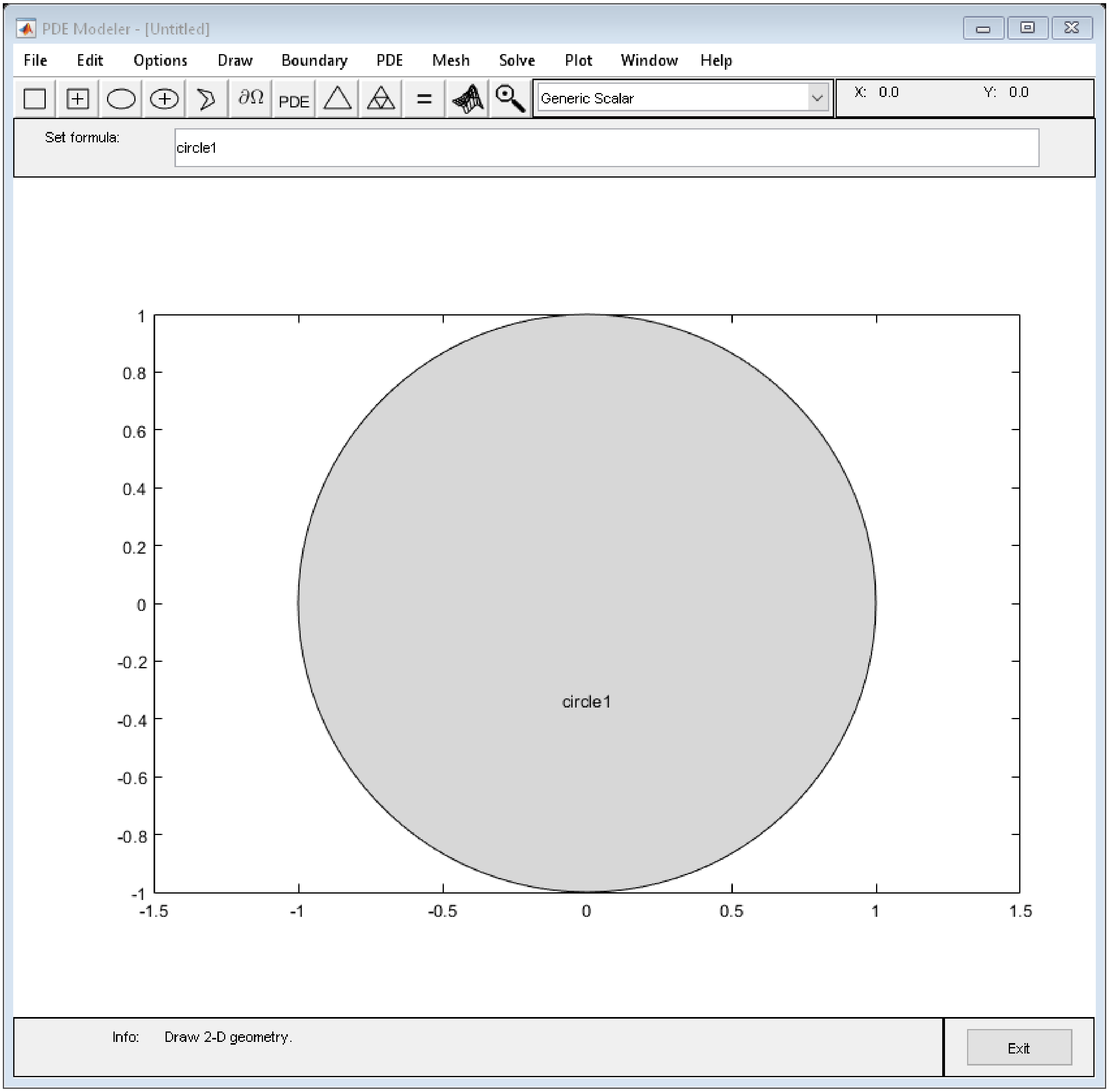
```
pdecirc(0,0.25,0.5)
```



Assign Name to Circle in PDE Modeler App

Open the PDE Modeler app window containing a circle with the center at (0,0) and the radius 1. Assign the name `circle1` to this circle.

```
pdecirc(0,0,1,"circle1")
```



Set formula: circle1

Info: Draw 2-D geometry.

Input Arguments

xc — *x*-coordinate of center

real number

x-coordinate of the center of the circle, specified as a real number.

Data Types: `double`

yc — *y*-coordinate of center
real number

y-coordinate of the center of the circle, specified as a real number.

Data Types: `double`

R — Radius
positive number

Radius of the circle, specified as a positive number.

Data Types: `double`

`label` — Name
character vector | string scalar

Name of the circle, specified as a character vector or string scalar.

Data Types: `char` | `string`

Tips

`pdecirc` opens the PDE Modeler app and draws a circle. If, instead, you want to draw circles in a MATLAB figure window, choose one of these approaches:

- Use the `plot` command, for example:

  ```
  t = linspace(0,2*pi);
  plot(cos(t),sin(t))
  ```
- Use the `rectangle` function with the `Curvature` name-value pair set to `[1 1]`.
- Use the Image Processing Toolbox™ `viscircles` function.

Version History

Introduced before R2006a

See Also

`pdeellip` | `pdepoly` | `pderect` | **PDE Modeler**

pdecont

(To be removed) Contour plot of PDE node or triangle data

Note `pdecont` will be removed in a future release. Use `pdeplot` instead. For more information, see “Compatibility Considerations”.

Syntax

```
pdecont(p,t,u)
pdecont(p,t,u,n)
pdecont(p,t,u,v)
h = pdecont( ___ )
```

Description

`pdecont(p,t,u)` creates a contour plot of node data or triangle data. By default, `pdecont` uses 10 levels for a contour plot. The `p` and `t` arguments specify the geometry of the PDE problem.

If `u` is a column vector, `pdecont` treats it as a node data. If `u` is a row vector, `pdesurf` treats it as a triangle data.

`pdecont(p,t,u,n)` plots `n` levels.

`pdecont(p,t,u,v)` plots levels at the solution heights specified by `v`.

`h = pdecont( ___ )` uses any of the previous syntaxes and returns handles to the drawn axes objects.

Examples

Contour Plot of PDE Solution

Plot contours of the solution to the equation $-\Delta u = 1$ on the L-shaped membrane using the `pdecont` function.

First, create and refine a `[p,e,t]` mesh on the L-shaped membrane.

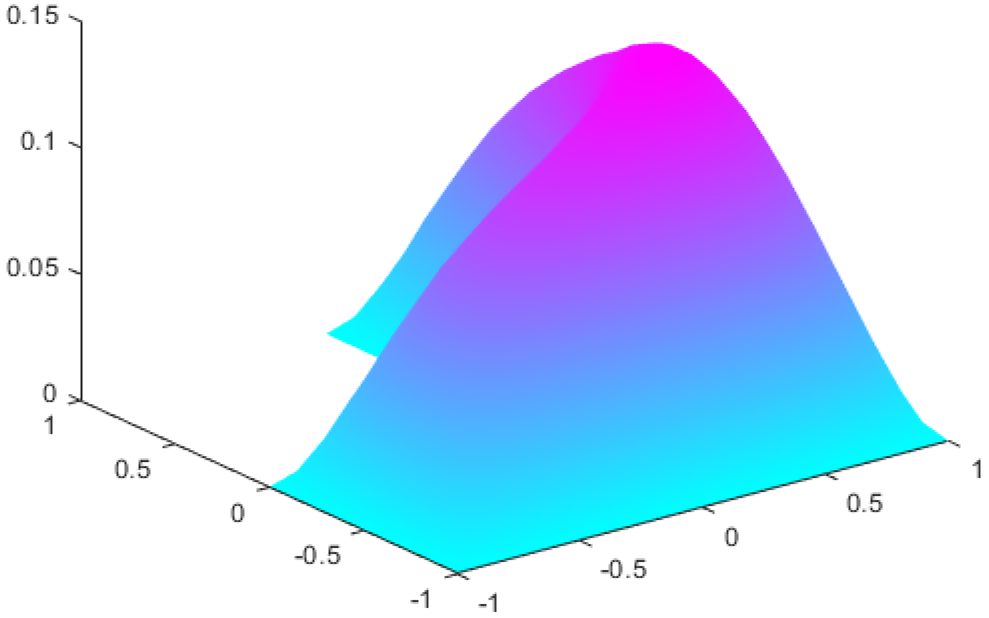
```
[p,e,t] = initmesh('lshapeg');
[p,e,t] = refinemesh('lshapeg',p,e,t);
```

Solve the equation using the Dirichlet boundary conditions $u = 0$ on $\partial\Omega$.

```
u = assempde('lshapeb',p,e,t,1,0,1);
```

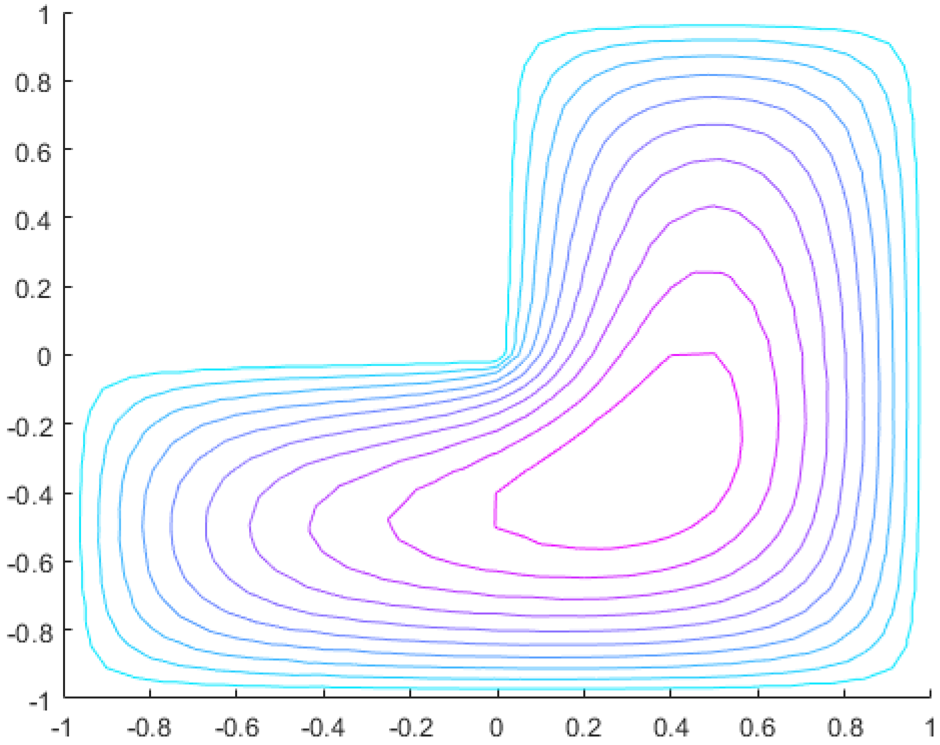
Plot the solution using the `pdesurf` function.

```
pdesurf(p,t,u)
```



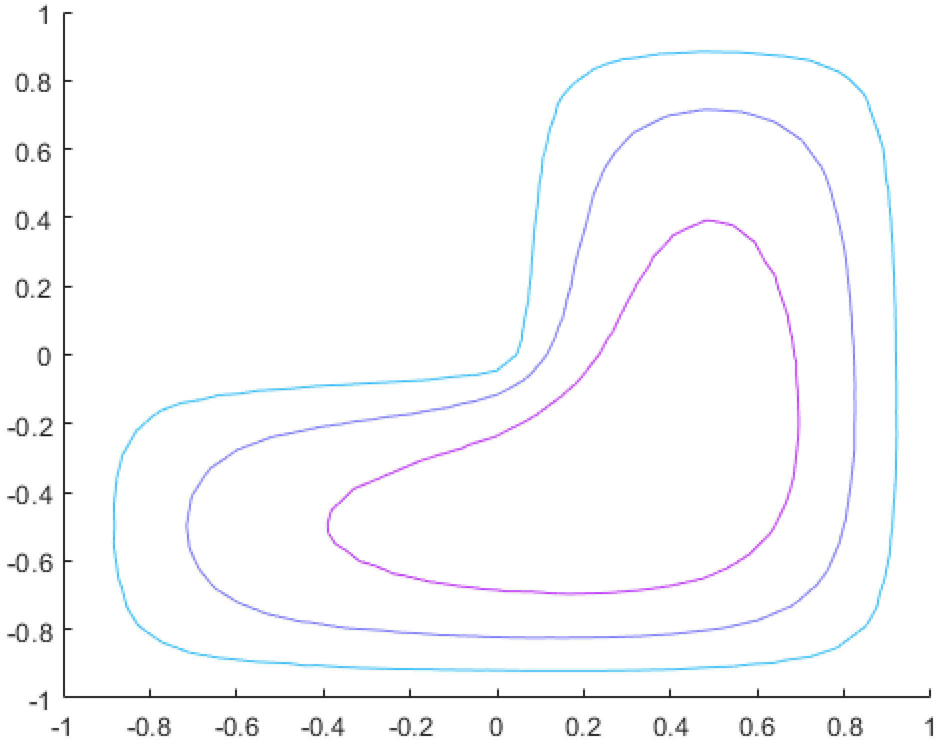
Plot contours of the solution using the `pdecont` function. By default, there are 10 levels.

```
pdecont(p,t,u)
```



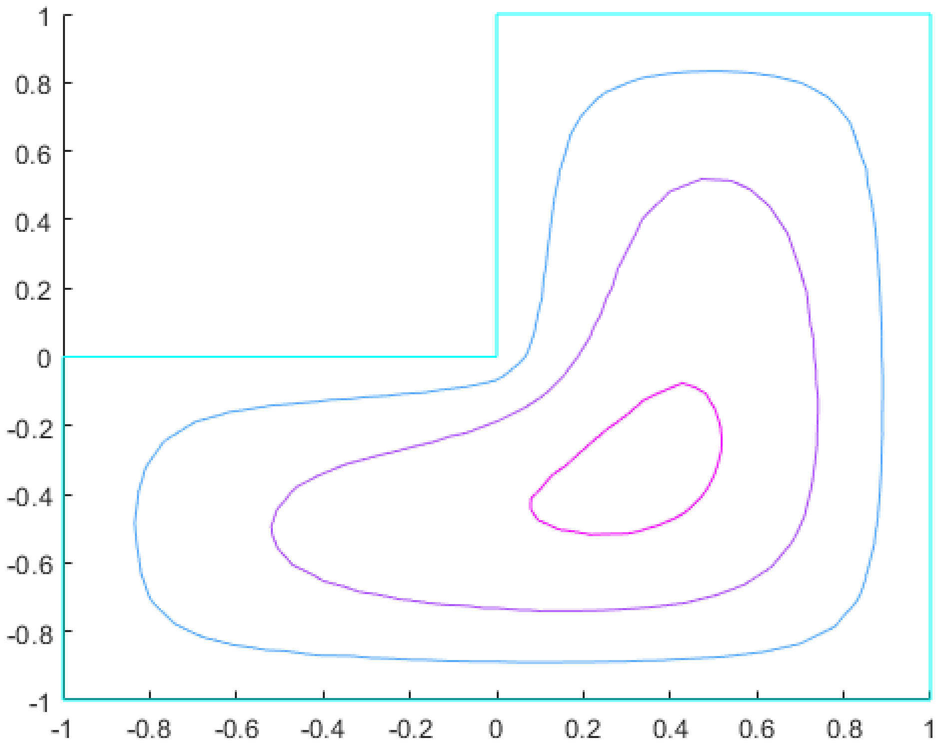
Now plot the contours using three levels.

```
pdecont(p,t,u,3)
```



Plot the contours of the solution at the heights 0, 0.05, 0.1, and 0.14.

```
pdecont(p,t,u,[0 0.05 0.1 0.14])
```



Input Arguments

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 details on the mesh data representation, see `initmesh`.

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 details on the mesh data representation, see `initmesh`.

Data Types: `double`

u — PDE solution

vector

PDE solution, specified as a vector.

The `pdesurf` function treats a column vector as node data and uses continuous style and interpolated shading. The function treats a row vector as triangle data and uses discontinuous style and flat shading.

Data Types: `double`

n — Number of levels
positive integer

Number of levels, specified as a positive integer.

Data Types: `double`

v — Levels to plot
vector of heights

Levels to plot, specified as a vector of heights.

Data Types: `double`

Output Arguments

h — Handles to graphics objects
vector

Handles to graphics objects, returned as a vector.

Tips

- For more control over a contour plot, use the `pdeplot` function.

Version History

Introduced before R2006a

`pdecont` will be removed
Not recommended starting in R2022a

`pdecont` will be removed. Use `pdeplot` instead.

To update your code for plotting the default 10 levels `u`, change instances of the function call `pdecont(p,t,u)` to the function call:

```
pdeplot(p,[],t,'XYData',u,'XStyle','off', ...
                'Contour','on','ColorBar','off')
```

To update your code for plotting the first `n` levels, change instances of the function call `pdecont(p,t,u,n)` to the function call:

```
pdeplot(p,[],t,'XYData',u,'XStyle','off', ...
                'Contour','on','Levels',n, ...
                'ColorBar','off')
```

To update your code for plotting levels specified as a vector `v`, change instances of the function call `pdecont(p,t,u,v)` to the function call:

```
pdeplot(p,[],t,'XYData',u,'XStyle','off', ...
                'Contour','on','Levels',v, ...
                'ColorBar','off')
```

`pdeplot` gives you more control over your contour plot.

Note that the legacy workflow that uses the `[p,e,t]` mesh is not recommended. New features might not be compatible with this legacy workflow. For description of the mesh data in the recommended workflow, see "Mesh Data" on page 2-175.

See Also

`pdemesh` | `pdeplot`

pdeeig

(Not recommended) Solve eigenvalue PDE problem

Note `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

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

or the system PDE eigenvalue problem

$$-\nabla\cdot(\mathbf{c}\otimes\nabla u)+\mathbf{a}u=\lambda du \text{ on } \Omega$$

with geometry, boundary conditions, and mesh specified in `model`, a `PDEModel` object.

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

$$Kc\ u_i = \lambda B' M B u_i$$
$$u = B u_i$$

with Real(λ) in the interval r.

Examples

Eigenvalues and Eigenvectors of 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,'GeometricOrder','linear','Hmax',0.02);
```

```
c = 1;
a = 0;
d = 1;
r = [-Inf 100];
[v,l] = pdeeig(model,c,a,d,r);

                Basis= 10,  Time=   0.36,  New conv eig=  0
                Basis= 11,  Time=   0.41,  New conv eig=  0
                Basis= 12,  Time=   0.41,  New conv eig=  0
                Basis= 13,  Time=   0.45,  New conv eig=  0
                Basis= 14,  Time=   0.47,  New conv eig=  0
                Basis= 15,  Time=   0.48,  New conv eig=  0
                Basis= 16,  Time=   0.50,  New conv eig=  1
                Basis= 17,  Time=   0.50,  New conv eig=  4
                Basis= 18,  Time=   0.52,  New conv eig=  4
                Basis= 19,  Time=   0.58,  New conv eig=  4
                Basis= 20,  Time=   0.59,  New conv eig=  4
                Basis= 21,  Time=   0.59,  New conv eig=  4
                Basis= 22,  Time=   0.59,  New conv eig=  4
                Basis= 23,  Time=   0.61,  New conv eig=  4
                Basis= 24,  Time=   0.67,  New conv eig=  4
                Basis= 25,  Time=   0.80,  New conv eig=  5
                Basis= 26,  Time=   0.86,  New conv eig=  5
                Basis= 27,  Time=   0.86,  New conv eig=  5
                Basis= 28,  Time=   0.86,  New conv eig=  6
                Basis= 29,  Time=   0.91,  New conv eig=  7
                Basis= 30,  Time=   0.91,  New conv eig=  7
                Basis= 31,  Time=   0.98,  New conv eig=  7
                Basis= 32,  Time=   0.98,  New conv eig=  8
                Basis= 33,  Time=   1.00,  New conv eig=  8
                Basis= 34,  Time=   1.00,  New conv eig=  8
                Basis= 35,  Time=   1.02,  New conv eig=  9
                Basis= 36,  Time=   1.02,  New conv eig=  9
                Basis= 37,  Time=   1.03,  New conv eig=  9
                Basis= 38,  Time=   1.05,  New conv eig=  9
                Basis= 39,  Time=   1.14,  New conv eig=  9
                Basis= 40,  Time=   1.20,  New conv eig=  9
                Basis= 41,  Time=   1.44,  New conv eig=  9
                Basis= 42,  Time=   1.61,  New conv eig= 11
                Basis= 43,  Time=   1.64,  New conv eig= 11
                Basis= 44,  Time=   1.69,  New conv eig=  9
                Basis= 45,  Time=   1.70,  New conv eig= 12
                Basis= 46,  Time=   1.77,  New conv eig= 14
                Basis= 47,  Time=   1.78,  New conv eig= 14
                Basis= 48,  Time=   1.86,  New conv eig= 15
                Basis= 49,  Time=   1.95,  New conv eig= 17
                Basis= 50,  Time=   2.00,  New conv eig= 17
                Basis= 51,  Time=   2.09,  New conv eig= 18
                Basis= 52,  Time=   2.16,  New conv eig= 19
                Basis= 53,  Time=   2.23,  New conv eig= 19
                Basis= 54,  Time=   2.31,  New conv eig= 20
                Basis= 55,  Time=   2.36,  New conv eig= 21
                Basis= 56,  Time=   2.44,  New conv eig= 24
                Basis= 57,  Time=   2.48,  New conv eig= 27
                Basis= 58,  Time=   2.59,  New conv eig= 28
End of sweep:   Basis= 58,  Time=   2.59,  New conv eig= 28
                Basis= 38,  Time=   3.00,  New conv eig=  0
                Basis= 39,  Time=   3.05,  New conv eig=  0
```

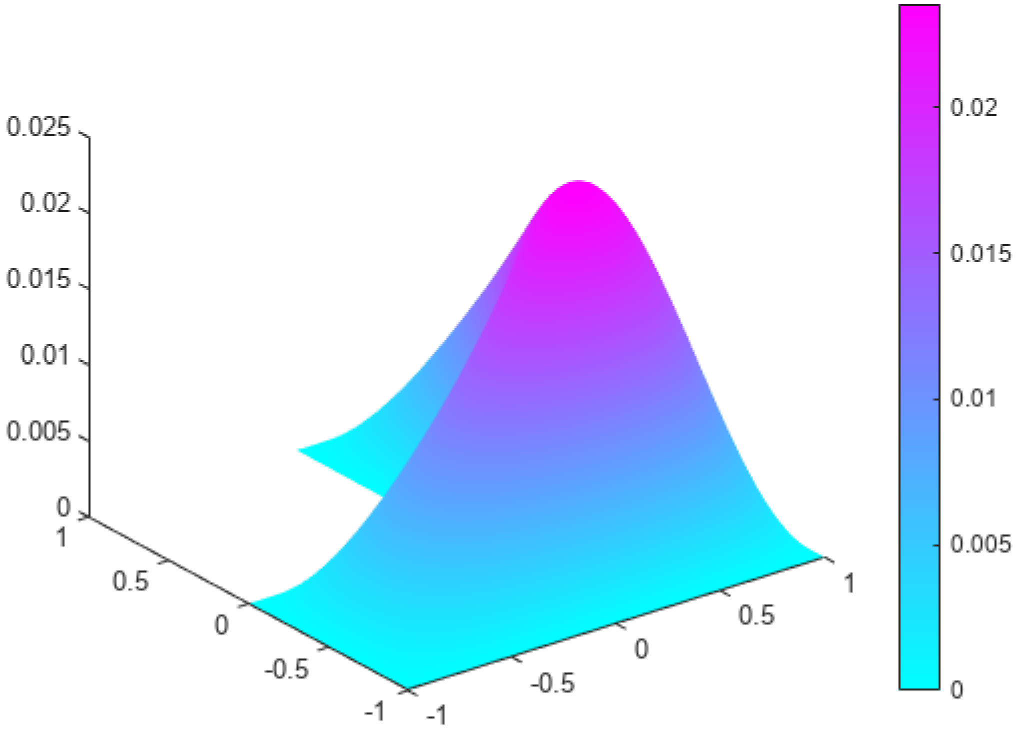
```
              Basis= 40,  Time=   3.06,  New conv eig=  0
End of sweep: Basis= 40,  Time=   3.06,  New conv eig=  0
```

```
l(1)                    % first eigenvalue
```

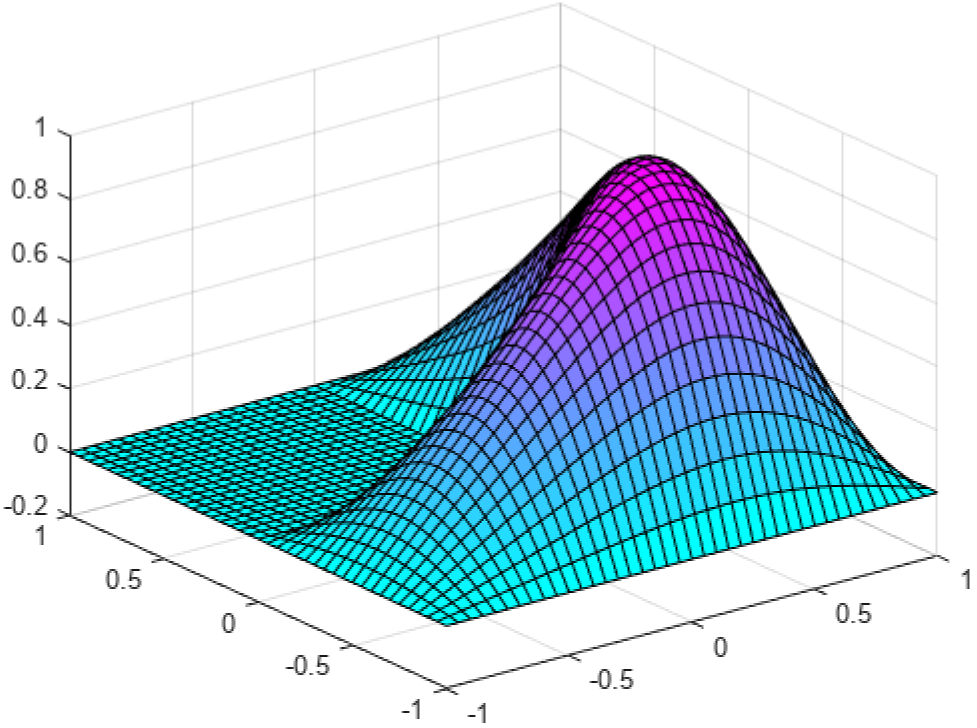
```
ans = 9.6506
```

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
```

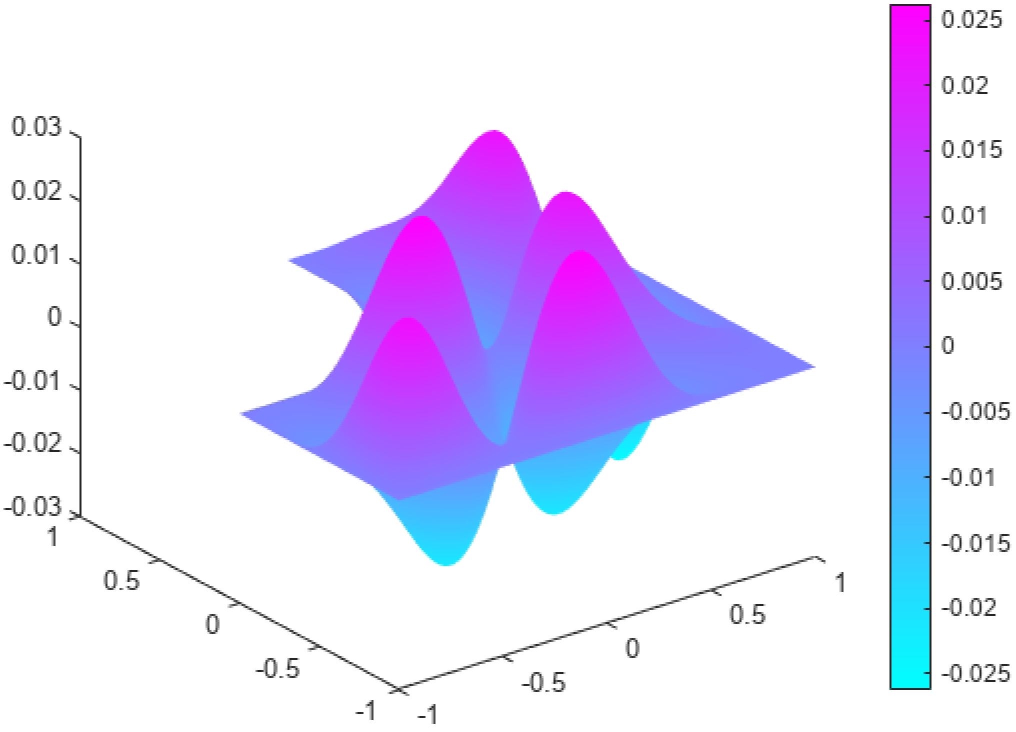


Compute the sixteenth eigenvalue, and plot the sixteenth eigenmode.

```
l(16)                      % sixteenth eigenvalue
```

```
ans = 92.5248
```

```
figure
pdeplot(model,'XYData',v(:,16),'ZData',v(:,16))    % sixteenth eigenmode
```

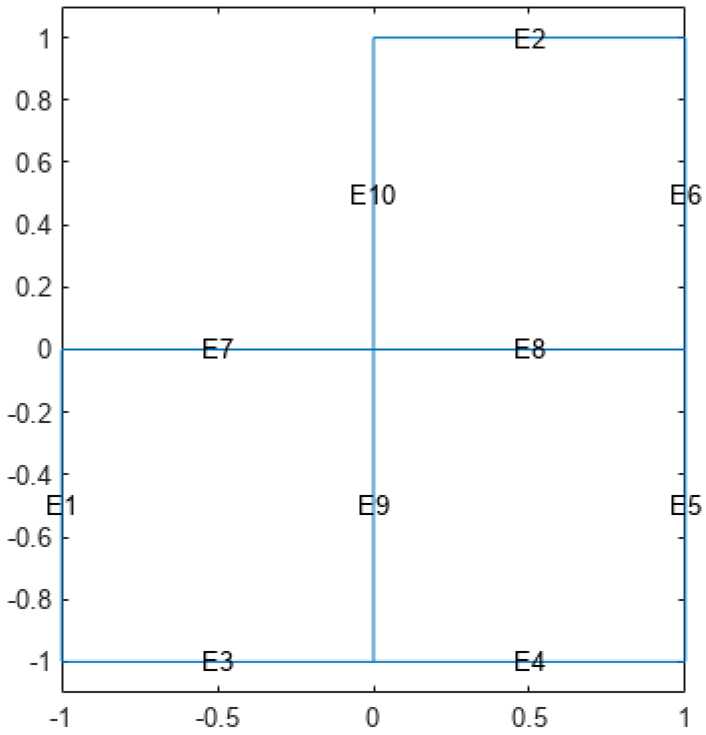


Eigenvalues and Eigenvectors of the L-Shaped Membrane Using Legacy Syntax

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 “Parametrized Function for 2-D Geometry Creation” on page 2-22.

```
g = @lshapeg;
pdegplot(g,'EdgeLabels','on')
axis equal
ylim([-1.1,1.1])
```



Set zero Dirichlet boundary conditions using the `lshapeb` function.

```
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=   1.58,  New conv eig=  0
              Basis= 11,  Time=   1.62,  New conv eig=  0
              Basis= 12,  Time=   1.69,  New conv eig=  0
              Basis= 13,  Time=   1.86,  New conv eig=  0
              Basis= 14,  Time=   1.86,  New conv eig=  0
              Basis= 15,  Time=   2.05,  New conv eig=  1
              Basis= 16,  Time=   2.08,  New conv eig=  1
              Basis= 17,  Time=   2.20,  New conv eig=  3
              Basis= 18,  Time=   2.38,  New conv eig=  4
              Basis= 19,  Time=   2.42,  New conv eig=  4
              Basis= 20,  Time=   2.64,  New conv eig=  4
              Basis= 21,  Time=   2.78,  New conv eig=  4
```

```
                 Basis= 22,  Time=   2.94,  New conv eig=  4
                 Basis= 23,  Time=   3.17,  New conv eig=  4
                 Basis= 24,  Time=   3.25,  New conv eig=  5
                 Basis= 25,  Time=   3.25,  New conv eig=  5
                 Basis= 26,  Time=   3.27,  New conv eig=  5
                 Basis= 27,  Time=   3.38,  New conv eig=  6
                 Basis= 28,  Time=   3.50,  New conv eig=  7
                 Basis= 29,  Time=   3.58,  New conv eig=  7
                 Basis= 30,  Time=   3.62,  New conv eig=  7
                 Basis= 31,  Time=   3.78,  New conv eig=  7
                 Basis= 32,  Time=   3.88,  New conv eig=  8
                 Basis= 33,  Time=   3.94,  New conv eig=  8
                 Basis= 34,  Time=   4.16,  New conv eig=  8
                 Basis= 35,  Time=   4.19,  New conv eig=  9
                 Basis= 36,  Time=   4.20,  New conv eig=  9
                 Basis= 37,  Time=   4.30,  New conv eig=  9
                 Basis= 38,  Time=   4.42,  New conv eig=  9
                 Basis= 39,  Time=   4.50,  New conv eig=  9
                 Basis= 40,  Time=   4.53,  New conv eig=  9
                 Basis= 41,  Time=   4.58,  New conv eig=  9
                 Basis= 42,  Time=   4.72,  New conv eig= 10
                 Basis= 43,  Time=   4.91,  New conv eig= 11
                 Basis= 44,  Time=   4.95,  New conv eig= 12
                 Basis= 45,  Time=   5.05,  New conv eig= 12
                 Basis= 46,  Time=   5.06,  New conv eig= 14
                 Basis= 47,  Time=   5.14,  New conv eig= 15
                 Basis= 48,  Time=   5.23,  New conv eig= 16
                 Basis= 49,  Time=   5.33,  New conv eig= 17
                 Basis= 50,  Time=   5.61,  New conv eig= 17
                 Basis= 51,  Time=   5.73,  New conv eig= 18
                 Basis= 52,  Time=   5.94,  New conv eig= 18
                 Basis= 53,  Time=   6.17,  New conv eig= 19
                 Basis= 54,  Time=   6.22,  New conv eig= 19
                 Basis= 55,  Time=   6.27,  New conv eig= 22
                 Basis= 56,  Time=   6.39,  New conv eig= 24
                 Basis= 57,  Time=   6.52,  New conv eig= 28
End of sweep: Basis= 57,  Time=   6.52,  New conv eig= 28
                 Basis= 38,  Time=   7.12,  New conv eig=  0
                 Basis= 39,  Time=   7.27,  New conv eig=  0
                 Basis= 40,  Time=   7.34,  New conv eig=  0
                 Basis= 41,  Time=   7.42,  New conv eig=  0
                 Basis= 42,  Time=   7.52,  New conv eig=  0
End of sweep: Basis= 42,  Time=   7.52,  New conv eig=  0
```

Find the first eigenvalue.

```
l(1)
```

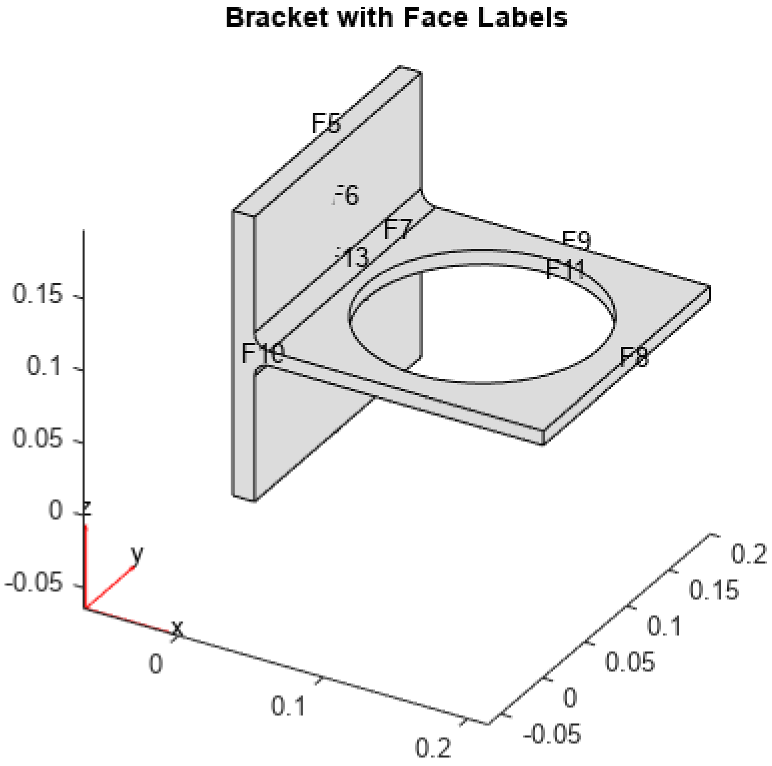
```
ans = 9.6481
```

Eigenvalues and Eigenvectors Using Finite Element Matrices

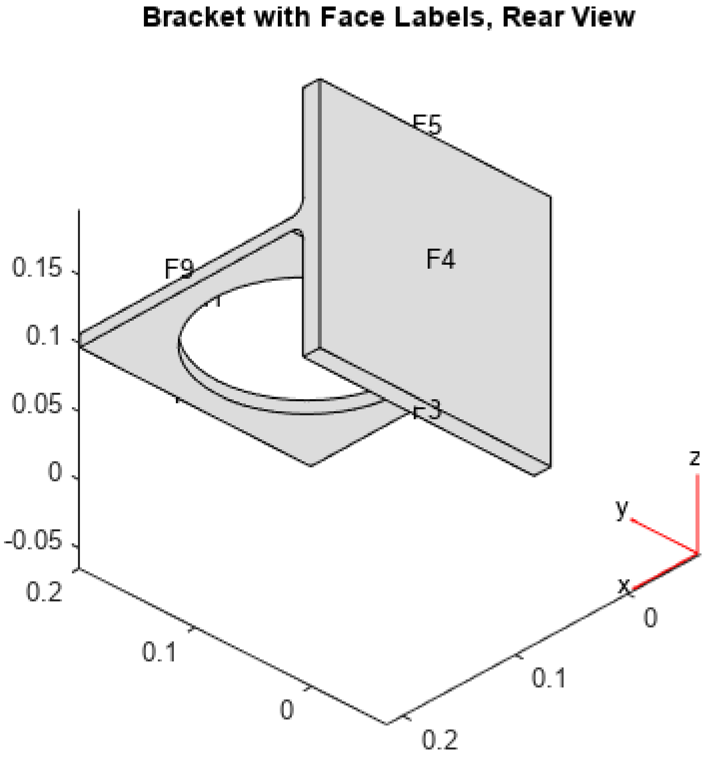
Import a simple 3-D geometry and find eigenvalues and eigenvectors from the associated finite element matrices.

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')
```



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=   1.16,  New conv eig=  0
              Basis= 11,  Time=   1.22,  New conv eig=  0
              Basis= 12,  Time=   1.28,  New conv eig=  0
              Basis= 13,  Time=   1.36,  New conv eig=  1
              Basis= 14,  Time=   1.36,  New conv eig=  1
              Basis= 15,  Time=   1.39,  New conv eig=  1
              Basis= 16,  Time=   1.42,  New conv eig=  2
              Basis= 17,  Time=   1.45,  New conv eig=  3
```

```
End of sweep: Basis= 17,  Time=   1.45,  New conv eig=  3
              Basis= 13,  Time=   1.52,  New conv eig=  0
End of sweep: Basis= 13,  Time=   1.62,  New conv eig=  0
```

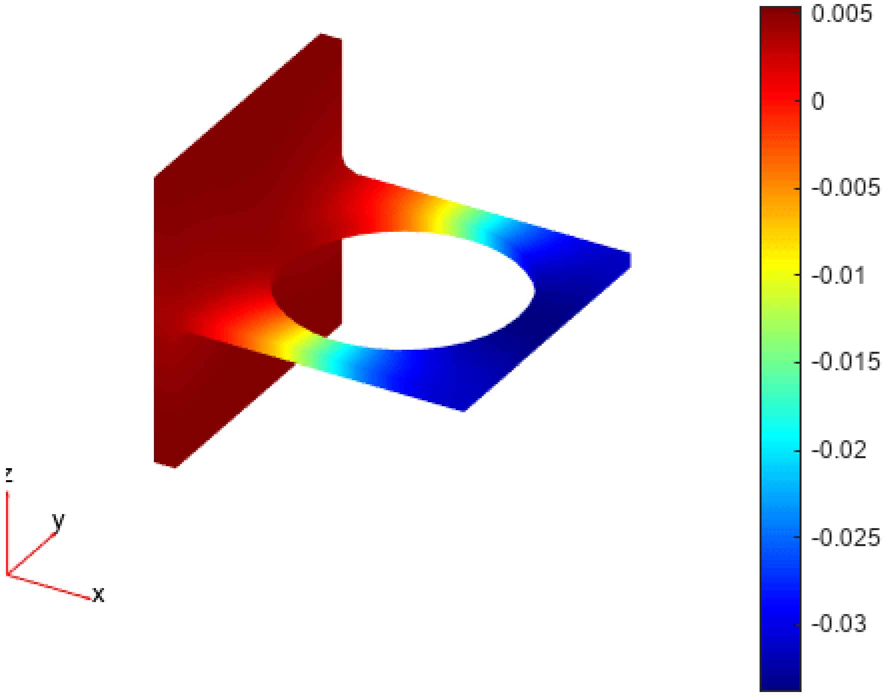
Look at the first two eigenvalues.

```
l([1,2])
```

```
ans = 2×1

   -0.0000
   42.8670
```

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`

c — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

$$-\nabla \cdot (\mathbf{c} \otimes \nabla u) + \mathbf{a}u = \lambda du \text{ on } \Omega$$

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

a — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

$$-\nabla \cdot (\mathbf{c} \otimes \nabla u) + \mathbf{a}u = \lambda du \text{ on } \Omega$$

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

d — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or 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

$$-\nabla \cdot (\mathbf{c} \otimes \nabla u) + \mathbf{a}u = \lambda du \text{ on } \Omega$$

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

r — Eigenvalue range
two-element real vector

Eigenvalue range, specified as a two-element real vector. Real parts of eigenvalues λ fall in the range `r(1)` ≤ λ ≤ `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 Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 5-101. Typically, `Kc` is the output of `assempde`.

`B` — Dirichlet nullspace
sparse matrix

Dirichlet nullspace, returned as a sparse matrix. See "Algorithms" on page 5-101. 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 5-101.

To obtain the input matrices for `pdeeig`, `hyperbolic` or `parabolic`, run both `assema` and `assempde`:

```
[Kc,Fc,B,ud] = assempde(model,c,a,f);
[~,M,~] = assema(model,0,d,f);
```

Note Create the `M` matrix using `assema` with `d`, not `a`, as the argument before `f`.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`v` — Eigenvectors
matrix

Eigenvectors, returned as a matrix. Suppose

- `Np` is the number of mesh nodes
- *N* is the number of equations
- `ev` is the number of eigenvalues returned in `l`

Then `v` has size `Np`**N*-by-`ev`. Each column of `v` corresponds to the eigenvectors of one eigenvalue. In each column, the first `Np` elements correspond to the eigenvector of equation 1 evaluated at the mesh nodes, the next `Np` elements correspond to equation 2, etc.

Note Eigenvectors are determined only up to multiple by a scalar, including a negative scalar.

`l` — 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.

Algorithms

Eigenvalue Equations

Partial Differential Equation Toolbox software handles the following basic eigenvalue problem:

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

where λ is an unknown complex number. In solid mechanics, this is a problem associated with wave phenomena describing, e.g., the natural modes of a vibrating membrane. In quantum mechanics λ is the energy level of a bound state in the potential well $a(\mathbf{x})$, where $\mathbf{x}$ represents a 2-D or 3-D point.

The numerical solution is found by discretizing the equation and solving the resulting algebraic eigenvalue problem. Let us first consider the discretization. Expand u in the FEM basis, multiply with a basis element, and integrate on the domain Ω. This yields the generalized eigenvalue equation

$$KU = \lambda MU$$

where the mass matrix corresponds to the right side, i.e.,

$$M_{i,j} = \int_{\Omega} d(\mathbf{x})\phi_j(\mathbf{x})\phi_i(\mathbf{x})\, d\mathbf{x}$$

The matrices K and M are produced by calling `assema` for the equations

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

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 L-Shaped Membrane” on page 3-305 or “Eigenvalues and Eigenmodes of Square” on page 3-317, where actual runs are reported.

First a shift μ is determined close to where we want to find the eigenvalues. When both K and M are positive definite, it is natural to take $\mu = 0$, and get the smallest eigenvalues; in other cases take any point in the interval [lb,ub] where eigenvalues are sought. Subtract μM from the eigenvalue equation and get $(K - \mu M)U = (\lambda - \mu)MU$. Then multiply with the inverse of this shifted matrix and get

$$\frac{1}{\lambda-\mu}U = (K-\mu M)^{-1}MU$$

This is a standard eigenvalue problem $AU = \theta U$, with the matrix $A = (K - \mu M)^{-1}M$ and eigenvalues

$$\theta_i = \frac{1}{\lambda_i - \mu}$$

where $i = 1, \dots, n$. The largest eigenvalues θ_i of the transformed matrix A now correspond to the eigenvalues $\lambda_i = \mu + 1/\theta_i$ of the original *pencil* (K,M) closest to the shift μ.

The Arnoldi algorithm computes an orthonormal basis V where the shifted and inverted operator A is represented by a Hessenberg matrix H,

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

(The subscripts mean that V_j and E_j have j columns and $H_{j,j}$ has j rows and columns. When no subscripts are used we deal with vectors and matrices of size n.)

Some of the eigenvalues of this Hessenberg matrix $H_{j,j}$ eventually give good approximations to the eigenvalues of the original pencil (K,M) when the basis grows in dimension j, and less and less of the eigenvector is hidden in the residual matrix E_j.

The basis V is built one column v_j at a time. The first vector v_1 is chosen at random, as n normally distributed random numbers. In step j, the first j vectors are already computed and form the $n \times j$ matrix V_j. The next vector v_{j+1} is computed by first letting A operate on the newest vector v_j, and then making the result orthogonal to all the previous vectors.

This is formulated as $h_{j+1}v_{j+1} = Av_j - V_jh_j$, where the column vector h_j consists of the Gram-Schmidt coefficients, and $h_{j+1,j}$ is the normalization factor that gives v_{j+1} unit length. Put the corresponding relations from previous steps in front of this and get

$$AV_j = V_jH_{j,j} + v_{j+1}h_{j+1,j}e_j^T$$

where $H_{j,j}$ is a $j \times j$ Hessenberg matrix with the vectors h_j as columns. The second term on the right-hand side has nonzeros only in the last column; the earlier normalization factors show up in the subdiagonal of $H_{j,j}$.

The eigensolution of the small Hessenberg matrix H gives approximations to some of the eigenvalues and eigenvectors of the large matrix operator $A_{j,j}$ in the following way. Compute eigenvalues θ_i and eigenvectors s_i of $H_{j,j}$,

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

Then $y_i = V_j s_i$ is an approximate eigenvector of A, and its residual is

$$r_i = Ay_i - y_i\theta_i = AV_js_i - V_js_i\theta_i = (AV_j - V_jH_{j,j})s_i = v_{j+1}h_{j+1,j}s_{i,j}$$

This residual has to be small in norm for θ_i to be a good eigenvalue approximation. The norm of the residual is

$$\|r_i\| = |h_{j+1,j}s_{j,i}|$$

the product of the last subdiagonal element of the Hessenberg matrix and the last element of its eigenvector. It seldom happens that $h_{j+1,j}$ gets particularly small, but after sufficiently many steps j there are always some eigenvectors s_i with small last elements. The long vector V_{j+1} is of unit norm.

It is not necessary to actually compute the eigenvector approximation y_i to get the norm of the residual; we only need to examine the short vectors s_i, and flag those with tiny last components as converged. In a typical case n may be 2000, while j seldom exceeds 50, so all computations that involve only matrices and vectors of size j are much cheaper than those involving vectors of length n.

This eigenvalue computation and test for convergence is done every few steps j, until all approximations to eigenvalues inside the interval [lb,ub] are flagged as converged. When n is much larger than j, this is done very often, for smaller n more seldom. When all eigenvalues inside the interval have converged, or when j has reached a prescribed maximum, the converged eigenvectors, or more appropriately *Schur vectors*, are computed and put in the front of the basis V.

After this, the Arnoldi algorithm is restarted with a random vector, if all approximations inside the interval are flagged as converged, or else with the best unconverged approximate eigenvector y_i. In each step j of this second Arnoldi run, the vector is made orthogonal to all vectors in V including the converged Schur vectors from the previous runs. This way, the algorithm is applied to a projected matrix, and picks up a second copy of any double eigenvalue there may be in the interval. If anything in the interval converges during this second run, a third is attempted and so on, until no more approximate eigenvalues θ_i show up inside. Then the algorithm signals convergence. If there are still unconverged approximate eigenvalues after a prescribed maximum number of steps, the algorithm signals nonconvergence and reports all solutions it has found.

This is a heuristic strategy that has worked well on both symmetric, nonsymmetric, and even defective eigenvalue problems. There is a tiny theoretical chance of missing an eigenvalue, if all the random starting vectors happen to be orthogonal to its eigenvector. Normally, the algorithm restarts p times, if the maximum multiplicity of an eigenvalue is p. At each restart a new random starting direction is introduced.

The shifted and inverted matrix $A = (K - \mu M)^{-1}M$ is needed only to operate on a vector v_j in the Arnoldi algorithm. This is done by computing an LU factorization,

$$P(K - \mu M)Q = LU$$

using the sparse MATLAB command `lu` (P and Q are permutations that make the triangular factors L and U sparse and the factorization numerically stable). This factorization needs to be done only once, in the beginning, then $x = Av_j$ is computed as,

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

with one sparse matrix vector multiplication, a permutation, sparse forward- and back-substitutions, and a final renumbering.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`pdeeig` is not recommended. Use `solvepdeeig` instead. There are no plans to remove `pdeeig`.

See Also

`solvepdeeig`

pdeellip

Package: `pde`

Draw ellipse in PDE Modeler app

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 the center at (`xc`,`yc`), the semiaxes `a` and `b`, and the rotation `phi` (in radians). The `pdeellip` command opens the PDE Modeler app with the specified ellipse drawn in it. If the app is already open, `pdeellip` adds the specified ellipse to the app window without deleting any existing shapes.

`pdeellip` updates the state of the geometry description matrix inside the PDE Modeler app to include the ellipse. You can export the geometry description matrix from the PDE Modeler app to the MATLAB Workspace by selecting **DrawExport Geometry Description, Set Formula, Labels...**. For details on the format of the geometry description matrix, see `decsg`.

`pdeellip(xc,yc,a,b,phi,label)` assigns a name to the ellipse. Otherwise, `pdeellip` uses a default name, such as `E1`, `E2`, and so on.

Examples

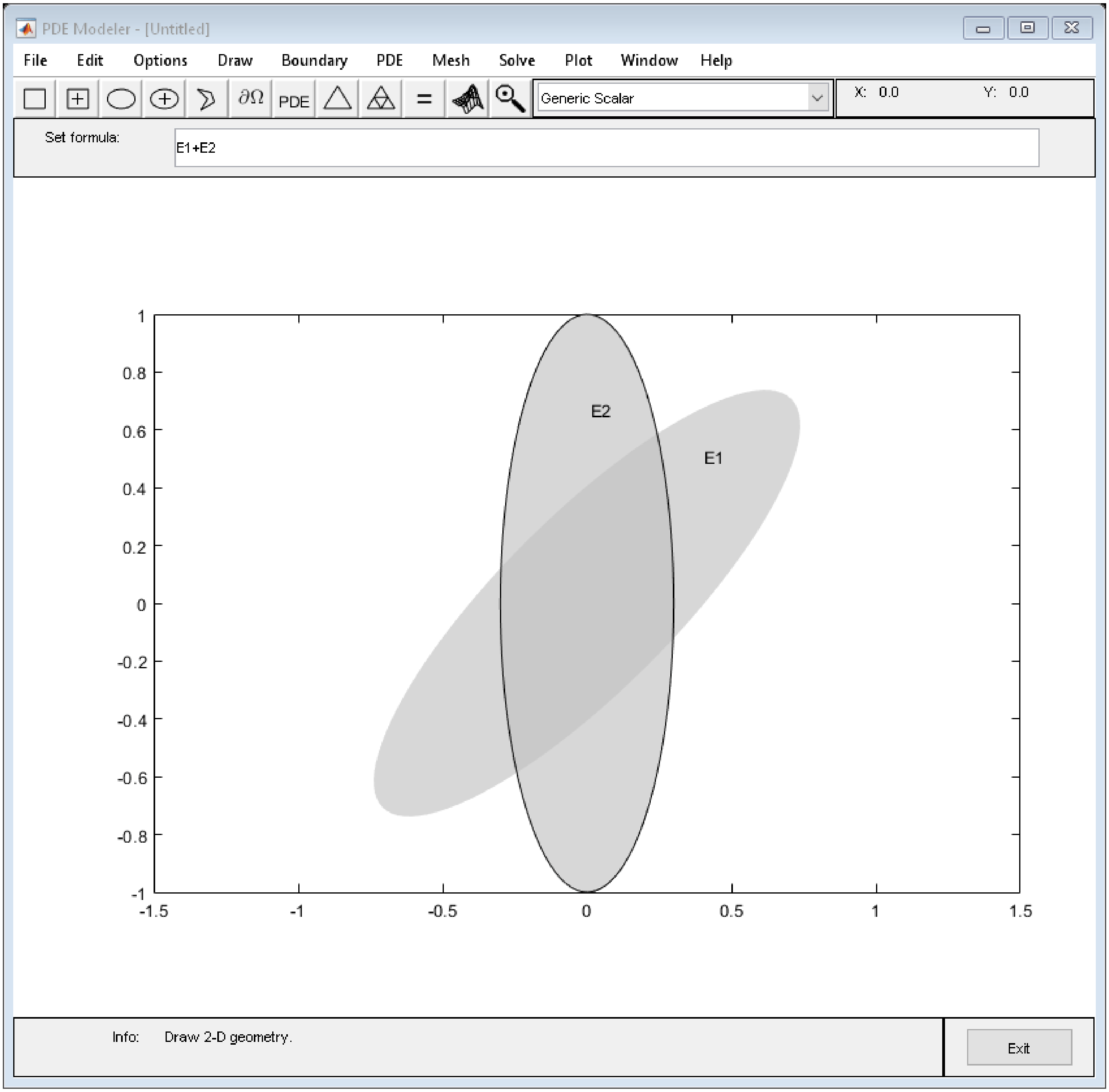
Draw Ellipse in PDE Modeler App

Open the PDE Modeler app window containing an ellipse with the center at (0,0) and the semiaxes 1 and 0.3. Rotate the ellipse by $\pi/4$ counterclockwise.

```
pdeellip(0,0,1,0.3,pi/4)
```

Call the `pdeellip` command again to draw an ellipse with the same center and semiaxes, but rotate it by $\pi/2$ counterclockwise. The `pdeellip` command adds the second ellipse to the app window without deleting the first.

```
pdeellip(0,0,1,0.3,pi/2)
```



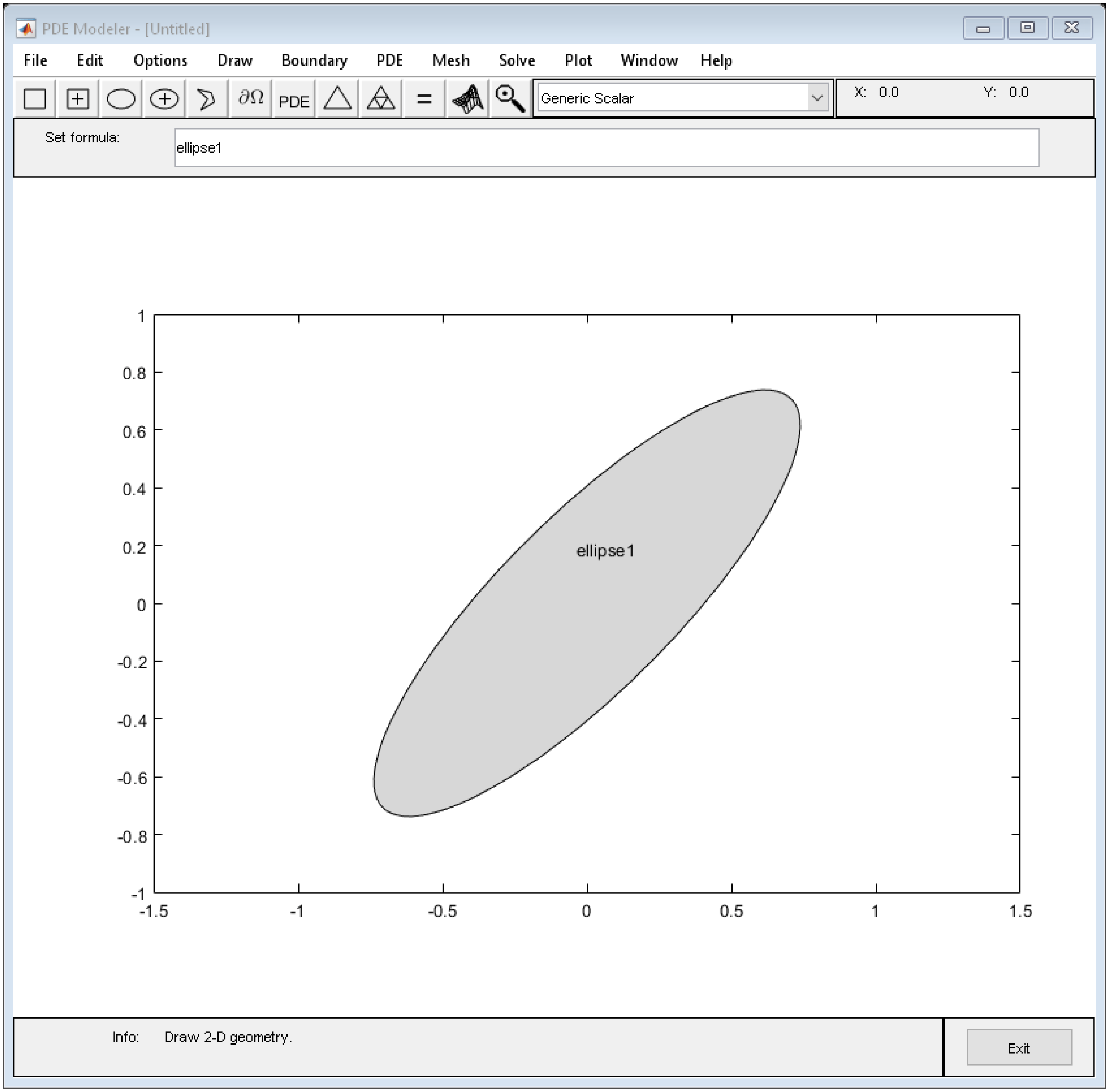
Set formula: E1+E2

Info: Draw 2-D geometry.

Assign Name to Ellipse in PDE Modeler App

Open the PDE Modeler app window containing an ellipse with the center at (0,0) and the semiaxes 1 and 0.3. Rotate the ellipse by $\pi/4$ counterclockwise. Assign the name `ellipse1` to this ellipse.

```
pdeellip(0,0,1,0.3,pi/4,"ellipse1")
```



Set formula: ellipse1

Info: Draw 2-D geometry.

Input Arguments

xc — *x*-coordinate of center

real number

x-coordinate of the center of the ellipse, specified as a real number.

Data Types: `double`

`yc` — *y*-coordinate of center
real number

y-coordinate of the center of the ellipse, specified as a real number.

Data Types: `double`

`a` — Semiaxis
positive number

Semiaxis of the ellipse, specified as a positive number.

Data Types: `double`

`b` — Semiaxis
positive number

Semiaxis of the ellipse, specified as a positive number.

Data Types: `double`

`phi` — Rotation
real number

Rotation of the ellipse, specified as a real number. The rotation value is measured in radians.

Data Types: `double`

`label` — Name
character vector | string scalar

Name of the ellipse, specified as a character vector or string scalar.

Data Types: `char` | `string`

Version History

Introduced before R2006a

See Also

`pdecirc` | `pdepoly` | `pderect` | **PDE Modeler**

pdegplot

Plot PDE geometry

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.

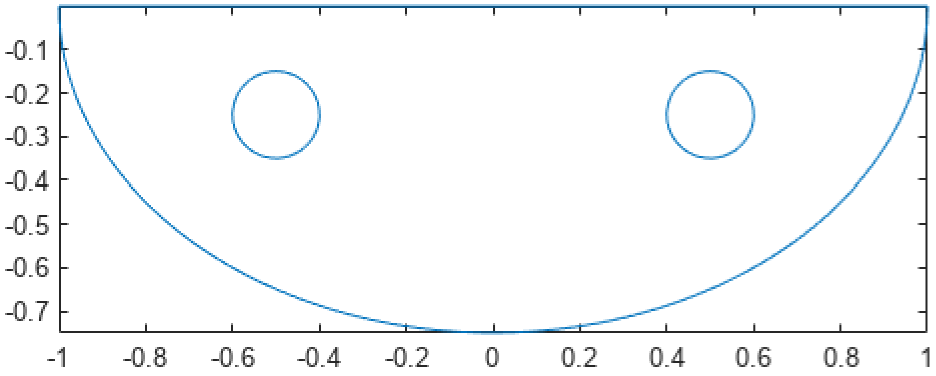
`h = pdegplot( ___ )` returns handles to the graphics, using any of the previous syntaxes.

Examples

Plot 2-D Geometry with and Without Labels

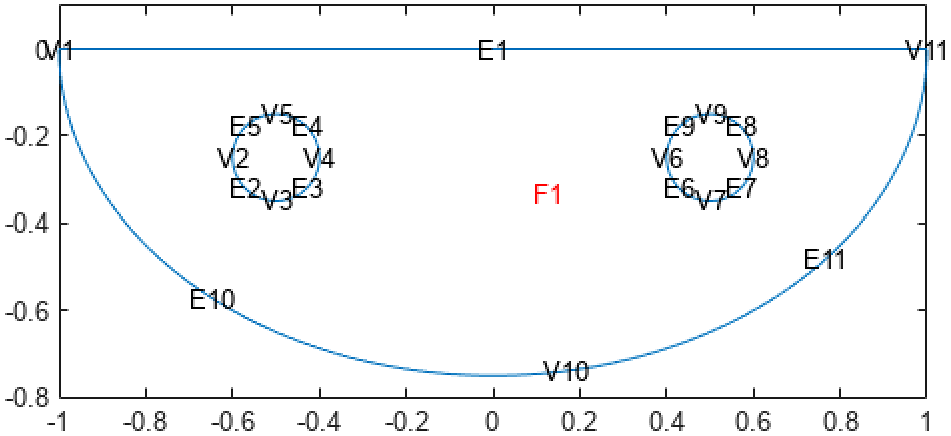
Plot the geometry of a region defined by a few simple shapes.

```
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.

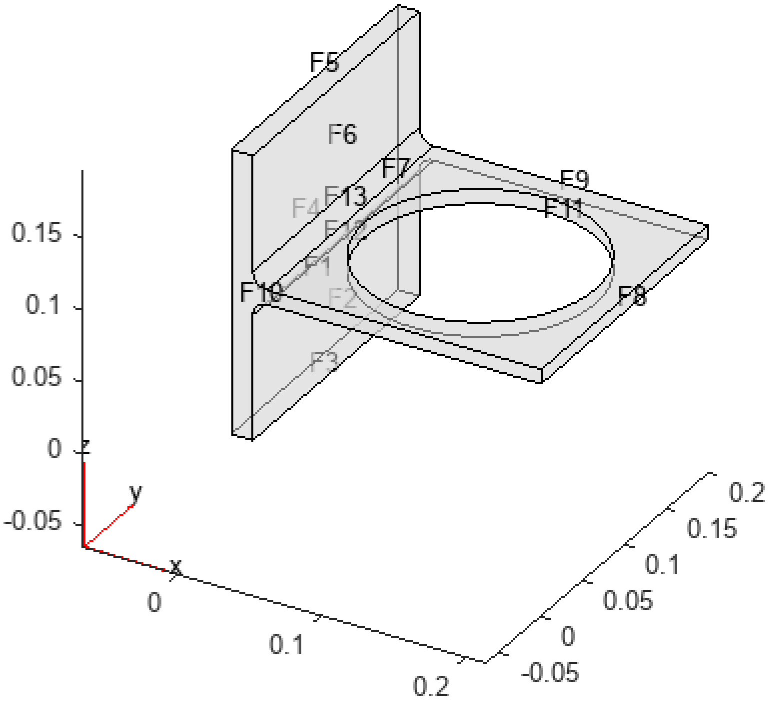
```
pdegplot(g,"VertexLabels","on","EdgeLabels","on","FaceLabels","on")
ylim([-.8,.1])
```



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.

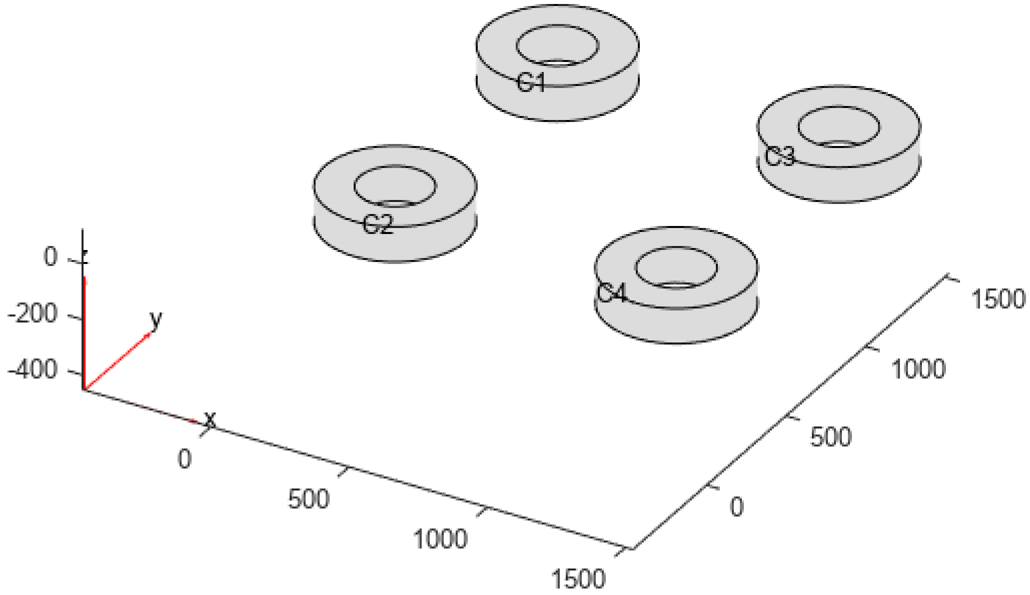
```
model = createpde;
importGeometry(model,"BracketWithHole.stl");
pdegplot(model,"FaceLabels","on","FaceAlpha",0.5)
```



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")
```



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 2-20)
- Name of geometry file (see “Parametrized Function for 2-D Geometry Creation” on page 2-22)
- Function handle to geometry file (see “Parametrized Function for 2-D Geometry Creation” on page 2-22)

Data Types: `double` | `char` | `string` | `function_handle`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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` | `string`

`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` | `string`

`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` | `string`

`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` | `string`

`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

Handles to graphics objects, returned as a vector.

Alternative Functionality

App

If you create 2-D geometry in the PDE Modeler 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**.

Version History

Introduced before R2006a

Improved performance for plots with many text labels

Performance change in R2020a

`pdegplot` shows faster rendering and better responsiveness for plots that display many text labels. Code containing `findobj(fig,'Type','Text')` no longer returns labels on figures produced by `pdegplot`.

Transparency, vertex and cell labels

You can now set plot transparency by using `FaceAlpha`, and display vertex and cell labels by using `VertexLabels` and `CellLabels`, respectively.

The argument `SubdomainLabels` is no longer recommended. Use `FaceLabels` for 2-D geometries instead.

Edge and subdomain labels

Display edge and subdomain labels by setting `edgeLabels` or `subdomainLabels` to `'on'`.

See Also

`pdeplot` | `pdeplot3D` | `pdemesh` | `decsg` | `importGeometry` | **PDE Modeler**

Topics

“STL File Import” on page 2-43
“Solve Problems Using PDEModel Objects” on page 2-2

pdegrad

(Not recommended) Gradient of PDE solution

Note `pdegrad` is not recommended. Use `evaluateGradient` instead.

Syntax

```
[ux,uy] = pdegrad(p,t,u)
[ux,uy] = pdegrad(p,t,u,FaceID)
```

Description

`[ux,uy] = pdegrad(p,t,u)` returns the gradient of `u` evaluated at the center of each mesh triangle.

The gradient is the same everywhere in the triangle interior because `pdegrad` uses only linear basis functions. The derivatives at the boundaries of the triangles can be discontinuous.

`[ux,uy] = pdegrad(p,t,u,FaceID)` restricts the computation to the faces listed in `FaceID`.

Examples

Gradient of PDE Solution

Create a `[p,e,t]` mesh on the L-shaped membrane.

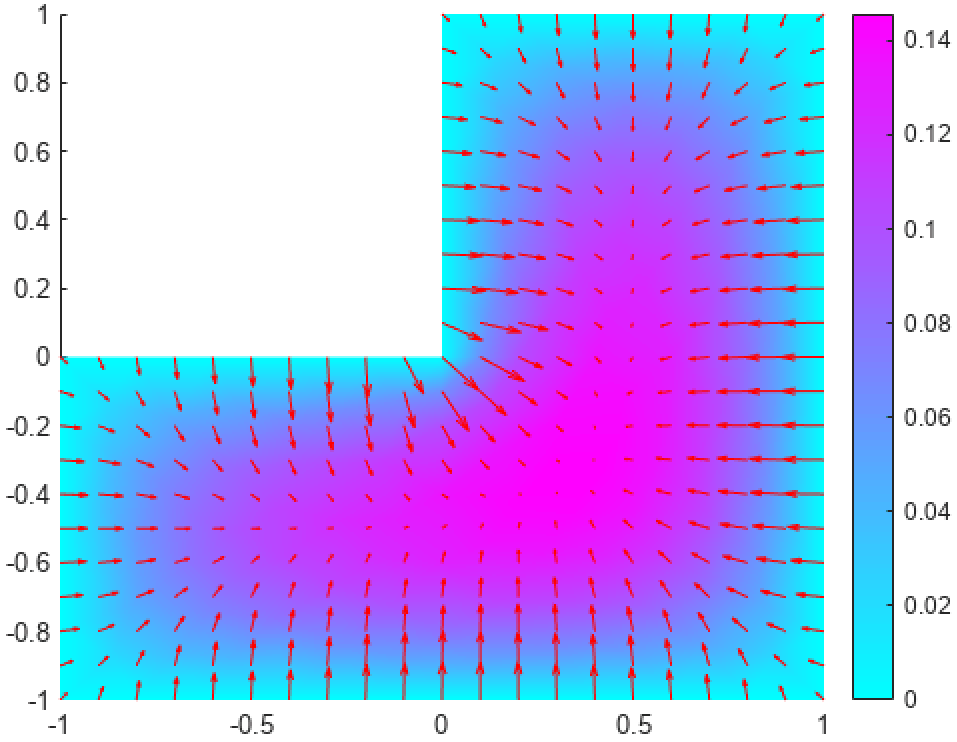
```
[p,e,t] = initmesh('lshapeg');
```

Solve the equation using the Dirichlet boundary condition $u = 0$ on $\partial\Omega$.

```
c = 1;
a = 0;
f = 1;
u = assempde('lshapeb',p,e,t,c,a,f);
```

Compute the gradient of the solution and plot the results.

```
[gradx,grady] = pdegrad(p,t,u);
pdeplot(p,e,t,'XYData',u,'FlowData',[gradx;grady])
```



Input Arguments

p — Mesh nodes
matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh elements
matrix

Mesh elements, specified as a 4-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

u — Data at nodes
column vector

Data at nodes, specified as a column vector.

For a PDE system of *N* equations and a mesh with `Np` node points, the first `Np` values of `u` describe the first component, the following `Np` values of `u` describe the second component, and so on.

Data Types: `double`

`FaceID` — Face IDs
vector of integers

Face IDs, specified as a vector of integers.

Data Types: `double`

Output Arguments

`ux` — *x*-component of the gradient of `u` evaluated at the center of each triangle
row vector | matrix

x-component of the gradient of `u` evaluated at the center of each triangle, returned as a row vector for a scalar PDE or a matrix for a system of PDEs. The number of elements in a row vector or columns in a matrix corresponds to the number `Nt` of mesh triangles. For a PDE system of N equations, each row i from 1 to N contains $\frac{\partial u_i}{\partial x}$.

`uy` — *y*-component of the gradient of `u` evaluated at the center of each triangle
row vector | matrix

y-component of the gradient of `u` evaluated at the center of each triangle, returned as a row vector for a scalar PDE or a matrix for a system of PDEs. The number of elements in a row vector or columns in a matrix corresponds to the number `Nt` of mesh triangles. For a PDE system of N equations, each row i from 1 to N contains $\frac{\partial u_i}{\partial y}$.

Version History

Introduced before R2006a

Not recommended
Not recommended starting in R2018a

`pdegrad` is not recommended. Use `evaluateGradient` instead. There are no plans to remove `pdegrad`.

See Also

`evaluateGradient` | `pdecgrad`

pdeInterpolant

Interpolant for nodal data to selected locations

Note `pdeInterpolant` and `[p,e,t]` representation of `FEMesh` data are 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.

Creation

Syntax

`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`.

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 2-175.

For 2-D problems, construct `p` using the `initmesh` function, or export from the **Mesh** menu of the PDE Modeler 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 2-175.

For 2-D problems, construct `t` using the `initmesh` function, or export from the **Mesh** menu of the PDE Modeler 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 Modeler 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
model = createpde;
gm = geometryFromEdges(model,g);
% Zero Dirichlet conditions
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",(1:gm.NumEdges), ...
                             "u",0);
[p,e,t] = initmesh(gm);
c = 1;
a = 0;
f = 1;
u = assempde(model,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 = 4×1

    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.

Version History

Introduced in R2014b

See Also

`evaluate` | `tri2grid`

Topics

“Mesh Data” on page 2-175

pdeintrp

(Not recommended) Interpolate mesh nodal data to triangle midpoints

---

Note `pdeintrp` is not recommended. Use `interpolateSolution` and `evaluateGradient` instead.

Syntax

```
ut = pdeintrp(p,t,un)
```

Description

`ut = pdeintrp(p,t,un)` uses the data `un` at mesh nodes to linearly interpolate data at mesh triangle midpoints.

`pdeintrp` and `pdeprtni` are not inverse functions because the interpolation introduces some averaging.

Examples

Data at Mesh Nodes and Triangle Midpoints

Solve the equation $-\Delta u = 1$ on the L-shaped membrane and interpolate the solution from nodes to triangle midpoints.

First, create a `[p,e,t]` mesh on the L-shaped membrane.

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

Solve the equation using the Dirichlet boundary condition $u = 0$ on $\partial\Omega$. The result is the solution at the mesh nodes.

```
un = assempde('lshapeb',p,e,t,1,0,1);
```

Interpolate the solution from the mesh nodes to the triangle midpoints.

```
ut = pdeintrp(p,t,un);
```

Interpolate the solution back to nodes by using the `pdeprtni` function. Compare the result and the original solution at the mesh nodes. The `pdeprtni` and `pdeintrp` functions are not inverse.

```
un2 = pdeprtni(p,t,ut);
isequal(un,un2)
```

```
ans = logical
   0
```

Input Arguments

`p` — Mesh nodes
matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

`t` — Mesh elements
matrix

Mesh elements, specified as a `4`-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

`un` — Data at nodes
column vector

Data at nodes, specified as a column vector.

For a PDE system of N equations and a mesh with `Np` node points, the first `Np` values of `un` describe the first component, the following `Np` values of `un` describe the second component, and so on.

Data Types: `double`

Output Arguments

`ut` — Data at triangle midpoints
row vector

Data at triangle midpoints, returned as a row vector.

For a PDE system of N equations and a mesh with `Nt` elements, the first `Nt` values of `ut` describe the first component, the following `Nt` values of `ut` describe the second component, and so on.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`pdeintrp` is not recommended. Use `interpolateSolution` and `evaluateGradient` instead. There are no plans to remove `pdeintrp`.

See Also

`pdeprtni` | `evaluate` | `pdeInterpolant`

Topics

“Mesh Data as [p,e,t] Triples” on page 2-172

pdejmps

(Not recommended) Error estimates for adaptation

Note `pdejmps` is not recommended. Use meshes represented as `FEMesh` objects instead of `[p,e,t]` meshes. For more information, see “Compatibility Considerations”.

Syntax

```
errf = pdejmps(p,t,c,a,f,u,alpha,beta,m)
```

Description

`errf = pdejmps(p,t,c,a,f,u,alpha,beta,m)` calculates the error indication function used for mesh adaptation. The columns of `errf` correspond to triangles, and the rows correspond to the equations in the PDE system.

The function computes the error indicator $E(K)$ for each triangle K as

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

where n_τ is the unit normal of edge τ and the braced term is the jump in flux across the element edge. Here, α and β are weight indices, and m is an order parameter. The norm is an L_2 norm computed over the element K.

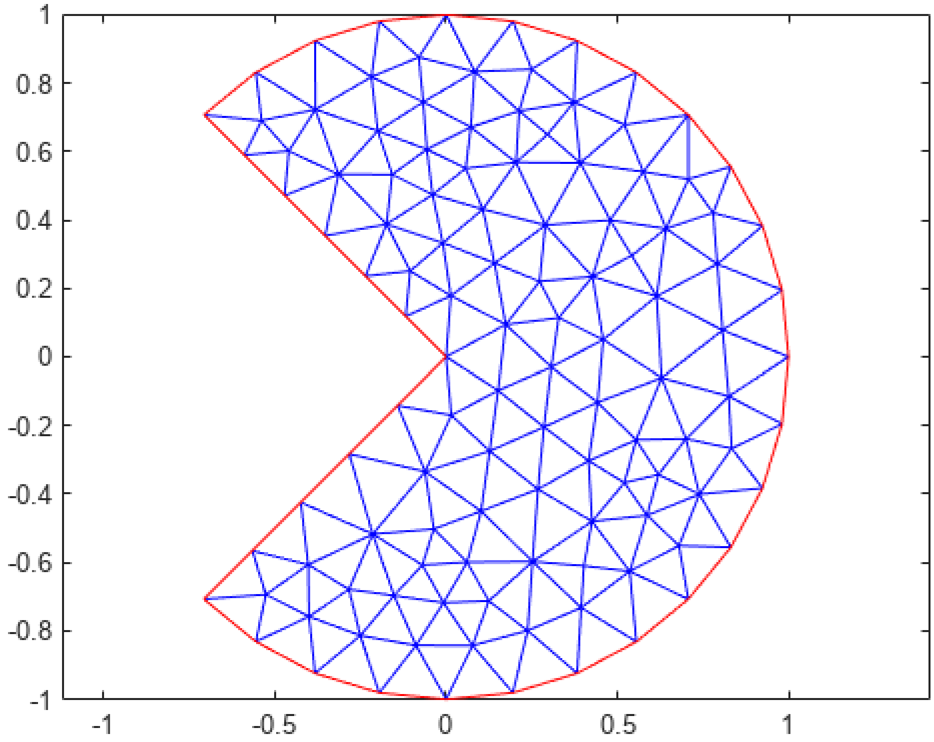
Examples

Error Indication Function

Solve the Laplace equation over a circle sector, with Dirichlet boundary conditions u = cos(2/3atan2(y,x)) along the arc and u = 0 along the straight lines. Use the original coarser mesh and the refined mesh, and calculate the error indication function in both cases.

Generate and plot a mesh for the circle sector geometry.

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



Solve the Laplace equation.

```
u = assempde('cirsb',p,e,t,1,0,0);
```

Calculate the error indication function for each mesh triangle. Use the weight indices $\alpha = 0.15$, $\beta = 0.15$, and the order parameter $m = 1$.

```
alpha = 0.15;
beta = 0.15;
m = 1;
errf = pdejmps(p,t,1,0,0,u,alpha,beta,m);
```

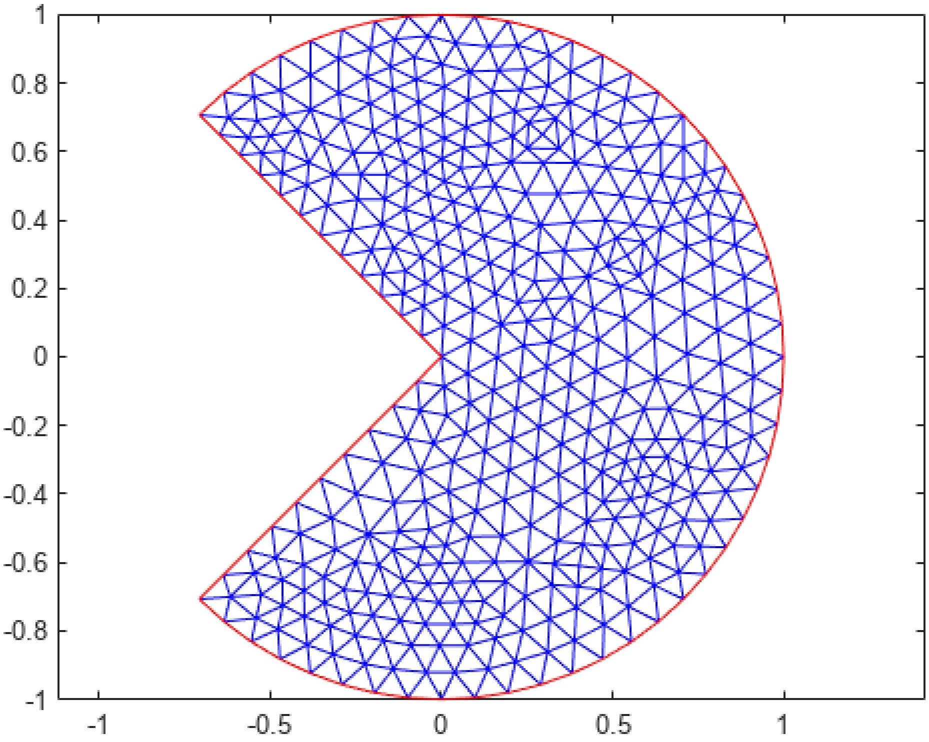
Find the maximum value of the error indication function.

```
max(abs(errf))
```

```
ans = 0.0306
```

Refine the original mesh and plot the result.

```
[p,e,t] = refinemesh('cirsg',p,e,t);
pdemesh(p,e,t)
```



Solve the same equation on the refined mesh, and calculate the error indication function for each mesh triangle. Use the same values for the weight indices and the order parameter.

```
u = assempde('cirsb',p,e,t,1,0,0);
errf = pdejmps(p,t,1,0,0,u,alpha,beta,m);
```

Find the maximum value of the error indication function.

```
max(abs(errf))
```

```
ans = 0.0194
```

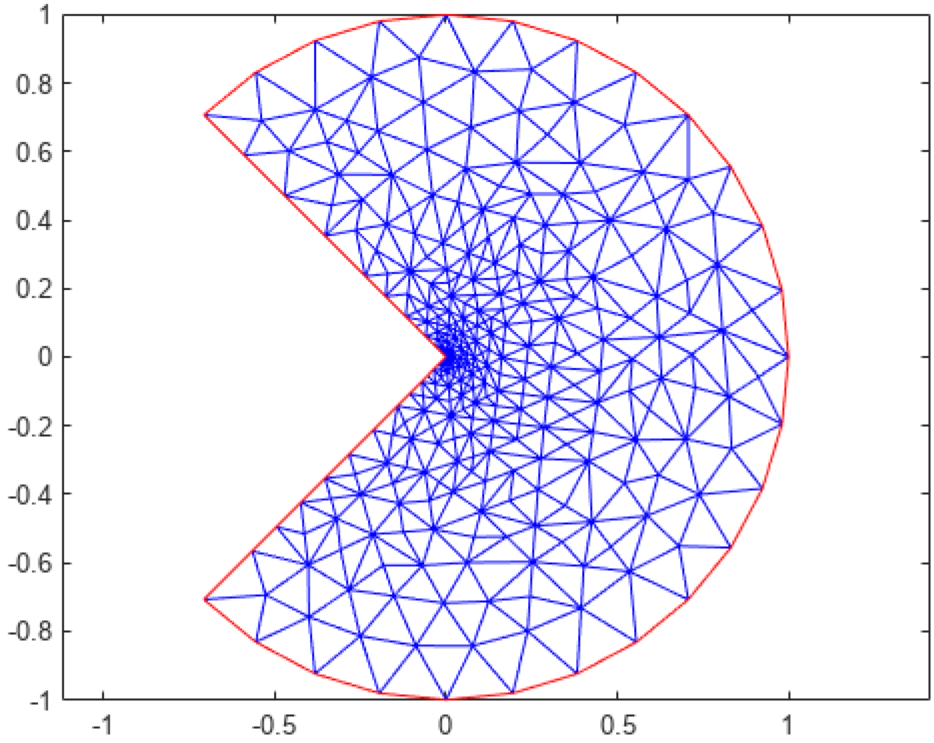
Solve the same equation using the `adaptmesh` function.

```
[u,p,e,t] = adaptmesh('cirsg','cirsb',1,0,0);
```

```
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 refinement passes obtained.
```

Plot the mesh.

```
pdemesh(p,e,t)
```



Calculate the error indication function for each mesh triangle.

```
errf = pdejmps(p,t,1,0,0,u,alpha,beta,m);
```

Find the maximum value of the error indication function.

```
max(abs(errf))
```

```
ans = 0.0024
```

Input Arguments

p — Mesh node points

matrix

Mesh node points, specified as a 2-by-`Np` matrix of points (nodes), where `Np` is the number of nodes in the mesh. For details on mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh elements

4-by-`Nt` matrix

Mesh elements, specified as a `4`-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on mesh data representation, see `initmesh`.

Data Types: `double`

`c` — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `c` represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

The coefficients `c`, `a`, and `f` can depend on the solution `u` if you use the nonlinear solver by setting the value of `'Nonlin'` to `'on'`. The coefficients cannot be functions of the time `t`.

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`

`a` — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `a` represents the *a* coefficient in the scalar PDE

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

or in the system of PDEs

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

The coefficients `c`, `a`, and `f` can depend on the solution `u` if you use the nonlinear solver by setting the value of `'Nonlin'` to `'on'`. The coefficients cannot be functions of the time `t`.

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`

`f` — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `f` represents the *f* coefficient in the scalar PDE

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

or in the system of PDEs

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

The coefficients `c`, `a`, and `f` can depend on the solution `u` if you use the nonlinear solver by setting the value of `'Nonlin'` to `'on'`. The coefficients cannot be functions of the time `t`.

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `string` | `function_handle`

`u` — PDE solution
vector

PDE solution, specified as a vector.

- If the PDE is scalar, meaning that it has 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, the next `Np` elements represent the solution of equation 2, and so on.

`alpha` — Weight index
number

Weight index, specified as number.

Data Types: `double`

`beta` — Weight index
number

Weight index, specified as a number.

Data Types: `double`

`m` — Order parameter
number

Order parameter, specified as number.

Data Types: `double`

Output Arguments

`errf` — Error indicator
matrix

Error indicator, returned as a matrix with the number of columns equal to the number of triangles `t` and the number of rows equal to the number of PDEs in the system.

- Each matrix row corresponds to an equation in the PDE system.
- Each column corresponds to a triangle.

Version History

Introduced before R2006a

`pdejmps` is not recommended

Not recommended starting in R2016a

`pdejmps` and `[p,e,t]` meshes are not recommended. Use meshes represented as `FEMesh` objects instead. There are no plans to remove `pdejmps` and `[p,e,t]` meshes.

Starting in R2016a, use the `generateMesh` function to create meshes as `FEMesh` objects. For details about these meshes, see “Mesh Data” on page 2-175.

See Also

`adaptmesh` | `initmesh` | `refinemesh`

pdemesh

Plot PDE mesh

Syntax

```
pdemesh(model)
pdemesh(mesh)
pdemesh(nodes,elements)
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(mesh)` plots the mesh defined as a `Mesh` property of a 2-D or 3-D `model` object of type `PDEModel`.

`pdemesh(nodes,elements)` plots the mesh defined by `nodes` and `elements`.

`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

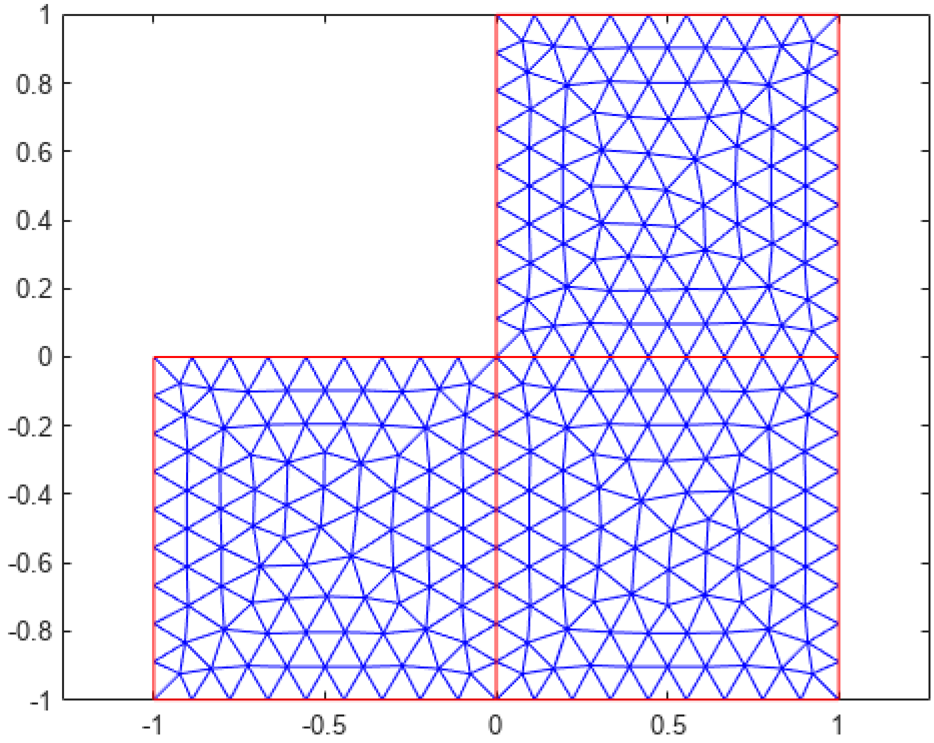
Create a mesh plot and display the node and element labels of the mesh.

Create a PDE model. Include the geometry of the built-in function `lshapeg`. Mesh the geometry.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
mesh = generateMesh(model);
```

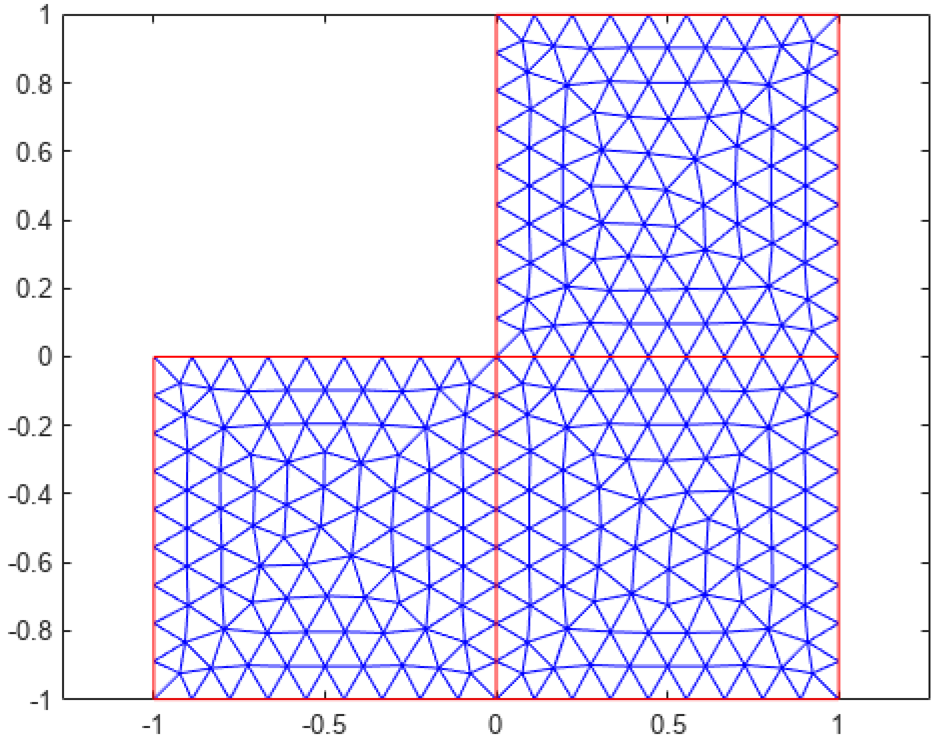
Plot the mesh.

```
pdemesh(model)
```



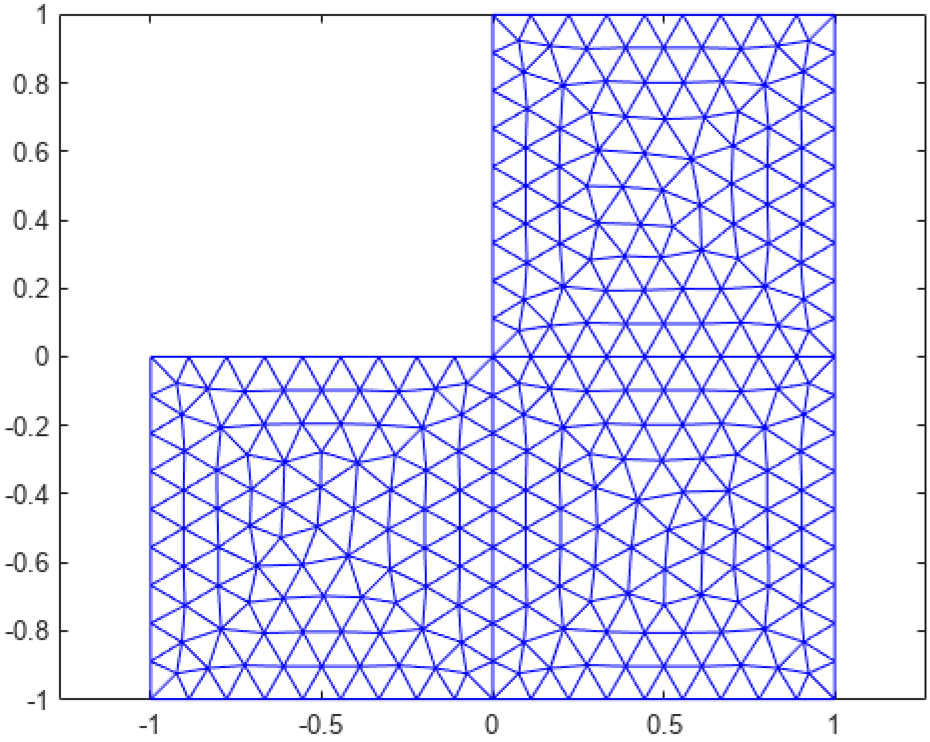
Alternatively, you can plot a mesh by using `mesh` as an input argument.

```
pdemesh(mesh)
```



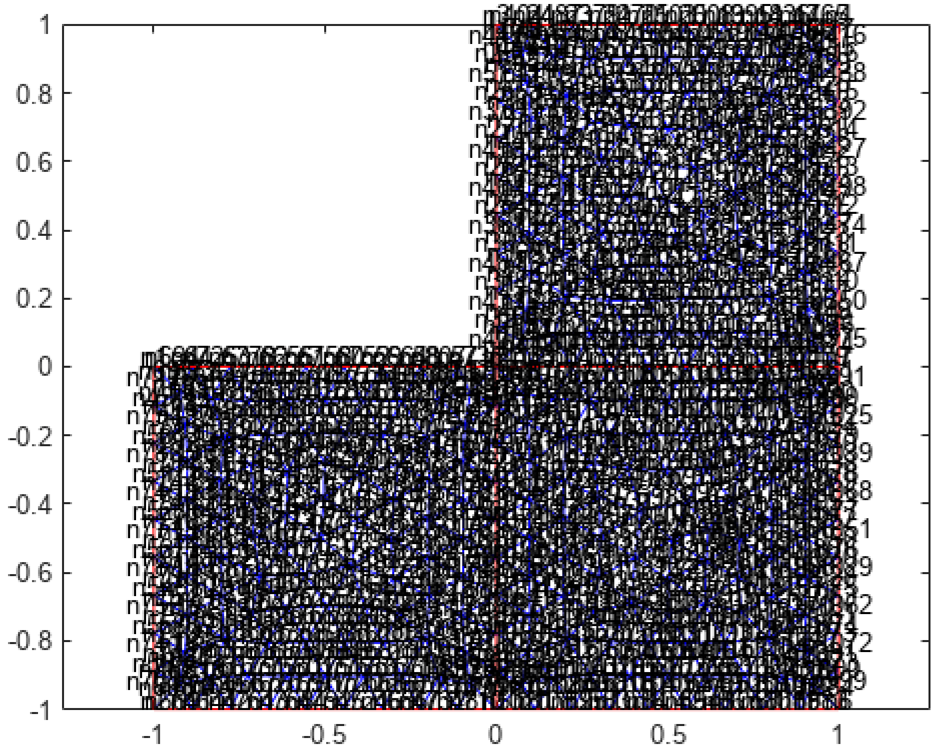
Another approach is to use the nodes and elements of the mesh as input arguments for `pdemesh`.

```
pdemesh(mesh.Nodes,mesh.Elements)
```



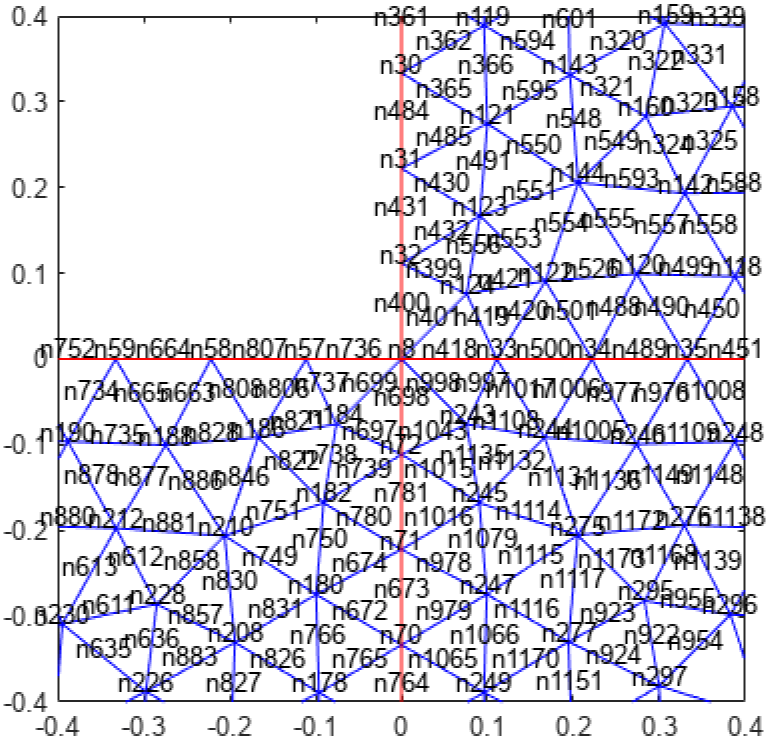
Display node labels.

```
pdemesh(model,"NodeLabels","on")
```



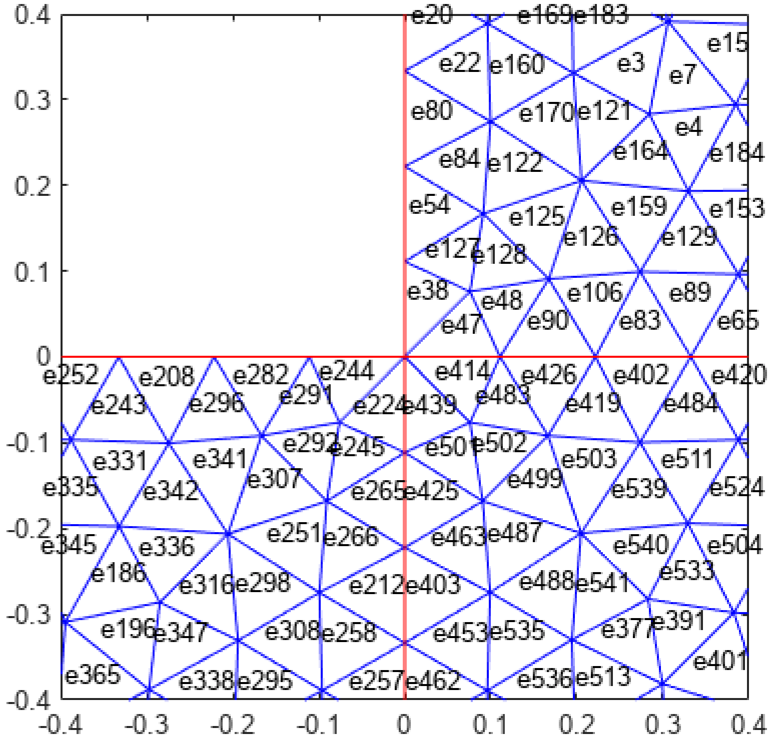
Use `xlim` and `ylim` to zoom in on particular nodes.

```
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.

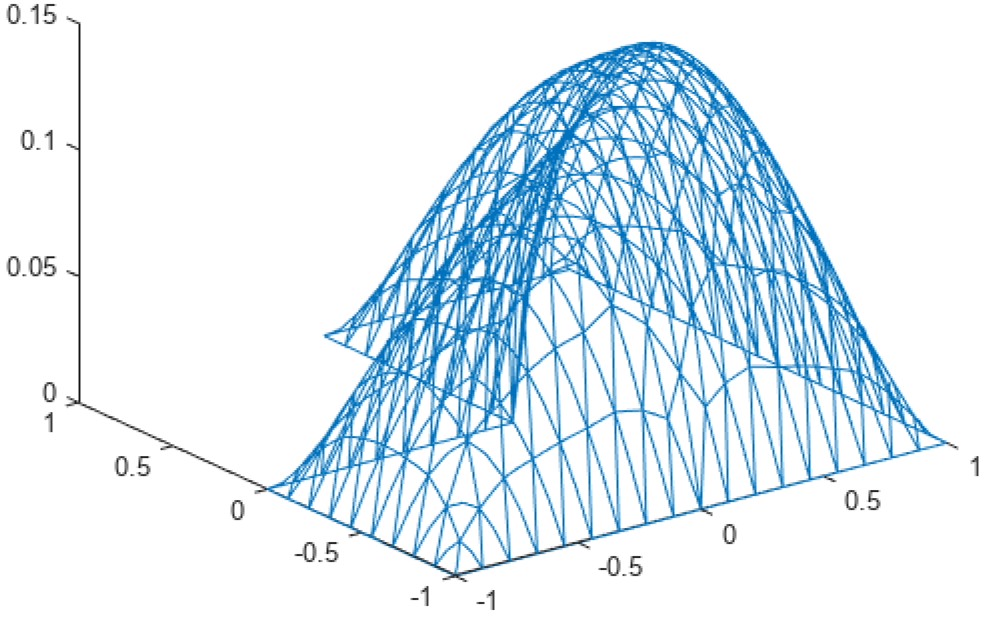
```
applyBoundaryCondition(model,"dirichlet", ...
                             "Edge",1:model.Geometry.NumEdges, ...
                             "u",0);
specifyCoefficients(model,"m",0,...
                          "d",0,...
                          "c",1,...
                          "a",0,...
                          "f",1);
generateMesh(model);
results = solvepde(model)
```

```
results = 
  StationaryResults with properties:

    NodalSolution: [1177x1 double]
       XGradients: [1177x1 double]
       YGradients: [1177x1 double]
       ZGradients: []
             Mesh: [1x1 FEMesh]
```

```
u = results.NodalSolution;
```

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.

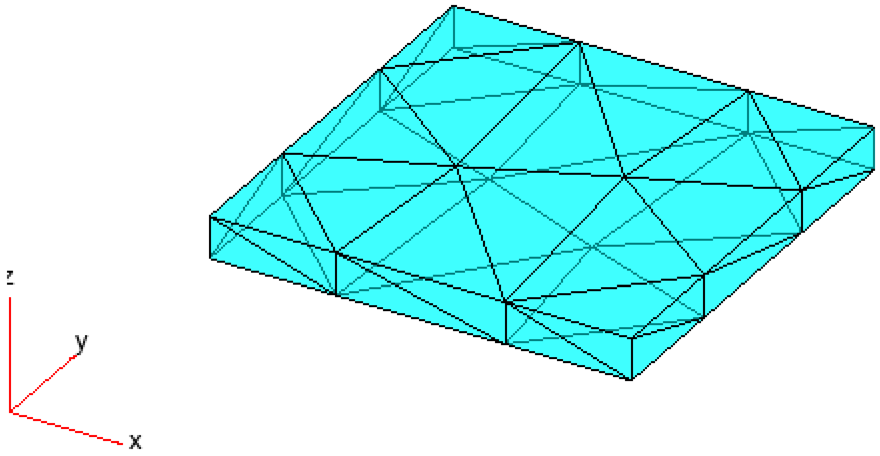
Transparency for 3-D Mesh

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)
```



Elements Associated with Particular Face

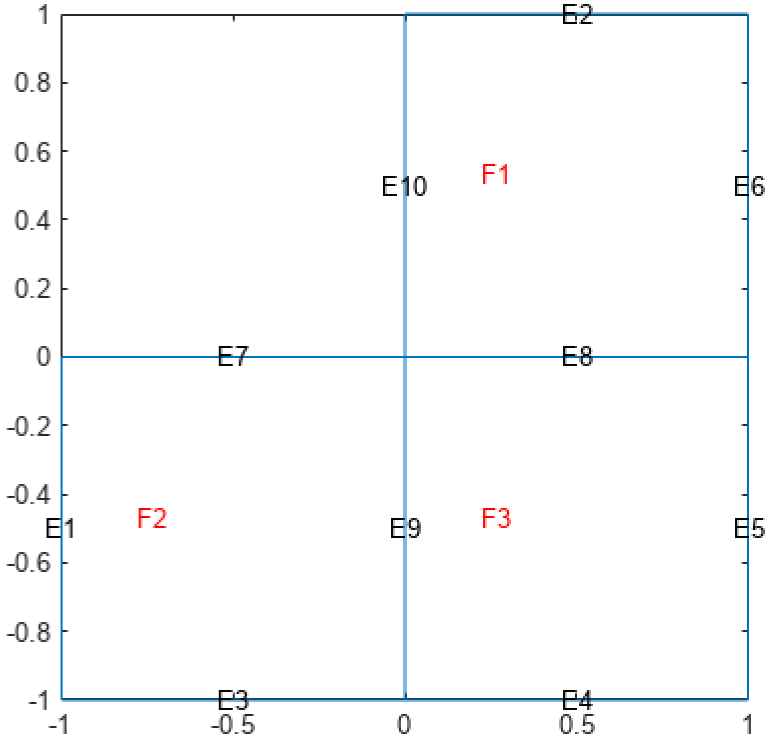
Find the elements associated with a geometric region.

Create a PDE model.

```
model = createpde;
```

Include the geometry of the built-in function `lshapeg`. Plot the geometry.

```
geometryFromEdges(model,@lshapeg);
pdegplot(model,"FaceLabels","on","EdgeLabels","on")
```



Generate a mesh.

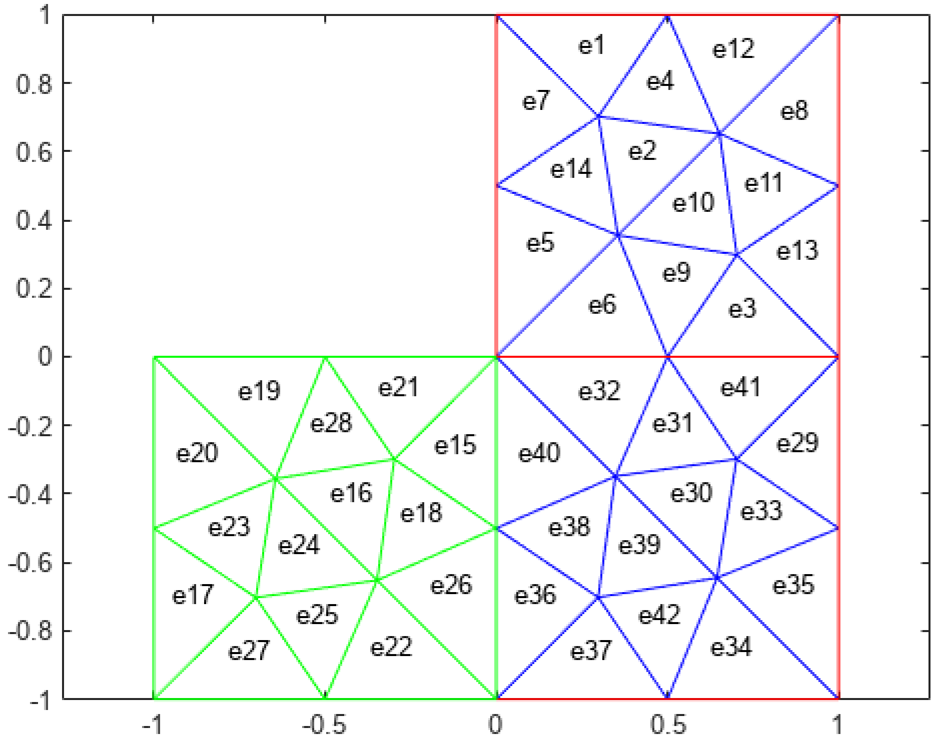
```
mesh = generateMesh(model,"Hmax",0.5);
```

Find the elements associated with face 2.

```
Ef2 = findElements(mesh,"region","Face",2);
```

Highlight these elements in green on the mesh plot.

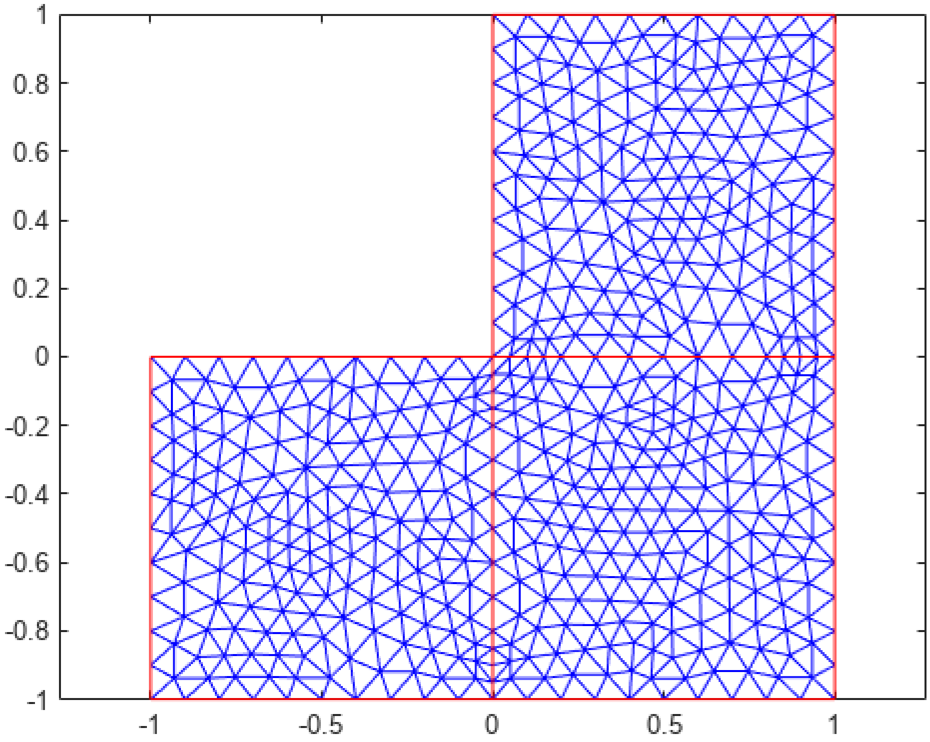
```
figure
pdemesh(mesh,"ElementLabels","on")
hold on
pdemesh(mesh.Nodes,mesh.Elements(:,Ef2),"EdgeColor","green")
```



[p,e,t] Mesh Plot

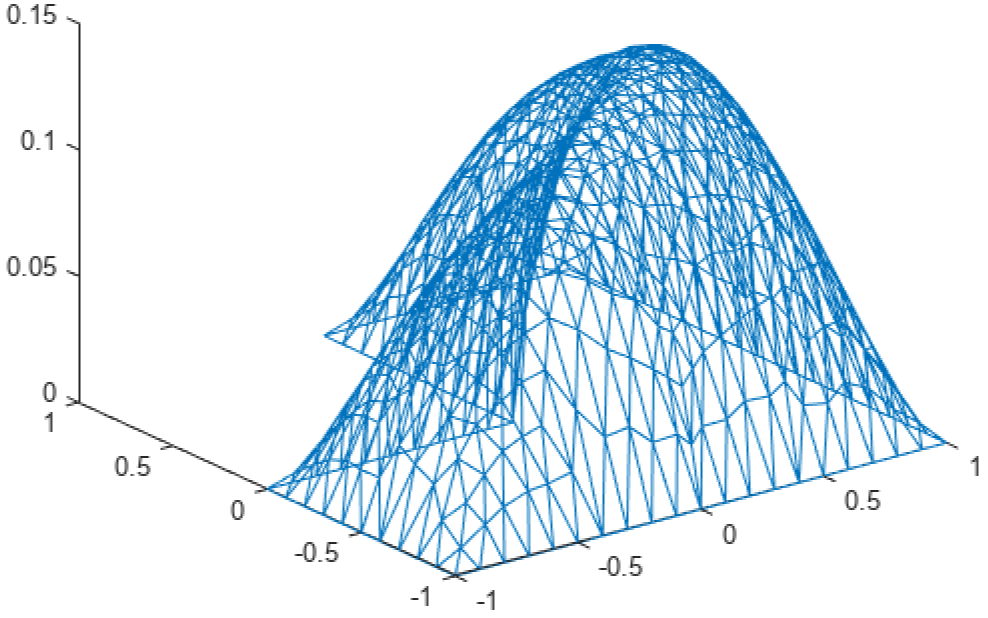
Plot the mesh for the geometry of the L-shaped membrane.

```
[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)
```



Input Arguments

`model` — Model container

`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectromagneticModel` object

Model container, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectromagneticModel` object.

Example: `model = createpde(3)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","electrostatic")`

`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);`

`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`

nodes — Nodal coordinates

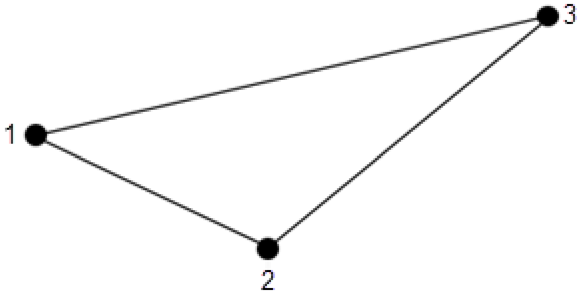
2-by-*NumNodes* matrix | 3-by-*NumNodes* matrix

Nodal coordinates, specified as a 2-by-*NumNodes* matrix for a 2-D mesh and 3-by-*NumNodes* matrix for a 3-D mesh. *NumNodes* is the number of nodes.

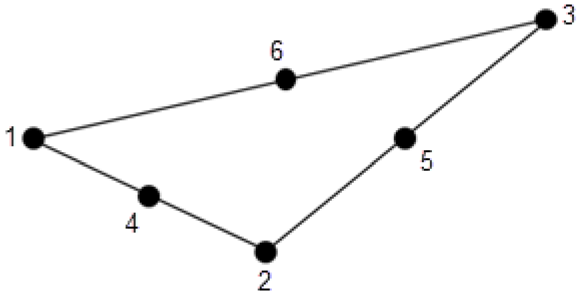
`elements` — Element connectivity matrix in terms of node IDs

NodesPerElem-by-*NumElements* matrix

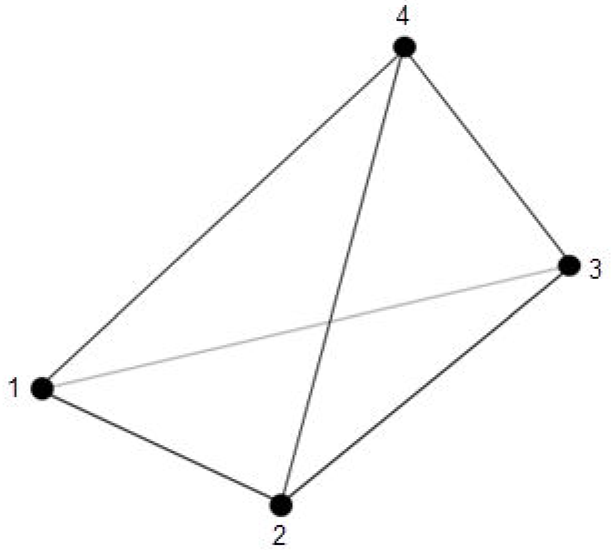
Element connectivity matrix in terms of node IDs, specified as an *NodesPerElem*-by-*NumElements* matrix. *NodesPerElem* is the number of nodes per element. Linear meshes contain only corner nodes, so there are three nodes per a 2-D element and four nodes per a 3-D element. Quadratic meshes contain corner nodes and nodes in the middle of each edge of an element. For quadratic meshes, there are six nodes per a 2-D element and 10 nodes per a 3-D element.



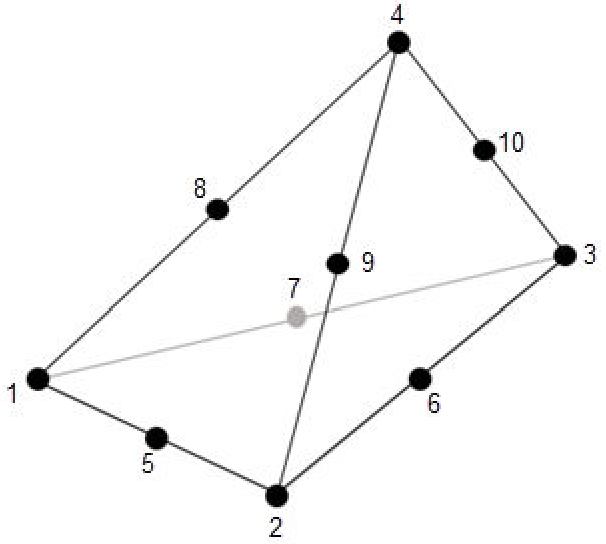
2-D linear element
showing node numbering



2-D quadratic element
showing node numbering



3-D linear element
showing node numbering



3-D quadratic element
showing node numbering

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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** app, or generated by `initmesh` or `refinemesh`.

Example: `[p,e,t] = initmesh(gd)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `pdemesh(model,"NodeLabels","on")`

`NodeLabels` — Node labels

`"off"` (default) | `"on"`

Node labels, specified as the comma-separated pair consisting of `"NodeLabels"` and `"off"` or `"on"`.

`pdemesh` ignores `NodeLabels` when you plot solution data as a 3-D plot.

Example: `"NodeLabels","on"`

Data Types: `char` | `string`

`ElementLabels` — Element labels

`"off"` (default) | `"on"`

Element labels, specified as the comma-separated pair consisting of `"ElementLabels"` and `"off"` or `"on"`.

`pdemesh` ignores `ElementLabels` when you plot solution data as a 3-D plot.

Example: `"ElementLabels","on"`

Data Types: `char` | `string`

`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`

`EdgeColor` — Color of mesh edges

short color name | long color name | RGB triplet

Color of mesh edges, specified as a short or long color name or an RGB triplet. By default, for 2-D meshes the edges within one face are blue (RGB triplet `[0 0 1]`) and the edges between faces are red (RGB triplet `[1 0 0]`). For 3-D meshes, the default edge color is black (RGB triplet `[0 0 0]`).

The short names and long names are character vectors that specify one of eight predefined colors. The RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color; the intensities must be in the range [0 1]. The following table lists the predefined colors and their RGB triplet equivalents.

RGB Triplet	Short Name	Long Name
`[1 1 0]`	`y`	`yellow`
`[1 0 1]`	`m`	`magenta`
`[0 1 1]`	`c`	`cyan`
`[1 0 0]`	`r`	`red`
`[0 1 0]`	`g`	`green`
`[0 0 1]`	`b`	`blue`
`[1 1 1]`	`w`	`white`
`[0 0 0]`	`k`	`black`

Example: `"EdgeColor","green"`

Data Types: `double` | `char` | `string`

`FaceColor` — Color of mesh faces for 3-D meshes
`[0 1 1]` | short color name | long color name | RGB triplet

Color of mesh faces for 3-D meshes, specified as a short or long color name or an RGB triplet. The default face color is cyan (RGB triplet `[0 1 1]`). For details about available colors, see “EdgeColor” on page 5-0 .

Example: `"FaceColor","green"`

Data Types: `double` | `char` | `string`

Output Arguments

h — Handles to graphics objects
vector

Handles to graphics objects, returned as a vector.

Version History

Introduced before R2006a

Improved performance for plots with many text labels
Performance change in R2020a

`pdemesh` shows faster rendering and better responsiveness for plots that display many text labels. Code containing `findobj(fig,'Type','Text')` no longer returns labels on figures produced by `pdemesh`.

Highlighting particular nodes and elements on mesh plots

`pdemesh` accepts node and element IDs as input arguments, letting you highlight particular nodes and elements on mesh plots.

Transparency, node and element labels

You can now set plot transparency by using `FaceAlpha`, and display node and element labels by using `NodeLabels` and `ElementLabels`, respectively.

See Also

`pdeplot` | `pdeplot3D` | `pdegplot`

Topics

"Mesh Data" on page 2-175

PDEModel

PDE model object

Description

A `PDEModel` object contains information about a PDE problem: the number of equations, geometry, mesh, and boundary conditions.

Creation

Create a `PDEModel` object using `createpde`. Initially, the only nonempty property is `PDESystemSize`. It is `1` for scalar problems.

Properties

`PDESystemSize` — Number of equations
1 (default) | positive integer

Number of equations, *N*, specified as a positive integer. See “Equations You Can Solve Using PDE Toolbox” on page 1-3.

Example: `1`

Data Types: `double`

`BoundaryConditions` — PDE boundary conditions
vector of `BoundaryCondition` objects

PDE boundary conditions, specified as a vector of BoundaryCondition objects. You create boundary conditions using the `applyBoundaryCondition` function

`Geometry` — Geometry description
`AnalyticGeometry` | `DiscreteGeometry`

Geometry description, specified as `AnalyticGeometry` for a 2-D geometry or `DiscreteGeometry` for a 2-D or 3-D geometry.

`Mesh` — Mesh for solution
`FEMesh` object

Mesh for solution, specified 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, specified 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, specified as a vector of CoefficientAssignment objects. See `specifyCoefficients`.

`InitialConditions` — Initial conditions or initial solution
`GeometricInitialConditions` object | `NodalInitialConditions` object

Initial conditions or initial solution, specified 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, specified 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 from decomposed geometry matrix
geometryFromMesh	Create 2-D or 3-D geometry from mesh
importGeometry	Import geometry from STL or STEP file
setInitialConditions	Give initial conditions or initial solution
specifyCoefficients	Specify coefficients in a PDE model
solvepde	Solve PDE specified in a PDEModel
solvepdeeig	Solve PDE eigenvalue problem specified in a PDEModel

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: 0
                 Geometry: []
     EquationCoefficients: []
       BoundaryConditions: []
        InitialConditions: []
                     Mesh: []
            SolverOptions: [1x1 pde.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: [12913x1 double]
       XGradients: [12913x1 double]
       YGradients: [12913x1 double]
       ZGradients: [12913x1 double]
             Mesh: [1x1 FEMesh]
```

Version History

Introduced in R2015a

See Also

`createpde` | `applyBoundaryCondition` | `generateMesh` | `geometryFromEdges` | `geometryFromMesh` | `importGeometry` | `pdegplot` | `pdeplot` | `pdeplot3D` | `setInitialConditions` | `specifyCoefficients`

Topics

“Solve Problems Using PDEModel Objects” on page 2-2

pdenonlin

(Not recommended) Solve nonlinear elliptic PDE problem

Note `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

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

`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.

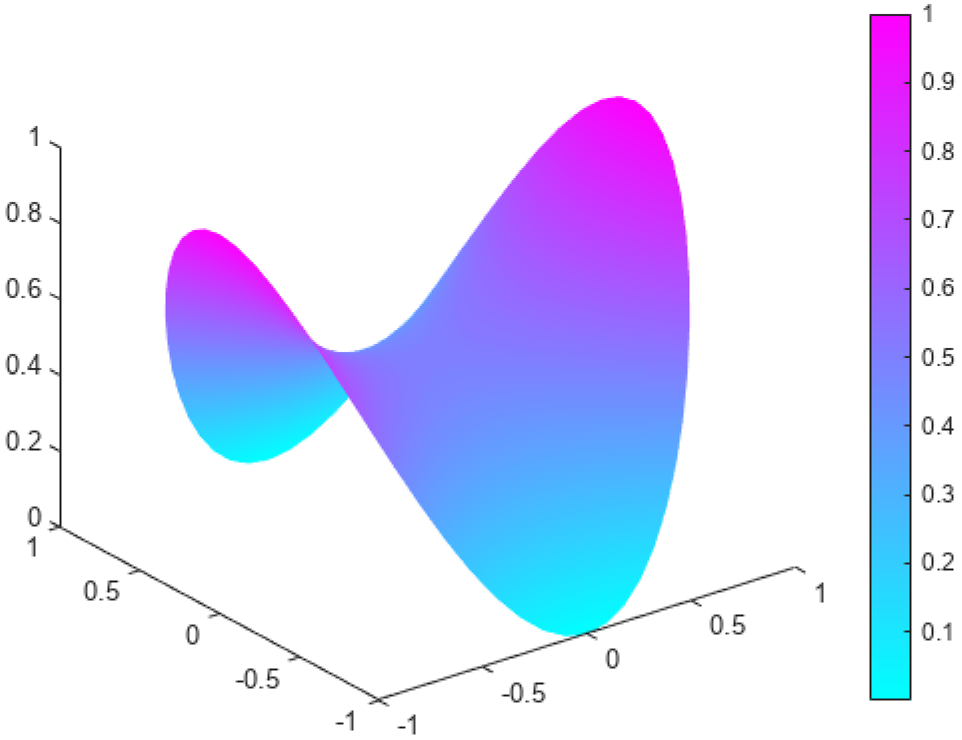
```
a = 0;
f = 0;
c = '1./sqrt(1+ux.^2+uy.^2)';
```

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,'GeometricOrder','linear','Hmax',0.1);
u = pdenonlin(model,c,a,f);
pdeplot(model,'XYData',u,'ZData',u)
```

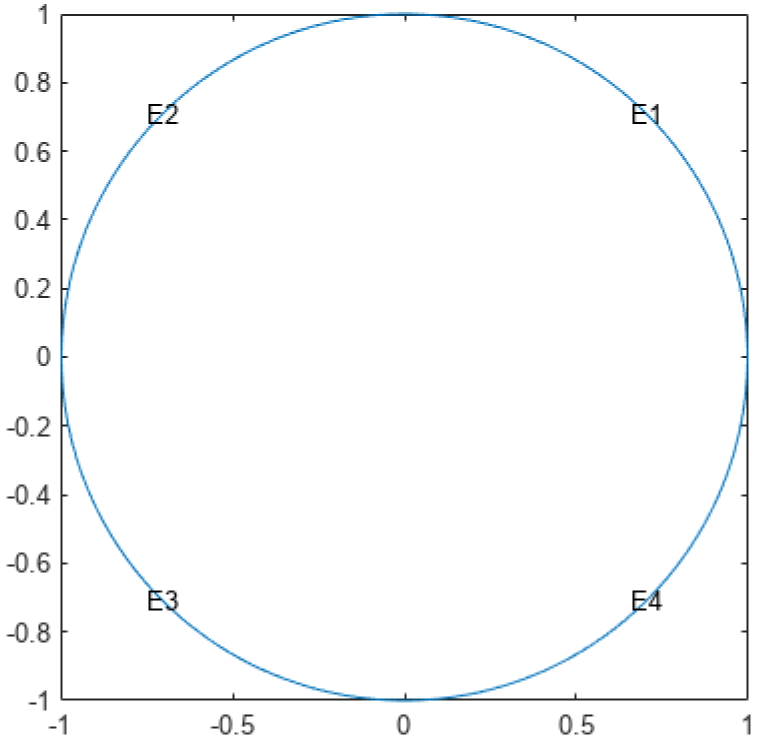


Minimal Surface Problem Using [p,e,t] Mesh

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.

```
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;
```

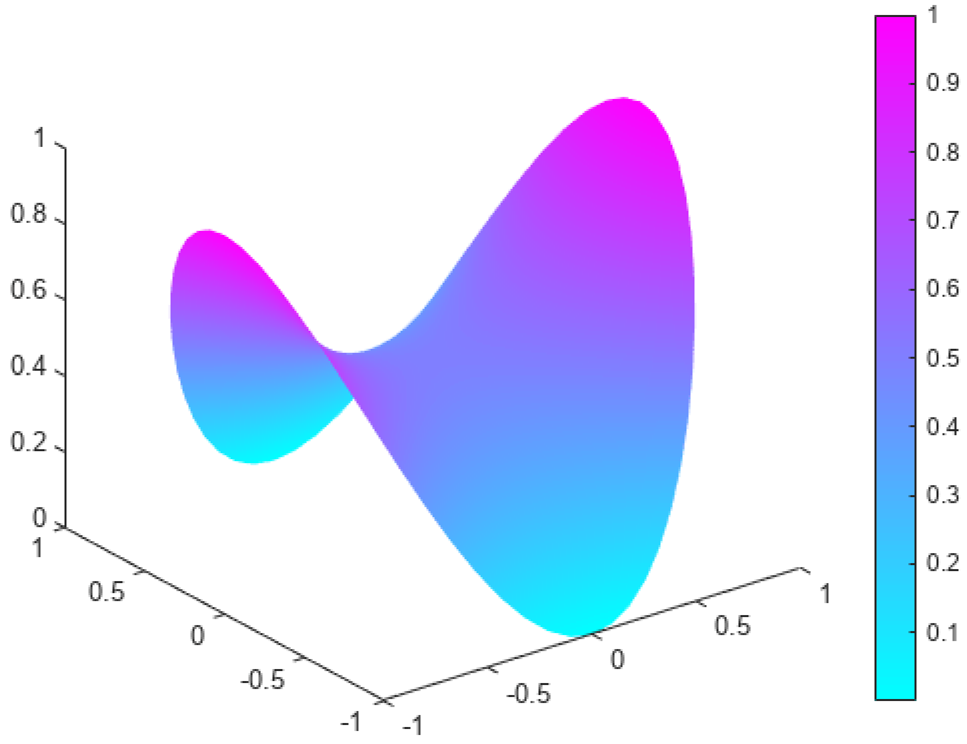
```
    end
end
```

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)
```



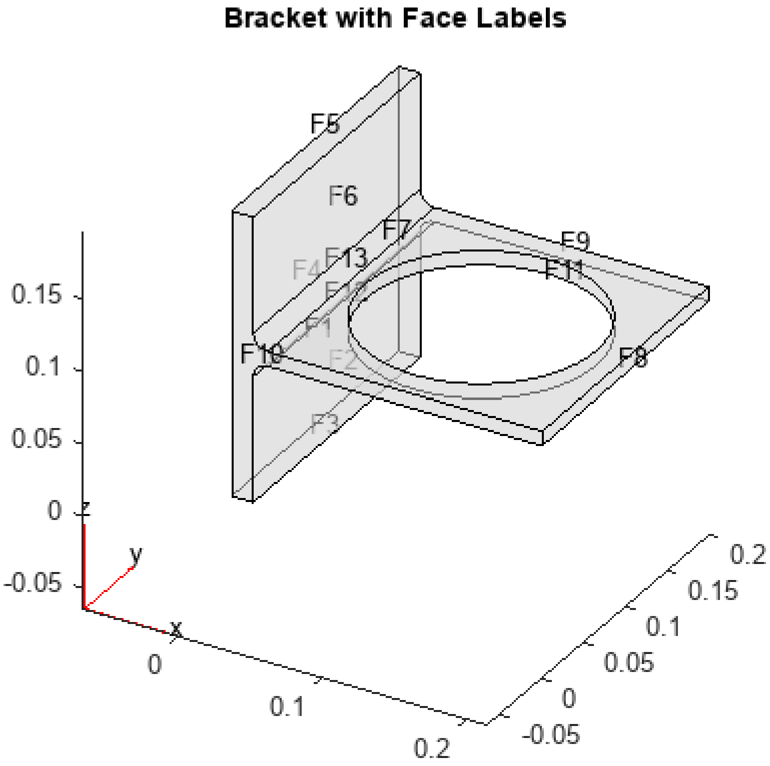
Nonlinear Problem with 3-D Geometry

Solve a nonlinear 3-D problem with nontrivial geometry.

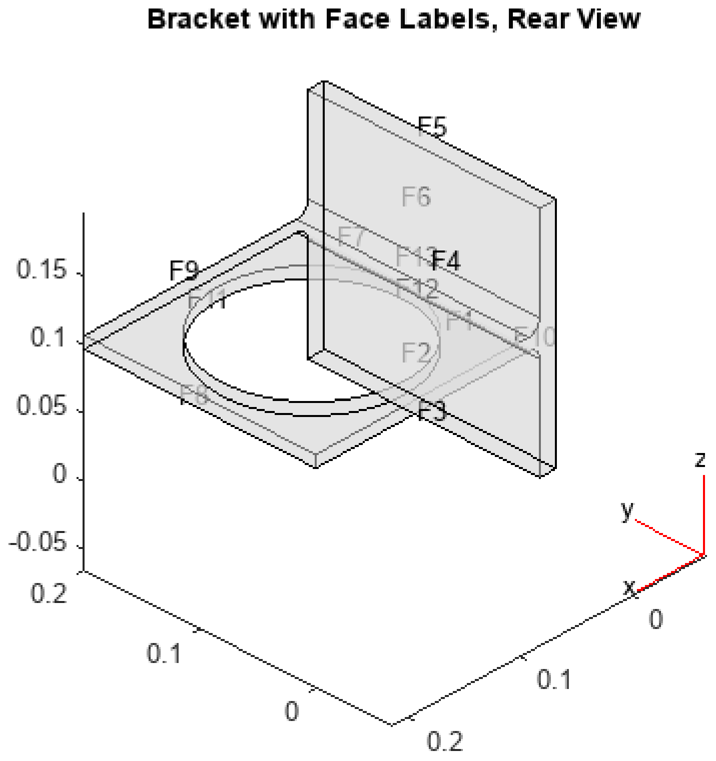
Import the geometry from the `BracketWithHole.stl` file. Plot the geometry and face labels.

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

```
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
view(30,30)
title('Bracket with Face Labels')
```



```
figure
pdegplot(model,'FaceLabels','on','FaceAlpha',0.5)
view(-134,-32)
title('Bracket with Face Labels, Rear View')
```



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';
```

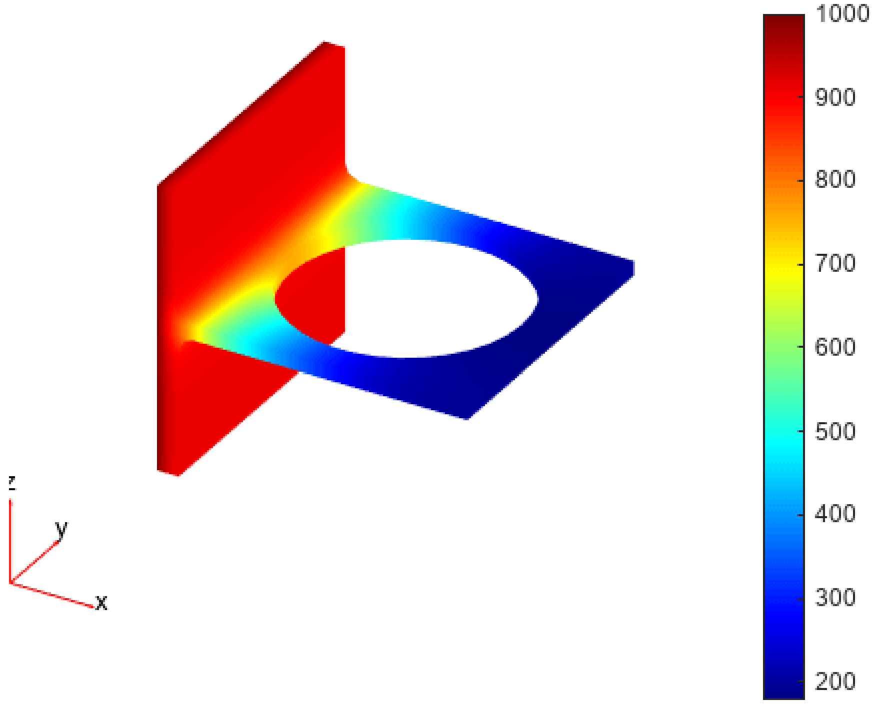
Generate the mesh and solve the PDE. Start from the initial guess `u0 = 1000`, which matches the value you set on face 4. Turn on the `Report` option to observe the convergence during the solution.

```
generateMesh(model);
u = pdenonlin(model,c,a,f,'U0',1000,'Report','on');
```

```
Iteration     Residual     Step size  Jacobian: full
   0          7.2059e-01
   1          1.3755e-01   1.0000000
   2          4.0800e-02   1.0000000
   3          1.1344e-02   1.0000000
   4          2.2739e-03   1.0000000
   5          1.7786e-04   1.0000000
   6          1.4054e-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`

`c` — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `c` represents the *c* coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `'cosh(x+y.^2)'`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

a — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `a` represents the a coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `2*eye(3)`

Data Types: `double` | `char` | `string` | `function_handle`
Complex Number Support: Yes

f — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function. `f` represents the f coefficient in the scalar PDE

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

or in the system of PDEs

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

Example: `char('sin(x)';'cos(y)';'tan(z)')`

Data Types: `double` | `char` | `string` | `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 Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** app, or generated by `initmesh` or `refinemesh`.

Example: `[p,e,t] = initmesh(gd)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Jacobian','full'`

`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 5-923. This approximation is based on the numerical differentiation of the coefficients.

Example: `u = pdenonlin(model,c,a,f,'Jacobian','full')`

Data Types: `char` | `string`

`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. A scalar specifies a constant initial condition for either a scalar or PDE system.

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 as [p,e,t] Triples” on page 2-172.

The first $\mathtt{N}_\mathtt{p}$ elements contain the values of component 1, where the value of element `k` corresponds to node `p(k)`. The next $\mathtt{N}_\mathtt{p}$ points contain the values of component 2, etc. It can be convenient to first represent the initial conditions `u0` as an $\mathtt{N}_\mathtt{p}$-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` | `string`
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` | `string`

`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` | `string`

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 `i`th column of `model.Mesh.Nodes` or the `i`th 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 “3-D Solution and Gradient Plots with MATLAB Functions” on page 3-344.

`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 5-923.

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

Nonlinear Equations

The basic idea is to use Gauss-Newton iterations to solve the nonlinear equations. Say you are trying to solve the equation

$$r(u) = -\nabla \cdot (c(u)\nabla u) + a(u)u - f(u) = 0.$$

In the FEM setting you solve the weak form of $r(u) = 0$. Set as usual

$$u(\mathbf{x}) = \sum U_j \phi_j$$

where $\mathbf{x}$ represents a 2-D or 3-D point. Then multiply the equation by an arbitrary test function ϕ_i, integrate on the domain Ω, and use Green's formula and the boundary conditions to obtain

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

which has to hold for all indices i.

The residual vector $\rho(U)$ can be easily computed as

$\rho(U) = (K + M + Q)U - (F + G)$

where the matrices K, M, Q and the vectors F and G are produced by assembling the problem

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

Assume that you have a guess $U^{(n)}$ of the solution. If $U^{(n)}$ is close enough to the exact solution, an improved approximation $U^{(n+1)}$ is obtained by solving the linearized problem

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

where α is a positive number. (It is not necessary that $\rho(U) = 0$ have a solution even if $\rho(u) = 0$ has.) In this case, the Gauss-Newton iteration tends to be the minimizer of the residual, i.e., the solution of $\min_U \|\rho(U)\|$.

It is well known that for sufficiently small α

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

and

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

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

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

where $\alpha \le 1$ is chosen as large as possible such that the step has a reasonable descent.

The *Gauss-Newton method* is local, and convergence is assured only when $U^{(0)}$ is close enough to the solution. In general, the first guess may be outside the region of convergence. To improve convergence from bad initial guesses, a *damping* strategy is implemented for choosing α, the *Armijo-Goldstein line search*. It chooses the largest damping coefficient α out of the sequence 1, 1/2, 1/4, . . . such that the following inequality holds:

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

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 ith column of J_n can be approximated by

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

which implies the assembling of the FEM matrices for the triangles containing grid point i. A very simple approximation to J_n, which gives a fixed point iteration, is also possible as follows. Essentially, for a given $U^{(n)}$, compute the FEM matrices K and F and set

$$U^{(n+1)} = K^{-1}F\ .$$

This is equivalent to approximating the Jacobian with the stiffness matrix. Indeed, since $\rho(U^{(n)}) = KU^{(n)} - F$, putting $J_n = K$ yields

$$U^{(n+1)} = U^{(n)} - J_n^{-1}\rho\left(U^{(n)}\right) = U^{(n)} - K^{-1}\left(KU^{(n)} - F\right) = K^{-1}F$$

In many cases the convergence rate is slow, but the cost of each iteration is cheap.

The Partial Differential Equation Toolbox nonlinear solver also provides for a compromise between the two extremes. To compute the derivative of the mapping $U \rightarrow KU$, proceed as follows. The *a* term has been omitted for clarity, but appears again in the final result.

$$\begin{aligned}\frac{\partial (KU)_i}{\partial U_j} &= \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \sum_l \Bigg(\int_\Omega c(U + \varepsilon \phi_j) \nabla \phi_l \nabla \phi_i \, d\mathbf{x} (U_l + \varepsilon \delta_{l,\,j}) \\ &\qquad - \int_\Omega c(U) \nabla \phi_l \nabla \phi_i \, d\mathbf{x} U_l \Bigg) \\ &= \int_\Omega c(U) \nabla \phi_j \nabla \phi_i \, d\mathbf{x} + \sum_l \int_\Omega \phi_j \frac{\partial c}{\partial u} \nabla \phi_l \nabla \phi_i \, d\mathbf{x} U_l\end{aligned}$$

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 $\widetilde{U}$ 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.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`pdenonlin` is not recommended. Use `solvepde` instead. There are no plans to remove `pdenonlin`.

See Also

`solvepde`

pdeplot

Plot solution or mesh for 2-D problem

Syntax

```
pdeplot(model,"XYData",results.NodalSolution)
pdeplot(model,"XYData",results.Temperature,"ColorMap","hot")
pdeplot(
model,"XYData",results.VonMisesStress,"Deformation",results.Displacement)
pdeplot(model,"XYData",results.ModeShapes.ux)
pdeplot(model,"XYData",results.ElectricPotential)

pdeplot(model)
pdeplot(mesh)
pdeplot(nodes,elements)

pdeplot(p,e,t)

pdeplot(___,Name,Value)
h = pdeplot(___)
```

Description

`pdeplot(model,"XYData",results.NodalSolution)` plots the solution of a `model` at nodal locations as a colored surface plot using the default `"cool"` colormap.

`pdeplot(model,"XYData",results.Temperature,"ColorMap","hot")` plots the temperature at nodal locations for a 2-D thermal analysis model. This syntax creates a colored surface plot using the `"hot"` colormap.

`pdeplot( model,"XYData",results.VonMisesStress,"Deformation",results.Displacement)` plots the von Mises stress and shows the deformed shape for a 2-D structural analysis model.

`pdeplot(model,"XYData",results.ModeShapes.ux)` plots the x-component of the modal displacement for a 2-D structural modal analysis model.

`pdeplot(model,"XYData",results.ElectricPotential)` plots the electric potential at nodal locations for a 2-D electrostatic analysis model.

`pdeplot(model)` plots the mesh specified in `model`.

`pdeplot(mesh)` plots the mesh defined as a `Mesh` property of a 2-D `model` object of type `PDEModel`.

`pdeplot(nodes,elements)` plots the mesh defined by its `nodes` and `elements`.

`pdeplot(p,e,t)` plots the mesh described by `p`,`e`, and `t`.

`pdeplot(___,Name,Value)` plots the mesh, the data at the nodal locations, or both the mesh and the data, depending on the `Name,Value` pair arguments. Use any arguments from the previous syntaxes.

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.

- For a thermal model, you can plot temperature or gradient of temperature.
- For a structural model, you can plot displacement, stress, strain, and von Mises stress. In addition, you can show the deformed shape and specify the scaling factor for the deformation plot.
- For an electromagnetic model, you can plot electric or magnetic potentials, fields, and flux densities.

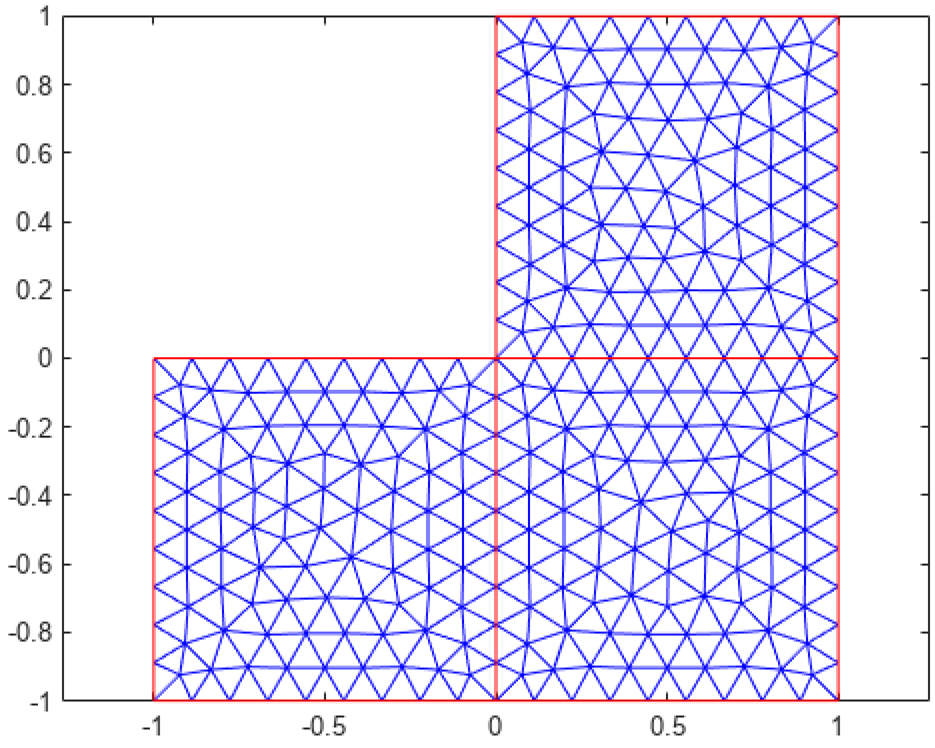
`h = pdeplot( ___ )` returns a handle to a plot, using any of the previous syntaxes.

Examples

2-D Mesh Plot

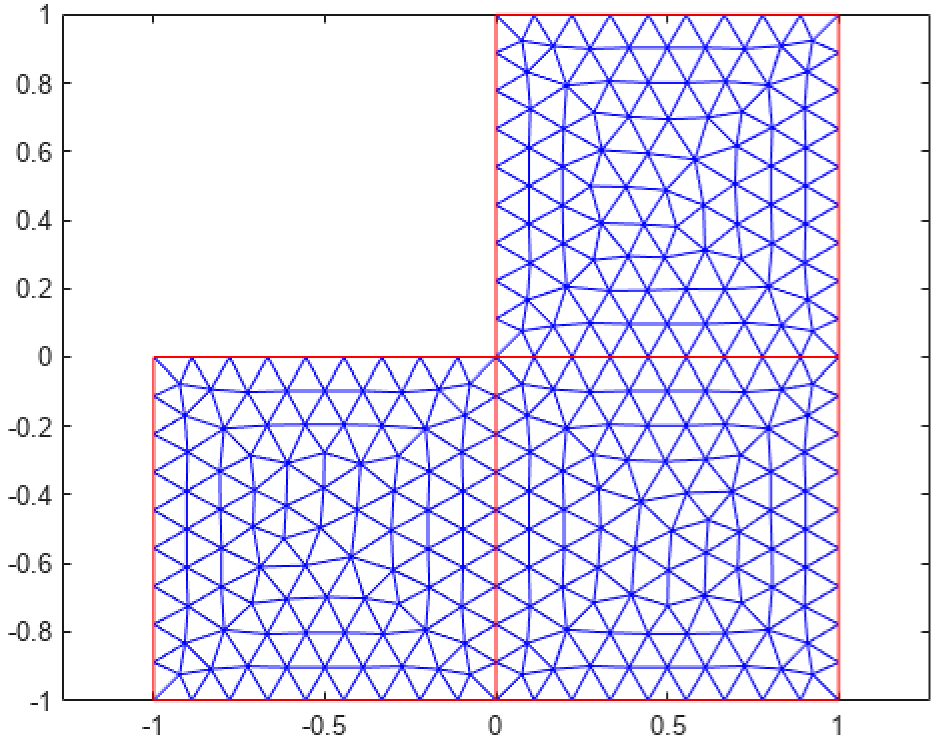
Create a PDE model. Include the geometry of the built-in function `lshapeg`. Mesh the geometry and plot it.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
mesh = generateMesh(model);
pdeplot(model)
```



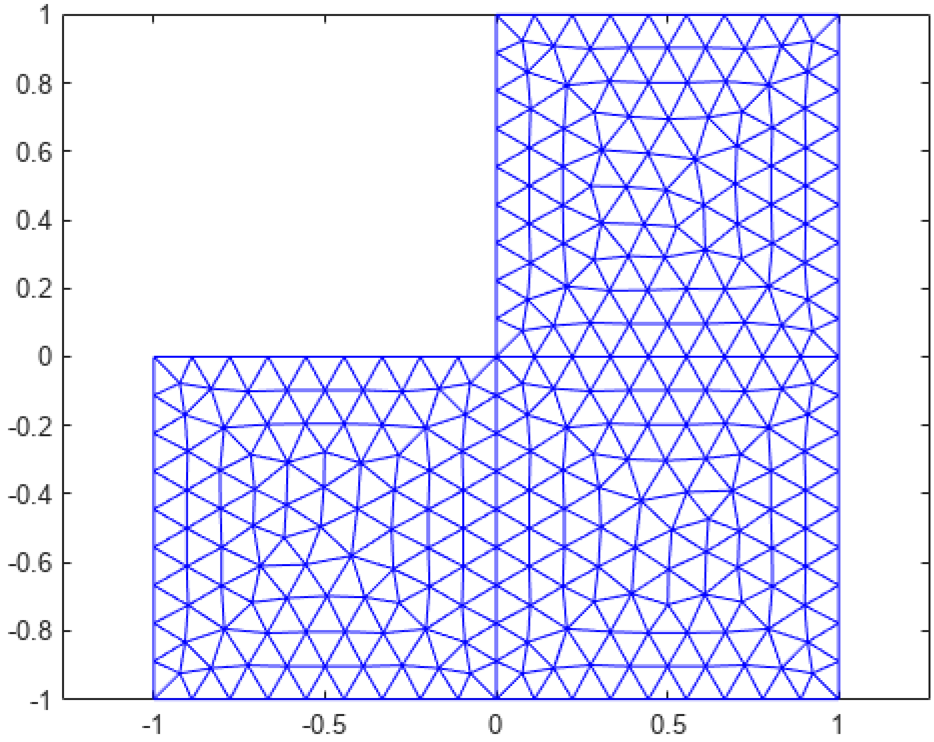
Alternatively, you can plot a mesh by using `mesh` as an input argument.

```
pdeplot(mesh)
```



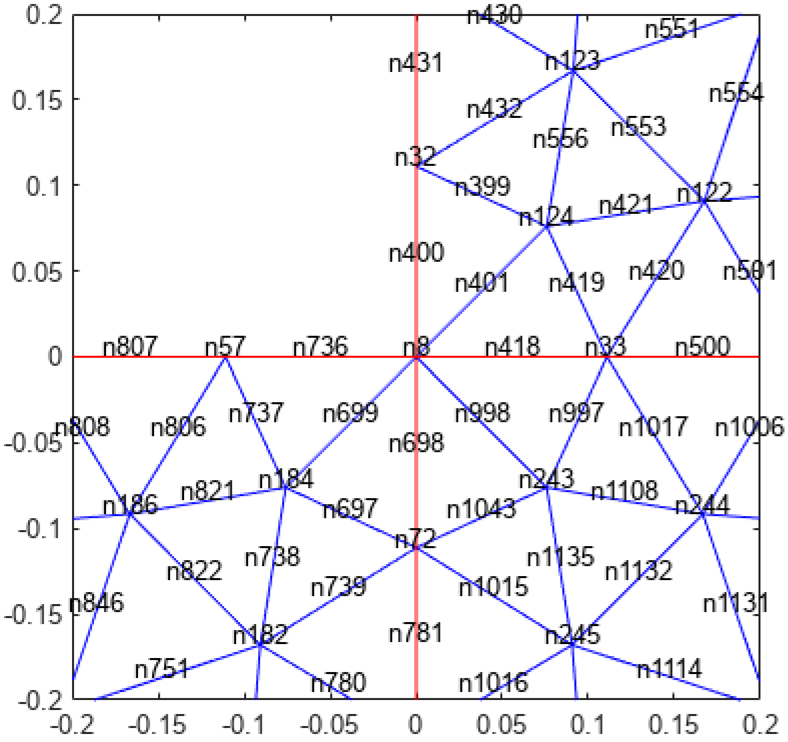
Another approach is to use the nodes and elements of the mesh as input arguments for `pdeplot`.

```
pdeplot(mesh.Nodes,mesh.Elements)
```



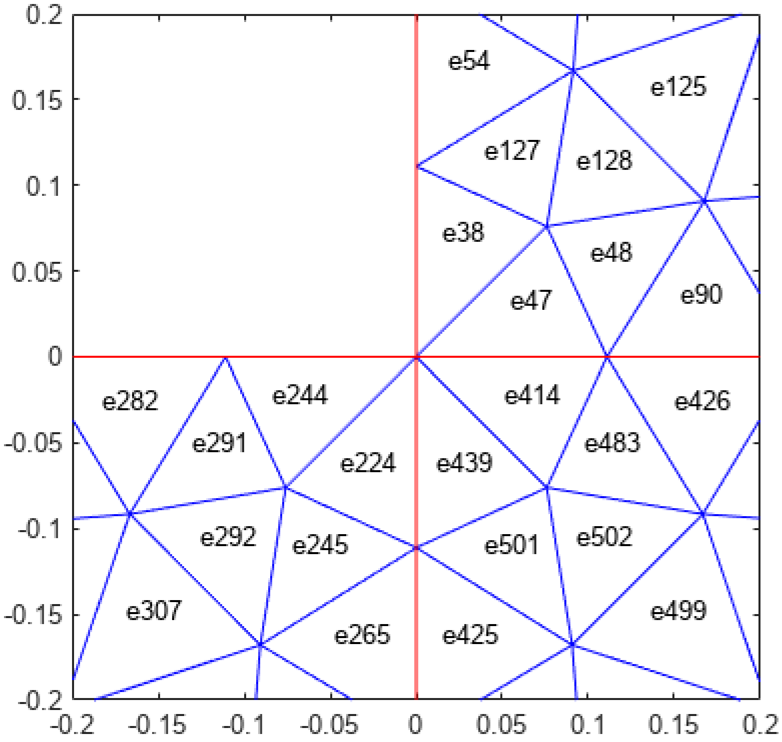
Display the node labels. Use `xlim` and `ylim` to zoom in on particular nodes.

```
pdeplot(model,"NodeLabels","on")
xlim([-0.2,0.2])
ylim([-0.2,0.2])
```



Display the element labels.

```
pdeplot(model,"ElementLabels","on")
xlim([-0.2,0.2])
ylim([-0.2,0.2])
```



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 the zero Dirichlet boundary conditions on all edges.

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

Specify the coefficients and solve the PDE.

```
specifyCoefficients(model,"m",0, ...
                          "d",0, ...
                          "c",1, ...
                          "a",0, ...
                          "f",1);
results = solvepde(model)
```

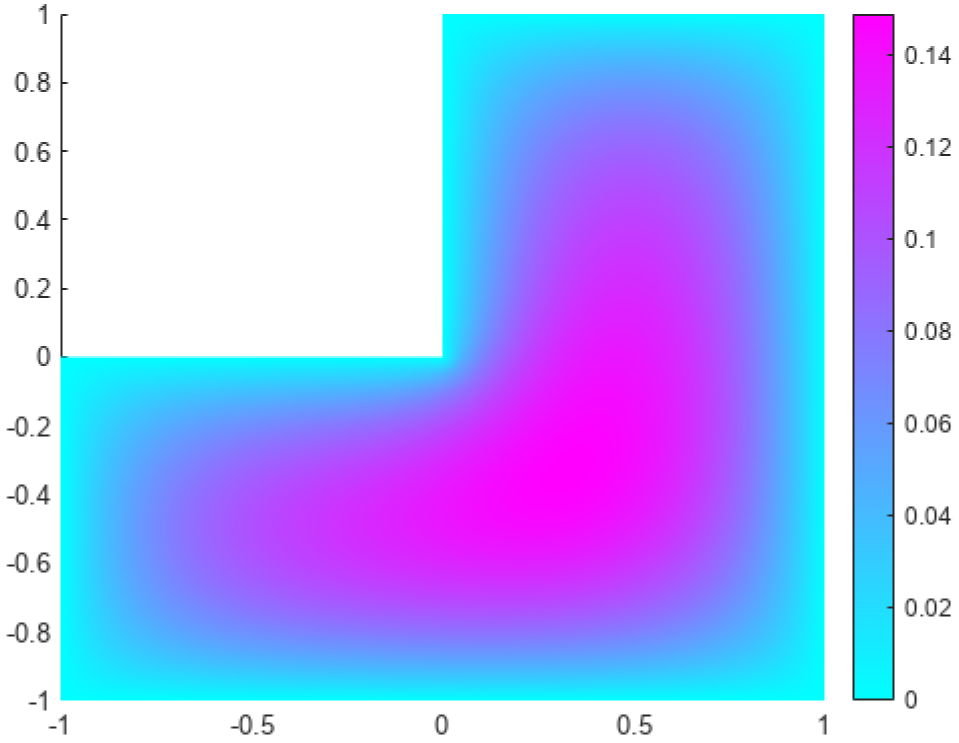
```
results = 
  StationaryResults with properties:

    NodalSolution: [1177x1 double]
       XGradients: [1177x1 double]
       YGradients: [1177x1 double]
       ZGradients: []
             Mesh: [1x1 FEMesh]
```

Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

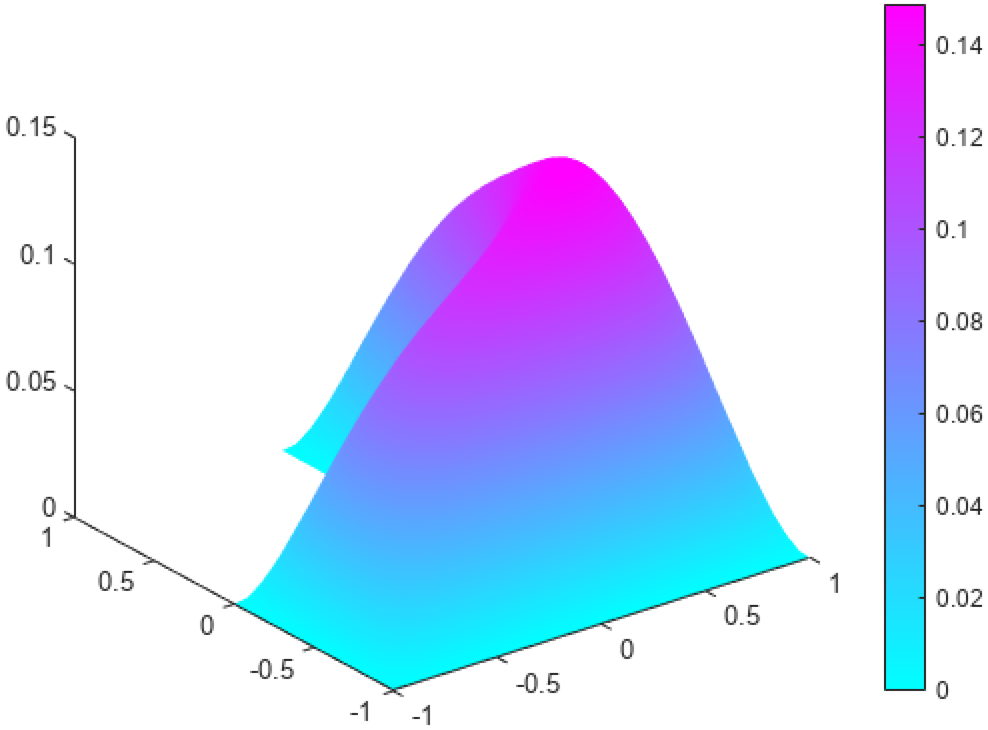
Plot the 2-D solution.

```
pdeplot(model,"XYData",u)
```



Plot the 3-D solution.

```
pdeplot(model,"XYData",u,"ZData",u)
```



Solution Quiver Plot

Plot the gradient of a PDE solution as a quiver plot.

Create a PDE model. Include the geometry of the built-in function `lshapeg`. Mesh the geometry.

```
model = createpde;
geometryFromEdges(model,@lshapeg);
generateMesh(model);
```

Set the 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)
```

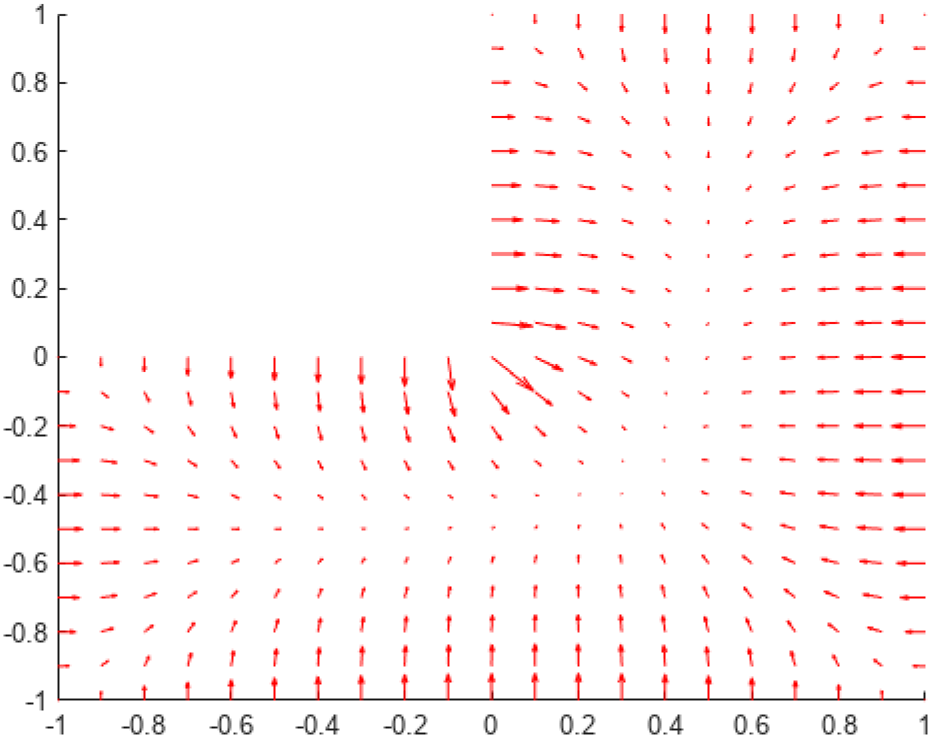
```
results = 
  StationaryResults with properties:

    NodalSolution: [1177x1 double]
       XGradients: [1177x1 double]
       YGradients: [1177x1 double]
       ZGradients: []
             Mesh: [1x1 FEMesh]
```

Access the gradient of the solution at the nodal locations.

```
ux = results.XGradients;
uy = results.YGradients;
```

Plot the gradient as a quiver plot.

```
pdeplot(model,"FlowData",[ux,uy])
```



Composite Plot

Plot the solution of a 2-D PDE in 3-D with the `"jet"` coloring and a mesh, and include a quiver plot. Get handles to the axes 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)
```

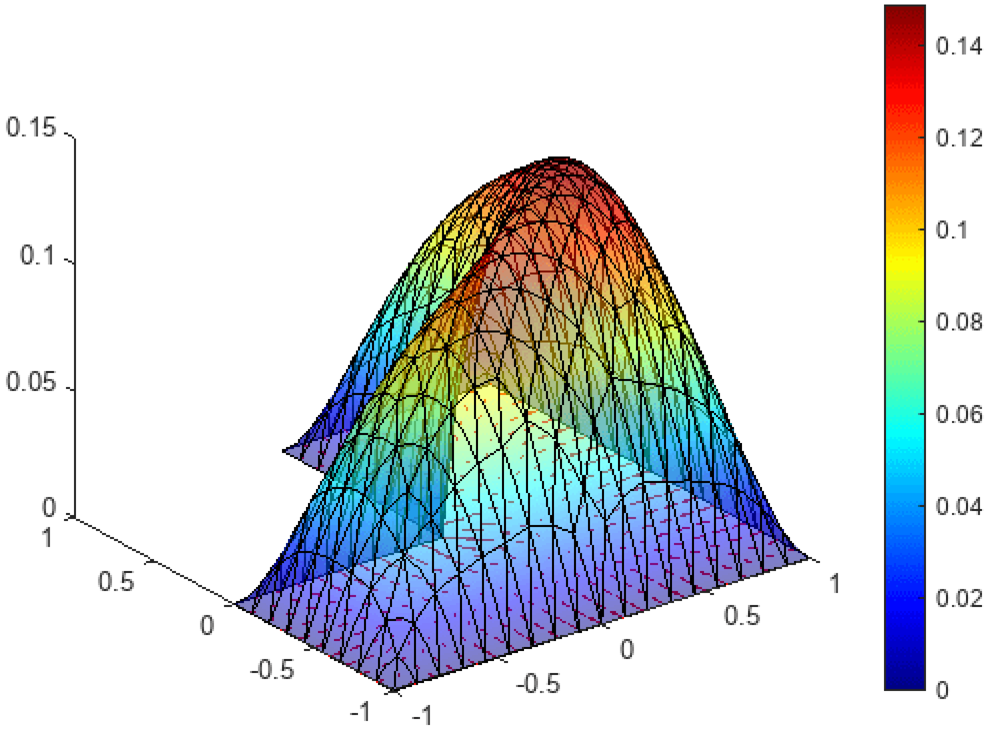
```
results = 
  StationaryResults with properties:

    NodalSolution: [1177x1 double]
       XGradients: [1177x1 double]
       YGradients: [1177x1 double]
       ZGradients: []
             Mesh: [1x1 FEMesh]
```

Access the solution and its gradient at the nodal locations.

```
u = results.NodalSolution;
ux = results.XGradients;
uy = results.YGradients;
```

Plot the solution in 3-D with the `"jet"` coloring and a mesh, and include the gradient as a quiver plot.

```
h = pdeplot(model,"XYData",u,"ZData",u, ...
                  "FaceAlpha",0.5, ...
                  "FlowData",[ux,uy], ...
                  "ColorMap","jet", ...
                  "Mesh","on");
```



Solution to Transient Thermal Model

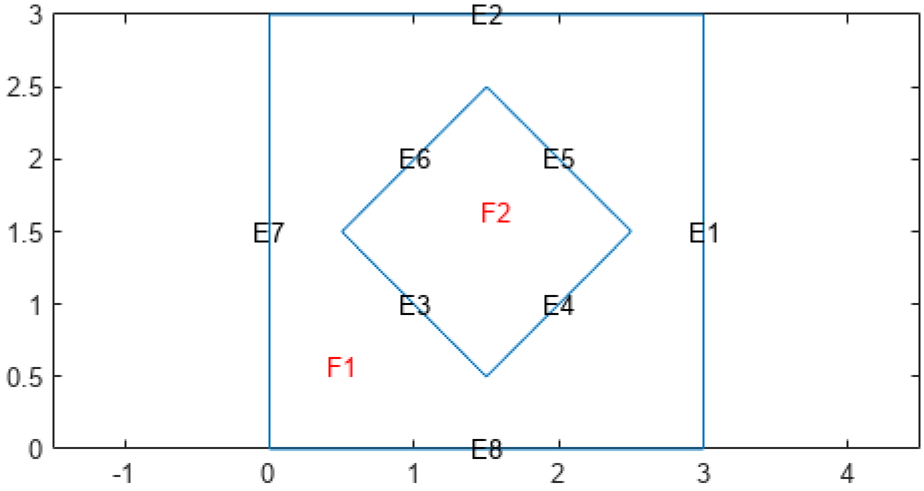
Solve a 2-D transient thermal problem.

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 these 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 these 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 a density of 4 W/m^2.

```
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);
```

Generate the mesh.

```
generateMesh(thermalmodel);
```

The dynamics for this problem are very fast. The temperature reaches a steady state in about 0.1 second. To capture the most active part of the dynamics, set the solution time to `logspace(-2,-1,10)`. This command returns 10 logarithmically spaced solution times between 0.01 and 0.1.

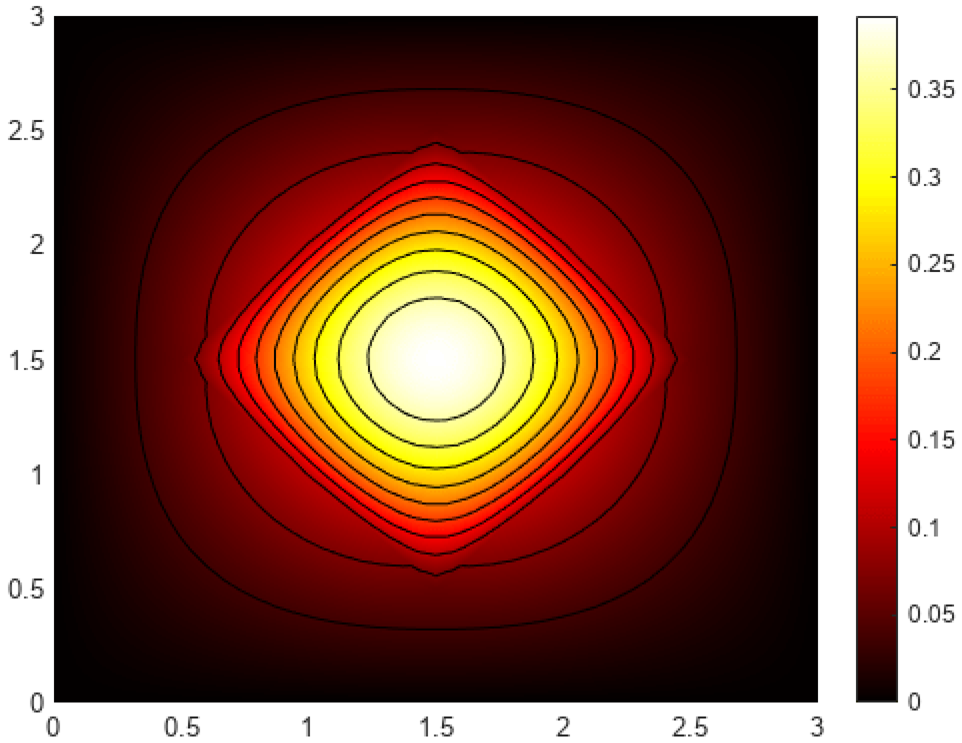
```
tlist = logspace(-2,-1,10);
```

Solve the equation.

```
thermalresults = solve(thermalmodel,tlist);
```

Plot the solution with isothermal lines by using a contour plot.

```
T = thermalresults.Temperature;
pdeplot(thermalmodel,"XYData",T(:,10),"Contour","on","ColorMap","hot")
```



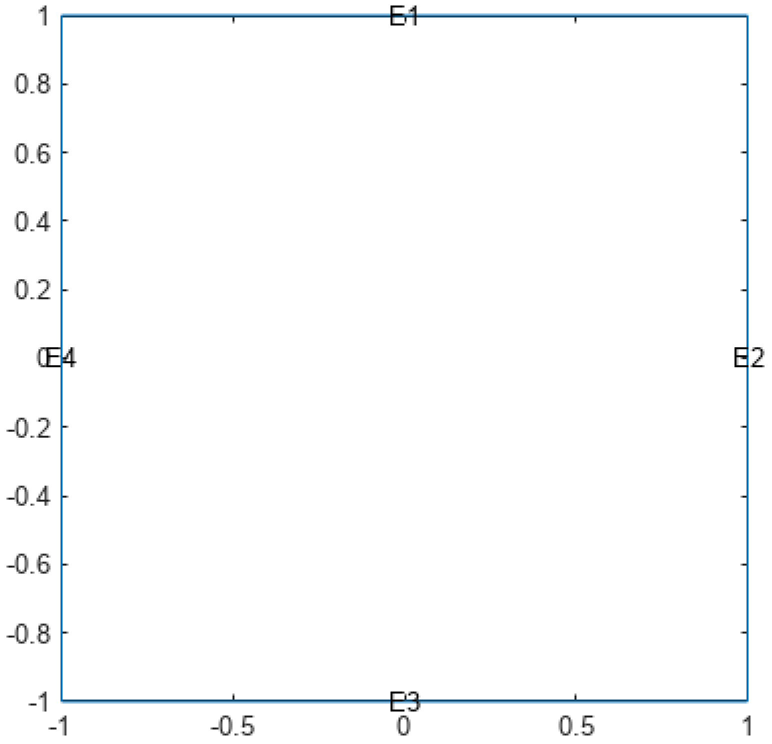
Plot Deformed Shape for Static Plane-Strain Problem

Create a structural analysis model for a static plane-strain problem.

```
structuralmodel = createpde("structural","static-planestrain");
```

Create the geometry and include it in the model. Plot the geometry.

```
geometryFromEdges(structuralmodel,@squareg);
pdegplot(structuralmodel,"EdgeLabels","on")
axis equal
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"PoissonsRatio",0.3, ...
                                     "YoungsModulus",210E3);
```

Specify the *x*-component of the enforced displacement for edge 1.

```
structuralBC(structuralmodel,"XDisplacement",0.001,"Edge",1);
```

Specify that edge 3 is a fixed boundary.

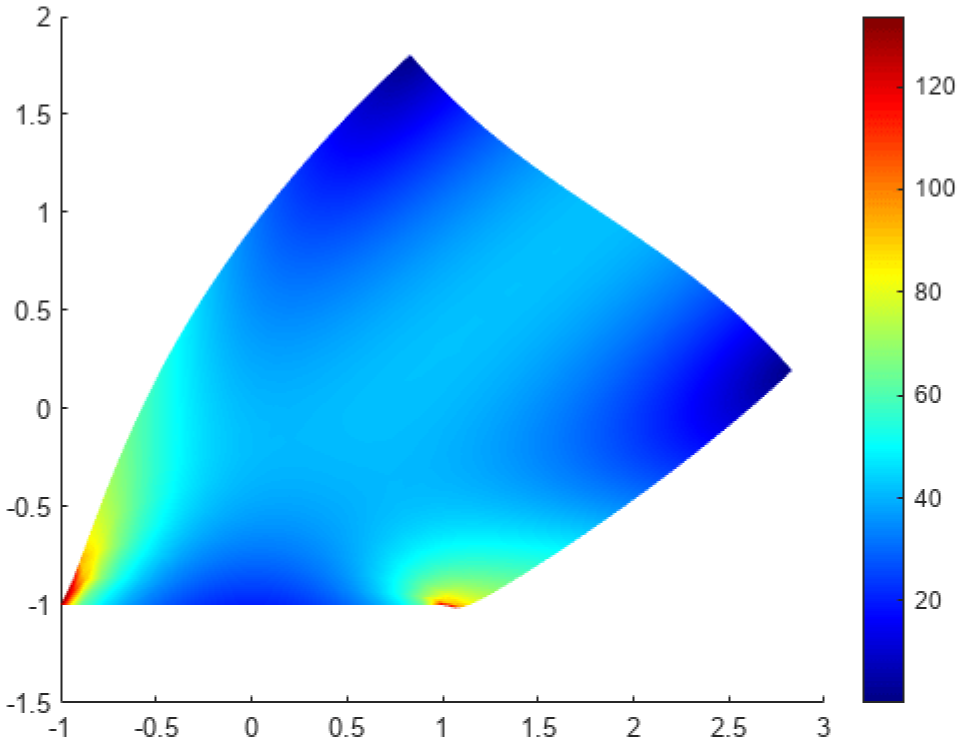
```
structuralBC(structuralmodel,"Constraint","fixed","Edge",3);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel);
```

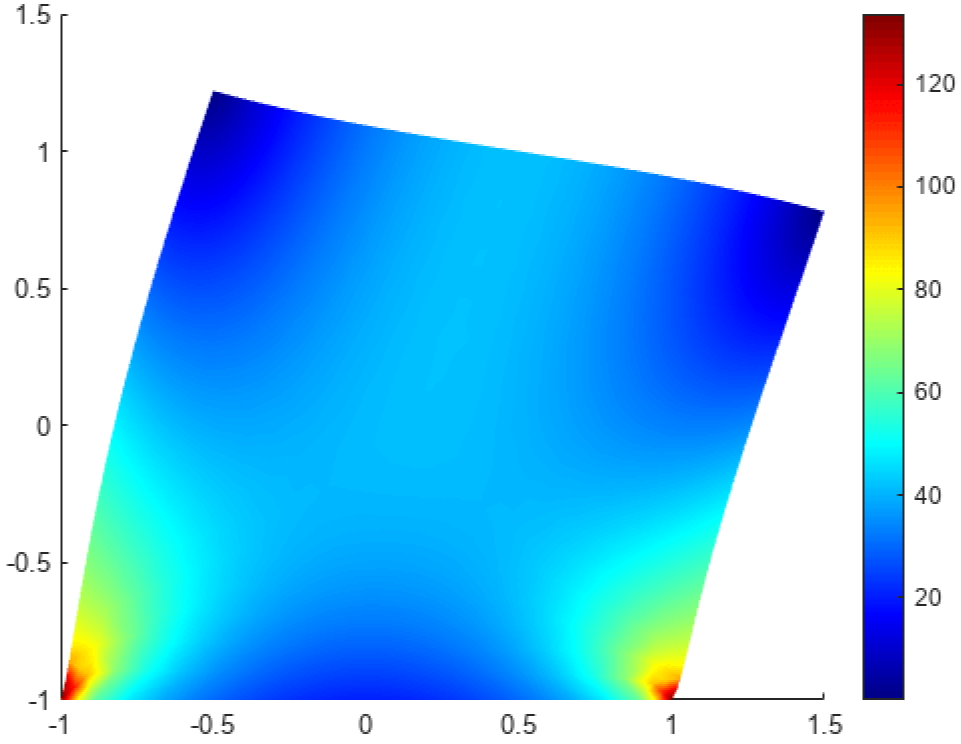
Plot the deformed shape using the default scale factor. By default, `pdeplot` internally determines the scale factor based on the dimensions of the geometry and the magnitude of deformation.

```
pdeplot(structuralmodel, ...
        "XYData",structuralresults.VonMisesStress, ...
        "Deformation",structuralresults.Displacement, ...
        "ColorMap","jet")
```



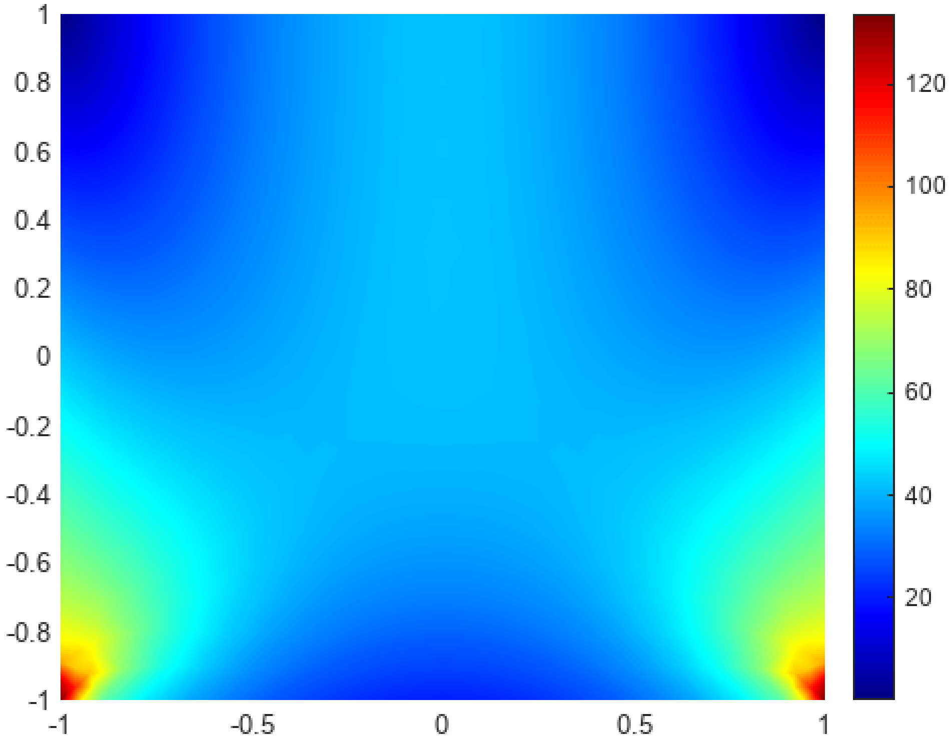
Plot the deformed shape with the scale factor 500.

```
pdeplot(structuralmodel, ...
        "XYData",structuralresults.VonMisesStress, ...
        "Deformation",structuralresults.Displacement, ...
        "DeformationScaleFactor",500,...
        "ColorMap","jet")
```



Plot the deformed shape without scaling.

```
pdeplot(structuralmodel,"XYData",structuralresults.VonMisesStress, ...
                        "ColorMap","jet")
```



Solution to Modal Analysis Structural Model

Find the fundamental (lowest) mode of a 2-D cantilevered beam, assuming prevalence of the plane-stress condition.

Specify geometric and structural properties of the beam, along with a unit plane-stress thickness.

```
length = 5;
height = 0.1;
E = 3E7;
nu = 0.3;
rho = 0.3/386;
```

Create a modal plane-stress model, assign a geometry, and generate a mesh.

```
structuralmodel = createpde("structural","modal-planestress");
gdm = [3;4;0;length;length;0;0;0;height;height];
g = decsg(gdm,'S1',('S1')');
geometryFromEdges(structuralmodel,g);
```

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

```
hmax = height/5;
msh=generateMesh(structuralmodel,"Hmax",hmax);
```

Specify the structural properties and boundary constraints.

```
structuralProperties(structuralmodel,"YoungsModulus",E, ...
                                     "MassDensity",rho, ...
                                     "PoissonsRatio",nu);
structuralBC(structuralmodel,"Edge",4,"Constraint","fixed");
```

Compute the analytical fundamental frequency (Hz) using the beam theory.

```
I = height^3/12;
analyticalOmega1 = 3.516*sqrt(E*I/(length^4*(rho*height)))/(2*pi)
```

```
analyticalOmega1 = 126.9498
```

Specify a frequency range that includes an analytically computed frequency and solve the model.

```
modalresults = solve(structuralmodel,"FrequencyRange",[0,1e6])
```

```
modalresults = 
  ModalStructuralResults with properties:

    NaturalFrequencies: [32x1 double]
            ModeShapes: [1x1 FEStruct]
                  Mesh: [1x1 FEMesh]
```

The solver finds natural frequencies and modal displacement values at nodal locations. To access these values, use `modalresults.NaturalFrequencies` and `modalresults.ModeShapes`.

```
modalresults.NaturalFrequencies/(2*pi)
```

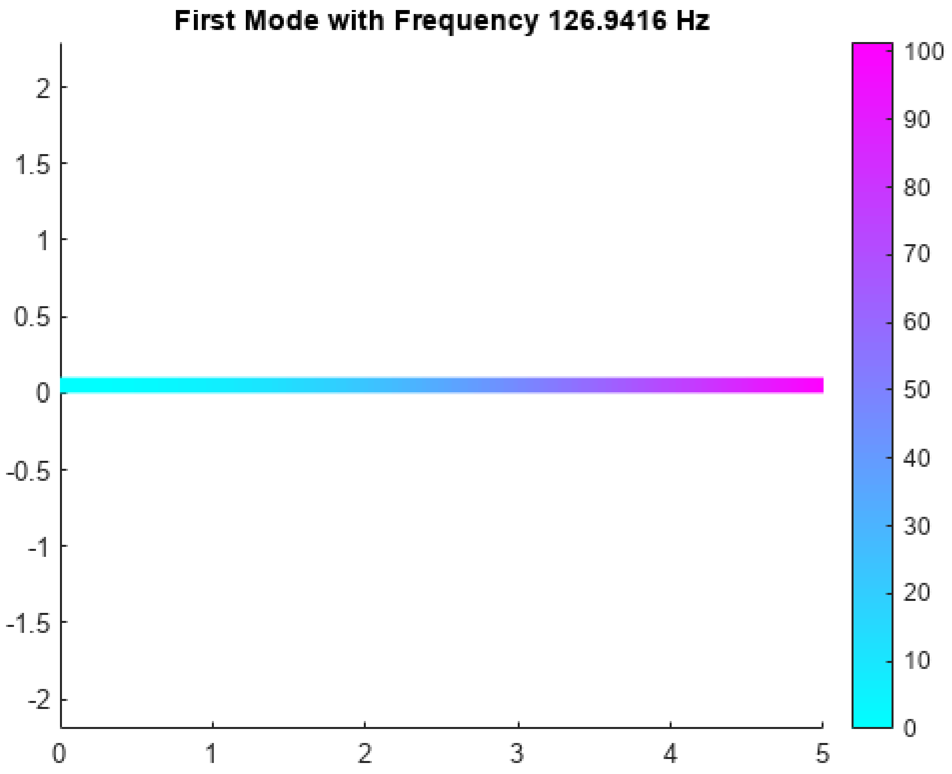
```
ans = 32×1
10^5 ×

    0.0013
    0.0079
    0.0222
    0.0433
    0.0711
    0.0983
    0.1055
    0.1462
    0.1930
    0.2455
      ⋮
```

```
modalresults.ModeShapes
```

```
ans = 
  FEStruct with properties:

           ux: [6511x32 double]
           uy: [6511x32 double]
    Magnitude: [6511x32 double]
```

Plot the *y*-component of the solution for the fundamental frequency.

```
pdeplot(structuralmodel,"XYData",modalresults.ModeShapes.uy(:,1))
title(['First Mode with Frequency ', ...
        num2str(modalresults.NaturalFrequencies(1)/(2*pi)),' Hz'])
axis equal
```



Solution to 2-D Electrostatic Analysis Model

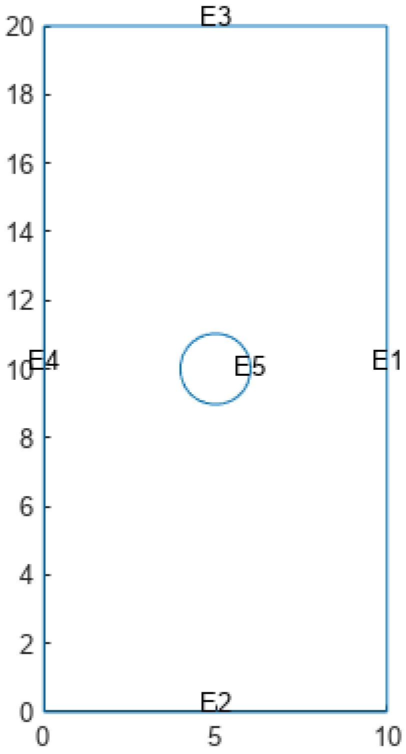
Solve an electromagnetic problem and find the electric potential and field distribution for a 2-D geometry representing a plate with a hole.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(emagmodel,"EdgeLabels","on")
```



Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Apply the voltage boundary conditions on the edges framing the rectangle and the circle.

```
electromagneticBC(emagmodel,"Voltage",0,"Edge",1:4);
electromagneticBC(emagmodel,"Voltage",1000,"Edge",5);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

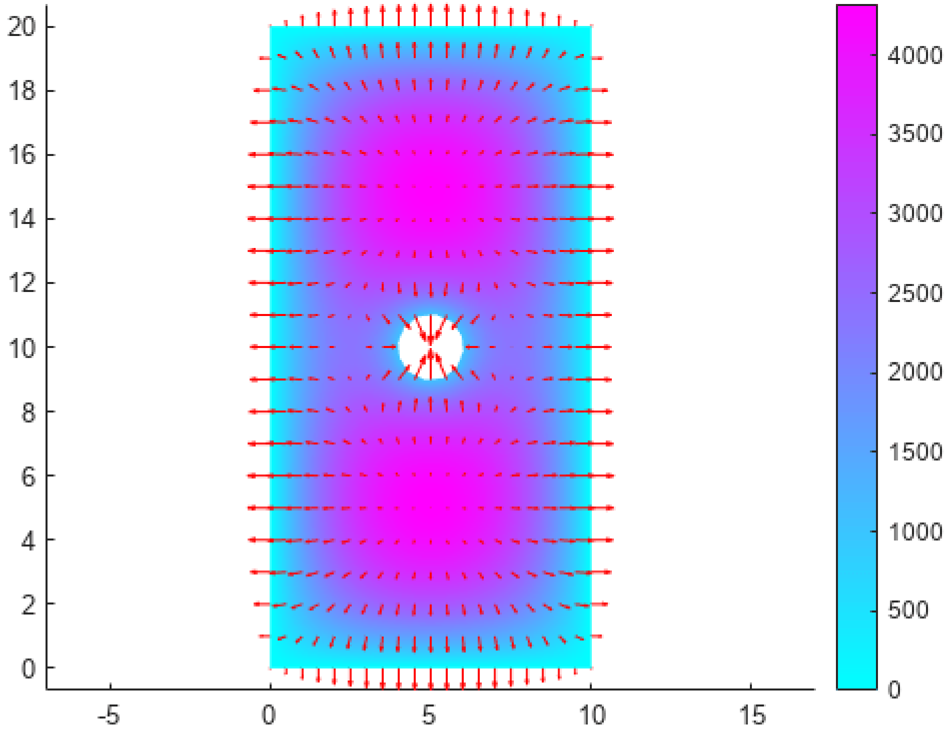
```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:
```

```
     ElectricPotential: [1218x1 double]
         ElectricField: [1x1 FEStruct]
   ElectricFluxDensity: [1x1 FEStruct]
                  Mesh: [1x1 FEMesh]
```

Plot the electric potential and field.

```
pdeplot(emagmodel,"XYData",R.ElectricPotential, ...
                  "FlowData",[R.ElectricField.Ex ...
                              R.ElectricField.Ey])
axis equal
```



[p,e,t] Mesh and Solution Plots

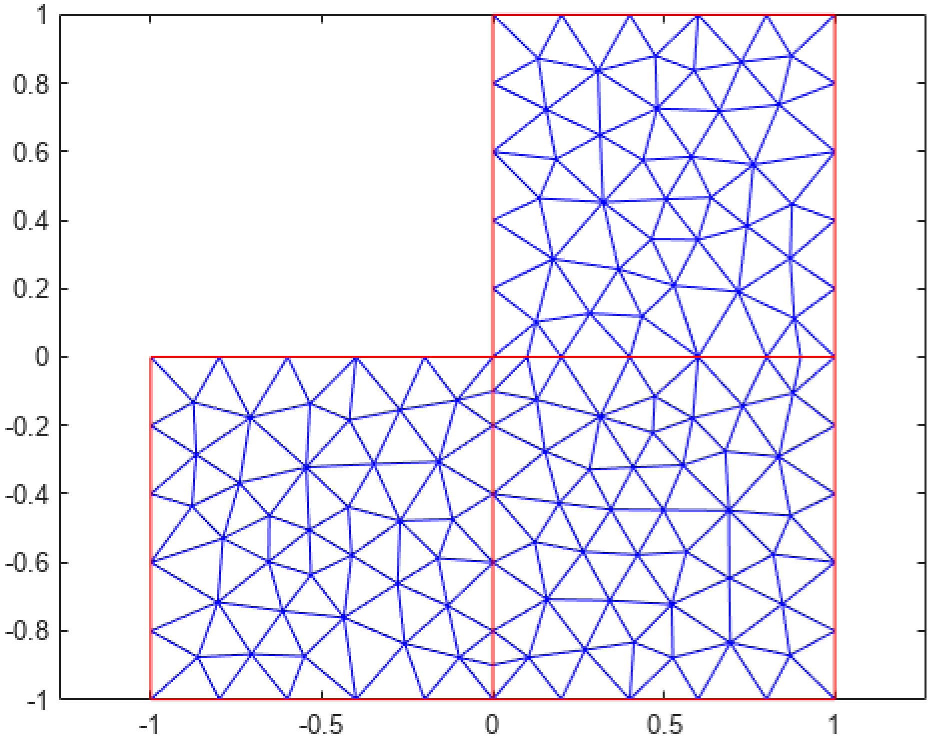
Plot the `p,e,t` mesh. Display the solution using 2-D and 3-D colored plots.

Create the geometry, mesh, boundary conditions, PDE coefficients, and solution.

```
[p,e,t] = initmesh('lshapeg');
u = assempde("lshapeb",p,e,t,1,0,1);
```

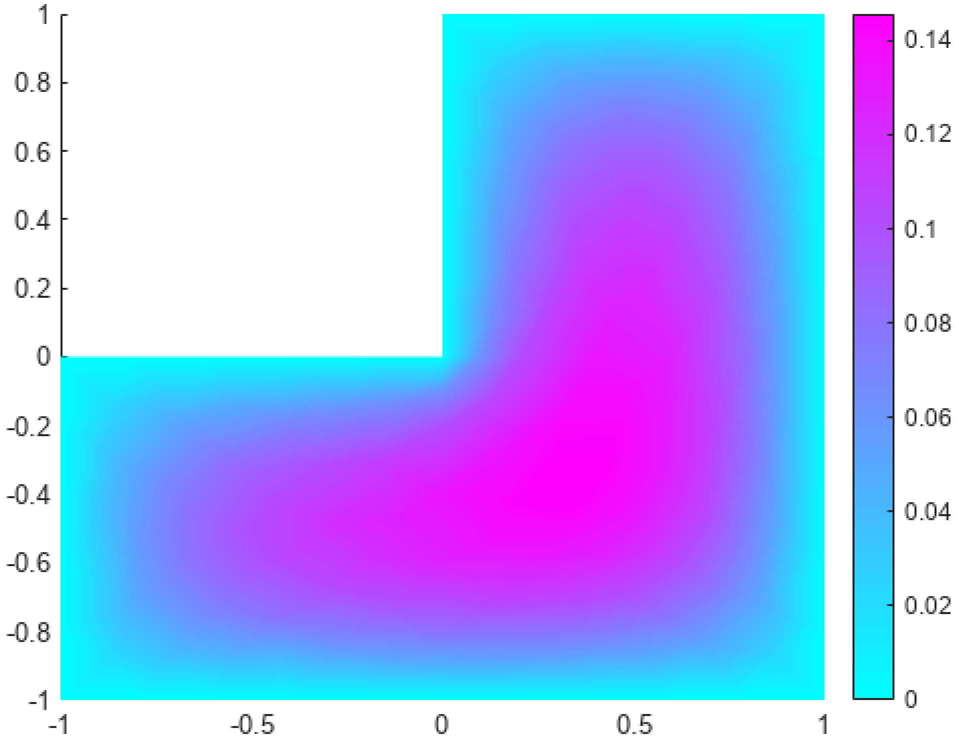
Plot the mesh.

```
pdeplot(p,e,t)
```



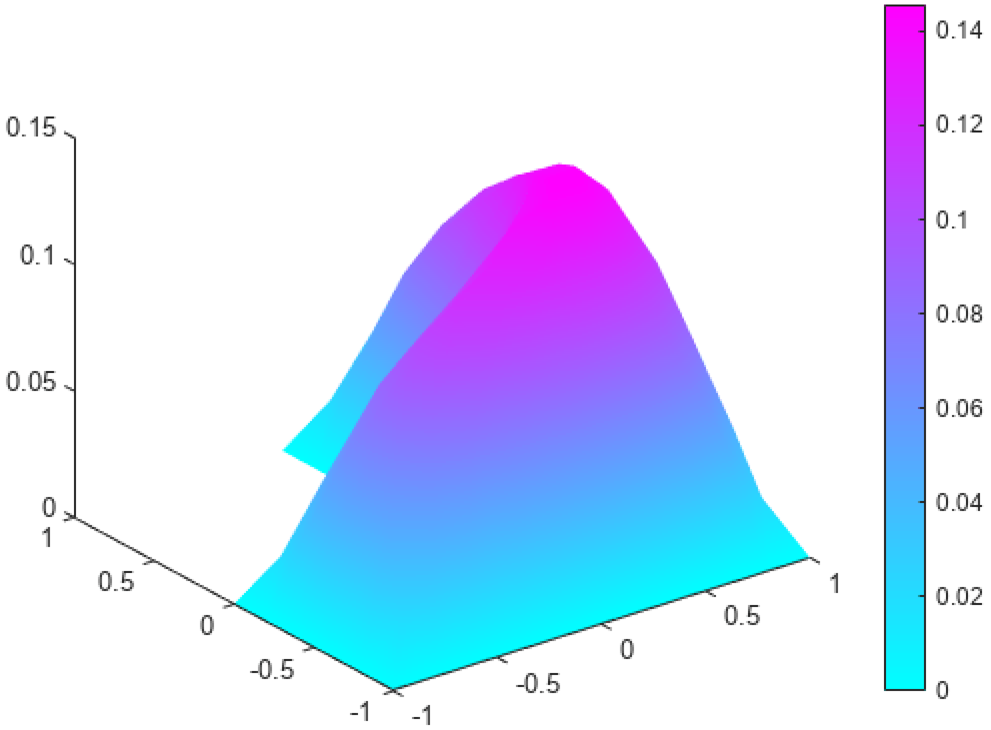
Plot the solution as a 2-D colored plot.

```
pdeplot(p,e,t,"XYData",u)
```



Plot the solution as a 3-D colored plot.

```
pdeplot(p,e,t,"XYData",u,"ZData",u)
```



Input Arguments

`model` — Model object
`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectromagneticModel` object

Model object, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectromagneticModel` object.

Example: `model = createpde(1)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","magnetostatic")`

`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`

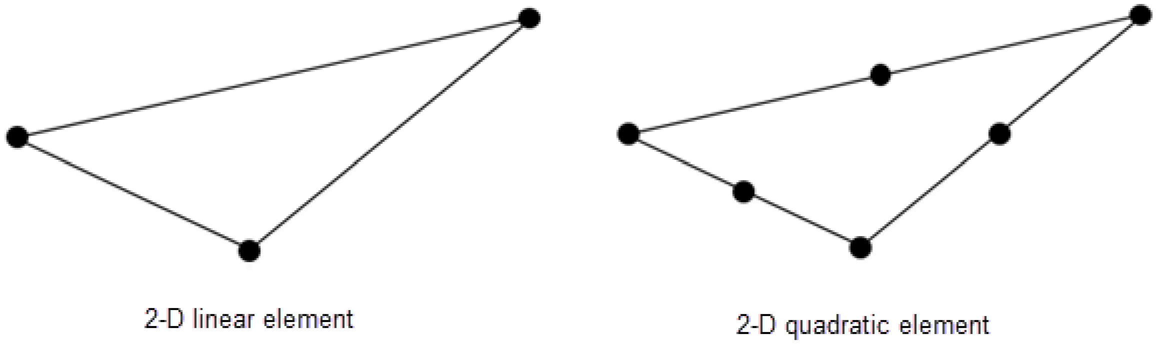
`nodes` — Nodal coordinates
2-by-*NumNodes* matrix

Nodal coordinates, specified as a 2-by-*NumNodes* matrix. *NumNodes* is the number of nodes.

`elements` — Element connectivity matrix in terms of node IDs

3-by-*NumElements* matrix | 6-by-*NumElements* matrix

Element connectivity matrix in terms of the node IDs, specified as a 3-by-*NumElements* or 6-by-*NumElements* matrix. Linear meshes contain only corner nodes. For linear meshes, the connectivity matrix has three nodes per 2-D element. Quadratic meshes contain corner nodes and nodes in the middle of each edge of an element. For quadratic meshes, the connectivity matrix has six nodes per 2-D element.



`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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** 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 as [p,e,t] Triples” on page 2-172.

Typically, you use the `p`, `e`, and `t` data exported from the **PDE Modeler** app, or generated by `initmesh` or `refinemesh`.

Example: `[p,e,t] = initmesh(gd)`

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `pdeplot(model,"XYData",u,"ZData",u)`

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 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.

Data Plots

`XYData` — Colored surface plot data

vector

Colored surface plot data, specified as a vector. If you use a `[p,e,t]` representation, specify data for points in a vector of length `size(p,2)`, or specify data for triangles in a vector of length `size(t,2)`.

- Typically, you set `XYData` to the solution `u`. The `pdeplot` function 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 `k`th component of a solution to a PDE system, extract the relevant part of the solution. For example, when using a `PDEModel` object, specify:

```
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 using a `[p,e,t]` representation, specify:

```
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`

`XYStyle` — Coloring choice
`"interp"` (default) | `"off"` | `"flat"`

Coloring choice, specified as one of the following values:

- `"off"` — No shading, only mesh is displayed.
- `"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` | `string`

`ZData` — Data for 3-D plot heights
matrix

Data for the 3-D plot heights, specified as a matrix. If you use a `[p,e,t]` representation, provide 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 using a `PDEModel` object, specify:

```
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 using a `[p,e,t]` representation, specify:

```
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 these values:

- `"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.

If you use `ZStyle` without specifying the `ZData` name-value pair, then `pdeplot` ignores `ZStyle`.

Example: `"ZStyle","discontinuous"`

Data Types: `char` | `string`

`FlowData` — Data for quiver plot

matrix

Data for the quiver plot on page 5-958, specified as an `M`-by-`2` matrix, where `M` is the number of mesh nodes. `FlowData` contains the *x* and *y* values of the field at the mesh points.

When you use a `PDEModel` object, set `FlowData` as follows:

```
results = solvepde(model);
gradx = results.XGradients;
grady = results.YGradients;
pdeplot(model,"FlowData",[gradx grady])
```

When you use a `[p,e,t]` representation, set `FlowData` as follows:

```
[gradx,grady] = pdegrad(p,t,u); % Calculate gradient
pdeplot(p,e,t,"FlowData",[gradx;grady])
```

When you use `ZData` to represent a 2-D PDE solution as a 3-D plot and you also include a quiver 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 the quiver plot, specified as `"arrow"` or `"off"`. Here, `"arrow"` displays the quiver plot on page 5-958 specified by the `FlowData` name-value pair.

Example: `"FlowStyle","off"`

Data Types: `char` | `string`

`XYGrid` — Indicator to convert mesh data to *x*-*y* grid

`"off"` (default) | `"on"`

Indicator to convert the mesh data to *x*-*y* grid before plotting, specified as `"off"` or `"on"`.

Note This conversion can change the geometry and lessen the quality of the plot.

By default, the grid has about `sqrt(size(t,2))` elements in each direction.

Example: `"XYGrid","on"`

Data Types: `char` | `string`

`GridParam` — Customized *x*-*y* grid
`[tn;a2;a3]` from an earlier call to `tri2grid`

Customized *x*-*y* grid, 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`. The `tri2grid` function does not work with `PDEModel` objects.

Example: `"GridParam",[tn;a2;a3]`

Data Types: `double`

Mesh Plots

`NodeLabels` — Node labels
`"off"` (default) | `"on"`

Node labels, specified as `"off"` or `"on"`.

`pdeplot` ignores `NodeLabels` when you use it with `ZData`.

Example: `"NodeLabels","on"`

Data Types: `char` | `string`

`ElementLabels` — Element labels
`"off"` (default) | `"on"`

Element labels, specified as `"off"` or `"on"`.

`pdeplot` ignores `ElementLabels` when you use it with `ZData`.

Example: `"ElementLabels","on"`

Data Types: `char` | `string`

Structural Analysis Plots

`Deformation` — Data for plotting deformed shape
`Displacement` property of `StaticStructuralResults` object

Data for plotting the deformed shape for a structural analysis model, specified as the `Displacement` property of the `StaticStructuralResults` object.

In an undeformed shape, center nodes in quadratic meshes are always added at half-distance between corners. When you plot a deformed shape, the center nodes might move away from the edge centers.

Example: `"Deformation",structuralresults.Displacement`

`DeformationScaleFactor` — Scaling factor for plotting deformed shape
real number

Scaling factor for plotting the deformed shape, specified as a real number. Use this argument with the `Deformation` name-value pair. The default value is defined internally, based on the dimensions of the geometry and the magnitude of the deformation.

Example: `"DeformationScaleFactor",100`

Data Types: `double`

Annotations and Appearance

`ColorBar` — Indicator to include color bar

`"on"` (default) | `"off"`

Indicator to include a color bar, specified as `"on"` or `"off"`. Specify `"on"` to display a bar giving the numeric values of colors in the plot. For details, see `colorbar`. The `pdeplot` function uses the colormap specified in the `ColorMap` name-value pair.

Example: `"ColorBar","off"`

Data Types: `char` | `string`

`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` must be used with the `XYData` name-value pair.

Example: `"ColorMap","jet"`

Data Types: `double` | `char` | `string`

`Mesh` — Indicator to show mesh

`"off"` (default) | `"on"`

Indicator to show the mesh, specified as `"on"` or `"off"`. Specify `"on"` to show the mesh in the plot.

Example: `"Mesh","on"`

Data Types: `char` | `string`

`Title` — Title of plot

character vector

Title of plot, specified as a character vector or a string.

Example: `"Title","Solution Plot"`

Data Types: `char` | `string`

`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`

`Contour` — Indicator to plot level curves

`"off"` (default) | `"on"`

Indicator to plot level curves, specified as `"off"` or `"on"`. Specify `"on"` to plot level curves for the `XYData` data. Specify the levels with the `Levels` name-value pair.

Example: `"Contour","on"`

Data Types: `char` | `string`

`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 of level values.

- Positive integer — Plot `Levels` as 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`

Output Arguments

`h` — Handles to graphics objects
vector

Handles to graphics objects, returned as a vector.

More About

Quiver Plot

A 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`.

Version History

Introduced before R2006a

Electromagnetic Analysis

You can now plot electromagnetic results, such as electric and magnetic potentials, fields, and fluxes.

Improved performance for plots with many text labels

Performance change in R2020a

`pdeplot` shows faster rendering and better responsiveness for plots that display many text labels. Code containing `findobj(fig,'Type','Text')` no longer returns labels on figures produced by `pdeplot`.

Highlighting particular nodes and elements on mesh plots

`pdeplot` accepts node and element IDs as input arguments, letting you highlight particular nodes and elements on mesh plots.

Structural Analysis

You can now plot structural results, such as displacements, stresses, and strains.

Thermal Analysis

You can now plot thermal results, such as temperatures and temperature gradients.

Transparency, node and element labels

You can now set plot transparency by using `FaceAlpha`, and display node and element labels by using `NodeLabels` and `ElementLabels`, respectively.

See Also

`pdegplot` | `pdemesh` | `pdeplot3D` | `PDEModel`

Topics

“Solution and Gradient Plots with pdeplot and pdeplot3D” on page 3-329
“Deflection of Piezoelectric Actuator” on page 3-11
“Mesh Data” on page 2-175
“Solve Problems Using PDEModel Objects” on page 2-2

pdeplot3D

Plot solution or surface mesh for 3-D problem

Syntax

```
pdeplot3D(model,"ColorMapData",results.NodalSolution)
pdeplot3D(model,"ColorMapData",results.Temperature)
pdeplot3D(
model,"ColorMapData",results.VonMisesStress,"Deformation",results.Displacemen
t)
pdeplot3D(model,"ColorMapData",results.ElectricPotential)
```

```
pdeplot3D(model)
pdeplot3D(mesh)
pdeplot3D(nodes,elements)
```

```
pdeplot3D( ___ ,Name,Value)
h = pdeplot3D( ___ )
```

Description

`pdeplot3D(model,"ColorMapData",results.NodalSolution)` plots the solution at nodal locations as colors on the surface of the 3-D geometry specified in `model`.

`pdeplot3D(model,"ColorMapData",results.Temperature)` plots the temperature at nodal locations for a 3-D thermal analysis model.

`pdeplot3D( model,"ColorMapData",results.VonMisesStress,"Deformation",results.Displacement)` plots the von Mises stress and shows the deformed shape for a 3-D structural analysis model.

`pdeplot3D(model,"ColorMapData",results.ElectricPotential)` plots the electric potential at nodal locations for a 3-D electrostatic analysis model.

`pdeplot3D(model)` plots the surface mesh specified in `model`.

`pdeplot3D(mesh)` plots the mesh defined as a `Mesh` property of a 3-D `model` object of type `PDEModel`.

`pdeplot3D(nodes,elements)` plots the mesh defined by `nodes` and `elements`.

`pdeplot3D( ___ ,Name,Value)` plots the surface mesh, the data at nodal locations, or both the mesh and data, depending on the `Name,Value` pair arguments. Use any arguments from the previous syntaxes.

`h = pdeplot3D( ___ )` returns a handle to a plot, 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)
```

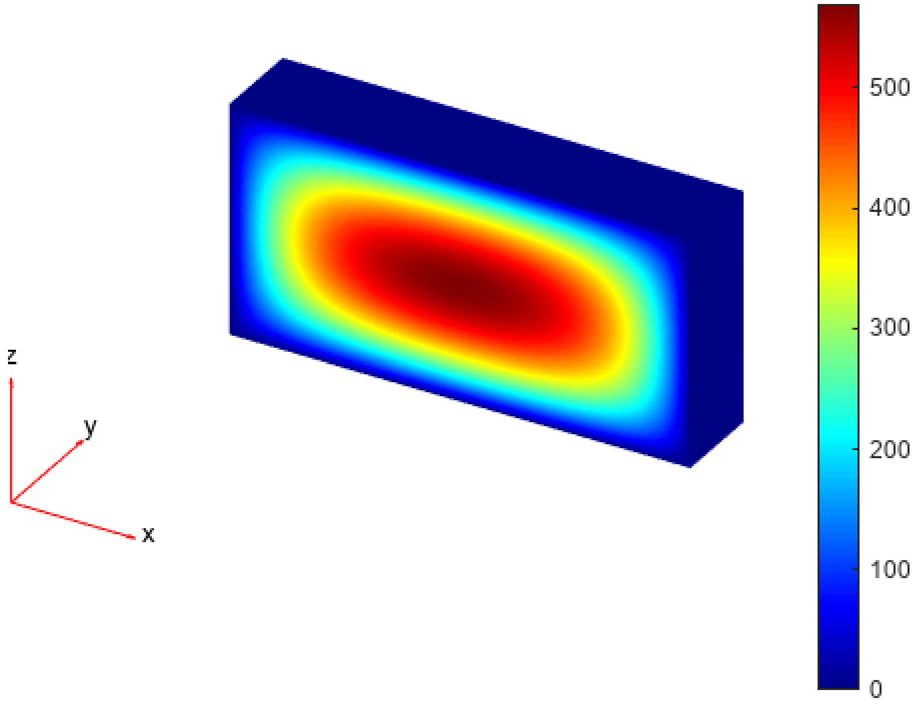
```
results = 
  StationaryResults with properties:

    NodalSolution: [12691x1 double]
       XGradients: [12691x1 double]
       YGradients: [12691x1 double]
       ZGradients: [12691x1 double]
             Mesh: [1x1 FEMesh]
```

Access the solution at the nodal locations.

```
u = results.NodalSolution;
```

Plot the solution `u` on the geometry surface.

```
pdeplot3D(model,"ColorMapData",u)
```



Solution to Steady-State Thermal Model

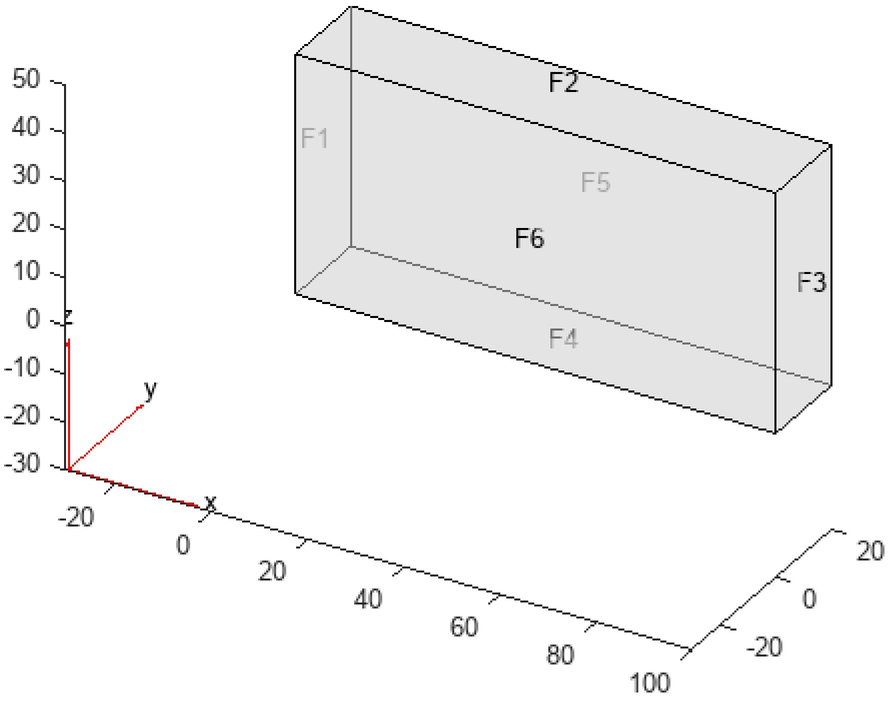
Solve a 3-D steady-state thermal problem.

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 °C to the left side of the block (face 1) and a constant temperature of 300 °C to 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);
```

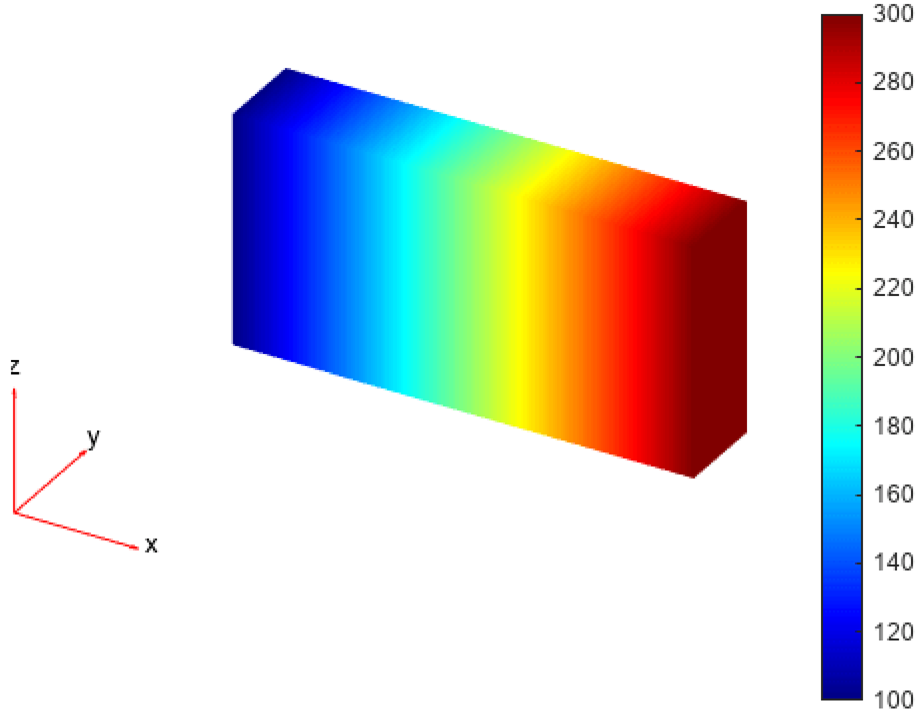
Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)
```

```
thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [12691x1 double]
     XGradients: [12691x1 double]
     YGradients: [12691x1 double]
     ZGradients: [12691x1 double]
           Mesh: [1x1 FEMesh]
```

The solver finds the 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 the nodal locations.

```
pdeplot3D(thermalmodel,"ColorMapData",thermalresults.Temperature)
```



Heat Flux for 3-D Steady-State Thermal Model

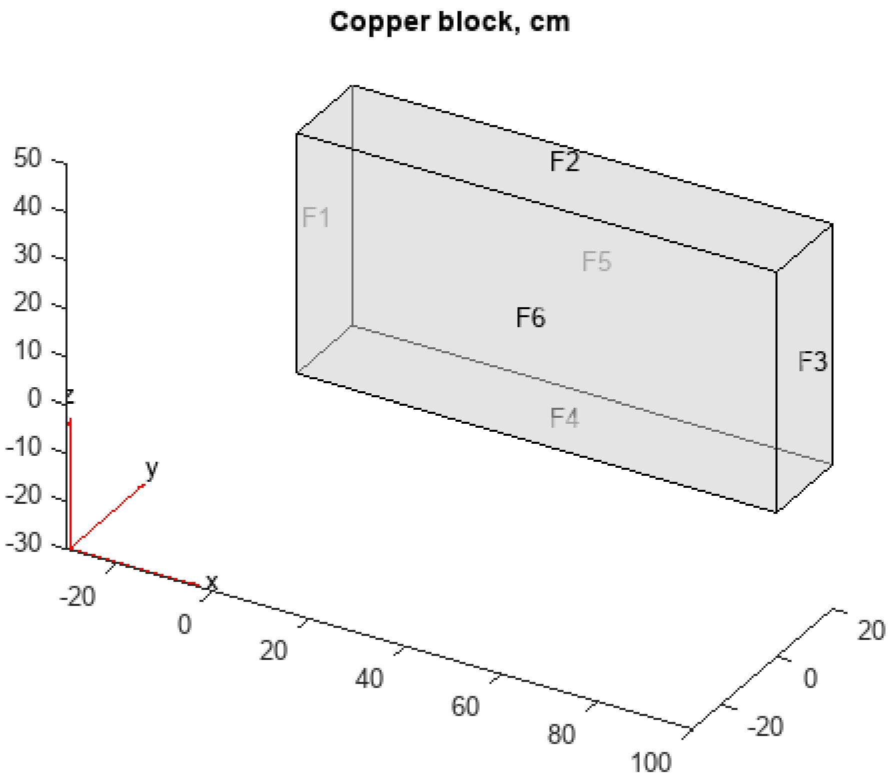
For a 3-D steady-state thermal model, evaluate heat flux at the nodal locations and at the points specified by `x`, `y`, and `z` coordinates.

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
```



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);
```

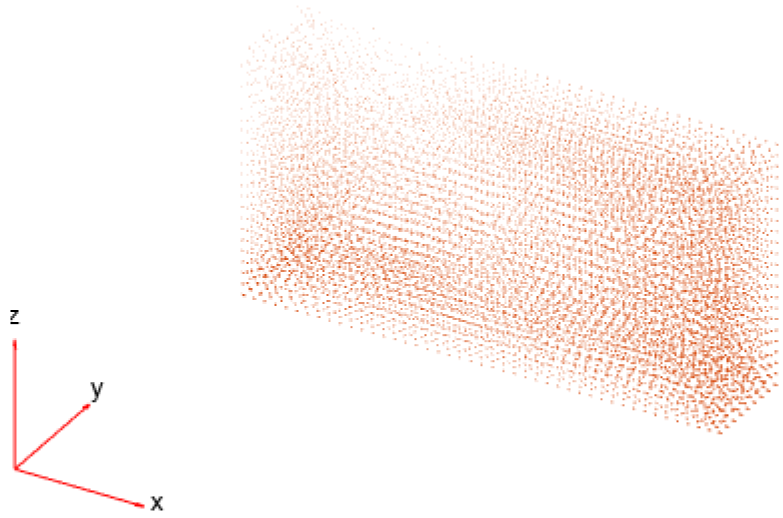
Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)
```

```
thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [12691x1 double]
     XGradients: [12691x1 double]
     YGradients: [12691x1 double]
     ZGradients: [12691x1 double]
           Mesh: [1x1 FEMesh]
```

Evaluate heat flux at the nodal locations.

```
[qx,qy,qz] = evaluateHeatFlux(thermalresults);

figure
pdeplot3D(thermalmodel,"FlowData",[qx qy qz])
```



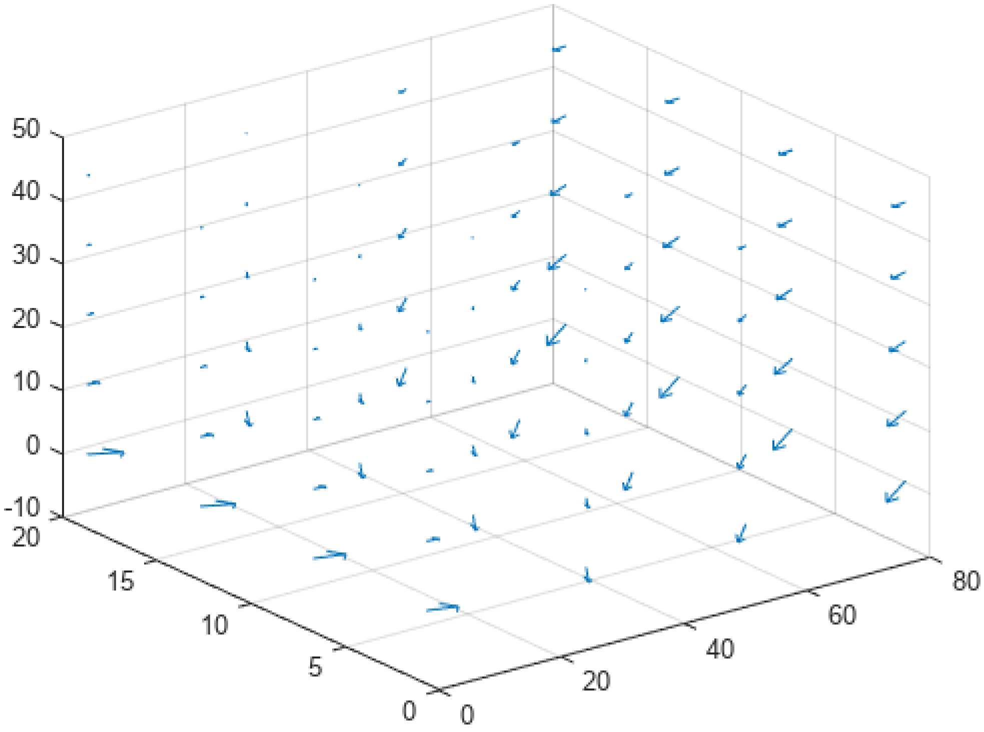
Create a grid specified by `x`, `y`, and `z` coordinates, and evaluate heat flux to the grid.

```
[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.

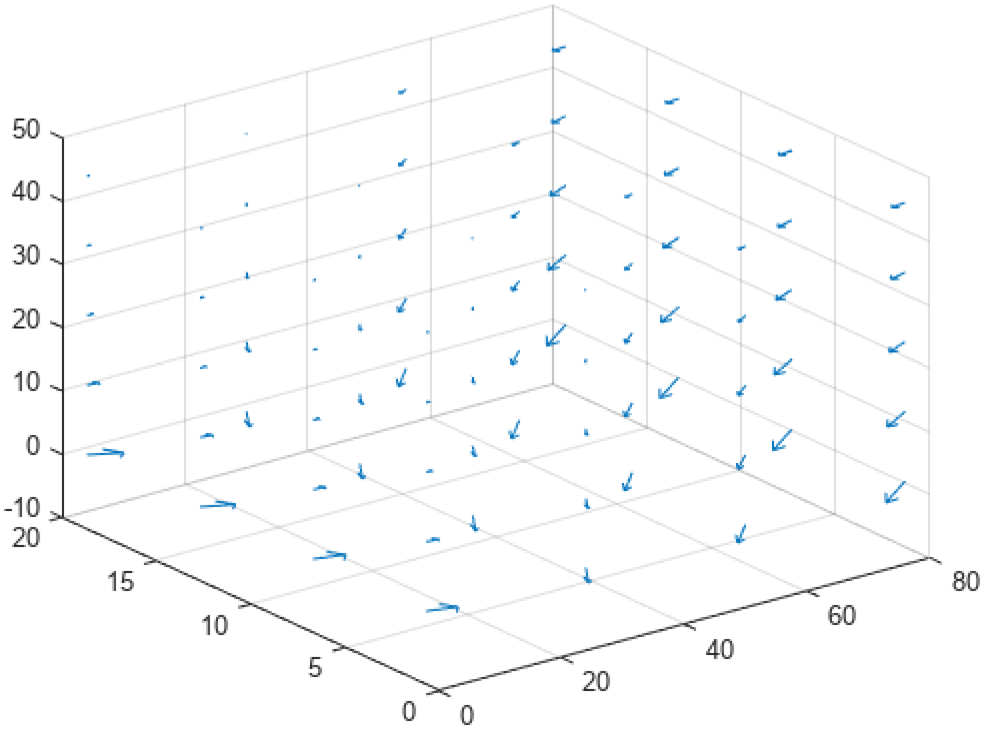
```
qx = reshape(qx,size(X));
qy = reshape(qy,size(Y));
qz = reshape(qz,size(Z));
```

```
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)
```



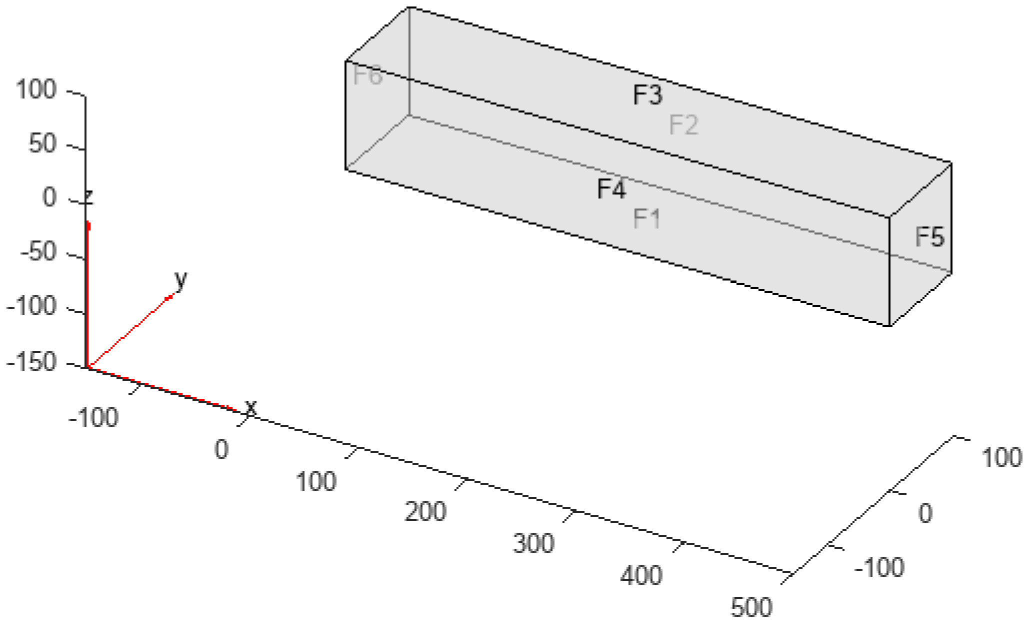
Deformed Shape for Cantilever Beam Problem

Create a structural analysis model for a 3-D problem.

```
structuralmodel = createpde("structural","static-solid");
```

Import the geometry and plot it.

```
importGeometry(structuralmodel,"SquareBeam.stl");
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"PoissonsRatio",0.3, ...
                                     "YoungsModulus",210E3);
```

Specify that face 6 is a fixed boundary.

```
structuralBC(structuralmodel,"Face",6,"Constraint","fixed");
```

Specify the surface traction for face 5.

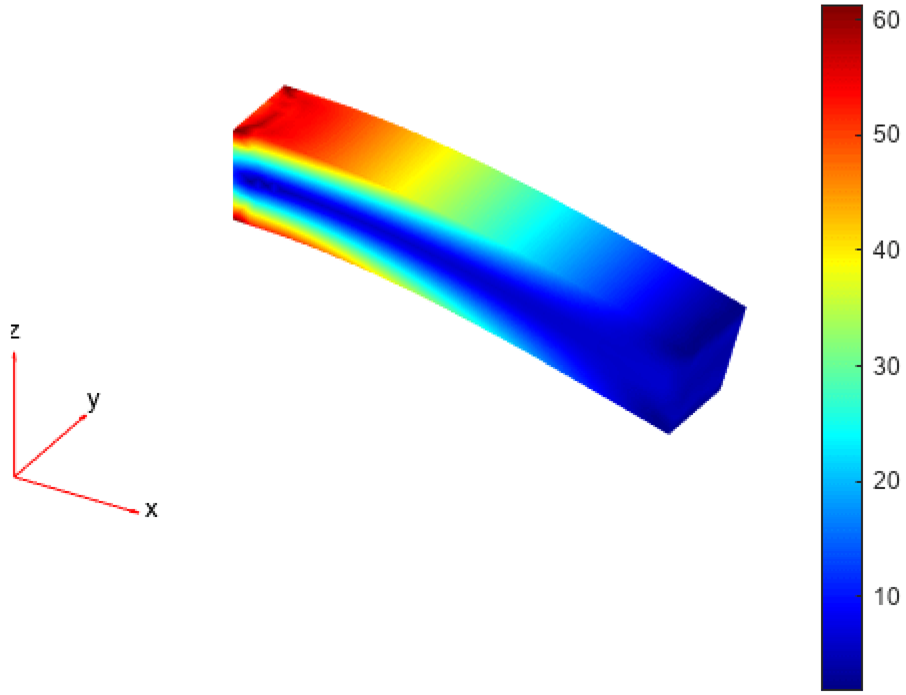
```
structuralBoundaryLoad(structuralmodel,"Face",5, ...
                                       "SurfaceTraction", ...
                                       [0;0;-2]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel);
```

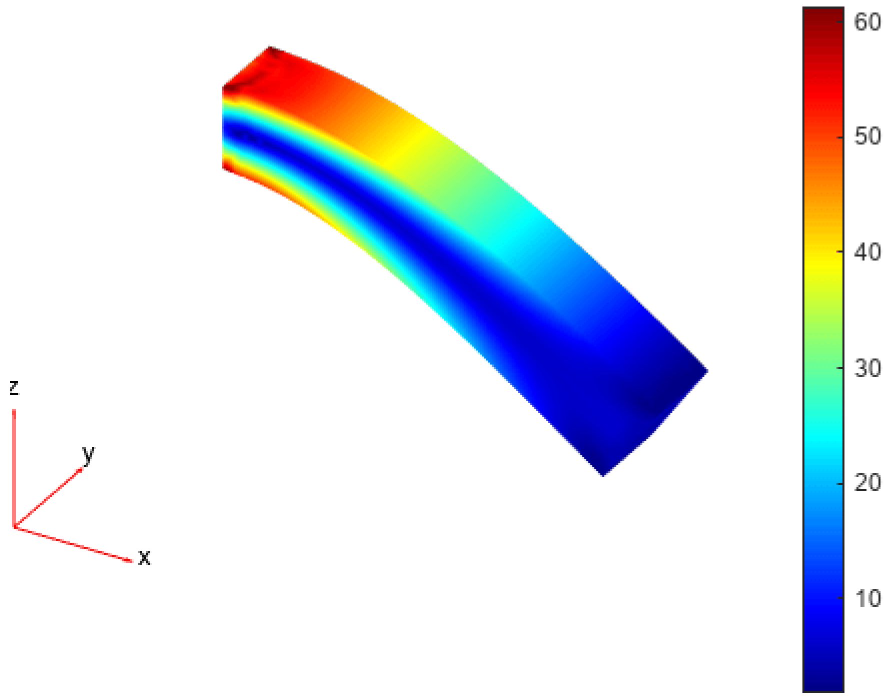
Plot the deformed shape with the von Mises stress using the default scale factor. By default, `pdeplot3D` internally determines the scale factor based on the dimensions of the geometry and the magnitude of deformation.

```
figure
pdeplot3D(structuralmodel, ...
          "ColorMapData",structuralresults.VonMisesStress, ...
          "Deformation",structuralresults.Displacement)
```



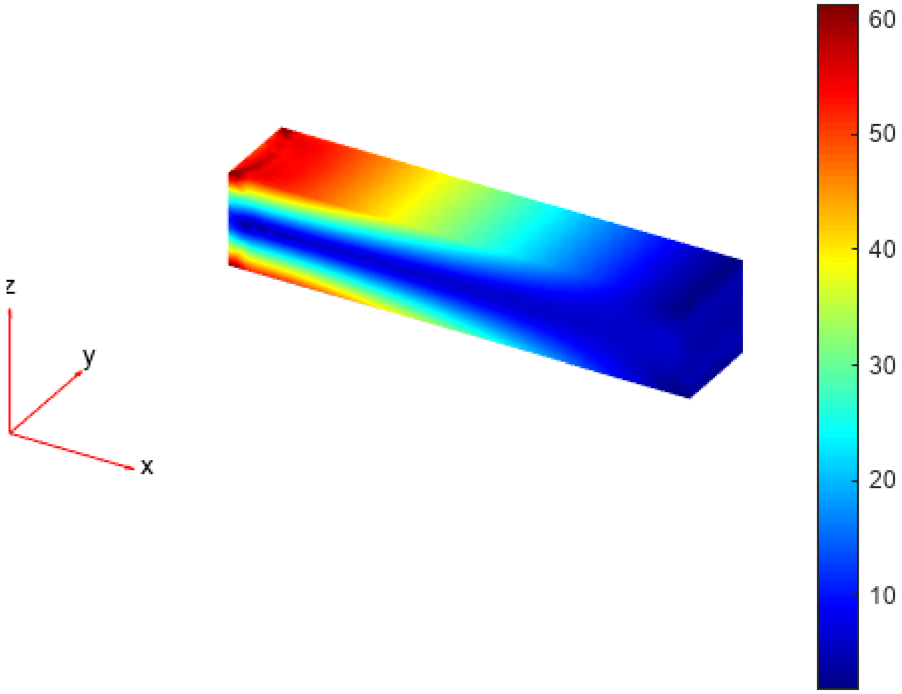
Plot the same results with the scale factor 500.

```
figure
pdeplot3D(structuralmodel, ...
          "ColorMapData",structuralresults.VonMisesStress, ...
          "Deformation",structuralresults.Displacement, ...
          "DeformationScaleFactor",500)
```



Plot the same results without scaling.

```
figure
pdeplot3D(structuralmodel, ...
          "ColorMapData",structuralresults.VonMisesStress)
```



von Mises Stress for 3-D Structural Dynamic Problem

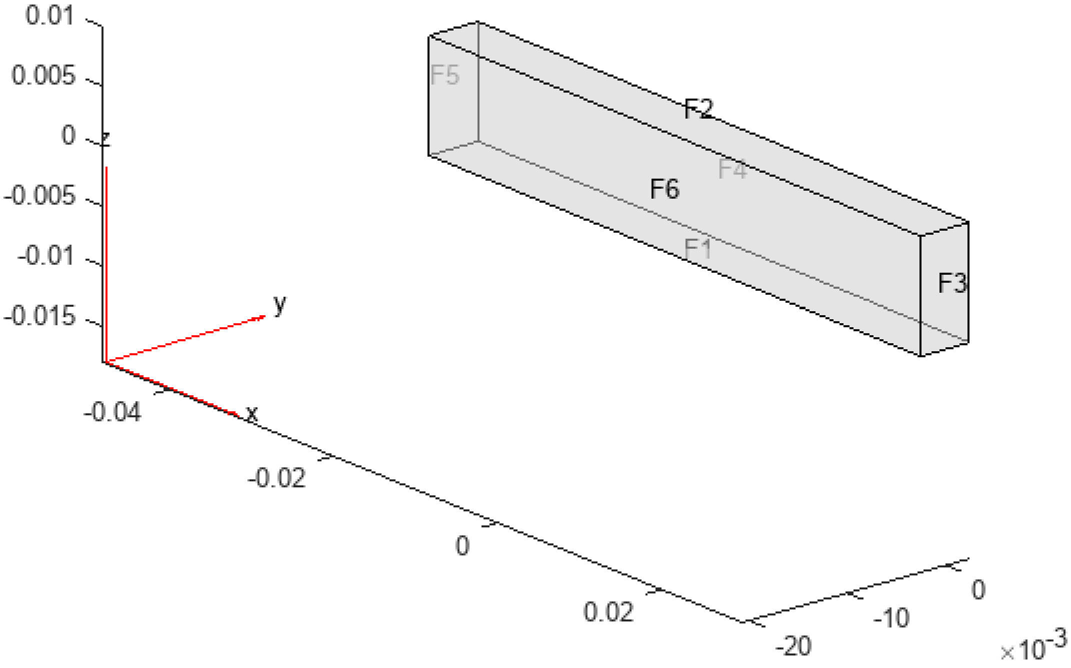
Evaluate the von Mises stress in a beam under a harmonic excitation.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Generate a mesh.

```
generateMesh(structuralmodel,"Hmax",0.01);
```

Specify the zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

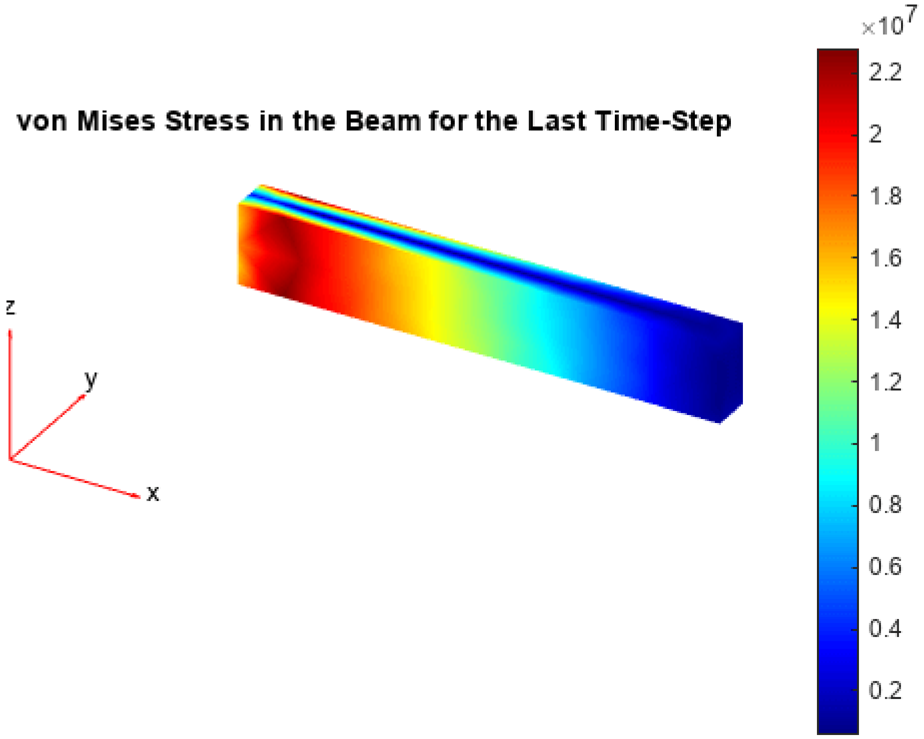
```
tlist = 0:0.002:0.2;
structuralresults = solve(structuralmodel,tlist);
```

Evaluate the von Mises stress in the beam.

```
vmStress = evaluateVonMisesStress(structuralresults);
```

Plot the von Mises stress for the last time-step.

```
figure
pdeplot3D(structuralmodel,"ColorMapData",vmStress(:,end))
title("von Mises Stress in the Beam for the Last Time-Step")
```



Solution to 3-D Electrostatic Analysis Model

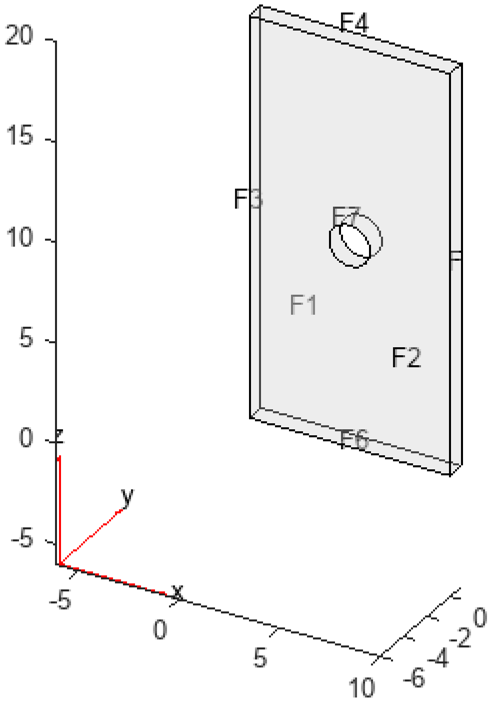
Solve an electromagnetic problem and find the electric potential and field distribution for a 3-D geometry representing a plate with a hole.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permittivity in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Apply the voltage boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
electromagneticBC(emagmodel,"Voltage",1000,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

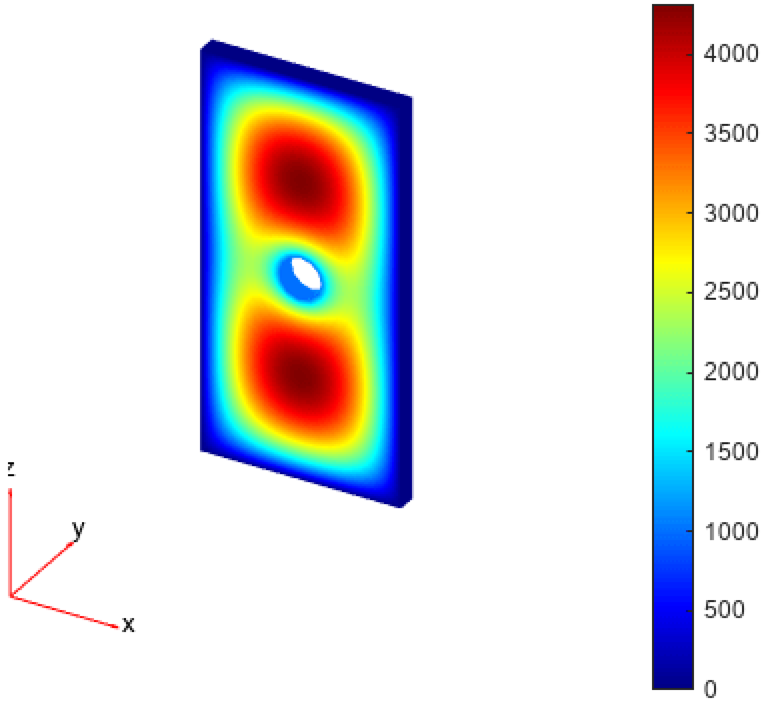
```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:
```

```
      ElectricPotential: [4359x1 double]
          ElectricField: [1x1 FEStruct]
    ElectricFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

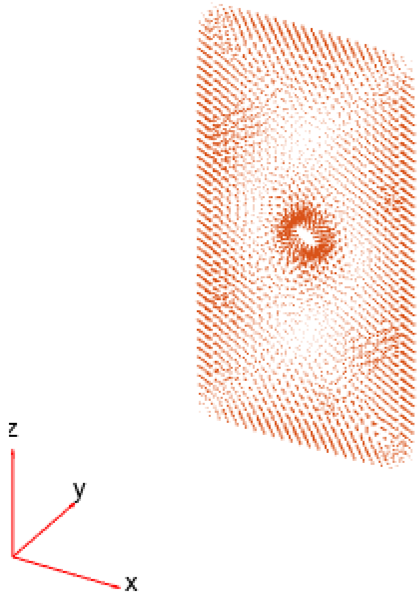
Plot the electric potential.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.ElectricPotential)
```



Plot the electric field.

```
pdeplot3D(emagmodel,"FlowData",[R.ElectricField.Ex ...
                                R.ElectricField.Ey ...
                                R.ElectricField.Ez])
```



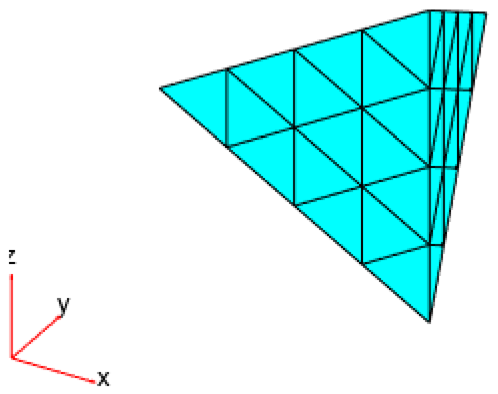
3-D Mesh Plot

Create a PDE model, include the geometry, and generate a mesh.

```
model = createpde;
importGeometry(model,"Tetrahedron.stl");
mesh = generateMesh(model,"Hmax",20,"GeometricOrder","linear");
```

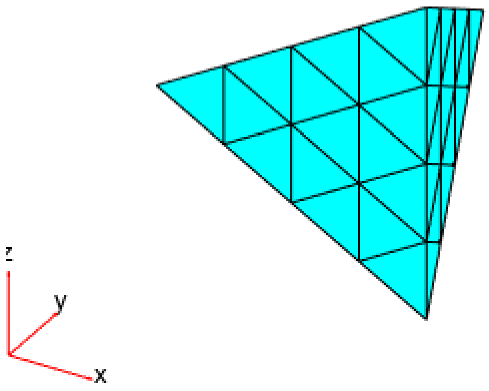
Plot the surface mesh.

```
pdeplot3D(model)
```



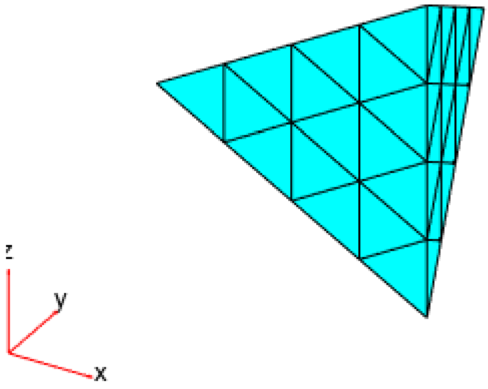
Alternatively, you can plot a mesh by using `mesh` as an input argument.

```
pdeplot3D(mesh)
```



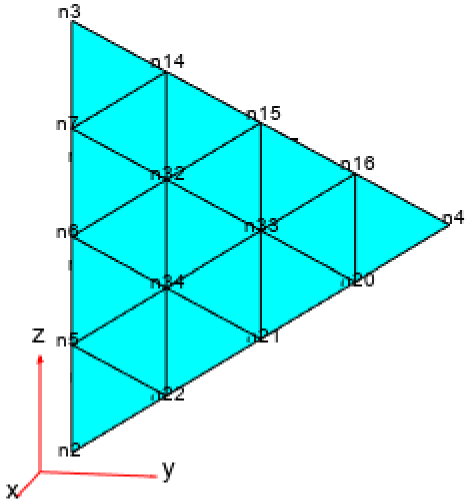
Another approach is to use the nodes and elements of the mesh as input arguments for `pdeplot3D`.

```
pdeplot3D(mesh.Nodes,mesh.Elements)
```



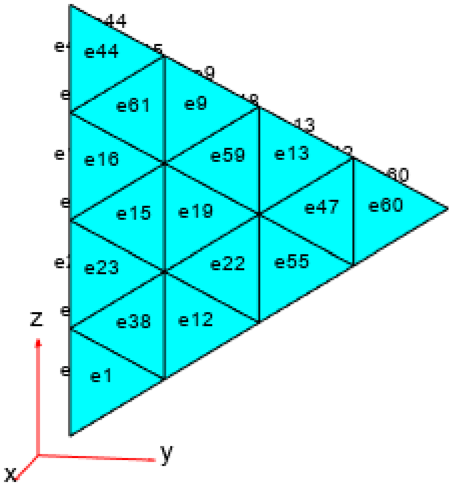
Display the node labels on the surface of a simple mesh.

```
pdeplot3D(model,"NodeLabels","on")
view(101,12)
```



Display the element labels.

```
pdeplot3D(model,"ElementLabels","on")
view(101,12)
```



Input Arguments

`model` — Model container

`PDEModel` object | `ThermalModel` object | `StructuralModel` object | `ElectromagneticModel` object

Model container, specified as a `PDEModel` object, `ThermalModel` object, `StructuralModel` object, or `ElectromagneticModel` object.

Example: `model = createpde(3)`

Example: `thermalmodel = createpde("thermal","steadystate")`

Example: `structuralmodel = createpde("structural","static-solid")`

Example: `emagmodel = createpde("electromagnetic","electrostatic")`

`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`

`nodes` — Nodal coordinates

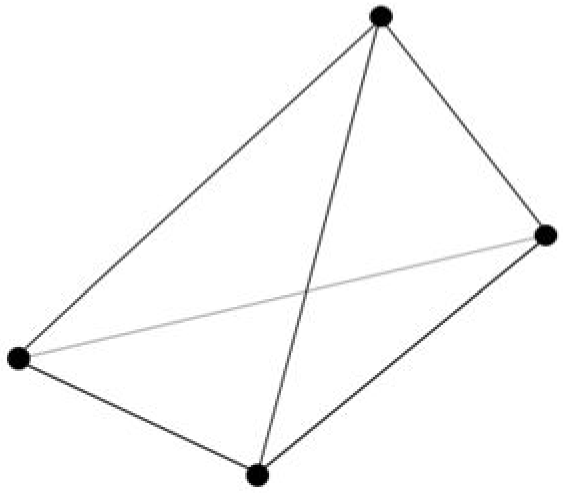
3-by-*NumNodes* matrix

Nodal coordinates, specified as a 3-by-*NumNodes* matrix. *NumNodes* is the number of nodes.

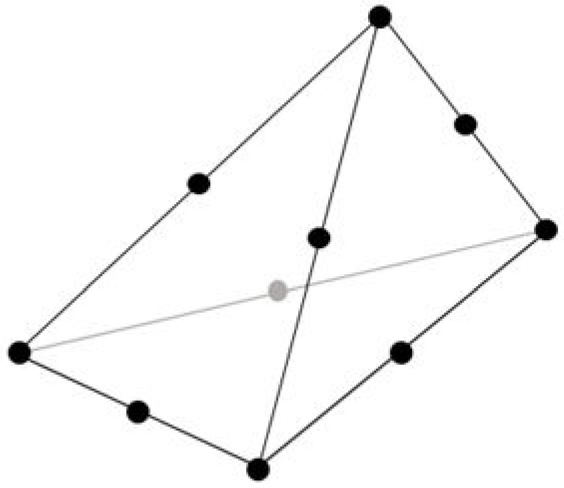
`elements` — Element connectivity matrix in terms of node IDs

4-by-*NumElements* matrix | 10-by-*NumElements* matrix

Element connectivity matrix in terms of the node IDs, specified as a 4-by-*NumElements* or 10-by-*NumElements* matrix. Linear meshes contain only corner nodes. For linear meshes, the connectivity matrix has four nodes per 3-D element. Quadratic meshes contain corner nodes and nodes in the middle of each edge of an element. For quadratic meshes, the connectivity matrix has 10 nodes per 3-D element.



3-D linear element



3-D quadratic element

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `pdeplot3D(model,"NodeLabels","on")`

`ColorMapData` — Data to plot as colored surface

column vector

Data to plot as a colored surface, specified as the comma-separated pair consisting of `"ColorMapData"` and a column vector with the number of elements that equals the number of points in the mesh. Typically, this data is the solution returned by `solvepde` for a scalar PDE problem and a component of the solution for a multicomponent PDE system.

Example: `"ColorMapData",results.NodalSolution`

Example: `"ColorMapData",results.NodalSolution(:,1)`

Data Types: `double`

`FlowData` — Data for quiver plot

matrix

Data for the quiver plot on page 5-958, specified as the comma-separated pair consisting of `"FlowData"` and an `M`-by-`3` matrix, where `M` is the number of mesh nodes. `FlowData` contains the *x*, *y*, and *z* values of the field at the mesh points. Set `FlowData` as follows:

```
results = solvepde(model);
[cgradx,cgrady,cgradz] = evaluateCGradient(results);
pdeplot3D(model,"FlowData",[cgradx cgrady cgradz])
```

`pdeplot3D` plots the real part of complex data.

Example: `"FlowData",[cgradx cgrady cgradz]`

Data Types: `double`

`Mesh` — Indicator to show mesh

`"off"` (default) | `"on"`

Indicator to show the mesh, specified as the comma-separated pair consisting of `"Mesh"` and `"on"` or `"off"`. Specify `"on"` to show the mesh in the plot.

Example: `"Mesh","on"`

Data Types: `char` | `string`

`NodeLabels` — Node labels

`"off"` (default) | `"on"`

Node labels, specified as the comma-separated pair consisting of `"NodeLabels"` and `"off"` or `"on"`.

Example: `"NodeLabels","on"`

Data Types: `char` | `string`

`ElementLabels` — Element labels

`"off"` (default) | `"on"`

Element labels, specified as the comma-separated pair consisting of `"ElementLabels"` and `"off"` or `"on"`.

Example: `"ElementLabels","on"`

Data Types: `char` | `string`

`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`

`Deformation` — Deformed shape for structural analysis models

`FEStruct` object representing displacement values at nodes

Deformed shape for structural analysis models, specified as the comma-separated pair consisting of `"Deformation"` and the `FEStruct` object representing displacement values at nodes. The displacement `FEStruct` object is a property of `StaticStructuralResults`, `TransientStructuralResults`, and `FrequencyStructuralResults`.

In an undeformed shape, center nodes in quadratic meshes are always added at half-distance between corners. When you plot a deformed shape, the center nodes might move away from the edge centers.

Example: `"Deformation",results.Displacement`

`DeformationScaleFactor` — Scaling factor for plotting deformed shape
positive number

Scaling factor for plotting the deformed shape, specified as the comma-separated pair consisting of `"DeformationScaleFactor"` and a positive number. Use this argument together with the `Deformation` name-value pair argument. The `pdeplot3D` function chooses the default value based on the geometry itself and on the magnitude of deformation.

Example: `"DeformationScaleFactor",1000`

Data Types: `double`

Output Arguments

`h` — Handles to graphics objects
vector

Handles to graphics objects, returned as a vector.

Version History

Introduced in R2015a

Electromagnetic Analysis

You can now plot electromagnetic results, such as electric and magnetic potentials, fields, and fluxes.

Improved performance for plots with many text labels

Performance change in R2020a

`pdeplot3d` shows faster rendering and better responsiveness for plots that display many text labels. Code containing `findobj(fig,'Type','Text')` no longer returns labels on figures produced by `pdeplot3d`.

Highlighting particular nodes and elements on mesh plots

`pdeplot3d` accepts node and element IDs as input arguments, letting you highlight particular nodes and elements on mesh plots.

Structural Analysis

You can now plot structural results, such as displacements, stresses, and strains.

Thermal Analysis

You can now plot thermal results, such as temperatures and temperature gradients.

Transparency, node and element labels

You can now set plot transparency by using `FaceAlpha`, and display node and element labels by using `NodeLabels` and `ElementLabels`, respectively.

See Also

`PDEModel` | `pdeplot` | `pdegplot` | `pdemesh`

Topics

"3-D Solution and Gradient Plots with MATLAB Functions" on page 3-344
"Solve Problems Using PDEModel Objects" on page 2-2

pdepoly

Package: `pde`

Draw polygon in PDE Modeler app

Syntax

```
pdepoly(X,Y)
pdepoly(X,Y,label)
```

Description

`pdepoly(X,Y)` draws a polygon with the corner coordinates (vertices) defined by `X` and `Y`. The `pdepoly` command opens the PDE Modeler app with the specified polygon drawn in it. If the app is already open, `pdepoly` adds the specified polygon to the app window without deleting any existing shapes.

`pdepoly` updates the state of the geometry description matrix inside the PDE Modeler app to include the polygon. You can export the geometry description matrix from the PDE Modeler app to the MATLAB Workspace by selecting **DrawExport Geometry Description, Set Formula, Labels...**. For details on the format of the geometry description matrix, see `decsg`.

`pdepoly(X,Y,label)` assigns a name to the polygon. Otherwise, `pdepoly` uses a default name, such as `P1`, `P2`, and so on.

Examples

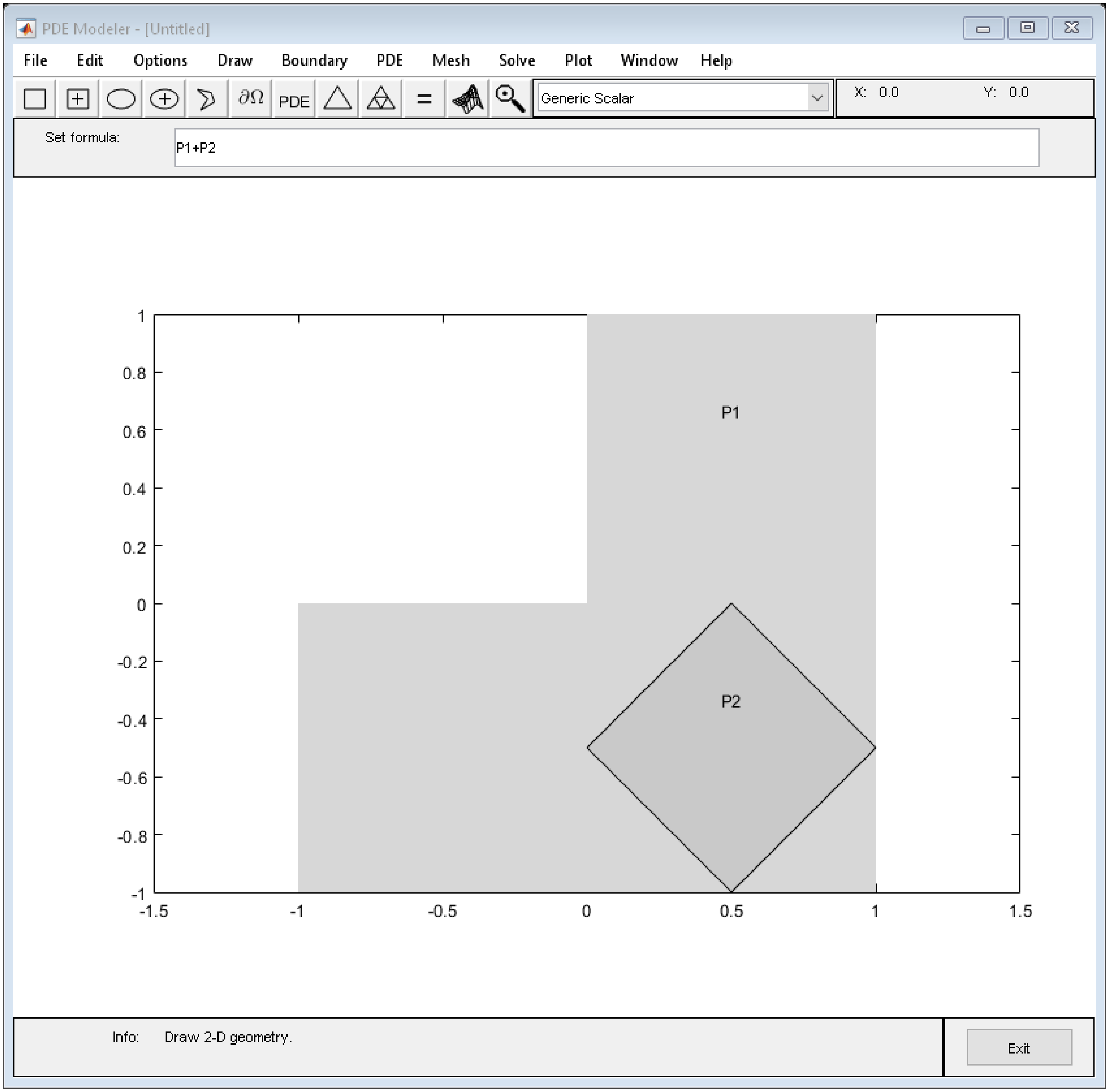
Draw Polygon in PDE Modeler App

Open the PDE Modeler app window containing a polygon representing the L-shaped membrane geometry.

```
pdepoly([-1 0 0 1 1 -1],[0 0 1 1 -1 -1])
```

Call the `pdepoly` command again to draw the diamond-shaped region with corners in `(0.5,0)`, `(1,-0.5)`, `(0.5,-1)`, and `(0,-0.5)`. The `pdepoly` command adds the second polygon to the app window without deleting the first.

```
pdepoly([0.5 1 0.5 0],[0 -0.5 -1 -0.5])
```



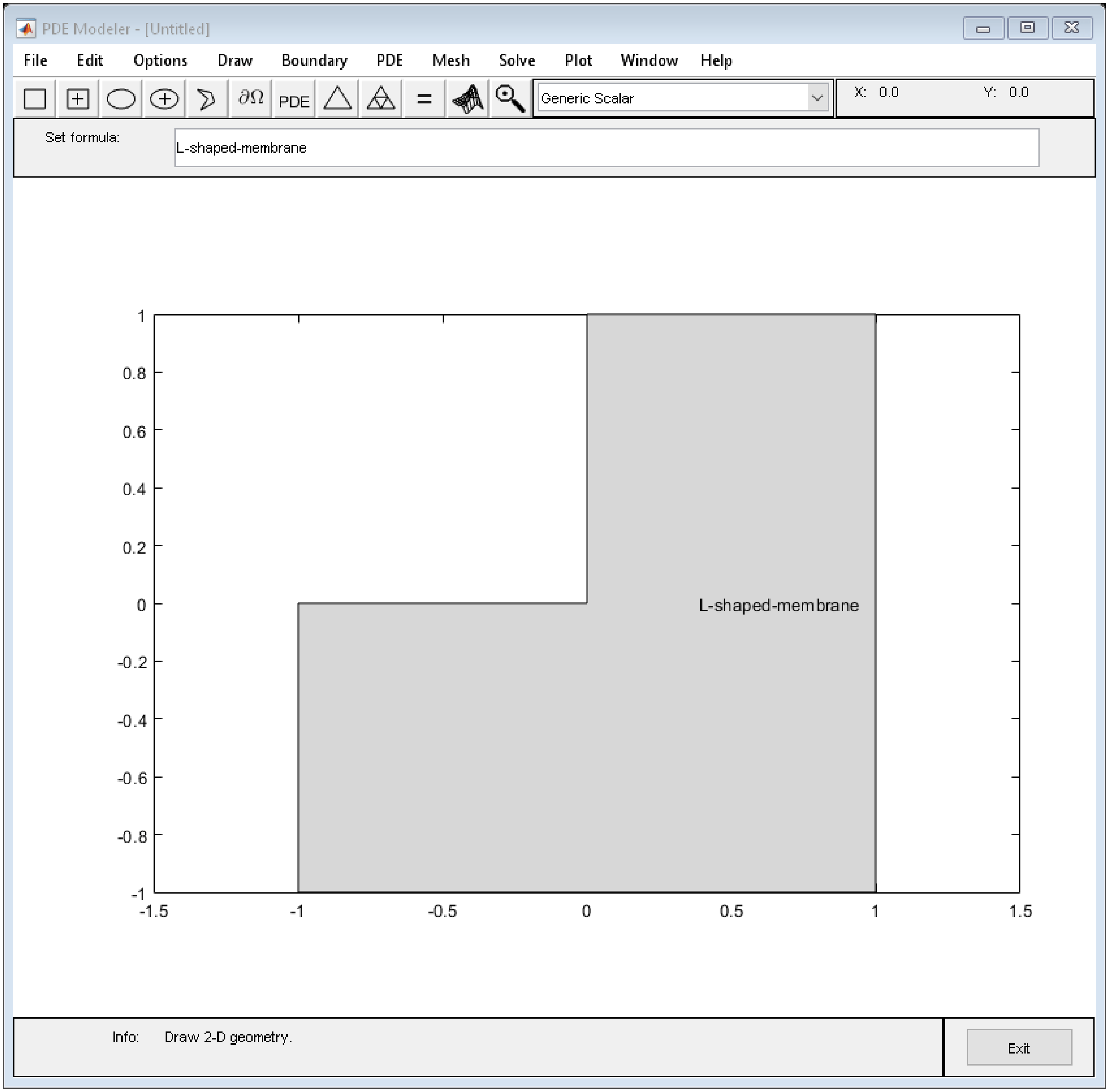
Set formula: P1+P2

Info: Draw 2-D geometry.

Assign Name to Polygon in PDE Modeler App

Open the PDE Modeler app window with a polygon representing the L-shaped membrane geometry. Assign the name `L-shaped-membrane` to this polygon.

```
pdepoly([-1 0 0 1 1 -1],[0 0 1 1 -1 -1],"L-shaped-membrane")
```



Generic Scalar

Set formula: L-shaped-membrane

Info: Draw 2-D geometry.

Input Arguments

X — *x*-coordinates of vertices

vector of real numbers

x-coordinates of vertices defining the polygon, specified as a vector of real numbers.

Example: `pdepoly([-1 0 0 1 1 -1],[0 0 1 1 -1 -1])`

Data Types: `double`

Y — *y*-coordinates of vertices
vector of real numbers

y-coordinates of vertices defining the polygon, specified as a vector of real numbers.

Example: `pdepoly([-1 0 0 1 1 -1],[0 0 1 1 -1 -1])`

Data Types: `double`

`label` — Name
character vector | string scalar

Name of the polygon, specified as a character vector or string scalar.

Data Types: `char` | `string`

Tips

- `pdepoly` opens the PDE Modeler app and draws a polygon. If, instead, you want to draw polygons in a MATLAB figure, use the `plot` function, for example:

```
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,'.-')
```

Version History

Introduced before R2006a

See Also

`pdecirc` | `pdeellip` | `pderect` | **PDE Modeler**

pdeprtni

(Not recommended) Interpolate triangle midpoint data to mesh nodes

---

Note `pdeprtni` is not recommended. Use `interpolateSolution` and `evaluateGradient` instead.

Syntax

`un = pdeprtni(p,t,ut)`

Description

`un = pdeprtni(p,t,ut)` uses the data `ut` at mesh triangle midpoints to linearly interpolate data at mesh nodes.

`pdeprtni` and `pdeintrp` are not inverse functions because the interpolation introduces some averaging.

Examples

Data at Mesh Nodes and Triangle Midpoints

Solve the equation $-\Delta u = 1$ on the L-shaped membrane and interpolate the solution from nodes to triangle midpoints.

First, create a `[p,e,t]` mesh on the L-shaped membrane.

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

Solve the equation using the Dirichlet boundary condition $u = 0$ on $\partial\Omega$. The result is the solution at the mesh nodes.

```
un = assempde('lshapeb',p,e,t,1,0,1);
```

Interpolate the solution from the mesh nodes to the triangle midpoints.

```
ut = pdeintrp(p,t,un);
```

Interpolate the solution back to nodes by using the `pdeprtni` function. Compare the result and the original solution at the mesh nodes. The `pdeprtni` and `pdeintrp` functions are not inverse.

```
un2 = pdeprtni(p,t,ut);
isequal(un,un2)
```

```
ans = logical
   0
```

Input Arguments

p — Mesh nodes
matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

`t` — Mesh elements
matrix

Mesh elements, specified as a `4`-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

`ut` — Data at triangle midpoints
row vector

Data at triangle midpoints, specified as a row vector.

For a PDE system of *N* equations and a mesh with `Nt` elements, the first `Nt` values of `ut` describe the first component, the following `Nt` values of `ut` describe the second component, and so on.

Output Arguments

un — Data at nodes
column vector

Data at nodes, returned as a column vector.

For a PDE system of *N* equations and a mesh with `Np` node points, the first `Np` values of `un` describe the first component, the following `Np` values of `un` describe the second component, and so on.

Data Types: `double`

Version History
Introduced before R2006a

Not recommended
Not recommended starting in R2016a

`pdeprtni` is not recommended. Use `interpolateSolution` and `evaluateGradient` instead. There are no plans to remove `pdeprtni`.

See Also
`pdeintrp` | `evaluate` | `pdeInterpolant`

Topics
“Mesh Data as [p,e,t] Triples” on page 2-172

pderect

Package: `pde`

Draw rectangle in PDE Modeler app

Syntax

```
pderect([xmin xmax ymin ymax])
pderect([xmin xmax ymin ymax], label)
```

Description

`pderect([xmin xmax ymin ymax])` draws a rectangle with the corner coordinates defined by `[xmin xmax ymin ymax]`. The `pderect` command opens the PDE Modeler app with the specified rectangle drawn in it. If the app is already open, `pderect` adds the specified rectangle to the app window without deleting any existing shapes.

`pderect` updates the state of the geometry description matrix inside the PDE Modeler app to include the rectangle. You can export the geometry description matrix from the PDE Modeler app to the MATLAB Workspace by selecting **DrawExport Geometry Description, Set Formula, Labels...**. For details on the format of the geometry description matrix, see `decsg`.

`pderect([xmin xmax ymin ymax], label)` assigns a name to the rectangle. Otherwise, `pderect` uses a default name, such as `R1`, `R2`, and so on. For squares, `pderect` uses the default names `SQ1`, `SQ2`, and so on.

Examples

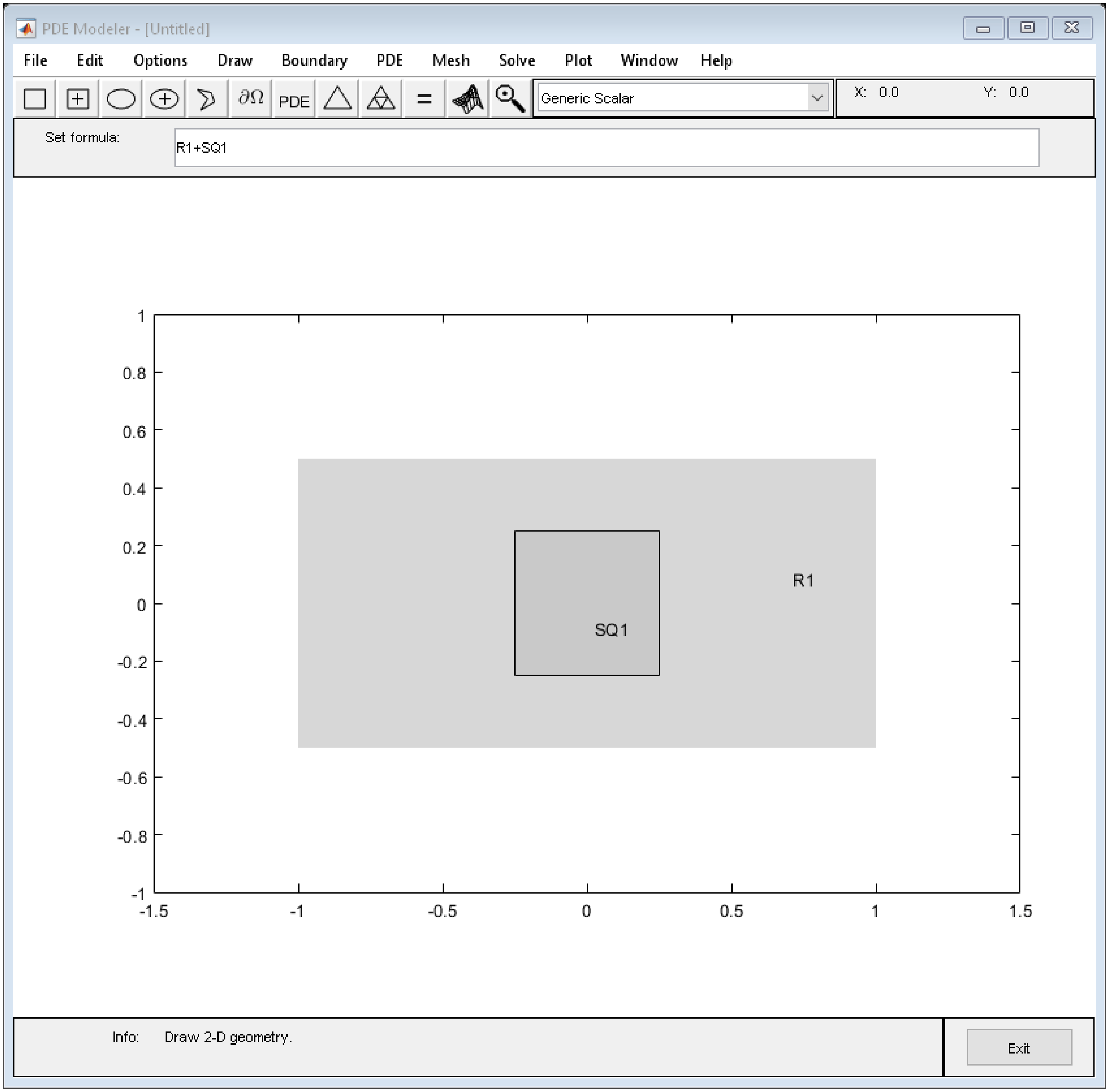
Draw Rectangle in PDE Modeler App

Open the PDE Modeler app window containing a rectangle with the corners at `(-1,-0.5)`, `(-1,0.5)`, `(1,0.5)`, and `(1,-0.5)`.

```
pderect([-1 1 -0.5 0.5])
```

Call the `pderect` command again to draw a square with the corners at `(-0.25,-0.25)`, `(-0.25,0.25)`, `(0.25,0.25)`, and `(0.25,-0.25)`. The `pderect` command adds the square to the app window without deleting the rectangle.

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



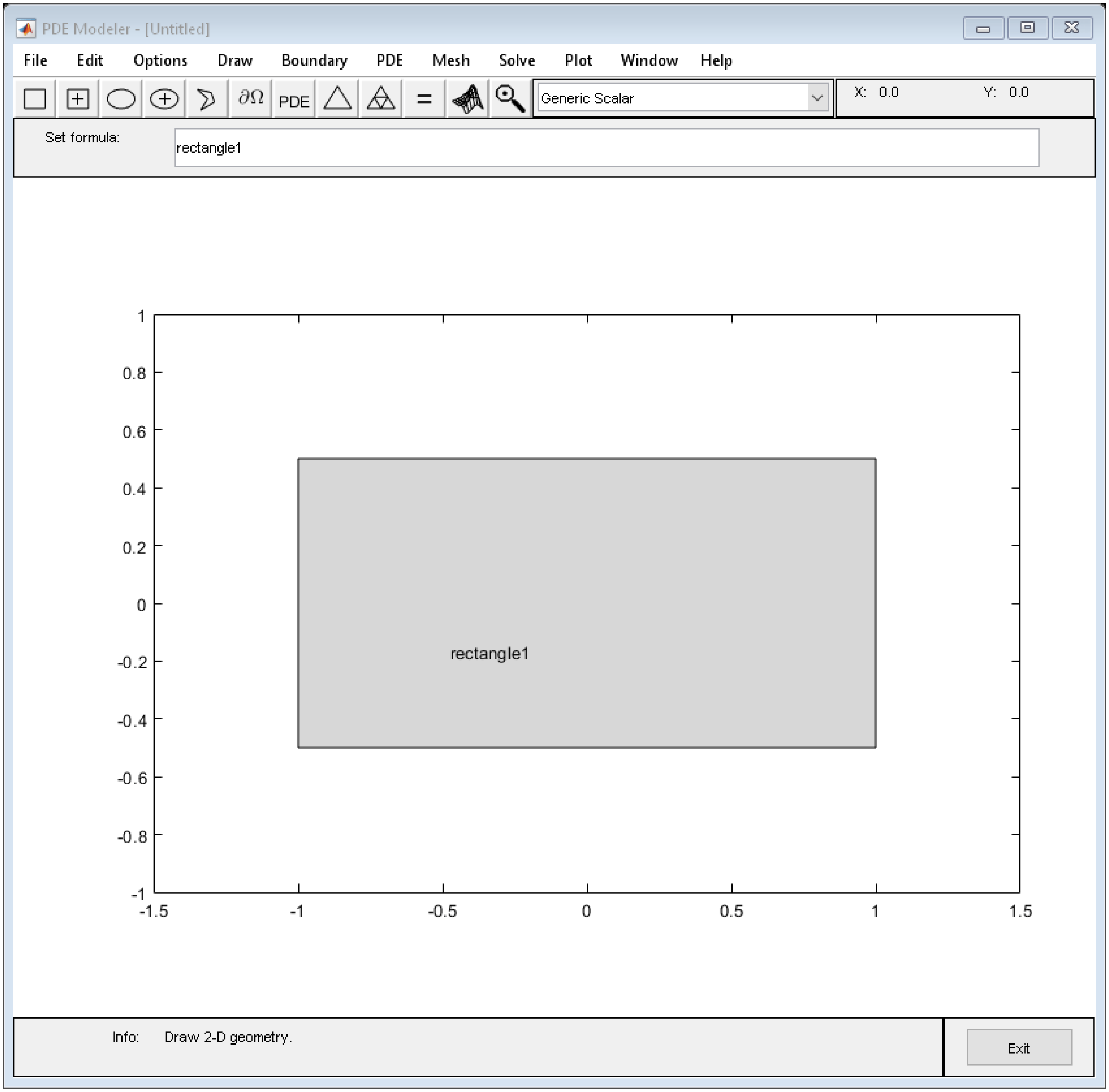
Set formula: R1+SQ1

Info: Draw 2-D geometry.

Assign Name to Rectangle in PDE Modeler App

Open the PDE Modeler app window and draw a rectangle with the corners at `(-1,-0.5)`, `(-1,0.5)`, `(1,0.5)`, and `(1,-0.5)`. Assign the name `rectangle1` to this rectangle.

```
pderect([-1 1 -0.5 0.5],"rectangle1")
```



Set formula: rectangle1

rectangle1

Info: Draw 2-D geometry.

Input Arguments

`[xmin xmax ymin ymax]` — Corner coordinates

vector of real numbers

Corner coordinates defining the rectangle, specified as a vector of real numbers.

Example: `pderect([-1 0 -1 0])`

Data Types: `double`

`label` — Name
character vector | string scalar

Name of the rectangle, specified as a character vector or string scalar.

Data Types: `char` | `string`

Tips

- `pderect` opens the PDE Modeler app and draws a rectangle. If, instead, you want to draw rectangles in a MATLAB figure, use the `rectangle` function, for example, `rectangle('Position',[1,2,5,6])`.

Version History

Introduced before R2006a

See Also

`pdecirc` | `pdeellip` | `pdepoly` | **PDE Modeler**

pdesdp

(Not recommended) Indices of subset of mesh nodes belonging to specified faces of 2-D geometry

---

Note `pdesdp` is not recommended. Use `findNodes` instead.

Syntax

```
s = pdesdp(p,e,t,FaceID)
[i,c] = pdesdp(p,e,t,FaceID)
___ = pdesdp(p,e,t)
```

Description

`s = pdesdp(p,e,t,FaceID)` returns the indices of the nodes of the `[p,e,t]` mesh shared between two or more faces listed in `FaceID`.

`[i,c] = pdesdp(p,e,t,FaceID)` returns the indices of the nodes of the `[p,e,t]` mesh belonging strictly to faces `FaceID` as `i`. It also returns the indices of the nodes shared between `FaceID` and faces not listed in `FaceID` as `c`.

`___ = pdesdp(p,e,t)` uses any of the previous syntaxes, assuming that `FaceID` is a list of all faces of a 2-D geometry.

Examples

Mesh Nodes of Specified Faces

Find the indices of the mesh nodes belonging strictly to the specified faces and the indices of the nodes shared between faces.

Define two circles and a rectangle and place these in one matrix.

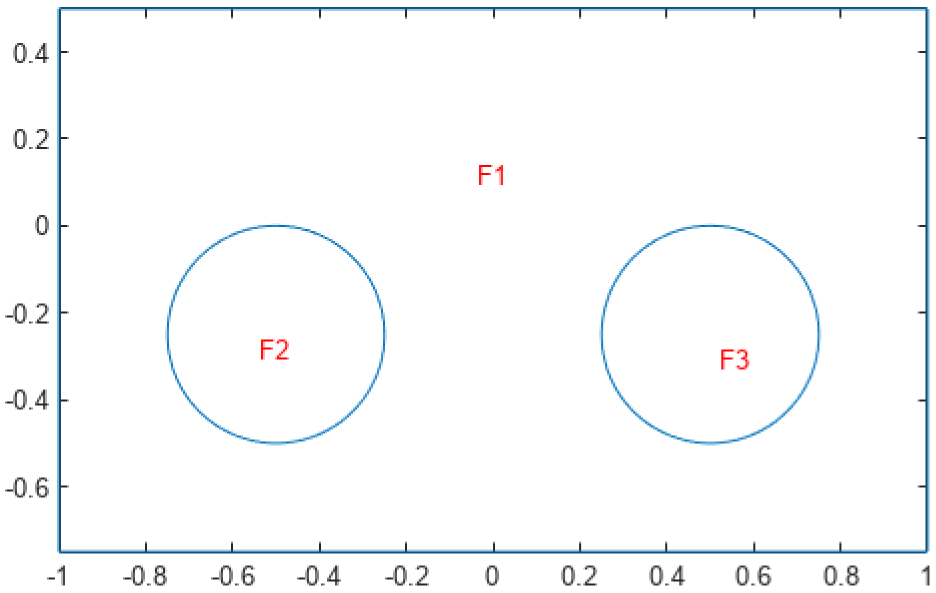
```
R1 = [3,4,-1,1,1,-1,0.5,0.5,-0.75,-0.75]';
C1 = [1,-0.5,-0.25,0.25]';
C2 = [1,0.5,-0.25,0.25]';
C1 = [C1;zeros(length(R1) - length(C1),1)];
C2 = [C2;zeros(length(R1) - length(C2),1)];
gd = [R1,C1,C2];
```

Create a set formula that adds the circles to the rectangle.

```
sf = 'R1+C1+C2';
```

Create and plot the geometry.

```
ns = char('R1','C1','C2');
ns = ns';
gd = decsg(gd,sf,ns);
pdegplot(gd,'FaceLabels','on')
```

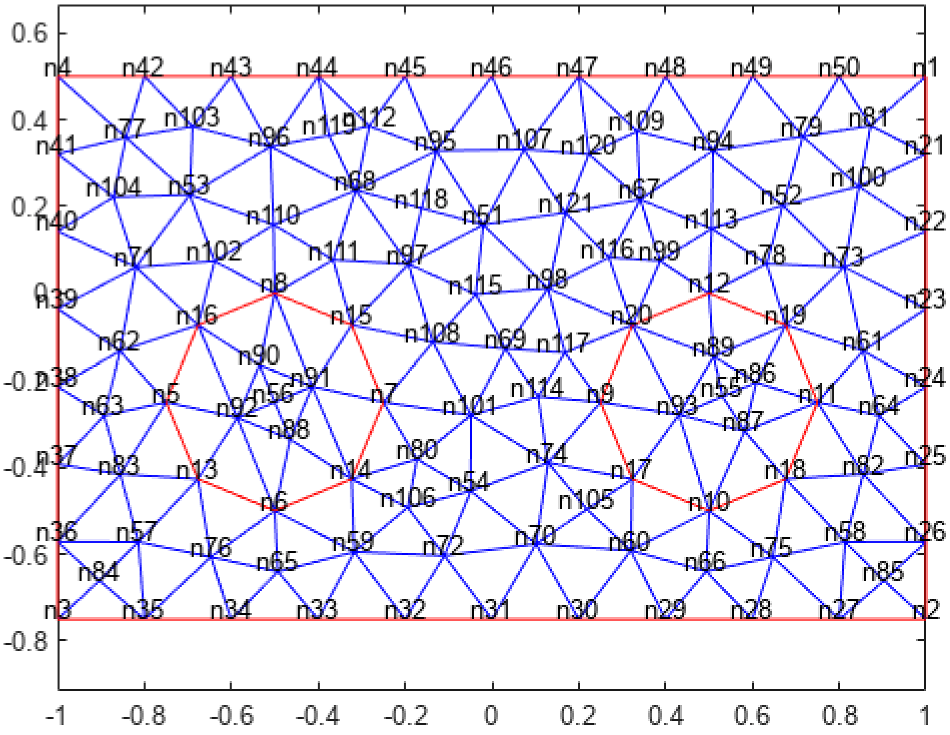


Create a mesh.

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

Plot the mesh with the node labels.

```
pdemesh(p,e,t,'NodeLabels','on')
```



Find the indices of the mesh nodes shared between faces 1 and 2.

```
s1 = pdesdp(p,e,t,[1 2])
```

```
s1 = 1×8

     5     6     7     8    13    14    15    16
```

Find the indices of the mesh nodes shared between faces 2 and 3. Since these faces do not share any nodes, `pdesdp` returns an empty vector.

```
s2 = pdesdp(p,e,t,[2 3])
```

```
s2 =

  1x0 empty double row vector
```

Find the nodes belonging strictly to face 2 and also the nodes shared between face 2 and other faces. Face 2 shares nodes only with face 1, therefore, vectors `c` and `s1` consist of the same face IDs.

```
[i,c] = pdesdp(p,e,t,2)
```

```
i = 1×5

    56    88    90    91    92
```

```
c = 1×8

     5     6     7     8    13    14    15    16
```

Input Arguments

p — Mesh nodes

matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

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 details on the mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh elements

matrix

Mesh elements, specified as a `4`-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

FaceID — Face IDs

vector of integers

Face IDs, specified as a vector of integers.

Data Types: `double`

Output Arguments

s — Mesh nodes shared between two or more specified faces

row vector of positive integers

Mesh nodes shared between two or more specified faces, returned as a row vector of positive integers representing the indices of the nodes.

i — Mesh nodes belonging only to specified faces

row vector of positive integers

Mesh nodes belonging only to specified faces, returned as a row vector of positive integers representing the indices of the nodes.

c — Mesh nodes shared between specified and other faces

row vector of positive integers

Mesh nodes shared between specified and other faces, returned as a row vector of positive integers representing the indices of the nodes.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2018a

`pdesdp` is not recommended. Use `findNodes` instead. There are no plans to remove `pdesdp`.

See Also

`pdesde` | `pdesdt`

Topics

"Mesh Data as [p,e,t] Triples" on page 2-172

pdesde

(Not recommended) Indices of edges of mesh elements belonging to the specified faces of 2-D geometry

---

Note `pdesde` is not recommended. Use `faceEdges` instead.

Syntax

```
i = pdesde(e,FaceID)
i = pdesde(e)
```

Description

`i = pdesde(e,FaceID)` returns the indices of the `[p,e,t]` mesh edges that belong to outer boundaries of the geometry for a set of faces listed in `FaceID`.

`i = pdesde(e)` assumes that `FaceID` is a list of all faces of a 2-D geometry.

Examples

Edges of Mesh Elements for Specified Faces

Find the indices of the mesh elements' edges located on the outer boundaries of the geometry and belonging to the specified faces.

Define two circles and a rectangle and place these in one matrix.

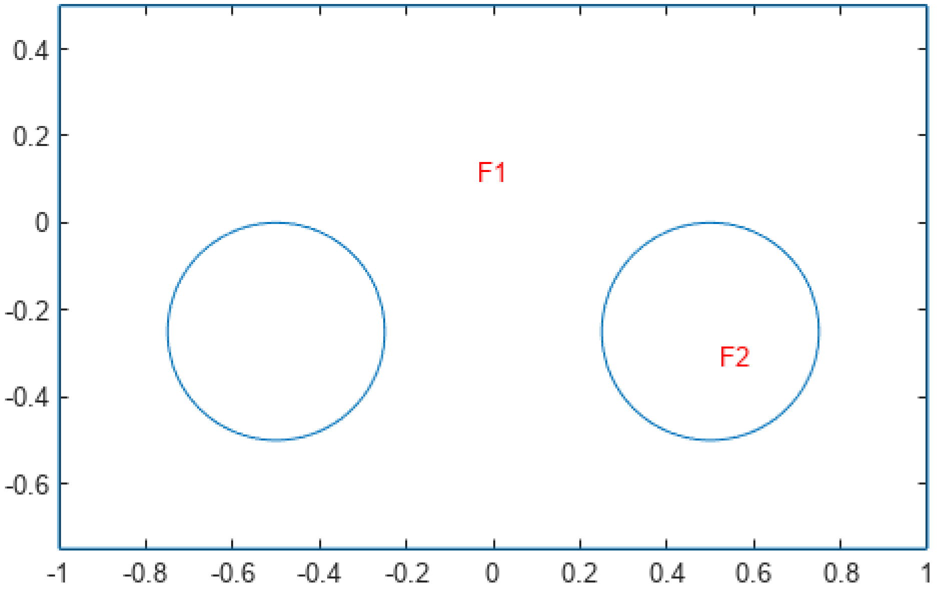
```
R1 = [3,4,-1,1,1,-1,0.5,0.5,-0.75,-0.75]';
C1 = [1,-0.5,-0.25,0.25]';
C2 = [1,0.5,-0.25,0.25]';
C1 = [C1;zeros(length(R1) - length(C1),1)];
C2 = [C2;zeros(length(R1) - length(C2),1)];
gd = [R1,C1,C2];
```

Create a set formula that subtracts one circle from the rectangle and adds the other circle to the rectangle.

```
sf = 'R1-C1+C2';
```

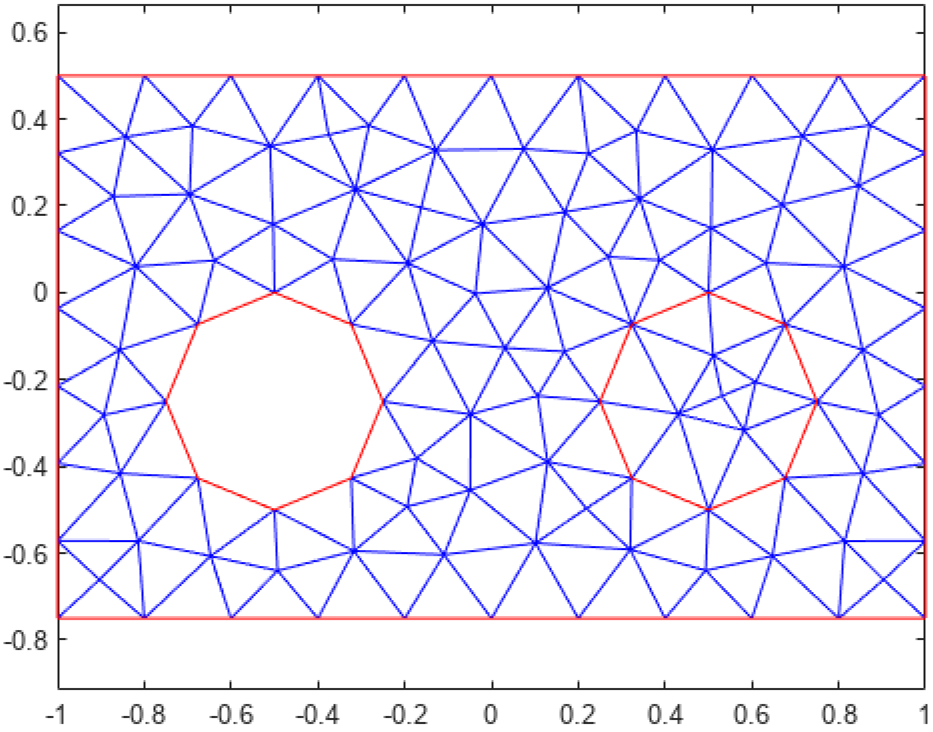
Create and plot the geometry.

```
ns = char('R1','C1','C2');
ns = ns';
gd = decsg(gd,sf,ns);
pdegplot(gd,'FaceLabels','on')
```



Create and plot a mesh.

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



Find the indices of the mesh elements' edges located on the outer boundaries and belonging to face 1. Display the result as a column vector.

```
i1 = pdesde(e,1);
i1.'
```

```
ans = 42×1

     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
      ⋮
```

The resulting vector contains indices of all mesh edges in this geometry, except the eight internal edges surrounding face 2.

```
length(e) - length(i1)
```

```
ans = 8
```

Use the `pdesde` function to find the mesh edges surrounding face 2. The result is an empty vector because none of these mesh edges belong to the outer boundary of the geometry.

```
i2 = pdesde(e,2)
```

```
i2 =

  1x0 empty double row vector
```

Input Arguments

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 details on the mesh data representation, see `initmesh`.

Data Types: `double`

`FaceID` — Face IDs

vector of integers

Face IDs, specified as a vector of integers.

Data Types: `double`

Output Arguments

`i` — Indices of mesh edges on outer boundaries

vector of integers

Indices of the mesh edges on the outer boundaries, returned as a vector of integers.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2021b

`pdesde` is not recommended. Use `faceEdges` instead. There are no plans to remove `pdesde`.

See Also

`pdesdp` | `pdesdt`

Topics

“Mesh Data as [p,e,t] Triples” on page 2-172

pdesdt

(Not recommended) Indices of subset of mesh elements belonging to specified faces of 2-D geometry

---

Note `pdesdt` is not recommended. Use `findElements` instead.

Syntax

```
i = pdesdt(t,FaceID)
i = pdesdt(t)
```

Description

`i = pdesdt(t,FaceID)` returns the indices of the mesh elements of the `[p,e,t]` mesh that belong to the set of faces listed in `FaceID`.

`i = pdesdt(t)` assumes that `FaceID` is a list of all faces of a 2-D geometry.

Examples

Mesh Elements of Specified Faces

Find the indices of the mesh elements belonging to the specified faces.

Define two circles and a rectangle and place these in one matrix.

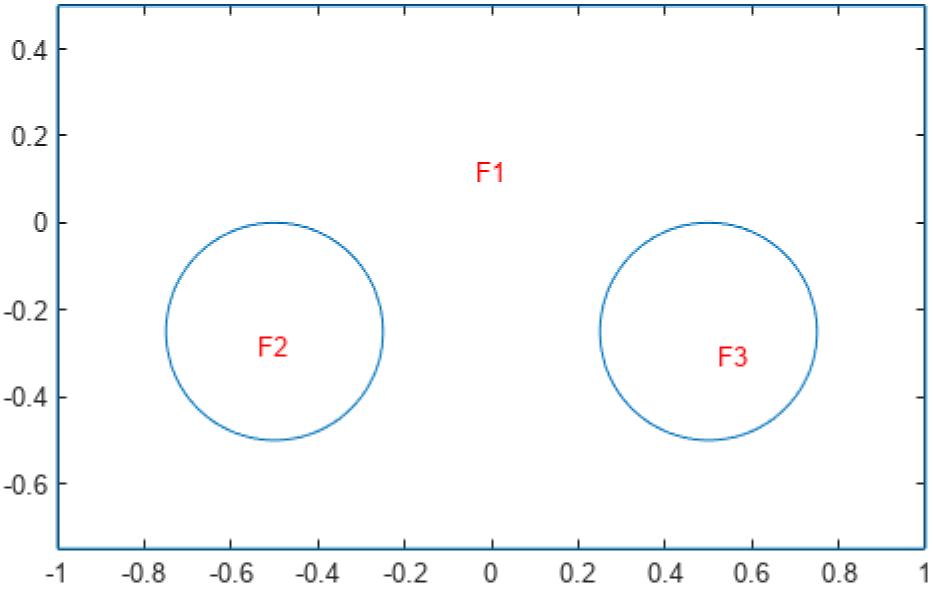
```
R1 = [3,4,-1,1,1,-1,0.5,0.5,-0.75,-0.75]';
C1 = [1,-0.5,-0.25,0.25]';
C2 = [1,0.5,-0.25,0.25]';
C1 = [C1;zeros(length(R1) - length(C1),1)];
C2 = [C2;zeros(length(R1) - length(C2),1)];
gd = [R1,C1,C2];
```

Create a set formula that adds the circles to the rectangle.

```
sf = 'R1+C1+C2';
```

Create and plot the geometry.

```
ns = char('R1','C1','C2');
ns = ns';
gd = decsg(gd,sf,ns);
pdegplot(gd,'FaceLabels','on')
```

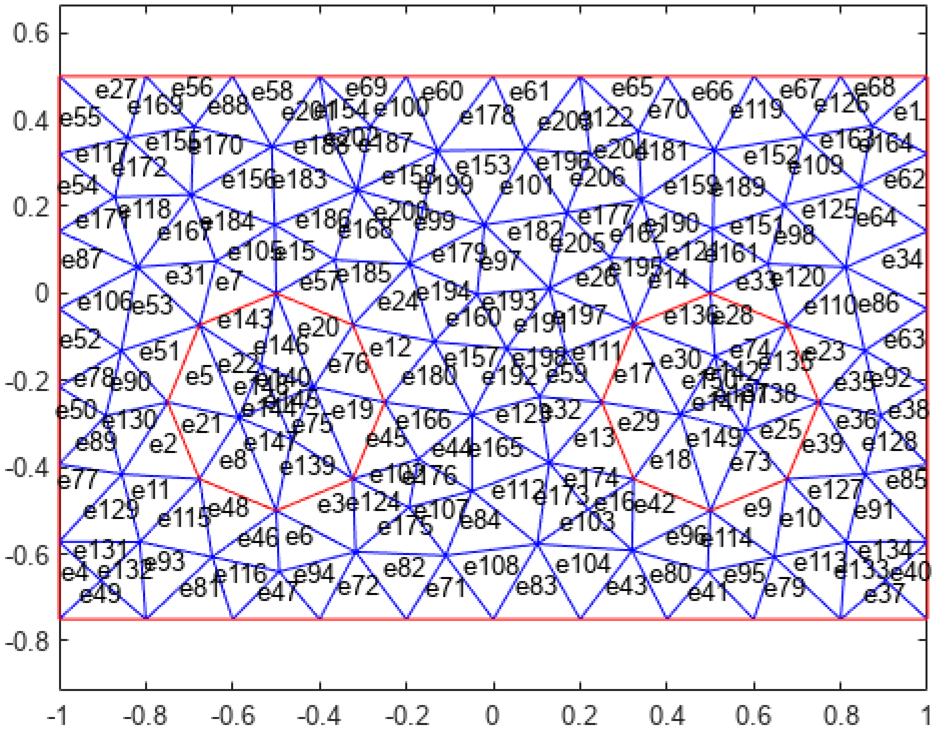


Create a mesh.

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

Plot the mesh with the element labels.

```
pdemesh(p,e,t,'ElementLabels','on')
```



Find the indices of the mesh elements that belong to face 2. Display the result as a column vector.

```
i = pdesdt(t,2);
i.'
```

```
ans = 16×1

     5
     8
    19
    20
    21
    22
    75
    76
   139
   140
     ⋮
```

Input Arguments

t — Mesh elements
matrix

Mesh elements, specified as a `4`-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

`FaceID` — Face IDs
vector of integers

Face IDs, specified as a vector of integers.

Data Types: `double`

Output Arguments

`i` — Mesh elements belonging to specified faces
vector of integers

Mesh elements belonging to the specified faces, returned as a vector of integers.

Version History

Introduced before R2006a

Not recommended
Not recommended starting in R2018a

`pdesdt` is not recommended. Use `findElements` instead. There are no plans to remove `pdesdt`.

See Also

`pdesdp` | `pdesde`

Topics
“Mesh Data as [p,e,t] Triples” on page 2-172

pdesmech

(Not recommended) Calculate structural mechanics tensor

Note `pdesmech` is not recommended. Use the PDE Modeler app or the “Structural Mechanics” workflow instead. For more information, see “Compatibility Considerations”.

Syntax

`structMechT = pdesmech(p,t,c,u,Name,Value)`

Description

`structMechT = pdesmech(p,t,c,u,Name,Value)` returns a tensor evaluated at the center of each triangle in `t`. The tensors are stresses and strains for structural mechanics applications with plane stress or plane strain conditions. If you do not specify `Name,Value`, the function returns the von Mises effective stress for plane stress conditions.

Export the solution, the mesh, and the PDE coefficients to the MATLAB workspace. Then use `pdesmech` for postprocessing of a solution computed in one of the structural mechanics application modes of the **PDE Modeler** app.

When calculating shear stresses and strains and the von Mises effective stress in plane strain mode, specify Poisson's ratio.

Examples

Stresses in Clamped Plate

Solve a structural mechanics problem in the **PDE Modeler** app, and export the solution to the MATLAB workspace. Then use `pdesmech` to compute the x-component of the stress and the von Mises effective 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 these properties:

- Dimensions are 1-by-1-by-0.001 m.
- Inset is 1/3-by-1/3 m.
- Rounded cut runs from (2/3, 1) to (1, 2/3).
- Young's modulus is $196 \cdot 10^3$ (MN/m^2).
- Poisson's ratio is 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.

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)
```

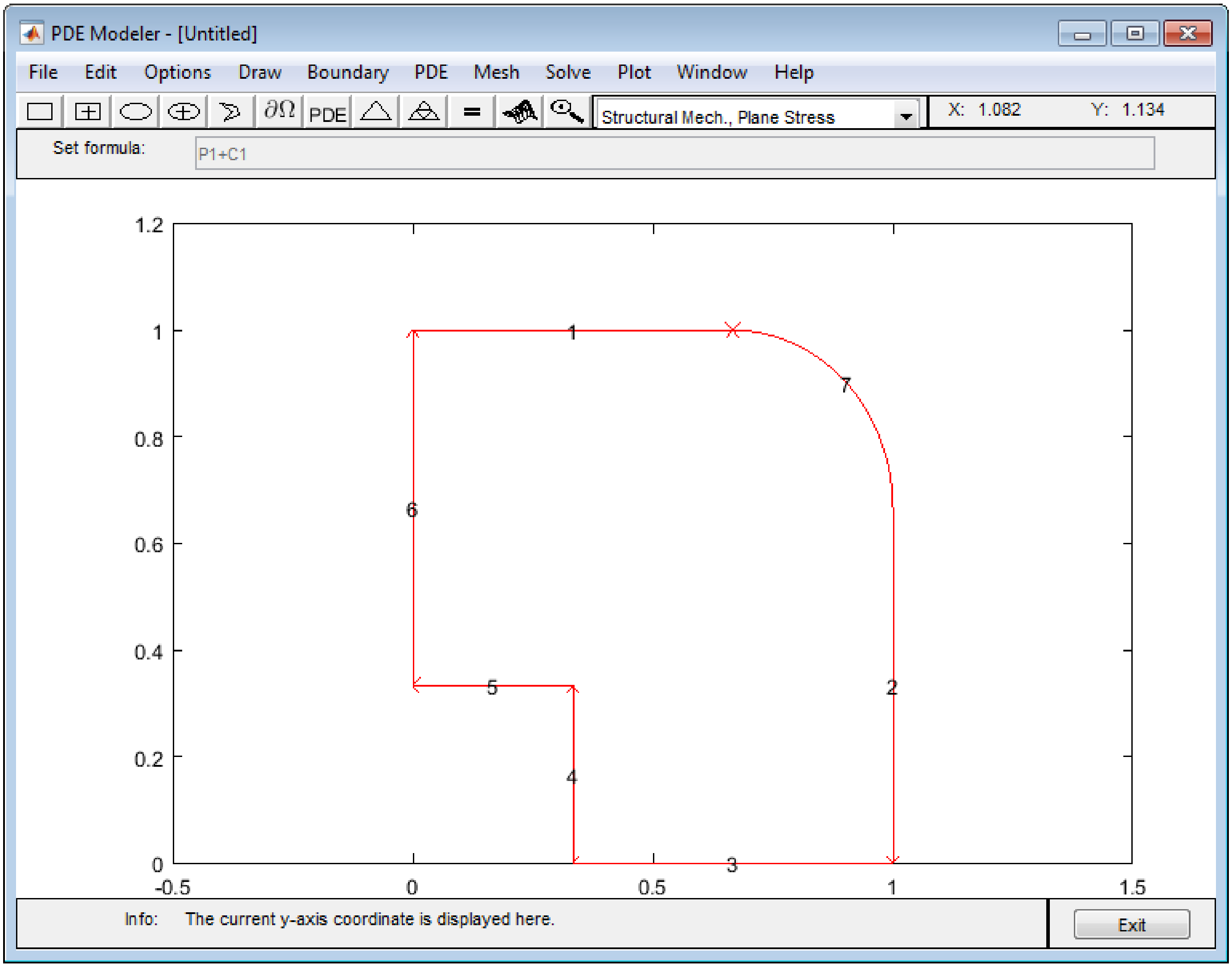
Set the *x*-axis limit to `[-0.5 1.5]` and *y*-axis limit to `[0 1.2]`. To do this, select **Options** > **Axes Limits** and set the corresponding ranges.

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

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

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

Display the edge labels by selecting **Boundary** > **Show Edge Labels**.



PDE Modeler - [Untitled]

Structural Mech., Plane Stress

X: 1.082 Y: 1.134

Set formula: P1+C1

Info: The current y-axis coordinate is displayed here.

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`.

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 `rho = 0`.

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

Further refine 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`.

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

Export the PDE coefficients, mesh, and solution to the MATLAB workspace.

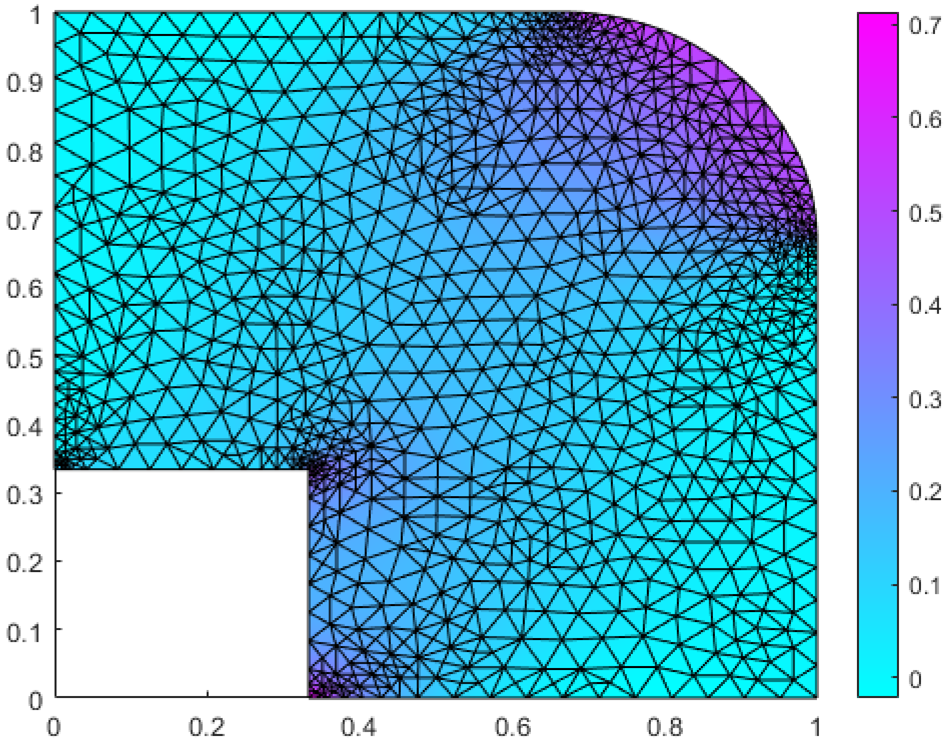
- To export the PDE coefficients, select **PDE** > **Export PDE Coefficients**.
- To export the mesh, select **Mesh** > **Export Mesh**.
- To export the solution, select **Solve** > **Export Solution**.

Find the *x*-component of the stress using the `pdesmech` function.

```
sx = pdesmech(p,t,c,u,'tensor','sxx');
```

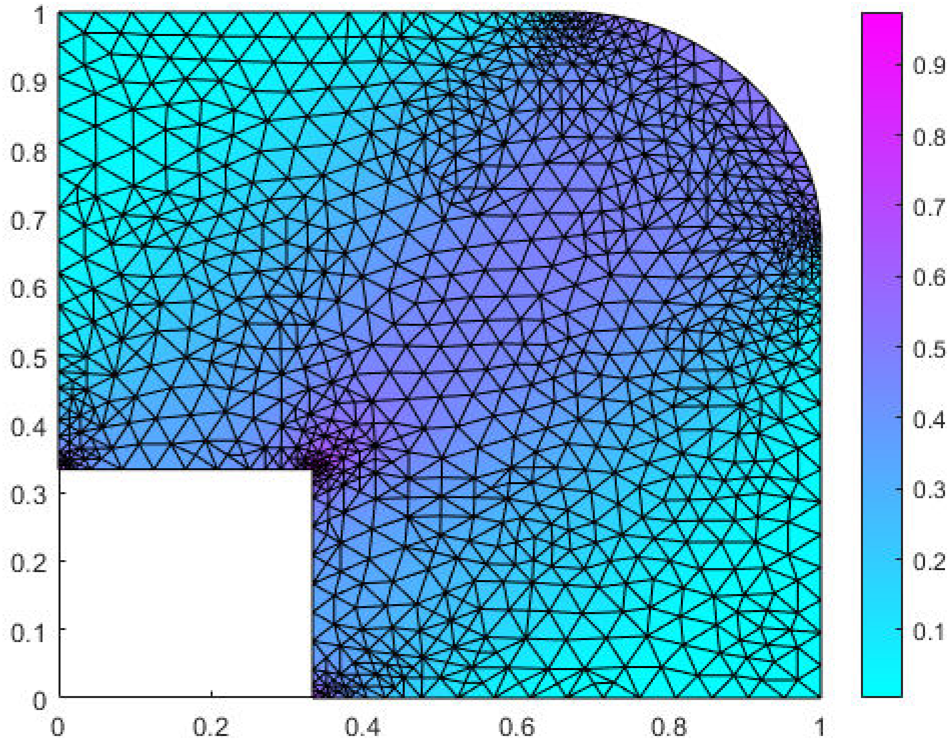
Plot the result.

```
pdeplot(p,e,t,'XYData',sx,'Mesh','on')
```



Find and plot the von Mises effective stress.

```
vM = pdesmech(p,t,c,u,'tensor','vonmises');
pdeplot(p,e,t,'XYData',vM,'Mesh','on')
```



Input Arguments

p — Mesh nodes
matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh elements
matrix

Mesh elements, specified as a 4-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

c — PDE coefficient
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

PDE coefficient, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function.

Data Types: `double` | `char` | `string` | `function_handle`

u — PDE solution
vector

PDE solution, specified as a column vector of `2Np` elements, where `Np` is the number of nodes in the mesh. The first `Np` elements of `u` represent the *x*-displacement, and the next `Np` elements represent *y*-displacement.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `vM = pdesmech(p,t,c,u,'tensor','ux');`

`tensor` — Tensor expression

`'vonmises'` (default) | `'ux'` | `'uy'` | `'vx'` | `'vy'` | `'exx'` | `'eyy'` | `'exy'` | `'sxx'` | `'syy'` | `'sxy'` | `'e1'` | `'e2'` | `'s1'` | `'s2'`

Tensor expression, specified as one of these values:

- `'vonmises'` – 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`.
- `'ux'` – *x*-gradient of *u*-displacement
- `'uy'` – *y*-gradient of *u*-displacement
- `'vx'` – *x*-gradient of *v*-displacement
- `'vy'` – *y*-gradient of *v*-displacement
- `'exx'` – Normal strain along *x*-direction
- `'eyy'` – Normal strain along *y*-direction
- `'exy'` – Shear strain
- `'sxx'` – Normal stress along *x*-direction
- `'syy'` – Normal stress along *y*-direction
- `'sxy'` – Shear stress
- `'e1'` – First principal strain
- `'e2'` – Second principal strain
- `'s1'` – First principal stress
- `'s2'` – Second principal stress

Data Types: `char` | `string`

`application` — Plane stress or plane strain application

`'ps'` (default) | `'pn'`

Plane stress or plane strain application, specified as `'ps'` or `'pn'`.

Data Types: `char` | `string`

`nu` — Poisson's ratio
0.3 (default) | number | vector | character vector | string scalar

Poisson's ratio, specified as a number, vector, character vector, or string scalar. The `pdesmech` function uses the value of Poisson's ratio to calculate shear stresses and strains, and the von Mises effective stress in the plane strain mode. Specify a scalar if the value is constant over the entire geometry. Otherwise, specify a row vector whose length is equal to the number of elements, a character vector, or a string scalar in coefficient form.

Output Arguments

`structMechT` — Resulting tensor
vector

Resulting tensor, returned as a vector. Depending on the value of the `tensor` input argument, the result is one of these values:

- von Mises effective stress
- *x*- or *y*-component of displacement gradients
- normal stress or strain
- shear stress or strain
- first or second principal stress
- first or second principal strain

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2016a

`pdesmech` is not recommended. Use the **PDE Modeler** app instead. There are no plans to remove `pdesmech`.

Starting in R2016a, use the **PDE Modeler** app to calculate stresses and strains for structural mechanics applications with plane stress or plane strain conditions. For example, see “von Mises Effective Stress and Displacements: PDE Modeler App” on page 3-3. Alternatively, starting in R2017b, you can use the recommended structural mechanics workflow. For details, see “Structural Mechanics”.

See Also

PDE Modeler

Topics

“Structural Mechanics”

PDESolverOptions Properties

Algorithm options for solvers

Description

A `PDESolverOptions` object contains options used by the solvers when solving a structural, thermal, or general PDE problem specified as a `StructuralModel`, `ThermalModel`, or `PDEModel` object, respectively. `StructuralModel`, `ThermalModel`, and `PDEModel` objects contain a `PDESolverOptions` object in their `SolverOptions` property.

Solvers for structural modal analysis problems and reduced-order modeling use the Lanczos algorithm.

Properties

Statistics and Convergence Report

`ReportStatistics` — Flag to display internal solver statistics and convergence report during the solution process
`'off'` (default) | `'on'`

Flag to display the internal solver statistics and the convergence report during the solution process, specified as `'off'` or `'on'`.

Example: `model.SolverOptions.ReportStatistics = 'on'`

Data Types: `char`

ODE Solver

`AbsoluteTolerance` — Absolute tolerance for internal ODE solver
1.0000e-06 (default) | positive number

Absolute tolerance for the internal ODE solver, specified as a positive 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 number

Relative tolerance for the internal ODE solver, specified as a positive 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`

Nonlinear Solver

`ResidualTolerance` — Acceptable residual tolerance for internal nonlinear solver
1.0000e-04 (default) | positive number

Acceptable residual tolerance for the internal nonlinear solver, specified as a positive 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 for internal nonlinear solver
25 (default) | positive integer

Maximal number of Gauss-Newton iterations for the internal nonlinear solver, specified as a positive integer.

Example: `model.SolverOptions.MaxIterations = 30`

Data Types: `double`

`MinStep` — Minimum damping of search direction for internal nonlinear solver
1.5259e-05 (default) | positive number

Minimum damping of the search direction for the internal nonlinear solver, specified as a positive number. For details, see "Nonlinear Solver Algorithm" on page 5-1018.

Example: `model.SolverOptions.MinStep = 1.5259e-7`

Data Types: `double`

`ResidualNorm` — Type of norm for computing residual for internal nonlinear solver
`Inf` (default) | `-Inf` | positive number | `'energy'`

Type of norm for computing the residual for the internal nonlinear solver, specified as `Inf`, `-Inf`, a positive number, or `'energy'`.

The infinity norms of a vector are

$$\|\rho\|_{\infty} = \max_i(|\rho(i)|)$$

$$\|\rho\|_{-\infty} = \min_i(|\rho(i)|)$$

The L^p-norm of a vector ρ that has `N` elements is

$$\|\rho\|_p = \frac{\left[\sum_{k=1}^{N} |\rho_k|^p\right]^{\frac{1}{p}}}{N^{\frac{1}{p}}}$$

The energy norm of a vector ρ is

$$\|\rho\| = \rho^T K \rho$$

Here, K is the combined stiffness matrix defined in "Nonlinear Solver Algorithm" on page 5-1018.

Example: `model.SolverOptions.ResidualNorm = 'energy'`

Data Types: `double` | `char`

Lanczos Solver

`MaxShift` — Maximum number of Lanczos shifts

100 (default) | positive integer

Maximum number of Lanczos shifts, specified as a positive integer. Increase this value when computing a large number of eigenpairs.

Example: `model.SolverOptions.MaxShift = 500`

Data Types: `double`

`BlockSize` — Block size for block Lanczos recurrence

ranges from 7 to 25 (default) | positive integer

Block size for block Lanczos recurrence, specified as a positive integer. The default number ranges from 7 to 25, depending on the size of the stiffness matrix `K`.

Example: `model.SolverOptions.BlockSize = 20`

Data Types: `double`

Algorithms

Nonlinear Solver Algorithm

The residual equation of a nonlinear PDE is as follows:

$$r(u) = -\nabla \cdot (c(u)\nabla(u)) + a(u)u - f(u) = 0$$

To obtain a discretized residual equation, apply the finite element method (FEM) to a partial differential equation as described in "Finite Element Method Basics" on page 1-11:

$$\rho(U) = K(U)U - F(U) = 0$$

The nonlinear solver uses a Gauss-Newton iteration scheme applied to the finite element matrices. Use a Taylor series expansion to obtain the linearized system for the residual:

$$\rho\left(U^{n+1}\right) \cong \rho\left(U^{n}\right) + \frac{\partial\rho\left(U^{n}\right)}{\partial U}\left(U^{n+1} - U^{n}\right) + \ldots = 0$$

Neglecting the higher-order terms, write the linearized system of equations as

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

The descent direction for the residual is

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

The Gauss-Newton iteration minimizes the residual, that is, the solution of $\min_U \|\rho(U)\|$, using the equation

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

Here, α ≤ 1 is a positive number, that must be set as large as possible so that the step has a reasonable descent. For a sufficiently small α,

$$\left\|\rho\left(U^n + \alpha p_n\right)\right\| < \left\|\rho\left(U^n\right)\right\|$$

For the Gauss-Newton algorithm to converge, U^0 must be close enough to the solution. The first guess is often outside the region of convergence. The Armijo-Goldstein line search (a damping strategy for choosing α) helps to improve convergence from bad initial guesses. This method chooses the largest damping coefficient α out of the sequence 1, 1/2, 1/4, . . . such that the following inequality holds:

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

Using the Armijo-Goldstein line search guarantees a reduction of the residual norm by at least $1 - \alpha/2$. Each step of the line-search algorithm must evaluate the residual $\left\|\rho\left(U^n + \alpha p_n\right)\right\|$.

With this strategy, when U^n approaches the solution, $\alpha \to 1$, thus, the convergence rate increases.

Version History

Introduced in R2016a

See Also

`PDEModel` | `solvepde` | `solvepdeeig`

pdesurf

(To be removed) Surface plot of PDE node or triangle data

Note `pdesurf` will be removed in a future release. Use `pdeplot` instead. For more information, see "Compatibility Considerations".

Syntax

```
pdesurf(p,t,u)
h = pdesurf(p,t,u)
```

Description

`pdesurf(p,t,u)` plots a 3-D surface using PDE node or triangle data as a height for a 2-D problem. The `p` and `t` arguments specify the geometry of the PDE problem.

If `u` is a column vector, `pdesurf` treats it as node data and uses continuous style and interpolated shading. If `u` is a row vector, `pdesurf` treats it as triangle data and uses discontinuous style and flat shading.

`h = pdesurf(p,t,u)` returns handles to the drawn axes objects.

Examples

Surface Plot of PDE Solution

Plot the solution of the equation $-\Delta u = 1$ on the L-shaped membrane using the `pdesurf` function.

First, create a `[p,e,t]` mesh on the L-shaped membrane.

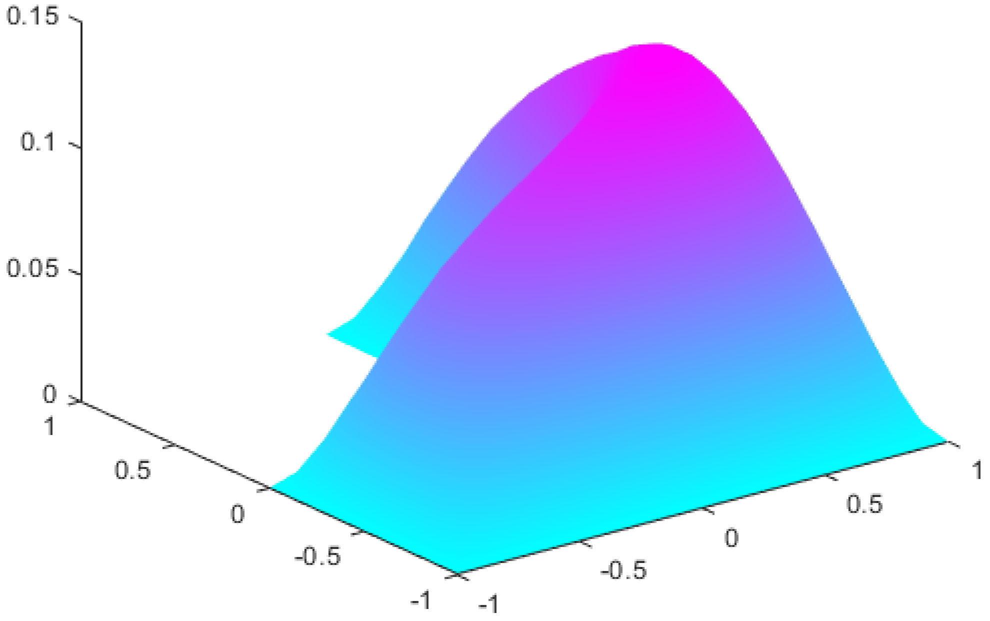
```
[p,e,t] = initmesh('lshapeg');
```

Solve the equation using the Dirichlet boundary conditions $u = 0$ on $\partial\Omega$.

```
u = assempde('lshapeb',p,e,t,1,0,1);
```

Plot the solution at the mesh nodes. When plotting the solution at the nodes, the function uses continuous style and interpolated shading.

```
pdesurf(p,t,u)
```

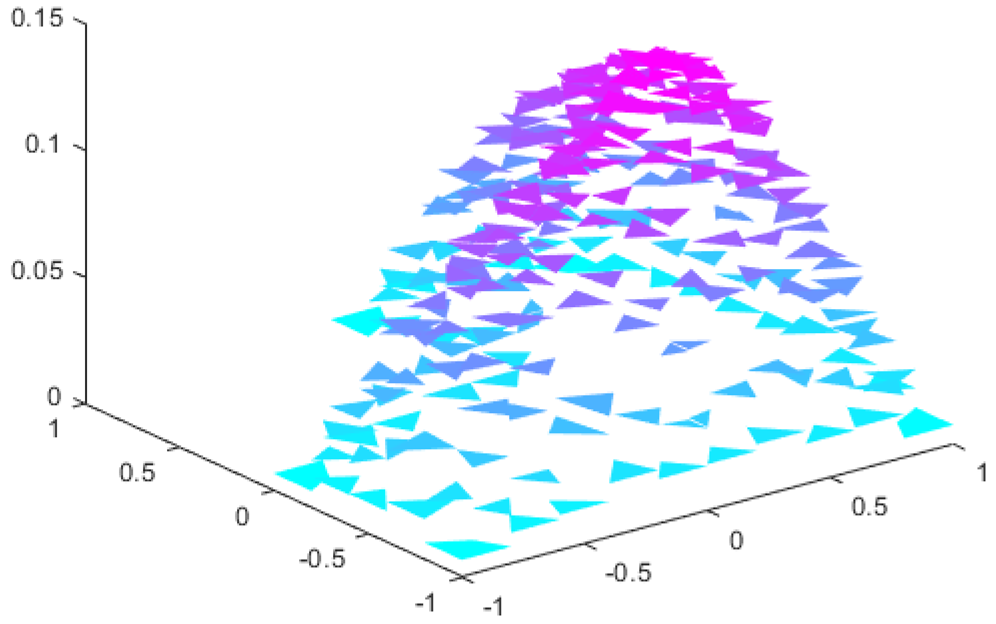


Interpolate the solution from the mesh nodes to the triangle midpoints.

```
ut = pdeintrp(p,t,u);
```

Plot the interpolated solution. When plotting the solution as a triangle data, the function uses discontinuous style and flat shading.

```
pdesurf(p,t,ut)
```



Input Arguments

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 details on the mesh data representation, see `initmesh`.

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 details on the mesh data representation, see `initmesh`.

Data Types: `double`

u — PDE solution

vector

PDE solution, specified as a vector.

The `pdesurf` function treats a column vector as node data and uses continuous style and interpolated shading. The function treats a row vector as triangle data and uses discontinuous style and flat shading.

Data Types: `double`

Output Arguments

h — Handles to graphics objects
vector

Handles to graphics objects, returned as a vector.

Tips

- For more control over a surface plot, use the `pdeplot` function.

Version History

Introduced before R2006a

`pdesurf` will be removed

Not recommended starting in R2022a

`pdesurf` will be removed. Use `pdeplot` instead.

To update your code for plotting a solution `u` at nodes, change instances of the function call `pdesurf(p,t,u)` to the function call:

```
pdeplot(p,[],t,'XYData',u,'ZData',u,'ColorBar','off')
```

To update your code for plotting a solution `ut` at midpoints of mesh triangles, change instances of the function call `pdecont(p,t,ut)` to the function call:

```
pdeplot(p,[],t,'XYData',ut,'XStyle','flat', ...
               'ZData',ut,'ZStyle','discontinuos', ...
               'ColorBar','off')
```

`pdeplot` gives you more control over your surface plot.

Note that the legacy workflow that uses the `[p,e,t]` mesh is not recommended. New features might not be compatible with this legacy workflow. For description of the mesh data in the recommended workflow, see “Mesh Data” on page 2-175.

See Also

`pdemesh` | `pdeplot`

PDE Modeler

Create complex 2-D geometries by drawing, overlapping, and rotating basic shapes

Description

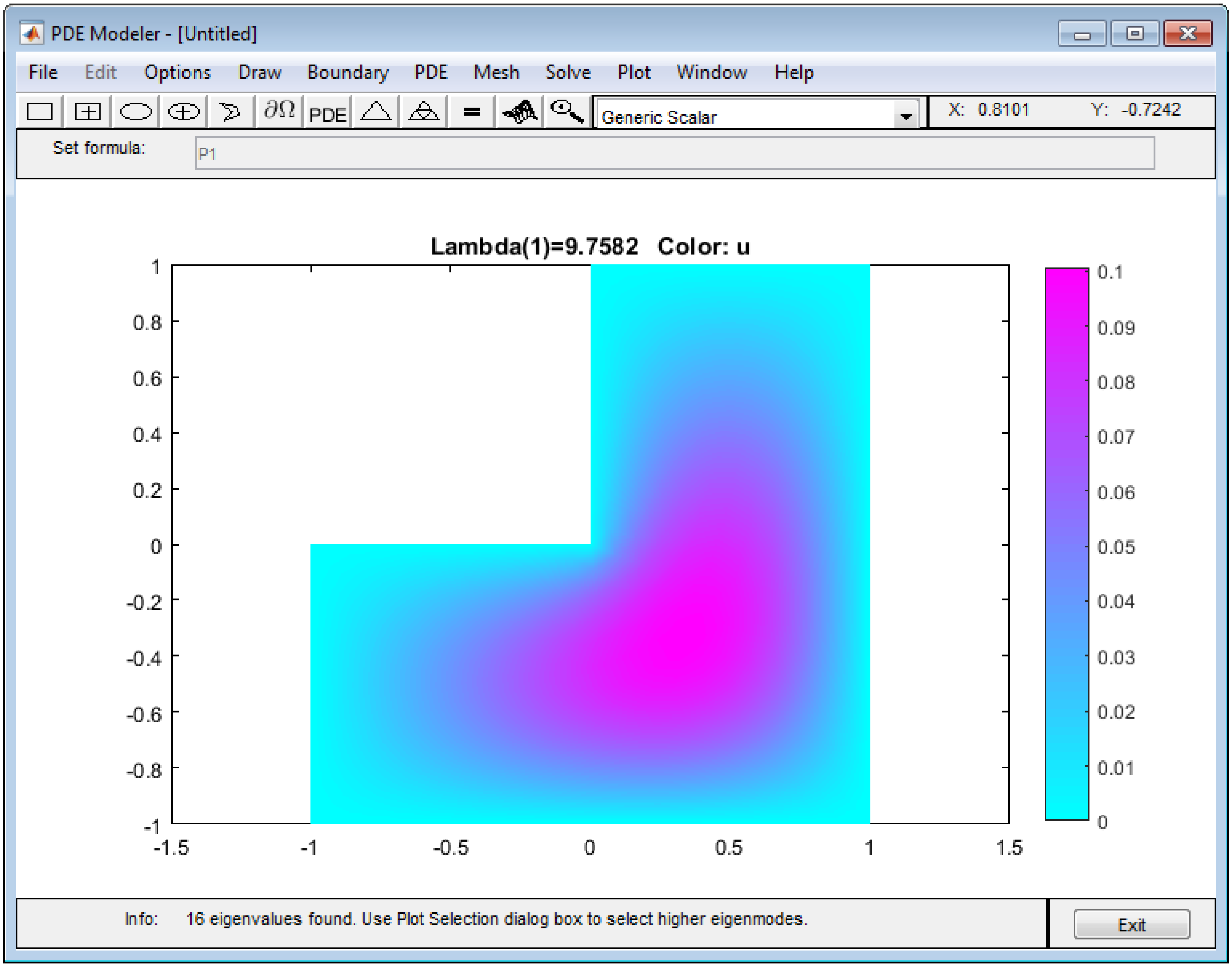
The **PDE Modeler** app provides an interactive interface for solving 2-D geometry problems. Using the app, you can create complex geometries by drawing, overlapping, and rotating basic shapes, such as circles, polygons and so on. The app also includes preset modes for applications, such as electrostatics, 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 a 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.



Info: 16 eigenvalues found. Use Plot Selection dialog box to select higher eigenmodes.

Open the PDE Modeler App

- MATLAB Toolstrip: On the **Apps** tab, under **Math, Statistics and Optimization**, click the app icon.
- MATLAB command prompt: Enter `pdeModeler`.

Examples

- "Solve 2-D PDEs Using the PDE Modeler App" on page 1-5
- "Open the PDE Modeler App" on page 4-2
- "2-D Geometry Creation in PDE Modeler App" on page 4-3
- "Specify Boundary Conditions in the PDE Modeler App" on page 4-12
- "Specify Coefficients in PDE Modeler App" on page 4-14

- “Specify Mesh Parameters in the PDE Modeler App” on page 4-24
- “Adjust Solve Parameters in the PDE Modeler App” on page 4-26
- “Plot the Solution in the PDE Modeler App” on page 4-31
- “von Mises Effective Stress and Displacements: PDE Modeler App” on page 3-3
- “Heat Transfer in Block with Cavity: PDE Modeler App” on page 3-254
- “Heat Distribution in Circular Cylindrical Rod: PDE Modeler App” on page 3-295
- “Heat Transfer Between Two Squares Made of Different Materials: PDE Modeler App” on page 3-195
- “Poisson's Equation on Unit Disk: PDE Modeler App” on page 3-218
- “Poisson’s Equation with Complex 2-D Geometry: PDE Modeler App” on page 1-7
- “Electrostatic Potential in Air-Filled Frame: PDE Modeler App” on page 3-133
- “Magnetic Field in Two-Pole Electric Motor: PDE Modeler App” on page 3-166
- “Wave Equation on Square Domain: PDE Modeler App” on page 3-302
- “Scattering Problem: PDE Modeler App” on page 3-237
- “Skin Effect in Copper Wire with Circular Cross Section: PDE Modeler App” on page 3-184
- “Poisson's Equation on Unit Disk: PDE Modeler App” on page 3-218
- “Minimal Surface Problem: PDE Modeler App” on page 3-247
- “Current Density Between Two Metallic Conductors: PDE Modeler App” on page 3-192
- “L-Shaped Membrane with Rounded Corner: PDE Modeler App” on page 3-314
- “Eigenvalues and Eigenmodes of Square: PDE Modeler App” on page 3-322
- “Eigenvalues and Eigenmodes of L-Shaped Membrane: PDE Modeler App” on page 3-311

Programmatic Use

`pdeModeler` opens the PDE Modeler app or brings focus to the app if it is already open.

`pdecirc(xc,yc,r)` opens the PDE Modeler app and draws a circle with center in `(xc,yc)` and radius `r`.

`pdeellip(xc,yc,a,b,phi)` opens the PDE Modeler 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 Modeler app and draws a polygon with corner coordinates defined by `x` and `y`.

`pderect([xmin xmax ymin ymax])` opens the PDE Modeler app and draws a rectangle with corner coordinates defined by `[xmin xmax ymin ymax]`.

Version History

Introduced before R2006a

`pdetool` has been changed to `pdeModeler`

Behavior change in future release

The function name `pdetool` has been changed to `pdeModeler`.

Calling `pdetool` or opening the app via the Apps tab does not overwrite existing information in the app

Behavior change in future release

In previous releases, starting the PDE Modeler app and then reopening the app by calling `pdetool` or by using the **Apps** tab overwrites any existing information in the app. Now, reopening the PDE Modeler app brings focus to the app window. To overwrite the existing information in the PDE Modeler app, select **File** > **New**.

Meshing algorithm performance and robustness enhancements

Performance change in R2013a

The mesher provides an enhancement option for increased meshing speed and robustness. Choose the enhanced algorithm by selecting **Mesh** > **Parameters** and setting `Mesher version` to `R2013a`.

Graphics export from `pdetool`

You can now save the current figure in a variety of image formats by using the **File** > **Export Image** menu.

See Also

Functions

`pdecirc` | `pdeellip` | `pdepoly` | `pderect`

Topics

“Solve 2-D PDEs Using the PDE Modeler App” on page 1-5
“Open the PDE Modeler App” on page 4-2
“2-D Geometry Creation in PDE Modeler App” on page 4-3
“Specify Boundary Conditions in the PDE Modeler App” on page 4-12
“Specify Coefficients in PDE Modeler App” on page 4-14
“Specify Mesh Parameters in the PDE Modeler App” on page 4-24
“Adjust Solve Parameters in the PDE Modeler App” on page 4-26
“Plot the Solution in the PDE Modeler App” on page 4-31
“von Mises Effective Stress and Displacements: PDE Modeler App” on page 3-3
“Heat Transfer in Block with Cavity: PDE Modeler App” on page 3-254
“Heat Distribution in Circular Cylindrical Rod: PDE Modeler App” on page 3-295
“Heat Transfer Between Two Squares Made of Different Materials: PDE Modeler App” on page 3-195
“Poisson's Equation on Unit Disk: PDE Modeler App” on page 3-218
“Poisson’s Equation with Complex 2-D Geometry: PDE Modeler App” on page 1-7
“Electrostatic Potential in Air-Filled Frame: PDE Modeler App” on page 3-133
“Magnetic Field in Two-Pole Electric Motor: PDE Modeler App” on page 3-166
“Wave Equation on Square Domain: PDE Modeler App” on page 3-302
“Scattering Problem: PDE Modeler App” on page 3-237
“Skin Effect in Copper Wire with Circular Cross Section: PDE Modeler App” on page 3-184
“Poisson's Equation on Unit Disk: PDE Modeler App” on page 3-218
“Minimal Surface Problem: PDE Modeler App” on page 3-247
“Current Density Between Two Metallic Conductors: PDE Modeler App” on page 3-192
“L-Shaped Membrane with Rounded Corner: PDE Modeler App” on page 3-314
“Eigenvalues and Eigenmodes of Square: PDE Modeler App” on page 3-322

"Eigenvalues and Eigenmodes of L-Shaped Membrane: PDE Modeler App" on page 3-311

pdetrg

(Not recommended) Triangle geometry data

Note `pdetrg` is not recommended. Use `area` instead.

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 areas of individual mesh triangles as a vector `ar` and half of the negative cotangent of each angle as vectors `a1,a2,a3`.

`[ar,g1x,g1y,g2x,g2y,g3x,g3y] = pdetrg(p,t)` returns the areas of individual mesh triangles as a vector `ar` and the gradient components of the triangle base functions as vectors `g1x,g1y,g2x,g2y,g3x,g3y`.

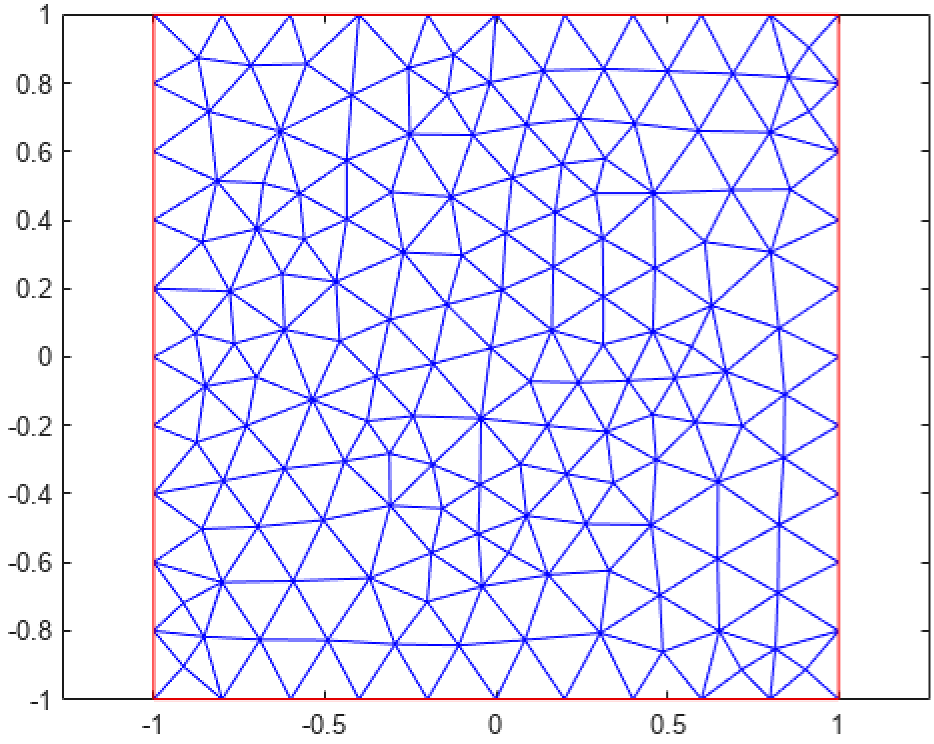
Examples

Areas of Individual Mesh Elements

Find the area of each element of a 2-D mesh. In addition, return half of the negative cotangent of each angle and the gradient components of the triangle base functions for each element.

Generate and plot a mesh for a unit square geometry.

```
[p,e,t] = initmesh(@squareg);
pdemesh(p,e,t)
```



Compute the area of each individual element of the mesh and half of the negative cotangent of each angle. Display the first 5 elements for each result.

```
[ar,a1,a2,a3] = pdetrg(p,t);
ar(1:5)
```

```
ans = 1×5

    0.0126    0.0148    0.0144    0.0156    0.0118
```

```
a1(1:5)
```

```
ans = 1×5

   -0.2819   -0.3905   -0.5332   -0.1812   -0.5237
```

```
a2(1:5)
```

```
ans = 1×5

   -0.5124   -0.2842   -0.1613   -0.4616   -0.3267
```

```
a3(1:5)
```

```
ans = 1×5
```

```
   -0.1329   -0.2061   -0.2362   -0.2588   -0.0928
```

Find the area of the smallest and the largest element of the mesh.

```
min(ar)
```

```
ans = 0.0061
```

```
max(ar)
```

```
ans = 0.0216
```

Use the syntax with seven output arguments to compute the gradient components of the triangle base functions for each element.

```
[ar,g1x,g1y,g2x,g2y,g3x,g3y] = pdetrg(p,t);
```

Input Arguments

`p` — Mesh nodes

matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

`t` — Mesh elements

matrix

Mesh elements, specified as a `4`-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

Output Arguments

`ar` — Areas of individual elements

row vector of positive numbers

Areas of individual elements, returned as a row vector of positive numbers.

`a1,a2,a3` — Half of negative cotangent of each angle

three row vectors of numbers

Half of the negative cotangent of each angle, returned as three row vectors of numbers.

`g1x,g1y,g2x,g2y,g3x,g3y` — Gradient components of triangle base functions

six row vectors of numbers

Gradient components of the triangle base functions, returned as six row vectors of numbers.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2018a

`pdetrg` is not recommended. Use `area` instead. There are no plans to remove `pdetrg`.

See Also

`area` | `pdetriq`

Topics

“Mesh Data as [p,e,t] Triples” on page 2-172

pdetriq

(Not recommended) Triangle quality measure

Note `pdetriq` is not recommended. Use `meshQuality` instead.

Syntax

```
q = pdetriq(p,t)
```

Description

`q = pdetriq(p,t)` returns a row vector of numbers from 0 through 1 representing the triangle quality of all the elements of the `[p,e,t]` mesh.

`pdetriq` evaluates the quality of a triangle as

$$q = \frac{4a\sqrt{3}}{h_1^2 + h_2^2 + h_3^2}$$

where a is the area and h_1, h_2, and h_3 are the lengths of the edges of the triangle.

The value 0 corresponds to a degenerate triangle with zero area. The value 1 corresponds to a triangle with $h_1 = h_2 = h_3$.

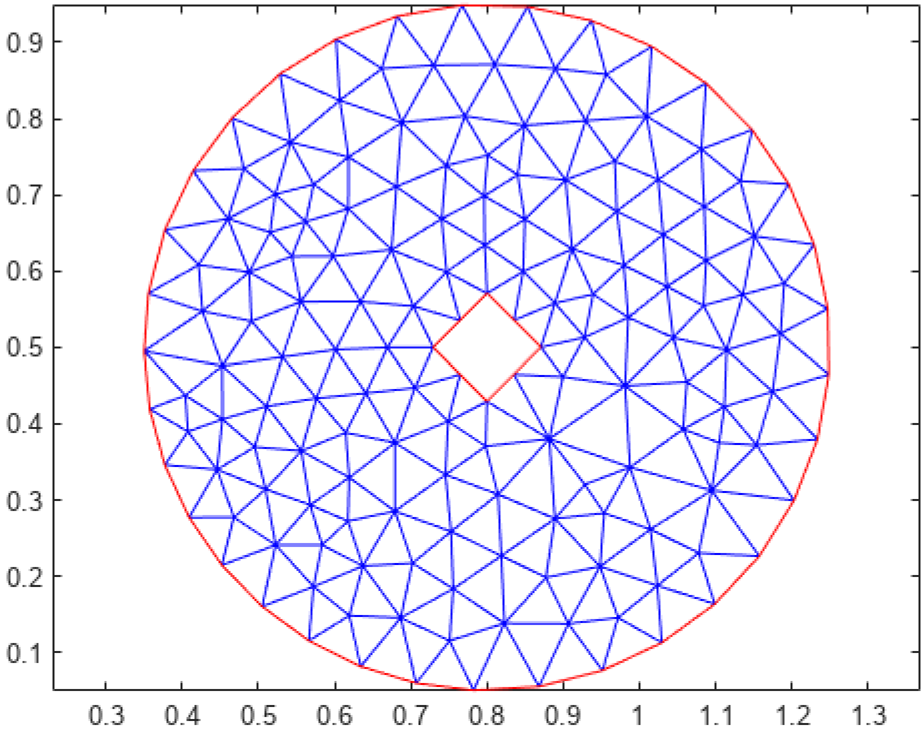
Examples

Mesh Element Quality for `[p,e,t]` Data

Evaluate the quality for each triangle of a `[p,e,t]` mesh.

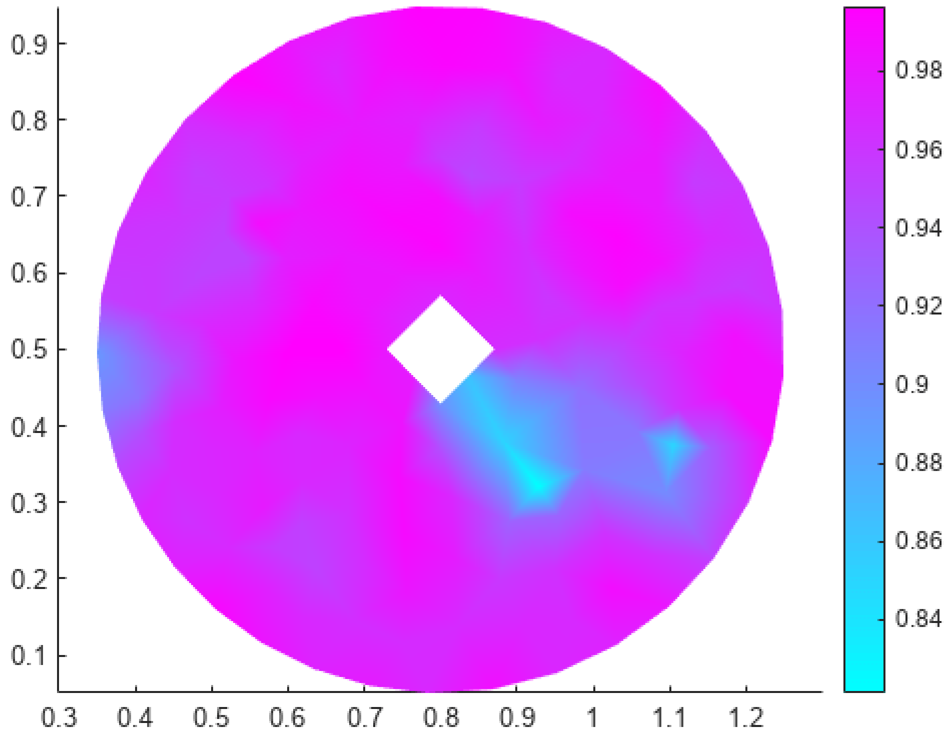
Generate and plot a mesh for the geometry consisting of a circle with a diamond hole.

```
[p,e,t] = initmesh(@scatterg);
pdemesh(p,e,t)
axis equal
```



Evaluate the triangle quality for each mesh triangle. Plot the resulting quality values.

```
q = pdetriq(p,t);
figure
pdeplot(p,e,t,'XYData',q);
axis equal
```



Input Arguments

p — Mesh nodes

matrix

Mesh nodes, specified as a 2-by-`Np` matrix of nodes (points), where `Np` is the number of nodes in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh elements

matrix

Mesh elements, specified as a 4-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

Output Arguments

q — Triangle quality

row vector

Triangle quality, returned as a row vector of numbers from 0 through 1.

Version History

Introduced before R2006a

Not recommended

Not recommended starting in R2018a

`pdetriq` is not recommended. Use `meshQuality` instead. There are no plans to remove `pdetriq`.

References

[1] Bank, Randolph E. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, User's Guide 6.0*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1990.

See Also

`pdetrg` | `meshQuality`

Topics

“Mesh Data as [p,e,t] Triples” on page 2-172

pdeviz

Create and plot PDE visualization object

Syntax

```
pdeviz(MeshData,NodalData)
pdeviz(MeshData)
pdeviz( ___ ,Name,Value)
pdeviz(figure, ___ )
V = pdeviz( ___ )
```

Description

`pdeviz(MeshData,NodalData)` creates a `PDEVisualization` object and plots the data at the mesh nodes as a surface plot. For details, see PDEVisualization Properties.

`pdeviz(MeshData)` creates a `PDEVisualization` object and plots the mesh.

`pdeviz( ___ ,Name,Value)` customizes the plot appearance using one or more `Name,Value` arguments. Use name-value arguments with any combination of arguments from the previous syntaxes.

`pdeviz(figure, ___ )` specifies the graphics container for the `PDEVisualization` object. For example, you can plot the object in Figure 3 by specifying `pdeviz(figure(3), ___)`.

`V = pdeviz( ___ )` returns a handle to the `PDEVisualization` object, using any of the previous syntaxes.

Examples

Mesh and Solution of Structural Model

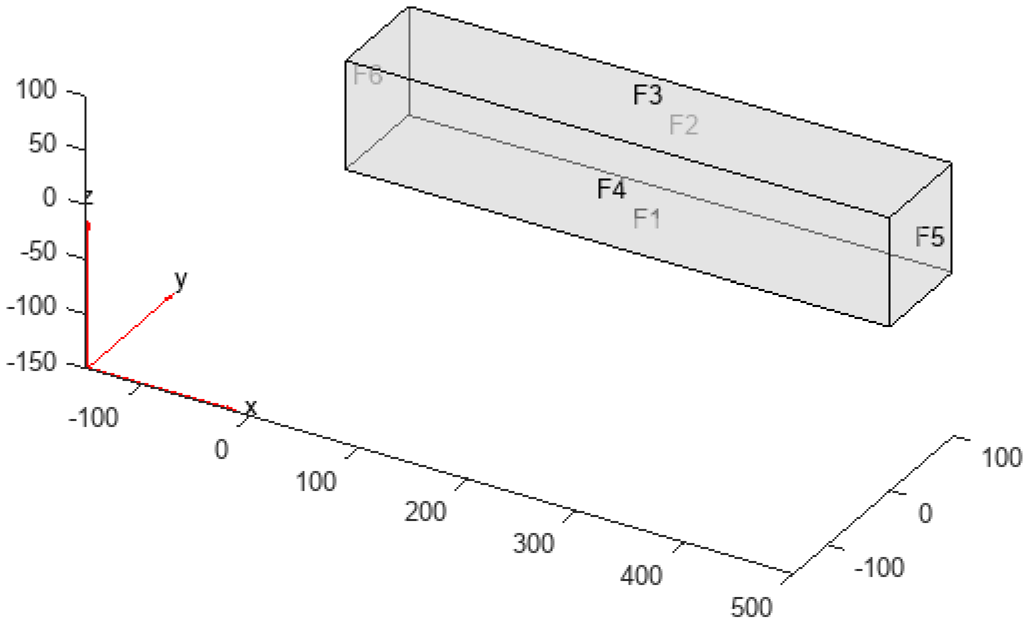
Use the `pdeviz` function to create a PDE visualization object and plot it. Change the properties of this object to interact with the resulting plot.

Create a structural analysis model for a 3-D problem.

```
structuralmodel = createpde("structural","static-solid");
```

Import the beam geometry and plot it.

```
importGeometry(structuralmodel,"SquareBeam.stl");
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralmodel,"PoissonsRatio",0.3, ...
                                     "YoungsModulus",210E3);
```

Specify that face 6 is a fixed boundary.

```
structuralBC(structuralmodel,"Face",6,"Constraint","fixed");
```

Specify the surface traction for face 5.

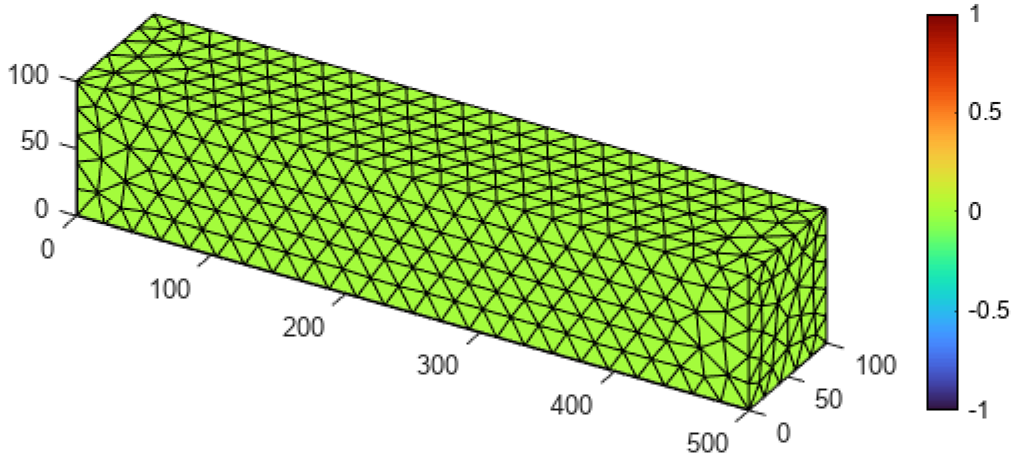
```
structuralBoundaryLoad(structuralmodel,"Face",5, ...
                      "SurfaceTraction",[0;0;-2]);
```

Generate a mesh and solve the problem.

```
msh = generateMesh(structuralmodel);
structuralresults = solve(structuralmodel);
```

Call `pdeviz` with only the mesh data. This call creates a `PDEVisualization` object and plots the mesh.

```
figure
v = pdeviz(msh)
```



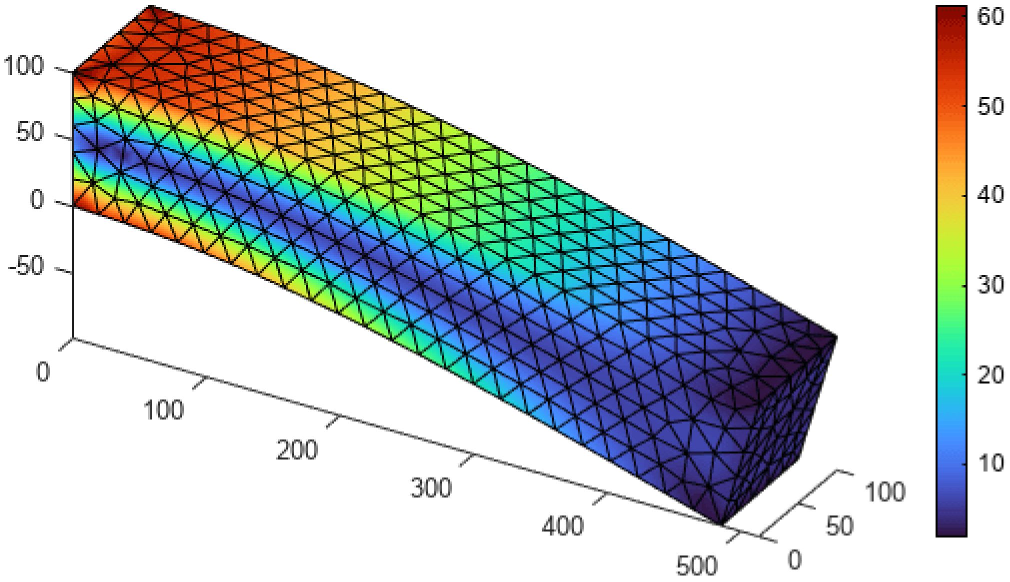
```
v = 
  PDEVisualization with properties:

         MeshData: [1x1 FEMesh]
        NodalData: [0x1 double]
      MeshVisible: on
     Transparency: 1
         Position: [0.1300 0.1100 0.6705 0.8150]
            Units: 'normalized'

  Show all properties
```

Update the plot by adding the von Mises stress as the `NodalData` property of the `PDEVisualization` object v. The plot now shows the von Mises stress and the mesh.

```
figure
v.NodalData = structuralresults.VonMisesStress;
```

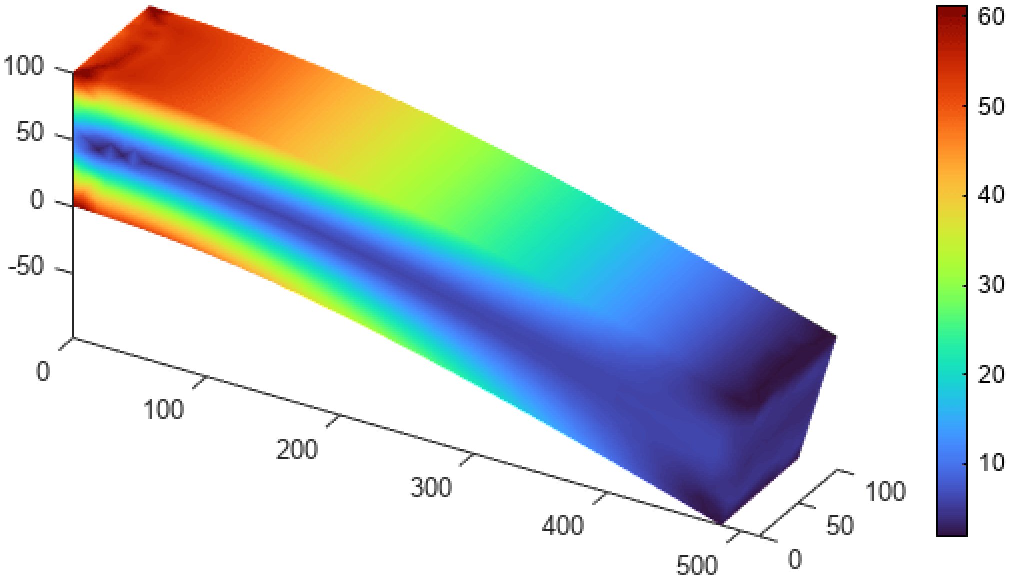
Update the plot by adding the displacement as the `DeformationData` property of the `PDEVisualization` object v. The plot shows the deformed shape with the von Mises stress.

```
figure
v.DeformationData = structuralresults.Displacement;
```



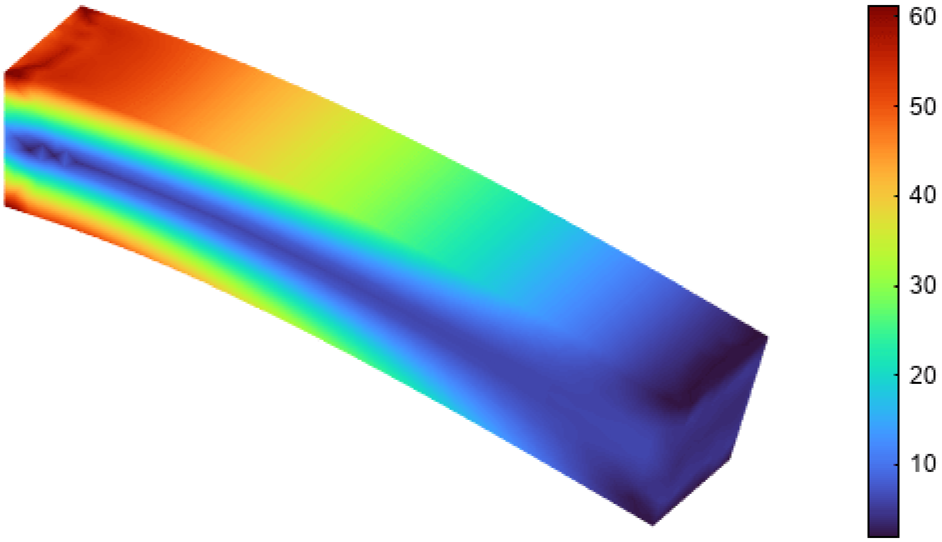
Update the plot to hide the mesh.

```
figure
v.MeshVisible = "off";
```



Update the plot to hide the axes.

```
figure
v.AxesVisible = "off";
```



Input Arguments

`MeshData` — Finite element mesh

`FEMesh` object

Finite element mesh, specified as an FEMesh object.

`NodalData` — Data at mesh nodes

column vector

Data at mesh nodes, specified as a column vector.

Data Types: `double`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example:
```
pdeviz(model.Mesh,results.NodalSolution,"MeshVisible","on","DeformationData",
results.Displacement)
```

`Transparency` — Surface transparency

`1` (default) | real number from `0` through `1`

Surface transparency, specified as a real number from `0` through `1`. The default value `1` indicates no transparency. The value `0` indicates complete transparency.

Data Types: `double`

`MeshVisible` — Toggle to show mesh

`"on"` | `"off"` | on/off logical value

Toggle to show mesh, specified as `"on"` or `"off"`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `"on"` is equivalent to `true`, and `"off"` is equivalent to `false`. Thus, you can use the value of this argument as a logical value.

When plotting only the mesh, the default is `"on"`. Otherwise, the default is `"off"`.

Data Types: `char` | `string`

`DeformationData` — Mesh deformation data

`FEStruct` object | matrix | structure array

Mesh deformation data, specified as one of the following:

- An `FEStruct` object with the properties `ux`, `uy`, and, for a 3-D geometry, `uz`
- A structure array with the fields `ux`, `uy`, and, for a 3-D geometry, `uz`
- A matrix with either two columns for a 2-D geometry or three columns for a 3-D geometry

`DeformationScaleFactor` — Level of mesh deformation

nonnegative number

Level of mesh deformation, specified as a nonnegative number. Use this name-value argument together with `DeformationData`.

`pdeviz` computes the default value of `DeformationScaleFactor` based on the mesh and the value of `DeformationData`.

Data Types: `double`

`AxesVisible` — Toggle to hide or show axes

`"on"` (default) | `"off"` | on/off logical value

Toggle to hide or show axes, specified as `"on"` or `"off"`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `"on"` is equivalent to `true`, and `"off"` is equivalent to `false`. Thus, you can use the value of this argument as a logical value.

`AxesColor` — Background color

RGB triplet | hexadecimal color code | `"r"` | `"g"` | `"b"`

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

`ColorbarVisible` — Colorbar visibility

`"on"` (default) | on/off logical value

Colorbar visibility, specified as `"on"` or `"off"`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `"on"` is equivalent to `true`, and `"off"` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

`ColorLimits` — Color limits
two-element vector

Color limits, specified as a two-element vector of the form `[min max]`. The color limits indicate the color data values that map to the first and last colors in the colormap.

`Title` — Chart title
character vector | cell array of character vectors | string array | categorical array

Chart title, specified as a character vector, cell array of character vectors, string array, or categorical array.

Example: `"My Title Text"`

To create a multi-line title, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text.

Example: `{"My","Title"};`

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

If you create the chart using tabular data, the default chart has an autogenerated title. If you do not want a title, specify `""`.

`View` — Azimuth and elevation of view
`[0 90]` (default) | two-element vector of the form `[azimuth elevation]`

Azimuth and elevation of view, specified as a two-element vector of the form `[azimuth elevation]` defined in degree units. Alternatively, use the `view` function to set the view.

Output Arguments

`V` — Visualization container
handle

Visualization container, returned as a handle to the `PDEVisualization` object. For details, see PDEVisualization Properties.

Version History

Introduced in R2021a

See Also

`pdeplot` | `pdeplot3D` | `pdemesh` | PDEVisualization Properties

PDEVisualization Properties

PDE visualization of mesh and nodal results

Description

`PDEVisualization` properties control the appearance and behavior of a `PDEVisualization` object. By changing property values, you can modify certain aspects of the visualization.

Create a `PDEVisualization` object using the `pdeviz` function.

Properties

Displayed Data

`MeshData` — Finite element mesh
`FEMesh` object

Finite element mesh, specified as an FEMesh object.

`NodalData` — Data at mesh nodes
vector

Data at mesh nodes, specified as a vector.

Data Types: `double`

`DeformationData` — Mesh deformation data
`FEStruct` object | matrix | structure array

Mesh deformation data, specified as one of the following:

- An `FEStruct` object with the properties `ux`, `uy`, and, for a 3-D geometry, `uz`
- A structure array with the fields `ux`, `uy`, and, for a 3-D geometry, `uz`
- A matrix with either two columns for a 2-D geometry or three columns for a 3-D geometry

Data Types: `double`

`DeformationScaleFactor` — Level of mesh deformation
nonnegative number

Level of mesh deformation, specified as a nonnegative number.

The toolbox computes the default value of `DeformationScaleFactor` based on the mesh and the value of `DeformationData`.

Example: `v.DeformationScaleFactor = 1000;`

Data Types: `double`

`XLimits`, `YLimits`, `ZLimits` — Axis limits
two-element vector of the form `[min max]`

Axis limits, specified as a two-element vector of the form `[min max]`, where `max` is greater than `min`. You can specify the limits as numeric, categorical, datetime, or duration values. The type of values that you specify must match the type of values along the axis.

You can specify both limits or you can specify one limit and let the axes automatically calculate the other. For an automatically calculated minimum or maximum limit, use `-Inf` or `Inf`, respectively.

Example: `ax.XLim = [0 10]`

Example: `ax.YLim = [-Inf 10]`

Example: `ax.ZLim = [0 Inf]`

Color and Styling

`ColorLimits` — Color limits

two-element vector

Color limits, specified as a two-element vector of the form `[min max]`. The color limits indicate the color data values that map to the first and last colors in the colormap.

Example: `v.ColorLimits = [0 10];`

`ColorbarVisible` — Colorbar visibility

`'on'` (default) | on/off logical value

Colorbar visibility, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

Example: `v.ColorbarVisible = 'off';`

`MeshVisible` — Mesh visibility

`'on'` | `'off'` | on/off logical value

Mesh visibility, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

Example: `v.MeshVisible = 'on';`

Data Types: `char` | `string`

`Transparency` — Surface transparency

`1` (default) | real number from `0` through `1`

Surface transparency, specified as a real number from `0` through `1`. The default value `1` indicates no transparency. The value `0` indicates complete transparency.

When you use the `Transparency` argument for solution plots, the plot colors might not match the color bar values. Always use a fully opaque plot to estimate the solution values.

Data Types: `double`

`AxesVisible` — Toggle to hide or show axes

`'on'` (default) | `'off'` | on/off logical value

Toggle to hide or show axes, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this argument as a logical value.

`AxesColor` — Background color

RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'`

Background color, specified as an RGB triplet, a hexadecimal color code, a color name, or a short name.

Labels

`XLabel`, `YLabel`, `ZLabel` — Text object for axis label

character vector | cell array of character vectors | string array | categorical array

Axis labels, specified as character vectors, cell arrays of character vectors, string arrays, or categorical arrays.

Example: `v.XLabel = 'time';`

`Title` — Chart title

character vector | cell array of character vectors | string array | categorical array

Chart title, specified as a character vector, cell array of character vectors, string array, or categorical array.

Example: `v.Title = 'My Title Text';`

To create a multi-line title, specify a cell array of character vectors or a string array. Each element in the array corresponds to a line of text.

Example: `v.Title = {'My','Title'};`

If you specify the title as a categorical array, MATLAB uses the values in the array, not the categories.

If you create the chart using tabular data, the default chart has an autogenerated title. If you do not want a title, specify `''`.

Interactivity

`Visible` — State of visibility

`'on'` (default) | on/off logical value

State of visibility, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- `'on'` — Display the chart.
- `'off'` — Hide the chart without deleting it. You still can access the properties of chart when it is not visible.

Parent/Child

`Parent` — Parent container

`Figure` object (default) | `Panel` object | `Tab` object | `TiledChartLayout` object

Parent container of the chart, specified as a `Figure`, `Panel`, `Tab`, or `TiledChartLayout` object.

`HandleVisibility` — Visibility of object handle

`'on'` (default) | `'off'` | `'callback'`

Visibility of the object handle in the `Children` property of the parent, specified as one of these values:

- `'on'` — Object handle is always visible.
- `'off'` — Object handle is invisible at all times. This option is useful for preventing unintended changes by another function. Set the `HandleVisibility` to `'off'` to temporarily hide the handle during the execution of that function.
- `'callback'` — Object handle is visible from within callbacks or functions invoked by callbacks, but not from within functions invoked from the command line. This option blocks access to the object at the command line, but permits callback functions to access it.

If the object is not listed in the `Children` property of the parent, then functions that obtain object handles by searching the object hierarchy or querying handle properties cannot return it. Examples of such functions include the `get`, `findobj`, `gca`, `gcf`, `gco`, `newplot`, `cla`, `clf`, and `close` functions.

Hidden object handles are still valid. Set the root `ShowHiddenHandles` property to `'on'` to list all object handles regardless of their `HandleVisibility` property setting.

Position

`OuterPosition` — Chart size and location, including margins

`[left bottom width height]`

Chart size and location, including the margins for decorations such as axis labels and tick marks. Specify this property as a vector of form `[left bottom width height]`. The values are in the units specified by the `Units` property.

- `left` — Distance from the left edge of the parent container to the outer-left edge of the chart that includes the margins. Typically, the parent container is a figure, panel, or tab.
- `bottom` — Distance from the bottom edge of the parent container to the outer-bottom edge of the chart that includes the margins.
- `width` — Width of chart, including the margins.
- `height` — Height of chart, including the margins.

Note Setting this property has no effect when the parent of the chart is a `TiledChartLayout`.

`InnerPosition` — Chart size and location, excluding margins

`[left bottom width height]`

Chart size and location, excluding the margins for decorations such as axis labels and tick marks. Specify this property as a vector of form `[left bottom width height]`. The values are in the units specified by the `Units` property.

- `left` — Distance from the left edge of the parent container to the inner-left edge of the chart that excludes the margins. Typically, the parent container is a figure, panel, or tab.

- `bottom` — Distance from the bottom edge of the parent container to the inner-bottom edge of the chart that excludes the margins.
- `width` — Width of chart, excluding the margins.
- `height` — Height of chart, excluding the margins.

Note Setting this property has no effect when the parent of the chart is a `TiledChartLayout`.

`Position` — Chart size and location, excluding margins

`[left bottom width height]`

Chart size and location, excluding the margins for decorations such as axis labels and tick marks. Specify this property as a vector of form `[left bottom width height]`. This property is equivalent to the `InnerPosition` property.

Note Setting this property has no effect when the parent of the chart is a `TiledChartLayout`.

`PositionConstraint` — Position to hold constant

`'outerposition'` | `'innerposition'`

Position property to hold constant when adding, removing, or changing decorations, specified as one of the following values:

- `'outerposition'` — The `OuterPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `InnerPosition` property.
- `'innerposition'` — The `InnerPosition` property remains constant when you add, remove, or change decorations such as a title or an axis label. If any positional adjustments are needed, MATLAB adjusts the `OuterPosition` property.

Note Setting this property has no effect when the parent container is a `TiledChartLayout`.

`Units` — Position units

`'normalized'` (default) | `'inches'` | `'centimeters'` | `'characters'` | `'points'` | `'pixels'`

Position units, specified as a value from the following table. To change the position of the chart in specific units, set the `Units` property before specifying the `Position` property. If you specify the `Units` and `Position` properties in a single command (using name-value pairs), be sure to specify `Units` before `Position`.

Units	**Description**
`'normalized'` (default)	Normalized with respect to the parent container, which is typically the figure, panel, or tab. The lower left corner of the container maps to `(0,0)`, and the upper right corner maps to `(1,1)`.
`'inches'`	Inches.
`'centimeters'`	Centimeters.

Units	Description
`'characters'`	Based on the default font of the graphics root object: • Character width = width of letter `x`. • Character height = distance between the baselines of two lines of text.
`'points'`	Typography points. One point equals 1/72 inch.
`'pixels'`	Distances in pixels are independent of your system resolution on Windows® and Macintosh systems: • On Windows systems, a pixel is 1/96th of an inch. • On Macintosh systems, a pixel is 1/72nd of an inch. On Linux® systems, the size of a pixel is determined by your system resolution.

`Layout` — Layout options
empty `LayoutOptions` array (default) | `TiledChartLayoutOptions`

Layout options, specified as a `TiledChartLayoutOptions` object. This property specifies options when an instance of your chart is a child of a tiled chart layout. If the instance is not a child of a tiled chart layout (for example, it is a child of a figure or panel), then this property is empty and has no effect. Otherwise, you can position the chart within the layout by setting the `Tile` and `TileSpan` properties on the `TiledChartLayoutOptions` object.

For example, this code places chart object `c` into the third tile of a tiled chart layout.

```
c.Layout.Tile = 3;
```

To make the chart span multiple tiles, specify the `TileSpan` property as a two-element vector. For example, this chart spans `2` rows and `3` columns of tiles.

```
c.Layout.TileSpan = [2 3];
```

Note Tiled chart layouts are not supported for the axes returned by the `getAxes` method. Instead, you can place an instance of your chart into a tiled chart layout.

View

`View` — Azimuth and elevation of view
`[0 90]` (default) | two-element vector of the form `[azimuth elevation]`

Azimuth and elevation of view, specified as a two-element vector of the form `[azimuth elevation]` defined in degree units. Alternatively, use the `view` function to set the view.

Version History

Introduced in R2021a

See Also

`pdeviz`

poimesh

(Not recommended) Generate regular mesh on rectangular geometry

---

Note `poimesh` is not recommended. To solve Poisson's equations, use `solvepde`. For details, see “Solve Problems Using PDEModel Objects”.

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 by dividing the rectangle into `nx` pieces along the *x*-direction and `ny` pieces along the *y*-direction, thus resulting in `(nx + 1)*(ny + 1)` nodes in the domain. The *x*-direction is the direction along the edge that makes the smallest angle with the *x*-axis.

For best performance with `poisolv`, the larger of `nx` and `ny` must be a power of 2.

If `g` is not a rectangle, `poimesh` returns `p` as zero.

`[p,e,t] = poimesh(g,n)` divides each edge into `n` pieces, that is, `nx = ny = n`.

`[p,e,t] = poimesh(g)` uses the value `nx = ny = n = 1`.

Examples

Fast Poisson Solver

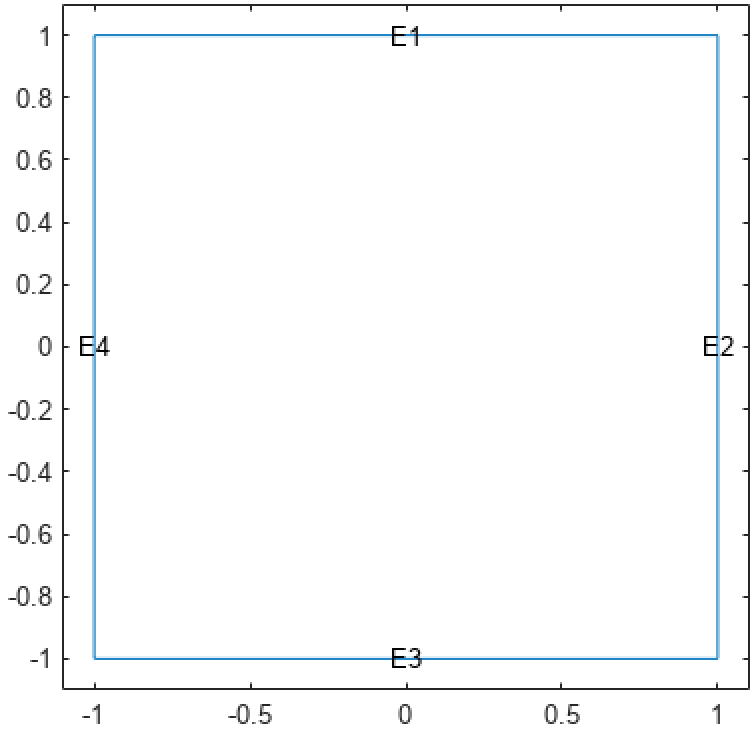
Solve the Poisson's equation $-\Delta u = 3x^2$ on a square domain with Dirichlet boundary conditions using the `poisolv` function.

Create a `model` object and include the square geometry created using the `squareg` function.

```
model = createpde;
g = @squareg;
geometryFromEdges(model,g);
```

Plot the geometry with the edge labels.

```
pdegplot(model,'EdgeLabels','on')
axis([-1.1 1.1 -1.1 1.1])
```

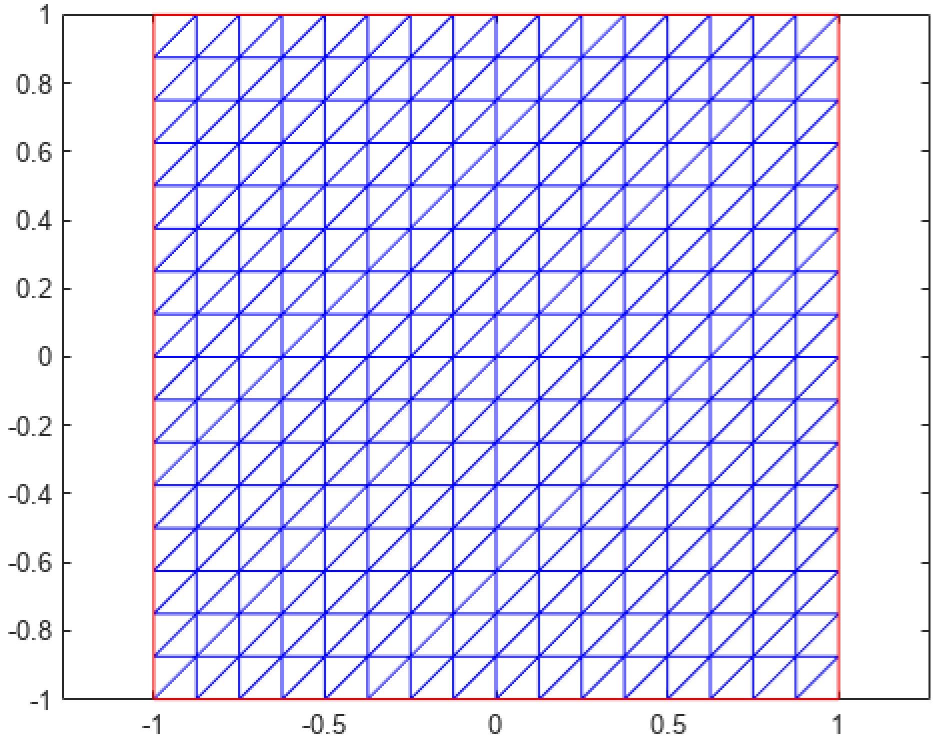


Apply the following Dirichlet boundary conditions. The solution is $0.2\cos(\pi y/2)$ on the right boundary (edge 2) and zero on all other boundaries.

```
innerBC = @(region,state) 0.2*cos(pi/2*region.y);
applyBoundaryCondition(model,'Dirichlet','Edge',2,'u',innerBC);
applyBoundaryCondition(model,'Dirichlet','Edge',[1 3 4],'u',0);
```

The fast Poisson solver requires a regular rectangular grid. Use the `poimesh` function to generate a mesh meeting this requirement. Plot the mesh.

```
[p,e,t] = poimesh(g,16);
figure;
pdemesh(p,e,t);
axis equal
```



Specify the PDE coefficients.

```
c = 1;
a = 0;
f = '3*x.^2';
```

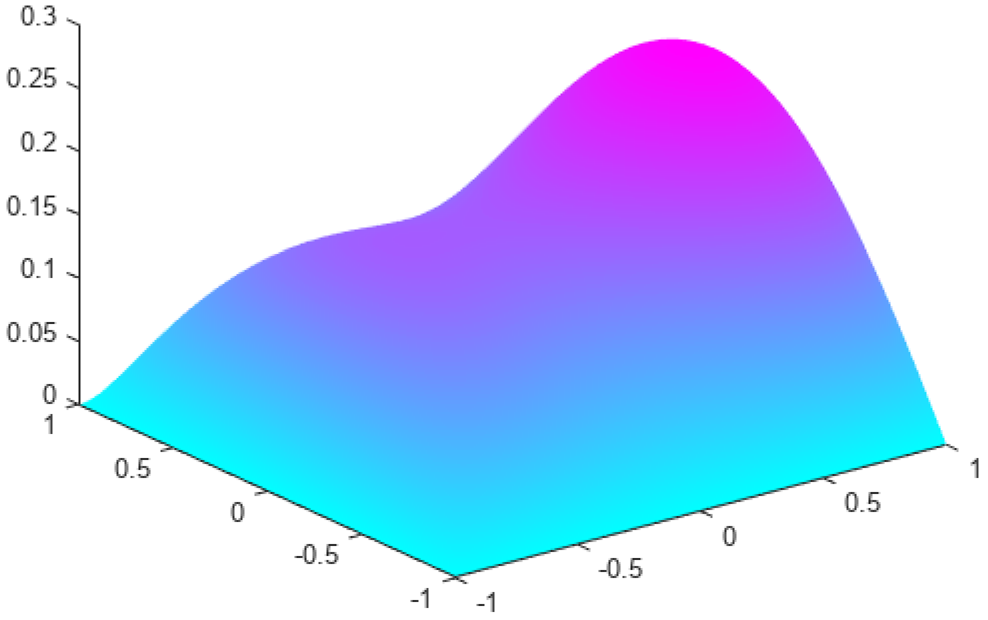
Solve the equation on different meshes using the `poisolv` function.

```
for n = [16 32 64 128 256 512]
    [p,e,t] = poimesh(g,n);
    tic;
      u = poisolv(model,p,e,t,f);
    tfast = toc;
    fprintf('%-5d|%15.5g\n',n,tfast);
end
```

```
16   |        0.43981
32   |        0.12702
64   |        0.12674
128  |        0.16597
256  |         0.5345
512  |         1.0985
```

Plot the solution on the finest mesh.

```
figure;
pdeplot(p,[],t,'XYData',u,'ZData',u,'ColorBar','off')
```



Input Arguments

g — Rectangular geometry

decomposed geometry matrix | geometry function | handle to geometry function

Rectangular geometry, specified as a decomposed geometry matrix, a geometry function, or a handle to the geometry function. For details about a decomposed geometry matrix, see `decsg`. For details about geometry functions, see “Parametrized Function for 2-D Geometry Creation” on page 2-22.

A geometry function must return the same result for the same input arguments in every function call. Thus, it must not contain functions and expressions designed to return a variety of results, such as random number generators.

If `g` is not a rectangle, `poimesh` returns `p` as zero.

Data Types: `double` | `char` | `string` | `function_handle`

nx — Number of divisions along *x*-direction

positive integer

Number of divisions along the *x*-direction, specified as a positive integer.

Data Types: `double`

ny — Number of divisions along *y*-direction

positive integer

Number of divisions along the *y*-direction, specified as a positive integer.

Data Types: `double`

n — Number of divisions
positive integer

Number of divisions along both the *x*- and *y*-direction, specified as a positive integer. In this case, both the *x*- and *y*-edges are divided into the same number of pieces.

Data Types: `double`

Output Arguments

p — Mesh points
matrix

Mesh points, returned as a 2-by-`Np` matrix of points, where `Np` is the number of points in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

e — Mesh edges
matrix

Mesh edges, returned as a `7`-by-`Ne` matrix of edges, where `Ne` is the number of edges in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh triangles
matrix

Mesh triangles, returned as a `4`-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on the mesh data representation, see `initmesh`.

Data Types: `double`

Version History

Introduced before R2006a

Not recommended
Not recommended starting in R2016a

`poimesh` is not recommended. To solve Poisson's equations, use `solvepde`. For details, see “Solve Problems Using PDEModel Objects” on page 2-2. There are no plans to remove `poimesh`.

See Also

`poisolv`

Topics
“Mesh Data as [p,e,t] Triples” on page 2-172

poisolv

(Not recommended) Fast solver for Poisson's equation on rectangular grid

---

Note `poisolv` is not recommended. To solve Poisson's equations, use `solvepde`. For details, see "Solve Problems Using PDEModel Objects".

Syntax

```
u = poisolv(model,p,e,t,f)
u = poisolv(b,p,e,t,f)
```

Description

`u = poisolv(model,p,e,t,f)` solves a Poisson's equation $\Delta u = f$ on a regular rectangular `[p,e,t]` mesh. The model must have only Dirichlet boundary conditions. A combination of sine transforms and tridiagonal solutions is used for increased performance.

`u = poisolv(b,p,e,t,f)` solves a Poisson's equation with Dirichlet boundary conditions $u = b$ on a regular rectangular `[p,e,t]` mesh.

Examples

Fast Poisson Solver

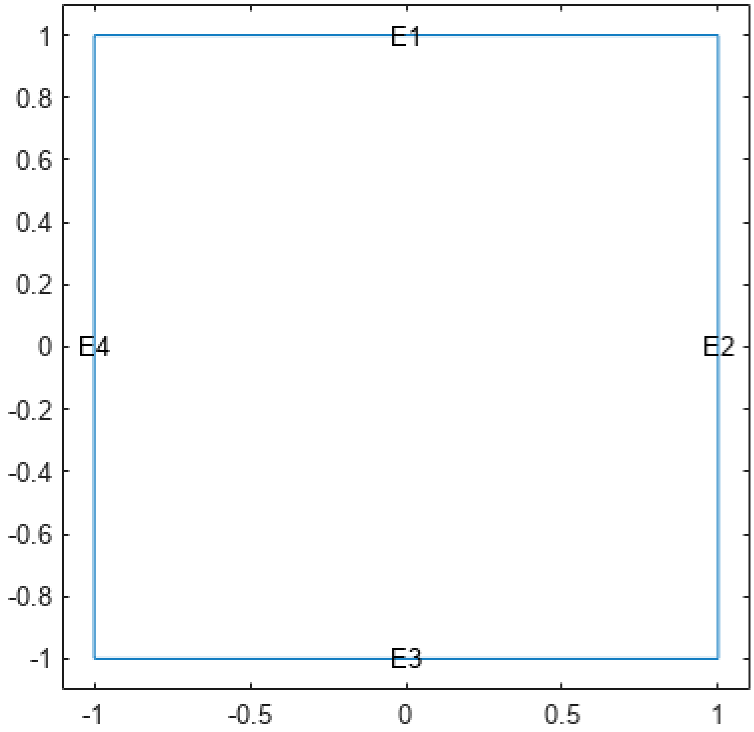
Solve the Poisson's equation $-\Delta u = 3x^2$ on a square domain with Dirichlet boundary conditions using the `poisolv` function.

Create a `model` object and include the square geometry created using the `squareg` function.

```
model = createpde;
g = @squareg;
geometryFromEdges(model,g);
```

Plot the geometry with the edge labels.

```
pdegplot(model,'EdgeLabels','on')
axis([-1.1 1.1 -1.1 1.1])
```

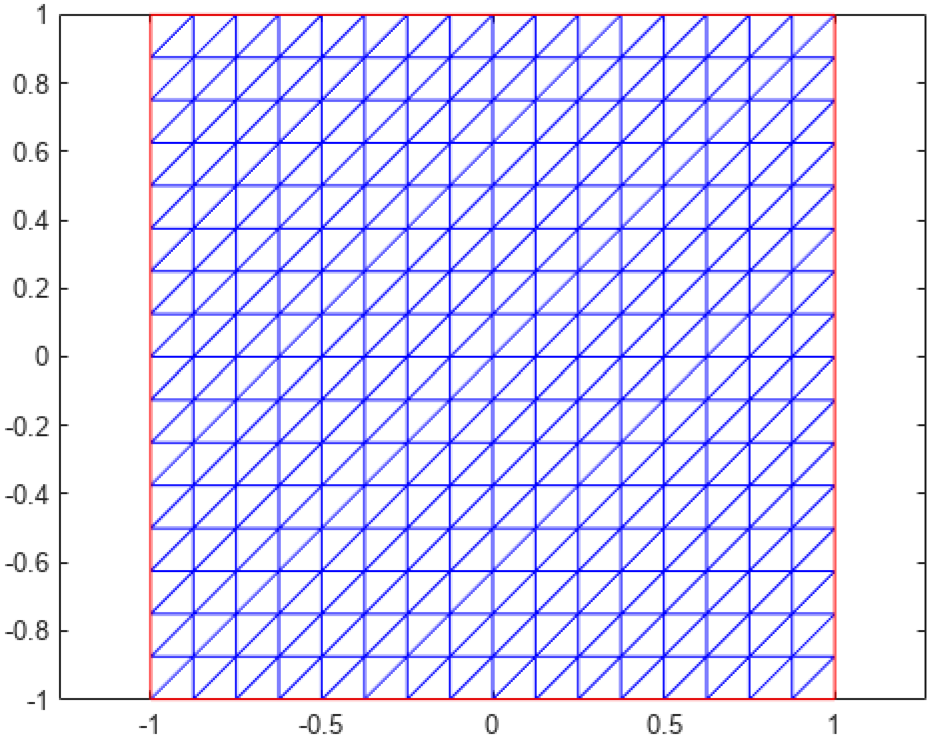


Apply the following Dirichlet boundary conditions. The solution is $0.2\cos(\pi y/2)$ on the right boundary (edge 2) and zero on all other boundaries.

```
innerBC = @(region,state) 0.2*cos(pi/2*region.y);
applyBoundaryCondition(model,'Dirichlet','Edge',2,'u',innerBC);
applyBoundaryCondition(model,'Dirichlet','Edge',[1 3 4],'u',0);
```

The fast Poisson solver requires a regular rectangular grid. Use the `poimesh` function to generate a mesh meeting this requirement. Plot the mesh.

```
[p,e,t] = poimesh(g,16);
figure;
pdemesh(p,e,t);
axis equal
```



Specify the PDE coefficients.

```
c = 1;
a = 0;
f = '3*x.^2';
```

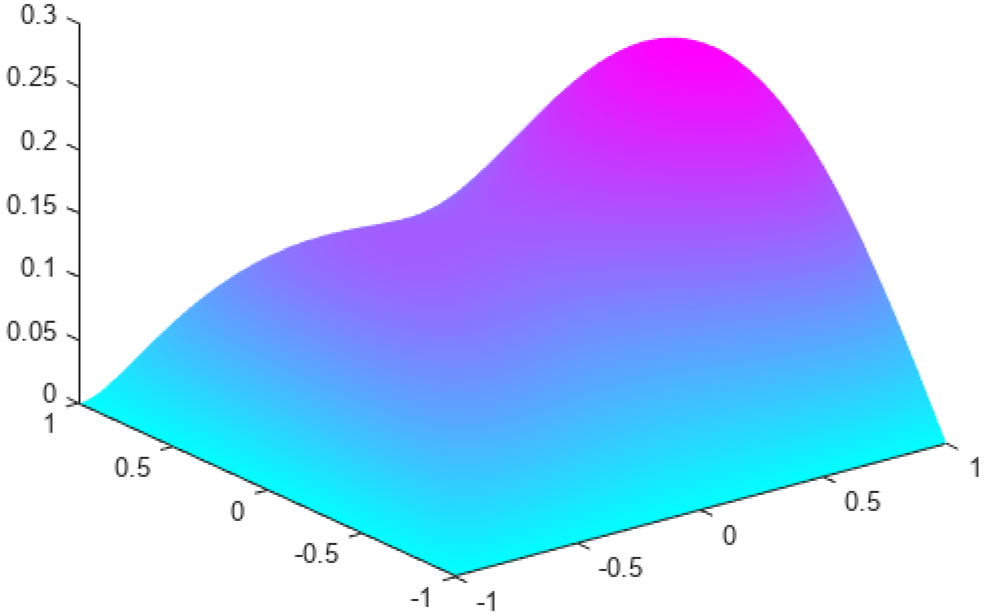
Solve the equation on different meshes using the `poisolv` function.

```
for n = [16 32 64 128 256 512]
    [p,e,t] = poimesh(g,n);
    tic;
      u = poisolv(model,p,e,t,f);
    tfast = toc;
    fprintf('%-5d|%15.5g\n',n,tfast);
end
```

```
16   |        0.43981
32   |        0.12702
64   |        0.12674
128  |        0.16597
256  |         0.5345
512  |         1.0985
```

Plot the solution on the finest mesh.

```
figure;
pdeplot(p,[],t,'XYData',u,'ZData',u,'ColorBar','off')
```



Input Arguments

`model` — PDE model
`PDEModel` object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde`

`b` — Dirichlet boundary conditions for all boundary points
boundary matrix | boundary file

Dirichlet boundary conditions for all boundary points, 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 Modeler app.

Example: `b = 'circleb1'`, `b = "circleb1"`, or `b = @circleb1`

Data Types: `double` | `char` | `string` | `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 details on the mesh data representation, see `initmesh`.

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 details on the mesh data representation, see `initmesh`.

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 details on the mesh data representation, see `initmesh`.

Data Types: `double`

`f` — Right side of Poisson's equation
scalar | matrix | character vector | character array | string scalar | string vector | coefficient function

Right side of a Poisson's equation, specified as a scalar, matrix, character vector, character array, string scalar, string vector, or coefficient function.

Data Types: `double` | `char` | `string` | `function_handle`

Output Arguments

`u` — PDE solution
vector

PDE solution, returned as a vector.

Version History
Introduced before R2006a

Not recommended
Not recommended starting in R2016a

`poisolv` is not recommended. To solve Poisson's equations, use `solvepde`. For details, see “Solve Problems Using PDEModel Objects” on page 2-2. There are no plans to remove `poisolv`.

References

[1] Strang, G. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Cambridge, MA, 1986, pp. 453–458.

See Also
`poimesh`

Topics
“Mesh Data as [p,e,t] Triples” on page 2-172

reconstructSolution

Package: `pde`

Recover full-model transient solution from reduced-order model (ROM)

Syntax

```
structuralresults = reconstructSolution(Rcb,u,ut,utt,tlist)
thermalresults = reconstructSolution(Rtherm,u_therm,tlist)
```

Description

`structuralresults = reconstructSolution(Rcb,u,ut,utt,tlist)` recovers the full transient structural solution from the reduced-order model `Rcb`, displacement `u`, velocity `ut`, and acceleration `utt`. Typically, the displacement, velocity, and acceleration are the values returned by Simscape.

`thermalresults = reconstructSolution(Rtherm,u_therm,tlist)` recovers the full transient thermal solution from the reduced-order model `Rtherm`, temperature in modal coordinates `u_therm`, and the time-steps `tlist` that you used to solve the reduced model.

Examples

Reconstruct Structural Solution from ROM Results

Knowing the solution in terms of the interface degrees of freedom (DoFs) and modal DoFs, reconstruct the solution for the full structural transient model.

Create a structural model for transient analysis.

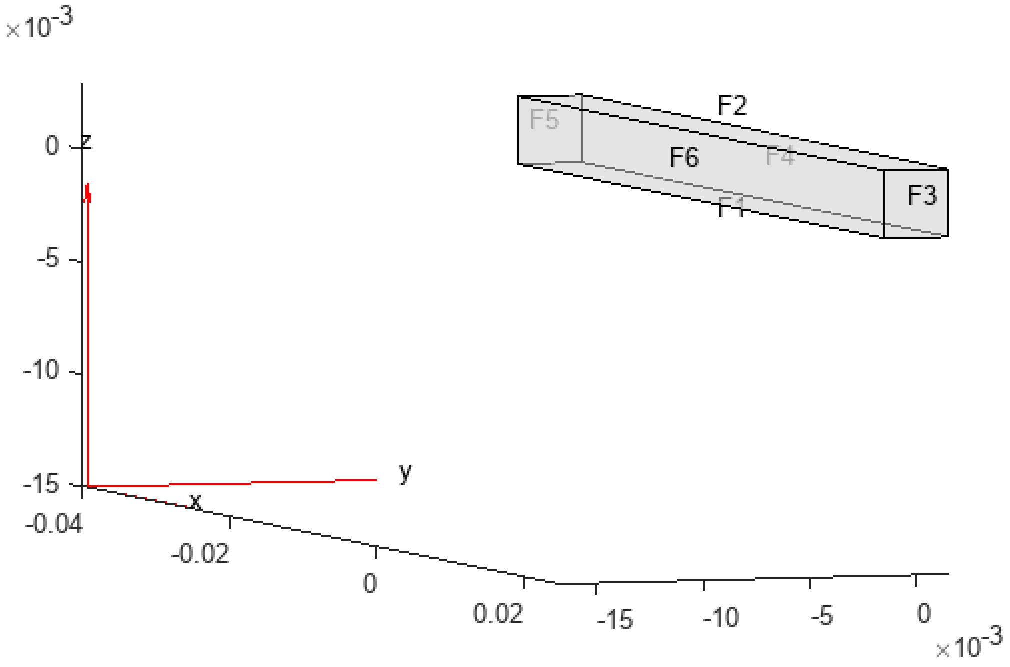
```
modelT = createpde("structural","transient-solid");
```

Create a square cross-section beam geometry and include it in the model.

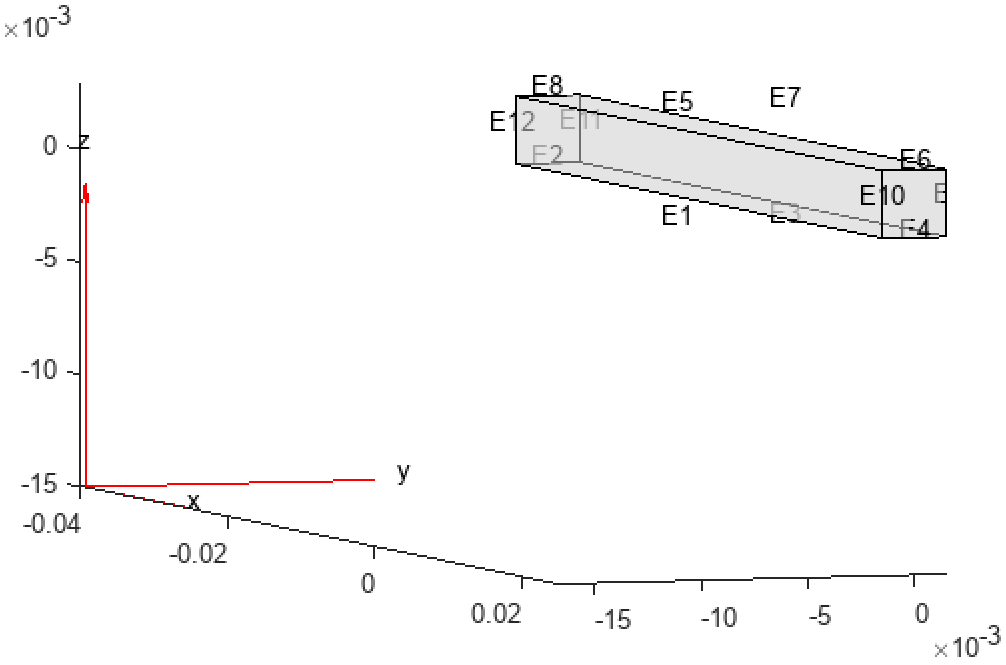
```
gm = multicuboid(0.05,0.003,0.003);
modelT.Geometry = gm;
```

Plot the geometry, displaying face and edge labels.

```
figure
pdegplot(modelT,"FaceLabels","on","FaceAlpha",0.5)
view([71 4])
```



```
figure
pdegplot(modelT,"EdgeLabels","on","FaceAlpha",0.5)
view([71 4])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelT,"YoungsModulus",210E9, ...
                            "PoissonsRatio",0.3, ...
                            "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(modelT,"Edge",[2 8 11 12],"Constraint","fixed");
```

Add a vertex at the center of face 3.

```
loadedVertex = addVertex(gm,"Coordinates",[0.025 0.0 0.0015]);
```

Generate a mesh.

```
generateMesh(modelT);
```

Apply a sinusoidal concentrated force in the *z*-direction on the new vertex.

```
structuralBoundaryLoad(modelT,"Vertex",loadedVertex, ...
                              "Force",[0;0;10],"Frequency",6000);
```

Specify zero initial conditions.

```
structuralIC(modelT,"Velocity",[0 0 0],"Displacement",[0 0 0]);
```

Define superelement interfaces using the fixed and loaded boundaries. In this case, the reduced-order model retains the DoFs on the fixed face and the loaded vertex while condensing all other DoFs in favor of modal DoFs. For better performance, use the set of edges bounding face 5 instead of using the entire face.

```
structuralSEInterface(modelT,"Edge",[2 8 11 12]);
structuralSEInterface(modelT,"Vertex",loadedVertex);
```

Reduce the structure, retaining all fixed interface modes up to `5e5`.

```
rom = reduce(modelT,"FrequencyRange",[-0.1,5e5]);
```

Next, use the reduced-order model to simulate the transient dynamics. Use the `ode15s` function directly to integrate the reduced system ODE. Working with the reduced model requires indexing into the reduced system matrices `rom.K` and `rom.M`. First, construct mappings of indices of `K` and `M` to loaded and fixed DoFs by using the data available in `rom`.

DoFs correspond to translational displacements. If the number of mesh points in a model is `Nn`, then the toolbox assigns the IDs to the DoFs as follows: the first `1` to `Nn` are *x*-displacements, `Nn+1` to `2*Nn` are *y*-displacements, and `2Nn+1` to `3*Nn` are *z*-displacements. The reduced model object `rom` contains these IDs for the retained DoFs in `rom.RetainedDoF`.

Create a function that returns DoF IDs given node IDs and the number of nodes.

```
getDoF = @(x,numNodes) [x(:); x(:) + numNodes; x(:) + 2*numNodes];
```

Knowing the DoF IDs for the given node IDs, use the `intersect` function to find the required indices.

```
numNodes = size(rom.Mesh.Nodes,2);

loadedNode = findNodes(rom.Mesh,"region","Vertex",loadedVertex);
loadDoFs = getDoF(loadedNode,numNodes);
[~,loadNodeROMIds,~] = intersect(rom.RetainedDoF,loadDoFs);
```

In the reduced matrices `rom.K` and `rom.M`, generalized modal DoFs appear after the retained DoFs.

```
fixedIntModeIds = (numel(rom.RetainedDoF) + 1:size(rom.K,1))';
```

Because fixed-end DoFs are not a part of the ODE system, the indices for the ODE DoFs in reduced matrices are as follows.

```
odeDoFs = [loadNodeROMIds;fixedIntModeIds];
```

The relevant components of `rom.K` and `rom.M` for time integration are:

```
Kconstrained = rom.K(odeDoFs,odeDoFs);
Mconstrained = rom.M(odeDoFs,odeDoFs);
numODE = numel(odeDoFs);
```

Now you have a second-order system of ODEs. To use `ode15s`, convert this into a system of first-order ODEs by applying linearization. Such a first-order system is twice the size of the second-order system.

```
Mode = [eye(numODE,numODE),   zeros(numODE,numODE); ...
        zeros(numODE,numODE), Mconstrained];
Kode = [zeros(numODE,numODE), -eye(numODE,numODE); ...
        Kconstrained,         zeros(numODE,numODE)];
Fode = zeros(2*numODE,1);
```

The specified concentrated force load in the full system is along the *z*-direction, which is the third DoF in the ODE system. Accounting for the linearization to obtain the first-order system gives the loaded ODE DoF.

```
loadODEDoF = numODE + 3;
```

Specify the mass matrix and the Jacobian for the ODE solver.

```
odeoptions = odeset;
odeoptions = odeset(odeoptions,"Jacobian",-Kode);
odeoptions = odeset(odeoptions,"Mass",Mode);
```

Specify zero initial conditions.

```
u0 = zeros(2*numODE,1);
```

Solve the reduced system by using ode15s and the helper function `CMSODEf`, which is defined at the end of this example.

```
tlist = 0:0.00005:3E-3;
sol = ode15s(@(t,y) CMSODEf(t,y,Kode,Fode,loadODEDoF), ...
                                 tlist,u0,odeoptions);
```

Compute the values of the ODE variable and the time derivatives.

```
[displ,vel] = deval(sol,tlist);
```

Knowing the solution in terms of the interface DoFs and modal DoFs, you can reconstruct the solution for the full model. The `reconstructSolution` function requires the displacement, velocity, and acceleration at all DoFs in `rom`. Construct the complete solution vector, including the zero values at the fixed DoFs.

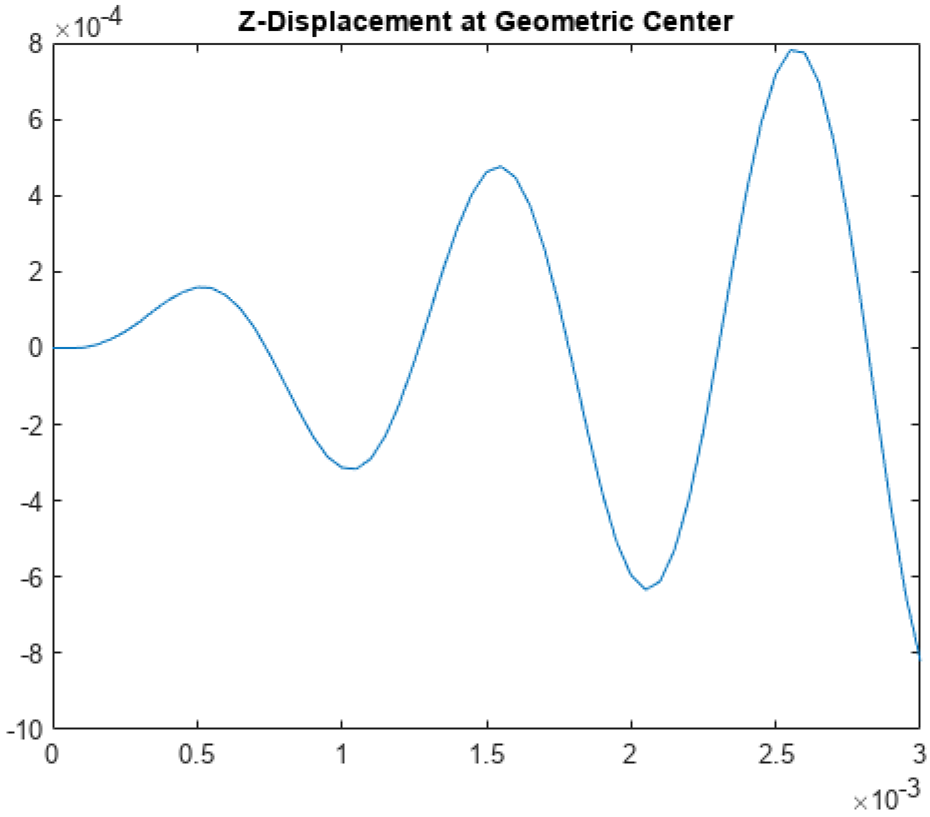
```
u = zeros(size(rom.K,1),numel(tlist));
ut = zeros(size(rom.K,1),numel(tlist));
utt = zeros(size(rom.K,1),numel(tlist));
u(odeDoFs,:) = displ(1:numODE,:);
ut(odeDoFs,:) = vel(1:numODE,:);
utt(odeDoFs,:) = vel(numODE+1:2*numODE,:);
```

Construct a transient results object using this solution.

```
RTrom = reconstructSolution(rom,u,ut,utt,tlist);
```

Compute the displacement in the interior at the center of the beam using the reconstructed solution.

```
coordCenter = [0;0;0];
iDispRTrom = interpolateDisplacement(RTrom, coordCenter);
figure
plot(tlist,iDispRTrom.uz)
title("Z-Displacement at Geometric Center")
```



ODE Helper Function

```
function f = CMSODEf(t,u,Kode,Fode,loadedVertex)
Fode(loadedVertex) = 10*sin(6000*t);
f = -Kode*u +Fode;
end
```

Reconstruct Thermal Solution from ROM Results

Reconstruct the solution for a full thermal transient model from the reduced-order model.

Create a transient thermal model.

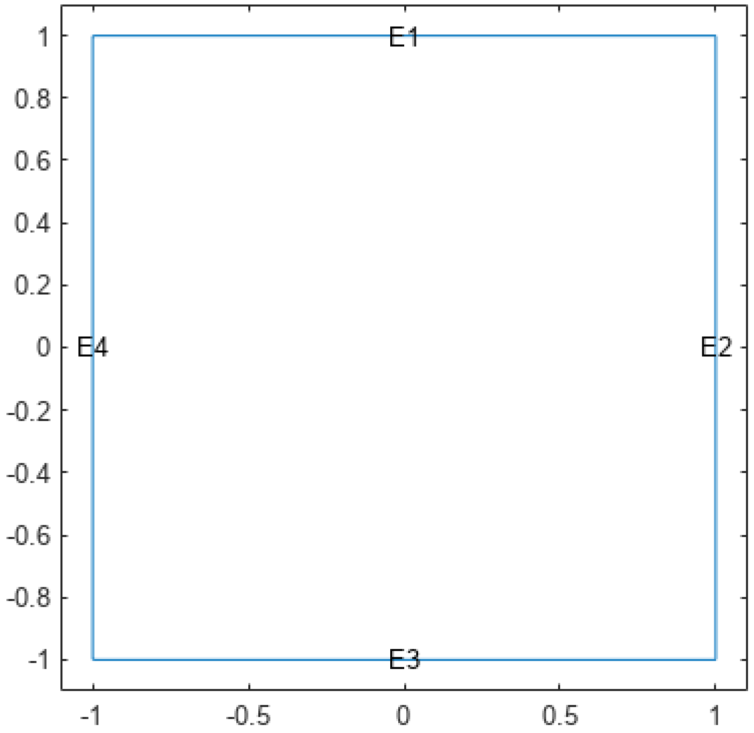
```
thermalmodel = createpde("thermal","transient");
```

Create a unit square geometry and include it in the model.

```
geometryFromEdges(thermalmodel,@squareg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",400, ...
                               "MassDensity",1300, ...
                               "SpecificHeat",600);
```

Set the temperature on the right edge to `100`.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Set an initial value of `50` for the temperature.

```
thermalIC(thermalmodel,50);
```

Generate a mesh.

```
generateMesh(thermalmodel);
```

Solve the model for three different values of heat source and collect snapshots.

```
tlist = 0:10:600;
snapShotIDs = [1:10 59 60 61];
Tmatrix = [];

heatVariation = [10000 15000 20000];
for q = heatVariation
    internalHeatSource(thermalmodel,q);
    results = solve(thermalmodel,tlist);
```

```
    Tmatrix = [Tmatrix,results.Temperature(:,snapShotIDs)];
end
```

Switch the thermal model analysis type to modal.

```
thermalmodel.AnalysisType = "modal";
```

Compute the POD modes.

```
RModal = solve(thermalmodel,"Snapshots",Tmatrix);
```

Reduce the thermal model.

```
Rtherm = reduce(thermalmodel,"ModalResults",RModal)
```

```
Rtherm = 
  ReducedThermalModel with properties:

                    K: [6x6 double]
                    M: [6x6 double]
                    F: [6x1 double]
    InitialConditions: [6x1 double]
                 Mesh: [1x1 FEMesh]
           ModeShapes: [1541x5 double]
     SnapshotsAverage: [1541x1 double]
```

Next, use the reduced-order model to simulate the transient dynamics. Use the `ode15s` function directly to integrate the reduced system ODE. Specify the mass matrix and the Jacobian for the ODE solver.

```
odeoptions = odeset;
odeoptions = odeset(odeoptions,"Mass",Rtherm.M);
odeoptions = odeset(odeoptions,"JConstant","on");
f = @(t,u) -Rtherm.K*u + Rtherm.F;
df = -Rtherm.K;
odeoptions = odeset(odeoptions,"Jacobian",df);
```

Solve the reduced system by using `ode15s`.

```
sol = ode15s(f,tlist,Rtherm.InitialConditions,odeoptions);
```

Compute the values of the ODE variable.

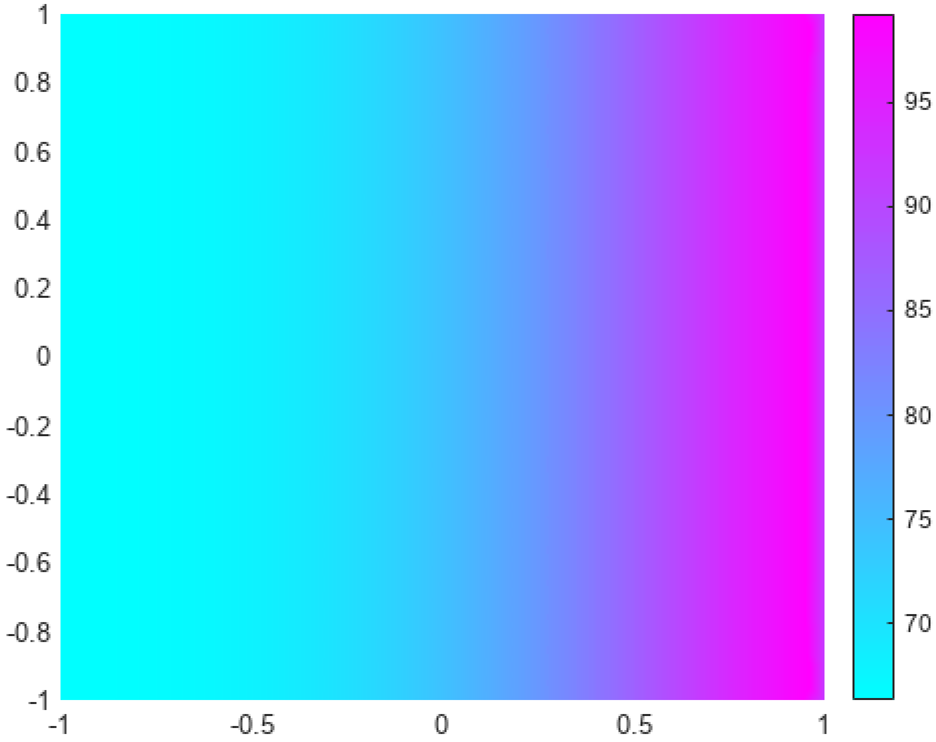
```
u = deval(sol,tlist);
```

Reconstruct the solution for the full model.

```
R = reconstructSolution(Rtherm,u,tlist);
```

Plot the temperature distribution at the last time step.

```
pdeplot(thermalmodel,"XYData",R.Temperature(:,end))
```



Input Arguments

Rcb — Structural results obtained using Craig-Bampton order reduction method
`ReducedStructuralModel` object

Structural results obtained using the Craig-Bampton order reduction method, specified as a `ReducedStructuralModel` object.

u — Displacement
matrix

Displacement, specified as a matrix. The number of rows in the matrix must equal the sum of the numbers of interface degrees of freedom and the number of modes. The *x*-displacements at the retained degrees of freedom must appear first, then the *y*-displacements, and, for a 3-D geometry, *z*-displacements, followed by the generalized modal degrees of freedom. The number of columns must equal the number of elements in `tlist`.

Data Types: `double`

ut — Velocity
matrix

Velocity, specified as a matrix. The number of rows in the matrix must equal the sum of the numbers of interface degrees of freedom and the number of modes. The *x*-velocities at the retained degrees of freedom must appear first, then the *y*-velocities, and, for a 3-D geometry, *z*-velocities, followed by the generalized modal degrees of freedom. The number of columns must equal the number of elements in `tlist`.

Data Types: `double`

`utt` — Acceleration
matrix

Acceleration, specified as a matrix. The number of rows in the matrix must equal the sum of the numbers of interface degrees of freedom and the number of modes. The *x*-accelerations at the retained degrees of freedom must appear first, then the *y*-accelerations, and, for a 3-D geometry, *z*-accelerations, followed by the generalized modal degrees of freedom. The number of columns must equal the number of elements in `tlist`.

Data Types: `double`

`tlist` — Solution times for solving reduced-order model
real vector

Solution times for solving the reduced-order model, specified as a real vector.

Data Types: `double`

`Rtherm` — Reduced-order thermal model
`ReducedThermalModel` object

Reduced-order thermal model, specified as a `ReducedThermalModel` object.

`u_therm` — Temperature in modal coordinates
matrix

Temperature in modal coordinates, specified as a matrix. The number of rows in the matrix must equal the number of modes. The number of columns must equal the number of elements in `tlist`.

Data Types: `double`

Output Arguments

`structuralresults` — Transient structural analysis results
`TransientStructuralResults` object

Transient structural analysis results, returned as a `TransientStructuralResults` object. The object contains the displacement, velocity, and acceleration values at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`.

`thermalresults` — Transient thermal analysis results
`TransientThermalResults` object

Transient thermal analysis results, returned as a `TransientThermalResults` object. The object contains the temperature and gradient values at the nodes of the triangular or tetrahedral mesh generated by `generateMesh`.

Version History

Introduced in R2019b

ROM support for thermal analysis

`reconstructSolution` now also reconstructs transient thermal solutions.

See Also

`StructuralModel` | `ReducedStructuralModel` | `ThermalModel` | `ReducedThermalModel` | `ModalThermalResults` | `reduce` | `structuralBC` | `structuralSEInterface` | `solve`

reduce

Package: pde

Reduce structural or thermal model

Syntax

```
Rcb = reduce(structuralmodel,"FrequencyRange",[omega1,omega2])
```

```
Rtherm = reduce(thermalmodel,"ModalResults",thermalModalR)
Rtherm = reduce(thermalmodel,"ModalResults",thermalModalR,"NumModes",N)
```

Description

`Rcb = reduce(structuralmodel,"FrequencyRange",[omega1,omega2])` reduces a structural analysis model to the fixed interface modes in the frequency range `[omega1,omega2]` and the boundary interface degrees of freedom.

`Rtherm = reduce(thermalmodel,"ModalResults",thermalModalR)` reduces a thermal analysis model to the modes specified in `thermalModalR`. When reducing a thermal model, thermal properties of materials, internal heat sources, and boundary conditions cannot depend on time or temperature.

`Rtherm = reduce(thermalmodel,"ModalResults",thermalModalR,"NumModes",N)` also truncates the number of modes to `N`. Using this syntax, you can compute a larger number of modes and then use a subset of these modes to construct a reduced-order model.

Examples

Reduce Transient Structural Model

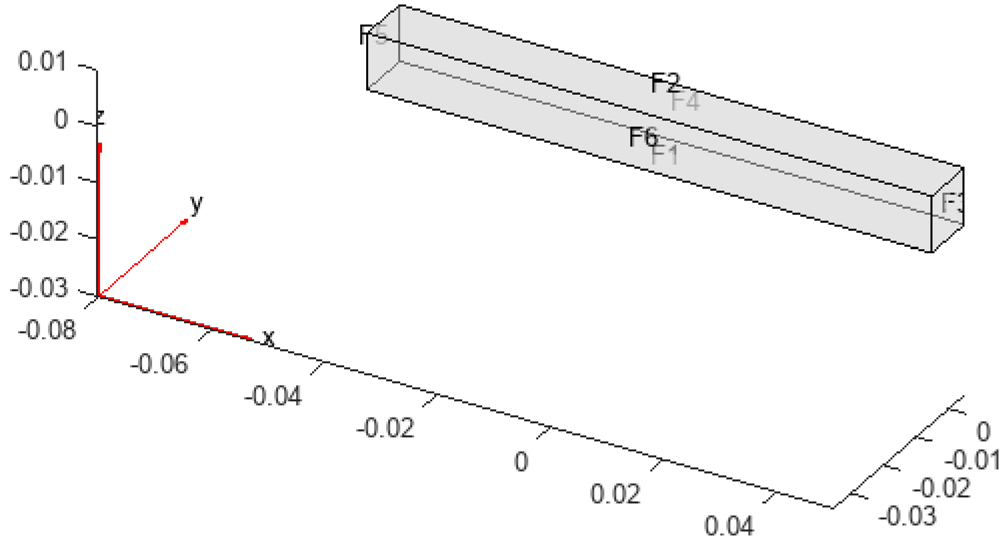
Reduce a transient structural model to the fixed interface modes in a specified frequency range and the boundary interface degrees of freedom.

Create a transient structural model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.1,0.01,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",70E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",2700);
```

Generate a mesh.

```
generateMesh(structuralmodel);
```

Specify the ends of the beam as structural superelement interfaces. The reduced-order model technique retains the degrees of freedom on the superelement interfaces while condensing the degrees of freedom on all other boundaries. For better performance, use the set of edges that bound each side of the beam instead of using the entire face.

```
structuralSEInterface(structuralmodel,"Edge",[4,6,9,10]);
structuralSEInterface(structuralmodel,"Edge",[2,8,11,12]);
```

Reduce the model to the fixed interface modes in the frequency range `[-Inf,500000]` and the boundary interface degrees of freedom.

```
R = reduce(structuralmodel,"FrequencyRange",[-Inf,500000])
```

```
R =
  ReducedStructuralModel with properties:

                     K: [166x166 double]
                     M: [166x166 double]
```

```
            NumModes: 22
          RetainedDoF: [144x1 double]
  ReferenceLocations: []
                 Mesh: [1x1 FEMesh]
```

Reduce Thermal Model

Reduce a thermal model using all modes or the specified number of modes from the modal solution.

Create a transient thermal model.

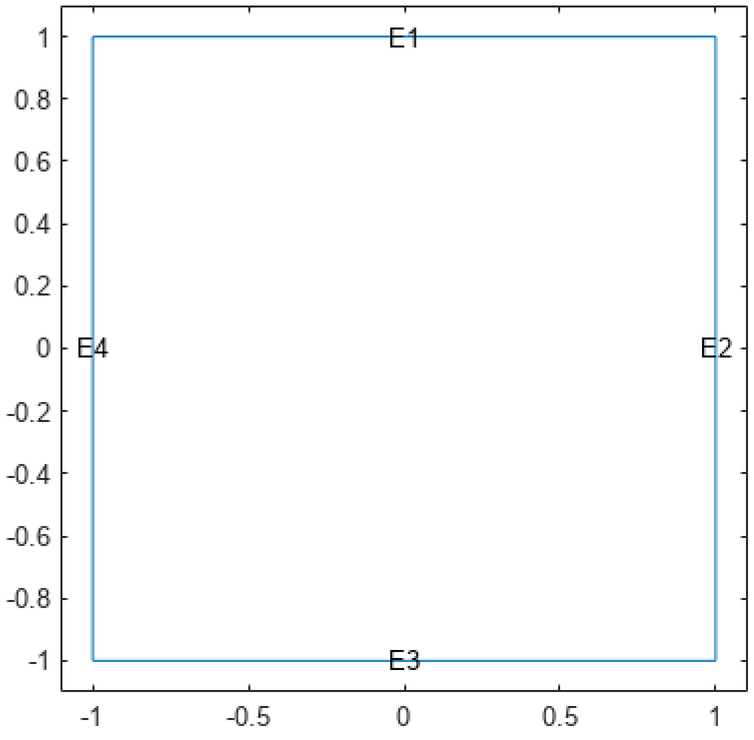
```
thermalmodel = createpde("thermal","transient");
```

Create a unit square geometry and include it in the model.

```
geometryFromEdges(thermalmodel,@squareg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",400, ...
                               "MassDensity",1300, ...
                               "SpecificHeat",600);
```

Set the temperature on the right edge to `100`.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Set an initial value of `0` for the temperature.

```
thermalIC(thermalmodel,0);
```

Generate a mesh.

```
generateMesh(thermalmodel);
```

Solve the model for three different values of heat source and collect snapshots.

```
tlist = 0:10:600;
snapShotIDs = [1:10 59 60 61];
Tmatrix = [];

heatVariation = [10000 15000 20000];
for q = heatVariation
    internalHeatSource(thermalmodel,q);
    results = solve(thermalmodel,tlist);
    Tmatrix = [Tmatrix,results.Temperature(:,snapShotIDs)];
end
```

Switch the thermal model analysis type to modal.

```
thermalmodel.AnalysisType = "modal";
```

Compute the POD modes.

```
RModal = solve(thermalmodel,"Snapshots",Tmatrix)
```

```
RModal = 
  ModalThermalResults with properties:

        DecayRates: [6x1 double]
        ModeShapes: [1541x6 double]
    SnapshotsAverage: [1541x1 double]
          ModeType: "PODModes"
              Mesh: [1x1 FEMesh]
```

Reduce the thermal model using all modes in `RModal`.

```
Rtherm = reduce(thermalmodel,"ModalResults",RModal)
```

```
Rtherm = 
  ReducedThermalModel with properties:

                    K: [7x7 double]
                    M: [7x7 double]
                    F: [7x1 double]
    InitialConditions: [7x1 double]
                 Mesh: [1x1 FEMesh]
```

```
          ModeShapes: [1541x6 double]
     SnapshotsAverage: [1541x1 double]
```

Reduce the thermal model using only three modes.

```
Rtherm3 = reduce(thermalmodel,"ModalResults",RModal, ...
                             "NumModes",3)
```

```
Rtherm3 = 
  ReducedThermalModel with properties:

                    K: [4x4 double]
                    M: [4x4 double]
                    F: [4x1 double]
    InitialConditions: [4x1 double]
                 Mesh: [1x1 FEMesh]
           ModeShapes: [1541x3 double]
     SnapshotsAverage: [1541x1 double]
```

Input Arguments

`structuralmodel` — Structural model

`StructuralModel` object

Structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`[omega1,omega2]` — Frequency range

vector of two elements

Frequency range, specified as a vector of two elements. Define `omega1` as slightly lower than the lowest mode's frequency and `omega2` as slightly higher than the highest mode's frequency. For example, if the lowest expected frequency is zero, then use a small negative value for `omega1`.

You can find natural frequencies and mode shapes for the specified frequency range by solving a modal analysis problem first. Then you can use a more precise frequency range to reduce the model. Note that a modal analysis problem still requires you to specify a frequency range. For example, see “Modal Superposition Method for Structural Dynamics Problem” on page 3-99.

Example: `[-0.1,1000]`

Data Types: `double`

`thermalmodel` — Modal thermal analysis model

`ThermalModel` object

Modal thermal analysis model, specified as a `ThermalModel` object. `ThermalModel` contains the geometry, mesh, thermal properties of the material, internal heat source, Stefan-Boltzmann constant, boundary conditions, and initial conditions.

Example: `thermalmodel = createpde("thermal","modal")`

`thermalModalR` — Modal analysis results for thermal model
`ModalThermalResults` object

Modal analysis results for a thermal model, specified as a `ModalThermalResults` object.

Example: `thermalModalR = solve(thermalmodel,"DecayRange",[0,1000])`

`N` — Number of modes
positive integer

Number of modes, specified as a positive integer.

Output Arguments

`Rcb` — Structural results obtained using Craig-Bampton order reduction method
`ReducedStructuralModel` object

Structural results obtained using the Craig-Bampton order reduction method, returned as a `ReducedStructuralModel` object.

`Rtherm` — Reduced-order thermal model
`ReducedThermalModel` object

Reduced-order thermal model, returned as a `ReducedThermalModel` object.

Version History

Introduced in R2019b

ROM support for thermal analysis

`reduce` now also reduces thermal models.

See Also

`reconstructSolution` | `StructuralModel` | `solve` | `structuralBC` | `structuralSEInterface` | `ReducedStructuralModel` | `ThermalModel` | `ModalThermalResults`

ReducedStructuralModel

Reduced-order structural model results

Description

A `ReducedStructuralModel` object contains the stiffness matrix K, mass matrix M, mesh, multipoint constraint reference locations, and IDs of retained degrees of freedom.

To expand this data to a full solution that includes displacement, velocity, and acceleration, use `reconstructSolution`.

Creation

Reduce a structural model by using the `reduce` function. This function returns structural results obtained using the Craig-Bampton reduced order method as a `ReducedStructuralModel` object.

Properties

`K` — Reduced stiffness matrix
real matrix

Reduced stiffness matrix, specified as a real `N`-by-`N` matrix.

- For models without multipoint constraints, `N` is the sum of the number of retained degrees of freedom and the number of fixed interface modes.
- For models with `Nmp` multipoint constraints, `N` is the sum of `6*Nmp` and the number of fixed interface modes.

Data Types: `double`

`M` — Reduced mass matrix
real matrix

Reduced mass matrix, specified as a real `N`-by-`N` matrix.

- For models without multipoint constraints, `N` is the sum of the number of retained degrees of freedom and the number of fixed interface modes.
- For models with `Nmp` multipoint constraints, `N` is the sum of `6*Nmp` and the number of fixed interface modes.

Data Types: `double`

`NumModes` — Number of fixed interface modes
integer

Number of fixed interface modes, specified as an integer.

Data Types: `double`

`RetainedDoF` — IDs of retained degrees of freedom
real vector

IDs of retained degrees of freedom, specified as a real vector.

Data Types: `double`

`ReferenceLocations` — Multipoint constraint reference locations
real matrix

Multipoint constraint reference locations, specified as a real 2-by-`Nmp` or 3-by-`Nmp` matrix for a 2-D or 3-D geometry, respectively. Here, `Nmp` is the number of multipoint constraints. If there are no multipoint constraints, `ReferenceLocations` is an empty matrix.

Data Types: `double`

`Mesh` — Finite element mesh
`FEMesh` object

Finite element mesh, specified as a `FEMesh` object. For details, see FEMesh.

Object Functions

reconstructSolution	Recover full-model transient solution from reduced-order model (ROM)

Examples

Reduce Transient Structural Model

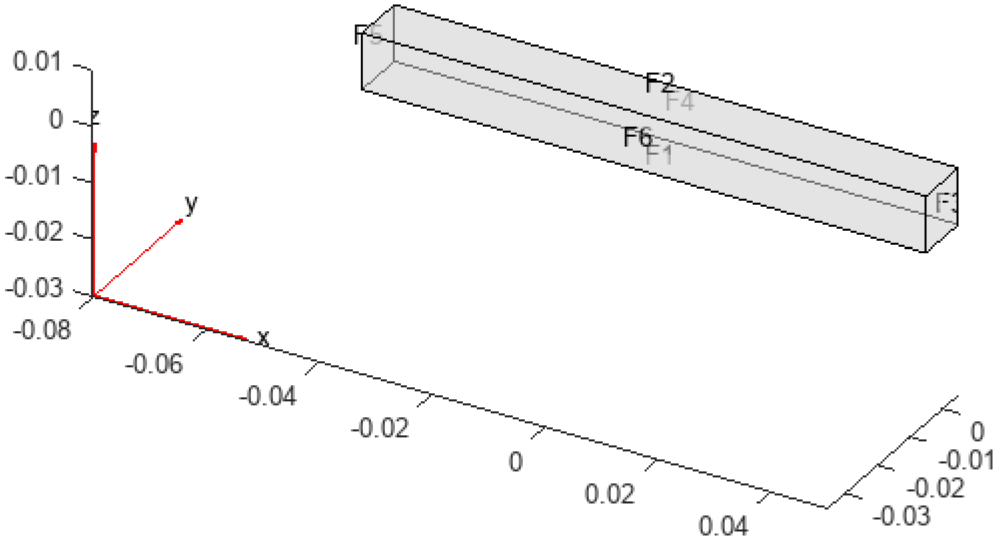
Reduce a transient structural model to the fixed interface modes in a specified frequency range and the boundary interface degrees of freedom.

Create a transient structural model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.1,0.01,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",70E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",2700);
```

Generate a mesh.

```
generateMesh(structuralmodel);
```

Specify the ends of the beam as structural superelement interfaces. The reduced-order model technique retains the degrees of freedom on the superelement interfaces while condensing the degrees of freedom on all other boundaries. For better performance, use the set of edges that bound each side of the beam instead of using the entire face.

```
structuralSEInterface(structuralmodel,"Edge",[4,6,9,10]);
structuralSEInterface(structuralmodel,"Edge",[2,8,11,12]);
```

Reduce the model to the fixed interface modes in the frequency range `[-Inf,500000]` and the boundary interface degrees of freedom.

```
R = reduce(structuralmodel,"FrequencyRange",[-Inf,500000])
```

```
R = 
  ReducedStructuralModel with properties:

                     K: [166x166 double]
                     M: [166x166 double]
```

```
             NumModes: 22
          RetainedDoF: [144x1 double]
   ReferenceLocations: []
                 Mesh: [1x1 FEMesh]
```

More About

Degrees of Freedom (DoFs)

In Partial Differential Equation Toolbox, each node of a 2-D or 3-D geometry has two or three degrees of freedom (DoFs), respectively. DoFs correspond to translational displacements. If the number of mesh points in a model is `NumNodes`, then the toolbox assigns the IDs to the degrees of freedom as follows:

- Numbers from 1 to `NumNodes` correspond to an `x`-displacement at each node.
- Numbers from `NumNodes+1` to `2*NumNodes` correspond to a `y`-displacement at each node.
- Numbers from `2*NumNodes+1` to `3*NumNodes` correspond to a `z`-displacement at each node of a 3-D geometry.

Version History

Introduced in R2019b

See Also

`reduce` | `reconstructSolution` | `structuralSEInterface` | `structuralBC` | `StructuralModel`

ReducedThermalModel

Reduced-order thermal model

Description

A `ReducedThermalModel` object contains the reduced stiffness matrix K, reduced mass matrix M, reduced load vector F, initial conditions, mode shapes, mesh, and the average of snapshots used for proper orthogonal decomposition (POD).

To expand this data to a full transient thermal solution, use `reconstructSolution`.

Creation

Reduce a thermal model by using the `reduce` function. This function returns a reduced-order thermal model as a `ReducedThermalModel` object.

Properties

`K` — Reduced stiffness matrix
matrix

Reduced stiffness matrix, specified as a matrix.

Data Types: `double`

`M` — Reduced mass matrix
matrix

Reduced mass matrix, specified as a matrix.

Data Types: `double`

`F` — Reduced load vector
column vector

Reduced load vector, specified as a column vector.

Data Types: `double`

`InitialConditions` — Initial conditions in modal coordinates
column vector

Initial conditions in modal coordinates, specified as a column vector.

Data Types: `double`

`ModeShapes` — Modes used to obtain reduced-order model
matrix

Modes used to obtain a reduced-order model, specified as a matrix.

Data Types: `double`

`Mesh` — Finite element mesh

`FEMesh` object

Finite element mesh, specified as an `FEMesh` object. For details, see FEMesh.

`SnapshotsAverage` — Average of snapshots used for POD

column vector

Average of snapshots used for POD, returned as a column vector.

Data Types: `double`

Object Functions

reconstructSolution	Recover full-model transient solution from reduced-order model (ROM)

Examples

Reduce Thermal Model

Reduce a thermal model using all modes or the specified number of modes from the modal solution.

Create a transient thermal model.

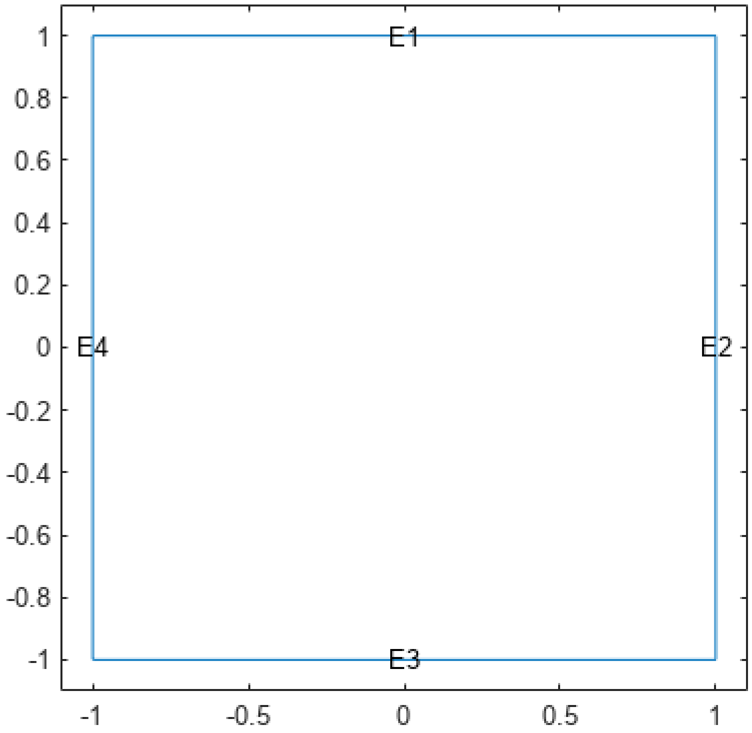
```
thermalmodel = createpde("thermal","transient");
```

Create a unit square geometry and include it in the model.

```
geometryFromEdges(thermalmodel,@squareg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",400, ...
                               "MassDensity",1300, ...
                               "SpecificHeat",600);
```

Set the temperature on the right edge to `100`.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Set an initial value of `0` for the temperature.

```
thermalIC(thermalmodel,0);
```

Generate a mesh.

```
generateMesh(thermalmodel);
```

Solve the model for three different values of heat source and collect snapshots.

```
tlist = 0:10:600;
snapShotIDs = [1:10 59 60 61];
Tmatrix = [];

heatVariation = [10000 15000 20000];
for q = heatVariation
    internalHeatSource(thermalmodel,q);
    results = solve(thermalmodel,tlist);
```

```
    Tmatrix = [Tmatrix,results.Temperature(:,snapShotIDs)];
end
```

Switch the thermal model analysis type to modal.

```
thermalmodel.AnalysisType = "modal";
```

Compute the POD modes.

```
RModal = solve(thermalmodel,"Snapshots",Tmatrix)
```

```
RModal = 
  ModalThermalResults with properties:

          DecayRates: [6x1 double]
          ModeShapes: [1541x6 double]
    SnapshotsAverage: [1541x1 double]
            ModeType: "PODModes"
                Mesh: [1x1 FEMesh]
```

Reduce the thermal model using all modes in `RModal`.

```
Rtherm = reduce(thermalmodel,"ModalResults",RModal)
```

```
Rtherm = 
  ReducedThermalModel with properties:

                    K: [7x7 double]
                    M: [7x7 double]
                    F: [7x1 double]
    InitialConditions: [7x1 double]
                 Mesh: [1x1 FEMesh]
           ModeShapes: [1541x6 double]
     SnapshotsAverage: [1541x1 double]
```

Reduce the thermal model using only three modes.

```
Rtherm3 = reduce(thermalmodel,"ModalResults",RModal, ...
                              "NumModes",3)
```

```
Rtherm3 = 
  ReducedThermalModel with properties:

                    K: [4x4 double]
                    M: [4x4 double]
                    F: [4x1 double]
    InitialConditions: [4x1 double]
                 Mesh: [1x1 FEMesh]
           ModeShapes: [1541x3 double]
     SnapshotsAverage: [1541x1 double]
```

Version History

Introduced in R2022a

See Also

`reduce` | `reconstructSolution` | `solve` | `ThermalModel`

refinemesh

Package: `pde`

Refine triangular mesh

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 `generateMesh`.

Syntax

```
[p1,e1,t1] = refinemesh(g,p,e,t)
[p1,e1,t1,u1] = refinemesh(g,p,e,t,u)
[ ___ ] = refinemesh( ___ ,it)
[ ___ ] = refinemesh( ___ ,"longest")
```

Description

Note This function does not support quadratic 2-D elements.

`[p1,e1,t1] = refinemesh(g,p,e,t)` returns a refined version of the triangular mesh given by the mesh data `p`, `e`, and `t`. For details on the mesh data representation, see “Mesh Data as [p,e,t] Triples” on page 2-172.

`[p1,e1,t1,u1] = refinemesh(g,p,e,t,u)` refines the mesh and extends the solution `u` to the new mesh nodes by linear interpolation. The number of rows in `u` must correspond to the number of columns in `p`, and `u1` has as many rows as there are points in `p1`.

`refinemesh` interpolates each column of `u` separately.

`[ ___ ] = refinemesh( ___ ,it)` uses the input and output arguments from the previous syntaxes and specifies the list `it` of geometric faces or triangles to refine. A scalar or a row vector specifies faces. A column vector specifies triangles.

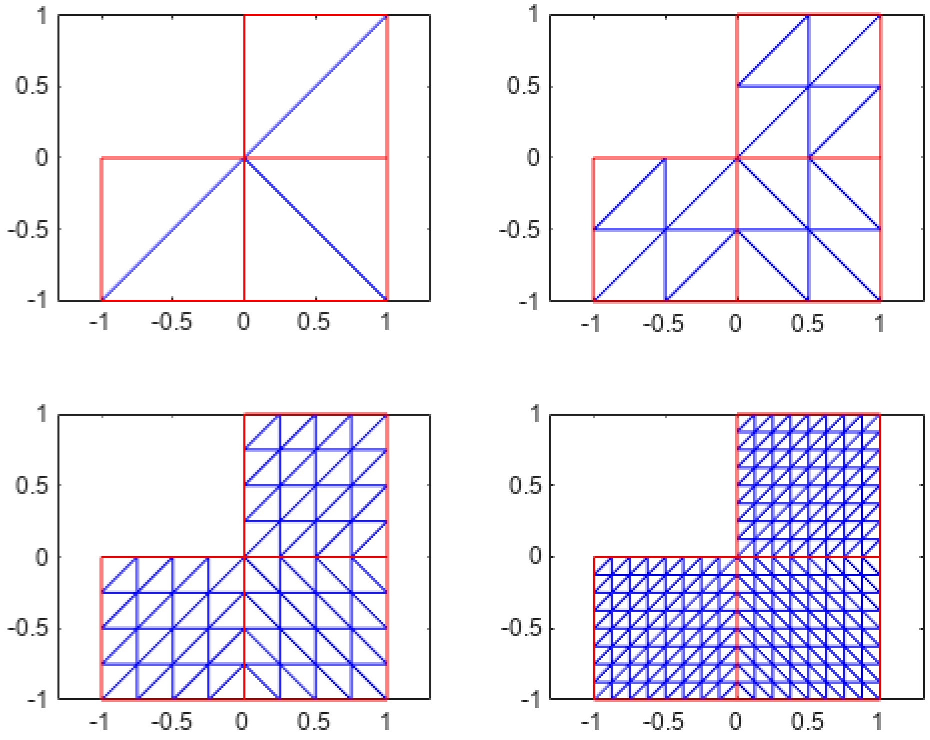
`[ ___ ] = refinemesh( ___ ,"longest")` uses the longest edge refinement, where the longest edge of each triangle is bisected. By default, `refinemesh` uses the regular refinement, where all triangles are divided into four triangles of the same shape. You also can explicitly specify `"regular"` instead of `"longest"`. If you use a column vector `it` to specify the triangles to refine, then `refinemesh` can refine some triangles outside of the specified set to preserve the triangulation and its quality.

Examples

Mesh Refinement

Refine the mesh of the L-shaped membrane several times. Plot the initial mesh and refined meshes at each step.

```
[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)
```

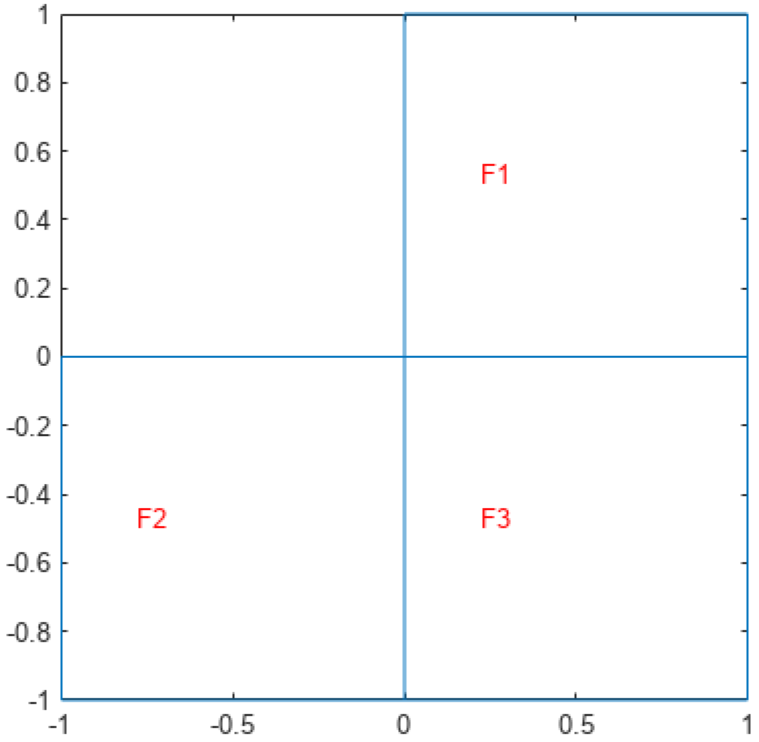


Mesh Refinement for Specified Faces

Refine the mesh for a particular face of the L-shaped membrane.

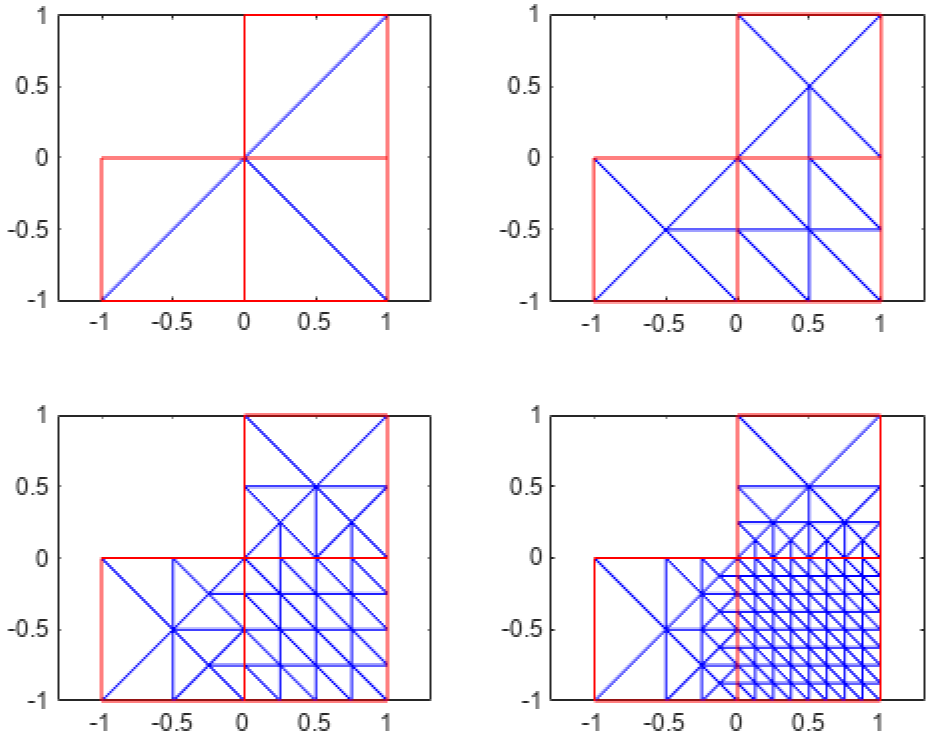
Plot the L-shaped membrane to identify the face numbers.

```
pdegplot("lshapeg","FaceLabels","on")
```



Create the initial mesh for the entire geometry, then refine the mesh for face 3 several times. Plot the initial mesh and refined meshes at each step.

```
[p,e,t] = initmesh("lshapeg","Hmax",Inf);
subplot(2,2,1)
pdemesh(p,e,t)

[p,e,t] = refinemesh("lshapeg",p,e,t,3);
subplot(2,2,2)
pdemesh(p,e,t)

[p,e,t] = refinemesh("lshapeg",p,e,t,3);
subplot(2,2,3)
pdemesh(p,e,t)

[p,e,t] = refinemesh("lshapeg",p,e,t,3);
subplot(2,2,4)
pdemesh(p,e,t)
```



Input Arguments

g — Geometry description

decomposed geometry matrix | geometry function | handle to geometry function

Geometry description, specified as a decomposed geometry matrix, a geometry function, or a handle to the geometry function. For details about a decomposed geometry matrix, see `decsg`. For details about a geometry function, see “Parametrized Function for 2-D Geometry Creation” on page 2-22.

A geometry function must return the same result for the same input arguments in every function call. Thus, it must not contain functions and expressions designed to return a variety of results, such as random number generators.

Data Types: `double` | `char` | `string` | `function_handle`

p — Mesh points

2-by-`Np` matrix

Mesh points, specified as a 2-by-`Np` matrix. `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)` and the *y*-coordinate of point `k` in `p(2,k)`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

e — Mesh edges

7-by-`Ne` matrix

Mesh edges, specified as a 7-by-`Ne` matrix, where `Ne` is the number of edges in the mesh. An edge is a pair of points in `p` containing a boundary between subdomains, or containing an outer boundary. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

`t` — Mesh elements

4-by-`Nt` matrix

Mesh elements, specified as a 4-by-`Nt` matrix. `Nt` is the number of triangles in the mesh.

The `t(i,k)`, with `i` ranging from 1 through `end - 1`, contain indices to the corner points of element `k`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172. The last row, `t(end,k)`, contains the subdomain number of the element.

`u` — PDE solution

vector

PDE solution, specified as a vector.

- If the PDE is scalar, meaning that it has 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, the next `Np` elements represent the solution of equation 2, and so on.

`it` — Faces or triangles to refine

positive number | vector of positive numbers

Faces or triangles to refine, specified as a positive number or a row or column vector of positive numbers. A scalar or a row vector specifies faces. A column vector specifies triangles.

Output Arguments

`p1` — Refined mesh points

2-by-`Np` matrix

Refined mesh points, returned as a 2-by-`Np` matrix. `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)` and the *y*-coordinate of point `k` in `p(2,k)`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

`e1` — Refined mesh edges

7-by-`Ne` matrix

Refined mesh edges, returned as a 7-by-`Ne` matrix, where `Ne` is the number of edges in the mesh. An edge is a pair of points in `p` containing a boundary between subdomains, or containing an outer boundary. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172.

`t1` — Refined mesh elements

4-by-`Nt` matrix

Refined mesh elements, returned as a 4-by-`Nt` matrix. `Nt` is the number of triangles in the mesh.

The `t(i,k)`, with `i` ranging from 1 through `end-1`, contain indices to the corner points of element `k`. For details, see “Mesh Data as [p,e,t] Triples” on page 2-172. The last row, `t(end,k)`, contains the subdomain number of the element.

u1 — PDE solution
vector

PDE solution, returned as a vector.

- If the PDE is scalar, meaning that it has only one equation, then `u1` is a column vector representing the solution *u1* at each node in the mesh.
- If the PDE is a system of $N > 1$ equations, then `u1` is a column vector with `N*Np` elements, where `Np` is the number of nodes in the mesh. The first `Np` elements of `u1` represent the solution of equation 1, the next `Np` elements represent the solution of equation 2, and so on.

Algorithms

The refinement algorithm follows these steps:

1. Pick the initial set of triangles to refine.
2. 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 more edges are divided.
5. Introduce new points of all divided edges, and replace all divided entries in `e` by two new entries.
6. Form the new triangles. If all three sides are divided, new triangles are formed by joining the side midpoints. If two sides are divided, the midpoint of the longest edge is joined with the opposing corner and with the other midpoint. If only the longest edge is divided, its midpoint is joined with the opposing corner.

Version History

Introduced before R2006a

See Also

`initmesh`

Topics

“Mesh Data as [p,e,t] Triples” on page 2-172

rotate

Package: `pde`

Rotate geometry

Syntax

```
rotate(g,theta)
rotate(g,theta,refpoint)
rotate(g,theta,refpoint1,refpoint2)

h = rotate(g,___)
```

Description

`rotate(g,theta)` rotates the geometry `g` about the *z*-axis by the angle `theta`, specified in degrees. Rotation follows the right-hand rule: a positive angle `theta` rotates counterclockwise, while sighting along the *z*-axis toward the origin.

`rotate(g,theta,refpoint)` uses the rotation axis specified by the reference point `refpoint`. The axis of rotation is the line in the *z*-direction passing through the reference point.

`rotate(g,theta,refpoint1,refpoint2)` uses the rotation axis specified by two reference points. This syntax is only valid for a 3-D geometry.

`h = rotate(g,___)` returns a handle `h` to the resulting geometry object `g`. Use this syntax with any input arguments from the previous syntaxes.

If the original geometry is a `DiscreteGeometry` object, then the function modifies the original geometry and returns the handle `h` to the modified `DiscreteGeometry` geometry object. If the original geometry is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` geometry object. In this case, the original geometry remains unchanged.

Examples

Rotate 2-D Geometry

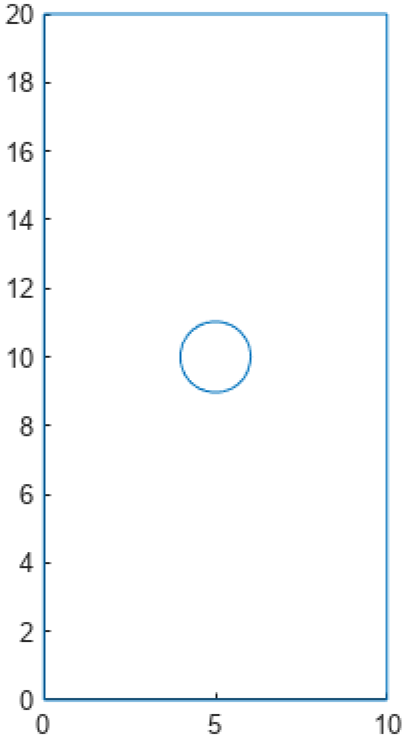
Rotate a geometry with and without specifying the reference point for the axis of rotation.

Create a model.

```
model = createpde;
```

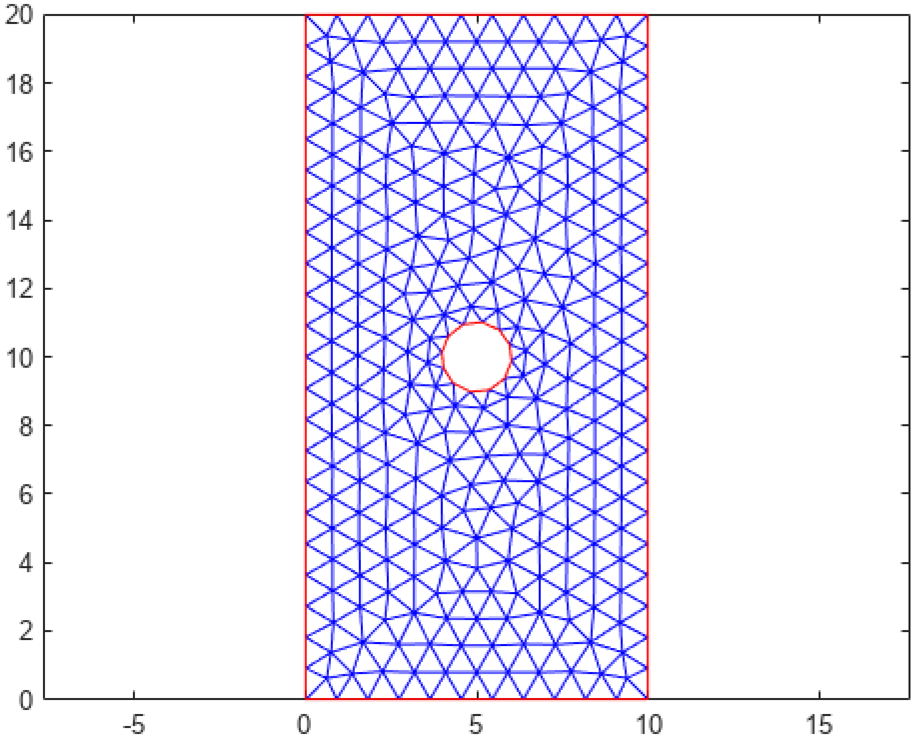
Import and plot a geometry.

```
g = importGeometry(model,"PlateHolePlanar.stl");
pdegplot(g)
```



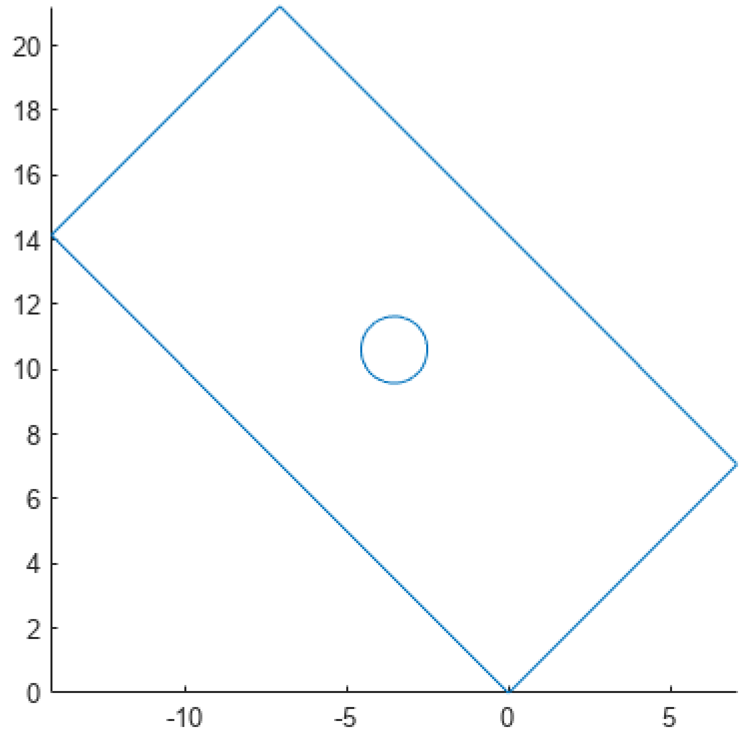
Mesh the geometry and plot the mesh.

```
generateMesh(model);

figure
pdemesh(model)
```



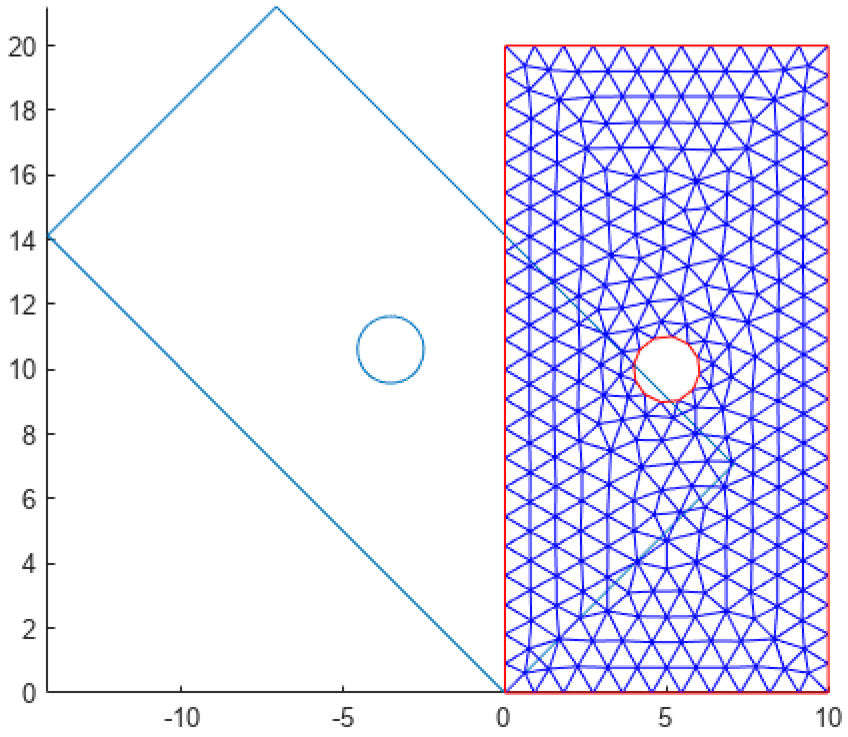
Rotate the geometry around the default *z*-axis by 45 degrees. Plot the result.

```
rotate(g,45);

figure
pdegplot(g)
```



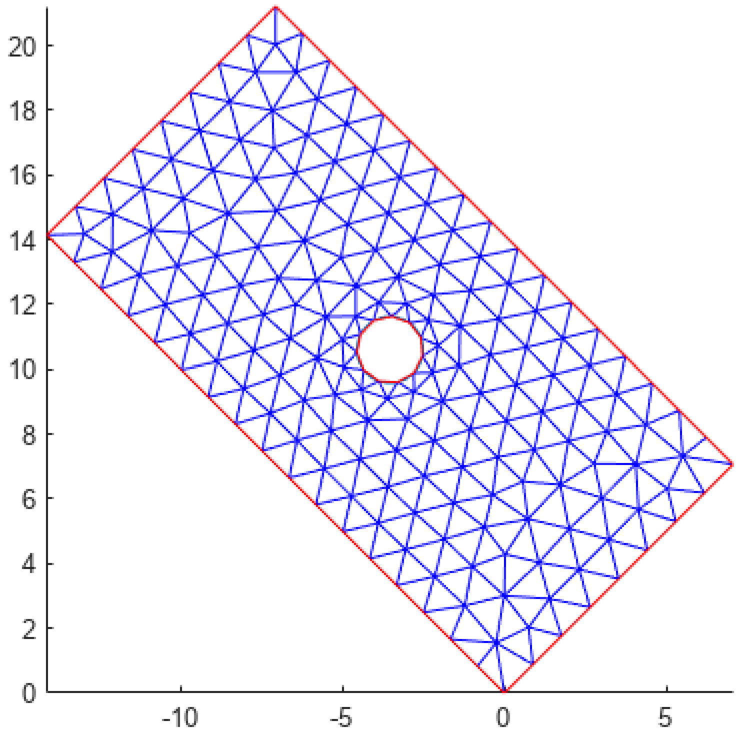
Plot the geometry and mesh. The `rotate` function modifies a geometry, but it does not modify a mesh.

```
figure
pdegplot(g)
hold on
pdemesh(model)
```



After modifying the geometry, always regenerate the mesh.

```
generateMesh(model);

figure
pdegplot(g)
hold on
pdemesh(model)
```



Restore the original geometry position.

```
rotate(g,-45);
```

Rotate the geometry by the same angle, but this time use the center of the geometry as a reference point. The axis of rotation is the line in the z-direction passing through the reference point.

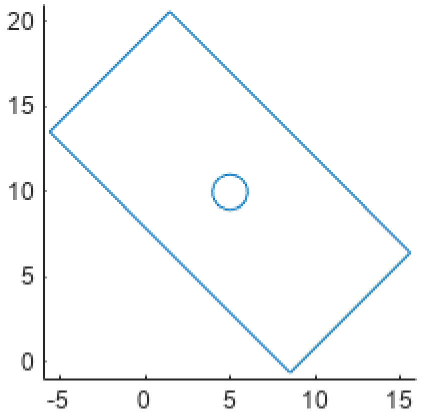
```
rotate(g,45,[5 10]);
```

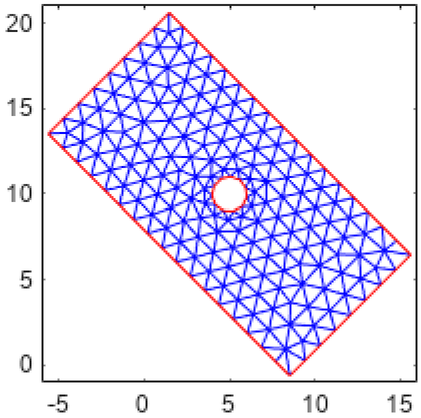
Regenerate the mesh.

```
generateMesh(model);
```

Plot the resulting geometry and mesh.

```
figure
subplot(1,2,1)
pdegplot(model)
axis([-6 16 -1 21])
subplot(1,2,2)
pdemesh(model)
axis([-6 16 -1 21])
```



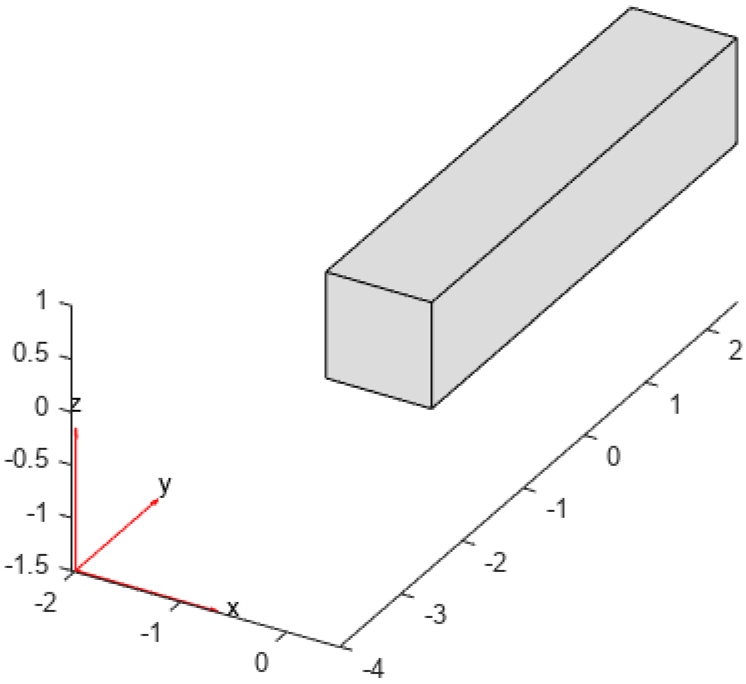


Rotate 3-D Geometry

Rotate a geometry with and without specifying the reference points for the axis of rotation.

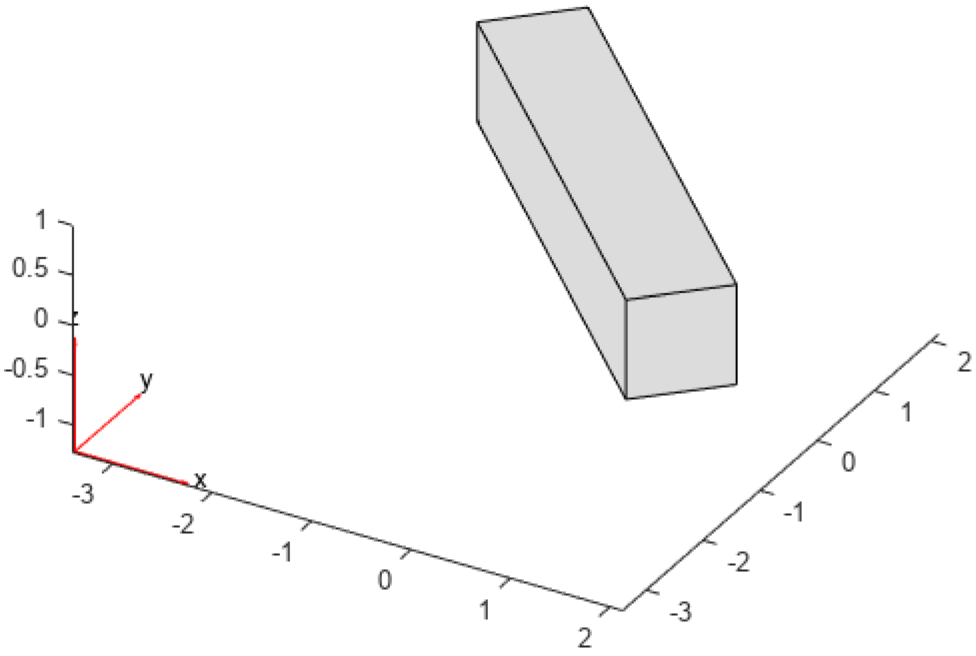
Create and plot a geometry.

```
g = multicuboid(1,5,1);
pdegplot(g)
```



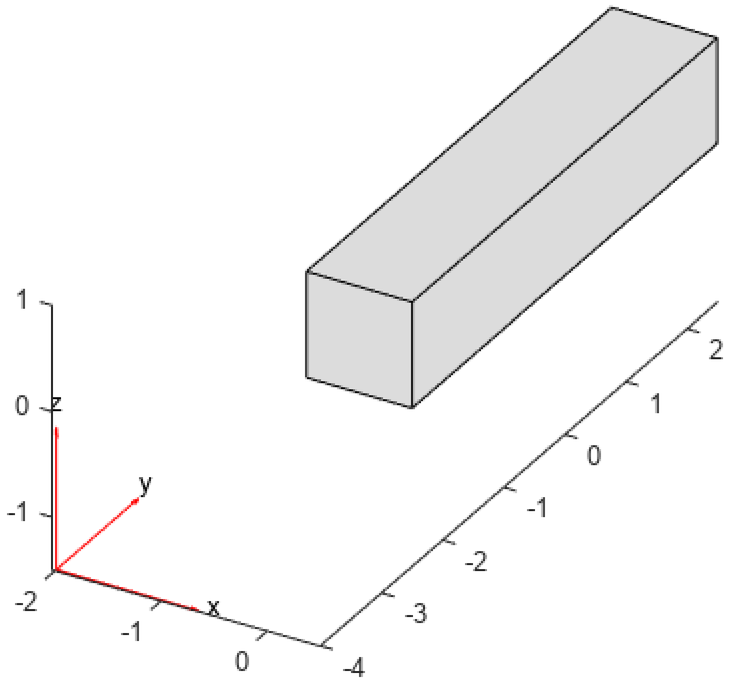
Rotate a 3-D geometry around the default *z*-axis by 45 degrees. Plot the result.

```
rotate(g,45);
pdegplot(g)
```



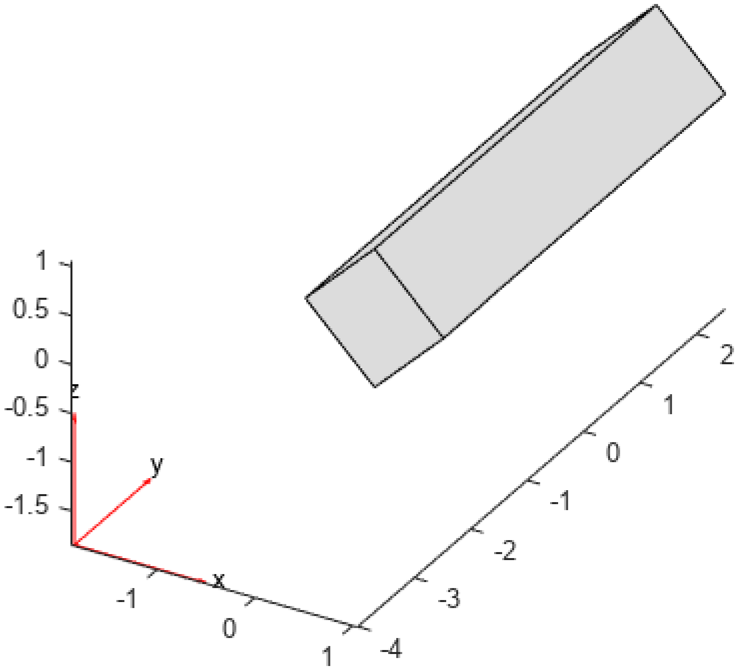
Restore the original geometry position.

```
rotate(g,-45);
pdegplot(g)
```



Rotate the geometry by the same angle, but this time around the *y*-axis.

```
rotate(g,45,[0 0 0],[0 1 0]);
pdegplot(g)
```



Input Arguments

g — Geometry
`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

Example: `g = model.Geometry`

`theta` — Rotation angle in degrees
real number

Rotation angle in degrees, specified as a real number.

Example: `rotate(g,90)`

`refpoint` — Reference point for rotation axis
vector of two or three real numbers

Reference point for a rotation axis, specified as a vector of two or three real numbers. The axis of rotation is the line in the *z*-direction passing through the reference point.

Example: `rotate(g,45,[1 1.5])`

`refpoint1,refpoint2` — Reference points that define rotation axis
vector of three real numbers

Reference points that define a rotation axis for a 3-D geometry, specified as a vector of three real numbers.

Example: `rotate(g,45,[0 0 0],[1 1 1])`

Output Arguments

h — Resulting geometry
handle

Resulting geometry, returned as a handle. If the original geometry `g` is a `DiscreteGeometry` object, then `h` is a handle to the modified `DiscreteGeometry` geometry object `g`. If `g` is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` geometry object. In this case, the original geometry `g` remains unchanged.

Tips

- `rotate` modifies a geometry, but it does not modify a mesh. After modifying a geometry, regenerate the mesh to ensure a proper mesh association with the new geometry.
- If `g` is an `AnalyticGeometry` object, and you want to replace it with the resulting discrete geometry, assign the output to the original geometry, for example, `g = rotate(g,90)`.

Version History

Introduced in R2020a

Geometry transformation for analytic geometries

`rotate` now works with `AnalyticGeometry` objects.

See Also

`scale` | `translate` | AnalyticGeometry Properties | DiscreteGeometry Properties | `pdegplot` | `importGeometry` | `geometryFromMesh` | `generateMesh`

scale

Package: `pde`

Scale geometry

Syntax

```
scale(g,s)
scale(g,s,refpoint)
```

```
h = scale(g, ___ )
```

Description

`scale(g,s)` scales the geometry `g` by the factor `s` with respect to the origin.

`scale(g,s,refpoint)` scales the geometry with respect to the reference point `refpoint`.

`h = scale(g, ___ )` returns a handle `h` to the resulting geometry object `g`. Use this syntax with any input arguments from the previous syntaxes.

If the original geometry is a `DiscreteGeometry` object, then the function modifies the original geometry and returns the handle `h` to the modified `DiscreteGeometry` geometry object. If the original geometry is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` geometry object. In this case, the original geometry remains unchanged.

Examples

Scale 2-D Geometry

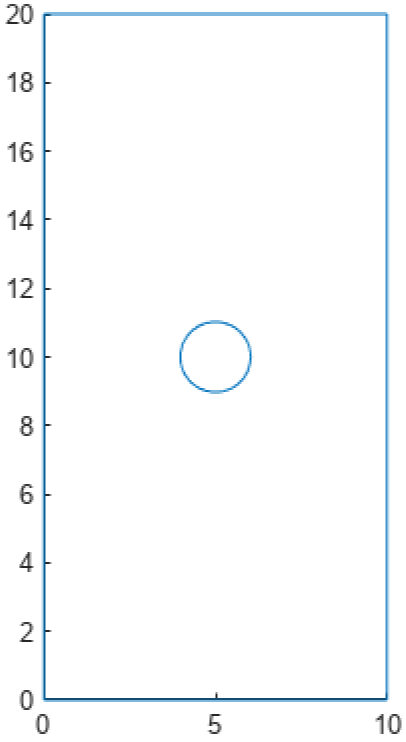
Scale a 2-D geometry along the *x*- and *y*-axis and ensure consistency with the mesh.

Create a model.

```
model = createpde;
```

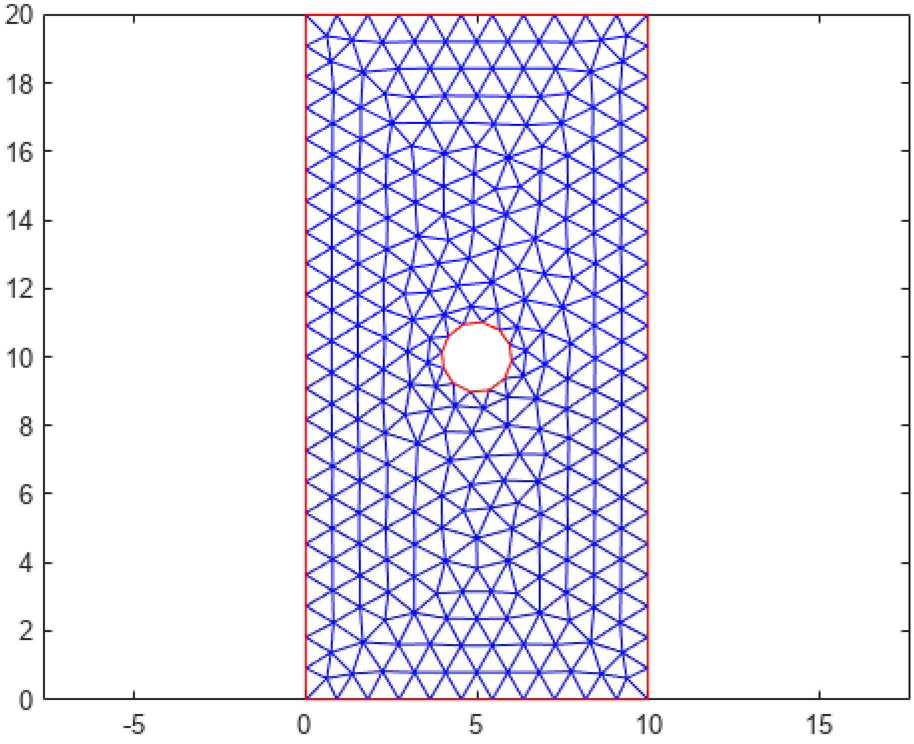
Import and plot a geometry.

```
g = importGeometry(model,"PlateHolePlanar.stl");
pdegplot(model)
```



Mesh the geometry and plot the mesh.

```
generateMesh(model);

figure
pdemesh(model)
```



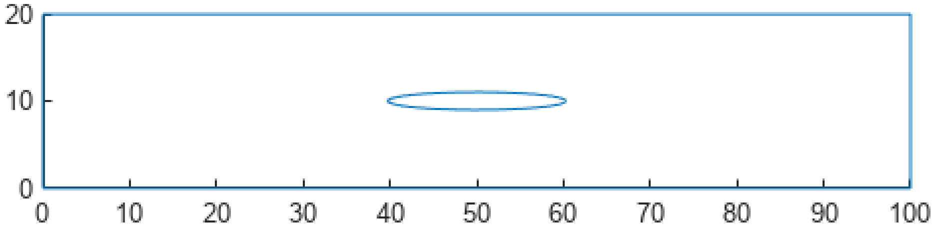
Scale the geometry by a factor of 10 along the *x*-axis.

```
scale(g,[10 1])
```

```
ans = 
  DiscreteGeometry with properties:

       NumCells: 0
       NumFaces: 1
       NumEdges: 5
    NumVertices: 5
       Vertices: [5x3 double]
```

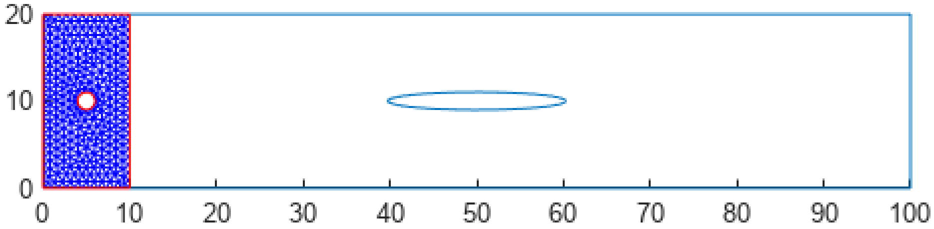
Plot the geometry.

```
figure
pdegplot(model)
```



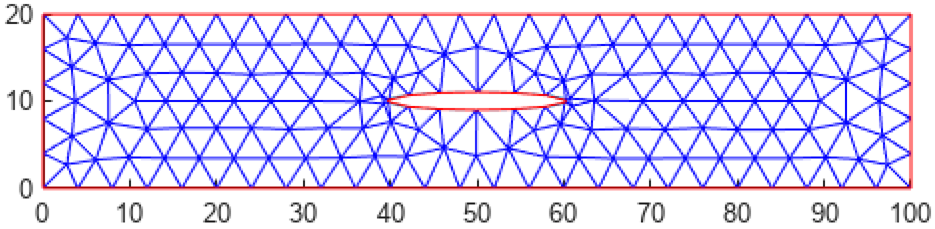
Plot the geometry and mesh. The `scale` function modifies a geometry, but it does not modify a mesh.

```
figure
pdegplot(model)
hold on
pdemesh(model)
```



After modifying the geometry, always regenerate the mesh.

```
generateMesh(model);

figure
pdegplot(model)
hold on
pdemesh(model)
```

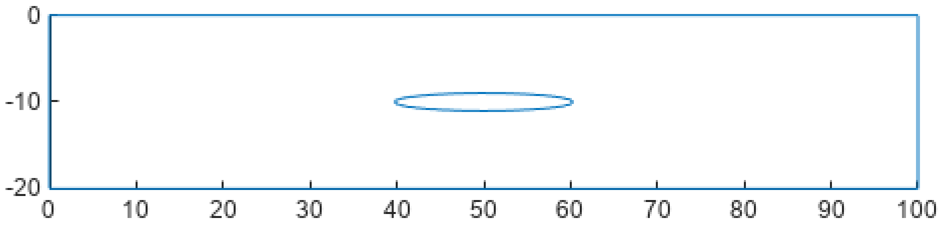


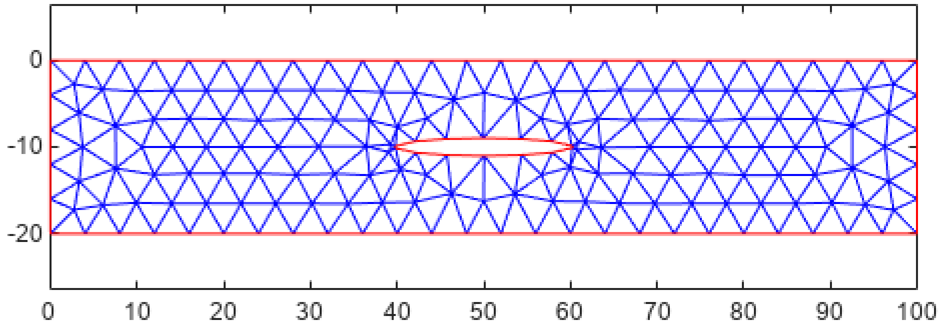
Reflect the geometry across the *x*-axis and regenerate the mesh.

```
scale(g,[1 -1]);
generateMesh(model);
```

Plot the resulting geometry and mesh.

```
figure
subplot(2,1,1)
pdegplot(model)
subplot(2,1,2)
pdemesh(model)
```



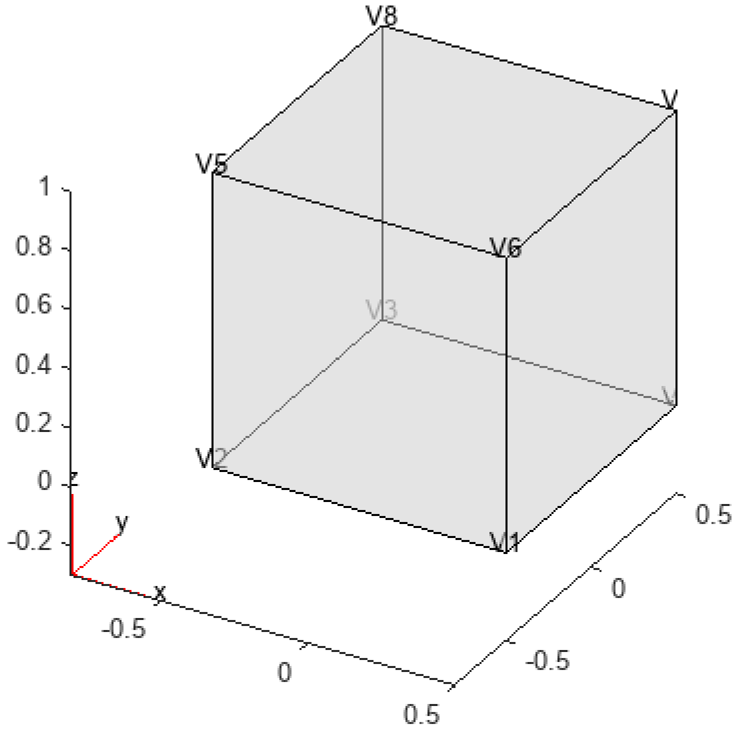


Scale 3-D Geometry

Enlarge a geometry: first uniformly in all directions, and then using different scaling factors along different axes.

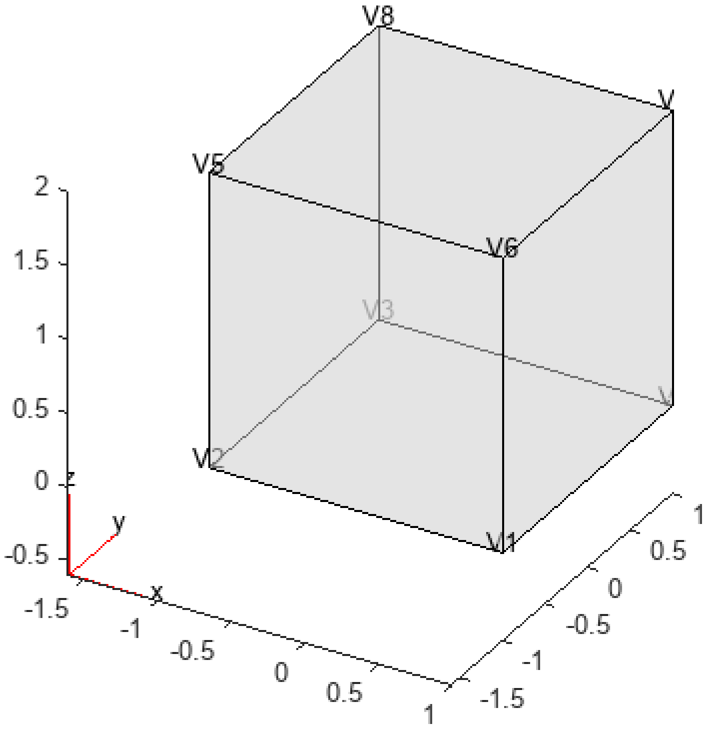
Create and plot a geometry.

```
g = multicuboid(1,1,1);
pdegplot(g,"VertexLabels","on","FaceAlpha",0.5)
```



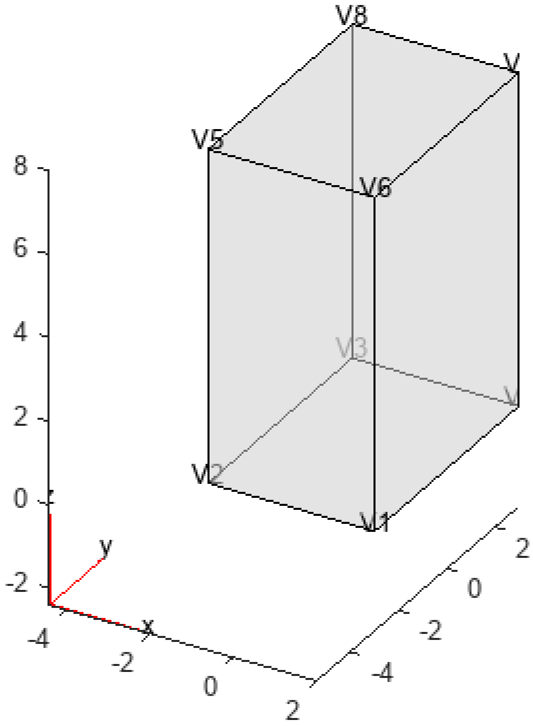
Scale the geometry by a factor of 2 uniformly along all coordinate axes. Plot the result.

```
scale(g,2);
pdegplot(g,"VertexLabels","on","FaceAlpha",0.5)
```



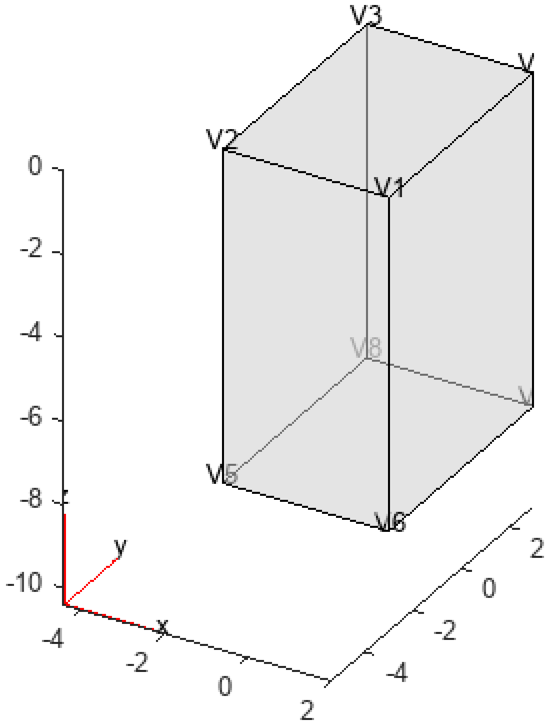
Now scale by factors of 2, 3, and 4 along the *x*-, *y*-, and *z*-axes, respectively. Plot the result.

```
scale(g,[2 3 4]);
pdegplot(g,"VertexLabels","on","FaceAlpha",0.5)
```



Flip the geometry upside down by scaling it with the factor `-1` and using the bottom front corner (vertex 1) as a reference point.

```
scale(g,[1 1 -1], [2 -3 0]);
pdegplot(g,"VertexLabels","on","FaceAlpha",0.5)
```



Input Arguments

g — Geometry
`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

s — Scaling factor
nonzero real number | vector of two or three nonzero real numbers

Scaling factor, specified as a real number or vector of two or three real numbers. Use one value for uniform scaling in all directions. Use a vector of two or three elements to specify different scaling factors along the *x*-, *y*-, and, for a 3-D geometry, *z*-axes.

`refpoint` — Reference point for scaling
vector of two or three real numbers

Reference point for scaling specified as a vector of two or three real numbers for a 2-D and 3-D geometry, respectively

Output Arguments

h — Resulting geometry
handle

Resulting geometry, returned as a handle. If the original geometry `g` is a `DiscreteGeometry` object, then `h` is a handle to the modified `DiscreteGeometry` geometry object `g`. If `g` is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` geometry object. In this case, the original geometry `g` remains unchanged.

Tips

- `scale` modifies a geometry, but it does not modify a mesh. After modifying a geometry, regenerate the mesh to ensure a proper mesh association with the new geometry.
- If the scaling factor is negative, then the coordinates will flip their signs. The scaling factor of `-1` mirrors the existing geometry if the reference point is the origin.
- If `g` is an `AnalyticGeometry` object, and you want to replace it with the resulting discrete geometry, assign the output to the original geometry, for example, `g = scale(g,20)`.

Version History

Introduced in R2020a

Geometry transformation for analytic geometries

`scale` now works with `AnalyticGeometry` objects.

See Also

`rotate` | `translate` | AnalyticGeometry Properties | DiscreteGeometry Properties | `pdegplot` | `importGeometry` | `geometryFromMesh` | `generateMesh`

setInitialConditions

Package: `pde`

Give initial conditions or initial solution

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)
```

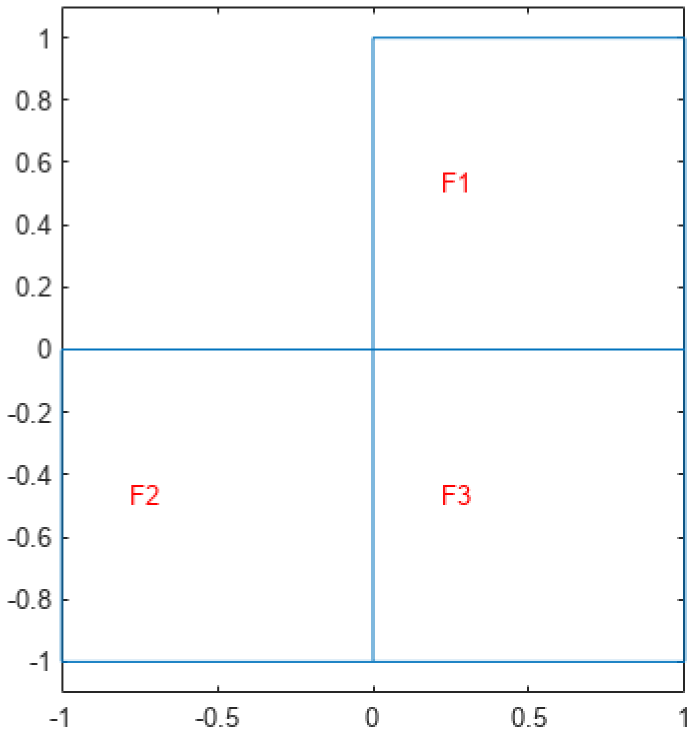
```
ans = 
  GeometricInitialConditions with properties:

           RegionType: 'cell'
             RegionID: 1
         InitialValue: [5x1 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.

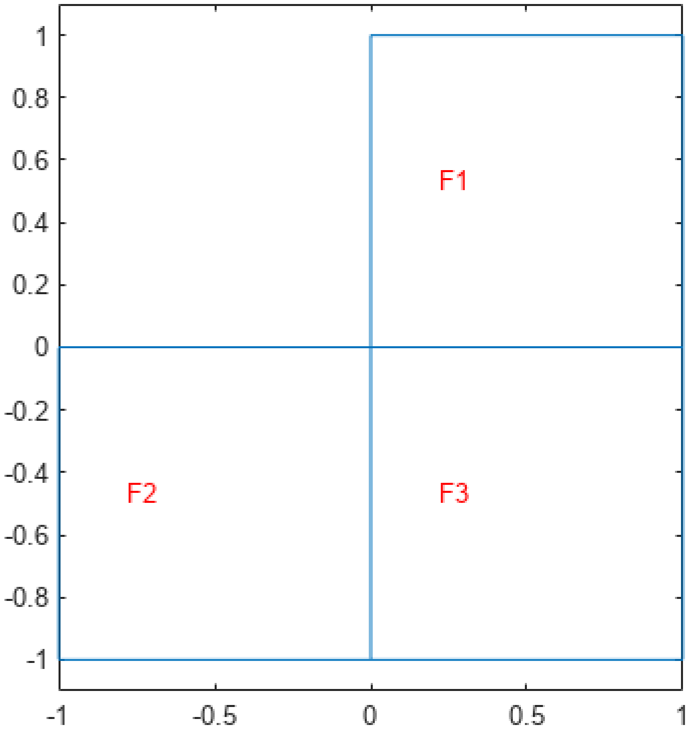
```
setInitialConditions(model,-1);
setInitialConditions(model,1,"Face",2);
setInitialConditions(model,5,"Face",3);
```

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 = @(location)location.x.^2 + location.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 = @(location)location.x.^2 - location.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.

$$ut0 = \begin{bmatrix} 4 + \dfrac{x}{x^2 + y^2 + z^2} \\ 5 - \tanh(z) \\ 10\dfrac{y}{x^2 + y^2 + z^2} \end{bmatrix}$$

To create this initial gradient, write a function file, and ensure that the function is on your MATLAB path.

```
function ut0 = ut0fun(location)

M = length(location.x);

ut0 = zeros(3,M);

denom = location.x.^2+location.y.^2+location.z.^2;

ut0(1,:) = 4 + location.x./denom;

ut0(2,:) = 5 - tanh(location.z);

ut0(3,:) = 10*location.y./denom;

end
```

Set the initial conditions.

```
setInitialConditions(model,0,@ut0fun)
```

```
ans =
  GeometricInitialConditions with properties:

           RegionType: 'cell'
             RegionID: 1
         InitialValue: 0
    InitialDerivative: @ut0fun
```

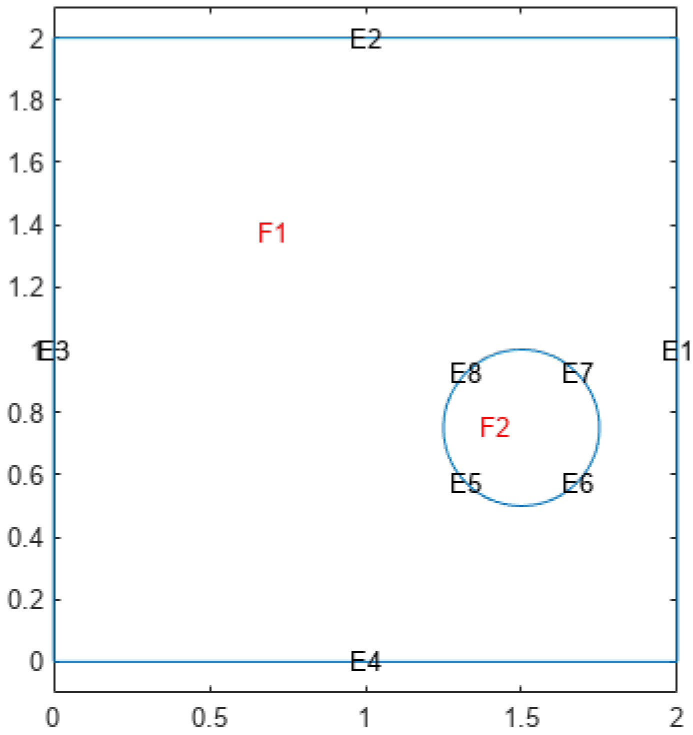
Initial Condition Is Previously Obtained Solution

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.

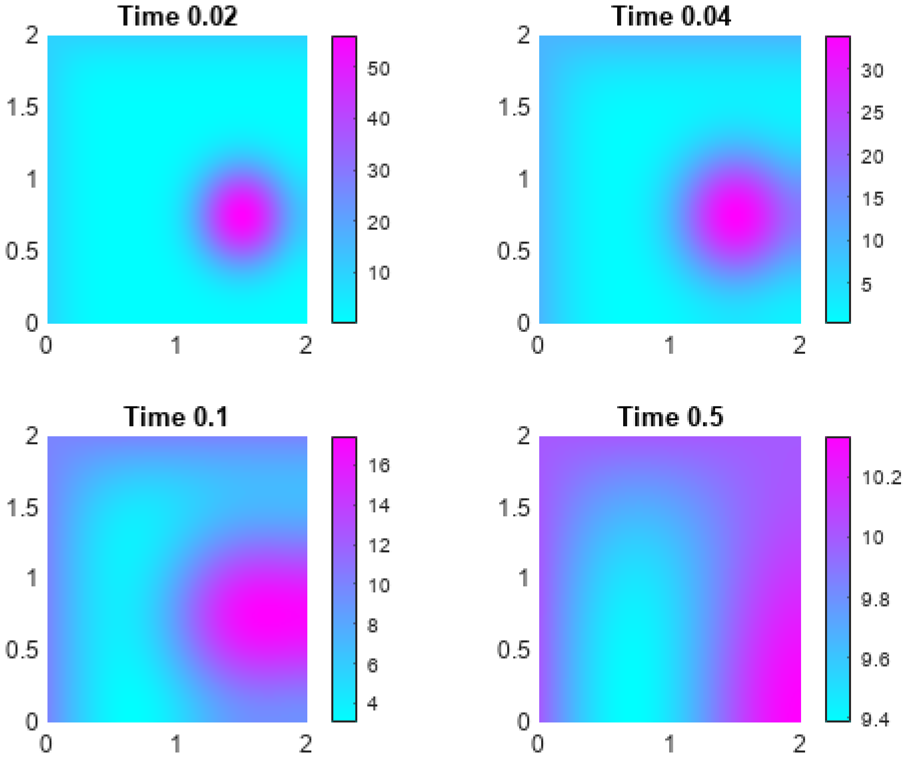
```
model = createpde();
geometryFromEdges(model,dl);

% 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)
```

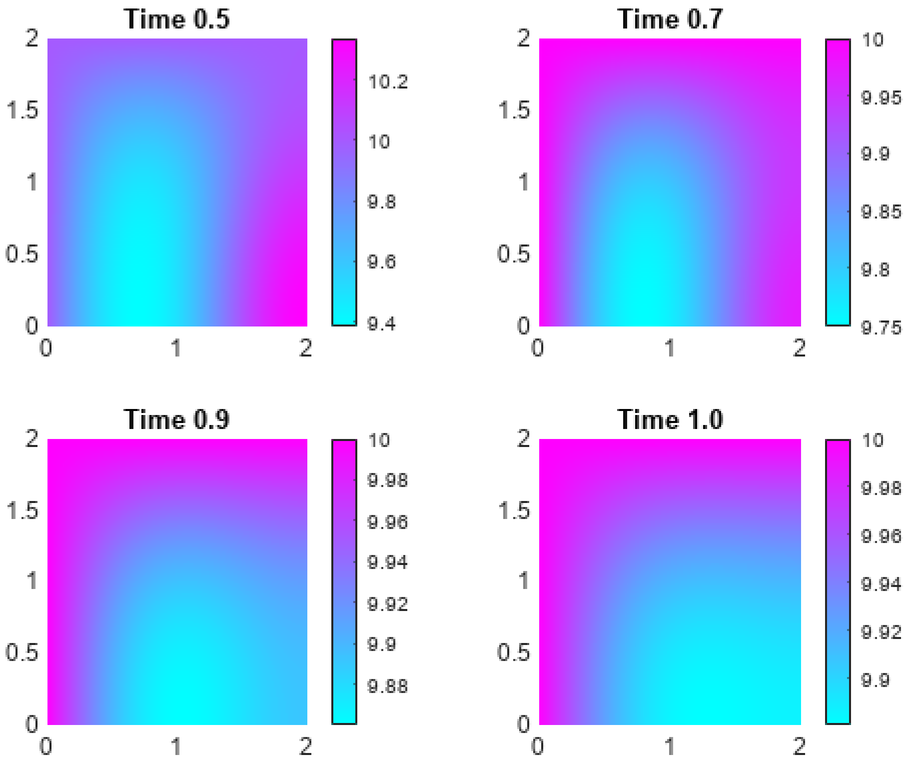
```
ans = 
  NodalInitialConditions with properties:

           InitialValue: [7289x1 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: [7289x1 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);
```

Input Arguments

`model` — PDE model

`PDEModel` object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde`

`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 PDE Toolbox" on page 1-3.

- 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(location)
  ```

  Solvers pass `location` as a structure with fields `location.x`, `location.y`, and, for 3-D problems, `location.z`. `initfun` must return a matrix `u0` of size *N*-by-`M`, where `M = length(location.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 PDE Toolbox" on page 1-3. 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(location)
  ```

  Solvers pass `location` as a structure with fields `location.x`, `location.y`, and, for 3-D problems, `location.z`. `initfun` must return a matrix `u0` of size *N*-by-`M`, where `M = length(location.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` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

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.

Version History

Introduced in R2016a

Nodal initial conditions

You can now set initial conditions at the mesh nodes by using the solution from a previous analysis on the same geometry and mesh.

See Also

`findInitialConditions` | `pdegplot` | `PDEModel`

Topics

"Set Initial Conditions" on page 2-114
"Set Initial Condition for Model with Fine Mesh Using Solution Obtained with Coarser Mesh" on page 2-121
"Nonlinear System with Cross-Coupling Between Components" on page 2-117
"Solve Problems Using PDEModel Objects" on page 2-2
"Equations You Can Solve Using PDE Toolbox" on page 1-3

solve

Package: `pde`

Solve heat transfer, structural analysis, or electromagnetic analysis problem

Syntax

```
structuralStaticResults = solve(structuralStatic)
structuralTransientResults = solve(structuralTransient,tlist)
structuralFrequencyResponseResults = solve(structuralFrequencyResponse,flist)

structuralModalResults = solve(structuralModal,"FrequencyRange",
[omega1,omega2])
structuralTransientResults = solve(structuralTransient,tlist,"ModalResults",
structuralModalR)
structuralFrequencyResponseResults = solve(structuralFrequencyResponse,
flist,"ModalResults",structuralModalR)

thermalSteadyStateResults = solve(thermalSteadyState)
thermalTransientResults = solve(thermalTransient,tlist)

thermalModalResults = solve(thermalModal,"DecayRange",[lambda1,lambda2])
thermalModalResults = solve(thermalModal,"Snapshots",Tmatrix)
thermalTransientResults = solve(thermalTransient,"ModalResults",
thermalModalR)

emagStaticResults = solve(emagmodel)
emagHarmonicResults = solve(emagmodel,"Frequency",omega)
```

Description

`structuralStaticResults = solve(structuralStatic)` returns the solution to the static structural analysis model represented in `structuralStatic`.

`structuralTransientResults = solve(structuralTransient,tlist)` returns the solution to the transient structural dynamics model represented in `structuralTransient` at the times specified in `tlist`.

`structuralFrequencyResponseResults = solve(structuralFrequencyResponse,flist)` returns the solution to the frequency response model represented in `structuralFrequencyResponse` at the frequencies specified in `flist`.

`structuralModalResults = solve(structuralModal,"FrequencyRange", [omega1,omega2])` returns the solution to the modal analysis model for all modes in the frequency range `[omega1,omega2]`. Define `omega1` as slightly lower than the lowest expected frequency and `omega2` as slightly higher than the highest expected frequency. For example, if the lowest expected frequency is zero, then use a small negative value for `omega1`.

`structuralTransientResults = solve(structuralTransient,tlist,"ModalResults", structuralModalR)` and `structuralFrequencyResponseResults = solve(`

`structuralFrequencyResponse,flist,"ModalResults",structuralModalR)` solves a transient and a frequency response structural model, respectively, by using the modal superposition method to speed up computations. First, perform modal analysis to compute natural frequencies and mode shapes in a particular frequency range. Then, use this syntax to invoke the modal superposition method. The accuracy of the results depends on the modes in the modal analysis results.

`thermalSteadyStateResults = solve(thermalSteadyState)` returns the solution to the steady-state thermal model represented in `thermalSteadyState`.

`thermalTransientResults = solve(thermalTransient,tlist)` returns the solution to the transient thermal model represented in `thermalTransient` at the times specified in `tlist`.

`thermalModalResults = solve(thermalModal,"DecayRange",[lambda1,lambda2])` performs an eigen decomposition of a linear thermal model `thermalModal` for all modes in the decay range `[lambda1,lambda2]`. The resulting modes enable you to:

- Use the modal superposition method to speed up a transient thermal analysis.
- Extract the reduced modal system to use, for example, in Simulink®.

`thermalModalResults = solve(thermalModal,"Snapshots",Tmatrix)` obtains the modal basis of a linear or nonlinear thermal model using proper orthogonal decomposition (POD). You can use the resulting modes to speed up a transient thermal analysis or, if your thermal model is linear, to extract the reduced modal system.

`thermalTransientResults = solve(thermalTransient,"ModalResults", thermalModalR)` solves a transient thermal model by using the modal superposition method to speed up computations. First, perform modal decomposition to compute mode shapes for a particular decay range. Then, use this syntax to invoke the modal superposition method. The accuracy of the results depends on the modes in the modal analysis results.

`emagStaticResults = solve(emagmodel)` returns the solution to the electrostatic, magnetostatic, or DC conduction model represented in `emagmodel`.

`emagHarmonicResults = solve(emagmodel,"Frequency",omega)` returns the solution to the harmonic electromagnetic analysis model represented in `emagmodel` at the frequencies specified in `omega`.

Examples

Solution to Static Structural Model

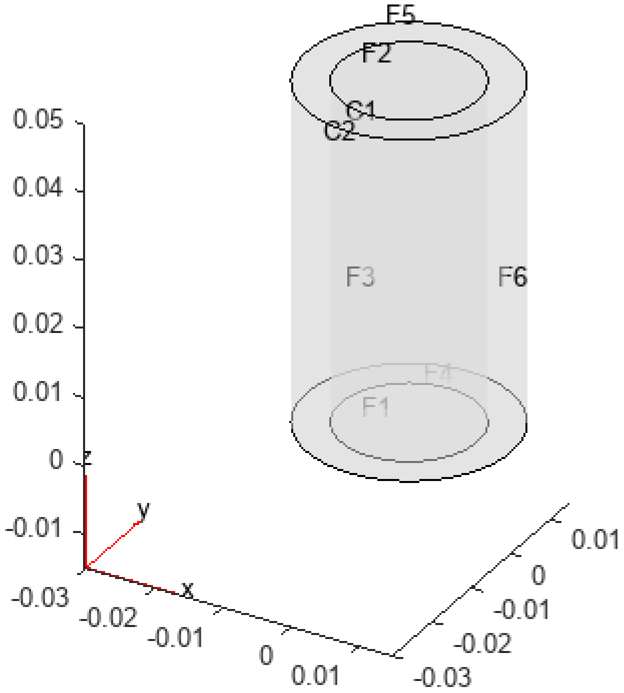
Solve a static structural model representing a bimetallic cable under tension.

Create a static structural model for a solid (3-D) problem.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01 0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

```
            Mesh: [1x1 FEMesh]
```

The solver finds the values of the displacement, stress, strain, and von Mises stress at the nodal locations. To access these values, use `structuralresults.Displacement`, `structuralresults.Stress`, and so on. The displacement, stress, and strain values at the nodal locations are returned as `FEStruct` objects with the properties representing their components. Note that properties of an `FEStruct` object are read-only.

```
structuralresults.Displacement
```

```
ans = 
  FEStruct with properties:

           ux: [22281x1 double]
           uy: [22281x1 double]
           uz: [22281x1 double]
    Magnitude: [22281x1 double]
```

```
structuralresults.Stress
```

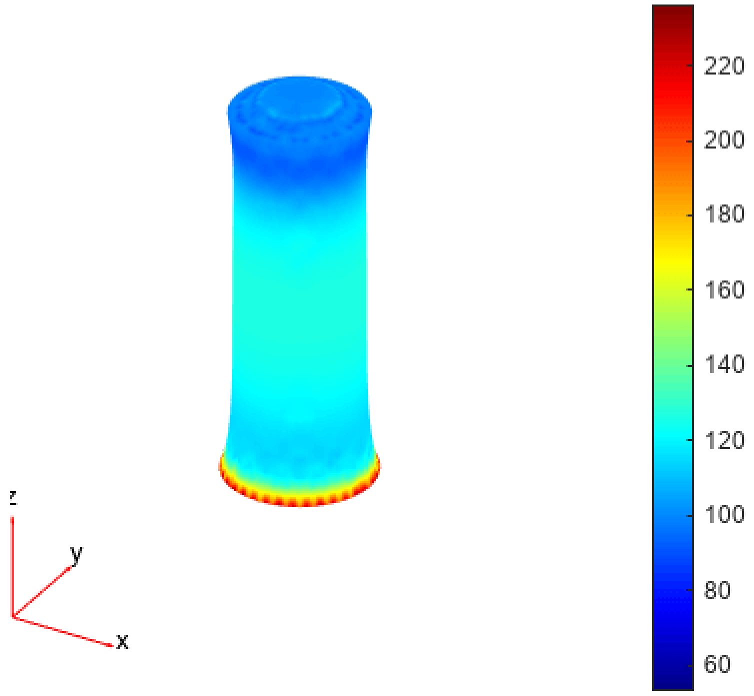
```
ans = 
  FEStruct with properties:

    sxx: [22281x1 double]
    syy: [22281x1 double]
    szz: [22281x1 double]
    syz: [22281x1 double]
    sxz: [22281x1 double]
    sxy: [22281x1 double]
```

```
structuralresults.Strain
```

```
ans = 
  FEStruct with properties:

    exx: [22281x1 double]
    eyy: [22281x1 double]
    ezz: [22281x1 double]
    eyz: [22281x1 double]
    exz: [22281x1 double]
    exy: [22281x1 double]
```

Plot the deformed shape with the *z*-component of normal stress.

```
pdeplot3D(structuralmodel, ...
          "ColorMapData",structuralresults.Stress.szz, ...
          "Deformation",structuralresults.Displacement)
```



Solution to Transient Structural Model

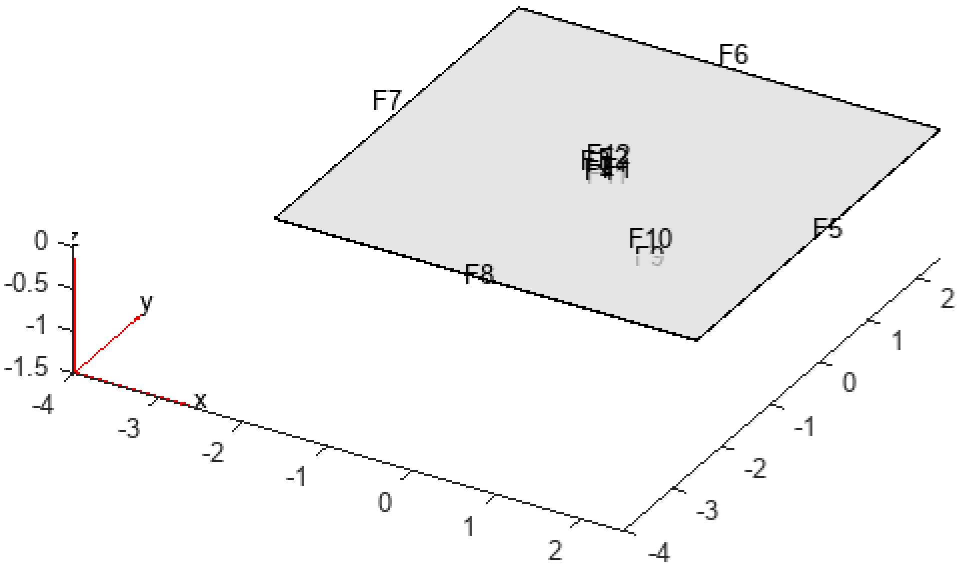
Solve for the transient response of a thin 3-D plate under a harmonic load at the center.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

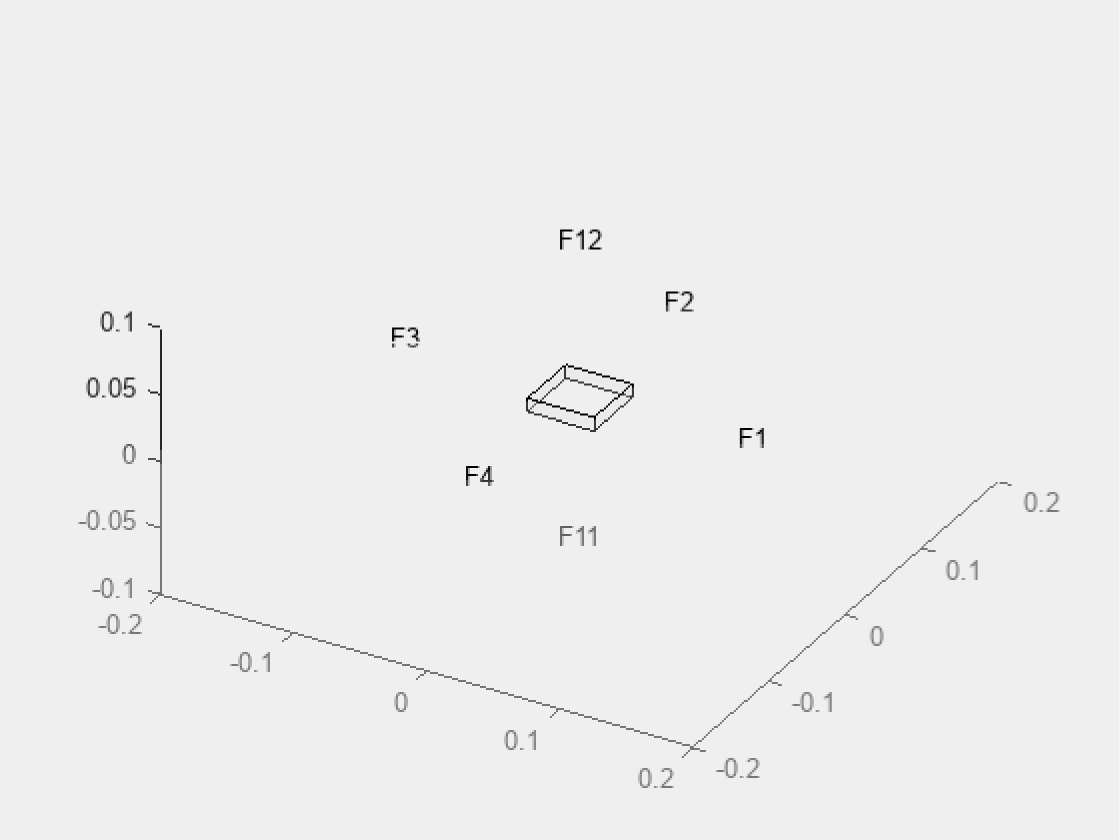
Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid([5,0.05],[5,0.05],0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Zoom in to see the face labels on the small plate at the center.

```
figure
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.25)
axis([-0.2 0.2 -0.2 0.2 -0.1 0.1])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Specify that all faces on the periphery of the thin 3-D plate are fixed boundaries.

```
structuralBC(structuralmodel,"Constraint","fixed","Face",5:8);
```

Apply a sinusoidal pressure load on the small face at the center of the plate.

```
structuralBoundaryLoad(structuralmodel,"Face",12, ...
                                       "Pressure",5E7, ...
                                       "Frequency",25);
```

Generate a mesh with linear elements.

```
generateMesh(structuralmodel,"GeometricOrder","linear","Hmax",0.2);
```

Specify zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

```
tlist = linspace(0,1,300);
structuralresults = solve(structuralmodel,tlist);
```

The solver finds the values of the displacement, velocity, and acceleration at the nodal locations. To access these values, use `structuralresults.Displacement`, `structuralresults.Velocity`, and so on. The displacement, velocity, and acceleration values are returned as `FEStruct` objects with the properties representing their components. Note that properties of an `FEStruct` object are read-only.

```
structuralresults.Displacement
```

```
ans = 
  FEStruct with properties:

           ux: [1873x300 double]
           uy: [1873x300 double]
           uz: [1873x300 double]
    Magnitude: [1873x300 double]
```

```
structuralresults.Velocity
```

```
ans = 
  FEStruct with properties:

           vx: [1873x300 double]
           vy: [1873x300 double]
           vz: [1873x300 double]
    Magnitude: [1873x300 double]
```

```
structuralresults.Acceleration
```

```
ans = 
  FEStruct with properties:

           ax: [1873x300 double]
           ay: [1873x300 double]
           az: [1873x300 double]
    Magnitude: [1873x300 double]
```

Frequency Response Analysis

Perform frequency response analysis of a tuning fork.

First, create a structural model for modal analysis of a solid tuning fork.

```
model = createpde("structural","frequency-solid");
```

Import the tuning fork geometry.

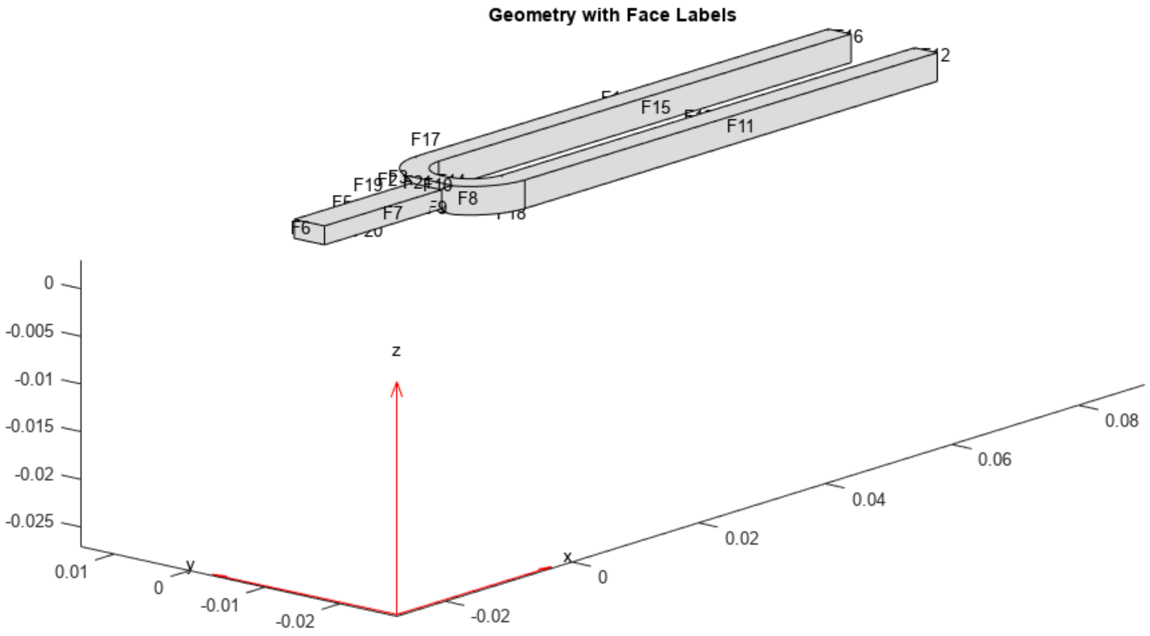
```
importGeometry(model,"TuningFork.stl");
```

Specify Young's modulus, Poisson's ratio, and the mass density to model linear elastic material behavior. Specify all physical properties in consistent units.

```
structuralProperties(model,"YoungsModulus",210E9, ...
                           "PoissonsRatio",0.3, ...
                           "MassDensity",8000);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face labels.

```
figure("units","normalized","outerposition",[0 0 1 1])
pdegplot(model,"FaceLabels","on")
view(-50,15)
title("Geometry with Face Labels")
```



Impose sufficient boundary constraints to prevent rigid body motion under applied loading. Typically, you hold a tuning fork by hand or mount it on a table. To create a simple approximation of this boundary condition, fix a region near the intersection of tines and the handle (faces 21 and 22).

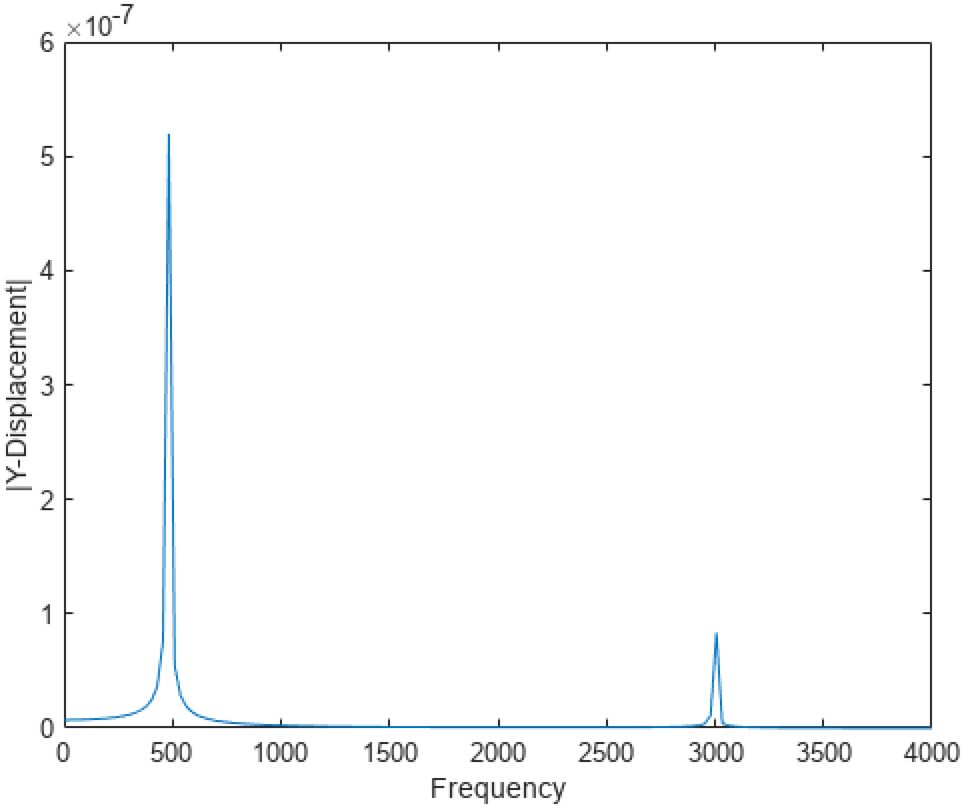
```
structuralBC(model,"Face",[21,22],"Constraint","fixed");
```

Specify the pressure loading on a tine (face 11) as a short rectangular pressure pulse. In the frequency domain, this pressure pulse is a unit load uniformly distributed across all frequencies.

```
structuralBoundaryLoad(model,"Face",11,"Pressure",1);
flist = linspace(0,4000,150);
mesh = generateMesh(model,"Hmax",0.005);
R = solve(model,2*pi*flist);
```

Plot the vibration frequency of the tine tip, which is face 12. Find nodes on the tip face and plot the *y*-component of the displacement over the frequency, using one of these nodes.

```
excitedTineTipNodes = findNodes(mesh,"region","Face",12);
tipDisp = R.Displacement.uy(excitedTineTipNodes(1),:);

figure
plot(flist,abs(tipDisp))
xlabel("Frequency");
ylabel("|Y-Displacement|");
```



Solution to Modal Analysis Structural Model

Find the fundamental (lowest) mode of a 2-D cantilevered beam, assuming prevalence of the plane-stress condition.

Specify geometric and structural properties of the beam, along with a unit plane-stress thickness.

```
length = 5;
height = 0.1;
E = 3E7;
nu = 0.3;
rho = 0.3/386;
```

Create a modal plane-stress model, assign a geometry, and generate a mesh.

```
structuralmodel = createpde("structural","modal-planestress");
gdm = [3;4;0;length;length;0;0;0;height;height];
g = decsg(gdm,'S1',('S1')');
geometryFromEdges(structuralmodel,g);
```

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

```
hmax = height/5;
msh=generateMesh(structuralmodel,"Hmax",hmax);
```

Specify the structural properties and boundary constraints.

```
structuralProperties(structuralmodel,"YoungsModulus",E, ...
                                     "MassDensity",rho, ...
                                     "PoissonsRatio",nu);
structuralBC(structuralmodel,"Edge",4,"Constraint","fixed");
```

Compute the analytical fundamental frequency (Hz) using the beam theory.

```
I = height^3/12;
analyticalOmega1 = 3.516*sqrt(E*I/(length^4*(rho*height)))/(2*pi)
```

```
analyticalOmega1 = 126.9498
```

Specify a frequency range that includes an analytically computed frequency and solve the model.

```
modalresults = solve(structuralmodel,"FrequencyRange",[0,1e6])
```

```
modalresults = 
  ModalStructuralResults with properties:

    NaturalFrequencies: [32x1 double]
            ModeShapes: [1x1 FEStruct]
                  Mesh: [1x1 FEMesh]
```

The solver finds natural frequencies and modal displacement values at nodal locations. To access these values, use `modalresults.NaturalFrequencies` and `modalresults.ModeShapes`.

```
modalresults.NaturalFrequencies/(2*pi)
```

```
ans = 32×1
10^5 ×

    0.0013
    0.0079
    0.0222
    0.0433
    0.0711
    0.0983
    0.1055
```

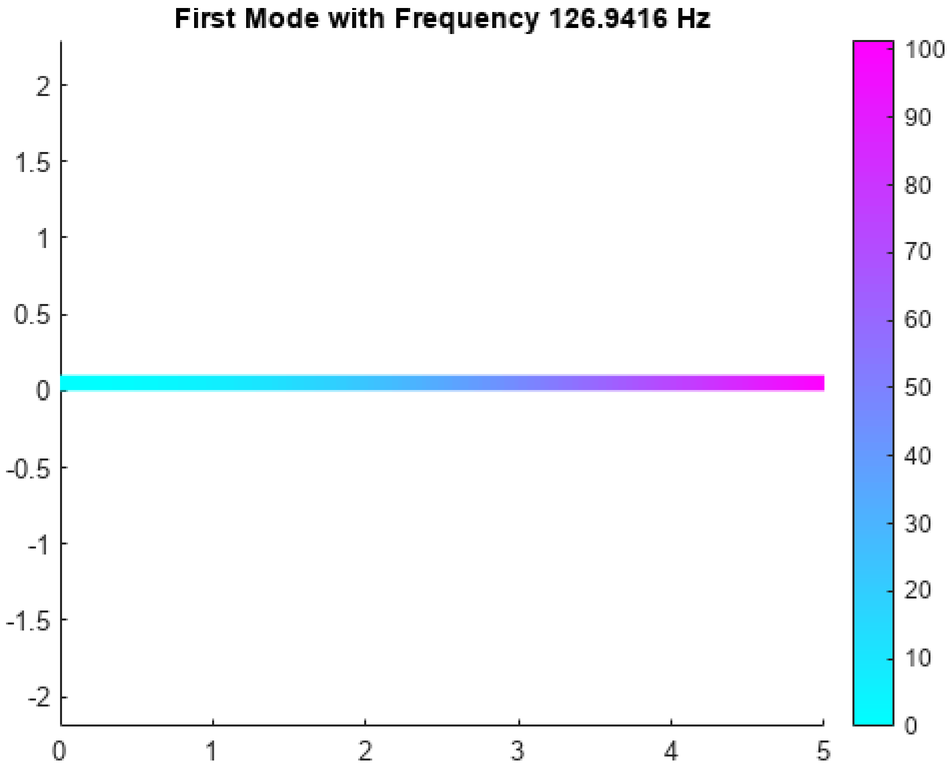
```
    0.1462
    0.1930
    0.2455
      ⋮
```

```
modalresults.ModeShapes
```

```
ans = 
  FEStruct with properties:

           ux: [6511x32 double]
           uy: [6511x32 double]
    Magnitude: [6511x32 double]
```

Plot the *y*-component of the solution for the fundamental frequency.

```
pdeplot(structuralmodel,"XYData",modalresults.ModeShapes.uy(:,1))
title(['First Mode with Frequency ', ...
        num2str(modalresults.NaturalFrequencies(1)/(2*pi)),' Hz'])
axis equal
```



Solution to Transient Structural Model Using Modal Superposition Method

Solve for the transient response at the center of a 3-D beam under a harmonic load on one of its corners.

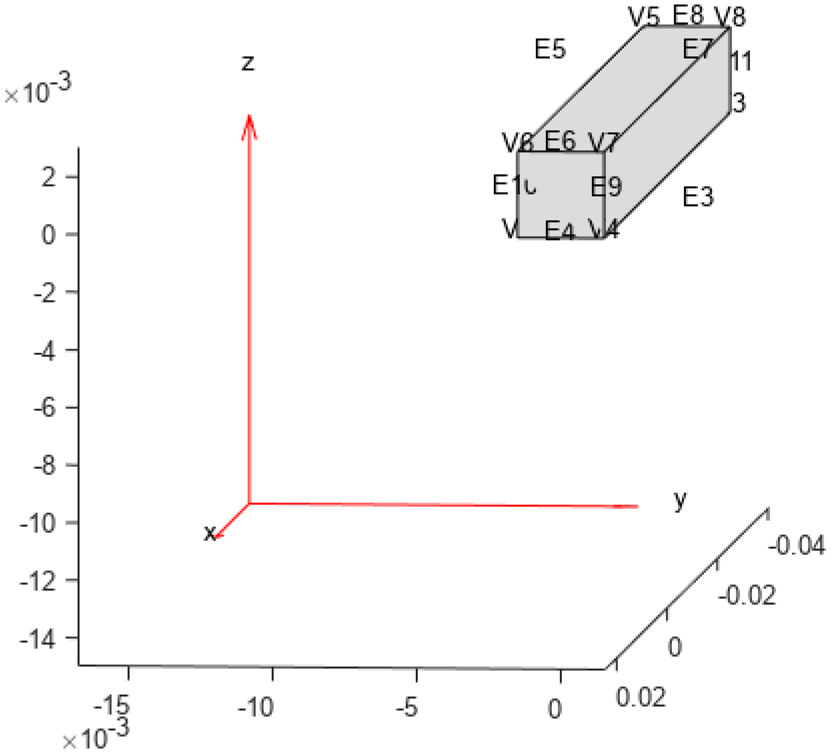
Modal Analysis

Create a modal analysis model for a 3-D problem.

```
structuralmodel = createpde("structural","modal-solid");
```

Create the geometry and include it in the model. Plot the geometry and display the edge and vertex labels.

```
gm = multicuboid(0.05,0.003,0.003);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"EdgeLabels","on", ...
                         "VertexLabels","on");
view([95 5])
```



Generate a mesh.

```
msh = generateMesh(structuralmodel);
```

Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Specify minimal constraints on one end of the beam to prevent rigid body modes. For example, specify that edge 4 and vertex 7 are fixed boundaries.

```
structuralBC(structuralmodel,"Edge",4,"Constraint","fixed");
structuralBC(structuralmodel,"Vertex",7,"Constraint","fixed");
```

Solve the problem for the frequency range from 0 to 500,000. The recommended approach is to use a value that is slightly lower than the expected lowest frequency. Thus, use `-0.1` instead of `0`.

```
Rm = solve(structuralmodel,"FrequencyRange",[-0.1,500000]);
```

Transient Analysis

Switch the analysis type of the model to transient.

```
structuralmodel.AnalysisType = "transient-solid";
```

Apply a sinusoidal force on the corner opposite of the constrained edge and vertex.

```
structuralBoundaryLoad(structuralmodel,"Vertex",5, ...
                                       "Force",[0,0,10], ...
                                       "Frequency",7600);
```

Specify zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Velocity",[0;0;0], ...
                             "Displacement",[0;0;0]);
```

Specify the relative and absolute tolerances for the solver.

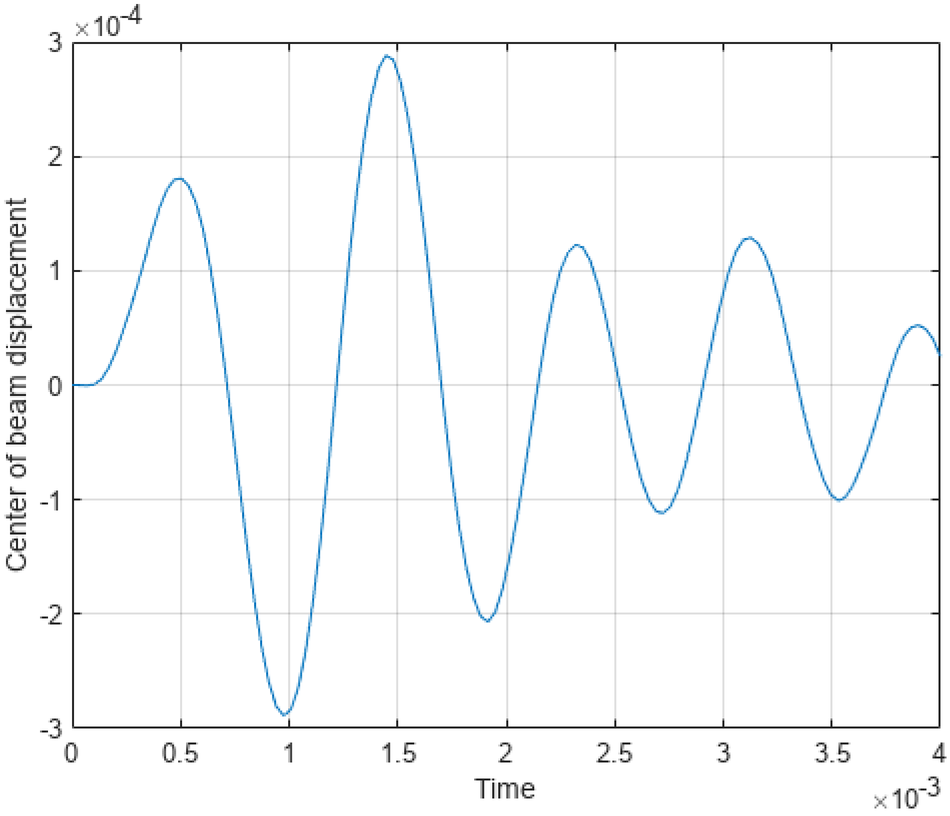
```
structuralmodel.SolverOptions.RelativeTolerance = 1E-5;
structuralmodel.SolverOptions.AbsoluteTolerance = 1E-9;
```

Solve the model using the modal results.

```
tlist = linspace(0,0.004,120);
Rdm = solve(structuralmodel,tlist,"ModalResults",Rm);
```

Interpolate and plot the displacement at the center of the beam.

```
intrpUdm = interpolateDisplacement(Rdm,0,0,0.0015);

plot(Rdm.SolutionTimes,intrpUdm.uz)
grid on
xlabel("Time");
ylabel("Center of beam displacement")
```



Expansion of Cantilever Beam Under Thermal Load

Find the deflection of a 3-D cantilever beam under a nonuniform thermal load. Specify the thermal load on the structural model using the solution from a transient thermal analysis on the same geometry and mesh.

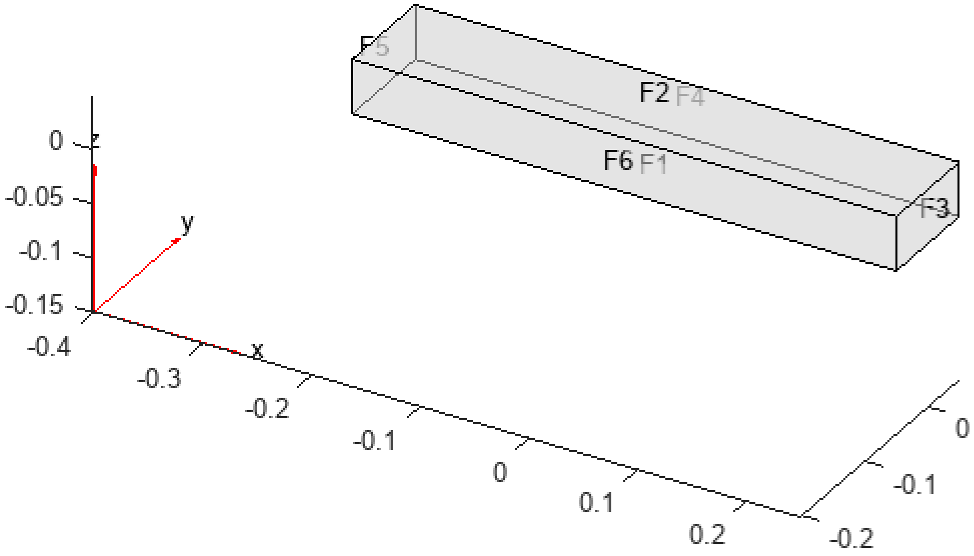
Transient Thermal Model Analysis

Create a transient thermal model.

```
thermalmodel = createpde("thermal","transient");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.1,0.05);
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Generate a mesh.

```
mesh = generateMesh(thermalmodel);
```

Specify the thermal properties of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",5e-3, ...
                               "MassDensity",2.7*10^(-6), ...
                               "SpecificHeat",10);
```

Specify the constant temperatures applied to the left and right ends of the beam.

```
thermalBC(thermalmodel,"Face",3,"Temperature",100);
thermalBC(thermalmodel,"Face",5,"Temperature",0);
```

Specify the heat source over the entire geometry.

```
internalHeatSource(thermalmodel,10);
```

Set the initial temperature.

```
thermalIC(thermalmodel,0);
```

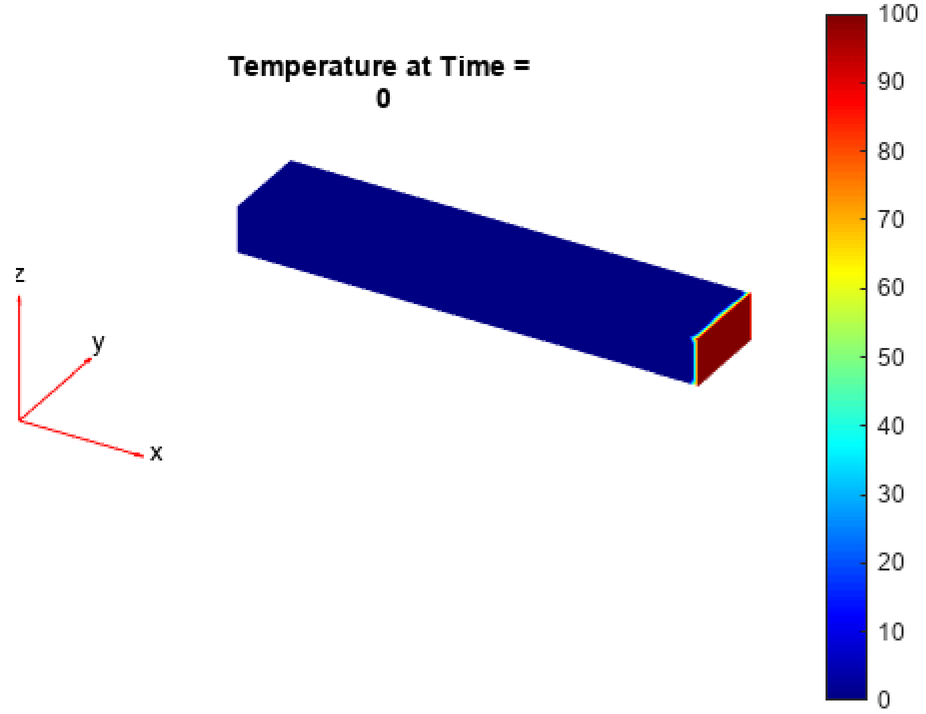
Solve the model.

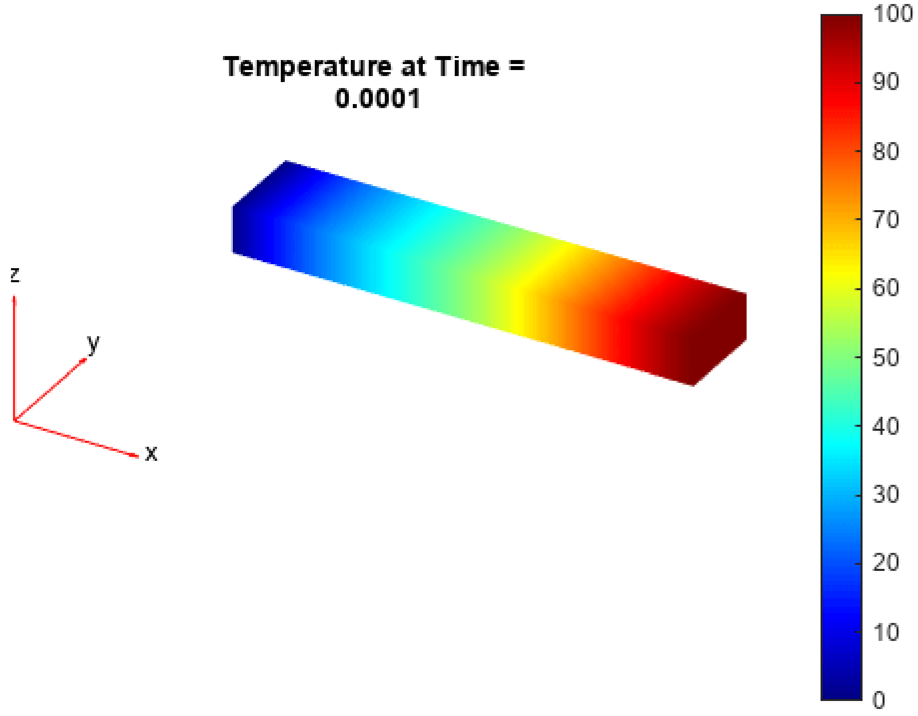
```
tlist = [0:1e-4:2e-4];
thermalresults = solve(thermalmodel,tlist)
```

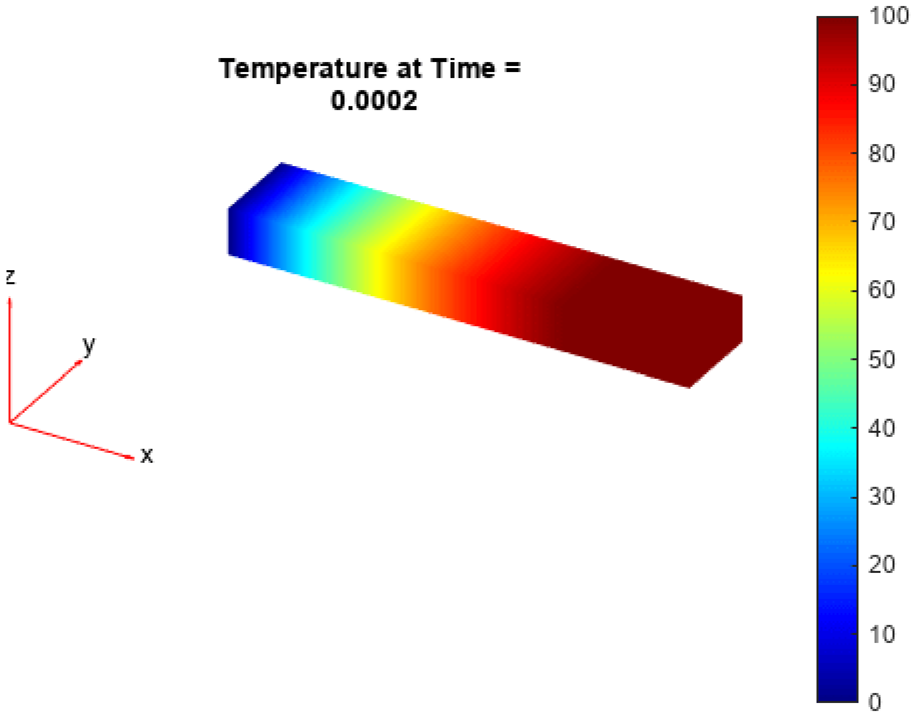
```
thermalresults = 
  TransientThermalResults with properties:

      Temperature: [3870x3 double]
    SolutionTimes: [0 1.0000e-04 2.0000e-04]
       XGradients: [3870x3 double]
       YGradients: [3870x3 double]
       ZGradients: [3870x3 double]
             Mesh: [1x1 FEMesh]
```

Plot the temperature distribution for each time step.

```
for n = 1:numel(thermalresults.SolutionTimes)
    figure
    pdeplot3D(thermalmodel, ...
                "ColorMapData", ...
                thermalresults.Temperature(:,n))
    title(["Temperature at Time = " ...
                num2str(tlist(n))])
    caxis([0 100])
end
```




Temperature at Time =
0.0001
z
y
x
100
90
80
70
60
50
40
30
20
10
0



Structural Analysis with Thermal Load

Create a static structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Include the same geometry as for the thermal model.

```
structuralmodel.Geometry = gm;
```

Use the same mesh that you used to obtain the thermal solution.

```
structuralmodel.Mesh = mesh;
```

Specify Young's modulus, Poisson's ratio, and the coefficient of thermal expansion.

```
structuralProperties(structuralmodel,"YoungsModulus",1e10, ...
                                     "PoissonsRatio",0.3, ...
                                     "CTE",11.7e-6);
```

Apply a fixed boundary condition on face 5.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a body load using the transient thermal model solution. By default, `structuralBodyLoad` uses the solution for the last time step.

```
structuralBodyLoad(structuralmodel,"Temperature",thermalresults);
```

Specify the reference temperature.

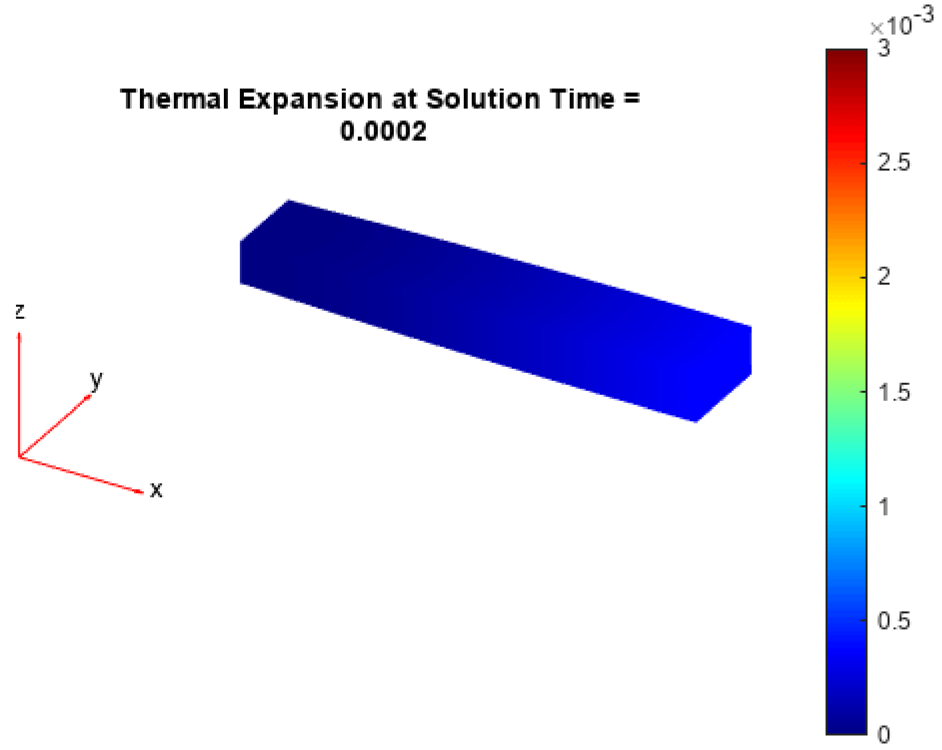
```
structuralmodel.ReferenceTemperature = 10;
```

Solve the structural model.

```
thermalstressresults = solve(structuralmodel);
```

Plot the deformed shape of the beam corresponding to the last step of the transient thermal model solution.

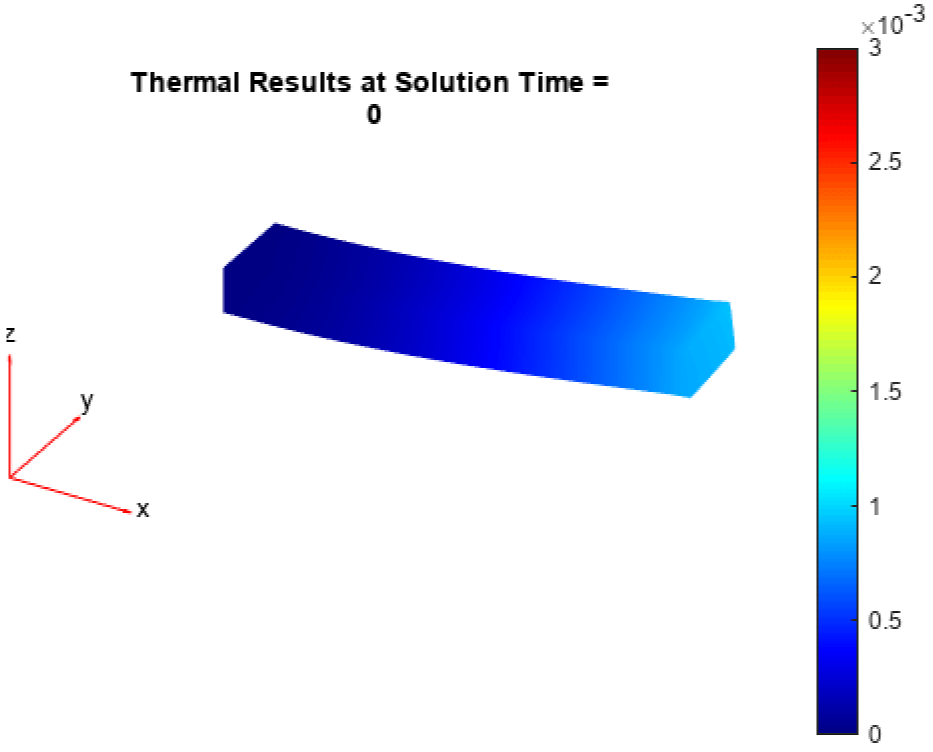
```
pdeplot3D(structuralmodel, ...
            "ColorMapData", ...
             thermalstressresults.Displacement.Magnitude, ...
            "Deformation", ...
             thermalstressresults.Displacement)
title(["Thermal Expansion at Solution Time = " ...
                         num2str(tlist(end))])
caxis([0 3e-3])
```

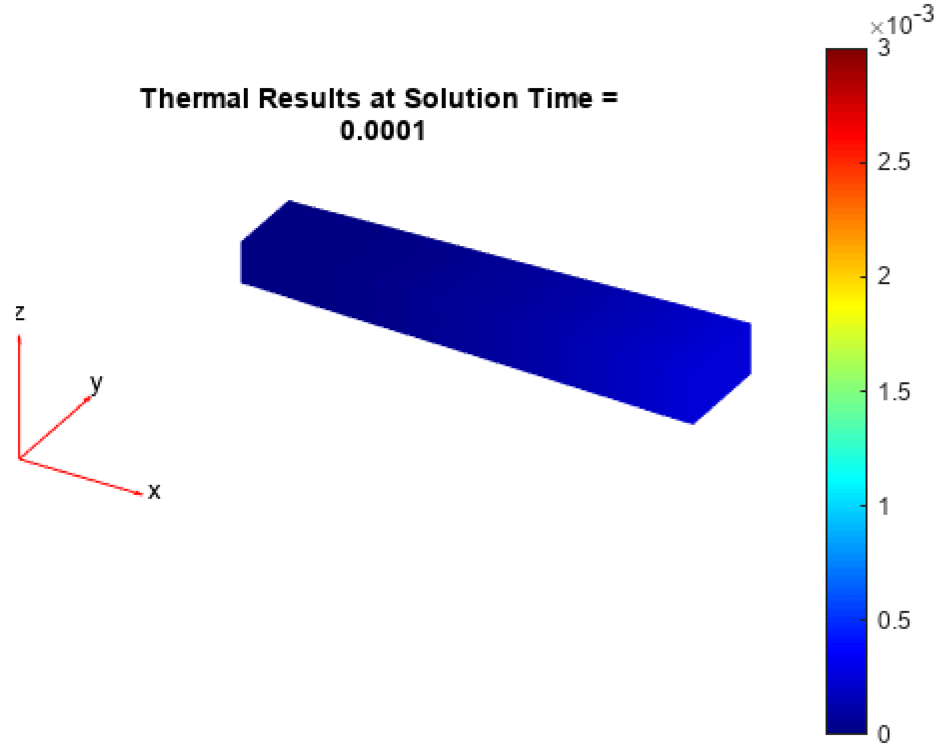


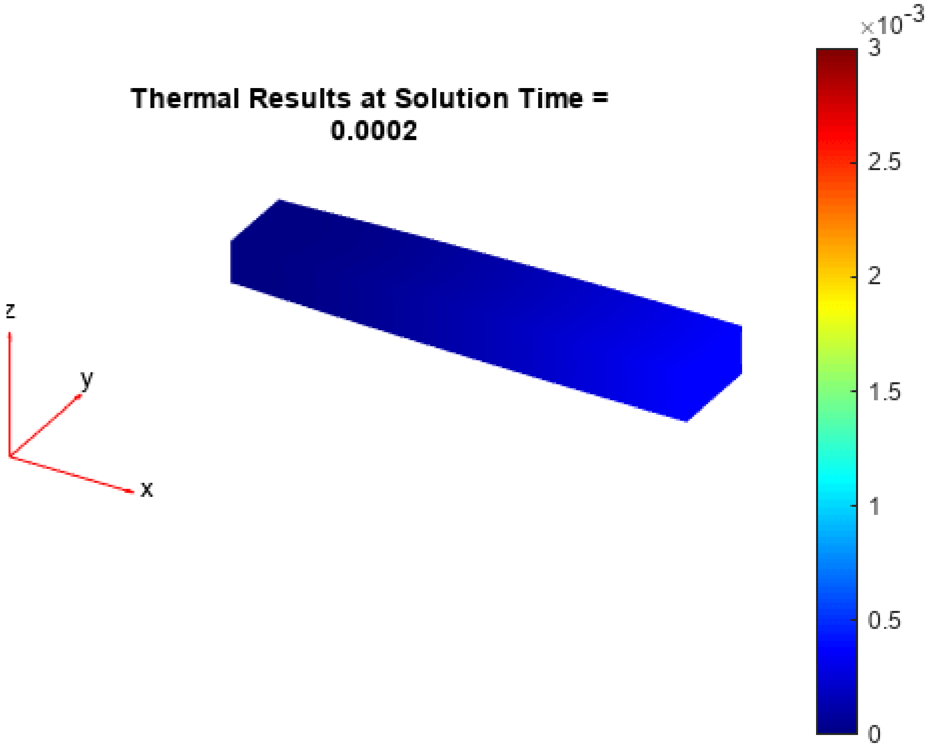
Now specify the body loads as the thermal model solutions for all time steps. For each body load, solve the structural model and plot the corresponding deformed shape of the beam.

```
for n = 1:numel(thermalresults.SolutionTimes)
    structuralBodyLoad(structuralmodel, ...
                         "Temperature", ...
                          thermalresults, ...
                         "TimeStep",n);
```

```
    thermalstressresults = solve(structuralmodel);
    figure
    pdeplot3D(structuralmodel, ...
                 "ColorMapData", ...
                  thermalstressresults.Displacement.Magnitude, ...
                 "Deformation", ...
                  thermalstressresults.Displacement)
    title(["Thermal Results at Solution Time = " ...
                                num2str(tlist(n))])
    caxis([0 3e-3])
end
```




Thermal Results at Solution Time =
0.0001
×10^-3
3
2.5
2
1.5
1
0.5
0
z
y
x



Solution to Steady-State Thermal Model

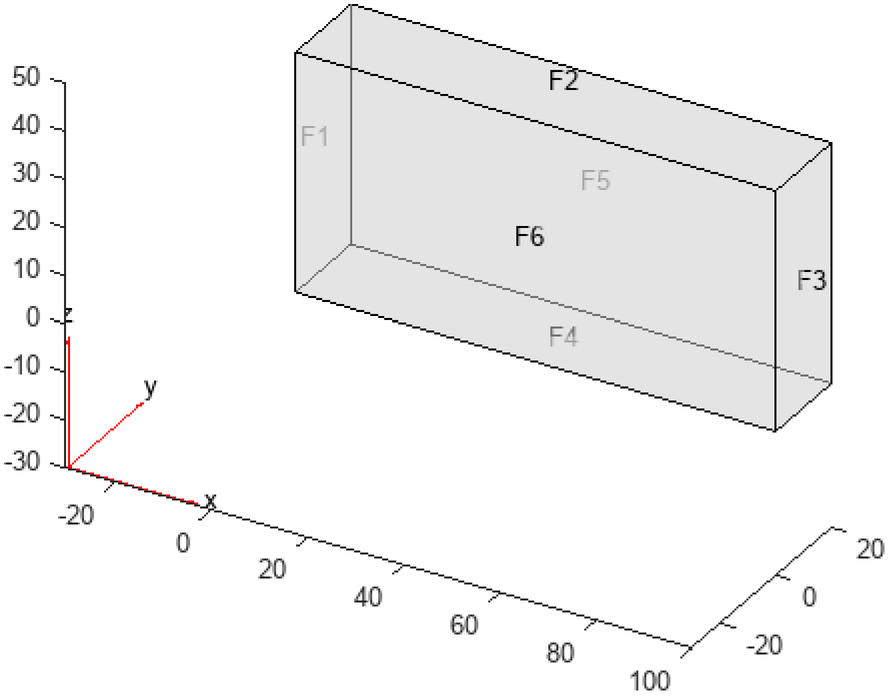
Solve a 3-D steady-state thermal problem.

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 °C to the left side of the block (face 1) and a constant temperature of 300 °C to 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);
```

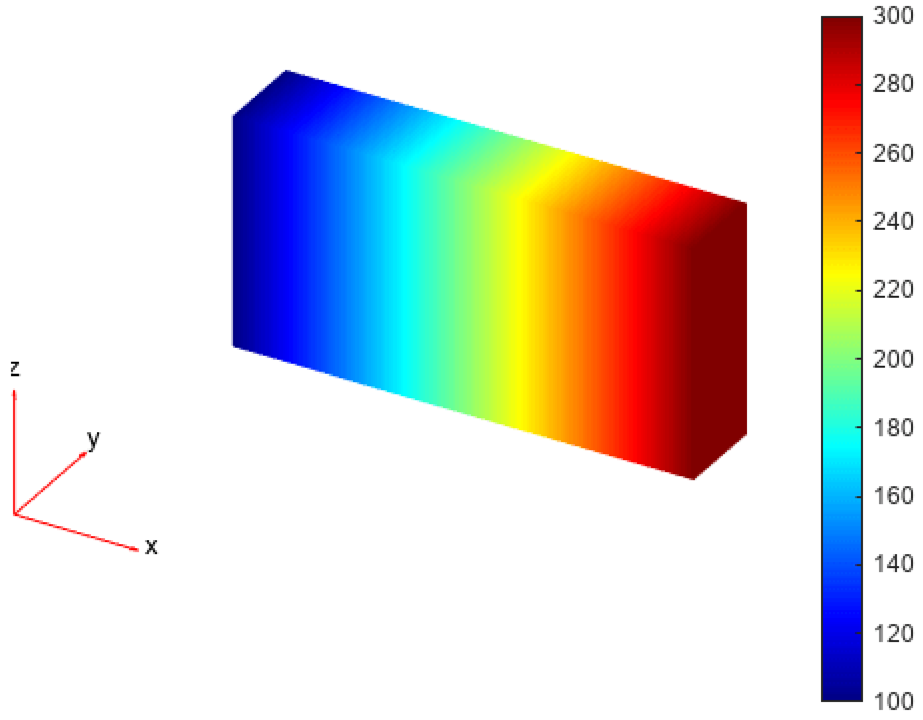
Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)
```

```
thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [12691x1 double]
     XGradients: [12691x1 double]
     YGradients: [12691x1 double]
     ZGradients: [12691x1 double]
           Mesh: [1x1 FEMesh]
```

The solver finds the 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 the nodal locations.

```
pdeplot3D(thermalmodel,"ColorMapData",thermalresults.Temperature)
```



Solution to Transient Thermal Model

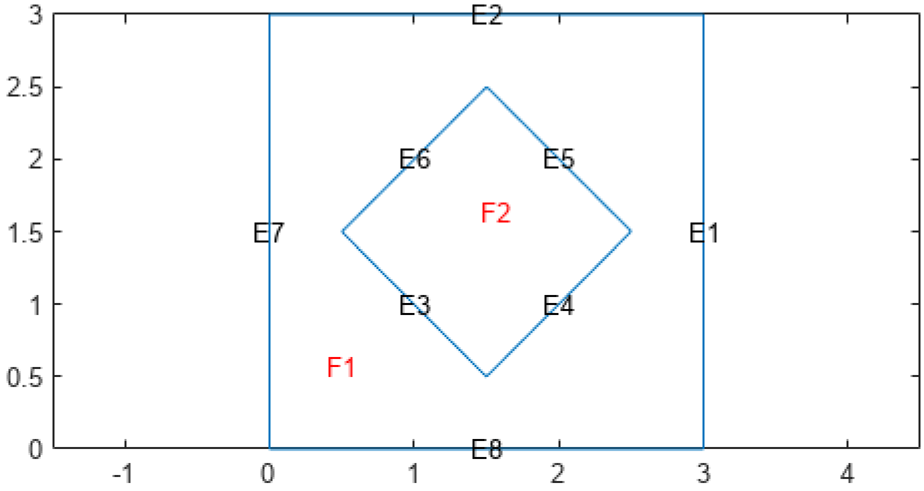
Solve a 2-D transient thermal problem.

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 these 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 these 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 a density of 4 W/m^2.

```
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);
```

Generate the mesh.

```
generateMesh(thermalmodel);
```

The dynamics for this problem are very fast. The temperature reaches a steady state in about 0.1 second. To capture the most active part of the dynamics, set the solution time to `logspace(-2,-1,10)`. This command returns 10 logarithmically spaced solution times between 0.01 and 0.1.

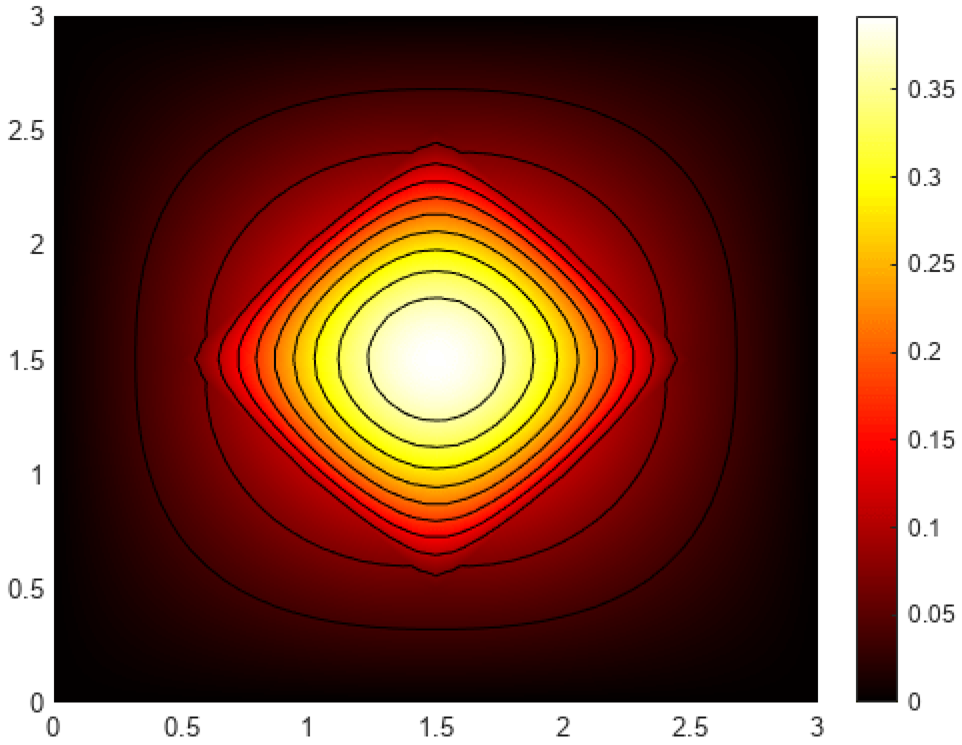
```
tlist = logspace(-2,-1,10);
```

Solve the equation.

```
thermalresults = solve(thermalmodel,tlist);
```

Plot the solution with isothermal lines by using a contour plot.

```
T = thermalresults.Temperature;
pdeplot(thermalmodel,"XYData",T(:,10),"Contour","on","ColorMap","hot")
```



Solution to Transient Thermal Model Using Modal Superposition Method

Solve a transient thermal problem by first obtaining mode shapes for a particular decay range and then using the modal superposition method.

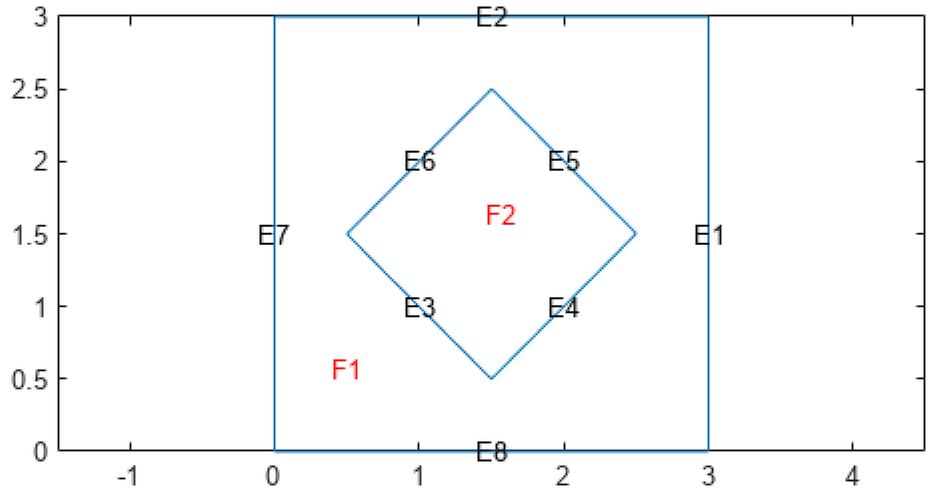
Modal Decomposition

First, create a modal thermal model.

```
thermalmodel = createpde("thermal","modal");
```

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 these 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 these 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 a density of 4 W/m^2.

```
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);
```

Generate the mesh.

```
generateMesh(thermalmodel);
```

Compute eigenmodes of the thermal model in the decay range [100,10000] s^{-1}.

```
RModal = solve(thermalmodel,"DecayRange",[100,10000])
```

```
RModal = 
  ModalThermalResults with properties:

    DecayRates: [164x1 double]
    ModeShapes: [1481x164 double]
      ModeType: "EigenModes"
          Mesh: [1x1 FEMesh]
```

Transient Analysis

Knowing the mode shapes, you can now use the modal superposition method to solve the transient thermal problem. First, switch the thermal model analysis type to transient.

```
thermalmodel.AnalysisType = "transient";
```

The dynamics for this problem are very fast. The temperature reaches a steady state in about 0.1 second. To capture the most active part of the dynamics, set the solution time to `logspace(-2,-1,100)`. This command returns 100 logarithmically spaced solution times between 0.01 and 0.1.

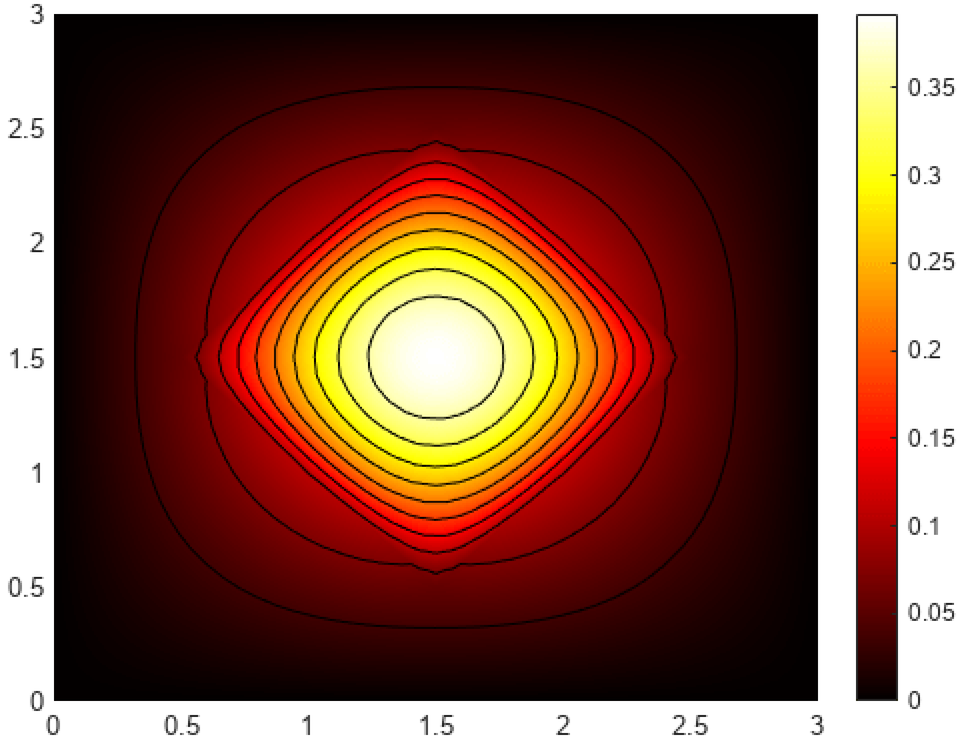
```
tlist = logspace(-2,-1,10);
```

Solve the equation.

```
Rtransient = solve(thermalmodel,tlist,"ModalResults",RModal);
```

Plot the solution with isothermal lines by using a contour plot.

```
T = Rtransient.Temperature;
pdeplot(thermalmodel,"XYData",T(:,end), ...
                     "Contour","on", ...
                     "ColorMap","hot")
```



Snapshots for Proper Orthogonal Decomposition

Obtain POD modes of a linear thermal model using several instances of the transient solution (snapshots).

Create a transient thermal model.

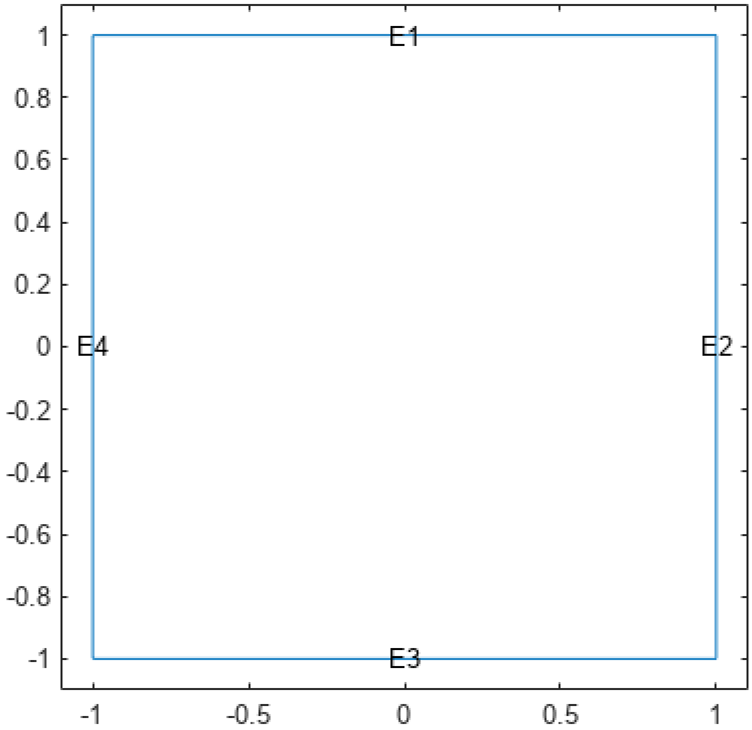
```
thermalmodel = createpde("thermal","transient");
```

Create a unit square geometry and include it in the model.

```
geometryFromEdges(thermalmodel,@squareg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",400, ...
                               "MassDensity",1300, ...
                               "SpecificHeat",600);
```

Set the temperature on the right edge to `100`.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Set an initial value of `0` for the temperature.

```
thermalIC(thermalmodel,0);
```

Generate a mesh.

```
generateMesh(thermalmodel);
```

Solve the model for three different values of heat source and collect snapshots.

```
tlist = 0:10:600;
snapShotIDs = [1:10 59 60 61];
Tmatrix = [];

heatVariation = [10000 15000 20000];
for q = heatVariation
    internalHeatSource(thermalmodel,q);
    results = solve(thermalmodel,tlist);
    Tmatrix = [Tmatrix,results.Temperature(:,snapShotIDs)];
end
```

Switch the thermal model analysis type to modal.

```
thermalmodel.AnalysisType = "modal";
```

Compute the POD modes.

```
RModal = solve(thermalmodel,"Snapshots",Tmatrix)
```

```
RModal = 
  ModalThermalResults with properties:

          DecayRates: [6x1 double]
          ModeShapes: [1541x6 double]
    SnapshotsAverage: [1541x1 double]
            ModeType: "PODModes"
                Mesh: [1x1 FEMesh]
```

Solution to 2-D Electrostatic Analysis Model

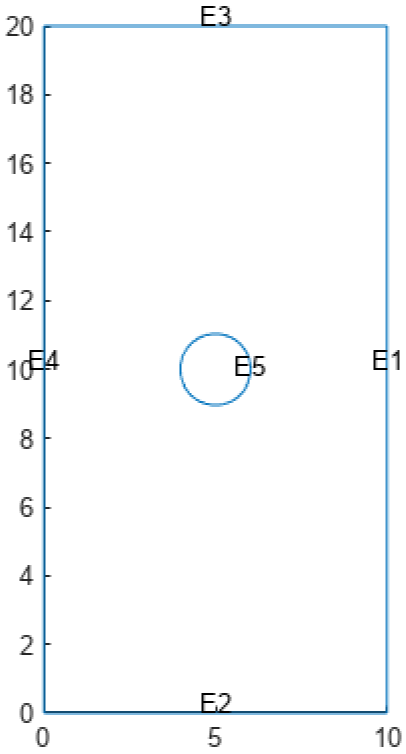
Solve an electromagnetic problem and find the electric potential and field distribution for a 2-D geometry representing a plate with a hole.

Create an electromagnetic model for electrostatic analysis.

```
emagmodel = createpde("electromagnetic","electrostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHolePlanar.stl");
pdegplot(emagmodel,"EdgeLabels","on")
```



Specify the vacuum permittivity value in the SI system of units.

```
emagmodel.VacuumPermittivity = 8.8541878128E-12;
```

Specify the relative permittivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1);
```

Apply the voltage boundary conditions on the edges framing the rectangle and the circle.

```
electromagneticBC(emagmodel,"Voltage",0,"Edge",1:4);
electromagneticBC(emagmodel,"Voltage",1000,"Edge",5);
```

Specify the charge density for the entire geometry.

```
electromagneticSource(emagmodel,"ChargeDensity",5E-9);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

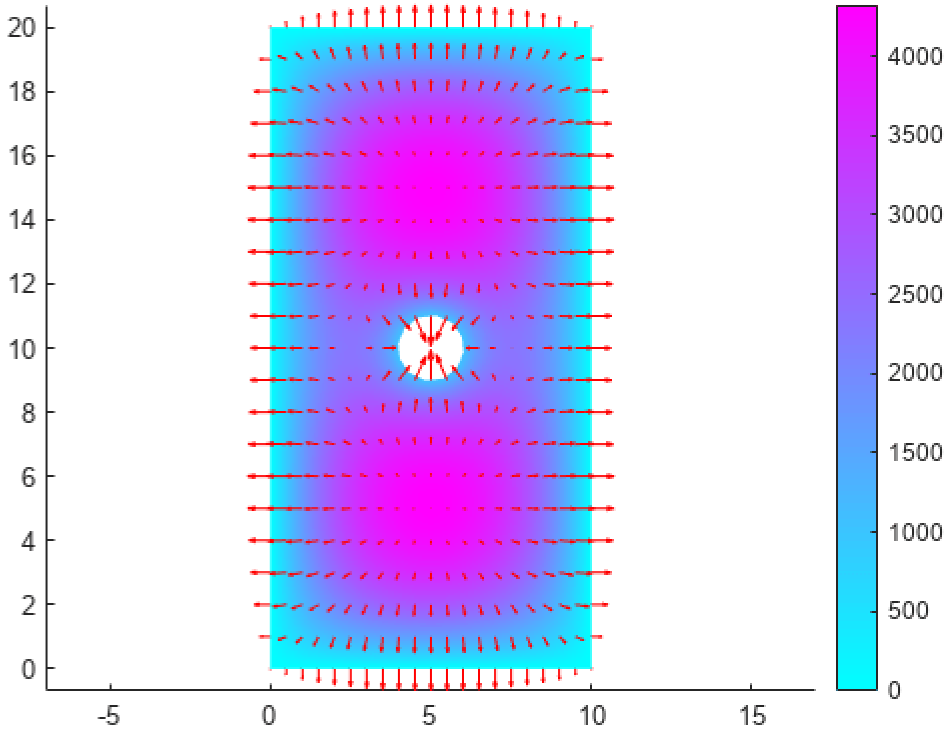
```
R = solve(emagmodel)
```

```
R = 
  ElectrostaticResults with properties:
```

```
      ElectricPotential: [1218x1 double]
          ElectricField: [1x1 FEStruct]
    ElectricFluxDensity: [1x1 FEStruct]
                   Mesh: [1x1 FEMesh]
```

Plot the electric potential and field.

```
pdeplot(emagmodel,"XYData",R.ElectricPotential, ...
                  "FlowData",[R.ElectricField.Ex ...
                              R.ElectricField.Ey])
axis equal
```



Solution to 3-D Magnetostatic Analysis Model

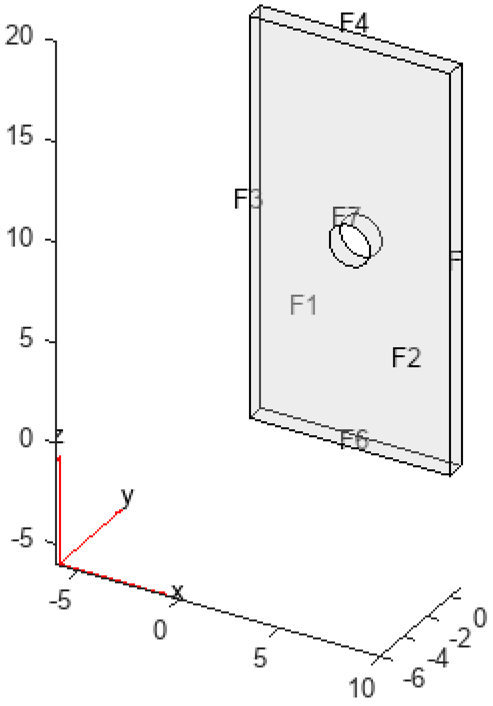
Solve a 3-D electromagnetic problem on a geometry representing a plate with a hole in its center. Plot the resulting magnetic potential and field distribution.

Create an electromagnetic model for magnetostatic analysis.

```
emagmodel = createpde("electromagnetic","magnetostatic");
```

Import and plot the geometry representing a plate with a hole.

```
importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(emagmodel,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Specify the current density for the entire geometry.

```
electromagneticSource(emagmodel,"CurrentDensity",[0;0;0.5]);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

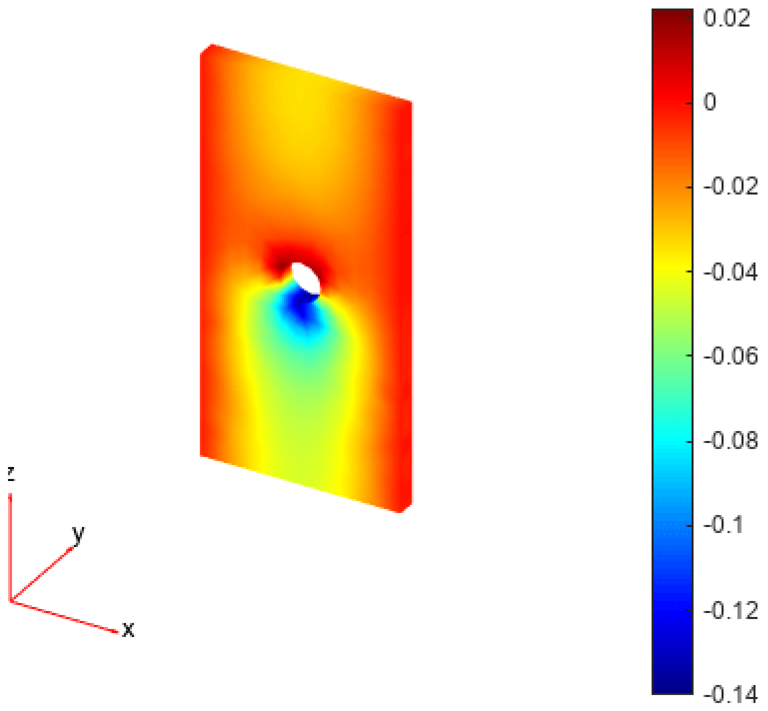
```
R = solve(emagmodel)
```

```
R = 
  MagnetostaticResults with properties:
```

```
       MagneticPotential: [1x1 FEStruct]
           MagneticField: [1x1 FEStruct]
     MagneticFluxDensity: [1x1 FEStruct]
                    Mesh: [1x1 FEMesh]
```

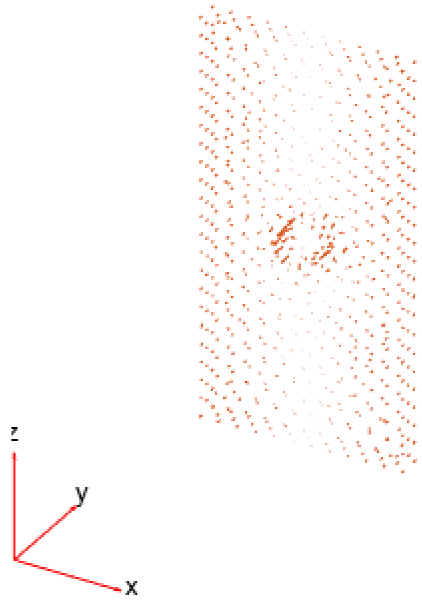
Plot the *z*-component of the magnetic potential.

```
pdeplot3D(emagmodel,"ColormapData",R.MagneticPotential.Az)
```



Plot the magnetic field.

```
pdeplot3D(emagmodel,"FlowData",[R.MagneticField.Hx ...
                                R.MagneticField.Hy ...
                                R.MagneticField.Hz])
```



Solution to 3-D DC Conduction Model

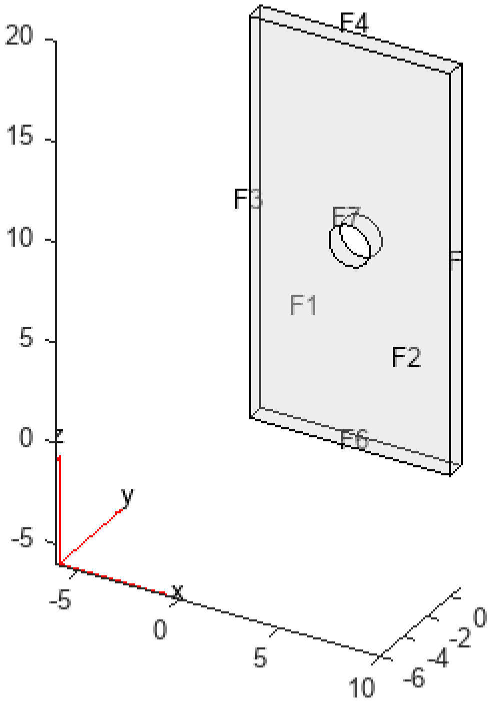
Solve a DC conduction problem on a geometry representing a 3-D plate with a hole in its center. Plot the electric potential and the components of the current density.

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the conductivity of the material.

```
electromagneticProperties(emagmodel,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the left, right, top, and bottom faces of the plate.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
```

Specify the surface current density on the face bordering the hole.

```
electromagneticBC(emagmodel,"SurfaceCurrentDensity",100,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

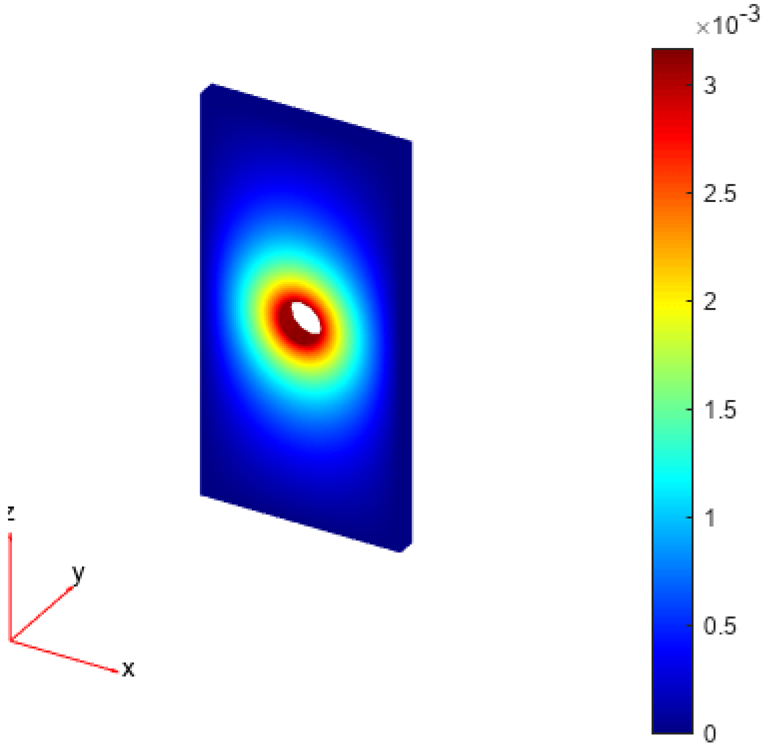
Solve the model.

```
R = solve(emagmodel)
```

```
R = 
  ConductionResults with properties:

    ElectricPotential: [4359x1 double]
        ElectricField: [1x1 FEStruct]
       CurrentDensity: [1x1 FEStruct]
                 Mesh: [1x1 FEMesh]
```

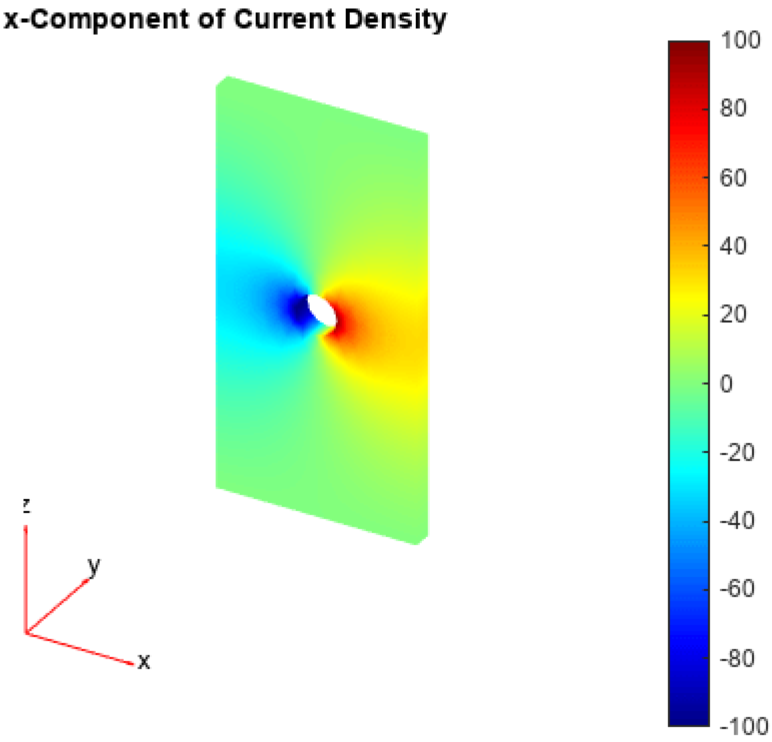
Plot the electric potential.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.ElectricPotential)
```



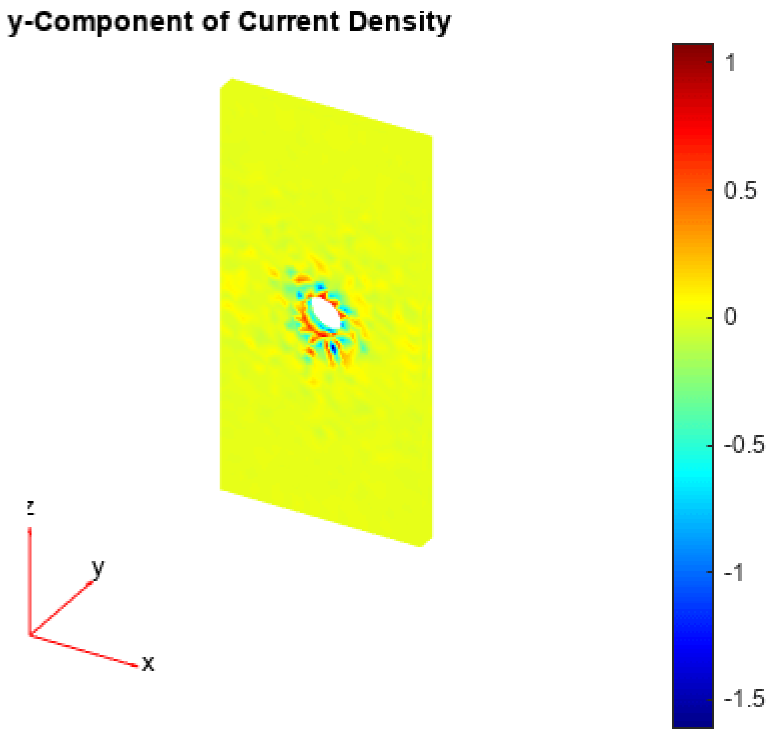
Plot the *x*-component of the current density.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.CurrentDensity.Jx)
title("x-Component of Current Density")
```



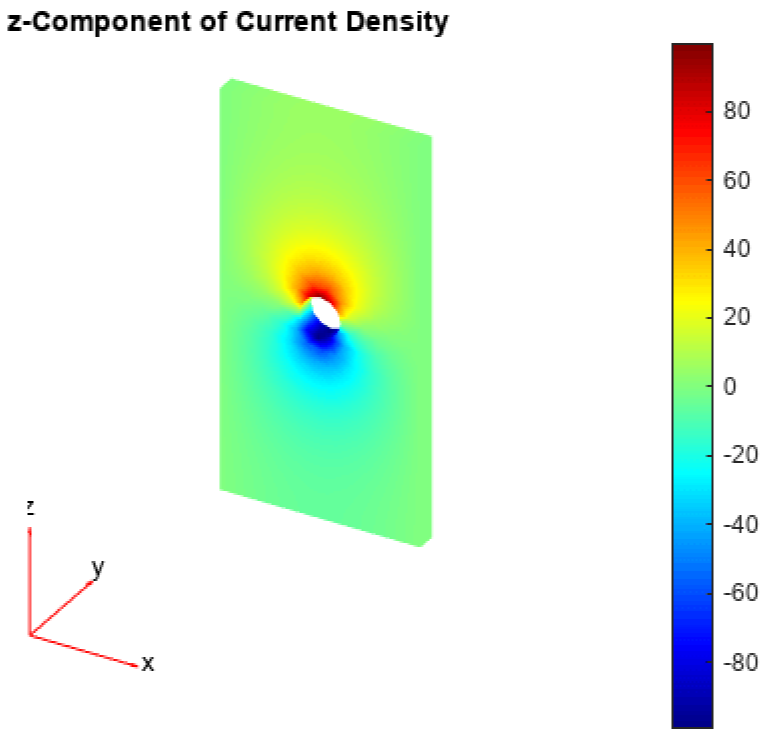
Plot the *y*-component of the current density.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.CurrentDensity.Jy)
title("y-Component of Current Density")
```



Plot the *z*-component of the current density.

```
figure
pdeplot3D(emagmodel,"ColorMapData",R.CurrentDensity.Jz)
title("z-Component of Current Density")
```



DC Conduction Solution as Current Density for Magnetostatic Model

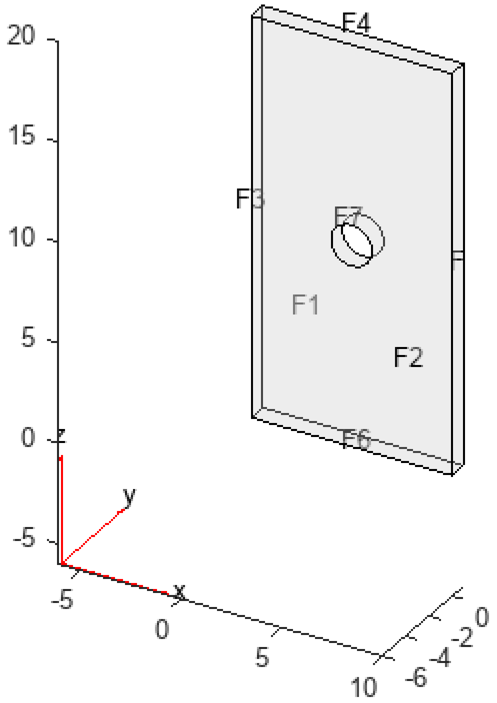
Use a solution obtained by performing a DC conduction analysis to specify current density for a magnetostatic model.

Create an electromagnetic model for DC conduction analysis.

```
emagmodel = createpde("electromagnetic","conduction");
```

Import and plot a geometry representing a plate with a hole.

```
gm = importGeometry(emagmodel,"PlateHoleSolid.stl");
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.3)
```



Specify the conductivity of the material.

```
electromagneticProperties(emagmodel,"Conductivity",6e4);
```

Apply the voltage boundary conditions on the left, right, top, and bottom faces of the plate.

```
electromagneticBC(emagmodel,"Voltage",0,"Face",3:6);
```

Specify the surface current density on the face bordering the hole.

```
electromagneticBC(emagmodel,"SurfaceCurrentDensity",100,"Face",7);
```

Generate the mesh.

```
generateMesh(emagmodel);
```

Solve the model.

```
R = solve(emagmodel);
```

Change the analysis type of the model to magnetostatic.

```
emagmodel.AnalysisType = "magnetostatic";
```

This model already has a quadratic mesh that you generated for the DC conduction analysis. For a 3-D magnetostatic model, the mesh must be linear. Generate a new linear mesh. The `generateMesh` function creates a linear mesh by default if the model is 3-D and magnetostatic.

```
generateMesh(emagmodel);
```

Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the material.

```
electromagneticProperties(emagmodel,"RelativePermeability",5000);
```

Apply the magnetic potential boundary conditions on the side faces and the face bordering the hole.

```
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0],"Face",3:6);
electromagneticBC(emagmodel,"MagneticPotential",[0;0;0.01],"Face",7);
```

Specify the current density for the entire geometry using the DC conduction solution.

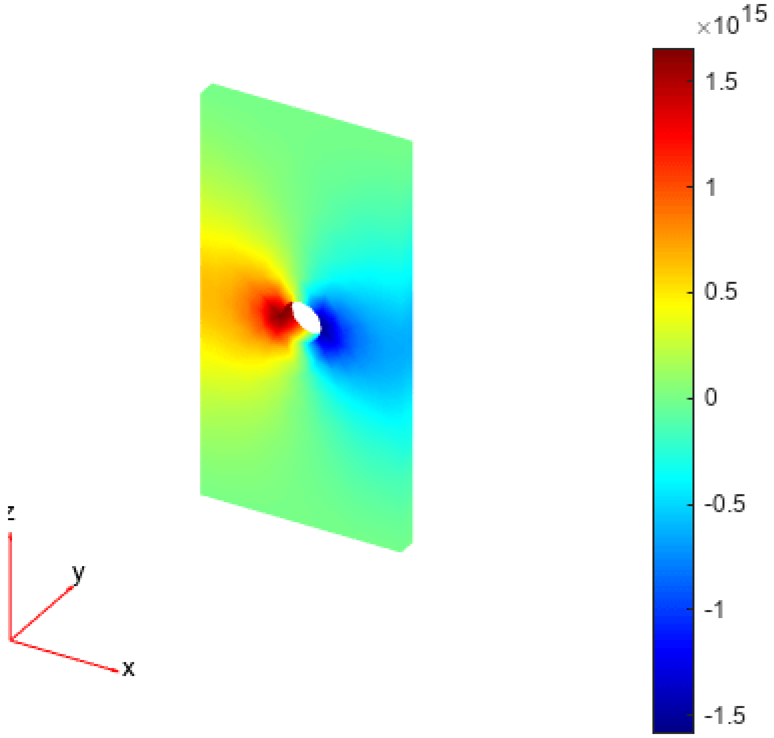
```
electromagneticSource(emagmodel,"CurrentDensity",R);
```

Solve the model.

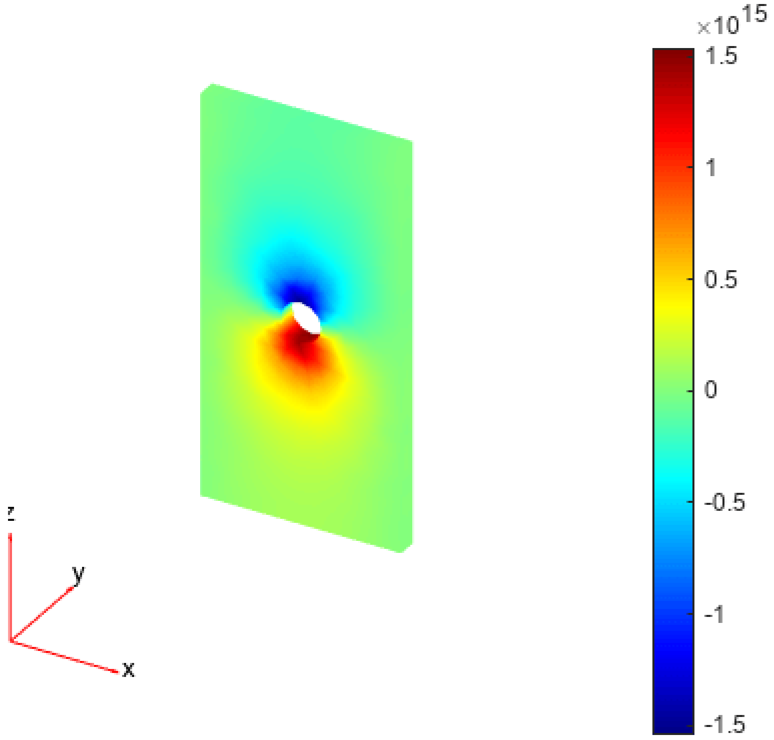
```
Rmagnetostatic = solve(emagmodel);
```

Plot the *x*- and *z*-components of the magnetic potential.

```
pdeplot3D(emagmodel,"ColormapData",Rmagnetostatic.MagneticPotential.Ax)
```



```
pdeplot3D(emagmodel,"ColormapData",Rmagnetostatic.MagneticPotential.Az)
```



Solution to 2-D Magnetostatic Model with Permanent Magnet

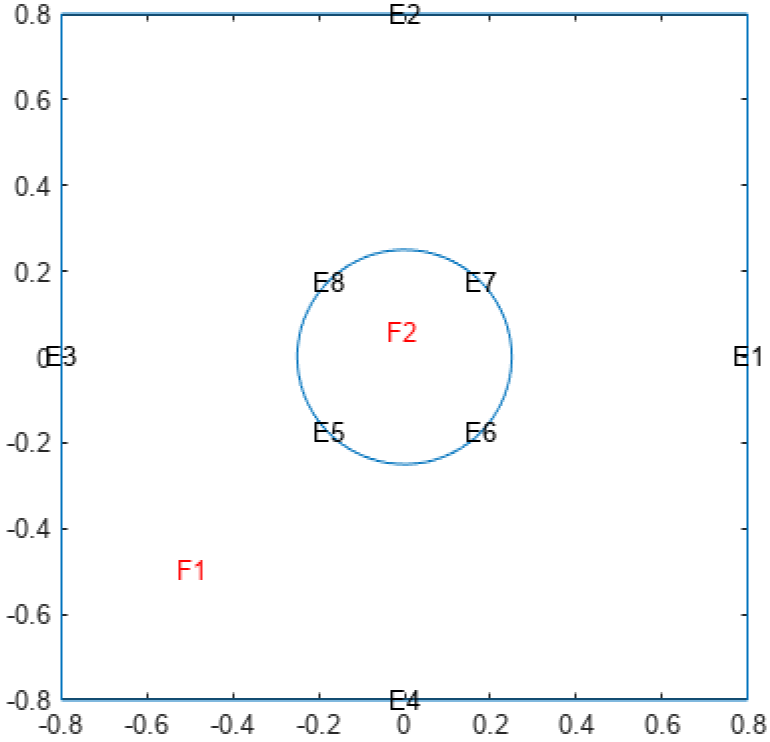
Solve a magnetostatic model of a copper square with a permanent neodymium magnet in its center.

Create the unit square geometry with a circle in its center.

```
L = 0.8;
r = 0.25;
sq = [3 4 -L L L -L -L -L L L]';
circ = [1 0 0 r 0 0 0 0 0 0]';
gd = [sq,circ];
sf = "sq + circ";
ns = char('sq','circ');
ns = ns';
g = decsg(gd,sf,ns);
```

Plot the geometry with the face and edge labels.

```
pdegplot(g,"FaceLabels","on","EdgeLabels","on")
```



Create a magnetostatic model and include the geometry in the model.

```
emagmodel = createpde("electromagnetic","magnetostatic");
geometryFromEdges(emagmodel,g);
```

Specify the vacuum permeability value in the SI system of units.

```
emagmodel.VacuumPermeability = 1.2566370614e-6;
```

Specify the relative permeability of the copper for the square.

```
electromagneticProperties(emagmodel,"Face",1, ...
                                    "RelativePermeability",1);
```

Specify the relative permeability of the neodymium for the circle.

```
electromagneticProperties(emagmodel,"Face",2, ...
                                    "RelativePermeability",1.05);
```

Specify the magnetization magnitude for the neodymium magnet.

```
M = 1e6;
```

Specify magnetization on the circular face in the positive *x*-direction. Magnetization for a 2-D model is a column vector of two elements.

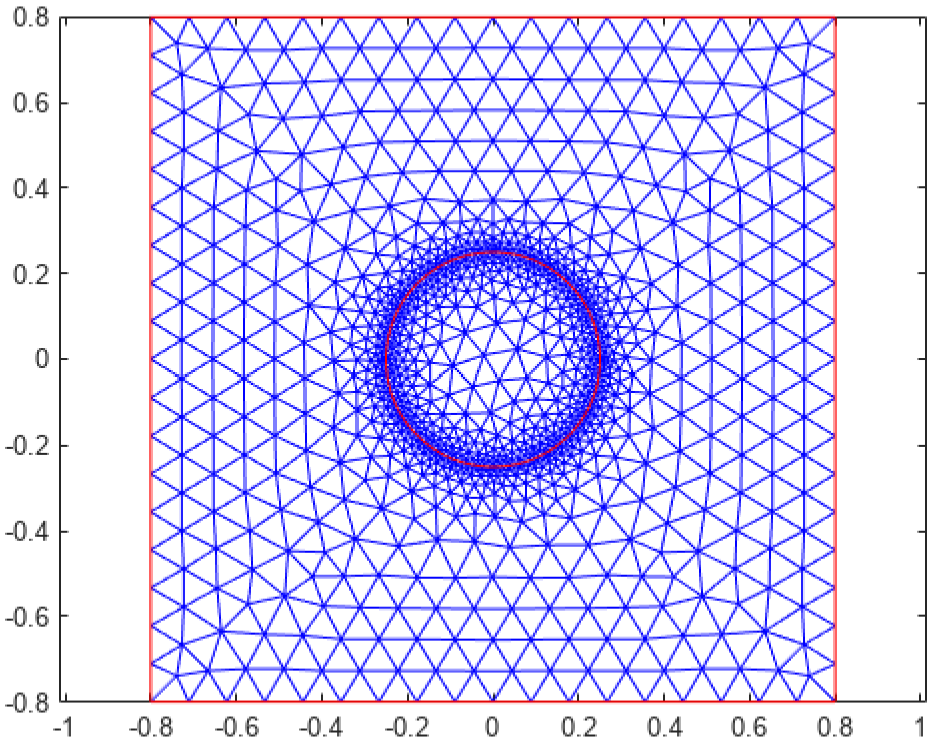
```
dir = [1;0];
electromagneticSource(emagmodel,"Face",2,"Magnetization",M*dir);
```

Apply the magnetic potential boundary conditions on the edges framing the square.

```
electromagneticBC(emagmodel,"Edge",1:4,"MagneticPotential",0);
```

Generate the mesh with finer meshing near the edges of the circle.

```
generateMesh(emagmodel,"Hedge",{5:8,0.007});
figure
pdemesh(emagmodel)
```

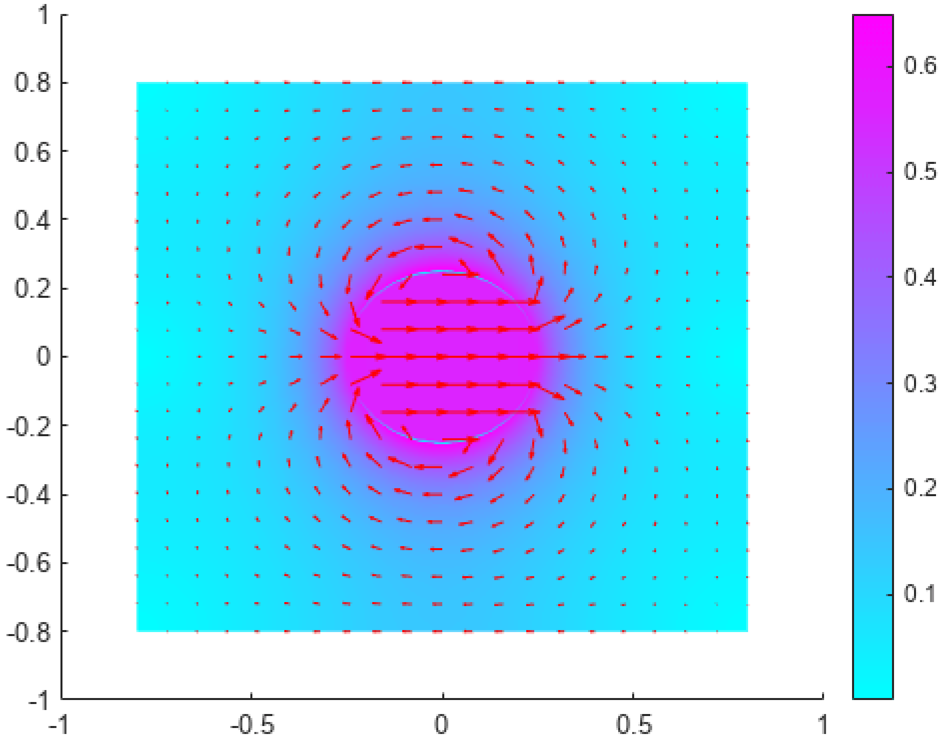


Solve the model, and find the resulting magnetic fields B and H. Here, $B = \mu H + \mu_0 M$, where μ is the absolute magnetic permeability of the material, μ_0 is the vacuum permeability, and M is the magnetization.

```
R = solve(emagmodel);
Bmag = sqrt(R.MagneticFluxDensity.Bx.^2 + R.MagneticFluxDensity.By.^2);
Hmag = sqrt(R.MagneticField.Hx.^2 + R.MagneticField.Hy.^2);
```

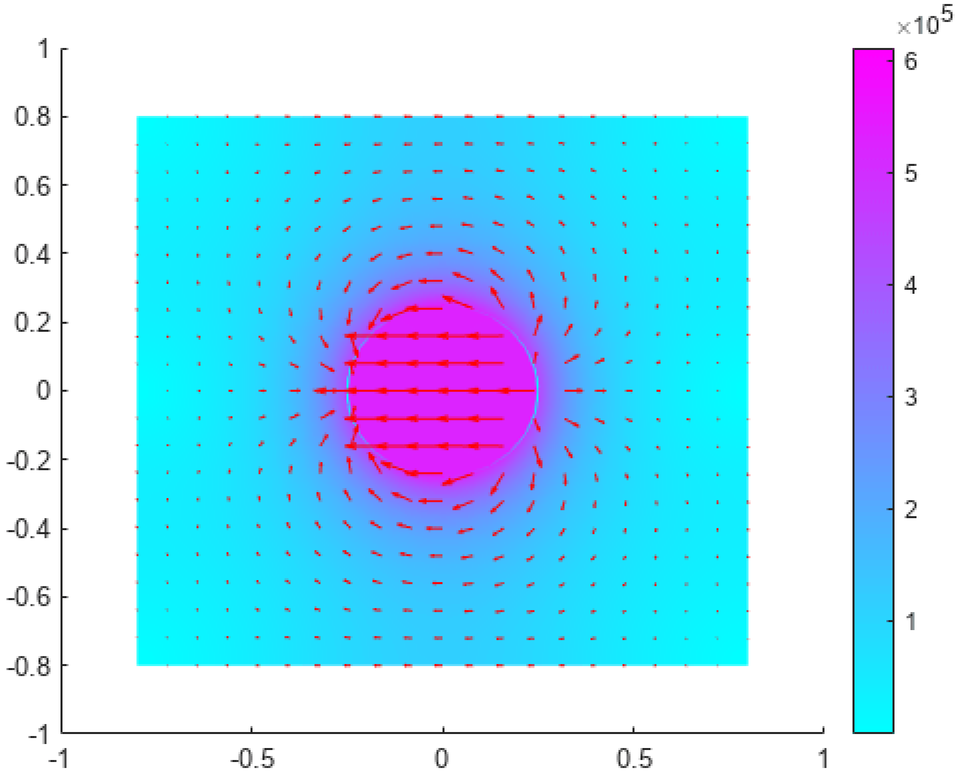
Plot the magnetic field B.

```
figure
pdeplot(emagmodel,"XYData",Bmag, ...
                  "FlowData",[R.MagneticFluxDensity.Bx ...
                              R.MagneticFluxDensity.By])
```



Plot the magnetic field *H*.

```
figure
pdeplot(emagmodel,"XYData",Hmag, ...
                  "FlowData",[R.MagneticField.Hx R.MagneticField.Hy])
```



Solution to 2-D Harmonic Electromagnetic Model

For an electromagnetic harmonic analysis problem, find the *x*- and *y*-components of the electric field. Solve the problem on a domain consisting of a square with a circular hole.

Create an electromagnetic model for harmonic analysis.

```
emagmodel = createpde("electromagnetic","harmonic");
```

Define a circle in a square, place them in one matrix, and create a set formula that subtracts the circle from the square.

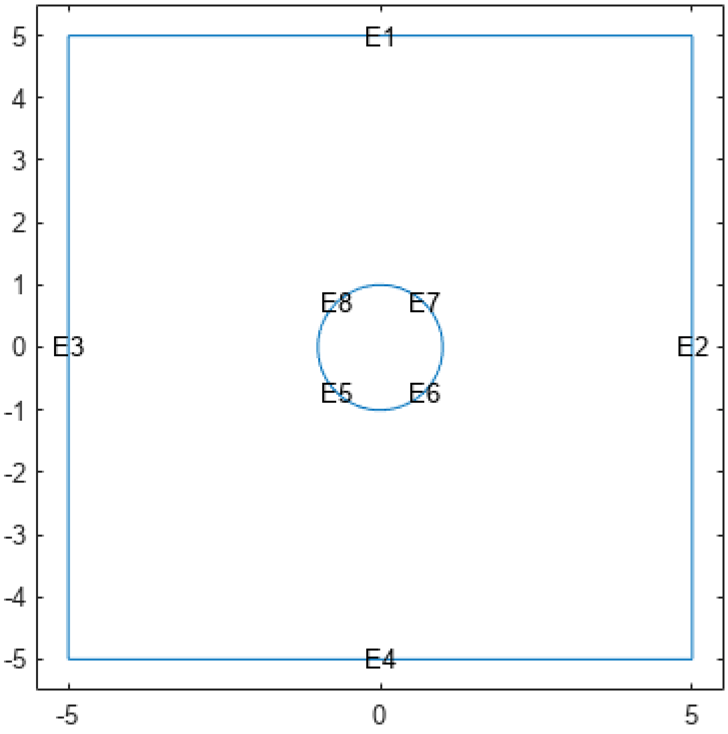
```
SQ = [3,4,-5,-5,5,5,-5,5,5,-5]';
C = [1,0,0,1]';
C = [C;zeros(length(SQ) - length(C),1)];
gm = [SQ,C];
sf = 'SQ-C';
```

Create the geometry.

```
ns = char('SQ','C');
ns = ns';
g = decsg(gm,sf,ns);
```

Include the geometry in the model and plot the geometry with the edge labels.

```
geometryFromEdges(emagmodel,g);
pdegplot(emagmodel,"EdgeLabels","on")
xlim([-5.5 5.5])
ylim([-5.5 5.5])
```



Specify the vacuum permittivity and permeability values as 1.

```
emagmodel.VacuumPermittivity = 1;
emagmodel.VacuumPermeability = 1;
```

Specify the relative permittivity, relative permeability, and conductivity of the material.

```
electromagneticProperties(emagmodel,"RelativePermittivity",1, ...
                                    "RelativePermeability",1, ...
                                    "Conductivity",0);
```

Apply the absorbing boundary condition with a thickness of 2 on the edges of the square. Use the default attenuation rate for the absorbing region.

```
electromagneticBC(emagmodel,"Edge",1:4, ...
                            "FarField","absorbing", ...
                            "Thickness",2);
```

Specify an electric field on the edges of the hole.

```
E = @(location,state) [1;0]*exp(-1i*2*pi*location.y);
electromagneticBC(emagmodel,"Edge",5:8,"ElectricField",E);
```

Generate a mesh.

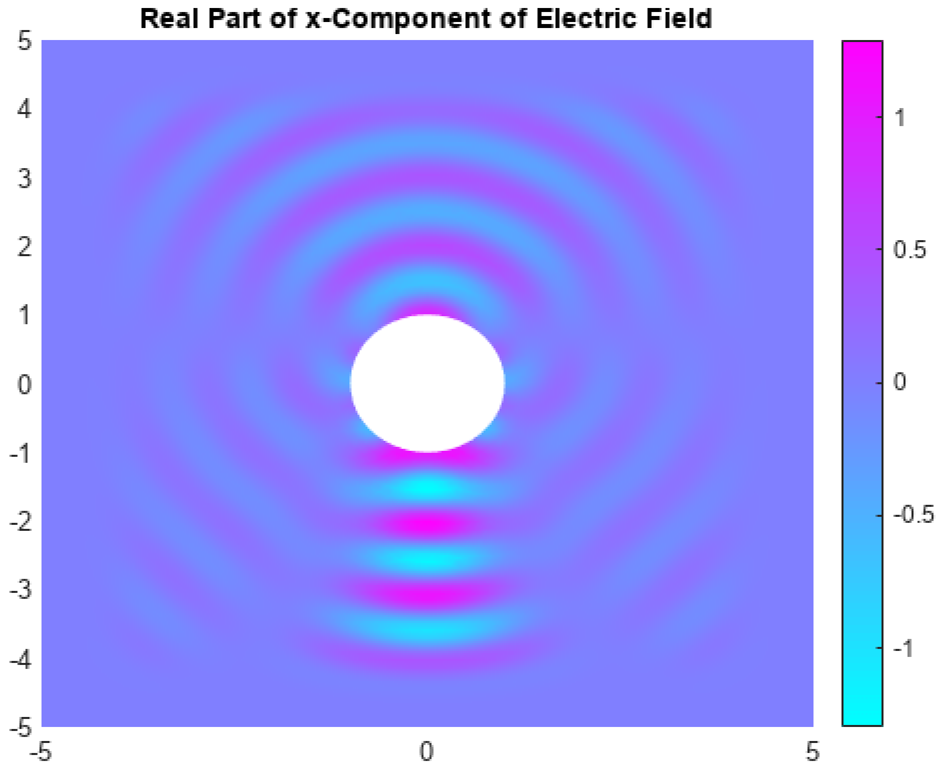
```
generateMesh(emagmodel,"Hmax",1/2^3);
```

Solve the model for a frequency of 2π.

```
result = solve(emagmodel,"Frequency",2*pi);
```

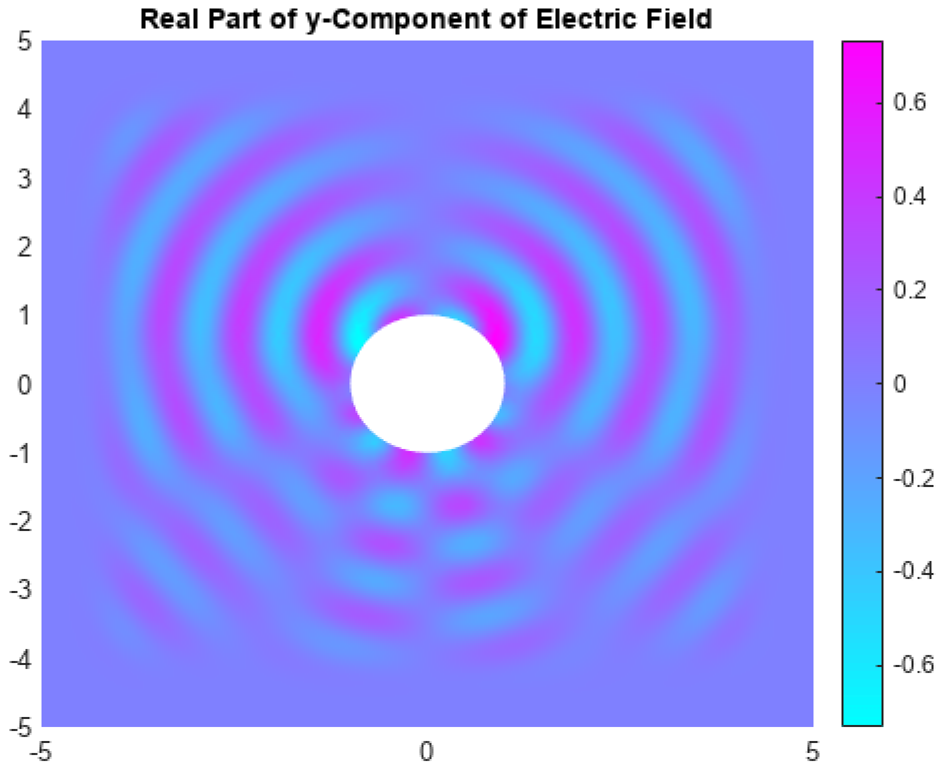
Plot the real part of the *x*-component of the resulting electric field.

```
figure
pdeplot(emagmodel,"XYData",real(result.ElectricField.Ex));
title("Real Part of x-Component of Electric Field")
```



Plot the real part of the *y*-component of the resulting electric field.

```
figure
pdeplot(emagmodel,"XYData",real(result.ElectricField.Ey));
title("Real Part of y-Component of Electric Field")
```



Input Arguments

`structuralStatic` — Static structural analysis model
`StructuralModel` object

Static structural analysis model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","static-solid")`

`structuralTransient` — Transient structural analysis model
`StructuralModel` object

Transient structural analysis model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`structuralFrequencyResponse` — Frequency response analysis structural model
`StructuralModel` object

Frequency response analysis structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","frequency-solid")`

`structuralModal` — Modal analysis structural model
`StructuralModel` object

Modal analysis structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","modal-solid")`

`tlist` — Solution times for structural or thermal transient analysis
real vector

Solution times for structural or thermal transient analysis, specified as a real vector of monotonically increasing or decreasing values.

Example: `0:20`

Data Types: `double`

`flist` — Solution frequencies for frequency response structural analysis
real vector

Solution frequencies for a frequency response structural analysis, specified as a real vector of monotonically increasing or decreasing values.

Example: `linspace(0,4000,150)`

Data Types: `double`

`[omega1,omega2]` — Frequency range for structural modal analysis
vector of two elements

Frequency range for a structural modal analysis, specified as a vector of two elements. Define `omega1` as slightly lower than the lowest expected frequency and `omega2` as slightly higher than the highest expected frequency. For example, if the lowest expected frequency is zero, then use a small negative value for `omega1`.

Example: `[-0.1,1000]`

Data Types: `double`

`structuralModalR` — Modal analysis results for structural model
`ModalStructuralResults` object

Modal analysis results for a structural model, specified as a `ModalStructuralResults` object.

Example: `structuralModalR = solve(structuralmodel,"FrequencyRange",[0,1e6])`

`thermalSteadyState` — Steady-state thermal analysis model
`ThermalModel` object

Steady-state thermal analysis model, specified as a `ThermalModel` object. `ThermalModel` contains the geometry, mesh, thermal properties of the material, internal heat source, Stefan-Boltzmann constant, boundary conditions, and initial conditions.

Example: `thermalmodel = createpde("thermal","steadystate")`

`thermalTransient` — Transient thermal analysis model
`ThermalModel` object

Transient thermal analysis model, specified as a `ThermalModel` object. `ThermalModel` contains the geometry, mesh, thermal properties of the material, internal heat source, Stefan-Boltzmann constant, boundary conditions, and initial conditions.

`thermalModal` — Modal thermal analysis model
`ThermalModel` object

Modal thermal analysis model, specified as a `ThermalModel` object. `ThermalModel` contains the geometry, mesh, thermal properties of the material, internal heat source, Stefan-Boltzmann constant, boundary conditions, and initial conditions.

`[lambda1,lambda2]` — Decay range for modal thermal analysis
vector of two elements

Decay range for modal thermal analysis, specified as a vector of two elements. The `solve` function solves a modal thermal analysis model for all modes in the decay range.

Data Types: `double`

`Tmatrix` — Thermal model solution snapshots
matrix

Thermal model solution snapshots, specified as a matrix.

Data Types: `double`

`thermalModalR` — Modal analysis results for thermal model
`ModalThermalResults` object

Modal analysis results for a thermal model, specified as a `ModalThermalResults` object.

Example: `thermalModalR = solve(thermalmodel,"DecayRange",[0,1000])`

emagmodel — Electromagnetic model for electrostatic, magnetostatic, or DC conduction analysis
`ElectromagneticModel` object

Electromagnetic model for electrostatic, magnetostatic, or DC conduction analysis, specified as an `ElectromagneticModel` object. The model contains the geometry, mesh, material properties, electromagnetic sources, and boundary conditions.

Example: `emagmodel = createpde("electromagnetic","magnetostatic")`

omega — Solution frequencies for harmonic electromagnetic analysis
nonnegative number | vector of nonnegative numbers

Solution frequencies for a harmonic electromagnetic analysis, specified as a nonnegative number or a vector of nonnegative numbers.

Data Types: `double`

Output Arguments

`structuralStaticResults` — Static structural analysis results
`StaticStructuralResults` object

Static structural analysis results, returned as a `StaticStructuralResults` object.

`structuralTransientResults` — Transient structural analysis results
`TransientStructuralResults` object

Transient structural analysis results, returned as a `TransientStructuralResults` object.

`structuralFrequencyResponseResults` — Frequency response structural analysis results
`FrequencyStructuralResults` object

Frequency response structural analysis results, returned as a `FrequencyStructuralResults` object.

`structuralModalResults` — Modal structural analysis results
`ModalStructuralResults` object

Modal structural analysis results, returned as a `ModalStructuralResults` object.

`thermalSteadyStateResults` — Steady-state thermal analysis results
`SteadyStateThermalResults` object

Steady-state thermal analysis results, returned as a `SteadyStateThermalResults` object.

`thermalTransientResults` — Transient thermal analysis results
`TransientThermalResults` object

Transient thermal analysis results, returned as a `TransientThermalResults` object.

`thermalModalResults` — Modal thermal analysis results
`ModalThermalResults` object

Modal thermal analysis results, returned as a `ModalThermalResults` object.

`emagStaticResults` — Electrostatic, magnetostatic or DC conduction analysis results
`ElectrostaticResults` object | `MagnetostaticResults` object | `ConductionResults` object

Electrostatic, magnetostatic, or DC conduction analysis results, returned as an `ElectrostaticResults`, `MagnetostaticResults`, or `ConductionResults` object.

`emagHarmonicResults` — Harmonic electromagnetic analysis results
`HarmonicResults` object

Harmonic electromagnetic analysis results, returned as a `HarmonicResults` object.

Tips

- When you use modal analysis results to solve a transient structural dynamics model, the `modalresults` argument must be created in Partial Differential Equation Toolbox from R2019a or newer.

- For a frequency response model with damping, the results are complex. Use functions such as `abs` and `angle` to obtain real-valued results, such as the magnitude and phase.

Version History

Introduced in R2017a

DC conduction and permanent magnets

You can now solve stationary current distribution in conductors due to applied voltage. You also can solve electromagnetic problems accounting for magnetization of materials.

Harmonic electromagnetic analysis

You can now solve 2-D and 3-D time-harmonic Maxwell's equations (the Helmholtz equation).

Reduced-order modeling for thermal analysis

You can now compute modes of a thermal model using eigenvalue or proper orthogonal decomposition. You also can speed up computations for a transient thermal model by using the computed modes.

3-D electrostatic and magnetostatic problems

You can now solve 3-D electrostatic and magnetostatic problems.

2-D electrostatic and magnetostatic problems

You can now solve 2-D electrostatic and magnetostatic problems.

Axisymmetric analysis

You can now solve axisymmetric structural and thermal problems. Axisymmetric analysis simplifies 3-D thermal problems to 2-D using their symmetry around the axis of rotation.

Frequency response structural analysis

You can now solve frequency response structural problems and find displacement, velocity, acceleration, and solution frequencies at nodal locations of the mesh. To speed up computations, you can use modal analysis results for frequency response analysis. The `ModalResults` argument triggers the `solve` function to use the modal superposition method.

Lanczos algorithm for structural modal analysis problems

You can now specify the maximum number of Lanczos shifts and the block size for block Lanczos recurrence by using the `SolverOptions` property of `StructuralModel`. For details, see PDESolverOptions.

Modal superposition method for transient structural analysis

The new `ModalResults` argument triggers the `solve` function to switch to the modal transient solver instead of using the direct integration approach.

Thermal stress

The solver now solves accounts for mechanical and thermal effects when solving a static structural analysis model. The function returns a displacement, stress, strain, and von Mises stress induced by both mechanical and thermal loads.

Transient and modal structural analyses

You can now solve dynamic linear elasticity problems and find displacement, velocity, and acceleration at nodal locations of the mesh.

You also can solve modal analysis problems and find natural frequencies and mode shapes of a structure. When solving a modal analysis model, the solver requires a frequency range parameter and returns the modal solution in that frequency range.

Static structural analysis

You can now solve static linear elasticity problems and find displacement, stress, strain, and von Mises stress at nodal locations of the mesh.

See Also

`PDEModel` | `ThermalModel` | `StructuralModel` | `ElectromagneticModel` | `geometryFromEdges` | `geometryFromMesh` | `importGeometry` | `reduce`

solvepde

Package: `pde`

Solve PDE specified in a PDEModel

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

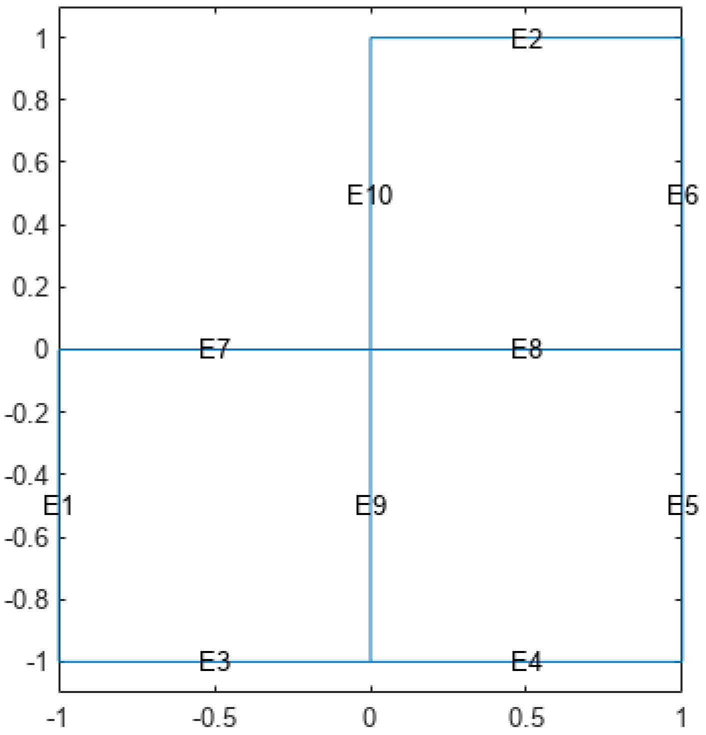
Solve a Stationary Problem: Poisson's Equation for the L-shaped Membrane

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

$$-\nabla \cdot \nabla u = 1 .$$

Toolbox solvers address equations of the form

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

Include the coefficients for Poisson's equation in the model.

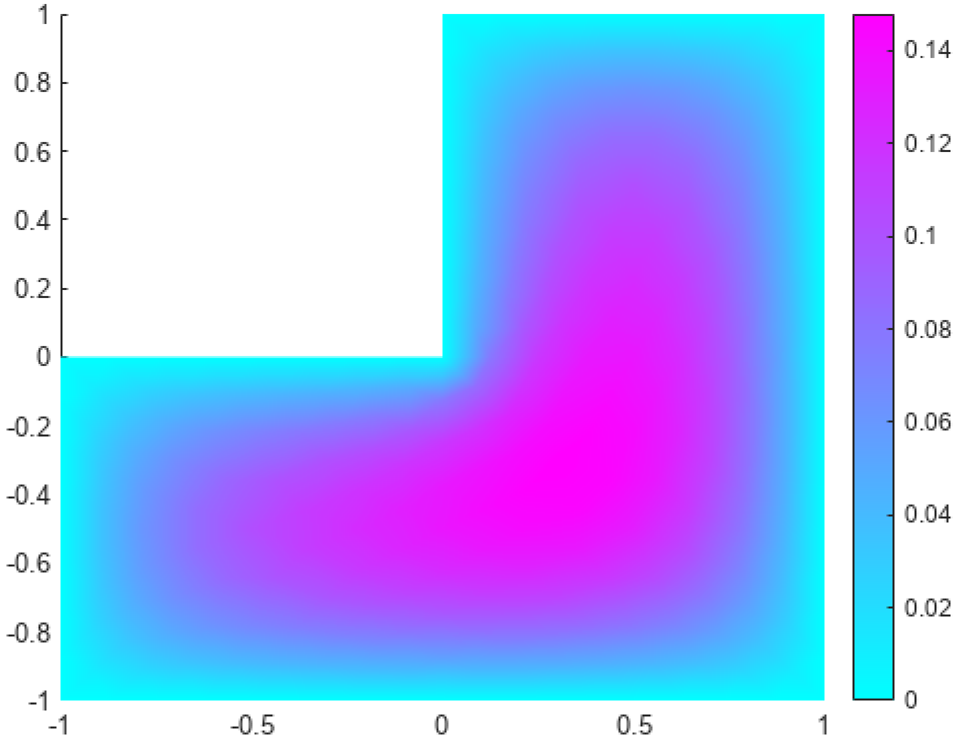
```
specifyCoefficients(model,"m",0,...
                          "d",0,...
                          "c",1,...
                          "a",0,...
                          "f",1);
```

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.

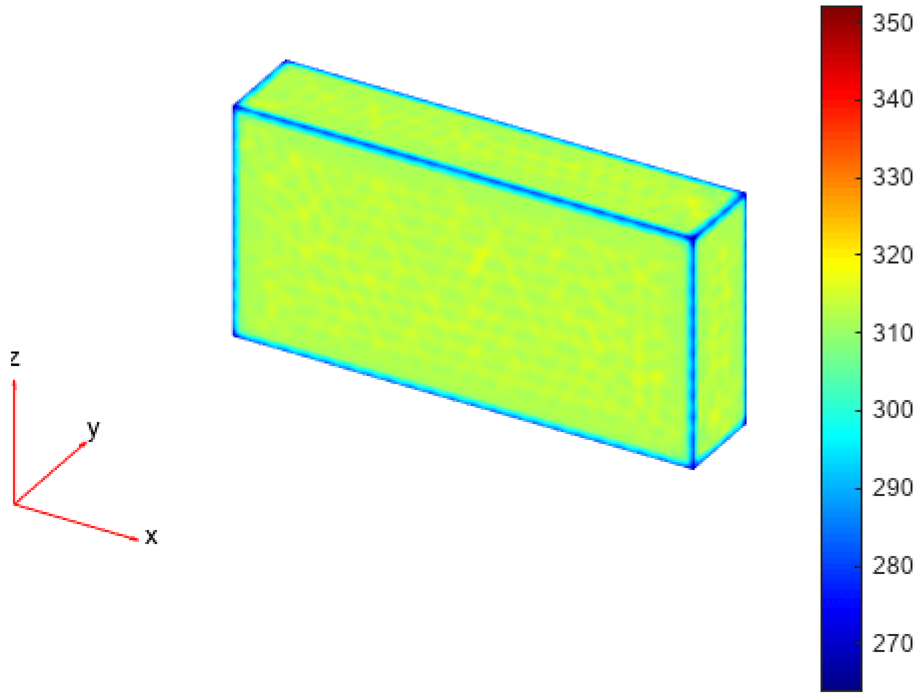
```
setInitialConditions(model,350);
```

Mesh the geometry and solve the model for times 0 through 20.

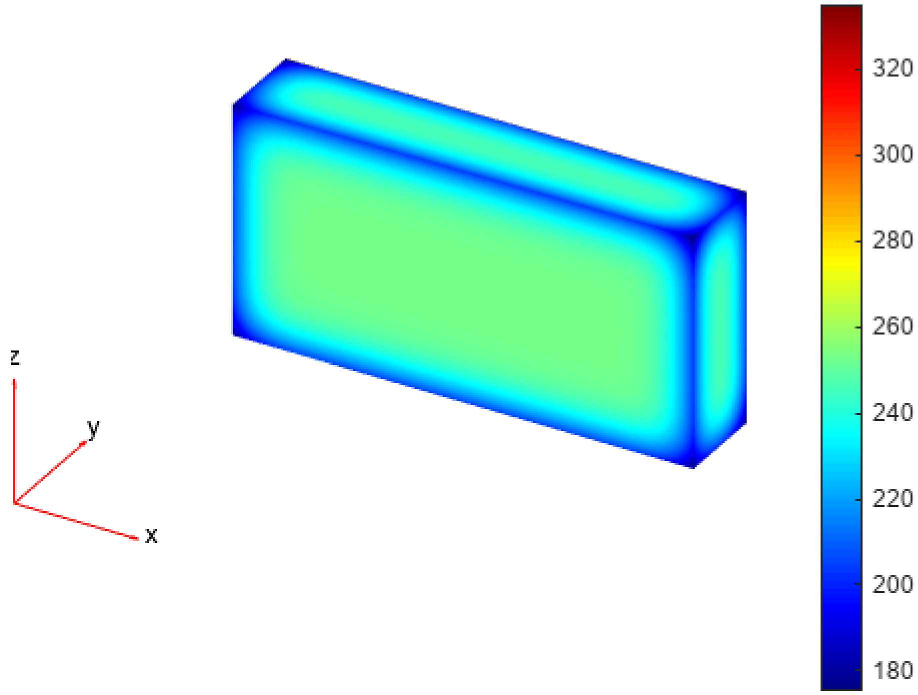
```
generateMesh(model);
tlist = 0:20;
results = solvepde(model,tlist);
```

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))
```



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`).

Tips

- If the Newton iteration does not converge, `solvepde` displays the error message `Too many iterations` or `Stepsize too small`.
- If the initial guess produces matrices containing `NaN` or `Inf` elements, `solvepde` displays the error message `Unsuitable initial guess U0 (default: U0 = 0)`.
- If you have very small coefficients, or very small geometric dimensions, `solvepde` can fail to converge, or can converge to an incorrect solution. In this case, you might obtain better results by scaling the coefficients or geometry dimensions to be of order one.

Version History

Introduced in R2016a

See Also

`applyBoundaryCondition` | `setInitialConditions` | `solvepdeeig` | `specifyCoefficients` | `PDEModel`

Topics

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

solvepdeeig

Package: `pde`

Solve PDE eigenvalue problem specified in a PDEModel

Syntax

`result = solvepdeeig(model,evr)`

Description

`result = solvepdeeig(model,evr)` solves the PDE eigenvalue problem in `model` for eigenvalues in the range `evr`. If the range does not contain any eigenvalues, `solvepdeeig` returns an `EigenResults` object with the empty `EigenVectors`, `EigenValues`, and `Mesh` properties.

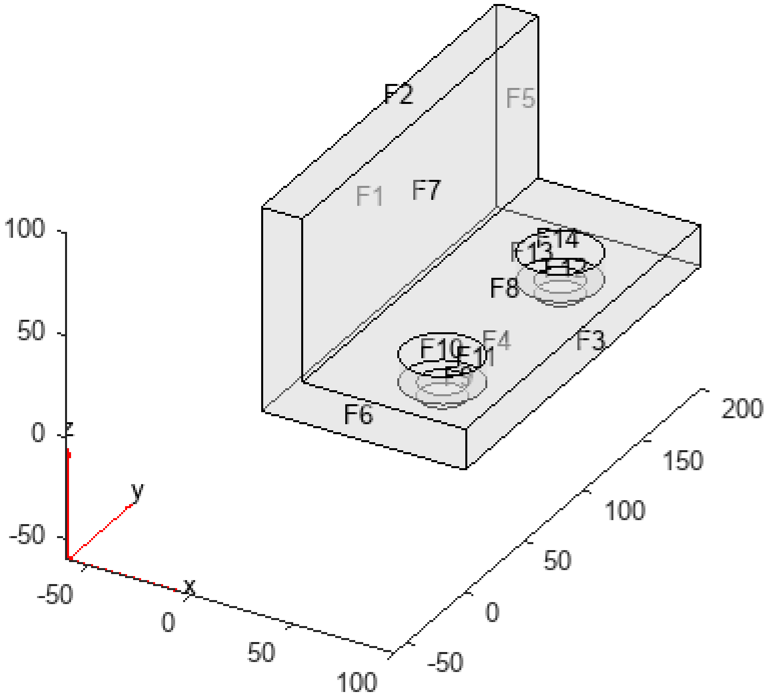
Examples

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",0.4)
```



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 “Linear Elasticity Equations” on page 3-156. When specifying the `f`-coefficient, assume that all body forces are zero.

```
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]);
```

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=  18.52,  New conv eig=  0
              Basis= 11,  Time=  18.59,  New conv eig=  0
              Basis= 12,  Time=  18.69,  New conv eig=  0
              Basis= 13,  Time=  18.78,  New conv eig=  0
```

```
               Basis= 14,  Time=  18.89,  New conv eig=  0
               Basis= 15,  Time=  19.16,  New conv eig=  2
               Basis= 16,  Time=  19.39,  New conv eig=  2
               Basis= 17,  Time=  19.59,  New conv eig=  2
               Basis= 18,  Time=  19.80,  New conv eig=  4
End of sweep: Basis= 18,  Time=  19.86,  New conv eig=  4
               Basis= 14,  Time=  21.08,  New conv eig=  0
End of sweep: Basis= 14,  Time=  21.11,  New conv eig=  0
```

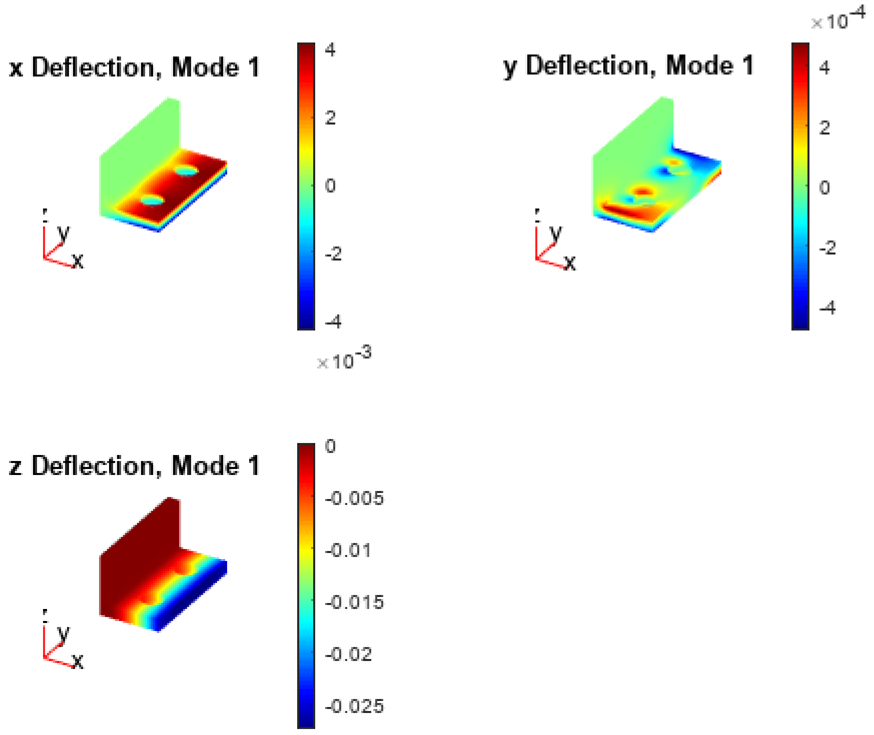
How many results did `solvepdeeig` return?

```
length(results.Eigenvalues)
```

```
ans = 3
```

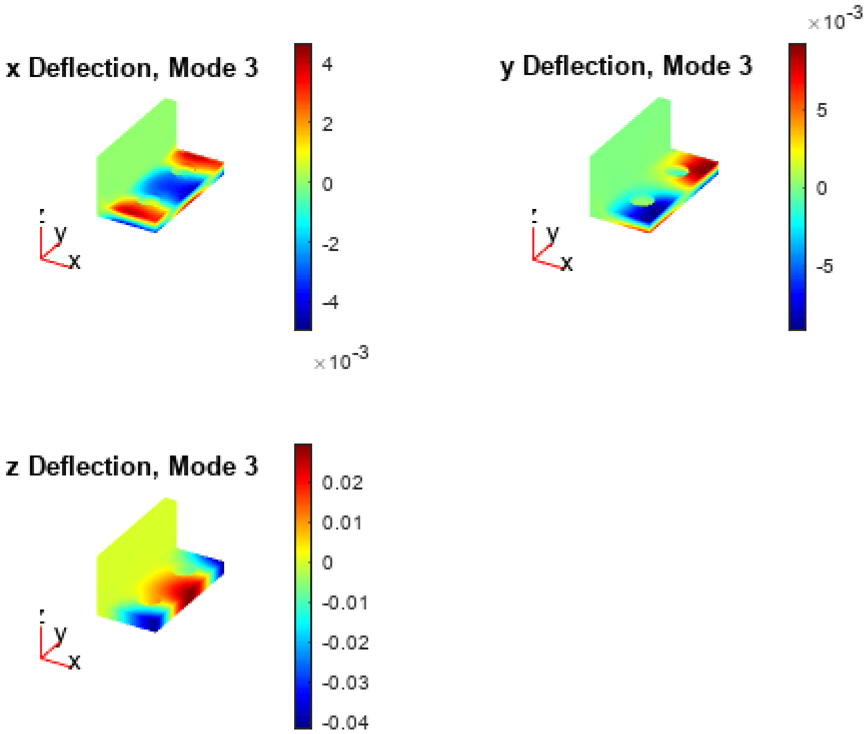
Plot the solution on the geometry boundary for the lowest eigenvalue.

```
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")
```



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. If the range `env` does not contain any eigenvalues, the returned `EigenResults` object has the empty `EigenVectors`, `EigenValues`, and `Mesh` properties.

Tips

- The equation coefficients cannot depend on the solution `u` or its gradient.

Version History

Introduced in R2016a

See Also

`applyBoundaryCondition` | `solvepde` | `specifyCoefficients` | `PDEModel`

Topics

“Eigenvalues and Eigenmodes of L-Shaped Membrane” on page 3-305
“Eigenvalues and Eigenmodes of Square” on page 3-317
“Solve Problems Using PDEModel Objects” on page 2-2

specifyCoefficients

Package: `pde`

Specify coefficients in a PDE model

Syntax

```
specifyCoefficients(model,Name,Value)
specifyCoefficients(model,Name,Value,RegionType,RegionID)
CA = specifyCoefficients( ___ )
```

Description

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.

`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.

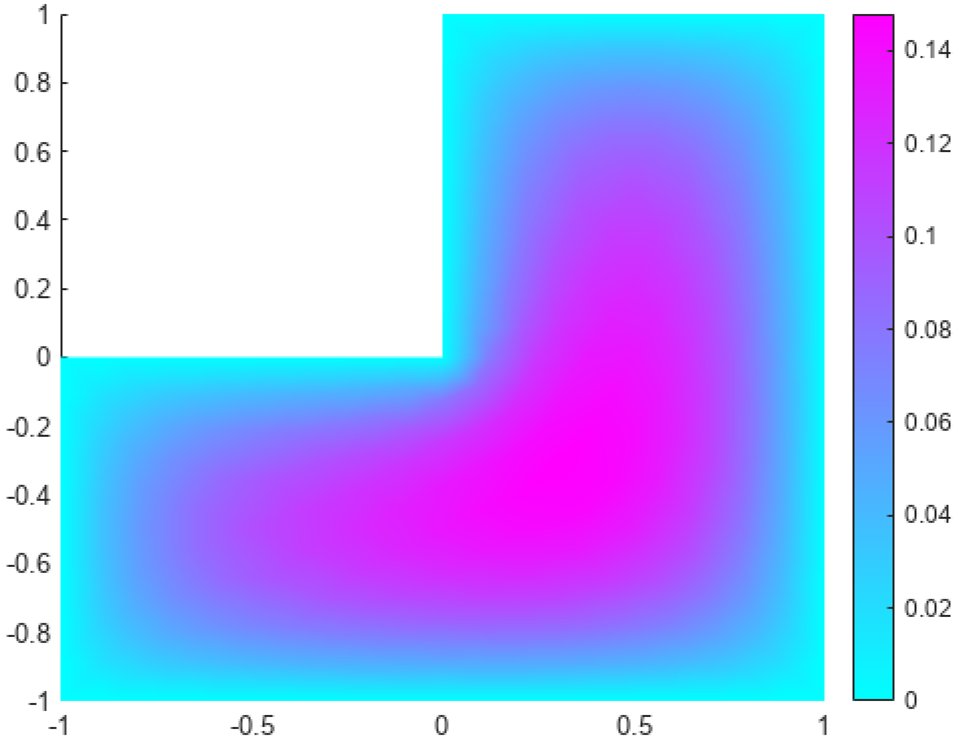
```
model = createpde();
geometryFromEdges(model,@lshapeg);
specifyCoefficients(model,"m",0,...
                          "d",0,...
                          "c",1,...
                          "a",0,...
                          "f",1);
```

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 = @(location,state)location.y.^2.*tanh(location.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)
```

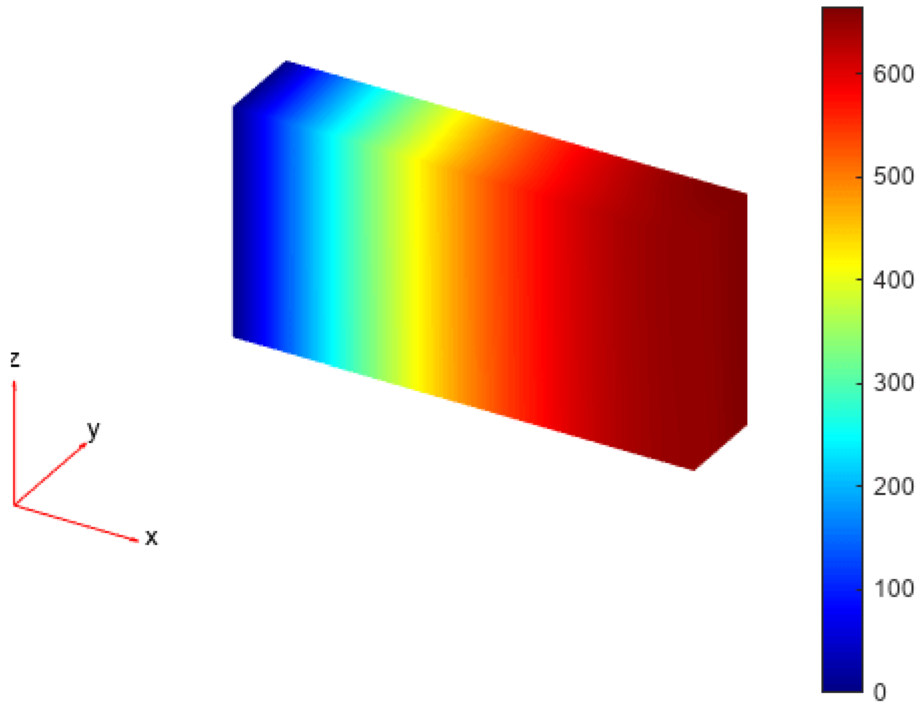
```
CA = 
  CoefficientAssignment with properties:

    RegionType: 'cell'
      RegionID: 1
             m: 0
             d: 0
             c: 1
             a: 0
             f: @(location,state)location.y.^2.*tanh(location.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.

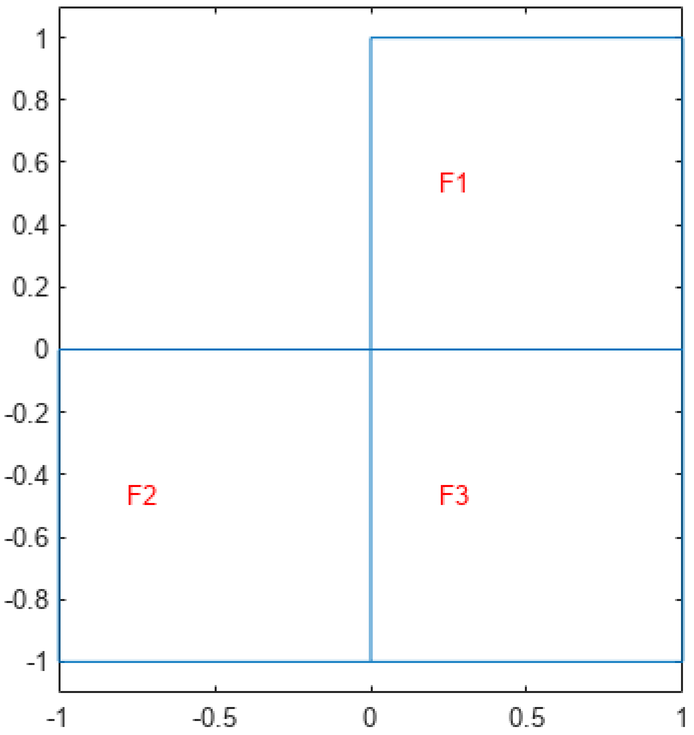
```
pdeplot3D(model,"ColorMapData",results.NodalSolution)
```



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])
```

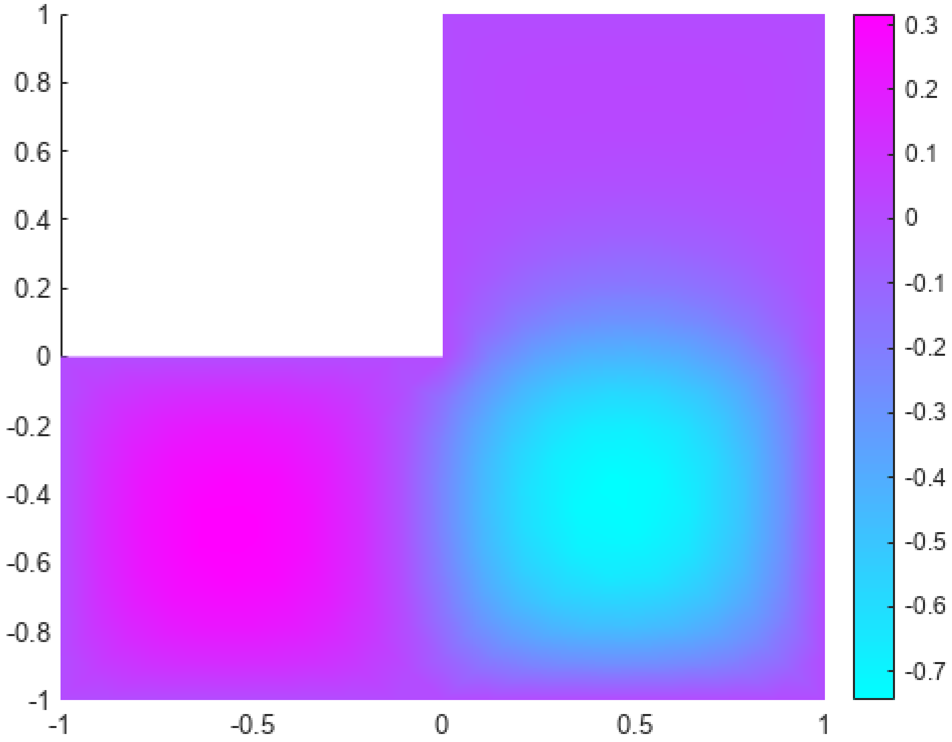


Set the `c` coefficient to 1 in all domains, but the `f` coefficient to 1 in subdomain 1, 5 in subdomain 2, and -8 in subdomain 3. Set all other coefficients to 0.

```
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)
```



Input Arguments

`model` — PDE model

`PDEModel` object

PDE model, specified as a `PDEModel` object.

Example: `model = createpde`

Name-Value Pair Arguments

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 2-107.

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 2-107.

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 5-1205.

Specify `0` if the term is not part of your problem.

Example: `specifyCoefficients("m",0,"d",@dcoef,"c",1,"a",0,"f",1,"Face",1:4)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

c — Second-order space derivative coefficient

scalar | column vector | function handle

Second-order space derivative coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see “c Coefficient for specifyCoefficients” on page 2-92.

Example: `specifyCoefficients("m",0,"d",0,"c",@ccoef,"a",0,"f",1,"Face",1:4)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

a — Solution multiplier coefficient

scalar | column vector | function handle

Solution multiplier coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see “m, d, or a Coefficient for specifyCoefficients” on page 2-107.

Specify `0` if the term is not part of your problem.

Example: `specifyCoefficients("m",0,"d",0,"c",1,"a",@acoef,"f",1,"Face",1:4)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

f — Source coefficient

scalar | column vector | function handle

Source coefficient, specified as a scalar, column vector, or function handle. For details on the sizes, and for details of the function handle form of the coefficient, see “f Coefficient for specifyCoefficients” on page 2-90.

Specify `0` if the term is not part of your problem.

Example: `specifyCoefficients("m",0,"d",0,"c",1,"a",0,"f",@fcoeff,"Face",1:4)`

Data Types: `double` | `function_handle`
Complex Number Support: Yes

`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` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

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.

More About

`d` Coefficient When `m` is Nonzero

The `d` coefficient takes a special matrix form when `m` is nonzero. You must specify `d` as a matrix of a particular size, and not as a function handle.

`d` represents a damping coefficient in the case of nonzero `m`. To specify `d`, perform these two steps:

1. Call `results = assembleFEMatrices(...)` for the problem with your original coefficients and using `d` = 0. Use the default `"none"` method for `assembleFEMatrices`.
2. Take the `d` coefficient as a matrix of size `results.M`. Generally, `d` is either proportional to `results.M`, or is a linear combination of `results.M` and `results.K`.

See “Dynamics of Damped Cantilever Beam” on page 3-21.

Tips

- For eigenvalue equations, the coefficients cannot depend on the solution `u` or its gradient.
- You can transform a partial differential equation into the required form by using Symbolic Math Toolbox. The `pdeCoefficients` converts a PDE into the required form and extracts the coefficients into a structure that can be used by `specifyCoefficients`.

The `pdeCoefficients` function also can return a structure of symbolic expressions, in which case you need to use `pdeCoefficientsToDouble` to convert these expressions to double format before passing them to `specifyCoefficients`.

Version History

Introduced in R2016a

See Also

`findCoefficients` | `PDEModel` | `pdeCoefficients` | `pdeCoefficientsToDouble`

Topics

"m, d, or a Coefficient for specifyCoefficients" on page 2-107
"c Coefficient for specifyCoefficients" on page 2-92
"f Coefficient for specifyCoefficients" on page 2-90
"Specify Nonconstant PDE Coefficients" on page 2-146
"Solve Problems Using PDEModel Objects" on page 2-2
"Put Equations in Divergence Form" on page 2-87

ModalStructuralResults

Structural modal analysis solution

Description

A `ModalStructuralResults` object contains the natural frequencies and modal displacement in a form convenient for plotting and postprocessing.

Modal displacement is reported for the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. The modal displacement values at the nodes appear as an `FEStruct` object in the `ModeShapes` property. The properties of this object contain the components of the displacement at the nodal locations.

You can use a `ModalStructuralResults` object to approximate solutions for transient dynamics problems. For details, see `solve`.

Creation

Solve a modal analysis problem by using the `solve` function. This function returns a modal structural solution as a `ModalStructuralResults` object.

Properties

`NaturalFrequencies` — Natural frequencies
column vector

This property is read-only.

Natural frequencies of the structure, returned as a column vector.

Data Types: `double`

`ModeShapes` — Modal displacement values at nodes
`FEStruct` object

This property is read-only.

Modal displacement values at the nodes, returned as an `FEStruct` object. The properties of this object contain components of modal displacement at nodal locations.

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as a `FEMesh` object. For details, see FEMesh.

Examples

Solution to Modal Analysis Structural Model

Find the fundamental (lowest) mode of a 2-D cantilevered beam, assuming prevalence of the plane-stress condition.

Specify geometric and structural properties of the beam, along with a unit plane-stress thickness.

```
length = 5;
height = 0.1;
E = 3E7;
nu = 0.3;
rho = 0.3/386;
```

Create a modal plane-stress model, assign a geometry, and generate a mesh.

```
structuralmodel = createpde("structural","modal-planestress");
gdm = [3;4;0;length;length;0;0;0;height;height];
g = decsg(gdm,'S1',('S1')');
geometryFromEdges(structuralmodel,g);
```

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

```
hmax = height/5;
msh=generateMesh(structuralmodel,"Hmax",hmax);
```

Specify the structural properties and boundary constraints.

```
structuralProperties(structuralmodel,"YoungsModulus",E, ...
                                     "MassDensity",rho, ...
                                     "PoissonsRatio",nu);
structuralBC(structuralmodel,"Edge",4,"Constraint","fixed");
```

Compute the analytical fundamental frequency (Hz) using the beam theory.

```
I = height^3/12;
analyticalOmega1 = 3.516*sqrt(E*I/(length^4*(rho*height)))/(2*pi)
```

```
analyticalOmega1 = 126.9498
```

Specify a frequency range that includes an analytically computed frequency and solve the model.

```
modalresults = solve(structuralmodel,"FrequencyRange",[0,1e6])
```

```
modalresults = 
  ModalStructuralResults with properties:

    NaturalFrequencies: [32x1 double]
            ModeShapes: [1x1 FEStruct]
                  Mesh: [1x1 FEMesh]
```

The solver finds natural frequencies and modal displacement values at nodal locations. To access these values, use `modalresults.NaturalFrequencies` and `modalresults.ModeShapes`.

```
modalresults.NaturalFrequencies/(2*pi)
```

```
ans = 32×1
10⁵ ×
```

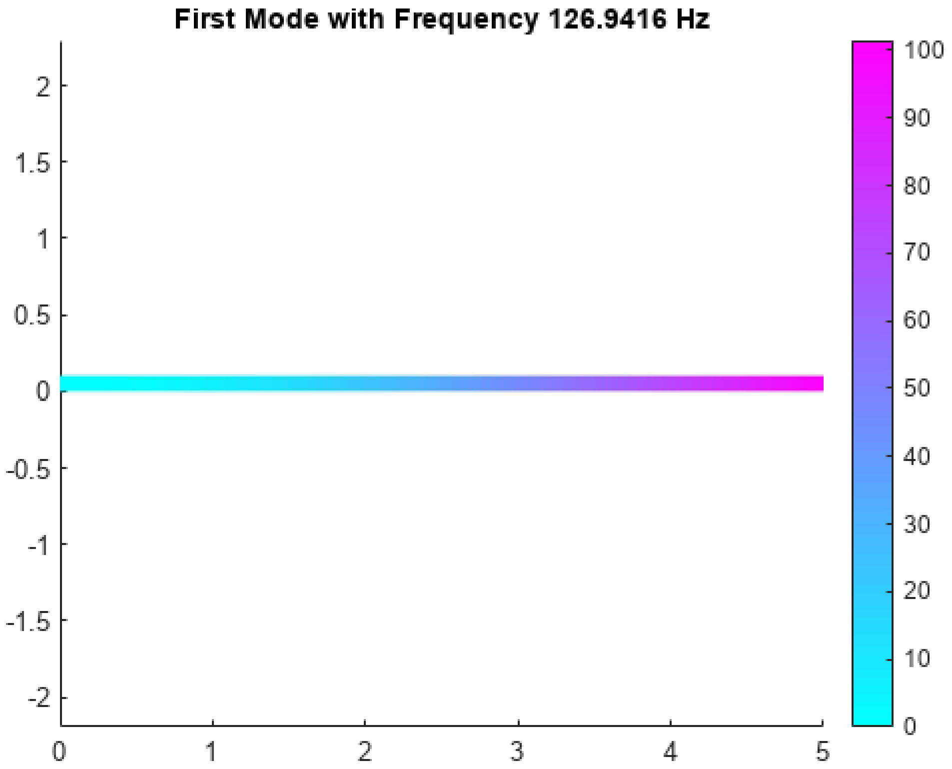
```
    0.0013
    0.0079
    0.0222
    0.0433
    0.0711
    0.0983
    0.1055
    0.1462
    0.1930
    0.2455
       ⋮
```

```
modalresults.ModeShapes
```

```
ans = 
  FEStruct with properties:

           ux: [6511x32 double]
           uy: [6511x32 double]
    Magnitude: [6511x32 double]
```

Plot the *y*-component of the solution for the fundamental frequency.

```
pdeplot(structuralmodel,"XYData",modalresults.ModeShapes.uy(:,1))
title(['First Mode with Frequency ', ...
        num2str(modalresults.NaturalFrequencies(1)/(2*pi)),' Hz'])
axis equal
```



Version History

Introduced in R2018a

See Also

`StructuralModel` | `solve` | `StaticStructuralResults` | `TransientStructuralResults`

FrequencyStructuralResults

Frequency response structural solution and its derived quantities

Description

A `FrequencyStructuralResults` object contains the displacement, velocity, and acceleration in a form convenient for plotting and postprocessing.

Displacement, velocity, and acceleration are reported for the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. The displacement, velocity, and acceleration values at the nodes appear as `FEStruct` objects in the `Displacement`, `Velocity`, and `Acceleration` properties. The properties of these objects contain the components of the displacement, velocity, and acceleration at the nodal locations.

To evaluate the stress, strain, von Mises stress, principal stress, and principal strain at the nodal locations, use `evaluateStress`, `evaluateStrain`, `evaluateVonMisesStress`, `evaluatePrincipalStress`, and `evaluatePrincipalStrain`, respectively.

To evaluate the reaction forces on a specified boundary, use `evaluateReaction`.

To interpolate the displacement, velocity, acceleration, stress, strain, and von Mises stress to a custom grid, such as the one specified by `meshgrid`, use `interpolateDisplacement`, `interpolateVelocity`, `interpolateAcceleration`, `interpolateStress`, `interpolateStrain`, and `interpolateVonMisesStress`, respectively.

For a frequency response model with damping, the results are complex. Use functions such as `abs` and `angle` to obtain real-valued results, such as the magnitude and phase. See "Solution to Frequency Response Structural Model with Damping" on page 5-1212.

Creation

Solve a frequency response problem by using the `solve` function. This function returns a frequency response structural solution as a `FrequencyStructuralResults` object.

Properties

`Displacement` — Displacement values at nodes
`FEStruct` object

This property is read-only.

Displacement values at the nodes, returned as an `FEStruct` object. The properties of this object contain the components of the displacement at the nodal locations.

`Velocity` — Velocity values at nodes
`FEStruct` object

This property is read-only.

Velocity values at the nodes, returned as an `FEStruct` object. The properties of this object contain the components of the velocity at the nodal locations.

`Acceleration` — Acceleration values at nodes
`FEStruct` object

This property is read-only.

Acceleration values at the nodes, returned as an `FEStruct` object. The properties of this object contain the components of the acceleration at the nodal locations.

`SolutionFrequencies` — Solution frequencies
real vector

This property is read-only.

Solution frequencies, returned as a real vector. `SolutionFrequencies` is the same as the `flist` input to `solve`.

Data Types: `double`

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as a `FEMesh` object. For details, see FEMesh.

Object Functions

evaluateStress	Evaluate stress for dynamic structural analysis problem
evaluateStrain	Evaluate strain for dynamic structural analysis problem
evaluateVonMisesStress	Evaluate von Mises stress for dynamic structural analysis problem
evaluateReaction	Evaluate reaction forces on boundary
evaluatePrincipalStress	Evaluate principal stress at nodal locations
evaluatePrincipalStrain	Evaluate principal strain at nodal locations
interpolateDisplacement	Interpolate displacement at arbitrary spatial locations
interpolateVelocity	Interpolate velocity at arbitrary spatial locations for all time or frequency steps for dynamic structural model
interpolateAcceleration	Interpolate acceleration at arbitrary spatial locations for all time or frequency steps for dynamic structural model
interpolateStress	Interpolate stress at arbitrary spatial locations
interpolateStrain	Interpolate strain at arbitrary spatial locations
interpolateVonMisesStress	Interpolate von Mises stress at arbitrary spatial locations

Examples

Solution to Frequency Response Structural Model with Damping

Solve a frequency response problem with damping. The resulting displacement values are complex. To obtain the magnitude and phase of displacement, use the `abs` and `angle` functions, respectively. To speed up computations, solve the model using the results of modal analysis.

Modal Analysis

Create a modal analysis model for a 3-D problem.

```
modelM = createpde("structural","modal-solid");
```

Create the geometry and include it in the model.

```
gm = multicuboid(10,10,0.025);
modelM.Geometry = gm;
```

Generate a mesh.

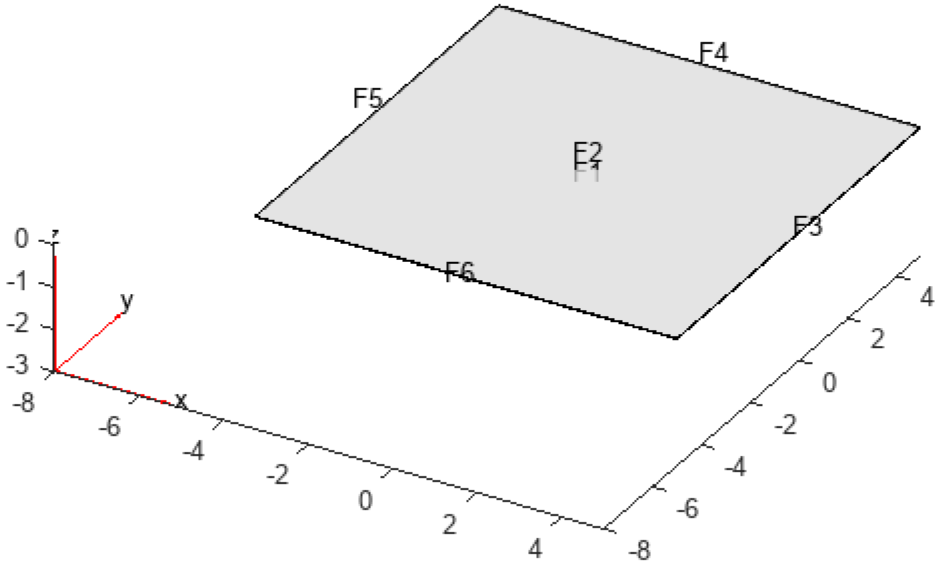
```
msh = generateMesh(modelM);
```

Specify Young's modulus, Poisson's ratio, and the mass density of the material.

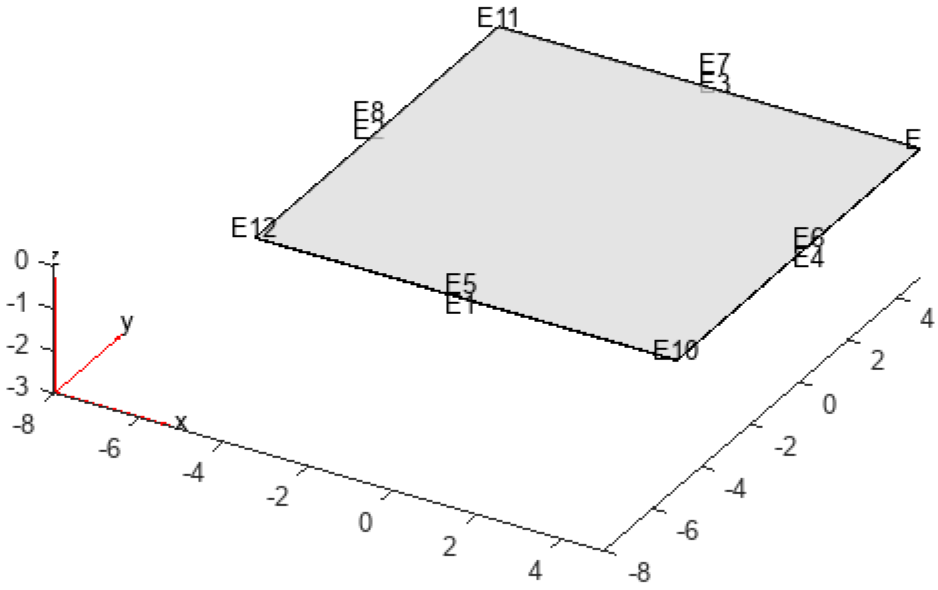
```
structuralProperties(modelM,"YoungsModulus",2E11, ...
                            "PoissonsRatio",0.3, ...
                            "MassDensity",8000);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face and edge labels.

```
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.5)
```



```
figure
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.5)
```



Specify constraints on the sides of the plate (faces 3, 4, 5, and 6) to prevent rigid body motions.

```
structuralBC(modelM,"Face",[3,4,5,6],"Constraint","fixed");
```

Solve the problem for the frequency range from `-Inf` to `12*pi`.

```
Rm = solve(modelM,"FrequencyRange",[-Inf,12*pi]);
```

Frequency Response Analysis

Create a frequency response analysis model for a 3-D problem.

```
modelFR = createpde("structural","frequency-solid");
```

Use the same geometry and mesh as you used for the modal analysis.

```
modelFR.Geometry = gm;
modelFR.Mesh = msh;
```

Specify the same values for Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelFR,"YoungsModulus",2E11, ...
                             "PoissonsRatio",0.3, ...
                             "MassDensity",8000);
```

Specify the same constraints on the sides of the plate to prevent rigid body modes.

```
structuralBC(modelFR,"Face",[3,4,5,6],"Constraint","fixed");
```

Specify the pressure loading on top of the plate (face 2) to model an ideal impulse excitation. In the frequency domain, this pressure pulse is uniformly distributed across all frequencies.

```
structuralBoundaryLoad(modelFR,"Face",2,"Pressure",1E2);
```

First, solve the model without damping.

```
flist = [0,1,1.5,linspace(2,3,100),3.5,4,5,6]*2*pi;
RfrModalU = solve(modelFR,flist,"ModalResults",Rm);
```

Now, solve the model with damping equal to 2% of critical damping for all modes.

```
structuralDamping(modelFR,"Zeta",0.02);
RfrModalAll = solve(modelFR,flist,"ModalResults",Rm);
```

Solve the same model with frequency-dependent damping. In this example, use the solution frequencies from `flist` and damping values between 1% and 10% of critical damping.

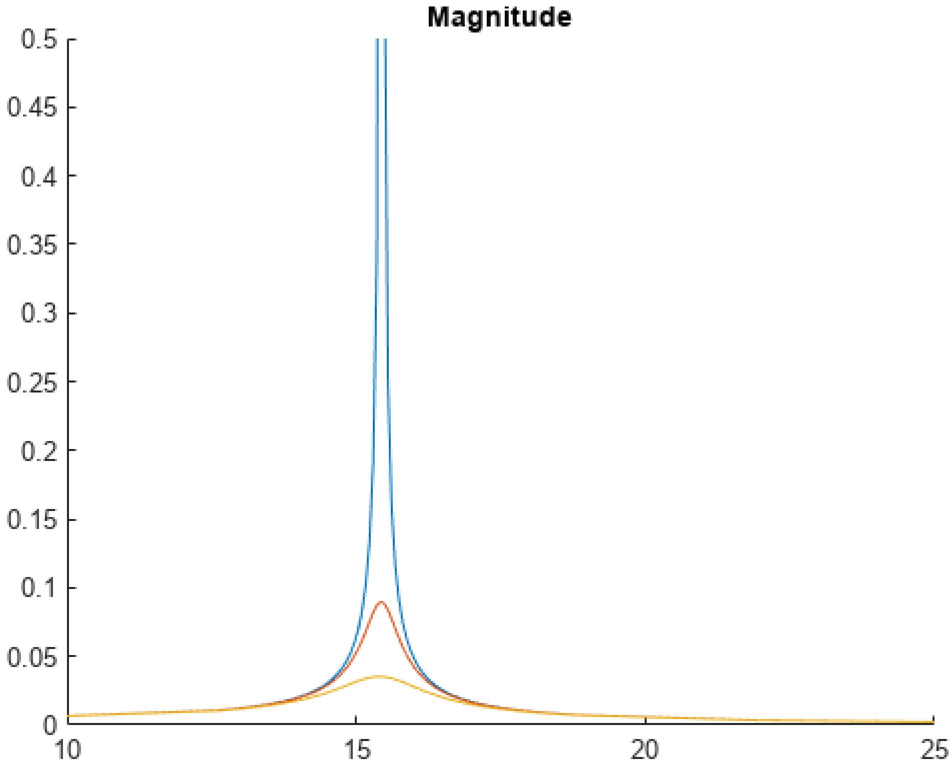
```
omega = flist;
zeta = linspace(0.01,0.1,length(omega));
zetaW = @(omegaMode) interp1(omega,zeta,omegaMode);
structuralDamping(modelFR,"Zeta",zetaW);

RfrModalFD = solve(modelFR,flist,"ModalResults",Rm);
```

Interpolate the displacement at the center of the top surface of the plate for all three cases.

```
iDispU = interpolateDisplacement(RfrModalU,[0;0;0.025]);
iDispAll = interpolateDisplacement(RfrModalAll,[0;0;0.025]);
iDispFD = interpolateDisplacement(RfrModalFD,[0;0;0.025]);
```

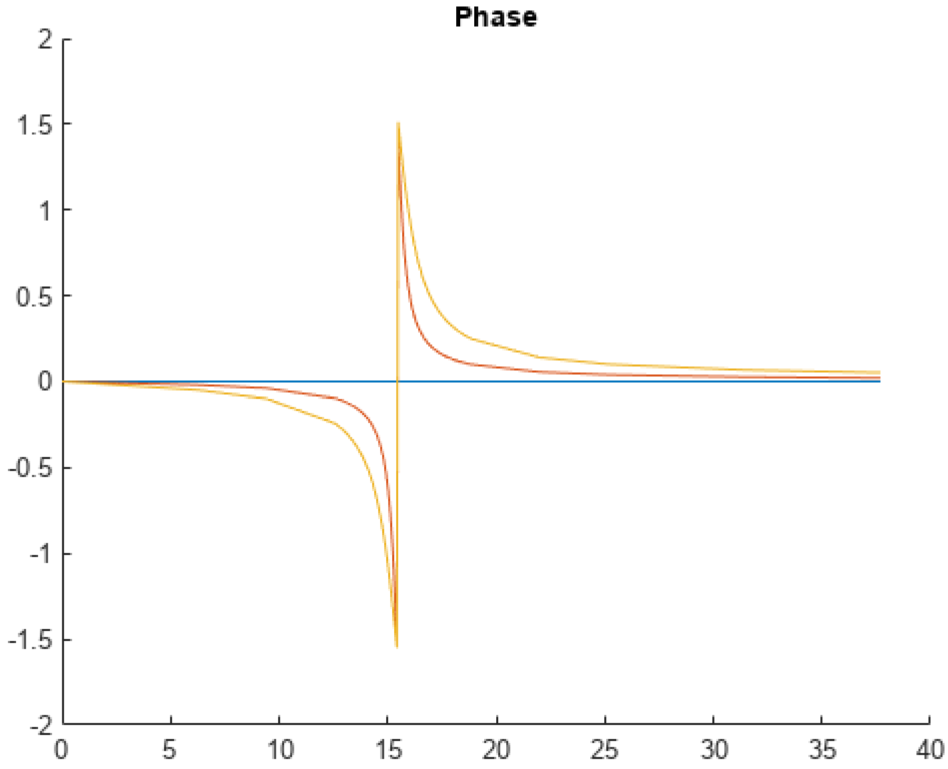
Plot the magnitude of the displacement. Zoom in on the frequencies around the first mode.

```
figure
hold on
plot(RfrModalU.SolutionFrequencies,abs(iDispU.Magnitude));
plot(RfrModalAll.SolutionFrequencies,abs(iDispAll.Magnitude));
plot(RfrModalFD.SolutionFrequencies,abs(iDispFD.Magnitude));
title("Magnitude")
xlim([10 25])
ylim([0 0.5])
```



Plot the phase of the displacement.

```
figure
hold on
plot(RfrModalU.SolutionFrequencies,angle(iDispU.Magnitude));
plot(RfrModalAll.SolutionFrequencies,angle(iDispAll.Magnitude));
plot(RfrModalFD.SolutionFrequencies,angle(iDispFD.Magnitude));
title("Phase")
```



Version History

Introduced in R2019b

See Also

`StructuralModel` | `solve` | `StaticStructuralResults` | `ModalStructuralResults` | `TransientStructuralResults`

StaticStructuralResults

Static structural solution and its derived quantities

Description

A `StaticStructuralResults` object contains the displacement, stress, strain, and von Mises stress in a form convenient for plotting and postprocessing.

Displacements, stresses, and strains are reported for the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. Displacement values at the nodes appear as an `FEStruct` object in the `Displacement` property. The properties of this object contain components of displacement at nodal locations.

Stress and strain values at the nodes appear as `FEStruct` objects in the `Stress` and `Strain` properties, respectively.

von Mises stress at the nodes appears as a vector in the `VonMisesStress` property.

To interpolate the displacement, stress, strain, and von Mises stress to a custom grid, such as the one specified by `meshgrid`, use `interpolateDisplacement`, `interpolateStress`, `interpolateStrain`, and `interpolateVonMisesStress`, respectively.

To evaluate reaction forces on a specified boundary, use `evaluateReaction`. To evaluate principal stress and principal strain at nodal locations, use `evaluatePrincipalStress` and `evaluatePrincipalStrain`, respectively.

Creation

Solve a static linear elasticity problem by using the `solve` function. This function returns a static structural solution as a `StaticStructuralResults` object.

Properties

`Displacement` — Displacement values at nodes
`FEStruct` object

This property is read-only.

Displacement values at the nodes, returned as an `FEStruct` object. The properties of this object contain components of displacement at nodal locations.

`Stress` — Stress values at nodes
`FEStruct` object

This property is read-only.

Stress values at the nodes, returned as an `FEStruct` object. The properties of this object contain components of stress at nodal locations.

`Strain` — Strain values at nodes

`FEStruct` object

This property is read-only.

Strain values at the nodes, returned as an `FEStruct` object. The properties of this object contain components of strain at nodal locations.

`VonMisesStress` — Von Mises stress values at nodes

vector

This property is read-only.

Von Mises stress values at the nodes, returned as a vector.

Data Types: `double`

`Mesh` — Finite element mesh

`FEMesh` object

This property is read-only.

Finite element mesh, returned as a `FEMesh` object. For details, see FEMesh.

Object Functions

interpolateDisplacement	Interpolate displacement at arbitrary spatial locations
interpolateStress	Interpolate stress at arbitrary spatial locations
interpolateStrain	Interpolate strain at arbitrary spatial locations
interpolateVonMisesStress	Interpolate von Mises stress at arbitrary spatial locations
evaluateReaction	Evaluate reaction forces on boundary
evaluatePrincipalStress	Evaluate principal stress at nodal locations
evaluatePrincipalStrain	Evaluate principal strain at nodal locations

Examples

Solution to Static Structural Model

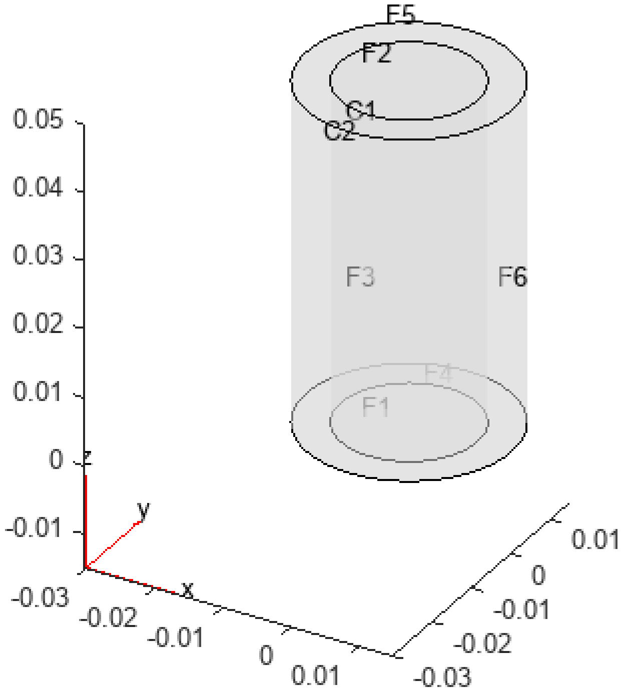
Solve a static structural model representing a bimetallic cable under tension.

Create a static structural model for a solid (3-D) problem.

```
structuralmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicylinder([0.01 0.015],0.05);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on", ...
                         "CellLabels","on", ...
                         "FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28);
structuralProperties(structuralmodel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[1,4],"Constraint","fixed");
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralmodel,"Face",[2,5], ...
                                       "SurfaceTraction",[0;0;100]);
```

Generate a mesh and solve the problem.

```
generateMesh(structuralmodel);
structuralresults = solve(structuralmodel)
```

```
structuralresults = 
  StaticStructuralResults with properties:

      Displacement: [1x1 FEStruct]
            Strain: [1x1 FEStruct]
            Stress: [1x1 FEStruct]
    VonMisesStress: [22281x1 double]
```

```
             Mesh: [1x1 FEMesh]
```

The solver finds the values of the displacement, stress, strain, and von Mises stress at the nodal locations. To access these values, use `structuralresults.Displacement`, `structuralresults.Stress`, and so on. The displacement, stress, and strain values at the nodal locations are returned as `FEStruct` objects with the properties representing their components. Note that properties of an `FEStruct` object are read-only.

```
structuralresults.Displacement
```

```
ans = 
  FEStruct with properties:

           ux: [22281x1 double]
           uy: [22281x1 double]
           uz: [22281x1 double]
    Magnitude: [22281x1 double]
```

```
structuralresults.Stress
```

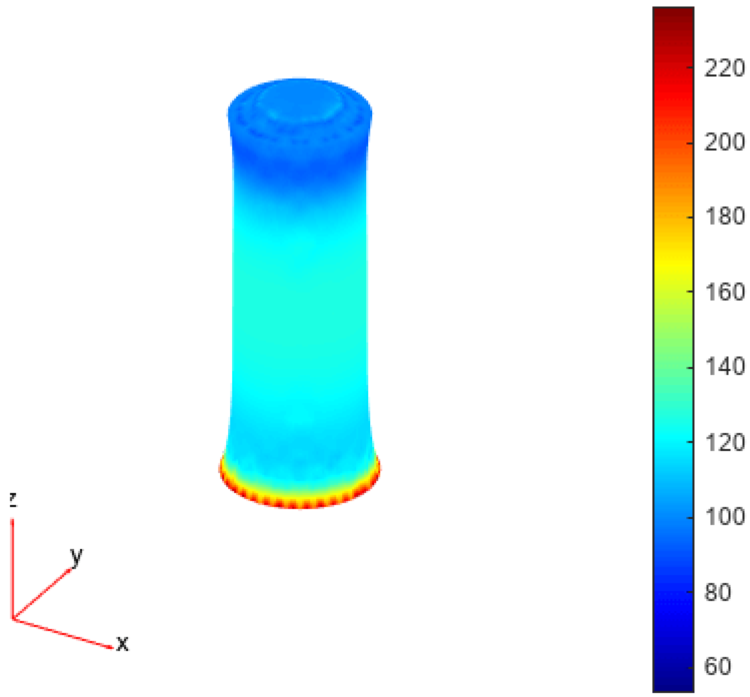
```
ans = 
  FEStruct with properties:

    sxx: [22281x1 double]
    syy: [22281x1 double]
    szz: [22281x1 double]
    syz: [22281x1 double]
    sxz: [22281x1 double]
    sxy: [22281x1 double]
```

```
structuralresults.Strain
```

```
ans = 
  FEStruct with properties:

    exx: [22281x1 double]
    eyy: [22281x1 double]
    ezz: [22281x1 double]
    eyz: [22281x1 double]
    exz: [22281x1 double]
    exy: [22281x1 double]
```

Plot the deformed shape with the *z*-component of normal stress.

```
pdeplot3D(structuralmodel, ...
          "ColorMapData",structuralresults.Stress.szz, ...
          "Deformation",structuralresults.Displacement)
```



Version History

Introduced in R2017b

See Also

`StructuralModel` | `solve` | `TransientStructuralResults` | `ModalStructuralResults`

TransientStructuralResults

Transient structural solution and its derived quantities

Description

A `TransientStructuralResults` object contains the displacement, velocity, and acceleration in a form convenient for plotting and postprocessing.

Displacement, velocity, and acceleration are reported for the nodes of the triangular or tetrahedral mesh generated by `generateMesh`. The displacement, velocity, and acceleration values at the nodes appear as `FEStruct` objects in the `Displacement`, `Velocity`, and `Acceleration` properties. The properties of these objects contain the components of the displacement, velocity, and acceleration at the nodal locations.

To evaluate the stress, strain, von Mises stress, principal stress, and principal strain at the nodal locations, use `evaluateStress`, `evaluateStrain`, `evaluateVonMisesStress`, `evaluatePrincipalStress`, and `evaluatePrincipalStrain`, respectively.

To evaluate the reaction forces on a specified boundary, use `evaluateReaction`.

To interpolate the displacement, velocity, acceleration, stress, strain, and von Mises stress to a custom grid, such as the one specified by `meshgrid`, use `interpolateDisplacement`, `interpolateVelocity`, `interpolateAcceleration`, `interpolateStress`, `interpolateStrain`, and `interpolateVonMisesStress`, respectively.

Creation

Solve a dynamic linear elasticity problem by using the `solve` function. This function returns a transient structural solution as a `TransientStructuralResults` object.

Properties

`Displacement` — Displacement values at nodes
`FEStruct` object

This property is read-only.

Displacement values at the nodes, returned as an `FEStruct` object. The properties of this object contain components of displacement at nodal locations.

`Velocity` — Velocity values at nodes
`FEStruct` object

This property is read-only.

Velocity values at the nodes, returned as an `FEStruct` object. The properties of this object contain components of velocity at nodal locations.

`Acceleration` — Acceleration values at nodes
`FEStruct` object

This property is read-only.

Acceleration values at the nodes, returned as an `FEStruct` object. The properties of this object contain components of acceleration at nodal locations.

`SolutionTimes` — Solution times
real vector

This property is read-only.

Solution times, returned as a real vector. `SolutionTimes` is the same as the `tlist` input to `solve`.

Data Types: `double`

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as a `FEMesh` object. For details, see FEMesh.

Object Functions

evaluateStress	Evaluate stress for dynamic structural analysis problem
evaluateStrain	Evaluate strain for dynamic structural analysis problem
evaluateVonMisesStress	Evaluate von Mises stress for dynamic structural analysis problem
evaluateReaction	Evaluate reaction forces on boundary
evaluatePrincipalStress	Evaluate principal stress at nodal locations
evaluatePrincipalStrain	Evaluate principal strain at nodal locations
interpolateDisplacement	Interpolate displacement at arbitrary spatial locations
interpolateVelocity	Interpolate velocity at arbitrary spatial locations for all time or frequency steps for dynamic structural model
interpolateAcceleration	Interpolate acceleration at arbitrary spatial locations for all time or frequency steps for dynamic structural model
interpolateStress	Interpolate stress at arbitrary spatial locations
interpolateStrain	Interpolate strain at arbitrary spatial locations
interpolateVonMisesStress	Interpolate von Mises stress at arbitrary spatial locations

Examples

Solution to Transient Structural Model

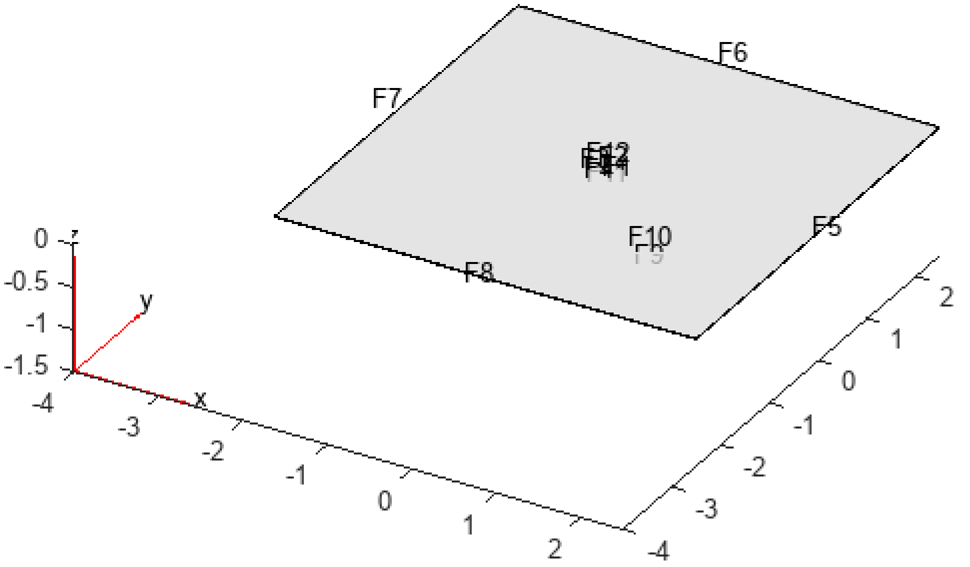
Solve for the transient response of a thin 3-D plate under a harmonic load at the center.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

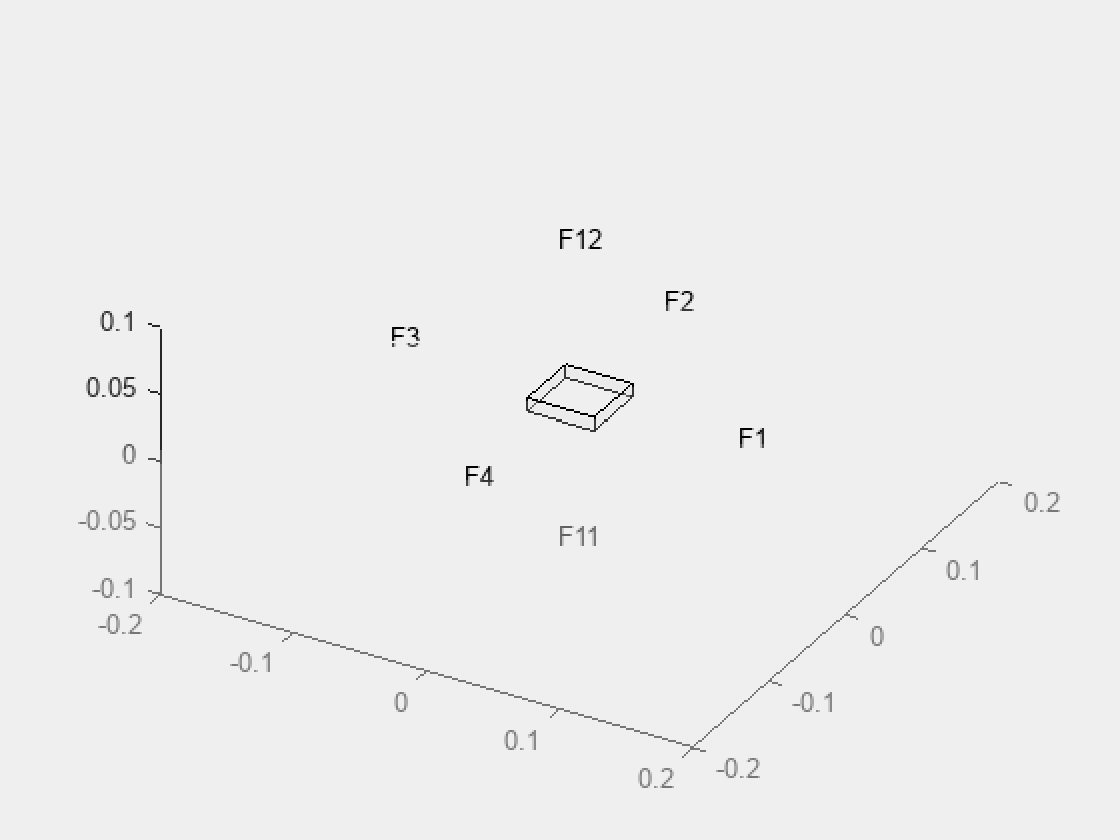
Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid([5,0.05],[5,0.05],0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Zoom in to see the face labels on the small plate at the center.

```
figure
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.25)
axis([-0.2 0.2 -0.2 0.2 -0.1 0.1])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Specify that all faces on the periphery of the thin 3-D plate are fixed boundaries.

```
structuralBC(structuralmodel,"Constraint","fixed","Face",5:8);
```

Apply a sinusoidal pressure load on the small face at the center of the plate.

```
structuralBoundaryLoad(structuralmodel,"Face",12, ...
                                       "Pressure",5E7, ...
                                       "Frequency",25);
```

Generate a mesh with linear elements.

```
generateMesh(structuralmodel,"GeometricOrder","linear","Hmax",0.2);
```

Specify zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0]);
```

Solve the model.

```
tlist = linspace(0,1,300);
structuralresults = solve(structuralmodel,tlist);
```

The solver finds the values of the displacement, velocity, and acceleration at the nodal locations. To access these values, use `structuralresults.Displacement`, `structuralresults.Velocity`, and so on. The displacement, velocity, and acceleration values are returned as `FEStruct` objects with the properties representing their components. Note that properties of an `FEStruct` object are read-only.

```
structuralresults.Displacement
```

```
ans = 
  FEStruct with properties:

           ux: [1873x300 double]
           uy: [1873x300 double]
           uz: [1873x300 double]
    Magnitude: [1873x300 double]
```

```
structuralresults.Velocity
```

```
ans = 
  FEStruct with properties:

           vx: [1873x300 double]
           vy: [1873x300 double]
           vz: [1873x300 double]
    Magnitude: [1873x300 double]
```

```
structuralresults.Acceleration
```

```
ans = 
  FEStruct with properties:

           ax: [1873x300 double]
           ay: [1873x300 double]
           az: [1873x300 double]
    Magnitude: [1873x300 double]
```

Version History

Introduced in R2018a

See Also

`StructuralModel` | `solve` | `StaticStructuralResults` | `ModalStructuralResults`

structuralBC

Package: `pde`

Specify boundary conditions for structural model

Syntax

```
structuralBC(structuralmodel,RegionType,RegionID,"Constraint",Cval)
structuralBC(structuralmodel,RegionType,RegionID,"Displacement",Dval)
structuralBC(structuralmodel,RegionType,RegionID,"XDisplacement",
XDval,"YDisplacement",YDval,"ZDisplacement",ZDval)
structuralBC(structuralmodel,RegionType,RegionID,"RDisplacement",
RDval,"ZDisplacement",ZDval)
```

```
structuralBC(structuralmodel,RegionType,RegionID,"XDisplacement",XDval,
Name,Value)
```

```
structuralBC(structuralmodel,RegionType,RegionID,"Constraint","multipoint")
structuralBC(___,"Reference",Coords)
structuralBC(___,"Reference",Coords,"Radius",R)
```

```
structuralBC(___,"Label",labeltext)
```

```
structuralBC(___,"Vectorized","on")
```

```
bc = structuralBC(___)
```

Description

Standard Boundary Constraints and Displacements

`structuralBC(structuralmodel,RegionType,RegionID,"Constraint",Cval)` specifies one of the standard structural boundary constraints. Here, `Cval` can be `"fixed"`, `"free"`, `"roller"`, or `"symmetric"`. The default value is `"free"`.

Avoid using `"symmetric"` for transient and modal analysis, since the symmetric constraint can prevent the participation of some structural modes.

`structuralBC(structuralmodel,RegionType,RegionID,"Displacement",Dval)` enforces displacement on the boundary of type `RegionType` with `RegionID` ID numbers.

`structuralBC(structuralmodel,RegionType,RegionID,"XDisplacement",XDval,"YDisplacement",YDval,"ZDisplacement",ZDval)` specifies the x-, y-, and z-components of the enforced displacement.

`structuralBC` does not require you to specify all three components. Depending on your structural analysis problem, you can specify one or more components by picking the corresponding arguments and omitting others.

`structuralBC(structuralmodel,RegionType,RegionID,"RDisplacement",RDval,"ZDisplacement",ZDval)` specifies the r- and z-components of the enforced displacement for an axisymmetric model. The radial component (r-component) must be zero on the axis of rotation.

`structuralBC` does not require you to specify both components.

Harmonic, Rectangular, Triangular, and Trapezoidal Displacement Pulses

`structuralBC(structuralmodel,RegionType,RegionID,"XDisplacement",XDval,Name,Value)` specifies the form and duration of the time-varying value of the x-component of the enforced displacement. You can also specify the form and duration of the other components of the displacement as follows:

- `structuralBC(...,"YDisplacement",YDval,Name,Value)` for the y-component.
- `structuralBC(...,"ZDisplacement",ZDval,Name,Value)` for the z-component. Use this syntax for a 3-D or axisymmetric model.
- `structuralBC(...,"RDisplacement",RDval,Name,Value)` for the radial component in an axisymmetric model.

Multipoint Constraint

`structuralBC(structuralmodel,RegionType,RegionID,"Constraint","multipoint")` sets the multipoint constraint using all degrees of freedom on the combination of geometric regions specified by `RegionType` and `RegionID`. The reference location for the constraint is the geometric center of all nodes on the combination of all specified geometric regions.

This syntax is required if you intend to use results obtained with the model order reduction technique in the Simscape Multibody™ `Reduced Order Flexible Solid` block. Simscape models expect the connections at all joints to have six degrees of freedom, while Partial Differential Equation Toolbox uses two or three degrees of freedom at each node. Setting a multipoint constraint ensures that all nodes and all degrees of freedom for the specified geometric regions have a rigid constraint with the geometric center of all specified geometric regions altogether as the reference point. The reference location has six degrees of freedom.

For better performance, specify geometric regions with a minimal number of nodes. For example, use a set of edges instead of using a face, and a set of vertices instead of using an edge.

`structuralBC(___,"Reference",Coords)` specifies the reference point for the multipoint constraint instead of using the geometric center of all specified regions as a reference point.

Use this syntax with the input arguments from the previous syntax.

`structuralBC(___,"Reference",Coords,"Radius",R)` restricts the region for multipoint constraint to nodes within the circle (for a 2-D geometry) or sphere (for a 3-D geometry) of radius `R` around the reference point.

Sparse Linear Models for Use with Control System Toolbox

`structuralBC(___,"Label",labeltext)` adds a label for the structural boundary condition to be used by the `linearizeInput` function. This function lets you pass boundary conditions to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

Vectorized Evaluation for Function Handles

`structuralBC(___,"Vectorized","on")` uses vectorized function evaluation when you pass a function handle as an argument. If your function handle computes in a vectorized fashion, then using this argument saves time. See “Vectorization”. For details on this evaluation, see “Nonconstant Boundary Conditions” on page 2-132.

Use this syntax with any of the input arguments from previous syntaxes.

Structural Boundary Condition Object

`bc = structuralBC(___)` returns the structural boundary condition object using any of the input arguments from previous syntaxes.

Examples

Apply Fixed Boundaries and Specify Surface Traction

Apply fixed boundaries and traction on two ends of a bimetallic cable.

Create a structural model.

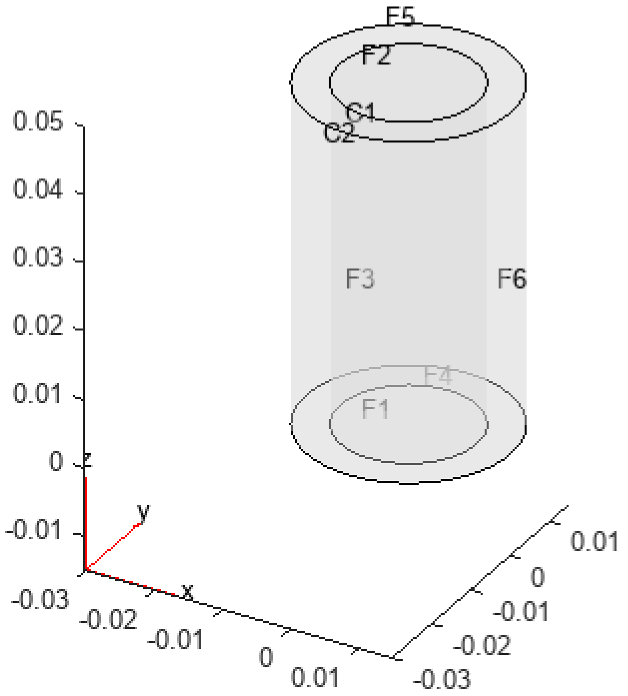
```
structuralModel = createpde("structural","static-solid");
```

Create nested cylinders to model a bimetallic cable.

```
gm = multicylinder([0.01,0.015],0.05);
```

Assign the geometry to the structural model and plot the geometry.

```
structuralModel.Geometry = gm;
pdegplot(structuralModel,"CellLabels","on", ...
                         "FaceLabels","on", ...
                         "FaceAlpha",0.4)
```



For each metal, specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralModel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28);
structuralProperties(structuralModel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralModel,"Face",[1,4],"Constraint","fixed")
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: [1 4]
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: "fixed"
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
           SurfaceTraction: []
                  Pressure: []
    TranslationalStiffness: []
                     Label: []
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralModel, ...
                          "Face",[2,5], ...
                          "SurfaceTraction",[0;0;100])
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: [2 5]
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []
```

```
   Boundary Loads
                    Force: []
          SurfaceTraction: [3x1 double]
                 Pressure: []
    TranslationalStiffness: []
                    Label: []
```

Specify Displacements

Create a structural model.

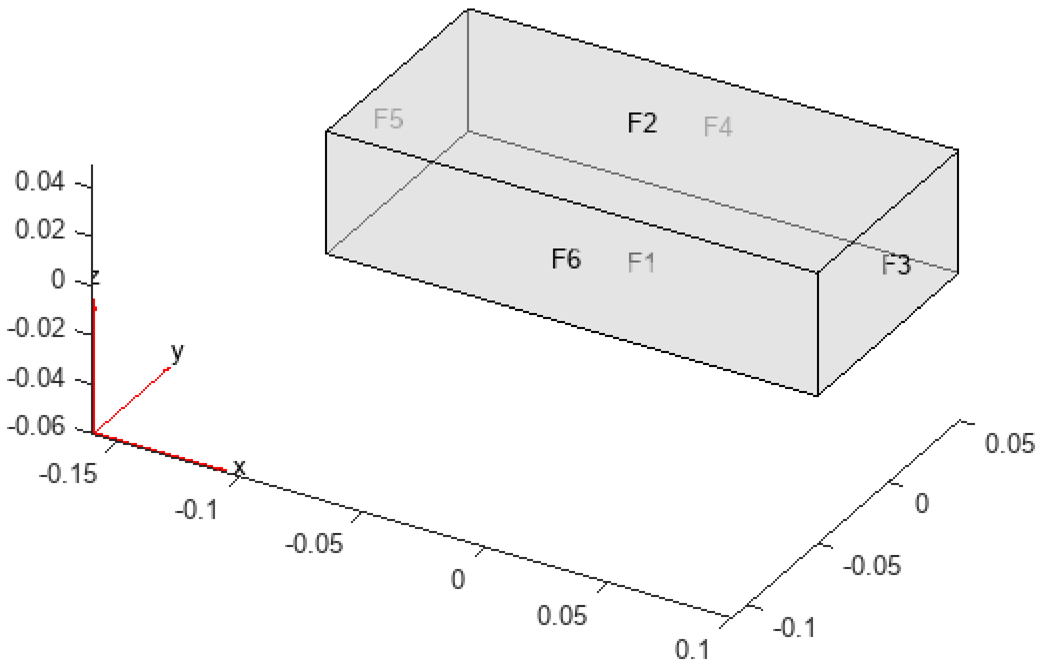
```
structuralModel = createpde("structural","static-solid");
```

Create a block geometry.

```
gm = multicuboid(0.2,0.1,0.05);
```

Assign the geometry to the structural model and plot the geometry.

```
structuralModel.Geometry = gm;
pdegplot(structuralModel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(structuralModel,"YoungsModulus",74e9,...
                                     "PoissonsRatio",0.42,...
                                     "MassDensity",19.29e3);
```

Specify the gravity load on the beam.

```
structuralBodyLoad(structuralModel, ...
    "GravitationalAcceleration",[0;0;-9.8]);
```

Specify that face 5 is a fixed boundary.

```
structuralBC(structuralModel,"Face",5,"Constraint","fixed");
```

Specify *z*-displacement on face 3 of the model. By leaving the *x*- and *y*-displacements unspecified, you enable face 3 to move in the *x*- and *y*-directions.

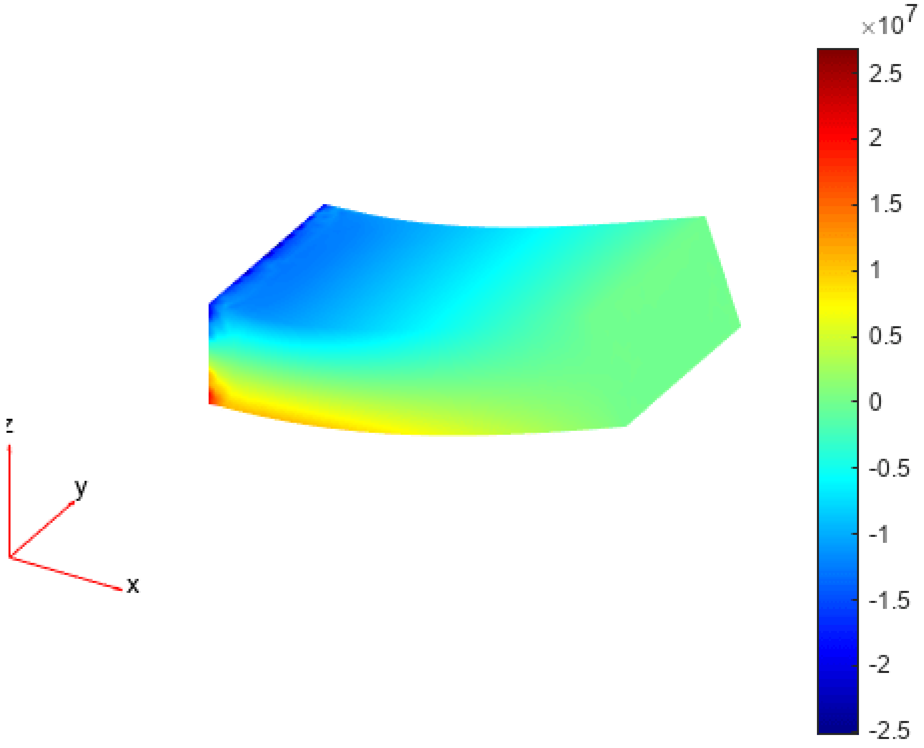
```
structuralBC(structuralModel,"Face",3,"ZDisplacement",0.0001);
```

Generate a mesh and solve the model.

```
generateMesh(structuralModel);
R = solve(structuralModel);
```

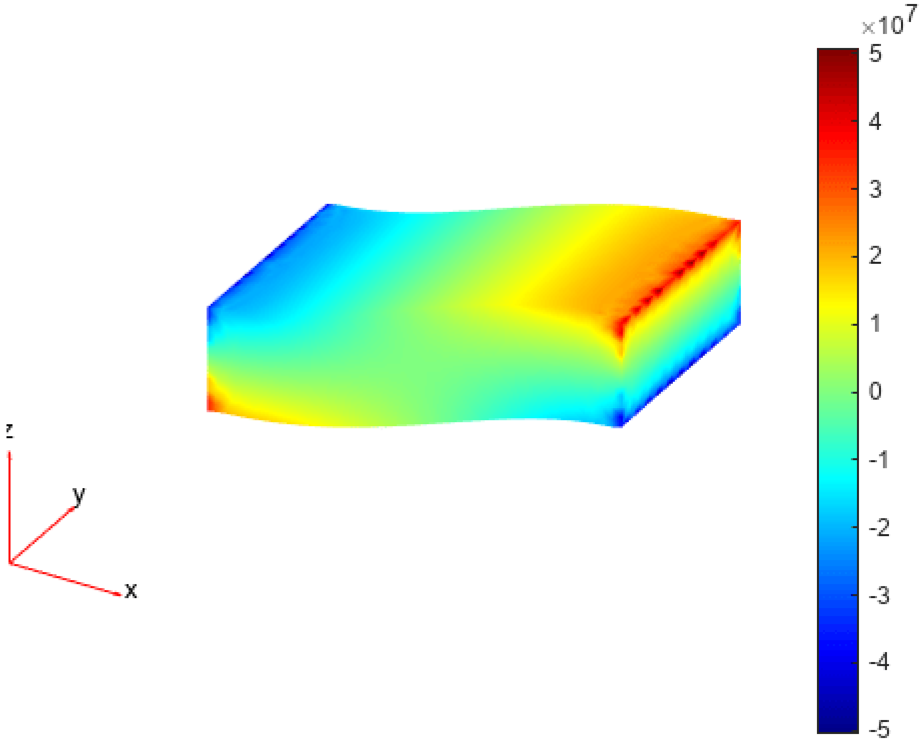
Plot the deformed shape with the *x*-component of normal stress.

```
pdeplot3D(structuralModel,"ColorMapData",R.Stress.sxx, ...
                          "Deformation",R.Displacement)
```



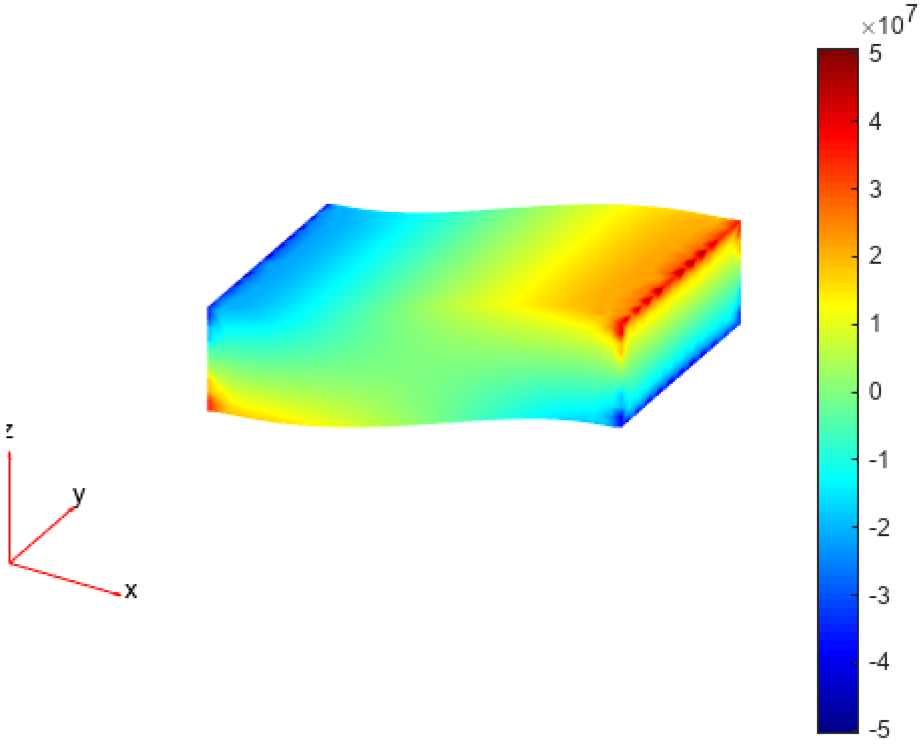
Now specify all three displacements on the same face. Here, the *z*-displacement is the same, but the *x*- and *y*-displacements are both zero. Face 3 cannot move in the *x*- and *y*-directions.

```
structuralBC(structuralModel,"Face",3, ...
                             "Displacement",[0;0;0.0001]);
R = solve(structuralModel);
pdeplot3D(structuralModel,"ColorMapData",R.Stress.sxx, ...
                          "Deformation",R.Displacement)
```



Thus, specifying `"Displacement",[0;0;0.0001]` is equivalent to specifying `"XDisplacement",0,"YDisplacement",0,"ZDisplacement",0.0001`.

```
structuralBC(structuralModel,"Face",3,"XDisplacement",0, ...
                                      "YDisplacement",0, ...
                                      "ZDisplacement",0.0001);
R = solve(structuralModel);
pdeplot3D(structuralModel,"ColorMapData",R.Stress.sxx, ...
                          "Deformation",R.Displacement)
```



Static Analysis of Spinning Disk with Press-Fit at Hub

Analyze a spinning disk with radial compression at the hub due to press-fit. The inner radius of the disk is 0.05, and the outer radius is 0.2. The thickness of the disk is 0.05 with an interference fit of 50E-6. For this analysis, simplify the 3-D axisymmetric model to a 2-D model.

Create a static structural analysis model for solving an axisymmetric problem.

```
structuralmodel = createpde("structural","static-axisymmetric");
```

The 2-D model is a rectangular strip whose *x*-dimension extends from the hub to the outer surface, and whose *y*-dimension extends over the height of the disk. Create the geometry by specifying the coordinates of the strip's four corners. For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through $r = 0$, which is equivalent to $x = 0$.

```
g = decsg([3 4 0.05 0.2 0.2 0.05 -0.025 -0.025 0.025 0.025]');
```

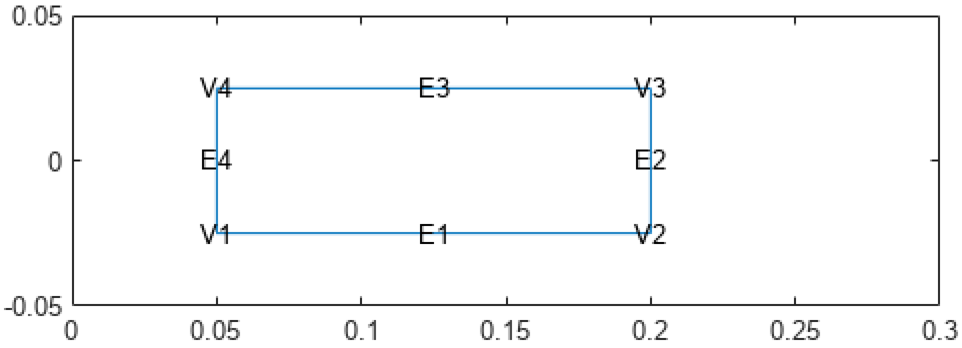
Include the geometry in the model.

```
geometryFromEdges(structuralmodel,g);
```

Plot the geometry with the edge and vertex labels.

```
figure
pdegplot(structuralmodel,"EdgeLabels","on","VertexLabels","on")
```

```
xlim([0 0.3])
ylim([-0.05 0.05])
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(structuralmodel,"YoungsModulus",210e9, ...
                                     "PoissonsRatio",0.28, ...
                                     "MassDensity",7700);
```

Apply centrifugal load due to spinning of the disk. Assume that the disk is spinning at 104.7 rad/s.

```
structuralBodyLoad(structuralmodel,"AngularVelocity",1047);
```

Apply radial displacement at the hub of the disk to model press-fit.

```
structuralBC(structuralmodel,"Edge",4,"RDisplacement",50e-6);
```

Fix axial displacement of a point on the hub to prevent rigid body motion.

```
structuralBC(structuralmodel,"Vertex",1,"ZDisplacement",0);
```

Generate a mesh.

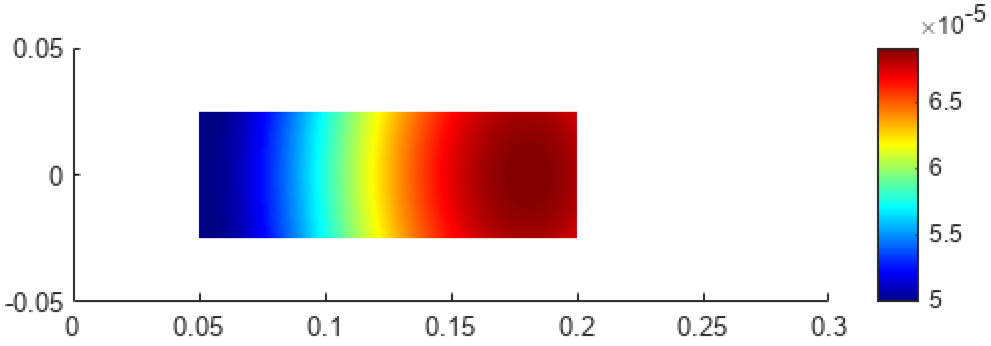
```
generateMesh(structuralmodel);
```

Solve the model.

```
structuralresults = solve(structuralmodel);
```

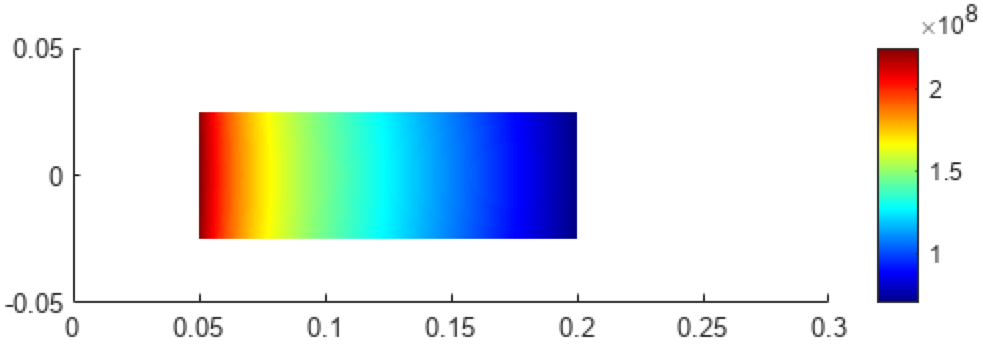
Plot the radial displacement of the disk.

```
figure
pdeplot(structuralmodel, ...
        "XYData",structuralresults.Displacement.ur, ...
        "ColorMap","jet")
axis equal
xlim([0 0.3])
ylim([-0.05 0.05])
```



Plot circumferential (hoop) stress.

```
figure
pdeplot(structuralmodel, ...
        "XYData",structuralresults.Stress.sh, ...
        "ColorMap","jet")
axis equal
xlim([0 0.3])
ylim([-0.05 0.05])
```



Specify Nonconstant Displacement by Using Function Handle

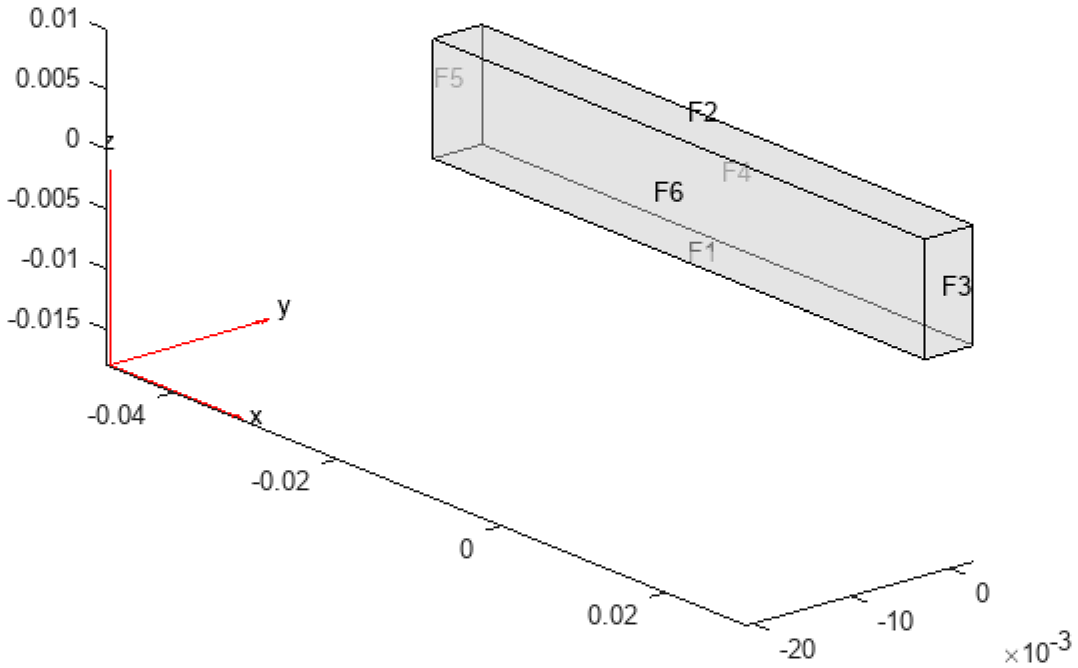
Use a function handle to specify a harmonically varying excitation in a beam.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite to the fixed end of the beam.

```
yDisplacementFunc = ...
@(location,state) ones(size(location.y))*1E-4*sin(50*state.time);
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",yDisplacementFunc);
```

Apply Sinusoidal Displacement by Specifying Frequency

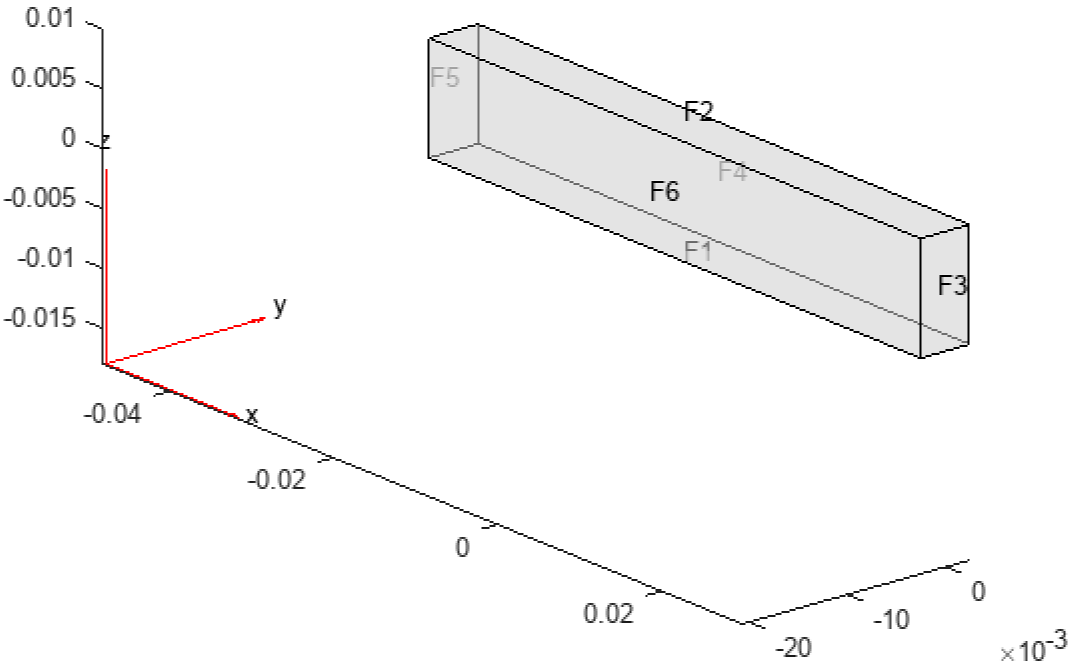
Specify a harmonically varying excitation by specifying its frequency.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Fix one end of the beam.

```
structuralBC(structuralmodel,"Face",5,"Constraint","fixed");
```

Apply a sinusoidal displacement along the *y*-direction on the end opposite to the fixed end of the beam.

```
structuralBC(structuralmodel,"Face",3, ...
                             "YDisplacement",1E-4, ...
                             "Frequency",50);
```

Specify Displacement at Corner of Rectangle

Fix one corner of a rectangular plate to restrain all rigid body motions of the model.

Create a structural model for static plane-stress analysis.

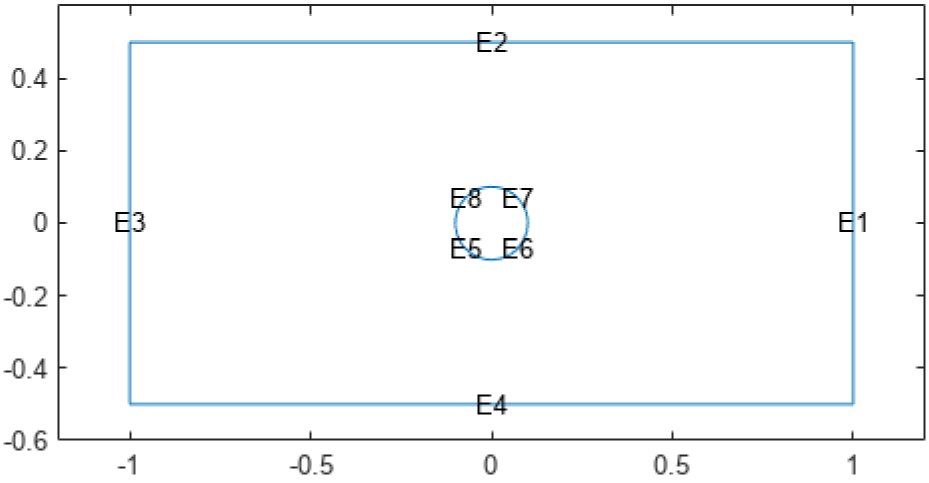
```
model = createpde("structural","static-planestress");
```

Create the geometry and include it in the structural model.

```
length = 1;
width = 0.5;
radius = 0.1;
R1 = [3 4 -length length length -length ...
          -width -width  width   width]';
C1 = [1 0 0 radius 0 0 0 0 0 0]';
gdm = [R1 C1];
ns = char('R1','C1');
g = decsg(gdm,'R1- C1',ns');
geometryFromEdges(model,g);
```

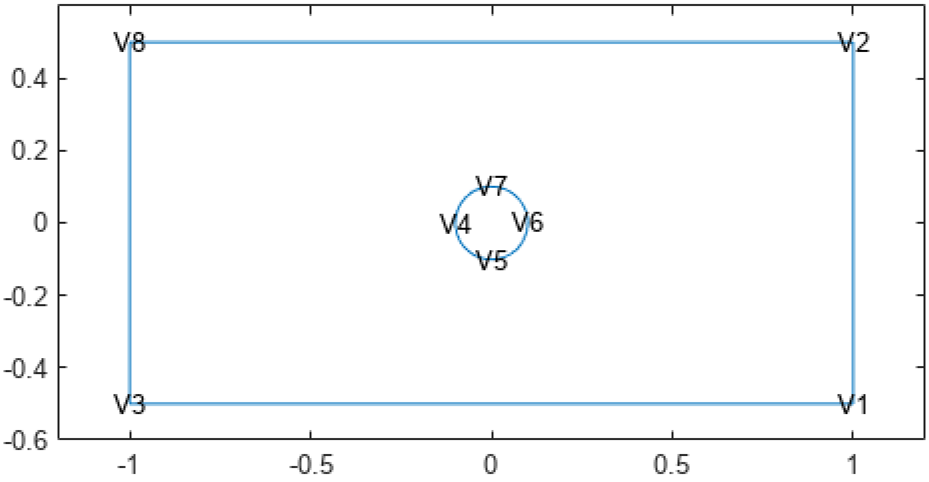
Plot the geometry, displaying edge labels.

```
figure
pdegplot(model,"EdgeLabels","on");
axis([-1.2*length 1.2*length ...
      -1.2*width  1.2*width])
```



Plot the geometry, displaying vertex labels.

```
figure
pdegplot(model,"VertexLabels","on");
axis([-1.2*length 1.2*length ...
      -1.2*width  1.2*width])
```



Specify Young's modulus and Poisson's ratio of the material.

```
structuralProperties(model,"YoungsModulus",210E9,"PoissonsRatio",0.3);
```

Set the *x*-component of displacement on the left edge of the plate to zero to resist the applied load.

```
structuralBC(model,"Edge",3,"XDisplacement",0);
```

Apply the surface traction with a nonzero *x*-component on the right edge of the plate.

```
structuralBoundaryLoad(model,"Edge",1,"SurfaceTraction",[100000 0]);
```

Set the *y*-component of displacement at the bottom-left corner (vertex 3) to zero to restraint the rigid body motion.

```
structuralBC(model,"Vertex",3,"YDisplacement",0);
```

Generate the mesh, using `Hmax` to control the mesh size. A fine mesh lets you capture the gradation in the solution accurately.

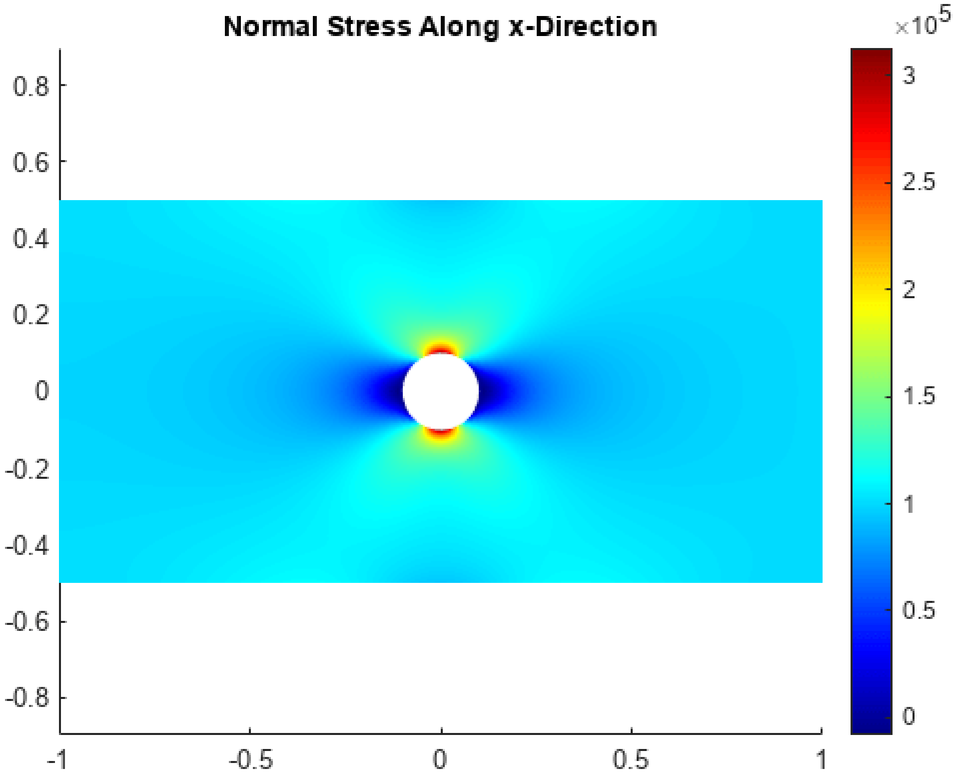
```
generateMesh(model,"Hmax",radius/6);
```

Solve the problem.

```
R = solve(model);
```

Plot the x-component of the normal stress distribution.

```
pdeplot(model,"XYData",R.Stress.sxx);
axis equal
colormap jet
title("Normal Stress Along x-Direction")
```



Set Multipoint Constraint and Obtain ROM Results Compatible with Simscape Multibody™

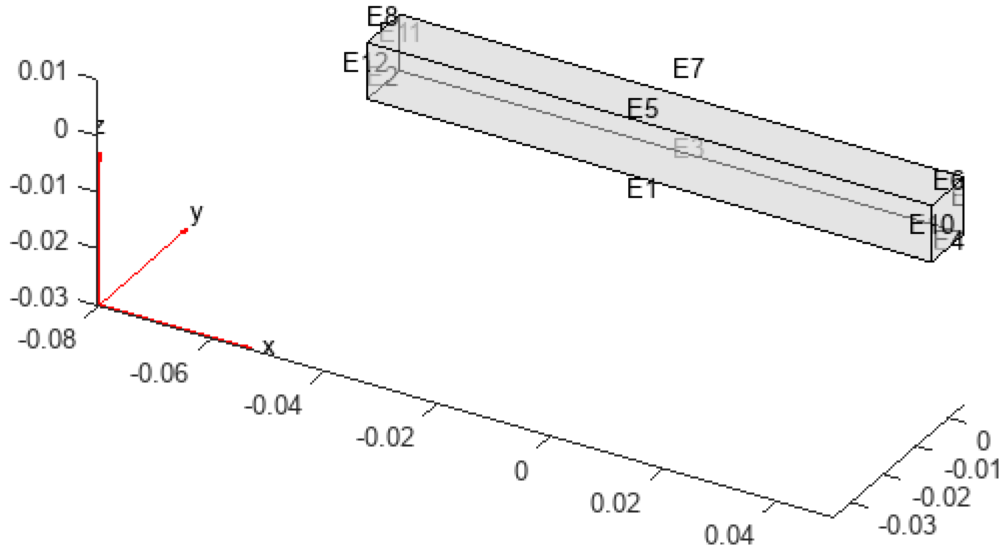
Set multipoint constraints on two opposite sides of a beam.

Create a transient structural model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.1,0.01,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"EdgeLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",70E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",2700);
```

Generate a mesh.

```
generateMesh(structuralmodel);
```

Set the multipoint constraint on the right side of the beam. For better performance, set the constraint on the set of edges bounding the right side of the beam instead of setting it on the entire face.

```
structuralBC(structuralmodel,"Edge",[4,6,9,10], ...
                             "Constraint","multipoint")
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Edge'
                  RegionID: [4 6 9 10]
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
```

```
             Constraint: "multipoint"
                 Radius: []
              Reference: []
                  Label: []

   Boundary Loads
                  Force: []
         SurfaceTraction: []
               Pressure: []
  TranslationalStiffness: []
                  Label: []

   Time Variation of Force, Pressure, or Enforced Displacement
              StartTime: []
                EndTime: []
               RiseTime: []
               FallTime: []

   Sinusoidal Variation of Force, Pressure, or Enforced Displacement
              Frequency: []
                  Phase: []
```

Using the same approach, set the multipoint constraint on the left side of the beam.

```
structuralBC(structuralmodel,"Edge",[2,8,11,12], ...
                             "Constraint","multipoint")
```

```
ans = 
  StructuralBC with properties:

             RegionType: 'Edge'
               RegionID: [2 8 11 12]
             Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
           Displacement: []
          XDisplacement: []
          YDisplacement: []
          ZDisplacement: []
             Constraint: "multipoint"
                 Radius: []
              Reference: []
                  Label: []

   Boundary Loads
                  Force: []
         SurfaceTraction: []
               Pressure: []
  TranslationalStiffness: []
                  Label: []

   Time Variation of Force, Pressure, or Enforced Displacement
              StartTime: []
                EndTime: []
               RiseTime: []
               FallTime: []
```

```
   Sinusoidal Variation of Force, Pressure, or Enforced Displacement
                  Frequency: []
                       Phase: []
```

Reduce the model to all modes in the frequency range `[-Inf,500000]` and the interface degrees of freedom.

```
R = reduce(structuralmodel,"FrequencyRange",[-Inf,500000]);
```

Input Arguments

`structuralmodel` — Structural model

`StructuralModel` object

Structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`RegionType` — Geometric region type

`"Vertex"` | `"Edge"` | `"Face"` (for a 3-D model only)

Geometric region type, specified as `"Vertex"`, `"Edge"`, or, for a 3-D model, `"Face"`.

You cannot use the following geometric region types if you specify the `"roller"` or `"symmetric"` value for the boundary constraint `Cval`:

- `"Edge"` for a 3-D model
- `"Vertex"` for a 2-D or 3-D model

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.1)`

Data Types: `char` | `string`

`RegionID` — Geometric region ID

vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01)`

Data Types: `double`

Displacement

`Dval` — Enforced displacement

numeric vector | function handle

Enforced displacement, specified as a numeric vector or function handle. A numeric vector must contain two elements for a 2-D model (including axisymmetric models) and three elements for a 3-D model. The function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix must correspond to an enforced displacement vector at the boundary coordinates provided by the solver. In case of a transient or frequency response analysis, `Dval` also can be a function of time or frequency, respectively. For details, see “More About” on page 5-1251.

Note that when you specify `Dval` for an axisymmetric model, the radial displacement on the axis of rotation must always be zero.

Example: `structuralBC(structuralmodel,"Face",[2,5],"Displacement",[0;0;0.01])`

Data Types: `double` | `function_handle`

`XDval` — x-component of enforced displacement

number | function handle

x-component of enforced displacement, specified as a number or function handle. The function must return a row vector. Each element of this vector corresponds to the x-component value of the enforced displacement at the boundary coordinates provided by the solver. In case of a transient or frequency response analysis, `XDval` also can be a function of time or frequency, respectively. For details, see “More About” on page 5-1251.

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01)`

Data Types: `double` | `function_handle`

`YDval` — y-component of enforced displacement

number | function handle

y-component of enforced displacement, specified as a number or function handle. The function must return a row vector. Each element of this vector corresponds to the y-component value of the enforced displacement at the boundary coordinates provided by the solver. In case of a transient or frequency response analysis, `YDval` also can be a function of time or frequency, respectively. For details, see “More About” on page 5-1251.

Example: `structuralBC(structuralmodel,"Face",[2,5],"YDisplacement",0.01)`

Data Types: `double` | `function_handle`

`ZDval` — z-component of enforced displacement

number | function handle

z-component of enforced displacement, specified as a number or function handle. The function must return a row vector. Each element of this vector corresponds to the z-component value of the enforced displacement at the boundary coordinates provided by the solver. For a transient or frequency response analysis, `ZDval` also can be a function of time or frequency, respectively. For details, see “More About” on page 5-1251.

You can specify `ZDval` for a 3-D or axisymmetric model.

Example: `structuralBC(structuralmodel,"Face",[2,5],"ZDisplacement",0.01)`

Data Types: `double` | `function_handle`

`RDval` — r-component of enforced displacement

number | function handle

r-component of enforced displacement, specified as a number or function handle. The function must return a row vector. Each element of this vector corresponds to the r-component value of the enforced displacement at the boundary coordinates provided by the solver. For a transient or frequency response analysis, `RDval` also can be a function of time or frequency, respectively. For details, see “More About” on page 5-1251.

You can specify `RDval` only for an axisymmetric model. `RDval` must be zero on the axis of rotation.

Example: `structuralBC(structuralmodel,"Face",[2,5],"RDisplacement",0.01)`

Data Types: `double` | `function_handle`

`Cval` — Standard structural boundary constraints

`"free"` (default) | `"fixed"` | `"roller"` | `"symmetric"` | `"multipoint"`

Standard structural boundary constraints, specified as `"free"`,`"fixed"`,`"roller"`, `"symmetric"`, or `"multipoint"`.

You cannot use the `"roller"` and `"symmetric"` values with the following geometric region types:

- `"Edge"` for a 3-D model
- `"Vertex"` for a 2-D or 3-D model

Example: `structuralBC(structuralmodel,"Face",[2,5],"Constraint","fixed")`

Data Types: `char` | `string`

`Coords` — Reference point location for multipoint constraint

2-by-1 numeric vector | 3-by-1 numeric vector

Reference point location for the multipoint constraint, specified as a 2-by-1 (for a 2-D geometry) or 3-by-1 (for a 3-D geometry) numeric vector.

Example: `structuralBC(structuralmodel,"Vertex",[1,3,5:10],"Constraint","multipoint","Reference",[0;0;1])`

Data Types: `double`

`R` — Radius of circle (for 2-D geometry) or sphere (for 3-D geometry) around reference point location for multipoint constraint

positive number

Radius of a circle (for a 2-D geometry) or a sphere (for a 3-D geometry) around the reference point location for the multipoint constraint, specified as a positive number.

Example: `structuralBC(structuralmodel,"Vertex",[1,3,5:10],"Constraint","multipoint","Reference",[0;0;1],"Radius",0.2)`

Data Types: `double`

`labeltext` — Label for structural boundary condition

character vector | string

Label for the structural boundary condition, specified as a character vector or a string.

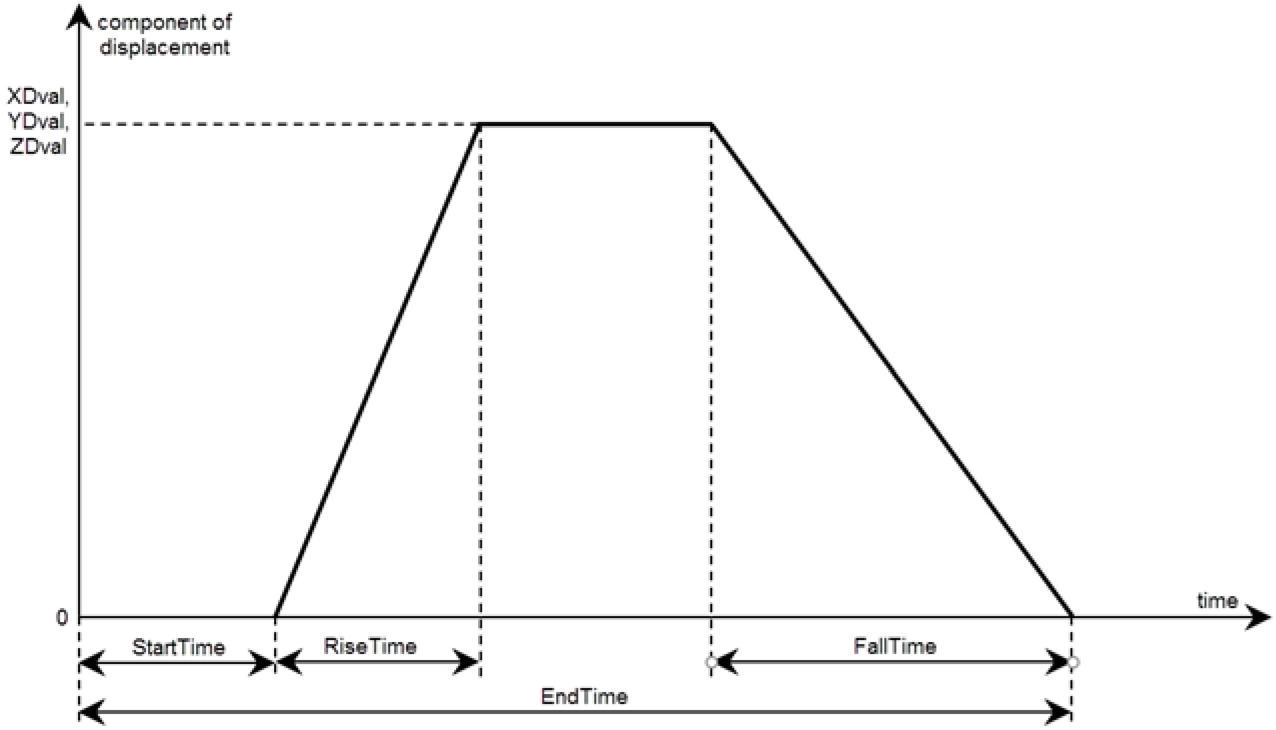
Data Types: `char` | `string`

Name-Value Pair Arguments

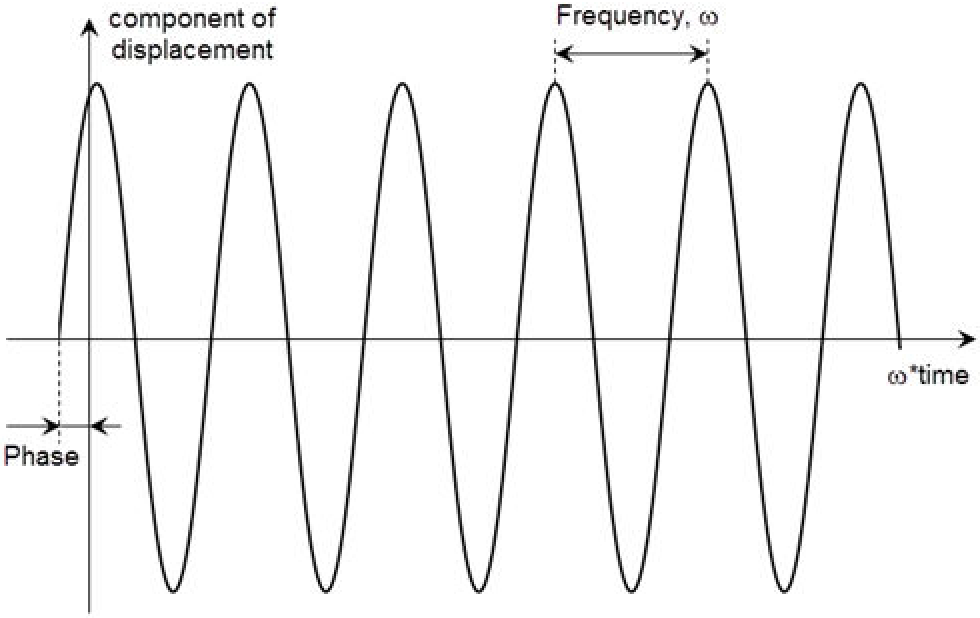
Use one or more name-value pair arguments to specify the form and duration of the time-varying value of a component of displacement. Specify the displacement value using one of the following arguments: `XDval`, `YDval`, `ZDval`, or `RDval`. You cannot use these name-value pair arguments to specify more than one time-varying component or to specify the `Dval` value.

You can model rectangular, triangular, and trapezoidal displacement pulses. If the start time is 0, you do not need to specify it.

- For a rectangular pulse, specify the start and end times.
- For a triangular pulse, specify the start time and any two of the following times: rise time, fall time, and end time. You also can specify all three times, but they must be consistent.
- For a trapezoidal pulse, specify all four times.



You can model a harmonic displacement by specifying its frequency and initial phase. If the initial phase is 0, you do not need to specify it.



Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01,"RiseTime",0.5,"FallTime",0.5,"EndTime",3)`

Rectangular, Triangular, or Trapezoidal Pulse

`StartTime` — Start time for displacement component

0 (default) | positive number

Start time for the displacement component, specified as 0 or a positive number. Specify this argument only for transient structural models.

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01,"StartTime",1,"EndTime",3)`

Data Types: `double`

`EndTime` — End time for displacement component

positive number

End time for the displacement component, specified as a positive number equal or greater than the start time value. Specify this argument only for transient structural models.

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01,"StartTime",1,"EndTime",3)`

Data Types: `double`

`RiseTime` — Rise time for displacement component

positive number

Rise time for the displacement component, specified as a positive number. Specify this argument only for transient structural models.

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01,"RiseTime",0.5,"FallTime",0.5,"EndTime",3)`

Data Types: `double`

`FallTime` — Fall time for displacement component
positive number

Fall time for the displacement component, specified as a positive number. Specify this argument only for transient structural models.

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01,"RiseTime",0.5,"FallTime",0.5,"EndTime",3)`

Data Types: `double`

Harmonic Displacement

`Frequency` — Frequency of sinusoidal displacement component
positive number

Frequency of a sinusoidal displacement component value, specified as a positive number in radians per unit of time. Specify this argument only for transient structural models.

Example: `structuralBC(structuralmodel,"Face","XDisplacement",0.01,"Frequency",25)`

Data Types: `double`

`Phase` — Frequency of sinusoidal displacement component
0 (default) | positive number

Phase of a sinusoidal displacement component value, specified as a positive number in radians. Specify this argument only for transient structural models.

Example: `structuralBC(structuralmodel,"Face",[2,5],"XDisplacement",0.01,"Frequency",25,"Phase",pi/6)`

Data Types: `double`

Output Arguments

`bc` — Handle to boundary condition
`StructuralBC` object

Handle to the boundary condition, returned as a `StructuralBC` object. See StructuralBC Properties.

More About

Degrees of Freedom (DoFs)

In Partial Differential Equation Toolbox, each node of a 2-D or 3-D geometry has two or three degrees of freedom (DoFs), respectively. DoFs correspond to translational displacements. If the number of mesh points in a model is `NumNodes`, then the toolbox assigns the IDs to the degrees of freedom as follows:

- Numbers from 1 to `NumNodes` correspond to an `x`-displacement at each node.
- Numbers from `NumNodes+1` to `2*NumNodes` correspond to a `y`-displacement at each node.

- Numbers from `2*NumNodes+1` to `3*NumNodes` correspond to a `z`-displacement at each node of a 3-D geometry.

Specifying Nonconstant Parameters of a Structural Model

Use a function handle to specify the following structural parameters when they depend on space and, depending of the type of structural analysis, either time or frequency:

- Surface traction on the boundary
- Pressure normal to the boundary
- Concentrated force at a vertex
- Distributed spring stiffness for each translational direction used to model elastic foundation
- Enforced displacement and its components
- Initial displacement and velocity (can depend on space only)

For example, use function handles to specify the pressure load, *x*-component of the enforced displacement, and the initial displacement for this model.

```
structuralBoundaryLoad(model,"Face",12, ...
                             "Pressure",@myfunPressure)
structuralBC(model,"Face",2, ...
                   "XDisplacement",@myfunBC)
structuralIC(model,"Face",12, ...
                   "Displacement",@myfunIC)
```

For all parameters, except the initial displacement and velocity, the function must be of the form:

```
function structuralVal = myfun(location,state)
```

For the initial displacement and velocity the function must be of the form:

```
function structuralVal = myfun(location)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments (or one argument if the function specifies the initial displacement or initial velocity).

- `location` — A structure containing these fields:
  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes these data in the `location` structure:

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, *r*-component of the normal vector at the evaluation point or points

- `state` — A structure containing these fields for dynamic structural problems:
  - `state.time` contains the time at evaluation points.
  - `state.frequency` contains the frequency at evaluation points.

  `state.time` and `state.frequency` are scalars.

Boundary constraints and loads get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`
- `state.time` or `state.frequency` (depending of the type of structural analysis)

Initial conditions get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

If a parameter represents a vector value, such as surface traction, spring stiffness, force, or displacement, your function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix corresponds to the parameter value (a vector) at the boundary coordinates provided by the solver.

If a parameter represents a scalar value, such as pressure or a displacement component, your function must return a row vector where each element corresponds to the parameter value (a scalar) at the boundary coordinates provided by the solver.

If boundary conditions depend on `state.time` or `state.frequency`, ensure that your function returns a matrix of `NaN` of the correct size when `state.frequency` 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.

Additional Arguments in Functions for Nonconstant Structural Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
structuralVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
structuralBC(model,"Face",2,"XDisplacement",structuralVal)

structuralVal = ...
@(location) myfunWithAdditionalArgs(location,arg1,arg2...)
structuralIC(model,"Face",2,"Displacement",structuralVal)
```

Tips

- Restrain all rigid body motions by specifying as many boundary conditions as needed. If you do not restrain all rigid body motions, the entire geometry can freely rotate or move. The resulting linear system of equations is singular. The system can take a long time to converge, or it might not converge at all. If the system converges, the solution includes a large rigid body motion in addition to deformation.

Version History

Introduced in R2017b

Label to extract sparse linear models for use with Control System Toolbox

Now you can add a label for structural boundary conditions to be used by the `linearizeInput` function. This function lets you pass structural boundary conditions to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

Axisymmetric analysis

You can now specify an enforced displacement for a structural axisymmetric model.

Concentrated boundary constraints at arbitrary locations on geometry surfaces

You can now use `addVertex` to create new vertices at any points on boundaries of a 2-D or 3-D geometry represented by a `DiscreteGeometry` object. Then set concentrated boundary constraints at these vertices.

Multipoint Constraint

You can now set a multipoint constraint, which ensures that all nodes and all degrees of freedom have a rigid constraint with the geometric center of all specified geometric regions together as the reference point. The reference location has six degrees of freedom.

Boundary constraints on edges and vertices

You can now specify an enforced displacement and the `'free'` and `'fixed'` boundary constraints on vertices and edges for both 2-D and 3-D models. The previous versions of this function let you specify the displacement and boundary constraints values only on edges for 2-D models and faces on 3-D models.

Time-dependent boundary conditions

You can now specify time-dependent boundary conditions by using function handles or specify the form and duration of *x*-, *y*-, or *z*-component of the enforced displacement and the frequency and phase of sinusoidal displacement.

See Also

`StructuralModel` | `structuralProperties` | `structuralDamping` | `structuralBodyLoad` | `structuralBoundaryLoad` | `structuralSEInterface` | `reduce` | `solve` | `reconstructSolution`

StructuralBC Properties

Boundary condition or boundary load for structural analysis model

Description

A `StructuralBC` object specifies the type of PDE boundary condition or boundary load on a set of geometry boundaries. A `StructuralModel` object contains a vector of `StructuralBC` objects in its `BoundaryConditions.StructuralBCAssignments` property.

To specify boundary conditions for your model, use the `structuralBC` function. To specify boundary loads, use `structuralBoundaryLoad`.

Properties

Properties of StructuralBC

`RegionType` — Geometric region type
`'Face'` for 3-D geometry | `'Edge'` for 2-D geometry

Geometric region type, specified as `'Face'` for a 3-D geometry or `'Edge'` for a 2-D geometry.

Data Types: `char` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot` with `'FaceLabels'` (3-D) or `'EdgeLabels'` (2-D) set to `'on'`.

Data Types: `double`

`Vectorized` — Vectorized function evaluation
`'off'` (default) | `'on'`

Vectorized function evaluation, specified as `'off'` or `'on'`. This evaluation applies when you pass a function handle as an argument. To save time in the function handle evaluation, specify `'on'`, assuming that your function handle computes in a vectorized fashion. See “Vectorization”. For details on this evaluation, see “Nonconstant Boundary Conditions” on page 2-132.

Data Types: `char` | `string`

Boundary Constraints and Enforced Displacements

`Displacement` — Enforced displacement
numeric vector | function handle

Enforced displacement, specified as a numeric vector or function handle. The numeric vector must contain two elements for a 2-D model and three elements for a 3-D model. The function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix must correspond to the enforced displacement vector at the boundary coordinates provided by the solver.

Data Types: `double` | `function_handle`

`XDisplacement` — x-component of enforced displacement
number | function handle

x-component of the enforced displacement, specified as a number or function handle. The function must return a row vector. Each column of the vector must correspond to the value of the x-component of the enforced displacement at the boundary coordinates provided by the solver.

For axisymmetric models, this property contains the radial component (r-component) of the enforced displacement.

Data Types: `double` | `function_handle`

`YDisplacement` — y-component of enforced displacement
number | function handle

y-component of the enforced displacement, specified as a number or function handle. The function must return a row vector. Each column of the vector must correspond to the value of the y-component of the enforced displacement at the boundary coordinates provided by the solver.

For axisymmetric models, this property contains the axial component (z-component) of the enforced displacement.

Data Types: `double` | `function_handle`

`ZDisplacement` — z-component of enforced displacement
number | function handle

z-component of the enforced displacement, specified as a number or function handle. The function must return a row vector. Each column of the vector must correspond to the value of the z-component of the enforced displacement at the boundary coordinates provided by the solver.

Data Types: `double` | `function_handle`

`Constraint` — Standard structural boundary constraints
`'free'` | `'fixed'` | `'roller'` | `'symmetric'` | `'multipoint'`

Standard structural boundary constraints, specified as `'free'`,`'fixed'`,`'roller'`, `'symmetric'`, or `'multipoint'`.

Data Types: `char`

`Radius` — Radius of circle (for 2-D geometry) or sphere (for 3-D geometry) around reference point location for multipoint constraint
positive number

Radius of a circle (for a 2-D geometry) or a sphere (for a 3-D geometry) around the reference point location for the multipoint constraint, specified as a positive number.

Data Types: `double`

`Reference` — Reference point location for multipoint constraint
2-by-1 numeric vector | 3-by-1 numeric vector

Reference point location for the multipoint constraint, specified as a 2-by-1 (for a 2-D geometry) or 3-by-1 (for a 3-D geometry) numeric vector.

Data Types: `double`

Boundary Loads

`Force` — Concentrated force

numeric vector | function handle

Concentrated force at a vertex, specified as a numeric vector or function handle.

Data Types: `double` | `function_handle`

`SurfaceTraction` — Normal and tangential distributed forces on boundary

numeric vector | function handle

Normal and tangential distributed forces on the boundary (in the global Cartesian coordinates system), specified as a numeric vector or function handle. The numeric vector must contain two elements for a 2-D model and three elements for a 3-D model. The function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix must correspond to the surface traction vector at the boundary coordinates provided by the solver.

Data Types: `double` | `function_handle`

`Pressure` — Pressure normal to boundary

number | function handle

Pressure normal to the boundary, specified as a number or function handle. The function must return a row vector in which each column corresponds to the value of pressure at the boundary coordinates provided by the solver. A positive value of pressure acts in the direction of the outward normal to the boundary.

Data Types: `double` | `function_handle`

`TranslationalStiffness` — Distributed spring stiffness

numeric vector | function handle

Distributed spring stiffness for each translational direction used to model an elastic foundation, specified as a numeric vector or function handle. The numeric vector must contain two elements for a 2-D model and three elements for a 3-D model. The custom function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of this matrix corresponds to the stiffness vector at the boundary coordinates provided by the solver.

Data Types: `double` | `function_handle`

Time Variation of Force, Pressure, or Enforced Displacement

`StartTime` — Start time for displacement component, pressure, or concentrated force load

nonnegative number

Start time for a displacement component, the pressure, or the concentrated force load, specified as a nonnegative number.

Data Types: `double`

`EndTime` — End time for displacement component, pressure, or concentrated force load

nonnegative number

End time for a displacement component, the pressure, or the concentrated force load, specified as a nonnegative number.

Data Types: `double`

`RiseTime` — Rise time for displacement component, pressure, or concentrated force load
nonnegative number

Rise time for a displacement component, the pressure, or the concentrated force load, specified as a nonnegative number.

Data Types: `double`

`FallTime` — Fall time for displacement component, pressure, or concentrated force load
nonnegative number

Fall time for a displacement component, the pressure, or the concentrated force load, specified as a nonnegative number.

Data Types: `double`

Sinusoidal Variation of Force, Pressure, or Enforced Displacement

`Frequency` — Frequency of sinusoidal displacement component, sinusoidal pressure, or concentrated force
positive number

Frequency of a sinusoidal displacement component, the sinusoidal pressure, or the concentrated force, specified as a positive number, in radians per unit of time.

Data Types: `double`

`Phase` — Phase of sinusoidal displacement component, sinusoidal pressure, or concentrated force
nonnegative number

Phase of a sinusoidal displacement component, the sinusoidal pressure, or the concentrated force, specified as a nonnegative number, in radians per unit of time.

Data Types: `double`

`Label` — Label for use with `linearizeInput`
character vector | string

Label for use with `linearizeInput`, specified as a character vector or a string.

Data Types: `char` | `string`

Version History

Introduced in R2017b

See Also

`findStructuralBC` | `structuralBC` | `structuralBoundaryLoad` | `structuralSEInterface` | StructuralSEIAssignment Properties

structuralIC

Package: `pde`

Set initial conditions for a transient structural model

Syntax

```
structuralIC(structuralmodel,"Displacement",u0,"Velocity",v0)
structuralIC( ___ RegionType,RegionID)
structuralIC(structuralmodel,Sresults)
structuralIC(structuralmodel,Sresults,iT)
struct_ic = structuralIC( ___ )
```

Description

`structuralIC(structuralmodel,"Displacement",u0,"Velocity",v0)` sets initial displacement and velocity for the entire geometry.

`structuralIC( ___ RegionType,RegionID)` sets initial displacement and velocity for a particular geometry region using the arguments from the previous syntax.

`structuralIC(structuralmodel,Sresults)` sets initial displacement and velocity using the solution `Sresults` from a previous structural analysis on the same geometry. If `Sresults` is obtained by solving a transient structural problem, then `structuralIC` uses the solution `Sresults` for the last time-step.

`structuralIC(structuralmodel,Sresults,iT)` uses the solution `Sresults` for the time-step `iT` from a previous structural analysis on the same geometry.

`struct_ic = structuralIC( ___ )` returns a handle to the structural initial conditions object.

Examples

Specify Initial Velocity

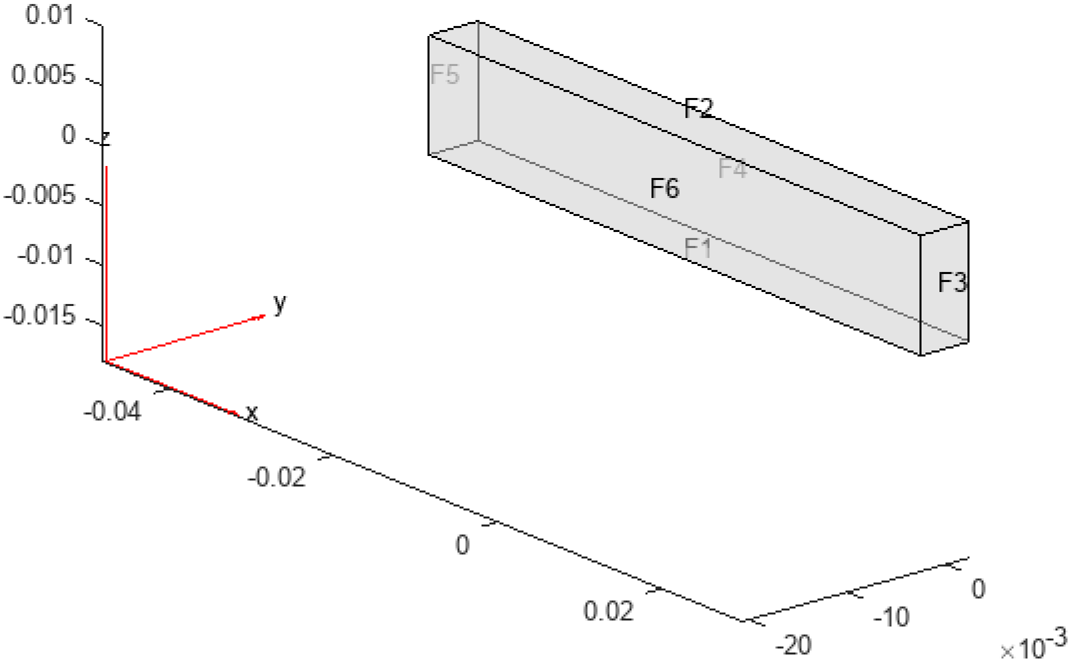
Specify initial velocity values for the entire geometry and for a particular face.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create a geometry and include it into the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify the zero initial velocity on the entire geometry. When you specify only the initial velocity or initial displacement, `structuralIC` assumes that the omitted parameter is zero. For example, here the initial displacement is also zero.

```
structuralIC(structuralmodel,"Velocity",[0;0;0])
```

```
ans = 
  GeometricStructuralICs with properties:

             RegionType: 'Cell'
               RegionID: 1
    InitialDisplacement: []
        InitialVelocity: [3x1 double]
```

Update the initial velocity on face 2 to model impulsive excitation.

```
structuralIC(structuralmodel,"Face",2,"Velocity",[0;60;0])
```

```
ans = 
  GeometricStructuralICs with properties:

             RegionType: 'Face'
               RegionID: 2
    InitialDisplacement: []
        InitialVelocity: [3x1 double]
```

Specify Nonconstant Initial Displacement by Using Function Handle

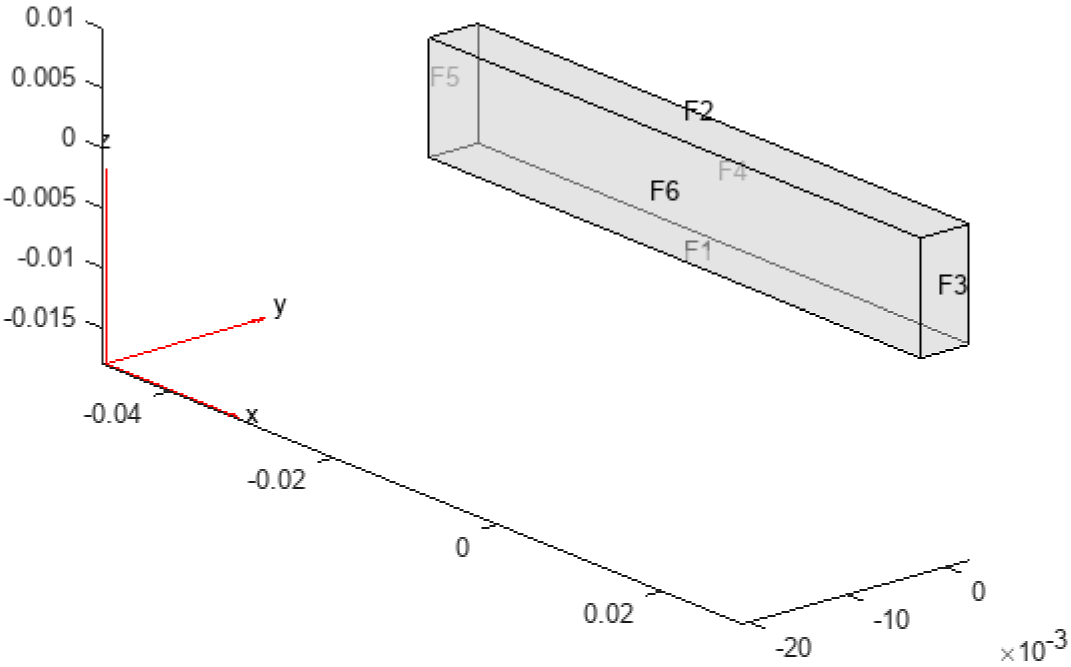
Specify initial *z*-displacement to be dependent on the coordinates *x* and *y*.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry and include it into the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify the zero initial displacement on the entire geometry.

```
structuralIC(structuralmodel,"Displacement",[0;0;0])
```

```
ans =
  GeometricStructuralICs with properties:

             RegionType: 'Cell'
               RegionID: 1
    InitialDisplacement: [3x1 double]
```

```
        InitialVelocity: []
```

Now change the initial displacement in the z-direction on face 2 to a function of the coordinates x and y:

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

Write the following function file. Save it to a location on your MATLAB® path.

```
function uinit = initdisp(location)

M = length(location.x);

uinit = zeros(3,M);

uinit(3,:) = location.x.^2 + location.y.^2;
```

Pass the initial displacement to your structural model.

```
structuralIC(structuralmodel,"Face",2,"Displacement",@initdisp)
```

```
ans = 
  GeometricStructuralICs with properties:

             RegionType: 'Face'
               RegionID: 2
    InitialDisplacement: @initdisp
        InitialVelocity: []
```

Use Static Solution as Initial Condition

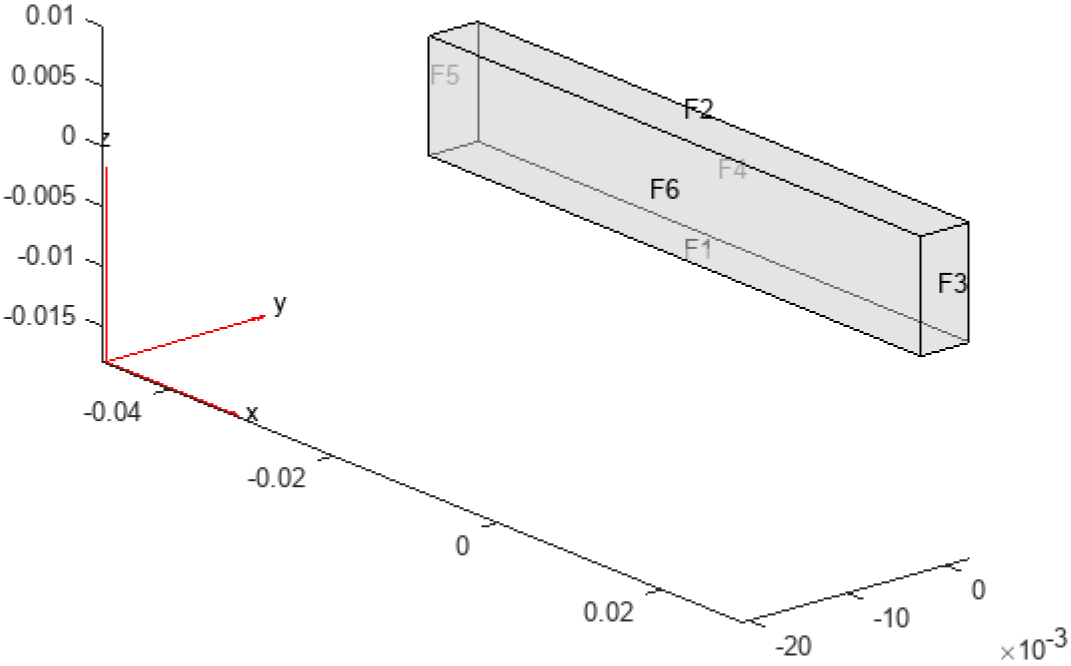
Use a static solution as an initial condition for a dynamic structural model.

Create a static model.

```
staticmodel = createpde("structural","static-solid");
```

Create the geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.06,0.005,0.01);
staticmodel.Geometry = gm;
pdegplot(staticmodel,"FaceLabels","on","FaceAlpha",0.5)
view(50,20)
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(staticmodel,"YoungsModulus",210E9, ...
                                 "PoissonsRatio",0.3,...
                                 "MassDensity",7800);
```

Apply the boundary condition and static load.

```
structuralBC(staticmodel,"Face",5,"Constraint","fixed");
structuralBoundaryLoad(staticmodel,"Face",3, ...
                                   "SurfaceTraction", ...
                                   [0;1E6;0]);
```

Generate a mesh and solve the model.

```
generateMesh(staticmodel,"Hmax",0.02);
Rstatic = solve(staticmodel);
```

Create a dynamic model and assign geometry.

```
dynamicmodel = createpde("structural","transient-solid");
gm = multicuboid(0.06,0.005,0.01);
dynamicmodel.Geometry = gm;
```

Apply the boundary condition.

```
structuralBC(dynamicmodel,"Face",5,"Constraint","fixed");
```

Generate a mesh.

```
generateMesh(dynamicmodel,"Hmax",0.02);
```

Specify the initial condition using the static solution.

```
structuralIC(dynamicmodel,Rstatic)
```

```
ans = 
  NodalStructuralICs with properties:

    InitialDisplacement: [113x3 double]
        InitialVelocity: [113x3 double]
```

Input Arguments

`structuralmodel` — Transient structural model

`StructuralModel` object

Transient structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, boundary conditions, and initial conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`u0` — Initial displacement

numeric vector | function handle

Initial displacement, specified as a numeric vector or function handle. A numeric vector must contain two elements for a 2-D model and three elements for a 3-D model. The elements represent the components of initial displacement.

Use a function handle to specify spatially varying initial displacement. The function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix corresponds to the initial displacement at the coordinates provided by the solver. For details, see “More About” on page 5-1266.

Example: `structuralIC(structuralmodel,"Face",[2,5],"Displacement",[0;0;0.01])`

Data Types: `double` | `function_handle`

`v0` — Initial velocity

numeric vector | function handle

Initial velocity, specified as a numeric vector or function handle. A numeric vector must contain two elements for a 2-D model and three elements for a 3-D model. The elements represent the components of initial velocity.

Use a function handle to specify spatially varying initial velocity. The function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix corresponds to the initial velocity at the coordinates provided by the solver. For details, see “More About” on page 5-1266.

Example: `structuralIC(structuralmodel,"Face",[2,5],"Displacement",[0;0;0.01],"Velocity",[0;60;0])`

Data Types: `double` | `function_handle`

`RegionType` — Geometric region type
`"Face"` | `"Edge"` | `"Vertex"` | `"Cell"`

Geometric region type, specified as `"Face"`, `"Edge"`, `"Vertex"`, or `"Cell"`.

When you apply multiple initial condition assignments, the solver uses these precedence rules for determining the initial condition.

- For multiple assignments to the same geometric region, the solver uses the last applied setting.
- For separate assignments to a geometric region and the boundaries of that region, the solver uses the specified assignment on the region and chooses the assignment on the boundary as follows. The solver gives 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).
- For an assignment made with the `results` object, the solver uses that assignment instead of all previous assignments.

Example: `structuralIC(structuralmodel,"Face",[2,5],"Displacement",[0;0;0.01],"Velocity",[0;60;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 by using `pdegplot`.

Example: `structuralIC(structuralmodel,"Face",[2,5],"Displacement",[0;0;0.01],"Velocity",[0;60;0])`

Data Types: `double`

`Sresults` — Structural model solution
`StaticStructuralResults` object | `TransientStructuralResults` object

Structural model solution, specified as a `StaticStructuralResults` or `TransientStructuralResults` object. Create `Sresults` by using `solve`.

`iT` — Time index
positive integer

Time index, specified as a positive integer.

Example: `structuralIC(structuralmodel,Sresults,21)`

Data Types: `double`

Output Arguments

`struct_ic` — Handle to initial conditions
`GeometricStructuralICs` object | `NodalStructuralICs` object

Handle to initial conditions, returned as a `GeometricStructuralICs` or `NodalStructuralICs` object. See GeometricStructuralICs Properties and NodalStructuralICs Properties.

`structuralIC` associates the structural initial condition with the geometric region in the case of a geometric assignment, or the nodes in the case of a results-based assignment.

More About

Specifying Nonconstant Parameters of a Structural Model

Use a function handle to specify the following structural parameters when they depend on space and, depending of the type of structural analysis, either time or frequency:

- Surface traction on the boundary
- Pressure normal to the boundary
- Concentrated force at a vertex
- Distributed spring stiffness for each translational direction used to model elastic foundation
- Enforced displacement and its components
- Initial displacement and velocity (can depend on space only)

For example, use function handles to specify the pressure load, *x*-component of the enforced displacement, and the initial displacement for this model.

```
structuralBoundaryLoad(model,"Face",12, ...
                             "Pressure",@myfunPressure)
structuralBC(model,"Face",2, ...
                   "XDisplacement",@myfunBC)
structuralIC(model,"Face",12, ...
                   "Displacement",@myfunIC)
```

For all parameters, except the initial displacement and velocity, the function must be of the form:

```
function structuralVal = myfun(location,state)
```

For the initial displacement and velocity the function must be of the form:

```
function structuralVal = myfun(location)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments (or one argument if the function specifies the initial displacement or initial velocity).

- `location` — A structure containing these fields:

  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes these data in the `location` structure:

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points

  - `location.nz` — For a 3-D or an axisymmetric geometry, *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, *r*-component of the normal vector at the evaluation point or points
- `state` — A structure containing these fields for dynamic structural problems:
  - `state.time` contains the time at evaluation points.
  - `state.frequency` contains the frequency at evaluation points.

  `state.time` and `state.frequency` are scalars.

Boundary constraints and loads get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`
- `state.time` or `state.frequency` (depending of the type of structural analysis)

Initial conditions get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

If a parameter represents a vector value, such as surface traction, spring stiffness, force, or displacement, your function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix corresponds to the parameter value (a vector) at the boundary coordinates provided by the solver.

If a parameter represents a scalar value, such as pressure or a displacement component, your function must return a row vector where each element corresponds to the parameter value (a scalar) at the boundary coordinates provided by the solver.

If boundary conditions depend on `state.time` or `state.frequency`, ensure that your function returns a matrix of `NaN` of the correct size when `state.frequency` 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.

Additional Arguments in Functions for Nonconstant Structural Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
structuralVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
structuralBC(model,"Face",2,"XDisplacement",structuralVal)

structuralVal = ...
@(location) myfunWithAdditionalArgs(location,arg1,arg2...)
structuralIC(model,"Face",2,"Displacement",structuralVal)
```

Version History

Introduced in R2018a

See Also

`StructuralModel` | `structuralProperties` | `structuralDamping` | `structuralBodyLoad` | `structuralBoundaryLoad` | `structuralBC` | `structuralSEInterface` | `solve` | `reduce` | `findStructuralIC` | GeometricStructuralICs Properties | NodalStructuralICs Properties

structuralDamping

Specify damping parameters for transient or frequency response structural model

Syntax

```
structuralDamping(structuralmodel,"Alpha",a,"Beta",b)
structuralDamping(structuralmodel,"Zeta",z)

damping = structuralDamping( ___ )
```

Description

`structuralDamping(structuralmodel,"Alpha",a,"Beta",b)` specifies proportional (Rayleigh) damping parameters `a` and `b` for a `structuralmodel` object.

For a frequency response model with damping, the results are complex. Use the `abs` and `angle` functions to obtain real-valued magnitude and phase, respectively.

`structuralDamping(structuralmodel,"Zeta",z)` specifies the modal damping ratio. Use this parameter when you solve a transient or frequency response model using the results of modal analysis.

`damping = structuralDamping( ___ )` returns the damping parameters object, using any of the previous input syntaxes.

Examples

Rayleigh Damping Parameters

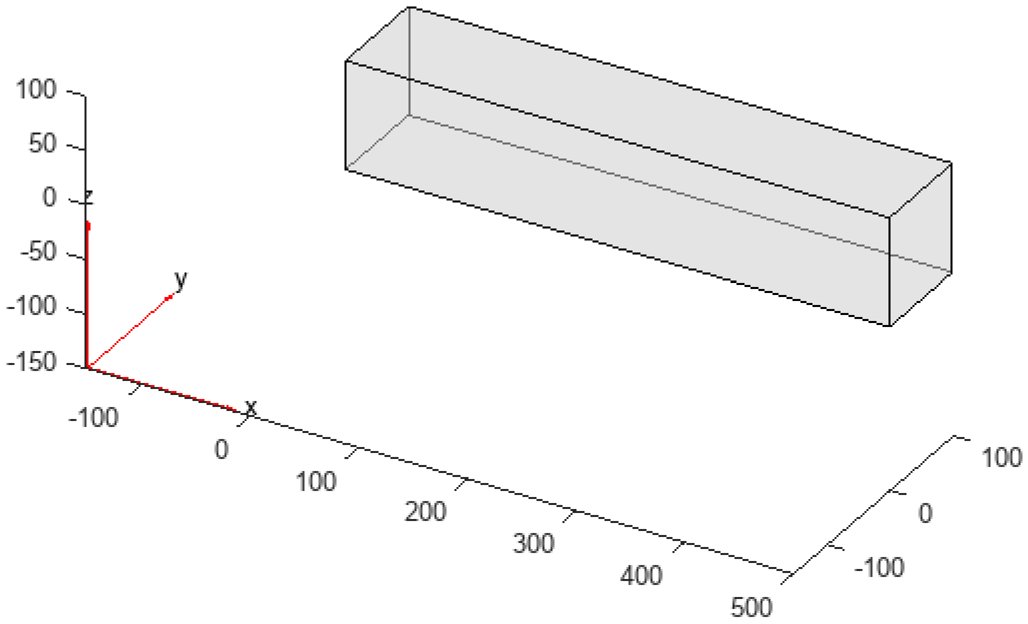
Specify proportional (Rayleigh) damping parameters for a beam.

Create a transient structural model.

```
structuralModel = createpde("structural","transient-solid");
```

Import and plot the geometry.

```
gm = importGeometry(structuralModel,"SquareBeam.stl");
pdegplot(structuralModel,"FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(structuralModel,"YoungsModulus",210E9,...
                                     "PoissonsRatio",0.3,...
                                     "MassDensity",7800);
```

Specify the Rayleigh damping parameters.

```
structuralDamping(structuralModel,"Alpha",10,"Beta",2)
```

```
ans =
  StructuralDampingAssignment with properties:

      RegionType: 'Cell'
        RegionID: 1
    DampingModel: "proportional"
           Alpha: 10
            Beta: 2
            Zeta: []
```

Solution to Frequency Response Structural Model with Damping

Solve a frequency response problem with damping. The resulting displacement values are complex. To obtain the magnitude and phase of displacement, use the `abs` and `angle` functions, respectively. To speed up computations, solve the model using the results of modal analysis.

Modal Analysis

Create a modal analysis model for a 3-D problem.

```
modelM = createpde("structural","modal-solid");
```

Create the geometry and include it in the model.

```
gm = multicuboid(10,10,0.025);
modelM.Geometry = gm;
```

Generate a mesh.

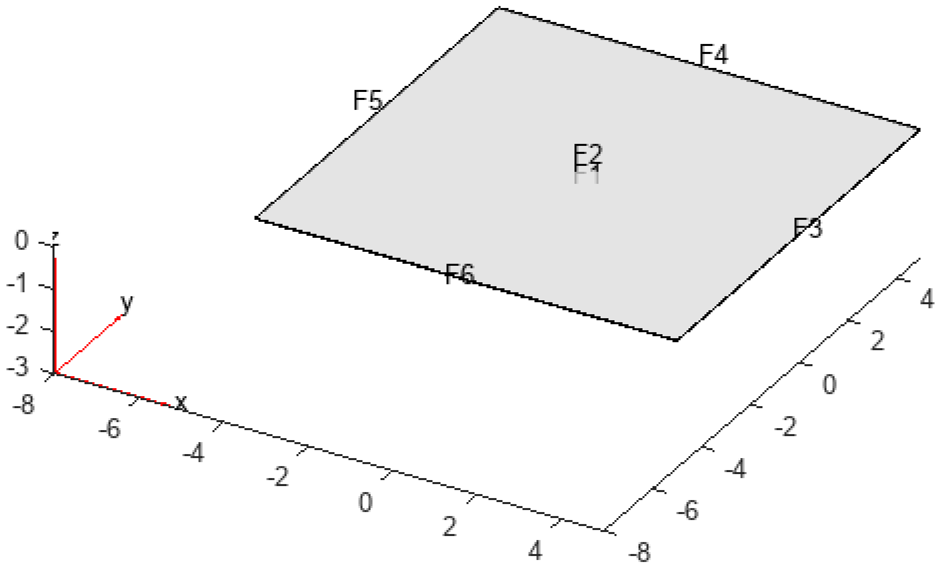
```
msh = generateMesh(modelM);
```

Specify Young's modulus, Poisson's ratio, and the mass density of the material.

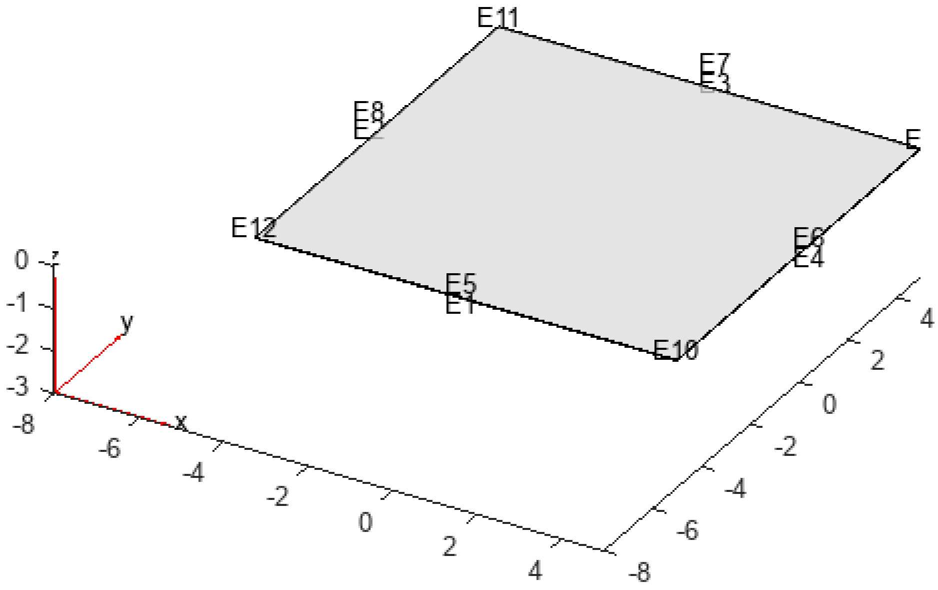
```
structuralProperties(modelM,"YoungsModulus",2E11, ...
                            "PoissonsRatio",0.3, ...
                            "MassDensity",8000);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with the face and edge labels.

```
pdegplot(gm,"FaceLabels","on","FaceAlpha",0.5)
```



```
figure
pdegplot(gm,"EdgeLabels","on","FaceAlpha",0.5)
```



Specify constraints on the sides of the plate (faces 3, 4, 5, and 6) to prevent rigid body motions.

```
structuralBC(modelM,"Face",[3,4,5,6],"Constraint","fixed");
```

Solve the problem for the frequency range from `-Inf` to `12*pi`.

```
Rm = solve(modelM,"FrequencyRange",[-Inf,12*pi]);
```

Frequency Response Analysis

Create a frequency response analysis model for a 3-D problem.

```
modelFR = createpde("structural","frequency-solid");
```

Use the same geometry and mesh as you used for the modal analysis.

```
modelFR.Geometry = gm;
modelFR.Mesh = msh;
```

Specify the same values for Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(modelFR,"YoungsModulus",2E11, ...
                             "PoissonsRatio",0.3, ...
                             "MassDensity",8000);
```

Specify the same constraints on the sides of the plate to prevent rigid body modes.

```
structuralBC(modelFR,"Face",[3,4,5,6],"Constraint","fixed");
```

Specify the pressure loading on top of the plate (face 2) to model an ideal impulse excitation. In the frequency domain, this pressure pulse is uniformly distributed across all frequencies.

```
structuralBoundaryLoad(modelFR,"Face",2,"Pressure",1E2);
```

First, solve the model without damping.

```
flist = [0,1,1.5,linspace(2,3,100),3.5,4,5,6]*2*pi;
RfrModalU = solve(modelFR,flist,"ModalResults",Rm);
```

Now, solve the model with damping equal to 2% of critical damping for all modes.

```
structuralDamping(modelFR,"Zeta",0.02);
RfrModalAll = solve(modelFR,flist,"ModalResults",Rm);
```

Solve the same model with frequency-dependent damping. In this example, use the solution frequencies from `flist` and damping values between 1% and 10% of critical damping.

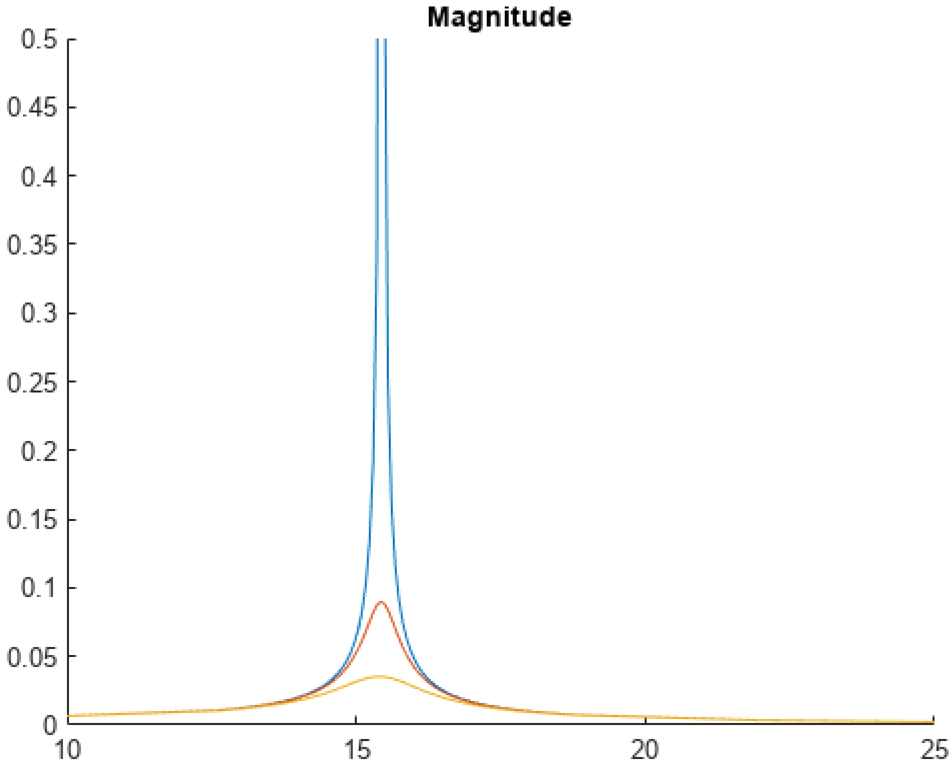
```
omega = flist;
zeta = linspace(0.01,0.1,length(omega));
zetaW = @(omegaMode) interp1(omega,zeta,omegaMode);
structuralDamping(modelFR,"Zeta",zetaW);

RfrModalFD = solve(modelFR,flist,"ModalResults",Rm);
```

Interpolate the displacement at the center of the top surface of the plate for all three cases.

```
iDispU = interpolateDisplacement(RfrModalU,[0;0;0.025]);
iDispAll = interpolateDisplacement(RfrModalAll,[0;0;0.025]);
iDispFD = interpolateDisplacement(RfrModalFD,[0;0;0.025]);
```

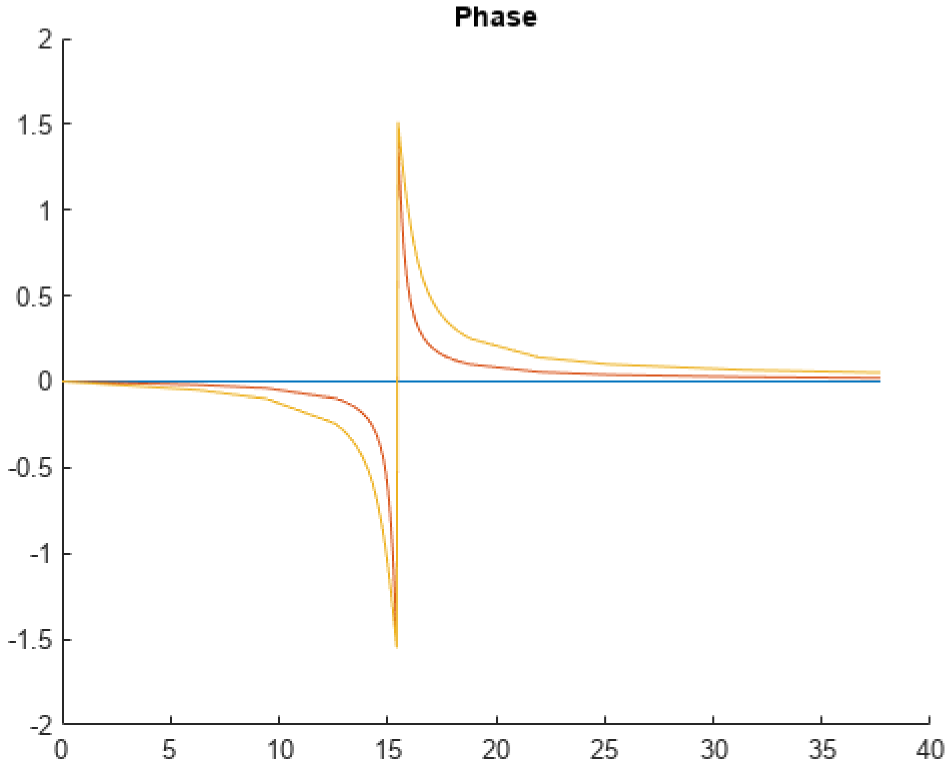
Plot the magnitude of the displacement. Zoom in on the frequencies around the first mode.

```
figure
hold on
plot(RfrModalU.SolutionFrequencies,abs(iDispU.Magnitude));
plot(RfrModalAll.SolutionFrequencies,abs(iDispAll.Magnitude));
plot(RfrModalFD.SolutionFrequencies,abs(iDispFD.Magnitude));
title("Magnitude")
xlim([10 25])
ylim([0 0.5])
```



Plot the phase of the displacement.

```
figure
hold on
plot(RfrModalU.SolutionFrequencies,angle(iDispU.Magnitude));
plot(RfrModalAll.SolutionFrequencies,angle(iDispAll.Magnitude));
plot(RfrModalFD.SolutionFrequencies,angle(iDispFD.Magnitude));
title("Phase")
```



Input Arguments

`structuralmodel` — Transient or frequency response structural model
`StructuralModel` object

Transient or frequency response structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, boundary conditions, and initial conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

a — Mass proportional damping
nonnegative number

Mass proportional damping, specified as a nonnegative number.

Data Types: `double`

b — Stiffness proportional damping
nonnegative number

Stiffness proportional damping, specified as a nonnegative number.

Data Types: `double`

z — Modal damping ratio
nonnegative number | function handle

Modal damping ratio, specified as a nonnegative number or a function handle. Use a function handle when each mode has its own damping ratio. The function must accept a vector of natural frequencies as an input argument and return a vector of corresponding damping ratios. It must cover the full frequency range for all modes used for modal solution. For details, see “Modal Damping Depending on Frequency” on page 5-1277.

Data Types: `double` | `function_handle`

Output Arguments

`damping` — Handle to damping parameters
`StructuralDampingAssignment` object

Handle to damping parameters, returned as a `StructuralDampingAssignment` object. See StructuralDampingAssignment Properties.

More About

Modal Damping Depending on Frequency

To use the individual value of modal damping for each mode, specify `z` as a function of frequency.

```
function z = dampingFcn(omega)
```

Typically, the damping ratio function is a linear interpolation of frequency versus the modal damping parameter:

```
structuralDamping(modelD,"Zeta",@(omegaMode) ...
                  interp1(omega,zeta,omegaMode))
```

Here, `omega` is a vector of frequencies, and `zeta` is a vector of corresponding damping ratio values.

Version History

Introduced in R2018a

Modal damping

For modal transient and modal frequency response models, specify damping as a percentage of critical damping for a selected vibration frequency.

See Also

`StructuralModel` | `structuralProperties` | `structuralBodyLoad` | `structuralBoundaryLoad` | `structuralBC` | `solve` | `findStructuralDamping` | StructuralDampingAssignment Properties

findStructuralDamping

Package: `pde`

Find damping model assigned to structural dynamics model

Syntax

```
dma = findStructuralDamping(structuralmodel.DampingModels)
```

Description

`dma = findStructuralDamping(structuralmodel.DampingModels)` returns the damping model and its parameters assigned to the entire structural dynamics model. The toolbox supports the proportional (Rayleigh) damping model and the modal damping model. The parameters of the proportional damping model are the mass and stiffness proportional damping parameters. The parameter of the modal damping model is the modal damping ratio.

Use this function to find which damping model and parameters are currently active if you made multiple damping assignments.

Examples

Find Damping Model Assignment

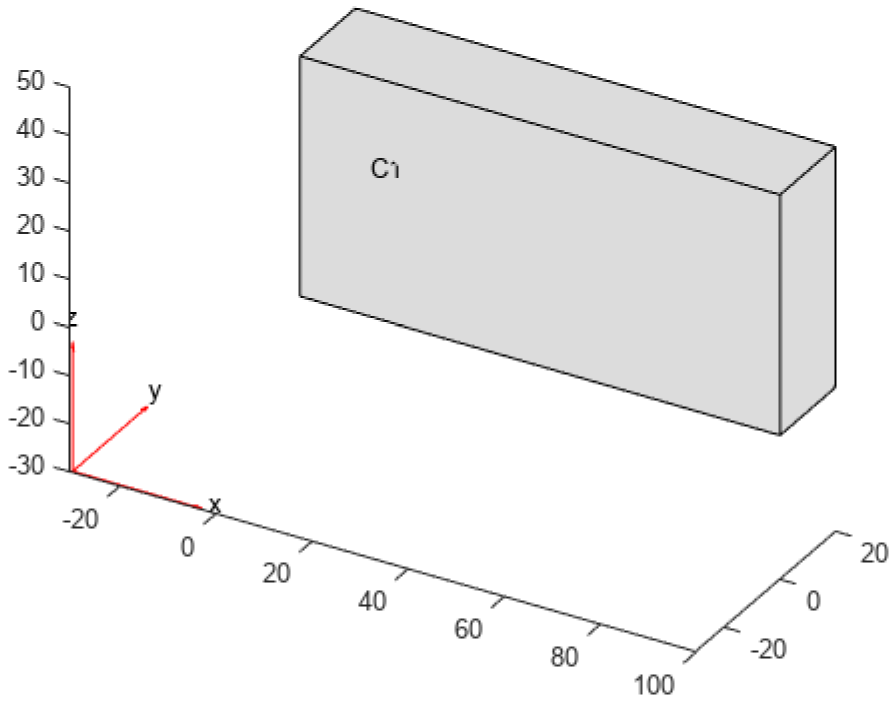
Find the damping model assignment for a 3-D model.

Create a transient structural model.

```
structuralModel = createpde("structural","transient-solid");
```

Import and plot the geometry.

```
importGeometry(structuralModel,"Block.stl");
pdegplot(structuralModel,"CellLabels","on")
```



Specify the stiffness proportional damping parameter.

```
structuralDamping(structuralModel,"Beta",40);
```

Now specify the mass proportional damping parameter.

```
structuralDamping(structuralModel,"Alpha",10);
```

Check the damping parameter assignment for `structuralModel`. Notice that the `Beta` parameter is empty.

```
findStructuralDamping(structuralModel.DampingModels)
```

```
ans = 
  StructuralDampingAssignment with properties:

      RegionType: 'Cell'
        RegionID: 1
    DampingModel: "proportional"
           Alpha: 10
            Beta: []
            Zeta: []
```

When you specify damping parameters by calling the `structuralDamping` function several times, the toolbox uses the last assignment. Specify both the mass and stiffness parameters.

```
structuralDamping(structuralModel,"Alpha",10,"Beta",40);
```

Check the damping parameter assignment for `structuralModel`.

```
findStructuralDamping(structuralModel.DampingModels)
```

```
ans = 
  StructuralDampingAssignment with properties:

      RegionType: 'Cell'
        RegionID: 1
    DampingModel: "proportional"
           Alpha: 10
            Beta: 40
            Zeta: []
```

Input Arguments

`structuralmodel.DampingModels` — Damping model
`DampingModels` property of `StructuralModel` object

Damping model of the structural model, specified as a `DampingModels` property of a `StructuralModel` object.

Output Arguments

dma — Damping model assignment
`StructuralDampingAssignment` object

Damping model assignment, returned as a `StructuralDampingAssignment` object. For details, see StructuralDampingAssignment Properties.

Version History

Introduced in R2018a

See Also

`structuralDamping` | StructuralDampingAssignment Properties

StructuralDampingAssignment Properties

Damping assignment for a structural analysis model

Description

A `StructuralDampingAssignment` object contains the damping model and its parameters for a structural analysis model. A `StructuralModel` container has a vector of `StructuralDampingAssignment` objects in its `DampingModels.StructuralDampingAssignments` property.

To set damping parameters for your structural model, use the `structuralDamping` function.

Properties

Properties

`RegionType` — Region type
`'Face'` | `'Cell'`

Region type, specified as `'Face'` for a 2-D region, or `'Cell'` for a 3-D region.

Data Types: `char`

`RegionID` — Region ID
positive integer

Region ID, specified as a positive integer.

Data Types: `double`

`DampingModel` — Damping model type
`"proportional"` | `"modal"` | `"hysteretic"`

Damping model type, specified as `"proportional"`, `"modal"`, or `"hysteretic"`.

Data Types: `string`

`Alpha` — Mass proportional damping parameter
nonnegative number

Mass proportional damping parameter, specified as a nonnegative number.

Data Types: `double`

`Beta` — Stiffness proportional damping parameter
nonnegative number

Stiffness proportional damping parameter, specified as a nonnegative number.

Data Types: `double`

`Zeta` — Modal damping ratio
nonnegative number | function handle

Modal damping ratio, specified as a nonnegative number or a function handle. Use a function handle when each mode has its own damping ratio. The function must accept a vector of natural frequencies as an input argument and return a vector of corresponding damping ratios. It must cover the full frequency range for all modes used for modal solution.

Data Types: `double` | `function_handle`

Version History

Introduced in R2018a

See Also

`structuralDamping` | `findStructuralDamping`

StructuralMaterialAssignment Properties

Structural material property assignments

Description

A `StructuralMaterialAssignment` object contains the description of material properties of a structural analysis model. A `StructuralModel` container has a vector of `StructuralMaterialAssignment` objects in its `MaterialProperties.MaterialAssignments` property.

To create the material properties assignments for your structural analysis model, use the `structuralProperties` function.

Properties

Properties of StructuralMaterialAssignment

`RegionType` — Region type
`'Face'` | `'Cell'`

Region type, specified as `'Face'` for a 2-D region, or `'Cell'` for a 3-D region.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified as a vector of positive integers. To determine which ID corresponds to which portion of the geometry, use the `pdegplot` function, setting the `'FaceLabels'` name-value pair to `'on'`.

Data Types: `double`

`YoungsModulus` — Young's modulus
positive number

Young's modulus of the material, specified as a positive number.

Data Types: `double`

`PoissonsRatio` — Poisson's ratio
positive number

Poisson's ratio of the material, specified as a positive number.

Data Types: `double`

`MassDensity` — Mass density
positive number

Mass density of the material, specified as a positive number. This property is required when modeling gravitational effects.

Data Types: `double`

`CTE` — Coefficient of thermal expansion
real number

Coefficient of thermal expansion, specified as a real number. This argument is required for thermal stress analysis. Thermal stress analysis requires the structural model to be static.

Data Types: `double`

`HystereticDamping` — Hysteretic damping parameter
nonnegative number

Hysteretic damping parameter, specified as a nonnegative number. This type of damping is also called structural damping.

Data Types: `double`

Version History

Introduced in R2017b

See Also

`findStructuralProperties` | `structuralProperties`

structuralBodyLoad

Package: `pde`

Specify body load for structural model

Syntax

`structuralBodyLoad(structuralmodel,"GravitationalAcceleration",GAval)`

`structuralBodyLoad(structuralmodel,"AngularVelocity",omega)`

```
structuralBodyLoad(structuralmodel,"Temperature",Tval)
structuralBodyLoad(structuralmodel,"Temperature",Tresults)
structuralBodyLoad(structuralmodel,"Temperature",Tresults,"TimeStep",iT)
```

`structuralBodyLoad(structuralmodel, ___ )`

`structuralBodyLoad( ___ ,"Label",labeltext)`

`bodyLoad = structuralBodyLoad( ___ )`

Description

`structuralBodyLoad(structuralmodel,"GravitationalAcceleration",GAval)` specifies acceleration due to gravity as a body load for a static or transient structural model. Structural models for modal analysis cannot have body loads.

`structuralBodyLoad(structuralmodel,"AngularVelocity",omega)` specifies an angular velocity to model centrifugal loading for an axisymmetric structural model.

`structuralBodyLoad(structuralmodel,"Temperature",Tval)` specifies a thermal load on a static structural analysis model.

Tip If `Tval` is the temperature itself, and not a change in temperature, you must specify a reference temperature using `structuralmodel.ReferenceTemperature`. Otherwise, the toolbox uses the default value (zero) for the reference temperature. For details, see `StructuralModel`.

`structuralBodyLoad(structuralmodel,"Temperature",Tresults)` uses the steady-state or transient thermal analysis results `Tresults` to specify a thermal load on a static structural analysis model. If `Tresults` is the solution of a transient thermal problem, then this syntax uses the temperature and its gradients from the last time step.

`structuralBodyLoad(structuralmodel,"Temperature",Tresults,"TimeStep",iT)` uses the transient thermal analysis results `Tresults` and the time step index `iT` to specify a thermal load on a static structural analysis model.

`structuralBodyLoad(structuralmodel, ___ )` specifies several body loads for the same structural model. Use any arguments from the previous syntaxes applicable to your `structuralmodel`. For example, specify the gravity and thermal loads as `structuralBodyLoad(structuralmodel,"GravitationalAcceleration",[0;0;-9.8],"Temperature",300)`. Do not use subsequent function calls when assigning several body loads because the toolbox uses only the last assignment.

`structuralBodyLoad( ___ ,"Label",labeltext)` adds a label for the structural body load to be used by the `linearizeInput` function. This function lets you pass body loads to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

`bodyLoad = structuralBodyLoad( ___ )` returns the body load object.

Examples

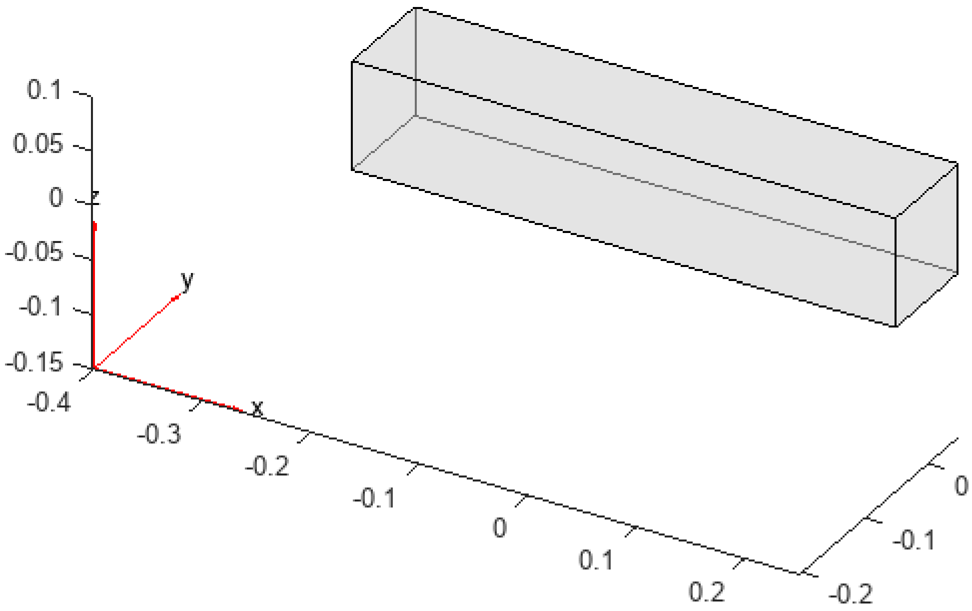
Gravity Load on Beam

Create a structural model.

```
structuralModel = createpde("structural","static-solid");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.1,0.1);
structuralModel.Geometry = gm;
pdegplot(structuralModel,"FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density. The mass density value is required for modeling gravitational effects.

```
structuralProperties(structuralModel,"YoungsModulus",210E3, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",2.7E-6);
```

Specify the gravity load on the beam.

```
structuralBodyLoad(structuralModel, ...
    "GravitationalAcceleration",[0;0;-9.8])
```

```
ans = 
  BodyLoadAssignment with properties:

                   RegionType: 'Cell'
                     RegionID: 1
    GravitationalAcceleration: [3x1 double]
              AngularVelocity: []
                  Temperature: []
                     TimeStep: []
                        Label: []
```

Static Analysis of Spinning Disk with Press-Fit at Hub

Analyze a spinning disk with radial compression at the hub due to press-fit. The inner radius of the disk is 0.05, and the outer radius is 0.2. The thickness of the disk is 0.05 with an interference fit of 50E-6. For this analysis, simplify the 3-D axisymmetric model to a 2-D model.

Create a static structural analysis model for solving an axisymmetric problem.

```
structuralmodel = createpde("structural","static-axisymmetric");
```

The 2-D model is a rectangular strip whose *x*-dimension extends from the hub to the outer surface, and whose *y*-dimension extends over the height of the disk. Create the geometry by specifying the coordinates of the strip's four corners. For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through $r = 0$, which is equivalent to $x = 0$.

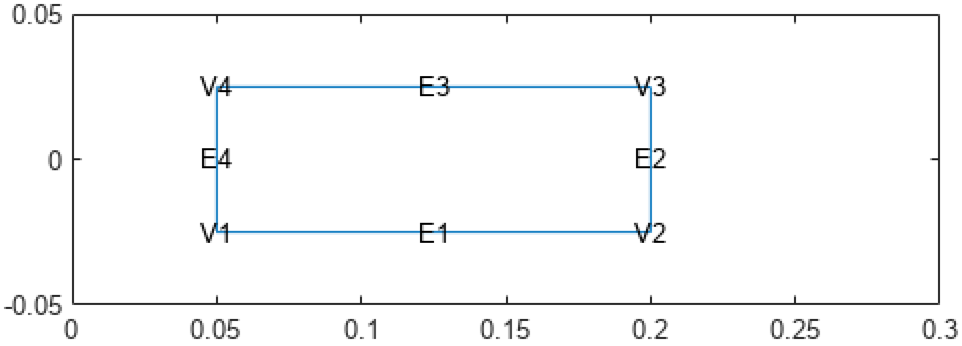
```
g = decsg([3 4 0.05 0.2 0.2 0.05 -0.025 -0.025 0.025 0.025]');
```

Include the geometry in the model.

```
geometryFromEdges(structuralmodel,g);
```

Plot the geometry with the edge and vertex labels.

```
figure
pdegplot(structuralmodel,"EdgeLabels","on","VertexLabels","on")
xlim([0 0.3])
ylim([-0.05 0.05])
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(structuralmodel,"YoungsModulus",210e9, ...
                                     "PoissonsRatio",0.28, ...
                                     "MassDensity",7700);
```

Apply centrifugal load due to spinning of the disk. Assume that the disk is spinning at 104.7 rad/s.

```
structuralBodyLoad(structuralmodel,"AngularVelocity",1047);
```

Apply radial displacement at the hub of the disk to model press-fit.

```
structuralBC(structuralmodel,"Edge",4,"RDisplacement",50e-6);
```

Fix axial displacement of a point on the hub to prevent rigid body motion.

```
structuralBC(structuralmodel,"Vertex",1,"ZDisplacement",0);
```

Generate a mesh.

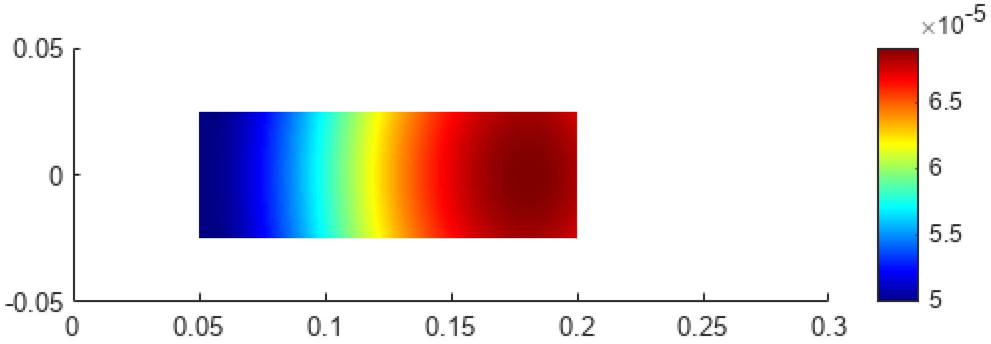
```
generateMesh(structuralmodel);
```

Solve the model.

```
structuralresults = solve(structuralmodel);
```

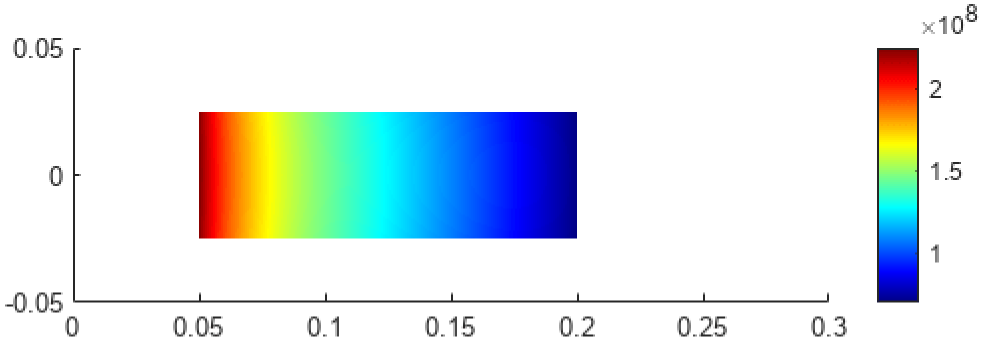
Plot the radial displacement of the disk.

```
figure
pdeplot(structuralmodel, ...
        "XYData",structuralresults.Displacement.ur, ...
        "ColorMap","jet")
axis equal
xlim([0 0.3])
ylim([-0.05 0.05])
```



Plot circumferential (hoop) stress.

```
figure
pdeplot(structuralmodel, ...
        "XYData",structuralresults.Stress.sh, ...
        "ColorMap","jet")
axis equal
xlim([0 0.3])
ylim([-0.05 0.05])
```



Constant Thermal Load

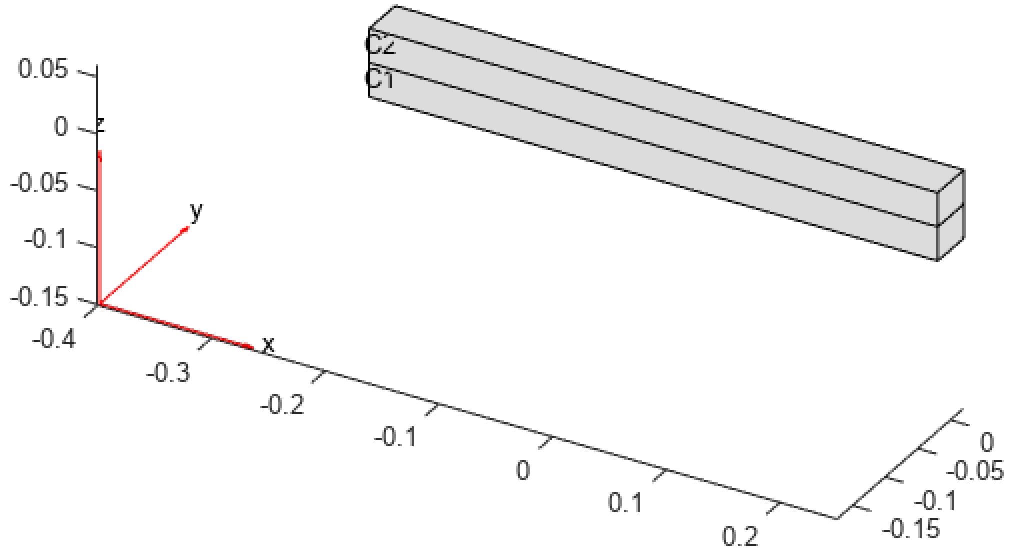
Specify a constant temperature rise for a thermal stress analysis of a bimetallic cantilever beam.

Create a static structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.04,[0.03,0.03],"Zoffset",[0,0.03]);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"CellLabels","on")
```



Set the reference temperature. This temperature corresponds to the state of zero thermal stress of the model.

```
structuralmodel.ReferenceTemperature = 20
```

```
structuralmodel = 
  StructuralModel with properties:

              AnalysisType: "static-solid"
                  Geometry: [1x1 DiscreteGeometry]
        MaterialProperties: []
                 BodyLoads: []
        BoundaryConditions: []
      ReferenceTemperature: 20
    SuperelementInterfaces: []
                      Mesh: []
             SolverOptions: [1x1 pde.PDESolverOptions]
```

Apply the constant temperature as a structural body load.

```
structuralBodyLoad(structuralmodel,"Temperature",300)
```

```
ans = 
  BodyLoadAssignment with properties:

                   RegionType: 'Cell'
                     RegionID: [1 2]
```

```
    GravitationalAcceleration: []
              AngularVelocity: []
                  Temperature: 300
                     TimeStep: []
                        Label: []
```

Thermal Load as Steady-State Thermal Model Solution

Specify a thermal load using the solution from a steady-state thermal analysis on the same geometry and mesh.

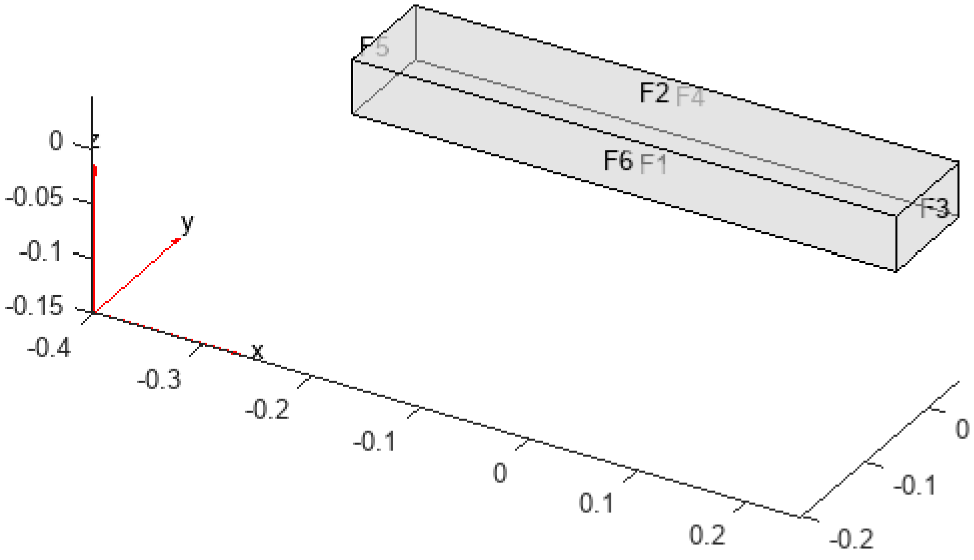
Steady-State Thermal Model Analysis

Create a steady-state thermal model.

```
thermalmodel = createpde("thermal","steadystate");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.1,0.05);
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Generate a mesh.

```
generateMesh(thermalmodel);
```

Specify the thermal conductivity of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",5e-3);
```

Specify constant temperatures on the left and right ends on the beam.

```
thermalBC(thermalmodel,"Face",3,"Temperature",100);
thermalBC(thermalmodel,"Face",5,"Temperature",0);
```

Specify the heat source over the entire geometry.

```
internalHeatSource(thermalmodel,10);
```

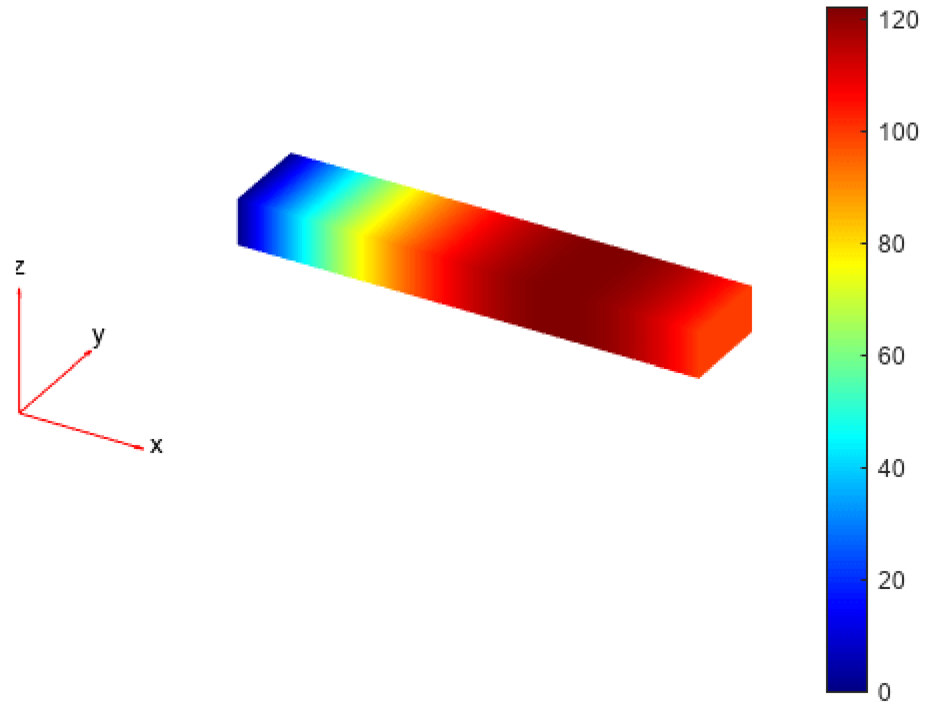
Solve the model.

```
thermalresults = solve(thermalmodel)
```

```
thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [3870x1 double]
     XGradients: [3870x1 double]
     YGradients: [3870x1 double]
     ZGradients: [3870x1 double]
           Mesh: [1x1 FEMesh]
```

Plot the temperature distribution.

```
pdeplot3D(thermalmodel,"ColorMapData",thermalresults.Temperature)
```



Static Structural Analysis with Thermal Load

Create a static structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Include the same geometry as for the thermal model.

```
structuralmodel.Geometry = gm;
```

Apply the solution of the thermal model analysis as a body load for the structural model.

```
structuralBodyLoad(structuralmodel,"Temperature",thermalresults)
```

```
ans = 
  BodyLoadAssignment with properties:

                   RegionType: 'Cell'
                     RegionID: 1
    GravitationalAcceleration: []
              AngularVelocity: []
                  Temperature: [1x1 pde.SteadyStateThermalResults]
                     TimeStep: []
                        Label: []
```

Thermal Load as Transient Thermal Model Solution

Specify a thermal load using the solution from a transient thermal analysis on the same geometry and mesh.

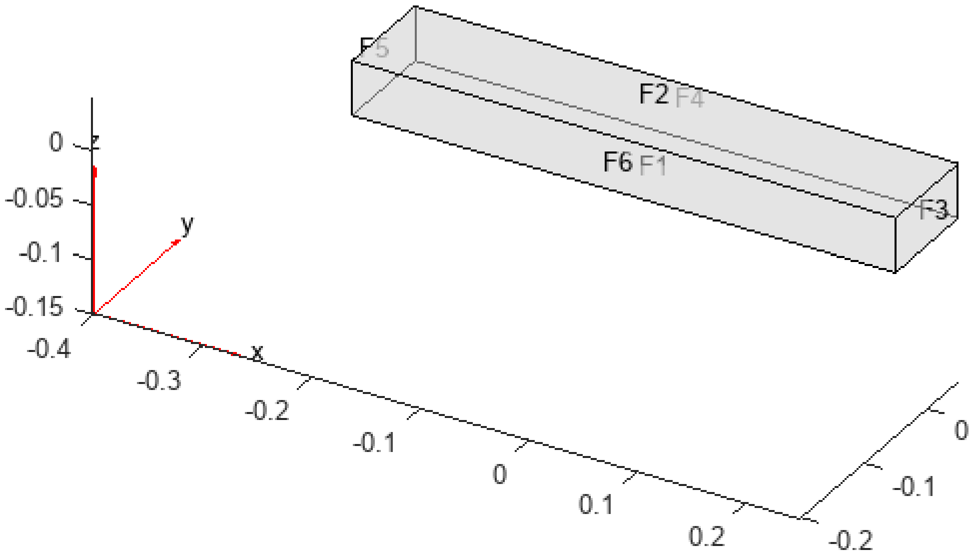
Transient Thermal Model Analysis

Create a transient thermal model.

```
thermalmodel = createpde("thermal","transient");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.1,0.05);
thermalmodel.Geometry = gm;
pdegplot(thermalmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Generate a mesh.

```
generateMesh(thermalmodel);
```

Specify the thermal properties of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",5e-3, ...
                               "MassDensity",2.7*10^(-6), ...
                               "SpecificHeat",10);
```

Specify the constant temperatures on the left and right ends on the beam.

```
thermalBC(thermalmodel,"Face",3,"Temperature",100);
thermalBC(thermalmodel,"Face",5,"Temperature",0);
```

Specify the heat source over the entire geometry.

```
internalHeatSource(thermalmodel,10);
```

Set the initial temperature.

```
thermalIC(thermalmodel,0);
```

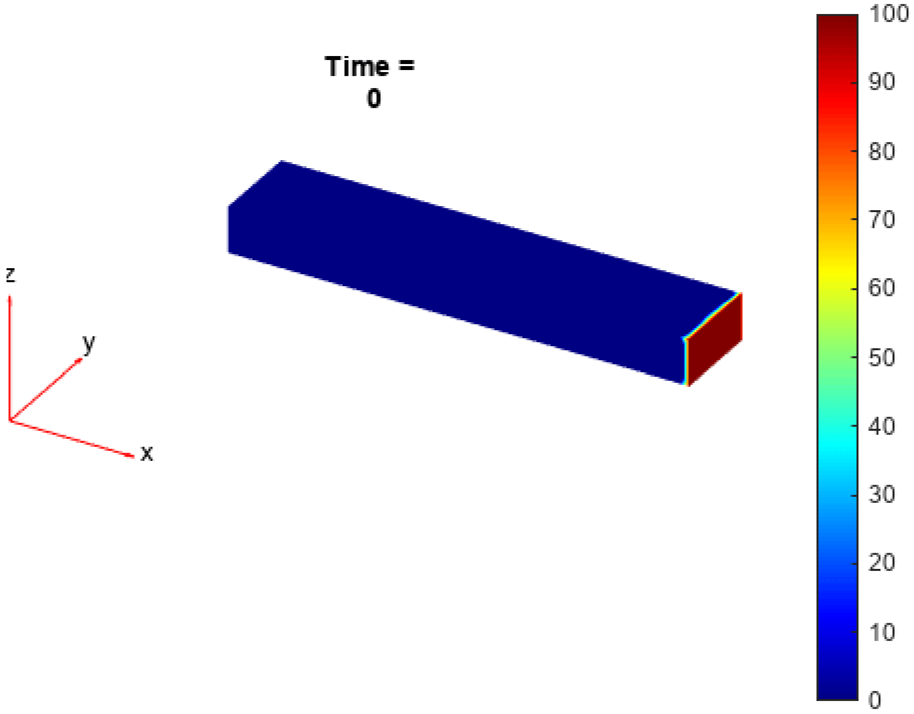
Solve the model.

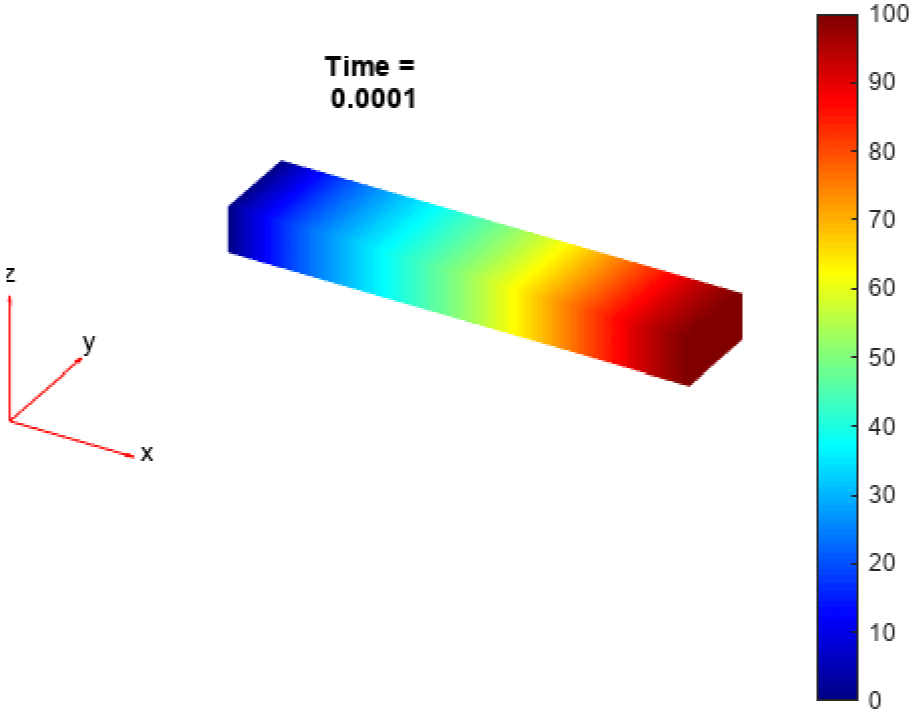
```
tlist = [0:1e-4:2e-4];
thermalresults = solve(thermalmodel,tlist)
```

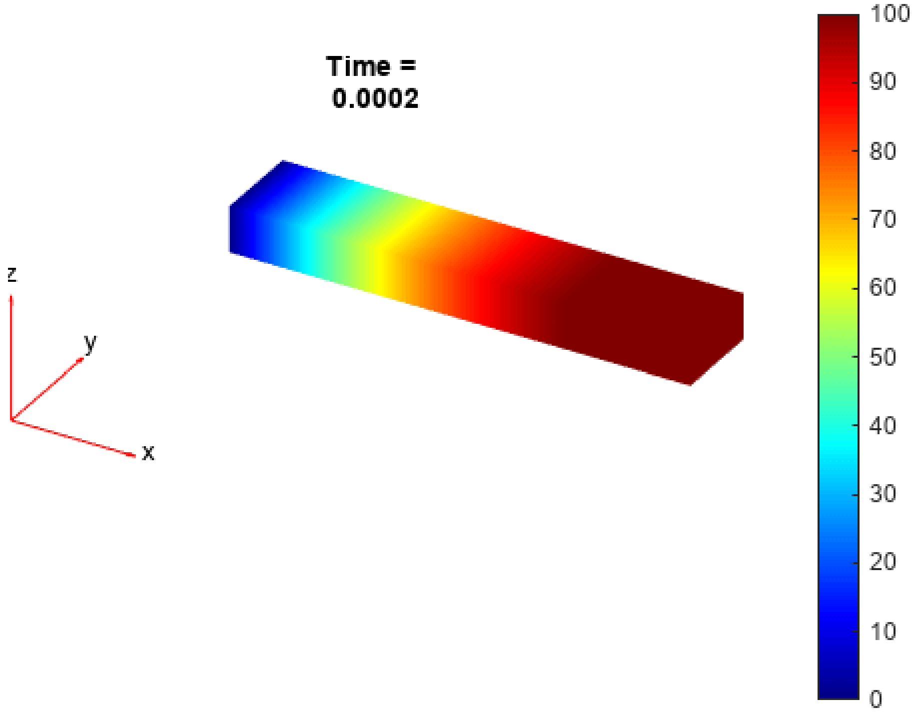
```
thermalresults = 
  TransientThermalResults with properties:

      Temperature: [3870x3 double]
    SolutionTimes: [0 1.0000e-04 2.0000e-04]
       XGradients: [3870x3 double]
       YGradients: [3870x3 double]
       ZGradients: [3870x3 double]
             Mesh: [1x1 FEMesh]
```

Plot the temperature distribution for each time step.

```
for n = 1:numel(thermalresults.SolutionTimes)
    figure
    pdeplot3D(thermalmodel,"ColorMapData", ...
          thermalresults.Temperature(:,n))
    title(["Time = " num2str(tlist(n))])
    caxis([0 100])
end
```


Time =
0
z
y
x
100
90
80
70
60
50
40
30
20
10
0


Time =
0.0001
z
y
x
100
90
80
70
60
50
40
30
20
10
0



Static Structural Analysis with Thermal Load

Create a static structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Include the same geometry as for the thermal model.

```
structuralmodel.Geometry = gm;
```

Apply the solution of the thermal model analysis as a body load for the structural model. By default, `structuralBodyLoad` uses the thermal model solution for the last time step.

```
structuralBodyLoad(structuralmodel,"Temperature",thermalresults);
```

You also can specify the time step you want to use. For example, apply the thermal model solution for the second time step as a body load for the structural model.

```
structuralBodyLoad(structuralmodel,"Temperature",thermalresults, ...
                                   "TimeStep",2);
```

Input Arguments

`structuralmodel` — Static or transient structural model
`StructuralModel` object

Static or transient structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`GAval` — Acceleration due to gravity

numeric vector

Acceleration due to gravity, specified as a numeric vector. `GAval` must be specified in units consistent with those of the geometry and material properties.

Example: `structuralBodyLoad(structuralmodel,"GravitationalAcceleration", [0;0;-9.8])`

Data Types: `double`

omega — Angular velocity for axisymmetric model

positive number

Angular velocity for an axisymmetric model, specified as a positive number. `omega` must be specified in units consistent with those of the geometry and material properties.

For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through $r = 0$, which is equivalent to $x = 0$.

Example: `structuralBodyLoad(structuralmodel,"AngularVelocity",2.3)`

Data Types: `double`

`Tval` — Constant thermal load

real number

Constant thermal load on a static structural model, specified as a real number. `Tval` must be specified in units consistent with those of the geometry and material properties.

Example: `structuralBodyLoad(structuralmodel,"Temperature",300)`

Data Types: `double`

`Tresults` — Thermal model solution

`StaticThermalResults` object | `TransientThermalResults` object

Thermal model solution applied as a body load on a static structural model, specified as a `StaticThermalResults` or `TransientThermalResults` object. Create `Tresults` by using `solve`.

Example: `Tresults = solve(thermalmodel);`
`structuralBodyLoad(structuralmodel,"Temperature",Tresults)`

`iT` — Time index

positive integer

Time index, specified as a positive integer.

Example:
`structuralBodyLoad(structuralmodel,"Temperature",Tresults,"TimeStep",21)`

Data Types: `double`

`labeltext` — Label for structural body load
character vector | string

Label for the structural body load, specified as a character vector or a string.

Data Types: `char` | `string`

Output Arguments

bodyLoad — Handle to body load
`BodyLoadAssignment` object

Handle to body load, returned as a `BodyLoadAssignment` object. See BodyLoadAssignment Properties.

Version History

Introduced in R2017b

Label to extract sparse linear models for use with Control System Toolbox

Now you can add a label for structural body loads to be used by the `linearizeInput` function. This function lets you pass structural body loads to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

Axisymmetric analysis

You can now model the effect of a spinning axisymmetric structure by specifying a centrifugal load.

Thermal load

You can now specify a thermal load.

See Also

`StructuralModel` | `structuralProperties` | `structuralDamping` | `structuralBoundaryLoad` | `structuralBC` | BodyLoadAssignment Properties

structuralBoundaryLoad

Package: `pde`

Specify boundary loads for structural model

Syntax

```
structuralBoundaryLoad(structuralmodel,RegionType,RegionID,"SurfaceTraction",
STval,"Pressure",Pval,"TranslationalStiffness",TSval)
structuralBoundaryLoad(structuralmodel,"Vertex",VertexID,"Force",Fval)

structuralBoundaryLoad( ___ ,"Vectorized","on")

structuralBoundaryLoad( ___ ,"Pressure",Pval,Name,Value)
structuralBoundaryLoad(structuralmodel,"Vertex",VertexID,"Force",Fval,
Name,Value)

structuralBoundaryLoad( ___ ,"Label",labeltext)

boundaryLoad = structuralBoundaryLoad( ___ )
```

Description

`structuralBoundaryLoad(structuralmodel,RegionType,RegionID,"SurfaceTraction",STval,"Pressure",Pval,"TranslationalStiffness",TSval)` specifies the surface traction, pressure, and translational stiffness on the boundary of type `RegionType` with `RegionID` ID numbers.

- Surface traction is determined as distributed normal and tangential forces acting on a boundary, resolved along the global Cartesian coordinate system.
- Pressure must be specified in the direction that is normal to the boundary. A positive pressure value acts into the boundary (for example, compression). A negative pressure value acts away from the boundary (for example, suction).
- Translational stiffness is a distributed spring stiffness for each translational direction. Translational stiffness is used to model an elastic foundation.

`structuralBoundaryLoad` does not require you to specify all three boundary loads. Depending on your structural analysis problem, you can specify one or more boundary loads by picking the corresponding arguments and omitting others. You can specify translational stiffness for any structural model. To specify pressure or surface traction, `structuralmodel` must be a static, transient, or frequency response model. Structural models for modal analysis cannot have pressure or surface traction.

The default boundary load is a stress-free boundary condition.

`structuralBoundaryLoad(structuralmodel,"Vertex",VertexID,"Force",Fval)` specifies concentrated force at a vertex with the `VertexID` number. You can specify force only if `structuralmodel` is a static, transient, or frequency response model. Structural models for modal analysis cannot have concentrated force.

`structuralBoundaryLoad( ___ ,"Vectorized","on")` uses vectorized function evaluation when you pass a function handle as an argument. If your function handle computes in a vectorized fashion, then using this argument saves time. See “Vectorization”. For details on this evaluation, see “Nonconstant Boundary Conditions” on page 2-132.

Use this syntax with any of the input arguments from previous syntaxes.

`structuralBoundaryLoad( ___ ,"Pressure",Pval,Name,Value)` lets you specify the form and duration of a nonconstant pressure pulse and harmonic excitation for a transient structural model without creating a function handle. When using this syntax, you must specify the model, region type and region ID, and pressure. Surface traction and translational stiffness are optional arguments. This syntax does not work for static, modal analysis, and frequency response models.

`structuralBoundaryLoad(structuralmodel,"Vertex",VertexID,"Force",Fval, Name,Value)` lets you specify the form and duration of a nonconstant concentrated force and harmonic excitation for a transient structural model without creating a function handle.

`structuralBoundaryLoad( ___ ,"Label",labeltext)` adds a label for the structural boundary load to be used by the `linearizeInput` function. This function lets you pass boundary loads to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

`boundaryLoad = structuralBoundaryLoad( ___ )` returns the boundary load object.

Examples

Apply Fixed Boundaries and Specify Surface Traction

Apply fixed boundaries and traction on two ends of a bimetallic cable.

Create a structural model.

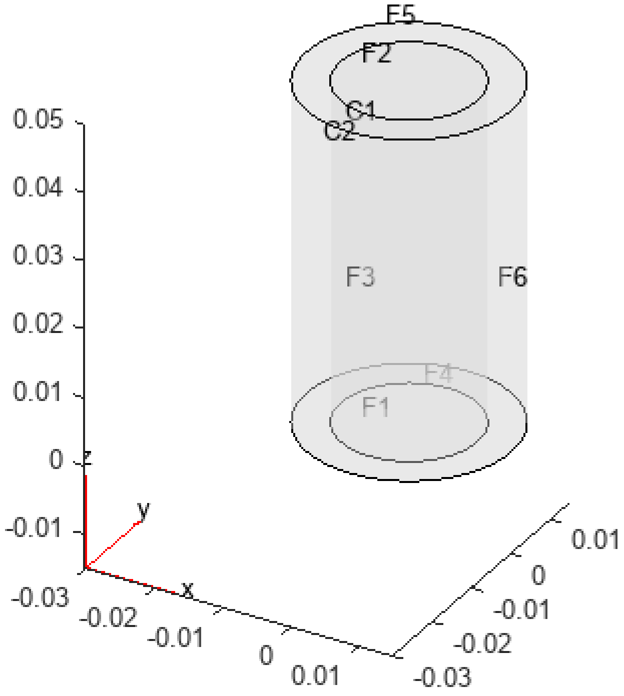
```
structuralModel = createpde("structural","static-solid");
```

Create nested cylinders to model a bimetallic cable.

```
gm = multicylinder([0.01,0.015],0.05);
```

Assign the geometry to the structural model and plot the geometry.

```
structuralModel.Geometry = gm;
pdegplot(structuralModel,"CellLabels","on", ...
                         "FaceLabels","on", ...
                         "FaceAlpha",0.4)
```



For each metal, specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralModel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28);
structuralProperties(structuralModel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3);
```

Specify that faces 1 and 4 are fixed boundaries.

```
structuralBC(structuralModel,"Face",[1,4],"Constraint","fixed")
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: [1 4]
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: "fixed"
                    Radius: []
                 Reference: []
                     Label: []
```

```
   Boundary Loads
                      Force: []
             SurfaceTraction: []
                    Pressure: []
    TranslationalStiffness: []
                       Label: []
```

Specify the surface traction for faces 2 and 5.

```
structuralBoundaryLoad(structuralModel, ...
                          "Face",[2,5], ...
                          "SurfaceTraction",[0;0;100])
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: [2 5]
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
           SurfaceTraction: [3x1 double]
                  Pressure: []
    TranslationalStiffness: []
                     Label: []
```

Specify Translational Stiffness

Create a structural model.

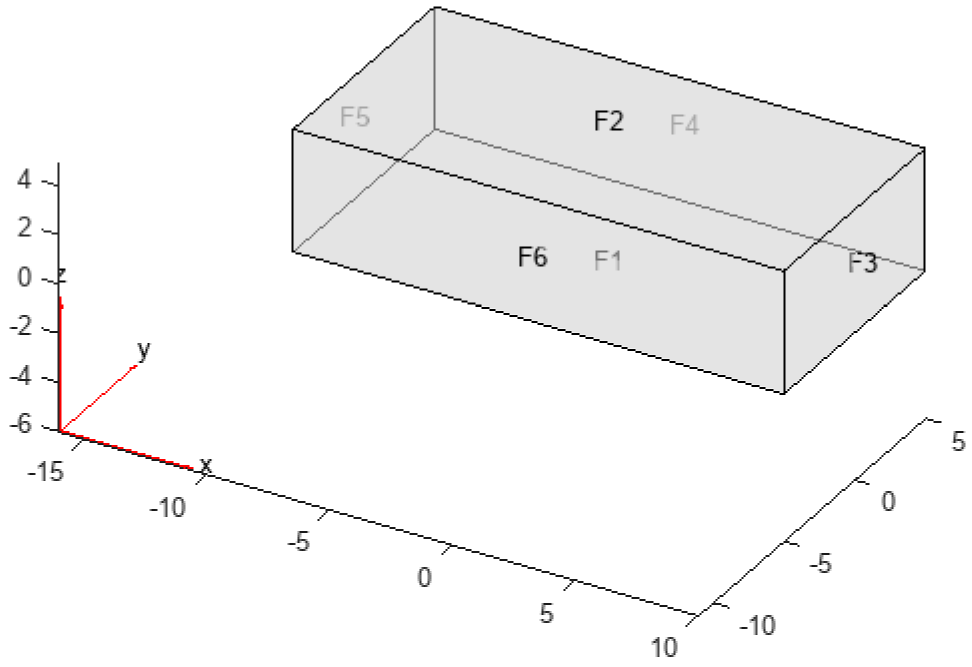
```
structuralModel = createpde("structural","static-solid");
```

Create a block geometry.

```
gm = multicuboid(20,10,5);
```

Assign the geometry to the structural model and plot the geometry.

```
structuralModel.Geometry = gm;
pdegplot(structuralModel,"FaceLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralModel,"YoungsModulus",30, ...
                                     "PoissonsRatio",0.3);
```

The bottom face of the block rests on an elastic foundation (a spring). To model this foundation, specify the translational stiffness.

```
structuralBoundaryLoad(structuralModel, ...
                              "Face",1, ...
                              "TranslationalStiffness",[0;0;30])
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 1
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []
```

```
   Boundary Loads
                    Force: []
           SurfaceTraction: []
                  Pressure: []
    TranslationalStiffness: [3x1 double]
                     Label: []
```

Apply Concentrated Force at Point

Specify a force value at a vertex of a geometry.

Create a structural model for static analysis of a solid (3-D) problem.

```
model = createpde("structural","static-solid");
```

Create the geometry, which consists of two cuboids stacked on top of each other.

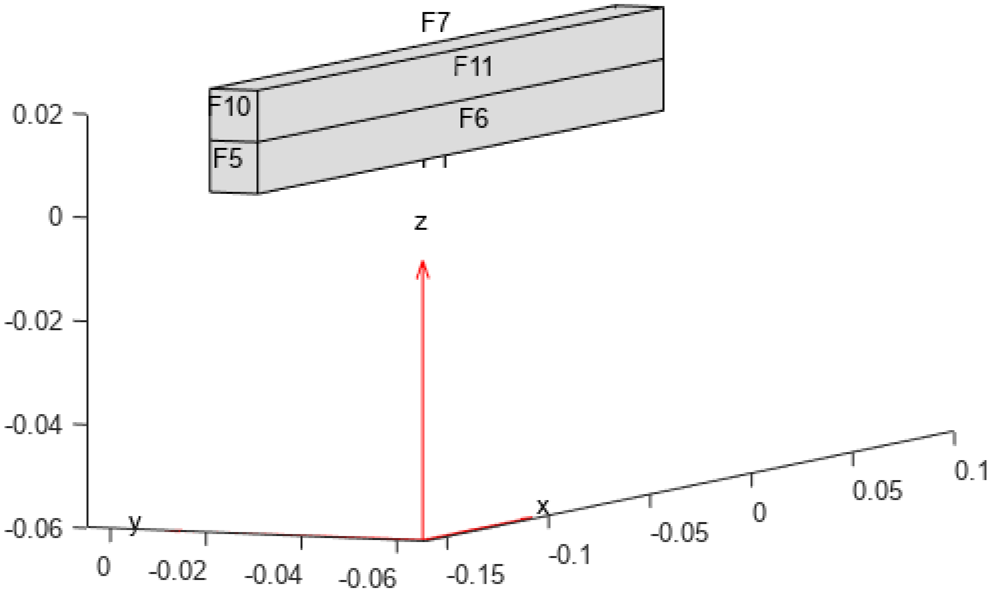
```
gm = multicuboid(0.2,0.01,[0.01 0.01],"Zoffset",[0 0.01]);
```

Include the geometry in the structural model.

```
model.Geometry = gm;
```

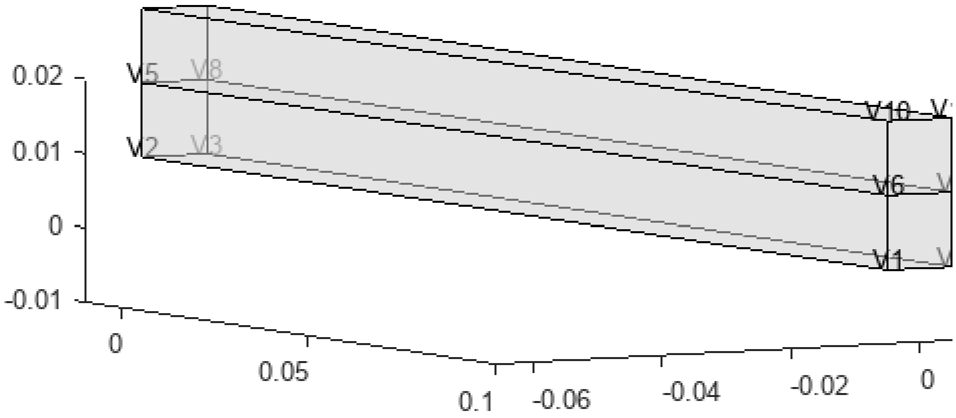
Plot the geometry and display the face labels. Rotate the geometry so that you can see the face labels on the left side.

```
figure
pdegplot(model,"FaceLabels","on");
view([-67 5])
```



Plot the geometry and display the vertex labels. Rotate the geometry so that you can see the vertex labels on the right side.

```
figure
pdegplot(model,"VertexLabels","on","FaceAlpha",0.5)
xlim([-0.01 0.1])
zlim([-0.01 0.02])
view([60 5])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(model,"YoungsModulus",201E9,"PoissonsRatio",0.3);
```

Specify that faces 5 and 10 are fixed boundaries.

```
structuralBC(model,"Face",[5 10],"Constraint","fixed");
```

Specify the concentrated force at vertex 6.

```
structuralBoundaryLoad(model,"Vertex",6,"Force",[0;10^4;0])
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Vertex'
                  RegionID: 6
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []
```

```
   Boundary Loads
                     Force: [3x1 double]
            SurfaceTraction: []
                  Pressure: []
    TranslationalStiffness: []
                     Label: []
```

Specify Pressure for Frequency Response Model

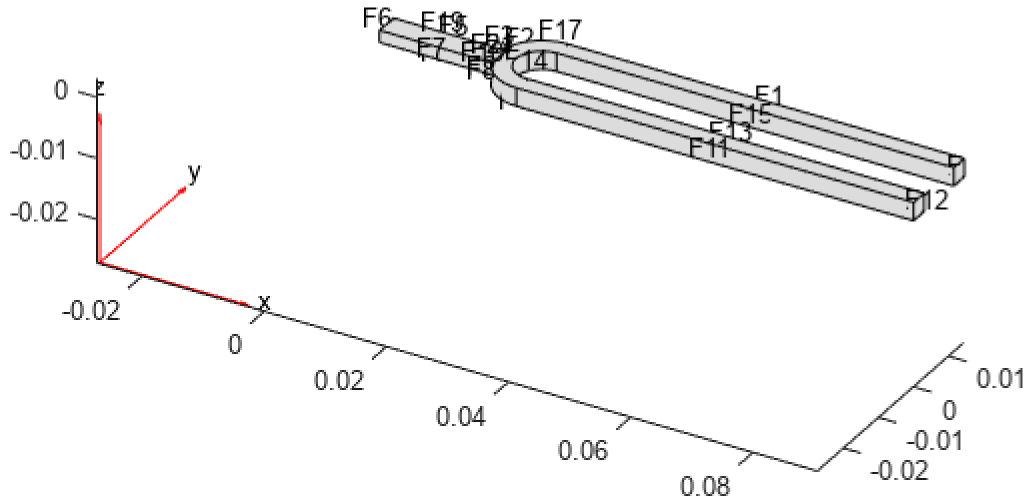
Use a function handle to specify a frequency-dependent pressure for a frequency response model.

Create a frequency response model for a 3-D problem.

```
fmodel = createpde("structural","frequency-solid");
```

Import and plot the geometry.

```
importGeometry(fmodel,"TuningFork.stl");
figure
pdegplot(fmodel,"FaceLabels","on")
```



Specify the pressure loading on a tine (face 11) as a short rectangular pressure pulse. In the frequency domain, this pressure pulse is a unit load uniformly distributed across all frequencies.

```
structuralBoundaryLoad(fmodel,"Face",11,"Pressure",1);
```

Now specify a frequency-dependent pressure load, for example, $p = e^{-(\omega - 1000)^2/100000}$.

```
pLoad = @(location,state) exp(-(state.frequency-1E3).^2/1E5);
structuralBoundaryLoad(fmodel,"Face",12,"Pressure",pLoad);
```

Specify Nonconstant Pressure For Transient Model by Using Function Handle

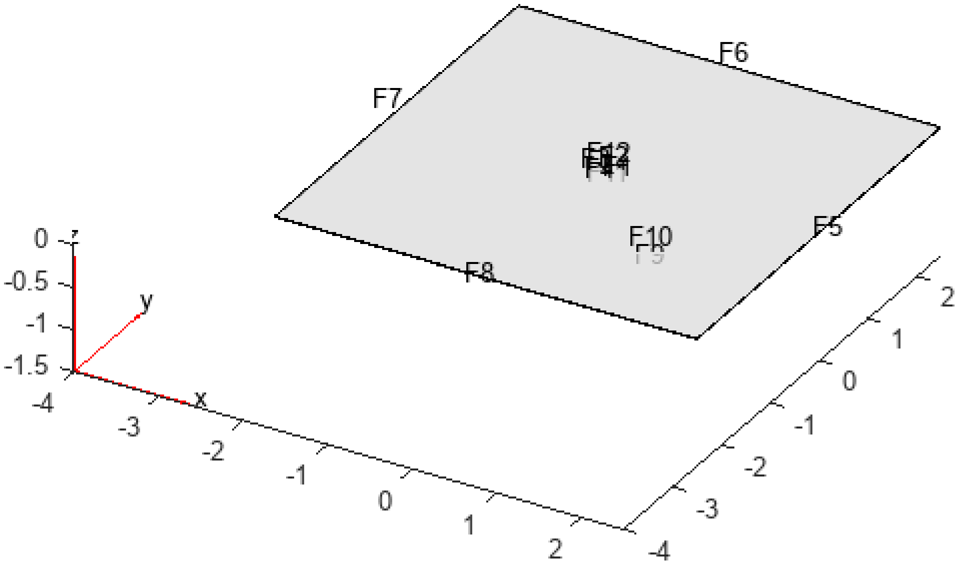
Use a function handle to specify a harmonically varying pressure at the center of a thin 3-D plate.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

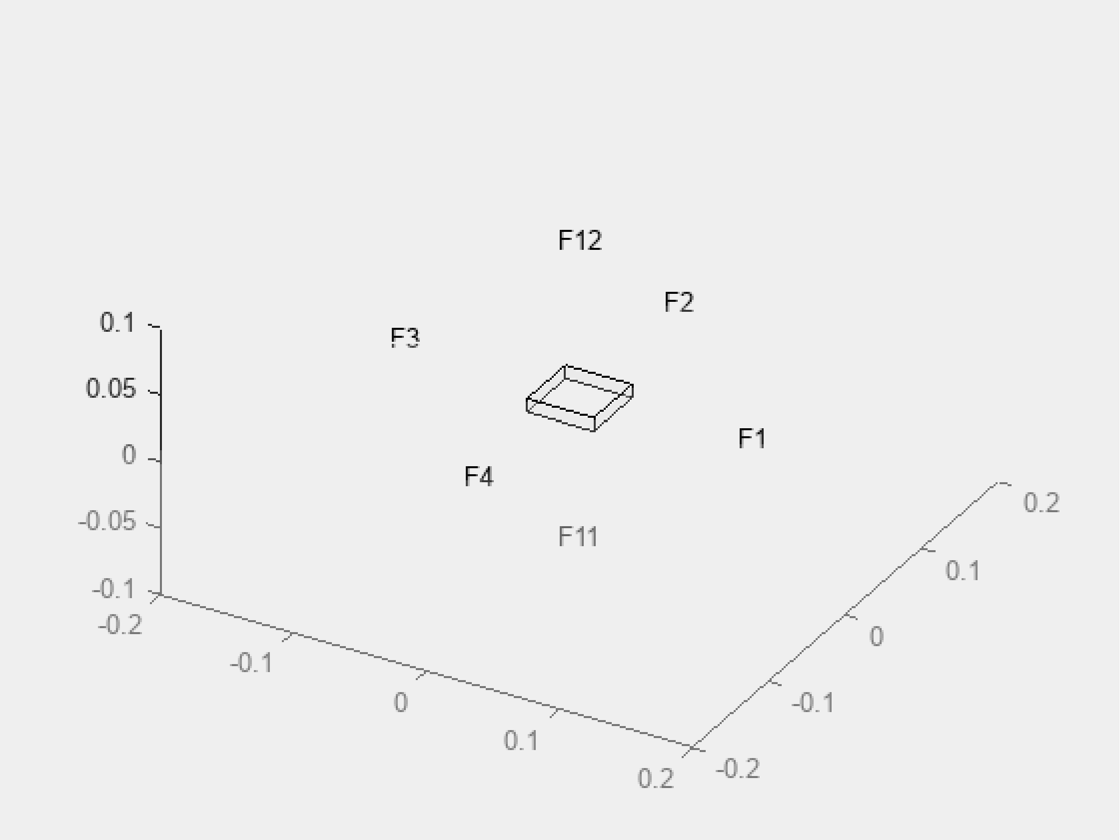
Create a geometry consisting of a thin 3-D plate with a small plate at the center. Include the geometry in the model and plot it.

```
gm = multicuboid([5,0.05],[5,0.05],0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Zoom in to see the face labels on the small plate at the center.

```
figure
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.25)
axis([-0.2 0.2 -0.2 0.2 -0.1 0.1])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9,...
                                     "PoissonsRatio",0.3,...
                                     "MassDensity",7800);
```

Specify that all faces on the periphery of the thin 3-D plate are fixed boundaries.

```
structuralBC(structuralmodel,"Constraint","fixed","Face",5:8);
```

Apply a harmonically varying pressure load on the small face at the center of the plate.

```
plungerLoad = @(location,state)5E7.*sin(25.*state.time);
structuralBoundaryLoad(structuralmodel,"Face",12,"Pressure",plungerLoad)
```

```
ans =
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 12
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
```

```
          XDisplacement: []
          YDisplacement: []
          ZDisplacement: []
             Constraint: []
                 Radius: []
              Reference: []
                  Label: []

Boundary Loads
                  Force: []
        SurfaceTraction: []
               Pressure: @(location,state)5E7.*sin(25.*state.time)
 TranslationalStiffness: []
                  Label: []

Time Variation of Force, Pressure, or Enforced Displacement
              StartTime: []
                EndTime: []
               RiseTime: []
               FallTime: []

Sinusoidal Variation of Force, Pressure, or Enforced Displacement
              Frequency: []
                  Phase: []
```

Apply Sinusoidal Pressure by Specifying Frequency

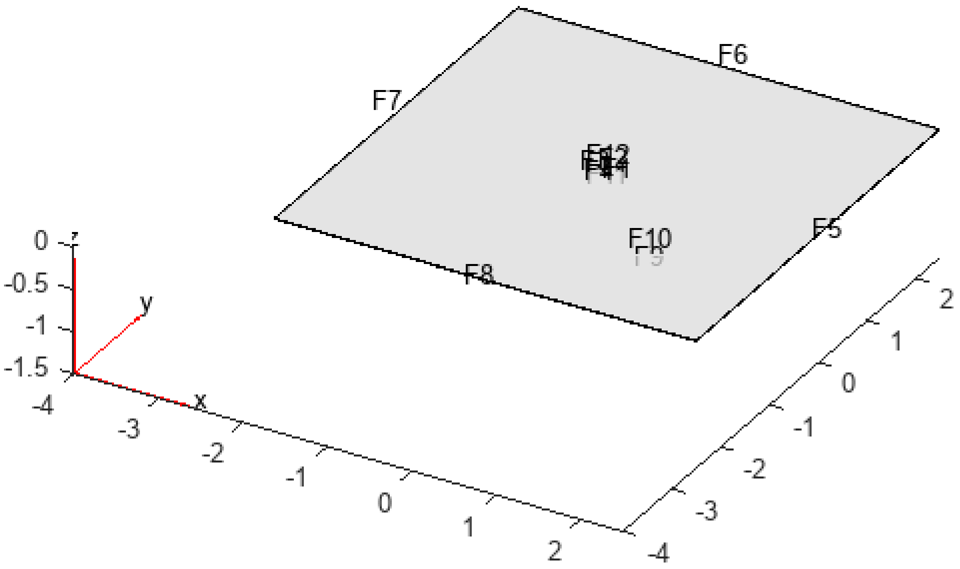
Specify a harmonically varying pressure at the center of a thin 3-D plate by specifying its frequency.

Create a transient dynamic model for a 3-D problem.

```
structuralmodel = createpde("structural","transient-solid");
```

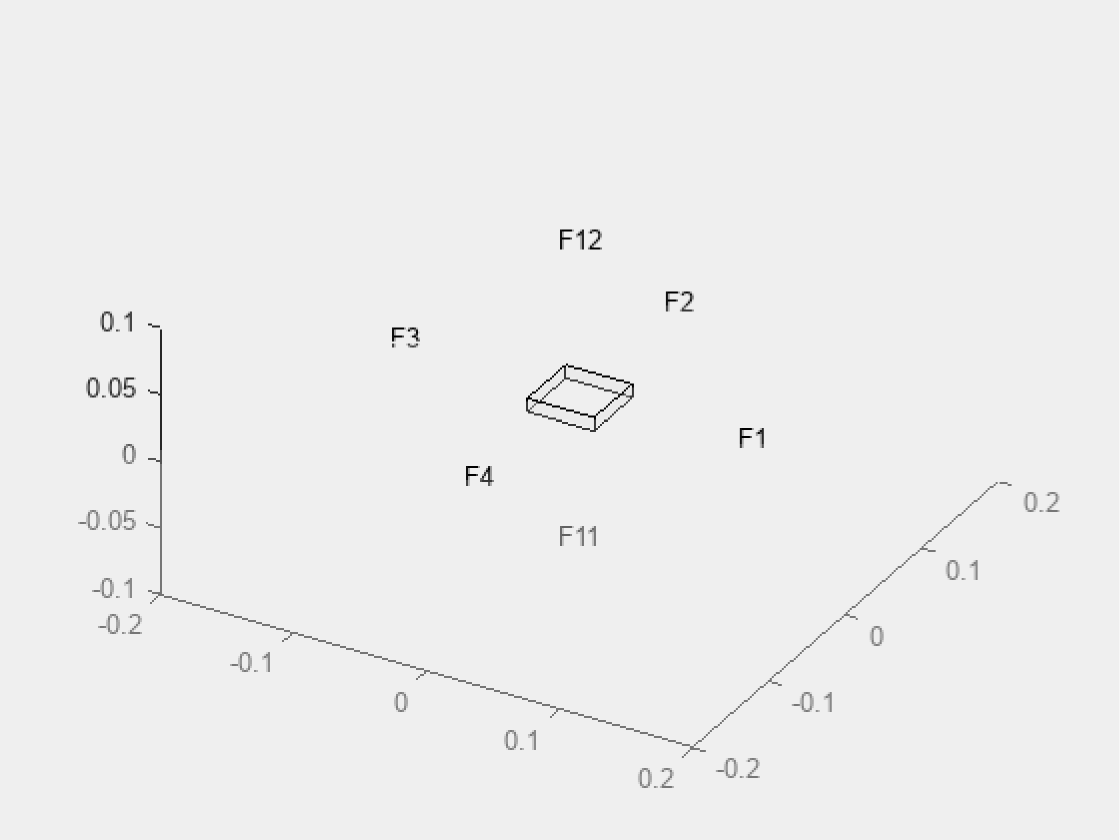
Create a geometry consisting of a thin 3-D plate with a small plate at the center. Include the geometry in the model and plot it.

```
gm = multicuboid([5,0.05],[5,0.05],0.01);
structuralmodel.Geometry=gm;
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.5)
```



Zoom in to see the face labels on the small plate at the center.

```
figure
pdegplot(structuralmodel,"FaceLabels","on","FaceAlpha",0.25)
axis([-0.2 0.2 -0.2 0.2 -0.1 0.1])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",210E9,...
                                     "PoissonsRatio",0.3,...
                                     "MassDensity",7800);
```

Specify that all faces on the periphery of the thin 3-D plate are fixed boundaries.

```
structuralBC(structuralmodel,"Constraint","fixed","Face",5:8);
```

Apply a harmonically varying pressure load on the small face at the center of the plate.

```
structuralBoundaryLoad(structuralmodel,"Face",12, ...
                                       "Pressure",5E7, ...
                                       "Frequency",25)
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 12
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
```

```
                 Constraint: []
                     Radius: []
                  Reference: []
                      Label: []

    Boundary Loads
                      Force: []
            SurfaceTraction: []
                   Pressure: 50000000
     TranslationalStiffness: []
                      Label: []

    Time Variation of Force, Pressure, or Enforced Displacement
                  StartTime: []
                    EndTime: []
                   RiseTime: []
                   FallTime: []

    Sinusoidal Variation of Force, Pressure, or Enforced Displacement
                  Frequency: 25
                      Phase: []
```

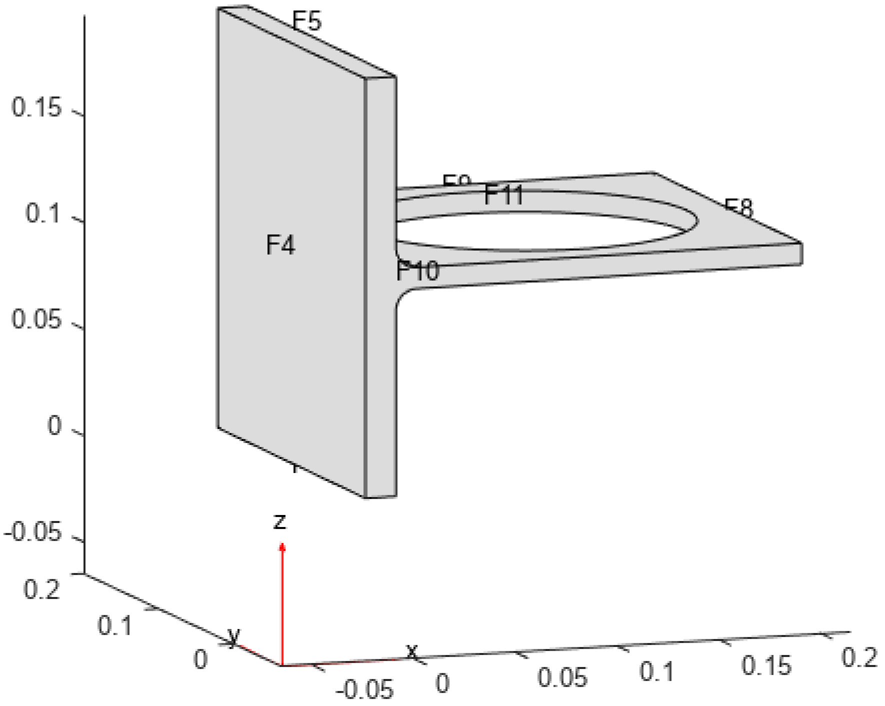
Apply Rectangular Pressure Pulse on Boundary

Create a transient structural model.

```
structuralModel = createpde("structural","transient-solid");
```

Import and plot the geometry.

```
importGeometry(structuralModel,"BracketWithHole.stl");
pdegplot(structuralModel,"FaceLabels","on")
view(-20,10)
```



Specify Young's modulus and Poisson's ratio.

```
structuralProperties(structuralModel,"YoungsModulus",200e9, ...
                                     "PoissonsRatio",0.3,...
                                     "MassDensity",7800);
```

Specify that face 4 is a fixed boundary.

```
structuralBC(structuralModel,"Face",4,"Constraint","fixed");
```

Apply a rectangular pressure pulse on face 7 in the direction normal to the face.

```
structuralBoundaryLoad(structuralModel,"Face",7,"Pressure",10^5,...
                                       "StartTime",0.1,"EndTime",0.5)
```

```
ans =
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 7
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
```

```
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
           SurfaceTraction: []
                  Pressure: 100000
    TranslationalStiffness: []
                     Label: []

   Time Variation of Force, Pressure, or Enforced Displacement
                 StartTime: 0.1000
                   EndTime: 0.5000
                  RiseTime: []
                  FallTime: []

   Sinusoidal Variation of Force, Pressure, or Enforced Displacement
                 Frequency: []
                     Phase: []
```

Apply Rectangular Force Pulse at Point

Specify a short concentrated force pulse at a vertex of a geometry.

Create a structural model for static analysis of a solid (3-D) problem.

```
structuralmodel = createpde("structural","transient-solid");
```

Create the geometry, which consists of two cuboids stacked on top of each other.

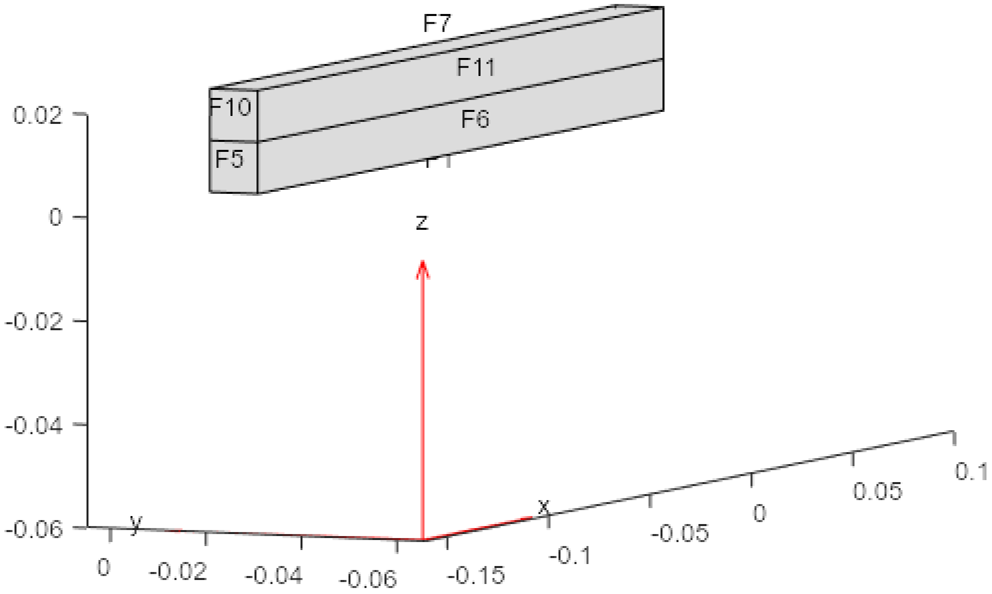
```
gm = multicuboid(0.2,0.01,[0.01 0.01],"Zoffset",[0 0.01]);
```

Include the geometry in the structural model.

```
structuralmodel.Geometry = gm;
```

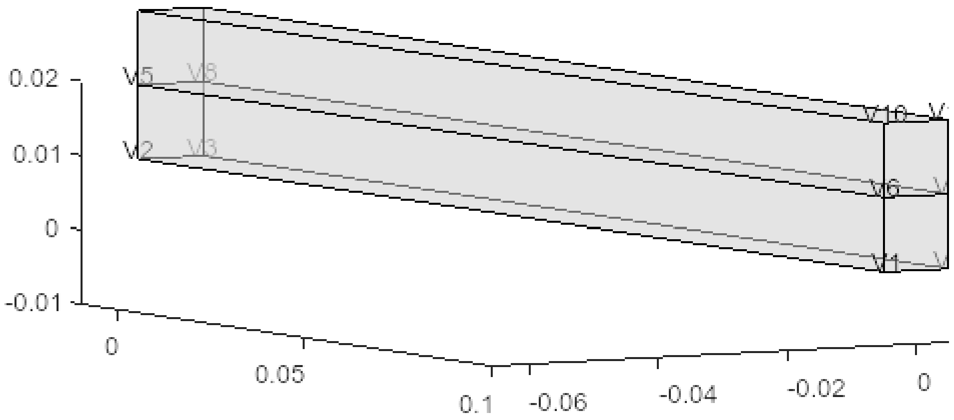
Plot the geometry and display the face labels. Rotate the geometry so that you can see the face labels on the left side.

```
figure
pdegplot(structuralmodel,"FaceLabels","on");
view([-67 5])
```



Plot the geometry and display the vertex labels. Rotate the geometry so that you can see the vertex labels on the right side.

```
figure
pdegplot(structuralmodel,"VertexLabels","on","FaceAlpha",0.5)
xlim([-0.01 0.1])
zlim([-0.01 0.02])
view([60 5])
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",201E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800);
```

Specify that faces 5 and 10 are fixed boundaries.

```
structuralBC(structuralmodel,"Face",[5 10],"Constraint","fixed");
```

Specify a short concentrated force pulse at vertex 6.

```
structuralBoundaryLoad(structuralmodel,"Vertex",6, ...
                                       "Force",[0;1000;0], ...
                                       "StartTime",1, ...
                                       "EndTime",1.05)
```

```
ans =
  StructuralBC with properties:

                RegionType: 'Vertex'
                  RegionID: 6
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
```

```
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: [3×1 double]
           SurfaceTraction: []
                  Pressure: []
    TranslationalStiffness: []
                     Label: []

   Time Variation of Force, Pressure, or Enforced Displacement
                 StartTime: 1
                   EndTime: 1.0500
                  RiseTime: []
                  FallTime: []

   Sinusoidal Variation of Force, Pressure, or Enforced Displacement
                 Frequency: []
                     Phase: []
```

Specify zero initial displacement and velocity.

```
structuralIC(structuralmodel,"Displacement",[0;0;0],"Velocity",[0;0;0])
```

```
ans = 
  GeometricStructuralICs with properties:

             RegionType: 'Cell'
               RegionID: [1 2]
    InitialDisplacement: [3×1 double]
        InitialVelocity: [3×1 double]
```

Generate a fine mesh.

```
generateMesh(structuralmodel,"Hmax",0.02);
```

Because the load is zero for the initial time span and is applied for only a short time, solve the model for two time spans. Use the first time span to find the solution before the force pulse.

```
structuralresults1 = solve(structuralmodel,0:1E-2:1);
```

Use the second time span to find the solution during and after the force pulse.

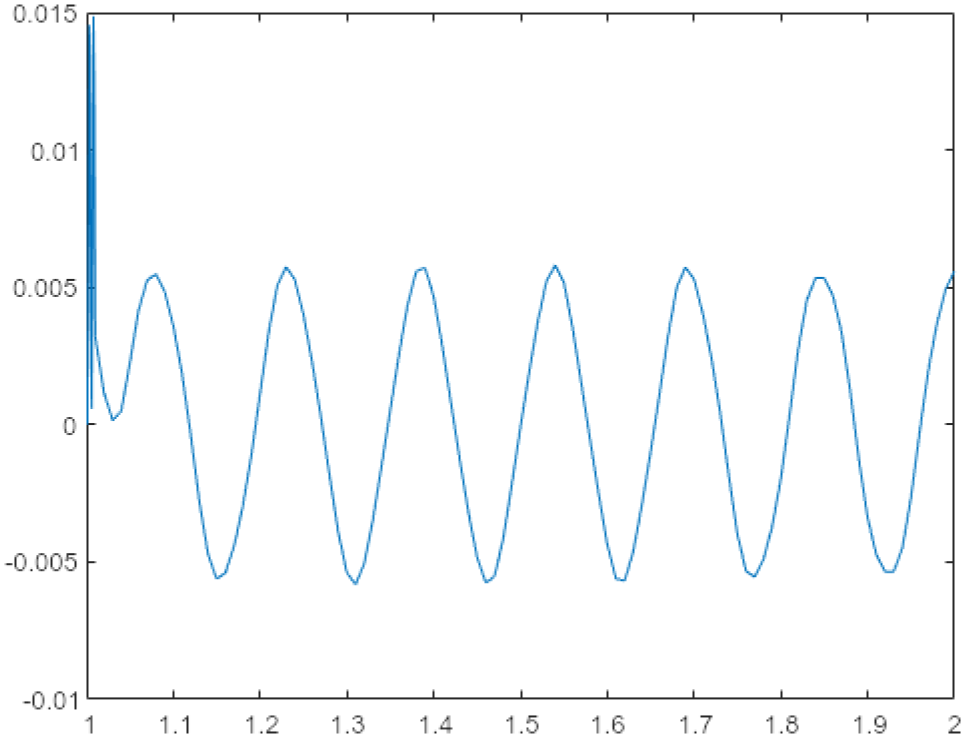
```
structuralIC(structuralmodel,structuralresults1)
```

```
ans = 
  NodalStructuralICs with properties:

    InitialDisplacement: [511×3 double]
        InitialVelocity: [511×3 double]
```

```
structuralresults2 = solve(structuralmodel, ...
                     [1.001:0.001:1.01 1.02:1E-2:2]);
```

Plot the displacement value at the node corresponding to vertex 6, where you applied the concentrated force pulse.

```
loadedNd = findNodes(structuralmodel.Mesh,"region","Vertex",6);
plot(structuralresults2.SolutionTimes, ...
     structuralresults2.Displacement.uy(loadedNd,:))
```



Input Arguments

`structuralmodel` — Structural model

`StructuralModel` object

Structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`RegionType` — Geometric region type

`"Edge"` for a 2-D model | `"Face"` for a 3-D model

Geometric region type, specified as `"Edge"` for a 2-D model or `"Face"` for a 3-D model.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"SurfaceTraction",[0,0,100])`

Data Types: `char` | `string`

`RegionID` — Geometric region ID
positive integer | vector of positive integers

Geometric region ID, specified as a positive integer or vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"SurfaceTraction",[0,0,100])`

Data Types: `double`

`VertexID` — Vertex ID
positive integer | vector of positive integers

Vertex ID, specified as a positive integer or vector of positive integers. Find the vertex IDs using `pdegplot`.

Example: `structuralBoundaryLoad(structuralmodel,"Vertex",6,"Force",[0;10^4;0])`

Data Types: `double`

`STval` — Distributed normal and tangential forces on boundary
numeric vector | function handle

Distributed normal and tangential forces on the boundary, resolved along the global Cartesian coordinate system, specified as a numeric vector or function handle. A numeric vector must contain two elements for a 2-D model and three elements for a 3-D model.

The function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix must correspond to the surface traction vector at the boundary coordinates provided by the solver. In case of a transient or frequency response analysis, `STval` also can be a function of time or frequency, respectively. For details, see “More About” on page 5-1327.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"SurfaceTraction",[0;0;100])`

Data Types: `double` | `function_handle`

`Pval` — Pressure normal to boundary
number | function handle

Pressure normal to the boundary, specified as a number or function handle. A positive-value pressure acts into the boundary (for example, compression), while a negative-value pressure acts away from the boundary (for example, suction).

If you specify `Pval` as a function handle, the function must return a row vector where each column corresponds to the value of pressure at the boundary coordinates provided by the solver. In case of a transient structural model, `Pval` also can be a function of time. In case of a frequency response structural model, `Pval` can be a function of frequency (when specified as a function handle) or a constant pressure with the same magnitude for a broad frequency spectrum. For details, see “More About” on page 5-1327.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5)`

Data Types: `double` | `function_handle`

`TSval` — Distributed spring stiffness
numeric vector | function handle

Distributed spring stiffness for each translational direction used to model elastic foundation, specified as a numeric vector or function handle. A numeric vector must contain two elements for a 2-D model and three elements for a 3-D model. The custom function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of this matrix corresponds to the stiffness vector at the boundary coordinates provided by the solver. In case of a transient or frequency response analysis, `TSval` also can be a function of time or frequency, respectively. For details, see “More About” on page 5-1327.

Example: `structuralBoundaryLoad(structuralmodel,"Edge",[2,5],"TranslationalStiffness",[0;5500])`

Data Types: `double` | `function_handle`

`Fval` — Concentrated force

numeric vector | function handle

Concentrated force at a vertex, specified as a numeric vector or function handle. Use a function handle to specify concentrated force that depends time or frequency. For details, see “More About” on page 5-1327.

Example: `structuralBoundaryLoad(structuralmodel,"Vertex",5,"Force",[0;0;10])`

Data Types: `double` | `function_handle`

`labeltext` — Label for structural boundary load

character vector | string

Label for the structural boundary load, specified as a character vector or a string.

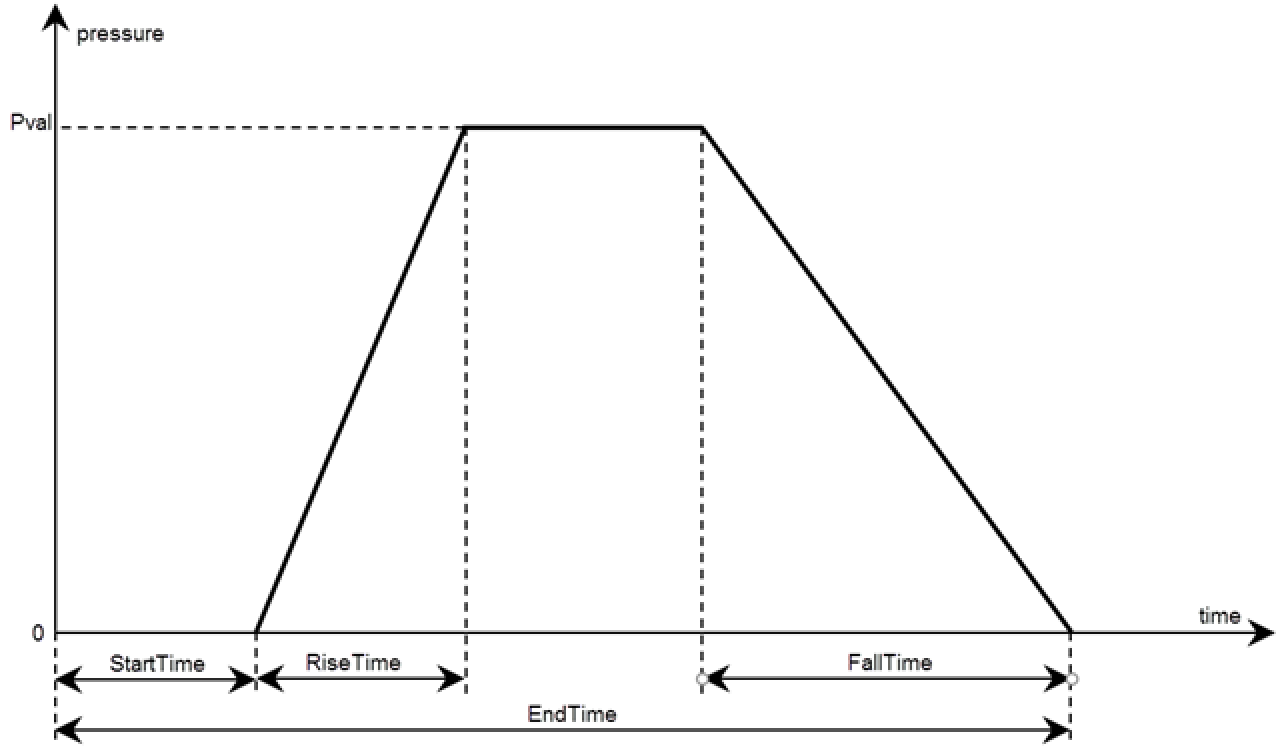
Data Types: `char` | `string`

Name-Value Pair Arguments

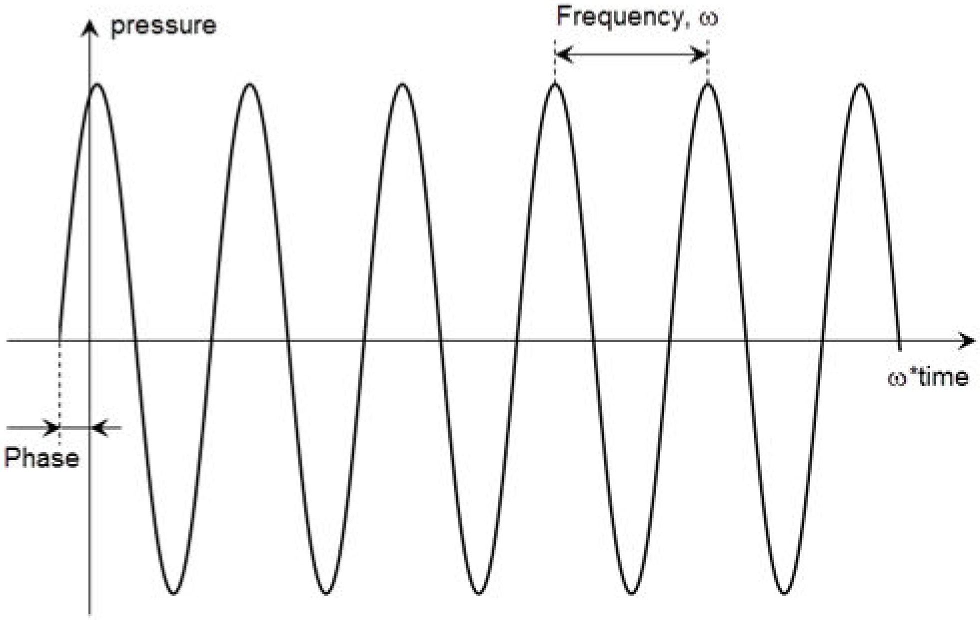
Use one or more of the name-value pair arguments to specify the form and duration of the pressure or concentrated force pulse and harmonic excitation **for a transient structural model only**. Specify the pressure or force value using the `Pval` or `Fval` argument, respectively.

You can model rectangular, triangular, and trapezoidal pressure or concentrated force pulses. If the start time is 0, you can omit specifying it.

- For a rectangular pulse, specify the start and end times.
- For a triangular pulse, specify the start time and any two of the following times: rise time, fall time, and end time. You also can specify all three times, but they must be consistent.
- For a trapezoidal pulse, specify all four times.



You can model a harmonic pressure or concentrated force load by specifying its frequency and initial phase. If the initial phase is 0, you can omit specifying it.



Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5,"RiseTime",0.5,"FallTime",0.5,"EndTime",3)`

Rectangular, Triangular, or Trapezoidal Pulse

`StartTime` — Start time for pressure or concentrated force load

nonnegative number

Start time for pressure or concentrated force load, specified as a nonnegative number. Specify this argument only for transient structural models.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5,"StartTime",1,"EndTime",3)`

Data Types: `double`

`EndTime` — End time for pressure or concentrated force load

nonnegative number

End time for pressure or concentrated force load, specified as a nonnegative number equal or greater than the start time value. Specify this argument only for transient structural models.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5,"StartTime",1,"EndTime",3)`

Data Types: `double`

`RiseTime` — Rise time for pressure or concentrated force load

nonnegative number

Rise time for pressure or concentrated force load, specified as a nonnegative number. Specify this argument only for transient structural models.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5,"RiseTime",0.5,"FallTime",0.5,"EndTime",3)`

Data Types: `double`

`FallTime` — Fall time for pressure or concentrated force load

nonnegative number

Fall time for pressure or concentrated force load, specified as a nonnegative number. Specify this argument only for transient structural models.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5,"RiseTime",0.5,"FallTime",0.5,"EndTime",3)`

Data Types: `double`

Harmonic Pressure or Force

`Frequency` — Frequency of sinusoidal pressure or concentrated force

positive number

Frequency of sinusoidal pressure or concentrated force, specified as a positive number, in radians per unit of time. Specify this argument only for transient structural models.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5,"Frequency",25)`

Data Types: `double`

Phase — Phase of sinusoidal pressure or concentrated force
nonnegative number

Phase of sinusoidal pressure or concentrated force, specified as a nonnegative number, in radians. Specify this argument only for transient structural models.

Example: `structuralBoundaryLoad(structuralmodel,"Face",[2,5],"Pressure",10^5,"Frequency",25,"Phase",pi/6)`

Data Types: `double`

Output Arguments

`boundaryLoad` — Handle to boundary load
`StructuralBC` object

Handle to boundary load, returned as a `StructuralBC` object. See StructuralBC Properties.

More About

Specifying Nonconstant Parameters of a Structural Model

Use a function handle to specify the following structural parameters when they depend on space and, depending of the type of structural analysis, either time or frequency:

- Surface traction on the boundary
- Pressure normal to the boundary
- Concentrated force at a vertex
- Distributed spring stiffness for each translational direction used to model elastic foundation
- Enforced displacement and its components
- Initial displacement and velocity (can depend on space only)

For example, use function handles to specify the pressure load, *x*-component of the enforced displacement, and the initial displacement for this model.

```
structuralBoundaryLoad(model,"Face",12, ...
                             "Pressure",@myfunPressure)
structuralBC(model,"Face",2, ...
                   "XDisplacement",@myfunBC)
structuralIC(model,"Face",12, ...
                   "Displacement",@myfunIC)
```

For all parameters, except the initial displacement and velocity, the function must be of the form:

```
function structuralVal = myfun(location,state)
```

For the initial displacement and velocity the function must be of the form:

```
function structuralVal = myfun(location)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments (or one argument if the function specifies the initial displacement or initial velocity).

- `location` — A structure containing these fields:
  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes these data in the `location` structure:

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, *r*-component of the normal vector at the evaluation point or points
- `state` — A structure containing these fields for dynamic structural problems:
  - `state.time` contains the time at evaluation points.
  - `state.frequency` contains the frequency at evaluation points.

  `state.time` and `state.frequency` are scalars.

Boundary constraints and loads get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`
- `state.time` or `state.frequency` (depending of the type of structural analysis)

Initial conditions get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

If a parameter represents a vector value, such as surface traction, spring stiffness, force, or displacement, your function must return a two-row matrix for a 2-D model and a three-row matrix for a 3-D model. Each column of the matrix corresponds to the parameter value (a vector) at the boundary coordinates provided by the solver.

If a parameter represents a scalar value, such as pressure or a displacement component, your function must return a row vector where each element corresponds to the parameter value (a scalar) at the boundary coordinates provided by the solver.

If boundary conditions depend on `state.time` or `state.frequency`, ensure that your function returns a matrix of `NaN` of the correct size when `state.frequency` 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.

Additional Arguments in Functions for Nonconstant Structural Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
structuralVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
structuralBC(model,"Face",2,"XDisplacement",structuralVal)

structuralVal = ...
@(location) myfunWithAdditionalArgs(location,arg1,arg2...)
structuralIC(model,"Face",2,"Displacement",structuralVal)
```

Version History

Introduced in R2017b

Label to extract sparse linear models for use with Control System Toolbox

Now you can add a label for structural boundary loads to be used by the `linearizeInput` function. This function lets you pass structural boundary loads to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

Concentrated boundary loads at arbitrary locations on geometry surfaces

You can now use `addVertex` to create new vertices at any points on boundaries of a 2-D or 3-D geometry represented by a `DiscreteGeometry` object. Then set concentrated boundary loads at these vertices.

Concentrated force at a vertex

You can now specify concentrated force at a vertex.

Time-dependent boundary loads

You can now specify time-dependent boundary loads by using function handles or specify the form and duration of the pressure pulse and the frequency and phase of sinusoidal pressure.

See Also

`StructuralModel` | `structuralProperties` | `structuralDamping` | `structuralBodyLoad` | `structuralBC` | StructuralBC Properties

StructuralModel

Structural model object

Description

A `StructuralModel` object contains information about a structural analysis problem: the geometry, material properties, damping parameters, body loads, boundary loads, boundary constraints, superelement interfaces, initial displacement and velocity, and mesh.

Creation

To create a `StructuralModel` object, use `createpde` and specify `'structural'` as its first argument .

Properties

`AnalysisType` — Type of structural analysis
`'static-solid'` | `'static-planestress'` | `'static-planestrain'` | `'static-axisymmetric'` | `'transient-solid'` | `'transient-planestress'` | `'transient-planestrain'` | `'transient-axisymmetric'` | `'modal-solid'` | `'modal-planestress'` | `'modal-planestrain'` | `'modal-axisymmetric'` | `'frequency-solid'` | `'frequency-planestress'` | `'frequency-planestrain'` | `'frequency-axisymmetric'`

Type of structural analysis, specified as one of these values.

Static analysis:

- `'static-solid'` for static structural analysis of a solid (3-D) problem
- `'static-planestress'` for static structural analysis of a plane-stress problem
- `'static-planestrain'` for static structural analysis of a plane-strain problem
- `'static-axisymmetric'` for static structural analysis of an axisymmetric (2-D) problem

Transient analysis:

- `'transient-solid'` for transient structural analysis of a solid (3-D) problem
- `'transient-planestress'` for transient structural analysis of a plane-stress problem
- `'transient-planestrain'` for transient structural analysis of a plane-strain problem
- `'transient-axisymmetric'` for transient structural analysis of an axisymmetric (2-D) problem

Modal analysis:

- `'modal-solid'` for modal analysis of a solid (3-D) problem
- `'modal-planestress'` for modal analysis of a plane-stress problem
- `'modal-planestrain'` for modal analysis of a plane-strain problem

- `'modal-axisymmetric'` for modal analysis of an axisymmetric (2-D) problem

Frequency response analysis:

- `'frequency-solid'` for frequency response analysis of a solid (3-D) problem
- `'frequency-planestress'` for frequency response analysis of a plane-stress problem
- `'frequency-planestrain'` for frequency response analysis of a plane-strain problem
- `'frequency-axisymmetric'` for frequency response analysis of an axisymmetric (2-D) problem

To change a structural analysis type, assign a new type to `model.AnalysisType`. Ensure that all other properties of the model are consistent with the new analysis type. Note that you cannot change the spatial dimensionality. For example, you can change the analysis type from `'static-solid'` to `'modal-solid'`, but cannot change it to `'static-planestress'`.

Example: `model = createpde('structural','static-solid')`

Data Types: `char`

`Geometry` — Geometry description
`AnalyticGeometry` | `DiscreteGeometry`

Geometry description, specified as `AnalyticGeometry` for a 2-D geometry or `DiscreteGeometry` for a 2-D or 3-D geometry.

`MaterialProperties` — Material properties
`StructuralMaterialAssignment` object containing material property assignments

Material properties within the domain, specified as a `StructuralMaterialAssignment` object containing the material property assignments. For details, see StructuralMaterialAssignment Properties.

To create the material properties assignments for your structural analysis model, use the `structuralProperties` function.

`BodyLoads` — Loads acting on domain or subdomain
`BodyLoadAssignment` object containing body load assignments

Loads acting on the domain or subdomain, specified as a `BodyLoadAssignment` object containing body load assignments. For details, see BodyLoadAssignment Properties.

To create body load assignments for your structural analysis model, use the `structuralBodyLoad` function.

`BoundaryConditions` — Structural loads and boundary conditions
`StructuralBC` object containing boundary condition assignments

Structural loads and boundary conditions applied to the geometry, specified as a `StructuralBC` object containing the boundary condition assignments. For details, see StructuralBC Properties.

To specify boundary conditions for your model, use the `structuralBC` function. To specify boundary loads, use `structuralBoundaryLoad`.

`DampingModels` — Damping model for transient or frequency response analysis
`StructuralDampingAssignment` object containing damping assignments

Damping model for transient or frequency response analysis, specified as a `StructuralDampingAssignment` object containing damping assignments. For details, see StructuralDampingAssignment Properties.

To set damping parameters for your structural model, use the `structuralDamping` function.

`ReferenceTemperature` — Reference temperature for thermal load
0 (default) | number

Reference temperature for a thermal load, specified as a number. The reference temperature corresponds to state of zero thermal stress of the model. The default value 0 implies that the thermal load is specified in terms of the temperature change and its derivatives.

To specify the reference temperature for a thermal load in your static structural model, assign the property value directly, for example, `structuralmodel.ReferenceTemperature = 10`. To specify the thermal load itself, use the `structuralBodyLoad` function.

Data Types: `double`

`InitialConditions` — Initial displacement and velocity
`GeometricStructuralICs` object | `NodalStructuralICs` object

Initial displacement and velocity, specified as a `GeometricStructuralICs` or `NodalStructuralICs` object. For details, see GeometricStructuralICs Properties and NodalStructuralICs Properties.

To set initial conditions for your transient structural model, use the `structuralIC` function.

`SuperelementInterfaces` — Superelement interfaces for component mode synthesis
`StructuralSEIAssignment` object containing superelement interfaces assignments

Superelement interfaces for the component mode synthesis, specified as a `StructuralSEIAssignment` object containing superelement interface assignments. For details, see StructuralSEIAssignment Properties.

To specify superelement interfaces for your frequency response structural model, use the `structuralSEInterface` function.

`Mesh` — Mesh for solution
`FEMesh` object

Mesh for solution, specified as a `FEMesh` object. For property details, see FEMesh.

To create the mesh, use the `generateMesh` function.

`LinearizeInputs` — Inputs for linearized model
structure array

Inputs for a linearized model, specified as a structure array. The inputs are used by the `linearize` that extracts `mechss` model from a structural model.

`LinearizeOutputs` — Outputs for linearized model
structure array

Inputs for a linearized model, specified as a structure array. The outputs are used by the `linearize` that extracts `mechss` model from a structural model.

`SolverOptions` — Algorithm options for PDE solvers
`PDESolverOptions` object

Algorithm options for the PDE solvers, specified 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 from decomposed geometry matrix
geometryFromMesh	Create 2-D or 3-D geometry from mesh
importGeometry	Import geometry from STL or STEP file
structuralBC	Specify boundary conditions for structural model
structuralSEInterface	Specify structural superelement interface for component mode synthesis
structuralBodyLoad	Specify body load for structural model
structuralBoundaryLoad	Specify boundary loads for structural model
structuralIC	Set initial conditions for a transient structural model
structuralProperties	Assign structural properties of material for structural model
solve	Solve heat transfer, structural analysis, or electromagnetic analysis problem
reduce	Reduce structural or thermal model

Examples

Create and Populate Structural Analysis Model

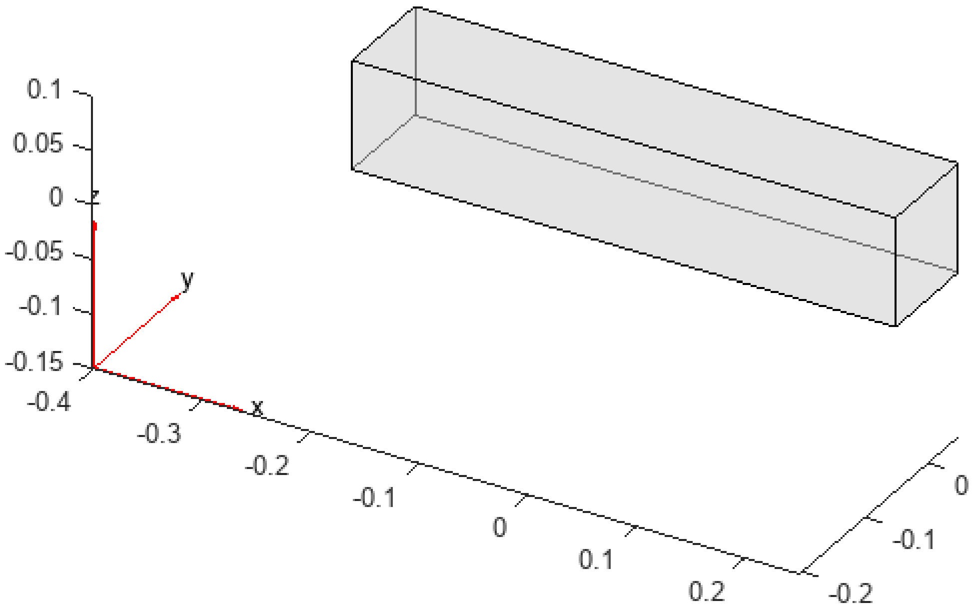
Create a static structural model for solving a solid (3-D) problem.

```
structuralModel = createpde("structural","static-solid")
```

```
structuralModel = 
  StructuralModel with properties:

              AnalysisType: "static-solid"
                  Geometry: []
        MaterialProperties: []
                 BodyLoads: []
        BoundaryConditions: []
      ReferenceTemperature: []
    SuperelementInterfaces: []
                      Mesh: []
             SolverOptions: [1x1 pde.PDESolverOptions]
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.1,0.1);
structuralModel.Geometry = gm;
pdegplot(structuralModel,"FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(structuralModel,"Cell",1,"YoungsModulus",210E3, ...
                                              "PoissonsRatio",0.3, ...
                                              "MassDensity",2.7E-6)
```

```
ans = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 1
        YoungsModulus: 210000
        PoissonsRatio: 0.3000
          MassDensity: 2.7000e-06
                  CTE: []
    HystereticDamping: []
```

Specify the gravity load on the rod.

```
structuralBodyLoad(structuralModel, ...
    "GravitationalAcceleration",[0;0;-9.8])
```

```
ans = 
  BodyLoadAssignment with properties:

                   RegionType: 'Cell'
                     RegionID: 1
```

```
    GravitationalAcceleration: [3x1 double]
              AngularVelocity: []
                  Temperature: []
                     TimeStep: []
                        Label: []
```

Specify that face 6 is a fixed boundary.

```
structuralBC(structuralModel,"Face",6,"Constraint","fixed")
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 6
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: "fixed"
                    Radius: []
                 Reference: []
                     Label: []

   Boundary Loads
                     Force: []
           SurfaceTraction: []
                  Pressure: []
    TranslationalStiffness: []
                     Label: []
```

Specify the surface traction for face 5.

```
structuralBoundaryLoad(structuralModel, ...
                       "Face",5, ...
                       "SurfaceTraction",[0;0;100])
```

```
ans = 
  StructuralBC with properties:

                RegionType: 'Face'
                  RegionID: 5
                Vectorized: 'off'

   Boundary Constraints and Enforced Displacements
              Displacement: []
             XDisplacement: []
             YDisplacement: []
             ZDisplacement: []
                Constraint: []
                    Radius: []
                 Reference: []
                     Label: []
```

```
   Boundary Loads
                     Force: []
           SurfaceTraction: [3x1 double]
                  Pressure: []
    TranslationalStiffness: []
                     Label: []
```

Generate a mesh.

```
generateMesh(structuralModel)
```

```
ans = 
  FEMesh with properties:

             Nodes: [3x7800 double]
          Elements: [10x4857 double]
    MaxElementSize: 0.0208
    MinElementSize: 0.0104
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

View the properties of `structuralModel`.

```
structuralModel
```

```
structuralModel = 
  StructuralModel with properties:

              AnalysisType: "static-solid"
                  Geometry: [1x1 DiscreteGeometry]
        MaterialProperties: [1x1 StructuralMaterialAssignmentRecords]
                 BodyLoads: [1x1 BodyLoadAssignmentRecords]
        BoundaryConditions: [1x1 StructuralBCRecords]
      ReferenceTemperature: []
    SuperelementInterfaces: []
                      Mesh: [1x1 FEMesh]
             SolverOptions: [1x1 pde.PDESolverOptions]
```

Version History

Introduced in R2017b

Hysteretic damping

You can now specify hysteretic damping of a material for direct and modal frequency response models.

Sparse linear models for use with Control System Toolbox

`StructuralModel` now has additional properties, `LinearizeInputs` and `LinearizeOutputs` containing inputs and outputs for linearized model. They are used by the `linearize` that extracts a `mechss` model from a structural model.

Axisymmetric analysis

You can now specify `AnalysisType` for axisymmetric models. Axisymmetric analysis simplifies 3-D structural problems to 2-D using their symmetry around the axis of rotation.

Reduced-order modeling for structural analysis

The toolbox now supports the frequency response analysis for structural models.

Frequency response analysis

You can now specify `AnalysisType` for frequency response analysis, including `'frequency-solid'`, `'frequency-planestress'`, `'frequency-planestrain'`, and `'frequency-axisymmetric'`.

Reference temperature for thermal stress

The programmatic workflow for static structural analysis problems now enables you to account for thermal loading. When setting up a static structural problem, you can specify a reference temperature as a property of `StructuralModel`. This temperature corresponds to the state of the model at which both thermal stress and strain are zeros.

Modal analysis

The programmatic workflow for modal analysis problems now enables you to find natural frequencies and mode shapes of a structure. When solving a modal analysis model, the solver requires a frequency range parameter and returns the modal solution in that frequency range. The `AnalysisType` values for modal analysis are `'modal-solid'`, `'modal-planestress'`, `'modal-planestrain'`, and `'modal-axisymmetric'`.

Transient analysis

The programmatic workflow for structural analysis problems now enables you to set up, solve, and analyze dynamic linear elasticity problems. The `AnalysisType` values for transient analysis are `'transient-solid'`, `'transient-planestress'`, `'transient-planestrain'`, and `'transient-axisymmetric'`.

See Also

`assembleFEMatrices` | `createpde` | `generateMesh` | `geometryFromEdges` | `geometryFromMesh` | `importGeometry` | `pdegplot` | `pdeplot` | `pdeplot3D` | `solve` | `structuralBC` | `structuralBodyLoad` | `structuralBoundaryLoad` | `structuralProperties` | `structuralSEInterface` | `reduce`

structuralProperties

Package: `pde`

Assign structural properties of material for structural model

Syntax

```
structuralProperties(structuralmodel,"YoungsModulus",YMval,"PoissonsRatio",
PRval)
structuralProperties( ___ ,"MassDensity",MDval)
structuralProperties( ___ ,"CTE",CTEval)
structuralProperties( ___ ,"HystereticDamping",g)
structuralProperties( ___ ,RegionType,RegionID)
mtl = structuralProperties( ___ )
```

Description

`structuralProperties(structuralmodel,"YoungsModulus",YMval,"PoissonsRatio",PRval)` assigns Young's modulus and Poisson's ratio for the entire geometry. Use this syntax if your model is static and does not account for gravitational and thermal effects.

Tip A structural model supports only homogeneous isotropic materials. Therefore, all material properties must be numeric scalars.

`structuralProperties( ___ ,"MassDensity",MDval)` assigns the mass density of the material for the entire geometry, and can include any of the arguments used in the previous syntax. Specify the mass density of the material if your model is transient or modal, or if it accounts for gravitational effects.

`structuralProperties( ___ ,"CTE",CTEval)` assigns the coefficient of thermal expansion for a thermal stress analysis. Use this syntax if your model is static and accounts for thermal effects.

`structuralProperties( ___ ,"HystereticDamping",g)` assigns the hysteretic damping parameter that models damping forces counteracting velocity but independent of frequency. Use this syntax for direct and modal frequency response analyses.

`structuralProperties( ___ ,RegionType,RegionID)` assigns material properties for the specified geometry region.

`mtl = structuralProperties( ___ )` returns the material properties object.

Examples

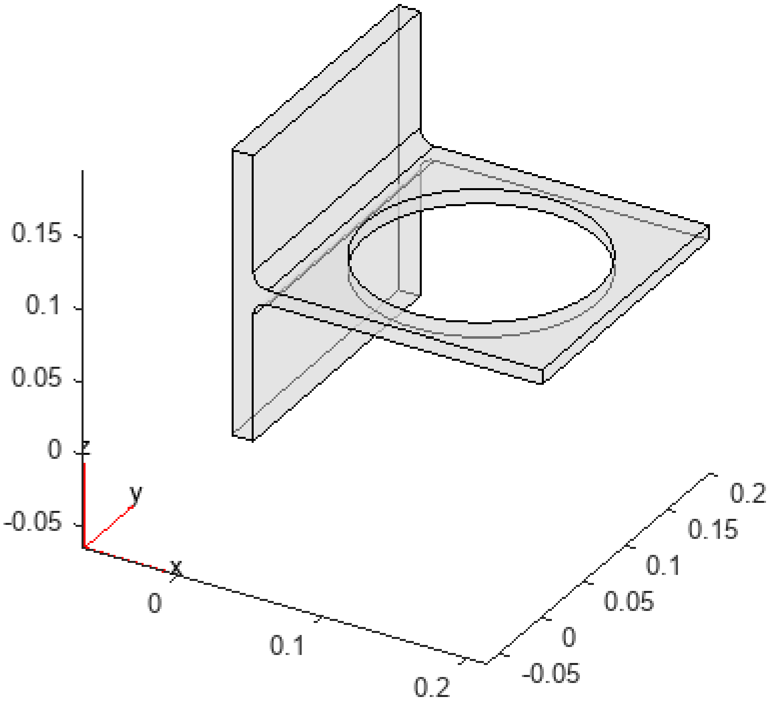
Structural Material Properties for Static Model Accounting for Gravity

Create a structural model.

```
structuralModel = createpde("structural","static-solid");
```

Import and plot the geometry.

```
importGeometry(structuralModel,"BracketWithHole.stl");
pdegplot(structuralModel,"FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(structuralModel,"YoungsModulus",200e9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800)
```

```
ans = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 1
        YoungsModulus: 2.0000e+11
        PoissonsRatio: 0.3000
          MassDensity: 7800
                  CTE: []
    HystereticDamping: []
```

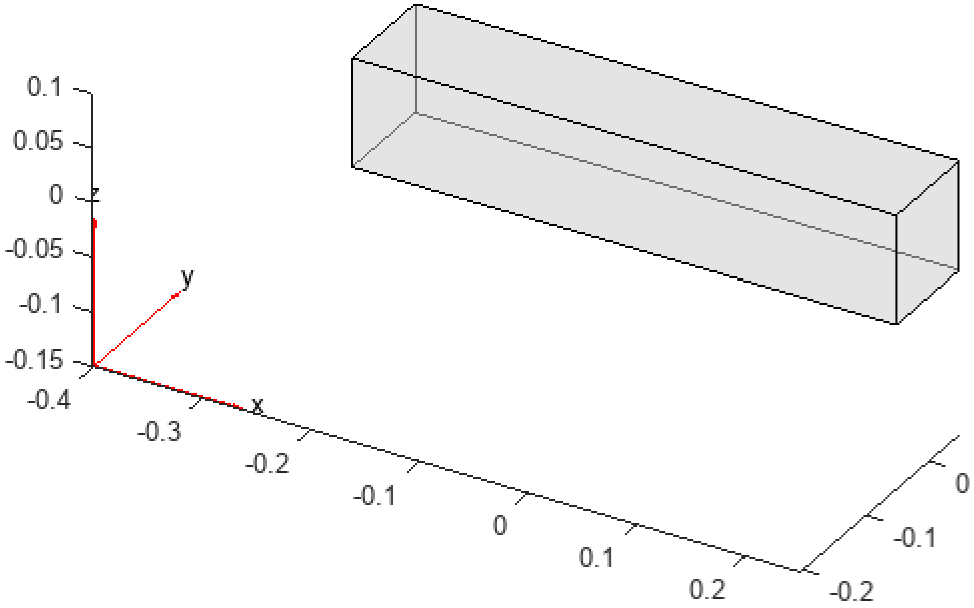
Structural Material Properties for Modal Analysis

Create a structural model for modal analysis.

```
structuralModel = createpde("structural","modal-solid");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.1,0.1);
structuralModel.Geometry = gm;
pdegplot(structuralModel,"FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density.

```
structuralProperties(structuralModel,"YoungsModulus",210E3, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",2.7E-6)
```

```
ans = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 1
        YoungsModulus: 210000
        PoissonsRatio: 0.3000
          MassDensity: 2.7000e-06
                  CTE: []
    HystereticDamping: []
```

Structural Material Properties for Thermal Stress Analysis

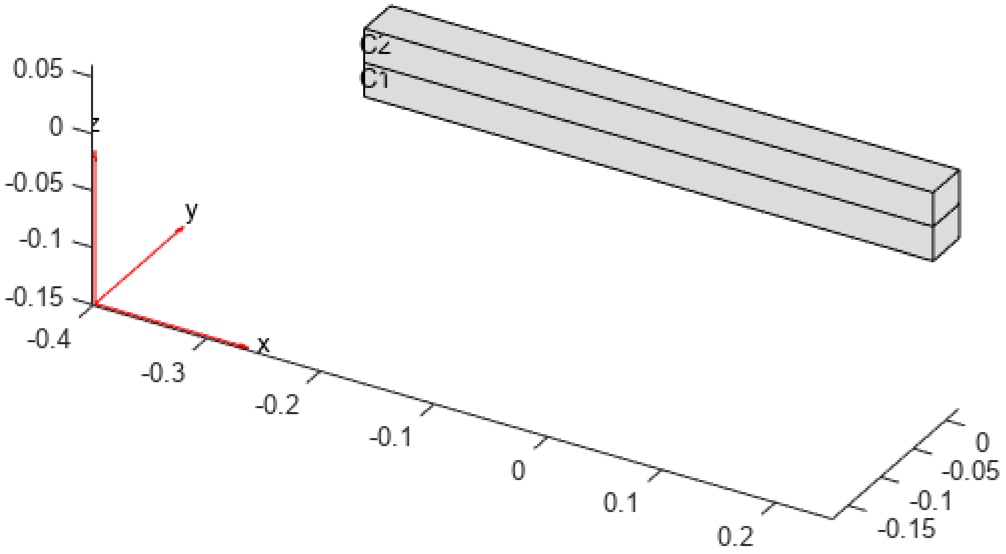
Specify the coefficients of thermal expansion for a bimetallic cantilever beam. The bottom layer is steel. The top layer is copper.

Create a static structural model.

```
structuralmodel = createpde("structural","static-solid");
```

Create and plot the geometry.

```
gm = multicuboid(0.5,0.04,[0.03,0.03],"Zoffset",[0,0.03]);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"CellLabels","on")
```



Specify Young's modulus, Poisson's ratio, and the coefficient of thermal expansion for the bottom cell C1.

```
structuralProperties(structuralmodel,"Cell",1, ...
                                    "YoungsModulus",210e9, ...
                                    "PoissonsRatio",0.28, ...
                                    "CTE",1.3e-5)
```

```
ans = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 1
```

```
         YoungsModulus: 2.1000e+11
         PoissonsRatio: 0.2800
           MassDensity: []
                   CTE: 1.3000e-05
     HystereticDamping: []
```

Specify Young's modulus, Poisson's ratio, and the coefficient of thermal expansion for the top cell C2.

```
structuralProperties(structuralmodel,"Cell",2, ...
                                     "YoungsModulus",110e9, ...
                                     "PoissonsRatio",0.37, ...
                                     "CTE",2.4e-5)
```

```
ans = 
  StructuralMaterialAssignment with properties:

            RegionType: 'Cell'
              RegionID: 2
         YoungsModulus: 1.1000e+11
         PoissonsRatio: 0.3700
           MassDensity: []
                   CTE: 2.4000e-05
     HystereticDamping: []
```

Hysteretic Damping

Create a frequency response analysis model for a 3-D problem.

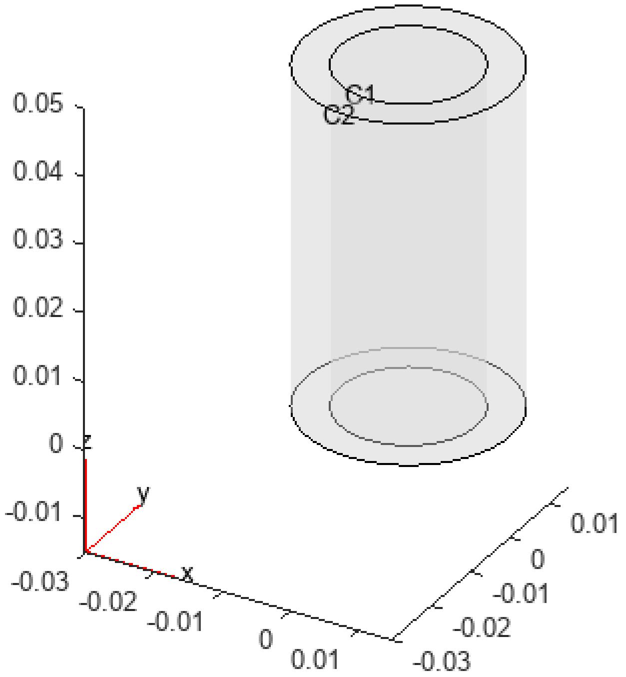
```
structuralmodel = createpde("structural","frequency-solid");
```

Create nested cylinders to model a bimetallic cable.

```
gm = multicylinder([0.01,0.015],0.05);
```

Assign the geometry to the structural model and plot the geometry.

```
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"CellLabels","on","FaceAlpha",0.4)
```



Specify Young's modulus, Poisson's ratio, mass density, and hysteretic damping for both cylinders.

```
structuralProperties(structuralmodel,"YoungsModulus",110E9, ...
                                     "PoissonsRatio",0.28, ...
                                     "MassDensity",7800, ...
                                     "HystereticDamping",0.2, ...
                                     "Cell",1)
```

```
ans = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 1
        YoungsModulus: 1.1000e+11
        PoissonsRatio: 0.2800
          MassDensity: 7800
                  CTE: []
    HystereticDamping: 0.2000
```

```
structuralProperties(structuralmodel,"YoungsModulus",210E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",7800, ...
                                     "HystereticDamping",0.1, ...
                                     "Cell",2)
```

```
ans = 
  StructuralMaterialAssignment with properties:
```

```
            RegionType: 'Cell'
              RegionID: 2
          YoungsModulus: 2.1000e+11
          PoissonsRatio: 0.3000
            MassDensity: 7800
                    CTE: []
      HystereticDamping: 0.1000
```

Structural Material Properties for Each Geometric Region

Create a structural model.

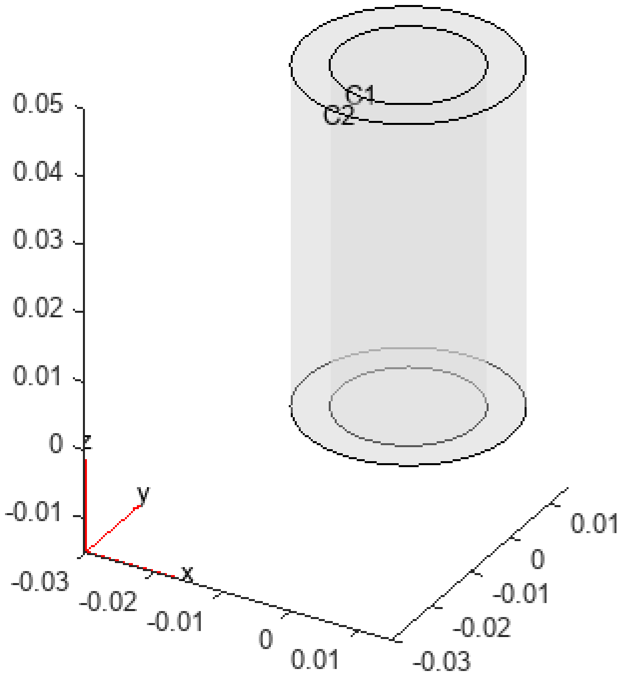
```
structuralModel = createpde("structural","static-solid");
```

Create nested cylinders to model a bimetallic cable.

```
gm = multicylinder([0.01,0.015],0.05);
```

Assign the geometry to the structural model and plot the geometry.

```
structuralModel.Geometry = gm;
pdegplot(structuralModel,"CellLabels","on","FaceAlpha",0.4)
```



Specify Young's modulus and Poisson's ratio for each metal.

```
structuralProperties(structuralModel,"Cell",1,"YoungsModulus",110E9, ...
                                              "PoissonsRatio",0.28)
```

```
ans = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 1
        YoungsModulus: 1.1000e+11
        PoissonsRatio: 0.2800
          MassDensity: []
                  CTE: []
    HystereticDamping: []
```

```
structuralProperties(structuralModel,"Cell",2,"YoungsModulus",210E9, ...
                                              "PoissonsRatio",0.3)
```

```
ans = 
  StructuralMaterialAssignment with properties:

           RegionType: 'Cell'
             RegionID: 2
        YoungsModulus: 2.1000e+11
        PoissonsRatio: 0.3000
          MassDensity: []
                  CTE: []
    HystereticDamping: []
```

Input Arguments

`structuralmodel` — Structural model

`StructuralModel` object

Structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`YMval` — Young's modulus

positive number

Young's modulus of the material, specified as a positive number.

Example:
`structuralProperties(structuralmodel,"YoungsModulus",210e3,"PoissonsRatio",0.3)`

Data Types: `double`

`PRval` — Poisson's ratio

number greater than 0 and less than 0.5

Poisson's ratio of the material, specified as a number greater than 0 and less than 0.5.

Example:
`structuralProperties(structuralmodel,"YoungsModulus",210e3,"PoissonsRatio",0.3)`

Data Types: `double`

`MDval` — Mass density

positive number

Mass density of the material, specified as a positive number. This argument is required for transient and modal models. `MDval` is also required when modeling gravitational effects.

Example:
`structuralProperties(structuralmodel,"YoungsModulus",210e3,"PoissonsRatio",0.3,"MassDensity",11.7e-6)`

Data Types: `double`

`CTEval` — Coefficient of thermal expansion

real number

Coefficient of thermal expansion, specified as a real number. This argument is required for thermal stress analysis. Thermal stress analysis requires the structural model to be static.

Example:
`structuralProperties(structuralmodel,"YoungsModulus",210e3,"PoissonsRatio",0.3,"MassDensity",2.7e-6,"CTE",11.7e-6)`

Data Types: `double`

`g` — Hysteretic damping

nonnegative number

Hysteretic damping, specified as a nonnegative number. This type of damping is also called structural damping.

Example:
`structuralProperties(structuralmodel,"YoungsModulus",210E9,"PoissonsRatio",0.3,"MassDensity",7800,"HystereticDamping",0.1)`

Data Types: `double`

`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:
`structuralProperties(structuralmodel,"Cell",1,"YoungsModulus",110E9,"PoissonsRatio",0.3)`

Data Types: `char` | `string`

`RegionID` — Cell or face ID

vector of positive integers

Cell or face ID, specified as a vector of positive integers. Find the region IDs using `pdegplot` with the `"CellLabels"` (3-D) or `"FaceLabels"` (2-D) value set to `"on"`.

Example: `structuralProperties(structuralmodel,"Cell",1:3,"YoungsModulus",110E9,"PoissonsRatio",0.3)`

Data Types: `double`

Output Arguments

`mtl` — Handle to material properties
`StructuralMaterialAssignment` object

Handle to material properties, returned as a `StructuralMaterialAssignment` object. See StructuralMaterialAssignment Properties.

`mtl` associates the material properties with the geometric region.

Version History

Introduced in R2017b

Hysteretic damping

You can now specify hysteretic damping of a material for direct and modal frequency response models.

Coefficient of thermal expansion

You can now specify a coefficient of thermal expansion.

See Also

`StructuralModel` | `createpde` | `structuralBodyLoad` | `structuralDamping` | `structuralBoundaryLoad` | `structuralBC` | StructuralMaterialAssignment Properties

structuralSEInterface

Package: pde

Specify structural superelement interface for component mode synthesis

Syntax

```
structuralSEInterface(structuralmodel,RegionType,RegionID)
sei = structuralSEInterface( ___ )
```

Description

`structuralSEInterface(structuralmodel,RegionType,RegionID)` defines the specified geometric region `RegionType`, `RegionID` as a superelement interface for component mode synthesis. For better performance, specify geometric regions with a minimal number of nodes. For example, use a set of edges instead of a face, or a set of vertices instead of an edge.

If you intend to use a reduced-order model in Simscape Multibody, use `structuralBC` instead of `structuralSEInterface`.

`sei = structuralSEInterface( ___ )` returns the superelement interface assignment object using the previous syntax.

Examples

Superelement Interfaces for Component Mode Synthesis

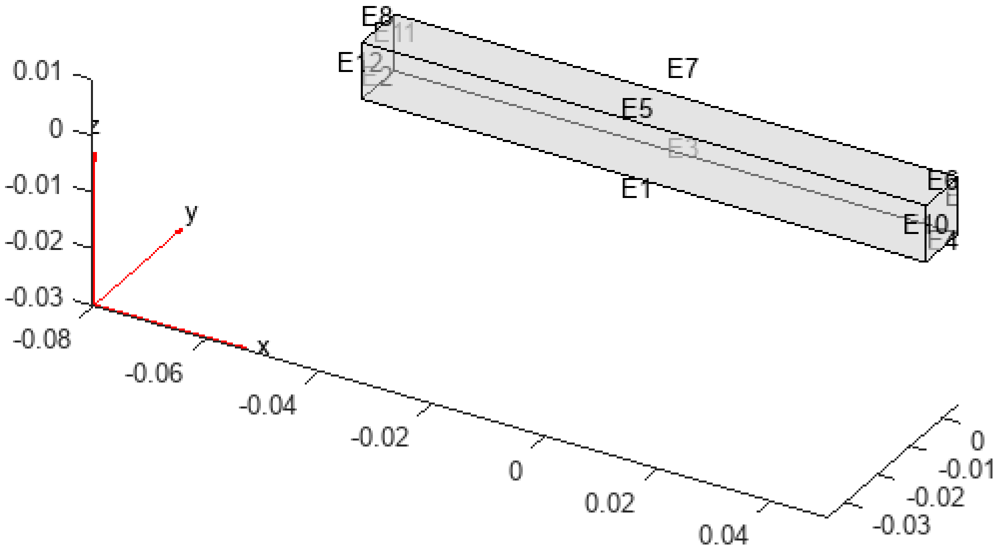
Define the two ends of the beam as structural superelement interfaces. The reduced-order modeling technique retains the degrees of freedom on these boundaries while condensing all other degrees of freedom.

Create a structural model for modal analysis of a 3-D problem.

```
structuralmodel = createpde("structural","modal-solid");
```

Create a geometry and include it in the model. Plot the geometry.

```
gm = multicuboid(0.1,0.01,0.01);
structuralmodel.Geometry = gm;
pdegplot(structuralmodel,"EdgeLabels","on","FaceAlpha",0.5)
```



Specify Young's modulus, Poisson's ratio, and the mass density of the material.

```
structuralProperties(structuralmodel,"YoungsModulus",70E9, ...
                                     "PoissonsRatio",0.3, ...
                                     "MassDensity",2700);
```

Generate a mesh.

```
generateMesh(structuralmodel);
```

Specify the ends of the beam as structural superelement interfaces. For better performance, use the set of edges bounding each side of the beam instead of using the entire face.

```
structuralSEInterface(structuralmodel,"Edge",[4,6,9,10]);
structuralSEInterface(structuralmodel,"Edge",[2,8,11,12]);
```

Reduce the model to all modes in the frequency range `[-Inf,500000]` and the interface degrees of freedom.

```
R = reduce(structuralmodel,"FrequencyRange",[-Inf,500000])
```

```
R = 
  ReducedStructuralModel with properties:

                   K: [166x166 double]
                   M: [166x166 double]
            NumModes: 22
         RetainedDoF: [144x1 double]
```

```
    ReferenceLocations: []
                  Mesh: [1x1 FEMesh]
```

Input Arguments

`structuralmodel` — Structural model

`StructuralModel` object

Structural model, specified as a `StructuralModel` object. The model contains the geometry, mesh, structural properties of the material, body loads, boundary loads, and boundary conditions.

Example: `structuralmodel = createpde("structural","transient-solid")`

`RegionType` — Geometric region type

`"Vertex"` | `"Edge"` | `"Face"` (for a 3-D model only)

Geometric region type, specified as `"Vertex"`, `"Edge"`, or, for a 3-D model, `"Face"`.

Example: `structuralSEInterface(structuralmodel,"Face",[2,5])`

Data Types: `char` | `string`

`RegionID` — Geometric region ID

vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `structuralSEInterface(structuralmodel,"Face",[2,5])`

Data Types: `double`

Output Arguments

`sei` — Handle to superelement interface

`StructuralSEIAssignment` object

Superelement interface assignment, returned as a `StructuralSEIAssignment` object. See StructuralSEIAssignment Properties.

Version History

Introduced in R2019b

See Also

`StructuralModel` | `structuralBC` | `reduce` | `reconstructSolution` | `solve` | `ReducedStructuralModel` | StructuralSEIAssignment Properties

StructuralSEIAssignment Properties

Superelement interface assignment for structural model

Description

A `StructuralSEIAssignment` object contains a description of the superelement interfaces for a structural analysis model. A `StructuralModel` container has a vector of `StructuralSEIAssignment` objects in its `SuperelementInterfaces.StructuralSEIAssignments` property.

Properties

Properties of StructuralSEIAssignment

`RegionType` — Region type
`'Vertex'` | `'Edge'` | `'Face'` (for a 3-D model only)

Region type, specified as `'Vertex'`, `'Edge'`, or, for a 3-D model, `'Face'`.

Data Types: `char` | `string`

`RegionID` — Region ID
positive integer

Geometric region ID, specified as a positive integer. Find the region IDs by using `pdegplot`.

Data Types: `double`

Version History

Introduced in R2019b

See Also

`structuralSEInterface` | `structuralBC` | `reduce` | `reconstructSolution` | `solve` | `ReducedStructuralModel` | `StructuralModel` | StructuralBC Properties

StationaryResults

Time-independent PDE solution and derived quantities

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`.

Creation

There are several ways to create a `StationaryResults` object:

- Solve a time-independent problem using the `solvepde` function. This function returns a PDE solution as a `StationaryResults` object. This is the recommended approach.
- Solve a time-independent problem using the `assempde` or `pdenonlin` function. Then use the `createPDEResults` function to obtain a `StationaryResults` object from a PDE solution returned by `assempde` or `pdenonlin`. Note that `assempde` and `pdenonlin` are legacy functions. They are not recommended for solving PDE problems.

Properties

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as a FEMesh object.

`NodalSolution` — Solution values at the nodes
vector | array

This property is read-only.

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 3-356.

Data Types: `double`

`XGradients` — *x*-component of gradient at the nodes

vector | array

This property is read-only.

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 3-356.

Data Types: `double`

`YGradients` — *y*-component of gradient at the nodes

vector | array

This property is read-only.

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 3-356.

Data Types: `double`

`ZGradients` — *z*-component of gradient at the nodes

vector | array

This property is read-only.

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 3-356.

Data Types: `double`

Object Functions

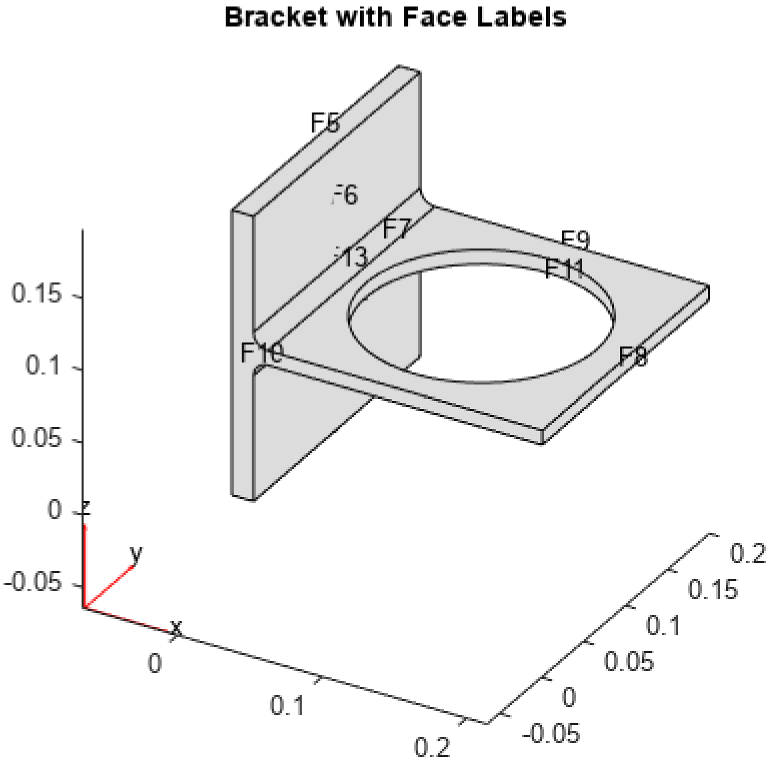
evaluateCGradient	Evaluate flux of PDE solution
evaluateGradient	Evaluate gradients of PDE solutions at arbitrary points
interpolateSolution	Interpolate PDE solution to arbitrary points

Examples

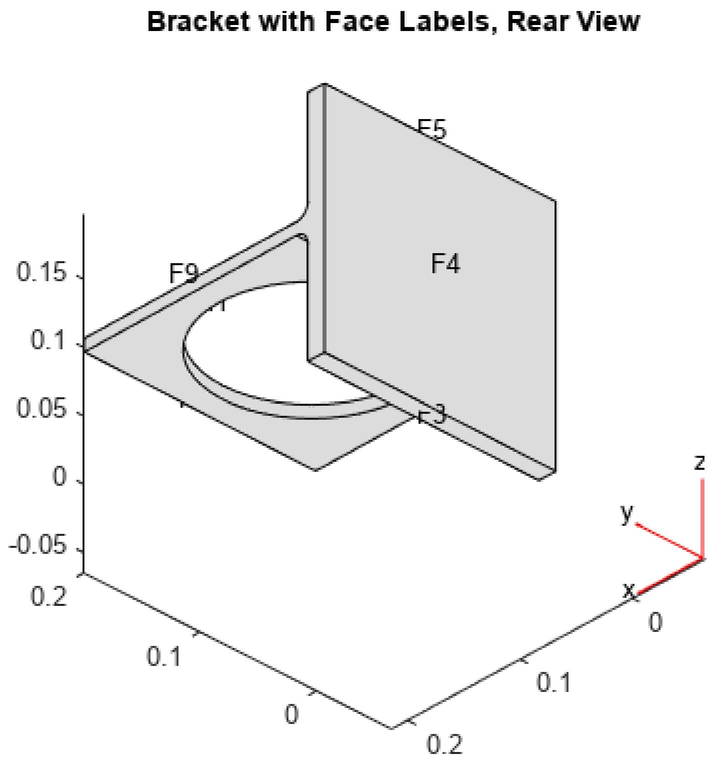
Obtain a `StationaryResults` Object from `solvepde`

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")
```



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 “Linear Elasticity Equations” on page 3-156.

```
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:
```

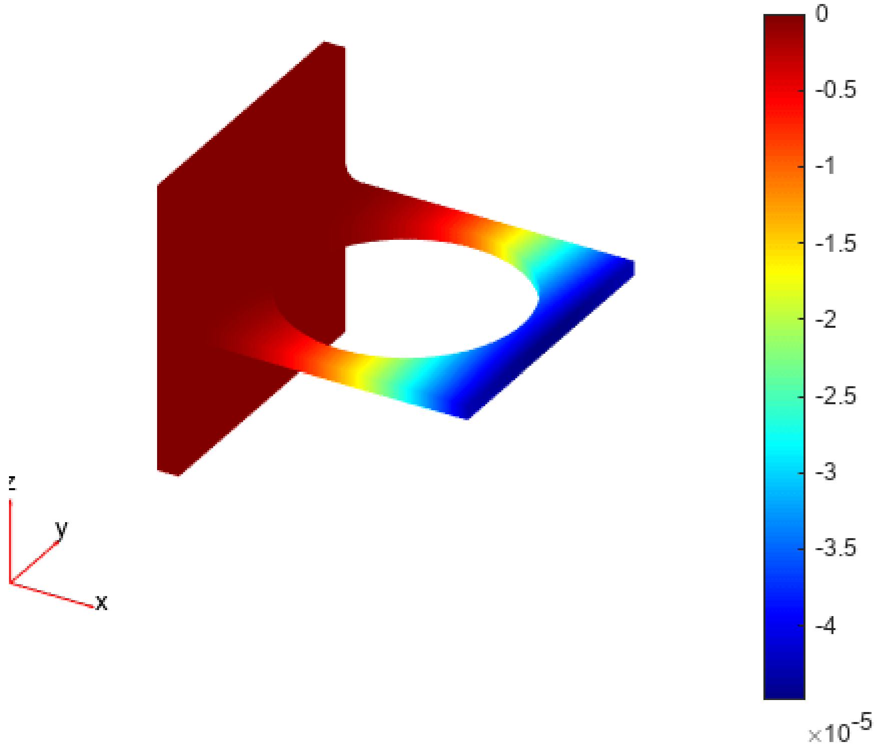
```
    NodalSolution: [13911x3 double]
       XGradients: [13911x3 double]
       YGradients: [13911x3 double]
       ZGradients: [13911x3 double]
             Mesh: [1x1 FEMesh]
```

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))
```



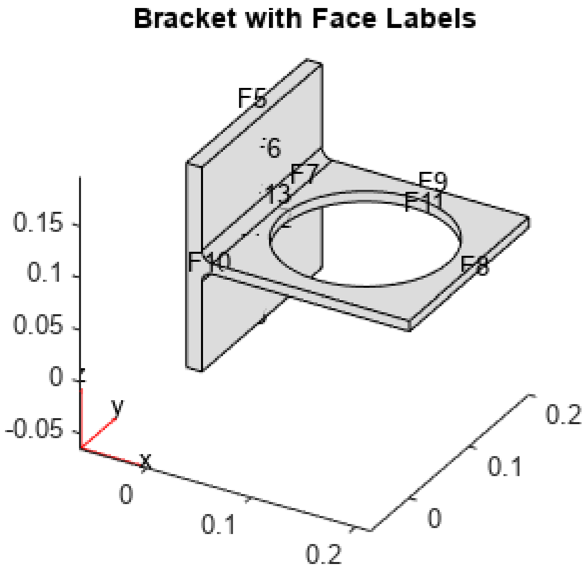
Results from `createPDEResults`

Obtain a `StationaryResults` object from a legacy solver together with `createPDEResults`.

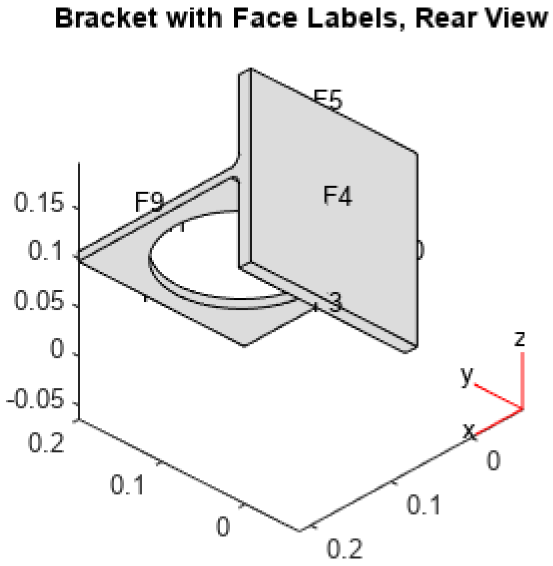
Create a PDE model for a system of three equations. Import the geometry of a bracket and plot the face labels.

```
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")
```



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 “Linear Elasticity Equations” on page 3-156.

```
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)
```

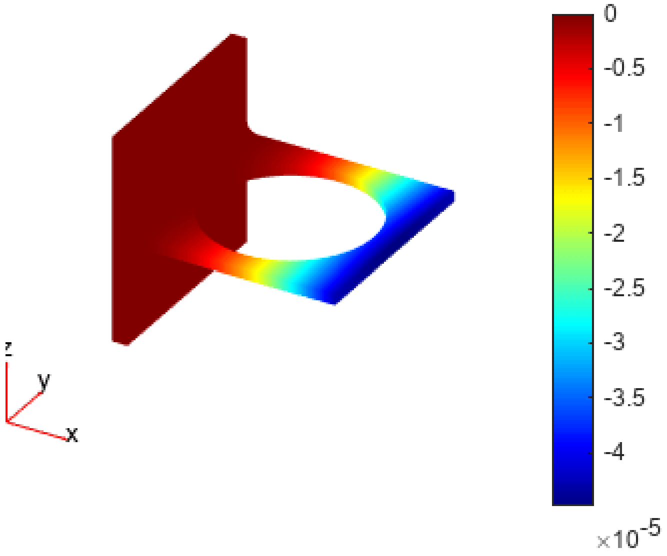
```
results = 
  StationaryResults with properties:

    NodalSolution: [13911x3 double]
       XGradients: [13911x3 double]
       YGradients: [13911x3 double]
       ZGradients: [13911x3 double]
             Mesh: [1x1 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))
```



Version History

Introduced in R2016a

Added `evaluateCGradient` function

You can now evaluate flux of PDE solution as a tensor product of c-coefficient and gradient of PDE solution.

See Also

`solvepde` | `interpolateSolution` | `evaluateGradient` | `evaluateCGradient` | `EigenResults` | `TimeDependentResults`

Topics

"Poisson's Equation on Unit Disk" on page 3-224
"Minimal Surface Problem" on page 3-241
"Solve Problems Using PDEModel Objects" on page 2-2

SteadyStateThermalResults

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 a specified boundary, use `evaluateHeatRate`.

Creation

Solve a steady-state thermal problem using the `solve` function. This function returns a steady-state thermal solution as a `SteadyStateThermalResults` object.

Properties

All Steady-State Thermal Models

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as an FEMesh object.

`Temperature` — Temperature values at nodes
vector

This property is read-only.

Temperature values at nodes, returned as a vector.

Data Types: `double`

Non-Axisymmetric Steady-State Thermal Models

`XGradients` — *x*-component of temperature gradient at nodes
vector

This property is read-only.

x-component of the temperature gradient at nodes, returned as a vector.

Data Types: `double`

`YGradients` — *y*-component of temperature gradient at nodes
vector

This property is read-only.

y-component of the temperature gradient at nodes, returned as a vector.

Data Types: `double`

`ZGradients` — *z*-component of temperature gradient at nodes
vector

This property is read-only.

z-component of the temperature gradient at nodes, returned as a vector.

Data Types: `double`

Axisymmetric Steady-State Thermal Models

`RGradients` — *r*-component of temperature gradient at nodes
vector

This property is read-only.

r-component of the temperature gradient at nodes, returned as a vector.

Data Types: `double`

`ZGradients` — *z*-component of temperature gradient at nodes for axisymmetric model
vector

This property is read-only.

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

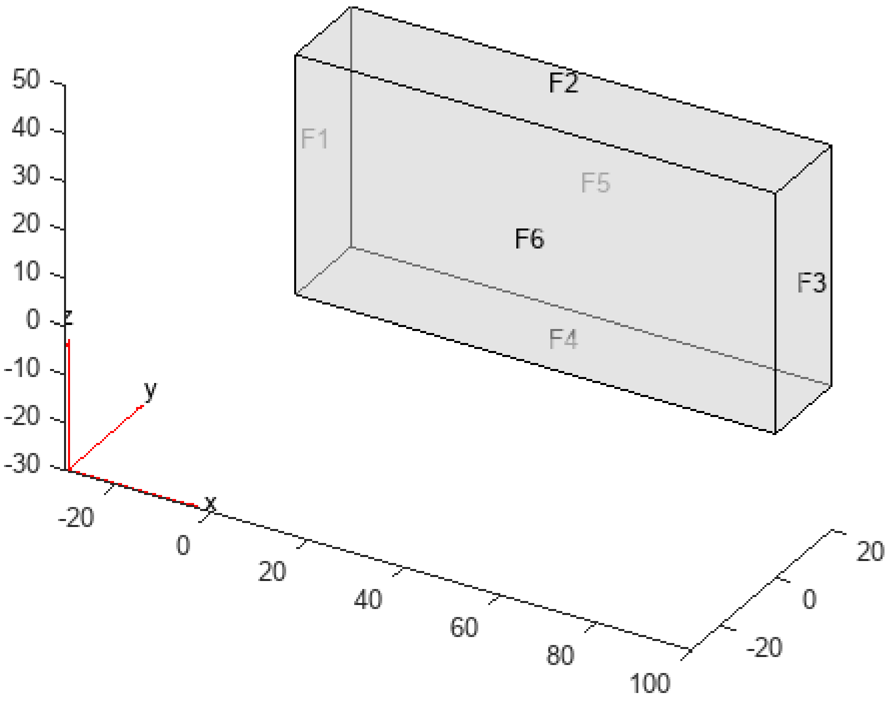
Solve a 3-D steady-state thermal problem.

Create a thermal model for this problem.

```
thermalmodel = createpde("thermal");
```

Import and plot the block geometry.

```
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 °C to the left side of the block (face 1) and a constant temperature of 300 °C to 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);
```

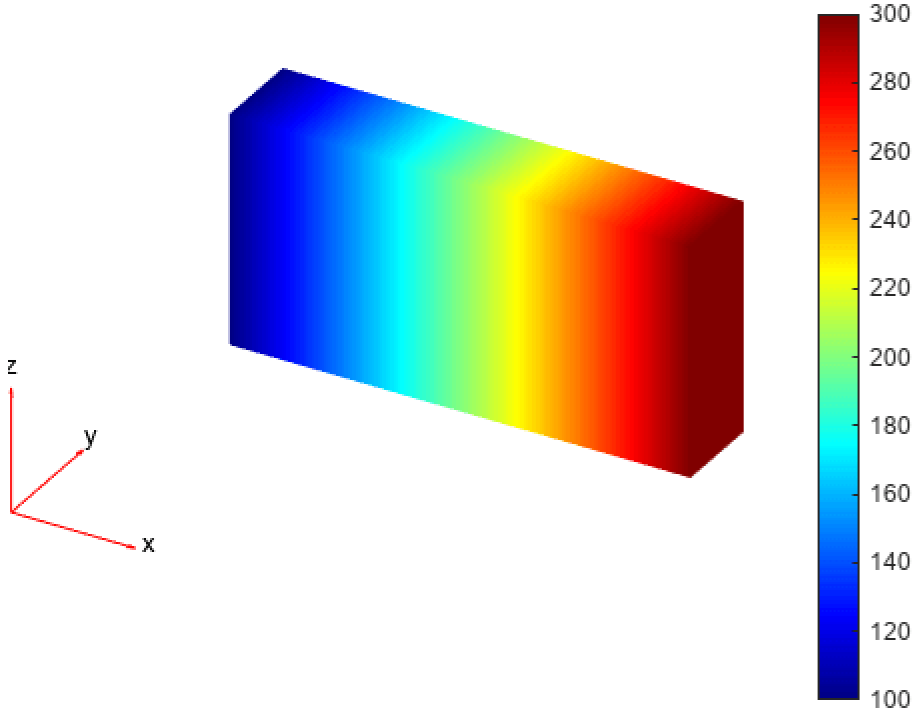
Mesh the geometry and solve the problem.

```
generateMesh(thermalmodel);
thermalresults = solve(thermalmodel)
```

```
thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [12691x1 double]
     XGradients: [12691x1 double]
     YGradients: [12691x1 double]
     ZGradients: [12691x1 double]
           Mesh: [1x1 FEMesh]
```

The solver finds the 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 the nodal locations.

```
pdeplot3D(thermalmodel,"ColorMapData",thermalresults.Temperature)
```



Solution to Steady-State Axisymmetric Thermal Model

Analyze heat transfer in a rod with a circular cross-section and internal heat generation by simplifying a 3-D axisymmetric model to a 2-D model.

Create a steady-state thermal model for solving an axisymmetric problem.

```
thermalmodel = createpde("thermal","steadystate-axisymmetric");
```

The 2-D model is a rectangular strip whose *x*-dimension extends from the axis of symmetry to the outer surface and whose *y*-dimension extends over the actual length of the rod (from -1.5 m to 1.5 m). Create the geometry by specifying the coordinates of its four corners. For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through *r* = 0.

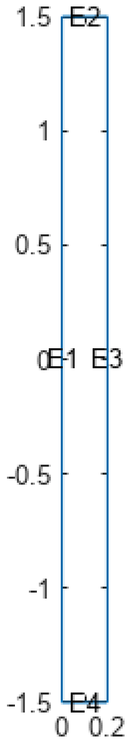
```
g = decsg([3 4 0 0 .2 .2 -1.5 1.5 1.5 -1.5]');
```

Include the geometry in the model.

```
geometryFromEdges(thermalmodel,g);
```

Plot the geometry with the edge labels.

```
figure
pdegplot(thermalmodel,"EdgeLabels","on")
axis equal
```



The rod is composed of a material with these thermal properties.

```
k = 40; % thermal conductivity, W/(m*C)
q = 20000; % heat source, W/m^3
```

For a steady-state analysis, specify the thermal conductivity of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",k);
```

Specify the internal heat source.

```
internalHeatSource(thermalmodel,q);
```

Define the boundary conditions. There is no heat transferred in the direction normal to the axis of symmetry (edge 1). You do not need to change the default boundary condition for this edge. Edge 2 is kept at a constant temperature T = 100 °C.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Specify the convection boundary condition on the outer boundary (edge 3). The surrounding temperature at the outer boundary is 100 °C, and the heat transfer coefficient is 50 $W/(m\cdot^\circ C)$.

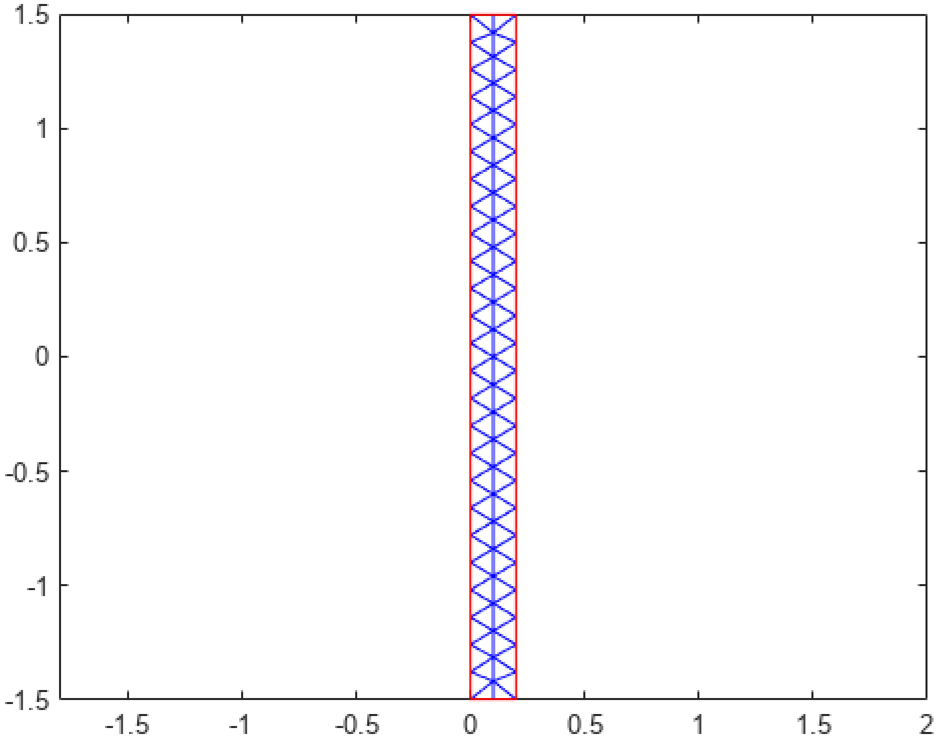
```
thermalBC(thermalmodel,"Edge",3,...
                       "ConvectionCoefficient",50,...
                       "AmbientTemperature",100);
```

The heat flux at the bottom of the rod (edge 4) is 5000 W/m^2.

```
thermalBC(thermalmodel,"Edge",4,"HeatFlux",5000);
```

Generate the mesh.

```
msh = generateMesh(thermalmodel);
figure
pdeplot(thermalmodel)
axis equal
```

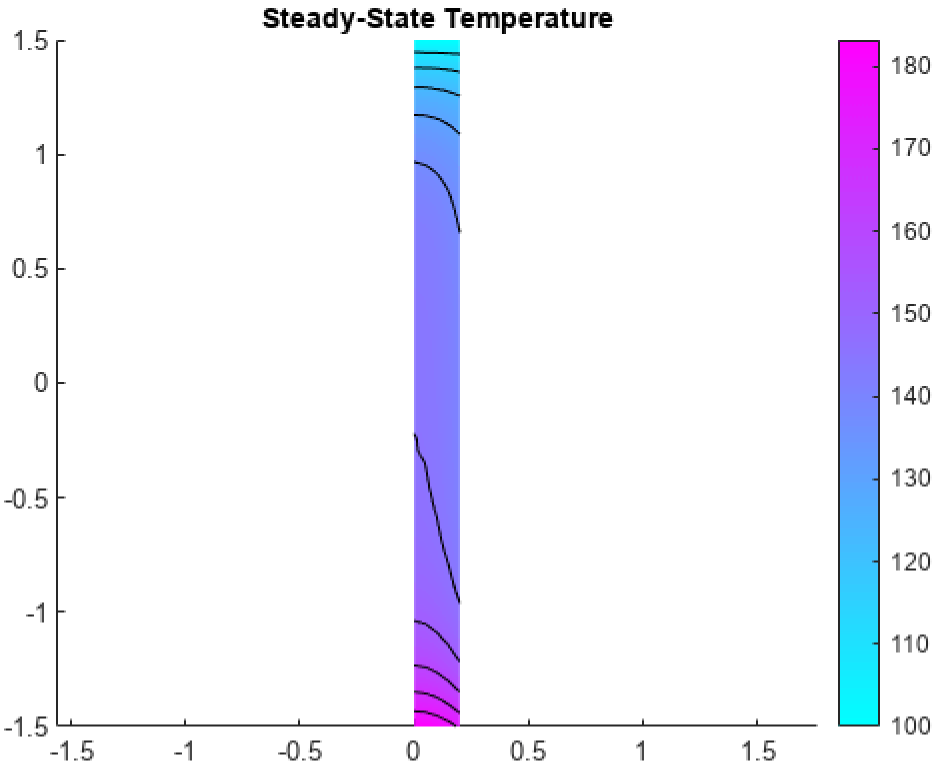


Solve the problem.

```
thermalresults = solve(thermalmodel)

thermalresults = 
  SteadyStateThermalResults with properties:

    Temperature: [259x1 double]
     RGradients: [259x1 double]
     ZGradients: [259x1 double]
           Mesh: [1x1 FEMesh]
```

The solver finds the temperatures and temperature gradients at the nodal locations. To access these values, use `thermalresults.Temperature`, `thermalresults.RGradients`, and `thermalresults.ZGradients`. For example, plot temperatures at the nodal locations.

```
T = thermalresults.Temperature;
figure
pdeplot(thermalmodel,"XYData",T,"Contour","on")
axis equal
title("Steady-State Temperature")
```



Version History

Introduced in R2017a

Axisymmetric analysis

`SteadyStateThermalResults` now supports axisymmetric thermal results. Axisymmetric analysis simplifies 3-D structural and thermal problems to 2-D using their symmetry around the axis of rotation.

See Also

`evaluateHeatFlux` | `evaluateHeatRate` | `evaluateTemperatureGradient` | `interpolateTemperature` | `TransientThermalResults` | `ModalThermalResults`

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

Properties

`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.

Data Types: `char` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by 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, specified 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, specified 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, specified 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, specified 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, specified 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”. For details of this evaluation, see “Nonconstant Boundary Conditions” on page 2-132.

Data Types: `char` | `string`

`Label` — Label for use with `linearizeInput`
character vector | string

Label for use with `linearizeInput`, specified as a character vector or a string.

Data Types: `char` | `string`

Version History

Introduced in R2017a

See Also

`thermalBC` | `findThermalBC` | `ThermalModel`

ThermalMaterialAssignment Properties

Thermal 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

Properties

`RegionType` — Region type
`'Face'` | `'Cell'`

Region type, specified as `'Face'` for a 2-D region, or `'Cell'` for a 3-D region.

Data Types: `char` | `string`

`RegionID` — Region ID
vector of positive integers

Region ID, specified 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, specified 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, specified 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, specified as a nonnegative number or a function handle.

Data Types: `double` | `function_handle`

Tips

- When there are multiple assignments to the same geometric region, the toolbox uses the last applied setting.
- To avoid assigning material properties to a wrong region, ensure that you are using the correct geometric region IDs by plotting and visually inspecting the geometry.

Version History

Introduced in R2017a

See Also

`findThermalProperties` | `thermalProperties`

ThermalModel

Thermal model object

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.

Creation

Create a `ThermalModel` object using `createpde` with the first argument `"thermal"`.

Properties

`AnalysisType` — Type of thermal analysis
`"steadystate"` | `"transient"` | `"modal"` | `"steadystate-axisymmetric"` | `"transient-axisymmetric"` | `"modal-axisymmetric"`

Type of thermal analysis, specified as `"steadystate"`, `"transient"`, `"modal"`, `"steadystate-axisymmetric"`, `"transient-axisymmetric"`, or `"modal-axisymmetric"`.

To change a thermal analysis type, assign a new type to `model.AnalysisType`. Ensure that all other properties of the model are consistent with the new analysis type.

`Geometry` — Geometry description
`AnalyticGeometry` | `DiscreteGeometry`

Geometry description, specified as `AnalyticGeometry` for a 2-D geometry or `DiscreteGeometry` for a 2-D or 3-D geometry.

`MaterialProperties` — Material properties within the domain
object containing material property assignments

Material properties within the domain, specified 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, specified 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, specified 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, specified as an object containing the initial temperature assignments within the geometric domain.

`Mesh` — Finite element mesh

`FEMesh` object

Finite element mesh, specified as an `FEMesh` object. For details, see FEMesh. You create the mesh by 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·K^4)
- CGS – 5.6704e-5 erg/(cm^2·s·K^4)
- US customary – 1.714e-9 BTU/(hr·ft^2·R^4)

`LinearizeInputs` — Inputs for linearized model

structure array

Inputs for a linearized model, specified as a structure array. The inputs are used by the `linearize` that extracts a `sparss` model from a thermal model.

`LinearizeOutputs` — Outputs for linearized model

structure array

Inputs for a linearized model, specified as a structure array. The outputs are used by the `linearize` that extracts a `sparss` model from a thermal model.

`SolverOptions` — Algorithm options for PDE solvers

`PDESolverOptions` object

Algorithm options for the PDE solvers, specified 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 from decomposed geometry matrix
geometryFromMesh	Create 2-D or 3-D geometry from mesh
importGeometry	Import geometry from STL or STEP file
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, structural analysis, or electromagnetic analysis problem

Examples

Create and Populate 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: [1x1 pde.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
       Vertices: [4x2 double]
```

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 of 25.

```
internalHeatSource(thermalmodel,25)
```

```
ans = 
  HeatSourceAssignment with properties:

    RegionType: 'face'
      RegionID: 1
    HeatSource: 25
         Label: []
```

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'
                    Label: []
```

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 the mesh.

```
generateMesh(thermalmodel)
```

```
ans = 
  FEMesh with properties:

             Nodes: [2x1541 double]
          Elements: [6x734 double]
    MaxElementSize: 0.1131
```

```
    MinElementSize: 0.0566
     MeshGradation: 1.5000
    GeometricOrder: 'quadratic'
```

`thermalmodel` now contains the following properties.

```
thermalmodel
```

```
thermalmodel = 
  ThermalModel with properties:

               AnalysisType: "transient"
                   Geometry: [1x1 AnalyticGeometry]
         MaterialProperties: [1x1 MaterialAssignmentRecords]
                HeatSources: [1x1 HeatSourceAssignmentRecords]
    StefanBoltzmannConstant: 5.6704e-08
         BoundaryConditions: [1x1 ThermalBCRecords]
          InitialConditions: [1x1 ThermalICRecords]
                       Mesh: [1x1 FEMesh]
              SolverOptions: [1x1 pde.PDESolverOptions]
```

Version History

Introduced in R2017a

Reduced-order modeling for thermal analysis

You can now approximate the dynamics of the original system with a smaller system while retaining most of the dynamic characteristics. See `reduce` for details.

Sparse linear models for use with Control System Toolbox

`ThermalModel` now has additional properties, `LinearizeInputs` and `LinearizeOutputs` containing inputs and outputs for linearized model. They are used by the `linearize` that extracts a `sparss` model from a thermal model.

Axisymmetric analysis

You can now specify `AnalysisType` for axisymmetric models. Axisymmetric analysis simplifies 3-D thermal problems to 2-D using their symmetry around the axis of rotation.

See Also

`createpde` | `generateMesh` | `geometryFromEdges` | `geometryFromMesh` | `importGeometry` | `internalHeatSource` | `thermalProperties` | `pdegplot` | `pdeplot` | `pdeplot3D` | `thermalBC` | `thermalIC`

thermalProperties

Package: `pde`

Assign thermal properties of a material for a thermal model

Syntax

```
thermalProperties(thermalmodel,"ThermalConductivity",TCval,"MassDensity",
MDval,"SpecificHeat",SHval)
thermalProperties( ___ ,RegionType,RegionID)
mtl = thermalProperties( ___ )
```

Description

`thermalProperties(thermalmodel,"ThermalConductivity",TCval,"MassDensity",MDval,"SpecificHeat",SHval)` assigns material properties, such as thermal conductivity, mass density, and specific heat. For transient analysis, specify all three properties. For steady-state analysis, specifying thermal conductivity is enough. This syntax sets material properties for the entire geometry.

For a nonconstant or nonlinear material, specify `TCval`, `MDval`, and `SHval` as function handles.

`thermalProperties( ___ ,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.

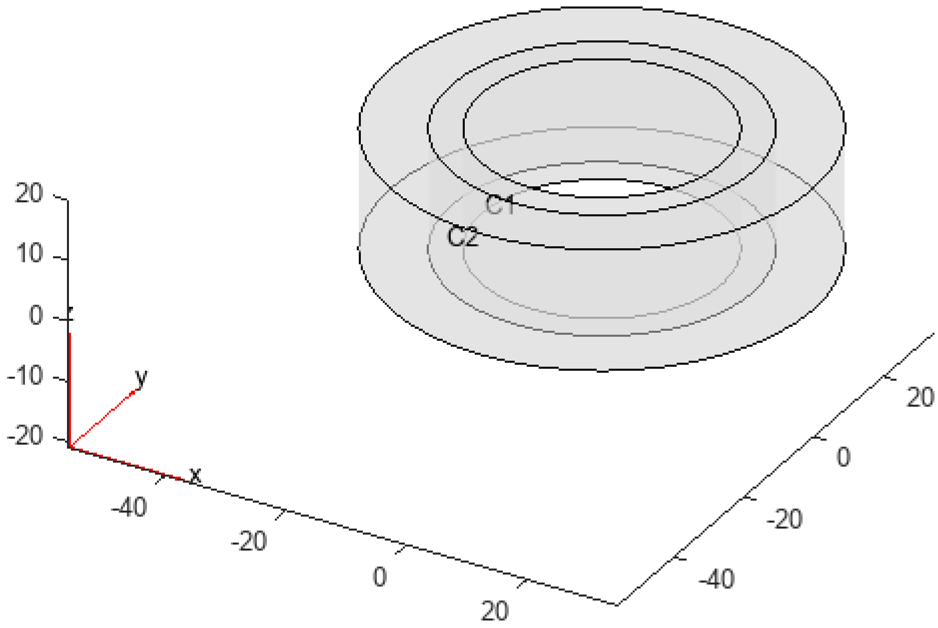
```
thermalModel = createpde("thermal");
```

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)
```

```
ans = 
  ThermalMaterialAssignment with properties:

             RegionType: 'cell'
               RegionID: 1
    ThermalConductivity: 0.4000
            MassDensity: []
           SpecificHeat: []
```

```
thermalProperties(thermalModel,"Cell",2,"ThermalConductivity",0.0015)
```

```
ans = 
  ThermalMaterialAssignment with properties:

             RegionType: 'cell'
               RegionID: 2
    ThermalConductivity: 0.0015
            MassDensity: []
           SpecificHeat: []
```

Specify Nonconstant Thermal Properties

Use function handles to specify a thermal conductivity that depends on temperature and specific heat that depends on coordinates.

Create a thermal model for transient analysis and include the geometry. The geometry is a rod with a circular cross section. The 2-D model is a rectangular strip whose *y*-dimension extends from the axis of symmetry to the outer surface, and whose *x*-dimension extends over the actual length of the rod.

```
thermalmodel = createpde("thermal","transient");

g = decsg([3 4 -1.5 1.5 1.5 -1.5 0 0 .2 .2]');
geometryFromEdges(thermalmodel,g);
```

Specify the thermal conductivity as a linear function of temperature, $k = 40 + 0.003T$.

```
k = @(location,state)40 + 0.003*state.u;
```

Specify the specific heat as a linear function of the *y*-coordinate, $cp = 500y$.

```
cp = @(location,state)500*location.y;
```

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",k,...
                               "MassDensity",2.7*10^(-6),...
                               "SpecificHeat",cp)
```

```
ans = 
  ThermalMaterialAssignment with properties:

             RegionType: 'face'
               RegionID: 1
    ThermalConductivity: @(location,state)40+0.003*state.u
            MassDensity: 2.7000e-06
           SpecificHeat: @(location,state)500*location.y
```

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","steadystate")`

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` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

Example: `thermalProperties(thermalmodel,"Cell",1:3,"ThermalConductivity",100)`

Data Types: `double`

`TCval` — 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.

Use a function handle to specify the thermal conductivity that depends on space, time, or temperature. For details, see “More About” on page 5-1382.

Example: `thermalProperties(thermalmodel,"Cell",1,"ThermalConductivity",100)` or `thermalProperties(thermalmodel,"ThermalConductivity",[80;10;80])` for orthotropic thermal conductivity

Data Types: `double` | `function_handle`

`MDval` — 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.

Use a function handle to specify the mass density that depends on space, time, or temperature. For details, see “More About” on page 5-1382.

Example:
`thermalProperties(thermalmodel,"Cell",1,"ThermalConductivity",100,"MassDensity",2730e-9,"SpecificHeat",910)`

Data Types: `double` | `function_handle`

`SHval` — 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.

Use a function handle to specify the specific heat that depends on space, time, or temperature. For details, see “More About” on page 5-1382.

Example:
`thermalProperties(thermalmodel,"Cell",1,"ThermalConductivity",100,"MassDensity",2730e-9,"SpecificHeat",910)`

Data Types: `double` | `function_handle`

Output Arguments

`mtl` — Handle to material properties
`ThermalMaterialAssignment` object

Handle to material properties, returned as a `ThermalMaterialAssignment` object. See ThermalMaterialAssignment Properties.

`mtl` associates material properties with the geometric region.

More About

Specifying Nonconstant Parameters of a Thermal Model

Use a function handle to specify these thermal parameters when they depend on space, temperature, and time:

- Thermal conductivity of the material
- Mass density of the material
- Specific heat of the material
- Internal heat source
- Temperature on the boundary
- Heat flux through the boundary
- Convection coefficient on the boundary
- Radiation emissivity coefficient on the boundary
- Initial temperature (can depend on space only)

For example, use function handles to specify the thermal conductivity, internal heat source, convection coefficient, and initial temperature for this model.

```
thermalProperties(model,"ThermalConductivity", ...
                        @myfunConductivity)
internalHeatSource(model,"Face",2,@myfunHeatSource)
thermalBC(model,"Edge",[3,4], ...
               "ConvectionCoefficient",@myfunBC, ...
               "AmbientTemperature",27)
thermalIC(model,@myfunIC)
```

For all parameters, except the initial temperature, the function must be of the form:

```
function thermalVal = myfun(location,state)
```

For the initial temperature the function must be of the form:

```
function thermalVal = myfun(location)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments (or one argument if the function specifies the initial temperature).

- `location` — A structure containing these fields:

  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes these data in the `location` structure:

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, *r*-component of the normal vector at the evaluation point or points
- `state` — A structure containing these fields for transient or nonlinear problems:
  - `state.u` — Temperatures at the corresponding points of the location structure
  - `state.ux` — Estimates of the *x*-component of temperature gradients at the corresponding points of the location structure
  - `state.uy` — Estimates of the *y*-component of temperature gradients at the corresponding points of the location structure
  - `state.uz` — For a 3-D or an axisymmetric geometry, estimates of the *z*-component of temperature gradients at the corresponding points of the location structure
  - `state.ur` — For an axisymmetric geometry, estimates of the *r*-component of temperature gradients at the corresponding points of the location structure
  - `state.time` — Time at evaluation points

Thermal material properties (thermal conductivity, mass density, and specific heat) and internal heat source get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID
- `state.u`, `state.ux`, `state.uy`, `state.uz`, `state.r`, `state.time`

Boundary conditions (temperature on the boundary, heat flux, convection coefficient, and radiation emissivity coefficient) get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`
- `state.u`, `state.time`

Initial temperature gets the following data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

For all thermal parameters, except for thermal conductivity, your function must return a row vector `thermalVal` with the number of columns equal to the number of evaluation points, for example, `M = length(location.y)`.

For thermal conductivity, your function must return a matrix `thermalVal` with number of rows equal to 1, `Ndim`, `Ndim*(Ndim+1)/2`, or `Ndim*Ndim`, where `Ndim` is 2 for 2-D problems and 3 for 3-D problems. The number of columns must equal the number of evaluation points, for example, `M = length(location.y)`. For details about dimensions of the matrix, see “c Coefficient for specifyCoefficients” on page 2-92.

If properties depend on the time or temperature, 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 time dependent by passing `NaN` state values and looking for returned `NaN` values.

Additional Arguments in Functions for Nonconstant Thermal Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
thermalVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
thermalBC(model,"Edge",3,"Temperature",thermalVal)

thermalVal = @(location) myfunWithAdditionalArgs(location,arg1,arg2...)
thermalIC(model,thermalVal)
```

Version History

Introduced in R2017a

See Also

`internalHeatSource` | `specifyCoefficients` | ThermalMaterialAssignment Properties

Topics

“Heat Conduction in Multidomain Geometry with Nonuniform Heat Flux” on page 3-270

translate

Package: `pde`

Translate geometry

Syntax

```
translate(g,s)
h = translate(g,s)
```

Description

`translate(g,s)` translates the geometry `g` by the distance `s`.

`h = translate(g,s)` returns a handle `h` to the resulting geometry object `g`. If the original geometry is a `DiscreteGeometry` object, then the function modifies the original geometry and returns the handle `h` to the modified `DiscreteGeometry` geometry object. If the original geometry is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` geometry object. In this case, the original geometry remains unchanged.

Examples

Move 2-D Geometry Along Coordinate Axes

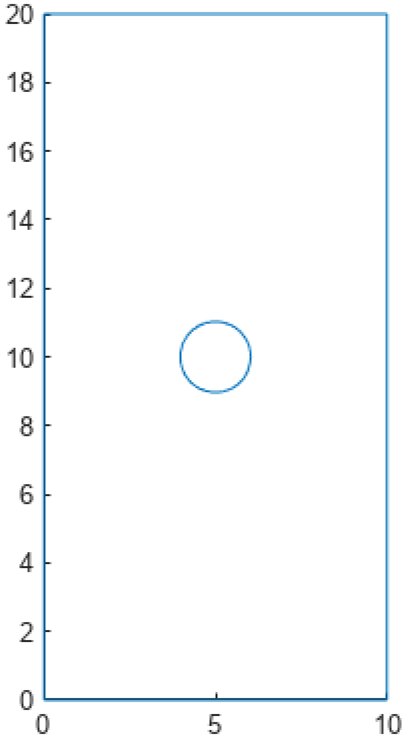
Translate a geometry by different distances along the *x*- and *y*-axes.

Create a model.

```
model = createpde;
```

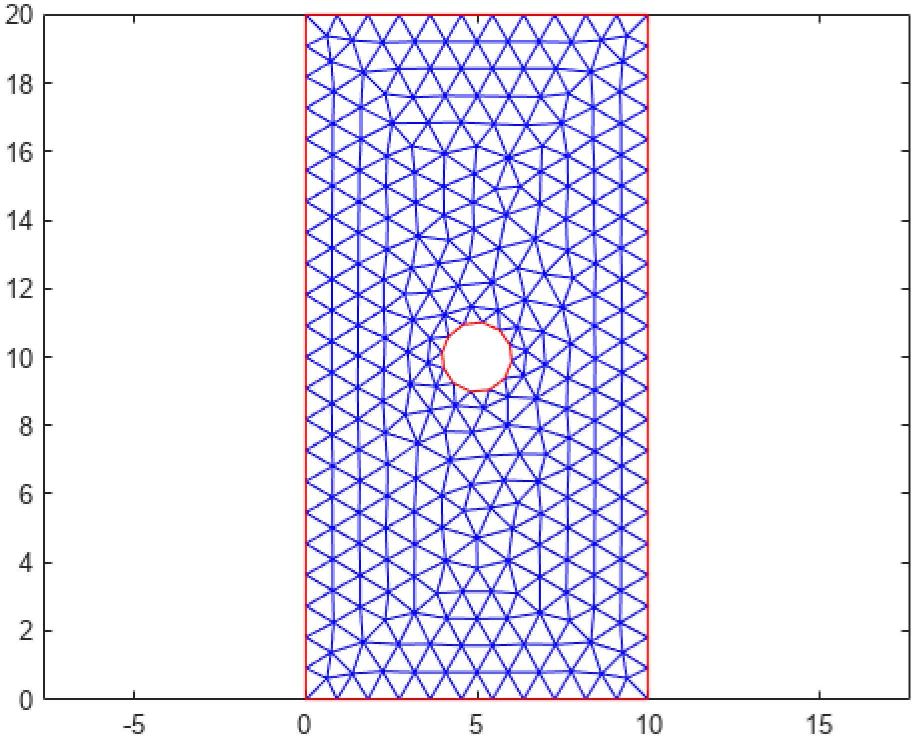
Import and plot a geometry.

```
g = importGeometry(model,"PlateHolePlanar.stl");
pdegplot(g)
```



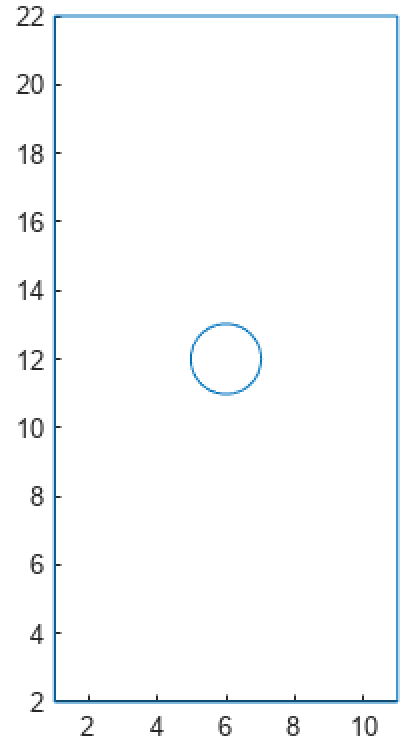
Mesh the geometry and plot the mesh.

```
generateMesh(model);

figure
pdemesh(model)
```



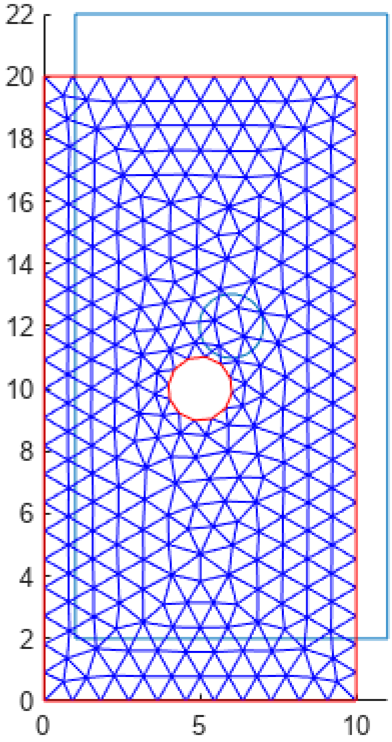
Translate the 2-D geometry by 1 along the *x*-axis and by 2 along the *y*-axis. Plot the result.

```
translate(g,[1 2]);
pdegplot(g)
```



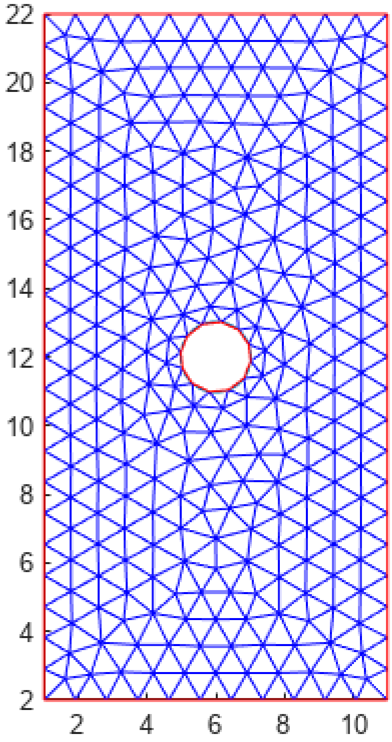
Plot the geometry and mesh. The `translate` function modifies a geometry, but it does not modify a mesh.

```
figure
pdegplot(g)
hold on
pdemesh(model)
```



After modifying the geometry, always regenerate the mesh.

```
generateMesh(model);

figure
pdegplot(g)
hold on
pdemesh(model)
```



Input Arguments

g — Geometry

`DiscreteGeometry` object | `AnalyticGeometry` object

Geometry, specified as a `DiscreteGeometry` or `AnalyticGeometry` object.

s — Translation distance

vector of two or three nonzero real numbers

Translation distance, specified as a vector of two or three real numbers. The distance is a vector of two elements for a 2-D geometry or three elements for a 3-D geometry. The elements specify the distance along the *x*-, *y*-, and, for a 3-D geometry, *z*-axes.

Output Arguments

h — Resulting geometry

handle

Resulting geometry, returned as a handle. If the original geometry `g` is a `DiscreteGeometry` object, then `h` is a handle to the modified `DiscreteGeometry` geometry object `g`. If `g` is an `AnalyticGeometry` object, then `h` is a handle to a new `DiscreteGeometry` geometry object. In this case, the original geometry `g` remains unchanged.

Tips

- `translate` modifies a geometry, but it does not modify a mesh. After modifying a geometry, regenerate the mesh to ensure a proper mesh association with the new geometry.
- If `g` is an `AnalyticGeometry` object, and you want to replace it with the resulting discrete geometry, assign the output to the original geometry, for example, `g = translate(g,[1 2])`.

Version History

Introduced in R2020a

Geometry transformation for analytic geometries

`translate` now works with `AnalyticGeometry` objects.

See Also

`rotate` | `scale` | AnalyticGeometry Properties | DiscreteGeometry Properties | `pdegplot` | `importGeometry` | `geometryFromMesh` | `generateMesh`

TimeDependentResults

Time-dependent PDE solution and derived quantities

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`.

Creation

There are several ways to create a `TimeDependentResults` object:

- Solve a time-dependent problem using the `solvepde` function. This function returns a PDE solution as a `TimeDependentResults` object. This is the recommended approach.
- Solve a time-dependent problem using the `parabolic` or `hyperbolic` function. Then use the `createPDEResults` function to obtain a `TimeDependentResults` object from a PDE solution returned by `parabolic` or `hyperbolic`. Note that `parabolic` and `hyperbolic` are legacy functions. They are not recommended for solving PDE problems.

Properties

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as a FEMesh object.

`NodalSolution` — Solution values at the nodes
vector | array

This property is read-only.

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 3-356.

Data Types: `double`
Complex Number Support: Yes

`SolutionTimes` — Solution times

real vector

This property is read-only.

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

This property is read-only.

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 3-356.

Data Types: `double`
Complex Number Support: Yes

`YGradients` — *y*-component of gradient at the nodes

vector | array

This property is read-only.

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 3-356.

Data Types: `double`
Complex Number Support: Yes

`ZGradients` — *z*-component of gradient at the nodes

vector | array

This property is read-only.

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 3-356.

Data Types: `double`

Object Functions

evaluateCGradient	Evaluate flux of PDE solution
evaluateGradient	Evaluate gradients of PDE solutions at arbitrary points
interpolateSolution	Interpolate PDE solution to arbitrary points

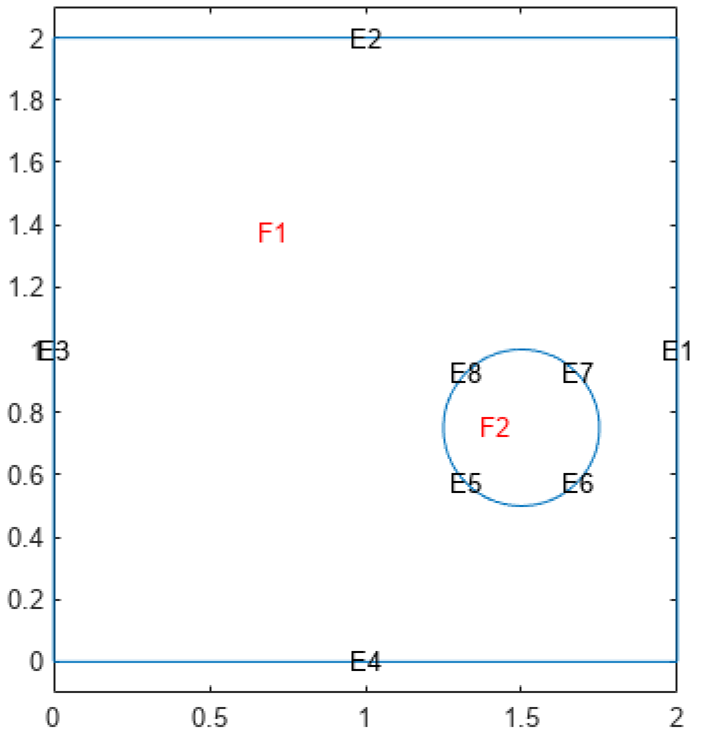
Examples

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.

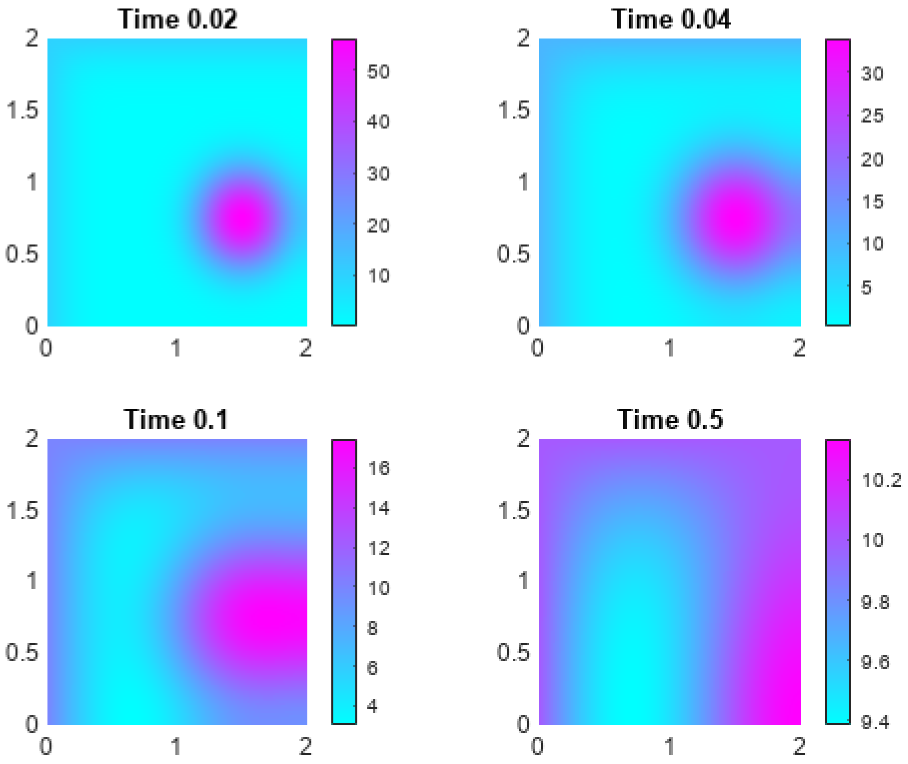
```
specifyCoefficients(model,"m",0,"d",1,"c",1,"a",0,"f",0);
```

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")
```



Version History

Introduced in R2016a

See Also

`interpolateSolution` | `evaluateGradient` | `evaluateCGradient` | `EigenResults` | `StationaryResults`

Topics

"Heat Transfer in Block with Cavity" on page 3-258
"Wave Equation on Square Domain" on page 3-298
"Solve Problems Using PDEModel Objects" on page 2-2

TransientThermalResults

Transient thermal solution and derived quantities

Description

A `TransientThermalResults` object contains the temperature and gradient 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. You can extract solution and gradient values for specified time indices from `Temperature`, `XGradients`, `YGradients`, and `ZGradients`.

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 a specified boundary, use `evaluateHeatRate`.

Creation

Solve a transient thermal problem using the `solve` function. This function returns a transient thermal solution as a `TransientThermalResults` object.

Properties

All Transient Thermal Models

`Mesh` — Finite element mesh
`FEMesh` object

This property is read-only.

Finite element mesh, returned as an FEMesh object.

`Temperature` — Temperature values at nodes
vector | matrix

This property is read-only.

Temperature values at nodes, returned as a vector or matrix.

Data Types: `double`

`SolutionTimes` — Solution times
real vector

This property is read-only.

Solution times, returned as a real vector. `SolutionTimes` is the same as the `tlist` input to `solve`.

Data Types: `double`

Non-Axisymmetric Models

`XGradients` — *x*-component of temperature gradient at nodes

vector | matrix

This property is read-only.

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

This property is read-only.

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

This property is read-only.

z-component of the temperature gradient at nodes, returned as a vector or matrix.

Data Types: `double`

Axisymmetric Models

`RGradients` — *r*-component of temperature gradient at nodes

vector | matrix

This property is read-only.

r-component of the temperature gradient at nodes, returned as a vector or matrix.

Data Types: `double`

`ZGradients` — *z*-component of temperature gradient at nodes for axisymmetric model

vector | matrix

This property is read-only.

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

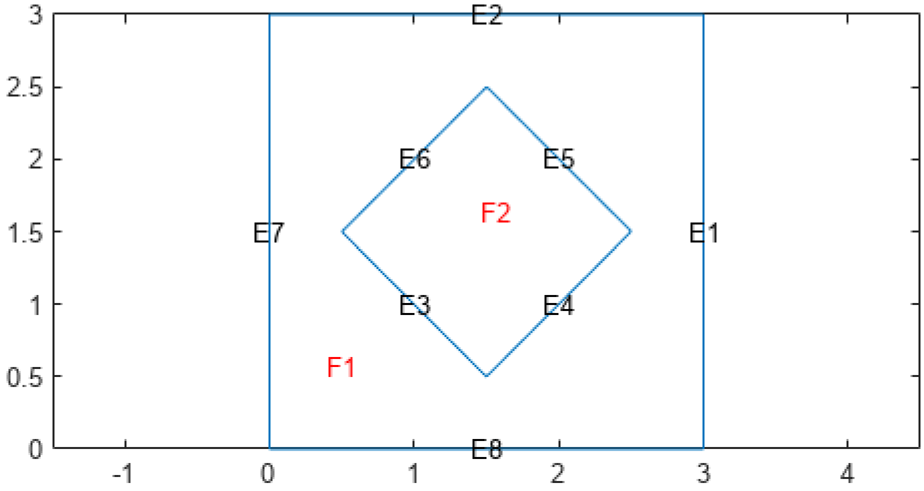
Solve a 2-D transient thermal problem.

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 these 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 these 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 a density of 4 W/m^2.

```
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);
```

Generate the mesh.

```
generateMesh(thermalmodel);
```

The dynamics for this problem are very fast. The temperature reaches a steady state in about 0.1 second. To capture the most active part of the dynamics, set the solution time to `logspace(-2,-1,10)`. This command returns 10 logarithmically spaced solution times between 0.01 and 0.1.

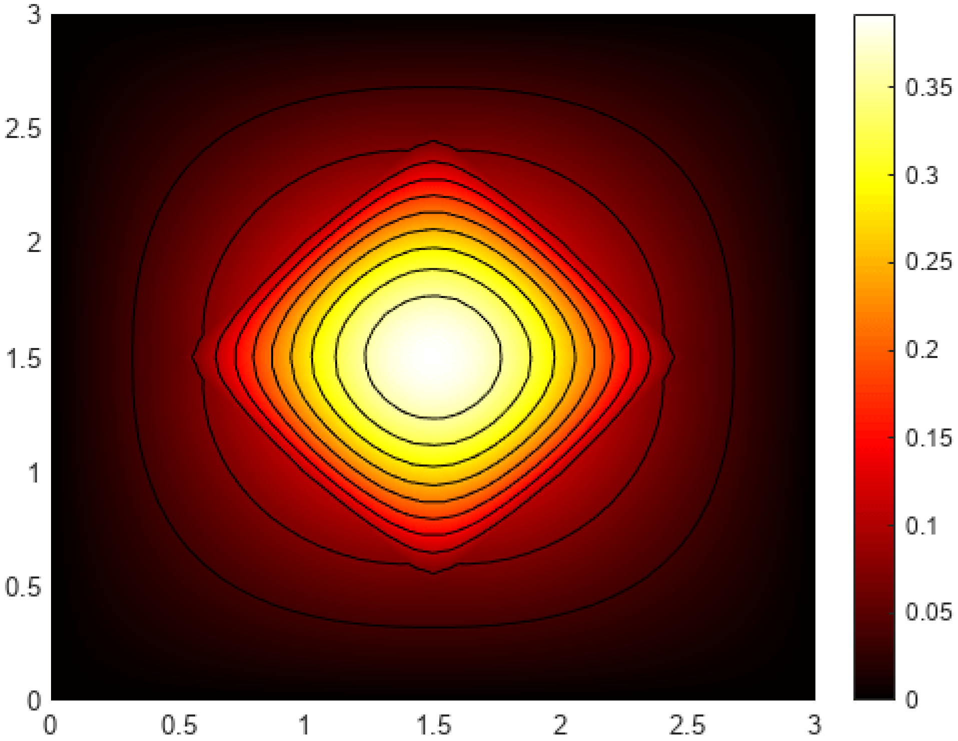
```
tlist = logspace(-2,-1,10);
```

Solve the equation.

```
thermalresults = solve(thermalmodel,tlist);
```

Plot the solution with isothermal lines by using a contour plot.

```
T = thermalresults.Temperature;
pdeplot(thermalmodel,"XYData",T(:,10),"Contour","on","ColorMap","hot")
```



Solution to Transient Axisymmetric Model

Analyze heat transfer in a rod with a circular cross-section and internal heat generation by simplifying a 3-D axisymmetric model to a 2-D model.

Create a transient thermal model for solving an axisymmetric problem.

```
thermalmodel = createpde("thermal","transient-axisymmetric");
```

The 2-D model is a rectangular strip whose *x*-dimension extends from the axis of symmetry to the outer surface and whose *y*-dimension extends over the actual length of the rod (from -1.5 m to 1.5 m). Create the geometry by specifying the coordinates of its four corners. For axisymmetric models, the toolbox assumes that the axis of rotation is the vertical axis passing through $r = 0$.

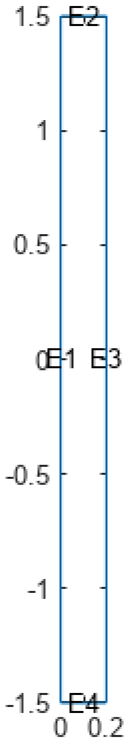
```
g = decsg([3 4 0 0 .2 .2 -1.5 1.5 1.5 -1.5]');
```

Include the geometry in the model.

```
geometryFromEdges(thermalmodel,g);
```

Plot the geometry with the edge labels.

```
figure
pdegplot(thermalmodel,"EdgeLabels","on")
axis equal
```



The rod is composed of a material with these thermal properties.

```
k = 40; % thermal conductivity, W/(m*C)
rho = 7800; % density, kg/m^3
cp = 500; % specific heat, W*s/(kg*C)
q = 20000; % heat source, W/m^3
```

Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",k,...
                                "MassDensity",rho,...
                                "SpecificHeat",cp);
```

Specify internal heat source and boundary conditions.

```
internalHeatSource(thermalmodel,q);
```

Define the boundary conditions. There is no heat transferred in the direction normal to the axis of symmetry (edge 1). You do not need to change the default boundary condition for this edge. Edge 2 is kept at a constant temperature $T = 100$ °C.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Specify the convection boundary condition on the outer boundary (edge 3). The surrounding temperature at the outer boundary is 100 °C, and the heat transfer coefficient is $50\,\mathrm{W/(m\cdot{}^\circ C)}$.

```
thermalBC(thermalmodel,"Edge",3,...
                       "ConvectionCoefficient",50,...
                       "AmbientTemperature",100);
```

The heat flux at the bottom of the rod (edge 4) is $5000\,\mathrm{W/m^2}$.

```
thermalBC(thermalmodel,"Edge",4,"HeatFlux",5000);
```

Specify that the Initial temperature in the rod is zero.

```
thermalIC(thermalmodel,0);
```

Generate the mesh.

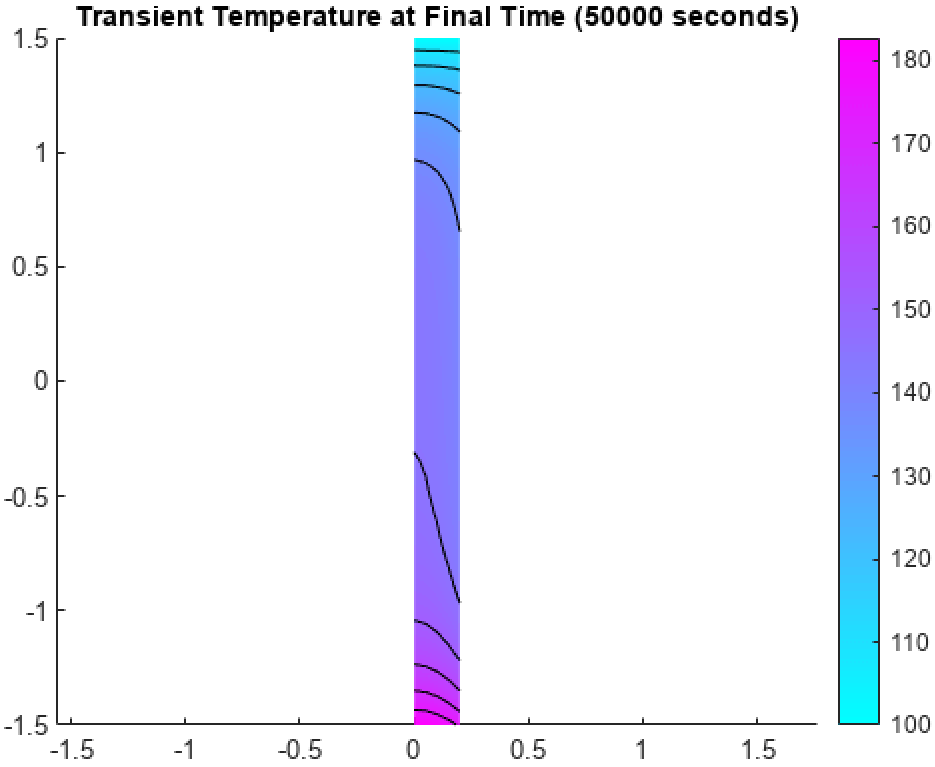
```
generateMesh(thermalmodel);
```

Compute the transient solution for solution times from `t = 0` to `t = 50000` seconds.

```
tfinal = 50000;
tlist = 0:100:tfinal;
result = solve(thermalmodel,tlist);
```

Plot the temperature distribution at `t = 50000` seconds.

```
T = result.Temperature;

figure
pdeplot(thermalmodel,"XYData",T(:,end), ...
                     "Contour","on")
axis equal
title(sprintf(['Transient Temperature ' ...
               'at Final Time (%g seconds)'],tfinal))
```



Version History

Introduced in R2017a

Axisymmetric analysis

`TransientThermalResults` now supports axisymmetric thermal results. Axisymmetric analysis simplifies 3-D structural and thermal problems to 2-D using their symmetry around the axis of rotation.

See Also

`evaluateHeatFlux` | `evaluateHeatRate` | `evaluateTemperatureGradient` | `interpolateTemperature` | `SteadyStateThermalResults` | `ModalThermalResults`

ModalThermalResults

Modal thermal solution

Description

A `ModalThermalResults` object contains the eigenvalues and eigenvector matrix of a thermal model, and average of snapshots used for proper orthogonal decomposition (POD).

Creation

Solve a modal thermal problem using the `solve` function. This function returns a modal thermal solution as a `ModalThermalResults` object.

Properties

`DecayRates` — Eigenvalues of thermal model
column vector

This property is read-only.

Eigenvalues of a thermal model, returned as a column vector.

Data Types: `double`

`ModeShapes` — Eigenvector matrix
matrix

This property is read-only.

Eigenvector matrix, returned as a matrix.

Data Types: `double`

`SnapshotsAverage` — Average of snapshots used for POD
column vector

This property is read-only.

Average of snapshots used for POD, returned as a column vector.

Data Types: `double`

`ModeType` — Type of modes
`"EigenModes"` | `"PODModes"`

This property is read-only.

Type of modes, returned as `"EigenModes"` or `"PODModes"`.

Data Types: `string`

`Mesh` — Finite element mesh

`FEMesh` object

This property is read-only.

Finite element mesh, returned as an `FEMesh` object. For details, see FEMesh.

Examples

Solution to Transient Thermal Model Using Modal Superposition Method

Solve a transient thermal problem by first obtaining mode shapes for a particular decay range and then using the modal superposition method.

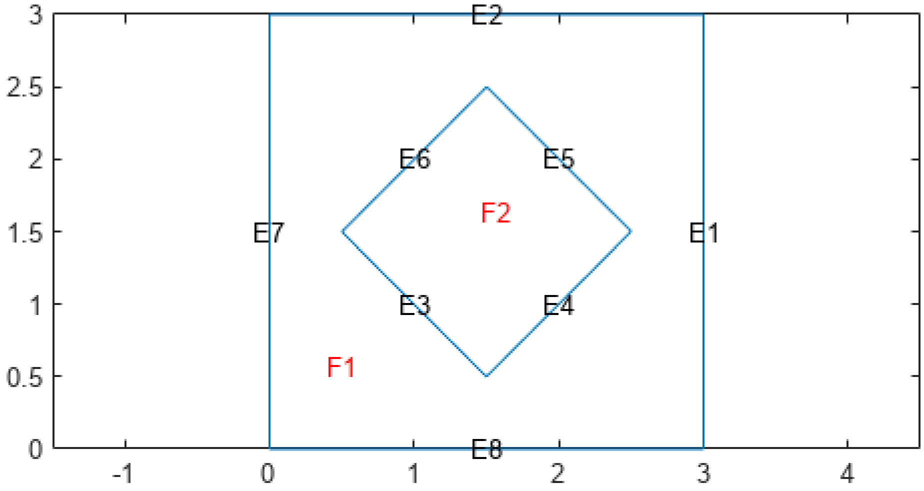
Modal Decomposition

First, create a modal thermal model.

```
thermalmodel = createpde("thermal","modal");
```

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 these 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 these 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 a density of 4 W/m^2.

```
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);
```

Generate the mesh.

```
generateMesh(thermalmodel);
```

Compute eigenmodes of the thermal model in the decay range [100,10000] s^{-1}.

```
RModal = solve(thermalmodel,"DecayRange",[100,10000])
```

```
RModal = 
  ModalThermalResults with properties:

    DecayRates: [164x1 double]
    ModeShapes: [1481x164 double]
      ModeType: "EigenModes"
          Mesh: [1x1 FEMesh]
```

Transient Analysis

Knowing the mode shapes, you can now use the modal superposition method to solve the transient thermal problem. First, switch the thermal model analysis type to transient.

```
thermalmodel.AnalysisType = "transient";
```

The dynamics for this problem are very fast. The temperature reaches a steady state in about 0.1 second. To capture the most active part of the dynamics, set the solution time to `logspace(-2,-1,100)`. This command returns 100 logarithmically spaced solution times between 0.01 and 0.1.

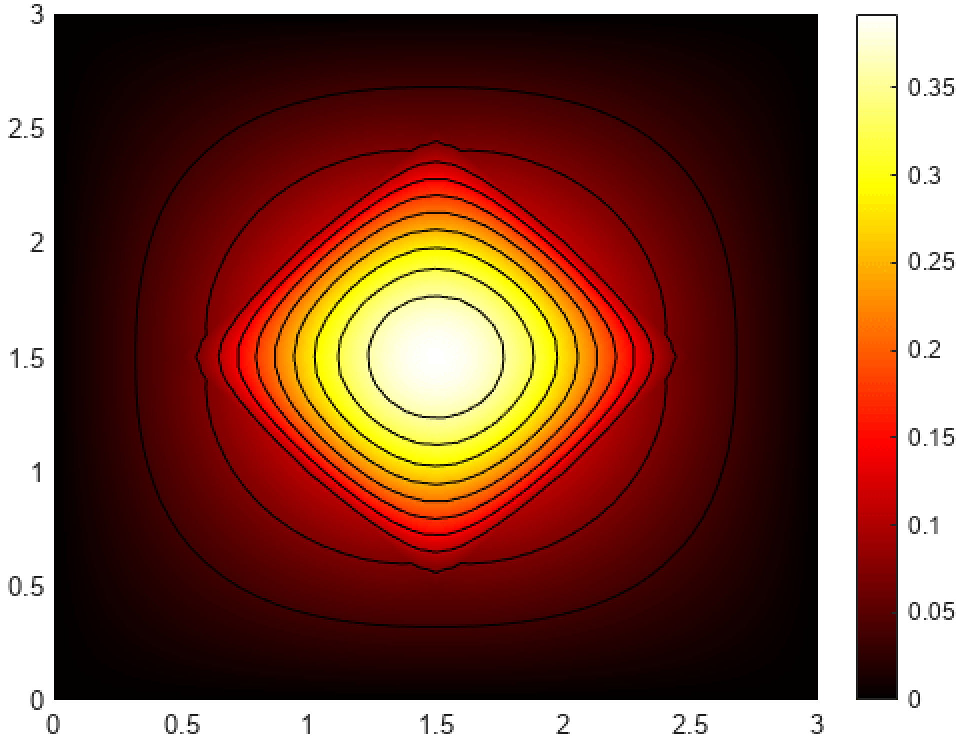
```
tlist = logspace(-2,-1,10);
```

Solve the equation.

```
Rtransient = solve(thermalmodel,tlist,"ModalResults",RModal);
```

Plot the solution with isothermal lines by using a contour plot.

```
T = Rtransient.Temperature;
pdeplot(thermalmodel,"XYData",T(:,end), ...
                     "Contour","on", ...
                     "ColorMap","hot")
```



Snapshots for Proper Orthogonal Decomposition

Obtain POD modes of a linear thermal model using several instances of the transient solution (snapshots).

Create a transient thermal model.

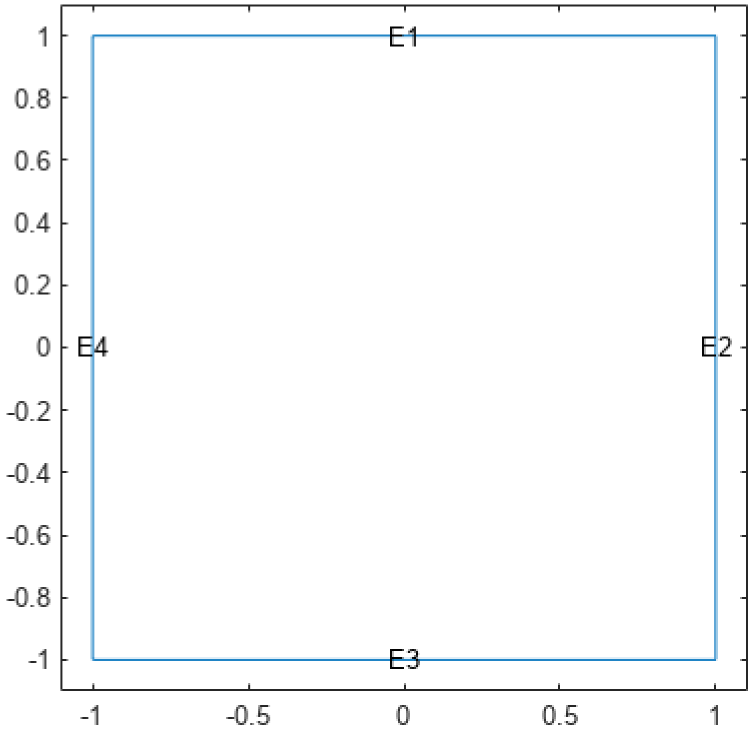
```
thermalmodel = createpde("thermal","transient");
```

Create a unit square geometry and include it in the model.

```
geometryFromEdges(thermalmodel,@squareg);
```

Plot the geometry, displaying edge labels.

```
pdegplot(thermalmodel,"EdgeLabels","on")
xlim([-1.1 1.1])
ylim([-1.1 1.1])
```



Specify the thermal conductivity, mass density, and specific heat of the material.

```
thermalProperties(thermalmodel,"ThermalConductivity",400, ...
                               "MassDensity",1300, ...
                               "SpecificHeat",600);
```

Set the temperature on the right edge to `100`.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100);
```

Set an initial value of `0` for the temperature.

```
thermalIC(thermalmodel,0);
```

Generate a mesh.

```
generateMesh(thermalmodel);
```

Solve the model for three different values of heat source and collect snapshots.

```
tlist = 0:10:600;
snapShotIDs = [1:10 59 60 61];
Tmatrix = [];

heatVariation = [10000 15000 20000];
for q = heatVariation
    internalHeatSource(thermalmodel,q);
    results = solve(thermalmodel,tlist);
```

```
    Tmatrix = [Tmatrix,results.Temperature(:,snapShotIDs)];
end
```

Switch the thermal model analysis type to modal.

```
thermalmodel.AnalysisType = "modal";
```

Compute the POD modes.

```
RModal = solve(thermalmodel,"Snapshots",Tmatrix)
```

```
RModal = 
  ModalThermalResults with properties:

           DecayRates: [6x1 double]
           ModeShapes: [1541x6 double]
    SnapshotsAverage: [1541x1 double]
             ModeType: "PODModes"
                 Mesh: [1x1 FEMesh]
```

Version History

Introduced in R2022a

See Also

`solve` | `SteadyStateThermalResults` | `TransientThermalResults` | `ModalStructuralResults`

thermalBC

Package: `pde`

Specify boundary conditions for a thermal model

Syntax

```
thermalBC(thermalmodel,RegionType,RegionID,"Temperature",Tval)
thermalBC(thermalmodel,RegionType,RegionID,"HeatFlux",HFval)
thermalBC(thermalmodel,RegionType,RegionID,"ConvectionCoefficient",
CCval,"AmbientTemperature",ATval)
thermalBC(thermalmodel,RegionType,RegionID,"Emissivity",
REval,"AmbientTemperature",ATval)
thermalBC( ___ ,"Label",labeltext)
thermalBC = thermalBC( ___ )
```

Description

`thermalBC(thermalmodel,RegionType,RegionID,"Temperature",Tval)` adds a temperature boundary condition to `thermalmodel`. The boundary condition applies to regions of type `RegionType` with ID numbers in `RegionID`.

`thermalBC(thermalmodel,RegionType,RegionID,"HeatFlux",HFval)` adds a heat flux boundary condition to `thermalmodel`. The boundary condition applies to regions of type `RegionType` with ID numbers in `RegionID`.

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(thermalmodel,RegionType,RegionID,"ConvectionCoefficient",CCval,"AmbientTemperature",ATval)` adds a convection boundary condition to `thermalmodel`. The boundary condition applies to regions of type `RegionType` with ID numbers in `RegionID`.

`thermalBC(thermalmodel,RegionType,RegionID,"Emissivity",REval,"AmbientTemperature",ATval)` adds a radiation boundary condition to `thermalmodel`. The boundary condition applies to regions of type `RegionType` with ID numbers in `RegionID`.

`thermalBC( ___ ,"Label",labeltext)` adds a label for the thermal boundary condition to be used by the `linearizeInput` function. This function lets you pass thermal boundary conditions to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

`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'
                    Label: []
```

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'
                    Label: []
```

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'
                    Label: []
```

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'
                    Label: []
```

Specify Nonconstant Thermal Boundary Conditions

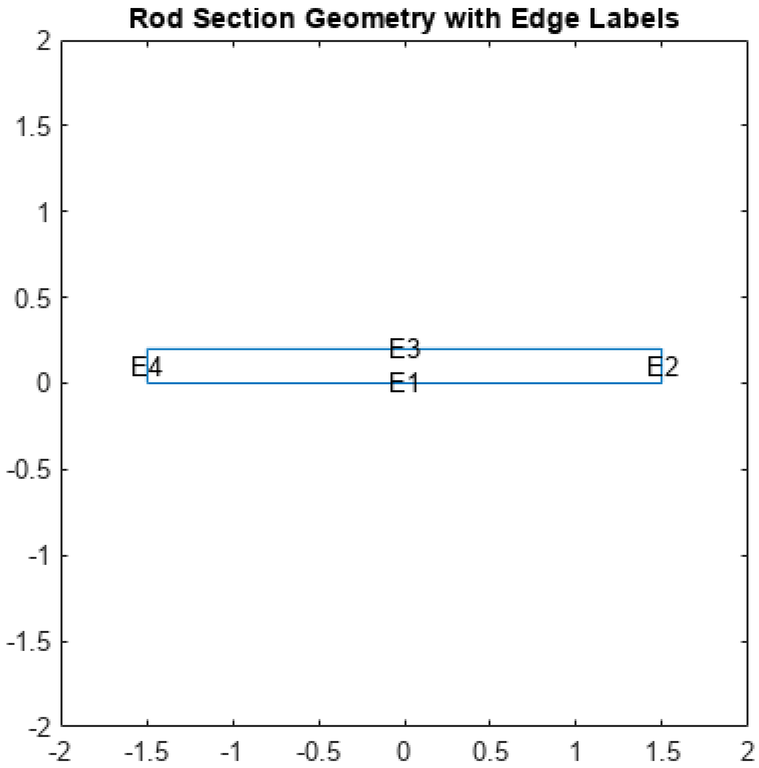
Use function handles to specify thermal boundary conditions that depend on coordinates.

Create a thermal model for transient analysis and include the geometry. The geometry is a rod with a circular cross section. The 2-D model is a rectangular strip whose *y*-dimension extends from the axis of symmetry to the outer surface, and whose *x*-dimension extends over the actual length of the rod.

```
thermalmodel = createpde("thermal","transient");
g = decsg([3 4 -1.5 1.5 1.5 -1.5 0 0 .2 .2]');
geometryFromEdges(thermalmodel,g);
```

Plot the geometry.

```
figure
pdegplot(thermalmodel,"EdgeLabels","on");
xlim([-2 2]);
ylim([-2 2]);
title 'Rod Section Geometry with Edge Labels'
```



Assume that 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.

Define the boundary conditions for the model. The edge at $y = 0$ (edge 1) is along the axis of symmetry. No heat is transferred in the direction normal to this edge. This boundary is modeled as an insulated boundary, by default.

The temperature at the right end of the rod (edge 2) is a fixed temperature, T = 100 C. Specify the boundary condition for edge 2 as follows.

```
thermalBC(thermalmodel,"Edge",2,"Temperature",100)
```

```
ans =
  ThermalBC with properties:

               RegionType: 'Edge'
                 RegionID: 2
              Temperature: 100
                 HeatFlux: []
    ConvectionCoefficient: []
               Emissivity: []
```

```
       AmbientTemperature: []
               Vectorized: 'off'
                    Label: []
```

The convection coefficient for the outer surface of the rod (edge 3) depends on the *y*-coordinate, 50*y*. Specify the boundary condition for this edge as follows.

```
outerCC = @(location,~) 50*location.y;
thermalBC(thermalmodel,"Edge",3,...
                       "ConvectionCoefficient",outerCC,...
                       "AmbientTemperature",100)
```

```
ans = 
  ThermalBC with properties:

               RegionType: 'Edge'
                 RegionID: 3
              Temperature: []
                 HeatFlux: []
    ConvectionCoefficient: @(location,~)50*location.y
               Emissivity: []
       AmbientTemperature: 100
               Vectorized: 'off'
                    Label: []
```

The heat flux at the left end of the rod (edge 4) is also a function of the *y*-coordinate, 5000*y*. Specify the boundary condition for this edge as follows.

```
leftHF = @(location,~) 5000*location.y;
thermalBC(thermalmodel,"Edge",4,"HeatFlux",leftHF)
```

```
ans = 
  ThermalBC with properties:

               RegionType: 'Edge'
                 RegionID: 4
              Temperature: []
                 HeatFlux: @(location,~)5000*location.y
    ConvectionCoefficient: []
               Emissivity: []
       AmbientTemperature: []
               Vectorized: 'off'
                    Label: []
```

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","steadystate")`

`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 by using `pdegplot` with the `"FaceLabels"` (3-D) or `"EdgeLabels"` (2-D) value set to `"on"`.

Example: `thermalBC(thermalmodel,"Edge",2:5,"Temperature",72)`

Data Types: `double`

`Tval` — Temperature boundary condition

number | function handle

Temperature boundary condition, specified as a number or a function handle. Use a function handle to specify the temperature that depends on space and time. For details, see “More About” on page 5-1418.

Example: `thermalBC(thermalmodel,"Face",1,"Temperature",72)`

Data Types: `double` | `function_handle`

`HFval` — 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 the heat flux that depends on space and time. For details, see “More About” on page 5-1418.

Example: `thermalBC(thermalmodel,"Face",[1,3],"HeatFlux",20)`

Data Types: `double` | `function_handle`

`CCval` — 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 the convection coefficient that depends on space and time. For details, see “More About” on page 5-1418.

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`

`REval` — 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 the radiation emissivity that depends on space and time. For details, see “More About” on page 5-1418.

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`

`ATval` — 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`

`labeltext` — Label for thermal boundary condition
character vector | string

Label for the thermal boundary condition, specified as a character vector or a string.

Data Types: `char` | `string`

Output Arguments

`thermalBC` — Handle to thermal boundary condition
`ThermalBC` object

Handle to thermal boundary condition, returned as a `ThermalBC` object. See ThermalBC Properties.

`thermalBC` associates the thermal boundary condition with the geometric region.

More About

Specifying Nonconstant Parameters of a Thermal Model

Use a function handle to specify these thermal parameters when they depend on space, temperature, and time:

- Thermal conductivity of the material
- Mass density of the material
- Specific heat of the material
- Internal heat source
- Temperature on the boundary

- Heat flux through the boundary
- Convection coefficient on the boundary
- Radiation emissivity coefficient on the boundary
- Initial temperature (can depend on space only)

For example, use function handles to specify the thermal conductivity, internal heat source, convection coefficient, and initial temperature for this model.

```
thermalProperties(model,"ThermalConductivity", ...
                        @myfunConductivity)
internalHeatSource(model,"Face",2,@myfunHeatSource)
thermalBC(model,"Edge",[3,4], ...
                "ConvectionCoefficient",@myfunBC, ...
                "AmbientTemperature",27)
thermalIC(model,@myfunIC)
```

For all parameters, except the initial temperature, the function must be of the form:

```
function thermalVal = myfun(location,state)
```

For the initial temperature the function must be of the form:

```
function thermalVal = myfun(location)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments (or one argument if the function specifies the initial temperature).

- `location` — A structure containing these fields:
  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes these data in the `location` structure:

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, *r*-component of the normal vector at the evaluation point or points
- `state` — A structure containing these fields for transient or nonlinear problems:
  - `state.u` — Temperatures at the corresponding points of the location structure
  - `state.ux` — Estimates of the *x*-component of temperature gradients at the corresponding points of the location structure
  - `state.uy` — Estimates of the *y*-component of temperature gradients at the corresponding points of the location structure

  - `state.uz` — For a 3-D or an axisymmetric geometry, estimates of the *z*-component of temperature gradients at the corresponding points of the location structure
  - `state.ur` — For an axisymmetric geometry, estimates of the *r*-component of temperature gradients at the corresponding points of the location structure
  - `state.time` — Time at evaluation points

Thermal material properties (thermal conductivity, mass density, and specific heat) and internal heat source get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID
- `state.u`, `state.ux`, `state.uy`, `state.uz`, `state.r`, `state.time`

Boundary conditions (temperature on the boundary, heat flux, convection coefficient, and radiation emissivity coefficient) get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`
- `state.u`, `state.time`

Initial temperature gets the following data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

For all thermal parameters, except for thermal conductivity, your function must return a row vector `thermalVal` with the number of columns equal to the number of evaluation points, for example, `M = length(location.y)`.

For thermal conductivity, your function must return a matrix `thermalVal` with number of rows equal to 1, `Ndim`, `Ndim*(Ndim+1)/2`, or `Ndim*Ndim`, where `Ndim` is 2 for 2-D problems and 3 for 3-D problems. The number of columns must equal the number of evaluation points, for example, `M = length(location.y)`. For details about dimensions of the matrix, see “c Coefficient for specifyCoefficients” on page 2-92.

If properties depend on the time or temperature, 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 time dependent by passing `NaN` state values and looking for returned `NaN` values.

Additional Arguments in Functions for Nonconstant Thermal Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
thermalVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
thermalBC(model,"Edge",3,"Temperature",thermalVal)

thermalVal = @(location) myfunWithAdditionalArgs(location,arg1,arg2...)
thermalIC(model,thermalVal)
```

Version History

Introduced in R2017a

Label to extract sparse linear models for use with Control System Toolbox

Now you can add a label for thermal boundary conditions to be used by the `linearizeInput` function. This function lets you pass thermal boundary conditions to the `linearize` function that extracts sparse linear models for use with Control System Toolbox.

See Also

`thermalProperties` | `internalHeatSource` | `thermalIC` | `applyBoundaryCondition` | ThermalBC Properties

Topics

"Heat Conduction in Multidomain Geometry with Nonuniform Heat Flux" on page 3-270

thermalIC

Package: `pde`

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,0)
```

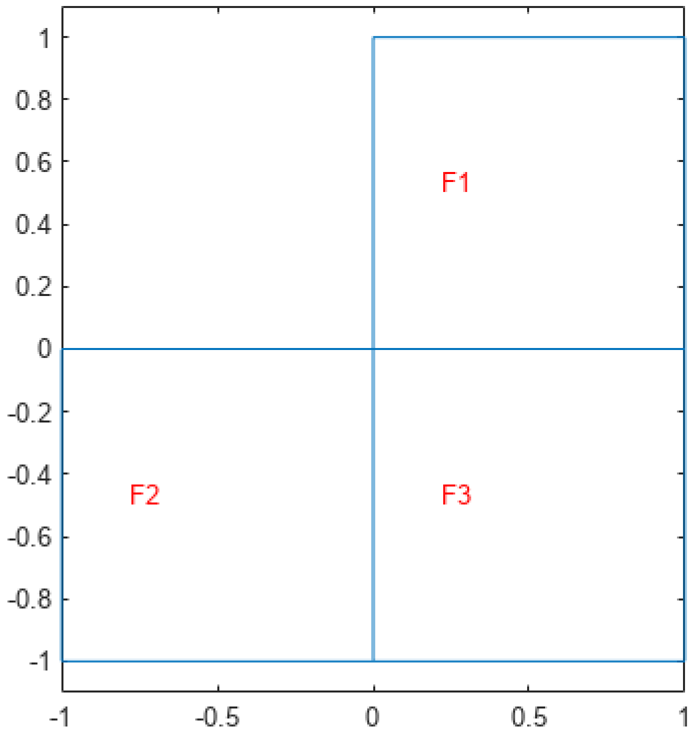
```
ans =
  GeometricThermalICs with properties:

            RegionType: 'face'
              RegionID: [1 2 3]
    InitialTemperature: 0
```

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)
```

```
ans = 
  GeometricThermalICs with properties:

             RegionType: 'face'
               RegionID: 1
    InitialTemperature: 0
```

```
thermalIC(thermalModel,10,"Face",2)
```

```
ans = 
  GeometricThermalICs with properties:

             RegionType: 'face'
               RegionID: 2
    InitialTemperature: 10
```

```
thermalIC(thermalModel,75,"Face",3)
```

```
ans = 
  GeometricThermalICs with properties:

             RegionType: 'face'
               RegionID: 3
    InitialTemperature: 75
```

Nonconstant Initial Temperature

Use a function handle to specify an initial temperature that depends on coordinates.

Create a thermal model for transient analysis and include the geometry. The geometry is a rod with a circular cross section. The 2-D model is a rectangular strip whose *y*-dimension extends from the axis of symmetry to the outer surface, and whose *x*-dimension extends over the actual length of the rod.

```
thermalmodel = createpde("thermal","transient");
g = decsg([3 4 -1.5 1.5 1.5 -1.5 0 0 .2 .2]');
geometryFromEdges(thermalmodel,g);
```

Set the initial temperature in the rod to be dependent on the *y*-coordinate, for example, $10^3(0.2-y^2)$.

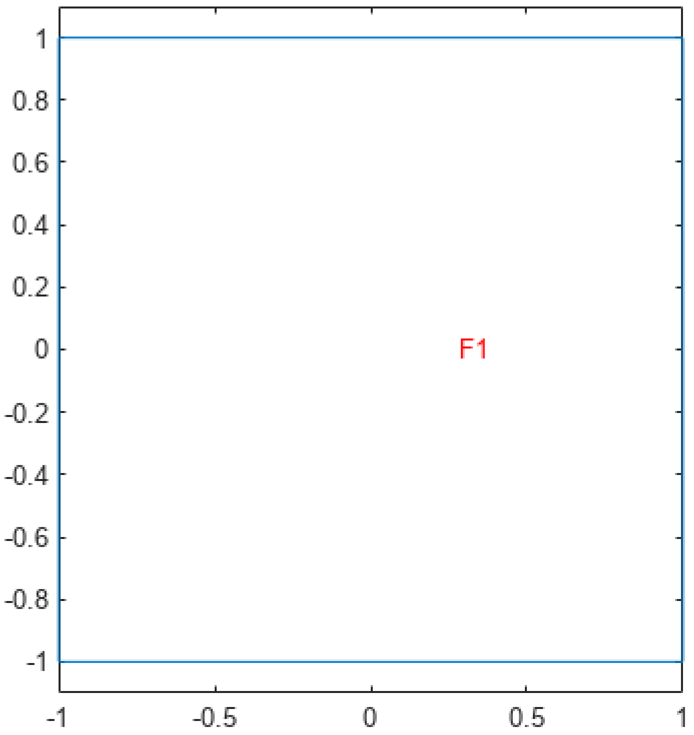
```
T0 = @(location)10^3*(0.2 - location.y.^2);
thermalIC(thermalmodel,T0)
```

```
ans = 
  GeometricThermalICs with properties:

             RegionType: 'face'
               RegionID: 1
    InitialTemperature: @(location)10^3*(0.2-location.y.^2)
```

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
```

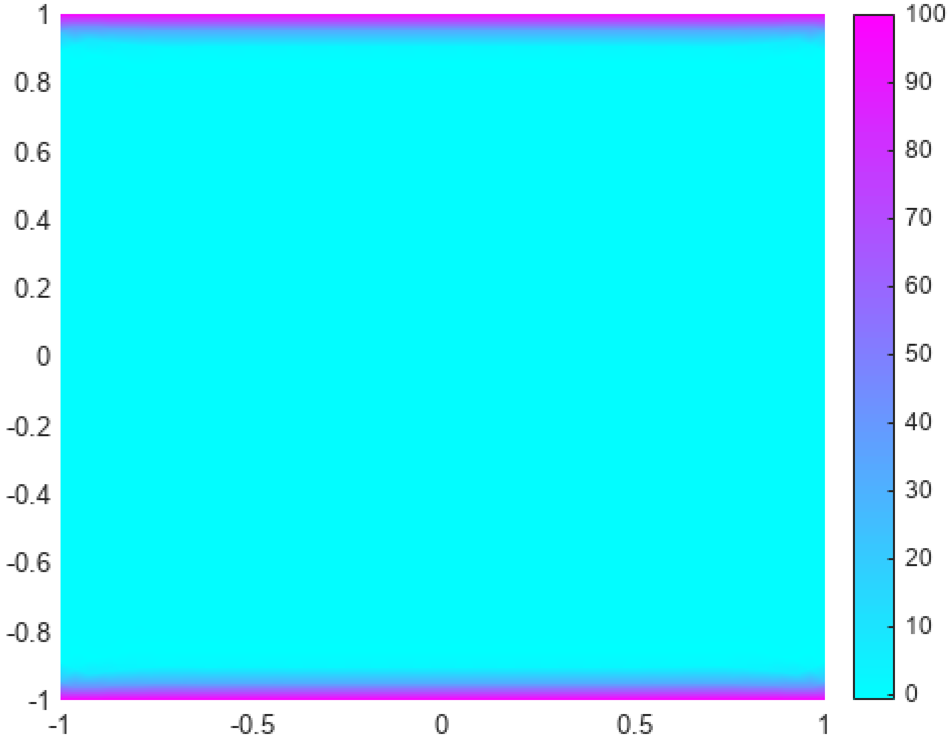


Specify material properties and internal heat source, and set boundary conditions and initial conditions.

```
thermalProperties(thermalmodel, ...
                  "ThermalConductivity",40,...
                  "MassDensity",7800,...
                  "SpecificHeat",500);

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 = 0:10:100;
result1 = solve(thermalmodel,tlist);
pdeplot(thermalmodel,"XYData",result1.Temperature(:,end))
```

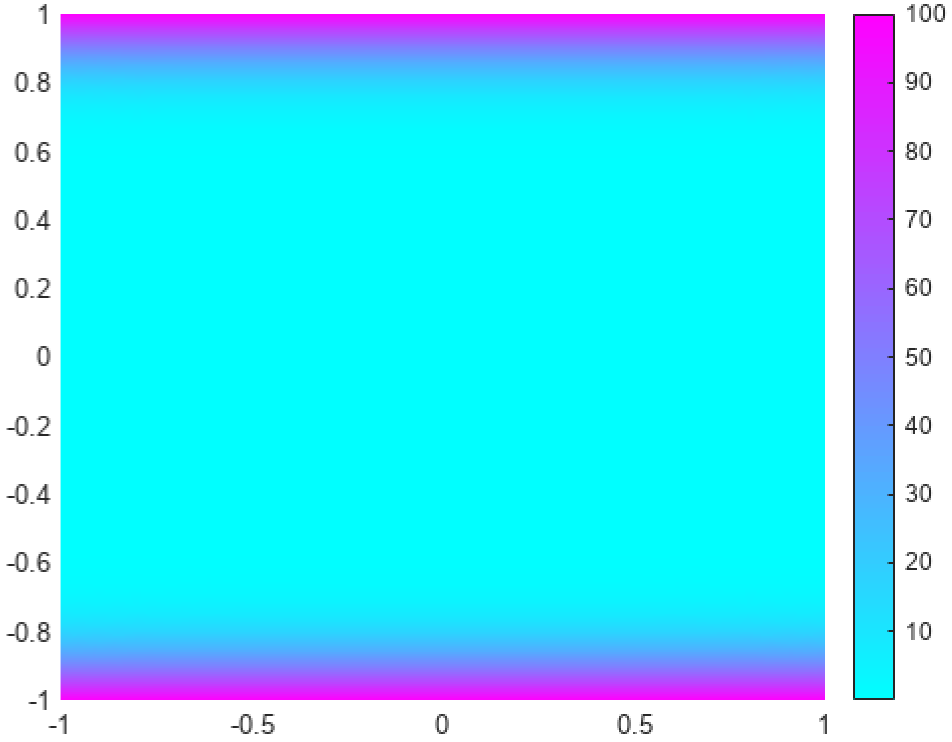


Now, resume the analysis and solve the problem for times from 100 to 1000 seconds. Use the previously obtained solution for 100 seconds as an initial condition. Since 10 seconds is the last element in `tlist`, you do not need to specify the solution time index. By default, `thermalIC` uses the last solution index.

```
thermalIC(thermalmodel,result1);
```

Solve the problem and plot the solution.

```
result2 = solve(thermalmodel,100:100:1000);
pdeplot(thermalmodel,"XYData",result2.Temperature(:,end))
```



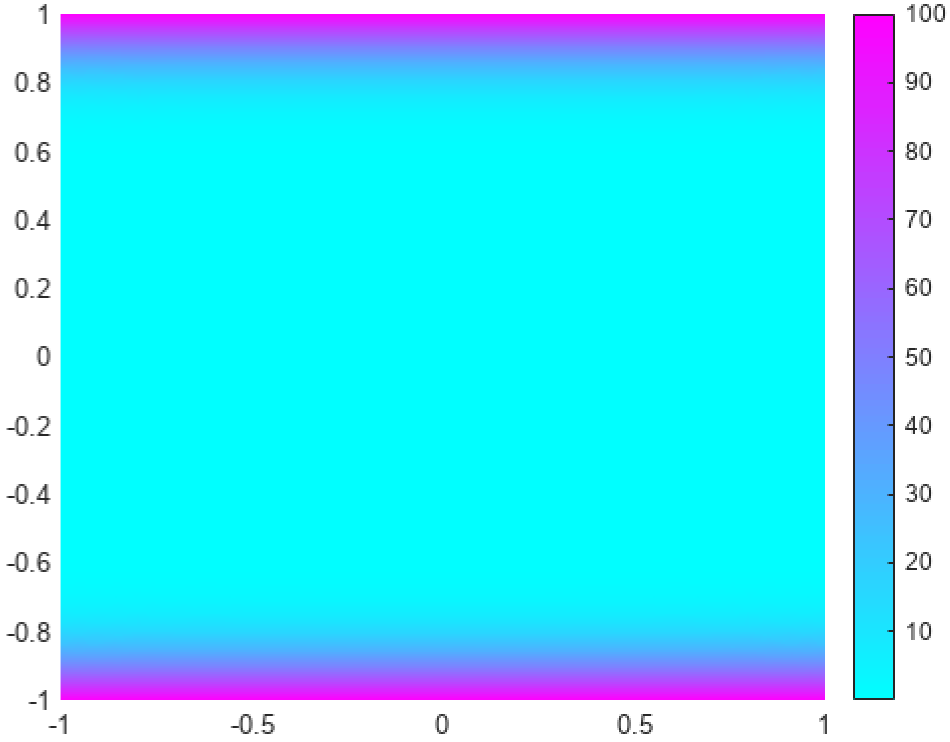
To use the previously obtained solution for a particular solution time instead of the last one, specify the solution time index as a third parameter of `thermalIC`. For example, use the solution at time 50 seconds, which is the 6th element in `tlist`.

```
tlist(6)
```

```
ans = 50
```

```
thermalIC(thermalmodel,result1,6);
```

```
result2 = solve(thermalmodel,50:100:1000);
pdeplot(thermalmodel,"XYData",result2.Temperature(:,end))
```



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","steadystate")`

`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 initial temperature. For details, see “More About” on page 5-1429.

Data Types: `double` | `function_handle`

`RegionType` — Geometric region type

`"Vertex"` | `"Edge"` | `"Face"` | `"Cell"` for a 3-D model only

Geometric region type, specified as `"Vertex"`, `"Edge"`, `"Face"`, or `"Cell"` for a 3-D model. For a 2-D model, use `"Vertex"`, `"Edge"`, or `"Face"`.

Example: `thermalIC(thermalmodel,10,"Face",1)`

Data Types: `char` | `string`

`RegionID` — Geometric region ID
vector of positive integers

Geometric region ID, specified as a vector of positive integers. Find the region IDs by using `pdegplot`.

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
`GeometricThermalICs` object | `NodalThermalICs` object

Handle to initial condition, returned as a `GeometricThermalICs` or `NodalThermalICs` object. See GeometricThermalICs Properties and NodalThermalICs Properties.

`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.

More About

Specifying Nonconstant Parameters of a Thermal Model

Use a function handle to specify these thermal parameters when they depend on space, temperature, and time:

- Thermal conductivity of the material
- Mass density of the material
- Specific heat of the material
- Internal heat source
- Temperature on the boundary
- Heat flux through the boundary
- Convection coefficient on the boundary
- Radiation emissivity coefficient on the boundary

- Initial temperature (can depend on space only)

For example, use function handles to specify the thermal conductivity, internal heat source, convection coefficient, and initial temperature for this model.

```
thermalProperties(model,"ThermalConductivity", ...
                        @myfunConductivity)
internalHeatSource(model,"Face",2,@myfunHeatSource)
thermalBC(model,"Edge",[3,4], ...
                "ConvectionCoefficient",@myfunBC, ...
                "AmbientTemperature",27)
thermalIC(model,@myfunIC)
```

For all parameters, except the initial temperature, the function must be of the form:

```
function thermalVal = myfun(location,state)
```

For the initial temperature the function must be of the form:

```
function thermalVal = myfun(location)
```

The solver computes and populates the data in the `location` and `state` structure arrays and passes this data to your function. You can define your function so that its output depends on this data. You can use any names instead of `location` and `state`, but the function must have exactly two arguments (or one argument if the function specifies the initial temperature).

- `location` — A structure containing these fields:
  - `location.x` — The *x*-coordinate of the point or points
  - `location.y` — The *y*-coordinate of the point or points
  - `location.z` — For a 3-D or an axisymmetric geometry, the *z*-coordinate of the point or points
  - `location.r` — For an axisymmetric geometry, the *r*-coordinate of the point or points

  Furthermore, for boundary conditions, the solver passes these data in the `location` structure:

  - `location.nx` — *x*-component of the normal vector at the evaluation point or points
  - `location.ny` — *y*-component of the normal vector at the evaluation point or points
  - `location.nz` — For a 3-D or an axisymmetric geometry, *z*-component of the normal vector at the evaluation point or points
  - `location.nr` — For an axisymmetric geometry, *r*-component of the normal vector at the evaluation point or points
- `state` — A structure containing these fields for transient or nonlinear problems:
  - `state.u` — Temperatures at the corresponding points of the location structure
  - `state.ux` — Estimates of the *x*-component of temperature gradients at the corresponding points of the location structure
  - `state.uy` — Estimates of the *y*-component of temperature gradients at the corresponding points of the location structure
  - `state.uz` — For a 3-D or an axisymmetric geometry, estimates of the *z*-component of temperature gradients at the corresponding points of the location structure
  - `state.ur` — For an axisymmetric geometry, estimates of the *r*-component of temperature gradients at the corresponding points of the location structure

  - `state.time` — Time at evaluation points

Thermal material properties (thermal conductivity, mass density, and specific heat) and internal heat source get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID
- `state.u`, `state.ux`, `state.uy`, `state.uz`, `state.r`, `state.time`

Boundary conditions (temperature on the boundary, heat flux, convection coefficient, and radiation emissivity coefficient) get these data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- `location.nx`, `location.ny`, `location.nz`, `location.nr`
- `state.u`, `state.time`

Initial temperature gets the following data from the solver:

- `location.x`, `location.y`, `location.z`, `location.r`
- Subdomain ID

For all thermal parameters, except for thermal conductivity, your function must return a row vector `thermalVal` with the number of columns equal to the number of evaluation points, for example, `M = length(location.y)`.

For thermal conductivity, your function must return a matrix `thermalVal` with number of rows equal to 1, `Ndim`, `Ndim*(Ndim+1)/2`, or `Ndim*Ndim`, where `Ndim` is 2 for 2-D problems and 3 for 3-D problems. The number of columns must equal the number of evaluation points, for example, `M = length(location.y)`. For details about dimensions of the matrix, see “c Coefficient for specifyCoefficients” on page 2-92.

If properties depend on the time or temperature, 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 time dependent by passing `NaN` state values and looking for returned `NaN` values.

Additional Arguments in Functions for Nonconstant Thermal Parameters

To use additional arguments in your function, wrap your function (that takes additional arguments) with an anonymous function that takes only the `location` and `state` arguments. For example:

```
thermalVal = ...
@(location,state) myfunWithAdditionalArgs(location,state,arg1,arg2...)
thermalBC(model,"Edge",3,"Temperature",thermalVal)

thermalVal = @(location) myfunWithAdditionalArgs(location,arg1,arg2...)
thermalIC(model,thermalVal)
```

Version History

Introduced in R2017a

See Also

`thermalProperties` | `internalHeatSource` | `thermalBC` | `setInitialConditions` | GeometricThermalICs Properties | NodalThermalICs Properties

tri2grid

(Not recommended) Interpolate from PDE triangular mesh to rectangular grid

---

Note `tri2grid` is not recommended. Use `interpolateSolution` instead. For more information, see “Compatibility Considerations”.

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 interpolated function values `uxy` over the grid defined by the vectors `x` and `y` using the function values `u` on the triangular mesh defined by `p` and `t`.

`tri2grid` uses linear interpolation in the triangle containing the grid point.

`[uxy,tn,a2,a3] = tri2grid(p,t,u,x,y)` also returns the index `tn` of the triangle containing each grid point, and interpolation coefficients `a2` and `a3`.

`uxy = tri2grid(p,t,u,tn,a2,a3)` uses the values `tn`, `a2`, and `a3` returned by prior `tri2grid` call using the previous syntax to interpolate `u` to the same grid as in the prior call. This syntax is efficient for interpolating several functions to the same grid, such as interpolating hyperbolic or parabolic solutions at multiple times.

Examples

Interpolate Solution to Rectangular Grid

Generate a triangular mesh of the L-shaped membrane. The geometry of the L-shaped membrane is described in the file `lshapeg`.

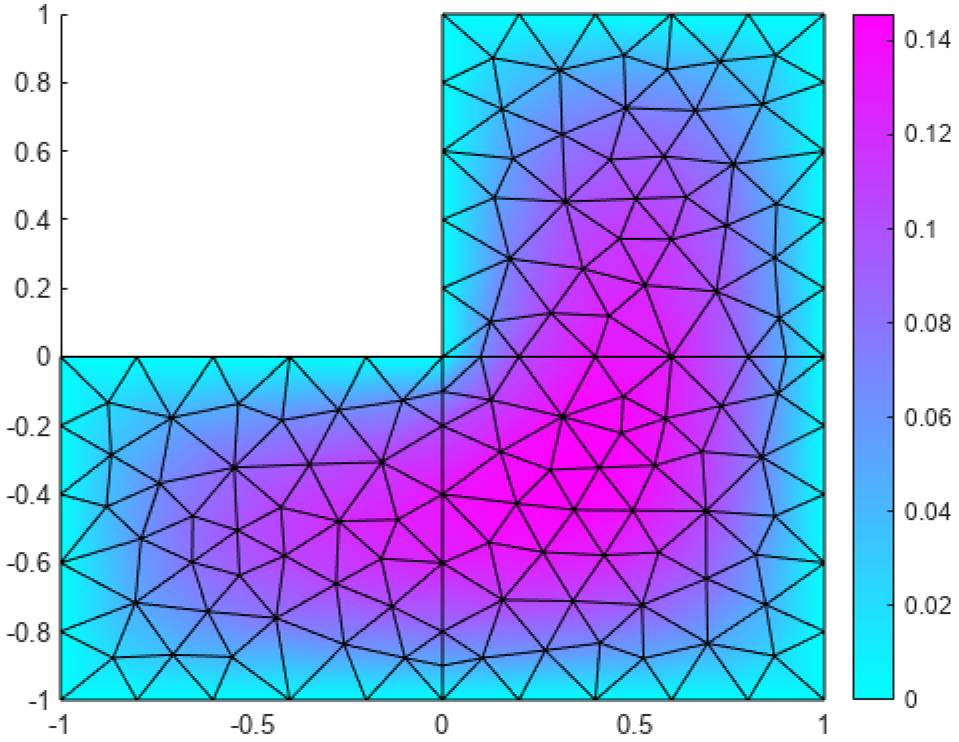
```
[p,e,t] = initmesh('lshapeg');
```

Solve an elliptic PDE on the L-shaped geometry.

```
u = assempde('lshapeb',p,e,t,1,0,1);
```

Plot the solution and the mesh.

```
pdeplot(p,e,t,'XYData',u,'Mesh','on')
```

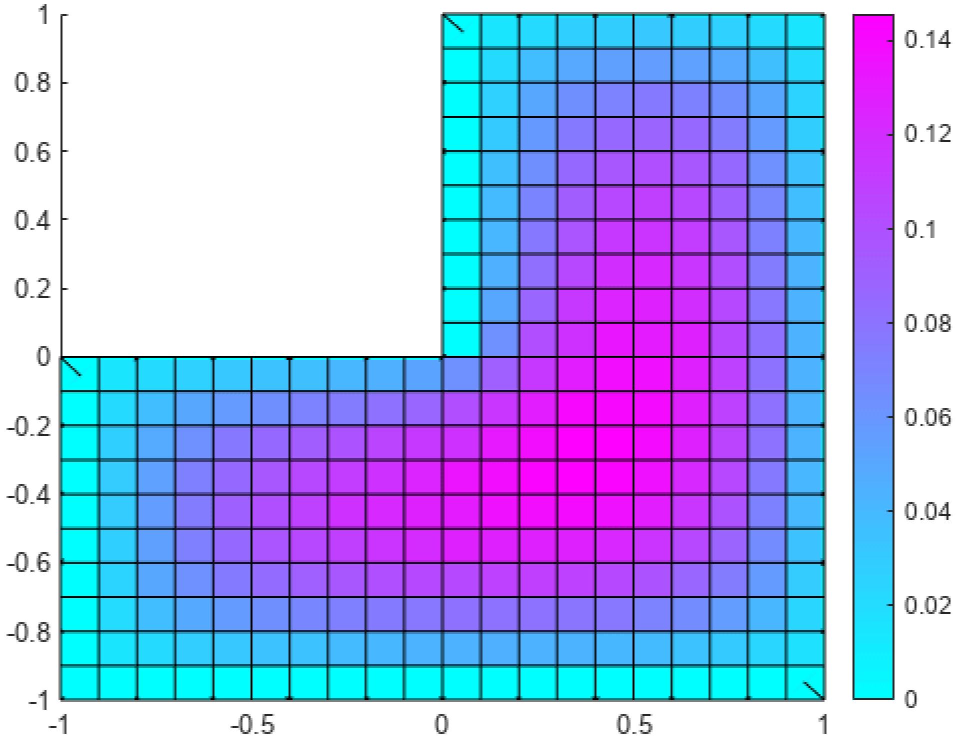


Interpolate the solution to the rectangular grid defined by the vectors `x` and `y`.

```
x = -1:0.1:1;
y = x;
uxy = tri2grid(p,t,u,x,y);
```

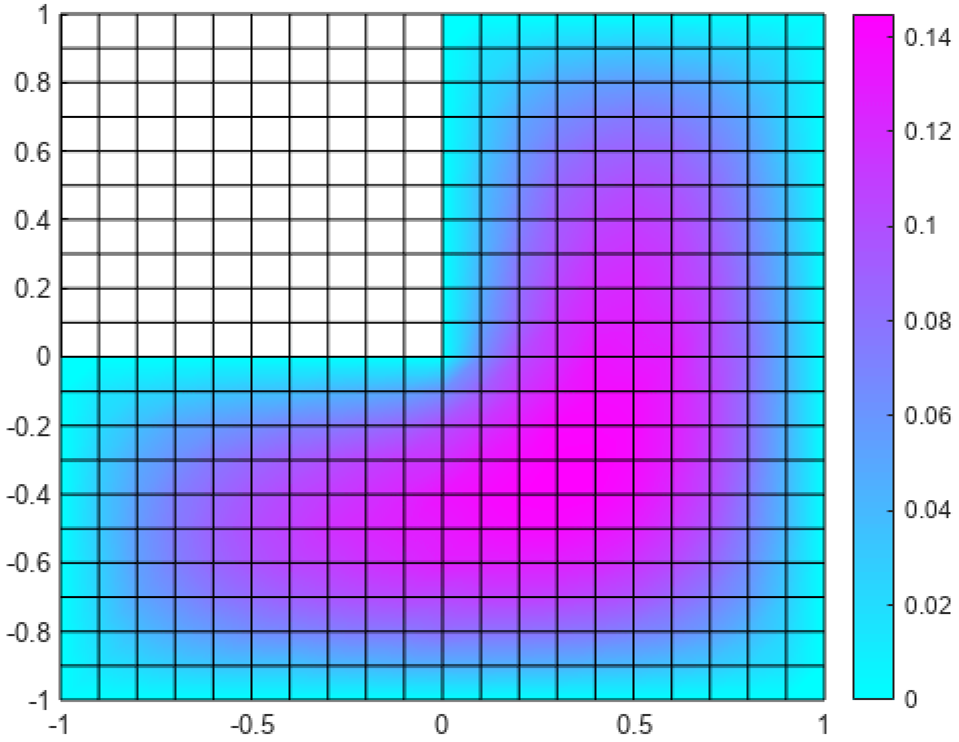
Plot the interpolated solution and the rectangular grid.

```
surface(x,y,uxy)
```



For comparison, plot the original solution and the rectangular grid.

```
[~,tn,a2,a3] = tri2grid(p,t,u,x,y);
pdeplot(p,e,t,'XYGrid','on','GridParam',[tn;a2;a3], ...
              'XYData',u,'Mesh','on')
```



Input Arguments

p — Mesh points
matrix

Mesh points, specified as a 2-by-`Np` matrix of points (nodes), where `Np` is the number of nodes in the mesh. For details on mesh data representation, see `initmesh`.

Data Types: `double`

t — Mesh elements
4-by-`Nt` matrix

Mesh elements, specified as a 4-by-`Nt` matrix of triangles, where `Nt` is the number of triangles in the mesh. For details on mesh data representation, see `initmesh`.

Data Types: `double`

u — PDE solution
vector

PDE solution, specified as a vector. For systems of equations, `tri2grid` interpolates only the first component. For solutions returned by `hyperbolic` or `parabolic`, pass `u` as a vector of values at one time, `u(:,k)`.

x — *x*-coordinates for rectangular grid
vector

x-coordinates for rectangular grid, specified as a vector of elements in ascending order.

Data Types: `double`

y — *y*-coordinates for rectangular grid
vector

y-coordinates for rectangular grid, specified as a vector of elements in ascending order.

Data Types: `double`

Output Arguments

uxy — Interpolated values
`ny`-by-`nx` matrix

Interpolated values, returned as an `ny`-by-`nx` matrix, where `nx` and `ny` are the lengths of the vectors `x` and `y`, respectively.

`tri2grid` uses linear interpolation in the triangle containing the grid point. At grid points outside of the triangular mesh, interpolated values are `NaN`.

`tn` — Indices of triangles containing each grid point
`ny`-by-`nx` matrix

Indices of triangles containing each grid point, returned as an `ny`-by-`nx` matrix, where `nx` and `ny` are the lengths of the vectors `x` and `y`, respectively. At grid points outside of the triangular mesh, indices of triangles are `NaN`.

a2 — Interpolation coefficient
`ny`-by-`nx` matrix

Interpolation coefficient, returned as an `ny`-by-`nx` matrix, where `nx` and `ny` are the lengths of the vectors `x` and `y`, respectively. At grid points outside of the triangular mesh, values of the interpolation coefficients are `NaN`.

a3 — Interpolation coefficient
`ny`-by-`nx` matrix

Interpolation coefficient, returned as an `ny`-by-`nx` matrix, where `nx` and `ny` are the lengths of the vectors `x` and `y`, respectively. At grid points outside of the triangular mesh, values of the interpolation coefficients are `NaN`.

Version History

Introduced before R2006a

Not recommended
Not recommended starting in R2016a

`tri2grid` is not recommended. Use `interpolateSolution` instead. There are no plans to remove `tri2grid`.

Starting in R2016a, use the `interpolateSolution` function to interpolate PDE solutions to arbitrary points. To use this function, solve your equation by using the recommended general PDE workflow. For details, see “Solve Problems Using PDEModel Objects” on page 2-2.

See Also

`solvepde` | `interpolateSolution`

volume

Package: `pde`

Volume of 3-D mesh elements

Syntax

```
V = volume(mesh)
[V,VE] = volume(mesh)
V = volume(mesh,elements)
```

Description

`V = volume(mesh)` returns the volume `V` of the entire mesh.

`[V,VE] = volume(mesh)` also returns a row vector `VE` containing volumes of each individual element of the mesh.

`V = volume(mesh,elements)` returns the combined volume of the specified elements of the mesh.

Examples

Volume of 3-D Mesh

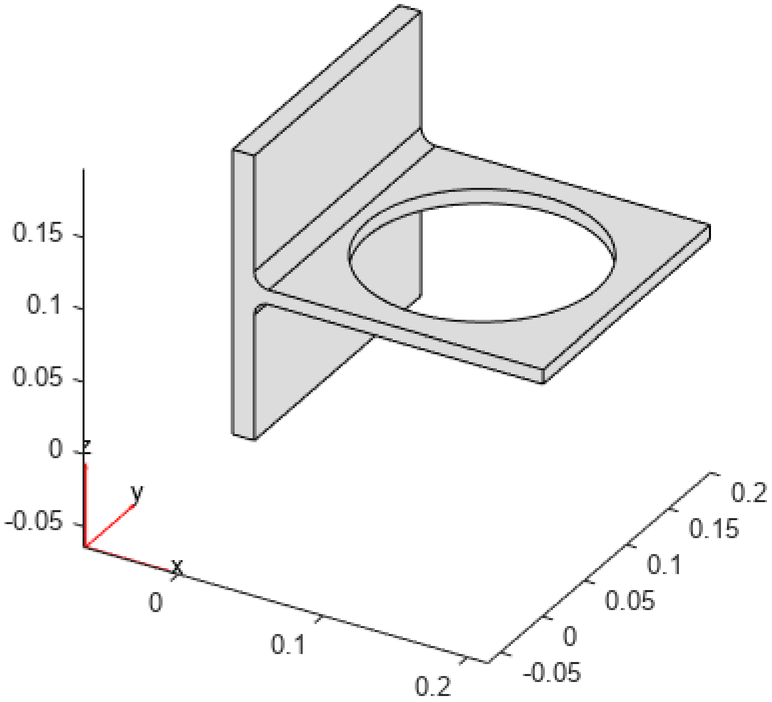
Generate a 3-D mesh and find its volume.

Create a PDE model.

```
model = createpde;
```

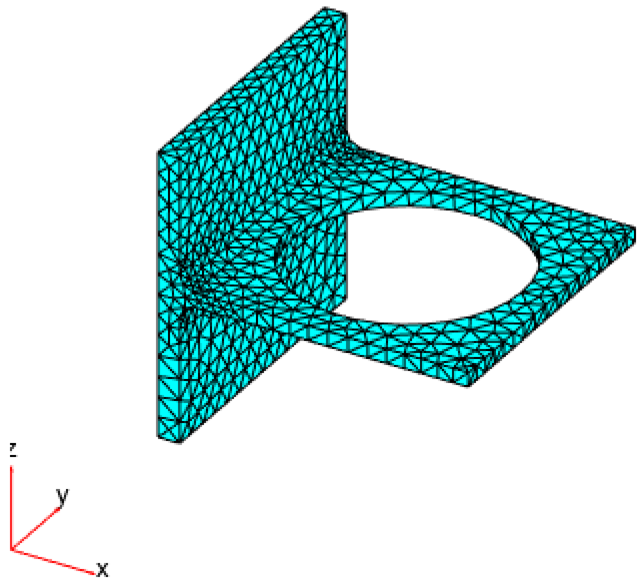
Import and plot the geometry.

```
importGeometry(model,"BracketWithHole.stl");
pdegplot(model)
```



Generate a mesh and plot it.

```
mesh = generateMesh(model);
figure
pdemesh(model)
```



Compute the volume of the entire mesh.

```
mv = volume(mesh)
```

```
mv = 8.0295e-04
```

Volume of Individual Elements of 3-D Mesh

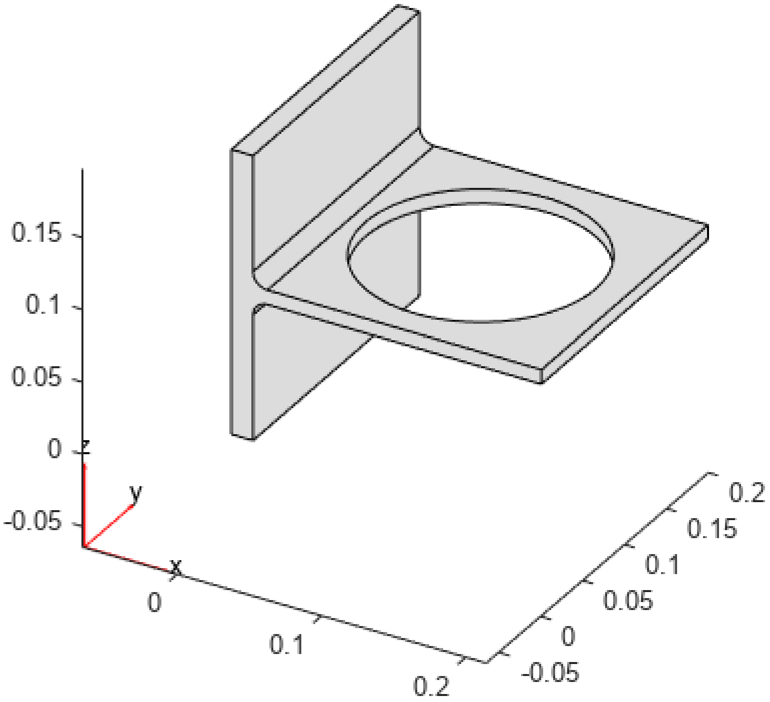
Generate a 3-D mesh and find the volume of each element.

Create a PDE model.

```
model = createpde;
```

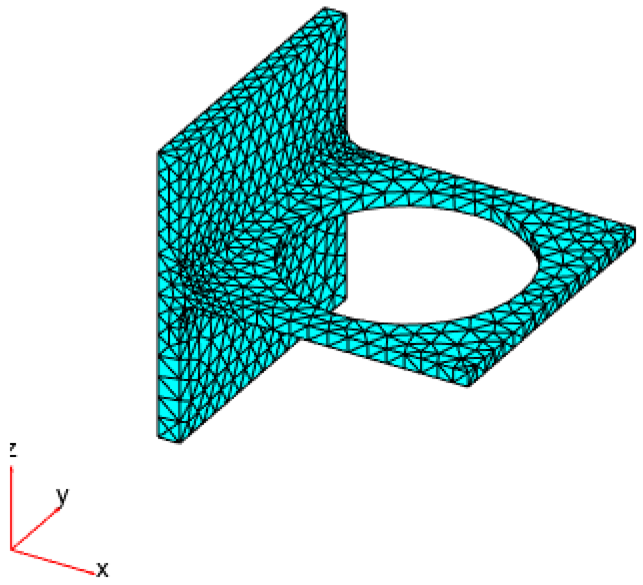
Import and plot the geometry.

```
importGeometry(model,"BracketWithHole.stl");
pdegplot(model)
```



Generate a mesh and plot it.

```
mesh = generateMesh(model);
figure
pdemesh(model)
```



Compute the volume of the entire mesh and the volume of each individual element of the mesh. Display the volumes of the first 5 elements.

```
[va,vi] = volume(mesh);
vi(1:5)
```

```
ans = 1×5
10^-6 ×

    0.5427    0.2243    0.4379    0.2740    0.4541
```

Total Volume of Group of Elements

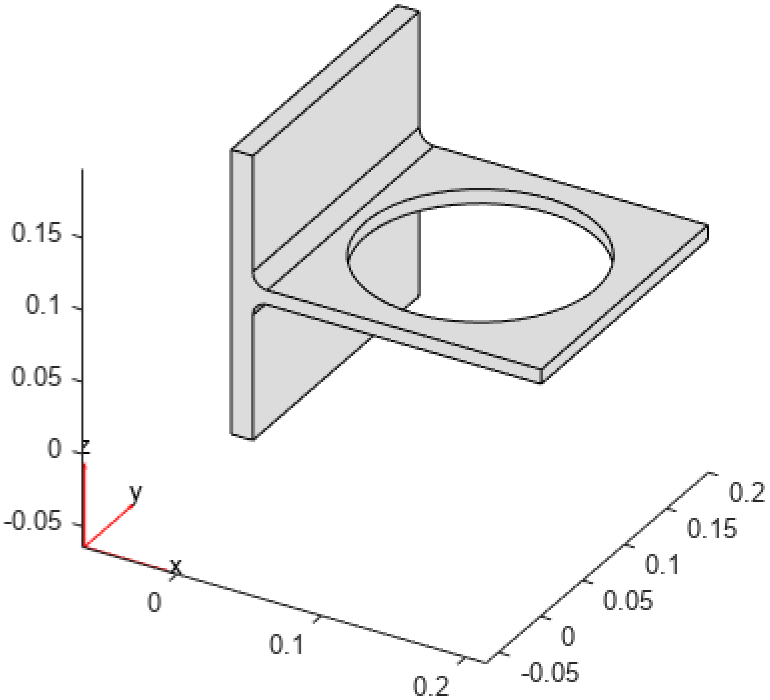
Find the combined volume of a group of elements of a 3-D mesh.

Create a PDE model.

```
model = createpde;
```

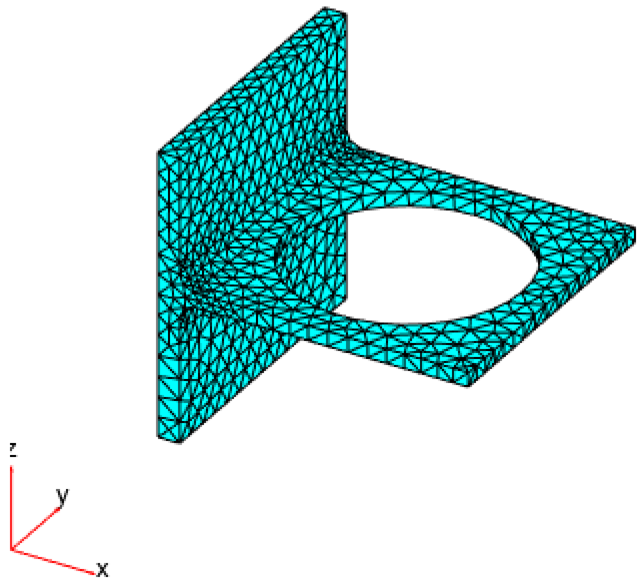
Import and plot the geometry.

```
importGeometry(model,"BracketWithHole.stl");
pdegplot(model)
```



Generate a mesh and plot it.

```
mesh = generateMesh(model);
figure
pdemesh(model)
```



Evaluate the shape quality of the mesh elements and find the elements with the quality values less than 0.5.

```
Q = meshQuality(mesh);
elemIDs = find(Q < 0.5);
```

Compute the total volume of these elements.

```
mv05 = volume(mesh,elemIDs)
```

```
mv05 = 4.2568e-06
```

Find how much of the total mesh volume belongs to these elements. Return the result as a percentage.

```
mv05_percent = mv05/volume(mesh)*100
```

```
mv05_percent = 0.5301
```

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`

`elements` — Element IDs
positive integer | matrix of positive integers

Element IDs, specified as a positive integer or a matrix of positive integers.

Example: `[10 68 81 97 113 130 136 164]`

Output Arguments

`V` — Volume
positive number

Volume of the entire mesh or the combined volume of the specified elements of the mesh, returned as a positive number.

`VE` — Volume of individual elements
row vector of positive numbers

Volume of individual elements, returned as a row vector of positive numbers.

Version History

Introduced in R2018a

See Also

`area` | `findElements` | `findNodes` | `meshQuality` | FEMesh Properties

Topics

"Finite Element Method Basics" on page 1-11

wbound

(Not recommended) Write boundary condition file

Note `wbound` is not recommended. Use `applyBoundaryCondition` instead.

Syntax

```
fid = wbound(b,filename)
```

Description

`fid = wbound(b,filename)` writes a boundary function, specified by the boundary condition matrix `b`, to a file with the name `filename.m`.

Examples

Create Boundary Condition File

Create a 2-D geometry and specify boundary conditions in the PDE Modeler app, export them to the MATLAB workspace, and then write the boundary conditions to a file.

Start the PDE Modeler 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.

Use the default Dirichlet boundary condition $u = 0$ for all boundaries. To verify the boundary condition, 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`.

Export the geometry and the boundary conditions to the MATLAB workspace by selecting the **Export Decomposed Geometry, Boundary Cond's** option from the **Boundary** 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.

Write the resulting boundary condition matrix to a file. Name the file `boundary.m`.

```
fid = wbound(b,"boundary");
```

Input Arguments

b — Boundary conditions
boundary matrix

Boundary conditions, specified as a boundary matrix. Typically, you export a boundary matrix from the PDE Modeler app.

Data Types: `double`

`filename` — Geometry file name
string | character vector

Geometry file name, specified as a string or a character vector.

Data Types: `char` | `string`

Output Arguments

`fid` — File identifier
integer | `-1`

File identifier, returned as an integer. If `wbound` cannot write the file, `fid` is `-1`. For more information about file identifiers, see `fopen`.

Version History

Introduced before R2006a

Not recommended
Not recommended starting in R2016a

`wbound` is not recommended. Use `applyBoundaryCondition` instead. There are no plans to remove `wbound`.

See Also

`decsg` | `wgeom`

wgeom

Write geometry function to file

Note This page describes the legacy workflow. New features might not be compatible with the legacy workflow.

Syntax

`fid = wgeom(dl,filename)`

Description

`fid = wgeom(dl,filename)` writes a geometry function, specified by the geometry matrix `dl`, to a file with the name `filename.m`. For information about the geometry file format, see “Parametrized Function for 2-D Geometry Creation” on page 2-22.

Examples

Create Geometry File

Create a 2-D geometry in the PDE Modeler app, export it to the MATLAB workspace, and then write it to a file.

Start the PDE Modeler 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
```

Write the resulting geometry to a file. Name the file `geometry.m`.

```
fid = wgeom(dl,"geometry");
```

Input Arguments

`dl` — Decomposed geometry matrix

matrix of double-precision numbers

Decomposed geometry matrix, specified as a matrix of double-precision numbers. It contains a representation of the decomposed geometry in terms of disjoint 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

Row number	Circle edge segment	Line edge segment	Ellipse edge segment
12			Angle in radians between *x*-axis and first semiaxis

Data Types: `double`

`filename` — Geometry file name
string | character vector

Geometry file name, specified as a string or a character vector.

Data Types: `char` | `string`

Output Arguments

`fid` — File identifier
integer | `-1`

File identifier, returned as an integer. If `wgeom` cannot write the file, `fid` is `-1`. For more information about file identifiers, see `fopen`.

Version History

Introduced before R2006a

See Also

`decsg` | `wbound`