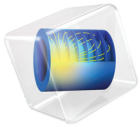


LiveLink™ *for* MATLAB®

User's Guide

6.0

COMSOL

LiveLink™ *for* MATLAB® User's Guide

Version: COMSOL 6.0

Contact Information

Visit the Contact COMSOL page at www.comsol.com/contact to submit general inquiries or search for an address and phone number. You can also visit the Worldwide Sales Offices page at www.comsol.com/contact/offices for address and contact information.

If you need to contact Support, an online request form is located at the COMSOL Access page at www.comsol.com/support/case. Other useful links include:

- Support Center: www.comsol.com/support
- Product Download: www.comsol.com/product-download
- Product Updates: www.comsol.com/support/updates
- COMSOL Blog: www.comsol.com/blogs
- Discussion Forum: www.comsol.com/forum
- Events: www.comsol.com/events
- COMSOL Video Gallery: www.comsol.com/videos
- Support Knowledge Base: www.comsol.com/support/knowledgebase

Part number: CM020008

Contents

Chapter 1: Introduction

Chapter 2: Getting Started

Chapter 3: Building Models

Chapter 4: Working With Models

Chapter 5: Calling External Functions

Chapter 6: Command Reference

1

Introduction

This guide introduces you to LiveLink™ *for* MATLAB®, which extends your COMSOL modeling environment with an interface between COMSOL Multiphysics® and MATLAB®. The *COMSOL Multiphysics Programming Reference Manual* provides additional documentation of the COMSOL API.

In this chapter:

About This Product

LiveLink™ *for* MATLAB® connects COMSOL Multiphysics to the MATLAB scripting environment. Using this functionality you can do the following:

Set Up Models from a Script

LiveLink™ *for* MATLAB® includes the COMSOL API, which has all the necessary functions and methods to implement models from scratch. For each operation done in the COMSOL Desktop there is a corresponding command that is entered at the MATLAB prompt. It is a simplified syntax based on Java® and does not require any Java knowledge. The easiest way to learn this syntax is to save the model as an M-file directly from the COMSOL Desktop. Read more about building a model using the command line in the section Building Models.

Use MATLAB Functions in Model Settings

Use LiveLink™ *for* MATLAB® to set model properties with a MATLAB function. For example, define material properties or boundary conditions as a MATLAB routine that is evaluated while the model is solved. Read more in Calling External Functions.

Leverage MATLAB Functionality for Program Flow

Use the API syntax together with MATLAB functionality to control the flow of your programs. For example, implement nested loops using `for` or `while` commands, implement conditional model settings with `if` or `switch` statements, or handle exceptions using `try` and `catch`. Some of these operations are described in Running Models in a Loop and Handling Errors and Warnings.

Analyze Results in MATLAB

The API wrapper functions included make it easy to extract data at the command line. Functions are available to access results at node points or arbitrary locations. You can also get low level information about the extended mesh, such as finite element mesh coordinates, and connection information between the elements and nodes. Extracted data are available as MATLAB variables ready to be used with any MATLAB function. See Extracting Results and Retrieving Xmesh Information.

Create Custom Interfaces for Models

Use the *MATLAB Guide* functionality to create a user-defined graphical interface that is combined with a COMSOL Multiphysics model. Make your models available for others by creating graphical user interfaces tailored to expose settings and parameters of your choice.

Connect to COMSOL Server™

LiveLink™ *for* MATLAB® has the ability to connect to COMSOL Server™ as well as COMSOL Multiphysics Server. This means that MATLAB scripts and GUIs that utilize COMSOL functionality can be distributed to and used by any user that have access to COMSOL Server™.

Help and Documentation

In this section:

Getting Help

COMSOL Multiphysics and LiveLink™ *for* MATLAB® have several sources of help and information.

THE INTRODUCTION TO LIVELINK™ FOR MATLAB®

To get started with LiveLink™, it is recommended that you read the *Introduction to LiveLink™ for MATLAB®*. It contains detailed examples about how to get you started with the product.

ONLINE DOCUMENTATION AND OTHER RESOURCES

- Read this user's guide to get detailed information about the different parts of the model object and how these are accessed from MATLAB. In the section Command Reference the function available for use with LiveLink™ *for* MATLAB® are described.
- The *COMSOL Multiphysics Programming Reference Manual* contains reference documentation that describes the methods in the model object.

You can access the online documentation directly at the MATLAB prompt using the function `mphdoc`.

To open the COMSOL Documentation Help Desk enter:

```
mphdoc
```

To open the help window for a specific entry node enter:

```
mphdoc(node)
```

where `node` is the model object node (for instance, `model.geom`).

To view the help of a specific feature of a node enter:

```
mphdoc(node,<fname>)
```

where *<fname>* is a string defining the feature name in the COMSOL API, for example, `mphdoc(model,'Rectangle')`.

It is of course also possible to use MATLAB's own help function such as

```
help mphinterp
```

and

```
doc mphinterp
```

M-FILES

You can save COMSOL Multiphysics models as M-files. Use the COMSOL Desktop to get your first model implemented using the COMSOL API.

Set up the model using the graphical user interface, then save the model as an M-file. Next go to the **File** menu and select **Save**, in the save window locate **Save as type** list and select **Model File for MATLAB (*.m)**. This generates an M-function that can be run using **COMSOL with MATLAB**.

Since version 5.3 a new syntax closer to the structure in the Model Builder is available. This new syntax includes the component node as in the example below:

```
model.component(<cTag>).geom(<geomTag>)
```

In the previous syntax to access the geometry node you need to enter:

```
model.geom(<geomTag>)
```

Both versions are fully supported, and the new syntax is used by default when saving a model in the M-file format. To save a model M-file using the old syntax, you need to change the preferences settings as described in the steps below:

1. In the COMSOL Desktop, go to the **File** menu and click **Preferences.**
2. In the Preferences window, click **Methods**. In the **Code generation settings** clear the option **Use component syntax**.
3. Click **OK**.

At the MATLAB prompt use `mphsave` to save the `model` object model in the `*.m` format as in the command below:

```
mphsave(model,<filename.m>)
```

where *<filename.m>* is the name of the file with the *.m extension.

The component syntax is not used by default.

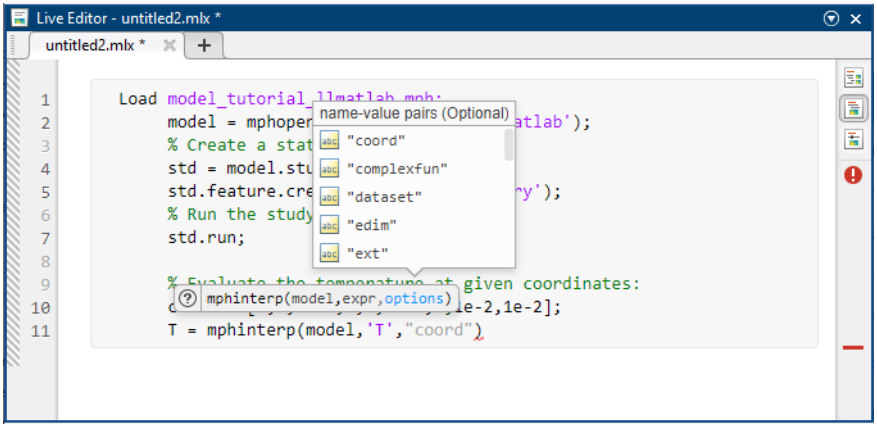
To save the model object in the *.m format using the component syntax enter:

```
mphsave(model,<filename.m>,'component','on');
```

Models created with older versions than COMSOL 5.3 cannot be saved using the component syntax unless File>Compact History is used.

AUTOCOMPLETION FOR WRAPPER FUNCTION

Autocomplemention using the Tab key is available for the wrapper functions provided by Livelink for MATLAB wherever you can enter a command: the Command Window, the regular Editor, and Live Editor for Live Scripts. The autocompletion helps you to find the wrapper function and the available property names and values where applicable. In the Command Window or a Live Editor, autocompletion provides a list of the available arguments as well as their values where applicable. For example, `'on'` or `'off'` for properties that only take these two values.

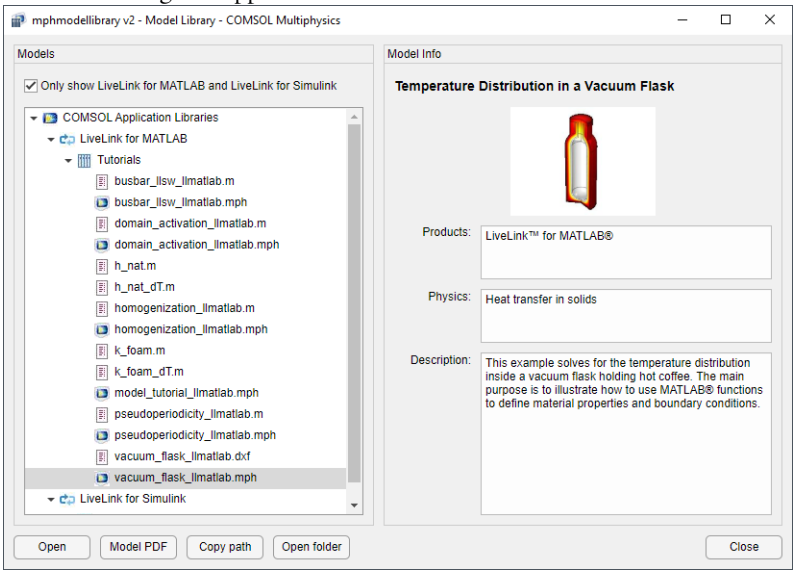


Note: Autocompletion will suggest property names and property values surrounded by double quotation marks, for example `"on"`, whereas this documentation uses single quotation marks such as in `'on'`. The wrapper functions will accept both types of strings. Wrapper functions always return strings as character arrays (that is, strings that use single quotation marks).

THE APPLICATION LIBRARIES WINDOW

Study the LiveLink™ for MATLAB® *Application Library*

LiveLink™ *for* MATLAB® includes an Application Library with detailed example models. Use the function `mphapplicationlibraries` at the command line to start a GUI for viewing the Application Libraries.



If you have installed the COMSOL apps in the MATLAB Apps ribbon, click the COMSOL Application Libraries icon ( ).

User defined library can also be listed in this window. Such a library has to be added using the COMSOL Desktop.

The following are some models that can help you get started.

Model Examples

- Learn how to activate and deactivate domains alternatively during a transient analysis. See the model *Domain Activation and Deactivation* (model name `domain_activation_llmatlab`).
- *Homogenization in a Chemical Reactor* (model name `homogenization_llmatlab`) shows how to simulate a periodic homogenization process in a space-dependent chemical reactor model. This homogenization removes concentration gradients in the reactor at a set time interval.
- *Convective Heat Transfer with Pseudo-Periodicity* (model name `pseudoperiodicity_llmatlab`) simulates convective heat transfer in a channel filled with water. To reduce memory requirements, the model is solved repeatedly on a pseudo-periodic section of the channel. Each solution corresponds to a different section, and before each solution step the temperature at the outlet boundary from the previous solution is mapped to the inlet boundary.
- *Temperature Distribution in a Vacuum Flask* (model name `vacuum_flask_llmatlab`) shows how to use the MATLAB function callback. This example solves for the temperature distribution inside a vacuum flask with hot coffee.
- *Electrical Heating of a Busbar Solved with LiveLink™* for *SOLIDWORKS®* *and LiveLink™* for *MATLAB®* (model name `busbar_llsw_llmatlab`) performs geometry optimization using COMSOL Multiphysics, MATLAB, and SOLIDWORKS®.

Where Do I Access the Documentation and the Application Libraries?

A number of internet resources have more information about COMSOL, including licensing and technical information. The electronic documentation, topic-based (or context-based) help, and the application libraries are all accessed through the COMSOL Desktop.

If you are reading the documentation as a PDF file on your computer, the blue links do not work to open an application or content referenced in a different guide. However, if you are using the Help system in COMSOL Multiphysics, these links work to other modules (as long as you have a license), application examples, and documentation sets.

THE DOCUMENTATION AND ONLINE HELP

The *COMSOL Multiphysics Reference Manual* describes all core physics interfaces and functionality included with the COMSOL Multiphysics license. This book also has instructions about how to use COMSOL Multiphysics and how to access the electronic Documentation and Help content.

Opening Topic-Based Help

The Help window is useful as it is connected to many of the features on the GUI. To learn more about a node in the Model Builder, or a window on the Desktop, click to highlight a node or window, then press F1 to open the Help window, which then displays information about that feature (or click a node in the Model Builder followed by the **Help** button (). This is called *topic-based* (or *context) help*.

Win

To open the **Help** window:

- In the **Model Builder**, **Application Builder**, or **Physics Builder** click a node or window and then press F1.
- On any toolbar (for example, **Home**, **Definitions**, or **Geometry**), hover the mouse over a button (for example, **Add Physics** or **Build All**) and then press F1.
- From the **File** menu, click **Help** ().
- In the upper-right corner of the COMSOL Desktop, click the **Help**() button.

Mac

Linux

To open the **Help** window:

- In the **Model Builder** or **Physics Builder** click a node or window and then press F1.
- On the main toolbar, click the **Help** () button.
- From the main menu, select **Help>Help**.

Opening the Documentation Window

Win

To open the **Documentation** window:

- Press Ctrl+F1.
- From the **File** menu select **Help>Documentation** ().

Mac

Linux

To open the **Documentation** window:

- Press Ctrl+F1.
- On the main toolbar, click the **Documentation** () button.
- From the main menu, select **Help>Documentation**.

THE APPLICATION LIBRARIES WINDOW

Each application includes documentation with the theoretical background and step-by-step instructions to create a model application. The applications are available in COMSOL as MPH-files that you can open for further investigation. You can use the step-by-step instructions and the actual applications as a template for your own modeling and applications. In most models, SI units are used to describe the relevant properties, parameters, and dimensions in most examples, but other unit systems are available.

Once the Application Libraries window is opened, you can search by name or browse under a module folder name. Click to view a summary of the application and its properties, including options to open it or a PDF document.

The Application Libraries Window in the *COMSOL Multiphysics Reference Manual.*

Opening the Application Libraries Window

To open the **Application Libraries** window ():

Win

- From the **Home** toolbar, **Windows** menu, click () **Applications Libraries**.
- From the **File** menu select **Application Libraries**.

To include the latest versions of model examples, from the **File>Help** menu, select () **Update COMSOL Application Library**.

Mac

Linux

Select **Application Libraries** from the main **File>** or **Windows>** menus.

To include the latest versions of model examples, from the **Help** menu select () **Update COMSOL Application Library**.

CONTACTING COMSOL BY EMAIL

For general product information, contact COMSOL at info@comsol.com.

To receive technical support from COMSOL for the COMSOL products, please contact your local COMSOL representative or send your questions to support@comsol.com. An automatic notification and case number is sent to you by email.

COMSOL WEBSITES

COMSOL website	www.comsol.com
Contact COMSOL	www.comsol.com/contact
COMSOL Access	www.comsol.com/access
Support Center	www.comsol.com/support
Product Download	www.comsol.com/product-download
Product Updates	www.comsol.com/support/updates
COMSOL Blog	www.comsol.com/blogs
Discussion Forum	www.comsol.com/community
Events	www.comsol.com/events
COMSOL Video Gallery	www.comsol.com/video
Support Knowledge Base	www.comsol.com/support/knowledgebase

2

Getting Started

In this chapter:

The Client/Server Architecture

LiveLink™ *for* MATLAB® uses the client/server mode to connect a COMSOL *server* and MATLAB.

The term COMSOL *server* designates either the COMSOL Multiphysics Server or the COMSOL Server™.

When starting COMSOL with MATLAB, two processes are started — a COMSOL Multiphysics Server and the MATLAB desktop. The COMSOL Desktop does not have to be started, but it possible to have both MATLAB and COMSOL Desktop connected to the same COMSOL Multiphysics Server at the same time.

.

COMSOL Desktop cannot connect to COMSOL Server™.

The communication between the two processes is based on a TCP /IP communication protocol. You provide login information the first time COMSOL is started with MATLAB. The login information is not related to the system's username and password. This information is stored in the user preferences file and is not required again when using COMSOL with MATLAB. The same login information can be used when exchanging the model object between the COMSOL *server* and the COMSOL Desktop.

The communication between the COMSOL *server* and MATLAB is established by default using port number 2036. If this port is in use, port number 2037 is used instead, and so on.

You can manually specify the port number. See the *COMSOL Multiphysics Installation Guide* for more information on the COMSOL *server* startup properties.

You can manually specify the port number. See the *COMSOL Multiphysics Installation Guide* for more information on the COMSOL *server* startup properties.

A connection can be local (on the same computer), which is the common case, or remote to a COMSOL *server* located on a different computer, in the later case you to connect manually MATLAB to the COMSOL *server* as described in the section Connecting a COMSOL Server and MATLAB® Manually.

Running COMSOL Models at the Command Line

The command to run COMSOL with MATLAB® automatically connects a COMSOL process with MATLAB. You can also connect the process manually. This section describes this process as well as how to change the MATLAB path in the COMSOL settings.

The System Requirements section in the *COMSOL Multiphysics Installation Guide* lists the versions of MATLAB supported by LiveLink™ *for* MATLAB®.

In this section:

Starting COMSOL® with MATLAB® on Windows ®/ Mac OSX / Linux®

To run a COMSOL Multiphysics model at the MATLAB® prompt, start COMSOL with MATLAB:

- On Windows® use the **COMSOL Multiphysics with MATLAB** shortcut icon that is created on the desktop after the automatic installation. A link is also available in the Windows start menu under **COMSOL Multiphysics 6.0>COMSOL Multiphysics 6.0 with MATLAB**.
- On Mac OS X, use the **COMSOL Multiphysics 6.0 with MATLAB** application available in the **Application** folder.
- On Linux®, enter the command `comsol mphserver matlab` at a terminal window.

See the *COMSOL Multiphysics Installation Guide* for a complete description about how to start COMSOL with MATLAB on these supported platforms.

The first time **COMSOL Multiphysics with MATLAB** is started, login and password information is requested to establish the client/server connection. The information is saved in the user preference file and is not required again.

Launching **COMSOL Multiphysics with MATLAB** enables LiveLink™ *for* Simulink® as long as it is installed on the machine.

Win

To reset the login information, add the flag `-login force` to the icon target path.

Mac

Linux

To reset the login information, enter the command `comsol mphserver matlab -login force` at a system command prompt.

RUNNING A MODEL M-FILE FROM TERMINAL PROMPT

If you want to run an M-file directly at a terminal prompt immediately after having started **COMSOL Multiphysics with MATLAB** enter the startup command as described below:

Win

```
comsolmphserver.exe matlab filename
```

Mac

Linux

```
comsol mphserver matlab filename
```

where `filename.m` is the file containing both MATLAB and COMSOL API command to be executed using COMSOL with MATLAB.

RUNNING WITHOUT DISPLAY

If you need to run COMSOL with MATLAB on a machine without support for graphics display, add the flags `nodesktop` and `mlnosplash` to the startup command as described below:

Win

```
comsolmphserver.exe matlab -nodesktop -mlnosplash
```

Mac

Linux

```
comsol mphserver matlab -nodesktop -mlnosplash
```

The above command starts MATLAB without splash screen and without the MATLAB desktop.

To avoid the splash screen on the COMSOL *server* you need to create the environment variable COMSOL_MATLAB_INIT and set it to `matlab -nosplash`.

Mac

Linux

```
COMSOL_MATLAB_INIT='matlab -nosplash'
```

Connecting a COMSOL Server and MATLAB® Manually

Manually connecting MATLAB® to a COMSOL *server* can be useful if you want to start a MATLAB standalone and then connect to a COMSOL *server*, or if you need to connect MATLAB and a COMSOL *server* running on different computers. LiveLink™ *for* MATLAB® provides the function `mphstart` to operate the client/server connection. This section contains the instruction to follow to connect MATLAB to either a COMSOL Multiphysics Server or the COMSOL Server™.

CONNECTING MATLAB TO A COMSOL MULTIPHYSICS SERVER

Starting a COMSOL Multiphysics Server

- On Windows, click **COMSOL Multiphysics Server** in the **COMSOL Launchers** folder underneath your COMSOL Multiphysics folder on the Windows Start menu.
- On Mac OS X or Linux enter `comsol mphserver` at a terminal window.

Connecting MATLAB to the COMSOL Multiphysics Server

1 In MATLAB, add the path of the COMSOL6.0/mli directory.

2 Enter this command at the MATLAB prompt:

```
mphstart
```

If the COMSOL Multiphysics Server started listening to a different port than the default one (which is 2036) use the `mphstart` function as in the command below:

```
mphstart(<portnumber>)
```

where `<portnumber>` is an integer corresponding to the port used by the COMSOL *server*.

CONNECTING MATLAB AND A SERVER ON DIFFERENT COMPUTERS

Connecting MATLAB and a COMSOL Multiphysics Server requires a Floating Network License (FNL).

To connect MATLAB and a COMSOL *server* that are running on different computers, specify the IP address of the computer where the COMSOL *server* is running in the function `mphstart`:

```
mphstart(<ipaddress>, <portnumber>)
```

`<ipaddress>` can also be defined with the COMSOL *server* domain name.

The command above assume that the same user login information are set on the server and client machine. In case the login information are not accessible from the client machine, specify manually the user name and password to the COMSOL *server* with the command:

```
mphstart(<ipaddress>, <portnumber>, <username>, <password>)
```

If the COMSOL Multiphysics installation folder cannot be found automatically, you can specify its location manually as in the command below:

```
mphstart(<ipaddress>, <portnumber>, <comsolpath>)
```

where *<comsolpath>* is the path of the COMSOL installation folder.

You can also specify all the information to connect a COMSOL *server* within the same command, use the following command:

```
mphstart(<ipaddress>, <portnumber>, <comsolpath>, ...
   <username>, <password>)
```

MEMORY SETTINGS

To be able to manipulate the model object and extract data at the MATLAB prompt, you may need to modify the Java® heap size in MATLAB. See Improving Performance for Large Models.

IMPORTING THE COMSOL CLASS

Once MATLAB and the COMSOL *server* are manually connected, import the COMSOL class by entering the following command at the MATLAB prompt:

```
import com.comsol.model.util.*
```

Disconnecting MATLAB and the COMSOL server

To disconnect MATLAB and the COMSOL *server*, run this command at the MATLAB prompt:

```
ModelUtil.disconnect;
```

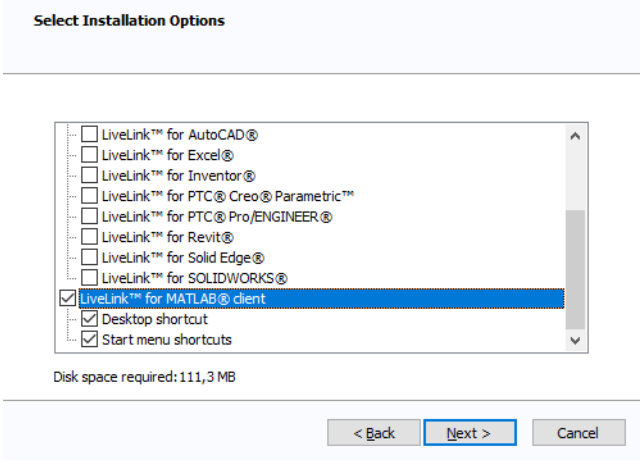
Connecting to COMSOL Server™

When using a COMSOL Multiphysics installation with LiveLink for MATLAB the connected is made between MATLAB and a COMSOL Multiphysics Server. It is also possible to connect MATLAB to a COMSOL Server™ if a COMSOL Server™ is available with a LiveLink for MATLAB License.

When using LiveLink for MATLAB with COMSOL Server™ this way MATLAB is installed on the local computer where the user is executing commands and COMSOL Server™ will most often be on another computer that is handled by an IT-department.

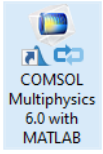
In order to be able to connect from MATLAB to a COMSOL Server™ without having COMSOL Multiphysics installed some files must be installed on the computer where MATLAB is installed. This is handled by using the COMSOL Server™ Client installer.

When installing the COMSOL Server™ Client it is important to select LiveLink™ for MATLAB© in the installation window.

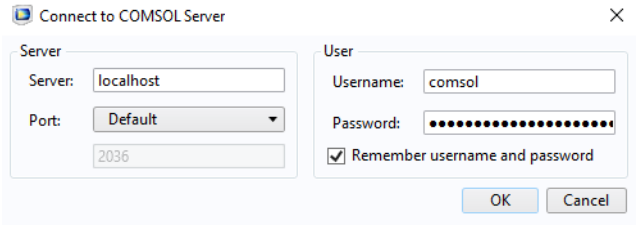


It is recommended to install Desktop and Start menu shortcuts. If these are not installed, the only way to create a link between MATLAB and COMSOL Server™ is to use the `mphstart` command as previously described for use with COMSOL Multiphysics Server.

After the installation a shortcut is available.



When clicking on this shortcut MATLAB will start and a dialog box appears that makes it possible to connect to the COMSOL Server™ that is already assumed running.



Fill in the missing information and click **OK** to connect.

After a connection has been made the LiveLink work in the same way as it does when connected to a COMSOL Multiphysics Server with these exceptions:

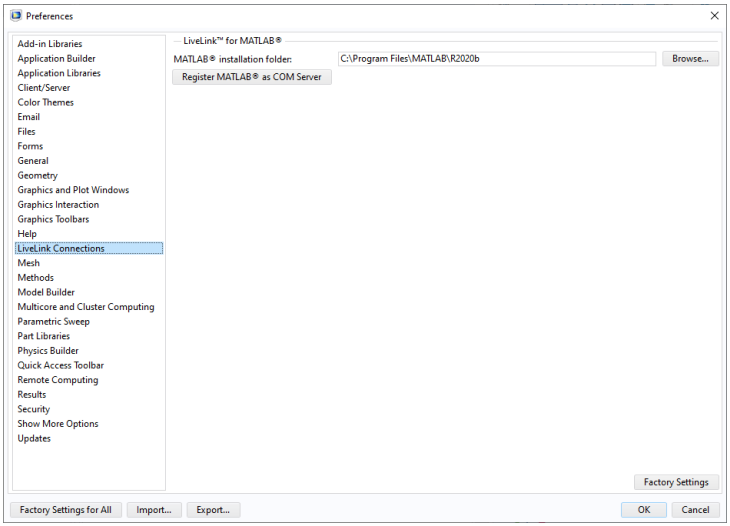
- Some graphical user interfaces that are included in a regular COMSOL Multiphysics license do not work
- A few utility functions that are mainly used when developing new models are not supported (e.g. `mphlaunch`, `mphmodel`, etc.)
- Plot on the server is not supported, use `mphplot` to plot using a local MATLAB figure.

Changing the MATLAB® Version

The path of the MATLAB® version connected to COMSOL Multiphysics is defined during the initial COMSOL installation. The MATLAB root path can be changed using the **Preferences** dialog box:

1. From the **File** (Windows users) or **Options** menu (Mac and Linux users), select **Preferences** ().
2. In the **Preferences** dialog box, click **LiveLink Connections**.
3. Set the MATLAB root directory path in the **MATLAB® installation folder** field.
4. Windows OS users also need to click the **Register MATLAB® as COM Server** button; otherwise, the specified MATLAB version does not start when calling external MATLAB functions from the COMSOL model.

5 Click **OK**.



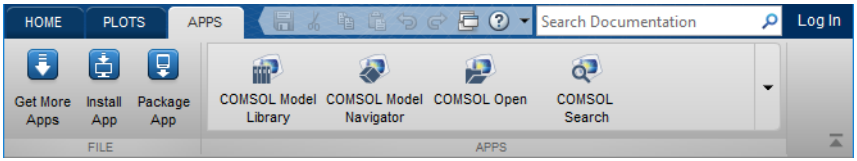
6 To update the preferences file, close and reopen the COMSOL Desktop.

Mac

On Mac OS X, select the **COMSOL with MATLAB** application available in the **Application** folder. The correct path includes the `.app` extension.

The COMSOL Apps

Install apps in the MATLAB desktop for an easy access to COMSOL information and navigation functions that use Graphical User Interfaces.



In the section the term COMSOL apps designates COMSOL wrapper function using a Graphical User Interface. It does not refer to application created using the COMSOL Application Builder.

Installing Apps in the MATLAB Apps Ribbon

The automatic COMSOL installation does not include installation of the COMSOL apps in the MATLAB Apps Ribbon. To install the apps follow the steps below:

1. In the MATLAB Desktop, go to the **Apps** Tab and select **Install App**.
2. Browse to the COMSOL Installation directory and go to the folder: COMSOL60/Multiphysics/mli/toolbox
3. Change File name extension to **All Files (*.*)** and select the file LiveLink for MATLAB.mltbx
4. Click **Open**. This opens the Install LiveLink for MATLAB window.
5. The installed apps are now listed in the Add-On Manager window.

Removing Apps in the MATLAB Apps Ribbon.

To remove Apps from the MATLAB Apps Ribbon, right-click on the apps icon and select Uninstall. The operation is individual for each apps and need to be repeated for every apps to be removed.

The COMSOL Apps

The available COMSOL apps that can be installed in the MATLAB Apps ribbon are listed below:

- COMSOL Model Library, opens a GUI for viewing the Model Library, see also The Application Libraries Window.
- COMSOL Model Navigator, opens a GUI for viewing the COMSOL model object defined as `model` in MATLAB. You can get more information in the section Navigating the Model Object Using a GUI.
- COMSOL Open, opens a GUI for opening recent files, see also Loading a Model from a List of Existing files.
- COMSOL Search, opens a GUI for searching for expressions in the COMSOL model object defined as `model` in MATLAB. See also Finding Model Expressions.

To run these apps you need a connection between MATLAB and a COMSOL *server*, either using COMSOL with MATLAB or using manual connection.

Calling External Functions From Within the Model

Use LiveLink™ *for* MATLAB® to call MATLAB functions from within the model — for instance, when working in the COMSOL Desktop. The procedure is different than implementing a model using a script as you do not need to run COMSOL with MATLAB.

Start COMSOL Multiphysics as a standalone application. The external MATLAB function needs to be defined in the COMSOL model so that a MATLAB process can automatically start when the function needs to be evaluated. The result of the function evaluation in MATLAB is then sent back to the COMSOL environment.

Calling External Functions

To run a MATLAB function, enable **Allow external MATLAB® functions** in the Preferences window; see Allowing External MATLAB Functions for more information.

3

Building Models

This chapter gives an overview of the model object and provides an introduction to building models using the LiveLink™ interface.

In this chapter:

The Model Object

While working with the LiveLink™ interface in MATLAB® you work with models through the *model object*. Use *methods* to create, modify, and access models.

In this section:

Detailed documentation about model object methods is in About General Commands in the *COMSOL Multiphysics Programming Reference Manual.*

Important Notes About the Model Object

Consider the following information regarding the model object:

- All algorithms and data structures for the model are integrated in the model object.
- The model object is used by the COMSOL Desktop to represent your model. This means that the model object and the COMSOL Desktop behavior are virtually identical.
- The model object includes methods to set up and run *sequences of operations* to create geometry, meshes, and to solve your model.

LiveLink™ *for* MATLAB® includes the COMSOL API, which is a programming interface based on Java®. In addition, the product includes a number of M-file utility functions that wrap API functionality for greater ease of use.

The Model Object in the *COMSOL Multiphysics Programming Reference Manual.*

The Model Object Methods

The model object has a large number of methods. The methods are structured in a tree-like way, very similar to the nodes in the model tree in the *Model Builder* window on the COMSOL Desktop. The top-level methods just return references that support further methods. At a certain level the methods perform actions, such as adding data to the model object, performing computations, or returning data.

Detailed documentation about model object methods is in About General Commands in the *COMSOL Multiphysics Programming Reference Manual*.

The General Utility Functionality

The model object utility methods are available with the *ModelUtil* object. These methods can be used, for example, to create or remove a new model object, but also to enable the progress bar or list the model object available in the COMSOL *server*.

MANAGING THE COMSOL MODEL OBJECT

Use the method `ModelUtil.create` to create a new model object in the COMSOL *server*:

```
model = ModelUtil.create(<ModelTag>);
```

This command creates a model object `Model` on the COMSOL *server* and a MATLAB object `model` that is linked to the model object *<ModelTag>* in the COMSOL *server*.

It is possible to have several model objects on the COMSOL *server*, each with a different name. To access each model object requires different MATLAB variables linked to them and each MATLAB variable must have a different name.

Create a MATLAB variable linked to an existing model object with the function `ModelUtil.model`. For example, to create a MATLAB variable `model` that is linked to the existing model object *<ModelTag>* on the COMSOL *server*, enter the command:

```
model = ModelUtil.model(<ModelTag>);
```

Alternatively you can use the function `mphload` as in the command below:

```
model = mphload(<ModelTag>);
```

To remove a specific model object use the method `ModelUtil.remove`. For example, to remove the model object *<ModelTag>* from the COMSOL *server* enter the command:

```
ModelUtil.remove(<ModelTag>);
```

Alternatively remove all the COMSOL objects stored in the COMSOL *server* with the command:

```
ModelUtil.clear
```

List the names of the model objects available on the COMSOL *server* with the command:

```
mphtags -show
```

ACTIVATING THE PROGRESS BAR

By default no progress information is displayed while running COMSOL with MATLAB. To manually enable a progress bar and visualize the progress of operations (such as loading a model, creating a mesh, assembling matrices, or computing the solution), enter the command:

```
ModelUtil.showProgress(true)
```

To deactivate the progress bar enter:

```
ModelUtil.showProgress(false)
```

Mac OS X does not support the progress bar.

The Model History

The model contains its entire modeling history corresponding to every settings added once to the model. When you save a model as an M-file, you get all the operations performed to the model, including settings that are no longer part of the model.

Using the model history is a convenient way to learn the COMSOL API. The latest settings enter in the command Desktop being listed at the end of the M-file.

The model history is automatically enabled when the model is created in the COMSOL Desktop. It is however possible to manually disable the model history recording from the MATLAB prompt with the command:

```
model.hist.disable
```

The functions `mphload` and `mphopen` automatically disable model history when loading a model.

To enable the model history, enter the command:

```
model.hist.enable
```

COMPACTING THE MODEL HISTORY

To clean the M-file for the model so that it contains only the settings that are part of the current model you need to compact the model history before saving the model as an M-file.

To compact the model history in the COMSOL Desktop, from **File** menu (Windows users) or from the toolbar (Mac and Linux users), select **Compact History** ().

To compact the model history at the MATLAB prompt enter the command:

```
model.resetHist
```

Loading and Saving a Model

LOADING A MODEL AT THE MATLAB PROMPT

To load an existing model saved as an MPH-file use the function `mphopen`. To load the model with the name <filename> enter:

```
model = mphopen(<filename>)
```

where `<filename>` is a string. This creates a model object `Model` on the COMSOL *server* that is accessible using the MATLAB variable `model`.

A shorter form is to simply use

```
mphopen <filename>
```

that will load the model with the given filename and use the variable name `model` for accessing the model later. Any existing variable `model` will be overwritten without warning.

The function `mphload` can also be used with the same property. In the following documentation the commands also work with `mphload`.

`mphload` does not store the filename in the recent file list as `mphopen` does by default.

Once the model is loaded, the file name and its associated model object tag are displayed in the COMSOL *server* window.

If there is already a model object `Model` in the COMSOL *server*, `mphopen` overrides the existing model object unless the model is also open in a COMSOL Multiphysics Client. In the later case, an index number is appended to the new model object name, for instance `Model1`.

`mphopen` and `mphload` do not look for lock file when opening a model in the COMSOL *server*.

If you want to manually specify the model object in the COMSOL *server*. use the command:

```
model = mphopen(<filename>, <ModelTag>);
```

where *<ModelTag>* is a string defining the tag that defines the loaded model in the COMSOL *server*.

When using the function `mphopen`, the model history is automatically disabled to prevent large history information when running a model in a loop. To turn model history on, use the function `mphopen`:

```
model = mphopen(<filename>,'-history');
```

The history recording can be useful when using the COMSOL Desktop. All the operations are then stored in the saved M-file.

If you do not want to update the recent opened file list with the model you are about to open, use the `-nostore` flag with the function `mphopen` as in the command below:

```
model = mphopen(<filename>, <ModelTag>, '-nostore')
```

If the model mph-file is protected using a password, use `mphload` as in the command below:

```
model = mphopen(<filename>, <ModelTag>, <password>)
```

where *<password>* is a string defining the password protecting the file.

If you want to get the full filename of the loaded file, add a second output as in the command below:

```
[model, filenameloaded]= mphopen(<filename>, ...)
```

LOADING A MODEL FROM A LIST OF EXISTING FILES

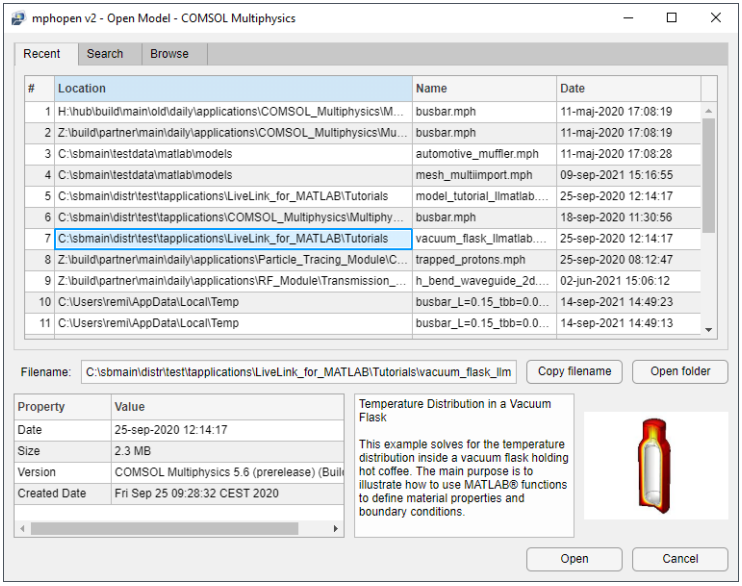
You can use a GUI where to load the model from a list files corresponding to the recent opened file or the files in a specified directory.

At the MATLAB prompt enter the command:

```
mphopen
```

If you have installed the COMSOL apps in the MATLAB Apps ribbon, click the COMSOL Open icon ().

This starts a GUI with a list of the recent opened files.



For each selected files, the model information is available in the **File Info** section.

Click the **Recent** button to get the list of the recent opened file. Click the **Search** button to search for a file using file pattern. Click the **Browse** button to browse the directory where to get filename list.

To clear the recent opened file list enter the command:

```
mphopen -clear
```

To open the GUI with the list of files in a specific directory (*<dirpath>*), enter the command:

```
mphopen -dir <dirpath>
```

LOADING A MODEL FROM A DATABASE

To open a model from a database at the command line use mphopen as in the command below:

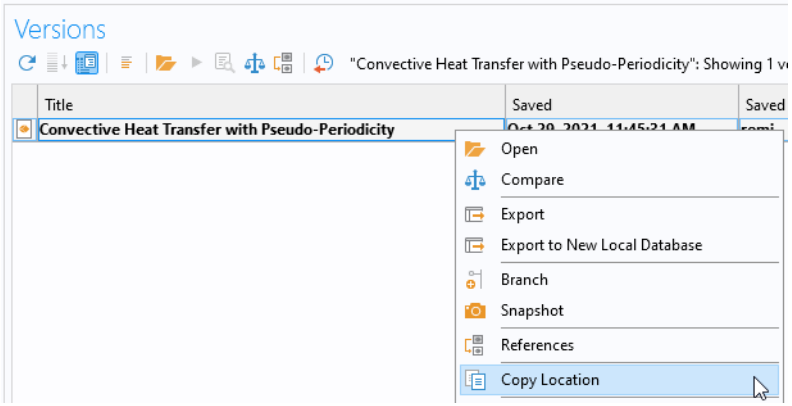
```
mphopen '<location>'
```

where *<location>* is the location in the database of the model you want to load.

The location of a model can only be obtained from the COMSOL Desktop. Follow the step below to proceed:

1 In the Model Manager right-click on the model and select Versions.

2 In the Versions section right-click the desired versions and select Copy Location.



3 Now the location is in your clipboard, make sure you paste it in the mphopen command within quotes (single ' or double ") so that the location is interpreted as a string.

SAVING A MODEL

Use the function `mphsave` to save the model object linked to the MATLAB object model:

```
mphsave(model,<filename>)
```

where *<filename>* is a string. If the filename specified *<filename>* does not provide a path, the file is saved relative to the current MATLAB path. The file extension determines the format to use (*.mph, *.m, *.java, or *.vba).

When saving the model as an M-file `mphsave` does not automatically use the component syntax to save model using the COMSOL API, to enable this syntax run the command below:

```
mphsave(model,<filename>,'component','on')
```

To save the model as a copy set the property `copy` to `on` as in the line below:

```
mphsave(model,<filename>), 'copy', 'on');
```

After saving a copy, the model does not remember where the copy was saved. Instead it remembers its previous save location.

When saving a COMSOL files (MPH-files) you can choose the file to be optimized for speed (using uncompressed files that are faster to save), or to be optimized for file size (using compressed files). To do so set the property `optimize` to `speed`, or `size` respectively as in the command below:

```
mphsave(model,<filename>), 'optimize', 'size');
```

To save a clean model, i.e. without built, computed, and plotted data set the property `excludedata` to `on`:

```
mphsave(model,<filename>), 'excludedata', 'on');
```

The models are not automatically saved between MATLAB sessions.

SET A MODEL THUMBNAIL

Before saving your model, you may want to include a model thumbnail to quickly identify your model in your own Application Library or when using `mphopen`. To set the model thumbnail enter the command:

```
mphthumbnail(model,<filename>)
```

where *<filename>* is the image file name.

You can also use a MATLAB figure to set the thumbnail. The following command will set the thumbnail to the image of the current figure:

```
mphthumbnail(model,gcf)
```

Note that the thumbnail is stored in memory. In order to save the thumbnail in the model file the model must be saved.

You can extract the image and image filename for the thumbnail stored in model, enter the command:

```
[image, imagefilename] = mphthumbnail(model)
```

Example

The code below shows how to get the model thumbnail as MATLAB image data, show the image in a MATLAB figure and store the new image as thumbnail in the model.

```
mphopen model_tutorial_llmatlab
im = mphthumbnail(model);
imshow(im)
mphthumbnail(model, gcf)
```

Sharing the Model Between the COMSOL Desktop® and the MATLAB® Prompt

It is possible to connect a COMSOL Desktop to the COMSOL Multiphysics Server that is already connected with MATLAB and then access the model from both client (the COMSOL Desktop and MATLAB). The change performed from either client are directly accessible from the other one; for instance, type a command at the MATLAB prompt and see the resulting modification in the Model Builder window, or extract data at the MATLAB prompt from a model set up in the COMSOL Desktop.

CONNECT THE COMSOL DESKTOP TO THE COMSOL MULTIPHYSICS SERVER FROM THE PROMPT

At the prompt call `mphlaunch` to start a COMSOL Desktop, connect it to the same COMSOL Multiphysics Server to which MATLAB is connected to, and import a model.

Run the command below:

```
mphlaunch
```

This starts a new COMSOL Desktop, connect it to the COMSOL Multiphysics Server that is already connected with MATLAB, and import the model available in the server. In case several model are available in the server you can specify which one to import by running the command below:

```
mphlaunch ModelTag
```

where `ModelTag` is the tag of the model to import.

List the tags of the application loaded in the server with the command `mphtags`.

You can also specify the MATLAB object name that is link to the application to be imported in the COMSOL Desktop, enter the command:

```
mphlaunch(model)
```

If a COMSOL Multiphysics client is already connected to the COMSOL Multiphysics Server you will be asked to disconnect the connected client and connect the new one or cancel the operation.

`mphlaunch` sets automatically a timeout to make MATLAB wait 0.5 second until the COMSOL *server* is free again. If you need to increase the timeout run the command below:

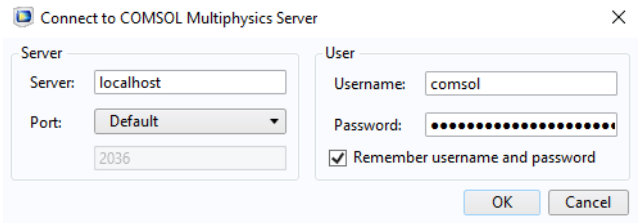
```
mphlaunch(model,<timeout>)
```

where `<timeout>` is the time in milliseconds to wait for the server to be free again.

CONNECT THE COMSOL DESKTOP TO THE COMSOL MULTIPHYSICS SERVER

Connect the COMSOL Desktop to a COMSOL Multiphysics Server manually using the **Connect to Server** dialog box:

1. From the **File** (Windows users) or **Options** menu (Mac and Linux users), select **COMSOL Multiphysics Server>Connect to Server** ( ).
2. In the **Connect to Server** window, you specify the Server configuration and the user settings. In the **Server** section enter the COMSOL Multiphysics Server name (the default name is `localhost`) and the **Port** number (the default is `2036`). This number corresponds to the port that the COMSOL Multiphysics Server is listening to, the number is displayed at the COMSOL Multiphysics Server window.
3. In the **User** section enter a **Username** and a **Password** (if they are empty); these are defined the first time you are connected to the COMSOL Multiphysics Server.
4. Click **OK**.



The first time you connect the COMSOL Desktop to the COMSOL Multiphysics Server no model is loaded to the GUI. See Import An application from the COMSOL Multiphysics Server to the COMSOL Desktop to know how connect the GUI to a model loaded in the COMSOL Multiphysics Server.

IMPORT AN APPLICATION FROM THE COMSOL MULTIPHYSICS SERVER TO THE COMSOL DESKTOP

Once you have the COMSOL Desktop connected to the COMSOL Multiphysics Server you can import the model in the GUI:

1. From the **File** (Windows users) or **Options** menu (Mac and Linux users), select **COMSOL Multiphysics Server>Import Application from Server** ().
2. In the **Import Application from Server** window, specify the application you want to import.

IMPORT A MODEL FROM THE COMSOL MULTIPHYSICS SERVER TO MATLAB

To access a model stored in the COMSOL Multiphysics Server from the MATLAB prompt enter the command:

```
model = mphload(<ModelTag>)
```

where `model` is the variable in MATLAB used to access the model stored on the COMSOL *server* and *<ModelTag>* is the tag of the COMSOL Model.

You can get the list of the models stored in the COMSOL Multiphysics Server with the command:

```
mphtags -show
```

Set up a time-out in MATLAB

To prevent MATLAB sending command to the COMSOL Multiphysics Server while it is busy to update the COMSOL Desktop, you need to set up a time-out in MATLAB and specify how long to wait the COMSOL Multiphysics Server to be free again. Enter the command:

```
ModelUtil.setServerBusyHandler(ServerBusyHandler(<timeout>))
```

Where *<timeout>* is the time in millisecond to wait the server to be free again.

Working with Geometries

This section describes how to set up and run a geometry sequence. In this section:

- The Geometry Sequence Syntax
- Displaying the Geometry
- Working with Geometry Sequences
- Retrieving Geometry Information
- Modeling with a Parameterized Geometry
- Images and Interpolation Data
- Measuring Entities in Geometry

- Geometry Modeling and CAD Tools in the *COMSOL Multiphysics Reference Manual*
- Geometry in the *COMSOL Multiphysics Programming Reference Manual*

The Geometry Sequence Syntax

In the *COMSOL Multiphysics Programming Reference Manual*:

- For a list of geometry operations, see About Geometry Commands.
- For a property list available for the geometry features see Geometry.

Create a geometry node using the syntax:

```
model.component(<ctag>).geom.create(<geomtag>, sdim)
```

where *<geomtag>* is a string used to refer to the geometry and *<ctag>* is the string defined when the component is created. The integer *sdim* specifies the space dimension of the geometry and it can be either `0`, `1`, `2`, or `3`.

To add an operation to a geometry sequence, use the syntax:

```
geometry.feature.create(<ftag>, operation)
```

where `geometry` is a link to the geometry node. The string *<ftag>* is used to refer to the operation.

To set the feature property with different values than the default, use the `set` method:

```
geometry.feature(<ftag>).set(property, <value>)
```

where *<ftag>* is the string defined when creating the operation.

To build the geometry sequence, enter:

```
geometry.run
```

Alternatively, to build the geometry sequence up to and including a given feature *ftag* enter:

```
geometry.run(<ftag>)
```

Displaying the Geometry

Use the function `mphgeom` to display the geometry in a MATLAB figure:

```
mphgeom(model)
```

To specify the geometry to display, enter:

```
mphgeom(model, <geomtag>)
```

where *<geomtag>* is the tag of the geometry node to display. If the model only contains a single geometry the tag *<geomtag>* can be left empty. When specifying a property the geometry tag is required.

Adding a `view` property will add some view settings from the COMSOL model such as axes labels (units) and grid and supports hiding of geometric entities. Usually it is sufficient to use the `auto` value for the view property:

```
mphgeom(model, <geomtag>, 'view', 'auto')
```

When running `mphgeom` the geometry node is automatically built. Set the `build` property to specify how the geometry node is supposed to be built before displaying it. Enter:

```
mphgeom(model, <geomtag>, 'build', build)
```

where *build* is a string with the value: `'off'`, `'current'`, or the geometry feature tag *<ftag>*, which, respectively, does not build the geometry (`off`), builds the geometry up to the current feature (current), or builds the geometry up to the specified geometry feature node (*ftag*).

If the geometry contains workplane, you can plot the geometry entities inside a specified workplane as a 2D geometry. Enter:

```
mphgeom(model, <geomtag>, 'workplane', <wptag>)
```

where *<wptag>* is the tag of the workplane to use. It is also possible to combine the workplane geometry display with the `build` property to display the geometry built up to a certain feature.

Use the `parent` property to specify the axes handle where to display the plot:

```
mphgeom(model, <geomtag>, 'parent', <axes>)
```

The following properties are also available to specify the vertex, edge, or face rendering:

- `vertexmode`
- `edgemode`
- `facemode`
- `vertexlabels`
- `edgelabels`
- `facelabels`
- `domainlabels`
- `edgecolor`
- `facecolor`
- `vertexlabelscolor`
- `edgelabelscolor`
- `facelabelscolor`
- `domainlabelscolor`
- `facealpha`

Use `mphgeom` to display a specified geometric entity. To set the geometric entity, enter the `entity` property and set the geometric entity index in the `selection` property to:

```
mphgeom(model, <geomtag>, 'entity', entity, 'selection', <idx>)
```

where *entity* can be either `'point'`, `'edge'`, `'boundary'`, or `'domain'`, and *<idx>* is a positive integer array that contains the list of the geometric entity indices.

You can get the handle of the plotted geometry entities with the command:

```
h = mphgeom(model, <geomtag>, ...)
```

where `h` is a Patch array of the plotted entities.

Working with Geometry Sequences

This section shows how to create geometry sequences using the syntax outlined in The Geometry Sequence Syntax. This section has these examples:

- Creating a 1D Geometry
- Creating a 2D Geometry Using Primitive Geometry Objects

- Creating a 2D Geometry Using Boundary Modeling
- Creating a 3D Geometry Using Solid Modeling

For more information about geometry modeling, see the Geometry chapter in the *COMSOL Multiphysics Reference Manual*.

CREATING A 1D GEOMETRY

From the MATLAB command prompt, create a 1D geometry model by adding a geometry sequence and then adding geometry features. The last step is to run the sequence using the `run` method.

First create a model object:

```
model = ModelUtil.create('Model');
```

Then continue with the commands:

```
model.component.create('comp1',true);

geom1 = model.component('comp1').geom.create('geom1',1);

i1 = geom1.feature.create('i1','Interval');
i1.set('intervals','many');
i1.set('p','0,1,2');

geom1.run;
```

This creates a geometry sequence with a 1D solid object consisting of vertices at $x = 0$, 1, and 2, and edges joining the vertices adjacent in the coordinate list.

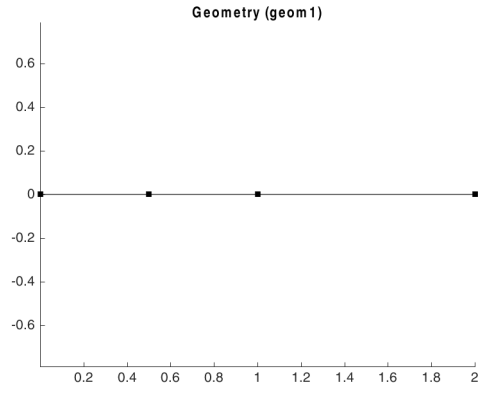
Then enter:

```
p1 = geom1.feature.create('p1','Point');
p1.set('p',0.5);

geom1.run;
```

to add a point object located at $x = 0.5$ to the geometry.

To plot the result, enter:

```
mphgeom(model,'geom1','vertexmode','on');
```



Code for Use with MATLAB®

```
model = ModelUtil.create('Model');
model.component.create('comp1',true);
geom1 = model.component('comp1').geom.create('geom1',1);
i1 = geom1.feature.create('i1','Interval');
i1.set('intervals','many');
i1.set('p','0,1,2');
geom1.run;
p1 = geom1.feature.create('p1','Point');
p1.set('p',0.5);
geom1.run;
mphgeom(model,'geom1','vertexmode','on')
```

CREATING A 2D GEOMETRY USING PRIMITIVE GEOMETRY OBJECTS

Creating Composite Objects

Use a model object with a 2D geometry. Enter:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1',true);
geom2 = comp1.geom.create('geom2',2);
```

Continue by creating a rectangle with side length of `2` and centered at the origin:

```
sq1 = geom2.feature.create('sq1','Square');
sq1.set('size',2);
sq1.set('base','center');
```

The property `size` describes the side lengths of the rectangle and the property `pos` describes the positioning. The default is to position the rectangle about its lower-left corner. Use the property `base` to control the positioning.

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

```
c1 = geom2.feature.create('c1','Circle');
c1.set('r',0.5);
c1.set('pos',[0 0]);
```

The property `r` describes the radius of the circle, and the property `pos` describes the positioning.

The property `pos` could have been excluded because the default position is the origin. The default is to position the circle about its center.

Drill a hole in the rectangle by subtracting the circle from it:

```
co1 = geom2.feature.create('co1','Compose');
co1.selection('input').set({'c1' 'sq1'});
co1.set('formula','sq1-c1');
```

A selection object is used to refer to the input object. The operators `+`, `*`, and `-` correspond to the set operations union, intersection, and difference, respectively.

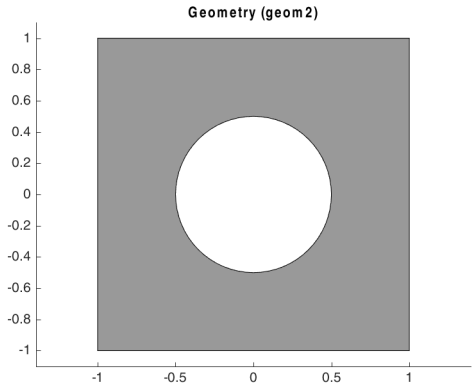
The `Compose` operation allows you to work with a formula. Alternatively use the `Difference` operation instead of `Compose`. The following sequence of commands starts with disabling the `Compose` operation:

```
co1.active(false);

dif1 = geom2.feature.create('dif1','Difference');
dif1.selection('input').set({'sq1'});
dif1.selection('input2').set({'c1'});
```

Run the geometry sequence to create the geometry and plot the result:

```
geom2.run;
```

```
mphgeom(model,'geom2');
```



Trimming Solids

Continue with rounding the corners of the rectangle with the `Fillet` operation:

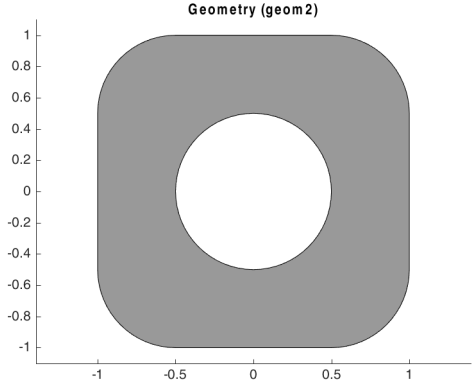
```
fil1 = geom2.feature.create('fil1','Fillet');
fil1.selection('point').set('dif1', [1 2 7 8]);
fil1.set('radius','0.5');
```

Run the sequence again:

```
geom2.run;
```

The geometry sequence is updated with rounded corners. To view the result, enter:

```
mphgeom(model,'geom2');
```



Code for use with MATLAB®

```
model = ModelUtil.create('Model');
```

```
comp1 = model.component.create('comp1',true);
geom2 = comp1.geom.create('geom2',2);
sq1 = geom2.feature.create('sq1','Square');
sq1.set('size',2);
sq1.set('base','center');
c1 = geom2.feature.create('c1','Circle');
c1.set('r',0.5);
c1.set('pos',[0 0]);
co1 = geom2.feature.create('co1','Compose');
co1.selection('input').set({'c1' 'sq1'});
co1.set('formula','sq1-c1');
co1.active(false)
dif1 = geom2.feature.create('dif1','Difference');
dif1.selection('input').set({'sq1'});
dif1.selection('input2').set({'c1'});
geom2.run;
mphgeom(model,'geom2');
fil1 = geom2.feature.create('fil1','Fillet');
fil1.selection('point').set('dif1', [1 2 7 8]);
fil1.set('radius','0.5');
geom2.run;
mphgeom(model,'geom2');
```

CREATING A 2D GEOMETRY USING BOUNDARY MODELING

Use the following commands to create six open curve segments that together form a closed curve:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1',true);
g1 = comp1.geom.create('g1',2);
w=1/sqrt(2);
c1 = g1.feature.create('c1','BezierPolygon');
c1.set('type','open');
c1.set('degree',2);
c1.set('p',[-0.5 -1 -1;-0.5 -0.5 0]);
c1.set('w',[1 w 1]);
c2 = g1.feature.create('c2','BezierPolygon');
c2.set('type','open');
c2.set('degree',2);
c2.set('p',[-1 -1 -0.5;0 0.5 0.5]);
c2.set('w',[1 w 1]);
c3 = g1.feature.create('c3','BezierPolygon');
c3.set('type','open');
c3.set('degree',1);
c3.set('p',[-0.5 0.5; 0.5 0.5]);
c4 = g1.feature.create('c4','BezierPolygon');
c4.set('type','open');
c4.set('degree',2);
c4.set('p',[0.5 1 1; 0.5 0.5 0]);
```

```
c4.set('w',[1 w 1]);
c5 = g1.feature.create('c5','BezierPolygon');
c5.set('type','open');
c5.set('degree',2);
c5.set('p',[1 1 0.5; 0 -0.5 -0.5]);
c5.set('w',[1 w 1]);
c6 = g1.feature.create('c6','BezierPolygon');
c6.set('type','open');
c6.set('degree',1);
c6.set('p',[0.5 -0.5; -0.5 -0.5]);
```

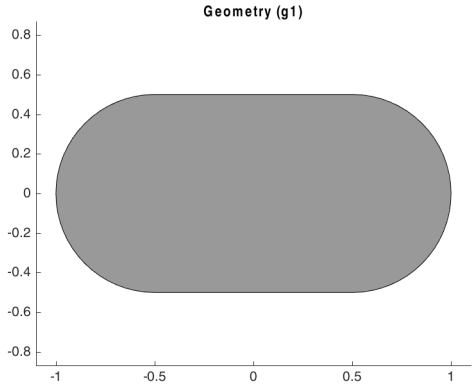
The objects `c1`, `c2`, `c3`, `c4`, `c5`, and `c6` are all `curve2` objects. The vector `[1 w 1]` specifies the weights for a rational Bézier curve that is equivalent to a quarter-circle arc. The weights can be adjusted to create elliptical or circular arcs.

Convert the curve segments to a solid with the following conversion command:

```
csol1 = g1.feature.create('csol1','ConvertToSolid');
csol1.selection('input').set({'c1' 'c2' 'c3' 'c4' 'c5' 'c6'});
```

Then issue a final `run` command:

```
g1.run;
mphgeom(model,'g1');
```



Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1',true);
g1 = comp1.geom.create('g1',2);
w=1/sqrt(2);
c1 = g1.feature.create('c1','BezierPolygon');
c1.set('type','open');
c1.set('degree',2);
c1.set('p',[-0.5 -1 -1;-0.5 -0.5 0]);
```

```
c1.set('w',[1 w 1]);
c2 = g1.feature.create('c2','BezierPolygon');
c2.set('type','open');
c2.set('degree',2);
c2.set('p',[-1 -1 -0.5;0 0.5 0.5]);
c2.set('w',[1 w 1]);
c3 = g1.feature.create('c3','BezierPolygon');
c3.set('type','open');
c3.set('degree',1);
c3.set('p',[-0.5 0.5; 0.5 0.5]);
c4 = g1.feature.create('c4','BezierPolygon');
c4.set('type','open');
c4.set('degree',2);
c4.set('p',[0.5 1 1; 0.5 0.5 0]);
c4.set('w',[1 w 1]);
c5 = g1.feature.create('c5','BezierPolygon');
c5.set('type','open');
c5.set('degree',2);
c5.set('p',[1 1 0.5; 0 -0.5 -0.5]);
c5.set('w',[1 w 1]);
c6 = g1.feature.create('c6','BezierPolygon');
c6.set('type','open');
c6.set('degree',1);
c6.set('p',[0.5 -0.5; -0.5 -0.5]);
csol1 = g1.feature.create('csol1','ConvertToSolid');
csol1.selection('input').set({'c1' 'c2' 'c3' 'c4' 'c5' 'c6'});
g1.run;
mphgeom(model,'g1');
```

CREATING A 3D GEOMETRY USING SOLID MODELING

This section shows how to create 3D solids using work planes and Boolean operations.

Create a 3D geometry with an *xy* work plane at $z = 0$:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1',true);

geom1 = comp1.geom.create('geom1', 3);

wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
```

Add a rectangle to the work plane, then add fillet to its corners:

```
r1 = wp1.geom.feature.create('r1', 'Rectangle');
r1.set('size',[1 2]);

geom1.run;
```

```
fil1 = wp1.geom.feature.create('fil1', 'Fillet');
fil1.selection('point').set('r1', [1 2 3 4]);
fil1.set('radius', '0.125');

geom1.runCurrent;

ext1 = geom1.feature.create('ext1', 'Extrude');
ext1.set('distance', '0.1');
```

Add another *yz* work plane, at $x = 0.5$:

```
wp2 = geom1.feature.create('wp2', 'WorkPlane');
wp2.set('planetype', 'quick');
wp2.set('quickplane', 'yz');
wp2.set('quickx', '0.5');

b1 = wp2.geom.feature.create('b1', 'BezierPolygon');
b1.set('type', 'open');
b1.set('degree', [1 1 1 1]);
b1.set('p',
{'0.75','1','1','0.8','0.75';'0.1','0.1','0.05','0.05','0.1'});
b1.set('w', {'1','1','1','1','1','1','1','1'});

wp2.geom.feature.create('csol1', 'ConvertToSolid');
wp2.geom.feature('csol1').selection('input').set({'b1'});
```

Revolve the triangle from the *yz* work plane:

```
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.selection('input').set({'wp2'});
rev1.setIndex('pos', '1', 0);
```

Add the difference operation that computes the final 3D geometry:

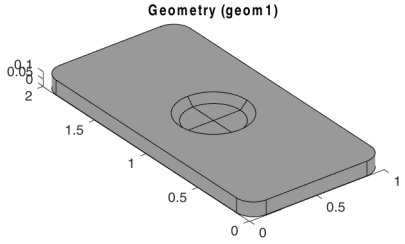
```
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'ext1'});
dif1.selection('input2').set({'rev1'});
```

To run the sequence, enter:

```
geom1.run;
```

To view the geometry enter:

```
mphgeom(model);
```



Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1',true);
geom1 = comp1.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
r1 = wp1.geom.feature.create('r1', 'Rectangle');
r1.set('size',[1 2]);
geom1.run
fil1 = wp1.geom.feature.create('fil1', 'Fillet');
fil1.selection('point').set('r1', [1 2 3 4]);
fil1.set('radius', '0.125');
geom1.runCurrent;
ext1 = geom1.feature.create('ext1', 'Extrude');
ext1.set('distance', '0.1');
wp2 = geom1.feature.create('wp2', 'WorkPlane');
wp2.set('planetype', 'quick');
wp2.set('quickplane', 'yz');
wp2.set('quickx', '0.5');
b1 = wp2.geom.feature.create('b1', 'BezierPolygon');
b1.set('type', 'open');
b1.set('degree', [1 1 1 1]);
b1.set('p',
{'0.75','1','1','0.8','0.75';'0.1','0.1','0.05','0.05','0.1'});
b1.set('w', {'1','1','1','1','1','1','1','1'});
wp2.geom.feature.create('csol1', 'ConvertToSolid');
wp2.geom.feature('csol1').selection('input').set({'b1'});
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.selection('input').set({'wp2'});
rev1.setIndex('pos', '1', 0);
dif1 = geom1.feature.create('dif1', 'Difference');
```

```
dif1.selection('input').set({'ext1'});
dif1.selection('input2').set({'rev1'});
geom1.run;
mphgeom(model);
```

Retrieving Geometry Information

With the function `mphgeominfo` you can access detailed information of a geometry object as well its data.

To get the information of a geometry object enter the command:

```
info = mphgeominfo(model, <geomtag>);
```

where *<geomtag>* is the tag of either a geometry node or a part geometry node you want the information from. In case of the model has only geometry node, the tag *<geomtag>* is optional.

The output `info` is a MATLAB® structure. The default fields available in the structure are listed in the table:

FIELDS	DESCRIPTION
`sdim`	Space dimension
`label`	Label of the selected geometry object
`component`	Tag of the component
`geometricmodel`	Node defining the geometry for the physics
`autobuildnew`	Build geometric operation automatically when added
`autorebuild`	Geometry sequence automatically rebuilt
`lengthunit`	Current length unit
`angularunit`	Current angular unit
`objectnames`	Names of all objects that exist in the current state
`current`	Tag of the current feature
`geomrep`	Geometry representation
`scaleunitvalue`	Scale numeric values in the geometry and meshing sequences
`repairtol`	Relative repair tolerance
`repairtoltype`	Repair tolerance type
`absrepairtol`	Absolute repair tolerance
`useconstrdim`	Constraints and dimensions functionality enabled
`constrdimbuild`	Constraints and dimensions used when building the geometry object

FIELDS	DESCRIPTION
constrdimstatus	Overall status of the constraints and dimensions
ispart	Object is a geometry part
view	Current view
exists	Geometry object exists
isaxisymmetric	Geometry is axisymmetric
boundingbox	Bounding box of the geometry objects
type	Object type
Ndomains	Number of domains
Nboundaries	Number of boundaries
Nedges	Number of edges
Nvertices	Number of vertices
Nfinitevoids	Number of finite voids
Nfaces	Number of faces
problems	List of error/warning messages

In addition to geometry information, you can retrieve geometric entity data, such as local parameters, coordinates, curvature, up and down domain indices, etc... To get the geometry data enter:

```
[info, data] = mphgeominfo(model, <geomtag>, 'entity', <entitytype>);
```

where *<entitytype>* is the type of the entity to get the data from. It can be either `'face'`, `'edge'`, or `'vertex'`.

`data` is a MATLAB structure which fields depend on the entity type. These are listed in the table below:

FIELDS	ENTITY	DESCRIPTION
domainnumber	vertex	Domain index for isolated vertices
edgex	edge	Edge coordinates
edgedx	edge	Edge first order derivatives
edgeddx	edge	Edge second order derivatives
edgecurvature	edge	Edge curvature values
edgenormal	edge	Edge normal values (in 2D only)
edgetorsion	edge	Edge torsion values (in 3D only)
facex	face	Face coordinates
facedx	face	Face first order derivatives

FIELDS	ENTITY	DESCRIPTION
faceddx	face	Face second order derivatives
facenormal	face	Face normal
faceff1	face	Face first fundamental form
faceff2	face	Face second fundamental form
facegausscurvature	face	Face Gauss curvature
meancurvature	face	Face mean curvature
updown	face|edge	Up and down domain indices
p	vertex	Vertex coordinates
paramrange	face|edge	Parameter ranges

In addition you can specify a selection of the entity type to get the data from as in the command below:

```
[info, data] = mphgeominfo(model, <geomtag>, ...
  'entity', <entitytype>, 'selection', <sel>);
```

where *<sel>* is an array of entity numbers.

The geometric data are evaluated within the local parameter range on a structured grid for faces, or interval for edge, with default size of 10x10, 10 respectively. To change the size of geometric data evaluation point use the `steps` property as in the command below:

```
[info, data] = mphgeominfo(model, <geomtag>, ...
  'entity', <entitytype>, 'selection', <sel>, 'steps', <steps>);
```

where *<steps>* is an integer or a NxM integer array for a face type in case you need to evaluate the data on an uneven grid.

To retrieve the detailed information about the geometry in a model using the COMSOL API, see Geometry Object Information Methods (GeomInfo) in the *COMSOL Multiphysics Programming Reference Manual*.

RETRIEVING GEOMETRY INFORMATION

This example shows how to use `mphgeominfo` to retrieve geometry information.

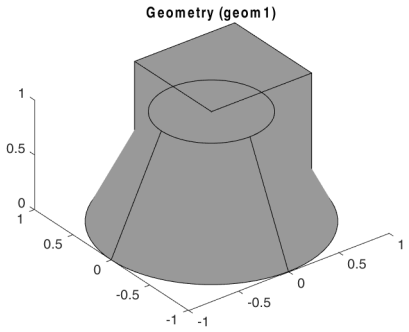
First create a simple 3D geometry:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1',true);
```

```
geom1 = comp1.geom.create('geom1', 3);
geom1.feature.create('blk1','Block');
geom1.feature.create('con1','Cone');
geom1.run;
```

To visualize the geometry in a MATLAB figure window enter:

```
mphgeom(model)
```



As only one geometry node is available in the model, to access the geometry information enter:

```
info = mphgeominfo(model)
```

To determine the space dimension of the geometry, enter:

```
info.sdim
```

To inquire about the number of domains and the number of boundaries:

```
info.Ndomains
info.Nboundaries
```

The bounding box coordinates of the geometry are accessible using:

```
info.boundingbox
```

To get the geometry data for the face number 1 enter the command:

```
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'face',...
  'selection', 1);
```

To get the range of the surface parameters enter:

```
data.paramrange
```

this returns a 4x1 array following the given format: [*s1min*; *s1max*; *s2min*; *s2max*] where *s1min* and *s1max* are the minimum, and maximum respectively, of the first surface parameter. *s2min* and *s2max* are the minimum, and maximum respectively, of the second surface parameter.

To evaluate the face coordinates, in the global coordinate system, enter:

```
data.facex
```

The face coordinates are evaluated with the local parameter range on a 10 x 10 points grid. If you want to reduce or increase the size of evaluation grid, use the property `'steps'`. For instance to evaluate the face coordinates on a 5 x 10 grid enter:

```
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'face',...
  'selection', 1,'steps', [5 10]);
```

To get the parameter range of an edge and, for example, to get the length of edge number 3, enter:

```
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'edge',...
  'selection', 3);
paramrange = data.paramrange;
edgelength = paramrange(end)
```

To get the coordinates and the curvature data at the middle of edge number 3 enter:

```
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'edge',...
  'selection', 3,'steps', 1);
pt = data.edgex
curvature = data.edgecurvature
```

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1',true);
geom1 = comp1.geom.create('geom1', 3);
geom1.feature.create('blk1','Block');
geom1.feature.create('con1','Cone');
geom1.run;
mphgeom(model)
info = mphgeominfo(model);
info.sdim
info.Ndomains
info.Nboundaries
info.boundingbox
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'face',...
'selection', 1);
data.paramrange
data.facex
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'face',...
'selection', 1,'steps', [5 10]);
```

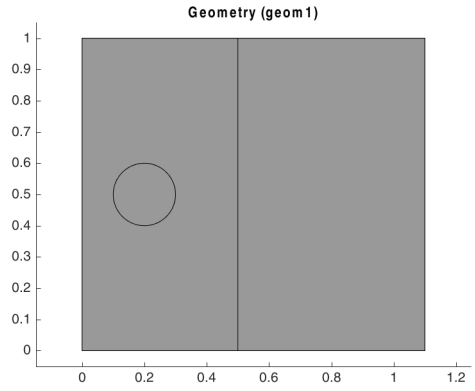
```
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'edge',...
'selection', 3);
paramrange = data.paramrange;
edgelength = paramrange(end)
[info, data] = mphgeominfo(model, 'geom1', 'entity', 'edge',...
'selection', 3,'steps', 1);
pt = data.edgex
curvature = data.edgecurvature
```

Modeling with a Parameterized Geometry

COMSOL Multiphysics has built-in support for parameterized geometries. Parameters can be used in most geometry operations. To exemplify parameterizing a geometry, the following script studies the movement of a circular source through two adjacent rectangular domains:

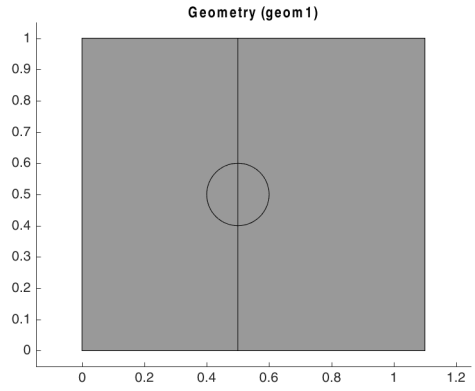
```
model = ModelUtil.create('Model');
model.param.set('a',0.2);
comp1 = model.component.create('comp1',true);

geom1 = comp1.geom.create('geom1',2);

r1 = geom1.feature.create('r1','Rectangle');
r1.set('size',[0.5 1]);
r1.set('pos',[0 0]);

r2 = geom1.feature.create('r2','Rectangle');
r2.set('size',[0.6 1]);
r2.set('pos',[0.5 0]);

c1 = geom1.feature.create('c1','Circle');
c1.set('r',0.1);
c1.set('pos',{'a','0.5'});
```

```
mphgeom(model);
```



Change the position of the circle by changing the value of parameter `a`:

```
model.param.set('a',0.5);

mphgeom(model);
```



Create a loop that changes the position of the circle in increments:

```
for a=0.2:0.1:0.5
  model.param.set('a',a);
  geom1.run;
end
```

Create a mesh:

```
comp1.mesh.create('mesh1');
```

Add a Weak Form PDE interface:

```
w = comp1.physics.create('w', 'WeakFormPDE', 'geom1');
w.feature('wfeq1').set('weak', 1, '-test(ux)*ux-test(uy)*uy');

dir1 = w.feature.create('dir1', 'DirichletBoundary', 1);
dir1.selection.set([1 2 3 6 7]);

src1 = w.feature.create('src1', 'SourceTerm', 2);
src1.set('f', 1, '1');
src1.selection.set([3]);
```

Then, create a stationary study step:

```
std1 = model.study.create('std1');

stat1 = std1.feature.create('stat1', 'Stationary');
```

Create a parametric sweep feature:

```
p1 = model.batch.create('p1','Parametric');
p1.set('pname', 'a');
p1.set('plist','range(0.2,0.1,0.8)');
p1.run;
```

Alternatively, you can run the parametric sweep using a MATLAB for loop:

```
for a=0.2:0.1:0.8
  model.param.set('a',a);
  std1.run;
end
```

After updating a parameter that affects the geometry, COMSOL detects this change and automatically updates the geometry and mesh before starting the solver. The geometry is associative, which means that physics settings are preserved as the geometry changes.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
model.param.set('a',0.2);
comp1 = model.component.create('comp1',true);
geom1 = comp1.geom.create('geom1',2);
r1 = geom1.feature.create('r1','Rectangle');
r1.set('size',[0.5 1]);
r1.set('pos',[0 0]);
r2 = geom1.feature.create('r2','Rectangle');
r2.set('size',[0.6 1]);
r2.set('pos',[0.5 0]);
c1 = geom1.feature.create('c1','Circle');
```

```
c1.set('r',0.1);
c1.set('pos',{'a','0.5'});
mphgeom(model);
model.param.set('a',0.5);
mphgeom(model);
for a=0.2:0.1:0.5
model.param.set('a',a);
geom1.run;
end
comp1.mesh.create('mesh1');
w = comp1.physics.create('w', 'WeakFormPDE', 'geom1');
w.feature('wfeq1').set('weak', 1, '-test(ux)*ux-test(uy)*uy');
dir1 = w.feature.create('dir1', 'DirichletBoundary', 1);
dir1.selection.set([1 2 3 6 7]);
src1 = w.feature.create('src1', 'SourceTerm', 2);
src1.set('f', 1, '1');
src1.selection.set([3]);
std1 = model.study.create('std1');
stat1 = std1.feature.create('stat1', 'Stationary');
p1 = model.batch.create('p1','Parametric');
p1.set('pname', 'a');
p1.set('plist','range(0.2,0.1,0.8)');
p1.run;
for a=0.2:0.1:0.8
model.param.set('a',a);
std1.run;
end
```

Images and Interpolation Data

This section describes how to generate geometry from a set of data points by using interpolation curves and how to create geometry from image data.

- Creating a Geometry Using Curve Interpolation
- Creating Geometry from Image Data

CREATING A GEOMETRY USING CURVE INTERPOLATION

Use the interpolation spline feature to import a set of data points that describe a 2D geometry. To create an interpolation spline feature, enter:

```
model.geom(<geomtag>).feature.create(<ftag>,'InterpolationCurve')
```

Then specify data points in a table:

```
ftag.set('table', <data>)
```

where `ftag` is the curve interpolation node and *<data>* can either be a 2-by-*N* cell array or a 2-by-*N* array.

Control the type of geometry generated by the operation with the command:

```
ftag.set('type', type)
```

where *type* can either be `'solid'` to generate a solid object, `'closed'` to generate a closed curve or `'open'` to generate an open curve.

Example: Curve Interpolation

Create a set of data points in MATLAB, then use these to construct a 2D geometry.

1. Create data points that describe a circle, 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+0.02*randn(size(p));
```

3. Create a 2D geometry with a square:

```
model = ModelUtil.create('Model');
```

4. Add a square geometry:

```
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);

sq1 = geom1.feature.create('sq1', 'Square');
sq1.set('base', 'center');
sq1.set('size', '3');
```

5. Add an interpolation curve feature:

```
ic1 = geom1.feature.create('ic1', 'InterpolationCurve');
```

6. Use the variable `p` for the data points:

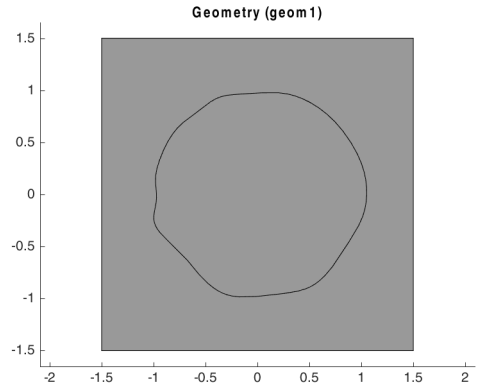
```
ic1.set('table', p');
```

7. Specify a closed curve:

```
ic1.set('type', 'closed');
```

8. Plot the geometry with the `mphgeom` command:

```
mphgeom(model)
```



Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
sq1 = geom1.feature.create('sq1', 'Square');
sq1.set('base', 'center');
sq1.set('size', '3');
phi = 0:0.2:2*pi;
phi([1 3 6 7 10 20 21 25 28 32]) = [];
p = [cos(phi);sin(phi)];
randn('state',17)
p = p+0.02*randn(size(p));
ic1 = geom1.feature.create('ic1', 'InterpolationCurve');
ic1.set('table', p');
ic1.set('type', 'closed');
mphgeom(model);
```

CREATING GEOMETRY FROM IMAGE DATA

Use the function `mphimage2geom` to create geometry from image data. The image data format can be *M*-by-*N* array for a grayscale image or *M*-by-*N*-by-3 array for a true color image. This section also includes an example (see Example: Convert Image Data to Geometry).

See the MATLAB function `imread` to convert an image file to image data.

If you specify the image data and the level value that represents the geometry contour you want to extract, the function `mphimage2geom` returns a model object with the desired geometry:

```
model = mphimage2geom(<imagedata>, <level>)
```

where *imagedata* is a C array containing the image data, and *level* is the contour level value used to generate the geometry contour.

Specify the type of geometry object generated:

```
model = mphimage2geom(<imagedata>, <level>, 'type', type)
```

where *type* is `'solid'` and generates a solid object, `'closed'` generates a closed curve object, or `'open'` generates an open curve geometry object.

Use the property `curvetype` to specify the type of curve used to generate the geometry object:

```
model = mphimage2geom(<imagedata>, <level>, 'curvetype', curvetype)
```

where *curvetype* can be set to `'polygon'` to use a polygon curve. The default curve type creates a geometry with the best suited geometrical primitives. For interior curves it uses *interpolation curves*, while for curves that are touching the perimeter of the image a *polygon curve* is used.

To scale the geometry use the `scale` property where *scale* is a double value:

```
model = mphimage2geom(<imagedata>, <level>, 'scale', scale)
```

Set the minimum distance (in pixels) between coordinates in curve with the `mindist` property where *mindist* is a double value:

```
model = mphimage2geom(<imagedata>, <level>, 'mindist', mindist)
```

Set the minimum area (in square pixels) for interior curves where *minarea* is a double value:

```
model = mphimage2geom(<imagedata>, <level>, 'minarea', minarea)
```

In case of overlapping solids, the function `mphimage2geom` automatically creates a `Compose` node in the model object. If you do not want this geometry feature, set the property `compose` to `off`:

```
model = mphimage2geom(<imagedata>, <level>, 'compose', 'off')
```

To create a rectangle domain surrounding the object generated use the property `rectangle`:

```
model = mphimage2geom(<imagedata>, <level>, 'rectangle', 'on')
```

`mphimage2geom` returns a model object with the created geometry stored in a geometry node. The default geometry node has the tag geom1, to specify manually the geometry tag use the function as below:

```
model = mphimage2geom(<imagedata>, <level>, 'geom', <geomtag>)
```

where *<geomtag>* is a string corresponding to the tag of the geometry node.

It is also possible to create a geometry object and include it in an existing model object, to proceed use the command below:

```
mphimage2geom(<imagedata>, <level>, 'geom', <geomnode>)
```

where *<geomnode>* is the geometry node object where to include the newly generated geometry.

To manually specify the tag of the model object created in the COMSOL *server* use the command below:

```
model = mphimage2geom(<imagedata>, <level>, 'modeltag', <Modeltag>)
```

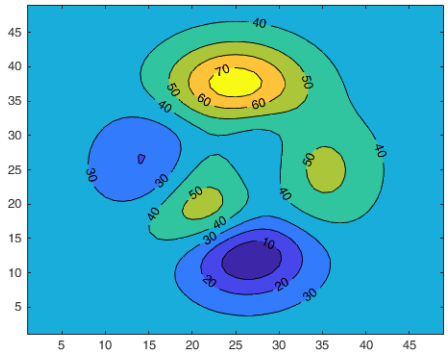
where *<Modeltag>* is a string defining the tag of the model object in the COMSOL *server*.

Example: Convert Image Data to Geometry

This example shows how to create geometry based on gray scale image data. First generate the image data in MATLAB and display the contour in a figure. Then, create a model object including the geometry represented by the contour value 40.

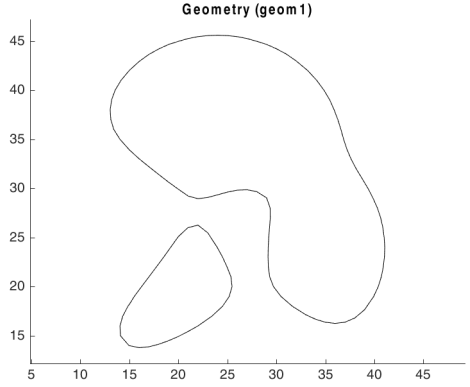
At the MATLAB prompt enter these commands:

```
p = (peaks+7)*5;
[c,h] = contourf(p);
clabel(c, h);
model = mphimage2geom(p, 40);
figure(2)
mphgeom(model)
```



Use the property `type` to create closed or open curves. For example, to create a geometry following contour 40 with closed curves, enter:

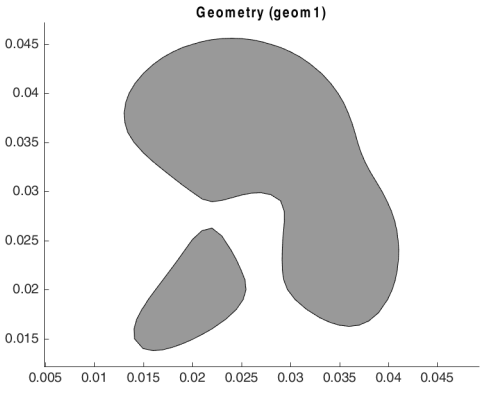
```
model = mphimage2geom(p, 40, 'type', 'closed');
mphgeom(model)
```



To scale the geometry, use the `scale` property. Using the current model scale the geometry with a factor of 0.001 (1e-3):

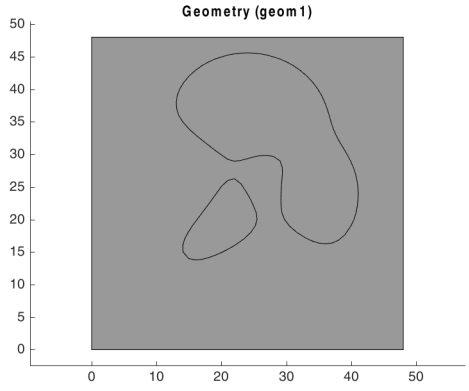
```
model = mphimage2geom(p, 40, 'scale', 1e-3);
```

```
mphgeom(model)
```



To insert a rectangle in the geometry that has an outer domain surrounding the created contour, set the property `rectangle` to `on`:

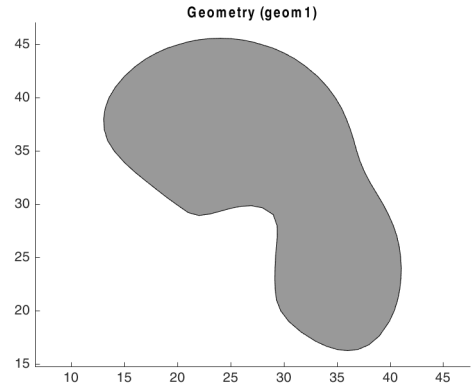
```
model = mphimage2geom(p, 40, 'rectangle', 'on');
mphgeom(model)
```



Only include the interior curves with an area larger than 100 square pixels:

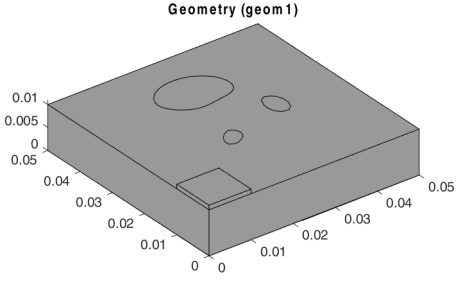
```
model = mphimage2geom(p, 40, 'minarea', 100);
```

```
mphgeom(model)
```



Insert the geometry in an existing geometry object:

```
model = mphopen('model_tutorial_llmatlab');
geom1 = model.component('comp1').geom('geom1');
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('quickz',1e-2);
mphimage2geom(p, 50, 'scale', 1e-3, 'geom', wp1.geom);
mphgeom(model)
```



Code for use with MATLAB®

```
p = (peaks+7)*5;
[c,h] = contourf(p);
clabel(c, h);
model = mphimage2geom(p, 40);
figure(2)
mphgeom(model)
```

```
model = mphimage2geom(p, 40, 'type', 'closed');
mphgeom(model)
model = mphimage2geom(p, 40, 'scale', 1e-3);
mphgeom(model)
model = mphimage2geom(p, 40, 'rectangle', 'on');
mphgeom(model)
model = mphimage2geom(p, 40, 'minarea', 100);
mphgeom(model)
model = mphopen('model_tutorial_llmatlab');
geom1 = model.component('comp1').geom('geom1');
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('quickz',1e-2);
mphimage2geom(p, 50, 'scale', 1e-3, 'geom', wp1.geom);
mphgeom(model)
```

Measuring Entities in Geometry

Use the function `mphmeasure` to measure the geometry entities in the model.

Enter the command:

```
measure = mphmeasure(model, <geomtag>, entity, …)
```

to get the measurement of the entity type `entity` in the geometry *<geomtag>*. *entity* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

The output `measure` return the value of a coordinates, a length, an area or volume, respectively. For point entities, if you select two or more points, the output correspond to the midpoint coordinates.

To specify the entity selection to measure enter:

```
measure = mphmeasure(model, <geomtag>, entity, 'selection', sel, …)
```

where *sel* is an integer array that contains the selection number of the entities to measure.

When you select several entities you can get another measurement value with the command:

```
[msr1, msr2]= mphmeasure(model, <geomtag>, entity, …)
```

where `msr2` corresponds to a surface area when the entities are volumes and a distance when the input entities are two points.

Working with Meshes

This section describes how to set up and run meshing sequences in a model.

- Meshing in the *COMSOL Multiphysics Reference Manual*
- Mesh in the *COMSOL Multiphysics Programming Reference Manual*

The Meshing Sequence Syntax

Create a meshing sequence by using the syntax:

```
model.component(<ctag>).mesh.create(<meshtag>, <geomtag>)
```

where *<meshtag>* is a string that you use to refer to the meshing sequence. The tag *geomtag* specifies the geometry to use for this mesh node.

To add an operation to a sequence, use the syntax:

```
mesh.feature.create(<ftag>, operation)
```

where `mesh` is a link to a mesh node and the string *<ftag>* is a string that you use to refer to the operation.

About Mesh Commands in the *COMSOL Multiphysics Programming Reference Manual*

To set a property to a value in a operation, enter:

```
mesh.feature(<ftag>).set(property, <value>)
```

To build the mesh sequence, enter:

```
mesh.run
```

To run the mesh node up to a specified feature node *<ftag>*, enter:

```
mesh.run(ftag)
```

For more details on available operations and properties in the sequence, see Mesh in the *COMSOL Multiphysics Programming Reference Manual.*

Displaying the Mesh

To display the mesh in a MATLAB figure, use the function `mphmesh`. Make sure that the mesh is built before calling this command:

```
mphmesh(model)
```

If there are several meshes in a model, specify the mesh to display using the command:

```
mphmesh(model, <meshtag>)
```

Adding a `view` property will add some view settings from the COMSOL model such as axes labels (units) and grid and supports hiding of mesh entities. Usually it is sufficient to use the `auto` value for the view property:

```
mphmesh(model, <meshtag>, 'view', 'auto')
```

If the model only contains one mesh then the `<meshtag>` may be left empty.

Use the `parent` property to specify the axes handle where to display the plot:

```
mphmesh(model, <meshtag>, 'parent', <axes>)
```

The following properties are also available to specify the vertex, edge, or face rendering:

• `vertexmode` • `edgemode` • `facemode` • `vertexlabels` • `edgelabels` • `facelabels`	• `meshcolor` • `edgecolor` • `vertexcolor` • `edgelabelscolor` • `vertexlabelscolor` • `facelabelscolor` • `facealpha`

Mesh Creation Functions

Several mesh features are discussed, with examples in this section:

- Mesh Sizing Properties
- Creating a 2D Mesh with Triangular Elements
- Changing The Tessellation Method
- Creating a 2D Mesh with Quadrilateral Elements
- Creating Structured Meshes
- Creating a Structured Quadrilateral Mesh
- Building a Mesh Incrementally
- Revolving a Mesh by Sweeping
- Extruding a Mesh by Sweeping
- Combining Unstructured and Structured Meshes
- Creating Boundary Layer Meshes
- Refining Meshes
- Copying Boundary Meshes
- Converting Mesh Elements

MESH SIZING PROPERTIES

The `Size` attribute provides a number of input properties that can control the mesh element size, such as the following properties:

- Maximum and minimum element size

- Element growth rate
- Curvature factor
- Resolution of narrow regions

These properties are available both globally and locally. The following examples are included: Creating a 2D Mesh with Triangular Elements and Creating a 2D Mesh with Quadrilateral Elements. Also discussed is Changing The Tessellation Method.

There are several predefined settings that can be used 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 resolution:

• Extremely fine (1) • Extra fine (2) • Finer (3) • Fine (4) • Normal (5) (the default)	• Coarse (6) • Coarser (7) • Extra coarse (8) • Extremely coarse (9)

For details about predefined mesh size settings and mesh element size parameters, see `Size` in the *COMSOL Multiphysics Programming Reference Manual.*

CREATING A 2D MESH WITH TRIANGULAR ELEMENTS

Generate a triangular mesh of a unit square:

```
model = ModelUtil.create('Model');

comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');

mesh1 = comp1.mesh.create('mesh1');
ftri1 = mesh1.feature.create('ftri1','FreeTri');
mesh1.run;
```

```
mphmesh(model)
```

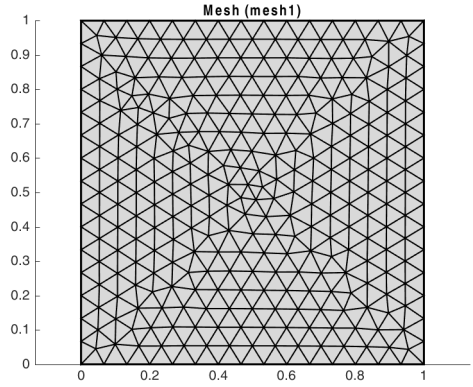


Figure 3-1: Default mesh on a unit square.

The default size feature is generated with the property `hauto` set to `5`, that is:

```
mesh1.feature('size').set('hauto',5);
```

To override this behavior, set `hauto` to another integer. Override this by setting specific size properties, for example, making the mesh finer than the default by specifying a maximum element size of 0.02:

```
mesh1.feature('size').set('hmax',0.02);
mesh1.run;

mphmesh(model)
```

This value corresponds to 1/50 of the largest axis-parallel distance, whereas the default value is 1/15.

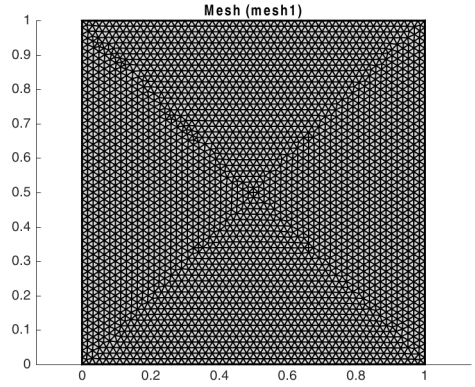


Figure 3-2: Fine mesh (maximum element size = 0.02).

Sometimes a nonuniform mesh is desirable. Make a mesh that is denser on the left side by specifying a smaller maximum element size only on the edge segment to the left (edge number 1):

```
mesh1.feature('size').set('hauto',5);

size1 = ftri1.feature.create('size1','Size');
size1.set('hmax',0.02);
size1.selection.geom('geom1',1);
size1.selection.set(1);

mesh1.run

mphmesh(model)
```

.

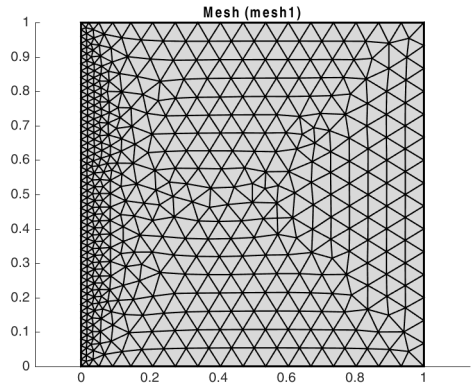


Figure 3-3: Refined mesh on boundary 1 (maximum element size = 0.02).

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
mesh1 = comp1.mesh.create('mesh1');
ftri1 = mesh1.feature.create('ftri1','FreeTri');
mesh1.run;
mphmesh(model)
mesh1.feature('size').set('hauto',5);
mesh1.feature('size').set('hmax',0.02);
mesh1.run;
mphmesh(model)
mesh1.feature('size').set('hauto',5);
size1 = ftri1.feature.create('size1','Size');
size1.set('hmax',0.02);
size1.selection.geom('geom1',1);
size1.selection.set(1);
mesh1.run
mphmesh(model)
```

CHANGING THE TESSELLATION METHOD

You can set the tessellation method for generating triangular meshes using the `method` property. The default value `auto` lets the software use the method that is best suited for the geometry. The values `af` and `del` allow you to switch to an advancing front algorithm and a Delaunay algorithm, respectively. Enter the following to create a 2D geometry, then generate a triangular mesh with the Delaunay algorithm:

```
model = ModelUtil.create('Model');

comp1 = model.component.create('comp1', true);

geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');

c1 = geom1.feature.create('c1','Circle');
c1.set('r','0.5');

co1 = geom1.feature.create('co1','Compose');
co1.selection('input').set({'c1' 'r1'});
co1.set('formula','r1-c1');

geom1.runAll;

mesh1 = comp1.mesh.create('mesh1');

ftri1 = mesh1.feature.create('ftri1','FreeTri');
ftri1.set('method','del');

mesh1.run;

mphmesh(model,'mesh1')
```

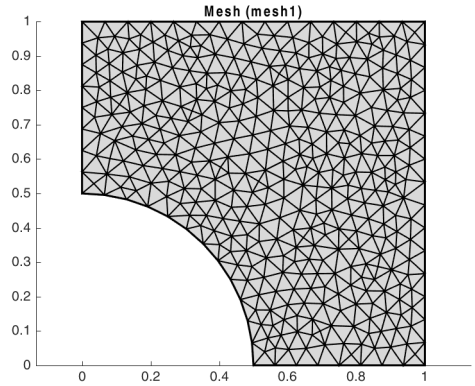


Figure 3-4: Mesh created with the Delaunay method.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
c1 = geom1.feature.create('c1','Circle');
c1.set('r','0.5');
```

```
co1 = geom1.feature.create('co1','Compose');
co1.selection('input').set({'c1' 'r1'});
co1.set('formula','r1-c1');
geom1.runAll;
mesh1 = comp1.mesh.create('mesh1');
ftri1 = mesh1.feature.create('ftri1','FreeTri');
ftri1.set('method','del');
mesh1.run;
mphmesh(model,'mesh1')
```

CREATING A 2D MESH WITH QUADRILATERAL ELEMENTS

To create an unstructured quadrilateral mesh on a unit circle, enter:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);

geom1 = comp1.geom.create('geom1',2);

geom1.feature.create('c1','Circle');

mesh1 = comp1.mesh.create('mesh1');

mesh1.feature.create('ftri1','FreeQuad');

mesh1.run;

mphmesh(model)
```

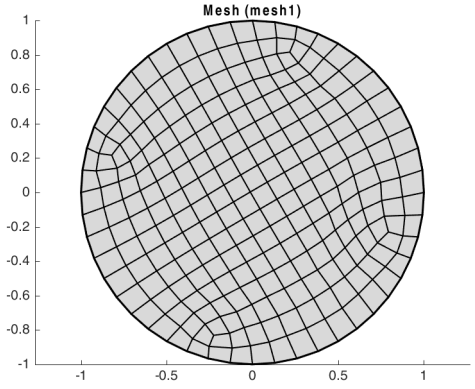


Figure 3-5: An unstructured quad mesh.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
```

```
geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('c1','Circle');
mesh1 = comp1.mesh.create('mesh1');
mesh1.feature.create('ftri1','FreeQuad');
mesh1.run;
mphmesh(model)
```

CREATING STRUCTURED MESHES

To create a structured quadrilateral mesh in 2D, use the `Map` operation. This operation uses a mapping technique to create the quadrilateral mesh.

`Map` in the *COMSOL Multiphysics Programming Reference Manual*

Use the `EdgeGroup` attribute to group the edges (boundaries) into four edge groups, one for each edge of the logical mesh. To control the edge element distribution use the `Distribution` attribute, which determines the overall mesh density.

CREATING A STRUCTURED QUADRILATERAL MESH

Create a structured quadrilateral mesh on a geometry where the domains are bounded by more than four edges:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
r2 = geom1.feature.create('r2','Rectangle');
r2.set('pos',[1 0]);
c1 = geom1.feature.create('c1','Circle');
c1.set('r','0.5');
c1.set('pos',[1.1 -0.1]);
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'r1' 'r2'});
dif1.selection('input2').set({'c1'});
geom1.run('dif1');

mesh1 = comp1.mesh.create('mesh1');

map1 = mesh1.feature.create('map1','Map');

eg1 = map1.feature.create('eg1', 'EdgeGroup');
eg1.selection.set(1);
eg1.selection('edge1').set([1 3]);
eg1.selection('edge2').set(2);
eg1.selection('edge3').set(8);
```

```
eg1.selection('edge4').set(4);

eg2 = map1.feature.create('eg2', 'EdgeGroup');
eg2.selection.set(2);
eg2.selection('edge1').set(4);
eg2.selection('edge2').set([6 9 10]);
eg2.selection('edge3').set(7);
eg2.selection('edge4').set(5);

mesh1.run;
mphmesh(model);
```

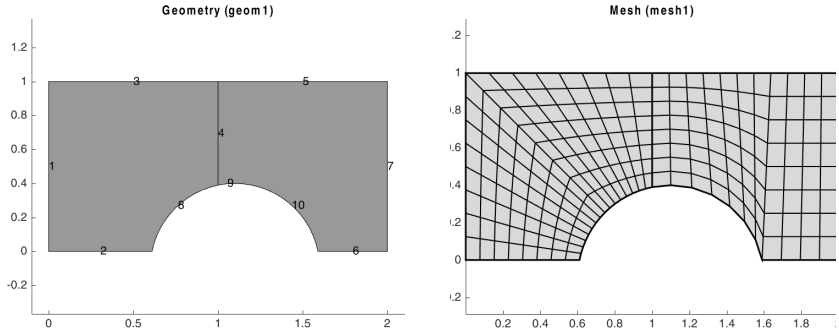


Figure 3-6: Structured quadrilateral mesh (right) and its underlying geometry.

The left-hand side plot in Figure 3-6 is obtained with this command:

```
mphgeom(model, 'geom1', 'edgelabels','on')
```

The `EdgeGroup` attributes specify that the four edges enclosing domain 1 are boundaries 1 and 3; boundary 2; boundary 8; and boundary 4. For domain 2 the four edges are boundary 4; boundary 5; boundary 7; and boundaries 9, 10, and 6.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
r2 = geom1.feature.create('r2','Rectangle');
r2.set('pos',[1 0]);
c1 = geom1.feature.create('c1','Circle');
c1.set('r','0.5');
c1.set('pos',[1.1 -0.1]);
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'r1' 'r2'});
dif1.selection('input2').set({'c1'});
geom1.run('dif1');
```

```
mesh1 = comp1.mesh.create('mesh1');
map1 = mesh1.feature.create('map1','Map');
eg1 = map1.feature.create('eg1', 'EdgeGroup');
eg1.selection.set(1);
eg1.selection('edge1').set([1 3]);
eg1.selection('edge2').set(2);
eg1.selection('edge3').set(8);
eg1.selection('edge4').set(4);
eg2 = map1.feature.create('eg2', 'EdgeGroup');
eg2.selection.set(2);
eg2.selection('edge1').set(4);
eg2.selection('edge2').set([6 9 10]);
eg2.selection('edge3').set(7);
eg2.selection('edge4').set(5);
mesh1.run;
mphmesh(model);
mphgeom(model, 'geom1', 'edgelabels','on')
```

BUILDING A MESH INCREMENTALLY

To build meshes in a step-by-step fashion, create selections for the parts of the geometry that you want to mesh in each step, as in this example:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);

geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
geom1.feature.create('c1','Circle');
uni1 = geom1.feature.create('uni1', 'Union');
uni1.selection('input').set({'c1' 'r1'});
geom1.runCurrent;
del1 = geom1.feature.create('del1', 'Delete');
del1.selection('input').init(1);
del1.selection('input').set('uni1', 8);
geom1.run('del1');

mesh1 = comp1.mesh.create('mesh1');

dis1 = mesh1.feature.create('dis1', 'Distribution');
dis1.selection.set([2 4]);
dis1.set('type', 'predefined');
dis1.set('method', 'geometric');
dis1.set('elemcount', 20);
dis1.set('reverse', 'on');
dis1.set('elemratio', 20);

dis2 = mesh1.feature.create('dis2', 'Distribution');
dis2.selection.set([1 3]);
dis2.set('type', 'predefined');
```

```
dis2.set('method', 'geometric');
dis2.set('elemcount', 20);
dis2.set('elemratio', 20);

map1 = mesh1.feature.create('map1','Map');
map1.selection.geom('geom1', 2);
map1.selection.set(2);
mesh1.feature.create('frt1','FreeTri');

mesh1.run;

mphmesh(model)
```

The final mesh is in Figure 3-7. Note the effect of the `Distribution` feature, with which the distribution of vertex elements along geometry edges can be controlled.

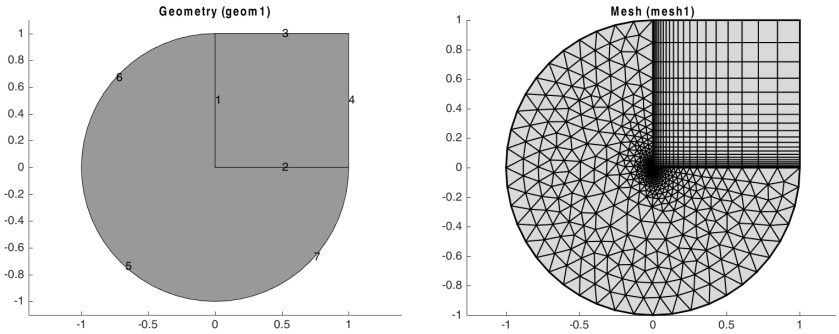


Figure 3-7: Incrementally generated mesh (right).

The left-hand side plot in Figure 3-7 is obtained with this command:

```
mphgeom(model, 'geom1', 'edgelabels','on')
```

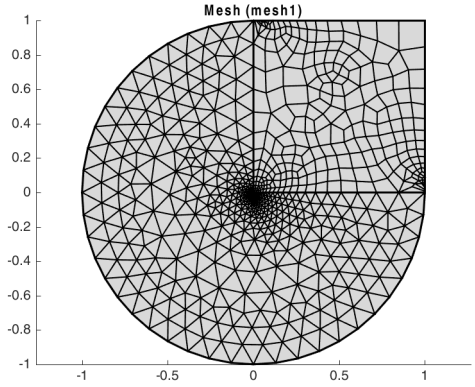
To replace the structured quad mesh by an unstructured quad mesh, delete the `Map` feature and replace it by a `FreeQuad` feature:

```
mesh1.feature.remove('map1');
mesh1.run('dis1');
fq1 = mesh1.feature.create('fq1', 'FreeQuad');
fq1.selection.geom('geom1', 2).set(2);
mesh1.run;
```

Analogous to working with the meshing sequence in the Model Builder in the COMSOL Desktop, new features are always inserted after the current feature.

Thus, to get the `FreeQuad` feature before the `FreeTri` feature, the `dis1` feature needs to be made the current feature by building it with the `run` method. Alternatively, parts of a mesh can be selectively removed by using the `Delete` feature. For example, to remove the structured mesh from domain 2 (along with the adjacent edge mesh on edges 3 and 4), and replace it with an unstructured quad mesh, enter these commands:

```
del1 = mesh1.feature.create('del1','Delete');
del1.selection.geom('geom1', 2).set(2);
del1.set('deladj','on');
frq1 = mesh1.feature.create('frq1','FreeQuad');
frq1.selection.geom('geom1', 2).set(2);
mesh1.run;
```



For further details on the various commands and their properties see the *COMSOL Multiphysics Programming Reference Manual.*

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1',2);
geom1.feature.create('r1','Rectangle');
geom1.feature.create('c1','Circle');
uni1 = geom1.feature.create('uni1', 'Union');
uni1.selection('input').set({'c1' 'r1'});
geom1.runCurrent;
del1 = geom1.feature.create('del1', 'Delete');
del1.selection('input').init(1);
del1.selection('input').set('uni1', 8);
```

```
geom1.run('del1');
mesh1 = comp1.mesh.create('mesh1');
dis1 = mesh1.feature.create('dis1', 'Distribution');
dis1.selection.set([2 4]);
dis1.set('type', 'predefined');
dis1.set('method', 'geometric');
dis1.set('elemcount', 20);
dis1.set('reverse', 'on');
dis1.set('elemratio', 20);
dis2 = mesh1.feature.create('dis2', 'Distribution');
dis2.selection.set([1 3]);
dis2.set('type', 'predefined');
dis2.set('method', 'geometric');
dis2.set('elemcount', 20);
dis2.set('elemratio', 20);
map1 = mesh1.feature.create('map1','Map');
map1.selection.geom('geom1', 2);
map1.selection.set(2);
mesh1.feature.create('frt1','FreeTri');
mesh1.run;
mphmesh(model);
mphgeom(model, 'geom1', 'edgelabels','on')
mesh1.feature.remove('map1');
mesh1.run('dis1');
fq1 = mesh1.feature.create('fq1', 'FreeQuad');
fq1.selection.geom('geom1', 2).set(2);
mesh1.run;
del1 = mesh1.feature.create('del1','Delete');
del1.selection.geom('geom1', 2).set(2);
del1.set('deladj','on');
frq1 = mesh1.feature.create('frq1','FreeQuad');
frq1.selection.geom('geom1', 2).set(2);
mesh1.run;
mphmesh(model);
```

REVOLVING A MESH BY SWEEPING

Create 3D volume meshes by extruding and revolving face meshes with the `Sweep` feature. Depending on the 2D mesh type, the 3D meshes can be hexahedral (brick) meshes or prism meshes.

Create and visualize a revolved prism mesh as follows:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);

geom1 = comp1.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
```

```
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('pos', [2, 0]);
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.set('angle2', '60').set('angle1', -60);
rev1.selection('input').set({'wp1'});
geom1.run('rev1');

mesh1 = comp1.mesh.create('mesh1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.feature('ftri1').selection.geom(2);
mesh1.feature('ftri1').selection.set(2);
mesh1.runCurrent;

swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection.geom(3);
swe1.selection.add(1);

mesh1.run;
mphmesh(model)
```

To obtain a torus, leave the `angles` property unspecified; the default value gives a complete revolution.

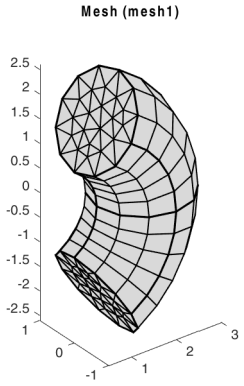


Figure 3-8: 3D prism mesh created with the Sweep feature.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('pos', [2, 0]);
```

```
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.set('angle2', '60').set('angle1', -60);
rev1.selection('input').set({'wp1'});
geom1.run('rev1');
mesh1 = comp1.mesh.create('mesh1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.feature('ftri1').selection.geom(2);
mesh1.feature('ftri1').selection.set(2);
mesh1.runCurrent;
swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection.geom(3);
swe1.selection.add(1);
mesh1.run;
mphmesh(model)
```

EXTRUDING A MESH BY SWEEPING

To generate a 3D prism mesh from the same 2D mesh by extrusion and then to plot it, enter these commands:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('pos', [2, 0]);
ext1 = geom1.feature.create('ext1', 'Extrude');
ext1.selection('input').set({'wp1'});
geom1.runAll;

mesh1 = comp1.mesh.create('mesh1');

ftri1 = mesh1.feature.create('ftri1', 'FreeTri');
ftri1.selection.geom('geom1', 2);
ftri1.selection.set(3);

dis1 = mesh1.feature.create('dis1', 'Distribution');
dis1.selection.set(1);
dis1.set('type', 'predefined');
dis1.set('elemcount', 20);
dis1.set('elemratio', 100);

swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection('sourceface').geom('geom1', 2);
swe1.selection('targetface').geom('geom1', 2);

mesh1.run;
mphmesh(model);
```

The result is shown in Figure 3-9. With the properties `elemcount` and `elemratio` the number and distribution of mesh element layers is controlled in the extruded direction.

	`Distribution` in the *COMSOL Multiphysics Programming Reference Manual* or at the MATLAB prompt: `mphdoc(model.mesh,'Distribution')`

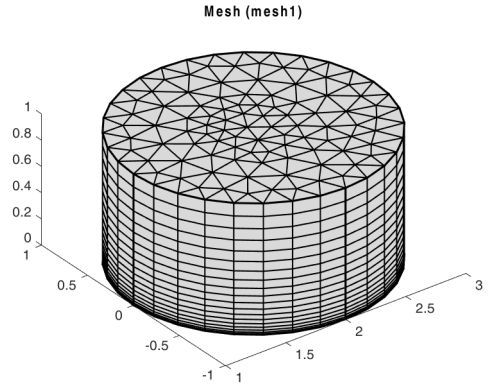


Figure 3-9: Extruded 3D prism mesh.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('pos', [2, 0]);
ext1 = geom1.feature.create('ext1', 'Extrude');
ext1.selection('input').set({'wp1'});
geom1.runAll;
mesh1 = comp1.mesh.create('mesh1');
ftri1 = mesh1.feature.create('ftri1', 'FreeTri');
ftri1.selection.geom('geom1', 2);
ftri1.selection.set(3);
dis1 = mesh1.feature.create('dis1', 'Distribution');
dis1.selection.set(1);
dis1.set('type', 'predefined');
dis1.set('elemcount', 20);
```

```
dis1.set('elemratio', 100);
swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection('sourceface').geom('geom1', 2);
swe1.selection('targetface').geom('geom1', 2);
mesh1.run;
mphmesh(model);
```

COMBINING UNSTRUCTURED AND STRUCTURED MESHES

By specifying selections for the meshing operations, swept meshing can also be combined with unstructured meshing. In this case, start by creating a triangular mesh for domain 2, then sweep the resulting surface mesh through domain 1, as in this example:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 3);
cone1 = geom1.feature.create('cone1', 'Cone');
cone1.set('r', 0.3).set('h', 1).set('ang', 9);
cone1.set('pos', [ 0 0.5 0.5]).set('axis', [-1 0 0]);
geom1.feature.create('blk1', 'Block');

mesh1 = comp1.mesh.create('mesh1');

ftet1 = mesh1.feature.create('ftet1', 'FreeTet');
ftet1.selection.geom('geom1', 3);
ftet1.selection.set(2);

swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection('sourceface').geom('geom1', 2);
swe1.selection('targetface').geom('geom1', 2);

mesh1.run;
mphmesh(model);
```

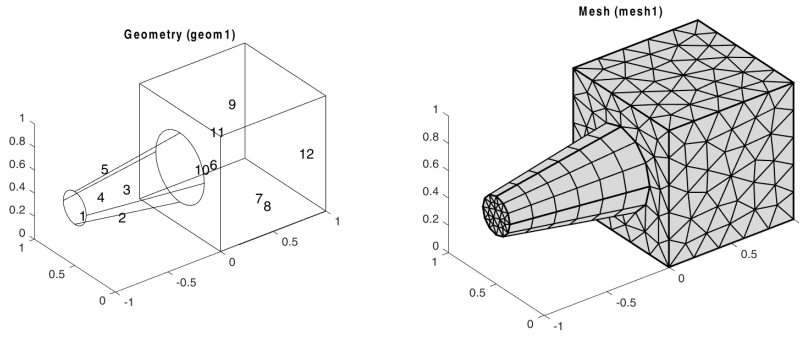


Figure 3-10: Combined structured/unstructured mesh.

The left-hand side plot in Figure 3-10 is obtained with this command:

```
mphgeom(model,'geom1','facemode','off','facelabels','on')
```

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 3);
cone1 = geom1.feature.create('cone1', 'Cone');
cone1.set('r', 0.3).set('h', 1).set('ang', 9);
cone1.set('pos', [ 0 0.5 0.5]).set('axis', [-1 0 0]);
geom1.feature.create('blk1', 'Block');
mesh1 = comp1.mesh.create('mesh1');
ftet1 = mesh1.feature.create('ftet1', 'FreeTet');
ftet1.selection.geom('geom1', 3);
ftet1.selection.set(2);
swe1 = mesh1.feature.create('swe1', 'Sweep');
swe1.selection('sourceface').geom('geom1', 2);
swe1.selection('targetface').geom('geom1', 2);
mesh1.run;
mphmesh(model);
mphgeom(model,'geom1','facemode','off','facelabels','on')
```

CREATING BOUNDARY LAYER MESHES

For 2D and 3D geometries it is also possible to create boundary layer meshes using the `BndLayer` feature. 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

whether the corresponding boundary layer boundaries contain a triangular or a quadrilateral mesh.

If starting with an empty mesh, an initial mesh is automatically created before inserting the 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 this in 2D:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);

geom1 = comp1.geom.create('geom1', 2);
r1 = geom1.feature.create('r1', 'Rectangle');
r1.set('size', [10, 5]);
c1 = geom1.feature.create('c1', 'Circle');
c1.set('pos', [3.5 2.5]);
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'r1'});
dif1.selection('input2').set({'c1'});
geom1.runAll;

mesh1 = comp1.mesh.create('mesh1');

bl1 = mesh1.feature.create('bl1', 'BndLayer');
bl1.feature.create('blp1', 'BndLayerProp');
bl1.feature('blp1').selection.set([2 3 5 6 7 8]);

mesh1.run;
mphmesh(model);
```

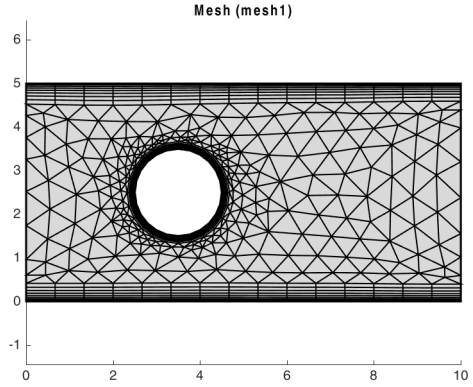


Figure 3-11: Boundary layer mesh based on an unstructured triangular mesh.

It is also possible to insert boundary layers in an existing mesh. Use the following meshing sequence with the geometry sequence from the previous example:

```
bl1.active(false);

fq1 = mesh1.feature.create('fq1', 'FreeQuad');
fq1.selection.set(1);
mesh1.run;
mphmesh(model)

bl1 = mesh1.feature.create('bl2', 'BndLayer');
bl1.feature.create('blp2', 'BndLayerProp');
bl1.feature('blp2').selection.set([2 3 5 6 7 8]);
mesh1.run;
mphmesh(model);
```

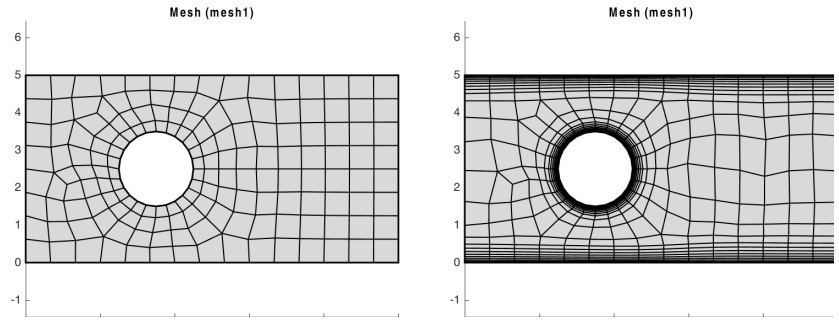


Figure 3-12: Initial unstructured quad mesh (left) and resulting boundary layer mesh (right).

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
r1 = geom1.feature.create('r1', 'Rectangle');
r1.set('size', [10, 5]);
c1 = geom1.feature.create('c1', 'Circle');
c1.set('pos', [3.5 2.5]);
dif1 = geom1.feature.create('dif1', 'Difference');
dif1.selection('input').set({'r1'});
dif1.selection('input2').set({'c1'});
geom1.runAll;
mesh1 = comp1.mesh.create('mesh1');
bl1 = mesh1.feature.create('bl1', 'BndLayer');
bl1.feature.create('blp1', 'BndLayerProp');
bl1.feature('blp1').selection.set([2 3 5 6 7 8]);
mesh1.run;
mphmesh(model);
```

```
bl1.active(false);
fq1 = mesh1.feature.create('fq1', 'FreeQuad');
fq1.selection.set(1);
mesh1.run;
mphmesh(model)
bl1 = mesh1.feature.create('bl2', 'BndLayer');
bl1.feature.create('blp2', 'BndLayerProp');
bl1.feature('blp2').selection.set([2 3 5 6 7 8]);
mesh1.run;
mphmesh(model);
```

REFINING MESHES

Given a mesh consisting only of *simplex elements* (lines, triangles, and tetrahedra) you can create a finer mesh using the feature `Refine`. Enter this command to refine the mesh:

```
mesh1.feature.create('ref1', 'Refine');
```

By specifying the property `tri`, either as a row vector of element numbers or a 2-row matrix, the elements to be refined can be controlled. 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 can use adaptive mesh refinement at the solver stage with the `adaption` solver step. See `Adaption` in the *COMSOL Multiphysics Programming Reference Manual*.

COPYING BOUNDARY MESHES

Use the `CopyEdge` feature in 2D and the `CopyFace` feature in 3D to copy a mesh between boundaries.

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 domain:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);

geom1 = comp1.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('r', 0.5).set('pos', [1, 0]);
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.set('angle1', 0).set('angle2', 180);
rev1.selection('input').set({'wp1'});
geom1.run('wp1');

mesh1 = comp1.mesh.create('mesh1');

size1 = mesh1.feature.create('size1', 'Size');
size1.selection.geom('geom1', 1);
size1.selection.set(18);
size1.set('hmax', '0.06');

ftri1 = mesh1.feature.create('ftri1', 'FreeTri');
ftri1.selection.geom('geom1', 2);
ftri1.selection.set(10);

cpf1 = mesh1.feature.create('cpf1', 'CopyFace');
cpf1.selection('source').geom('geom1', 2);
cpf1.selection('destination').geom('geom1', 2);
cpf1.selection('source').set(10);
cpf1.selection('destination').set(1);

sw1 = mesh1.feature.create('sw1', 'Sweep');
sw1.selection('sourceface').geom('geom1', 2);
sw1.selection('targetface').geom('geom1', 2);

mesh1.run;
mphmesh(model);
```

The algorithm automatically determines how to orient the source mesh on the target boundary. The result is shown in Figure 3-13.

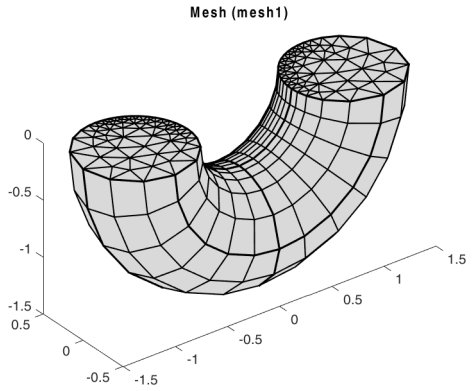


Figure 3-13: Prism element obtained with the CopyFace and Sweep features.

To explicitly control the orientation of the copied mesh, use the `EdgeMap` attribute. The command sequence:

```
em1 = cpf1.feature.create('em1', 'EdgeMap');
em1.selection('srcedge').set(18);
em1.selection('dstedge').set(2);
mesh1.feature.remove('sw1');
mesh1.feature.create('ftet1', 'FreeTet');

mesh1.run;
mphmesh(model);
```

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 domain is then meshed by the tetrahedral mesher, resulting in the mesh in Figure 3-14. In this case it is not possible to create a swept mesh on the domain because the boundary meshes do not match in the sweeping direction.

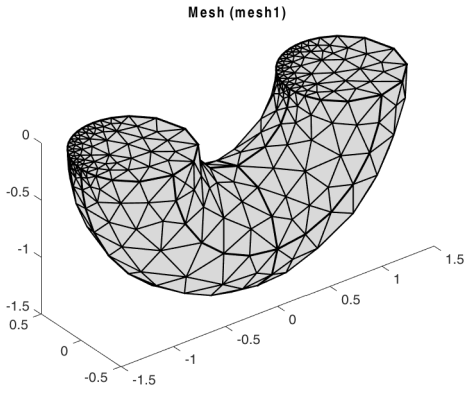


Figure 3-14: Tetrahedral mesh after the use of the CopyFace feature.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 3);
wp1 = geom1.feature.create('wp1', 'WorkPlane');
wp1.set('planetype', 'quick');
wp1.set('quickplane', 'xy');
c1 = wp1.geom.feature.create('c1', 'Circle');
c1.set('r', 0.5).set('pos', [1, 0]);
rev1 = geom1.feature.create('rev1', 'Revolve');
rev1.set('angle1', 0).set('angle2', 180);
rev1.selection('input').set({'wp1'});
geom1.run('wp1');
mesh1 = comp1.mesh.create('mesh1');
size1 = mesh1.feature.create('size1', 'Size');
size1.selection.geom('geom1', 1);
size1.selection.set(18);
size1.set('hmax', '0.06');
ftri1 = mesh1.feature.create('ftri1', 'FreeTri');
ftri1.selection.geom('geom1', 2);
ftri1.selection.set(10);
cpf1 = mesh1.feature.create('cpf1', 'CopyFace');
cpf1.selection('source').geom('geom1', 2);
cpf1.selection('destination').geom('geom1', 2);
cpf1.selection('source').set(10);
cpf1.selection('destination').set(1);
sw1 = mesh1.feature.create('sw1', 'Sweep');
sw1.selection('sourceface').geom('geom1', 2);
sw1.selection('targetface').geom('geom1', 2);
```

```
mesh1.run;
mphmesh(model);
em1 = cpf1.feature.create('em1', 'EdgeMap');
em1.selection('srcedge').set(18);
em1.selection('dstedge').set(2);
mesh1.feature.remove('sw1');
mesh1.feature.create('ftet1', 'FreeTet');
mesh1.run;
mphmesh(model);
```

CONVERTING MESH ELEMENTS

Use the `Convert` feature to convert meshes containing quadrilateral, hexahedral, or prism elements into triangular meshes and tetrahedral meshes. In 2D, the function splits each quadrilateral element into either two or four triangles. In 3D, it converts each prism into three tetrahedral elements and each hexahedral element into five, six, or 28 tetrahedral elements. To control the method used to convert the elements, use the property `splitmethod`.

For additional properties supported, see `Convert` in the *COMSOL Multiphysics Programming Reference Manual* or at the MATLAB prompt: `mphdoc(model.mesh,'Convert')`

This example demonstrates how to convert a quad mesh into a triangle mesh:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('c1', 'Circle');
geom1.feature.create('r1', 'Rectangle');
int1 = geom1.feature.create('int1', 'Intersection');
int1.selection('input').set({'c1' 'r1'});
mesh1 = comp1.mesh.create('mesh1', 'geom1');
mesh1.feature.create('fq1', 'FreeQuad');
mesh1.runCurrent;
mphmesh(model);
mesh1.feature.create('conv1', 'Convert');
mesh1.run;
mphmesh(model);
```

The result is illustrated in the Figure 3-15:

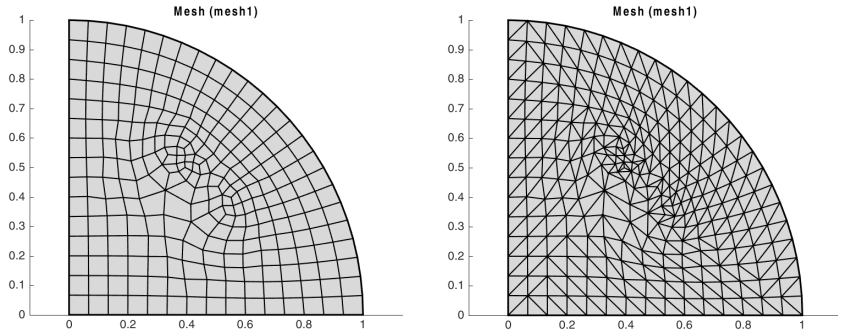


Figure 3-15: Mesh generated by the quad mesher (left) and the mesh after conversion from quad to triangle elements (right).

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('c1', 'Circle');
geom1.feature.create('r1', 'Rectangle');
int1 = geom1.feature.create('int1', 'Intersection');
int1.selection('input').set({'c1' 'r1'});
mesh1 = comp1.mesh.create('mesh1', 'geom1');
mesh1.feature.create('fq1', 'FreeQuad');
mesh1.runCurrent;
mphmesh(model);
mesh1.feature.create('conv1', 'Convert');
mesh1.run;
mphmesh(model);
```

Importing External Meshes and Mesh Objects

It is possible to import meshes to COMSOL Multiphysics using the following formats:

- COMSOL Multiphysics text files (extension `.mphtxt`),
- COMSOL Multiphysics binary files (extension `.mphbin`), and
- NASTRAN® files (extension `.nas` or `.bdf`).

IMPORTING MESHES

To import a mesh stored in a supported format use the `Import` feature. The following commands import and plot a mesh defined in a COMSOL Multiphysics text file:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
comp1.geom.create('geom1', 3);

mesh1 = comp1.mesh.create('mesh1', 'geom1');
imp1 = mesh1.feature.create('imp1', 'Import');
filenamepath = fullfile(COMSOL, 'applications', ...
'COMSOL_Multiphysics','Meshing_Tutorials');
model.modelPath(filenamepath);
imp1.set('filename','mesh_example_1.mphtxt');
mesh1.feature('imp1').importData;

mesh1.run;
mphmesh(model);
```

Where `COMSOL` is the path of root directory where COMSOL Multiphysics is installed. The above command sequence results in Figure 3-16.

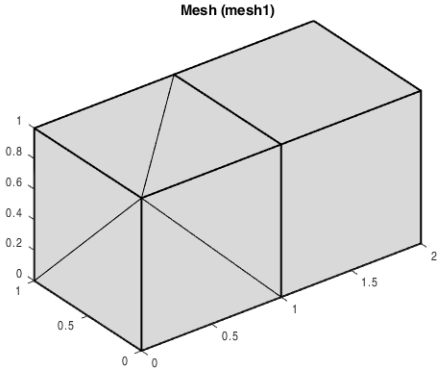


Figure 3-16: Imported mesh.

For additional properties supported, see `Import` in the *COMSOL Multiphysics Programming Reference Manual.*

For a description of the text file format see the *COMSOL Multiphysics Reference Manual.*

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
comp1.geom.create('geom1', 3);
mesh1 = comp1.mesh.create('mesh1');
imp1 = mesh1.feature.create('imp1', 'Import');
```

```
filenamepath = fullfile(COMSOL,'applications',...
'COMSOL_Multiphysics','Meshing_Tutorials');
model.modelPath(filenamepath);
imp1.set('filename','mesh_example_1.mphtxt');
mesh1.feature('imp1').importData;
mesh1.run;
mphmesh(model);
```

Visualizing Mesh Quality

The following commands show how to visualize the mesh quality for a mesh on the unit circle:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('c1', 'Circle');
geom1.runAll;

mesh1 = comp1.mesh.create('mesh1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.run;

meshdset1 = model.result.dataset.create('mesh1', 'Mesh');
meshdset1.set('mesh', 'mesh1');

pg1 = model.result.create('pg1', 2);

meshplot1 = pg1.feature.create('mesh1', 'Mesh');
meshplot1.set('data', 'mesh1');
meshplot1.set('filteractive', 'on');
meshplot1.set('elemfilter', 'quality');
meshplot1.set('tetkeep', 0.25);
mphplot(model,'pg1','rangenum',1);
meshplot1.set('elemfilter','qualityrev');
mphplot(model,'pg1','rangenum',1);
```

These commands display the worst 25% and the best 25% elements in terms of mesh element quality. In Figure 3-17, the triangular mesh elements in the right-hand side plot are more regular than those in the left-hand side 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.

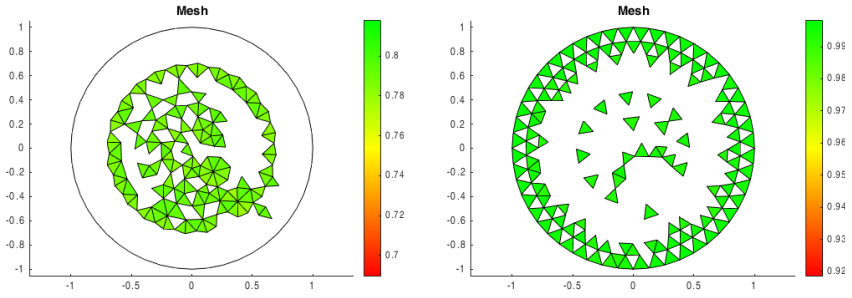


Figure 3-17: Visualizations of the mesh quality: worst 25% (left) and best 25% (right).

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('c1', 'Circle');
geom1.runAll;
mesh1 = comp1.mesh.create('mesh1', 'geom1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.run;
meshdset1 = model.result.dataset.create('mesh1', 'Mesh');
meshdset1.set('mesh', 'mesh1');
pg1 = model.result.create('pg1', 2);
meshplot1 = pg1.feature.create('mesh1', 'Mesh');
meshplot1.set('data', 'mesh1');
meshplot1.set('filteractive', 'on');
meshplot1.set('elemfilter', 'quality');
meshplot1.set('tetkeep', 0.25);
mphplot(model,'pg1','rangenum',1);
meshplot1.set('elemfilter','qualityrev');
mphplot(model,'pg1','rangenum',1);
```

Getting Mesh Statistics Information

Use the function `mphmeshstats` to get mesh statistics and mesh information where `stats` is a structure containing the mesh statistics information. Enter:

```
stats = mphmeshstats(model)
```

The statistics structure has the following fields:

- `meshtag`, the tag of the mesh sequence;
- `geomtag`, the tag of the associated geometry;
- `geometricmodel`, the geometric model used by the mesh sequence;
- `component`, the tag of the component the mesh belongs to;
- `componentgeometricmodel`, the geometric model used by the physics;
- `current`, the current mesh feature tag;
- `isempty`, boolean variable that indicates if the mesh is empty (1) or not(0);
- `hasproblems`, boolean variable that indicates if the mesh contains error or warning nodes (1) or not (0);
- `iscomplete`, boolean variable that indicates if the mesh is built to completion(1) or not(0);
- `secondorderelements`, boolean variable that indicates if the mesh has second order elements(1) or not (0);
- `sdim`, the space dimension;
- `contributing`, the contributing physics and multiphysics feature for physics-controlled mesh;
- `types`, the element types present in the mesh. The element type can be vertex (`vtx`), edge (`edg`), triangle (`tri`), quadrilateral (`quad`), tetrahedra (`tet`), pyramid (`pyr`), prism (`prism`), hexahedra (`hex`). The type can also be of all elements of maximal dimension in the selection (`all`);
- `numelem`, the number of elements for each element type;
- `qualitymeasure`, the quality measure used to evaluate the quality of the mesh.
- `minquality`, the minimum element quality;
- `meanquality`, the mean element quality;
- `qualitydistr`, the distribution of the element quality;
- `minvolume`, the minimum element volume/area;
- `maxvolume`, the maximum element volume/area;
- `volume`, the total volume/area of the mesh;
- `maxgrowthrate`, the maximal growth rate value for the entire selection, regardless of the element type property;
- `meangrowthrate`, the average growth rate value for the entire selection, regardless of the element type property.

The fields `maxgrowthrate` and `meangrowthrate` provide statistics for the entire selection regardless of the element type property.

If several mesh cases are available in the model object, specify the mesh tag:

```
stats = mphmeshstats(model, <meshtag>)
```

Set the number of bins in the quality distribution histogram (`qualitydistr`) with the property `qualityhistogram`:

```
stats = mphmeshstats(model,<meshtag>,'qualityhistogram',<num>)
```

where *<num>* is an integer corresponding to the desired number of bins.

Set the mesh quality measure from for the mesh statistics with the property `qualitymeasure`:

```
stats = mphmeshstats(model,<meshtag>,'qualitymeasure',<quality>)
```

where *<quality>* is one of `'condition'` (condition number), `'growth'` (growth rate), `'maxangle'` (maximum number), `'skewness'` (skewness), `'volcircum'` (volume versus circumradius) or `'vollength'` (volume versus length).

Mesh Element Quality in the *COMSOL Multiphysics Programming Reference Manual*.

GET MESH STATISTICS ON SPECIFIED ENTITY OR SELECTION

Specify the entity where to evaluate mesh quality with the property entity as in the command below:

```
stats = mphmeshstats(model,<meshtag>,'entity',entity)
```

where *entity* is one of `'domain'`, `'boundary'`, `'edge'`, or `'point'`.

Use the selection property to specify the entity number where to get the mesh statistics:

```
stats = mphmeshstats(model,<meshtag>,'entity',entity,...
'selecti on',<selnum>)
```

where *<selnum>* is an integer array defining the selection number.

It is also possible to use the selection property to define a selection node defined in the model:

```
stats = mphmeshstats(model,<meshtag>,'selection',<seltag>)
```

where *<seltag>* is the tag of the desired selection node.

The properties `entity` and `selection` cannot be set if the mesh data information is returned.

RETRIEVE MESH STATISTICS OF A SPECIFIC ELEMENT TYPE

Restrict the mesh statistics information structure a specific element type with the property type:

```
stats = mphmeshstats(model,<meshtag>,'type',type)
```

where *type* is one of the mesh element type available: vertex (`'vtx'`), edge (`'edg'`), triangle (`'tri'`), quad (`'quad'`), tetrahedron (`'tet'`), pyramid (`'pyr'`), prism (`'prism'`) or hexahedron (`'hex'`). *type* can also be a cell array to return the mesh statistics of several mesh element type.

Getting and Setting Mesh Data

The function `mphmeshstats` also returns the mesh data such as element coordinates. Use the function with two output variables to get the mesh data. Enter:

```
[meshstats,meshdata] = mphmeshstats(model)
```

where `meshdata` is a MATLAB structure with the following fields:

- `types`, which contains the element type;
- `vertex`, which contains the mesh vertex coordinates;
- `elem`, which contains the element data information;
- `elementity`, which contains the element entity information for each element type.

Selection and/or Entity properties cannot be set if the mesh data structure is returned.

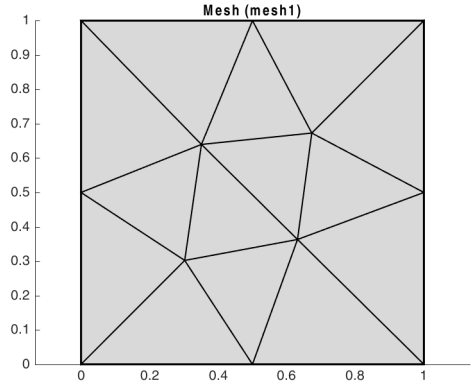
EXTRACT AND CREATE MESH INFORMATION

A mesh can be manually created based on a grid generated in MATLAB. However, before inserting this mesh into the model, a default coarse mesh is generated to get the mesh information, which enables you to understand the requested mesh structure to use with the `createMesh` method. Then a complete mesh can be constructed and stored in the meshing sequence. If the geometry is not empty, the new mesh is checked

to ensure that it matches the geometry. In other words, to create an arbitrary mesh, an empty geometry sequence and a corresponding empty meshing sequence need to be created and the mesh is then constructed on the empty meshing sequence.

Start by creating a 2D model containing a square, and mesh it with triangles:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('sq1', 'Square');
geom1.run;
mesh1 = comp1.mesh.create('mesh1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.feature('size').set('hmax', 0.5);
mesh1.run('ftri1');
mphmesh(model);
```



To get the mesh data information, enter:

```
[meshstats,meshdata] = mphmeshstats(model);

meshdata =
         types: {'edg'  'tri'  'vtx'}
        vertex: [2×12 double]
          elem: {[2×8 int32]  [3×14 int32]  [0 5 7 11]}
    elementity: {[8×1 int32]  [14×1 int32]  [4×1 int32]}
```

The mesh node coordinates are stored in the `vertex` field:

```
vtx = meshdata.vertex

vtx =
```

```
0     0   0.302   0.5   0.351   0   0.631   1   0.673   0.5     1   1
0   0.5   0.302     0   0.639   1   0.363   0   0.672     1   0.5   1
```

In the `elem` field the element information is retrieved, such as the node indices (using a 0 based) connected to the elements:

```
tri = meshdata.elem{2}
tri =
2    2    2    1    2    2    3    8    9    8   10    8    8    8
    1    0    4    4    3    6    7    4    5    9    6    6   10   11
    0    3    1    5    6    4    6    6    4    4    7   10   11    9
```

In the above command, notice that element number 1 is connected to nodes 0, 1, and 2, and element number 2 is connected to nodes 2, 0, and 3.

Then create manually a mesh using a data distribution generated in MATLAB. Enter the command:

```
[x,y] = meshgrid([0 0.5 1], [0 0.5 1]);
coord = [x(:) y(:)]';
```

The node distribution obtained with this command corresponds to the mesh in Figure 3-18.

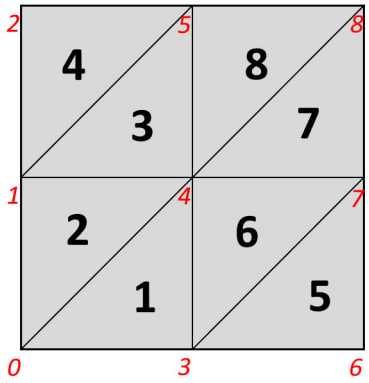


Figure 3-18: Mesh with elements (bold) and nodes (italic) indices.

Table 3-1 lists the nodes and element connectivity in the mesh.

TABLE 3-1: ELEMENT AND NODES CONNECTIVITY

ELEMENT	NODES
1	0, 3, 4
2	0, 1, 4

TABLE 3-1: ELEMENT AND NODES CONNECTIVITY

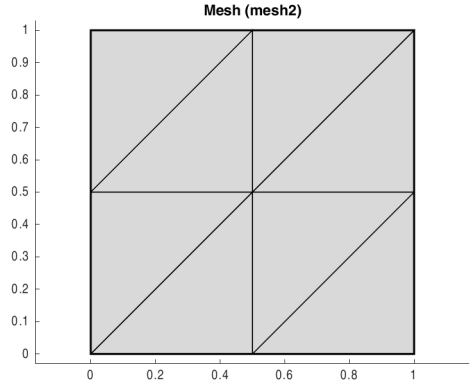
ELEMENT	NODES
3	1, 4, 5
4	1, 2, 5
5	3, 6, 7
6	3, 4, 7
7	4, 7, 8
8	4, 5, 8

To create the elements and nodes connectivity information use the command:

```
new_tri(:,1)=[0;3;4];
new_tri(:,2)=[0;1;4];
new_tri(:,3)=[1;4;5];
new_tri(:,4)=[1;2;5];
new_tri(:,5)=[3;6;7];
new_tri(:,6)=[3;4;7];
new_tri(:,7)=[4;7;8];
new_tri(:,8)=[4;5;8];
```

Assign the element information, node coordinates, and elements connectivity information, into a new mesh. Use the method `createMesh` to create the new mesh:

```
comp2 = model.component.create('comp2', true);
comp2.geom.create('geom2',2);
mesh2 = comp2.mesh.create('mesh2');
mesh2.data.setElem('tri',new_tri)
mesh2.data.setVertex(coord)
mesh2.data.createMesh
mphmesh(model,'mesh2');
```



Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('sq1', 'Square');
geom1.run;
mesh1 = comp1.mesh.create('mesh1');
mesh1.feature.create('ftri1', 'FreeTri');
mesh1.feature('size').set('hmax', 0.5);
mesh1.run('ftri1');
mphmesh(model);
[meshstats,meshdata] = mphmeshstats(model);
vtx = meshdata.vertex
tri = meshdata.elem{2}
[x,y] = meshgrid([0 0.5 1], [0 0.5 1]);
coord = [x(:) y(:)]';
new_tri(:,1)=[0;3;4];
new_tri(:,2)=[0;1;4];
new_tri(:,3)=[1;4;5];
new_tri(:,4)=[1;2;5];
new_tri(:,5)=[3;6;7];
new_tri(:,6)=[3;4;7];
new_tri(:,7)=[4;7;8];
new_tri(:,8)=[4;5;8];
comp2 = model.component.create('comp2', true);
comp2.geom.create('geom2',2);
mesh2 = comp2.mesh.create('mesh2');
mesh2.data.setElem('tri',new_tri);
mesh2.data.setVertex(coord);
mesh2.data.createMesh;
mphmesh(model,'mesh2');
```

Modeling Physics

This section describes how to set up physics interfaces in a model. The physics interface defines the equations that COMSOL solves.

The Physics Interface Syntax

Create a physics interface instance using the syntax:

```
comp = model.component(<ctag>);
phys = comp.physics.create(<phystag>, physint, <geomtag>)
```

where *<phystag>* is a string that identifies the physics interface node. Once defined, you can always refer to a physics interface, or any other feature, by its tag. The string *physint* is the ***constructor name*** of the physics interface. To get the constructor name, the best way is to create a model using the desired physics interface in the GUI and save the model as an M-file. The string *<geomtag>* refers to the geometry where you want to specify the interface.

To add a feature to a physics interface, use the syntax:

```
phys.feature.create(<ftag>, operation)
```

where *<ftag>* is a string that you use to refer to the operation. To set a property to a value in a operation, enter:

```
phys.feature(<ftag>).set(property, <value>)
```

where *<ftag>* is the string that identifies the feature.

There are alternative syntaxes available. See `model.physics()` in the *COMSOL Multiphysics Programming Reference Manual* or type at the MATLAB prompt: `mphdoc(model.physics)`.

To disable or remove a feature node, use the methods `active` or `remove`, respectively.

The command:

```
phys.feature(<ftag>).active(false)
```

disables the feature *<ftag>*.

To activate the feature node you can set the `active` method to true:

```
phys.feature(<ftag>).active(true)
```

To remove a feature from the model, use the method `remove`:

```
phys.feature.remove(<ftag>)
```

EXAMPLE: IMPLEMENT AND SOLVE A HEAT TRANSFER PROBLEM

This example shows how to add a physics interface and set the boundary conditions in the model object.

Start to create a model object including a 3D geometry. The geometry consists in a block with default settings. Enter the following commands at the MATLAB prompt:

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);

geom1 = comp1.geom.create('geom1', 3);
geom1.feature.create('blk1', 'Block');
geom1.run;
```

Add a Heat Transfer in Solids interface to the model:

```
phys = comp1.physics.create('ht', 'HeatTransfer', 'geom1');
```

The tag of the interface is `ht`. The interface constructor is `HeatTransfer`. The physics is defined on geometry `geom1`.

The physics interface automatically creates a number of default features. To examine these, enter:

```
comp1.physics('ht')
ans =
Type: Heat Transfer in Solids
Tag: ht
Identifier: ht
Operation: HeatTransfer
Child nodes: solid1, init1, ins1, idi1, os1, cib1
```

The physics method has the following child nodes: `solid1`, `init1`, `ins1`, `idi1`, `os1`, and `cib1`. These are the default features that come with the Heat Transfer in Solids

interface. The first feature, `solid1`, consists of the heat balance equation. Confirm this by entering:

```
solid = phys.feature('solid1')
ans =
Type: Solid
Tag: solid1
Operation: SolidHeatTransferModel
```

The settings of the `solid1` feature node can be modified, for example, to manually set the material property. To change the thermal conductivity to 400 W/(m*K) enter:

```
solid.set('k_mat', 1, 'userdef');
solid.set('k', 400);
```

The Heat Transfer in Solids interface has features you can use to specify domain or boundary settings. For example, to add a heat source of 10^5 W/m^3 in the study domain, enter the commands:

```
hs = phys.feature.create('hs1', 'HeatSource', 3);
hs.selection.set(1);
hs.set('Q', 1, 1e5);
```

To create a temperature boundary condition on boundaries 3, 5, and 6, enter:

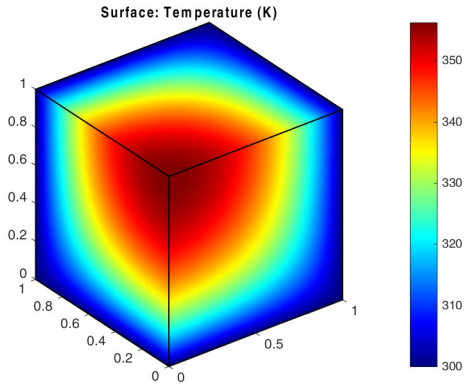
```
temp = phys.feature.create('temp1', 'TemperatureBoundary', 2);
temp.selection.set([3 5 6]);
temp.set('T0', 1, '300[K]');
```

Then add a mesh and a study feature and compute the solution:

```
comp1.mesh.create('mesh1');
std = model.study.create('std1');
std.feature.create('stat', 'Stationary');
std.run
```

To visualize the solution, create a 3D surface plot group, which is displayed in a MATLAB figure with the function `mphplot`:

```
pg = model.result.create('pg1', 'PlotGroup3D');
pg.feature.create('surf1', 'Surface');
mphplot(model,'pg1','rangenum',1)
```



Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 3);
geom1.feature.create('blk1', 'Block');
geom1.run;
phys = comp1.physics.create('ht', 'HeatTransfer', 'geom1');
comp1.physics('ht')
solid = phys.feature('solid1')
solid.set('k_mat', 1, 'userdef');
solid.set('k', 400);
hs = phys.feature.create('hs1', 'HeatSource', 3);
hs.selection.set(1);
hs.set('Q', 1, 1e5);
temp = phys.feature.create('temp1', 'TemperatureBoundary', 2);
temp.selection.set([3 5 6]);
temp.set('T0', 1, '300[K]');
comp1.mesh.create('mesh1');
std = model.study.create('std1');
std.feature.create('stat', 'Stationary');
std.run
pg = model.result.create('pg1', 'PlotGroup3D');
pg.feature.create('surf1', 'Surface');
mphplot(model,'pg1','rangenum',1)
```

Getting the Geometric Model Defined for the Physics

In case of at least one meshing sequence defines its own geometric model, you have the possibility to change the geometric model defining the physics. As geometric

models can have different topology, you may want to retrieve the component geometric model in order to set up the physics.

The functions mphgeominfo, mphmeshstats or mphcomponentinfo return a structure with a field describing the geometric model used by the physics.

The Material Syntax

In addition to changing material properties directly inside the physics interfaces, materials available in the entire model can also be created. Such a material can be used by all physics interfaces in the model.

Create a material using the syntax:

```
model.component(<ctag>).material.create(<mattag>)
```

where `<mattag>` is a string that you use to refer to a material definition.

A *Material* is a collection of material models, where each material model defines a set of material properties, material functions, and model inputs. To add a material model, use the syntax:

```
mat.materialmodel.create(<mtag>)
```

where `mat` is a link to a material node. The string `<mtag>` refers to the material model.

To define material properties for the model, set the property value pairs by entering:

```
mat.materialmodel(<mtag>).set(property, <value>)
```

COPYING MATERIAL DATA FROM ANOTHER MODEL

Instead of creating the material model from scratch you can copy the material node from another model object, to proceed use the command:

```
comp.material.insert(modelMat, <mattag>, '')
```

where `comp` is the link to the component, `modelMat` the model MPH-file name that contains the source material node and `<mattag>` the tag of the material node to import.

EXAMPLE: CREATE A MATERIAL NODE

The section Example: Implement and Solve a Heat Transfer Problem shows how to change a material property inside a physics interface. This example shows how to define a material available globally in the model. These steps assume that the previous example has been followed. Enter:

```
mat = model.component('comp1').material.create('mat1');
```

The material automatically creates a material model, `def`, which can be used to set up basic properties. For example, use it to define the density and the heat capacity:

```
mat.materialmodel('def').set('density', 400);
mat.materialmodel('def').set('heatcapacity', 2e3);
```

To use the defined material in a model, set the `solid1` feature to use the material node. Enter:

```
solid.set('k_mat',1,'from_mat');
```

`model.material()` in the *COMSOL Multiphysics Programming Reference Manual* or type at the MATLAB prompt: `mphdoc(model.material)`.

Modifying the Equations

The equation defining the physics node can be edited with the method `featureInfo('info')` applied to a feature of the physics node `physics(<phystag>).feature(<ftag>)`, where *<phystag>* and *<ftag>* identify the physics interface and the feature, respectively:

```
info = phystag.feature(<ftag>).featureInfo('info');
```

Use the method `getInfoTable(type)` to return the tables available in the Equation View node:

```
infoTable = info.getInfoTable(type);
```

where *type* defines the type of table to return. It can have the value `'Weak'` to return the weak form equations, `'Constraint'` to return the constraint types table, or `'Expression'` to return the variable expressions table.

EXAMPLE: ACCESS AND MODIFY THE EQUATION WEAK FORM

This example continues from the Example: Implement and Solve a Heat Transfer Problem and modifies the model equation.

To retrieve information about the physics interface create an info object:

```
ht = model.component('comp1').physics('ht');
info = ht.feature('solid1').featureInfo('info');
```

From the `info` object access the weak form equation by entering:

```
infoTable = info.getInfoTable('Weak')
```

This returns a string variable that contains both the name of the weak equation variable and the equation of the physics implemented in the weak form as in below:

```
infoTable =

[] 'ht.streamline' 'root.comp1.ht…' '4' 'Spatial' 'Domain 1' '3'
[] '(ht.dfluxx*t…' 'root.comp1.ht…' '4' 'Spatial' 'Domain 1' '3'
[] '-ht.C_eff*(h…' 'root.comp1.ht…' '4' 'Spatial' 'Domain 1' '3'
```

The output shows that the heat equation is defined at the second row and the weak expression at the second column. Enter:

```
infoTable(2,2)
```

to get the weak equation as a string variable. The result of this command is:

```
ans =
(ht.dfluxx*test(Tx)+ht.dfluxy*test(Ty)+ht.dfluxz*test(Tz))*ht.d
```

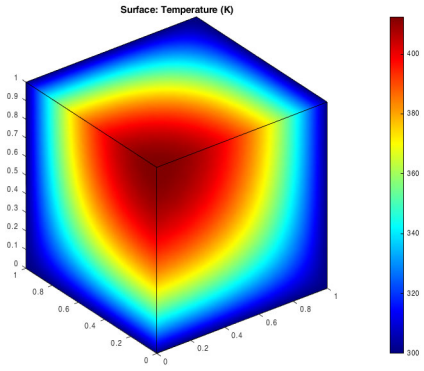
To access the equation in the node `root.comp1.ht.solid1.weak$2`; for example, to modify the equation and lock the expression, run the commands:

```
equExpr = '200[W/(m*K)]*(-Tx*test(Tx)-Ty*test(Ty)-Tz*test(Tz))';
info.set(infoTable(2,3), {equExpr});
```

These commands set the heat conductivity to a constant value directly within the heat balance equation.

You can now compute the solution and display the results in a figure:

```
model.study('std1').run;;
mphplot(model,'pg1','rangenum',1);
```



Adding Global Equations

To add a global equation in the model use the command:

```
model.component(<ctag>).physics.create(<odestag>,
'GlobalEquations')
```

To define the name of the variable to be solved by the global equation, enter:

```
ode.set('name', <idx>, <name>)
```

where `ode` is a link to a Global Equations node and *<idx>* is the index of the global equation, and *<name>* a string with the name of the variable.

Set the expression *<expr>* of the global equation with:

```
ode.set('equation', <idx>, <expr>)
```

where *<expr>* is defined as a string variable.

Initial value and initial velocity can be set with the commands:

```
ode.set('initialValueU', <idx>, <init>)
ode.set('initialValueUt', <idx>, <init_t>)
```

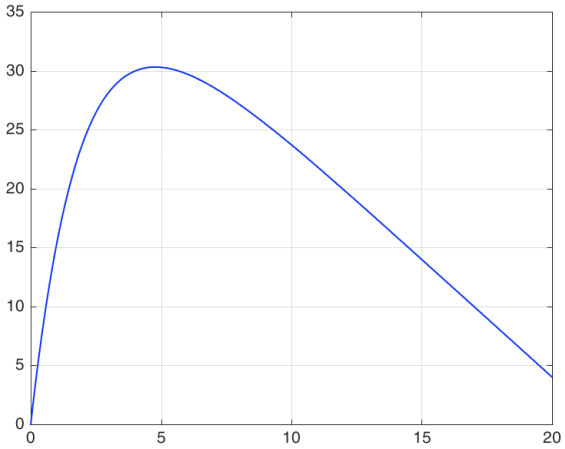
where *<init>* and *<init_t>* are the initial value expression for the variable and its time derivative respectively.

EXAMPLE: SOLVE AN ODE PROBLEM

This example illustrates how to solve the following ODE in a COMSOL model:

$$\ddot{u} + \frac{\dot{u}}{2} + 1 = 0$$

$$u0 = 0$$
$$\dot{u}0 = 20$$



Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
ge = comp1.physics.create('ge', 'GlobalEquations');
ge1 = ge.feature('ge1');
ge1.set('name', 1, 1, 'u');
ge1.set('equation', 1, 1, 'utt+0.5*ut+1');
ge1.set('initialValueU', 1, 1, 'u0');
ge1.set('initialValueUt', 1, 1, 'u0t');
model.param.set('u0', 0);
model.param.set('u0t', 20);
std1 = model.study.create('std1');
std1.feature.create('time', 'Transient');
std1.feature('time').set('tlist', 'range(0,0.1,20)');
std1.run;
model.result.create('pg1', 1);
model.result('pg1').set('data', 'dset1');
model.result('pg1').feature.create('glob1', 'Global');
model.result('pg1').feature('glob1').set('expr', {'comp1.u'});
mphplot(model,'pg1')
```

Defining Model Settings Using External Data File

To use tabulated data from files in a model, use the interpolation function available under the Global Definitions node or the Definitions node of the model.

To add an interpolation function under the Global Definitions node, enter:

```
model.func.create(<functag>, 'Interpolation')
```

If you have several model nodes in your model and you want to attach it to the specified component node *<ctag>*, enter:

```
model.component(<ctag>).func.create(<functag>, 'Interpolation')
```

where *<ctag>* is the tag of the model node to attach the interpolation function.

Then you can interpolate data specified by a table inside the function (default), or specified in an external file.

When using an interpolation table, set the interpolation data for each row of the table with the commands:

```
func.setIndex('table', <t_value>, <i>, 1)
func.setIndex('table', <ft_value>, <i>, 2)
```

where `func` is a link to a function node and *<t_value>* is the interpolation parameter value and *<ft_value>* is the function value. *<i>* is the index (0-based) in the interpolation table. If the number of rows is large then it takes a long time to create the table element by element. Instead store all the `data` as a cell array of strings and set the values all at once:

```
func.set('table', data)
```

To use an external file change the source for the interpolation and specify the file, where *filename* is the name (including the path) of the data file:

```
func.set('source', 'file')
func.set('filename', <filename>)
```

Several interpolation methods are available. Choose the one to use with the command:

```
func.set('interp', method)
```

The string *method* can be set as one of the following alternatives:

- `'neighbor'`, for interpolation according to the nearest neighbor method,
- `'linear'`, for linear interpolation method,
- `'cubicspline'`, for cubic spline interpolation method, or
- `'piecewisecubic'`, piecewise cubic interpolation method.

You can also decide how to handle parameter values outside the range of the input data by selecting an extrapolation method:

```
func.set('extrap', method)
```

The string *method* can be one of these values:

- `'const'`, to use a constant value outside the interpolation data,
- `'linear'`, for linear extrapolation method,
- `'nearestfunction'`, to use the nearest function as extrapolation method, or
- `'value'`, to use a specific value outside the interpolation data.

`model.func()` in the *COMSOL Multiphysics Programming Reference Manual* or type at the MATLAB prompt: `mphdoc(model.func)`.

Access the User-Defined Physics Interface

Using COMSOL with MATLAB, to run a model made with a user-defined physics interface created with the COMSOL Physics Builder you need to save the compiled archive (`.jar`) in your user home folder `.comsol/`*`<version>`*`/archives`, where you replace *`<version>`* with the current version of COMSOL. Any compressed archive (with extension `.jar`) is loaded next time COMSOL with MATLAB starts.

Creating Selections

In this section:

Creating Named Selections

The Selection Node

Use a Selection node to define a collection of geometry entities in a central location in the model. The selection can easily be accessed in physics or mesh features or during results analysis. For example, you can refer collectively to a set of boundaries that have the same boundary conditions, which also have the same mesh size settings.

A selection feature can be one of these types:

- explicit, to include entities explicitly defined by their definitions indices,
- ball, to include entities that fall with a set sphere,
- cylinder, to include entities that fall with a set cylinder, and
- box, to include entities that fall within a set box.

Selection can also be combined by Boolean operations, such as Union, Intersection, and Difference.

SETTING AN EXPLICIT SELECTION

Create an explicit selection with the command:

```
model.component(<ctag>).selection.create(<seltag>, 'Explicit')
```

To specify the domain entity dimension to use in the selection node, enter:

```
sel.geom(sdim)
```

where `sel` is a link to an Explicit Selection node and *`sdim`* is the space dimension that represents the different geometric entities:

- `3` for domains,
- `2` for boundaries/domains,
- `1` for edges/boundaries, and
- `0` for points.

Set the domain entity indices in the selection node with the command:

```
sel.set(<idx>)
```

where *<idx>* is an array of integers that list the geometric entity indices to add in the selection.

Coordinate-Based Selections

DEFINING A BALL SELECTION NODE

The Ball selection node is defined by a centerpoint and a radius. The selection can include geometric entities that are completely or partially inside the ball. The selection can be set up by using either the COMSOL API directly or the `mphselectcoords` function. There are different ways to define the ball selections: Ball Selection Using the COMSOL API or Ball Selection Using mphselectcoords.

Ball Selection Using the COMSOL API

To add a ball selection to a model object enter:

```
model.component(<ctag>).selection.create(<seltag>, 'Ball')
```

To set the coordinates (*<x0>*, *<y0>*, *<z0>*) of the selection centerpoint, enter:

```
sel.set('posx', <x0>)
sel.set('posy', <y0>)
sel.set('posz', <z0>)
```

where `sel` is a link to a Ball Selection node and *<x0>*, *<y0>*, *<z0>* are double values.

Specify the ball radius *<r0>* with the command:

```
sel.set('r', <r0>)
```

where *<r0>* is a double floating-point value.

To specify the geometric entity level, enter:

```
sel.set('entitydim', edim)
```

where *edim* is an integer defining the space dimension value (3 for domains, 2 for boundaries/domains, 1 for edges/boundaries, and 0 for points).

The selection also specifies the condition for geometric entities to be selected:

```
sel.set('condition', condition)
```

where *condition* can be:

- `'inside'`, to select all geometric entities completely inside the ball,
- `'intersects'`, to select all geometric entities that intersect the ball (default),
- `'somevertex'`, to select all geometric entities where at least some vertex is inside the ball, or
- `'allvertices'`, to select all geometric entities where all vertices are inside the ball.

Ball Selection Using mphselectcoords

The function `mphselectcoords` retrieves geometric entities enclosed by a ball.

To get the geometric entities enclosed by a ball of radius `r0`, with its center positioned at (`x0,y0,z0`) enter the command:

```
idx = mphselectcoords(model, <geomtag>, [<x0>,<y0>,<z0>], ...
entitytype,'radius',<r0>)
```

where *<geomtag>* is the tag of geometry where the selection, and *entitytype*, can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

The above function returns the entity indices list. Use it to specify a feature selection or to create an explicit selection as described in Setting an Explicit Selection.

You can also refine the search using several search ball. To do so set the coordinates as a NxM array where N corresponds of the number of point to use and M the space dimension of the geometry as in the command below:

```
idx = mphselectcoords(model, <geomtag>, ...
[<x0>,<y0>,<z0>; <x1>,<y1>,<z1>;...], entitytype)
```

This returns the geometric entity indices that have vertices near both the given coordinates using the tolerance radius.

To include any geometric entities in the selection that have at least one vertex inside the search ball, set the property `include` to `'any'`:

```
idx = mphselectcoords(model, <geomtag>, ...
[<x0>,<y0>,<z0>; <x1>,<y1>,<z1>], entitytype, 'include', 'any');
```

In case the model geometry is finalized as an assembly, you have distinct geometric entities for each part of the assembly (pair). Specify the adjacent domain index to avoid selection of any overlapping geometric entities. Set the `adjnumber` property with the domain index:

```
idx = mphselectcoords(model, <geomtag>, [<x0>,<y0>,<z0>], ...
entitytype,'radius',<r0>,'adjnumber',<idx>)
```

where *<idx>* is the domain index adjacent to the desired geometric entities.

DEFINING A BOX SELECTION NODE

The Box selection node is defined by two diagonally opposite points of a box (in 3D) or rectangle (in 2D). There are different ways to define the box selections: Box Selection Using the COMSOL API or Box Selection Using mphselectbox

Box Selection Using the COMSOL API

This command adds a box selection to the model object:

```
model.component(<ctag>).selection.create(<seltag>, 'Box')
```

To specify the points (*<x0>*, *<y0>*, *<z0>*) and (*<x1>*, *<y1>*, *<z1>*), enter:

```
sel.set('xmin', <x0>)
sel.set('ymin', <y0>)
sel.set('zmin', <z0>)
sel.set('xmax', <x1>)
sel.set('ymax', <y1>)
sel.set('zmax', <z1>)
```

where `sel` is a link to a Box Selection node and *<x0>*, *<y0>*, *<z0>*, *<x1>*, *<y1>*, *<z1>* are double values.

To specify the geometric entities levels use the command:

```
sel.set('entitydim', edim)
```

where *edim* is an integer defining the space dimension value (`3` for domains, `2` for boundaries/domains, `1` for edges/boundaries, and `0` for points).

The selection also specifies the condition for geometric entities to be selected:

```
sel.set('condition', condition)
```

where *condition* can be:

- `'inside'`, to select all geometric entities completely inside the ball,
- `'intersects'`, to select all geometric entities that intersect the ball (default),

- `'somevertex'`, to select all geometric entities where at least some vertex is inside the ball, or
- `'allvertices'`, to select all geometric entities where all vertices are inside the ball.

Box Selection Using mphselectbox

The function `mphselectbox` retrieves geometric entities enclosed by a box (in 3D) or rectangle (in 2D).

To get the geometric entities of type *entitytype* enclosed by the box defined by the points (`x0,y0,z0`) and (`x1,y1,z1`), enter the command:

```
idx = mphselectbox(model,<geomtag>,...
[<x0> <x1>;<y0> <y1>;<z0> <z1>], entitytype)
```

where *<geomtag>* is the geometry tag where the selection is applied, and *entitytype* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

The above function returns the entity indices list. Use it to specify a feature selection or to create an explicit selection as described in Setting an Explicit Selection.

By default the function searches for the geometric entity vertices near these coordinates using the tolerance radius. It returns only the geometric entities that have all vertices inside the box or rectangle. To include any geometric entities in the selection that have at least one vertex inside the search ball, set the property `include` to `'any'`:

```
idx = mphselectbox(model,<geomtag>,...
[<x0> <x1>;<y0> <y1>;<z0> <z1>], entitytype,'include','any')
```

In case the model geometry is finalized as an assembly (pair), you have distinct geometric entities for each part of the assembly. Specify the adjacent domain index to avoid selection of overlapping geometric entities. Set the `adjnumber` property with the domain index:

```
idx = mphselectbox(model,<geomtag>,...
[<x0> <x1>;<y0> <y1>;<z0> <z1>], entitytype, 'adjnumber', <idx>)
```

where *<idx>* is the domain index adjacent to the desired geometric entities.

RETRIEVING POINT COORDINATES USING A SELECTION

Use `mphgetcoords` to retrieve coordinates of the points that belong to a given geometry. Run the command below to get the coordinates of the points that belong to the desired geometric entity:

```
c = mphgetcoords(model,<geomtag>,entitytype,<idx>)
```

where *<geomtag>* is the geometry tag where the selection is applied, *entitytype* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'` and *<idx>* is a integer array containing the geometric entity indices. `c` is a Nx2 double array containing the point coordinates where N is the number of points.

Selection Using Adjacent Geometry

Another approach is to select geometric entities and define the adjacent object. For example, select edges adjacent to a specific domain or boundaries adjacent to a specific point. There are different ways to create an adjacent selection: Adjacent Selection Using the COMSOL API or Adjacent Selection Using mphgetadj

Adjacent Selection Using the COMSOL API

This command creates a selection node using adjacent geometric entities:

```
model.component(<ctag>).selection.create(<seltag>, 'Adjacent')
```

The geometric entity level needs to be specified with the command:

```
sel.set(edim)
```

where `sel` is a link to an Adjacent Selection node and *edim* is an integer defining the space dimension value (`3` for domains, `2` for boundaries/domains, `1` for edges/boundaries, and `0` for points).

The Adjacent selection node only supports the Selection node as an input:

```
sel.set( 'Adjacent')
```

and specify the ball radius *<r0>* with the command:

```
sel.set('input', <seltag>)
```

where *<seltag>* is the tag of an existing Selection node.

Select the level of geometric entities to add in the selection with the command:

```
sel.set('outputdim', edim)
```

where *edim* is an integer defining the space dimension value (`3` for domains, `2` for boundaries/domains, `1` for edges/boundaries, and `0` for points).

If there are multiple domains in the geometry to include in the interior and exterior selected geometric entities, then enter:

```
sel.set('interior', 'on')
sel.set('exterior', 'on')
```

To exclude the interior/exterior, select geometric entities and set the respective property to `'off'`.

Adjacent Selection Using mphgetadj

An alternative to the COMSOL API is to use the function `mphgetadj` to select geometric entities using an adjacent domain.

To get a list of entities of type *entitytype* adjacent to the entity with the index *<adjnumber>* of type *adjtype*, enter:

```
idx = mphgetadj(model, <geomtag>, returntype, adjtype, <adjnumber>)
```

where *<geomtag>* is the tag of geometry where the selection applies, *returntype* is the type of geometry entities whose index are returned and *adjtype* is the type of input geometric entity. The string variables *returntype* and *adjtype* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

If *<adjnumber>* is an array, you can get the list of adjacent entities that connect the input entities the best, to do so enter:

```
[idx,idx_cnct] = mphgetadj(model, <geomtag>, ...
  returntype, adjtype, <adjnumber>)
```

The list returned by the function can be used to specify the selection for a model feature or to create an explicit selection as described in Setting an Explicit Selection.

Displaying Selections

Use the function `mphviewselection` to display the selected geometric entities in a MATLAB figure. This section also includes sections to Specify What to Display with the Selection and Change Display Color and Transparency.

You can either specify the geometric entity index and its entity type or specify the tag of a selection node available in the model.

To display the entity of type *entitytype* with the index *<idx>* enter:

```
mphviewselection(model, <geomtag>, <idx>, entitytype)
```

where *<geomtag>* is the geometry node tag, and *<idx>* is a positive integer array that contains the entity indices. The string *entitytype* can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

If the model contains a selection node with the tag *<seltag>*, this selection can be displayed with the command:

```
mphviewselection(model, <seltag>)
```

To plot the selection in an existing axes, set the property `'parent'` with the axes handle. For instance, the command below displays the selection in the current axis:

```
mphviewselection(model, <seltag>, 'parent', gca)
```

SPECIFY WHAT TO DISPLAY WITH THE SELECTION

- If the selected selection node is a Ball or Box selection, the ball or box selector is display by default, to not show the selector, set the property `'showselector'` to `'off'`.

  ```
  mphviewselection(model, <seltag>, 'showselector', 'off')
  ```

- To deactivate the geometry representation, set the property `'geommode'` to `'off'` as in this command:

  ```
  mphviewselection(model, <seltag>, 'geommode', 'off')
  ```

- The property `'vertexmode'`, `'edgemode'` and `'facemode'` support the value `'on'` or `'off'` in order to render the vertex, the edge and the face respectively in the figure, as in this example line:

  ```
  mphviewselection(model, <seltag>, 'facemode', 'off')
  ```

- To include vertex, edge and face number, set the property `'vertexlabels'`, `'facelabels'` and `'edgelabels'` respectively to `'on'`.

- Change the marker used to represent the vertex with the property `'facemode'`. In the example command below the vertex are represented in the figure with a `'+'` marker instead of the default `'.'`:

  ```
  mphviewselection(model, <seltag>, 'marker', '+')
  ```

- Specify the size of the marker with the property `'edgelabels'`, you can specify an integer value corresponding to the number of pixels.

CHANGE DISPLAY COLOR AND TRANSPARENCY

- To change the color of the edge and the face use the property `'edgecolor'` and `'facecolor'` respectively. Specify the color of the vertex with the property `'markercolor'`. Set the property with a character or using a RGB array. In this example the edges are displayed in blue while the faces are displayed in the color defined by the RGB array (0.5,0.5,0.5):

  ```
  mphviewselection(model, <seltag>, 'edgecolor', 'b',...
  'facecolor', [0.5 0.5 0.5])
  ```

- Specify the color for the selected edge and face with the properties `'edgecolorselected'` and `'facecolorselected'` respectively. Specify the color

of the selected vertex with the property `'markercolorselected'`. Use a character or specify the color by its RGB array. These commands show how to set the edge to a blue color and the face with the color defined by the RGB array (0.5, 0.5,0.5):

```
mphviewselection(model, <seltag>, 'edgecolorselected', 'b',...
'facecolorselected', [0.5 0.5 0.5])
```

- Specify the color for the vertex, edge, and face labels with the properties `'vertexlabelscolor'`, `'edgelabelscolor'` and `'facelabelscolor'` respectively. You can use a character or the RGB array to specify the color.
- Control the transparency of the geometry representation with the property `'facealpha'`. Set the property with a double included between 0 and 1. Using this command the geometry is displayed with a transparency of 50%:

  ```
  mphviewselection(model, <seltag>, 'facealpha', 0.5)
  ```

- Control the transparency of the selector representation with the property `'selectoralpha'`. Set the property with a double included between 0 and 1. Using this command, the selector is displayed with plain color:

  ```
  mphviewselection(model, <seltag>, 'selectoralpha', 1)
  ```

Computing the Solution

This section describes the commands to use to compute the solution at the MATLAB prompt. How to set up and run a study node but also how to set manual solver sequence. This includes the following paragraphs:

- The Study Node
- The Solver Sequence Syntax
- Run the Solver Sequence
- Adding a Parametric Sweep
- Adding a Job Sequence
- Plot While Solving

- Introduction to Solvers and Studies in the *COMSOL Multiphysics Reference Manual*
- Solvers and Study Steps in the *COMSOL Multiphysics Programming Reference Manual*

The Study Node

A study node holds the nodes that define how to solve a model. These nodes are divided into these broad categories:

- Study steps, which determines overall settings suitable for a certain study type,
- Solver sequence, and
- Job configurations for distributed parametric jobs, batch jobs, and cluster computing.

Introduction to Solvers and Studies in the *COMSOL Reference Manual*

Create a study node by using the syntax:

```
model.study.create(<studytag>)
```

where *studytag* is a string that is used to define the study node.

The minimal definition for the study node consists in a study step that define the type of study to use to compute the solution. To add a study step to the study node, use the syntax:

```
study.feature.create(<ftag>, operation)
```

where `study` is a link to the study node. The string `<ftag>` is a string that is defined to refer to the study step. The string *operation* is one of the basic study types, such as `Stationary`, `Transient`, or `Eigenfrequency`, and more.

To specify a property value pair for a study step, enter:

```
study.feature(<ftag>).set(property, <value>)
```

where `<ftag>` is the string identifying the study step.

To generate the default solver sequence associated with the physics solved in the model and compute the solution, run the study node with the command:

```
study.run
```

`model.study()` in the *COMSOL Multiphysics Programming Reference Manual*

The Solver Sequence Syntax

If you do not want to use the default solver sequence created by the study node, you can manually create one. To create a solver sequence, enter:

```
model.sol.create(<soltag>)
```

where `<soltag>` is a string used to refer to the solver sequence associated to a solution object.

A solver sequence has to be connected to a study node, which is done with the command:

```
sol.study(<studytag>)
```

where `<studytag>` is the tag of the study you want to associate the solver sequence `sol`.

A solver sequence also requires the definition of these nodes:

- Study Step, where the study and study step is specified for compiling the equations and computing the current solver sequence;

- Dependent Variables, this node handles settings for the computation of dependent variables, such as initial value and scaling settings but also the dependent variables not solved for; and
- Solver node, where the type of solver to use is specified to compute the solution.

Add the nodes to the solver sequence with the command:

```
sol.feature.create(<ftag>, operation)
```

where `sol` is a link to a solver sequence node. The string *<ftag>* is a string that is defined to refer to the node, for example, a study step. *operation* can be `'StudyStep'`, `'Variables'`, or `'Stationary'`.

To specify a property value pair for a solver feature, enter:

```
feat.set(property, <value>)
```

where `feat` is a link to the solver sequence feature node.

For a list of the operations available for the solver node, see Features Producing and Manipulating Solutions and Solvers and Study Steps, in the *COMSOL Multiphysics Programming Reference Manual.*

Run the Solver Sequence

There are different ways to run the solver sequence:

- run the entire sequence,
- run up to a specified feature, or
- run from a specified feature.

Use the methods `run` or `runAll` to run the entire solver configuration node:

```
model.sol(<soltag>).run
model.sol(<soltag>).runAll
```

You can also use the method `run(`*<ftag>*`)` to run the solver sequence up to the solver feature with the tag *<ftag>*:

```
model.sol(<soltag>).run(<ftag>)
```

When you want to continue solving a sequence, use the method `runFrom(`*<ftag>*`)` to run the solver configuration from the solver feature with the tag *<ftag>*:

```
model.sol(<soltag>).runFrom(<ftag>)
```

Adding a Parametric Sweep

In addition to the study step that defines a study type, you can add a parametric sweep to the study node. This is a study step that does not generate equations and can only be used in combination with other study steps. You can formulate the sequence of problems that arise when some parameters are varied in the model.

To add a parametric sweep to the study node, enter:

```
study.feature.create(<ftag>, 'Parametric')
```

where `study` is a link to a valid study node where to include the parametric sweep defined with the tag *<ftag>*.

To add one or several parameters to the sweep, enter the command:

```
study.feature(<ftag>).setIndex('pname', <pname>, <idx>)
```

where *<pname>* is the name of the parameter to use in the parametric sweep and *<idx>* the index number of the parameter. Set the *<idx>* to `0` to define the first parameter, `1` to define the second parameter, and so on.

Set the list of the parameter values with the command:

```
study.feature(<ftag>).setIndex('plistarr', <pvalue>, <idx>)
```

where *<pvalue>* contains the list of parameter values defined with either a string or with a double array, and *<idx>* is the index number of the parameter and uses the same value as for the parameter name.

If there are several parameters listed in the parametric sweep node, select the type of sweep by entering:

```
study.feature(<ftag>).set('sweeptype', type)
```

where *type* is a string defining the sweep type, it can take either the value `'filled'` or `'sparse'`, referring to all combinations or specified combinations of the parameter values, respectively.

Adding a Job Sequence

In the study node you can define a job sequence such as distributed parametric jobs, batch jobs, and cluster computing. To create a batch node enter:

```
model.batch.create(<batchtag>, type)
```

where *<batchtag>* is the tag of the job sequence and *type* is the type of job to define. It can be either `Parametric`, `Batch`, or `Cluster`.

For a solver sequence you need to attach the job sequence to an existing study node. Enter the command:

```
model.batch(<batchtag>).atach(<studytag>)
```

where *<studytag>* is the tag of the study node.

Each job type, such as parametric, batch, or cluster job, can be defined with specific properties. Use the `set` method to add a property to the batch job:

```
model.batch(<batchtag>).set(property, <value>)
```

You can get the list of the properties in `model.batch()` in the *COMSOL Multiphysics Programming Reference Manual* or type at the MATLAB prompt: `mphdoc(model.batch)`.

To run the batch sequence use the `run` method:

```
model.batch(<batchtag>).run
```

Plot While Solving

With the Plot While Solving functionality you can monitor the development of the computation by updating predefined plots during the computation. Since the plots are displayed on a COMSOL Multiphysics graphics window, start COMSOL with MATLAB using a graphics COMSOL Multiphysics Server.

See the section *Starting COMSOL Multiphysics with MATLAB using the Graphics Server* in the *COMSOL Multiphysics Installation Guide*.

To activate Plot While Solving, enter the command:

```
study.feature(<studysteptag>).set('plot', 'on')
```

where `study` is a link to a valid study node and *<studysteptag>* is a string that refers to the study step.

Specify the plot group to plot by setting the plot group tag:

```
study.feature(<studysteptag>).set('plotgroup', <ptag>)
```

Only one plot group can be plotted during a computation. Use the probe feature instead if you need to monitor several variables.

To activate Plot While Solving for a probe plot, enter the command:

```
study.feature(<studysteptag>).set('probesel', seltype)
```

where *seltype* is the type of probe selection, that can be `'none'`, `'all'`, or `'manual'`.

In case the probe selection is set to manual you have to specify the list of the probe variable to display. Enter the command:

```
study.feature(<studysteptag>).set('probes', <list>)
```

where *<list>* is the a cell array containing the list of the probe to use.

Analyzing the Results

In this section:

- Results Analysis and Plots in the *COMSOL Multiphysics Reference Manual*
- Results in the *COMSOL Multiphysics Programming Reference Manual*

The Plot Group Syntax

Result plots always appear in plot groups, which are added to the model by the `create` method:

```
model.result.create(<pgtag>, sdim)
```

Select the string *<pgtag>* to identify the plot group and the integer *sdim* to set the space dimension (1, 2, or 3) of the group.

To add a plot to a plot group, use the syntax:

```
pg.feature.create(<ftag>, plottype)
```

where `pg` is a link to a plot group node and *plottype* is a string that defines the plot type.

Plots can have different attributes that modify the display. For example, the Deformation attribute deforms the plot according to a vector quantity, the Height Expression attribute introduces 3D height on a 2D table surface plot, and the Filter attribute filters the plot using a logical expression. The type of plot determines which attributes are available. Add an attribute to a plot with the command:

```
pg.feature(<ftag>).feature.create(<attrtag>, attrtype)
```

where `attrtype` is a string that defines the attribute type.

For a list of available plot types and corresponding attribute types, see Results and `model.result()` in the *COMSOL Multiphysics Programming Reference Manual* or type at the MATLAB prompt: `mphdoc(model.result)`.

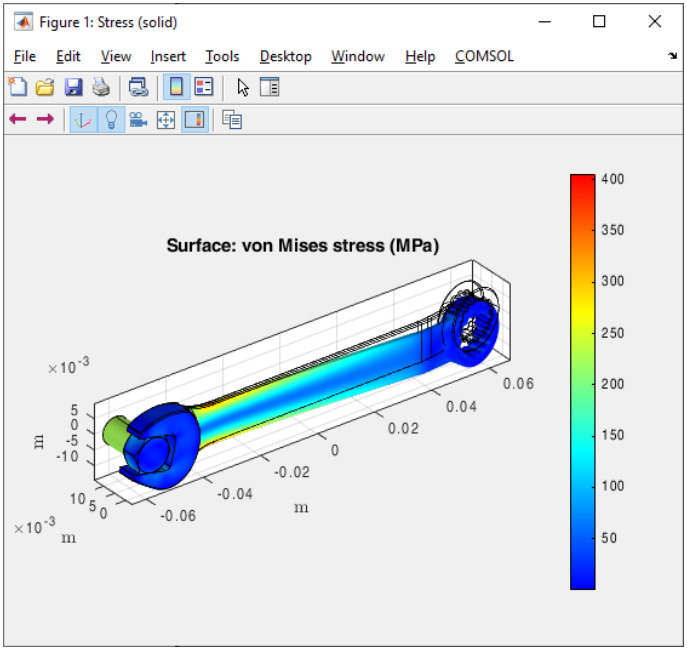
Displaying The Results

There are different commands available to Display Plot Groups, Extract Plot Data, Plot External Data, and to Add Data Plot to Model. A practical example of this is included in Example: Plot mpheval Data.

DISPLAY PLOT GROUPS

Use the command `mphplot` to display a plot group available from the model

```
mphplot(model)
```



The figure contains a toolbar that allows the user to control the use of views, lights, and camera settings set in the model. The COMSOL menu list the plot group available, as well as geometry and mesh plot.

You can specify the plotgroup to display at the command line. For example, to display the plot group *<pgtag>* enter:

```
mphplot(model, <pgtag>)
```

`mphplot` does not support image features, Material Appearance subfeatures, and clipping views.

This renders the graphics in a MATLAB figure window. In addition you can plot results in a COMSOL Multiphysics Graphics window if you start COMSOL with MATLAB using a graphics COMSOL Multiphysics Server. To do this for a plot group *<pgtag>* enter:

```
mphplot(model, <pgtag>, 'server', 'on')
```

See the section *Starting COMSOL Multiphysics with MATLAB using the Graphics Server* in the *COMSOL Multiphysics Installation Guide*.

Another way to plot in a COMSOL Graphics window is to use the `run` method:

```
model.result(<pgtag>).run
```

Mac OS does not support plotting in a COMSOL Graphics window.

The default settings for plotting in a MATLAB figure do not display the color legend. To include the color legend in a figure, use the property `rangenum`:

```
mphplot(model, <pgtag>, 'rangenum', <idx>)
```

where the integer *<idx>* identifies the plot for which the color legend should be displayed.

EXTRACT PLOT DATA

In some situation it can be useful to extract data from a plot, for example, if you need to manually edit the plot as it is allowed in MATLAB. To get a cell array, `pd`, which contains the data for each plot feature available in the plot group *<pgtag>* enter:

```
pd = mphplot(model, <pgtag>)
```

The data fields contained in `pd` returned by mphplot are subject to change. The most important fields are:

- `p`, the coordinates for each point that are used for creating lines or triangles.
- `n`, the normals in each point for the surfaces. These are not always available.
- `t`, contains the indices to columns in `p` of a simplex mesh, each column in `t` representing a simplex.
- `d`, the data values for each point.
- `rgb`, the color values (red, green and blue) entities at each point.

If you don't want to generate a figure when extracting the plot data structure, set the property `createplot` to `off` as in the command below:

```
pd = mphplot(model, <pgtag>, 'createplot','off')
```

This is useful for instance on machine without graphics display support.

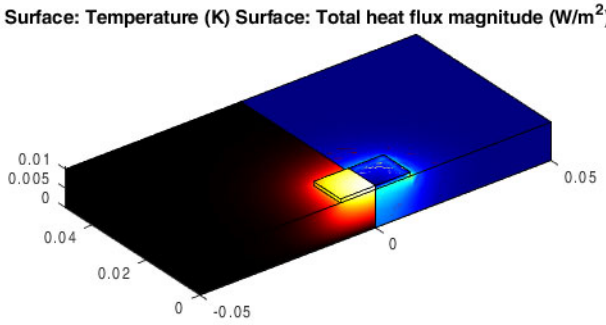
EXAMPLE: EXAMINING THE PLOT DATA

Reuse the first on-line example available for mphplot:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;
model.result.dataset.create('mir', 'Mirror3D');
pg = model.result.create('pg', 'PlotGroup3D');
pg.set('data', 'mir');
surf1 = pg.feature.create('surf1', 'Surface');
surf1.set('colortable', 'Thermal');
mphplot(model,'pg')
surf2 = pg.feature.create('surf2', 'Surface');
surf2.set('data', 'dset1').set('expr', 'ht.tfluxMag');
```

Now plot the result and extract the associated plot data structure:

```
pd = mphplot(model,'pg');
```



`pd` is a cell array containing three plot data structure, the first one corresponds the outline of the geometry, the title, the legend and the color bar information (if any) in the figure. The second and the third plot data structures correspond to the plot defined by the features added to the plot group pg: surf1 and surf2 respectively.

To inspect the outline data of the geometry enter:

```
pd1 = pd{1}{1}
```

```
pd1 =
             p: [3×1310 single]
             t: [2×1048 int32]
           rgb: [3×1 double]
          type: 'line'
      plottype: 'PlotGroup3D'
           tag: 'pg'
preserveaspect: 'on'
         title: 'Surface: Temperature (K) Surface: Total heat ...'
    titlecolor: [3×1 double]
   legendcolor: [3×1 double]
```

To investigate the plot data information of the second surface plot feature (surf2) enter:

```
pd3 = pd{3}{1}
pd3 =
             p: [3x4917 single]
```

```
            n: [3x4917 single]
            t: [3x8964 int32]
            d: [4917x1 single]
  colortable: 'Rainbow'
     cminmax: [0.3193 6.7120e+05]
         rgb: [4917x3 single]
        type: 'surface'
    plottype: 'Surface'
         tag: 'surf2'
```

Code for use with MATLAB®

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;
model.result.dataset.create('mir', 'Mirror3D');
pg = model.result.create('pg', 'PlotGroup3D');
pg.set('data', 'mir');
surf1 = pg.feature.create('surf1', 'Surface');
surf1.set('colortable', 'Thermal');
mphplot(model,'pg')
surf2 = pg.feature.create('surf2', 'Surface');
surf2.set('data', 'dset1').set('expr', 'ht.tfluxMag');
pd = mphplot(model,'pg');
```

PLOT EXTERNAL DATA

Using the function `mphplot` you can also plot data that is specified directly as an input argument. The supported data format is according to the structure provided by the functions `mphplot`, `mpheval` and `mphmesh`.This allows you to plot data that has first been extracted from the model. To plot the structure *<data>*, run the command:

```
mphplot(<data>)
```

If the data structure contains the value of several expressions, set the one to display in the plot with the index property:

```
mphplot(<data>, 'index', <idx>)
```

where *<idx>* is a positive integer that corresponds to the expression to plot.

`mphplot` supports only plotting of data structures that are of the type point, line or surface evaluations from `mpheval`.

Using the `colortable` option to select from several available color tables when visualizing data:

```
mphplot(<data>, 'colortable', colorname)
```

Obtain a list of alternatives for *colorname* from the on-line help by entering:

```
help colortable
```

To disable the mesh displayed together with the data results, set the property `mesh` to `off` as in this command:

```
mphplot(<data>, 'mesh', 'off')
```

ADD DATA PLOT TO MODEL

Use the command `mphaddplotdata` to create a plot group and incorporate the data to a COMSOL Model. To create a plot group in the model using the plotdata `data` and using the plottype *type*, enter the command:

```
mphaddplotdata(model, 'type', type, 'data', data)
```

where *type* is the type of plot, that can be `'arrow'`, `'line'`, `'surface'`, `'annotation'`, `'tube'` or `'point'`.

You can create a plottype using the plotdata in an existing plotgroup using the property plotgroup as in the command below:

```
mphaddplotdata(model, 'type', type, 'data', data,...
'plotgroup', <pgtag> )
```

If you want to remove all existing plottype in the plotgroup enter:

```
mphaddplotdata(model, 'type', type, 'data', data,...
'plotgroup', <pgtag> , 'clearplot', 'on')
```

EXAMPLE: PLOT MPHEVAL DATA

This example extracts COMSOL data at the MATLAB prompt, modifies it and plots the data in a MATLAB figure.

First load the Busbar model from the COMSOL Multiphysics Applications Libraries. Enter:

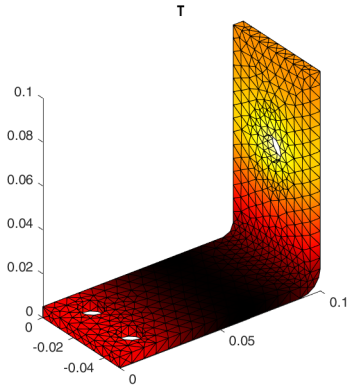
```
model = mphopen('busbar');
```

To extract the temperature and the electric potential field, use the command `mpheval`:

```
dat = mpheval(model,{'T','V'},'selection',1);
```

To display the temperature field, using the `thermal` color table:

```
mphplot(dat,'index',1,'colortable','thermal');
```

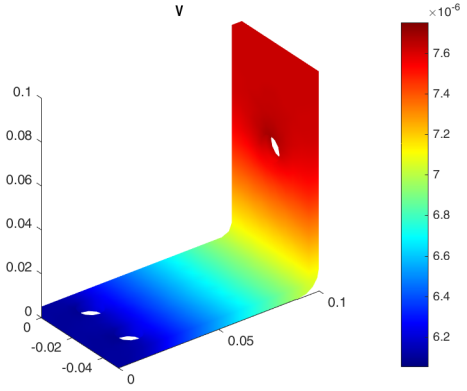


Do a simple scaling of the electric potential then plot it using the default color table:

```
dat.d2 = dat.d2*1e-3;
```

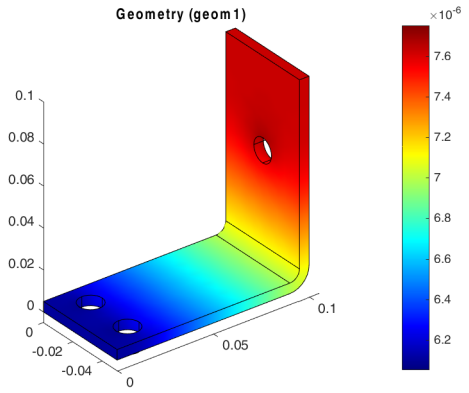
Plot the newly evaluated data without the mesh:

```
mphplot(dat, 'index', 2, 'rangenum', 2, 'mesh', 'off');
```



To emphasize the geometry use the function `mphgeom` to display line plot on the same figure:

```
hold on;
mphgeom(model, 'geom1', 'facemode', 'off')
```



Code for use with MATLAB®

```
model = mphopen('busbar');
dat = mpheval(model,{'T','V'},'selection',1);
mphplot(dat,'index',1,'colortable','thermal');
dat.d2 = dat.d2*1e-3;
mphplot(dat, 'index', 2, 'rangenum', 2, 'mesh', 'off');
hold on;
mphgeom(model, 'geom1', 'facemode', 'off')
```

The Dataset Syntax

Use datasets to make solutions and meshes available for visualization and data analysis. You can create Solution datasets, Mesh datasets, or visualization datasets (such as, for instance, Cut Plane or Edge datasets). While Solution and Mesh datasets are self defined, visualization datasets always refer to an existing Solution dataset.

	See Datasets in the section Commands Grouped by Function of the *COMSOL Multiphysics Programming Reference Manual* to get a list of the available datasets.

	All plots refer to datasets; the solutions are always available as the default dataset.

To create a dataset at the MATLAB prompt, use the command:

```
model.result.dataset.create(<dsettag>, dsettype);
```

where *dsettype* is one of the available dataset types.

- Datasets in the *COMSOL Multiphysics Reference Manual*
- Use of Datasets in the *COMSOL Multiphysics Programming Reference Manual*

The Numerical Node Syntax

Use the numerical node to perform numerical evaluation from within the COMSOL Multiphysics model. Numerical operations such as computing averages, integrations, maxima, or minima of a given expression are available. You can also perform point and global evaluations.

To create a numerical node, enter:

```
model.result.numerical.create(<numtag>, numtype)
```

where *numtype* is the type of operation to be performed by the node.

For a list of the syntax of the numerical results type available, see About Results Commands in the *COMSOL Multiphysics Programming Reference Manual.*

To store the data needed to create a table and associate the table to the numerical node:

```
model.result.table.create(<tabletag>,'Table')
model.result.numerical(<numtag>).set('table',<tabletag>)
```

where *<tabletag>* is the tag of the table where you want to store the data evaluated with the numerical operations defined with the tag *<numtag>*.

To extract the data stored in MATLAB into a table, use the methods `getRealRow` and `getImagRow`, such as:

```
realRow = model.result.table(<tabletag>).getRealRow(<idx>)
imagRow = model.result.table(<tabletag>).getImagRow(<idx>)
```

where *<idx>* is the column index of the table *<tabletag>*.

For data evaluation in MATLAB you can also use the functions `mpheval`, `mphevalpoint`, `mphglobal`, `mphint2`, `mphinterp`, `mphmax`, `mphmean` and `mphmin`.

Extracting Results

Exporting Data

Use the `export` node to generate an animation or to export data to an external file (ASCII format). This section includes information about Animation Export, Data Export, and the Animation Player.

ANIMATION EXPORT

Animations can be defined as two different types: a movie or an image sequence. The movie generates file formats such as GIF (`.gif`), AVI (`.avi`), or flash (`.swf`); the image sequence generates a sequence of images. Make sure COMSOL with MATLAB using a graphics COMSOL Multiphysics Server to enable plot on server.

To learn how to start COMSOL with MATLAB using a graphics COMSOL Multiphysics Server, see the *COMSOL Multiphysics Installation Guide*.

To generate an animation, add an `Animation` node to the `export` method:

```
anim = model.result.export.create(<animtag>, 'Animation')
```

To change the animation type use the `'type'` property according to:

```
anim.set('type', type)
```

where `anim` is a link to an animation node and *`type`* is either `'imageseq'` or `'movie'`.

To set the filename and finally create the animation, enter:

```
anim.set(typefilename, <filenname>)
anim.run
```

In the above, *`typefilename`* depends on the type of animation export: `'imagefilename'` for an image sequence, `'giffilename'` for a gif animation, `'flashfilename'` for a flash animation, and `'avifilename'` for an avi animation.

For a `movie` type animation, it is possible to change the number of frames per second with the command:

```
anim.set('fps', <fps_number>)
```

where *<fps_number>* is a positive integer that corresponds to the number of frames per second to use.

For all animation types you can modify the width and the height of the plot with the `set` method:

```
anim.set('width', <width_px>)
anim.set('height', <height_px>)
```

where, the positive integers *<width_px>* and *<height_px>* are the width and height size (in pixels), respectively, to use for the animation.

DATA EXPORT

In order to save data to an ASCII file, create a `Data` node to the `export` method:

```
model.result.export.create(<datatag>, 'Data')
```

Set the expression *expr* and the file name *filenname*, and run the export:

```
model.result.export(<datatag>).setIndex('expr', <expr>, 0)
model.result.export(<datatag>).set('filename', <filenname>)
```

Set the export data format with the `struct` property:

```
model.result.export(<datatag>).set('struct', datastruct)
```

where *datastruct* can be set to `'spreadsheet'` or `'sectionwise'`.

See Data Formats in the *COMSOL Multiphysics Programming Reference Manual* for details about the data formats used in the exported data files.

To export the data in the specified file, run the `export` node:

```
model.result.export.(<datatag>).run
```

ANIMATION PLAYER

For transient and parametric studies, an animation player can be generated to create interactive animations.

The player displays the figure on a COMSOL Graphics window. Make sure COMSOL with MATLAB is started using a graphics COMSOL Multiphysics Server.

To learn how to start COMSOL with MATLAB using a graphics COMSOL Multiphysics Server, see the *COMSOL Multiphysics Installation Guide*.

To create a player feature node to the model enter the command:

```
model.result.export.create(<playtag>, 'Player')
```

Then associate the player with an existing plot group by setting the `plotgroup` property:

```
model.result.export(<playtag>).set('plotgroup', <pgtag>)
```

where `<pgtag>` refers to the plot group, which is animated in the player.

The default frame number used to generate the animation is 25, you can also specify the number of frame with the command:

```
model.result.export(<playtag>).set('maxframe', <maxnum>)
```

where `<maxnum>` is a positive integer value that corresponds to the maximum number of frames to generate with the player.

Use the `run` method to generate the player:

```
model.result.export(<playtag>).run
```

Generating Report

You can use the Report Generator tool to report and documentate the model created in COMSOL Multiphysics. At the command prompt, the function `mphreport` helps to automatically add a default report template to the current model and generate it in the HRTML or Micorsoft Word® format.

To add a default intermediate report templatein the model, enter the command:

```
mphreport(model, 'action', 'add')
```

This create a new report node with a unique tag, which `'rpt1'` if the model did not contain any report. You can specify the tag to identify the report node using the property `'tag'` as in the command below:

```
mphreport(model, 'action', 'add', 'tag', <rpttag>)
```

where *<rpttag>* is a string.

When creating a default report template you can choose between different template format, enter the command:

```
mphreport(model, 'action', 'add', 'type', <type>)
```

where *<type>* is either `'brief'`, `'intermediate'`, or `'complete'`.

To generate and visualize the report with the tag *<rpttag>*, enter:

```
mphreport(model, 'action', 'run', 'tag', <rpttag>,...
          'filename', <fname>)
```

where *<fname>* is a string defining the name of the report file.

By default the output format is HTML, you can also generate report in the Micorsoft Word or Micorsoft PowerPoint format. To generate a report in the Micorsoft Word format enter:

```
mphreport(model, 'action', 'run', 'tag', <rpttag>,...
          'filename', <fname>, 'format', 'docx')
```

To generate a report in the Micorsoft PowerPoint format enter:

```
mphreport(model, 'action', 'run', 'tag', <rpttag>,...
          'filename', <fname>, 'format', 'pptx')
```

4

Working With Models

This section introduces you to the functionality available for LiveLink™ *for* MATLAB® including the wrapper functions and the MATLAB tools that can be used and combined with a COMSOL Multiphysics® model object.

In this chapter:

Using Workspace Variables in Model Settings

LiveLink™ *for* MATLAB® allows you to define the model properties with MATLAB variables or a MATLAB M-function.

In this section:

The Set and SetIndex Methods

You can use MATLAB® variables to set properties of a COMSOL Multiphysics model. Use the `set` or `setIndex` methods to pass the variable value from MATLAB to the COMSOL model.

THE SET METHODS

Use the `set` method to assign parameter and/or property values. All assignments return the parameter object, which means that assignment methods can be appended to each other.

The basic method for assignment is:

```
something.set(name, <value>)
```

The *name* argument is a string with the name of the parameter/property. The *<value>* argument can for example be a MATLAB integer or double array variable. *<value>* can also be a string, in this case the value or expression is defined within the model object.

When using a MATLAB variable, make sure that the value corresponds to the model unit system. COMSOL can also take care of the unit conversation automatically; in this case convert the MATLAB integer/double variable to a string variable and use the `set` method as:

```
something.set(property, [num2str(<value>)'[unit]'])
```

where is the *unit* you want to set the `value` property.

THE SETINDEX METHODS

Use the `setIndex` method to assign values to specific indices (0-based) in an array or matrix property. All assignment methods return the parameter object, which means that assignment methods can be appended to each other:

```
something.setIndex(name, <value>, <index>)
```

The ***name*** argument is a string with the name of the property, `<value>` is the value to set the property, which can be a MATLAB variable value or a string, and `<index>` is the index in the property table.

When using a MATLAB variable make sure that the value corresponds to the model unit system. COMSOL can automatically take care of the unit conversation; in this case converting the MATLAB integer/double variable to a string variable and using the `set` method as:

```
something.setIndex(name, [num2str(<value>)'[unit]'], <index>)
```

where `[unit]` is the *unit* you want to set the value property.

Using a MATLAB® Function to Define Model Properties

Use MATLAB® Function to define the model property. The function can either be declared within the model object or called at the MATLAB prompt.

CALLING MATLAB FUNCTIONS WITHIN THE COMSOL MODEL OBJECT

LiveLink™ *for* MATLAB® enables you to declare a MATLAB M-function directly from within the COMSOL Multiphysics model object. This is typically the case if you want to call a MATLAB M-function from the COMSOL Desktop. The function being declared within the model object accepts any parameter, variable, or expression arguments defined in the COMSOL model object. However, to use a variable defined at the MATLAB prompt, the variable has to be transferred first in the COMSOL model as a parameter, for example (see how to set a MATLAB variable in the COMSOL model in The Set and SetIndex Methods).

The function is evaluated any time the model needs to be updated. The model object cannot be called as an input argument of the M-function.

Calling External Functions

CALLING MATLAB FUNCTIONS AT THE MATLAB PROMPT

Use a MATLAB function to define a model property with the `set` method:

```
feature.set(property, myfun(<arg>))
```

where *`myfun()`* is an M-function defined in MATLAB.

The function is called only when the command is run at the MATLAB prompt. The argument of the function *`<arg>`* called can be MATLAB variables. To include an expression value from the model object, first extract it at the MATLAB prompt, as described in Extracting Results.

The function *`myfun()`* accepts the model object `model` as an input argument as any MATLAB variable.

Extracting Results

Use LiveLink™ *for* MATLAB® to extract at the MATLAB prompt the data computed in the COMSOL Multiphysics® model. A suite of wrapper functions is available to perform evaluation operations at the MATLAB prompt.

In this section:

Extracting Data at Arbitrary Points

At the MATLAB® prompt, the function `mphinterp` evaluates the result at arbitrary points. To evaluate an expression at specific point coordinates, call the function `mphinterp` as in the command:

```
[d1,...] = mphinterp(model,{'e1',...},'coord',<coord>)
```

where *e1,...* are the COMSOL Multiphysics expressions to evaluate, *<coord>* the evaluation point coordinates defined with a NxM double array, N the space dimension of the evaluation domain, and M is the number of evaluation points. The output `d1,...` is a PxM double array, where P is the length of the inner solution. If an evaluation point is outside the expressions definition domain the output value is NaN.

Alternatively, specify the evaluation coordinates using a selection dataset:

```
data = mphinterp(model, <expr>, 'dataset', <dsettag>)
```

where *<dsettag>* is a selection dataset tag defined in the model, for example, Cut point, Cut Plane, Revolve, and so forth. *<dsettag>* can also be a mesh dataset tag, in this case the evaluation is performed on the geometric mesh vertices.

The rest of this section has additional information for the function `mphinterp`:

- Specify the Evaluation Data
- Output Format
- Small-Signal Analysis, Prestressed Analysis, and Harmonic Perturbation Settings
- Specify the Evaluation Quality
- Other Evaluation Properties

SPECIFY THE EVALUATION DATA

The function `mphinterp` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'dataset',<dsettag>)
  ```

  <dsettag> is the tag of a solution dataset or a mesh dataset. The default value is the current solution dataset of the model. When a mesh dataset is specified the expression *<expr>* can only be geometry or mesh expression.

- `selection`, specify the domain selection for evaluation:

  ```
  data =
  mphinterp(model,<expr>,'coord',<coord>,'selection',<seltag>)
  ```

  where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is All domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `edim`, specify the element dimension for evaluation:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'edim',edim)
  ```

  where *edim* is one of the strings `'point'`, `'edge'`, `'boundary'` or `'domain'`. One can also use numerical values instead, which in 3D are the values from 0 to 3. The default settings correspond to the model geometry space dimension. When using a lower space dimension value, make sure that the evaluation point coordinates dimension has the same size.

- `ext`, specify extrapolation control value to define how much outside the mesh the interpolation searches. This ensures you return data for points that are outside the geometry:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'ext',<ext>)
  ```

  where *<ext>* is a double value corresponding to the search distance as a scale in terms of the local element size. The default value is 0.1.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,solnum',<solnum>)
  ```

  where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string:`'end'` or `'all'` to evaluate the expression for the last inner solution, or all inner solution respectively. By default the evaluation is performed on all inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,...
  'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution respectively. The default settings use the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

- `phase`, specify the phase in degrees:

  ```
  data = mphinterp(model,<expr>,'coord',<coord>,'phase',<phase>)
  ```

  where *<phase>* is a double value.

OUTPUT FORMAT

The function `mphinterp` returns in the MATLAB workspace a double array. It also supports other output formats.

To evaluate several expressions at once, make sure that the same number of output variables are defined as there are expressions specified:

```
[d1,...] = mphinterp(model,{'e1',...},'coord',<coord>)
```

To extract the unit of the evaluated expression, define an extra output variable:

```
[data, unit] = mphinterp(model,<expr>,'coord',<coord>)
```

with `unit` is a 1xN cell array where N is the number of expressions to evaluate.

Returns only the real part in the data evaluation with the property `complexout`:

```
data = mphinterp(model,<expr>,'coord',<coord>,'complexout','off')
```

To disable the error message when all evaluation points are outside the geometry, set the property `coorderr` to `off`:

```
data = mphinterp(model,<expr>,'coord',<coord>,'coorderr','off')
```

the output `data` for evaluation points will only contains NaN.

SMALL-SIGNAL ANALYSIS, PRESTRESSED ANALYSIS, AND HARMONIC PERTURBATION SETTINGS

For solutions with a stored linearization point, such as harmonic perturbation, small-signal analysis, or prestressed analysis you can specify the evaluation method. Use the `mphinterp` function with the property `evalmethod`:

```
data = mphinterp(model,<expr>,'coord',<coord>,'evalmethod',method)
```

where *method* can be one of the following value:

- `'harmonic'`, for harmonic perturbation analysis.
- `'linpoint'`, the expression is evaluated by taking the values of any dependent variables from the linearization point of the solution.
- `'lintotal'`, the expression is evaluated by adding the linearization point and the harmonic perturbation and taking the real part of this sum.
- `'lintotalavg'`, this is the same as evaluating using the `lintotal` property and then averaging over all phases of the harmonic perturbation.
- `'lintotalrms'`, this is the same as evaluating using the `lintotal` property and then taking the RMS over all phases of the harmonic perturbation.
- `'lintotalpeak'`, this is the same as evaluating using the `lintotal` property solution and then taking the maximum over all phases of the harmonic perturbation.

If the property `evalmethod` is set to `harmonic`, you specify whether the expression should be linearized or not with the property `differential` as shown below:

```
data = mphinterp(model,<expr>,'coord',<coord>,...
'evalmethod','harmonic','differential',diffvalue)
```

The default property value settings (`'on'`) evaluates the differential of the expression with respect to the perturbation at the linearization point. If *`diffvalue`* is `off`, it evaluates the expression by taking the values of any dependent variables from the harmonic perturbation part of the solution.

SPECIFY THE EVALUATION QUALITY

With the property `recover`, you can specify the accurate derivative recovery:

```
data = mphinterp(model,<expr>,'coord',<coord>,'recover',recover)
```

where *`recover`* is either `'ppr'`, `'pprint'`, or `'off'` (the default). Set the property to `ppr` to perform recovery inside domains or set to `pprint` to apply recovery to all domain boundaries. Because the accurate derivative processing takes time, the property is disabled by default.

OTHER EVALUATION PROPERTIES

Set the `unit` property to specify the unit of the evaluation:

```
data = mphinterp(model,<expr>,'coord',<coord>,'unit',<unit>)
```

where *`unit`* is a cell array with the same size as *`expr`*.

To not use complex-value functions with real inputs, use the property `complexfun`:

```
data = mphinterp(model,<expr>,'coord',<coord>,'complexfun','off')
```

The default value uses complex-value functions with real inputs.

Use the property `matherr` to return an error for undefined operations or expressions:

```
data = mphinterp(model,<expr>,'coord',<coord>,'matherr','on')
```

Evaluating a Minimum of Expression

Use the function `mphmin` to evaluate the minimum of a given expression over an inner solution list.

To evaluate the minimum of the COMSOL expressions *`e1,...`* use the command `mphmin`:

```
[d1,...] = mphmin(model,{'e1',...},edim)
```

where *`edim`* is a string to define the element entity dimension: `'volume'`, `'surface'`, or `'line'`. *`edim`* can also be a positive integer (`3`, `2`, or `1` respectively). The output variables `d1,...` are an NxP array where N is the number of inner solutions and P the number of outer solutions.

The rest of this section has additional information for the function `mphmin`:

- Specify the Evaluation Data
- Output Format

SPECIFY THE EVALUATION DATA

The function `mphmin` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  ```
  data = mphmin(model,<expr>,edim,'dataset',<dsettag>)
  ```

  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model.

- `selection`, specify the domain selection for evaluation:

  ```
  data = mphmin(model,<expr>,edim,'selection',<seltag>)
  ```

  where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

  ```
  data = mphmin(model,<expr>,edim,'solnum',<solnum>)
  ```

  where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution or all inner solutions, respectively. By default the evaluation is performed on all inner solutions.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  data = mphmin(model,<expr>,edim,'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer array corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution, respectively. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

```
data = mphmin(model,<expr>,edim,'t',<time>)
```

where *<time>* is a double array. The default value corresponds to all the stored time steps.

- In case of data series, such as from a parametric or a transient study, an operation can be applied. To perform data series operation use the function `mphmin` as in the command below:

  ```
  data = mphmin(model,<expr>,edim,'dataseries',<dataoperation>)
  ```

  where *<dataoperation>* can be one of the following value: `'none'` (no operation performed), `'average'` (to compute average of the selected series), `'integral'` (to integrate over series), `'maximum'` (to evaluate the maximum over series), `'minimum'` (to evaluate the minimum), `'rms'` (to compute the root mean square), `'stddev'` (to compute the standard deviation), or `'variance'` (to compute the variance).

- To get the position of the minimum, you can set the property `position` to `on` as in the command below:

  ```
  data = mphmin(model,<expr>,edim,'position', on)
  ```

  where `data(:,1)` contains the minimum value of the expression for each solution step, and `data(:,2:Sdim+1)` returns the position of the minimum, with `Sdim` the space dimension number.

OUTPUT FORMAT

The function `mphmin` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable:

```
[data,unit] = mphmin(model,<expr>,edim)
```

where `unit` is a 1xN cell array, and N is the number of expressions to evaluate.

By default `mphmin` returns the results as a squeezed singleton. To get the full singleton set the `squeeze` property to `off`:

```
data = mphmin(model,<expr>,edim,'squeeze','off')
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphmin(model,<expr>,edim,'matrix','off')
```

Evaluating a Maximum of Expression

Use the function `mphmax` to evaluate the maximum of a given expression over an inner solution list.

To evaluate the maximum of the COMSOL Multiphysics expressions *e1,...* use the command:

```
[d1,...] = mphmax(model,{'e1',...},edim)
```

where *edim* is a string to define the element entity dimension: `'volume'`, `'surface'`, or `'line'`. *edim* can also be a positive integer (`3`, `2`, or `1` respectively). The output variables `d1,...` are an NxP array where N is the number of inner solutions and P the number of outer solutions.

The rest of this section has additional information for the function `mphmax`:

- Specify the Evaluation Data
- Output Format

SPECIFY THE EVALUATION DATA

The function `mphmax` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  ```
  data = mphmax(model,<expr>,edim,'dataset',<dsettag>)
  ```

  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model.

- `selection`, specify the domain selection for evaluation:

  ```
  data = mphmax(model,<expr>,edim,'selection',<seltag>)
  ```

  where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

```
data = mphmax(model,<expr>,edim,'solnum',<solnum>)
```

where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution or all inner solutions, respectively. By default the evaluation is performed on all inner solutions.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  data = mphmax(model,<expr>,edim,'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer array corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution, respectively. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphmax(model,<expr>,edim,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

- In case of data series, such as from a parametric or a transient study, an operation can be applied. To perform data series operation use the function `mphmax` as in the command below:

  ```
  data = mphmax(model,<expr>,edim,'dataseries', <dataoperation>)
  ```

  where *<dataoperation>* can be one of the following value: `'none'` (no operation performed), `'average'` (to compute average of the selected series), `'integral'` (to integrate over series), `'maximum'` (to evaluate the maximum over series), `'minimum'` (to evaluate the minimum), `'rms'` (to compute the root mean square), `'stddev'` (to compute the standard deviation), or `'variance'` (to compute the variance).

- To get the position of the maximum, you can set the property `position` to `on` as in the command below:

  ```
  data = mphmax(model,<expr>,edim,'position', on)
  ```

  where `data(:,1)` contains the maximum value of the expression for each solution step, and `data(:,2:Sdim+1)` returns the position of the maximum, with `Sdim` the space dimension number.

OUTPUT FORMAT

The function `mphmax` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable:

```
[data,unit] = mphmax(model,<expr>,edim)
```

where `unit` is a 1xN cell array and N is the number of expressions to evaluate.

By default `mphmax` returns the results as a squeezed singleton. To get the full singleton set the `squeeze` property to `off`:

```
data = mphmax(model,<expr>,edim,'squeeze','off')
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphmax(model,<expr>,edim,'matrix','off')
```

Evaluating an Integral

Evaluate an integral of expression with the function `mphint2`.

To evaluate the integral of the expression over the domain with the highest space domain dimension call the function `mphint2` as in this command:

```
[d1,...] = mphint2(model,{'e1',...},edim)
```

where *e1,...* are the expressions to integrate. The values `d1,...` are returned as a 1xP double array, with P the length of inner parameters. *edim* is the integration dimension, which can be `'line'`, `'surface'`, `'volume'`, or an integer value that specifies the space dimension (1, 2, or 3).

The rest of this section has additional information for the function `mphint2`:

- Specify the Integration Data
- Output Format
- Specify the Integration Settings

SPECIFY THE INTEGRATION DATA

The function `mphint2` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the integration:

  ```
  data = mphint2(model,<expr>,edim,'dataset',<dsettag>)
  ```

  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model.

- `selection`, specify the integration domain:

```
data = mphint2(model,<expr>,edim,'selection',<seltag>)
```

where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

  ```
  data = mphint2(model,<expr>,edim,'solnum',<solnum>)
  ```

  where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution, or all inner solutions, respectively. By default the evaluation is performed on all inner solutions.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  data = mphint2(model,<expr>,edim,'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution respectively. The default settings use the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphint2(model,<expr>,edim,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

- In case of data series, such as from a parametric or a transient study, an operation can be applied. To perform data series operation use the function `mphint2` as in the command below:

  ```
  data = mphint2(model,<expr>,edim,'dataseries',<dataoperation>)
  ```

  where *<dataoperation>* can be one of the following value: `'none'` (no operation performed), `'average'` (to compute average of the selected series), `'integral'` (to integrate over series), `'maximum'` (to evaluate the maximum over series), `'minimum'` (to evaluate the minimum), `'rms'` (to compute the root mean square), `'stddev'` (to compute the standard deviation), or `'variance'` (to compute the variance).

OUTPUT FORMAT

The function `mphint2` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable:

```
[data,unit] = mphint2(model,<expr>,edim)
```

with `unit` is a 1xN cell array where N is the number of expressions to evaluate.

By default `mphint2` returns the results as a squeezed singleton. To get the full singleton, set the `squeeze` property to `off`:

```
data = mphint2(model,<expr>,edim,'squeeze','off')
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphint2(model,<expr>,edim,'matrix','off')
```

SPECIFY THE INTEGRATION SETTINGS

To specify integration settings such as the integration method, integration order, or axisymmetry assumption using these properties:

- `method`, specify the integration method, which can be either integration or summation:

  ```
  data = mphint2(model,<expr>,edim,'method',method)
  ```

  where *method* can be `'integration'` or `'summation'`. The default uses the appropriate method for the given expression.

- `intorder`, specify the integration order:

  ```
  data = mphint2(model,<expr>,edim,'intorder',<order>)
  ```

  where *order* is a positive integer. The default value is 4.

For datasets other than Solution, Particle, Cut*, Time Integral, Time Average, Surface, and Line, the integration order does correspond to an element refinement.

- `intsurface` or `intvolume`, compute surface or volume integral for axisymmetric models:

  ```
  data = mphint2(model,<expr>,edim,'intsurface','on')
  data = mphint2(model,<expr>,edim,'intvolume','on')
  ```

Evaluating an Expression Average

Use the function `mphmean` to evaluate the average of a given expression over inner solution lists. To evaluate the mean of the expressions *e1,...* use the command `mphmean`:

```
[d1,...] = mphmean(model,{'e1',...},edim)
```

where *edim* is a string to define the element entity dimension: `'volume'`, `'surface'`, or `'line'`. *edim* can also be a positive integer (`3`, `2`, or `1` respectively). The output variables `d1,...` are an NxP array where N is the number of inner solutions and P the number of outer solutions.

The rest of this section has additional information for the function `mphmean`:

- Specify the Evaluation Data
- Output Format
- Specify the Integration Settings

SPECIFY THE EVALUATION DATA

The function `mphmean` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  ```
  data = mphmean(model,<expr>,edim,'dataset',<dsettag>)
  ```

  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model.

- `selection`, specify the domain selection for evaluation:

  ```
  data = mphmean(model,<expr>,edim,'selection',<seltag>)
  ```

  where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

```
data = mphmean(model,<expr>,edim,'solnum',<solnum>)
```

where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution or all inner solutions, respectively. By default the evaluation is performed on all inner solutions.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  data = mphmean(model,<expr>,edim,'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer array corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution, respectively. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphmean(model,<expr>,edim,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

- In case of data series, such as from a parametric or a transient study, an operation can be applied. To perform data series operation use the function `mphmean` as in the command below:

  ```
  data = mphmean(model,<expr>,edim,'dataseries',<dataoperation>)
  ```

  where *<dataoperation>* can be one of the following value: `'none'` (no operation performed), `'average'` (to compute average of the selected series), `'integral'` (to integrate over series), `'maximum'` (to evaluate the maximum over series), `'minimum'` (to evaluate the minimum), `'rms'` (to compute the root mean square), `'stddev'` (to compute the standard deviation), or `'variance'` (to compute the variance).

OUTPUT FORMAT

The function `mphmean` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable:

```
[data,unit] = mphmean(model,<expr>,edim)
```

where `unit` is a 1xN cell array and N is the number of expressions to evaluate.

By default `mphmean` returns the results as a squeezed singleton. To get the full singleton set the `squeeze` property to `off`:

```
data = mphmean(model,<expr>,edim,'squeeze','off')
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphmean(model,<expr>,edim,'matrix','off')
```

SPECIFY THE INTEGRATION SETTINGS

You can specify integration settings such as an integration method or integration order to perform the mean operation. The available integration properties are:

- `method`, specify the integration method, which can be either integration or summation:

  ```
  data = mphmean(model,<expr>,edim,'method',method)
  ```

  where *method* can be `'integration'` or `'summation'`. The default uses the appropriate method for the given expression.

- `intorder`, specify the integration order:

  ```
  data = mphmean(model,<expr>,edim,'intorder',<order>)
  ```

  where *<order>* is a positive integer. The default value is 4.

Extracting Data at Node Points

The function `mpheval` lets you evaluate expressions on node points.

Call the function `mpheval` as in this command:

```
pd = mpheval(model, <expr>)
```

where *<expr>* is a string cell array that lists the expression to evaluate. The expression has to be defined in the COMSOL model object in order to be evaluated.

`pd` is a structure with the following fields:

- `expr` contains the list of names of the expressions evaluated with `mpheval`;
- `d1` contains the value of the expression evaluated. The columns in the data value fields correspond to node point coordinates in columns in the field `p`. In case of several expressions are evaluated in `mpheval`, additional field `d2`, `d3`,... are available;
- `p` contains the node point coordinates information. The number of rows in p is the number of space dimensions;
- `t` contains the indices to columns in `pd.p` of a simplex mesh; each column in `pd.t` represents a simplex;

- `ve` contains the indices to mesh elements for each node points; and
- `unit` contains the list of the unit for each evaluated expressions.

The rest of this section has additional information for the function `mpheval`:

- Specify the Evaluation Data
- Output Format
- Specify the Evaluation Quality
- Display the Expression in Figures

SPECIFY THE EVALUATION DATA

The function `mpheval` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  ```
  pd = mpheval(model, <expr>, 'dataset', <dsettag>)
  ```

  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model. Selection datasets such as Cut Point, Cut Line, Edge, Surface, and so forth are not supported.

- `selection`, specify the domain selection for evaluation:

  ```
  pd = mpheval(model, <expr>, 'selection', <seltag>)
  ```

  where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection, the output value is NaN.

- `edim`, specify the element dimension for evaluation:

  ```
  pd = mpheval(model, <expr>, 'edim', edim)
  ```

  where *edim* is one of the strings `'point'`, `'edge'`, `'boundary'`, or `'domain'`. It is also possible to use the corresponding integer which in 3D is in the range from 0 to 3. The default settings correspond to the model geometry space dimension. When using a lower space dimension value, make sure that the evaluation point coordinates dimension has the same size.

Use the function `mphevalpoint` to evaluate expressions at geometric points (see Evaluating an Expression at Geometry Vertices).

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

  ```
  pd = mpheval(model, <expr>, 'solnum', <solnum>)
  ```

  where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution, or all inner solution respectively. By default the evaluation is performed on all inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  pd = mpheval(model, <expr>, 'outersolnum', <outersolnum>)
  ```

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution respectively. The default setting uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  pd = mpheval(model, <expr>, 't', <time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

- `phase`, specify the phase in degrees:

  ```
  pd = mpheval(model, <expr>, 'phase', <phase>)
  ```

  where *<phase>* is a double value.

- `pattern`, use Gauss point evaluation:

  ```
  pd = mpheval(model, <expr>, 'pattern','gauss')
  ```

  The default evaluation is performed on the Lagrange points.

OUTPUT FORMAT

The function `mpheval` returns a structure in the MATLAB workspace. You can specify other output data formats.

To only obtain the data evaluation as a double array, set the property `dataonly` to `on`. This is speeds up the call to COMSOL since the coordinate and element information is not retrieved.

```
pd = mpheval(model, <expr>, 'dataonly', 'on')
```

Returns only the real part in the data evaluation with the property `complexout`:

```
pd = mpheval(model, <expr>, 'complexout', 'off')
```

SPECIFY THE EVALUATION QUALITY

Define `mpheval` function settings to specify the evaluation quality using these properties:

- `refine`, specify the element refinement for evaluation:

  ```
  pd = mpheval(model, <expr>, 'refine', <refine>)
  ```

  where *<refine>* is a positive integer. The default value is `1` which set the simplex mesh identical to the geometric mesh. Many model use second order elements for which a `refine` value of 2 must be used to use all the data in the model.

- `smooth`, specify the smoothing method to enforce continuity on discontinuous data evaluation:

  ```
  pd = mpheval(model, <expr>, 'smooth', smooth)
  ```

  where *smooth* is either `'none'`, `'everywhere'`, or `'internal'` (default). Set the property to `none` to evaluate the data on elements independently, set to `everywhere` to apply the smoothing to the entire geometry, and set to `internal` to smooth the quantity inside the geometry (but no smoothing takes place across borders between domains with different settings). The output with the same data and same coordinates are automatically merged, which means that the output size can differ depending on the smoothing method.

- `recover`, specify the accurate derivative recovery:

  ```
  pd = mpheval(model, <expr>, 'recover', recover)
  ```

  where *recover* is either `'ppr'`, `'pprint'`, or `'off'` (default). Set the property to `ppr` to perform recovery inside domains or set to `pprint` to perform recovery inside domains. Because the accurate derivative processing takes time, the property is disabled by default.

OTHER EVALUATION PROPERTIES

To not use complex-value functions with real inputs, use the property `complexfun`:

```
pd = mpheval(model, <expr>, 'complexfun', 'off')
```

The default value uses complex-valued functions with real inputs.

Use the property `matherr` to return an error for undefined operations or expressions:

```
pd = mpheval(model, <expr>, 'matherr', 'on')
```

DISPLAY THE EXPRESSION IN FIGURES

You can display an expression evaluated with `mpheval` in an external figure with the function `mphplot` (see Displaying The Results).

Evaluating an Expression at Geometry Vertices

The function `mphevalpoint` returns the result of a given expression evaluated at the geometry points:

```
[d1,...] = mphevalpoint(model,{'e1',...})
```

where *e1*, … are the COMSOL expressions to evaluate. The output `d1, …` is an *N*-by-*P* double array, where *N* is the number of evaluation points and *P* the length of the inner solution.

The rest of this section has additional information for the function `mphevalpoint`:

- Specify the Evaluation Data
- Output Format

SPECIFY THE EVALUATION DATA

The function `mphevalpoint` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  `data = mphevalpoint(model,<expr>,'dataset',<dsettag>)`
  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model.

- `selection`, specify the domain selection for evaluation:

  `data = mphevalpoint(model,<expr>,'selection',<seltag>)`

  where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection, the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

```
data = mphevalpoint(model,<expr>,'solnum',<solnum>)
```

where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution, or all inner solution respectively. By default the evaluation is performed on all inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  data = mphevalpoint(model,<expr>,'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution respectively. The default settings use the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphevalpoint(model,<expr>,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

Perform a data series operation with the `dataseries` property:

```
data = mphevalpoint(model,<expr>,'dataseries',dataseries)
```

where *dataseries* is either `'mean'`, `'int'`, `'max'`, `'min'`, `'rms'`, `'std'`, or `'var'`. Depending on the property value, `mphevalpoint` performs the following operations — mean, integral, maximum, minimum, root mean square, standard deviation, or variance, respectively.

When performing a minimum or maximum operation on the data series, you can specify to perform the operation using the real or the absolute value. Set the property `minmaxobj` to `'real'` or `'abs'`, respectively:

```
data = mphevalpoint(model,<expr>,'dataseries',dataseries,...
'minmaxobj', valuetype)
```

By default *valuetype* is `'real'`.

OUTPUT FORMAT

The function `mphevalpoint` supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable:

```
[data,unit] = mphevalpoint(model,<expr>)
```

with `unit` is a 1xN cell array where N is the number of expressions to evaluate.

By default, `mphevalpoint` returns the results as a squeezed singleton. To get the full singleton set the `squeeze` property to `off`:

```
data = mphevalpoint(model,<expr>,'squeeze','off')
```

Set the property `matrix` to `off` to return the data as a cell array instead of a double array:

```
data = mphevalpoint(model,<expr>,'matrix','off')
```

Evaluating Expressions on Particle/Ray Trajectories

Evaluate expressions on particle trajectories with the function `mphparticle` and on and ray trajectories with the function `mphray`.

`mphray` supports only Ray Trajectories datasets. `mphparticle` supports both Ray Trajectories and Particle Trajectories datasets.

In this section you can replace the command `mphparticle` and `mphray` as they support the same properties.

Evaluate expressions on particle and ray trajectories with either the function `mphparticle` or `mphray`.

To evaluate the particle position and the particle velocity run `mphparticle` as in this command:

```
pd = mphparticle(model)
```

`pd` is a structure containing the information about particle position and particle velocity at every time step. The information is stored in the following fields:

- `p` contains the coordinates of the particle position along the trajectories. The data are stored in a NxMxL array where N is the number of time steps, M the number of evaluation point along the particle trajectories, and L the evaluation space dimension.
- `v` contains the value of the particle velocity along the trajectories. The data are stored in a NxMxL array where N is the number of time steps, M the number of evaluation points along the particle trajectories, and L the evaluation space dimension.
- `t` contains the list of evaluation time.

You can also specify expressions to evaluate along the particle trajectories. Run the function `mphparticle` as in this command:

```
pd = mphparticle(model,'expr','e1')
```

where `'e1'` is the expression to evaluate along the particle trajectories. The output structure `pd` contains the fields `p`, `v`, and `t` (described above) with the following ones:

- `unit` contains the unit of the evaluated expression;
- `d1` contains the value of the expression. The data are stored in a NxM array where N is the number of time steps and M the number of evaluation points along the particle trajectories; and
- `expr` contains the list of the evaluated expression.

Use a string cell array to evaluate several expressions at once. The result of the evaluation is then stored in the field `d1,...` corresponding to each evaluated expression.

SPECIFY THE EVALUATION DATA

The function `mphparticle` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  ```
  pd = mphparticle(model,'expr',<expr>,'dataset',<dsettag>)
  ```

  <dsettag> is the tag of a particle solution dataset. The default value is the current particle solution dataset of the model.
- To evaluate the expression data at a specific time use the property `t`:

  ```
  pd = mphparticle(model,'expr',<expr>,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

OUTPUT FORMAT

The function `mphparticle` also supports other output formats.

Set the property `dataonly` to `on` to return only the data related to the specified expression:

```
pd = mphparticle(model,'expr',<expr>,'dataonly','on')
```

The output structure `pd` only contains the field `unit`, `d#`, `expr`, and `t` (described above).

Evaluating a Global Expression

Evaluate a global expression with the function `mphglobal`.

To evaluate a global expression at the MATLAB® prompt, call the function `mphglobal` as in this command:

```
[d1,...] = mphglobal(model,{'e1',...})
```

where *e1,...* are the COMSOL Multiphysics global expressions to evaluate. The output values `d1,...` are returned as a Px1 double array, with P the length of inner parameters.

The rest of this section has additional information for the function `mphglobal`:

- Specify the Evaluation Data
- Output Format
- Other Evaluation Properties

SPECIFY THE EVALUATION DATA

The function `mphglobal` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  ```
  data = mphglobal(model,<expr>,'dataset',<dsettag>)
  ```

  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

  ```
  data = mphglobal(model,<expr>,'solnum',<solnum>)
  ```

  where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution or all inner solutions, respectively. By default the evaluation is performed on all inner solutions.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  data = mphglobal(model,<expr>,'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'` to evaluate the

expression for all or the last outer solution, respectively. The default settings uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  data = mphglobal(model,<expr>,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

- `phase`, specify the phase in degrees:

  ```
  data = mphglobal(model,<expr>,'phase',<phase>)
  ```

  where *<phase>* is a double value.

OUTPUT FORMAT

The function `mphglobal` also supports other output formats.

To extract the unit of the evaluated expression, define an extra output variable:

```
[data,unit] = mphglobal(model,<expr>)
```

with `unit` is a 1xN cell array where N is the number of expressions to evaluate.

Returns only the real part in the data evaluation with the property `complexout`:

```
data = mphglobal(model,<expr>,'complexout','off')
```

OTHER EVALUATION PROPERTIES

Set the `unit` property to specify the unit of the evaluation:

```
data = mphglobal(model,<expr>,'unit',<unit>)
```

where *<unit>* is a cell array with the same length as *<expr>*.

Use the property `matherr` to return an error for undefined operations or expressions:

```
data = mphglobal(model,<expr>,'matherr','on')
```

Evaluating a Matrix Expression at Points

The function `mphevalpointmatrix` returns the result of a given matrix expression evaluated at the points:

```
M = mphevalpointmatrix(model,<expr>,...)
```

where *<expr>* is the COMSOL expressions to evaluate. The output `M` is a matrix.

The rest of this section has additional information for the function `mphevalpointmatrix`:

- Specify the Evaluation Data
- Output Format

SPECIFY THE EVALUATION DATA

The function `mphevalpoint` supports the following properties to set the data of the evaluation to perform:

- `dataset`, specify the solution dataset to use in the evaluation:

  `M = mphevalpointmatrix(model,<expr>,'dataset',<dsettag>)`

  <dsettag> is the tag of a solution dataset. The default value is the current solution dataset of the model.

- `selection`, specify the domain selection for evaluation:

  `M = mphevalpointmatrix(model,<expr>,'selection',<seltag>)`

  where *<seltag>* is the tag of a selection node to use for the data evaluation. *<seltag>* can also be a positive integer array that corresponds to the domain index list. The default selection is all domains where the expression is defined. If the evaluation point does not belong to the specified domain selection, the output value is NaN.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

  `M = mphevalpointmatrix(model,<expr>,'solnum',<solnum>)`

  where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution, or all inner solution respectively. By default the evaluation is performed on all inner solution.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  `M = mphevalpoint(model,<expr>,'outersolnum',<outersolnum>)`

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'`, to evaluate the expression for all or the last outer solution respectively. The default settings use the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

```
M = mphevalpointmatrix(model,<expr>,'t', <time>)
```

where *<time>* is a double array. The default value corresponds to all the stored time steps.

- Perform a data series operation with the `dataseries` property:

```
M = mphevalpointmatrix(model,<expr>,'dataseries', dataseries)
```

where *dataseries* is either `'none'`, `'average'` or `'sum'`.

Evaluating a Global Matrix

`mphevalglobalmatrix` evaluates the matrix variable such as S-parameters in a model with several ports activated as a parametric sweep and a frequency-domain study.

Note: S-parameters evaluation requires the AC/DC Module or the RF Module.

To evaluate the global matrix associated to the expression *<expr>*, enter the command:

```
M = mphevalglobalmatrix(model,<expr>)
```

The output data `M` is an *N*x*N* double array, where *N* is the number of port boundary condition set in the model.

The rest of this section has additional information for the function `mphevalglobalmatrix`:

- Specify the Evaluation Data
- Specify Matrix transformation

SPECIFY THE EVALUATION DATA

The function `mphevalglobalmatrix` supports the following properties to set the data of the evaluation to perform:

- Set the solution dataset for evaluation with the property `dataset`:

```
M = mphevalglobalmatrix(model,<expr>,'dataset',<dsettag>)
```

where *<dsettag>* is the tag of a solution data.

- `solnum`, specify the inner solution number for data evaluation. Inner solutions are generated for the following analysis types: time domain, frequency domain, eigenvalue, or stationary with continuation parameters:

```
M = mphevalglobalmatrix(model,<expr>,'solnum',<solnum>)
```

where *<solnum>* is an integer array corresponding to the inner solution index. *<solnum>* can also be a string: `'end'` or `'all'` to evaluate the expression for the last inner solution or all inner solutions, respectively. By default the evaluation is performed on all inner solutions.

- `outersolnum`, specify the outer solution number for data evaluation. Outer solutions are generated with parametric sweeps:

  ```
  M = mphevalglobalmatrix(model,<expr>,'outersolnum',<outersolnum>)
  ```

  where *<outersolnum>* is a positive integer corresponding to the outer solution index. *<outersolnum>* can also be a string, `'all'` or `'end'` to evaluate the expression for all or the last outer solution, respectively. The default settings uses the first outer solution for the data evaluation.

- To evaluate the expression data at a specific time use the property `t`:

  ```
  dM = mphevalglobalmatrix(model,<expr>,'t',<time>)
  ```

  where *<time>* is a double array. The default value corresponds to all the stored time steps.

- Perform a data series operation with the `dataseries` property:

  ```
  M = mphevalglobalmatrix(model,<expr>,'dataseries', dataseries)
  ```

  where *dataseries* is either `'none'`, `'average'`, or `'sum'`.

- Perform a data series operation on outer data with the `outerdataseries` property:

  ```
  M = mphevalglobalmatrix(model,<expr>,'outerdataseries',
  outerdataseries)
  ```

  where *outerdataseries* is either `'none'`, `'average'`, or `'sum'`.

SPECIFY MATRIX TRANSFORMATION

To apply a transformation operation to compute the inverse of the matrix variable or to convert between the impedance matrix, Z, the admittance matrix, Y, and the S-parameter matrix S, use the trans property:

```
M = mphevalglobalmatrix(model,<expr>,'trans',<trans>)
```

where *<trans>* can be either: `'maxwellmutual'` (from Maxwell to mutual capacitance), `'mutualmaxwell'` (from mutual to Maxwell capacitance), `'none'` (no transformation), `'inverse'` (compute the inverse of the matrix), `'sy'` (from S to Y transformation), `'sz'` (from S to Z transformation), `'ys'` (from Y to S transformation), `'yz'` (from Y to Z transformation), `'zs'` (from Z to S transformation), or `'zy'` (from Z to Y transformation).

For S to Y and Y to S transformation you need to specify the characteristic admittance, to proceed use the command:

```
M = mphevalglobalmatrix(model,<expr>,'trans','sy','y0',<value>)
M = mphevalglobalmatrix(model,<expr>,'trans','ys','y0',<value>)
```

where *<value>* is the characteristic admittance in siemens (S). The default value is 1 S.

For S to Z and Z to S transformation you need to specify the characteristic impedance, to proceed use the command:

```
M = mphevalglobalmatrix(model,<expr>,'trans','sz','z0',<value>)
M = mphevalglobalmatrix(model,<expr>,'trans','zs','z0',<value>)
```

where *<value>* is the characteristic admittance in ohm (Ω).The default value is 1 Ω.

Global Matrix Evaluation section in the *COMSOL Reference Manual*.

Extracting Data From Tables

In the Table node you can store the data evaluated with the COMSOL Multiphysics built-in evaluation method (see The Numerical Node Syntax).

Use `mphtable` to extract the data stored in the table with the tag *<tbltag>*. Enter:

```
tabl = mphtable(model,<tbltag>)
```

This creates a structure `tabl` made with the following fields:

- `headers` for the table,
- `tag` of the table,
- `data` of the extracted table, and

`filename` when the table is exported to file.

Running Models in a Loop

A common use of LiveLink™ *for* MATLAB® is to run models in a loop. MATLAB provides several functionalities to run loops, including conditional statements and error handling, and this section shows how to use that functionality together with the COMSOL API syntax to run COMSOL Multiphysics® models in loops.

In this section:

The Parametric Sweep Node

Using the COMSOL API you can run models in loops. See Adding a Parametric Sweep in the section ***Building Model***s.

By using the COMSOL built-in function to run models in loops, you can ensure the model is saved automatically at each iteration. COMSOL also offers tools to take advantage of clusters and distributed computer architectures.

Running Model in a Loop Using the MATLAB® Tools

Use MATLAB® tools such as `for` or `while` statements to run your model in a loop. The COMSOL API commands can be included in scripts using MATLAB commands. To evaluate such a script you need to have MATLAB connected to a COMSOL *server*.

To run a model in a loop you do not need to run the entire M-file's commands from scratch. It is recommended to load a COMSOL model in MATLAB and run the loop only over the desired operations. The COMSOL model is automatically updated when running the study node.

You can run an M-file for a model from scratch if required, for example, to generate the geometry in loop.

The model run inside a MATLAB loop is not automatically saved. Make sure to save the model at each iteration using the command `mphsave` to save the model object.

If you are not interested in saving the entire model object at each iteration, you can extract data and store it in the MATLAB workspace. See Extracting Results to find the most suitable function to your model.

When running loops in MATLAB, the iteration progress is taken care of by MATLAB; only the COMSOL commands are run in the COMSOL *server*.

You can generate as many nested loops as needed and combine the loop with other MATLAB conditional statements such as `if` and `switch` or error handling statements such as `try/catch`. Or break the loop with `break`, or jump to the next loop iteration with `continue`.

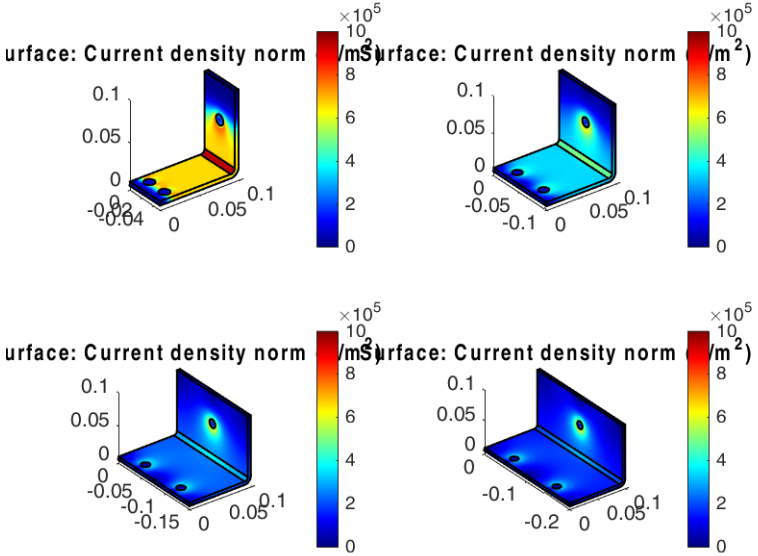
See the MATLAB help for more information about the MATLAB commands `for`, `while`, `if`, `switch`, `try/catch`, `break`, and `continue`.

GEOMETRY PARAMETERIZATION

This example shows how to proceed to geometry parameterization using a MATLAB `for` loop. The model consists of the busbar example available in the COMSOL Multiphysics Applications Libraries; see the *Introduction to COMSOL Multiphysics*.

In this example the loop iterates over the busbar's width, `wbb`. The solution for each parameter value is displayed using the second plot group defined in the COMSOL model. All the results are plotted in the same figure.

The results from the computation display in these plots:



Code for use with MATLAB®

```
model = mphopen('busbar');
w = [5e-2 10e-2 15e-2 20e-2];
for i = 1:4
    model.param.set('wbb',w(i));
    model.study('std1').run;
    subplot(2,2,i);
    mphplot(model,'pg5','rangenum',1);
end
```

Running Models in Batch Mode

Use LiveLink™ *for* MATLAB® to model in batch mode. At the MATLAB prompt you can run commands to set up the batch job using the COMSOL Multiphysics® built-in method or run custom scripts directly from a command line. In this section:

- The Batch Node
- Running an M-File in Batch Mode
- Running an M-File in Batch Mode Without Display

The Batch Node

Using the COMSOL API you can run models in a loop. See Adding a Job Sequence.

Running an M-File in Batch Mode

Running COMSOL with MATLAB® in batch mode requires that you have xterm installed on your machine. If this is not the case see Running an M-File in Batch Mode Without Display.

To run in batch an M-script that runs COMSOL Model is required. Start COMSOL with MATLAB at a terminal window with this command:

```
comsol mphserver matlab myscript
```

where `myscript` is the M-script, saved as `myscript.m`, that contains the operation to run at the MATLAB prompt.

COMSOL Multiphysics does not automatically save the model. You need to make sure that the model is saved before the end of the execution of the script. See Loading and Saving a Model.

You can also run the script in batch without the MATLAB desktop and the MATLAB splash screen. Enter this command:

```
comsol mphserver matlab myscript -nodesktop -mlnosplash
```

Running an M-File in Batch Mode Without Display

To connect COMSOL with a MATLAB® terminal requires that xterm is installed on the machine. If this is not the case as it might be for a computation COMSOL *server*, a workaround is to connect manually MATLAB to a COMSOL *server* with the function `mphstart`.

These steps describe how to run an M-script that runs a COMSOL model:

1. In a system terminal prompt start a COMSOL Multiphysics Server with the command:

   ```
   comsol mphserver -silent &
   ```

2. In the same terminal window change the path to the COMSOL installation directory:

   ```
   cd COMSOL_path/mli
   ```

3. From that location, start MATLAB without display and run the `mphstart` function in order to connect MATLAB to COMSOL:

   ```
   matlab -nodesktop -nosplash -r "mphstart; myscript"
   ```

For more information about how to connect MATLAB to a COMSOL *server* see Starting COMSOL® with MATLAB® on Windows ®/ Mac OSX / Linux®.

Working with Matrices

In this section:

Extracting System Matrices

Extract the matrices of the COMSOL Multiphysics linearized system with the function `mphmatrix`. To call the function `mphmatrix`, specify a solver node and the list of the system matrices to extract:

```
str = mphmatrix(model, <soltag>, 'out', out)
```

where *<soltag>* is the solver node tag used to assemble the system matrices and *out* is a cell array containing the list of the matrices to evaluate. The output data `str` returned by `mphmatrix` is a MATLAB® structure, and the fields correspond to the assembled system matrices.

See the Advanced section and the Assemble section in the *COMSOL Multiphysics Reference Manual*, for more information about matrix evaluation.

The system matrices that can be extracted with `mphmatrix` are listed in the table:

EXPRESSION	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian
D	Damping matrix
E	Mass matrix
NF	Constraint force Jacobian

EXPRESSION	DESCRIPTION
NP	Optimization constraint Jacobian (*)
MP	Optimization constraint vector (*)
MLB	Lower bound constraint vector (*)
MUB	Upper bound constraint vector (*)
Kc	Eliminated stiffness matrix
Lc	Eliminated load vector
Dc	Eliminated damping matrix
Ec	Eliminated mass matrix
Null	Constraint null-space basis
Nullf	Constraint force null-space matrix
ud	Particular solution ud
uscale	Scale vector
(*) Requires the Optimization Module.	

SELECTING LINEARIZATION POINTS

The default selection of linearization points for the system matrix assembly is the current solution of the solver node associated to the assembly.

If the linearization point is not specified when calling `mphmatrix`, the COMSOL Multiphysics software automatically runs the entire solver configuration before assembling and extracting the matrices.

Save time during the evaluation by manually setting the linearization point. Use the `initmethod` property as in this command:

```
str = mphmatrix(model,<soltag>,'out',out,'initmethod',method)
```

where *`method`* corresponds to the type of linearization point — the initial value expression (`'init'`) or a solution (`'sol'`).

To set the solution to use for the linearization point, use the property `initsol`:

```
str = mphmatrix(model,<soltag>,'out',out,'initsol',<initsoltag>)
```

where *`<initsoltag>`* is the solver tag to use for linearization points. You can also set the `initsol` property to `'zero'`, which corresponds to using a null solution vector as a linearization point. The default is the current solver node where the assemble node is associated.

For continuation, time-dependent, or eigenvalue analyses you can set the solution number to use as a linearization point. Use the `solnum` property:

```
str = mphmatrix(model,<soltag>,'out',out,'solnum',<solnum>)
```

where *<solnum>* is an integer value corresponding to the solution number. The default value is the last solution number available with the current solver configuration.

See Retrieving Xmesh Information to learn how to get relation between the degrees of freedom information in the matrix system and coordinates or element information.

SPECIFYING WHEN TO ASSEMBLE THE MATRICES IN THE SOLUTION SEQUENCE

You can specify when in the solution sequence to assemble the system matrices, for instance after computing the solution or if you have a solution sequence combining different solver. By default the system matrices are assembled before running the first solver, just after the first dependent variable node in the solution sequence. To specify the node that precede the matrix extraction use the `extractafter` property:

```
str = mphmatrix(model,<soltag>,'out',out,'extractafter',<nodetag>)
```

where *<nodetag>* is the tag of a solution sequence node such as dependent variable or solver nodes.

EIGENVALUE PROBLEMS

For eigenvalue problems, it is necessary to specify the eigenvalue name and the eigenvalue linearization point. Used the property `eigname` to specify the name of the eigenvalue and `eigref` to specify the value of eigenvalue linearization point:

```
str = mphmatrix(model,<soltag>,'out',out,'eigname',<eigname>)
```

```
str = mphmatrix(model,<soltag>,'out',out,'eigname',<eigname>,...
'eigref', <eigref>)
```

where *<eigname>* is a string and *<eigref>* a double.

ROW EQUILIBRATION, MATRIX SYMMETRY AND NULL-SPACE FUNCTION

The default assembly of the system matrices assumes row equilibration of the system matrices. It is however possible to extract the unscaled matrices, to proceed set the `rowscale` property to `off`:

```
str = mphmatrix(model,<soltag>,'out',out,'rowscale','off')
```

Set `symmetry` property to specify manually the symmetry type for the matrix evaluation. The `symmetry` property support the following values:

```
str = mphmatrix(model,<soltag>,'out',out,'symmetry',sym)
```

where *sym* can be either of one of the following value:

- `'on'`, to assemble and extract the system matrices as symmetric.
- `'off'`, to assemble and extract the system matrices as non-symmetric.
- `'hermitian'`, to assemble and extract the system matrices as hermitian.
- `'auto'`, to let the solver assembly determine the type of the system matrices.

Use the `nullfun` property to specify the method for computation of matrices needed for constraint handling:

- `'flnullorth'`, a method based on singular value decomposition;
- `'flspnull'`, to handle constraint matrices with nonlocal couplings using singular sparse algorithm;
- `'explicitsp'`, to handle constraints by explicitly eliminating the DOFs on the destination side of the explicit constraints. The remaining constraints are handled using the Sparse method.
- `'explicitorth'`, to handle constraints by explicitly eliminating the DOFs on the destination side of the explicit constraints. The remaining constraints are handled using the Orthonormal method.
- `'auto'`, to let the software automatically determine the most appropriate method, which uses an explicit handling of nodal constraints and one of the Orthonormal or Sparse methods for the remaining constraints.

COMPLEX FUNCTION

If the system contains complex function, use the property `complexfun` to specify how to handle such a function. Set this property to on to use complex-valued function with real input:

```
str = mphmatrix(model,<soltag>,'out',out,'complexfun','on');
```

HANDLING UNDEFINED OPERATIONS

It is possible to disable the error for undefined operations during the assembly and matrix evaluation, to proceed set the property `matherr` to off as in the command below:

```
str = mphmatrix(model,<soltag>,'out',out,'matherr','off')
```

EXTRACTING THE SYSTEM MATRICES

The following illustrates how to use the `mphmatrix` command to extract eliminated system matrices of a stationary analysis and linear matrix system at the MATLAB prompt.

The model consists of a linear heat transfer problem solved on a unit square with a `1e5 W/m^2` surface heat source and temperature constraint. Only one quarter of the geometry is represented in the model. For simplification reasons, the mesh is made of four quad elements and the discretization is set with linear element.

These commands set the COMSOL model object:

```
model = ModelUtil.create('Model2');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('sq1', 'Square');
geom1.run;

mat1 = comp1.material.create('mat1');
def = mat1.materialModel('def');
def.set('thermalconductivity',4e2);

ht = comp1.physics.create('ht', 'HeatTransfer', 'geom1');
ht.prop('ShapeProperty').set('boundaryFlux_temperature',false);
ht.prop('ShapeProperty').set('order_temperature',1);
hs1 = ht.feature.create('hs1','HeatSource',2);
hs1.selection.set(1);
hs1.set('Q',1,1e5);

temp1 = ht.feature.create('temp1','TemperatureBoundary',1);
temp1.selection.set([1 2]);

mesh1 = comp1.mesh.create('mesh1');
dis1 = mesh1.feature.create('dis1','Distribution');
dis1.selection.set([1 2]);
dis1.set('numelem',2);
mesh1.feature.create('map1','Map');

std1 = model.study.create('std1');
std1.feature.create('stat','Stationary');
std1.run;
```

To extract the solution vector of the computed solution, run the function `mphgetu` as in this command:

```
U = mphgetu(model);
```

To assemble and extract the eliminated stiffness matrix and the eliminated load vector, set the linearization point to the initial value expression by entering:

```
MA = mphmatrix(model ,'sol1', ...
    'Out', {'Kc','Lc','Null','ud','uscale'},...
    'initmethod','sol','initsol','zero');
```

Solve for the eliminated solution vector using the extracted eliminated system:

```
Uc = MA.Null*(MA.Kc\MA.Lc);
```

Combine the eliminated solution vector and the particular vector:

```
U0 = Uc+MA.ud;
```

Scale back the solution vector:

```
U1 = U0.*MA.uscale;
```

Now compare both solution vector `U` and `U1` computed by COMSOL Multiphysics and by the matrix operation, respectively.

Code for use with MATLAB®

```
model = ModelUtil.create('Model');
comp1 = model.component.create('comp1', true);
geom1 = comp1.geom.create('geom1', 2);
geom1.feature.create('sq1', 'Square');
geom1.run;
mat1 = comp1.material.create('mat1');
def = mat1.materialModel('def');
def.set('thermalconductivity',4e2);
ht = comp1.physics.create('ht', 'HeatTransfer', 'geom1');
ht.prop('ShapeProperty').set('boundaryFlux_temperature',false);
ht.prop('ShapeProperty').set('order_temperature',1);
hs1 = ht.feature.create('hs1','HeatSource',2);
hs1.selection.set(1);
hs1.set('Q',1,1e5);
temp1 = ht.feature.create('temp1','TemperatureBoundary',1);
temp1.selection.set([1 2]);
mesh1 = comp1.mesh.create('mesh1');
dis1 = mesh1.feature.create('dis1','Distribution');
dis1.selection.set([1 2]);
dis1.set('numelem',2);
mesh1.feature.create('map1','Map');
std1 = model.study.create('std1');
std1.feature.create('stat','Stationary');
std1.run;
U = mphgetu(model);
MA = mphmatrix(model ,'sol1', ...
    'Out', {'Kc','Lc','Null','ud','uscale'},...
    'initmethod','sol','initsol','zero');
```

```
Uc = MA.Null*(MA.Kc\MA.Lc);
U0 = Uc+MA.ud;
U1 = U0.*MA.uscale;
```

Set System Matrices in the Model

Use the function `mphinputmatrix` to set a linear matrix system to a model:

```
mphinputmatrix(model,<str>,<soltag>,<soltypetag>)
```

This command set the matrices of a linear system stored in the MATLAB® structure *<str>* into the model. The linear system is associated to the solver sequence *<soltag>* and is to be solved by the solver *<soltypetag>*.

`mphinputmatrix` only supports the solver types Stationary, Eigenvalue, and Time.

A valid structure *<str>* for a stationary solver includes the following fields:

FIELD	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian

A valid structure *<str>* for a time-dependent or an eigenvalue solver includes the following fields:

EXPRESSION	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian
D	Damping matrix
E	Mass matrix

You can also include the Constraint force Jacobian vector, defined in the field `NF`.

Once the linear system is loaded in the model, you can directly run the solver.

The system matrices are not stored in the model when it is saved in the MPH-format or loaded to the COMSOL Desktop.

SETTING A MODEL WITH A MODIFIED MATRIX SYSTEM

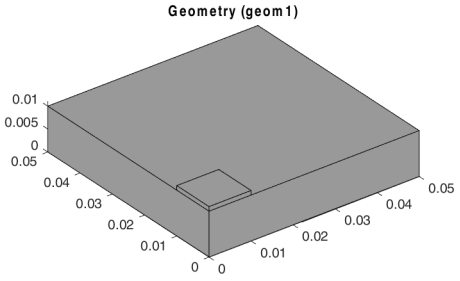
This example deals with heat transfer in solids physics. The geometry and physics settings are already set in the model and saved in the MPH-format. The Model MPH-file comes with the COMSOL installation.

At the MATLAB prompt you load the model and add an additional line heat source to the model directly in the system matrix by manually changing the load vector. Then compute the solution of the modified system in COMSOL.

Load the base Model MPH-file and display the geometry:

```
model = mphopen('model_tutorial_llmatlab.mph');
mphgeom(model)
```

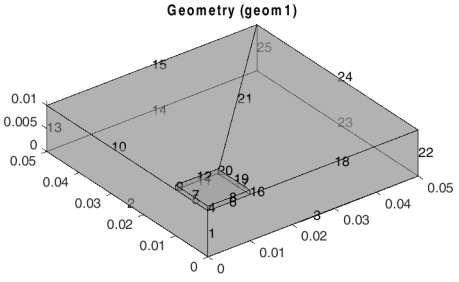
This results in the following MATLAB figure:



Draw the line to be used as a line heat source in the model and plot the modified geometry:

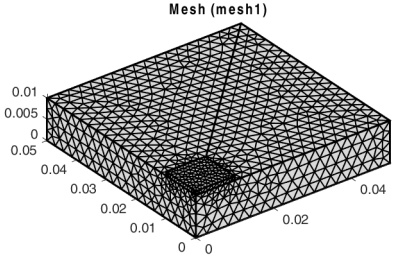
```
comp1 = model.component('comp1');
b1 = comp1.geom('geom1').feature.create('b1', 'BezierPolygon');
b1.set('p', {'1e-2' '5e-2'; '1e-2' '5e-2'; '1e-2' '1e-2'});
mphgeom(model,'geom1','edgelabels','on','facealpha',0.5);
```

In the figure below you can see that the added line as the index 21:



Generate a mesh with finer mesh settings:

```
mesh1 = comp1.mesh('mesh1');
mesh1.feature.create('ftet1', 'FreeTet');
mesh1.feature('size').set('hauto', 3);
mesh1.run;
mphmesh(model)
```



Set the solver sequence associated to a stationary study node:

```
std1 = model.study.create('std1');
std1.feature.create('stat', 'Stationary');
sol1 = model.sol.create('sol1');
sol1.study('std1');
st1 = sol1.feature.create('st1', 'StudyStep');
st1.set('studystep', 'stat');
```

```
v1 = sol1.feature.create('v1', 'Variables');
v1.set('control', 'stat');
sol1.feature.create('s1', 'Stationary');
```

Set the dependent variable discretization with linear shape function:

```
Shape = comp1.physics('ht').prop('ShapeProperty');
Shape.set('order_temperature', 1, 1);
```

The heat transfer interface automatically compute for internal DOFs in order to evaluate fluxes accurately at the boundaries. Deactivate the internal DOFs with this command:

```
Shape.set('boundaryFlux_temperature', false);
```

Now extract the matrices of the linear system associated to the solver sequence `sol1`:

```
ME = mphmatrix(model,'sol1','Out',{'K' 'L' 'M' 'N'},...
    'initmethod','sol','initsol','zero');
```

To retrieve the degrees of freedom that belong to edge 21, you need to get the geometric mesh data:

```
[stats,data] = mphmeshstats(model);
```

With the mesh data structure `data`, you can get the element indices that belong to edge 2. Use the MATLAB `find` function to list all the indices:

```
elem_idx = find(data.elementity{1}==21)'
```

With the function `mphxmeshinfo`, retrieve the finite element mesh information associated to solver sequence `sol1`:

```
info = mphxmeshinfo(model,'soltag','sol1','studysteptag','v1');
```

In the `info` structure you can get the DOFs indices that belong to the edge element defined with the indices `elem_idx`:

```
dofs = info.elements.edg.dofs;
edgdofs_idx = [];
for i = 1:length(elem_idx)
   edgdofs_idx = [edgdofs_idx; dofs(:,elem_idx(i))];
end
```

`edgdofs_idx` might contain duplicate DOFs indices. This is because the information is from the element level; the duplicate indices correspond to the connecting node between two adjacent elements.

First remove the duplicate entities:

```
unique_idx  = unique(edgdofs_idx);
```

Edit the load vector for the DOF that belong to edge 21, the total applied power is 50 W:

```
ME.L(unique_idx+1) = 50/length(unique_idx);
```

Now that the linear system has been modified, set it back in the model:

```
mphinputmatrix(model,ME,'sol1','s1')
```

Note: mphmatrix only assembles the matrix system for the dofs solved in the specified solver configuration. mphinputmatrix insert the matrix system as defined by the user. When inserting matrices in an existing model, the solution format may not be compatible with the inserted system matrices.

In order to have a compatible xmesh solution format compatible with the size of the inserted matrices, add a new equation form physics interface, solving only for one variable.

```
gForm = comp1.physics.create('g', 'GeneralFormPDE', {'u'});
gForm.prop('ShapeProperty').set('order', 1);
gForm.prop('ShapeProperty').set('boundaryFlux', false);
```

Disable the Heat Transfer physics interface.

```
comp1.physics('ht').active(false);
```

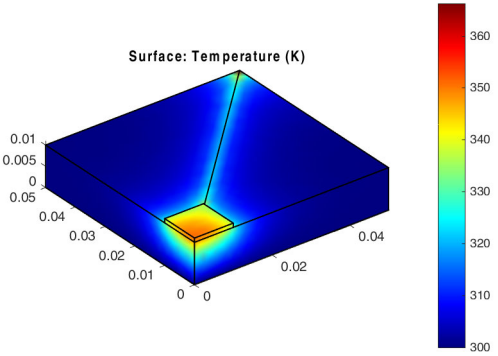
Compute the solution of the added system:

```
model.sol('sol1').runAll;
```

Display the solution:

```
pg1 = model.result.create('pg1', 'PlotGroup3D');
pg1.feature.create('surf1', 'Surface');
```

```
mphplot(model,'pg1','rangenum',1)
```



Code for use with MATLAB®

```
model = mphopen('model_tutorial_llmatlab.mph');
mphgeom(model)
comp1 = model.component('comp1');
b1 = comp1.geom('geom1').feature.create('b1', 'BezierPolygon');
b1.set('p', {'1e-2' '5e-2'; '1e-2' '5e-2'; '1e-2' '1e-2'});
mphgeom(model,'geom1','edgelabels','on','facealpha',0.5);
mesh1 = comp1.mesh('mesh1');
mesh1.feature.create('ftet1', 'FreeTet');
mesh1.feature('size').set('hauto', 3);
mesh1.run;
mphmesh(model)
std1 = model.study.create('std1');
std1.feature.create('stat', 'Stationary');
sol1 = model.sol.create('sol1');
sol1.study('std1');
st1 = sol1.feature.create('st1', 'StudyStep');
st1.set('studystep', 'stat');
v1 = sol1.feature.create('v1', 'Variables');
v1.set('control', 'stat');
sol1.feature.create('s1', 'Stationary');
Shape = comp1.physics('ht').prop('ShapeProperty');
Shape.set('order_temperature', 1, 1);
Shape.set('boundaryFlux_temperature', false);
ME = mphmatrix(model,'sol1','Out',{'K' 'L' 'M' 'N'},...
    'initmethod','sol','initsol','zero');
[stats,data] = mphmeshstats(model);
elem_idx = find(data.elementity{1}==21)'
info = mphxmeshinfo(model,'soltag','sol1','studysteptag','v1');
dofs = info.elements.edg.dofs;
edgdofs_idx = [];
for i = 1:length(elem_idx)
```

```
    edgdofs_idx = [edgdofs_idx; dofs(:,elem_idx(i))];
end
unique_idx  = unique(edgdofs_idx);
ME.L(unique_idx+1) = 50/length(unique_idx);
mphinputmatrix(model,ME,'sol1','s1')
gForm = comp1.physics.create('g', 'GeneralFormPDE', {'u'});
gForm.prop('ShapeProperty').set('order', 1);
gForm.prop('ShapeProperty').set('boundaryFlux', false);
comp1.physics('ht').active(false);
model.sol('sol1').runAll;
pg1 = model.result.create('pg1', 'PlotGroup3D');
pg1.feature.create('surf1', 'Surface');
mphplot(model,'pg1','rangenum',1)
```

Extracting State-Space Matrices

Use state-space export to create a linearized state-space model corresponding to a COMSOL Multiphysics model. You can export the matrices of the state-space form directly to the MATLAB® workspace with the command `mphstate`.

This section includes information about The State-Space System, how to Extract State-Space Matrices and Set Linearization Points and has an Extracting State-Space Matrices.

THE STATE-SPACE SYSTEM

A state-space system is the mathematical representation of a physical model. The system consistent in an ODE linking input, output, and state-space variable. A dynamic system can be represented with the following system:

$$\begin{cases} \frac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

An alternative representation of the dynamic system is:

$$M_C\dot{x} = M_CAx + M_CBu$$
$$y = Cx + Du$$

where x is the state variable vector.

If the components of the mass matrix M_C are small, it is possible to approximate the dynamic state-space model with a static model, where $M_C\dot{x} = 0$:

$$y = (D - C(M_C A)^{-1} M_C B)u$$

Let *Null* be the PDE constraint null-space matrix and ud a particular solution fulfilling the constraints. The solution vector U for the PDE problem can then be written

$$U = \text{Null}x + ud + u_0$$

where u_0 is the linearization point, which is the solution stored in the sequence once the state-space export feature is run.

CHOOSING THE INPUT

The input parameters should contain all parameters that are of interest as input to the model. Moreover if you have any settings in model that are connected to the degrees of freedom, like a constraint condition or a spring condition. These have to be declared as input in your state-space system. When solving the state-space system in MATLAB, subtract to these inputs the initial value of the corresponding DOF, as it is done in the example `Extracting State-Space Matrices`.

EXTRACT STATE-SPACE MATRICES

The function `mphstate` requires that the input variables, output variables, and the list of the matrices to extract in the MATLAB workspace are all defined:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out);
```

where *<soltag>* is the solver node tag used to assemble the system matrices listed in the cell array *out*, and *<input>* and *<output>* are cell arrays containing the list of the input and output variables, respectively.

The output data `str` returned by `mphstate` is a MATLAB structure and the fields correspond to the assembled system matrices.

The input variables need to be defined as parameters in the COMSOL model. The output variables are defined as domain point probes or global probes in the COMSOL model.

The system matrices that can be extracted with `mphstate` are listed in the table:

EXPRESSION	DESCRIPTION
MA	McA matrix
MB	McB matrix
A	A matrix

EXPRESSION	DESCRIPTION
B	B matrix
C	C matrix
D	D matrix
Mc	Mc matrix
Null	Null matrix
ud	ud vector
x0	x0 vector

To extract sparse matrices set the property `sparse` to `on`:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'sparse', 'on')
```

To keep the state-space feature node, set the property `keepfeature` to `on`:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'keepfeature', 'on')
```

SET LINEARIZATION POINTS

`mphstate` uses linearization points to assemble the state-space matrices. The default linearization point is the current solution provided by the solver node, to which the state-space feature node is associated. If there is no solver associated to the solver configuration, a null solution vector is used as a linearization point unless you manually set the linearization point to an existing solution.

The linearization point needs to be a steady-state solution.

You can manually select the linearization point to use. Use the `initmethod` property to select a linearization point:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'initmethod', method)
```

where *method* corresponds to the type of linearization point — the initial value expression ('init') or a solution ('sol').

To set the solution to use for the linearization point, use the property `initsol`:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'initsol', <initsoltag>)
```

where *<initsoltag>* is the solver tag to use for a linearization point. You can also set the `initsol` property to 'zero', which corresponds to using a null solution vector as a linearization point. The default is the current solver node where the assemble node is associated.

For continuation, time-dependent, or eigenvalue analyses you can set which solution number to use as a linearization point. Use the `solnum` property:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'solnum', <solnum>)
```

where *<solnum>* is an integer value corresponding to the solution number. The default value is the last solution number available with the current solver configuration.

If there is a solver associated to the solver configuration *<soltag>*, you need to extract the matrices after the Dependent Variables node in the solver configuration, to proceed use the property `extractafter` as in the command below:

```
str = mphstate(model, <soltag>, 'input', <input>, ...
    'output', <output>, 'out', out, 'solnum', <solnum>, ...
    'extractafter', <vtag>)
```

where *<vtag>* is the tag of the Dependent Variable node.

SIMULATION USING THE CONTROL SYSTEM TOOLBOX

The Control System Toolbox makes it possible to analyze state space models in MATLAB for control design and to simulate such systems.

State space models can be defined using the `ss` or `sparss` function. The `sparss` function can be used if the state space matrices are sparse. The system can be simulated using the `lsim` function. In order to create a reduced order system using MATLAB we will use the `balred` function. This function only accepts the use of full matrices. Hence, the function `ss` is used for defining the state space system in MATLAB. Note that calling the function `ss` with argument matrices that are sparse result in a set of warnings. These warnings can be ignored.

The code below show how to use the Control System Toolbox to solve a dynamic system from the A, B, C, and D matrices obtained using `mphstate`:

```
sys = ss(M.A, M.B, M.C, M.D);
u = repmat(<input>, length(<tspan>), 1);
[y,t] = lsim(sys, u, t);
```

where *<input>* is the input vector of the state space system, *<tspan>* the time step list.

One may obtain a reduced order model using the `balred` function.

```
redsys = balred(sys, <order>);
[y,t] = step(redsys, <tfinal>);
```

where *<order>* is the desired order reduction, and *<tfinal>* the simulation end time.

EXTRACTING STATE-SPACE MATRICES

In this section you will find an example that illustrate how to use the `mphstate` function to extract the state-space matrices.

The problem studied here is a heat transfer model set with a heat source. In the expected state space system the heat source is set as input. The temperature at specified location is used as output. The tutorial shows how to extract the state space matrices and solve the system using the MATLAB functionalities.

This is the same problem solved as in Extracting Reduced Order State-Space Matrices so you can compare the solution, and computational performance when solving the problem with reduced order model state space system matrices.

First, load the model model_tutorial_llmatlab from the Application Library:

```
model = mphopen('model_tutorial_llmatlab');
```

This is model is used as base model for documentation tutorial. The geometry, mesh, physics is set but for this specific problem you need to edit the model to use parameter for the initial and external temperature as they will be used later in the state space matrices export:

```
power = 30; Temp = 300; Text = 300; T0 = 293.15;
model.param.set('power', power);
model.param.set('Temp', Temp);
model.param.set('T0', T0);
model.param.set('Text', Text);
comp1 = model.component('comp1');
ht = comp1.physics('ht');
ht.feature('init1').set('Tinit', 'T0');
ht.feature('hf1').set('Text', 'Text');
```

You need now to add a time dependent study:

```
std = model.study.create('std');
time = std.feature.create('time','Transient');
time.set('tlist','range(0,1,50)');
time.set('usertol',true);
time.set('rtol','1e-4');
```

Add a domain point probe plot, which will defines the output of the state space system:

```
pdom = comp1.probe.create('pdom', 'DomainPoint');
pdom.model('comp1');
```

```
pdom.set('coords3',[1e-2 1e-2 1e-2]);
```

Run the study and create a plot group to display the probe:

```
std.run;
pg1 = model.result.create('pg1', 'PlotGroup1D');
glob1 = pg1.create('glob1', 'Global');
glob1.set('expr', 'comp1.ppb1');
```

Extract the state-space system matrices `Mc`, `MA`, `MB`, `C`, and `D`, of the model with `power`, `Temp`, and `Text` as input and the probe evaluation `comp1.ppb1` as output:

```
M = mphstate(model,'sol1','out',{'Mc','MA','MB','C','D'},...
    'input',{'power','Temp','Text'},...
    'output','comp1.ppb1');
```

Set the input power parameter and the reference temperature:

```
input = [power Temp-T0 Text-T0];
```

You can noticed that some of the input value are subtracted with the initial condition. This is because these inputs are directly linked to the temperature degree of freedom using a constraint condition.

```
func = @(t,x) M.MA*x + M.MB*input';
```

Set the solver option such as the mass and the Jacobian:

```
opt = odeset('mass',M.Mc,'jacobian',M.MA);
```

It is not required to define the Jacobian, but doing so speeds up the simulation.

The state space formulation that uses the mass matrix are prepared to use sparse and hence can often be simulated much faster. In order to simulate systems that involve a mass matrix on the left hand side one must switch to one of the simulation functions that support a mass matrix. Such functions are ode15s, ode23s and ode23t. ode15s and ode23t can solve problems with a singular mass matrix. Compute the state-space system with the extracted matrices,

```
[t,x] = ode23s(func,0:1:50,zeros(size(M.MA,1),1),opt);
y = M.C*x';
y = y+T0;
```

Compare the solution computed with the system and the one computed with COMSOL Multiphysics (see Figure 4-1):

```
plot(t,y,'r+');
hold on;
mphplot(model,'pg1');
```

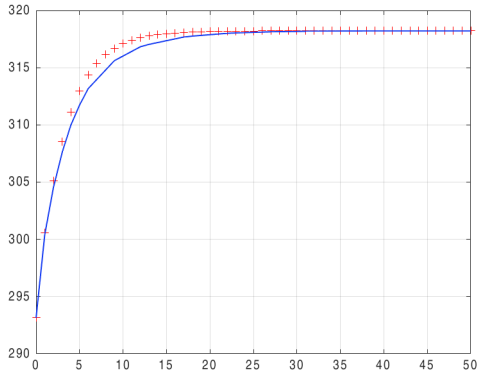


Figure 4-1: Temperature distribution computed with the state-space system (red marker) and COMSOL Multiphysics (blue line).

Evaluate the steady-state temperature value:

```
G = M.D-M.C*(inv(M.MA))*M.MB;
y_inf = full(G*input');
y_inf = y_inf + T0
```

For some types of control system design the use of the matrices A, B, C, and D are commonly used. Note that you need to keep the size of the matrices lower as you are not using a mass matrix. Hence, you need to change the size of mesh in order to reduce the number of degrees of freedom of the assembled system:

```
comp1.mesh('mesh1').autoMeshSize(7);
```

Extract the state-space system matrices A, B, C, and D, of the model with `power`, `Temp`, and `Text` as input and the probe evaluation `comp1.ppb1` as output:

```
M2 = mphstate(model,'sol1','out',{'A','B','C','D'},...
    'input',{'power','Temp','Text'},...
    'output','comp1.ppb1');
```

Compute the state-space system with the extracted matrices,

```
func = @(t,x) M2.A*x + M2.B*input';
[t,x] = ode45(func,0:1:50,zeros(size(M2.A,1),1));
y2 = M2.C*x';
y2 = y2+T0;
```

Compare the solution computed with the previous system (see Figure 4-2):

```
plot(t,y2,'k.')
```

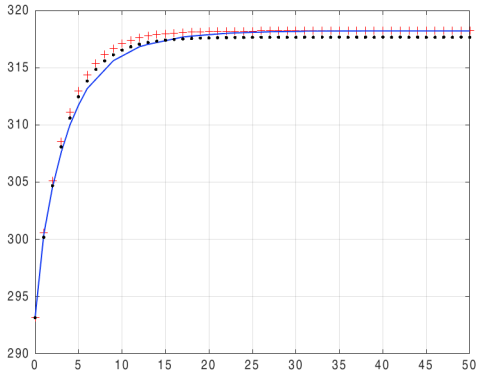


Figure 4-2: Temperature distribution computed with the state-space system 1 (red '+' marker), state-space system without mass matrix (black '.' marker) and COMSOL Multiphysics (blue line).

Code for use with MATLAB®

```
model = mphopen('model_tutorial_llmatlab');
power = 30; Temp = 300; Text = 300; T0 = 293.15;
model.param.set('power', power);
model.param.set('Temp', Temp);
model.param.set('T0', T0);
model.param.set('Text', Text);
comp1 = model.component('comp1');
ht = comp1.physics('ht');
ht.feature('init1').set('Tinit', 'T0');
ht.feature('hf1').set('Text', 'Text');
std = model.study.create('std');
time = std.feature.create('time','Transient');
time.set('tlist','range(0,1,50)');
time.set('usertol',true);
time.set('rtol','1e-4');
pdom = comp1.probe.create('pdom', 'DomainPoint');
pdom.model('comp1');
pdom.set('coords3',[1e-2 1e-2 1e-2]);
std.run;
pg1 = model.result.create('pg1', 'PlotGroup1D');
glob1 = pg1.create('glob1', 'Global');
glob1.set('expr', 'comp1.ppb1');
M = mphstate(model,'sol1','out',{'Mc','MA','MB','C','D'},...
   'input',{'power','Temp','Text'},'output','comp1.ppb1');
input = [power Temp-T0 Text-T0];
func = @(t,x) M.MA*x + M.MB*input';
opt = odeset('mass',M.Mc,'jacobian',M.MA);
```

```
[t,x] = ode23s(func,0:1:50,zeros(size(M.MA,1),1),opt);
y = M.C*x';
y = y+T0;
plot(t,y,'r+');
hold on;
mphplot(model,'pg1');
G = M.D-M.C*(inv(M.MA))*M.MB;
y_inf = full(G*input');
y_inf = y_inf + T0
comp1.mesh('mesh1').autoMeshSize(7);
M2 = mphstate(model,'sol1','out',{'A','B','C','D'},...
   'input',{'power','Temp','Text'},...
   'output','comp1.ppb1');
func = @(t,x) M2.A*x + M2.B*input';
[t,x] = ode45(func,0:1:50,zeros(size(M2.A,1),1));
y2 = M2.C*x';
y2 = y2+T0;
plot(t,y2,'k.')
```

Extracting Reduced Order State-Space Matrices

COMSOL Multiphysics models often have a large number of degrees of freedom. This leads to large state-space model when exported using `mphstate`. COMSOL Multiphysics provides model reduction functionality, which can reduce the number of states using an eigenvalue (or eigenfrequency) study leading to a low number of states that can be used for simulation and analysis.

The function `mphreduction` returns state-space matrices or a MATLAB state-space model using the `ss` function of a given reduced-order model set in the COMSOL model.

See the Modal Reduced-Order Model and The Modal Solver Algorithm in the *COMSOL Multiphysics Reference Manual* for more information.

EXTRACTING REDUCED ORDER STATE-SPACE MATRICES

In this section you will find an example that illustrate how to use the `mphreduction` function to extract the reduced order model state-space matrices.

The problem studied here is a heat transfer model set with a heat source. In the expected state space system the heat source is set as input. The temperature at specified location is used as output. The tutorial shows how to extract the state space matrices and solve the system using the MATLAB functionalities.

This is the same problem solved as in Extracting State-Space Matrices so you can compare the solution, and computational performance when solving the problem with full state space system matrices.

The following example sets up a model using commands from MATLAB. You need the Heat Transfer Module or MEMS Module to run the model, as reduced order modeling requires eigenvalue of the physical problem. If you don't have access to one of these modules, you can find the commands that sets up and solve the same problem using the General Form PDE interface in the section Code for use with MATLAB® - General Form PDE.

First, load the model model_tutorial_llmatlab from the Application Library:

```
model = mphopen('model_tutorial_llmatlab');
```

This is model is used as base model for documentation tutorial. The geometry, mesh, physics is set but for this specific problem you need to edit the model to use parameter for the initial and external temperature as they will be used later in the reduced order model:

```
power = 30; Temp = 300; Text = 300; T0 = 293.15;
model.param.set('power', power);
model.param.set('Temp', Temp);
model.param.set('T0', T0);
model.param.set('Text', Text);
comp1 = model.component('comp1');
ht = comp1.physics('ht');
ht.feature('init1').set('Tinit', 'T0');
ht.feature('hf1').set('Text', 'Text');
```

This model is defined with a temperature constraint, you need to replace it with by a heat flux condition as constraints are not supported as input for reduced order models:

```
ht.feature('temp1').active(false);
hf2 = ht.create('hf2', 'HeatFluxBoundary', 2);
hf2.selection().set(3);
hf2.set('HeatFluxType', 'ConvectiveHeatFlux');
hf2.set('h', '1e6');
hf2.set('Text', 'Temp');
```

You need now to add a time dependent study:

```
std1 = model.study.create('std1');
time = std1.feature.create('time','Transient');
time.set('tlist','range(0,1,50)');
```

Add a domain point probe plot, which will defines the output of the reduced order model:

```
pdom = comp1.probe.create('pdom', 'DomainPoint');
pdom.model('comp1');
pdom.set('coords3',[1e-2 1e-2 1e-2]);
```

Run the study and create a plot group to display the probe:

```
std1.run;
pg1 = model.result.create('pg1', 'PlotGroup1D');
glob1 = pg1.create('glob1', 'Global');
glob1.set('expr', 'comp1.ppb1');
```

A reduced order model requires eigenvalue solution of the problem. To create an eigenfrequency study and specify the number of eigenvalue to 30 enter:

```
std2 = model.study().create('std2');
eig = std2.create('eig', 'Eigenfrequency');
eig.activate('ht', true);
eig.set('neigsactive', true);
eig.set('neigs', 30);
eig.set('shiftactive', true);
std2.run;
```

Before adding a reduced-order study to your model you need to set the input, here in this example the power variable `power`, the bottom temperature `Temp`, and the exterior temperature `Text`:

```
grmi1 =
model.common.create('grmi1','GlobalReducedModelInputs','');
grmi1.setIndex('name', 'power', 0);
grmi1.setIndex('name', 'Temp', 1);
grmi1.setIndex('name', 'Text', 2);
```

Now you can add a model reduction study.

```
std3 = model.study.create('std3');
mr = std3.create('mr','ModelReduction');
mr.set('trainingStudy','std2');
mr.set('trainingStep','eig');
mr.set('unreducedModelStudy','std1');
mr.set('unreducedModelStep','time');
mr.setIndex('qoiname','Tout',0);
mr.setIndex('qoiexpr','comp1.ppb1',0);
std3.run;
```

Here the model reduction study generates a Time Dependent, Modal Reduced-Order Model node that you can identify with the tag `rom1`. To get the newly generated reduced-order model you can either type the command:

```
model.reduced;
```

or inspect the model using the **mphnavigator** window by typing:

```
mphnavigator
```

To visualize the reduced order node in the mphnavigator window, you need to go the Settings menu and select Advanced.

A call to `mphreduction` creates the state-space matrices needed to simulate the reduced-order system.

```
MR = mphreduction(model, 'rom1', ...
    'out', {'MA' 'MB' 'C' 'D' 'Mc' 'x0'})
```

Now the reduced-order system can be simulated using the function `ode23s`.

```
input = [power Temp-T0 Text-T0];
func = @(t,x) MR.MA*x + MR.MB*input';
opt = odeset('mass',MR.Mc);
x0 = zeros(size(MR.MA,1),1);
[t,x2] = ode23s(func,0:1:50,x0,opt);
y2t = MR.C*x2';
y20 = MR.C*MR.x0;
y2 = y2t+y20;
```

Compare the solution computed with the system and the one computed with COMSOL Multiphysics (see Figure 4-3):

```
plot(t,y2,'r+');
hold on;
mphplot(model,'pg1');
```

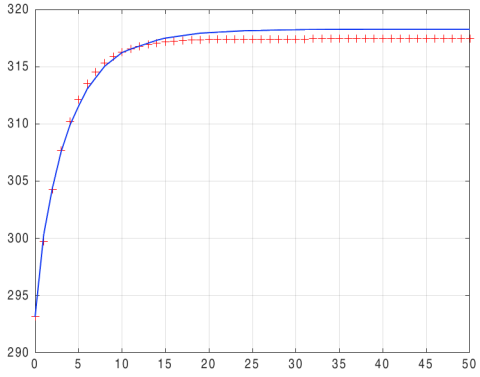


Figure 4-3: Temperature distribution computed with the reduced order model state-space system (red marker) and COMSOL Multiphysics (blue line).

Evaluate the steady-state temperature value:

```
G = -MR.C*(inv(MR.MA))*MR.MB;
y_inf = full(G*input');
y_inf = y_inf + T0
```

As an alternative to using `ode23s`, you can use functions in the Control System Toolbox, which is an add-on to MATLAB. The matrices stored in `MR` can be used to manually construct a state-space system using the function `ss`, or you can call `mphreduction` using the return option to specify that `mphreduction` should do the conversion.

```
sys = mphreduction(model, 'rom1', ...
    'out', {'MA' 'MB' 'A' 'B' 'C' 'D' 'Mc' 'x0' 'Y0' 'Kr'}, ...
    'return', 'ss')
```

The system can, for example, be simulated using the `lsim` function.

```
t = 0:1:50;
u = repmat(input,length(t),1);
figure(2)
[y,t] = lsim(sys, u,t);
grid on
```

Code for use with MATLAB®

```
model = mphopen('model_tutorial_llmatlab');
power = 30; Temp = 300; Text = 300; T0 = 293.15;
model.param.set('power', power);
model.param.set('Temp', Temp);
model.param.set('T0', T0);
model.param.set('Text', Text);
comp1 = model.component('comp1');
ht = comp1.physics('ht');
ht.feature('init1').set('Tinit', 'T0');
ht.feature('hf1').set('Text', 'Text');
ht.feature('temp1').active(false);
hf2 = ht.create('hf2', 'HeatFluxBoundary', 2);
hf2.selection().set(3);
hf2.set('HeatFluxType', 'ConvectiveHeatFlux');
hf2.set('h', '1e6');
hf2.set('Text', 'Temp');
std1 = model.study.create('std1');
time = std1.feature.create('time','Transient');
time.set('tlist','range(0,1,50)');
pdom = comp1.probe.create('pdom', 'DomainPoint');
pdom.model('comp1');
pdom.set('coords3',[1e-2 1e-2 1e-2]);
std1.run;
pg1 = model.result.create('pg1', 'PlotGroup1D');
glob1 = pg1.create('glob1', 'Global');
```

```
glob1.set('expr', 'comp1.ppb1');
std2 = model.study().create('std2');
eig = std2.create('eig', 'Eigenfrequency');
eig.activate('ht', true);
eig.set('neigsactive', true);
eig.set('neigsactive', true);
eig.set('neigs', 30);
eig.set('shiftactive', true);
std2.run;
grmi1 =
model.common.create('grmi1','GlobalReducedModelInputs','');
grmi1.setIndex('name', 'power', 0);
grmi1.setIndex('name', 'Temp', 1);
grmi1.setIndex('name', 'Text', 2);
std3 = model.study.create('std3');
mr = std3.create('mr','ModelReduction');
mr.set('trainingStudy','std2');
mr.set('trainingStep','eig');
mr.set('trainingRecompute','always');
mr.set('unreducedModelStudy','std1');
mr.set('unreducedModelStep','time');
mr.setIndex('qoiname','Tout',0);
mr.setIndex('qoiexpr','comp1.ppb1',0);
std3.run;
MR = mphreduction(model, 'rom1', ...
    'out', {'MA' 'MB' 'C' 'D' 'Mc' 'x0'})
input = [power Temp-T0 Text-T0];
func = @(t,x) MR.MA*x + MR.MB*input';
opt = odeset('mass',MR.Mc);
x0 = zeros(size(MR.MA,1),1);
[t,x2] = ode23s(func,0:0.1:50,x0,opt);
y2t = MR.C*x2';
y20 = MR.C*MR.x0;
y2 = y2t+y20;
plot(t,y2,'r+');
hold on;
mphplot(model,'pg1');
G = -MR.C*(inv(MR.MA))*MR.MB;
y_inf = full(G*input');
y_inf = y_inf + T0
```

Code for use with MATLAB® - General Form PDE

```
model = mphopen('model_tutorial_llmatlab');
power = 30; Temp = 300; Text = 300; T0 = 293.15;
model.param.set('power', power);
model.param.set('Temp', Temp);
model.param.set('T0', T0);
model.param.set('Text', Text);
comp1 = model.component('comp1');
comp1.physics('ht').active(false);
```

```
g = comp1.physics.create('g', 'GeneralFormPDE', {'u'});
g.prop('Units').set('DependentVariableQuantity', 'temperature');
g.prop('Units').setIndex('CustomSourceTermUnit', 'W/m^3', 0, 0);
gfeq1 = g.feature('gfeq1');
gfeq1.setIndex('Ga', {'-material.k11*ux' '-material.k22*uy'
'-material.k33*uz'}, 0);
g.feature('init1').set('u', 'T0');
gfeq1.setIndex('f', 0, 0);
gfeq1.setIndex('da', 'material.rho*material.Cp', 0);
src1 = g.create('src1', 'SourceTerm', 3);
src1.selection.set([2]);
src1.setIndex('f', 'power/(1[cm]*1[cm]*1[mm])', 0);
flux1 = g.create('flux1', 'FluxBoundary', 2);
flux1.selection.set([7 8 10 11 12]);
flux1.setIndex('g', '10[W/(m^2*K)]*(Text-u)', 0);
flux2 = g.create('flux2', 'FluxBoundary', 2);
flux2.selection.set([3]);
flux2.setIndex('g', '1e6[W/(m^2*K)]*(Temp-u)', 0);
std1 = model.study.create('std1');
time = std1.feature.create('time','Transient');
time.set('tlist','range(0,1,50)');
pdom = comp1.probe.create('pdom', 'DomainPoint');
pdom.model('comp1');
pdom.set('coords3',[1e-2 1e-2 1e-2]);
std1.run;
pg1 = model.result.create('pg1', 'PlotGroup1D');
glob1 = pg1.create('glob1', 'Global');
glob1.set('expr', 'comp1.ppb1');
std2 = model.study().create('std2');
eig = std2.create('eig', 'Eigenfrequency');
eig.activate('ht', true);
eig.set('neigsactive', true);
eig.set('neigsactive', true);
eig.set('neigs', 30);
eig.set('shiftactive', true);
std2.run;
grmi1 =
model.common.create('grmi1','GlobalReducedModelInputs','');
grmi1.setIndex('name', 'power', 0);
grmi1.setIndex('name', 'Temp', 1);
grmi1.setIndex('name', 'Text', 2);
std3 = model.study.create('std3');
mr = std3.create('mr','ModelReduction');
mr.set('trainingStudy','std2');
mr.set('trainingStep','eig');
mr.set('trainingRecompute','always');
mr.set('unreducedModelStudy','std1');
mr.set('unreducedModelStep','time');
mr.setIndex('qoiname','Tout',0);
mr.setIndex('qoiexpr','comp1.ppb1',0);
```

```
std3.run;
MR = mphreduction(model, 'rom1', ...
    'out', {'MA' 'MB' 'A' 'B' 'C' 'D' 'Mc' 'x0'})
input = [power Temp-T0 Text-T0];
func = @(t,x) MR.MA*x + MR.MB*input';
opt = odeset('mass',MR.Mc);
x0 = zeros(size(MR.MA,1),1);
[t,x2] = ode23s(func,0:1:50,x0,opt);
y2t = MR.C*x2';
y20 = MR.C*MR.x0;
y2 = y2t+y20;
plot(t,y2,'r+');
hold on;
mphplot(model,'pg1');
G = -MR.C*(inv(MR.MA))*MR.MB;
y_inf = full(G*input');
y_inf = y_inf + T0
```

Extracting Solution Information and Solution Vectors

In this section:

Obtaining Solution Information

Get the solution object information with the function `mphsolinfo`. Specify only the model object to obtain the information of the default solution object:

```
info = mphsolinfo(model)
```

This section includes information about Specifying the Solution Object and the Output Format.

The function `mphsolinfo` replaces the function `mphgetp`. If you are using the later you can now replace it as it will be removed in a future version.

SPECIFYING THE SOLUTION OBJECT

To retrieve the information of a specific solution object, set the `soltag` property with the solver tag *`soltag`* associated to the solution object:

```
info = mphsolinfo(model, 'soltag', <soltag>)
```

If there are several solution datasets attached to the solver, for example, solution datasets with different selections, specify the dataset to use to get the solution object information with the `dataset` property:

```
info = mphsolinfo(model, 'dataset', <dsettag>)
```

where *`dsettag`* the tag of the solution dataset to use.

OUTPUT FORMAT

The output info is a MATLAB® structure. The default fields available in the structure are listed in the table:

FIELDS	DESCRIPTION
`soltag`	Tag of the solver associated to the solution object
`study`	Tag of the study associated to the solution object
`size`	Size of the solution vector
`nummesh`	Number of mesh in the solution (for automatic remeshing)
`sizes`	Size of solution vector and inner parameters for each mesh
`soltype`	Solver type
`solpar`	Parameter name
`sizesolvals`	Length of parameter list
`solvals`	Inner parameter value
`paramsweepnames`	Outer parameter name
`paramsweepvals`	Outer parameter value
`label`	Solution node label
`batch`	Batch information
`dataset`	Tag of the solution dataset associated to the solution object

To get the information about the number of solutions, set the property `nu` to `on`:

```
info = mphsolinfo(model, 'nu', 'on')
```

The `info` structure is added with the following fields:

FIELDS	DESCRIPTION
`NUsol`	Number of solutions vectors stored
`NUreacf`	Number of reaction forces vectors stored
`NUadj`	Number of adjacency vectors stored
`NUfsens`	Number of functional sensitivity vectors stored
`NUsens`	Number of forward sensitivity vectors stored

The `batch` field is a structure including the following fields:

BATCH FIELDS	DESCRIPTION
`type`	The type of batch
`psol`	Tag of the associated solver node

BATCH FIELDS	DESCRIPTION
`sol`	Tag of the stored solution associated to psol
`seq`	Tag of the solver sequence associated to psol

Retrieving Solution Information and Solution Datasets Based on Parameter Values

A model can contain several solution vectors computed with different values of parameters, such as time, eigenvalue, or model parameters. These solution vectors can be available in different solution datasets. Use the function `mphsolutioninfo` to retrieve the solution vector corresponding to a specified study parameter value.

The parameters used in a study can be group in two distinct solution number types:

- The *inner solution*, containing the solution computed with parameters such as eigenvalues, time steps, or continuation parameter combinations.
- The *outer solution*, containing the solution computed with parameters defined in parametric sweep.

To get information about all solution object and solution dataset combinations in the model enter the command:

```
info = mphsolutioninfo(model)
```

The output `info` is a structure containing these fields:

FIELDS	DESCRIPTION
`solutions`	List of the solution object tags available in the model
`sol#`	Substructure containing information related to the solution number #

The substructure `info.sol#` has these fields:

FIELDS	DESCRIPTION
`dataset`	List of the tags of the dataset associated to the solution
`study`	Tag of the study that computed the solution
`sequencetype`	Type of solution node
`cellmap`	Connections between parameters and inner/outer solution numbers; the field is not available by default.
`values`	Values of the parameters used in the solution
`parameters`	Names of the parameters used in the solution

FIELDS	DESCRIPTION
`mapheaders`	Headers of the table stored in the map field
`map`	Connections between the parameter values and the solution number (inner and outer solutions)

You can also retrieve the solution objects and solution datasets related to a specific parameter value with the command:

```
info = mphsolutioninfo(model,'parameters',{'e1','v1','tol1'})
```

where *e1* is the expression name, *v1* the value of the expression.

The property parameters can also be set as a 1x*N* cell array, where *N* corresponds to the number of parameters to specify.

This section includes information about Specifying the Solution Object and the Output Format. It also includes the section, Retrieving Solution Information.

SPECIFYING THE SOLUTION OBJECT

To retrieve the information of a specific solution object, set the `soltag` property with the solver tag *soltag* associated to the solution object:

```
info = mphsolutioninfo(model, 'soltag', <soltag>)
```

If there are several solution datasets attached to the solver, for example, solution datasets with different selections, specify the dataset to use to get the solution object information with the `dataset` property:

```
info = mphsolutioninfo(model, 'dataset', <dsettag>)
```

where *dsettag* the tag of the solution dataset to use.

OUTPUT FORMAT

To include the `cellmap` field in the `info.sol#` substructure set the property `cellmap` to `on`:

```
info = mphsolutioninfo(model, 'cellmap', 'on')
```

Improve the visibility of the map table by sorting the row using either the column number or the name in the map header:

```
info = mphsolutioninfo(model, 'sort', <idx>)
```

where *<idx>* is a positive integer equal to the column number or a string corresponding to the name of the column header.

RETRIEVING SOLUTION INFORMATION

This example shows how to use the function `mphsolutioninfo` to retrieve solution information in a mode combining a parametric sweep and transient analysis.

Start by loading the base model `model_tutorial_llmatlab` from the COMSOL Multiphysics Application Libraries; this model contains base settings for a thermal analysis:

```
model = mphopen('model_tutorial_llmatlab');
```

Now create a study combining a parametric sweep and a transient study step. The parametric sweep consist by varying the parameters that set the heat source and the bottom temperature. This is done with these commands:

```
std = model.study.create('std');
param = std.feature.create('param', 'Parametric');
param.setIndex('pname', 'power', 0);
param.setIndex('plistarr', '30 60 90',0);
param.setIndex('pname', 'Temp', 1);
param.setIndex('plistarr', '300 320', 1);
time = std.feature.create('time', 'Transient');
time.set('tlist', 'range(0,1,25)');
```

Set the sweep type to generate all possible combinations of the parameters `power` and `tf` and compute the study:

```
param.set('sweeptype', 'filled');
std.run;
```

Once the solution is computed (it takes about 90 seconds), you can retrieve the solution information in the model:

```
info = mphsolutioninfo(model)
```

The output `info` is a structure containing nine fields. By navigating in the `info` structure you can retrieve how the solutions are stored in the model.

- `info.sol1` contains the solution information related to the solver sequence `sol1`. The associated dataset is `dset1`.
- `info.sol2` contains the solution information for the parametric sequence. This regroups the solution vectors computed for all outer parameters.

The other substructures contain the solution information for all possible outer solution combinations.

Get the relation between the parameter values and the inner and outer solution numbers:

```
map = info.sol2.map
```

Retrieve the solution information related to the parameters power = 60 W:

```
info = mphsolutioninfo(model, 'parameters', {'power',60,0})
```

Retrieve the solution information related to the parameters power = 60 W, Temp = 300 K and t = 10.4 seconds, for the time use a tolerance of 0.5 seconds to find the appropriate inner solution number:

```
info = mphsolutioninfo(model, 'parameters', {{'power',60,0},...
{'Temp',300,0},{'t',10.4,0.5}})
```

To get the list of the solutions that contain the given parameters enter:

```
solnum = info.solutions
```

Code for use with MATLAB®

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param', 'Parametric');
param.setIndex('pname', 'power', 0);
param.setIndex('plistarr', '30 60 90',0);
param.setIndex('pname', 'Temp', 1);
param.setIndex('plistarr', '300 320', 1);
time = std.feature.create('time', 'Transient');
time.set('tlist', 'range(0,1,25)');
param.set('sweeptype', 'filled');
std.run;
info = mphsolutioninfo(model)
map = info.sol2.map
info = mphsolutioninfo(model, 'parameters', {'power',60,0})
info = mphsolutioninfo(model, 'parameters', {{'power',60,0},...
{'Temp',300,0},{'t',10.4,0.5}})
solnum = info.solutions
```

Extracting Solution Vector

To extract the solution vector with the function `mphgetu`, enter:

```
U = mphgetu(model)
```

where `U` is an Nx1 double array, where N is the number of degrees of freedom of the COMSOL Multiphysics model.

This section includes information about Specifying the Solution and the Output Format.

You can refer to the function `mphxmeshinfo` to receive the DOF name or the node coordinates in the solution vector, see Retrieving Xmesh Information.

SPECIFYING THE SOLUTION

Change the solver node to extract the solution vector with the property `solname`:

```
U = mphgetu(model, 'soltag', <soltag>)
```

where *<soltag>* is the tag of the solver node.

For solver settings that compute for several inner solutions, select the inner solution to use with the `solnum` property:

```
U = mphgetu(model, 'solnum', <solnum>)
```

where *<solnum>* a positive integer vector that corresponds to the solution number to use to extract the solution vector. For time-dependent and continuation analyses, the default value for the `solnum` property is the last solution number. For an eigenvalue analysis, it is the first solution number.

A model can contain different types of solution vectors — the solution of the problem, the reaction forces vector, the adjoint solution vector, the functional sensitivity vector, or the forward sensitivity. In `mphgetu`, you can specify the type of solution vector to extract with the `type` property:

```
U = mphgetu(model, 'type', type)
```

where *type* is one of these strings `'sol'`, `'reacf'`, `'adj'`, or `'sens'` used to extract the solution vector, the reaction forces, the functional sensitivity, or the forward sensitivity, respectively.

OUTPUT FORMAT

`mphgetu` returns the default the solution vector. Get the time derivative of the solution vector `Udot` by adding a second output variable:

```
[U, Udot] = mphgetu(model)
```

In case the property `solnum` is set as a 1xM array and the solver node only uses one mesh to create the solution, the default output is an NxM array, where N is the number of degrees of freedom of the model. Otherwise, the output `U` is a cell array that contains

each solution vector. If you prefer to have the output in a cell array format, set the property `matrix` to `off`:

```
U = mphgetu(model, 'solnum', <solnum>, 'matrix', 'off')
```

Retrieving Xmesh Information

Use LiveLink™ *for* MATLAB® to retrieve low level information of the COMSOL Multiphysics finite element model.

In this section:

The Extended Mesh (Xmesh)

The extended mesh (xmesh) is the finite element mesh used to compute the solution. This contains the information about elements, nodes, and degrees of freedom such as DOF names, position of the nodes in the assembled matrix system, or how elements and nodes are connected.

Extracting Xmesh Information

The function `mphxmeshinfo` returns the extended mesh information. To get the xmesh information of the current solver and mesh node, enter the command:

```
info = mphxmeshinfo(model)
```

where info is a MATLAB structure that contains the fields in the table:

FIELDS	DESCRIPTION
`soltag`	Tag of the solver node
`ndofs`	Number of degrees of freedom
`fieldnames`	List of field variables names
`fieldndofs`	Number of degrees of freedom for each field variable
`meshtypes`	List of the mesh type
`geoms`	Tag of the geometry node used in the model
`dofs`	Structure containing the dofs information
`nodes`	Structure containing the nodes information
`elements`	Structure containing the elements information

The `dofs` substructure contains the fields listed in the table:

FIELDS	DESCRIPTION
`geomnums`	1-based geometry numbers for all DOFs.
`coords`	Global coordinates for all DOFs in the model length unit. The `k`th column of this matrix contains the coordinates of DOF number `k`.
`nodes`	0-based node numbers for all DOFs.
`dofnames`	DOF names.
`nameinds`	0-based indices into `dofNames()` for all DOFs.
`solvectorinds`	0-based indices into solution vector for all DOFs.

The `nodes` substructure contains the fields listed in the table:

FIELDS	DESCRIPTION
`coords`	Global coordinates for all nodes. The `n`th column contains the coordinates of node point number `n`.
`dofnames`	DOF names in this geometry.
`dofs`	0-based DOF numbers for all nodes in this geometry. `dofs()[k][n]` is the DOF number for DOF name `dofNames()[k]` at node point `n`. A value of `-1` means that there is no DOF with this name at the node. Note: If there is a slit, only one of the DOFs is given for each node point.

The `elements` substructure contains the fields listed in the table:

FIELDS	DESCRIPTION
`meshtypes`	List of the type of mesh available.
`type`	Substructure containing the information of element of type *`type`*.

The *`type`* substructure lists the information for each element. The possible mesh types are `vtx`, `edg`, `quad`, `tri`, `quad`, `tet`, `hex`, `prism`, and `pyr`. The substructure *`type`* contains the fields listed in the table:

FIELDS	DESCRIPTION
`localcoords`	Local coordinates of nodes. The `k`th column contains the coordinates of local node point number `k`.
`localdofcoords`	The local coordinates for each local DOF (one column for each local DOF).
`localdofnames`	The name for each local DOF.

FIELDS	DESCRIPTION
`nodes`	0-based node point indices for all mesh elements of type *type*. A value -1 means that there is no node point at this location.
`dofs`	0-based DOF numbers for all mesh elements of type *type*. A value -1 means that there is no DOF at this location.

SPECIFY THE INFORMATION TO RETRIEVE

To specify the solver node to retrieve the xmesh information, set the property `soltag` as in this command:

```
info = mphxmeshinfo(model, 'soltag', <soltag>)
```

where *<soltag>* is the tag of the solver used to extract the xmesh information.

To retrieve the xmesh information for a specific study step node, specify it with the property `studysteptag`:

```
info = mphxmeshinfo(model, 'studysteptag', <studysteptag>)
```

where *<studysteptag>* is the tag of either a compiled equation node or a variable node.

In case several mesh cases have been used by a specific solver, for example, with an automatic remeshing procedure, you can specify which mesh case to use to get the discretization information:

```
info = mphxmeshinfo(model, 'meshcase', <meshcase>)
```

where *<meshcase>* is the mesh case number or the tag of the mesh case.

Navigating the Model

The model object contains all the finite element model settings. To retrieve the model information you can navigate in the model object using a graphical user interface or directly at the MATLAB® prompt. Learn how to get the list of predefined expressions available for a given model and how to extract the value of these expressions and also the properties of the method used in the model.

In this section:

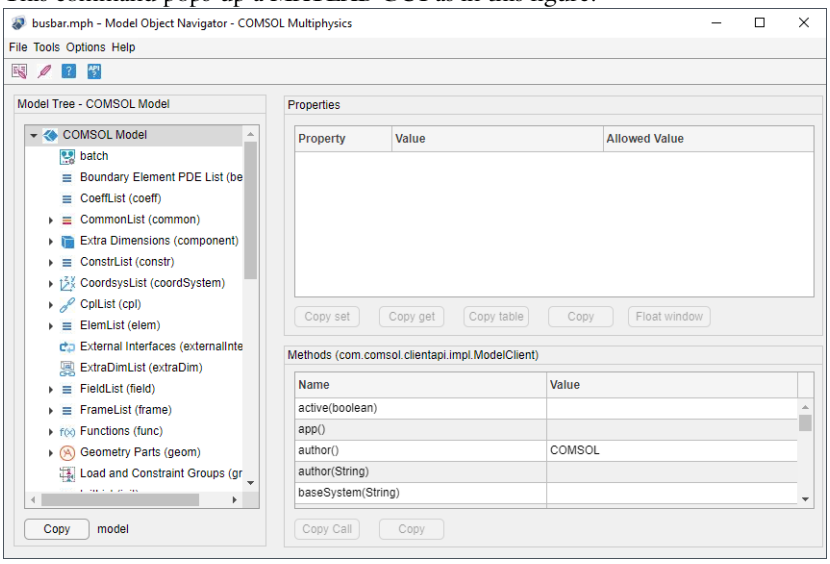
Navigating the Model Object Using a GUI

The usual approach to navigate through the model object in a graphical user interface (GUI) is to load the model object at the COMSOL Desktop. Then transfer the model object from the COMSOL Multiphysics Server to the COMSOL Desktop as in Sharing the Model Between the COMSOL Desktop® and the MATLAB® Prompt.

An alternative approach is to call the function `mphnavigator` that displays the model object information in a MATLAB® GUI. To run the function at the MATLAB prompt enter the command:

```
mphnavigator
```

This command pops-up a MATLAB GUI as in this figure:



If you have installed the COMSOL apps in the MATLAB Apps ribbon, click the COMSOL Model Navigator icon ( ).

Using the COMSOL Model Navigator apps ( ) only model objects with the name `model` are supported.

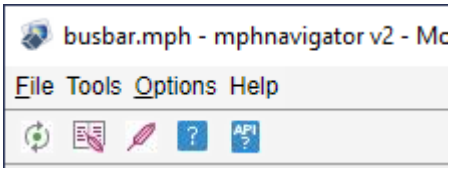
If the COMSOL model objected is not stored in the MATLAB variable `model` enter the command:

```
mphnavigator(<modelvar>)
```

where *<modelvar>* is the variable name in MATLAB that contains the model.

The appearance and behavior of the mphnavigator window depend on the version of MATLAB. Using MATLAB version 2020a or newer will show an updated user interface based on App Designer. These user interfaces can be resized and in many cases support a faster workflow.

THE MENU BAR ITEMS



In the **File** menu you will find the file handling commands such as **Open** and **Save as** to load a new model object in the COMSOL Multiphysics server and update the Model Object Navigator window, and save the current model object in the MPH-format respectively. If you want to visualize the current model object in the COMSOL Desktop click **Launch COMSOL Multiphysics**. Click **Show model thumbnail**, to show the model thumbnail in a separate window.

In the **Tools** menu you will find additional functionalities to inspect the model. Click **Plot** to displays the geometry, the mesh, or a plot group in a MATLAB figure. This button is only active when one of the corresponding node in the model tree is selected. Select **Search** to open the Model Search window where to search expressions or tags in the model object (see Finding Model Expressions). Select **Solutions** to open the Solution Info window that displays the solution object available in the COMSOL Multiphysics model object. Select **Warnings and Errors** to list the error or warning nodes available in the model object (see Handling Errors and Warnings). Click **Add report...** (also accessible with the shortcut icon below the menu list) generate report node in the model, you can choose between brief, intermediate or complete report. Select **Write report** (also accessible using the icon) to write a report, that has been previously added to the model, in a document. This opens the Run Report window where you can specify the report node to run and the report settings such as the filename and the output format.

Using Livelink for MATLAB, it is quite common to run specific node in loop. To get automatically the corresponding code ready to be pasted in a code editor click **Copy code: loop**. This will generate the following commands:

```
tag1 = <feattag1>;
tag2 = <feattag2>;
tags = cell(model.geom(tag1).feature(tag2).feature.tags);
for i=1:length(tags)
obj = model.geom(tag1).feature(tag2).feature(tags{i});
end
```

where *<feattag1>* and *<feattag2>* are the feature tags to reach the selected feature in the model.

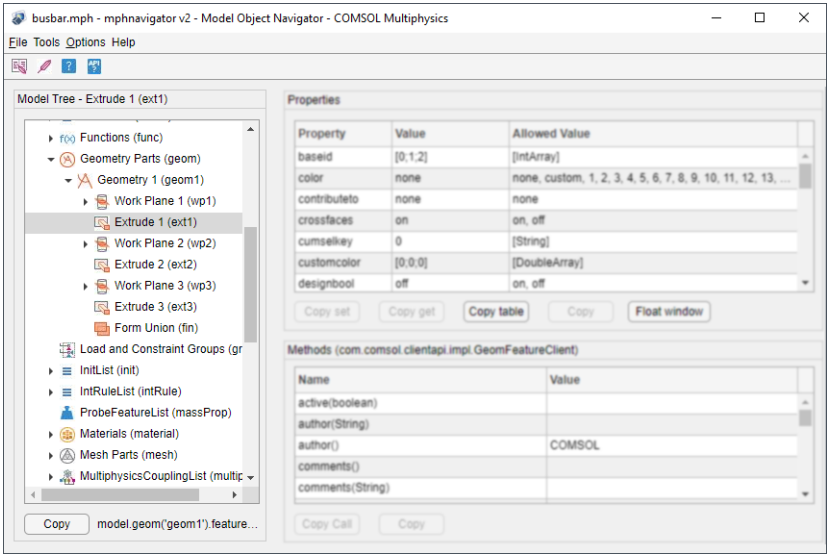
In th he **Options** menu you can set some preferences for the Model Object Navigator window. Select **Automatic update** to get the model tree and properties automatically updated as the current model object is changed. A manual update can be achieved by pressing on the toolbar.

Select **Use component syntax**, to include the component in the COMSOL API command. To get the property value as a string when clicking **Copy get** button, select **Get property as string**. You can also select **Show name** to display the node name in the model tree, instead of the tag node only.

The **Help** menu you can get the help of a selected node in the model tree (**Help** button or) as well as the corresponding API help (**API Help** button or).

THE MODEL TREE SECTION

The **Model Tree** section has the list of the nodes of the model object. Use the scrollbar to the right to scroll down the list, and click the + icon to expand the model object feature nodes.

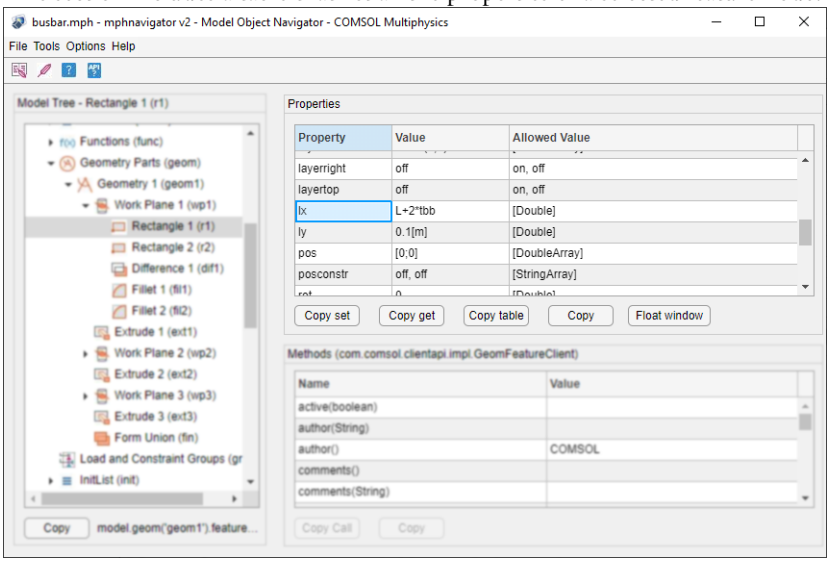


When a feature node is selected, its associated command is listed just beneath the model tree. Click **Copy** to copy syntax to the clipboard and then paste it in your script.

The **Model Tree** list is different from the **Model Builder** tree available in the COMSOL Desktop. This is because `mphnavigator` displays all feature nodes as seen in the COMSOL API which does not use the same filter as in the COMSOL Desktop to order the available feature nodes.

THE PROPERTIES SECTION

This section includes a table that list all the properties of a selected feature node.



The **Properties** and the **Value** columns list the properties of the selected feature node and the associated values respectively. The **Allowed Value** column list the allowed value for the corresponding property.

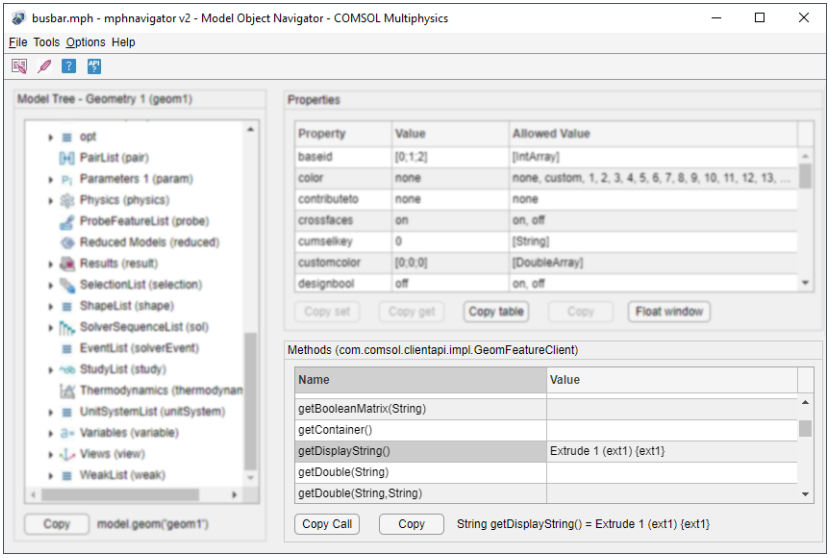
Not all feature node returns a list of allowed value for the properties.

Click **Copy set**, or **Copy get**, to copy to the clipboard the command that sets the selected property to its current value, respectively get the value of the currently selected property. Click **Copy Table** to copy the entire properties table to the clipboard, then

paste into a text or spreadsheet editor. To copy a selected cell in the table, click **Copy**. Using the **Float window** button you can open the Properties table in a separate window.

THE METHODS SECTION

The **Methods** section lists all the methods associated to the feature node selected in the Model Tree section.



Select a method in the list to get its associated syntax at the button of the **Methods** section. Use the **Copy** button to copy the syntax to the clipboard. Click **Copy call** to copy the method syntax associated to the selected feature node, the syntax is ready to use at the MATLAB prompt.

Navigating The Model Object At The Command Line

Use the command `mphmodel` at the MATLAB® prompt to retrieve model object information, such as tags for nodes and subnodes of a COMSOL Multiphysics model object.

To get the list of the main feature nodes and the tags of the model object `model`, enter the command:

```
mphmodel(model)
```

To list the subfeature of the node type `model.feature` enter the command:

```
mphmodel(model.feature)
```

To list the subfeature node of the feature node `model.feature(<ftag>)`, enter:

```
mphmodel(model.feature(<ftag>))
```

Use the flag `-struct` to return the model object information to MATLAB structure:

```
str = mphmodel(model,'-struct')
```

str is a MATLAB structure and the fields consist of each feature node associated to the root node.

Retrieving Component Information

A model can contain several components, each component having their own space dimension. Each component can have several geometry and mesh nodes. The geometric model used by the physics in each component can be set using either the geometric node or one of the meshes. Use the function mphcomponentinfo to get the component information.

```
info = mphcomponentinfo(model,<comptag>)
```

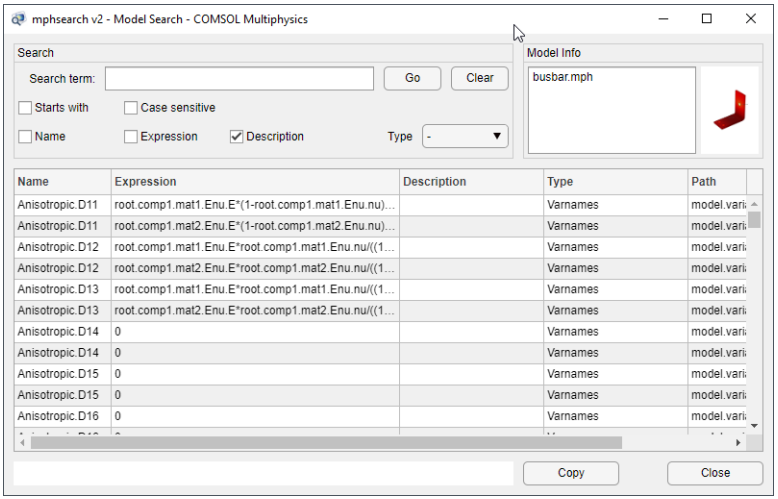
where *<comptag>* is the tag of the component. `info` is a MATLAB structure with the following fields:

- **tag**, the tag of the component (*<comptag>*),
- **geom**, the list of geometries in the component,
- **mesh**, the list of meshes in the component,
- **geometricmodel**, the geometry/mesh node that defines the geometric model for the physics,
- **scope**, the scope expression,
- **basesystem**, the base unit system used in the component,
- **sorder**, the shape order,
- **geometrycoord**, the coordinate variables for the geometry frame,
- **mateialcoord**, the coordinate variables for the material frame,
- **meshcoord**, the coordinate variables for the mesh frame,
- **spatialcoord**, the coordinate variables for the spatial frame.

Finding Model Expressions

Each model object contains predefined expressions that depend on the physics interface used in the model. Use the Model Search window to get the list of all the expressions, constants, solution variables, or parameters available in the model object. To open the Model Search window type at the MATLAB prompt:

```
mphsearch
```



The Model Search window shows all the variables available in the model. These are listed in the table containing the following information: the **Name** of the expression, the **Expression** as it is set in the property value, the **Description** (if there is one) set for the expression, the**Type** of the expression, and the **Path** in the model object.

If you have installed the COMSOL apps in the MATLAB Apps ribbon, click the COMSOL Search icon ( ).

Using the COMSOL Model Search apps( ) only model object with name `model` are supported.

If the COMSOL model objected is linked with a MATLAB object with a different name than `model`, enter the command:

```
mphsearch(<modelname>)
```

where *<modelname>* is the name of the model object link in MATLAB.

The **Search** section has a search tool to filter the list. Enter any string in the text field and select where to search the string — in the name, the expression, or the description of the table entry. You can also select the type you want to list. The expression type can be Equation, Field, Function, Geom, Mesh, Tag, VarNames, or Weak. You can select **Starts with** to search for any expression that start with the string enter in the text field. Click **Go** to display the result of the search. Click **Clear** to clear the search settings.

Additionally you can copy the selected cell in the table to the clipboard just by clicking **Copy**.

Evaluating the Model Parameters

Use the command `mphevaluate` to evaluate the expression defined in the Parameters node as in the command below:

```
value = mphevaluate(model, <expr>)
```

The evaluation does not require an existing solution dataset in the model, which means you can evaluate the expression even if there is no solution computed in the model.

To evaluate multiple expressions at once, define *<expr>* as a cell array of expressions as in the command below:

```
str = mphevaluate(model, {<expr1>, <expr2>, ...})
```

The output `str` is an array of structs with the same size as for the cell array of expressions. For multiple parameters evaluation only a single output is permitted. The struct contains the following fields: **name**, the parameter name; **value**, its value in the unit specified in the field **unit**; **def**, the string for the definition of the parameter; and **descr** the string description of the parameter.

To evaluate all the parameters defined in the table enter:

```
str = mphevaluate(model)
```

The output `str` is an array of structs with the same size as for the parameters table.

Get the full information of a model parameter expression with the command:

```
[value, unit, def, descr] = mphevaluate(model, <expr>)
```

where `value`, `unit`, `def`, and `descr` are the parameter value, the parameter unit, the parameter definition and the parameter description respectively.

You can specify the unit of the expression to evaluate with the command:

```
value = mphevaluate(model, <expr>, <unit> )
```

where *<unit>* is the unit to evaluate the expression *<expr>* in.

To evaluate and return only the value of the expression use the command:

```
value = mphevaluate(model, <expr>, <unit>, 'value')
```

To evaluate and return only the unit of the expression use the command:

```
value = mphevaluate(model, <expr>, <unit>, 'unit')
```

To evaluate and return the definition string of the expression use the command:

```
value = mphevaluate(model, <expr>, <unit>, 'valueunit')
```

The above command can be used to set a new parameter using an existing one.

Getting Feature Model Properties

Use the command `mphgetproperties` to extract at the MATLAB® prompt the properties of a specified node of the model object. Not all nodes contains properties. Use the command:

```
str = mphgetproperties(model.<feature>)
```

where `str` is a MATLAB structure that lists all the properties and the value of the feature node *<feature>*.

Some model node provides also a list of allowed value for their properties, to get such a list enter the command:

```
[str, allowed] = mphgetproperties(model.<feature>)
```

this also returns the MATLAB structure `allowed` containing the list of the allowed values for the properties of the feature node *<feature>*.

If you want to filter the properties in the output use the property `propnames` as in the command below:

```
str = mphgetproperties(model.<feature>,'propnames',{<prop>,...})
```

where *<prop>* is the property name to show only in the output structure.

Set the property `returnstrings` to `off` to get the property data not as string as in the command below:

```
str = mphgetproperties(model.<feature>,'returnstrings','off',...)
```

You can get the selection of the properties with the property `showsel` set to `on` as described below:

```
str = mphgetproperties(model.<feature>,'showsel','on',...)
```

Getting Parameter and Variable Definitions

Use the command `mphgetexpressions` to get the expressions and the descriptions of parameters and variables. Use the command:

```
expr = mphgetexpressions(<node>)
```

where `<node>` is the node to get the expressions from. Nodes that can be used are `model.param`, `model.variable(<tag>)`, and `model.result.param`.

`expr` is an *N*x3 cell array where *N* is the number of expressions for this node.

Getting Selection Information

Use the function `mphgetselection` to retrieve the model selection information:

```
str = mphgetselection(model.selection(<seltag>))
```

where `<seltag>` is the tag a selection node defined in the model object. The output `str` is a MATLAB® structure with the following fields:

- **dimension**, the space dimension of the geometric entity selected,
- **geom**, the tag of the geometry node used in the selection,
- **entities**, the list of the entity indexes listed in the selection, and
- **isGlobal**, Boolean value to indicate if the selection is global or not.

Handling Errors and Warnings

In this section:

Errors and Warnings

COMSOL Multiphysics reports these types of problems:

- Errors, which prevents the program from completing a task, and
- Warnings, which are problems that do not prevent the completion of a task but that might affect the accuracy or other aspects of the model.

For both errors and warnings a message is stored in a separate node located just below the problematic model feature node.

In case of errors, a Java® Exception is thrown to MATLAB®, which also breaks the execution of the script.

Using MATLAB® Tools to Handle COMSOL® Exceptions

When running a model that returns an error in MATLAB®, the execution of the script is automatically stopped. You can use MATLAB tools to handle exceptions and prevent the script from breaking. Use the `try` and `catch` MATLAB statements to offer alternatives to a failed model.

In a loop, for example, use the `try` and `catch` statements to continue to the next iteration. For automatic geometry or mesh generation you can use it to set the model properties with alternative values that circumvent the problem.

Displaying Warnings and Errors in the Model

Use the command `mphshowerrors` to search in a given model object for warning or error nodes. To display the error and warning messages and their location in the model object enter the command:

```
mphshowerrors(model)
```

Alternatively `mphshowerrors` can also return the error and warning information in a MATLAB® variable:

```
str = mphshowerrors(model)
```

where `str` is an *N*x2 cell array, where *N* is the number of error and warning nodes that contain the model object. `str{i,1}`, which contains the location in the model of the *i*:th error or warning message, `str{i,2}` contains the message of the ith error or warning message, and `str{i,3}` contains a cell arrays of the model tree nodes that contain the error information. This last information makes it easier to extract error information for automated processing of error and warning conditions.

Improving Performance for Large Models

Memory management is key to successful modeling. In COMSOL Multiphysics the finite element model can store a large amount of data depending on the complexity of the model. Exchanging such a large amount of data between MATLAB® and the COMSOL *server* can be problematic in terms of memory management or execution time. This section discusses the model settings if you are experiencing memory problems or slowness of command execution.

- Allocating Memory
- Disabling Model Feature Update
- Disabling The Model History

Allocating Memory

COMSOL Multiphysics stores the data in Java®. If you are experiencing memory problems during meshing, postprocessing operations, or when exchanging data between the COMSOL *server* and MATLAB®, this can mean that the Java heap size is set with too low a value.

Increasing the memory allocated for the Java process necessarily decreases the memory available for the solver.

Either set The COMSOL Multiphysics Server Java Heap Size or Setting Manually the Memory in MATLAB.

THE COMSOL MULTIPHYSICS SERVER JAVA HEAP SIZE

The Java heap size settings for the COMSOL Multiphysics Server process are stored in the `comsolmphserver.ini` file. You can find this file in the `COMSOL56/Multiphysics/bin/<arch>` directory. `<arch>` correspond to the architecture of the machine where the COMSOL Multiphysics Server is running (win64, maci64, or glnxa64).

Edit the file with a text editor. The Java heap settings are defined as in the following lines:

```
-Xss4m
-Xms40m
-Xmx1024m
-XX:MaxPermSize=256m
```

The values are given in Mb; modify these value to satisfy the model requirements.

SETTING MANUALLY THE MEMORY IN MATLAB

To modify the Java heap size you need to edit the `java.opts` file available under the COMSOL with MATLAB startup directory. The `java.opts` file is stored by default with the following settings:

```
-Xss4m
-Xmx768m
-XX:MaxPermSize=256m
```

The values are given in Mb; modify these value to satisfy the model requirements.

To modify the MATLAB Java Heap size the `java.opts` file has to be stored at the MATLAB startup directory. This is the case when starting COMSOL with MATLAB.

If you are manually connecting MATLAB with a COMSOL Multiphysics Server, make sure you have the `java.opts` at the MATLAB startup directory.

Disabling Model Feature Update

Every time a setting in changed in a model, COMSOL Multiphysics automatically checks the settings for that particular feature and updates any other feature that may depend on the new setting. This operation ensures that the features are built with updated expressions and that any error messages appear as soon as possible after a COMSOL command has been run.

For models that contain a large amount of physics feature nodes, this update operation can take some time. For small models this is not an issue, but for larger models the checks can be time consuming. It can then help to deactivate the model feature update. To disable the feature model update, enter the command:

```
model.disableUpdates(true)
```

You have to enable the feature update again prior to computing the solution in order to make sure that COMSOL works on an updated model definition. Enabling the

feature update is also necessary before building the geometry or the mesh in case these are defined using expressions.

To enable the feature model update, enter the command:

```
model.disableUpdates(false)
```

Disabling The Model History

If you run a model in a loop you can experience a slowdown when the number of iterations increases. This happens only with a large amount of iterations. The increasing memory requirements for storing the model history explains this slowdown. You can see all the operations performed on the model when saving it as an M-file. If you run a model in a loop you do not need to store the model history because it contains the same operations as many times as you have iterations in the loop. The solution is to disable the history recording. To do this, enter the command:

```
model.hist.disable
```

When the model history is disabled you no longer see the commands used to set up the model when saving it as an M-file.

The functions `mphload` and `mphopen` automatically disables the model history when loading a model.

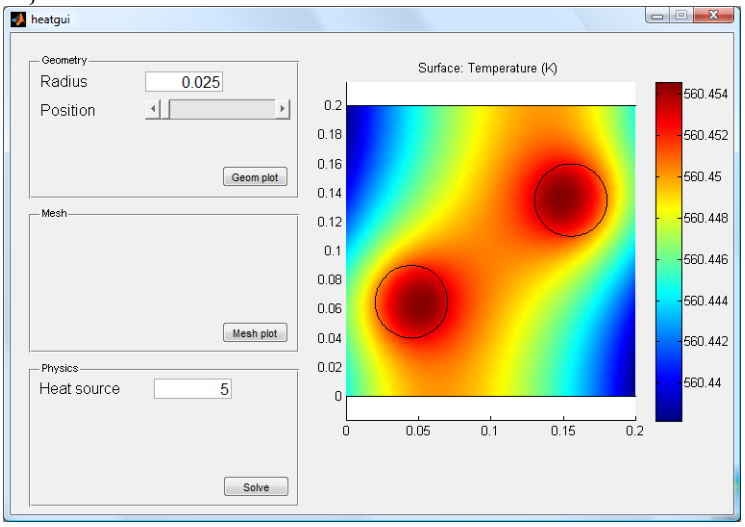
To activate the model history, enter the command:

```
model.hist.enable
```

Creating a Custom User Interface

You can use the MATLAB® `Guide` or App Designer functionality to create a GUI and connect the interface to a COMSOL Multiphysics model object. Each operation in the GUI sets the value of a MATLAB variable or calls a MATLAB command. You can call commands at the MATLAB prompt to set up a COMSOL model object or set MATLAB variables in the COMSOL model object.

The figure below illustrates a GUI made in MATLAB and linked to a COMSOL model object.



The simplified GUI only allows the user to compute a heat transfer problem on a given geometry. The user can only change the radius and the position of the bottom circle geometry. The heat source applied to the bottom circle is also defined by the user.

The button runs the building operation of the geometry and mesh. Another button runs the computation of the solution.

5

Calling External Functions

This section introduces you to the MATLAB® function callback from the COMSOL Desktop® and COMSOL Multiphysics® model object.

In this chapter:

Running External Function

When running the model containing a MATLAB function feature node, COMSOL Multiphysics automatically starts a MATLAB process that evaluates the function and returns the value to the COMSOL model.

You do not need to start COMSOL with MATLAB to call a MATLAB function from within the model; starting the COMSOL Desktop is sufficient. The MATLAB process starts automatically to evaluate the function.

Linux

On Linux operating systems, specify the MATLAB root directory path `MLROOT` and load the gcc library when starting the COMSOL Desktop: `comsol -mlroot MLROOT -forcegcc`.

Allowing External MATLAB Functions

To run MATLAB functions you need to allow external processes in the security preferences.

In the COMSOL Desktop go to the **Preferences** and select **Security**, locate the **General** section. To enable permanently external MATLAB function, in the list **Allow external MATLAB® functions** select **Yes**. To enable external MATLAB function only once set the list **Allow external MATLAB® functions** to **Ask**. Using the later option, you will be asked to enable external functions when the function is run.

In the COMSOL Server™, you need to connect as administrator then go to **Administration>Preferences**, in the **Security** section select **Allow external MATLAB® functions**. Then click **Save**.

ALLOWING EXTERNAL PROCESSES IN THE COMMAND LINE

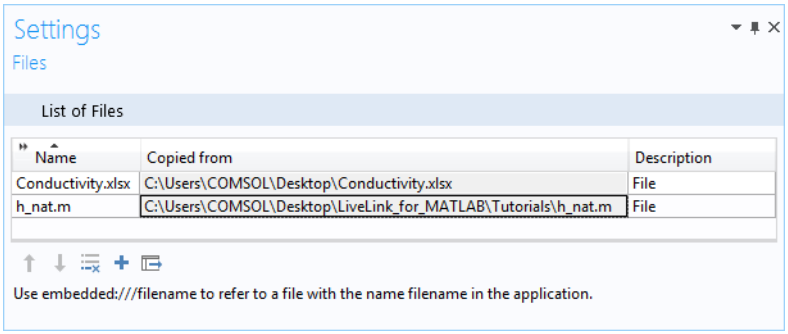
Add the flag `-allowexternalmatlab on` to the COMSOL startup command to enable external processes.

Disabling MATLAB® Splash Screen at Startup

To disable the MATLAB splash screen that pops up when the MATLAB engine is started you need to create the environment variable `COMSOL_MATLAB_INIT` before starting COMSOL and set this variable with the value `"matlab -nosplash"`.

Running a MATLAB® Function in Applications

To run an application from the COMSOL Server™ that uses an external MATLAB function, it is recommended to embed the function M-file in the application. The Application Builder offers the possibility to upload file that can then be accessible on the server. Under the **Libraries** node, select **Files**, and in the **Files** node's **Settings** window, click the **Add File to Library** button ( ) to add the M-file to the library..



	Applications that use external MATLAB function are not supported using the COMSOL Client. It is only possible to use such applications using a browser.

	After embedding an M-file used by an existing MATLAB function, the embedded M-file will only be used for new solutions. So it is recommended to update any solutions that are saved in the application, if they depend on the embedded M-file.

The MATLAB® Function Feature Node

MATLAB® functions are global in scope and you can use them in a model to define model settings such as

- Parameters
- Geometry settings
- Mesh settings
- Material properties
- Physics settings (domain conditions, boundary conditions, etc.)

Material properties and physics settings are evaluated while the model is solved whereas the features can be used while the model is constructed.

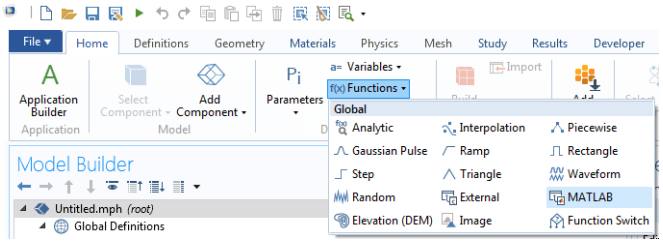
Defining a MATLAB® Function in the COMSOL® Model

These topics are described for the MATLAB® function:

ADDING THE MATLAB FUNCTION NODE

To evaluate a MATLAB function from in the COMSOL Multiphysics model you need to add a MATLAB node in the model object where the function name, the list of the arguments, and, if required, the function derivatives, are defined.

To add a MATLAB function node, on the **Home** toolbar, click **Functions** and select **Global>MATLAB**( ).

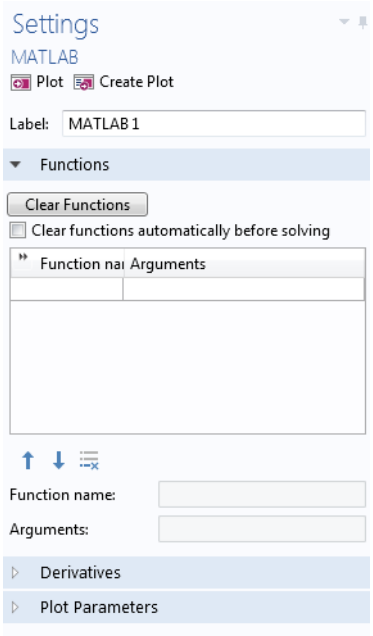


The **Settings** window of the **MATLAB** node has these sections:

- **Functions**, where you declare the name of the MATLAB functions and their arguments.
- **Derivatives**, where you define the derivative of the MATLAB functions with respect to all function arguments.
- **Plot Parameters**, where you can define the limit of the arguments value in order to display the function in the COMSOL Desktop Graphics window.

DEFINING THE MATLAB FUNCTION

This figure illustrates the **Settings** window for **MATLAB**:



Under **Functions**, you define the function name and the list of the function arguments.

In the table columns and rows, enter the **Function name** and the associated function **Arguments**. The table supports multiple function definitions. You can define several functions in the same table or add several **MATLAB** nodes, as you prefer.

About Requirements for Functions and How to Test Them

Any function that you want to call using a **MATLAB** node must fulfill the following requirements:

- It can take any number of inputs as vectors and must return a single output vector.
- The input vectors can be of arbitrary size, but in a single call the inputs vectors will all have the same length. The returned vector must have exactly the same length as the input vectors.

For example, functions such as `+` and `-` (plus and minus) work well on vector inputs, but matrix multiplication (`*`, `mtimes`) and matrix power (`^`, `mpower`) do not. Instead, use the elementwise array operators `.*` and `.^`. See also Function Input/Output Considerations.

It is good practice to test your own functions and any MALTLAB functions that you want to call from COMSOL Multiphysics by running them on the MATLAB command line using vectors with suitable values as inputs. For example, using `besselj`, a Bessel function of the first kind:

```
input = (1:10)'
size(input)
out = besselj(input, input)
size(out)
```

Here there are no errors, and the size of the input and output is the same.

As another example, test the `corrcoef` function for computing correlation coefficients:

```
input = (1:10)'
size(input)
out = corrcoef(input, input)
size(out)
```

There are no errors when calling `corrcoef` using vector inputs, but the result does not have the same size as the input and hence a call to `corrcoef` in this way will not work.

PLOTTING THE FUNCTION

Click the **Plot** button () to display a plot of the function.

Click the **Create Plot** button () to create a plot group under the **Results** node.

To plot the function you first need to define limits for the arguments. Expand the **Plot Parameters** section and enter the desired value in the **Lower limit** and **Upper limit** columns. In the **Plot Parameters** table the number of rows correspond to the number of input arguments of the function. The first input argument corresponds to the top row.

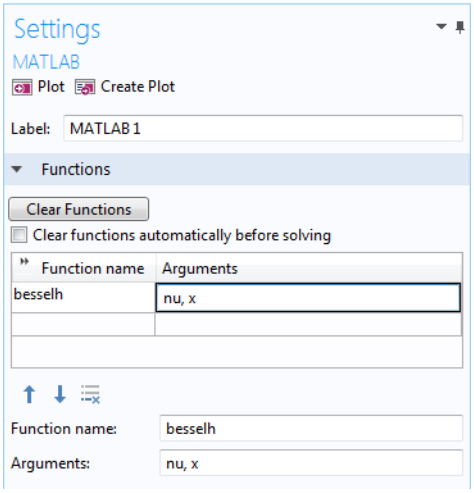
In case there are several functions declared in the **Functions** table, only the function that has the same number of input arguments as the number of filled in rows in the **Plot Parameters** table is plotted.

If several functions have the same number of input arguments, the first function in the table (from top to bottom) is plotted. Use the **Move up** (↑) and **Move down** (↓) buttons to change the order of functions in the table.

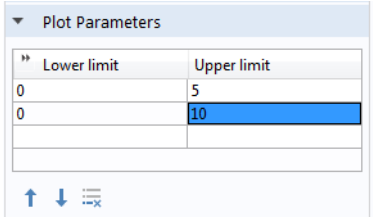
EXAMPLE: DEFINE THE HANKEL FUNCTION

Assume that you want to use MATLAB's Bessel function of the third kind (Hankel function) in a COMSOL model. Add a MATLAB function node, then define the following settings:

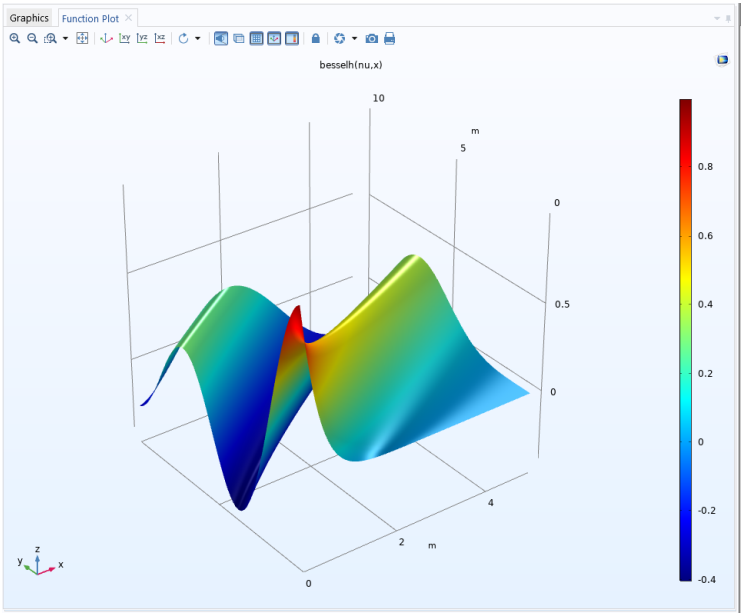
FUNCTION NAME	ARGUMENTS
`besselh`	`nu, x`



To plot the function you need first to define the lower and upper limits for both `nu` and `x`. In the **Plot Parameters** table set the first row (which corresponds to the first argument `nu`) of the **Lower limit** column to 0 and the **Upper limit** column to 5 and set the second row (corresponding of `x`) of the **Lower limit** column to 0 and the **Upper limit** column to 10:



Click the **Plot** button ( ) to get this plot:



Setting the Function Directory Path in MATLAB®

To be able to run a model that use an external MATLAB® function, the path directory of the function has to be set in MATLAB before it is called by COMSOL Multiphysics to evaluate the function.

To proceed you have these options to set the directory path in MATLAB:

- the model MPH-file in the same directory as for the M-functions;
- Set the system environment variable COMSOL_MATLAB_PATH with the M-functions directory path; or
- Use the **Set Path** window to specify the MATLAB search path. To open the window type `pathtool` at the MATLAB prompt or in the MATLAB desktop go the **Home** toolbar, **Environment** group.

Adding a MATLAB® Function with the COMSOL® API Syntax

To add a MATLAB® feature node to the COMSOL Multiphysics model using the COMSOL API, enter the command:

```
model.func.create(<ftag>, 'MATLAB')
```

Define the function name and function arguments with the command:

```
model.func(<ftag>).setIndex('funcs', <function_name>, 0, 0)
model.func(<ftag>).setIndex('funcs', <arglist>, 0, 1)
```

where *<function_name>* is a string set with the function name and *<arglist>* is a string that defines the list of the input arguments.

Function Input/Output Considerations

The functions called from COMSOL Multiphysics must support vector arguments of any length. COMSOL calls a MATLAB® function using vector arguments to reduce the number of expensive calls from COMSOL to MATLAB. All common MATLAB functions such as `sin`, `abs`, and other mathematical functions support vector arguments.

When you write your own functions, remember that the input arguments are vectors. The output must have the same size as the input. All arguments and results must be double-precision vectors real or complex valued.

Consider the following example function where the coefficient c depends on the x coordinate:

```
function c = func1(x)
if x > 0.6
  c = x/1.6;
else
  c = x^2+0.3;
end
```

This function looks good at first but it does not work in COMSOL Multiphysics because the input x is a vector:

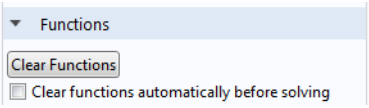
- Element-by-element multiplication, division, and power must be used—that is, the operators `.*`, `./`, and `.^`. Replace expressions such as `x/1.6` and `x^2+0.3` with `x./1.6` and `x.^2+0.3`, respectively.
- The comparison $x > 0.6$ returns a matrix with ones (true) for the entries where the expression holds true and zeros (false) where it is false. The function evaluates the conditional statement if, and only if, all the entries are true (1).

You can replace the `if` statement with a single assignment to the indices retrieved from the $x > 0.6$ operation and another assignment to the indices where $x \leq 0.6$. The function could then look like this:

```
function c = func2(x)
c = (x./1.6).*(x>0.6) + (x.^2+0.3).*(x<=0.6);
```

Updating Functions

If the function M-file is modified using a text editor, click **Clear Functions** to ensure that the functions' modifications are updated in the COMSOL Multiphysics model.



An alternative is to select the **Clear functions automatically before solving** check box.

Defining Function Derivatives

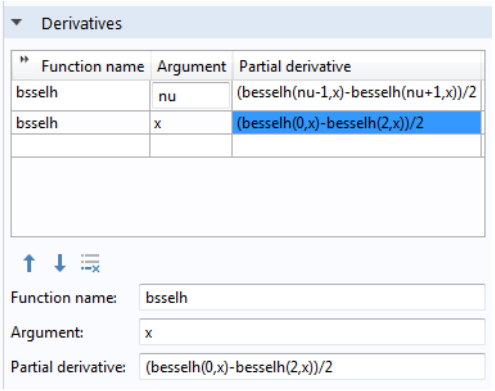
Automatic differentiation is not supported with MATLAB® functions. In case the MATLAB function has Jacobian contributions, its derivatives with respect to the function input arguments need to be defined. By default COMSOL Multiphysics assumes the derivatives to be null.

Expand the **Derivatives** section to define the derivatives of the function with respect to the function arguments. In the table define the derivative for each function argument. In the **Function** column enter the function name, in the **Argument** column enter the argument. Finally in the **Function derivative** column enter the expression for the corresponding derivative.

The function derivatives can also be defined by additional MATLAB functions.

The section, Example: Define the Hankel Function, defined the function derivative by entering the following settings in the table:

FUNCTION	ARGUMENT	FUNCTION DERIVATIVE
`besselh`	`nu`	`(besselh(nu-1,x)-besselh(nu+1,x))/2`
`besselh`	`x`	`(besselh(0,x)-besselh(2,x))/2`


Derivatives
Function name
Argument
Partial derivative
bsselh
nu
(besselh(nu-1,x)-besselh(nu+1,x))/2
bsselh
x
(besselh(0,x)-besselh(2,x))/2
Function name:
bsselh
Argument:
x
Partial derivative:
(besselh(0,x)-besselh(2,x))/2

6

Command Reference

The main reference for the syntax of the commands available with LiveLink™ *for* MATLAB® is the *COMSOL Multiphysics Programming Reference Manual*. This section documents additional interface functions that come with the product.

In this chapter:

Summary of Commands

colortable
mphaddplotdata
mphapplicationlibraries
mphcd
mphcomponentinfo
mphdoc
mpheval
mphevalglobalmatrix
mphevalpoint
mphevalpointmatrix
mphevaluate
mphgeom
mphgeominfo
mphgetadj
mphgetcoords
mphgetexpressions
mphgetproperties
mphgetselection
mphgetu
mphglobal
mphimage2geom
mphinputmatrix
mphint2
mphinterp
mphinterpolationfile
mphlaunch
mphload
mphmatrix
mphmax
mphmean
mphmeasure
mphmesh
mphmeshstats
mphmin
mphmodel
mphnavigator
mphopen
mphparticle
mphplot
mphquad2tri
mphray
mphreadstl
mphreduction
mphreport
mphsave
mphsearch
mphselectbox
mphselectcoords
mphshowerrors
mphsolinfo
mphsolutioninfo
mphstart
mphstate
mphsurf
mphtable
mphtags
mphthumbnail
mphversion
mphviewselection
mphwritestl
mphxmeshinfo

Commands Grouped by Function

INTERFACE FUNCTIONS

FUNCTION	PURPOSE
mphcd	Change the directory to the directory of the model.
mphdoc	Open help window for a certain topic.
mphlaunch	Launch a COMSOL Multiphysics Client, connect it to the server and load a model.
mphload	Load a COMSOL model MPH-file.
mphopen	GUI for opening recent model files.
mphsave	Save a COMSOL model.
mphstart	Connect MATLAB to a COMSOL *server*.
mphthumbnail	Set or get model thumbnail.
mphversion	Return the version number of COMSOL Multiphysics

GEOMETRY FUNCTIONS

FUNCTION	PURPOSE
mphgeom	Plot a geometry in a MATLAB figure.
mphgeominfo	Get geometry information.
mphimage2geom	Convert image data to geometry.
mphmeasure	Measure entities in a geometry.
mphviewselection	Display a geometric entity selection in a MATLAB figure.

MESH FUNCTIONS

FUNCTION	PURPOSE
mphmesh	Plot a mesh in a MATLAB figure.
mphmeshstats	Return mesh statistics and mesh data information.

UTILITY FUNCTIONS

FUNCTION	PURPOSE
mphevaluate	Evaluate Parameters expressions in model.
mphgetadj	Return geometric entity indices adjacent to each other.

FUNCTION	PURPOSE
mphgetcoords	Return point coordinates of geometry entities.
mphgetu	Return solution vectors.
mphinputmatrix	Add matrix system for a linear solver.
mphinterpolationfile	Save data in file readable by the Interpolation feature.
mphmatrix	Get model matrices.
mphquad2tri	Convert plot data quad mesh into simplex mesh.
mphreadstl	Read an STL file and returns the data as a struct.
mphreduction	Return reduced-order state-space matrices for a model.
mphselectbox	Select a geometric entity using a rubber band/box.
mphselectcoords	Select a geometric entity using point coordinates.
mphsolinfo	Get information about a solution object.
mphsolutioninfo	Get information about solution objects and datasets containing given parameters.
mphstate	Get state-space matrices for dynamic systems.
mphsurf	Create plot data structure from surf data.
mphwritestl	Export plot data as an STL file.
mphxmeshinfo	Extract information about the extended mesh.

POSTPROCESSING FUNCTIONS

FUNCTION	PURPOSE
mphaddplotdata	Add a data plot to a model.
mpheval	Evaluate expressions on node points.
mphevalglobalmatrix	Evaluate global matrix variables.
mphevalpoint	Evaluate expressions at geometry vertices.
mphevalpointmatrix	Evaluate matrix quantities at points in the geometry
mphglobal	Evaluate global quantities.
mphint2	Perform integration of expressions.
mphinterp	Evaluate expressions in arbitrary points or datasets.
mphmax	Perform maximum of expressions.
mphmean	Perform mean of expressions.
mphmin	Perform minimum of expressions.
mphparticle	Evaluate expressions on particle trajectories.
mphplot	Render a plot group in a figure window.
mphray	Evaluate expressions on particle and ray trajectories.
mphreport	Generate report to model or write report.
mphtable	Get table data.
mphray	Evaluate expressions on ray trajectories.

MODEL INFORMATION AND NAVIGATION

FUNCTION	PURPOSE
mphapplicationlibraries	GUI for viewing the product Application Libraries.
mphcomponentinfo	Get information about a component.
mphgetexpressions	Get the model variables and parameters.
mphgetproperties	Get properties from a model node.
mphgetselection	Get information about a selection node.
mphmodel	Return tags for the nodes and subnodes in the COMSOL model object.
mphnavigator	GUI for viewing the COMSOL model object.
mphsearch	GUI for searching expressions in the COMSOL model object.

FUNCTION	PURPOSE
mphshowerrors	Show messages in error and warning nodes in the COMSOL model object.
mphtags	Get tags and names for nodes in a COMSOL model.
mphthumbnail	Get and set model thumbnail images

colortable

Return a MATLAB® colormap for a COMSOL Multiphysics color table.

SYNTAX

```
map = colortable(name)
colortable(name)
list = colortable
```

DESCRIPTION

`map = colortable(name)` returns the color table (of 256 colors) for `name`, where `name` can be one of the list below.

`colortable(name)` creates a plot of the colortable `name`.

`list = colortable` returns a list of the available colortables.

The available colortable are listed below:

- `AuroraAustralis`, `AuroraAustralisDark`, `AuroraBorealis`, and `JupiterAuroraBorealis` - color tables resemble the colors in the aurora australis (southern light), aurora borealis (northern light), and Jupiter's aurora, respectively. The AuroraAustralis color table spans from white through green and indigo to blue. The AuroraAustralisDark color is similar to AuroraAustralis but does not start with absolute white so that the end value's color is different from a white background. The AuroraBorealis color table also spans from white through green and indigo to blue but with a larger indigo portion. The JupiterAuroraBorealis color table spans from black through blue to white.
- `Twilight` - color table uses colors associated with twilight (the illumination of the Earth's lower atmosphere when the Sun is not directly visible), spanning colors from pink through white to blue.
- `Cividis` - color table uses yellow and blue colors in a color table that is suited for normal vision, a deuteranomaly, or red-green colorblindness. It was created by Jamie Nuñez, Ryan Renslow, and Chris Anderton at the Biological Sciences

Division of the Pacific Northwest National Laboratory (located in Richland, Washington state, United States).

- `Cyclic` and `CyclicClassic` - color tables are useful for displaying periodic functions because they have a sharp color gradient — it varies the hue component of the hue-saturation-value (HSV) color model, keeping the saturation value constant (equal to 1). The colors begin with red, then pass through yellow, green, cyan, blue, magenta, and finally return to red.
- `Dipole` - color table has been developed for plots where the majority of the solution is close to zero (or centered around some meaningful reference point) and where, in some places, positive and negative excitations occur. Typical cases are the electric potential distribution around positively and negatively charged objects with the field relaxing to zero at infinity and the pressure distribution of an acoustic wave propagating in a large open space. For those cases, the scale tends to white in areas of relative inactivity, allowing for a good contrast with titles, legends, dataset edges, and other plot elements. This reasoning is similar to the one used for the Prism color scale. The main difference is that Dipole is symmetric (or "diverging"), making it suitable for positive and negative scalar fields, whereas Prism is more suitable for vector field norms (which are positive only).
- `DipoleDark` - similar but uses slightly darker colors.

  You can use a combination of Dipole color tables to visualize streamlines, contour lines, and arrows using DipoleDark on top of a surface plot using Dipole.
- `Disco` and `DiscoClassic` - color tables span from red through magenta and cyan to blue. DiscoDark and DiscoLight are similar but use darker and lighter colors, respectively.

  You can use a combination of Disco color tables to visualize streamlines or contour lines using DiscoLight on top of a surface plot using Disco. The same way, DiscoDark can be used to draw on top of Disco.
- `Gaia` - color table spans colors that make it suitable for visualizing plots related to topography and bathymetry.
- `GaiaLight` - similar but use lighter colors.

  You can use a combination of Gaia color tables to visualize streamlines or contour lines using GaiaLight on top of a surface plot using Gaia.
- `GrayBody` - color table is based on the Planckian locus for blackbody radiation. It is useful within the context of metal processing, for example.

- `GrayBodyLight` - is similar but use lighter colors.

  You can use a combination of GrayBody color tables to visualize streamlines or contour lines using GrayBodyLight on top of a surface plot using GrayBody.

- `GrayScale` - color table uses the linear gray scale from black to white — the easiest palette to understand and order.

- `GrayPrint` - color table varies linearly from dark gray (RGB: 0.95, 0.95, 0.95) to light gray (RGB: 0.05, 0.05, 0.05). Choose this color table to overcome two difficulties that the GrayScale color table has when used for printing on paper — it gives the impression of being dominated by dark colors, and white is indistinguishable from the background.

  Gray scale plots are often easier to use for publication. People can also better perceive structural detail in a gray scale than with color.

- `Inferno`, `Magma`, and `Plasma` - color tables use a general bluish to reddish to yellowish color sequence, which is relatively friendly to common forms of color vision deficiencies. They are designed so that they are analytically perceptually uniform, both in regular form and when converted to black-and-white images. They were created by Stéfan van der Walt and Nathaniel Smith.

- `Prism` - slightly similar to the Rainbow color table but includes a white tip and is brighter. It has been developed for plots where the majority of the solution is close to zero (typically at the outer perimeter of the modeling domain) and where, in some places, singularities or "hot spots" occur. This is especially true for electric and magnetic field norms in electromagnetic models, but may also occur for stress and strain norms in structural mechanics models, for example. For those cases, the scale tends to white in areas of relative inactivity, allowing for a good contrast with titles, legends, dataset edges, and other plot elements. This is the default color table for plots when using structural mechanics physics interfaces.

- `PrismDark` - similar but uses slightly darker colors. It can be used, in combination with Prism, to add arrows or streamlines on top of a plane, for instance. Close to the hot spots the contrast between the two color scales increases, and in the wake regions the contrast will fade.

  You can use a combination of Prism color tables to visualize streamlines, contour lines, and arrows using PrismDark on top of a surface plot using Prism.

- `Rainbow` - the default for most plots that support color tables. The color ordering corresponds to the wavelengths of the visible part of the electromagnetic spectrum. It starts at the small-wavelength end with dark blue. The colors range through shades of blue, cyan, green, yellow, and red. The disadvantage of this color table is

that people with color vision deficiencies (affecting up to 10% of technical audiences) cannot see distinctions between reds and greens.

- `RainbowClassic` - the rainbow color table used in versions of COMSOL Multiphysics earlier than version 6.0. Compared to RainbowClassic, the current Rainbow color table is less saturated, more uniform, more smooth, and more balanced.
- `RainbowDark` and `RainbowLight` - similar to Rainbow but use darker and lighter colors, respectively.

  You can use a combination of Rainbow color tables to visualize streamlines or contour lines using RainbowLight on top of a surface plot using Rainbow. The same way, RainbowDark can be used to draw on top of Rainbow.
- `Spectrum` and `SpectrumClassic` are similar to the Rainbow color tables but include violet at the small-wavelength end of the visible spectrum. They also include richer shades of green to more closely replicate the human perception of visible light.
- `SpectrumLight` - similar but use lighter colors. You can use them with the Ray Optics Module, for example, to accurately visualize polychromatic light.

  You can use a combination of Spectrum color tables to visualize streamlines or contour lines using SpectrumLight on top of a surface plot using Spectrum.
- `Thermal` and `ThermalClassic` - differ in that the Thermal color table uses equal distances from dark red to orange, yellow, and white, which means that the region with the lowest values is red instead of black as in the ThermalClassic color table. The colors correspond to the colors iron takes as it heats up.
- `ThermalLight` and `ThermalLightClassic` - similar but use lighter colors.
- `ThermalDark` - similar but uses darker colors.

  You can use a combination of Thermal color tables to visualize streamlines or contour lines using ThermalLight on top of a surface plot using Thermal. The same way, ThermalDark can be used to draw on top of Thermal.
- `ThermalWave` - designed for wave phenomena with a thermal character. It is calibrated to have a 100% symmetric luminance.
- `ThermalWaveDark` - similar to ThermalWave but with slightly darker colors.

  You can use a combination of ThermalWave color tables to visualize streamlines, contour lines, and arrows using ThermalWaveDark on top of a surface plot using ThermalWave.

- `Traffic` and `TrafficClassic` - color tables span from green through yellow to red.
- `TrafficLight` and `TrafficLightClassic` - similar but use lighter colors.

  You can use a combination of Traffic color tables to visualize streamlines or contour lines using TrafficLight on top of a surface plot using Traffic.
- `Viridis` - color table uses a general bluish to greenish to yellowish color sequence, which is relatively friendly to common forms of color vision deficiencies. It is designed so that it is analytically perceptually uniform, both in regular form and when converted to a black-and-white image. It was created by Stéfan van der Walt and Nathaniel Smith.
- `Wave` and `WaveClassic` - color tables are useful for data that naturally has positive and negative attributes in addition to a magnitude. As an example of a double-ended color scheme, it ranges linearly from blue to light gray, and then linearly from white to red. When the range of the visualized quantity is symmetric around zero, the color red or blue indicates whether the value is positive or negative, and the saturation indicates the magnitude.

  People with color vision deficiencies can interpret the Wave color table because it does not use red-green-gray distinctions, making it efficient for 99.98% of the population.
- `WaveLight` and `WaveLightClassic` - similar and range linearly from a lighter blue to a lighter red.

  You can use a combination of Wave color tables to visualize streamlines or contour lines using WaveLight on top of a surface plot using Wave.

Note: The classic versions of some color tables correspond to that same color table in COMSOL Multiphysics versions earlier than version 6.0 (for example, `RainbowClassic` correspond to the `Rainbow` color table in versions before 6.0. The new versions of these color tables are typically less saturated, more uniform, more smooth, and more balanced.

EXAMPLE

Create a rainbow color map

```
map = colortable('Rainbow');
colortable('Prism')
```

SYNTAX

```
mphaddplotdata(model,'type',type,'data',data,...)
```

DESCRIPTION

`mphaddplotdata(model,'type',type,'data',data,...)` create a new plot using the data `data` as a plottype `type`.

The function `mphaddplotdata` accepts the following property/value pairs:

TABLE 6-1: PROPERTY/VALUE PAIRS FOR THE MPHADDPLOTDATA COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`arrowtype`	`arrow | arrowhead | cone`	`arrow`	Arrow type
`arrowbase`	`head | tail | center`	`tail`	Arrow base
`clearplot`	`off | on`	`off`	Clear plot group
`color`	`vector | string | index | matrix`		Color specification
`colortable`	`string | cell array`		Color table name
`data`	`matrix`		Coordinates data
`edges`	`off| on`	`on`	Plot dataset edges
`elementdata`	`matrix`		Element data (low level)
`latex`	off | on	`off`	Use LaTeX markup
`normaldata`	`matrix`		Data for normals (low level)
`plotgroup`	string		Plot group tag
`radius`	vector		Radius for tubes
`type`	`arrow | line | surface | annotation | tube | point`		Plot type
`vector`	matrix		Direction vector

SEE ALSO

`mphplot`

mphapplicationlibraries

Graphical user interface (GUI) for viewing the Application Libraries.

SYNTAX

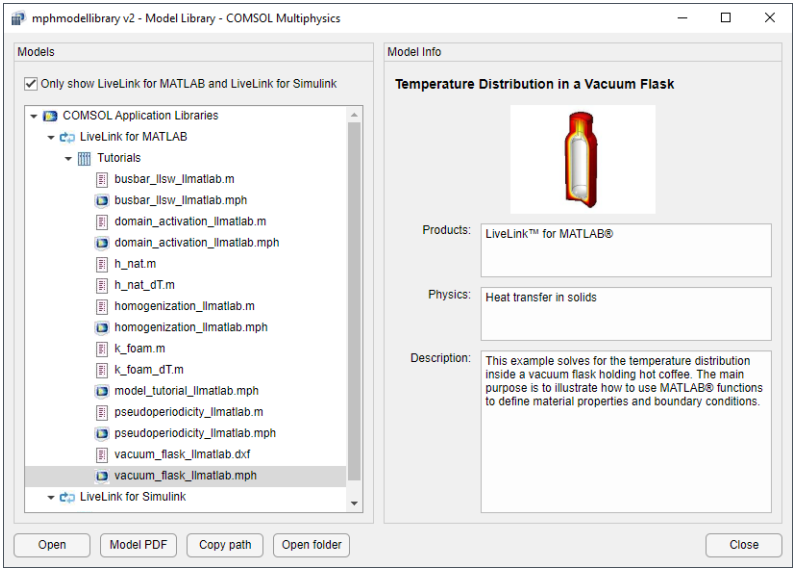
```
mphapplicationlibraries
```

DESCRIPTION

`mphapplicationlibraries` starts a GUI to visualize and access the example model available in the COMSOL Application Libraries. The model MPH-file can be loaded in MATLAB® and the model documentation PDF-file is accessible directly.

Models that are specific to LiveLink™ *for* MATLAB® also contains the script M-file.

Models that are specific to LiveLink™ *for* Simulink® also contains the simulation SLX-file and opens directly in Simulink.



mphcd

Change directory to the directory of the model.

SYNTAX

```
mphcd(model)
```

DESCRIPTION

`mphcd(model)` changes the current directory in MATLAB® to the directory where the `model` was last saved.

SEE ALSO

`mphload`, `mphsave`

mphcomponentinfo

Get information about a component.

SYNTAX

```
info = mphcomponent(model, comptag)
```

DESCRIPTION

`info = mphcomponent(model, comptag)` returns information about the component `comptag`.

If the model contains only one component, the tag `comptag` is optional.

The output structure `info` contains the following fields:

TABLE 6-2: FIELDS IN THE INFO STRUCTURE

FIELD	DESCRIPTION
`tag`	Component tag of the component
`geom`	Geometries in the component
`mesh`	Meshes in the component
`geometricmodel`	Geometric model defining the physics
`scope`	Full scope
`basesystem`	Tag of the unit system
`sorder`	Geometry shape function order
`geometrycoord`	Geometry frame coordinate variables
`materialcoord`	Material frame coordinate variables
`meshcoord`	Mesh frame coordinate variables
`spatialcoord`	Spatial frame coordinate variables

SEE ALSO

`mphnavigator`

mphdoc

Open help window for a certain topic.

SYNTAX

```
mphdoc
mphdoc(node)
mphdoc(node,fname)
mphdoc api
```

DESCRIPTION

`mphdoc` opens the COMSOL documentation Help Desk.

`mphdoc(node)` opens the help window for the entry on `node`.

`mphdoc(node,fname)` opens the help window for the entry on `node` with the feature `fname`.

`mphdoc api` opens a window with the JavaDoc help for the COMSOL API.

EXAMPLE

Create a model a model object:

```
model = ModelUtil.create('Model');
```

Get the documentation for the mesh node;

```
mphdoc(model.mesh)
```

Get the documentation of the geometry feature Rectangle:

```
mphdoc(model.geom,'Rectangle')
```

SEE ALSO

`mphapplicationlibraries`

mpheval

Evaluate expressions on node points.

SYNTAX

```
pd = mpheval(model,{e1,...,en},...)
```

DESCRIPTION

`pd = mpheval(model,{e1,...,en},...)` returns the post data `pd` for the expressions `e1,...,en`.

The output value `pd` is a structure with fields `expr`, `p`, `t`, `ve`, `unit` and fields for data values.

- The field `expr` contains the expression name evaluated.
- For each expression `e1,...,en` a field with the name `d1,... dn` is added with the numerical values. The columns in the data value fields correspond to node point coordinates in columns in `p`. The data contains only the real part of complex-valued expressions.
- 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` representing a simplex.
- The field `ve` contains indices to mesh elements for each node point.
- The field `unit` contains the list of the unit for each expression.

The function `mpheval` accepts the following property/value pairs:

TABLE 6-3: PROPERTY/VALUE PAIRS FOR THE MPHEVAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`complexfun`	`off | on`	`on`	Use complex-valued functions with real input
`complexout`	`off | on`	`on`	Return complex values
`dataonly`	`off | on`	`off`	Only return expressions value
`dataset`	`String`		Dataset tag
`edim`	`point | edge | boundary | domain | 0 | 1 | 2 | 3`	`Geometry space dimension`	Evaluate on elements with this space dimension
`matherr`	`off | on`	`off`	Error for undefined operations or expressions
`outersolnum`	`Positive integer | all | end`	`1`	Solution number for parametric sweep
`pattern`	`lagrange | gauss`	`lagrange`	Specifies if evaluation takes place in Lagrange points or in Gauss points

TABLE 6-3: PROPERTY/VALUE PAIRS FOR THE MPHEVAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`phase`	Scalar	`0`	Phase angle in degrees
`recover`	`off` | `ppr` | `pprint`	`off`	Accurate derivative recovery
`refine`	Integer	`1`	Refinement of elements for evaluation points
`selection`	Integer vector | string | `all`	`all`	Set selection tag or entity number
`smooth`	`internal` | `none` | `everywhere`	`internal`	Smoothing setting
`solnum`	Integer vector | `all` | `end`	`all`	Solutions for evaluation
`t`	Double array		Times for evaluation
`unit`	String | Cell array		Unit to use for the evaluation

The property `Dataset` controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution datasets.

The property `Edim` decides which elements to evaluate on. Evaluation takes place only on elements with space dimension `Edim`. If not specified, `Edim` equal to the space dimension of the geometry is used. The setting is specified as one of the following strings `'point'`, `'edge'`, `'boundary'` or `'domain'`. In previous versions it was only possible to specify `Edim` as a number. For example, in a 3D model, if evaluation is done on edges (1D elements), `Edim` is 1. Similarly, for boundary evaluation (2D elements), `Edim` is 2, and for domain evaluation (3D elements), `Edim` is 3 (default in 3D).

Use `Recover` to recover fields using polynomial-preserving recovery. This techniques recover fields with derivatives such as stresses or fluxes with a higher theoretical convergence than smoothing. Recovery is expensive so it is turned off by default. The value `pprint` means that recovery is performed inside domains. The value `ppr` means that recovery is also applied on all domain boundaries.

The property `Refine` constructs evaluation points by making a regular refinements of each element. Each mesh edge is divided into `Refine` equal parts.

The property `Smooth` controls if the post data is forced to be continuous on element edges. When `Smooth` is set to internal, only elements not on interior boundaries are made continuous.

The property `Solnum` is used to select the solution to plot when a parametric, eigenvalue, or time-dependent solver has been used to solve the problem.

The property `Outersolnum` is used to select the solution to plot when a parametric sweep has been used in the study.

When the property `Phase` is used, the solution vector is multiplied with `exp(i*phase)` before evaluating the expression.

The expressions `e1,...,en` are evaluated for one or several solutions. Each solution generates an additional row in the data fields of the post data output structure. The properties `Solnum` and `t` control which solutions are used for the evaluations. The `Solnum` property is available when the dataset has multiple solutions — for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided, solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

EXAMPLE

Evaluate the temperature at node points:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

dat = mpheval(model,'T');
```

Evaluate both the total heat flux magnitude and the temperature:

```
data = mpheval(model,{'ht.tfluxMag', 'T'});
```

Evaluate the temperature and return the data only:

```
data = mpheval(model,'T','dataonly','on');
```

Evaluate the temperature at the node points in domain 2:

```
data = mpheval(model,'T','selection',2);
```

Evaluate the temperature at the node points on boundary 7:

```
data = mpheval(model,'T','selection',7,'edim','boundary');
```

Evaluate the temperature at second order Lagrange points:

```
data = mpheval(model,'T','refine',2);
```

Evaluate the temperature at the Gauss points:

```
data = mpheval(model,'T','pattern','gauss');
```

Evaluate the temperature at every time step computed with power set to 30:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;
```

```
data = mpheval(model,'T','dataset','dset2');
```

Evaluate the temperature at the fifth time step:

```
data = mpheval(model,'T','dataset','dset2','solnum',5);
```

Evaluate the temperature at 10.5 sec and 15.2 sec:

```
data = mpheval(model,'T','dataset','dset2','t',[10.5,15.2]);
```

Evaluate the temperature at every time step computed with power set to 90:

```
data = mpheval(model,'T','dataset','dset2','outersolnum',3);
```

SEE ALSO

`mphevalglobalmatrix`, `mphevalpoint`, `mphevalpointmatrix`, `mphglobal`, `mphint2`, `mphinterp`, `mphparticle`, `mphray`

mphevalglobalmatrix

Evaluate global matrix variables.

SYNTAX

```
M = mphevalglobalmatrix(model,expr,...)
```

DESCRIPTION

`M = mphevalglobalmatrix(model,expr,...)` evaluates the global matrix of the variable `expr` and returns the full matrix `M`.

The function `mphevalglobalmatrix` accepts the following property/value pairs:

TABLE 6-4: PROPERTY/VALUE PAIRS FOR THE MPHEVALGLOBALMATRIX COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`dataset`	String		Dataset tag
`dataseries`	`none` | `average` | `sum`	`none`	Data series operation
`outerdataseries`	`none` | `average` | `sum`	`none`	Outer data series operation
`outersolnum`	Positive integer | `end` | `all`	1	Solution number for parametric sweep
`solnum`	`Integer vector` | `end` | `all`	`all`	Solution for evaluation
`t`	`Double array`		Time for evaluation
`trans`	`none` | `inverse` | maxwellmutual | mutualmaxwell | `sy` | `sz` | `ys` | `yz` | `zs` | `zy`	`none`	The transformation to apply to the matrix
`y0`	`Double array`	Taken from the physics interfaces	If trans is sy or ys: The characteristic admittance
`z0`	`Double array`	Taken from the physics interfaces	If trans is sz or zs: The characteristic impedance

Note: S-parameters evaluation requires the RF module.

EXAMPLE

Load `lossy_circulator_3d.mph` from the RF Module's Applications Libraries:

```
model = mphopen('lossy_circulator_3d.mph');
```

Evaluate the S-parameters matrix using the solution dataset `dset4`:

```
M = mphevalglobalmatrix(model,'emw.SdB','dataset','dset4');
```

SEE ALSO

mpheval, mphevalpoint, mphevalpointmatrix, mphglobal, mphint2, mphinterp, mphparticle, mphray

mphevalpoint

Evaluate expressions at geometry vertices.

SYNTAX

```
[v1,...,vn] = mphevalpoint(model,{e1,...,en},...)
[v1,...,vn,unit] = mphevalpoint(model,{e1,...,en},...)
```

DESCRIPTION

`[v1,...,vn] = mphevalpoint(model,{e1,...,en},...)` returns the results from evaluating the expressions `e1,...,en` at the geometry vertices. The values `v1,...,vn` can either be a cell array or a matrix depending on the options.

`[v1,...,vn,unit] = mphevalpoint(model,{e1,...,en},...)` also returns the unit of all expressions `e1,...,en` in the 1xN cell array `unit`.

The function `mphevalpoint` accepts the following property/value pairs:

TABLE 6-5: PROPERTY/VALUE PAIRS FOR THE MPHEVALPOINT COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`dataset`	String		Dataset tag
`dataseries`	`none` | `mean` | `int` | `max` | `min` | `rms` | `std` | `var`	`none`	Data series operation
`matrix`	`off` | `on`	`on`	Return a matrix if possible
`minmaxobj`	`real` | `abs`	`real`	The value being treated if `dataseries` is set to `max` or `min`
`outersolnum`	Positive integer | `all` | `end`	`1`	Solution number for parametric sweep
`selection`	Integer vector | string | `all`	`All domains`	Set selection tag or entity number
`solnum`	Integer vector | `all` | `end`	`all`	Solutions for evaluation
`squeeze`	`on` | `off`	`on`	Squeeze singleton dimension
`t`	Double array		Times for evaluation

The property `Dataset` controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution datasets.

The `Dataseries` property is used to control any filtering of the data series. The supported operations are: average (`mean`), integral (`int`), maximum (`max`), minimum (`min`), root mean square (`rms`), standard deviation (`std`) and variance (`var`).

Set the property `Matrix` to `off` to get the results in a cell array format.

In case the property `Datseries` is either `min` or `max`, you can specify how the values are treated using the property `Minmaxobj`. Use either the real data or the absolute data.

The property `Solnum` is used to select the solution to plot when a parametric, eigenvalue, or time-dependent solver has been used to solve the problem.

The expressions `e1,...,en` are evaluated for one or several solutions. Each solution generates an additional row in the data fields of the post data output structure. The properties `Solnum` and `t` control which solutions are used for the evaluations. The `Solnum` property is available when the dataset has multiple solutions — for example in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `Solnum` is provided, the solutions indicated by the indices provided with the `Solnum` property are used. If `t` is provided, solutions are interpolated. If neither `Solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `Solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

EXAMPLE

Evaluate the temperature on all geometry points:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary'); std.run;

T = mphevalpoint(model,'T');
```

Evaluate the temperature on point 5:

```
T = mphevalpoint(model,'T','selection',5);
```

Evaluate the temperature and the magnitude of the total heat flux on point 5:

```
[T, heatflux, unit] = mphevalpoint(model,{'T','ht.tfluxMag'},...
   'selection',5);
```

Evaluate the temperature at every time step computed with power set to 30:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0)
param.setIndex('plistarr','30 60 90',0);
 std.run;

T = mphevalpoint(model,'T','selection',5,'dataset','dset2');
```

Evaluate the temperature at the seventh time step:

```
T = mphevalpoint(model,'T','selection',5,'dataset','dset2',...
   'solnum',7);
```

Evaluate the temperature at 10.5 sec:

```
T = mphevalpoint(model,'T','selection',5,'dataset','dset2',...
   't',10.5);
```

Evaluate the temperature on point 5 computed with power set to 90:

```
T = mphevalpoint(model,'T','selection',5,'dataset','dset2',...
   'outersolnum',3)
```

Evaluate the temperature average over all time steps:

```
T_avg = mphevalpoint(model,'T','selection',5,...
   'dataset','dset2','dataseries','average');
```

SEE ALSO

`mpheval`, `mphevalglobalmatrix`, `mphevalpointmatrix`, `mphglobal`, `mphint2`, `mphinterp`, `mphparticle`, `mphray`

mphevalpointmatrix

Evaluate matrix quantities at points in the geometry.

SYNTAX

```
M = mphevalpointmatrix(model, expr, ...)
```

DESCRIPTION

`M = mphevalpointmatrix(model, expr, ...)` evaluates the point matrix of the variable `expr` and returns the full matrix M.

The function `mphevalpointmatrix` accepts the following property/value pairs:

TABLE 6-6: PROPERTY/VALUE PAIRS FOR THE MPHEVAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`dataset`	String		Dataset tag
`dataseries`	`none` | `average` | `sum`	`none`	Dataseries operation
`outersolnum`	Positive integer | `end` | `all`	1	Solution number for parametric sweep
`selection`	String | positive integer array		Set selection tag or entity number
`solnum`	Integer vector | `end` | `all`	`all`	Solution for evaluation
`t`	Double array		Time for evaluation

SEE ALSO

`mpheval`, `mphevalglobalmatrix`, `mphevalpoint`, `mphglobal`, `mphint2`, `mphinterp`, `mphparticle`, `mphray`

mphevaluate

Evaluate parameter expressions in models.

SYNTAX

```
mphevaluate(model,expr)
str = mphevaluate(model)
[value,unit,def,descr] = mphevaluate(model,expr,...)
[value,...] = mphevaluate(model,expr,unit)
[value,...] = mphevaluate(model,expr,asvalue)
```

DESCRIPTION

`mphevaluate(model,expr)` evaluates the expression `expr` defined in the Parameters node.

`str = mphevaluate(model)` returns all the expressions defined in the Parameters node as a structs arrays.

`str = mphevaluate(model,{e1,...})` returns all the expressions defined in the cell array `{e1,...}` as a structs arrays. When multiple expressions are evaluated only one output is supported.

`[value,unit,def,descr] = mphevaluate(model,expr,...)` evaluates the expression `expr` and return the unit (`unit`), the definition in the model (`def`) and the description (`descr`).

`[value,...] = mphevaluate(model,expr,unit)` evaluates the expression `expr` in the unit defined by `unit`.

`value = mphevaluate(model,expr,unit,'value')` returns only the value of the expression `expr`.

`unit = mphevaluate(model,expr,unit,'unit')` returns only the unit of the expression `expr`.

`def = mphevaluate(model,expr,unit,'valueunit')` returns as a string the value and the unit of the expression `expr`. This is useful to set a new parameter based on an existing one.

The evaluation does not require an existing solution dataset in the model.

EXAMPLE

Evaluate the parameter power defined in the model:

```
model = mphopen('model_tutorial_llmatlab');
power = mphevaluate(model,'power');
```

Evaluate Temp in degrees Celsius and its definition in the model:

```
[Temp,unit,def] = mphevaluate(model,'Temp','degC');
```

Evaluate an expression of parameters:

```
Temp = mphevaluate(model,'Temp+20[degC]','degF');
```

SEE ALSO

`mpheval`, `mphglobal`, `mphinterp`, `mphparticle`, `mphray`

mphgeom

Plot a geometry in a MATLAB® figure.

SYNTAX

```
mphgeom(model)
mphgeom(model,geomtag,...)
mphgeom(model,geomtag,'workplane',wptag,...)
h = mphgeom(model,geomtag,...)
```

DESCRIPTION

`mphgeom(model)` plots the model geometry in a MATLAB figure. If the model only contains one geometry then the `geomtag` can be empty.

`mphgeom(model,geomtag,...)` plots the model geometry with the tag `geomtag` in a MATLAB figure.

`mphgeom(model,geomtag,'workplane',wptag...)` plots the 2D geometry defined in the workplane with the tag `wptag` in a MATLAB figure.

`h = mphgeom(model,geomtag,...)` also returns a handle of the plotted entities.

The function `mphgeom` accepts the following property/value pairs:

TABLE 6-7: PROPERTY/VALUE PAIRS FOR THE MPHGEOM COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`build`	`on` | `off` | `current` | `string`	`on`	Build the geometry before plotting
`domainlabels`	`on` | `off`	`off`	Show domain labels
`domainlabelscolor`	Char	`k`	Color for domain labels
`edgecolor`	Char	`k`	Edge color
`edgelabels`	`on` | `off`	`off`	Show edge labels
`edgelabelscolor`	Char	`k`	Color for edge labels
`edgemode`	`on` | `off`	`on`	Show edges
`entity`	`point` | `edge` | `boundary` | `domain`		Geometric entity to select
`facealpha`	Double	1	Set transparency value
`facecolor`	Char | vector of RGB values	`gray`	Face color
`facelabels`	`on` | `off`	`off`	Show face labels
`facelabelscolor`	Char	`k`	Color for face labels
`facemode`	`on` | `off`	`on`	Show faces

TABLE 6-7: PROPERTY/VALUE PAIRS FOR THE MPHGEOM COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`parent`	Double		Parent axes
`selection`	Positive integer array		Selection
`vertexlabels`	`on` | `off`	`off`	Show vertex labels
`vertexlabelscolor`	Char	`k`	Color for vertex labels
`vertexmode`	`on` | `off`	`off`	Show vertices
`view`	String | `auto`	`''`	View settings

The `Build` property determines if `mphgeom` build the geometry prior to display it. If the `Build` property is set with a geometry object tag, the geometry is built up to that object. `mphgeom` only displays built geometry objects.

Use the `Workplane` property to show the 2D geometry that is define inside the workplane with specified tag.

EXAMPLE

Plot the model geometry:

```
model = mphopen('model_tutorial_llmatlab.mph');

mphgeom(model)
```

Plot the model geometry with face labels:

```
mphgeom(model,'geom1','facelabels','on','facelabelscolor','r');
```

Plot boundaries 7, 8, 9 and 11:

```
mphgeom(model,'geom1','entity','boundary',...
   'selection',[7:9,11]);
```

The geometry can be plotted with view settings applied. This results in a geometry plot with grid, axes labels, lights, hiding etc. applied to the plot. Usually it is sufficient to use the `auto` setting, but any valid view can be applied:

```
mphmesh(model, 'mesh1', 'view', 'auto')
```

Plot the model geometry on an existing axis:

```
figure(2);
mphgeom(model, 'geom1','parent', gca);
```

SEE ALSO

`mphmesh`, `mphviewselection`

mphgeominfo

Get geometry information.

SYNTAX

```
info = mphgeominfo(model, geomtag)
info = mphgeominfo(model)
[info,data] = mphgeominfo(model,geomtag,...)
```

DESCRIPTION

`info = mphgeominfo(model,geomtag)` returns the information of the geometry defined with the tag `geomtag`. The tag `geomtag` can also be the tag of a geometry part. If only a unique geometry node is defined in the model, the tag `geomtag` is optional.

`[info,data] = mphgeominfo(model,geomtag,...)` returns the information and geometry data of a specific entity.

The function `mphgeominfo` accepts the following property/value pairs to return geometry data:

TABLE 6-8: PROPERTY/VALUE PAIRS FOR THE MPHGEOMINFO COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`build`	`on | off | string`	`off`	Build the geometry object while retrieving the information
`entity`	`face | edge | vertex`		Geometric entity to select
`selection`	Positive integer array		Selection
`steps`	Positive integer array	`[10 10]`	Number of point for data evaluation

The output structure `info` contains the following fields:

TABLE 6-9: FIELDS IN THE INFO STRUCTURE

FIELD	DESCRIPTION
`sdim`	Space dimension
`label`	Label of the selected geometry
`component`	Component tag
`geometricmodel`	Geometric model used by the physics
`autobuildnew`	Build geometric operation automatically when added

TABLE 6-9: FIELDS IN THE INFO STRUCTURE

FIELD	DESCRIPTION
`autorebuild`	Geometry sequence automatically rebuilt
`lengthunit`	Current length unit
`angularunit`	Current angular unit
`objectnames`	Names of all objects that exist in the current state
`current`	Tag of the current feature
`geomrep`	Geometry representation
`scaleunitvalue`	Scale numeric values in the geometry and meshing sequences
`repairtol`	Relative repair tolerance
`repairtoltype`	Repair tolerance type
`absrepairtol`	Absolute repair tolerance
`useconstrdim`	Constraints and dimensions functionality enabled
`constrdimbuild`	Constraints and dimensions used when building the geometry object
`constrdimstatus`	Overall status of the constraints and dimensions
`ispart`	Object is a geometry part
`view`	Current view
`exists`	Geometry object exists
`isaxisymmetric`	Geometry is axisymmetric
`boundingbox`	Bounding box of the geometry objects
`type`	Object type
`Ndomains`	Number of domains
`Nboundaries`	Number of boundaries
`Nedges`	Number of edges
`Nvertices`	Number of vertices
`Nfinitevoids`	Number of finite voids
`Nfaces`	Number of faces
`problems`	List of error/warning messages

For a face evaluation, the output structure `data` contains the following fields:

TABLE 6-10: FIELDS IN THE DATA STRUCTURE FOR FACE ENTITY

FIELD	DESCRIPTION
`paramrange`	Parameter ranges of face
`facex`	Face coordinates
`facedx`	Face first order derivatives
`faceddx`	Face second order derivatives
`facenormal`	Face normal
`faceff1`	Face first fundamental form
`faceff2`	Face second fundamental form
`facegausscurvature`	Face Gauss curvature
`meancurvature`	Face mean curvature
`updown`	Up and down domain indices

For an edge evaluation, the output structure `data` contains the following fields:

TABLE 6-11: FIELDS IN THE DATA STRUCTURE FOR FACE ENTITY

FIELD	DESCRIPTION
`paramrange`	Parameter ranges of face
`edgex`	Edge coordinates
`edgedx`	Edge first order derivatives
`edgeddx`	Edge second order derivatives
`edgecurvature`	Edge curvature values
`edgenormal`*	Edge normal values
`edgetorsion`**	Edge torsion values
`updown`	Up and down domain indices

* `edgenormal` is only returned for 2D space dimension.

** `edgetorsion` is only returned for 3D space dimension.

For a vertex evaluation, the output structure `data` contains the following fields:

TABLE 6-12: FIELDS IN THE DATA STRUCTURE FOR FACE ENTITY

FIELD	DESCRIPTION
`p`	Vertex coordinates
`domainnumber`	Domain index for isolated vertices

EXAMPLE

Get model geometry information:

```
model = mphopen('model_tutorial_llmatlab.mph');

info = mphgeominfo(model)
```

Get geometry data for face number 4:

```
[info, data] = mphgeominfo(model,'geom1','entity','face',...
'selection',4);
```

Get geometry data for face number 4 using a 5x10 grid:

```
[info, data] = mphgeominfo(model,'geom1','entity','face',...
'selection',4,'steps',[5 10]);
```

SEE ALSO

`mphcomponentinfo`, `mphmeshstats`

mphgetadj

Return geometric entity indices that are adjacent to each other.

SYNTAX

```
n = mphgetadj(model,geomtag,returntype,adjtype,adjnumber)
[n,m] = mphgetadj(model,geomtag,returntype,adjtype,adjnumber)
```

DESCRIPTION

`n = mphgetadj(model,geomtag,returntype,adjtype,adjnumber)` returns the indices of the adjacent geometry entities.

`[n,m] = mphgetadj(model,geomtag,returntype,adjtype,adjnumber)` returns the indices of the adjacent geometry entities in `n`. `m` contains the indices of entities that connect the entities `adjnumber` the best.

`returntype` is the type of the geometry entities whose index are returned.

`adjtype` is the type of the input geometric entity.

The entity type can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'` following the entity space dimension defined below:

- `'domain'`: maximum geometry space dimension
- `'boundary'`: maximum geometry space dimension – 1

- `'edges'`: 1 (for 3D geometries only)
- `'point'`: 0

EXAMPLE

Return the indices of the boundaries adjacent to point 2:

```
model = mphopen('model_tutorial_llmatlab');
bnd_idx = mphgetadj(model, 'geom1', 'boundary', 'point', 2);
```

Return the indices of the points adjacent to domain 2:

```
pt_idx = mphgetadj(model, 'geom1', 'point', 'domain', 2);
```

Return the indices of the adjacent edges to boundaries 1, 2 and 9 and the indices of the edges that connect the boundaries the best. Then show the results in a figure:

```
[idx1, idx2] = mphgetadj(model, 'geom1', 'edge',...
   'boundary', [1, 2, 9]);
mphviewselection(model,'geom1',[1,2,9],'entity','boundary',...
   'edgemode','off')
hold on
mphviewselection(model,'geom1',idx1,'entity','edge',...
   'edgemode','off','facemode','off','edgecolorselected','b')
mphviewselection(model,'geom1',idx2,'entity','edge',...
   'edgemode','off','facemode','off','edgecolorselected','g')
```

SEE ALSO

`mphgetcoords`, `mphselectbox`, `mphselectcoords`

mphgetcoords

Return point coordinates of geometry entities.

SYNTAX

```
c = mphgetcoords(model,geomtag,entitytype,entitynumber)
```

DESCRIPTION

`c = mphgetcoords(model,geomtag,entitytype,entitynumber)` returns the coordinates of the points that belong to the entity object with the type `entitytype` and the index `entitynumber`.

The `entitytype` property can be one of `'point'`, `'edge'`, `'boundary'` or `'domain'` following the entity space dimension defined below:

- `'domain'`: maximum geometry space dimension
- `'boundary'`: maximum geometry space dimension -1
- `'edge'`: 1 (only for 3D geometry)
- `'point'`: 0

EXAMPLE

Return the coordinates of points that belong to domain 2:

```
model = mphopen('model_tutorial_llmatlab');
c0 = mphgetcoords(model, 'geom1', 'domain', 2);
```

Return the coordinates of points that belong to boundary 5:

```
c1 = mphgetcoords(model, 'geom1', 'boundary', 5);
```

Return the coordinates of point number 10:

```
c2 = mphgetcoords(model, 'geom1', 'point', 10);
```

SEE ALSO

`mphgetadj`, `mphselectbox`, `mphselectcoords`

mphgetexpressions

Get the model variables and model parameters expressions.

SYNTAX

```
expr = mphgetexpressions(modelnode)
```

DESCRIPTION

`expr = mphgetexpressions(modelnode)` returns expressions from the node `modelnode` as a cell array. `expr` contains the list of the variable names, the variable expressions and the variable descriptions.

Note that not all nodes have expressions defined.

EXAMPLE

Get the expressions defined in the parameters node

```
model = mphopen('model_tutorial_llmatlab');
expr = mphgetexpressions(model.param)
```

SEE ALSO

mphgetproperties, mphgetselection, mphmodel, mphnavigator, mphsearch

mphgetproperties

Get the properties from a model node.

SYNTAX

```
str = mphgetproperties(modelnode)
[str,allowed] = mphgetproperties(modelnode)
```

DESCRIPTION

`str = mphgetproperties(modelnode)` returns the structure `str` containing the properties that are defined for the node `modelnode`.

`[str,allowed] = mphgetproperties(modelnode)` also returns the structure `allowed` containing the allowed values for the corresponding properties.

The function `mphgetproperties` accepts the following property/value pairs:

TABLE 6-13: PROPERTY/VALUE PAIRS FOR THE MPHGETPROPERTIES COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`propnames`	`String cell array`		List of properties to show only
`returnstrings`	`on` | `off`	`on`	Return data as string
`showsel`	`on` | `off`	`off`	Return the properties selection

EXAMPLE

Build the mesh in the model `model_tutorial_llmatlab.mph` and get the mesh size properties and their allowed values:

```
model = mphopen('model_tutorial_llmatlab');
mesh1 = model.component('comp1').mesh('mesh1');
mesh1.run;
msize = mesh1.feature('size');
[prop, allowed] = mphgetproperties(msize)
```

Get the min and max mesh size properties only:

```
prop = mphgetproperties(msize,'propnames',{'hmin','hmax'})
```

Return the property data not as string:

```
prop = mphgetproperties(msize,'propnames',{'hmin','hmax'},...
```

```
'returnstrings','off')
```

SEE ALSO

mphgetexpressions, mphgetselection, mphmodel, mphnavigator, mphsearch

mphgetselection

Get information about a selection node.

SYNTAX

```
info = mphgetselection(selnode)
```

DESCRIPTION

`info = mphgetselection(selnode)` returns the selection data of the selection node `selnode`.

The output `info` is a MATLAB® structure defined with the following fields:

- `dimension`, the space dimension of the geometric entity selected.
- `geom`, the geometry tag.
- `entities`, the indices of the selected entities.
- `isGlobal`, a Boolean expression that indicates if the selection is global.

EXAMPLE

Add a selection node to the model `busbar.mph` and retrieve its information:

```
model = mphopen('model_tutorial_llmatlab.mph');
ball = model.selection.create('ball','Ball');
ball.set('entitydim',2);
ball.set('posz',11e-3');
ball.set('r',1e-5);
info = mphgetselection(model.selection('ball'))
```

SEE ALSO

mphgetexpressions, mphgetproperties, mphmodel, mphnavigator, mphsearch

mphgetu

Return a solution vector.

SYNTAX

```
U = mphgetu(model,...)
[U,Udot] = mphgetu(model,...)
```

DESCRIPTION

`U = mphgetu(model)` returns the solution vector `U` for the default solution dataset.

`[U,Udot] = mphgetu(model,...)` returns in addition `Udot`, which is the time derivative of the solution vector. This syntax is available for a time-dependent solution only.

For a time-dependent and parametric analysis type, the last solution is returned by default. For an eigenvalue analysis type the first solution number is returned by default.

The function `mphgetu` accepts the following property/value pairs:

TABLE 6-14: PROPERTY/VALUE PAIRS FOR THE MPHGETU COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`soltag`	String		Solver node tag
`solnum`	Positive integer vector		Solution for evaluation
`type`	String	`Sol`	Solution type
`matrix`	`off` | `on`	`on`	Store as matrix if possible

The `Solname` property set the solution dataset to use associated with the defined solver node.

Type is used to select the solution type. This is `'Sol'` by default. The valid types are: `'Sol'` (main solution), `'Reacf'` (reaction force), `'Adj'` (adjoint solution), `'Fsens'` (functional sensitivity) and `'Sens'` (forward sensitivity).

If `Solnum` is a vector and the result has been obtained with the same mesh then the solution is stored in a matrix if the `Matrix` option is set to `'on'`.

EXAMPLE

Extract the solution vector:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

U = mphgetu(model);
```

Extract the reaction force vector:

```
reacf = mphgetu(model,'type','reacf');
```

Extract the solution vectors for the first and the last time step:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;

U = mphgetu(model,'solnum',[1,26]);
```

Extract the solution vector computed with power set to 30:

```
U = mphgetu(model,'soltag','sol3');
```

SEE ALSO

mphsolinfo

mphglobal

Evaluate global quantities.

SYNTAX

```
[d1,...,dn] = mphglobal(model,{e1,...,en},...)
[d1,...,dn,unit] = mphglobal(model,{e1,...,en},...)
```

DESCRIPTION

`[d1,...,dn] = mphglobal(model,{e1,...,en},...)` returns the results from evaluating the global quantities specified in the string expression `e1,..., en`.

`[d1,...,dn,unit] = mphglobal(model,{e1,...,en},...)` also returns the unit of the expressions `e1,..., en`. `unit` is a `nx1` cell array.

The function `mphglobal` accepts the following property/value pairs:

TABLE 6-15: PROPERTY/VALUE PAIRS FOR THE MPHGLOBAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`complexfun`	`off` | `on`	`on`	Use complex-valued functions with real input
`complexout`	`off` | `on`	`on`	Return complex values
`dataset`	String	Active solution dataset	Dataset tag

TABLE 6-15: PROPERTY/VALUE PAIRS FOR THE MPHGLOBAL COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
matherr	off | on	off	Error for undefined operations or expressions
outersolnum	Positive integer | all | end	1	Solution number for parametric sweep
phase	Scalar	0	Phase angle in degrees
solnum	Integer vector | all | end	all	Solution for evaluation
t	Double array		Time for evaluation
unit	String | cell array		Unit to use for the evaluation

The property Dataset controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on solution datasets.

When the property Phase is used, the solution vector is multiplied with exp(i*phase) before evaluating the expression.

The expressions ei are evaluated for one or several solutions. Each solution generates an additional row in the output data array di. The properties solnum and t control which solutions are used for the evaluations. The solnum property is available when the dataset has multiple solutions — for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The t property is available only for time-dependent problems. If solnum is provided, the solutions indicated by the indices provided with the solnum property are used. If t is provided, solutions are interpolated. If neither solnum nor t is provided, all solutions are evaluated.

For time-dependent problems, the variable t can be used in the expressions ei. The value of t is the interpolation time when the property t is provided, and the time for the solution, when solnum is used. Similarly, lambda and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

In case of multiple expression if the unit property is defined with a string, the same unit is used for both expressions. To use different units, set the property with a cell array. In case of inconsistent unit definition, the default unit is used instead.

Solnum is used to select the solution number when a parametric, eigenvalue, or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

EXAMPLE

Evaluate the maximum temperature in the model

```
model = mphopen('model_tutorial_llmatlab');
model.cpl.create('maxop','Maximum','geom1').selection.all;
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

maxT = mphglobal(model,'maxop(T)')
```

Evaluate the maximum temperature in the model in degrees Celsius

```
maxT = mphglobal(model,'maxop(T)','unit','degC')
```

Evaluate a global expression at every time step computed with power set to 30:

```
model = mphopen('model_tutorial_llmatlab');
model.cpl.create('maxop', 'Maximum', 'geom1').selection.all;
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;

maxT = mphglobal(model,'maxop(T)','dataset','dset2');
```

Evaluate `maxop(T)` for the first and fifth time step:

```
maxT = mphglobal(model,'maxop(T)',dataset','dset2',...
   'solnum',[1,5]);
```

Evaluate `maxop(T)` at 20.512 sec:

```
maxT = mphglobal(model,'maxop(T)',dataset','dset2',...
   't',20.512);
```

Evaluate `maxop(T)` at every time step computed with power set to 90:

```
maxT = mphglobal(model,'maxop(T)','dataset','dset2',...
   'outersolnum',3);
```

SEE ALSO

`mpheval`, `mphevalglobalmatrix`, `mphevalpoint`, `mphevalpointmatrix`, `mphint2`, `mphinterp`

mphimage2geom

Convert image data to a geometry.

SYNTAX

```
model = mphimage2geom(imagedata,level,...)
```

DESCRIPTION

`model = mphimage2geom(imagedata,level,...)` converts the image contained in `imagedata` into a geometry which is returned in the model object `model`.

The contour of the image is defined by the value level. `imagedata` must be a 2D matrix.

The function `mphimage2geom` accepts the following property/value pairs:

TABLE 6-16: PROPERTY/VALUE PAIRS FOR THE MPHIMAGE2GEOM COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`compose`	`on` | `off`	`on`	Create compose nodes for overlapping solids
`curvetype`	`auto` | `polygon`	auto	Type of curve to create the geometry object
`geom`	Geometry node	`geom1`	Geometry creation
`minarea`	Value	1	Minimum area for interior curves (in square pixels)
`mindist`	Value	1	Minimum distance between coordinates in curves (in pixels)
`modeltag`	String	`Model`	Model tag in a COMSOL *server*
`rectangle`	`on` | `off`	`off`	Insert rectangle in the geometry
`rtol`	Value	`1e-3`	Relative tolerance for interpolation curves
`scale`	Value	1	Scale factor from pixels to geometry scale
`type`	`solid` | `closed` | `open`	`solid`	Type of geometry object

The default curve types creates a geometry with the best suited geometrical primitives. For interior curves this is Interpolation Curves, and for curves that are touching the perimeter of the image, Polygons is used.

To add the geometry created with `mphimage2geom`, specify the geometry node with property `geom`.

EXAMPLE

Create a set of point coordinates:

```
p = (peaks+7)*5;
```

Display contour plot of the point data:

```
figure(1); [c,h] = contourf(p); clabel(c, h); colorbar
```

Create a geometry object following the contour made with point of value 50:

```
model = mphimage2geom(p, 50);
figure(2); mphgeom(model)
```

Create the same geometry object with a scale factor of 1e-3 and add it into an existing 3D model:

```
model = mphopen('model_tutorial_llmatlab');
wp1 = model.component('comp1').geom('geom1').feature.create('wp1',
'WorkPlane');
wp1.set('quickz', 1e-2);
mphimage2geom(p, 50,'scale',1e-3,wp1.geom);
mphgeom(model)
```

Create a geometry using MRI data. The geometry object is created following the contour made with point of value 30 and disregard objects with an area (in pixel) lower than 2:

```
mri = load('mri');
im = mri.D(:,:,1,1);
figure(1); image(im);
mphimage2geom(im, 30,'minarea',2);
```

mphinputmatrix

Add a matrix system for linear solvers.

SYNTAX

```
mphinputmatrix(model,str,soltag,soltypetag)
```

DESCRIPTION

`mphinputmatrix(model,str,soltag,soltypetag)` adds the system matrices and vectors stored in the MATLAB® structure `str` to the `model`. The system matrices is associated to the linear solver configuration defined with the tag `soltag` and solved with the solver defined with the tag `soltypetag`.

`soltypetag` can only be one of the following solver type: Stationary, Eigenvalue, Time.

A valid structure for a stationary solver includes the following fields:

FIELD NAME	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian

A valid structure for a time-dependent/ eigenvalue solver includes the following fields:

FIELD NAME	DESCRIPTION
K	Stiffness matrix
L	Load vector
M	Constraint vector
N	Constraint Jacobian
D	Damping matrix
E	Mass matrix

There is also the possibility to include the constraint force Jacobian vector `NF`.

Once the matrix system is loaded in the model, the solver configuration is set ready to run.

Note: The system matrices are not stored in the model when it is saved as an MPH-file or loaded into the COMSOL Desktop.

EXAMPLE

Create a model with a square geometry

```
model = ModelUtil.create('Model');
```

```
comp = model.component.create('comp1', true);
geom = comp.geom.create('geom1', 2);
geom.create('sq1', 'Square');
geom.run;
```

Add an Equation General Form physics interface

```
g = comp.physics.create('g', 'GeneralFormPDE', 'geom1');
g.prop('ShapeProperty').set('order', 1)
g.prop('ShapeProperty').set('boundaryFlux', false);
cons = g.create('cons1', 'Constraint', 1).set('R', 'u');
cons.selection.set([1 2]);
```

Create a mapped mesh

```
map = comp.mesh.create('mesh1').create('map1', 'Map');
map.create('dis1', 'Distribution').set('numelem', 2);
map.feature('dis1').selection.set([1 2]);
```

Set-up the study and the solver configuration for a stationary problem:

```
std = model.study.create('std1');
std.create('stat', 'Stationary');
sol = model.sol.create('sol1');
sol.study('std1');
sol.feature.create('st1', 'StudyStep').set('studystep', 'stat');
sol.feature.create('v1', 'Variables');
sol.feature.create('s1', 'Stationary');
```

Extract the linear stationary matrix system in MATLAB:

```
str = mphmatrix(model,'sol1','out',{'K','L','M','N'},...
'initmethod','sol','initsol','zero');
```

Change the linear system by scaling the stiffness matrix:

```
str.K = str.K*0.5;
```

Insert the system matrix back to the model:

```
mphinputmatrix(model,str,'sol1','s1')
```

Run the solver configuration:

```
model.sol('sol1').runAll;
```

SEE ALSO

`mphmatrix`, `mphxmeshinfo`

mphint2

Perform integration of expressions.

SYNTAX

```
[v1,...,v2] = mphint2(model,{e1,...,en},edim,...)
[v1,...,v2,unit] = mphint2(model,{e1,...,en},edim,...)
```

DESCRIPTION

`[v1,...,vn] = mphint2(model,{e1,...,en},...)` evaluates the integrals of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension, as a string: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphint2(model,{e1,...,en},...)` also returns the units of the integral in a 1xN cell array.

The function `mphint2` accepts the following property/value pairs:

TABLE 6-17: PROPERTY/VALUE PAIRS FOR THE MPHINT2 COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`dataseries`	`none` | `average` | `integral` | `maximum` | `rms` | `stddev` | `variance`	`none`	Data series operation
`dataset`	String	Active solution dataset	Dataset tag
`intorder`	Positive integer	4	Integration order
`intsurface`	`on` | `off`	`off`	Compute surface integral
`intvolume`	`on` | `off`	`off`	Compute volume integral
`matrix`	`on` | `off`	`on`	Returns data as a matrix or as a cell
`method`	`auto` | `integration` | `summation`	`auto`	Integration method
`outersolnum`	Positive integer | `all` | `end`	1	Solution number for parametric sweep
`selection`	Integer vector | string | `all`	`all`	Selection list or named selection
`solnum`	Integer vector | `end` | `all`	`all`	Solution for evaluation
`squeeze`	`on` | `off`	`on`	Squeeze singleton dimensions
`t`	Double array		Time for evaluation

The property `dataset` controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution datasets.

The expressions `e1,...,en` are integrated for one or several solutions. Each solution generates an additional column in the returned matrix. The properties `solnum` and `t` control which solutions are used for the integrations. The solnum property is available when the dataset has multiple solutions — for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `solnum` is provided, the solutions indicated by the indices provided with the `solnum` property are used. If `t` is provided, solutions are interpolated. If neither `solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

The `unit` property defines the unit of the integral, if a inconsistent unit is entered, the default unit is used. In case of multiple expression, if the `unit` property is defined with a string, the same unit is used for both expressions. To use different units, set the property with a cell array. In case of inconsistent unit definition, the default unit is used instead.

`Solnum` is used to select the solution number when a parametric, eigenvalue, or time-dependent solver has been used.

`Outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

EXAMPLE

Integrate the normal heat flux across all boundaries:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

[Q, unit] = mphint2(model,'ht.ntflux','surface');
```

Integrate the normal heat flux across all exterior boundaries

```
[Q, unit] = mphint2(model,'ht.ntflux','surface',...
   'selection',[1:5,7:12]);
```

SEE ALSO

mpheval, mphevalglobalmatrix, mphevalpoint, mphevalpointmatrix, mphint2, mphinterp, mphparticle, mphray

mphinterp

Evaluate expressions in arbitrary points or datasets.

SYNTAX

```
[v1,...,vn] = mphinterp(model,{e1,...,en},'coord',coord,...)
[v1,...,vn] = mphinterp(model,{e1,...,en},'dataset',dsettag,...)
[v1,...,vn,unit] = mphinterp(model,{e1,...,en},...)
```

DESCRIPTION

`[v1,...,vn] = mphinterp(model,{e1,...,en},'coord',coord,...)` evaluates expressions `e1,...en` at the coordinates specified in the double matrix `coord`. Evaluation is supported only on Solution datasets.

`[v1,...,vn] = mphinterp(model,{e1,...,en},'dataset',dsettag,...)` evaluates expressions `e1,...en` on the specified dataset `dsettag`. In this case the dataset needs to be of a type that defines an interpolation in itself, such as cut planes, revolve, and so forth.

`[v1,...,vn,unit] = mphinterp(model,{e1,...,en},...)` returns in addition the unit of the expressions.

The function `mphinterp` accepts the following property/value pairs:

TABLE 6-18: PROPERTY/VALUE PAIRS FOR THE MPHINTERP COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`complexfun`	`off` | `on`	`on`	Use complex-valued functions with real input
`complexout`	`off` | `on`	`on`	Return complex values
`coord`	Double array		Coordinates for evaluation
`coorderr`	`off` | `on`	`on`	Give an error message if all coordinates are outside the geometry
`dataset`	String	`Auto`	Dataset tag

TABLE 6-18: PROPERTY/VALUE PAIRS FOR THE MPHINTERP COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`differential`	`off` | `on`	`on`	Whether the expression should be linearized at the linearization point. Applicable only if `evalmethod` is `harmonic`
`edim`	`point` | `edge` | `boundary` | `domain` | `0` | `1` | `2` | `3`	Geometry space dimension	Element dimension for evaluation
`evalmethod`	`linpoint` | `harmonic` | `lintotal` | `lintotalavg` | `lintotalrms` | `lintotalpeak`	`harmonic`	Applicable only for solutions with a stored linearization point. Controls if the linearization point, the perturbation, or a combination should be used when evaluating the expression.
`ext`	Double between 0 and 1	`0.1`	Extrapolation distance: How much outside the mesh that the interpolation searches. The scale is in terms of the local element size.
`matherr`	`off` | `on`	`off`	Error for undefined operations or expressions
`outersolnum`	Positive integer | `all` | `end`	`1`	Solution number for parametric sweep
`phase`	Scalar	0	Phase angle in degrees
`recover`	`off` | `ppr` | `pprint`	`off`	Accurate derivative recovery
`selection`	Positive integer array | `all`	`all`	Selection list
`solnum`	Positive integer array | `all` | `end`	`all`	Inner solutions for evaluation
`t`	Double array		Time for evaluation
`unit`	String | Cell array		Unit to use for the evaluation

The columns of the matrix `coord` are the coordinates for the evaluation points. If the number of rows in `coord` equals the space dimension, then `coord` are global coordinates, and the property `edim` determines the dimension in which the expressions are evaluated. For instance, `edim='boundary'` means that the expressions are evaluated on boundaries in a 3D model. If `edim` is less than the space dimension, then the points in `coord` are projected onto the closest point on a domain of dimension

`edim`. If, in addition, the property `selection` is given, then the closest point on domain number `selection` in dimension `edim` is used.

If the number of rows in `coord` is less than the space dimension, then these coordinates are parameter values on a geometry face or edge. In that case, the domain number for that face or edge must be specified with the property `selection`.

The expressions that are evaluated can be expressions involving variables, in particular physics interface variables.

The matrices `v1,...,vn` are of the size `k`-by-size(`coord`,2), where `k` is the number of solutions for which the evaluation is carried out, see below. The value of expression `ei` for solution number `j` in evaluation point `coord(:,m)` is `vi(j,m)`.

The vector `pe` contains the indices `m` for the evaluation points `code(:,m)` that are outside the mesh, or, if a domain is specified, are outside that domain.

The property `Data` controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution datasets. The active solution dataset is used by default.

The property `edim` decides which elements to evaluate on. Evaluation takes place only on elements with space dimension `edim`. If not specified, `edim` equal to the space dimension of the geometry is used. The setting is specified as one of the following strings `'point', 'edge', 'boundary'` or `'domain'`. In previous versions it was only possible to specify `edim` as a number. For example, in a 3D model, if evaluation is done on edges (1D elements), `edim` is 1. Similarly, for boundary evaluation (2D elements), `edim` is 2, and for domain evaluation (3D elements), `edim` is 3 (default in 3D).

The property `evalmethod` decides which solution to use in presence of linearization point. Set the property value to `harmonic` to harmonic perturbation analysis, `linpoint` evaluates the expression by taking the values of any dependent variables from the linearization point of the solution, `lintotal` evaluates the expression by adding the linearization point and the harmonic perturbation and taking the real part of this sum. `lintotalavg` (`lintotalrms` and `lintotalpeak`) do the same as with `lintotal` and then averaging (taking the RMS and taking the maximum respectively) over all phases of the harmonic perturbation. When `harmonic` is selected, the property differential specify to evaluate the differential of the expression with respect to the perturbation at the linearization point (`on`) or to evaluates the expression by taking the values of any dependent variables from the harmonic perturbation part of the solution (`off`).

Use `Recover` to recover fields using polynomial-preserving recovery. This techniques recover fields with derivatives such as stresses or fluxes with a higher theoretical convergence than smoothing. Recovery is expensive so it is turned off by default. The value `pprint` means that recovery is performed inside domains. The value `ppr` means that recovery is also applied on all domain boundaries.

The property `Refine` constructs evaluation points by making a regular refinements of each element. Each mesh edge is divided into `Refine` equal parts.

When the property `phase` is used, the solution vector is multiplied with `exp(i*phase)` before evaluating the expression.

The expressions `e1,...,en` are evaluated for one or several solutions. Each solution generates an additional row in the data fields of the post data output structure. The properties `solnum` and `t` control which solutions are used for the evaluations. The `solnum` property is available when the dataset has multiple solutions — for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `solnum` is provided, the solutions indicated by the indices provided with the `solnum` property are used. If `t` is provided, solutions are interpolated. If neither `solnum` nor `t` is provided, all solutions are evaluated.

For time-dependent problems, the variable `t` can be used in the expressions `ei`. The value of `t` is the interpolation time when the property `t` is provided, and the time for the solution, when `solnum` is used. Similarly, `lambda` and the parameter are available as eigenvalues for eigenvalue problems and as parameter values for parametric problems, respectively.

In case of multiple expression, if the `unit` property is defined with a string, the same unit is used for both expressions. To use different units, set the property with a cell array. In case of inconsistent unit definition, the default unit is used instead.

The property `solnum` is used to select the solution number when a parametric, eigenvalue, or time-dependent solver has been used.

The property `outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

EXAMPLE

Evaluate the temperature at given coordinates:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
```

```
std.feature.create('stat','Stationary');
std.run;
```

```
coord = [0,0,1e-2;0,0,1e-2;0,1e-2,1e-2];
T = mphinterp(model,'T','coord',coord);
```

Evaluate both the temperature and the heat flux magnitude:

```
[T,tfluxMag] = mphinterp(model,{'T','ht.tfluxMag'},...
   'coord',coord);
```

Evaluate the temperature field on a structure grid:

```
x0 = [0,1e-2,2.5e-2,5e-2]; y0 = x0; z0 = [5e-3,1e-2,1.1e-2];
[x,y,z] = meshgrid(x0,y0,z0); xx = [x(:),y(:),z(:)]';
T = mphinterp(model,'T','coord',xx);
```

Evaluate the temperature on boundary 7 using global coordinates:

```
x0 = [0,5e-3,1e-2]; y0 = x0; z0 = [1.1e-2];
[x,y,z] = meshgrid(x0,y0,z0); xx = [x(:),y(:),z(:)]';
T  = mphinterp(model,'T','coord',xx,'edim','boundary',...
   'selection',7);
```

Evaluate the temperature and evaluation point global coordinates on boundary 7 using local coordinates:

```
s10 = [0,0.25,0.5]; s20 = [0,0.25,0.5];
[s1,s2] = meshgrid(s10,s20); ss = [s1(:),s2(:)]';
[x,y,z,T]  = mphinterp(model,{'x','y','z','T'},'coord',ss,...
   'edim','boundary','selection',7);
```

Modify the extrapolation distance for point coordinates outside of the geometry:

```
coord = [5e-2;5e-2;1.1e-2];
T = mphinterp(model,'T','coord',coord)
T = mphinterp(model,'T','coord',coord,'ext',0.5);
```

Extract data using a cut line dataset. First create the cut line dataset, then evaluate the temperature field along the line:

```
cln = model.result.dataset.create('cln', 'CutLine3D');
cln.setIndex('genpoints','1e-2',1,0);
cln.setIndex('genpoints','1e-2',0,2);
cln.setIndex('genpoints','5e-2',1,0);
T = mphinterp(model,'T','dataset','cln');
```

Evaluation including several solution

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
```

```
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;
```

Evaluate the temperature at every time step computed with power set to 30:

```
coord = [0 0 1e-2;0 0 1e-2;0 1e-2 1e-2];
T = mphinterp(model,'T','coord',coord,'dataset','dset2');
```

Evaluate the temperature at the fifth time step:

```
T = mphinterp(model,'T','coord',coord,'dataset','dset2',...
   'solnum',5);
```

Evaluate the temperature at 10.5 sec:

```
T = mphinterp(model,'T','coord',coord,'dataset','dset2',...
    't',10.5);
```

Evaluate the temperature at every time step computed with power set to 90:

```
T = mphinterp(model,'T','coord',coord,'dataset','dset2',...
   'outersolnum',3)
```

SEE ALSO

mpheval, mphevalglobalmatrix, mphevalpoint, mphevalpointmatrix, mphint2, mphint2, mphparticle, mphray

mphinterpolationfile

Save data in files readable by the Interpolation feature.

SYNTAX

```
mphinterpolationfile(filename,type,data)
mphinterpolationfile(filename,type,data,xdata)
mphinterpolationfile(filename,type,data,xdata,ydata)
```

DESCRIPTION

`mphinterpolationfile(filename,type,data)` saves the NxM matrix `data` into the text file `filename` with the format `type`. The interpolation coordinates are vectors with values from 1 to N and 1 to M.

`mphinterpolationfile(filename,type,data,xdata)` saves the vector `data` and the interpolation coordinate `xdata` into the text file `filename` with the format `type`.

`mphinterpolationfile(filename,type,data,xdata,ydata)` saves the matrix `data` and the interpolation coordinate vectors `xdata` and `ydata` into the text file `filename` with the format `type`.

`type` can be either `'grid'`, `'sectionwise'` or `'spreadsheet'`.

EXAMPLE

Create random 10x10 interpolation data to file using grid format:

```
data = cumsum(0.1*randn(size(10)));
mphinterpolationfile('datagrid.txt','grid',data);
```

1D interpolation data to file using spreadsheet format:

```
t = 0:0.05:2*pi;
z = sin(cos(t)*4)+sin(51*t)*0.05;
mphinterpolationfile('dataspread.txt','spreadsheet',z,t);
```

2D interpolation data to file using sectionwise format:

```
z = magic(9); x = 1:9; y = 0:8;
mphinterpolationfile('datasection.txt','sectionwise',z,x,y);
```

SEE ALSO

`mphquad2tri`, `mphreadstl`, `mphsurf`, `mphwritestl`

mphlaunch

Launch COMSOL Desktop, connect it to the running server, and import an application.

SYNTAX

```
mphlaunch
mphlaunch(model)
mphlaunch ModelTag
mphlaunch(..., timeout)
```

DESCRIPTION

`mphlaunch` launches a COMSOL Multiphysics Client and connect it to the same server as MATLAB® is connected to. Then it imports the model on the server into the COMSOL Multiphysics Client.

`mphlaunch(model)` does the same as above, but uses the `model` argument to select which model is imported.

`mphlaunch ModelTag` uses the model with the tag `'ModelTag'` in the server to be imported. This can also be done using the syntax: `mphlaunch('ModelTag')`

`mphlaunch(..., tms)` uses the timeout `tms` (in milliseconds) to force MATLAB to wait until the COMSOL *server* is free again. The default timeout value is 500. A negative value results in no timeout.

EXAMPLE

Load the file `model_tutorial_llmatlab.mph`:

```
model = mphopen('model_tutorial_llmatlab');
```

Launch a COMSOL Multiphysics Client, connect it with the running server, import the model defined as `model`:, and set a timeout of 1 s:

```
mphlaunch(model,1000);
```

mphload

Load a COMSOL Multiphysics model MPH-file.

SYNTAX

```
model = mphload(filename)
model = mphload(filename, mtag)
model = mphload(filename, mtag, '-history')
model = mphload(filename, mtag, pwd)
[model, filename] = mphload(filename, ...)
```

DESCRIPTION

`model = mphload(filename)` loads a COMSOL model object saved with the name `filename` and assigns the default tag `Model` in the COMSOL *server*. If a model with tag `Model` already exists and is also open in a COMSOL Multiphysics client, the loaded model an index number is appended to the tag, for instance `Model1`. The model object is accessible at the MATLAB prompt using the variable `model`.

`model = mphload(filename, mtag)` loads a COMSOL model object and assigns the tag `mtag` in the COMSOL *server*.

`model = mphload(filename, mtag, '-history')` turns on model history recording.

`model = mphload(filename, mtag, pwd)` loads the COMSOL model object saved with the name `filename` protected with the password `pwd`.

`model = mphload(mtag)` link the model already loaded on the COMSOL *server* with the tag `mtag`. The model object is accessible at the MATLAB prompt using the variable `model`.

`[model, filenameloaded] = mphload(filename, ...)` also returns the full file name `filenameloaded` of the file that was loaded.

The model tag `mtag` and the password `pwd` are defined as string.

If the model tag is the same as a model that is currently in the COMSOL *server* the loaded model overwrites the existing one.

Note that MATLAB® searches for the model on the MATLAB path if an absolute path is not supplied.

`mphload` turns off the model history recording by default, unless the property `'-history'` is used.

The extension `mph` can be omitted.

`mphload` does not look for lock file when opening a model in the COMSOL *server*.

EXAMPLE

Load the file `model_tutorial_llmatlab.mph`:

```
model = mphload('model_tutorial_llmatlab');
```

Load the file `model_tutorial_llmatlab.mph` and set the model name in the COMSOL *server* to `Model2`:

```
model = mphload('model_tutorial_llmatlab','Model2');
```

Load `model_tutorial_llmatlab.mph` and return the filename:

```
[model, filename] = mphload('model_tutorial_llmatlab');
```

SEE ALSO

`mphopen`, `mphsave`

mphmatrix

Get model matrices.

SYNTAX

```
str = mphmatrix(model,soltag,'Out',...)
```

DESCRIPTION

`str = mphmatrix(model,soltag,'Out',{'A'},...)` returns a MATLAB® structure `str` containing the matrix `A` assembled using the solver node `soltag` and accessible as `str.A`, `A` being taken from the `Out` property list.

`str = mphmatrix(model,soltag,fname,'Out',{'A','B',...})` returns a MATLAB structure `str` containing the matrices `A`, `B`, … assembled using the solver node `solname` and accessible as `str.A` and `str.B`, `A` and `B` being taken from the `Out` property list.

The function `mphmatrix` accepts the following property/value pairs:

TABLE 6-19: PROPERTY/VALUE PAIRS FOR THE MPHMATRIX COMMAND

PROPERTY	EXPRESSION	DEFAULT	DESCRIPTION
`complexfun`	`on` | `off`	`off`	Use complex-valued functions with real input
`eigname`	String	`lambda`	Eigenvalue name
`eigref`	Double	`0`	Value of eigenvalue linearization point
`extractafter`	Solution feature tag	end	Specify where in the solver sequence to extract the matrices
`initmethod`	`init` | `sol`	`init`	Use linearization point
`initsol`	String | `zero`	Active solver tag	Solution to use for linearization
`matherr`	`on` | `off`	`on`	Error for undefined operations
`nullfun`	`flnullorth` | `flspnull` | `flexplicit` | `auto`	`auto`	Null-space function
`out`	Cell array of strings		List of matrices to assemble
`rowscale`	`on` | `off`	`on`	Row equilibration
`solnum`	Positive integer | `auto`	`auto`	Solution number
`study`	Study tag	`{First study}`	Study to use with `initmethod`
`symmetry`	`on` | `off` | `hermitian` | `auto`	`auto`	Symmetric matrices

The following values are valid for the `out` property:

Property/Value Pairs for the property out.

PROPERTY	EXPRESSION	DESCRIPTION
`out`	`K`	Stiffness matrix
	`L`	Load vector
	`M`	Constraint vector
	`N`	Constraint Jacobian
	`D`	Damping matrix
	`E`	Mass matrix
	`NF`	Constraint force Jacobian
	`NP`	Optimization constraint Jacobian (*)
	`MP`	Optimization constraint vector (*)
	`MLB`	Lower bound constraint vector (*)
	`MUB`	Upper bound constraint vector (*)
	`Kc`	Eliminated stiffness matrix
	`Lc`	Eliminated load vector
	`Dc`	Eliminated damping matrix
	`Ec`	Eliminated mass matrix
	`Null`	Constraint null-space basis
	`Nullf`	Constraint force null-space matrix
	`ud`	Particular solution ud
	`uscale`	Scale vector

(*) Requires the Optimization Module.

Note that the assembly of the eliminated matrices uses the current solution vector as scaling method. To get the unscaled eliminated system matrices, it is required to set the scaling method to `'none'` in the variables step of the solver configuration node.

The matrices are assembled using the current solution available as linearization point unless the `initmethod` property is provided. In case of the presence of a solver step node you need to either disable the solver step node in the model or set the property `extractafter` with the tag of the Dependent Variables node.

In case the matrices are assembled after a solver step node (which is the case by default), the load vector corresponds then to the residual of the problem.

The function `mphmatrix` does not solve the problem as the assembly is performed before the solver node in the solution sequence. You can specify the solution feature node after which to assemble the system matrices with the property `extractafter`. This is useful if you need to compute the solution before extracting the matrices or if you have a solution sequence using different solver sequences and you want to extract the matrices for a specific one.

The section in the *COMSOL Multiphysics Reference Manual*, describes the functionality corresponding to the properties `complexfun`, `nullfun`, and `rowscale`.

Use the property `symmetric` to assemble the model matrix system as symmetric/Hermitian, or you can use the automatic feature to find out (see Advanced in the *COMSOL Multiphysics Reference Manual*).

EXAMPLE

Evaluate the system matrices of a stationary problem

```
model = mphopen('model_tutorial_llmatlab');
model.mesh('mesh1').autoMeshSize(8);
std = model.study.create('std1');
std.feature.create('stat', 'Stationary');
std.run;
```

Get the stationary matrix system, use the initial solution as linearization point:

```
str = mphmatrix(model,'sol1','out',{'K','L','M','N'},...
   'initmethod','init');
```

Display the sparsity of the stiffness matrix and the constraint Jacobian and compute the total load applied in the matrix system:

```
subplot(2,1,1); spy(str.K);subplot(2,1,2);spy(str.N)
Q = sum(str.L)
```

Get the eliminated matrix system, use the initial solution as linearization point:

```
str = mphmatrix(model,'sol1','out',{'Kc'},'initmethod','init');
```

Compare the sparsity between the eliminated and non-eliminated stiffness matrix:

```
subplot(2,1,1); hold on; spy(str.Kc,'r')
```

Evaluate the eliminated load vector using the current solution as linearization point:

```
str = mphmatrix(model,'sol1','out',{'Lc'},'initmethod','sol');
```

Evaluate the system matrices of a dynamic problem

```
model = mphopen('model_tutorial_llmatlab');
model.mesh('mesh1').autoMeshSize(8);
```

```
std = model.study.create('std1');
time = std.feature.create('time', 'Transient');
time.set('tlist', 'range(0,1,25)');
model.param.set('timestep', '1[s]');
std.run;
```

Get the dynamic matrix system:

```
str = mphmatrix(model,'sol1','out',{'E','D','K','L','M','N'});
```

Display the sparsity of the mass and stiffness matrices:

```
subplot(1,2,1); spy(str.D); subplot(1,2,2); spy(str.K);
```

Get the eliminated dynamic matrix system:

```
str = mphmatrix(model,'sol1','out',{'Ec','Dc','Kc','Lc','M','N'});
```

Assemble the Jacobian using solution number 15 as linearization point. First run the model to get available linearization point list:

```
std.run;
str = mphmatrix(model,'sol1','out',{'K'},...
   'initmethod','sol','initsol','sol1','solnum',15);
```

Assemble the Jacobian using the zero vector as linearization point:

```
str = mphmatrix(model,'sol1','out',{'K'},...
   'initmethod','sol','initsol','zero');
```

SEE ALSO

`mphstate`, `mphxmeshinfo`, `mphinputmatrix`

mphmax

Perform a maximum of expressions.

SYNTAX

```
[v1,...,vn] = mphmax(model,{e1,...,en},edim,...)
[v1,...,vn,unit] = mphmax(model,{e1,...,en},edim,...)
```

DESCRIPTION

`[v1,...,vn] = mphmax(model,{e1,...,en},edim,...)` evaluates the maximum of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphmax(model,{e1,...,en},edim,...)` also returns the units of the maximum in a 1xN cell array.

The function `mphmax` accepts the following property/value pairs:

TABLE 6-20: PROPERTY/VALUE PAIRS FOR THE MPHMAX COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`dataseries`	`none` | `average` | `integral` | `maximum` | `minimum` | `rms` | `stddev` | `variance`	`none`	Data series operation
`dataset`	String	Active solution dataset	Dataset tag
`matrix`	`on` | `off`	`on`	Returns data as a matrix or as a cell
`outersolnum`	Positive integer array | `all` | `end`	`1`	Solution number for parametric sweep
`position`	`on`|`off`	`off`	Extract position as well as value
`selection`	Integer vector | string | `all`	`all`	Selection list or named selection
`solnum`	Integer vector | `end` | `all`	`all`	Solution for evaluation
`squeeze`	`on` | `off`	`on`	Squeeze singleton dimensions
`t`	Double array		Time for evaluation

The property `dataset` controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution datasets.

The maximum expressions `e1,...,en` is evaluated for one or several solutions. Each solution generates an additional column in the returned matrix. The properties `solnum` and `t` control which solutions are used for the evaluation. The `solnum` property is available when the dataset has multiple solutions — for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `solnum` is provided, the solutions indicated by the indices provided with the `solnum` property are used. If `t` is provided, solutions are interpolated. If neither `solnum` nor `t` is provided, all solutions are evaluated.

The property `solnum` is used to select the solution number when a parametric, eigenvalue, or time-dependent solver has been used.

The property `outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

If the `matrix` property is set to `off` the output is cell arrays of length P containing cell arrays of length M.

EXAMPLE

Evaluate the maximum temperature in the model domain:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

maxT = mphmax(model,'T','volume');
```

Evaluate the maximum temperature on boundary 9:

```
maxT = mphmax(model,'T','surface','selection',9);
```

Evaluate maximum of expression using several solution:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;
```

Evaluate the maximum of the temperature at every time step computed with power set to 30:

```
maxT = mphmax(model,'T','volume','dataset','dset2');
```

Evaluate the maximum of the temperature at the fifth time step:

```
maxT = mphmax(model,'T','volume','dataset','dset2',...
   'solnum',5);
```

Evaluate the maximum of the temperature at 10.5 sec and 15.2 sec:

```
maxT = mphmax(model,'T','volume','dataset','dset2',...
   't',[10.5,15.2]);
```

Evaluate the maximum of the temperature at every time step computed with power set to 90:

```
maxT = mphmax(model,'T','volume','dataset','dset2',....
   'outersolnum',3);
```

SEE ALSO

mphmean, mphmin

mphmean

Perform a mean of expressions.

SYNTAX

```
[v1,...,vn] = mphmean(model,{e1,...,en},edim,...)
[v1,...,vn,unit] = mphmean(model,{e1,...,en},edim,...)
```

DESCRIPTION

`[v1,...,vn] = mphmean(model,{e1,...,en},edim,...)` evaluates the means of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphmean(model,{e1,...,en},edim,...)` also returns the units of the maximum in a 1xN cell array.

The function `mphmean` accepts the following property/value pairs:

TABLE 6-21: PROPERTY/VALUE PAIRS FOR THE MPHMEAN COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`dataseries`	`none` | `average` | `integral` | `maximum` | `minimum` | `rms` | `stddev` | `variance`	`none`	Data series operation
`dataset`	String	Active solution dataset	Dataset tag
`intorder`	Positive integer	4	Integration order
`matrix`	`off` | `on`	`on`	Returns data as a matrix or as a cell
`method`	`auto` | `integration` | `summation`	`auto`	Integration method
`outersolnum`	Positive integer array | `all` | `end`	1	Solution number for parametric sweep

TABLE 6-21: PROPERTY/VALUE PAIRS FOR THE MPHMEAN COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
selection	Integer vector | string | all	all	Selection list or named selection
solnum	Integer vector | end | all	all	Solution for evaluation
squeeze	on | off	on	Squeeze singleton dimensions
t	Double array		Time for evaluation

The property dataset controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution datasets.

The mean of expressions e1,...,en is evaluated for one or several solutions. Each solution generates an additional column in the returned matrix. The properties solnum and t control which solutions are used for the evaluation. The solnum property is available when the dataset has multiple solutions — for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The t property is available only for time-dependent problems. If solnum is provided, the solutions indicated by the indices provided with the solnum property are used. If t is provided, solutions are interpolated. If neither solnum nor t is provided, all solutions are evaluated.

The property solnum is used to select the solution number when a parametric, eigenvalue, or time-dependent solver has been used.

The property outersolnum is used to select the outer solution number when a parametric sweep has been used in the study step node.

If the matrix property is set to off the output is cell arrays of length P containing cell arrays of length M.

EXAMPLE

Evaluate the mean temperature in the model domain:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

maxT = mphmean(model,'T','volume');
```

Evaluate the mean temperature on boundary 9:

```
maxT = mphmean(model,'T','surface','selection',9);
```

Evaluate mean of expression using several solution:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;
```

Evaluate the mean of the temperature at every time step computed with power set to 30:

```
maxT = mphmean(model,'T','volume','dataset','dset2');
```

Evaluate the mean of the temperature at the fifth time step:

```
maxT = mphmean(model,'T','volume','dataset','dset2',...
   'solnum',5);
```

Evaluate the mean of the temperature at 10.5 sec and 15.2 sec:

```
maxT = mphmean(model,'T','volume','dataset','dset2',...
   't',[10.5,15.2]);
```

Evaluate the mean of the temperature at every time step computed with power set to 90:

```
maxT = mphmean(model,'T','volume','dataset','dset2',....
'outersolnum',3);
```

SEE ALSO

`mphmax`, `mphmin`

mphmeasure

Measure entities in geometry

SYNTAX

```
[measure, misc] = mphmeasure(model,geomtag,entity,...)
```

DESCRIPTION

`[measure, misc] = mphmeasure(model,geomtag,entity,...)` measure the entity with the type `entity` in the geometry defined with the tag `geomtag`. `entity` can be one of `'point'`, `'edge'`, `'boundary'`, or `'domain'`.

The function `mphmeasure` accepts the following property/value pairs:

TABLE 6-22: PROPERTY/VALUE PAIRS FOR THE MPHMEASURE COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Build`	`on` | `off` | string	`on`	Build the geometry before plotting
`Objects`	Cell array		Selection matrix
`Selection`	Positive integer array	`all`	Selection (finalized geometry)
`Usefinal`	`on` | `off`	`on`	Measure on the finalized geometry

EXAMPLE

Get the volume of all domains in the geometry:

```
model = mphopen('busbar');
vm = mphmeasure(model,'geom1','domain')
```

Get the volume and the surface area of all domains in the geometry:

```
[vm, am]= mphmeasure(model,'geom1','domain')
```

Get midpoint and distance between points 2 and 4:

```
[m, dist] = mphmeasure(model,'geom1','point','selection',[2,4])
```

Length of edges bounded by vertex 1, 29, 41 and 42:

```
[~, a] = mphgetadj(model,'geom1','edge','point',[1,29,41,42])
mphviewselection(model,'geom1',[1,29,41,42],'point')
hold on
mphviewselection(model,'geom1',a,'edge','edgecolorselected','b')
m = mphmeasure(model,'geom1','edge','selection',a)
```

mphmesh

Plot a mesh in a MATLAB® figure window.

SYNTAX

```
mphmesh(model)
mphmesh(model,meshtag,...)
pd=mphmesh(model,meshtag,...)
```

DESCRIPTION

`mphmesh(model)` plots the mesh in a MATLAB figure.

`mphmesh(model,meshtag,...)` plots the mesh `meshtag` in a MATLAB figure. If there is only one mesh in the model the `meshtag` can be left empty.

The function `mphmesh` accepts the following property/value pairs:

TABLE 6-23: PROPERTY/VALUE PAIRS FOR THE MPHMESH COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Parent`	Double		Parent axis
`Edgecolor`	Char	`k`	Edge color
`Edgelabels`	`on` | `off`	`off`	Show edge labels
`Edgelabelscolor`	Char	`k`	Color for edge labels
`Edgemode`	`on` | `off`	`on`	Show edges
`Facealpha`	Double	`1`	Set transparency value
`Facelabels`	`on` | `off`	`off`	Show face labels
`Facelabelscolor`	Char	`k`	Color for face labels
`Facemode`	`on` | `off`	`on`	Show faces
`Meshcolor`	Char | vector	`gray`	Color for face element
`Vertexcolor`	Char | vector	`k`	Color for vertices
`Vertexlabels`	`on` | `off`	`off`	Show vertex labels
`Vertexlabelscolor`	Char | vector	`k`	Color for vertex labels
`Vertexmode`	`on` | `off`	`off`	Show vertices
`View`	String | `'auto'` | `''`		View settings

EXAMPLE

Plot the mesh

```
model = mphopen('model_tutorial_llmatlab');
model.component('comp1').mesh.run;
mphmesh(model)
```

Create a second mesh with an “extra fine” default mesh settings and plot it:

```
mesh = model.component('comp1').create('mesh2');
mesh.autoMeshSize(2);
mesh.run;

mphmesh(model,'mesh2','meshcolor','r');
```

The mesh can be plotted with view settings applied. This results in a mesh with grid, axes labels, lights, hiding etc. applied to the plot. Usually it is sufficient to use the `auto` setting, but any valid view can be applied:

```
mphmesh(model, 'mesh1', 'view', 'auto')
```

Plot data can be returned from `mphmesh`. This can be used to create the plot later or to extract information used to create the plot for further analysis

```
pd = mphmesh(model, 'mesh1');
mphplot(pd)
```

SEE ALSO

`mphgeom`, `mphmeshstats`, `mphplot`

mphmeshstats

Return mesh statistics and mesh data information.

SYNTAX

```
stats = mphmeshstats(model)
stats = mphmeshstats(model, meshtag, ...)
[stats,data] = mphmeshstats(model, meshtag, ...)
```

DESCRIPTION

`stats = mphmeshstats(model)` returns mesh statistics of the model mesh in the structure `str`.

`stats = mphmeshstats(model, meshtag, ...)` returns mesh statistics of a mesh `meshtag` in the structure `str`.

`[stats,data] = mphmeshstats(model, meshtag, ...)` returns in addition the mesh data information such as vertex coordinates and definitions of elements in the structure `data`.

The function `mphmeshstats` accepts the following property/value pairs:

TABLE 6-24: PROPERTY/VALUE PAIRS FOR THE MPHMESHSTATS COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`entity`*	`domain` | `boundary` | `edge` | `point`		Selected entity type
`qualityhistogram`	Integer	20	Number of bins in the quality distribution histogram

TABLE 6-24: PROPERTY/VALUE PAIRS FOR THE MPHMESHSTATS COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
qualitymeasure	condition | growth | maxangle | skewness | volcircum | vollength	volcircum	Quality measure
selection	String | Positive integer array		Selection tag or entity number
type*	ctx | edg | tri | quad | tet | pyr | prism | hex		Restrict statistics to element types. Can also be a cell array.

* Selection and Entity properties cannot be set if the data structure is returned.

The output structure stats contains the following fields:

TABLE 6-25: FIELDS IN THE STATS STRUCTURE

FIELD	DESCRIPTION
meshtag	Mesh tag
geomtag	Geometry tag
geometricmodel	Geometric model used by the mesh sequence
component	Component tag
componentgeometricmodel	Geometric model used by the physics
current	Current mesh feature tag
isempty	Is the mesh empty?
hasproblems	Does the mesh have problems?
iscomplete	Is the mesh built to completion?
secondorderelements	Does the mesh have second-order elements?
sdim	Space dimension
contributing	Contributing physics or multiphysics interface for the physics-controlled mesh
types	Element types present in the mesh
numelem	Number of elements for each mesh type
qualitymeasure	Quality measure
minquality	Minimum quality
meanquality	Mean quality

TABLE 6-25: FIELDS IN THE STATS STRUCTURE

FIELD	DESCRIPTION
qualitydistr	Quality distribution
minvolume	Volume/area/length of the smallest element
maxvolume	Volume/area/length of the largest element
volume	Volume/area/length of the mesh
maxgrowthrate	Maximum growth rate*
meangrowthrate	Mean growth rate*

* Provides statistics for the entire selection regardless of the element type property.

The output structure `data` contains the following fields:

TABLE 6-26: FIELDS IN THE DATA STRUCTURE

FIELD	DESCRIPTION
vertex	Coordinates of mesh vertices
elem	Cell array of definition of each element type
elementity	Entity information for each element type

EXAMPLE

Get the mesh statistics:

```
model = mphopen('model_tutorial_llmatlab');
model.component('comp1').mesh.run;

stats = mphmeshstats(model)
```

Show the mesh quality distribution in a figure:

```
bar(linspace(0,1,20),stats.qualitydistr)
```

Get the mesh statistics and the mesh data:

```
[stats,data] = mphmeshstats(model);
```

Show the element vertices in a plot:

```
plot3(data.vertex(1,:), data.vertex(2,:), data.vertex(3,:), '.')
axis equal
```

Get the number of edge element:

```
numedgeelem = stats.numelem(strcmp(stats.types,'edg'))
```

SEE ALSO

mphmesh

mphmin

Perform a minimum of expressions.

SYNTAX

```
[v1,...,vn] = mphmin(model,{e1,...,en},edim,...)
[v1,...,vn,unit] = mphmin(model,{e1,...,en},edim,...)
```

DESCRIPTION

`[v1,...,vn] = mphmin(model,{e1,...,en},edim,...)` evaluates the minimum of the string expressions `e1,...,en` and returns the result in N matrices `v1,...,vn` with M rows and P columns. M is the number of inner solution and P the number of outer solution used for the evaluation. `edim` defines the element dimension: `line`, `surface`, `volume` or as an integer value.

`[v1,...,vn] = mphmin(model,{e1,...,en},edim,...)` also returns the units in a 1xN cell array.

The function `mphmin` accepts the following property/value pairs:

TABLE 6-27: PROPERTY/VALUE PAIRS FOR THE MPHMIN COMMAND.

PROPERTY	PROPERTY VALUE	DEFAULT	DESCRIPTION
`dataseries`	`none` | `average` | `integral` | `maximum` | `minimum` | `rms` | `stddev` | `variance`	`none`	Data series operation
`dataset`	String	Active solution dataset	Dataset tag
`matrix`	`off` | `on`	`on`	Returns data as a matrix or as a cell
`outersolnum`	Positive integer array | `all` | `end`	`1`	Solution number for parametric sweep
`position`	`on`|`off`	`off`	Extract position as well as value
`selection`	Integer vector | string | `all`	`all`	Selection list or named selection
`solnum`	Integer vector | `end` | `all`	`all`	Solution for evaluation
`squeeze`	`on` | `off`	`on`	Squeeze singleton dimensions
`t`	Double array		Time for evaluation

The property `dataset` controls which dataset is used for the evaluation. Datasets contain or refer to the source of data for postprocessing purposes. Evaluation is supported only on Solution datasets.

The mean of expressions `e1,...,en` is evaluated for one or several solutions. Each solution generates an additional column in the returned matrix. The properties `solnum` and `t` control which solutions are used for the evaluation. The `solnum` property is available when the dataset has multiple solutions — for example, in the case of parametric, eigenfrequency, or time-dependent solutions. The `t` property is available only for time-dependent problems. If `solnum` is provided, the solutions indicated by the indices provided with the `solnum` property are used. If `t` is provided, solutions are interpolated. If neither `solnum` nor `t` is provided, all solutions are evaluated.

The property `solnum` is used to select the solution number when a parametric, eigenvalue, or time-dependent solver has been used.

The property `outersolnum` is used to select the outer solution number when a parametric sweep has been used in the study step node.

If the `matrix` property is set to `off` the output is cell arrays of length P containing cell arrays of length M.

EXAMPLE

Evaluate the minimum temperature in the model domain:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

maxT = mphmin(model,'T','volume');
```

Evaluate the minimum temperature on boundary 9:

```
maxT = mphmin(model,'T','surface','selection',9);
```

Evaluate minimum of expression using several solution:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;
```

Evaluate the minimum of the temperature at every time step computed with power set to 30:

```
maxT = mphmin(model,'T','volume','dataset','dset2');
```

Evaluate the minimum of the temperature at the fifth time step:

```
maxT = mphmin(model,'T','volume','dataset','dset2',...
   'solnum',5);
```

Evaluate the minimum of the temperature at 10.5 sec and 15.2 sec:

```
maxT = mphmin(model,'T','volume','dataset','dset2',...
   't',[10.5,15.2]);
```

Evaluate the minimum of the temperature at every time step computed with power set to 90:

```
maxT = mphmin(model,'T','volume','dataset','dset2',....
'outersolnum',3);
```

SEE ALSO

`mphmax`, `mphmean`

mphmodel

Return tags for the nodes and subnodes in the COMSOL model object.

SYNTAX

```
mphmodel(model)
str = mphmodel(model,'-struct')
```

DESCRIPTION

`mphmodel(model)` returns the tags for the nodes and subnodes of the object `model`.

`str = mphmodel(model,'-struct')` returns the tags for the nodes and subnodes of the object `model` as a MATLAB® structure `str`.

The function `mphmodel` can be used when navigating the model object and learning about its structure. The `mphmodel` function is mainly designed for usage when working on the command line and one needs to learn what nodes are placed under a particular node.

EXAMPLE

Load the model `busbar.mph` and get the list of the nodes available under the root node:

```
model = mphopen('busbar')
mphmodel(model)
```

Get the model information as a structure:

```
res = mphmodel(model, '-struct')
```

SEE ALSO

mphgetexpressions, mphgetproperties, mphgetselection, mphnavigator, mphsearch, mphshowerrors

mphnavigator

Graphical user interface (GUI) for viewing the COMSOL Multiphysics model object.

SYNTAX

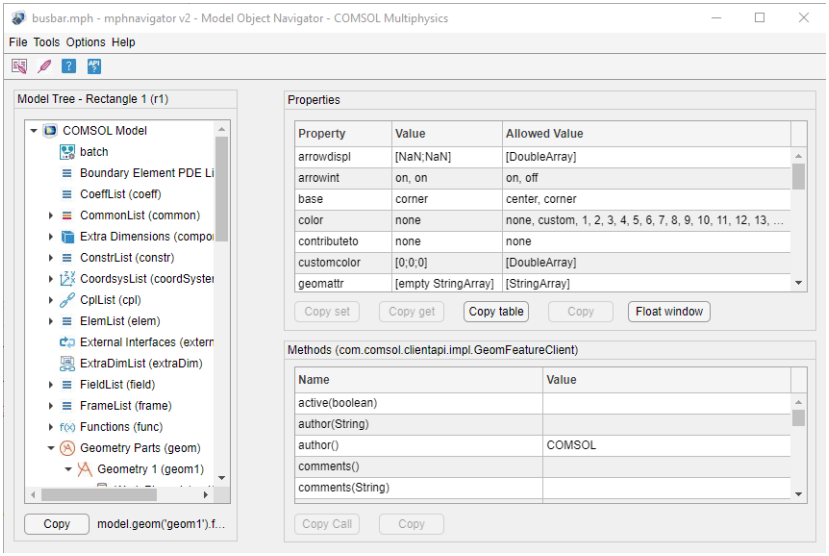
```
mphnavigator
mphnavigator(model)
```

DESCRIPTION

`mphnavigator` opens the Model Object Navigator which is a graphical user interface that can be used to navigate the model object and to view the properties and methods of the nodes in the model tree.

The GUI requires that the COMSOL object is stored in a variable in the base workspace (at the MATLAB® command prompt) with the name `model`.

`mphnavigator(model)` opens the model object defined with the name `model` in Model Object Navigator window.



EXAMPLE

Load `busbar.mph` from the Model Library:

```
mphopen busbar
```

Navigate the model object that is accessible with the variable `model`

```
mphnavigator
```

Load `effective_diffusivity.mph` from the Applications Libraries and set the model object with the variable `eff_diff`:

```
eff_diff = mphopen('effective_diffusivity');
```

Navigate the model object that is accessible with the variable `eff_diff`

```
mphnavigator(eff_diff)
```

SEE ALSO

`mphgetexpressions`, `mphgetproperties`, `mphgetselection`, `mphmodel`, `mphsearch`, `mphshowerrors`

mphopen

Graphical user interface (GUI) to open recent model files.

SYNTAX

```
mphopen
mphopen -dir dirpath
mphopen -clear
model = mphopen(filename)
model = mphopen(filename, mtag)
model = mphopen(filename, mtag, '-nostore')
model = mphopen(filename, mtag, '-history')
model = mphopen(filename, mtag, pwd)
[model, filenameloaded] = mphopen(filename,...)
```

DESCRIPTION

`mphopen` starts a GUI with the recent opened files list.

`mphopen -dir dirpath` starts a GUI with a list of the files in the specified directory `dirpath`. If `dirpath` is not specified the working directory is taken by default.

`mphopen -clear` resets the recent opened files list.

`model = mphopen(filename)` loads a COMSOL model object saved with the name `filename` and assigns the default tag `Model` in the COMSOL *server*. If a model with tag `Model` already exists and is also open in a COMSOL Multiphysics client, the loaded model an index number is appended to the tag, for instance `Model1`.

`model = mphopen(filename, mtag)` loads a COMSOL model object and assigns the tag `mtag` in the COMSOL *server*.

`model = mphopen(filename, mtag, '-nostore')` does not update the recent opened model list.

`model = mphopen(filename, mtag, '-history')` turns on history recording.

`model = mphopen(filename, mtag, pwd)` loads the COMSOL model in the file protected with the password `pwd`.

`[model, filenameloaded] = mphopen(filename, …)` also returns the full file name `filenameloaded` of the file that was loaded.

The model tag `mtag` and the password `pwd` are defined as string.

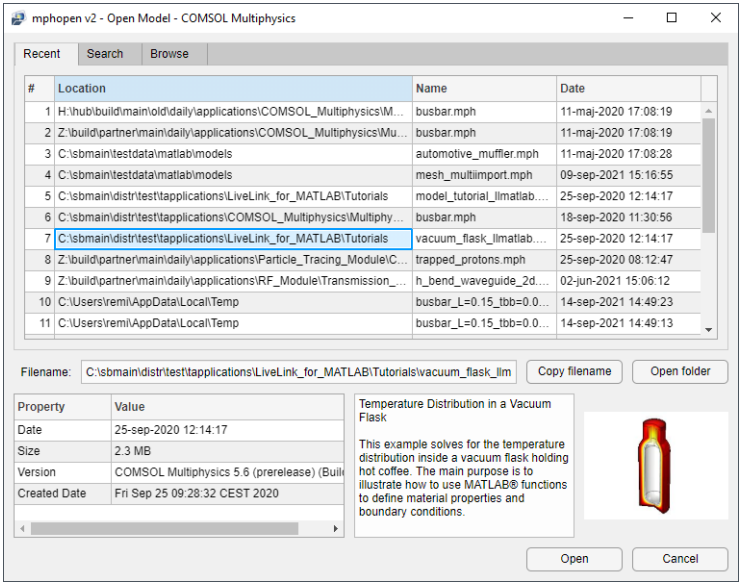
If the model tag is the same as a model that is currently in the COMSOL *server* the loaded model overwrites the existing one.

Note that MATLAB® searches for the model on the MATLAB path if an absolute path is not supplied.

`mphopen` turns off the model history recording by default, unless the property `'-history'` is used.

The extension `mph` can be omitted.

`mphopen` does not look for a lock file when opening a model in the COMSOL *server*.



SEE ALSO

`mphload`, `mphsave`

mphparticle

Evaluate expressions on particle and ray trajectories.

SYNTAX

```
pd = mphparticle(model)
pd = mphparticle(model,'expr',{e1,...,en},...)
```

DESCRIPTION

`mphparticle(model)` returns particle position and particle velocity at all time steps stored in the first particle dataset.

`mphparticle(model,'expr',{e1,...,en},...)` returns particle position, particle velocity and expressions `e1,..., en` evaluated on particle trajectories.

The function `mphparticle` accepts the following property/value pairs:

TABLE 6-28: PROPERTY/VALUE PAIRS FOR THE MPHPARTICLE COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`expr`	String | Cell array		Expressions to evaluate
`dataonly`	`on` | `off`	`off`	Return only expression values
`dataset`	String	First particle dataset	Dataset tag
`t`	Double array		Time for evaluation

The returned value `pd` is a structure with the following content

TABLE 6-29: FIELDS IN THE INFO STRUCTURE

FIELD	CONTENT
`v`	Velocity of the particles
`p`	Position of the particles
`d#`	Result of evaluation #
`t`	Time for evaluation
`expr`	Evaluated expressions
`unit`	Unit of evaluations

Note: `mphparticle` only evaluates expressions using particle and ray datasets.

EXAMPLE

Load the model `trapped_protons` from the Applications Libraries:

```
model = mphopen('trapped_protons');
```

Extract the particle positions and particle velocities along the computed trajectories at every time steps stored in the model:

```
pd = mphparticle(model)
```

Evaluate the mirror point latitude (Lm) and the particle equatorial pitch angle (Ea) at t = 0.7 sec., extract only the data:

```
pd = mphparticle(model,'dataset','dset2',...
```

```
'expr',{'Lm','Ea*180/pi'},'t',0.7,'dataonly','on')
```

SEE ALSO

`mpheval`, `mphevalpoint`, `mphint2`, `mphinterp`, `mphray`

mphplot

Render a plot group in a figure window.

SYNTAX

```
mphplot(model)
mphplot(model,pgtag,...)
pd = mphplot(model,pgtag,...)
mphplot(pd,...)
```

DESCRIPTION

`mphplot(model)` opens a figure window and adds a menu where it is possible to switch between all the different result plots in a model as well as any geometry and mesh plots. A toolbar is added to the figure that allows the user to control the use of views, lights, and camera settings.

`mphplot(model,pgtag,...)` renders the plot group tagged `pgtag` from the model object `model` in a figure window in MATLAB®.

`pd = mphplot(model,pgtag,...)` also returns the plot data used in the MATLAB figure in a cell array `pd`. `pd` contains ordinary MATLAB data and can later be used to recreate the plot using `mphplot`. It is also possible to investigate and extract data from the plot this way in order to create plots in other ways or to further analyze the data. Set `createplot` to `off` if a plot should be not be created. Note that pd contains data in single precision even though all calculations in COMSOL Multiphysics are carried out in double precision.

`mphplot(pd,...)` makes a plot using the post data structure `pd` that is generated using the function `mpheval`. Plots involving points, lines and surfaces are supported.

The function `mphplot` accepts the following property/value pairs:

TABLE 6-30: PROPERTY/VALUE PAIRS FOR THE MPHPLOT COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`colortable`	char	`rainbow`	Color table used for plotting post data structure
`createplot`	`on` | `off`	`on`	Create a plot

TABLE 6-30: PROPERTY/VALUE PAIRS FOR THE MPHPLOT COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
index	Positive integer	1	Index of variable to use for plotting post data structure
mesh	on | off	on	Plot the mesh when using post data structure
parent	Double		Set the parent axes
rangenum	Positive integer	none	Color range bar (or legend) to display
server	on | off	off	Plot on server
view	char | 'auto' | ''	''	View settings tag

Note: The plot on server option requires that you start COMSOL with MATLAB in graphics mode.

Only one color range bar and one legend bar is supported in a MATLAB figure. When the option plot on server is active, all active color range bar are displayed.

The property `createplot` is useful when extracting plot data structure on machines without a graphics display.

The data fields returned by `mphplot` are subject to change. The most important fields are:

- `p`, the coordinates for each point that are used for creating lines or triangles.
- `n`, the normals in each point for the surfaces. These are not always available.
- `t`, contains the indices to columns in `p` of a simplex mesh, each column in `t` representing a simplex.
- `d`, the data values for each point.
- `rgb`, the color values (red, green and blue) entities at each point.

EXAMPLE

Display the plot settings pg using a MATLAB figure

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;
model.result.dataset.create('mir', 'Mirror3D');
```

```
pg = model.result.create('pg', 'PlotGroup3D');
pg.set('data', 'mir');
surf1 = pg.feature.create('surf1', 'Surface');
surf1.set('colortable', 'Thermal');

mphplot(model,'pg')
```

Combine plot types on the same plot group:

```
surf2 = pg.feature.create('surf2', 'Surface');
surf2.set('data', 'dset1');
surf2.set('expr', 'ht.tfluxMag');
```

Display the plot group and the color range bar of the second plot type:

```
mphplot(model,'pg','rangenum',2)
```

Display the plot group on the server:

```
mphplot(model,'pg','server','on')
```

Display expression value evaluated using `mpheval`:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;
```

Extract temperature and total heat flux magnitude in domain 2:

```
pd = mpheval(model,{'T','ht.tfluxMag'},'selection',2);
```

Plot the temperature data using thermal color table:

```
mphplot(pd,'index',1,'colortable','Thermal','rangenum',1)
```

SEE ALSO

`colortable`, `mpheval`

mphquad2tri

Convert plot data quad mesh into simplex mesh.

SYNTAX

```
pdout = mphquad2tri(pdin)
```

DESCRIPTION

`pdout = mphquad2tri(pdin)` converts the plot data stored in the structure `pdin` into the structure `pdout` using a simplex mesh.

The input and output structures, respectively `pdin` and `pdout`, are structures with fields `p`, `d`, `t`, `rgb` and `expr`.

- The field `p` is a 2xN array containing the vertex coordinates.
- The field saved is a Nx1 array containing the data value at the vertices.
- The field `t` is an array containing the indices to columns in `p` of a quad mesh for `pdin` and of a simplex mesh for `pdout`, each column in `t` representing respectively a quad and a simplex.
- The field `rgb` is a Nx3 array containing the color model data as RGB values. This field is optional.
- The field `expr` is a string containing the description of the data. This field is optional.

EXAMPLE

Generate 3D surf data

```
[x,y] = meshgrid(-0.1:0.2:1.1,-0.4:0.2:0.4);
z = cumsum(0.1*randn(size(x)));
```

Create plotdata in a quad mesh:

```
pd = mphsurf(x,y,z);
```

Convert the plot data into triangle mesh:

```
pd = mphquad2tri(pd);
mphplot(pd)
```

SEE ALSO

`mphsurf`, `mphreadstl`, `mphwritestl`

mphray

Evaluate expressions on particle and ray trajectories.

SYNTAX

```
pd = mphray(model)
pd = mphray(model,'expr',{e1,...,en},...)
```

DESCRIPTION

`pd = mphray(model)` returns particle position and particle velocity at all time steps stored in the first particle dataset.

`pd = mphray(model,'expr',{e1,...,en},...)` returns particle position, particle velocity and expressions `e1,...,en` evaluated on particle or ray trajectories.

`pd` is a structure with fields `p`, `v`, `d#`, `t`, `expr`, and `unit`.

- The field `p` contains the position of the particles.
- The field `v` contains the velocity of the particles.
- The field `d#` contains the result of evaluation #.
- The field `t` contains the time for evaluation.
- The field `expr` contains the evaluated expressions.
- The field `unit` contains the unit of evaluations.

The function `mphray` accepts the following property/value pairs:

TABLE 6-31: PROPERTY/VALUE PAIRS FOR THE MPHRAY COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`dataonly`	`on | off`	`off`	Return only expression values
`dataset`	`String`		Dataset tag
`expr`	`String | Cell array`		Expressions to evaluate
`outersolnum`	`Positive integer`	`1`	Outer solution for evaluation
`solnum`	`Positive integer array | all | end`	`all`	Inner solution for evaluation
`t`	`Double array`		Time for evaluation
`times`	`on | off`	`on`	Return times
`velocities`	`on | off`	`on`	Return velocities

Note: `mphray` only evaluates expressions using particle and ray datasets.

SEE ALSO

`mpheval`, `mphevalpoint`, `mphevalpoint`, `mphint2`, `mphinterp`, `mphparticle`

mphreadstl

Read an STL file and return the data into a plot data structure.

SYNTAX

```
pd = mphreadstl(filename)
```

DESCRIPTION

`pd = mphreadstl(filename)` reads the STL file `filename` and returns the data into the plot data structure `pd`.

`pd` is a structure with fields `n`, `p`, `t`, `name`, `ref`, `rgb`, `expr`, and `d1`.

- The field `n` contains the normal vector for the triangle.
- The field `p` contains node point coordinate information.
- The field `t` contains the indices to columns in `p` of a simplex mesh, each column in `t` representing a simplex.
- The field `name` contains the name of the file the data come from.
- The field `ref` contains the header of the file.
- The field `rgb` contains the color model data at each vertices. The columns correspond to node point coordinates in columns in `p`.
- The field `expr` contains the data description.
- The field `d1` contains the data value at each vertices. The columns correspond to node point coordinates in columns in `p`.

Both binary and ASCII STL files are supported.

SEE ALSO

`mphquad2tri`, `mphsurf`, `mphwritestl`

mphreduction

Return reduced-order state-space matrices for a model.

SYNTAX

```
data = mphreduction(model, romtag, ...)
```

DESCRIPTION

`data = mphreduction(model, romtag, ...)` extracts the reduced order state-space matrices from the reduced order model `romtag`. The reduced order model can either be created in the COMSOL GUI or via the API.

The function `mphreduction` accepts the following property/value pairs:

TABLE 6-32: PROPERTY/VALUE PAIRS FOR THE MPHREDUCTION COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`out`	Cell array of strings	`'all'`	Names of output matrices
`return`	`struct` | `ss`	`struct`	Return type

The following values are valid for the `out` property:

TABLE 6-33: PROPERTY/VALUE PAIRS FOR THE PROPERTY OUT.

PROPERTY	EXPRESSION	DESCRIPTION
`out`	`Kr`	Stiffness matrix
	`Dr`	Damping matrix
	`Dra`	Damping ratio matrix
	`Er`	Mass matrix
	`Br`	Input matrix
	`Cr`	Output matrix
	`F`	Input feedback matrix
	`BOr`	Initial value input matrix
	`BOrdot`	Initial value time derivative input matrix
	`Brdot`	Time derivative input matrix
	`Brdotdot`	Second time derivative input matrix
	`Mc`	Damping matrix
	`MA | A`	Stiffness matrix
	`MB | B`	Input matrix
	`D`	Input feedback matrix
	`C`	Output matrix
	`L`	Load vector
	`YO`	Output bias
	`UO`	Initial value vector
	`UdotO`	Initial derivative vector
	`Kud`	Stiffness matrix times ud

The return type ss requires that the Control System Toolbox is installed.

SEE ALSO

`mphstate`

mphreport

Generate report to model or write report.

SYNTAX

```
mphreport(model,...)
mphreport(model,'action','run','tag',rpttag,...)
```

DESCRIPTION

`mphreport(model,...)` generate report to the model `model`.

`mphreport(model,'action','run','tag',rpttag,...)` write report defined with the tag `rpttag`.

The function `mphreport` accepts the following property/value pairs:

TABLE 6-34: PROPERTY/VALUE PAIRS FOR THE MPHREPORT COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`action`	`add|run`	`add`	Add or run report
`filename`	`String`		Report filename
`format`	`html | docx | pptx`	`html`	Output format
`open`	`on | off`	`on`	Open report when finished
`tag`	`String`		Report tag
`template`	`String`		User template
`type`	`brief | intermediate | complete`	`intermediate`	Report type

EXAMPLE

Generate a brief report set with the tag `'myreport'`:

```
mphreport(model,'tag','myreport','type','brief')
```

Write report set with the tag `'myreport'` to the Microsoft Word document `'modelreport.docx'`:

```
mphreport(model,'action','run','tag','myreport',...
   'filename','modelreport','format','docx')
```

Do not open the report once written:

```
mphreport(model,'action','run','tag','myreport',...
   'open','off')
```

mphsave

Save a COMSOL Multiphysics model.

SYNTAX

```
mphsave(model)
mphsave(model,filename,...)
mphsave(filename)
mphsave(mtag)
mphsave(mtag,filename,...)
```

DESCRIPTION

`mphsave(model)` saves the COMSOL model object `model`.

`mphsave(model,filename,...)` saves the COMSOL model object `model` to the file named `filename`.

`mphsave(filename)` saves the unique COMSOL model that is loaded in the COMSOL server to the file named `filename`.

`mphsave(mtag)` saves the COMSOL model loaded in the COMSOL server with the tag `mtag`.

`mphsave(mtag,filename,...)` aves the COMSOL model loaded in the COMSOL server with the tag `mtag` to the file named `filename`.

The function `mphsave` accepts the following property/value pairs:

TABLE 6-35: PROPERTY/VALUE PAIRS FOR THE MPHSAVE COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`component`	`on | off`	`off`	Save M-file using the component syntax
`copy`	`on | off`	`off`	Save a copy of the model
`description`	`String`		Set or append model description
`excludedata`	`on | off`		Exclude built, computed, an plotted data
`filenameis`	`fullpath | name | path`	`fullpath`	Use filename as the selected type
`optimize`	`size | speed`		Optimize for speed or file size
`store`	`on | off`	`on`	Store the filename in most recently used files

If the file name is not provided, the model has to be saved previously on disk.

If the file name does not provide a path, the file is saved relatively to the current path in MATLAB®.

The model can be saved as an MPH-file, Java file, or M-file. The file extension determines which format that is saved.

Note: Model created with older version than COMSOL 5.3 cannot be saved using the component syntax.

SEE ALSO

`mphopen`, `mphload`

mphsearch

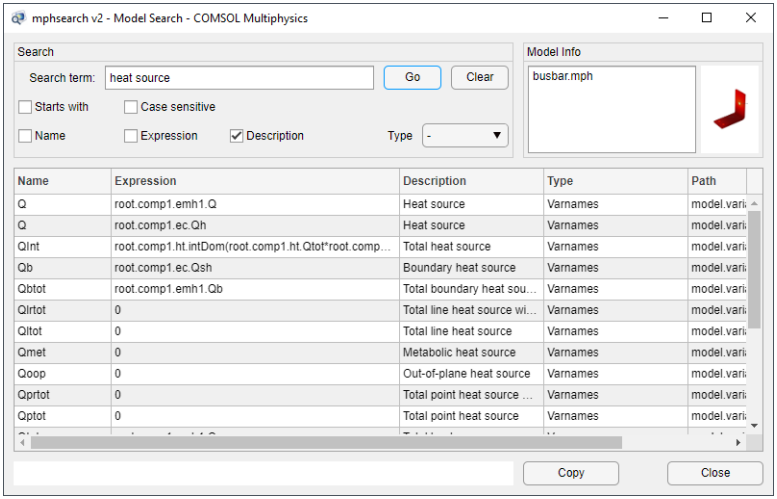
Graphical user interface (GUI) for searching expressions in the COMSOL model object.

SYNTAX

```
mphsearch(model)
```

DESCRIPTION

`mphsearch(model)` opens a graphical user interface that can be used to search expressions in the model object `model`. Search using a text available in the name, expression or description of the variable.



SEE ALSO

`mphgetexpressions`, `mphgetproperties`, `mphgetselection`, `mphmodel`, `mphnavigator`

mphselectbox

Select geometric entity using a rubberband/box.

SYNTAX

```
n = mphselectbox(model,geomtag,boxcoord,entity,...)
```

DESCRIPTION

`n = mphselectbox(model,geomtag,boxcoord,entity,...)` returns the indices of the geometry entities that are inside the rubberband domain (rectangle or box). This method looks only on the vertex coordinates and does not observe all points on curves and surfaces.

`boxcoord` set the coordinates of the selection domain, specified as a Nx2 array, where N is the geometry space dimension.

`entity` can be one of `point`, `edge`, `boundary`, or `domain` following the entity space dimension defined below:

- `domain`: maximum geometry space dimension

- `boundary`: maximum geometry space dimension – 1
- `edges`: 1 (for 3D geometries only)

The function `mphpselectbox` accepts the following property/value pairs:

TABLE 6-36: PROPERTY/VALUE PAIRS FOR THE MPHSELECTBOX COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`adjnumber`	Scalar	None	Adjacent entity number

When a model uses *form an assembly*, more than one vertex can have the same coordinate if the coordinate is shared by separate geometry objects. In that case you can use the `adjnumber` property to identify the domain that the vertices should be adjacent to.

EXAMPLE

Find the domains using a box selection:

```
model = mphopen('model_tutorial_llmatlab');
coordBox = [-1e-3 11e-3;-1e-3 11e-3;9e-3 11e-3];
n = mphselectbox(model,'geom1',coordBox,'domain');
```

Find the boundaries inside the selection box:

```
n = mphselectbox(model,'geom1',coordBox,'boundary');
```

Find the boundaries inside the selection box that are adjacent to domain number 1:

```
n = mphselectbox(model,'geom1',coordBox,'boundary',...
   'adjnumber',1);
```

Find geometric entity number in an assembly

```
model = mphopen('model_tutorial_llmatlab');
geom = model.component('comp1').geom('geom1');
geom.feature('fin').set('action','assembly');
geom.run('fin');
```

Find the boundaries within a box:

```
coordBox = [-1e-3,51e-3;-1e-3,51e-3;9e-3,11e-3];
n = mphselectbox(model,'geom1',coordBox,'boundary');
```

Find the boundary adjacent to domain 2:

```
n = mphselectbox(model,'geom1',coordBox,'boundary',...
   'adjnumber',2);
```

SEE ALSO

`mphgetadj`, `mphgetcoords`, `mphselectcoords`, `mphviewselection`

mphselectcoords

Select a geometric entity using point coordinates.

SYNTAX

```
n = mphselectcoords(model,geomtag,coord,entity,...)
```

DESCRIPTION

`n = mphselectcoords(model,geomtag,coord,entity,...)` finds geometric entity numbers based on their vertex coordinates.

One or more coordinates can be provided. The function searches for vertices near these coordinates using a tolerance radius. The list of the entities that are adjacent to such vertices is returned.

`coord` is a NxM array where N correspond of the number of point to use and M the space dimension of the geometry.

`entity` can be one of `point`, `edge`, `boundary` or `domain` following the entity space dimension defined below:

- `domain`: maximum geometry space dimension
- `boundary`: maximum geometry space dimension -1
- `edges`: 1(only for 3D geometry)

The function `mphpselectcoords` accepts the following property/value pairs:

TABLE 6-37: PROPERTY/VALUE PAIRS FOR THE MPHSELECTCOORDS COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`adjnumber`	Scalar		Adjacent entity number
`radius`	Scalar	`auto`	Search radius
`include`	`all` | `any`	`all`	Include all or any vertices

When a model uses *form an assembly*, more than one vertex can have the same coordinate if the coordinate is shared by separate geometry objects. In that case you can use the `adjnumber` property to identify the domain that the vertices should be adjacent to.

The `radius` property is used to specify the radius of the sphere or circle that the search should be within. A small positive radius (based on the geometry size) is used by default in order to compensate for rounding errors.

Use the property `include` when two point coordinates are used. Set it to `all` to select objects within the search radius of all points. any returns objects within the search radius of any points.

EXAMPLE

Find the geometric entity number:

```
model = mphopen('model_tutorial_llmatlab');
coord = [10e-3 0 10e-3;0 10e-3 10e-3];
n = mphselectcoords(model,'geom1',coord,'point')
```

Return the indices of the point at coordinates within a search radius of 0.011:

```
n = mphselectcoords(model,'geom1',coord,'point',...
   'radius',0.011)
```

Return the indices of the boundaries that have a vertex within the search radius:

```
n = mphselectcoords(model,'geom1',coord,'boundary',...
    'radius',11e-3)
```

Return the indices of the edges that have a vertex within the search radius from all points:

```
coord = [5e-3 0 10e-3;0 5e-3 10e-3];
n = mphselectcoords(model,'geom1',coord,'edge',...
    'radius',6e-3);
```

Return the indices of the edges that have a vertex within the search radius from at least one point:

```
n = mphselectcoords(model,'geom1',coord,'edge',...
   'radius',6e-3,'include','any');
```

Find geometric entity index in an assembly

```
model = mphopen('model_tutorial_llmatlab');
geom = model.component('comp1').geom('geom1');
geom.feature('fin').set('action', 'assembly');
geom.run('fin');
```

Return the indices of the boundaries that have any vertices within the search range of a point:

```
coord = [0,0,10e-3];
n0 = mphselectcoords(model,'geom1',coord,'boundary')
```

Return the indices of the boundaries that also are adjacent to domain 1:

```
n1 = mphselectcoords(model,'geom1',coord,'boundary',...
   'adjnumber',1);
```

Return the indices of the boundaries that also are adjacent to domain 2:

```
n1 = mphselectcoords(model,'geom1',coord,'boundary',...
   'adjnumber',2);
```

SEE ALSO

`mphgetadj`, `mphgetcoords`, `mphselectbox`, `mphviewselection`

mphshowerrors

Show the messages in error nodes in the COMSOL Multiphysics model.

SYNTAX

```
mphshowerrors(model)
list = mphshowerrors(model)
```

DESCRIPTION

`mphshowerrors(model)` shows the error and warning messages stored in the model and where they are located. The output is displayed in the command window.

`list = mphshowerrors(model)` returns the error and warning messages stored in the model and where they are located in the *N*-by-3 cell array list, *N* corresponding to the number of errors or warning found in the model object. The first column contains the node of the error, the second column contain the error message and the third column contains a cell arrays of the model tree nodes that contain the error information, which can help for automated processing of error and warning conditions.

mphsolinfo

Get information about a solution object.

SYNTAX

```
info = mphsolinfo(model,...)
info = mphsolinfo(model,'solname',soltag,...)
```

DESCRIPTION

`info = mphsolinfo(model,...)` returns information about the solution object.

The function `mphsolinfo` accepts the following property/value pairs:

TABLE 6-38: PROPERTY VALUE PAIRS FOR THE MPHSOLINFO COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`soltag`	String	Active solution object	Solution object tag
`dataset`	String	Active solution dataset	Dataset tag
`NU`	`on` | `off`	`off`	Get info about number of solutions

The returned value `info` is a structure with the following content

TABLE 6-39: FIELDS IN THE INFO STRUCTURE

FIELD	CONTENT
`soltag`	Solution node tag
`study`	Study node tag
`size`	Size of the solution vector
`nummesh`	Number of meshes in the solution (for automatic remeshing)
`sizes`	Size of the solution vector for each mesh and number of time steps/parameters for each mesh
`soltype`	Solver type (Stationary, Parametric, Time or Eigenvalue)
`solpar`	Name of the parameter
`sizesolvals`	Length of the parameter list
`solvals`	Values of the parameters, eigenvalues, or time steps
`paramsweepnames`	Parametric sweep parameter names
`paramsweepvals`	Parametric sweep parameter values
`label`	Solution node label
`batch`	Information about solutions for parametric sweeps
`dataset`	List of datasets that directly use the solution
`NUsol`	Number of solution vectors stored
`NUreacf`	Number of reaction forces vectors stored
`NUadj`	Number of adjacency vectors stored
`NUfsens`	Number of functional sensitivity vectors stored
`NUsens`	Number of forward sensitivity vectors stored

You can use the function `mphgetu` to obtain the actual values of the solution vector. Note that these functions are low level functions and you most often would use functions such as `mphinterp` and `mpheval` to extract numerical data from a model.

EXAMPLE

Get the information about the default solution object:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
std.feature.create('stat','Stationary');
std.run;

solinfo = mphsolinfo(model)
```

Get information of multiple solver solution:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90',0);
std.run;
```

Get the information about the 1st outer solution (power = 30):

```
solinfo = mphsolinfo(model,'soltag','sol3');
```

Get the solution vector for 2nd outer solution (power = 60):

```
solinfo = mphsolinfo(model,'soltag','sol4');
```

SEE ALSO

`mphgetu`, `mphxmeshinfo`, `mphsolutioninfo`

mphsolutioninfo

Get information about solution objects and datasets containing given parameters.

SYNTAX

```
info = mphsolutioninfo(model)
info = mphsolutioninfo(model,'parameters',{{ei,vi,toli},...},…)
```

DESCRIPTION

`info = mphsolutioninfo(model)` returns information about all solution object and solution dataset combinations in `model`.

`info = mphsolutioninfo(model,'parameters',{{ei,vi,toli}, ...},…)` returns information about solution object and solution dataset containing the given

inner/outer solution parameters `ei` with the value equal to `vi` within the tolerance `toli`.

The function `mphsolutioninfo` accepts the following property/value pairs:

TABLE 6-40: PROPERTY VALUE PAIRS FOR THE MPHSOLUTIONINFO COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`cellmap`	`off` | `on`	off	Set to return a cell version of the map with headers
`dataset`	String	Active solution dataset	Dataset tag
`parameters`	Cell | Cell array		Filter parameters, values, and tolerances
`soltag`	String | String cell array		Solution object tag
`sort`	String | Scalar | `auto`	auto	Sort the map by column number or header tag

The returned value `info` is a structure with the following content

TABLE 6-41: FIELDS IN THE INFO STRUCTURE

FIELD	CONTENT
`solutions`	List of matched solution tags
`sol#`	Substructure containing information related to solution number #

The substructure `info.sol#` has the following content

TABLE 6-42: FIELDS IN THE INFO.SOL# SUBSTRUCTURE

FIELD	CONTENT
`dataset`	Tag of the solution dataset
`study`	Tag of the study associated to the solution
`sequencetype`	Type of solution object
`cellmap`	Cellmap describing the connections between parameters and inner/outer solution numbers
`values`	Parameters values used in the solution
`parameters`	Parameters names used in the solution
`mapheaders`	Headers for the map
`map`	Map describing the connections between parameters and inner/outer solution numbers

EXAMPLE

Load `model_tutorial_llmatlab.mph`:

```
model = mphopen('model_tutorial_llmatlab');
```

Create a study combining a parametric sweep and a transient study step:

```
std = model.study.create('std');
param = std.feature.create('param','Parametric');
time = std.feature.create('time','Transient');
```

Set the time stepping and the parametric sweep parameters:

```
time.set('tlist', 'range(0,1,25)');
param.setIndex('pname','power',0);
param.setIndex('plistarr','30 60 90', 0);
```

Run the study:

```
std.run;
```

Retrieve the solution information corresponding to power = 30 W:

```
info = mphsolutioninfo(model,'parameters',{'power',30,0})
```

Retrieve the solution information corresponding to power = 90 W and around t = 10.4 sec. and its associated solution dataset:

```
info = mphsolutioninfo(model,'parameters',{{'power',90,0},...
   {'t',10.4,0.5}})
```

Get the `solution` `solution` dataset associated:

```
dset = info.sol2.dataset
```

Get the inner and outer solution number:

```
solnum = info.sol2.map(end-1)
outersolnum = info.sol2.map(end)
```

SEE ALSO

`mphgetu`, `mphxmeshinfo`, `mphsolinfo`

mphstart

Connect MATLAB® to a COMSOL *server*.

SYNTAX

```
mphstart
mphstart(port)
mphstart(ipaddress, port)
mphstart(ipaddress, port, username, password)
mphstart(ipaddress, port, comsolpath)
mphstart(ipaddress, port, comsolpath, username, password)
```

DESCRIPTION

`mphstart` creates a connection with a COMSOL *server* using the default port number (which is 2036).

`mphstart(port)` creates a connection with a COMSOL *server* using the specified port number `port`.

`mphstart(ipaddress, port)` creates a connection with a COMSOL *server* using the specified IP address `ipaddress` and the port number `port`. This command assumes that the client and the server machine share the same login properties.

`mphstart(ipaddress, port, username, password)` creates a connection with a COMSOL *server* using the specified IP address `ipaddress` and the port number `port`, the username `username` and password `password`.

`mphstart(ipaddress, port, comsolpath)` creates a connection with a COMSOL *server* using the specified IP address and port number using the `comsolpath` that is specified. This is useful if `mphstart` cannot find the location of the COMSOL Multiphysics installation.

`mphstart(ipaddress, port, comsolpath, username, password)` creates a connection with a COMSOL *server* using the specified IP address, the port number, the username and password using the `comsolpath` that is specified. This is useful if `mphstart` cannot find the location of the COMSOL Multiphysics installation.

`mphstart` can be used to create a connection from within MATLAB when this is started without using the *COMSOL with MATLAB* option. `mphstart` then sets up the necessary environment and connect to COMSOL.

Prior to calling `mphstart` it is necessary to set the path of `mphstart.m` in the MATLAB path or to change the current directory in MATLAB (for example, using the cd command) to the location of the `mphstart.m` file.

A COMSOL *server* must be started prior to running `mphstart`.

`mphstart` connect to either a COMSOL Multiphysics Server (started with the command: `comsol mphserver`) or the COMSOL Server™. To connect to the

COMSOL Server™ from a computer that has just MATLAB installed it is necessary to run the COMSOL Server™ Client installer.

Once MATLAB is connected to the server, import the COMSOL class in order to use the ModelUtil commands. To import the COMSOL class enter:

```
import com.comsol.model.*
import com.comsol.model.util.*
```

EXAMPLE

Connect manually MATLAB to a COMSOL Multiphysics Server and create a model:

```
mphstart
import com.comsol.model.*
import com.comsol.model.util.*
model = ModelUtil.create('Model');
```

Connect manually MATLAB to a COMSOL Multiphysics Server running on the computer with the IP address 192.168.0.1 using port 2037:

```
mphstart('192.168.0.1',2037)
```

mphstate

Get state-space matrices for a dynamic system.

SYNTAX

```
str = mphstate(model,soltag,'Out',{'SP'})
str = mphstate(model,soltag,'Out',{'SP1','SP2',...})
```

DESCRIPTION

`str = mphstate(model,soltag,'out',{'SP'})` returns a MATLAB® structure `str` containing the state-space matrix `SP` assembled using the solver node `soltag` and accessible as `str.SP`, `SP` being taken from the `Out` property list.

`str = mphstate(model,soltag,'out',{'SP1','SP2',...})` returns a MATLAB structure `str` containing the state-space matrices `SP1`, `SP2`, ... assembled using the solver node `soltag` and accessible as `str.SP1`and `str.SP2`. `SP1` and `SP2` being taken from the out property list.

The function `mphstate` accepts the following property/value pairs:

TABLE 6-43: PROPERTY VALUE FOR THE MPHSTATE COMMAND

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`extractafter`	Solution feature tag		Specify where to extract the matrices
`initmethod`	`init` | `sol`	`init`	Use linearization point
`initsol`	`soltag` | `zero`	`soltag`	Solution to use for linearization
`input`	String		Input variables
`keepfeature`	`off` | `on`	`off`	Keep the StateSpace feature in the model
`out`	`MA` | `MB` | `A` | `B` | `C` | `D` | `Mc` | `Null` | `ud` | `x0`		Output matrix
`output`	String		Output variables
`solnum`	Positive integer	`auto`	Solution number
`sparse`	`off` | `on`	`off`	Return sparse matrices

The property `Sparse` controls whether the matrices `A`, `B`, `C`, `D`, `M`, `MA`, `MB`, and `Null` are stored in the sparse format.

The equations correspond to the system below:

$$\begin{cases} M_C\dot{x} = M_C Ax + M_C Bu \\ y = Cx + Du \end{cases}$$

where x are the state variables, u are the input variables, and y are the output variables.

A static linearized model of the system can be described by:

$$y = (D - C(M_C A)^{-1} M_C B)u$$

The full solution vector U can be then obtained from

$$U = \text{Null}x + ud + u_0$$

where Null is the null-space matrix, ud the constraint contribution, and u_0 is the linearization point, which is the solution stored in the sequence once the state-space export feature is run.

The matrices M_C and $M_C A$ are produced by the same algorithms that do the finite-element assembly and constraint elimination in COMSOL Multiphysics. M_C and

$M_C A$ are the same as the matrices D_C (eliminated mass matrix) and $-K_C$ (K_C is the eliminated stiffness matrix). The matrices are produced from an exact residual vector Jacobian calculation (that is, differentiation of the residual vector with respect to the degrees of freedoms x) plus an algebraic elimination of the constraints. The matrix C is produced in a similar way (that is, the exact output vector Jacobian matrix plus constraint elimination).

The matrices $M_C B$ and D are produced by a numerical differentiation of the residual and output vectors, respectively, with respect to the input parameters (the algorithm systematically perturbs the input parameters by multiplying them by a factor $1+10^{-8}$).

The input cannot be a variable constraint in the model.

The matrices are assembled using the current solution available as linearization point unless the `initmethod` property is provided. In case of the presence of a solver step node you need either to disable the solver step node in the model or to set the property `extractafter` with the Dependent Variables node tag.

EXAMPLE

Load `model_tutorial_llmatlab.mph`:

```
model = mphopen('model_tutorial_llmatlab');
comp1 = model.component('comp1');
comp1.mesh('mesh1').autoMeshSize(9);
std = model.study.create('std');
time = std.feature.create('time','Transient');
time.set('tlist','range(0,1,50)');
```

Edit the model to use parameter for the initial and external temperature:

```
model.param.set('T0','293.15[K]');
model.param.set('Text','300[K]');
ht = model.component('comp1').physics('ht');
ht.feature('init1').set('Tinit', 'T0');
ht.feature('hf1').set('Text', 'Text');
```

Add a domain point probe plot:

```
pdom = comp1.probe.create('pdom', 'DomainPoint');
pdom.model('comp1');
pdom.set('coords3',[0 0 1.1e-2]);
```

Run the study and create a plot group to display the probe:

```
std.run;
pg1 = model.result.create('pg1', 'PlotGroup1D');
glob1 = pg1.create('glob1', 'Global');
glob1.setIndex('expr', 'comp1.ppb1', 0);
```

Extract the state-space system matrices of the model with `power`, `Temp`, and `Text` as input and the probe evaluation `comp1.ppb1` as output:

```
M = mphstate(model,'sol1','out',{'A','B','C','D'},...
   'input',{'power','Temp','Text'},'output','comp1.ppb1');
```

Plot the sparsity of the matrix A:

```
subplot(1,2,1); spy(M.A); subplot(1,2,2); spy(abs(M.A)>1e-2)
```

Set the input power parameter and the reference temperature:

```
power = 30; Temp = 300; Text = 300; T0 = 293.15;
```

Compute the system solution:

```
input = [power Temp-T0 Text-T0];
func = @(t,x) M.A*x + M.B*input';
[t,x] = ode45(func,0:1:50,zeros(size(M.A,1),1));
y = M.C*x';
y = y+T0;
```

Plot the result:

```
plot(t,y,'r');
hold on;
mphplot(model,'pg1');
```

Evaluate the steady-state temperature value:

```
G = M.D-M.C*(inv(M.A))*M.B;
y_inf = full(G*input');
y_inf = y + T0
```

mphsurf

Create plot data structure from surf data.

SYNTAX

```
pd = mphsurf(x,y,z)
pd = mphsurf(z)
```

DESCRIPTION

`pd = mphsurf(x,y,z)` creates the plot data structure `pd` from surf data `x`, `y`, and `z`.

`pd = mphsurf(z)` creates the plot data structure `pd` from surf data `z`. A unit scale is assumed for the *x* and *y* coordinates.

EXAMPLE

Create random height data

```
[x,y] = meshgrid(-0.1:0.2:1.1,-0.4:0.2:0.4);
z = cumsum(0.1*randn(size(x)));
```

Create 3D surface plotdata structure

```
pd = mphsurf(x,y,z);
mphplot(pd)
```

SEE ALSO

`mphquad2tri`, `mphreadstl`, `mphwritestl`

mphtable

Get table data.

SYNTAX

```
info = mphtable(model,tabletag)
```

DESCRIPTION

`info = mphtable(model,tabletag)` returns the structure `info` containing the data with the `tabletag` tag and its headers.

The returned value `info` is a structure with the following content

TABLE 6-44: FIELDS IN THE INFO STRUCT

FIELD	CONTENT
`headers`	Headers of the table
`tag`	Tag of the table
`data`	Data of the extracted table
`filename`	Filename when table exported to file

EXAMPLE

Load `model_tutorial_llmatlab.mph`, add a stationary study and compute the solution for different power values:

```
model = mphopen('model_tutorial_llmatlab');
std = model.study.create('std');
stat = std.feature.create('stat','Stationary')
stat.setIndex('pname','power',0);
stat.setIndex('plistarr','30 60 90',0);
std.run;
```

Evaluate the maximum temperature in the model and set the results in a table:

```
max = model.result.numerical.create('max','MaxVolume');
max.selection.all;
tbl = model.result.table.create('tbl','Table');
tbl.comments('Volume Maximum (T)');
max.set('table','tbl');
max.setResult;
```

Extract the table data:

```
str = mphtable(model,'tbl');
tbl_data = str.data
```

SEE ALSO

mpheval, mphevalpoint, mphglobal, mphint2, mphinterp, mphmax, mphmean, mphmin

mphtags

Get tags and names for nodes in a COMSOL Multiphysics model.

SYNTAX

```
mphtags(model)
mphtags(node)
mphtags(model, type)
[tags,labels,displaystrings] = mphtags(...)
mphtags
mphtags -show
[tags,filename,fullfilename] = mphtags
```

DESCRIPTION

`mphtags` is used to retrieve tags from nodes in a COMSOL Multiphysics model or tags from models that are loaded on the server.

When `mphtags` is called with a `model` or `node` variable the tags are returned form the model. `mphtags` also be called using a model variable and a type, where type can be one of these strings: `result, dataset, table, numerical`, and `export` to give easy access to the nodes under the result node. It is sufficient to use the first letter of the types.

If `mphtags` is called with output arguments it is possible to get both the tags as well as labels and display names used for the nodes. For example,

```
[tags,labels,displaystrings] = mphtags(model.geom)
```

If `mphtags` is called with the root model node as argument, the filename of the model can be returned:

```
[tag,filename,displaystring] = mphtags(model)
```

`mphtags` can be used to return a list of files currently loaded on the server. For example,

```
[tags,filename,fullfilename] = mphtags
```

In order to see this information quickly it is possible to call `mphtags` like this:

```
mphtags -show
```

that just produces output that is useful viewing on screen.

mphthumbnail

Set or get model thumbnail.

SYNTAX

```
mphthumbnail(model,filename)
mphthumbnail(model,image)
mphthumbnail(model,fig)
mphthumbnail(model,'')
[image,imagefilename] = mphthumbnail(model)
```

DESCRIPTION

`mphthumbnail` sets or gets the model thumbnail for model loaded on the server. In order to update the model thumbnail on disk the model must be saved.

`mphthumbnail(model,filename)` sets the thumbnail for the model to the image contained in `filename`. The file must be a PNG- or JPG-file.

`mphthumbnail(model,image)` sets the thumbnail using the image data `image`. `image` is either a NxM or a NxMx3 matrix. The preferred size of the image is 280 by 210 pixels.

`mphthumbnail(model,fig)` sets the thumbnail using the image in the figure with handle `fig`.

`mphthumbnail(model, '')` clears the thumbnail from the model.

`[image,imagefilename] = mphthumbnail(model)` gets the image data `image` and the image filename `imagefilename` for the thumbnail stored in the model `model`.

EXAMPLE

Load model_tutorial_llmatlab.mph:

```
model = mphopen('model_tutorial_llmatlab');
```

Get the thumbnail image:

```
im = mphthumbnail(model);
```

Show the thumbnail in a MATLAB figure:

```
imshow(im)
```

Save the current figure image as an image file:

```
filename = fullfile(tempdir,'imagefile.png');
print(filename,'-dpng','-r48')
```

Set the thumbnail for the model:

```
mphthumbnail(model,filename)
```

SEE ALSO

mphload, mphsave

mphversion

Return the version number for COMSOL Multiphysics.

SYNTAX

```
v = mphversion
[v,vm] = mphversion(model)
```

DESCRIPTION

`v = mphversion` returns the COMSOL Multiphysics version number that MATLAB is connected to as a string.

`[v,vm] = mphversion(model)` returns the COMSOL Multiphysics version number that MATLAB is connected to as a string in the variable `v` and the version number of the model in the variable `vm`.

EXAMPLE

Load `model_tutorial_llmatlab.mph`:

```
model = mphopen('model_tutorial_llmatlab');
```

Get the version numbers:

```
[version, model_version] = mphversion(model)
```

SEE ALSO

mphload, mphsave

mphviewselection

Display a geometric entity selection in a MATLAB® figure.

SYNTAX

```
mphviewselection(model,geomtag,number,entity,...)
mphviewselection(model,seltag,...)
```

DESCRIPTION

`mphviewselection(model,geomtag,number,entity,...)` displays the geometric entity `number` of type `entity` in MATLAB figure including the representation of the geometry `geomtag`.

`mphviewselection(model,seltag,...)` displays the geometric entity selection `seltag` in a MATLAB figure including the representation of the geometry.

The function `mphviewselection` accepts the following property/value pairs:

TABLE 6-45: PROPERTY VALUE/PAIRS FOR THE MPHVIEWSELECTION FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`edgecolor`	Char | RGB array	`k`	Color for edges
`edgecolorselected`	RGB array	`[1,0,0]`	Color for selected edges
`edgelabels`	on | `off`	`off`	Show edge labels
`edgelabelscolor`	Char | RGB array	`g`	Color for edge labels
`edgemode`	on | `off`	`on`	Show edges
`entity`	`Domain` | `boundary` | `edge` | `point`		Set the selected entity type
`facealpha`	Double	`1`	Set transparency value
`facecolor`	RGB array	`[0.6,0.6, 0.6]`	Color for face
`facecolorselected`	RGB array	`[1,0,0]`	Color for selected face
`facelabels`	on | `off`	`off`	Show face labels
`facelabelscolor`	Char | RGB array	`b`	Color for face labels

TABLE 6-45: PROPERTY VALUE/PAIRS FOR THE MPHVIEWSELECTION FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
facemode	on | off	on	Show faces
geommode	on | off	on	Show entire geometry
marker		.	Vertex marker
markercolorselected	Char | RGB array	r	Color for selected vertex marker
markersize	Int	12	Font size of marker
parent	Double		Parent axis
renderer	Opengl | zbuffer	opengl	Set the rendering method
selection	String | Positive integer array		Set selection name or entity number
selectoralpha	Double	0.25	Set selector transparency value
selectorcolor	RGB array	[0,0,1]	Color for selected marker
showselector	on | off	on	Show Selector
vertexlabels	on | off	off	Show vertex labels
vertexlabelscolor	Char | RGB array	r	Color for vertex labels
vertexmode	on | off	off	Show vertices

EXAMPLE

Plot boundary 6 using yellow color:

```
model = mphopen('model_tutorial_llmatlab');
mphviewselection(model,'geom1',6,'boundary',...
   'facecolorselected',[1 1 0],'facealpha',0.5)
```

Plot edges 1 to 8 using green color:

```
mphviewselection(model,'geom1',1:8,'edge',...
   'edgecolorselected',[0 1 0])
```

Add an explicit selection for boundaries 7 to 12 and plot the selection in a figure:

```
model.selection.create('sel1','Explicit').geom(2).set(7:12);
mphviewselection(model,'sel1');
```

Add a selection to get the vertex indices with the box delimited with the coordinates [-1e-3 11e-3;-1e-3 11e-3;9e-3 11e-3] and plot both the selected entities and the selector:

```
box = model.selection.create('box1', 'Box');
box.set('entitydim', '0');
box.set('xmin', '-1e-3').set('xmax', '11e-3');
box.set('ymin', '-1e-3').set('ymax', '11e-3');
box.set('zmin', '10e-3').set('zmax', '11e-3');
mphviewselection(model,'box1','facemode','off');
```

SEE ALSO

mphgeom, mphselectbox, mphselectcoords

mphwritestl

Export plot data as an STL file.

SYNTAX

```
mphwritestl(filename, pd)
mphwritestl(filename, pd, '-binary')
```

DESCRIPTION

`mphwritestl(filename, pd)` exports data in the plot data structure `pd` as the STL file `filename`.

`mphwritestl(filename, pd, '-binary')` exports data in the plot data structure `pd` as the STL file `filename` using the binary file format.

`pd` is a structure with fields `pd` and `t`.

- The field `p` contains node point coordinate information.
- The field `t` contains the indices to columns in `p` of a simplex mesh, each column in `t` representing a simplex.

Other fields in the plot data structure are not considered to generate the surface mesh.

EXAMPLE

.Generate a surface mesh from solution plot:

```
model = mphopen('vacuum_flask_llmatlab')
pd = mphplot(model, 'pg1')
pd2stl = pd{2}{1};
mphwritestl('vacuum_flask.stl', pd2stl)
```

Generate a surface mesh from a volume mesh

```
model = mphopen('model_tutorial_llmatlab');
model.component('comp1').mesh('mesh1').run;
[s,d] = mphmeshstats(model, 'mesh1');
```

```
idx = strcmp(s.types, 'tri');
pdmesh.p = d.vertex;
pdmesh.t = d.elem{idx};
mphwritestl('mesh2geom.stl', pdmesh);
```

SEE ALSO

`mphquad2tri`, `mphreadstl`, `mphsurf`

mphxmeshinfo

Extract information about the extended mesh.

SYNTAX

```
info = mphxmeshinfo(model, ...)
```

DESCRIPTION

`info = mphxmeshinfo(model,...)` extracts extended mesh information from the active solution object.

The function `mphxmeshinfo` accepts the following property/value pairs:

TABLE 6-46: PROPERTY VALUE/PAIRS FOR THE MPHXMESHINFO FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`soltag`	String	Active solution object	Solution object tag
`studysteptag`	String		Study step node tag
`meshcase`	Positive integer | String	1	Mesh case tag

The function `xmeshinfo` returns a structure with the fields shown in the table below

TABLE 6-47: FIELD IN THE RETURNED STRUCTURE FROM MPHXMESHINFO

FIELD	DESCRIPTION
`soltag`	Tag of the solution object
`ndofs`	Number of DOFs
`fieldnames`	Names of the field variables
`fieldndofs`	Number of DOFs per field name
`meshtypes`	Types of mesh element
`dofs`	Structure with information about the degrees of freedom
`nodes`	Structure with information about the nodes
`elements`	Structure with information about each element type

The extended mesh information provide information about the numbering of elements, nodes, and degrees of freedom (DOFs) in the extended mesh and in the matrices returned by `mphmatrix` and `mphgetu`.

EXAMPLE

Extract xmesh information:

```
model = mphopen('model_tutorial_llmatlab.mph');
std = model.study.create('std');
std.feature.create('stat', 'Stationary');
std.run;
info = mphxmeshinfo(model)
```

Get the number of degrees of freedom and the nodes coordinates:

```
dofs = info.ndofs
coords = info.dofs.coords;
```

Get the DOFs indices connected to the tetrahedron:

```
idx = info.elements.tet.dofs
```

Retrieve the xmesh information with several physics

```
model = mphopen('model_tutorial_llmatlab.mph');
comp1 = model.component('comp1');
ec = comp1.physics.create('ec','ConductiveMedia','geom1');
ec.feature.create('gnd1','Ground',2).selection.set(3);
pot = ec.feature.create('pot','ElectricPotential',2);
pot.selection.set(7);
pot.set('V0',1,'50[mV]');
hs = comp1.physics('ht').feature('hs1');
hs.set('heatSourceType',1,'generalSource');
hs.set('Q_src',1,'root.comp1.ec.Qh');
std = model.study.create('std');
std.feature.create('stat', 'Stationary');
std.run;
info = mphxmeshinfo(model)
```

Get the index of the nodes for element with the index 100:

```
idx_nodes = info.elements.tet.nodes(:,100)
```

Get the index of the DOFs for element with the index 100:

```
idx_dofs = info.elements.tet.dofs(:,100)
```

Get the index of the variables names corresponding to the DOFs with the index `idx_dofs`:

```
idx_names = info.dofsinds(idx_dofs);
```

Find the `dofnames` index corresponding to the variable V:

```
idx_dofnames = find(strcmp(info.dofs.dofnames,'comp1.V'))-1;
```

Get the list of DOFs that correspond to the variable *V*:

```
idx = find(idx_names==idx_dofnames)
```

Get the coordinates of the DOFs corresponding to the dependent variable *V* that belong to element 100:

```
info.dofs.coords(:,idx_dofs(idx))
```

SEE ALSO:

`mphgetu`, `mphmatrix`, `mphsolinfo`, `mphsolutioninfo`

Index