

INTRODUCTION TO Application Builder

Introduction to Application Builder

© 1998–2019 COMSOL

Protected by 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; 9,146,652; 9,323,503; 9,372,673; 9,454,625; and 10,019,544. 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, the COMSOL logo, COMSOL Multiphysics, COMSOL Desktop, COMSOL Compiler, COMSOL Server, and LiveLink 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 5.5

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: CM02001 |

Contents

Preface.	9
Introduction	10
The Application Builder Desktop Environment	12
The Application Builder and the Model Builder.	19
Parameters, Variables, and Scope	20
Running Applications.	22
Running Applications in COMSOL Multiphysics	22
Running Applications with COMSOL Server	30
Compiling and Running Standalone Applications	35
Publishing COMSOL Applications	41
Getting Started with the Application Builder	43
The Form Editor	48
The Forms Settings Window.	48
The Individual Form Settings Windows	49
Form Editor Preferences	51
Form Objects	52
Editor Tools in the Form Editor	57
Button.	59
Graphics	71
Input Field	88
Unit.	95
Text Label	95
Data Display	96
Data Access in the Form Editor	99
Sketch and Grid Layout	105

Copying Between Applications	120
Using Forms in the Model Builder	121
Inputs	123
The Main Window	127
Menu Bar and Toolbar	129
Ribbon	133
Events	134
Events at Startup and Shutdown	135
Global Events	136
Form and Form Object Events	139
Using Local Methods	140
Declarations	141
Scalar	145
Array 1D	148
Array 2D	149
Choice List	151
File	153
Unit Set	154
Shortcuts	159
Graphics Data	161
The Method Editor	165
Converting a Command Sequence to a Method	165
Language Elements Window	170
Editor Tools in the Method Editor	171
Data Access in the Method Editor	173
Recording Code	175
Checking Syntax	178

Find and Replace	179
Model Expressions Window	180
Use Shortcut	180
Syntax Highlighting, Code Folding, and Indentation	182
Method Editor Preferences	184
Ctrl+Space and Tab for Code Completion.	185
Creating Local Variables.	187
Local Methods.	188
Methods with Input and Output Arguments.	190
Debugging	192
Stopping a Method	194
The Model Object	194
Language Element Examples	194
Running Methods in the Model Builder	198
Creating Add-Ins	205
Add-In Libraries.	208
Workflow when Creating and Editing Add-Ins	210
Libraries.	211
Images.	212
Sounds	213
Files	216
Appendix A — Form Objects	217
List of All Form Objects.	217
Toggle Button	218
Check Box	221
Combo Box.	225
Equation	245
Line.	246

Web Page	247
Image	248
Video	248
Progress Bar	250
Log	253
Message Log	254
Results Table	255
Form	257
Form Collection	259
Card Stack	261
File Import	265
Information Card Stack	268
Array Input	271
Radio Button	275
Selection Input	277
Text	281
List Box	282
Table	286
Slider	291
Hyperlink	293
Toolbar	295
Spacer	296
Appendix B — Copying Between Applications	298
Appendix C — File Handling and File Scheme Syntax	300
File Handling with COMSOL Server	300
File Scheme Syntax	303
File Import	305
File Export	312

Appendix D — Keyboard Shortcuts.....	321
Appendix E — Built-In Method Library	324
Appendix F — Guidelines for Building Applications.....	342
Appendix G — The Application Library Examples.....	345

Preface

The typical user of a simulation package is someone who holds a PhD or an MSc, has several years of experience in modeling and simulation, and underwent thorough training to use the specific package. He or she typically works as a scientist in the R&D department of a big organization or as an academic researcher. Because the theory of simulation is complicated and the typical simulation package presents many options, it is up to the user to employ his or her expertise to validate the model and the simulation.

This means that a small group of simulation experts is serving a much larger group of people working in product development, production, or as students studying physics effects. Simulation models are oftentimes so complicated that the person who implemented the model is the only one who can safely provide input data to get useful output. Hence the use of computer modeling and simulation creates a bottleneck in product development, production, and education.

In order to make it possible for this small group to service the much larger group, the Application Builder offers a solution. It enables simulation experts to create an intuitive and very specific user interface for his or her otherwise general computer model – a ready-to-use application. The general model can serve as a starting point for several different applications, with each application presenting the user with input and output options relevant only to the specific task at hand. The application can include user documentation, checks for “inputs within bounds”, and predefined reports at the click of a button.

Creating an application often requires a collaborative effort by experts within the areas of: physics, numerical analysis, programming, user-interface design, and graphic design.

To a reasonable extent, COMSOL’s Technical Support team can recommend physics and numerical analysis settings for your application. In addition, the COMSOL documentation and online resources can be of great help. For programming and design, the Technical Support team can provide very limited help. These are areas where your own development efforts are critical.

The Application Builder makes it easy for a team to create well-crafted applications that avoid accidental user input errors while keeping the focus on relevant output details.

We at COMSOL are convinced that this is the way to spread the successful use of simulation in the world and we are fully committed to helping make this possible.

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 COMSOL Multiphysics license, applications can be run from the COMSOL Desktop in Windows[®], macOS, and Linux[®].

RUNNING COMPILED APPLICATIONS

By using COMSOL Compiler[™] you can compile your application to an executable file for Windows[®], Linux[®], and macOS. You can freely distribute the executable and it can be run without any license file.

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[®], macOS, 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, or 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 programming, you may want to read “Appendix F — Guidelines for Building Applications” on page 342.

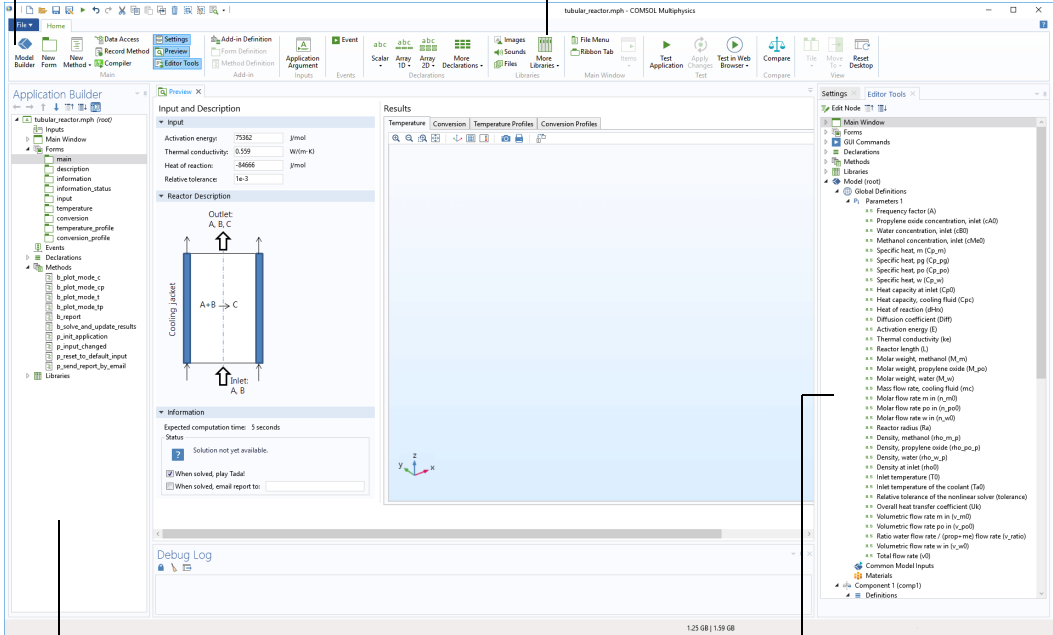
ADDITIONAL DOCUMENTATION

Additional documentation with information relevant to building applications can be found in the books: *Application Programming Guide*, *Application Builder Reference Manual*, and *Programming Reference Manual*.

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 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.

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 48)
- Method editor (see page 165)

THE APPLICATION TREE

The application tree consists of the following nodes:

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

The **Inputs** node contains subnodes that are of the type **Application Argument**. These can be used for input arguments to the application when starting it from the command line of the operating system.

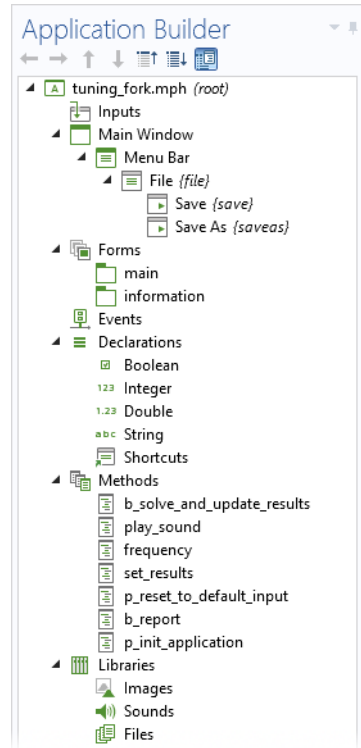
The **Main Window** node represents the main window of an application and is also the top-level node for the user interface. It contains the window layout, the main menu specification, and an optional ribbon specification.

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. Methods can modify the model object of a running application or the model object represented by the Model Builder in the current session.





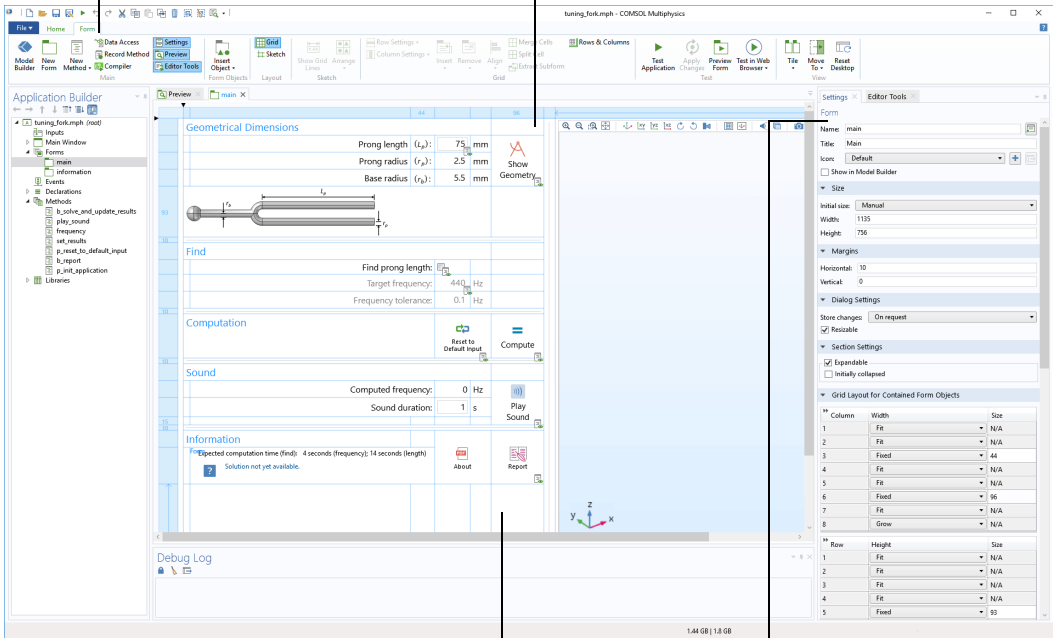
Previous versions provided two different types of methods depending on if they modified the model object of a running application or the model object represented by the Model Builder in the current session. This distinction is no longer necessary but methods can now serve both purposes.

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, for example **form1**, and you would like to edit it, you can open the Form editor in either of two ways:

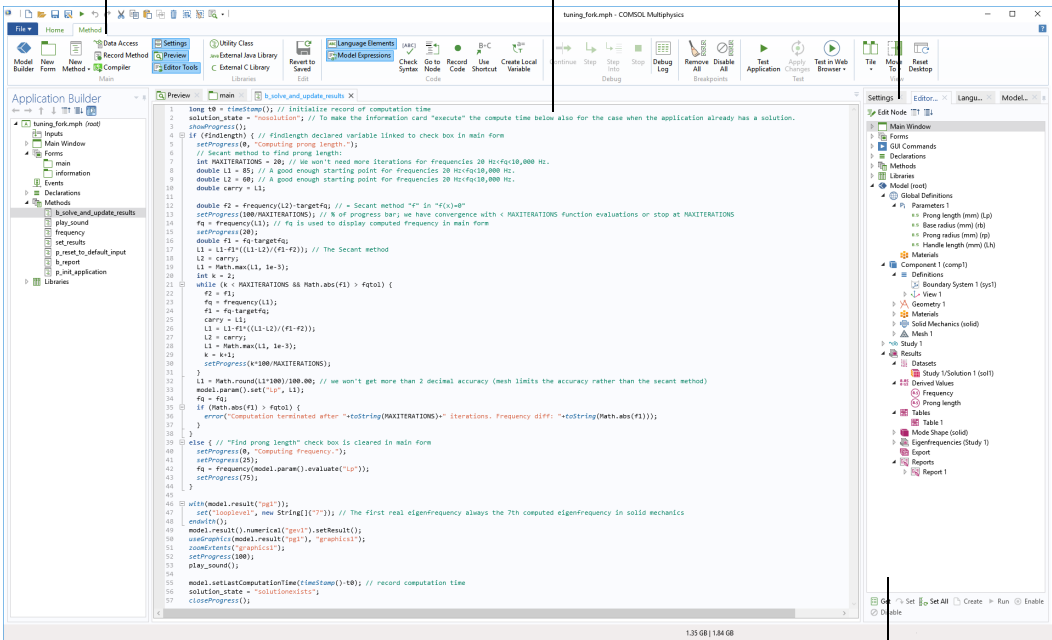
- In the application tree, double-click the **form1** node.
- In the application tree, right-click the **form1** node and select **Edit**.

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.

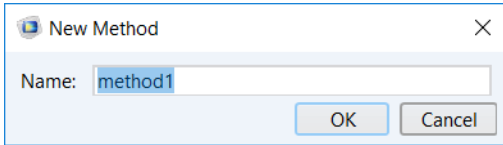
Use the Method editor to write methods for actions not covered by the standard use of the model tree nodes. A method is another name for what is known in other programming languages as a subroutine, function, or procedure.

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)

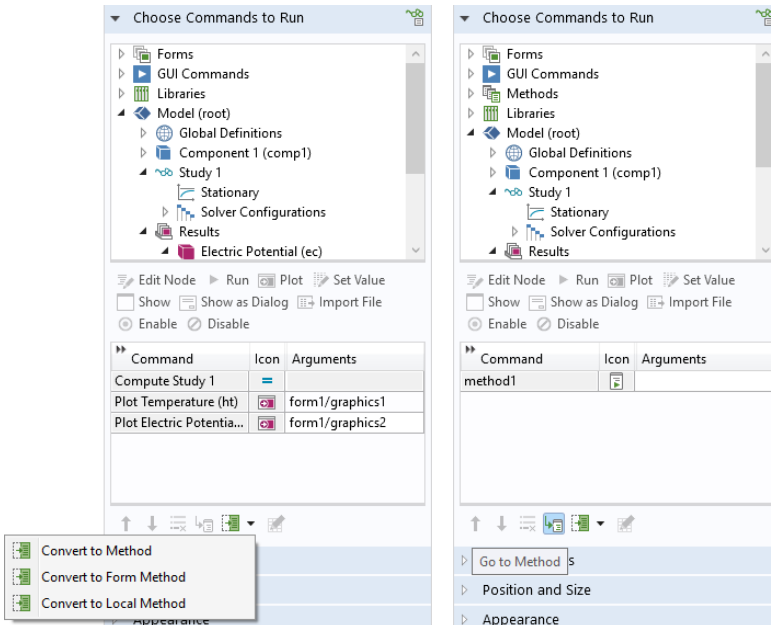
Creating a New Method

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. In the **New Method** dialog box you can change the name of the method.



Creating a new method will automatically open the Method editor. Methods created in this way are global methods and accessible from all methods, form objects, and from the Developer tab in the Model Builder ribbon.

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



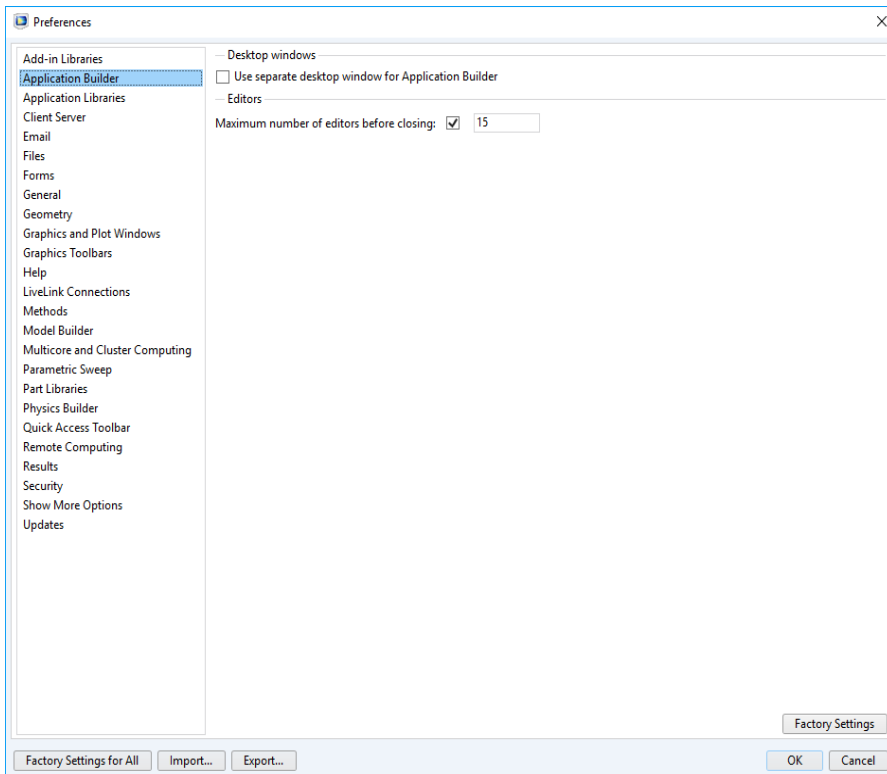
If a method already exists, say with the name `method1`, then you can open the Method editor in any of these ways:

- In the application tree, double-click the **method1** node.

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

APPLICATION BUILDER PREFERENCES

To access **Preferences** for the Application Builder, choose **Preferences** from the **File menu** and select the **Application Builder** page.



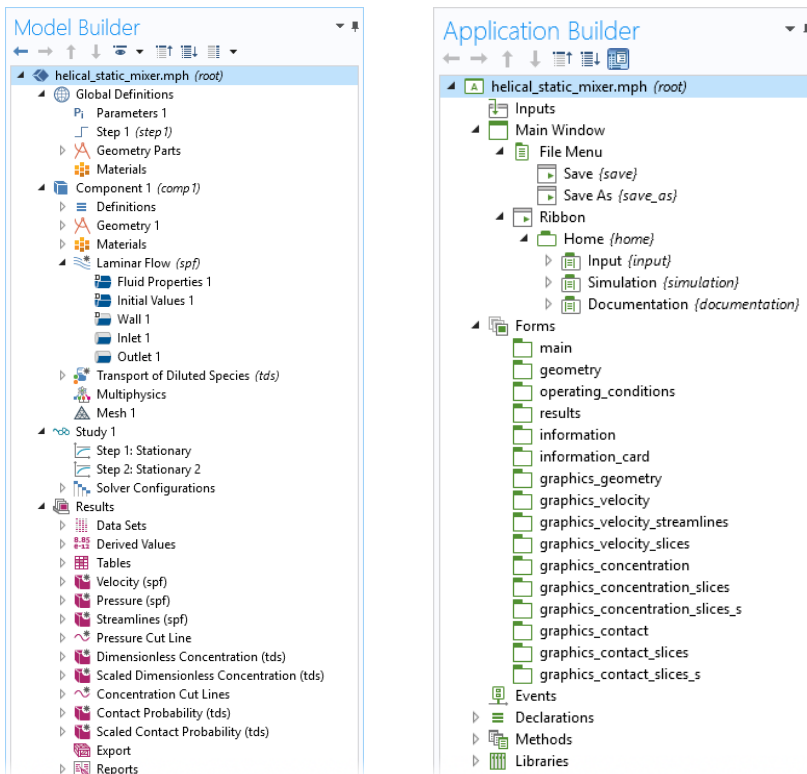
You can configure the COMSOL Desktop environment so that the Application Builder is displayed in a separate desktop window. Select the check box **Use separate desktop window for Application Builder**.

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

You can set an upper limit to the number of open Form editor or Method editor window tabs. Select the check box **Maximum number of editors before closing** and edit the number (default 15). Keeping this number low can speed up the loading of applications that contain a large number of forms.

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 figures below show the Model Builder and Application Builder windows.



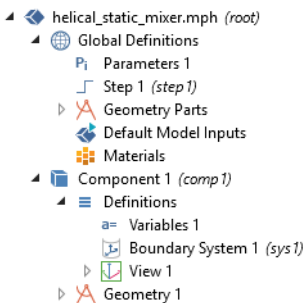
When creating an application, you typically start from an existing model. However, you can just as well build an application user interface and the underlying model simultaneously. You can easily, at any time, switch between the Model Builder and Application Builder. The model part of an application, as represented by the model tree, is sometimes 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 57 and 88.
- 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 43 and 57.
- The 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 57, 99, and 171.
- 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 175.

Parameters, Variables, and Scope

The model tree may contain both parameters and variables that are used to control the settings of a model. The figure 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:

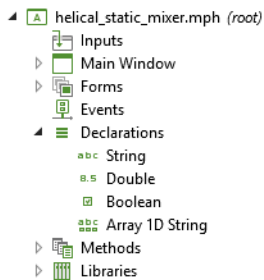
- Parameterizing geometric dimensions

- 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 figure 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 141 and 324. For more information on typed variables, see the *Application Programming Guide*.

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®.

By using COMSOL Compiler™, you can compile your application to an executable file that can be run in the Windows®, Linux®, and macOS operating systems.

The following two sections explain how to run applications in these different settings. The third section, “Publishing COMSOL Applications” on page 41, describes your rights to publish applications.

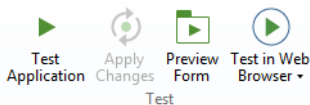
Running Applications in COMSOL Multiphysics

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

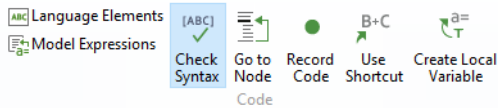
Test Application is used for quick tests. It opens a separate window with the application user interface while keeping the Application Builder desktop environment running.



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.

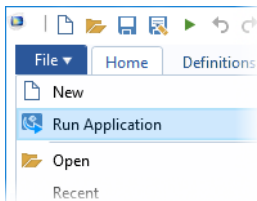
When **Test Application** is used, all methods are automatically compiled with the built-in Java[®] compiler. Any syntax errors will generate error messages and the process of testing the application will be stopped. To check for syntax errors before testing an application, click the **Check Syntax** 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 165.

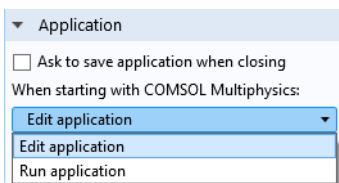
RUNNING AN APPLICATION

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.

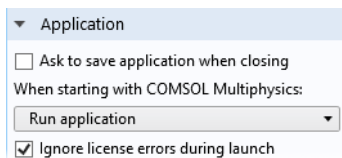


The option **Edit application** will open the application in the Application Builder. The option **Run application** will open the application in runtime mode for production purpose use. This option is similar to selecting **Run Application** in the **File** menu with the difference that double-clicking 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.

IGNORING LICENSE ERRORS

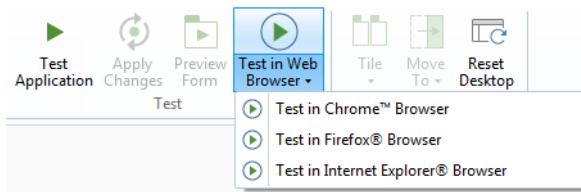
In the **Settings** window for the application tree root node, the check box **Ignore license errors during launch** is used to control the behavior with respect to licensed products when running applications.



When selected, an application can be started even if all required licenses are not available. It is still not possible to use the functionality of products for which the license is not available. However, you can write methods to create an application such that the functionality used is dynamically adapted to which types of licenses are available.

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 application 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 application 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 macOS, iOS, Linux®, and Android™, a COMSOL Server installation is required.

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

SERVER SOFTWARE	CLIENT SOFTWARE TOOL OR COMPONENT
COMSOL Multiphysics	Test Application
COMSOL Multiphysics	Test in Web Browser
COMSOL Multiphysics	Run Application
COMSOL Server	COMSOL Client for Windows®
COMSOL Server	Web Browser
N/A	Executable file compiled with COMSOL Compiler

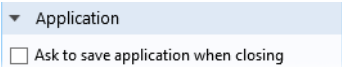
The Server column represents the software components that perform the CPU-heavy computations. The Client column represents the software tool or component used to present the application user interface. In the case of executable files, all computations are done locally. For more information on compiled applications, see “Compiling and Running Standalone Applications” on page 35.

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**

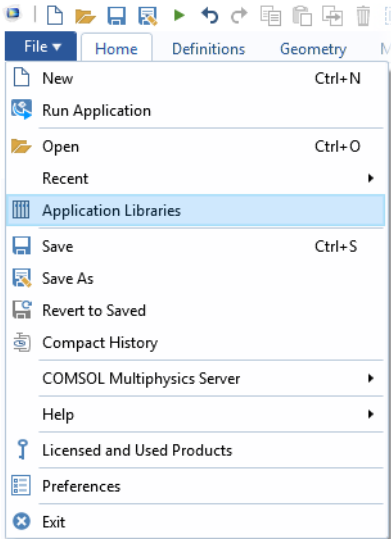
in which you may select the check box **Ask to save application when closing**, as shown in the figure below.



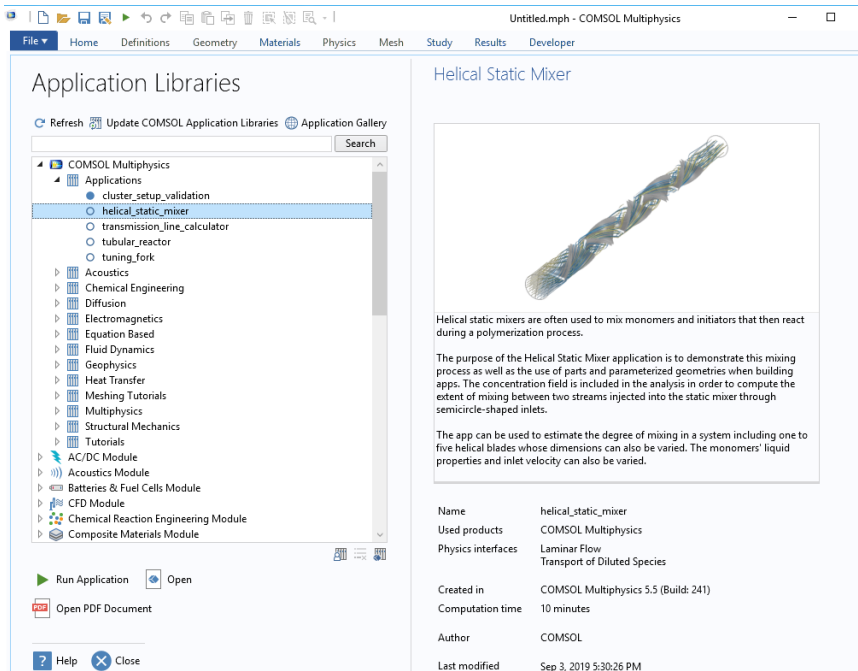
As an alternative, you can add a button or menu item with a command to save the application. For more information, see page 132.

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**.

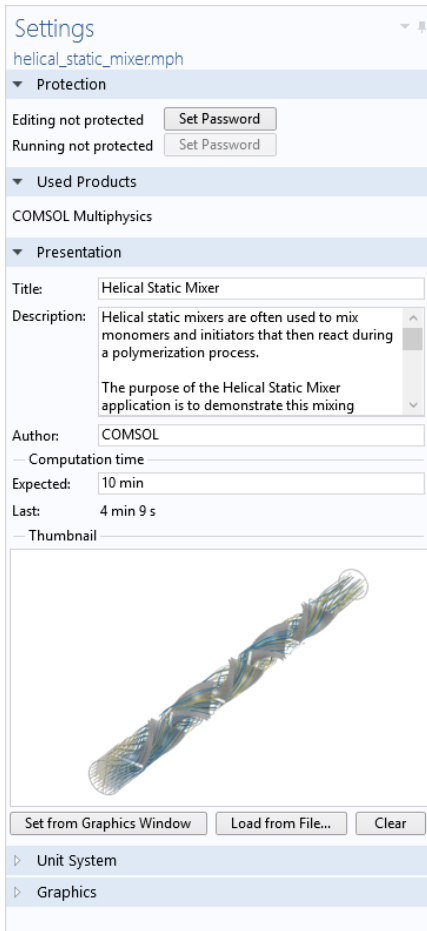
The applications in the Application Libraries are continuously improved and updated. You can update the Application Libraries by clicking Update COMSOL Application Libraries.

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. 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 for choosing the image: **Set from Graphics Window** and **Load from File**. You can also **Clear** the image.

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.

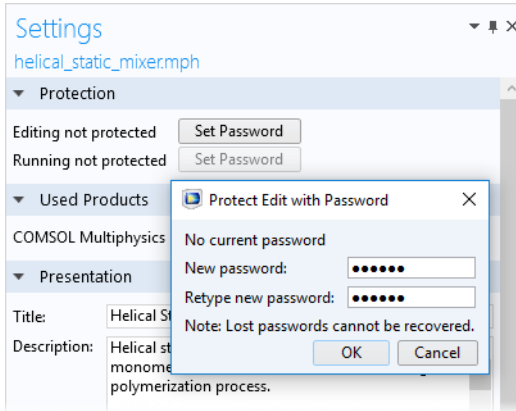


The **Set from Graphics Window** option automatically creates two thumbnail images:

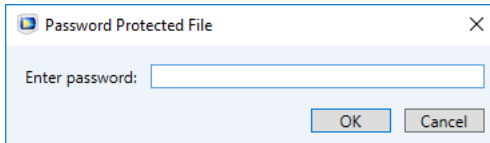
- An image of size 280-by-210 pixels shown in the **Settings** window of the application tree root node and in the Application Libraries.
- An image of size 1024-by-768 used as the default title page image in reports and in the Application Libraries in COMSOL Server.

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, is 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, and more.

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.

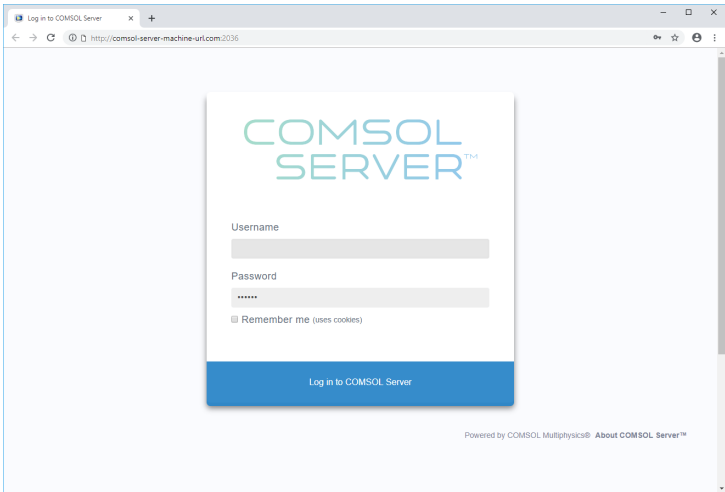
Running Applications with COMSOL Server

COMSOL applications can be run by connecting to COMSOL Server from a web browser or 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](#).

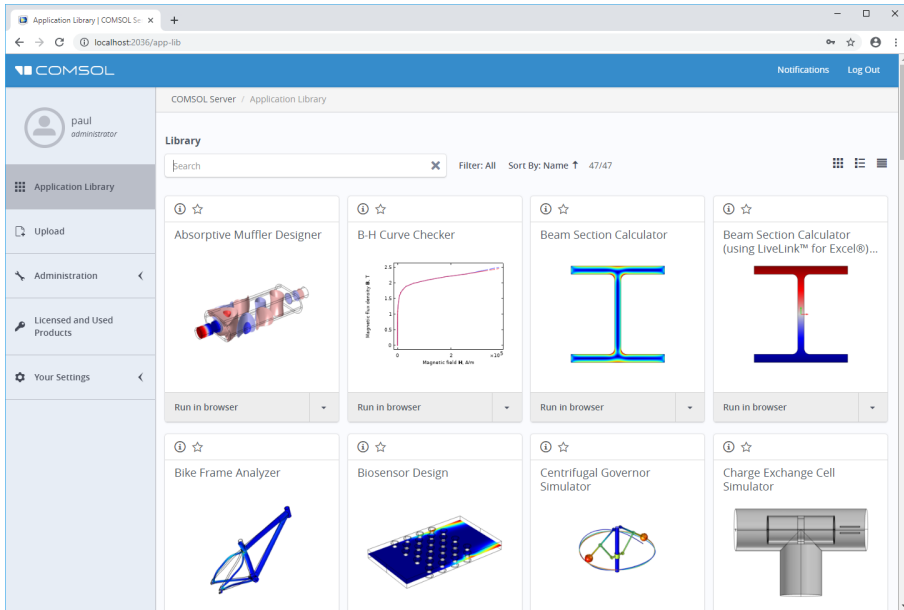
Running applications in a web browser does not require any installation or web browser plug-ins. Running an application in a web browser supports interactive graphics in 1D, 2D, and 3D. In a web browser, graphics rendering in 3D is based on WebGL™ technology, which is included with all major web browsers.

RUNNING APPLICATIONS IN A WEB BROWSER

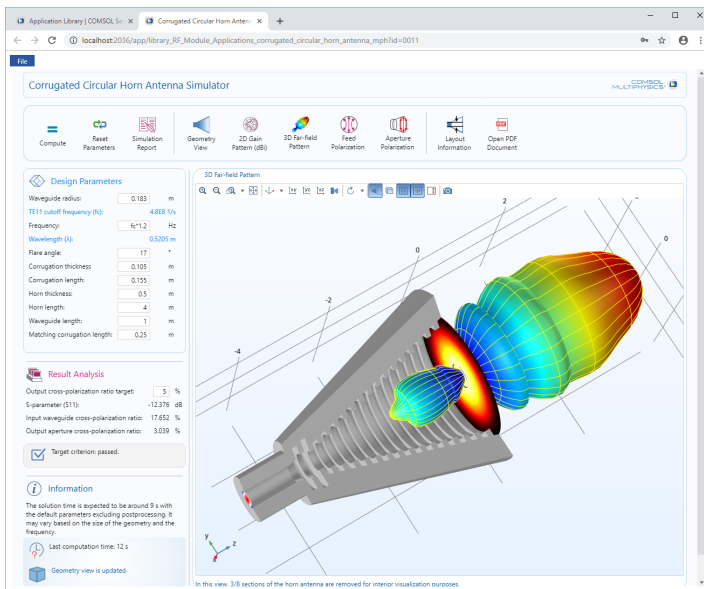
Using a web browser, you can point directly to the computer name and port number of a COMSOL Server web interface — for example, `http://comsol-server-machine-url.com:2036`, assuming that port number 2036 is used by your COMSOL Server installation. You need to provide a username and password to log in. If you are running COMSOL Server locally, the address field will typically be `localhost:2036`.



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



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



Limitations When Running Applications in Web Browsers

When you create applications to run in a web browser, make sure you use the grid layout mode in the Application Builder; See “Sketch and Grid Layout” on page 105. 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 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.

Note that file browsing functionality is slightly different depending on the web browser and depending on the version of the web browser. This may impact the user experience when running an application that has functionality for saving files to the client computer. For example, the location of the downloads folder can be changed in the settings of many web browsers. A web browser may also allow the user to manually specify the download location for each file. Please refer to the documentation of your target web browsers for details.

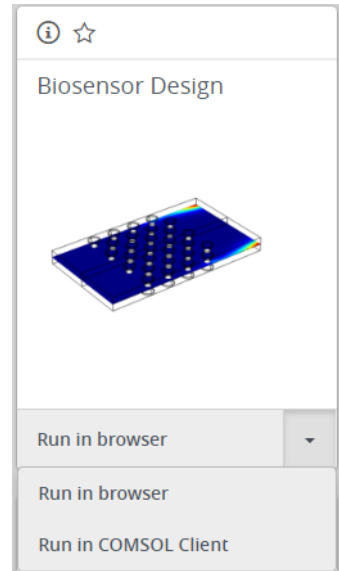
RUNNING APPLICATIONS IN THE COMSOL CLIENT

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

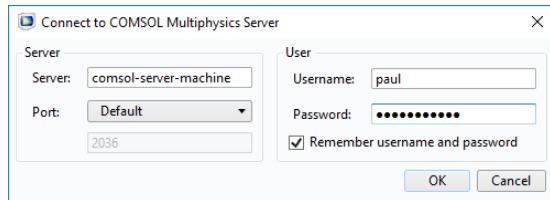
applications that require a LiveLink™ product for CAD, provided that the COMSOL Server you connect to has the required licenses. You can open an application with the COMSOL Client for Windows® in two different ways:

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

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

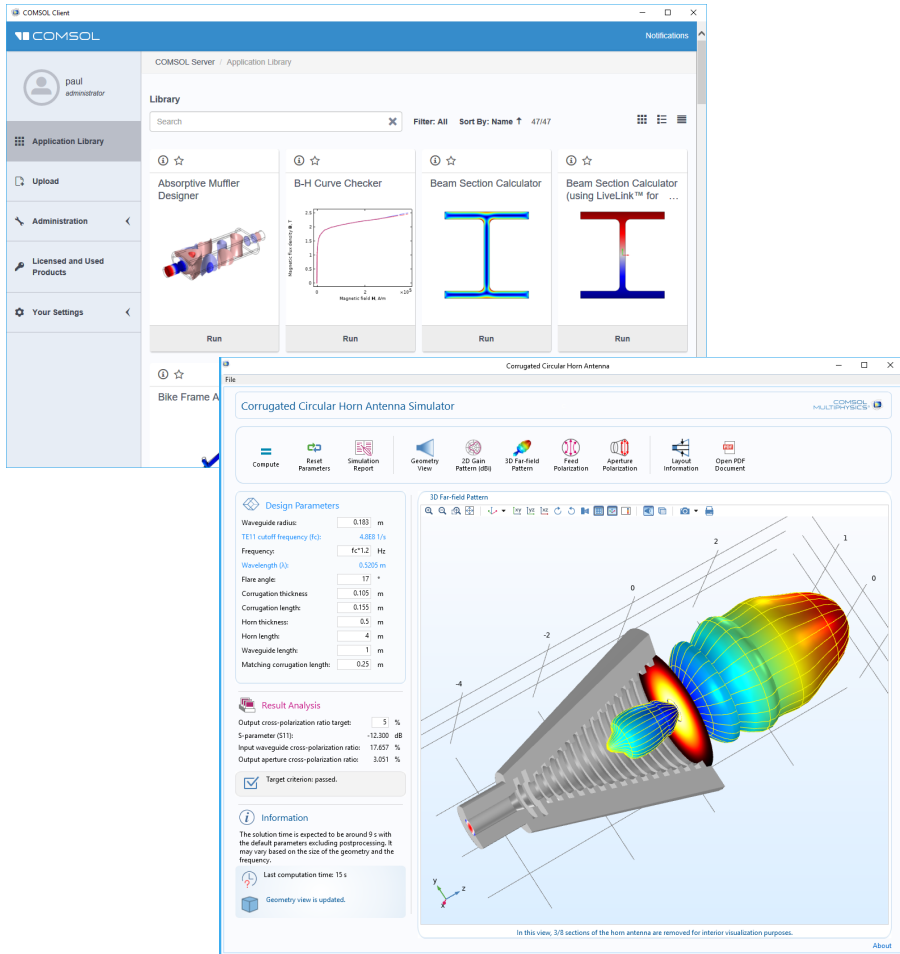


- If you have the COMSOL Client for Windows® already installed, a desktop shortcut will be available. You can double-click its desktop icon and before you can use the COMSOL Client to run applications, you will be prompted to log into a COMSOL Server with a valid username and password. After login, the 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).



RUNNING COMSOL SERVER ON MULTIPLE COMPUTERS OR A CLUSTER

COMSOL applications can be run on multiple computers or clusters in two main ways:

- By installing COMSOL Server with primary and secondary instances.
- By configuring one of the study nodes in the Model Builder for a particular cluster.

Primary and Secondary Instances

Running COMSOL Server on multiple computers using primary and secondary instances allows for more concurrent users and applications than a single computer instance (or installation). The main COMSOL Server instance is called primary and the other instances are called secondary. The primary server is used for all incoming connections — for example, to show the web interface or to run applications in a web browser or with COMSOL Client. The actual computations are offloaded to the secondary server computers. This type of installation has a major benefit: Applications do not need to be custom-built for a particular cluster. Load balancing is managed automatically by the primary server, which distributes the work load between the secondary servers. A COMSOL Server installation can consist of multiple primary and secondary server installations without additional license requirements. You can perform administrative tasks using the COMSOL Server web interface without checking out license keys for users running applications. License keys are only checked out when running applications.

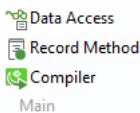
Configuring a Study Node for Cluster Sweep or Cluster Computing

If you want to utilize a cluster for applications that require large parametric sweeps or high-performance computing power, then you can configure the Model Builder study node(s) of an application using the Cluster Sweep and Cluster Computing options. Note that for building such applications, you will need a Floating Network License. You can find more information on configuring a study node for clusters in the books *Introduction to COMSOL Multiphysics* and the *COMSOL Multiphysics Reference Manual*. For running such cluster-enabled applications, you can use either COMSOL Server or a Floating Network License of COMSOL Multiphysics. Cluster system configurations are available from the COMSOL Server web interface.

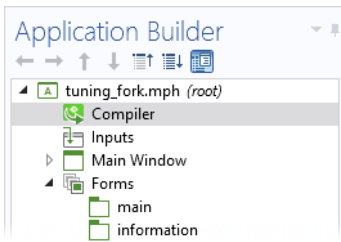
For more information on COMSOL Server, see the *COMSOL Server Manual* available with a COMSOL Server installation or from http://www.comsol.com/documentation/COMSOL_ServerManual.pdf.

Compiling and Running Standalone Applications

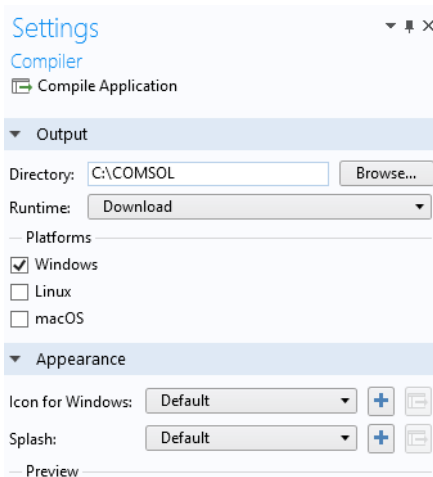
If you have a license of COMSOL Compiler™, there will be a **Compiler** button in the ribbon section **Main**, as shown below.



Clicking this button will add a **Compiler** node to the application tree, shown in the figure below.



The corresponding **Settings** window is shown below.



COMPILING APPLICATIONS

To compile an application, you need to make a few selections in this window. Specify an output **Directory**, where the executable files will be saved after compilation.

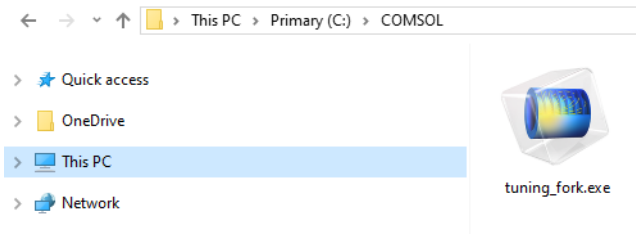
The **Runtime** option can be left at **Download** for most situations. COMSOL Runtime contains all the COMSOL Multiphysics software components needed to run the application as a standalone program. The **Runtime** setting specifies where the COMSOL Runtime environment will be stored and affects the behavior when the compiled application is started for the first time on a computer. If this setting is **Download** then the first time a user is starting the compiled application the COMSOL Runtime environment files will be downloaded (a service provided by COMSOL). If the COMSOL Runtime environment already exists on the computer, with a matching version number, then no download will be performed.

The option **Enabled** will bundle the COMSOL Runtime files in the executable file. Note that with this option, the file size may be several hundred megabytes even for smaller applications.

The **Platforms** settings determine which target-platform executables should be generated at compilation. The extensions of the executables for the Windows[®] and Linux[®] operating systems will be `.exe` and `.sh`, respectively. For macOS, a `.tar` archive is created; unpack this archive on macOS to extract the app.

The **Icon for Windows** lets you specify the desktop icon. The **Splash** setting lets you specify a BMP-image file to be displayed at startup.

After compilation, in the Windows[®] operating system, the executable file will be available in the output directory, as shown in the figure below.

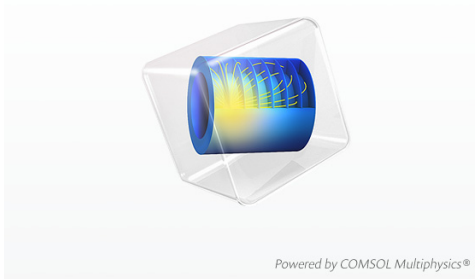


As a next step, you can, for example, right-click the EXE-file and create a shortcut that you then place on the Windows[®] desktop.

You can also compile an application from the operating system command line. For more information, see the *COMSOL Multiphysics Reference Manual*.

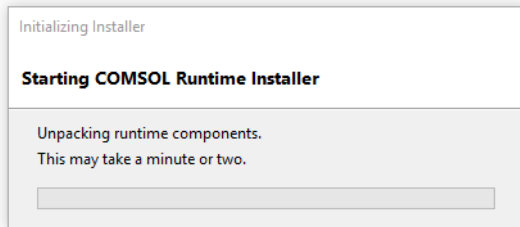
RUNNING COMPILED APPLICATIONS

When running a compiled application, for example, by double-clicking the `.exe` file in the Windows[®] operating system, a splash screen is shown and the application will start. If the application has the **Splash** option set to **Default**, then a neutral-looking built-in splash screen will be shown.

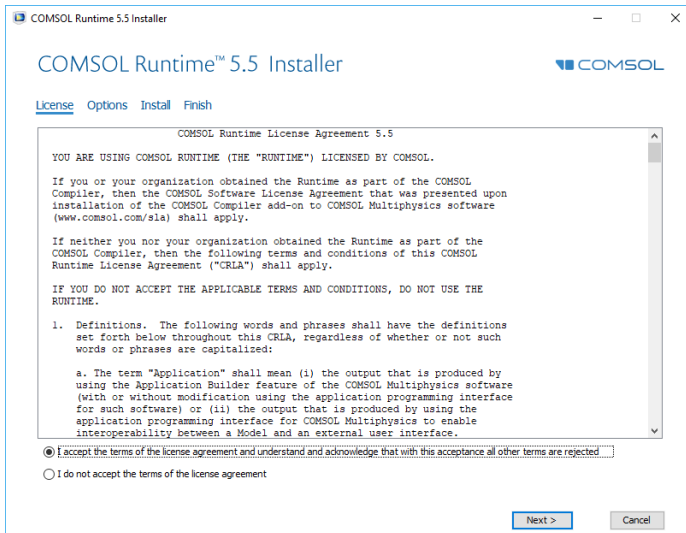


It is recommended that you replace this with your own splash screen.

If this is the first time you are running an application on a particular computer, then, in addition, a click-through agreement and an **Initializing Installer** progress window will be displayed. The initialization progress window is shown below.

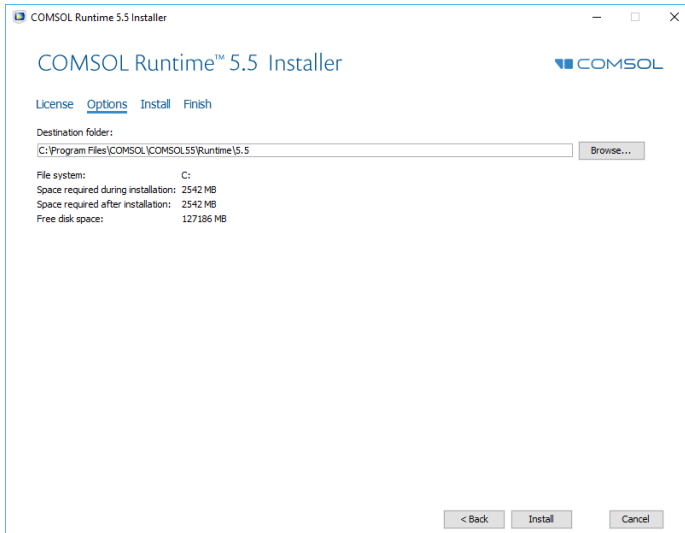


After a short moment, the COMSOL Runtime Installer window is displayed, as shown below.



The COMSOL Runtime Installer and its click-through agreement are only shown once, and the next time you start the same application, it will not be shown. The click-through agreement and initialization progress window will also not be shown if you run another application on the same computer that was generated with the same COMSOL Compiler version (having the same version of the COMSOL Runtime).

Click **Next** to select the location of the COMSOL Runtime files and the click **Install**, as shown in the figure below.



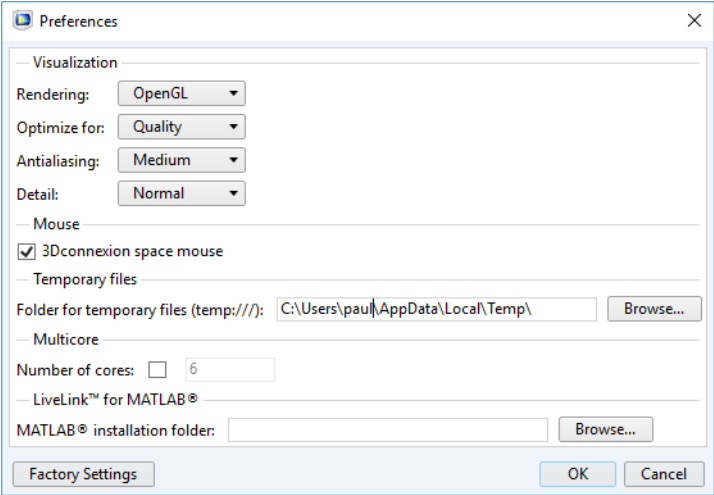
The installation takes a few minutes and, when finished, the installer will prompt the user to start the application.

An option for showing the COMSOL **About** dialog box is always available in a compiled application. The author of the application controls how this information

is available from the **Settings** of the **Main Window**; see “About Dialog” on page 129. The figure below shows the **About** dialog box.



In the **About** dialog box, the user of a compiled application can access the **Preferences** by clicking the corresponding button. The **Preferences** dialog box for a compiled application is shown below.



Here, the user can change settings for **Visualization**, **Mouse**, **Temporary files**, **Multicore**, and **LiveLink™ for MATLAB®**. These settings represent a subset of the **Preferences** available in COMSOL Multiphysics and more information can be found in the *COMSOL Multiphysics Reference Manual*.

- ⚠ If the compiled application detects that OpenGL® graphics hardware acceleration is not supported, then the application will automatically switch to software rendering and exit. The next time the application starts, software rendering will be used.

Publishing COMSOL Applications

The COMSOL Software License Agreement (SLA) gives you permission to publish your COMSOL applications for others to use, including commercially, with certain restrictions spelled out in the SLA available here: www.comsol.com/sla. This permission enables you to share your applications with others and to charge them for using your applications through three different mechanisms.

First, you can make an application available to others to be run by a COMSOL Multiphysics installation. For using an application with COMSOL Multiphysics, the user needs to belong to the same organization that purchased the COMSOL Multiphysics license.

Second, you can make an application available to others to be run by a COMSOL Server installation. This approach allows for greater flexibility, as it allows you to set up a COMSOL Server installation and let users from around the world access your Application. You just need to provide them with the address, a username, and password to your COMSOL Server installation. Alternatively, users can purchase their own COMSOL Server license. If you use COMSOL Server to host and run applications, the SLA also gives you permission to make time on your COMSOL Server License (CSL) available to persons outside your organization to host and run applications that you are publishing to others, subject to certain restrictions.

Third, you can use COMSOL Compiler to compile your application into a standalone program that contains all of the functionality required to make it run. This approach gives you the greatest flexibility, as the end user of your application will not need a license for COMSOL Multiphysics or COMSOL Server to run the Application. The compiled application can then be run by that user and anyone else to whom you allow the user to publish the compiled application, around the world, inside or outside of your organization.

The COMSOL Application License, also available at www.comsol.com/sla, further lets you modify applications available in the Application Libraries and

publish those modified applications for others to use, including commercially, with certain restrictions spelled out in the Application License. This allows you to, for example, use one of the applications in the Application Libraries as a starting point for your own applications by adding or removing your own features.

If you wish to apply the Application License to applications that you create, the Application License contains instructions on how to do so. The Application License also addresses how you can use terms that you choose for modifications you make to applications available in the Application Libraries, while the original portions of those applications remain available under the Application License.

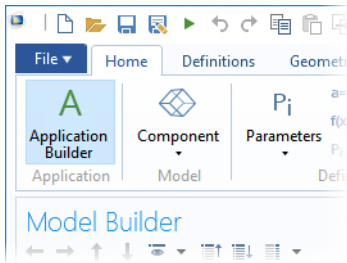
The results from a simulation software such as COMSOL Multiphysics can shorten design times dramatically by, for example, reducing the number of experiments or product tests. However, simulation software is not a substitute for real-world testing. This is especially important if there are risks for physical or environmental damage.

Getting Started with the Application Builder

STARTING FROM A COMSOL MULTIPHYSICS MODEL

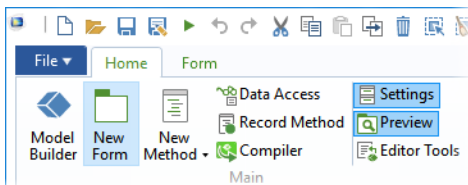
If you do not have a model already loaded to the COMSOL Desktop environment, select **File>Open** to select an MPH file from your file system or select a file from the Application Libraries. Note that, in the Application Libraries, the files in the **Applications** folders are ready-to-use applications. 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

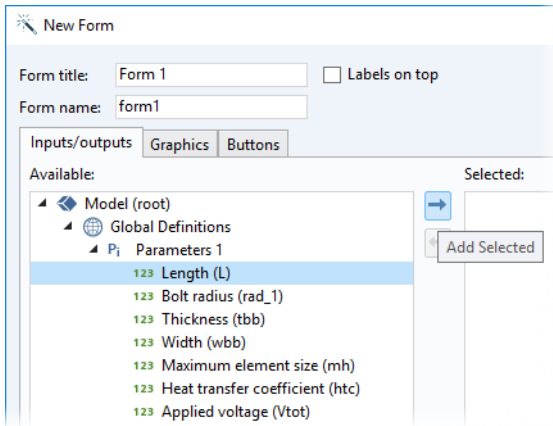
To start working on the user interface layout, click the **New Form** button in the **Home** tab. This will launch the **New Form** wizard.




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**

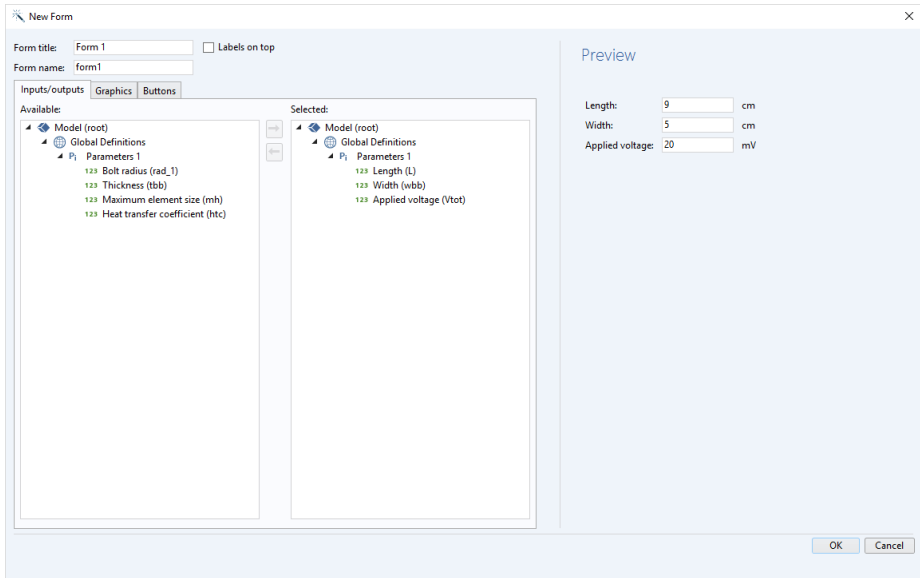


Double-click a node or click the **Add Selected**  button to move a node from the **Available** area to the **Selected** area. The selected nodes will become form objects in the application, and a preview of the form is shown in the **Preview** area to the right. The size as well as other settings for 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 49. You can also choose to exit the **New Form** window at this stage by clicking **Done**, and then manually add form objects.

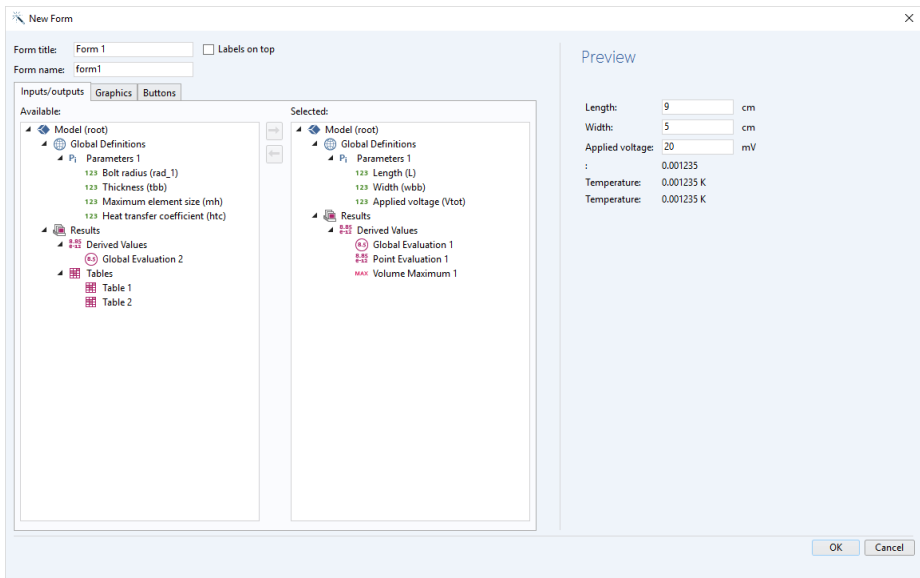
The Inputs/Outputs Tab

The **Inputs/outputs** tab displays the model tree nodes that can serve as an input field, data display object, check box, or 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 **Data Access** (see page 99). Check box and combo box objects are, for example, 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 **Data Access**.

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



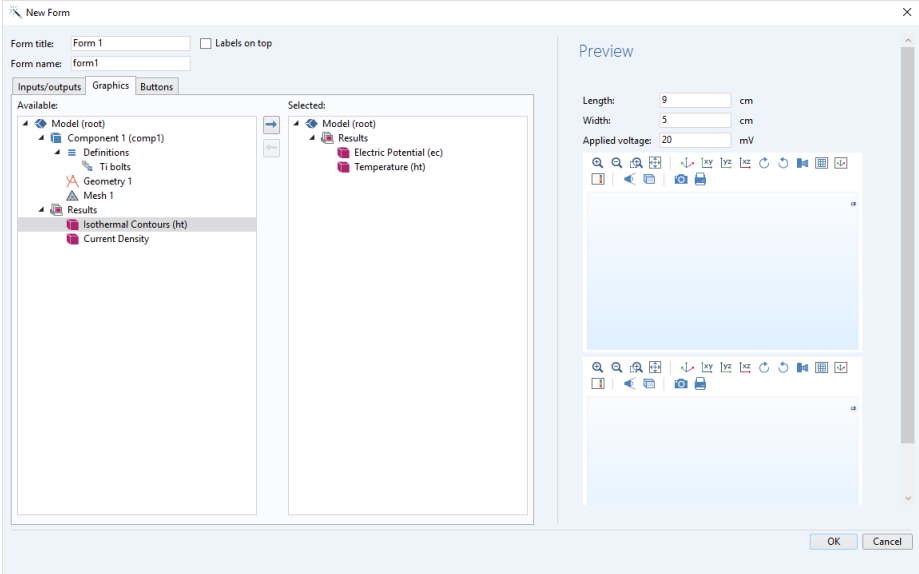
In the figure below, corresponding to a different model, three **Derived Values** nodes have been selected to serve as data display objects.



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

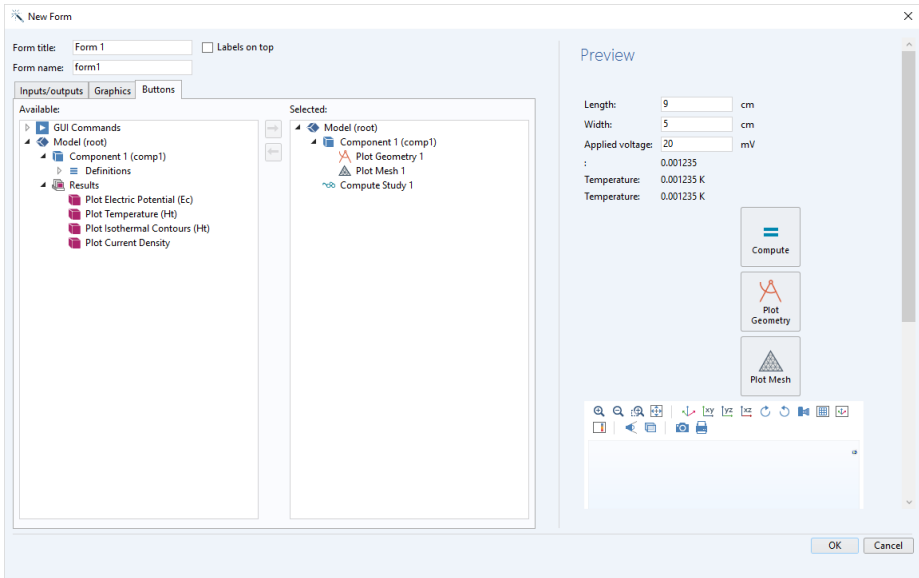
The **Graphics** tab displays the model tree nodes that can serve as graphics objects: **Geometry**, **Selection**, **Mesh**, and **Results**. In the figure below, two such nodes have been selected.



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

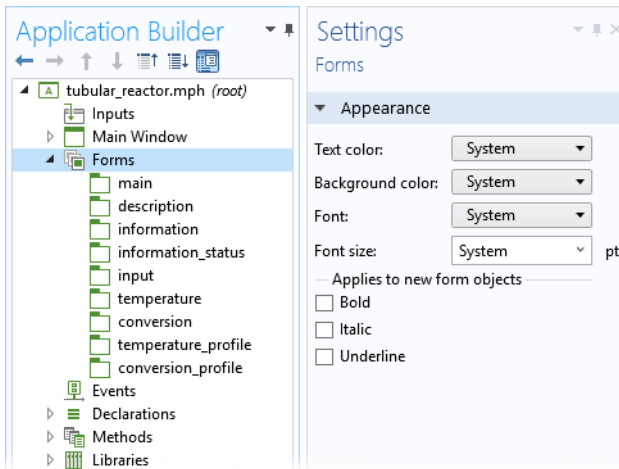
To save an application, from the **File** menu, select **File>Save As**. Browse to a folder where you have write permissions, and save the file in the MPH file format. The MPH file contains all of the information about the application, including information about the embedded model created with the Model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

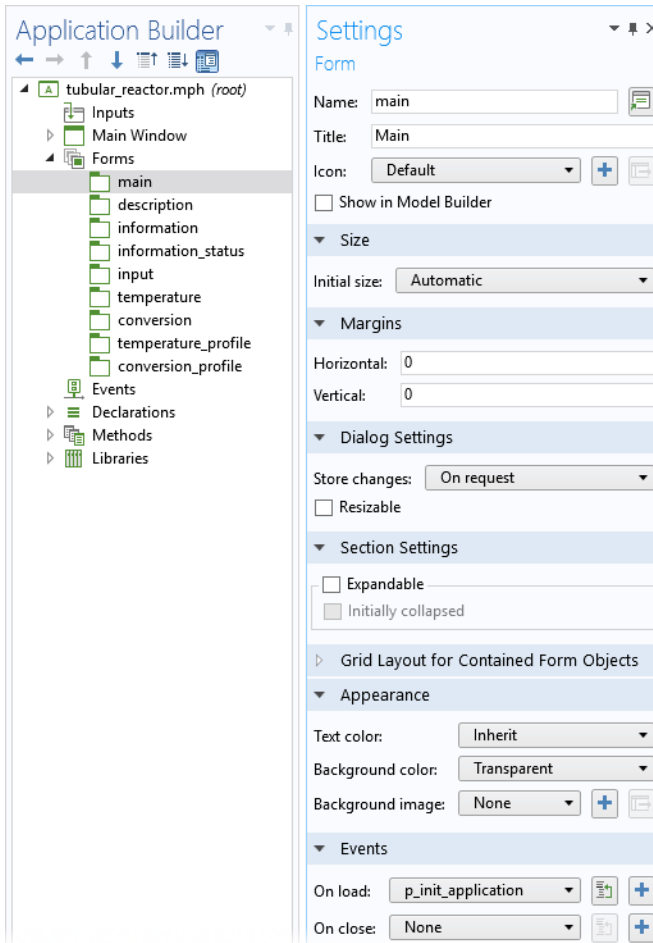


The default is that all new forms and new form objects inherit these settings when applicable.

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.

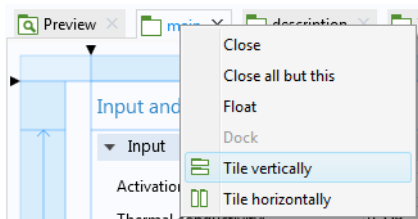


Each form has its own **Settings** window with settings for:

- **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 67.
- **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

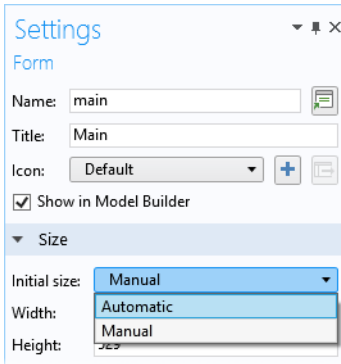
The Application Builder defaults to sketch layout mode, which lets you use fixed object positions and size. The instructions in the section “The Form Editor” assume that the Form editor is in sketch layout mode unless otherwise specified. For information on grid layout mode, see “Sketch and Grid Layout” on page 105.

INITIAL SIZE OF A FORM

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

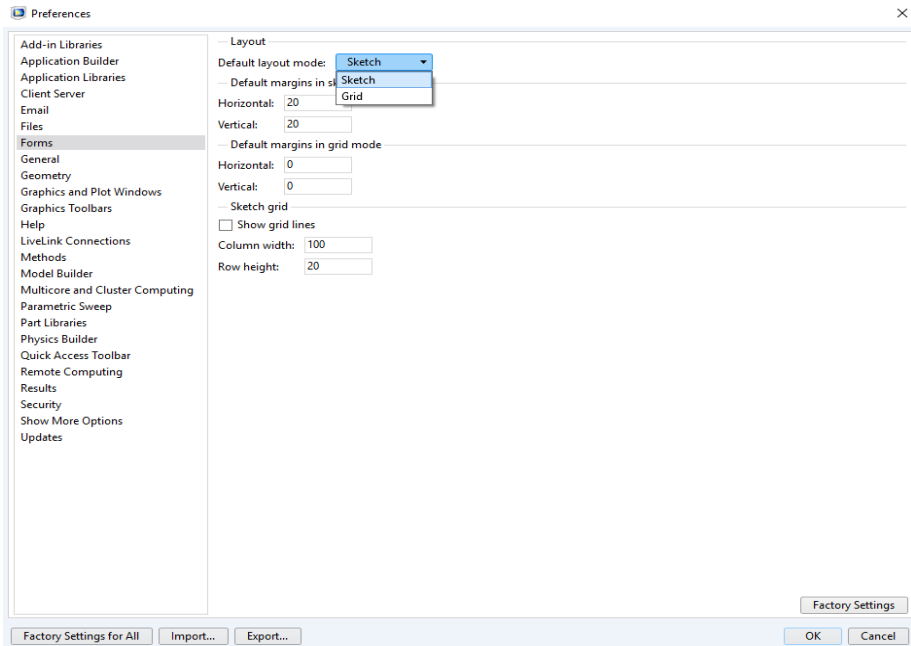
- **Manual** lets you enter the pixel size for the width and height.
- **Automatic** determines the size based on the form objects that the form contains. If you are using grid layout mode and there are columns or rows set to **Grow**, then the size is not defined by the form objects. In this case, the 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 108.



Form Editor Preferences

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



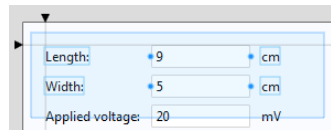
The **Forms** section includes settings for changing the defaults for **Layout mode**, **Margins**, and **Sketch grid**.

Form Objects

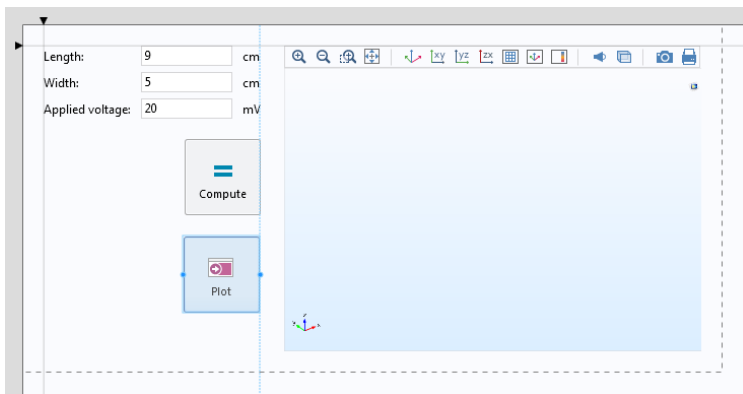
POSITIONING FORM OBJECTS

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

- 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

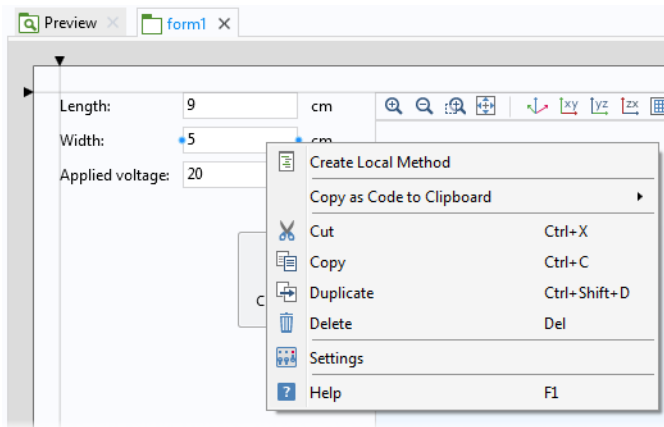
To resize an object:

- 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

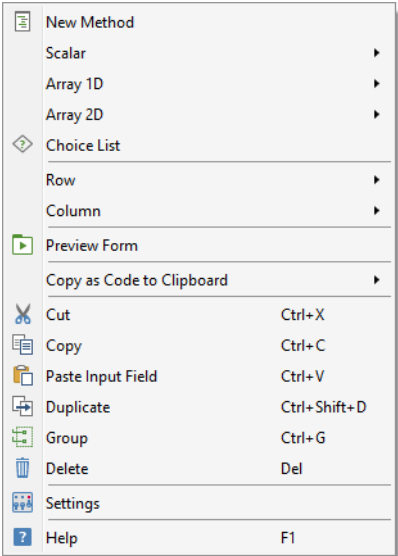
To delete an object, click to select it and then press Delete on your keyboard. You can also click the **Delete** button in the Quick Access Toolbar.

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 an empty area in the form and right-click again. Depending on the copied object, a **Paste** menu option will be

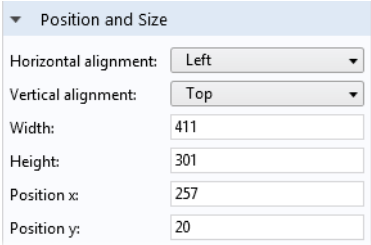
shown. In the figure below, an **Input Field** has previously been copied and as a result, a **Paste Input Field** option is shown.



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.

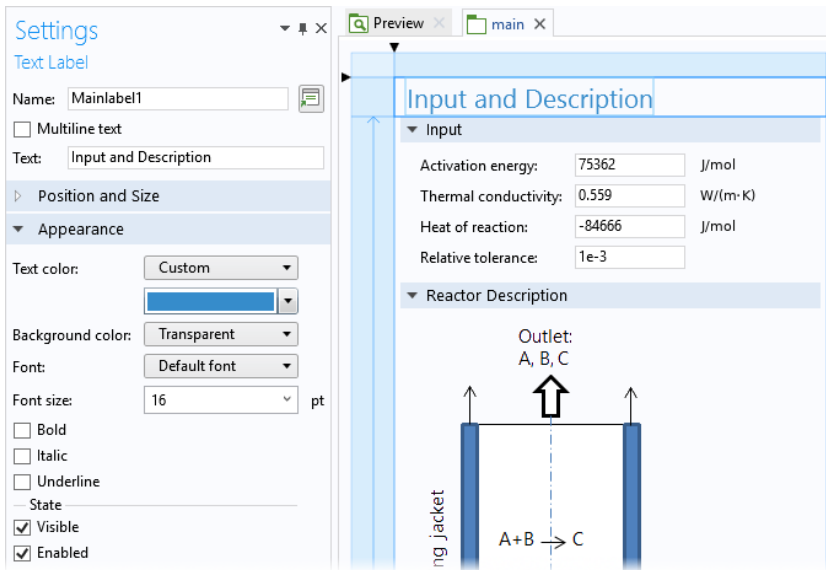


The Position and Size section will have different options depending on the type of form object. For grid layout mode, there are additional settings for the position of the object with respect to rows and columns. For details, see “Sketch and Grid Layout” on page 105.

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, 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.



By using grid layout mode (see “Sketch and Grid Layout” on page 105) you can gain further control over the size of form objects, such as setting an arbitrary size for a button.

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.

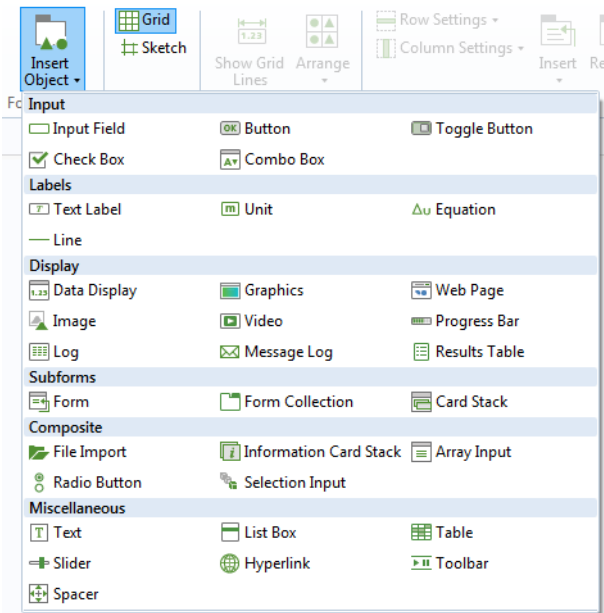
THE NAME OF A FORM OBJECT

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](#), only describes 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 **Data Access** (see page 99), the additional form objects may be added, including:

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

For more information on the check box, combo box, and other form objects, see “Appendix A — Form Objects” on page 217.

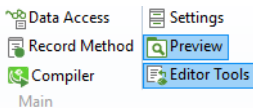
EVENTS AND ACTIONS ASSOCIATED WITH FORMS AND FORM OBJECTS

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 forms or 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**).

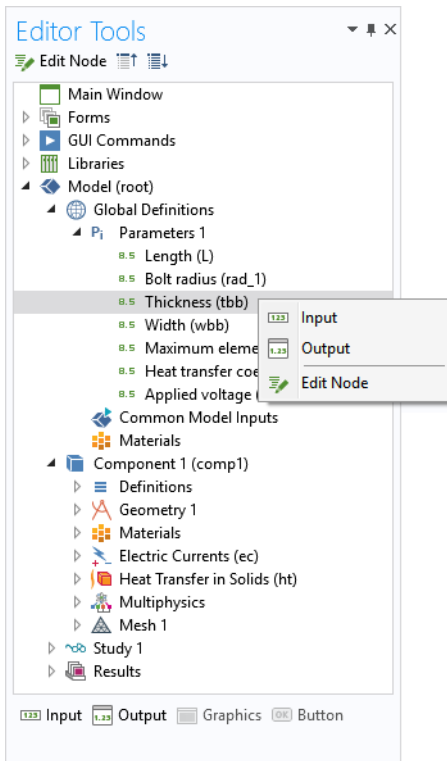
Using Ctrl+Alt+click on a form object opens any associated method in the **Method** editor. If there is no method associated with the form object, a new local method will be created, associated with the form object, and opened in the **Method** editor. If the form object has an associated command sequence, this sequence is converted to code and inserted in the local method.

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. You can also right-click for a list of these options. Depending on the node, the following options are available:

- **Input**

An **Input Field**, **Check Box**, **Combo Box**, or **File Import** object 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 **Check Box** using the selected node as **Source**.
- 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 **Data Access** (see page 99) to make the corresponding node available in the editor tree.
- Inserts a **File Import** object using the selected node as **File Destination**.

- **Output**
 - Inserts a **Data Display** object accompanied by a **Text Label** when applicable.
 - Inserts a **Results Table** object when the selected node is a **Table**.
- **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.

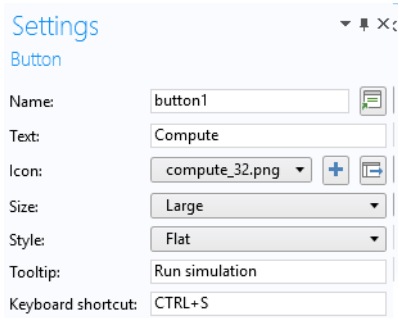
The Editor Tools window is also an important tool when working with the Method editor. In the Method editor, it is used to generate code associated with the nodes of the editor tree. For more information, see “Editor Tools in the Method Editor” on page 171.

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:

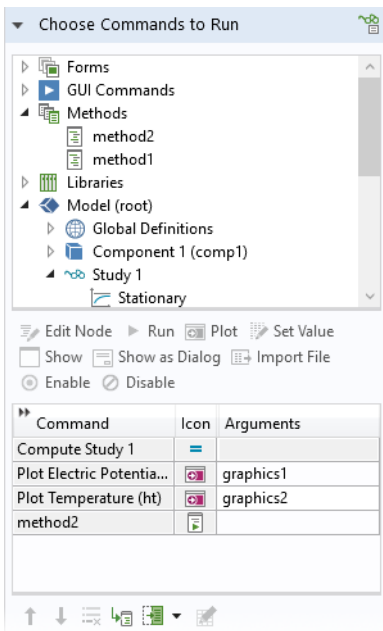
- Edit the form object **Name** of the button.
- Edit the **Text** displayed on the button.
- Use an **Icon** instead of the default rendering of a button.
- Set the button **Size** to **Large** or **Small**.
- Set the button **Style** to **Flat** or **Raised**.

- 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.



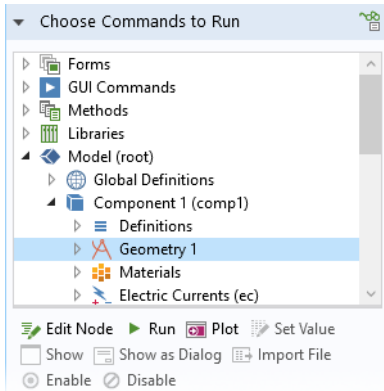
CHOOSING COMMANDS TO RUN

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 77, “The Main Window” on page 127, “Table” on page 286, and “Toolbar” on page 295.

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 first builds and then plots the geometry. The option **Edit Node** will take you to the corresponding node in the model tree or the application tree.



⚠ You do not need to precede a **Plot Geometry** command with a **Build Geometry** command (that you get by clicking **Run**). The **Plot Geometry** command will first build and then plot the geometry. In a similar way, the **Plot Mesh** command will first build and then plot the mesh.

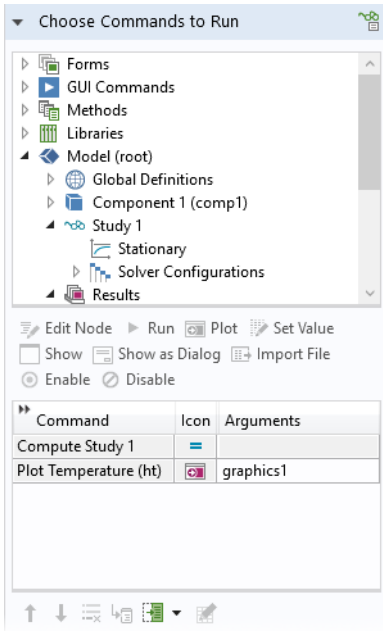
The command icons highlighted for selection are those applicable to the particular tree node. This is a list of the command icons that may be available, depending upon the node:

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

- **Enable**
- **Disable**

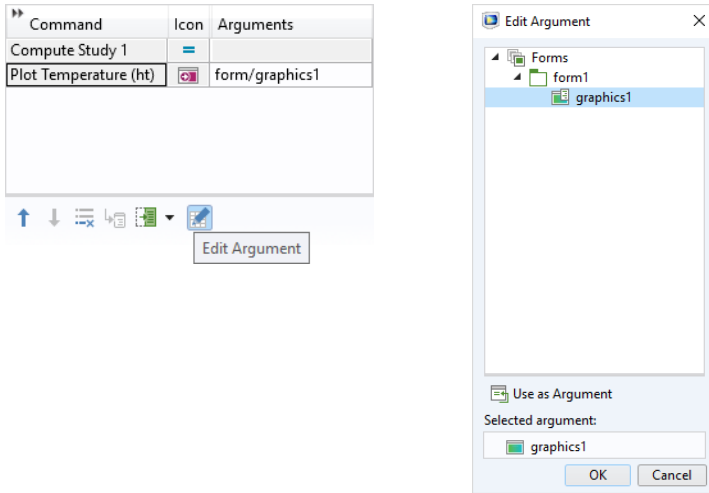
Some commands, such as the various plot commands, require an argument. The argument to a plot command, for example, defines which of the different graphics objects the plot should be rendered in.

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.

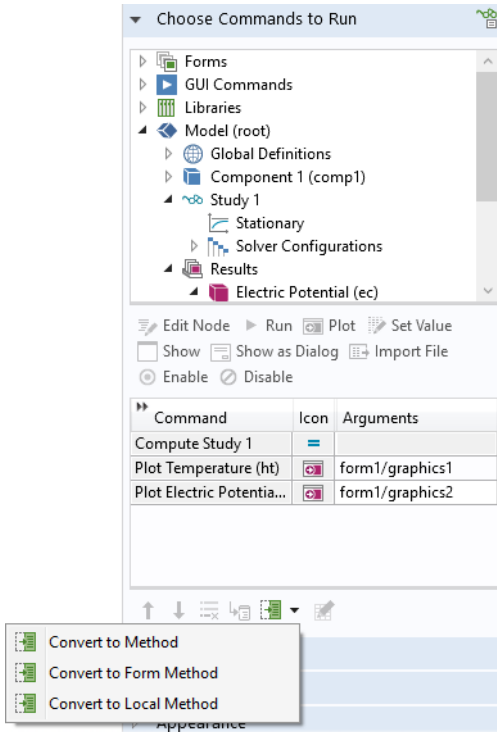


To reference graphics objects in a specific form, the following syntax is used: /form1/graphics2, /form3/graphics1, etc. If a specific form is not specified, for example, graphics1, then the form where the button is located is used.

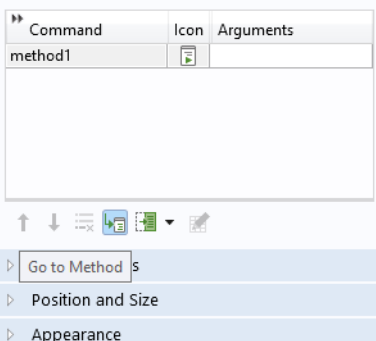
To control the order and contents of the sequence of commands, use the **Move Up**, **Move Down**, and **Delete** buttons located below the command sequence table.

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 Method**.



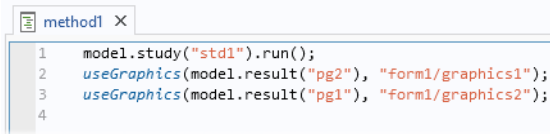
Open the new method by clicking **Go to Method**.





You can also create a method that is local to a form or form object by clicking **Convert to Form Method** or **Convert to Local Method**, respectively.

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");
4
```

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");
```

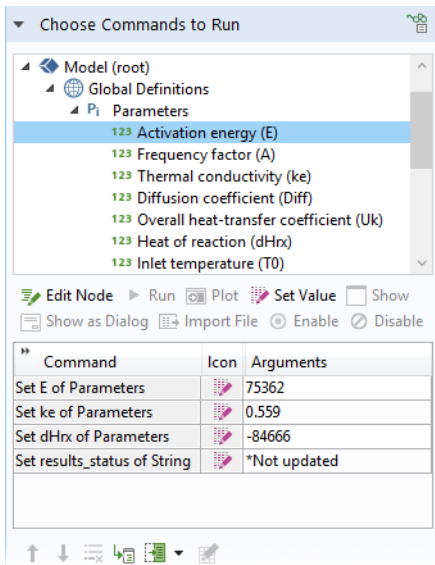
use the built-in method `useGraphics` to display plots corresponding to plot groups `pg1` and `pg2`, respectively. In this example, the plots are displayed in two different graphics objects, `graphics1` and `graphics2`, respectively.

For more information on methods, see “The Method Editor” on page 165.

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 **Data Access** (see

page 99). The figure below shows a command sequence used to initialize a set of parameters and a string variable.

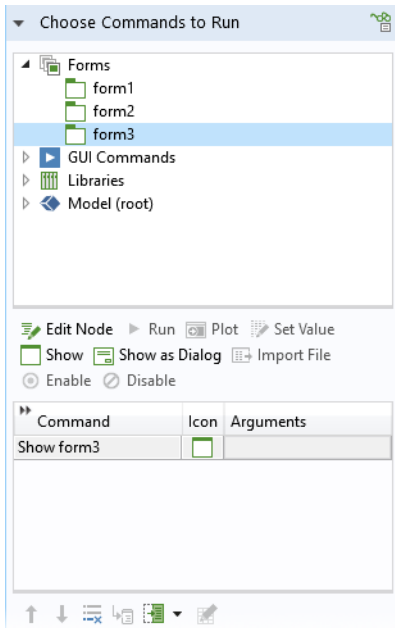


To learn how to perform the same sequence of operations from a method, select **Convert to Method** under the command table.

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 request 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**.

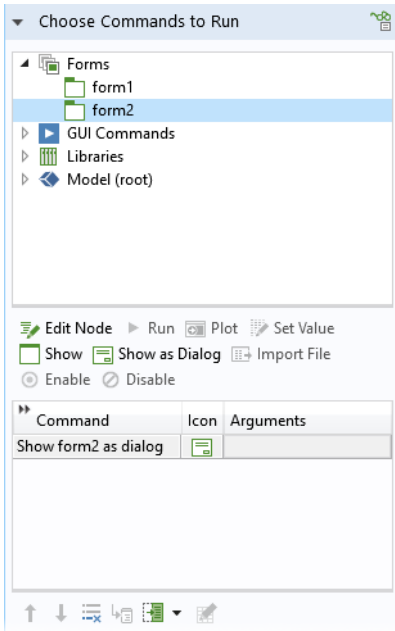


This command will leave the form associated with the button and make the specified form visible to the user.

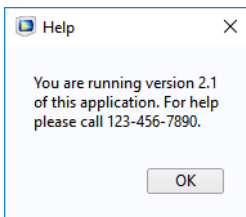
SHOWING A FORM AS A DIALOG BOX

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

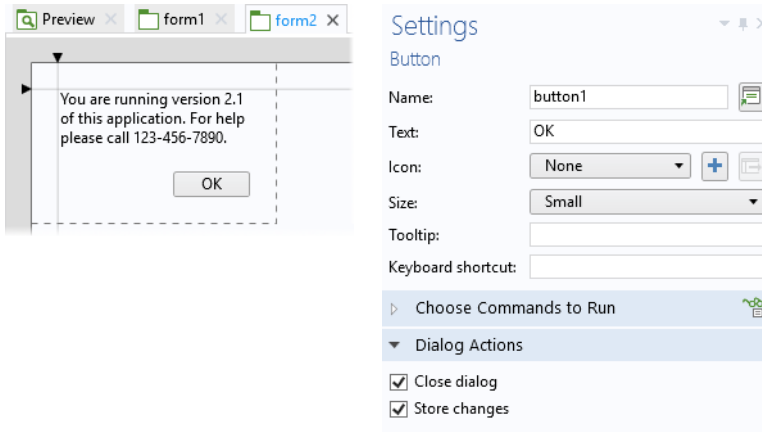
shows an example of the settings for a button with the command **Show form2 as dialog**.



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.

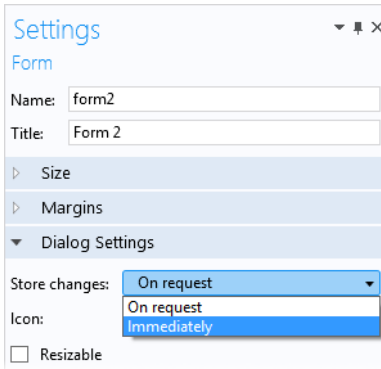
Typical dialog box buttons and their associated dialog actions are:

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

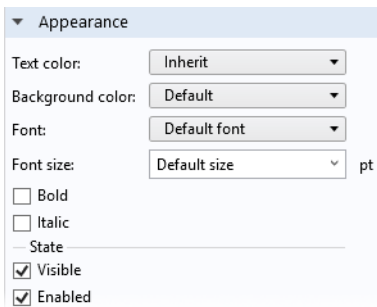
whether to store data **On request** or **Immediately** when the change occurs, as shown in the figure below.



When the **Store changes** option **On request** is selected, the variables that have been changed by the user in the dialog box will not be updated until the OK button (or similar) in the dialog box has been clicked. This requires that the check box **Store changes** is selected, in the **Settings** window of the OK button. When the option **Immediately** is selected, variables changed by the user in the dialog box is updated immediately including while the dialog box is still open.

APPEARANCE

In the **Settings** window for a button, the **Appearance** section contains font settings as well as settings that control the state of the button object.



Changing the Enabled and Visible State of a Form Object

Whether or not the button object should be **Visible** or **Enabled** is controlled from the check boxes under the **State** subsection. The **Appearance** section for most form objects has similar settings, but some have additional options; for example, input field objects.

A button, or another form object, with the **Visible** check box cleared will not be shown in the user interface of the running application. A form object with the **Enabled** check box cleared will be disabled, or “grayed out”, but still visible. The state of a form object can also be controlled using built-in methods. For example, assume that a Boolean variable `enabled_or_disabled` is used to determine the enabled/disabled state of a button with **Name** `button3`. In this case, you can control the state of the button as follows:

```
setFormObjectEnabled("button3", enabled_or_disabled);
```

In a similar way, the call

```
setFormObjectVisible("button3", visible_or_not);
```

lets a Boolean variable `visible_or_not` control whether the button is shown to the user or not.

For more information, see “GUI-Related Methods” on page 330 and the *Application Programming Guide*.

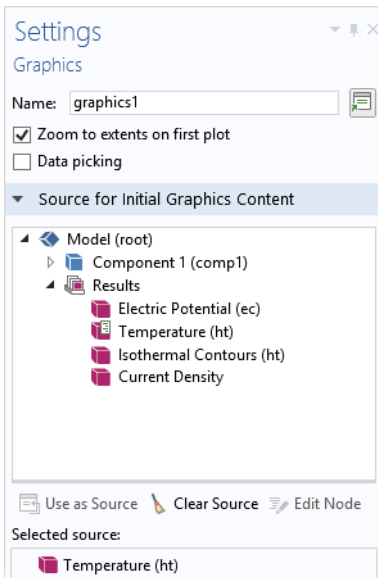
Graphics

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 or animation 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 starts. 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 **Animation**, **Selection**, **Geometry**, and **Mesh** nodes as the **Selected source**.

Selecting the check box **Zoom to extents on first plot** ensures that the first plot that appears in the graphics canvas shows the entire model (zoom extents). This action is triggered once the first time that graphics content is sent to the graphics object.

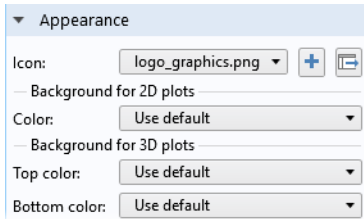
Selecting the check box **Data picking** makes the graphics object interactive so that you can, for example, click on a plot at a particular point and retrieve a numerical value for the temperature at that coordinate. For more information, see “Data Picking” on page 86.

APPEARANCE

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

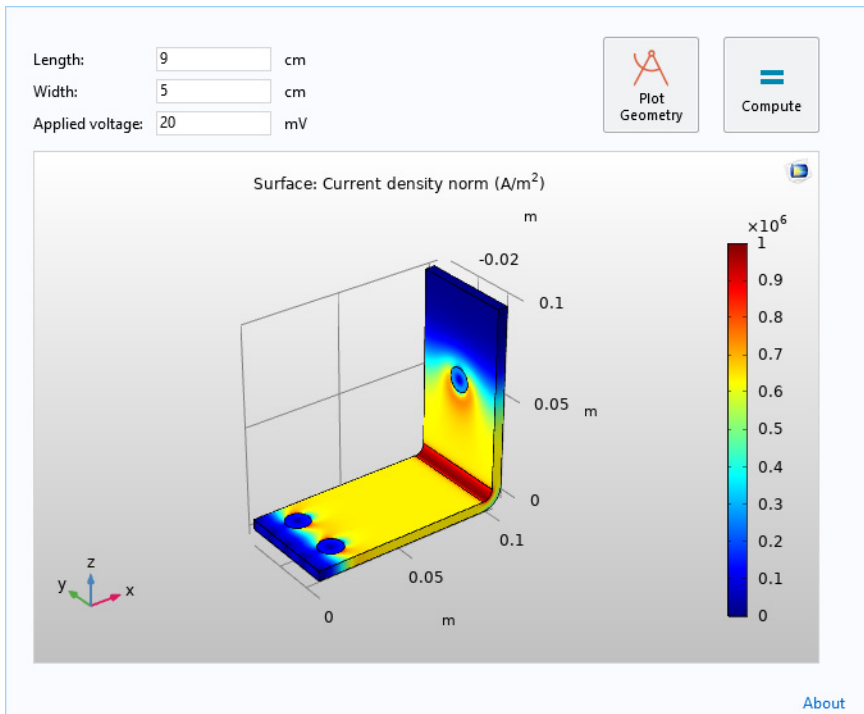
- Include an **Icon**, such as a logo image, in the upper-right corner.

- 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**.



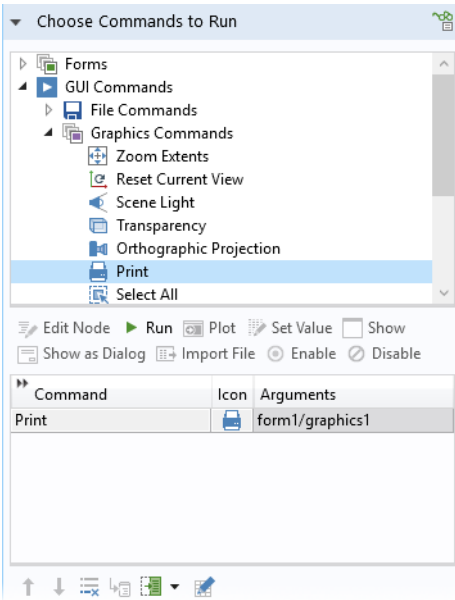
In addition, the subsection **State** (not shown in the figure above) contains settings for the visible and enabled state of the graphics object. For more information, see “Changing the Enabled and Visible State of a Form Object” on page 70.

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:

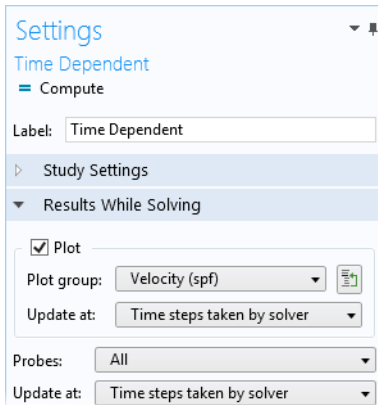
- **Zoom Extents**
 - Makes the entire model visible.
- **Reset Current View**
 - Resets the currently active view to the state it had when the application was launched; also see “Views” on page 80.
- **Scene Light**
 - Toggles the scene light (on or off).
- **Transparency**
 - Toggles the transparency setting (on or off).
- **Orthographic projection**
 - Enable orthographic projection (as opposed to perspective projection).
- **Print**
 - Prints the contents of the graphics object.

- **Select All**
 - Selects all objects.
- **Clear Selection**
 - Clear the selection of all objects.
- **Show Selection Colors**
 - Enable visualization of selection colors.
- **Show Material Color and Texture**
 - Enable visualization of material color and texture.

Note that the commands **Zoom Extents**, **Reset Current View**, **Scene Light**, **Transparency**, **Show Selection Colors**, **Show Material Color and Texture**, and **Print** have corresponding toolbar buttons in the standard graphics toolbar. See the next section “Graphics Toolbar”.

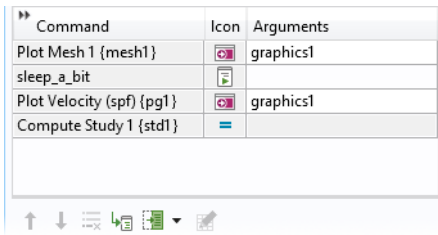
Plot While Solving





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.



You can include a method that calls the built-in `sleep` method for briefly displaying graphics information before switching to displaying other types of

graphics. Insert it in a command sequence after a plot command, as shown in the figure below.



Command	Icon	Arguments
Plot Mesh 1 {mesh1}		graphics1
sleep_a_bit		
Plot Velocity (spf) {pg1}		graphics1
Compute Study 1 {std1}		

In this example, the `sleep_a_bit` method contains one line of code:

```
sleep(1000); // sleep for 1000 ms
```

For more information on the method `sleep`, see “sleep” on page 335.

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 257), 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 does not use subforms.

If you need to use methods to change a plot, use the `useGraphics` method. For more information on writing methods, see “The Method Editor” on page 165.

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;
}
```

CLEARING THE CONTENTS OF A GRAPHICS OBJECT

You can clear the contents of a graphics object by a call to the `useGraphics` method, such as:

```
useGraphics(null, "/form1/graphics1")
```

which clears the contents of the graphics object `graphics1` in the form `form1`.

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.

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**). In addition, you can choose between the options **Small** or **Large** for **Icon size**, **Background color**, and whether to **Include standard toolbar items** or not..

Name	Icon	Text	Tooltip
------	------	------	---------

Graphics Toolbar for Geometry and Mesh

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



Graphics Toolbar for Selection

When the **Source for Initial Graphics Content** is set to an **Explicit** selection, the graphics toolbar will contain three additional items: **Zoom to Selection**, **Select Box**, and **Deselect Box**. This is shown in the figure below.



For more information on selections, see “Selections” on page 83.

Graphics Toolbar for Plot Groups

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



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 for results with four additional buttons to the right.



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

▼ **Toolbar**


Position:


Icon size:


Background color:


— Standard toolbar —


Include standard toolbar items:


 Zoom:


 Go to view:

 Rotate:

 Select box:






 Deselect box:







 View:

 Image:

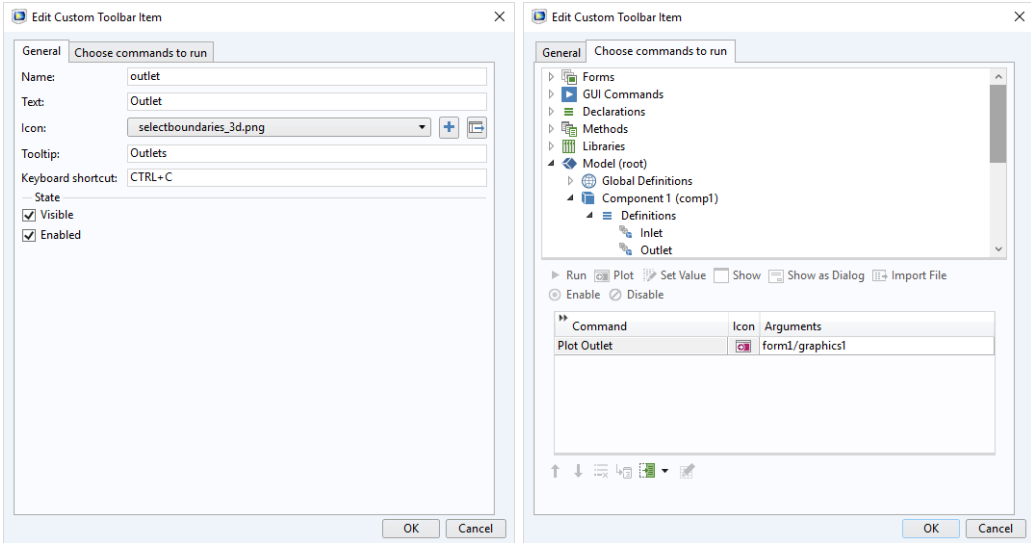
Place standard toolbar before custom items

— Custom toolbar items —

Name	Icon	Text	Tooltip
separator1		-	
geometry			Geometry
inlet			Inlet
outlet			Outlet
flow_field			Flow field

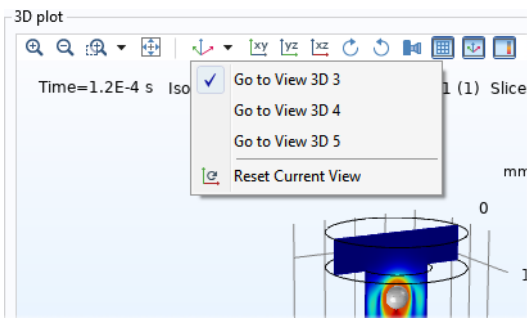
To edit the command sequence for a toolbar item, click the **Edit** button to open the **Edit Custom Toolbar Item** dialog box.



This dialog box has settings that are similar to those of a button or a toolbar item with the contents divided into two or three tabs depending on if the item is a toggle item or not. For details, see “Button” on page 59 and “Toolbar” on page 295.

Views

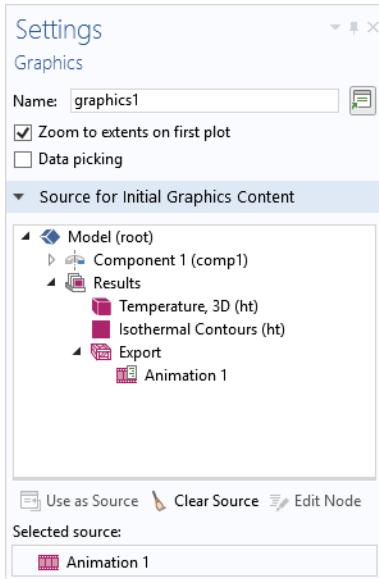
In the graphics toolbar, the **Go to Default View** button (for 3D graphics only) will display a menu with all applicable views. The currently active view is indicated with a check mark.



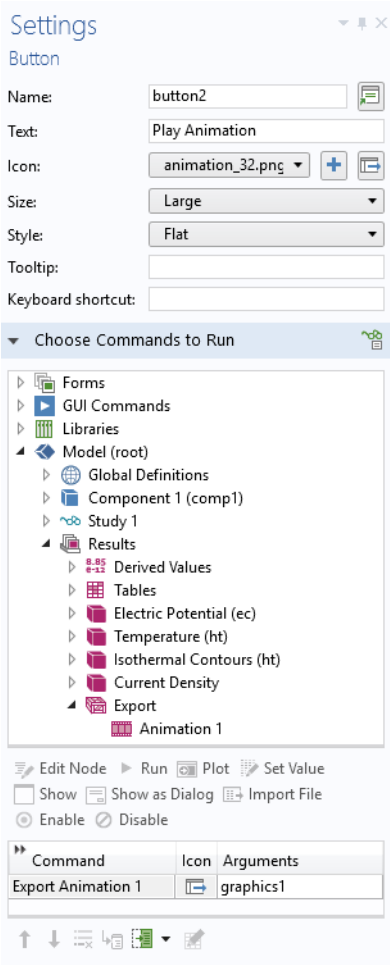
In addition to a list of all views, there is an option **Reset Current View** that will reset the currently active view to the state it had when the application was launched.

ANIMATIONS

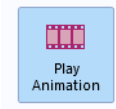
You can display animations in an application by using a **Results > Animation** node as the **Source for Initial Graphics Content**.



To run the animation, use the New Form wizard or the Editor Tools window to create a command from, for example, a button that runs a **Results > Animation** node.



When using the New Form wizard or Editor Tools, the animation button will have the following default appearance:

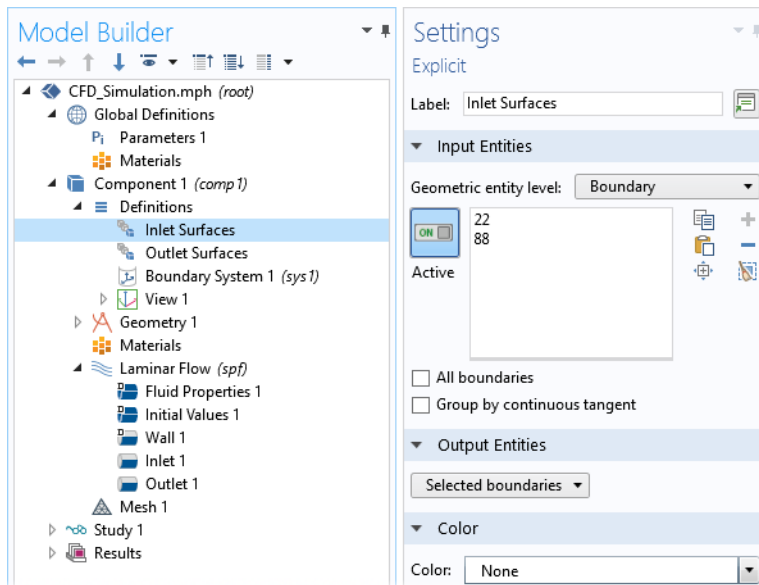


SELECTIONS

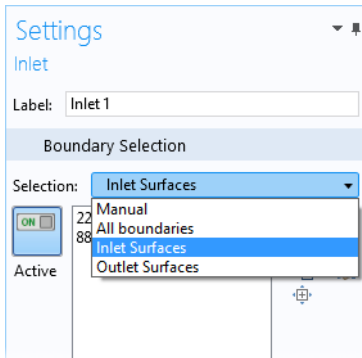
Selections in the Model Builder

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.

As an example of how selections can be used, consider selections for boundary conditions. When you select which boundaries should be associated with a certain boundary condition, you can click directly on those boundaries in the graphics window of the COMSOL Desktop environment. This is the default option called **Manual** selection (see below). These boundaries will then be added to a selection that is local to that boundary condition. Named selections instead let you define global selections that can be reused for several different kinds of boundary conditions by just selecting from a drop-down list. The figure below shows an **Explicit** selection given the name **Inlet Surfaces** with two associated boundaries (22 and 88).



The figure below shows the **Settings** window for an **Inlet** boundary condition where the **Inlet Surfaces** selection is used. In this example, there is also an Outlet Surfaces selection.

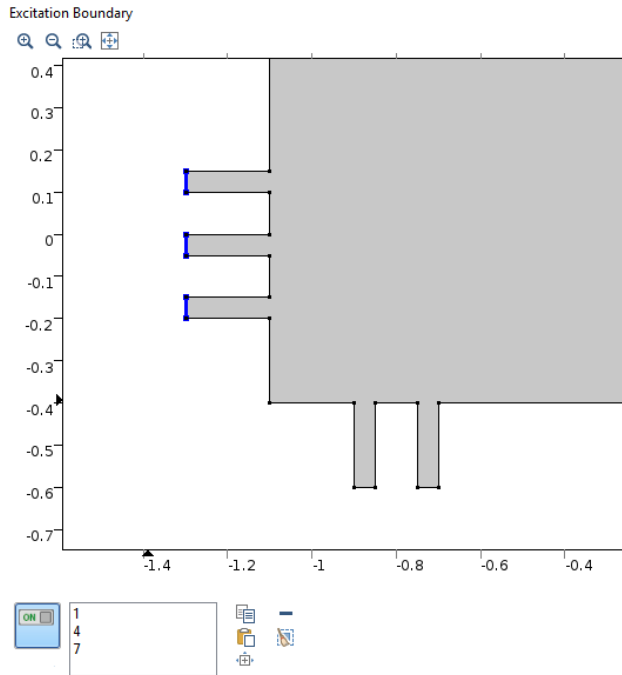


For convenience, in addition to the **Manual** option, there is also a shortcut for **All boundaries**.

Selections in the Application Builder

The **Explicit** selection type lets you group domains, boundaries, edges, or points based on entity number, and is the type of selection most readily available for use with 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**

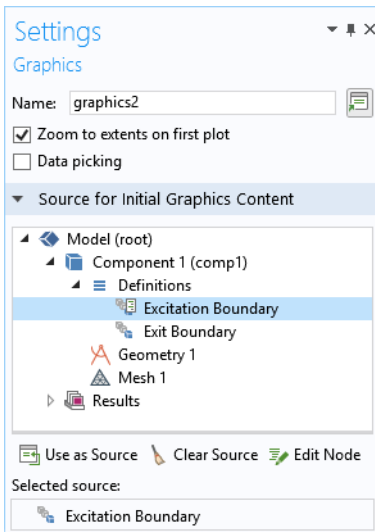
object and a **Graphics** object are used to let the user select boundaries to be excited by an incoming wave.



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, 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 a 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 selection.

To make it possible to select by number, you can link a **Selection Input** object to an explicit selection. For more information, see “Selection Input” on page 277.

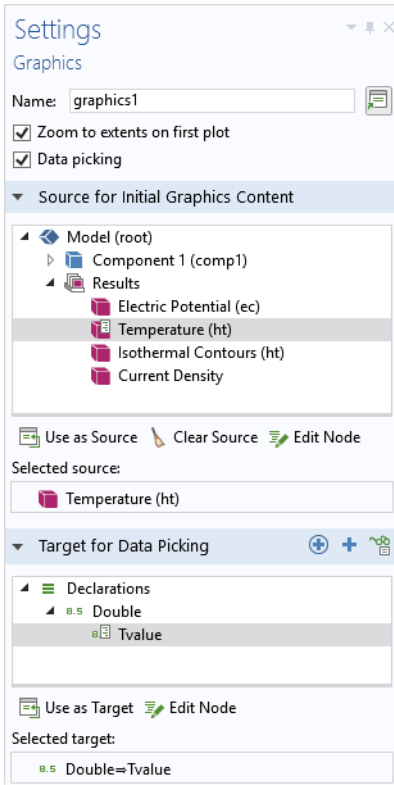
The **Editor Tools** window provides a quick way of adding a **Graphics** object or a **Selection Input** object that is linked to an **Explicit** selection. To get these options, right-click an **Explicit** selection node in the editor tree.

You can let a global **Event** be triggered by an **Explicit** selection. This allows a command sequence or method to be run when the user clicks a geometry object, domain, face, edge, or point. For more information on using global events, see “Events” on page 134 and “Source For Data Change Event” on page 137.

DATA PICKING

In the **Settings** window for a graphics object, select the check box **Data picking** to make the graphics object interactive so that you can, for example, click on a plot at a particular point and retrieve a numerical value for the temperature at that coordinate. In the figure below, in the section **Target for Data Picking**, a scalar double variable `Tva1ue` is selected. This variable is declared under the **Declarations**

node. In the running application, the value of the temperature at the pointer position will be stored in the variable Tvalue.



If the **Target for Data Picking** is a 1D double array, then the stored value will instead correspond to the x, y (2D) or x, y, and z coordinates at the clicked position.

The **Target for Data Picking** can be any one of the following:

- Scalar double variable
- 1D double array
- Domain Point Probe
- Boundary Point Probe
- Graphics Data declaration

For more information on Graphics Data declaration, see “Graphics Data” on page 161.

Input Field

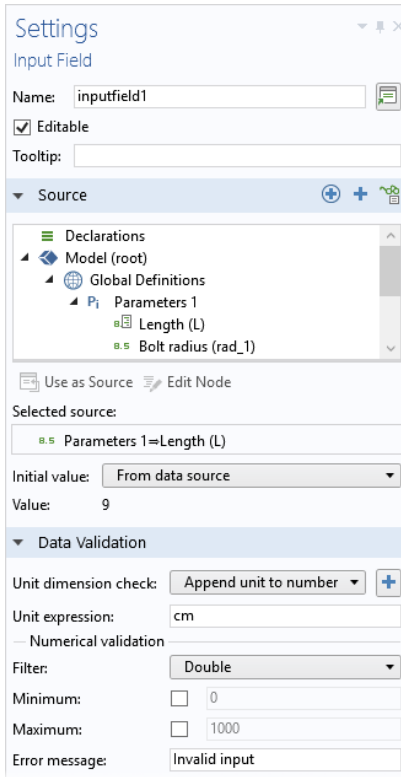
An **Input Field** allows a user to change the value of a parameter or variable. In the New Form wizard, when a parameter or variable is selected, three form objects are created:

- 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.

By selecting a parameter or variable using the **Editor Tools** window, the same three form objects are created.

Assuming you do not use the **Editor Tools** window: 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.

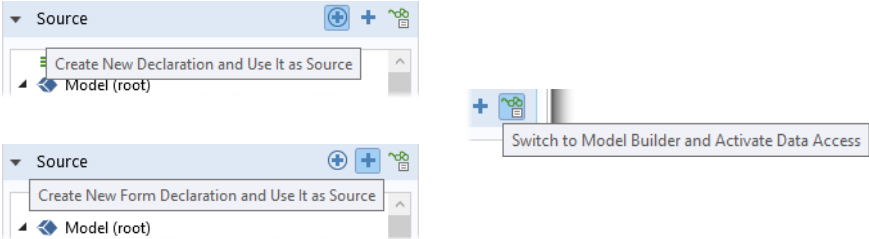


In addition to parameters and variables, input fields can use an **Information** node as **Source**.

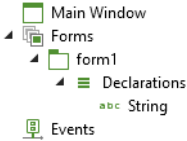
The default setting for the **Initial value** is **From data source**. This means that if the source is a parameter, then the initial value displayed in the input field is the same as the value of the parameter as specified in the **Parameters** node in the Model Builder. The other **Initial value** 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 buttons for easy access to tools that are used to make additional properties and variables available as sources to the input field.



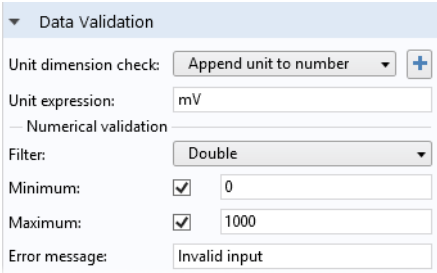
The **Create New Declaration and Use It as Source** button can be used to add new variables under the **Declarations** node. For more information, see “Declarations” on page 141. The **Create New Form Declaration and Use It as Source** button can be used to add new variables under the **Declarations** nodes local to forms, as shown below.



The **Switch to Model Builder and Activate Data Access** button can be used to access low-level model properties as described in the next section. For more information on **Data Access**, see “Data Access in the Form Editor” on page 99.

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.



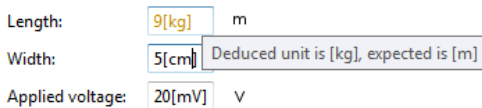
When creating an input field in the New Form wizard, the setting **Append unit to number** is used when applicable. This setting assumes that a user enters a number into the input field, but it can also handle a number followed by a unit using the

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:

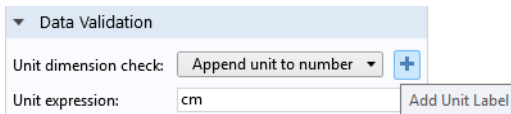
- **None**
- **Compatible with physical quantity**
- **Compatible with unit expression**
- **Append unit to number** (default)
- **Append unit from unit set**

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.



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.

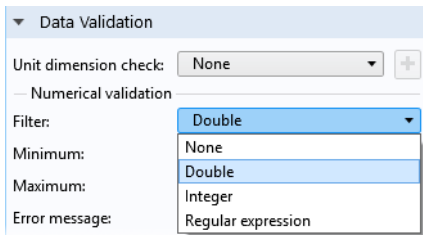


Clicking this button will add a unit label to the right of the input field if there is not already a unit label placed there.

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

Numerical Validation

The options **Append unit to number**, **Append unit from unit set**, and **None** allow you to use a filter for numerical validation of the input numbers.



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

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

The **Filter** list for the options **Append unit to number** and **Append unit from unit set** 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. You may use global parameters in these fields. If global parameters are used, you can define such parameters with or without units. If you use global parameters without a unit, then only the numerical value of these parameters is considered when they are used as **Minimum** and **Maximum** values. For example, consider data validation of an input field for a length parameter L with unit cm . Further, assume that a global parameter L_{max} is used as the **Maximum** value. If you would like the maximum value of L to be 15 cm , then the following values for the parameter L_{max} will work: 15 (with no unit), $15[\text{cm}]$, $0.15[\text{m}]$, $150[\text{mm}]$, etc.

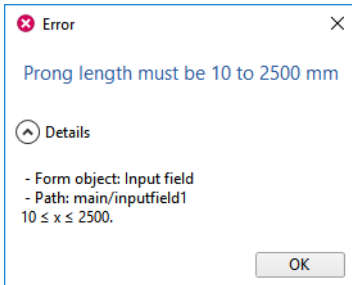
For the **Append unit from unit set** option, the **Minimum** and **Maximum** values are always with respect to the **Initial value** for the unit set by the unit set. For more information on unit sets, see “Unit Set” on page 154.

The **Regular expression** option, available when the **Unit dimension check** is set to **None**, 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”.

For more advanced requirements, note that virtually any kind of validation of the contents of an input field can be made by calling a method using the **Events** section in the **Settings** window of an input field.

Error Message

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

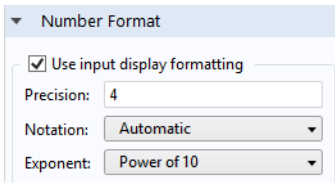


The debugging information typically consists of the type of form object, the path to the form object, and the reason for the failure; for example, $10 \leq x \leq 2500$.

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

NUMBER FORMAT

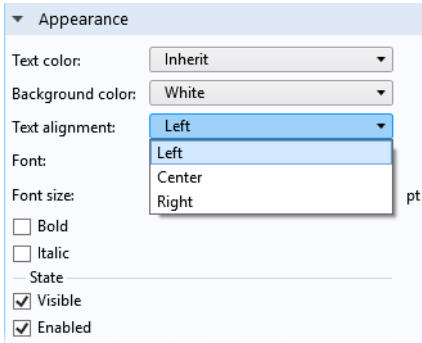
The **Number Format** section contains a check box **Use input display formatting**. If selected, it enables the same type of display formatting as a **Data Display** object.



For more information, see “Data Display” on page 96.

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.



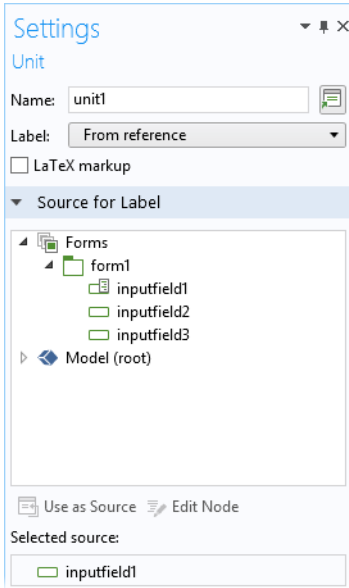
The image shows a settings panel titled "Appearance" with a dropdown arrow on the left. The panel contains the following options:

- Text color: Inherit (dropdown)
- Background color: White (dropdown)
- Text alignment: Left (dropdown menu is open, showing options: Left, Center, Right)
- Font: (dropdown menu is open, showing options: Left, Center, Right) pt
- Font size: (input field)
- State section:
 - Bold
 - Italic
 - Visible
 - Enabled

Whether the input field should be **Visible** or **Enabled** is controlled from the check boxes under the **State** subsection. For more information, see “Changing the Enabled and Visible State of a Form Object” on page 70.

Unit

In the **Settings** window for a **Unit** object, you can set the unit to a fixed string, or link it to an source, such as an input field. The figure below shows the **Settings** window for a unit object.

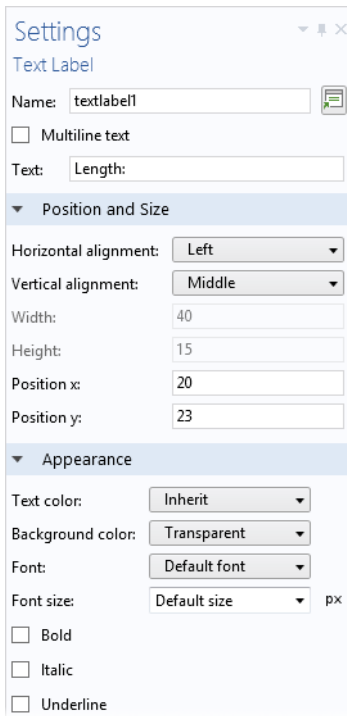


When adding an input field using the New Form wizard, a unit object is automatically added when applicable. By default, the unit is displayed using Unicode rendering. As an alternative, you can use LaTeX rendering by selecting the **LaTeX markup** check box. Then, the display of units will not depend 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**. The contents of the section **Position and Size** will change depending on if you are working in sketch layout mode or grid layout mode.

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 shown below. Valid parameters, variables, and properties include:

- The output from a **Derived Values** node, such as a **Global Evaluation** or a **Volume Maximum** node
- Variables declared under the **Declarations > Scalar, 1D Array, and 2D Array** nodes
- Properties made available by using the **Data Access** tool; See “Data Access in the Form Editor” on page 99
- 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 **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:

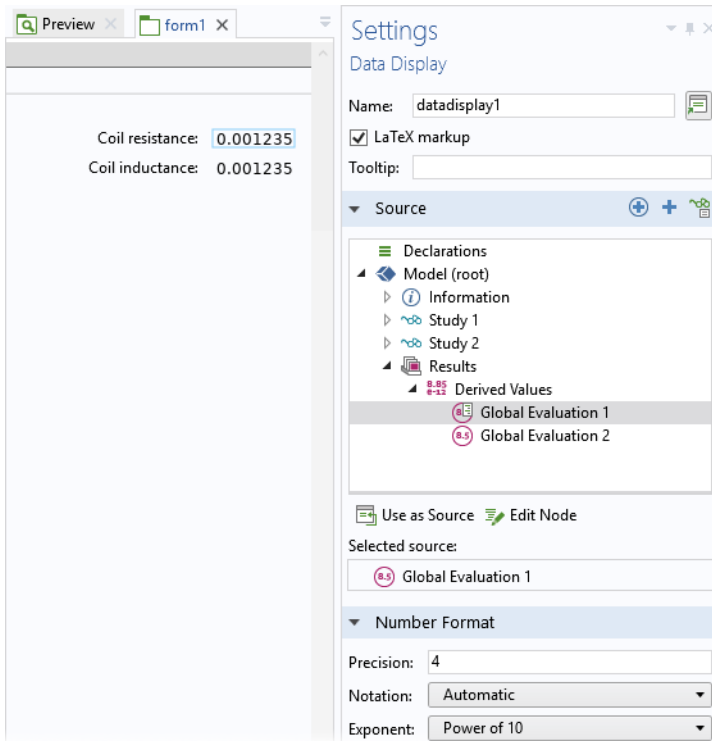
- a **Text Label** object for the **Description** of the variable
- a **Data Display** object for the value of the variable

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.

A formatted display of arrays and matrices is only supported with LaTeX rendering. The figure below shows a 2D double array (see page 149) displayed using a **Data Display** object with **LaTeX markup** selected.

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

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

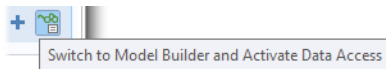
Data Access in the Form Editor

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, and parts of the application 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 and application trees that are not made available by default, because there may be 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 **Data Access**.

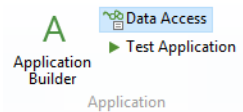
The remainder of this section gives an introduction to using **Data Access**, with examples for input fields and buttons.

DATA ACCESS FOR INPUT FIELDS

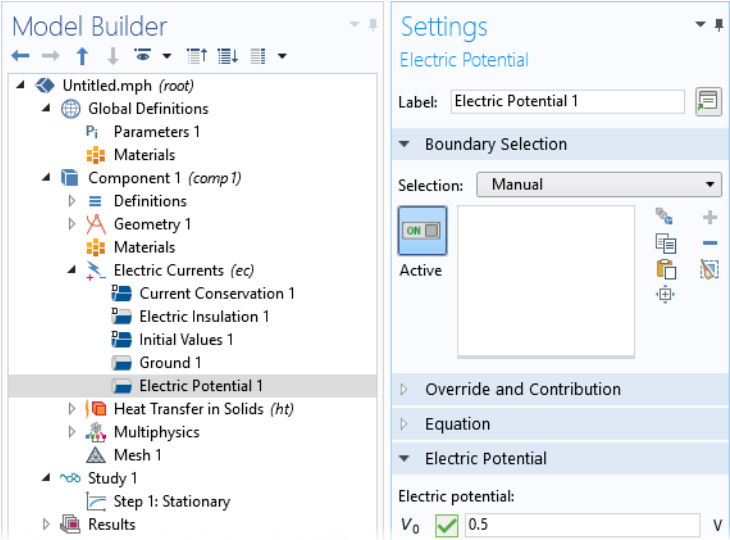
By default, you can link input fields to parameters and variables defined in the model tree under the **Parameters** or **Variables** nodes and to variables declared in the application tree under the **Declarations** node. To access additional model tree node properties, click the **Switch to Model Builder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



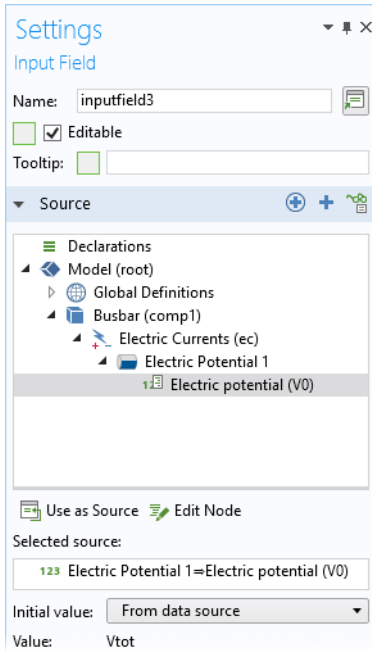
You can also access it from the **Application** group of the **Developer** tab of the Model Builder.



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:



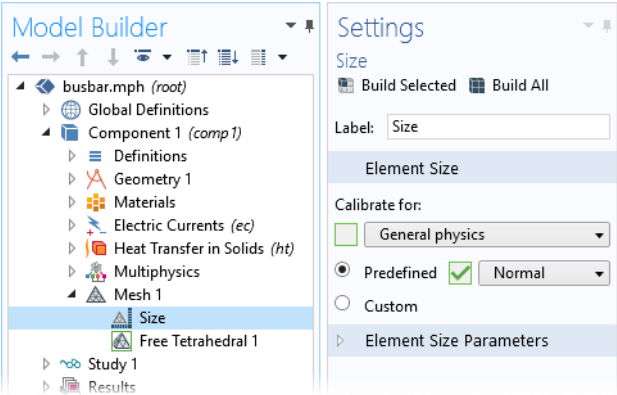
The figure below shows the **Settings** window for an input field. The list of possible sources for this field now contains the **Electric potential**.



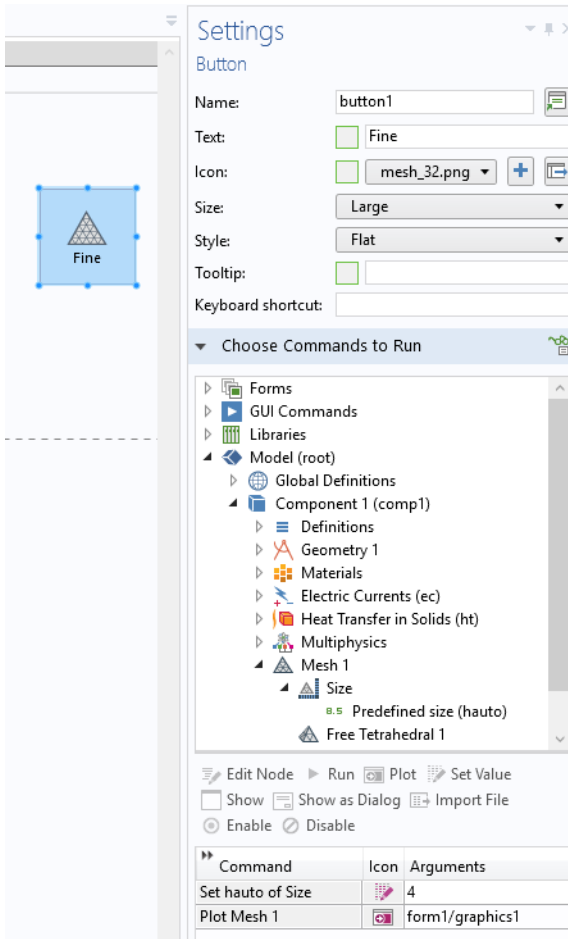
In addition, as seen in the figure above, **Data Access** makes it possible to access the check box **Editable** and the **Tooltip** text of the input field form object. In addition to the settings of the Model Builder, **Data Access** lets you access certain properties of the Application Builder.

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. The settings shown in the figure below are available when, in the **Settings** window of the **Mesh** node, the **Sequence type** is set to **User-controlled mesh**. In this

example, 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 the above example, a **Set Value** command is used to set the value of the **Predefined mesh size (hauto)** property. The property **Predefined mesh size (hauto)** corresponds to the following settings in the **Size** node shown earlier:

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

The value of the `hauto` property is a double and can take any positive value. For non-integer values, linear interpolation is used for the custom mesh parameters. You can, for example, let a slider object adjust the predefined mesh size. For more information on the slider object, see “Slider” on page 291.

In general, for individual model tree properties, you can quickly learn about their allowed values by recording code while changing their values and then inspecting the automatically generated code. For more information, see “Recording Code” on page 175.

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 225.

SUMMARY OF DATA ACCESS

The table below summarizes the availability of **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, Menu Toggle Item, and Ribbon Toggle Item	Source and Choose Commands to Run
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

FORM OBJECT, EVENT, OR ITEM	SECTION IN SETTINGS WINDOW
Ribbon Item	Choose Commands to Run
Event (Global)	Choose Commands to Run Source for Data Change Event

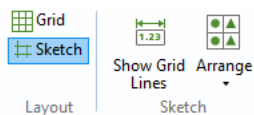
A global event, menu, ribbon, or toolbar item provides a **Choose Commands to Run** section in its **Settings** window, to which the functionality described above in the section on buttons also applies. Global events and many form objects provide a **Source** section in its **Settings** window, and the 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 77, “The Main Window” on page 127, “Events” on page 134, “Table” on page 286, and “Toolbar” on page 295.

Sketch and Grid Layout

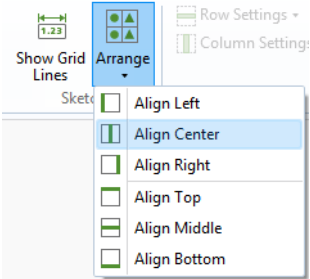
The Form editor provides two layout modes for positioning form objects: sketch layout mode and grid layout mode. The default is sketch layout mode, which lets you use fixed positions and sizes of objects in pixels. Use grid layout mode to position and size objects based on a background grid with cells. 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 to be 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.



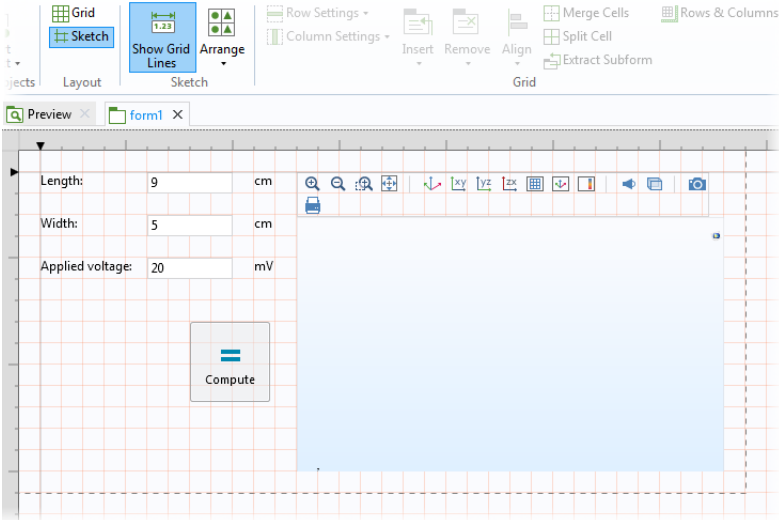
The **Sketch** group in the **Form** tab has two options: **Show Grid Lines** and **Arrange**. The **Arrange** menu allows you to align groups of form objects relative to each other.



Sketch Grid

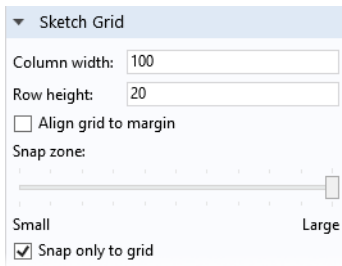
The **Show Grid Lines** option displays a sketch grid to which objects are snapped. Note that the grid used in sketch layout mode is different from the grid used in grid layout mode. The default setting for sketch layout mode is to show no grid lines. Without grid lines visible, a form object being dragged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



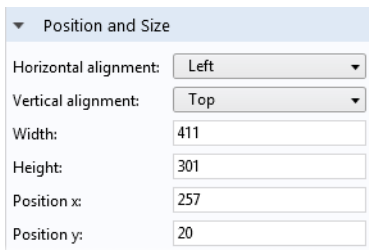
In the **Settings** window of the form, you can change the settings for the sketch grid:

- **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.

Button and toggle button form objects have an **Automatic** and **Manual** option for the **Width** and **Height** values. The **Manual** option allows for pixel-based input and the **Automatic** option adapts the size of the button to the size of the **Text** string.

GRID LAYOUT

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

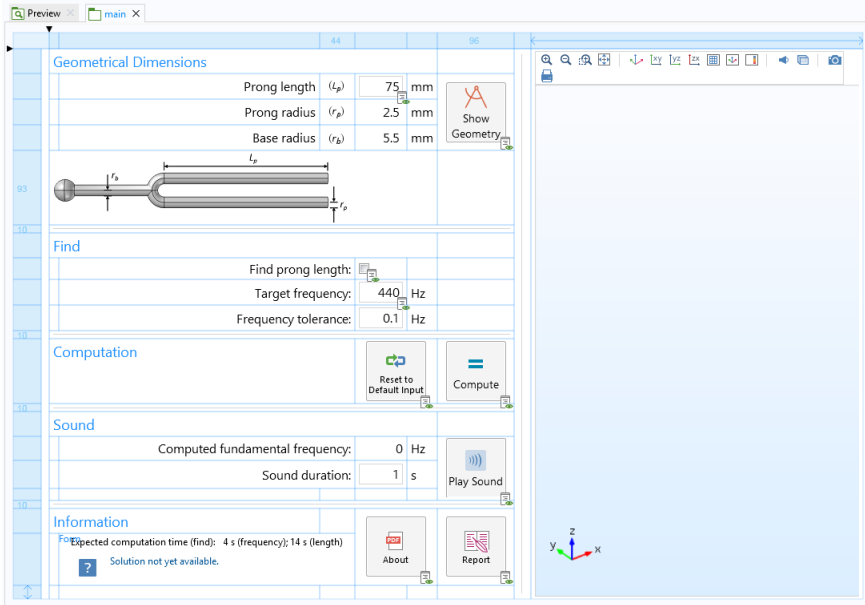


The buttons and menus in the ribbon **Grid** group give you easy access to commands for:

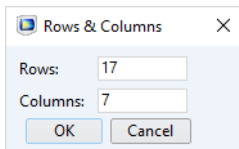
- Changing the row and column growth rules between **Fit**, **Grow**, and **Fixed**, which determine the layout when the user interface is resized (**Row Settings** and **Column Settings**).
- Inserting or removing rows and columns (**Insert** and **Remove**).
- Aligning form objects within grid cells (**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

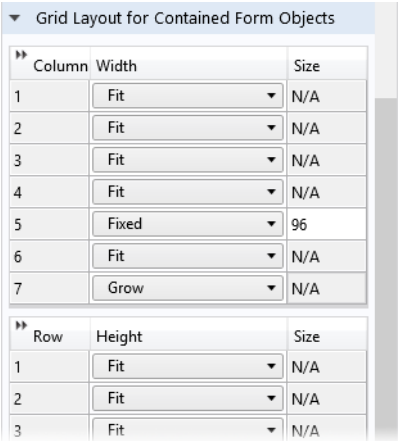
After switching to grid layout mode, the form window shows blue grid lines.



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



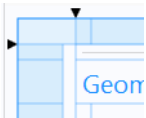
The section **Grid Layout for Contained Form Objects** in the **Settings** window shows column widths and row heights.



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
3	Fit	N/A

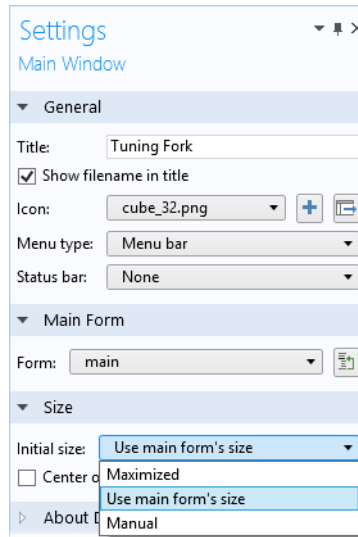
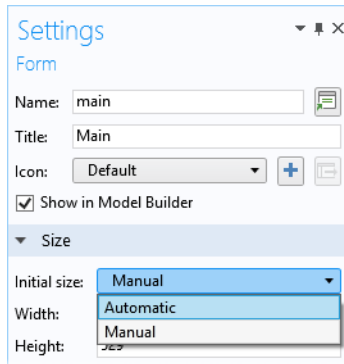
To interactively select a form, as displayed in the Form editor, click the top-left corner of the form.



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.

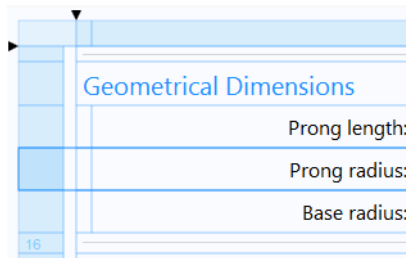
Note that if you switch from sketch to grid layout mode, all rows and columns will have the setting **Fit** and the handles for the frame will not be displayed. If any of the rows and columns have the **Height** or **Width** setting set to **Grow**, then the frame will display handles for resizing in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

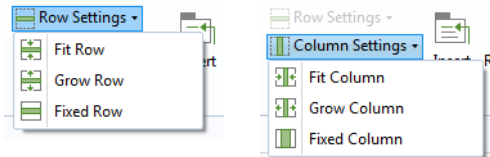


Similarly, to select a column, click the cell at the top. This cell cannot contain any form objects. The figure below shows the third column highlighted. In this case, the **Column Settings** menu is enabled in the ribbon tab.

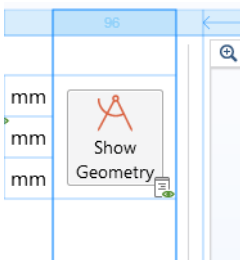
Geometrical Dimensions		
Prong length:	75	mm
Prong radius:	2.5	mm
Base radius:	5.5	mm

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.

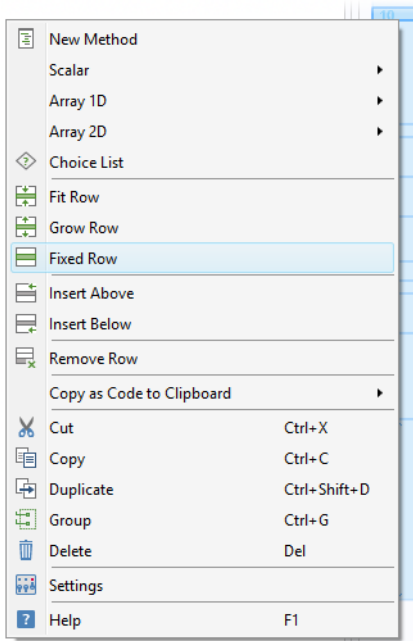


You can interactively change the row height and column width by dragging the grid lines.



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**.

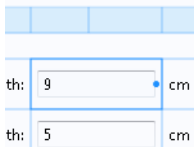
As an alternative to changing the **Row Settings** or **Column Settings** from the ribbon, you can right-click in a row or column and select from a menu.



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

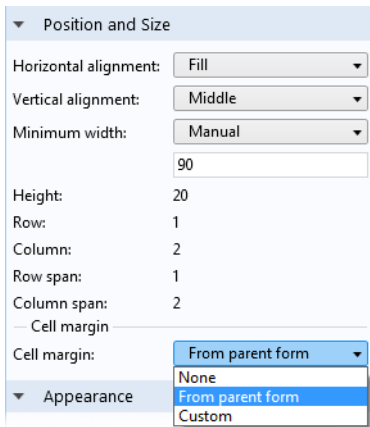
Cells

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 of 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 119
- **Custom**
 - Custom margins applied only to this form object

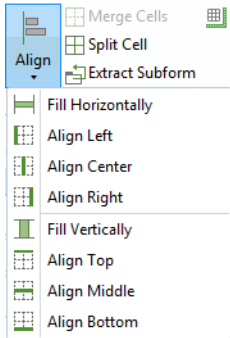
If the **Horizontal alignment** or **Vertical alignment** is set to **Fill** and the growth policy of the column or row allows the form object to be resized, then you can specify a minimum width or height, respectively. The minimum size can be set to **Manual** or **Automatic**. The **Manual** option 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 runtime to ensure that scroll bars are shown before the form object shrinks below its minimum size.

Depending on the type of form object contained in a cell, the **Width** and **Height** values can be set to **Automatic** or **Manual**, as described in “Position and Size” on page 107.

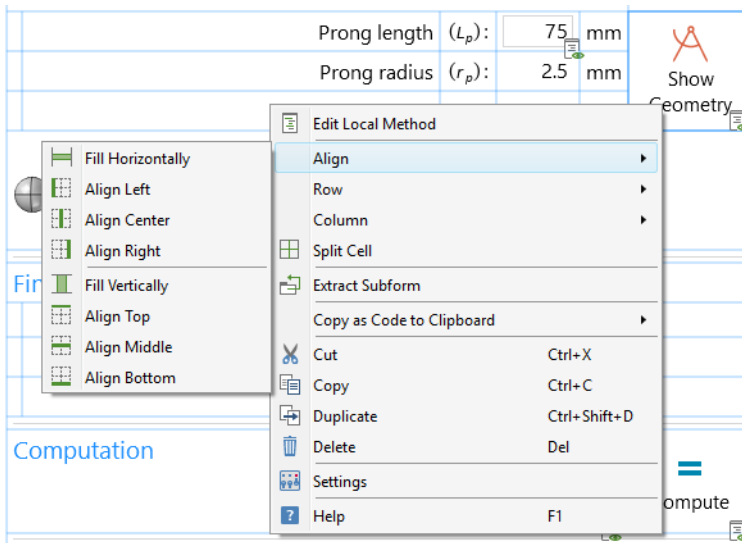
You can click and drag a rubber box to select multiple cells.

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

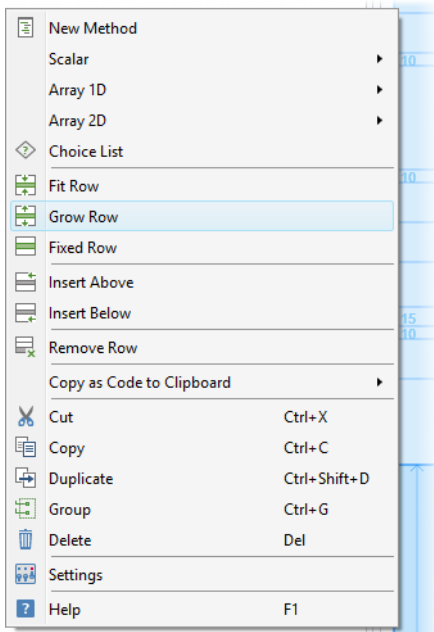
Thermal conductivity:	0.559	W/(m·K)
Heat of reaction:	-84666	J/mol
<input type="button" value="Compute"/>		

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

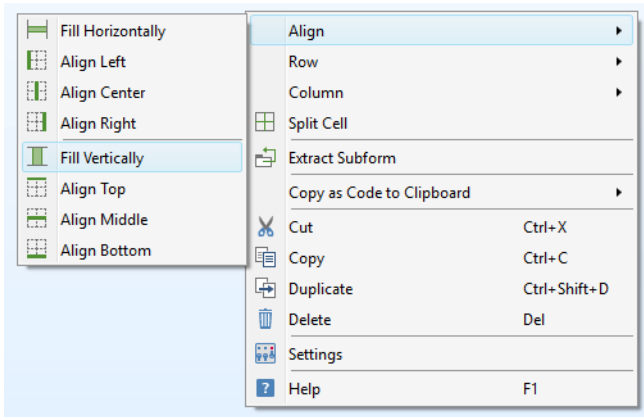
Automatic Resizing of Graphics Objects

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**.



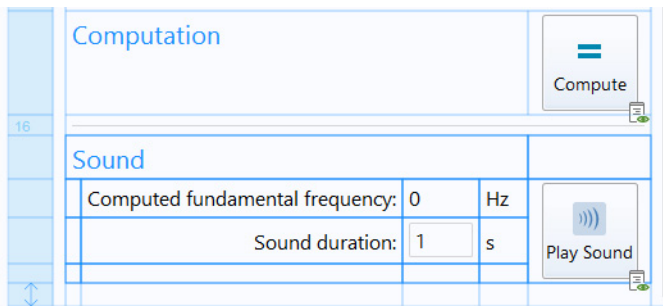
- Change the **Width** for any column covering the graphics object to **Grow**. To change this, right-click the uppermost row of the column you would like to access and select **Grow Column**.
- Select the graphics object and change both the **Horizontal alignment** and **Vertical alignment** to **Fill**.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



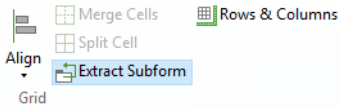
Following the steps above, you may find it easier to make graphics objects resizable by performing grid layout mode operations, such as adding empty rows and columns as well as merging cells. If you are already in grid layout mode, then a graphics object will default to **Fill** in both directions.

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.



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 form 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 a software interface with a worksheet and a settings window. The worksheet contains a form object with the following sections:

- Geometrical Dimensions:**

Prong length (l_p)	75 mm
Prong radius (r_p)	2.5 mm
Base radius (r_b)	5.5 mm
- Find:**

Find prong length:	
Target frequency:	440 Hz
Frequency tolerance:	0.1 Hz
- Computation:** Includes 'Reset to Default input' and 'Compute' buttons.
- Sound:**

Computed fundamental frequency:	0 Hz
Sound duration:	1 s

The settings window on the right shows the following configuration for the form:

- Name: subform2
- Form: form1
- Position and Size:
 - 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

The screenshot shows a worksheet with the extracted subform object. The subform contains the following sections:

- Sound:**

Computed fundamental frequency:	0	Hz
Sound duration:	1	s

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 257.

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.

Settings Form

Name: geometry_beam

Title: Beam dimensions

- Size
- Margins
- Dialog Settings
- Section Settings
- Grid Layout for Contained Form Objects

Column	Width	Size
1	Fixed	280
2	Fixed	100
3	Fixed	45
4	Grow	N/A

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

Inherit columns: None

Cell margins

Horizontal: 5

Vertical: 3

Settings Form

Name: geometry_rebars

Title: Reinforcement bars

- Size
- Margins
- Dialog Settings
- Section Settings
- Grid Layout for Contained Form Objects

Row	Height	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 113.

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.

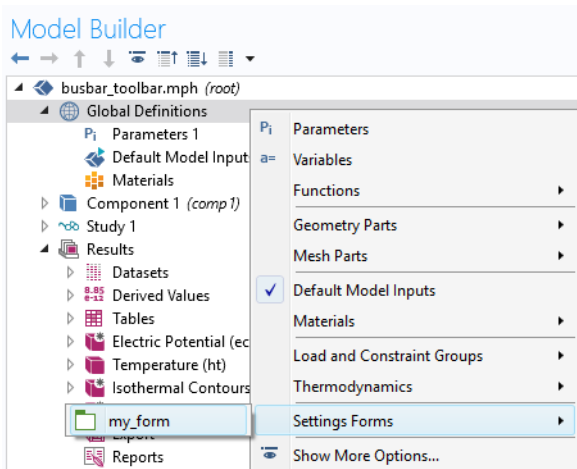
When you copy and paste forms and form objects between applications, the copied objects may contain references to other forms and form objects. Such references may or may not be meaningful in the application to which they are copied. For more information on the set of rules applied when pasting objects, see “Appendix B — Copying Between Applications” on page 298.

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.

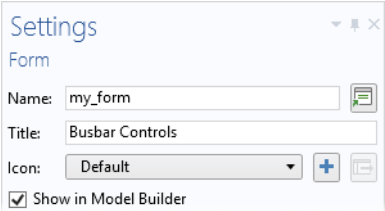
Using Forms in the Model Builder

Forms without graphics form objects can be used in the Model Builder. You can use this functionality to create customized **Settings** windows for, for example, common or repetitive tasks.

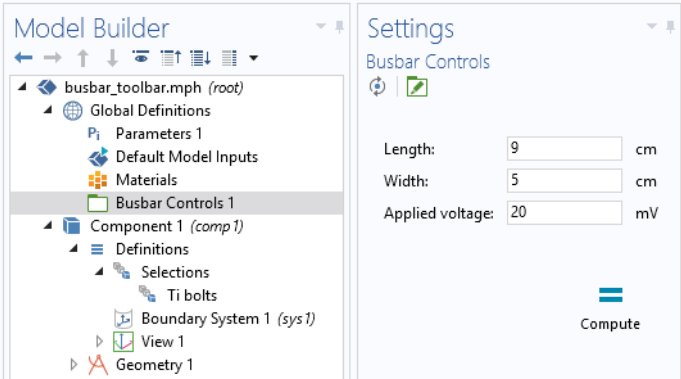
To use a form, right-click **Global Definitions** and select the form under **Settings Forms**.



You can control whether a form should be visible or not in the Model Builder as a **Settings Form** via the check box **Show in Model Builder**. This check box is available in the Application Builder in the **Settings** window of the corresponding form.

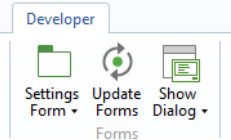


Once added to the model tree, the form is shown as a **Settings** window, shown in the figure below.



To show a **Settings Form** you can:

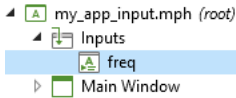
- Click the corresponding model tree node.
- Select it from the **Settings Form** menu button in the Developer tab in the ribbon.
- Show it as a dialog box by selecting it from the **Show Dialog** menu button in the **Developer** tab in the ribbon.



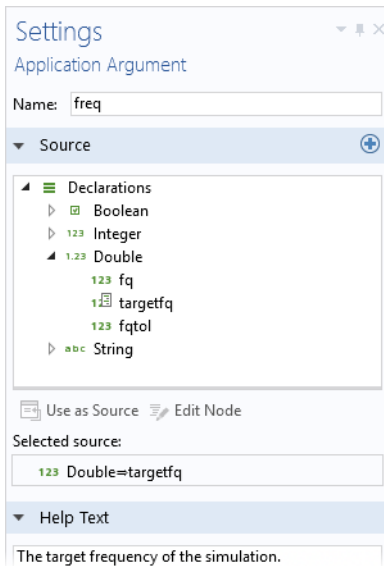
For reusing a **Settings Form** between sessions, you can create an **Add-in**. For more information, see “Creating Add-Ins” on page 205.

Inputs

When starting an application from the operating system command line, you can provide input arguments. In the application tree, you specify such input arguments under the **Inputs** node.

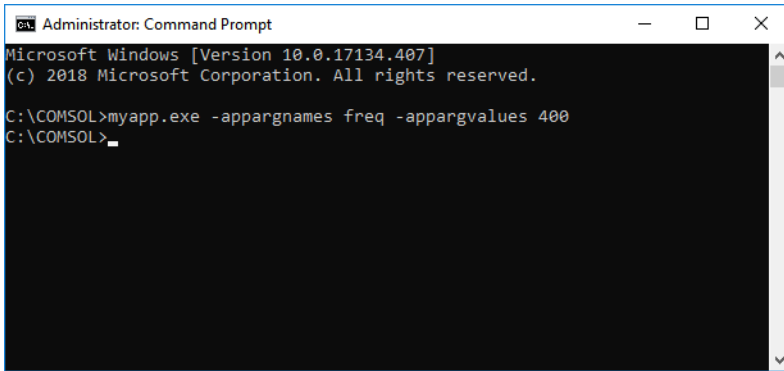


Command-line arguments are automatically written to the declarations you define as **Selected source**, in the corresponding **Settings** window for an **Application Argument**. They can be used, for example, to provide input data or configuration settings.



Command-line arguments can be used when starting applications with COMSOL Multiphysics, COMSOL Server, as well as when starting applications that have been compiled with COMSOL Compiler. In the example below, for a compiled

application in Windows®, an input argument `freq` is given that takes a (double) value 400.



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.17134.407]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\COMSOL>myapp.exe -appargnames freq -appargvalues 400
C:\COMSOL>
```

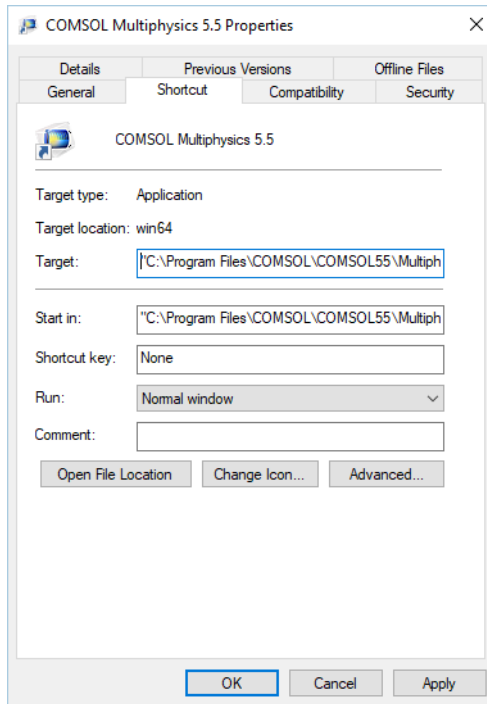
For COMSOL Multiphysics, the corresponding command would be

```
comsol.exe -run myapp.mph -appargnames freq -appargvalues 400
```

When running this command, you need to be positioned in the COMSOL Multiphysics installation directory where the executable `comsol.exe` is located, for example

```
C:\Program Files\COMSOL\COMSOL55\Multiphysics\bin\win64
```

Alternatively, you can copy and paste the **COMSOL Multiphysics 5.5 Windows®** Desktop shortcut icon (in order to keep the original shortcut), right-click the icon, and select **Properties**; as shown in the figure below.



You can, for example, modify the **Target** text field to be:

```
"C:\Program Files\COMSOL\COMSOL55\Multiphysics\bin\win64\comsol.exe" -run myapp.mph -appargnames freq -appargvalues 400
```

To provide input arguments with special characters, you need to use single quotes. The following example of a compiled application shows how to provide a file path, such as for a configuration file, as an input argument:

```
myapp.exe -appargnames configfile -appargvalues 'C:\\COMSOL\\my_conf.dat'
```

For COMSOL Server, you can provide the arguments directly in the address field of your browser (URL); for example:

```
http://<host:port>/app/myapp_mph?appargnames=freq&appargvalues=400
```

You can also use a file declaration as an input argument. This is useful, for example, when you want to let users supply input files. For example:

```
comsol.exe -run file_arguments.mph -appargnames interpfile -appargvalues 'C:\data\functions\simpleinterp.txt'
```

This example uses an application argument `interpfile`, which is linked to a file declaration to read the interpolation file `simpleinterp.txt` when launching the application. This file is then used in an interpolation function in the application's embedded model.

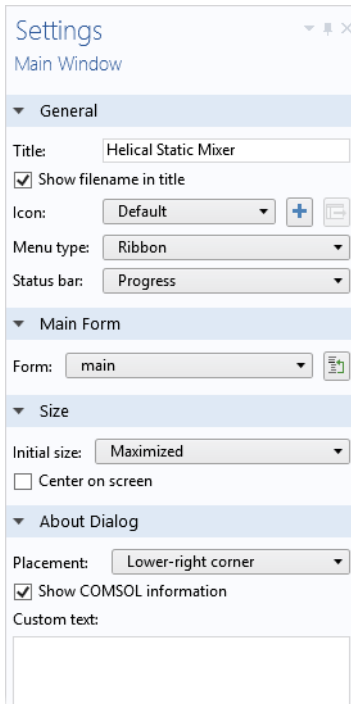
The Main Window

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

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

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



The screenshot shows the 'Settings' window for the 'Main Window'. The 'General' section is expanded and contains the following settings:

- Title:** Helical Static Mixer
- Show filename in title**
- Icon:** Default (with a '+' button and a file icon button)
- Menu type:** Ribbon
- Status bar:** Progress

The 'Main Form' section is also expanded and contains:

- Form:** main (with a file icon button)

The 'Size' section contains:

- Initial size:** Maximized
- Center on screen**

The 'About Dialog' section contains:

- Placement:** Lower-right corner
- Show COMSOL information**
- Custom text:** (empty text area)

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

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.

SIZE

In the **Size** section, 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 49. 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 that there is enough room for the window title, ribbon, and menu bar.

In addition there is a check box **Center on screen** that is applicable to any **Size** setting that does not correspond to a maximized window.

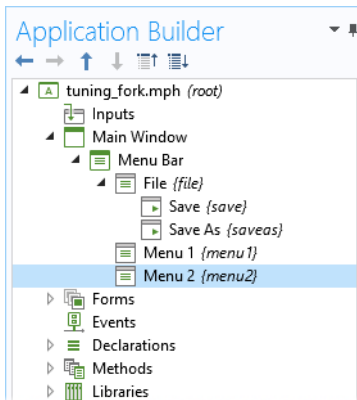
For more information on the option **Use main form's size**, see “The Form Settings Window and the Grid” on page 109.

ABOUT DIALOG

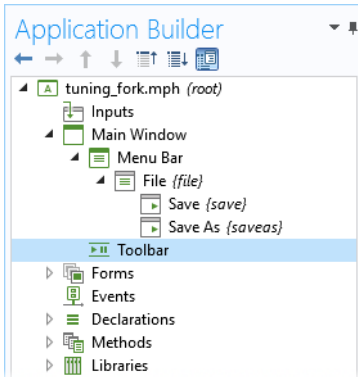
The **About Dialog** section contains settings for customizing parts of the **About This Application** dialog box, which contains legal information. The **Placement** option lets you choose between **Automatic**, **File menu**, **Ribbon**, or **Lower-right corner**. The **Lower-right corner** option will place a hyperlink to the **About This Application** dialog box in the lower-right corner of the application user interface. If selected, the check box **Show COMSOL information** will display COMSOL software version and product information. Any text entered in the **Custom text** field will be displayed above the legal text in the dialog box. In the **Custom text** field, words containing `http` or `www` will be interpreted as hyperlinks, if possible. For example, `http://www.comsol.com` or `www.comsol.com` will be replaced with a hyperlink.

Menu Bar and Toolbar

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

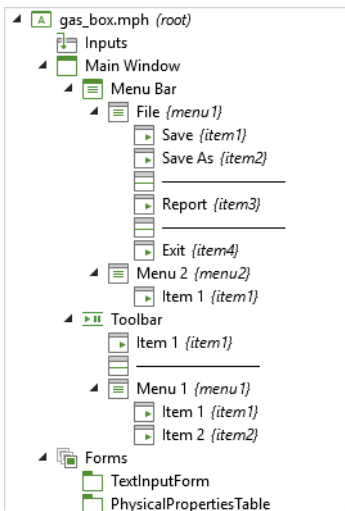


For the **Menu Bar** option, you can also add a **Toolbar**. The **Toolbar** node and the **Menu** nodes have the same type of child nodes.

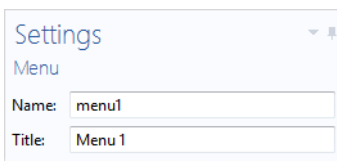


MENU, ITEM, AND SEPARATOR

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



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

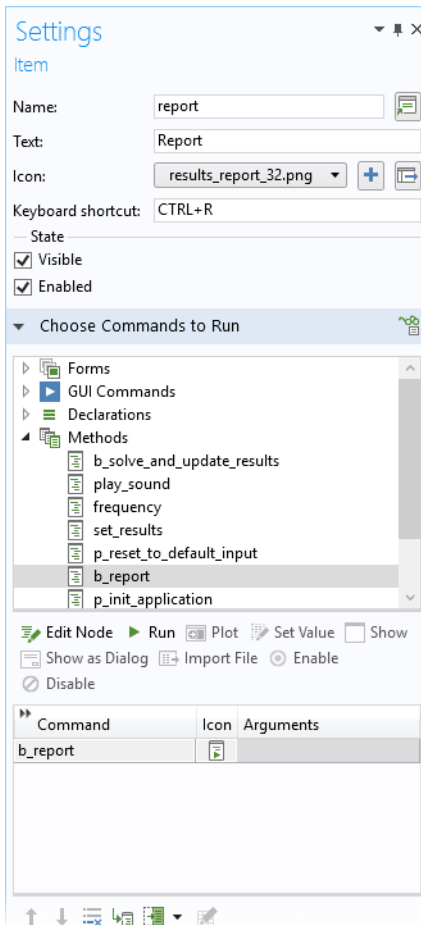


A **Menu** node can have child **Menu** nodes that represent submenus.

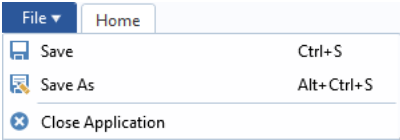
A **Separator** displays a horizontal line between groups of menus and items, and has no settings.

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 59. In a similar way, the **Settings** window for a **Toggle Item** node is similar to that of a toggle button.

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

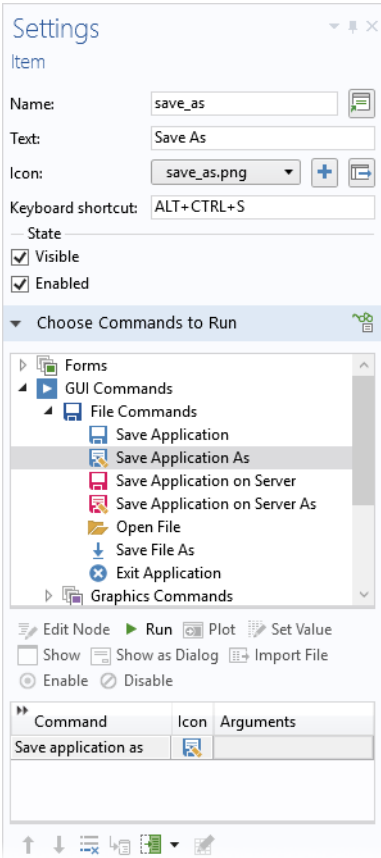


The figure below shows an example of an application with a **File** menu.



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 324.

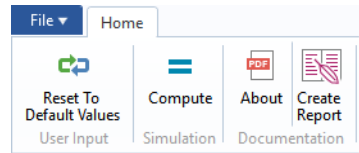
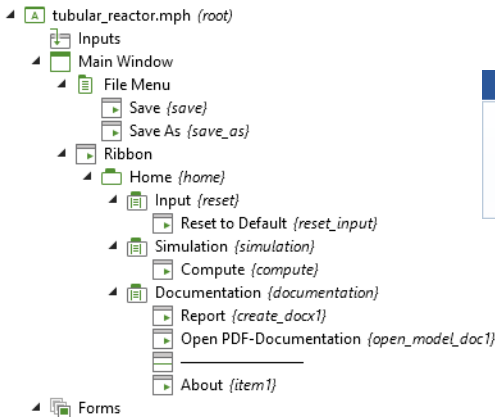
Ribbon

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**, **Toggle Item**, **Menu**, or **Separator**.

Item and **Menu** provide the same functionality as described previously for the **Menu Bar** 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

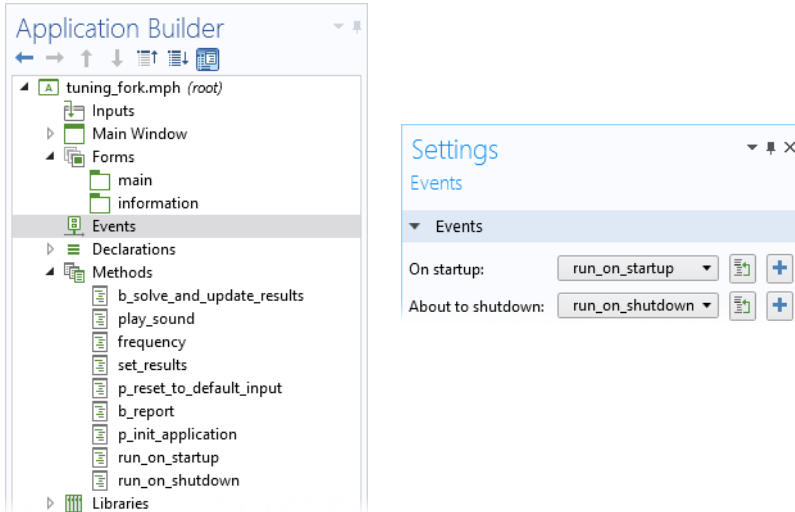
An event is any activity (for example, clicking a button, typing a keyboard shortcut, loading a form, or changing the value of a variable) that signals a need for the application to carry out one or more actions. Each action can be a sequence of commands of the type described earlier, or may also include the execution of methods. The methods themselves may be local methods associated with particular forms or 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 form methods are listed under the nodes of the respective form. 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 of 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 (**About to 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()`

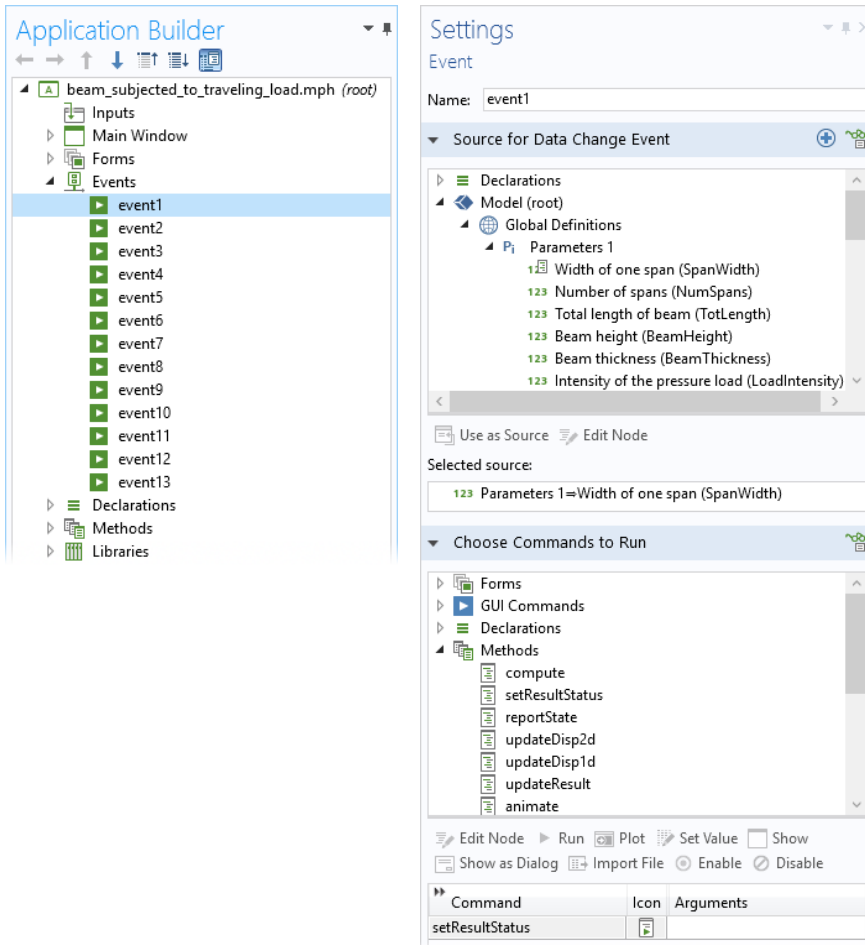
A method run at a shutdown event can, for example, automatically save critical data or prompt the user to save data. In addition, a method run at a shutdown event may cancel the shutdown by returning a Boolean `true` value.

LIMITATIONS WITH ON STARTUP EVENTS

Methods used for an **On startup** event cannot utilize Application Builder functionality related to graphics or user interfaces. This is due to the fact that an **On startup** event is run before the full application user interface is loaded. For example, a method that is used for initializing graphics, such as **Zoom Extents**, needs to be run as an **On load** event for a form and not as a global **On startup** event. Another example is showing a dialog box using a built-in method such as `confirm`. In this case, no dialog box will be shown and the operation will simply be ignored.

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.



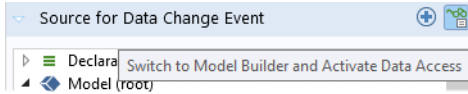
Note that since this type of event has global scope and is not associated with a particular form, the full path: `/form1/graphics1` needs to be used when referencing graphics objects.

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.

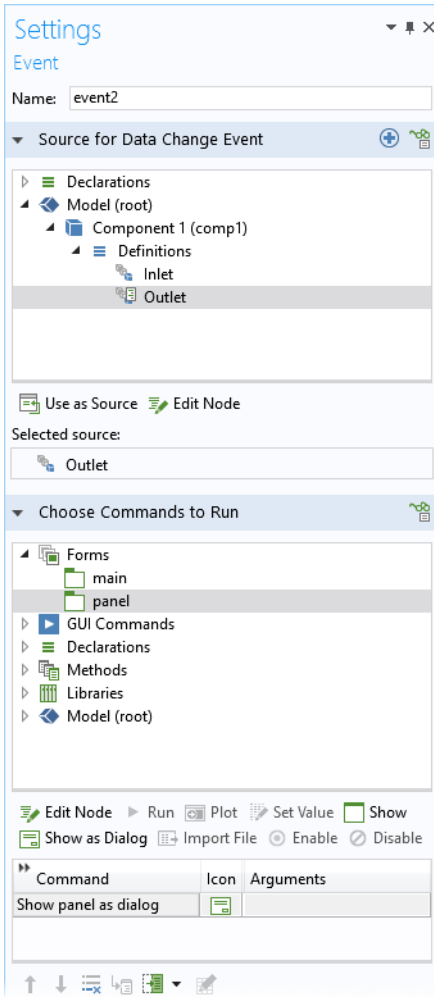
You can extend the list of available data nodes by clicking on the **Switch to Model Builder and Activate Data Access** button in the header of the section **Source For Data Change Event**.



For more information, see “Data Access in the Method Editor” on page 173.

Note that **Explicit** selections are also allowed as **Source for Data Change Event**. This allows a command sequence or a method to be run when the user clicks a geometry object, domain, face, edge, or point. The figure below shows a dialog

box for a global event that opens a form panel as a dialog box when the user changes the contents of the **Explicit** selection named **Outlet**.



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 59.

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, form, 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 **Settings** window of a 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 or under the **Methods** node of the corresponding form, and a local method (optional).

The form objects supporting this type of event are:

- **Input Field**
- **Check Box**
- **Combo Box**
- **Graphics**
- **File Import**
- **Array Input**
- **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 them. 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 59.

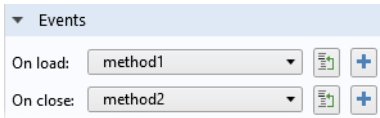
Selecting Multiple Form Objects

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. This functionality is not available for all combinations of form objects.

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. Note that a method that is used for initializing graphics, such as **Zoom Extents**, needs to be run as an **On load** event for a form and not as a global **On startup** event.

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 188.

Declarations

The **Declarations** node in the application tree is used to declare global variables and objects, 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 197.

You can create a **Declarations** node that is local to a form. Such **Declarations** for a form can only be used in that particular form, including form objects and methods that are local to the form.

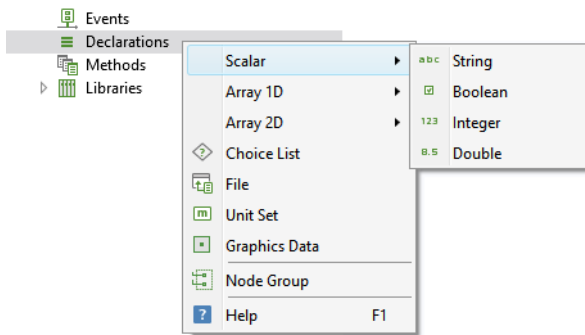
These are the different types of global **Declarations**:

- **Scalar**
- **Array 1D**
- **Array 2D**
- **Choice List**
- **File**
- **Unit Set**
- **Shortcuts**
- **Graphics Data**

Form **Declarations** can only be of the types:

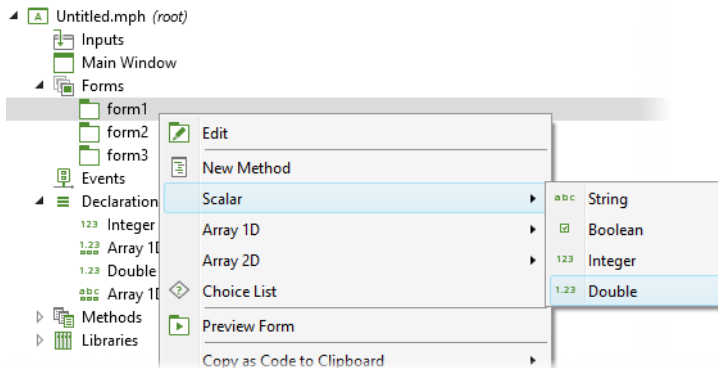
- **Scalar**
- **Array 1D**
- **Array 2D**
- **Choice List**

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

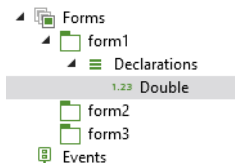


Note that **Shortcuts** are not created from this menu but by clicking the **Create Shortcut** button next to the **Name** in the **Settings** window of a form object or by using Ctrl+K for a selected form object.

To create **Declarations** that are local to a form, right-click the corresponding form and select the variable type, as shown below.



Variables that are local to a form are organized under a **Declarations** node that is a child node to the form, as shown below.

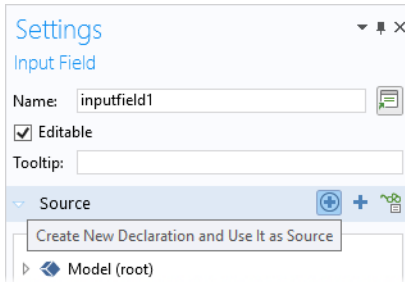


The first three types of declarations, **Scalar**, **Array 1D**, and **Array 2D**, can be of the following data types:

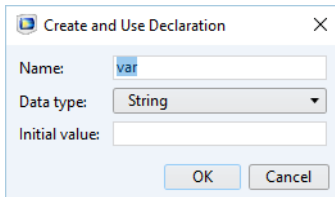
- **String**

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

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



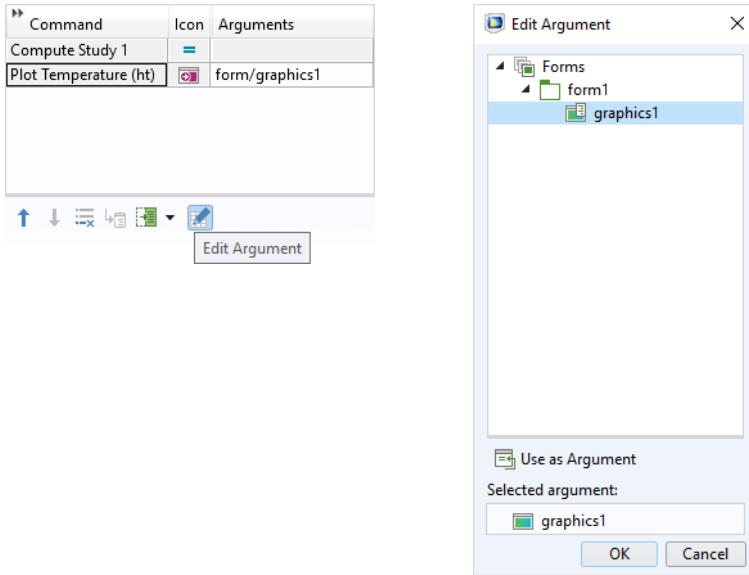
This will open a dialog box that lets you quickly declare scalar variables.



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 59.

The figure below shows a command sequence that includes a **Plot Temperature** command 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, use indexes. 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 indexes, for example, `my_matrix(2,3)`. The indexes can themselves be other declared variables, for example, `my_variable(n)`.

For commands requiring a graphics object as an input argument, only string type declarations are allowed with appropriate indexes, 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 underscores. The reserved names `root` and `parent` are not allowed and Java[®] programming language keywords cannot be used.

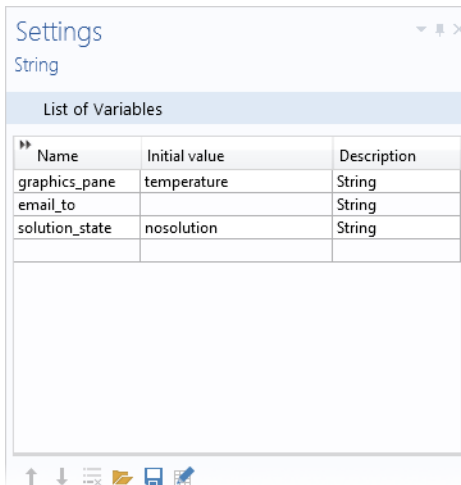
Scalar

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 337. 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`.



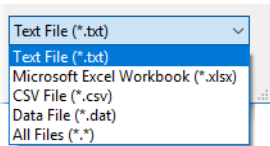
String declarations, as well as other declarations, can be loaded and saved from or to a file by using the **Load from File** and **Save to File** buttons below the **List of Variables** table.

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

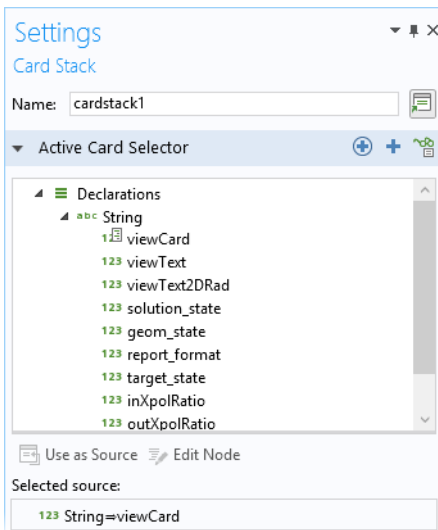
- Text File (.txt)

- 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 is shown in the figure below.



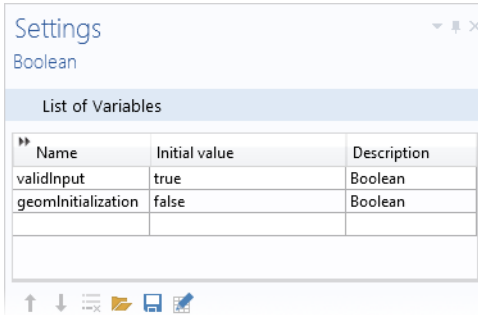
To illustrate the use of declared strings, the figure below shows the **Settings** window of a card stack object where the string variable `viewCard` is used as the source (**Active Card Selector**).



For more information on using card stacks, see “Card Stack” on page 261.

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 'Settings Boolean' dialog box with a 'List of Variables' table. The table has three columns: Name, Initial value, and Description. Two variables are listed: 'validInput' with an initial value of 'true' and 'geomInitialization' with an initial value of 'false'. Both are described as 'Boolean'.

Name	Initial value	Description
validInput	true	Boolean
geomInitialization	false	Boolean

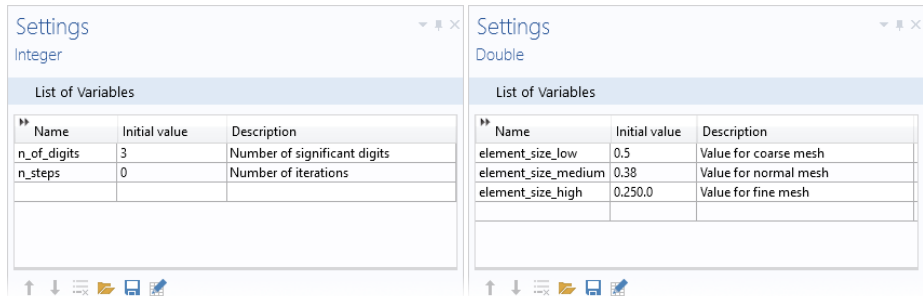
Example Code

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.



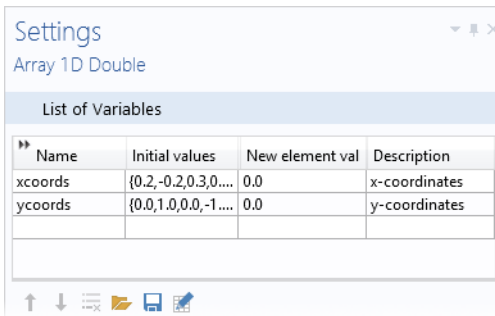
The image shows two side-by-side screenshots of settings dialog boxes. The left one is 'Settings Integer' and the right one is 'Settings Double'. Both show a 'List of Variables' table.

Name	Initial value	Description
n_of_digits	3	Number of significant digits
n_steps	0	Number of iterations

Name	Initial value	Description
element_size_low	0.5	Value for coarse mesh
element_size_medium	0.38	Value for normal mesh
element_size_high	0.250.0	Value for fine mesh

Array ID

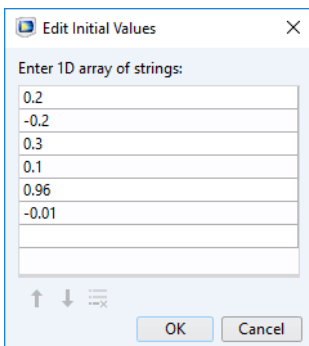
The **Array ID** node declares one or more named arrays of strings, Booleans, integers, or doubles that you can access from form objects and methods. The number of elements in a 1D array is not restricted in any way, and you can, for example, use a 1D array to store a column in a table with a variable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



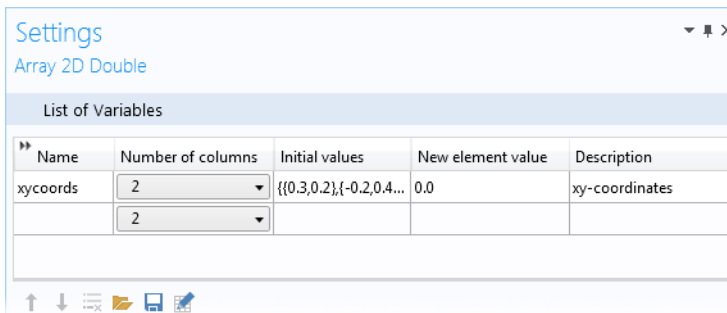
ARRAY SYNTAX

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 a 3-element array and a 2-element array

Array 2D

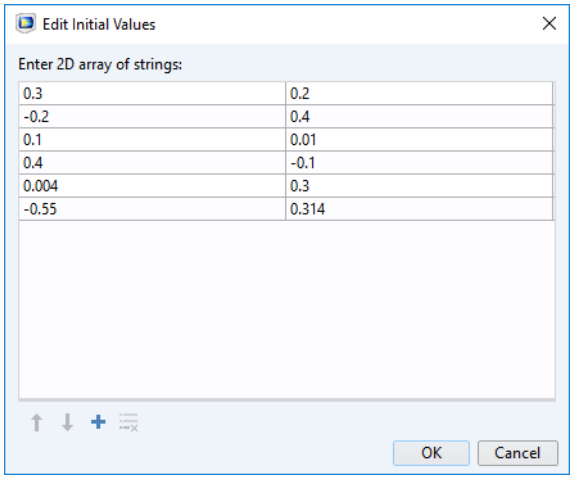
The **Array 2D** node declares one or more 2D arrays that you can access using form objects and methods. In the figure below, the 2D double array `xycoords` is declared.



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:

```

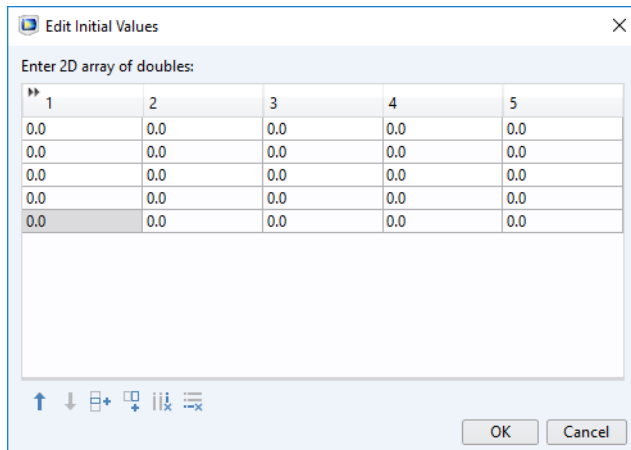
1 2 3
4 5 6

```

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

To interactively define the **Initial values** of a 2D array, select the **Undefined** option for the **Number of columns**. The **Edit Initial Values** button opens a dialog box where

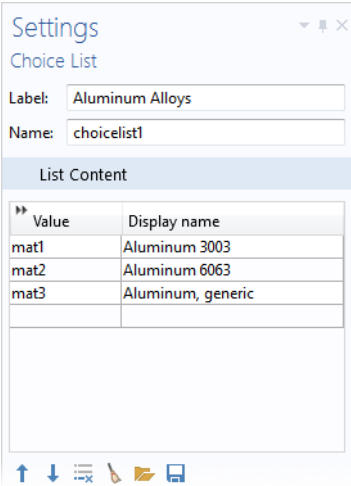
the number of rows and columns can be interactively changed, as shown in the figure below.



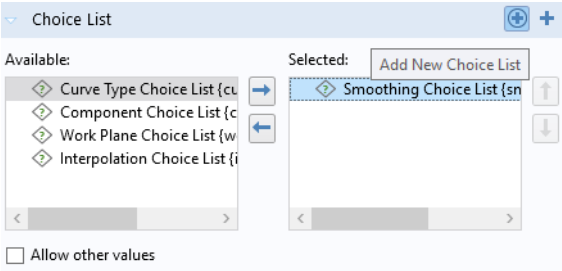
Choice List

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 (**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 (**Display name**).

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.



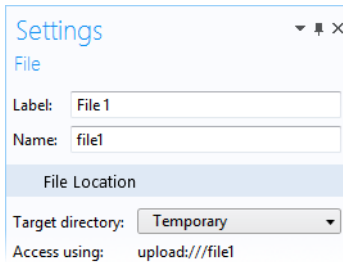
In addition you can click the adjacent **Add New Form Choice List** to create a choice list local to the form.

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 233.

File

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 325. 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 300.

Unit Set

The **Unit Set** node contains lists that can be used by combo boxes, radio buttons, or list boxes for the purpose of changing units. The **Settings** window for a unit set contains two sections: **Unit Groups** and **Unit Lists**.

Settings
Unit Set

Label: Unit System
Name: unitset1

Unit Groups

Value	Display name
SI	SI
imperial	Imperial

Initial value: SI

Unit Lists

Name	SI	Imperial
length	cm	in
potential	mV	mV

Each row in the **Unit Groups** table is a unit group that represents a collection of units with a particular meaning in the context of the application user interface. Each column represents a group of units labeled by a **Value** and a **Display name**.

Each row in the **Unit Lists** table is a unit list with columns containing units with the same dimension; for example, mm, cm, dm, m, km. The headings of the **Unit Lists** table are **Name** and the **Display names** are defined in the **Unit Groups** section. A unit list specifies the possible units that a form object that references the **Unit Set** can switch between when running the application.

The figure above demonstrates the use of a **Unit Set** for an application that allows for switching between metric and imperial units. In this example, two unit groups are defined: **SI** and **Imperial**. The **Label** of the **Unit Set** has been changed to **Unit System**.

The **Value** column contains string values that represent the current choice of unit group. These string values can be manipulated from methods. The **Display name**

column is the string displayed in the user interface. The **Initial value** list contains the default unit group (SI in the example above).

In the example above, the **Unit Lists** table has three columns: **Name**, **SI**, and **Imperial**. The **SI** and **Imperial** columns are created dynamically based on the groups in the **Unit Groups** section. Each row in the table corresponds to a physical quantity such as, in this example, length and potential. Each column in the table corresponds to the allowed units of length and units of potential, respectively. The figure below shows an example application where a combo box form object is used to choose between the **SI** and **Imperial** unit groups.

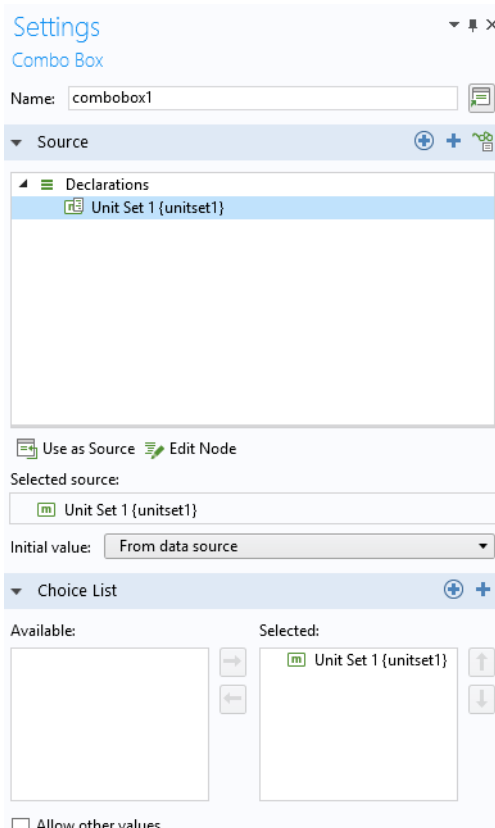
Length: cm
Width: cm
Applied voltage: mV

Unit system:
SI
Imperial

Length: in
Width: in
Applied voltage: mV

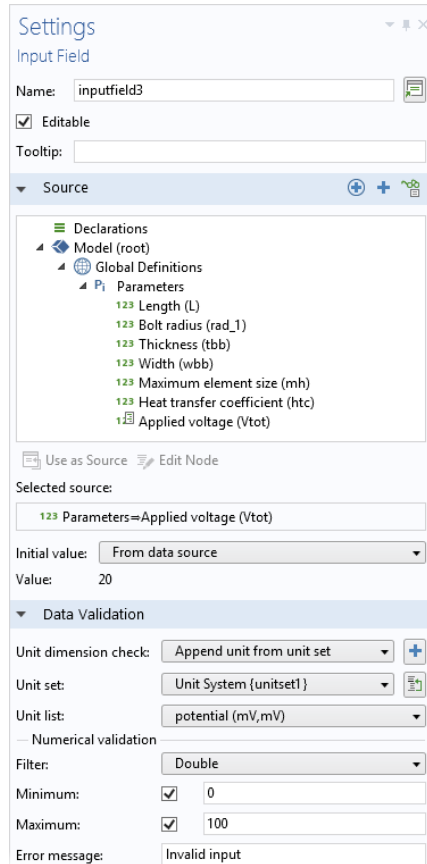
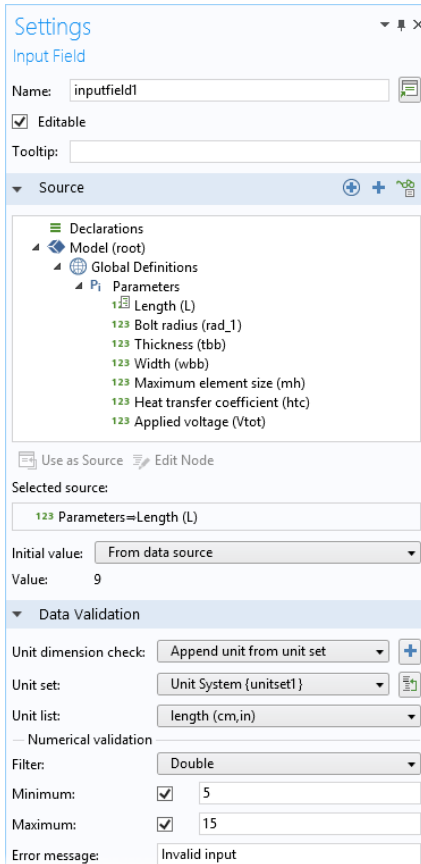
Unit system:

The figure below shows the **Settings** window of a combo box using the **Unit Set** of the above example as the **Source**.



In this way, a **Unit Set** can be used instead of a **Choice List** to create a combo box for unit selection. Instead of a combo box, you can use a list box or a radio button object in a similar way.

The two figures below show the corresponding **Settings** windows for the two input fields for **Length** and **Applied voltage**.



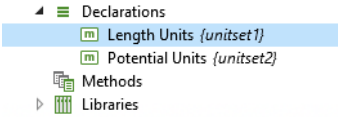
The **Unit dimension check** is set to **Append unit from unit set**. The **Unit set** is set to **Unit System {unitset1}** (the user-defined label for the **Unit Set** declaration used in this example). The **Unit list** is set to **length** and **potential**, respectively. When using **Append unit from unit set**, the **Numerical validation** section (under **Data Validation**) refers to the **Initial value** of a **Unit Set**; in this case, **cm** and **mV**, respectively. The **Minimum** and **Maximum** values are scaled automatically when the application is run and the unit is changed by the user of the application. For more information on the settings for an input field object, see “Input Field” on page 88.

The figures below illustrate the use of two **Unit Set** declarations for separately setting the unit for length and potential, respectively.

Length: cm
 Width: cm
 Applied voltage: mV

=
Compute

Length unit:
 Potential unit:



The figures below show the corresponding **Settings** window for the **Unit Set** declarations.

Settings
Unit Set

Label:
 Name:

Unit Groups

Value	Display name
cm	cm
m	m
inch	inch

Initial value:

Unit Lists

Name	cm	m	inch
length	cm	m	in

Settings
Unit Set

Label:
 Name:

Unit Groups

Value	Display name
V	V
mV	mV

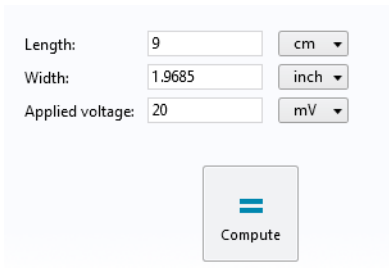
Initial value:

Unit Lists

Name	V	mV
potential	V	mV

Note that, in this example, by using three **Unit Set** declarations, you can have individual length unit settings for the **Length** and **Width** input fields. The figure

below shows such an example, where three combo boxes have been used to replace the unit labels and each combo box uses a separate **Unit Set** declaration as its source.



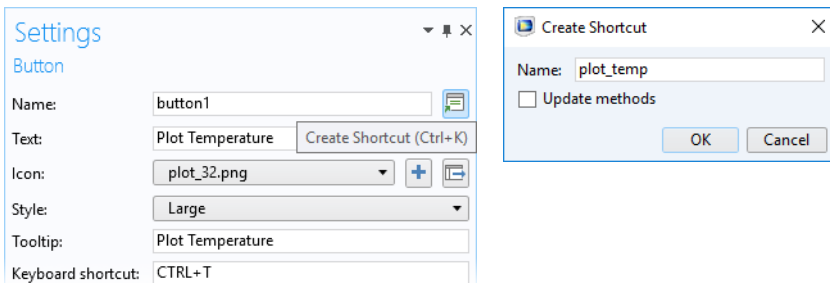
The screenshot shows a form with three rows of input fields and dropdown menus. The first row is labeled 'Length:' with a text input containing '9' and a dropdown menu set to 'cm'. The second row is labeled 'Width:' with a text input containing '1.9685' and a dropdown menu set to 'inch'. The third row is labeled 'Applied voltage:' with a text input containing '20' and a dropdown menu set to 'mV'. Below these fields is a button with a blue equals sign icon and the text 'Compute'.

When more flexibility is required, you can combine the use of a **Choice List** and a **Unit Set**. For example, for a combo box, you can use the **Unit Set** as the **Selected source** (string) and select a **Choice List** that is not a **Unit Set**.

Shortcuts

Form objects and other user interface components are referenced in methods by using a certain syntax. For example, using the default naming scheme `form3/button5` refers to a button with the name `button5` in `form3` and `form2/graphics3` refers to a graphics object with the name `graphics3` in `form2`. You can also change the default names of forms and form objects. For example, if `form1` is your main form, then you can change its name to `main`.

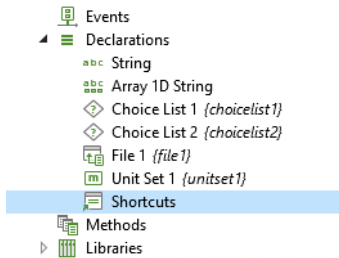
To simplify referencing form objects as well as menu, ribbon, and toolbar items by name, you can create shortcuts with a custom name. In the **Settings** window of an object or item, click the button to the right of the **Name** field and type a name of your choice.



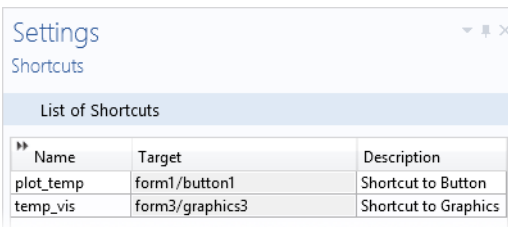
The screenshot shows two windows. On the left is the 'Settings' window for a 'Button' object. It has fields for Name (button1), Text (Plot Temperature), Icon (plot_32.png), Style (Large), Tooltip (Plot Temperature), and Keyboard shortcut (CTRL+T). A 'Create Shortcut (Ctrl+K)' button is visible next to the Text field. On the right is the 'Create Shortcut' dialog box, which has a 'Name' field containing 'plot_temp', an unchecked 'Update methods' checkbox, and 'OK' and 'Cancel' buttons.

To create or edit a shortcut, you can also use the keyboard shortcut `Ctrl+K`.

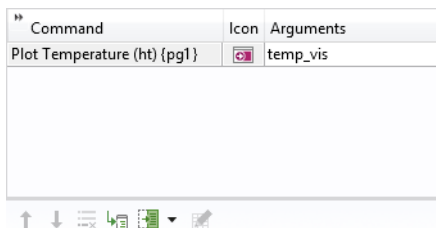
All shortcuts that you create are made available in a **Shortcuts** node under **Declarations** in the application tree.



In the **Settings** window for **Shortcuts** below, two shortcuts, `plot_temp` and `temp_vis`, have been created for a button and a graphics object, respectively.



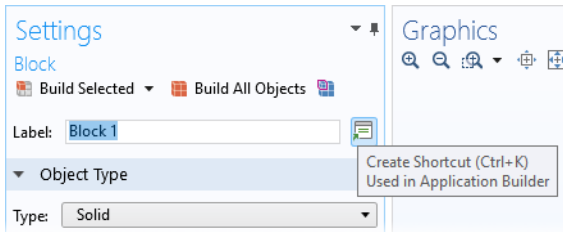
The shortcuts can be referenced in other form objects or in code in the Method editor. The example below shows a shortcut, `temp_vis`, used as an input argument to a temperature plot.



Shortcuts are automatically updated when objects are renamed, moved, copied, and duplicated. They are available in methods as read-only Java[®] variables, just like `string`, `int`, `double`, and `Boolean` declarations.

Using shortcuts is recommended because it avoids the need to adjust method editor code when the structure of the application user interface changes.

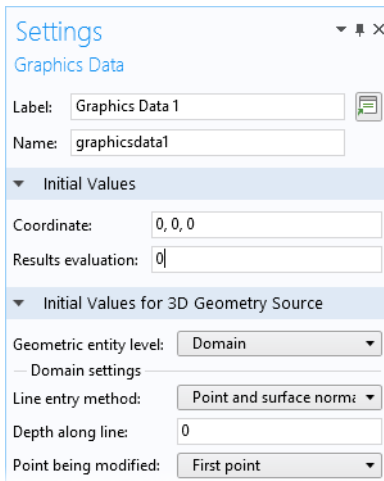
Shortcuts are also available in the Model Builder, for use with the Application Builder. In the **Settings** window of a model tree node, click the button to the right of the **Label** field and type a name of your choice.



The custom name of a shortcut becomes available as a global variable in methods and will be used, for example, when recording code or new methods.

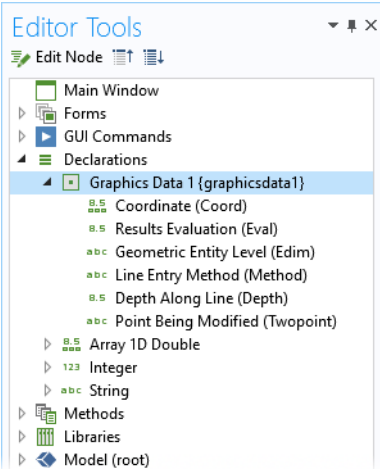
Graphics Data

A **Graphics Data** declaration node is used to pick data at a specific coordinate from a graphics object based on mouse clicks by the user. The figure below shows the corresponding **Settings** window.

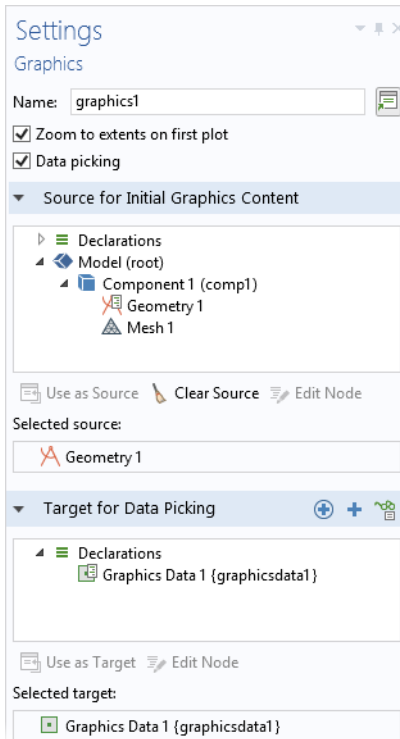


The **Initial Values** section contains default values for the extracted data properties **Coordinate** and **Results evaluation**. The section **Initial Values for 3D Geometry Source** contains settings for the selection methods available when the **Source for Initial Graphics Content** of a graphics object is set to a geometry node.

The different properties of a graphics data declaration are available from the Editor Tools window as shown in the figure below.



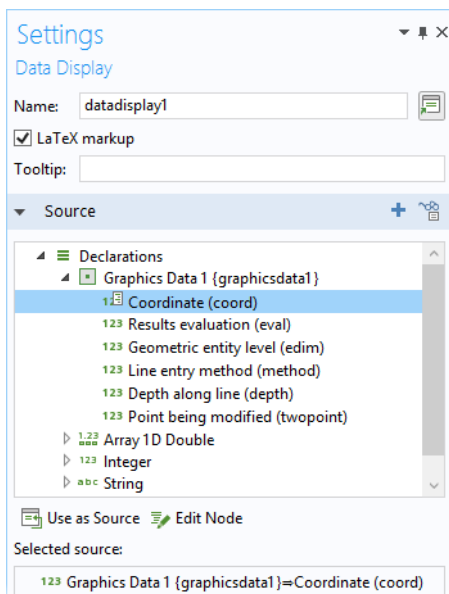
To use a **Graphics Data** declaration node for data picking, select the **Data picking** check box in the **Settings** window of a graphics object and select the **Graphics Data** node as the **Target for Data Picking**, as shown in the figure below.



GRAPHICS DATA FROM RESULTS

When the **Source for Initial Graphics Content** of a graphics object is set to a plot group node, then the **Results Evaluation** value corresponds to the field value at the position determined by the mouse pointer. The **Coordinate** value corresponds to the coordinate at that position. Note that in the Model Builder, this corresponds to the data displayed in the **Evaluation 2D** or **Evaluation 3D** tables.

The figure below shows a data display object where the **Coordinate** property is used as **Source**.



You can also use the **Coordinate** property as the **Source** for an array input object. The **Results Evaluation** property can be used as the **Source** for several form objects including data display and input field objects.

GRAPHICS DATA FROM GEOMETRY

The settings **Geometry Entity Level**, **Line Entry Method**, **Depth Along Line**, and **Point Being Modified** only apply when the **Source for Initial Graphics Content** of a graphics object is set to a 3D geometry node. These settings provide the same point selection methods as a **Domain Point Probe**, when **Geometry Entity Level** is set to **Domain**; and **Boundary Point Probe**, when **Geometry Entity Level** is set to **Boundary**. The settings **Line Entry Method**, **Depth Along Line**, and **Point Being Modified** are only applicable when **Geometry Entity Level** is set to **Domain**.

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 built-in method can be found in “Appendix E — Built-In Method Library” on page 324 and in the *Application Programming Guide*.

The contents of the application tree in the Application Builder are accessed through the application object, which is an important part of the model object. You can record and write code using the Method editor that directly accesses and changes user interface aspects of the running application, such as button texts, icons, colors, and fonts.

There are global methods, form methods, and local methods. Global methods are displayed in the application tree and are accessible from all methods and form objects. Form methods are displayed in the application tree as child nodes to the form it belongs to. 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 188.



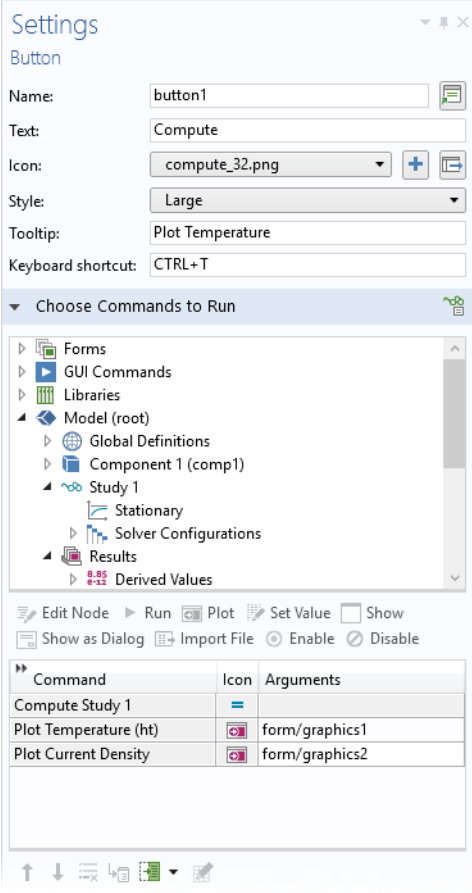
A number of tools and resources are available to help you create code for methods. These are covered in the following sections and will make you more productive by allowing you to copy-paste or autogenerate blocks of code, for example.

Converting a Command Sequence to a Method

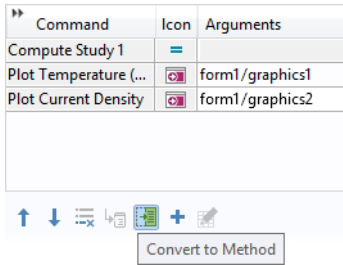
In the Form editor, select **Convert to Method** from the menu button displayed in the **Settings** window below an existing command sequence. The command sequence is automatically replaced by an equivalent method. In the same way you can select **Convert to Form Method** and **Convert to Local Method**.

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

You will 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.

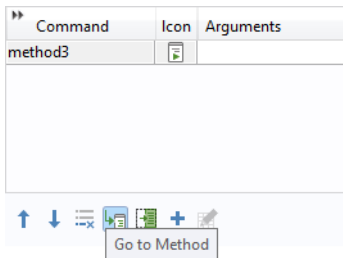


Click the **Convert to Method** button below the command sequence.



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.

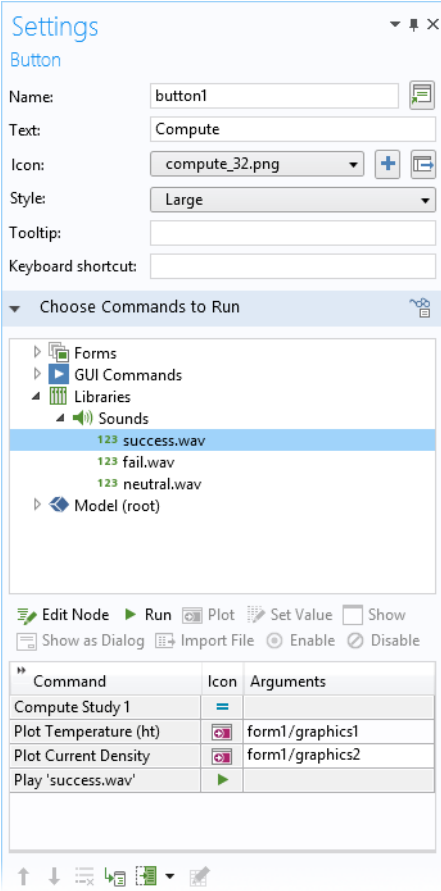


In the Method editor, add a call to the built-in method `playSound` to play the sound file `success.wav`, available in the COMSOL sound library, by using the syntax shown in the figure below.

```
Preview × method3 ×  
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 to the left.

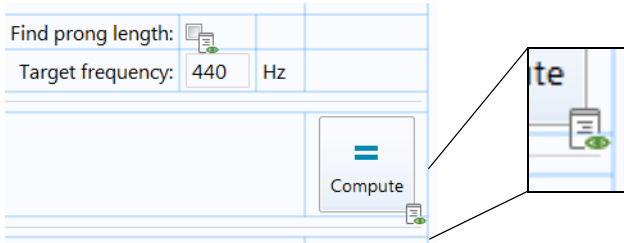
Note that in the example above, you do not have to use the Method editor. In the command sequence, select the file `success.wav` under **Libraries > Sounds** and click the **Run** command button under the tree, as shown in the figure below.



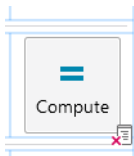
However, there are many built-in methods that do not have corresponding command sequence nodes. For more information, see “Appendix E — Built-In Method Library” on page 324.

FORM OBJECT WITH ASSOCIATED METHOD

A form object that has an associated method is indicated with a special icon in the Form editor, as shown in the figure below. In this example, both the check box called **Find prong length** and the **Compute** button have associated methods.

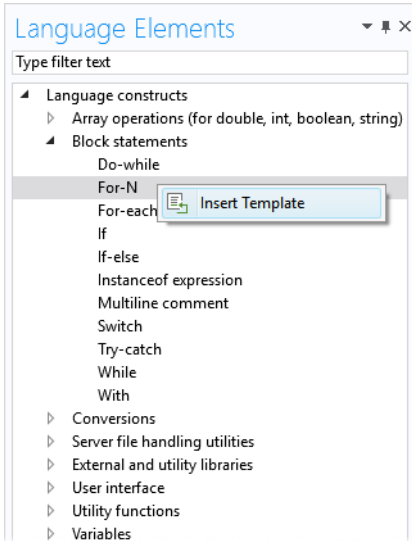


Performing Ctrl+Alt+Click on the form object opens the method in the Method editor. If there is no method associated with the form object, a new local method associated with the form object will be created and opened in the Method editor. If the associated method has a compile error, then this is shown with a different icon, as shown in the figure below.



Language Elements Window

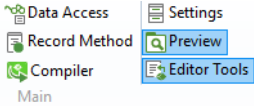
The **Language Elements** window in the Method editor shows a list of some language constructs. Double-click or right-click one of the items in the list to insert template code into the selected method.



See also “Language Element Examples” on page 194.

Editor Tools in the Method Editor

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

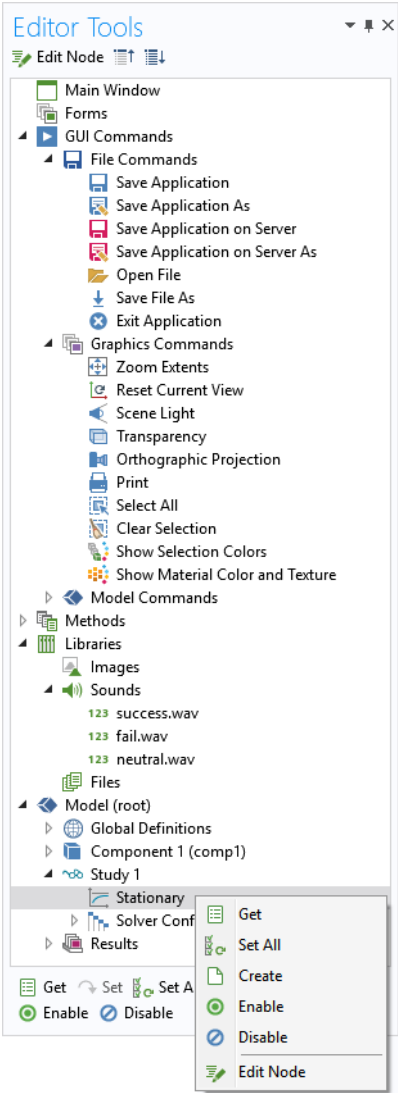


When using the **Editor Tools** window in the Method editor, you can right-click a node in the editor tree to generate code associated with that node. Depending on the node, up to eight different options are available:

- **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 57.

KEYBOARD SHORTCUTS

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

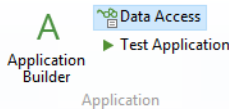
```
model.result("pg3").feature("surf1").create("hght1", "Height");
```

If you position the mouse pointer in "surf1" and press F11 on the keyboard, right-click and select **Go to Node**, or click **Go to Node** in the ribbon, then the corresponding **Surface** plot note 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 321.

Data Access in the Method Editor

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

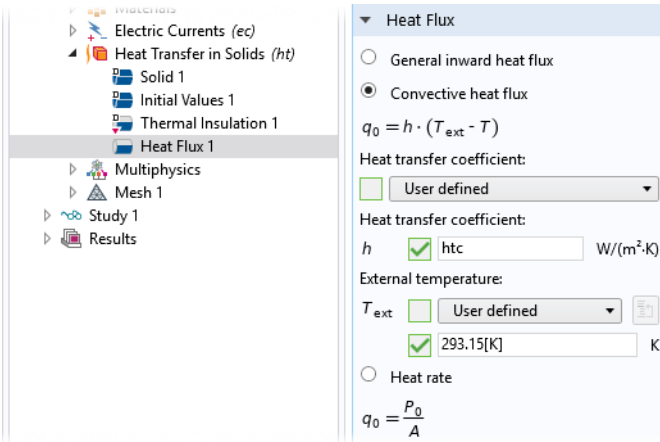


Alternatively, for certain form objects, you can click the **Data Access** button in the header of the **Source** section of the **Settings** window. See also “Data Access in the Form Editor” on page 99.

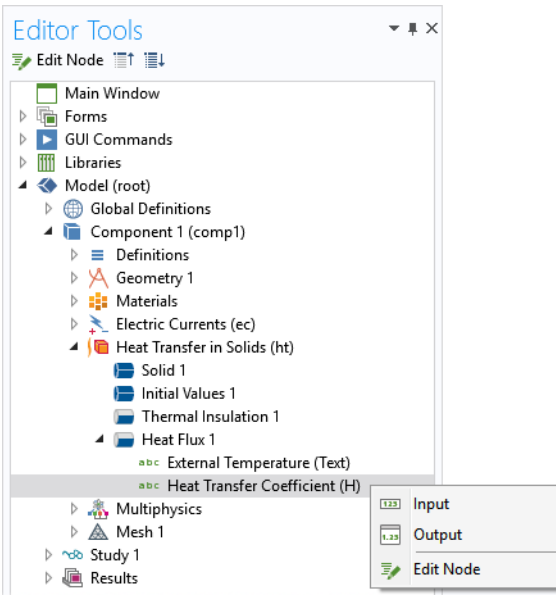
Data Access needs to be enabled this way because a model typically contains hundreds or even thousands of properties that could be accessed, and the list would be too long to be practical.

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.



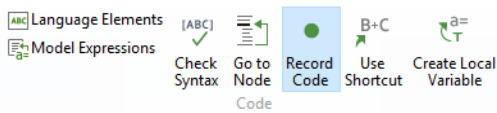
In the example above, **Get** and **Set** for the **Heat transfer coefficient** and the **External temperature** properties will generate the following code:

```
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]");
```

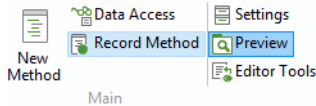
Recording Code

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.



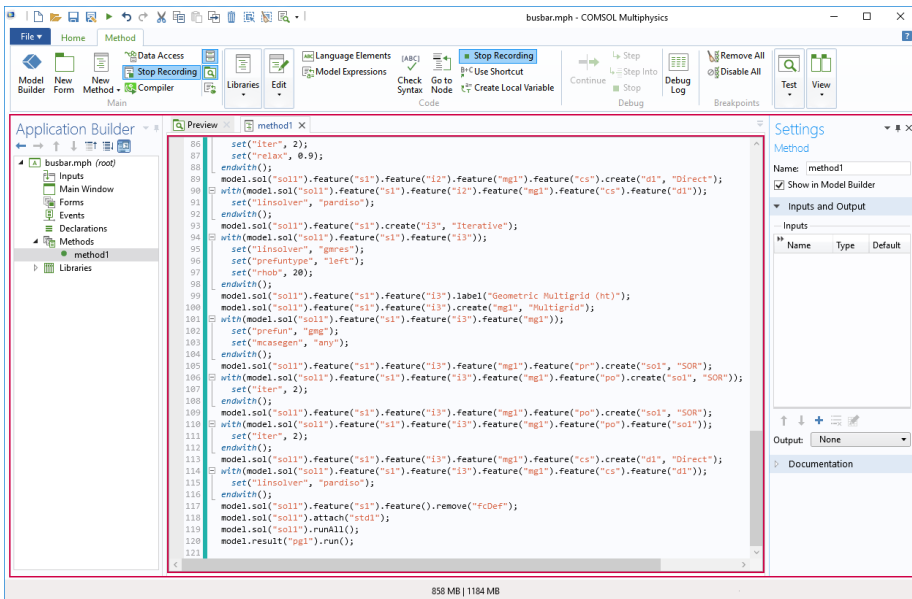
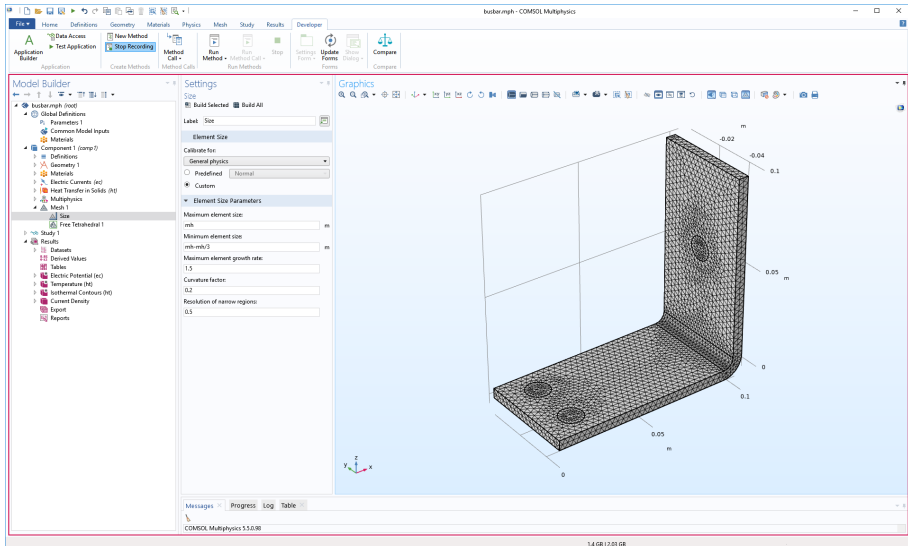
Certain operations in the application tree can also be recorded, including methods used to modify the user interface while the application is running such as changing the color of a text label.

To record a new method, click the **Record Method** button in the **Main** section of the Form editor or Method editor ribbon.

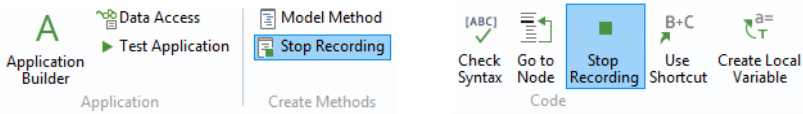


You can also click the **Record Method** button in the **Developer** tab of the Model Builder ribbon.

While recording code, the COMSOL Desktop windows are surrounded by a red frame:



To stop recording code, click one of the **Stop Recording** buttons in the ribbon of either the Model Builder or the Application Builder.



The previous section on **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 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.physics("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 196.

To generate code corresponding to changes to the application object, use **Record Code** or **Record Method**, then go to the Form editor and, for example, change the appearance of a form object. The following code corresponds to changing the color of a text label from the default **Inherit** to **Blue**:

```
with(app.form("form1").formObject("textlabel1"));  
    set("foreground", "blue");  
endwith();
```

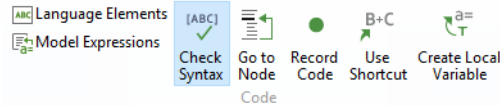
For more information on modifying the model object and the application object, see the *Application Programming Guide*.

Use the tools for recording code to quickly learn how to interact with the model object or the application object. The autogenerated 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 **Data Access** while recording, you can, for example, extract a parameter value using `get`, process its

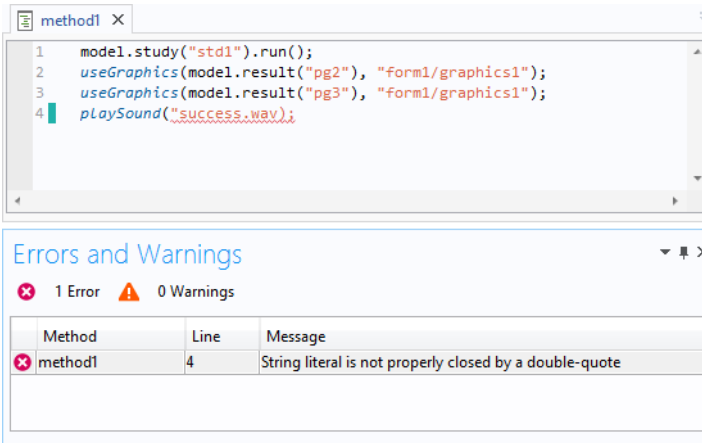
value in a method, and set it back into the model object using `set`. For more information on **Data Access**, see “Data Access in the Method Editor” on page 173.

Checking Syntax

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

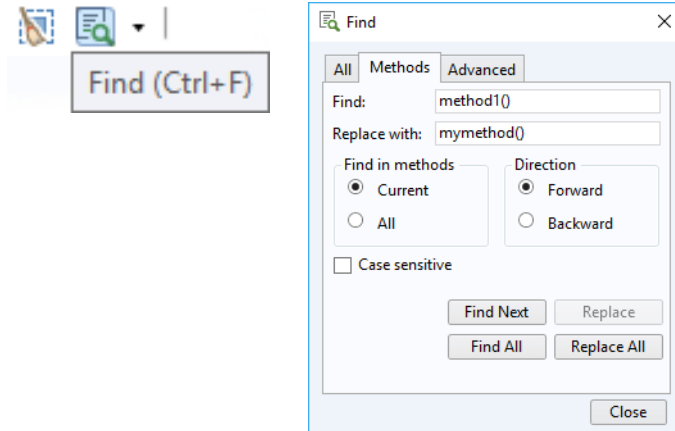


In addition to messages in the **Errors and Warnings** window, syntax errors are indicated with a wavy red underline, as shown in the figure below.



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.

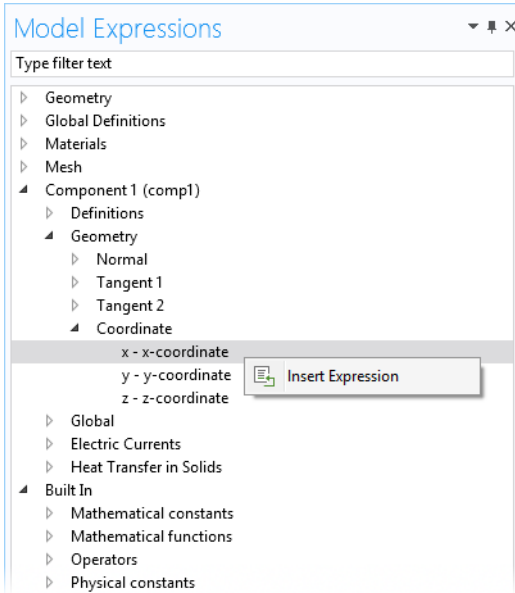


The Quick Access Toolbar is located above the ribbon to the left, in the COMSOL Desktop user interface.

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

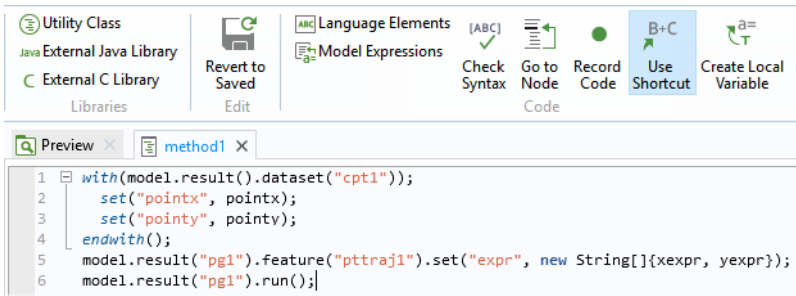
Model Expressions Window

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:



Use Shortcut

If you look at the example below, you will notice that the two last lines of code begins with `model.result("pg1")`.



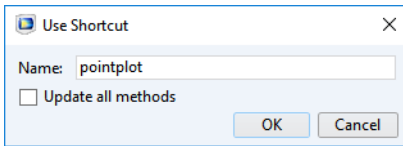
The **Use Shortcut** button simplifies code by replacing these instances with a variable name.

In the example above, the mouse pointer has been positioned at the first occurrence of `model.result("pg1")`. Click the **Use Shortcut** button to transform the source code into what is shown in the figure below.

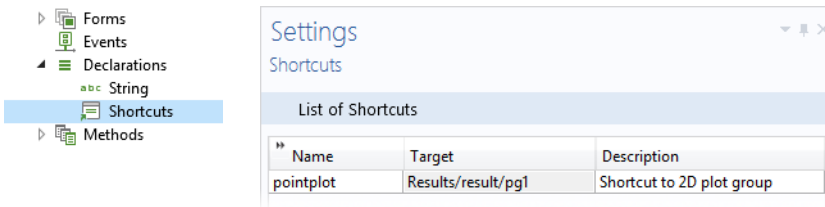


```
1 with(model.result().dataset("cpt1"));
2     set("pointx", pointx);
3     set("pointy", pointy);
4 endwith();
5 pointplot.feature("pttraj1").set("expr", new String[] {xexpr, yexpr});
6 pointplot.run();
```

The code starting with the prefix `model.result("pg1")` has been replaced with the variable `pointplot`. When you click the **Use Shortcut** button, a **Use Shortcut** dialog box opens where you can enter a suitable variable name in the **Name** field, in this case `pointplot`.



This variable is stored as a shortcut in the **Declarations** node, as shown in the figure below together with the corresponding **Settings** window.



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:

```
64 with(model.result("pg1"));
65     set("looplevel", 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:

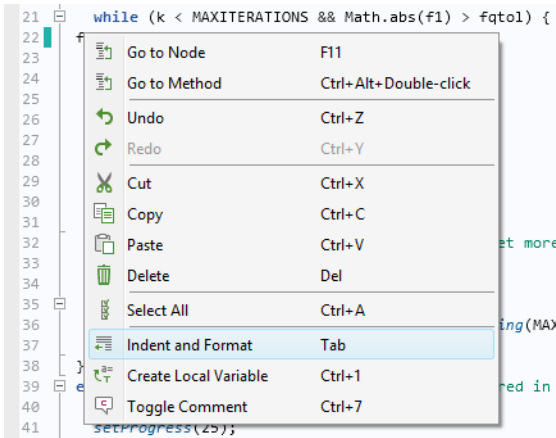
- Keywords, such as `if`, `else`, `for`, `while`, `double`, and `int` are displayed in bold blue font
- Built-in methods are displayed in italic blue font
- Strings are displayed in red font
- Comments are displayed in green font
- The remainder of the code is displayed in black font

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, use the Tab key on your keyboard to automatically indent a line of code and insert white spaces where needed. As an alternative, you can

right-click in the method editor and select **Indent and Format**, as shown in the figure below.



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**.

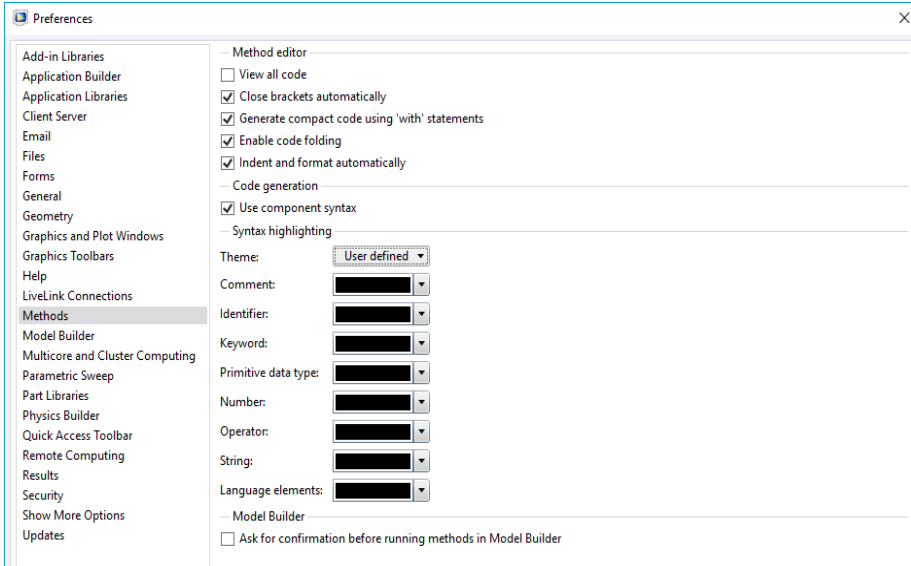
Using the context menu shown above, when right-clicking, you can toggle comments on and off for an entire block of code that you have selected. This is available by selecting **Toggle Comment** from the menu or the keyboard shortcut Ctrl+7.

THE NAME OF A METHOD

The **Name** of a method is a text string without spaces. The string can contain letters, numbers, and underscores. 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 196.

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.

Selecting the check box **Indent and format automatically** will ensure that code is consistently indented and formatted.

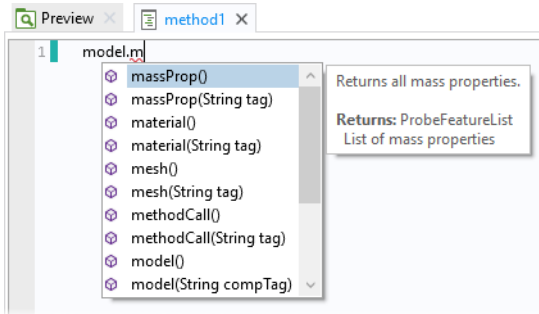
The **Use component syntax** option will generate method syntax that also includes the model component scope.

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.

Clear the option **Ask for confirmation before running methods in the Model Builder** if you do not want to confirm when running methods in this way.

Ctrl+Space and Tab for Code Completion

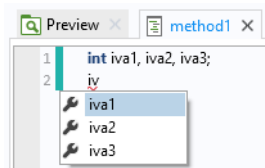
While typing code in the Method editor, the Application Builder can provide suggestions for code completions. The list of possible completions are shown in a separate completion list that opens while typing. In some situations, detailed information appears in a separate window when an entry is selected 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.

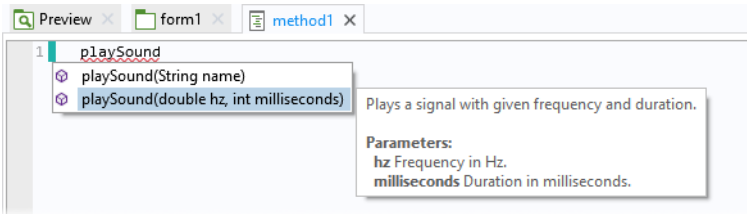
If the list is long, you can filter by typing the first few characters of the completion you are looking for.

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



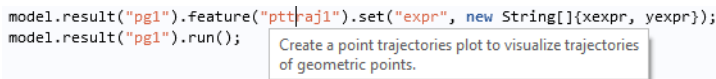
In the example above, only variables that match the string `iv` are shown. This example shows that variables local to the method also appear in the completion suggestions.

You can also use Ctrl+Space to learn about the syntax for the built-in methods that are not directly related to the model object. Type the name of the command and use Ctrl+Space to open a window with information on the various calling signatures available.

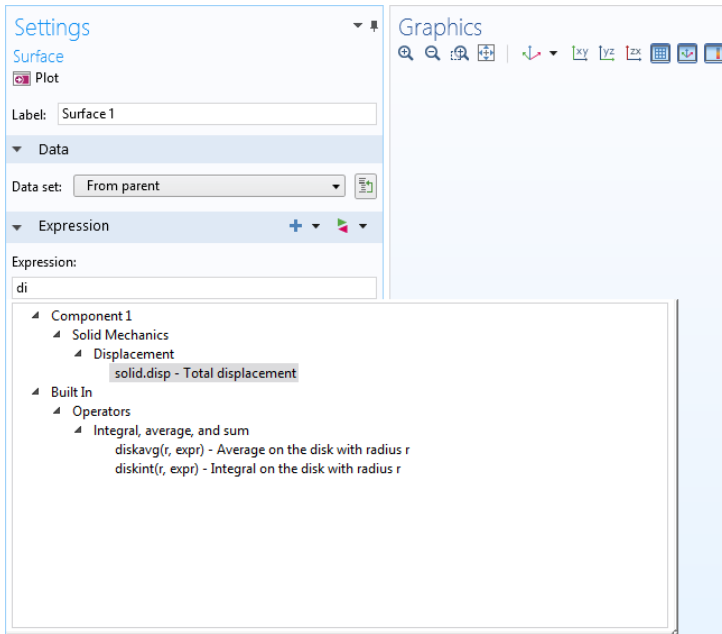


For a list of available built-in methods, you can use the **Language Elements** window described on page 170 or see “Appendix E — Built-In Method Library” on page 324.

Similar information is displayed in a tooltip when hovering over the different parts of a method call, property name, declaration, or shortcut.



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

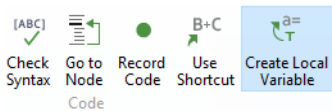


Creating Local Variables

You can automatically set the type of a local variable. For example, you can type

```
x = model.geom()
```

and click the **Create Local Variable** button in the Code group of the Method tab in the ribbon.



The code is then changed to

```
GeomList x = model.geom()
```

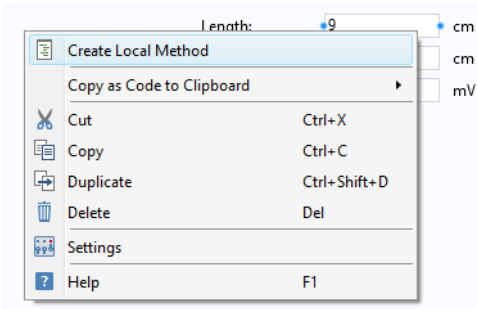
where GeomList is the data type of model.geom().

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.

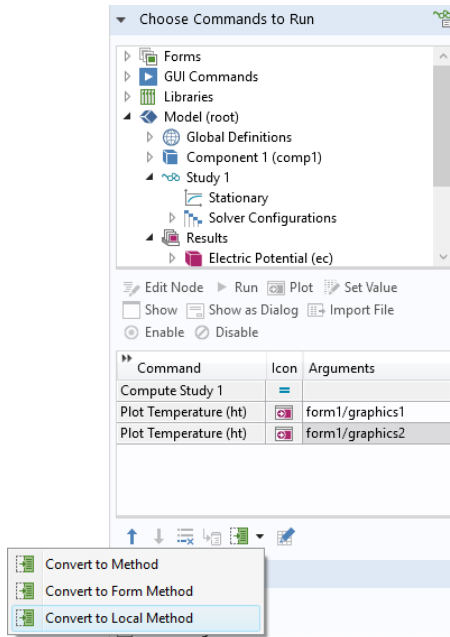
```
main: checkbox1: onDataChange X
1 setFormObjectEditable("main/inputfield1", !findlength);
2 setFormObjectEditable("main/inputfield5", findlength);
3 setFormObjectEnabled("main/inputfield5", findlength);
```

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.

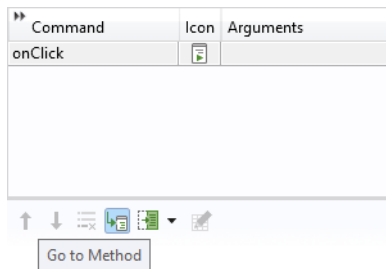


LOCAL METHODS FOR BUTTONS, MENU ITEMS, AND GLOBAL EVENTS

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



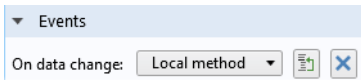
The function of this button is similar to the **Convert to Method** and **Convert to Form Method** buttons, described in the section “Creating a New Method” on page 17. The only difference is that it creates a local method not visible in the global method list in the application tree. It also opens the 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, you are unable to use **Convert to Method** when there is a local method present in the command sequence.

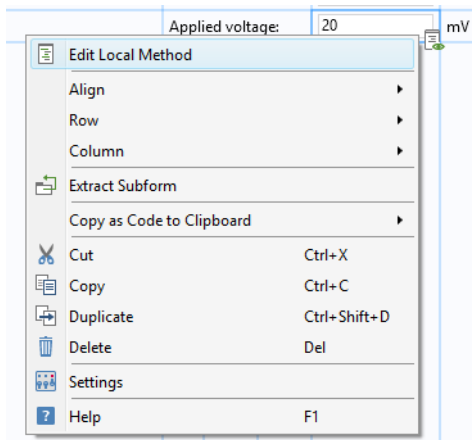
LOCAL METHODS FOR FORM AND FORM OBJECT EVENTS

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** or **Edit Method** from its context menu.

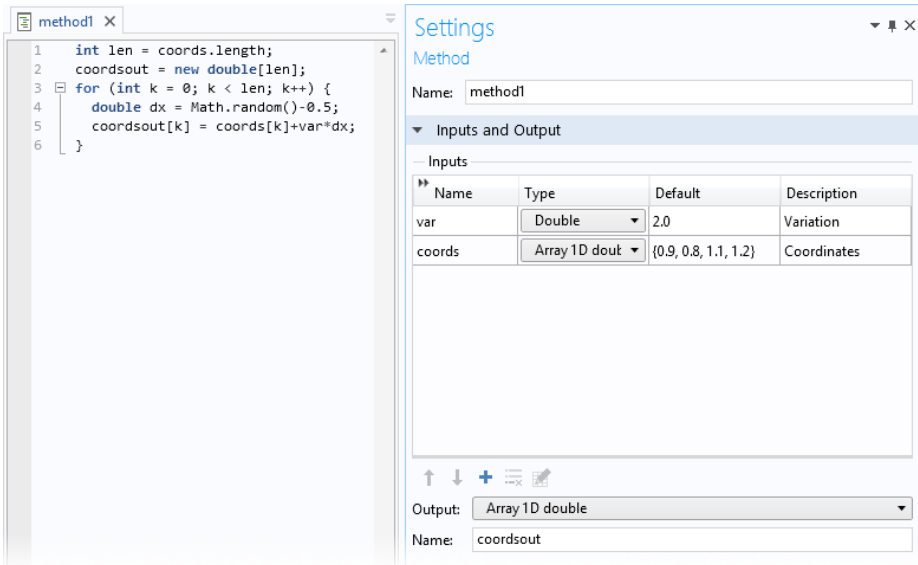


For more information, see “Events” on page 134.

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 argument, `coordsout`. The method adds random values to

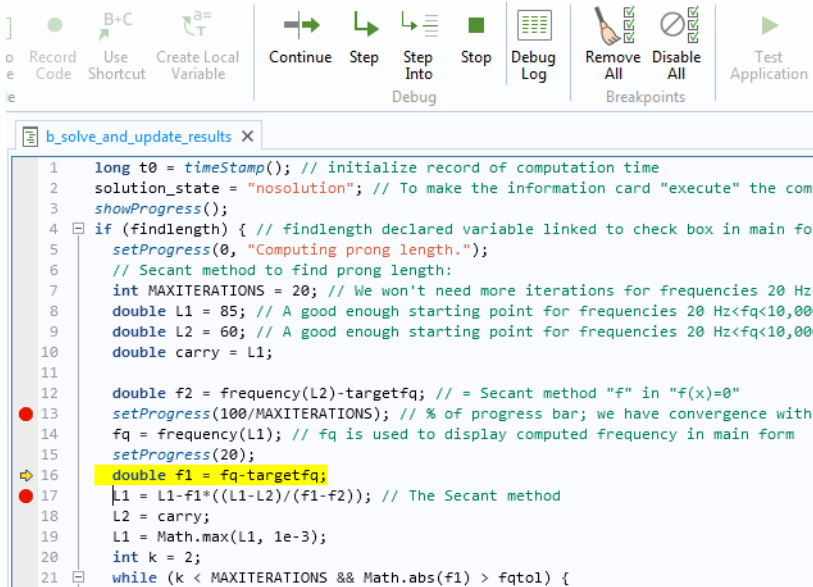
the array, `coords`. The degree of randomness is controlled by the input variable `var`. The new values are stored in the array `coordsout`.



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

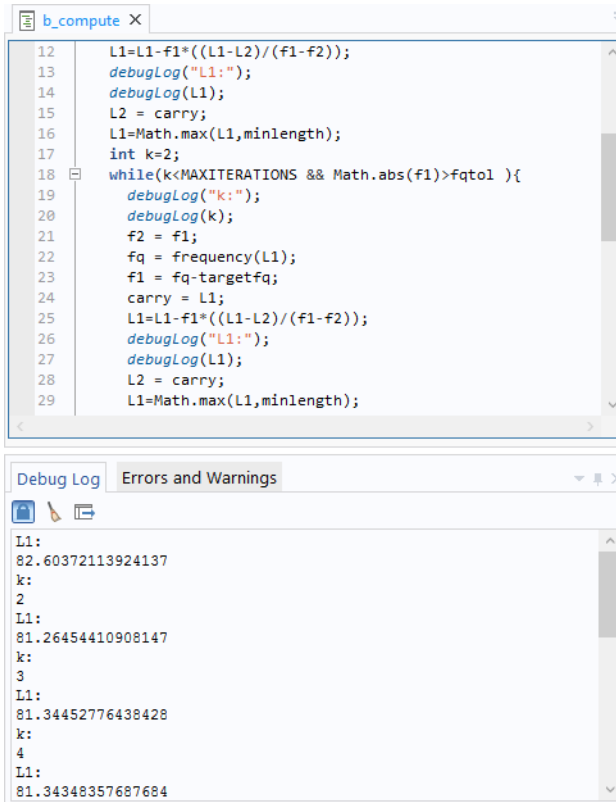
For debugging purposes, click in the gray column to the left of the code line numbers to set breakpoints, as shown in the figure below.



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 breakpoints. 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.



```
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

```
L1:
82.60372113924137
k:
2
L1:
81.26454410908147
k:
3
L1:
81.34452776438428
k:
4
L1:
81.34348357687684
```

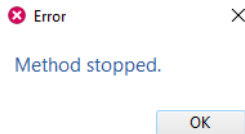
Use the `debugLog` command to display the value of variables in the **Debug Log** window. The code below illustrates using the `debugLog` command to display the values of strings and components of a 1D double array.

```
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 333 and the *Application Programming Guide*.

Stopping a Method

You can stop the execution of a method while testing an application by using the keyboard shortcut Ctrl+Pause. A dialog box appears, as shown below.



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 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*, the *Application Programming Guide*, as well as the *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. This section contains simple examples of some of the most common language elements. For more information and examples, see the *Application Programming Guide* and the *Programming Reference Manual*.

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	!	Logical NOT
6	< <= > >=	comparisons: less than, less than or equal, greater than, greater than or equal
7	== !=	comparisons: equal, not equal
8	&&	binary: logical AND
9		binary: logical OR
10	?:	conditional ternary

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
11	= += -= *= /= %= >>= <<= &= ^= =	assignments
12	,	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 for generating code for setting the value of a global parameter. While in the Method editor, right-click the parameter and select **Set**.

To set the value of the global parameter L to 10 cm:

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

To get the global parameter L and store it in a double variable Length:

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

The evaluation is in this case with respect to the base **Unit System** defined in the model tree root node.

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]");
```

To return the value of a parameter in a different unit than the base Unit System, use:

```
double Length_real = model.param().evaluate("L","cm");
```

If the parameter is complex valued, the real and imaginary part can be returned as a double vector of length 2:

```
double[] realImag = model.param().evaluateComplex("Ex", "V/m");
```

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.");
}

```

Running Methods in the Model Builder

Running methods in the Model Builder is similar to calling methods from applications with the most important difference being that from the Model Builder methods can directly modify the model object in the current session. Running methods from the Model Builder can be used to automate modeling tasks that consist of several manual steps. For example, in a model with multiple studies, you can record code for the process of first computing Study 1; then computing Study 2, which may be based on the solution from Study 1; and so on. From the Model Builder you can call methods directly through **Method Calls** or the **Developer** tab in the ribbon, described later in this section. You can call methods indirectly through **Settings Forms**, for example, by calling a method at the click of a button. For more information on **Settings Forms**, see “Using Forms in the Model Builder” on page 121.

In contrast to methods that are called from applications, methods called from the Model Builder cannot use built-in graphics methods such as `printGraphics`, `useGraphics`, and `zoomExtents`. This restriction is due to the fact that a **Settings**

Form cannot include a graphics object (this restriction will be lifted in future versions).

Methods called in the Model Builder may have input and output arguments. Input arguments to such methods that are called directly, and not indirectly from a **Settings Form**, are given by adding a **Method Call** node under **Global Definitions**, see “Method Calls” on page 202.

CONTROLLING WHICH MODEL TREE NODE SHOULD BE ACTIVE

To control which model tree node should be active after running a method in the Model Builder you can use the built-in method `selectNode`. For example, a method modifying the geometry can have as its last line of code:

```
selectNode(model.component("comp1").geom("geom1"));
```

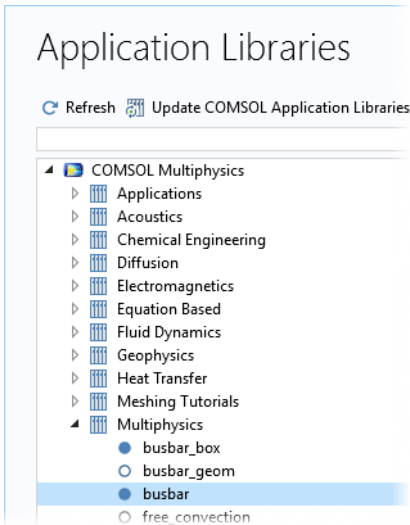
which will display the geometry and select the **Geometry** node.

The method `selectNode` has no function when used in an application.

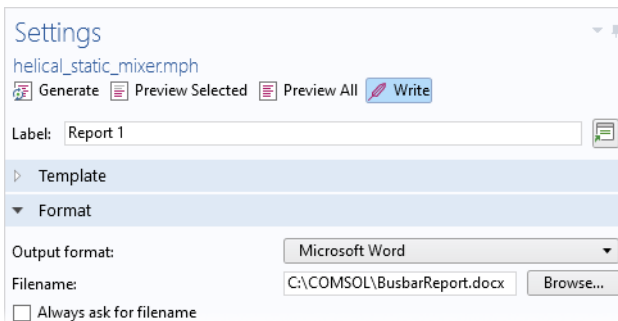
GENERATING A REPORT AUTOMATICALLY AFTER COMPUTING

As an example of using a method from the Model Builder, consider the process of first computing the solution and then generating a report. This can be automated by first recoding the corresponding operations in the Model Builder and then running a method.

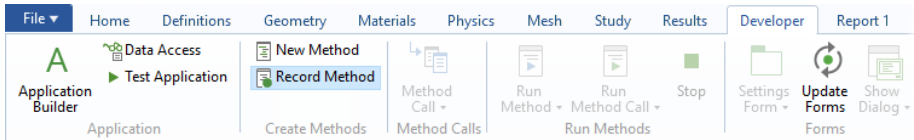
Let's start from the busbar example described in the book *Introduction to COMSOL Multiphysics*. You can load this example MPH file from the Application Libraries, as shown in the figure below.



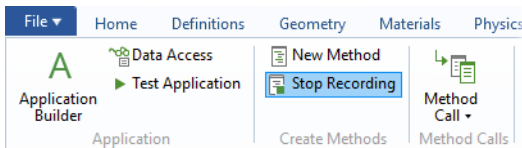
- **Open** the model.
- In the model tree, right-click the **Report** node under **Results**.
- Select **Brief Report**.
- Change the **Output format** to Microsoft® Word (this example would also work with the default HTML format).
- Click the **Browse** button and select a file name in a location on your system that you have write permissions to, for example:
C:\COMSOL\BusbarReport.docx
- Click **Write** to generate and save the report to file.



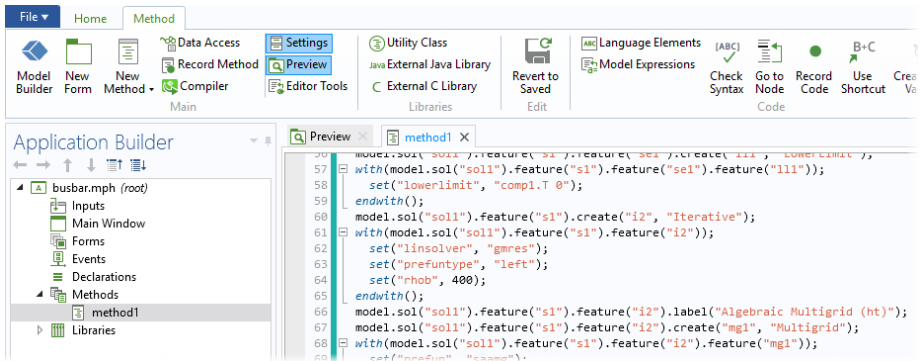
- Close the Microsoft® Word document that was automatically opened.
- Click the **Developer** tab in the ribbon (of the **Model Builder**) and click the **Record Method** button.



- In the model tree, right-click the **Study 1** node and select **Compute** (or use the ribbon option for **Compute**).
- In the model tree, right-click the **Report 1** node and select **Write**.
- Click the **Developer** tab in the ribbon and click on the **Stop Recording** button.

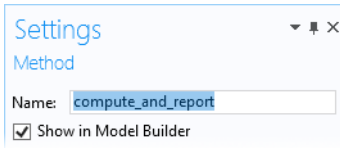


- Close the Microsoft® Word document.
- You can now switch over to the Application Builder, by clicking on the **Application Builder** button in the ribbon, and see the recorded method in the application tree and in the Method editor.

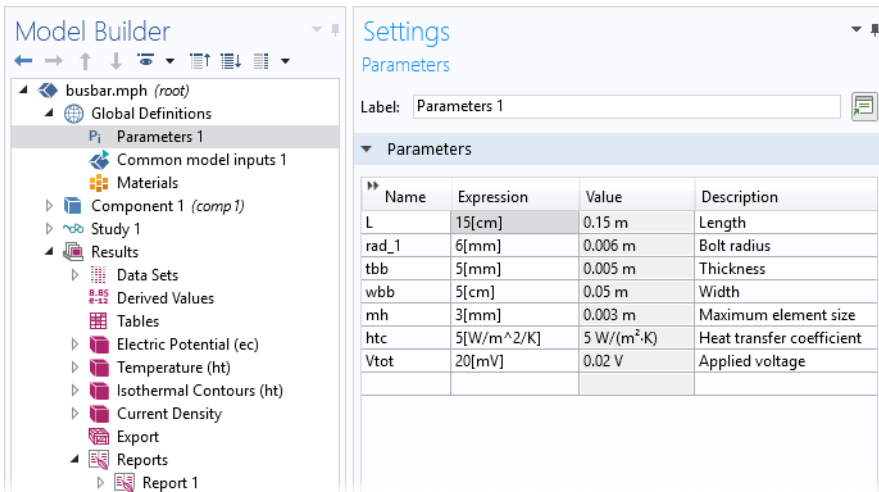


For more information on code generated from the **Study** node, see the *Application Programming Guide*.

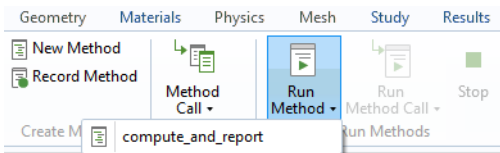
- Here you can inspect and edit the generated code and also change the name of the method to, for example, **compute_and_report**.



- Go back to the Model Builder by clicking on the **Model Builder** button in the ribbon.
- In **Global Definitions > Parameters**, change the **Length** to 15[cm].



- In the ribbon, click on the **Developer** tab and select **compute_and_report** from the **Run Method** menu.

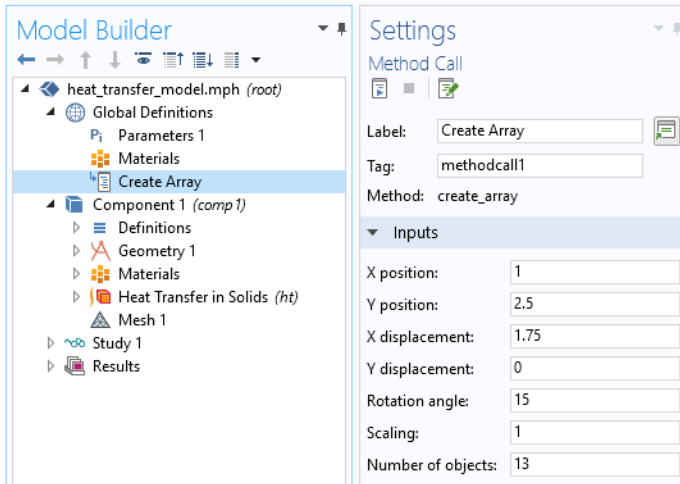


Note that you can create multiple methods and call them from the Model Builder.

METHOD CALLS

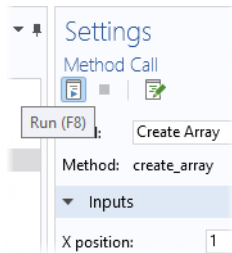
A call, in the Model Builder, to a method for a specific set of input argument values can be made by adding a **Method Call** node under **Global Definitions**. To add

a **Method Call** node, right-click **Global Definitions** and select one of the methods that you have created. The figure below shows a **Method Call** to a method for creating a geometric array.

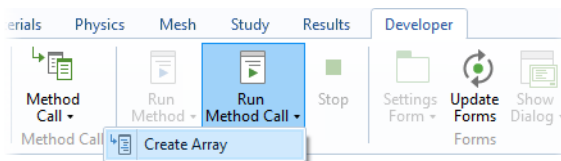


The user interface layout of a **Method Call** cannot be customized. Instead, for customizing use a **Settings Form**, see “Using Forms in the Model Builder” on page 121.

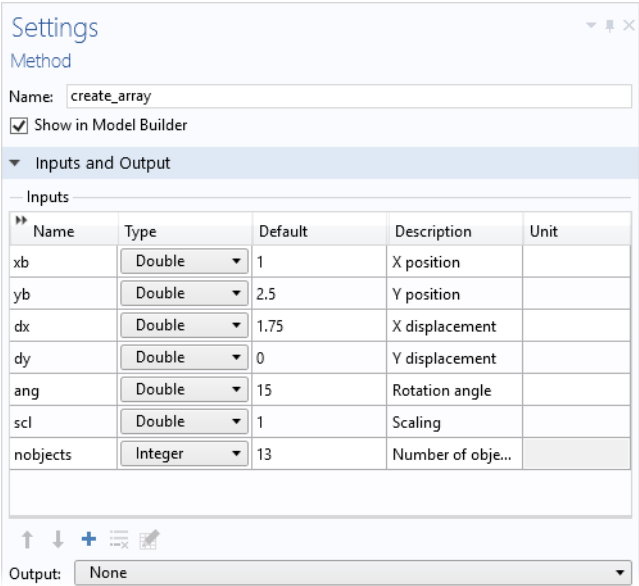
You can run, stop, or edit a **Method Call** by clicking the corresponding toolbar button in the **Settings** window, as shown in the figure below.



This functionality is also available from the **Developer** tab in the ribbon of the Model Builder, as shown in the figure below.



The figure below show the corresponding method's **Settings** window in the Application Builder with the definitions of the input arguments.



You can add multiple **Method Call** nodes for the same method where each call can contain a different set of input argument values.

There is no direct way of using output arguments from a method in the Model Builder. However, you can use calls to the built-in method `message` to display variables used in a method in the **Messages** window in the COMSOL Desktop environment. The following example shows how to display the value of two double variables `width` and `depth` in the **Messages** window:

```
message("Width: "+toString(width));
message("Depth: "+toString(depth));
```

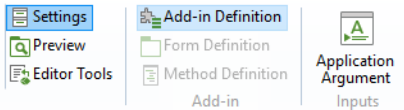
For debugging purposes you can instead use the `debugLog` method described in the section “Debugging” on page 192.

For reusing a **Settings Form** between sessions, you can create an **Add-in**. For more information, see “Creating Add-Ins” on page 205.

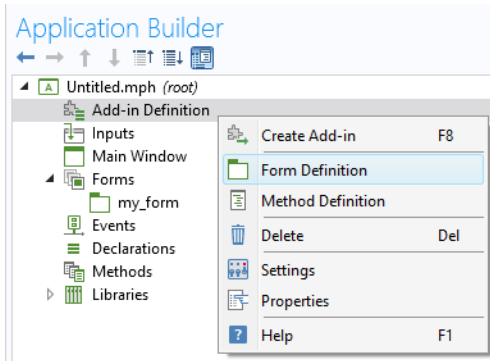
Creating Add-Ins

To customize the workflow in the Model Builder, you can use a **Method Call** or a **Settings Form**. However, these are associated with a specific MPH file, and you may want to reuse them between sessions or share them with colleagues. To make this possible, you can create an add-in based on a **Method Call** or **Settings Form**. Such add-ins can then be stored in a user-defined add-in library. In addition, COMSOL Multiphysics comes with built-in add-in libraries. For the add-ins in the built-in library, you can review their Application Builder settings, including forms and methods, to quickly learn how to build your own add-ins. Creating an add-in is similar to creating an application, with a few differences. Add-ins do not have their own graphics window, but instead use the main **Graphics** window in the Model Builder. An add-in should work, or give controlled error messages, for any type of model.

To create an add-in, start from a form that you have created in the Application Builder and click **Add-in Definitions** in the ribbon, as shown below.



Right-click the **Add-in Definitions** node in the application tree and select **Form Definition**.



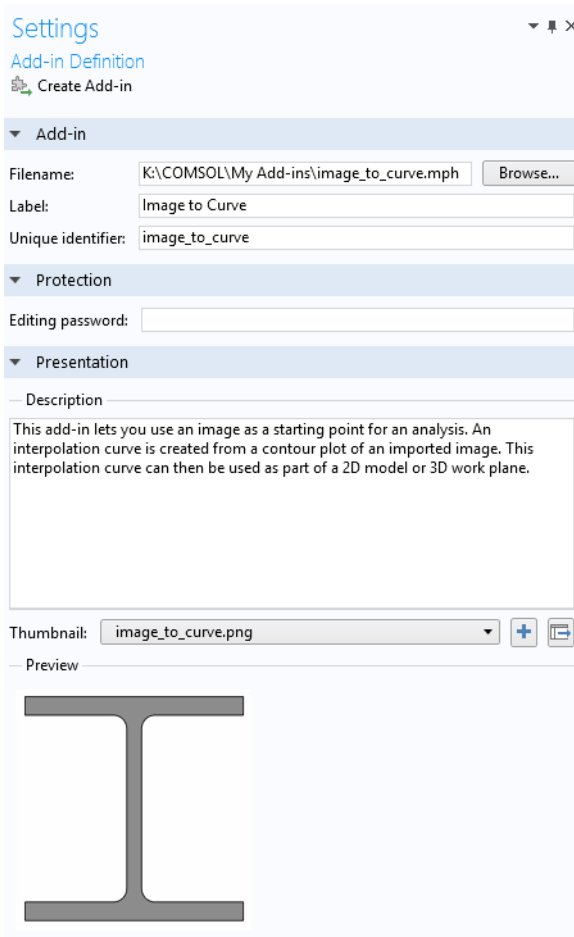
The figure below shows the **Settings** window for the **Form Definition**. Here, you can type a **Label** for the add-in form as well as select which form to use for the add-in. The **Label** will be displayed in the user-defined add-in library. You can select whether the form should be displayed as a **Settings form** in the model tree or as a **Dialog** box. The **Allow multiple settings forms** check box is used to allow for more than one instance of the **Settings form** in the model tree. The **Description** is

displayed in the add-in library and as a tooltip when choosing among add-ins in the ribbon.

The image shows a 'Settings' dialog box with a title bar containing a dropdown arrow, a maximize icon, and a close icon. Below the title bar, the text 'Form Definition' is displayed. The dialog contains the following elements:

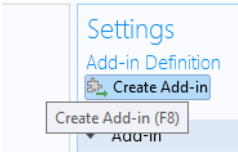
- A 'Label' field with the text 'Image to Curve'.
- A 'Form' dropdown menu currently set to 'main', with a small icon to its right.
- A 'Show as' dropdown menu currently set to 'Settings form'.
- An unchecked checkbox labeled 'Allow multiple settings forms'.
- A section header 'Description' with a downward-pointing arrow.
- A text area containing the description: 'Use the contour plot of an imported image to create an interpolation curve, which can be included as part of a 2D model or 3D work plane.'

Click the **Add-in Definition** node to see its **Settings** window, as shown below.



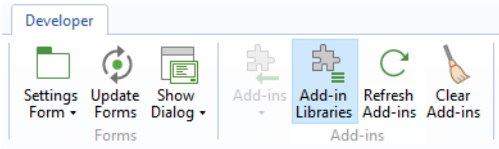
The **Filename** is the location of the add-in MPH file in the user-defined add-in library. This location can be on a shared network drive if you wish to share the add-in with your colleagues. The **Label** will be displayed in the **Add-in Libraries** window. The **Unique identifier** is what identifies the add-in and is intended to be unique for any COMSOL Multiphysics session. The unique identifier is recommended to be in a format similar to <company name>.<Add-in name> ;for example, my_company.my_add-in. The **Editing password** will be applied to the created add-in and is different from the editing password that you can specify in the root node **Settings** window of the MPH file used to create the add-in.

To create the add-in, which is a special type of MPH file, click the **Create Add-in** button.

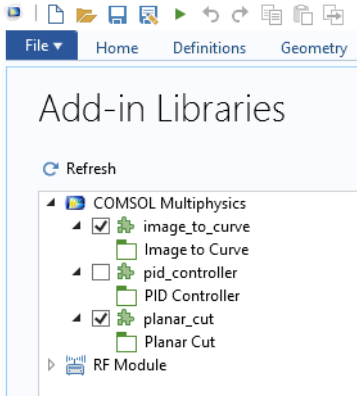


Add-In Libraries

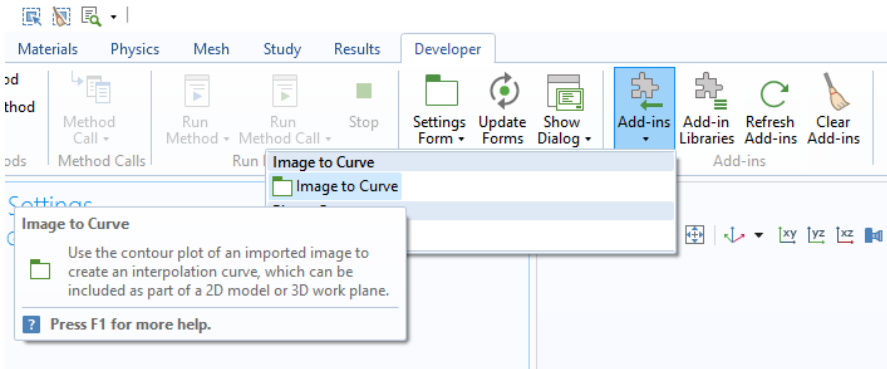
To use an add-in from the **Add-in Libraries**, you first need to enable it. In the **Developer** tab in the Model Builder, click **Add-in Libraries**.



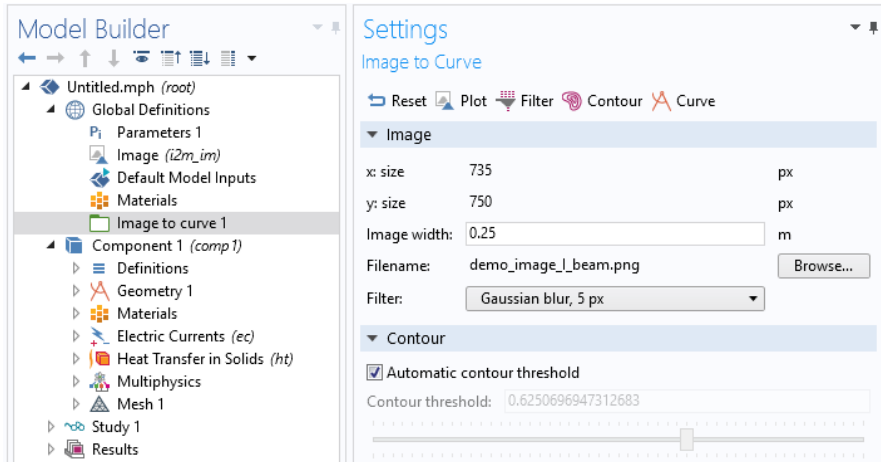
In the list of add-ins, select the check boxes of those add-ins that you want to enable.



Once enabled, the corresponding add-ins will be displayed when clicking the **Add-ins** button in the **Developer** tab.



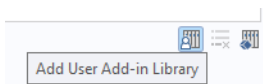
The figure below shows the **Settings Form** for one of the built-in add-ins.



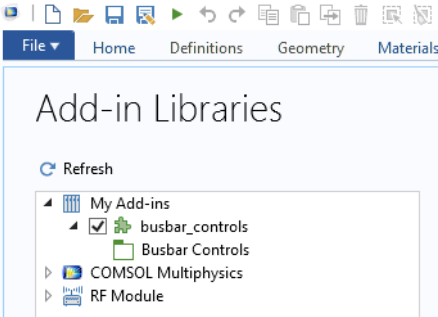
If you want to review and edit the Application Builder settings for a built-in add-in, you can open the corresponding MPH file. In a typical Windows® installation, the built-in add-in library is located at:

C:\Program Files\COMSOL\COMSOL55\Multiphysics\addins

You can browse to a user-defined add-in library by clicking the **Add User Add-in Library** button at the bottom of the **Add-in Libraries** window.

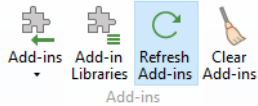


The user-defined add-in library will be displayed alongside the built-in add-in libraries, as shown below.



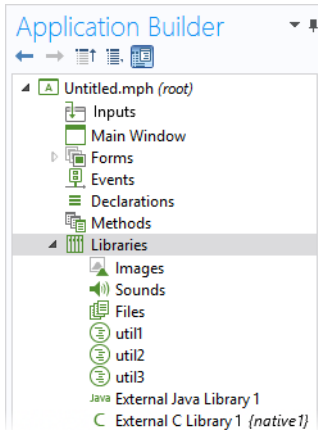
Workflow when Creating and Editing Add-Ins

When creating and editing add-ins, you will find it useful to have two sessions of COMSOL Multiphysics open at the same time: one session for the original add-in MPH file where you work mostly in the Application Builder, and one session for testing the add-in in the Model Builder. When testing an add-in using the Model Builder, make sure to test for a great variety of models, including models of different spatial dimensions as well as models with more than one model component. Use the **Refresh Add-ins** button to make sure you always use an updated version of the add-in you are editing.



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. For more information on using utility classes and external libraries, see the *Application Builder Reference Manual*.



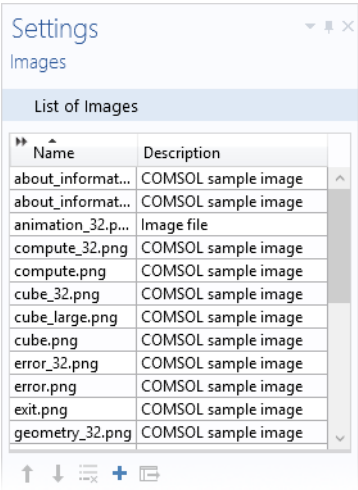
Embedded files can, for example, be referenced in form objects or in methods by using the syntax `embedded:///file1`, `embedded:///file2`, and so on. For example, to reference the image file `compute.png`, use the syntax `embedded:///compute.png`.

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” on page 153, “File Import” on page 265, and “Appendix C — File Handling and File Scheme Syntax” on page 300.

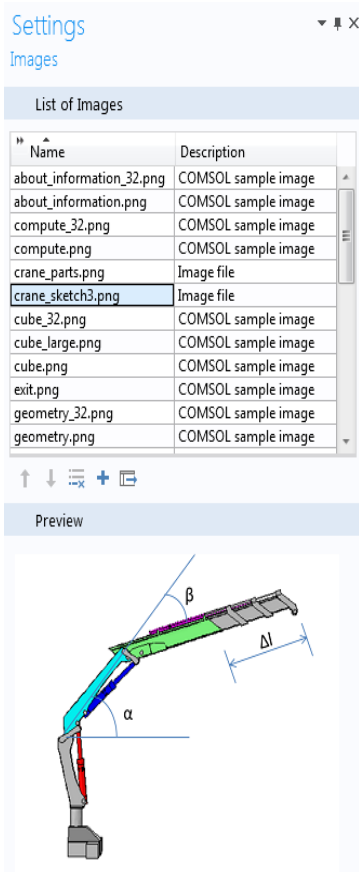
Images

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 available: 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 in the **List of Images**. The image is displayed in the **Preview** section, as shown in the figure below.

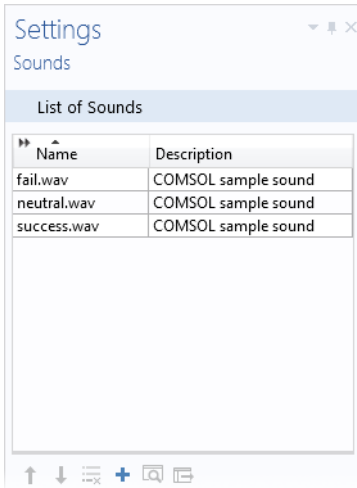


To export a selected image, click the **Export Selected Image File** button to the right of the **Preview** button.

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.

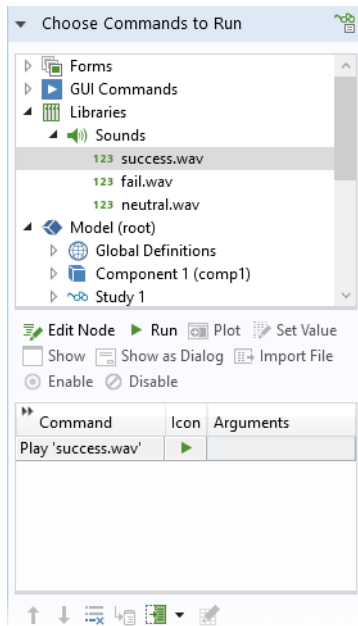


To play a sound, click the name of the sound and then click the **Preview** button below the **List of Sounds**.

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.

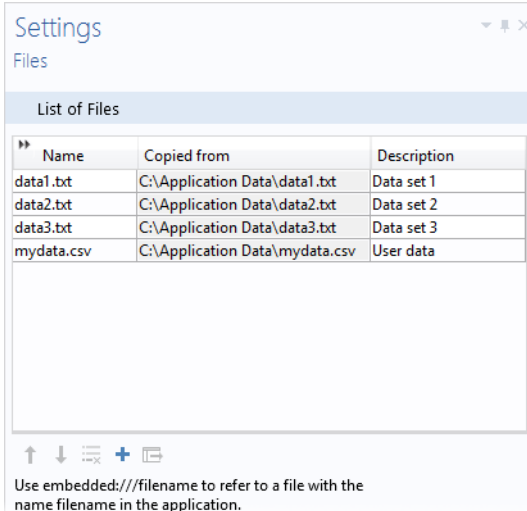


In methods, you can play sounds using the built-in method, `playSound`, such as:

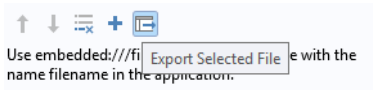
```
playSound("success.wav");
```

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” on page 153, “Appendix C — File Handling and File Scheme Syntax” on page 300, and “File Methods” on page 325.

Appendix A — Form Objects

This appendix provides information about forms and form objects and expands upon the section “The Form Editor” on page 48. 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
 - Video
 - 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
 - Hyperlink
 - Toolbar
 - Spacer

Toggle Button

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

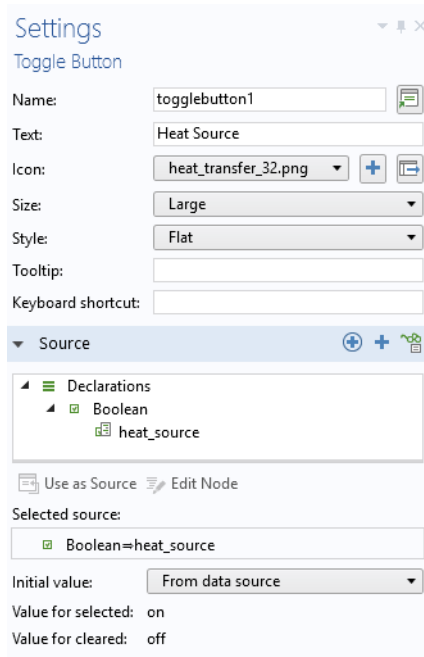


The information in this section is also applicable to **Menu Toggle Item** and **Ribbon Toggle Item**.

USING A TOGGLE BUTTON TO ENABLE AND DISABLE A HEAT SOURCE

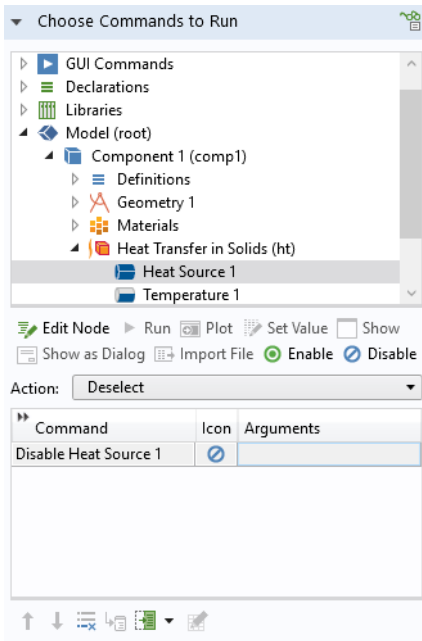
The two states of a toggle button are stored by linking it to a Boolean variable. The figure below shows the **Settings** window of a button that enables and disables

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

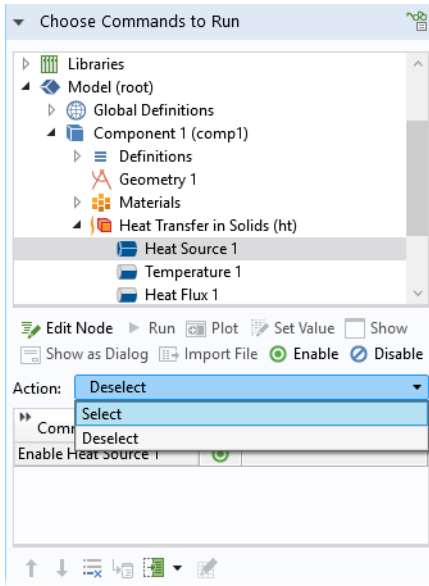


Enabled corresponds to the Boolean variable `heat_source` being equal to `true`, which in turn corresponds to the toggle button being selected. Disabled corresponds to the Boolean variable `heat_source` being equal to `false`, which in turn corresponds to the toggle button being deselected.

Below the **Source** section is the **Choose Commands to Run** section, 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.



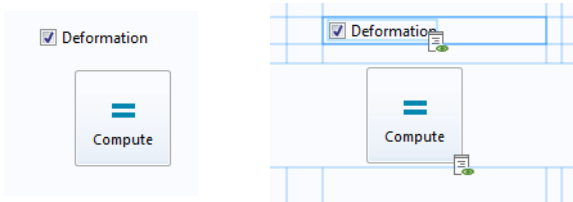
A toggle button is similar to a check box in that it is linked to a Boolean variable. For a toggle button, you define the action by using a command sequence, whereas for a check box, you define the action by using an event. This is described in the next section.

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.

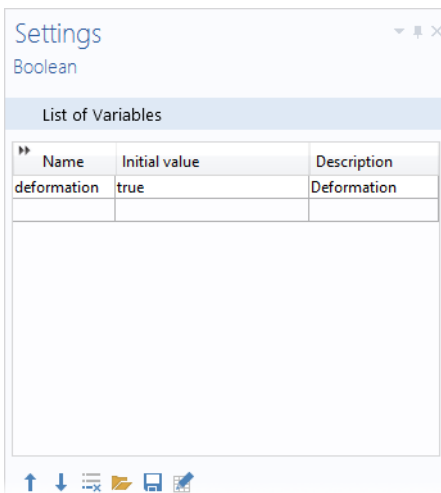
USING A CHECK BOX TO CONTROL VISUALIZATION

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

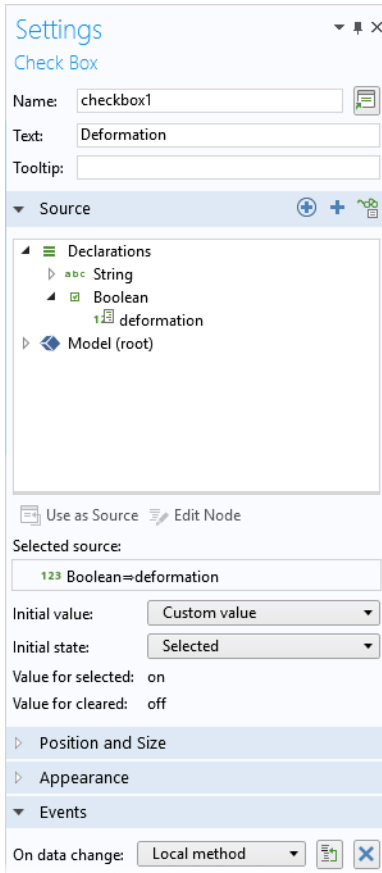


The screenshot on the left shows the running application. The screenshot on the right shows the corresponding form objects in grid layout mode.

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.

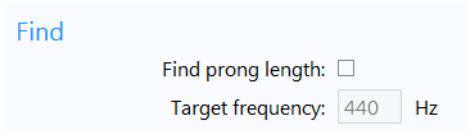
The **Initial value** of the variable `deformation` is overwritten by the **Value for selected** (on) or the **Value for cleared** (off) and does not need to be edited. When used in methods, the values on and off are aliases for `true` and `false`, respectively. These values can be used as Booleans in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

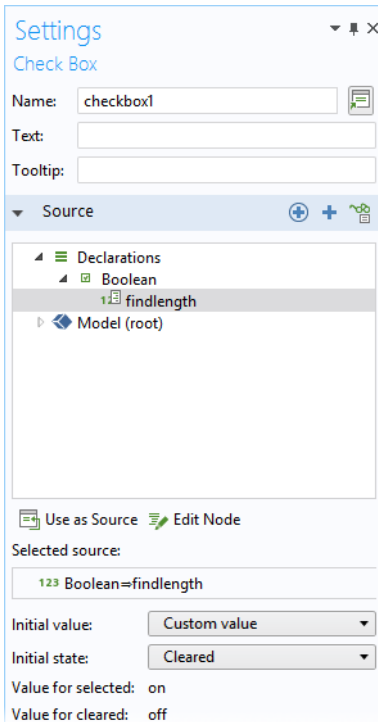
```
model.result("pg1").feature("surf1").feature("def").active(deformation);  
useGraphics(model.result("pg1"), "graphics1");
```

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";
```

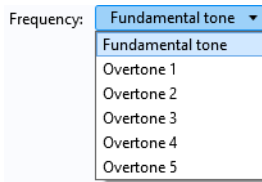

Combo Box

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.

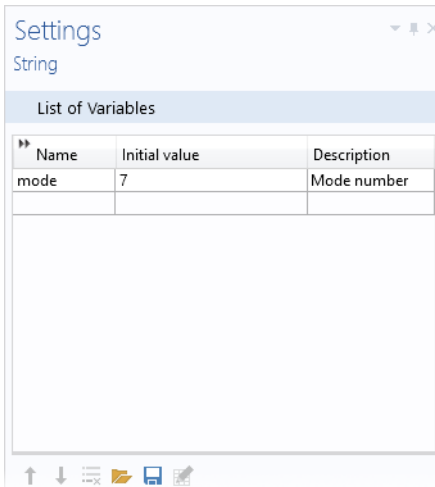
USING A COMBO BOX TO CHANGE PARAMETERS IN RESULTS

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:

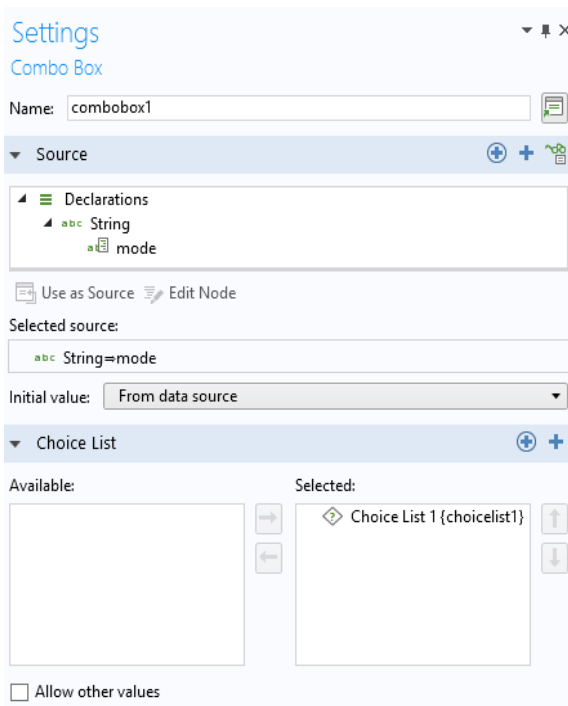


In this example, the combo box is used to control the value of a string variable **mode**. The figure below shows the **Settings** window for this variable.



Selecting the Source

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

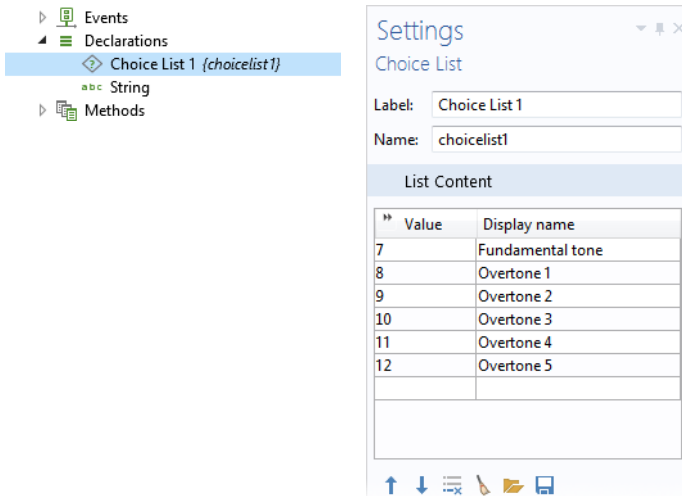


In the **Source** section, you select a scalar variable that should have its value controlled by the combo box and click **Use as Source**. In the **Initial values** list of the **Settings** window of the combo box, choose a method to define a default value for the combo box. The options are **First 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.



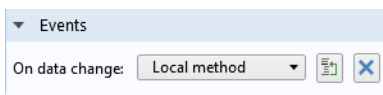
The string variable `mode` is allowed to have one of these six values: 7, 8, 9, 10, 11, or 12. The text strings in the **Display name** column are shown in the combo box.

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” on page 151.

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.



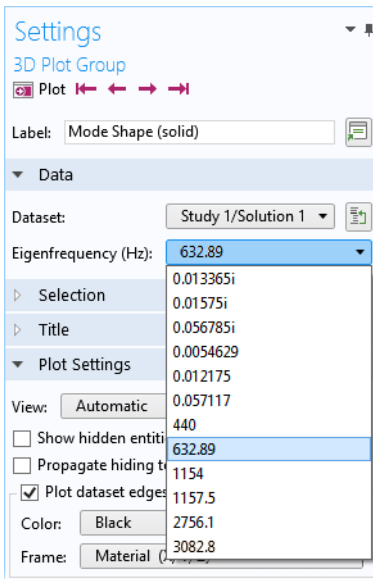
The code for the local method is listed below.

```
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, in the **Developer** tab, click **Record 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.

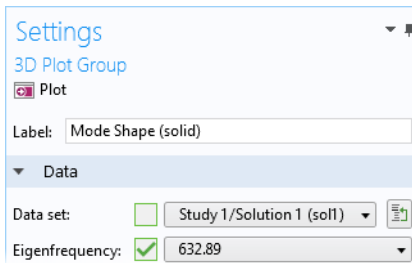
```
with(model.result("pg1"));
  set("looplevel", new String[]{"8"});
endwith();
model.result("pg1").run();
```

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

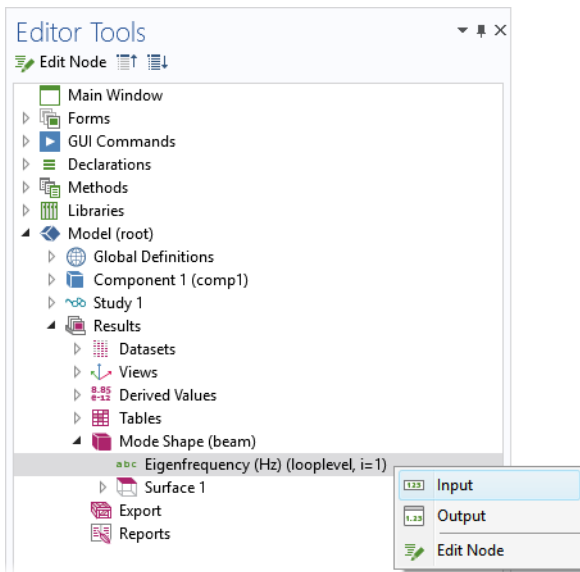
associated with the combo box, copy the code from `method1`. Then, delete `method1`.

Using Data Access

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

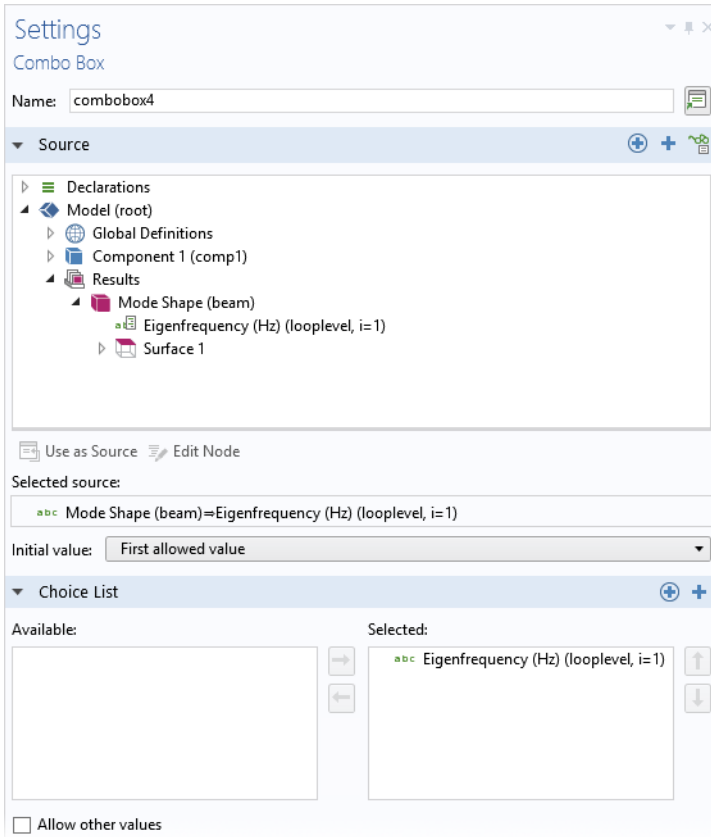


In the **Editor Tools** window, the **Eigenfrequency** parameter is visible as **Loop Level**. To create a combo box, right-click **Loop Level** and select **Input**.



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.

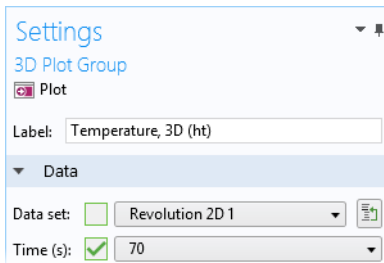


The choice list **Loop Level** is automatically generated when inserting a combo box using **Editor Tools**. Note that a choice list generated in this way is not displayed under the **Declarations** node and cannot be modified by the user. For greater flexibility, such as giving names to each parameter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

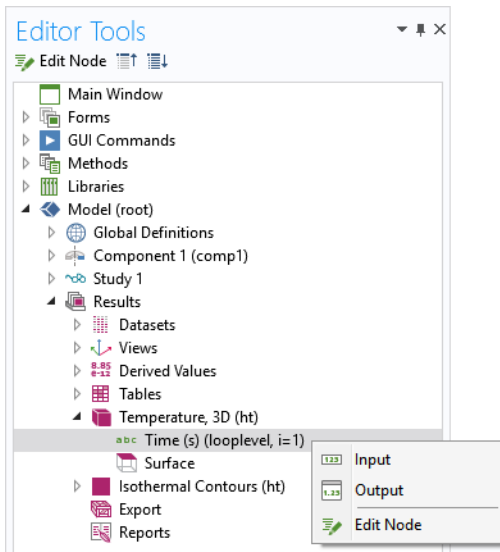
USING A COMBO BOX TO CHANGE TIMES

The time parameter list specified in a **Time Dependent** study step can be used in many places under the **Results** node. In an application, the individual time parameters can be accessed in a similar way to what was described in the last section for parameters, by using **Data Access** in combination with **Editor Tools**.

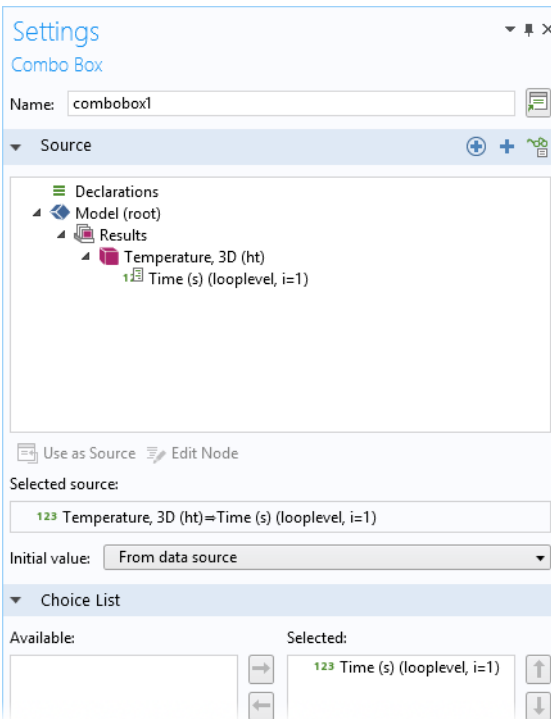
In the **Settings** window in the figure below, **Data Access** has been used to access the **Time** parameter list in a temperature plot.



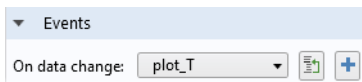
In **Editor Tools**, a handle to the **Time** list is now available, as shown in the figure below.



By selecting **Input**, you can create a combo box using it as **Source**, as shown in the figure below.



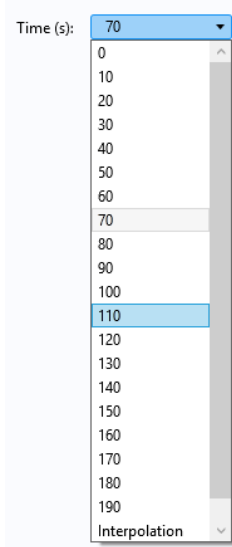
The combo box can be used for multiple purposes, for example, to update a plot corresponding to a different time parameter. In order for a plot to automatically update when a user uses the combo box to select a new time parameter, add an event to the combo box at the bottom of its **Settings** window. In the figure below, a method `plot_T` is called for updating a temperature plot.



The line of code below shows the contents of the method `plot_T`:

```
model.result("pg1").run();
```

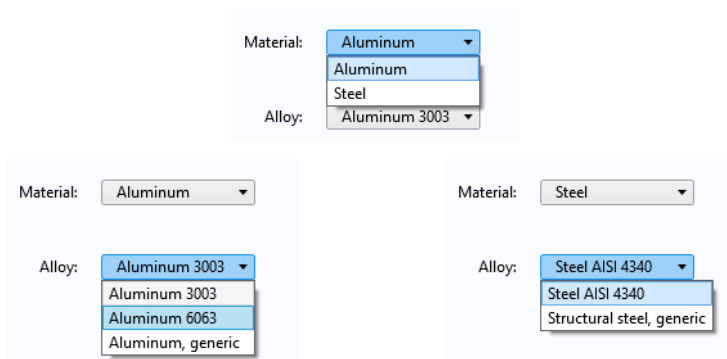

The end result is a combo box in the application user interface, shown in the figure below, which automatically updates a temperature plot when the user selects a new value for the **Time** list.



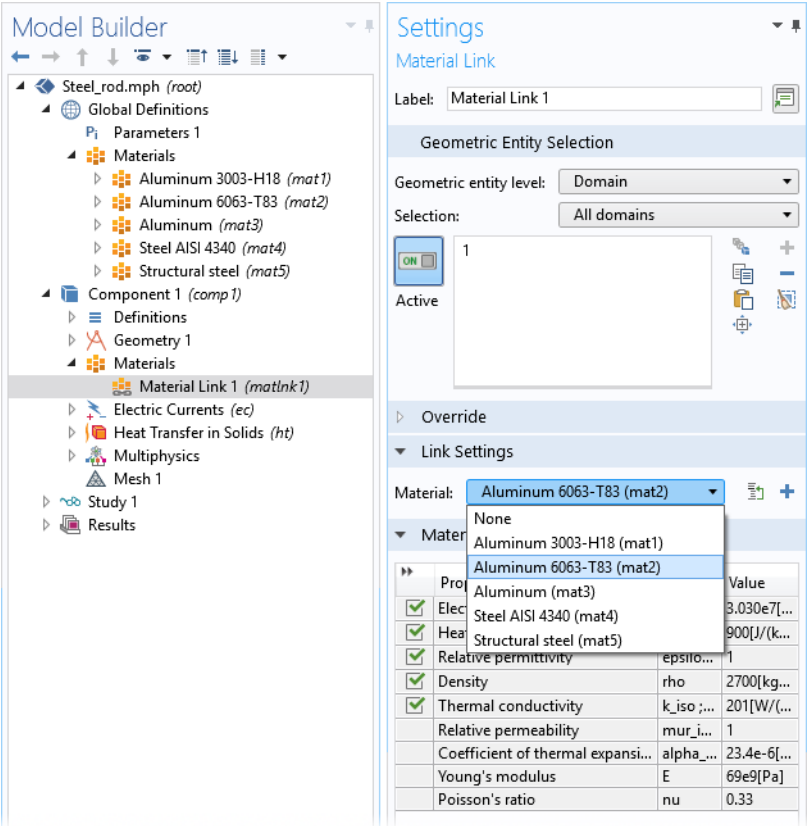
USING A COMBO BOX TO CHANGE MATERIAL

Consider an application where combo boxes are used to select the material. In this case, an activation condition (see “Activation Condition” on page 152) 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.



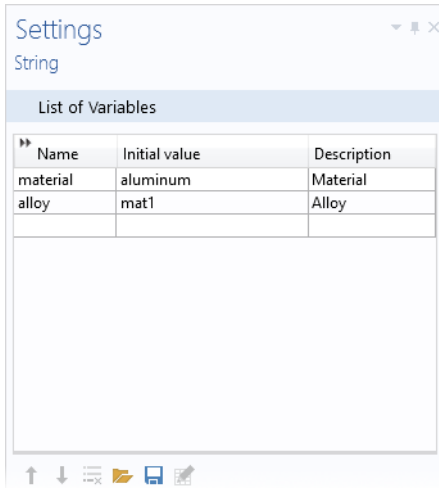
The material choice is implemented in the embedded model using global materials and a material link, as shown below.



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.



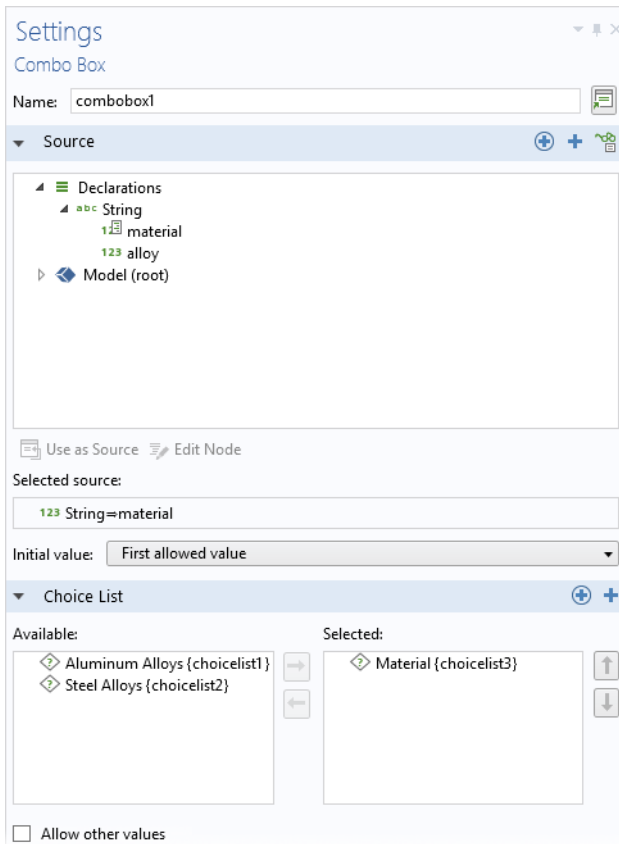
The application utilizes three choice lists: **Aluminum Alloys**, **Steel Alloys**, and **Material**.

Activation Condition

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

- ▲ Declarations
 - ◆ Aluminum Alloys {choicelist1}
 - ◆ Activation Condition {actcond1}
 - ◆ Steel Alloys {choicelist2}
 - ◆ Activation Condition {actcond1}
 - ◆ Material {choicelist3}
 - ◆ String

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.

Settings

Choice List

Label: Material

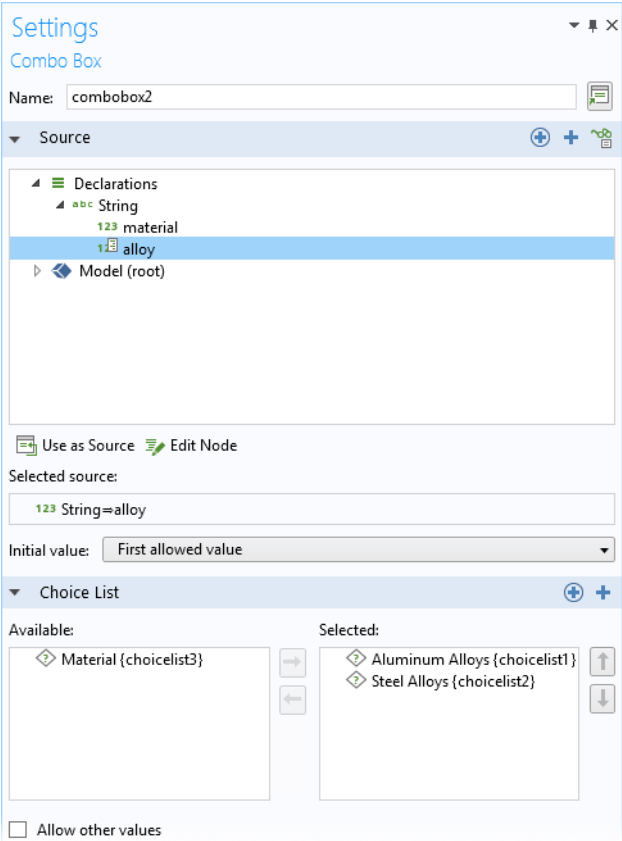
Name: choicelist3

List Content

Value	Display name
aluminum	Aluminum
steel	Steel

↑ ↓ ☰ ✎ 📄

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.

Settings

Choice List

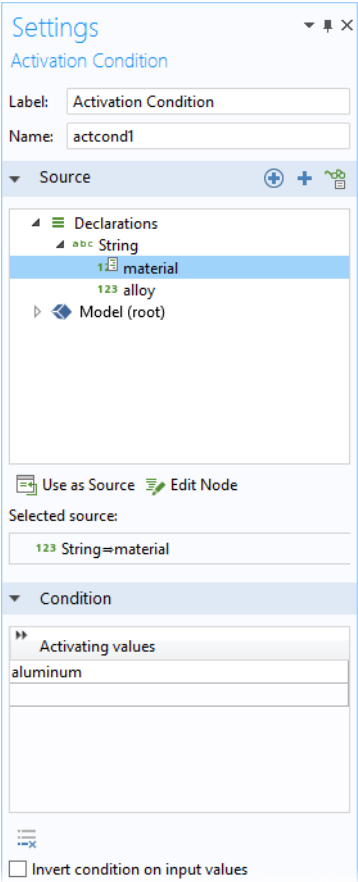
Label: Aluminum Alloys

Name: choicelist1

List Content

Value	Display name
mat1	Aluminum 3003
mat2	Aluminum 6063
mat3	Aluminum, generic

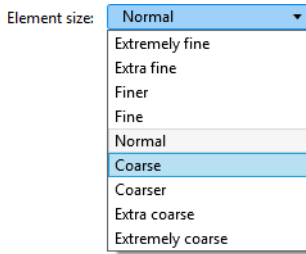
The activation condition for the **Aluminum Alloys** choice list is shown in the figure below.



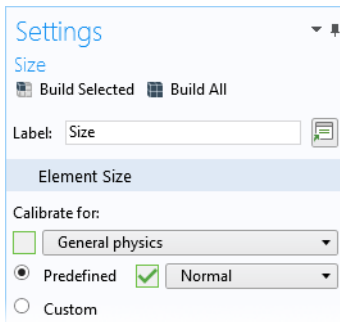
USING A COMBO BOX TO CHANGE ELEMENT SIZE

When creating a combo box, you can use the **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.



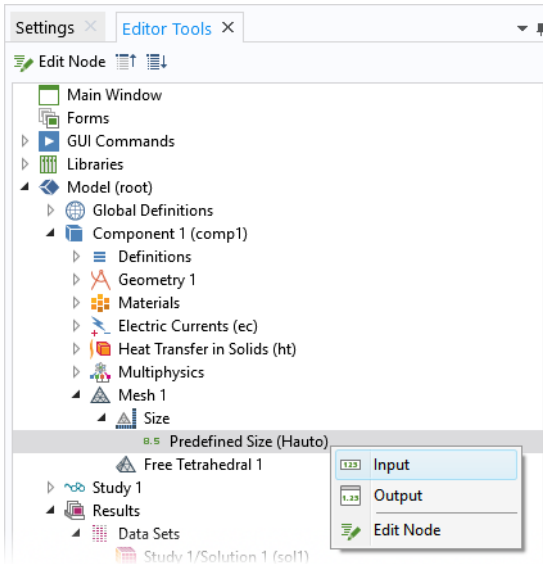
Switch to the Model Builder and select the **Mesh** node (we assume here that the model has just a single mesh). In the **Settings** window of the **Mesh** node, select **User-controlled mesh** (if not already selected). In the **Size** node, directly under the **Mesh** node, select the option **Predefined**.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



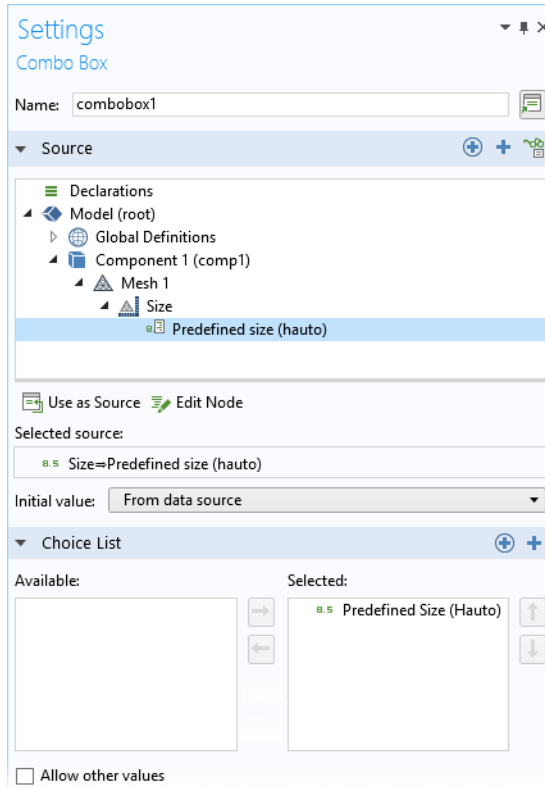
Select the green check box to the left of the list to make it available as a source for a combo box in the Application Builder. Then, when you return to the Application Builder, you will find that the choice list for mesh size is now revealed as a potential **Source** in the **Settings** for a new combo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

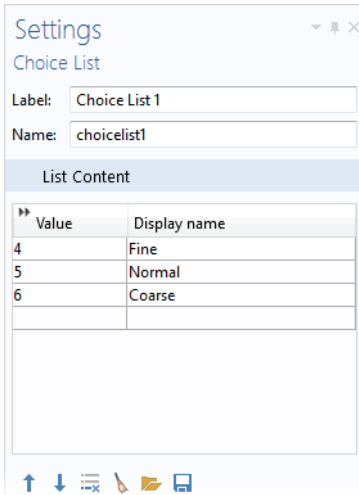


The corresponding **Settings** window for the combo box is shown in the figure below.



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 (haut0)**, 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 autogenerated 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();
```

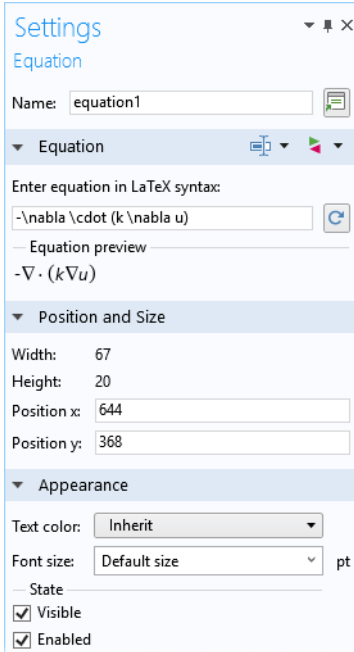
The `hauto` property can also take non-integer values. For more information on **Element size**, see “Data Access for Input Fields” on page 99.

USING A UNIT SET INSTEAD OF A CHOICE LIST

If the combo box will be used for the purpose of changing units, then a **Unit Set** can be used instead of a **Choice List** (you still select it in the **Choice List** section of the **Settings** window of the combo box).

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.

Settings

Line

Name: line1

Settings

Orientation: Horizontal

Include divider text

Text:

Position and Size

Width: 200

Height: 1

Position x: 622

Position y: 579

Appearance

Text color: Inherit

Font: Default font

Font size: Default size pt

Bold

Italic

Line thickness: 1

Line color: Default

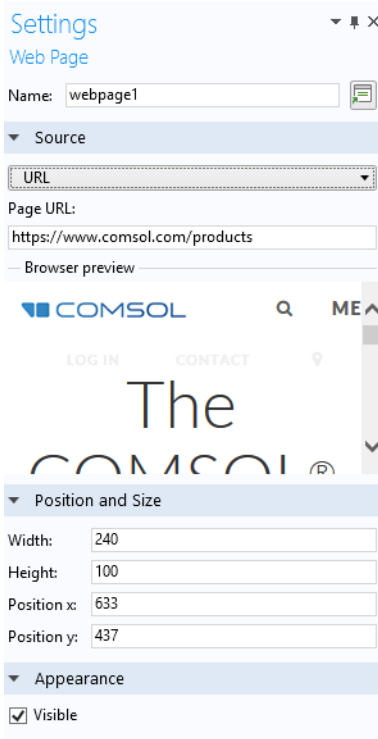
State

Visible

Enabled

Web Page

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

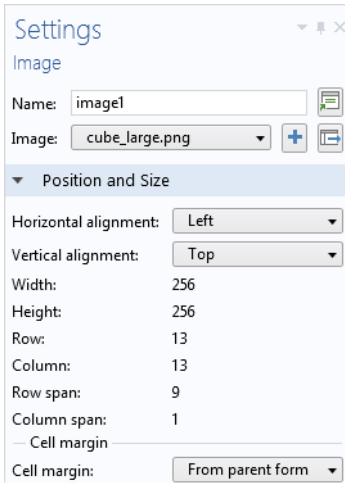


You can specify the page source in four different ways from the **Source** list:

- 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.

Video

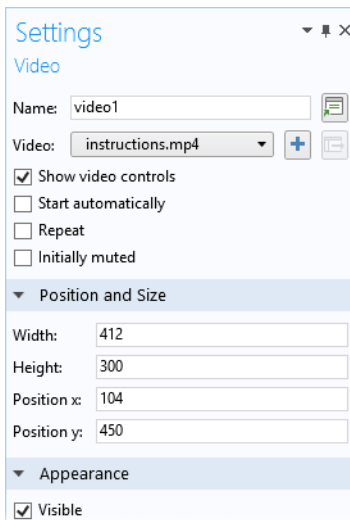
A **Video** object embeds a video file in a form. The supported video file formats are MP4 (`.mp4`), OGV (`.ogv`), and WebM (`.webm`). However, not all video file

formats are supported on all platforms. When running an application by connecting to COMSOL Server from a web browser, which formats are supported depend on the web browser and may vary with different versions of the same web browser. When running an application with the COMSOL Client and with COMSOL Multiphysics, the Internet Explorer version installed on your computer is used as a software component for displaying the video object.

After added to a form, the **Video** object is represented, in the Form editor by an image, as shown in the figure below.



The figure below shows the **Settings** window for the **Video** object.



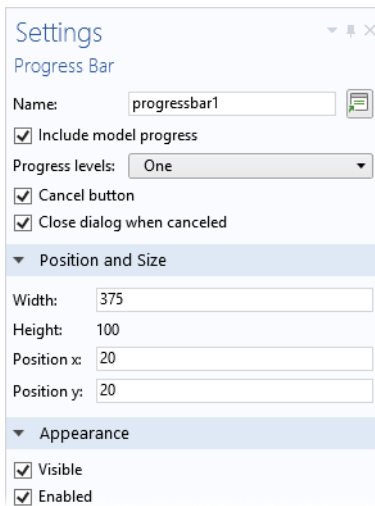
The available settings are:

- **Show video controls**
- **Start automatically**
- **Repeat**
- **Initially muted**

The option **Show video controls** enables the video controls such as Play and Stop. The option **Initially muted** is intended for the case where you want to play a video with the sound initially turned off. For example, if the video is set to start automatically, it can be useful to let the user choose whether the sound should be on. The user can enable the sound either from the video controls, if the check box **Show video controls** is selected, or by right-clicking in the video player.

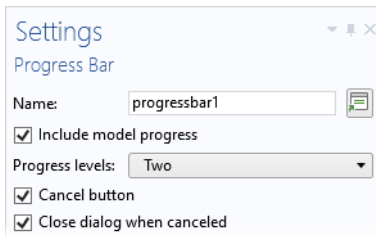
Progress Bar

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.



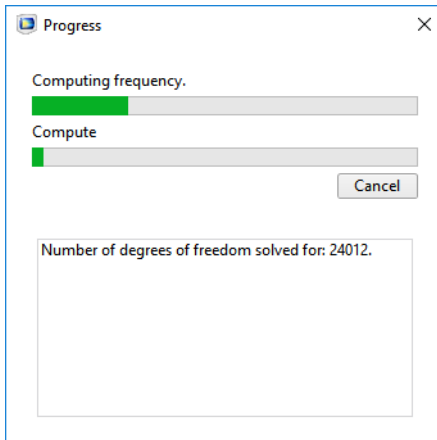
Note that the built-in progress bar that is visible in the status bar of an application is controlled by the **Settings** window of the **Main Window** node. By default, the built-in progress bar shows the progress of the built-in COMSOL Multiphysics core algorithms, such as geometry operations, meshing, 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 335 and the *Application Programming Guide*.

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

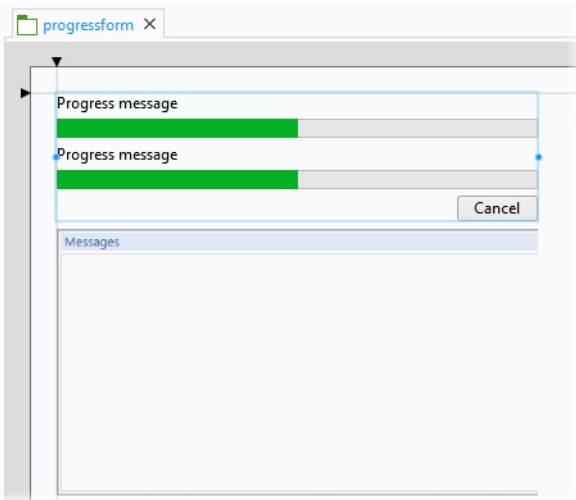


In this example, the progress bar object is part of a form `progressform` used to present a two-level progress bar and a message log.

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

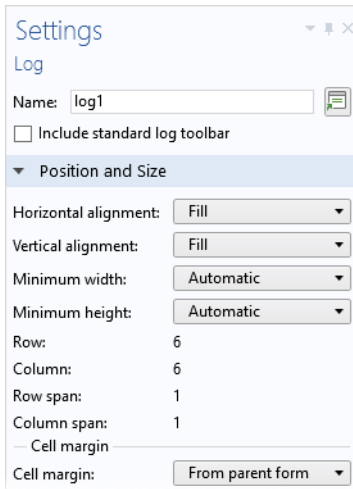
```
setProgressInterval("Computing frequency", lastProgress,
k*100/MAXITERATIONS).
```

For detailed information on the built-in methods and their syntax, see “Progress Methods” on page 335 and the *Application Programming Guide*.

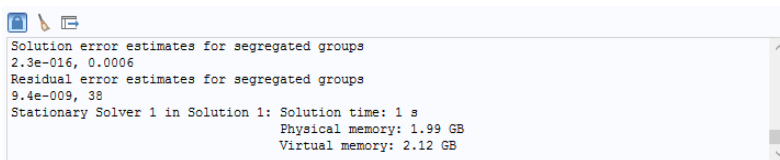
Log

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.

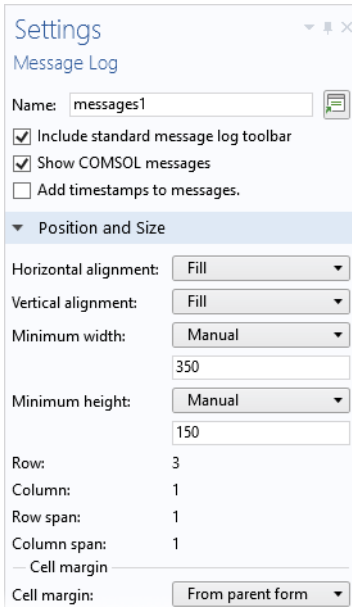


The figure below shows a part of an application user interface containing a log window.



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 330.

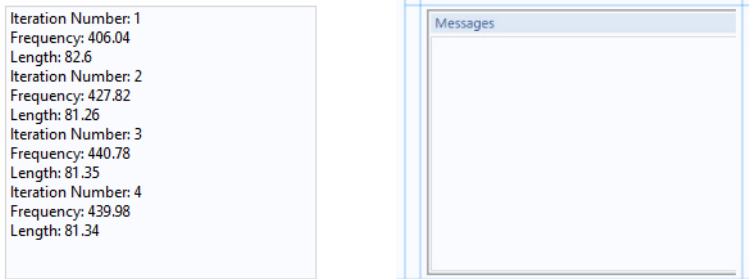


The screenshot shows the 'Settings' dialog for the 'Message Log' object. The dialog has a title bar with 'Settings', a close button, and a maximize button. Below the title bar, the 'Message Log' settings are displayed. The 'Name' field is set to 'messages1'. There are three checked checkboxes: 'Include standard message log toolbar', 'Show COMSOL messages', and 'Add timestamps to messages.'. Below these is a section titled 'Position and Size' with a dropdown arrow. Under this section, there are several settings: 'Horizontal alignment' is set to 'Fill', 'Vertical alignment' is set to 'Fill', 'Minimum width' is set to 'Manual' with a value of 350, and 'Minimum height' is set to 'Manual' with a value of 150. At the bottom, there are settings for 'Row' (3), 'Column' (1), 'Row span' (1), 'Column span' (1), 'Cell margin' (From parent form), and 'Cell margin' (From parent form).

You can also display the value of a variable, for example: `message(double xcoordinate)`.

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. You can include time stamps to message by selecting the check box **Add timestamps to messages**.

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



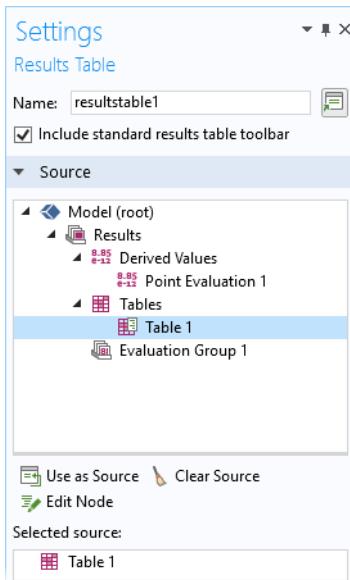
Results Table

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

The source of the results table data is taken from **Results** and can be a child node of **Derived Values**, a **Table**, or an **Evaluation Group**. 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:

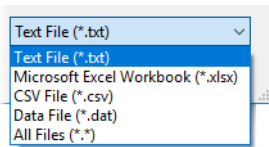
- **Full Precision**
- **Automatic Notation**
- **Scientific Notation**
- **Engineering Notation**
- **Decimal Notation**
- **Rectangular Complex Numbers**
- **Polar Complex Numbers**
- **Copy Table and Headers to Clipboard**
- **Export**

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)

This is shown in the figure below.

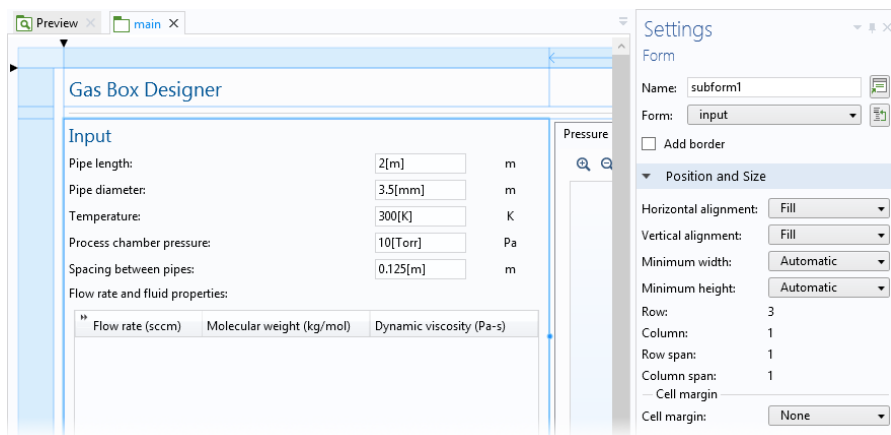


CONTROLLING RESULTS TABLES FROM METHODS

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 330.

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.

The screenshot shows a software window with three tabs: 'Preview', 'main', and 'input'. The 'input' tab is active, displaying a form titled 'Input'. The form is organized into a grid layout. The first row contains the numbers '230' and '172'. Below this, the form has several rows of input fields:

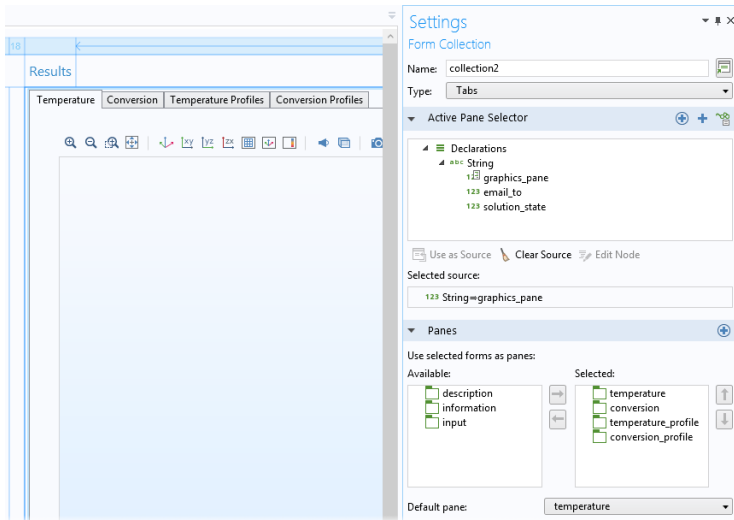
Input	
Pipe length:	2[m] m
Pipe diameter:	3.5[mm] m
Temperature:	300[K] K
Process chamber pressure:	10[Torr] Pa
Spacing between pipes:	0.125[m] m
Flow rate and fluid properties:	
» Flow rate (sccm)	Molecular weight (kg/mol)
Dynamic viscosity (Pa-s)	

At the bottom of the window, there is a toolbar with various icons for file operations and editing.

If you are using grid layout mode, then you can quickly create subforms using the **Extract Subform** button in the ribbon. See “Extracting Subforms” on page 117.

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 **Panels** 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**.

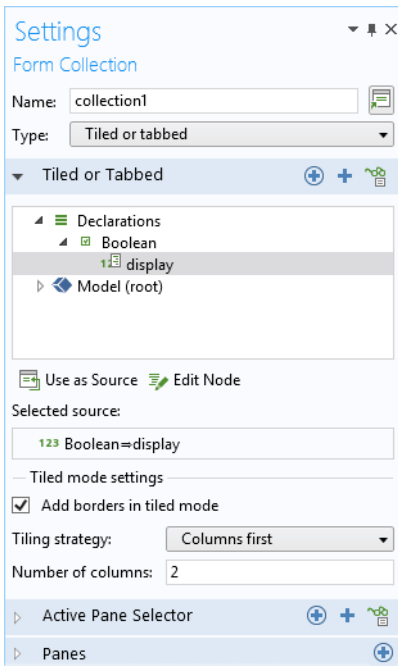
The string variable needs to be equal to one of the form names in the form collection, such as `temperature` or `conversion` in the example above. Otherwise, it will be ignored.

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, as shown in the example below.

```
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

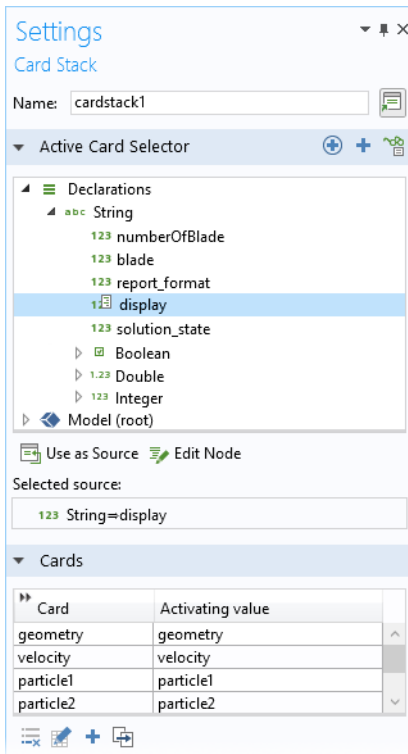
Consider an application where the graphics shown to the user depend on the value of a scalar variable. This variable may change when a user clicks, for example, a radio button. The variable may also change depending on a computed value; for example, the value of a **Global Evaluation** node in the model 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**.

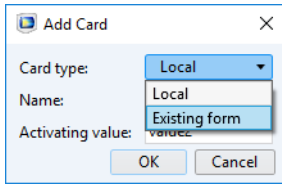


By clicking a row in the table of cards in the **Cards** section, followed by clicking one of the toolbar buttons below the table, you can perform the following operations on cards:

- **Delete**
- **Edit**
- **Add Card**
- **Duplicate**

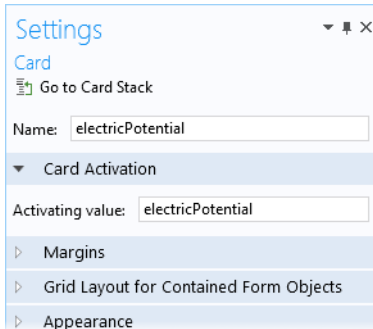
Each row in the table contains the name of the card in the **Card** column and their associated activating values in the **Activating value** column. The stack decides which cards to display based on their activating values. In this example, the activating values are the strings `geometry`, `velocity`, `particle1`, etc.

Clicking the **Add Card** button displays the following dialog box.

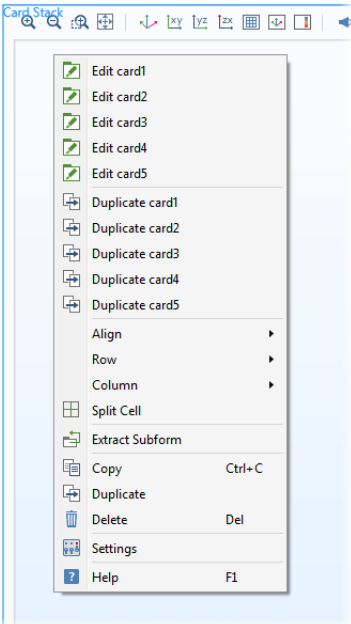


By default, the **Card type** is set to **Local**, which means that the card is defined locally in its containing card stack object. If the **Card type** is set to **Existing form**, then you can instead select one of the existing forms. The settings for an **Existing form** are accessed directly from the Form editor by clicking its node or by clicking the **Edit** button in the **Card** section of the corresponding card stack **Settings** window.

The figure below shows the **Settings** window of a **Card** as shown after clicking **Edit** in the table in the section **Cards**.

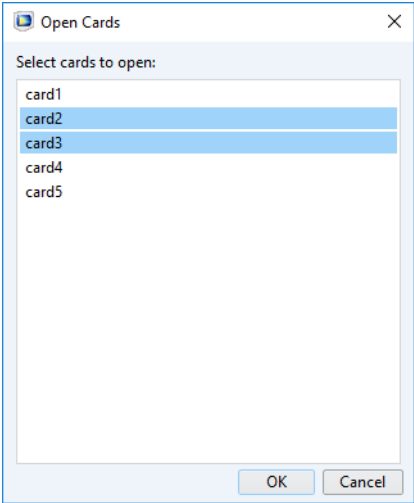


To access locally defined cards, right-click the card stack in a form window to select between the different cards in the card stack, as shown in the figure below.

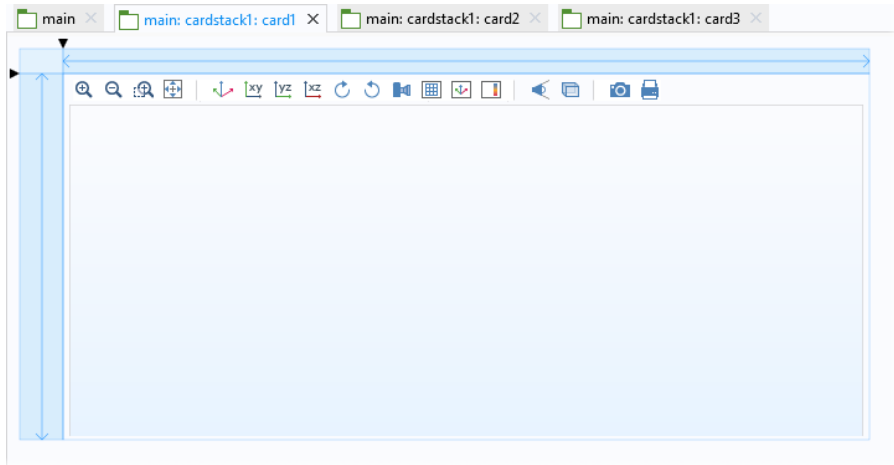


From this menu, you can also duplicate cards.

To edit cards, you can also use Alt+Click, which opens a dialog box that lets you select multiple cards at once.



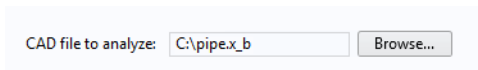
The figure below shows card1 with its graphics form object.



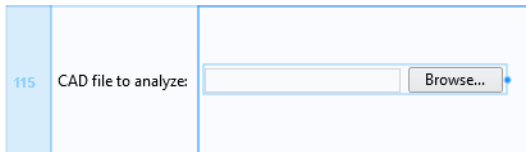
File Import

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.

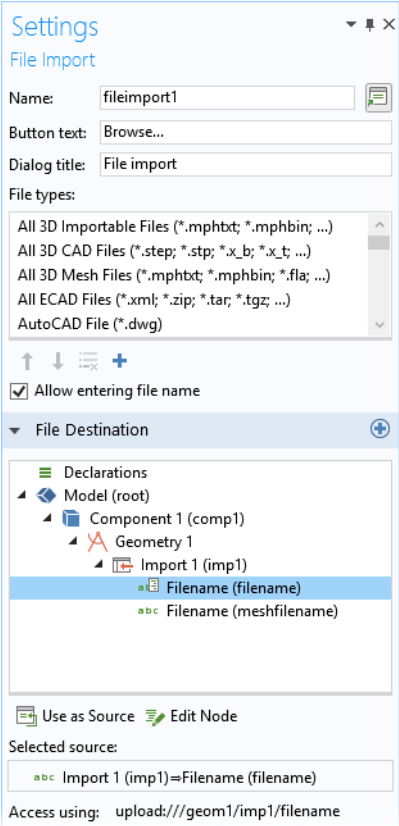
Consider an application where a CAD file can be selected and imported at run time, as shown in the figure below.



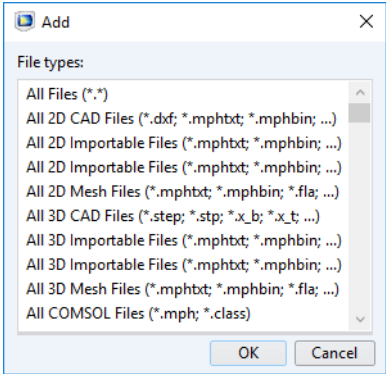
The corresponding **File Import** object is shown in the figure below.



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 as input. This is shown in the figure below, where the **Filename** for a geometry **Import** node is selected.

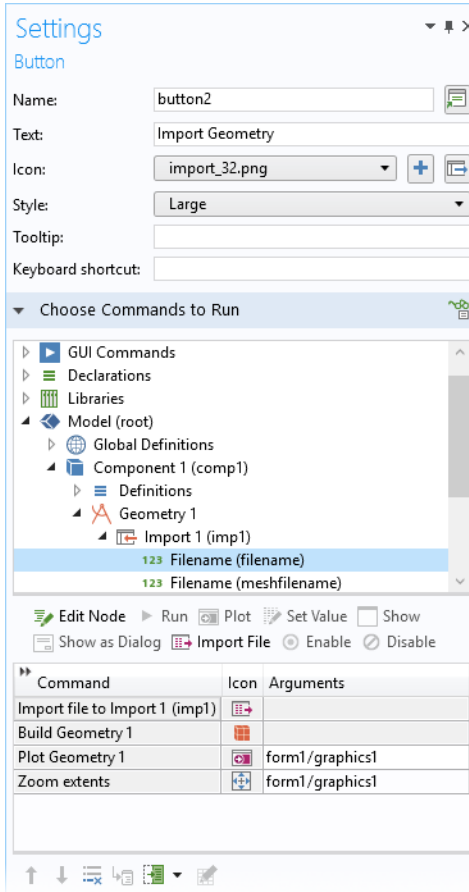


In this application, the **File types** table specifies that only CAD files are allowed. You can further control which **File types** are allowed by clicking the **Add** and **Delete** buttons below the list of **File types**. Clicking the **Add** button displays the dialog box shown to the right:



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 that allow a user to pick a file in a file browser. For example, you can use a menu, ribbon, toolbar item, or a button to open a file browser. 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” on page 153. For more information on file handling in general, see “Appendix C — File Handling and File Scheme Syntax” on page 300.

The built-in method that corresponds to the command **Import file** is `importFile`. For example, for importing an image you can use:

```
success=importFile("file1",new  
String[]{"ALL_IMPORTABLE_IMAGES", "PNG", "JPEG", "BMP", "GIF"});
```

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.

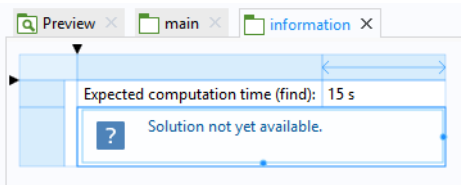
Information

Expected computation time (find): 15 s

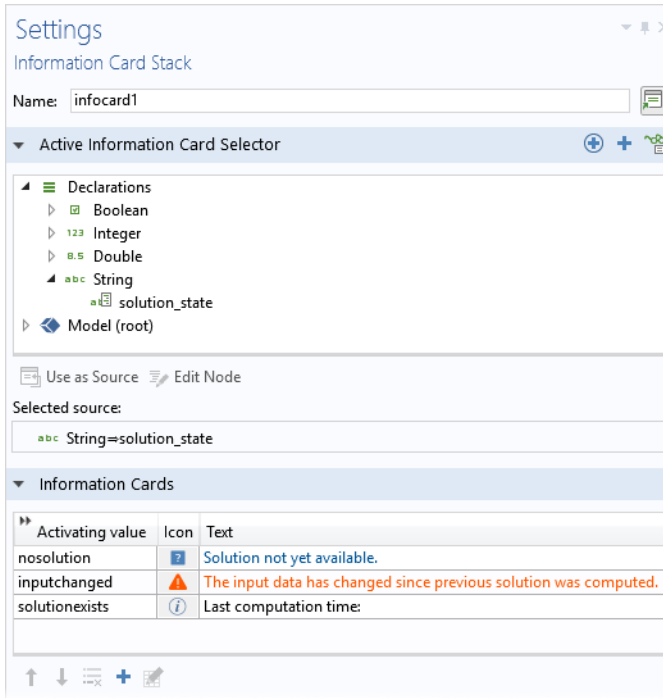


Last computation time: 13 s

The corresponding form objects are shown below:

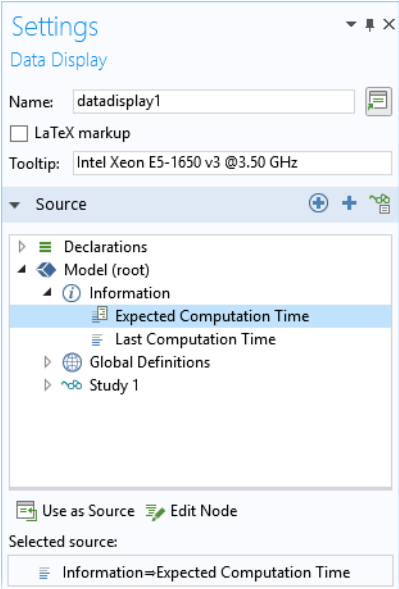


The figure below shows the **Settings** window where a string variable `solution_state` is used as the source.



There are similarities with a **Card Stack** object, but for the **Information Cards**, each card has an icon and 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.

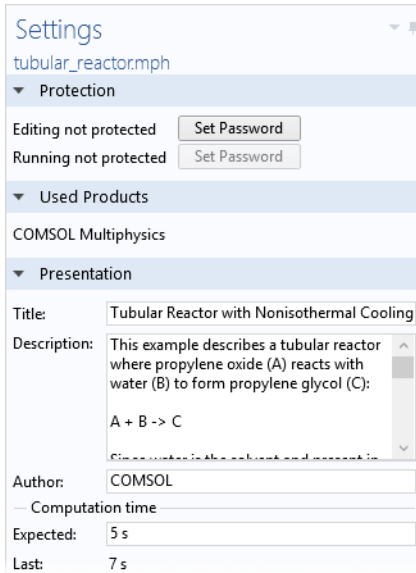


Note that information nodes in the model tree are only shown when working with the Application Builder. They are made available in the **Source** section in the **Settings** window for form objects, when applicable.

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.



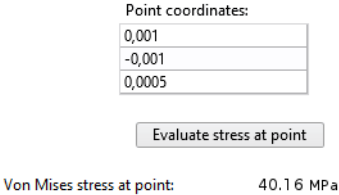
If the computation time is predominantly spent in a method, such as when the same study is called repeatedly, then you can manually measure the computation time by using the built-in methods `timeStamp` and `setLastComputationTime`. For more information, see “Date and Time Methods” on page 335.

Array Input

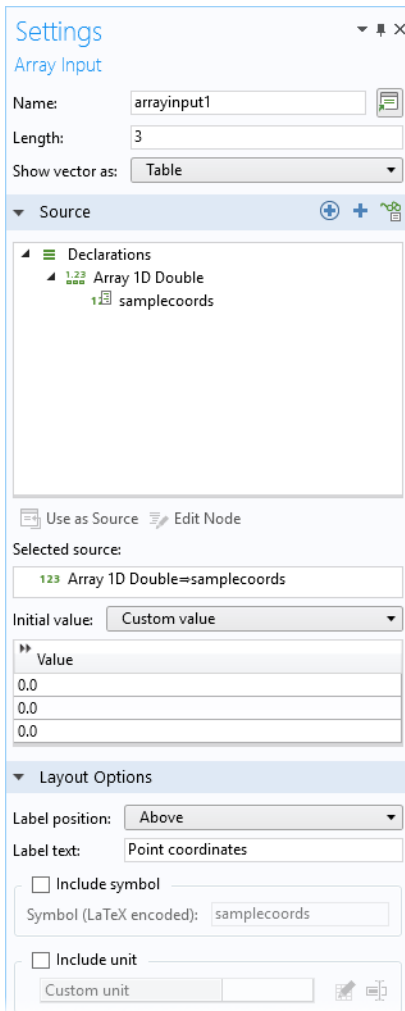
An **Array Input** object has an input table used to enter array or vector-valued input data. An array input object supports string arrays as data sources. You can add an optional label, symbol, and unit.

USING AN ARRAY INPUT OBJECT FOR 3D POINT COORDINATE INPUT

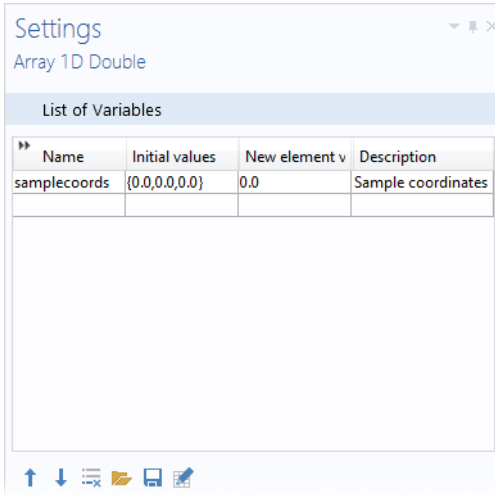
Consider an application where the user enters 3D coordinates for a point where the stress is evaluated. The figure below shows a screenshot from an application with an array input, button, text label, and data display object.



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 **1D 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();
```

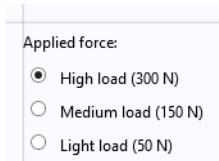
where the values `pointx`, `pointy`, and `pointz` will be used subsequently as coordinates in the evaluation of the stress.

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

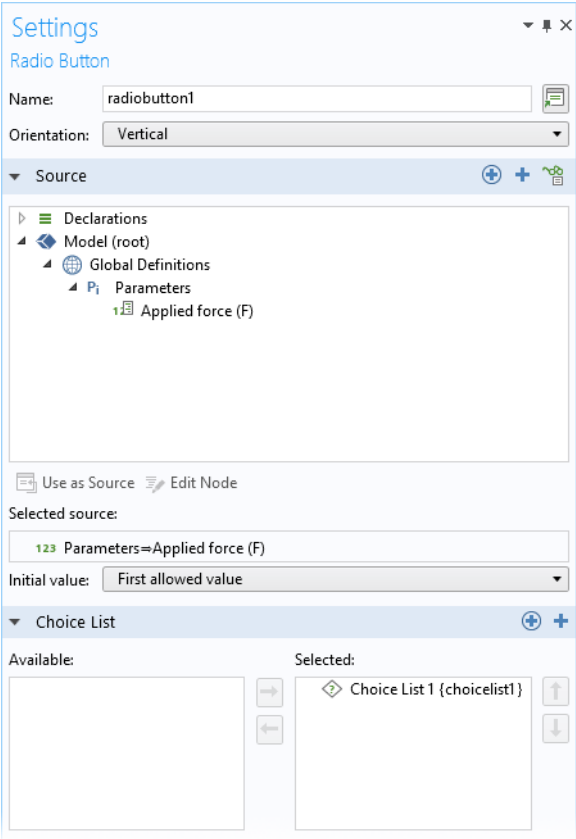
Consider an application where the user can select one of three predefined loads, as shown in the following figure.



Applied force:

- High load (300 N)
- Medium load (150 N)
- Light load (50 N)

The corresponding **Settings** window is shown below, where the global parameter F is used as the source.

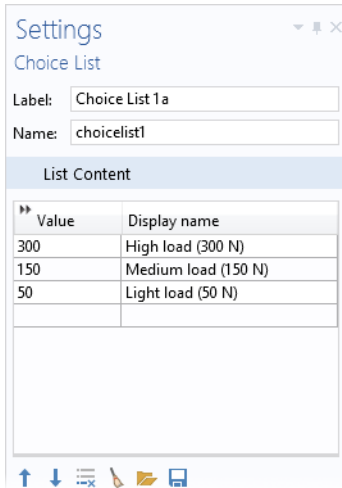


The **Orientation** can be set to **Vertical** (default) or **Horizontal**.

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.



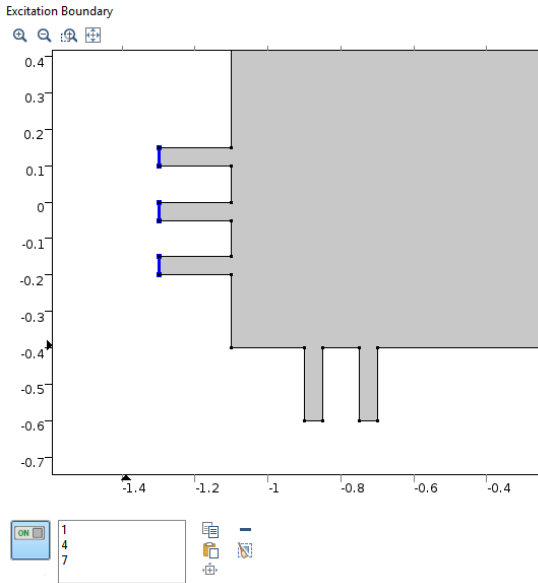
USING A UNIT SET INSTEAD OF A CHOICE LIST

If the radio button will be used for the purpose of changing units, then a **Unit Set** can be used instead of a **Choice List** (You still select it in the **Choice List** section of the **Settings** window of the radio button object).

Selection Input

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. For more information on selections, see “Selections” on page 83. In the example below, the embedded model has a boundary condition defined

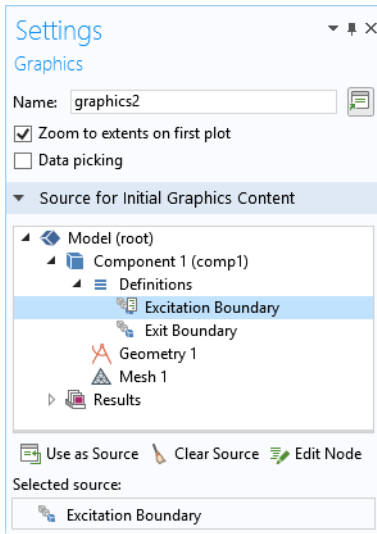
with an **Explicit** selection. Both a **Selection Input** object and a **Graphics** object are used to let the user select boundaries to be excited by an incoming wave.



The user can select boundaries here 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.

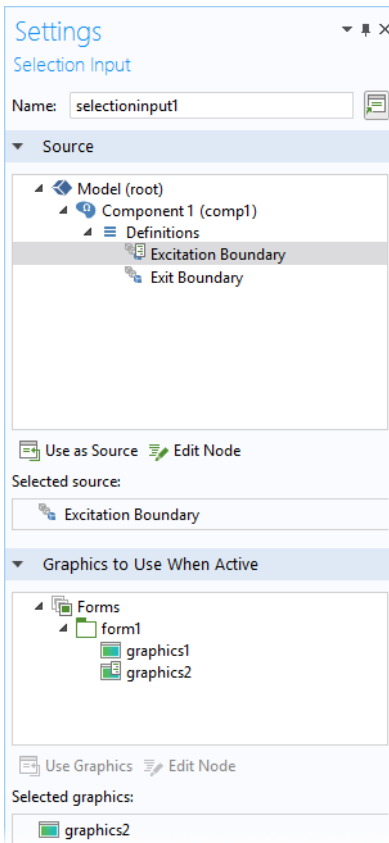
To make it possible to directly select a boundary by clicking on it, you can link a graphics object to an explicit **Selection** used to group boundaries, as shown in the figure below. Select the explicit selection and click **Use as Source**.

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.

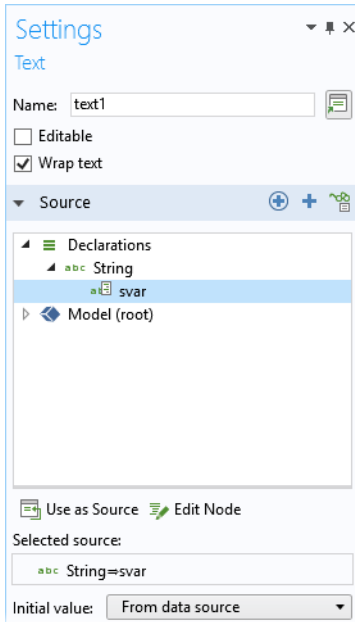


In a selection input object, you can copy, paste, remove, clear, and zoom into selections.

- ⚠ 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 or an **Information** node. The **Settings** window for a text object is shown below.



Select a string variable or **Information** node 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.

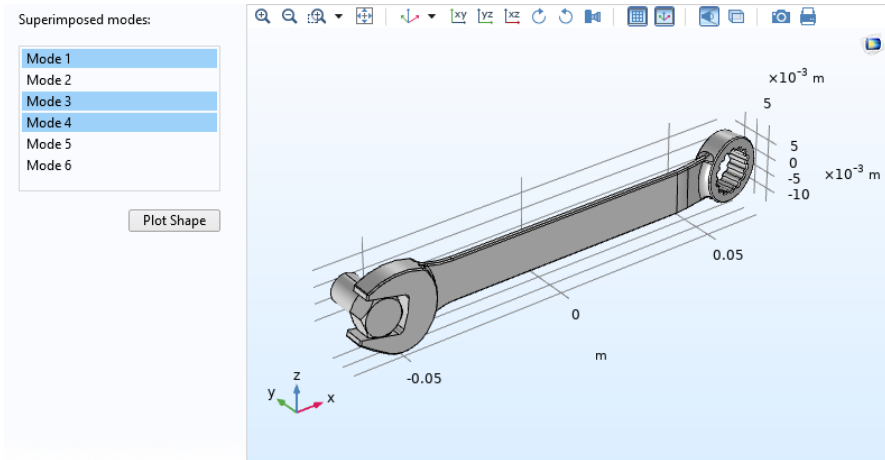
For more information on **Information** nodes, see “Data Display” on page 96.

List Box

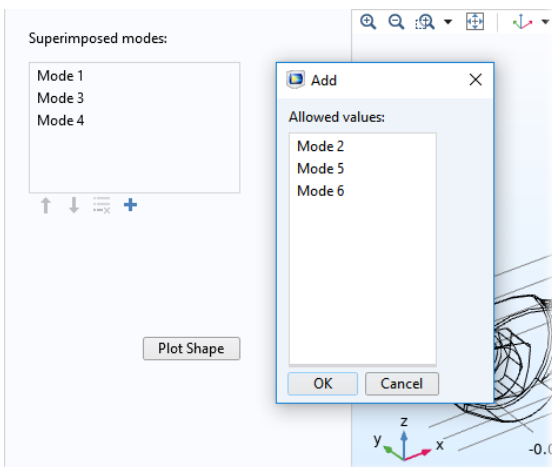
A **List Box** object is similar to a radio button object, except that it allows for the simultaneous selection of multiple options.

USING A LIST BOX TO SUPERIMPOSE VIBRATIONAL MODES

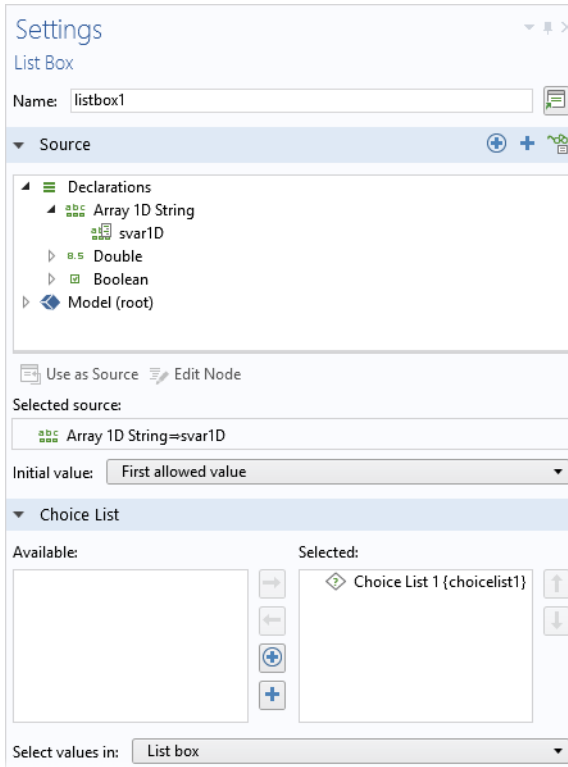
Consider an application where the first six vibrational modes of a mechanical part can be superimposed and visualized by selecting them from a list box, as shown in the figure below.



As an alternative, the following figure shows that a list can be displayed as a dialog box.



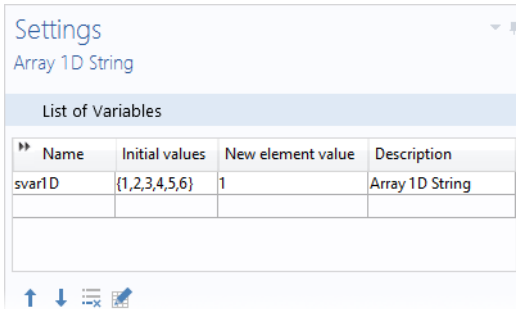
The **Settings** window for the list box of this example is shown in the figure below.



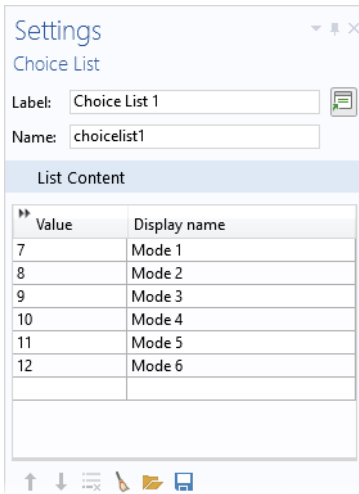
The **Select values in** list allows you to choose between two alternatives, **List box** or **Dialog**, for displaying the list.

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 nonrigid 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 $\text{with}(1,u)+\text{with}(3,u)+\text{with}(5,u)$. 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 the results 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 339.

USING A UNIT SET INSTEAD OF A CHOICE LIST

If the list box will be used for the purpose of changing units, then a **Unit Set** can be used instead of a **Choice List** (You still select it in the **Choice List** section of the **Settings** window of the list box).

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 two windows from a software application. The left window, titled 'input', shows a form with several input fields and a table. The table, labeled 'Flow rate and fluid properties', has three columns: 'Flow rate (sccm)', 'Molecular weight (kg/mol)', and 'Dynamic viscosity (Pa-s)'. The right window, titled 'Settings', shows the configuration for a 'Table' object. It includes a 'Name' field set to 'b', checkboxes for 'Show headers', 'Automatically add new rows', and 'Sortable'. Under 'Sources', there are three declarations: 'Array ID String', 'flow_rate', 'molecular_weight', and 'dynamic_viscosity'. Below this is a table for column settings:

Header	Width	Grow	Editable	Alignment	Data source
Flow rate (sccm)	120	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Left	Data 'flow_rate' from 'Array ID String'
Molecular weight (kg/mol)	160	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Left	Data 'molecular_weight' from 'Array ID String'
Dynamic viscosity (Pa-s)	160	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Left	Data 'dynamic_viscosity' from 'Array ID String'

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 ID 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**
- **Grow**
- **Editable**
- **Alignment**
- **Data source**

Each row in this table defines a column in the table object. The option **Grow** allows individual columns to grow when a form is resized. This option is only applicable to grid mode and if the **Horizontal alignment** of the table is set to **Fill**.

In the example, the string arrays define the initial values for the rows corresponding to the three columns, as shown in the figure below:

Settings
Array 1D String

List of Variables

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 items (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**

The **Icon size** setting allows you to choose **Small** or **Large** icons.

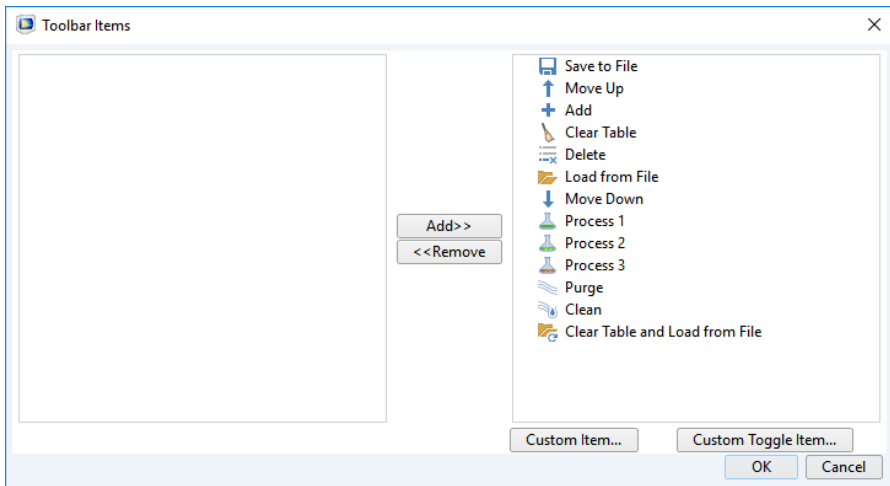
To add an item to the toolbar, click the **Add Toolbar Item** button below the table.

item5		Delete
item6	📁	Load from file
item7	↓	Move down
process1	🧪	
process2	🧪	

↑ ↓ ↻ 📁 📄 🖨 + 📄

Add Toolbar Item

The following dialog box is then shown.

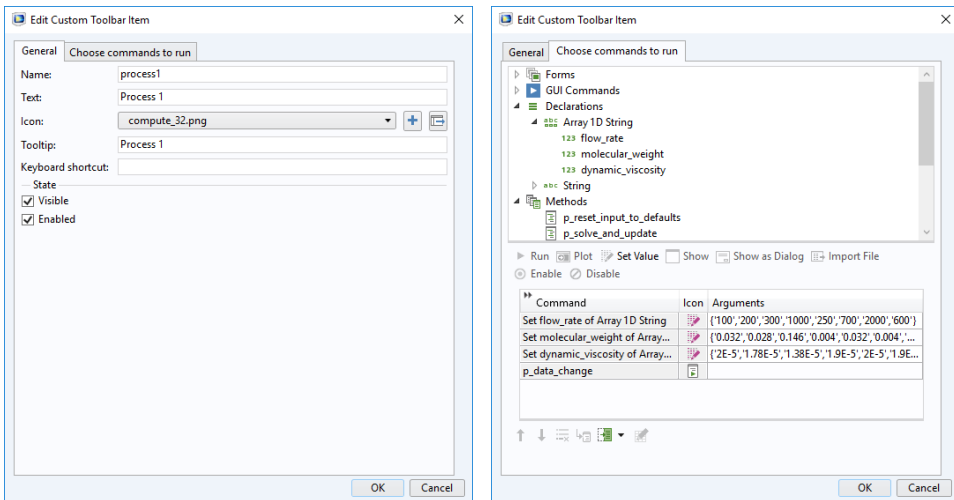


You can add the following items:

- **Save to File**
- **Move Up**
- **Move Down**
- **Add**
- **Delete**
- **Clear Table**
- **Clear Table and Load from File**
- **Load from File**

In addition, you can add customized items by clicking **Custom Item** or **Custom Toggle Item** in the **Toolbar Items** dialog box. The figure below shows the **Edit Custom Toolbar Item** dialog box used to define a customized button. The dialog

box has two tabs for a regular item and three tabs for a toggle item. In this case, the button **Process 1** is used to set default values for a certain process.

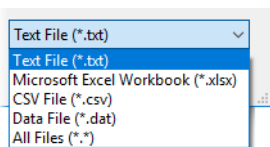


The **Choose commands to run** tab is similar to that of menu, ribbon, and toolbar items, as well as buttons.

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)

This is shown in the figure below.



The allowed separators are comma, semicolon and tab for CSV files, and space and tab for DAT and TXT files.

Slider

A **Slider** is a form object for choosing numerical input using a slider control.

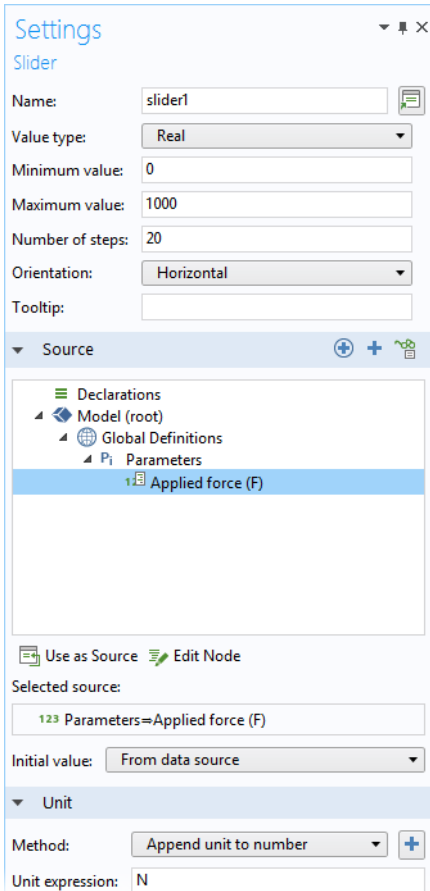
USING A SLIDER TO CHANGE THE MAGNITUDE OF A STRUCTURAL LOAD

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 application tree and click **Use as Source**.

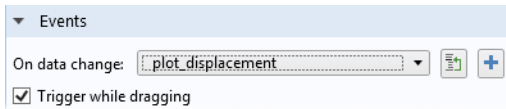
From the **Value type** list, choose **Integer** or **Real** (default), depending on the type of data in the data source for the slider.

You determine the range of values for the data source by defining the **Maximum value**, **Minimum value**, and **Number of steps** for the slider. The **Orientation** can be **Horizontal** or **Vertical**. 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. As an alternative to **Append unit to number**, you can choose **Append unit from unit set**. See “Unit Set” on page 154 for more information.

In the **Initial value** list, select **From data source** or **Custom value** for the initial value for the slider.

In the **Events** section, in addition to specifying which method to call for an **On data change** event, you can select the check box **Trigger while dragging**. This setting determines if the event method should be called continuously while the slider is being dragged or only upon its release.



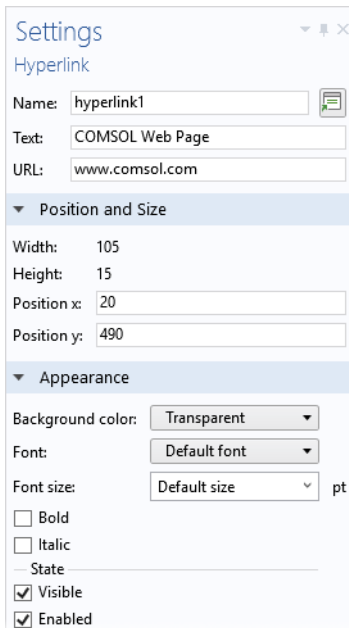
This setting can be useful if the method that is called by the **On data change** event is computationally heavy, so that there is a lag when dragging the slider.

Hyperlink

A **Hyperlink** object embeds a hyperlink in a form. The figure below shows an example of a hyperlink.



The figure below show the corresponding **Settings** window.



The **Hyperlink** object supports the types of URLs that you can use in a web browser, including:

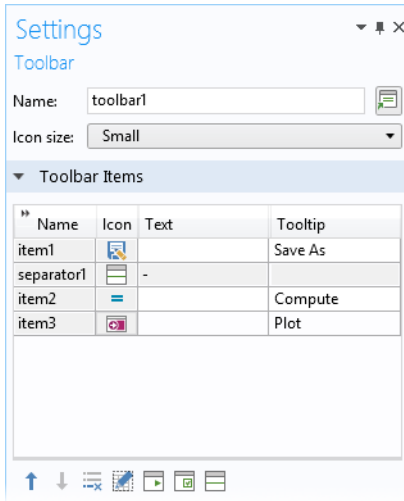
- **Web Page:** When a user clicks the hyperlink for a web page, it opens in the user's default browser. The URL string needs to be on the form `protocol://address`, where protocol is the transmission protocol; for example, HTTP or HTTPS. The web address can be partial or complete, but it is recommended to use a complete web address.
- **Email:** An email address is specified on the form `mailto:emailaddress`. This will launch the user's default email application program and prepare a new message where the To field is set to the address specified. This way of interactively sending an email from a COMSOL application is different from using the built-in method. For more information on the built-in methods for email, see “Email Methods” on page 327.

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**.

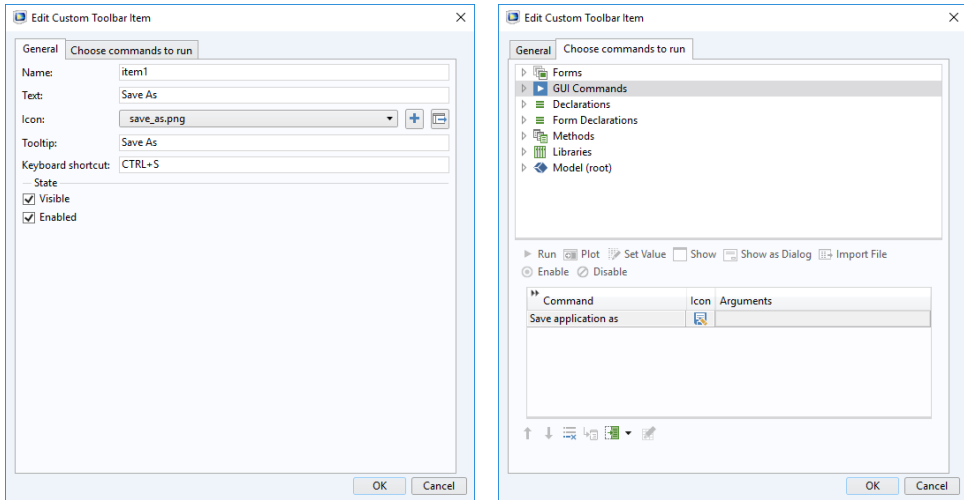


The **Settings** window for this toolbar is shown in the figure below.



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 the row order, or delete a row. Click the **Edit** button to display

the **Settings** window associated with each row. The figure below shows the **Settings** window of **item 1**, the **Save As** item.

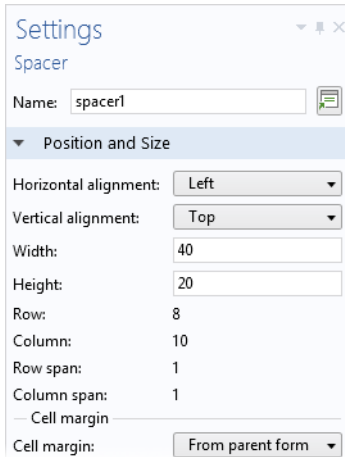


The text in the **Tooltip** field will be shown as a tooltip when hovering over the toolbar button. The text in the **Text** field will be shown next to the icon, if any; otherwise just the text is shown. The **Icon** list, the **Keyboard shortcut** field, and the **Choose commands to run** tree represent the same functionality as a button object. For more information, see “Button” on page 59.

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.



The image shows a 'Settings' dialog box for a 'Spacer' object. The window title is 'Settings' and the subtitle is 'Spacer'. The 'Name' field contains 'spacer1'. The 'Position and Size' section is expanded and contains the following settings:

Horizontal alignment:	Left
Vertical alignment:	Top
Width:	40
Height:	20
Row:	8
Column:	10
Row span:	1
Column span:	1
Cell margin:	From parent form

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 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 run once it is deployed. For more in-depth information on reading and writing various types of data to file, see the *Application Programming Guide*.

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).

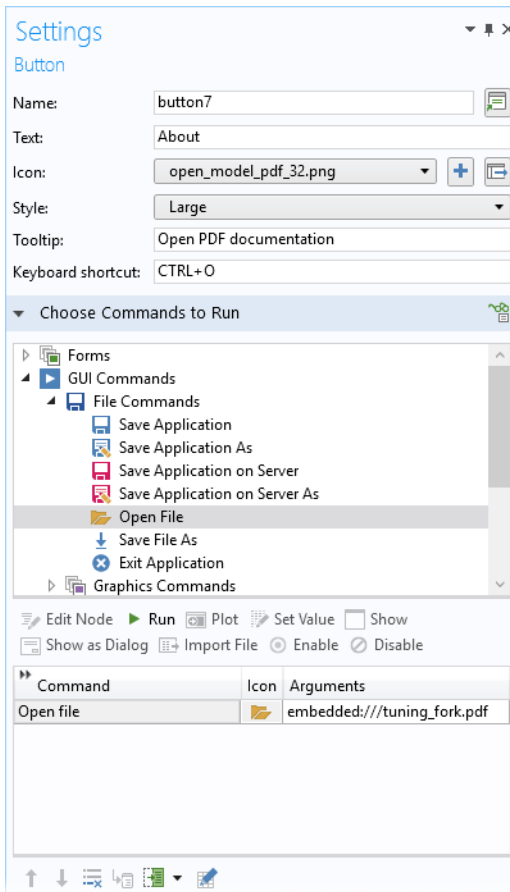
However, there are techniques for transferring files to and from the client file system when running an application both with a web browser and the COMSOL Client.

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 305.

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 312.

SAVING AND OPENING FILES USING FILE COMMANDS

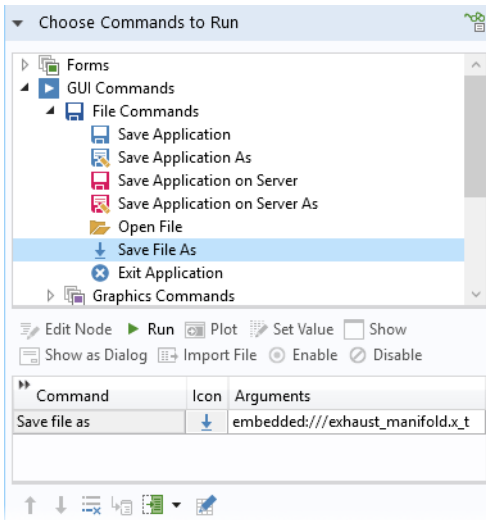
In the editor tree used in a command sequence, the **File Commands** folder contains commands to save and load applications and files, as well as exiting an application.



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 above, an **Open File** command is used to open the PDF documentation for an application. The corresponding PDF file is embedded in the application by being stored in the **Libraries > Files** node. Files located there are referenced using the `embedded:///` file scheme syntax described in the next section, “File Scheme Syntax” on page 303.

To open files from a method, use the built-in method `fileOpen`; see also “Operating System Methods” on page 327.

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. In the figure below, a **Save File As** command is used to save a CAD model that is stored in the **Libraries > Files** node.



To save files from a method, use the built-in method `fileSaveAs`; See also “GUI Command Methods” on page 333. For more information on saving and exporting files, see “File Export” on page 312.

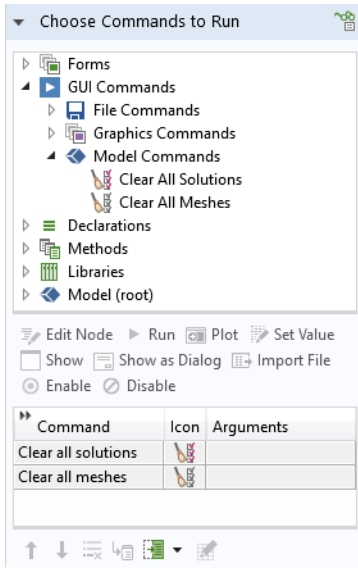
The **Save Application** and **Save Application As** commands are available for use in the command sequence for certain form objects. The **Save Application As** command will display a **Save As** dialog box where the user can specify a client path where the entire application will be saved.

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 “GUI Command Methods” on page 333.

In summary, both uploading and downloading files from the client file system is supported, but, due to web browser and system security settings, 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 commands 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 (this is set in the **Preferences** dialog box, see below.)

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.
- 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
embedded:///	Files embedded in the application using Libraries > Files	N/A	Experimental data, CAD files, mesh files, interpolation data
upload:///	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
temp:///	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 data export to a file saved on the client (for use with COMSOL Server)
user:///	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
common:///	Files in a directory shared by all users	Determined by the settings in Preferences > Files	Files shared between many users or applications

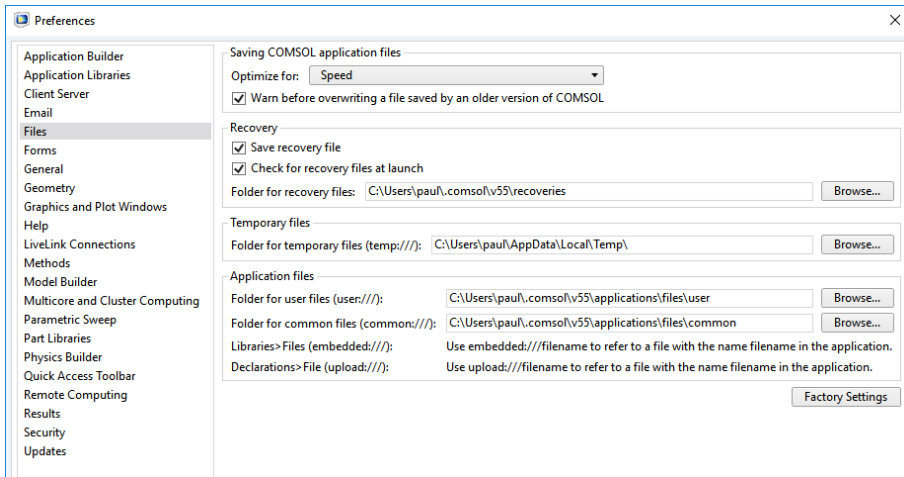
For more information on files in the Libraries node accessible by the **embedded:///** syntax, see “Libraries” on page 211.

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)	√	√	√	√

USAGE	EMBEDDED	UPLOAD	TEMP	USER	COMMON
Method writing a file			√ (r)	√	
File is client-side	√	√	√ (r)	√	√

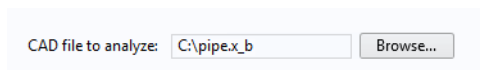
You can set the preferences for the paths to temporary, user, and common files in the **Files** page of the **Preferences** dialog box, which is accessible from the **File** menu, as shown in the figure below.



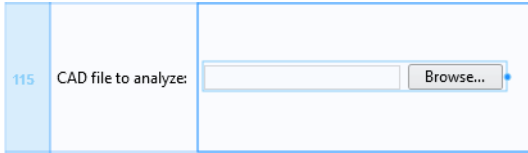
File Import

CAD IMPORT USING THE MODEL TREE AND A FILE IMPORT OBJECT

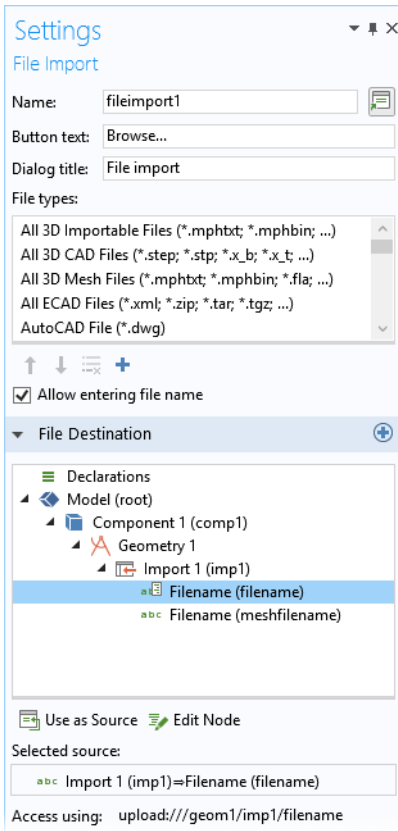
A **File Import** object is used to display a file browser with an associated input field for browsing to a file or entering its path and name. It is used to enable file import by the user of an application at run time, when the file is not available in the application beforehand. You can directly link 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.

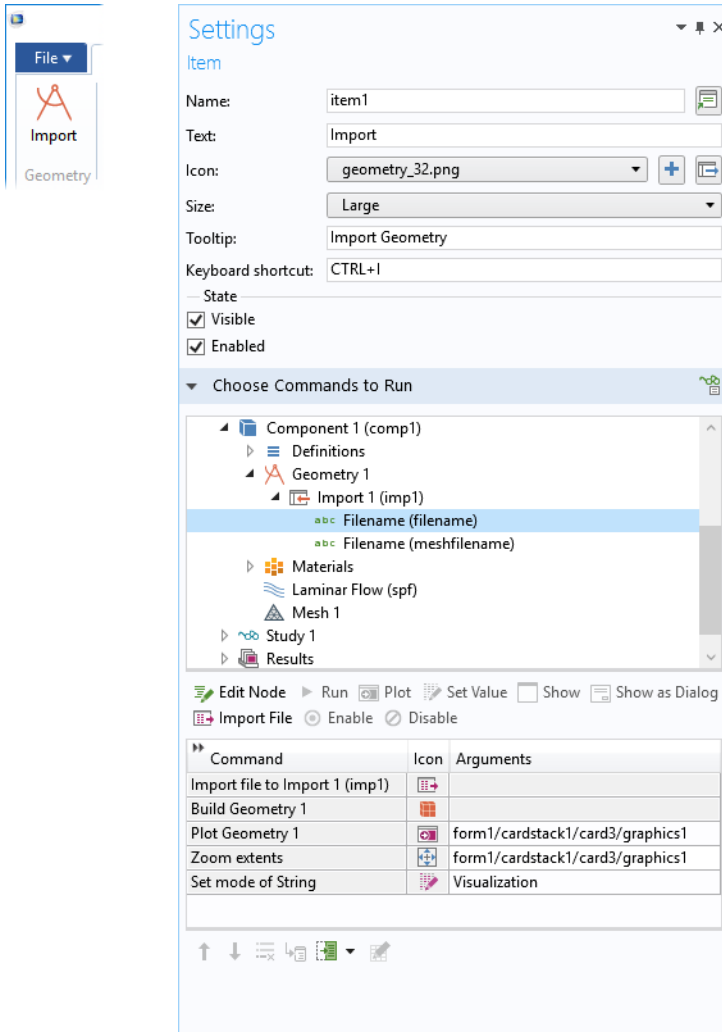


If you do not wish to use a **File Import** object, you can directly reference a **Filename** from a button or an item in a menu, ribbon, or toolbar, or alternatively create a method that calls the built-in method `importFile` as an event, for example

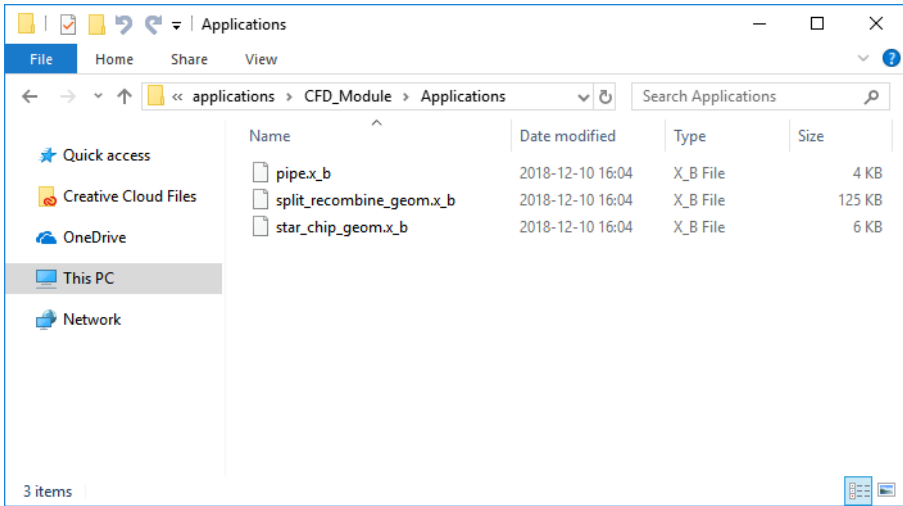
```
importFile("file1");
```

assuming there is a file declaration `file1`.

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 265.

FILE IMPORT IN METHODS

Continuing the example of the previous section, assume that we click **Convert to New Method** in the **Settings** window. 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 325 and the *Application Programming Guide*.

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**

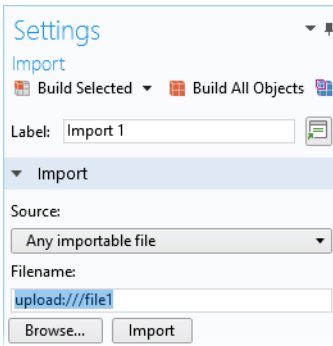
using). The figure below shows an example where a **File Destination** and **Filename** are used.



The file scheme syntax, `upload:///geom1/imp1/filename`, 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” on page 153.

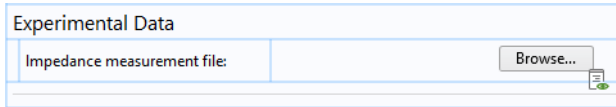
This syntax can also be used in any file browser fields within the Model Builder nodes. The figure below shows a file reference used in the **Filename** field of the **Import** model tree node for a model using geometry import.



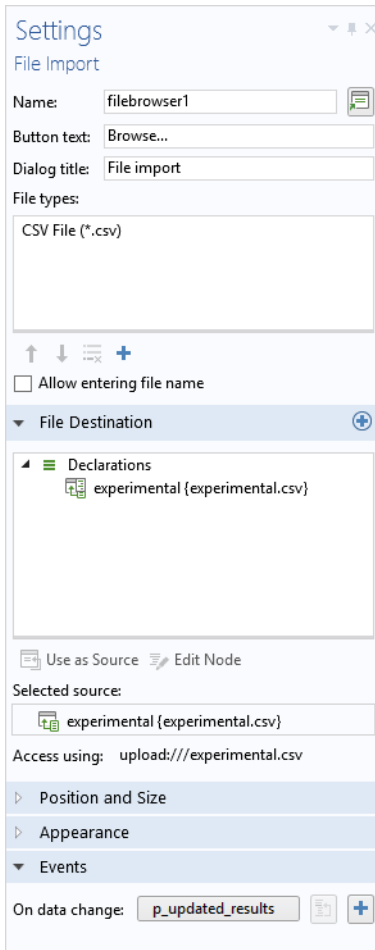
However, a quicker way is to link a file import object directly to the **Filename** field, as described previously in the section “CAD Import using the Model Tree and a File Import Object” on page 305.

IMPORTING EXPERIMENTAL DATA

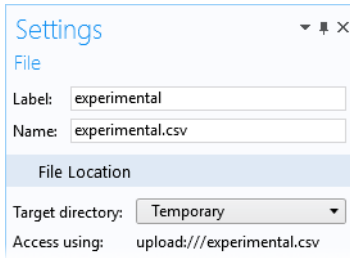
Consider an application where the user is providing a file with experimental data at run time. The figure below shows the file import object of such an application as it appears in grid layout mode.



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.

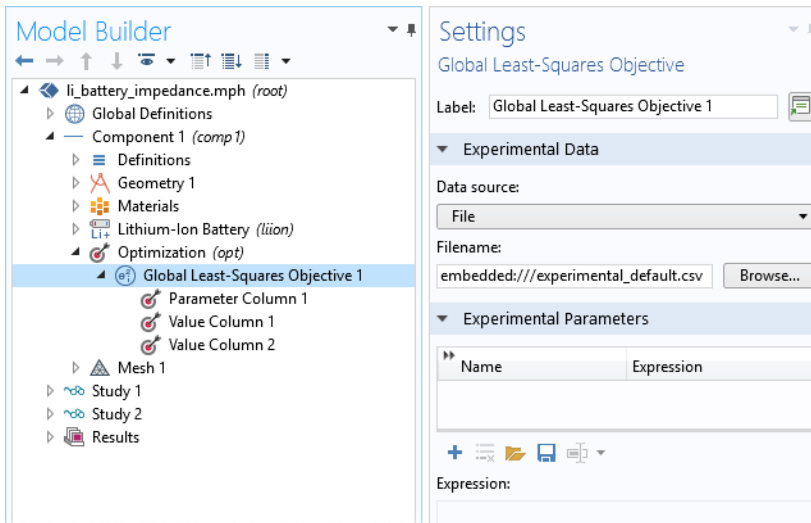


The file declaration serves as the “destination” of the imported data, which is written to the file `upload:///experimental.csv`.

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;
  }
}
```

If a user-provided file exists, the code replaces embedded:///experimental_default.csv with upload:///experimental.csv in the physics interface glsobj1.

More information on file import can be found in the *Application Programming Guide*.

File Export

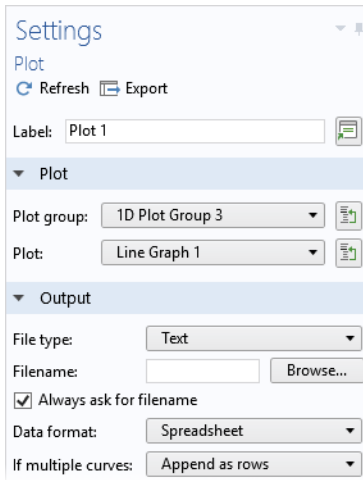
FILE EXPORT USING THE MODEL TREE

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.

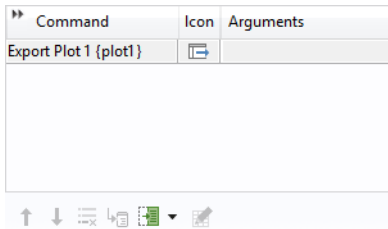
In the model tree, the **Export** node may contain the following types of subnodes for file export:

- **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.

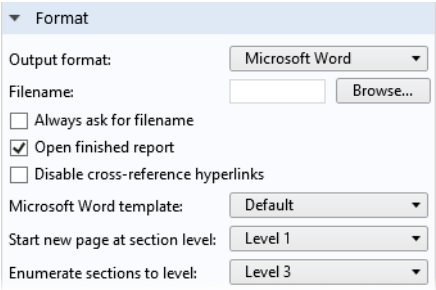


You can leave the **Filename** field blank, as shown in the figure above. In the command sequence of, for example, a button, you can run the corresponding **Export > Plot** node and, at run time, it will open a file browser window for the user to select a location and file name, as seen in the figure 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 **Format** section with a **Filename** field, as seen in the figure below.



The image shows a 'Format' section in a software interface. It contains the following elements:

- Output format:** A dropdown menu set to 'Microsoft Word'.
- Filename:** A text input field followed by a 'Browse...' button.
- Always ask for filename
- Open finished report
- Disable cross-reference hyperlinks
- Microsoft Word template:** A dropdown menu set to 'Default'.
- Start new page at section level:** A dropdown menu set to 'Level 1'.
- Enumerate sections to level:** A dropdown menu set to 'Level 3'.

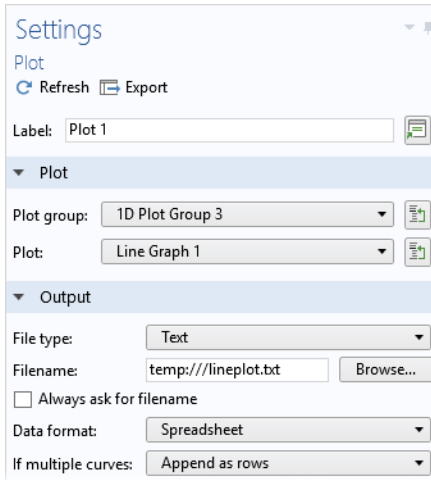
By running a **Report** subnode, a file browser window is opened for the user to select a location and file name for the report.

For more detailed control over file import and export, you can instead use a file scheme.

FILE EXPORT USING A TEMPORARY FILE

Some applications may need to produce temporary files, and this 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.

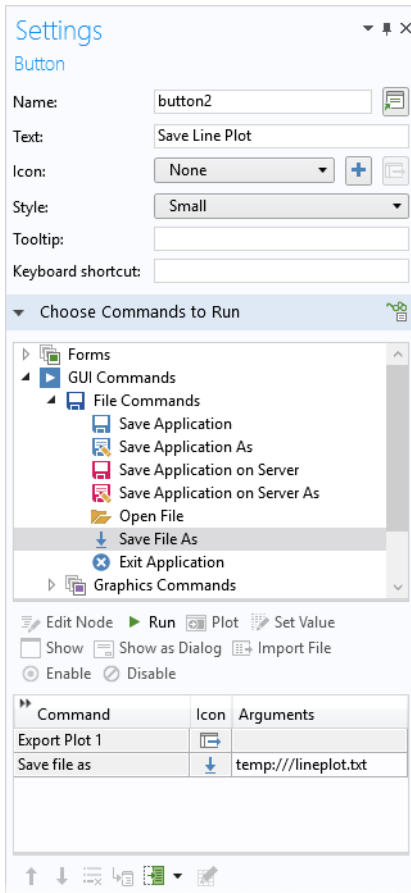
The example below shows the **Settings** window of an **Export > Plot** node that is used to export plot data as numerical values.



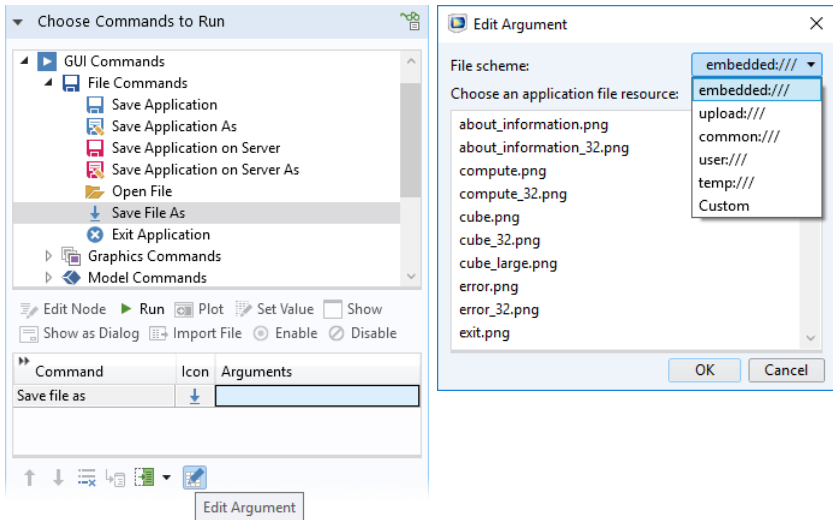
The **Filename** in its **Output** section is set to `temp:///lineplot.txt`.

To save the plot to disc in this example, a button is 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.



The corresponding method code is as follows:

```
model.result().export("plot1").run();
fileSaveAs("temp:///lineplot.txt");
```

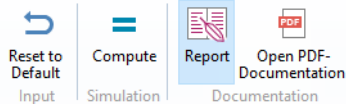
The use of Temporary Files for File Export

Note that as a first step, in the example above, the file is written to a temporary file, using the call to `model.result().export("plot1").run()`. This step is done automatically by the application. In the second step, the method `fileSaveAs` opens a file browser and lets the user of the application choose the file location, for example, a folder on the computer's local file system or to a network folder. This extra step is needed in order for the application to function in a web browser. Due to the security settings of a typical web browser, the application is not permitted to automatically save a file to an arbitrary location. Instead, the application is allowed to save to a few specific locations including the temp folder, whose location is specified in the **Preferences** dialog box settings. The other locations are the user and common folders, also specified in the **Preferences** settings. For more examples of file export, see the *Application Programming Guide*.

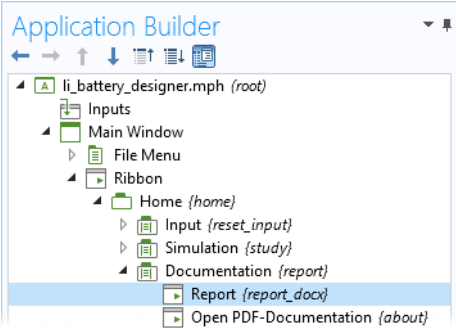
CREATING REPORTS USING LOW-LEVEL FUNCTIONALITY

This section describes creating reports using low-level functionality. For a more direct method, see "File Export" on page 312.

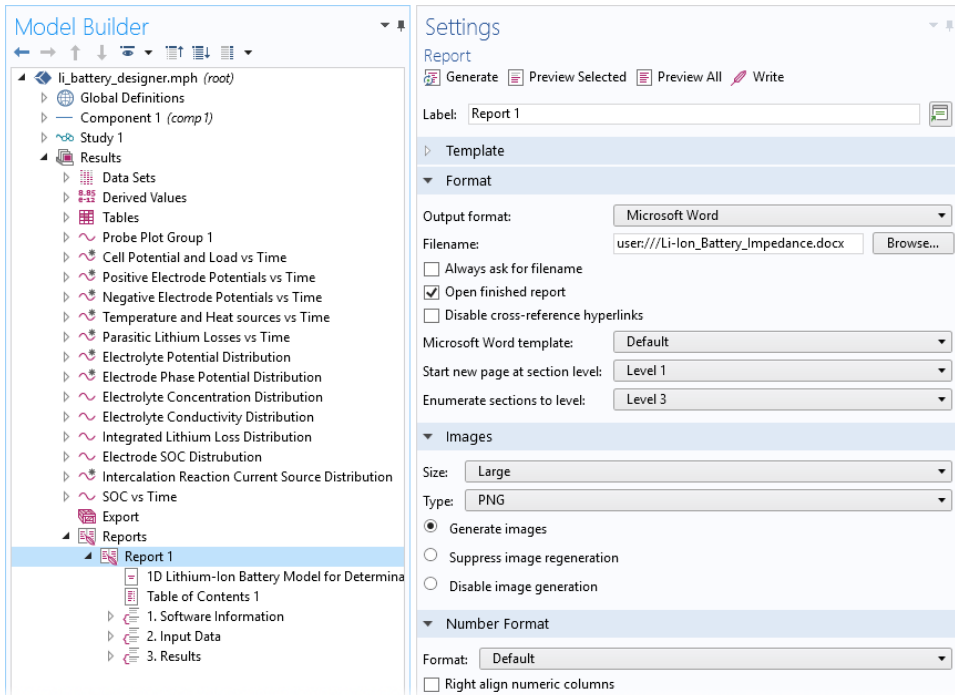
The example below shows an application where a report in the Microsoft® Word® format (.docx) can be saved by the user. The figure below shows a tab in the ribbon of the application. In this tab, there is a **Report** button in the **Documentation** section.



The associated application tree node is shown in the figure below.



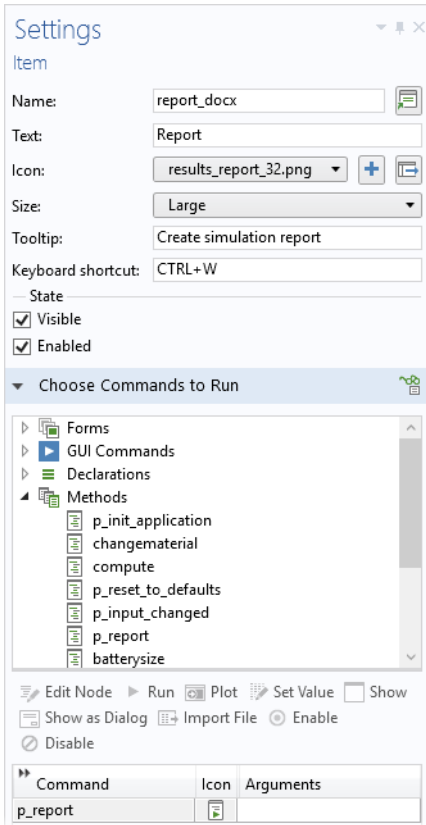
The following figure shows how the syntax `user:///file` was used in the **Filename** field in the **Settings** window of the **Report** node of the Model Builder.



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 `p_report` contains the following code:

```
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.

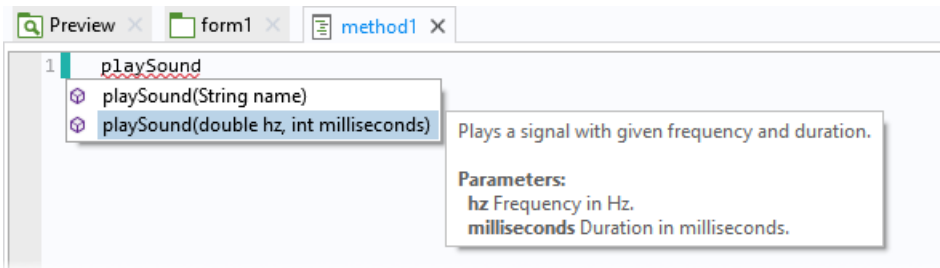
SHORTCUT	ACTION	APPLICATION BUILDER	FORM EDITOR	METHOD EDITOR
Ctrl+A	Select all	√	√	√
Ctrl+D	Deselect all		√	
Ctrl+C	Copy	√	√	√
Ctrl+V	Paste		√	√
Ctrl+X	Cut	√	√	√
Del	Delete	√	√	√
Ctrl+N	Create a new application	√	√	√
Ctrl+S	Save an application	√	√	√
Ctrl+F8	Test an application	√	√	√
Alt+Click	Edit certain form objects		√	
Ctrl+Pause	Stop a method	√		
Ctrl+Shift+F8	Apply changes	√	√	√
Ctrl+R	Record code			√
F11	Go to node			√
Ctrl+K	Create shortcut	√	√	√
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	Compile Application/Create Add-in	√		
F9	Check syntax			√

SHORTCUT	ACTION	APPLICATION BUILDER	FORM EDITOR	METHOD EDITOR
Ctrl+F	Find and replace text in methods			√
Ctrl+Space, Ctrl+/, or Ctrl+OEM2	Autocomplete method code			√
Ctrl+U	Make selected code lowercase			√
Ctrl+Shift+U	Make selected code uppercase			√
Ctrl+B	Toggle breakpoint on selected line			√
Ctrl+M	Toggle between matching parentheses, square brackets, or curly braces			√
Ctrl+Shift+M	Select all characters between matching parentheses, square brackets, or curly braces			√
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+Shift+A	Go to Application Builder window		√	√
Ctrl+Shift+M	Go to Model Builder	√	√	
Ctrl+Alt+Left-Click	Create a local method or open a method associated with a form object		√	
Ctrl+Alt+Double-Click	Open a method from Method editor code			√
Alt+F4	Close window	√	√	√
Ctrl+F4	Close document		√	√
Ctrl+Shift+F4	Close all documents		√	√
Ctrl+7	Toggle comment on and off			√
Press Ctrl and left-click. While holding down the key and button, drag the mouse.	Copy form object		√	

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 and the application object. For detailed information on using the built-in methods and for full information on the syntax used, see the *Application Programming Guide* and the *Programming Reference Manual*.

As an alternative method of learning the syntax of these methods, you can use code completion by typing the name of the method and then use Ctrl+Space. A window will open with information on the syntax and method signature.



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	DESCRIPTION
<code>clearModel</code>	Clears the model object contents.
<code>createModel</code>	Creates a new model with a given tag.
<code>removeModel</code>	Removes a model. The embedded model cannot be removed.
<code>modelTags</code>	Returns an array of model tags for all loaded models, including the embedded model.
<code>uniqueModeltag</code>	Returns a model tag that is not in use.
<code>getModel</code>	Returns a model with a specified tag.
<code>loadModel</code>	Loads a model with a specified tag from a file.
<code>loadProtectedModel</code>	Loads a password protected model with a specified tag from a file.
<code>loadRecoveryModel</code>	Loads a model from a recovery directory/folder structure.
<code>saveModel</code>	Saves a model to a file. The filename can be a file scheme path or, if allowed by security settings, a server file path.
<code>getComsolVersion</code>	Returns the current software version as a string.

File Methods

NAME	DESCRIPTION
<code>readFile</code>	Returns the contents in a given file as a string.
<code>openFileStreamReader</code>	Returns a CsReader object that can be used to read line-by-line or character-by-character from a given file name .
<code>openBinaryFileStreamReader</code>	Returns a CsBinaryReader object that can be used to read from a given file byte-by-byte.
<code>readMatrixFromFile</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.
<code>readStringMatrixFromFile</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.
<code>readCSVFile</code>	Reads a file with comma-separated values (CSV file) into a string matrix. It expects the file to use the RFC 4180 format for CSV.
<code>writeFile</code>	Writes array data to a given file. If the spreadsheet format is used, then the data can be read by readMatrixFromFile or readStringMatrixFromFile .
<code>openFileStreamWriter</code>	Returns a CsWriter object that can write to a given file.
<code>openBinaryFileStreamWriter</code>	Returns a CsBinaryWriter object that can be used to write to a given file byte-by-byte.
<code>writeCSVFile</code>	Writes a given double or string array to a CSV file. The RFC 4180 format is used for the CSV.
<code>exists</code>	Tests whether a file with a given name exists.
<code>deleteFile</code>	Deletes a file with a given name if it exists. The file is deleted on the server.
<code>copyFile</code>	Copies a file on the server. Both the source and target names can use file scheme paths.
<code>importFile</code>	Displays a file browser dialog box and uploads the selected file to the file declaration with the given name . Alternatively, it uploads the selected file to the Filename text field in a given model object entity.
<code>writeExcelFile</code>	Writes the given string array data starting from a specified cell in a specified sheet of an Excel file.
<code>readExcelFile</code>	Reads a specified sheet of an Excel file, starting from a specified cell, into a 2D string array.

NAME	DESCRIPTION
getFilePath	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 temp:/// scheme to external code or an application.
getClientFileName	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). 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.
getClientFilePath	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). 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.

Operating System Methods

NAME	DESCRIPTION
<code>executeOSCommand</code>	Executes the OS command with a given command (full path) and parameters. When applicable, the command is run server side.
<code>fileOpen</code>	Opens a file with the associated program on the client. See also the section "File Methods".
<code>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 General>Username>Name is returned.
<code>openURL</code>	Opens a URL in the default browser on the client.
<code>playSound</code>	Plays a sounds on the client.

Email Methods

NAME	DESCRIPTION
<code>emailFromAddress</code>	Returns the email from address from the COMSOL Server or preferences setting.
<code>sendEmail</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.
<code>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

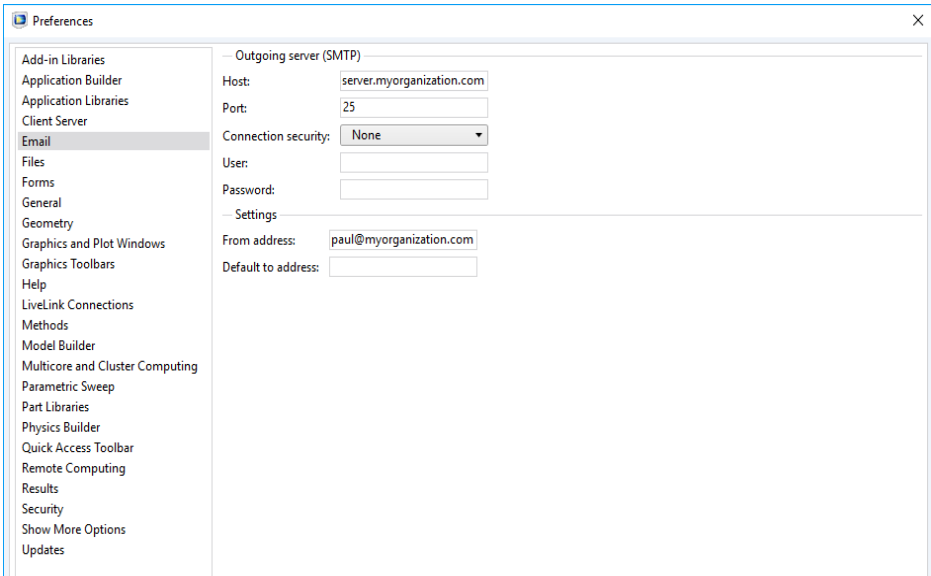
The class `EmailMessage` can be used to create custom email messages.

NAME	DESCRIPTION
<code>EmailMessage</code>	Creates a new EmailMessage object.
<code>EmailMessage.setServer</code>	Sets the email (SMTP) server host and port to use for this email message.
<code>EmailMessage.setUser</code>	Sets the username and password to use for email (SMTP) server authentication. This method must be called after the setServer method.
<code>EmailMessage.setSecurity</code>	Sets the connection security type for email (SMTP) server communication.
<code>EmailMessage.setFrom</code>	Sets the from address.

NAME	DESCRIPTION
<code>EmailMessage.setTo</code>	Sets the to addresses.
<code>EmailMessage.setCc</code>	Sets the cc addresses.
<code>EmailMessage.setBcc</code>	Sets the bcc addresses.
<code>EmailMessage.setSubject</code>	Sets the email subject line. Note that newline characters are not allowed.
<code>EmailMessage.setBodyText</code>	Sets the email body as plain text. An email can contain both a text and an HTML body.
<code>EmailMessage.setBodyHtml</code>	Sets the email body as HTML text. An email can contain both a text and an HTML body.
<code>EmailMessage.attachFile</code>	Adds an attachment from a file. The attachment MIME type is determined by the file name extension.
<code>EmailMessage.attachFile</code>	Adds an attachment from a file with a specified MIME type.
<code>EmailMessage.attachFromModel</code>	Adds an attachment created from a report, export, or table feature in the model.
<code>EmailMessage.attachText</code>	Adds a text attachment with a specified sub-MIME type, such as plain or HTML.
<code>EmailMessage.attachBinary</code>	Adds an attachment from a byte array with a specified MIME type.
<code>EmailMessage.send</code>	Sends the email to the email (SMTP) server. An email object can only be sent once.

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.



The screenshot shows the 'Preferences' dialog box with the 'Email' category selected in the left-hand sidebar. The main area is titled 'Outgoing server (SMTP)' and contains the following fields:

- Host:
- Port:
- Connection security:
- User:
- Password:

Below this section is a 'Settings' section with the following fields:

- From address:
- Default to address:

COMSOL Server provides a similar set of email preferences.

GUI-Related Methods

NAME	DESCRIPTION
Call a method directly	Call a method from the Methods list by using its name; for example, <code>method1()</code> , <code>method2()</code> .
<code>callMethod</code>	Alternate way to call a method from the Methods list; used internally and in cases of name collisions.
<code>useGraphics</code>	Plots a given entity (Plot Group, Geometry, Mesh, or Explicit Selection) in the graphics form object given by a name or name path in the second argument.
<code>useForm</code>	Shows the form with a 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>	Closes the form, shown as a dialog box, with a given name.
<code>dialog</code>	Shows the form with a given name as a dialog box. Equivalent to the <code>dialog</code> method of a <code>Form</code> object; see below.
<code>alert</code>	Stops execution and displays an alert message with a given text.
<code>alert</code>	Stops execution and displays an alert message with a given text and title.
<code>confirm</code>	Stops execution and displays a confirmation dialog box with a given text and title. It also displays two or three buttons, such as "Yes", "No", and "Cancel".
<code>error</code>	Stops execution and opens an error dialog box with a given message.
<code>request</code>	Stops execution and displays a dialog box with a text field, requesting input from the user.
<code>message</code>	Sends a message to the message log.
<code>evaluateToResultsTable</code>	Evaluates a 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.

NAME	DESCRIPTION
<code>evaluateToDoubleArray2D</code>	Evaluates the given entity, a Derived Value, and returns the nonparameter 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.
<code>evaluateToIntegerArray2D</code>	Evaluates the given entity, a Derived Value, and returns the nonparameter 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.
<code>evaluateToStringArray2D</code>	Evaluates the given entity, a Derived Value, and returns the nonparameter column part of the potentially complex valued 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>	Shows the values from the tableFeature in the resultsTable form object.
<code>getChoiceList</code>	Returns an object of the type ChoiceList , representing a choice list node under the declarations branch. The type ChoiceList has associated methods that make it easy to change values and display names, see the <i>Application Programming Guide</i> .
<code>setFormObjectEnabled</code>	Sets the enable state for the form object specified by the name or name path.
<code>setFormObjectVisible</code>	Sets the visible state for the form object specified by the name or name path.
<code>setFormObjectText</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>	Sets the editable state for the form object specified by the name or name path. This functionality is only available for text field objects.
<code>setMenuBarItemEnabled</code>	Sets the enable state for the menu bar item specified by the name or name path (from the menu bar) in the first argument.
<code>setMainToolbarItemEnabled</code>	Sets the enable state for the main toolbar item specified by the name or name path (from the main toolbar) in the first argument.

NAME	DESCRIPTION
setFileMenuItemEnabled	Sets the enable state for the file menu item specified by the name or name path (from the file menu) in the first argument.
setRibbonItemEnabled	Sets the enable state for the ribbon item specified by the name or name path (from the main window) in the first argument.
setToolbarItemEnabled	Sets the enable state for the toolbar form object item specified by the name or name path in the first argument.
useView	Applies a view to the graphics contents given by the name or name path in the second argument.
resetView	Resets the view to its initial state in the graphics contents given by the name or name path in the second argument.
getView	Returns the view currently used by the graphics contents given by the name or name path in the second argument.
setWebPageSource	Sets the source for the form object specified by the name or name path in the first argument.
getScreenHeight	Returns the height in pixels of the primary screen on the client system, or of the browser window if Web Client is used.
getScreenWidth	Returns the width in pixels of the primary screen on the client system, or of the browser window if Web Client is used.

GUI Command Methods

NAME	DESCRIPTION
<code>clearAllMeshes</code>	Clears all meshes.
<code>clearAllSolutions</code>	Clears all solutions.
<code>clearSelection</code>	Clears the selection in the given graphics object.
<code>exit</code>	Exits the application.
<code>fileOpen</code>	Opens a file with the associated program on the client.
<code>fileSaveAs</code>	Downloads a file to the client. See also the section "File Methods".
<code>printGraphics</code>	Prints the given graphics object.
<code>saveApplication</code>	Saves the application.
<code>saveApplicationAs</code>	Saves the application under a different name. (Or as an MPH file.)
<code>scenelight</code>	Toggles scene light in the given graphics object.
<code>selectAll</code>	Selects all objects in the given graphics object.
<code>transparency</code>	Toggles transparency in the given graphics object.
<code>zoomExtents</code>	Makes the entire model visible in the given graphics object.

Debug Methods

NAME	DESCRIPTION
<code>clearDebugLog</code>	Clears the Debug Log window.
<code>debugLog</code>	Prints the value of an input argument to the Debug Log window. The input argument can be a scalar, 1D array, or 2D array of the types string, double, integer, or Boolean.

Methods for External C Libraries

EXTERNAL METHOD

NAME	DESCRIPTION
<code>external</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. For more information, see the <i>Application Programming Guide</i> .

METHODS RETURNED BY THE EXTERNAL METHOD

The external method returns an object of type External with the following methods:

NAME	DESCRIPTION
invoke	Invokes a named native method in the library with the supplied arguments.
invokeWideString	Invokes the named native method in the library with the supplied arguments.
close	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	DESCRIPTION
<code>setProgressInterval</code>	Sets a progress interval to use for the top-level progress and display message at that level. Calling this method implicitly resets any manual progress previously set by calls to <code>setProgress()</code> .
<code>setProgress</code>	Sets a value for the user-controlled progress level.
<code>resetProgress</code>	Removes all progress levels and resets progress to <code>0</code> and the message to an empty string.
<code>showIndeterminateProgress</code>	Shows a progress dialog box with an indeterminate progress bar, given <code>message</code> , and an optional cancel button.
<code>showProgress</code>	Shows a progress dialog box with an optional cancel button, optional model progress, and one or two levels of progress information.
<code>closeProgress</code>	Closes the currently shown progress dialog box.
<code>startProgress</code>	Resets the value of a given progress bar form object name to <code>0</code> .
<code>setProgressBar</code>	Sets the value of a given progress bar form object name in the range <code>0 –100</code> and the associated progress message.

Date and Time Methods

NAME	DESCRIPTION
<code>currentDate</code>	Returns the current date as a string (formatted according to the server's defaults) for the current date.
<code>currentTime</code>	Returns the current time as a string (not including date and formatted according to the server's defaults).
<code>formattedTime</code>	Returns a formatted time using the given format. The format can either be a time unit or text describing a longer format.
<code>sleep</code>	Sleep for a specified number of milliseconds.
<code>timeStamp</code>	Current time in milliseconds since midnight, January 1, 1970 UTC.
<code>getExpectedComputationTime</code>	Returns a string describing the approximate computation time of the application. The string can be altered by the method <code>setExpectedComputationTime</code> .

NAME	DESCRIPTION
setLastComputationTime	Set the last computation time, overwriting the automatically generated time. You can use the timeStamp method to record time differences and then set the measured time in ms (a long integer).
getLastComputationTime	Returns the last computation time in the given format. The format can either be a time unit or text describing a longer format. This format is localized and the output is translated to the current language setting.

License Methods

NAME	DESCRIPTION
<code>checkoutLicense</code>	Checks out one license for each specified product.
<code>checkoutLicenseForFile</code>	Checks out one license for each product required to open an MPH file.
<code>checkoutLicenseForFileOnServer</code>	Checks out one license for each product required to open an MPH file.
<code>getLicenseNumber</code>	Returns a string with the license number for the current session. Example: <code>licensenumber=getLicenseNumber()</code>
<code>hasProduct</code>	Returns true if the COMSOL installation contains the software components required for running the specified products.
<code>hasProductForFile</code>	Returns true if the COMSOL installation contains the software components required for running the specified MPH file.
<code>hasProductForFileOnServer</code>	Returns true if the COMSOL installation contains the software components required for running the specified MPH file.

Conversion Methods

NAME	DESCRIPTION
<code>toBoolean</code>	Converts strings and string arrays to Booleans. ('true' returns true, all other strings return false).
<code>toDouble</code>	Converts floats, float arrays, strings, and string arrays to doubles.
<code>toInt</code>	Converts strings and string arrays to integers.
<code>toString</code>	Converts Booleans, integers, and doubles, including arrays, to strings.

Array Methods

NAME	DESCRIPTION
<code>getColumn</code>	Returns a string, double, integer, or Boolean array for a specified column in a 2D array (matrix). This is, for example, useful when values have been read from a file and only certain columns should be shown in a table.
<code>getSubMatrix</code>	Returns a rectangular submatrix of an input matrix. Available for string, double, integer, or Boolean 2D arrays.
<code>insert</code>	Inserts one or more elements in an array and returns the expanded array. Available for string, double, integer, or Boolean arrays.

NAME	DESCRIPTION
append	Adds one or more elements to the end of an array and returns the expanded array. Available for string, double, integer, or Boolean arrays.
remove	Removes one or more elements from an array and returns the shortened array. Available for string, double, integer, or Boolean arrays.
insertRow	Inserts one or more rows into a rectangular 2D array and returns the expanded array. Available for string, double, integer, or Boolean arrays.
appendRow	Adds one or more rows to the end of a rectangular 2D array and returns the expanded array. Available for string, double, integer, or Boolean arrays.
removeRow	Removes one or more rows from a 2D array and returns the reduced array. Available for string, double, integer, or Boolean arrays.
replaceRow	Replaces one or more rows in a rectangular 2D array and returns the array. Available for string, double, integer, or Boolean arrays.
insertColumn	Adds one or more columns into a rectangular 2D array and returns the expanded array. Available for string, double, integer, or Boolean arrays.
appendColumn	Adds one or more columns at the end of a rectangular 2D array and returns the expanded array. Available for string, double, integer, or Boolean arrays.
removeColumn	Removes one or more columns from a rectangular 2D array and returns the smaller array. Available for string, double, integer, or Boolean arrays.
replaceColumn	Replaces one or more columns in a rectangular 2D array and returns the array. Available for string, double, integer, or Boolean arrays.
matrixSize	Returns the number of rows and columns of a matrix as an integer array of length 2. Available for string, double, integer, or Boolean arrays.

String Methods

NAME	DESCRIPTION
concat	Concatenates a given array or matrix of strings into a single string using the given separators.
contains	Returns true if a given string array contains a given string.
find	Returns an array with the indices to all occurrences of a string in a string array.
findIn	Returns the index to the first occurrence of a string in a string array or the first occurrence of a substring in a string.
length	Returns the length of a string.
replace	Returns a string where a string has been replaced with another string.
split	Returns an array of strings by splitting the given string at a given separator.
substring	Returns a substring with the given length starting at the given position.
unique	Returns an array of strings with the unique values in the given array of strings.

Collection Methods

NAME	DESCRIPTION
copy	Returns a copy of the given array or matrix. Available for string, double, integer, or Boolean arrays.
equals	Returns true if all elements in the given array are equal and they have the same number of elements. Available for string, double, integer, or Boolean arrays. For doubles, comparisons are made using a relative tolerance.
sort	Sorts the given array. Note: The array is sorted in place. Available for string, double, or integer arrays. If the array is two-dimensional (a matrix), the columns are sorted by their row values from top to bottom.
merge	Returns an array with all of the elements merged from the given arrays. Available for string, double, or integer arrays.

With, Get, and Set Methods

NAME	DESCRIPTION
with	Used to make code more compact.
endwith	The ending of a with statement.
set	Sets a Boolean, integer, double, or string property value. Allows for a scalar, array, or matrix property.

NAME	DESCRIPTION
setIndex	Sets a string, double, or integer property value for a matrix or vector at a given index..
getIntArray	Gets an integer vector property.
getIntMatrix	Gets an integer matrix property.
getBoolean	Gets a Boolean property.
getBooleanArray	Gets a Boolean vector property.
getBooleanMatrix	Gets a Boolean matrix property.
getDouble	Gets a double property.
getString	Gets a string scalar, vector, or matrix property.
getDoubleArray	Gets a double vector property or parameter.
getDoubleMatrix	Gets a double matrix property or parameter.
getStringArray	Gets a string vector property or parameter.
getStringMatrix	Gets a string matrix property or parameter.
getDbfStringArray	Returns the value as a matrix of strings.
getInt	Gets an integer property.
get	Returns a variable expression.
descr	Returns a variable description.

Model Builder Methods for use in Add-Ins

For writing add-in method code that operates on the current component, current mesh, current physics etc. use the methods in the table below.

NAME	DESCRIPTION
getCurrentComponent	Returns an object of the type ModelNode for the current component.
getCurrentPhysics	Returns an object of the type Physics for the current physics interface.
getCurrentMesh	Returns an object of the type MeshSequence for the current mesh.
getCurrentStudy	Returns an object of the type Study for the current component.
getCurrentPlotGroup	Returns an object of the type ResultFeature for the current component.

NAME	DESCRIPTION
getCurrentNode	Returns an object of the type ModelEntity for the current component.
selectNode	Select a particular model tree node.

These methods return the corresponding entity such that the method code in an add-in can operate on it. When called from an application a method in this category returns `null`. Also, `null` is returned if no entity of the corresponding type exists such that nothing is current.

Appendix F — Guidelines for Building Applications

General Tips

- 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”, etc.
- 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 for which it is intended.
- Be minimalistic. From the developer’s point of view, it is much easier to make sure logic works, organize, debug, maintain, and further develop the app. From a user’s point of view, it is 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

- Do not create more methods than necessary.
Fewer methods give you a shorter list of methods to browse through when looking for something. Fewer methods usually mean fewer lines of code to worry about.
 - If several methods you wrote do essentially the same thing, consider merging them into one method and dealing with the different cases by input arguments.
 - Do not 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, or 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 have less to

remember and you will find what you are looking for easier. Long names are better than hard-to-understand short names.

Method naming examples:

- Start all methods that do not 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 menu item methods).
- The points above apply to method code as well: be minimalistic, use as few lines of code and variables as possible, use descriptive names for variables, use long names instead of hard-to-understand short names, and optimize code to run efficiently.
 - The above points apply to declarations as well: use good names, don't use more than necessary, and declare variables where they are used (in forms and methods or in the model).

Forms

- Do not 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, while some are good for others.
- Do not 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 is 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 anyway.
 - 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 with information about how long it takes to run the simulation with default input data on a typical computer. It could be seconds, hours, or even days depending on the application, so that is something the user would like to know before hitting the compute button. Consider playing a sound to alert the user when the 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 organized as they appear in the Application Libraries tree.

NAME	APPLICATION LIBRARY
Helical Static Mixer	COMSOL Multiphysics
Transmission Line Calculator	COMSOL Multiphysics
Tubular Reactor	COMSOL Multiphysics
Tuning Fork	COMSOL Multiphysics
Effective Nonlinear Magnetic Curves	AC/DC Module
Induction Heating of a Billet	AC/DC Module
Touchscreen Simulator	AC/DC Module ¹
Absorptive Muffler Designer	Acoustics Module
One-Family House Analyzer	Acoustics Module
Li-Ion Battery Impedance	Batteries & Fuel Cells Module ²
Li-Ion Battery Designer	Batteries & Fuel Cells Module
Water Treatment Basin	CFD Module
Biosensor Design	Chemical Reaction Engineering Module
Gas Phase Equilibrium Reaction	Chemical Reaction Engineering Module
Electrical Impedance Spectroscopy	Electrochemistry Module
Cyclic Voltammetry	Electrochemistry Module
Parameterized Concrete Beam	Geomechanics Module ³
Concentric Tube Heat Exchanger	Heat Transfer Module
Equivalent Properties of Periodic Microstructures	Heat Transfer Module
Forced Air Cooling with Heat Sink	Heat Transfer Module
Inline Induction Heater	Heat Transfer Module ⁴
Thermoelectric Cooler	Heat Transfer Module ⁵
MEMS Pressure Sensor Swelling	MEMS Module

NAME	APPLICATION LIBRARY
Red Blood Cell Separation	Microfluidics Module, Particle Tracing Module ⁶
Mixer	Mixer Module ⁷
Charge Exchange Cell Simulator	Molecular Flow Module, Particle Tracing Module ⁸
Centrifugal Governor	Multibody Dynamics Module
Truck Mounted Crane Analyzer	Multibody Dynamics Module
Laminar Static Particle Mixer Designer	Particle Tracing Module
Geothermal Heat Pump	Pipe Flow Module
Solar Dish Receiver Designer	Ray Optics Module
Corrugated Circular Horn Antenna	RF Module
Frequency Selective Surface Simulator	RF Module
Microstrip Patch Antenna Array Synthesizer	RF Module
Rotor Bearing System Simulator	Rotordynamics Module ³
Si Solar Cell with Ray Optics	Semiconductor Module ⁹
Beam Section Calculator	Structural Mechanics Module
Bike Frame Analyzer	Structural Mechanics Module ¹⁰
MEMS Pressure Sensor Swelling	Structural Mechanics Module
Truss Bridge Designer	Structural Mechanics Module
Truss Tower Buckling	Structural Mechanics Module
Fiber Simulator	Wave Optics Module
Plasmonic Wire Grating	Wave Optics Module
Polarizing Beam Splitter	Wave Optics Module

¹Requires the CAD Import Module.

²Requires the Batteries & Fuel Cells Module and the Optimization Module.

³ requires the Structural Mechanics Module.

⁴Also requires the AC/DC Module.

⁵Also requires the Optimization Module and one of the AC/DC Module, MEMS Module, or Plasma Module.

⁶Requires the Microfluidics Module, the Particle Tracing Module, and either the AC/DC Module or the MEMS Module.

⁷Requires the CFD Module and the Mixer Module.

⁸Requires the Molecular Flow Module and the Particle Tracing Module.

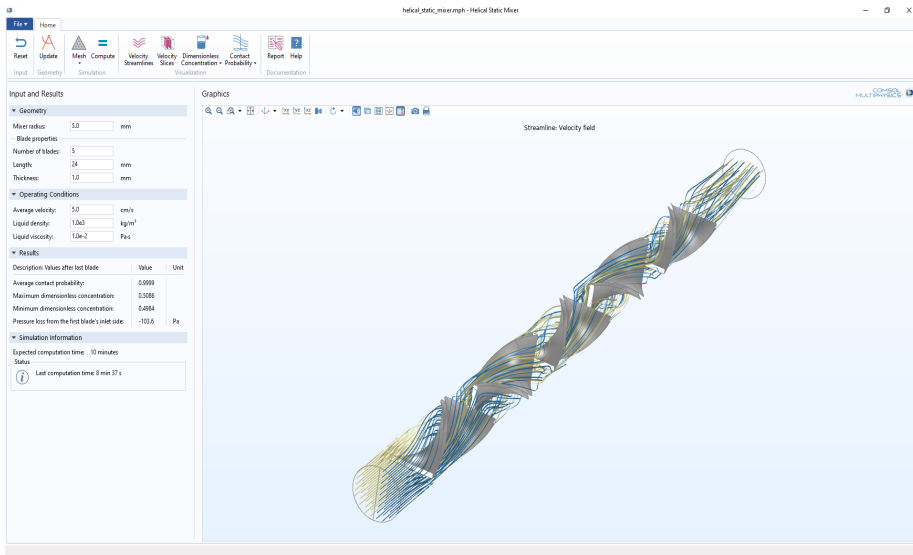
⁹Requires the Semiconductor Module and the Ray Optics Module.

¹⁰For full parametric functionality, this application requires the LiveLink™ for SOLIDWORKS®.

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 the import of experimental data.

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 of form collections of the type **Tiled or tabbed**. This application does not require any add-on products.



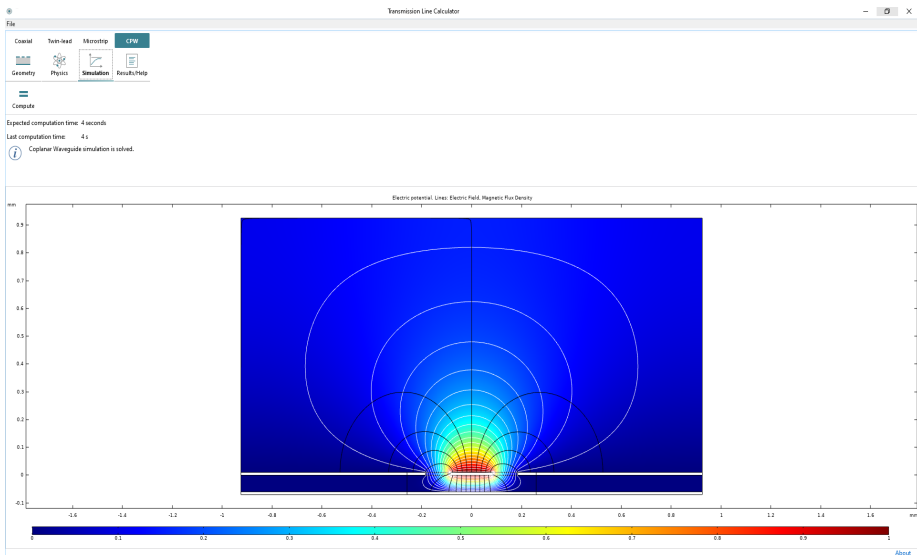
Transmission Line Calculator

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 is designed to be run on handheld devices and does not require any add-on products.



Tubular Reactor

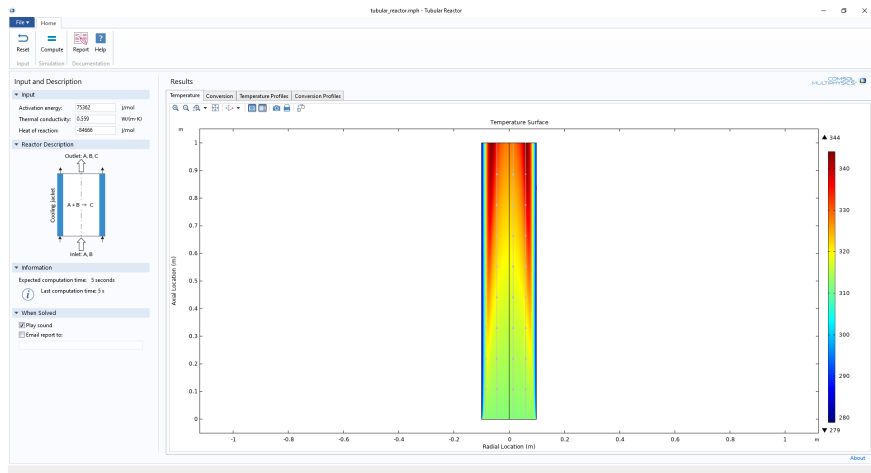
With this application, students in chemical engineering can model a nonideal tubular reactor, including radial and axial variations in temperature and composition, and investigate the impact of different operating conditions. The process described by the application is the exothermic reaction of propylene 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, 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 email. This application does not require any add-on products.



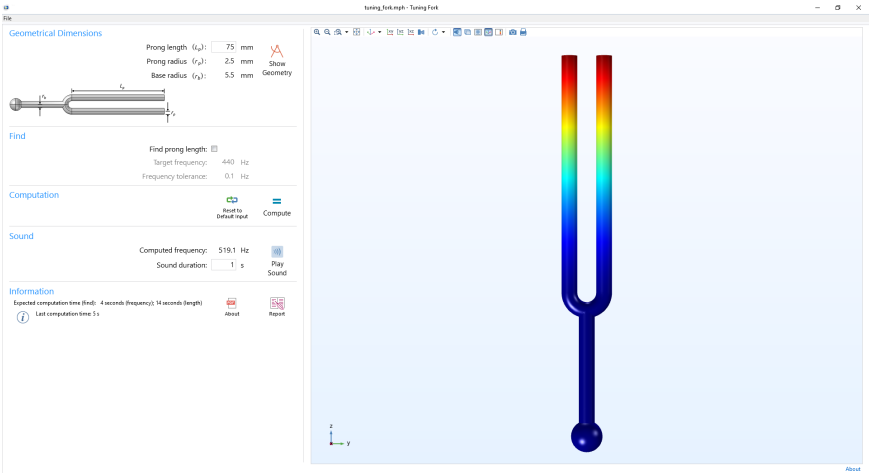
Tuning Fork

This application computes the resonant frequency of a tuning fork with a user-defined prong length. Alternatively, you can 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.

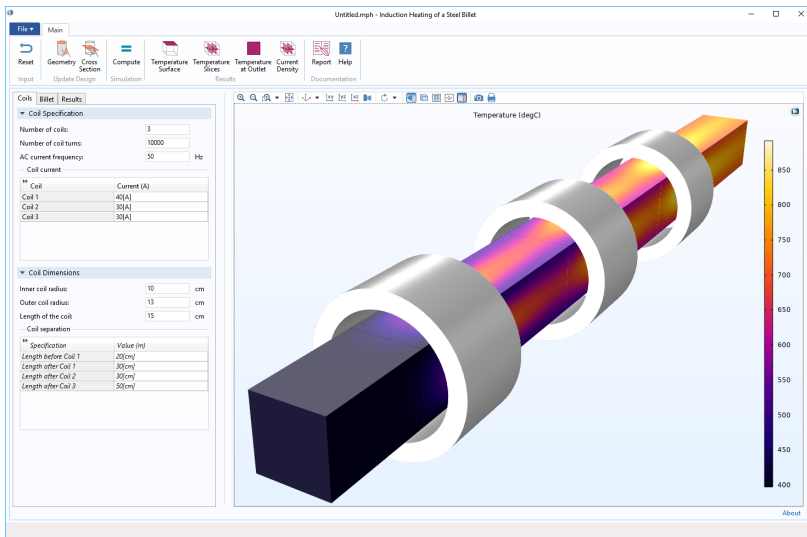
For more background theory, see the Application Library documentation for the model `tuning_fork.mph`.



Induction Heating of a Steel Billet

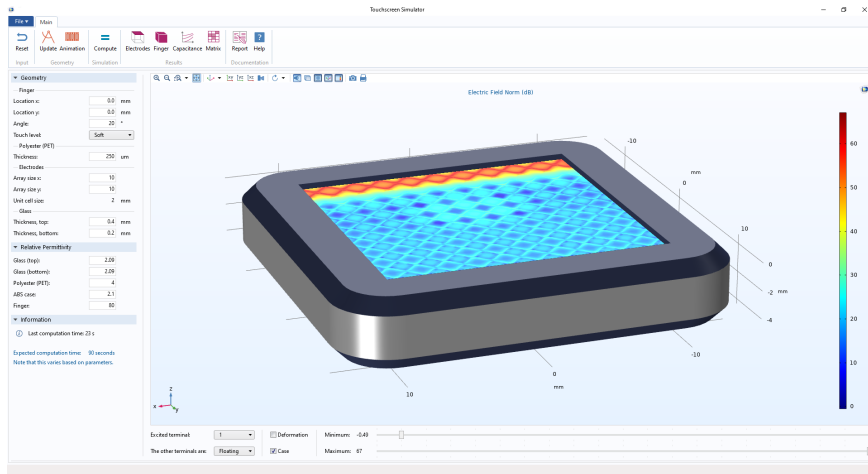
This application can be used to design a simple induction heating system for a steel billet, consisting of one or more electromagnetic coils through which the billet is moved at a constant velocity. The coils are energized with alternating currents and induce eddy currents in the metallic billet, generating heat due to Joule heating.

The billet cross section, coil number, placement, size, initial and ambient temperatures, and 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.



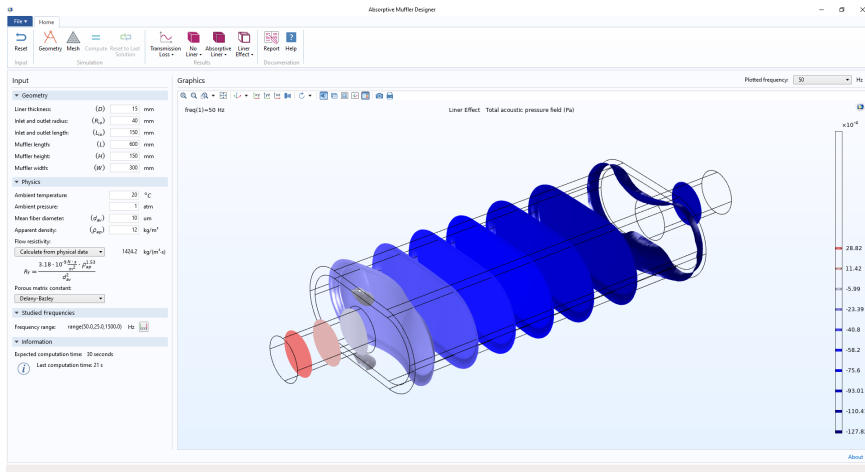
Touchscreen Simulator

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 to study and design 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; and 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.



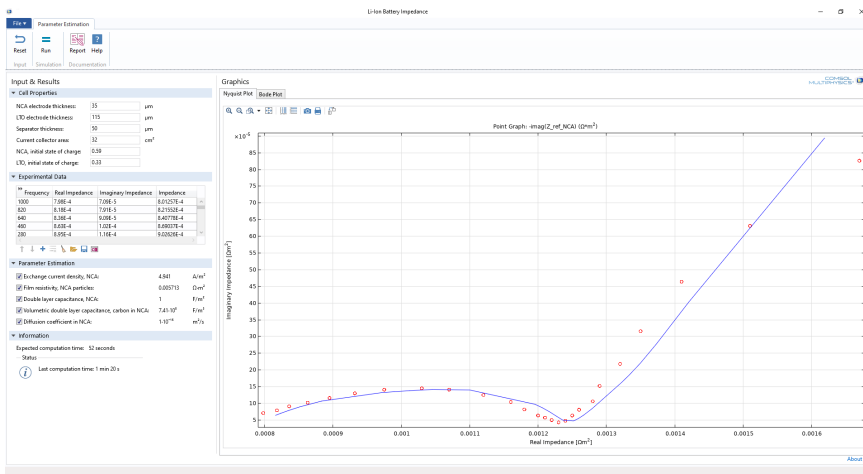
Li-Ion Battery Impedance

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.

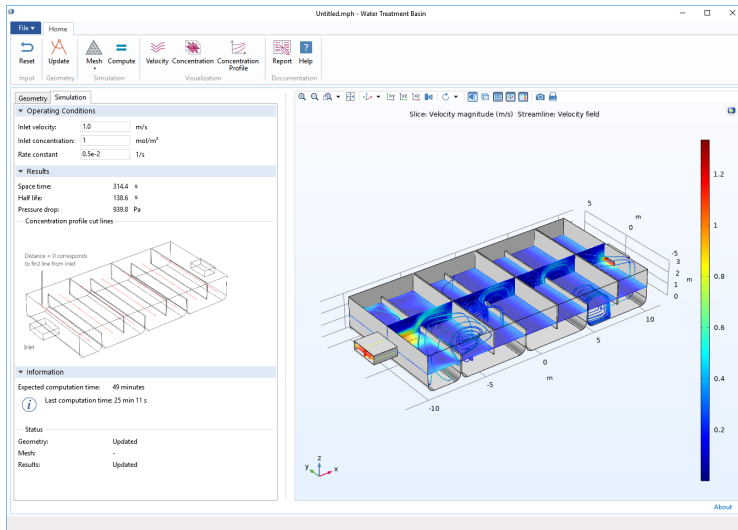


Water Treatment Basin

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 the turbulent steady flow of liquids with reactions of highly diluted solutes.

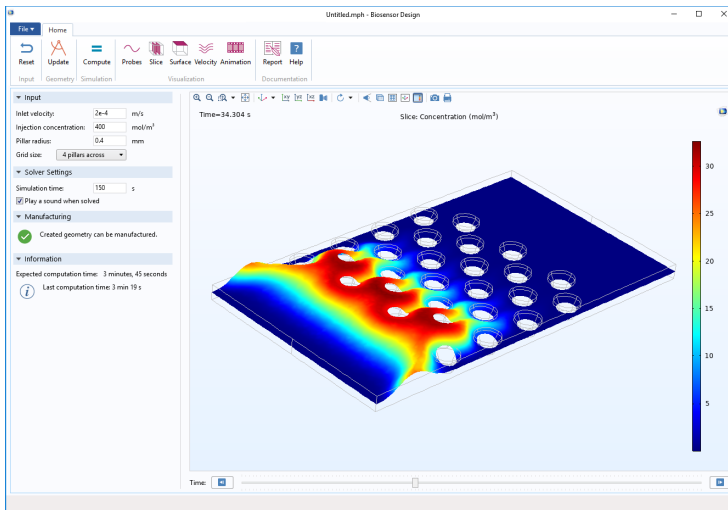
The exemplified system is a chlorination basin in a water treatment process.



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.

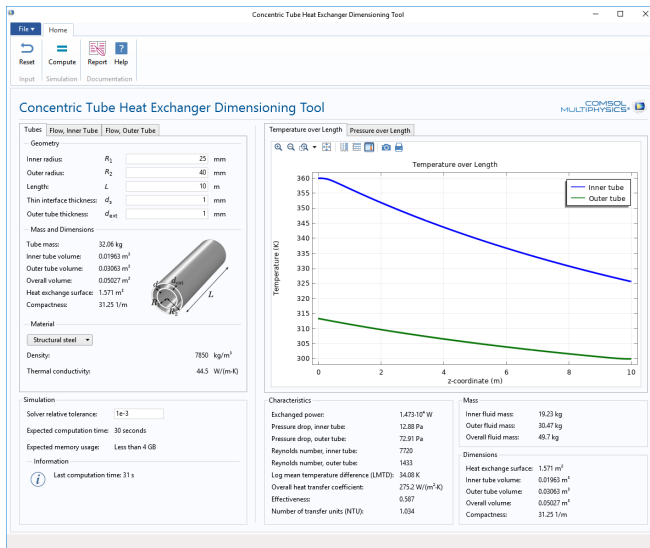


Concentric Tube Heat Exchanger

Dimensioning quantities are the first indicators of the behavior of a heat exchanger. This application aims to compute 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.



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, simulation may replace all of the details of a composite material with a homogeneous domain with equivalent properties.

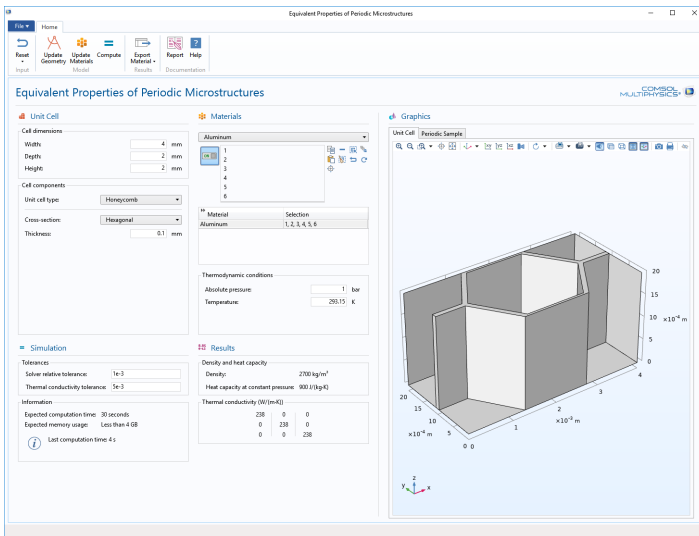
This application computes equivalent properties from the geometrical configuration and the material properties of a unit cell. It offers a choice between nine parameterizable unit cells and a list of 13 predefined materials. Extending this application to additional parallelepiped unit cells or adding 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

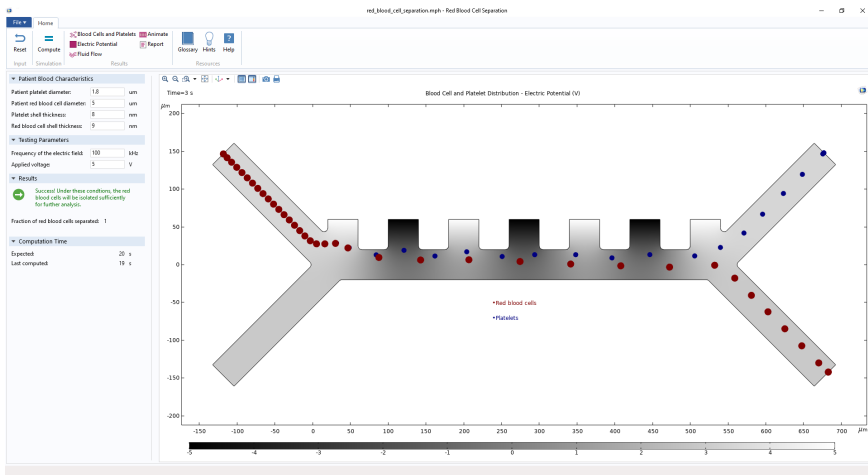
The built-in unit cell library in this application includes several widely used cell types, such as parallel stacked layers, fiber-reinforced composites, or honeycomb structures. Once the geometry is set, the physics consists of periodic heat conditions at opposite boundaries of the cell.



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.

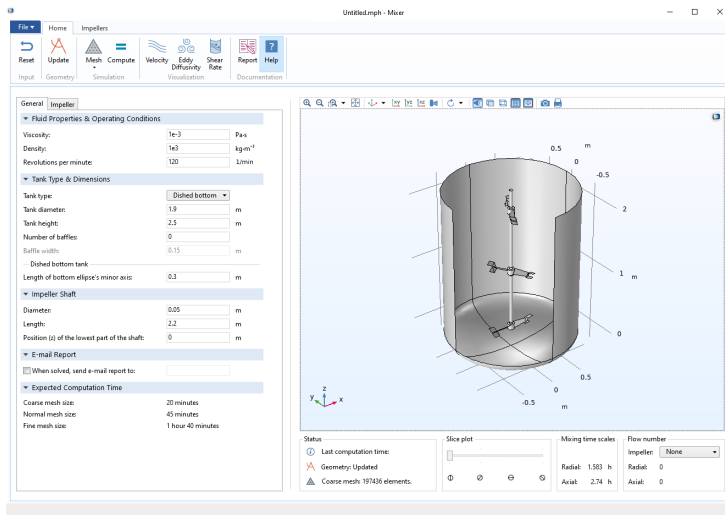
The DEP force is sensitive to the size, shape, and dielectric properties of the particles. This allows DEP to be used to separate different kinds of particles. One application of this process is in the field of bioengineering, where DEP can be used to separate different kinds of cells from a mixture. 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 because platelets cause blood to clot, which can lead samples contaminated with platelets to be unsuitable for subsequent testing once a clot has formed.



Mixer

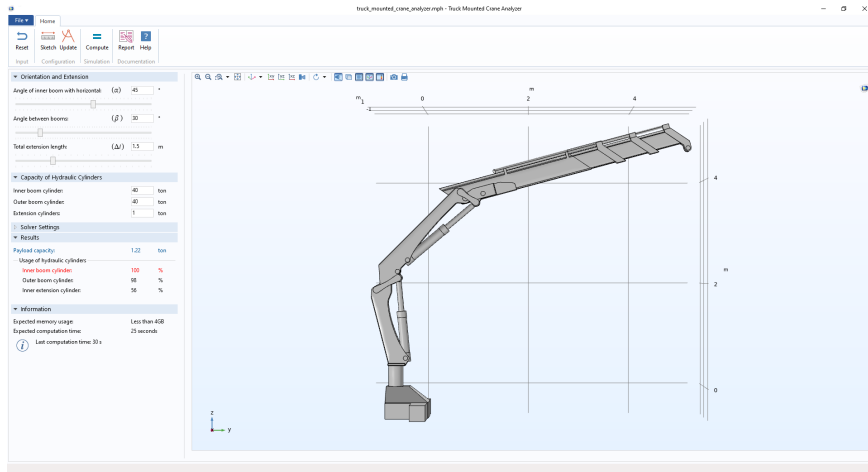
The purpose of the Mixer application is to provide a user-friendly interface where scientists, process designers, and process engineers can investigate the influence that vessel, impeller, and operational conditions have on the mixing efficiency and power required to drive the impellers. The application can be used to understand and optimize the design and operation of the mixer for a given fluid. But, perhaps most importantly, the Mixer application 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 made by an application user create very diverse geometries.



Truck Mounted Crane Analyzer

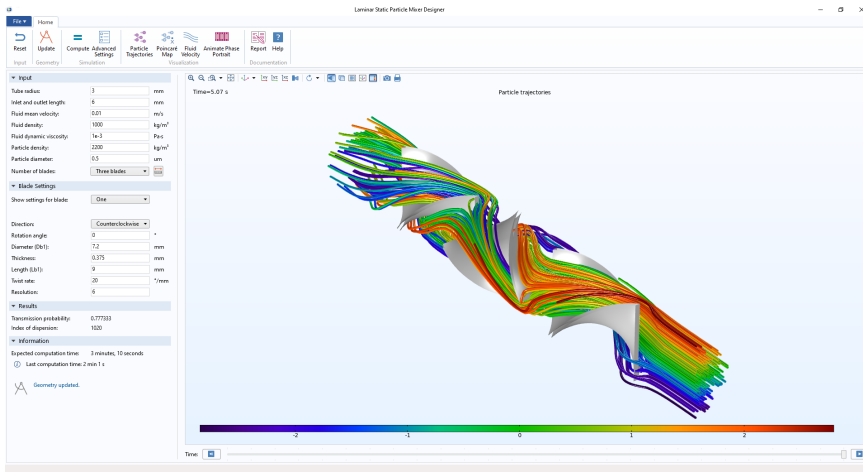
Many trucks are equipped with cranes for load handling. Such cranes have a number of hydraulic cylinders that control the crane's motion, and several mechanisms.



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

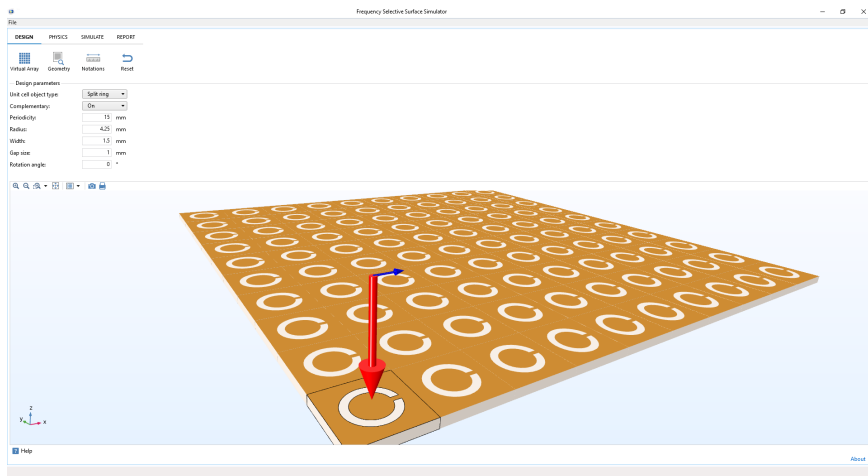
In static mixers, also called motionless or in-line mixers, a fluid is pumped through a pipe containing stationary blades. This mixing technique is particularly well suited for laminar flow mixing because it generates only small pressure losses in this flow regime. This application studies the flow in 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.



Frequency Selective Surface Simulator

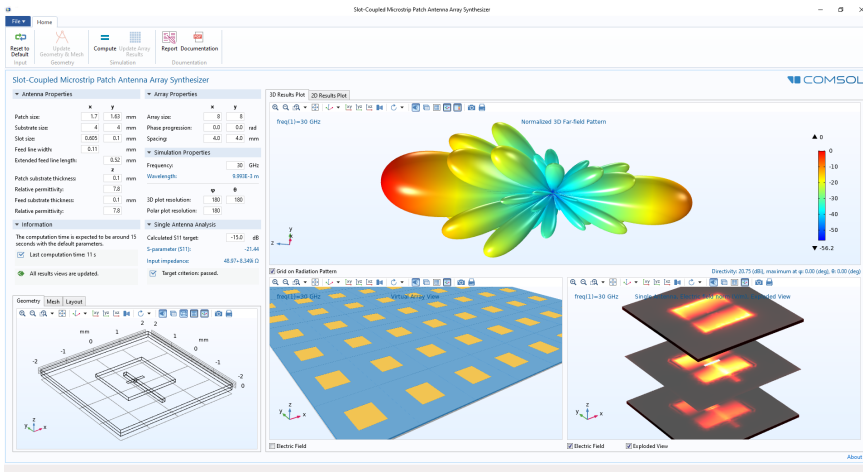
Frequency selective surfaces (FSS) are periodic structures that generate a bandpass or a bandstop frequency response. This application simulates a user-specified periodic structure chosen from the built-in unit cell types. It provides five popular FSS unit cell types, with two predefined polarizations 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. This application is designed to be run on handheld devices.



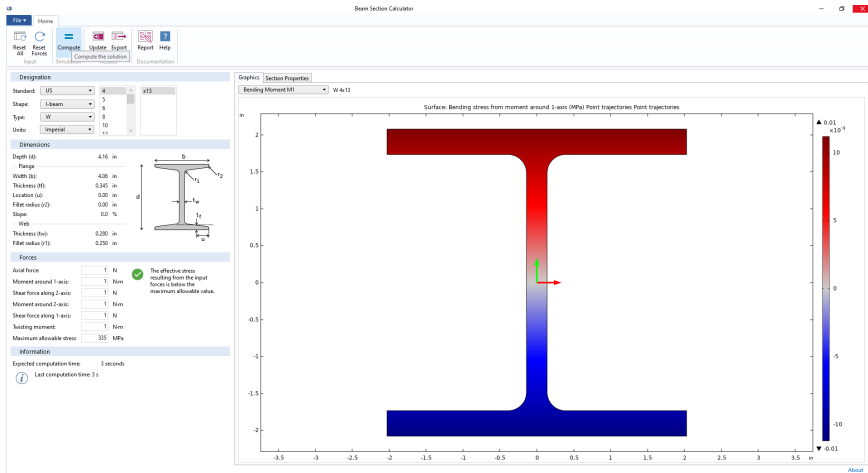
Microstrip Patch Antenna Array Synthesizer

This application simulates a single slot-coupled microstrip patch antenna that is fabricated on a multilayered low-temperature cofired 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.



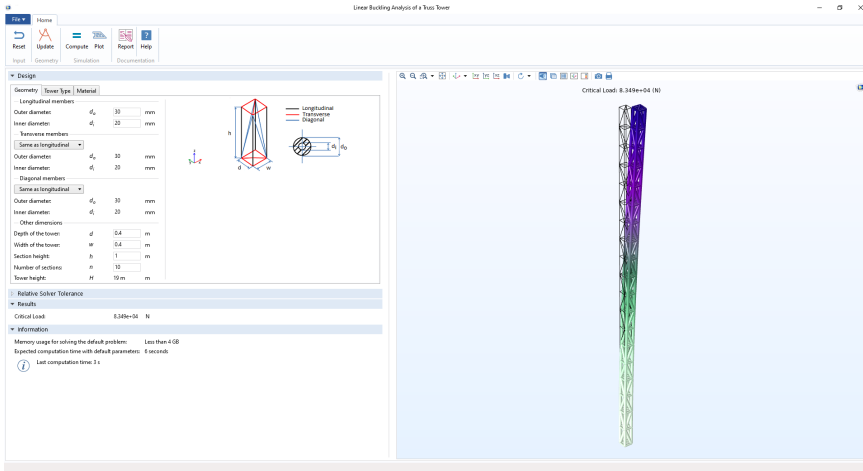
Beam Section Calculator

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 reimport this data back into the application.



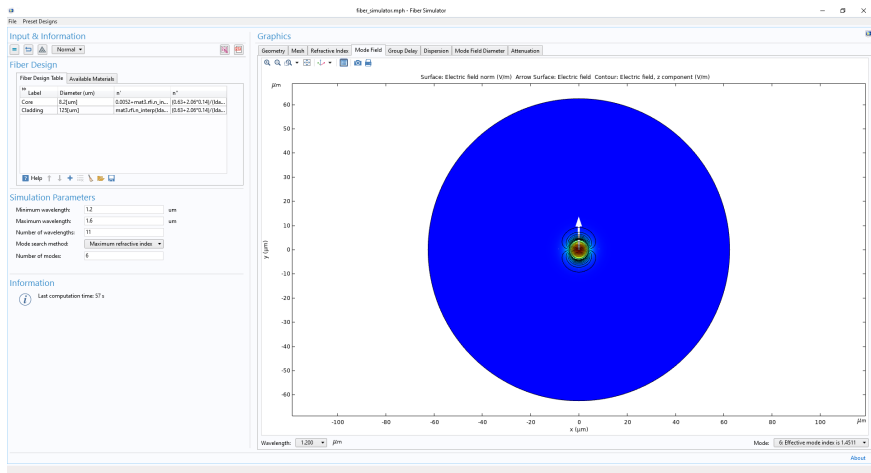
Truss Tower Buckling

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, such as various tower heights and cross-sectional areas, 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.



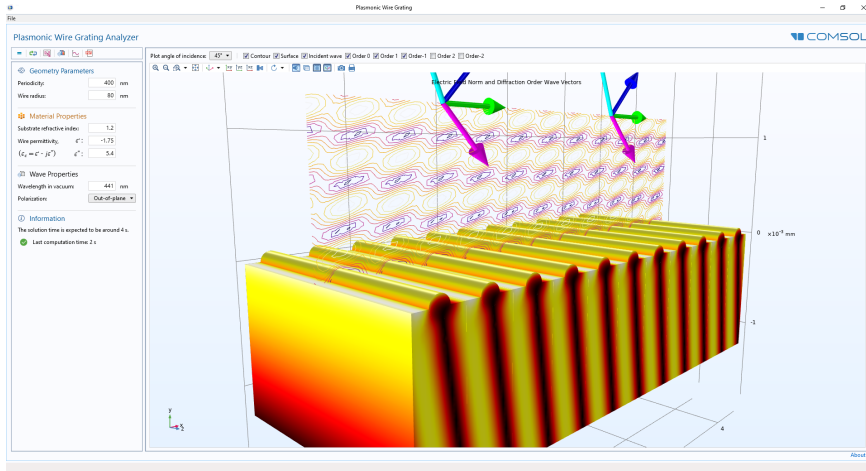
Fiber Simulator

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 148

2D array 149

3D coordinates 272

A About dialog box 39

about to shutdown event 135

action 57, 60, 134

Activation Condition 152

activation condition 233, 240

active card selector 262

Add New Choice List 152

Add New Form Choice List 152

add-in 122, 204, 205

Add-in Libraries 208

add-on products 10, 345

alert 198, 330

aligning form objects 106, 115

animation 72, 81, 347

appearance

button object 70

forms 48, 50

graphics object 72

input field object 94

multiple form objects 55

table 287

text 55

append unit from unit set 157

append unit to number 90, 292

application

saving 302

Application Argument 13, 123

Application Builder 19

desktop environment 12, 18

window 12, 15, 16

Application Builder Reference Manual

11, 211

application example

absorptive muffler designer 355

beam section calculator 367

biosensor design 358

concentric tube heat exchanger 359

equivalent properties of periodic microstructures 360

fiber simulator 369

frequency selective surface simulator 365

helical static mixer 348

induction heating of a steel billet 353

laminar static particle mixer designer 364

li-ion battery impedance 356

microstrip patch antenna array synthesizer 366

mixer 362

plasmonic wire grating 370

red blood cell separation 361

touchscreen simulator 354

transmission line calculator 349

truck mounted crane analyzer 363

truss tower buckling 368

tubular reactor 350

tuning fork 351

water treatment basin 357

Application Gallery 27

Application Libraries 10, 26, 27, 43, 345

Application Library

COMSOL Server 31

application object 165

Application Programming Guide 11, 165, 194

application tree 12, 14, 15, 16

applications

- publishing 41
- applications folder 10, 27, 345
- apply changes 22
- arguments
 - input and output 190
- arranging form objects 106
- array 148
 - ID 148
 - 2D 149
 - 2D, interactively defining 150
 - syntax 149
- array input object 139, 271
- auto complete 185
- Automatic Notation 256
- B** background color 50
- background image 50
- BMP-file 212
- Boolean variable 145, 147, 148, 221
 - conversion 337
- Boundary Point Probe 87, 164
- breakpoint 192
- browser
 - web 30, 125
- built-in method library 324
- button 59, 71, 104, 188
 - command sequence 60
 - icon 59
 - keyboard shortcut 60
 - on click event 59, 60
 - size 59, 108
 - style 59
 - text 108
 - tooltip 60
- buttons tab, New Form wizard 46
- C** C libraries
 - external 333
- CAD-file import 265, 304, 305
- cancel shutdown 135
- card 261
- card stack object 104, 145, 146, 261, 268
- cell margins 114, 120
- cells
 - merging 113
 - splitting 113
- check box object 104, 139, 147, 169, 188, 221
- check syntax 178
- choice list 58, 141, 151, 152, 226, 230, 235, 243, 244, 276, 277, 284, 285, 331
- clear selection
 - graphics 75
- click-through agreement 38
- clipboard 248, 256, 298
- close application icon 135
- Close brackets automatically 184
- code completion 185
 - tooltip 186
- code folding 184
- color
 - material 75
 - selection 75
- column settings 112, 119
- combo box object 104, 139, 145, 151, 154, 225
- command line 123
- command sequence 17, 46, 47, 60, 63, 64, 71, 80, 131, 134, 138, 165, 189, 190, 214, 290, 296
- comments
 - toggle on and off 183
- common, file scheme 304, 319
- compatible with physical quantity, unit
 - dimension check 91
- compatible with unit expression, unit
 - dimension check 91

- Compiler
 - button 35
 - node 36
- compiler 10, 22, 35
- complex numbers 256
- component syntax 184
- computation time 271
 - expected 97, 268, 270
 - last 97, 270, 336
- COMSOL Client 10, 22, 24, 25, 30, 32
 - file handing 300
 - running applications in 32, 300
- COMSOL Compiler 10, 22, 35
- COMSOL Desktop environment 12, 18
- COMSOL Multiphysics 10, 22, 23, 28, 29, 32, 120, 165, 194, 250, 253, 254
- COMSOL Runtime 36
- COMSOL Server 10, 22, 24, 25, 27, 29, 30, 32, 33, 304
 - manual 35
- COMSOL Software License Agreement 41
- confirm 198, 330
- Continue 192
- Convert to Form Method 17, 65, 165, 189
- Convert to Local Method 17, 65, 165, 189
- Convert to Method 17, 64, 66, 165, 167, 189, 308
- Coordinate 161, 163
- Copy Table and Headers to Clipboard 256
- copying
 - forms and form objects 120, 298
 - objects 53
 - rows or columns 113
- Create Local Method 188
- Create Local Variable 187
- Create New Declaration and Use It as Source 90
- Create New Form Declaration and Use It as Source 90
- creating
 - forms 15, 43
 - methods 17
- CSV file 146, 257, 290, 356
- curly brackets 184
- custom settings window 121
- D**
 - DAT file 146, 257, 290
 - Data Access 101, 173, 177
 - data change 57, 139, 190, 223, 224, 343
 - data display object 93, 96, 104
 - information node 270
 - tooltip 99
 - data file 146, 257, 290
 - Data picking 163
 - data picking 86, 163
 - data validation 90, 157
 - date 335
 - Debug Log 193
 - debug log window 193, 333
 - debugging 192, 333
 - Decimal Notation 256
 - Declaration and Use it as Source 143
 - Declarations 13, 141, 143
 - form 90, 141, 142
 - global 90
 - local 90
 - declarations node 221
 - delete button 53, 63, 266
 - deleting an object 53
 - Depth Along Line 164
 - derived values 97, 255
 - description text
 - Boolean variable 223
 - derived value 95

- desktop icon 23, 37, 125
- desktop shortcut 23, 125
- Developer tab 17, 99, 122, 175, 201
- dialog box 330
- disable form object 331
- display name, for choice list 151, 154, 227, 277, 284, 331
- displayed text 55
- Domain Point Probe 164
- domain point probe 87
- double variable 145, 147, 148
 - conversion 337
- double, data validation 92
- drag and drop, form objects 52
- duplicating
 - rows or columns 113
- duplicating an object 53
- E** edit local method 190
- edit node 61, 171, 173
- Editor Tools 171
- editor tools 57, 174, 229, 230
 - window 16
- editor tree 58, 61, 74, 171, 301, 303
- element size 103
 - change 241
- email 294, 347
 - class 327
 - methods 327
- email attachment
 - export 327
 - report 327
 - table 327
- embedded, file scheme 211, 216, 304, 311
- Enable code folding 184
- enable form object 331
- enabled state, for form objects 70
- Engineering Notation 256
- equation object 245
- error message, data validation 91, 93
- errors and warnings window 178
- Evaluation 2D 163
- Evaluation 3D 163
- evaluation tables 163
- event 57, 134, 139, 188, 190, 232
 - about to shutdown 135
 - button on click 60
 - for multiple form objects 55, 139
 - form 139
 - form object 139
 - global 13, 105, 134
 - keyboard shortcut 60
 - local 134
 - node 136
 - on close 140
 - on data change 57, 139, 190, 223, 224, 343
 - on load 57, 140
 - on startup 135
 - slider 293
- Events 13
- Excel® file 146, 256, 290
- executable 10, 22, 35
- experimental data 347
- explicit selection 83, 84
- exponent, number format 97
- export
 - email attachment 327
- export button, results table 256
- export node 300, 312
- Export Selected Image File 213
- exporting
 - results 300, 312
- external C libraries 333
- extracting subform 108
- F** file
 - commands 301
 - declaration 153

- destination 266, 306
- download 32, 302
- import 61, 139, 153, 211
- menu 132
- methods 325
- opening 301
- saving 302
- types 266
- upload 32, 302
- file browser 317
- file import object 139, 153, 211, 265, 300, 305
- file open
 - system method 327
- file scheme
 - common 304, 319
 - embedded 211, 216, 304, 311
 - syntax 211
 - temp 304
 - upload 153, 304, 309, 312
 - user 304, 319
- filename 266, 306, 325
- files library 216
- Find 179
- fit, row and column setting 108, 112
- fixed, row and column setting 108, 112
- for statement 196
- form 14, 49
 - Declarations 141
- form collection 104, 128, 259
- Form editor 19
 - desktop location 12
 - overview 14
 - preferences 18, 51
 - using 48
- form event 139
- form method 17, 134, 139, 165
- form object 15, 52, 57, 217
 - event 139
 - with associated methods 169
- form reference 259
- Form tab, in ribbon 15
- form window 15
- Forms 13
- forms node 48
- Full Precision 256
- function 16

G

- geometry 29, 46, 61, 72, 77, 86, 279, 298, 307, 309, 330
 - import 265, 304, 305
 - operations 250, 253, 254
- Geometry Entity Level 164
- geometry node 61
- get 340
- GIF-file 212
- global evaluation 97, 261
- global event 13, 134
- global method 17, 139, 165
- global parameter 197
- Go to Method 17
- go to method 64, 167
- graphics 65
 - clearing contents 77
 - commands 74
 - hardware acceleration 41
 - hardware limitations 76
 - object 44, 46, 71, 139, 330
 - plot group 77
 - Source for Initial Graphics Content 161
 - source for initial graphics content 71
 - tab, New Form wizard 46
 - toolbar 77, 104
 - using multiple objects 76
 - view 74, 80, 332
- graphics data 87, 141, 161
- grid layout mode 32, 50, 96, 105
- grid lines, sketch layout mode 106

- grow, row and column setting *108, 112*
- growth rules *108*
- H** Home tab, in ribbon *43*
- HTML
 - code *247*
 - report *247, 320*
- HTTP and HTTPS protocols *294*
- hyperlink object *293*
- I** icon *212, 296*
 - button *59*
 - close application *135*
 - command *61*
 - desktop *23, 37, 125*
 - graphics *72*
 - help *92*
 - main window *127*
 - menu item *131*
 - method *169*
 - ribbon item *131*
 - toolbar *296*
- if statement *196*
- ignore license errors *24*
- image
 - background *50*
 - formats *212*
 - object *248*
 - Preview *213*
 - thumbnail *27*
- Images library *212*
- Immediately
 - Store changes *70*
- import
 - file *61, 139, 153, 211, 266, 306*
- Indent and Format *183*
- Indent and format automatically *184*
- information card stack object *104, 268*
- information node *270*
- inherit columns *119*
- initial size, of main window *128*
- initial values, of array *148*
- initialize
 - parameter *66*
 - variable *66*
- Initializing Installer progress window *38*
- input arguments *123, 190*
- input field object *88, 104, 139, 145, 157*
 - adding *88*
 - information node *270*
 - text object *95*
 - tooltip *89*
 - unit object *95*
- Inputs *13, 123*
- inputs/outputs tab, New Form wizard *44*
- inserting
 - form objects *56, 57*
 - rows and columns *108, 111*
 - rows or columns *113*
- integer
 - data validation *92*
 - variable *145, 147, 148*
 - variable conversion *337*
- item
 - menu *130, 188*
 - ribbon *133*
 - toolbar *295*
- J** Java utility class *211*
- Java® programming language *165, 194*
- JPG file *28*
- JPG-file *212*
- K** keyboard shortcut *18, 57, 134, 173, 185, 187, 194, 321*
 - event *60, 131, 296*
- L** language elements window *16, 170, 194*
- LaTeX *95, 98, 245*
- layout mode *50, 105*

- layout options, form collection 259
- Libraries 14
- libraries node 211, 248
- license agreement 41
- license errors
 - ignoring 24
- Line Entry Method 164
- line object 246
- list box object 104, 139, 145, 151, 154, 282
- Liveliink™ for Excel® 146, 256, 290
- Liveliink™ products 33
- local event 134
- local method 17, 57, 65, 134, 139, 140, 165, 169, 188, 190, 223, 227, 298
- local variable 187
- log object 253
- logo image 72
- low-resolution displays 32
- M**
 - main form 128
 - Main Window 13
 - main window 128, 250
 - node 127
 - margins
 - cell 114, 120
 - material 233
 - material color and texture 75
 - math functions 196
 - menu 130, 133
 - bar 128, 129
 - item 71, 104, 130, 188
 - toggle item 130, 218
 - menu toggle item 104
 - merging cells 108, 113
 - mesh 46, 72, 77, 103
 - change element size 241
 - size 103
 - meshing 250, 253, 254
 - message log object 254, 330
 - method 13, 16, 56, 64, 71, 141, 165, 324
 - event 135, 139
 - form 17, 134, 139, 165
 - form object 169
 - global 17, 139, 165
 - local 17, 57, 65, 134, 139, 140, 165, 169, 188, 223, 227, 298
 - Model Builder 198
 - window 16
 - Method Call 199, 205
 - Method editor 19, 324
 - desktop location 12
 - overview 16
 - Preferences 184
 - using 165
 - Method tab, in ribbon 16
 - method, called from the Model Builder 198
 - Methods 13
 - Microsoft® Word® format 318
 - minimum size
 - form objects 114
 - Model Builder
 - method 185, 198
 - model commands 303
 - model data access 104, 137, 298
 - Model Expressions 180
 - model expressions window 16
 - model object 165, 194, 324
 - model tree node, controlling if active 199
 - model utility methods 324
 - move down
 - command sequence 63
 - table 289
 - move up
 - command sequence 63
 - table 289
 - MP4 file 248

- MPH file *14, 22, 23, 25, 29, 43, 47, 211, 303, 333*
- multiline text *96*
- multiple form objects
 - selecting *55, 139*
- N** name
 - button *59*
 - check box *223*
 - choice list *151*
 - form *49*
 - form object *55*
 - graphics object *71*
 - menu *130*
 - method *183*
 - shortcut *159*
 - variable *144*
- named selection *83*
- new element value *148*
- new form *15*
- New Form wizard *56, 57, 58, 95*
 - buttons tab *46*
 - graphics tab *46*
 - inputs/outputs tab *44*
- new method *17*
- notation
 - data display number format *97*
- number format *93, 97*
- number of rows and columns *108*
- numerical validation *92, 157*
- O** OGV file *248*
 - on click event, button *59*
 - on close event *140*
 - on data change event *57, 139, 190, 223, 224, 343*
 - on load event *57, 140*
 - On request
 - Store changes *70*
 - on startup event *135*
- open file *301*
- OpenGL graphics hardware acceleration *41*
- operating system command line *123*
- operators *195*
- optimization *347, 356*
- orthographic projection *74*
- OS commands *327*
- output arguments *190*
- Output directory
 - for compiled applications *36*
- P** panes *259*
- parameter *16, 45, 65, 89, 91, 145, 197, 298, 340*
 - combo box object *225*
 - declarations *13, 141*
 - estimation *347*
 - events *13, 134*
 - input field object *88*
 - method *177, 196*
 - slider object *292*
 - text label object *95*
- parentheses *184*
- password protected application *29*
- pasting
 - form objects *53*
 - forms and form objects *121*
 - image *248*
 - rows or columns *113*
- pixels *54, 105*
- play sound *32, 215, 327*
- plot *46, 61, 71, 77, 147, 222, 228, 285, 295, 308, 316, 330, 343*
- plot geometry command *61*
- plot group *65, 147*
- PNG file *28, 212*
- PNG-file *212*
- Point Being Modified *164*
- Polar Complex Numbers *256*

- position and size *54, 105, 107*
 - multiple form objects *55*
- positioning form objects *52*
- precedence, of operators *195*
- precision *256*
- precision, number format *97*
- preferences *18, 51, 184, 304, 305*
 - for compiled applications *40*
 - security *29*
- Preview
 - image *213*
- preview form *22*
- printing
 - graphics *74, 333*
- Probe *164*
- procedure *16*
- Programming Reference Manual *11*
- progress *250, 335*
- progress bar object *250, 335*
- progress bar, built in *250*
- progress dialog box *252, 335*
- publishing applications *41*

Q Quick Access Toolbar *22*

- Find *179*

R radio button object *104, 139, 151, 154, 275*

- Record Code *175*
- recording code *175*
- Rectangular Complex Numbers *256*
- recursion *191*
- regular expression *92*
- removing
 - password protection *29*
 - rows and columns *108, 111*
 - rows or columns *113*
- report *342, 345*
 - creating *131, 300, 312, 318, 319*
 - creating automatically *199*

- email attachment *327*
- embedding *247*
- HTML *247, 320*
- image *28*
 - node *300, 312, 319*
- request *198, 330*
- reset current view *74, 80*
- resizable graphics *32*
- resizing form objects *53*
- Results Evaluation *161, 163*
- results table object *255, 331*
- ribbon *128, 133*
 - item *105, 133*
 - section *133*
 - tab *133*
 - toggle item *104, 133, 218*
- row settings *111*
- run application *22, 24*
- running applications
 - compiled *35*
 - in a web browser *30, 300*
 - in the COMSOL Client *32*
- runtime *36*

S save

- application *47*
- running application *25*
- save application command *302*
- save as *132, 333*
- save file *302*
- Scalar *145*
- scalar variable *145, 226, 261, 292*
- scene light *74, 333*
- Scientific Notation *256*
- security settings *29*
- select all
 - graphics *75*
- selection *46, 72, 77*
 - explicit *84*
- selection colors *75*

- selection input object 84, 278
- selections 83
- selectNode method 199
- separator
 - menu 130
 - ribbon 133
 - toolbar 130, 295
- separators
 - CSV, DAT, and TXT files 290
- set value command 103
- Settings Form 122, 198, 205
- Settings Forms 121
- settings window
 - customized 121
 - Form editor 12, 14
 - Method editor 16
- shortcut
 - desktop 23, 125
- shortcuts 141, 159
 - use 180
- Show as Dialog command 66
- Show Dialog 122
- show form command 67
- Show in Model Builder 122
- shutdown
 - cancel 135
- sketch grid 106
- sketch layout mode 50, 96, 105
- slider object 104, 139, 291
- smartphones
 - running applications on 32
- software rendering 41
- solving 250, 253, 254
- sound
 - play 215
- Sounds library 213
- Source for Initial Graphics Content 161, 163
- spacer object 296
- special characters 125
- splash screen 37
- splitting cells 108, 113
- state
 - enabled, for form objects 70
 - visible, for form objects 70
- status bar 250
- Step 192
- Stop 192
- Stop Recording 177
- stopping a method 194
- Store changes 70
- string variable 13, 66, 134, 136, 145, 148, 225, 227, 235, 259, 269, 281, 308
 - conversion 337
 - methods 339
- subroutine 16
- Switch to Model Builder and Activate
 - Data Access 90
- syntax errors 178
- syntax highlighting 182, 184
- system methods 327
 - OS commands 327

T

- table
 - email attachment 327
- table object 139, 148, 286, 330, 347
- tables, model tree 255
- tablets
 - running applications on 32
- Target for Data Picking 86, 163
- temp, file scheme 304
- temporary file 317
- test application 22, 24
- test in web browser 22
- text 131
 - color 50
 - file 145, 256, 290
 - label object 88, 95, 97
 - object 104, 139, 281

- information node 270
 - Theme 184
 - thumbnail image 27
 - time 335
 - time parameter
 - combo box object 230
 - timestamps 254
 - title
 - form 49
 - main window 127
 - menu 130
 - toggle button 104, 218
 - size 108
 - text 108
 - toggle item
 - menu 104, 130, 218
 - ribbon 104, 133, 218
 - toolbar 130, 256, 295
 - button, table object 289
 - graphics 77, 104
 - item 104, 295
 - separator 130, 295
 - tooltip
 - button 60
 - data display object 99
 - input field object 89
 - method editor 186
 - slider object 292
 - toolbar button 296
 - unit mismatch 91
 - transparency 74, 333
 - TXT file 145, 256, 290
- U** Unicode 95, 98
- unit
 - changing using unit set 154
 - dimension check 91, 157
 - expression 91
 - groups 154
 - lists 154
 - object 88, 95
 - Unit Groups 154
 - Unit Lists 154
 - Unit Set 154
 - unit set 92, 141, 244, 277, 285
 - Untitled.mph 25
 - upload
 - file scheme 153, 304, 309, 312
 - URL 125, 247, 294
 - use as source
 - array input object 273
 - card stack object 262
 - check box object 223
 - combo box object 226
 - data display object 97
 - declaration 143
 - explicit selection 85, 278
 - graphics object 71
 - information card stack object 269
 - input field object 88
 - list box object 283
 - radio button object 276
 - results table object 256
 - selection input object 85, 278
 - slider object 292
 - table object 286
 - text object 281
 - Use component syntax 184
 - use shortcuts 180, 181
 - user
 - file scheme 304, 319
 - user interface layout 15
 - username 327
- V** Value 151, 154
- variable 13, 141, 177
 - accessing from method 196
 - activation condition 152
 - Boolean 147, 148, 221
 - declaration 13, 141

- derived values 97
 - double 147, 148
 - events 13, 134, 136
 - find and replace 179
 - input field object 88
 - integer 147, 148
 - name completion 185
 - scalar 226, 261, 292
 - slider object 292
 - string 145, 148, 225, 227, 259, 269, 281
 - text label object 95
- video
 - controls 250
 - player 250
- video object 248
- view
 - go to default 3D 80
 - graphics 74, 80, 332
 - reset current 74, 80
- View all code 184
- visible state, for form objects 70
- volume maximum 97
- W** WAV-file 213
- web browser 10, 25, 30, 125
 - file handling 300
- web page
 - hyperlink 294
- web page object 247
- WebGL 30
- WebM file 248
- while statement 196
- with statement 184, 196, 339
- with statements 184
- wrap text
 - text label object 96
- Z** zoom extents 72, 74, 285, 308, 333