

INTRODUCTION TO Application Builder

5.2

 COMSOL

Introduction to Application Builder

© 1998–2015 COMSOL

Protected by U.S. Patents listed on www.comsol.com/patents, and U.S. Patents 7,519,518; 7,596,474; 7,623,991; 8,457,932; 8,954,302; 9,098,106; and 9,146,652. Patents pending.

This Documentation and the Programs described herein are furnished under the COMSOL Software License Agreement (www.comsol.com/comsol-license-agreement) and may be used or copied only under the terms of the license agreement.

COMSOL, COMSOL Multiphysics, Capture the Concept, COMSOL Desktop, LiveLink, and COMSOL Server are either registered trademarks or trademarks of COMSOL AB. All other trademarks are the property of their respective owners, and COMSOL AB and its subsidiaries and products are not affiliated with, endorsed by, sponsored by, or supported by those trademark owners. For a list of such trademark owners, see www.comsol.com/trademarks.

Version: COMSOL 5.2

Contact Information

Visit the Contact COMSOL page at www.comsol.com/contact to submit general inquiries, contact Technical Support, 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/community
- Events: www.comsol.com/events
- COMSOL Video Gallery: www.comsol.com/video
- Support Knowledge Base: www.comsol.com/support/knowledgebase

Part number: CM020011

Contents

Introduction	7
The Application Builder Desktop Environment	9
The Application Builder and the Model Builder	16
Parameters, Variables, and Scope	17
Running Applications	19
Running Applications in COMSOL Multiphysics	19
Running Applications in COMSOL Server	27
Getting Started with the Application Builder	32
The Form Editor	37
The Forms Settings Window	37
The Individual Form Settings Windows	38
Form Editor Preferences	40
Form Objects	41
Editor Tools in the Form Editor	46
Button	48
Graphics	59
Input Field	70
Unit	75
Text Label	76
Data Display	77
Model Data Access in the Form Editor	80
Sketch and Grid Layout	86
Copying Between Applications	100
The Main Window	102
Menu Bar and Toolbar	104

Ribbon	107
Events	109
Events at Startup and Shutdown.....	110
Global Events	110
Form and Form Object Events.....	112
Using Local Methods	114
Declarations	115
Scalar Declarations.....	118
Array 1D Declarations.....	121
Array 2D Declarations.....	123
Choice List Declarations	124
File Declarations	126
The Method Editor	127
Converting a Command Sequence to a Method.....	127
Language Elements Window.....	131
Editor Tools in the Method Editor.....	132
Model Data Access in the Method Editor.....	134
Recording Code	136
Checking Syntax	139
Find and Replace.....	140
Model Expressions Window.....	141
Extracting Variables.....	141
Syntax Highlighting, Code Folding, and Indentation	142
Method Editor Preferences	144
Ctrl+Space and Tab for Code Completion	145
Local Methods.....	146
Methods with Input and Output Arguments.....	149
Debugging	151

Stopping a Method	153
The Model Object	153
Language Element Examples	153
Libraries.....	159
Images.....	159
Sounds	161
Files.....	163
Appendix A—Form Objects	164
List of All Form Objects.....	164
Toggle Button	165
Check Box	168
Combo Box.....	172
Equation	189
Line.....	190
Web Page	191
Image	192
Progress Bar.....	192
Log	195
Message Log	196
Results Table	198
Form.....	199
Form Collection	201
Card Stack	203
File Import	206
Information Card Stack	210
Array Input.....	214
Radio Button	218
Selection Input.....	220

Text	224
List Box	225
Table.	229
Slider.	234
Toolbar	236
Spacer.	237
Appendix B—Copying Between Applications.....	239
Appendix C—File Handling and File Scheme Syntax.....	241
File Handling with COMSOL Server	241
File Scheme Syntax.....	244
File Import	246
File Export	254
Appendix D—Keyboard Shortcuts.....	262
Appendix E—Built-in Method Library.....	264
Appendix F—Guidelines for Building Applications	321
Appendix G—The Application Library Examples	324

Introduction

A COMSOL® application is an intuitive and efficient way of interacting with a COMSOL Multiphysics® model through a highly specialized user interface. This book gives a quick overview of the Application Builder desktop environment with examples that show you how to use the Form editor and the Method editor. Reference materials are also included in this book featuring a list of the built-in methods and functions that are available. For detailed information on how to use the Model Builder, see the book *Introduction to COMSOL Multiphysics*.



If you want to check out an example application before reading this book, open and explore one of the applications from the Application Libraries, in one of the Applications folders. Keep it open while reading this book to try things out. Only the Applications folders contain applications with user interfaces. The other folders in the Application Libraries are tutorial models with no user interfaces.

The Application Builder is included in the Windows® version of COMSOL Multiphysics and accessible from the COMSOL Desktop® environment.

COMSOL Multiphysics and its add-on products are used to create an application. A license for the same add-on products is required to run the application from the COMSOL Multiphysics or COMSOL Server™ products.

Additional resources, including video tutorials, are available online at www.comsol.com.

RUNNING APPLICATIONS WITH COMSOL MULTIPHYSICS

With a license of COMSOL Multiphysics, applications can be run from the COMSOL Desktop in Windows®, OS X, and Linux®.

RUNNING APPLICATIONS WITH COMSOL SERVER

With a COMSOL Server license, a web implementation of an application can be run in major web browsers on platforms such as Windows®, OS X, iOS, Linux®, and Android™. In Windows®, you can also run COMSOL applications by connecting to a COMSOL Server with an easy-to-install COMSOL Client, available for download from www.comsol.com. COMSOL Server does not include the Application Builder, Physics Builder, and Model Builder tools that come with the COMSOL Desktop environment.

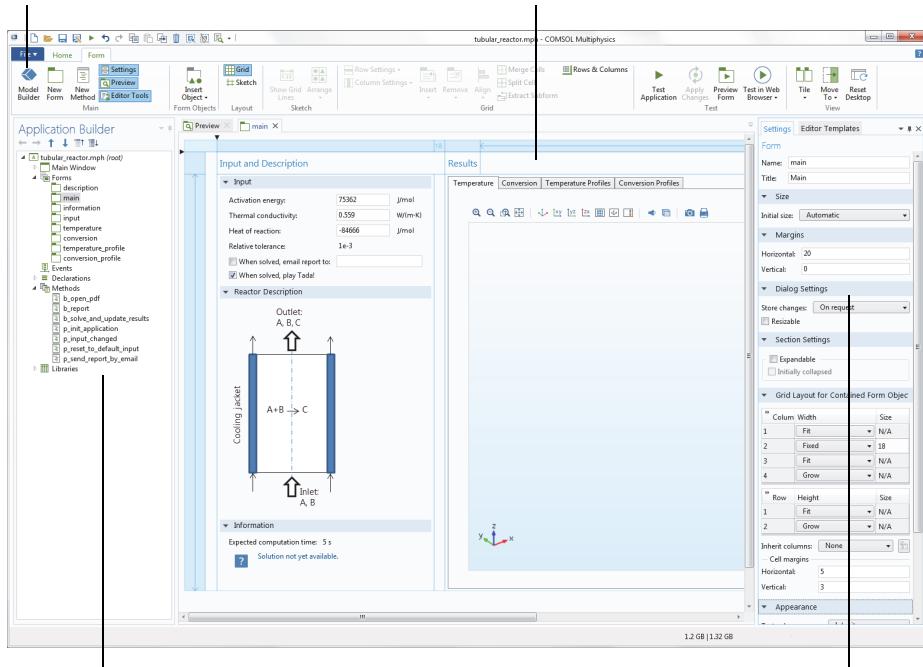
GUIDELINES FOR BUILDING APPLICATIONS

If you are not experienced in building a graphical user interface or in programming, you may want to read “Appendix F—Guidelines for Building Applications” on page 321.

The Application Builder Desktop Environment

MODEL BUILDER and **APPLICATION BUILDER**—Switch between the Model Builder and the Application Builder by clicking this button.

COMSOL DESKTOP ENVIRONMENT—The COMSOL Desktop environment provides access to the Application Builder, including the Form and Method editors, as well as the Model Builder.



APPLICATION BUILDER WINDOW—The Application Builder window with the application tree.

SETTINGS WINDOW—Click any node in the application tree, including those for form objects or methods, to see its associated Settings window.

The screenshot above is representative of what you will see when you are working with the Application Builder. The main components of the Application Builder desktop environment are:

- Application Builder window and ribbon tab
- COMSOL Desktop environment
- Form editor (see page 37)
- Method editor (see page 127)

THE APPLICATION TREE

The application tree consists of the following nodes

- **Main Window**
- **Forms**
- **Events**
- **Declarations**
- **Methods**
- **Libraries**

The **Main Window** node represents the main window of an application and is also the top-level node for the user interface. It contains the window layout, the main menu specification, and an optional ribbon specification.

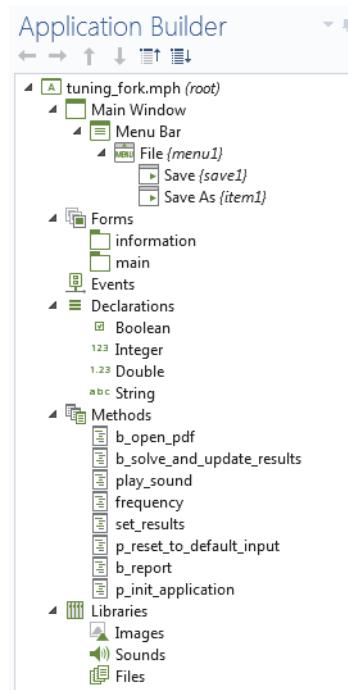
The **Forms** node contains subnodes that are forms. Each form may contain a number of form objects such as input fields, graphics objects, and buttons.

The **Events** node contains subnodes that are global events. These include all events that are triggered by changes to the various data entities, such as global parameters or string variables. Global events can also be associated with the startup and shutdown of the application.

The **Declarations** node is used to declare global variables, which are used in addition to the global parameters and variables defined in the model.

The **Methods** node contains subnodes that are methods. Methods contain code for actions not included among the standard run commands of the model tree nodes in the Model Builder. The methods may, for example, execute loops, process inputs and outputs, and send messages and alerts to the user of the application.

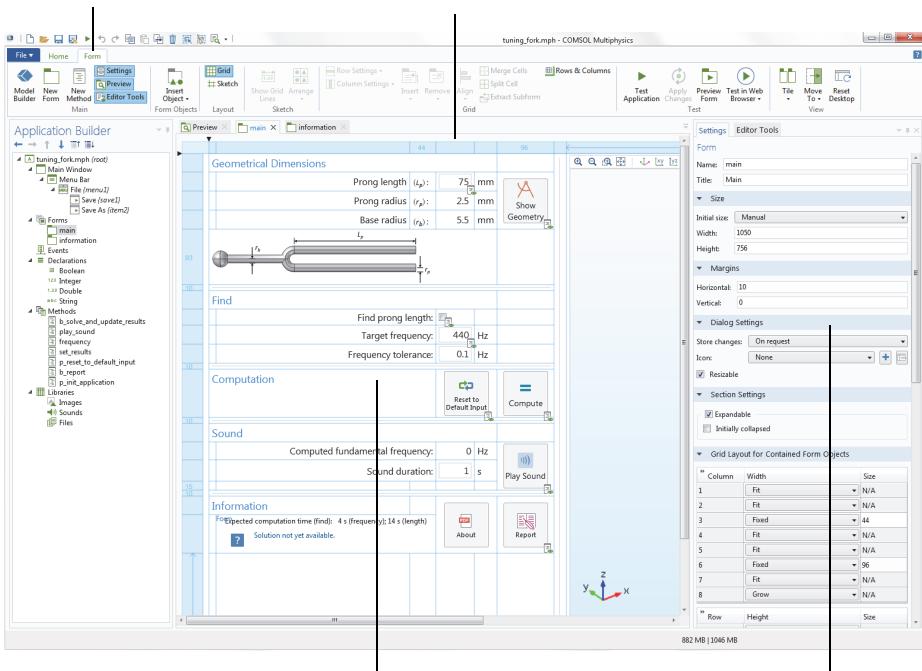
The **Libraries** node contains images, sounds, and files to be embedded in an MPH file so that you do not have to distribute them along with the application. In addition, the **Libraries** node may contain Java® utility class nodes and nodes for external Java® and C libraries.



THE FORM EDITOR

FORM TAB—The Form tab in the ribbon gives easy access to the Form editor.

FORM EDITOR WINDOW—The tabbed Form editor window allows you to move objects around by dragging. Click an object to edit its settings.



FORM OBJECTS—Each form contains form objects such as input fields, check boxes, graphics, images, buttons, and more.

SETTINGS and EDITOR TOOLS WINDOWS—Click any application tree node or form object to see its associated Settings window. The Editor Tools window is used to quickly create form objects.

Use the Form editor for user interface layout by creating forms with form objects such as input fields, graphics, and buttons.

The main components of the Form editor are:

- Form ribbon tab
- Application Builder window with the application tree
- Form window
- Editor Tools window
- Settings window

Creating a New Form

To create a new form, right-click the **Forms** node of the application tree and select **New Form**. You can also click **New Form** in the ribbon. Creating a new form will automatically open the **New Form** wizard.

If your application already has a form, let's say **form1**, and you would like to edit it, you can open the Form editor in either of these two ways:

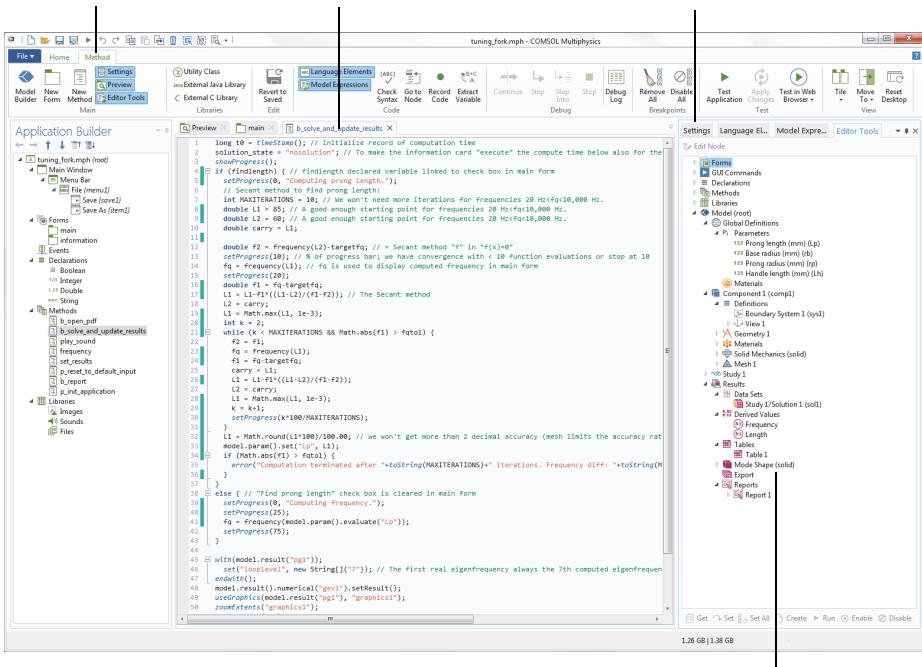
- In the application tree, double-click the **form1** node.
- In the application tree, right-click the **form1** node and select **Edit**.

THE METHOD EDITOR

METHOD TAB—The Method tab in the ribbon gives easy access to tools for writing and debugging code.

METHOD WINDOW—The tabbed Method Window allows you to switch between editing different methods.

SETTINGS WINDOW—Click any application tree node to see its associated Settings window.



MODEL EXPRESSIONS, LANGUAGE ELEMENTS, and EDITOR TOOLS WINDOWS—These windows display tools for writing code. The Model Expressions window shows all constants, parameters, variables, and functions available in the model. The Language Elements window is used to insert template code for built-in methods. The Editor Tools window is used to extract code for editing and running model tree nodes.

Use the Method editor to write methods for actions not covered by the standard use of the model tree nodes. A method is another name for what in other programming languages is known as a subroutine, function, or procedure.

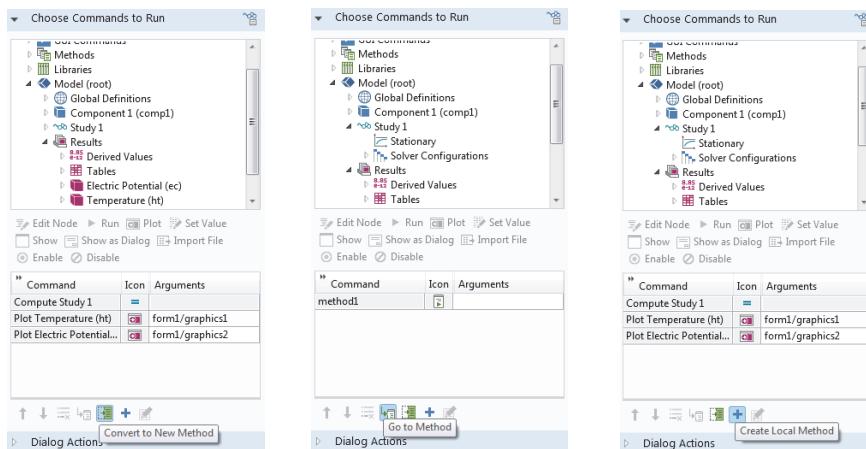
The main components of the Method editor are:

- Method ribbon tab
- Application Builder window with the application tree
- Method window
- Model Expressions, Language Elements, Editor Tools, and Settings windows (these are stacked together in the figure above)

Creating a New Method

To create a new method, right-click the **Methods** node, in the application tree, and select **New Method**. You can also click **New Method** in the ribbon. Creating a new method will automatically open the Method editor. Methods created in this way are global methods and are accessible from all methods and form objects.

-  A sequence of commands associated with a button or menu item can be automatically converted to a new method by clicking **Convert to New Method**. Open the new method by clicking **Go to Method**. You can also create a method that is local to a form object by clicking **Create Local Method**. These options as shown in the figure below.



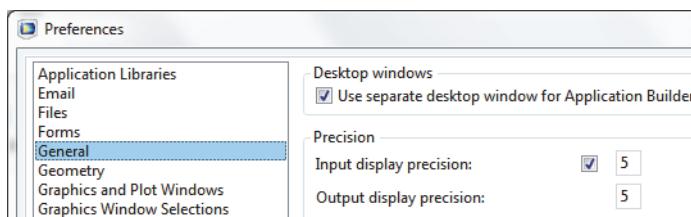
If a method already exists, say with the name `method1`, then you open the Method editor using, for example, any of these ways:

- In the application tree, double-click the **method1** node.
- In the application tree, right-click the **method1** node and select **Edit**.
- Below the command sequence in the **Settings** window of a form object or an event, click **Go to Method**.

THE APPLICATION BUILDER IN A SEPARATE DESKTOP WINDOW

You can configure the COMSOL Desktop environment so that the Application Builder is displayed in a separate desktop window. In the **File** menu, select

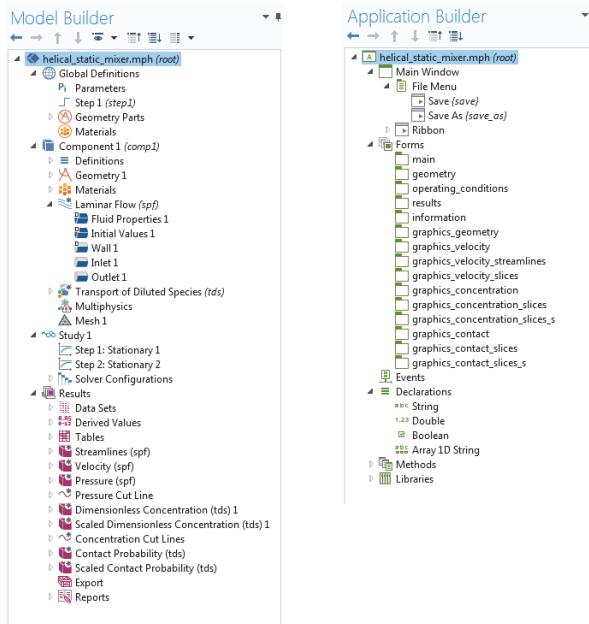
Preferences. In the **General** page, select the check box **Use separate desktop window for Application Builder**.



You can use the keyboard shortcuts Ctrl+Alt+M and Ctrl+Alt+A to switch between the Model Builder and Application Builder, respectively.

The Application Builder and the Model Builder

Use the Application Builder to create an application based on a model built with the Model Builder. The Application Builder provides two important tools for creating applications: The Form editor and the Method editor. In addition, an application can have a menu bar or a ribbon. The Form editor includes drag-and-drop capabilities for user interface components such as input fields, graphics objects, and buttons. The Method editor is a programming environment that allows you to modify the data structures that represent the different parts of a model. The pictures below show the Model Builder and Application Builder windows.



When creating an application, you typically start from an existing model. However, you can just as well build an application user interface and the underlying model simultaneously. You can easily, at any time, switch between the Model Builder and Application Builder. The model part of an application, as represented by the model tree, is called an embedded model.

The tools in the Application Builder can access and manipulate the settings in the embedded model in several ways; For example:

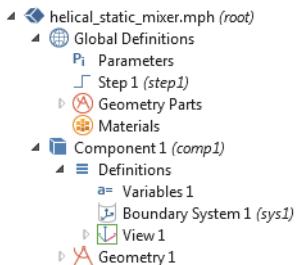
- If the model makes use of parameters and variables, you link these directly to input fields in the application by using the New Form wizard or Editor

Tools. In this way, the user of an application can directly edit the values of the parameters and variables that affect the model. For more information, see pages 46 and 70.

- By using the New Form wizard or Editor Tools, you can include a button in your application that runs a study node and thereby starts the solver. In addition, you can use this wizard to include graphics, numerical outputs, check boxes, and combo boxes. For more information, see pages 32 and 46.
- The Model Data Access tool and the Editor Tools window can be used to directly access low-level settings in the model for use with form objects or in methods. For more information, see pages 46, 80, and 132.
- By using the Record Code tool, you can record the commands that are executed when you perform operations within the model tree and its nodes. These will then be available in a method for further editing. For more information, see page 136.

Parameters, Variables, and Scope

The model tree may contain both parameters and variables that are used to control the settings of a model. The picture below shows the model tree of an application with nodes for both **Parameters** and **Variables**.



Parameters are defined under the **Global Definitions** node in the model tree and are user-defined constant scalars that are usable throughout the Model Builder. That is to say, they are “global” in nature. Important uses are:

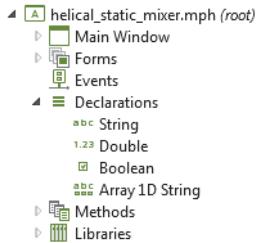
- Parameterizing geometric dimensions.
- Specifying mesh element sizes.
- Defining parametric sweeps.

Variables can be defined in either the **Global Definitions** node or in the **Definitions** subnode of any model **Component** node. A globally defined variable can be used throughout a model whereas a model component variable can only be used within

that component. Variables can be used for spatially or time-varying expressions, including dependent field variables for which you are solving.

In the Model Builder, a parameter or variable is a string with the additional restriction that its value is a valid model expression. For more information on the use of parameters and variables in a model, see the book *Introduction to COMSOL Multiphysics*.

An application may need additional variables for use in the Form editor and the Method editor. Such variables are declared in the Application Builder under the **Declarations** node in the application tree. The picture below shows the application tree of an application with multiple declarations.



The declared variables in the Application Builder are typed variables, including scalars, arrays, Booleans, strings, integers, and doubles. Before using a variable, you have to declare its type.

The fact that these variables are typed means that they can be used directly in methods without first being converted using one of the built-in methods. This makes it easier to write code with typed variables than with parameters and variables representing model expressions. However, there are several tools available in the Application Builder for converting between the different kinds of variables. For more information, see pages 115 and 264.

Running Applications

With a COMSOL Multiphysics license, applications can be run from the COMSOL Desktop environment. With a COMSOL Server license, applications can be run in major web browsers on a variety of operating systems and hardware platforms. In addition, you can run applications by connecting to COMSOL Server with an easy-to-install COMSOL Client for Windows®.

The following two sections explain how to run applications from the COMSOL Multiphysics and COMSOL Server environments.

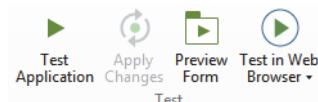
Running Applications in COMSOL Multiphysics

In COMSOL Multiphysics, you run an application using any of these ways:

- Click **Test Application** in the ribbon or in the Quick Access Toolbar.
- Select **Run Application** in the **File** menu or in the Quick Access Toolbar.
- Double-click an MPH file icon on the Windows® Desktop.
- Select **Test in Web Browser** in the ribbon.

TESTING AN APPLICATION

Test Application is used for quick tests. It opens a separate window with the application user interface while keeping the Application Builder desktop environment running.

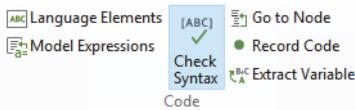


While testing an application, you can apply changes to forms, methods, and the embedded model at run time by clicking the **Apply Changes** button. Not all changes can be applied at run time and in such a case you are prompted to close the application and click **Test Application** again.

To preview the layout of a form without running the application, click **Preview Form** in the ribbon.

When **Test Application** is used, all methods are automatically compiled with the built-in Java® compiler. Any syntax errors will generate error messages and the

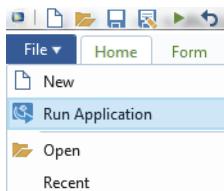
process of testing the application will be stopped. To check for syntax errors before testing an application, click the **Check Syntax** button in the **Method** tab.



Check Syntax finds syntax errors by compiling the methods using the built-in Java® compiler. Any syntax errors will, in this case, be displayed in the **Errors and Warnings** window in the Method editor. For more information, see “The Method Editor” on page 127.

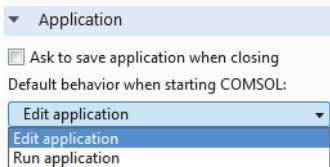
RUNNING AN APPLICATION

Run Application starts the application in the COMSOL Desktop environment. Select **Run Application** to use an application for production purposes. For example, you can run an application that was created by someone else that is password protected from editing, but not from running.



DOUBLE-CLICKING AN MPH FILE

When you double-click an MPH file icon on the Windows® Desktop, the application opens in COMSOL Multiphysics, provided the MPH file extension is associated with COMSOL Multiphysics. The application may either be opened for editing or for running. You control this behavior from the root node of the application tree. The **Settings** window for this node has a section titled **Application** in which you may select either **Edit application** or **Run application**. A change in this setting will be applied when you save the MPH file.



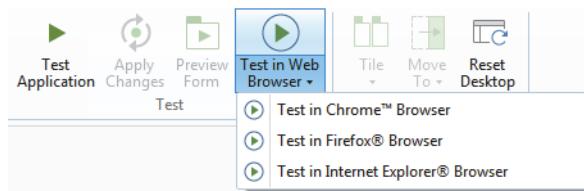
The option **Edit application** will open the application in the Application Builder.

The option **Run application** will open the application in runtime mode for production purposes. This option is similar to selecting **Run Application** in the **File** menu with the difference that double-clicking an MPH file will start a new COMSOL Multiphysics session.

If you have installed the COMSOL Client for Windows®, the MPH file extension may instead be associated with the COMSOL Client, and double-clicking an MPH file will prompt you to log in to a COMSOL Server installation.

TESTING AN APPLICATION IN A WEB BROWSER

Test in Web Browser is used for testing the application in a web browser. This functionality makes it easy to test the look-and-feel of the application when it is accessed from a web browser connected to a COMSOL Server installation.



You can choose which of the installed web browsers you would like the application to launch in. **Test in Web Browser** opens a separate browser window with the application user interface while keeping the Application Builder desktop environment running.

TEST APPLICATION VS. TEST IN WEB BROWSER

Test Application launches the application with a user interface based on Microsoft® .NET Framework components whereas **Test in Web Browser** launches the application with a user interface based on HTML5 components. **Test Application** will display the user interface as it would appear when the app is run with COMSOL Multiphysics or COMSOL Server, provided the COMSOL Client for Windows® is used to connect with the COMSOL Server installation. **Test in Web Browser** will display the user interface as it would appear when the app is run with COMSOL Server, provided a web browser is used to connect with the COMSOL Server installation.

For testing the appearance and function of an application user interface in web browsers for OS X, iOS, Linux®, and Android™, a COMSOL Server installation is required.

The table below summarizes the different options for running an application.

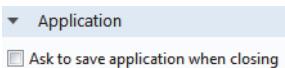
SERVER	CLIENT
COMSOL Multiphysics	Test Application
COMSOL Multiphysics	Test in Web Browser
COMSOL Multiphysics	Run Application
COMSOL Server	COMSOL Client for Windows®
COMSOL Server	Web Browser

The Server column represents the software components that perform the CPU-heavy computations. The Client column represents the software components used to present the application user interface.

SAVING A RUNNING APPLICATION

When you test an application, it is assigned the name `Untitled.mph` and is a copy of the original MPH file. This is not the case when running an application.

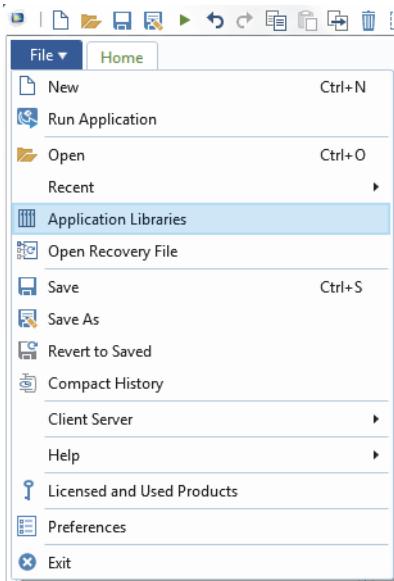
By default, the user of an application will not be prompted to save changes when exiting the application. You control this behavior from the root node of the application tree. The **Settings** window for this node has a section titled **Application** in which you may select the check box **Ask to save application when closing**, as shown in the figure below.



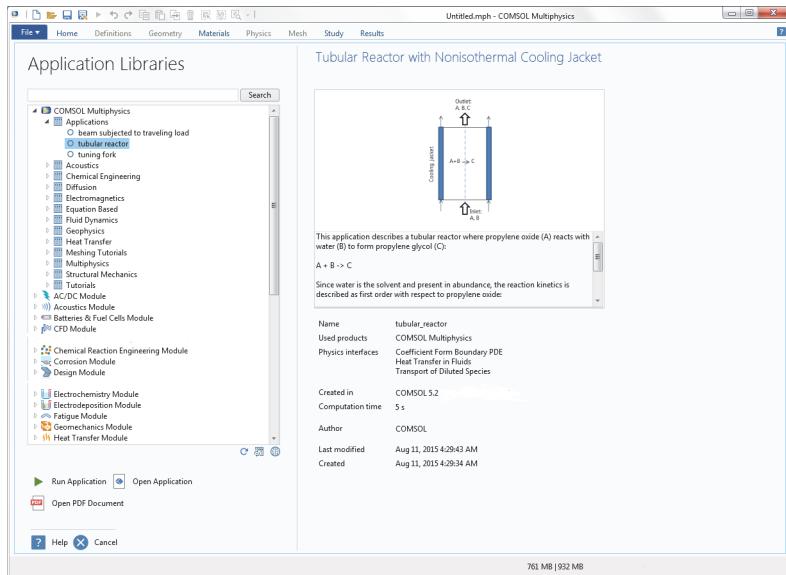
As an alternative, you can add a button or menu item with a command to save the application. For more information, see page 106.

APPLICATION LIBRARIES

From the **File** menu, select **Application Libraries** to run and explore the example applications that are included in the COMSOL installation. Many of the screenshots in this book are taken from these examples.



You run an application, or open it for editing, by clicking the corresponding buttons below the Application Libraries tree.



Applications that contain a model, but no additional forms or methods, cannot be run and only opened for editing. Applications that contain forms and methods are collected in folders named **Applications**.

The applications in the Application Libraries are continuously improved and updated. You can update the Application Libraries by clicking Update COMSOL Application Library below the tree.



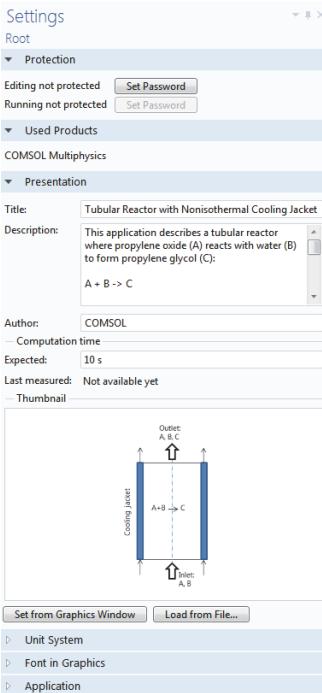
Additional applications that are not part of the Application Libraries may be available from the COMSOL website in the Application Gallery. To find these applications, click the Application Gallery button below the tree. This will open a browser with the web page for the Application Gallery.



Each application has an associated thumbnail image that is displayed in the Application Libraries. In the COMSOL Server web interface, the thumbnail image is displayed on the Application Library page.

To set the thumbnail image, click the root node of the application tree. The **Settings** window has two options: **Set from Graphics Window** and **Load from File**.

The **Load from File** option allows you to load images in the PNG or JPG file formats. Choose an image size from 280-by-210 to 1024-by-768 pixels to ensure that the image displays properly as a thumbnail in COMSOL Multiphysics and COMSOL Server.



The **Set from Graphics Window** option automatically creates two thumbnail images:

- An image of size 280-by-210 pixels shown in the **Settings** window of the application tree root node and in the Application Libraries.
 - An image of size 1024-by-768 used as the default title page image in reports and in the Application Libraries in COMSOL Server.

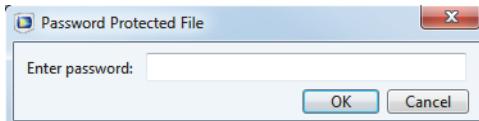
PASSWORD PROTECTION

An application can be password protected to manage permissions. You assign separate passwords for editing and running in the **Settings** window, accessible by clicking the root node of the application tree in the Application Builder window.

You must have permission to edit an application in order to create passwords for running it.



When you open a password-protected MPH file, for editing or running, a dialog box prompts you for the password:



To remove password protection, create an empty password.

The password protection is used to encrypt all model and application settings, including methods. However, binary data such as the finalized geometry including embedded CAD files, mesh data, and solution data are not encrypted.

SECURITY SETTINGS

When creating an application with the Application Builder, it is important to consider the security of the computer hosting the application. Both COMSOL Multiphysics and COMSOL Server provide a very similar set of security settings for controlling whether or not an application should be allowed to perform external function calls, contain links to C libraries, run MATLAB functions, access external processes, etc.

The security settings in COMSOL Multiphysics can be found in the **Security** page in the **Preferences** window accessed from the **File** menu. In COMSOL Server, they are available in the Preferences page in the COMSOL Server web interface if you

are logged in as an administrator. If you are not sure what security settings to use, contact your systems administrator.

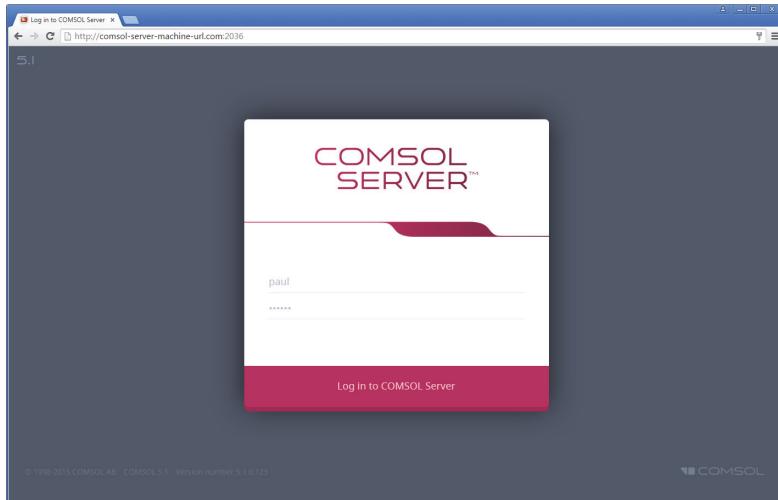
Running Applications in COMSOL Server

COMSOL applications can be run by connecting to COMSOL Server from a web browser, or from a COMSOL Client for Windows®. The COMSOL Client for Windows® allows a user to run applications that require a LiveLink™ product for CAD, as described in “Running Applications in the COMSOL Client”.

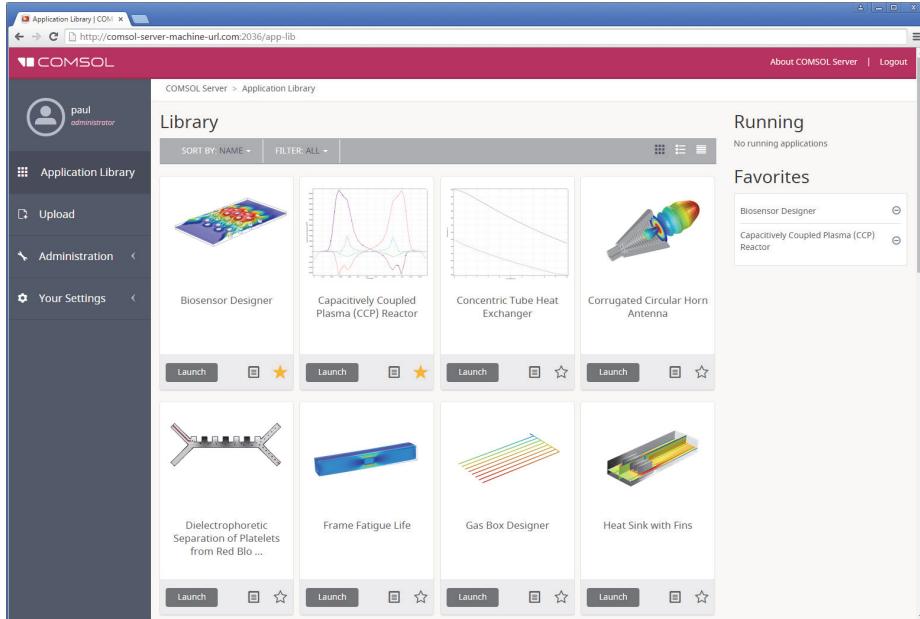
Running applications in a web browser does not require any installation and no web browser plugins are needed. Running an application in a web browser supports interactive graphics in 1D, 2D, and 3D. In a web browser, graphics rendering in 3D is based on WebGL™ technology, which is included with all major web browsers.

RUNNING APPLICATIONS IN A WEB BROWSER

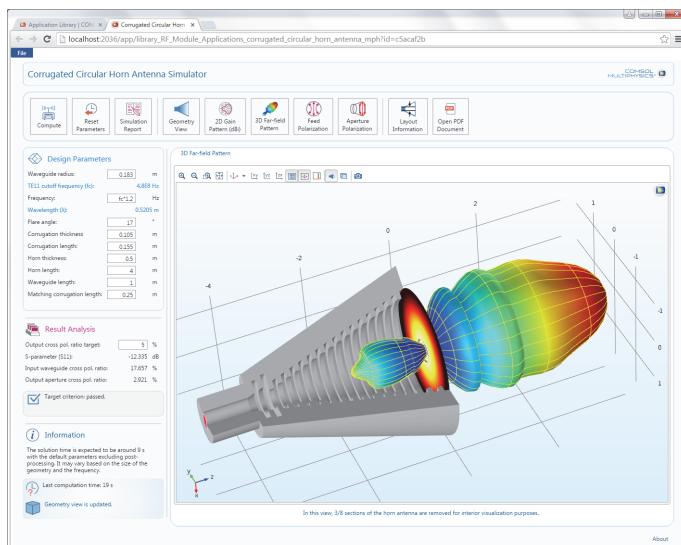
Using a web browser, you can point directly to the computer name and port number of a COMSOL Server web interface—for example, <http://comsol-server-machine-url.com:2036>, assuming that port number 2036 is used by your COMSOL Server installation. You need to provide a username and password to log in.



When logged in, the **Application Library** page displays a list of applications to run.



Click **Launch** to run an application. Applications are run in separate tabs in the browser.



Limitations When Running Applications in Web Browsers

When you create applications for running in a web browser, make sure you use the grid layout mode in the Application Builder, see “Sketch and Grid Layout” on page 86. This will ensure that the user interface layout adapts to the size and aspect ratio of the browser window. For low-resolution displays, make sure to test the user interface layout in the target platform to check that all form objects are visible. Applications that contain resizable graphics forms may not fit in low-resolution displays. In such cases, use graphics with fixed width and height to make sure all form objects fit in the target browser window. Depending on the type of web browser and the graphics card, there may be restrictions on how many graphics objects that can be used in an application. You can get around such limitations by, instead of using multiple graphics objects, reuse the same graphics object by switching its source.

When running in a web browser, the LiveLink™ products for CAD software packages are not supported.

When running COMSOL applications in web browsers for smartphones and certain tablets, not all functionality is supported. Typical limitations include the ability to play sounds or open documents. In addition, file upload and download may not be supported.

If the application allows the user to make selections, such as clicking on boundaries to set boundary conditions, running in a web browser is different from running in COMSOL Multiphysics or the COMSOL Client for Windows®. In a web browser, boundaries are not automatically highlighted when hovering. Instead, it is required to click once to highlight a boundary. A second click will make the selection. A third click will highlight for deselection and a fourth click will deselect. The process is similar for domains, edges, and points.

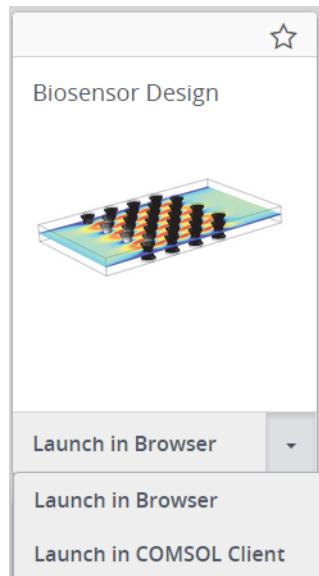
RUNNING APPLICATIONS IN THE COMSOL CLIENT

As an alternative to using a web browser for running applications, the COMSOL Client for Windows® can be used to connect to COMSOL Server for running applications natively in the Windows® operating system. This typically gives better graphics performance and supports more sophisticated graphics rendering in 1D, 2D, and 3D. In addition, the COMSOL Client for Windows® allows running applications that require a LiveLink™ product for CAD, provided that the

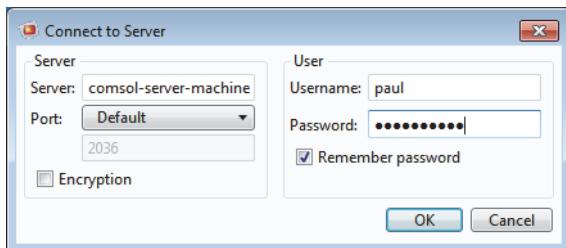
COMSOL Server you connect to has the required licenses. You can open an application with the COMSOL Client for Windows® in two different ways:

- The COMSOL Server web interface will allow you to choose between launching an application in a web browser or with the COMSOL Client for Windows®.

If you try to launch an application with the COMSOL Client in this way, but it is not yet installed, you will be prompted to download and install it.

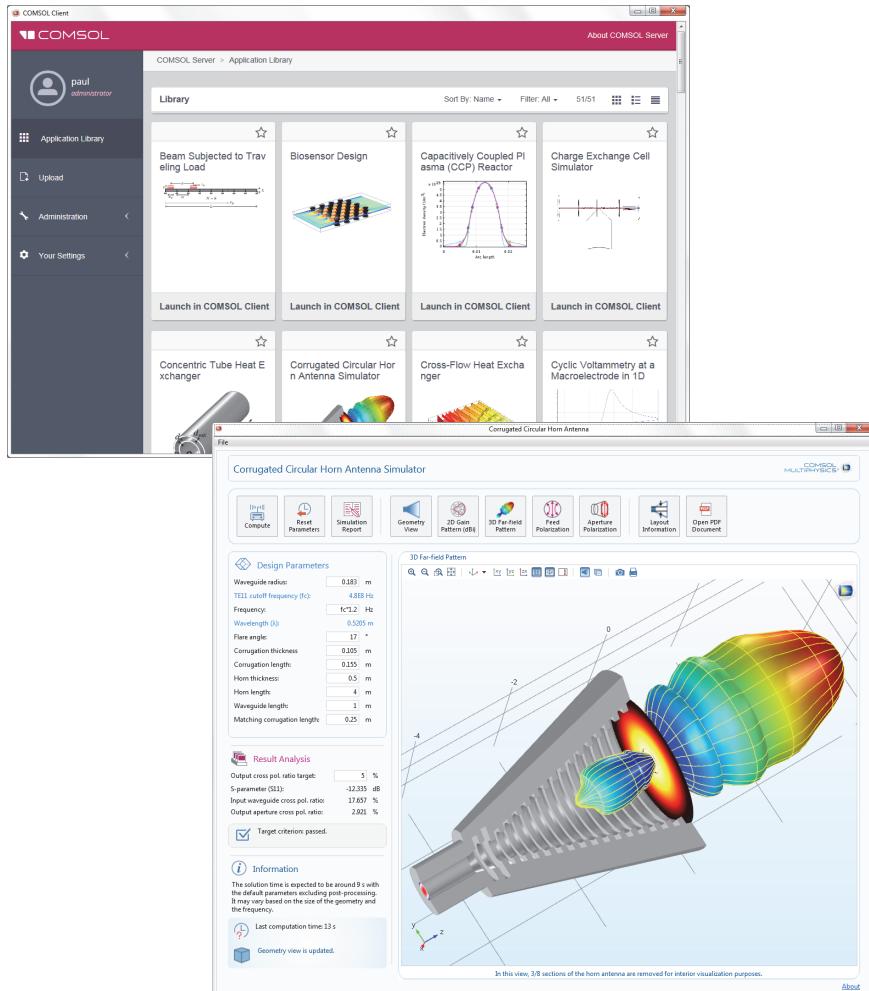


- If you have the COMSOL Client for Windows® already installed, a desktop shortcut will be available. You can double-click its desktop icon and before you can use the COMSOL Client to run applications, you must log into a COMSOL Server with a valid username and password. After login, the COMSOL Client displays a COMSOL Server web interface identical to that seen when logging in from a web browser.



Using the COMSOL Client, applications run as native Windows® applications in separate windows. For example, applications run in the COMSOL Client may have a Windows® ribbon with tabs. When run in a web browser, ribbons are represented by a toolbar.

In the figure below, the COMSOL Server web interface is shown (top) with an application launched in the COMSOL Client for Windows® (bottom).



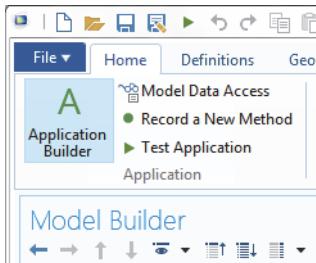
For more information on running applications with COMSOL Server, see the *COMSOL Server Manual* available with a COMSOL Server installation or from http://www.comsol.com/documentation/COMSOL_ServerManual.pdf.

Getting Started with the Application Builder

STARTING FROM A COMSOL MULTIPHYSICS MODEL

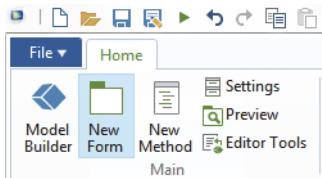
If you don't have a model already loaded to the COMSOL Desktop environment, select **File>Open** to select an MPH file from your file system or select a file from the Application Libraries. Note that the files in the **Applications** folders are ready-to-use applications. All other files in the Application Libraries contain a model and documentation, but not an application user interface.

Once the model is loaded, click the **Application Builder** button in the ribbon Home tab. This will take you to the Application Builder desktop environment.



CREATING A NEW FORM

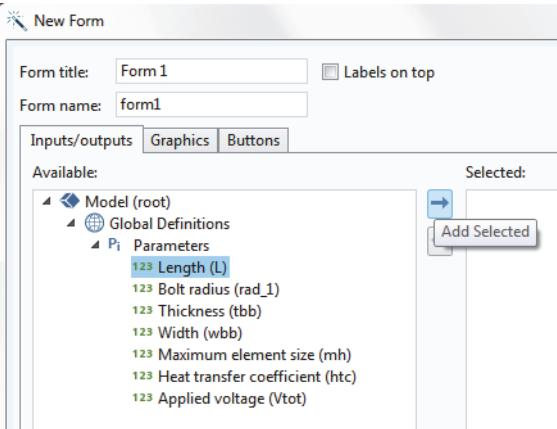
To start working on the user interface layout, click the **New Form** button in the **Home** tab. This will launch the **New Form** wizard.



The **New Form** wizard assists you with adding the most common user interface components, so-called form objects, to the first form of your application. It has three tabs:

- **Inputs/outputs**

- **Graphics**
- **Buttons**



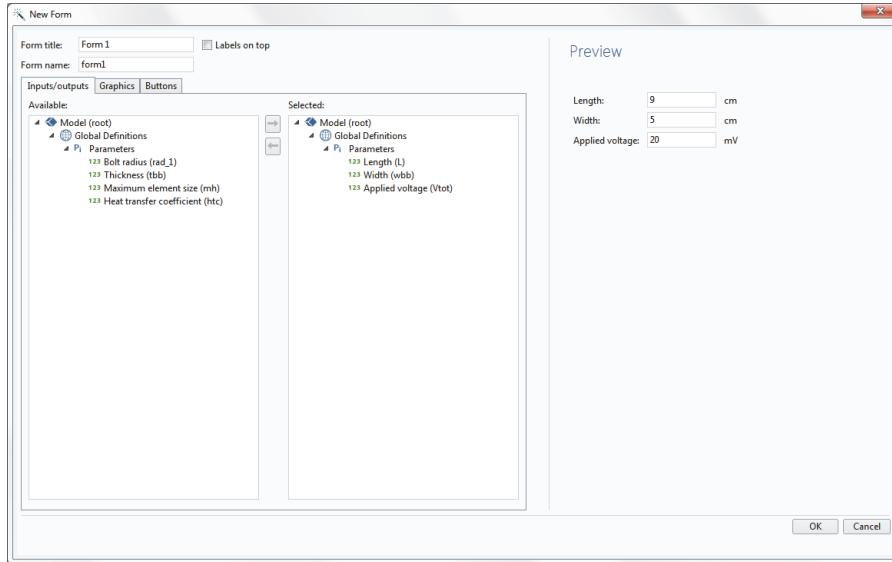
Double-click a node or click the **Add Selected** button to move a node from the **Available** area to the **Selected** area. The selected nodes will become form objects in the application, and a preview of the form is shown in the **Preview** area to the right. The size as well as other settings for form objects can be edited after exiting the wizard. At the top of the wizard window you can change the name and title of the form. For details see “The Individual Form Settings Windows” on page 38.

You can also choose to exit the **New Form** window at this stage by clicking **Done**, and then manually add form objects.

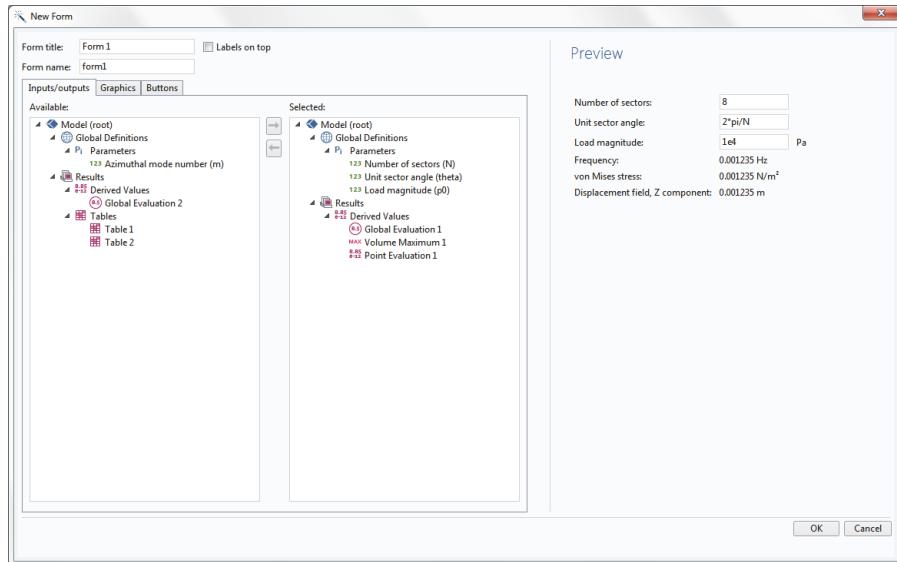
The Inputs/Outputs Tab

The **Inputs/outputs** tab displays the model tree nodes that can serve as an input field, a data display object, a check box, or a combo box. Input fields added by the wizard will be accompanied by a text label and a unit, when applicable. You can make other parts of the model available for input and output by using **Model Data Access** (see page 80). Check box and combo box objects are only available in this way. For example, you can make the **Predefined** combo box for **Element Size** under the **Mesh** node available in the wizard by enabling it with **Model Data Access**.

In the figure below, three parameters including Length, Width, and Applied voltage have been selected to serve as input fields:



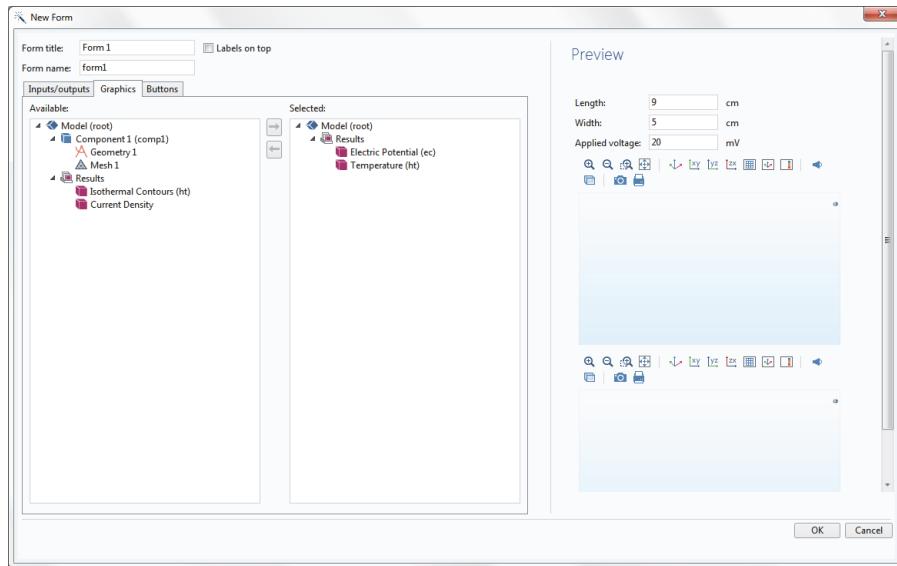
In the figure below, corresponding to a different model, three **Derived Values** nodes have been selected to serve as data display objects:



After exiting the wizard, you can edit the size and font color as well as other settings for input fields and data display objects.

The *Graphics* Tab

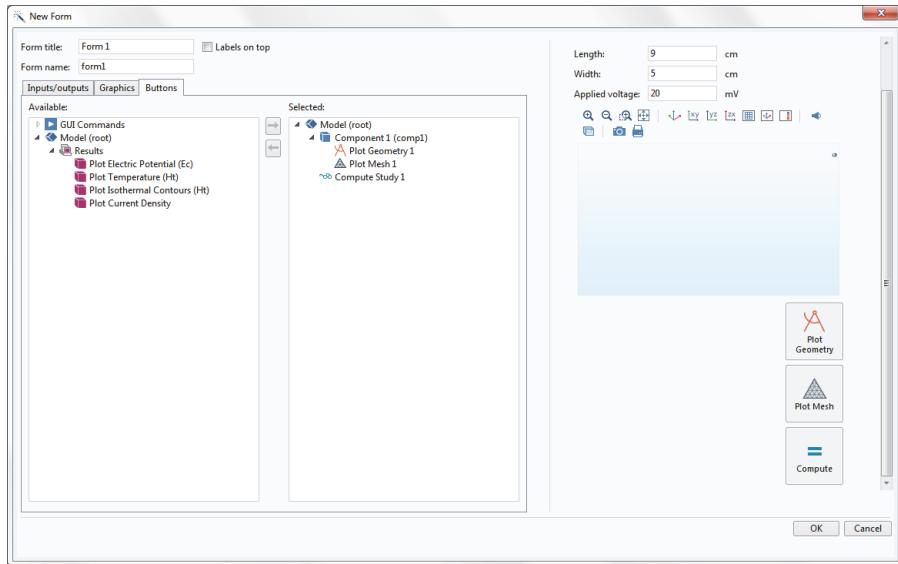
The **Graphics** tab displays the model tree nodes that can serve as graphics objects: **Geometry**, **Selection**, **Mesh**, and **Results**. In the figure below, two such nodes have been selected.



The *Buttons* Tab

The **Buttons** tab displays the model and application tree nodes that can be run by clicking a button in the application user interface. Examples of such tree nodes are **Plot Geometry**, **Plot Mesh**, **Compute Study**, and each of the different plot groups under **Results**. In addition, you can add buttons for **GUI Commands**, **Forms**, and **Methods**.

In the figure below, three buttons have been added: **Plot Geometry**, **Plot Mesh**, and **Compute**.



Using the Form editor, you can add buttons that run your own custom command sequences or methods.

EXITING THE WIZARD

Click **OK** to exit the wizard. This automatically takes you to the Form editor.

SAVING AN APPLICATION

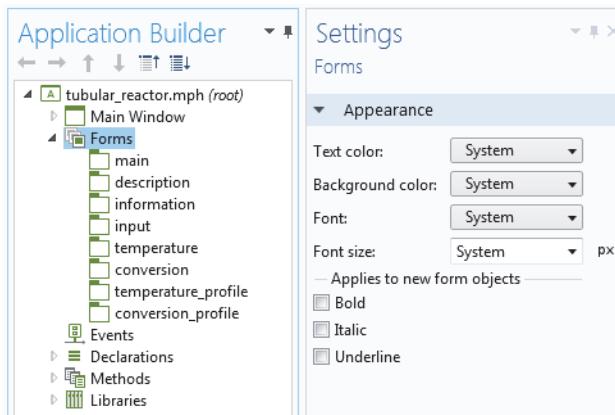
To save an application, from the **File** menu, select **File>Save As**. Browse to a folder where you have write permissions, and save the file in the MPH file format. The MPH file contains all of the information about the application, including information about the embedded model created with the Model Builder.

The Form Editor

Use the Form editor for user interface layout to create forms with form objects such as input fields, graphics, buttons, and more.

The Forms Settings Window

The **Settings** window for forms is displayed when you click the **Forms** node in the application tree. It lets you change the overall appearance of forms with settings for **Text color**, **Background color**, **Font**, **Font size**, **Bold**, **Italic**, and **Underline**.

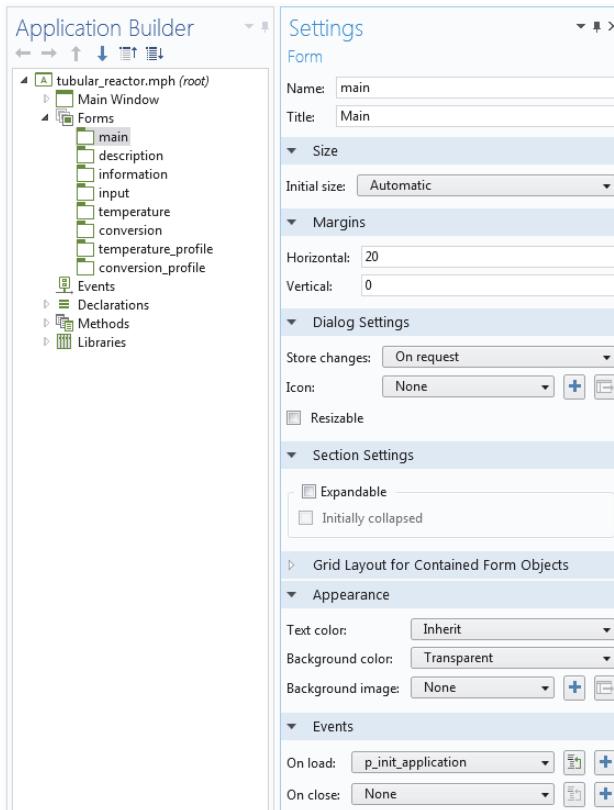


The default is that all new forms and new form objects inherit these settings when applicable.

In the figure above, and in some of the figures below, the **Settings** window is docked to the right of the **Application Builder** window. By default, the **Settings** window is docked to the far right in the Application Builder desktop environment.

The Individual Form Settings Windows

The figure below shows the **Settings** window for a form.

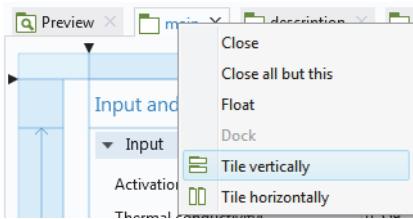


Each form has its own **Settings** window with settings for:

- **Name** used to reference the form in other form objects and methods.
- Form **Title** that is used in applications with several forms.
- **Initial size** of the form when used as a dialog box or when the **Main Window** is set to have its size determined by the form.
- **Margins** with respect to the upper-left corner (**Horizontal** and **Vertical**).
- Choices of when to store changes in dialog boxes (**Store changes**), see also “Showing a Form as a Dialog Box” on page 56.
- **Icon** shown in the upper left corner of a dialog box.

- Choices of whether the form should be **Resizable** or not when used as a dialog box.
- Choices of whether to view sections as **Expandable** and whether they should be **Initially collapsed** (**Section Settings**).
- Table with the formatting of all columns and rows included in the form (**Grid Layout for Contained Form Objects**).
- **Appearance** with settings for **Text color**, **Background color**, and **Background image**.
- **Events** that are triggered when a form is loaded or closed. (**On load** and **On close**.)

Double-click a form node to open its window in the Form editor. Alternatively, you can right-click a form node and select **Edit**. Right-click a form window tab to see its context menu with options for closing, floating, and tiling form windows.



SKETCH AND GRID LAYOUT MODES

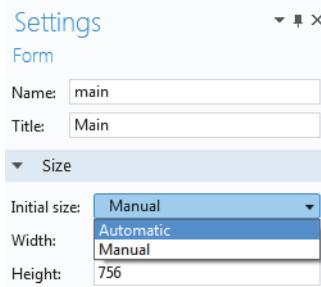
The Application Builder defaults to sketch layout mode, which lets you use fixed object positions and size. The instructions in the section “The Form Editor” assume that the Form editor is in sketch layout mode unless otherwise specified. For information on grid layout mode, see “Sketch and Grid Layout” on page 86.

INITIAL SIZE OF A FORM

There are two options for the initial size of a form:

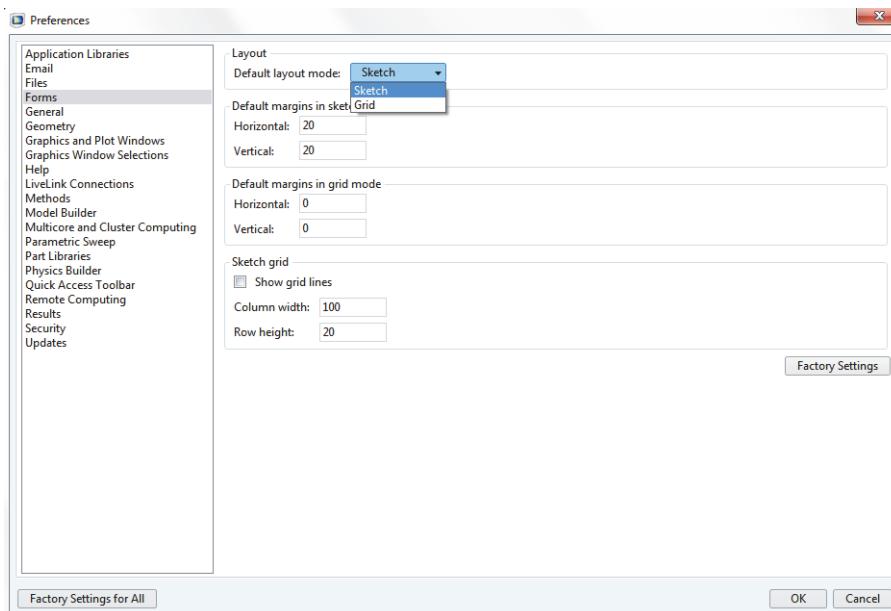
- **Manual** lets you enter the pixel size for the width and height.
- **Automatic** determines the size based on the form objects that the form contains. If you are using grid layout mode and there are columns or rows set to **Grow**, then the size is not defined by the form objects. In this case, the size is estimated using the form editor grid size as a base point. (It will typically be slightly larger.) You can change the grid size by dragging the

right or bottom border of the grid. For more information on grid layout mode, see “Grid Layout” on page 88.



Form Editor Preferences

To access **Preferences** for the Form editor, choose **Preferences** from the **File menu** and select the **Forms** page.



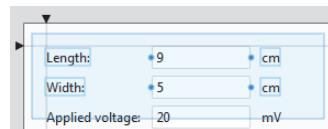
The **Forms** section includes settings for changing the defaults for **Layout mode**, **Margins**, and **Sketch grid**.

Form Objects

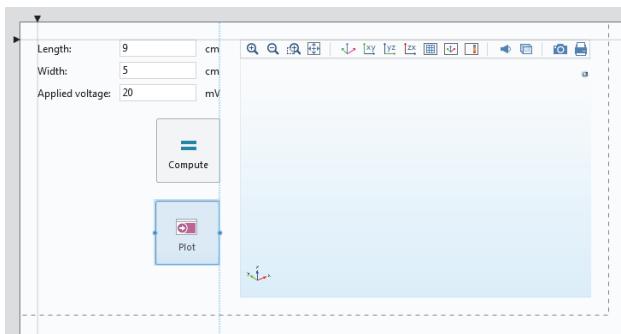
POSITIONING FORM OBJECTS

You can easily change the positioning of form objects such as input fields, graphics objects, and buttons:

- Click an object to select it. A selected object is highlighted with a blue frame.
- To select multiple objects, use Ctrl+click. You can also click and drag to create a selection box in the form window to select all objects within it.
- Hold and drag to move to the new position. Blue guidelines will aid in the positioning relative to other objects.
- In sketch layout mode, you can also use the keyboard arrow keys to move objects. Use Ctrl+arrow keys to fine-tune the position.



In the figures below, a **Plot** button is being moved from its original position. Blue guide lines show its alignment relative to the unit objects and the **Compute** button.



RESIZING FORM OBJECTS

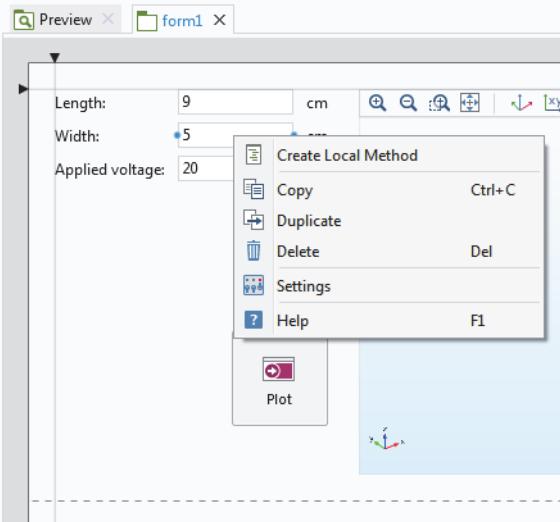
To resize an object:

- Click an object to select it.
- Hold and drag one of the handles, shown as blue dots, of the highlighted blue frame. If there are no handles, this type of form object cannot be resized.

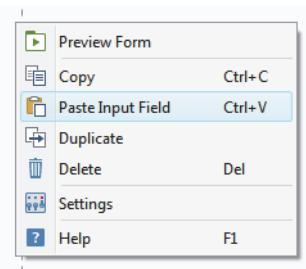
COPYING, PASTING, DUPLICATING, AND DELETING AN OBJECT

To delete an object, click to select it and then press Delete on your keyboard. You can also click the delete button in the Quick Access Toolbar.

You can copy-paste an object by pressing Ctrl+C and Ctrl+V. Alternatively, you can right-click an object to get menu options for **Copy**, **Duplicate**, **Delete**, and more.



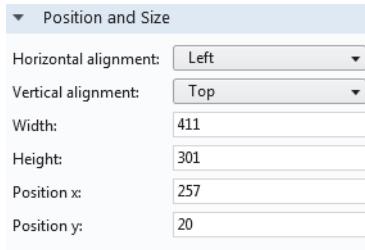
To paste an already copied object, right-click in an empty area in the form and right-click. Depending on the copied object, a **Paste** menu option will be shown. In the figure below, an **Input Field** has previously been copied and as a result a **Paste Input Field** option is shown.



ADJUSTING POSITION AND SIZE BY THE NUMBER OF PIXELS

When in sketch layout mode, you can adjust the position and size of an object by typing the number of pixels in the **Position and Size** section of its **Settings** window:

- Click an object to select it. Make sure its **Settings** window is shown. If not, double-click the object or click the **Settings** button in the **Form** tab.
- Edit the numbers in the **Position and Size** section.

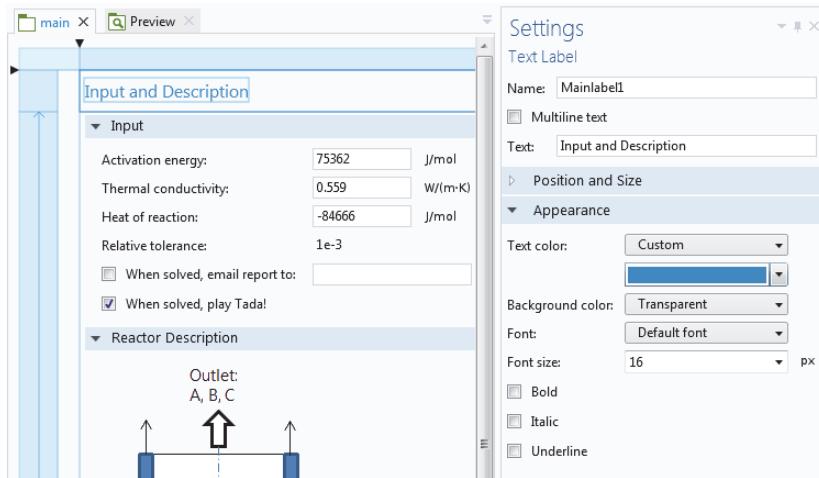


The Position and Size section will have different options depending on the type of form object. For grid layout mode, there are additional settings for the position of the object with respect to rows and columns. For details, see “Sketch and Grid Layout” on page 86.

CHANGING THE APPEARANCE OF DISPLAYED TEXT

For form objects that display text, the **Appearance** section in the **Settings** window lets you change properties such as the text displayed, its font, font color, and font size. For some form objects such as a button, the size of the object will adapt to the length of the text string.

In the figure below, the **Settings** window for a text label object is shown where the font size and color is changed.



SELECTING MULTIPLE FORM OBJECTS

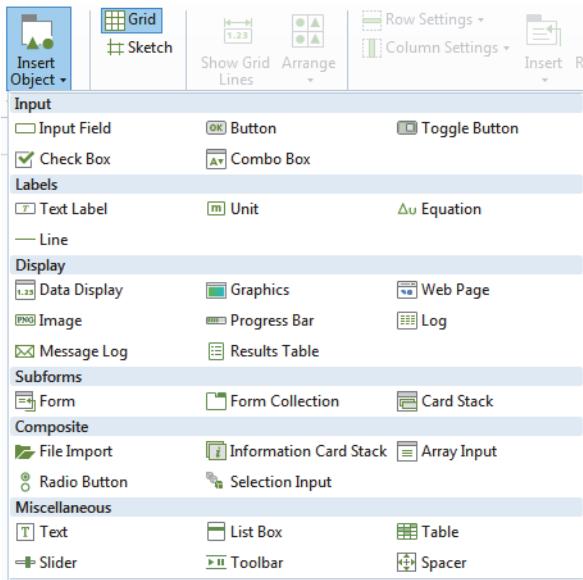
If you select more than one form object, for example, by using **Ctrl+click**, then the **Settings** window will contain a set of properties that can be shared between the selected objects. Shared properties will always originate from the **Appearance** section, the **Position and Size** section, or the **Events** section.

THE NAME OF A FORM OBJECT

A form object has a **Name**, which is a text string without spaces. The string can contain letters, numbers and underscore. In addition, the reserved names **root** and **parent** are not allowed. The **Name** string is used in other form objects and methods to reference the object. The path to the object is shown as a tooltip when hovering over the **Name** field in the **Settings** window.

INSERTING FORM OBJECTS

You can insert form objects in addition to those created by the **New Form** wizard. In the **Form** ribbon tab, select the **Insert Object** menu to see a listing of all available objects.



The remainder of this section “The Form Editor” describes only the types of form objects that are added by the **New Form** wizard. The form objects added by using the wizard may include:

- **Button**
- **Graphics**
- **Input Field**
- **Text Label** (associated with Input Field)
- **Unit** (associated with Input Field)
- **Data Display**

However, when using **Model Data Access** (see page 80), the following form objects may also be added:

- **Check Box**
- **Combo Box**

For more information on the check box, combo box, and other form objects, see “Appendix A—Form Objects” on page 164.

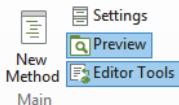
EVENTS AND ACTIONS ASSOCIATED WITH FORM OBJECTS

You can associate objects such as buttons, menu items, ribbon buttons, forms, and form objects with actions triggered by an event. An action can be a sequence of commands including global methods or local methods. Local methods are not accessible or visible outside of the objects where they are defined. The events that can be associated with an object depend on the type of object and include: button click, keyboard shortcut, load of a form (**On load**), close of a form (**On close**), and change of the value of a variable (**On data change**).

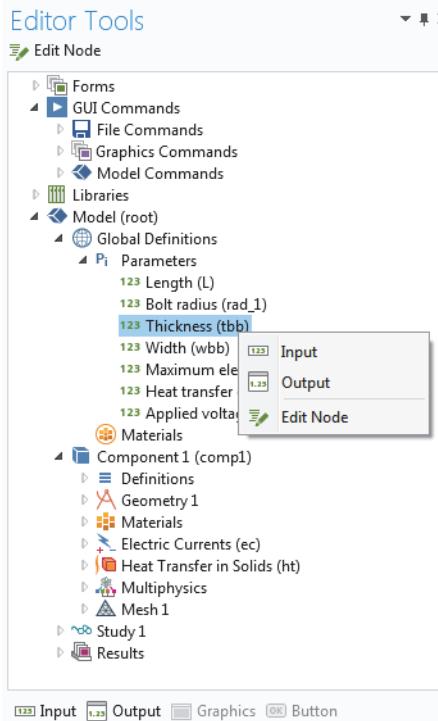
Using Ctrl+Alt+click on a form object opens its local method in the **Method** editor. If there is no method associated with the form object, a new local method will be created, associated with the form object, and opened in the **Method** editor.

Editor Tools in the Form Editor

The **Editor Tools** window is an important complement to the **New Form** wizard and the **Insert Object** menu for quickly creating composite form objects. To display the **Editor Tools** window, click the corresponding button in the **Main** group in the **Form** tab.



You can right-click the nodes in the editor tree to add the same set of form objects available with the **New Form** wizard.



When a node is selected, the toolbar below the editor tree shows the available options for inserting an object.

Depending on the node, the following options are available:

- **Input**

An **Input Field** or a **Combo Box** is inserted as follows:

- Inserts an **Input Field** using the selected node as **Source**. It is accompanied by a **Text Label** and a **Unit** object, when applicable.
- Inserts a **Combo Box** using the selected node as **Source**. A choice list is automatically created, corresponding to the list in the node. This option is only available when used with **Model Data Access** (see page 80) to make the corresponding node available in the editor tree.

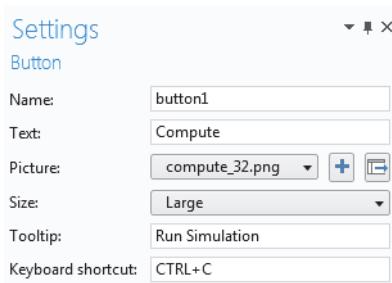
- **Output**
 - Inserts a **Data Display** object accompanied by a **Text Label** when applicable.
- **Button**
 - Inserts a **Button** object with a command sequence running the selected node.
- **Graphics**
 - Inserts a **Graphics** object using the selected node as **Source for Initial Graphics Content**.
- **Edit Node**
 - Brings you to the **Settings** window for the corresponding model tree node.

The Editor Tools window is also an important tool when working with the Method editor. In the Method editor it is used to generate code associated with the nodes of the editor tree. For more information, see “Editor Tools in the Method Editor” on page 132.

Button

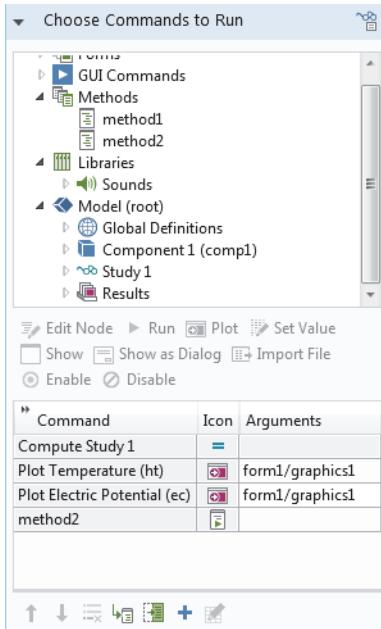
Clicking on a **Button** is an event that triggers an action defined by its command sequence. The main section of the **Settings** window for a button allows you to:

- Edit the form object **Name** of the button.
- Edit the **Text** displayed on the button.
- Use a **Picture** instead of the default rendering of a button.
- Change the button **Size** from **Large** to **Small**.
- Add a **Tooltip** with text that is shown when hovering over the button.
- Add a **Keyboard shortcut** by clicking the input field and entering a combination of the modifier keys Shift, Ctrl, and Alt together with another keyboard key. Alt must be accompanied by at least one additional modifier.



CHOOSING COMMANDS TO RUN

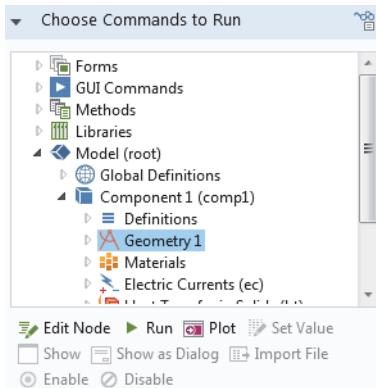
The section **Choose Commands to Run** lets you control the action associated with a button-click event. The figure below shows the **Settings** window for a button that triggers a sequence of four commands.



A menu, ribbon, or toolbar item will also provide a **Choose Commands to Run** section in its **Settings** window, and the functionality described in this section applies. For more information on using menu, ribbon, and toolbar items, see “Graphics Toolbar” on page 65, “The Main Window” on page 102, “Table” on page 229, and “Toolbar” on page 236.

To define a sequence of commands, in the **Choose Commands to Run** section, select a node in the editor tree. Then click one of the highlighted buttons under the tree, or right-click and select the command. In the figure below, the **Geometry** node is selected and the available commands **Run** and **Plot** are highlighted. Click **Run** to add a geometry-building command to the command sequence. Click **Plot** to add

a command that plots the geometry. The option **Edit Node** will take you to the corresponding node in the model tree or the application tree.

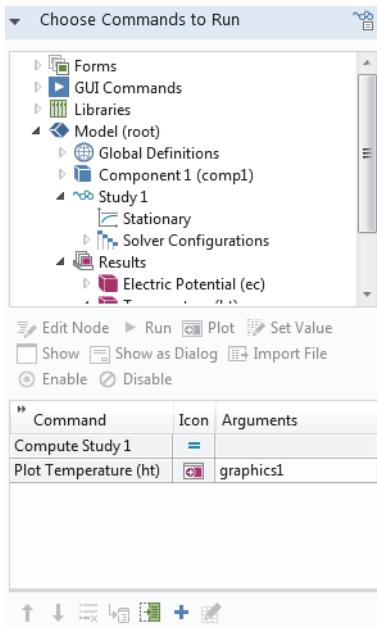


The command icons highlighted for selection are those applicable to the particular tree node. This is a list of the command icons that may be available, depending upon the node:

- **Run**
- **Plot**
- **Set Value**
- **Show**
- **Show as Dialog**
- **Import File**
- **Enable**
- **Disable**

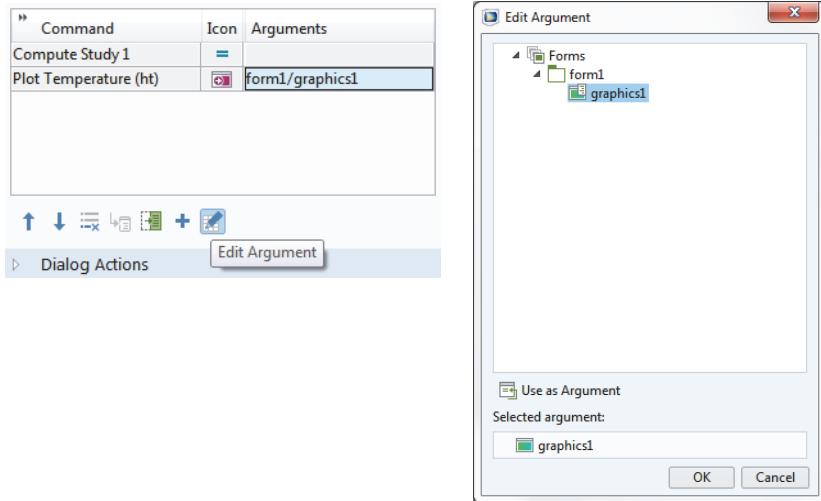
Some commands, such as the various plot commands, require an argument. The argument to a plot command, for example, defines which of the different graphics objects the plot should be rendered in.

The example below shows the **Settings** window and command sequence for a **Compute** button as created by the **New Form** wizard. This button has a command sequence with two commands: **Compute Study 1** and **Plot Temperature**.



The **Plot Temperature** command has one argument **graphics1**.

To add or edit an input argument, click the **Edit Argument** button below the command sequence, as shown in the figure below.

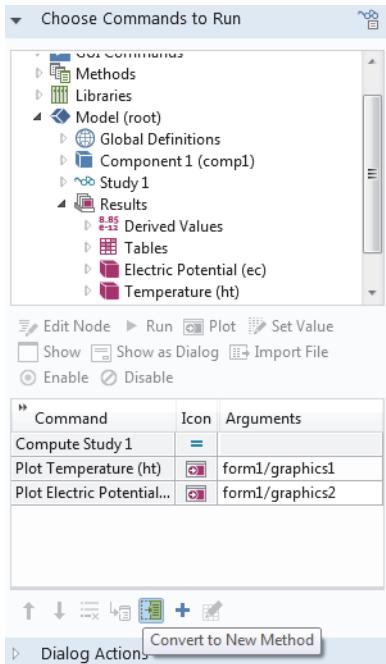


To reference graphics objects in a specific form, the following syntax is used: /form1/graphics2, /form3/graphics1, etc. If a specific form is not specified, for example, graphics1, then the form where the button is located is used.

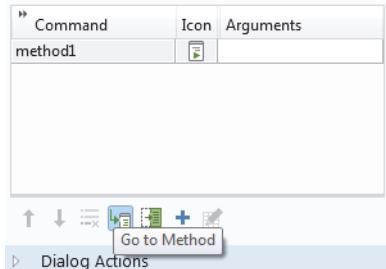
To control the order and contents of the sequence of commands, use the **Move Up**, **Move Down**, and **Delete** buttons located below the command sequence table.

CONVERTING A COMMAND SEQUENCE TO A METHOD

A sequence of commands can be automatically converted to a new method, and further edited in the Method editor, by clicking **Convert to New Method**.

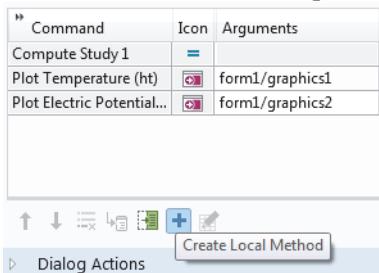


Open the new method by clicking **Go to Method**.





You can also create a method that is local to a form object by clicking **Create Local Method**. These options as shown in the figure below.



The method contains calls to built-in methods corresponding to the commands in the command sequence, as shown in the figure below.

```
1 model.study("std1").run();
2 useGraphics(model.result("pg2"), "form1/graphics1");
3 useGraphics(model.result("pg1"), "form1/graphics2");
```

In this example, the first line:

```
model.study("std1").run()
```

runs the model tree node corresponding to the first study **std1** (the first study node is called **Study 1** unless changed by the user). The second and third lines:

```
useGraphics(model.result("pg2"), "form1/graphics1");
useGraphics(model.result("pg1"), "form1/graphics2");
```

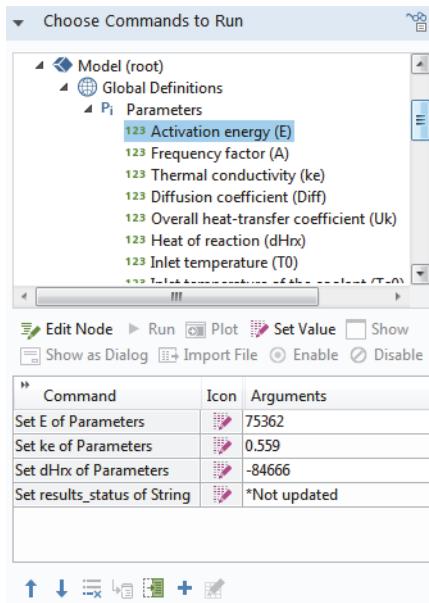
use the built-in method **useGraphics** to display plots corresponding to plot groups pg1 and pg2, respectively. In this example, the plots are displayed in two different graphics objects **graphics1**, and **graphics2**, respectively.

For more information on methods, see “The Method Editor” on page 127.

SETTING VALUES OF PARAMETERS AND VARIABLES

The **Set Value** command allows you to set values of parameters and variables that are available in the **Parameters**, **Variables**, and **Declarations** nodes. In addition, **Set Value** can be used to set the values of properties made accessible by **Model Data**.

Access (see page 80). The figure below shows a command sequence used to initialize a set of parameters and a string variable.

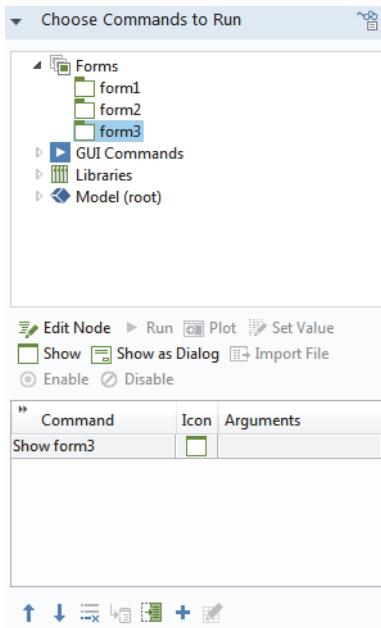


To learn how to perform the same sequence of operations from a method, click the **Convert to New Method** button under the command table.

CHANGING WHICH FORM IS VISIBLE

A button on a form can also be used to display a new form. This can be done in two ways. The first is to use the **Show** command, which will replace the original form with the new form. The second is to use the **Show as Dialog** command. In this case, the new form will pop up as a dialog box over the current form, and will usually be requesting some input from the user.

In the section **Choose Commands to Run**, you can select the **Show** command. The figure below shows the command sequence for a button with a command **Show form3**.

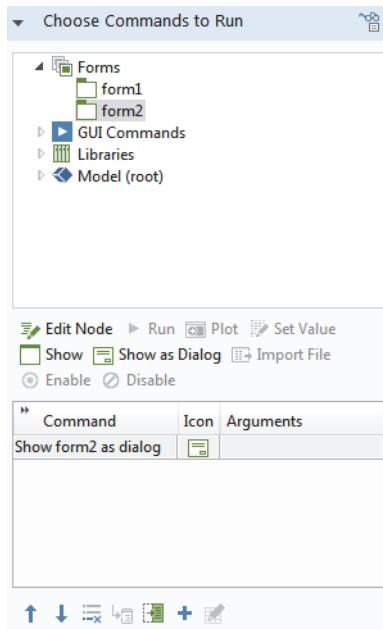


This command will leave the form associated with the button and make the specified form visible to the user.

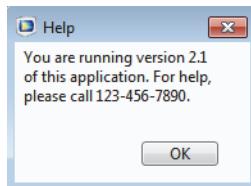
SHOWING A FORM AS A DIALOG BOX

In order to use the **Show as Dialog** command, begin with the **Choose Commands to Run** section and select the form that you would like to show. The figure below

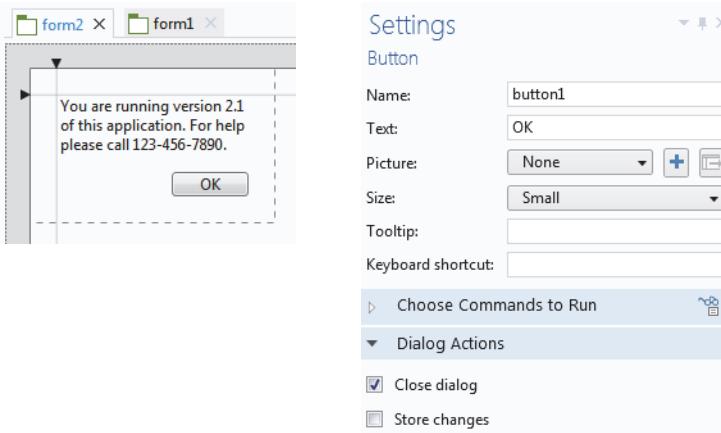
shows an example of the settings for a button with the command **Show form2 as dialog**.



With these settings, clicking the button in the application will launch the following dialog box corresponding to **form2**:



The **form2** window in this example contains a text label object and an **OK** button, as shown in the figure below.



In the **Settings** window, the **Dialog Actions** section has two check boxes:

- **Close dialog**
- **Store changes**

In the example above, the **Close dialog** check box is selected. This ensures that the **form2** window is closed when the **OK** button is clicked. Since **form2** does not have any user inputs, there is no need to select the **Store changes** check box.

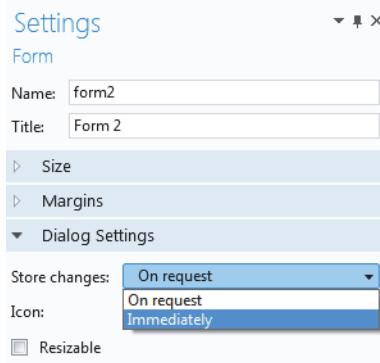
Typical dialog box buttons and their associated dialog actions are:

BUTTON	DIALOG ACTIONS
OK	Close dialog and Store changes
Cancel	Close dialog
Apply	Store changes

A dialog box blocks any other user interaction with the application until it is closed.

In order to control when data entered in a dialog box is stored, there is a list in the **Dialog Settings** section of the **Settings** window of a form where you can select

whether to store data **On request** or **Immediately** when the change occurs, as shown in the figure below.



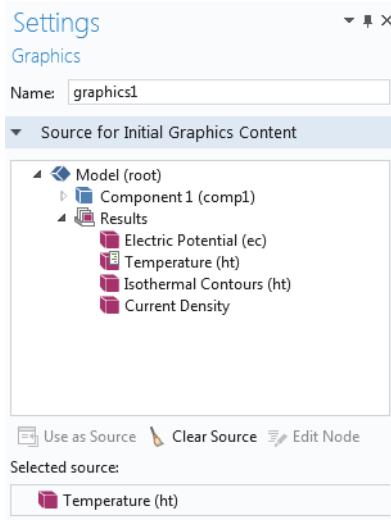
Graphics

Each **Graphics** object gets a default name such as `graphics1`, `graphics2`, etc., when it is created. These names are used to reference graphics objects in command sequences for buttons, menu items, and in methods. To reference graphics objects in a specific form, use the syntax: `/form1/graphics2`, `/form3/graphics1`, etc.

SELECTING THE SOURCE FOR INITIAL GRAPHICS CONTENT

In the **Settings** window for a graphics object, use the section **Source for Initial Graphics Content** to set the plot group to be displayed as default. To select, click **Use as Source** or double-click a node in the tree. If a solution exists for the displayed plot group, the corresponding solution will be visualized when the

application is started. The figure below shows the **Settings** window for a graphics object with a **Temperature** plot selected as the source.

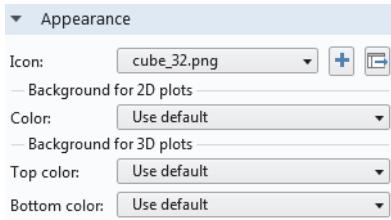


In addition to **Results** plot nodes, you can also use **Selection**, **Geometry**, and **Mesh** nodes as the **Selected source**.

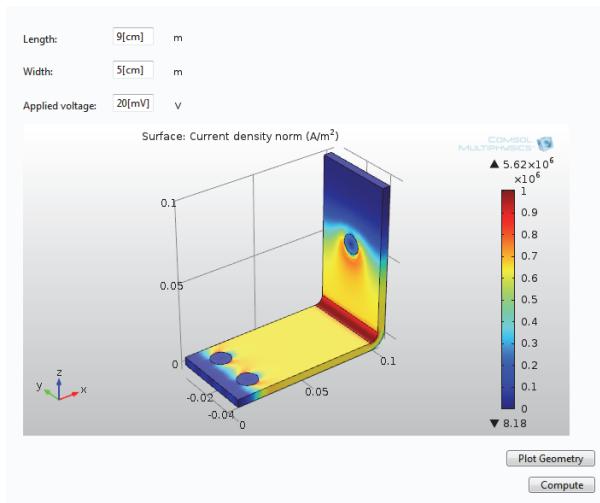
APPEARANCE

For a graphics object, the **Appearance** section of the **Settings** window has the following options:

- Include an **Icon** such as a logo image in the upper-right corner.
- Set the background **Color** for 2D plots.
- Set a flat or graded background color for 3D plots by choosing a **Top color** and **Bottom color**.



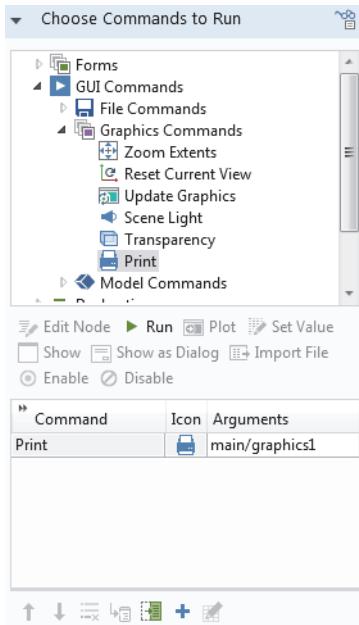
The figure below shows an application where the background **Top color** is set to white and the **Bottom color** to gray. In addition, the standard plot toolbar is not included.



GRAPHICS COMMANDS

In the editor tree used in a command sequence of, for example, a button, the **Graphics Commands** folder contains commands to process or modify a graphics

object. The figure below shows a command sequence with one command for printing the contents of a graphics object.



The available **Graphics Commands** are:

- **Zoom Extents**
 - Makes the entire model visible.
- **Reset Current View**
 - Resets the currently active view to the state it had when the application was launched, see also “Views” on page 68.
- **Update Graphics**
 - Updates the displayed graphics during the execution of a command sequence.
 - Use this when you wish to display several different plots in a sequence, otherwise only the last graphics output of the command sequence is displayed.
- **Scene Light**
 - Toggles the scene light (on or off).

- **Transparency**
 - Toggles the transparency setting (on or off).
- **Print**
 - Prints the contents of the graphics object.

Note that the commands **Zoom Extents**, **Reset Current View**, **Scene Light**, **Transparency**, and **Print** have corresponding toolbar buttons in the standard graphics toolbar, see the next section “Graphics Toolbar”.

Update Graphics and Plot While Solving

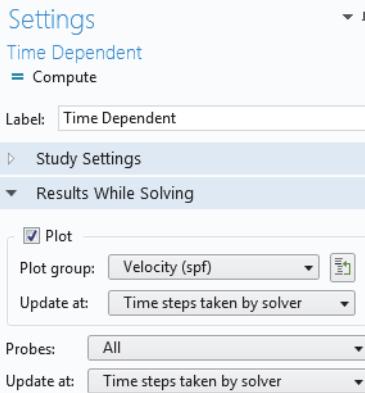
You can use an **Update Graphics** command for briefly displaying graphics information before switching to displaying other types of graphics. To use the **Update Graphics** commands, insert it in a command sequence after a plot command, as shown in the figure below.

Command	Icon	Arguments
Plot Mesh 1		graphics1
Update graphics		
sleep		
Plot Velocity (spf)		graphics1
Compute Study 1	=	

In this example, the mesh is first plotted and then made visible to the user with the **Update graphics** command. The method **sleep** ensures that the mesh plot is visible for a fixed amount of time before the velocity is plotted. For more information on the method **sleep**, see “Sleep” on page 320.

To let the user monitor convergence, you can plot the results while solving. In this example, assume that the **Plot** option is enabled for **Results While Solving**. This

option is available in the **Settings** window of a **Study** node in the model tree, as shown in the figure below.



In the command sequence above, the **Plot Velocity** command comes before the **Compute Study** command. This ensures that the graphics object displays the velocity plot while solving.

USING MULTIPLE GRAPHICS OBJECTS

Due to potential graphics hardware limitations on the platforms where your application will be running, you should strive to minimize the number of graphics objects used. This is to ensure maximum portability of your applications. In addition, if you intend to run an application in a web browser, there may be additional restrictions on how many graphics objects can be used. Different combinations of hardware, operating systems, and web browsers have different limitations.

In this context, two graphics objects with the same name but in different forms count as two different graphics objects. For example, `form1/graphics1` and `form2/graphics2` represent two different graphics objects. In addition, if a graphics object is used in a subform (see “Form” on page 199), then each use of that subform counts as a different graphics object.

To display many different plots in an application, you can, for example, create buttons, toggle buttons, or radio buttons, that simply plot to the same graphics object in a form that doesn’t use subforms.

If you need to use methods to change a plot, use the `useGraphics` command. The example code below switches plot groups by reusing the same graphics object, based on the value of a Boolean variable.

```
if (my_boolean) {  
    useGraphics(model.result("pg1"), "form1/graphics1");
```

```

    my_boolean=!my_boolean; // logical NOT to change between true and false
} else {
    useGraphics(model.result("pg2"), "form1/graphics1");
    my_boolean=!my_boolean;
}

```

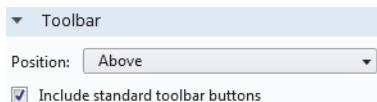
GRAPHICS TOOLBAR

The type of tree node used in the **Source for Initial Graphics Content** determines the type of toolbar that is shown. The toolbar will be different depending on the space dimension and whether the referenced source is a **Geometry**, **Mesh**, **Selection**, or **Plot Group** node. For example, the **Plot Group** node displays an additional **Show Legends** button.

The figure below shows the standard graphics toolbar as it appears when the **Geometry** node, for a 3D model, is used as a **Source for Initial Graphics Content**.

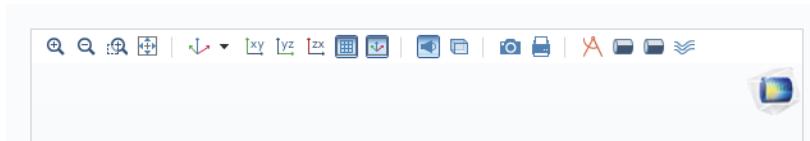


In the **Settings** window of a graphics object, in the **Toolbar** section, you can control whether or not to include the graphics toolbar, as well as its position (**Below**, **Above**, **Left**, **Right**).



Custom Graphics Toolbar Buttons

In the Toolbar section, you can also add custom buttons to the graphics toolbar. Use the buttons under the table to add or remove custom toolbar buttons (items). You can also move toolbar buttons up or down, add a **Separator**, and **Edit** a button. The figure below shows a standard graphics toolbar with four additional buttons to the right.



The figure below shows the corresponding table of graphics toolbar items.

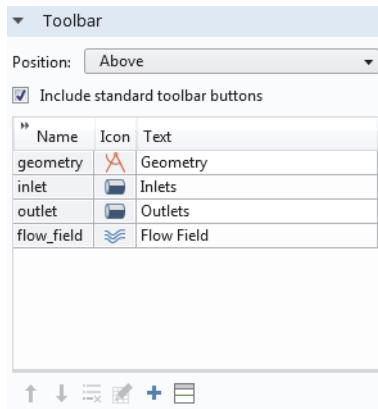
Toolbar

Position: Above

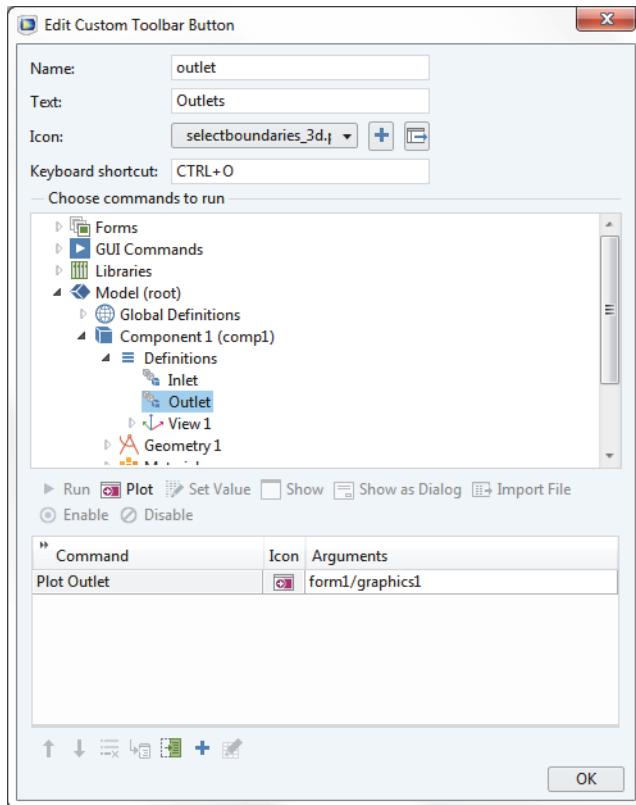
Include standard toolbar buttons

Name	Icon	Text
geometry	triangle icon	Geometry
inlet	square icon	Inlets
outlet	square icon	Outlets
flow_field	wavy lines icon	Flow Field

↑ ↓ ☰ + ☱



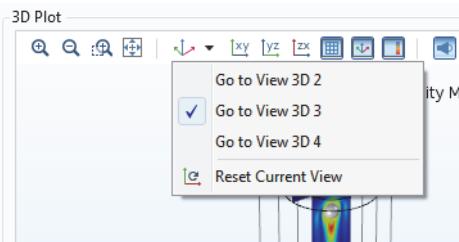
To edit the command sequence for a toolbar item, click the **Edit** button to open the **Edit Custom Toolbar Button** dialog box.



This dialog box has settings that are similar to those of a button. For details, see “Button” on page 48.

Views

In the graphics toolbar, the **Go to Default 3D View** button (for 3D graphics only) will display a menu with all applicable views. The currently active view is indicated with a check mark.

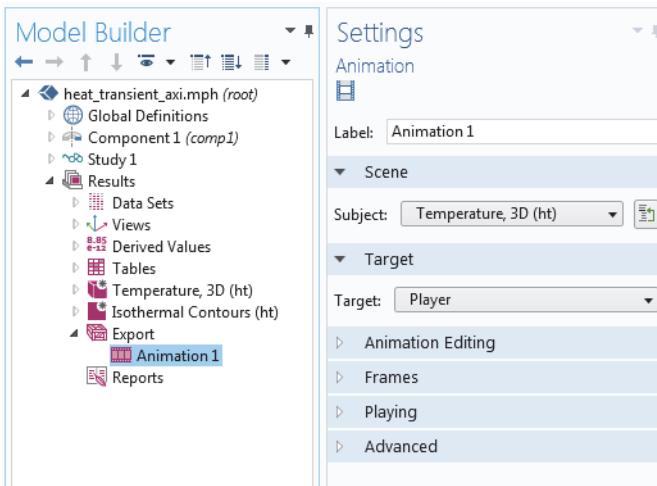


In addition to a list of all views, there is an option **Reset Current View** that will reset the currently active view to the state it had when the application was launched.

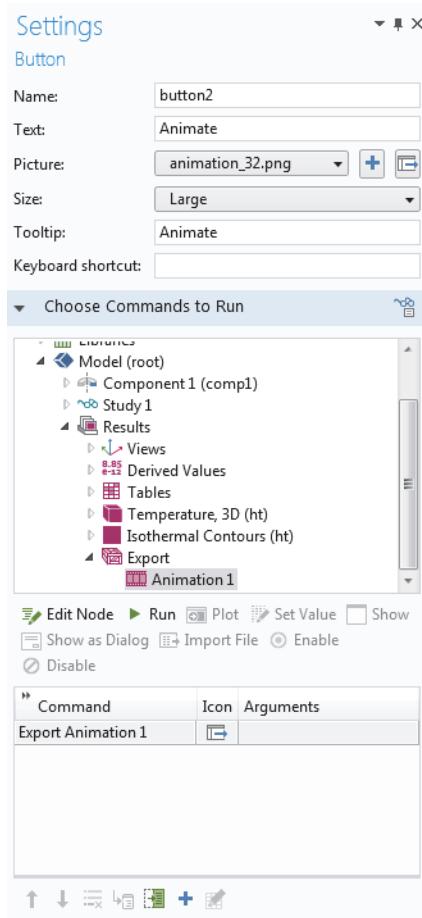
ANIMATIONS

You can display animations in an application by creating a link to the graphics object that is used in the Model Builder as the **Subject** (source) for an **Animation**. You create such a link by following these steps:

- In the Model Builder, add an **Animation** node under the **Export** node.
- Change the **Target** to **Player**. Make sure the **Subject** is set to a plot group that is used as a **Source for Initial Graphics Content** for the graphics object where you wish to run the animation.



- Create a button, menu, ribbon, or toolbar item with a **Run** command applied to the **Export Animation** node.



Setting the **Target** as **Player** ensures that the animation is run in the same graphics window as the **Subject** plot group, instead of being exported to a file.

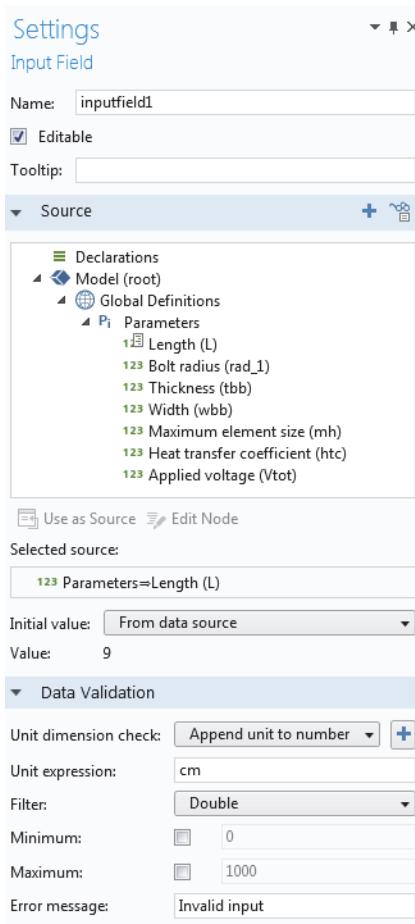
Input Field

An **Input Field** allows a user to change the value of a parameter or variable. In the New Form wizard, when a parameter or variable is selected, three form objects are created:

- A **Text Label** object for the parameter or variable description.
- An **Input Field** object for the value.
- A **Unit** object (if applicable) that carries the unit of measure.

To insert an additional input field, use the **Insert Object** menu in the ribbon and select **Input Field**. In the Form editor, you link an input field to a certain parameter or variable by selecting it from the tree in the **Source** section and click **Use as Source**.

In the **Source** section of the **Settings** window, you can also set an **Initial value**. The figure below shows the **Settings** window for an input field.



The default setting for the **Initial value** is **From data source**. This means that if the source is a parameter, then the initial value displayed in the input field is the same as the value of the parameter as specified in the **Parameters** node in the Model Builder. The other **Initial value** option is **Custom value**, which allows an initial value different from that of the source. If the **Editable** check box is cleared, then the **Initial value** will be displayed by the application and cannot be changed.

You can add a **Tooltip** with text that is shown when hovering the mouse pointer over the input field.

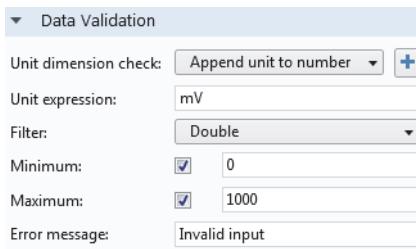
The header of the **Source** section contains two buttons for easy access to tools that are used to make additional properties and variables available as sources to the input field.



The **Create New Variable and Use It as Source** button can be used to add new variables under the **Declarations** node. For more information, see “Declarations” on page 115. The **Switch to Model Builder and Activate Model Data Access** button can be used to access low-level model properties as described in the next section. For more information on **Model Data Access**, see “Model Data Access in the Form Editor” on page 80.

DATA VALIDATION

The **Data Validation** section of the **Settings** window for an input field allows you to validate user inputs with respect to units and values.



When creating an input field in the New Form wizard, the setting **Append unit to number** is used when applicable. This setting assumes that a user enters a number into the input field, but it can also handle a number followed by a unit using the COMSOL square bracket [] unit syntax. If the **Unit expression** is `mm`, then `1[mm]` is allowed, as well as any length unit, for example, `0.1[cm]`. An incompatible unit type will display the **Error message**. A parameter that has the expression `1.23[mm]`, and that is used as a source, will get the appended unit `mm` and the initial value displayed in the edit field will be `1.23`.

The **Unit dimension check** list has the following options:

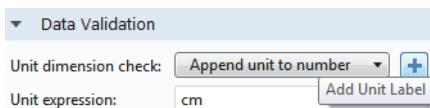
- **None**
- **Compatible with physical quantity**
- **Compatible with unit expression**
- **Append unit to number** (default)

A value or expression can be highlighted in orange to provide a warning when the user of an application enters an incompatible unit, which is any unit of measure that cannot be converted to the units specified in the **Data Validation** settings. Enable this feature by selecting **Compatible with physical quantity** or **Compatible with unit expression**. In addition, the user will see a tooltip explaining the unit mismatch, as shown in the figure below.

Length:	9[kg]	m
Width:	5[cm]	Deduced unit is [kg], expected is [m]
Applied voltage:	20[mV]	V

If there is a unit mismatch, and if no further error control is performed by the application, the numeric value of the entered expression will be converted to the default unit. In the above figure, 9[kg] will be converted to 9[m].

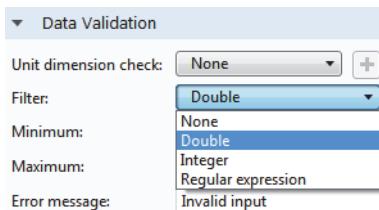
A button **Add Unit Label** is available to the right of the **Unit dimension check** list.



Clicking this button will add a unit label to the right of the input field if there is not already a unit label placed there.

The **None** option does not provide unit validation.

The options **Append unit to number** and **None** allow you to use a filter to validate the input.



The **Filter** list for the option **None** has the following options:

- **None**
- **Double**
- **Integer**
- **Regular expression**

The **Filter** list for the option **Append unit to number** only allows for the **Double** and **Integer** options.

The **Double** and **Integer** options filter the input based on **Minimum** and **Maximum** values. If the input is outside of these values, the **Error message** is displayed.

The **Regular expression** option allows you to use a regular expression for matching the input string. For more information on regular expressions, see the dynamic help. Click the help icon in the upper-right corner of a window and search for “regular expression”.

Error Messages

You can customize the text displayed by the **Error message**. During development and debugging of an application it can sometimes be hard to deduce where such errors originate from. Therefore, when using **Test Application** additional debugging information is displayed, as shown in the figure below.

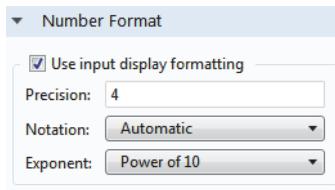


The debugging information typically consists of the type of form object, the path to the form object, and the reason for the failure, for example, $5 \leq x \leq 10$.

No extra information is added when launching an application by using Run Application or COMSOL Server.

NUMBER FORMAT

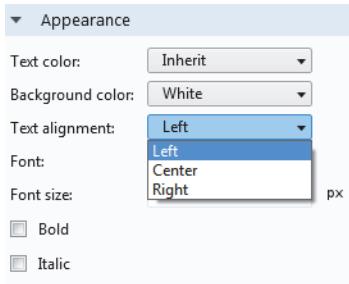
The **Number Format** section contains a check box **Use input display formatting** that, if selected, enables the same type of display formatting as a **Data Display** object.



For more information, see “Data Display” on page 77.

APPEARANCE

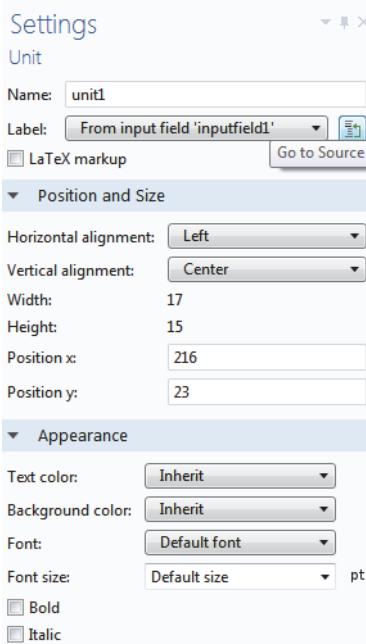
In addition to color and font settings, the **Appearance** section for an input field contains a **Text alignment** setting that allows the text to be **Left**, **Center**, or **Right** aligned.



Unit

In the **Settings** window for a **Unit** object, you can set the unit to a fixed string, or link it to an input field. Click the **Go to Source** button to the right of the unit **Label**

list to show, in the form, the input field object it is linked to. The figure below shows the **Settings** window for a unit object.

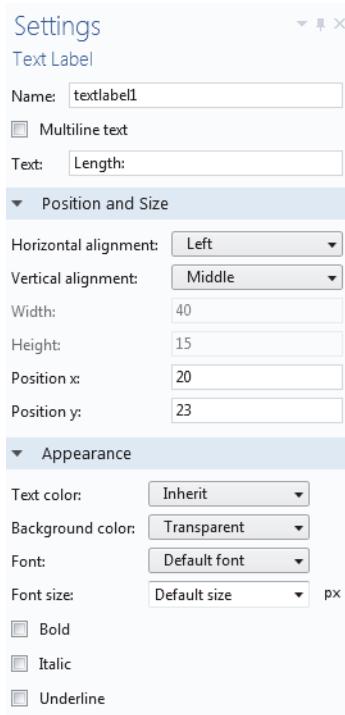


When adding an input field using the New Form wizard, a unit object is automatically added when applicable. By default, the unit is displayed using Unicode rendering. As an alternative, you can use LaTeX rendering by selecting the **LaTeX markup** check box. Then, the display of units will not depend upon the selected font.

Text Label

A **Text Label** object simply displays text in a form. When adding an input field using the New Form wizard, a **Text Label** object is automatically added for the description text of the associated parameter or variable. There is a check box

allowing for **Multiline text**. If selected, the **Wrap text** check box is enabled. The figure below shows the Settings window for a **Text Label** object.



To insert an additional **Text Label**, use the **Insert Object** menu in the ribbon and select **Text Label**.

Data Display

A **Data Display** object is used to display the numerical values of scalars and arrays. If there is an associated unit, it will be displayed as part of the **Data Display** object.

SOURCE

In the **Settings** window for a data display object, in the **Source** section, select a node in the model tree. Then click the **Use as Source** button below. Valid parameters, variables, and properties include:

- The output from a **Derived Values** node, such as a **Global Evaluation** or a **Volume Maximum**

- Variables declared under the **Declarations** > **Scalar**, **ID Array**, and **2D Array** nodes
- Properties made available by using the **Model Data Access** tool, see “Model Data Access in the Form Editor” on page 80
- One of the following **Information** node variables, which are under the root node and under each Study node:

- **Expected Computation Time**

This is a value that you enter in the **Expected** field in the **Settings** window of the root node.

- **Last Computation Time** (under the root node)

This is the last measured computation time for the last computed study.

- **Last Computation Time** (under a study node)

This is the last measured computation time for that study.

When you start an application for the first time, the last measured times are reset, displaying **Not available yet**.

USING THE NEW FORM WIZARD FOR GENERATING DATA DISPLAY OBJECTS

In the New Form wizard, in the **Inputs/outputs** tab, only the **Derived Values** nodes will generate **Data Display** objects. Variables under **Declarations** and constants made available with **Model Data Access** will instead generate **Input Field** objects.

When a **Derived Values** node is selected, two form objects are created based on the corresponding **Derived Values** node variable:

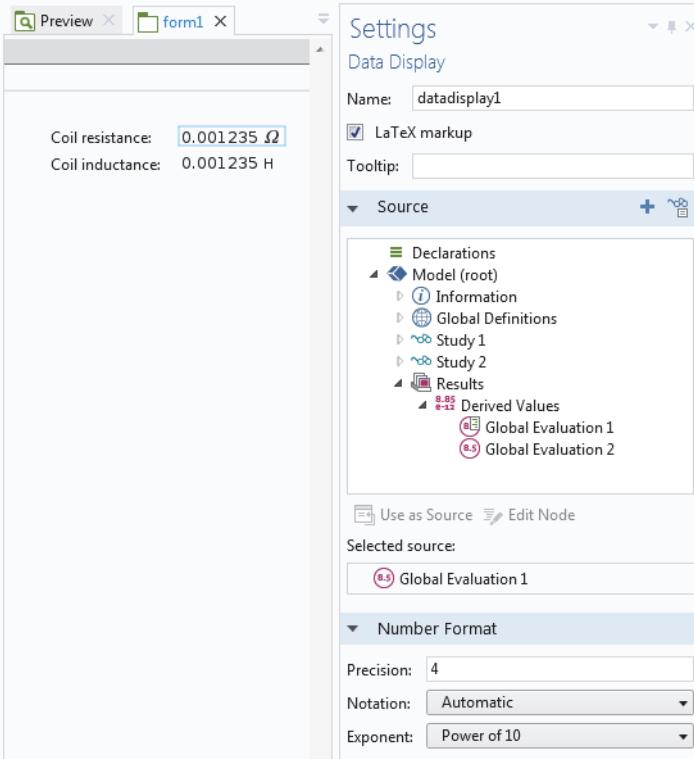
- a **Text Label** object for the **Description** of the variable
- a **Data Display** object for the value of the variable

The settings for these form objects can subsequently be edited. To insert additional data display objects, use the **Insert Object** menu in the ribbon and select **Data Display**.

NUMBER FORMAT

The **Number Format** section lets you set the **Precision**, **Notation**, and **Exponent**.

The figure below shows an example with data display objects for the variables `Coil resistance` and `Coil inductance`. A formatted unit label is automatically displayed as part of the object if applicable.



RENDERING METHOD

By default, the unit of a data display object is displayed using Unicode rendering. As an alternative, you can use LaTeX rendering by selecting the **LaTeX markup** check box. Then, the data display does not rely on the selected font.

Formatted display of arrays and matrices is only supported with LaTeX rendering. The figure below shows a 2D double array (see page 123) displayed using a **Data Display** object with **LaTeX markup** selected.

$$\begin{bmatrix} 0 & 0 & 0.9 & 0.8 & 0.7 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0.5 & 0.7 & 1.5 & 0.8 & 0.6 & 0.3 & 0.2 & 0.1 & 0.1 \\ 0 & 0 & 0.9 & 0.8 & 0.7 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

You can add a **Tooltip** with text that is shown when hovering over the data display object.

Model Data Access in the Form Editor

The **Settings** window of many types of form objects has a section that allows you to select a node in a tree structure that includes the model tree, or parts of the model tree. Examples include the **Source** section of an input field or the **Choose Commands to Run** section of a button. There are many properties in the model tree that are not made available by default, because a model typically contains hundreds or even thousands of properties, and the full list would be unwieldy. However, these “hidden” properties may be made available to your application by a technique called **Model Data Access**.

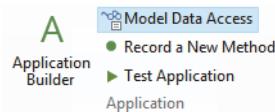
The remainder of this section gives an introduction to using **Model Data Access** with examples for input fields and buttons.

MODEL DATA ACCESS FOR INPUT FIELDS

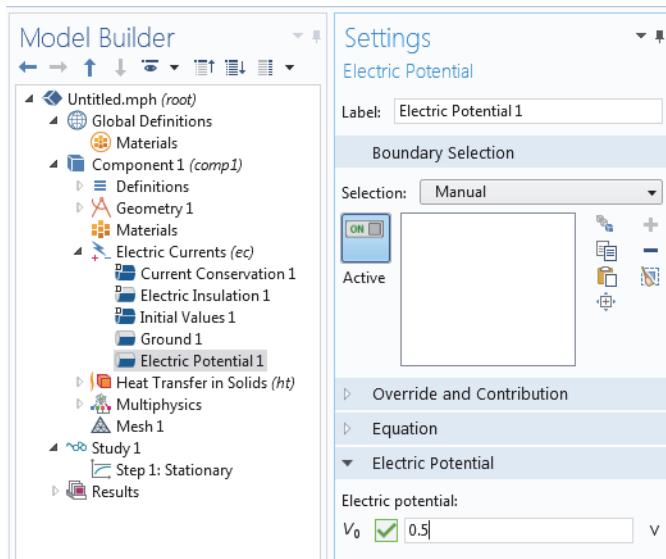
By default, you can link input fields to parameters and variables defined in the model tree under the **Parameters** or **Variables** nodes and to variables declared in the application tree under the **Declarations** node. To access additional model tree node properties, click the **Switch to Model Builder and Activate Model Data Access** button in the header of the **Source** section of the input field **Settings** window, as shown in the figure below.



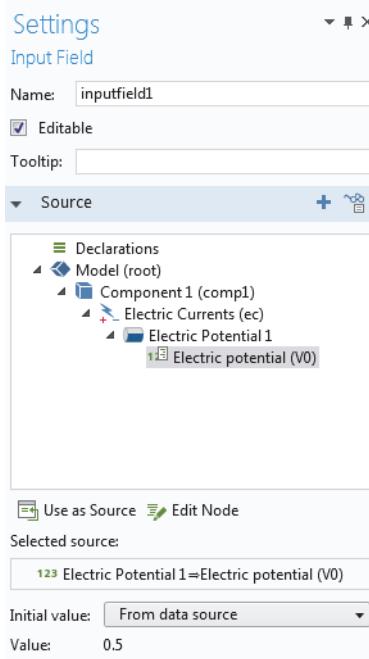
You can also access it from the **Application** group of the **Home** tab of the Model Builder.



Then, when you click on a model tree node, check boxes appear next to the individual settings. In the figure below, the check box for an **Electric potential** boundary condition is selected:



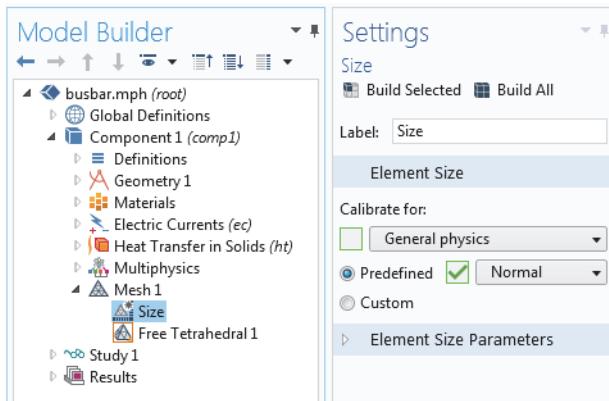
The figure below shows the **Settings** window for an input field. The list of possible sources for this field now contains the **Electric potential**.



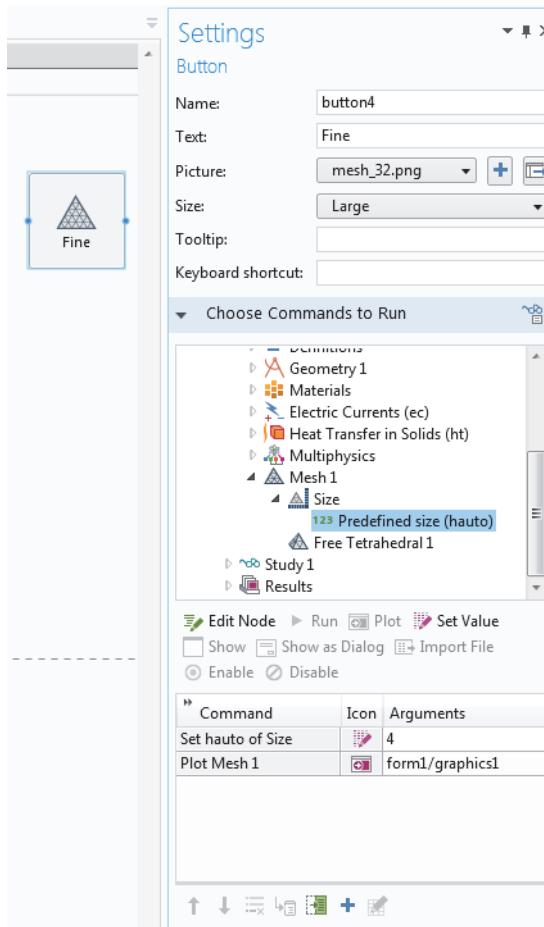
MODEL DATA ACCESS FOR BUTTONS

Model Data Access can be used for buttons to set the value of a parameter, variable, or a model property. For example, you can create buttons for predefined

mesh element sizes. In the figure below, the **Predefined** property for **Element Size** has been made available and then selected.



The figure below shows the **Settings** window for a button used to create a mesh with **Element Size > Predefined** set to **Fine**.



In this example, a **Set Value** command is used to set the value of the **Predefined mesh size (hauto)** property. The property **Predefined mesh size (hauto)** has to be set to an integer value according to the following table:

PREDEFINED MESH SIZE	VALUE
Extremely fine	1
Extra fine - Extra coarse	2 - 8
Extremely coarse	9

In general, for individual model tree properties you can quickly learn about their allowed values by recording code while changing their values and then inspect the automatically generated code. For more information, see “Recording Code” on page 136.

You can also use a combo box object to give direct access to all of the options from **Extremely fine** through **Extremely coarse**. For more information, see “Combo Box” on page 172.

SUMMARY OF MODEL DATA ACCESS

The table below summarizes the availability of **Model Data Access** for form objects and events, as well as menu, toolbar, and ribbon items.

FORM OBJECT, EVENT, OR ITEM	SECTION IN SETTINGS WINDOW
Input Field	Source
Button	Choose Commands to Run
Toggle Button	Source
Check Box	Source
Combo Box	Source
Data Display	Source
Graphics (Graphics Toolbar Item)	Choose Commands to Run
Form Collection	Active Pane Selector Tiled or Tabbed
Card Stack	Active Card Selector
Information Card Stack	Active Information Card Selector
Radio Button	Source
Text	Source
List Box	Source
Slider	Source
Toolbar (Toolbar Item)	Choose Commands to Run
Menu Item	Choose Commands to Run
Ribbon Item	Choose Commands to Run
Event (Global)	Choose Commands to Run Source for Data Change Event

A global event, menu, ribbon, or toolbar item provides a **Choose Commands to Run** section in its **Settings** window, to which the functionality described above in the section on buttons also applies. Global events and many form objects provide a

Source section in its **Settings** window, and the functionality described above in the section on input fields applies. For information on global events, menus, ribbons, and toolbar items, see “Graphics Toolbar” on page 65, “The Main Window” on page 102, “Events” on page 109, “Table” on page 229, and “Toolbar” on page 236.

Sketch and Grid Layout

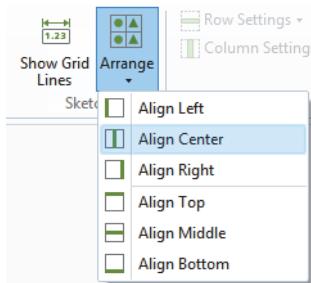
The Form editor provides two layout modes for positioning form objects: sketch layout mode and grid layout mode. The default is sketch layout mode, which lets you use fixed positions and sizes of objects in pixels. Use grid layout mode to position and size objects based on a background grid with cells. In grid layout mode, a form is divided into a number of intersecting rows and columns, with at most one form object at each intersection. This layout mode is recommended for designing a resizable user interface such as when designing an application for being run in a web browser on multiple platforms.

SKETCH LAYOUT

Switch between sketch and grid layout mode by clicking **Sketch** or **Grid** in the **Layout** group in the ribbon.



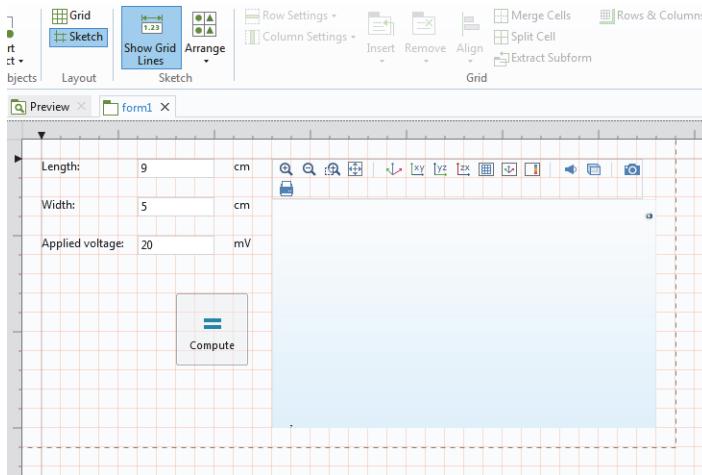
The **Sketch** group in the **Form** tab has two options: **Show Grid Lines** and **Arrange**. The **Arrange** menu allows you to align groups of form objects relative to each other.



Sketch Grid

The **Show Grid Lines** option displays a sketch grid that objects are snapped to. Note that the grid used in sketch layout mode is different from the grid used in grid layout mode. The default setting for sketch layout mode is to show no grid lines. Without grid lines visible, a form object being dragged is snapped relative to the position of the other form objects.

If the **Show Grid Lines** option is selected, the upper left corner of a form object being dragged is snapped to the grid line intersection points.

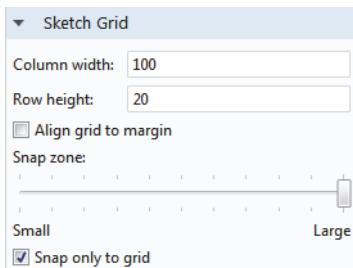


In the **Settings** window of the form, you can change the settings for the sketch grid:

- **Column width**
- **Row height**
- **Align grid to margin**
- **Snap zone**
 - A slider allows you to change the snap zone size from **Small** to **Large**.

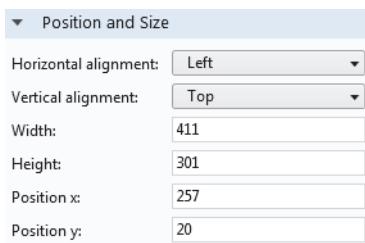
- **Snap only to grid**

- Clear this check box to snap both to the grid and the position of other form objects.



Position and Size

The sketch layout mode is pixel based, and the positioning of form objects is indicated as the coordinates of the top-left corner of the form object measured from the top-left corner of the screen. The x-coordinate increases as the object moves to the right, and the y-coordinate increases as the object moves from the top of the screen to the bottom. You can set the absolute position of a form object in the **Position and Size** section of its **Settings** window.



Form objects are allotted as much space as required, or as specified by their **Width** and **Height** values. Form objects are allowed to overlap.

GRID LAYOUT

Switch to grid layout mode by clicking **Grid** in the **Layout** group in the ribbon.

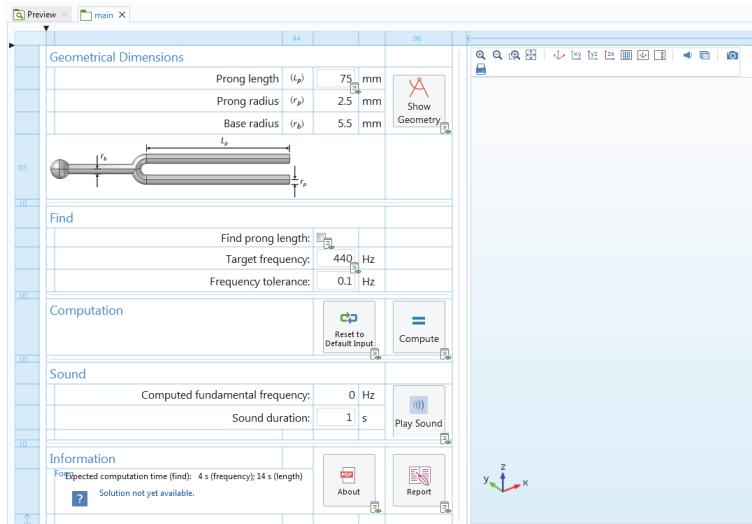


The buttons and menus in the ribbon **Grid** group give you easy access to commands for:

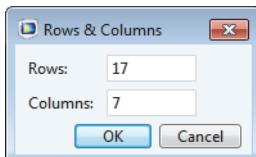
- Changing the row and column growth rules between **Fit**, **Grow**, and **Fixed**, which determine the layout when the user interface is resized (**Row Settings** and **Column Settings**).
- Inserting or removing rows and columns (**Insert** and **Remove**).
- Aligning form objects within grid cells (**Align**).
- Merging and splitting cells (**Merge Cells** and **Split Cells**).
- Extracting a rectangular array of cells as a subform and inserting it into a new form (**Extract Subform**).
- Defining the number of rows and columns (**Rows & Columns**).

The Form Settings Window and the Grid

After switching to grid layout mode, the form window shows blue grid lines.



To define the number of rows and columns, click the **Rows & Columns** button in the ribbon.



The section **Grid Layout for Contained Form Objects**, in the **Settings** window, shows column widths and row heights.

Grid Layout for Contained Form Objects		
	Column Width	Size
1	Fit	N/A
2	Fit	N/A
3	Fit	N/A
4	Fit	N/A
5	Fixed	96
6	Fit	N/A
7	Grow	N/A

	Row	Height	Size
1	Fit	N/A	
2	Fit	N/A	

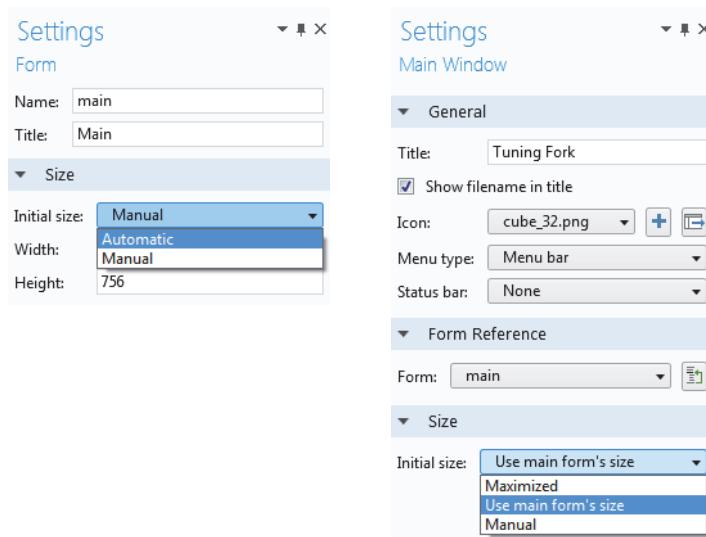
To interactively select a form, as displayed in the Form editor, click the top-left corner of the form.



A blue frame is now shown. To interactively change the overall size of a form, you can drag its right and bottom border. The form does not need to be selected for this to work.

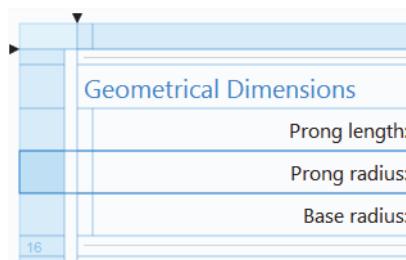
Note that if you switch from sketch to grid layout mode, all rows and columns will have the setting **Fit** and the handles for the frame will not be displayed. If any of the rows and columns have the **Height** or **Width** setting set to **Grow**, then the frame will display handles for resizing in the vertical or horizontal direction, respectively.

The size of the interactively resized frame will affect the initial size of the form only if the **Initial size** setting is set to **Automatic**. The size of the frame will also affect the initial size of the **Main Window** if its **Initial size** setting is set to **Use main form's size**.

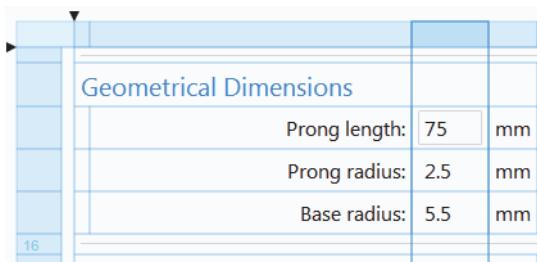


Rows and Columns

Click the leftmost cell of a row to select it. The leftmost cells are only used for selecting rows; form objects cannot be inserted there. When a row is selected, the **Row Settings** menu as well as the **Insert** and **Remove** commands are enabled in the ribbon tab. The figure below shows the fourth row highlighted.



Similarly, to select a column, click the cell at the top. This cell cannot contain any form objects. The figure below shows the third column highlighted. In this case, the **Column Settings** menu is enabled in the ribbon tab.

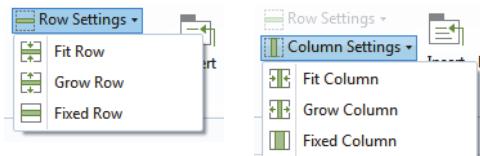


A screenshot of a Microsoft Word table. The table has three columns and four rows. The first row contains the text "Geometrical Dimensions". The second row contains "Prong length: 75 mm". The third row contains "Prong radius: 2.5 mm". The fourth row contains "Base radius: 5.5 mm". The third column is highlighted with a blue selection bar. The ribbon tab "Column Settings" is active.

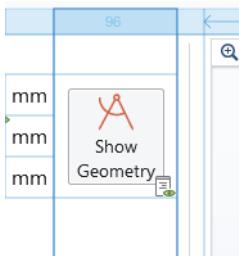
	Geometrical Dimensions	
	Prong length:	75 mm
	Prong radius:	2.5 mm
	Base radius:	5.5 mm

The **Row Settings** and the **Column Settings** have the same three options:

- **Fit** sets the row height or column width to the smallest possible value given the size of the form objects in that row or column
- **Grow** sets the row height or column width to grow proportionally to the overall size of the form
- **Fixed** sets a fixed value for the number of pixels for the row height or column width

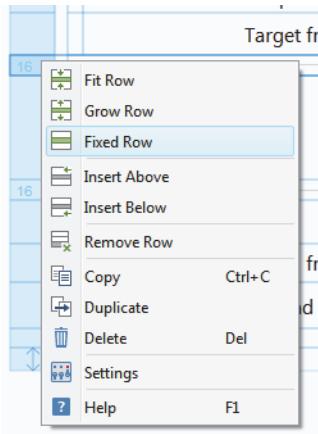


You can interactively change the row height and column width by dragging the grid lines.



In this case, the number of pixels will be displayed and the **Row Settings** or **Column Settings** growth policy will be changed automatically to **Fixed**.

As an alternative to changing the **Row Settings** or **Column Settings** from the ribbon, you can right-click in a row or column and select from a menu.



The menu shown when right-clicking a row or column also gives you options for inserting, removing, copying, pasting, and duplicating rows or columns.

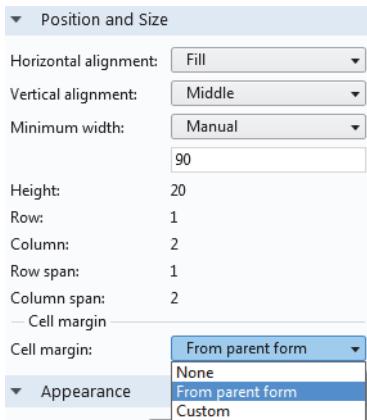
Cells

Click an individual cell to select it. A selected cell is shown with deeper blue grid lines.



You can select **Merge Cells** and **Split Cells** to adjust the cell size and layout to your form objects.

When in grid layout mode, you can specify the margins that are added between the form object and the borders of its containing cell.



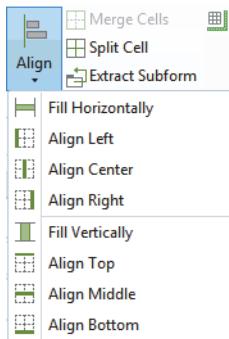
In the **Settings** window of a form object, the **Position and Size** section has the following options for **Cell margin**:

- **None**
 - No cell margins
- **From parent form** (default)
 - The margins specified in the **Settings** window of the form, see “Inherit Columns and Cell Margins” on page 99
- **Custom**
 - Custom margins applied only to this form object

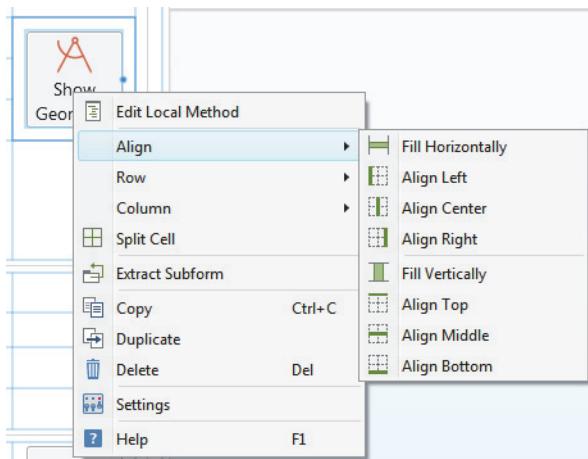
If the **Horizontal alignment** or **Vertical alignment** is set to **Fill** and the growth policy of the column or row allows the form object to be resized, then you can specify a minimum width or height, respectively. The minimum size can be set to **manual** or **automatic**. The **manual** option lets you specify a pixel value for the minimum size. The **automatic** option allows for a minimum size of zero pixels, unless the form object contents require a higher value. The minimum size setting is used at run time to ensure that scrollbars are shown before the form object shrinks below its minimum size.

Aligning Form Objects

The **Align** menu gives you options for aligning form objects within a cell. You can also let a form object dynamically fill a cell horizontally or vertically.



As an alternative, you can right-click a form object and select from a context menu.



Drag and Drop Form Objects

You can move form objects by drag and drop. Click a form object to select it, and then drag it to another cell that is not already occupied with another form object.

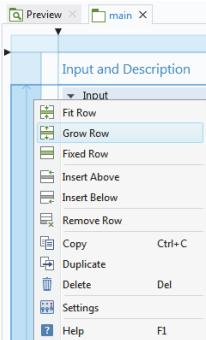
Thermal conductivity:	0.559	W/(m·K)
Heat of reaction:	-84666	J/mol
<input type="button" value="Compute"/>		

If you drop the object in an already occupied cell, then the objects switch places.

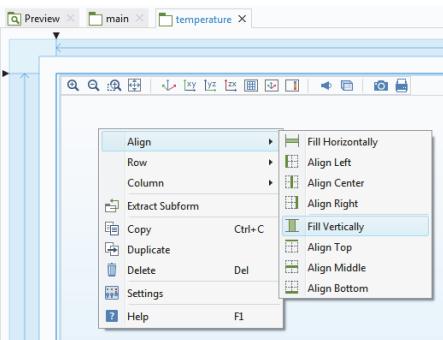
Automatic Resizing of Graphics Objects

In order to make the graphics object of an application resizable, follow these steps:

- Change the layout mode of the form containing the graphics object from sketch to grid layout mode.
- Change the **Height** setting for any row covering the graphics object to **Grow**. To change this, click the leftmost column of the row you would like to access. Then change the **Height** setting in the **Settings** window of the form. Alternatively, right-click and select **Grow Row**.



- Change the **Width** for any column covering the graphics object to **Grow**. To change this, right-click the uppermost row of the column you would like to access and select **Grow Column**.
- Select the graphics object and change both the **Horizontal alignment** and **Vertical alignment** to **Fill**. You can do this from the **Settings** window or by right-clicking the graphics object and selecting **Align>Fill Horizontally** and **Align>Fill Vertically**.



Following the steps above, you may find it easier to make graphics objects resizable by performing grid layout mode operations such as adding empty rows

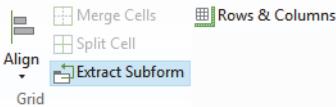
and columns as well as merging cells. If you are already in grid layout mode, then a graphics object will default to **Fill** in both directions.

Extracting Subforms

You can select a rectangular array of cells in a form and move it to a new form. First select the cells by using Ctrl+click or Shift+click.

The screenshot shows a Microsoft Word ribbon with the 'Grid Layout' tab selected. Below the ribbon is a form containing two sections: 'Computation' and 'Sound'. The 'Sound' section contains fields for 'Computed fundamental frequency' (0 Hz) and 'Sound duration' (1 s). To the right of the 'Sound' section is a 'Play Sound' button. A selection box surrounds the entire 'Sound' section, indicating it has been selected for extraction. The 'Grid Layout' tab has several options: 'Merge Cells', 'Rows & Columns', 'Split Cell', 'Align' (with dropdown arrows), and 'Extract Subform'. The 'Extract Subform' button is highlighted with a blue border.

Then click the Extract Subform button in the ribbon.



This operation creates a new form with the selected cells and replaces the original cells with a subform object of type **Form**. In the **Settings** window of the subform, the **Form** reference points to the new form containing the original cells.

The screenshot shows the Form Editor interface with two forms:

- main**: The parent form containing the following sections:
 - Geometrical Dimensions**: Includes fields for Prong length (l_p) set to 75 mm, Prong radius (r_p) set to 2.5 mm, and Base radius (r_b) set to 5.5 mm. A "Show Geometry" button is present.
 - Find**: Includes fields for Find prong length, Target frequency (440 Hz), and Frequency tolerance (0.1 Hz).
 - Computation**: Contains "Reset to Default Input" and "Compute" buttons.
 - Sound**: Includes fields for Computed fundamental frequency (0 Hz) and Sound duration (1 s). It also includes a "Play Sound" button.
- form1**: The subform being created, which only contains the **Sound** section from the main form.

The **Settings** window on the right shows the configuration for the subform:

- Name**: subform2
- Form**: form1
- Position and Size** settings:
 - Horizontal alignment: Fill
 - Vertical alignment: Fill
 - Minimum width: Automatic
 - Minimum height: Automatic
 - Row: 14
 - Column: 1
 - Row span: 4
 - Column span: 6
 - Cell margin: None

Inherit Columns and Cell Margins

By using subforms, you can organize your user interface, for example, by grouping sets of input forms. The figure below shows part of a running application with two subforms for **Beam dimensions** and **Reinforcement bars**.

Beam dimensions	
Height of the beam:	200[mm] m
Width of the beam:	300[mm] m
Length of the beam:	4[m] m
Reinforcement bars	
Diameter of the bar:	10[mm] m
Number of bar layers:	2
Layer spacing:	20[mm] m
Distance from surface of first rebars layer:	10[mm] m
Width spacing:	60[mm] m
Minimal lateral distance from rebars to beam surface:	10[mm] m
Number of bars across the width:	5

For more information on adding subforms to a form, see the previous section and “Form” on page 199.

When aligning subforms vertically, as in the example above, you may want to ensure that all columns are of equal width. For this purpose you can use the **Inherit columns** option in the **Settings** window of a subform. The figure below shows part of the **Settings** window for the **Beam dimensions** subform (left) with **Name** `geometry_beam` and for the **Reinforcement bars** subform (right) with **Name**

`geometry_rebars`. The `geometry_rebars` subform has its **Inherit columns** set to `geometry_beam`.

The image shows two side-by-side screenshots of the COMSOL Form Editor's Settings window.

Left Window (geometry_beam):

- Form:**
 - Name: `geometry_beam`
 - Title: Beam dimensions
- Grid Layout for Contained Form Objects:**
 - Column Width:**

Column	Size
1	Fixed 280
2	Fixed 100
3	Fixed 45
4	Grow N/A
 - Row Height:**

Row	Size
1	Fit N/A
2	Fit N/A
3	Fit N/A
 - Inherit columns:** None
 - Cell margins:**
 - Horizontal: 5
 - Vertical: 3

Right Window (geometry_rebars):

- Form:**
 - Name: `geometry_rebars`
 - Title: Reinforcement bars
- Grid Layout for Contained Form Objects:**
 - Row Height:**

Row	Size
1	Fit N/A
2	Fit N/A
3	Fit N/A
4	Fit N/A
5	Fit N/A
6	Fit N/A
7	Fit N/A
 - Inherit columns:** `geometry_beam`
 - Cell margins:**
 - Horizontal: 5
 - Vertical: 3

In the subsection **Cell margins**, you can specify the **Horizontal** and **Vertical** margins that are added between form objects and the borders of their containing cells. These settings will affect all form objects, contained in the form, with their individual **Cell margins** set to **From parent form**, see “Cells” on page 93.

Copying Between Applications

You can copy and paste forms and form objects between multiple COMSOL Multiphysics sessions running simultaneously. You can also copy and paste within one session from the current application to a newly loaded application.

In grid layout mode, a cell, multiple cells, entire rows, and entire columns may be copied between sessions.

When you copy and paste forms and form objects between applications, the copied objects may contain references to other forms and form objects. Such references may or may not be meaningful in the application to which they are copied. For more information on the set of rules applied when pasting objects, see “Appendix B—Copying Between Applications” on page 239.

When copying and pasting between applications, a message dialog box will appear if a potential compatibility issue is detected. In this case, you can choose to cancel the paste operation.

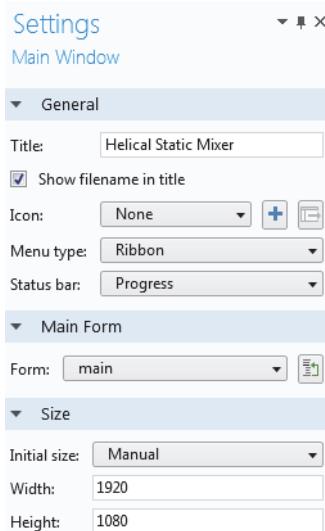
The Main Window

In the application tree, the **Main Window** node represents the main window of an application and is also the top-level node for the user interface. It contains the window layout, the main menu specification, and an optional ribbon specification.

GENERAL SETTINGS

The **Settings** window contains a **General** section with settings for:

- **Title**
- **Show filename in title**
- **Icon**
- **Menu type**
- **Status bar**



The **Title** is the text at the top of the Main Window in an application, with the **Icon** shown to the far left of this text. By default, the **Title** is the same as the title of the model used to create the application. Keep the check box **Show filename in title** selected if you wish to display the file name of the application to the left of the **Title**.

In the **Icon** list, select an image from the library or add an image (*.png) from the local file system to the library and use it as an icon. If you add a new image, it will be added to the image library and thereby embedded into the application.

You can also export an icon by clicking the **Export** button to the right of the button **Add Image to Library and Use Here**.

The **Main Window** node of the application tree has one child node, named **Menu Bar**. Using the **Menu** type setting, you can change this child node from **Menu Bar** to **Ribbon**.

The **Status bar** list controls what is shown in the status bar: Select **Progress** to display a progress bar when applicable (the default), or **None**. Note that you can also create custom progress bars by using methods.

MAIN FORM SETTINGS

The **Main Form** section contains a reference to the form that the main window displays. This setting is important when using a form collection because it determines which form is displayed as the main window when the application is opened for the first time.

INITIAL SIZE SETTINGS

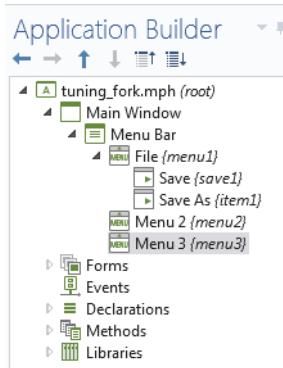
The **Initial size** setting determines the size of the main window when the application is first started. There are three options:

- **Maximized** results in the window being maximized when the application is run.
- **Use main form's size** uses the size of the main form, see “The Individual Form Settings Windows” on page 38. The main form is defined by the **Main Form** section. This option further adds the size required by the main window itself, including: the window frame and title bar, main menu, main toolbar, and ribbon. This size is computed automatically and depends on whether the menu type is **Menu bar** or **Ribbon**.
- **Manual** lets you enter the pixel size for the width and height. In this case, nothing is added to the width and height. When using this option, you need to ensure there is enough room for the window title, ribbon and menu bar.

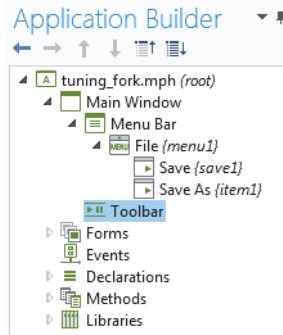
For more information on the option **Use main form's size**, see “The Form Settings Window and the Grid” on page 89.

Menu Bar and Toolbar

The **Menu Bar** node can have **Menu** child nodes that represent menus at the top level of the Main Window.

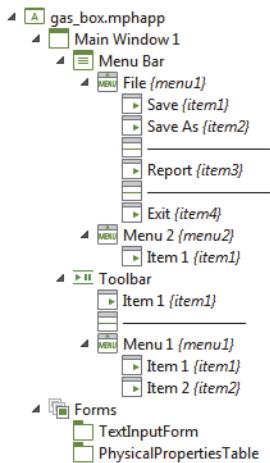


For the **Menu Bar** option, you can also add a **Toolbar**. The **Toolbar** node and the **Menu** nodes have the same type of child nodes.

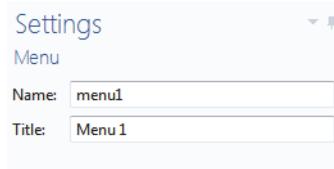


MENU, ITEM, AND SEPARATOR

The child nodes of the **Menu** and **Toolbar** nodes can be of type **Menu**, **Item**, or **Separator**, as shown in the figure below:



A **Menu** node has settings for **Name** and **Title**.

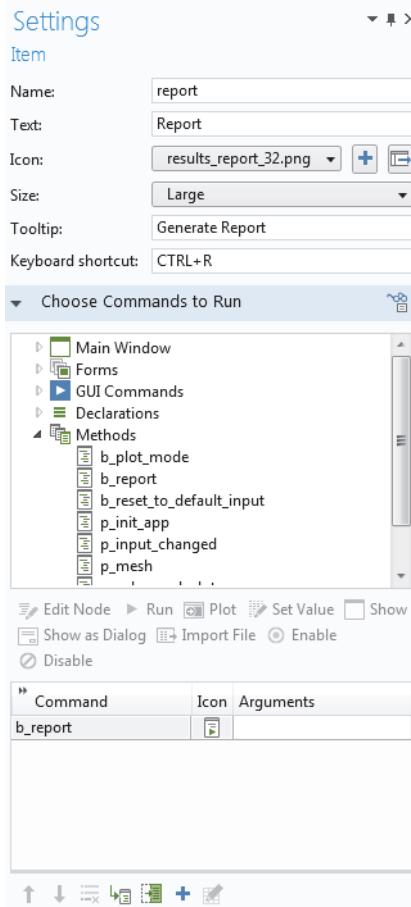


A **Menu** node can have child **Menu** nodes that represent submenus.

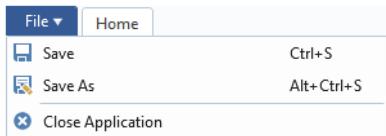
A **Separator** displays a horizontal line between groups of menus and items, and has no settings.

The **Settings** window for an **Item** node is similar to that of a button and contains a sequence of commands. Just like a button, an item can have associated text, an icon, and a keyboard shortcut. For more information, see “Button” on page 48.

The figure below shows the **Settings** window for an **Item** associated with a method for creating a report.

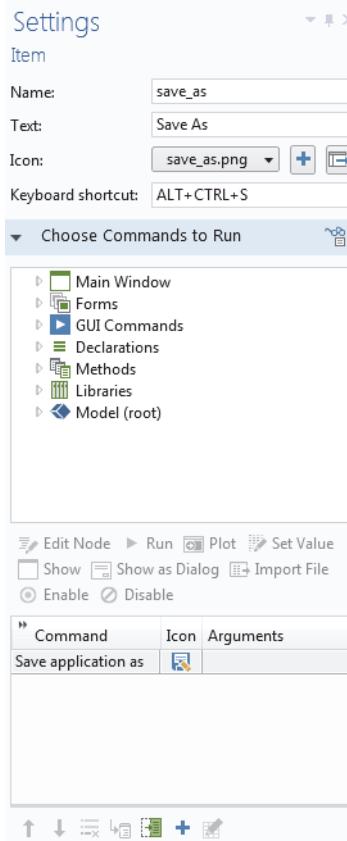


The figure below shows an example of an application with a **File** menu.



When running an application in the COMSOL Desktop environment, a **Close Application** menu item is always present.

The **Settings** window for the **Save As** item is shown in the figure below.



You can enable and disable ribbon, menu, and main toolbar items from methods. For more information, see “Appendix E—Built-in Method Library” on page 264.

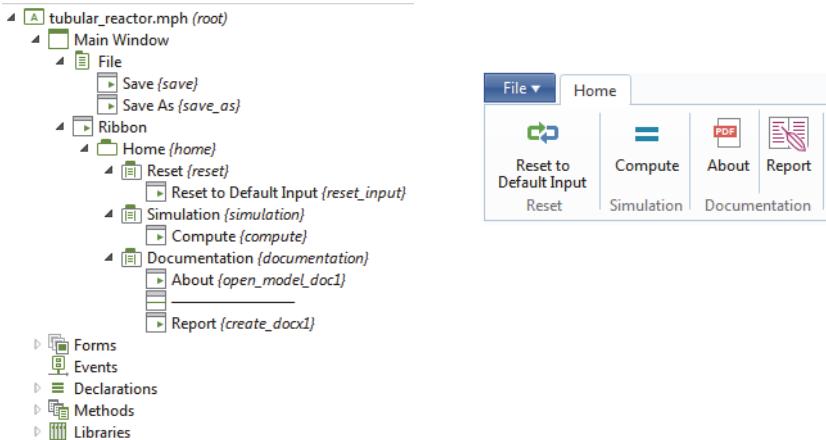
Ribbon

You can opt to add a **Ribbon** to the Main Window instead of a **Menu Bar**. The **Ribbon** node contains the specifications of a ribbon with toolbars placed on one or several tabs. For the **Ribbon** option, a **File** menu is made available directly under the Main Window node.

RIBBON TAB AND RIBBON SECTION

Child nodes to the **Ribbon** node are of the type **Ribbon Tab**. Child nodes to a **Ribbon Tab** are of the type **Ribbon Section**. Child nodes to a **Ribbon Section** can be of the type **Item**, **Menu**, or **Separator**.

Item and **Menu** provide the same functionality as described previously for the **MenuBar** and **Toolbar**. A **Separator** added as a child to a **Ribbon Section** is a vertical line that separates groups of **Items** and **Menus** in the running application. A **Separator** is displayed as a horizontal line in the application tree. The figure below shows an example.



Events

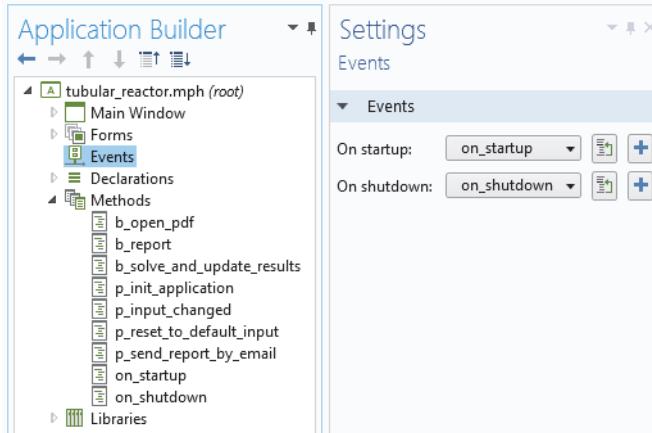
An event is any activity (e.g. clicking a button, typing a keyboard shortcut, loading a form, or changing the value of a variable) that signals a need for the application to carry out one or more actions. Each action can be a sequence of commands of the type described earlier, or may also include the execution of methods. The methods themselves may be local methods associated with particular form objects or global methods that can be initiated from anywhere in the application. The global methods are listed in the **Methods** node of the application tree. The local methods are defined in the **Settings** windows of the forms or form objects with which they are associated. When a form object has an associated method, it may be opened for editing by performing a Ctrl+Alt+click on the object. If the Ctrl+Alt+click is performed on a form object that has no method, then a new local method, associated with the object, will be created and opened for editing.

The events that initiate these actions may also be global or local. The global events are listed in the **Events** node of the application tree and include all events that are triggered by changes to the various data entities, such as global parameters or string variables. Global events can also be associated with the startup and shutdown of the application. The local events, like local objects, are defined in the **Settings** windows of the forms or form objects with which they are associated.

Event nodes trigger whenever the source data changes, regardless if it is changed from a method, form object, or in any other way. Events associated with form objects only trigger when the user changes the value in the form object.

Events at Startup and Shutdown

Global or local methods can be associated with the events at startup (**On startup**) and shutdown (**On shutdown**) of an application. To access these events, click the **Events** node in the application tree.



A shutdown event is triggered when:

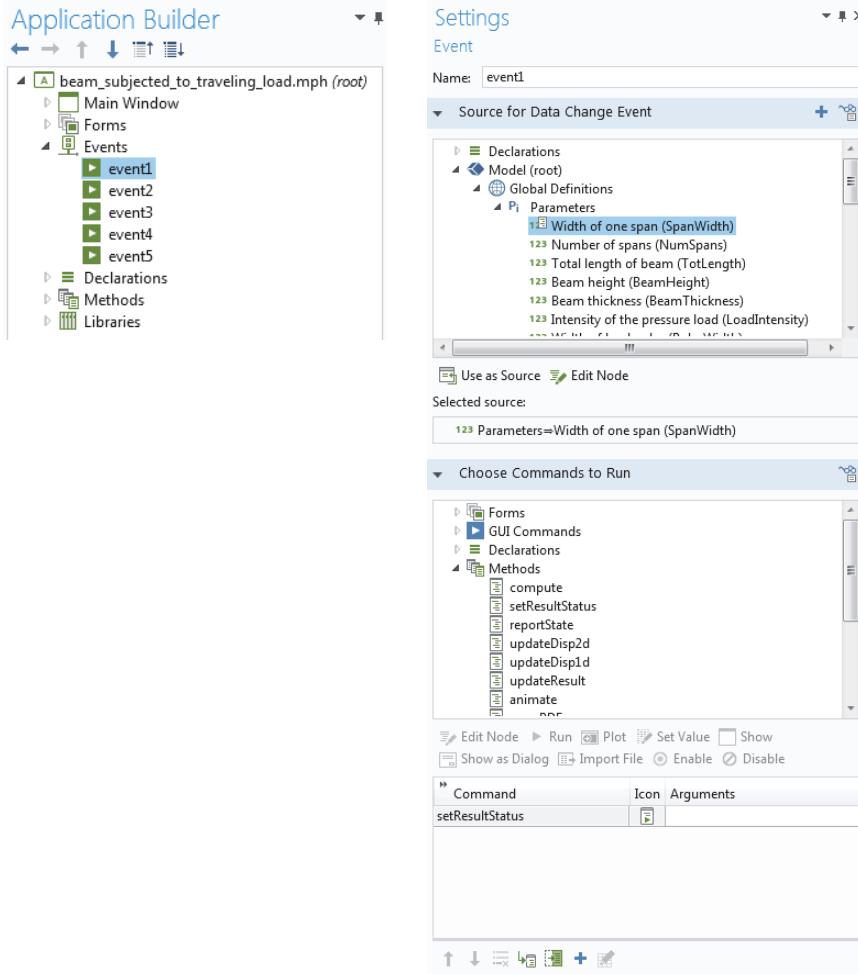
- The user of an application closes the application window by clicking the **Close Application** icon in the upper-right corner of the application window
- The **Exit Application** command is issued by a form object
- A method is run using the command `exit()`

A method run at a shutdown event can, for example, automatically save critical data or prompt the user to save data.

Global Events

Right-click the **Events** node and choose **Event** to add an event to an application. An event listens for a change in a running application. If a change occurs, it runs

a sequence of commands. In the figure below, when the value of the string variable `SpanWidth` is changed, the method `setResultsStatus` is run.



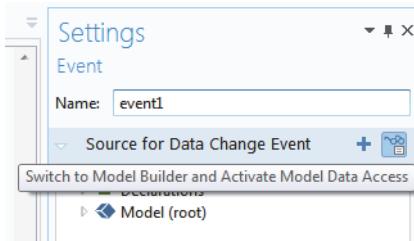
Note that since this type of event has global scope and is not associated with a particular form, the full path: `/form1/graphics1` needs to be used when referencing graphics objects.

The following two sections describe the options available in the **Settings** window of an event.

SOURCE FOR DATA CHANGE EVENT

This section presents a filtered view of the tree from the Application Builder window. The nodes represent some sort of data or have children that do.

You can extend the list of available data nodes by clicking on the **Switch to Model Builder and Activate Model Data Access** button in the header of the section **Source For Data Change Event**.



For more information, see “Model Data Access in the Method Editor” on page 134.

CHOOSE COMMANDS TO RUN

In the **Settings** window for an **Event**, the section **Choose Commands to Run** is similar to that of a button and allows you to define a sequence of commands. For more information, see “Button” on page 48.

Form and Form Object Events

Form and form object events are similar to global events, but are defined for forms or individual form objects. These events have no associated list of commands, but refer directly to one global or local method.

EVENTS TRIGGERED BY DATA CHANGE

For certain types of form objects, you can specify a method to run when data is changed. This setting is available in the **Events** section of the form object, as shown in the figure below.



The drop-down list **On data change** contains **None** (the default), any available methods under the **Methods** node of the application tree, and a local method (optional).

The form objects supporting this type of event are:

- **Input Field**
- **Check Box**
- **Combo Box**
- **File Import**
- **Radio Button**
- **Text**
- **List Box**
- **Table**
- **Slider**

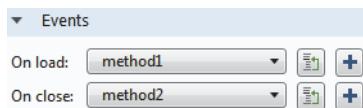
Buttons have associated events triggered by a click. Menu, ribbon, and toolbar items have associated events triggered by selecting. The corresponding action is a command sequence defined in the **Settings** window of a button object or item. For more information on command sequences, see “Button” on page 48.

Selecting Multiple Form Objects

You can specify an **On data change** event for multiple form objects simultaneously by using Ctrl+click and then selecting the method to run. In this way you can, for example, quickly specify that a data change event initiated by any of the selected form objects should run a method that informs the user that plots and outputs are invalid.

EVENTS TRIGGERED BY LOADING OR CLOSING A FORM

Forms can run methods when they are loaded (**On load**) or closed (**On close**).



This type of event is available in the **Settings** window of a form and is typically used when a form is shown as a dialog box, or to activate forms used as panes in a form collection.

Using Local Methods

Events can call local methods that are not displayed in the application tree. For more information on local methods, see “Local Methods” on page 146.

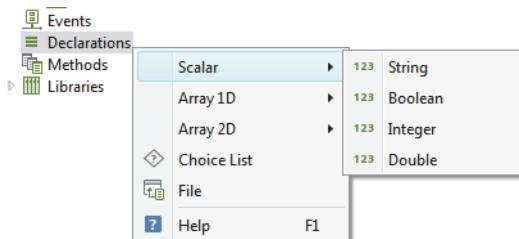
Declarations

The **Declarations** node in the application tree is used to declare global variables, which are used in addition to the global parameters and variables already defined in the model. Variables defined under the **Declarations** node are used in form objects and methods. In form objects, they store values to be used by other form objects or methods. Variables that are not passed between form objects and methods, but that are internal to methods, do not need to be declared in the **Declarations** node. In methods, variables defined under the **Declarations** node have global scope and can be used directly with their name. For information on how to access global parameters defined in the model tree, see “Accessing a Global Parameter” on page 156.

There are five different types of declarations:

- **Scalar**
- **Array 1D**
- **Array 2D**
- **Choice List**
- **File**

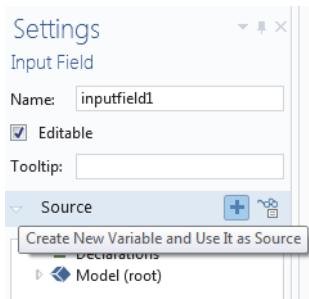
Right-click the **Declarations** node to access the declaration types or use the ribbon.



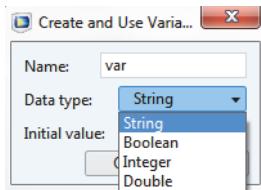
In addition, the first three types of declarations can be of the following data types:

- **String**
- **Boolean**
- **Integer**
- **Double**

In addition to right-clicking the **Declarations** node, you can click the **Create New Variable and Use it as Source** button in the **Source** section of many types of form objects.



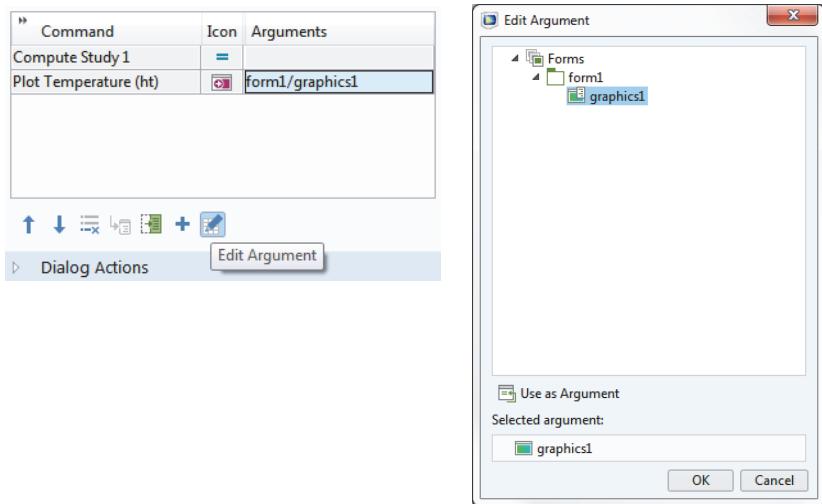
This will open a dialog box that lets you quickly declare scalar variables.



USING DECLARATIONS AS INPUT ARGUMENTS TO COMMANDS

Certain commands used in the commands sequence of, for example, a button can take an input argument. For more information, see “Button” on page 48.

The figure below shows a command sequence that includes a command Plot Temperature with an input argument `form1/graphics`.



You can use declarations as input arguments to commands.

To use a scalar variable, 1D array, or 2D array as input arguments, you use the corresponding variable name. To access a single element of an array, or a row or column of a 2D array, you use indices. For example, to access the first component in a 1D array `my_variable` you use `my_variable(1)`. A 2D array element can be retrieved as a scalar by using two indices, e.g. `my_matrix(2,3)`. The indices can themselves be other declared variables, e.g. `my_variable(n)`.

For commands requiring a graphics object as an input argument, only string type declarations are allowed with appropriate indices if necessary. If there is a graphics object named `graphics1` and also a string declaration named `graphics1`, then the contents of the string declaration will be used. An exception is if single quotes are used, such as '`graphics1`', in which case the graphics object `graphics1` is used. This rule is also applied to other combinations of commands and input arguments.

THE NAME OF A VARIABLE

The **Name** of a variable is a text string without spaces. The string can contain letters, numbers and underscore. The reserved names `root` and `parent` are not allowed and Java® programming language keywords cannot be used.

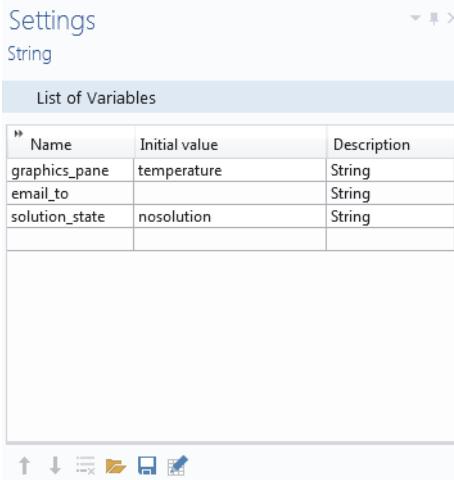
Scalar Declarations

Scalar declarations are used to define variables to be used as strings, Booleans, integers, or doubles.

STRING

A scalar string variable is similar to a global parameter or variable in a model, but there is a difference. A parameter or variable in a model has the restriction that its value has to be a valid model expression, while a scalar string variable has no such restrictions. You can use a string variable to represent a double, integer, or Boolean by using conversion functions in a method. For more information see “Conversion Methods” on page 265. You can also use a string variable as a source in many form objects, such as input fields, combo boxes, card stacks, and list boxes.

The figure below shows the **Settings** window for the string variables `graphics_pane`, `email_to`, and `solution_state`.



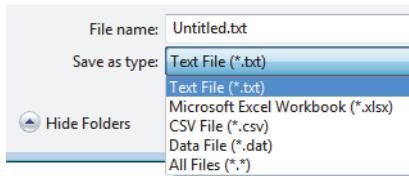
String declarations, as well as other declarations, can be loaded and saved from or to a file by using the **Load from File** and **Save to File** buttons below the **List of Variables** table.

The **Load from File** and **Save to File** buttons are used to load and save from/to the following file formats:

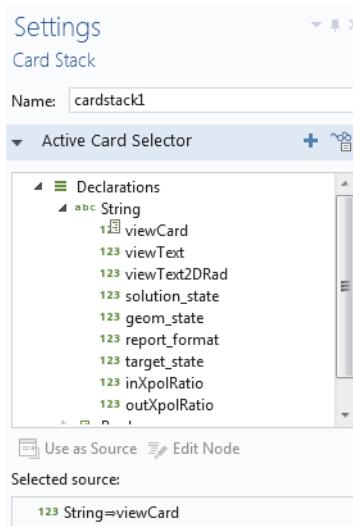
- Text File (.txt)

- Microsoft® Excel® Workbook (.xlsx)
 - Requires LiveLink™ for Excel®
- CSV File (.csv)
- Data File (.dat)

The drop-down list where these file formats can be selected from is shown in the figure below.



To illustrate the use of declared strings, the figure below shows the **Settings** window of a card stack object where the string variable `viewCard` is used as the source (**Active Card Selector**).



For more information on using card stacks, see “Card Stack” on page 203.

BOOLEAN

You can use a Boolean variable as a source in check boxes, other form objects, and methods. A Boolean variable can have two states: `true` or `false`. The default value is `false`. The figure below shows the declaration of two Boolean variables.

The screenshot shows a software interface for managing variables. At the top, there's a blue header bar with the word 'Settings'. Below it, a sub-header 'Boolean' is visible. The main area is titled 'List of Variables'. Inside, there's a table with three columns: 'Name', 'Initial value', and 'Description'. Two rows are present: one for 'validInput' with 'true' as the initial value and 'Boolean' as the description, and another for 'geomInitialization' with 'false' as the initial value and 'Boolean' as the description. At the bottom of the dialog are several small icons for navigating through the list of variables.

Name	Initial value	Description
validInput	true	Boolean
geomInitialization	false	Boolean

Example Code

In the example code below, the boolean variable `bvar` has its value controlled by a check box. If `bvar` is `true`, then plot group 4 (`pg4`) is plotted in `graphics1`. Otherwise, plot group 1 (`pg1`) is plotted.

```
if (bvar) {  
    useGraphics(model.result("pg4"), "graphics1");  
} else {  
    useGraphics(model.result("pg1"), "graphics1");  
}
```

INTEGER AND DOUBLE

Integer and double variables are similar to strings, with the additional requirement that the value is an integer or double, respectively.

The image shows two separate 'Settings' windows side-by-side. Both windows have a title bar with a close button (X) and a toolbar below it with icons for up/down arrows, a list icon, a folder icon, a copy icon, and a pencil icon.

Left Window (Integer Type):

- Title: Settings
- Section: Integer
- Table:

Name	Initial value	Description
n_of_digits	3	Number of sign
n_steps	0	Number of nor

Right Window (Double Type):

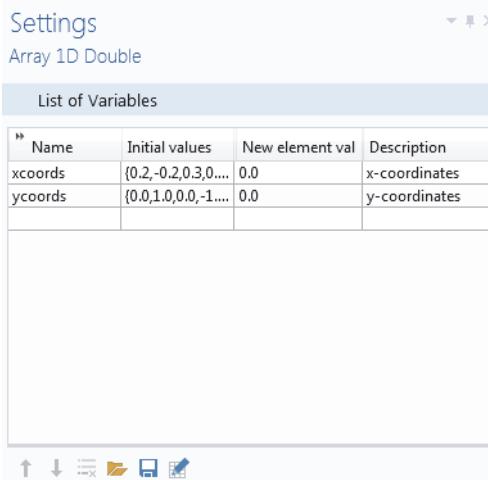
- Title: Settings
- Section: Double
- Table:

Name	Initial value	Description
element_size_low	0.5	Double
element_size_medium	0.38	Double
element_size_high	0.25	Double

Array 1D Declarations

The **Array 1D** node declares one or more named arrays of strings, Booleans, integers, or doubles that you can access from form objects and methods. The number of elements in a 1D array is not restricted in any way, and you can, for example, use a 1D array to store a column in a table with a variable number of rows. The **Settings** window contains a single table, where you specify one variable

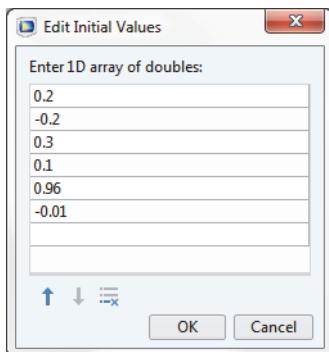
array per row. In the figure below, two double arrays are declared, `xcoords` and `ycoords`.



The values in the column **New element value** are assigned to new elements of the array when a row is added to a table form object. Arrays for strings, Booleans, and integers are similar in function to arrays of doubles.

INITIAL VALUES

The **Initial values** can be a 1D array of arbitrary length. To edit the initial values, click the **Edit Initial Values** button below the **List of Variables**. This opens a dialog box where the value of each component can be entered. See the figure below for an example of a 1D array of doubles.



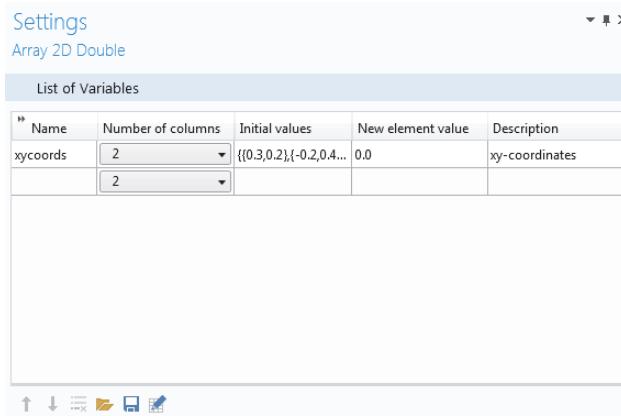
ARRAY SYNTAX

An array definition must start and end with curly braces ({ and }) and each element must be separated with a comma. When you need special characters inside an array element (spaces and commas, for example), surround the element with single quotes (''). The table below shows a few examples of 1D arrays:

ARRAY SYNTAX	RESULTING ARRAY
{1, 2, 3}	A 3-element array with the elements 1, 2, and 3
{}	An empty array
{'one, two', 'three by four'}	A 2-element array with elements containing special characters
{{1, 2, 3}, {'one, two', 'three by four'}}}	A 2-element array containing one 3-element array and one 2-element array

Array 2D Declarations

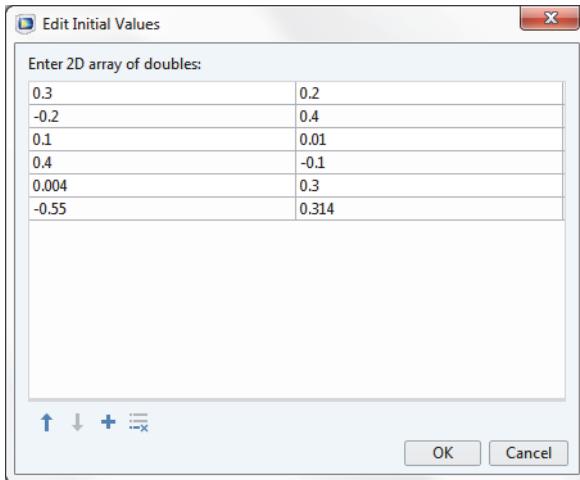
The **Array 2D** node declares one or more 2D arrays that you can access using form objects and methods. In the figure below, the 2D double array `xycoords` is declared.



INITIAL VALUES

The default (or initial) value can be a 2D array of arbitrary size. To edit the initial values, click the **Edit Initial Values** button below the **List of Variables**. This opens a

dialog box where the value of each component can be entered. See the figure below for an example of a 2D array of doubles.



ARRAY SYNTAX

The table below shows a few examples of 2D arrays:

ARRAY SYNTAX	RESULTING ARRAY
<code>{}{}</code>	An empty 3D array
<code>{{"5", "6"}, {"7", "8"}}</code>	A 2-by-2 matrix of strings
<code>{{1, 2, 3}, {4, 5, 6}}</code>	A 2-by-3 matrix of doubles

For 2D arrays, rows correspond to the first index so that `{{1,2,3},{4,5,6}}` is equivalent to the matrix:

1 2 3

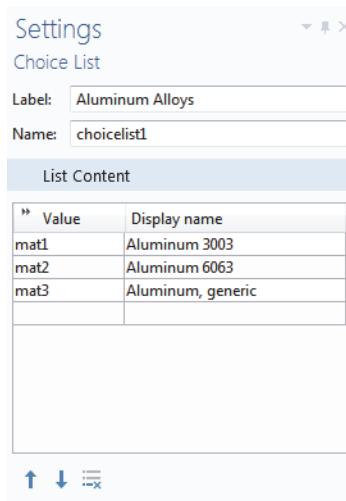
4 5 6

Assuming that the above 2-by-3 matrix is stored in the 2D array variable `arr`, then the element `arr[1][0]` equals 4.

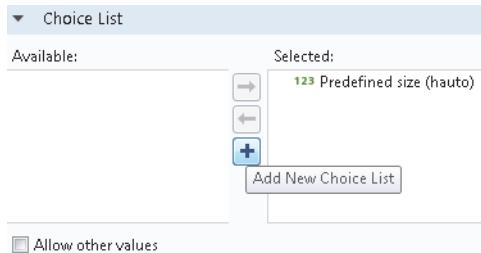
Choice List Declarations

The **Choice List** node contains lists that can be used by combo boxes, radio buttons, or list boxes. The **Settings** window for a choice list contains a **Label**, a **Name**, and a table with a **Value** column and a **Display name** column. Enter the

property value in the first column and the corresponding text to display to the user (for example, in a combo box list) in the second column. The **Value** is always interpreted as a string. In the example below, `mat1` will become the string “`mat1`” when returned from the combo box.



As an alternative to creating a choice list by right-clicking the **Declarations** node, you can click the **Add New Choice List** button in the Settings window for form objects that use such a list, as shown in the figure below.

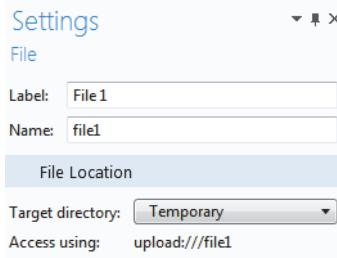


ACTIVATION CONDITION

You can right-click the **Choice List** node to add an **Activation Condition** subnode. Use an activation condition to switch between two or more choice lists contingent on the value of a variable. For an example of using choice lists with activation conditions, see “Using a Combo Box to Change Material” on page 177.

File Declarations

File declarations are primarily used for file import in method code when using the built-in method `importFile`. For more information on the method `importFile` and other methods for file handling, see “File Methods” on page 279. However, an entry under the **File** declaration node can also be used by a **File Import** object. The figure below shows the **Settings** window of a file declaration.



The file chosen by the user can be referenced in a form object or method using the syntax `upload:///file1`, `upload:///file2`, etc. The file name handle (`file1`, `file2`, etc.) can then be used to reference an actual file name picked by the user at run time.

For more information on file declarations and file handling, see “Appendix C—File Handling and File Scheme Syntax” on page 241.

The Method Editor

Use the Method editor to write code for actions not included among the standard run commands of the model tree nodes in the Model Builder. The methods may, for example, execute loops, process inputs and outputs, and send messages and alerts to the user of the application.

The Java® programming language is used to write COMSOL methods, which means that all Java® syntax and Java® libraries can be used. In addition to the Java® libraries, the Application Builder has its own built-in library for building applications and modifying the model object. The model object is the data structure that stores the state of the underlying COMSOL Multiphysics model that is embedded in the application. More information on these functions can be found in “Appendix E—Built-in Method Library” on page 264.

Methods can be global or local. Global methods are displayed in the application tree and are accessible from all methods and form objects. A local method is associated with a form object or event and can be opened from the corresponding Settings window. For more information about local methods, see “Local Methods” on page 146.



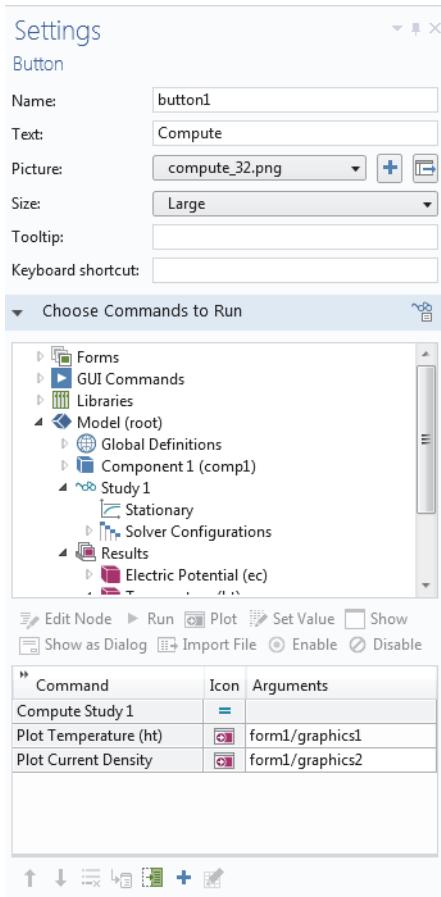
- A number of tools and resources are available to help you create code for methods. These are covered in the following sections, and will make you more productive by allowing you to copy-paste or auto-generate blocks of code.

Converting a Command Sequence to a Method

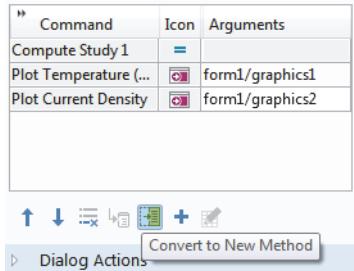
In the Form editor, click the **Convert to New Method** button displayed in the **Settings** window below an existing command sequence. The command sequence is automatically replaced by an equivalent method.

Consider a case where you have created a compute button and you want to be alerted by a sound when the computation has finished. Let’s see how this could be done using the Method editor.

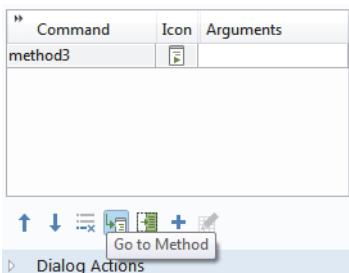
You'll also learn how to do this without using the Method editor later in this section. The figure below shows the **Settings** window of the compute button.



Click the **Convert to New Method** button below the command sequence.



The command sequence in this example is replaced by a method, `method3`. Click the **Go to Method** button. The Method editor opens with the tab for `method3` active.

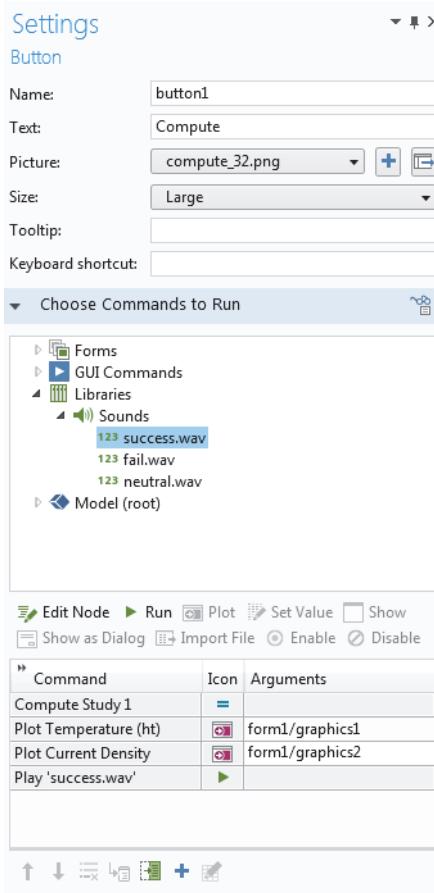


In the Method editor, add a call to the built-in method `playSound` to play the sound file `success.wav`, available in the COMSOL sound library, by using the syntax shown in the figure below..

```
1 model.study("std1").run();
2 useGraphics(model.result("pg2"), "form1/graphics1");
3 useGraphics(model.result("pg4"), "form1/graphics2");
4 playSound("success.wav");
```

The newly added line is indicated by the green bar shown at left.

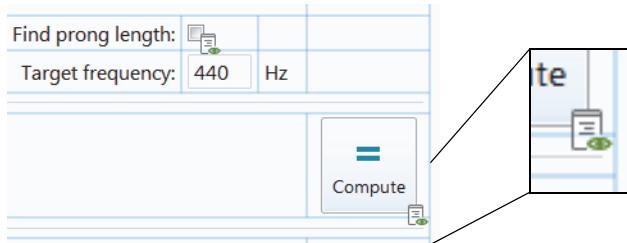
Note that in the example above, you do not have to use the Method editor. In the command sequence, select the file `success.wav` under **Libraries>Sounds** and click the **Run** command button under the tree, as shown in the figure below.



However, there are many built-in methods that do not have corresponding command sequence nodes. For more information, see “Appendix E—Built-in Method Library” on page 264.

FORM OBJECT WITH ASSOCIATED METHODS

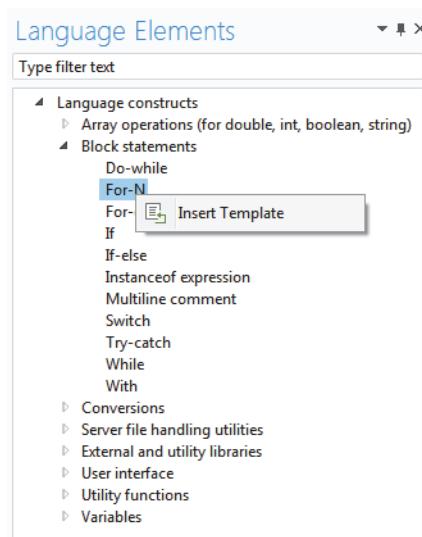
Form objects that have associated methods are indicated with a special icon, as shown in the figure below. In this example, both the check box called **Find prong length** and the **Compute** button have associated methods.



Performing Ctrl+Alt+click on the form object opens the method in the Method editor. If there is no method associated with the form object, a new local method associated with the form object will be created and opened in the Method editor.

Language Elements Window

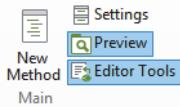
The **Language Elements** window in the Method editor shows a list of some language constructs. Double-click or right-click one of the items in the list to insert template code into the selected method:



See also “Language Element Examples” on page 153.

Editor Tools in the Method Editor

To display the **Editor Tools** window, click the corresponding button in the **Main** group in the **Method** tab.

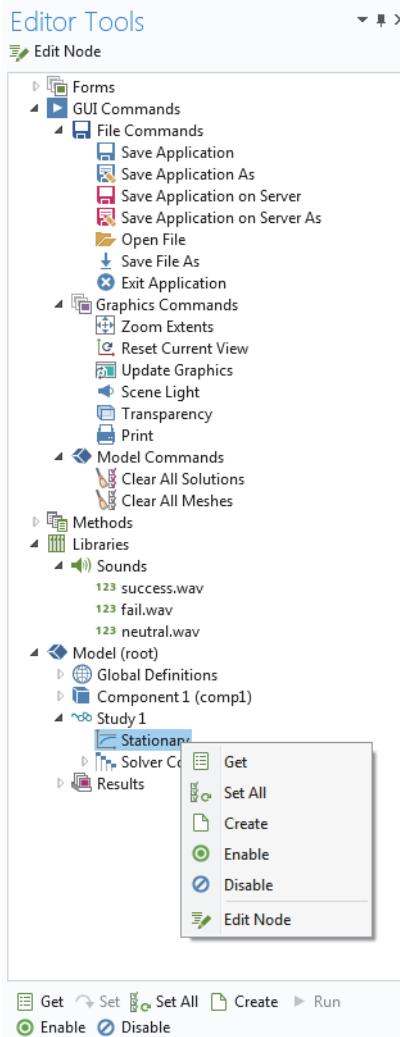


When using the **Editor Tools** window in the Method editor, you can right-click a node in the editor tree to generate code associated with that node. Depending on the node, up to eight different options are available:

- **Get**
- **Set**
- **Set All**
- **Create**
- **Run**
- **Enable**
- **Disable**
- **Edit Node**

Selecting one of the first seven options will add the corresponding code to the currently selected method. The **Edit Node** option brings you to the **Settings** window for the model tree node.

The figure below shows an example of a node with six options.



When a node is selected, the toolbar below the editor tree shows the available options for generating code.

The **Editor Tools** window is also an important tool when working with the Form editor. For more information, see “Editor Tools in the Form Editor” on page 46.

KEYBOARD SHORTCUTS

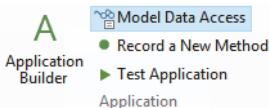
Consider a method with a line of code that refers to a model object in the following way:

```
model.result("pg3").feature("surf1").create("hght1", "Height");
```

If you position the mouse pointer in “surf1” and press F11 on the keyboard. You can also right-click and select **Go to Node** or click **Go to Node** in the ribbon. The corresponding **Surface** plot node is highlighted in the **Editor Tools** window. Click **Edit Node** to open its **Settings** window. For more information on keyboard shortcuts, see “Appendix D—Keyboard Shortcuts” on page 262.

Model Data Access in the Method Editor

To access individual properties of a model tree node, click the **Model Data Access** button in the **Application** section of the Model Builder ribbon tab:

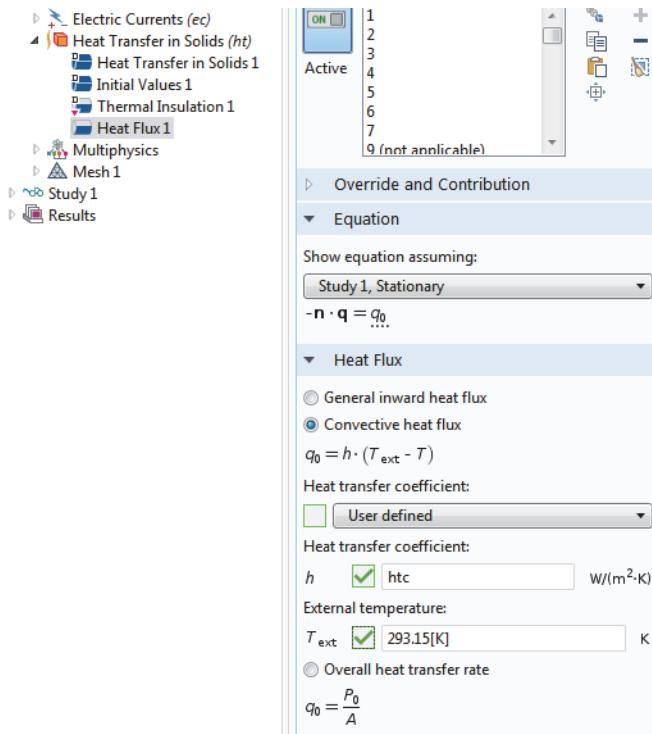


Alternatively, for certain form objects, you can click the **Model Data Access** button in the header of the **Source** section of the **Settings** window. See also “Model Data Access in the Form Editor” on page 80.

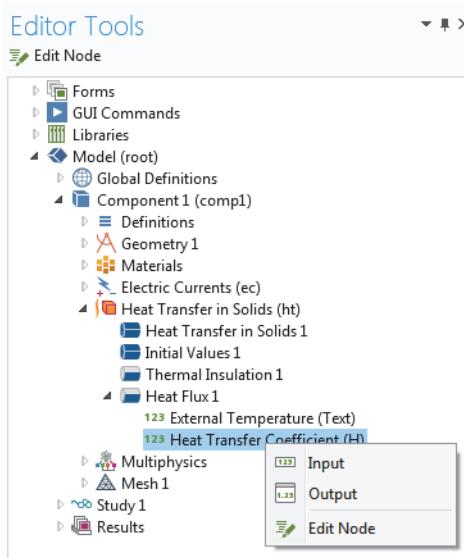
Model Data Access needs to be enabled this way because a model typically contains hundreds or even thousands of properties that could be accessed, and the list would be too long to be practical.

When you click a model tree node, such as the **Heat Flux** node in the figure below, check boxes appear next to the individual properties. This example is based on the busbar tutorial model described in *Introduction to COMSOL Multiphysics*.

In the figure below, the check boxes for **Heat transfer coefficient** and **External temperature** are selected:



If you switch to the **Editor Tools** window, you will see additional nodes appear under the **Heat Flux** node. Right-click and use **Get** or **Set** to generate code in an active method window, as shown in the figure below:



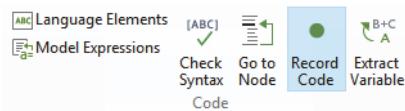
In the example above, **Get** and **Set** for the **Heat transfer coefficient** and the **External temperature** properties will generate the following code:

```
model.physics("ht").feature("hf1").getString("h");
model.physics("ht").feature("hf1").getString("Text");

model.physics("ht").feature("hf1").set("h", "htc");
model.physics("ht").feature("hf1").set("Text", "293.15[K]");
```

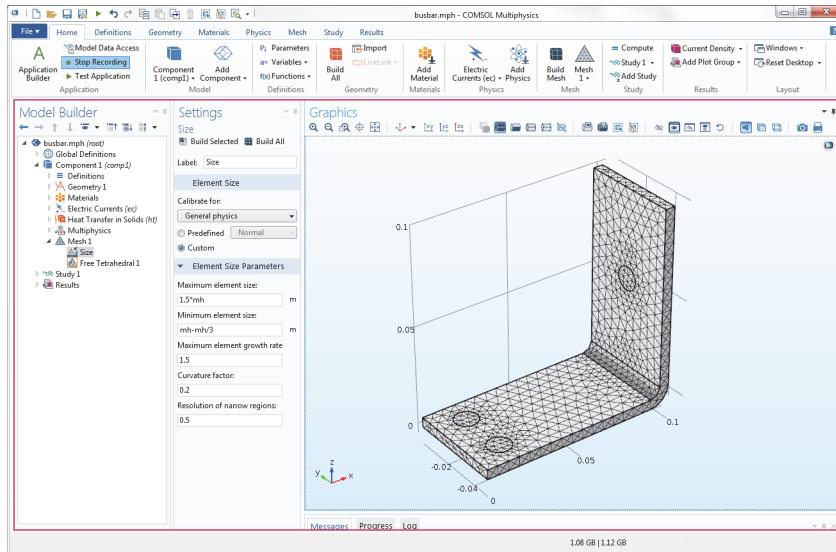
Recording Code

Click the **Record Code** button in the **Code** section of the Method editor ribbon to record a sequence of operations that you perform using the model tree, as shown in the figure below.

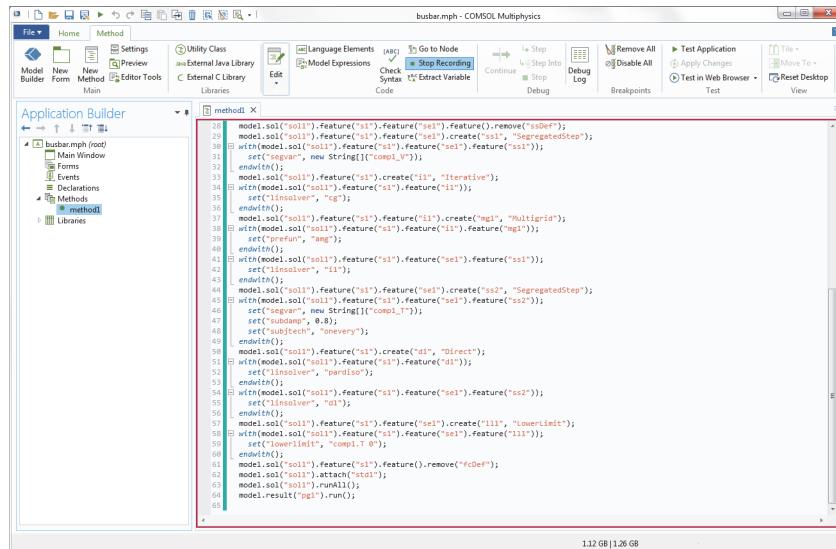


The code is recorded in the active method window, if any. If no method window is open, a new method will be created.

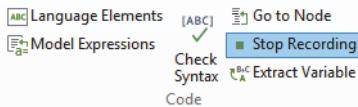
While recording code, the focus is shifted to the Model Builder window, which is surrounded by a red frame during recording:



If you switch back to the Method editor, the method window also has a red frame:



To stop recording code, click the **Stop Recording** button.



You can also record code directly from the Model Builder window by clicking **Record a New Method**. In this case, a new method is always created.

The previous section on **Model Data Access** explained how to set the values of the **Heat transfer coefficient** and the **External temperature** properties of the busbar tutorial model. To generate similar code using **Record Code**, follow these steps:

- Create a simple application based on the busbar model (MPH file).
- In the Model Builder window, click **Record a New Method**, or with the Method editor open, click **Record Code**.
- Change the value of the **Heat transfer coefficient** to 5.
- Change the value of the **External temperature** to 300[K].
- Click **Stop Recording**.
- If it is not already open, open the method with the recorded code.

The resulting code is listed below:

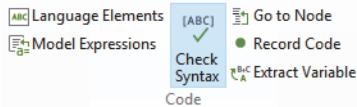
```
with(model.physcis("ht").feature("hf1"));
  set("h", "5");
  set("Text", "300[K]");
endwith();
```

In this case, the automatic recording contains a `with()` statement in order to make the code more compact. For more information on the use of `with()`, see “The With Statement” on page 155.

Use **Record Code** or **Record a New Method** to quickly learn how to interact with the model object. The auto-generated code shows you the names of properties, parameters, and variables. Use strings and string-number conversions to assign new parameter values in model properties. By using **Model Data Access** while recording you can, for example, extract a parameter value using `get`, process its value in a method, and set it back into the model object using `set`. For more information on **Model Data Access**, see “Model Data Access in the Method Editor” on page 134.

Checking Syntax

Click **Check Syntax** in the ribbon to see messages in the **Errors and Warnings** window related to syntax errors or unused variables.



In addition to messages in the **Errors and Warnings** window, syntax errors are indicated with a wavy red underline, as shown in the figure below.

The image shows two windows. The top window is a code editor titled 'method1' containing the following code:

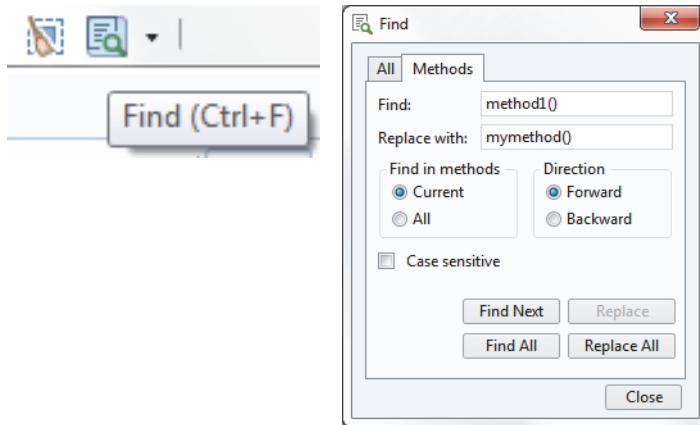
```
1 model.study("std1").run();
2 useGraphics(model.result("pg2"), "form1/graphics1");
3 useGraphics(model.result("pg3"), "form1/graphics1");
4 playSound("success.wav");
```

The bottom window is the 'Errors and Warnings' window, which displays one error:

Method	Line	Message
method1	4	String literal is not properly closed by a double-quote

Find and Replace

Click **Find** in the Quick Access Toolbar to open a dialog box used to find and replace strings in methods, as shown in the figure below.

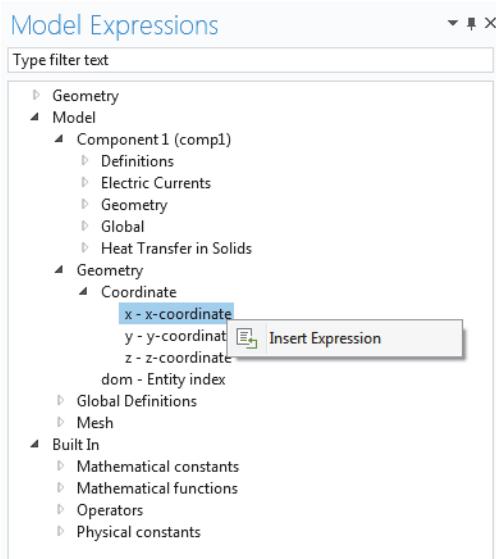


The Quick Access Toolbar is located above the ribbon to the left, in the COMSOL Desktop user interface.

The **All** tab is used to find strings and variables in both the Model Builder and the Application Builder.

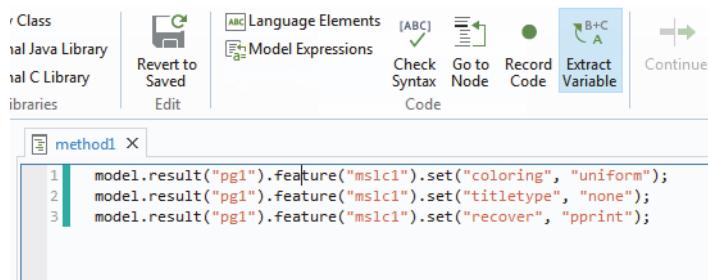
Model Expressions Window

The **Model Expressions** window in the Method editor shows a list of predefined expressions used as input and output arguments. Double-click or right-click one of the items in the list to insert an expression:



Extracting Variables

If you look at the example below, you will notice that each line of code has a repeating prefix.



Readers familiar with object-oriented programming will recognize such a prefix as the name of an object instance. The **Extract Variable** button simplifies code by replacing these instances with a variable name.

In the example above, the mouse pointer has been positioned at the first occurrence of `feature`. Click the **Extract Variable** button to transform the source code into what is shown in the figure below.

The screenshot shows the Eclipse IDE's Method Editor. At the top, there's a toolbar with various icons: Class, Java Library, C Library, Revert to Saved, Language Elements, Model Expressions, Check Syntax, Go to Node, Record Code, and Extract Variable. Below the toolbar is a code editor window titled "method1". The code contains four lines of pseudocode:

```
1 mslc1 = model.result("pg1").feature("mslc1");
2 mslc1.set("coloring", "uniform");
3 mslc1.set("titletype", "none");
4 mslc1.set("recover", "pprint");
```

The code starting with the prefix `feature` has been replaced with the variable `mslc1`. When you click the **Extract Variable** button, an **Extract Variable** dialog box opens where you can enter a suitable variable name in the **Name** field.

Syntax Highlighting, Code Folding, and Indentation

Different language elements in the code are displayed using different styles. Refer to the figure below for an example:

The screenshot shows the Method Editor with syntax highlighting applied to the code. Lines 64 through 83 are visible, each with a line number on the left. The code uses color coding for different elements: blue for keywords like `with`, `if`, and `else`; green for strings like `"pg1"` and `"7"`; and orange for comments like `// 7th frequency`. Some lines are collapsed using the minus sign icon to the left of the line number, demonstrating code folding.

```
64 with(model.result("pg1"));
65     set("loopelevel", new String[]{"7"}); // 7th frequency
66 endwith();
67 useGraphics(model.result("pg1"), "graphics1");
68 zoomExtents("graphics1");
69
70 if (customProgress) {
71     setProgressBar("/progressform/progress1", 100);
72 }
73 else {
74     setProgress(100);
75 }
76 play_sound();
77
78 if (customProgress) {
79     closeDialog("progressform");
80 }
81 else {
82     closeProgress();
83 }
```

This example includes five styles:

- Keywords, such as `if`, `else`, `for`, `while`, `double`, and `int` are displayed in bold, blue font
- Built-in methods are displayed in italic, blue font
- Strings are displayed in red font
- Comments are displayed in green font
- The remainder of the code is displayed in black font

You can customize the syntax highlighting theme in the **Preferences** dialog box. See the next section “Method Editor Preferences”.

You can expand and collapse parts of the code corresponding to code blocks that are part of `for`, `while`, `if`, and `else` statements. This feature can be disabled, as described in the next section “Method Editor Preferences”.

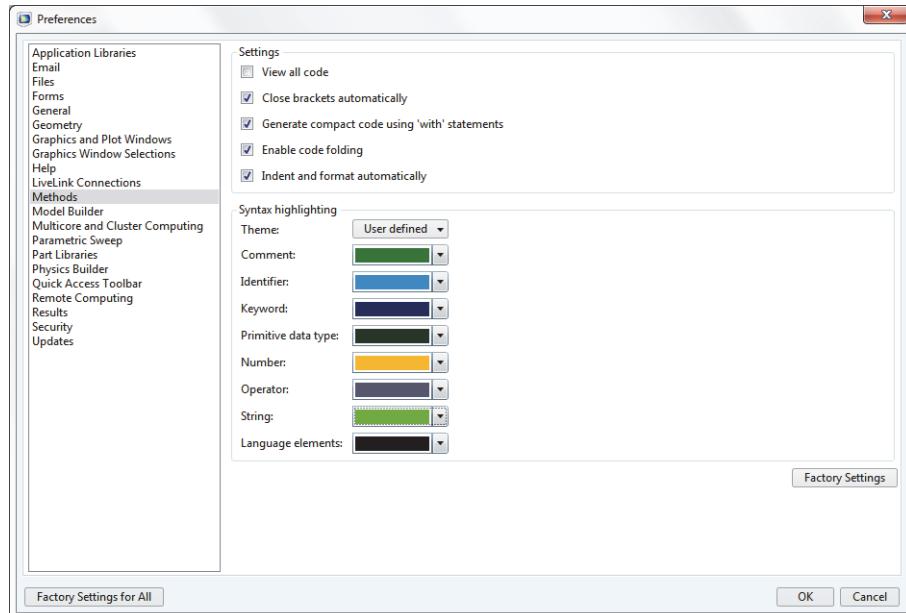
When writing code, press the Tab key on your keyboard to automatically indent a line of code and to insert white spaces where needed. Indentation and whitespace formatting also happen automatically when the keyboard focus leaves the Method editor. You can disable this behavior in **Preferences**, in the **Method** section, by clearing the check box **Indent and format automatically**.

THE NAME OF A METHOD

The **Name** of a method is a text string without spaces. The string can contain letters, numbers and underscore. The reserved names `root` and `parent` are not allowed and Java® programming language keywords cannot be used.

Method Editor Preferences

To access the **Preferences** for the methods, choose **File>Preferences** and select the **Methods** section.



By default, the Method editor only shows the most relevant code. To see all code in a method, select the **View all code** check box.

The check box **Close brackets automatically** controls whether the Method editor should automatically add closing brackets such as curly brackets {}, brackets [], and parentheses () .

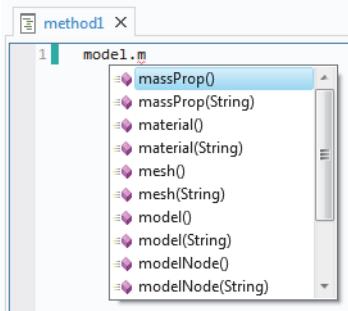
The check box **Generate compact code using 'with' statements** controls the utilization of with statements in automatically generated code. For more information, see “The With Statement” on page 155.

If the check box **Enable code folding** is selected, you can expand and collapse parts of the code corresponding to code blocks associated with `for`, `while`, `if`, and `else` statements.

Under **Syntax highlighting**, the **Theme** list contains two predefined themes, **Modern** (the default) and **Classic**. Choose **User defined** to define a syntax highlighting mode where the colors can be assigned to individual language elements.

Ctrl+Space and Tab for Code Completion

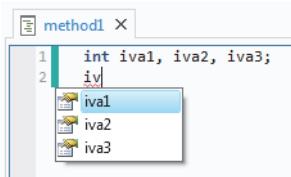
While typing code in the Method editor, the Application Builder can provide suggestions for code completions. The list of possible completions are shown in a separate completion list that opens while typing. In some situations, detailed information appears in a separate window when an entry is selected in the list. Code completion can always be requested with the keyboard shortcut Ctrl+Space. When accessing parts of the model object, you will get a list of possible completions, as shown in the figure below:



Select a completion by using the arrow keys to choose an entry in the list and press the Tab or Enter key to confirm the selection.

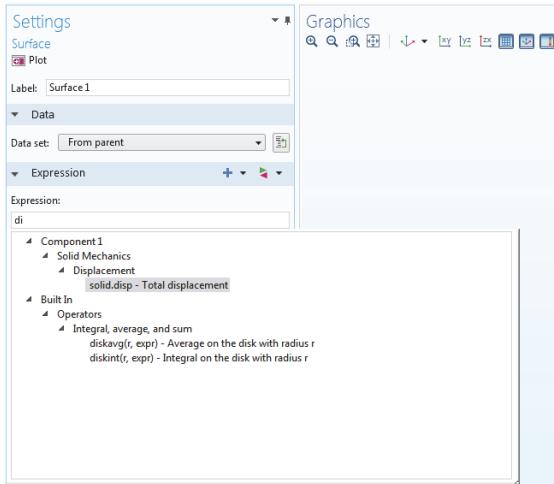
If the list is long, you can filter by typing the first few characters of the completion you are looking for.

For example, if you enter the first few characters of a variable or method name and press Ctrl+Space, the possible completions are shown:



In the example above, only variables that match the string `iv` are shown. This example shows that variables local to the method also appear in the completion suggestions.

The keyboard shortcut Ctrl+Space can also be used in the Model Builder. When typing in an **Expression** field in **Results**, use Ctrl+Space to see matching variables, as shown in the figure below.



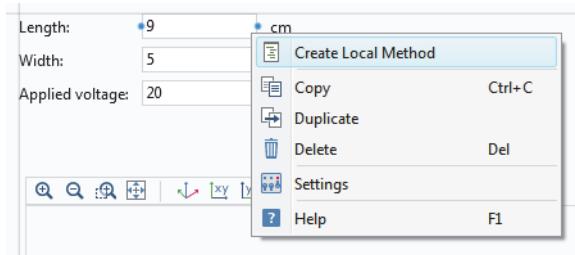
Local Methods

You can add local methods to buttons, menu items, and events. Local methods do not have nodes displayed under the **Methods** node in the application tree. In the method window for a local method, its tab displays the path to its associated user interface component, as shown in the figure below for the case of a check box object.

```
main: checkbox1: onDataChange X
1  setFormObjectEditable("main/inputfield1", !findlength);
2  setFormObjectEditable("main/inputfield5", findlength);
3  setFormObjectEnabled("main/inputfield5", findlength);
```

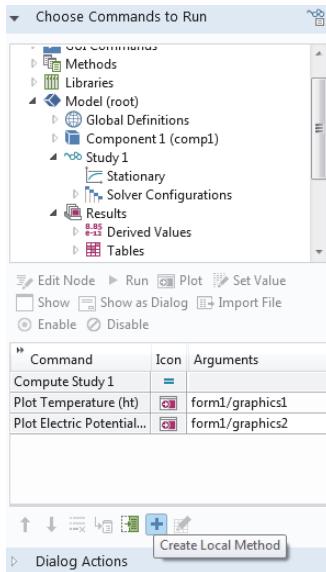
A screenshot of the Method Editor window. It shows a single method entry: 'main: checkbox1: onDataChange'. The code block contains three lines of VBA-like pseudocode: 'setFormObjectEditable("main/inputfield1", !findlength);', 'setFormObjectEditable("main/inputfield5", findlength);', and 'setFormObjectEnabled("main/inputfield5", findlength);'. The tabs at the bottom of the editor are 'Script' and 'Properties'.

In the Form editor, you can right-click a form object and select **Create Local Method** from a menu, as shown in the figure below.



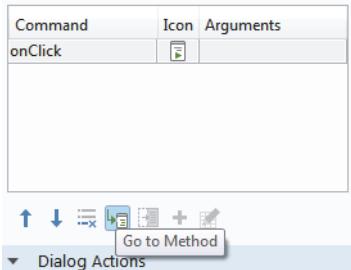
LOCAL METHODS FOR BUTTONS, MENU ITEMS, AND GLOBAL EVENTS

For buttons, ribbons, menus, toolbar items, and global events, you can add a local method by clicking the **Create Local Method** toolbar button under the sequence of commands, as shown in the figure below.



The function of this button is similar to the **Convert to New Method** button, described in the section “Creating a New Method” on page 14. The only difference is that it creates a local method not visible in the global method list in the application tree. It also opens the new method in the Method editor after creating it. Ctrl+Alt+click can be used as a shortcut for creating the local method.

Clicking the button **Go to Method** will open the local method. The figure below shows a call to a local method associated with a button.



To avoid any risk of corrupting code in a local method, using **Convert to New Method** when there is a local method present in the command sequence, is not allowed.

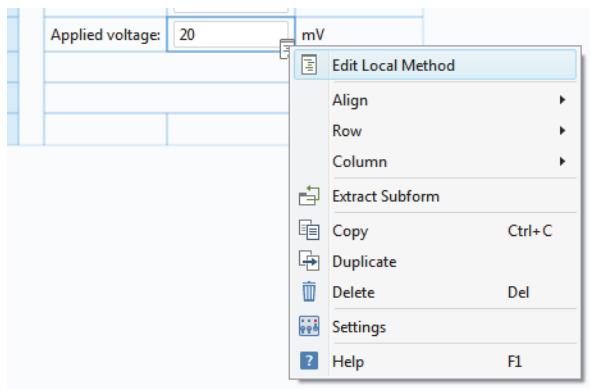
LOCAL METHODS FOR FORM AND FORM OBJECT EVENTS

To add a local method for a form or form object event, click the **Create Local Method** button in the **Events** section of the **Settings** window. The selected **On data change** method changes from **None** to **Local method**, as shown in the figure below, and the Method editor is opened.



To open an existing local method in the Method editor, click the **Go to Source** button. Click the **Remove Local Method** button to delete the local method.

As an alternative to Ctrl+Alt+click, you can right-click a form object and select **Edit Local Method** from its context menu.

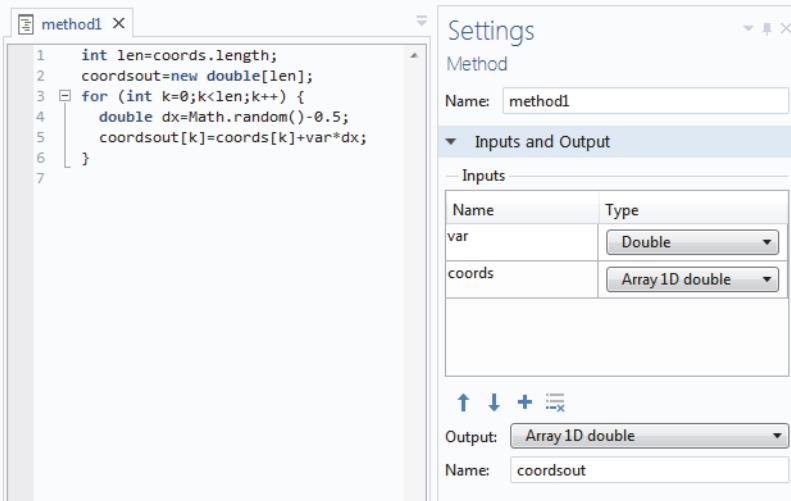


For more information, see “Events” on page 109.

Methods with Input and Output Arguments

A method is allowed to have several input arguments and one output argument. You define input and output arguments in the **Settings** window of an active method window. If the **Settings** window is not visible, click **Settings** in the **Method** tab of the ribbon. The figure below shows a method with two input arguments, `var` and `coords`, and one output, `coordsout`. The method adds random values to

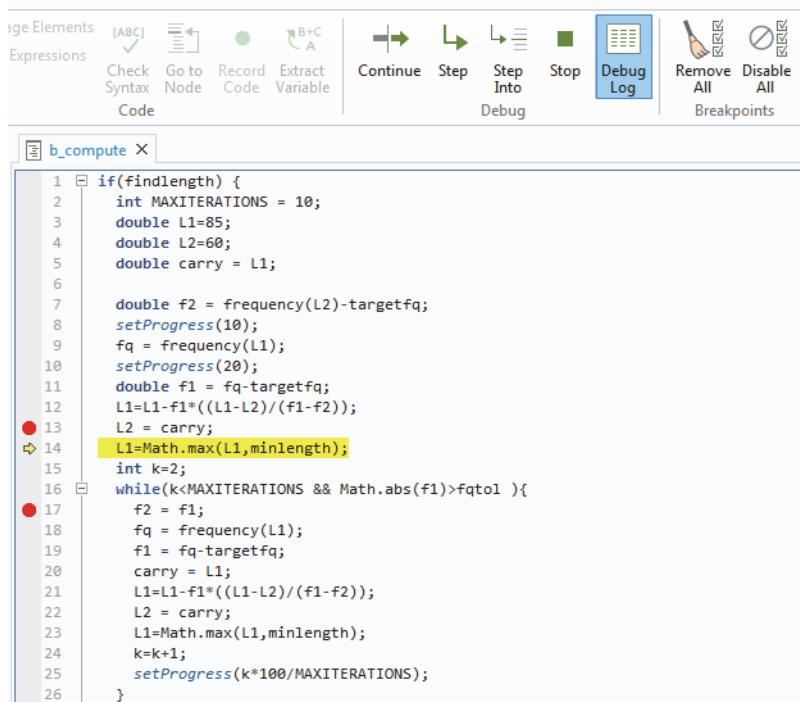
the array, `coords`. The degree of randomness is controlled by the input variable `var`. The new values are stored in the array `coordsout`.



When you call another method from a method, Ctrl+Alt+double-click opens the window for that method. A method is allowed to call itself for the purpose of recursion.

Debugging

For debugging purposes, click in the gray column to the left of the code line numbers to set breakpoints, as shown in the figure below.



The screenshot shows the Java IDE interface. At the top, there's a ribbon bar with several tabs: 'Code' (selected), 'Design Elements', 'Expressions', 'Check Syntax', 'Go to Node', 'Record Code', 'Extract Variable', 'Continue', 'Step', 'Step Into', 'Stop', 'Debug Log' (selected), 'Remove All Breakpoints', and 'Disable All Breakpoints'. Below the ribbon is a code editor window titled 'b_compute'. The code is a Java method with numbered lines from 1 to 26. Lines 13 and 14 are highlighted in yellow, indicating they are breakpoints. Line 14 contains the code 'L1=Math.max(L1,minlength);'. The code itself is as follows:

```
1 if(findlength) {  
2     int MAXITERATIONS = 10;  
3     double L1=85;  
4     double L2=60;  
5     double carry = L1;  
6  
7     double f2 = frequency(L2)-targetfq;  
8     setProgress(10);  
9     fq = frequency(L1);  
10    setProgress(20);  
11    double f1 = fq-targetfq;  
12    L1=L1-f1*((L1-L2)/(f1-f2));  
13    L2 = carry;  
14    L1=Math.max(L1,minlength);  
15    int k=2;  
16    while(k<MAXITERATIONS && Math.abs(f1)>fqtol ){  
17        f2 = f1;  
18        fq = frequency(L1);  
19        f1 = fq-targetfq;  
20        carry = L1;  
21        L1=L1-f1*((L1-L2)/(f1-f2));  
22        L2 = carry;  
23        L1=Math.max(L1,minlength);  
24        k=k+1;  
25        setProgress(k*100/MAXITERATIONS);  
26    }
```

In the ribbon, the **Debug** group contains the tools available for debugging methods. When you run the application, the method will stop at the breakpoints. Click the **Step** button to go to the next line in the method. The figure above shows a method currently stopped at the line highlighted in yellow.

Click **Continue** to run the method up until the next breakpoint. Click **Stop** to stop running the method. Click **Step Into** to step into the next method, if possible. Use **Remove All** to remove all break points. Instead of removing, you can disable all

break points by clicking **Disable All**. Click the **Debug Log** to display debugging messages in a separate **Debug Log** window, as shown in the figure below.

The figure shows two windows from a software development environment. The top window is titled 'b_compute' and contains Java code for a 'compute' method. The bottom window is titled 'Debug Log' and shows the output of the debug log command, displaying variable values at each iteration of a loop.

b_compute

```
12 L1=L1-f1*((L1-L2)/(f1-f2));
13 debugLog("L1:");
14 debugLog(L1);
15 L2 = carry;
16 L1=Math.max(L1,minlength);
17 int k=2;
18 while(k<MAXITERATIONS && Math.abs(f1)>fqtol ){
19     debugLog("k:");
20     debugLog(k);
21     f2 = f1;
22     fq = frequency(L1);
23     f1 = fq-targetfq;
24     carry = L1;
25     L1=L1-f1*((L1-L2)/(f1-f2));
26     debugLog("L1:");
27     debugLog(L1);
28     L2 = carry;
29     L1=Math.max(L1,minlength);
```

Debug Log

Variables	Values
L1:	82.60372113924137
k:	2
L1:	81.26454410908147
k:	3
L1:	81.34452776438428
k:	4
L1:	81.34348357687684

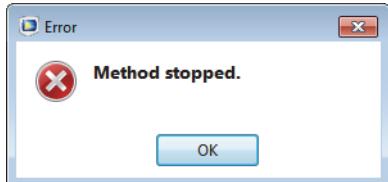
Use the `debugLog` command to display the value of variables in the **Debug Log** window. The code below illustrates using the `debugLog` command to display the values of strings and components of a 1D double array.

```
int len=xcoords.length;
if (selected==0) {
    for (int i = 0; i < len; i++) {
        double divid=double(i)/len;
        xcoords[i] = Math.cos(2.0*Math.PI*divid);
        ycoords[i] = Math.sin(2.0*Math.PI*divid);
        debugLog("x:");
        debugLog(xcoords[i]);
        debugLog("y:");
        debugLog(ycoords[i]);
        debugLog("selected is 0");
    }
}
```

For more information on built-in methods for debugging, see “Debug Methods” on page 310.

Stopping a Method

You can stop the execution of a method while testing an application by using the keyboard shortcut Ctrl+Pause. The following dialog box appears:



The Model Object

The model object provides a large number of methods, including methods for setting up and running sequences of operations. The **Convert to Method**, **Record Code**, **Editor Tools**, and **Language Elements** utilities of the Method editor produce statements using such model object methods. For more information on and example code related to the model object and its methods, see “Appendix C—Language Elements and Reserved Names” in the book *Introduction to COMSOL Multiphysics*, as well as the *COMSOL Programming Reference Manual*.

Language Element Examples

The Java® programming language is used to write COMSOL methods, which means that Java® statements and syntax in general can be used.

UNARY AND BINARY OPERATORS IN THE MODEL OBJECT

The table below describes the unary and binary operators that can be used when accessing a model object, such as when defining material properties and boundary conditions, and in results expressions used for postprocessing and visualization.

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
1	() { } .	grouping, lists, scope
2	^	power
3	! - +	unary: logical not, minus, plus
4	[]	unit
5	* /	binary: multiplication, division
6	+ -	binary: addition, subtraction
7	< <= > >=	comparisons: less-than, less-than or equal, greater-than, greater-than or equal
8	== !=	comparisons: equal, not equal
9	&&	logical and
10		logical or
11	,	element separator in lists

UNARY AND BINARY OPERATORS IN METHODS (JAVA® SYNTAX)

The table below describes the most important unary and binary operators used in Java® code in methods.

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
1	++ --	unary: postfix addition and subtraction
2	++ -- + - !	unary: addition, subtraction, positive sign, negative sign, logical not
3	* / %	binary: multiplication, division, modulus
4	+ -	binary: addition, subtraction
5	< <= > >=	comparisons: less-than, less-than or equal, greater-than, greater-than or equal
6	== !=	comparisons: equal, not equal
7	&&	binary: logical and
8		binary: logical or
9	? :	conditional ternary

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
10	= += -= *= /= %= %>= <>= &= ^= =	assignments
11	,	element separator in lists

ACCESSING A VARIABLE IN THE DECLARATIONS NODE

Variables defined in the Declarations node are available as global variables in a method and need no further declarations.

BUILT-IN ELEMENTARY MATH FUNCTIONS

Elementary math functions used in methods rely on the Java® math library. Some examples:

```
Math.sin(double)
Math.cos(double)
Math.random()
Math.PI
```

THE IF STATEMENT

```
if(a<b) {
    alert(toString(a));
} else {
    alert(toString(b));
}
```

THE FOR STATEMENT

```
// Iterate i from 1 to N:
int N=10;
for (int i = 1; i <= N; i++) {
    // Do something
}
```

THE WHILE STATEMENT

```
double t=0,h=0.1,tend=10;
while(t<tend) {
    //do something with t
    t=t+h;
}
```

THE WITH STATEMENT

```
// Set the global parameter L to a fixed value
with(model.param());
set("L", "10[cm]");
```

```
endwith();
```

The code above is equivalent to:

```
model.param().set("L", "10[cm]");
```

ACCESSING A GLOBAL PARAMETER

You would typically use the Editor Tools window to access global parameters.

Get the global parameter L, defined in the model, and store it in the double variable Length:

```
double Length=model.param().evaluate("L");
```

To return the unit of the parameter L, if any, use:

```
String Lunit=model.param().evaluateUnit("L");
```

To write the value of a double to a global parameter, you need to convert it to a string. The reason is that global parameters are model expressions and may contain units.

Multiply the value of the variable Length with 2 and write the result to the parameter L including the unit of cm.

```
Length=2*Length;  
model.param().set("L", toString(Length)+"[cm]");
```

COMPARING STRINGS

Comparing string values in Java® has to be done with .equals() and not with the == operator. This is due to the fact that the == operator compares whether the strings are the same objects and does not consider their values. The below code demonstrates string comparisons:

```
boolean streq=false;  
String a="string A";  
String b="string B";  
streq=a.equals(b);  
// In this case streq==false  
  
streq=(a==b);  
// In this case streq==false  
  
b="string A";  
streq=a.equals(b);  
// In this case streq==true
```

ALERTS AND MESSAGES

The methods alert, confirm, and request display a dialog box with a text string and optional user input. The following example uses confirm to ask the user if a direct or an iterative solver should be used in an application. Based on the answer,

the `alert` function is then used to show the estimated memory requirement for the selected solver type in a message dialog box:

```
String answer = confirm("Which solver do you want to use?",  
"Solver Selection", "Direct", "Iterative");  
if(answer.equals("Direct")) {  
    alert("Using the direct solver will require about 4GB of memory when  
solving.");}  
} else {  
    alert("Using the iterative solver will require about 2GB of memory when  
solving.");}  
}
```

CREATING AND REMOVING MODEL TREE NODES

Remove a square object `sq1` and a circle object `c1`:

```
model.geom("geom1").feature().remove("sq1");  
model.geom("geom1").feature().remove("c1");
```

Remove a series of geometry objects (circles) with tags `c1`, `c2`, ..., `c10`:

```
for(int n=1;n<=10;n=n+1) {  
    model.geom("geom1").feature().remove("c"+n);  
}
```

The syntax "`c`"`+n` automatically converts the integer `n` to a string before concatenating it to the string "`c`".

To remove all geometry objects:

```
for(String tag : model.geom("geom1").feature().tags()) {  
    model.geom("geom1").feature().remove(tag);  
}
```

However, the same can be achieved with the shorter:

```
model.geom("geom1").feature().clear();
```

Create a circle with tag `c1`, build the circle object (run the node), and create a rectangle with tag `r1`:

```
model.geom("geom1").create("c1", "Circle");  
model.geom("geom1").run("c1");  
model.geom("geom1").create("r1", "Rectangle");
```

Example

Below is a larger block of code that removes, creates, and accesses physics interface feature nodes. It uses the `Iterator` class and method available in the `java.util` package. For more information, see the Java® documentation.

```
String[] flowrate = column1;  
String[] Mw = column2;
```

```

java.util.Iterator<PhysicsFeature> iterator =
model.physics("pfl").feature().iterator();
while (iterator.hasNext()) {
    if (iterator.next().getType().equals("Inlet"))
        iterator.remove();
}

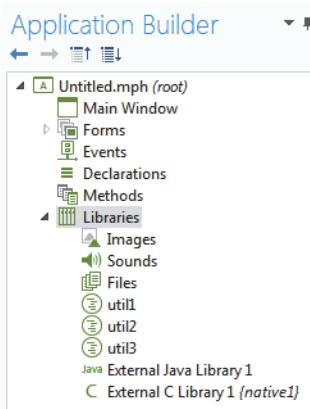
if (flowrate != null) {
    for (int i=0; i<flowrate.length; i++) {
        if (flowrate[i] != "") {
            if (Mw[i] != "") {
                int d = 1+i;
                model.physics("pfl").create("inl" +d, "Inlet");
                model.physics("pfl").feature("inl" +d).setIndex("spec", "3", 0);
                model.physics("pfl").feature("inl" +d).set("qscm0", flowrate[i]);
                model.physics("pfl").feature("inl" +d).set("Mn", Mw[i]);
                model.physics("pfl").feature("inl" +d).selection().set(new int[]{d});
            }
        }
    }
}

```

The need to remove and create model tree nodes is fundamental when writing methods because the state of the model object is changing each time a model tree node is run. In the method above, the number of physics feature nodes are dynamically changing depending on user inputs. Each time the simulation is run, old nodes are removed first and then new nodes are added.

Libraries

In the application tree, the **Libraries** node contains images, sounds, and files to be embedded in an MPH file so that you do not have to distribute them along with the application. In addition, the **Libraries** node may contain Java® utility class nodes and nodes for external Java® and C libraries.



The embedded files can, for example, be referenced in form objects or in methods by using the syntax `embedded:///file1`, `embedded:///file2`, and so on. For example, to reference the image file `compute.png`, use the syntax `embedded:///compute.png`.

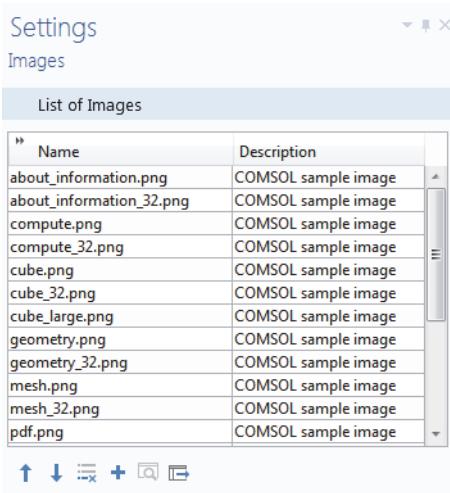
Note that you are not required to have the file extension as part of the file name; instead, arbitrary names can be used. To minimize the size of your MPH file, delete unused images, sounds, or other files.

- ! To manage files loaded by the user of an application at run time, you have several options including using **File** declarations and **File Import** form objects. For more information on files to be loaded at run time, see “File Declarations” on page 126, “File Import” on page 206, and “Appendix C—File Handling and File Scheme Syntax” on page 241.

Images

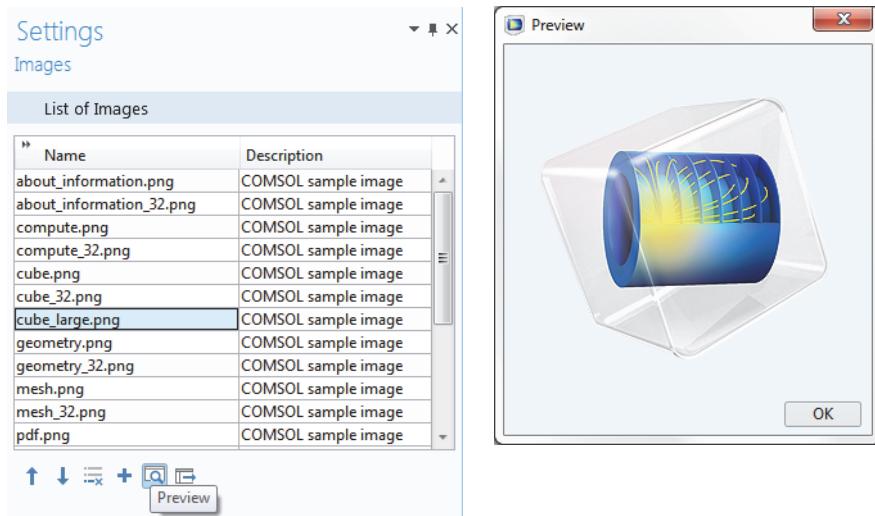
The **Images** library contains a number of preloaded sample images in the PNG-file format. If you wish to embed other image files, click the **Add File to Library** button below the **List of Images**. A large selection of images is available in the COMSOL installation folder in the location `data/images`. Images are used as icons and can

be referenced in image form objects or in methods. For images used as icons, two sizes are supported: 16-by-16 pixels (small) and 32-by-32 pixels (large).



Supported image formats are JPG, GIF, BMP, and PNG.

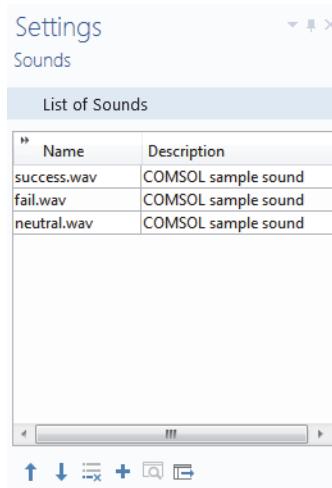
To preview an image, click the name of the image and then click the **Preview** button below the **List of Images**. This opens a dialog box displaying the image, as shown in the figure below.



Click the **Export Selected Image File** button, to the right of the **Preview** button, to export a selected image.

Sounds

The **Sounds** library contains a few preloaded sounds in the WAV file format. If you wish to embed other sound files, click the **Add File to Library** button below the **List of Sounds**. A larger selection of sounds is available in the COMSOL installation folder in the location `data/sounds`.

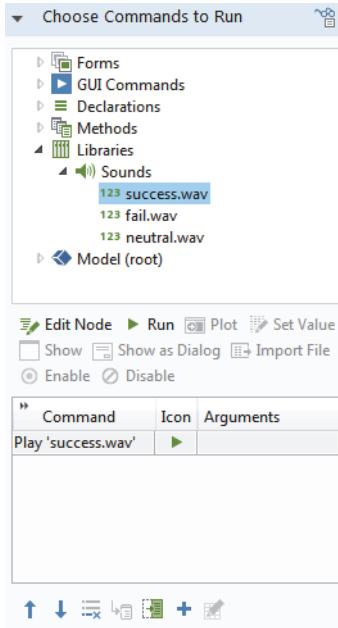


To play a sound, click the name of the sound and then click the **Preview** button below the **List of Sounds**.

Click the **Export Selected Sound File** button, to the right of the **Preview** button, to export a selected sound.

To play a sound in an application, add a command in the **Settings** window of a button, ribbon, menu, or toolbar item. In the **Choose Commands to Run** section,

select the sound and click the **Run** button below the tree. This adds a **Play** command to the command sequence, as shown in the figure below.

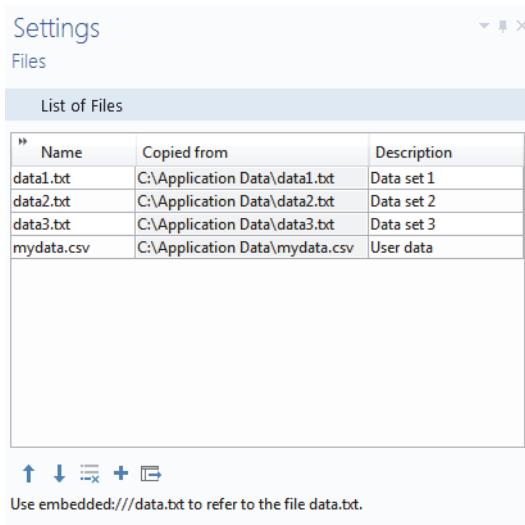


In methods, you can play sounds using the built-in method, `playSound`, such as:

```
playSound("success.wav");
```

Files

The **Files** library is empty by default. Click the **Add File to Library** button to embed files of any type in your application.



Click the **Export Selected File** button, to the right of the **Add File to Library** button, to export a selected file.



The embedded files can be referenced in a method by using the syntax `embedded:///data1.txt`, `embedded:///data2.txt`, and so on. For more information, see “File Declarations” on page 126, “Appendix C—File Handling and File Scheme Syntax” on page 241, and “File Methods” on page 279.

Appendix A—Form Objects

This appendix provides information about forms and form objects and expands upon the section “The Form Editor” on page 37. The items followed by a * in the following list have already been described in detail in that section. The remaining items are discussed in this appendix.

List of All Form Objects

- Input
 - Input Field*
 - Button*
 - Toggle Button
 - Check Box
 - Combo Box
- Labels
 - Text Label*
 - Unit*
 - Equation
 - Line
- Display
 - Data Display*
 - Graphics*
 - Web Page
 - Image
 - Progress Bar
 - Log
 - Message Log
 - Results Table
- Subforms
 - Form
 - Form Collection
 - Card Stack

- Composite
 - File Import
 - Information Card Stack
 - Array Input
 - Radio Button
 - Selection Input
- Miscellaneous
 - Text
 - List Box
 - Table
 - Slider
 - Toolbar
 - Spacer

Toggle Button

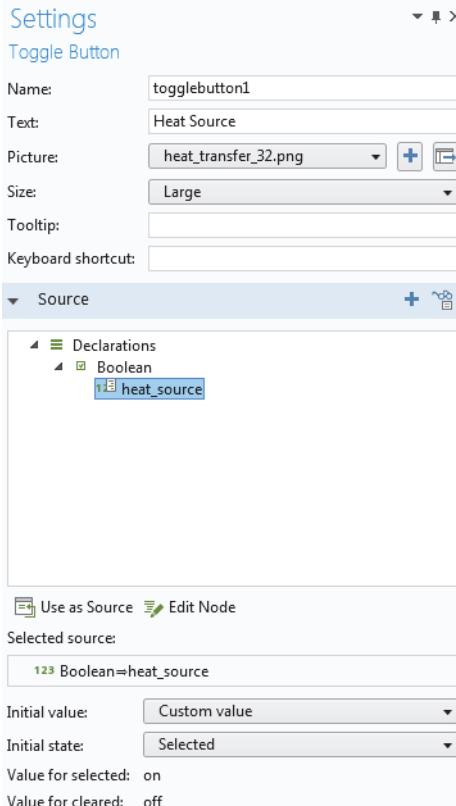
A **Toggle Button** object is a button with two states: selected and deselected, as shown in the figure below.



USING A TOGGLE BUTTON TO ENABLE AND DISABLE A HEAT SOURCE

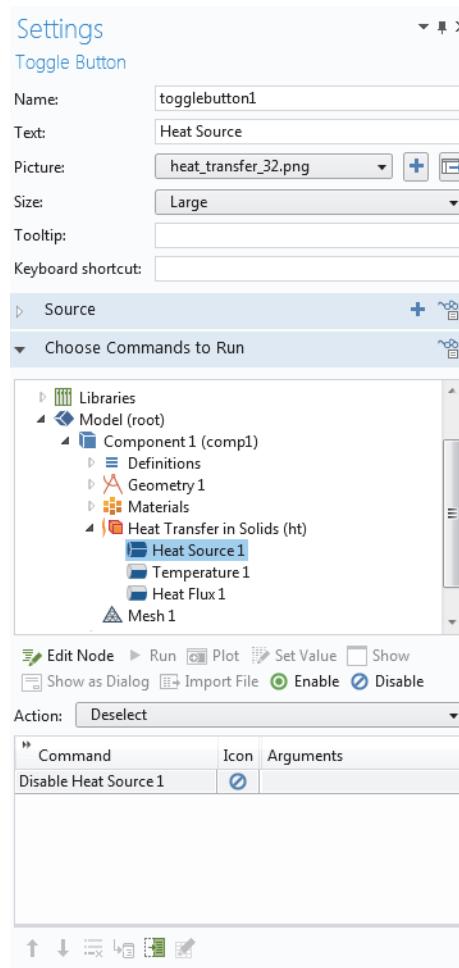
The two states of a toggle button are stored by linking it to a Boolean variable. The figure below shows the **Settings** window of a button that enables and disables

a heat source depending on its state. The Boolean variable `heat_source` is selected in the **Source** section.

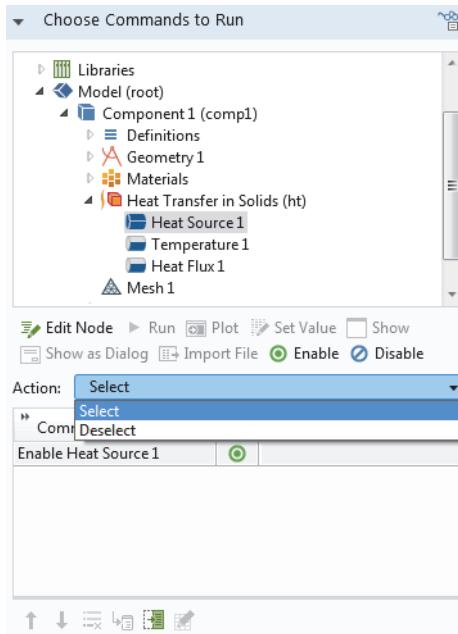


Enabled corresponds to the Boolean variable `heat_source` being equal to `true`, which in turn corresponds to the toggle button being selected. Disabled corresponds to the Boolean variable `heat_source` being equal to `false`, which in turn corresponds to the toggle button being deselected.

Below the **Source** section is a section **Choose Commands to Run** with a choice for **Action** that represents two different commands for **Select** and **Deselect**. The figure below shows the **Settings** window for **Deselect** with a command **Disable Heat Source**.



The next figure shows the command sequence for Select with a command Enable Heat Source.



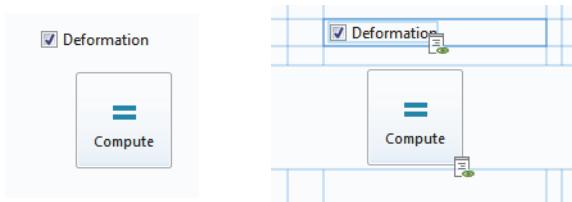
A toggle button is similar to a check box in that it is linked to a Boolean variable. For a toggle button, you define the action by using a command sequence whereas for a check box, you define the action by using an event. This is described in the next section.

Check Box

A **Check Box** has two values: **on** for selected and **off** for cleared. The state of a check box is stored in a Boolean variable in the **Declarations** node.

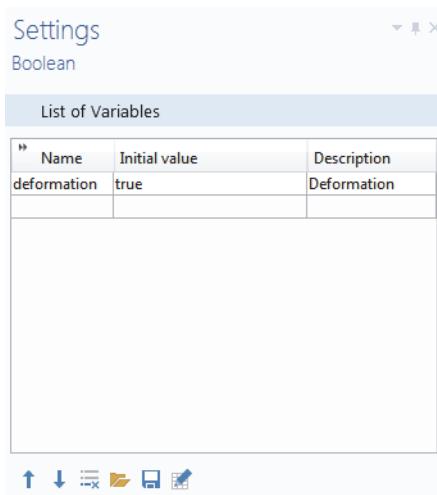
USING A CHECK BOX TO CONTROL VISUALIZATION

The figure below is from an application where a deformation plot is disabled or enabled, depending on whether the check box is selected.

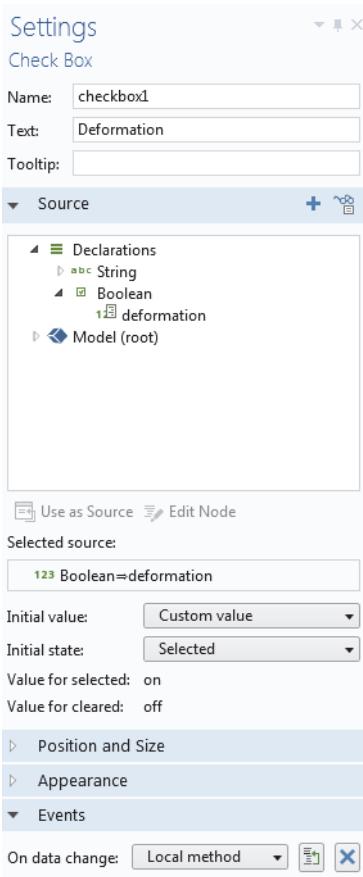


The screenshot on the left shows the running application. The screenshot on the right shows the corresponding form objects in grid layout mode.

In the example below, the state of the check box is stored in a Boolean variable **deformation**, whose **Settings** window is shown in the figure below.



The figure below shows the **Settings** window for the check box.



You associate a check box with a declared Boolean variable by selecting it from the tree in the **Source** section and clicking **Use as Source**.

The text label for a check box gets its name, by default, from the **Description** field of the Boolean variable with which it is associated.

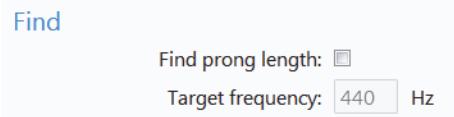
The **Initial value** of the variable `deformation` is overwritten by the **Value for selected** (on) or the **Value for cleared** (off) and does not need to be edited. When used in methods, the values on and off are aliases for `true` and `false`, respectively. These values can be used as Booleans in `if` statements, for example.

The code statements below come from a local method that is run for an **On data change** event when the value of the Boolean variable `deformation` changes.

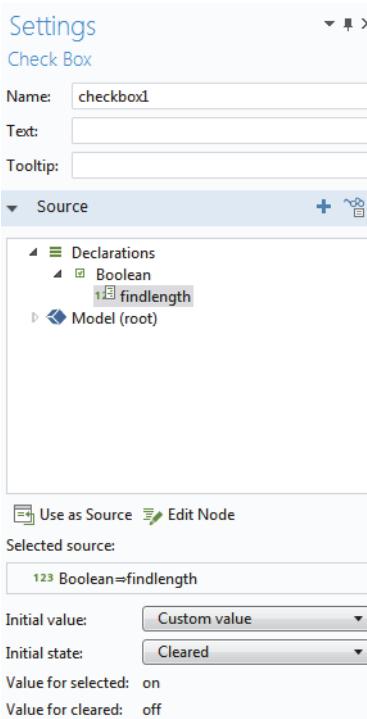
```
model.result("pg1").feature("surf1").feature("def").active(deformation);
useGraphics(model.result("pg1"), "graphics1");
```

USING A CHECK BOX TO ENABLE AND DISABLE FORM OBJECTS

The figure below shows a part of an application where certain input fields are disabled or enabled, depending on if the check box is selected.



The figure below shows the **Settings** window for a check box associated with a Boolean variable `findlength` used to store the state of the check box.



The code statements below come from a local method that is run for an **On data change** event when the value of the Boolean variable `findlength` changes.

```
setFormObjectEditable("main/inputfield1", !findlength);
setFormObjectEditable("main/inputfield5", findlength);
setFormObjectEnabled("main/inputfield5", findlength);
setFormObjectEditable("main/inputfield6", findlength);
setFormObjectEnabled("main/inputfield6", findlength);
solution_state = "inputchanged";
```

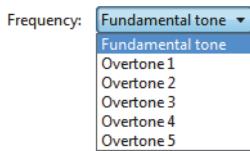
Combo Box

A **Combo Box** can serve as either a combination of a drop-down list box and an editable text field or as a drop-down list box without the capability of editing.

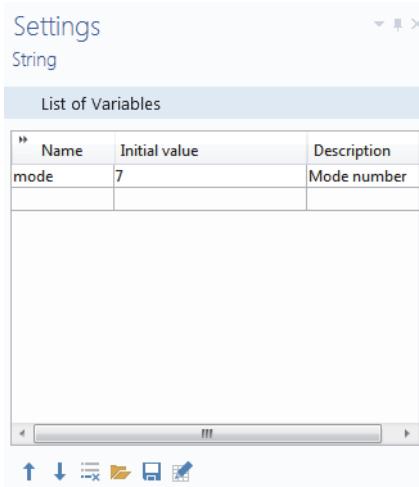
USING A COMBO BOX TO CHANGE PARAMETERS IN RESULTS

To illustrate the use of a combo box, consider an application where the user selects one of six different mode shapes to be visualized in a structural vibration analysis. This example uses a Solid Mechanics physics interface with an Eigenfrequency study and is applicable to any such analysis.

These six mode shapes correspond to six different eigenfrequencies that the user selects from a combo box:

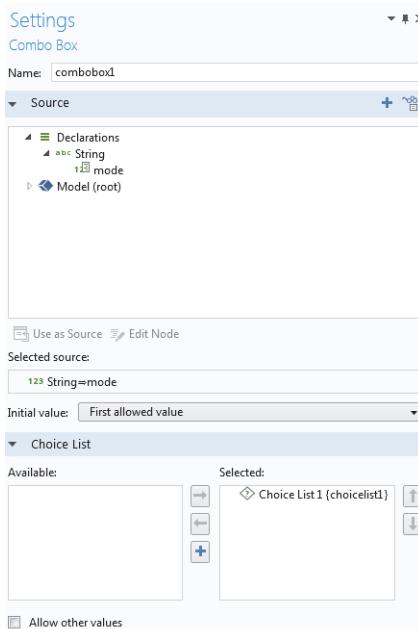


In this example, the combo box is used to control the value of a string variable **mode**. The figure below shows the **Settings** window for this variable.



Selecting the Source

The figure below shows the **Settings** window for this combo box.

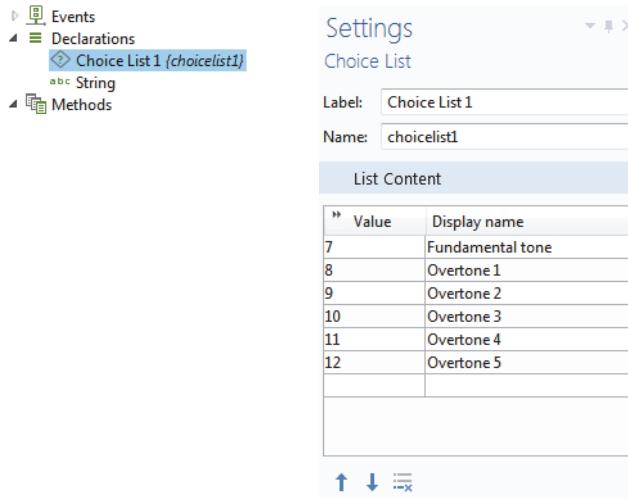


In the **Source** section, you select a scalar variable that should have its value controlled by the combo box and click **Use as Source**. In the **Initial values** list of the **Settings** window of the combo box, choose a method to define a default value for the combo box. The options are **First allowed value** (the default) and **Custom default**. For the **Custom default** option, enter a default value in the associated field. The default value that you enter must exist among the allowed values.

Choice List

The vibrational modes 1–6 correspond to trivial rigid body modes and are not of interest in this application, hence the first mode of interest is 7. A choice list allows you to hide the actual mode values in the model from the user by only displaying the strings in the **Display name** column; the first nonrigid body modes are named Fundamental tone, Overtone 1, Overtone 2, etc.

In the section for **Choice List**, you can add choice lists that contribute allowed values to the combo box. The **Choice List** declaration associated with this example is shown in the figure below.



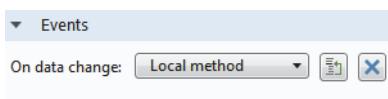
The string variable `mode` is allowed to have one of these six values: 7, 8, 9, 10, 11, or 12. The text strings in the **Display name** column are shown in the combo box.

In the **Settings** window of the combo box, you can select the **Allow other values** check box to get a combo box where you can type arbitrary values. Such combo boxes can accept any value and are not restricted to the values defined by the choice lists. In this example, however, only six predefined values are allowed.

For more information on choice lists, see “Choice List Declarations” on page 124.

Events

In the **Events** section, specify a method to run when the value of the combo box, and thereby the string variable used as the source, is changed by the user. In the present case, the value of the variable `mode` is changed, and a local method is run, as shown below.



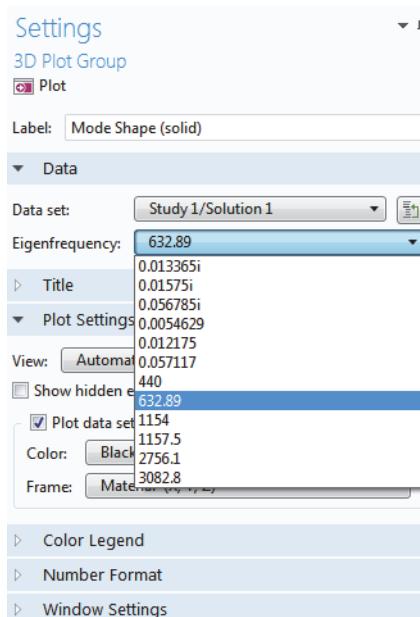
The code for the local method is listed below.

```
with(model.result("pg1"));
  set("looplevel", new String[]{mode});
endwith();
model.result("pg1").run();
```

This code links the value of the string `mode` to the Eigenfrequency setting in the Plot Group `pg1`. In this case, the string `svar` takes the values "7", "8", "9", "10", "11", or "12".

The code above can be generated automatically by using the recording facilities of the Method editor:

- Go to the Model Builder and click **Record a New Method**.
- By default, when using an Eigenfrequency study for a structural mechanical analysis, a **Mode Shape** plot group is created. In this plot group, change the **Eigenfrequency** from mode 7 to mode 8. In the figure below, this corresponds to changing from 440 Hz to 632.89 Hz in the **Settings** window for the **Mode Shape** plot group.



- Click **Stop Recording**.

The resulting code is shown below.

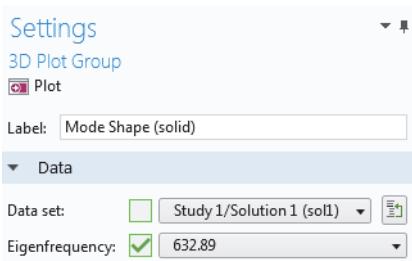
```
with(model.result("pg1"));
    set("loopelevel", new String[]{"8"});
endwith();
model.result("pg1").run();
```

Now change the string "8" with the variable `mode` to end up with the code listing above. This will be stored in a method, say, `method1`. To create the local method

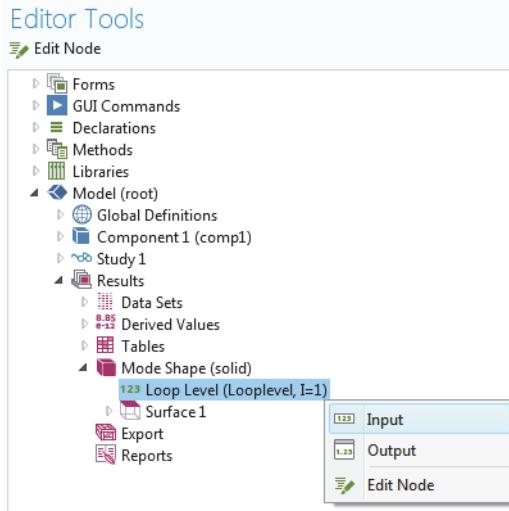
associated with the combo box, copy the code from `method1`. Then, delete `method1`.

Using Model Data Access

A quicker, but less general way, of using a combo box is to use **Model Data Access** in combination with **Editor Tools**. For the example used in this section, you then start by enabling **Model Data Access** and, in the **Settings** window of the **Mode Shape** plot group, select the **Eigenfrequency**, as shown in the figure below.

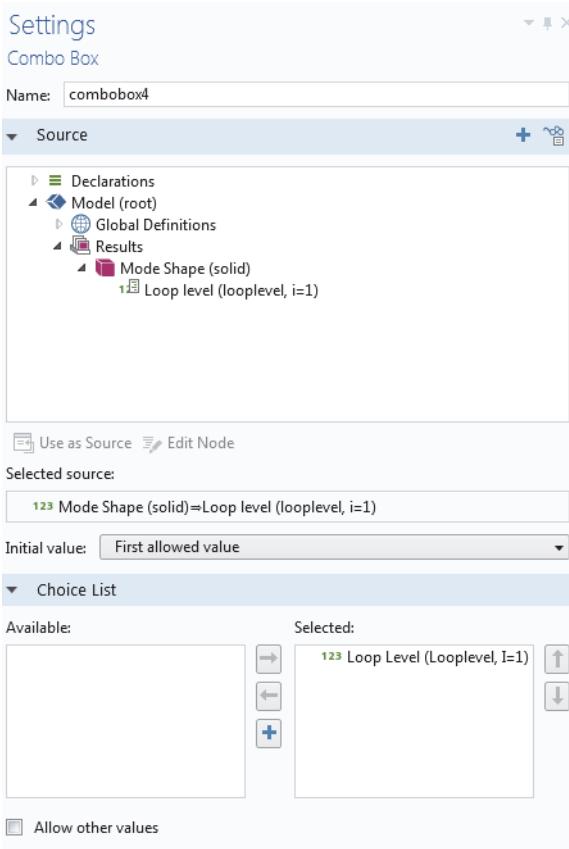


In the **Editor Tools** window, the **Eigenfrequency** parameter is visible as **Loop Level**. To create a combo box, right-click **Loop Level** and select **Input**.



The generic name **Loop Level** is used for a solution parameter. If a solution has two or more parameters, then there are two or more loop levels to choose from.

The figure below shows the **Settings** window of the corresponding combo box.



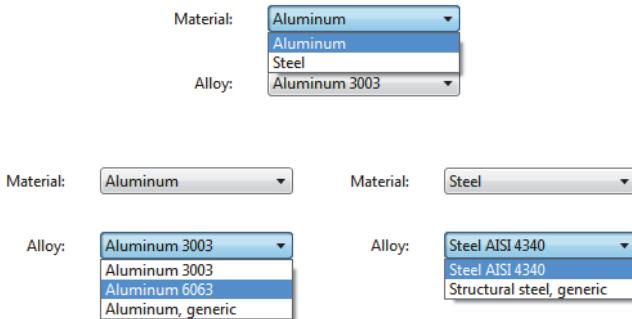
The choice list **Loop Level** is automatically generated when inserting a combo box using **Editor Tools**. Note that a choice list generated in this way is not displayed under the **Declarations** node and cannot be modified by the user. For greater flexibility, such as giving names to each parameter or eigenfrequency value, you need to declare the choice list manually as described in the previous section.

USING A COMBO BOX TO CHANGE MATERIAL

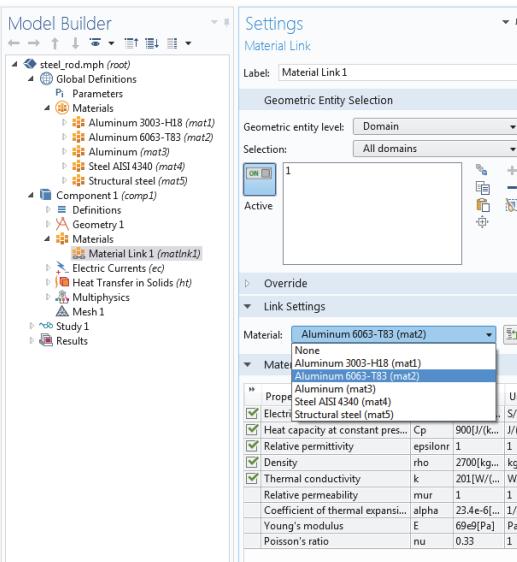
Consider an application where combo boxes are used to select the material. In this case, an activation condition (see “Activation Condition” on page 125) can also be used for greater flexibility in the user interface design.

The figure below shows screenshots from an application where the user can choose between two materials, **Aluminum** or **Steel**, using a combo box named **Material**. A

second combo box called **Alloy** shows a list of **Aluminum** alloys or **Steel** alloys, according to the choice made in the **Material** list.



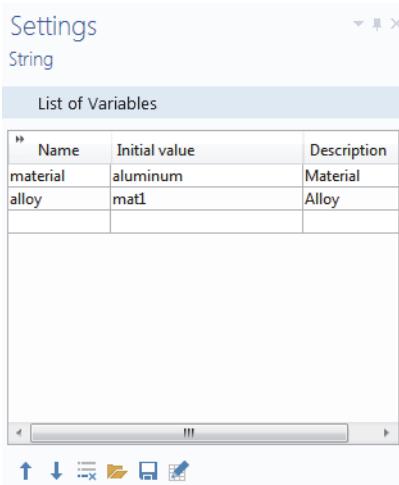
The material choice is implemented in the embedded model using global materials and a material link, as shown below.



Each material is indexed with a string: `mat1`, `mat2`, ..., `mat5`. An event listens for changes to the value of the global variable `alloy`, where the value is controlled by a combo box. When the value is changed, the method listed below is run.

```
with(model.material("matlnk1"));
  set("link", alloy);
endwith();
```

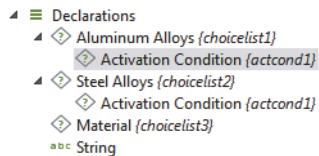
The figure below shows the declaration of two string variables, `material` and `alloy`, which are controlled by the **Material** and **Alloy** combo boxes, respectively.



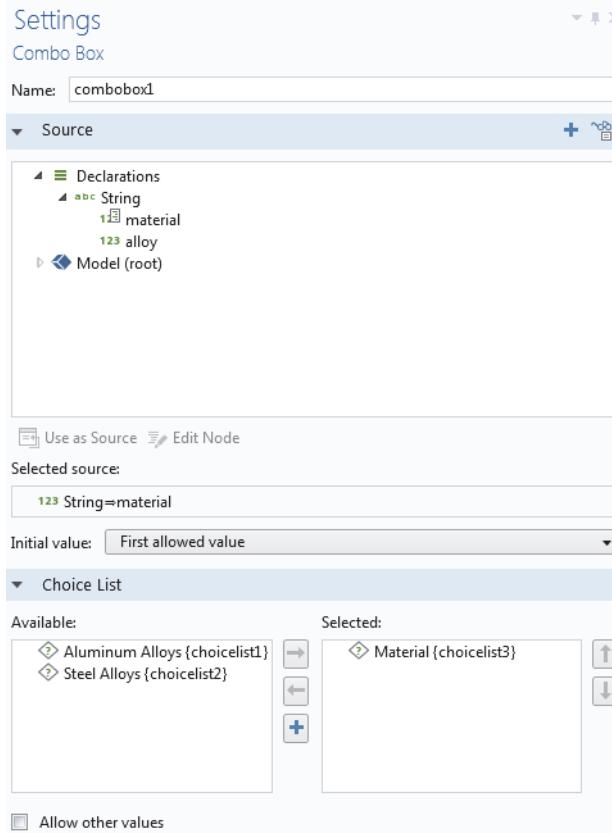
The application utilizes three choice lists: **Aluminum Alloys**, **Steel Alloys**, and **Material**.

Activation Condition

An activation condition is used for the **Aluminum Alloys** and **Steel Alloys** choice lists, as shown in the figure below.

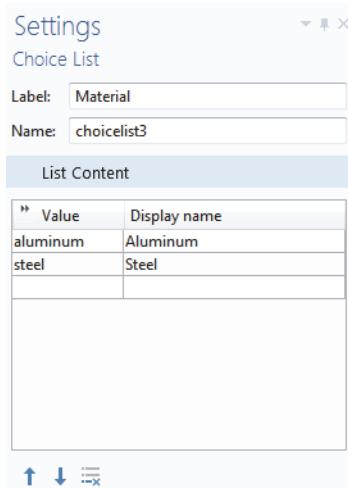


The **Settings** window for the **Material** combo box is shown below.

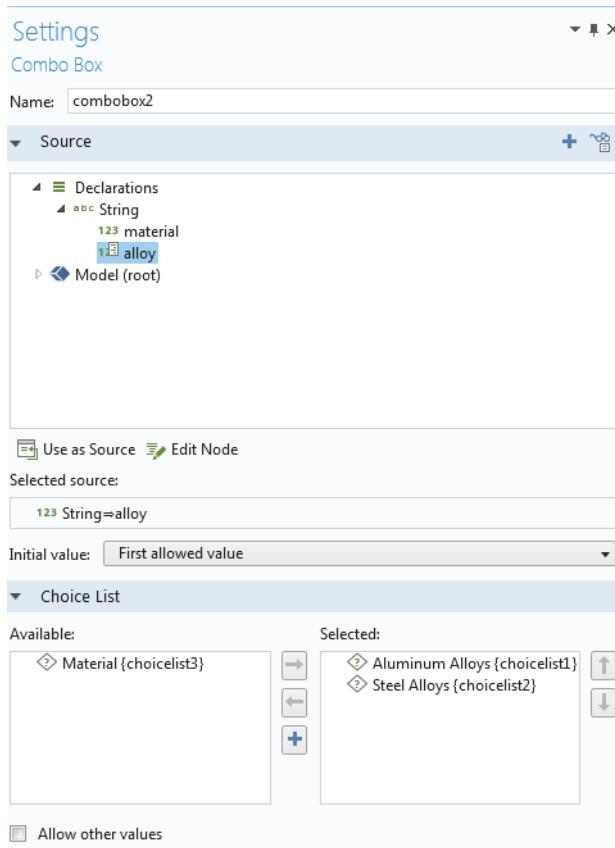


Note that the **Material** combo box uses the `material` string variable as its source. The **Material** choice list is used to define a discrete set of allowed values for the

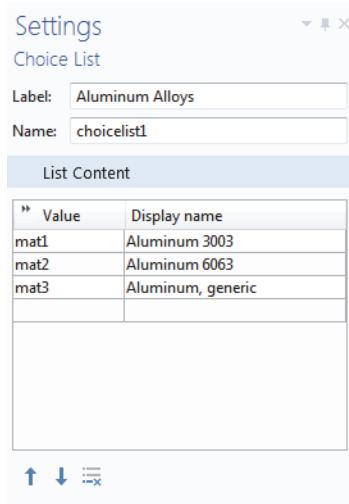
material string variable. The **Settings** window for the **Material** choice list is shown below.



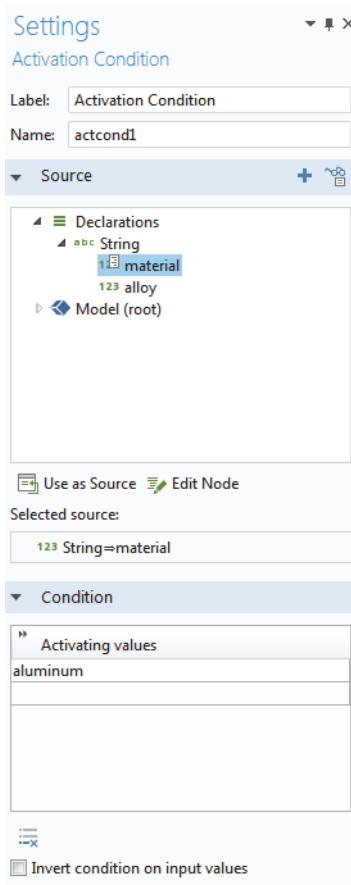
The **Settings** window for the **Alloy** combo box is shown in the figure below.



Note that the **Alloy** combo box uses both the **Aluminum Alloys** and the **Steel Alloys** choice lists. The choice list for **Aluminum Alloys** is shown in the figure below.



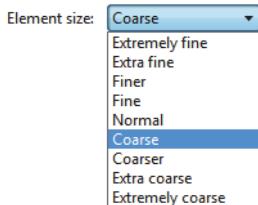
The activation condition for the **Aluminum Alloys** choice list is shown in the figure below.



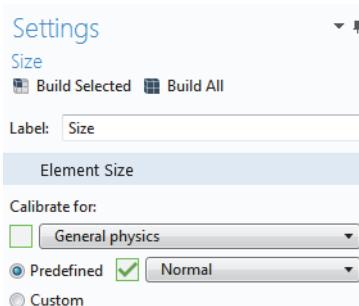
USING A COMBO BOX TO CHANGE ELEMENT SIZE

When creating a combo box, you can use the **Model Data Access** functionality to reproduce the features of a combo box that exists within the Model Builder. For

example, consider an application where a combo box is used to change the element size in a mesh, as in the figure below.



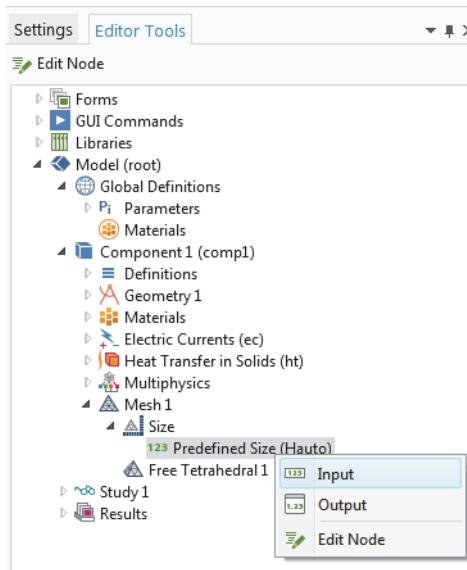
Switch to the Model Builder and select the **Mesh** node (we assume here that the model has just a single mesh). In the **Settings** window of the **Mesh** node, select **User-controlled mesh** (if not already selected). In the **Size** node, directly under the **Mesh** node, select the option **Predefined**. Click **Model Data Access** in the ribbon. This gives access to the combo box for a predefined element size, as shown in the figure below.



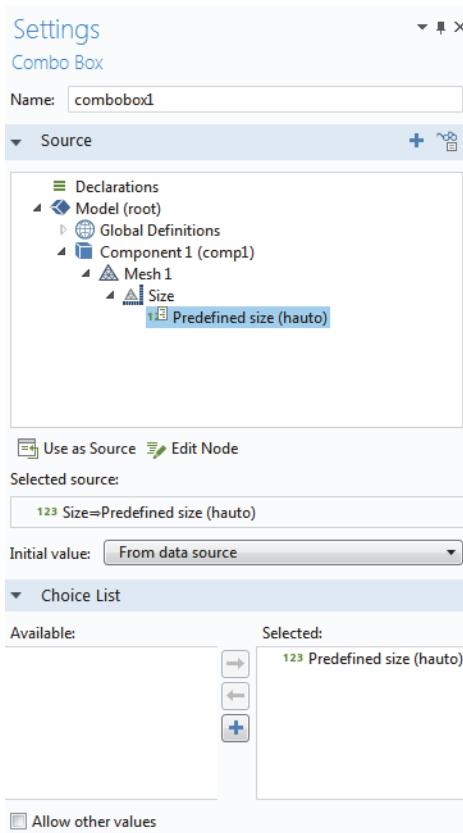
Select the green check box to the left of the list to make it available as a source for a combo box in the Application Builder. Then when you return to the Application Builder, you will find that the choice list for mesh size is now revealed as a

potential **Source** in the **Settings** for a new combo box. To insert the combo box object, you have two alternatives:

- Select **Combo Box** from the **Insert Object** menu in the ribbon. In the **Settings** window for the combo box, select the node **Predefined size (hauto)** in the **Source** section and then click the **Use as Source** button.
- In the **Editor Tools** window, select the node **Predefined size (hauto)** under the **Mesh>Size** node. Then right-click and select **Input**, as shown in the figure below.

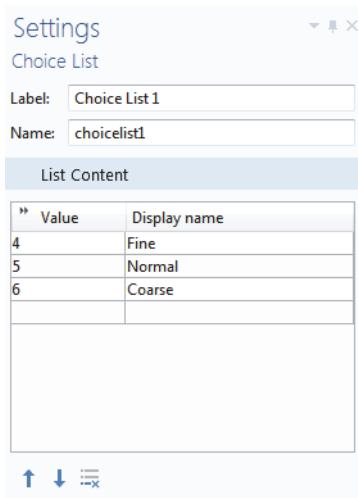


The corresponding **Settings** window for the combo box is shown in the figure below.



Changing the **Initial value** to **From data source** ensures that the element size setting of the model, in this case **Normal**, is used as the default element size in the application. The choice list, **Predefined size (hauto)**, from the Model Builder is now selected as the choice list for your combo box in the Application Builder. This choice list does not appear as a choice list under the **Declarations** node of the application tree because it is being referenced from the Model Builder. Therefore, if you want a list with a more limited set of choices, you cannot edit it. Instead, you have to remove the predefined list as the **Source** of your combo box and create a new choice list of your own by declaring it under the **Declarations** node. For

example, you can create a choice list with three entries, as shown in the figure below.



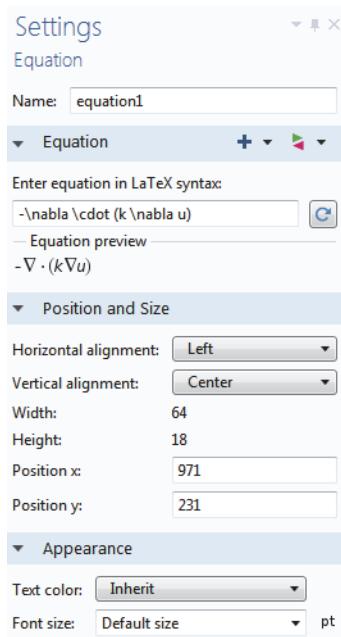
To learn which values are used by the **Element size** list in the model, use **Record a New Method** and change the value from **Normal** to **Fine**, then to **Coarse**, and then back to **Normal**. Click **Stop Recording** and read the values in the auto-generated code. The **Element size** property name is `hauto` and the values for **Fine**, **Normal**, and **Coarse** are 4, 5, and 6, respectively, as implied by the automatically generated code shown in the lines below.

```
with(model.mesh("mesh1").feature("size"));
  set("hauto", "4");
  set("hauto", "6");
  set("hauto", "5");
endwith();
```

For more information on **Element size**, see “Model Data Access for Buttons” on page 82.

Equation

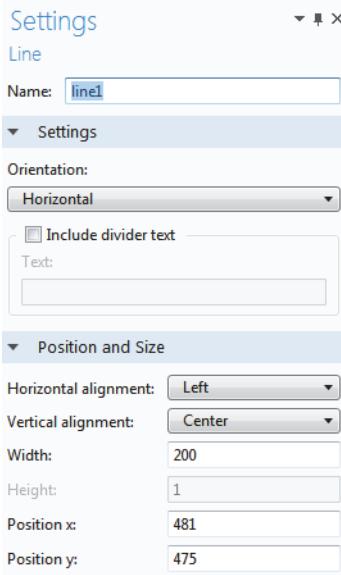
An **Equation** object can display a LaTeX equation by entering the expression in the Enter equation in LaTeX syntax field.



A preview is shown of the rendered LaTeX syntax after leaving the text field.

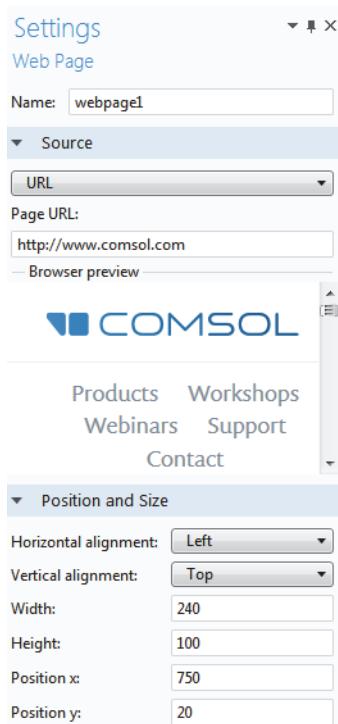
Line

Use the **Line** form object to add a horizontal or vertical line to a form, which can be used, for example, to separate groups of form objects. For the horizontal line option, you can also add text that appears within the line.



Web Page

A **Web Page** object can display the contents of a web page as part of the user interface.

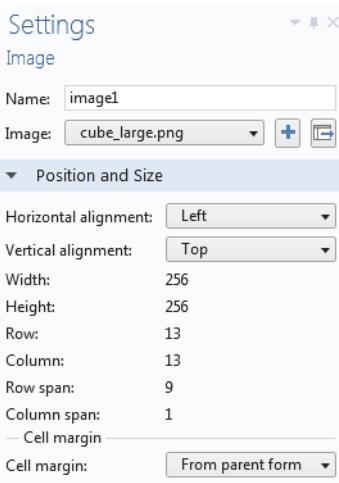


You can specify the page source in four different ways from the **Source** list:

- Use the default option **Page** to enter HTML code in a text area below the list, enclosed by the `<html>` and `</html>` start and end tags.
- Use the **URL** option to link to a web page on the Internet.
- Use the **File** option to point to a local file resource containing HTML code. Type the name of the file in the **File** field or click **Browse** to locate the file on the local file system.
- Use the **Report** option to embed an HTML report. The Browser preview is not active for this option.

Image

Use an **Image** form object to add an image to a form. An image object is different from a graphics object in that an image object is not interactive. Choose an image file from one of the library images, accessible from a drop-down list, or by clicking the **Add Image to Library and Use Here** button to select a file from the local file system. The figure below shows the **Settings** window for an image object referencing the image `cube_large.png` defined in the **Libraries** node.



If you select an image file from your file system, this file will be embedded in the application and added to the list of **Images** under the **Libraries** node.

While you can change the *x*- and *y*-position of the image, the width and height settings are determined by the image file.

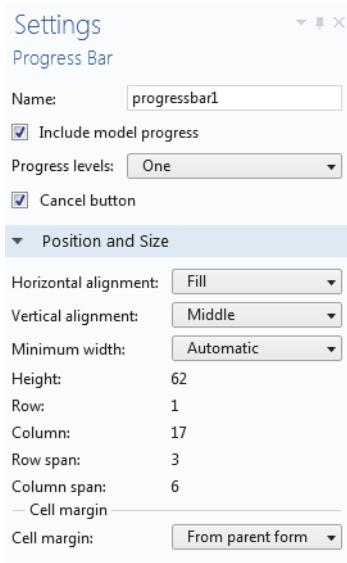


You can paste images from the clipboard to a form window by using **Ctrl+V**. For example, you can copy and paste images from the PowerPoint® slide presentation software. Such images will be added automatically to the **Images** library and embedded in the application. The names for pasted images are automatically set to: `pasted_image_1.png`, `pasted_image_2.png`, etc.

Progress Bar

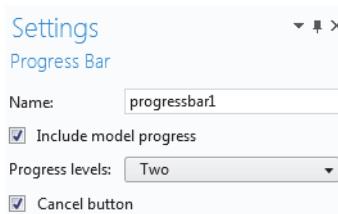
A **Progress Bar** object displays a customized progress bar, or set of progress bars, based on a value that is updated by a method. Use a progress bar to provide

feedback on the remaining run time for an application. The figure below shows the **Settings** window of a progress bar object with one progress level.



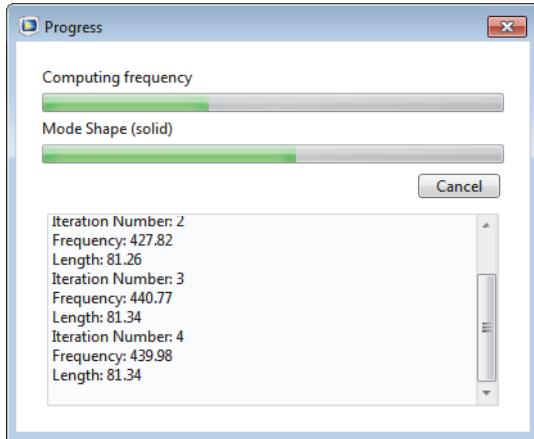
Note that the built-in progress bar that is visible in the status bar of an application is controlled by the **Settings** window of the **Main Window** node. By default, the built-in progress bar shows the progress of the built-in COMSOL Multiphysics core algorithms such as geometry operations, meshing, and solving. By using the `setProgress` method, you can customize the information shown in the built-in progress bar. For more information, see “Progress Methods” on page 312.

The figure below shows the **Settings** window of a progress bar object with two progress levels.

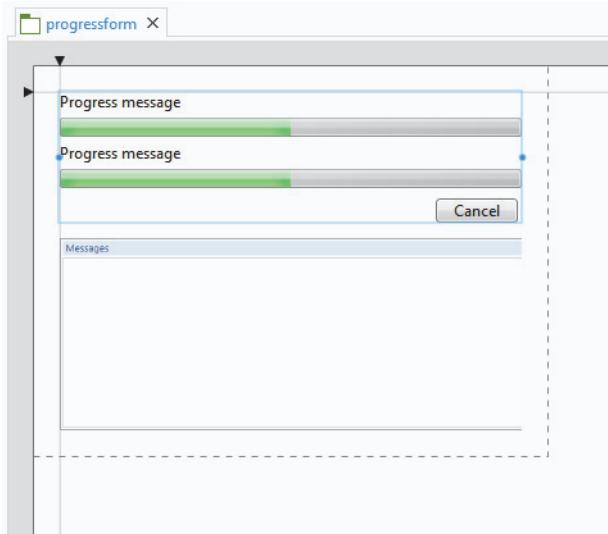


In this example, the progress bar object is part of a form `progressform` used to present a two-level progress bar and a message log.

The figure below shows the corresponding progress dialog box in the running application.



The figure below shows the form `progressform`.



The code segments below show typical built-in methods used to update the progress bar and the message log.

```
// show progress dialog box:  
dialog("progressform");  
setProgressBar("/progressform/progress1", 0, "Computing prong length.");  
  
// code for iterations goes here:
```

```

lastProgress = 20;
// ...

// update message log:
message("Iteration Number: " + k);
message("Frequency: " + Math.round(fq*100)/100.00);
message("Length: " + Math.round(L1*100)/100.00);

// update progress bar:
setProgressInterval("Computing frequency", lastProgress,
k*100/MAXITERATIONS);
// more code goes here:
// ...

// finished iterating:
setProgressBar("/progressform/progress1", 100);
closeDialog("progressform");

```

In the example above, the central functionality for updating the two levels of progress bars lies in the call

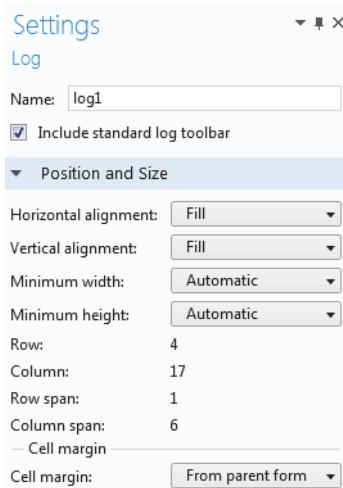
```
setProgressInterval("Computing frequency", lastProgress,
k*100/MAXITERATIONS).
```

For detailed information on the built-in methods and their syntax, see “Appendix E—Built-in Method Library” on page 264 and “Progress Methods” on page 312.

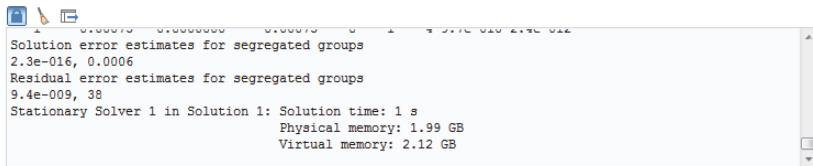
Log

The **Log** form object adds a log window that displays messages from the built-in COMSOL Multiphysics core algorithms such as geometry operations, meshing, and solving.

The **Include standard log toolbar** check box is selected by default. When selected, the toolbar in the **Log** window that you see in the COMSOL Desktop is included in the application.



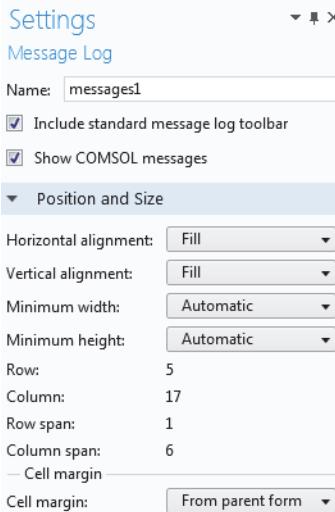
The figure below shows a part of an application user interface containing a log window.



Message Log

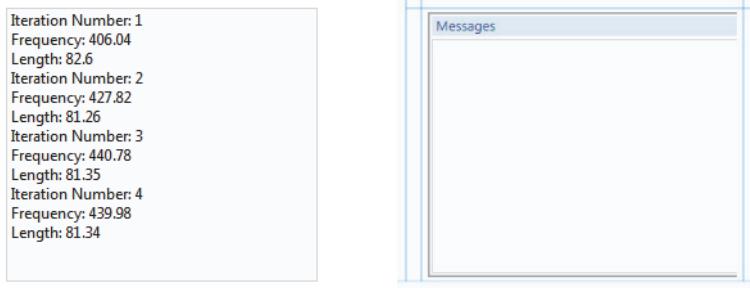
The **Message Log** object adds a window where you can display messages to inform the user about operations that the application carries out. Implement this feature

using the built-in `message` method with syntax: `message(String message)`. See also “GUI-Related Methods” on page 293.



The **Include standard message log toolbar** check box is selected by default. When selected, the toolbar in the **Messages** window that you see in the COMSOL Desktop is included in the application. The **Show COMSOL messages** check box is selected by default to enable messages from the built-in COMSOL Multiphysics core algorithms such as geometry operations, meshing, and solving. Clear the check box to only allow messages from the application itself.

The figure below shows a customized message window with convergence information from a method (left) and the corresponding **Message Log** form object (right).

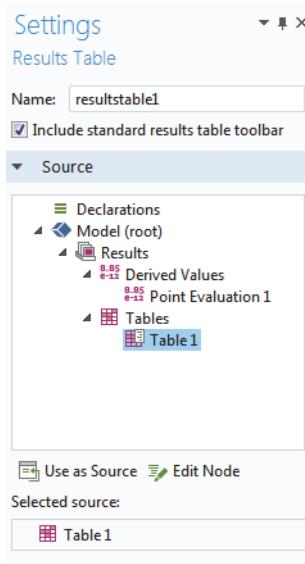


Results Table

The **Results Table** object is used to display numerical results in a table.

Time (s)	Temperature (degC), Point: (0.1, 0.3)
0	0.0000003365779548403225
10	0.0071499008730029345
20	0.10522781133681747
30	0.8747185765842573
40	3.407663238059911
50	8.385166250608506
60	15.835540221745532
70	25.366912864333813
80	36.42264267649
90	48.73369219163317
100	61.88339814841544
110	75.47835598614932
120	89.37132906334898
130	103.40392660608916
140	117.41553076867922
150	131.41380951262022
160	145.32287441615495
170	150.13507213627514

The source of the results table data is one of the child nodes to **Derived Values** or **Tables** under **Results**. In the figure below, a **Table** node is used as the source (by selecting this option in the tree and then clicking **Use as Source**.)



RESULTS TABLE TOOLBAR

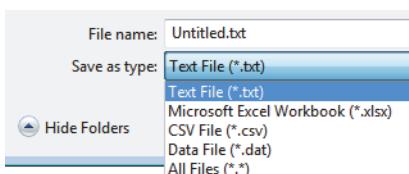
The **Include standard results table toolbar** check box is selected by default. When selected, a toolbar is included that provides the following buttons:

- **Full Precision**
- **Automatic Notation**
- **Scientific Notation**
- **Decimal Notation**
- **Copy Table and Headers to Clipboard**
- **Export**

The **Export** button is used to export to the following file formats:

- Text File (.txt)
- Microsoft® Excel® Workbook (.xlsx)
 - Requires LiveLink™ for Excel®
- CSV File (.csv)
- Data File (.dat)

as shown in the figure below.



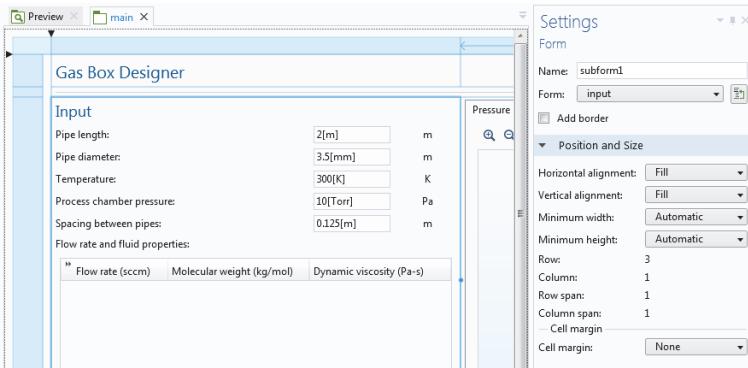
CONTROLLING RESULTS TABLES FROM METHODS

There is a built-in method `useResultsTable()` for changing which table is shown in a particular results table form object. For more information on this built-in method, see “GUI-Related Methods” on page 293.

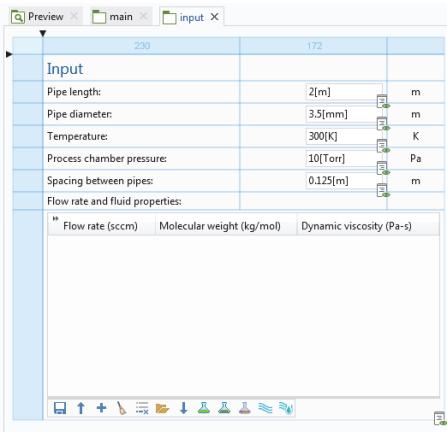
Form

A form object of the type **Form** is used to organize a main form in one or more subforms. To embed a subform, you create a link to it by selecting the form you would like to link to from the **Form** reference of the **Settings** window for the

subform. The figure below shows an example where one of the cells of the form main has a link to the form input.

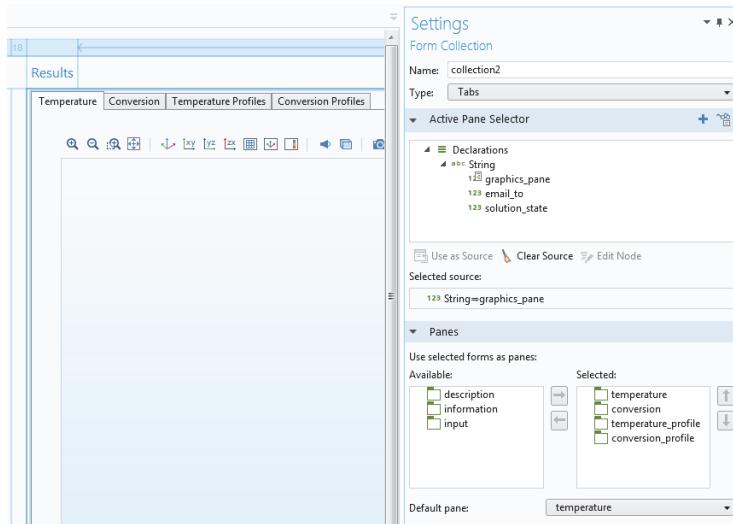


The figure below shows the referenced form input.



Form Collection

A **Form Collection** object consists of several forms, or panes, presented in a main form. In this example, there are four forms that appear as tabs in a single main window.



There are four different layout options. From the **Type** list, choose between:

- **Tabs**, the default setting, which displays the forms using tabbed panes.
- **List**, which displays a list to the left of the form panes, where you can select the form to display.
- **Sections**, which displays each form in a separate section.
- **Tiled or tabbed**, which displays the forms in one of two ways depending on the value of a Boolean variable. For more information, see the description later in this section.

In the **Panes** section, in the **Use selected forms as panes** list, each form represents a pane. These will be displayed in the application in the order they appear in the list. You can change the order by clicking the **Move Up** and **Move Down** buttons to the right.

You can control which tab (or list entry) is active by linking to a string variable in the section **Active Pane Selector**.

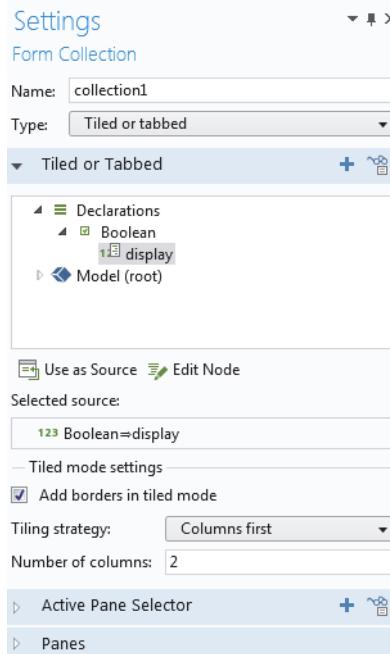
The string variable needs to be equal to one of the form names in the form collection such as `temperature` or `conversion` in the example above. Otherwise, it will be ignored.

If you change the value of the pane selector pane in the above example, in a method that will be run at some point (a button method, for example), then the pane with the new value will be activated. Example:

```
pane="conversion"; /* Activate the conversion pane on completion of this  
method */
```

For a form collection with the **Type** set to **Sections**, the **Active Pane Selector** has no effect. Using an **Active Pane Selector** is optional and is only needed if you wish to control which tab is active by some method other than clicking its tab. To remove a string variable used as an **Active Pane Selector**, click the **Clear source** toolbar button under the tree.

The **Tiled or tabbed** option displays the forms in one of two ways depending on the value of a Boolean variable used as source in a **Tiled or Tabbed** section at the top of the Settings window.



The tabbed mode is identical to a form collection with the **Type** set to **Tabs**. In tiled mode, all the forms are shown simultaneously in a grid. The layout for the tiled mode can be controlled by the settings in the subsection **Tiled mode settings**.

Card Stack

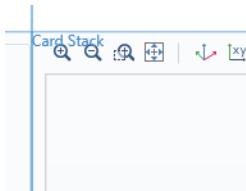
A **Card Stack** is a form object that contains cards.

A **Card** is another type of form object, one that is only used in the context of a card stack. Flip between cards in a card stack to show one at a time. You associate a card stack with a data source that controls which card to show. Each card specifies a value that is compared against the data source of the card stack. The card stack shows the first card with the matching value. If no cards match, nothing is shown.

USING A CARD STACK TO FLIP BETWEEN GRAPHICS OBJECTS

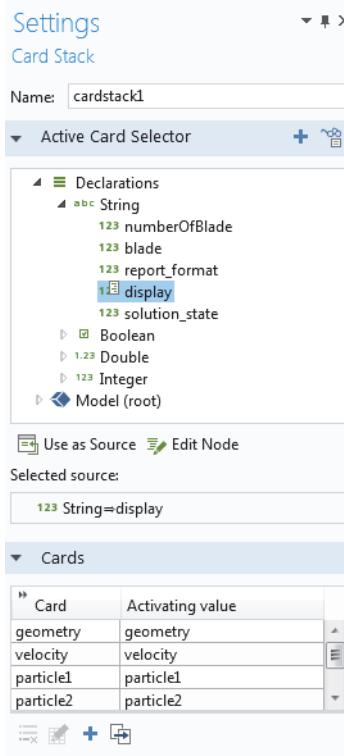
Consider an application where the graphics shown to the user depend on the value of a scalar variable. This variable may change when a user clicks, for example, a radio button. The variable may also change depending on a computed value; for example, the value of a **Global Evaluation** node in the model tree.

The figure below shows the card stack object in the Form editor.



In this example, the card stack contains cards with graphics objects.

The figure below shows a card stack **Settings** window with five cards and a string variable **display** as its **Active Card Selector**.

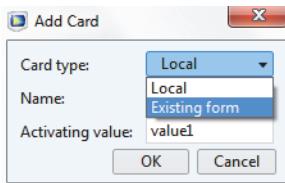


By clicking a card in the table of cards in the **Cards** section, followed by clicking one of the toolbar buttons below the table, you can perform the following operations on cards:

- **Delete**
- **Edit**
- **Add Card**
- **Duplicate**

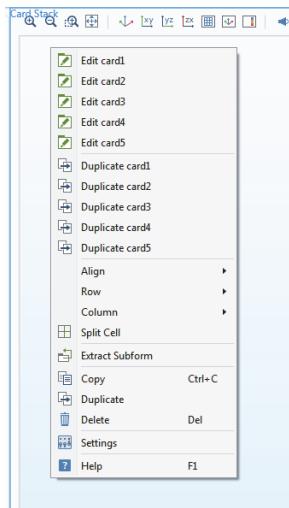
In the card stack **Settings** window, the table in the **Cards** section contains the cards in the **Card** column and their associated activating values in the **Activating value** column. The stack decides which cards to display based on their activating values. In this example, the activating values are the strings **geometry**, **velocity**, **particle1**, etc.

Clicking the **Add Card** button displays the following dialog box.



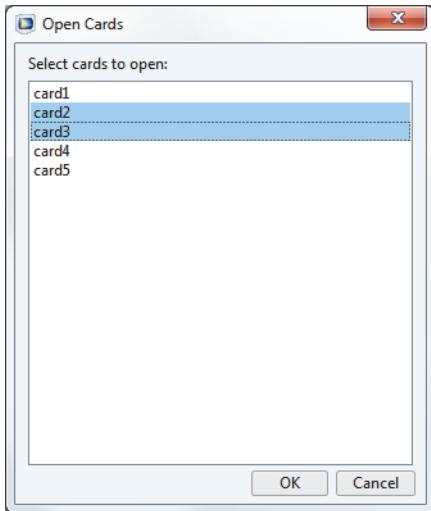
By default, the **Card type** is set to **Local**, which means that the card is defined locally in its containing card stack object. If the **Card type** is set to **Existing form**, then you can instead select one of the existing forms. The settings for an **Existing form** are accessed directly from the Form editor by clicking its node or by clicking the **Edit** button in the **Card** section of the corresponding card stack **Settings** window.

To access locally defined cards, right-click the card stack in a form window to select between the different cards in a card stack, as shown in the figure below.

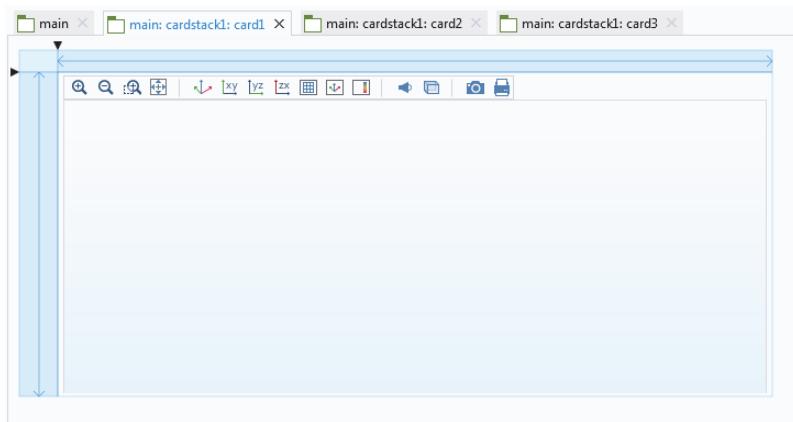


From this menu, you can also duplicate cards.

To edit cards, you can also use Alt+click, which opens a dialog box that lets you select multiple cards at once.



The figure below shows `card1` with its graphics form object.



File Import

A **File Import** object is used to display a file browser with an associated input field for browsing to a file or entering its path and name. It is used to enable file import

by the user of an application at run time, when the file is not available in the application beforehand.

Consider an application where a CAD file can be selected and imported at run time, as shown in the figure below.

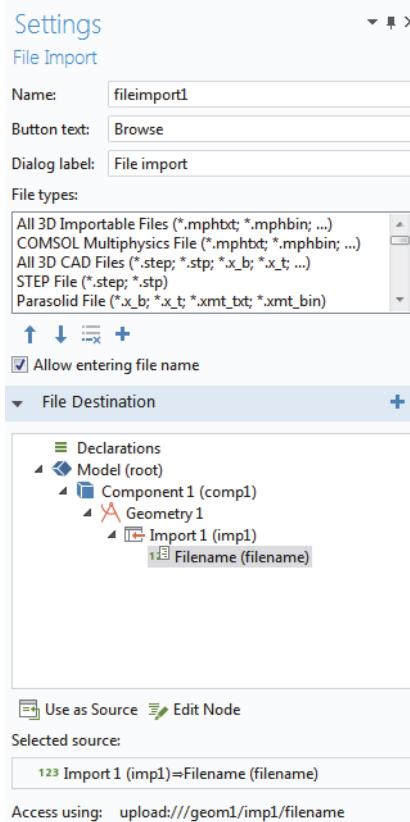


The corresponding **File Import** object is shown in the figure below.



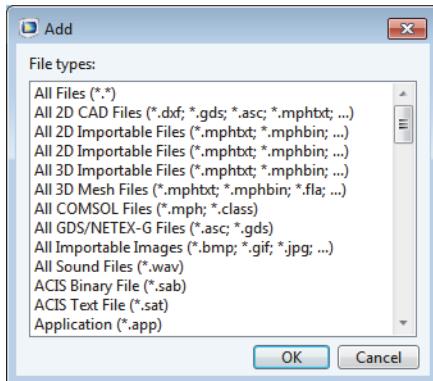
The **Settings** window for the **File Import** object has a section **File Destination**. In this section, you can select any tree node that allows a file name to be input. This is

shown in the figure below, where the **Filename** for a geometry **Import** node is selected.



In this application, the **File types** table specifies that only CAD files are allowed. You can further control which **File types** are allowed by clicking the **Add** and **Delete**

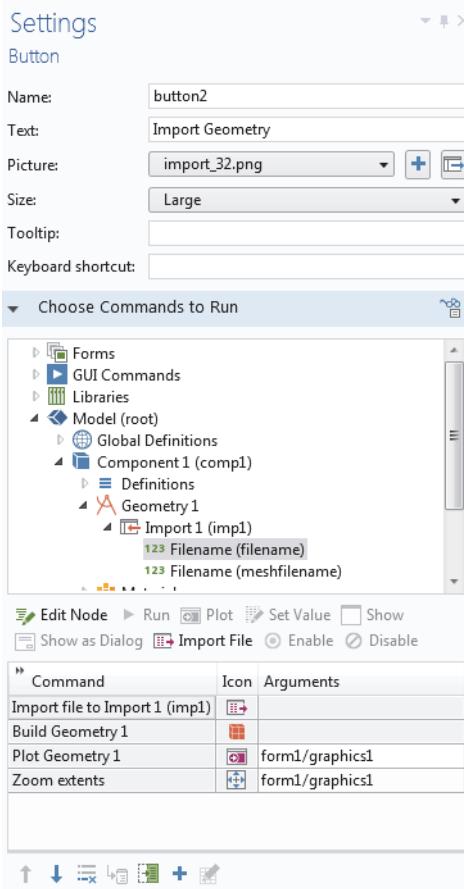
buttons below the list of **File types**. Clicking the **Add** button displays the dialog box shown below:



ALTERNATIVES TO USING A FILE IMPORT OBJECT

If an input field for the file path and name is not needed, then there are other methods for file import which allow a user to pick a file in a file browser. For example, you can use a menu, ribbon, toolbar item, or a button. In that case, you use an **Open File** command in the command sequence for that button or item.

The figure below shows the **Settings** window of a button used to import a CAD file.



A **File Import** object can also reference a **File** declaration. For more information, see “File Declarations” on page 126. For more information on file handling in general, see “Appendix C—File Handling and File Scheme Syntax” on page 241.

Information Card Stack

An **Information Card Stack** object is a specialized type of **Card Stack** object used to display information on the relationship between the inputs given by the user to an application and the solution. The figure below shows a portion of a running

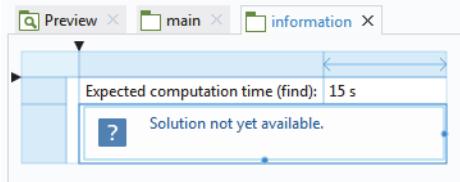
application in which an information card stack is used together with information on the expected computation time.

Information

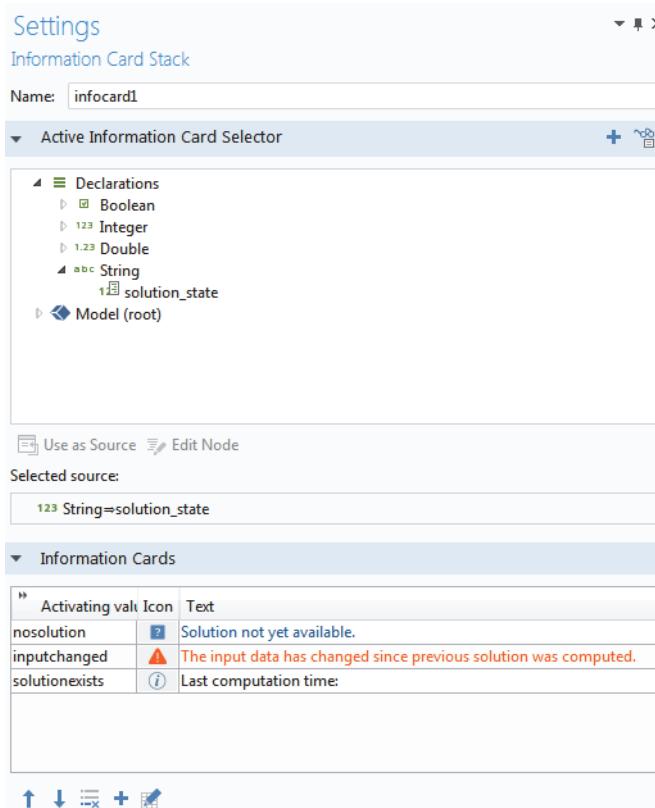
Expected computation time (find): 15 s

 Last computation time: 13 s

The corresponding form objects are shown below:

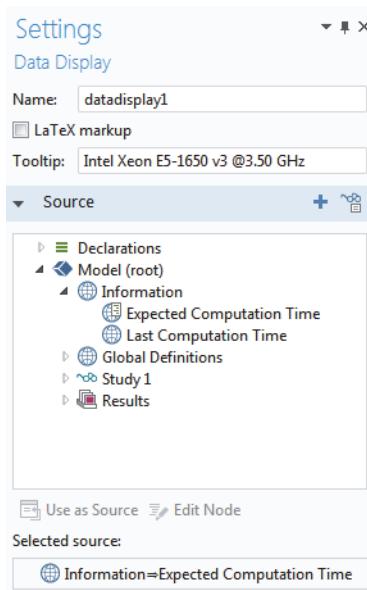


The figure below shows the **Settings** window where a string variable `solution_state` is used as the source.



There are similarities with a **Card Stack** object, but for the **Information Cards**, each card has an icon and a text. In the figure above, the string variable values `nosolution`, `inputchanged`, and `solutionexists` control which information card is shown.

In this example, the information card stack is accompanied by a data display object where a model tree information node for the **Expected Computation Time** is used as the source. The figure below shows its **Settings** window.

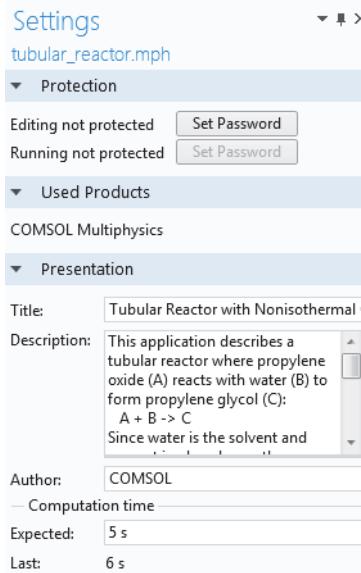


Note that information nodes in the model tree are only shown when working with the Application Builder. They are made available in the **Source** section in the **Settings** window for form objects, when applicable.

You can also find information nodes with **Last Computation Time** under each study. The information node **Last Computation**, found directly under the **Model** node, will correspond to the computation time for the last computed study.

Information nodes can be used as a source for input field objects, text objects, and data display objects. For input field objects and text objects, in order for the information nodes to be accessible, the **Editable** check box has to be cleared.

The **Expected Computation Time** take its data from the root node of the application tree, as shown below:



If the computation time is predominantly spent in a method, for example, if the same study is called repeatedly, then you can manually measure the computation time by using the built-in methods `timeStamp` and `setLastComputationTime`. For more information, see “Date and Time Methods” on page 318.

Array Input

An **Array Input** object has an input table used to enter array or vector-valued input data. An array input object supports string arrays as data sources. You can add an optional label, symbol, and unit.

USING AN ARRAY INPUT OBJECT FOR 3D POINT COORDINATE INPUT

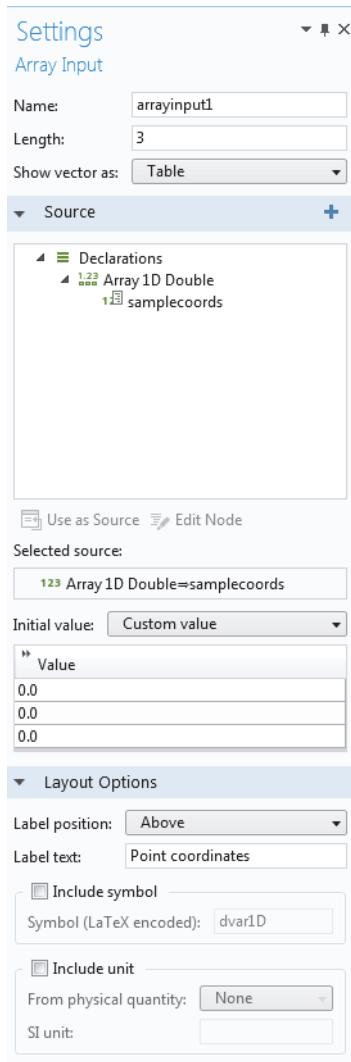
Consider an application where the user enters 3D coordinates for a point where the stress is evaluated. The figure below shows a screenshot from an application with an array input, button, and data display object.

Point coordinates:

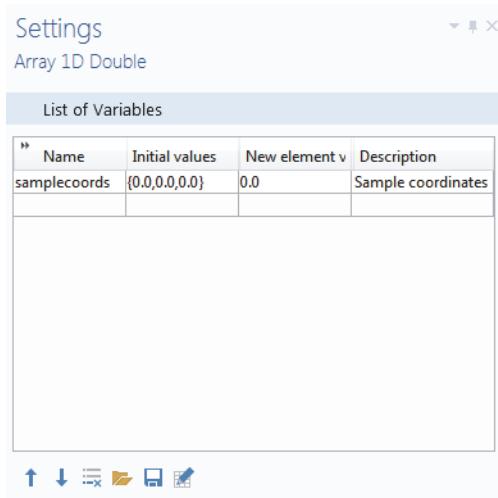
0.0001
-0.001
0.0005

Von Mises stress at point: 40.16 MPa

The figure below shows the **Settings** window of the array input object.



The **Array Input** form object uses a **Source** named `samplecoords`, which is a **ID Array** of type **Double**. This array is created prior to the creation of the **Array Input** object by declaring an **Array ID Double** with the following **Settings**.



In the **Settings** window of the array input object:

- In the **Length** field, enter the length of the array as a positive integer. The default is 3.
- From the **Show vector as** list, choose **Table** (the default) to show the array components as a table, or choose **Components** to show each array component as a separate input field with a label.
- In the **Value** table, enter the initial values for the components in the array.
- The **Layout Options** section provides settings for adding optional labels and units to the array input.

In this example, when the user clicks the button labeled **Evaluate stress at point**, the following method is run:

```
with(model.result().dataset("cpt1"));
  set("pointx", samplecoords[0]);
  set("pointy", samplecoords[1]);
  set("pointz", samplecoords[2]);
endwith();
```

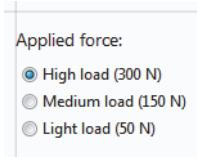
where the values `pointx`, `pointy`, and `pointz` will be used subsequently as coordinates in the evaluation of the stress.

Radio Button

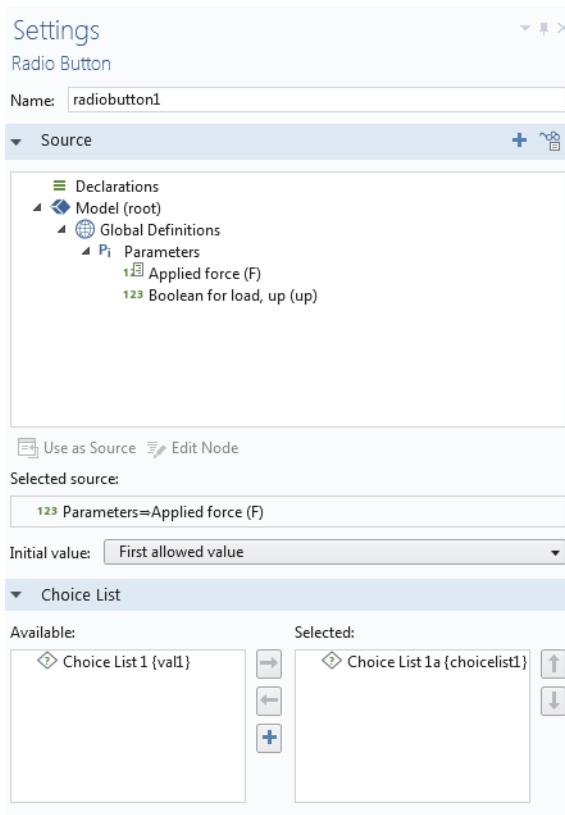
A **Radio Button** object has a fixed number of options from which you can choose one. It is most useful when you have just a handful of options.

USING RADIO BUTTONS TO SELECT A LOAD

Consider an application where the user can select one of three predefined loads, as shown in the following figure.



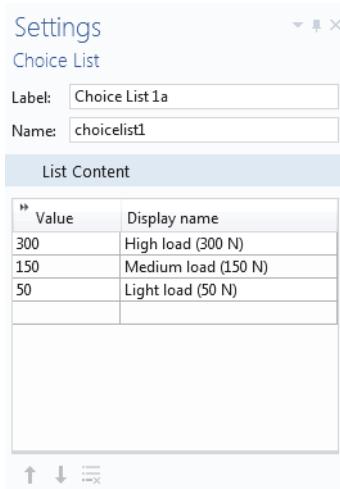
The corresponding **Settings** window is shown below, where the global parameter F is used as the source.



In the **Initial value** list, choose the manner in which the initial selection of the radio button should be made. The options are **From data source**, **First allowed value** (the default), and **Custom value**. For the **Custom value** option, select from a list of the allowed values given by the choice list.

In the **Choice List** section, you can add choice lists that contribute allowed values to the radio button object, where each valid value represents one radio button.

The radio button names are taken from the **Display name** column of their associated choice list. The figure below shows the choice list used in this example.

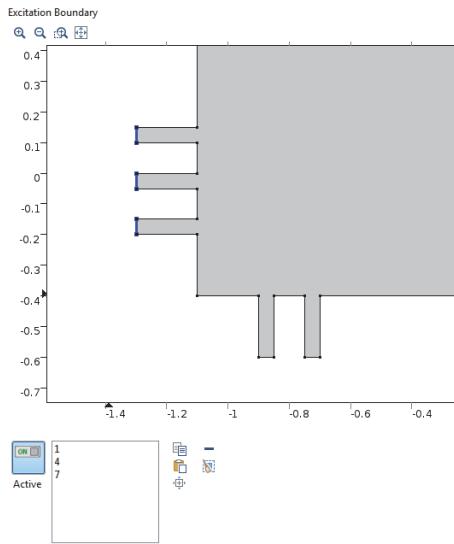


Selection Input

In the Model Builder, named **Selections** let you group domains, boundaries, edges, or points when assigning material properties, boundary conditions, and other model settings. You can create different types of **Selections** by adding subnodes under the **Component > Definitions** node. These can be reused throughout a model component. The **Explicit** selection type lets you group domains, boundaries, edges, or points based on entity number.

In the Application Builder, you can allow the user of an application to interactively change which entities belong to an **Explicit** selection with a **Selection Input** object or a **Graphics** object. In the example below, the embedded model has a boundary condition defined with an **Explicit** selection. Both a **Selection Input** object and a

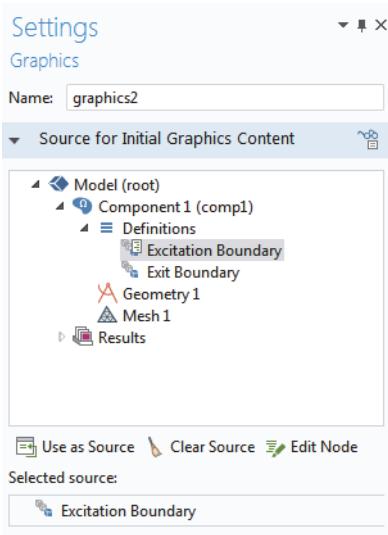
Graphics object are used to let the user select boundaries to be excited by an incoming wave.



The user can here select boundaries by clicking directly in the graphics window, corresponding to the **Graphics** object, or by adding geometric entity numbers in a list of boundary numbers corresponding to a **Selection Input** object.

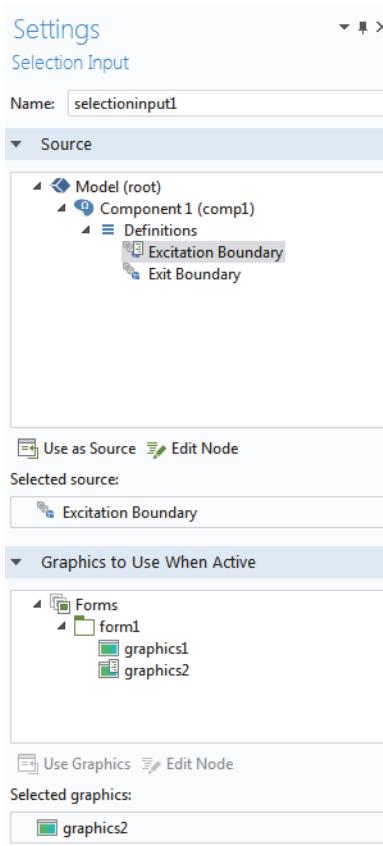
To make it possible to directly select a boundary by clicking on it, you can link a graphics object to an explicit **Selection** used to group boundaries, as shown in the figure below. Select the explicit selection and click **Use as Source**. In the figure below, there are two explicit selections, **Excitation Boundary** and **Exit**

Boundary, and the graphics object `graphics2` is linked to the selection `Excitation Boundary`.



When a graphics object is linked directly to an explicit selection in this way, the graphics object displays the geometry and the user can interact with it by clicking on the boundaries. The boundaries will then be added (or removed) to the corresponding explicit selection.

To make it possible to select by number, you can link a selection input object to an explicit selection, as shown in the figure below.

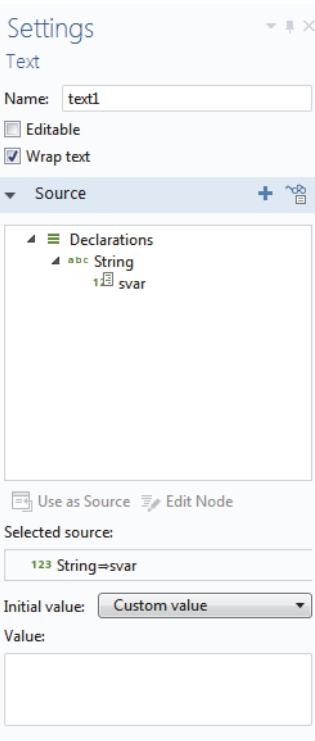


In a selection input object, you can copy, paste, remove, clear, and zoom in to selections.

- ! You can choose to use a graphics object as the source of a selection without having any selection input object. You can also link both a graphics object and a selection input object to the same explicit selection.

Text

A **Text** object is a text field with default text that is taken from a string variable. The **Settings** window for a text object is shown below.



Select a string variable from the tree in the **Source** section and then click **Use as Source**. In the **Value** field, enter the initial text. By default, the **Initial value** text is taken from this field. To instead use the string variable for the **Initial value** text, change the **Initial value** setting to **From data source**.

The check box **Editable** is cleared by default. If selected, the text object can be used, for example, to type comments in a running application. If the text is changed by the user, it is stored in the string variable that is used as the data source regardless of the **Initial value** setting.

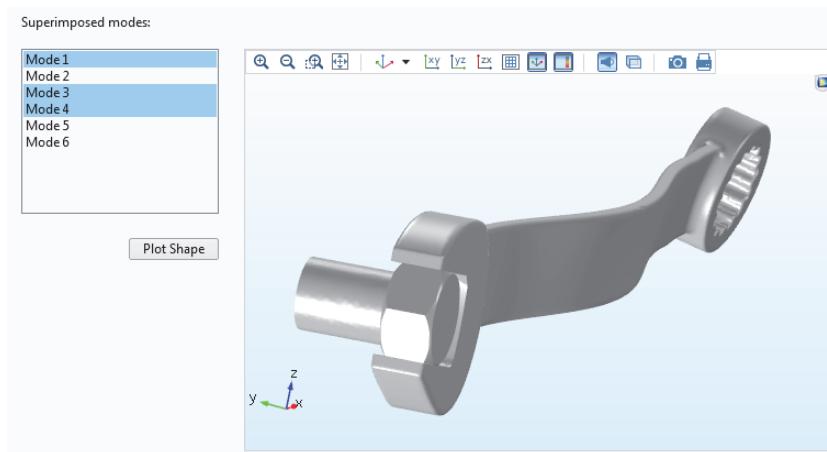
The check box **Wrap text** is selected by default. Clear this check box to disable wrapping of the text; a scroll bar appears if the text does not fit.

List Box

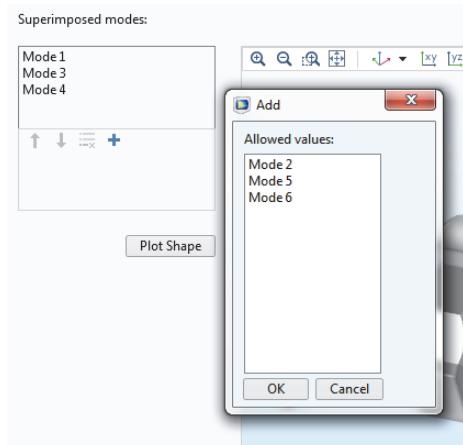
A **List Box** object is similar to a radio button object except that it allows for the simultaneous selection of multiple options.

USING A LIST BOX TO SUPERIMPOSE VIBRATIONAL MODES

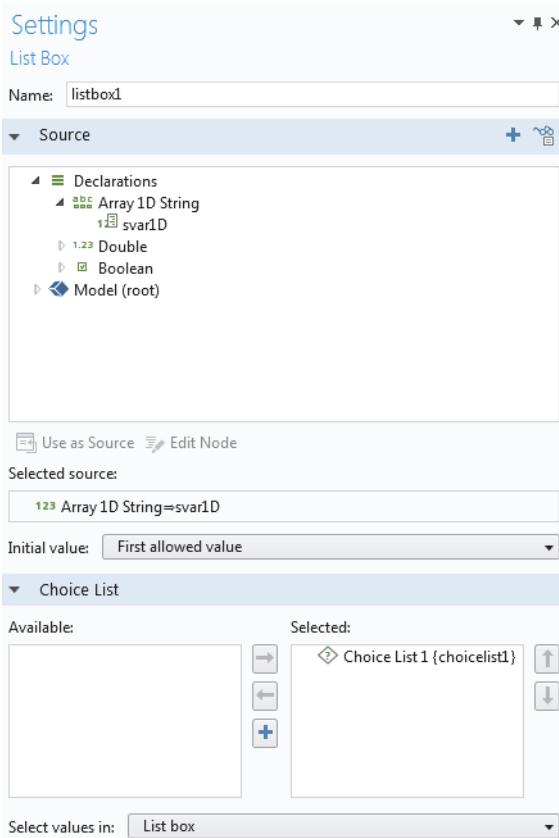
Consider an application where the first six vibrational modes of a mechanical part can be superimposed and visualized by selecting them from a list box, as shown in the figure below.



As an alternative, the following figure shows that a list can be displayed as a dialog box.



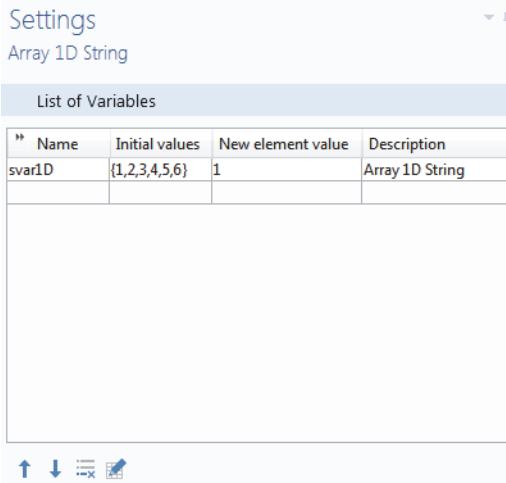
The **Settings** window for the list box of this example is shown in the figure below.



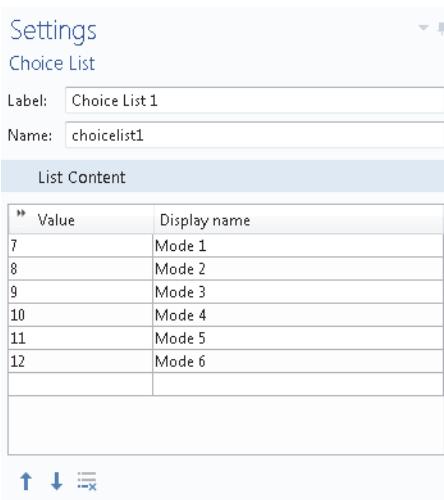
The **Select values in** list allows you to choose between two alternatives, **List box** or **Dialog**, for displaying the list.

You can use any scalar or array declaration as a source. Select from the tree and click **Use as Source**. If you use a string array as the source, you can, in the running application, select more than one item in the list using Shift+click or Ctrl+click.

For other sources, you can only select one value from the list. This example uses a 1D string array `svar1D`. Its **Settings** window is shown below.



In the **Choice List** section, you can add choice lists that contribute allowed values to the list box. The figure below shows the choice list used in this example.



The vibrational modes 1–6 correspond to trivial rigid body modes and are not of interest in this application, hence the **Value** column starts at 7. The choice list allows you to hide the actual mode values in the model from the user by only displaying the strings in the **Display name** column; the first non-rigid body modes are named Mode 1, Mode 2, etc.

The method below uses the COMSOL Multiphysics operator `with()` to visualize the superimposed modes. This example is somewhat simplified since it ignores the effects of amplitude and phase for the modes.

```
String withstru="0";
String withstrv="0";
String withstrw="0";
for(int i=0;i<svar1D.length;i++){
    withstru=withstru + "+" + "with(" + svar1D[i] + ",u)";
    withstrv=withstrv + "+" + "with(" + svar1D[i] + ",v)";
    withstrw=withstrw + "+" + "with(" + svar1D[i] + ",w)";
}

with(model.result("pg7").feature("surf1").feature("def"));
    setIndex("expr", withstru, 0);
    setIndex("expr", withstrv, 1);
    setIndex("expr", withstrw, 2);
endwith();
useGraphics(model.result("pg7"),"/form1/graphics8");
zoomExtents("/form1/graphics8");
```

Assuming the user selected the modes 1, 3, and 5 by using the list box, the method creates an expression `with(1,u)+with(3,u)+with(5,u)`. This expression is then used for the *x*-displacement (dependent variable *u*) in a displacement plot. In a similar way, the method automatically creates expressions for the variables *v* and *w* associated with the *y*- and *z*-displacement, respectively. Note that the command `with()`, used in results as in the example above, is different from the built-in `with()` command used to shorten syntax that is described in “With, Get, and Set Methods” on page 304.

Table

A **Table Object** represents a table with rows and columns that can be used to define input or output. The figure below shows an example of a running application with a table object used to accept input in three columns.

Flow rate and fluid properties:

Flow rate (sccm)	Molecular weight (kg/mol)	Dynamic viscosity (Pa-s)
100	0.032	2E-5
200	0.028	1.78E-5
300	0.146	1.38E-5
1000	0.004	1.9E-5
250	0.032	2E-5
700	0.004	1.9E-5
2000	0.04	2.1E-5
600	0.028	1.78E-5



The figure below shows the corresponding form object and its **Settings** window.

The screenshot displays a software interface with a main workspace and a settings window.

Main Workspace:

- The workspace contains a table object titled "Input".
- The table has columns: "Flow rate (sccm)", "Molecular weight (kg/mol)", and "Dynamic viscosity (Pa-s)".
- Rows include:
 - Pipe length: 2[m]
 - Pipe diameter: 3.5[mm]
 - Temperature: 300[K]
 - Process chamber pressure: 10[Tor]
 - Spacing between pipes: 0.125[m]
- Below the table is a section titled "Flow rate and fluid properties:" containing the same three columns.
- At the bottom of the workspace are standard toolbar icons.

Settings Window:

- Title Bar:** Shows tabs for "Preview", "main", and "input".
- Table Section:**
 - Name: `input_table`
 - Checkboxes: `Show headers`, `Automatically add new rows`, `Sortable`.
 - Sources: Declares `Array1D String` variables `flow_rate`, `molecular_weight`, and `dynamic_viscosity`.
- Properties Section:** Shows table header properties for "Flow rate (sccm)", "Molecular weight (kg/mol)", and "Dynamic viscosity (Pa-s)".
- Toolbar Section:** Shows a toolbar with items: "Save to file", "Move up", "Add", "Clear table", "Delete", "Load from file", "Move down", "Process 1", and "Process 2".

In this example, the data source references three 1D string arrays. You can select any type of array as the source and then click **Use as Source**.

Three check boxes control the overall appearance of the table:

- **Show headers**
- **Automatically add new rows**
- **Sortable**

The **Automatically add new rows** check box ensures that an additional empty row is always available when a user is filling out a table. If all of the 1D string arrays, which are used as a source to the table, have nonempty values for **New element value** in their declaration **Settings** window, then this functionality is deactivated. In this case, new rows can only be added by clicking the **Add** button in the associated table toolbar, if such a button has been made available.

The **Sortable** check box makes it possible to sort the table with respect to a particular column by clicking the corresponding column header.

The **Sources** section contains a table with five columns:

- **Header**
- **Width**
- **Editable**
- **Alignment**
- **Data source**

Each row in this table defines a column in the table object.

The string arrays define the initial values for the rows corresponding to the three columns, as shown in the figure below:

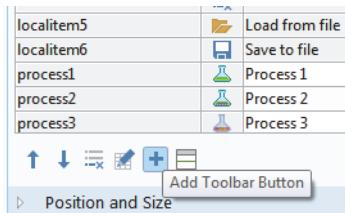
Name	Initial values	New element	Description
flow_rate	{'100','200','300','1000','250','700','2000','600'}	100	Flow rate
molecular_weight	{'0.032','0.028','0.146','0.004','0.032','0.004','0.04','0.028'}	0.032	Molecular weight
dynamic_viscosity	{'2E-5','1.78E-5','1.38E-5','1.9E-5','2E-5','1.9E-5','2.1E-5','1.78E-5'}	1.78E-5	Dynamic viscosity

TOOLBAR

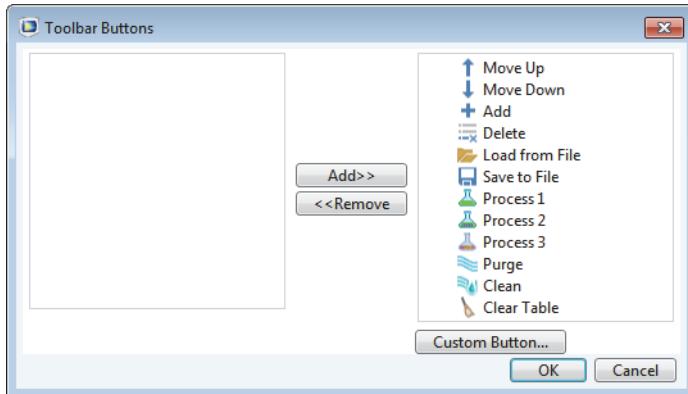
In this section, you can select which toolbar buttons should be used to control the contents of the table. The **Position** list defines the location of the toolbar relative to the table and provides the following options:

- **Below**
- **Above**
- **Left**
- **Right**

To add a button to the toolbar, click the **Add Toolbar Button** below the table:



The following dialog box is then shown:

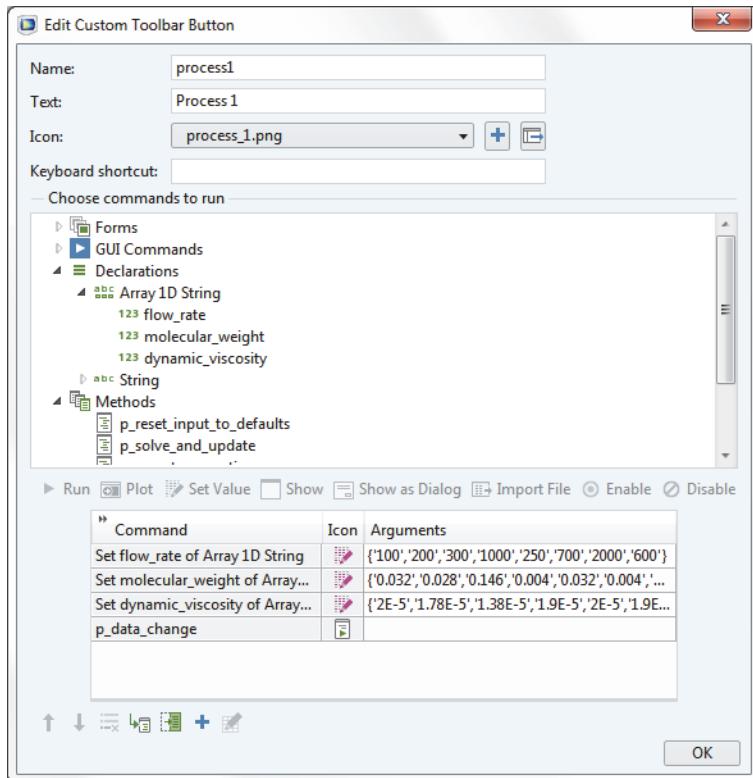


You can add the following buttons:

- **Move Up**
- **Move Down**
- **Add**
- **Delete**
- **Clear Table**
- **Clear Table and Load from File**
- **Load from File**
- **Save to File**

In addition, you can add customized buttons by clicking **Custom Button** in the **Toolbar Buttons** dialog box. The figure below shows the **Edit Custom Toolbar**

Button dialog box used to define a customized button. In this case, the button **Process 1** is used to set default values for a certain process.

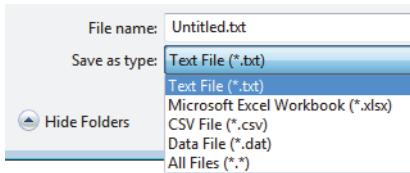


The **Choose commands to run** section is similar to that of menu, ribbon, and toolbar items, as well as buttons.

The **Load from File** and **Save to File** buttons are used to load and save from/to the following file formats:

- Text File (.txt)
- Microsoft® Excel® Workbook (.xlsx)
 - Requires LiveLink™ for Excel®
- CSV File (.csv)
- Data File (.dat)

as shown in the figure below.



Slider

A **Slider** is a form object for choosing numerical input using a slider control.

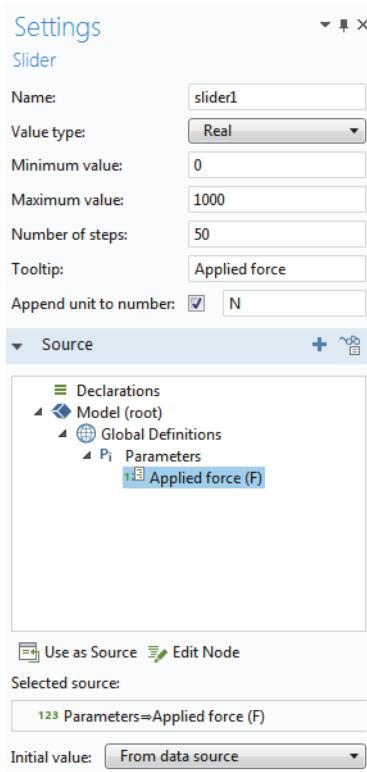
USING A SLIDER TO CHANGE THE MAGNITUDE OF A STRUCTURAL LOAD

Consider an application where the magnitude of a load can be changed by a slider control, such as in the figure below.



In this example, the slider is accompanied by an input field that is used to display the selected value.

The **Settings** window of the slider is shown in the figure below.



In this example, the slider uses a global parameter F as its source. You can select any parameter, variable, or declared scalar variable as a source. Select from the tree and click **Use as Source**.

From the **Value type** list, choose **Integer** or **Real** (default), depending on the type of data in the data source for the slider.

You determine the range of values for the data source by defining the **Maximum value**, **Minimum value**, and **Number of steps** for the slider. You can also set a **Tooltip** that is shown when hovering over the slider. The **Append unit to number** option lets you associate a unit with the slider. This unit is appended to the number using the standard bracket notation, such as [N], before being passed as a value to the source variable. In the example above, the input field and the slider both have the setting **Append unit to number** activated.

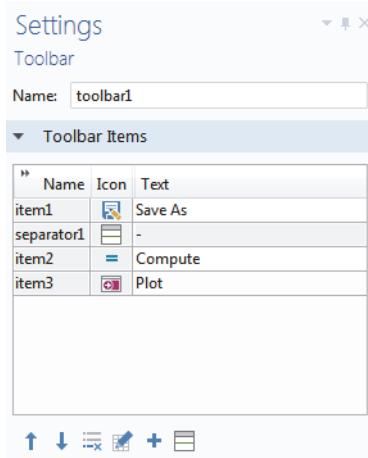
In the **Initial value** list, select **From data source** or **Custom value** for the initial value for the slider.

Toolbar

A **Toolbar** object contains the specifications of a toolbar with toolbar buttons. The figure below shows a toolbar with buttons for **Save as**, **Compute**, and **Plot**.

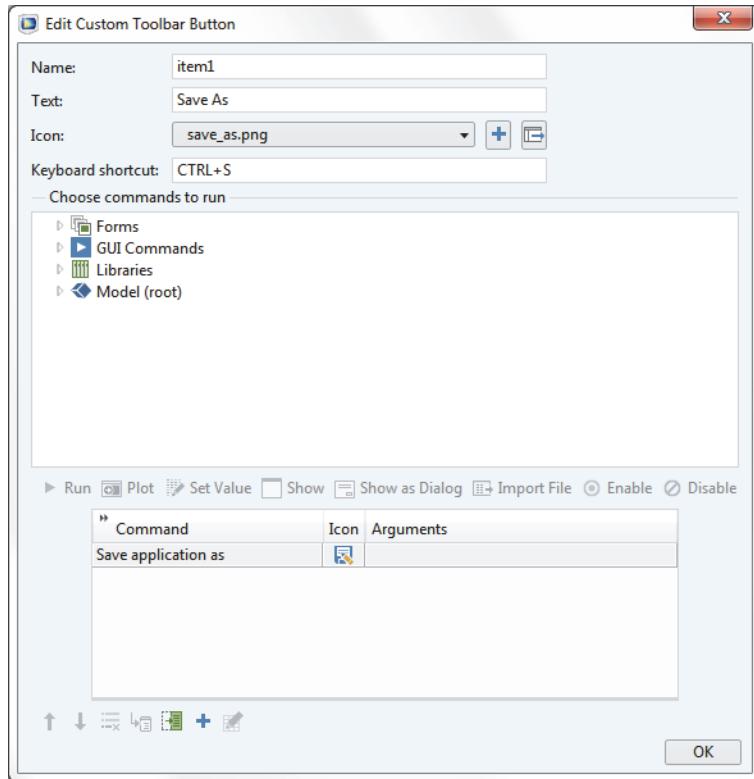


The **Settings** window for this toolbar is shown in the figure below.



Each row in the **Toolbar Items** table contains either an **item**, corresponding to a toolbar button, or a **separator**. Use the buttons below the table to add items or separators, change row order, or to delete a row. Click the **Edit** button to display

the **Settings** window associated with each row. The figure below shows the **Settings** window of **item1**, the **Save As** item.

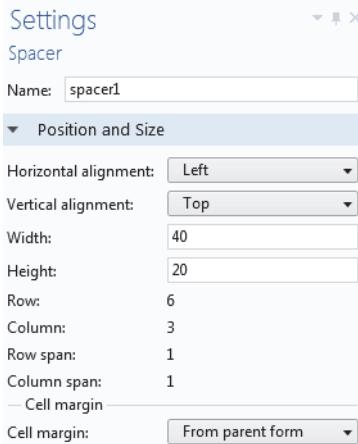


The text in the **Text** field will be shown as a tooltip when hovering over the toolbar button. The **Icon** list, the **Keyboard shortcut** field, and the **Choose commands to run** tree represent the same functionality as for a button object. For more information, see “Button” on page 48.

Spacer

A **Spacer** object is invisible in the user interface and is only used when working in grid layout mode. It defines a space of fixed size that you can use to ensure that neighboring form objects have enough space to show their contents. Typically, you would use a spacer next to a table or graphics object to ensure that they are rendered properly. If the user resizes the window so that it becomes smaller than

the size of the spacer, the effective size of the window is maintained by displaying scroll bars. The figure below shows the **Settings** window of a spacer object.



Appendix B—Copying Between Applications

Many nodes in the application tree can be copied and pasted between applications, including: forms, form objects, menu items, methods, Java® utility methods, external libraries, file declarations, choice list declarations, menus, menu items, ribbon sections, ribbon tabs, and ribbon items.

When you copy and paste forms, form objects, and items between applications, the copied objects may contain references to other objects and items. Such references may or may not be meaningful in the application to which it is copied. The following set of rules apply when objects are pasted from the clipboard:

- A declaration referenced in a form object or menu item is included when copying the object, but is not necessarily pasted. It is only pasted if there is no compatible declaration present. If a compatible declaration exists, that is used instead. A compatible declaration is defined as one having the same name and type. For example, a string declaration is not compatible with an integer declaration. An existing declaration may have an invalid default, but no such check is done when pasting.
- A referenced global parameter may have a different unit, but will still be considered compatible.
- A form or form object directly referenced from another form object is not included automatically when copying objects. The direct reference will point to an existing object if it has the same name. If the original reference is among the copied objects, then that object will be used in the reference instead of any existing objects having the same name. The name of the copied reference will be changed to avoid name collisions.
- No objects in the model tree will be automatically copied, for example, a graphics object referring to a geometry or an input field referring to a low-level setting exposed by Model Data Access. If the reference points to an object that exists in the model tree of the target application, then that reference will be used.
- References to nonexisting objects will be attempted to be removed when pasted. An exception is command sequences in buttons, where all commands are kept and marked as invalid if they point to a nonexisting reference.
- Local methods are included in the copy-paste operation. However, no attempt is made to update the code of the method. This also applies when copying a global method.
- Arguments to commands in the command sequence of a button or a menu item will be left as is.

- All image references are automatically copied and added to the image library when applicable. If there is an existing image with the same name, it will be used instead of the copied version.
- No files, sounds, or methods are automatically copied if referenced to. However, methods can be copied and pasted manually.
- All pasted objects that have a name that conflicts with that of an existing object will be renamed. Any references to the renamed object from other pasted objects will be updated.

Appendix C—File Handling and File Scheme Syntax

The handling of files may be an important feature of an application. For example, the application may require a spreadsheet file with experimental data as input, a CAD file to be imported, or a report to be generated and exported. The Application Builder provides tools for reading and writing entire files or portions of a file. The way that this is done will vary depending on the system where the application is running. The file system may be different on the computer running COMSOL Multiphysics, where the application is developed, and on the computer where COMSOL Server is installed and the application will be running once it is deployed.

File Handling with COMSOL Server

In general, you cannot read and write files to local directories when running applications with a web browser or the COMSOL Client for Windows®. The application and its methods are run on the server and have no knowledge of the client file system (where the web browser or COMSOL Client is run).

However, there are techniques for transferring files to and from the client file system when running an application both with a web browser and the COMSOL Client.

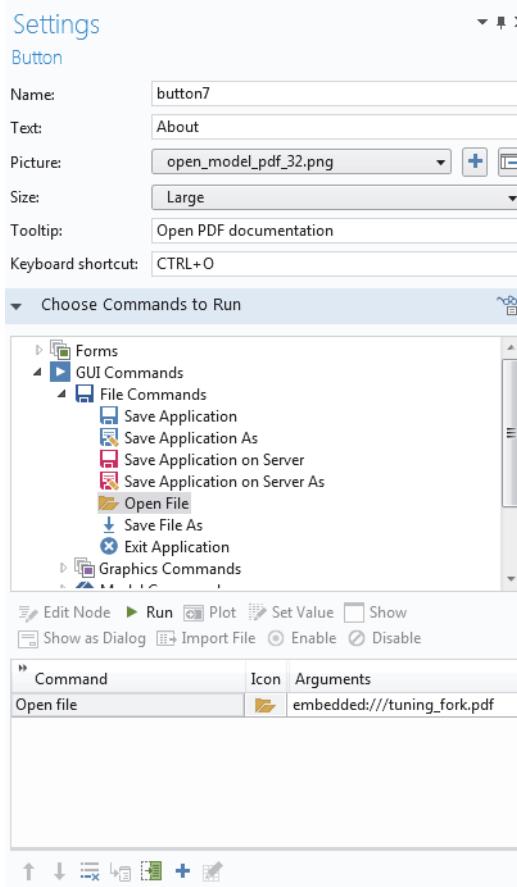
A **File Import** object can be used to ask the user for a file. The user then browses to a file on the client file system, which is then uploaded to the COMSOL Server file system and becomes available to the application and its methods. This can be used, for example, to provide a CAD file or an experimental data file from the user at run time. This is covered in the section “File Import” on page 246.

In a command sequence of, for example, a button, you can export data generated by the embedded model by running a subnode of the **Export** or **Report** nodes. This is covered in the section “File Export” on page 254.

FILE COMMANDS

In the editor tree used in a command sequence, the **File Commands** folder contains commands to save and load applications and files, as well as exiting an application. The command **Open File** will pick any file from the server produced by a method, the model, or embedded with the application, and open it using the associated application on the client. This can be used, for example, to open a PDF file in the client file system, or show a text file or an image exported from the model on the

client side. In the figure below, an **Open File** command is used to open the PDF documentation for an application.



To open files from a method, use the built-in method `fileOpen`; see also “System Methods” on page 283.

To save a file, use the command **Save File As**, which is similar to **Open File**. It will take any file from the server file system and display a **Save As** dialog box to the user where the user can browse to a client location to save the file. This is similar to downloading files from a link within a web browser.

To save files from a method, use the built-in method `fileSaveAs`; see also “Built-In GUI Methods” on page 302.

The **Save Application** and **Save Application As** commands are available for use in the command sequence for certain form objects. The **Save Application As** command

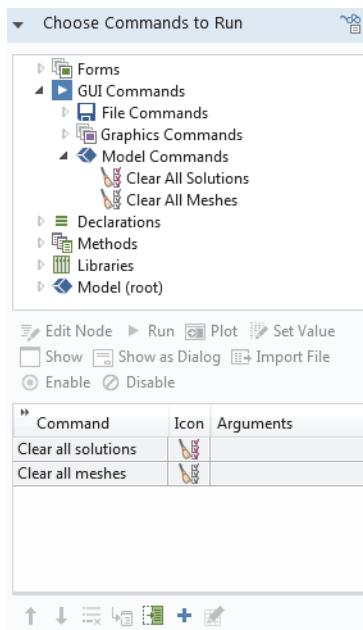
will display a **Save As** dialog box where the user can specify a client path where the entire application will be saved.

Similarly, the **Save Application on Server** and **Save Application on Server As** commands are available to save the entire application on the server file system. For information on the corresponding built-in methods, see “Built-In GUI Methods” on page 302.

In summary, both uploading and downloading of files from the client file system is supported, but the application can never do it silently in the background without the user browsing to the source or destination location of the file.

MODEL COMMANDS

In the editor tree used in a command sequence, the **Model Commands** folder contains two commands: **Clear all solutions** and **Clear all meshes**. Use these to make the MPH file size smaller before saving an application by erasing solution and mesh data, respectively.



File Scheme Syntax

To make applications portable, the Application Builder allows you to use virtual file locations using file schemes. A file scheme can be seen as a pointer to a file on the file system but the application does not need to know where the file is actually stored.

Different file schemes exist for different purposes. The `user` file scheme is for files that should be persistent between different runs of an application by the same user. The `common` file scheme behaves in the same way but is for files that should be shared between all users. The `temp` file scheme is for files that should be removed as soon as the application is closed. The `embedded` file scheme is used to store files in the application itself. This can be useful if you want to make the application self-contained and send it to someone else. Finally, the `upload` file scheme is for files that are uploaded to the application by the user at runtime, such as a CAD-file to which the user browses.

The table below summarizes all available file schemes.

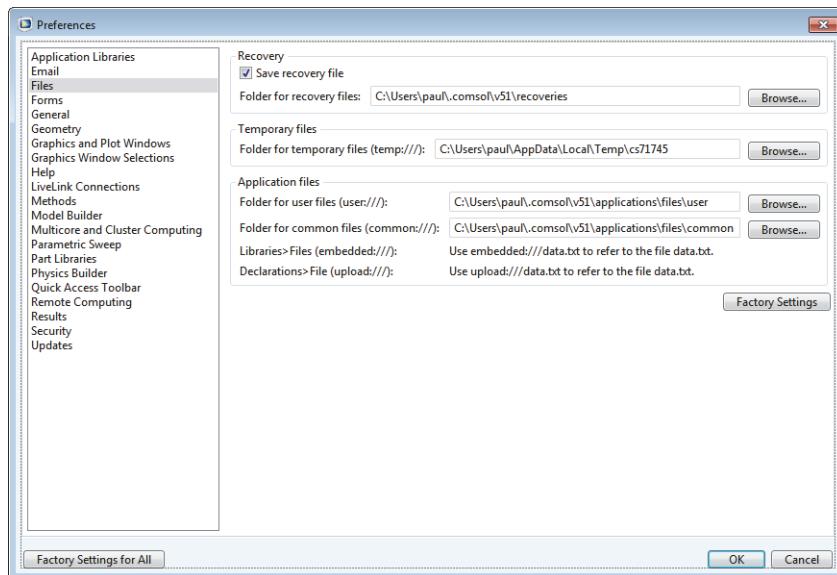
SCHEME	REFERS TO	DEFAULT PATH	TYPICAL USAGE
<code>embedded:///</code>	Files embedded in the application using Libraries>Files	N/A	Experimental data, CAD files, mesh files, interpolation data
<code>upload:///</code>	Files to be uploaded by the user at run time	Determined by the Target directory in the Settings window of the File declaration	Experimental data, CAD files, mesh files, interpolation data
<code>temp:///</code>	Files in a random temporary directory, which is unique for each started application instance. These files are deleted when the application is closed.	A random subdirectory to the folder for temporary files, as determined by the settings in Preferences>Files	Temporary files produced by command sequences or methods, or output to be saved on the client (for use with COMSOL Server)
<code>user:///</code>	Files in a directory shared by all applications for the current user	Determined by the settings in Preferences>Files	Output from methods to be saved between sessions
<code>common:///</code>	Files in a directory shared by all users	Determined by the settings in Preferences>Files	Files shared between many users or applications

For more information on files in the Libraries node accessible by the `embedded:///` syntax, see “Libraries” on page 159.

The table below summarizes the usage of the different file schemes. In the table, a check mark means that this scheme is available and (r) means that it is the recommended scheme.

USAGE	EMBEDDED	UPLOAD	TEMP	USER	COMMON
File is used as input	✓ (r)	✓			✓
File is output			✓ (r)	✓	
Method reading a file	✓ (r)	✓	✓	✓	✓
Method writing a file			✓ (r)	✓	
File is client-side	✓	✓	✓ (r)	✓	✓

You can set the preferences for the paths to temporary, user, and common files in the **Files** page of the **Preferences** dialog box, which is accessible from the **File** menu, as shown in the figure below.



File Import

CAD IMPORT USING THE MODEL TREE AND A FILE IMPORT OBJECT

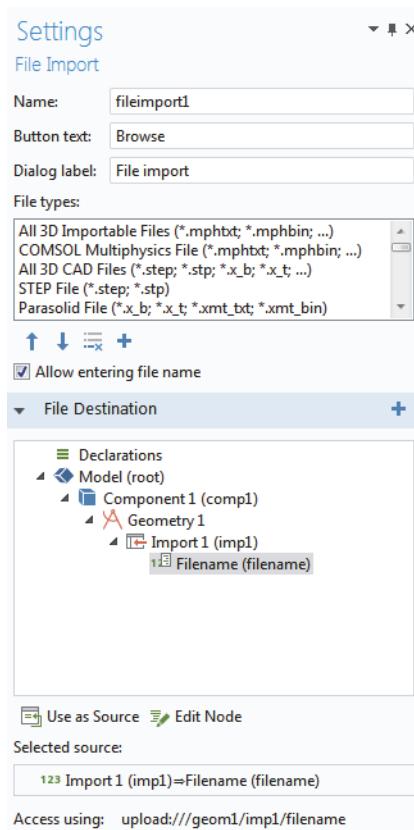
A **File Import** object is used to display a file browser with an associated input field for browsing to a file or entering its path and name. It is used to enable file import by the user of an application at run time, when the file is not available in the application beforehand. You can directly link a **File Import** object to a file **Import** node in the model tree, for example, a CAD **Import** node. Consider an application where a CAD file can be selected and imported at run time, as shown by the figure below.



The corresponding **File Import** object is shown in the figure below.

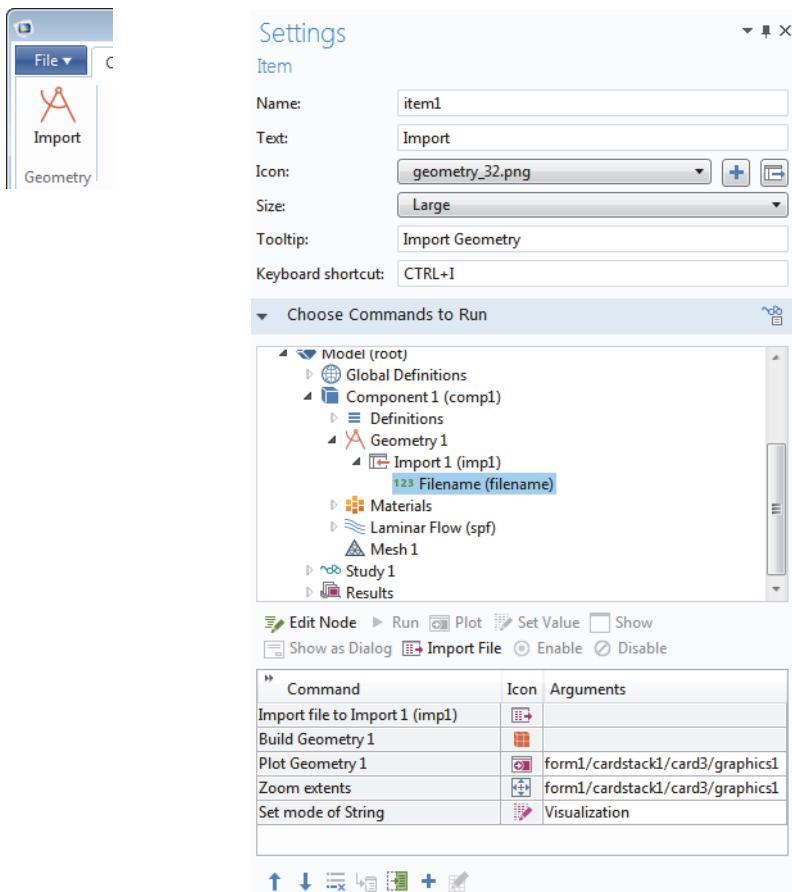


The **Settings** window for the **File Import** object has a section **File Destination**. In this section, you can select any tree node that allows a file name to be input. This is shown in the figure below, where the **Filename** for the **Import** node is selected.

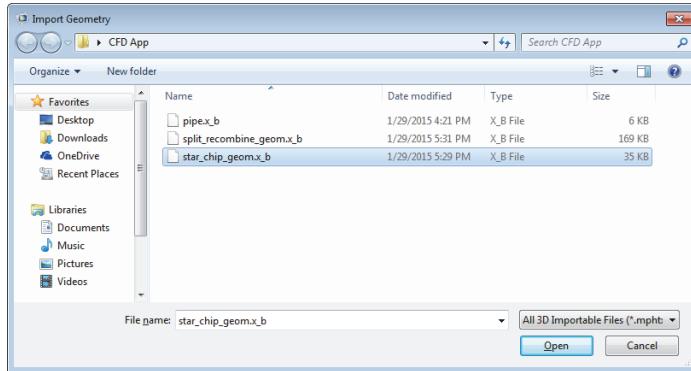


If you don't wish to use a file browser, you can directly reference a file name from a button, or an item in a menu, ribbon, or toolbar.

The figure below shows a ribbon item used for geometry import together with its **Settings** window.



In the **Settings** window above, the command **Import file to Import** I will open a file browser for the user to select a file, as shown in the figure below.



The subsequent commands build and plot the geometry, zoom out using zoom extents, and finally set the value of a string variable (in this case used to control a card stack).

For more information on the **File Import** object, see “File Import” on page 206.

FILE IMPORT IN METHODS

Continuing the example of the previous section, assume that we, in the **Settings** window, now click **Convert to New Method**. The corresponding lines of code show how CAD import can be accomplished from a method:

```
importFile(model.geom("geom1").feature("imp1"), "filename");
model.geom("geom1").run();
useGraphics(model.geom("geom1"), "form1/cardstack1/card3/graphics1");
zoomExtents("form1/cardstack1/card3/graphics1");
mode = "Visualization";
```

The first line illustrates using the built-in method `importFile`. For more information on the method `importFile` and other methods for file handling, see “File Methods” on page 279.

FILE ACCESS AND FILE DECLARATIONS

At the bottom of the **Settings** window of a **File Import** object, you can see which file scheme syntax to use to access an imported file from a method (next to **Access**

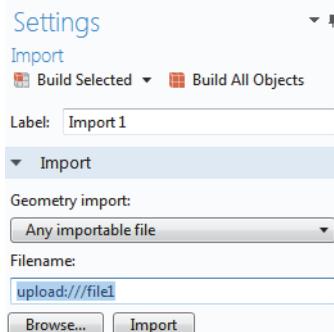
using:). The figure below shows an example where a **File Destination** and **Filename** are used.



The file scheme syntax is here `upload:///geom1/imp1/filename` and needs to be used whenever accessing this file.

As an alternative, you can use a **File** declaration under the **Declarations** node. (However, **File** declarations are primarily used for file import from method code.) In this case, the file chosen by the user can be referenced in a form object or method using the syntax `upload:///file1`, `upload:///file2`, etc. The file name handle (`file1`, `file2`, etc.) can then be used to reference an actual file name picked by the user at run time. See also “File Declarations” on page 126.

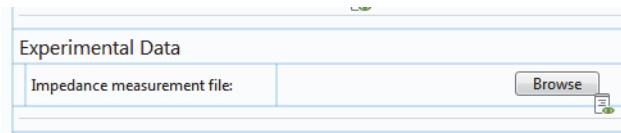
This syntax can also be used in any file browser text fields within the Model Builder nodes. The figure below shows a file reference used in the **Filename** field of the **Import** model tree node for a model using geometry import.



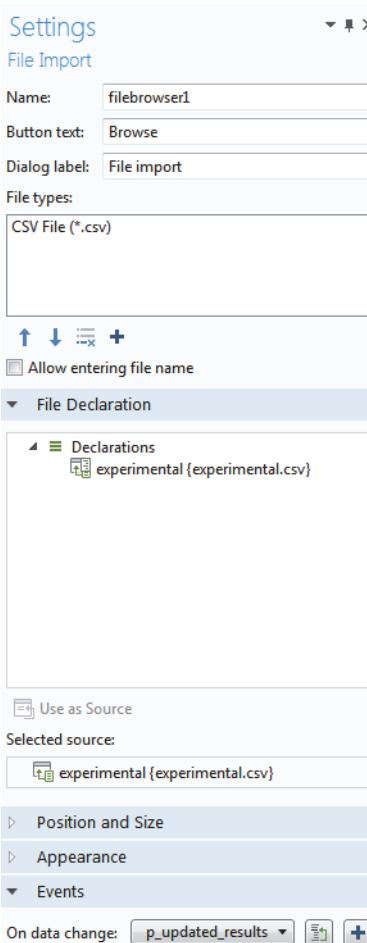
However, a quicker way is to link a file import object directly to the **Filename** field, as described previously in the section “CAD Import using the Model Tree and a File Import Object” on page 246.

IMPORTING EXPERIMENTAL DATA

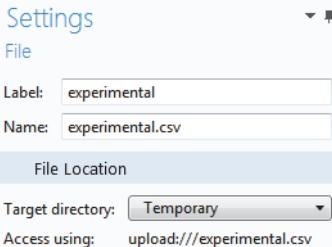
Consider an application where the user is providing a file with experimental data at run time. The figure below shows the file import object of such an application as it appears in grid layout mode.



The figure below shows the **Settings** window of the corresponding file import object and its link to a file declaration.



In this application, the **File types** table specifies that only CSV files are allowed. The **Settings** window for the **File** declaration is shown in the figure below.

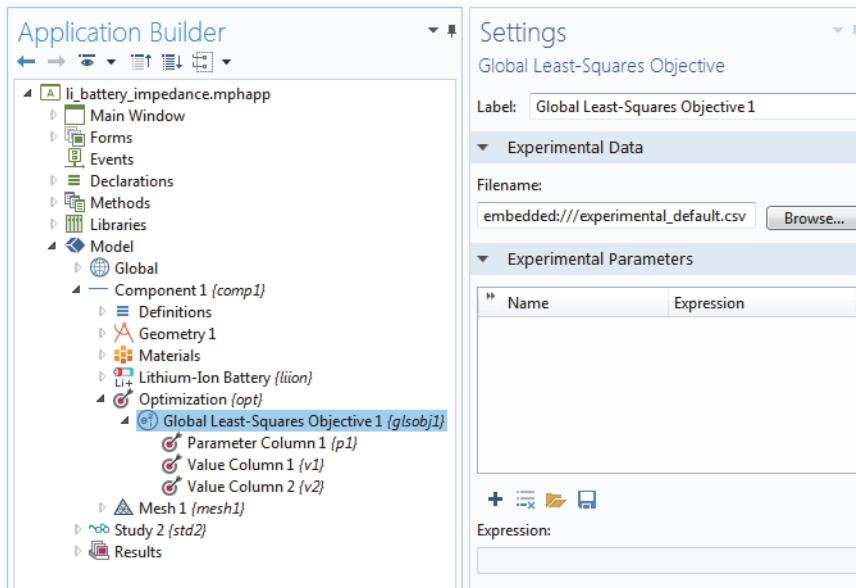


The file declaration serves as the “destination” of the imported data which is written to the file `upload:///experimental.csv`.

Note that the file extension `.csv` used in the declaration is optional and that the file picked by the user at run time can have any name. For example, the file name picked at run time can be `my_data.csv`, but when referenced in method code, the abstract file handle name `experimental.csv` is always used.

In order to make it possible to run the application without having to first provide experimental data, a file containing default experimental data is embedded in the application. This default data file is used by the application by accessing it with the `embedded:///` file scheme syntax, as shown in the figure below. In this example,

which uses the Optimization Module, the application performs a least-squares fit to the experimental data.



The following method handles the logic to determine if user-provided experimental data files exist or if the default data set should be used.

```

if (exists("upload:///experimental.csv")) {
    with(model.physics("opt").feature("glsobj1"));
        set("fileName", "upload:///experimental.csv");
    endwith();
}
else{
    String s_data = confirm("No experimental data file was uploaded. Do you
want to use the embedded data?", "Experimental Data", "Yes", "Cancel
Parameter Estimation");
    if(s_data.equals("Cancel Parameter Estimation")){
        return;
    }
}

```

In the case a user-provided file exists, the code replaces

`embedded:///experimental_default.csv` with `upload:///experimental.csv` in the physics interface `glsobj1`.

File Export

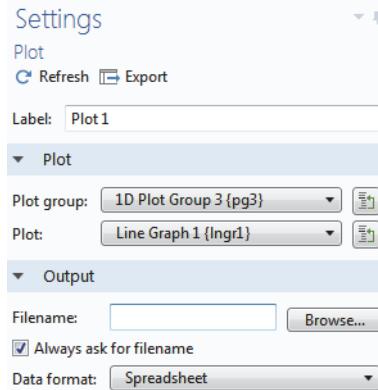
FILE EXPORT USING THE MODEL TREE

In a command sequence of, for example, a button, you can export data generated by the embedded model by running a subnode of the **Export** or **Report** nodes.

In the model tree, the **Export** node may contain the following types of subnodes for file export:

- **Data**
- **Plot**
- **Mesh**
- **Table**
- **3D Image**
- **2D Image**
- **ID Image**
- **Animation**

The **Settings** window for each of these nodes contains an **Output** section with a field for **Filename**. The figure below shows the **Settings** window for an **Export>Plot** node.



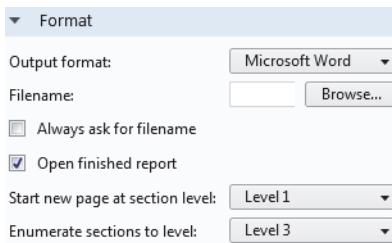
You can leave the **Filename** field blank, as shown in the figure above. In the command sequence of, for example, a button, you can run the corresponding

Export>Plot node and, at run time, it will open a file browser window for the user to select a location and file name, as seen in the figure below.



While developing an application, you may need to use the Model Builder repeatedly to check the exported data. In this case, you can use the **Filename** field for a test file and, by selecting the **Always ask for filename** check box, a file browser will still be opened at run time.

In a similar way to the **Export** subnodes, each **Report** subnode has a section **Format** with a **Filename** field, as seen in the figure below.



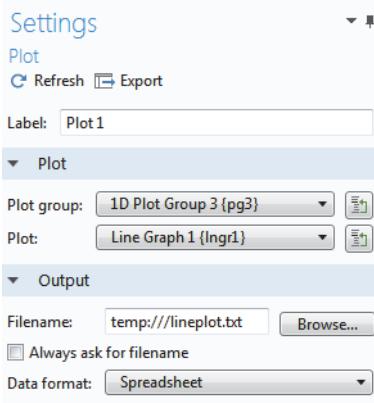
By running a **Report** subnode, a file browser window is opened for the user to select a location and file name for the report.

For more detailed control over file import and export, you can instead use a file scheme.

FILE EXPORT TO A TEMPORARY FILE

Some applications may need to produce temporary files, and it is accomplished by using the `temp:///` file scheme. The temporary files are stored in a random temporary directory, which is unique for each started application instance. These files are deleted when the application is closed. Temporary files can be produced by command sequences or methods, or output to be saved on the client when used with COMSOL Server.

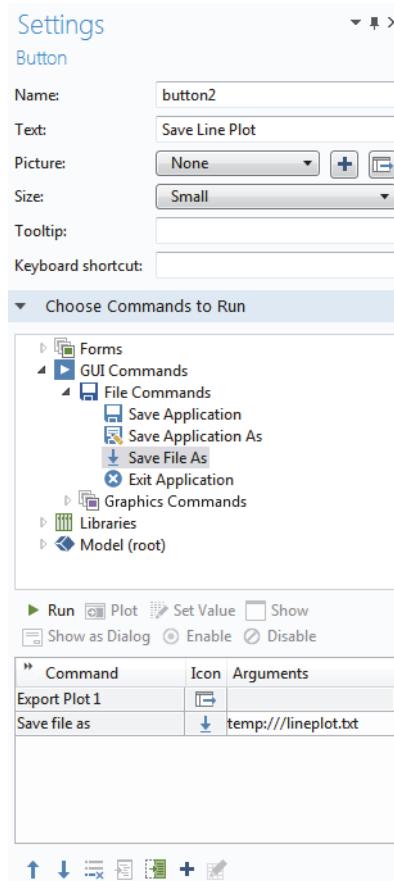
The example below shows the **Settings** window of an **Export>Plot** node that is used to export plot data as numerical values.



The **Filename** in its **Output** section is set to `temp:///lineplot.txt`.

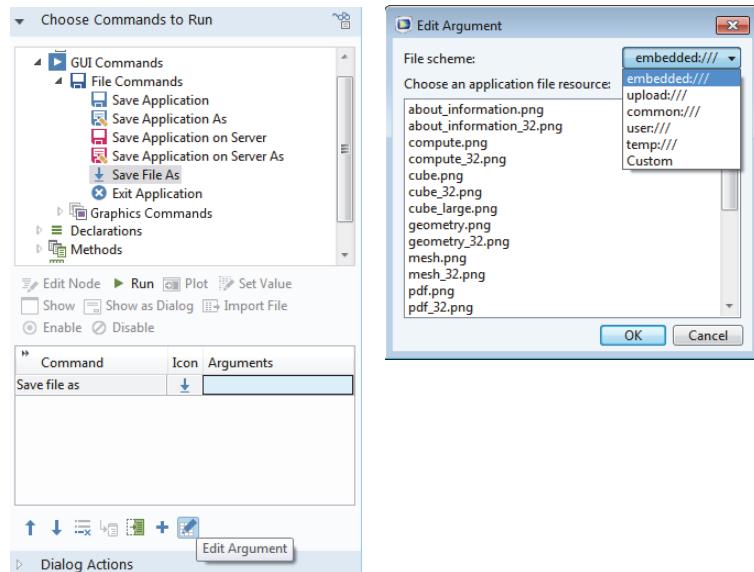
To save the plot to disc in this example, a button was created. In the **Settings** window for the button, in the section **Choose Commands to Run**, first create the output graph file by choosing the **Export>Plot** node created above and clicking **Run**. Second, choose **GUI Commands>File Commands>Save File As** and click **Run** again. In the **Output** section of the button **Settings**, set the filename to the name

of the temporary file created by the **Export Plot** command, in this case `temp:///lineplot.txt`.





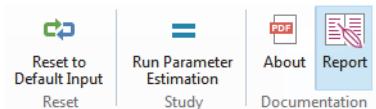
The **Save File As** command provides a dedicated **Edit Argument** dialog box with easy access to all embedded files as well as shortcuts for all file schemes.



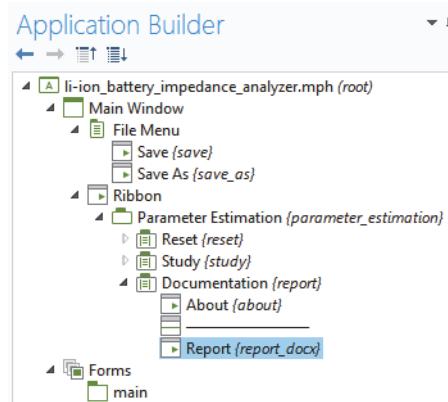
CREATING REPORTS USING LOW-LEVEL FUNCTIONALITY

This section describes creating reports using low-level functionality. For a more direct method, see “File Export” on page 254.

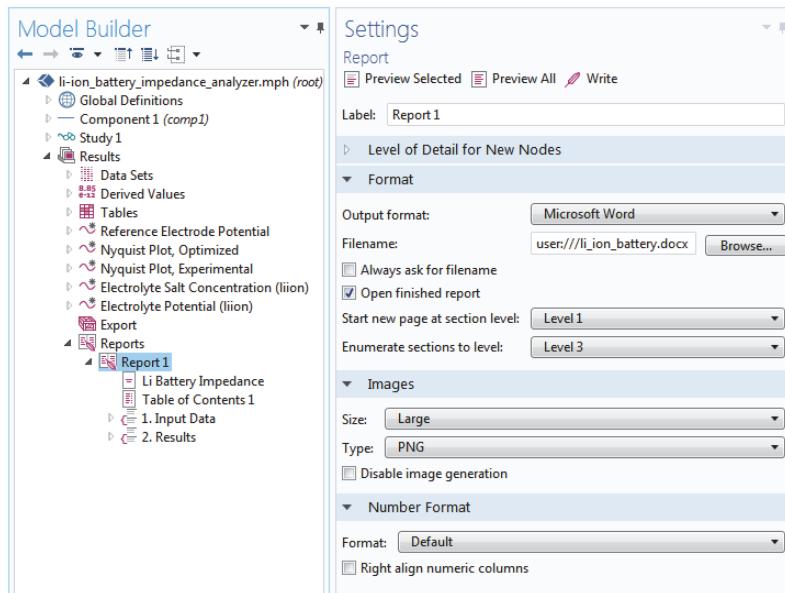
The example below shows an application where a report in the Microsoft® Word® format (.docx) can be saved by the user. The figure below shows a tab in the ribbon of the application. In this tab, there is a **Report** button in the **Documentation** section.



The associated application tree node is shown in the figure below.



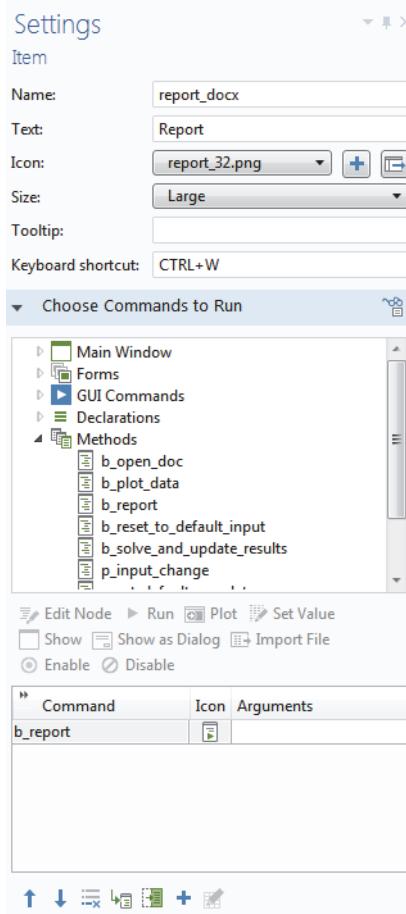
The following figure shows how the syntax `user://file` was used in the **Filename** field in the **Settings** window of the **Report** node of the Model Builder.



In this application, the check box **Open finished report** is selected, which means that the Word® document will open after the report has been created. The user of the application can then save the report from the Word® file menu.

In this example, the file scheme `common:///` could have been used in the same way. The **user** and **common** file schemes are primarily useful when the same files are used repeatedly by an application.

The figure below shows the **Settings** window of the **Report** ribbon item.



The method `b_report` contains the following code:

```
if(length(information_card)>0) {  
    alert("New input data. Compute to update results first.");  
}  
else {  
    model.result().report("rpt1").run();  
}
```

The file scheme syntax can also be used directly in methods. The code below is from a method used to export an HTML report.

```
String answerh = request("Enter file name", "File Name", "Untitled.html");
if(answerh != null){
    model.result().report("rpt1").set("format", "html");
    model.result().report("rpt1").set("filename", "user://"+answerh);
    model.result().report("rpt1").run();
}
```

Appendix D—Keyboard Shortcuts

The table below lists the keyboard shortcuts available in the Application Builder.

SHORTCUT	ACTION	APPLICATION BUILDER	FORM EDITOR	METHOD EDITOR
Ctrl+A	Select all	✓	✓	✓
Ctrl+D	Deselect all		✓	
Ctrl+C	Copy	✓	✓	✓
Ctrl+V	Paste		✓	✓
Del	Delete	✓	✓	✓
Ctrl+N	Create a new application	✓	✓	✓
Ctrl+S	Save an application	✓	✓	✓
Ctrl+F8	Test an application	✓	✓	✓
Alt+click	Edit certain form objects		✓	
Ctrl+Pause	Stop a method	✓		
Ctrl+Shift+F8	Apply changes	✓	✓	✓
Ctrl+R	Record code			✓
F11	Go to node			✓
F12	Extract variable			✓
F1	Display help	✓	✓	✓
F2	Rename applicable nodes	✓		
F3	Disable applicable nodes	✓		
F4	Enable applicable nodes	✓		
Ctrl+up arrow	Move applicable nodes up	✓		
Ctrl+down arrow	Move applicable nodes down	✓		
Ctrl+Z	Undo	✓	✓	✓
Ctrl+Y	Redo (Control+Shift+Z on Mac)	✓	✓	✓
F5	Continue (in debugger)			✓
F6	Step (in debugger)			✓
F7	Step into (in debugger)			✓
F8	Check syntax			✓

SHORTCUT	ACTION	APPLICATION BUILDER	FORM EDITOR	METHOD EDITOR
Ctrl+F	Find and replace text in methods			✓
Ctrl+Space	Autocomplete method code			✓
Ctrl+U	Make selected code lowercase			✓
Ctrl+Shift+U	Make selected code uppercase			✓
Ctrl+B	Toggle breakpoint on selected line			✓
Ctrl+scroll wheel up	Zoom in, in method code window			✓
Ctrl+scroll wheel down	Zoom out, in method code window			✓
Ctrl+all arrow keys	Fine-tune position of selected form objects		✓	
All arrow keys	Fine-tune position of selected form objects		✓	
Ctrl+Alt+A	Go to Application Builder window		✓	✓
Ctrl+Alt+M	Go to Model Builder	✓	✓	
Ctrl+Alt+left-click	Create a local method		✓	
Alt+F4	Close window	✓	✓	✓
Ctrl+F4	Close document		✓	✓
Ctrl+Shift+F4	Close all documents		✓	✓

Appendix E—Built-in Method Library

This appendix lists all of the built-in methods available in the Method editor, except for methods that operate on the model object. For more information on the model object and its methods, see the *COMSOL Programming Reference Manual*.

The syntax rules are those of the Java[®] programming language. Note that each line of code needs to end with a semicolon (;), but the semicolon is omitted in the listings below.

Conversion Methods

NAME	SYNTAX	DESCRIPTION
toBoolean	<code>boolean toBoolean(String str)</code>	Converts the given string to a Boolean. ('true' returns true, all other strings return false).
toBoolean	<code>boolean [] toBoolean(String[]... strs)</code>	Converts all the strings in the given array to Booleans ('true' returns true, all other strings return false) and returns a Boolean array.
toBoolean	<code>boolean [][] toBoolean(String[][]... strs)</code>	Converts all the strings in the given matrix to Booleans ('true' returns true, all other strings return false) and returns a Boolean matrix.
toDouble	<code>double toDouble(String str)</code>	Converts the given string to a double.
toDouble	<code>double[] toDouble(String... strs)</code>	Converts all the strings in the given array to doubles and returns a double array.
toDouble	<code>double[][] toDouble(String[]... strs)</code>	Converts all the strings in the given matrix to doubles and returns a double matrix.
toInt	<code>int toInt(String str)</code>	Converts the given string to an integer.
toInt	<code>int[] toInt(String... strs)</code>	Converts all the strings in the given array to integers and returns an integer array.
toInt	<code>int[][] toInt(String[]... strs)</code>	Converts all the strings in the given matrix to integers and returns an integer matrix.
toString	<code>String toString(int value)</code>	Converts the given integer to a string.
toString	<code>String toString(double value)</code>	Converts the given double to a string.
toString	<code>String toString(boolean value)</code>	Converts the given Boolean to a string.
toString	<code>String toString(double value, int decimals)</code>	Converts the given double to a string with the given number of decimals.

NAME	SYNTAX	DESCRIPTION
toString	<code>String toString(double value, String format)</code>	Converts the given double to a string using the given format specifier; which is the same as <code>java.util.Formatter</code> . See the corresponding Java format string documentation for more information.
toString	<code>String[] toString(double[] darray)</code>	Converts all the doubles in the given array to strings and returns a string array.
toString	<code>String[][] toString(double[][] dmatrix)</code>	Converts all the doubles in the given matrix to strings and returns a string matrix.
toString	<code>String[] toString(int[] iarray)</code>	Converts all the integers in the given array to strings and returns a string array.
toString	<code>String[][] toString(int[][] imatrix)</code>	Converts all the integers in the given matrix to strings and returns a string matrix.
toString	<code>String[] toString(boolean[] barray)</code>	Converts all the Booleans in the given array to strings and returns a string array.
toString	<code>String[][] toString(boolean[][] bmatrix)</code>	Converts all the Booleans in the given matrix to strings and returns a string matrix.

Array Methods

NAME	SYNTAX	DESCRIPTION
getColumn	<code>String[] getColumn(String[][] matrix, int column)</code>	Returns a String[] for a specified column in the matrix. Useful when values have been read from a file and only certain columns should be shown in a table.
getColumn	<code>double[] getColumn(double[][] matrix, int column)</code>	Returns a double[] for a specified column in the matrix.
getColumn	<code>int[] getColumn(int[][] matrix, int column)</code>	Returns an int[] for a specified column in the matrix.
getColumn	<code>boolean[] getColumn(boolean[][] matrix, int column)</code>	Returns a boolean[] for a specified column in the matrix.
getSubMatrix	<code>String[][] getSubMatrix(String[][] matrix, int startCol, int endCol, int startRow, int endRow)</code>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
getSubMatrix	<code>double[][] getSubMatrix(double[][] matrix, int startCol, int endCol, int startRow, int endRow)</code>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
getSubMatrix	<code>int[][] getSubMatrix(int[][] matrix, int startCol, int endCol, int startRow, int endRow)</code>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
getSubMatrix	<code>boolean[][] getSubMatrix(boolean[][] matrix, int startCol, int endCol, int startRow, int endRow)</code>	Returns a rectangular submatrix of the input matrix spanning columns from startCol to endCol, and rows from startRow to endRow.
insert	<code>String[] insert(String[] array, String value, int index)</code>	Inserts an element at position index in an array and returns the expanded array.
insert	<code>double[] insert(double[] array, double value, int index)</code>	Inserts an element at position index in an array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
insert	<code>int[] insert(int[] array, int value, int index)</code>	Inserts an element at position index in an array and returns the expanded array.
insert	<code>boolean[] insert(boolean[] array, boolean value, int index)</code>	Inserts an element at position index in an array and returns the expanded array.
insert	<code>String[] insert(String[] array, String[] value, int[] index)</code>	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	<code>double[] insert(double[] array, double[] value, int[] index)</code>	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	<code>int[] insert(int[] array, int[] value, int[] index)</code>	Inserts elements in an array at positions given by the index array and returns the expanded array.
insert	<code>boolean[] insert(boolean[] array, boolean[] value, int[] index)</code>	Inserts elements in an array at positions given by the index array and returns the expanded array.
append	<code>String[] append(String[] array, String value)</code>	Adds an element to the end of an array and returns the expanded array.
append	<code>double[] append(double[] array, double value)</code>	Adds an element to the end of an array and returns the expanded array.
append	<code>int[] append(int[] array, int value)</code>	Adds an element to the end of an array and returns the expanded array.
append	<code>boolean[] append(boolean[] array, boolean value)</code>	Adds an element to the end of an array and returns the expanded array.
append	<code>String[] append(String[] array, String[] value)</code>	Adds elements to the end of an array and returns the expanded array.
append	<code>double[] append(double[] array, double[] value)</code>	Adds elements to the end of an array and returns the expanded array.
append	<code>int[] append(int[] array, int[] value)</code>	Adds elements to the end of an array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
append	<code>boolean[] append(boolean[] array, boolean[] value)</code>	Adds elements to the end of an array and returns the expanded array.
remove	<code>String[] remove(String[] array, int index)</code>	Removes an element from an array and returns the shortened array.
remove	<code>double[] remove(double[] array, int index)</code>	Removes an element from an array and returns the shortened array.
remove	<code>int[] remove(int[] array, int index)</code>	Removes an element from an array and returns the shortened array.
remove	<code>boolean[] remove(boolean[] array, int index)</code>	Removes an element from an array and returns the shortened array.
remove	<code>String[] remove(String[] array, int[] index)</code>	Removes elements from an array and returns the shortened array.
remove	<code>double[] remove(double[] array, int[] index)</code>	Removes elements from an array and returns the shortened array.
remove	<code>int[] remove(int[] array, int[] index)</code>	Removes elements from an array and returns the shortened array.
remove	<code>boolean[] remove(boolean[] array, int[] index)</code>	Removes elements from an array and returns the shortened array.
insertRow	<code>String[][] insertRow(String[][] matrix, String[] value, int rowIndex)</code>	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	<code>double[][] insertRow(double[][] matrix, double[] value, int rowIndex)</code>	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	<code>int[][] insertRow(int[][] matrix, int[] value, int rowIndex)</code>	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	<code>boolean[][] insertRow(boolean[][] matrix, boolean[] value, int rowIndex)</code>	Inserts a row into a rectangular 2D array and returns the expanded array.
insertRow	<code>String[][] insertRow(String[][] matrix, String[][] value, int[] rowIndex)</code>	Adds rows to a rectangular 2D array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
insertRow	double[][] insertRow(double[][] matrix, double[][] value, int[] rowIndex)	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	int[][] insertRow(int[][] matrix, int[][] value, int[] rowIndex)	Adds rows to a rectangular 2D array and returns the expanded array.
insertRow	boolean[][] insertRow(boolean[][] matrix, boolean[][] value, int[] rowIndex)	Adds rows to a rectangular 2D array and returns the expanded array.
appendRow	String[][] appendRow(String[][] matrix, String[] value)	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	double[][] appendRow(double[][] matrix, double[] value)	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	int[][] appendRow(int[][] matrix, int[] value)	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	boolean[][] appendRow(boolean[][] matrix, boolean[] value)	Adds a row to the end of a rectangular 2D array and returns the expanded array.
appendRow	String[][] appendRow(String[][] matrix, String[] value)	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	double[][] appendRow(double[][] matrix, double[] value)	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	int[][] appendRow(int[][] matrix, int[] value)	Adds rows to the end of a rectangular 2D array and returns the expanded array.
appendRow	boolean[][] appendRow(boolean[][] matrix, boolean[] value)	Adds rows to the end of a rectangular 2D array and returns the expanded array.
removeRow	String[][] removeRow(String[][] matrix, int rowIndex)	Removes a row from a 2D array and returns the smaller array.
removeRow	double[][] removeRow(double[][] matrix, int rowIndex)	Removes a row from a 2D array and returns the smaller array.
removeRow	int[][] removeRow(int[][] matrix, int rowIndex)	Removes a row from a 2D array and returns the smaller array.

NAME	SYNTAX	DESCRIPTION
removeRow	boolean[][] removeRow(boolean[][] matrix, int rowIndex)	Removes a row from a 2D array and returns the smaller array.
removeRow	String[][] removeRow(String[][] matrix, int[] rowIndex)	Removes rows from a 2D array and returns the reduced array.
removeRow	double[][] removeRow(double[][] matrix, int[] rowIndex)	Removes rows from a 2D array and returns the reduced array.
removeRow	int[][] removeRow(int[][] matrix, int[] rowIndex)	Removes rows from a 2D array and returns the reduced array.
removeRow	boolean[][] removeRow(boolean[][] matrix, int[] rowIndex)	Removes rows from a 2D array and returns the reduced array.
insertColumn	String[][] insertColumn(String[][] matrix, String[] value, int columnIndex)	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	double[][] insertColumn(double[][] matrix, double[] value, int columnIndex)	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	int[][] insertColumn(int[][] matrix, int[] value, int columnIndex)	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	boolean[][] insertColumn(boolean[][] matrix, boolean[] value, int columnIndex)	Adds a column into a rectangular 2D array and returns the expanded array.
insertColumn	String[][] insertColumn(String[][] matrix, String[][] value, int[] columnIndex)	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	double[][] insertColumn(double[][] matrix, double[][] value, int[] columnIndex)	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	int[][] insertColumn(int[][] matrix, int[][] value, int[] columnIndex)	Adds columns to a rectangular 2D array and returns the expanded array.
insertColumn	boolean[][] insertColumn(boolean[][] matrix, boolean[][] value, int[] columnIndex)	Adds columns to a rectangular 2D array and returns the expanded array.
appendColumn	String[][] appendColumn(String[][] matrix, String[] value)	Adds a column at the end of a rectangular 2D array and returns the expanded array.

NAME	SYNTAX	DESCRIPTION
appendColumn	<code>double[][] appendColumn(double[][] matrix, double[] value)</code>	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	<code>int[][] appendColumn(int[][] matrix, int[] value)</code>	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	<code>boolean[][] appendColumn(boolean[][] matrix, boolean[] value)</code>	Adds a column at the end of a rectangular 2D array and returns the expanded array.
appendColumn	<code>String[][] appendColumn(String[][] matrix, String[] value)</code>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	<code>double[][] appendColumn(double[][] matrix, double[][] value)</code>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	<code>int[][] appendColumn(int[][] matrix, int[][] value)</code>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
appendColumn	<code>boolean[][] appendColumn(boolean[][] matrix, boolean[][] value)</code>	Adds columns to the end of a rectangular 2D array and returns the expanded array.
removeColumn	<code>String[][] removeColumn(String[][] matrix, int columnIndex)</code>	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	<code>double[][] removeColumn(double[][] matrix, int columnIndex)</code>	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	<code>int[][] removeColumn(int[][] matrix, int columnIndex)</code>	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	<code>boolean[][] removeColumn(boolean[][] matrix, int columnIndex)</code>	Removes a column from a rectangular 2D array and returns the smaller array.
removeColumn	<code>String[][] removeColumn(String[][] matrix, int[] columnIndex)</code>	Removes columns from a rectangular 2D array and returns the reduced array.
removeColumn	<code>double[][] removeColumn(double[][] matrix, int[] columnIndex)</code>	Removes columns from a rectangular 2D array and returns the reduced array.

NAME	SYNTAX	DESCRIPTION
removeColumn	<code>int[][] removeColumn(int[][] matrix, int[] columnIndex)</code>	Removes columns from a rectangular 2D array and returns the reduced array.
removeColumn	<code>boolean[][] removeColumn(boolean[][] matrix, int[] columnIndex)</code>	Removes columns from a rectangular 2D array and returns the reduced array.
matrixSize	<code>int[] matrixSize(String[][] matrix)</code>	Returns the number of rows and columns of a matrix as an integer array of length 2.
matrixSize	<code>int[] matrixSize(double[][] matrix)</code>	Returns the number of rows and columns of a matrix as an integer array of length 2.
matrixSize	<code>int[] matrixSize(int[][] matrix)</code>	Returns the number of rows and columns of a matrix as an integer array of length 2.
matrixSize	<code>int[] matrixSize(boolean[][] matrix)</code>	Returns the number of rows and columns of a matrix as an integer array of length 2.

String Methods

NAME	SYNTAX	DESCRIPTION
concat	<code>String concat(String separator, String ... strs)</code>	Concatenates the given varargs-array of strings into a single string using the given separator.
concat	<code>String[] concat(String colSepar, String rowSepar, String[]... matr)</code>	Concatenates the given string matrix (which can be given as a varargs of rows) into a single string. Puts <code>colSepar</code> between values of columns of a row, and <code>rowSepar</code> between rows.
contains	<code>boolean contains(String[] strs, String str)</code>	Returns true if the given string array <code>strs</code> contains the given string <code>str</code> .
find	<code>int[] find(String[] strs, String str)</code>	Returns an array with the indices to all occurrences of <code>str</code> in <code>strs</code> .
findIn	<code>int findIn(String[] strs, String str)</code>	Returns the index to the first occurrence of <code>str</code> in <code>strs</code> or -1 if no match.
findIn	<code>int findIn(String str, String toFind)</code>	Returns the first index of <code>str</code> that is the start of the substring <code>toFind</code> . If there is no substring matching <code>toFind</code> in <code>str</code> , -1 is returned.
length	<code>int length(String str)</code>	Returns the length of the string <code>str</code> .
replace	<code>String replace(String str, String orig, String replacement)</code>	Returns a string where <code>orig</code> has been replaced by <code>replacement</code> .
split	<code>String[] split(String str)</code>	Returns an array of strings by splitting the given string at spaces.
split	<code>String[] split(String str, String separator)</code>	Returns an array of strings by splitting the given string at the given separator.

NAME	SYNTAX	DESCRIPTION
substring	<code>String substring(String str, int start, int length)</code>	Returns a substring with the given length starting at the given position.
unique	<code>String[] unique(String[] strs)</code>	Returns an array of strings with the unique values in the given array of strings.

Collection Methods

NAME	SYNTAX	DESCRIPTION
copy	<code>String[] copy(String... toCopy)</code>	Returns a copy of the given array of strings, which can also be specified as a varargs of strings.
copy	<code>String[][] copy(String[]... toCopy)</code>	Returns a copy of the given string matrix, which can also be specified as a varargs of rows (string arrays).
copy	<code>double[] copy(double... toCopy)</code>	Returns a copy of the given array of doubles, which can also be specified as a varargs of doubles.
copy	<code>double[][] copy(double[]... toCopy)</code>	Returns a copy of the given double matrix, which can also be specified as a varargs of rows (double arrays).
copy	<code>int[] copy(int... toCopy)</code>	Returns a copy of the given array of integers, which can also be specified as a varargs of integers.
copy	<code>int[][] copy(int[]... toCopy)</code>	Returns a copy of the given integer matrix, which can also be specified as a varargs of rows (integer arrays).
copy	<code>boolean[] copy(boolean... toCopy)</code>	Returns a copy of the given array of booleans, which can also be specified as a varargs of booleans.
copy	<code>boolean[][] copy(boolean[]... toCopy)</code>	Returns a copy of the given boolean matrix, which can also be specified as a varargs of rows (boolean arrays).
equals	<code>boolean equals(String[] str1, String[] str2)</code>	Returns true if all strings in the given array are equal and they have the same number of elements.
equals	<code>boolean equals(String[][] matr1, String[][] matr2)</code>	Returns true if all strings in the given matrix are equal and they have the same number of elements.

NAME	SYNTAX	DESCRIPTION
equals	boolean equals(int[] ints1, int[] ints2)	Returns true if all integers in the given array are equal and they have the same number of elements.
equals	boolean equals(int[][] ints1, int[][] ints2)	Returns true if all integers in the given matrix are equal and they have the same number of elements.
equals	boolean equals(double d11, double d12, double relErrorTolerance)	Compares whether the relative error of two doubles is within allowed tolerance using $\text{abs}((a - b) / b)$, where b is the larger of the doubles (by absolute value).
equals	boolean equals(double d11, double d12)	Same as above, but uses a default <code>relErrorTolerance</code> of 0.0001.
equals	boolean equals(double[] dbls1, double[] dbls2, double relErrorTolerance)	Compares the relative errors ($\sim \text{abs}((a - b) / b)$) of elements in the arrays pairwise and returns true if all the relative errors are below <code>relErrorTolerance</code> and the arrays have the same number of elements.
equals	boolean equals(double[] dbls1, double[] dbls2)	Same as above, but uses a default <code>relErrorTolerance</code> of 0.0001.
equals	boolean equals(double[][] dbls1, double[][] dbls2, double relErrorTolerance)	Compares the relative errors ($\sim \text{abs}((a - b) / b)$) of elements in the matrices pairwise and returns true if all the relative errors are below <code>relErrorTolerance</code> and the matrices have the same number of elements.
equals	boolean equals(double[][] dbls1, double[][] dbls2)	Same as above, but uses a default <code>relErrorTolerance</code> of 0.0001.
sort	sort(String[] strs)	Sorts the given array of strings. NOTE: The array is sorted in place.

NAME	SYNTAX	DESCRIPTION
sort	<code>sort(int[] ints)</code>	Sorts the given array of integers. NOTE: The array is sorted in place.
sort	<code>sort(double[] doubles)</code>	Sorts the given array of doubles. NOTE: The array is sorted in place.
merge	<code>merge(String[]... toMerge)</code>	Returns an array of strings with all the strings merged from the given arrays.
merge	<code>merge(int[]... toMerge)</code>	Returns an array of integers with all the integers from the two given arrays.
merge	<code>merge(double[]... toMerge)</code>	Returns an array of doubles with all the doubles from the two given arrays.

File Methods

NAME	SYNTAX	DESCRIPTION
readFile	<code>String readFile(String name)</code>	Returns the contents in the given file name as a string. The string name is the absolute path to a file or a path given by the file scheme syntax as described in “Appendix C—File Handling and File Scheme Syntax” on page 241.
openFileStreamReader	<code>CsReader openFileStreamReader(String name)</code>	Returns a CsReader that can be used to read line by line or character-by-character from the given file name .
openBinaryFileStreamReader	<code>CsBinaryReader openBinaryFileStreamReader(String name)</code>	Returns a CsBinaryReader that can be used to read from the given file byte by byte.
readMatrixFromFile	<code>double[][] readMatrixFromFile(String name)</code>	Reads the contents of the given file into a double matrix. The file has the same spreadsheet type format as available in the model tree Export node.
readStringMatrixFromFile	<code>String[][] readStringMatrixFromFile(String name)</code>	Reads the contents of the given file into a string matrix. The file has the same spreadsheet type format as available in the model tree Export node.
readCSVFile	<code>String[][] readCSVFile(String name)</code>	Reads a file with comma-separated values (CSV file) into a string matrix. Expects file to use the RFC 4180 format for CSV.
writeFile	<code>writeFile(String name, String contents)</code>	Writes the given string contents to the given file name .
writeFile	<code>writeFile(String name, double[][] data)</code>	Writes the array data to the given file. The spreadsheet format is used, which means it can be read by readMatrixFromFile .

NAME	SYNTAX	DESCRIPTION
<code>writeFile</code>	<code>writeFile(String name, String[][] data)</code>	Writes the array <code>data</code> to the given file. The spreadsheet format is used, which means it can be read by <code>readStringMatrixFromFile</code> .
<code>openFileStreamWriter</code>	<code>CsStreamWriter openFileStreamWriter(String name)</code>	Returns a <code>CsStreamWriter</code> that can write to the given file.
<code>openBinaryFileStreamWriter</code>	<code>CsBinaryWriter openBinaryFileStreamWriter(String name)</code>	Returns a <code>CsBinaryWriter</code> that can be used to write to the given file byte by byte.
<code>writeCSVFile</code>	<code>writeCSVFile(String name, String[][] data)</code>	Writes the given string array <code>data</code> to a CSV file. The RFC 4180 format is used for the CSV.
<code>writeCSVFile</code>	<code>writeCSVFile(String name, double[][] data)</code>	Writes the given double array <code>data</code> to a CSV file. The RFC 4180 format is used for the CSV.
<code>exists</code>	<code>boolean exists(String name)</code>	Tests whether a file with the given <code>name</code> exists. If the <code>name</code> is not a file scheme path name or an absolute path, then the method first finds out whether a file with file scheme path <code>embedded:/// + argument</code> exists. If such a file does not exist, then it tests whether there is a file with a matching <code>name</code> in the current working directory.
<code>deleteFile</code>	<code>deleteFile(String file)</code>	Delete a file with the given name if it exists. The file is deleted on the server. The name can use a file scheme path.
<code>copyFile</code>	<code>copyFile(String sourceFile, String destFile)</code>	Copies a file on the server. Both the source and target names can use file scheme paths.

NAME	SYNTAX	DESCRIPTION
<code>importFile</code>	<code>importFile(String name)</code>	Displays a file browser dialog box and uploads the selected file to the file declaration with the given <code>name</code> . After this, the uploaded file can be accessed with <code>upload:///<name></code> .
<code>importFile</code>	<code>importFile(ModelEntity entity, String name)</code>	Displays a file browser dialog box and uploads the selected file to the Filename text field in the given model object entity. This defines an input file that the application will need at a later stage. For example, the Filename of an interpolation function accessed with <code>model.func('<tag>')</code> . The uploaded file can be accessed with <code>upload:///<tag>/filename</code> .
<code>writeExcelFile</code>	<code>writeExcelFile(String name, String[][] data)</code>	Writes the given string array <code>data</code> starting from the first cell in the first sheet of an Excel file.
<code>writeExcelFile</code>	<code>writeExcelFile(String name, String sheet, String cell, String[][] data)</code>	Writes the given string array <code>data</code> starting from the specified cell in the specified sheet of an Excel file.
<code>readExcelFile</code>	<code>String[][] readExcelFile(String name)</code>	Reads the first sheet of an Excel file, starting from the first cell, into a <code>String[][],</code>
<code>readExcelFile</code>	<code>String[][] readExcelFile(String name, String sheet, String cell)</code>	Reads the specified sheet of an Excel file, starting from the specified cell, into a <code>String[][],</code>
<code>getFilePath</code>	<code>String getFilePath(String name)</code>	Returns the absolute server file path of the server proxy file corresponding to a certain file scheme path, or null if the server proxy file for the given path does not exist. This method can be used to pass the path to, for example, a file using the <code>temp:///</code> scheme to external code or an application.

NAME	SYNTAX	DESCRIPTION
<code>getClientFileName</code>	<code>String getClientFileName(String name)</code>	<p>Returns the original name of an uploaded file on the client file system (or null if there is no uploaded file matching the given file scheme path).</p> <p>This method is only useful for providing user interface feedback. For example, to get information on which uploaded file is being used. There is no guarantee that the original file would still exist on the client or even that the current client would be the same as the original client.</p>
<code>getClientFilePath</code>	<code>String getClientFilePath(String name)</code>	<p>Returns the original path of an uploaded file on the client file system (or null if there is no uploaded file matching the given file scheme path).</p> <p>This method is only useful for providing user interface feedback. For example, to get information on which uploaded file is being used. There is no guarantee that the original file would still exist on the client or even that the current client would be the same as the original client.</p>

EXAMPLE CODE

This line of code copies the uploaded file `file1` to the `temp` folder with new file name `file2.mphbin`.

```
copyFile("upload:///file1", "temp:///file2.mphbin");
```

This line of code deletes the file `file2.mphbin` from the `temp` folder.

```
deleteFile("temp:///file2.mphbin");
```

For example code on file handling, see also the section “Appendix C—File Handling and File Scheme Syntax” on page 241.

System Methods

NAME	SYNTAX	DESCRIPTION
executeOSCommand	<code>String executeOSCommand(String command, String... params)</code>	Executes the OS command with the given command (full path) and parameters. Execution times out after a default 180 seconds. Returns everything the process printed to its out stream as a string. When applicable, the command is run server side.
executeOSCommand	<code>String executeOSCommand(String command, int timeoutSec, String params...)</code>	Executes the OS command with the given command (full path) and parameters. Returns everything the process printed to its out stream as a string. The execution is forcibly stopped after <code>timeoutSec</code> seconds if the command has not finished. To disable the timeout functionality, <code>timeoutSec</code> value 0 can be used. When applicable, the command is run server side.
fileOpen	<code>fileOpen(String name)</code>	Opens the file represented by name with the associated program on the client. Also see the section “File Methods”.
getUser	<code>String username = getUser()</code>	Returns the username of the user that is running the application. If the application is not run from COMSOL Server; then the value of the preference setting <code>General>Username>Name</code> is returned.
openURL	<code>openURL(String url)</code>	Opens a URL in the default browser on the client.

NAME	SYNTAX	DESCRIPTION
playSound	<code>playSound(String name)</code>	Plays the sounds in the given file on the client. Only <code>.wav</code> files are supported; no external libraries are required.
playSound	<code>playSound(double hz, int millis)</code>	Plays a signal at a given frequency <code>hz</code> and with given duration <code>millis</code> in milliseconds on the client.

EXAMPLE CODE

The line of code below plays one of the sounds available in the `data/sounds` folder of the COMSOL installation and has been embedded in the application and stored in the Sounds library.

```
playSound("embedded:///success_1.wav");
```

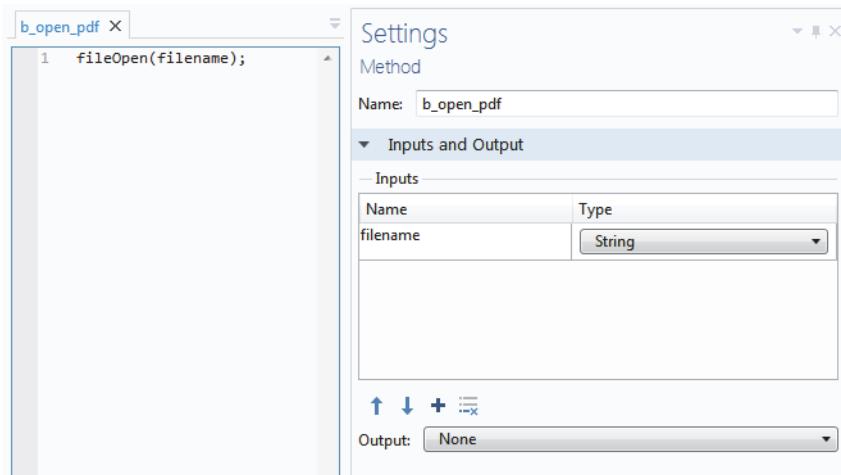
In the command sequence of a form object, this is equivalent to selecting a sound node under **Libraries** and clicking **Run**.

The line of code below opens a PDF file embedded in the application and stored in the File library.

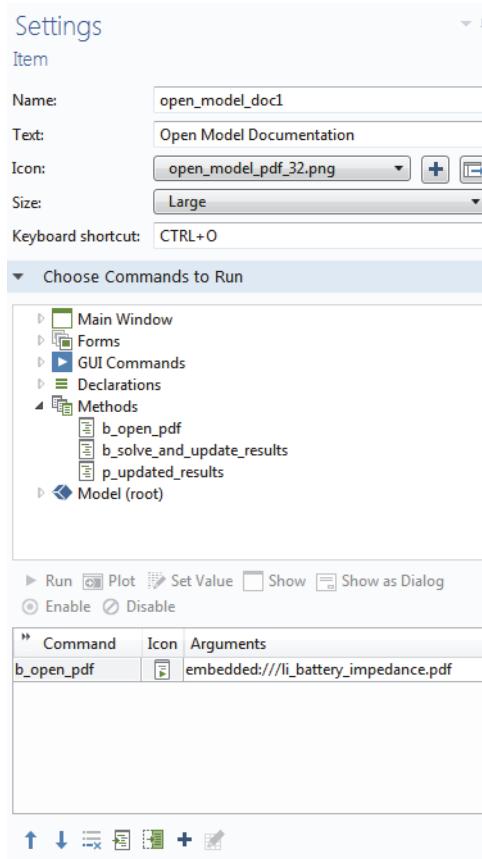
```
fileOpen("embedded:///tubular_reactor.pdf");
```

In the command sequence of a form object, this is equivalent to selecting an **Open File** node under **GUI Commands>File Commands** and clicking **Run**.

As an alternative technique, you can call a method in a command sequence with an input argument, as shown in the example below. The figure below shows a method `b_open_pdf` that opens a file with `filename` as an input argument.



The figure below shows the corresponding command sequence for a ribbon menu item.



Note that the same functionality is available from a command sequence by selecting the editor tree node **GUI Commands>File Commands>Open File**.

This line of code opens the COMSOL home page in the default browser:

```
openURL("http://www.comsol.com");
```

Email Methods

NAME	SYNTAX	DESCRIPTION
emailFromAddress	<code>String emailFromAddress()</code>	Returns the email from address from the COMSOL Server or preferences setting.
sendEmail	<code>sendEmail(String subject, String bodyText)</code>	Sends an email to the default recipient(s) with the specified subject and body text.
sendEmail	<code>sendEmail(String subject, String bodyText, ModelEntity... modelEntity)</code>	Sends an email to the default recipient(s) with the specified subject, body text, and zero or more attachments created from Report, Export, and Table nodes in the embedded model.
sendEmail	<code>sendEmail(String toAddress, String subject, String bodyText, ModelEntity... modelEntity)</code>	Sends an email to the specified recipient(s) with the specified subject, body text, and zero or more attachments created from Report, Export, and Table nodes in the embedded model.
userEmailAddress	<code>String userEmailAddress()</code>	Returns the user email address(es) corresponding to the currently logged in user; or an empty string if the user has not configured an email address.

Email Class Methods

The class `EmailMessage` can be used to create custom email messages.

NAME	SYNTAX	DESCRIPTION
<code>EmailMessage</code>	<code>EmailMessage mail = new EmailMessage()</code>	Creates a new <code>EmailMessage</code> object.
<code>EmailMessage.setServer</code>	<code>mail.setServer(String host, int port)</code>	Sets the email (SMTP) server host and port to use for this email message.
<code>EmailMessage.setUser</code>	<code>mail.setUser(String name, String password)</code>	Sets the user name and password to use for email (SMTP) server authentication. This method must be called after the <code>setServer</code> method.

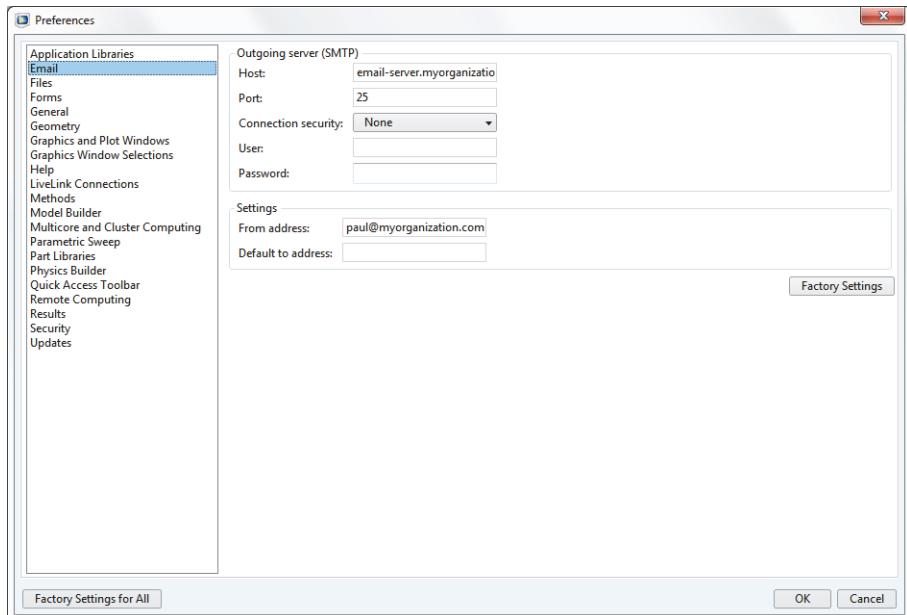
NAME	SYNTAX	DESCRIPTION
EmailMessage.setSecurity	<code>mail.setSecurity(String security)</code>	Sets the connection security type for email (SMTP) server communication. Valid values are ‘none’, ‘starttls’ and ‘tls’. This method must be called after the <code>setServer</code> method.
EmailMessage.setFrom	<code>mail.setFrom(String fromAddress)</code>	Sets the from address.
EmailMessage.setTo	<code>mail.setTo(String... to)</code>	Sets the to addresses.
EmailMessage.setCc	<code>mail.setCc(String... cc)</code>	Sets the cc addresses.
EmailMessage.setBcc	<code>mail.setBcc(String... bcc)</code>	Sets the bcc addresses.
EmailMessage.setSubject	<code>mail.setSubject(String subject)</code>	Sets the email subject line. Note that newline characters are not allowed.
EmailMessage.setBodyText	<code>mail.setBodyText(String body)</code>	Sets the email body as plain text. An email can contain both a text and an HTML body.
EmailMessage.setBodyHtml	<code>mail.setBodyHtml(String body)</code>	Sets the email body as html text. An email can contain both a text and an HTML body.
EmailMessage.attachFile	<code>mail.attachFile(String filename)</code>	Adds an attachment from a file. The attachment MIME type is determined by the file name extension.
EmailMessage.attachFile	<code>mail.attachFile(String filename, String mimeType)</code>	Adds an attachment from a file with the specified MIME type.
EmailMessage.attachFromModel	<code>mail.attachFromModel(ModelEntity modelEntity)</code>	Adds an attachment created from a report, export, or table feature in the model.
EmailMessage.attachText	<code>mail.attachText(String text, String mimeSubType)</code>	Adds a text attachment with a specified sub-MIME type, such as plain or HTML.

NAME	SYNTAX	DESCRIPTION
EmailMessage. attachBinary	<code>mail.attachBinary(byte[] binary, String mimeType)</code>	Adds an attachment from a byte array with the specified MIME type.
EmailMessage. send	<code>mail.send()</code>	Sends the email to the email (SMTP) server. An email object can only be sent once.

Each `to`, `cc`, and `bcc` address string can contain multiple email addresses separated by a comma or a semicolon character. Whitespace is allowed before and after the separator character.

EMAIL PREFERENCES

To set preferences for an outgoing email (SMTP) server, open the **Email** page of the **Preferences** dialog box, as shown in the figure below.



COMSOL Server provides a similar set of email preferences.

EXAMPLE CODE

The following code sends an email and attaches a report:

```
EmailMessage mail = new EmailMessage();
mail.setTo(email_to);
```

```

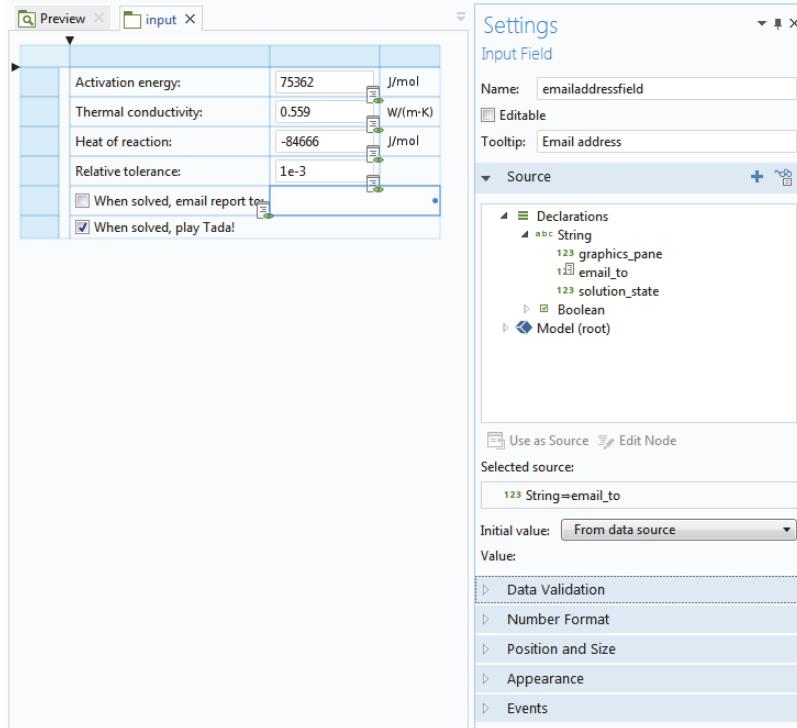
mail.setSubject("Tubular Reactor Simulation");
mail.setBodyText("The computation has finished. Please find the report
attached.");
mail.attachFromModel(model.result().report("rpt1"));
mail.send();

```

This code is run in the Tubular Reactor application, which is available as an application example in the Application Libraries. The figure below shows part of the user interface with an input field for the email address.



The figure below shows the corresponding form object and **Settings** window.



The following code is similar but also configures the email server settings.

```

EmailMessage mail = new EmailMessage();
mail.setServer("smtp.myemail.com", 587);
mail.setUser("user@myemail.com", "password");
mail.setSecurity("starttls");
mail.setFrom("user@myemail.com");

```

```

mail.setTo("otheruser@somedomain.com");
mail.setSubject("Tubular reaction simulation");
mail.setBodyText("The computation has finished");
mail.send();

```

Model Utility Methods

The model utility methods make it possible to load the model object part of an MPH file into a method for further processing.

NAME	SYNTAX	DESCRIPTION
createModel	Model createModel(String tag)	Creates a new model with the given tag.
removeModel	removeModel(String tag)	Removes a model. The embedded model cannot be removed.
modelTags	String[] modelTags()	Returns an array of model tags for all loaded models, including the embedded model.
uniqueModeltag	String uniqueModeltag(String prefix)	Returns a model tag that is not in use.
getModel	Model getModel(String tag)	Returns the model with a specified tag.
loadModel	Model loadModel(String tag, String filename)	Loads a model with a specified tag from a file.
loadProtectedModel	Model loadProtectedModel(String tag, String filename, String password)	Loads a password protected model with a specified tag from a file.
loadRecoveryModel	Model loadRecoveryModel(String tag, String foldername)	Loads a model from a recovery directory/folder structure.
saveModel	saveModel(Model model, String filename)	Saves a model to a file. The filename can be a file scheme path or (if allowed by security settings) a server file path.

EXAMPLE CODE

The code below loads a model using `loadModel` from the table above. It extracts the x-, y-, and z-coordinates of all the mesh nodes and stores them in a 2D double array `coords[3][N]`, where N is the number of mesh nodes. The individual x-,y-, and z- coordinates are available as the length-N 1D arrays `coords[0]`, `coords[1]`, `coords[2]`, respectively. (The node locations can be plotted by using the Cut Point 3D data set in combination with a 3D Point Trajectories plot.)

```
Model extmodel = loadModel("model", "C:\\Paul\\pacemaker_electrode.mph");
SolverFeature step = extmodel.sol("sol1").feature("v1");
XmeshInfo xmi = step.xmeshInfo();
XmeshInfoNodes nodes = xmi.nodes();
double[][][] coords = nodes.coords();
```

For more information on methods operating on the model object, see the *Programming Reference Manual*.

GUI-Related Methods

NAME	SYNTAX	DESCRIPTION
Call a method directly	<methodName>()	Call a method from the Methods list by using its name, for example, <code>method1()</code> , <code>method2()</code> .
<code>callMethod</code>	<code>callMethod(String name)</code>	Alternate way to call a method from the Methods list; used internally and in cases of name collisions.
<code>useGraphics</code>	<code>useGraphics(ModelEntity entity, String name)</code>	Plots the given entity (Plot Group, Geometry, Mesh or Explicit Selection) in the graphics form object given by the name or name path in the second argument.
<code>useForm</code>	<code>useForm(String name)</code>	Shows the form with the given name in the current main window. Equivalent to the <code>use</code> method of a <code>Form</code> object; see below.
<code>closeDialog</code>	<code>closeDialog(String name)</code>	Closes the form, shown as a dialog box, with the given name.
<code>dialog</code>	<code>dialog(String name)</code>	Shows the form with the given name as a dialog box. Equivalent to the <code>dialog</code> method of a <code>Form</code> object; see below.
<code>alert</code>	<code>alert(String text)</code>	Stops execution and displays an alert message with the given text.
<code>alert</code>	<code>alert(String text, String title)</code>	Stops execution and displays an alert message with the given text and title.

NAME	SYNTAX	DESCRIPTION
confirm	<code>String confirm(String text)</code>	Stops execution and displays a confirmation dialog box with the given text. It also displays two buttons, "Yes" and "No". The method returns "Yes" or "No" depending on what the user clicks.
confirm	<code>String confirm(String text, String title)</code>	Stops execution and displays a confirmation dialog box with the given text and title. It also displays two buttons, "Yes" and "No". The method returns "Yes" or "No" depending on what the user clicks.
confirm	<code>String confirm(String text, String title, String yes, String no)</code>	Stops execution and displays a confirmation dialog box with the given text and title. It also displays two buttons with the given strings as labels. The method returns the label of the button that the user clicks.
confirm	<code>String confirm(String text, String title, String yes, String no, String cancel)</code>	Stops execution and displays a confirmation dialog box with the given text and title. It also displays three buttons with the given strings as labels. The method returns the label of the button that the user clicks.
error	<code>error(String message)</code>	Stops execution and opens an error dialog box with the given message.

NAME	SYNTAX	DESCRIPTION
request	<code>String request(String text)</code>	Stops execution and displays a dialog box with a text field, requesting input from the user. The given text is the label of the text field. The method returns the entered text or <code>null</code> if the cancel button is clicked.
request	<code>String request(String text, String defaultString)</code>	Stops execution and displays a dialog box with a text field, requesting input from the user. The given text is the label of the text field and the default string is the text initially shown in the text field. The method returns the entered text or <code>null</code> if the cancel button is clicked.
request	<code>String request(String text, String title, String defaultString)</code>	Stops execution and displays a dialog box with a text field, requesting input from the user. The given text is the label of the text field, the default string is the text initially shown in the text field, and the title is the title of the dialog box. The method returns the entered text or <code>null</code> if the cancel button is clicked.
message	<code>message(String message)</code>	Sends a message to the message log if available in the application.

NAME	SYNTAX	DESCRIPTION
evaluateToResultsTable	<code>evaluateToResultsTable(Numeric alFeature entity, String name, boolean clear)</code>	Evaluates the given entity, a Derived Value, in the table object given by the name or name path in the second argument, which will then be the default target for the evaluations of the Derived Value. If the third argument is true, the table is cleared before adding the new data, otherwise the data is appended.
evaluate.ToDoubleArray2D	<code>double[][] evaluate.ToDoubleArray2D(Numeri calFeature entity)</code>	Evaluates the given entity, a Derived Value, and returns the non-parameter column part of the real table that is produced as a double matrix. All settings in the numerical feature are respected but those in the current table connected to the numerical feature are ignored.
evaluate.ToIntegerArray2D	<code>int[][] evaluate.ToIntegerArray2D(Nume ricalFeature entity)</code>	Evaluates the given entity, a Derived Value, and returns the non-parameter column part of the real table that is produced as an integer matrix. All settings in the numerical feature are respected but those in the current table connected to the numerical feature are ignored.

NAME	SYNTAX	DESCRIPTION
<code>evaluateToStringArray2D</code>	<code>String[][] evaluateToStringArray2D(NumericalFeature entity)</code>	Evaluates the given entity, a Derived Value, and returns the non-parameter column part of the, potentially complex, table that is produced as a string matrix. All settings in the numerical feature are respected but those in the current table connected to the numerical feature are ignored.
<code>useResultsTable</code>	<code>useResultsTable(TableFeature tableFeature, String resultsTable)</code>	Shows the values from the <code>tableFeature</code> in the <code>resultsTable</code> form object.
<code>getChoiceList</code>	<code>ChoiceList getChoiceList(String name)</code>	Returns an object of the type <code>ChoiceList</code> , representing a choice list node under the declarations branch. The type <code>ChoiceList</code> has methods that make it easier to change the matrix value with respect to changing and accessing values and display names individually.
<code>setFormObjectEnabled</code>	<code>setFormObjectEnabled(String name, boolean enabled)</code>	Sets the enable state for the form object specified by the name or name path.
<code>setFormObjectVisible</code>	<code>setFormObjectVisible(String name, boolean visible)</code>	Sets the visible state for the form object specified by the name or name path.
<code>setFormObjectText</code>	<code>setFormObjectText(String name, String text)</code>	Sets the text for the form object specified by the name or name path in the second argument. This method throws an error if it is impossible to set a text for the specified form object.
<code>setFormObjectEditable</code>	<code>setFormObjectEditable(String name, boolean editable)</code>	Sets the editable state for the form object specified by the name or name path. This functionality is only available for text field objects.

NAME	SYNTAX	DESCRIPTION
<code>setMenuBarItemEnabled</code>	<code>setMenuBarItemEnabled(String name, boolean enabled)</code>	Sets the enable state for the menu bar item specified by the <code>name</code> or name path (from menu bar) in the first argument.
<code>setMainToolbarItemEnabled</code>	<code>setMainToolbarItemEnabled(String name, boolean enabled)</code>	Sets the enable state for the main toolbar item specified by the name or name path (from main toolbar) in the first argument.
<code>setFileMenuItemEnabled</code>	<code>setFileMenuItemEnabled(String name, boolean enabled)</code>	Sets the enable state for the file menu item specified by the name or name path (from file menu) in the first argument.
<code>setRibbonItemEnabled</code>	<code>setRibbonItemEnabled(String name, boolean enabled)</code>	Sets the enable state for the ribbon item specified by the name or name path (from main window) in the first argument.
<code>setToolbarItemEnabled</code>	<code>setToolbarItemEnabled(String name, boolean enabled)</code>	Sets the enable state for the toolbar form object item specified by the name or name path in the first argument.
<code>useView</code>	<code>useView(View view, String name)</code>	Applies a view to the graphics contents given by the name or name path in the second argument.
<code>resetView</code>	<code>resetView(String name)</code>	Resets the view to its initial state in the graphics contents given by the name or name path in the second argument.
<code>getView</code>	<code>View getView(String name)</code>	Returns the view currently used by the graphics contents given by the name or name path in the second argument.

NAME	SYNTAX	DESCRIPTION
<code>setWebPageSource</code>	<code>setWebPageSource(String name, String source)</code>	Sets the source for the form object specified by the name or name path in the first argument. This method throws an error if the name does not refer to a Web Page form object.
<code>getScreenHeight</code>	<code>int getScreenHeight()</code>	Returns the height in pixels of the primary screen on client system, or of the browser window if Web Client is used.
<code>getScreenWidth</code>	<code>int getScreenWidth()</code>	Returns the width in pixels of the primary screen on client system, or of the browser window if Web Client is used.

EXAMPLE CODE

This line of code displays plot group 5 (pg5) in the graphics object `graphics1` in the form with the name `Temperature`:

```
useGraphics(model.result("pg5"), "/Temperature/graphics1");
```

The code below displays the mesh in the model tree node `mesh1` in the graphics object `graphics1` contained in the card of a card stack. The second line runs a `zoomExtents` command to ensure proper visualization of the mesh.

```
useGraphics(model.mesh("mesh1"), "/mesh/cardstack1/card1/graphics1");
zoomExtents("/mesh/cardstack1/card1/graphics1");
```

The code below displays a request dialog box that lets the user type in a file name for an HTML report. If the user has typed a file name, then a report is generated.

```
String answerh = request("Enter file name", "File Name", "Untitled.html");
if(answerh != null){
    model.result().report("rpt1").set("format", "html");
    model.result().report("rpt1").set("filename", "appuser:///"+answerh);
    model.result().report("rpt1").run();
}
```

The code below is similar to the code above, but in this case the report is saved in Microsoft® Word® format (.docx).

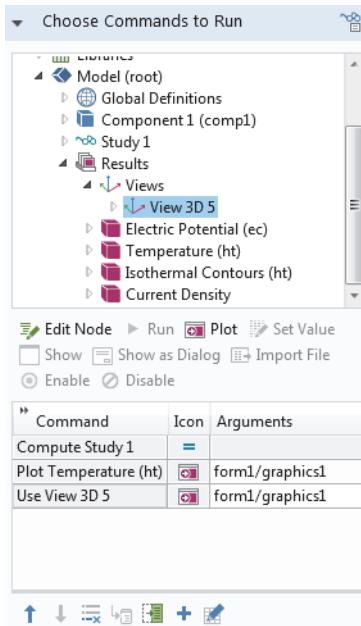
```
String answerw = request("Enter file name", "File Name", "Untitled.docx");
if(answerw != null){
    model.result().report("rpt2").set("format", "docx");
    model.result().report("rpt2").set("filename", "appuser:///"+answerw);
```

```
    model.result().report("rpt2").run();  
}
```

This line of code sets the view of the graphics object `form1/graphics1` to View 5, as defined in the model tree:

```
useView(model.view("view5"), "form1/graphics1");
```

Note that you can also set a view from the command sequence of, for example, a button: select a view subnode under the **Views** node in the editor tree and click the **Plot** button under the tree.



This line of code sets the URL source of the form object `webpage1` to the COMSOL web page:

```
setWebPageSource("/form1/webpage1", "http://www.comsol.com");
```

This line of code forms a string containing the screen width and height:

```
screenSize =  
    toString(getScreenWidth()) + "-by-" + toString(getScreenHeight());
```

You can present the string with an input field or a data display object using this string as a source (the string `screenSize` needs to be declared first).

ChoiceList Methods

NAME	SYNTAX	DESCRIPTION
<code>setListRow</code>	<code>setListRow(String value, String displayName, int row)</code>	Sets the value and display name for the given row (0-based). If the row is equal to the length of the list, a new row is added.
<code>setList</code>	<code>setList(String[] values, String[] displayNames)</code>	Sets all of the values and display names, replacing the contents of the choice list.
<code>setValue</code>	<code>setValue(String value, int row)</code>	Sets the value for the given row (0-based). If the row is equal to the length of the list, a new row is added with the value and an empty display name.
<code>setDisplayName</code>	<code>setDisplayName(String displayName, int row)</code>	Sets the display name for the given row (0-based). If the row is equal to the length of the list, a new row is added with the display name and an empty value.
<code>getValue</code>	<code>String getValue(int row)</code>	Returns the value for the given row (0-based).
<code>getDisplayName</code>	<code>String getDisplayName(int row)</code>	Returns the display name for the given row (0-based).
<code>getValues</code>	<code>String[] getValues()</code>	Returns all values as an array.
<code>getDisplayNames</code>	<code>String[] getDisplayNames()</code>	Returns all display names as an array.

EXAMPLE CODE

The code below adds the string Aluminum 3004 to the choice list from the example in the section “Using a Combo Box to Change Material” on page 177. Note that the choice list index starts at 0, whereas the material tags start at 1 (`mat1`, `mat2`, `mat3`, and `mat4`).

```
ChoiceList choiceList = getChoiceList("choicelist1");
choiceList.setListRow("mat4", "Aluminum 3004", 3);
```

Built-In GUI Methods

NAME	SYNTAX	DESCRIPTION
clearAllMeshes	clearAllMeshes()	Clears all meshes.
clearAllSolutions	clearAllSolutions()	Clears all solutions.
exit	exit()	Exits the application.
fileOpen	fileOpen(String name)	Opens a file with the associated program on the client.
fileSaveAs	fileSaveAs(String file)	Downloads a file to the client. See also the section "File Methods".
printGraphics	printGraphics(String graphicsName)	Prints the given graphics object.
saveApplication	saveApplication()	Saves the application.
saveApplicationAs	saveApplicationAs()	Saves the application under a different name. (Or as an MPH file.)
scenelight	sceneLight(String graphicsName)	Toggle scene light in the given graphics object.
transparency	transparency(String graphicsName)	Toggle transparency in the given graphics object.
updateGraphics	updateGraphics()	Immediately update all graphics form objects with any pending changes, such as those generated by useGraphics and zoomExtents. If this method is not called, the updates will occur when the current command sequence finishes and only the last updated graphics contents are displayed.
zoomExtents	zoomExtents(String graphicsName)	Makes the entire model visible in the given graphics object.

EXAMPLE CODE

The line of code below saves a document to the user folder (as specified in the Preferences).

```
fileSaveAs("user:///mixer.docx");
```

The following code changes the camera position and updates the graphics for each change by using the built-in method `updateGraphics`.

```
useView(model.view("view1"), "/form1/graphics1");
int N = 25;
for (int i = 0; i < N; i++) {
    sleep(2000);
    model.view("view1").camera().set("zoomanglefull", 12-i*5.0/N);
    useGraphics(model.geom("geom1"), "/form1/graphics1");
    updateGraphics();
}
```

With, Get, and Set Methods

NAME	SYNTAX	DESCRIPTION
with	<code>with(target)</code>	<p>Sets a target to use with subsequent calls to <code>set()</code>, <code>setIndex()</code>, <code>getString()</code> etc.</p> <p>Unset the target afterward with <code>endwith()</code>. The calls to <code>with()/endwith()</code> can be nested.</p> <p>Supported target types (from the COMSOL Model Object) are:</p> <ul style="list-style-type: none"> <code>PropFeature</code> <code>ParameterEntity</code> <code>PhysicsProp</code> <code>ElemFeature</code> <code>ExpressionBase</code> <code>GeomObjectSelection</code> <code>MaterialModel</code> <code>ShapeFeature</code> <code>FeatureInfo</code>
endwith	<code>endwith()</code>	Unsets the target set by the last call to <code>with()</code> , restoring the target that was set before it. The calls to <code>with()/endwith()</code> can be nested.
set	<code>set(String name, boolean value)</code>	Sets the Boolean property value for the target set by the last call to <code>with()</code> .
set	<code>set(String name, boolean[] value)</code>	Sets the boolean vector property or parameter value for the target set by the last call to <code>with()</code> .
set	<code>set(String name, boolean[][] value)</code>	Sets the boolean matrix property or parameter value for the target set by the last call to <code>with()</code> .
set	<code>set(String name, double value)</code>	Sets a double property or parameter value for the target set by the last call to <code>with()</code> . If the target set was of the <code>ExpressionBase</code> type, this sets the variable name and expression.

NAME	SYNTAX	DESCRIPTION
set	<code>set(String name, double[] value)</code>	Sets a double vector property or parameter value for the target set by the last call to <code>with()</code> .
set	<code>set(String name, double[][] value)</code>	Sets double matrix property or parameter value for the target set by the last call to <code>with()</code> .
set	<code>set(String name, int value)</code>	Sets an integer property or parameter value for the target set by the last call to <code>with()</code> . If the target is of the GeomObjectSelection type, this sets the selection of the given object to be an entity.
set	<code>set(String name, int[] value)</code>	Sets an integer vector property or parameter value for the target set by the last call to <code>with()</code> . If the target is of GeomObjectSelection type, this sets the selection to be the given entities of the given object and does not affect the selection of other objects.
set	<code>set(String name, int[][] value)</code>	Sets an integer matrix property or parameter value for the target set by the last call to <code>with()</code> .
set	<code>set(String name, String value)</code>	Sets a string property or parameter value for the target set by the last call to <code>with()</code> . If the target set was of the ExpressionBase type, this sets the variable name and expression.
set	<code>set(String name, String[] value)</code>	Sets the string vector property or parameter value for the target set by the last call to <code>with()</code> . If the target is of the FeatureInfo type, this sets the user-defined expressions for the first definition.
set	<code>set(String name, String[][] value)</code>	Sets the string matrix property or parameter value.

NAME	SYNTAX	DESCRIPTION
set	<code>set(String var, String expr, String descr)</code>	Sets a variable name, expression, and description for the target set by the last call to <code>with()</code> .
set	<code>set(String var, double expr, String descr)</code>	Sets a variable name, expression, and description for the target set by the last call to <code>with()</code> .
set	<code>set(String... onames)</code>	Sets the selection to be the given objects for the target set by the last call to <code>with()</code> .
set	<code>set(String[] onames, int[][] entities)</code>	Sets the selection to be the given entities of the given objects for the target set by the last call to <code>with()</code> . Does not affect the selection of other objects.
setIndex	<code>setIndex(String name, String value, int index)</code>	Sets a string vector property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> . Index in <code>vector</code> is 0-based.
setIndex	<code>setIndex(String name, String value, int firstIndex, int secondIndex)</code>	Sets a string matrix property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .
setIndex	<code>setIndex(String name, String[] value, int index)</code>	Sets a string matrix property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .
setIndex	<code>setIndex(String name, double value, int index)</code>	Sets a string vector property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .
setIndex	<code>setIndex(String name, double value, int firstIndex, int secondIndex)</code>	Sets a string matrix property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .
setIndex	<code>setIndex(String name, double[] value, int index)</code>	Sets a double matrix property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .

NAME	SYNTAX	DESCRIPTION
<code>setIndex</code>	<code>setIndex(String name, int value, int index)</code>	Sets a string vector property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .
<code>setIndex</code>	<code>setIndex(String name, int value, int firstIndex, int secondIndex)</code>	Sets a string matrix property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .
<code>setIndex</code>	<code>setIndex(String name, int[] value, int index)</code>	Sets an integer matrix property or parameter value at <code>index</code> for the target set by the last call to <code>with()</code> .
<code>getIntArray</code>	<code>int[] getIntArray(String name)</code>	Gets an integer vector property for the target set by the last call to <code>with()</code> .
<code>getIntMatrix</code>	<code>int[][] getIntMatrix(String name)</code>	Gets an integer matrix property for the target set by the last call to <code>with()</code> .
<code>getBoolean</code>	<code>boolean getBoolean(String name)</code>	Gets a Boolean property for the target set by the last call to <code>with()</code> .
<code>getBooleanArray</code>	<code>boolean[] getBooleanArray(String name)</code>	Gets boolean vector property or parameter for the target set by the last call to <code>with()</code> . If the target is of <code>FeatureInfo</code> type, this returns the user-defined expressions for the first definition.
<code>getBooleanMatrix</code>	<code>boolean[][] getBooleanMatrix(String name)</code>	Gets boolean matrix property or parameter for the target set by the last call to <code>with()</code>
<code>getDouble</code>	<code>double getDouble(String name)</code>	Gets a double property for the target set by the last call to <code>with()</code> .
<code>getString</code>	<code>String getString(String name)</code>	Gets a string property or parameter for the target set by the last call to <code>with()</code> .
<code>getString</code>	<code>String getString(String name, int index)</code>	Gets a string value from a string vector property for the target set by the last call to <code>with()</code> .

NAME	SYNTAX	DESCRIPTION
getString	<code>String getString(String name, int firstIndex, int secondIndex)</code>	Gets a string value from a string matrix property for the target set by the last call to <code>with()</code> .
getDoubleArray	<code>double[] getDoubleArray(String name)</code>	Gets a double vector property or parameter for the target set by the last call to <code>with()</code> .
getDoubleMatrix	<code>double[][] getDoubleMatrix(String name)</code>	Gets a double matrix property or parameter for the target set by the last call to <code>with()</code> .
getStringArray	<code>String[] getStringArray(String name)</code>	Gets a string vector property or parameter for the target set by the last call to <code>with()</code> . If the target is of the type <code>FeatureInfo</code> , this returns the user-defined expressions for the first definition
getStringArray	<code>String[] getStringArray(String name, int sdim)</code>	Returns the user-defined expressions for the given definition for the target set by the last call to <code>with()</code> . The argument <code>name</code> is the identifier name, for example, the variable name. The argument <code>sdim</code> is the occurrence of the definition, which can be larger than zero for variables that have more than one definition on different geometric entities.
getStringMatrix	<code>String[][] getStringMatrix(String name)</code>	Gets a string matrix property or parameter for the target set by the last call to <code>with()</code> .
getDblStringArray	<code>String[][] getDbIStringArray(String name)</code>	Returns the value as a matrix of strings for the target set by the last call to <code>with()</code> .
getInt	<code>int getInt(String name)</code>	Gets an integer property for the target set by the last call to <code>with()</code> .

NAME	SYNTAX	DESCRIPTION
get	<code>String get(String var)</code>	Returns a variable expression for the target set by the last call to <code>with()</code> .
descr	<code>String descr(String var)</code>	Returns a variable description for the target set by the last call to <code>with()</code> .

EXAMPLE CODE

The code below sets the global parameter L to a fixed value.

```
with(model.param());
  set("L", "10[cm]");
endwith();
```

The code below sets the material link index to the string variable alloy, defined under the **Declarations** node.

```
with(model.material("matlnk1"));
  set("link", alloy);
endwith();
```

The code below sets the coordinates of a cut point data set cpt1 to the values of the 1D array samplecoords[].

```
with(model.result().dataset("cpt1"));
  set("pointx", samplecoords[0]);
  set("pointy", samplecoords[1]);
  set("pointz", samplecoords[2]);
endwith();
```

The code below sets the components of a deformation plot.

```
with(model.result("pg7").feature("surf1").feature("def"));
  setIndex("expr", withstru, 0);
  setIndex("expr", withstrv, 1);
  setIndex("expr", withstrw, 2);
endwith();
```

The code below sets the title and color legend of a plot group pg2.

```
with(model.result("pg2"));
  set("titletype", "auto");
endwith();
with(model.result("pg2").feature("surf1"));
  set("colorlegend", "on");
endwith();
model.result("pg2").run();
```

Using Parameterized Solutions in Results

The code below changes the visualization of a plot group pg2 by setting the property **looplevel** that controls the solution parameter to the string variable **svar**, as defined under the **Declarations** node.

```
with(model.result("pg1"));
  set("looplevel", new String[]{svar});
endwith();
model.result("pg1").run();
```

The property **looplevel** has a central role in accessing parameterized solutions. Its argument is a 1D string array with one index per “loop level” in a study. The different loop levels corresponds to the different nested parameters in a parametric sweep with multiple parameters.

Debug Methods

NAME	SYNTAX	DESCRIPTION
debugLog	debugLog(arg)	Prints the value of arg to the Debug Log window. The input argument arg can be a scalar, 1D array, or 2D array of the types string, double, int or Boolean.

EXAMPLE CODE

The code below prints strings and doubles to the Debug Log window.

```
xcoords[i] = Math.cos(2.0*Math.PI*divid);
ycoords[i] = Math.sin(2.0*Math.PI*divid);
debugLog("These are component values for case 1:");
debugLog("x:");
debugLog(xcoords[i]);
debugLog("y:");
debugLog(ycoords[i]);
```

Methods for External C Libraries

EXTERNAL METHOD

NAME	SYNTAX	DESCRIPTION
external	<code>External external(String name)</code>	Returns an interface to an external C (native) library given by the name of the library feature. The External class uses the Java Native Interface (JNI) framework.

METHODS RETURNED BY THE EXTERNAL METHOD

The external method returns an object of type External with the following methods:

NAME	SYNTAX	DESCRIPTION
invoke	<code>long invoke(String method, Object... arguments)</code>	Invokes the named native method in the library with the supplied arguments. Strings are converted to char *. Returns the value returned by the method. (Only 32 bits are returned on a 32-bit platform.)
invokeWideString	<code>long invokeWideString(String method, Object... arguments)</code>	Invokes the named native method in the library with the supplied arguments. Strings are converted to wchar_t *. Returns the value returned by the method. (Only 32 bits are returned on a 32-bit platform.)
close	<code>void close()</code>	Releases the library and frees resources. If you do not call this method, it is automatically invoked when the external library is no longer needed.

Progress Methods

NAME	SYNTAX	DESCRIPTION
<code>setProgressInterval</code>	<code>setProgressInterval(String message, int intervalStart, int intervalEnd)</code>	<p>Sets a progress interval to use for the top-level progress and display <code>message</code> at that level. The top level will go from <code>intervalStart</code> to <code>intervalEnd</code> as the second level goes from 0 to 100. As the second level increases, the top level is increased by $(intervalEnd - intervalStart) * (second\ level\ progress\ (0-100) / 100)$.</p> <p>The value for <code>intervalStart</code> must be between 0 and <code>intervalEnd</code>, and the value for <code>intervalEnd</code> must be between <code>intervalStart</code> and 100.</p> <p>Calling this method implicitly resets any manual progress previously set by calls to <code>setProgress()</code>.</p>
<code>setProgress</code>	<code>setProgress(int value, String message)</code>	<p>Sets a value for the user-controlled progress level. By default, this is the top level, but if a progress interval is active (<code>setProgressInterval</code> has been called and <code>resetProgress</code> has not been called after that), then it is the second level.</p>
<code>setProgress</code>	<code>setProgress(int value)</code>	<p>Same as <code>setProgress(message, value)</code>, but uses the latest message or an empty string (if no message has been set).</p>

NAME	SYNTAX	DESCRIPTION
resetProgress	<code>resetProgress()</code>	Removes all progress levels and resets progress to 0 and the message to an empty string.
showIndeterminateProgress	<code>showIndeterminateProgress(String message)</code>	Shows a progress dialog box with an indeterminate progress bar; given message and a cancel button.
showIndeterminateProgress	<code>showIndeterminateProgress(String message, boolean cancelButton)</code>	Shows a progress dialog box with an indeterminate progress bar; given message and an optional cancel button
showProgress	<code>showProgress()</code>	Shows a progress dialog box with a cancel button. No model progress is included.
showProgress	<code>showProgress(boolean modelProgress)</code>	Shows a progress dialog box with a cancel button and an optional model progress.
showProgress	<code>showProgress(boolean modelProgress, boolean addSecondLevel)</code>	Shows a progress dialog box with a cancel button, optional model progress, and one or two levels of progress information. Two levels can only be used if modelProgress is true.
showProgress	<code>showProgress(boolean modelProgress, boolean addSecondLevel, boolean cancelButton)</code>	Shows a progress dialog box with optional model progress, one or two levels, and possibly a cancel button. Two levels can only be used if modelProgress is true.
closeProgress	<code>closeProgress()</code>	Closes the currently shown progress dialog box.

NAME	SYNTAX	DESCRIPTION
startProgress	<code>startProgress(String name)</code>	Resets the value of the given progress bar form object name to 0 . The progress bar to control can be specified with an absolute path, such as form1/progressbar1 , or a name relative to the context from which the method was called. Nothing is done if no progress bar corresponding to the given name is found.
setProgressBar	<code>setProgressBar(String name, int workDone, String message)</code>	Sets the value of the given progress bar form object name in the range 0 - 100 and the associated progress message. Values out of range are converted to 0 or 100 . The progress bar to control can be specified with an absolute path, such as form1/progressbar1 , or a name relative to the context from which the method was called. Nothing is done if no progress bar corresponding to the given name is found, or if the progress bar is used for showing model progress.
setProgressBar	<code>setProgressBar(String name, int workDone)</code>	Same as above but does not update the progress message.

EXAMPLE CODE

```

showProgress(true, true, true);
/* Opens a progress dialog box with cancel button showing two levels of
progress. The values shown in progress dialog box will be updated to match
the two levels of progress. */

setProgressInterval("Preparing application", 0, 20);
/* Sets the current progress scale to go from 0 to 20. This means that the
top-level progress will go from 0 to 20 when second-level progress goes from
0 to 100. */

```

```

setProgress(0, "Init step 1");
/* Sets the second-level progress to 0 and the second-level progress message
to "Init step 1". */

// do some work

setProgress(40);
/* Sets the second-level progress to 40, this causes the top-level progress
to be updated to 8 (40 % of 0-20). */

// do some work

setProgress(80, "Init step 2");
/* Sets the second-level progress to 80 and the progress message to "Init
step 2". The top-level message is still "Preparing application" and
top-level progress is now 16. */

// do some work

setProgressInterval("Meshing", 20, 40);
/* Sets the top-level interval to 20 - 40 and the progress message to
"Meshing" at this point the value shown at the top-level will be 20. The
second-level progress is cleared when the top-level interval is changed. */

<call-meshing algorithm>
/* The progress messages and values from the meshing algorithm are shown at
the second-level progress. The top-level progress message will be "Meshing",
but the top-level progress advances from 20 to 40 while second-level
progress advances from 0 to 100. */

setProgressInterval("Solving", 40, 100);
/* The top-level progress message is changed to "Solving" and its value to
40.

<call-solver>
/* Similar to meshing, the progress messages and values from the solver are
shown in the second-level progress bar and the top-level progress value goes
from 40 to 100 while the solver progress goes from 0 to 100. */

closeProgress();

```

Application Progress Information

Progress information can be displayed in three different ways: in the Status bar, in a progress form object, and in a dialog box. Application progress information is controlled by the `setProgress` methods, which take as their input an integer between 0 and 100 and an optional message. The integer represents how far the displayed progress bar has progressed. If no message is supplied, the last message provided is used. For example:

```

setProgress(10, "Computing data")
setProgress(25)

```

This will keep Computing data as the progress message.

Use the `setProgress` method by itself if you want to display custom progress in the task and status bar. Once you have done this, that progress bar will no longer be updated by progress information from the COMSOL model, but will be completely dependent on further calls to `setProgress` for changes in its value. Precede it with a call to `showProgress` to also display the built-in progress dialog box, see below.

Note that progress information from the COMSOL model will not be shown in between calls to `setProgress`. Progress is reset between method calls. If you want to combine custom steps of progress in methods with built-in model progress, then use `setProgressInterval` instead.

With `setProgressInterval`, you can control the top two levels of progress information. The second level can be displayed in a progress dialog box and a progress bar form object, see the code segment below. The second progress level, controlled by your own custom progress calculation, is connected to the first level such that one interval at the top level corresponds to the entire second level. Thus if the interval is 0-50, when the second level progress reaches 40, for example, the first level will be set to 20 ($= (40/100) * 50$).

Important uses of the method `setProgressInterval` are listed below:

- Combining calls to the COMSOL model so that you get continuous progress going from 0-100.
- Computing several studies as well as evaluating several plots. Call `setProgressInterval` before each call to the built-in methods with an interval that approximates how much time each model computation takes.

For example:

```
setProgressInterval("Computing solution", 0, 80);
model.study("std1").run();
setProgressInterval("Plotting", 80, 100);
useGraphics(model.result("pg3"), "energy_response_plot/graphics1");
```

- Combining one or more calls to COMSOL model methods with custom methods that in themselves take significant time. In this case, use `setProgressInterval` as in the previous example, followed by your own custom code with appropriate calls to `setProgress`. These calls should run from 0 to 100 as they are controlling the second progress level. For example:

```
setProgressInterval("Computing solution", 0, 60);
model.study("std1").run();
setProgressInterval("Working", 60, 80);
setProgress(0, "Specific message about what I'm doing");
// ...
// Code that does something
// ...
setProgress(60);
```

If you, in a running application, wish to no longer use progress intervals, call `resetProgress` to return to the original state. This will also reset progress to 0.

The Progress Dialog Box

A progress dialog box can be used to display application progress as described in the previous section. The progress dialog has the following options:

- Whether to show model progress or not. When off, no progress from the model part of the application is forwarded to the progress dialog.
- Whether to show one or two progress levels in the progress dialog.
- Whether to include a cancel button. Cancel also works for user-defined methods, as it halts execution when the next line in the method is reached.

Use the `showProgress` methods to enable or disable these options. To close the progress dialog, use the `closeProgress` method.

You can show a progress dialog with an indeterminate progress bar that keeps spinning until you close the progress dialog. Only one progress dialog can be shown at a time. Use the `showIndeterminateProgress` methods to display this progress dialog.

The Progress Bar Form Object

The **Progress Bar** form object can either show overall application progress information or customized partial progress information. If you have selected the **Include model progress** check box in the **Settings** window of the **Main Window** node, then the overall application progress information becomes available.

When **Include model progress** is selected, the progress bar will show the same information as the progress dialog box. That is, one or two levels of progress information and a cancel button, depending on the settings in the form object.

When **Include model progress** is cleared, you control the progress bar through the `setProgressBar` methods. These take the path name of the progress bar form object, for example, `main/progressbar1`.

Date and Time Methods

NAME	SYNTAX	DESCRIPTION
currentDate	<code>String currentDate()</code>	Returns the current date as a string (formatted according to the server's defaults) for the current date.
currentTime	<code>String currentTime()</code>	Returns the current time as a string (not including date, and formatted according to the server's defaults).
formattedTime	<code>String formattedTime(long timeInMs, String format)</code>	Returns a formatted time using the given format. The format can either be a time unit or a text describing a longer format. Supported formats are: ‘hr:min:sec’ which returns the time in hours, minutes and seconds in the form X hr Y min Z sec. ‘h:min:s’ which returns the time in hours, minutes and seconds in the form X h Y min Z s. ‘detailed’ which returns the time in seconds and also includes more readable units for longer times.
sleep	<code>sleep(long timeInMs)</code>	Sleep for the specified number of milliseconds.
timeStamp	<code>long timeStamp()</code>	Current time in milliseconds since midnight, January 1, 1970 UTC.
getExpectedComputationTime	<code>model.setExpectedComputationTime(String format)</code>	Returns a string describing the approximate computation time of the application. The string can be altered by the method <code>setExpectedComputationTime</code> .

NAME	SYNTAX	DESCRIPTION
<code>setLastComputationTime</code>	<code>model.setLastComputationTime(long time)</code>	Set the last computation time, overwriting the automatically generated time. You can use the <code>timeStamp</code> method to record time differences and then set the measured time in ms (a long integer).
<code>getLastComputationTime</code>	<code>String model.getLastComputationTime(String format)</code>	Returns the last computation time in the given format. The format can either be a time unit or text describing a longer format. Currently supported formats are: hr:min:sec Returns the time in hours, minutes, and seconds in the format X h Y min Z sec. h:min:s Returns the time in hours, minutes, and seconds in the format X h Y min Z s. detailed Returns the time in seconds and also includes more readable units for longer times. This format is localized and the output is translated to the current language setting. For example, you can retrieve the time in ms by using <code>getLastComputationTime("ms")</code> .

EXAMPLE CODE

The following code overrides the built-in computation time that is available in the information nodes in the model tree.

```
long t0 = timeStamp(); // initialize record of computation time
// code and computations
model.setLastComputationTime(timeStamp()-t0); // record computation time
```

If it is possible to give a rough estimate of the computation time based on the given inputs of an application, you can update the expected computation time and display it in an information card stack or a text object. Assume that there is an integer input called `objects` that controls the number of objects in a geometry array and that the computation roughly increases linearly with this number. The following code adjusts the expected computation time accordingly.

```
// Number of minutes of computation time per object  
int minutes = objects*2.1;  
model.setExpectedComputationTime("About " + minutes + " minutes" );
```

For more information on information nodes and information cards, see “Information Card Stack” on page 210.

Sleep

The code below makes the application idle for 1000 ms.

```
long delay = 1000;  
sleep(delay);
```

This technique can be used to display graphics in a sequence, see “Graphics Commands” on page 61.

Appendix F—Guidelines for Building Applications

General

- Include reports to files with input data and corresponding output data.
- Make it intuitive. Provide help, hints, and documentation as necessary.
- Make it foolproof: “Safe I/O”, “Reset to default data”,...
- Save a thumbnail image with the model.
- Include a description text (it will be visible in the COMSOL Server library).
- Test the application on the computer platforms it is intended for.
- Be minimalistic. From the developer’s point of view it makes it much easier to make sure logic works, to organize, to debug, to maintain, and to develop further. From a user’s point of view it makes it easier to use the application. The minimalistic approach requires more care while developing but much less maintenance later on and much higher adoption among users.
- Embed libraries in the model if they are of manageable size.
- Display the expected computation time and, after the computation, the actual computation time.
- When a computation is canceled, output data from the previous computation should be cleared
- Password protect as needed. (Remember: No one can help you if you forget the password.)

Methods

- Don’t create more methods than necessary.

Fewer methods gives you a shorter list of methods to browse through when looking for something. Fewer methods usually means fewer lines of code to worry about.

- If several methods you wrote do essentially the same thing, consider merging them into one method and deal with the different cases by input arguments.
 - Don’t create a method if it is only called from one place. Insert the code right into that place instead.
- Create a local method if it is only used in a form, triggered by a form event or a form object event.
 - Give methods descriptive names, and name them so that similar methods are grouped together when sorted alphabetically. You will need to remember

less and you will find what you are looking for easier than. Long names are better than hard to understand short names.

Method naming examples:

- Start all methods that don't deliver any output by "p". (p for procedure)
- Start all methods that deliver output with "f". (f for function)
- Start all menu item methods with "m". (m for menu)
- Start a method that you will visit frequently with "a" to make it appear first in the list.
- Start all your plot methods with "Plot". mPlotMesh, mPlotResults, for example, for menu item methods.
- The points above apply to method code as well. Minimalistic, as few lines of code as possible, as few variables as possible. Give variables descriptive names. Long names are better than hard-to-understand short names. Optimize the code to run efficiently.
- The above points apply to declarations as well. Good names, not more than necessary. Declare variables where they are used (in forms and methods or in the model).

Forms

- Don't create more forms than necessary.
- Give forms descriptive names. Same reasoning as for methods.
- Make good use of the many different types of form objects. Some are good for some things, some are good for others.
- Don't insert more form objects than necessary. Too many options for input data may make the application hard to use. Too much output data makes it hard to find important information.
- Insert a text field for the user to leave comments to save with the user's set of input and output data when saving the application.
- Consider inserting a button with a method to reset to default data.
- Apply "Safe I/O":
 - For input fields, alert the user about input data that are out of bounds. You can do that either by an alert triggered by an On Data Change event for an input field, or by setting limits in the form objects settings window (which then sets hard limits). In a method generating the alert you may just warn the user and then allow the input data if the user chooses to go ahead anyways.
 - On output fields give the precision that makes sense. If current results are not based on current input data, show it. If the computation failed, show it.

- Include tooltips, help, documentation, hints, and comprehensive reports.
- Provide the user information about how long it takes to run the simulation with default input data on a typical computer. It could be seconds, it could be hours, or even days, depending on the application, so that's something the user would like to know before hitting the compute button. Consider playing a sound to alert the user when computation has finished. The user may be doing something else while waiting for results. (Sending an email message with a report to the user or some other place when the computation is done may be a better alternative if the computation is really long.)
- Spend some time on the layout of a form. A good looking form makes it easier and more fun to use the application.
- Consider setting keyboard shortcuts for buttons and menu items.

Appendix G—The Application Library Examples

In the Application Libraries, you can find example applications that showcase the capabilities of the Application Builder. They are collected in folders with the name Applications and are available for many of the add-on products. You can edit these applications and use them as a starting point or inspiration for your own application designs. Each application contains documentation (PDF) describing the application and an option for generating a report.

Below is a partial list of the available application examples ordered as they appear in the Application Libraries tree.

NAME	APPLICATION LIBRARY
Beam Subjected to Traveling Load	COMSOL Multiphysics
Helical Static Mixer	COMSOL Multiphysics
Transmission Line Calculator	COMSOL Multiphysics
Tubular Reactor	COMSOL Multiphysics
Tuning Fork	COMSOL Multiphysics
Induction Heating of a Billet	AC/DC Module
Touch Screen Simulator	AC/DC Module
Absorptive Muffler Designer	Acoustics Module
Acoustic Reflection Analyzer	Acoustics Module
Li-Ion Battery Impedance	Batteries & Fuel Cells Module ¹
Inkjet	CFD Module, Microfluidics Module
Water Treatment Basin	CFD Module
Biosensor Design	Chemical Reaction Engineering Module
Membrane Dialysis	Chemical Reaction Engineering Module
Concentric Tube Heat Exchanger	Heat Transfer Module ²
Heat Sink with Fins	Heat Transfer Module
Equivalent Properties of Periodic Microstructures	Heat Transfer Module
Red Blood Cell Separation	Microfluidics Module, Particle Tracing Module ³
Mixer	Mixer Module ⁴
Truck Mounted Crane Analyzer	Multibody Dynamics Module ⁵
Laminar Static Particle Mixer Designer	Particle Tracing Module
Gas Box	Pipe Flow Module
Distributed Bragg Reflector Filter	Ray Optics Module

NAME	APPLICATION LIBRARY
Frequency Selective Surface Simulator	RF Module
Microstrip Patch Antenna Array Synthesizer	RF Module
Wavelength Tunable LED	Semiconductor Module
Beam Section Calculator	Structural Mechanics Module ⁶
Truss Tower Buckling	Structural Mechanics Module
Fiber Simulator	Wave Optics Module
Plasmonic Wire Grating	RF Module, Wave Optics Module ⁷

¹Requires the Batteries & Fuel Cells Module and the Optimization Module

²Will run with either the CFD Module, Heat Transfer Module, Microfluidics Module, or Plasma Module

³In the Microfluidics Module version, this application requires the Microfluidics Module and the Particle Tracing Module. In the Particle Tracing Module version, this application requires the Particle Tracing Module and either one of the CFD Module, Microfluidics Module, or Subsurface Flow Module.

⁴Requires the CFD Module and the Mixer Module

⁵Requires the Structural Mechanics Module and the Multibody Dynamics Module

⁶An extended version of this application is available that also requires the LiveLink™ for Excel® product.

⁷In the RF Module version, this application requires the RF Module. In the Wave Optics Module, this application requires the Wave Optics Module.

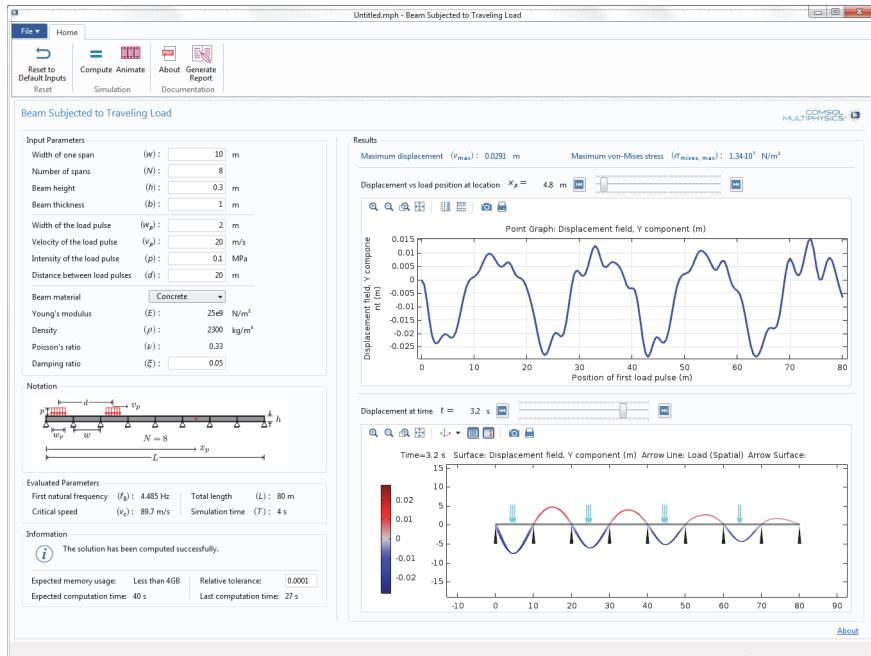
The following sections highlight the example applications listed in the table above. The highlighted applications exemplify a variety of important Application Builder features, including the use of animations, email, optimization, parameter estimation, tables, and import of experimental data.

Beam Subjected to Traveling Load

This application simulates the transient response of a beam that is placed on several equidistant supports and is subjected to a traveling load. The purpose of the application is to analyze the response of the bridge when vehicles pass over it. It is observed that for a bridge with given geometric and material properties, certain vehicular speeds cause resonance in the bridge and it undergoes high amplitude oscillation. In the application, a 2D plane stress approximation is assumed to model the beam. The beam is made of concrete.

The application demonstrates the use of animation and sliders. The first slider displays deformation versus load position and the second slider shows the time

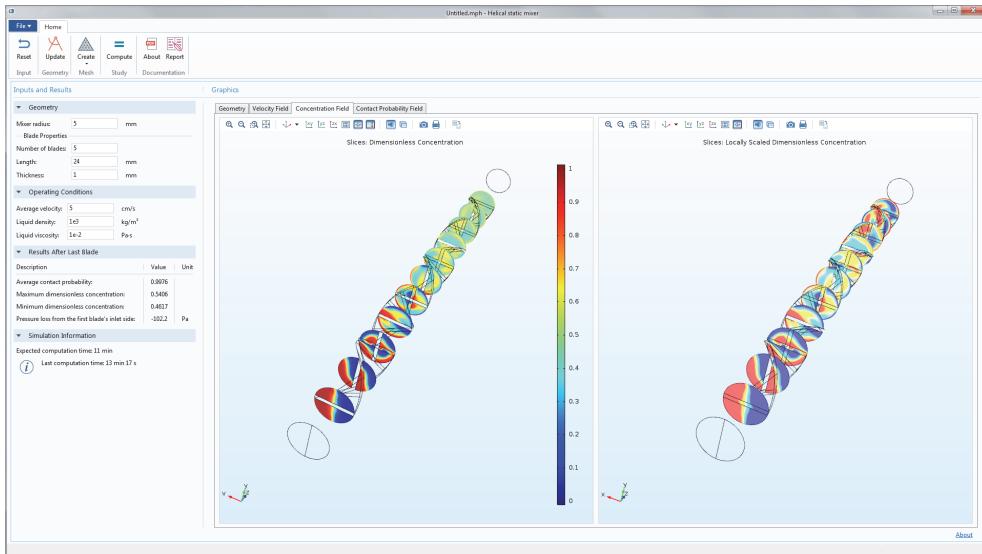
evolution of the displacement. This application does not require any add-on products.



Helical Static Mixer

The purpose of this application is to demonstrate the use of geometry parts and parameterized geometries. In addition, the application can be used to estimate the degree of mixing in a system including one to five helical blades, typically for mixing of monomers and initiators in polymerization reactions. The application is limited to Newtonian liquids, which is a good approximation in the cases where polymerization is negligible in the mixer itself. The application demonstrates the

use form collections of the type **Tiled or tabbed**. This application does not require any add-on products.



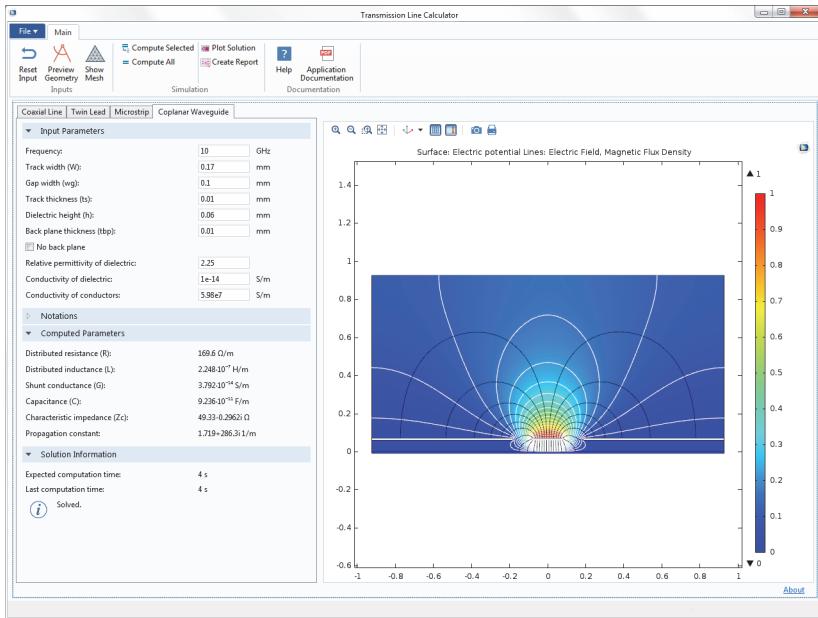
Transmission Line Calculator

Transmission lines are used to guide waves of alternating current and voltage at radio frequencies. Transmission lines exist in a variety of forms, many of which are adapted for easy fabrication and employment in printed circuit board designs. They are key elements in most modern electronic devices and are used to carry information, at minimal loss and distortion, from one place to another within a device and between devices.

This application provides predefined user interfaces for computing the transmission line parameters R , L , G , and C , as well as γ and Z_0 for parameterized cross sections of some common transmission line types:

- Coaxial line
- Twin lead
- Microstrip line
- Coplanar waveguide (CPW)

Plots of the geometry, mesh, electric potential, electric field line, and magnetic flux lines are also provided. This application does not require any add-on products.



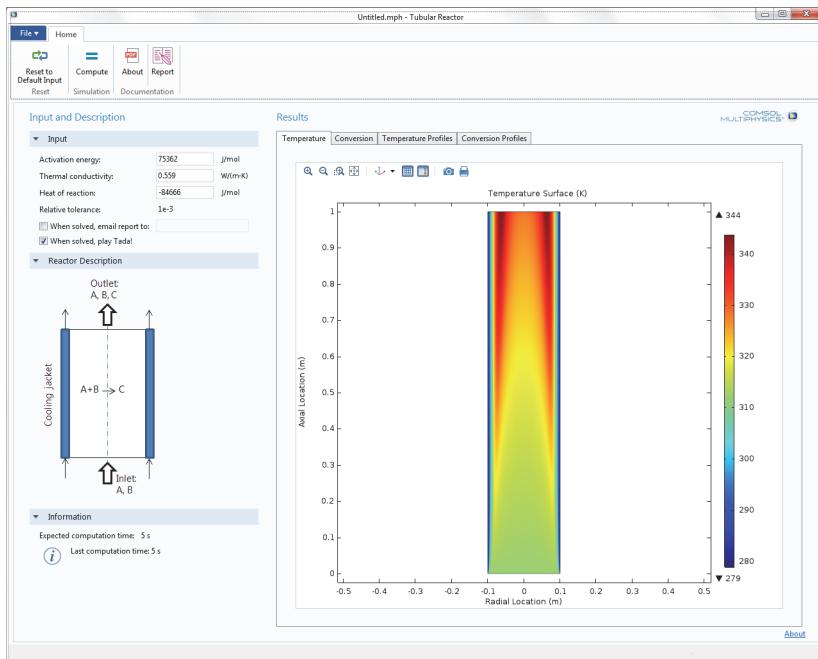
Tubular Reactor

With this application, students in chemical engineering can model a nonideal tubular reactor, including radial and axial variations in temperature and composition, and investigate the impact of different operating conditions. The process described by the application is the exothermic reaction of propylene oxide with water to form propylene glycol.

The application also exemplifies how teachers can build tailored interfaces for problems that challenge the students' imaginations. The model and exercise are originally described in Scott Fogler's *Elements of Chemical Reaction Engineering*.

The mathematical model consists of an energy balance and a material balance described in an axisymmetric coordinate system. The students can change the activation energy of the reaction, the thermal conductivity, and the heat of reaction in the reactor. The resulting solution gives the axial and radial conversion as well as temperature profiles in the reactor. For some data, the results from the simulation are not obvious, which means that the interpretation of the model results also becomes a problem-solving exercise.

Note that you may also have the application send an email when the computation is ready by selecting the email check box and entering an email address. This sends a report with the settings and the computed results. The functionality can be used by students to send the results to a supervisor. For computations that take a longer time to compute, this functionality may be of great use. For example, you can start a simulation and leave the office, or laboratory, and then get the full report from the application when the computation is done, which you can access on the road or wherever you have access to your mail. This application does not require any add-on products.

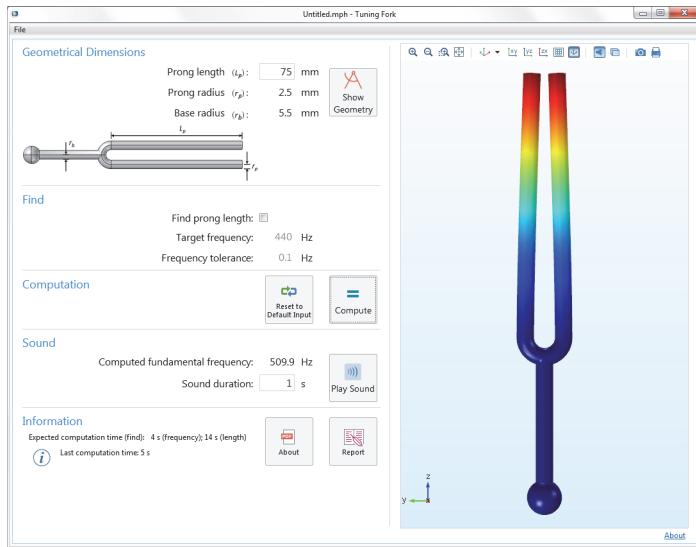


Tuning Fork

This application computes the resonant frequency of a tuning fork with a user-defined prong length. Alternatively, you can instead give a user-defined target frequency and the application will find the corresponding prong length. The prong and handle radii are taken from a commercially available tuning fork. The model embedded in the application is defined using the Solid Mechanics interface included in COMSOL Multiphysics and does not require any add-on products. The prong length search algorithm is a secant method.

At the end of the computation, the built-in method `playSound` is used to produce a sine wave sound at the computed frequency.

For more background theory, see the Application Library documentation for the model `tuning_fork.mph`.

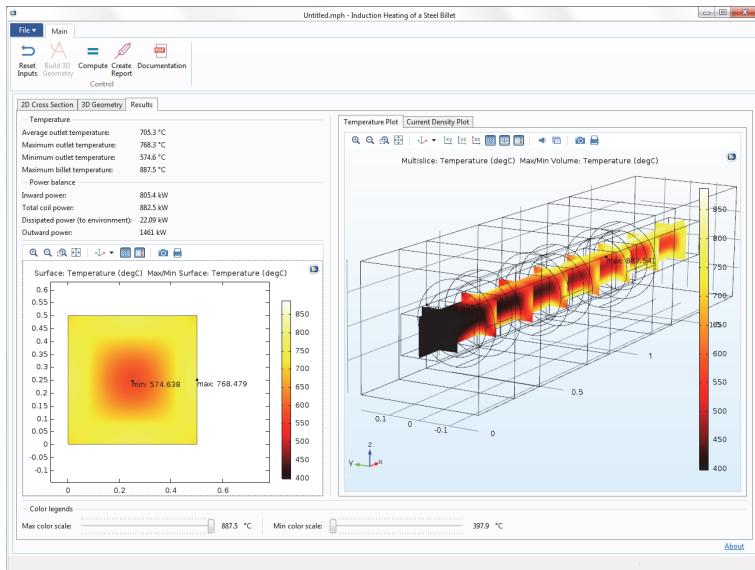


Induction Heating of a Steel Billet

This application can be used to design a simple induction heating system for a steel billet, consisting of one or more electromagnetic coils through which the billet is moved at a constant velocity. The coils are energized with alternating currents and induce eddy currents in the metallic billet, generating heat due to Joule heating.

The billet cross section, the coil number, placement and size, as well as the initial and ambient temperature and the individual coil currents can all be specified as inputs. After the solution has been computed, the application displays 3D plots of the billet temperature during processing, the induced electric current density, and a 2D plot of the temperature at the outlet cross section. Finally, the application

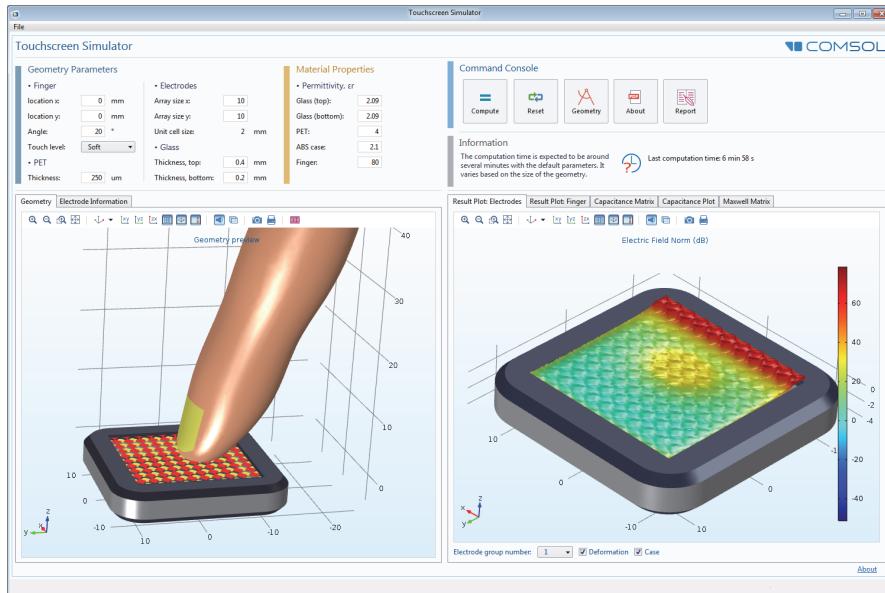
computes numerical data for the expected temperature ranges in the billet and the power balance of the system.



Touch Screen Simulator

This application computes the capacitive response of a small touchscreen in the presence of a human finger phantom. This information can be used by an electronic circuit to derive the position of the finger. In the application, the

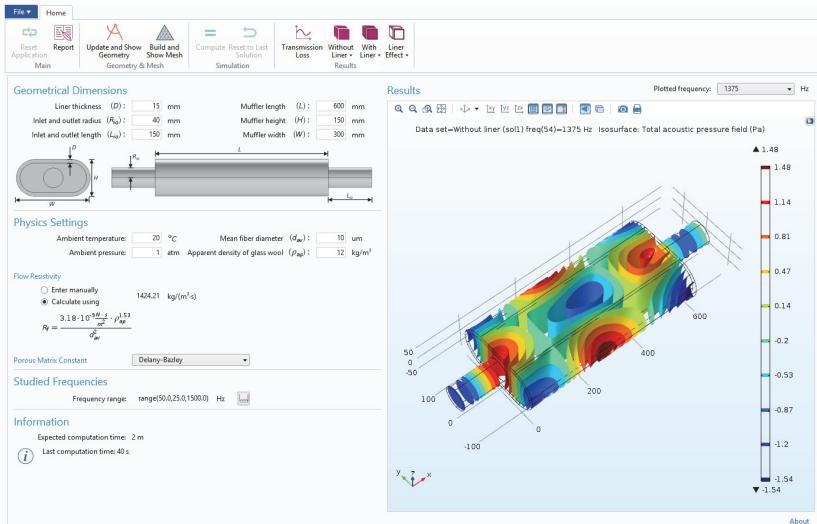
position and orientation of the finger are controlled via input parameters, and the resulting capacitance matrix is computed as output.



Absorptive Muffler Designer

The purpose of this application is for studying and designing a simple resonant muffler with a porous lining. Mufflers are used to attenuate noise emitted by, for example, a combustion engine or an HVAC system and should typically perform well in a specific frequency range. The measure of the attenuation is called the transmission loss (TL) and gives the damping in dB as a function of frequency. The transmission loss depends on the geometry of the muffler and on the characteristics of porous and fibrous materials that can be placed in the system. This application is used to study the results of modifying the dimensions of a muffler, the ambient working conditions, as well as the material properties of the porous liner. That is, how changes influence the transmission loss of the system. This application is an example of a “dynamic specification sheet” for a given muffler model. A sales engineer can bring this type of application to customers and show them the performance of a custom muffler designed specifically for them. A muffler may, for example, be designed to be placed in a vehicle with spatial

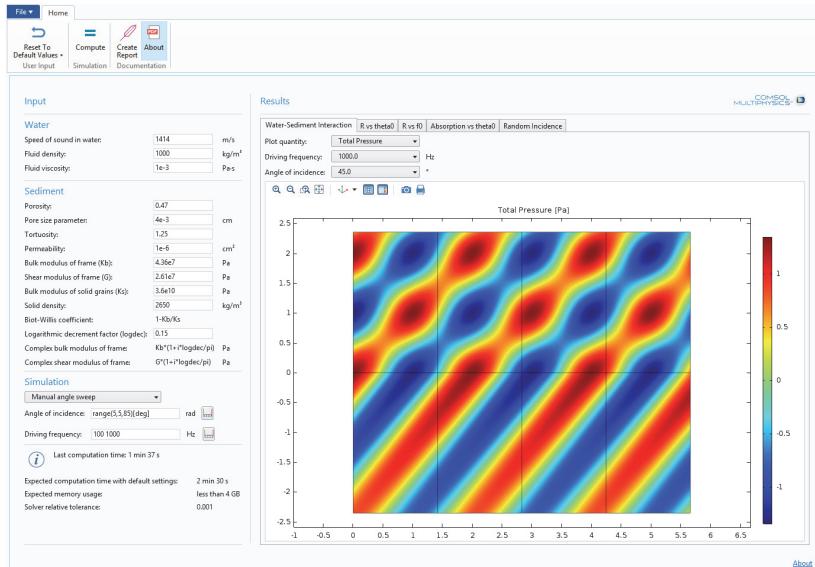
constraints. In this case, the performance can be readily visualized and different options can be investigated together with the customer.



Acoustic Reflection Analyzer

This application analyzes the reflections of plane waves off a water-sediment interface. The reflection and absorption coefficients are determined as functions of the angle of incidence and the frequency. Moreover, the random-incidence absorption coefficient, or diffuse field absorption coefficient, is calculated based on the simulated data. The material properties of the fluid, in this case water, and the

properties of the porous medium, here a semi-infinite sediment layer, can be modified.

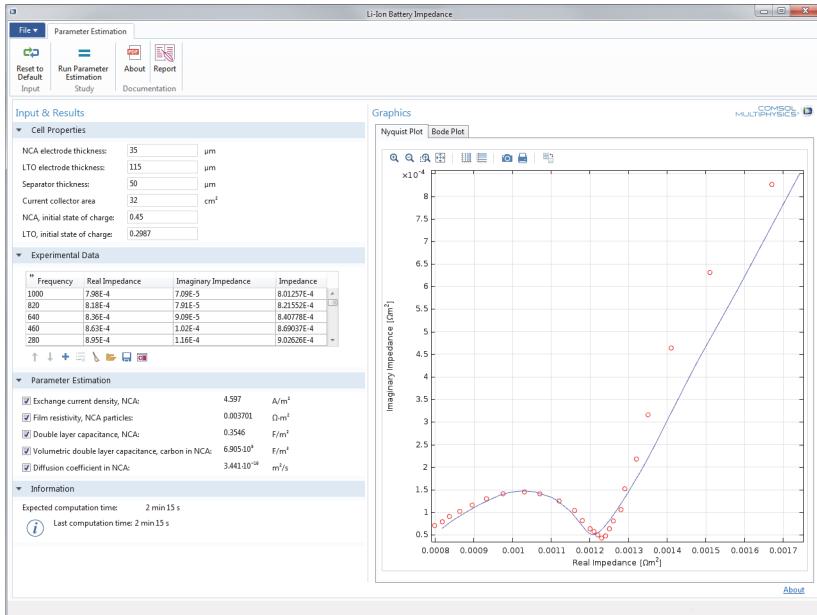


Li-Ion Battery Impedance

The goal with this application is to explain experimental electrochemical impedance spectroscopy measurements (EIS) and to show how you can use the model and the measurements to estimate the properties of lithium-ion batteries. The application takes experimental data from EIS measurements as input, simulates these measurements, and then runs a parameter estimation based on the experimental data.

The control parameters are the exchange current density, the resistivity of the resistive layer on the particles, the double-layer capacitance of NCA, and the double-layer capacitance of the carbon support in the positive electrode. The fitting is done to the measured impedance of the positive electrode at frequencies ranging from 10 mHz to 1 kHz.

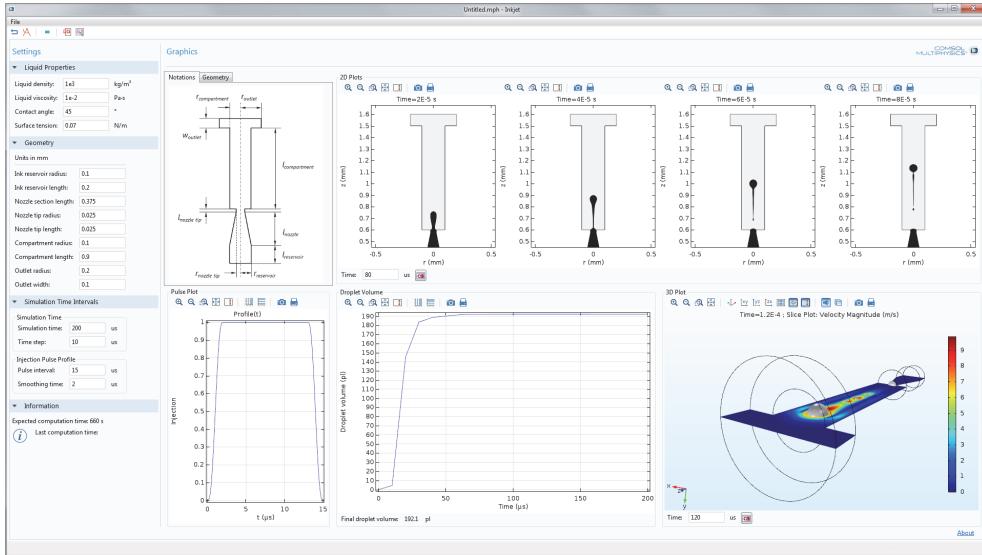
The application demonstrates loading experimental data on the comma-separated values (CSV) file format and utilizes the Optimization Module for parameter estimation.



Inkjet

The purpose of this application is to adapt the shape and operation of an inkjet nozzle for a desired droplet size, which depends on the contact angle, surface tension, viscosity, and density of the injected liquid. The results also reveal

whether the injected volume breaks up into several droplets before merging into a final droplet at the substrate.

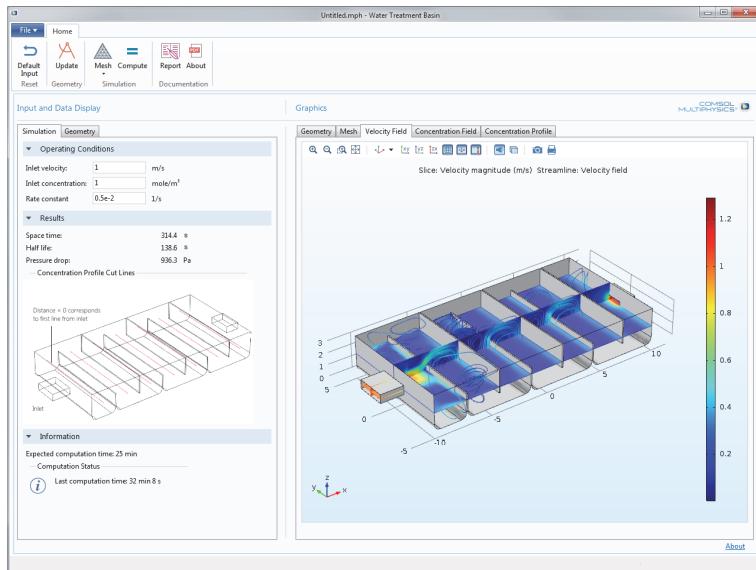


Water Treatment Basin

The purpose of the Water Treatment Basin application is to exemplify the use of applications for modeling turbulent flow in liquids in 3D. An interesting aspect of the application is that it also accounts for the material balance of a solute in a solution. This solute reacts in a first-order reaction, a common type of reaction for describing the decay of highly diluted chemical species. This application also shows how to use fully parameterized geometries and cumulative selections for modeling turbulent flows.

The application can be used as a starting point for your own application for modeling turbulent steady flow of liquids with reactions of highly diluted solutes.

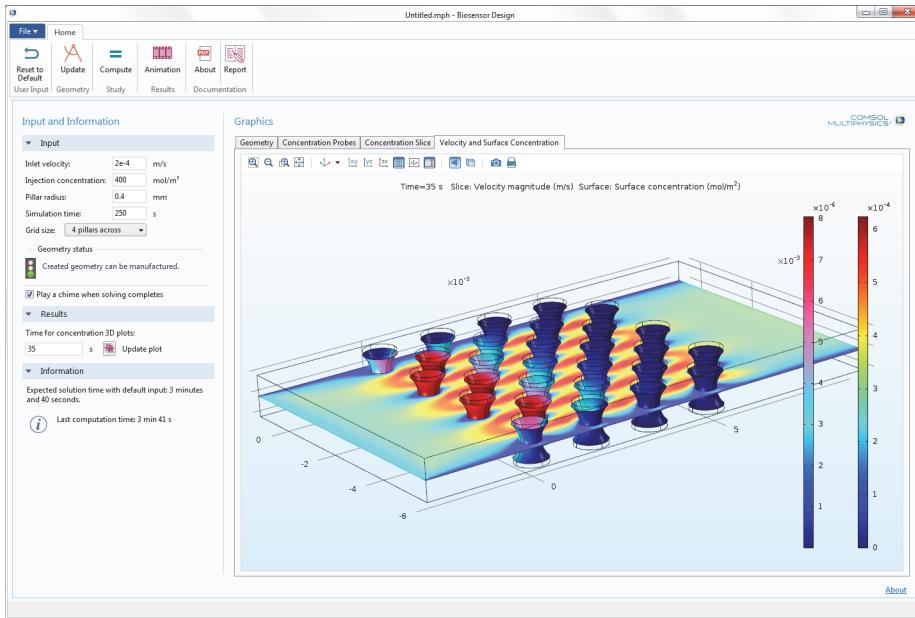
The exemplified system is a chlorination basin in a water treatment process.



Biosensor Design

A flow cell in a biosensor contains an array of micropillars used to detect biomolecules. The pillars are coated with an active material that selectively adsorbs biomolecules in the sample stream. These biomolecules then react on the surface. This application allows the user to change the design of the sensor by altering parameters such as pillar diameter, grid spacing, and inlet velocity to investigate how the design affects the detection results. The geometry and operating conditions have a great impact on the signal strength and diffuseness. Also,

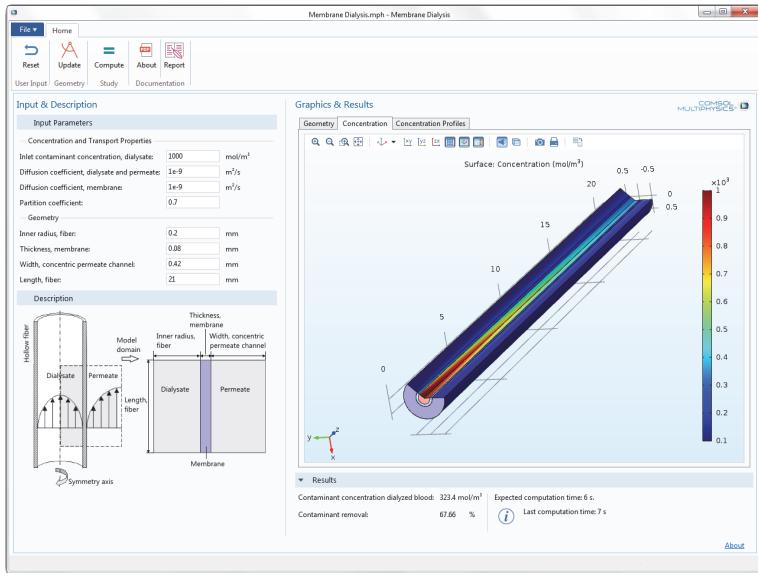
manufacturing constraints, set by a minimum distance between pillars, are reported in the application.



Membrane Dialysis

This application simulates the contaminant concentration within a bloodstream that is purified within a membrane dialysis device. The modeled dialysis device is made of a hollow fiber module, where the walls of the hollow fibers act as a membrane for removal of the contaminant. Within the inside of the fibers, the dialysate flows, whereas on the outside, the permeate passes. Through variation of

the input parameters, the application can examine approaches on how to maximize the contaminant removal within the device.



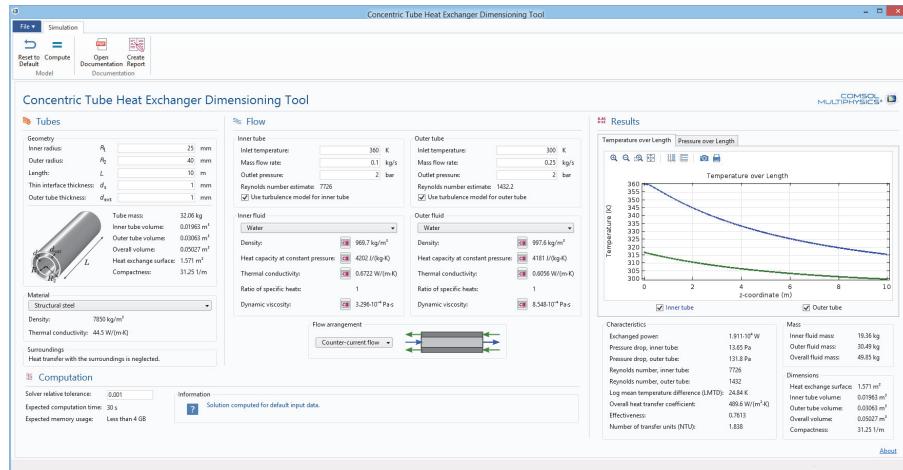
Concentric Tube Heat Exchanger

Dimensioning quantities are the first indicators of the behavior of a heat exchanger. This application aims at computing these quantities for a given configuration.

The example application studies the case of two concentric tubes separating two distinct fluids. The fluids can run either in counterflow or in parallel flow. Both tubes and fluids can be customized through the user interface.

After the computation, the temperature profile and several quantities are displayed. Additional inputs are fluid properties such as available volume and mass,

compactness (the ratio of exchanged surface to heat exchanger volume), and material properties.

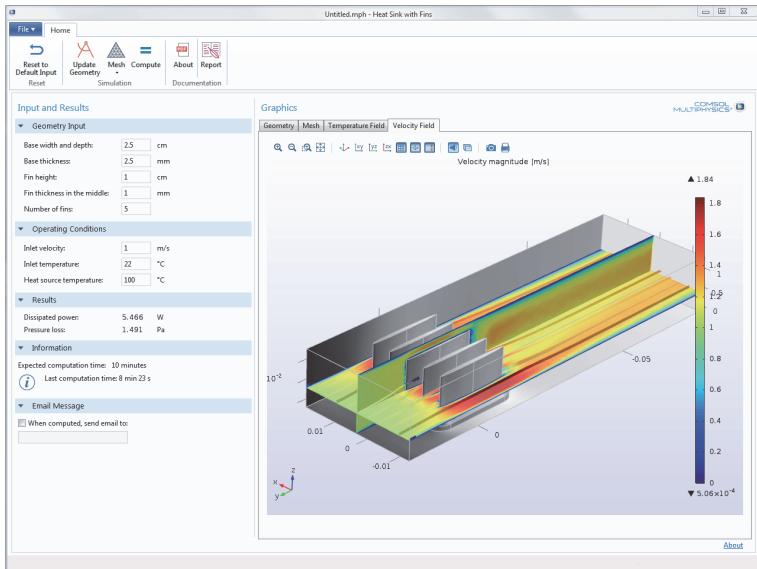


Heat Sink with Fins

Heat sinks are usually benchmarked with respect to their ability to dissipate heat for a given fan curve. One possible way to carry out this type of experiment is to place the heat sink in a rectangular channel with insulated walls. The temperature and pressure at the channel's inlet and outlet, as well as the power required to keep the heat sink base at a given temperature, is then measured. Under these conditions, it is possible to estimate the amount of heat dissipated by the heat sink and the pressure loss over the channel.

The purpose of this application is to carry out investigations of benchmark experiments using modeling and simulation. For example, the amount of heat dissipated may increase with the number of fins until the fins create such a large obstruction to the flow that the flow decreases and lowers the amount of heat dissipated. This implies that for a given total pressure loss over the channel, there

may be optimal dimensions and a number of fins that give the highest cooling power. This application allows you to perform such investigations.



Equivalent Properties of Periodic Microstructures

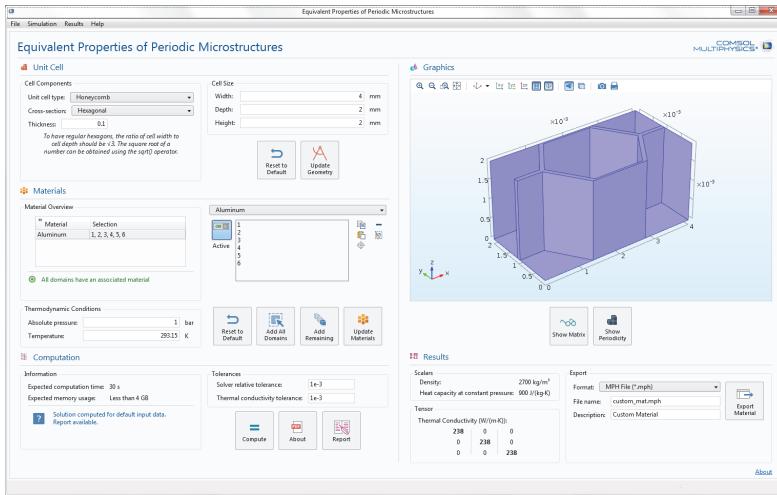
Periodic microstructures are frequently found in composite materials, such as carbon fibers and honeycomb structures. They can be represented by a unit cell repeated along three directions of propagation. To reduce computational costs, simulations may replace all the details of a composite material with a homogeneous domain with equivalent properties. This application computes equivalent properties from the geometrical configuration and the material properties of a unit cell. It offers a choice between nine parameterizable unit cells and a list of 13 predefined materials. Extending this application to additional parallelepiped unit cells or adding other materials is straightforward.

In diffusion-like equations such as the heat equation, the equivalent diffusion coefficient takes the general form of a tensor.

In this application, the following material properties are computed from a selected unit cell shape with given materials in the different regions of the unit cell:

- Density
- Heat capacity at constant pressure
- Thermal conductivity

The built-in unit cell library in this application includes several widely used cell types such as parallel stacked layers, fiber-reinforced composites, or honeycomb structures. Once the geometry is set, the physics consists of periodic heat conditions at opposite boundaries of the cell.

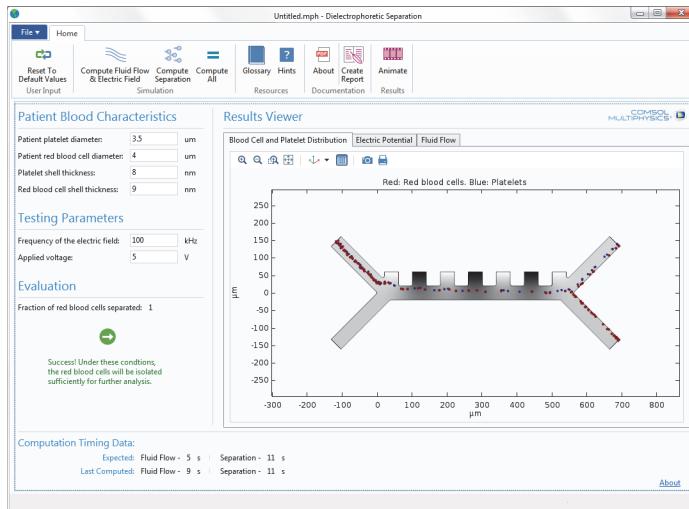


Red Blood Cell Separation

Dielectrophoresis (DEP) is a phenomenon in which a force is exerted on a dielectric particle when it is subjected to a nonuniform electric field. The electric field induces a polarization in the particles, which are then subject to a DEP force that is proportional to the gradient of the electric potential.

The DEP force is sensitive to the size, shape, and dielectric properties of the particles. This allows DEP to be used to separate different kinds of particles. One application of this process is in the field of bioengineering, where DEP can be used to separate different kinds of cells from a mixture. This example application shows how red blood cells can be selectively filtered from a blood sample in order to isolate red blood cells from platelets. This is useful as platelets cause blood to

clot, which can lead samples contaminated with platelets to be unsuitable for subsequent testing once a clot has formed.

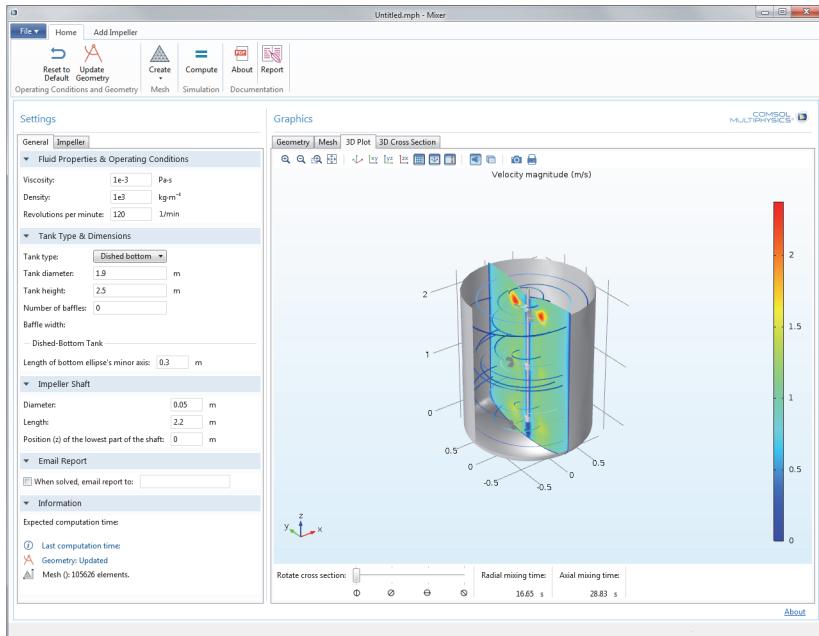


Mixer

The purpose of the Mixer application is to provide a user-friendly interface where scientists, process designers, and process engineers can investigate the influence that vessel, impeller, and operational conditions have on the mixing efficiency and the power required to drive the impellers. The application can be used to understand and optimize the design and operation of the mixer for a given fluid. But, perhaps most importantly, it can be used as a starting point for your own application for the modeling and simulation of mixers and reactors.

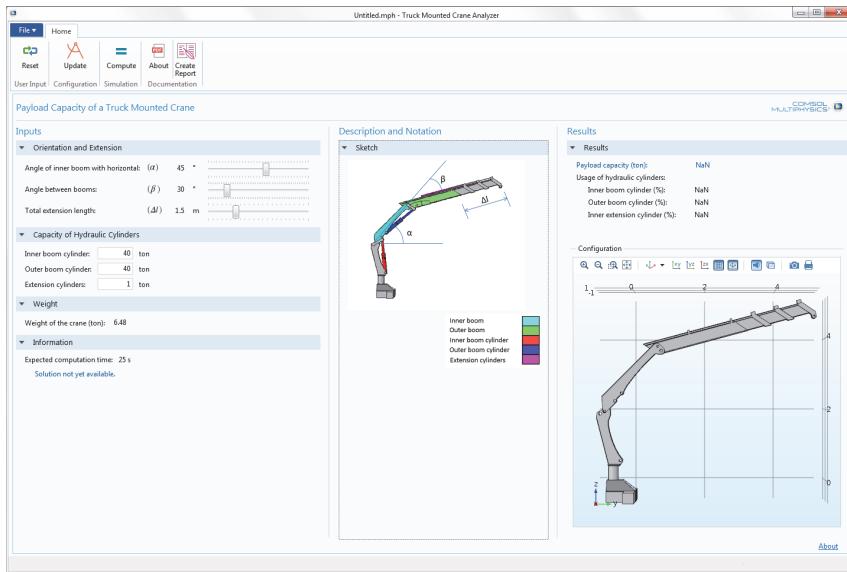
The application demonstrates how parts and cumulative selections can be used to automatically set domain and boundary settings in the embedded models. These

settings can be created automatically, even when the choices an application user makes create very diverse geometries.



Truck Mounted Crane Analyzer

Many trucks are equipped with cranes for load handling. Such cranes have a number of hydraulic cylinders that control the crane's motion, and several mechanisms.

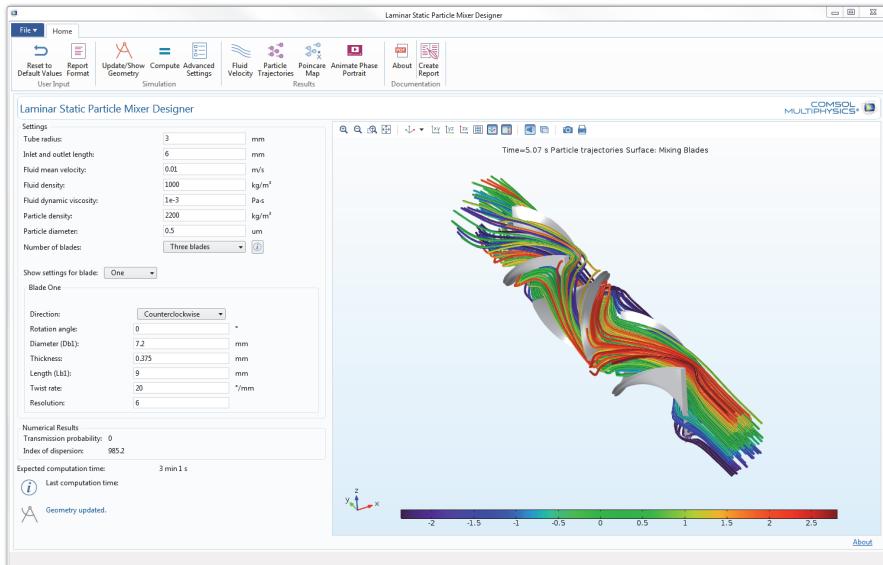


In this application, a rigid-body analysis of a crane is performed in order to find the payload capacity for the specified orientation and extension of the crane. This application also provides the usage of hydraulic cylinders and highlights the limiting cylinder. The capacity of the hydraulic cylinders can be modified in order to improve the payload capacity and the usage of the cylinders.

Laminar Static Particle Mixer Designer

In static mixers, also called motionless or in-line mixers, a fluid is pumped through a pipe containing stationary blades. This mixing technique is particularly well suited for laminar flow mixing because it generates only small pressure losses in this flow regime. This application studies the flow in a twisted-blade static mixer. It evaluates the mixing performance by calculating the trajectory of suspended particles through the mixer. The application computes the static mixing of one species dissolved in a solvent at room temperature. You can study the effect of fluid

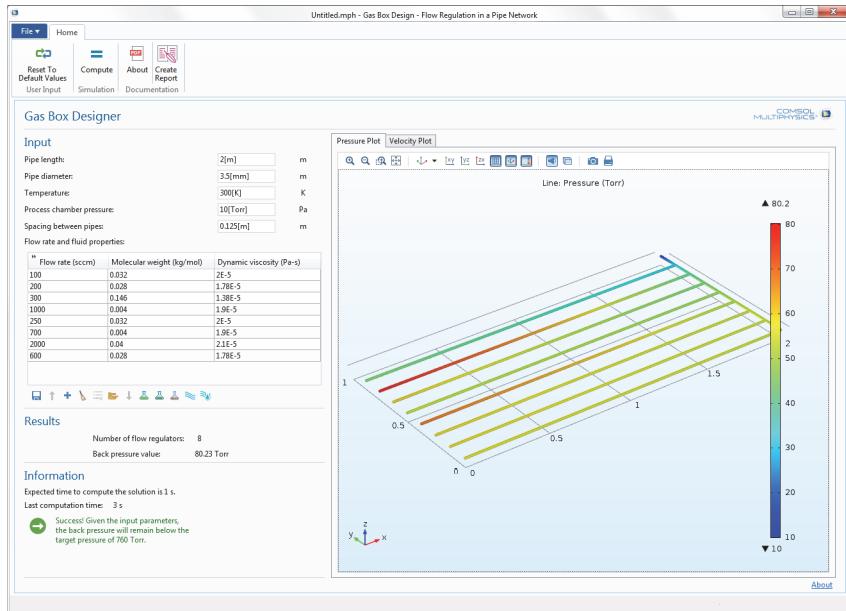
and particle properties as well as the stationary blades' configuration on the particle mixing.



Gas Box

A gas box, used in the semiconductor processing industry, consists of a bank of (usually 12) mass flow controllers (MFC) that regulate the flow of chemicals into a process chamber. In order for the MFCs to function correctly, the pressure immediately downstream needs to be below a certain threshold, typically atmospheric pressure. This is needed to maintain a choked flow condition across the flow controller. There is typically a network of piping that connects the flow controllers to the process chamber. If the flow rate through the controllers is high, the pipe thickness is very small, or if the process chamber is high, the back pressure in the pipe network exceeds atmospheric. The application computes the pressure distribution inside a network of pipes using the Pipe Flow Module, and indicates whether the back pressure will exceed the target threshold. This application

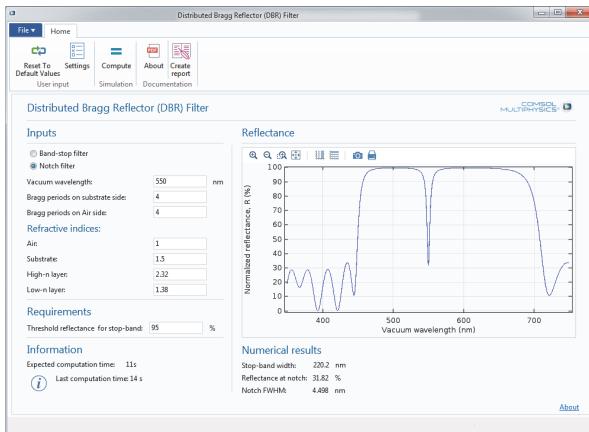
demonstrates using a table object for user input with an accompanying customized toolbar.



Distributed Bragg Reflector Filter

A distributed Bragg reflector (DBR) consists of alternating layers of two materials. Each material has a different refractive index, resulting in a repeating pattern of high and low refractive index in the direction perpendicular to the DBR layers. As light propagates through this structure, reflections occur at each interface between the layers. Interference effects between the multiple reflected waves cause the reflectivity of the DBR to be highly wavelength dependent. The main advantage of DBRs over ordinary metallic mirrors is that DBRs can be engineered to have custom reflectances at selected wavelengths.

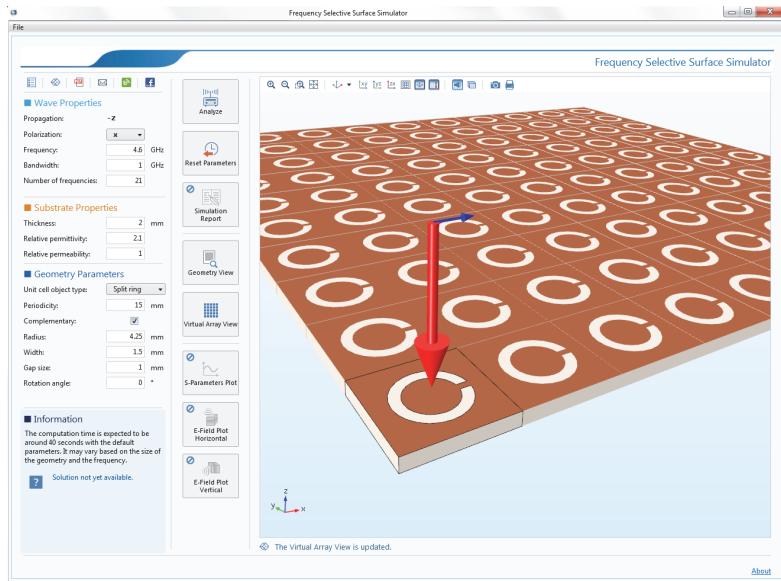
This application can be used to study the reflectance from a simple optical notch filter, based on a cavity sandwiched between two distributed Bragg reflectors.



Frequency Selective Surface Simulator

Frequency selective surfaces (FSS) are periodic structures that generate a bandpass or a bandstop frequency response. This application simulates a user-specified periodic structure chosen from the built-in unit cell types. It provides five popular FSS unit cell types, with two predefined polarizations and propagation at normal incidence. The analysis includes the reflection and transmission spectra, the

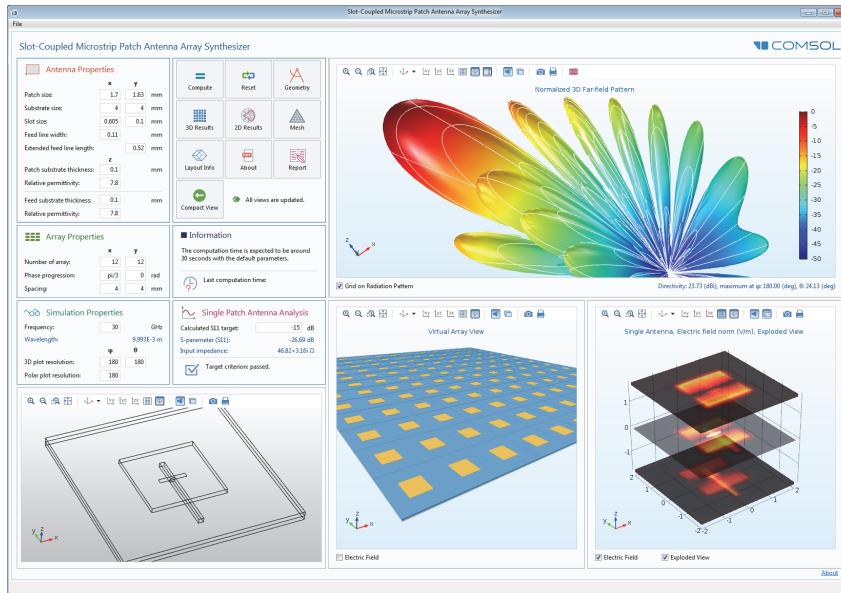
electric field norm on the top surface of the unit cell, and the dB-scaled electric field norm shown on a vertical cut plane in the unit cell domain.



Microstrip Patch Antenna Array Synthesizer

This application simulates a single slot-coupled microstrip patch antenna that is fabricated on a multilayered low temperature co-fired ceramic (LTCC) substrate. Results include the far-field radiation pattern of the antenna array and its directivity. The far-field radiation pattern is approximated by multiplying the array factor and the single antenna radiation pattern to perform an efficient far-field analysis without simulating a complicated full array model. Phased antenna array prototypes for 5G mobile networks can easily be evaluated with the default input

frequency, 30 GHz. The application also demonstrates an animation where the camera is moved around the antenna.

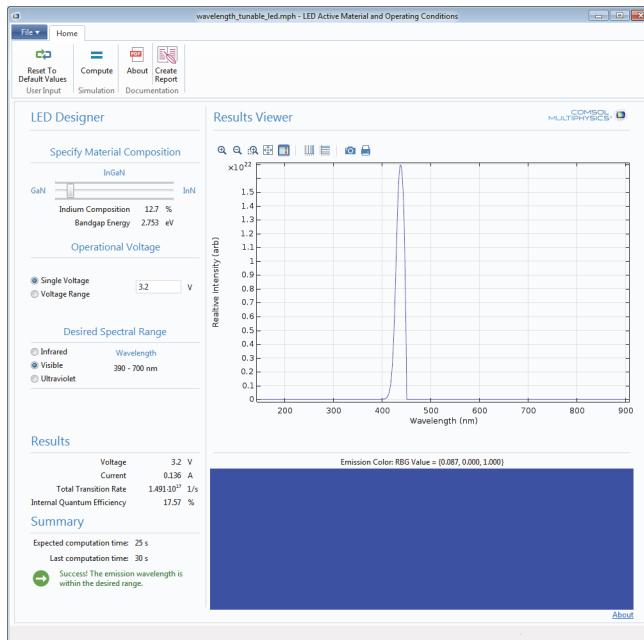


Wavelength Tunable LED

Blue LEDs are interesting because of their use in modern high-efficiency lighting. Due to their large bandgap energy, gallium nitrides are widely used for generating blue light. This application simulates the emission properties of a gallium-nitride-based light-emitting diode. The material used in the active region of the device is $\text{In}_x\text{Ga}_{1-x}\text{N}$, which contains a blend of both gallium and indium where the fraction of indium is given by x . The bandgap of this optically active region can be controlled by varying the composition of the material via changing the indium fraction. Because pure InN and GaN emit in the infrared and ultraviolet parts of the spectral range respectively, it is possible to tune the emission energy of $\text{In}_x\text{Ga}_{1-x}\text{N}$ across the entire visible spectrum using this technique.

This application enables the indium fraction and operating voltage of the device to be controlled. The current, emission intensity, electroluminescence spectrum, and internal quantum efficiency of the device can then be computed. Either a single operating voltage or a range of voltages can be input. If a range of voltages

is input, the current-voltage curve is also calculated, which allows the turn-on voltage of the device to be determined. This application uses methods extensively.

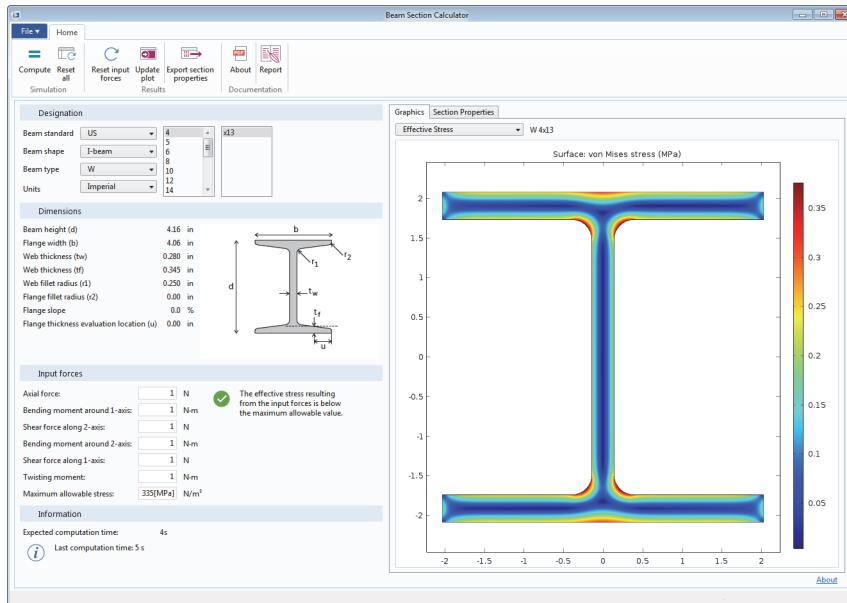


Beam Section Calculator

This application computes the beam section properties for a designated steel beam section. It also allows for computing the detailed stress distribution over the cross section given a set of forces and moments acting on it. A broad range of American and European standard beams are available.

With a license for the LiveLink™ for Excel® product, all input and results data is displayed in a table that can be exported to an Excel® file. It is possible to edit the

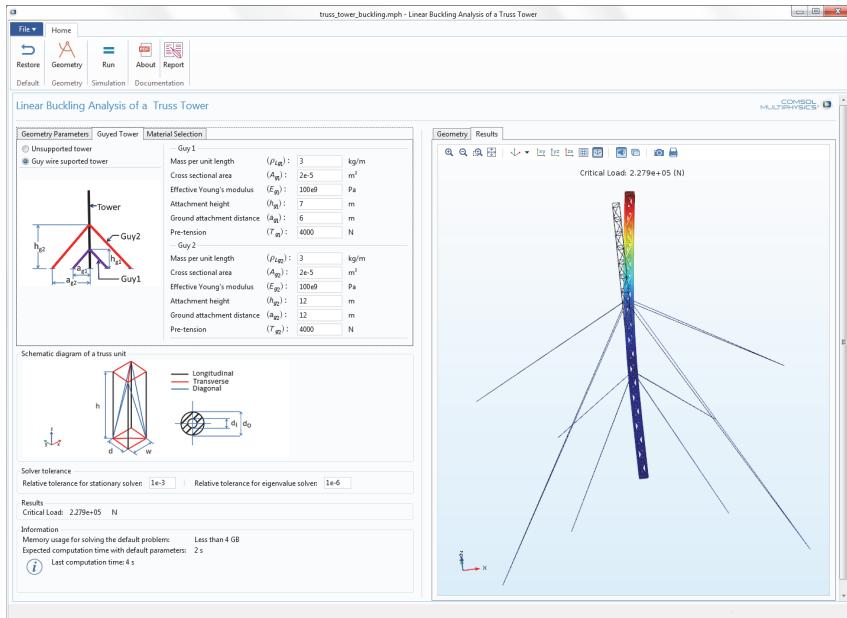
Excel® workbook that contains the beam dimensions data and re-import this data back into the application.



Truss Tower Buckling

Buckling analysis is the search for the critical compressive load beyond which structures become unstable. This application can simulate the buckling of a truss tower under vertical compressive loads. The tower can optionally be supported by guy wires. The purpose of the application is to compute and analyze the buckling load for towers under different conditions of geometry, i.e. various tower heights, cross-sectional area, as well as different materials. The application takes into

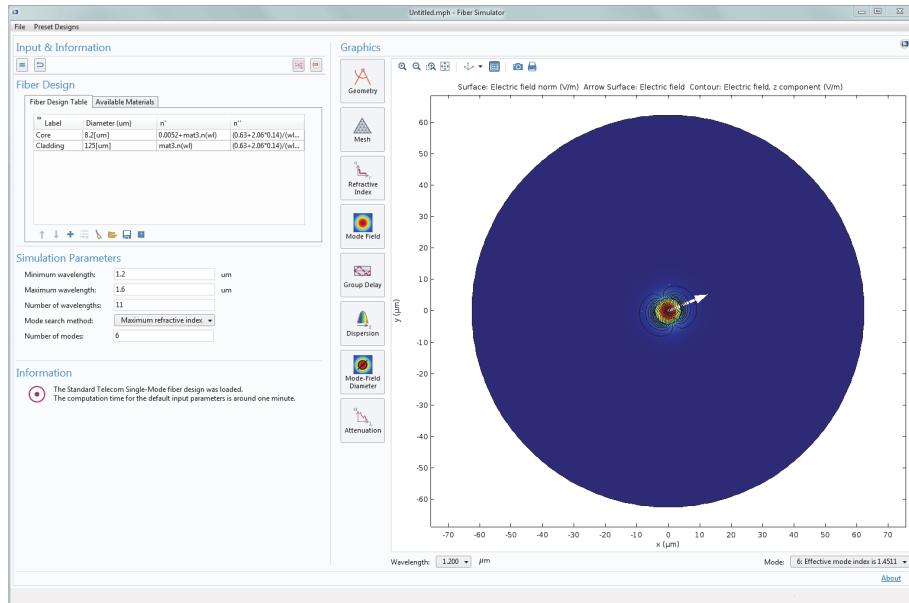
account the effect of dead load (self-weight of truss and supporting guy wires and their pretension) while performing the computation.



Fiber Simulator

This application performs mode analyses on concentric circular dielectric layer structures. Each layer is described by an outer diameter and the real and imaginary parts of the refractive index. The refractive index expressions can include a dependence on both wavelength and radial distance. Thus, the simulator can be used for analyzing both step-index fibers and graded-index fibers. These fibers can

have an arbitrary number of concentric circular layers. Computed results include group delay and dispersion coefficient.



Plasmonic Wire Grating

This application computes diffraction efficiencies for the transmitted and reflected waves ($m = 0$) and the first and second diffraction orders ($m = \pm 1$ and ± 2) as functions of the angle of incidence for a wire grating on a dielectric substrate. The incident angle of a plane wave is swept from normal incidence to grazing

incidence. The application also shows the electric field norm plot for multiple grating periods for a selected angle of incidence.



Index

- 1D array 121
- 2D array 123
- 3D coordinates 215
- A**
 - action 46, 49, 109
 - activation condition 177, 184
 - active card selector 204
 - add-on products 7, 324
 - alert 156, 293
 - aligning form objects 86, 95
 - animation 68, 325
 - appearance
 - forms 37, 39
 - graphics object 60
 - input field object 75
 - multiple form objects 44
 - table 230
 - text 43
 - append unit to number 72, 235
 - application
 - launching 28
 - saving 242
 - Application Builder 16
 - desktop environment 9, 14
 - window 9, 11, 13
 - application example
 - absorptive muffler designer 332
 - acoustic reflection analyzer 333
 - beam section calculator 351
 - beam subjected to traveling load 325
 - biosensor design 337
 - concentric tube heat exchanger 339
 - distributed bragg reflector filter 347
 - equivalent properties of periodic microstructures 341
 - fiber simulator 353
 - frequency selective surface simulator 348
 - gas box 346
 - heat sink with fins 340
 - helical static mixer 326
 - induction heating of a steel billet 330
 - inkjet 335
 - laminar static particle mixer designer 345
 - Li-ion battery impedance 334
 - membrane dialysis 338
 - microstrip patch antenna array synthesizer 349
 - mixer 343
 - plasmonic wire grating 354
 - red blood cell separation 342
 - touch screen simulator 331
 - transmission line calculator 327
 - truck mounted crane analyzer 345
 - truss tower buckling 352
 - tubular reactor 290, 328
 - tuning fork 329
 - water treatment basin 336
 - wavelength tunable LED 350
- Application Gallery 24
- Application Libraries 7, 23, 24, 32, 324
- Application Library
 - COMSOL Server 28
 - application tree 9, 11, 13
 - applications folder 7, 24, 324
 - apply changes 19
 - arranging form objects 86
 - array 121
 - 1D 121
 - 2D 123
 - methods 267

- syntax 123
- array input object 214
- B**
 - background color 39
 - background image 39
 - BMP file 160
 - Boolean variable 118, 120, 121, 122, 168
 - conversion 265
 - breakpoint 151
 - built-in method library 264
 - button 48, 59, 85, 146
 - command sequence 49
 - icon 48
 - keyboard shortcut 48
 - on click event 48, 49
 - picture 48
 - size 48
 - tooltip 48
 - buttons tab, New Form wizard 35
- C**
 - C libraries
 - external 311
 - CAD-file import 207, 244, 246
 - card 203
 - card stack object 85, 118, 119, 203, 210
 - cell margins 94, 100
 - cells
 - merging 93
 - splitting 93
 - check box object 85, 113, 120, 131, 146, 168
 - check syntax 139
 - choice list 47, 115, 124, 125, 173, 177, 179, 187, 219, 227, 297, 301
 - methods 301
 - clipboard 192, 199, 239
 - close application icon 110
 - code completion 145
 - column settings 92, 99
- combo box object 85, 113, 118, 124, 172
- command sequence 14, 35, 36, 49, 52, 53, 59, 67, 69, 105, 109, 112, 127, 147, 148, 161, 233, 237
- common, file scheme 244, 260
- compatible with physical quantity, unit
 - dimension check 72
- compatible with unit expression, unit
 - dimension check 72
- computation time 214, 319
 - expected 78, 211, 213
 - last 78, 213, 319
- COMSOL Client 7, 19, 21, 27, 29
 - file handing 241
 - running applications in 29, 241
- COMSOL Desktop environment 9, 14
- COMSOL Multiphysics 7, 19, 20, 25, 26, 29, 100, 127, 153, 193, 195, 197
- COMSOL Server 7, 19, 21, 24, 26, 27, 29, 30, 244
 - manual 31
- confirm 156, 294
- convert to new method 14, 53, 55, 127, 249
- copying
 - forms and form objects 100, 239
 - objects 42
 - rows or columns 93
- creating
 - feature node 157
 - forms 12, 32
 - local method 54, 147
 - methods 14
- CSV file 119, 199, 233, 335
- curly brackets 144
- cut point data set 291
- D**
 - DAT file 119, 199, 233
- data change 46, 112, 113, 148, 170, 171,

322

data display object 74, 77, 85
information node 213
tooltip 80

data file 119, 199, 233

data validation 72

date 318

debug log window 152, 310

debugging 151, 310

declarations node 10, 115, 168

delete button 42, 52, 208

deleting an object 42

derived values 77, 198

description text
Boolean variable 170
derived value 76

dialog box 293

disable form object 297

display name, for choice list 124, 174,
220, 227, 297

displayed text 43

double variable 118, 121, 122
conversion 265

double, data validation 73

drag and drop, form objects 41, 95

duplicating
rows or columns 93

duplicating an object 42

E edit local method 149
edit node 50, 132, 134
editor tools 46, 132, 136, 176
window 13

editor tree 47, 49, 61, 132, 241, 243

element size 84
change 185

email 325
class 287
methods 287

email attachment

export 287
report 287
table 287
embedded, file scheme 159, 163, 244,
252

enable form object 297

equation object 189

error message, data validation 72

errors and warnings window 139

event 46, 109, 112, 146
button on click 49
for multiple form objects 44, 113
form 112
form object 112
global 10, 85, 109
keyboard shortcut 48
local 109
node 110
on close 113
on data change 46, 113, 148, 170,
171, 322

on load 46, 113
on shutdown 110
on startup 110

events node 10

Excel® file 119, 199, 233

exit 302

experimental data 325

explicit selection 220

exponent, number format 78

export
email attachment 287
export button, results table 199
export node 241, 254
exporting
results 241, 254

external C libraries 311

extracting subform 89

extracting variable 141

- F** feature node
 creating 157
 removing 157
file
 commands 241
 declaration 126
 destination 207, 247
 download 29, 243
 import 50, 113, 126, 159
 menu 106
 methods 279
 opening 241
 saving 242
 types 208
 upload 29, 243
file import object 113, 126, 159, 206,
 241, 246
file open
 system method 283
file scheme
 common 244, 260
 embedded 159, 163, 244, 252
 syntax 159, 279
temp 244
upload 126, 244, 250, 253
user 244, 259
filename 208, 247, 281
files library 163
find 140
fit, row and column setting 89, 92
fixed, row and column setting 89, 92
for statement 155
form 11, 38
form collection 85, 103, 201
Form editor 16
 desktop location 9
 overview 11
 preferences 40
 using 37
form event 112
form object 11, 41, 45, 164
 event 112
 with associated methods 131
form reference 201
form tab, in ribbon 11
form window 11
forms node 10, 37
function 13
G geometry 26, 35, 49, 60, 65, 157, 222,
 239, 248, 250, 293
 import 207, 244, 246
 operations 193, 195, 197
get 309
GIF file 160
global evaluation 77, 203
global event 10, 109
global method 14, 112, 127
go to method 14, 53, 129
graphics 54
 animation 68
 commands 61
 hardware limitations 64
 object 33, 35, 59, 293
 plot group 64
 source for initial graphics content 59
tab, New Form wizard 35
toolbar 65, 85
 using multiple objects 64
view 62, 68, 298, 300
grid layout mode 29, 39, 86
grid lines, sketch layout mode 87
grow, row and column setting 89, 92
growth rules 89
H HTML
 code 191
 report 191, 261, 299
I icon 159, 237

button 48
close application 110
command 50
graphics 60
help 74
main window 102
menu item 105
method 131
ribbon item 105
toolbar 237
if statement 155
image
 background 39
 formats 160
 object 192
 thumbnail 24
images library 159
import
 file 50, 113, 126, 159, 208, 247
information card stack object 85, 210
information node 213, 319
inherit columns 99
initial size, of main window 103
initial values, of array 122
initialize
 parameter 55
 variable 55
input argument, to method 149
input field object 70, 85, 113, 118
 adding 70
 information node 213
 text object 76
tooltip 71
unit object 76
inputs/outputs tab, New Form wizard
 33
inserting
 form objects 45, 46
 rows and columns 89, 91
rows or columns 93
integer
 data validation 73
variable 118, 121, 122
variable conversion 265
item
 menu 105, 146
ribbon 108
toolbar 236

J Java® programming language 127, 153, 264
JPEG file 25, 160

K keyboard shortcut 15, 46, 109, 134, 145, 146, 153, 262
event 48, 105, 237

L language elements window 13, 131, 153
LaTeX 76, 79, 189
launching applications 28
layout mode 39, 86
layout options, form collection 201
libraries node 10, 159, 192
line object 190
list box object 85, 113, 118, 124, 225
LiveLink™ for Excel® 119, 199, 233
LiveLink™ products 29
local event 109
local method 14, 46, 54, 109, 112, 113, 114, 127, 131, 146, 147, 148, 170, 174, 239
log object 195
logo image 60
low-resolution displays 29

M main form 103
main window 103, 193
node 10, 102
margins
 cell 94, 100
material 177

math functions 155
menu 105, 108
 bar 103, 104
 item 59, 85, 146
merging cells 89, 93
mesh 35, 60, 65, 84
 change element size 185
 size 84
meshing 193, 195, 197
message log object 196, 295
method 10, 13, 44, 53, 59, 115, 127, 264
 event 110, 112
 form object 131
 global 14, 112, 127
 local 14, 46, 54, 109, 112, 114, 127,
 131, 146, 170, 174, 239
 tab, in ribbon 13
 window 13
Method editor 16, 264
 desktop location 9
 overview 13
 preferences 144
 using 127
methods node 10
Microsoft® Word® format 258, 299
minimum size
 form objects 94
model commands 243
model data access 85, 112, 134, 138,
 239
model expressions window 13, 141
model object 153, 158, 264, 291
model tree node 158
model utility methods 291
move down
 command sequence 52
 table 232
move up
 command sequence 52
table 232
MPH file 10, 19, 20, 22, 26, 32, 36, 159,
 243, 302
multiline text 77
multiple form objects
 selecting 44, 113
N name
 button 48
 check box 170
 choice list 124
 extract variable 142
 form 38
 form object 44
 graphics object 59
 menu 105
 method 143
 variable 117
new element value 122
new form 12
New Form wizard 45, 46, 47, 76
 buttons tab 35
 graphics tab 35
 inputs/outputs tab 33
new method 14
notation
 data display number format 78
 results table 199
number format 74, 78
number of rows and columns 89
O on click event, button 48
on close event 113
on data change event 46, 113, 148, 170,
 171, 322
on load event 46, 113
on shutdown event 110
on startup event 110
open file 241
operators 154

optimization 325, 335
OS commands 283
output argument, to method 149

P panes 201
parameter 13, 34, 54, 71, 72, 118, 239, 308, 309, 310
combo box object 172
declarations 10, 115
estimation 325
events 10, 109
input field object 70
method 138, 155
slider object 235
text label object 76
parentheses 144
password protected application 25
pasting
 form objects 42
 forms and form objects 101
image 192
 rows or columns 93
picture
 button 48
pixels 43, 86
play sound 29, 162, 284
player, animation 68
plot 35, 49, 50, 59, 65, 120, 134, 169, 175, 228, 236, 249, 257, 291, 293, 299, 309, 322
plot group 54
PNG file 25, 159, 160
point trajectories plot 291
position and size 43, 86, 88
 multiple form objects 44
positioning form objects 41
precedence, of operators 154
precision, number format 78, 199
preferences 15, 40, 144, 244, 245, 302
security 26

preview form 19
printing
 graphics 63, 302
procedure 13
progress 192, 312
progress bar object 192, 314, 316
progress bar, built-in 193
progress dialog box 194, 313

Q Quick Access Toolbar 19
 find 140

R radio button object 85, 113, 124, 218
recording code 136
recursion 150
regular expression 74
removing
 feature node 157
password protection 26
rows and columns 89, 91
 rows or columns 93
report 321, 324
 creating 106, 241, 254, 258, 259
email attachment 287
embedding 191
HTML 191, 261, 299
image 25
Microsoft® Word® format 299
node 241, 254, 259
request 156, 295
reset current view 62, 68
resizable graphics 29
resizing form objects 41
results table object 198, 297
ribbon 103, 107
 item 85, 108
section 108
tab 108
row settings 91
run application 19, 21

running applications
in a web browser 27, 241
in the COMSOL Client 29

S save
application 36
running application 22

save application command 242

save as 107, 302

save file 242

scalar variable 118, 173, 203, 235

scene light 62, 302

security settings 26

selection 35, 60, 65
explicit 220

selection input object 220

separator
menu 105
ribbon 108
toolbar 105, 236

set 304

set value command 84

settings window
form editor 11
method editor 13

show as dialog command 55

show form command 56

sketch grid 87

sketch layout mode 39, 86

sleep 320

slider object 85, 113, 234

smartphones
running applications on 29

solving 193, 195, 197

sound
play 162

sounds library 161

spacer object 237

splitting cells 89, 93

status bar 193

stopping a method 153

string variable 10, 55, 109, 111, 118, 121,
172, 174, 179, 201, 212, 224, 249,
310

conversion 265

methods 274

subroutine 13

syntax errors 139

syntax highlighting 143

system methods 283
OS commands 283

T table
email attachment 287

table object 113, 122, 229, 296, 325

tables, model tree 198

tablets
running applications on 29

temp, file scheme 244

test application 19, 21

test in web browser 19

text 105

text color 39

text file 118, 199, 233

text label object 70, 76, 78

text object 85, 113, 224
information node 213

thumbnail image 24, 25

time 318

title 293
form 38
main window 102
menu 105

toggle button 85, 165

toolbar 104, 199, 236
button, table object 232

graphics 65, 85

item 85, 236

separator 105, 236

tooltip

- button 48
- data display object 80
- input field object 71
- slider object 235
- toolbar button 237
- unit mismatch 73
- transparency 63, 302
- TXT file 118, 199, 233

- U** Unicode 76, 79
 - unit
 - dimension check 72, 73
 - expression 72
 - object 70, 75
 - Untitled.mph 22
 - update graphics 62
 - upload
 - file scheme 126, 244, 250, 253
 - URL 191
 - use as source
 - array input object 216
 - card stack object 204
 - check box object 170
 - combo box object 173
 - data display object 77
 - declaration 116
 - explicit selection 221
 - graphics object 59
 - information card stack object 212
 - input field object 70
 - list box object 226
 - radio button object 219
 - results table object 198
 - selection input object 221
 - slider object 235
 - table object 230
 - text object 224
 - user
 - file scheme 244, 259
 - user interface layout 11

- V** variable 10, 115, 138
 - accessing from method 155
 - activation condition 125
 - Boolean 120, 121, 168
 - declaration 10, 115
 - derived values 78
 - double 121
 - events 10, 109, 111
 - extracting 141
 - find and replace 140
 - input field object 70
 - integer 121
 - name completion 145
 - scalar 173, 203, 235
 - slider object 235
 - string 118, 121, 172, 174, 201, 212, 224
 - text label object 76
 - view
 - go to default 3D 68
 - graphics 62, 68, 298, 300
 - reset current 62, 68
 - volume maximum 77

- W** WAV file 161
 - web browser 7, 21
 - file handling 241
 - web page object 191
 - WebGL 27
 - while statement 155
 - with statement 144, 155, 304
 - wrap text
 - text label object 77

- Z** zoom extents 62, 228, 249, 299, 302

