COMSOL MULTIPHYSICS®

SCRIPTING GUIDE

VERSION 3.4

COMSOL

How to contact COMSOL:

Benelux
COMSOL BV
Röntgenlaan 19
2719 DX Zoetermeer
The Netherlands
Phone: +31 (0) 79 363 4230
Fax: +31 (0) 79 361 4212
info@femlab.nl
www.femlab.nl

Denmark
COMSOL A/S
Diplomvej 376
2800 Kgs. Lyngby
Phone: +45 88 70 82 00
Fax: +45 88 70 80 90
info@comsol.dk
www.comsol.dk

Finland
COMSOL OY
Arabianranta 6
FIN-00560 Helsinki
Phone: +358 9 2510 400
Fax: +358 9 2510 4010
info@comsol.fi
www.comsol.fi

France
COMSOL France
WTC, 5 pl. Robert Schuman
F-38000 Grenoble
Phone: +33 (0)4 76 46 49 01
Fax: +33 (0)4 76 46 07 42
info@comsol.fr
www.comsol.fr

Germany
FEMLAB GmbH
Berliner Str. 4
D-37073 Göttingen
Phone: +49-551-99721-0
Fax: +49-551-99721-29
info@femlab.de
www.femlab.de

Italy
COMSOL S.r.l.
Via Vittorio Emanuele II, 22
25122 Brescia
Phone: +39-030-3793800
Fax: +39-030-3793899
info.it@comsol.com
www.it.comsol.com

Norway
COMSOL AS
Søndre gate 7
NO-7485 Trondheim
Phone: +47 73 84 24 00
Fax: +47 73 84 24 01
info@comsol.no
www.comsol.no

Sweden
COMSOL AB
Tegnérgatan 23
SE-111 40 Stockholm
Phone: +46 8 412 95 00
Fax: +46 8 412 95 10
info@comsol.se
www.comsol.se

Switzerland
FEMLAB GmbH
Technoparkstrasse 1
CH-8005 Zürich
Phone: +41 (0)44 445 2140
Fax: +41 (0)44 445 2141
info@femlab.ch
www.femlab.ch

United Kingdom
COMSOL Ltd.
UH Innovation Centre
College Lane
Hatfield
Hertfordshire AL10 9AB
Phone:+44-(0)-1707 284747
Fax: +44-(0)-1707 284746
info.uk@comsol.com
www.uk.comsol.com

United States
COMSOL, Inc.
1 New England Executive Park
Suite 350
Burlington, MA 01803
Phone: +1-781-273-3322
Fax: +1-781-273-6603

COMSOL, Inc.
10850 Wilshire Boulevard
Suite 800
Los Angeles, CA 90024
Phone: +1-310-441-4800
Fax: +1-310-441-0868

COMSOL, Inc.
744 Cowper Street
Palo Alto, CA 94301
Phone: +1-650-324-9935
Fax: +1-650-324-9936

info@comsol.com
www.comsol.com

For a complete list of international representatives, visit
www.comsol.com/contact

Company home page
www.comsol.com

COMSOL user forums
www.comsol.com/support/forums

COMSOL Multiphysics Scripting Guide

© COPYRIGHT 1994–2007 by COMSOL AB. All rights reserved

Patent pending

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 COMSOL AB.

COMSOL, COMSOL Multiphysics, COMSOL Reaction Engineering Lab, and FEMLAB are registered trademarks of COMSOL AB. COMSOL Script is a trademark of COMSOL AB.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Version: October 2007 COMSOL 3.4

CONTENTS

Chapter 1: Introduction

Typographical Conventions 2

Chapter 2: Scripting from Scratch

Getting Started **4**
Running COMSOL Multiphysics from COMSOL Script or MATLAB. . . . 4
Setting Up the FEM Structure 4
Exporting and Importing the FEM Structure. 6
Saving and Loading an FEM Structure 7
Importing the FEM Structure from a Model M-file 8

Chapter 3: Specifying a Model

Data Structure and Function Overview **10**
FEM Structure Overview. 10
Data Structures and Functions. 11

The FEM Structure's Components **13**
The Geometry and the FEM Problem 14
The Mesh 19
Shape Functions 20
Variables and Functions 25
Units 31
Materials/Coefficients Libraries 33
Pairs 37
Equations and Constraints 38
Discretization 51
Initial Values. 53
The Extended Mesh. 54
Multiple Geometries 55

Application Structures 56

The Structure of a Model M-file **65**
Model M-files for Models with Multiple Geometries. 73

Chapter 4: Geometry Objects and Images

Geometry Objects **76**
The Geometry of a PDE Problem 76
The Geometry Object Hierarchy. 77
Geometry Classes 80
Geometry Functions 82

Geometry Modeling **85**
Working with Geometry Objects 85
Working with a Geometry Model 90
Importing and Exporting Geometries and CAD Models from File 92
Working With the Analyzed Geometry 93
Working with Assemblies 94
Retrieving Geometry Information 95
Modeling with a Parametrized Geometry. 95

Images, Interpolation, and MRI Data **98**
Working with Images and MRI Data. 98
Spline and Surface Interpolation 104

Chapter 5: Creating Meshes

Generating Meshes **110**
Overview. 110
Mesh Creation Functions. 110

Importing and Exporting Meshes **127**
Importing External Meshes and Mesh Objects 127
Exporting Meshes From COMSOL Multiphysics 129

Chapter 6: Solver Scripting

Solver Command Overview **132**
Computing the Solution 132

Working with the Assembled Matrices **136**
Exploring the Stiffness Matrix 137
Evaluating and Visualizing the Residual 143

Solver Scripting Examples **144**
Concatenating Several Transient Solutions 144
Parametric Studies 145
Using Scripting to Solve Multiphysics Problems Iteratively 148

Chapter 7: Postprocessing and Visualization

Introduction to Postprocessing **154**

Overview of Postprocessing Functions **155**

Interpolation and Integration **157**
Data Evaluation and Interpolation 157
Computing Integrated Quantities. 159
Maximum Temperature, Average Temperature, and Total Heat Flux. . . . 159

Working With Graphics Objects **161**
Editing Plots in the COMSOL Figure Windows 161
Handles—Editing Plots By Programming 164

Postprocessing and Associativity **167**

Chapter 8: Simulink and State-Space Export

Simulink Export **170**
Dynamic or Static Export? 170
General or Linearized Export? 171
The Simulink Export Dialog Box 171
The Simulink Export Types 172
Relation to Solver Parameters and Solver Manager Settings 173
The Input/Output Variables 174
Using the COMSOL Multiphysics Model in Simulink. 175

State-Space Export **177**
The State-Space Form 177
Model Reduction 178
The State-Space Export Dialog Box 179
The Input/Output Variables 180

1

Introduction

This manual describes how to model using the COMSOL Multiphysics® programming language in the COMSOL Script™ environment or at the MATLAB® command line. Its chapters cover geometry modeling and meshing, PDE problem definition, computing the solution, and performing visualization and postprocessing.

Throughout the book, the various steps in the modeling procedure are illustrated with examples chosen to highlight the particular strengths and benefits of script-based modeling, such as:

- Enhanced productivity through convenient and flexible interaction with the COMSOL Multiphysics graphical user interface
- Additional possibilities for parametric studies, for instance in the context of geometry optimization
- Powerful tools for retrieving and manipulating geometry and mesh data
- Full control over the solution stage, allowing for advanced adaptive solution strategies
- Extended visualization capabilities
- Advanced numerical postprocessing

For reference documentation of all scripting-related functionality in COMSOL Multiphysics, see the *COMSOL Multiphysics Reference Guide*. For documentation that describes the respective programming languages, see the *COMSOL Script User's Guide* and MATLAB's corresponding documentation.

Typographical Conventions

All COMSOL manuals use a set of consistent typographical conventions that should make it easy for you to follow the discussion, realize what you can expect to see on the screen, and know which data you must enter into various data-entry fields. In particular, you should be aware of these conventions:

- A **boldface** font of the shown size and style indicates that the given word(s) appear exactly that way on the COMSOL graphical user interface (for toolbar buttons in the corresponding tooltip). For instance, we often refer to the **Model Navigator**, which is the window that appears when you start a new modeling session in COMSOL; the corresponding window on the screen has the title **Model Navigator**. As another example, the instructions might say to click the **Multiphysics** button, and the boldface font indicates that you can expect to see a button with that exact label on the COMSOL user interface.
- The names of other items on the graphical user interface that do not have direct labels contain a leading uppercase letter. For instance, we often refer to the Draw toolbar; this vertical bar containing many icons appears on the left side of the user interface during geometry modeling. However, nowhere on the screen will you see the term "Draw" referring to this toolbar (if it were on the screen, we would print it in this manual as the **Draw** menu).
- The symbol **>** indicates a menu item or an item in a folder in the **Model Navigator**. For example, **Physics>Equation System>Subdomain Settings** is equivalent to: On the **Physics** menu, point to **Equation System** and then click **Subdomain Settings**. **COMSOL Multiphysics>Heat Transfer>Conduction** means: Open the **COMSOL Multiphysics** folder, open the **Heat Transfer** folder, and select **Conduction**.
- A `Code` (monospace) font indicates keyboard entries in the user interface. You might see an instruction such as "Type `1.25` in the **Current density** edit field." The monospace font also indicates COMSOL Script codes.
- An *italic* font indicates the introduction of important terminology. Expect to find an explanation in the same paragraph or in the Glossary. The names of books in the COMSOL documentation set also appear using an italic font.

2

Scripting from Scratch

The present chapter outlines the basics of how to use the COMSOL Multiphysics programming language and scripting capabilities to build models.

Getting Started

Running COMSOL Multiphysics from COMSOL Script or MATLAB

Start the COMSOL Script environment from the COMSOL Multiphysics user interface by choosing **COMSOL Script** from the **File** menu. If you want to use the features described in this manual in MATLAB, you must run COMSOL with MATLAB. The easiest way to do so in Windows is to double-click the icon **COMSOL with MATLAB**, which should be visible on the desktop. Under UNIX and Linux, start COMSOL with MATLAB with the command `comsol matlab`. See the *COMSOL Installation Guide* for more information about how to run COMSOL Multiphysics.

Note: All the following procedures refer to COMSOL Script unless otherwise indicated. If you run COMSOL Multiphysics with MATLAB, the commands and procedures are identical but the scripting environment is then MATLAB instead of COMSOL Script.

Setting Up the FEM Structure

When working on the command line you encounter the *FEM structure*. This is a data structure—more specifically, a scalar structure array in COMSOL Script and MATLAB—containing the complete definition of a COMSOL Multiphysics model.

This chapter reviews the basics on how to set up an FEM structure to solve a problem. Step-by-step instructions show how to specify a model by assigning values to the fields in the FEM structure. A more comprehensive description of the FEM structure appears in the next chapter, “Specifying a Model” .

A FIRST EXAMPLE—POISSON’S EQUATION ON THE UNIT DISK

A classic PDE with a well-known behavior is Poisson’s equation

$$\begin{cases} -\nabla \cdot (\nabla u) = f & \Omega \\ u = 0 & \partial\Omega \end{cases}$$

on the unit disk Ω with $f = 1$. For this problem, you can compare the exact solution

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

with the numerical solution at the node points on the mesh.

To set up the PDE on the command line follow these steps:

1. Make it a habit to clear the FEM structure when defining a new problem:

   ```
   clear fem
   ```

2. The `geom` field of the FEM structure contains the problem's geometry. Create a unit circle centered at the origin with the command `circ2`:

   ```
   fem.geom = circ2;
   ```

3. The `meshinit` function creates a triangular mesh on the geometry defined in the `fem.geom` field:

   ```
   fem.mesh = meshinit(fem);
   ```

4. To specify the PDE coefficients, use the coefficient form of the basic equations and set both c and f equal to 1 (see the section "Using the Coefficient Form PDEs" on page 239 of the *COMSOL Multiphysics Modeling Guide* for details on the PDE terminology in COMSOL Multiphysics). All boundaries should have $u=0$ as boundary conditions, which means setting h to 1. All coefficients you do not specify are zero by default.

   ```
   fem.equ.f = 1;
   fem.equ.c = 1;
   fem.bnd.h = 1;
   ```

   The default name of the dependent variable is `u`, but it is possible to use another name, for example `T`, by entering `fem.dim = 'T';` however, in this example you do not have to specify `fem.dim`.

   The PDE coefficients, coefficients in the boundary conditions, and initial conditions are not restricted to constant values. By specifying them as strings, you can define model parameters to be spatially varying (function of x, y, and z), time dependent (function of t), complex-valued (function of i or j), nonlinear (function of any and all dependent variables), and discontinuous. Furthermore, the string can reference an interpolation table or be a subroutine (any M-file in the current path or built-in functions in COMSOL Script or MATLAB). For more information about variable types that you can use and create, see "Variables and Functions" on page 25. For example, you can define an initial condition that varies with x as $\mathrm{atan}(\cos(0.5\pi x))$:

```
fem.init = atan(cos(0.5*pi*x));
```

Furthermore, the PDE coefficients can represent anisotropic material and describe a system with multiple dependent variables using brackets. For more information about this syntax, see "Equations and Constraints" on page 38.

5 Choose quadratic Lagrange elements:

```
fem.shape = 2;
```

This setting must be made explicitly because the default Lagrange element order on the command line is 1. Note that this differs from the graphical user interface where the default order is 2.

6 The `meshextend` function creates the extended mesh object, which is required for assembling the problem:

```
fem.xmesh = meshextend(fem);
```

Further information appears in the section "The Extended Mesh" on page 54.

7 Solve the PDE and plot the solution:

```
fem.sol = femstatic(fem);
postplot(fem,'tridata','u','triz','u');
```

8 Compute the maximal error by evaluating the difference with the exact solution:

```
pd = posteval(fem,'u-(1-x^2-y^2)/4');
er = max(max(pd.d))
```

9 If the error is not sufficiently small, refine the mesh and update the model:

```
fem.mesh = meshrefine(fem);
fem.xmesh = meshextend(fem);
```

You can then solve the problem on the new mesh, plot the solution, and recompute the error by repeating Steps 7 and 8.

For comparison, this example is discussed in the context of the COMSOL Multiphysics graphical user interface in the *COMSOL Multiphysics Modeling Guide* in the section "Poisson's Equation on the Unit Disk" on page 275.

Exporting and Importing the FEM Structure

A good way to get to know the FEM structure is by exporting it from the COMSOL Multiphysics graphical user interface to the command line. You can do this at any time during a modeling session. Conversely, when working at the command line, you can look at a model in COMSOL Multiphysics by importing the FEM structure from the command line.

EXPORTING THE FEM STRUCTURE

Export the current FEM structure to the command line by selecting **Export>FEM Structure as 'fem'** from the **File** menu. If you want to specify a different name for the exported FEM structure, instead choose **File>Export>FEM Structure**.

Including Default Values

By default, the export does not include the default values of the fields in the exported FEM structure. To include these default values, do the following:

1. From the **Options** menu open the **Preferences** dialog box.
2. On the **Modeling** page select the **Include default values** check box in the **Model M-file/FEM export** area.
3. Click **OK**.

Shrinking Coefficients

By default, the export "shrinks" the coefficients in the physics settings to a more compact notation than the full syntax (see "Equations and Constraints" on page 38 for more information). To get the full syntax in the exported FEM structure, proceed as follows:

1. From the **Options** menu open the **Preferences** dialog box.
2. On the **Modeling** page clear the **Shrink coefficients** check box in the **Model M-file/FEM export** area.
3. Click **OK**.

IMPORTING THE FEM STRUCTURE

You can set up a complete PDE problem by building an FEM structure on the command line and subsequently importing it into COMSOL Multiphysics. Select **Import>FEM Structure** from the **File** menu, then type the name of the FEM structure in the dialog box. COMSOL Multiphysics tries to import the FEM structure, attempting not to terminate the import if it encounters syntactic errors. Instead, it indicates any problems in the message log and sets up the parts of the FEM structure it can handle.

Saving and Loading an FEM Structure

COMSOL Multiphysics provides special commands for saving and loading FEM structures to and from a COMSOL Multiphysics Model MPH-file.

SAVING AN FEM STRUCTURE

Use the command `flsave` to save an FEM structure to a Model MPH-file. For example, to save the FEM structure `fem` to the model file `model.mph`, type

```
flsave model fem
```

You can then open the model file in the COMSOL Multiphysics graphical user interface or load the FEM structure using the command `flload`.

To save an FEM structure or any part of an FEM structure as a MAT-file in MATLAB, use the `save` command, and to load a MAT-file use the `load` command.

LOADING FEM STRUCTURES, GEOMETRY OBJECTS, AND MESH OBJECTS

Use the command `flload` to load an FEM structure from a Model MPH-file or a MAT-file.

If `filename` is a Model MPH-file, the command `flload filename` loads the FEM structure in the model file and assigns it the variable name `fem`.

If `filename` is a MAT-file created with an older version of COMSOL Multiphysics (FEMLAB), the `flload` command loads the FEM structures, the Simulink structures, or the geometry objects saved under the file name `filename`. In this case, the difference from the standard `load` command is that `flload` reobjectifies the structures for future compatibility; that is, the original objects are recreated from the saved structures.

You can also use `flload` to load COMSOL Multiphysics text and binary files (with extensions `.mphtxt` and `.mphbin`, respectively) for retrieving geometry and mesh objects.

Importing the FEM Structure from a Model M-file

Another way to learn command-line modeling is by saving a model from the COMSOL Multiphysics user interface as a Model M-file containing a complete description of a PDE problem. To generate the FEM structure stored in the Model M-file `filename.m` in the scripting environment, type `run filename` at the command line, or choose **Run M-file** from the **File** menu of the COMSOL Script window. The section "The Structure of a Model M-file" on page 65 provides additional information.

3

Specifying a Model

This chapter describes the general structure of a COMSOL Multiphysics model and outlines how you can use the COMSOL Multiphysics programming language and scripting capabilities to build models.

The first section provides an overview of the FEM structure—the basic data structure underpinning all COMSOL Multiphysics models—and the functions that you can use to control and modify it.

The subsequent section describes the different FEM structure components and how to work with them in the modeling process.

The final section explains the structure of a Model M-file, the sections that it can contain, and which sections a valid Model M-file must include.

Further details and more elaborate examples of some of the main stages in command-line modeling are deferred to subsequent chapters.

Data Structure and Function Overview

FEM Structure Overview

The data that defines a PDE problem is stored in a *scalar structure array* called the *FEM structure*. A scalar structure array—a data type similar to the structure in the C language—is an array of containers for a set of fields, which you can access with a hierarchical dot notation. The data structures in the fields of the FEM structure define different aspects of the PDE problem, and the various processing stages produce new data structures from existing ones. The most important fields for command-line modeling are now listed:

TABLE 3-1: FEM STRUCTURE FIELDS

FIELD	DETAILS IN	INTERPRETATION
`fem.appl`	page 57	Application modes
`fem.bnd`	page 13	Variables, equations, constraints, and initial values on boundaries (not present in 0D)
`fem.border`	page 47	Assembly on interior boundaries
`fem.const`	page 26	Definitions of constants
`fem.cporder`	page 52	Constraint point orders
`fem.descr`	page 27	User comments and descriptions
`fem.dim`	page 39	Names or number of dependent variables
`fem.draw`	page 66	The solid objects, face objects (3D only), curve objects (2D and 3D only), and point objects used for geometry modeling
`fem.edg`	page 13	Variables, equations, constraints, and initial values on edges (only in 3D)
`fem.equ`	page 13	Variables, equations, constraints, and initial values on subdomains
`fem.expr`	page 26	Definitions of variables as expressions
`fem.form`	page 38	Form of equations. Can be `coefficient` (default), `general`, or `weak`
`fem.frame`	page 15	Frame names
`fem.functions`	page 29	User-defined functions

TABLE 3-1: FEM STRUCTURE FIELDS

FIELD	DETAILS IN	INTERPRETATION
`fem.geom`	page 14	Geometry objects
`fem.globalexpr`	page 26	Definitions of variables as global expressions
`fem.gporder`	page 51	Numerical integration orders
`fem.lib`	page 33	Materials library
`fem.mesh`	page 19	Mesh object or structure. Contains the finite element mesh
`fem.meshtime`	page 18	Mesh object or structure. Contains the finite element mesh
`fem.ode`	page 13	ODEs and other global scalar variables, equations, and initial conditions
`fem.pnt`	page 13	Variables, equations, constraints, and initial values on vertices (only in 2D and 3D)
`fem.sdim`	page 15	Names or number of space coordinates (independent variables)
`fem.simplify`	page 57	Simplify expressions
`fem.shape`	page 22	Shape functions (finite elements)
`fem.sol`	`femsol`	Solution object. Subfields: `u`, `ut`, `lambda`, `tlist`, `plist`
`fem.solform`	page 54	The solution form. Can be `coefficient`, `general`, or `weak` (default)
`fem.sshape`	page 16	Geometry shape
`fem.units`	page 31	Base unit system
`fem.var`	page 61	Application scalar variables
`fem.version`	page 65	Version number information
`fem.xmesh`	page 54	Extended mesh structure

Data Structures and Functions

The following table indicates how the major commands in COMSOL Multiphysics affect the FEM structure. Most functions accept as input the full FEM structure and

by default return only a part of it. Optionally, most functions can also return the full FEM structure.

TABLE 3-2: MAJOR COMSOL MULTIPHYSICS FUNCTIONS

FUNCTION	INPUT FIELDS	OUTPUT FIELDS
`adaption`	`geom`, `mesh`, `xmesh`, `const`, `dim`, `init`	`mesh`, `xmesh`, `sol`
`assemble`	`xmesh`, `const`	Assembled matrices
`asseminit`	`xmesh`, `const`	Solution object
`femdiff`	`sdim`, `dim`, `equ`, `bnd`	`equ`, `bnd`
`femlin`, `femnlin`, `femstatic`, `femtime`, `femeig`	`xmesh`, `const`, `init`	`sol`
`geomanalyze`	many fields, especially `draw` and `geom`	`draw`, `geom`, `appl`, `equ`, `bnd`, `edg`, `pnt`
`geomcsg`	`draw`	`geom`
`meshextend`	most fields	`xmesh`
`meshinit`	`geom`	`mesh`
`meshrefine`	`geom`, `mesh`	`mesh`
`multiphysics`	many fields, especially `appl`	`dim`, `form`, `equ`, `bnd`, `edg`, `pnt`, `var`, `elemmph`, `eleminitmph`, `shape`, `sshape`, `border`
`posteval`, `postinterp`	`geom`, `mesh`, `xmesh`, `const`, `sol`, `equ.ind`, `bnd.ind`, `edg.ind`, `pnt.ind`	Postdata (values of expressions)
`postplot`, `postmovie`	`geom`, `mesh`, `xmesh`, `const`, `sol`, `equ.ind`, `bnd.ind`, `edg.ind`, `pnt.ind`	Graphics output
`xmeshinfo`	`xmesh`	Information about the extended mesh

The FEM Structure's Components

This section describes how to specify a model using the scripting environment, either interactively or by editing an M-file. As explained in the previous section, the FEM structure (denoted `fem` in the examples below) contains the complete data defining the model. Consequently, you can create and modify a model by assigning values to the fields of the FEM structure. The following descriptions of these fields are based on models that contain a single geometry. If there are several geometries in a model, you must use an *extended FEM structure*, a concept explained on page 55.

A convenient and efficient way to learn the correct syntax for the FEM structure is by first creating a model in the COMSOL Multiphysics graphical user interface and then exporting it to COMSOL Script or MATLAB. You can then work on the model by examining and modifying the FEM structure at the command line.

EQUATIONS, CONSTRAINTS, AND INITIAL VALUES

Store the equations, constraints (boundary conditions), and initial conditions of a PDE problem in the following fields:

- `fem.equ`—contains information about equations on *subdomains*
- `fem.bnd`—contains information belonging to *boundaries*
- `fem.edg`—in 3D, this field is related to *edges (curves)*
- `fem.pnt`—in 2D and 3D, this field corresponds to *vertices* (*points)*
- `fem.ode`—specifies *ODEs* and other scalar equations independent of position

You can specify equations in *strong form* (as PDEs) or in *weak form* (as integral equations). There are two versions of the strong form: the *coefficient form*, which is suitable for linear problems, and the *general form*, which is suitable both for linear and nonlinear problems. Use the field `fem.form` to specify the equation form.

DEPENDENT VARIABLES AND FINITE ELEMENTS

Specify the names of the dependent variables in the field `fem.equ.dim`. The field `fem.shape` describe the finite elements (shape functions). Use the field `fem.ode.dim` to add global dependent variables, which are available everywhere in a model.

THE EXTENDED MESH

When you have specified a PDE problem, use the COMSOL Multiphysics function `meshextend` to create the *extended mesh structure*, `fem.xmesh`. The extended mesh

contains a low-level description of the PDE problem. Remember to run `meshextend` before solving a model when you have made changes to it because the solution and postprocessing stages are based on the extended mesh.

The Geometry and the FEM Problem

The section described the relationship between COMSOL's geometry representation and the physics model. Find a more detailed explanation of COMSOL's geometry representation and additional examples of command-line geometry modeling in Chapter 4, "Geometry Objects and Images."

FEM.GEOM—THE ANALYZED GEOMETRY

The analyzed geometry is given in the field `fem.geom`. For details, see both the *COMSOL Multiphysics Reference Guide* entry for `geomcsg` and the chapter "Geometry Objects and Images" . For instance,

```
fem.geom = rect2(4,4,'pos',[-2 -2]) + circ2;
```

creates a plane geometry consisting of the unit circle inside a square with a side length of 4.

The "domains" make up the bounded region of space in which you solve the equations. Subdomains are numbered 1, 2, 3,…, as are the boundaries, edges, and vertices (if they exist). In the regions of space outside the domains no physics are allowed. These regions are sometimes given the number 0.

Data pertaining to subdomains, boundaries, edges, or vertices are stored in separate fields of the FEM structure as shown in the following table.

TABLE 3-3: FEM STRUCTURE FIELDS

DOMAIN NAME	FIELD IN FEM STRUCTURE
subdomain	`fem.equ`
boundary	`fem.bnd`
edge	`fem.edg`
vertex	`fem.pnt`

Note that the field name `equ` has historic reasons—originally the field contained only equations.

DOMAIN GROUPS

It is often convenient to treat several domains as one group using the same equations, material properties, or boundary conditions. You can do this by forming *domain*

groups. Subdomain groups are defined in the optional field `fem.equ.ind`. It can be a cell array of numeric vectors or a numeric vector. Two examples illustrate the alternative syntaxes. The first is

```
fem.equ.ind = { [1 2] [6 3 4] };
```

which indicates that subdomain group 1 consists of subdomains 1 and 2, while subdomain group 2 consists of subdomains 6, 3, and 4. Equivalent is the statement

```
fem.equ.ind = [1 1 2 2 0 2];
```

which specifies the group number for each subdomain. A 0 means that a particular subdomain (number 5 in this case) does not belong to any group.

If the field `fem.equ.ind` is not given, then each subdomain forms its own subdomain group, that is, the subdomain groups are the same as the subdomains.

Similarly, groupings of boundaries, edges, and vertices can be defined in `fem.bnd.ind`, `fem.edg.ind`, and `fem.pnt.ind`, respectively.

FEM.SDIM—SPACE COORDINATES

The default names of the (global) space coordinates are `x`, `y`, `z` in 3D, `x` and `y` in 2D, and `x` in 1D. You can override this by specifying your own names in the optional cell array `fem.sdim`, for example

```
fem.sdim = {'r' 'phi' 'z'};
```

in 3D. The space coordinates are sometimes called the *independent variables*.

Models using a moving mesh require a second set of coordinates to describe the mesh's motion. The coordinate sets are called *frames*. To specify two frames, make `fem.sdim` a cell array of two cell arrays of coordinates, for example,

```
fem.sdim = {{'X' 'Y'} {'x' 'y'}};
```

in 2D. One of the frames is the *reference frame*, the fixed frame where you draw the geometry; the other frame describes the mesh motion. The field `fem.sshape` (see "fem.sshape—Geometry Shape" on page 16) specifies which of the two is the reference frame. To ensure that postprocessing functions use the deformed mesh in their evaluations, make certain that the reference frame is the first of the two frames in `fem.sdim`.

FEM.FRAME—FRAME NAMES

When there is more than one frame you must give each a name. Do this in the field `fem.frame`, for example,

```
fem.frame = {'ref' 'ale'};
```

If you do not specify `fem.frame`, the frames get a default name, which is the concatenation of the coordinates.

FEM.SSHAPE—GEOMETRY SHAPE

The global coordinates are polynomials in the local element coordinates of a certain degree. This degree, k, can be specified in the field `fem.sshape`. For instance,

```
fem.sshape = 2;
```

gives quadratic shape functions for the global space coordinates. This makes it possible to have curved mesh elements at the boundary and thereby come closer to the true geometric boundary. The default value of k equals the maximum order of the shape function objects in `fem.shape`. The variable k is called the *geometry shape order*.

Moving Mesh

When solving problems with a moving mesh, it is not sufficient to specify only the geometry shape order; `fem.sshape` should then also contain definitions of the geometry shape due to mesh motion. In those cases, `fem.sshape` is a cell array of structures, each structure defining a geometry shape, and it should contain all such shape definitions that appear anywhere in the model. The field `fem.equ.sshape` specifies which geometry shape definitions apply in each domain. The structures in `fem.sshape` have the following fields:

- `fem.sshape.frame`—the frame that this structure defines. If absent, COMSOL Multiphysics uses the first frame in `fem.sdim`.
- `fem.sshape.sorder`—the geometry shape order.
- `fem.sshape.dvolname`—the name of the mesh element scale factor.
- `fem.sshape.type = 'fixed' | 'moving_abs' | 'moving_rel' | 'moving_expr'`
  - `'fixed'` means that the frame does not move. For the reference frame the type should be `'fixed'`.
  - `moving_abs` is used when the mesh equation describes the absolute mesh movement.
  - `moving_rel` is used when the mesh equation describes the relative mesh movement.
  - `moving_expr` is used when the mesh movement is given as an expression.
- `fem.sshape.sdimdofs`—a cell array of strings with the variables defining the spatial coordinates for relative mesh movement. Required and used only when

`fem.sshape.type = 'moving_rel'`. The spatial coordinates in `sdimdofs` must be defined directly by a shape function.

- `fem.sshape.refframe`—the name of the reference frame. Required and used only when `fem.sshape.type = 'moving_rel'` and indicates which frame the motion is relative to.
- `fem.sshape.sdimexprs`—a cell array of strings with the expressions giving the mesh displacement. Used when `fem.sshape.type = 'moving_expr'` and gives the expressions for the mesh deformation. The default is a cell array with zeros.

Domains of Usage for Geometry Shapes

In the fields `fem.equ.sshape`, `fem.bnd.sshape`, `fem.edg.sshape`, and `fem.pnt.sshape` you can specify where to use the geometry shape objects in `fem.sshape`. For example,

```
sshape1.frame = 'ref';
sshape1.type = 'fixed';
sshape1.sorder = 2;
sshape1.dvolname='dvol';
sshape2.frame = 'ale';
sshape2.type = 'moving_abs';
sshape2.sorder = 2;
sshape2.dvolname = 'dvol_ale';
sshape3.frame = 'ale';
sshape3.type = 'moving_rel';
sshape3.sorder = 2;
sshape3.sdimdofs = {'dx' 'dy'};
sshape3.refframe = 'ref';
sshape3.dvolname = 'dvol_ale';

fem.sshape = {sshape1 sshape2 sshape3};
fem.equ.sshape = { [1 3] [1 2] };
```

means that on the first subdomain group only the first and third geometry shape objects (`sshape1` and `sshape2`) are active. On the second subdomain group the first and third geometry shape objects are active.

To make sure that the spatial coordinates are well defined everywhere, one `sshape` object in each domain must define all frames. In this example, `sshape1` defines the coordinates in the reference frame, while `sshape2` and `sshape3` are two definitions of the coordinates in the frame `'ale'`. Therefore `sshape1` should be active everywhere, and one of `sshape2` or `sshape3` should be defined everywhere.

If you do not specify the field `fem.equ.sshape`, all geometry shape objects in `fem.sshape` apply in all subdomain groups.

Similarly, the field `fem.bnd.sshape` is a cell array that specifies for each boundary group which shape function objects are active. If `fem.bnd.sshape` is not given, then it is inherited from `fem.equ.sshape`. This means that a geometry shape object that is active in a subdomain is also active on the boundary of that subdomain (as well as boundaries lying within the subdomain). If more than one geometry shape object defining the coordinates of the same frame on the subdomains adjacent to a boundary exist then only one is selected. Any one of `type = 'moving_abs'` is chosen first.

In 3D, the field `fem.edg.sshape` similarly specifies the usage of geometry shape objects on edge groups. If it is not given, it is inherited from the usage on subdomains and boundaries.

In 2D and 3D, the field `fem.pnt.sshape` similarly specifies usage on vertices. It is defaulted by inheritance from subdomains, boundaries, and edges.

Domains of Usage for Geometry Shapes by each Equation

When there is more than one frame in a problem, you must specify on which frame each equation is formulated. Do this using the field `fem.sshapedim`. Assume you have a problem with four equations. Then you can, for example, have

```
fem.sshape = {sshape1 sshape2 sshape3};
fem.equ.sshapedim = {[1 1 3 3] [1 1 2 2]};
```

The objects in `fem.sshape` are the same ones as in the previous section. The first two equations live on the reference frame using `sshape1` in both subdomain groups. The third and fourth equations live on the frame `'ale'` using `sshape3` in the first subdomain group and `sshape2` in the second group.

fem.meshtime—Mesh Time Derivative

The name of the mesh time derivative is stored in the `fem.meshtime` field. For details about this variable see the section "Time Derivatives" on page 394 in the *COMSOL Multiphysics Modeling Guide*.

GEOMETRIC VARIABLES

Besides the space coordinates (typically `x`, `y`, and `z` in Cartesian coordinates, and `r` and `z` in axisymmetric models), there are a number of geometric variables that you can use in expressions. See "Geometric Variables" on page 165 in the *COMSOL Multiphysics User's Guide*.

For mesh terminology see "Meshing" on page 285 in the *COMSOL Multiphysics User's Guide*.

FEM.MESH—THE MESH OBJECT OR STRUCTURE

You can find a detailed description of the mesh object in the *COMSOL Multiphysics Reference Guide* entry for `femmesh`. If the model contains mesh cases, `fem.mesh` is a structure with the following fields:

- `fem.mesh.default` is the mesh corresponding to mesh case 0. This mesh is also used in all mesh cases that do not occur in `fem.mesh.mind`.
- `fem.mesh.case` is a cell array of meshes corresponding to the mesh cases in `fem.mesh.mind`. The default is an empty cell array.
- `fem.mesh.mind` is a cell array containing vectors of positive mesh case indices. That is, the mesh `fem.mesh.case{i}` is used in the mesh cases `fem.mesh.mind{i}`. The default `fem.mesh.mind` is `{1 2 ... n}`, where `n` is the length of `fem.mesh.case`.

GENERATING A MESH

Given an FEM structure with a geometry `fem.geom`, you create an initial unstructured mesh by

```
fem.mesh = meshinit(fem);
```

Several options allow you to fine-tune the meshing algorithm. The most important one is the overall maximum mesh size, `hmax`. For example,

```
fem.mesh = meshinit(fem,'hmax',0.01);
```

ensures that all mesh-element diameters are less than 0.01. You can also specify local values of `hmax` if you want the mesh to be finer in certain domains.

Given an unstructured mesh, you can refine it using `meshrefine`:

```
fem.mesh = meshrefine(fem);
```

Here you can also choose among different algorithms.

A 1D or 2D mesh can also be used as geometry data. For example, you can create a coarser mesh starting from a finer mesh:

```
fem.geom = fem.mesh;
fem.mesh = meshinit(fem,'hmax',0.2);
```

In 1D and 2D, the geometry that the `meshinit` function uses as input data can also be a geometry M-file, for example, the cardioid function:

```
fem.geom = 'cardg';
fem.mesh = meshinit(fem);
```

To visualize a mesh, use the `meshplot` function:

```
meshplot(fem);
```

Shape Functions

Finite elements are defined using *shape function objects*. Their task is to express field variables as linear combinations of certain basis functions (shape functions). The coefficients are called degrees of freedom (DOFs), and they form the discrete representation of the field variables. The shape function objects are objects from certain *shape function classes*. COMSOL Multiphysics has the following shape function classes:

- `shlag`—Lagrange element of arbitrary order
- `shherm`—Hermite element of arbitrary order (>2)
- `sharg_2_5`—5th-order Argyris element on triangular meshes in 2D
- `shcurl`—Vector (edge) element on triangular or tetrahedral meshes
- `shdiv`—Divergence element on triangular or tetrahedral meshes
- `shbub`—Bubble element on 1D, triangular, or tetrahedral meshes
- `shdisc`—Discontinuous element
- `shdens`—Density element
- `shgp`—Gauss-point element
- `shuwhelm`—Scalar plane-wave basis element

For more information about these elements see "Finite Elements" on page 452 of the *COMSOL Multiphysics Reference Guide* as well as the entries `shlag`, `shherm`, and so on in the same manual. There are also a number of shape function classes in the Structural Mechanics Module for beams, plates, and shells.

Note: When using shape function objects in COMSOL Script, their constructor must be specified as a string.

SHLAG—THE LAGRANGE ELEMENT

The syntax of a Lagrange shape function object is

```
shlag(k,basename);
```

of order *k* for the variable *basename*. For instance, create a Lagrange shape function by typing

```
s = 'shlag(1,''phi'')';
```

which means that linear elements represent the variable `phi`.

SHHERM—THE HERMITE ELEMENT

The syntax of a Hermite shape function object is

```
shherm(k,basename);
```

of order *k* for the variable *basename*. The order *k* must be at least 3. For instance,

```
s = 'shherm(3,''v'')';
```

means that piecewise 3rd-degree polynomials represent the variable `v`.

SHARG_2_5—THE ARGYRIS ELEMENT

The syntax of a Argyris shape function object is

```
sharg_2_5(basename);
```

for the variable *basename*. For instance,

```
s = 'sharg_2_5(''u'')';
```

creates an Argyris element of order 5 for the variable `u` in 2D.

SHCURL—THE VECTOR ELEMENT

Syntaxes for a vector shape function object are

```
shcurl(k,basename);
shcurl(k,compnames);
```

of order *k* for the vector field name *basename* or list of vector field components *compnames*. To construct a vector shape function object for the field name `u`, type

```
s = 'shcurl(2,''u'')';
```

and to create a second vector shape function object for the vector field components `Hx` and `Hy` type

```
s = 'shcurl(1,{''Hx'',''Hy''} )';
```

in 2D.

SHDIV—THE DIVERGENCE ELEMENT

Syntaxes for a divergence shape function object are

```
shdiv(k,basename);
shdiv(k,compnames);
```

of order *k* for the vector field name *basename* or list of vector field components *compnames*.

SHBUB—THE BUBBLE ELEMENT

The syntax for a bubble element shape function object is

```
shbub(mdim,basename);
```

where *mdim* is the dimension of the mesh elements for which the shape functions exist, and the variable *basename* is its name.

SHDISC—THE DISCONTINUOUS ELEMENT

The syntax for a discontinuous element shape function object is

```
shdisc(mdim,order,basename);
```

where *mdim* is the dimension of the mesh elements for which the shape functions exist, *order* is the order (a positive integer) of the variable, and the variable *basename* is its name.

FEM.SHAPE—DEFINITION OF SHAPE FUNCTIONS

The field `fem.shape` is a cell array of strings containing shape function objects. For example, the declaration

```
fem.shape = {'shlag(1,''u'')' 'shlag(2,''u'')' 'shcurl(2,''A'')'};
```

defines three shape function objects. You can choose the domains on which these objects should be active by using the fields `fem.equ.shape`, `fem.bnd.shape`, `fem.edg.shape`, and `fem.pnt.shape` (see "Domains of Usage for Shape Functions" on page 23).

Note: The syntax

```
fem.shape = {shlag(1,'u') shlag(2,'u') shcurl(2,'A')};
```

is obsolete. It is still supported for command-line modeling with MATLAB but does not work with COMSOL Script.

Alternatively, `fem.shape` can be a numeric vector. In this case its length must match the `dim` variables (see page 39). The software then assigns shape function objects for the `dim` variables that provide Lagrange elements of the order in `fem.shape`. For example,

```
fem.dim = {'u' 'v' 'p'};
fem.shape = [2 2 1];
```

is equivalent to

```
fem.dim = {'u' 'v' 'p'};
fem.shape = {'shlag(2,''u'')' 'shlag(2,''v'')' 'shlag(1,''p'')'};
```

A further alternative is to give a single number k in `fem.shape`, which results in Lagrange elements of order k for all `dim` variables.

If the model includes mesh cases, the field `fem.shape` can be a structure containing the following fields:

- `fem.shape.default` is the shape functions for mesh case 0. It has the same syntax as described above. These shape functions are also used in all mesh cases that do not occur in `fem.shape.mind`.
- `fem.shape.case` is a cell array of shape function definitions. Each entry `fem.shape.case{`*i*`}` has the same syntax as described earlier, and it applies to the mesh cases `fem.shape.mind{`*i*`}`. The default is an empty cell array `fem.shape.case`.
- `fem.shape.mind{`*i*`}` is a cell array of vectors of positive mesh case numbers. The default is `fem.shape.mind = {1 2 ... `*n*`}`, where *n* is the length of `fem.shape.case`.

Note: If you do not provide `fem.shape`, the software uses the value 1, that is, all `dim` variables have linear elements. In the COMSOL Multiphysics user interface, however, most application modes use a default Lagrange element order of 2.

DOMAINS OF USAGE FOR SHAPE FUNCTIONS

In the fields `fem.equ.shape`, `fem.bnd.shape`, `fem.edg.shape`, and `fem.pnt.shape` you can specify where to use the shape function objects in `fem.shape`. For example,

```
fem.shape = ...
  {'shlag(1,''u'')' 'shlag(2,''u'')' 'sharg_2_5(''v'')'};
fem.equ.shape = { [1 3] [2 3] [] 3 };
```

means that on the first subdomain group only the first and third shape function objects (`shlag(1,'u')` and `sharg_2_5('v')`) are active. On the second subdomain group the second and third shape function objects are active. On the third subdomain group no shape function objects are defined, while on the fourth subdomain group only `sharg_2_5('v')` is active. The variable `u` is defined in subdomain groups 1 and 2, having linear elements in subdomain group 1 and quadratic elements in subdomain group 2. A problem can arise if these subdomain groups are adjacent because COMSOL Multiphysics does not take care of the "hanging nodes" that appear. Thus you should not mix elements for the same variable in adjacent subdomains.

If you do not provide the field `fem.equ.shape`, all shape function objects in `fem.shape` apply in all subdomain groups.

Similarly, the field `fem.bnd.shape` is a cell array that specifies for each boundary group which shape function objects are active. If `fem.bnd.shape` is not given, then it is inherited from `fem.equ.shape`. This means that a shape function object that is active in a subdomain is also active on the boundary of that subdomain (as well as boundaries lying within the subdomain).

In 3D the field `fem.edg.shape` similarly specifies the usage of shape function objects on edge groups. If you do not provide it, it inherits the usage on subdomains and boundaries. That is, a shape function object that is active on some subdomain or some boundary is also active on all edges that touch this subdomain or boundary.

In 2D and 3D the field `fem.pnt.shape` similarly specifies usage on vertices. It inherits its default settings from subdomains, boundaries, and edges.

FRAME FOR SHAPE FUNCTIONS

For problems with moving meshes and more than one frame you must indicate on which frame each shape function lives. To do this, specify an additional property `frame` for the shape function. When adding this property you must use the shape function's property-value syntax. For example, if the shape function `shlag(2,'u')` should live on the frame `ale`, then the correct declaration reads

```
fem.shape = ...
  {'shlag(''order'',2,''basename'',''u'',''frame'',''ale'')'};
```

OVERVIEW OF VARIABLE TYPES

The variables you can use in expressions belong to the following categories:

- Geometric variables, see page 18.
- Field variables. These are the *dependent variables* that describe the physical quantities that you model. Field variables can be functions of the space coordinates and time. In the finite element discretization they are expressed in terms of the degrees of freedom.
- Constants. Besides the predefined constants (for instance `pi`, `i`, and `j`), you can use your own as noted in the next section.
- Special variables. These are the phase angle `phase`, the time `t` for a time-dependent problem, and the eigenvalue `lambda` for an eigenvalue problem. The variable `phase` is used only in postprocessing; it is equal to 0 in the solvers and the assembly (unless you override it with `fem.const`). The eigenvalue `lambda` cannot be used in equation coefficients; you can use it only in postprocessing.
- Weak form meta variables. These are formed by appending `_test` to a field variable name. They can be used only in weak terms, see "The Weak Form" on page 47.

The field variables are further divided into the following groups (the following sections provide more details for most of these types):

- *Shape function variables*, those that the shape function objects define, see "Shape Functions" on page 20. The `dim` variables are a subset of these, see page 39.
- *Expression variables*, those that are defined as expressions in terms of other variables.
- *Application mode variables* and *application scalar variables* are like expression variables except they are defined by an application mode. See "Application Structures" on page 56 and the application mode descriptions in the *COMSOL Multiphysics Modeling Guide*.
- *Material variables* are like expression variables except they are defined by a materials library.
- *Boundary coupled shape variables* are certain derivatives that a priori have no well-defined values on the boundaries. The values of some of these variables are formed as the average of the values in the adjacent subdomains.

- *Coupling variables* are variables defined in one domain in terms of other variables in possibly distant domains. This makes it possible to have nonlocal dependencies in a model.
- *Equation variables* are shorthands for certain terms in equations in coefficient and general form. Normally they should be used only in postprocessing.
- *Boundary coupled equation variables.*

CONSTANTS

In the field `fem.const` you define constants. Do this using a cell array with alternating constant names (strings) and numeric values. For example,

```
fem.const = { 'c0' 3e8 'mu0' 4*pi*1e-7 };
```

This gives `c0` the value $3 \cdot 10^8$ and `mu0` the value $4\pi \cdot 10^{-7}$.

An alternative syntax is to let `fem.const` be a structure with the fields corresponding to the names of the constants.

```
fem.const.c0 = 3e8;
fem.const.mu0 = 4*pi*1e-7;
```

is equivalent to the same cell array.

EXPRESSION VARIABLES

You can define new field variables in terms of other variables—known as expression variables or simply expressions, for short—in the fields `fem.globalexpr`, `fem.expr`, `fem.equ.expr`, `fem.equ.bnd.expr`, `fem.bnd.expr`, `fem.edg.expr`, and `fem.pnt.expr`. This can be convenient, for example, for simplifying the model definition if your equations contain the same expression several times, or for creating a common field variable for postprocessing when you have separate field variables in different domains describing the same physical property.

In the field `fem.expr` you put variable definitions that apply on all domain groups (of all dimensions) in the geometry. `fem.expr` is a cell array of alternating variable names and expressions, or a structure where the fields are the variable names. If a model contains more than one geometry, and, consequently, is described by an extended FEM structure, each ordinary FEM structure can have its own `expr` field with definitions that apply only to the corresponding geometry.

If you want to define expressions that should apply globally in a multiple-geometry model—that is, on all domain groups of all geometries in the model—put them in the field `fem.globalexpr` (now assuming that `fem` is the name of the extended FEM structure). Here you can also put expressions that are geometry independent, for

example, initial conditions for ODEs. The syntax for the `globalexpr` field is the same as that for the `expr` field.

In `fem.equ.expr` you put variable definitions that are active on subdomains. For example,

```
fem.equ.expr = { 'W' 'B*H/2' 'div' 'ux+vy' };
```

defines `W = B*H/2`, and `div = ux+vy` on all subdomain groups. The defining expressions in the `fem.equ.expr` can be 1D cell arrays. For instance

```
fem.equ.expr = { 'v' {'a+b' 'a-b'} 'w' 'pi*x*b' };
```

defines `v` to be `a+b` on subdomain group 1, but it equals `a-b` on subdomain group 2. The use of an empty vector `[]` instead of an expression means that the variable is not defined on the corresponding subdomain group. For example,

```
fem.equ.expr = { 'v' {'a+b' []} };
```

means that `v = a+b` on subdomain group 1, and that `v` is undefined on subdomain group 2.

An alternative syntax is to let `fem.equ.expr` be a structure where the fields are the variable names. For example,

```
fem.equ.expr.v = {'a+b' 'a-b'};
fem.equ.expr.w = 'pi*x*b';
```

is equivalent to

```
fem.equ.expr = { 'v' {'a+b' 'a-b'} 'w' 'pi*x*b' };
```

Similarly, you add variable definitions on boundaries, edges, and vertices in `fem.bnd.expr`, `fem.edg.expr`, and `fem.pnt.expr`, respectively.

To define interior mesh boundary variables use the field `fem.equ.bnd.expr`.

USER COMMENTS AND DESCRIPTIONS

You can optionally provide descriptions of constants and expressions defined in `fem.const`, `fem.expr`, `fem.*.expr`, `fem.globalexpr`, and `fem.ode` in the field `fem.descr`. For each constant field or expression field present in the FEM structure, define an associated subfield of `fem.descr` with the same name. For example,

```
fem.const = {'g' '9.81'};
fem.descr.const = {'g' 'Acceleration due to gravity'};
```

Generally, `fem.descr.*` is a cell array of alternating variable names and description strings. In a multiple-geometry model each FEM structure can have its own `descr` field.

BOUNDARY-COUPLED SHAPE VARIABLES

COMSOL Multiphysics can extrapolate the gradients of the dependent variables to the boundary of the domain making the boundary-coupled shape variables available. Use the property `cplbndsh` to the function `meshextend` in order to turn on the generation of these variables:

```
fem.xmesh = meshextend(fem, 'cplbndsh','on');
```

You find documentation about this set of variables in the section "Boundary Coupled Shape Variables" on page 174 in the *COMSOL Multiphysics User's Guide*.

EQUATION VARIABLES

When you use the coefficient or general solution forms, COMSOL Multiphysics can make variables representing certain terms in the coefficient-form equation available — through equation variables. The field `xfem.solform` in the extended FEM structure or the property `solform` to the `meshextend` function controls the solution form. In addition, the `meshextend` property `Eqvars` controls the generation of the equation variables. It is set to `on` by default, but the variables are not available unless the solution form is coefficient or general. For more information about this set of variables see the section "Special Variables" on page 175 in the *COMSOL Multiphysics User's Guide*.

BOUNDARY-COUPLED EQUATION VARIABLES

When you use solution forms coefficient or general, COMSOL Multiphysics can extrapolate the equation variables to the boundary—through boundary-coupled equation variables. You control the solution form with the field `xfem.solform` in the extended FEM structure or with the property `Solform` in the `meshextend` function. In addition, the `meshextend` property `cplbndeq` controls the generation of the equation variables. It is set to `on` by default, but the variables are not available unless the solution form is coefficient or general. This set of variables is documented in the section "Boundary-Coupled Equation Variables" on page 262 of the *COMSOL Multiphysics Modeling Guide*. Use boundary-coupled equation variables with caution in equations to avoid circular definitions.

USER-DEFINED FUNCTIONS

You can define additional functions for use in the model. Each function definition appears as a structure in `fem.functions`, which is a cell array with one function definition in each cell. There are two types of functions:

- Inline functions (analytic functions), which you indicate with the string `'inline'` in the `type` field. For these functions, you define the name of the function (with input arguments), a definition of the function, and a definition of the derivative of the function (the software can compute the derivative automatically). As an example, define `inlinefun` as a function of x and y: $\sin(x)\cos(y)$:

  ```
  fem.functions{1}.type = 'inline';
  fem.functions{1}.name = 'inlinefun';
  fem.functions{1}.expr = 'sin(x)*cos(y)';
  fem.functions{1}.dexpr = {'diff(sin(x)*cos(y),x)',
  'diff(sin(x)*cos(y),y)'}
  ```

  The last field, `dexpr`, contains the derivative with respect to each input variable as a cell array of strings. This example uses the built-in differentiation operator `diff`, which is the setting you get when using the **Auto** setting in the **Derivatives** area of the **Functions** dialog box in the COMSOL Multiphysics user interface.

  Function which can generate complex values from real data must have the `complex` field set to `'true'`. The `linear` property decides if the function is treated as linear when deciding whether to reassemble the Jacobian at each time step/iteration or not.

- Interpolation functions, which you indicate with the string `'interp'` in the `type` field. For these functions you define the name of the function, the interpolation method, the extrapolation method, and the data or the file containing the data. As an example, define `interpfun1` using values of x and $f(x)$:

  ```
  fem.functions{2}.type = 'interp';
  fem.functions{2}.name = 'interpfun1';
  fem.functions{2}.method = 'cubic';
  fem.functions{2}.extmethod = 'extrap';
  fem.functions{2}.x = {'1' '2' '3' '4'};
  fem.functions{2}.data = {'1' '4' '9' '16'}
  ```

  If you instead store the data for the interpolation in a text file, `C:\COMSOL34\mydata.txt` for example, replace the `x` and `data` fields with:

```
fem.functions{2}.filename = 'C:\COMSOL34\mydata.txt';
```

You can also supply the interpolation data as grid data in cell arrays for x, y (optional), and z (optional) coordinates and the interpolation data. Store the coordinates and data in the fields `x`, `y`, `z`, and `data`.

The `method` field contains the extrapolation method: `'nearest'` for nearest neighbor interpolation, `'linear'` for linear interpolation, `'cubic'` for piecewise cubic interpolation, or `'spline'` for cubic spline interpolation.

The `extmethod` field contains the extrapolation method: `'const'` for a constant-value cubic interpolation, `'extrap'` for extrapolation, or a string containing the value for all values outside of the range (for example, `'0'` or `'NaN'`).

- Piecewise functions (analytic functions with different definitions in different intervals), which you indicate with the string `'piecewise'` in the `type` field. For these functions, you define the name of the function (with input argument), what subtype it belongs to, the intervals, the extrapolation method, optional smoothing, and the definition of the function in the different intervals. As an example, define `piecewisefun` as a function of x:

$$\text{piecewisefun}(x) = 2.3x^2 - 4.5x^4 \qquad 100 \le x < 200$$

$$\text{piecewisefun}(x) = \frac{1.5}{x^3} - 6.3 \qquad 200 \le x < 1000$$

```
fem.functions{1}.type = 'piecewise';
fem.functions{1}.name = 'piecewisefun(x)';
fem.functions{1}.subtype = 'poly';
fem.functions{1}.extmethod = 'extrap';
fem.functions{1}.intervals = {'100','200','1000'};
fem.functions{1}.expr = {{'2','2.3','4','-4.5'},...
{'-3','1.5','0','-6.3'}};
```

The last field, `expr`, is a cell array with the definition in the different intervals. The syntax depends of the subtype. There are three different subtypes:

- `poly` (polynomial functions)
- `exppoly` (exponential function where the exponent is a polynomial)
- `general` (any type of function).

You specify the polynomials or general expressions in each subinterval in the `expr` field. For polynomial and exponential polynomial functions, you specify the polynomial exponent and corresponding coefficient as consecutive pairs in a cell

array. For a general function simply type in the expressions in the different intervals as they are defined.

Because the given expressions may be discontinuous at the interval boundaries, an optional smoothing option is available. If given, the `smoothzone` field specifies a relative size of the smoothing zone, interpreted as the fraction of each interval length that should be smoothed at each end of the interval. The `smoothorder` field gives the number of continuous derivatives that must exist at the boundary between smoothzone and interval.

The `extmethod` field contains the extrapolation method: `'const'` for a constant value outside the interval being the same as the value at the end of the interval, `'extrap'` for extrapolation, or a string containing the value for all values outside of the range (for example, `'0'` or `'NaN'`), `'none'` extrapolation is deactivated resulting in errors or `NaN` values for out-of-range values.

A function that can generate complex values from real data must have the `complex` field set to `'true'`. The `linear` property decides if the function is treated as linear when deciding whether to reassemble the Jacobian at each time step/iteration or not.

Units

For general information about available units and the unit support in COMSOL see the section "Using Units" on page 183 in the *COMSOL Multiphysics User's Guide*.

FEM.UNIT—BASE UNIT SYSTEM

The field `fem.units` contains the name of the model's base unit system. Valid values are `'none'`, `'SI'`, or the name of one of the unit-system files defined in the subfolder `data/unitsystems`.

SYNTAX FOR THE UNIT SYSTEM DEFINITION FILES

The unit system M-files define a variable `unitsystem`, which is a structure with the following fields names:

- `name`
- `base`
- `derived`
- `additional`

The `name` field contains the name of the unit system. This is what shows up in the base unit system list boxes in COMSOL Multiphysics. If the `name` field is missing, the software uses the file name instead.

The `base` field is a structure that defines the base units. It can have the following fields:

- `length`
- `mass`
- `time`
- `current`
- `temperature`
- `intensity`
- `substance`
- `radius`.

  It is not necessary to define a unit for the radius because COMSOL Multiphysics then generates it automatically. In that case it gets the same name as the unit for length plus `radius` and the same symbol as the unit for length plus `r`. In SI units, for example, `meterradius`, `metreradius`, and `mr` all represent the radius. The default name and symbol are the same as for length, so the user interface normally does not display the separate unit for radius. The radius helps to distinguish between properties such as torque (`N*mr`) and energy (`N*m`), and the definition of the radian is `m/mr`.

These fields are, in turn, structures with the fields:

- `units` sting or cell array of strings of unit names)
- `symbols` (sting or cell array of strings of unit symbols)
- `scale` (a scale relative to the SI system)
- `offset` (specified in its own units, used only for temperature)

For missing base fields, COMSOL Multiphysics uses the SI definition instead. This means that you only have to specify the base units that differ from the SI units.

If you specify several symbols, COMSOL Multiphysics uses the first symbol as the default symbol for the unit. If the symbol field is missing, the first unit name becomes the default symbol for the unit.

The `derived` field is a structure that defines derived units. The derived units can be expressed as integer powers of base units and have no additional scaling or offset. Each derived unit can have the following fields:

- `dimension` (string or integer array of base unit powers)
- `units`
- `symbols`

If the `dimension` field is missing, the name of the structure itself must be a dimension that COMSOL Multiphysics knows. If the `dimension` field is a string, it must also be a known dimension.

The `additional` field is a structure that defines additional units, that is, units that requires some scaling. The additional units can have the following fields:

- `units`
- `symbols`
- `scale` (the scale relative to this unit system)
- `offset`

Materials/Coefficients Libraries

FEM.LIB—LIBRARY DATA

The field `fem.lib` is a structure that you can use to define a *materials/coefficients library*, which defines material parameters, cross sections used in structural mechanics, and coordinate systems. The fields in `fem.lib` correspond either to materials, cross sections, or a coordinate system.

For example,

```
fem.lib.mat{1}.name = 'Water';
fem.lib.mat{1}.varname = 'mat1';
fem.lib.mat{1}.variables.C = '4200';
fem.lib.mat{1}.variables.k = 'k(T)';
fem.lib.mat{1}.functions{1}.type = 'inline';
fem.lib.mat{1}.functions{1}.name = 'k(T)';
fem.lib.mat{1}.functions{1}.expr = '0.0015*T+0.1689';
fem.lib.mat{1}.functions{1}.dexpr = {'diff(0.0015*T+0.1689,T)'};
```

defines the material `mat1`. This creates the *material variables* `mat1_C`, which is defined by the corresponding expression, and the local material function `mat1_k(T)`.

You can add units to the materials data using the COMSOL Multiphysics unit syntax, appending the unit in brackets. See "Using the Unit Syntax" on page 187 in the *COMSOL Multiphysics User's Guide* for more information. For example, to define the heat capacity in SI units, use

```
fem.lib.mat{1}.variables.C = '4200[J/(kg*k]';
```

This makes it possible to use this value also in models that use a base unit system other than SI. For the function that defines the thermal conductivity use

```
fem.lib.mat{1}.variables.k = 'k(T[1/K])[W/(m*K)]';
```

In this case, the syntax `[1/K]` converts the temperature value in the base unit system to kelvin.

The following tables show all supported materials properties and their SI units.

Note: The application modes in COMSOL Multiphysics and the add-on modules support a subset of these materials properties depending on the application.

TABLE 3-4: PHYSICS MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
C	Heat capacity	J/(kg·K)
k	Thermal conductivity	W/(m·K)
rho	Density	kg/m^3

TABLE 3-5: ELASTIC MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
C01	Model parameter (hyperelastic material)	Pa
C10	Model parameter (hyperelastic material)	Pa
Delastic2D	Elasticity matrix	Pa
Delastic3D	Elasticity matrix	Pa
E	Young's modulus	Pa
ETiso	Isotropic tangent modulus	Pa
ETkin	Kinematic tangent modulus	Pa
Ex	Young's modulus	Pa
Ey	Young's modulus	Pa
Ez	Young's modulus	Pa

TABLE 3-5: ELASTIC MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
Gxy	Shear modulus	Pa
Gxz	Shear modulus	Pa
Gyz	Shear modulus	Pa
Syfunc	Yield function	Pa
Sys	Yield stress level	Pa
alpha	Thermal expansion coefficient	1/K
alphavector2D	Thermal expansion vector	1/K
alphavector3D	Thermal expansion vector	1/K
alphax	Thermal expansion coefficient	1/K
alphay	Thermal expansion coefficient	1/K
alphaz	Thermal expansion coefficient	1/K
kappa	Initial bulk modulus (hyperelastic material)	Pa
mu	Initial shear modulus (hyperelastic material)	Pa
nu	Poisson's ratio	-
nuxy	Poisson's ratio	-
nuxz	Poisson's ratio	-
nuyz	Poisson's ratio	-
rho	Density	kg/m^3

TABLE 3-6: ELECTRIC MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
epsilonr	Relative permittivity	-
epsilonrtensor2D	Relative permittivity	-
epsilonrtensor3D	Relative permittivity	-
mur	Relative permeability	-
murtensor2D	Relative permeability	-
murtensor3D	Relative permeability	-
n	Refractive index	-
ntensor2D	Refractive index	-
ntensor3D	Refractive index	-
sigma	Electrical conductivity	S/m

TABLE 3-6: ELECTRIC MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
sigmatensor2D	Electrical conductivity	S/m
sigmatensor3D	Electrical conductivity	S/m

TABLE 3-7: FLUID MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
D	Diffusion coefficient	m^2/s
cs	Speed of sound	m/s
dtensor2D	Diffusion coefficient	m^2/s
dtensor3D	Diffusion coefficient	m^2/s
eta	Dynamic viscosity	Pa·s
gamma	Ration of specific heats	-
kperm	Permeability	m^2
nu0	Kinematic viscosity	m^2/s
rho	Density	kg/m^3

TABLE 3-8: PIEZOELECTRIC MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
cE	Elasticity matrix	Pa
d	Coupling matrix, strain-charge form	C/N
e	Coupling matrix, stress-charge form	C/m^2
epsilonrS	Permittivity matrix, stress-charge form	-
epsilonrT	Permittivity matrix, strain-charge form	-
sE	Compliance matrix	1/Pa

TABLE 3-9: THERMAL MATERIALS PROPERTIES

FIELD NAME	PROPERTY	SI UNIT
C	Heat capacity	J/(kg·K)
epsilon	Surface emissivity	-
h	Heat transfer coefficient	$W/(m^2·K)$
k	Thermal conductivity	W/(m·K)
ktensor2D	Thermal conductivity	W/(m·K)
ktensor3D	Thermal conductivity	W/(m·K)
rho	Density	kg/m^3

In the graphical user interface, you can read and write the materials library from and to a file; see "Using the Materials/Coefficients Library" on page 223 in the *COMSOL Multiphysics User's Guide*.

Another example is

```
fem.lib.sec{1}.name = 'HEA 100';
fem.lib.sec{1}.varname = 'sec1';
fem.lib.sec{1}.variables.A = '21.2*1E-4';
fem.lib.sec{1}.variables.Iyy = '349.0*1E-8';
fem.lib.sec{1}.variables.Izz = '134.0*1E-8';
fem.lib.sec{1}.variables.J = '5.2647e-008';
fem.lib.sec{1}.variables.heighty = '100*1E-3';
fem.lib.sec{1}.variables.heightz = '96*1E-3';
```

which defines a cross section `sec1`.

Coordinate systems are defined as

```
fem.lib.coord1.type = 'coordsys';
fem.lib.coord1.name = 'Coordnate system 1';
fem.lib.coord1.T = {'cos(pi/90)' '-sin(pi/90)';
                    'sin(pi/90)' 'cos(pi/90)'};
```

The field `T` defines the transformation matrix of the coordinate system. This is an $n \times n$ matrix where n is the number of space dimensions, and it transforms the coordinates in the local coordinate system to the global coordinate system. The field name `coord1` is arbitrary; the `type` field indicates which kind of object you define.

Pairs

In the fields `fem.bnd.pair`, `fem.edg.pair`, and `fem.pnt.pair` you can define *identity pairs* and *contact pairs* (only in `fem.bnd.pair`). These fields are in a cell array where each item is a structure defining a pair. The structures have the following fields:

- `pair.type = 'identity' | 'contact'` The type: identity pair or contact pair.
- `pair.name`. The name of the pair.
- `pair.suffix`. The suffix for contact variable names. This field is available only for contact pairs (contact pairs are available for modeling of structural contact in the Structural Mechanics Module and the MEMS Module).
- `pair.contname`. The name of the variable determining the contact. This field is available only for contact pairs.

- `pair.src`. A structure with data for the source of the pair.
- `pair.dst`. A structure with data for the destination of the pair.

The fields `pair.src` and `pair.dst` are structures with the same fields.

- `pair.src.dl`. An integer array with the source domains.
- `pair.src.operator`. The name of the operator mapping from the source to the destination.
- `pair.src.map.frame`. The frame to use when evaluating the coordinates. This field is available only for identity pairs.

For example,

```
pair.type = 'identity';
pair.name = 'Pair 1';
pair.src.dl = [1 2];
pair.src.operator = 'src2dst';
pair.dst.dl = [3 4];
pair.dst.operator = 'dst2src';
fem.bnd.pair={pair};
```

defines an identity boundary pair with the source boundaries 1 and 2 and the destination boundaries 3 and 4.

Note: Contact pairs are available with the Structural Mechanics Module and the MEMS Module.

Equations and Constraints

This section describes the syntax for equations and constraints.

FEM.FORM—FORM OF EQUATIONS

The field `fem.form` can be one of the strings `'coefficient'`, `'general'`, or `'weak'`. The default is `'coefficient'`. It is possible to use weak equations in all three forms; they are added to the weak reformulation of the coefficient or general formulation. If you use `fem.form = 'weak'` the software ignores contributions from the coefficient and general syntaxes.

FEM.DIM—NAMES OF VARIABLES IN EQUATIONS

To specify the names for the dependent variables $u_1, u_2, \ldots, u_N$ that occur in the coefficient and general forms, use the field `fem.dim`. For example,

```
fem.dim = {'u' 'v' 'p'};
```

for a system with $N = 3$. These are the *dependent variables*, and they are subset of the shape function variables (see "Variables and Functions" on page 25). In the case of a single variable you can skip the braces:

```
fem.dim = 'v';
```

Alternatively, `fem.dim` can be a number:

```
fem.dim = 3;
```

which gives the variable names `u1`, `u2`, and `u3`. However,

```
fem.dim = 1;
```

gives the variable name `u`, which is the default.

You can override the variables in `fem.dim` on subdomains, boundaries, edges, and points by using the fields `fem.equ.dim`, `fem.bnd.dim`, `fem.edg.dim`, and `fem.pnt.dim`, respectively.

In an FEM structure that you export from the COMSOL Multiphysics user interface, the dependent variables appear in `fem.equ.dim` instead of `fem.dim`.

THE COEFFICIENT FORM

Recall that a time-dependent problem in coefficient form can be written as

$$\begin{cases} e_{a\,lk}\dfrac{\partial^2 u_k}{\partial t^2} + d_{a\,lk}\dfrac{\partial u_k}{\partial t} + \nabla\cdot(-c_{lk}\nabla u_k - \alpha_{lk}u_k + \gamma_l) + \beta_{lk}\cdot\nabla u_k + a_{lk}u_k = f_l & \text{in } \Omega \\ \mathbf{n}\cdot(c_{lk}\nabla u_k + \alpha_{lk}u_k - \gamma_l) + q_{lk}u_k = g_l - h_{ml}\mu_m & \text{on } \partial\Omega \\ h_{mk}u_k = r_m & \text{on } \partial\Omega \end{cases}$$

where k and l range from 1 to N, and m ranges from 1 to M, where $M \le N$. The PDE and boundary coefficients in these equations are stored in the subfields `ea`, `da`, `c`, `al`, `ga`, `be`, `a`, and `f` of `fem.equ`, and the subfields `q`, `g`, `h`, and `r` of `fem.bnd`. In these fields nested cell arrays represent the coefficients. The values of coefficients that you do not specify are 0.

Examples

For a system with $N = 2$,

```
fem.equ.f = { {'x' 'y+1'} {'x*y' 3} };
```

means that on subdomain group 1 you have $f_1 = x$ and $f_2 = y+1$, while on subdomain group 2 you have $f_1 = xy$ and $f_2 = 3$. The outer pair of braces corresponds to the different subdomain groups, and the inner pairs of braces correspond to the equation index l.

The coefficient a is a matrix with components a_{lk}, so it can be entered as

```
fem.equ.a = { {'x' 1; 'y' 0} };
```

(if $N = 2$) using the syntax for matrices. In this case, the outermost pair of curly braces contains only one entry. In such cases that entry applies to all subdomain groups. Thus,

$$a = \begin{bmatrix} x & 1 \\ y & 0 \end{bmatrix}$$

on all subdomain groups.

Because the coefficient β_{lk} is a vector, you need a third level of braces to represent β. For example (in case $N = 2$ and the space dimension is 3):

```
fem.equ.be = { { {1 0 0} {2 4 5}; {'x' 3 4} {1 2 3} } };
```

means that

$$\beta_{11} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \beta_{12} = \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix}, \beta_{21} = \begin{bmatrix} x \\ 3 \\ 4 \end{bmatrix}, \beta_{22} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

on all subdomain groups, or, put in matrix form,

$$\beta = \begin{bmatrix} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 2 & 4 & 5 \end{bmatrix} \\ \begin{bmatrix} x & 3 & 4 \end{bmatrix} & \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \end{bmatrix}$$

General Syntax for Coefficient Specification

There is a general syntax for these nested cell arrays. The syntax has some shortcuts, but except for these, the levels of braces correspond to the following:

- Level 1: The different domains or domain groups.

- Level 2: The indices k and l, which enumerate equations or variables.
- Level 3: The indices i and j, which enumerate space coordinates.

Level 1 Coefficients

The content of any of the coefficient fields `ea`, `da`, `c`, `al`, `ga`, `be`, `a`, `f`, `q`, `g`, `h`, and `r` is a *Level 1 coefficient*. A Level 1 coefficient can be a cell array or something else. The latter case becomes the former case if you include the coefficient in braces. Thus,

```
fem.equ.c = 3;
```

is equivalent to

```
fem.equ.c = { 3 };
```

The entries in a Level 1 coefficient cell array are called *Level 2 coefficients*. If the cell array has length 1, then its single entry is valid in all subdomain/boundary groups. If the cell array has length >1, then its nth entry applies in the nth subdomain/boundary group.

Level 2 Coefficients

A Level 2 coefficient can be a cell array or something else. The software interprets the second case as a cell array with one entry. Thus you need to consider only the case of a cell array. The entries in this cell array are called *Level 3 coefficients*. The interpretation of the cell array depends on its dimensions according to the following table:.

COEFFICIENT	NUMBER OF CELL ARRAY ELEMENTS AT LEVEL 2	MATRIX/VECTOR INTERPRETATION
$e_a\ d_a\ c\ \alpha\ \beta\ a\ q\ h$	1	Diagonal N x N matrix with the same element throughout the diagonal
$e_a\ d_a\ c\ \alpha\ \beta\ a\ q$	N	Diagonal N x N matrix with the given elements on the diagonal
$e_a\ d_a\ c\ \alpha\ \beta\ a\ q$	$N(N+1)/2$	Symmetric (see below) N x N matrix, given by listing the columns in the upper triangular part in a vector
$e_a\ d_a\ c\ \alpha\ \beta\ a\ q$	N^2	Full N x N matrix given as cell array
$\gamma f g$	1	Vector of length N with the same element in all components
$\gamma f g$	N	Vector with the given components

COEFFICIENT	NUMBER OF CELL ARRAY ELEMENTS AT LEVEL 2	MATRIX/VECTOR INTERPRETATION
h	MN	Full M x N matrix given as cell array
r	M	Vector with the given components

The case of a c coefficient given as a vector with $N(N+1)/2$ components is interpreted as a symmetric matrix in the sense that $c_{kl} = c_{lk}^T$, where the T denotes transpose.

Level 3 Coefficients

The meaning of Level 3 coefficients is shown in the following table:.

FIELD NAME	INTERPRETATION OF LEVEL 3 COEFFICIENT
`fem.equ.ea`	e_{alk} (scalar)
`fem.equ.da`	d_{alk} (scalar)
`fem.equ.c`	c_{lk} (scalar or matrix)
`fem.equ.al`	α_{lk} (vector)
`fem.equ.ga`	γ_l (vector)
`fem.equ.be`	β_{lk} (vector)
`fem.equ.a`	a_{lk} (scalar)
`fem.equ.f`	f_l (scalar)
`fem.bnd.q`	q_{lk} (scalar)
`fem.bnd.g`	g_l (scalar)
`fem.bnd.h`	h_{mk} (scalar)
`fem.bnd.r`	r_m (scalar)

At Level 3, the coefficients a, γ, and β are vectors of length n (the space dimension). Hence they are represented as cell arrays. The entries in these cell arrays are called *Level 4 coefficients.*

The coefficients e_a, d_a, a, f, q, g, h, and r are scalars at Level 3. Hence they are represented directly as Level 4 coefficients. (You can optionally surround these Level 4 coefficients with a pair of braces.)

Finally, the c coefficient is a scalar or matrix at Level 3. If it is a scalar it can be represented directly as a Level 4 coefficient. If it is a matrix it is represented as a cell

array of Level 4 coefficients according to the following table (where n is the space dimension).

NUMBER OF CELL ARRAY ELEMENTS	MATRIX INTERPRETATION OF LEVEL 3 COEFFICIENT FOR FEM.EQU.C
1	diagonal n x n matrix with the same element throughout diagonal
n	diagonal n x n matrix with the given elements on the diagonal
$n(n + 1)/2$	symmetric n x n matrix, given by listing the columns of the upper triangular part as a vector
n^2	full n x n matrix, given as cell array

Level 4 Coefficients

A Level 4 coefficient represents the scalar quantity d_{alk}, c_{lkij}, β_{lkj}, γ_{lj}, α_{lkj}, a_{lk}, f_l, q_{lk}, g_l, h_{mk}, or r_m. You can specify it in one of the following forms:

- A numeric scalar (possibly complex), such as `3.14`.
- A string containing a COMSOL Multiphysics expression. In this expression you can use variables (see "Variables and Functions" on page 25) and functions, including your own M-files. The functions must be vectorized, that is, take numeric arrays as inputs, and output a numeric array of the same size as the inputs. Functions with string arguments are not supported.

THE GENERAL FORM

In the general form you specify the quantities e_a, d_a, Γ, F, G, and R in the fields `fem.equ.ea`, `fem.equ.da`, `fem.equ.ga`, `fem.equ.f`, `fem.bnd.g`, and `fem.bnd.r`, respectively. The syntax is the same as in the coefficient form. You must also specify the partial derivatives c, α, β, a, f, q, and h that occur in the linearized problem. See "The Linear or Linearized Model" on page 366 in the *COMSOL Multiphysics User's Guide*. The most convenient way to compute these partial derivatives is to use the function `femdiff`:

```
fem = femdiff(fem);
```

but you can also specify them by hand. If you use your own M-files in expressions, you might need to specify differentiation rules for them; see the *COMSOL Multiphysics Reference Guide* entry on `femdiff`.

EXAMPLE—NAVIER'S EQUATIONS

This example uses *Navier's equations* for structural mechanics to illustrate the syntax for equations in coefficient and general form. The following equations describe the plane stress case of Navier's equations for structural mechanics:

$$\begin{cases} -\frac{\partial}{\partial x}\left((2\mathrm{G}+\mu)\frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial y}\left(G\frac{\partial u}{\partial y}\right) - \frac{\partial}{\partial x}\left(\mu\frac{\partial v}{\partial y}\right) - \frac{\partial}{\partial y}\left(G\frac{\partial v}{\partial x}\right) = F_x & \text{in } \Omega \\ -\frac{\partial}{\partial x}\left(G\frac{\partial u}{\partial y}\right) - \frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial x}\right) - \frac{\partial}{\partial x}\left(G\frac{\partial v}{\partial x}\right) - \frac{\partial}{\partial y}\left((2\mathrm{G}+\mu)\frac{\partial v}{\partial y}\right) = F_y & \text{in } \Omega \end{cases}$$

where

$$\begin{cases} G = \dfrac{E}{2(1+\nu)} \\ \mu = 2G\dfrac{\nu}{1-\nu} \end{cases}$$

E is Young's modulus and ν is Poisson's ratio. Using the coefficient form you can identify the c and f coefficients.

$$c = \begin{bmatrix} \begin{bmatrix} 2G+\mu & 0 \\ 0 & G \end{bmatrix} & \begin{bmatrix} 0 & \mu \\ G & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & G \\ \mu & 0 \end{bmatrix} & \begin{bmatrix} G & 0 \\ 0 & 2G+\mu \end{bmatrix} \end{bmatrix} \qquad f = \begin{bmatrix} F_x \\ F_y \end{bmatrix}$$

The following commands set up the data structures:

```
clear fem
E = 1e2; nu = 0.3; G = E/2/(1+nu); mu = 2*G*nu/(1-nu);
Fx = 0; Fy = -1;
fem.dim = {'u' 'v'};
fem.equ.c = { { {2*G+mu 0; 0 G} {0 mu; G 0}; ...
                {0 G; mu 0}     {G 0; 0 2*G+mu} } };
fem.equ.f = { {Fx Fy} };
```

There is just one subdomain, so the length of the outermost cell array is one. The Level 4 coefficients are just numeric values. For the `c` coefficient, alternatively use the diagonal syntax at the space-coordinate level, and the symmetry syntax at Level 2:

```
fem.equ.c = { { {2*G+mu G} {0 mu; G 0} {G 2*G+mu} } };
```

You can solve a full structural mechanics problem with geometry, mesh, and boundary conditions by typing the following lines of code. The boundary conditions fix the left side of the rectangle.

```
fem.bnd.h = { 0 0 0 1 };
fem.geom = rect2(0, 1, 0, 0.1);
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postplot(fem, 'tridata', 'v')
```

The `postplot` command plots the displacement v in the y direction. Because there is no specification of `fem.shape`, the model uses linear elements.

You can also use the general form when identifying coefficients:

$$\Gamma = \begin{bmatrix} \begin{bmatrix} -(2G+\mu)\dfrac{\partial u}{\partial x} - \mu\dfrac{\partial v}{\partial y} \\ -G\dfrac{\partial u}{\partial y} - G\dfrac{\partial v}{\partial x} \end{bmatrix} \\ \begin{bmatrix} -G\dfrac{\partial u}{\partial y} - G\dfrac{\partial v}{\partial x} \\ -\mu\dfrac{\partial u}{\partial x} - (2\mathrm{G}+\mu)\dfrac{\partial v}{\partial y} \end{bmatrix} \end{bmatrix} \qquad F = \begin{bmatrix} F_x \\ F_y \end{bmatrix}$$

The following commands set up the data structures:

```
clear fem
E = 1e2; nu = 0.3; G = E/2/(1+nu);
Fx = 0; Fy = -1;
fem.const = {'G' G 'mu' 2*G*nu/(1-nu)};
fem.dim = {'u' 'v'};
fem.form = 'general';
fem.equ.ga = { { {'-(2*G+mu)*ux-mu*vy' '-G*uy-G*vx'} ...
                 {'-G*uy-G*vx' '-mu*ux-(2*G+mu)*vy'} } };
fem.equ.f={{Fx Fy}};
```

The symbols `ux`, `uy`, `vx`, and `vy` refer to the `x` and `y` derivatives of `u` and `v`.

There is just one subdomain, so the length of the outermost cell array is one. Because this is a 2D problem for two variables, you have two cell array entries at both Level 2 and 3. The previous example uses the string expression syntax for the Level 4 coefficients.

You can now solve the same problem using quadratic elements:

```
fem.bnd.r = { 0 0 0 {'u' 'v'} };
fem.shape = 2;
fem.geom = rect2(0, 1, 0, 0.1);
fem = femdiff(fem);
fem.mesh = meshinit(fem);
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem,'v')
```

For linear problems you can normally choose the form which suits you best. In some cases it is easier to write down a problem using coefficient form, and in other cases the general form is more convenient.

DEACTIVATED EQUATIONS AND BOUNDARY CONDITIONS

If a `dim` variable is not defined on the entire geometry (see "Domains of Usage for Shape Functions" on page 23), then the corresponding equations and boundary conditions (in the general and coefficient forms) are deactivated on the subdomains and boundaries where the variable does not exist. More precisely, assume that `dim` variable number k is not defined on a domain. Then the software ignores the kth equation or Neumann boundary condition on that domain. Also, the coefficients q_{lk} and h_{mk} are set equal to 0 on the domain.

Example With a Slit

This example shows how to implement a slit by defining different variables on different domains. Assume you want to solve Laplace's equation on a unit disk with a slit on the negative x-axis. For boundary conditions use $u = \mathrm{atan2}(y, x)$ on the circular boundary and a linearly varying value inside the slit.

To model the slit, use two equations and two dependent variables that are active only for $y \geq 0$ and $y \leq 0$, respectively.

```
clear fem
fem.draw.s.objs = {circ2(0,0,1)};
fem.draw.c.objs = {curve2([-1 1],[0 0])};
fem.draw.p.objs = {point2(0,0)};
fem.geom = geomcsg(fem);
geomplot(fem, 'edgelabels', 'on', 'sublabels', 'on')
fem.dim = 2;
fem.equ.shape = {1 2};
fem.equ.c = 1;
```

This means that the variable `u1` is defined on Subdomain 1 ($y \geq 0$), and variable `u2` is defined on Subdomain 2 ($y \leq 0$).

The boundary conditions are more tricky. Only boundary conditions on boundaries adjacent to active subdomains have any effect. Assume you would like to use u_1 = atan2(y, x) on Boundaries 4 and 6, and u_2 = atan2($y-\varepsilon$, x) on Boundaries 3 and 5 to ensure a negative value also at the point $(-1, 0)$. On Boundary 1 set $u_1 = -\pi x$ and $u_2 = \pi x$, and on boundary 2 set $u_1 = u_2$:

```
fem.bnd.h = {1 1 {1 -1} 1};
fem.bnd.r = { {'atan2(y,x)' 0} {0 'atan2(y-eps,x)'} 0 ...
              {'-pi*x' 'pi*x'} };
fem.bnd.ind = [4 3 2 1 2 1];
```

Introduce an auxiliary variable `U` to be `u1` on Subdomain 1 and `u2` on Subdomain 2. Then activate the interior boundaries, create the mesh and the extended mesh, solve, and plot the solution:

```
fem.equ.expr = {'U' {'u1' 'u2'}};
fem.border = 'on';
fem.mesh = meshinit(fem,'hmax',0.05);
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem,'U')
```

FEM.BORDER—ACTIVATE BOUNDARY CONDITIONS ON INTERIOR BOUNDARIES

The field `fem.border` determines if you can set boundary conditions on interior boundaries (in the coefficient or general forms). `fem.border` can be one of the strings `'on'` or `'off'` or even a cell array of such strings (of a length that matches the `dim` variables). Alternatively, `fem.border` can be `1` or `0` or even a numeric vector of Ones and Zeros (of a length that matches the `dim` variables). In this case `1` is interpreted as `'on'` and `0` as `'off'`. If `fem.border{k}='off'` the software ignores the kth Neumann boundary condition on interior boundaries for the kth `dim` variable. Furthermore, the coefficients q_{lk} and h_{mk} are put equal to 0 on these interior boundaries. If `fem.border` is not a cell array or vector it applies to all `dim` variables. The default is `fem.border = 'off'`. See "Example With a Slit" in the previous section for an example of how to activate boundary conditions on interior boundaries.

THE WEAK FORM

For a description of the weak problem formulation see the chapter "The Weak Form" in the *COMSOL Multiphysics Modeling Guide*. To store equations in the weak form use the fields `fem.equ.weak`, `fem.bnd.weak`, `fem.edg.weak`, and `fem.pnt.weak`.

The field `fem.equ.weak` contains the integrand in the integral over subdomains. The field `fem.equ.weak` can be a string expression or a cell array. In the first case the

expression applies to all subdomain groups. In the second case, `fem.equ.weak{k}` applies to the *k*th subdomain group. The entry `fem.equ.weak{k}` can be an expression or a cell array of expressions. In the latter case, the expressions in the cell array are added.

In the expressions representing the integrand, the test function corresponding to a variable `v` is denoted `v_test`. `v_test` and has the same shape functions as `v`.

Similarly, the fields `fem.bnd.weak`, `fem.edg.weak`, and `fem.pnt.weak` contain integrands in integrals over boundaries, edges, and points, respectively. All these integrals appear on the right-hand side of the weak equation.

Time-Dependent Problems

For a time-dependent problem you store the terms containing time derivatives in `fem.equ.dweak`, `fem.bnd.dweak`, `fem.edg.dweak`, and `fem.pnt.dweak`. These have the same syntax as the `weak` fields except that the time derivatives must enter linearly. The time derivative of a variable `v` is denoted `v_time`, but you can also use the syntax `vt`. The integrals defined by the `dweak` fields are put on the left-hand side of the weak equation. You can also use `vtt` for the second time derivative.

Eigenvalue Problems

You specify eigenvalue problems just like time-dependent problems. COMSOL Multiphysics then interprets `vt` as $-\lambda v$, where λ is the eigenvalue.

Constraints in the Weak Problem Formulation

Use the fields `fem.equ.constr`, `fem.bnd.constr`, `fem.edg.constr`, and `fem.pnt.constr` to store constraints, which are implemented pointwise as described in "Discretization of the Equations" on page 470 in the *COMSOL Multiphysics Reference Guide*. The Jacobians of these constraints are calculated correctly. In contrast, when using the general form, the Jacobian of $\boldsymbol{R}$ only accounts for derivatives with respect to the `dim` variables (and not with respect to their derivatives, for example).

Specify the constraints on subdomains in the field `fem.equ.constr`, which can be an expression or a cell array. In the first case, the software constrains the expression in `fem.equ.constr` to be zero on all subdomain groups. In the second case, `fem.equ.constr{k}` applies to the *k*th subdomain group. The entry `fem.equ.constr{k}` can be an expression or a (possibly empty) cell array of expressions. Constraints force these expressions to be zero on the *k*th subdomain group.

Similarly, you define constraints on boundaries, edges, and vertices in `fem.bnd.constr`, `fem.edg.constr`, and `fem.pnt.constr`, respectively.

Constraint Forces

The default behavior is that the constraint force is derived from the constraint. If that does not give the wanted behavior it is possible to define a separate constraint expression which will be used in the constraint force calculation. Specify this expression in `fem.equ.constrf`, `fem.bnd.constrf`, `fem.edg.constrf`, and `fem.pnt.constrf`, respectively.

`fem.bnd.constrf` can also be used when a constraint is given in coefficient or general form using `fem.bnd.h` and `fem.bnd.r`.

Example—Poisson's Equation with a Point Source

As an example of a problem that needs some weak contribution, consider

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}=\delta$$

in the unit disk Ω, where δ is Dirac's δ function at the origin. Multiplying with a test function v and integrating by parts give:

$$0=v(0,0)-\int_{\Omega}\left(\frac{\partial v}{\partial x}\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\frac{\partial u}{\partial y}\right)dA$$

using the boundary conditions $u = 0$ and $v = 0$. (By introducing a Lagrange multiplier you can remove the boundary condition on v.) To solve the problem with COMSOL Multiphysics, type the following commands:

```
clear fem
fem.geom = geomcoerce('solid', {circ2 point2(0,0)});
geomplot(fem, 'pointlabels', 'on')
```

The second command introduces a vertex at the origin. The plot shows that the vertex number of the origin is 3:

```
fem.mesh = meshinit(fem);
fem.form = 'weak';
fem.pnt.weak = 'u_test';
fem.pnt.ind = {3};
fem.equ.weak = '-ux_test*ux-uy_test*uy';
fem.bnd.constr = 'u';
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem,'u','triz','u')
```

The exact solution

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

is infinite at the origin—a singularity that the finite element approximation, of course, cannot represent.

Alternatively, you can use the coefficient form with a weak contribution only for the point source:

```
clear fem
fem.geom = geomcoerce('solid', {circ2 point2(0,0)});
fem.mesh = meshinit(fem);
fem.pnt.weak = 'u_test';
fem.pnt.ind = {3};
fem.equ.c = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem,'u','triz','u')
```

FEM.ODE—GLOBAL VARIABLES AND EQUATIONS

Use the field `fem.ode` to introduce single named degrees of freedom, which are available as variables everywhere in the model, and to set equations for these variables. Despite its suggestive name, `fem.ode` need not specify an ordinary differential equation. The variables introduced are available everywhere for use in domain equations, algebraic equations, or ODEs. It is also possible to set equations for variables introduced somewhere else.

Store the dependent variables in the `fem.ode.dim` field, which is a cell array of strings. It is possible to provide initial values both for the variables (in the `fem.ode.init` field) and for their time derivatives (in the `fem.ode.dinit` field).

You can specify scalar equations either in `fem.ode.f` or in `fem.ode.weak`. The former, if present, is a cell array of strings with the same number of entries as `fem.ode.dim`. COMSOL Multiphysics sets each entry equal to zero and adds them to the global set of equations, using the test function of the variable at the corresponding position in the `dim` field. The `weak` field can contain an arbitrary number of entries in the weak form. Expressions used in the equations can reference global variables, constants, and integration coupling variables with global destination.

Note that the introduction of degrees of freedom is very much independent of the specification of equations. The equation fields can be left blank if you only need the

degrees of freedom in a domain equation. Conversely, you can use the `weak` field on its own to set equations for integration coupling variables with global destination.

For example, to solve Volterra's prey-predator equations, enter the following:

```
clear fem
fem.geom = geom0(zeros(0,1));
fem.mesh = meshinit(fem);
fem.const = {'a',1,'b',0.1,'c',0.1,'d',0.001};
fem.ode.dim = {'r','f'};
fem.ode.f = {'rt-r*(a-b*f)','ft-f*(d*r-c)'};
fem.ode.init = {'120','8'};
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',0:0.1:100);
postcrossplot(fem,0,1,'pointxdata','r','pointdata','f')
```

Discretization

Use the `assemble` function to compute the global matrices K, L, M, N, and D that appear in the discretized version of the finite element problem:

```
[K, L, M, N, D] = assemble(fem, 'u', u0);
```

where `u0` is the solution object for which the linearization is done. See "The Linear or Linearized Model" on page 366 in the *COMSOL Multiphysics User's Guide*, and see the *COMSOL Multiphysics Reference Guide* entry `femsol` for information about the solution object.

The following data structures control the discretization.

GPORDER—ORDER OF QUADRATURE FORMULA

The integrals occurring in the assembly of the matrices are computed numerically using a quadrature formula. You specify the order of this quadrature formula in the fields `fem.equ.gporder`, `fem.bnd.gporder`, `fem.edg.gporder`, and `fem.pnt.gporder`.

The field `fem.equ.gporder` gives the order for integrals over subdomains. The field `fem.equ.gporder` can be a number or a cell array. In the first case, `fem.equ.gporder` applies to all subdomain groups. In the second case, `fem.equ.gporder{i}` applies to the *i*th subdomain group. The entry `fem.equ.gporder{i}` can be a number or a cell array of numbers. In the latter case, `fem.equ.gporder{i}{k}` applies to the *k*th equation in coefficient or general form, and the *k*th integrand `fem.equ.weak{i}{k}` in the weak formulation.

If the field `fem.gporder` is present, the interpretation of the syntax is different. Then `fem.gporder` can be a cell array with quadrature formula orders, and the numbers in `fem.equ.gporder` are indices into `fem.gporder`. When using mesh cases, the field `fem.gporder` is instead a structure with the following fields:

- `fem.gporder.default`, which is a cell array of quadrature formula orders used in mesh case 0. The numbers in `fem.equ.gporder` are indices into this cell array. These quadrature formula orders are also used for all mesh cases that do not occur in `fem.gporder.mind`.
- `fem.gporder.case`, which is a cell array of cell arrays of quadrature formula orders. `fem.gporder.case{i}` applies to the mesh cases `fem.gporder.mind{i}`. The numbers in `fem.equ.gporder` are indices into this cell array. The default is an empty cell array `fem.gporder.case`.
- `fem.gporder.mind`, which is a cell array of vectors of positive mesh case numbers. The default is `fem.gporder.mind = {1 2 ... n}`, where `n` is the length of `fem.gporder.case`.

Similarly, `fem.bnd.gporder`, `fem.edg.gporder`, and `fem.pnt.gporder` give the order for integrals over boundaries, edges, and vertices, respectively.

The default value of the `gporder` fields is twice the maximum order of the shape functions you are using.

CPORDER—DENSITY OF POINTWISE CONSTRAINTS

Pointwise constraints are enforced in the Lagrange points of a certain order. You specify this order in the fields `fem.equ.cporder`, `fem.bnd.cporder`, `fem.edg.cporder`, and `fem.pnt.cporder`. These fields have the same syntax as the `gporder` fields.

The field `fem.equ.cporder` gives the order for constraints on subdomains. The field `fem.equ.cporder` can be a number or a cell array. In the first case, `fem.equ.cporder` applies to all subdomain groups. In the second case, `fem.equ.cporder{i}` applies to the *i*th subdomain group. The entry `fem.equ.cporder{i}` can be a number or a cell array of numbers. In the latter case, `fem.equ.cporder{i}{k}` applies to the *k*th constraint on subdomain group `i`.

As for `gporder`, the interpretation of the syntax is different if the field `fem.cporder` is present; see the previous section on `gporder`.

Similarly, `fem.bnd.cporder`, `fem.edg.cporder`, and `fem.pnt.cporder` give the order for constraints on boundaries, edges, and vertices, respectively. The default value

of the cporder fields is equal to the maximum of the orders of the shape functions you are using.

If you use Hermite or Argyris shape functions and the boundary is curved, a locking phenomenon can occur due to the way COMSOL Multiphysics implements constraints. That is, if you choose cporder to be the same as the element order, then the derivative degrees of freedom become over-constrained at the mesh vertices on the boundary. To prevent this, use a cporder that is the element order minus 1.

Initial Values

For a time-dependent problem you must specify initial values for the variables. Also, for a nonlinear problem it helps to provide the solver with a good starting guess. Initial values are specified in the fields fem.equ.init, fem.bnd.init, fem.edg.init, and fem.pnt.init. There are also other ways to specify initial values (see asseminit in the *COMSOL Multiphysics Reference Guide*). If you do not specify any initial values they are assumed to be 0.

Initial values on subdomains are given in the field fem.equ.init. This field has the same syntax as fem.equ.f. That is, fem.equ.init is a numeric scalar, an expression, or a cell array. In the first two cases, this field defines the initial value for all dim variables on all subdomain groups. In the third case, fem.equ.init{i} applies to the *i*th subdomain group (but if the cell array fem.equ.init has length 1, its single entry applies to all subdomain groups). The entry fem.equ.init{i} can be a numeric scalar, an expression, or a cell array of scalars or expressions. In the first two cases, fem.equ.init{i} is the initial value for all dim variables. In the third case, the components of the cell array fem.equ.init{i} correspond to the different dim variables.

In a similar way, fem.equ.dinit defines the initial value for the time derivative when solving wave-equation problems.

The definitions given in fem.equ.init apply only where they have meaning, that is, only where the corresponding dim variable is defined. Similarly, you can provide initial conditions on boundaries, edges, and vertices in fem.bnd.init, fem.edg.init, and fem.pnt.init, although it is normally not necessary. If a variable is defined only on a boundary (for instance, a Lagrange multiplier in a weak constraint), then it is useful to define initial values for that variable using fem.bnd.init.

The solution object corresponding to an initial-value specification can be computed by the function asseminit (assuming that the extended mesh has been generated):

```
sol = asseminit(fem);
```

See the *COMSOL Multiphysics Reference Guide* entry `femsol` for information about the solution object.

The Extended Mesh

Once you have completed the model specification and before you use one of COMSOL Multiphysics' solvers, you must issue the command

```
fem.xmesh = meshextend(fem);
```

The function `meshextend` takes the model specification and puts it in a form suitable for the solvers. The resulting object, `fem.xmesh`, is called the *extended mesh*. Thus, if you change a model in any way, you must run `meshextend` before you solve it again.

The function `meshextend` does the following:

- Converts the standard syntax to element syntax. The result is taken together with the element syntax you have specified.
- Determines a partition of the domains into *domain groups*. Within each domain group there are the same variables, the same equations, and the same constraints. Similarly, the mesh elements are partitioned into *mesh element groups*. Within each mesh element group, the mesh elements are of the same type and the same shape functions are used for the variables.
- Collects all the node points that are introduced when using higher-order elements.
- Collects all degrees of freedom—or DOFs, for short. A DOF is a pair consisting of a DOF name and a node point. Each DOF is given a number and corresponds to an entry in the solution object.
- Collects all defined variables and checks for name collisions.

If you prefer to use only the element syntax, you can turn off the conversion from standard syntax by

```
fem.xmesh = meshextend(fem,'standard','off');
```

Also, you can turn off generation of boundary coupled variables by

```
fem.xmesh = meshextend(fem,'cplbndsh','off','cplbndeq','off');
```

Find out about additional property/value pairs that you can specify by looking up the entry for `meshextend` in the *COMSOL Multiphysics Reference Guide* or the function's command-line help entry.

SOLUTION FORM

You can generate the extended mesh using a form of the equation that is different from `fem.form`. This is called the *solution form*, and you specify it in the field `fem.solform`. The function `meshextend` reads this field and transforms the equations to this form before generating the extended mesh. Possible values for `fem.solform` are `coefficient`, `general`, and `weak` (the default).

Multiple Geometries

Up to now the discussion has covered how to set up an FEM structure for a model with a single geometry. If a model has several geometries, you must use an *extended FEM structure* called `xfem`. This extended FEM structure has a field `xfem.fem`, which is a cell array of ordinary FEM structures, one for each geometry. These FEM structures can contain all the fields described earlier except `const`, `elem`, `eleminit`, `functions`, `globalexpr`, `lib`, `sol`, `solform`, `version`, and `xmesh`. Instead, you add these field to `xfem`. The field `descr` can occur both at both the `xfem` and `fem` level.

As a simple example of the use of two geometries, first define the single geometry FEM structures `fem1` and `fem2`:

```
clear fem1 fem2 xfem
fem1.geom = solid1([0 1]);
fem1.mesh = meshinit(fem1);
fem1.equ.c = 1;
fem1.equ.f = 1;
fem1.bnd.h = 1;

fem2.geom = rect2;
fem2.mesh = meshinit(fem2);
fem2.equ.c = 1;
fem2.equ.f = 1;
fem2.bnd.h = 1;
```

This code specifies two uncoupled Poisson's equations with homogeneous Dirichlet conditions on the unit interval and the unit square. The dependent variable `u` has linear elements by default. Now define the extended FEM structure, extend the mesh, solve, and plot:

```
xfem.fem = {fem1 fem2};
xfem.xmesh = meshextend(xfem);
xfem.sol = femlin(xfem);
postplot(xfem,'geomnum',1,'liny','u');
figure
postplot(xfem,'geomnum',2,'tridata','u','tribar','on');
```

The solution object `xfem.sol.u` contains values of the DOFs for both geometries.

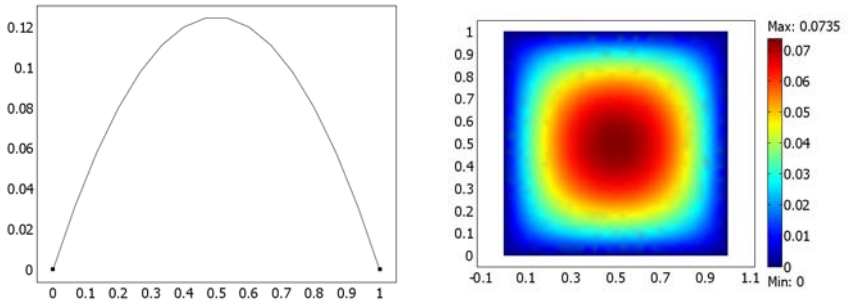


Figure 3-1: Solutions to Poisson's equations on the unit interval (left) and the unit square.

Application Structures

The COMSOL Multiphysics *application modes* contain predefined equations for a variety of physics applications and thus offer a convenient alternative way of setting up PDE problems; instead of specifying the equations explicitly, you just enter values for the relevant physical parameters. The standard application modes are described in the *COMSOL Multiphysics Modeling Guide*, while the additional application modes provided by the various add-on modules are presented in their respective user's guides.

In order to benefit from the convenience of the predefined application modes when modeling at the command line, use *application structures*. You can have several application structures at the same time, describing different physics phenomena in a single multiphysics model. Once you have created the application structures, the `multiphysics` function translates them to equations and boundary conditions in the FEM structure.

Store the application structures in the `fem.appl` field in the FEM structure. This field can be a cell array of application structures or just a single application structure. These structures have many fields in common with the FEM structure, as is evident from this table:

TABLE 3-10: APPLICATION STRUCTURE FIELDS

FIELD	INTERPRETATION
`appl.assignsuffix`	Application mode variable name suffix
`appl.assign`	Application mode variable name assignments
`appl.bnd`	Boundary coefficients/application data

TABLE 3-10: APPLICATION STRUCTURE FIELDS

FIELD	INTERPRETATION
`appl.border`	Assembly on interior boundaries
`appl.dim`	Cell array with names of the dependent variable names
`appl.edg`	Edge coefficients/application data
`appl.equ`	Subdomain coefficients/application data
`appl.mode`	Application mode
`appl.name`	Application mode name
`appl.pnt`	Point coefficients/application data
`appl.prop`	Application mode properties
`appl.shape`	Shape functions (a cell array of shape function objects or strings)
`appl.sshape`	Preferred element geometry order (an integer)
`appl.var`	Application-mode-specific variables

THE MULTIPHYSICS FUNCTION

The multiphysics function looks at the application (`fem.appl`) structure within the FEM structure and updates the FEM structure. See the entry "multiphysics" in the Command Reference for details and information on property/value pairs. The FEM structure fields `diff`, `outform`, `outsshape`, `rules`, and `simplify` can be used instead of property/value pairs. The FEM structure's fields affected by the application structures through the action of the `multiphysics` function are `dim`, `form`, `equ`, `bnd`, `edg`, `pnt`, `var`, `elemmph`, `eleminitmph`, `shape`, `sshape`, and `border`.

DEFINING THE APPLICATION MODE

You define which application mode the application structure refers to in the `mode` field. This field is a structure with two fields: `class`, specifying the application mode class, and `type`, specifying if the model is Cartesian or axisymmetric. For example,

```
appl.mode.class = 'Electrostatics';
```

specifies the Electrostatics application mode. To make the model axisymmetric use

```
appl.mode.type = 'axi';
```

For Cartesian models you can omit the `type` field because this is the default type. Instead provide the class name directly in the `mode` field. For example,

```
appl.mode = 'Electrostatics';
```

specifies a Cartesian Electrostatics application mode.

The application modes are divided into the following categories:

- Acoustics
- Convection and Diffusion
- Electromagnetics
- Fluid Dynamics
- Heat Transfer
- Structural Mechanics
- PDE Modes
- Deformed Mesh

The optional modules for acoustics, chemical engineering, earth sciences, electromagnetics (AC/DC Module and RF Module), heat transfer, MEMS, and structural mechanics contain additional application modes from these application areas.

EXPANDING APPLICATION MODE DATA TO THE FEM STRUCTURE

Once you have defined one or several application structures, the data in them can be used to generate fields in the FEM structure. To do this, use the `multiphysics` function.

The names of the dependent variables, contained in the `fem.equ.dim` field, come from the corresponding field in the application structure. If no such field exists there, a default value for the application mode is used. If the model contains several application modes from which the software generates the FEM structure fields, the `dim` field in the FEM structure is a cell array with the dependent variables of all application modes.

For the field `border` there are important differences between the application structure and the FEM structure. The `border` field in the application structure is used for disabling boundary settings on interior boundaries when generating the boundary conditions in the FEM structure. The `multiphysics` function sets the field `fem.border` to `1`.

APPL.EQU—SUBDOMAIN SETTINGS

When using the PDE modes, the classical PDEs, or the weak modes, the subfields in `appl.equ` have direct correspondence to the same fields in the FEM structure. You can therefore define the `equ` field in one of these application modes in the same way as when working directly with the FEM structure (see "Equations and Constraints" on page 38). One advantage with working with application structures is that you can

generate the weak form from a problem defined in general form, and if you have defined an application in coefficient form you can generate an FEM structure using the general or weak forms. It is possible to do a similar thing using the FEM structure and the function `flform`, but then you lose the original problem formulation. For more information about `flform` see the *COMSOL Multiphysics Reference Guide*.

For physics modes there is a much clearer difference between the fields in the application structure and those in the FEM structure. The quantities available for each physics mode correspond to some physical property appearing in the equation defined by the application mode. For example, the default values of the fields in `appl.equ` defined by the Heat Transfer by Conduction application mode are

```
          k: {'400'}
    ktensor: {{2x2 cell}}
      ktype: {'iso'}
        Dts: {'1'}
        rho: {'8700'}
          C: {'385'}
          Q: {'0'}
     htrans: {'0'}
       Text: {'0'}
     Ctrans: {'0'}
  Tambtrans: {'0'}
```

For instance, `appl.equ.k` is the thermal conductivity, `appl.equ.rho` is the density, and `appl.equ.C` is the heat capacity. Now consider an example with two materials. To use other values than the default for the physical properties, enter them as follows:

```
appl.equ.k = {'300' '500'};
```

This means that `k` has the values 300 and 500 in subdomain groups 1 and 2, respectively.

In addition to the fields just discussed, the `appl.equ` structure can have the fields `shape`, `usage`, `gporder`, `cporder`, and `init` (see `multiphysics` for details). The fields `shape`, `gporder`, `cporder`, and `init` have the same syntax as the corresponding FEM structure fields.

APPL.BND—BOUNDARY SETTINGS

One difference between setting up boundary conditions in the application structures and the FEM structure is the use of different *boundary condition types*. This is true also for the PDE modes and weak form application modes. This means that, for example, if the boundary condition on a certain boundary is defined to be of Neumann

type, the software ignores the `h` and `r` coefficients in the application structure when generating fields in the FEM structure.

For PDE modes there are two boundary condition types available. For Dirichlet conditions the type is labeled `'dir'`, and for Neumann conditions it is `'neu'`. The following example sets a Neumann condition on boundary 1 and Dirichlet conditions on all other boundaries:

```
appl.mode = 'FlPDEC';
appl.bnd.r = {1 1};
appl.bnd.h = {1 1};
appl.bnd.type = {'neu' 'dir'};
appl.bnd.ind = [1 2 2 2];
```

For physics modes it is of the same importance to define the boundary condition type. For each type there can be a number of additional quantities that you also need set to specify the boundary condition completely.

The default values in the `appl.bnd` structure for the Heat Transfer by Conduction application mode are

```
        q0: {'0'}
         h: {'0'}
      Tinf: {'0'}
     Const: {'0'}
      Tamb: {'0'}
        T0: {'0'}
      type: {'q0'}
```

To change boundary conditions enter

```
appl.bnd.type = {'q0' 'T'};
appl.bnd.T0 = {'0' '2'};
```

The type `q0` corresponds to the thermal insulation boundary condition, and the type `T` sets the temperature to the value given by `appl.bnd.T0`. Thus, in this case you have thermal insulation on boundary group number 1 and the temperature $T = 2$ on boundary group number 2. You can replace the `'0'` in `appl.bnd.T0` by anything because when `appl.bnd.type = 'q0'`, COMSOL Multiphysics does not use `appl.bnd.T0`.

In addition to the fields just discussed, the `appl.bnd` structure can have the fields `shape`, `gporder`, and `cporder`. For certain application modes that do not have an `appl.equ` field, the `appl.bnd` structure can also contain the fields `shape` and `usage`. The fields `shape`, `gporder`, `cporder`, and `init` have the same syntax as the corresponding FEM structure fields (see `multiphysics` for details).

APPL.EDG AND APPL.PNT—EDGE AND POINT SETTINGS

In the same way as `appl.equ` and `appl.bnd` define the equation and boundary conditions, you can use `appl.edg` to define equations and conditions on the edges (3D) and `appl.pnt` on points (2D and 3D).

APPL.BND.PAIR, APPL.EDG.PAIR, AND APPL.PNT.PAIR—SETTINGS ON PAIRS

To define settings on pairs, use the fields `appl.bnd.pair`, `appl.edg.pair`, and `appl.pnt.pair`. These fields have the same fields as `appl.bnd`, `appl.edg`, and `appl.pnt` with the exception of the field `appl.bnd.ind` for defining domain groups. Instead there is a field `appl.bnd.pair.pair` for defining groups of pairs. `appl.bnd.pair.pair` is a cell array where each item is a name of a pair or a cell array with names of pairs. For example, if there are pairs with the names `pair1`, `pair2`, `pair3`, and `pair4`, then

```
appl.bnd.pair.pair = {'pair1' {'pair2' 'pair4'} 'pair3'};
appl.bnd.pair.q0 = {2 4 7};
```

specifies that the variable `q0` has the value 2 on `pair1`, the value 4 on `pair2` and `pair4`, and the value 7 on `pair3`.

APPLICATION-SPECIFIC VARIABLES

For many application modes the software generates additional variables. These are similar to the expression variables described in "Expression Variables" on page 26, but the names and definitions of these application-specific variables are predefined.

The *application mode variables*, defined on subdomains, boundaries, edges, and points are automatically generated and stored in `var` fields in the `equ`, `bnd`, `edg`, and `pnt` fields in the FEM structure. There are no corresponding `var` fields in the application structure, though, because you should not change the expression that is used when generating the variables. You can, however, modify the fields in the FEM structure, but we do not recommended doing so—instead use the expression variables in the `expr` fields.

For the *application scalar variables*, which are defined everywhere and stored in `appl.var`, you can change the value in the application structure. For example, the AC Power Electromagnetics application mode uses this feature to define a frequency `nu` in the `appl.var` field. You can specify it as

```
appl.var.nu = '50';
```

or as a cell array, that is,

```
appl.var = {'nu' '50'};
```

ASSIGNED VARIABLE NAMES

Although the internal names used in the application structure are predefined, you can *assign* another name for the variable to use in the FEM structure. This is important to avoid variable name conflicts. To change a name of an application mode variable or an application scalar variable, use the `assign` field. If you set up the assigned field name as

```
appl.assign = {'nu' 'freq'}
```

before calling the `multiphysics` function, the generated FEM structure has the following `var` field:

```
fem.var = {'freq' '50'}
```

There is also a field `assignsuffix`, which you can use to add a suffix to the name of all application mode variables. Variables appearing in `appl.assign` use the given assigned name, while the other variables get the suffix defined by `appl.assignsuffix` added to their names.

APPLICATION MODE PROPERTIES

Some application modes define properties to, for example, specify which type of analysis to perform. They are specified in the `prop` field. As an example, both the Magnetostatics and the AC Power Electromagnetics application modes are implemented using the same application mode, `FlPerpendicularCurrents`. This is because they define the same type of physics, one for static fields and one for time-harmonic fields. The application mode has an analysis property to specify which type of analysis to perform.

```
appl.mode = 'FlPerpendicularCurrents';
appl.prop.analysis = 'static';
```

gives magnetostatics, while

```
appl.mode = 'FlPerpendicularCurrents';
appl.prop.analysis = 'harmonic';
```

gives AC power electromagnetics.

Default Element Property

In the application structure you can directly specify the contents of the fields `shape`, `sshape`, `***.shape`, `***.gporder`, and `***.cporder`, which are then moved into the appropriate parts of the FEM structure. There is, however, a simpler way of defining these properties in the application structure. You can use the `elemdefault` property.

Setting this property generates default values of the `shape`, `sshape`, `gporder`, and `cporder` fields when these are missing in the application structure.

For the PDE coefficient application mode, the default shape function is the quadratic Lagrange element with suitable values for `sshape`, `gporder`, and `cporder`. To solve a problem instead with linear Lagrange elements, enter

```
appl.prop.elemdefault = 'Lag1';
```

The software then generates shape functions and other fields in the FEM structure in correspondence with the default values for linear Lagrange elements. Because this field is available only in the application structure, this is another advantage of using application modes for equation modeling.

EXAMPLE: NAVIER'S EQUATION REVISITED

For examples of how to use the physics modes, see the *COMSOL Multiphysics Reference Guide* entry on `multiphysics`. The following example uses the PDE Coefficient Form application mode.

On page 44 you find an example of solving Navier's equation on a rectangle using the `equ` and `bnd` fields of the FEM structure directly. The example uses the coefficient form as well as the general form, which requires that you redefine the PDE coefficients and boundary conditions.

If you use the application structure for setting up the problem, you can use `multiphysics` repeatedly for redefining the problem. First set the geometry and mesh in the FEM structure as in the previous example:

```
clear fem
fem.geom = rect2(0, 1, 0, 0.1);
fem.mesh = meshinit(fem);
```

In this case define the constants in the `fem.expr` field (although `fem.const` is more efficient).

```
fem.expr = {'E' 1e2 'nu' 0.3 'G' 'E/2/(1+nu)' ...
            'mu' '2*G*nu/(1-nu)' 'Fx' '0' 'Fy' '-1'};
```

To set up the application structure, you must specify the name of the application mode and the nature of the subdomain and boundary settings. In this case, use the PDE coefficient application mode for two variables.

```
clear appl
appl.mode = 'FlPDEC';
appl.dim = {'u' 'v' 'u_t' 'v_t'};
appl.equ.c = { { {'2*G+mu' 'G'} {0 'mu'; 'G' 0}...
```

```
                {'G' '2*G+mu'} } };
appl.equ.f = { {'Fx' 'Fy'} };
appl.bnd.h = { 0 1 };
appl.bnd.type = {'neu' 'dir'};
appl.bnd.ind = [1 1 1 2];
fem.appl = appl;
```

To generate the fields in the FEM structure enter

```
fem = multiphysics(fem);
```

The following commands solve the problem and visualize the solution:

```
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem, 'v')
```

The solution uses quadratic Lagrange elements by default. To solve the problem using linear Lagrange elements, make the following changes:

```
fem.appl.prop.elemdefault = 'Lag1';
fem = multiphysics(fem);
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem);
postsurf(fem, 'v')
```

The Structure of a Model M-file

You can save an entire modeling session as a sequence of COMSOL Multiphysics commands called a *Model M-file*. You can:

- Run a Model M-file directly from COMSOL Script or MATLAB, or load it into COMSOL Multiphysics
- Modify a Model M-file using COMSOL or MATLAB commands

If you want to open a modified Model M-file in COMSOL Multiphysics, it is important to follow the syntactic rules outlined in the following sections.

A Model M-file consists of several parts, each describing different aspects of the COMSOL Multiphysics model. The semantic and syntactic rules are outlined here. They need not be considered if you modify a Model M-file from the command line. This section describes how COMSOL Multiphysics generates a Model M-file.

The order of the commands recorded in the Model M-file is important and should be roughly the same as in this document. A Model M-file must start with the geometry section and end with the multiphysics, meshextend, initial value, solver, and postprocessing sections (in that order, but not all sections are necessary in all models). All of the in-between commands can occur in any reasonable order. The following sections contain the detailed ordering rules.

FLCLEAR COMMAND

An automatically generated Model M-file starts with the following command:

```
flclear fem
```

The `flclear` function clears variables when running M-files, and it is good practice to remove any existing FEM structure with the name `fem` before creating a new one for the model in the Model M-file.

Note: Do not use the `clear` function instead of `flclear`.

VERSION SECTION

The *version section* assigns a version to the FEM structure.

```
fem.version = version;
```

The version must be a legal COMSOL version structure. That is, it must contain the fields

```
version.name = 'COMSOL 3.4';
version.major = 0;
version.build = numeric;
```

where `numeric` must be an integer less than or equal to the number visible in the **About COMSOL** window that you open from the **Help** menu.

If the version field is missing, the software assumes the FEM structure was created using your current version.

SPACE DIMENSION SECTION

The *space dimension section* determines the names of the space coordinates (that is, the independent variables). For single-frame models, the corresponding field, `fem.sdim`, is either an array of strings, such as

```
fem.sdim = {'r' 'z'};
```

for a 2D axisymmetric geometry, or an integer equal to the number of space dimensions. For models with multiple frames, `fem.sdim` is an array of string arrays, for example

```
fem.sdim = {{'X' 'Y'} {'x' 'y'}};
```

You can omit this section.

DRAW SECTION

The *draw section* determines the geometry objects in the drawing. It is generated each time you leave Draw mode and change the geometry.

```
fem.draw=draw;
```

The geometry model, `draw`, is documented in the *COMSOL Multiphysics Reference Guide* entry `geomcsg`. See also the section "Working with a Geometry Model" on page 90.

You can omit this section.

Separate File for Geometry Objects in Model M-Files

Sometimes the geometry objects in the draw section of a Model M-file, which defines `fem.draw`, are not generated from commands but saved in a separate file. This happens in the following cases:

- When you have opened a Model M-file and afterwards used **Reset Model** in the **File** menu.
- When you have imported the geometry or the FEM structure from COMSOL Script or MATLAB.

The reason is that, in those cases, the information on how to generate the geometry from commands is lost.

When you save a Model M-file—such as under the name `mymodel.m`—COMSOL Multiphysics also saves a file `mymodel.mphm` with the geometry objects. In `mymodel.m` the variable `flbinaryfile` specifies the geometry file:

```
flbinaryfile = 'mymodel.mphm';
```

The geometry objects are defined by statements such as

```
g1 = flbinary('g1','draw',flbinaryfile);
```

The function `flbinary` loads the object, which is identified by the tags `g1` and `draw`, from `mymodel.mphm`.

GEOMETRY SECTION

The *geometry section* defines the model's analyzed geometry. COMSOL Multiphysics generates this section each time you leave Draw mode and change the geometry.

```
fem.geom = geomcsg(fem);
```

You can assign any valid analyzed geometry—such as a geometry object, a mesh (in 1D and 2D), a decomposed geometry matrix list from the PDE Toolbox (2D), or a Geometry M-file—to the `fem.geom` field. However, as explained in "Working With the Analyzed Geometry" in Chapter 4, it is recommended to use the function `geomobject` to convert the analyzed geometry to a geometry object (if it is not one already) before you store it in `fem.geom`.

This section must be included.

APPLICATION MODE SECTION

The *application mode section* determines the application modes, if any.

```
fem.appl{i} = appl;
```

When the software loads a Model M-file, this section determines a number of settings in the user interface. The `appl{i}` cell array item contains the following fields:

- `mode`—creates the application mode object
- `dim`—sets the dependent variable names
- `border`—sets the flags for assembly on interior boundaries
- `name`—sets the name
- `usage`—sets the subdomain usage flags
- `var`—determines the **Application Mode Variables** dialog box settings
- `assign`—the assigned names
- `assignsuffix`—application mode variable name suffix
- `prop`—application mode properties
- `shape`—sets up shape functions and preferred shape order
- `sshape`—sets up shape functions and preferred shape order
- `equ`—determines the **Subdomain Settings** dialog box settings
- `bnd`—determines the **Boundary Settings** dialog box settings
- `edg`—determines the **Edge Settings** dialog box settings (3D only)
- `pnt`—determines the **Point Settings** dialog box settings (2D and 3D).

You can omit this section.

MESH SECTION

The *mesh section* contains a mesh object.

```
fem.mesh = mesh;
```

You can create the mesh using the mesh-generation commands described in Chapter 5, "Creating Meshes."

The first mesh section in a Model M-file must contain a `meshinit` statement:

```
fem.mesh = meshinit(fem,...);
```

When you open a Model M-file in the COMSOL Multiphysics GUI, the software uses the property/value pairs in the `meshinit` command to update the settings in the **Free Mesh Parameters** dialog box.

You can omit the mesh section if there is no solver section. At least one mesh section must appear before a solver section.

See Chapter 5 of this volume or the *COMSOL Multiphysics Reference Guide* entries on `meshinit`, `meshrefine`, and `meshsmooth` for more information.

INTERIOR BOUNDARIES SECTION

The *interior boundaries section* determines the interior boundaries settings.

```
fem.border = border;
```

You can omit this section, which is the same as specifying `fem.border='off'`.

FORM SECTION

The *form section* determines the equation system form. If the Model M-file calls `multiphysics`, the `outform` field determines the equation system form that the `multiphysics` call returns.

```
fem.outform = outform;
```

The `multiphysics` function returns an FEM structure on the desired equation system form, and the resulting form is stored in `fem.form`.

You can omit this section, which is equivalent to setting `fem.form='coefficient'`. The form section must be compatible with the application mode section. If any application mode form is in general form, you must have `fem.form='general'`.

The form that the software uses when solving the equations can be different from the form that you use to enter them. The function `meshextend` uses the field `fem.solform` to determine the solution form of the equations. You can omit `fem.solform`, which is equivalent to specifying `fem.solform='weak'`.

See the section "The following table lists the available PDE modes with their default application mode names:" on page 238 in the *COMSOL Multiphysics Modeling Guide* for more information about the form of a PDE.

SIMPLIFY SECTION

The *simplify section* determines if the software should simplify expressions or not. The `multiphysics` call uses this setting when combining the composite system of equations.

```
fem.simplify = simplify;
```

You can omit this section, which is equivalent to specifying `fem.simplify='on'`.

See the *COMSOL Multiphysics Reference Guide* entry on `multiphysics` for more information about the simplification of expressions.

FUNCTION DEFINITION SECTION

The *function section* defines functions used when deriving the composite system.

```
fem.functions = functions;
```

You can omit this section if the model does not include any functions.

MATERIALS LIBRARY SECTION

The *materials library section* defines material parameters, coordinate systems, and cross sections used in the model.

```
fem.lib = lib;
```

You can omit this section if the model does not use any properties from a materials library.

GEOMETRY SHAPE ORDER SECTION

The *geometry shape order* section defines the order of geometry approximation, `sshape`. The `outsshape` field determines `sshape`.

```
fem.outsshape = outsshape;
```

If you omit this section, the software computes a default value based on the shape functions in the model. The resulting geometry shape order is stored in the `sshape` field.

CONSTANTS SECTION

The *constants section* determines names of constants and their values.

```
fem.const = const;
```

You can omit this section if the model does not include any constants.

APPLICATION SCALAR VARIABLES SECTION

The *application scalar variables section* determines variables that the application modes define. There is a separate section for each application mode.

```
fem.appl{i}.var = var;
```

You can omit this section.

EXPRESSION SECTION

The *expression section* contains expression variables and their defining expressions.

```
fem.expr = expr;
```

These expressions are defined everywhere on the geometry. You define domain-based expressions in the point settings, edge settings, boundary conditions, and PDE coefficient sections using the subfield `expr`.

You can omit this section.

GLOBAL EXPRESSION SECTION

The *global expression section* contains expression variables that apply on all geometries of multiple-geometry models and their defining expressions.

```
fem.globalexpr = globalexpr;
```

You can omit this section.

SHAPE FUNCTION SECTION

The *shape function section* determines the shape functions used when solving the PDE problem. This is defined for each application mode separately:

```
fem.appl{i}.shape = shape;
```

POINT SETTINGS SECTION

The *point settings section* determines the point settings used by the solvers to solve the PDE problem. These settings are made for each application mode separately:

```
fem.appl{i}.pnt = pnt;
```

For 1D models there is no point section.

EDGE SETTINGS SECTION

The *edge settings section* determines the edge settings used by the solvers to solve the PDE problem. These settings are made for each application mode separately:

```
fem.appl{i}.edg = edg;
```

The edge section exists only for 3D models.

BOUNDARY CONDITION SECTION

The *boundary condition section* determines the boundary conditions in the model. These settings are made for each application mode separately:

```
fem.appl{i}.bnd = bnd;
```

PDE COEFFICIENT SECTION

The *PDE coefficient section* determines the PDE coefficients or material properties in the model. These settings are made for each application mode separately:

```
fem.appl{i}.equ = equ;
```

When you have loaded a Model M-file, the settings made in this section determine the contents of the **Subdomain Settings** dialog box for the equation system.

COUPLING VARIABLE ELEMENTS SECTION

The *coupling variable elements section* defines coupling variable elements. You define such variables in the dialog boxes for coupling variables.

```
fem.elemcpl = elemcpl;
```

MULTIPHYSICS SECTION

The *multiphysics section* combines the application modes in `fem.appl` to a composite PDE system.

```
fem = multiphysics(fem,...);
```

MESHEXTEND SECTION

The *meshextend section* extends the finite element mesh to the desired element types.

```
fem.xmesh = meshextend(fem,...);
```

This section must be included.

See the *COMSOL Multiphysics Reference Guide* entry on `meshextend` for details.

INITIAL VALUE SECTION

The *initial value section* computes a solution object corresponding to the initial conditions in the model.

```
init = asseminit(fem,...);
```

In some cases when, for example, a mesh change, a geometry change, or a change in the included dependent variables has occurred, it is necessary to map the solution object onto the new extended mesh before computing the initial solution object.

You can omit this section.

See the *COMSOL Multiphysics Reference Guide* entry `asseminit` for more information on initial values.

SOLVER SECTION

The *solver section* of the Model M-file should contain a call to one of the solvers to produce a solution structure.

```
fem.sol = sol;
```

Solvers that also modify other parts of the FEM structure, such as the adaptive solver, return the full FEM structure:

```
fem = adaption(fem,...);
```

The software uses the property/value pairs in the solver commands to restore the settings in the user interface when you load a Model M-file.

If there is a postprocessing section, there must be a solver section.

Immediately following the solver call, there is an assignment of the type

```
fem0 = fem;
```

This copy of the FEM structure contains the previous solution, which is useful for computing the initial value, for example, when clicking the **Restart** button. You can omit this statement unless following commands refer to the variable `fem0`.

POSTPROCESSING SECTION

The *postprocessing section* reproduces all the plot types available in COMSOL Multiphysics.

Standard Plots

The `postplot` command corresponds to plots available in the **Plot Parameters** dialog box (except animations).

```
postplot(fem,...
         'tridata',expr,...
         'contdata',expr,...
         ...)
```

The software uses the property/value pairs in the `postplot` command to restore the settings in the user interface when you open the Model M-file.

You can omit this section. At least one solver section must precede the postprocessing section.

Model M-files for Models with Multiple Geometries

The structure of a Model M-file for an extended multiphysics model, using multiple geometries, is slightly different from that of a single-geometry model. In the multiple-geometry case the FEM structure has an additional level. To differentiate this structure as a data type from a single-geometry FEM structure, it is called an *extended FEM structure*, denoted `xfem`. The top level of the extended FEM structure contains an `fem`

field that is a cell array holding an ordinary FEM structure `xfem.fem{`*g*`}` for each geometry *g*. In addition, there are fields describing global properties for the model.

More specifically, in addition to the `fem` field, the extended FEM structure can have the fields `const`, `descr`, `elem`, `eleminit`, `functions`, `globalexpr`, `lib`, `sol`, `solform`, `version`, and `xmesh`. The ordinary FEM structures in `xfem.fem` do not contain the fields `const`, `elem`, `eleminit`, `functions`, `globalexpr`, `lib`, `sol`, `solform`, `version`, and `xmesh`.

For a multiple-geometry model it is the `xfem` structure that you pass to `meshextend`, `asseminit`, the solvers, and all postprocessing functions. Because only one geometry is active at a time, you must activate a geometry prior to a section with geometry-specific commands. Do this by making the assignment

```
fem = xfem.fem{g};
```

where *g* is the geometry number. Any subsequent geometry-specific commands apply only to geometry number *g*. Before activating another geometry or using the `xfem` structure, you must store the changes to geometry *g* in the `xfem` structure again:

```
xfem.fem{g} = fem;
```

4

Geometry Objects and Images

This chapter describes the basic geometry objects that form the geometry of a PDE problem in COMSOL Multiphysics and the scripting tools for creating such geometries, and then ends by illustrating their application in a few examples.

Geometry Objects

This section contains brief definitions, descriptions, and explanations of some basic geometry-related concepts which are central for finite element modeling and, in particular, its implementation in COMSOL Multiphysics.

The Geometry of a PDE Problem

A PDE problem geometry is built out of a set of bounded *domains*—or *connected manifolds*—of different dimensions: volumes, surfaces (or *faces*), curves (or *edges*), and points (or *vertices*). In each number of space dimensions, the domains of the maximal dimension are called *subdomains*, while the domains of next-to-maximal dimension are referred to as *boundaries*.

In COMSOL Multiphysics, geometric domains are represented by *geometry objects*. For each space dimension, there is a base class of geometry objects containing the necessary information for the geometry modeling in that dimension: `geom0`, `geom1`, `geom2`, and `geom3`.

In 3D, there are *solid objects* (`solid3`), *face objects* (`face3`), *curve objects* (`curve3`), and *point objects* (`point3`), all of which are subclasses to the `geom3` class. These, in turn, have subclasses representing primitive geometry objects such as blocks, cylinders, spheres, and ellipsoids.

In 2D, there is one subclass to `geom2` for solid objects (`solid2`), one subclass for *rational Bézier curve* objects (`curve2`), and one subclass for point objects (`point2`). You can also create geometry objects representing the primitive objects: squares, rectangles, circles, and ellipses.

A set of similar geometry classes exists for 1D geometries; the base class `geom1` has the subclasses `solid1` and `point1` for intervals and points, respectively.

The crucial information contained in the geometry objects is the description of the boundaries together with information on the up and down subdomain of each boundary. The main difference between 1D, 2D, and 3D is the amount of information necessary to properly describe the boundaries; the higher the space dimension, the more information is necessary.

- In 1D, the boundaries are vertices (points) defined by one coordinate value. To each vertex there is an associated up and down subdomain.

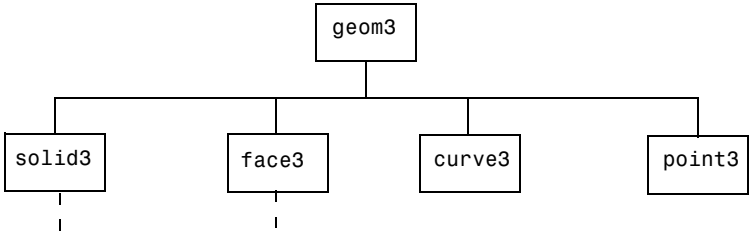
- In 2D, the subdomains are bounded by rational Bézier curves of degree one, two, and three. These curves, or edge segments, are defined by control points and weights, as described in the subsection "Creating a 2D Geometry Using Boundary Modeling" on page 88, and for each curve the up and down subdomain is given. There can also be isolated vertices and curves, unrelated to the subdomains.
- In 3D, the subdomains are bounded by *faces* or *trimmed surfaces*. A face is a 2D geometric entity bounded by curves. To properly describe a face, it is necessary to first of all have information about the underlying surface, that is, control points and weights of the rational Bézier patch. Furthermore, the bounding curves must be defined. These can either be rational Bézier curves with descriptions, as in the 2D case, or curves that are intersections between several surfaces. These can only be described in terms of the associated rational Bézier patches of the intersection. The up and down subdomain numbering of faces works analogously to the 1D and 2D cases. If a face has the same subdomain on the up and down side, the face is referred to as isolated. There can also be isolated vertices and curves. An isolated vertex can either lie on a face or inside a subdomain.

The Geometry Object Hierarchy

The base class of each dimension contains the properties described in the previous subsection. For each base class there are a number of subclasses with specific properties describing common geometric objects such as rectangles, cylinders, and right-angled blocks.

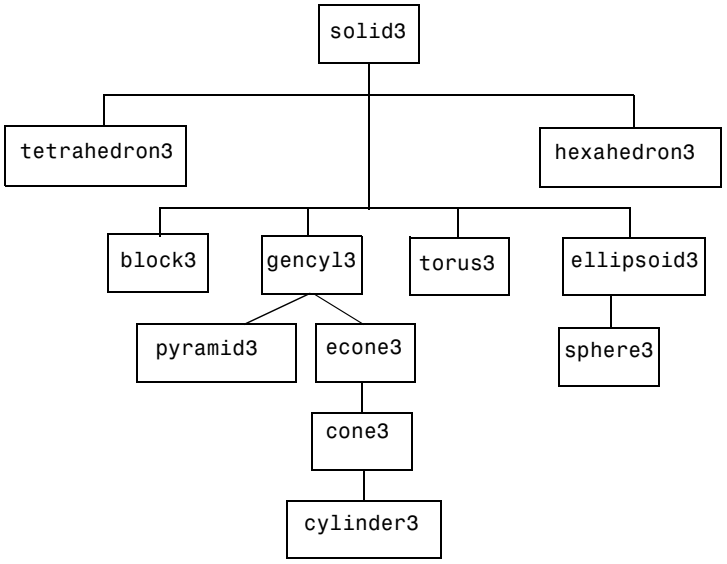
3D

The following is the hierarchical inheritance structure for the 3D geometry classes:



The four subclasses to the `geom3` class describe geometric entities of dimension 0, 1, 2, and 3. The `solid3` and `face3` classes have similar branches of subclasses that

represent primitive objects of surface or solid types. The following figure shows the inheritance structure for the 3D solid object branch:



An identical structure exists for the face object branch, where the class names are `tetrahedron2`, `block2`, and so on. For more information on the primitive objects, see the corresponding entries in the *COMSOL Multiphysics Reference Guide*.

You can also create 3D objects from 2D objects by extrusion or revolution. For example, the instructions

```
e3 = extrude(circ2,'displ',[-1;1],'distance',2);
figure, geomplot(e3);
```

create and plot a tilted cylinder, while the code lines

```
r3 = revolve(circ2(0.1,'pos',[1 0]),...
    'wrkpln',geomgetwrkpln('quick','xy'),...
    'revaxis',[0 0;0 -1],...
    'angles',[-pi/4,3*pi/4]);
figure, geomplot(r3);
```

produce a plot of a half torus; see Figure 4-1 below. A comprehensive list of functions for creating and manipulating geometry objects is given in the section "Geometry

Functions” on page 82.

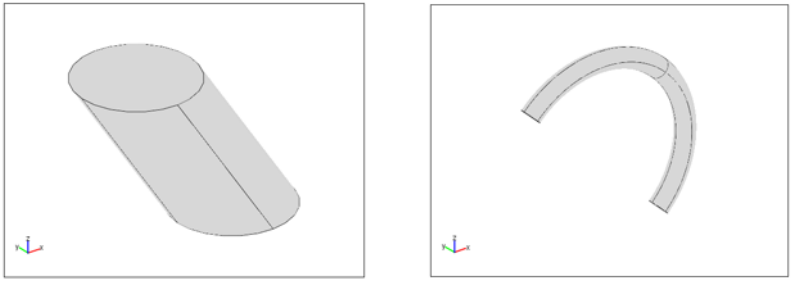
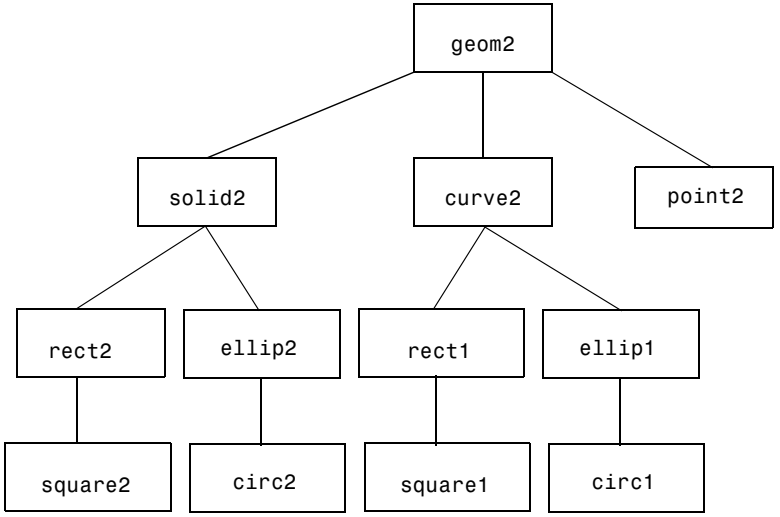


Figure 4-1: 3D geometry objects created from 2D objects.

2D

Below is the hierarchical inheritance structure of the 2D geometry classes:



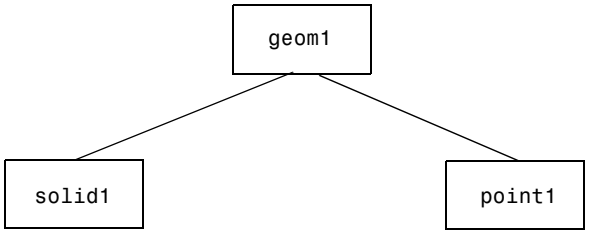
The base class `geom2` has three subclasses corresponding to solid objects, curve objects, and point objects. Curve and solid subclasses describe the primitive objects rectangle and ellipse.

An object of the class `solid2` consists of a boundary and an interior part as well as optional internal borders and subdomains. An object of the class `curve2` consists only

of a boundary, which can either be open or closed. An object of the class `curve2` can be coerced to a nonempty solid object of the class `solid2` if there is a closed boundary present in the object. A solid object or a curve object can always be coerced into a point object.

1D

Finally, the hierarchical inheritance structure of the 1D geometry classes is as follows:



The 1D geometry objects are either intervals (solid objects) or points. A point object can be coerced to a nonempty solid object if it contains more than one point.

Geometry Classes

3D

The 3D geometry classes are listed below.

TABLE 4-1: GEOMETRY CLASSES AND THEIR SUPERCLASSES, 3D.

FUNCTION	DESCRIPTION	SUPERCLASS
`block2`	Rectangular block face object	`face3`
`block3`	Rectangular block solid object	`solid3`
`cone2`	Cone face object	`econe2`
`cone3`	Cone solid object	`econe3`
`curve3`	3D curve object	`geom3`
`cylinder2`	Cylinder face object	`cone2`
`cylinder3`	Cylinder solid object	`cone3`
`econe2`	Eccentric cone face object	`gencyl2`
`econe3`	Eccentric cone solid object	`gencyl3`
`ellipsoid2`	Ellipsoid face object	`face3`

TABLE 4-1: GEOMETRY CLASSES AND THEIR SUPERCLASSES, 3D.

FUNCTION	DESCRIPTION	SUPERCLASS
ellipsoid3	Ellipsoid solid object	solid3
face3	3D face object	geom3
geom	Geometry object	
geom3	3D geometry object	geom
helix1	Helix curve object	curve3
helix2	Helix face object	face3
helix3	Helix solid object	solid3
hexahedron2	Hexahedron face object	face3
hexahedron3	Hexahedron solid object	solid3
point3	3D point object	geom3
pyramid2	Rectangular pyramid face object	gencyl2
pyramid3	Rectangular pyramid solid object	gencyl3
gencyl3	Straight homogeneous generalized cylinder solid object with linearly varying section	solid3
gencyl2	Straight homogeneous generalized cylinder face object with linearly varying section	face3
solid3	3D solid object	geom3
sphere2	Sphere face object	face3
sphere3	Sphere solid object	solid3
tetrahedron2	Tetrahedron face object	face3
tetrahedron3	Tetrahedron solid object	solid3
torus2	Torus face object	face3
torus3	Torus solid object	solid3

2D

The 2D geometry classes are listed below.

TABLE 4-2: GEOMETRY CLASSES AND THEIR SUPERCLASSES, 2D.

FUNCTION	DESCRIPTION	SUPERCLASS
circ1	Circle curve object	ellip1
circ2	Circle solid object	ellip2
curve2	2D rational Bézier curve object	geom2

TABLE 4-2: GEOMETRY CLASSES AND THEIR SUPERCLASSES, 2D.

FUNCTION	DESCRIPTION	SUPERCLASS
ellip1	Ellipse curve object	curve2
ellip2	Ellipse solid object	solid2
geom	Geometry object	
geom2	2D geometry object	geom
point2	2D point object	geom2
rect1	Rectangle curve object	curve2
rect2	Rectangle solid object	solid2
solid2	2D solid object	geom2
square1	Square curve object	rect1
square2	Square solid object	rect2

1D

The 1D geometry classes are listed below.

TABLE 4-3: GEOMETRY CLASSES AND THEIR SUPERCLASSES, 1D.

FUNCTION	DESCRIPTION	SUPERCLASS
geom	Geometry object	
geom1	1D geometry object	geom
point1	1D point object	geom1
solid1	1D solid object	geom1

Geometry Functions

The most important *geometry methods* and *functions* acting on the classes listed in the previous subsection are shown in the following table.

TABLE 4-4: GEOMETRY METHODS AND FUNCTIONS

METHOD	1D	2D	3D	DESCRIPTION
arc1		√		Create elliptical or circular arc
arc2		√		Create elliptical or circular solid sector
chamfer		√		Create flattened corners in geometry object
elevate		√		Elevate degrees of geometry object Bézier curves
embed			√	Embed a 2D object in a plane

TABLE 4-4: GEOMETRY METHODS AND FUNCTIONS

METHOD	1D	2D	3D	DESCRIPTION
`extrude`			√	Create an extruded 3D object
`fillet`		√		Create circular rounded corners in geometry object
`flim2curve`		√		Create curve object from image data
`geomanalyze`	√	√	√	Analyze geometry objects in draw structure
`geomarrayr`	√	√	√	Create rectangular array of geometry objects
`geomcoerce`	√	√	√	Coerce geometry objects
`geomcomp`	√	√	√	Analyze (compose) geometry objects
`geomcsg`	√	√	√	Combine any number of solid objects, face objects, curve objects, and point objects and return an analyzed geometry
`geomdel`	√	√	√	Delete interior boundaries
`geomexport`	√	√	√	Export geometry object to file
`geomfile`	√	√		Geometry M-file
`geomgetwrkpln`			√	Retrieve work plane information
`geomimport`	√	√	√	Import geometry object from file
`geominfo`	√	√	√	Retrieve geometry information
`geomplot`	√	√	√	Plot a geometry object
`geomspline`		√		Spline interpolation
`geomsurf`			√	Surface interpolation
`geom1/get`	√			Get 1D geometry object information
`geom2/get`		√		Get 2D geometry object information
`geom3/get`			√	Get 3D geometry object information
`line1`		√		Create open curve polygon
`line2`		√		Create solid polygon
`loft`			√	Loft 2D geometry sections to a 3D geometry
`mirror`		√	√	Reflect geometry
`move`	√	√	√	Move geometry object
`poly1`		√		Create curve polygon

TABLE 4-4: GEOMETRY METHODS AND FUNCTIONS

METHOD	1D	2D	3D	DESCRIPTION
`poly2`		√		Create solid polygon
`revolve`			√	Create a revolved object
`rotate`		√	√	Rotate geometry object
`scale`		√		Scale geometry object
`split`		√	√	Split solid, curve, or point object

Geometry Modeling

The basics of geometry modeling appear in the chapter "Geometry Modeling and CAD Tools" on page 23 of the *COMSOL Multiphysics User's Guide*. This section adds information about geometry modeling on the command line.

Working with Geometry Objects

CREATING A 1D GEOMETRY

For more information about 1D geometry modeling in COMSOL Multiphysics, see "Creating a 1D Geometry Model" on page 37 of the *COMSOL Multiphysics User's Guide*. From the COMSOL Script prompt, you create a 1D geometry model using the constructors `solid1`, `point1`, and the function `geomcsg`.

For example, the command

```
s = solid1([0 1 2])
```

creates a 1D solid object consisting of vertices at $x = 0$, 1, and 2, and edges joining the vertices adjacent in the coordinate list. More generally, `solid1` takes an n_v-dimensional vector as its argument, where n_v is the number of vertices.

Typing

```
p = point1(0.5)
```

creates a 1D point object located at $x = 0.5$.

The command

```
g = geomcsg({s},{p})
```

merges the two geometry objects to create the 1D *analyzed geometry* `g`—that is, a geometry that can be used for further PDE modeling; the concept is explained further in the section "Working With the Analyzed Geometry" on page 93.

CREATING A 2D GEOMETRY USING SOLID MODELING

For more information about 2D geometry modeling in COMSOL Multiphysics, see the example "2D Solid Modeling Techniques" on page 42 of the *COMSOL Multiphysics User's Guide*. You can set up the geometry discussed in that example—

reproduced here in Figure 4-2—using Boolean operations from the COMSOL Script command line.

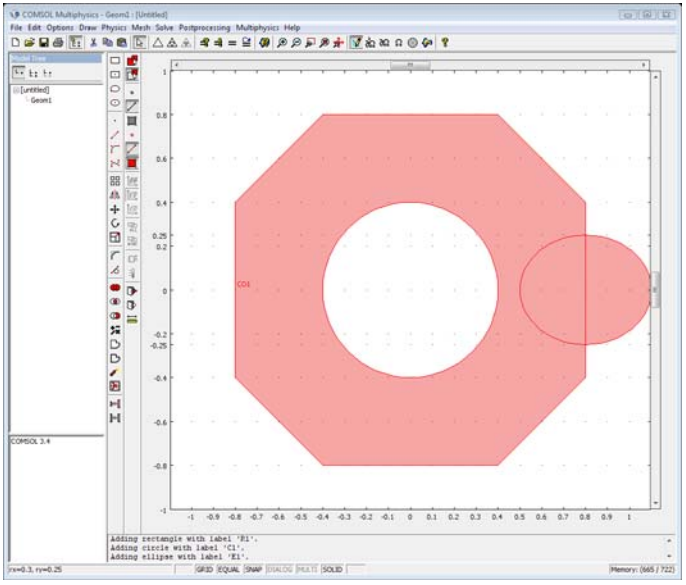


Figure 4-2: Sample 2D geometry.

Creating Composite Objects

First, create a rectangle with a hole in it by following these steps:

1. Create a solid rectangle:

   ```
   r1 = rect2(2,2,'pos',[-1 -1]);
   geomplot(r1)
   ```

   The function call `rect2(lx,ly,'pos',[x0 y0])` creates a solid rectangular object with the lower-left corner at (`x0`,`y0`) and sides of length `lx` and `ly`. The function `geomplot` plots the rectangle (which, of course, in this case is a square).

2. Create a circular hole with a radius of 0.5 centered at (0, 0):

   ```
   c1 = circ2(0.5,'pos',[0 0]);
   ```

   The function `circ2(r,'pos',[x0 y0])` creates a solid circular object (a disk) with radius `r` centered at (`x0`,`y0`) (the `'pos'` property defaults to `[0 0]` and could thus have been left out in this example). Use it to drill a hole in the first object either by forming the difference

```
s1 = r1-c1;
```

or by typing

```
s1 = geomcomp({r1,c1},'ns',{'r1','c1'},'sf','r1-c1');
```

`ns` is a cell array of variables name. `sf` represents a set formula with variable names from `ns`. The operators `+`, `*`, and `-` correspond to the set operations union, intersection and difference, respectively.

3 Finally, visualize the result:

```
geomplot(s1)
```

When defining `s1 = r1-c1`, you take advantage of the object-orientated capabilities in COMSOL Multiphysics. Both `r1` and `c1` are solid objects, and the operation is equivalent to calling the function `geomcomp` with the parameters shown earlier. The intersection operator (`*`) and the union operator (`+`) work similarly. The function `geomcomp` analyzes the specified geometric objects and returns the analyzed geometry.

Trimming Solids

To remove the corners from the solid object `s1` just created, follow the steps below:

1 Rotate `r1` 45 degrees (counterclockwise) around $(0, 0)$:

```
r2 = rotate(r1,45*pi/180,0,0);
```

2 Form the intersection between the rotated rectangle and `s1`:

```
s2 = s1*r2;
```

3 Visualize the result:

```
geomplot(s2)
```

The reason for inserting the factor $\pi/180$ in the second argument for `rotate` is that the function expects the angle of rotation in radians rather than degrees.

Adding Domains

To add a domain to a solid geometry object, use the union operator (+). Take the object, the solid `s2`, and attach an elliptical domain on its right side:

1 Create a solid ellipse centered at $(0, 0)$ and with semiaxes $(0.5, 0.25)$:

```
e = ellip2(0.5,0.25,'pos',[0 0],'rot',45*pi/180);
```

The final argument, `45*pi/180`, expresses the angle of rotation in radians of the semiaxes with respect to the coordinate axes.

2 Rotate the solid ellipse by –45 degrees to get an ellipse with semiaxes parallel to the coordinate axes:

```
e = rotate(e,-45*pi/180);
```

3 Translate the center of the ellipse from (0, 0) to (1, 0):

```
e = move(e,1,0);
```

The second and third parameters in the function call correspond to the *x* and *y* displacements, respectively.

4 Use the union operation to add the solid ellipse to the solid `s2`:

```
s3 = s2+e;
```

5 Visualize the result:

```
geomplot(s3)
```

CREATING A 2D GEOMETRY USING BOUNDARY MODELING

Use command-line boundary modeling to obtain the result from the example "2D Boundary Modeling" on page 46 of the *COMSOL Multiphysics User's Guide*.

1 Create the six boundary curve objects:

```
w = 1/sqrt(2);
c1 = curve2([-0.5 -1 -1],[-0.5 -0.5 0],[1 w 1]);
c2 = curve2([-1 -1 -0.5],[0 0.5 0.5],[1 w 1]);
c3 = curve2([-0.5 0.5],[0.5 0.5]);
c4 = curve2([0.5 1 1],[0.5 0.5 0],[1 w 1]);
c5 = curve2([1 1 0.5],[0 -0.5 -0.5],[1 w 1]);
c6 = curve2([0.5 -0.5],[-0.5 -0.5]);
```

The objects `c1`, `c2`, `c3`, `c4`, `c5`, and `c6` are all `curve2` objects. The vector `[1 w 1]` specifies the weights for a rational Bézier curve that is equivalent to a quarter-circle arc. The weights are adjusted to create elliptical or circular arcs. For more information on the available geometry objects, see "The Geometry Object Hierarchy" on page 77 and the individual geometry object entries in the *COMSOL Multiphysics Reference Guide*.

2 Create a solid object `s` of the object type `solid2`:

```
s = geomcoerce('solid',{c1,c2,c3,c4,c5,c6});
```

3 Visualize the result:

```
geomplot(s)
```

Create a `curve2` object `c` from the `solid2` object `s` using the `curve2` function:

```
c = curve2(s);
```

You can accomplish the same thing with a call to `geomcomp`:

```
c = geomcomp({c1,c2,c3,c4,c5,c6});
```

CREATING 3D GEOMETRIES USING SOLID MODELING

See the example "Solid Modeling Using Solid Objects and Boolean Operations" on page 67 of the *COMSOL Multiphysics User's Guide*. This section demonstrates the use of Boolean operations on solid objects from the COMSOL Script command line.

The examples show how to create composite solids starting from primitive solids. The programming language can be useful for creating geometry objects, for instance, when you need a sequence of commands to compute coordinates of geometry objects. To create the geometry, perform the following steps:

1 Create a solid rectangle with corners at (0, 0) and (1, 2):

```
r = rect2(1,2);
```

2 Use the fillet utility to take off its edges:

```
rf = fillet(r,'radii',0.125);
```

3 Create a circle object with radius 0.25:

```
c1 = circ2(0.25,'pos',[0.5 1]);
```

4 Form a triangle using three curve objects:

```
t1 = curve2([0.15 0.2],[0.75 0.75]);
t2 = curve2([0.2 0.2],[0.75 0.8]);
t3 = curve2([0.2 0.15],[0.8 0.75]);
```

5 Create a solid triangle by a call to `geomcoerce`:

```
t = geomcoerce('solid',{t1,t2,t3});
```

The function `geomcoerce` analyzes the specified geometric objects and coerces the resulting geometric object into the specified class.

6 Extrude the rectangle `rf` along the *z*-axis from 0 to 0.2 and extrude the circle `c1` from 0 to 0.2 along the same axis:

```
blk = extrude(rf,'distance',0.2);
hole = extrude(c1,'distance',0.2);
```

7 Create the chamfer by revolving the triangle object `t` 360 degrees about the center of the hole. To do so, define a work plane going through the center of the hole and then define an axis of rotation:

```
wrkpln = [[0;1;0],[0;1;1],[1;1;0]];
ax = [[0;0.5],[1;0]];
chamf = revolve(t,'angles',[0,2*pi],...
  'revaxis',ax,'wrkpln',wrkpln);
chamf = solid3(chamf);
```

8 Use the resulting 3D objects to create the composite solid object by subtracting the chamfer and the hole from the filleted block using the Boolean operations + and −:

```
plate = blk-(chamf+hole);
```

9 Plot the final geometry using `geomplot`:

```
geomplot(plate)
```

10 Use the **Headlight** or **Scene Light** toolbar buttons to turn on lighting in COMSOL Script.

IMPORTING GEOMETRY OBJECTS

You can import geometry objects from the COMSOL Script workspace into COMSOL Multiphysics by choosing **Import>Geometry Objects** from the **File** menu. The dialog box will list the geometry objects of the current space dimension. Select the geometry objects to import from the list and click **OK**.

EXPORTING GEOMETRY OBJECTS

You can export geometry objects from COMSOL Multiphysics to the COMSOL Script workspace by choosing **Export>Geometry Objects** from the **File** menu. Select the geometry objects to export from the list or in the draw area and click **OK**. The names of the geometry objects in COMSOL Multiphysics also serve as the names in the scripting workspace.

Working with a Geometry Model

A *geometry model* consists of a collection of geometry objects. Store the geometry model in the draw structure `fem.draw` using the fields `s`, `f`, `c`, and `p` for solids, faces, curves, and points, respectively. Each of these four fields, in turn, contains the following subfields:

- `objs`—an array of geometry objects of the appropriate type
- `name`—an array of strings containing the names of the objects in `objs`

The optional `tags` field of strings are used internally by COMSOL Multiphysics during the generation of model M-files.

The following example illustrates the construction of a simple geometry model, which is then stored in the structure `draw` and plotted:

```
g1=rect2;
g2=circ2(0.1,'pos',[0.5,0.5]);
g3=g1-g2;
```

```
clear s
s.objs={g3};
s.name={'CO1'};
draw=struct('s',s);
figure, geomplot(draw)
```

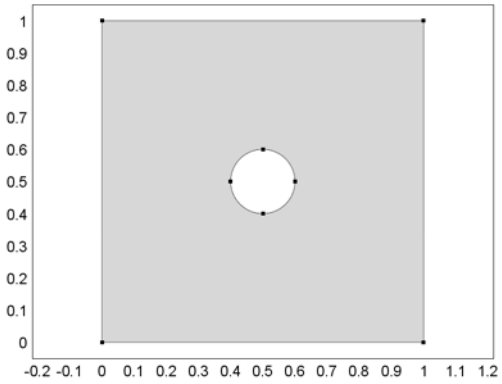


Figure 4-3: The geometry object contents of the geometry model draw.

GETTING GEOMETRY OBJECT PROPERTIES

Use the `get` function to get values for the geometry object properties. The available properties depend on the type of object. Type help `circ2/get` and help `geom2/get`, for example, to list the available properties for the `g2` and `g3` objects (the properties for a `geom2` object applies to the circle object `g2` as well). To list the number of vertices in the resulting geometry `g3` (8 vertices), type

```
get(g3,'nv')
```

To list the base point x-coordinate for the `circ2` object `g2` ($x = 0.5$), type

```
get(g2,'x')
```

The function geominfo provides additional information. To list the coordinates of the vertices in g3, for example, type

```
geominfo(g3,'out','mp')
```

IMPORTING A GEOMETRY MODEL

Import a geometry model, `draw`, from the scripting workspace to COMSOL Multiphysics by first typing

```
clear fem
fem.draw=draw;
```

and then choosing **Import>FEM Structure** on the **File** menu. Type the name `fem` in the dialog box.

EXPORTING A GEOMETRY MODEL

Export a geometry model from COMSOL Multiphysics to the command line by selecting **Export>FEM Structure as 'fem'** on the **File** menu. The geometry model becomes available in `fem.draw` in the scripting environment.

Importing and Exporting Geometries and CAD Models from File

With COMSOL Multiphysics, you can import and export geometries in a variety of file formats. See the *COMSOL Multiphysics Reference Guide* entries on `geomimport` and `geomexport` for details. Below is a short summary of the various file formats.

COMSOL MULTIPHYSICS FILES

A natural choice for storing geometries in 1D, 2D, and 3D is the native file format of COMSOL's geometry kernel (`.mphtxt`, `.mphbin`). Note that this file format is only used for geometry and mesh objects. It is not the same as a Model MPH-file (`.mph`).

2D CAD FORMATS

COMSOL Multiphysics supports import and export for the *DXF*® file format, the native format of the CAD system AutoCAD®. Furthermore, you can import files in the neutral GDS format.

3D CAD FORMATS

It is possible to import surface meshes in the STL and VRML formats. Moreover, COMSOL provides a CAD Import Module, which supports import of most 3D CAD file formats: Parasolid®, SAT®, STEP, IGES, CATIA® V4, CATIA® V5, Pro/ENGINEER®, Autodesk Inventor®, and VDA-FS. The *CAD Import Module User's Guide* covers these file formats in detail.

Working With the Analyzed Geometry

The geometry that COMSOL Multiphysics uses for the actual finite element analysis is called the *analyzed geometry* and is stored in the FEM structure field `fem.geom`. You generate the analyzed geometry from the geometry model using the command `geomcsg`. Alternatively, for geometries consisting entirely of objects from the same geometry class, you can use the command `geomcomp`. For further details, see the corresponding command-line help entries or consult the *COMSOL Multiphysics Reference Guide*.

A third way of generating an analyzed geometry is by using the function `geomanalyze`. In contrast to `geomcsg` and `geomcomp`, this function keeps track of the associative rules between the geometry objects. For this reason, `geomanalyze` is the most convenient option to choose when working with a model where the geometry changes, because physics settings are then automatically updated. In this behavior, `geomanalyze` mimics GUI-based modeling. Note, however, that M-files where `geomanalyze` is used cannot be opened in the COMSOL Multiphysics GUI. An example using `geomanalyze` is given in the section "Retrieving Geometry Information" on page 95.

In addition to a geometry object, you can store the name of a *Geometry M-file*, a mesh object, or a PDE Toolbox decomposed geometry matrix in `fem.geom`, but we no longer recommend this. Instead, to ensure efficient operation, convert Geometry M-file names, meshes, and PDE Toolbox decomposed geometry matrices to geometry objects by using the function `geomobject` before storing them in `fem.geom`.

For descriptions of Geometry M-files and mesh objects, see the *COMSOL Multiphysics Reference Guide* entries on `geomfile` and `femmesh`, respectively.

The function `geominfo` provides an information interface to the analyzed geometry. You can plot the analyzed geometry with the `geomplot` function.

IMPORTING THE ANALYZED GEOMETRY

To import an analyzed geometry, `geom`, into COMSOL Multiphysics, type

```
clear fem;
fem.geom=geom;
```

and then choose **Import>FEM Structure** from the **File** menu. Type the name `fem` in the dialog box.

EXPORTING THE ANALYZED GEOMETRY

Using the **Export>FEM Structure as 'fem'** option from the **File** menu, you can export the analyzed geometry to the command line. The analyzed geometry is then available in `fem.geom`.

Working with Assemblies

For complicated models, with geometries consisting of several components connected via well-defined interfaces, it can often be beneficial to mesh and define the physics on a component basis before joining the parts at a later modeling stage. In COMSOL Multiphysics you can do this using *assemblies*. This concept and the associated features and terminology are described in detail in the chapter "The Object Properties dialog box for a 3D solid object." on page 76 of the *COMSOL Multiphysics User's Guide*. For command-line modeling using COMSOL Script, the following functions are related to assembly objects:

- `geomgroup`—create an assembly from parts and get mate information
- `getpart`, `getparts`—extract parts from an assembly
- `geomanalyze`—create an assembly and update the entire FEM structure

The function geomgroup creates an assembly from a set of geometry objects.

```
[g,pairs] = geomgroup({rect2 move(rect2,[1 0.5])},...
                      'out',{'g' 'pairs'});
```

The example creates an assembly geometry object, `g`, and the paris associates with the assembly. See section "Pairs" on page 37 for more information how to use the pair information in assemblies. To access the parts in the assembly, use the geomparts function to extract the information.

```
[gg,stx,ctx,ptx] = getparts(g,'out',{'stx','ctx','ptx'});
```

This returns, in `gg`, a cellar array with geometry objects containing the parts in the assembly geometry object, and sparse association matrices that relates the entity numbering within `g` to the objects in `gg`.

The function `geomanalyze` works both for composite geometries and assemblies. Use `geomanalyze` for parameterizing the geometry. Find an example for how to apply geomanalyze in the context of parameterizing a geometry in section "Modeling with a Parametrized Geometry" on page 95.

See the *COMSOL Multiphysics Reference Guide* for more information about `geomgroup`, `getparts`, and `geomanalyze`.

Retrieving Geometry Information

Use the command `geominfo` to retrieve information about the analyzed geometry, such as the number of objects, object coordinates, and adjacency relations. You find complete information about command syntax and available properties in the *COMSOL Multiphysics Reference Guide* or by typing `help geominfo` at the command line. The following example illustrates the usage.

Start by creating the solid ellipse and retrieve its coordinates and curvatures:

```
e = ellip2(1,2);
[xy,c] = geominfo(e,'out',{'xx','crv'},'par',...
                  {ones(11,1)*[1 2 3 4],(0:0.1:1)'*ones(1,4)})
```

Plot the obtained coordinates of the ellipse

```
plot(xy(:,:,1),xy(:,:,2),'b-')
```

and plot the curvature of one of the edges via

```
figure,plot(0:0.1:1,c(:,1))
```

The command below retrieves the number of primitive objects from geometry file (`geomfile`) `cardg`:

```
no = geominfo('cardg','out',{'no'},'od',[0 1 2])
```

Modeling with a Parametrized Geometry

Associative geometry is a method that COMSOL Multiphysics uses in the graphical user interface to associate physics, mesh and other settings to the geometrical objects. Associative geometry means that simple geometric operations, such as dividing a subdomain into two, do not require you to re-enter for example physics settings for the new geometry parts. COMSOL figures this out.

You need to use the COMSOL Multiphysics function `geomanalyze` (instead of the default `geomcsg`) to coerce and analyze changed geometry objects to achieve the association at the command line.

The `geomanalyze` function mimics the behavior in the GUI when updating the changes made in the geometry. This means that you have a set of geometry objects in the COMSOL Script workspace from which you update the COMSOL Multiphysics model, with the geometry and the physics settings. geomanalyze keeps track of the associative rules between all geometry objects used in the construction of the model.

Note: M-files where `geomanalyze` is used cannot be opened in COMSOL Multiphysics GUI.

Start by creating a geometry. It consists of two adjacent rectangular domains. Study a moving circular-shaped source. First create the geometry

```
clear fem
draw{1} = rect2(.5,1,'pos',[0;0]);
draw{2} = rect2(.6,1,'pos',[0.5;0]);
draw{3} = circ2(.1,'pos',[0.2;0.5]);
fem=[];
fem = geomanalyze(fem,draw,'ns',{'R1','R2','C1'});
geomplot(fem)
```

Below you see what the geometry looks like. The parametric sweep moves the circle from the left subdomain over to the right, crossing the internal subdomain boundary.

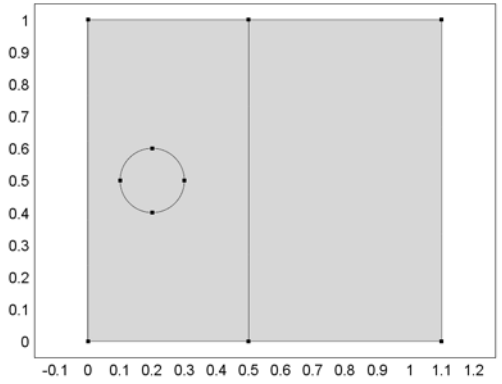


Figure 4-4: The original geometry.

Continue by creating the mesh and setting subdomain properties. the source is non-zero in the moving circular domain. Use the PDE application mode. Solve and plot the solution.

```
fem.mesh = meshinit(fem,'report','off');
fem.appl.mode = 'FlPDEC';
fem.appl.equ.f = {0 0 1};
fem.appl.equ.c = 1;
fem.appl.bnd.h = 1;
fem = multiphysics(fem);
```

```
fem.xmesh = meshextend(fem);
fem.sol = femlin(fem,'report','off');
postplot(fem,'tridata','u')
```

Now you introduce the loop that moves the circle from left to right. The loop replaces the circle in the draw structure, and then uses the geomanalyze function to update the FEM structure, including its PDE settings. Each p

```
nSteps = 5;              % number of steps in loop
dist = .75/nSteps;       % distance to move every step
for i = 1:nSteps
  c1 = drawgetobj(fem,'C1');                    % retrieve circle
  fem = drawsetobj(fem,'C1',move(c1, dist, 0));    % add new circle
  fem = geomanalyze(fem);
  fem.mesh = meshinit(fem,'report','off');
  fem = multiphysics(fem);
  fem.xmesh = meshextend(fem);
  fem.sol = femlin(fem,'report','off');
  figure, postplot(fem,'tridata','u','title',sprintf('i = %g',i))
end
```

This example continues in section "Postprocessing and Associativity" on page 167, showing how to use associativity information in postprocessing.

Images, Interpolation, and MRI Data

This section describes how to work with image files and MRI data as the basis for the geometry in COMSOL Multiphysics, while the following sections discuss creating geometry objects using spline and surface interpolation.

Working with Images and MRI Data

USING MRI DATA TO CREATE GEOMETRY OBJECTS

Magnetic resonance imaging (MRI) is an imaging technique used primarily in medical settings to produce high quality images of the inside of the human body. The more general name for the analytical technique is NMR (nuclear magnetic resonance spectroscopy). MRI data is typically represented as a sequence of 2D images.

Note: The functionality described in this section is only available when you run COMSOL Multiphysics with MATLAB.

MRI import consists of two steps. First you import the images and create the corresponding 2D geometry objects. After that, you can create 3D geometry objects using a lofting technique.

IMPORTING IMAGES

It is possible to import images on the formats JPEG, TIFF, BMP, and PNG and convert them into 2D or 3D geometry objects (on other platforms than 32-bit Windows, Linux, Solaris, and Macintosh only JPEG and PNG are available). From MATLAB, the import also supports images in the formats GIF, HDF, PCX, XWD, and ICO. To read graphic file format images use the `imread` function. This function converts images on graphic file format to a matrix format. Other useful functions are `imwrite` for saving images, `image` and `imagesc` for displaying images on matrix format. See the MATLAB documentation for more information on the MATLAB functions for image analysis.

Note: The Image Processing Toolbox for MATLAB contains additional functionality for advanced image processing.

In COMSOL Multiphysics you can create a 2D curve object from an image using the function `flim2curve` and create a 3D solid object from a sequence of images using the functions `flim2curve` and `loft`. For more information, see below in this section and the *COMSOL Multiphysics Reference Guide* entries `flim2curve` and `loft`.

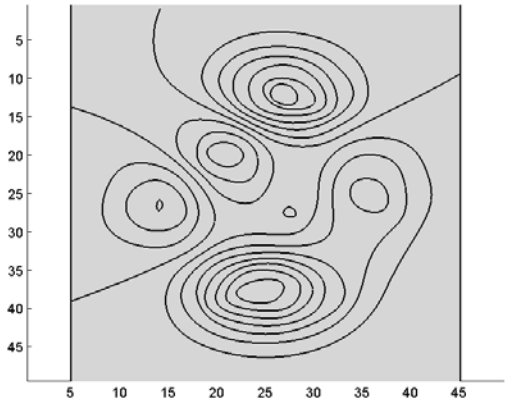
To create a 2D curve that describes the contour curves of an image, call the function `flim2curve` with appropriate property values. The function `flim2curve` utilizes the MATLAB function `imcontour` for detecting contours in images. The following example shows an image on matrix format, generated from MATLAB's sample function `peaks`, that the code converts to a 2D curve object. Instead of creating the image in MATLAB as in this example, you can import the image from a file using the `imread` command.

```
p = (peaks+7)*5;
figure
image(p)
v = axis;
c = flim2curve(p,{[],[5:5:75]});
g = geomcsg({rect2(5,45,0,50)},{c});
s = solid2(g);
```

Visualize the geometry object with the following commands.

```
figure
geomplot(s,'pointmode','off','sublabels','on');
axis(v)
```

```
axis ij
```



You can continue working on this model either from the MATLAB prompt or from the COMSOL Multiphysics user interface. To insert this geometry model into COMSOL Multiphysics: On the **File** menu, point to **Import**, and then click **Geometry Objects**. Once you have done this, you can use the **Split** and **Union** buttons for creating the geometry to be used in the simulation.

If the original image is noisy, the resulting geometry object may contain too many edge segments to be practical to work with. The function `flim2curve` provides basic functionality for filtering data by supporting the same input argument properties as `flmesh2spline` (see the *COMSOL Multiphysics Reference Guide* entries for these functions).

For more information on reducing noise in point data, see "Spline Interpolation for Creating Geometry Objects" on page 104.

If the processed image is large, the computation time for the MRI import functions can be long. To reduce the computation time it is often possible to subsample the image before processing it, that is, reduce the number of pixels in the image. The most simple subsampling method is to include every second pixel in the x and y direction, respectively. More advanced subsampling methods preprocess the image by low-pass filtering (also referred to as softening or blurring) before reducing the number of pixels. This often gives a higher quality of the subsampled image due to the fact that low-pass filtering reduces the amount of high-frequency components in the subsampled image (due to rasterization effects).

IMPORTING MRI DATA

When creating a 3D geometry object from a sequence of images, for example, MRI data, the work flows as follows:

1. Load MRI data to COMSOL Script or MATLAB. The 3D MRI data is typically represented as a sequence of 2D images.
2. Create a set of 2D geometry sections from the images using the `flim2curve` command.
3. Use the `loft` command to create an interpolating 3D surface between the 2D geometry sections.
4. Define your FEM problem on the resulting geometry.

Example—Creating a 3D Solid From MRI Data

The following example shows how to create a 3D solid object from MRI data.

1. Start by loading the MRI data from MATLAB:

```
load mri
```

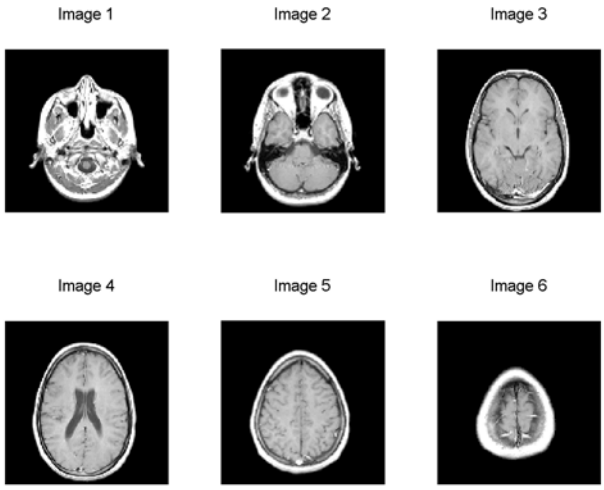
This loads a sequence of images stored in multidimensional array called `mri`. It also loads a colormap named `map`, for illustrating the images.

2. Create an index vector indicating which sections to include:

```
i = [1 6 12 17 22 27];
```

3. Visualize the sections:

```
figure
for k = 1:6
  subplot(2,3,k)
  image(D(:,:,1,i(k)))
  title(sprintf('Image %d',k))
  axis off
end
colormap(map)
```



4 Before creating curve objects from the images, you must define the `threshold` and `keepfrac` data. Use the threshold value to create a binary image where all parts of the image with a lower value than the threshold value are white. The function `imcontour` uses this binary image for creating contour curves. When the threshold value is set to 1, the contour curves are created on the boundaries of the regions of the images that are completely black. Use the `KeepFrac` property in `flim2curve` for determining how many points in the contours created from the images should be used for creating a curve object. You must tune this value for all images so that every created curve object consists of the same number of segments. This is crucial for creating a 3D geometry model of the cross-section images.

```
th = [1 1 1 1 1 1];
kf = [0.11 0.10 0.112 0.115 0.129 0.165];
```

5 Loop over sections and save curve data in the cell array `c`:

```
clear c
for k = 1:6
  [c{k},r] = ...
  flim2curve(D(:,:,1,i(k)),{th(k),[]},'KeepFrac',kf(k));
end
```

6 Convert 2D curve objects to 2D solid objects:

```
for k = 1:6
  c{k} = solid2(c{k});
end
```

Now, each of these solid objects could be used for creating a 2D mesh, and for solving 2D problems. Alternatively, the objects could be given specific names and then imported into COMSOL Multiphysics. For example, create

```
s1=c{1}
```

and import the solid object `s1` into COMSOL Multiphysics using **Import>Geometry Objects** from the **File** menu. You can then continue modeling from COMSOL Multiphysics, creating a mesh, specifying the boundary conditions and material parameters, solving, and visualizing the results.

Instead, in this example, the 2D solids are seen as 2D cross sections of a 3D geometry, as follows.

7 Create a mapping table that maps edges between sections. See the `loft` help text or the *COMSOL Multiphysics Reference Guide* entry:

```
el = {18 18 18 19 19 18};
```

This means that edge 18 of section 1 is mapped on edge 18 of section 2. The rest of the edge mappings between sections 1 and 2 are then automatically derived from this. The mappings between sections 2,3; 3,4; 4,5; and 5,6 are: 18-18, 18-19, 19-19, and 19-18, respectively.

8 Create 3D positioning data for the sections:

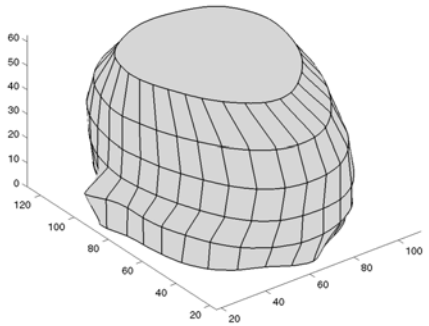
```
dvr = {repmat(12.5,1,5),repmat(0,2,6),repmat(0,1,6)};
```

9 Loft between the sections using cubic lofting (the default lofting method), with the lofting weights explicitly given:

```
lg = loft(c,'loftedge',el,'loftsecpos',dvr,...
  'loftweights',repmat(0.1,2,5));
```

10 Visualize the geometry:

```
figure
geomplot(lg)
```



11 To import the geometry object into COMSOL Multiphysics, choose **Import> Geometry Objects** from the **File** menu. You can then use this geometry as part of a simulation: create a mesh, solve, and postprocess the solution. Note that to be able to mesh the geometry on a computer with limited memory resources, you may need to select a coarse mesh setting.

Spline and Surface Interpolation

SPLINE INTERPOLATION FOR CREATING GEOMETRY OBJECTS

To create a curve object from a set of data points, use the function `geomspline`. It creates C^1 or C^2 continuous splines for the interpolation between the points.

Example—Creating a Geometry Using Spline Interpolation

The following example illustrates the use of `geomspline` by creating a geometry object with splines created from irregularly distributed points on a circle.

1 First create the circle data, sorted by the angle, and remove some of the points:

```
phi = 0:0.2:2*pi;
phi([1 3 6 7 10 20 21 25 28 32]) = [];
p = [cos(phi);sin(phi)];
```

2 Add some noise to the data points.

```
randn('state',17)
p = p+0.01*randn(size(p));
```

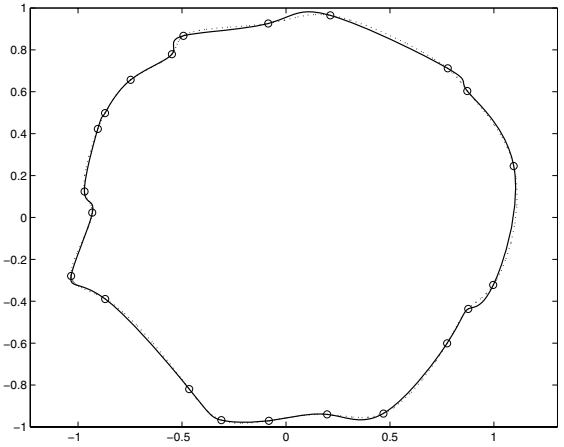
3 You can use different global parameterization techniques with `geomspline`. The global parameterization is a parameter that varies from 0 at the first interpolated point to 1 at the last interpolated point. For a closed curve, these two points coincide. For details on the different parameterization techniques, see the reference entry on `geomspline` in the *COMSOL Multiphysics Reference Guide*.

First use a uniform parameterization technique. This is the most straightforward technique, but it does not always work well on irregularly distributed data.

```
plot(p(1,:),p(2,:),'ro')
c = geomspline(p,'splinemethod','uniform','closed','on');
hold on
geomplot(c,'pointmode','off')
```

Then interpolate using centripetal parameterization and notice the difference:

```
c = geomspline(p,'splinemethod','centripetal','closed','on');
hold on
geomplot(c,'pointmode','off','edgecolor','b')
```



4 For creating a solid geometry from this, on which you can base an analysis, enter

```
s = solid2(c);
```

5 To use this geometry for analysis in COMSOL Multiphysics, choose **Import>Geometry Objects** from the **File** menu. Do this when the Draw mode is active, either when setting up a 2D problem or when you have defined a work plane in a 3D model. Once this is done, you can use any of the available geometry operations to continue working on the model.

Occasionally, point data can be too dense or contain too much noise to be practical to work with. The functions `flcontour2mesh` and `flmesh2spline` contain basic functionality to reduce the number of edge segments in a point set. To understand the MATLAB contour data format, see the functions `contour` and `contourc` in COMSOL Script or MATLAB.

You can also use `geomspline` for creating 3D curve objects. Using the data created in the previous example, define the *z* coordinates to create the following 3D curve object:

```
p = [p;linspace(0,3,22)];
c = geomspline(p,'splinemethod','centripetal','closed','off')
```

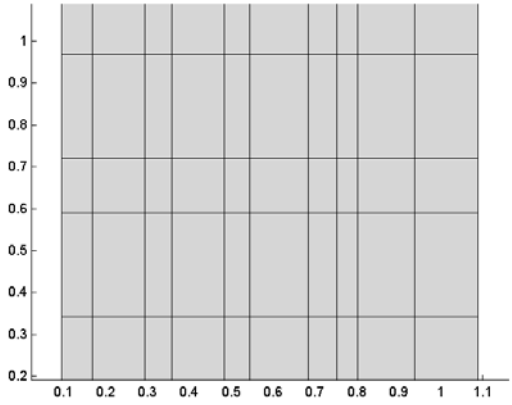
SURFACE INTERPOLATION

You can create a 3D face object or a 2D solid object from a 3D point set or a 2D point set, respectively, using the function `geomsurf`. See the *COMSOL Multiphysics Reference Guide* entry on `geomsurf` for more information.

2D Solid Objects

Using `geomsurf`, you can create a solid object with rectangular subdomains from a rectangular point set. In the following example, a solid object with nonuniformly distributed subdomains is generated from a nonuniform grid.

```
[x,y] = meshgrid(0:0.1:1,0:0.2:1);
rand('state',1);
x = x+repmat((0.1*rand(1,size(x,2))),size(x,1),1);
rand('state',1);
y= y+repmat((0.2*rand(size(y,1),1)),1,size(y,2));
s = geomsurf(x,y);
figure
geomplot(s,'pointmode','off')
```



This can be used for defining a domain with a nonuniform distribution of some material parameter.

You can import this geometry object into COMSOL Multiphysics by choosing **Import>Geometry Objects** from the **File** menu.

3D Face Objects

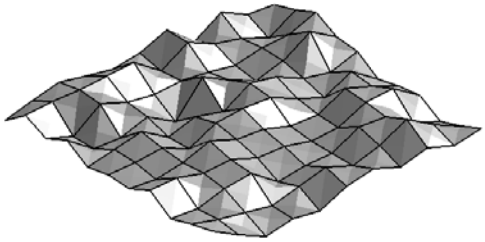
By using the function `geomsurf`, you can create a surface from height data defined on a grid by bilinear interpolation. The surface created is identical in shape to the surface created with the command `surf`. The following example illustrates how to generate a surface from simulated or measured data and how to insert the generated surface as the top surface of a solid block.

1. The height data is defined as deviations, with a maximum deviation of 0.03 from a reference plane situated at z equal to 0.7.

```
dev_max = 0.03;
ref_level = 0.7;
[x,y] = meshgrid(0:0.1:1,0:0.1:1);
randn('state',1);
z = ref_level+dev_max*randn(size(x));
```

2. Create a surface from the height data:

```
f = geomsurf(x,y,z);
```



3 Create a solid block whose top surface lies above the generated surface.

```
b = block3(1,1,ref_level+3*dev_max);
```

4 Coerce the face and the solid block to a composite solid object.

```
g = geomcoerce('solid',{f,b});
```

5 Split the object into its subdomains.

```
ss = split(g);
```

6 Visualize the solid object with the interpolated surface as top surface.

```
figure
geomplot(ss{1})
```

7 Use use the **Headlight** or **Scene Light** toolbar buttons to turn on lighting in COMSOL Script.

5

Creating Meshes

This chapter describes the scripting tools for meshing. Topics covered also include importing and exporting meshes.

Generating Meshes

Overview

The *mesh* of a PDE problem is stored in the *mesh object* or, if the model contains several mesh cases, the *mesh structure* `fem.mesh`. The mesh divides subdomains into elements, and it also divides boundaries into boundary elements.

To create a mesh from an analyzed geometry, use the functions `meshinit`, `meshmap`, `meshextrude`, `meshrevolve`, `meshsweep`, and `meshbndlayer`. You can then refine and smooth the mesh using the functions `meshrefine` and `meshsmooth`, respectively. To copy a mesh between boundaries, use the function `meshcopy`. The `adaption` function creates mesh data as part of the solution process.

You can plot a mesh with the `meshplot` function.

Use the `get` function to retrieve properties for a mesh object. The entries on `femmesh` and `femmesh/get` in the "Command Reference" chapter of the *COMSOL Multiphysics Reference Guide* contain details on the mesh data representation and how to access it. Type the name of the mesh object (for example, `fem.mesh`) and then press Return to get the most important statistics for the mesh: the number of mesh vertices, vertex elements, boundary elements, and mesh elements of different types (triangular, quadrilateral, and so on).

Except when you perform a mesh import, the mesh data structure contains only the node points of the vertices of the mesh elements. To perform finite element modeling on a mesh you must also add the node points corresponding to higher-order elements and the finite element types. This is handled by the `meshextend` function, which creates the field `fem.xmesh` that contains an *extended mesh*. Use the function `xmeshinfo` to query the extended mesh for information.

Mesh Creation Functions

CREATING FREE MESHES USING MESHINIT

To create a free mesh from an analyzed geometry, use the function `meshinit`. It provides a number of input properties that you can use to control, for example:

- The maximum mesh size and curvature mesh size
- The growth rate for the mesh size (away from small details)

- The resolution of narrow regions
- Geometry scaling before meshing
- The distribution of mesh elements on selected edges
- Which domains of the geometry to mesh
- Mesh element type for each domain (only available for subdomains in 2D and faces in 3D).

Many of these properties are available both globally and locally.

There are nine predefined settings you can use to set a suitable combination of values for many properties. To select one of these settings, use the property `hauto` and pass an integer from 1 to 9 as its value to describe the mesh quality:

- Extremely fine (1)
- Extra fine (2)
- Finer (3)
- Fine (4)
- Normal (5)
- Coarse (6)
- Coarser (7)
- Extra coarse (8)
- Extremely coarse (9)

The default value is 5, that is, the Normal mesh settings. See the *COMSOL Multiphysics Reference Guide* entry on `meshinit` for details about the mesh parameter values for the predefined settings and all other properties.

Example—Creating a 2D Mesh with Triangular Elements

Generate a triangular mesh of a unit square:

```
clear fem
fem.geom = geomcsg({square2(0,0,1)});
fem.mesh = meshinit(fem);
figure, meshplot(fem), axis equal
```

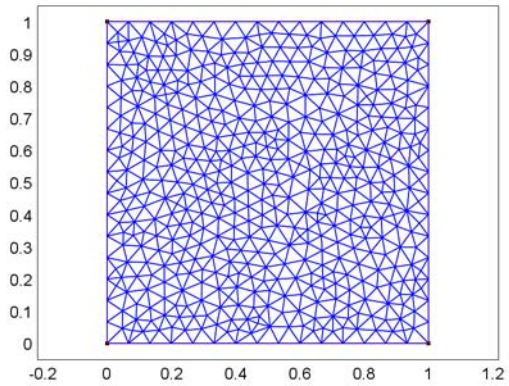


Figure 5-1: Default mesh on a unit square.

Make the mesh finer than the default by specifying a maximum element size of 0.02:

```
fem.mesh = meshinit(fem,'hmax',0.02);
figure, meshplot(fem), axis equal
```

This value corresponds to 1/50 of the largest axis-parallel distance, whereas the default value is 1/15.

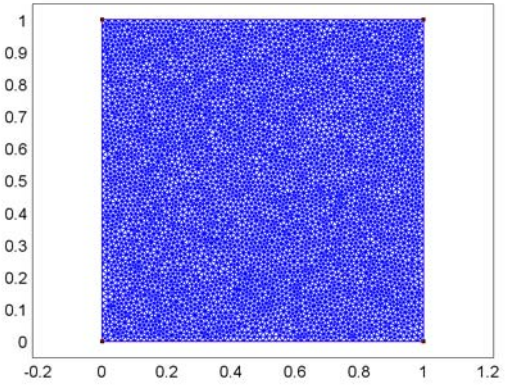


Figure 5-2: Fine mesh (maximum element size = 0.02).

Sometimes a nonuniform mesh is desirable. Make a mesh that is denser on the left-hand side by specifying a smaller maximum element size only on the edge segment to the left (edge number 1):

```
fem.mesh = meshinit(fem,'hmaxedg',[1; 0.02]);
figure, meshplot(fem), axis equal
```

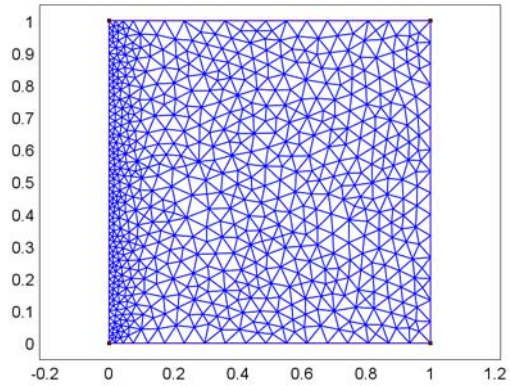


Figure 5-3: Nonuniform mesh.

Example—Creating a 2D Mesh with Quadrilateral Elements

By default the mesh element type generated by the meshinit command is triangles in 2D. To create an unstructured quadrilateral mesh on a unit circle, specify the property `methodsub` as `quad`:

```
clear fem
fem.geom = circ2;
fem.mesh = meshinit(fem,'hauto',3,'methodsub','quad');
figure, meshplot(fem), axis equal
```

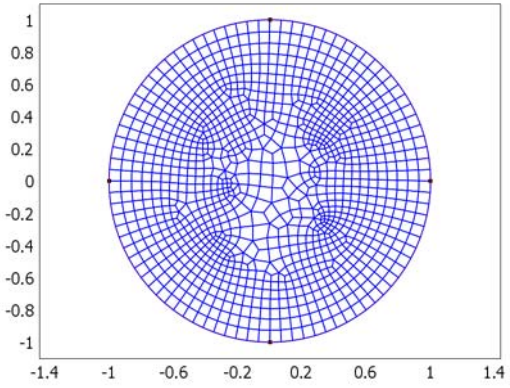


Figure 5-4: Free quad mesh.

CREATING MAPPED MESHES

To create mapped meshes, that is, structured quadrilateral meshes in 2D, use the `meshmap` function. By extruding or revolving 2D meshes you can create prism and block meshes in 3D (see the section "Extruding and Revolving 2D Meshes" on page 313 of the *COMSOL Multiphysics User's Guide*.). The software uses a mapping technique to create the quadrilateral mesh. For each subdomain, the algorithm generates a logical mesh on a block geometry and then maps it onto the real geometry by transfinite interpolation. The input geometry object must meet the following criteria for the mapping technique to work:

- Each subdomain must be bounded by one connected boundary component only.
- Each subdomain must be bounded by at least four boundary segments.
- The geometry must not contain isolated vertices or boundary segments.
- The shape of each subdomain must not differ too much from a rectangular shape.

Use the `edgegroups` property to group the edges (boundaries) into four edge groups, one for each edge of the logical mesh. You can also control the edge element distribution, which determines the overall mesh density.

Example—Creating a Mapped Mesh

Create a mapped quadrilateral mesh on a geometry where the subdomains are bounded by more than four edges:

```
clear fem
fem.geom = rect2+rect2(1,2,0,1)-circ2(1.1,-0.1,0.4);
figure, geomplot(fem,'edgelabels','on')
fem.mesh = meshmap(fem,'edgegroups',...
    {{1 3 2 [4 8]},{4 5 7 [9 10 6]}});
figure, meshplot(fem)
```

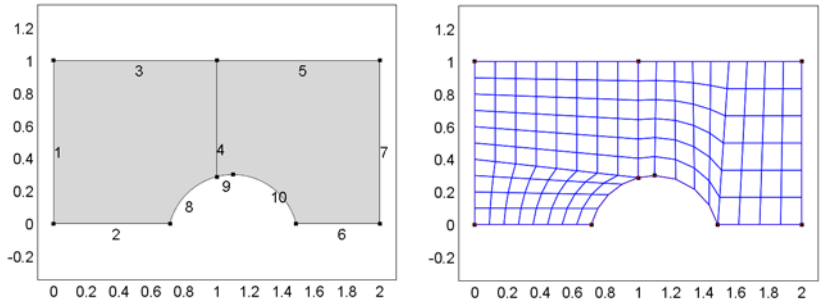


Figure 5-5: Mapped quadrilateral mesh (right) and its underlying geometry.

The `edgegroups` property specifies that the four edges enclosing Subdomain 1 are Boundary 1; Boundary 3; Boundary 2; and Boundaries 4 and 8. For Subdomain 2 the four edges are Boundary 4; Boundary 5; Boundary 7; and Boundaries 9, 10, and 6.

CREATING MESHES INTERACTIVELY

With the property `meshstart` you can create meshes in a step-by-step fashion using the `meshinit` and `meshmap` commands. The following example illustrates the procedure:

```
clear fem;
geom0 = rect2+circ2;
figure, geomplot(geom0,'edgelabels','on');
fem.geom = geomdel(geom0,'edge',8);
figure, geomplot(fem,'sublabels','on','edgelabels','on')
mesh0 = meshmap(fem,'subdomain',2,...
    'edgelem',{[1:4],[logspace(0,2,25)/100-0.01]});
fem.mesh = meshinit(fem,'meshstart',mesh0);
figure, meshplot(fem)
```

The final mesh appears in Figure 5-6. Also note the effect of the property `edgeelem` (available for both `meshinit` and `meshmap`), with which you can control the distribution of vertex elements along geometry edges.

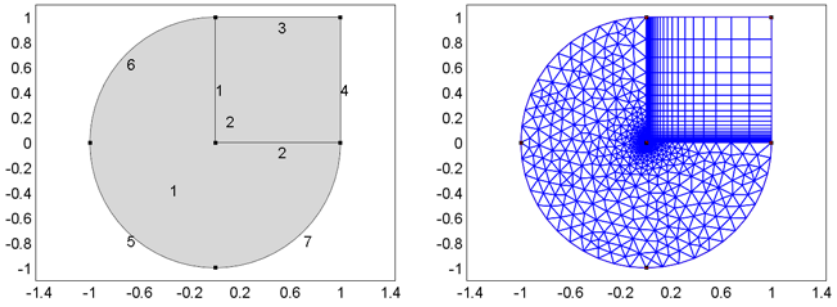


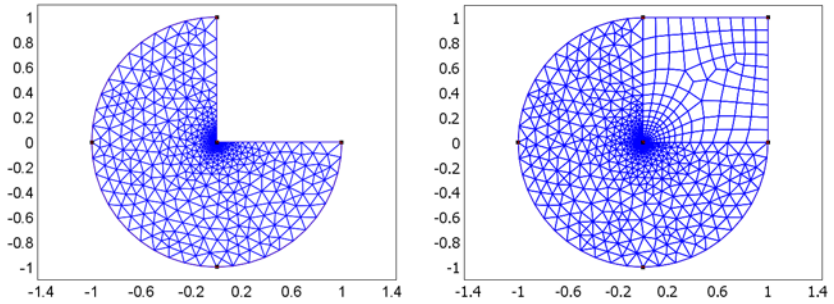
Figure 5-6: Interactively generated mesh (right).

Furthermore, you can selectively remove parts of a mesh by using the command `meshdel`. For example, to remove the mapped mesh from subdomain 2 along with the adjacent edge mesh on edges 3–4, and replace it with a free quad mesh, enter these commands:

```
fem.mesh = meshdel(fem,'subdomain',2,'edge',[3 4]);
figure, meshplot(fem)
fem.mesh = meshinit(fem,'meshstart',fem.mesh,'hauto',4,...
    'methodsub','quad');
```

```
figure, meshplot(fem)
```

The result looks like this:



As usual, for further details on the various commands and their properties consult the corresponding command-line help entries or the *COMSOL Multiphysics Reference Guide*.

CREATING 3D MESHES BY EXTRUDING AND REVOLVING 2D MESHES

Using the `meshextrude` and `meshrevolve` functions you can create 3D meshes by extruding and revolving 2D meshes. Depending on the 2D mesh type, the 3D meshes can be hexahedral (brick) meshes or prism meshes.

Example—Revolving a 2D Mesh

Create and visualize a revolved prism mesh as follows:

```
clear fem
fem.geom = circ2(1,'pos',[2 0]);
fem.mesh = meshinit(fem);
figure, meshplot(fem);
fem1 = meshrevolve(fem,'angles',[-pi/3 pi/3]);
figure, meshplot(fem1,'edgecolor','y');
```

To obtain a torus, leave the `angles` property unspecified; the default value gives a complete revolution.

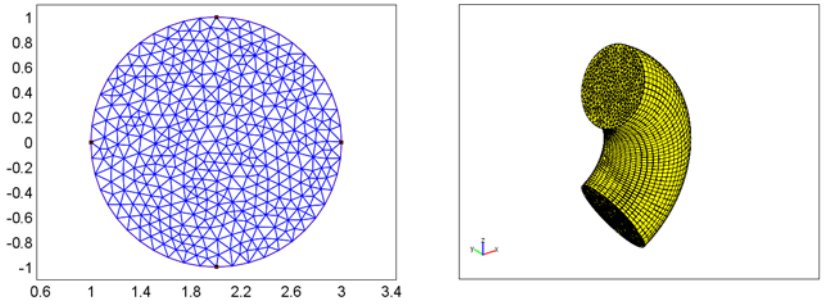


Figure 5-7: 2D triangular mesh (left) and 3D prism mesh created by `meshrevolve`.

Example—Extruding a 2D Mesh

To generate a 3D prism mesh from the same 2D mesh by extrusion and then plot it, enter the following commands:

```
fem2 = meshextrude(fem,'distance',1, ...
    'elextlayers',{logspace(0,1,11)-1});
figure, meshplot(fem2,'edgecolor','y');
```

The result appears in Figure 5-8. With the optional property `elextlayers` you control the number and distribution of mesh element layers in the extruded direction. Type `help meshextrude` at the command prompt to find out how to specify the argument and to learn about additional available properties.

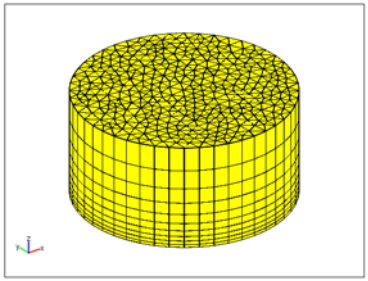


Figure 5-8: Extruded 3D prism mesh.

CREATING SWEPT MESHES

Another method of generating a structured 3D mesh from a 2D mesh on a specific face of a 3D object is by sweeping. The following example illustrates the procedure:

```
cyl = cylinder3(1,1);
figure, geomplot(cyl,'facelabels','on')
sm = meshsweep(cyl,'sourceface',{1 1});
figure, meshplot(sm,'edgecolor','y');
```

You can examine the resulting 3D mesh on the right side of Figure 5-9.

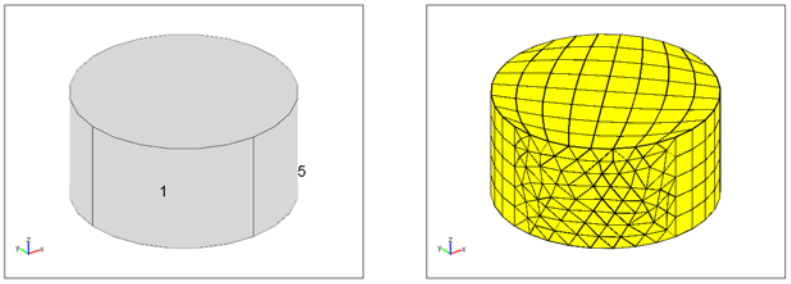


Figure 5-9: 3D mesh created by sweeping (right).

If, as in this case, there is no mesh on the source face when calling `meshsweep`, one is automatically generated. Moreover, if you do not specify a target face, the algorithm automatically identifies it on the basis of the source face and vice versa. To obtain a full list of the properties that you can specify and other details about `meshsweep`, see the function's help entry.

COMBINING FREE AND STRUCTURED MESHES

Like `meshinit` and `meshmap`, the function `meshsweep` also accepts the optional properties `meshstart` and `subdomain`, allowing you to generate meshes subdomain by subdomain. In this example, having created the geometry on the left in Figure 5-10, you begin by meshing subdomain 1 using `meshsweep`. To the structured partial mesh thus generated you then adjoin a free tetrahedral mesh of subdomain 2:

```
clear fem2
fem2.geom = ...
   cone3(0.3,1,pi/20,'pos',[0 0.5 0.5],'axis',[-1 0 0]) + ...
   block3(1,1,1,'pos',[0 0 0]);
figure, geomplot(fem2.geom,'facemode','off', ...
   'facelabels','on','sublabels','on')
fem2.mesh = meshsweep(fem2,'subdomain',[1],'sourceface',{1 1});
fem2.mesh = meshinit(fem2,'meshstart',fem2.mesh);
```

```
figure, meshplot(fem2.mesh,'edgecolor','y')
```

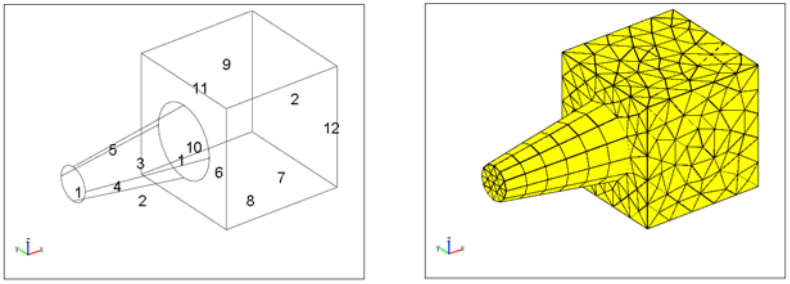


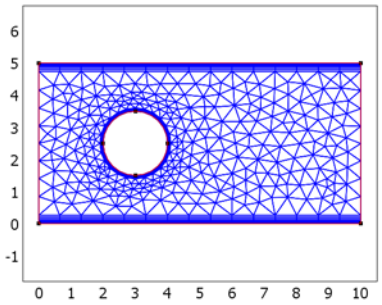
Figure 5-10: Combined structured/free mesh.

CREATING BOUNDARY LAYER MESHES

For 2D and 3D geometries it is also possible to create boundary layer meshes using the `meshbndlayer` function. A boundary layer mesh is a mesh with dense element distribution in the normal direction along specific boundaries. This type of mesh is typically used for fluid flow problems to resolve the thin boundary layers along the no-slip boundaries. In 2D, a layered quadrilateral mesh is used along the specified no-slip boundaries. In 3D, a layered prism mesh or hexahedral mesh is used depending on if the corresponding boundary layer boundaries contain a triangular or a quadrilateral mesh.

If you start with an empty mesh, the boundary layer mesher creates a free mesh before inserting boundary layers into the mesh. This generates a mesh with triangular and quadrilateral elements in 2D and tetrahedral and prism elements in 3D. The following example illustrates the procedure in 2D:

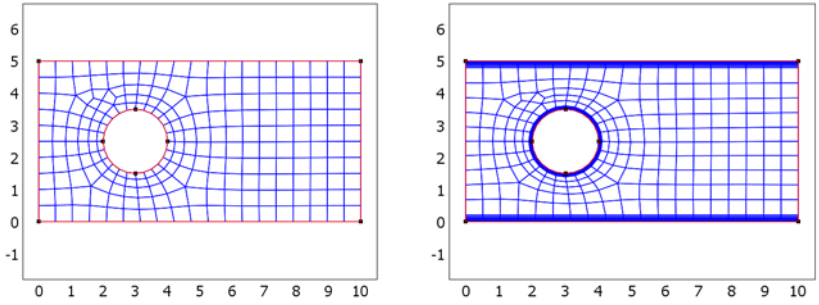
```
clear fem
fem.geom = rect2(10,5) - circ2(1,'pos',[3 2.5]);
fem.mesh = meshbndlayer(fem,'blbnd',[2:3 5:8]);
figure, meshplot(fem), axis equal
```



Boundary layer mesh based on a free triangular mesh.

It is also possible to insert boundary layers in an existing mesh. The following example shows how to add boundary layers to a free quadrilateral mesh:

```
clear fem
fem.geom = rect2(10,5) - circ2(1,'pos',[3 2.5]);
fem.mesh = meshinit(fem,'methodsub','quad',...
    'hauto',4);
figure, meshplot(fem), axis equal
fem.mesh = meshbndlayer(fem,'blbnd',...
    [2:3 5:8],'meshstart',fem.mesh,'subdomain',1);
figure, meshplot(fem), axis equal
```



Initial free quad mesh (left) and resulting boundary layer mesh (right).

MEASURING MESH QUALITY

Use the `meshqual` function to compute a measure of the mesh quality. This measure is a scalar quantity, defined for each mesh element, where 0 represents the lowest

quality and 1 represents the highest quality (see the command line help for `meshqual` or the *COMSOL Multiphysics Reference Guide* for details on the quality measure).

To visualize a mesh's quality use `meshplot`. The following commands illustrate what you can do using the meshed unit circle discussed earlier as a starting point:

```
qual = meshqual(fem.mesh); minq = min(qual); maxq = max(qual);
figure, meshplot(fem.mesh,'elmode','on','elcolor','qual',...
  'elkeep',0.25,'elkeeptype','min','qualdlim',[minq maxq],...
  'qualmap','bone')
figure, meshplot(fem.mesh,'elmode','on','elcolor','qual',...
  'elkeep',0.25,'elkeeptype','max','qualdlim',[minq maxq],...
  'qualmap','bone')
```

These `meshplot` commands display the worst 25% and the best 25% elements in terms of mesh element quality. Notice in Figure 5-11 how the triangular mesh elements in the plot to the right are more regular than those in the left plot; this reflects the fact that a quality measure of 1 corresponds to a uniform triangle, while 0 means that the triangle has degenerated into a line.

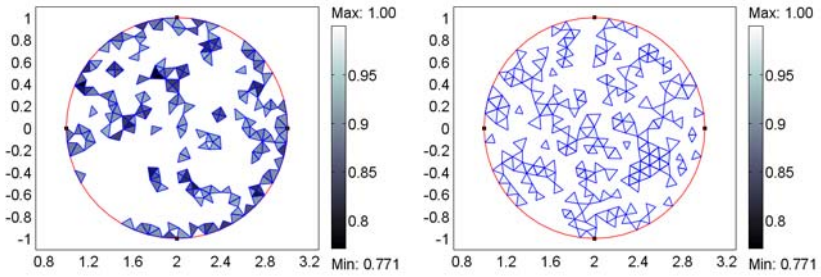


Figure 5-11: Visualizations of the mesh quality: worst 25% (left) and best 25% (right).

REFINING AND SMOOTHING MESHES

Use the function `meshsmooth` to improve the quality of triangular meshes in 2D, and tetrahedral meshes in 3D. The call

```
fem.mesh = meshsmooth(fem,...);
```

improves the quality of the elements in the mesh `fem.mesh` by adjusting the positions of nonboundary mesh vertices and swapping mesh edges. You can control the process by specifying the properties `jiggleiter` and `qualoptim`; see the *COMSOL Multiphysics Reference Guide* or the `meshsmooth` help entry for more details.

Given a mesh consisting only of simplex elements (lines, triangles, and tetrahedra) you can create a finer mesh using the function `meshrefine`. Thus, the call

```
fem.mesh = meshrefine(fem, ...);
```

returns a refined version of the mesh specified by the geometry `fem.geom` and the mesh `fem.mesh`.

By specifying the property `tri`, either as a row vector of element numbers or a 2-row matrix, you can control which elements are to be refined. In the latter case, the second row of the matrix specifies the number of refinements for the corresponding element.

The refinement method is controlled by the property `rmethod`. In 2D its default value is `regular`, corresponding to regular refinement, in which each specified triangular element is divided into four triangles of the same shape. Setting `rmethod` to `longest` gives longest edge refinement, where the longest edge of a triangle is bisected. Some triangles outside the specified set might also be refined in order to preserve the triangulation and its quality.

In 3D the default refinement method is `longest`, while regular refinement is only implemented for uniform refinements. In 1D the function always uses `regular` refinement, where each element is divided into two elements of the same shape.

For stationary or eigenvalue PDE problems you have the option of using adaptive mesh refinement at the solver stage by calling the function `adaption`. Further details on this strategy appear in the section "Adaptive Mesh Refinement" on page 388 in the *COMSOL Multiphysics User's Guide*.

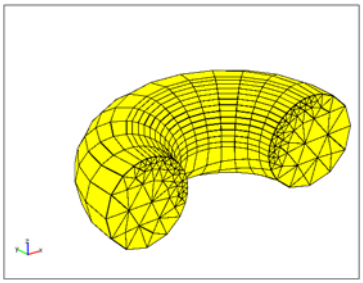
COPYING BOUNDARY MESHES

Use the `meshcopy` command to copy a mesh between boundaries in 2D and 3D. It is only possible to copy meshes between boundaries that have the same shape. However, a scaling factor between the boundaries is allowed. The following example demonstrates how to copy a mesh between two boundaries in 3D and then create a swept mesh on the subdomain.

```
clear fem
g1 = circ2(0.5,'pos',[0,1]);
g2 = revolve(g1,'angles',[0,pi],'revaxis',[0 1;0 0],...
  'wrkpln',[0 0 1;0 0 0;0 1 0]);
fem.geom = geomcsg({g2});
fem.mesh = meshinit(fem,'hmaxedg',[6,0.06],...
  'point',[],'edge',[],'face',[1],'subdomain',[]);
fem.mesh = meshcopy(fem,'source',1,'target',10);
fem.mesh = meshsweep(fem,'sourceface',{1 1},...
  'targetface',{1 10},'meshstart',fem.mesh);
```

```
figure, meshplot(fem,'edgecolor','y')
```

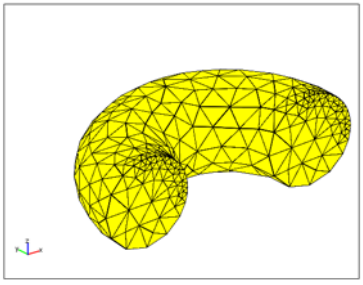
The algorithm automatically determines how to orient the source mesh on the target boundary, and the result looks like this:



You can explicitly control the orientation of the copied mesh by specifying the properties `sourceedg`, `targetedg` and `direction`. The command sequence

```
fem.mesh = meshinit(fem,'hmaxedg',[6,0.06], ...
  'point',[],'edge',[],'face',[1],'subdomain',[]);
fem.mesh = meshcopy(fem,'source',1,'target',10,...
  'sourceedg',1,'targetedg',17,'direction','same');
fem.mesh = meshinit(fem,'meshstart',fem.mesh);
figure, meshplot(fem,'edgecolor','y')
```

copies the mesh between the same boundaries as in the previous example, but now the orientation of the source mesh on the target boundary is different. The subdomain is then meshed by the free mesher, resulting in the mesh shown in the next figure. Note that in this case it is not possible to create a swept mesh on the subdomain, because the boundary meshes do not match in the sweeping direction.

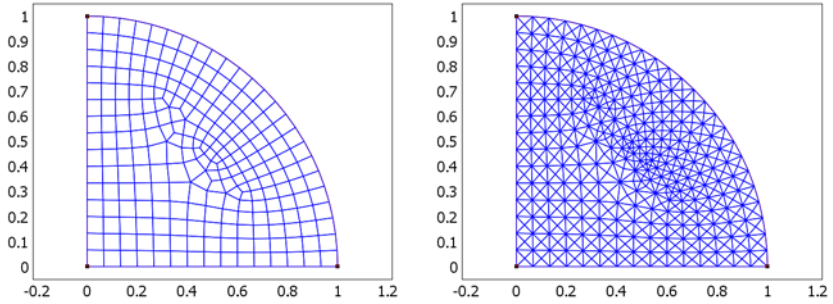


CONVERTING MESHES

Quadrilateral meshes in 2D, and hexahedral meshes in 3D, can be converted into triangular meshes and tetrahedral meshes, by using the `meshquad2tri` and the `meshhex2tet` command, in 2D and 3D, respectively. In 2D each quadrilateral element is split into either two or four triangles, and in 3D each hexahedral element is split into five or 28 tetrahedral elements. The method used to convert the elements is controlled by the property `splitquad` in 2D, and the property `splithex` in 3D. The default value for both properties is `diagonal`, which results in two triangular elements in 2D and five tetrahedral elements in 3D. For more information on the properties, see the *COMSOL Multiphysics Reference Guide* or the `meshquad2tri` and `meshhex2tet` help entries. The example below demonstrates how a free quad mesh can be converted into a triangle mesh.

```
clear fem
fem.geom = circ2*rect2;
fem.mesh = meshinit(fem,'methodsub','quad');
figure, meshplot(fem), axis equal
fem.mesh = meshquad2tri(fem,'splitquad','center');
figure, meshplot(fem), axis equal
```

The result looks like this:



CREATING AN ANALYZED GEOMETRY FROM A MESH

Use the `mesh2geom` command to convert a mesh to a geometry, for example to be able to remesh the geometry. This example starts by drawing two rectangles, one inside the other, and meshes them:

```
clear fem
g1=rect2(1.8,1.2,'base','corner','pos',[-0.8,-0.8]);
g2=rect2(0.2,0.4,'base','corner','pos',[-0.2,-0.4]);
fem.geom=geomcsg({g1,g2});
fem.mesh=meshinit(fem);
```

Set the inner rectangle to move along the *x*-axis:

```
clear appl
appl.mode.class = 'MovingMesh';
appl.sdim = {'X','Y','Z'};
appl.assignsuffix = '_ale';
appl.prop.analysis='transient';
appl.prop.weakconstr.value = 'off';
appl.bnd.defflag = {{1;1}};
appl.bnd.deform = {{0;0},{'t';0}};
appl.bnd.ind = [1,1,1,2,2,2,2,1];
appl.equ.type = {'free','pres'};
appl.equ.presexpr = {{0;0},{'t';0}};
appl.equ.ind = [1,2];
fem.appl{1} = appl;
fem.sdim = {{'X','Y'},{'x','y'}};
fem.frame = {'ref','ale'};
fem=multiphysics(fem);
fem.xmesh=meshextend(fem);
```

Solve the problem:

```
fem.sol=femtime(fem,'tlist',[0:0.01:0.1]);
```

Create an analyzed geometry from the deformed mesh:

```
fem=mesh2geom(fem,'srcdata','deformed','frame','ale');
```

Remesh and continue solving:

```
fem.mesh=meshinit(fem);
fem.xmesh=meshextend(fem);
fem.sol=femtime(fem,'tlist',[0:0.01:0.1]);
```

To plot the moving mesh you can use the command

```
postmovie(fem,'triedgestyle','k','refine',1,'tridata','x')
```

which plots the mesh's movement.

REMOVING HOLES IN DOMAIN NUMBERING

The command

```
mesh = meshsqueezedom(mesh1)
```

removes any gaps in the domain numbering of `mesh1`, thus making the highest domain index value in `mesh` equal to the number of domains.

ADDITIONAL MESH COMMANDS

For the special case of a rectangular geometry, you can create a regular mesh with the function `meshpoi`. The call

```
fem.mesh = meshpoi(fem,nx,ny);
```

constructs a regular mesh on the rectangular geometry specified by `fem.geom`. It does so by dividing the *x*-edge—defined as the edge making the smallest angle with the *x*-axis—into `nx` pieces and the *y*-edge into `ny` pieces; it then places `(nx+1)*(ny+1)` points at the intersections. For example, running the code lines

```
clear fem;
fem.geom = rect2(2,1,'rot',pi/6);
fem.mesh = meshpoi(fem,20,10);
figure, meshplot(fem)
```

produces the plot in Figure 5-12.

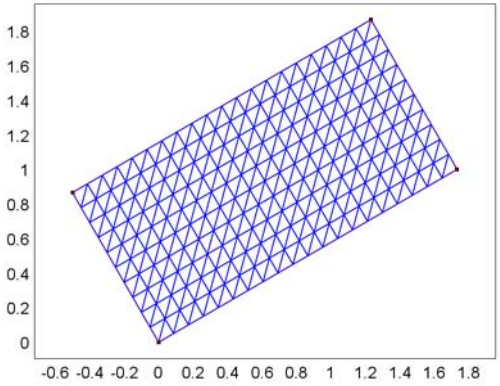


Figure 5-12: Regular mesh on rectangular 2D geometry created with the function `meshpoi`.

Importing and Exporting Meshes

Importing External Meshes and Mesh Objects

It is possible to import meshes to the workspace and to COMSOL Multiphysics using the following formats:

- COMSOL Multiphysics text files (extension `.mphtxt`)
- COMSOL Multiphysics binary files (extension `.mphbin`)
- NASTRAN files (extension `.nas` or `.bdf`)

For a description of the text file format see the *COMSOL Multiphysics Reference Guide*.

IMPORTING MESHES TO THE COMMAND LINE

To import a mesh stored in a supported format use the `meshimport` command. Assuming you have set the current working directory to `models/COMSOL_Multiphysics/Structural_Mechanics/` under the COMSOL Multiphysics installation directory, the following commands import and plot a NASTRAN mesh for a crankshaft:

```
m = meshimport('crankshaft.nas','linearelem','on');
meshplot(m{1})
```

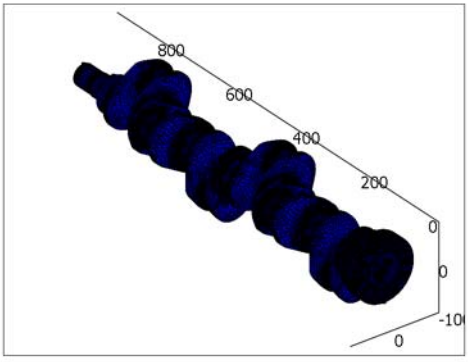


Figure 5-13: NASTRAN mesh imported into COMSOL Script.

The `linearelem` property uses linear elements in the COMSOL Multiphysics mesh object and ignores extended node points; here it serves to make a smaller mesh for

visualization purposes. See the *COMSOL Multiphysics Reference Guide* for additional properties supported by the `meshimport` function.

IMPORTING MESHES TO THE GRAPHICAL USER INTERFACE

You can import meshes in the supported file formats directly to COMSOL Multiphysics by choosing **File>Import>Mesh From File**. For more information about the settings that you can use in COMSOL Multiphysics see "Importing and Exporting Meshes" on page 346 in the *COMSOL Multiphysics User's Guide*.

Note: Importing a mesh from a file removes all geometry objects from the current geometry.

As the basis for a new model you can also import a mesh object from the workspace into COMSOL Multiphysics, either along with a compatible analyzed geometry or without a geometry. For instance, to import the mesh `mymesh` together with an associated analyzed geometry, `mygeom`, type

```
clear fem
fem.geom = mygeom;
fem.mesh = mymesh;
```

otherwise type

```
clear fem
fem.mesh = mymesh;
```

Then choose **Import>FEM Structure** from the COMSOL Multiphysics **File** menu.

Using a Mesh to Represent a Geometry

If you do not have a geometry, you can use a mesh to represent it. For instance, to generate a geometry from the mesh object `mymesh`, type the following commands:

```
clear fem
fem.geom = geomobject(mymesh);
fem.mesh = mymesh;
```

Then choose **Import>FEM Structure** from the **File** menu in COMSOL Multiphysics.

As an alternatively, use the mesh as a geometry without importing the actual mesh. To do so, enter

```
clear fem
fem.geom = geomobject(mymesh);
```

and choose **File>Import>FEM Structure**. You can then create a new mesh using the geometry represented by this mesh.

IMPORTING MESH DATA CREATED BY EXTERNAL PROGRAMS

Use the commands `femmesh` and `meshenrich` to create mesh objects from mesh data generated by an external program. To create an initial mesh object from the external mesh data use `femmesh`. Then use `meshenrich` to enrich the initial mesh object with additional element information needed to create a valid mesh object, which you can then use in simulations or for conversion into a geometry object.

Example—Using Text File Mesh Data

Consider the text files `coord.txt` and `tet.txt` (found in the `demo/` directory), which contain the mesh vertex coordinates and element vertex indices, respectively, of a tetrahedral mesh created with an external program. To import this data into a mesh object follow these steps:

1. Load the external mesh data to the command line:

```
load coord.txt
load tet.txt
```

2. Create an initial mesh object from the external mesh data:

```
el = cell(1,0);
tet = tet+1; % Lowest mesh point index is zero in tet.txt
el{1} = struct('type','tet','elem',tet');
m = femmesh(coord',el);
```

   (Note the transpositions of the matrices `tet` and `coord`.)

3. Enrich the initialized mesh object with information required for forming a valid mesh, and finally visualize the mesh:

```
m = meshenrich(m);
meshplot(m);
```

Exporting Meshes From COMSOL Multiphysics

It is possible to export a mesh as a mesh object to the COMSOL Script workspace and save the mesh as a COMSOL Multiphysics text file or binary file.

EXPORTING MESH OBJECTS TO THE WORKSPACE

Choose **File>Export>FEM Structure** to export to the command line the mesh object together with the complete FEM structure; the mesh object is available in `fem.mesh`.

EXPORTING MESHES TO COMSOL MULTIPHYSICS FILES

Choose **File>Export>Mesh to File** to save the current mesh as a COMSOL Multiphysics text file (extension `.mphtxt`) or as a COMSOL Multiphysics binary file (extension `.mphbin`).

6

Solver Scripting

The COMSOL Multiphysics graphical user interface provides limited solver-scripting capabilities through the **Script** page in the **Solver Manager** dialog box. However, if you require full flexibility and control of the solving stage, use COMSOL Script (or MATLAB). Examples of advanced procedures available only through M-files or at the command line include:

- Full access to the FEM structure during the solving stage, enabling tailored adaptive solver scripts
- The ability to write tailored sequential solver steps and segregated solver schemes where you first solve for one variable, then use that result to solve for another variable, and so on, or for switching between different solvers.

Solver Command Overview

This section describes the syntax and usage of the main solver commands.

Computing the Solution

Use one of the following solver commands to compute the solution of the PDE problem:

- `femstatic`—solves linear and nonlinear stationary problems and parametric problems (optionally using a segregated solution scheme).
- `femlin`—solves linear or linearized stationary problems
- `femnlin`—solves nonlinear stationary problems
- `femtime`—solves time-dependent (transient) problems
- `femeig`—solves eigenvalue problems
- `adaption`—solves stationary problems and eigenvalue problems using adaptive mesh refinement

The solution is stored in the *solution object* `fem.sol`. The solution object is described in the reference entry for `femsol`, the constructor function for such objects. The matrix `fem.sol.u` contains values for the degrees of freedom. To access the number of degrees of freedom in an FEM structure, use the command `flngdof`. To access the number of solutions in a solution object, use the command `solsize`. For more information about `flngdof` and `solsize` see the *COMSOL Multiphysics Reference Guide*.

For stationary PDE problems use `femstatic`. This function detects whether a problem is linear or nonlinear and selects the appropriate solver accordingly. In certain exceptional cases it might, however, be necessary to explicitly call `femlin` or `femnlin`.

When called with the syntax

```
fem.sol = femstatic(fem,'pname','p','plist',pl,...);
```

`femstatic` solves a problem that depends on the parameter `p` for the values given in the list `pl`. The resulting solution object then has the additional field `fem.sol.plist`, and `fem.sol.u` has one column for each parameter value in `fem.sol.plist`. The parameter name is stored in `fem.sol.pname`. For more information about parametric studies, see “Parametric Studies” on page 145.

For time-dependent problems choose `femtime`. The solution's time derivative is then available in the matrix `fem.sol.ut`, but the time derivative's accuracy is good only if the time derivative has been stored using the property `outcomp` (see `femsolver`).

For eigenvalue problems the designated solver command is `femeig`.

For `femtime` and `femeig`, `fem.sol` has the additional fields `fem.sol.tlist` and `fem.sol.lambda`, respectively. In these cases, each column in `fem.sol.u` is a solution object corresponding to a time value in `fem.sol.tlist` or an eigenvalue in `fem.sol.lambda`.

For stationary problems or eigenvalue problems for which the degree of resolution of the solution need not be uniform over the entire geometry—for instance, because of sharp geometry details, localized loads, or boundary layers—consider using adaptive mesh refinement. When modeling at the command line you do this by selecting the solver command `adaption`. Consider the following example, namely Poisson's equation on an L-shaped geometry:

```
clear fem
fem.geom = rect2(1,1)+rect2(1,1,'pos',[1 0])+rect2(1,1,'pos',[0
1]);
fem.mesh = meshinit(fem);
figure, meshplot(fem)
fem.shape = 2; fem.equ.c = 1; fem.equ.f = 1; fem.bnd.h = 1;
fem.solform = 'general';
fem.xmesh = meshextend(fem);
fem = adaption(fem);
figure, postsurf(fem,'u')
figure, meshplot(fem)
```

Figure 6-1 displays the two mesh plots generated by this code: the initial default mesh on the left, and the final, refined mesh used to obtain the solution u on the right. Notice how the mesh is particularly fine near the corners. This is a consequence of the underlying algorithm being able to select which mesh elements to refine on the basis of error estimates for the solution calculated for each element.

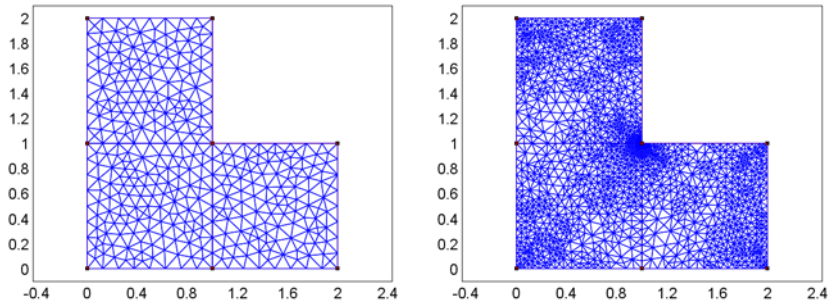


Figure 6-1: Initial mesh (left) and adaptively refined mesh (right).

This example uses the default values for all properties. To see how you can control the behavior of the adaptive mesh refinement algorithm and for additional examples (including comparisons between adaptive and uniform refinement) see the entry for `adaption` in the *COMSOL Multiphysics Reference Guide*.

Note that the `adaption` command does not work for time-dependent problems.

SYNTAX AND OUTPUT DATA

The default solver syntax is the following:

```
fem.sol = femstatic(fem);
```

The output from the solver is a solution object, which becomes the contents of `fem.sol`. The only exception to this syntax is `adaption`, which uses `fem` (the entire FEM structure) as the default output because the adaptive solver also updates the mesh.

You can control what to output from the solvers using the property `Out`, which accepts a number of outputs, of which the most important are:

- fem—the entire FEM structure
- sol—the solution object
- u—the solution vectors
- `plist`—list of parameter values (for parametric problems)
- `tlist`—list of output times (for time-dependent problems)
- `lambda`—list of eigenvalue (for eigenvalue problems)
- `mesh`—the updated mesh (for adaptive mesh refinement)

It is also possible to output the system matrices such as the stiffness matrix (after constraint elimination); see "Exploring the Stiffness Matrix" on page 137.

There are many more solver properties that you can use to control the solvers behavior. Type `help femsolver` or see the entry for femsolver in the *COMSOL Multiphysics Reference Guide* for a list of properties that are common to all solvers. See the help and documentation for each solver for the properties that are specific to each solver.

Working with the Assembled Matrices

The `assemble` command assembles the stiffness matrix, right-hand side, mass matrix, damping matrix, and constraints of a PDE problem using a finite element discretization. You can output all these matrices or one or more selected matrices. For a time-dependent problem, for example, the assembled system is the following system of ODEs

$$0 = L(U, \dot{U}, \ddot{U}, t) - N(U, t)^T \Lambda$$
$$0 = M(U, t)$$

where

- L is the *residual vector*
- M is the *constraint residual vector*
- U is the *solution vector*
- Λ is the *Lagrange multiplier vector*

The linearization of this system results in the following matrices:

$$K = -\frac{\partial L}{\partial U}, \quad D = -\frac{\partial L}{\partial \dot{U}}, \quad E = -\frac{\partial L}{\partial \ddot{U}}, \quad N = -\frac{\partial M}{\partial U}$$

where

- K is the *stiffness matrix*
- D is the *damping matrix*
- E is the *mass matrix*
- N is the *constraint Jacobian matrix*

All these matrices can depend on the solution vector U. The matrices K, D, and E can also depend on the time derivatives $\dot{U}$ and $\ddot{U}$.

For more information about the `assemble` command, see the corresponding entry in the "Command Reference" chapter of the *COMSOL Multiphysics Reference Guide*.

To get the stiffness matrix after constraint elimination, K_c, which is the matrix that COMSOL Multiphysics uses to solve the PDE problem, use the `femlin` or `femstatic` command.

Exploring the Stiffness Matrix

Because the assembled linearized systems in the finite element method are large, sparse linear systems, it is generally difficult to study the input and output of the numerical algorithms except as graphic visualizations. However, using scripting functions and the support for sparse matrices you can explore the fundamental linear-algebra aspects of the numerical model.

LINEAR-ALGEBRA ASPECTS OF THE STIFFNESS MATRIX

Some interesting aspects to consider are the following:

- If the stiffness matrix is *symmetric*. In that case, the solver can take advantage of that fact, and it is only necessary to store (approximately) half the stiffness matrix. COMSOL Multiphysics automatically detects if the stiffness matrix and other system matrices like the mass matrix are symmetric. Not all solvers take advantage of symmetry; for example, the UMFPACK direct solver and the Incomplete LU preconditioner do not treat symmetric matrices taking the symmetry into account.
- If all the eigenvalues of the stiffness matrix are positive, the matrix is *positive definite*. Another property for a positive definite matrix K is that

  $$x^T Kx > 0$$

  for any nonzero x. That the stiffness matrix is positive definite is advantageous from a numerical point of view, and it makes it possible to use a number of efficient solvers and preconditioners such as the TAUCS Cholesky direct solver and the Conjugate gradients iterative solver (the TAUCS Cholesky direct solver also requires real symmetric or complex Hermitian matrices). Simple preconditioners such as SSOR and diagonal scaling also benefit from a positive definite stiffness matrix, as do the multigrid solvers.
- If the stiffness matrix has zeros on the diagonal. This means that the matrix is *singular* (you cannot invert it). It can also be an indication that the matrix is *indefinite* (that is, $x^T Kx$ becomes both positive and negative for some different values of x). This is the case for the Navier-Stokes equations, for example, and it is generally an indication that iterative methods have trouble converging. For such cases, you must use special algorithms like the *Vanka* algorithm, where a local preconditioner/smoother acts on the so-called Vanka variables. In the case of the Navier-Stokes equations, the Vanka variable is the pressure. In COMSOL Multiphysics, the Incompressible Navier-Stokes application mode (in 3D) sets up appropriate solver settings using the Vanka algorithm with the geometric multigrid preconditioner.

THE STIFFNESS MATRIX FOR POISSON'S EQUATION

In the following examples, first define a 2D Poisson's equation as an example of a simple linear PDE; then use script coding to visualize sparsity patterns, asymmetrical parts, diagonal elements, and eigenvalues. Based on this information it is possible to classify the system and make a good judgement as to the best solver strategy. In a second step, use the same script functions to analyze the linear-algebra aspects of a fluid-flow model based on the Navier-Stokes equations.

First define a Poisson's equation on a unit disk with the boundary condition that the solution is 0 on the boundary:

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

Type the following commands to specify this PDE:

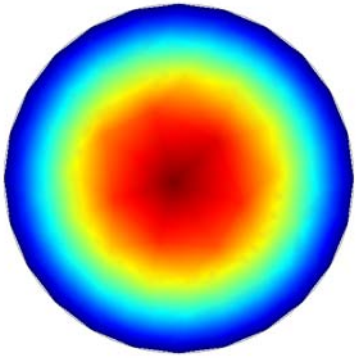
```
clear fem;
fem.geom = circ2;
fem.equ.c = 1;
fem.equ.f = 1;
fem.bnd.h = 1;
fem.bnd.r = 0;
```

Next, create a coarse mesh to make a model with a small stiffness matrix:

```
fem.mesh = meshinit(fem,'hmax',1);
meshplot(fem);
```

Extend the mesh with finite elements and solve the equation:

```
fem.xmesh = meshextend(fem);
fem.sol = femstatic(fem);
postsurf(fem,'u');
```



To get the assembled stiffness matrix (Jacobian) after constraint elimination, K_c, use the `femstatic` command but specify the output:

```
Kc = femstatic(fem,'out','Kc');
```

Next plot the sparsity pattern of K_c:

```
spy(Kc);
```

It seems that K_c is symmetric and has no zeros on the diagonal. To make sure that it is indeed symmetric, plot the nonsymmetric part:

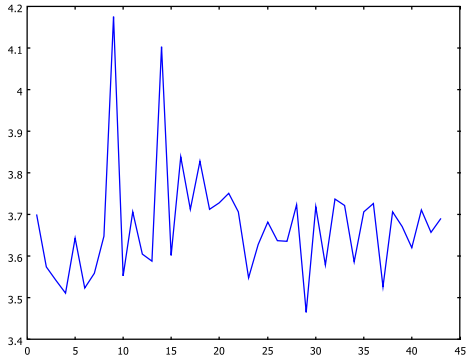
```
spy(abs(Kc-Kc')>100*eps);
```

(`Kc'` is the complex conjugate transpose of `Kc`.)

The reason to compare to a small but nonzero number is that numerically the difference between the stiffness matrix and its transpose does not become exactly zero. When deciding on a threshold value, keep in mind that the magnitude of the numerical values in the stiffness matrix can vary a lot depending on the application.

To make sure that there are no zeros on the diagonal, plot the values on the diagonal:

```
plot((abs(diag(Kc)));
```



To see if the stiffness matrix is positive definite, compute a few of its smallest eigenvalues using `eigs`, the sparse eigenvalue solver:

```
d = eigs(fem,6,'sm')

d =
   1.9350
   1.6173
   1.4120
   1.0363
   0.9321
   0.4962
```

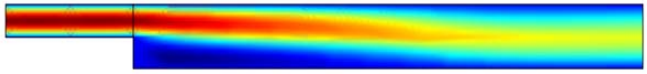
This means that all eigenvalues are positive, so the stiffness matrix is positive definite.

To summarize: The stiffness matrix (Jacobian) for Poisson's equation (after constraint elimination) is symmetric and positive definite, and it has no zeros on the diagonal, so it is nonsingular and invertible.

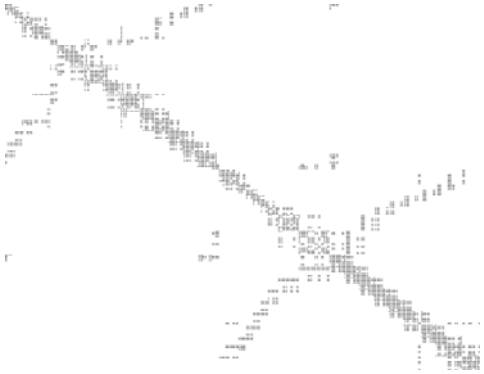
THE STIFFNESS MATRIX FOR THE NAVIER-STOKES EQUATIONS

To compare these properties with the ones that you can expect in a fluid-flow problem, use a version of the Stationary Incompressible Flow Over a Backstep model (`backstep.mph`) in the COMSOL Multiphysics Model Library, but use a coarser mesh to reduce the size of the system matrices:

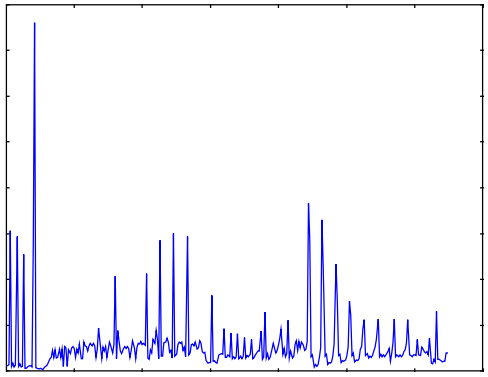
1. Open the COMSOL Multiphysics user interface.
2. Open the **Model Navigator**, and on the **Model Library** page, select **COMSOL Multiphysics>Fluid Dynamics>backstep**. Click **OK**.
3. From the **Mesh** menu, choose **Free Mesh Parameters**.
4. Click the **Predefined mesh sizes** button and make sure that **Normal** is the predefined mesh size.
5. Click the **Remesh** button and then click **OK**.
6. Re-solve the problem. You can see from the results plot that the solution is pretty coarse, but the new mesh reduces the number of degrees from roughly 22,000 to less than 500, but the characteristics of the assembled system does not change.
7. From the **File** menu, choose **Export>FEM Structure as 'fem'**.



Continue by repeating the previous commands for extracting the stiffness matrix and plotting the sparsity pattern, the nonsymmetric part, and the values of the diagonal.



The sparsity pattern for the fluid-flow model.



The values of the diagonal elements in the stiffness matrix.

This shows that the stiffness matrix is not symmetric and has a much more complex pattern. There are no zeros on the diagonal because stabilization using the GLS method for artificial diffusion is active by default. If you turn off the artificial diffusion, the stiffness matrix for this Navier-Stokes problem does contain zeros. The eigenvalues are complex, and you can compute a few of them by the following call to `eigs`:

```
d = eigs(Kc)
```

Clearly the stiffness matrix is not positive definite.

Evaluating and Visualizing the Residual

One of the outputs from the `assemble` command is L, the *residual vector*. To plot it, you can create a new FEM structure and use L as the solution (after eliminating the constraints). For good accuracy, create another mesh case in COMSOL Multiphysics and use the same mesh with elements that use shape functions that are one order higher than in the original case. For example, if the model initially used quadratic Lagrange elements, use cubic Lagrange elements in the new mesh case. The higher-order elements provide an accurate residual vector for the original model. To make the model use the new elements, choose **Update Model** from the **Solve** menu (you do not need to re-solve the model).

For information about creating mesh cases, see "Mesh Cases" on page 355 in the *COMSOL Multiphysics User's Guide*.

When you have performed these initial steps, proceed as follows:

1. Export the FEM structure to the command line as the variable `fem` (the default name).
2. Use `assemble` to compute the residual for Mesh Case 1 (the second mesh case; the first mesh case is number 0):

   ```
   L = assemble(fem,'out','L','mcase',1,'u',fem.sol.u);
   ```

3. Eliminate the constraints:

   ```
   Nu = femlin(fem,'out','Null','mcase',1);
   L = Nu*(Nu'*L);
   ```

4. Copy the FEM structure to a new variable and make its solution data be the residual:

   ```
   femO = fem;
   femO.sol = femsol(L,'mcase',1);
   ```

5. Assuming that the name of the dependent variable is u, you can plot the residual using the following call to `postplot`:

   ```
   postplot(femO,'tridata','u');
   ```

   You can also import the FEM structure to the COMSOL Multiphysics user interface to do additional postprocessing in that environment.

This method to compute the residual vector does not provide the residual within each element. Instead, the residual vector contains information about the level of the residual per mesh element.

Solver Scripting Examples

The following section provides some examples of solver scripting, sometimes in combination with using the COMSOL Multiphysics user interface.

Concatenating Several Transient Solutions

If you solve a transient problem several times (each time for a new time span), you can use scripting to concatenate the different solution into one contiguous solution for all time steps in all solutions. These solutions can come from simulations in the COMSOL Multiphysics user interface, and you can import the concatenated solution into that user interface for convenient interactive postprocessing or continued simulation in that environment. To concatenate several solutions, follow these steps:

1. Solve for the first time segment. If you do this from the user interface, choose **File>Export>FEM Structure** and type a name for the FEM structure in the **Export FEM Structure** dialog box, for example, `fem1`. Then click **OK**.
2. Update the solution time span and possibly other model settings.
3. Restart to solve for the next time span. Export that solution as, for example, `fem2`.
4. Continue doing this for any remaining time span.
5. At the command line, create a new FEM structure (for example, `fem`) by copying one of the original FEM structures and then build its solution data from all of the original FEM structures. First extract solution times and solution data:

   ```
   fem = fem1;
   tlist1 = fem1.sol.tlist;
   u1 = fem1.sol.u;
   tlist2 = fem2.sol.tlist(2:end);
   u2 = fem2.sol.u(:,2:end);
   ```

   The reason for not including the first time step in the second solution is that the restart typically appears at the same time as the ending time of the previous solution. The first solution in `fem2` is then identical to the last solution in `fem1`.
6. Continue like this for any remaining time segments.
7. Build the concatenated time steps and solutions, and then use the `femsol` function to create a new solution object:

   ```
   u = [u1, u2, ...];
   tlist = [tlist1, tlist2, ...]
   ```

```
fem.sol = femsol(u,'u','tlist',tlist);
```

Replace the ellipsis (`...`) in the previous code with all the additional time steps and solutions, if applicable.

8 If you want to postprocess the complete solution in the COMSOL Multiphysics user interface, choose **File>Import>FEM Structure**. Type `fem` in the **Import FEM Structure** dialog box if it does not appear as the default value. You can then use all the tools on the **Postprocessing** menu for visualizing and postprocessing the complete solution.

You can use the same approach to concatenate several solutions to a problem where each solution uses a different set of parameter values.

Parametric Studies

You can use `femstatic` as a parametric solver when solving stationary problems for different parameter values. For parametric studies of time-dependent or eigenvalue problems, you need to use additional scripting. The most convenient approach is to set up your model in the COMSOL Multiphysics user interface, solve it for one parameter value, and then save it as an M-file. Edit the M-file to add a loop over your parameter and save the results, and then run the M-file in COMSOL Script or MATLAB.

THE HEAT EQUATION—MODEL DEFINITION

As example, consider the basic heat equation on a square geometry, where the initial temperature is 400 K on a circle within the square, and 293 K outside the circle. The simple code that solves this problem uses direct equation-based modeling to set up the following basic time-dependent PDE:

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

```
clear fem
fem.geom = square2(2,'pos',[-1 -1]) + circ2(0.4);
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.da = 1; fem.equ.c = 1;
fem.bnd.h = 1;
fem.bnd.r = 293;
fem.equ.init = {293 400};
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'report','on','tlist',linspace(0,0.1,20));
```

The call to `femtime` solves the heat equation for 20 equally-spaced times from 0 to 0.1 s. Then plot the initial condition and the temperature at the ending time (the last time step) in to separate plot windows:

```
figure
postsurf(fem,'u','solnum',1)
figure
postsurf(fem,'u');
```

The slight overshoots and undershoots in the original solution depend on the discontinuity at the interior boundary between the circle and the rest of the geometry.

THE HEAT EQUATION—ADDING PARAMETRIC STUDIES

The following extension of the original code adds a for-loop to solve several times for different initial temperatures:

```
clear fem
fem.geom = square2(2,'pos',[-1 -1]) + circ2(0.4);
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.da = 1; fem.equ.c = 1;
fem.bnd.h = 1;
fem.bnd.r = 293;
initT = [400 478 563 617];  % Initial temperatures
n = length(initT);  % Number of parameter values
for i=1:n
   fem.equ.init = {293 initT(i)};
   fem.xmesh = meshextend(fem);
   fem.sol =
   femtime(fem,'report','on','tlist',linspace(0,0.1,20));
   femvector(i)=fem;
end
```

The code inside the loop updates the initial temperature in the circle, updates the extended mesh, solves the problem, and stores the solution for that parameter value (temperature) in a vector of FEM structures, one for each solution. You can then plot the last time step for the first solution (where the initial temperature is 400 K) and the last solution (where the initial temperature is 617 K:

```
figure;
postsurf(femvector(1),'u');
title(['Solution for initial temperature ',num2str(initT(1))]);
figure;
postsurf(femvector(n),'u');
title(['Solution for initial temperature ',num2str(initT(n))]);
```

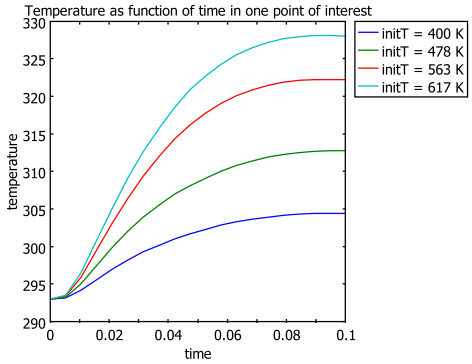
Saving the entire solution can requires a lot of memory, especially for a time-dependent 3D model. If you are only interested in some specific part of the solution, it can

therefore be a good idea to save only the relevant information. The next example shows how to store the temperature as a function of time in a specific point (in this case at $x = 0.6$ and $y = 0.4$), along with the overall maximum temperature at the end of each simulation:

```
clear fem
fem.geom = square2(2,'pos',[-1 -1]) + circ2(0.4);
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.da = 1; fem.equ.c = 1;
fem.bnd.h = 1;
fem.bnd.r = 293;
initT = [400 478 563 617];  % Initial temperatures
n = length(initT);  % Number of parameter values
times = linspace(0,0.1,20);
n_time_steps = length(times);
point_of_interest = [0.6;0.4];

for i=1:n
   fem.equ.init = {293 initT(i)};

   fem.xmesh = meshextend(fem);
   fem.sol = femtime(fem,'report','on','tlist',times);

   % Compute the maximum temperature for the last time step
   maxtemp(i) = postmax(fem,'u','solnum',n_time_steps);
   temperature(i,:) =
postinterp(fem,'u',point_of_interest,'solnum','all')';
   legendstr{i} = ['initT = ', num2str(initT(i)), ' K'];
end
```

The `postmax` function returns the maximum temperature in the entire geometry (at the last time step), whereas the `postinterp` function provides the interpolated temperature in the specific point of interest. Finally plot the temperature as a function of time in the studied point for each value of the initial temperature, using a legend to indicate what the initial temperature was for each curve:

```
figure;
plot(times,temperature');
xlabel('time');
ylabel('temperature');
title('Temperature as function of time in one point of interest');
legend(legendstr)
```



Using Scripting to Solve Multiphysics Problems Iteratively

The normal way to solve a multiphysics problem in COMSOL Multiphysics is to solve for all physics simultaneously. By computing the Jacobian for the full, coupled system, this approach provides the best possible convergence. However, there may be cases where you want to solve for a separate physics or variable at the time and use the solution from the previous step as the input to the next step. Such cases include:

- Multiphysics couplings in one direction only, so that one physics depends on the other but not vice versa. It is then sufficient to solve first for the "independent" physics and then use that as input to the subsequent step, where you compute the solution for the "dependent" physics. This makes it possible to solve using less memory than if you compute the full system directly.
- Fully-coupled multiphysics applications where the couplings are not stronger than what makes it possible to achieve convergence within a few iterations. Then such an iterative approach can be of interest as a trade-off between memory requirements and solution time.
- Testing various sequential or segregated solution schemes for special applications.

You can also perform such an iterative approach in the COMSOL Multiphysics user interface by using the Solver Manager to select variables to solve for and to restart using the previous solution. With solver scripting it is even possible to automate this approach. The script that you create by recording the solution steps provides a good starting point for a script at the command line. With command-line scripting it is

possible to add plots and convergence control that automatically stops the iterations when the solution no longer changes.

In addition, the COMSOL Multiphysics user interface provides a flexible segregated solver for linear and nonlinear stationary problems and parametric problems, where you have full control over each group of variables (tolerances, selection of solvers, and so on) and the segregated solver scheme (number of iterations and damping for each step. The corresponding command-line function is `femstatic`, which provides a number of properties for specifying groups of variables and their settings (see "Using the Built-In Segregated Solver" on page 151 for an example of how to call `femstatic` to make it use the built-in segregated solver).

CREATING INITIAL VALUES

A useful function for creating initial values is `asseminit`, which you can use to compute initial values and to map a solution from one mesh to another. Examples of its use are:

- `sol = asseminit(fem)`, which computes a solution object corresponding to the initial value expressions in the FEM structure `fem`.
- `sol = asseminit(fem,'u',fem_src.sol')`, which evaluates the initial value expressions in `fem` using the solution `fem_src.sol` in the source FEM structure `fem_src`.
- `sol = asseminit(fem,'init',fem_src)`, which transfers the solution `fem_src.sol` in the source FEM structure `fem_src` to the mesh in `fem` using interpolation.

ITERATIVE SOLUTION FOR THE RESISTIVE HEATING MODEL

The following example shows how to use an iterative approach for such a coupled multiphysics problem, the Resistive Heating model from the Multiphysics section of the COMSOL Multiphysics Model Library:

This is originally a transient model of resistive heating (Joule heating), but in this case the solution is for the steady-state case. The multiphysics couplings are bidirectional:

- The electric current generates a heat source that affects the temperature.
- The temperature affects the electric conductivity, which in turn affects the electric current.

To set up the physics, open `resistive_heating.mph` in COMSOL Multiphysics and then export the FEM structure to the command line. Alternatively, save the model as

an M-file from which you then can remove the solver calls and postprocessing commands to create a script that creates the geometry and sets up the physics.

To start the iterative solver process, save the exported FEM structure, compute the initial values and, and solve once for the temperature:

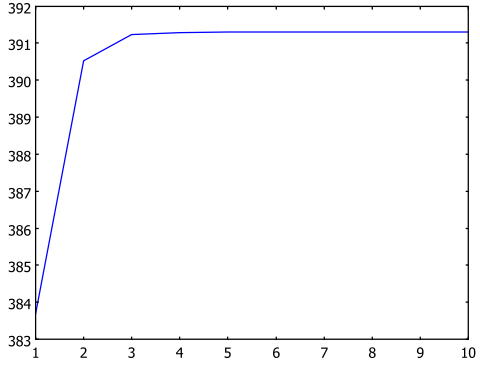
```
fem0 = fem;
init = asseminit(fem,'u',fem0.sol);
fem.sol = femstatic(fem,'init',init,...
          'solcomp',{'T'},...
          'outcomp',{'T','V'});
```

The property `solcomp` determines which variables to solve for (in this case, just the temperature, `T`), and the property `outcomp` determines which variables to provide in the output (in this case, both the temperature, `T`, and the electric potential, `V`). The following command computes the maximum temperature:

```
Tmax = postmax(fem,'T');
```

In this first step, the temperature in the solution is about 383.7 K. Proceed by iteratively solving for the electric potential and then the temperature, providing the solution from the previous step as the input to the next:

```
for i=2:10
  fem0 = fem;
  fem.sol=femstatic(fem, ...
                 'init',fem0.sol, ...
                 'solcomp',{'V'}, ...
                 'outcomp',{'T','V'});
  fem0 = fem;
  fem.sol=femstatic(fem, ...
                 'init',fem0.sol, ...
                 'solcomp',{'T'}, ...
                 'outcomp',{'T','V'});
  Tmax(i) = postmax(fem,'T');
end
plot(Tmax);
```



From the plot of the maximum temperature at each iteration you can see that the temperature quickly converges to 391.3 K, which is the same maximum temperature that you get when solving the fully coupled multiphysics problem directly.

Using the Built-In Segregated Solver

The `femstatic` solver command includes properties for the segregated solver:

- The `seggrps` property is a cell array, where each cell contains a cell array with alternating properties and values for each group. In the following call, the first cell contains the variable for temperature, `T`, and its tolerance, and the second cell contains the
- The `segorder`, `segdamp`, and `subiter` properties control the order, damping, and segregated substep iterations, respectively. See the entry on `femstatic` in the *COMSOL Multiphysics Reference Guide* for more information.

```
femO = fem;
init = asseminit(fem,'u',femO.sol);

% Solve problem
fem.sol=femstatic(fem, ...
                  'init',init, ...
                  'solcomp',{'T','V'}, ...
                  'outcomp',{'T','V'}, ...
                  'seggrps',{{'segcomp',{'T'},'ntol',1e-6},...
                  {'segcomp',{'V'},'ntol',1e-6}}, ...
                  'segorder',[1 2], ...
                  'segdamp',[0.5 0.5], ...
                  'subiter',[1 1], ...
                  'llimitdof',{}, ...
                  'llimitval',[]);
```

7

Postprocessing and Visualization

This chapter explains how to use scripting capabilities to postprocess and visualize COMSOL Multiphysics models.

Introduction to Postprocessing

Given a solution/mesh pair, a variety of tools are available for data visualization and processing:

- `posteval` and `postinterp`—evaluate *expressions* in selected points
- `postint`—integrate expressions over edges, boundaries, and subdomains
- `postplot` and `postcrossplot`—visualize expressions and variables
- `postmovie`—animate a plot

In addition, a number of shorthand commands handle various `postplot` plot types: `postarrow`, `postarrowbnd`, `postcont`, `postflow`, `postiso`, `postlin`, `postslice`, `postsurf`, and `posttet`. For example, the `postsurf` plot command used in "Example—Navier's Equations" on page 44, is shorthand for

```
postplot(fem,'tridata','u')
```

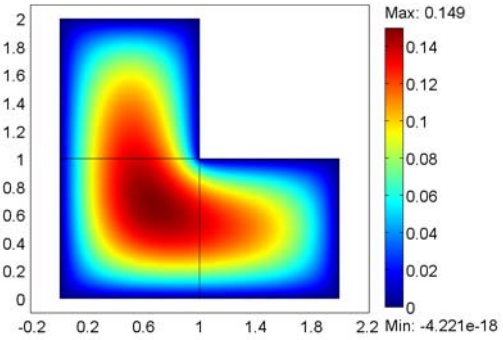


Figure 7-1: Sample surface plot in 2D.

Similarly, the command `postanim` is shorthand for certain standard animations in 1D, 2D, and 3D.

You can find details about the syntax for the various postprocessing commands, along with additional examples, by consulting their respective command-line help entries.

Overview of Postprocessing Functions

COMSOL Multiphysics provides a set of command-line functions for postprocessing and visualization, function that convert solution data from the solvers into plots or numbers.

In the following table, which lists these functions, the term *expression* refers to an expression containing any of the variables described in "Using Variables and Expressions" on page 138 of the *COMSOL Multiphysics User's Guide*.

The term *domain* refers to a geometric domain of a certain space dimension. For instance, in 3D there are subdomains (dimension 3), boundaries (dimension 2), edges (dimension 1), and vertices (dimension 0).

FUNCTION	DESCRIPTION
`meshintegrate`	Integrates an expression over an arbitrary cross-section (line or surface) intersecting the geometry
`postcoord`	Postprocessing function for getting coordinates
`posteval`	Evaluates an expression on any domain and returns the result in postdata format
`postglobaleval`	Evaluate global variables (for example, from ODEs or algebraic equations)
`postgp`	Extract Gauss points and Gauss point weights
`postint`	Integrates an expression on any domain
`postinterp`	Evaluates an expression on any domain of dimension > 0 in points specified by global coordinates (or parameter values on a geometry face or edge)
`postmax`	Compute maximum value for expression
`postmin`	Compute minimum value for expression

There is also a set of visualization functions. The general plot command is `postplot`. To conveniently plot an expression in some standard ways, COMSOL Multiphysics provides a set of shorthand commands for `postplot`:

FUNCTION	DESCRIPTION	WORKS IN
`postarrow`	Arrow plot on subdomains	2D, 3D
`postarrowbnd`	Arrow plot on boundaries	2D, 3D
`postcont`	Contour plot	2D

FUNCTION	DESCRIPTION	WORKS IN
`postflow`	Flow line plot	2D, 3D
`postiso`	Isosurface plot	3D
`postlin`	Line plot	1D, 2D, 3D
`postprinc`	Principal stress/strain plots plots in subdomains	2D, 3D
`postprincbnd`	Principal stress/strain plots plots on boundaries	2D, 3D
`postslice`	Slice plot	3D
`postsurf`	Surface plot	2D, 3D
`posttet`	Subdomain plot	3D

The `postmovie` function animates any `postplot` plots. The function `postanim` is a shorthand function for standard animations.

The function `postcrossplot` plots expressions in cross sections of the geometry. It can also plot expressions on any domain. For time-dependent problems, `postcrossplot` also provides time-extrusion plots and, letting the dimension of the cross section be zero, node plots.

The function `postglobalplot` plots global variables such as the solutions to ODEs.

All plot functions (except `postmovie` and `postanim`) provide handles to the graphics objects as an optional output argument. These handles make it possible to fine-tune the plots by working on individual graphics objects (see "Editing Plots in the COMSOL Figure Windows" on page 161).

Interpolation and Integration

Quantitative postprocessing involves computing lumped parameter values such as integrals that represent, for example, the total charge or the average temperature. It is also of interest to be able to interpolate the solution or derived quantities to an arbitrary position in the geometry or to a structured grid.

Data Evaluation and Interpolation

The main functions for data evaluation are `postinterp` and `posteval`:

- Use `postinterp` to compute values in arbitrary points in the geometry. For example, using `postinterp` you can compute values of the solution or any expression in a rectangular grid instead of the actual mesh elements. If the evaluation point is outside of the geometry, `postinterp` returns NaN (Not-A-Number), but it is also possible to use extrapolation to compute values just outside of the actual domain (using the `Ext` property).
- Use `posteval` to evaluate an expression in any domain. It is possible to evaluate expressions on any domain type: subdomains, boundaries, edges, and vertices (points) using one or several solutions. The output is *post data*, a structure with fields `p`, `t`, `q`, `d`, and `elind`. The field `p` contains node point coordinate information. The number of rows in `p` is the number of space dimensions. The field `t` contains the indices to columns in `p` of a simplex mesh (each column in `t` represents a simplex). The field `q` contains the indices to columns in `p` of a quadrilateral mesh, with each column in `q` representing a quadrilateral. The field `d` contains data values. The columns in `d` correspond to node point coordinates in columns in `p`.

EXAMPLE OF DATA EVALUATION AND INTERPOLATION

As a simple example, evaluate the solution to Poisson's equation on a unit disk with homogeneous Dirichlet boundary conditions (the solution is zero on the boundary). First set up and solve the problem:

```
clear fem
fem.geom = circ2;
fem.mesh = meshinit(fem);
fem.shape = 2;
fem.equ.c = 1;
fem.equ.f = 1;
fem.bnd.h = 1;
fem.xmesh = meshextend(fem);
```

```
fem.sol = femstatic(fem);
```

Next evaluate the solution at a point, on a grid, and on a mesh. First compute the solution u at the center of the square:

```
postinterp(fem,'u',[0;0])
```

The value at the center is 0.25, as you can determine from the analytic solution:

$$u = \frac{1 - x^2 - y^2}{4} \tag{7-1}$$

Create a grid of points in the interior of the disk, interpolate the solution to the grid points, and plot the solution on the grid using the `surf` function:

```
[x,y] = meshgrid(-0.5:0.1:0.5,-0.5:0.1:0.5);
p = [x(:)';y(:)'];
u = postinterp(fem,'u',p);
u = reshape(u,size(x));
surf(x,y,u)
```

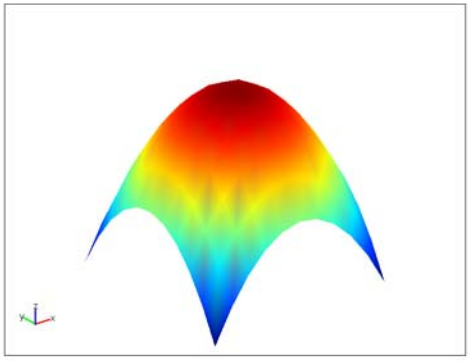


Figure 7-2: Surface height and color plot of the solution to Poisson's equation on the unit disk.

Finally create a finer mesh and interpolate the solution from the original mesh to the node points of the finer mesh:

```
fem2 = fem;
fem2.mesh = meshinit(fem2,'hmax',0.03);
fem2.xmesh = meshextend(fem2);
u = postinterp(fem,'u',fem2.mesh.p);
```

To compute the values of the solution in third-order Lagrange element points, use `posteval`:

```
pd = posteval(fem,'u','refine',3)
```

The default is to evaluate in all subdomains, so you do not have to specify the domains and element dimension in this case.

To compute values of the y-derivative of the solution in Gauss points of order four, use `posteval` together with `postgp`, which returns the Gauss points:

```
gp = postgp('tri',4);
pd = posteval(fem,'uy','spoint',gp);
```

Computing Integrated Quantities

The main function for computing integrals is `postint`, which you use to compute integrated quantities. You can integrate over subdomains, boundaries (in 2D and 3D), and edges (in 3D only).

To integrate values on a mesh, use `meshintegrate`, which is useful for computing integrals along cross sections plotted with `postcrossplot`. In that case, set the `outtype` property in `postcrossplot` to `postdata`. For more information, see the entry for `meshintegrate` in the *COMSOL Multiphysics Reference Guide*.

EXAMPLE OF INTEGRATION

Continuing with the same example (Poisson's equation on a unit disk), verify that the integral over the entire domain is equal to $\pi/8$ (or roughly 0.3927):

```
postint(fem,'u')
```

Maximum Temperature, Average Temperature, and Total Heat Flux

In a heat transfer model, the maximum temperature, the average temperature, and the total heat flux across a boundary are quantities that can be of interest. Returning to the Resistive Heating model, compute these three quantities. First export the model to the command line by first opening it and then pressing Ctrl+F (or use the **File>Export>FEM Structure as 'fem'** command).

MAXIMUM TEMPERATURE

To compute the maximum temperature, use the postmax function. By providing a second output argument, you also get the location of the maximum value:

```
[m,p]  = postmax(fem,'T')
```

The maximum temperature occurs at midpoint of the bottom boundary (at $x = 0.5$, $y = 0$), where the temperature is roughly 391 K.

AVERAGE TEMPERATURE

To find the average temperature, you need to compute two integrals: one where the integrand is the temperature, and the other where the integrand is 1 (to compute the area). Then divide the two to get the average temperature:

$$T_{\text{avg}} = \int T dA / \int dA$$

With scripting, this is the following line of code:

```
Tavg = postint(fem,'T')/postint(fem,'1')
```

which gives an average temperature of roughly 346 K.

TOTAL HEAT FLUX ACROSS A BOUNDARY

To compute the total heat flux over the right boundary, integrate the variable for the normal heat flux, `nflux_ht`, over Boundary 4:

```
flux_tot = postint(fem,'nflux_ht','edim',1,'dl',4)
```

The total normal heat flux is approximately $1.33 \cdot 10^5$ W/m.

Working With Graphics Objects

There are two ways to change the appearance of graphics objects in COMSOL Multiphysics plots:

- Using the interactive tools in the COMSOL figure windows. This makes it possible to access individual graphics objects unlike the postprocessing tools in the COMSOL Multiphysics user interface.
- Using the handles to the graphics objects and the `set` command to specify graphics properties in COMSOL Script. This makes it possible to create a number of plots interactively at the command prompt or by running a script file.

Both methods also work in MATLAB; see the MATLAB documentation for details on MATLAB graphics objects and figure windows.

Editing Plots in the COMSOL Figure Windows

When you use the plot commands in COMSOL Script, the plots appear in COMSOL figure windows.

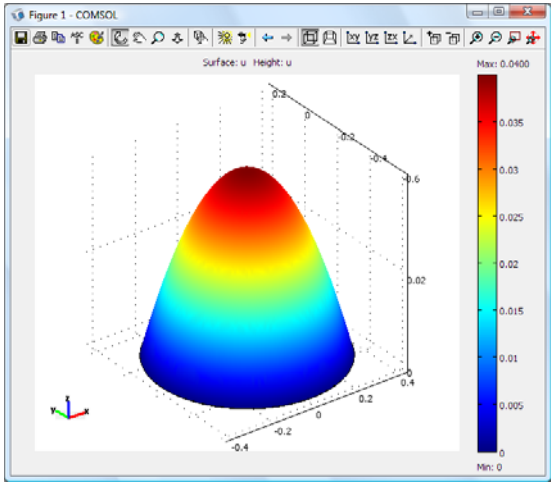


Figure 7-3: A COMSOL Multiphysics plot in a COMSOL figure window.

A figure window contains a toolbar with many visualization tools (see page 189 of the *COMSOL Script User's Guide* for a detailed description). Of particular interest is the Edit Plot tool, which you open by clicking the **Edit Plot** button (the one with the

palette icon). The **Edit Plot** dialog box provides an interactive tool for editing the graphics objects in the figure window.

EXAMPLE: CHANGING A SURFACE PLOT TO A WIREFRAME PLOT

This example solves a transient heat equation in 1D,

$$\frac{\partial u}{\partial t} - \Delta u = 1$$

with Dirichlet boundary conditions and zero initial condition:

$$u(0, t) = 0$$
$$u(1, t) = 0$$
$$u(x, 0) = 0$$

The following COMSOL Multiphysics code specifies the equation and solves it from $t = 0$ to 0.1 s in steps of 0.01 s:

```
clear fem
fem.dim = 1;
fem.geom = solid1([0 1]);
fem.mesh = meshinit(fem);
fem.equ.c = 1; fem.equ.f = 1; fem.equ.da = 1;
fem.bnd.h = 1;
fem.shape = 2;
fem.xmesh = meshextend(fem);
fem.sol = femtime(fem,'tlist',0:0.01:0.1);
```

To plot the solution, u, at the last six time steps as a time-extrusion plot use the `postcrossplot` command:

```
postcrossplot(fem,1,1,'surfdata','u','solnum',6:11)
```

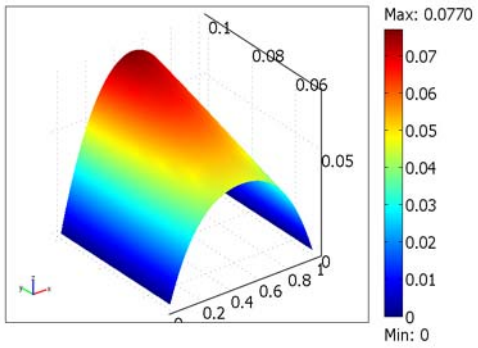


Figure 7-4: Time-extrusion surface plot.

To edit the figure so that it instead shows a wireframe plot, use the Edit Plot dialog box:

1. In the figure window toolbar click the **Edit Plot** button.
2. In the tree of graphics objects to the left select **Patch**.
3. From the **Face color** list select **None**.
4. From the **Edge color** list select **Interpolated**.
5. Click **OK** to close the dialog box and generate the new plot.

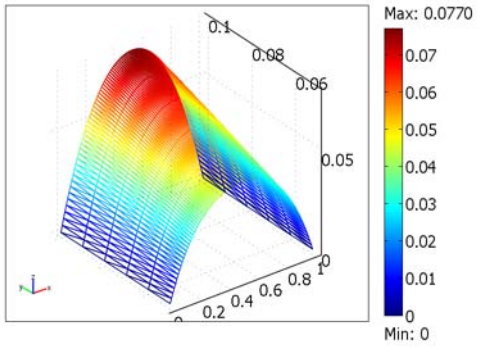


Figure 7-5: The same time-extrusion plot with patch settings for a wireframe plot.

Handles—Editing Plots By Programming

A handle is a variable that contains a numeric identifier of a graphics object. Using handles along with the `get` and `set` functions you can probe and change the properties of such objects. You can do so either in scripts or directly at the COMSOL Script command prompt to customize the visualization of results from COMSOL Multiphysics.

Returning to the example from the previous section, once again generate the time-extrusion surface plot and inspect the setting for the face color with the following commands:

```
h = postcrossplot(fem,1,1,'surfdata','u','solnum',6:11);
get(h,'facecolor')
```

The output should read

```
ans =
   interp
```

Now change the surface plot to a wireframe plot by typing

```
set(h,'facecolor','none','edgecolor','interp')
```

EXAMPLE: ADDING TRANSPARENCY AND LABELS

This example takes the `automotive_muffler` model in the COMSOL Multiphysics Model Library as a starting point for illustrating the use of handles to specify the transparency of a specific graphics object. In addition, the commands provide an enhanced plot with labels that point out the muffler's inlets and outlets.

1. From the **COMSOL Script** window go to the **File** menu and select **Start COMSOL Multiphysics** (if it is not already open). If it is already open, click the **New** button on the main toolbar in COMSOL Multiphysics.
2. In the **Model Navigator** click the **Model Library** tab. Select the model **COMSOL Multiphysics>Acoustics>automotive muffler**. Click **OK**.
3. From the **File** menu choose **Export>FEM Structure as 'fem'**.
4. Return to the **COMSOL Script** window. Type `fem` at the command prompt. You should see the following output:

```
fem =

  version:  [1x1 struct]
      fem:  {[1x1 struct] [1x1 struct]}
    const:  {'p0' '1'}
    xmesh:  [1x1 xmesh]
```

```
    sol:  [1x1 femsol]
```

In this model there are two geometries, and the results for the 3D model is in `fem.fem{1}` (the second geometry is a 2D geometry that contains no physics and served only as a basis for the 3D geometry).

Now create an attractive isosurface plot from the command line, adding text labels for the muffler's inputs and outputs. Make the muffler's boundary a transparent shell so that you can clearly see the inside surfaces:

1. Create an isosurface plot of the pressure at 490 Hz (the 40th solution vector) together with a black plot of the boundary using the following command on a single line:

   ```
   p = postplot(fem,'solnum',40,'isodata','p','isolevels',20,...
         'isomap','jet','tridata','1','tribar','off',...
         'trifacestyle','black');
   ```

   Enter `help postplot` for more information about this command.

2. The output to the `p` variable contains the handles to the objects that `postplot` created. Now use the third handle, which is the boundary plot, and set its transparency. The `set` command controls the properties of all graphical objects. Issue the command:

   ```
   set(p(3),'facealpha',0.2);
   ```

3. Use the `get` command to see the value of a certain property. You can also use `get` without any property name to see all the properties. For example, `get(p(3))` results in the following output:

   ```
   ans =

       ColorBar:  'off'
      EdgeColor:  'none'
      FaceAlpha:  [0.2000]
      FaceColor:  [0 0 0]
          Faces:  [50562x3 double]
           Hold:  'off'
         Parent:  [0.2921]
            Tag:  ''
           Type:  'patch'
       Vertices:  [56180x3 double]
        Visible:  'on'
          XData:  [3x50562 double]
          YData:  [3x50562 double]
          ZData:  [3x50562 double]
   ```

4 Using the `text` command, place three text objects at the inlets and outlet of the porous reactor. Store the handle for each created text object so that you can later change the font size. Execute these commands:

```
t1 = text(-0.2,0,0,'Inlet');
set(t1,'fontsize',14);
t2 = text(0.95,0,0.03,'Outlet');
set(t2,'fontsize',14);
```

5 Finally, click the **Headlight** toolbar button in the figure window, then rotate the axis to get a nice view. Turn off the axis and coordinate system with the following commands:

```
axis off
set(gca,'coordsys','off');
```

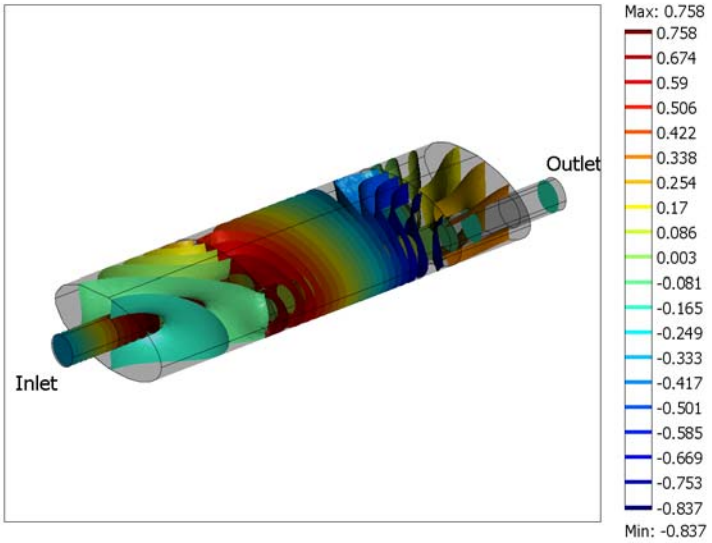


Figure 7-6: Modified plot of the automotive muffler.

Postprocessing and Associativity

This section continues the discussion "Modeling with a Parametrized Geometry" on page 95; it shows how to extend that example by using associativity for the postprocessing of a model.

It is possible to extract information on the numbering of the domains in a changing geometry, a feature that can be useful, for instance, if you want to modify the geometry in a `for` loop, and after each iteration perform an integration over one and the same domain. In the following example assume you want to evaluate an integral over the straight interior boundary, which has the index 4 before the first iteration is run. See the figure on page 96.

The model moves the circle from left to right in the geometry. After that object is moved, the model splits the interior boundary into three distinct segments. You can determine the indices of those domains by using the second output of `geomanalyze`:

```
[fem,assocmap] = geomanalyze(fem);
```

Now the variable `assocmap` contains a cell array with all the information on the domain mapping, where `assocmap{1}`, `assocmap{2}`, and `assocmap{3}` contain information about the vertices, edges, and subdomains, respectively. When the circle starts intersecting the internal subdomain boundary, the value becomes

```
assocmap{2} = [1 2 3 4 5 4 4 6 7 8 9 10 11]
```

Note that the number 4, which is the original index of the boundary that you are "tracking," is found in indices 4, 6, and 7 in the second version of `assocmap`. To find these indices in an automated way you can use the following statement:

```
integration_indices = find(assocmap{2}==4);
```

Thus you can extend the loop from the example "Modeling with a Parametrized Geometry" on page 95.

```
nSteps = 5;              % number of steps in loop
dist = .75/nSteps;       % distance to move every step
for i = 1:nSteps
  c1 = drawgetobj(fem,'C1');                    % retrieve circle
  fem = drawsetobj(fem,'C1',move(c1, dist, 0));   % add new circle
  [fem,assocmap] = geomanalyze(fem);
  fem.mesh = meshinit(fem,'report','off');
  fem = multiphysics(fem);
  fem.xmesh = meshextend(fem);
  fem.sol = femlin(fem,'report','off');
```

```
    figure, postplot(fem,'tridata','u','title',sprintf('i = %g',i))
    ind = find(assocmap{2}==4);
    postint(fem,'u','dl',ind,'edim',1)
  end
```

This computes the integral of the solution along the vertical original subdomain boundary, and the procedure works just as well when the circle intersects the subdomain boundary.

8

Simulink and State-Space Export

This chapter explains how to use the Simulink and state-space export functions in COMSOL Multiphysics to include finite element models in dynamic system simulations. For model examples of the Simulink export, see the chapter "Multidisciplinary Models" in the *COMSOL Multiphysics Model Library*.

Simulink Export

Simulink is a general software package for simulation of dynamic systems. It provides a graphical user interface for intuitive building of models, a block library with a large number of components, and the possibility to create new blocks. Furthermore, as Simulink is built on MATLAB, it has access to advanced mathematical and numerical routines as well as all other products in the MATLAB family.

It is of great interest to integrate static and time-dependent COMSOL Multiphysics models using Simulink export. This enhances COMSOL Multiphysics with functionality such as control loop validation and design, signal processing, and optimization, all easily accessible through Simulink's user interface. Many complex problems in these areas involve physical systems, and the use of COMSOL Multiphysics for the modeling of systems gives much more accurate results than empirical models.

For examples using Simulink export, see the models "Magnetic Brake" on page 302 and "Controlling Temperature" on page 329 in the *COMSOL Multiphysics Model Library*.

Dynamic or Static Export?

By default, COMSOL Multiphysics exports a dynamic model. This means that the output of the COMSOL Multiphysics Subsystem block can depend on both the input variables and the Simulink state vector. The COMSOL Multiphysics solution vector is a part of the Simulink state vector.

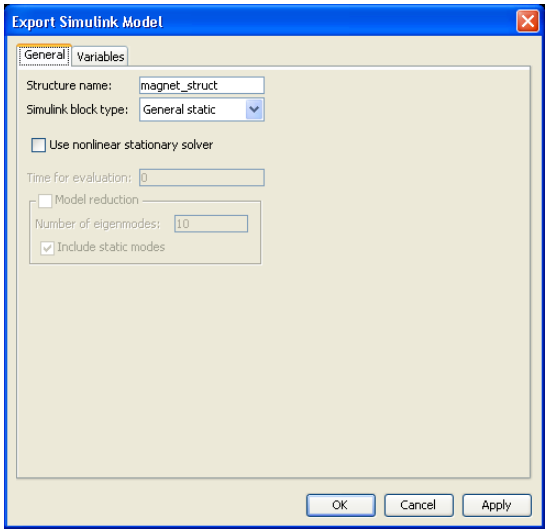
If the time scales in a time-dependent COMSOL Multiphysics model are very fast compared to the time scales in the Simulink model, a *static* model can be exported. This means that the terms of the equations that contain time derivatives are neglected, so that the output of the COMSOL Multiphysics Subsystem block can be computed from the input variables by solving a stationary PDE problem. If, for example, you use COMSOL Multiphysics to model electromagnetic phenomena as a part of a mechanical system, the dynamics of the COMSOL Multiphysics model can be neglected. The static model can be either linear or nonlinear.

General or Linearized Export?

By default, COMSOL Multiphysics uses *general export*. General export means that the COMSOL Multiphysics solvers will be called in every simulation step from Simulink. This is not necessary for linear models. To perform a simulation of a linear model or to perform a nonlinear simulation close to an equilibrium point, you can use *linearized export*. The linearized model is exported as a set of matrices, which gives faster simulations because the COMSOL Multiphysics solver does not need to be called during the simulation. It is also possible to simplify the exported model even further by using model reduction.

The Simulink Export Dialog Box

Open the **Export Simulink Model** dialog box from the **File** menu under **Export**.



Use this dialog box to create a Simulink structure in the MATLAB workspace. Simulink uses the Simulink structure to view your COMSOL Multiphysics model as an S-function. Define the variable name for the Simulink structure in the **Structure name** edit field. You need to enter this variable name in the COMSOL Multiphysics Subsystem block in Simulink.

When you have defined your Simulink model parameters, click **Apply** or **OK** to create and save the model as a Simulink structure.

The Simulink Export Types

There are four different types of export corresponding to combinations of dynamic/static and general/linearized export. Select the export type in the **Simulink export type** list. There is just one COMSOL Multiphysics Subsystem block in Simulink, but it functions differently depending on your choice of export type.

GENERAL DYNAMIC EXPORT

Perform export of a general dynamic model, where the COMSOL Multiphysics degrees of freedom are part of the Simulink state vector. The COMSOL Multiphysics solver is called several times for each time step to compute the time derivative of the state vector, with inputs from Simulink. Only linear, time-independent constraints and the elimination constraint handling method are supported.

GENERAL STATIC EXPORT

Export a general static model, where the COMSOL Multiphysics degrees of freedom are not part of the Simulink state vector. To compute the outputs of the COMSOL Multiphysics Subsystem block, the COMSOL Multiphysics linear or nonlinear stationary solver is called for each time step, with inputs from Simulink. Select the **Use nonlinear stationary solver** check box to use the nonlinear solver instead of the default linear solver.

LINEARIZED DYNAMIC EXPORT

Export a linearized dynamic model. The COMSOL Multiphysics model is linearized about an equilibrium solution, and the matrices in the state-space form are computed. The COMSOL Multiphysics degrees of freedom are part of the Simulink state vector. At each time step, the matrices in the state-space form are used to compute the time derivative of the state vector instead of calling the COMSOL Multiphysics solver. Only linear, time-independent constraints and the elimination constraint handling method are supported.

The linearization point should be an equilibrium point (stationary solution). You control it by using the settings in the **Values of variables not solved for and linearization point** area on the **Initial Value** tab of the **Solver Manager** dialog box. It can be useful to first compute a stationary solution and store that solution as the linearization point. Note that the inputs and the outputs are deviations from the equilibrium values.

You can use model reduction to approximate the linearized model with a model that has fewer degrees of freedom; select the **Model reduction** check box to enable it. The model reduction uses eigenmodes and static modes of the original PDE problem and

projects the linearized model onto the subspace of the eigenmodes. You control the number of eigenmodes using the **Number of eigenmodes** edit fields and whether or not to use the static modes using the **Include static modes** check box.

The model reduction uses all settings in the **Solver Parameters** dialog box for the eigenvalue solver. As an exception, the number of eigenvectors is taken from the **Export Simulink Model** dialog box.

When using model reduction on wave equation models that have been rewritten as a system of first-order equations (wave extension), the algorithm needs to know the names of the original (non time derivative) solution components. For structural mechanics application modes, these names are determined automatically. If you are using wave extension in another context, you need to perform the export from the MATLAB command prompt with the function `femsim` (see the *Command Reference*).

LINEARIZED STATIC EXPORT

Export a linearized static model. The COMSOL Multiphysics model is linearized about an equilibrium solution and a transfer matrix is computed. The COMSOL Multiphysics degrees of freedom are not part of the Simulink state vector. To compute the outputs of the COMSOL Multiphysics Subsystem block, the transfer matrix is used.

Relation to Solver Parameters and Solver Manager Settings

The Simulink export considers the settings from the **Solver Parameters** and **Solver Manager** dialog boxes.

For the dynamic block types, the export only considers certain solver parameters: the setting **Matrix symmetry** on the **General tab** and the settings on the **Advanced** tab in the **Solver parameters** dialog box. The time-dependent simulation in Simulink considers the settings in the **Scaling of variables** and **Manual control of assembly** areas during the simulation. Simulink itself provides the settings for the time stepping.

For the static block types, the solver type setting in COMSOL Multiphysics controls if the static solver is linear or nonlinear. All settings related to the linear or nonlinear solvers in **Solver Parameters** are considered for the static export blocks.

For the dynamic block types, COMSOL Multiphysics computes the initial value of the Simulink simulation using the settings in the **Initial value** area on the **Initial Value** tab of the **Solver Manager** dialog box. For choices that include initial value expressions,

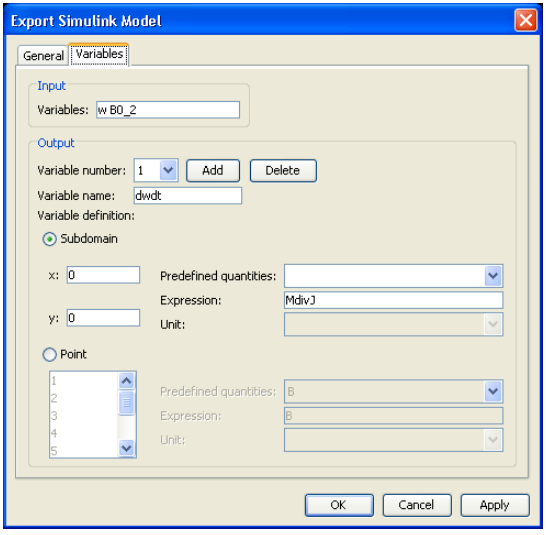
these software takes these expressions from the **Init** tab in the **Subdomain Settings**, **Boundary Settings**, **Edge Settings**, and **Point Settings** dialog boxes.

The linearized block types take the linearization points according to the settings in the **Values of variables not solved for and linearization point** area on the **Initial Value** tab of **Solver Manager** dialog box.

Also, the exported models take the settings on the **Solve for** tab into account. The variables not solved for are taken from the **Values of variables not solved for and linearization point** area on the **Initial Value** tab of the **Solver Manager** dialog box.

The Input/Output Variables

To make use of a PDE model within Simulink, it needs inputs and outputs. Use the **Variables** tab to define the inputs and outputs. The input and output variable names appear in the COMSOL Multiphysics Subsystem block as inputs and outputs, respectively. For a multigeometry model, you control what geometry the inputs and outputs belong to by opening the **Export Simulink Model** dialog box from that geometry.



CREATING INPUT VARIABLES

Enter input variables in the **Variables** edit field in the **Input** area. Use a space-separated list of variable names. Use the variables in expressions, coupling variables, or in the physics settings.

Note: For dynamic block types, the input variables must not occur in the Dirichlet boundary conditions (constraints). Avoid this problem by approximating Dirichlet boundary conditions with a Neumann boundary condition using the "stiff spring" method. See the model "Controlling Temperature" on page 329 of the *COMSOL Multiphysics Model Library*, which uses a heat flux condition with a heat transfer coefficient instead of a Dirichlet condition for the temperature.

CREATING OUTPUT VARIABLES

Define output variables in the **Output** area. Initially there are no output variables. Add an output variable by clicking **Add**. Enter the output variable name in the **Variable name** edit field. Define the value of the variable name by an expression. Click the **Subdomain** option button to evaluate the expression in any location by providing the coordinates, or click the **Point** option button to evaluate the expression in a geometry vertex (point). Then select a quantity from the **Predefined quantities** list or type in an expression in the **Expression** edit field.

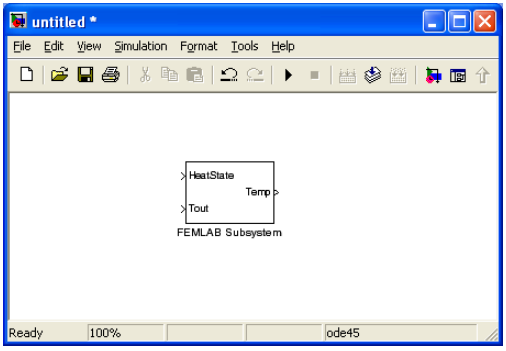
MODIFYING OR DELETING A VARIABLE

You can modify a variable by selecting it in **Variable number** list and then modifying it. Similarly, to remove a variable, select it in the **Variable number** list and then click **Delete**.

Using the COMSOL Multiphysics Model in Simulink

The result of the COMSOL Multiphysics Simulink export is a structure in the MATLAB main workspace. To use the model in Simulink, open the Blocksets & Toolboxes library in Simulink, double-click the COMSOL Multiphysics icon, and drag the **COMSOL Multiphysics Subsystem** block to your Simulink model. Double-click your

copy of the block and enter the name of the Simulink structure. This sets up the input and the output ports of the block as in the example below.



A COMSOL Multiphysics Subsystem block in Simulink with two inputs and one output.

Because COMSOL Multiphysics models usually are stiff, we recommend using an implicit stiff ODE solver like `ode15s` in Simulink's **Simulation Parameters** dialog box.

Note: Simulink does not support complex-valued inputs or outputs to the COMSOL Multiphysics Subsystem block.

State-Space Export

Use state-space export to create a linearized state-space model corresponding to your COMSOL Multiphysics model. The state-space export can create a state-space object suitable for use in the Control System Toolbox for MATLAB. You can also export the matrices of the state-space form directly to the MATLAB workspace.

The State-Space Form

The vector x is called the *state vector*. The state-space form of the model is

$$\begin{cases} \frac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

where A, B, C, and D are constant matrices. The constraints (Dirichlet boundary conditions) are eliminated by writing the solution vector U as

$$U = \text{Null}\, x + U_0$$

where U_0 is the linearization point, and Null is the null-space matrix for the constraints.

You can also export the system on the equivalent form

$$\begin{cases} M\frac{dx}{dt} = MAx + MBu \\ y = Cx + Du \end{cases}$$

This form is more suitable for large systems because the matrices M and MA usually become much more sparse than A.

If the mass matrix M is small, it is possible to approximate the dynamic state-space model with a static model, where $M = 0$:

$$\begin{cases} 0 = Ax + Bu \\ y = Cx + Du \end{cases}$$

Equivalently, $y = Hu$, where $H = D - CA^{-1}B$ is the *transfer matrix*.

Model Reduction

Finite element models often contain a large number of degrees of freedom, and the exported state-space models then become very large. ***Model reduction*** makes it possible to reduce the size of the system while keeping some of the important model properties.

First, the mode-reduction algorithm creates a matrix T whose columns consist of

- A given number of the eigenmodes with lowest eigenvalues
- The static solutions for unit values on each of the input variables

Next, it introduces state variables $\tilde{x}$ as $x = T\tilde{x}$ that projects the state-space equations onto the subspace spanned by T:

$$\begin{cases} T^T M T \dfrac{d}{dt}\tilde{x} = T^T M A T\tilde{x} + T^T M B u \\ y = CT\tilde{x} + Du \end{cases}$$

or, equivalently,

$$\begin{cases} \tilde{M}\dfrac{d}{dt}\tilde{x} = \tilde{M}\tilde{A}\tilde{x} + \tilde{M}Bu \\ y = \tilde{C}\tilde{x} + Du \end{cases}.$$

This is a new state-space form with the number of degrees of freedom equal to the number of modes in the basis T. You can then compute the solution vector by

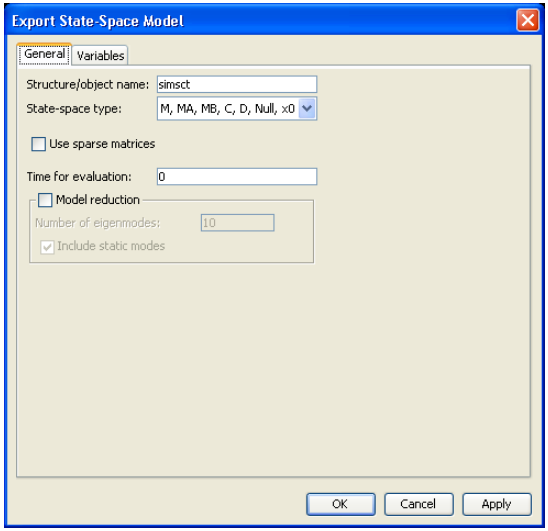
$$U = \tilde{\mathrm{Null}}\,\tilde{x} + U_0,$$

where U_0 is the linearization point.

The quality of the reduced system depends heavily on the properties of the system. Including eigenmodes with small eigenvalues ensures correct modeling of the low frequencies of the system, and including static solutions ensures correct stationary values. If the high frequencies of the system have a large influence on the outputs, however, the results may be unacceptable.

The State-Space Export Dialog Box

Open the **Export State-Space Model** dialog box from the **File** menu under **Export**.



Use this dialog box to create a structure containing the state-space form matrices or a state-space object for the MATLAB Control System toolbox in the MATLAB workspace. Define the variable name for the state-space object in the **Structure/object name** edit field. Use this variable name to refer to the result of the state-space export in MATLAB.

When you have defined the export parameters, click **Apply** or **OK** to perform the export to the MATLAB workspace.

Select the **Use sparse matrices** check box if you want to export the state-space model as sparse matrices. This keeps the sparsity of the matrices but is not recommended if you want to perform further analysis, for example, in the Control System Toolbox, because sparse matrices might not be supported.

The linearization point should be an equilibrium point (stationary solution). You control it by using the settings in the **Values of variables not solved for and linearization point** area on the **Initial Value** tab of the **Solver Manager** dialog box. The inputs and the outputs are deviations from the equilibrium values.

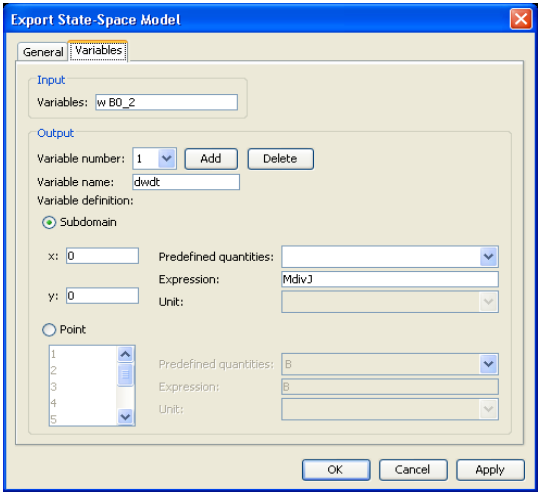
The export only considers certain solver parameters: the **Matrix symmetry** setting on the **General tab** and the settings on the **Advanced** tab in the **Solver parameters** dialog box.

You can use model reduction to approximate the linearized model with a model that has fewer degrees of freedom; select the **Model reduction** check box to enable it. The model reduction uses eigenmodes and a static mode of the original PDE problem, and the linearized model is projected onto the subspace of the eigenmodes. Use the **Number of eigenmodes** edit field to control the number of eigenmodes and the **Include static modes** check box to determine whether or not to use the static modes in the state-space model.

The model reduction uses all settings in the **Solver Parameters** dialog box for the eigenvalue solver. As an exception, the number of eigenmodes are taken from the **Export State-Space Model** dialog box.

The Input/Output Variables

The state-space model needs input and output. Use the **Variables** tab to define the input and output. For a multigeometry model, you control what geometry the inputs and outputs belong to by opening the **Export State-Space Model** dialog box from that geometry.



CREATING INPUT VARIABLES

Enter input variables in **Input**. Use a space-separated list of variable names. Use the variables in expressions, coupling variables, or in the physics settings.

CREATING OUTPUT VARIABLES

Define output variable in **Output**. Initially, there are no output variables. Click **Add** to add an output variable. Type the output variable name in the **Variable name** edit field. Click the **Subdomain** option button to evaluate the expression in any location by providing the coordinates, or click the **Point** option button to evaluate the expression in a geometry vertex (point). Then define the value of the variable name by select a quantity from the **Predefined quantities** list or by typing an expression in the **Expression** edit field

MODIFYING AND DELETING VARIABLES

You can modify a variable by selecting it in the **Variable number** list and then modifying it. Similarly, to delete a variable, select it in the **Variable number** list and then click **Delete**.

INDEX

_test 48
_time 48

1D geometry object 80
2D geometry object 79
3D geometry object 77

A
a PDE coefficient 39
adaptive mesh refinement 133
al PDE coefficient 39
algebraic equations 50
analyzed geometry 85, 93
  creating mesh from 110
  exporting 94
  importing 93
appl 56
application mode 56
  in Model M-file 67
  structure 56
application mode properties 62
application mode variable 61
application mode variables 61
application scalar variable 61
application scalar variables 61
application scalar variables section
  in Model M-file 70
application structure 56
Argyris element 21
assemble 51
assemble command 136
asseminit function 149
assigned variable names 62
average value 160

B
be PDE coefficient 39
bilinear interpolation 107
block library
  in Simulink 170
BMP 98
bnd 14
Boolean operation 89
border 47, 58
boundary coefficients 39, 60
boundary condition section
  in Model M-file 71
boundary conditions
  types of 59
boundary coupled equation variable 28
boundary coupled shape variable 28
boundary layer meshes 119
bubble element 22

C
CAD formats 92
cardioid geometry 20
clear function, vs. flclear function 65
coefficient form 39
coefficients
  defining using functions 5
coerce geometry object 80
COMSOL Multiphysics binary files 127
COMSOL Multiphysics text files 127
const field 26
constants 26
constants section
  in Model M-file 70
constr 48
constraints 38, 48, 172
contact pairs 37
contour 106
contour curves in images 99
contourc 106
Control System Toolbox 177
converting meshes 124
coordinate system 33
copying boundary meshes 122

coupling variable elements section
in Model M-file 72
cporder 52, 62
cross section 33
curve object 76
curve2 function 76
curve3 function 76
curved mesh elements 16

D da PDE coefficient 39
damping matrix 136
data evaluation 157
data structures 10
deactivated equations and boundary conditions 46
Decomposed Geometry matrix 67
degree of freedom 54
degrees of freedom 20, 50
delta function 49
dependent variables 39
difference 86
dim 39
discontinuous element 22
discretization of models 51
divergence element 22
DOF 54
domain groups 14
draw section
in Model M-file 66
dweak 48
DXF files 92
dynamic model 170, 172
dynamic systems 170

E ea PDE coefficient 39
edg 14
edge settings section 71
in Model M-file 71
eigenvalue 25
eigenvalue problems 133
eigenvalues
in solution objects 133
element
default settings 62
element types 20
equ 14
equation system form 69
equation variable 28
equations 38
equilibrium solution 172
exporting
FEM structure 7
expr 26
expression section
in Model M-file 70
expression variables 26, 61
expressions, using in coefficients 43
extended FEM structure 55, 73
extended mesh 54, 110
external mesh
importing 127
extrapolation 157
extrapolation methods 30, 31
extrude 89

F f PDE coefficient 39
face 77
face3 function 76
FEM structure 4, 10, 13
exporting 7
importing 7
loading 8
saving 8
fem.xmesh 110
femdiff function 43
field variables 25
file formats
.mphbin files 92
.mphtxt files 92

fillet function 89
finite elements 20
flclear function 65
flcontour2mesh function 106
flform function 59
flim2curve function 99, 100
flmesh2spline function 100, 106
flngdof 132
form 38
form section
in Model M-file 69
frame 15
for equation 18
for shape function 24
reference 15
frame 15
frames 15
function definition section
in Model M-file 70
functions 29
using in coefficients 5, 43

G g boundary coefficient 39
ga PDE coefficient 39
Gauss points 159
GDS format 92
general dynamic model 172
general form 43
general model 171
general static model 172
geom 14
geomcoerce 88, 89
geomcomp 87
geometric variables 18
geometry function 82
geometry methods 82
Geometry M-file 20, 67, 93
geometry model 66, 90
exporting 92
importing 92
geometry objects 67, 76, 80
exporting 90
importing 90
in Model M-files 67
geometry section
in Model M-file 67
geometry shape order 16
in Model M-file 70
geometry, mesh-based 128
geomspline 104
geomsurf 106
GIF 98
global expression section
in Model M-file 71
globalexpr 26
gporder 51, 62
grid, interpolating to 158

H h boundary coefficient 39
handles to plot objects 156
HDF 98
Hermite element 21

I ICO 98
identity pairs 37
image
filtering 100
importing 98
reducing noise 100
image 98
image contour curves 99
image processing 99
Image Processing Toolbox 99
imagesc 98
imcontour 99
importing
FEM structure 7
meshes 127
MRI data 101

imread 98
imwrite 98
including default values
in FEM structure 7
ind 15
indefinite matrices 137
independent variables 15
inheritance structure
for geometry classes 79, 80
init 53
initial value 53
initial value section
in Model M-file 72
inline functions 29
input variables 174
interior boundaries
boundary conditions on 47
interior boundaries section
in Model M-file 69
interpolation 157
interpolation functions 29
interpolation methods 29
intersection 87

J Joule heating 149
JPEG 98

L Lagrange element 21
lambda 25
level 1 coefficient 41
level 2 coefficient 41
level 4 coefficient 43
lib 33
linear algebra 137
linearization point 172, 179
linearized dynamic simulation 172
linearized model 171
linearized state-space model 177
linearized static model 173
loading an FEM structure 8
loft 99

M mass matrix 136
material variables 33
materials library section
in Model M-file 70
materials properties 34
materials/coefficients library 33
MATLAB 170, 177
maximum value 159
mesh 19, 67, 110
converting 124
copying 122
creating boundary layer 119
importing 127, 128
mesh 19
mesh cases 19
mesh element scale factor 16
mesh object 110
importing as geometry 93
mesh quality measures 120
mesh section
in Model M-file 68
mesh, interpolating to 158
mesh, using as geometry 128
meshextend 54
meshextend section
Model M-file 72
meta variables 25
M-files
in expressions 43
Model M-file 65, 71
application mode section 67
application scalar variables section 70
boundary condition section 71
constants section 70
coupling variable elements section 72
draw section 66
expression section 70

form section 69
function definition section 70
geometry section 67
geometry shape order section 70
global expression section 71
initial value section 72
interior boundaries section 69
materials library section 70
mesh section 68
meshextend section 72
multiphysics section 72
multiple geometry model 73
order of commands 65
PDE coefficient section 71
point settings section 71
postprocessing section 73
shape function section 71
simplify section 69
solver section 72
space dimension section 66
version section 65
Model M-files
geometry objects 67
Model MPH-files 8
model reduction 171, 172, 178, 180
moving mesh 15
MPH-files 8
MRI data
import of 101
importing 101
multiphysics 58
multiphysics function 56
multiphysics section
Model M-file 72
multiple geometries 55
multiple geometry model
in Model M-file 73

N NASTRAN mesh 127
native file formats 92
Navier's equations 44, 63
node point 54
noise
reducing, in point data 106
numerical quadrature 51

O ordinary differential equations 50
output variables 175

P pairs 37
PCX 98
PDE coefficient
c PDE coefficient 39
PDE coefficient section
in Model M-file 71
PDE coefficients 39, 71
PDE Toolbox 67
PDE Toolbox decomposed geometry matrix
importing 93
phase 25
piecewise functions 30
PNG 98
pnt 14
point data
for creating geometry models 106
point settings section
in Model M-file 71
point source 49
point1 function 76
point2 function 76
point3 function 76
Poisson's equation 133, 138
positive definite matrices 137
post data 157
postprocessing function 155
postprocessing section
in Model M-file 73
primitive geometry objects 76

Q q boundary coefficient 39
quadrature formula 51
quality of mesh elements 120

R r boundary coefficient 39
rational Bézier curve
object 76
reference frame 15
residual vector 143
resistive heating 149
revolve 89

S saving an FEM structure 8
sdim 15
several geometries 55
S-function 171
shape 22, 62
shape function classes 20
shape function objects 20, 22
shape function section
in Model M-file 71
shape function variables 25
sharg_2_5 21
shbub 22
shcurl 21
shdisc 22
shherm 21, 22
shlag 21
shrinking coefficients
in FEM structure 7
simplify section
in Model M-file 69
Simulink 170
Simulink export 170
dynamic model 170, 172
examples 170
general dynamic model 172
general model 171
general static model 172
input variables 174
linearized dynamic simulation 172
linearized model 171
linearized static model 173
model reduction 171, 172
output variables 175
S-function 171
solver manager 173
solver parameters 173
static model 170, 172
singular matrices 137
slit 46
solid
ellipse 87
rectangle 86
solid object 76
solid1 function 76
solid2 function 76
solid3 function 76
solsize 132
solution form 55, 69
coefficient 28
general 28
solution object 132
solver commands 132
solver manager 173
solver parameters 173
solver section
in Model M-file 72
space coordinates 15
space dimension section
in Model M-file 66
special variables 25
sshape 16, 62
sshapedim field 18
state-space export 177
model reduction 180
state-space form 172
static model 170, 172

stationary PDE problems 132
statistics, for mesh 110
stiffness matrix 137
subdomain 76
  settings 58
symmetric stiffness matrix 137

T t 25
test function 48
TIFF 98
time 25
time-dependent problems 133
transfer matrix
  in state-space model 177
trimmed surfaces 77
trimming solids 87
typographical conventions 2

U union 87, 88
units 31
units, for materials data 34

V Vanka algorithm 137
variable
  application mode 61
  application scalar 61
  expression 61
  materials 33
  weak form meta 25
variables 25
  application mode 61
  application scalar 61
  assigned name 62
  dependent 39
  expression 26, 61
  field 25
  geometric 18
  independent 15
  shape function 25
  special 25
vector element 21
vectorized functions 43
version section
  in Model M-file 65
Volterra's prey-predator equations 51

W weak 47
weak form 47
weak form, generating 59
weights
  of control polygon 88

X x space coordinate 15
xfem 55
xmesh 54
XWD 98

Y y space coordinate 15

Z z space coordinate 15