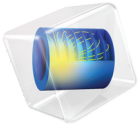


INTRODUCTION TO
Application Builder

6.2

COMSOL

Introduction to Application Builder

© 1998–2023 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; 10,019,544; 10,650,177; and 10,776,541. 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 6.2

Contact Information

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

If you need to contact Support, an online request form is located on 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/product-update
- COMSOL Blog: www.comsol.com/blogs
- Discussion Forum: www.comsol.com/forum
- Events: www.comsol.com/events
- COMSOL Video Gallery: www.comsol.com/videos
- Support Knowledge Base: www.comsol.com/support/knowledgebase

Part number: CM020011

Contents

Preface 9

Introduction 10

The Application Builder Desktop Environment 12

The Application Builder and the Model Builder 21

Parameters, Variables, and Scope 22

Running Applications 24

Running Applications in COMSOL Multiphysics 24

Running Applications with COMSOL Server 32

Compiling and Running Standalone Applications 38

Publishing COMSOL Applications 45

Getting Started with the Application Builder 47

Themes 54

The Form Editor 55

The Individual Form Settings Windows 55

Local Forms 58

Form Editor Preferences 59

Form Objects 59

Editor Tools in the Form Editor 65

Button 67

Graphics 79

Input Field 97

Unit 104

Text Label 104

Data Display 105

Data Access in the Form Editor 108
Sketch and Grid Layout 114
Show Errors. 131
Copying Between Applications 131
Using Forms in the Model Builder 132
Inputs. 134
The Main Window Editor. 138
Menu Bar and Toolbar. 142
Ribbon 144
Interactive Editing of Menus and Ribbon Tabs 146
Subwindows 147
Events 150
Events at Startup and Shutdown. 151
Global Events 151
Timer Events 155
Form and Form Object Events 156
Using Local Methods 159
Declarations 160
Scalar 164
Array 1D 167
Array 2D 168
Choice List. 170
File 172
File Type. 172
Unit Set 174
Shortcuts 179
Graphics Data 182

The Method Editor 185
Converting a Command Sequence to a Method 185
Language Elements Window 190
Editor Tools in the Method Editor 191
Data Access in the Method Editor 193
Recording Code 195
Checking Syntax 198
Find and Replace 199
Model Expressions Window 200
Use Shortcut 201
Syntax Highlighting, Code Folding, and Indentation 202
Method Editor Preferences 204
Ctrl+Space and Tab for Code Completion 205
Creating Local Variables 207
Local Methods 208
Methods with Input and Output Arguments 210
Debugging 212
The Model Object 215
Language Element Examples 215
Running Methods in the Model Builder 219
Creating Add-Ins 226
Add-In Libraries 230
Workflow when Creating and Editing Add-Ins 232
Libraries 233
Images 234
Sounds 234
Files 236
Appendix A — Form Objects 237

List of All Form Objects. 237
Check Box. 238
Toggle Button 242
Combo Box. 244
Equation. 265
Line. 266
Web Page 267
Image 268
Video 269
Progress Bar. 270
Gauge. 273
Log 276
Message Log 277
Results Table 279
Form. 281
Form Collection 283
Card Stack. 286
File Import 291
Information Card Stack 295
Array Input. 298
Radio Button 301
Selection Input 303
Text 307
List Box 307
Table. 312
Slider. 317
Knob. 319
Hyperlink 321

Toolbar. 323
Spacer. 324
Appendix B — Copying Between Applications. 326
Appendix C — File Handling and File Scheme Syntax 328
File Handling with COMSOL Server 328
File Scheme Syntax. 331
File Import 334
File Export 341
Appendix D — Keyboard Shortcuts. 350
Appendix E — Built-In Method Library 353
Appendix F — Guidelines for Building Applications 371
Appendix G — The Application Library Examples. 374
Index 419

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 simulation 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, Main Window 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. In addition, you can run

COMSOL applications by connecting to a COMSOL Server from your Android™ device via the COMSOL Client for Android™ app on the Google Play™ store. COMSOL Server does not include the Application Builder, Physics Builder, or Model Builder tools that come with the COMSOL Desktop environment. Any application created with the Application Builder will automatically work with a web browser or any client.

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 371.

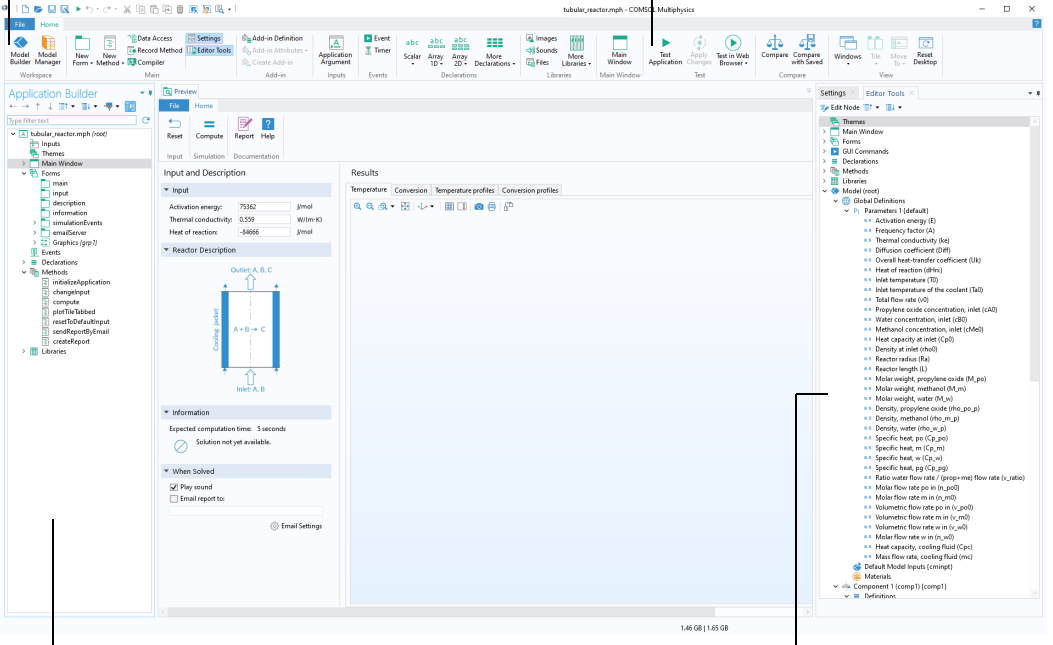
Additional Documentation

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, APPLICATION BUILDER, and MODEL MANAGER—Switch between the Model Builder, the Application Builder, and the Model Manager.

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



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.

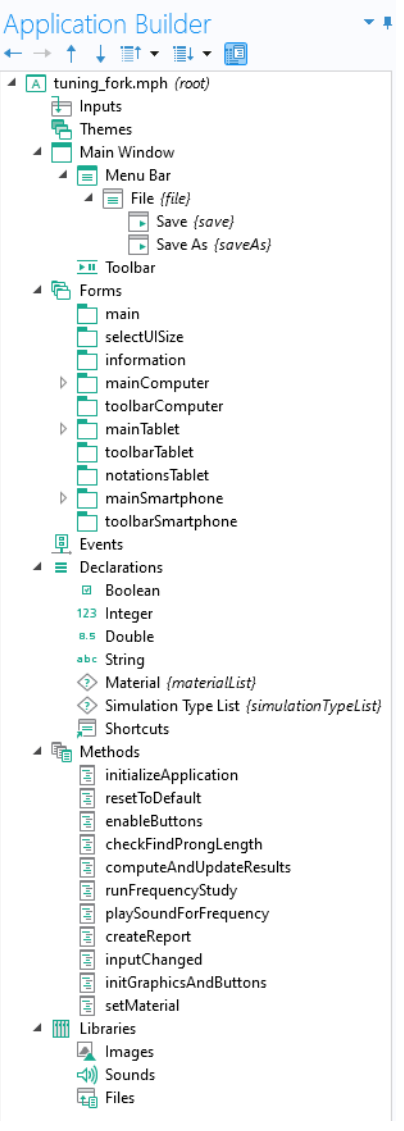
The screenshot above is representative of what you will see when you are working with the Application Builder. The main components of the Application Builder desktop environment are:

- Application Builder window and ribbon tab
- COMSOL Desktop environment
- Form Editor (see page 55)
- Main Window Editor (see page 138)
- Method Editor (see page 185)

The Application Tree

The application tree consists of the following nodes:

- **Inputs**
- **Themes**
- **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 **Themes** node has a **Settings** window with choices for the desktop color themes, as well as font, text color, and other settings that will affect the general appearance of an application.

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 or folders containing local 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. Timer events can be used to operate an app as a digital twin, enabling automated actions to be performed at regular intervals.

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

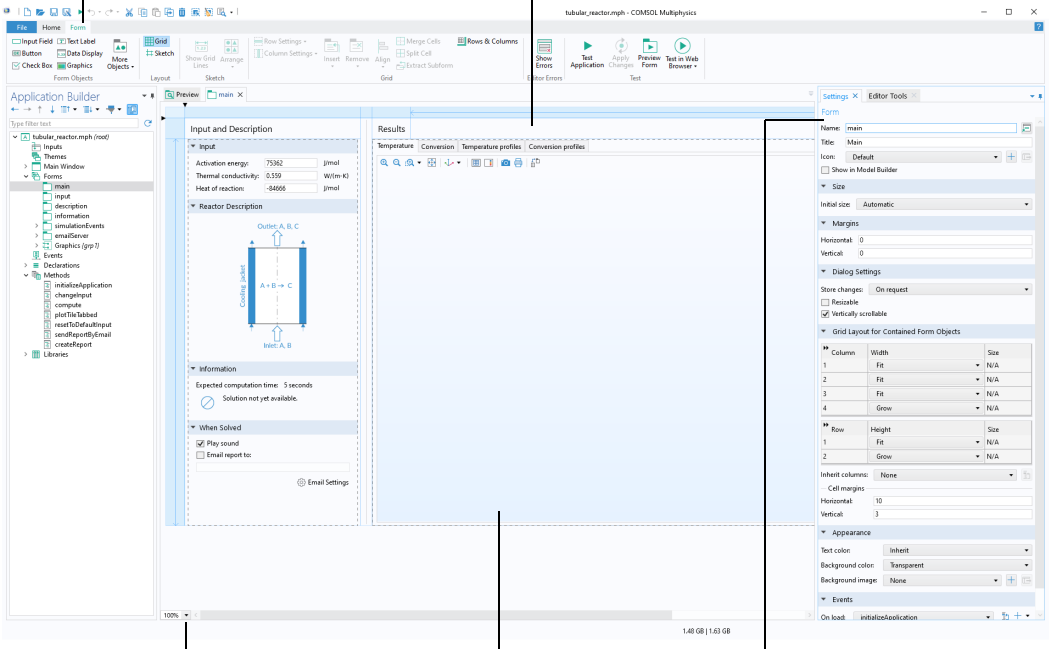
The **Methods** node contains subnodes that are methods. Methods contain code for actions not included among the standard run commands of the model tree nodes in the Model Builder. The methods may, for example, execute loops, process inputs and outputs, and send messages and alerts to the user of the application. Methods can modify the model object of a running application or the model object represented by the Model Builder in the current session. The latter being utilized when building an add-in.

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.



ZOOM — Zoom to get a better view of form objects.

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 Form Wizard with a number of layout templates.

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:

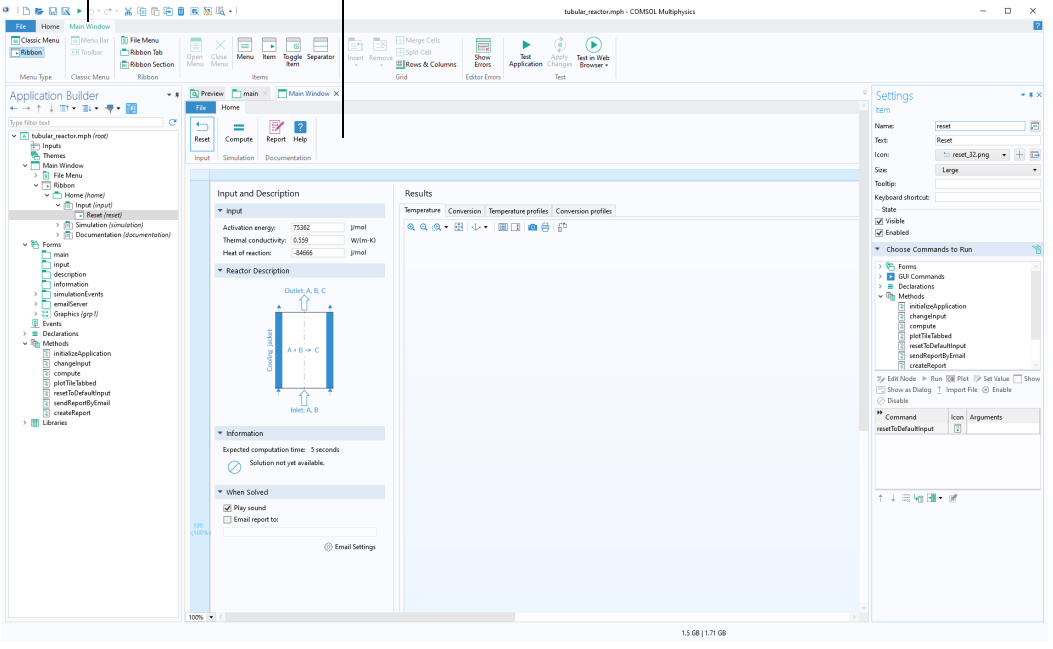
- In the application tree, double-click the **form1** node.
- In the application tree, right-click the **form1** node and select **Edit**.

You can also add forms that are local to other forms. When applicable, this option is available as a menu option from the **New Form** button.

The Main Window Editor

MAIN WINDOW TAB — The Main Window tab in the ribbon gives easy access to the Main Window Editor.

MAIN WINDOW EDITOR — The Main Window Editor allows you to design menus, menu items, ribbon tabs, ribbon items, and subwindows. You can move items around by dragging. Click an object to edit its settings.



Use the Main Window Editor for menu and ribbon layout by creating menus with menu items and ribbon tabs with ribbon items. In addition, the Main Window Editor, available directly from the Main Window node, is used for subwindow layout.

The main components of the Main Window Editor are:

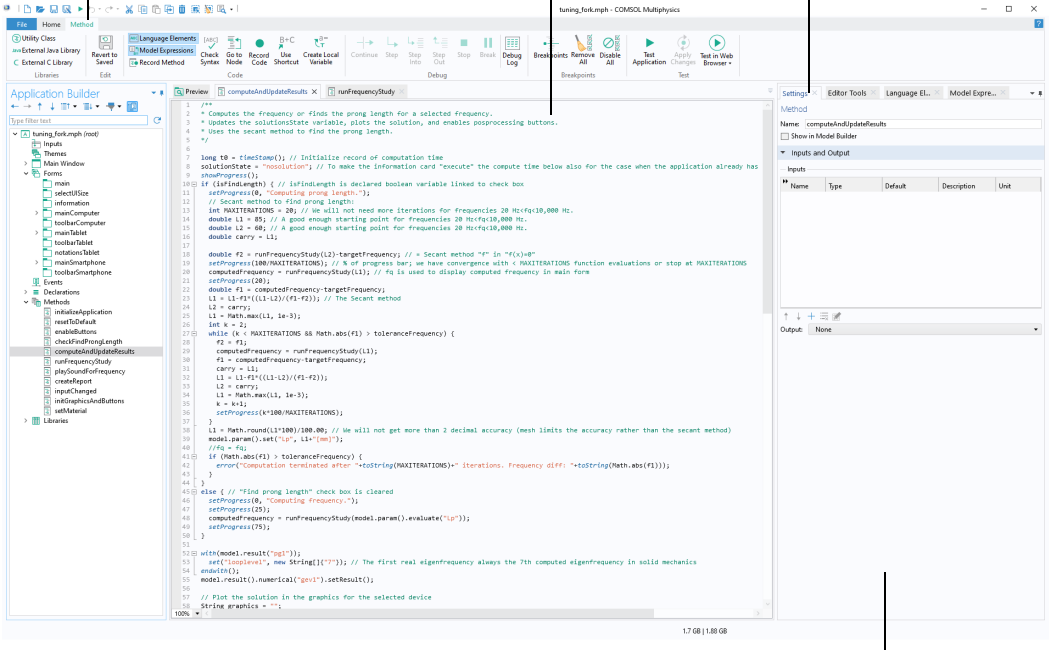
- Menu editor
- File menu editor
- Ribbon tab editor
- Subwindow editor

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.

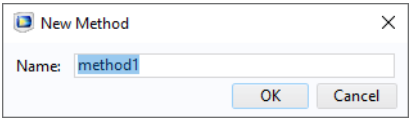
Use the Method Editor to write methods for actions not covered by the standard use of the model tree nodes. A method is another name for what is known in other programming languages as a subroutine, function, or procedure.

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

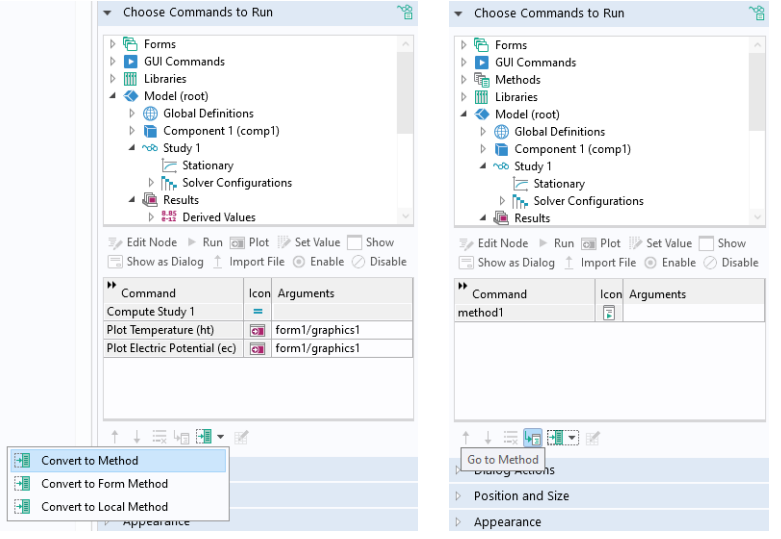
To create a new method, right-click the **Methods** node in the application tree and select **New Method**. You can also click **New Method** in the ribbon **Home** tab. In the **New Method** dialog box you can change the name of the method.



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. By first clicking a form node you also have the option of creating a **Form Method** which is local to a form.

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.

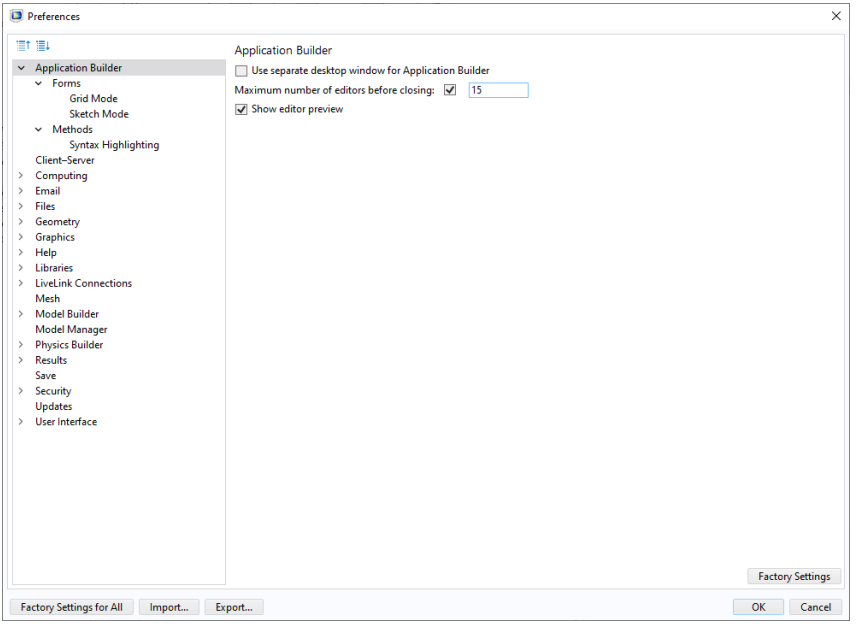


If a method already exists, say with the name `method1`, then you can open the Method Editor in any of these ways:

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



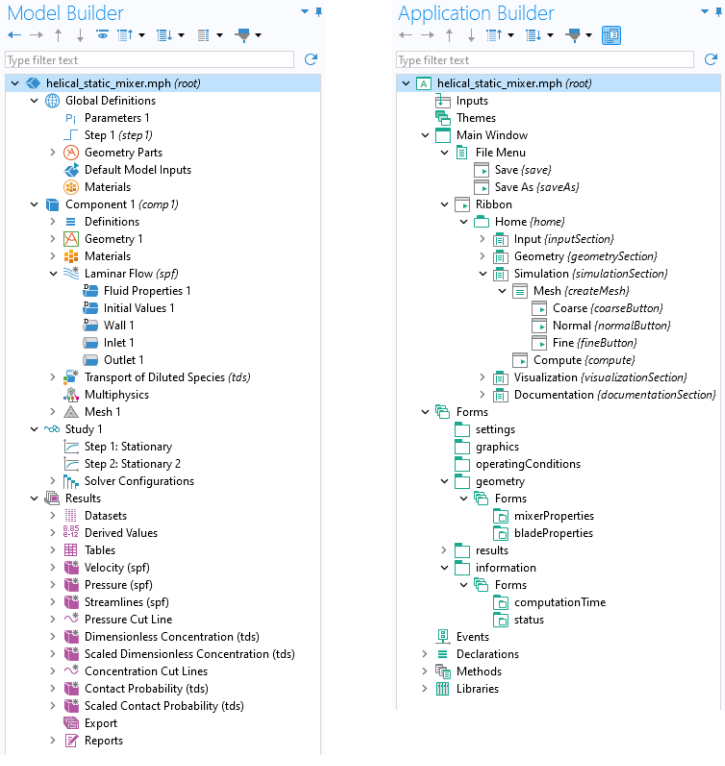
You can configure the COMSOL Desktop environment so that the Application Builder is displayed in a separate desktop window. Select the check box **Use separate desktop window for Application Builder**.

You can use the keyboard shortcuts Ctrl+Shift+M and Ctrl+Shift+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 three important tools for creating applications: The Form Editor, the Main Window Editor, and the Method Editor. The Form Editor includes drag-and-drop capabilities for user interface components such as input fields, graphics objects, and buttons. The Main Window Editor lets you design a menu bar or a ribbon. 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.



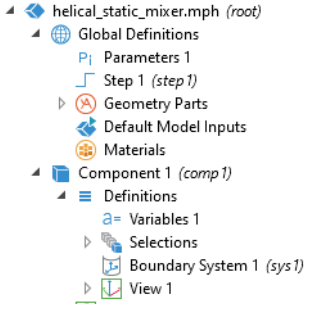
When creating an application, you typically start from an existing model. However, you can just as well build an application user interface and the underlying model simultaneously. You can easily, at any time, switch between the Model Builder and Application Builder. The model part of an application, 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 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 65 and 97.
- By using the 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 47 and 65.
- 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 65, 108, and 191.
- By using the Record Code and Record Method tools, 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 195.

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:

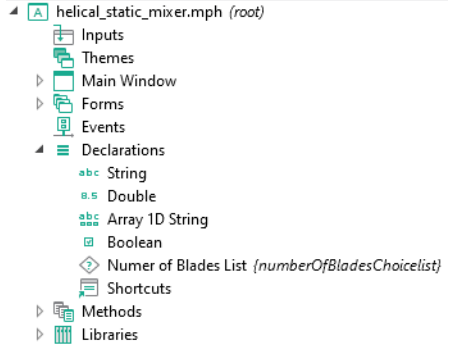
- Parameterizing geometric dimensions

- 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 160 and 353. 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 clients for Windows® or Android®.

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 45, describes your rights to publish applications.

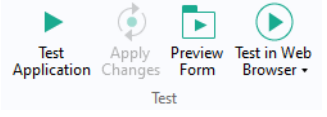
Running Applications in COMSOL Multiphysics

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

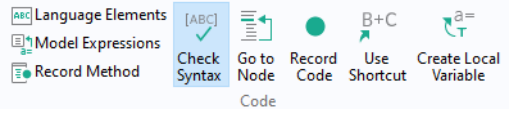
Test Application is used for quick tests. It opens a separate window with the application user interface while keeping the Application Builder desktop environment running. The figure below shows the **Test** section as it appears in the **Form** tab of the ribbon.



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.

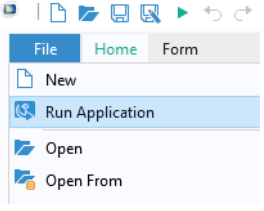
When **Test Application** is used, all methods are automatically compiled with the built-in Java® compiler. Any syntax errors will generate error messages and the process of testing the application will be stopped. To check for syntax errors before testing an application, click the **Check Syntax** button 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 **Method Errors and Warnings** window in the Method Editor. For more information, see "The Method Editor" on page 185.

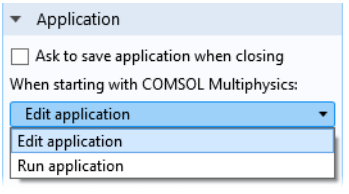
Running an Application

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.



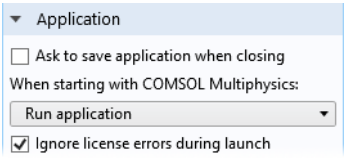
The option **Edit application** will open the application in the Application Builder.

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

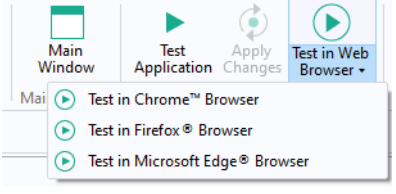
In the **Settings** window for the application tree root node, the check box **Ignore license errors during launch** is used to control the behavior with respect to licensed products when running applications.



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 as a standalone app, compiled using COMSOL Compiler, or 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	COMSOL Client for Android®
COMSOL Server	Web Browser
N/A	Executable file compiled with COMSOL Compiler

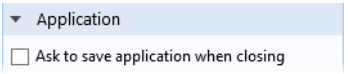
The Server column represents the software components that perform the CPU-heavy computations. The Client column represents the software tool or component used to present the application user interface. In the case of executable files, all computations are done locally. For more information on compiled applications, see “Compiling and Running Standalone Applications” on page 38.

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**

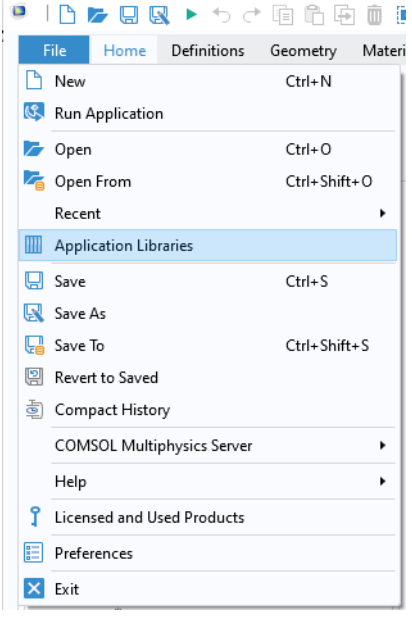
in which you may select the check box **Ask to save application when closing**, as shown in the figure below.



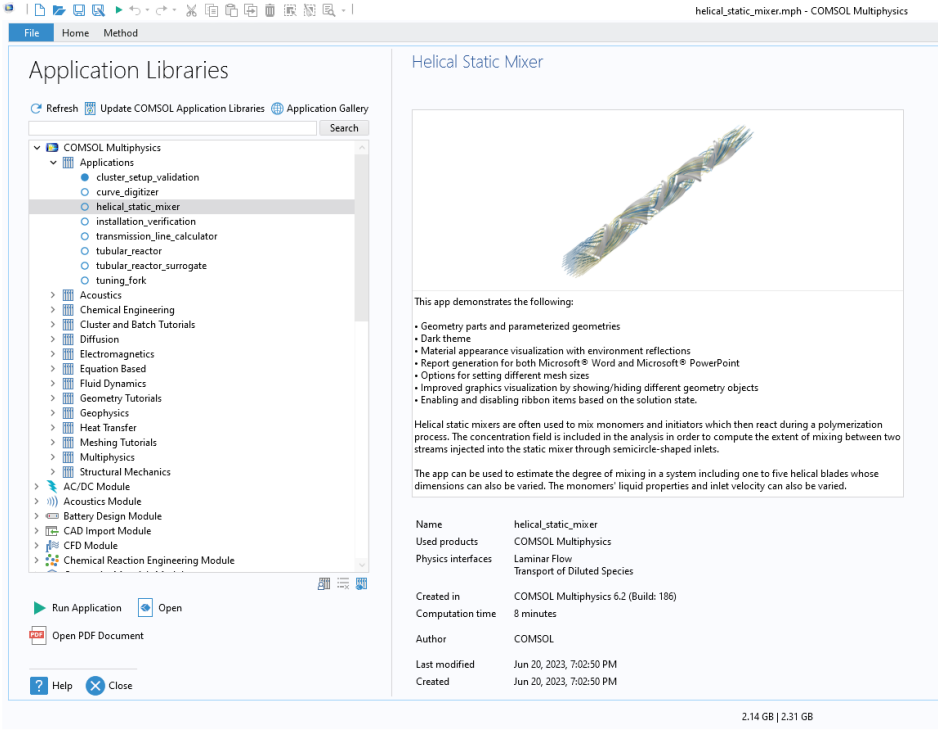
As an alternative, you can add a button or menu item with a command to save the application. For more information, see page 144.

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**.

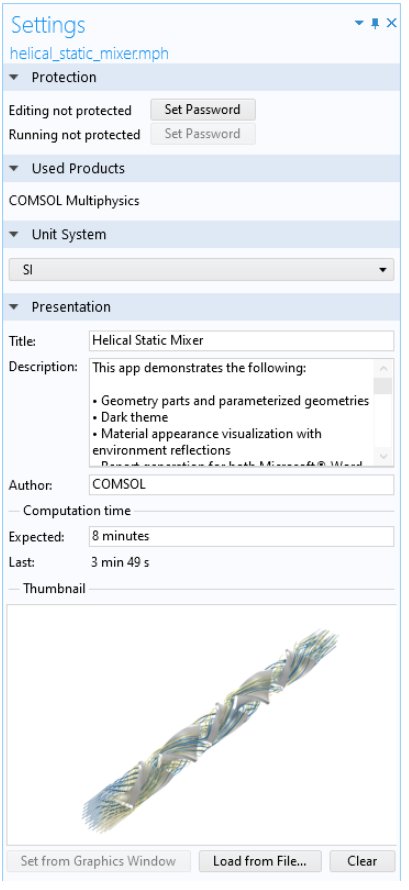
The applications in the Application Libraries are continuously improved and updated. You can update the Application Libraries by clicking Update COMSOL Application Libraries.

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.

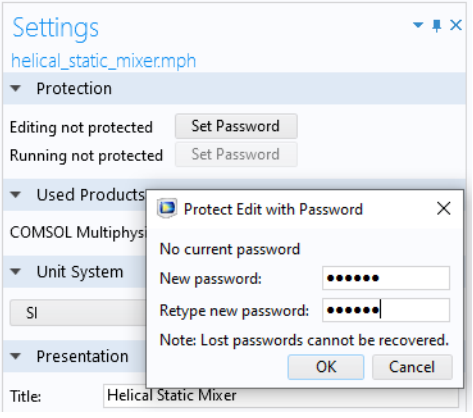


The **Set from Graphics Window** option automatically creates two thumbnail images:

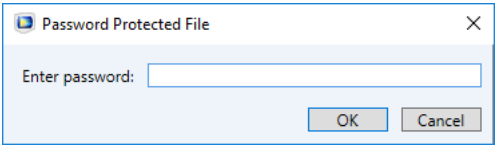
- An image of size 280-by-210 pixels shown in the **Settings** window of the application tree root node and in the Application Libraries.
- An image of size 1024-by-768 used as the default title page image in reports and in the Application Libraries in COMSOL Server.

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. Note that for add-ins the password is set in the **Add-in Definition** window. For more information, see "Creating Add-Ins" on page 226.

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

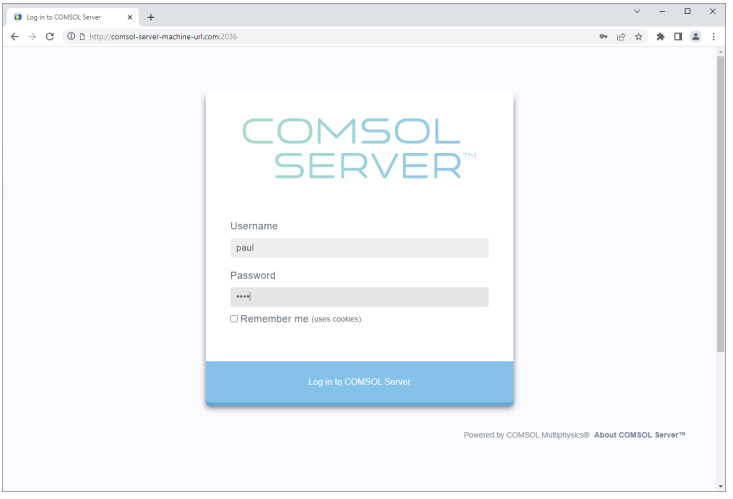
COMSOL applications can be run by connecting to COMSOL Server from a web browser or a COMSOL Client for Windows®. The COMSOL Client for Windows® allows a user to run applications that require a LiveLink™ product for CAD, as described in Running Applications in the COMSOL Client.

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.

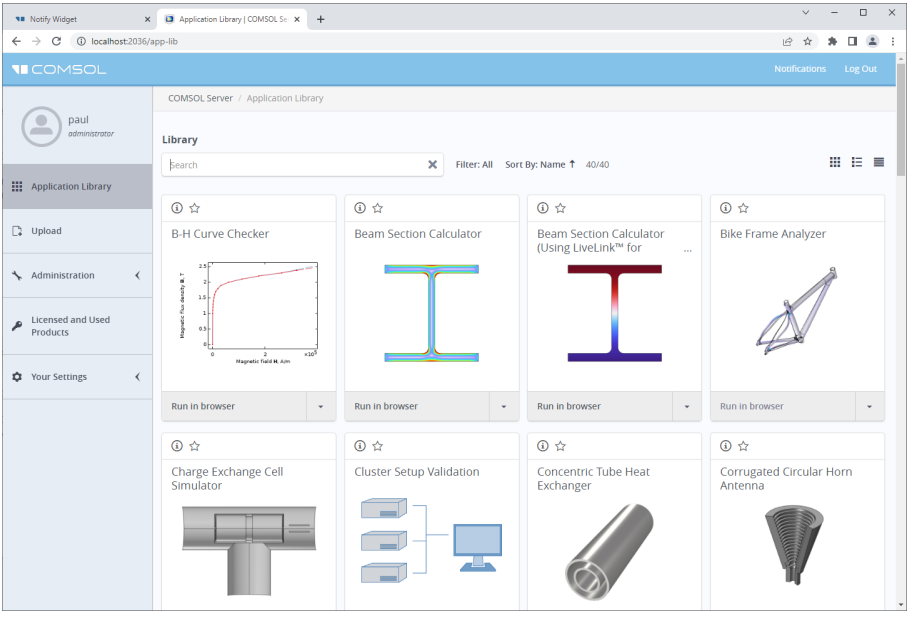
Running Applications in a Web Browser

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

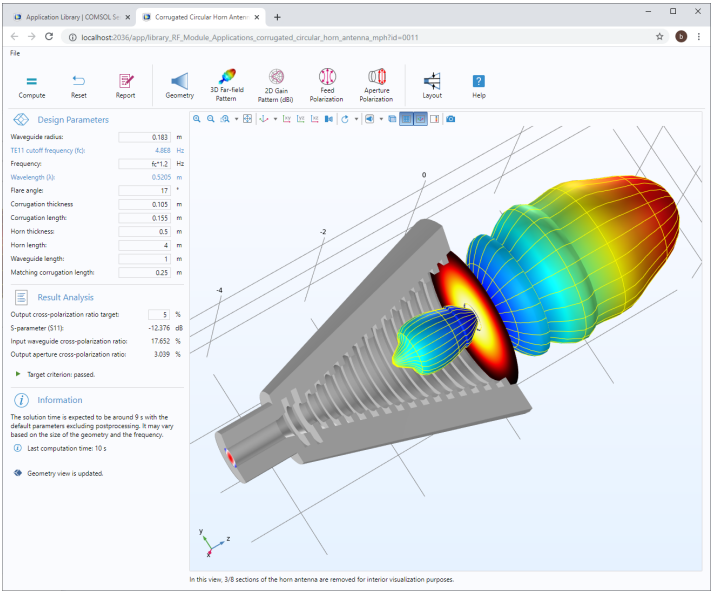
username and password to log in. If you are running COMSOL Server locally, the address field will typically be `localhost:2036`.



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 114. 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

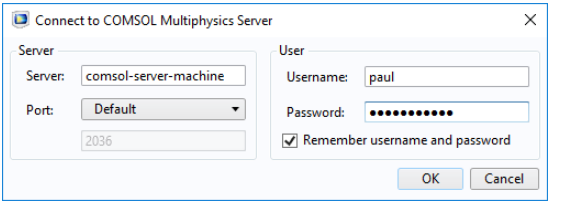
As an alternative to using a web browser for running applications, the COMSOL Client for Windows® can be used to connect to COMSOL Server for running applications natively in the Windows® operating system. This typically gives better graphics performance and supports more sophisticated graphics rendering 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®.

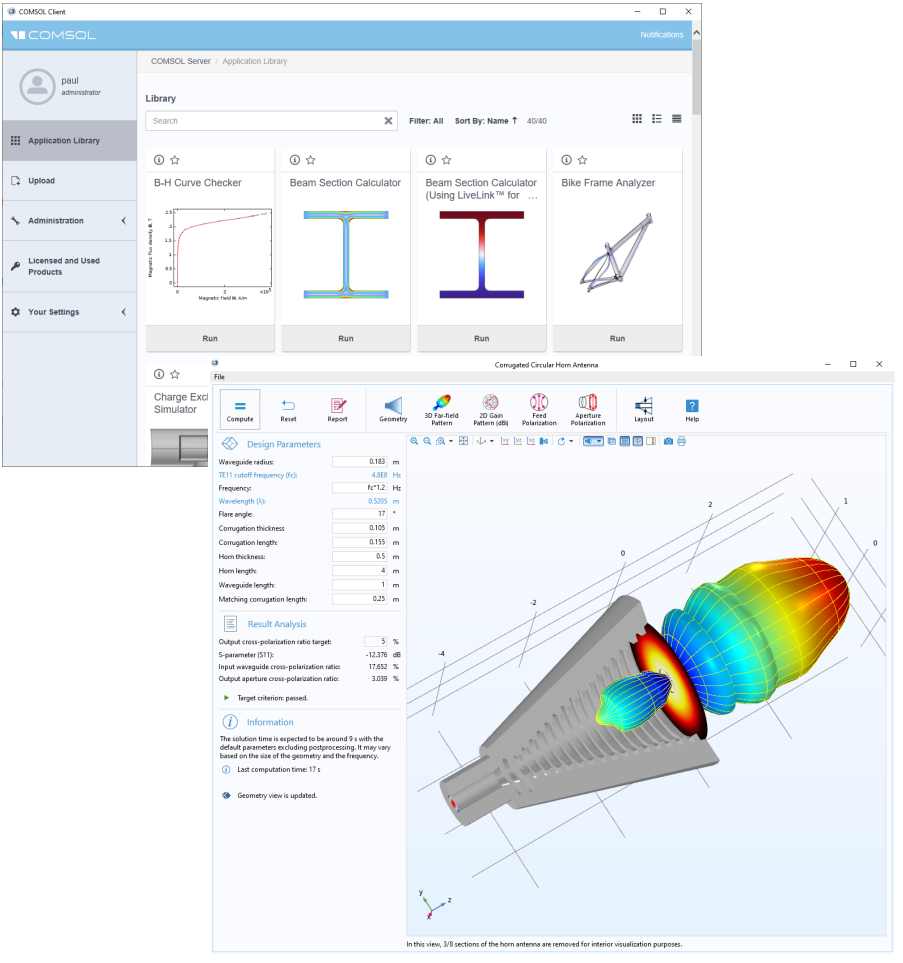
  If you try to run an application with the COMSOL Client in this way, but it is not yet installed, you will be prompted to download and install it.

- 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.

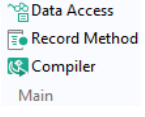
Configuring a Study Node for Cluster Sweep or Cluster Computing

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 nodes 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 *Introduction to COMSOL Multiphysics* and *COMSOL Multiphysics Reference Manual* books. 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.

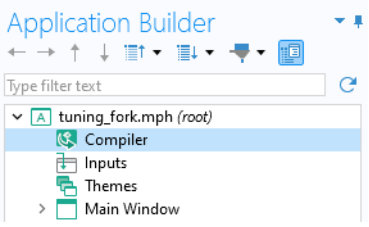
For more information on COMSOL Server, see the *COMSOL Server Manual* available with a COMSOL Server installation or from https://www.comsol.com/documentation/COMSOL_ServerManual.pdf.

Compiling and Running Standalone Applications

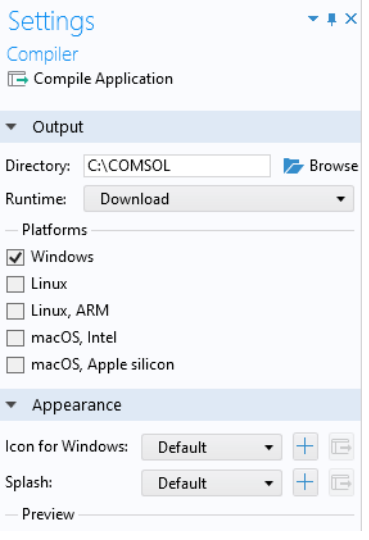
If you have a license of COMSOL Compiler™, there will be a **Compiler** button in the ribbon section **Main**, as shown below.



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 how the COMSOL Runtime environment will be acquired from the compiled application. 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 **Embed** 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.

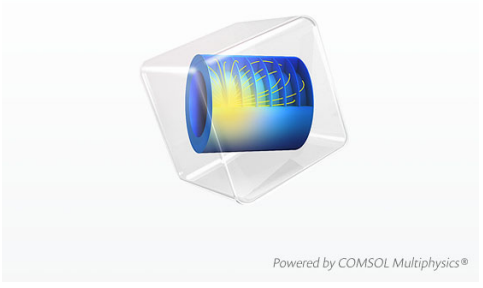
As a next step, you can, for example, right-click the EXE file and create a shortcut that you then place on the Windows® desktop. Note that the `.exe` file extension might not be shown, depending on the operating system settings.

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

Running Compiled Applications

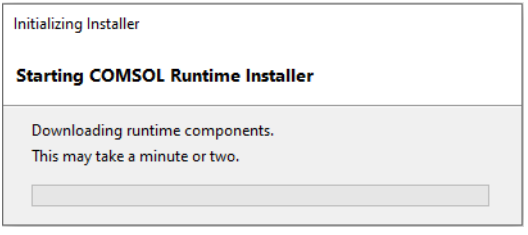
When running a compiled application, for example, by double-clicking the `.exe` file in the Windows® operating system, a splash screen is shown and the

application will start. If the application has the **Splash** option set to **Default**, then a neutral-looking built-in splash screen will be shown.

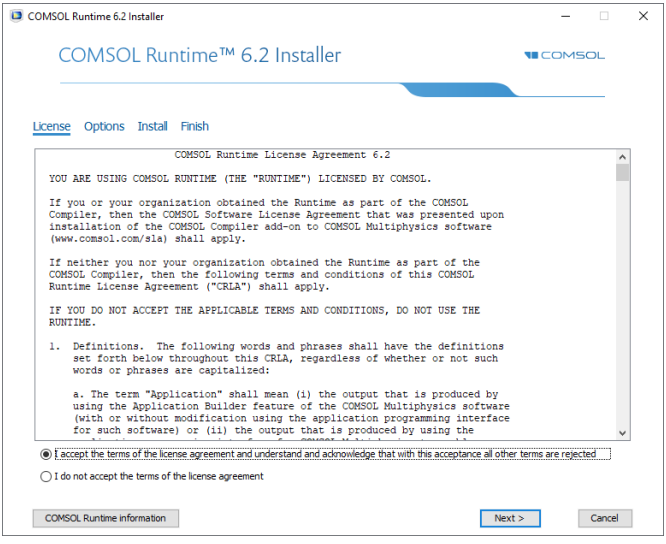


It is recommended that you replace this with your own splash screen.

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.

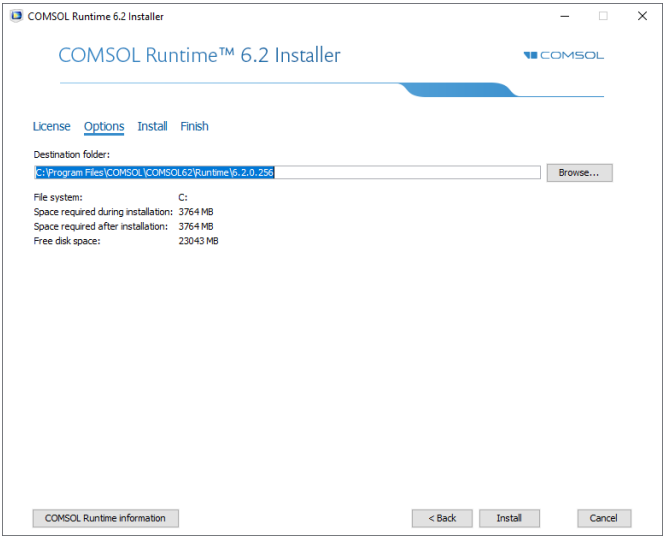


After a short moment, the COMSOL Runtime Installer window is displayed, as shown below.



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).

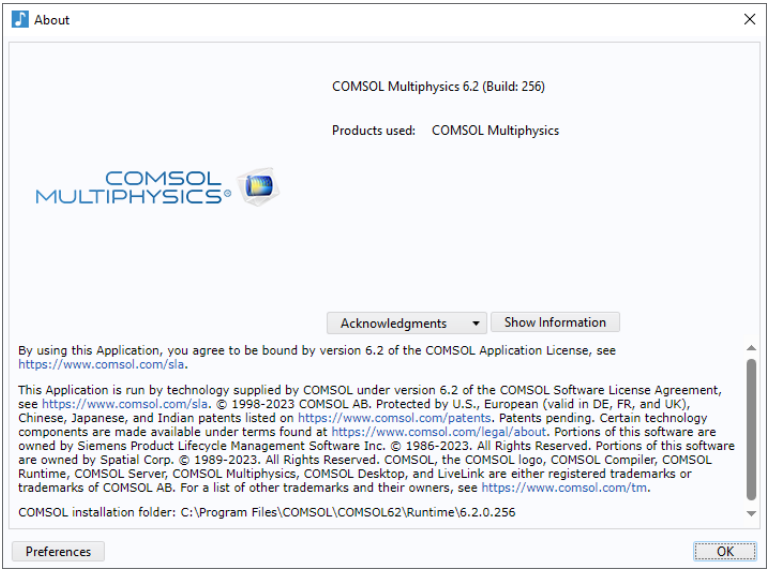
Click **Next** to select the location of the COMSOL Runtime files and the click **Install**, as shown in the figure below.



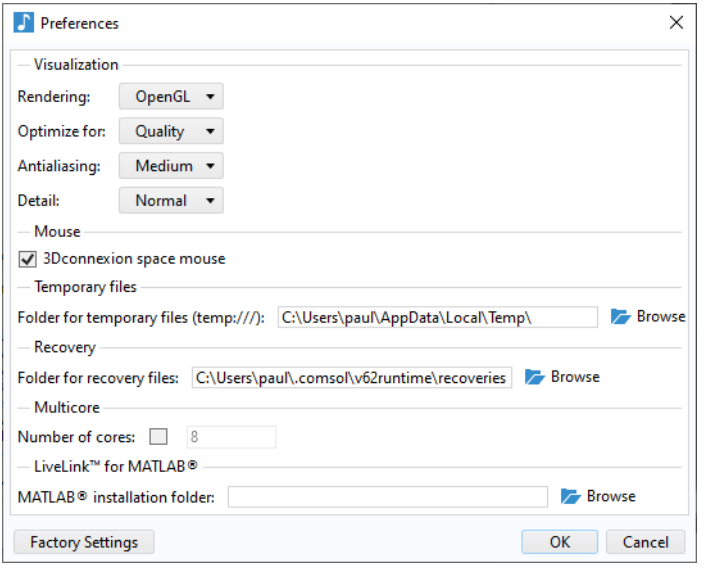
The installation takes a few minutes and, when finished, the installer will prompt the user to start the application.

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 140. 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.

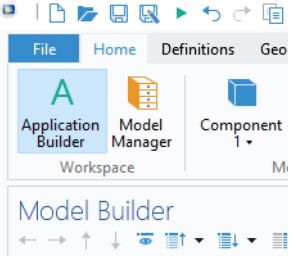
If you wish to apply the Application License to applications that you create, the Application License contains instructions on how to do so. The Application License also addresses how you can use terms that you choose for modifications you make to applications available in the Application Libraries,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

Starting From a COMSOL Multiphysics Model

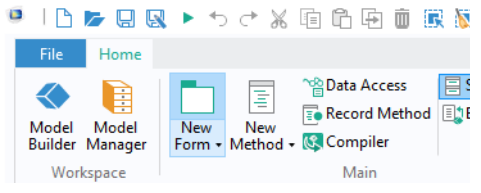
If you do not have a model already loaded to the COMSOL Desktop environment, select **File > Open** to select an MPH-file from your file system or select a file from the Application Libraries. Note that, in the Application Libraries, the files in the **Applications** folders are ready-to-use applications.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



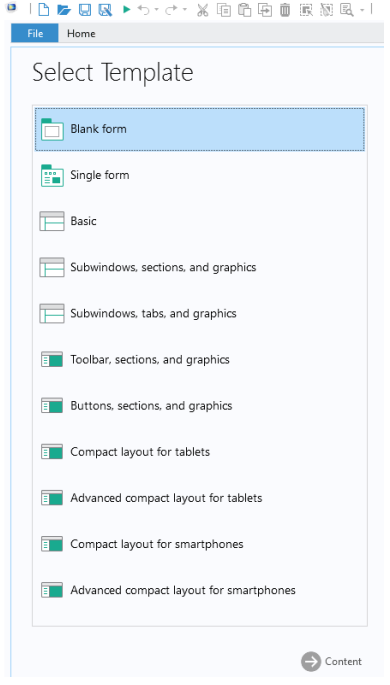
Creating a New Form Using Templates and the Form Wizard

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



The Form Wizard assists you with adding the most common user interface components, so-called form objects, to the first draft of your application.

In the Form Wizard, the first page is the **Select Template** page.



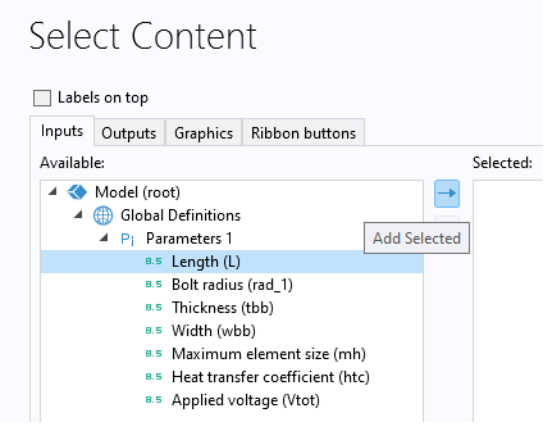
The different templates listed here will help you quickly create an organized application with different levels of sophistication and user-interface layouts for desktop, table, and smartphone use.

For this example, you can load the `busbar.mph` model from the Application Libraries at **COMSOL Multiphysics > Multiphysics**. This is one of the models used in the *Introduction to COMSOL Multiphysics* manual.

Select the **Basic** layout template and click **Content**. The **Select Content** page has four tabs:

- **Inputs**
- **Outputs**

- **Graphics**
- **Ribbon buttons**



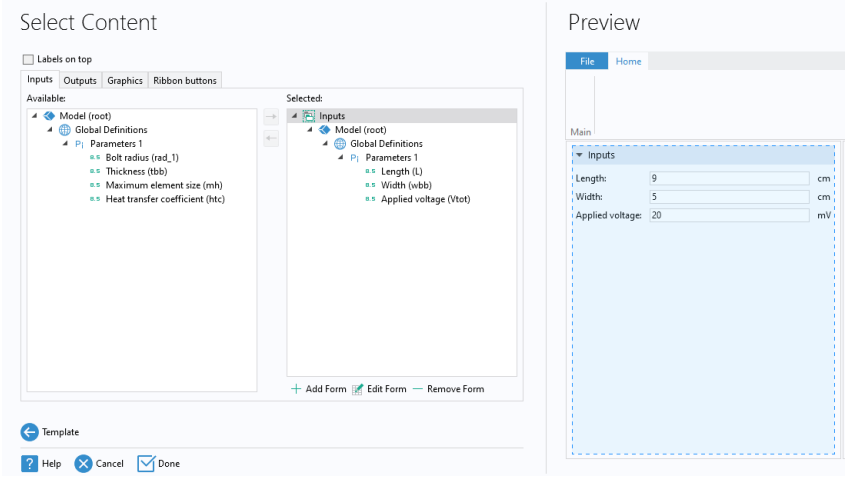
Double-click a node or click the **Add Selected** button to move a node from the **Available** area to the **Selected** area. The selected nodes will become form objects in the application and a preview of the form will be shown in the **Preview** area to the right.

The size as well as other settings for form objects can be edited after exiting the wizard. You can also choose to exit the wizard at this stage by clicking **Done**, and then manually add form objects.

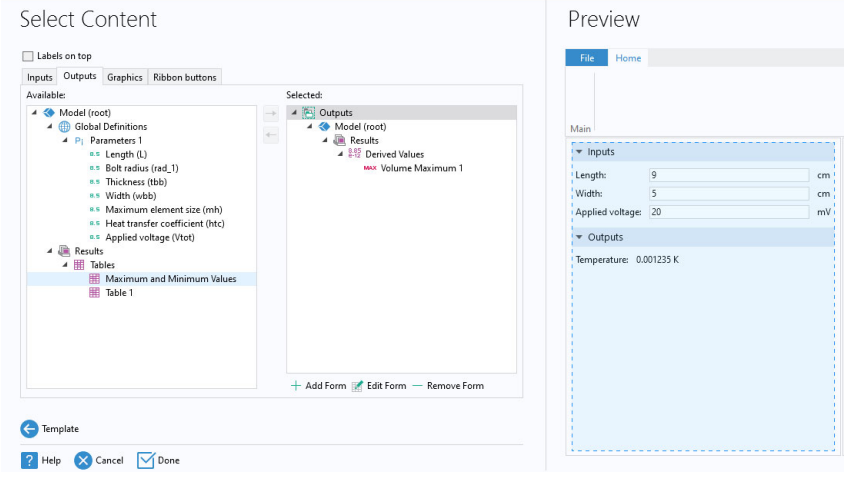
The Inputs and Outputs Tabs

The **Inputs** and **Outputs** tab display 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 108). 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 the **Data Access** feature.

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



In the figure below, a **Derived Values** node for the maximum temperature has been selected from the **Outputs** tab to serve as a data display object.

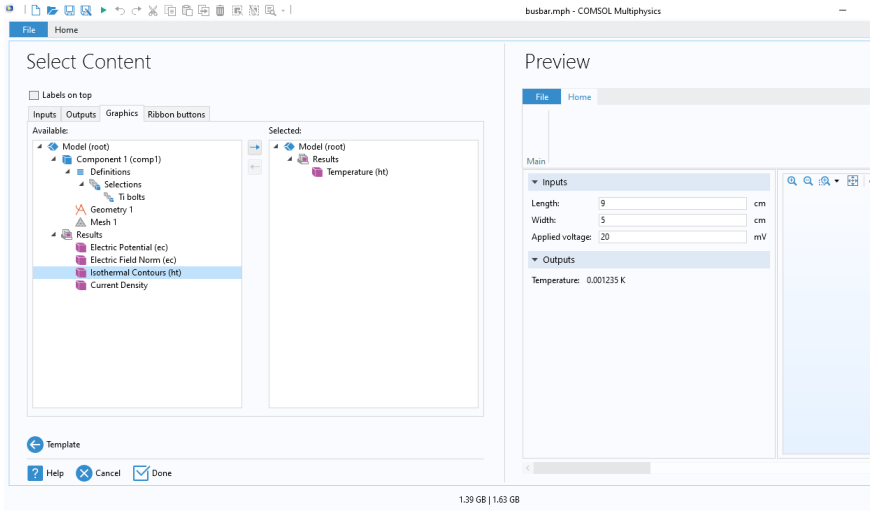


After exiting the wizard, you can edit the size and font color as well as other settings for input fields and data display objects.

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, a **Temperature** plot

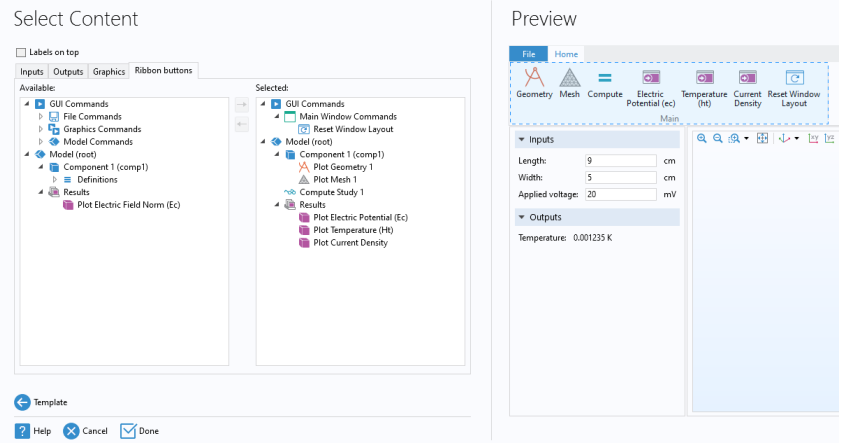
node has been selected. When using the **Basic** layout template, this selection determines the default plot shown when the app is started.



The Ribbon Buttons Tab

The **Ribbon buttons** tab displays the model and application tree nodes that can be run by clicking a button in the ribbon 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**. Note that in this example no **Forms** or **Methods** are available yet.

In the figure below, buttons have been added corresponding to the actions: **Plot Geometry**, **Plot Mesh**, **Compute Study**, **Plot Electric Potential**, **Plot Temperature**, **Plot Current Density**, **and Reset Window Layout**.



The **Reset Window Layout** option is available under **GUI Commands > Main Window Commands > Reset Window Layout**. The application consists of two subwindows, one for the inputs and outputs and one for the graphics. The **Reset Window Layout** button will reset the two subwindows in the application to their original size. The **Subwindows** templates are similar to the **Basic** template but additionally enable you to detach, move around, and dock the subwindows. In this case, the **Reset Window Layout** operation will rearrange all subwindows to their original position and size.

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

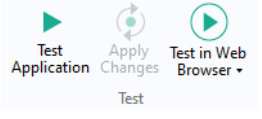
For an example of using the **Single form** template, see the *Introduction to COMSOL Multiphysics* manual.

Exiting the Wizard

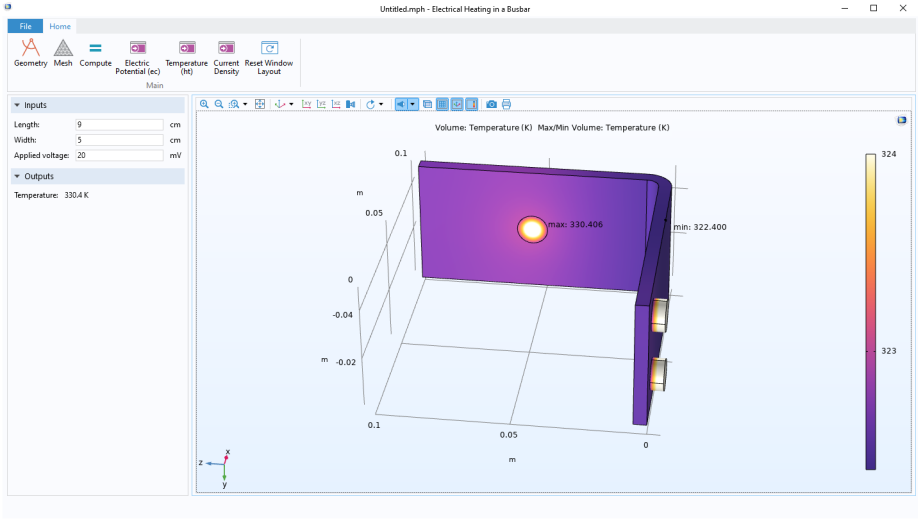
Click **Done** to exit the wizard. This automatically takes you to the Form Editor.

Testing the Application

You can now click **Test Application** in the **Test** section of the ribbon.



The figure below shows the running application.

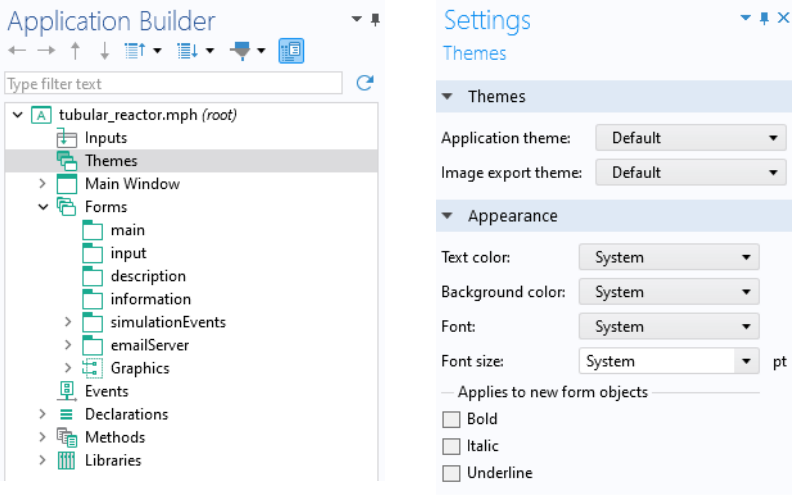


Saving an Application

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.

Themes

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



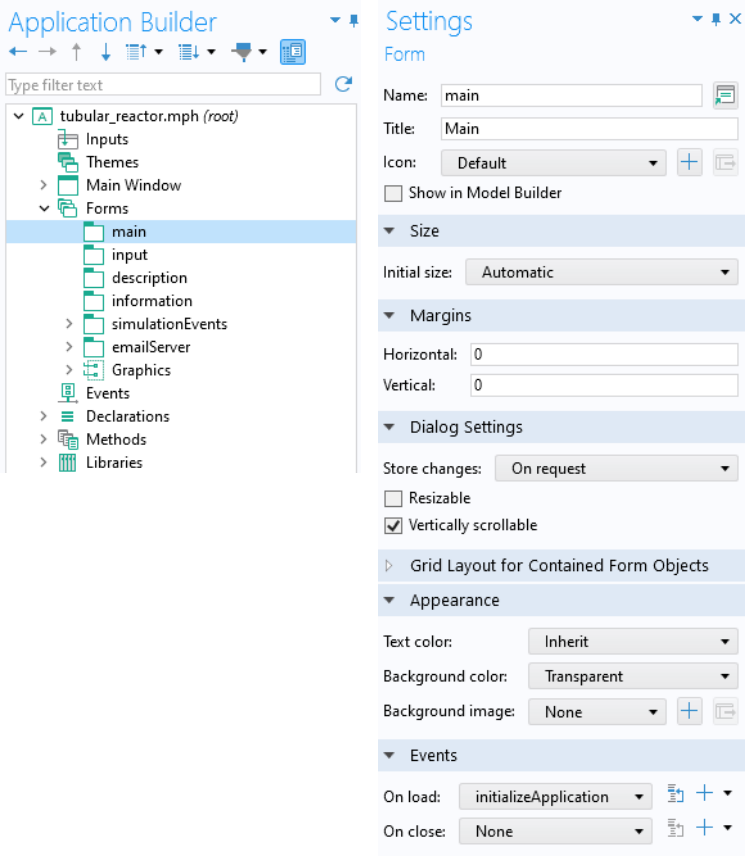
The default is that all new forms and new form objects inherit these settings when applicable.

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 Individual Form Settings Windows

The figure below shows the application tree node and **Settings** window for a form.

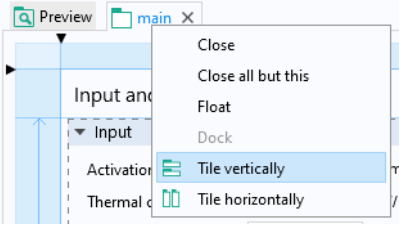


Each form has its own **Settings** window with settings for:

- **Name** used to reference the form in other form objects and methods.
- Form **Title** that is used in applications with several forms.

- **Icon** shown in the upper-left corner of a dialog box.
- **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 76.
- Choices of whether the form should be **Resizable** or not and **Vertically scrollable** or not when used as a dialog box.
- 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**.)

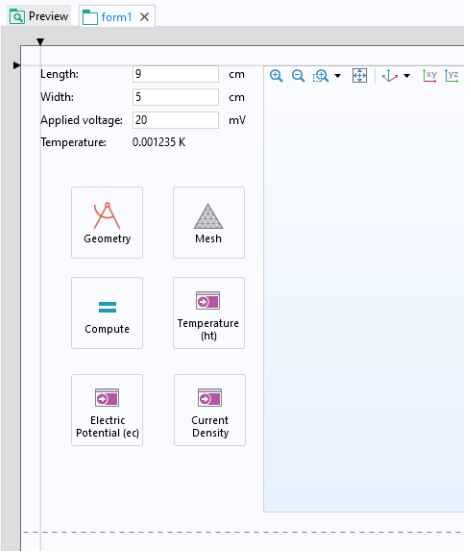
Double-click a form node to open its window in the Form Editor. Alternatively, you can right-click a form node and select **Edit**. Right-click a form window tab to see its context menu with options for closing, floating, and tiling form windows.



Sketch and Grid Layout Modes

When adding a form using the **Blank** or **Single form** layout templates, the Application Builder defaults to sketch layout mode, which lets you use fixed object positions and size. The other layout templates use the grid layout mode. Sketch layout mode is useful when you want control over the absolute positioning of form objects, such as when creating dialog boxes. Many, but not all, of 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 114.

If you use the **Single form** template and follow the same steps as for the earlier example using the **Basic** template, then the Form Editor will result in an form as shown in the figure below, after dragging to reorganize the buttons.



In this case, no ribbon is created and you can freely position form objects, such as buttons and edit fields, by dragging them. This is beneficial in some cases, however, the size of the graphics object will be fixed when using the **Single form** layout template. This is because for this template the sketch layout mode is used and automatic resizing of an app is only supported with the grid layout mode. To learn how to create resizable graphics objects, see “Automatic Resizing of Graphics Objects” on page 125.

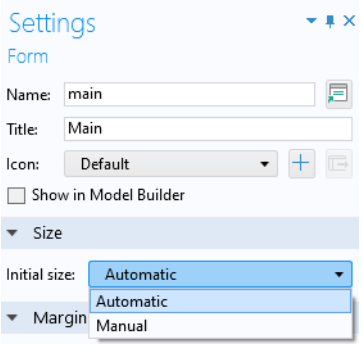
Note that any type of application that you create using templates can also be built using manual steps by starting from a **Blank form** or **Single form** template, however, using templates accelerates this process.

Initial Size of a Form

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

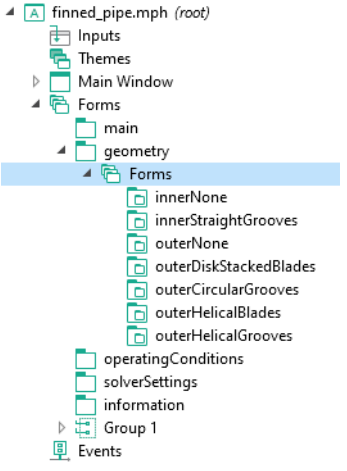
- **Manual** lets you enter the pixel size for the width and height.
- **Automatic** determines the size based on the form objects that the form contains. If you are using grid layout mode and there are columns or rows set to **Grow**, then the size is not defined by the form objects. In this case, the 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 117.



Local Forms

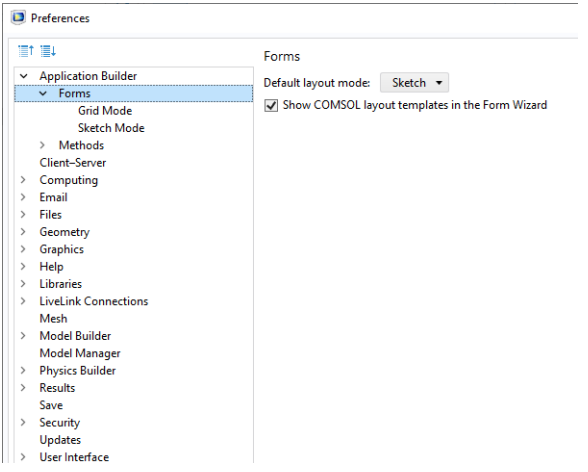
Forms can be local to other forms, which enables you to create a better structure when developing your applications. For instance, a complicated global form made up of many different subforms can have the auxiliary forms as local forms, displayed as children in the application tree.



You can add a local form by, for example, right-clicking a global form and selecting **Local Form**. A global form always appears directly under the **Forms** node in the application tree.

Form Editor Preferences

To access **Preferences** for the Form Editor, choose **Preferences** from the **File menu** and select the **Forms** node in the tree.



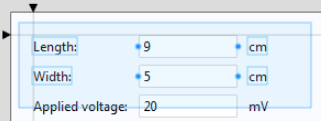
The **Forms** node and its child nodes, **Grid Mode** and **Sketch Mode**, includes settings for changing the defaults for layout mode, margins, sketch grid, and layout templates.

Form Objects

Positioning Form Objects

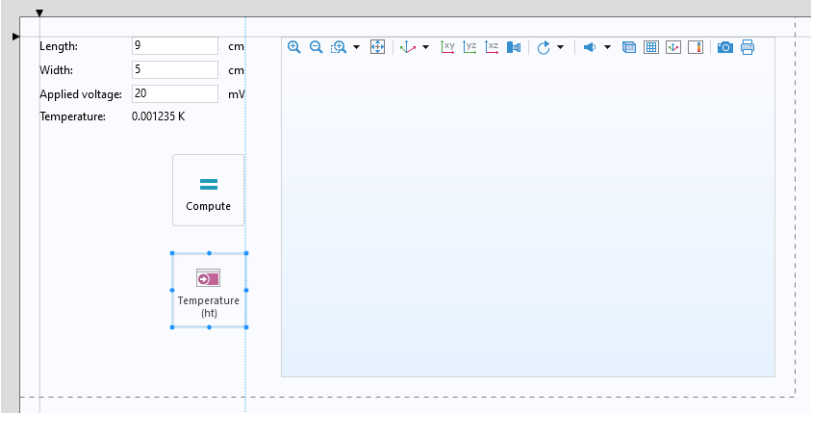
You can easily change the positioning of form objects such as input fields, graphics objects, and buttons in one of the following ways (which methods are applicable depends on which layout mode you are using):

- 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:

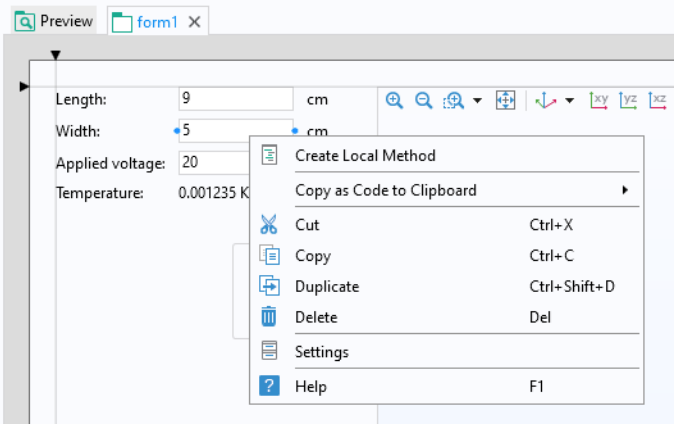
- Click an object to select it.
- Hold and drag one of the handles, shown as blue dots, of the highlighted blue frame. If there are no handles, this type of form object cannot be resized.

Note that some form objects that cannot be resized in grid mode can be resized in sketch mode.

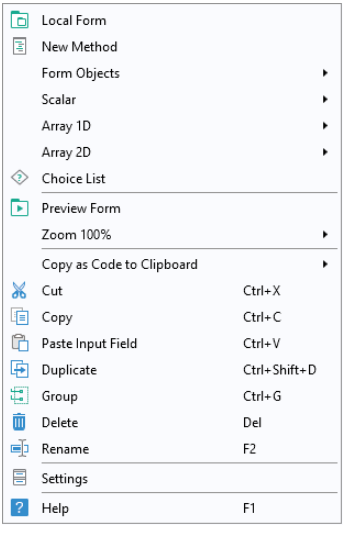
Copying, Pasting, Duplicating, and Deleting an Object

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.



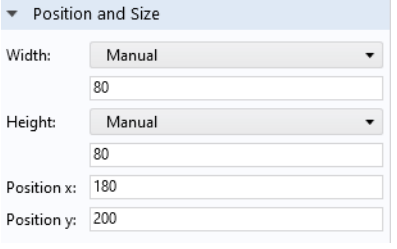
To paste an already copied object, right-click an empty area in the form and right-click again. Depending on the copied object, a **Paste** menu option will be shown. In the figure below, an **Input Field** has previously been copied and as a result, a **Paste Input Field** option is shown.



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.

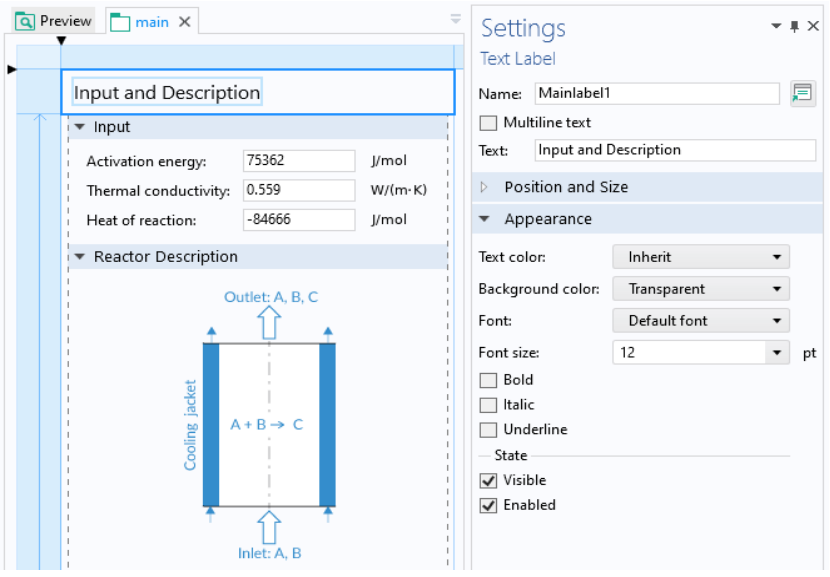


The Position and Size section will have different options depending on the type of form object. For grid layout mode, there are additional settings for the position of the object with respect to rows and columns. For details, see "Sketch and Grid Layout" on page 114.

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.



By using grid layout mode (see "Sketch and Grid Layout" on page 114) 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.

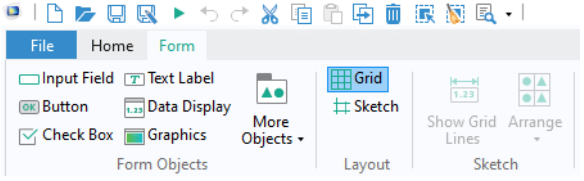
The Name of a Form Object

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.

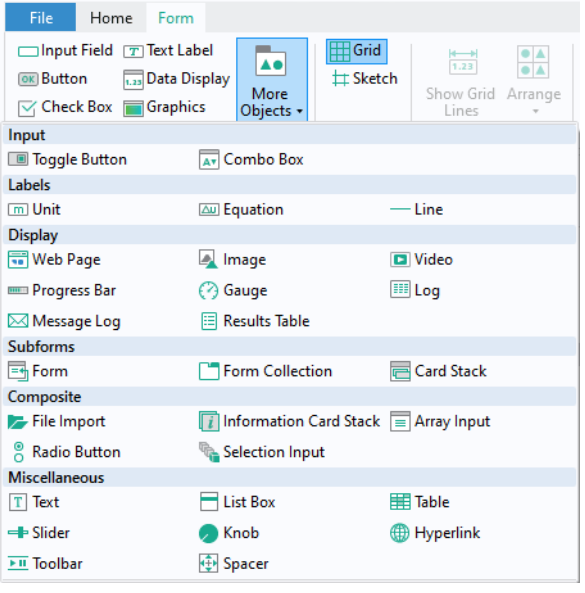
Inserting Form Objects

You can insert form objects in addition to those created by the Form Wizard. In the **Form** ribbon tab, in the **Form Objects** section, you can quickly select some of

the most common form objects: **Input Field**, **Button**, **Check Box**, **Text Label**, **Data Display**, and **Graphics**.



Additional form objects are available from the **More Objects** menu button.



The remainder of this section, The Form Editor, only describes the types of form objects that are added by the Form Wizard. The form objects added by using the wizard may include:

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

However, when using **Data Access** (see page 108), 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 237.

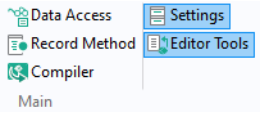
Events and Actions Associated with Forms and Form Objects

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**), change of the value of a variable (**On data change**), and focus gained.

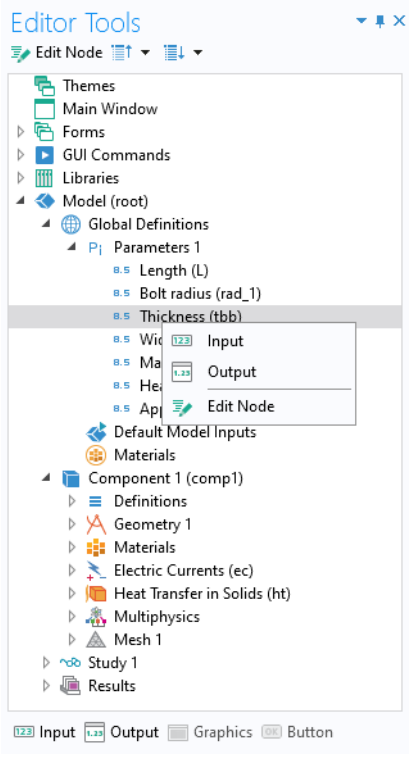
Using Ctrl+Alt+click on a form object opens any associated method in the Method Editor. If there is no method associated with the form object, a new local method will be created, associated with the form object, and opened in the Method Editor. If the form object has an associated command sequence, this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



You can right-click the nodes in the editor tree to add the same set of form objects available with the 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 108) 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.

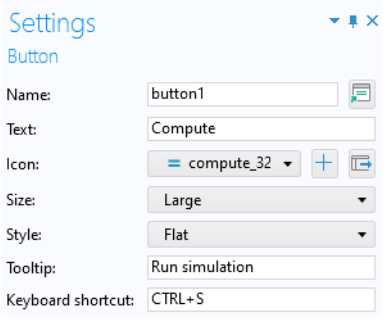
The Editor Tools window is also an important tool when working with the Method Editor. In the Method Editor, it is used to generate code associated with the nodes of the editor tree. For more information, see "Editor Tools in the Method Editor" on page 191.

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**, **Raised**, or **Outlined**.

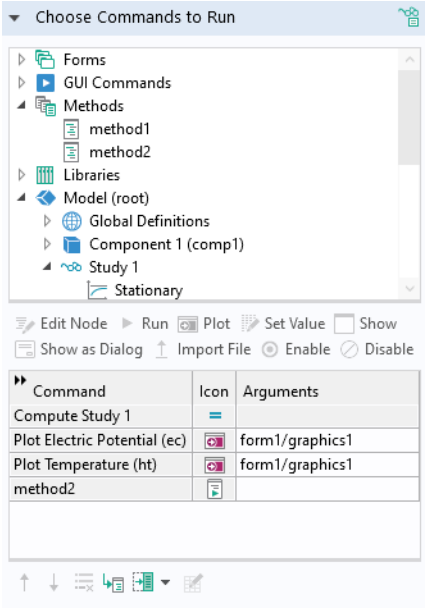
- Add a **Tooltip** with text that is shown when hovering over the button.
- Add a **Keyboard shortcut** by clicking the input field and entering a combination of the modifier keys Shift, Ctrl, and Alt together with another keyboard key. Alt must be accompanied by at least one additional modifier.



Note that the settings for a ribbon button, defined as **Item** in a **Ribbon Section**, are very similar to that of a button.

Choosing Commands to Run

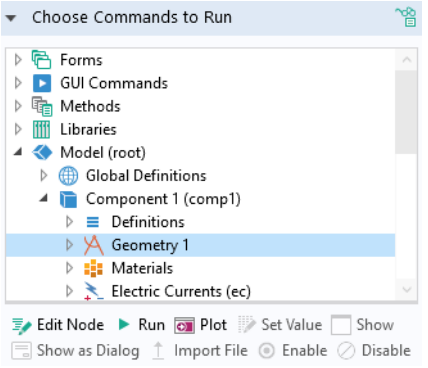
The section **Choose Commands to Run** lets you control the action associated with a button-click event. The figure below shows the **Settings** window for a button that triggers a sequence of four commands.



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 86, "The Main Window Editor" on page 138, "Table" on page 312, and "Toolbar" on page 323.

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.



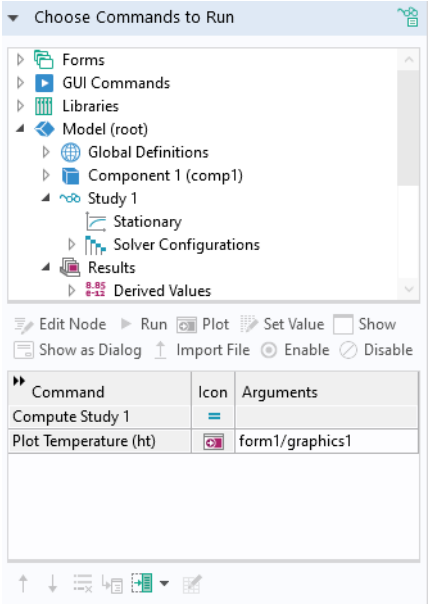
You do not need to precede a **Plot Geometry** command with a **Build Geometry** command (that you get by clicking **Run**). The **Plot Geometry** command will first build and then plot the geometry. In a similar way, the **Plot Mesh** command will first build and then plot the mesh.

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**

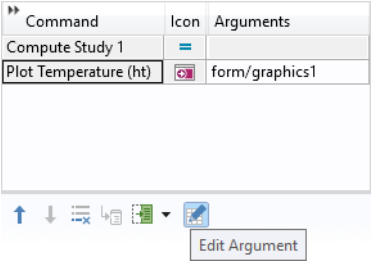
Some commands, such as the various plot commands, require an argument. The argument to a plot command, for example, defines which of the different graphics objects the plot should be rendered in.

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



The **Plot Temperature** command has one argument, `form1/graphics1`.

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

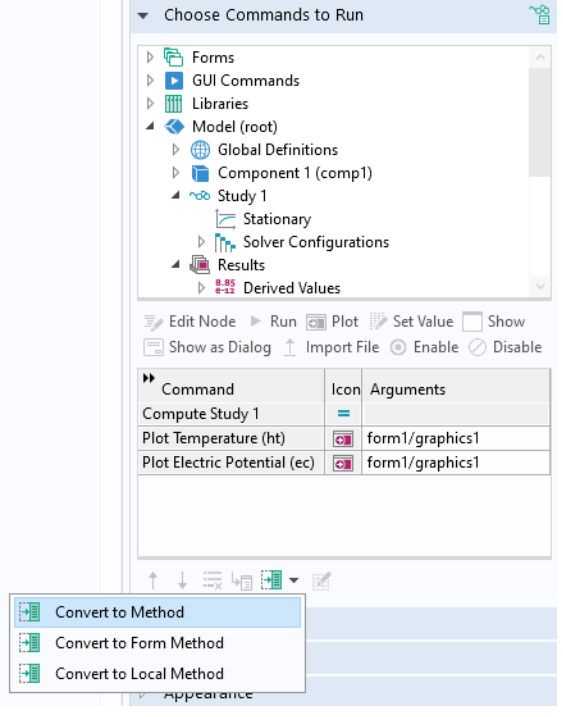


To reference graphics objects in a specific form, the following syntax is used: `form1/graphics2`, `form3/graphics1`, etc. If a specific form is not specified, for example, `graphics1`, then the form where the button is located is used.

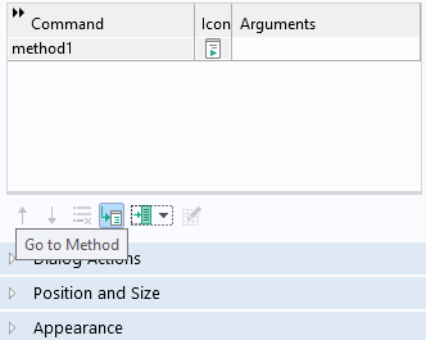
To control the order and contents of the sequence of commands, use the **Move Up**, **Move Down**, and **Delete** buttons located below the command sequence table.

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**.

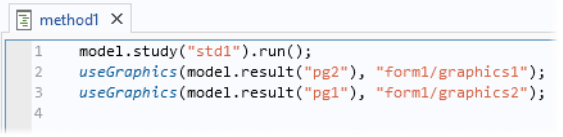


Open the new method by clicking **Go to Method**.



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 commend sequence, as shown in the figure below.



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");
```

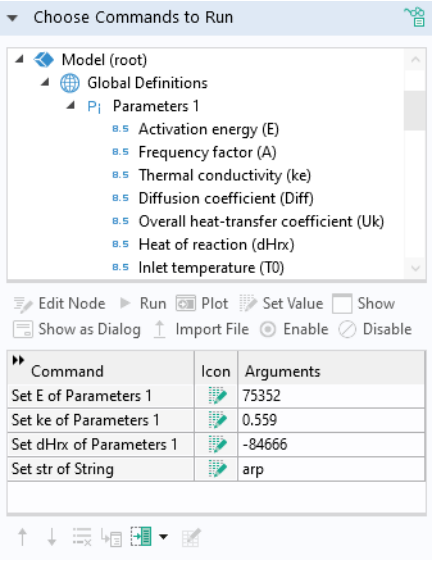
use the built-in method `useGraphics` to display plots corresponding to plot groups `pg1` and `pg2`, respectively. In this example, the plots are displayed in two different graphics objects, `graphics1` and `graphics2`, respectively.

For more information on methods, see “The Method Editor” on page 185.

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

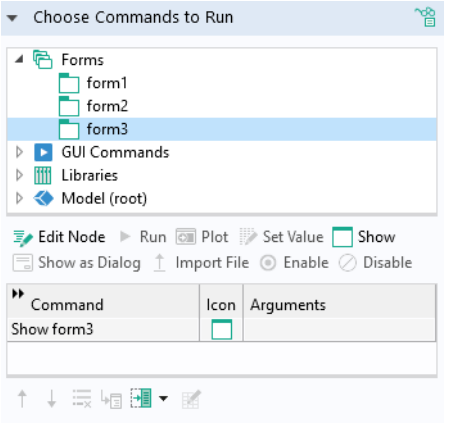


To learn how to perform the same sequence of operations from a method, select **Convert to Method** under the command table.

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.

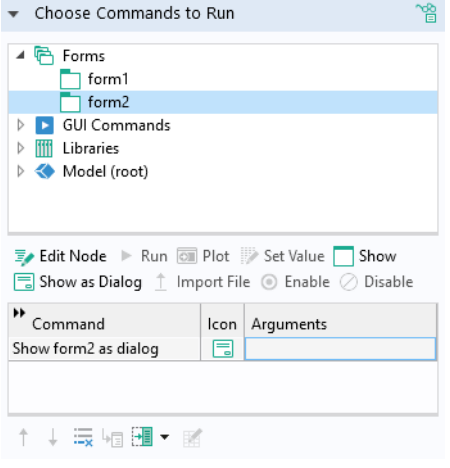
In the section **Choose Commands to Run**, you can select the **Show** command. The figure below shows the command sequence for a button with a command **Show form3**.



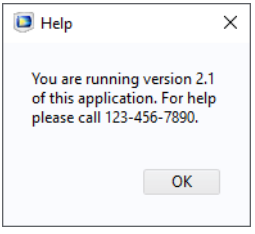
This command will leave the form associated with the button and make the specified form visible to the user.

Showing a Form as a Dialog Box

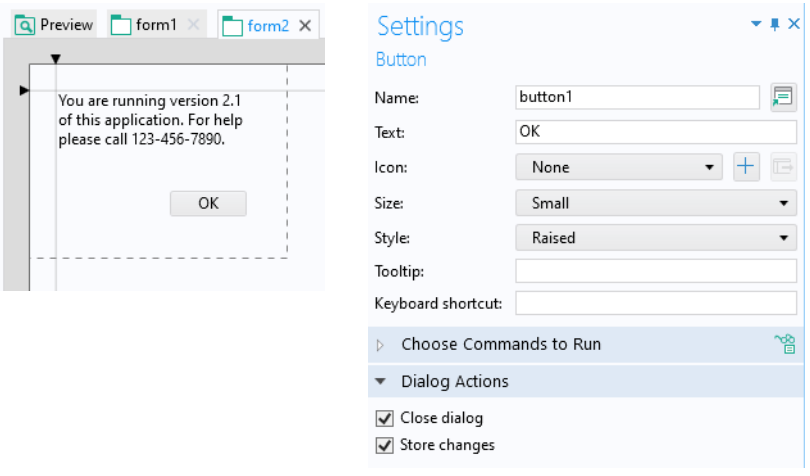
In order to use the **Show as Dialog** command, begin with the **Choose Commands to Run** section and select the form that you would like to show. The figure below shows an example of the settings for a button with the command **Show form2 as dialog**.



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**

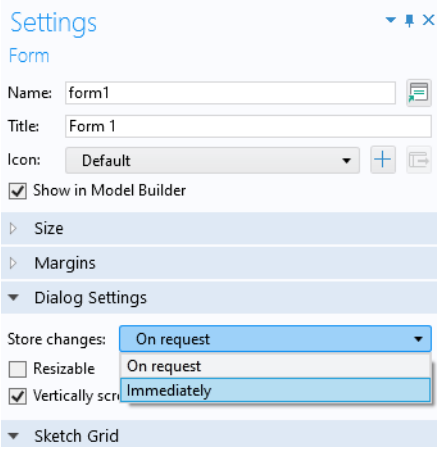
In the example above, the **Close dialog** check box is selected. This ensures that the **form2** window is closed when the **OK** button is clicked. Since **form2** does not have any user inputs, there is no need to select the **Store changes** check box.

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.

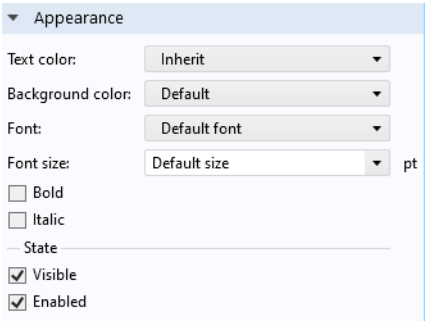


When the **Store changes** option **On request** is selected, the variables that have been changed by the user in the dialog box will not be updated until the OK button (or similar) in the dialog box has been clicked. This requires that the check box **Store changes** is selected, in the **Settings**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

When **Vertically scrollable** is cleared, the form will never get any vertical scrollbar. Instead, the scrollbar will appear on the form objects inside the form, if possible. To obtain a satisfactory result, the form has to be created in grid mode. The form object which should get the scrollbar must be in a row with setting **Grow Row** and its alignment set to **Fill Vertically**.

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 359 and the *Application Programming Guide*.

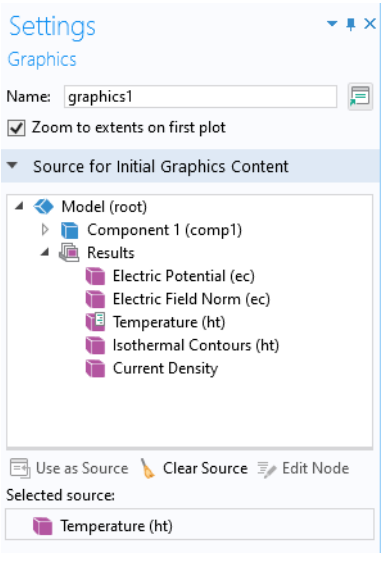
Graphics

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**.

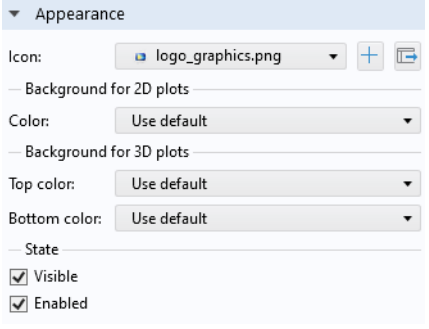
Selecting the check box **Zoom to extents on first plot** ensures that the first plot that appears in the graphics canvas shows the entire model (zoom extents). This action is triggered once the first time that graphics content is sent to the graphics object.

In the section **Data Picking**, below **Source for Initial Graphics Content**, selecting the check box **Enable 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 95.

Appearance

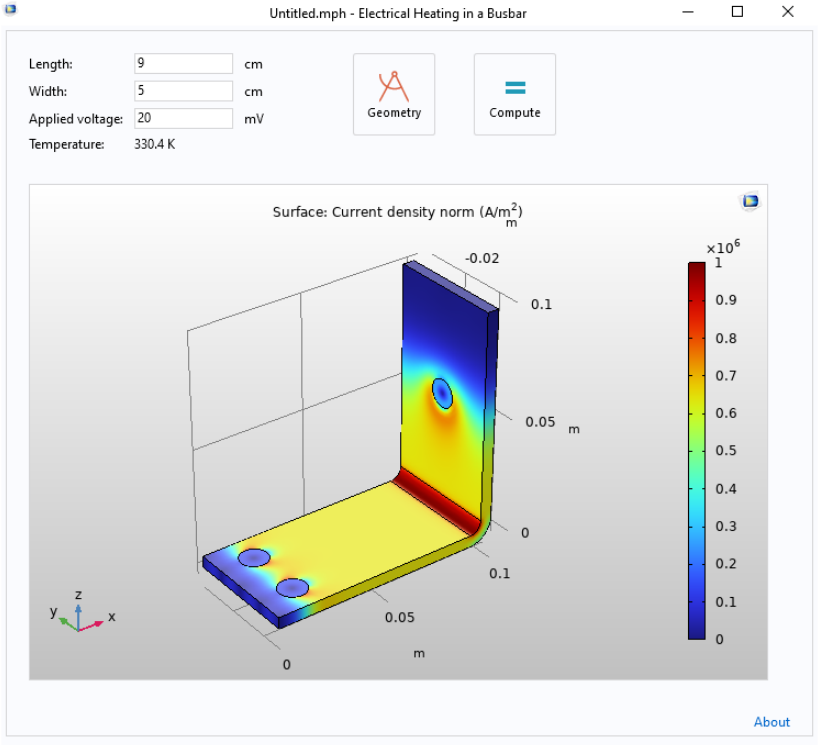
For a graphics object, the **Appearance** section of the **Settings** window has the following options:

- 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**.



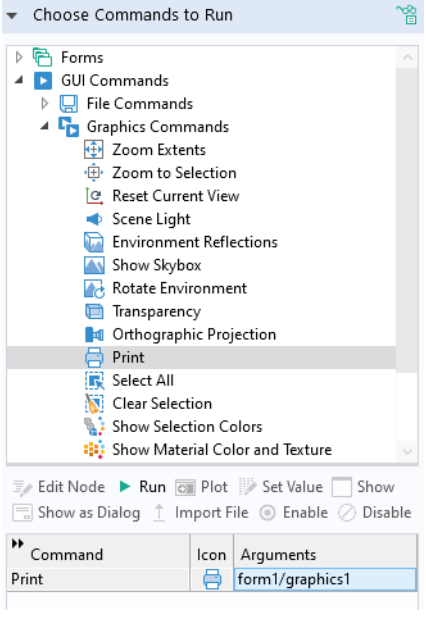
In addition, the subsection **State** contains settings for the visible and enabled state of the graphics object. For more information, see “Changing the Enabled and Visible State of a Form Object” on page 79.

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** include:

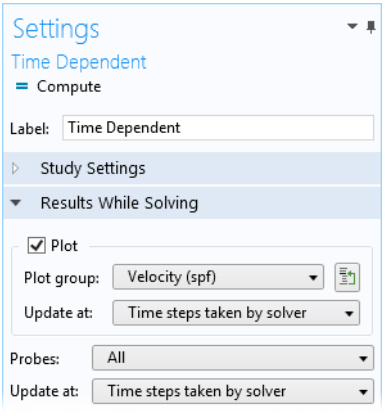
- **Zoom Extents**
  - Makes the entire model visible.
- **Reset Current View**
  - Resets the currently active view to the state it had when the application was launched; also see “Views” on page 89.
- **Scene Light**
  - Toggles the scene light (on or off).
- **Transparency**
  - Toggles the transparency 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 many of these commands have corresponding toolbar buttons in the standard graphics toolbar. See the next section “Graphics Toolbar”.

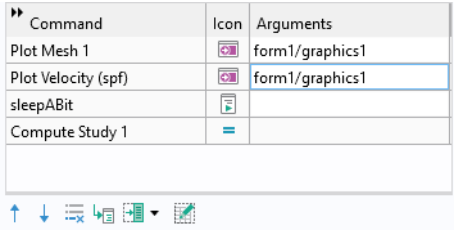
Plot While Solving

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.



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

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

For more information on the method `sleep`, see “sleep” on page 364.

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 281), 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 185.

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:

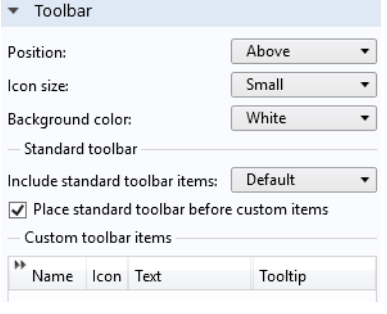
```
useGraphics(null, "/form1/graphics1");
```

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.



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 a **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 92.

Graphics Toolbar for Plot Groups

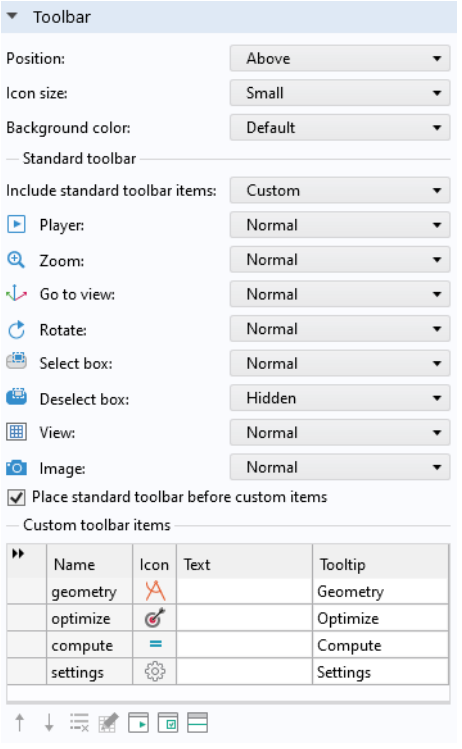
The figure below shows the standard graphics toolbar as it appears when a **3D Plot Group** node is used as a **Source for Initial Graphics Content**.

If the **Source for Initial Graphics Content** is set to an **Animation** node, then additional buttons for playing the animation are added to the graphics toolbar, as shown in the figure below.

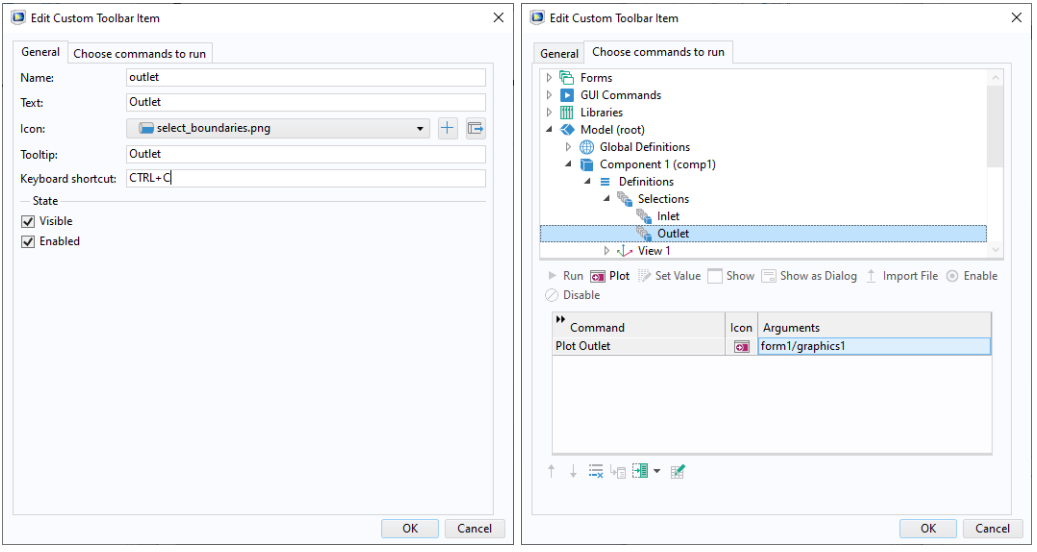
Custom Graphics Toolbar Buttons

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.



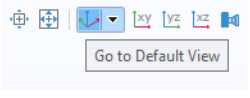
To edit the command sequence for a toolbar item, click the **Edit** button to open the **Edit Custom Toolbar Item** dialog box.



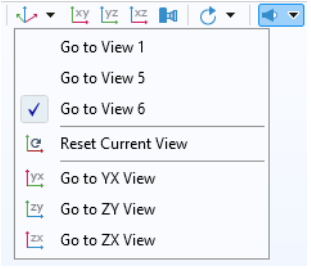
This dialog box has settings that are similar to those of a button or a toolbar item with the contents divided into two or three tabs depending on if the item is a toggle item or not. For details, see “Button” on page 67 and “Toolbar” on page 323.

Views

In the graphics toolbar of an application, the **Go to Default View** button (for 3D graphics only) will reset the current view to the default view. If you click the arrow



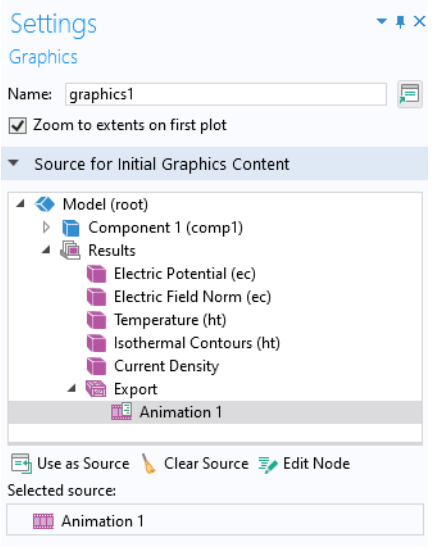
next to this button, a menu will be displayed with all applicable views. The currently active view is indicated with a check mark.



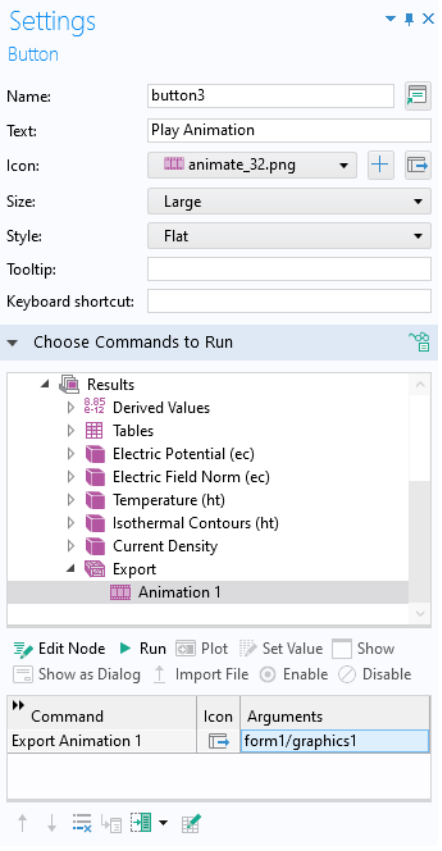
In addition to a list of all views, there is an option **Reset Current View** that will reset the currently active view to the state it had when the application was launched.

Animations

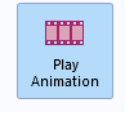
You can display animations in an application by using a **Results > Animation** node as the **Source for Initial Graphics Content**.



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

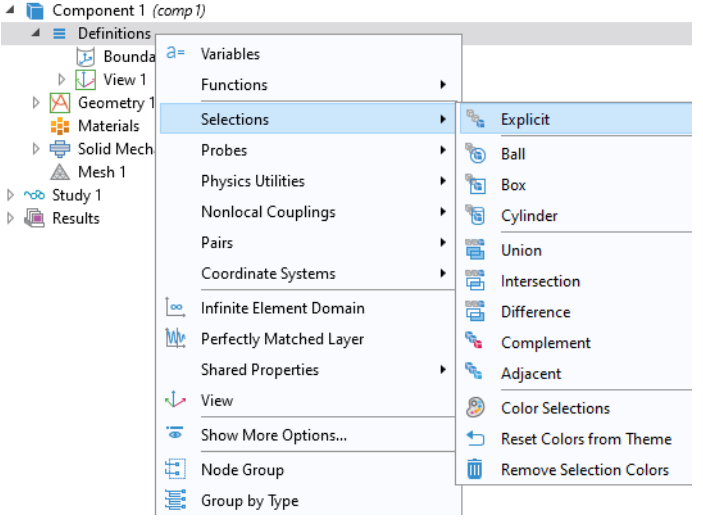


When using the Form Wizard or Editor Tools, the animation button will have the following default appearance:



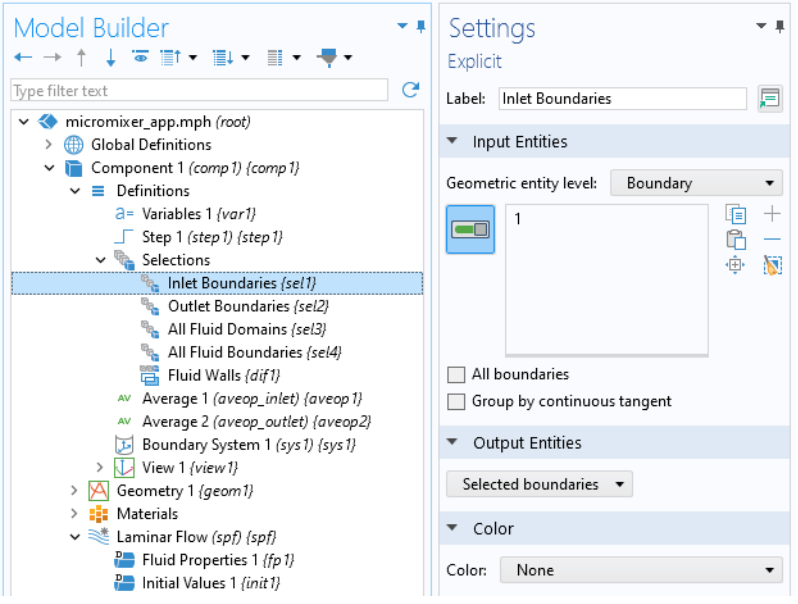
Selections in the Model Builder

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, for example, by adding subnodes under the **Component > Definitions** node, as shown in the figure below. You can also create selections generated by geometry or mesh sequences. These can be reused throughout a model component.

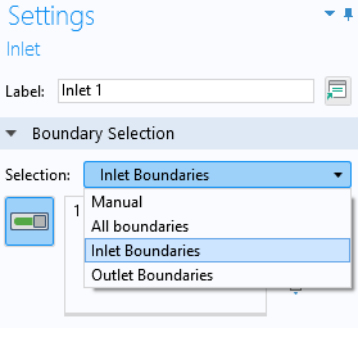


As an example of how selections can be used, consider selections for boundary conditions. When you select which boundaries should be associated with a certain boundary condition, you can click directly on those boundaries in the graphics window of the COMSOL Desktop environment. This is the default option 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 selections that can be reused for several different kinds of boundary conditions within a **Component** by just selecting from a drop-down list. The figure below

shows an **Explicit** selection given the name **Inlet Boundaries** with an associated boundary (1).



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

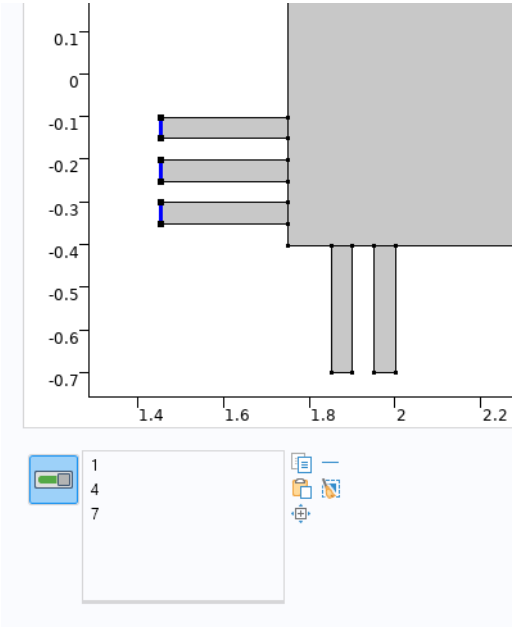


For convenience, in addition to the **Manual** option, there is also a shortcut for **All boundaries**.

Selections in the Application Builder

The **Explicit** selections, created under **Definitions**, or by geometry or mesh sequences, let you group domains, boundaries, edges, or points based on entity number, and the example below illustrates how to work with this type of selection. **Explicit** selections can be made interactive in an application, but other types of selections, such as a coordinate-based **Box** selection, are shown in read-only mode.

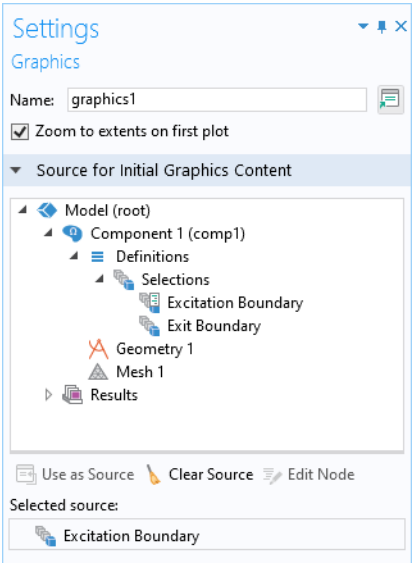
You can allow the user of an application to interactively change which entities belong to an **Explicit** selection with a **Selection Input** object or a **Graphics** object. In the example below, the embedded model has a boundary condition defined with an **Explicit** selection. Both a **Selection 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 303.

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.

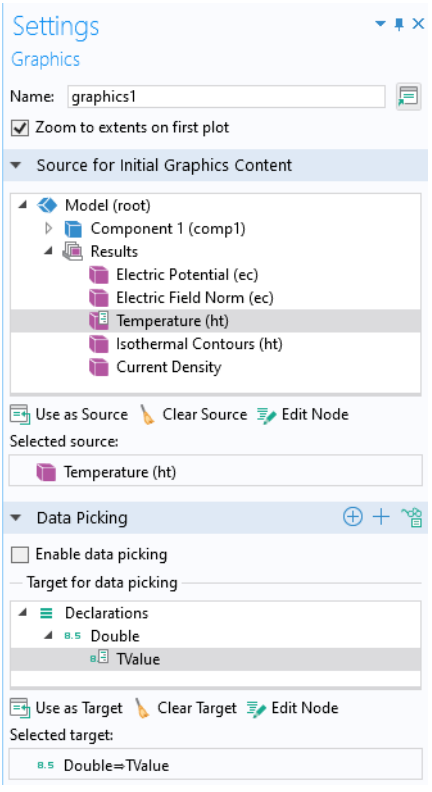
You can let a global **Event** be triggered by an **Explicit** selection. This allows a command sequence or method to be run when the user clicks a geometry object, domain, face, edge, or point. For more information on using global events, see “Events” on page 150 and “Source For Data Change Event” on page 152.

For an example of how to use selections in an add-in, see the *Application Programming Guide*.

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 `Tvalue` 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 182.

Input Field

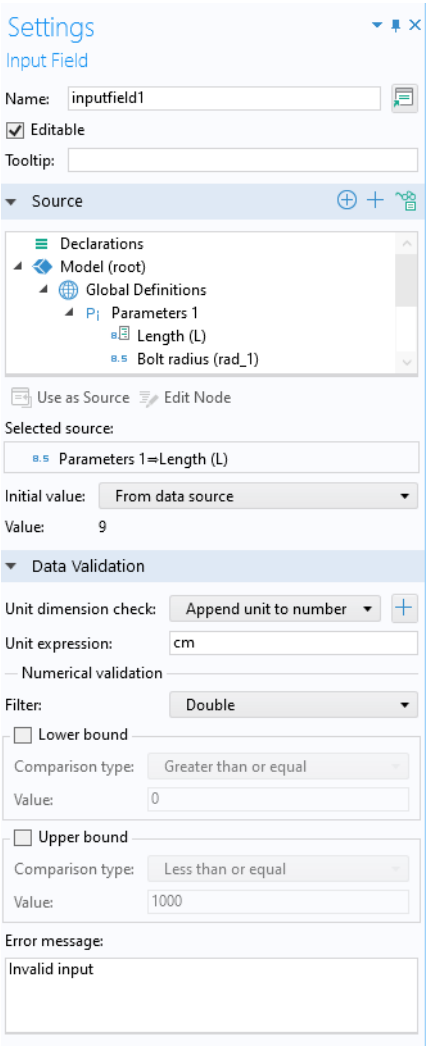
An **Input Field** allows a user to change the value of a parameter or variable. In the Form Wizard, when a parameter or variable is selected, three form objects are created:

- 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, click the **Input Field** button in the **Form Objects** section of the ribbon **Form** tab. 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.



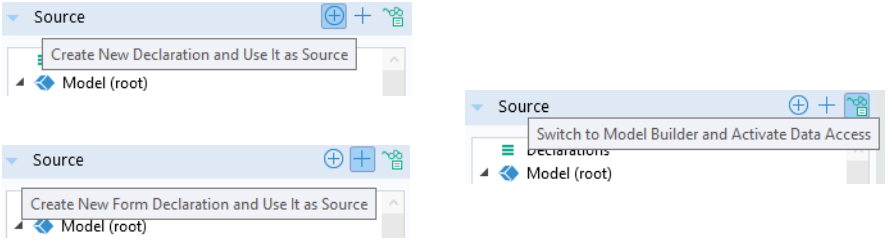
In addition to parameters and variables, input fields can use an **Information** node as **Source**.

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

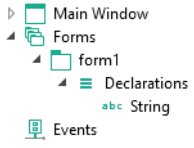
different from that of the source. If the **Editable** check box is cleared, then the **Initial value** will be displayed by the application and cannot be changed. This makes it possible to use an **Input Field** an alternative to, for example, a **Text Label** object for displaying text.

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.



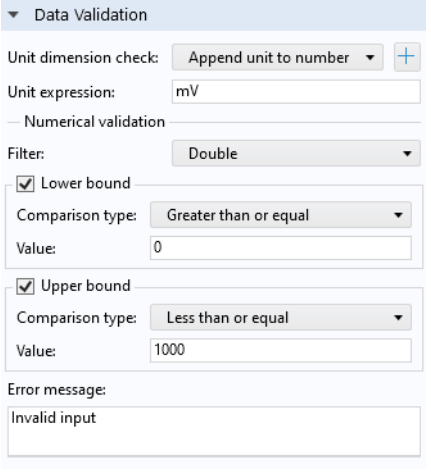
The **Create New Declaration and Use It as Source** button can be used to add new variables under the **Declarations** node. For more information, see “Declarations” on page 160. The **Create New Form Declaration and Use It as Source** button can be used to add new variables under the **Declarations** 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 108.

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.



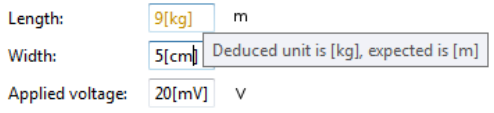
When creating an input field in the Form Wizard, the setting **Append unit to number** is used when applicable. This setting assumes that a user enters a number into the input field, but it can also handle a number followed by a unit using the COMSOL square bracket `[]` unit syntax. If the **Unit expression** 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**

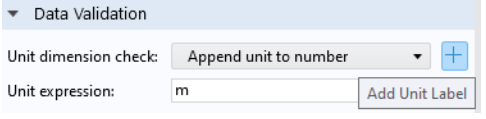
A value or expression can be highlighted in orange to provide a warning when the user of an application enters an incompatible unit, which is any unit of measure that cannot be converted to the units specified in the **Data Validation** settings. Enable this feature by selecting **Compatible with physical 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.

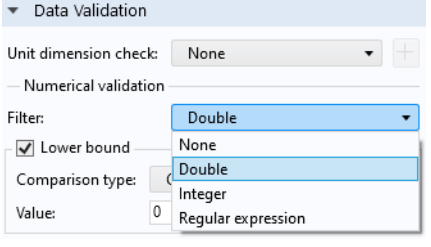


Clicking this button will add a unit label to the right of the input field if there is not already a unit label placed there.

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 the **Lower bound** and **Upper bound** 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. For example, consider data validation of an input field for a length parameter `L` with unit `cm`. Further, assume that a global parameter `Lmax` is used as the **Upper bound** value. If you would like the maximum value of `L` to be 15 cm, then the following values for the parameter `Lmax` will work: `15` (with no unit), `15[cm]`, `0.15[m]`, `150[mm]`, etc.

For the **Append unit from unit set** option, the **Lower bound** and **Upper bound** 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 174.

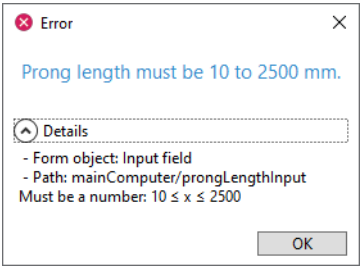
The **Regular expression** option, available when the **Unit dimension check** is set to **None**, allows you to use a regular expression for matching the input string. For more information on regular expressions, see the dynamic help. Click the help icon in the upper-right corner of a window and search 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.

For additional information on how to use more advanced **Data Validation** features and how to use method code to perform data validation, see the *Application Programming Guide*.

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.

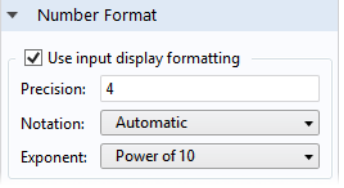


The debugging information typically consists of the type of form object, the path to the form object, and the reason for the failure; for example, `10<=x<=2500`.

No debugging information is added when launching an application by using the **Run Application** option or COMSOL Server.

Number Format

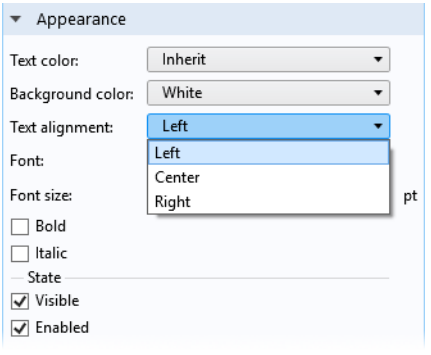
The **Number Format** section contains a check box **Use input display formatting**. If selected, it enables the same type of display formatting as a **Data Display** object.



For more information, see “Data Display” on page 105.

Appearance

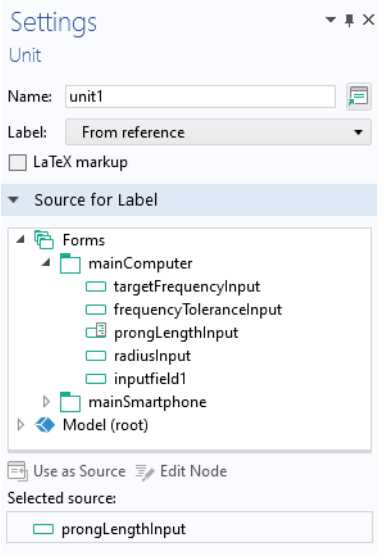
In addition to color and font settings, the **Appearance** section for an input field contains a **Text alignment** setting that allows the text to be **Left**, **Center**, or **Right** aligned.



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 79.

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.

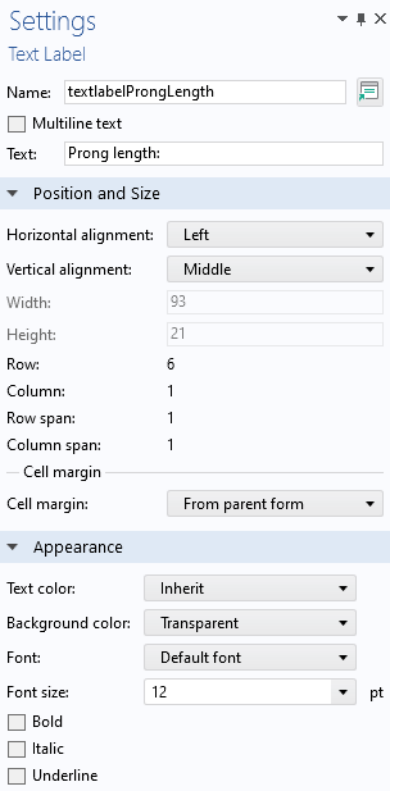


When adding an input field using the Form Wizard, a unit object is automatically added when applicable. By default, the unit is displayed using Unicode rendering. As an alternative, you can use LaTeX rendering by selecting the **LaTeX markup** check box. Then, the display of units will not depend upon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

Text Label

A **Text Label** object simply displays text in a form. When adding an input field using the 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**, click the **Text Label** button in the ribbon **Form** tab, in the section **Form Objects**. 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 108
- 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)

    The 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 Form Wizard for Generating Data Display Objects

In the Form Wizard in, for example, 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:

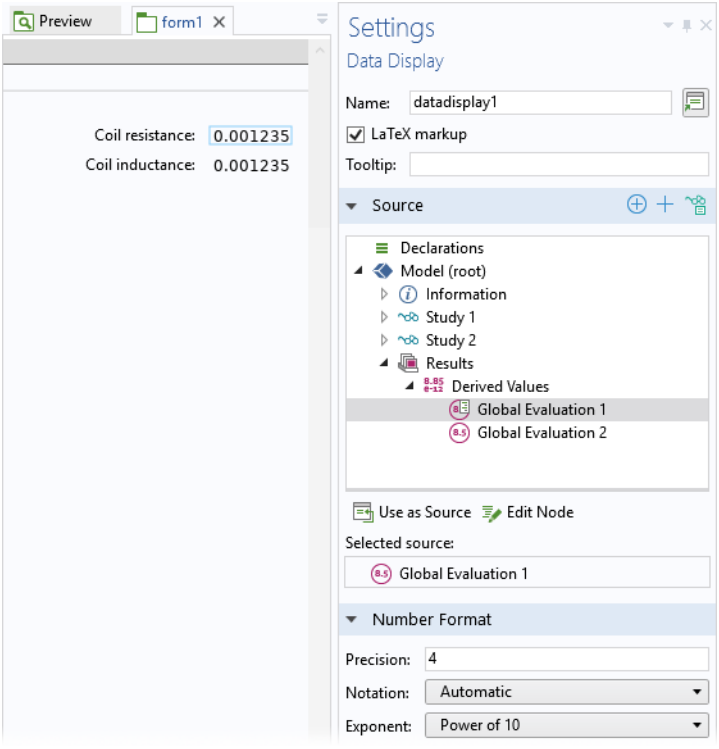
- a **Text Label** object for the **Description** of the variable
- a **Data Display** object for the value of the variable

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**, **Exponent**, and **Unit**.

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 168) 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.

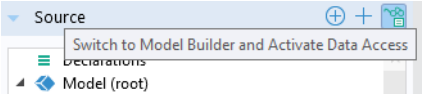
Data Access in the Form Editor

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**.

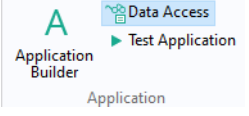
The remainder of this section gives an introduction to using **Data Access**, with examples for input fields and buttons.

Data Access for Input Fields

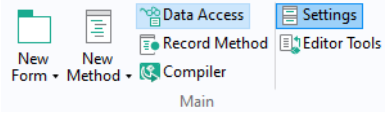
By default, you can link input fields to parameters and variables defined in the model tree under the **Parameters** or **Variables** nodes and to variables declared in the application tree under the **Declarations** node. To access additional model tree node properties, click the **Switch to Model Builder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



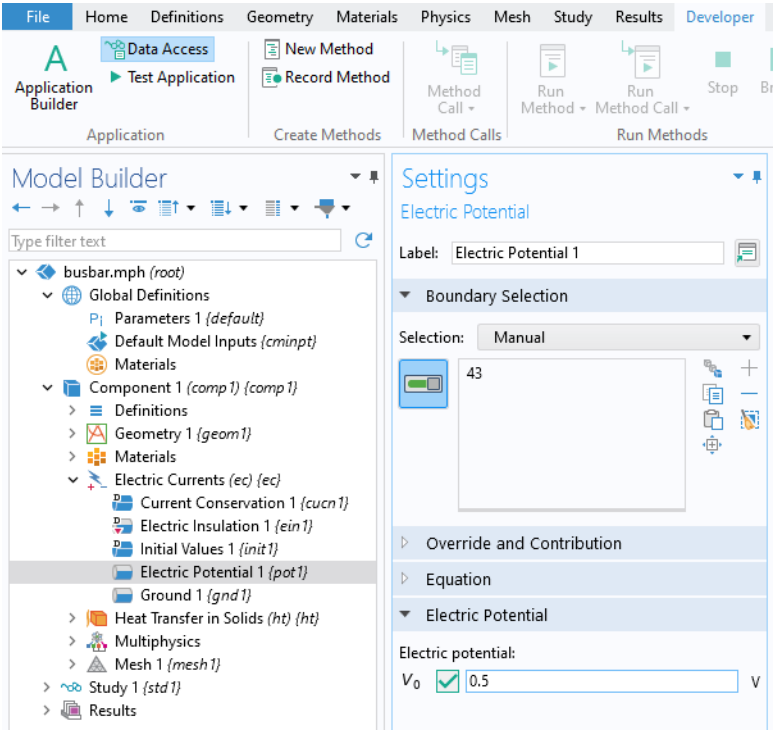
You can also access it from the **Application** group of the **Developer** tab in the Model Builder workspace



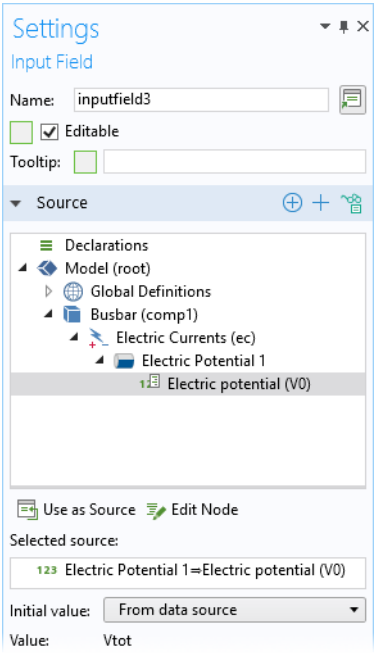
or from the **Home** tab in the Application Builder workspace.



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:



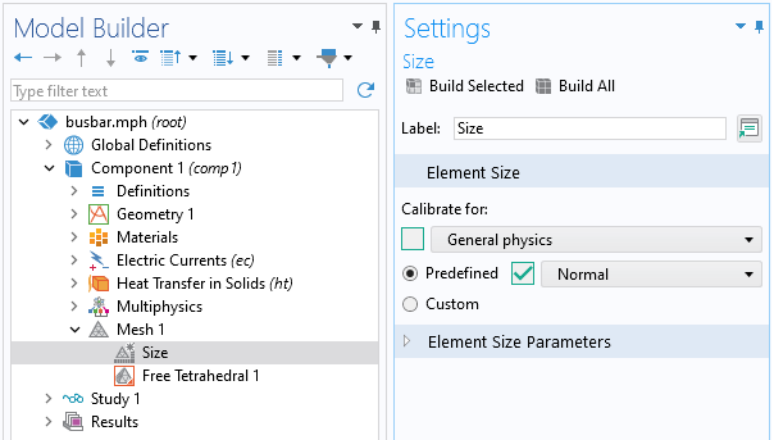
The figure below shows the **Settings** window for an input field. The list of possible sources for this field now contains the **Electric potential**.



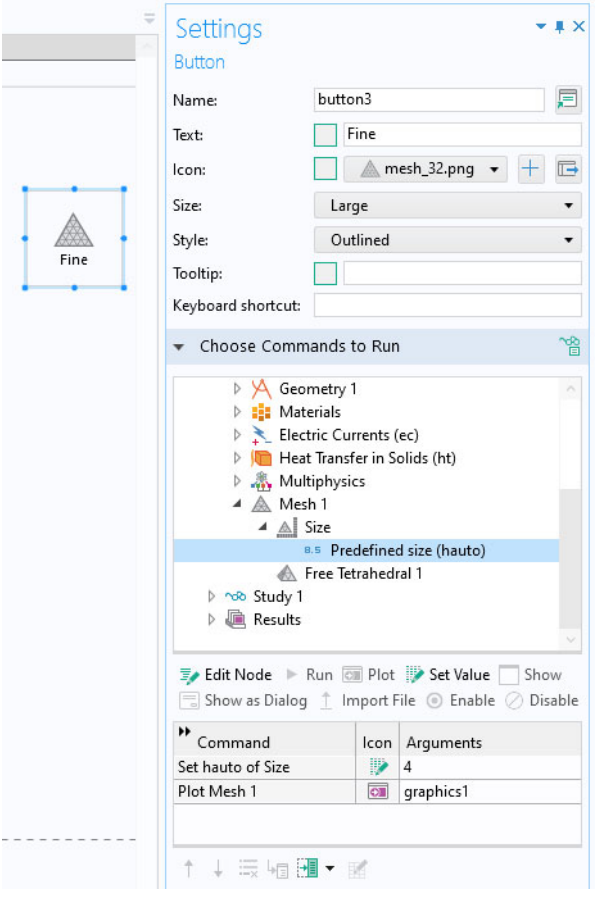
In addition, as seen in the figure above, **Data Access** makes it possible to access the check box **Editable** and the **Tooltip** text of the input field form object. In addition to the settings of the Model Builder, **Data Access** lets you access certain properties of the Application Builder.

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 non-negative 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 317.

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 195.

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 244.

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
Ribbon Item	Choose Commands to Run
Event (Global)	Choose Commands to Run Source for Data Change Event

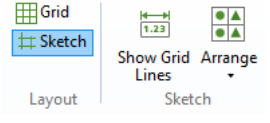
A global event, menu, ribbon, or toolbar item provides a **Choose Commands to Run** section in its **Settings** window, to which the functionality described above in the section on buttons also applies. Global events and many form objects provide a **Source** section in its **Settings** window, and the 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 86, “The Main Window Editor” on page 138, “Events” on page 150, “Table” on page 312, and “Toolbar” on page 323.

Sketch and Grid Layout

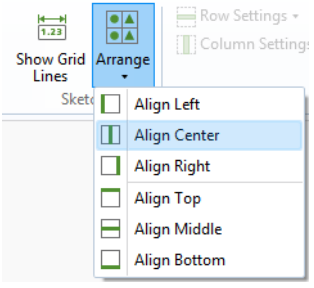
The Form Editor provides two layout modes for positioning form objects: sketch layout mode and grid layout mode. The default is sketch layout mode, which lets you use fixed positions and sizes of objects in pixels. Use grid layout mode to position and size objects based on a background grid with cells.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



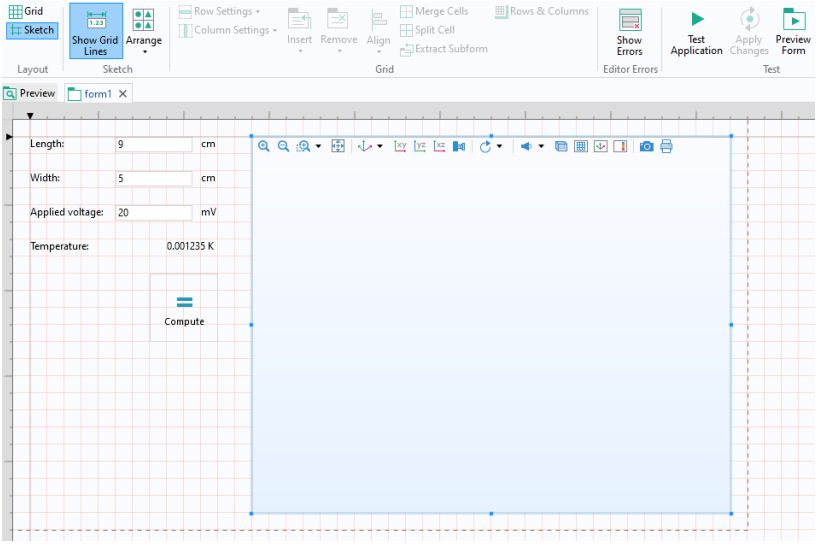
The **Sketch** group in the **Form** tab has two options: **Show Grid Lines** and **Arrange**. The **Arrange** menu allows you to align groups of form objects relative to each other.



Sketch Grid

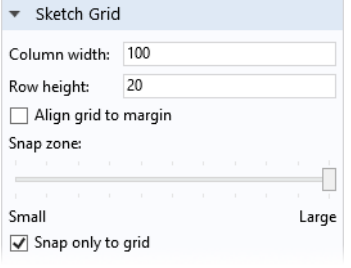
The **Show Grid Lines** option displays a sketch grid to which objects are snapped. Note that the grid used in sketch layout mode is different from the grid used in grid layout mode. The default setting for sketch layout mode is to show no grid lines. Without grid lines visible, a form object being dragged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



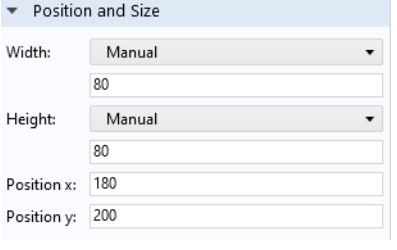
In the **Settings** window of the form, you can change the settings for the sketch grid:

- **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.

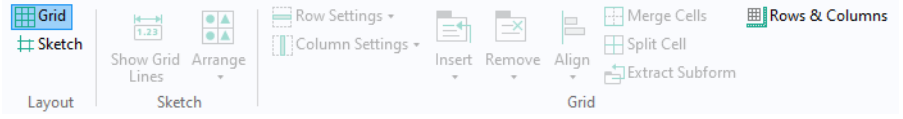


Form objects are allotted as much space as required or as specified by their **Width** and **Height** values. Form objects are allowed to overlap, when working in sketch layout mode.

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.

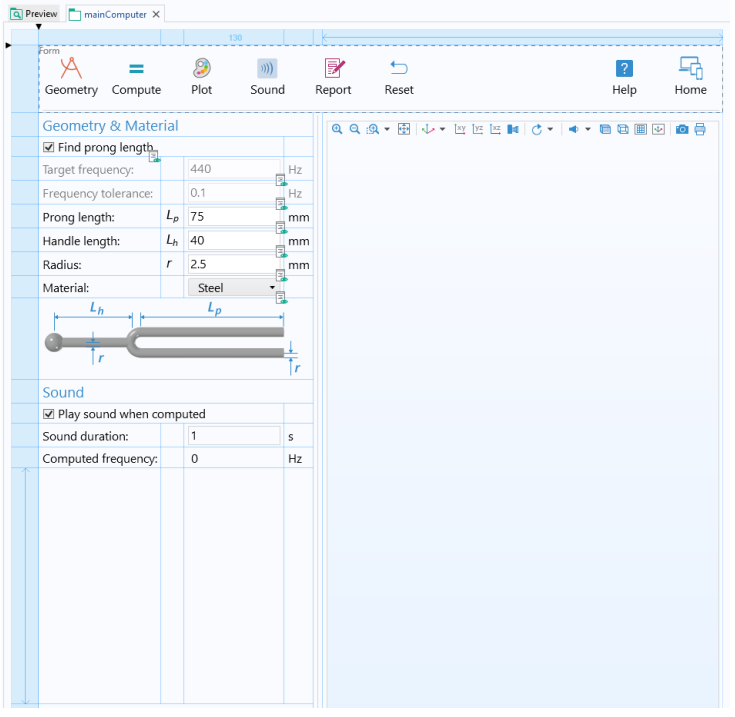


The buttons and menus in the ribbon **Grid** group give you easy access to commands for:

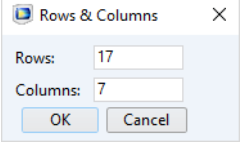
- Changing the row and column growth rules between **Fit**, **Grow**, and **Fixed**, which determine the layout when the user interface is resized (**Row Settings** and **Column Settings**).
- Inserting or removing rows and columns (**Insert** and **Remove**).
- Aligning form objects within grid cells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

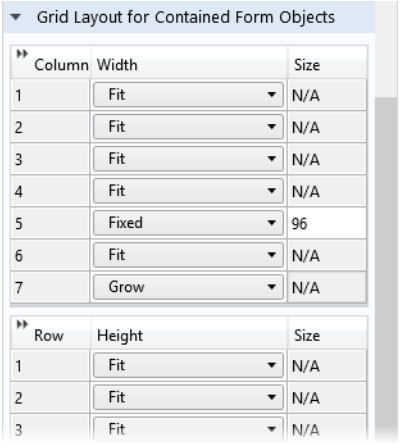
After switching to grid layout mode, the form window shows blue grid lines.



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.

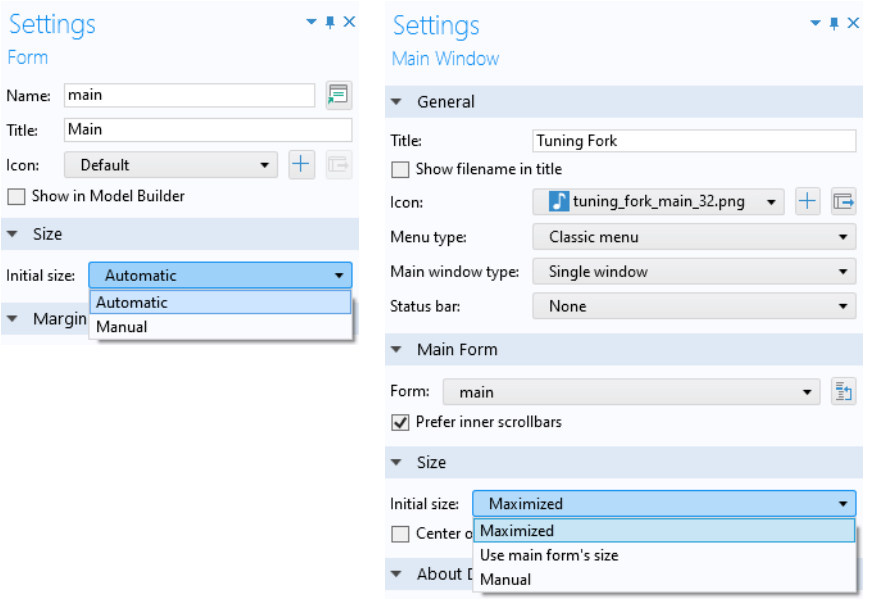


To interactively select a form, as displayed in the Form Editor, click the top-left corner of the form.

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.

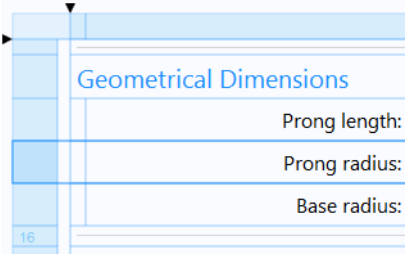
Note that if you switch from sketch to grid layout mode, all rows and columns will have the setting **Fit** and the handles for the frame will not be displayed. If any of the rows and columns have the **Height** or **Width** setting set to **Grow**, then the frame will display handles for resizing in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

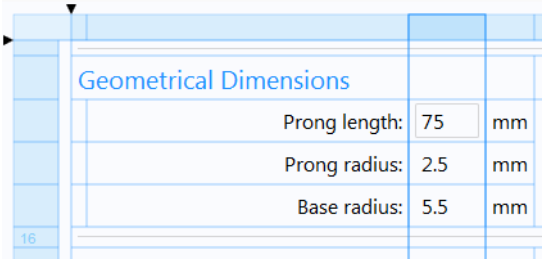


Rows and Columns

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.

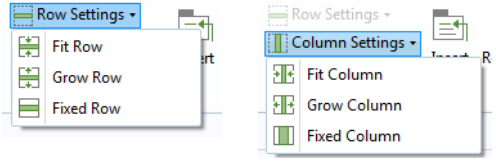


Similarly, to select a column, click the cell at the top. This cell cannot contain any form objects. The figure below shows the third column highlighted. In this case, the **Column Settings** menu is enabled in the ribbon tab.

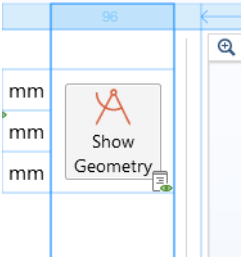


The **Row Settings** and the **Column Settings** have the same three options:

- **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.

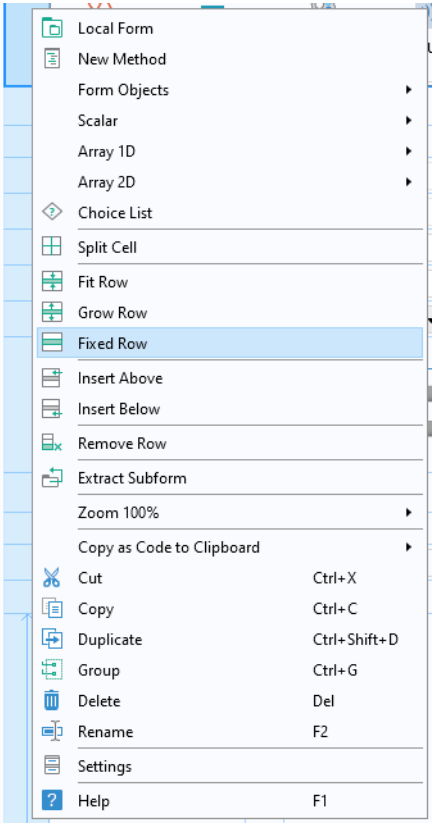


You can interactively change the row height and column width by dragging the grid lines.



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**.

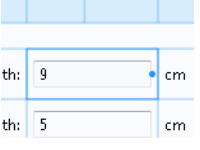
As an alternative to changing the **Row Settings** or **Column Settings** from the ribbon, you can right-click in a row or column (the light-blue area to the left or at the top of the form) and select from a menu.



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

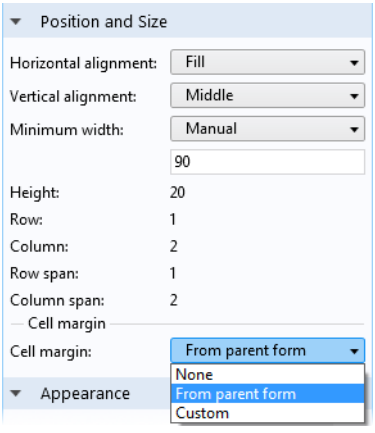
Cells

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 129
- **Custom**
  - Custom margins applied only to this form object

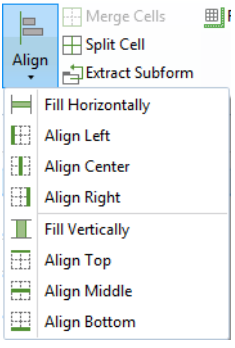
If the **Horizontal alignment** or **Vertical alignment** is set to **Fill** and the growth policy of the column or row allows the form object to be resized, then you can specify a minimum width or height, respectively. The minimum size can be set to **Manual** or **Automatic**. The **Manual** option 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 116.

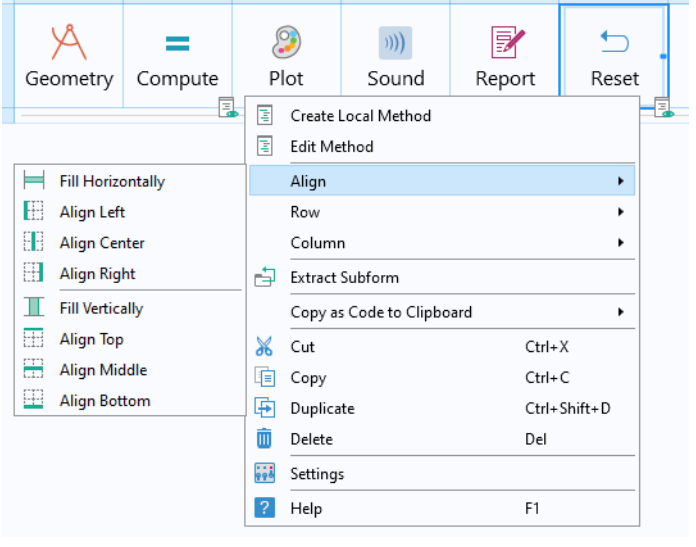
You can click and drag a rubber box to select multiple cells.

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.

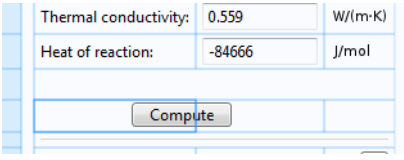


As an alternative, you can right-click a form object and select from a context menu.



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.



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

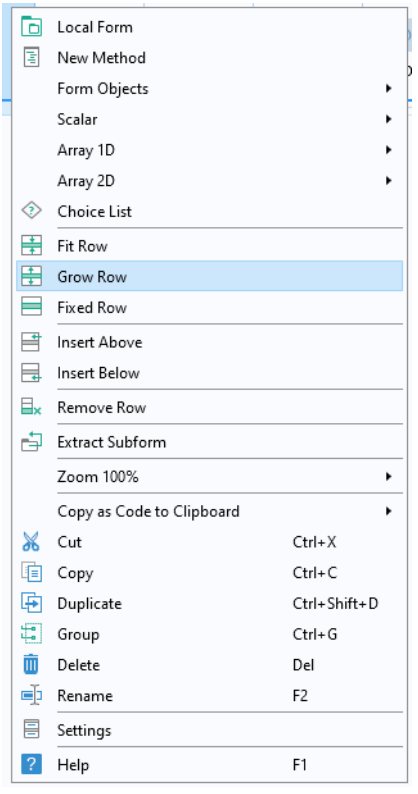
Automatic Resizing of Graphics Objects

If you use one of the more sophisticated Form Wizard templates you will automatically get a layout with a resizable graphics object.

If you don't use these templates, then, in order to make the graphics object of an application resizable, follow these steps:

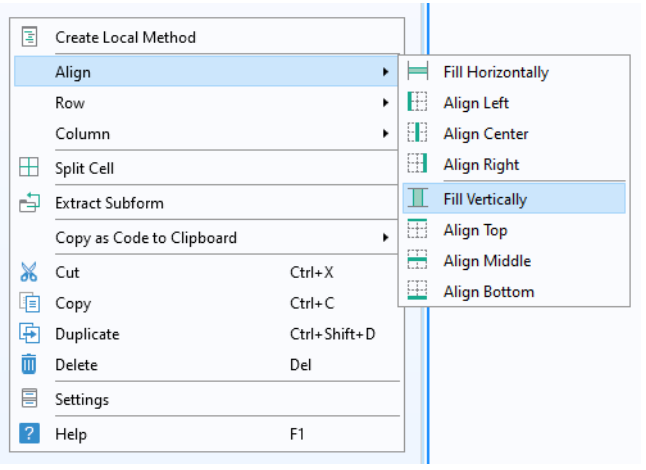
- Change the layout mode of the form containing the graphics object from sketch to grid layout mode.
- Change the **Height** setting for any row covering the graphics object to **Grow**. To change this, click the leftmost column of the row you would like to

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

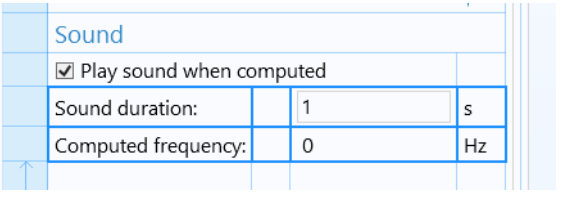
right-clicking the graphics object and selecting **Align > Fill Horizontally** and **Align > Fill Vertically**.



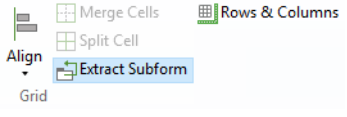
Following the steps above, you may find it easier to make graphics objects resizable by performing grid layout mode operations, such as adding empty rows and columns as well as merging cells. If you are already in grid layout mode, then a graphics object will default to **Fill** in both directions.

Extracting Subforms

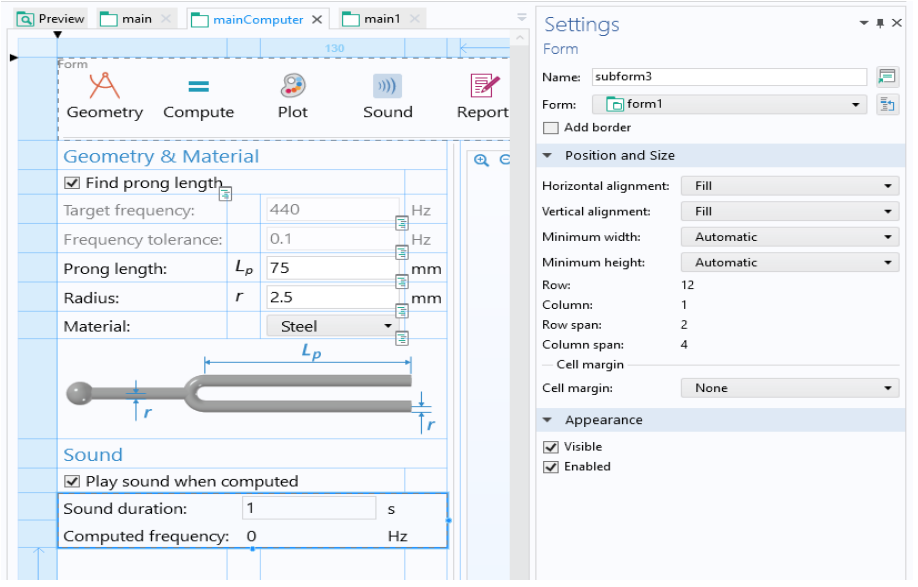
You can select a rectangular array of cells in a form and move it to a new form. First, select the cells by using Ctrl+click or Shift+click.

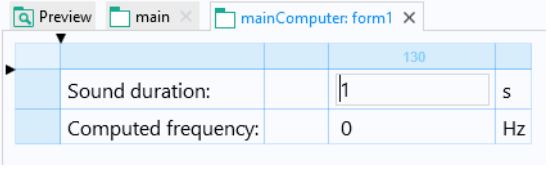


Then, click the Extract Subform button in the ribbon.



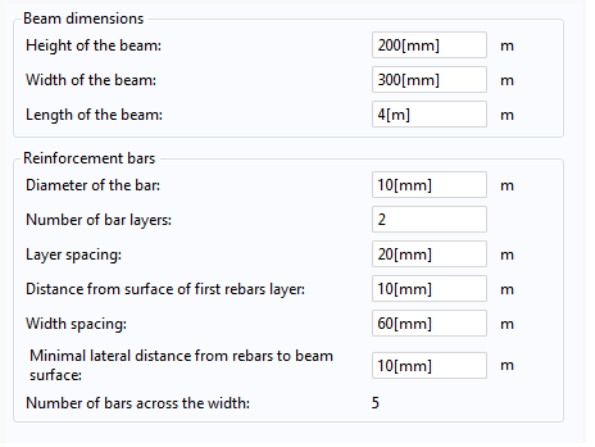
This operation creates a new form with the selected cells and replaces the original cells with a form object of type **Form**. In the **Settings** window of the subform, the **Form** reference points to the new form containing the original cells.





Inherit Columns and Cell Margins

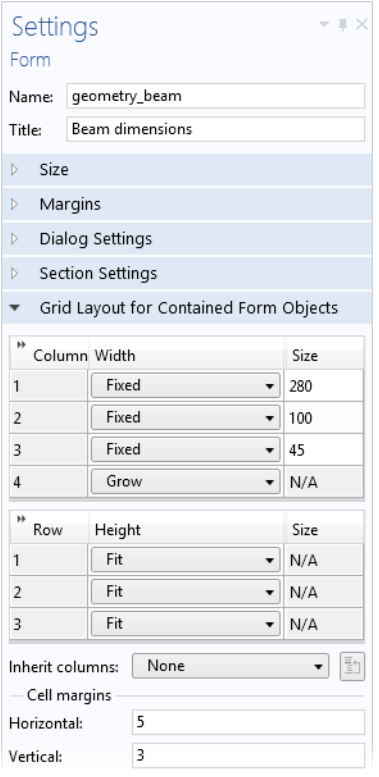
By using subforms, you can organize your user interface, for example, by grouping sets of input forms. The figure below shows part of a running application with two subforms for **Beam dimensions** and **Reinforcement bars**.

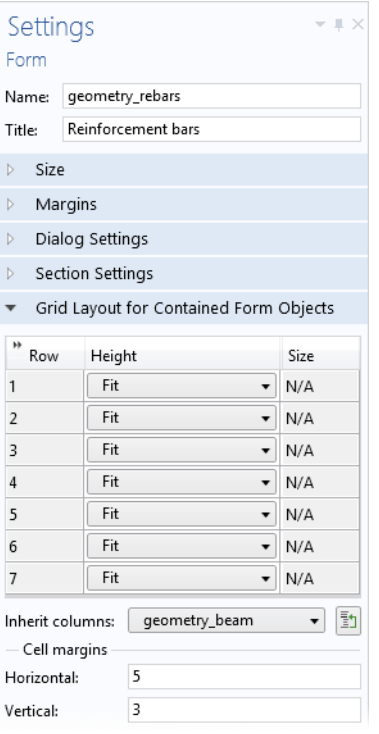


For more information on adding subforms to a form, see the previous section and “Form” on page 281.

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`.

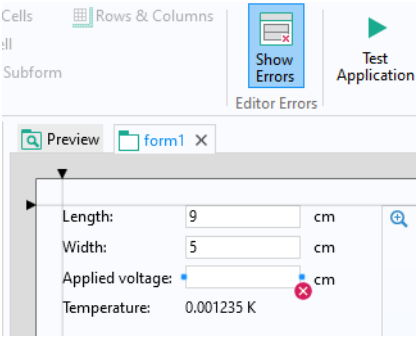




In the subsection **Cell margins**, you can specify the **Horizontal** and **Vertical** margins that are added between form objects and the borders of their containing cells. These settings will affect all form objects contained in the form, with their individual **Cell margins** set to **From parent form**; See “Cells” on page 122.

Show Errors

When using the Form Editor or Main Window Editor, there is a toggle button **Show Errors** in the ribbon.



When enabled, any object that is in an error or warning state, for example, a missing source for an input field object, will show an icon in the corner of that object. This makes it easy to locate errors in a form or main window design. By default, the **Show Errors** button is disabled.

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.

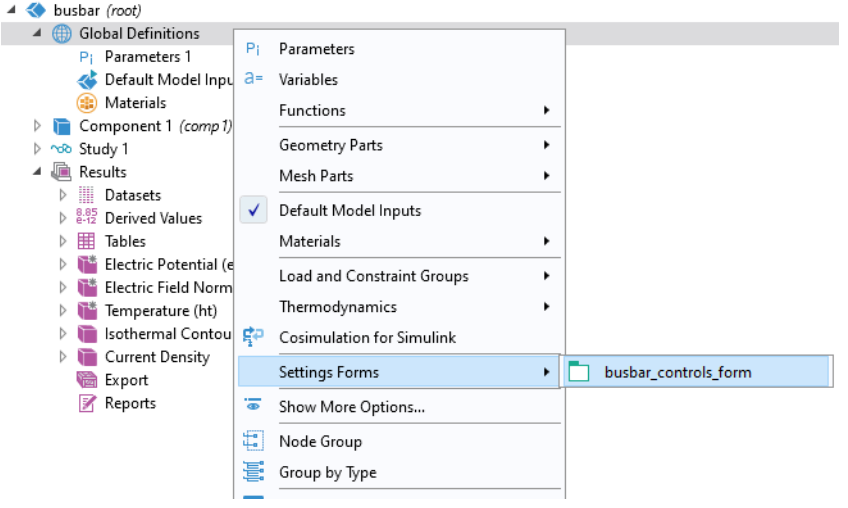
When you copy and paste forms and form objects between applications, the copied objects may contain references to other forms and form objects. Such references may or may not be meaningful in the application to which they are copied. For more information on the set of rules applied when pasting objects, see “Appendix B — Copying Between Applications” on page 326.

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.

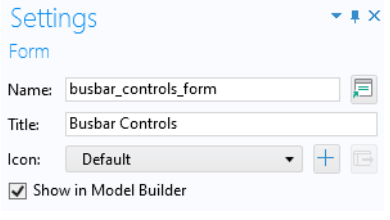
Using Forms in the Model Builder

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.

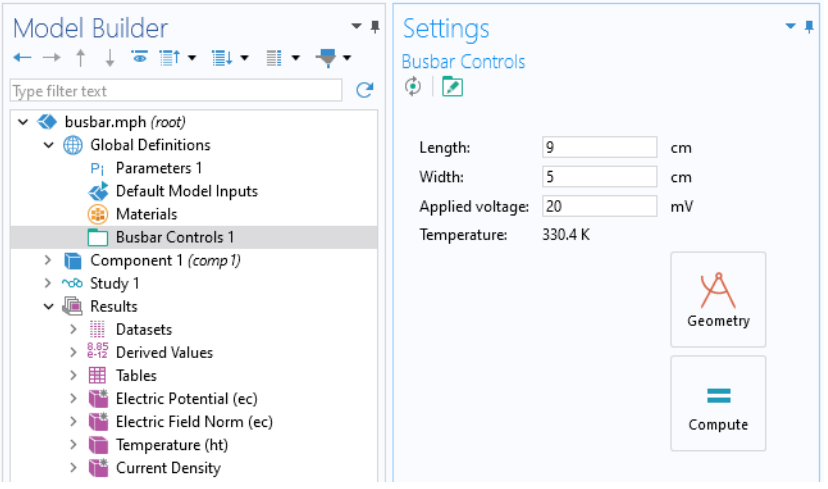
To use a form, right-click **Global Definitions** and select the form under **Settings Forms**.



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.

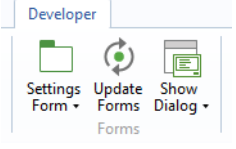


Once added to the model tree, the form is shown as a **Settings** window, shown in the figure below.



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

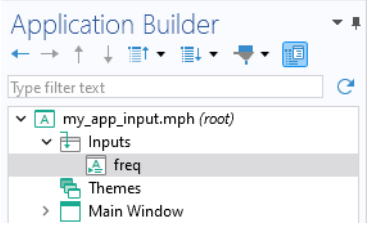
- Click the corresponding model tree node.
- Select it from the **Settings Form** menu button in the Developer tab in the ribbon.
- Show it as a dialog box by selecting it from the **Show Dialog** menu button in the **Developer** tab in the ribbon.



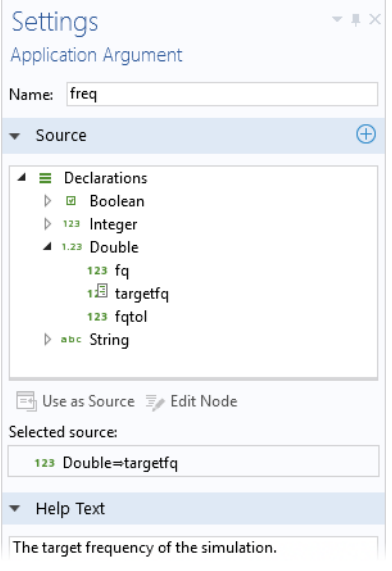
For reusing a **Settings Form** between sessions, you can create an **Add-in**. For more information, see “Creating Add-Ins” on page 226.

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.

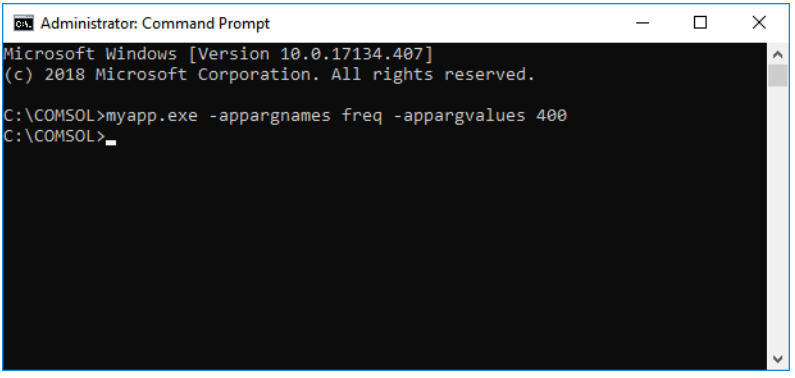


Command-line arguments are automatically written to the declarations you define as **Selected source**, in the corresponding **Settings** window for an **Application Argument**. They can be used, for example, to provide input data or configuration settings.



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`.



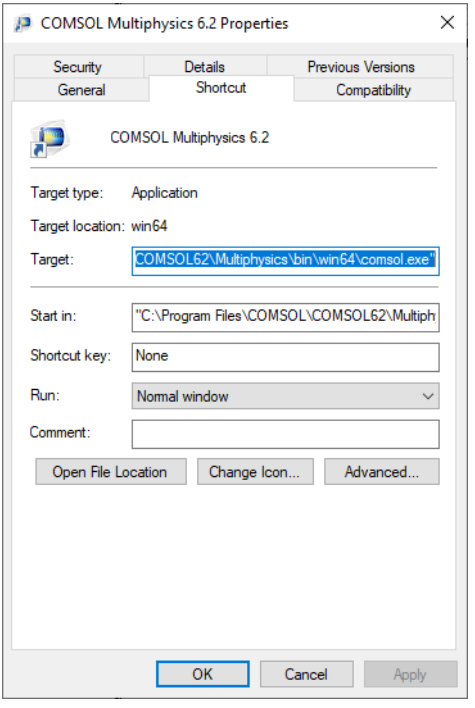
For COMSOL Multiphysics, the corresponding command would be

```
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\COMSOL62\Multiphysics\bin\win64
```

Alternatively, you can copy and paste the **COMSOL Multiphysics 6.2** 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\COMSOL62\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'
```

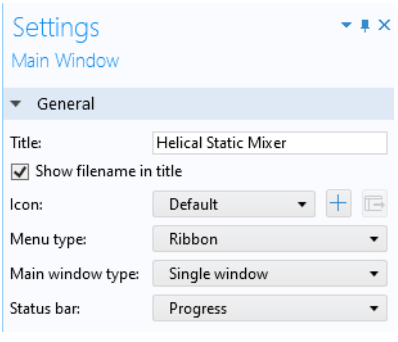
This example uses an application argument `interpfile`, which is linked to a file declaration to read the interpolation file `simpleinterp.txt` when launching the application. This file is then used in an interpolation function in the application's embedded model.

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 optional ribbon, and subwindow specifications. The Main Window Editor, visible whenever the **Main Window** node is active, lets you design menu bars, ribbon tabs, and subwindows.

General

The **Settings** window contains a **General** section with the following settings:

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



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

In the **Icon** list, select an image from the library or add an image (`*.png`) from the local file system to the library and use it as an icon. If you add a new image, it will be added to the image library and thereby embedded into the application. You can also export an icon by clicking the **Export** button to the right of the **Add Image to Library and Use Here** button.

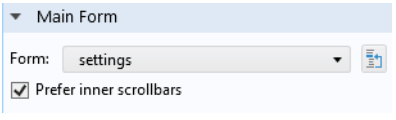
The **Main Window** node of the application tree has one child node, named **Menu Bar**. Using the **Menu type** setting, you can change this child node from **Classic menu** to **Ribbon**.

The **Main window type** setting lets you select the type of application layout: **Subwindows** or **Single window**. The **Subwindows** option makes it possible to organize forms into subwindows that, while the application is running, can be detached and docked.

The **Status bar** list controls what is shown in the status bar. From the list, select the **Progress** option to display a progress bar, otherwise, select **None**. Note that you can also create custom progress bars by writing method code.

Main Form

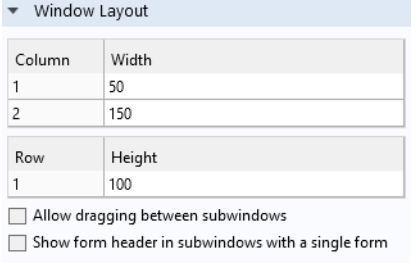
The **Main Form** section is visible if the **Main window type** is set to **Single window**. It contains a reference to the form that the main window displays.



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.

Window Layout

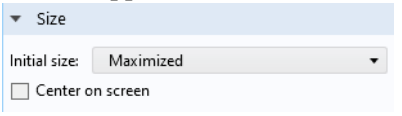
The **Window Layout** section is visible if the **Main window type** is set to **Subwindows**.



Subwindows can be defined in the lower part of the Main Window Editor. You work with subwindows in a way that is similar to forms when in grid mode. For more information, see “Subwindows” on page 147.

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 **Initial size** 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 55. 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.
- **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.

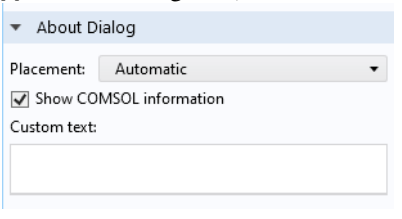
In addition, there is a **Center on screen** check box that is applicable to any **Size** setting that does not correspond to a maximized window.

When the **Main window type** is set to **Subwindows**, the **Use main form's size** option is not applicable and is therefore removed. In this case, the **Initial size** is changed to **Maximized**.

For more information on the **Use main form's size** option, see "The Form Settings Window and the Grid" on page 118.

About Dialog

The **About Dialog** section contains settings for customizing parts of the **About This Application** dialog box, which contains legal information.

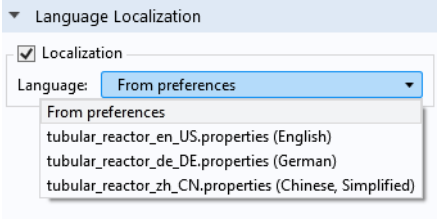


The **Placement** option lets you choose between **Automatic**, **File menu**, **Ribbon**, **Lower-right corner**, or **Lower-left corner**. The **Lower-right corner** and **Lower-left corner** options will place a hyperlink referencing the **About This Application** dialog

box in the corresponding corner of the application user interface. If selected, the **Show COMSOL information** check box will display the 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 `https` or `www` will be interpreted as hyperlinks, if possible. For example, `https://www.comsol.com` or `www.comsol.com` will be replaced with a hyperlink.

Language Localization

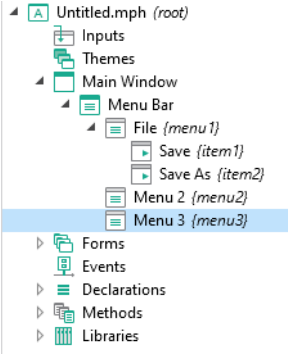
The **Language Localization** section contains settings for the localization of the application.



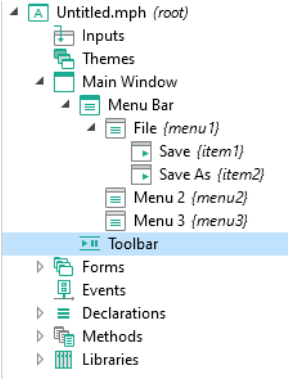
The **Localization** setting determines whether the application should be localized using language files or not and is disabled by default. The **Language** setting specifies the language files that are used for localization. By default, the **From preferences** option is selected and the language used when running the application is determined by the language preference setting in COMSOL Multiphysics, COMSOL Server, or a compiled application. For more information on localization, see the *Application Builder Reference Manual*.

Menu Bar and Toolbar

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

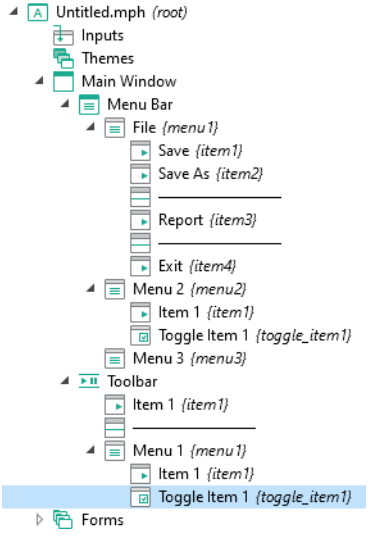


For the **Menu Bar** option, you can also add a **Toolbar** node. The **Toolbar** node and the **Menu** nodes have the same type of child nodes.

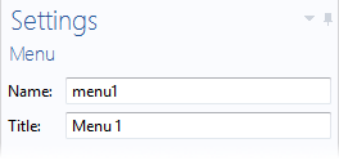


Menu, Item, and Separator

The child nodes of the **Menu** and **Toolbar** nodes can be of type **Menu**, **Item**, **Toggle Item**, or **Separator**, exemplified in the figure below:7



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

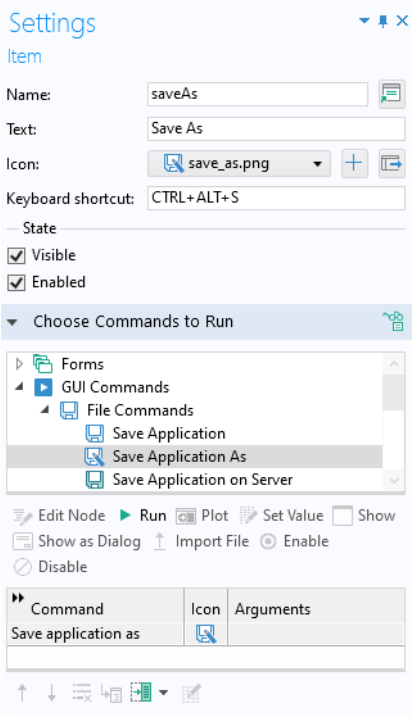


A **Menu** node can have child **Menu** nodes that represent submenus.

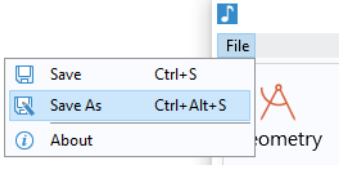
A **Separator** displays a horizontal line between groups of menus and items, and has no settings.

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 67. 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 saving an application using the command **Save Application As**.



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



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 353.

Ribbon

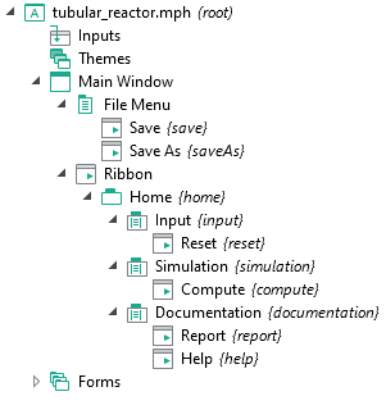
You can opt to add a **Ribbon** node to the **Main Window** instead of a **Menu Bar**. The **Ribbon** node contains the specifications of a ribbon with toolbars placed on one or

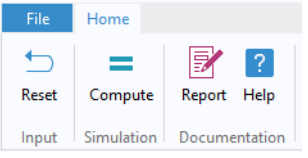
several tabs. For the **Ribbon** option, a **File** menu is made available directly under the **Main Window** node.

Ribbon Tab and Ribbon Section

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

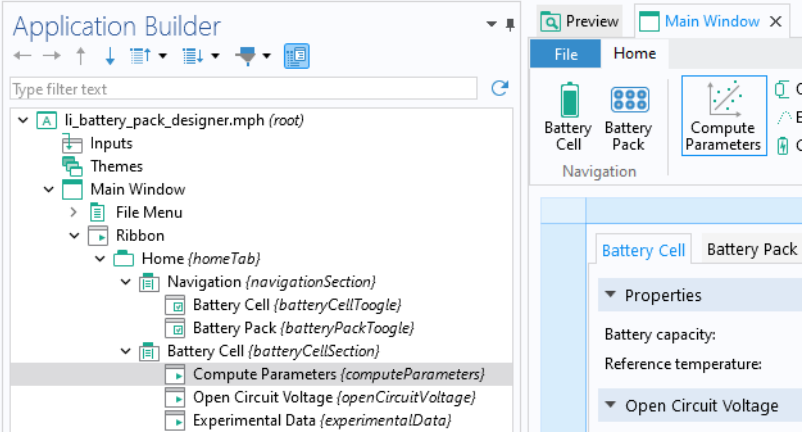
The **Item** and **Menu** options provide the same functionality as described previously for the **Menu Bar** and **Toolbar**. A **Separator** added as a child to a **Ribbon Section** is a vertical line that separates groups of **Items** and **Menus** in the running application. A **Separator** is displayed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



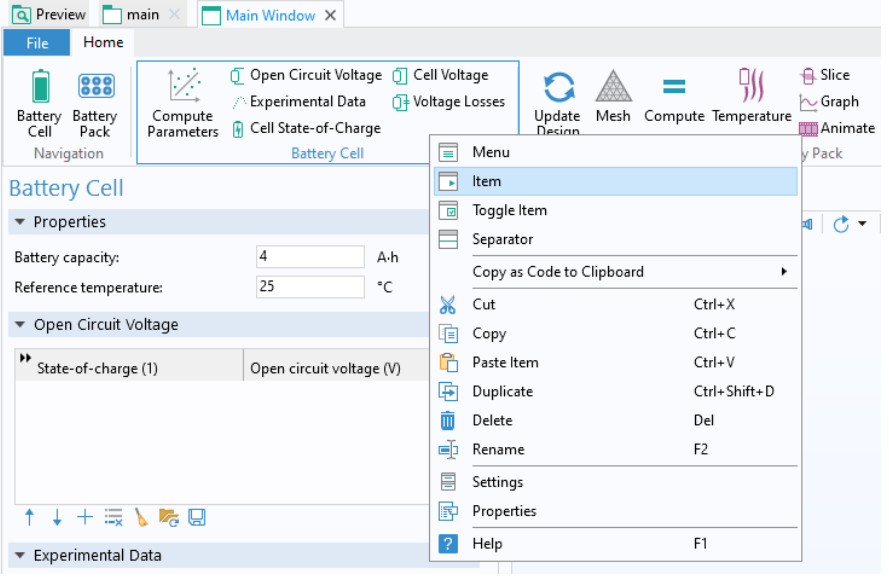


Interactive Editing of Menus and Ribbon Tabs

When creating menu and ribbon items, you can interactively position them in the Main Window Editor by dragging.

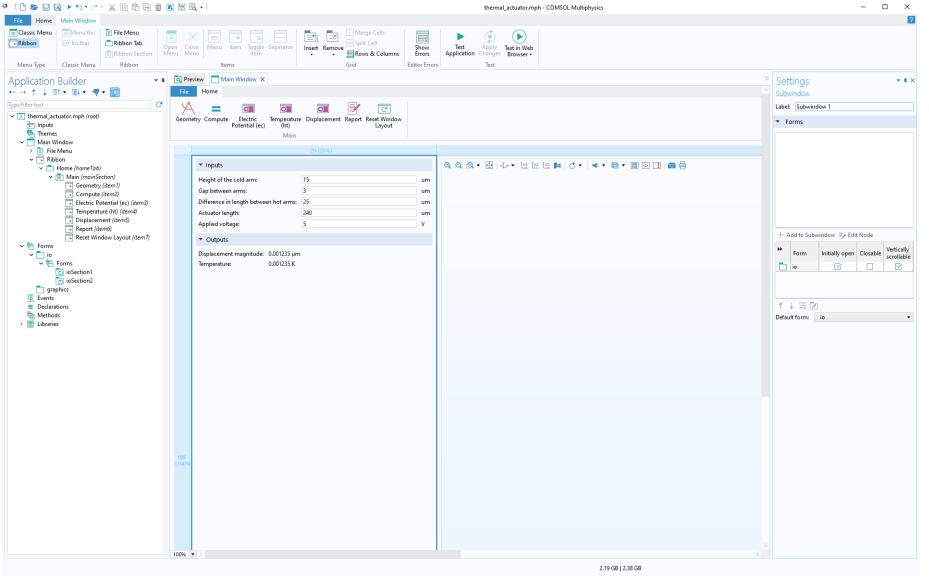


You can right-click the Main Window Editor, for example, a ribbon section, and add additional items from a context menu.



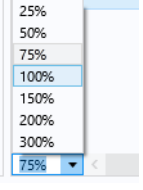
Subwindows

You can organize your forms and form collections into subwindows that can be resizable and detachable. In the **Main Window** node, if you change the **Main window type** to **Subwindows**, the lower part of the Main Window Editor becomes a workspace for the layout of subwindows.

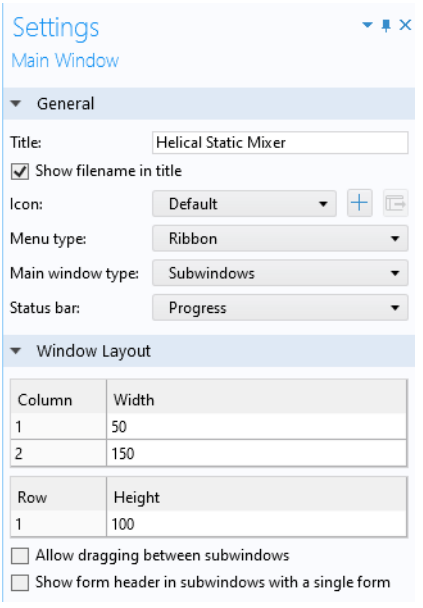


Note that the New Form Wizard templates **Basic**, **Subwindows, sections, and graphics**, and **Subwindows, tabs, and graphics** will create applications containing subwindows.

The Main Window Editor has a window scale feature, available in the lower left corner. Use this to get a better overview of the subwindow layout.



The **Window Layout** section is visible if the **Main window type** is set to **Subwindows.**



Working with subwindows is similar to working with forms in grid mode. You can insert columns to the left and to the right of existing columns. Rows can also be added above and below existing rows, and rows and columns can be deleted. Grid cells can be merged and then split again. On the ribbon of the Main Window Editor, there is a **Rows & Columns** button that enables you to quickly create a layout with a specified number of rows and columns.

The sizes of the rows and columns can be specified interactively by dragging. They can also be specified manually in the **Window Layout** section, in the **Settings** window of the **Main Window**.

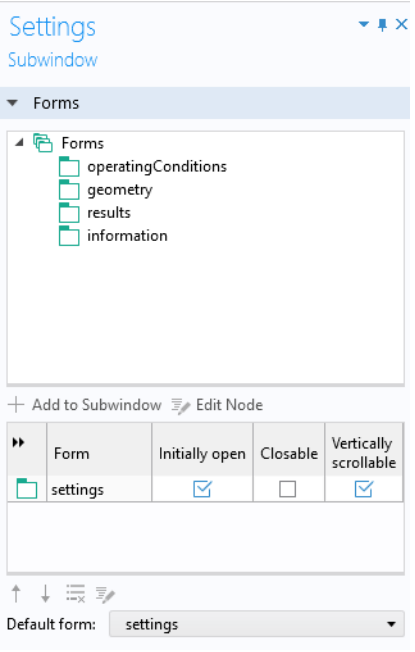
The column width and row height settings are not interpreted as absolute width and height, but rather summed together, and each individual row and column size is then interpreted as a fraction of the total size. A percentage value is displayed in the column and row headers in the **Subwindows** layout area.

The **Allow dragging between subwindows** setting controls whether forms can be dragged and moved between the different parts of the subwindow area.

The **Show form header in subwindows with a single form** setting controls whether to the forms, which are alone in a subwindow, should have a header. This setting is only available when dragging between subwindows is disabled.

Only one type of object can be placed in each of the grid cells used to define the window layout. This is called a subwindow object and is automatically added to a

grid cell as soon as it is created. The **Settings** window for a subwindow object is shown in the figure below.



This **Settings** window makes it possible to set which forms that are open in different subwindows. In the table below the model tree in the **Forms** section, you can specify if the form is **Initially open** in the subwindow, if it is **Closeable**, and if it is **Vertically scrollable**. A form that is **Closeable** will have a cross icon in the window title for closing it.

Since a form can only be shown in one subwindow, the **Forms** section, containing the available forms, is filtered to only show forms that have not yet been added to a subwindow. Also, the **Forms** section only shows global forms and not local forms.

When starting an application, each form that has been added to the **Forms** section, and where the check box in the **Initially open** column is selected, is shown in the corresponding subwindow. The **Title** of the form is used as the title on the corresponding tab.

The **Default form** setting specifies which form is initially active in the subwindow when launching the application.

Events

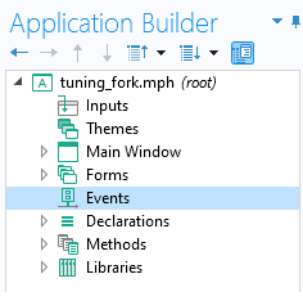
An event is any activity (for example, clicking a button, typing a keyboard shortcut, loading a form, or changing the value of a variable) that signals a need for the application to carry out one or more actions. Each action can be a sequence of commands of the type described earlier, or may also include 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.

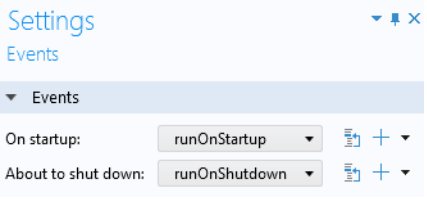
The events that initiate these actions may also be global or local. The global events are listed in the **Events** node of the application tree and include all events that are triggered by changes to the various data entities, such as global parameters or string variables. Global events can also be associated with the startup and shutdown of the application. In addition, **Timer** events can be used to periodically trigger events based on a set time interval (delay). 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, as well as command sequences, 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.

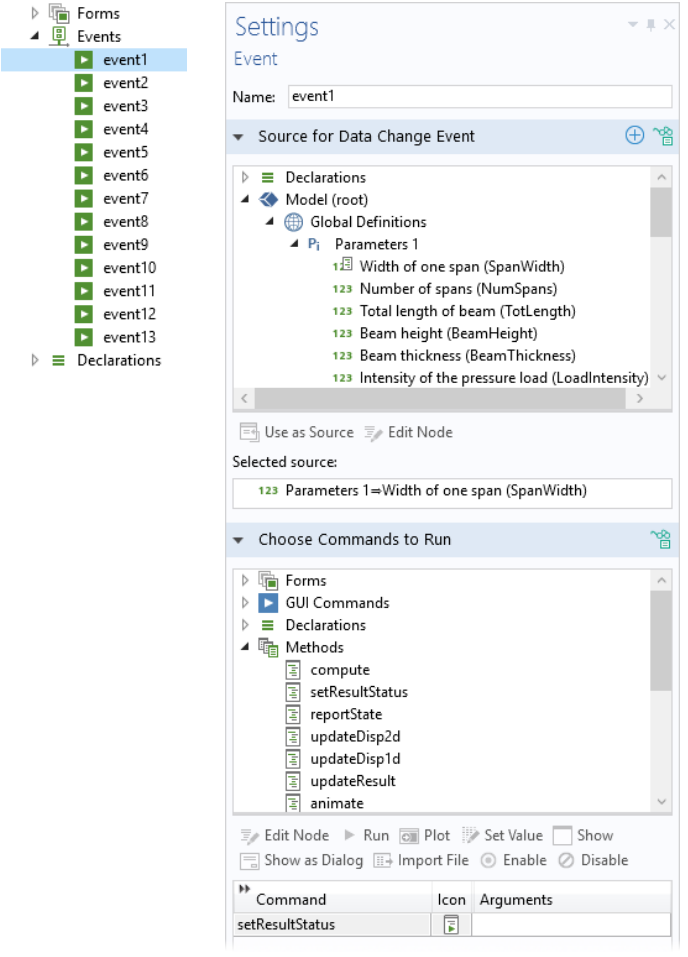
Limitations with On Startup Events

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.



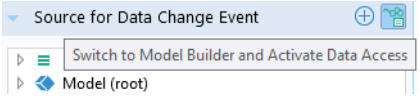
Note that since this type of event has global scope and is not associated with a particular form, the full path: `/form1/graphics1` needs to be used when referencing graphics objects.

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.

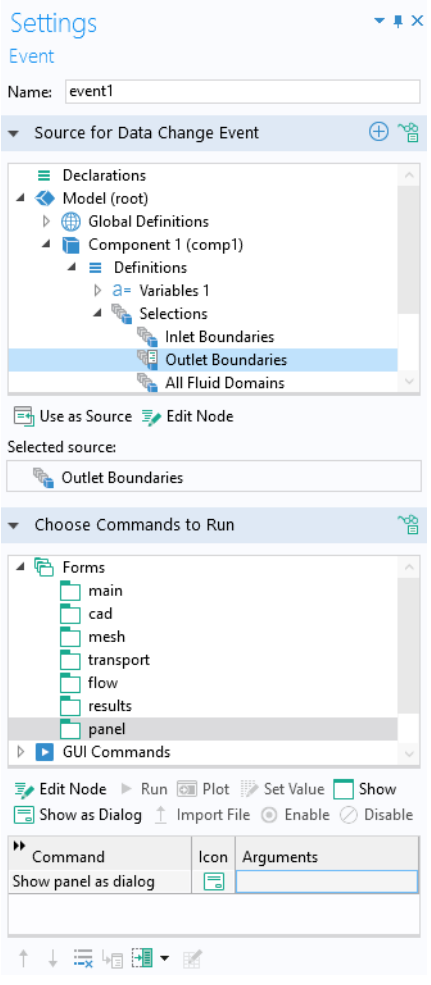
You can extend the list of available data nodes by clicking on the **Switch to Model Builder and Activate Data Access** button in the header of the section **Source For Data Change Event**.



For more information, see “Data Access in the Method Editor” on page 193.

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 Boundaries**.

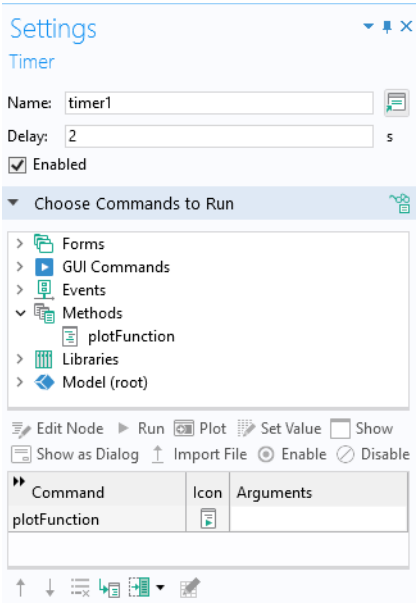


Choose Commands to Run

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 67.

Timer Events

Timer events adds a **Timer** that controls how often a command sequence is triggered. This functionality can be used to operate an app as a digital twin. Each time a delay has elapsed, a command sequence that you define is run. The command sequence can include method calls, as shown in the figure below.



You can use a **Timer** that accepts input from various external sources such as a hardware sensor, an external database, or a web service. Subsequently, it updates the app based on the acquired data.

In the **Delay** field, specify the delay time, in seconds, that the timer should wait before executing the command. The default delay is set to 1 second, determining the frequency at which the event is triggered.

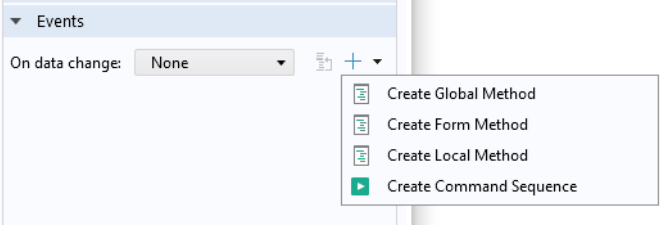
The **Timer** is enabled by default. If you want to disable the triggering of timer events, clear the **Enabled** box.

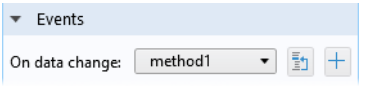
Form and Form Object Events

Form and form object events are similar to global events, but are defined for forms or individual form objects. These events can have an associated list of commands in a command sequence, or refer directly to a global, form, or local method.

Events Triggered by Data Change

For certain types of form objects, you can specify a method or command sequence 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 figures 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 or command sequence (optional).

The form objects supporting this type of event are:

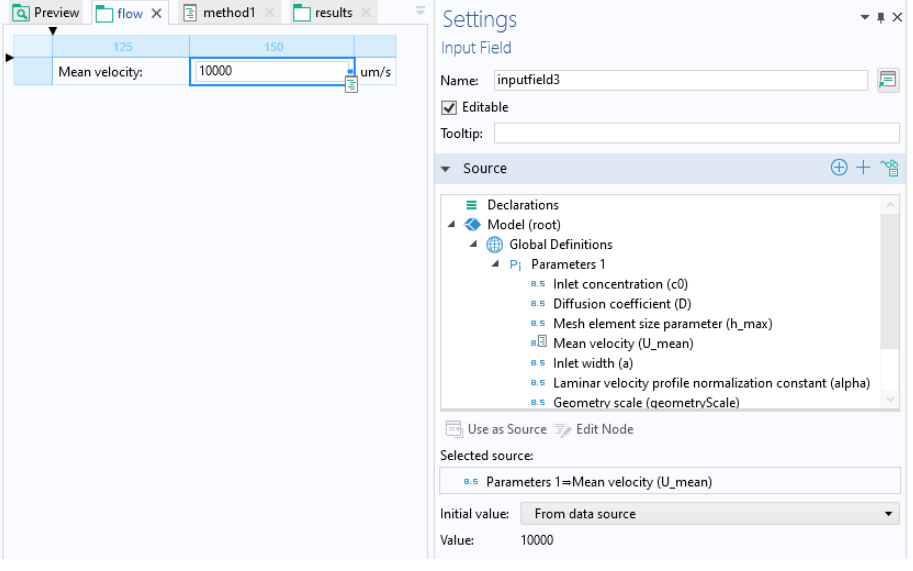
- **Input Field**
- **Check Box**
- **Combo Box**
- **Graphics**
- **File Import**
- **Array Input**
- **Radio Button**
- **Text**
- **List Box**
- **Table**
- **Slider**

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. Note, however, that such command sequences can also be used to run methods. For more information on command sequences, see “Button” on page 67.

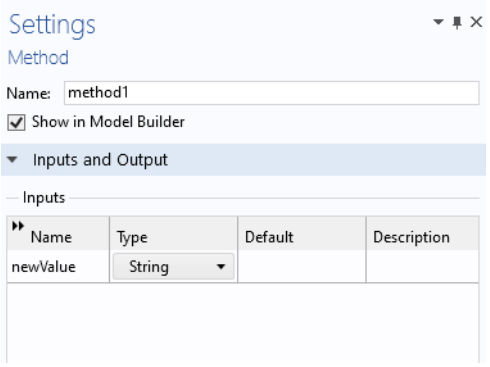
Input Arguments

Clicking **Create Global Method** or **Create Form Method** will open a dialog box where you can edit the name of the method to be created. Clicking **Create Local Method** will create a local method with an automatically created name indicating the form and type of event, see “Local Methods” on page 208. The method so created has a first argument called `newValue` which is based on the form object type. The argument added to the method will contain the new value in the form object when the event is triggered. The newly created method is also selected in the application tree and opened in the editor area.

For example, assume that you create a method based on an input field that links to a global parameter, as shown in the figure below.



The **Settings** window for the method created in this way is shown in the figure below.



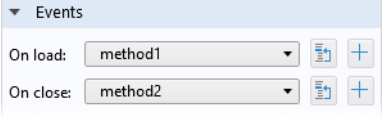
The variable `newValue` will then contain the new value of the **Mean velocity** entered by the user of the app. This functionality makes it possible to write method code for processing input data and provides a general way of checking the input data in an app.

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 or command sequence 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

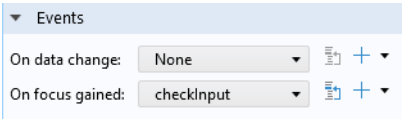
Forms can run methods or command sequences when they are loaded (**On load**) or closed (**On close**).



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.

Events Triggered by Focus

Certain form objects can have an **On focus gained** event, which is similar to an **On data change** event. This type of event is triggered when the user control associated with a form object comes into focus, for example, when clicking an input field. In general, a form object comes into focus when it becomes the target of keyboard input by the action of a mouse click or when cycling focus using the Tab-key.



The form objects supporting this type of event are:

- **Input Field**
- **File Import**
- **Text**

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 208.

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 217.

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.

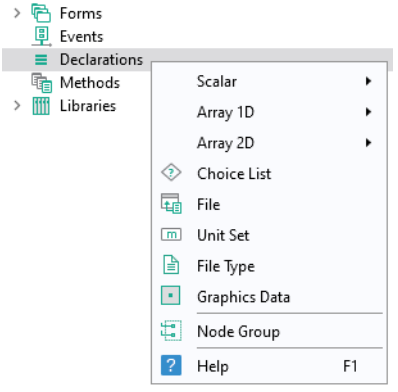
These are the different types of global **Declarations**:

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

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

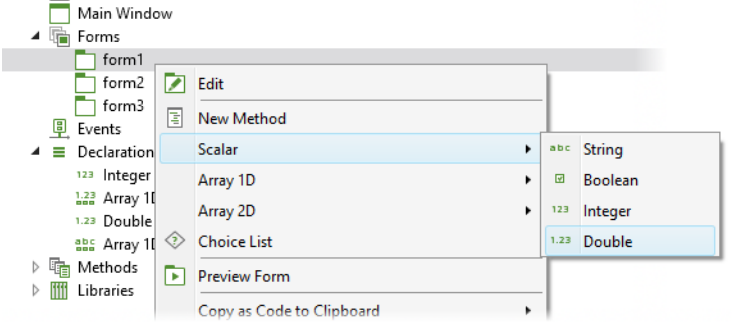
- **Scalar**
- **Array 1D**
- **Array 2D**
- **Choice List**

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

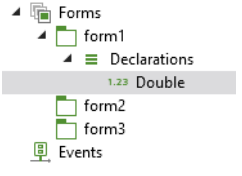


Note that **Shortcuts** are not created from this menu but by clicking the **Create Shortcut** button next to the **Name** in the **Settings** window of a form object or by using Ctrl+K for a selected form object.

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.

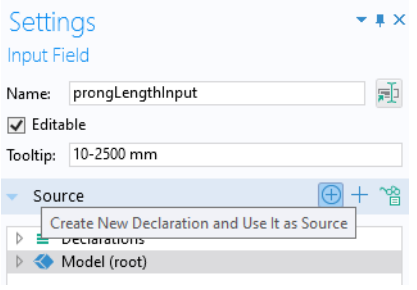


The first three types of declarations, **Scalar**, **Array 1D**, and **Array 2D**, can be of the following data types:

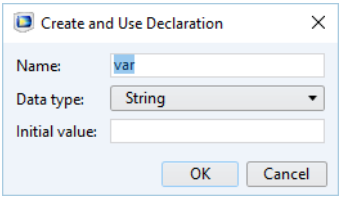
- **String**

- **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.



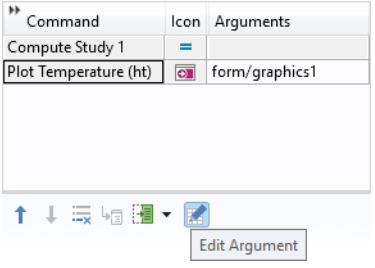
This will open a dialog box that lets you quickly declare scalar variables.



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 67.

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.

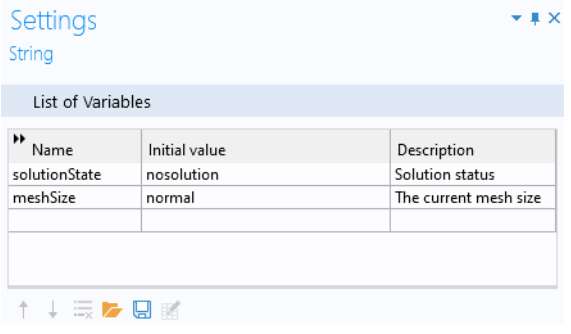
Scalar

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 366. 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 `solutionState` and `meshSize`.

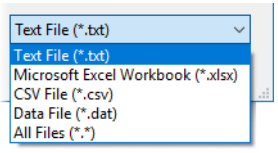


String declarations, as well as other declarations, can be loaded and saved from or to a file by using the **Load from File** and **Save to File** buttons below the **List of Variables** table.

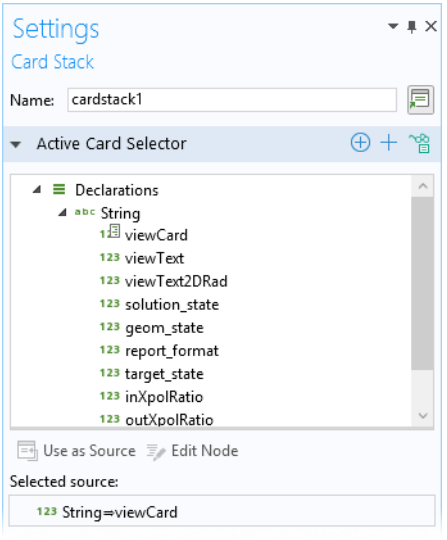
The **Load from File** and **Save to File** buttons are used to load and save from/to the following file formats:

- 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.



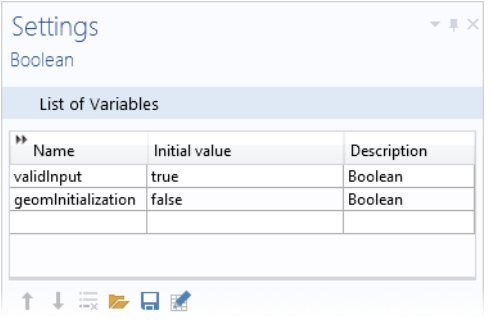
To illustrate the use of declared strings, the figure below shows the **Settings** window of a card stack object where the string variable `viewCard` is used as the source (**Active Card Selector**).



For more information on using card stacks, see “Card Stack” on page 286.

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.



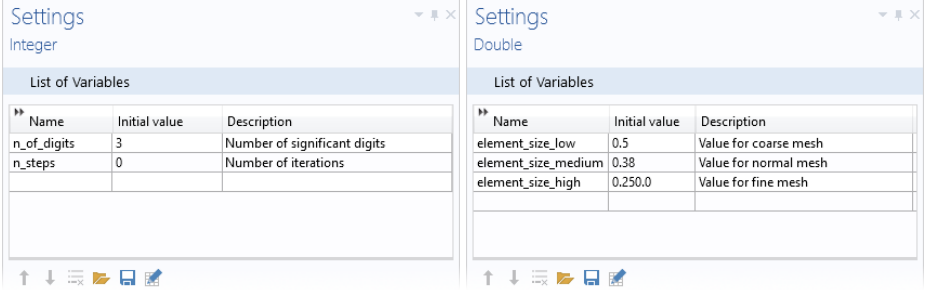
Example Code

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");
}
```

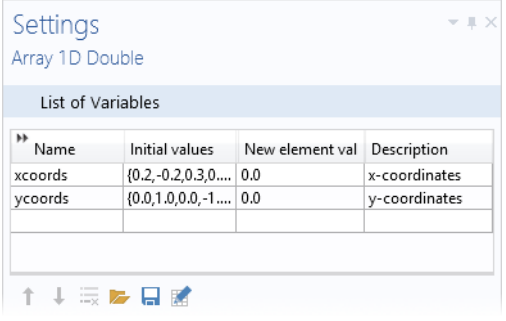
Integer and Double

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



Array 1D

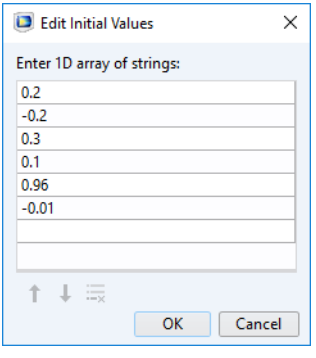
The **Array 1D** node declares one or more named arrays of strings, Booleans, integers, or doubles that you can access from form objects and methods. The number of elements in a 1D array is not restricted in any way, and you can, for example, use a 1D array to store a column in a table with a variable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



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

Initial Values

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



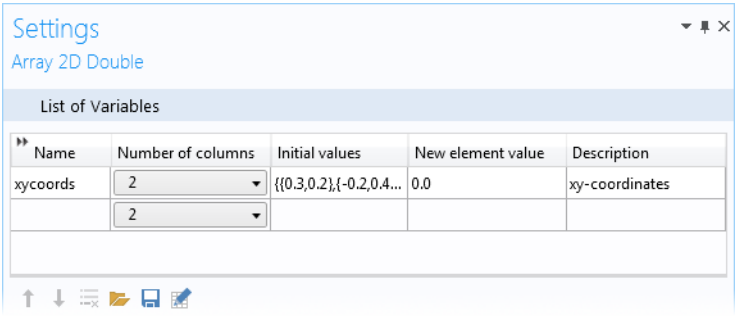
Array Syntax

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

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

Array 2D

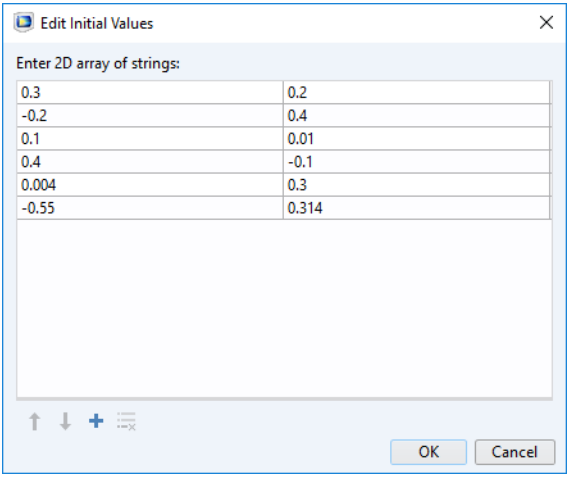
The **Array 2D** node declares one or more 2D arrays that you can access using form objects and methods. In the figure below, the 2D double array `xycoords` is declared.



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
`{{}}`	An empty 3D array
`{{'5','6'},{'7','8'}}`	A 2-by-2 matrix of strings
`{{1, 2, 3}, {4, 5, 6}}`	A 2-by-3 matrix of doubles

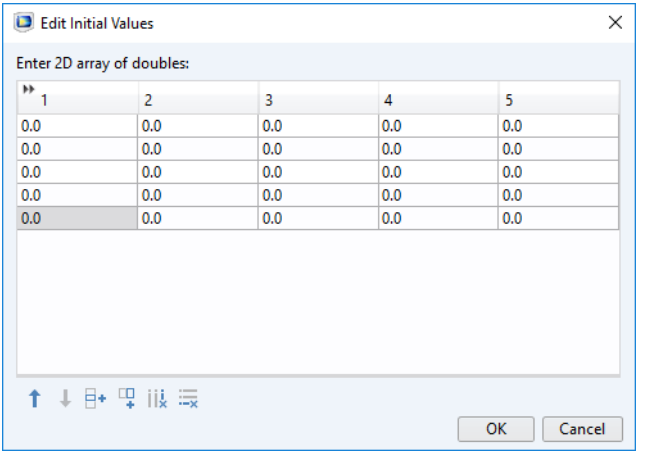
For 2D arrays, rows correspond to the first index so that {{1,2,3},{4,5,6}} is equivalent to the matrix:

```
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

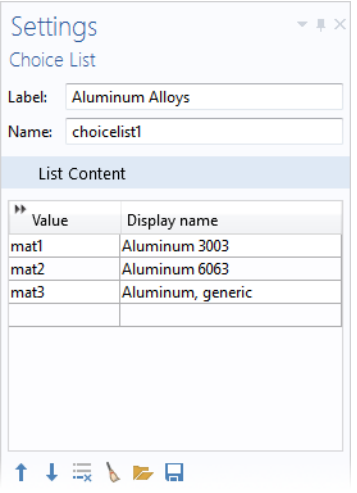
the number of rows and columns can be interactively changed, as shown in the figure below.



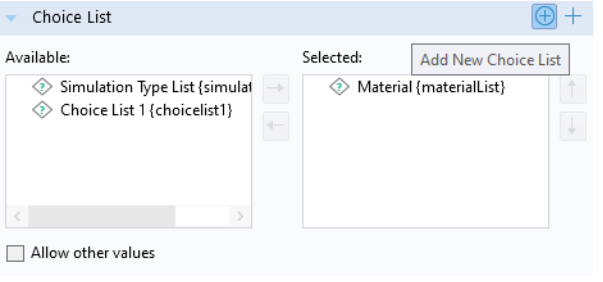
Choice List

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.



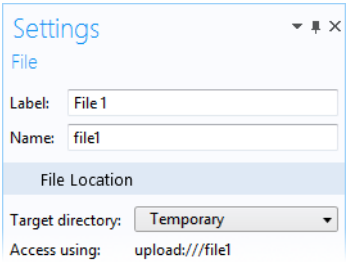
In addition you can click the adjacent **Add New Form Choice List** to create a choice list local to the form.

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 253.

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 354. 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.



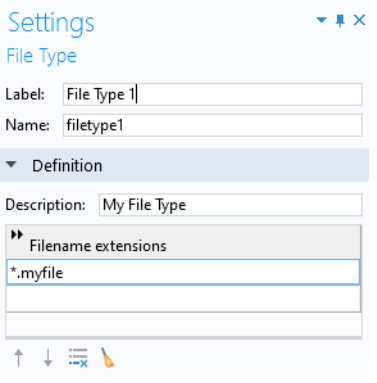
The file chosen by the user can be referenced in a form object or method using the syntax `upload:///file1`, `upload:///file2`, etc. The file name handle (`file1`, `file2`, etc.) can then be used to reference an actual file name picked by the user at run time.

For more information on file declarations and file handling, see “Appendix C — File Handling and File Scheme Syntax” on page 328.

File Type

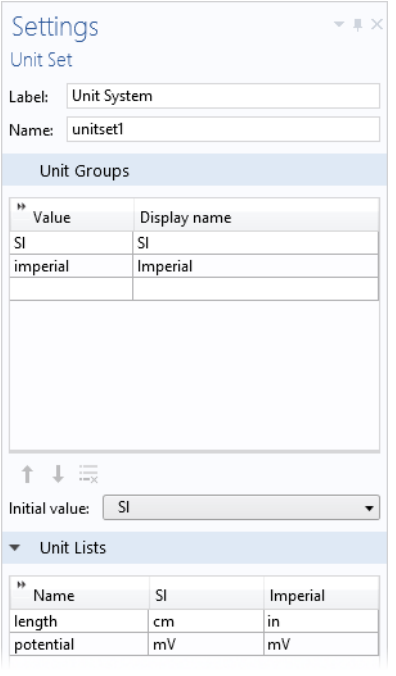
The **File Type** declaration lets you use a custom file type with a custom file extension. The custom file declaration will then become available in the **File Import**

form object as well as in the dialog boxes shown when using the methods `importFile` and `fileSaveAs`.



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**.



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, and 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.

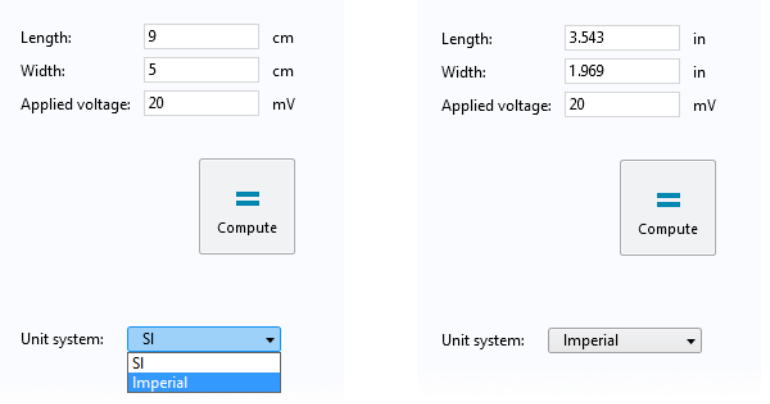
The figure above demonstrates the use of a **Unit Set** for an application that allows for switching between metric and imperial units. In this example, two unit groups are defined: `SI` and `Imperial`. The **Label** of the **Unit Set** has been changed to `Unit System`.

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**

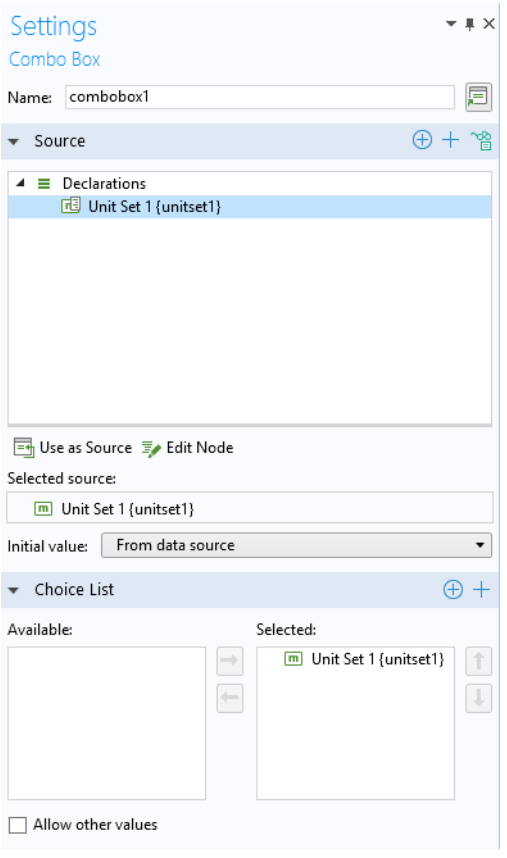
column is the string displayed in the user interface. The **Initial value** list contains the default unit group (`SI` in the example above).

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.

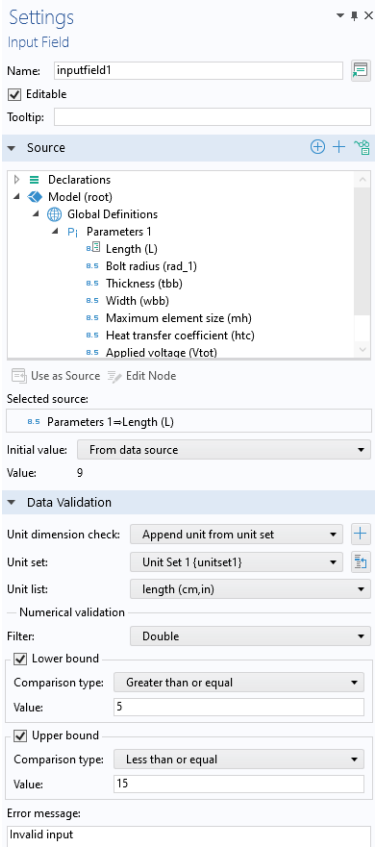


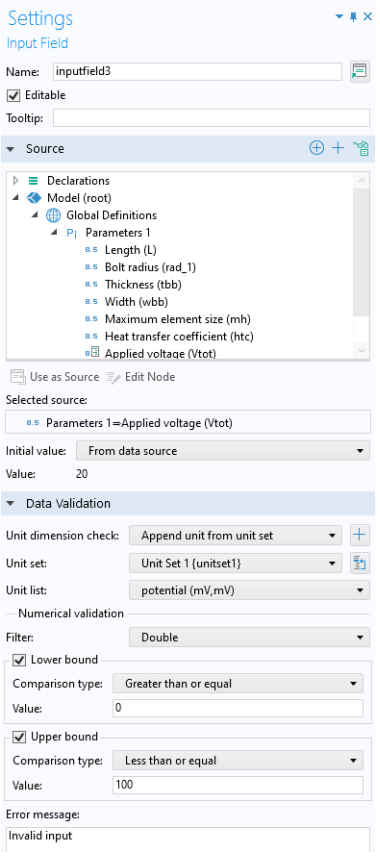
The figure below shows the **Settings** window of a combo box using the **Unit Set** of the above example as the **Source**.



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.

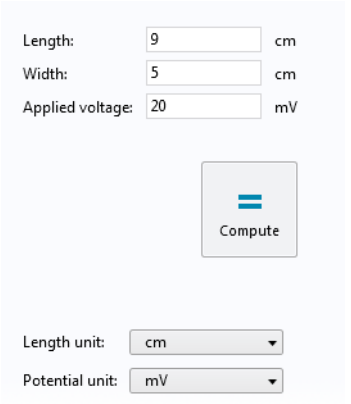
The two figures below show the corresponding **Settings** windows for the two input fields for **Length** and **Applied voltage**.

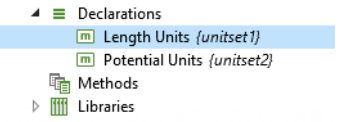




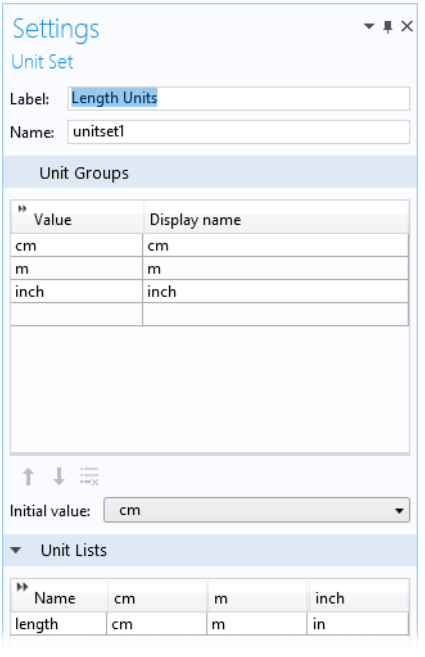
The **Unit dimension check** is set to **Append unit from unit set**. The **Unit set** is set to **Unit System {unitset1}** (the user-defined label for the **Unit Set** declaration used in this example). The **Unit list** is set to **length** and **potential**, respectively. When using **Append unit 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 **Lower bound** and **Upper bound** 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 97.

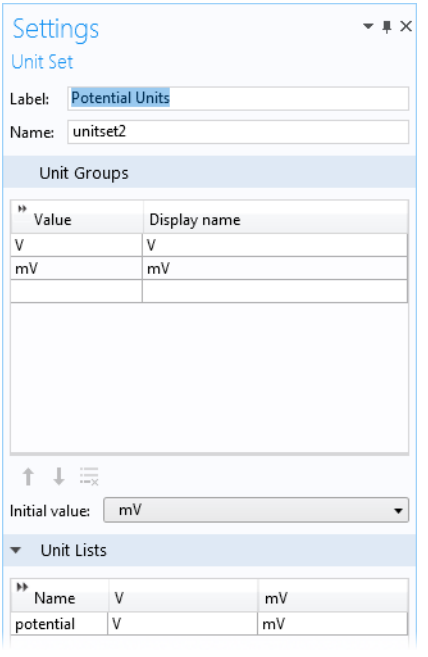
The figures below illustrate the use of two **Unit Set** declarations for separately setting the unit for length and potential, respectively.





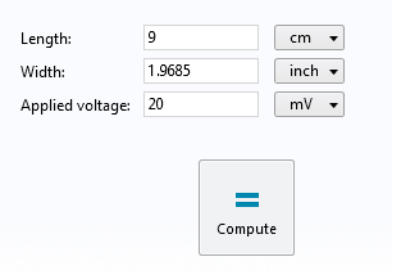
The figures below show the corresponding **Settings** window for the **Unit Set** declarations.





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.

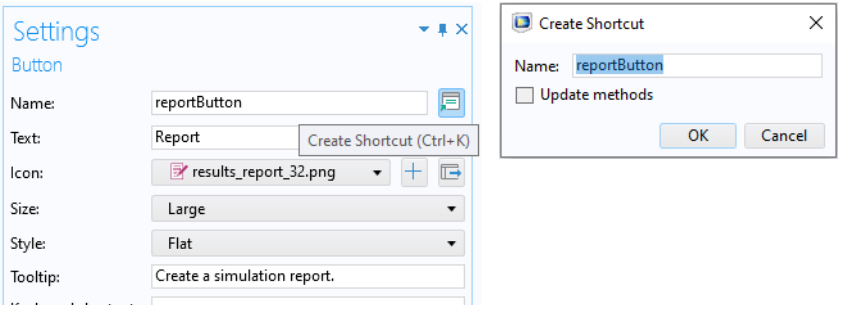


When more flexibility is required, you can combine the use of a **Choice List** and a **Unit Set**. For example, for a combo box, you can use the **Unit Set** as the **Selected source** (string) and select a **Choice List** that is not a **Unit Set**.

Shortcuts

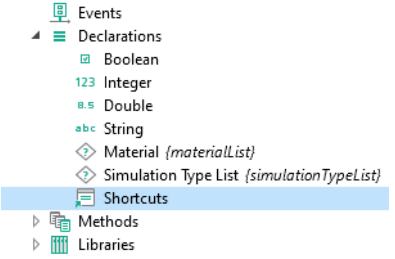
Form objects and other user interface components are referenced in methods by using a certain syntax. For example, using the default naming scheme `form3/button5` refers to a button with the name `button5` in `form3` and `form2/graphics3` refers to a graphics object with the name `graphics3` in `form2`. 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.

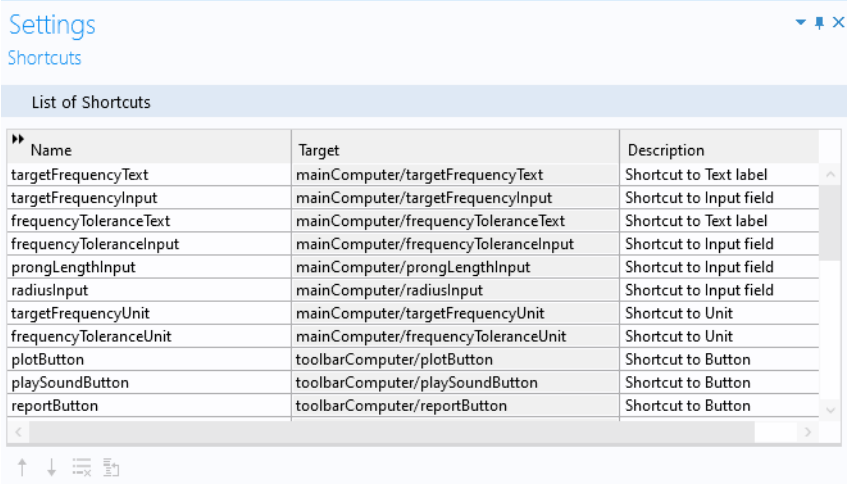


To create or edit a shortcut, you can also use the keyboard shortcut Ctrl+K.

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

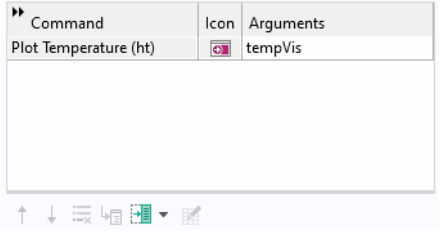


In the **Settings** window for **Shortcuts** below, two shortcuts, `plot_temp` and `temp_vis`, have been created for a button and a graphics object, respectively.



Name	Target	Description
targetFrequencyText	mainComputer/targetFrequencyText	Shortcut to Text label
targetFrequencyInput	mainComputer/targetFrequencyInput	Shortcut to Input field
frequencyToleranceText	mainComputer/frequencyToleranceText	Shortcut to Text label
frequencyToleranceInput	mainComputer/frequencyToleranceInput	Shortcut to Input field
prongLengthInput	mainComputer/prongLengthInput	Shortcut to Input field
radiusInput	mainComputer/radiusInput	Shortcut to Input field
targetFrequencyUnit	mainComputer/targetFrequencyUnit	Shortcut to Unit
frequencyToleranceUnit	mainComputer/frequencyToleranceUnit	Shortcut to Unit
plotButton	toolbarComputer/plotButton	Shortcut to Button
playSoundButton	toolbarComputer/playSoundButton	Shortcut to Button
reportButton	toolbarComputer/reportButton	Shortcut to Button

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.

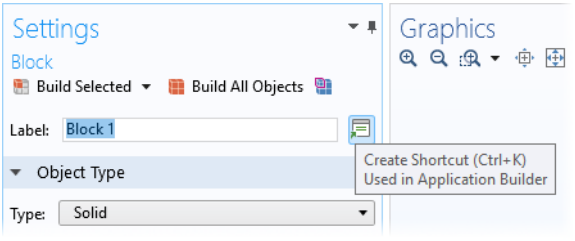


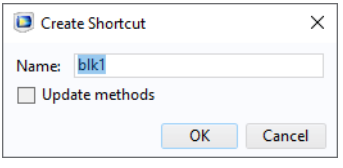
Command	Icon	Arguments
Plot Temperature (ht)		tempVis

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.

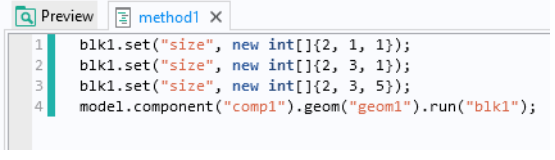
Using shortcuts is recommended because it avoids the need to adjust Method Editor code when the structure of the application user interface changes.

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.



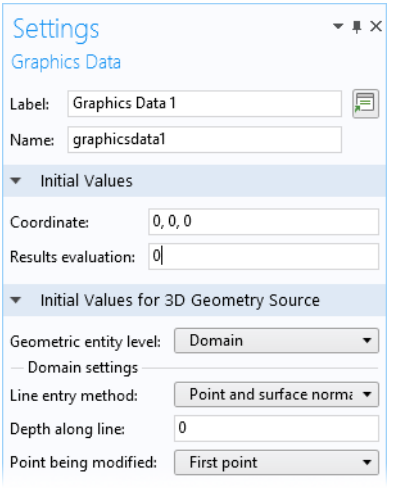


The custom name of a shortcut becomes available as a global variable in methods and will be used, for example, when recording code or new methods, as shown in the figure below.



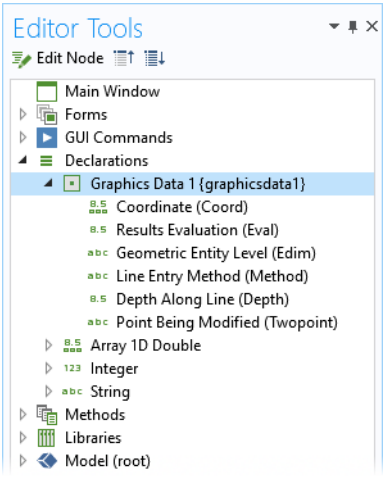
Graphics Data

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.

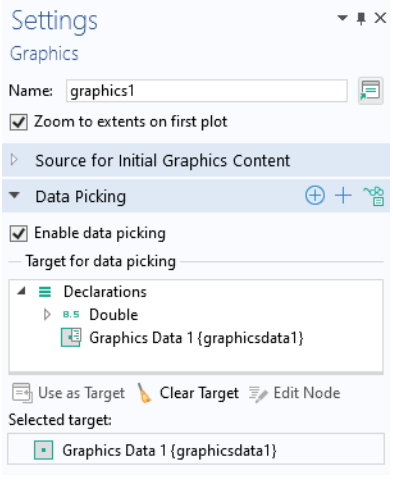


The **Initial Values** section contains default values for the extracted data properties **Coordinate** and **Results evaluation**. The section **Initial Values for 3D Geometry Source** contains settings for the selection methods available when the **Source for Initial Graphics Content** of a graphics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

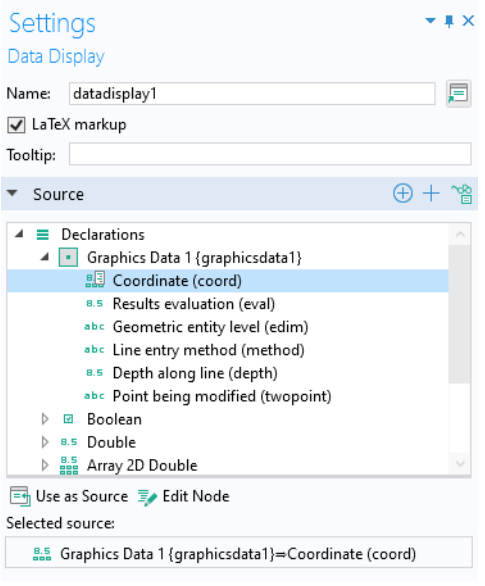


Graphics Data from Results

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. The Method Editor supports Java SE 6 syntax. 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 about these built-in methods can be found in "Appendix E — Built-In Method Library" on page 353 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 208.

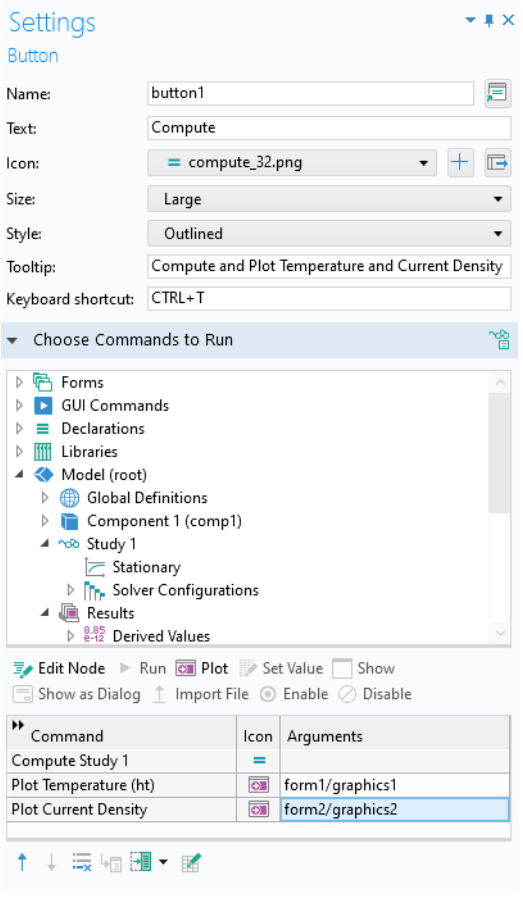
A number of tools and resources are available to help you create code for methods. These are covered in the following sections and will make you more productive by allowing you to copy-paste or autogenerate blocks of code, for example.

Converting a Command Sequence to a Method

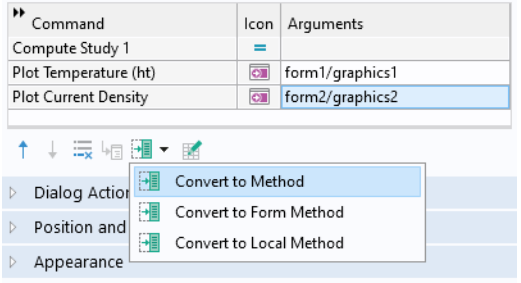
In the Form Editor, select **Convert to Method** from the menu button displayed in the **Settings** window below an existing command sequence. The command sequence is automatically replaced by an equivalent method. In the same way you can select **Convert to Form Method** and **Convert to Local Method**.

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 (this can also be done without writing code, see later in this section).

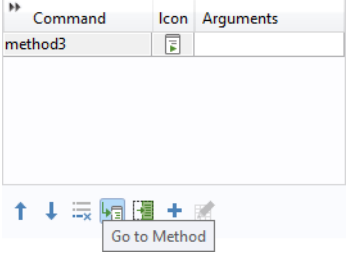
You will also learn how to do this without using the Method Editor later in this section. The figure below shows the **Settings** window of the **Compute** button.



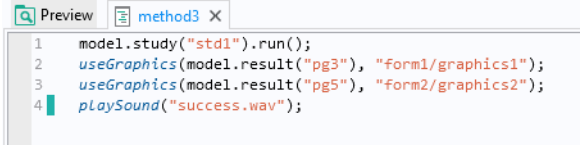
Select the **Convert to Method** option below the command sequence.



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.

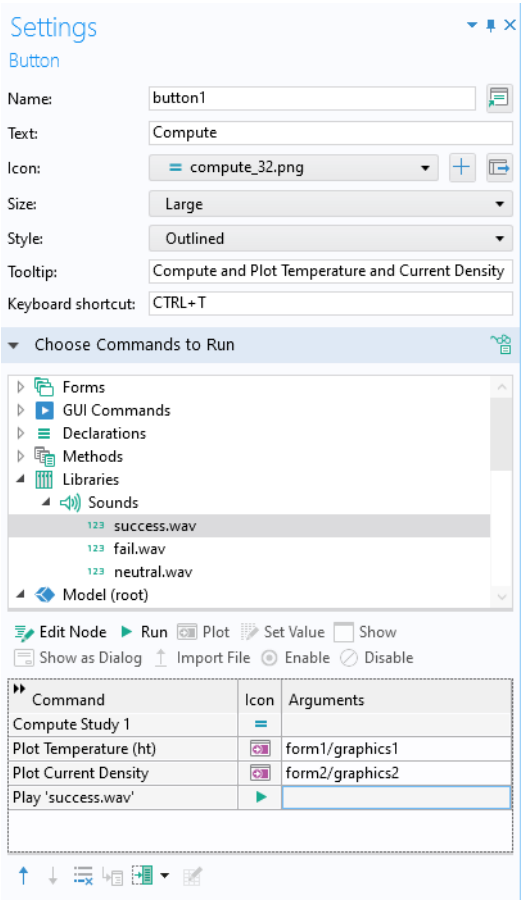


In the Method Editor, add a call to the built-in method `playSound` to play the sound file `success.wav`, available in the COMSOL sound library, by using the syntax shown in the figure below.



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

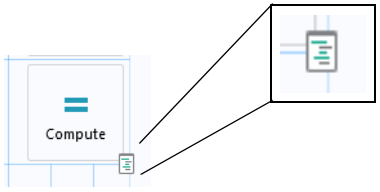
Note that in the example above, you do not have to use the Method Editor. In the command sequence, select the file `success.wav` under **Libraries > Sounds** and click the **Run** command button under the tree, as shown in the figure below.



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 353.

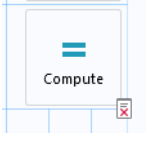
Form Object with Associated Method

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.



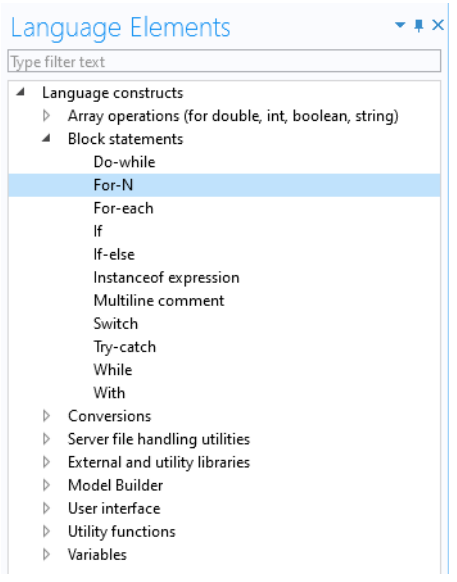
Performing Ctrl+Alt+Click on the form object opens the method in the Method Editor. If there is no method associated with the form object, a new local method associated with the form object will be created and opened in the Method Editor.

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

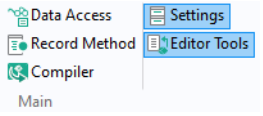
The **Language Elements** window in the Method Editor shows a list of some language constructs. Double-click or right-click one of the items in the list to insert template code into the selected method.



See also “Language Element Examples” on page 215.

Editor Tools in the Method Editor

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

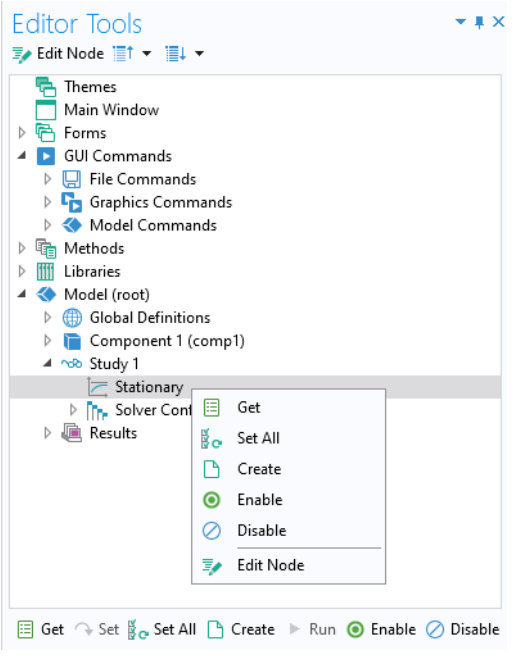


When using the **Editor Tools** window in the Method Editor, you can right-click a node in the editor tree to generate code associated with that node. Depending on the node, up to eight different options are available:

- **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 65.

Keyboard Shortcuts

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

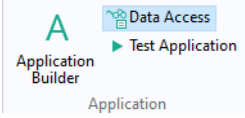
```
model.result("pg3").feature("surf1").create("hght1", "Height");
```

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 350.

Data Access in the Method Editor

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

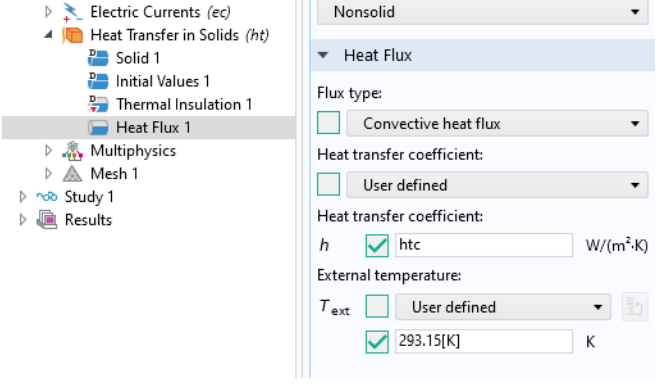


Alternatively, for certain form objects, you can click the **Data Access** button in the header of the **Source** section of the **Settings** window. See also “Data Access in the Form Editor” on page 108.

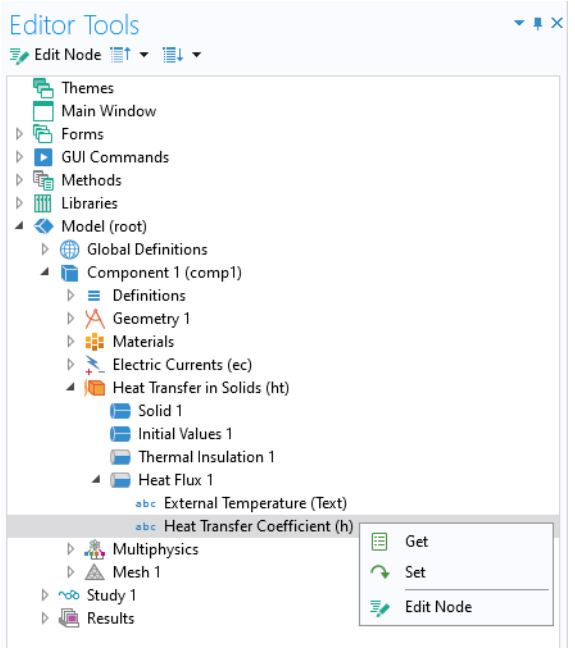
Data Access needs to be enabled this way because a model typically contains hundreds or even thousands of properties that could be accessed, and the list would be too long to be practical.

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.



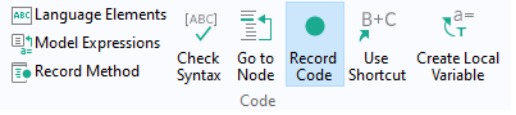
In the example above, **Get** and **Set** for the **Heat transfer coefficient** and the **External temperature** properties will generate the following code:

```
model.component("comp1").physics("ht").feature("hf1").getString("Text");
model.component("comp1").physics("ht").feature("hf1").getString("h");

model.component("comp1").physics("ht").feature("hf1").set("Text",
"293.15[K]");
model.component("comp1").physics("ht").feature("hf1").set("h", "htc");
```

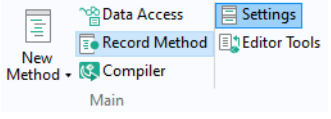
Recording Code

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.



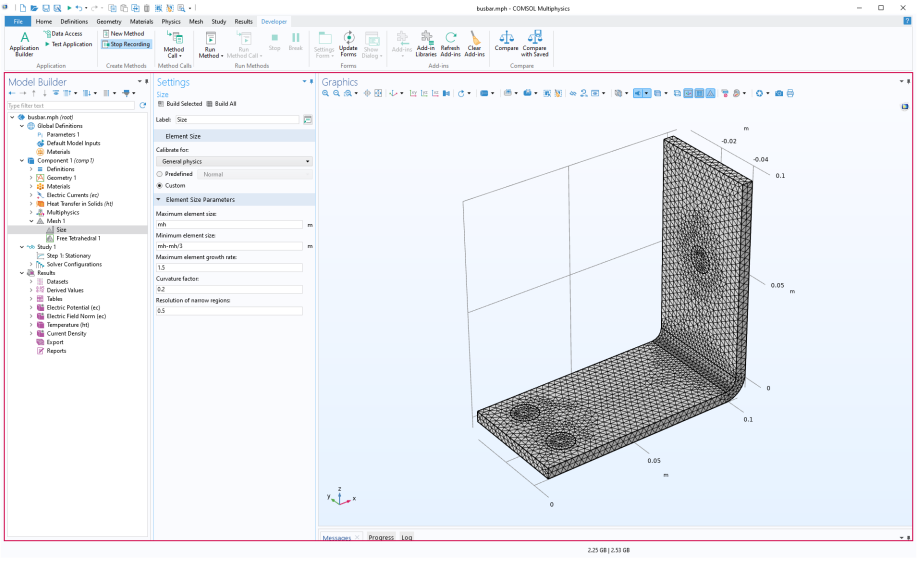
Certain operations in the application tree can also be recorded, including methods used to modify the user interface while the application is running such as changing the color of a text label.

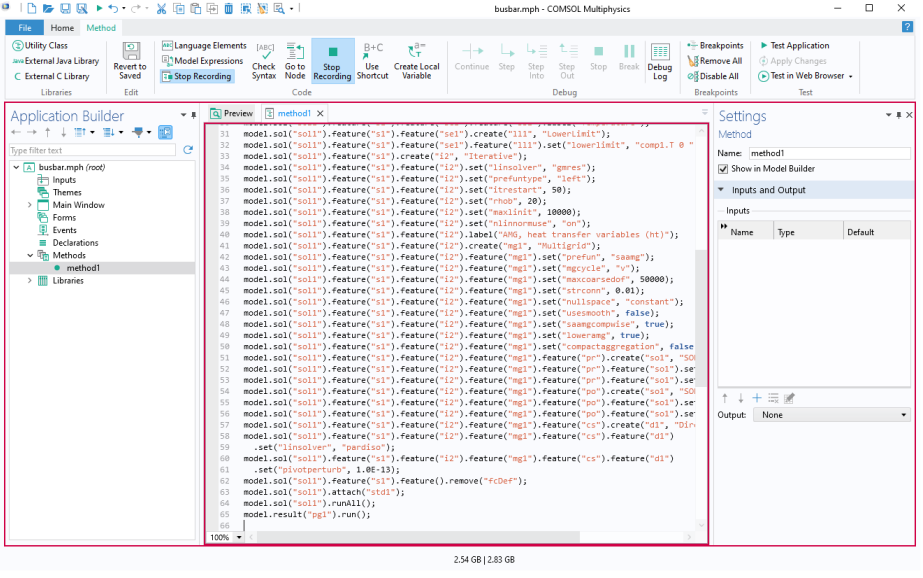
To record a new method, click the **Record Method** button. This button is also available in the **Main** section of the **Home** tab of the Form Editor or Method Editor ribbon.



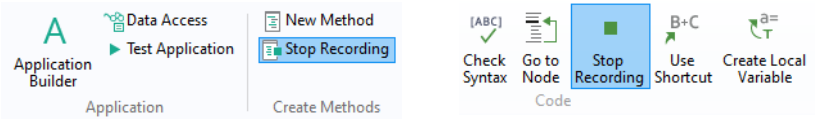
In addition, you can click the **Record Method** button in the **Developer** tab of the Model Builder ribbon.

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) available in the Application Libraries under COMSOL Multiphysics>Multiphysics.
- 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:

```
model.component("comp1").physics("ht").feature("hf1").set("h", 5);
model.component("comp1").physics("ht").feature("hf1").set("Text",
"300[K]");
```

Note that in **File>Preferences** you can enable the use of `with()` statements in order to make the code more compact. For more information on the use of `with()`, see "The With Statement" on page 217.

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**:

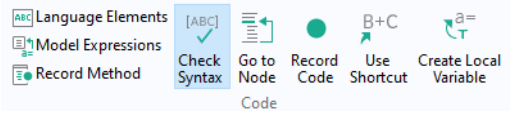
```
app.form("form1").formObject("textlabel1").set("foreground", "blue");
```

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

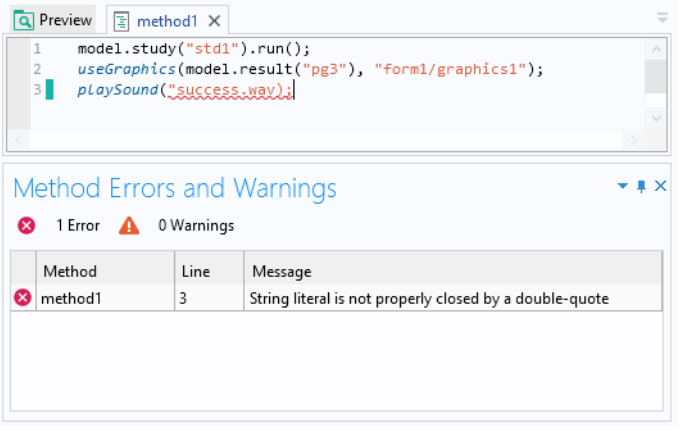
Use the tools for recording code to quickly learn how to interact with the model object or the application object. The autogenerated code shows you the names of properties, parameters, and variables. Use strings and string-number conversions to assign new parameter values in model properties. By using **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 193.

Checking Syntax

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

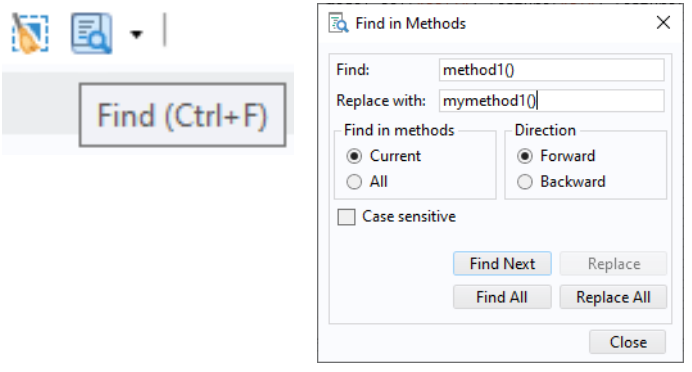


In addition to messages in the **Method Errors and Warnings** window, syntax errors are indicated with a wavy red underline, as shown in the figure below.



Find and Replace

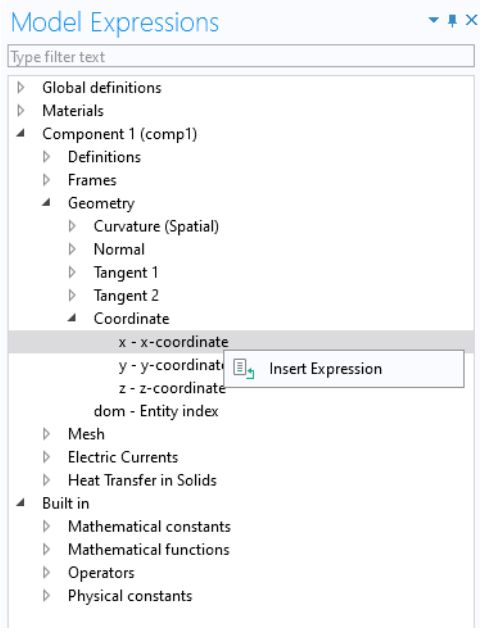
Click **Find** in the Quick Access Toolbar, or user the keyboard shortcut Ctrl+F, to open a dialog box used to find and replace strings in methods, as shown in the figure below.



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

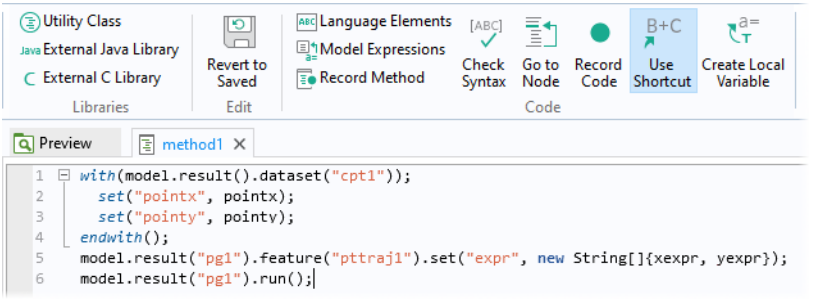
Model Expressions Window

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

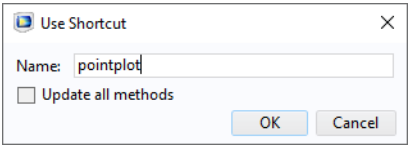
If you look at the example below, you will notice that the two last lines of code begins with `model.result("pg1")`.



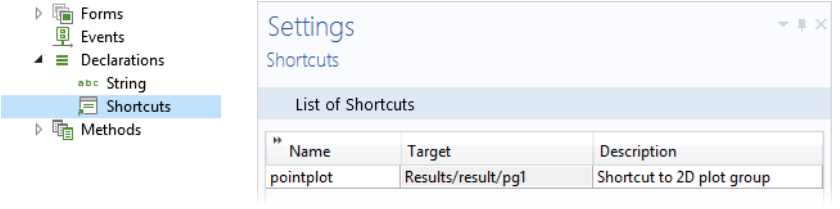
The **Use Shortcut** button simplifies code by replacing these instances with a variable name.

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.

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:

```
52  with(model.result("pg1"));
53    set("looplevel", new String[]{"7"}); // The first real eigenfrequency always
54  endwith();
55  model.result().numerical("gev1").setResult();
56
57  // Plot the solution in the graphics for the selected device
58  String graphics = "";
59  if (device.equals("computer")) {
60    graphics = "/mainComputer/graphics1";
61  }
62  else if (device.equals("tablet")) {
63    graphics = "/mainTablet/graphics1";
64  }
65  else if (device.equals("smartphone")) {
66    graphics = "/mainSmartphone/graphics1";
67  }
68  useGraphics(model.result("pg1"), graphics);
69  zoomExtents(graphics);
70
71  isGeometryActive = false;
72
73  setProgress(100);
74  if (isPlaySound) {
75    playSoundForFrequency();
76  }
77
```

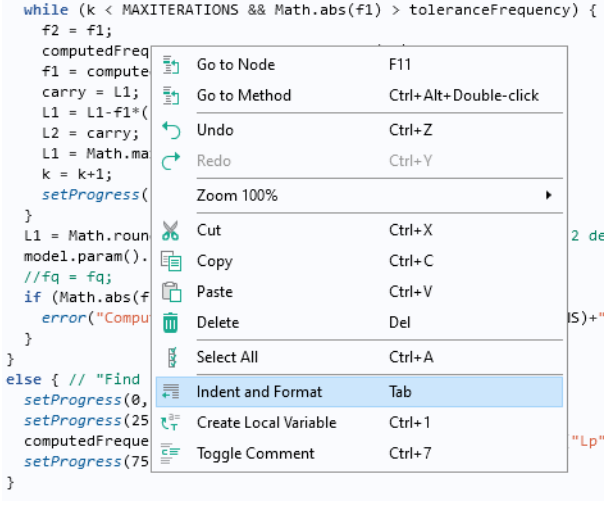
This example includes five styles:

- 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**.

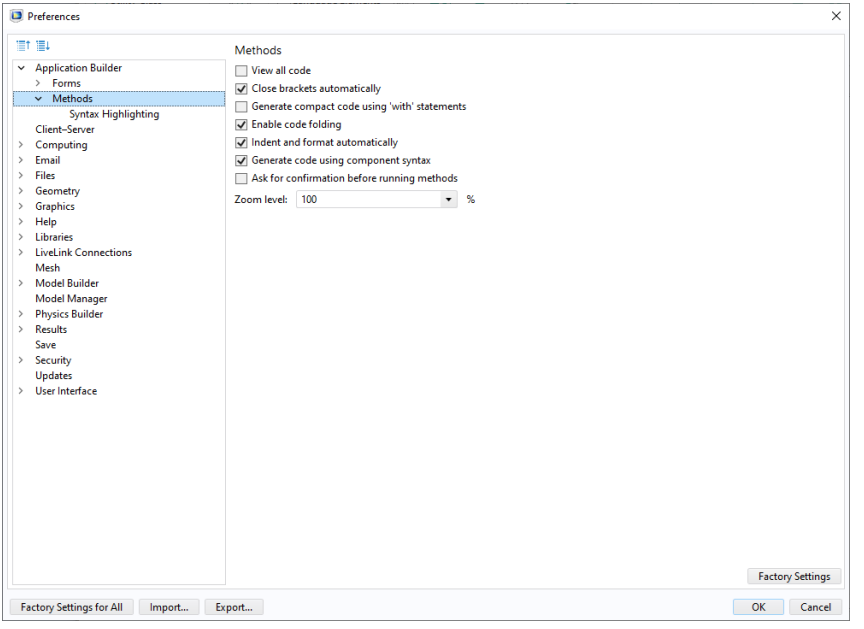
Using the context menu shown above, when right-clicking, you can toggle comments on and off for an entire block of code that you have selected. This is available by selecting **Toggle Comment** from the menu or the keyboard shortcut Ctrl+7.

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** node in the tree.



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 217.

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.

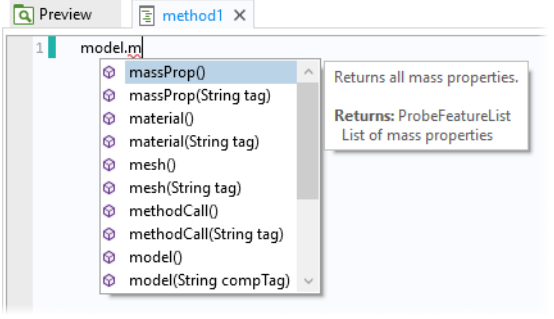
The **Generate code using component syntax** option will generate method syntax that also includes the model component scope.

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.

Under the **Syntax highlighting** node, the **Theme** list contains a few predefined themes. Choose **User defined** to define a syntax highlighting mode where the colors can be assigned to individual language elements.

Ctrl+Space and Tab for Code Completion

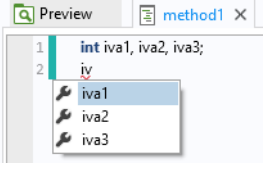
While typing code in the Method Editor, the Application Builder can provide suggestions for code completions. The list of possible completions are shown in a separate completion list that opens while typing. In some situations, detailed information appears in a separate window when an entry is selected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



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

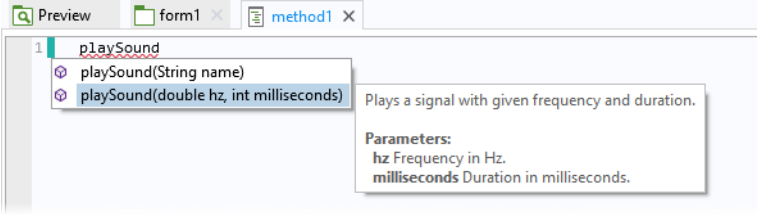
If the list is long, you can filter by typing the first few characters of the completion you are looking for.

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



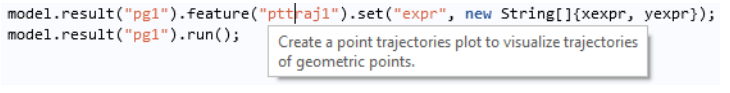
In the example above, only variables that match the string `iv` are shown. This example shows that variables local to the method also appear in the completion suggestions.

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.

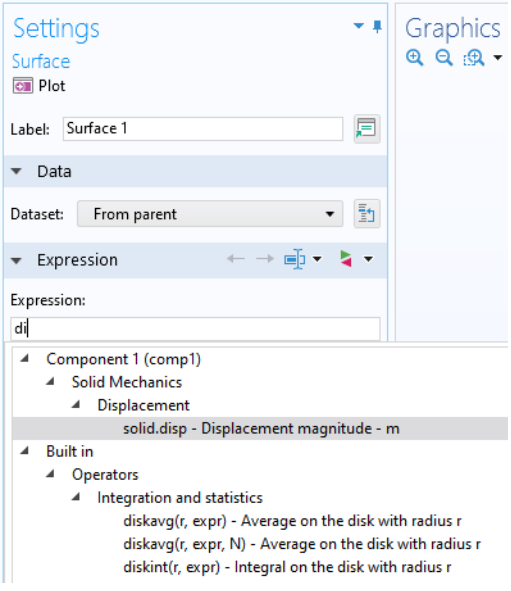


For a list of available built-in methods, you can use the **Language Elements** window described on page 190 or see “Appendix E — Built-In Method Library” on page 353.

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.

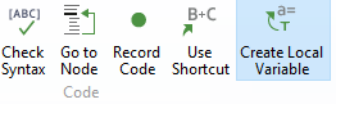


Creating Local Variables

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.



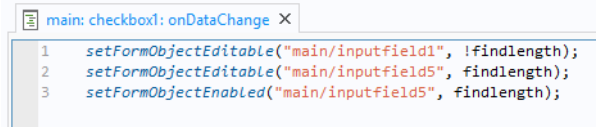
The code is then changed to

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

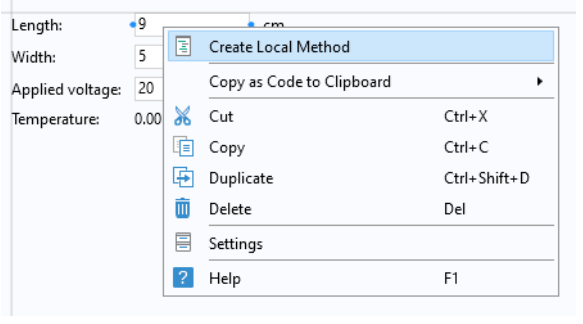
where `GeomList` is the data type of `model.geom()`.

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.

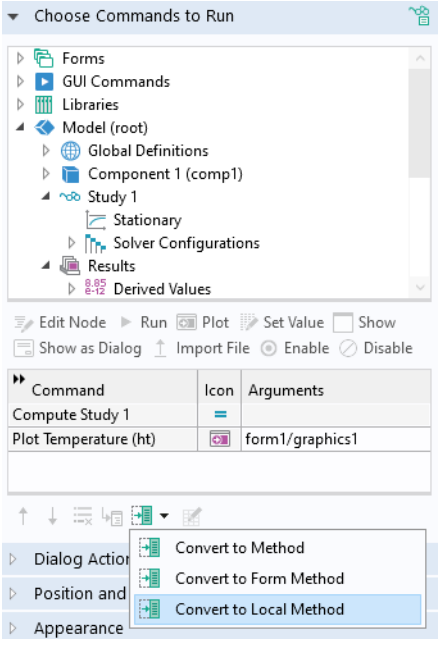


In the Form Editor, you can right-click a form object and select **Create Local Method** from a menu, as shown in the figure below.

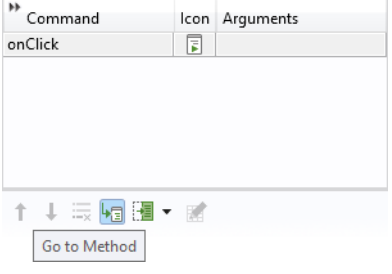


Local Methods for Buttons, Menu Items, and Global Events

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.



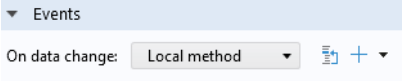
The function of this button is similar to the **Convert to Method** and **Convert to Form Method** buttons, described in the section "Creating a New Method" on page 18. The only difference is that it creates a local method not visible in the global method list in the application tree. It also opens the 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.

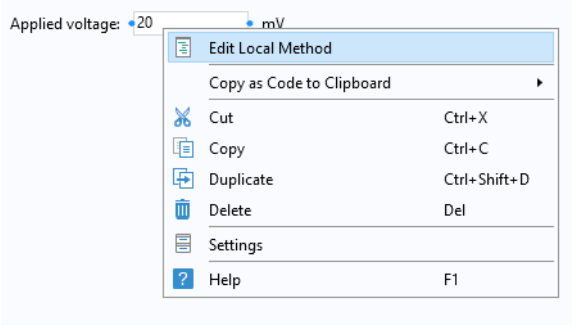
Local Methods for Form and Form Object Events

To add a local method for a form or form object event, select the **Create Local Method** option 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. Select **Remove Local Method** from the menu to delete the local method.

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

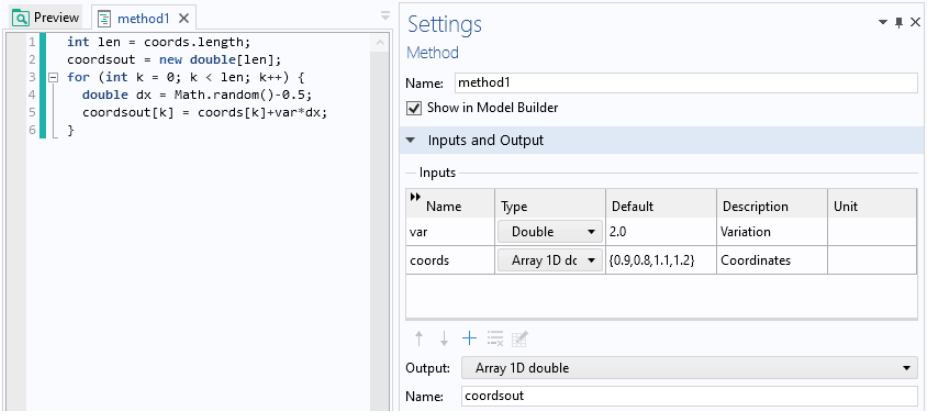


For more information, see “Events” on page 150.

Methods with Input and Output Arguments

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

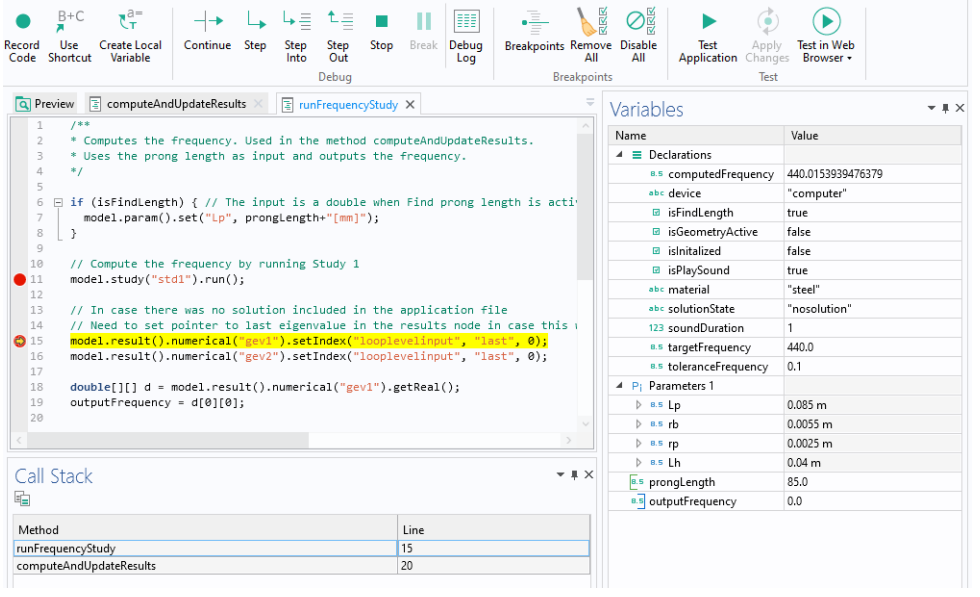
the array, `coords`. The degree of randomness is controlled by the input variable `var`. The new values are stored in the array `coordsout`.



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

Debugging

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

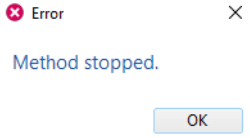


In the ribbon, the **Debug** group contains the tools available for debugging methods. When you run the application, the method will stop at the breakpoints. Click the **Step** button to go to the next line in the method. The figure above shows a method currently stopped at the line highlighted in yellow. The **Variables** window, seen to the right in the figure above, opens when a method is paused at a breakpoint. Using the **Variables** window, you inspect and modify local variables, declarations, and parameters, while debugging.

The **Call Stack** window, below the method window, as seen in the figure above, also opens when a method is paused at a breakpoint. You can select a method in the call stack see the corresponding variables in the **Variables** window.

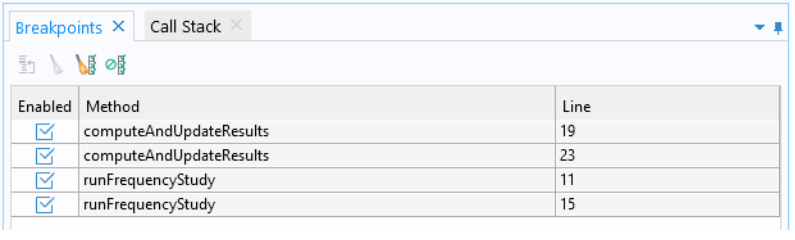
Click **Continue** to run the method up until the next breakpoint. Click **Stop** to stop running the method and exit. You can also stop the execution of a method while

testing an application by using the keyboard shortcut Ctrl+Pause. A dialog box appears, as shown below.

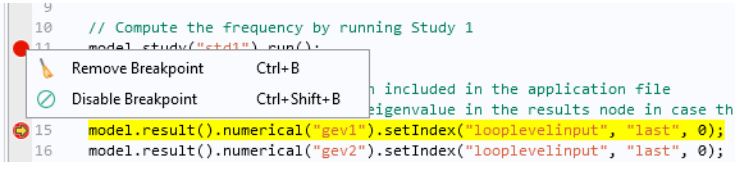


Click **Break** to suspend method execution at the next reached line in method code, as if there had been a breakpoint there. Click **Step Into** to step into the next method, if possible. The option **Step Out** continues until the current method returns. Use **Remove All** to remove all break points. Instead of removing, you can disable all break points by clicking **Disable All**.

To get an overview of all breakpoints, you can open the **Breakpoints** window, as seen in the figure below. To open this window, click the corresponding button in the **Method** tab, in the ribbon section **Breakpoints**.



You can enable and disable individual breakpoints, either from the **Breakpoints** window or by right-clicking a breakpoint in the gray column to the left of the code line numbers, as shown below.



Click the **Debug Log** to display customized debugging messages in a separate **Debug Log** window, as shown in the figure below.

As an alternative to using the **Variables** window, you can use the `debugLog` command a 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 362 and the *Application Programming Guide*.

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

PRECEDENCE LEVEL	SYMBOL	DESCRIPTION
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
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, with customized code in between the calls to Study 1 and Study 2.

From the Model Builder you can call methods directly through **Method Calls** or using the options available in 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 132.

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.

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 223.

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"));
```

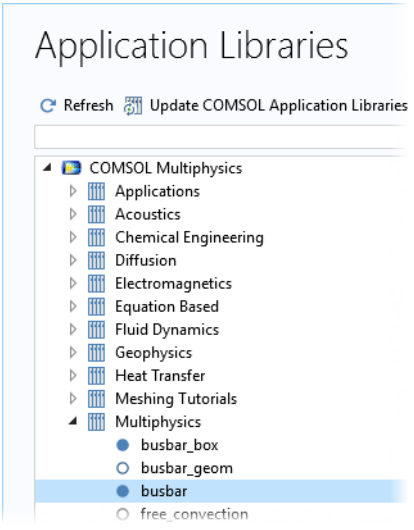
which will display the geometry and select the **Geometry** node.

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

Generating a Report Automatically after Computing

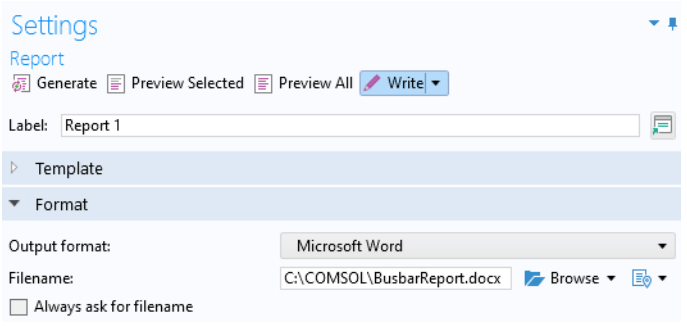
As an example of using a method from the Model Builder, consider the process of first computing the solution and then generating a report. This can be automated by first recoding the corresponding operations in the Model Builder and then running a method.

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.

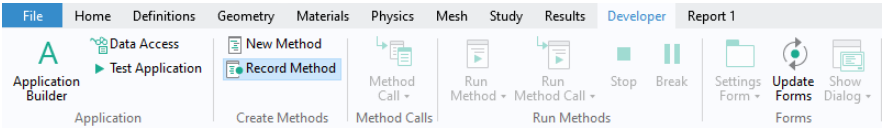


- **Open** the model.
- In the model tree, right-click the **Reports** node under **Results**.
- Select **Brief Report**.
- Change the **Output format** to Microsoft® Word (this example would also work with the default HTML format).
- Click the **Browse** button and select a file name in a location on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

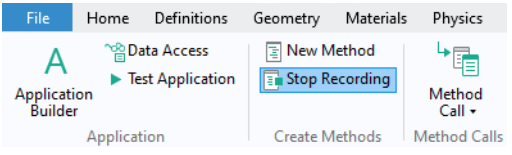
- Click **Write** to generate and save the report to file.



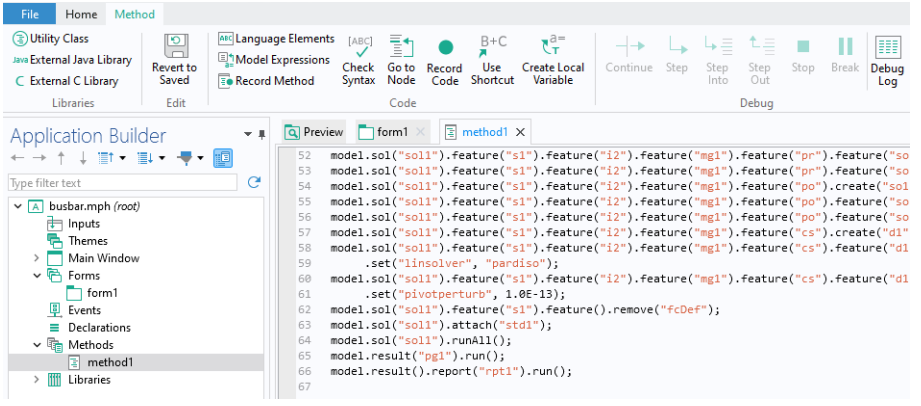
- Close the Microsoft® Word document that was automatically opened.
- Click the **Developer** tab in the ribbon (of the **Model Builder)** and click the **Record Method** button. Click **OK** in the dialog box that opens.



- 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** (if prompted to overwrite, answer **Yes**). Close the Microsoft® Word document that was automatically opened.
- Click the **Developer** tab in the ribbon and click the **Stop Recording** button.

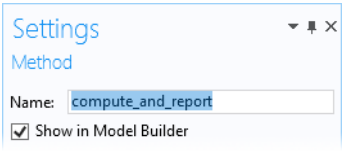


- You can now switch over to the Application Builder, by clicking on the **Application Builder** button in the ribbon, and see the recorded method in the application tree and in the Method Editor.



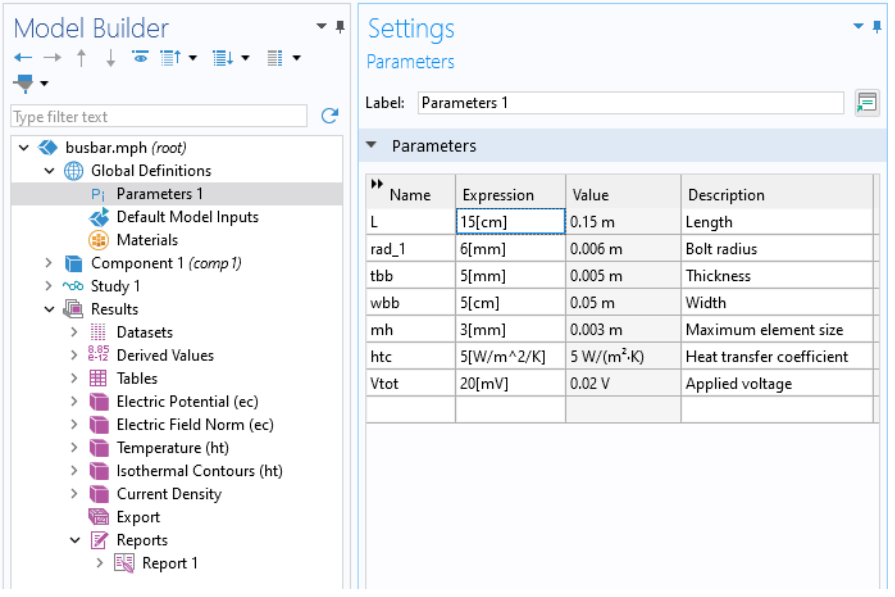
For more information on code generated from the **Study** node, see the *Application Programming Guide*.

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

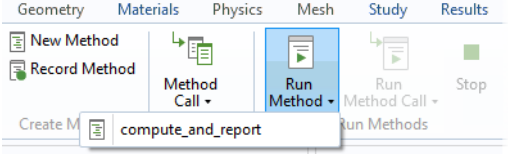


- Go back to the Model Builder by clicking on the **Model Builder** button in the ribbon.

- 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 (if prompted to confirm, answer **Yes**).



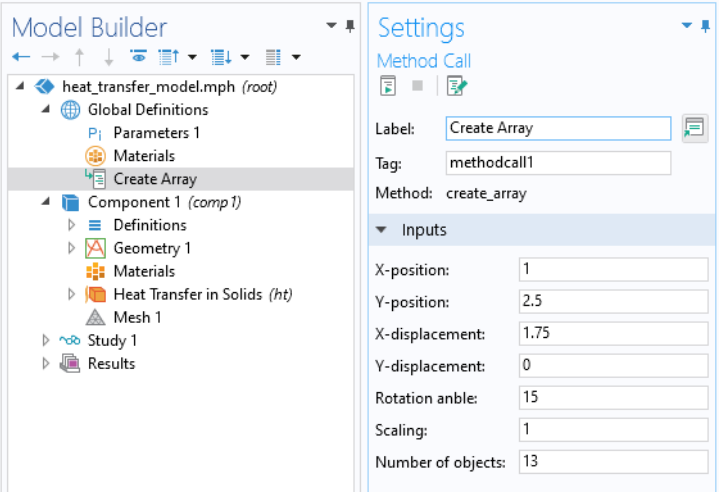
Depending on your security settings, you may get an error message. To avoid this error, open **File>Preferences**, go to the **Security** page, and change **File system access** to **All files**. You can change this back to its default setting after running this example.

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

Method Calls

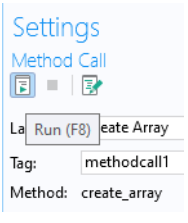
A call, in the Model Builder, to a method for a specific set of input argument values can be made by adding a **Method Call** node under **Global Definitions**. To add a **Method Call** node, right-click **Global Definitions** and select one of the methods that

you have created. The figure below shows a **Method Call** to a method for creating a geometric array.

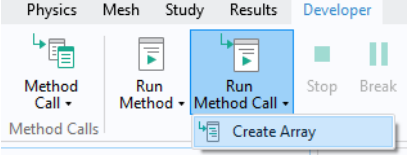


The user interface layout of a **Method Call** cannot be customized. Instead, for customizing use a **Settings Form**, see "Using Forms in the Model Builder" on page 132.

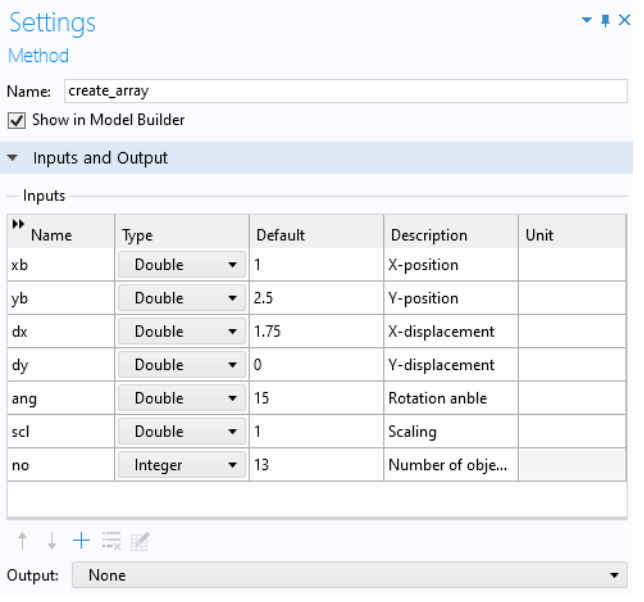
You can run, stop, or edit a **Method Call** by clicking the corresponding toolbar button in the **Settings** window, as shown in the figure below.



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:

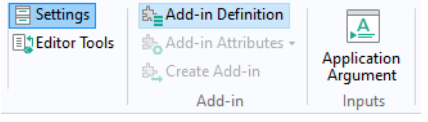
```
message("Width: "+toString(width));
message("Depth: "+toString(depth));
```

For debugging, you use the general techniques outlined in the section "Debugging" on page 212.

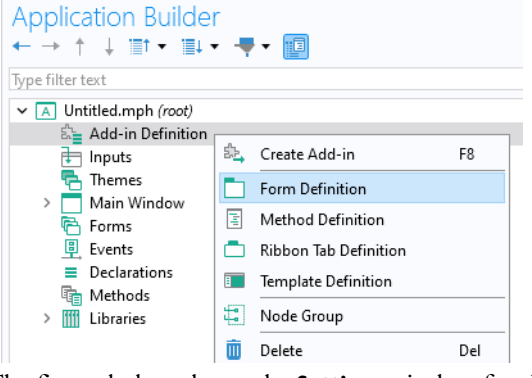
For reusing a **Settings Form** between sessions, you can create an **Add-in**. For more information, see "Creating Add-Ins" on page 226.

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**, **Settings Form**, or a custom ribbon tab. 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 **Home** tab of the ribbon, as shown below.

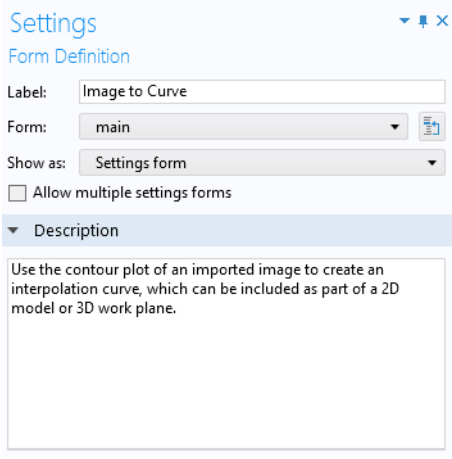


Right-click the **Add-in Definitions** node in the application tree and select **Form Definition.**

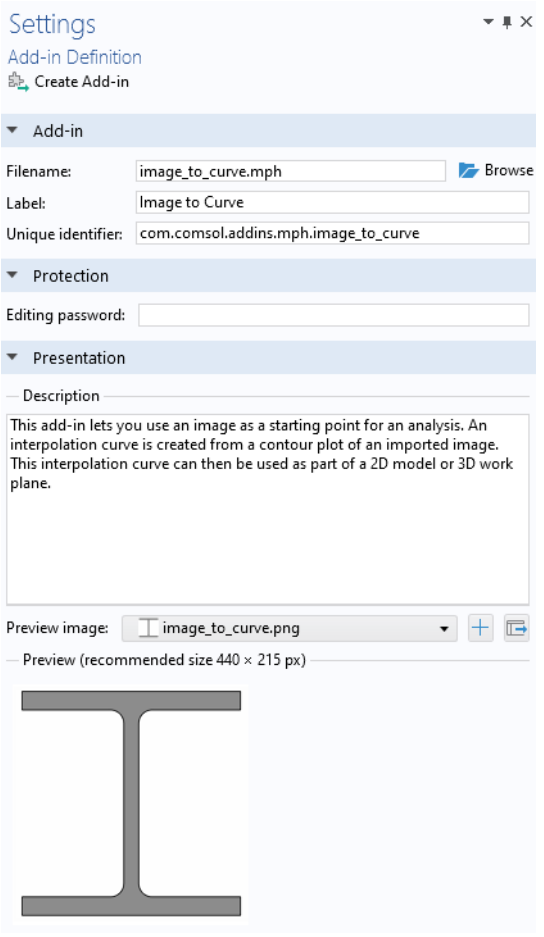


The figure below shows the **Settings** window for the **Form Definition**. Here, you can type a **Label** for the add-in form as well as select which form to use for the add-in. The **Label** will be displayed in the user-defined add-in library. You can select whether the form should be displayed as 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.

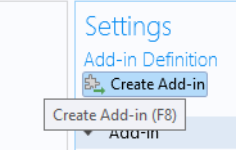


Click the **Add-in Definition** node to see its **Settings** window, as shown below.

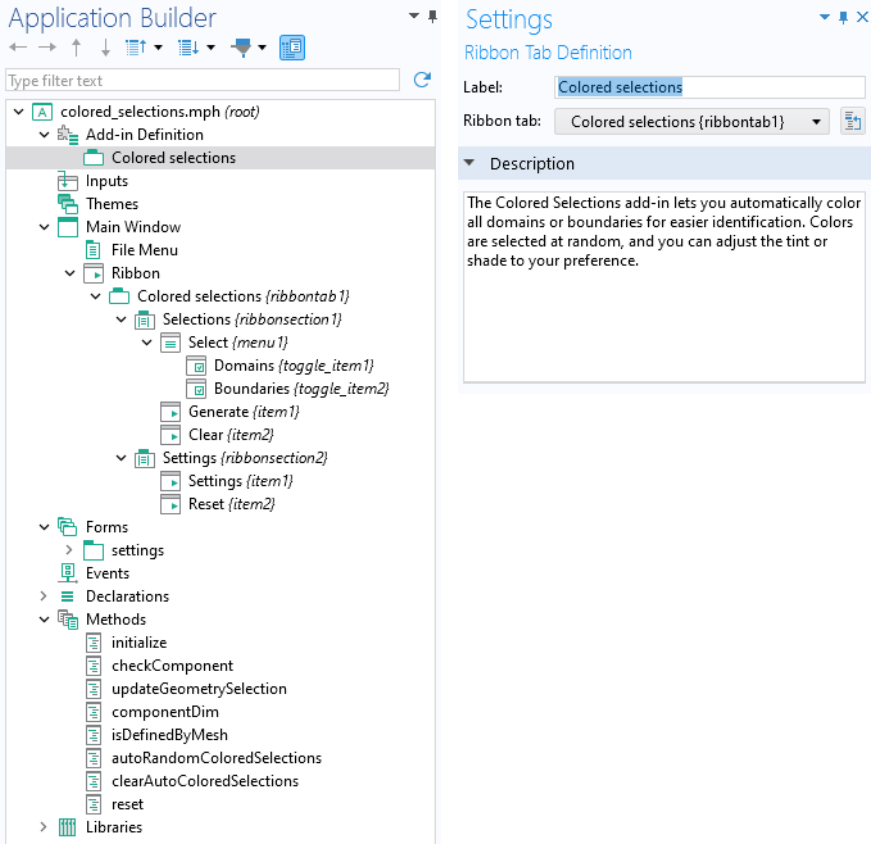


The **Filename** is the location of the add-in MPH-file in the user-defined add-in library. This location can be on a shared network drive if you wish to share the add-in with your colleagues. The **Label** will be displayed in the **Add-in Libraries** window. The **Unique identifier** is what identifies 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.

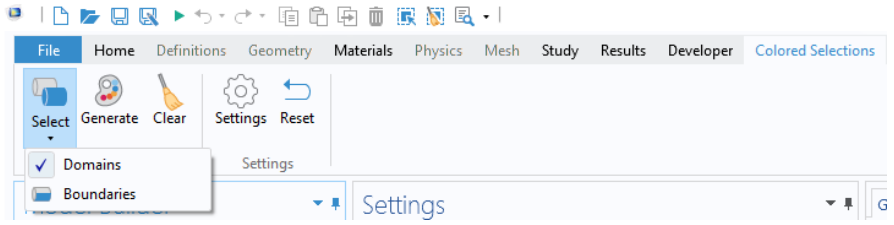
To create the add-in, which is a special type of MPH-file, click the **Create Add-in** button.



As an alternative to a **Form Definition** you can use a **Ribbon Tab Definition** to create one or more ribbon tabs with customized buttons (items) and menus. The figures below show the settings for an add-in that defines a custom ribbon tab with menus and buttons.

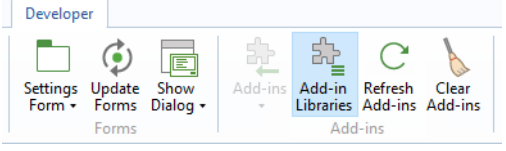


The figure below shows this custom ribbon tab after it has been added from the **Add-in Libraries**.

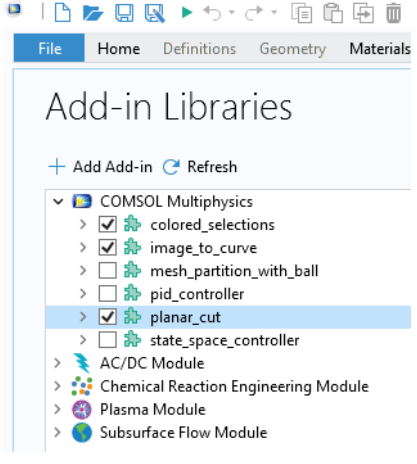


Add-In Libraries

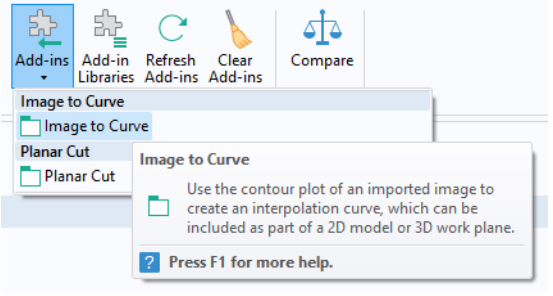
To use an add-in from the **Add-in Libraries**, you first need to enable it. In the **Developer** tab in the Model Builder, click **Add-in Libraries**.



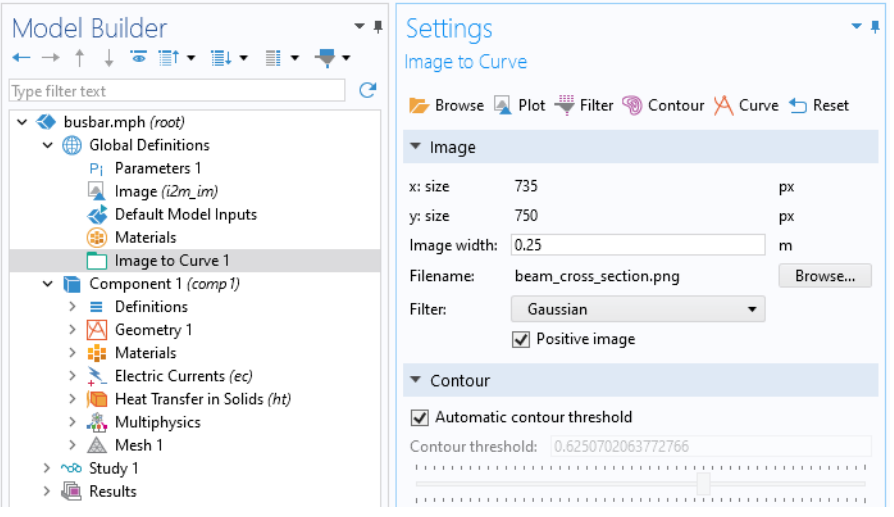
In the list of add-ins, select the check boxes of those add-ins that you want to enable.



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



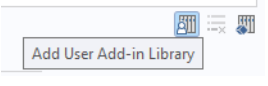
The figure below shows the **Settings Form** for one of the built-in add-ins.



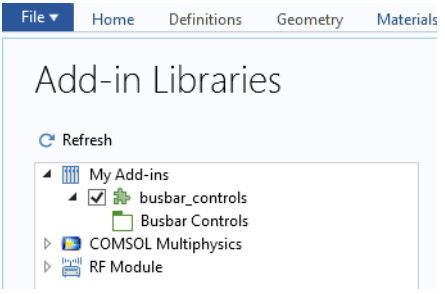
If you want to review and edit the Application Builder settings for a built-in add-in, you can open the corresponding MPH-file. In a typical Windows® installation, the built-in add-in library is located at:

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

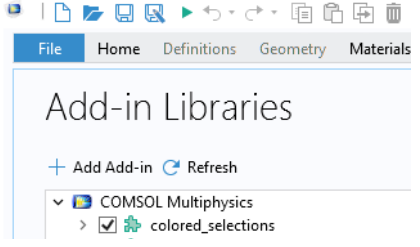
You can browse to a user-defined add-in library by clicking the **Add User Add-in Library** button at the bottom of the **Add-in Libraries** window.



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

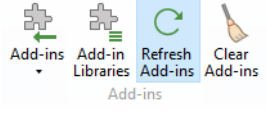


You can also click the **Add Add-in** button at the top of the Add in Libraries window to browse to a user-defined add-in which will then be copied to a standard location in the COMSOL Multiphysics installation folder (typically at `C:\Users\paul\.comsol\v62\addins` or similar).



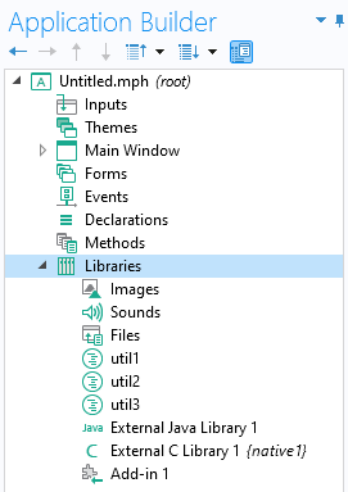
Workflow when Creating and Editing Add-Ins

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, as well as Add-ins. For more information on using utility classes and external libraries, see the *Application Builder Reference Manual*.



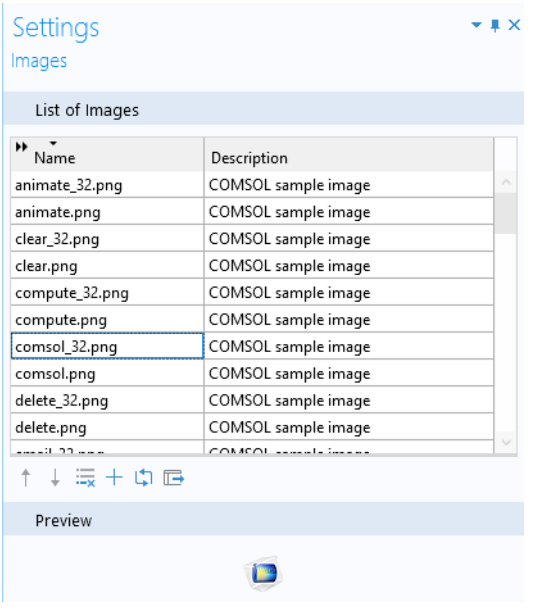
Embedded files can, for example, be referenced in form objects or in methods by using the syntax `embedded:///file1`, `embedded:///file2`, and so on. For example, to reference the image file `compute.png`, use the syntax `embedded:///compute.png`.

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 172, “File Import” on page 291, and “Appendix C — File Handling and File Scheme Syntax” on page 328.

Images

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

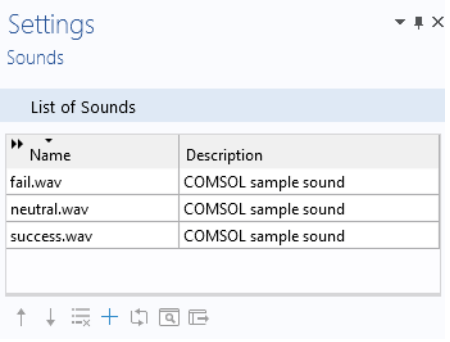
To preview an image, click the name of the image in the **List of Images.** The image is displayed in the **Preview** section

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**

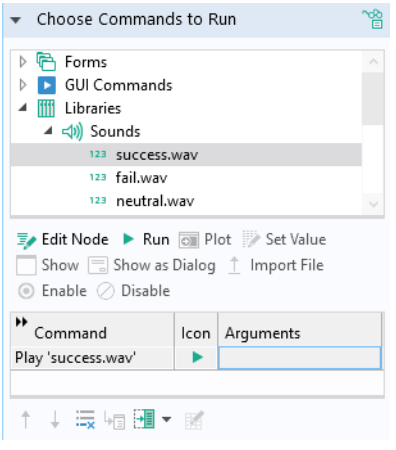
of Sounds. A larger selection of sounds is available in the COMSOL installation folder in the location `data/sounds`.



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.

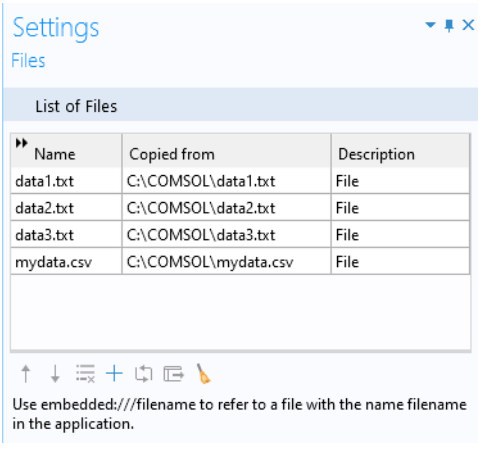


In methods, you can play sounds using the built-in method, `playSound`, such as:

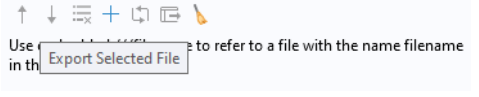
```
playSound("success.wav");
```

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 172, “Appendix C — File Handling and File Scheme Syntax” on page 328, and “File Methods” on page 354.

Appendix A — Form Objects

This appendix provides information about forms and form objects and expands upon the section "The Form Editor" on page 55. 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*
  - Check Box
  - Toggle Button
  - Combo Box
- Labels
  - Text Label*
  - Unit*
  - Equation
  - Line
- Display
  - Data Display*
  - Graphics*
  - Web Page
  - Image
  - Video
  - Progress Bar
  - Gauge
  - Log
  - Message Log
  - Results Table

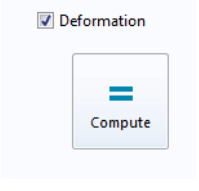
- Subforms
  - Form
  - Form Collection
  - Card Stack
- Composite
  - File Import
  - Information Card Stack
  - Array Input
  - Radio Button
  - Selection Input
- Miscellaneous
  - Text
  - List Box
  - Table
  - Slider
  - Knob
  - Hyperlink
  - Toolbar
  - Spacer

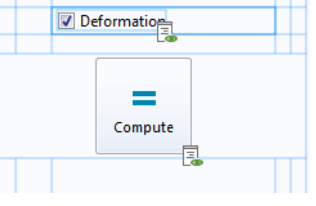
Check Box

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

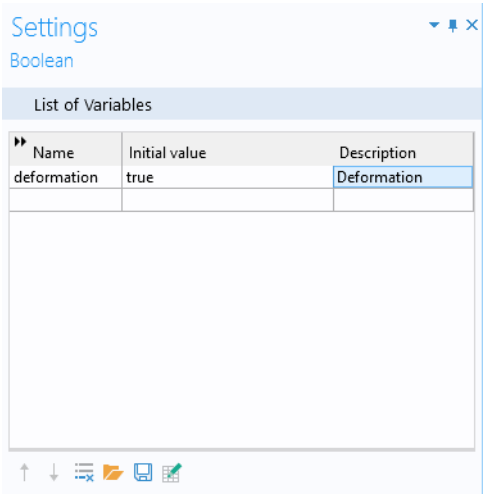
The figure below is from an application where a deformation plot is disabled or enabled, depending on whether the check box is selected.



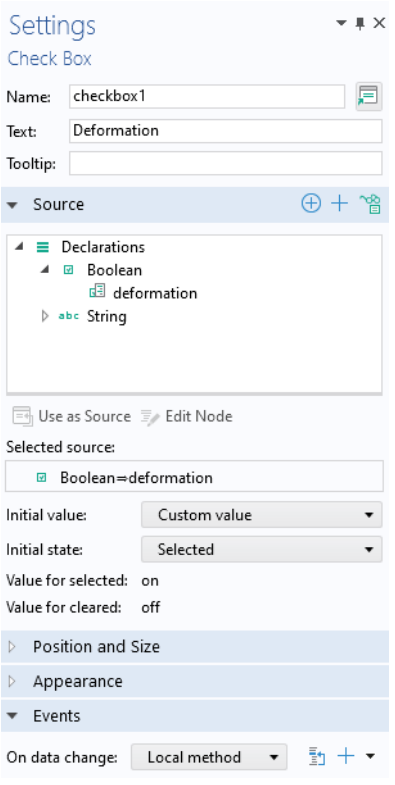


The screenshot on the left shows the running application. The screenshot on the right shows the corresponding form objects in grid layout mode.

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.

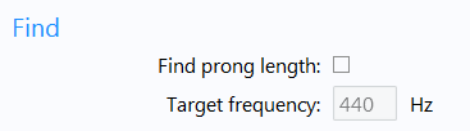
The **Initial value** of the variable `deformation` is overwritten by the **Value for selected** (`on`) or the **Value for cleared** (`off`) and does not need to be edited. When used in methods, the values `on` and `off` are aliases for `true` and `false`, respectively. These values can be used as Booleans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

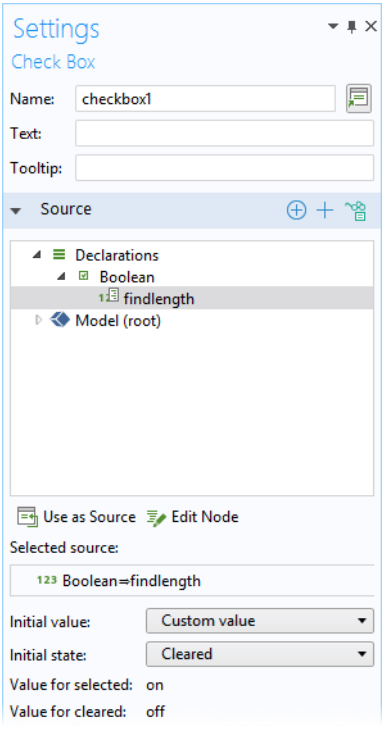
```
model.result("pg1").feature("surf1").feature("def").active(deformation);
useGraphics(model.result("pg1"), "graphics1");
```

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.

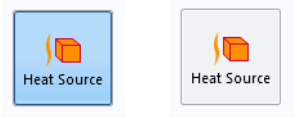


The code statements below come from a local method that is run for an **On data change** event when the value of the Boolean variable `findlength` changes.

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

Toggle Button

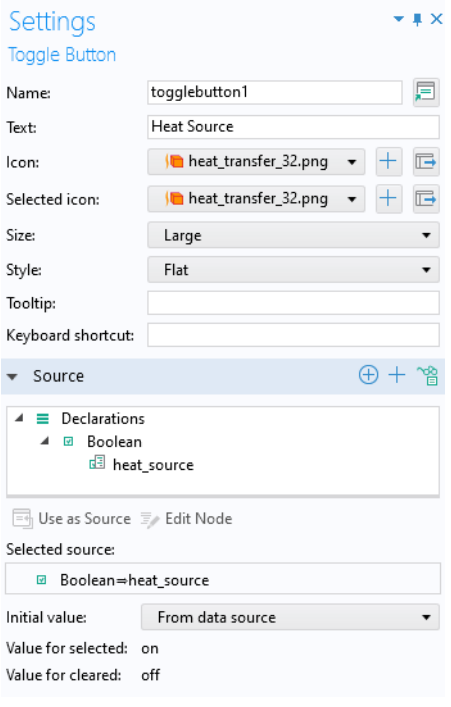
A **Toggle Button** object is a button with two states: selected and deselected, as shown in the figure below.



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

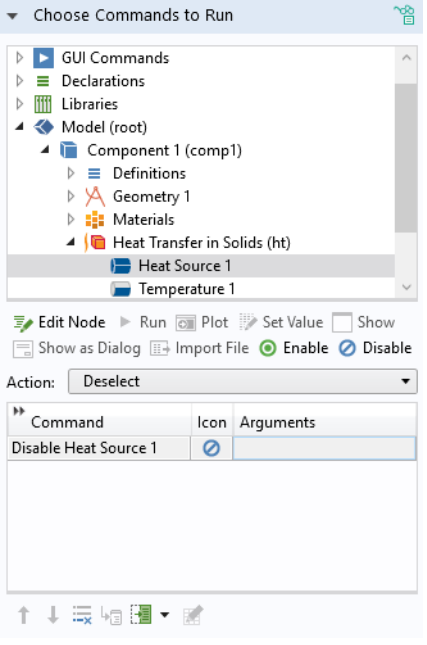
Using a Toggle Button to Enable and Disable a Heat Source

The two states of a toggle button are typically 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.

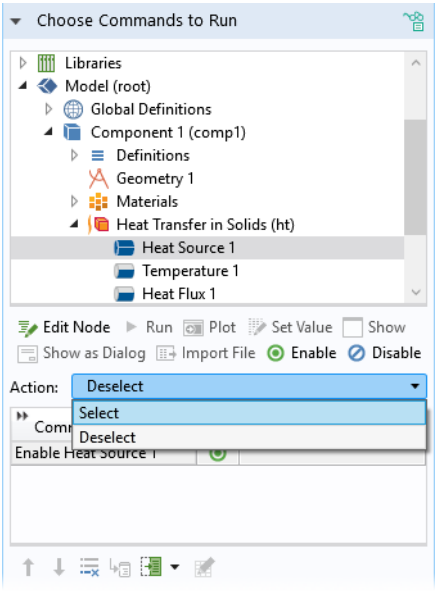


Enabled corresponds to the Boolean variable `heat_source` being equal to `true`, which in turn corresponds to the toggle button being selected. Disabled corresponds to the Boolean variable `heat_source` being equal to `false`, which in turn corresponds to the toggle button being deselected. The **Icon** is displayed when the toggle button is not selected. When the toggle button is selected, the **Selected icon** is displayed.

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 can be 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. For simultaneous control of multiple toggle buttons you can link to a String variable, instead of a Boolean variable. This enables you to group the behavior of multiple toggle buttons similar to that of a radio button.

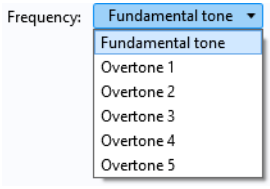
Combo Box

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.

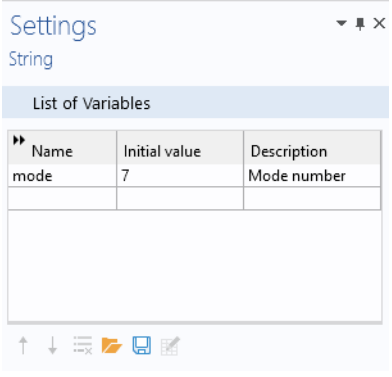
Using a Combo Box to Change Parameters in Results

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:

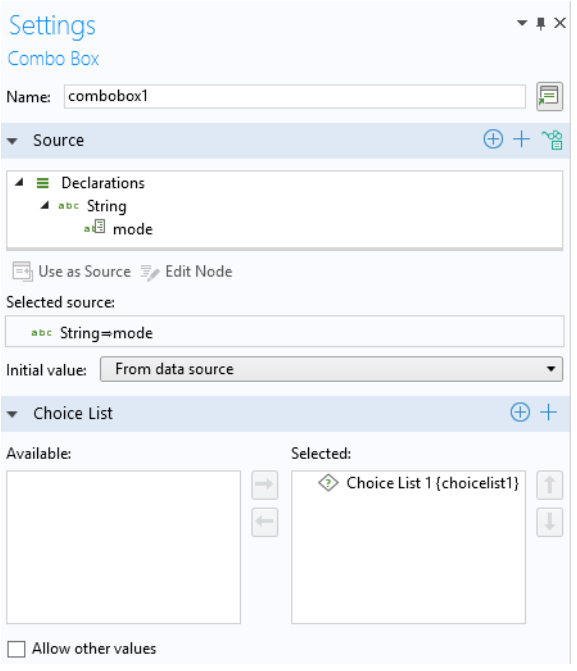


In this example, the combo box is used to control the value of a string variable `mode`. The figure below shows the **Settings** window for this variable.



Selecting the Source

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

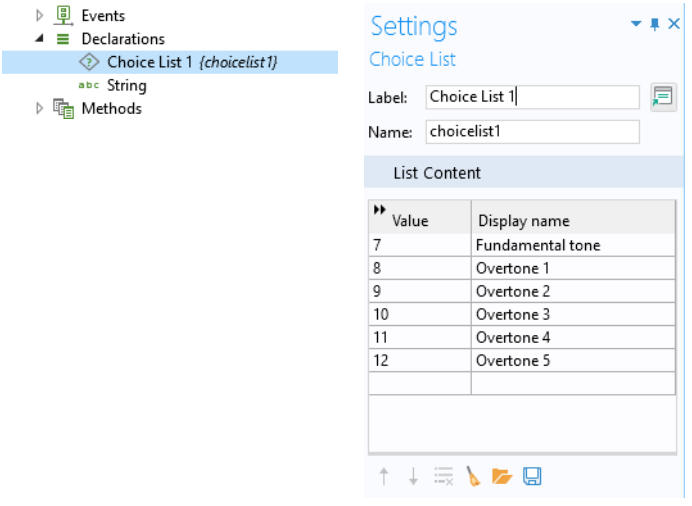


In the **Source** section, you select a scalar variable that should have its value controlled by the combo box and click **Use as Source**. In the **Initial values** list of the **Settings** window of the combo box, choose a method to define a default value for the combo box. The options are **First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



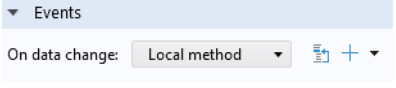
The string variable `mode` is allowed to have one of these six values: `7`, `8`, `9`, `10`, `11`, or `12`. The text strings in the **Display name** column are shown in the combo box.

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 170.

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.



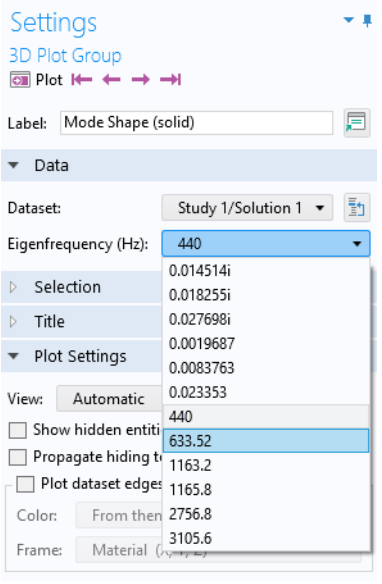
The code for the local method is listed below.

```
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 633.52 Hz in the **Settings** window for the **Mode Shape** plot group.



- Click **Stop Recording**.

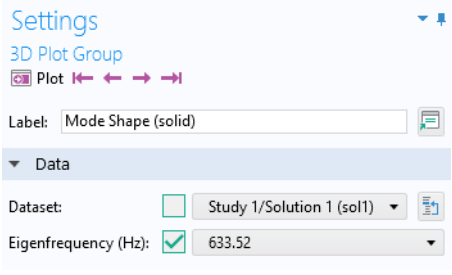
The resulting code is shown below.

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

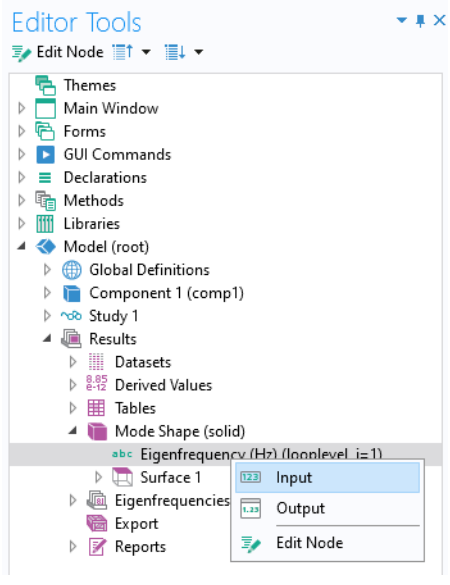
Now change the string `"8"` with the variable `mode` to end up with the code listing above. This will be stored in a method, say, `method1`. To create the local method associated with the combo box, copy the code from `method1`. Then, delete `method1`.

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.

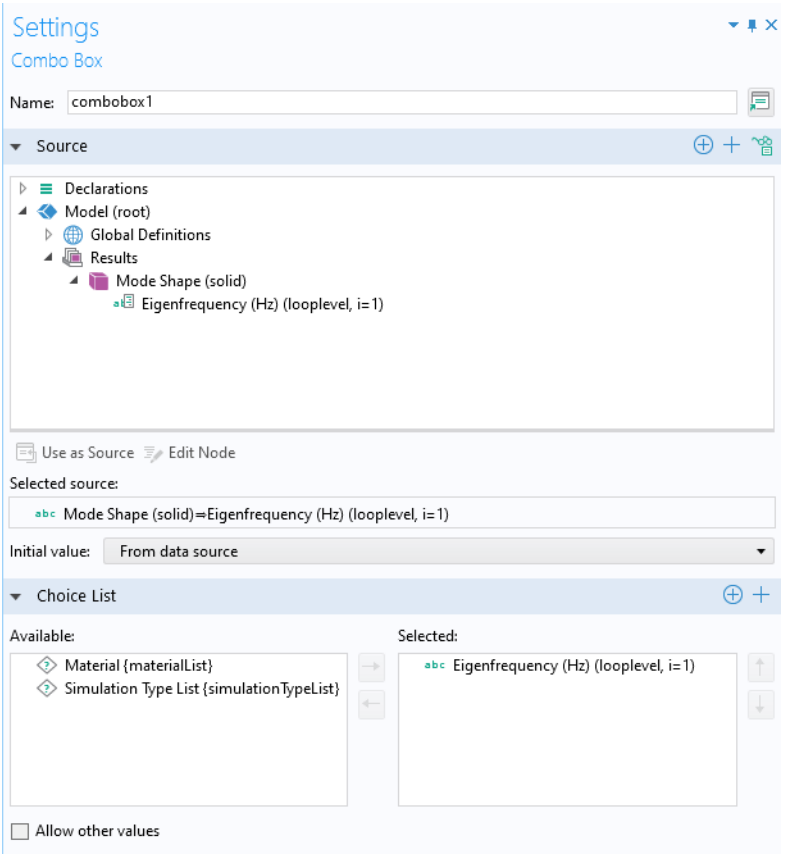


In the **Editor Tools** window, the **Eigenfrequency** parameter is visible as **Loop Level**. To create a combo box, right-click **Loop Level** and select **Input**.



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.

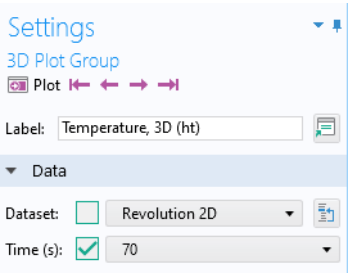


The choice list **Loop Level** is automatically generated when inserting a combo box using **Editor Tools.** Note that a choice list generated in this way is not displayed under the **Declarations** node and cannot be modified by the user. For greater flexibility, such as giving names to each parameter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

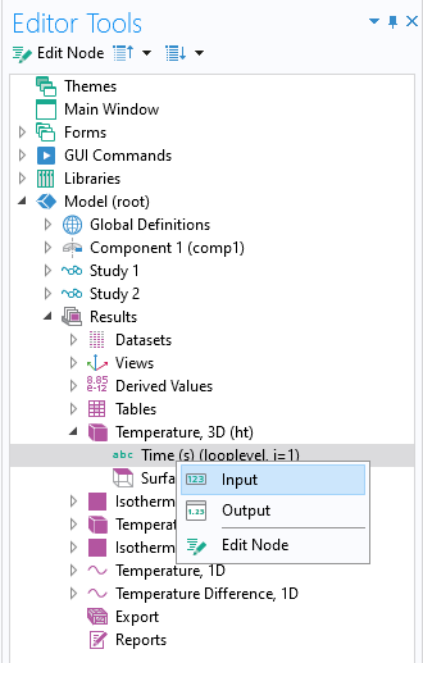
Using a Combo Box to Change Times

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**.

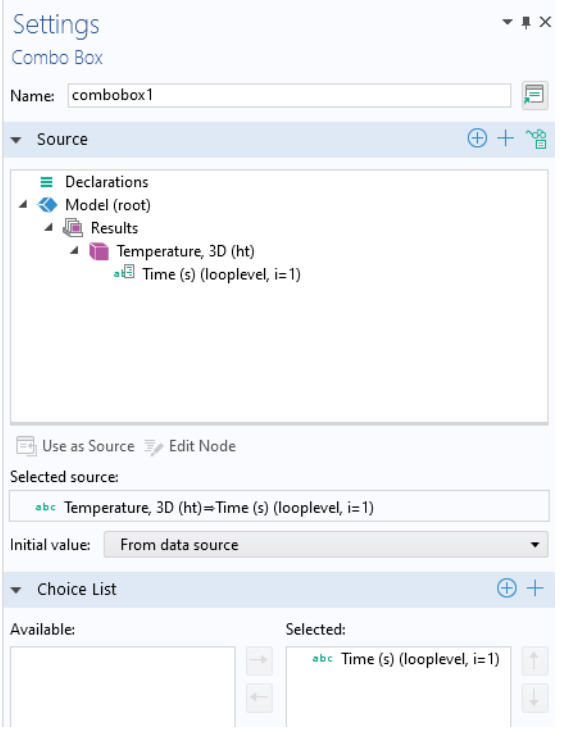
In the **Settings** window in the figure below, **Data Access** has been used to access the **Time** parameter list in a temperature plot.



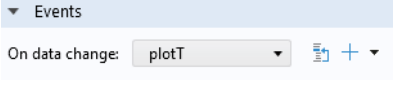
In **Editor Tools**, a handle to the **Time** list is now available, as shown in the figure below.



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



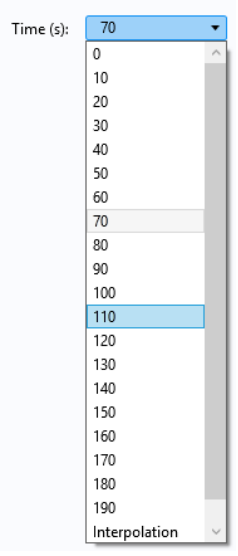
The combo box can be used for multiple purposes, for example, to update a plot corresponding to a different time parameter. In order for a plot to automatically update when a user uses the combo box to select a new time parameter, add an event to the combo box at the bottom of its **Settings** window.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



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

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

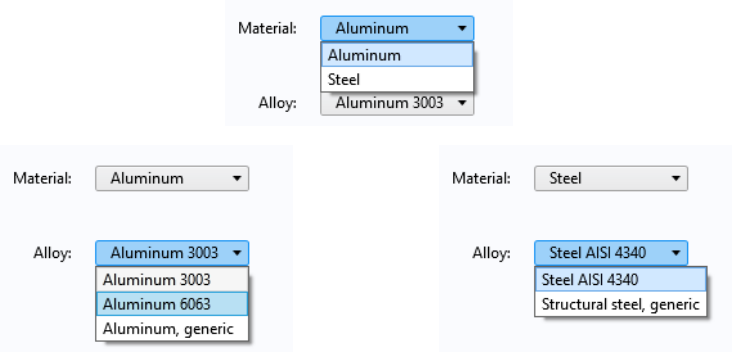
The end result is a combo box in the application user interface, shown in the figure below, which automatically updates a temperature plot when the user selects a new value for the **Time** list.



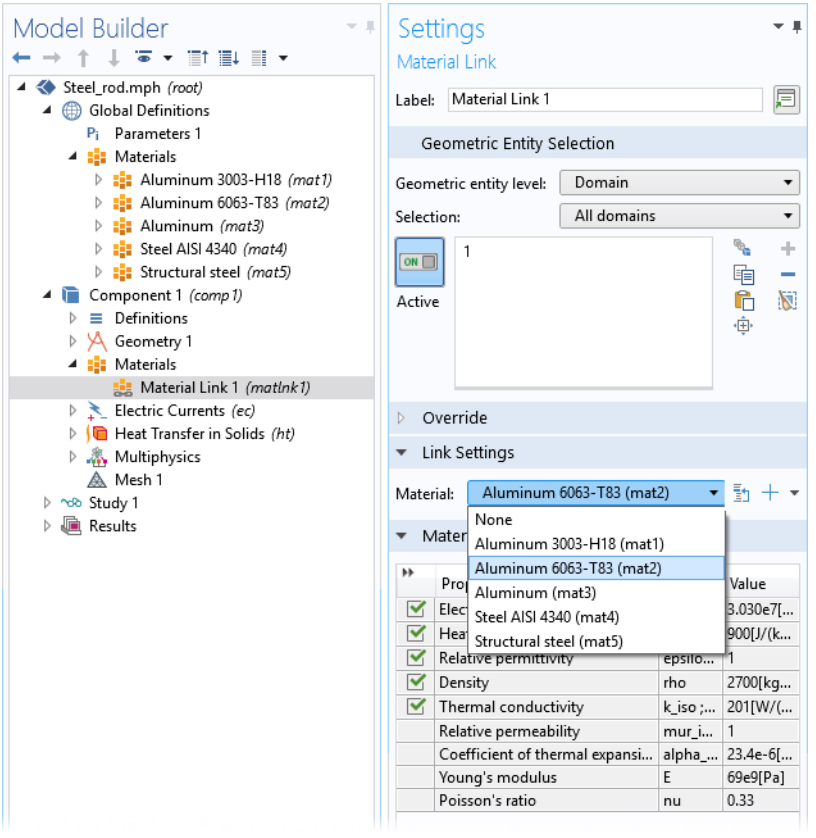
Using a Combo Box to Change Material

Consider an application where combo boxes are used to select the material. In this case, an activation condition (see "Activation Condition" on page 171) can also be used for greater flexibility in the user interface design.

The figure below shows screenshots from an application where the user can choose between two materials, **Aluminum** or **Steel**, using a combo box named **Material**. A second combo box called **Alloy** shows a list of **Aluminum** alloys or **Steel** alloys, according to the choice made in the **Material** list.



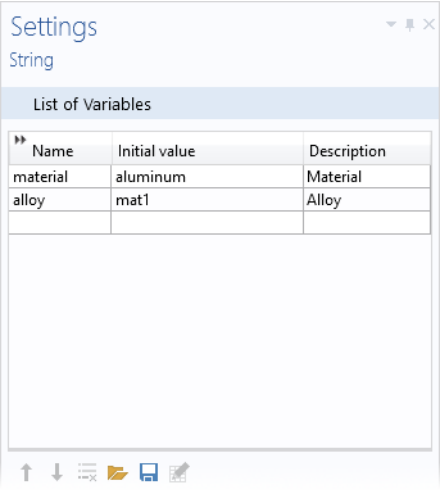
The material choice is implemented in the embedded model using global materials and a material link, as shown below.



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

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

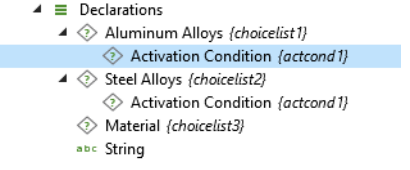
The figure below shows the declaration of two string variables, `material` and `alloy`, which are controlled by the **Material** and **Alloy** combo boxes, respectively.



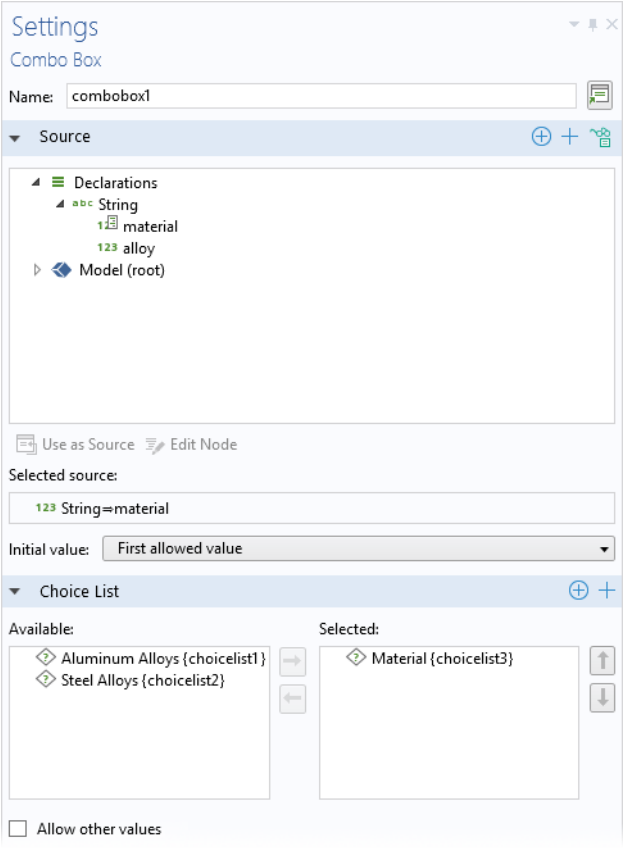
The application utilizes three choice lists: **Aluminum Alloys, Steel Alloys**, and **Material**.

Activation Condition

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

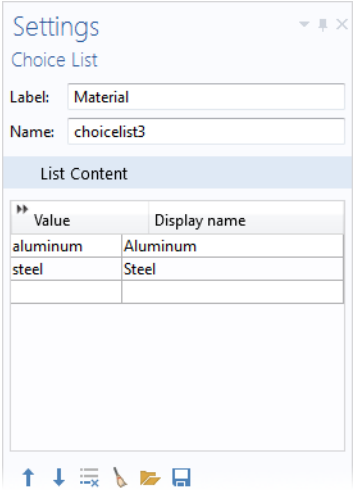


The **Settings** window for the **Material** combo box is shown below.

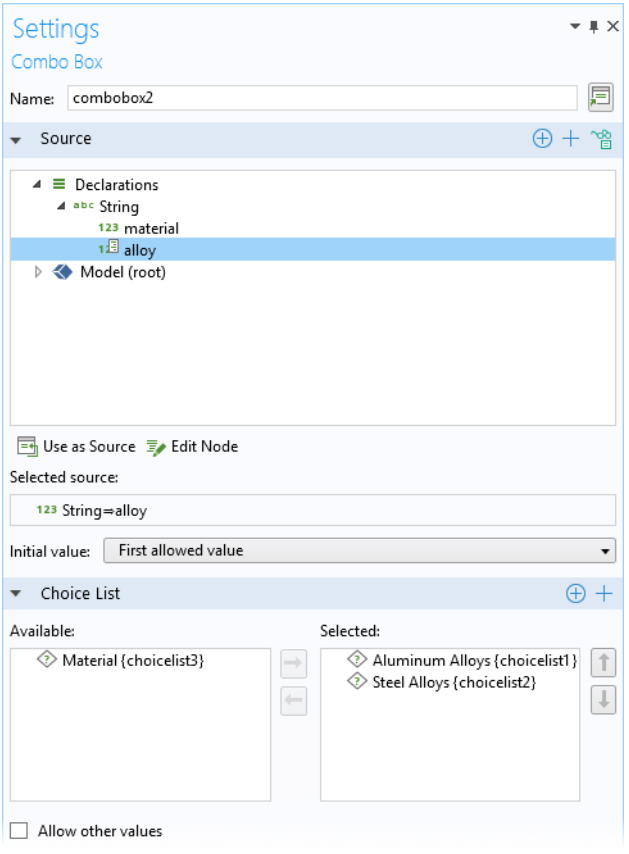


Note that the **Material** combo box uses the `material` string variable as its source. The **Material** choice list is used to define a discrete set of allowed values for the

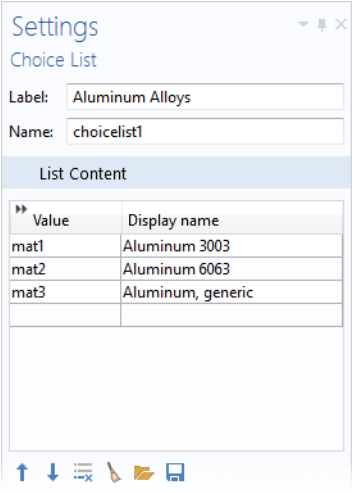
`material` string variable. The **Settings** window for the **Material** choice list is shown below.



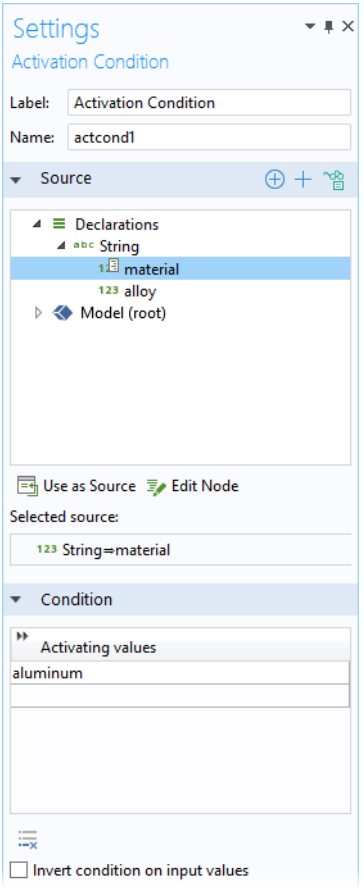
The **Settings** window for the **Alloy** combo box is shown in the figure below.



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



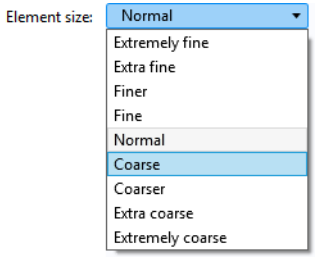
The activation condition for the **Aluminum Alloys** choice list is shown in the figure below.



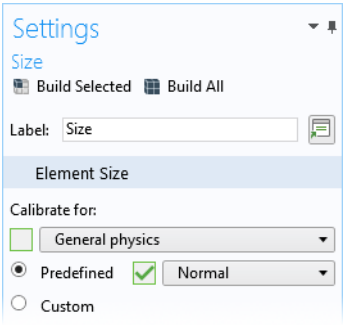
Using a Combo Box to Change Element Size

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



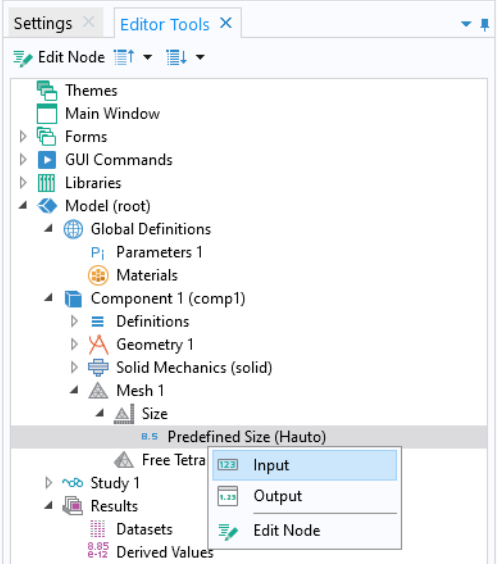
Switch to the Model Builder and select the **Mesh** node (we assume here that the model has just a single mesh). In the **Settings** window of the **Mesh** node, select **User-controlled mesh** (if not already selected). In the **Size** node, directly under the **Mesh** node, select the option **Predefined**.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



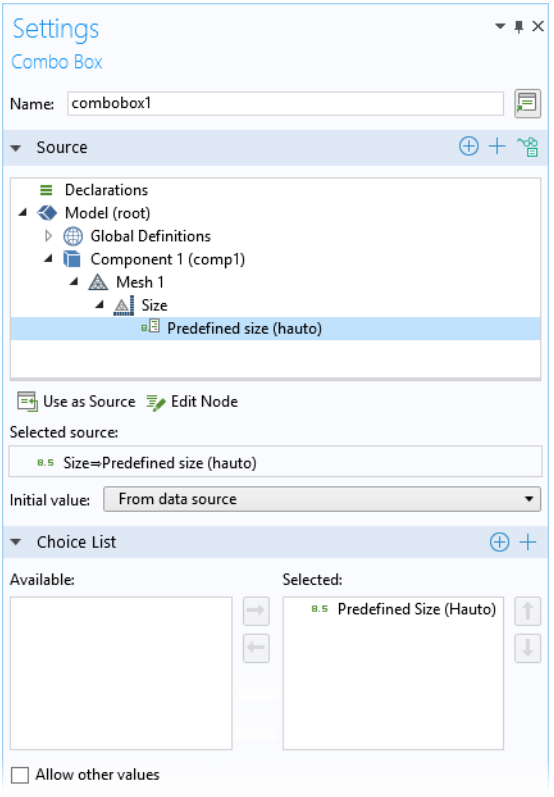
Select the green check box to the left of the list to make it available as a source for a combo box in the Application Builder. Then, when you return to the Application Builder, you will find that the choice list for mesh size is now revealed as a potential **Source** in the **Settings** for a new combo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

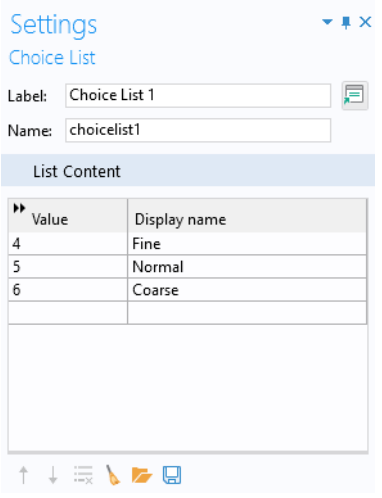


The corresponding **Settings** window for the combo box is shown in the figure below.



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 (hauto)**, from the Model Builder is now selected as the choice list for your combo box in the Application Builder. This choice list does not appear as a choice list under the **Declarations** node of the application tree because it is being referenced from the Model Builder. Therefore, if you want a list with a more limited set of choices, you cannot edit it. Instead, you have to remove the predefined list as the **Source** of your combo box and create a new choice list of your own by declaring it under the **Declarations** node. For

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



To learn which values are used by the **Element size** list in the model, use **Record a New Method** and change the value from **Normal** to **Fine**, then to **Coarse**, and then back to **Normal**. Click **Stop Recording** and read the values in the 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();
```

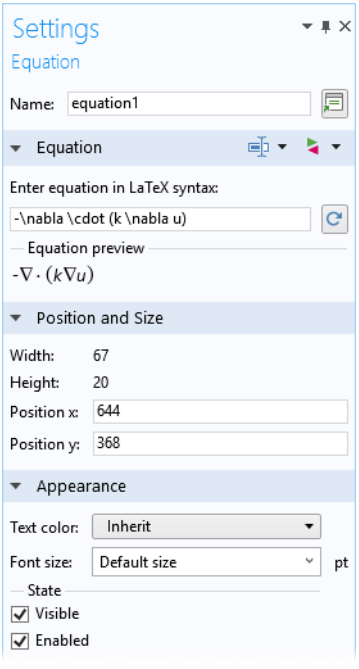
The `hauto` property can also take non-integer values. For more information on **Element size**, see “Data Access for Input Fields” on page 108.

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

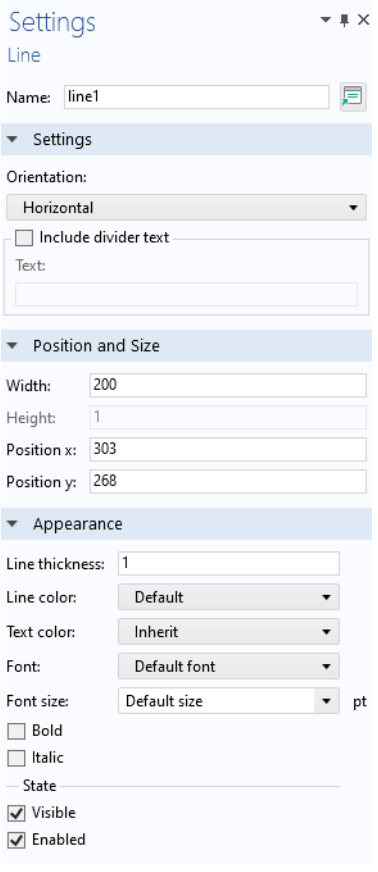
An **Equation** object can display a LaTeX equation by entering the expression in the **Enter equation in LaTeX syntax** field.



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

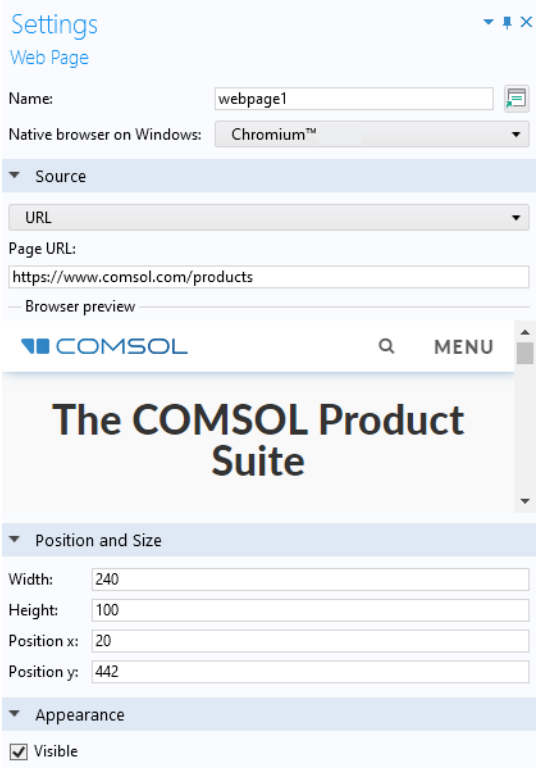
Line

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.



Web Page

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



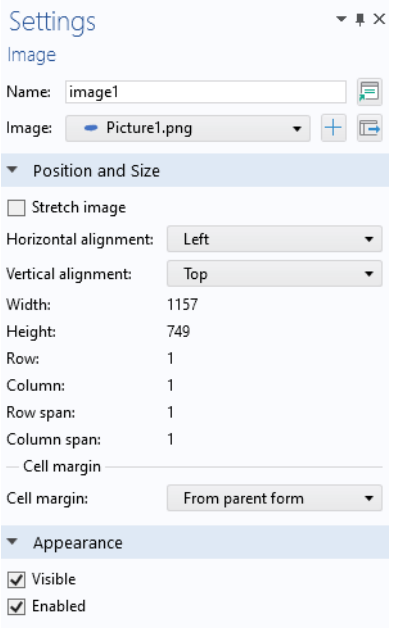
You can specify the page source in four different ways from the **Source** list:

- 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.

Use the setting **Native browser on Windows** to control which browser is used in the Windows® version of COMSOL Multiphysics.

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 `comsol_32.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.

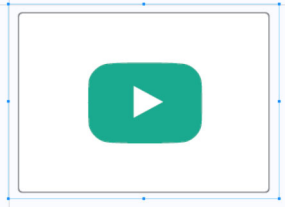
While you can change the *x*- and *y*-position of the image, the width and height settings are determined by the image file. However, if you use the grid layout mode, then there is an additional setting **Stretch image** which, if activated, will automatically scale the image to fill the available grid cell space.

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.

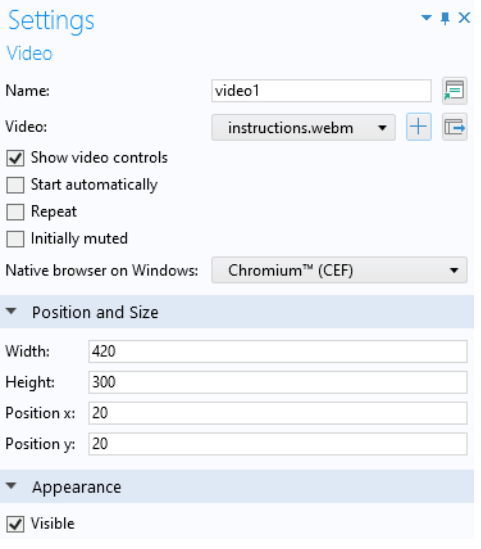
Video

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**

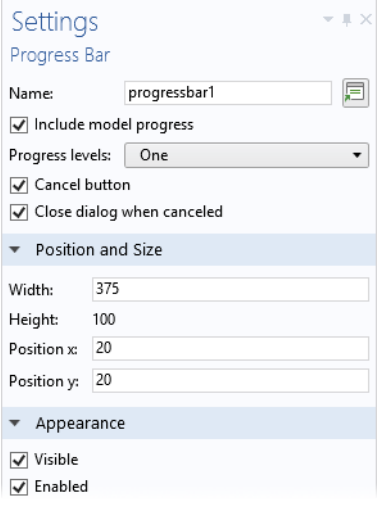
The option **Show video controls** enables the video controls such as Play and Stop.

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.

Use the setting **Native browser on Windows** to control which browser is used in the Windows® version of COMSOL Multiphysics.

Progress Bar

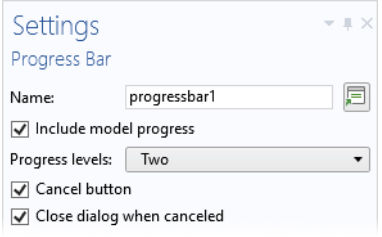
A **Progress Bar** object displays a customized progress bar, or set of progress bars, based on a value that is updated by a method. Use a progress bar to provide feedback on the remaining run time for an application. The figure below shows the **Settings** window of a progress bar object with one progress 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

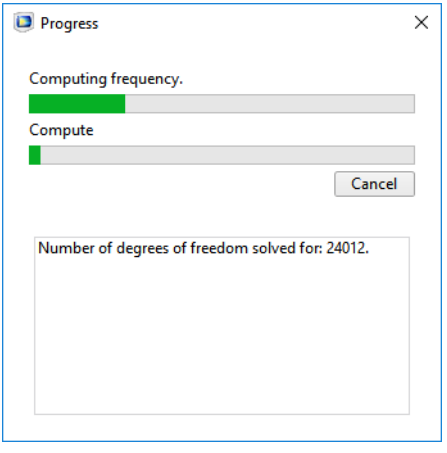
`setProgress` method, you can customize the information shown in the built-in progress bar. For more information, see “Progress Methods” on page 364 and the *Application Programming Guide*.

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

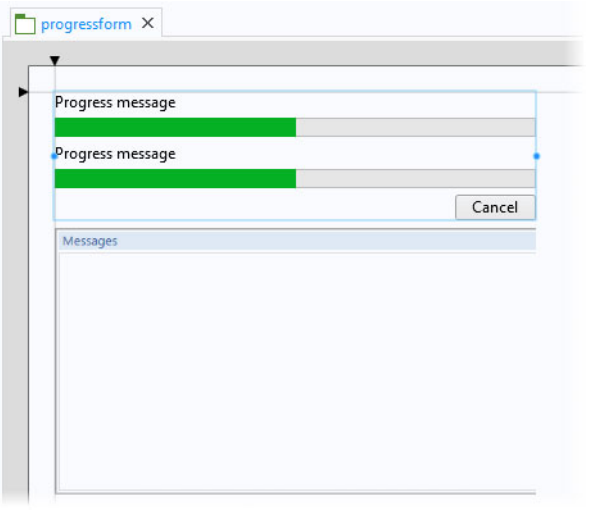


In this example, the progress bar object is part of a form `progressform` used to present a two-level progress bar and a message log.

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");
```

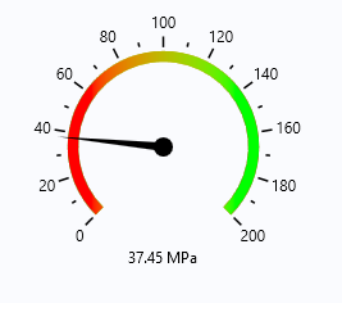
In the example above, the central functionality for updating the two levels of progress bars lies in the call

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

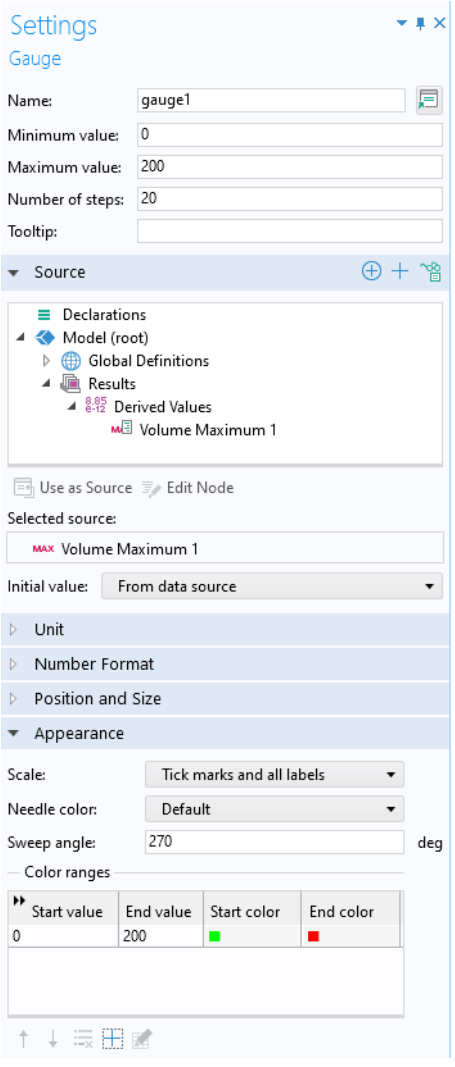
For detailed information on the built-in methods and their syntax, see “Progress Methods” on page 364 and the *Application Programming Guide*.

Gauge

The **Gauge** form object is similar to the knob form object but is read-only and used to display a value by the position of a needle or arrow of a gauge. Optional tick marks and labels are used to indicate the range of values that the gauge can display. In addition to tick marks and labels, you can optionally configure a color scale to indicate the values. The figure below shows a gauge object together with a data display object, displaying the maximum stress in a structural mechanics application.



The **Settings** window for a gauge object is similar to that of a knob object. However, there are some differences. For example, there is no **Events** section for a gauge object. The figure below shows the **Settings** window for a gauge object.



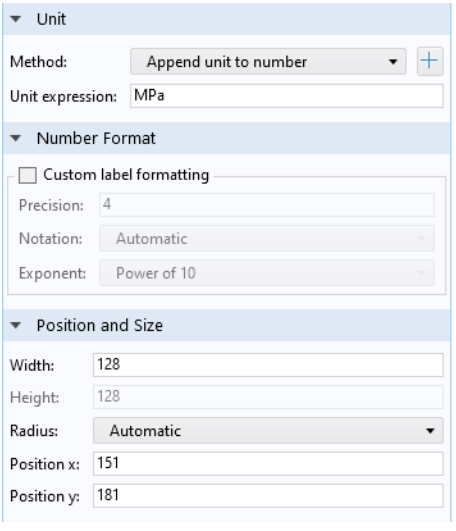
You can select any parameter, variable, or declared scalar variable as a source. Select from the application tree and click **Use as Source**.

You determine the range of values for the data source by defining the **Minimum value**, **Maximum value**, and **Number of steps** for the gauge.

You can enter a **Tooltip** that is shown when hovering over the gauge. The settings for units are similar to that of a slider or knob.

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

The figure below shows the **Unit**, **Number Format**, and **Position and Size** sections.

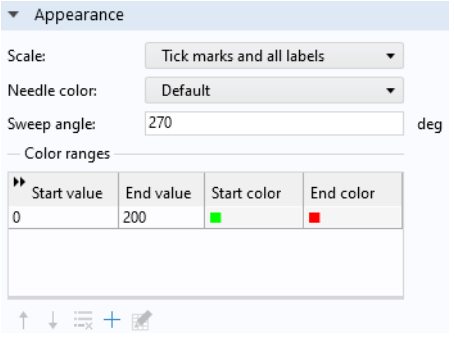


The **Unit** section is similar to that of a slide or knob object.

In the **Number Format** section, you can specify a custom format for the tick labels.

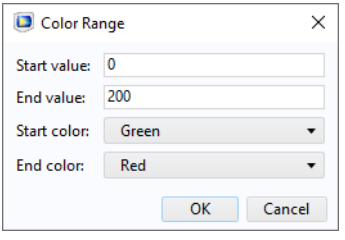
The **Position and Size** section will have different contents depending on if the form is using sketch mode or grid mode. You specify the **Width** and the **Height** is automatically set to the same value as the **Width** (the **Height** edit field is disabled). The **Radius** value specifies the radius for the tip of the gauge object's needle. By default this is calculated automatically, but you can set it manually to make sure that gauge objects placed next to each other have the same needle radius.

The figure below shows the **Appearance** section.



The **Needle color** takes its **Default** value from the current color theme.

You can specify a sequence of color ranges for the gauge. Click the **Add** button in the table toolbar to add another color range, which opens a dialog box.



In this dialog box you give the start and end values for a color range as well as the start and end color. To edit an existing color range, select its row in the table and click the **Edit** button. The **Move up** and **Move down** buttons can be used to reorder the color ranges and the **Delete** button deletes the color ranges in the selected rows.

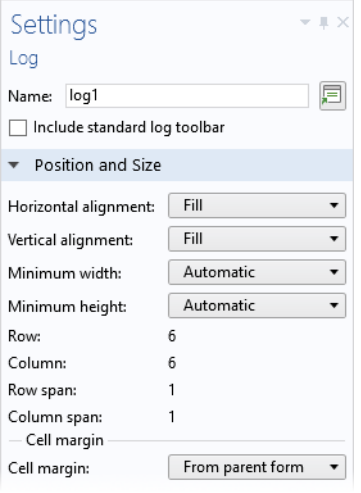
The **Start color** and **End color** can take the value **Transparent** which means that they get the same color as the current background color.

If the **End value** is less than the **Start value**, then the color range is rendered backwards.

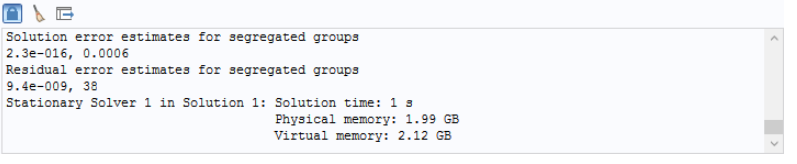
Log

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.



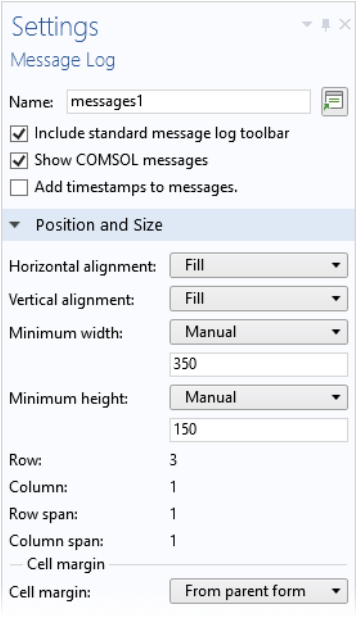
The figure below shows a part of an application user interface containing a log window.



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

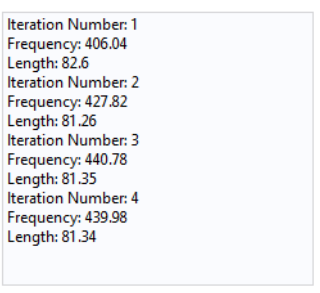
using the built-in `message` method with syntax: `message(String message)`. See also "GUI-Related Methods" on page 359.

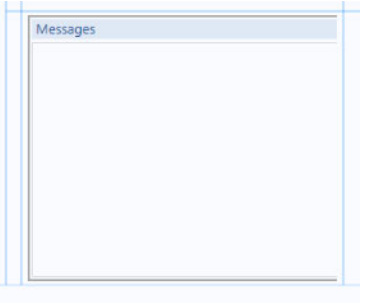


You can also display the value of a variable, for example: `message(double xcoordinate)`.

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**.

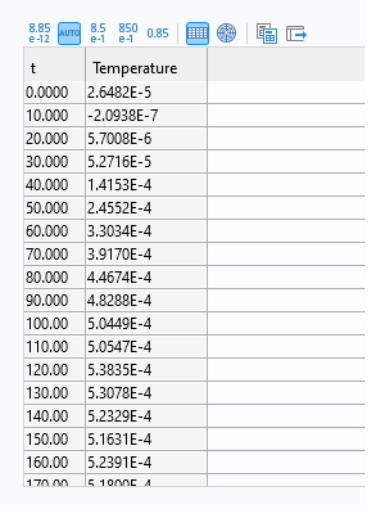
The figure below shows a customized message window with convergence information from a method (left) and the corresponding **Message Log** form object (right).





Results Table

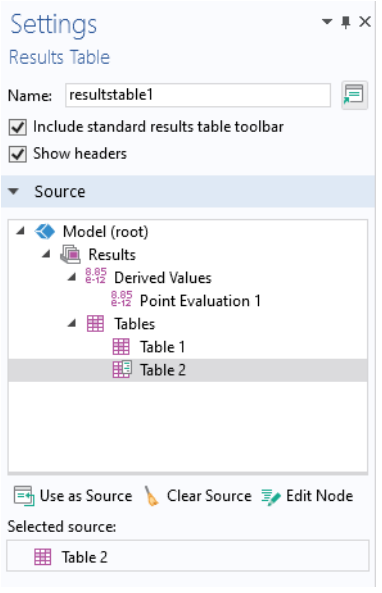
The **Results Table** object is used to display numerical results in a table.



t	Temperature
0.0000	2.6482E-5
10.000	-2.0938E-7
20.000	5.7008E-6
30.000	5.2716E-5
40.000	1.4153E-4
50.000	2.4552E-4
60.000	3.3034E-4
70.000	3.9170E-4
80.000	4.4674E-4
90.000	4.8288E-4
100.00	5.0449E-4
110.00	5.0547E-4
120.00	5.3835E-4
130.00	5.3078E-4
140.00	5.2329E-4
150.00	5.1631E-4
160.00	5.2391E-4
170.00	5.1800E-4

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**.)



By clearing the check box **Show headers**, you can choose to hide the column headers of the results table.

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:

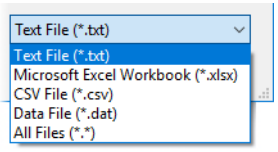
- **Full Precision**
- **Automatic Notation**
- **Scientific Notation**
- **Engineering Notation**
- **Decimal Notation**
- **Rectangular Complex Numbers**
- **Polar Complex Numbers**
- **Copy Table and Headers to Clipboard**
- **Export**

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.



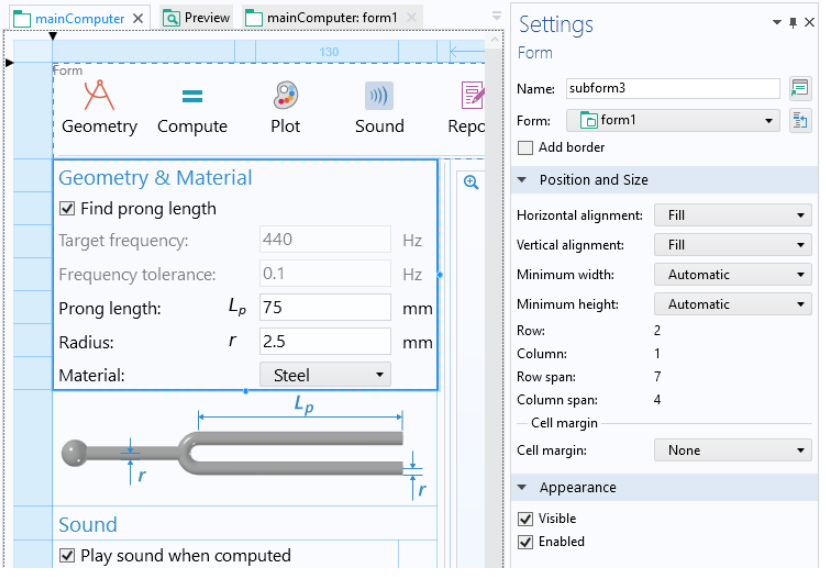
Controlling Results Tables from Methods

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 359.

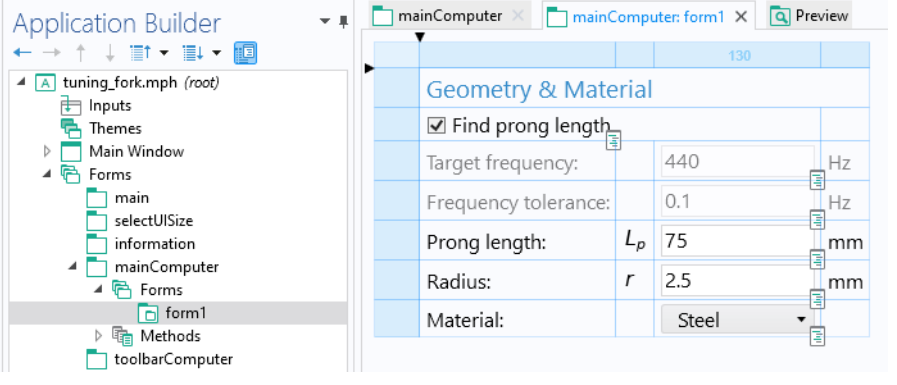
Form

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 a form `mainComputer` has a link to the subform `form1`.



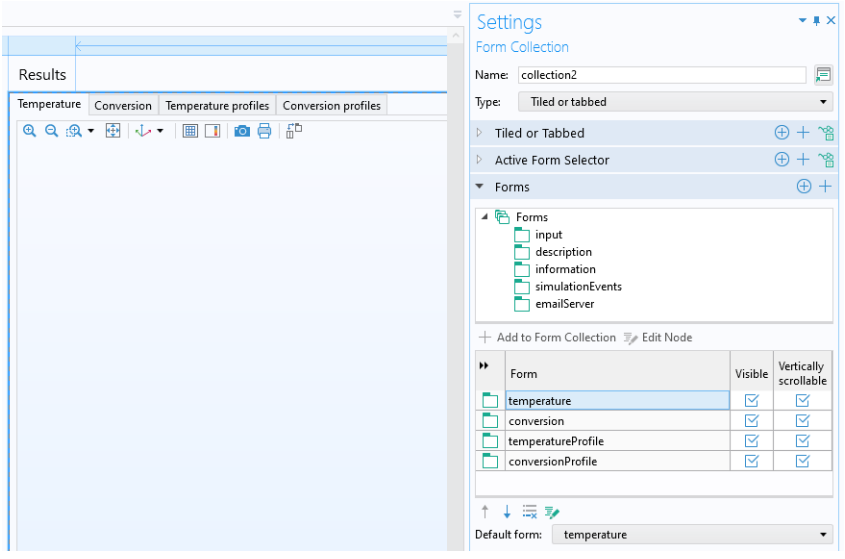
The figure below shows the referenced form `form1`, which is a local form to the form `mainComputer`.



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 127.

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.



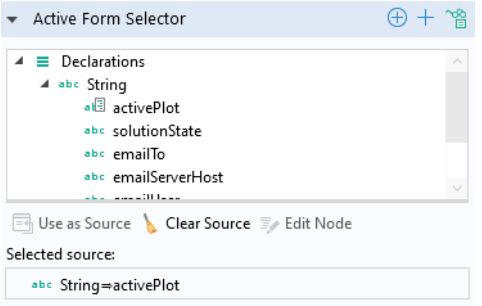
There are four different layout options. From the **Type** list, choose between:

- **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 **Forms** section you can select which forms to display in the **Form Collection**. These will be displayed in the app in the order they appear in the list. You can change the order by clicking the **Move Up** and **Move Down** buttons.

The **Default form** list specifies which form to be shown by default. This setting is not available for the option **Sections**.

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



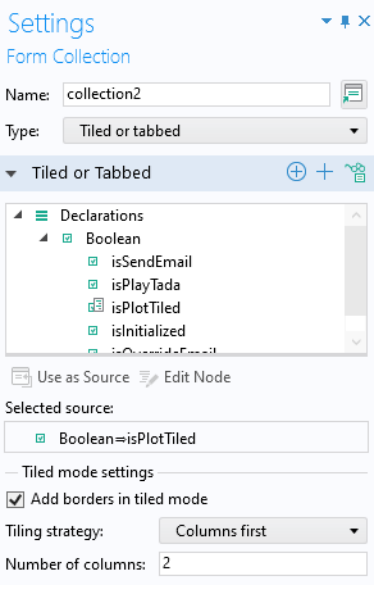
The string variable needs to be equal to one of the form names in the form collection, such as `temperature` or `conversion` in the example above. Otherwise, it will be ignored.

If you change the value of the form selector `activePlot` in the above example, in a method that will be run at some point (a button method, for example), then the form with the new value will be activated, as shown in the example below.

```
activePlot="conversion"; /* Activate the conversion form on completion of
this method */
```

For a form collection with the **Type** set to **Sections**, the **Active Form Selector** has no effect. Using an **Active Form 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 Form Selector**, click the **Clear source** toolbar button under the tree.

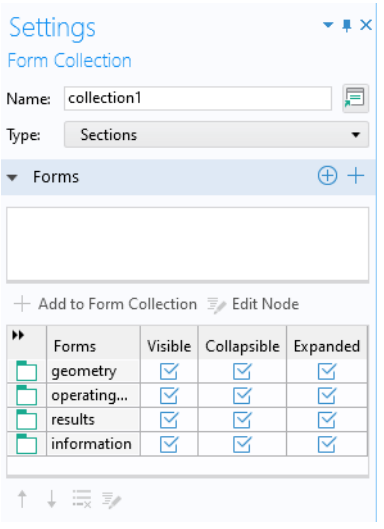
The **Tiled or tabbed** option displays the forms in one of two ways depending on the value of a Boolean variable used as source in a **Tiled or Tabbed** section at the top of the **Settings** window.



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**.

In the table below the **Forms** section you can select which forms should be **Visible**, by selecting the corresponding check box. For a form collection of the

type **Sections**, you can, in addition, control which forms should be **Collapsible** and **Expanded**.



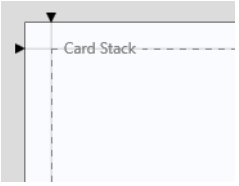
Card Stack

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

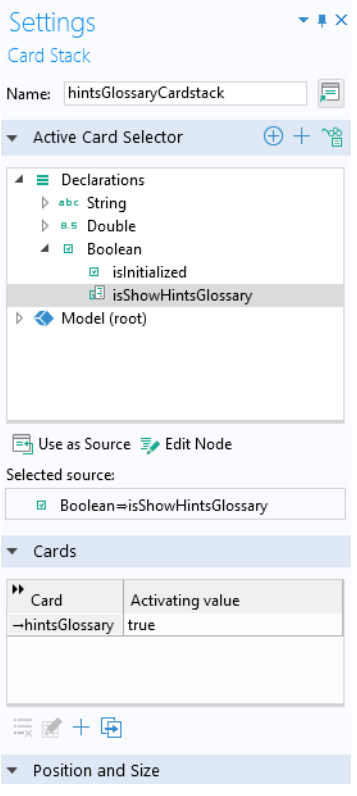
Consider an application where the graphics shown to the user depend on the value of a scalar variable. This variable may change when a user clicks, for example, a radio button. The variable may also change depending on a computed value; for example, the value of a **Global Evaluation** node in the model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



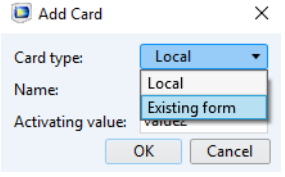
By clicking a row in the table of cards in the **Cards** section, followed by clicking one of the toolbar buttons below the table, you can perform the following operations on cards:

- **Delete**
- **Edit**

- **Add Card**
- **Duplicate**

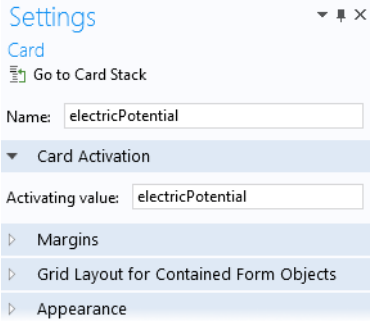
Each row in the table contains the name of the card in the **Card** column and their associated activating values in the **Activating value** column. The stack decides which cards to display based on their activating values. In this example, the activating value is a Boolean variable.

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

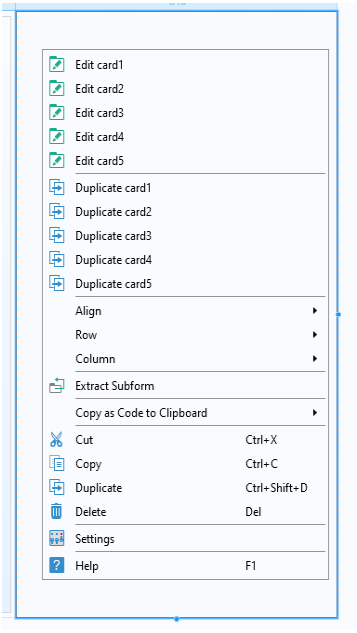


By default, the **Card type** is set to **Local**, which means that the card is defined locally in its containing card stack object. If the **Card type** is set to **Existing form**, then you can instead select one of the existing forms. The settings for an **Existing form** are accessed directly from 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**.

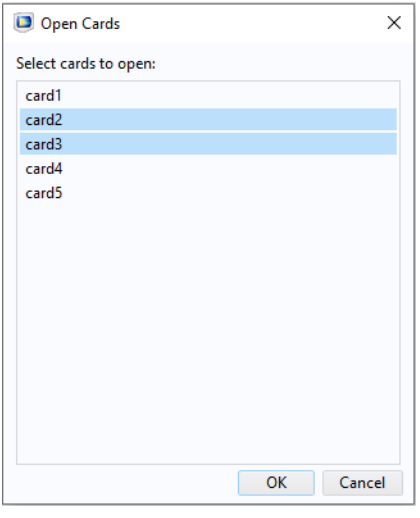


To access locally defined cards, right-click the card stack in a form window to select between the different cards in the card stack, as shown in the figure below.

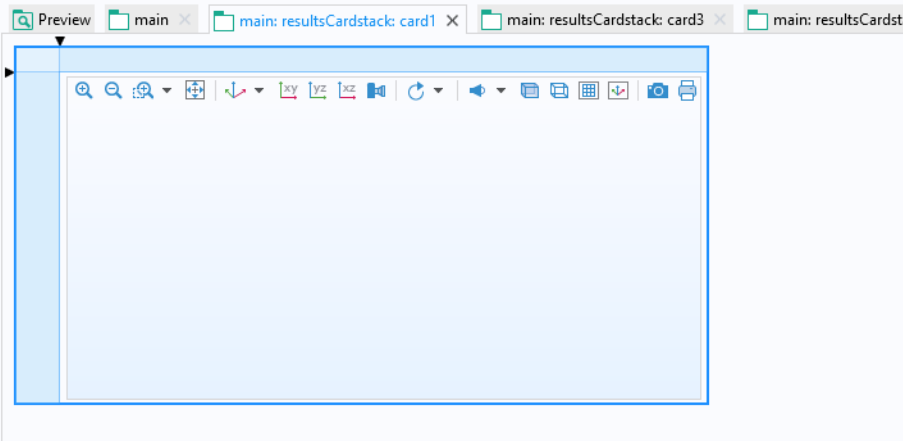


From this menu, you can also duplicate cards.

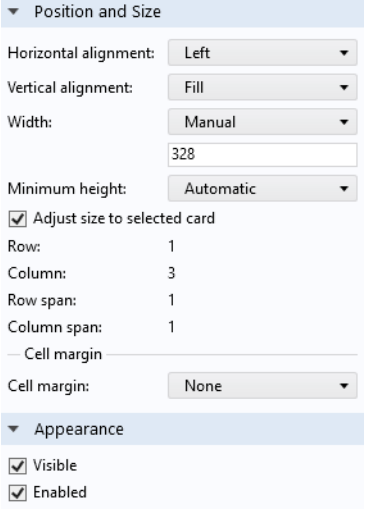
To edit cards, you can also use Alt+Click, which opens a dialog box that lets you select multiple cards at once.



The figure below shows `card1` of the card stack `resultsCardstack` with a graphics form object.



In the **Position and Size** section you can change the alignment and size of the card stack and cards.



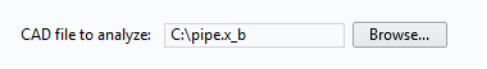
When selected, the check box **Adjust size to selected card** makes it possible to have the card stack adjust its size to the currently selected card. When not selected, the card stack will be as large as the largest card, regardless of which card is selected. Before version 5.6, the card stack always took the size of its largest individual card, which meant that even small or empty cards still took up space in the layout. Now,

when a card is empty, the card stack will disappear, which is a desirable feature in many cases. Using it you can for example have a dynamic documentation card stack appear and disappear depending on the user's actions.

File Import

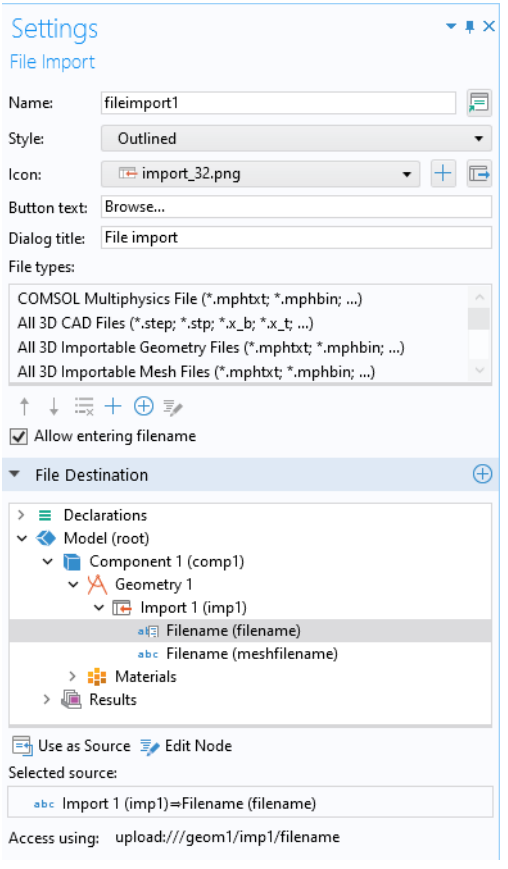
A **File Import** object is used to display a file browser with an associated input field for browsing to a file or entering its path and name. It is used to enable file import by the user of an application at run time, when the file is not available in the application beforehand.

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

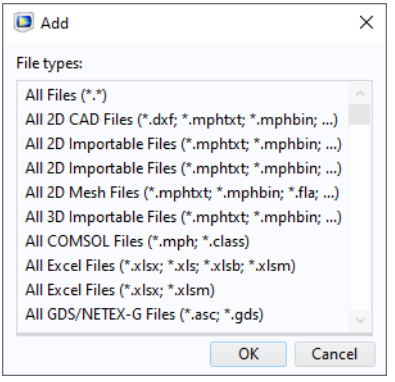


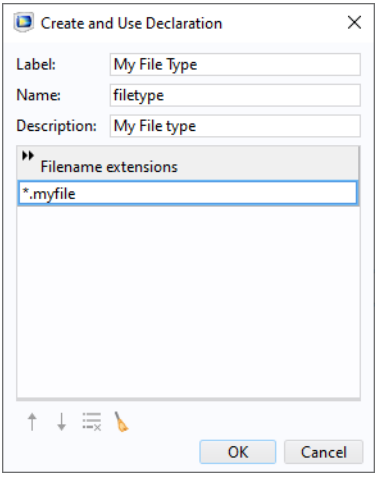
The corresponding **File Import** object is shown in the figure below.

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.

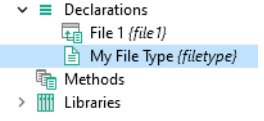


In this application, the **File types** table specifies that only CAD files are allowed. You can further control which **File types** are allowed by clicking the **Add** and **Delete** buttons below the list of **File types**. Clicking the **Add** button displays the dialog box shown to the right:





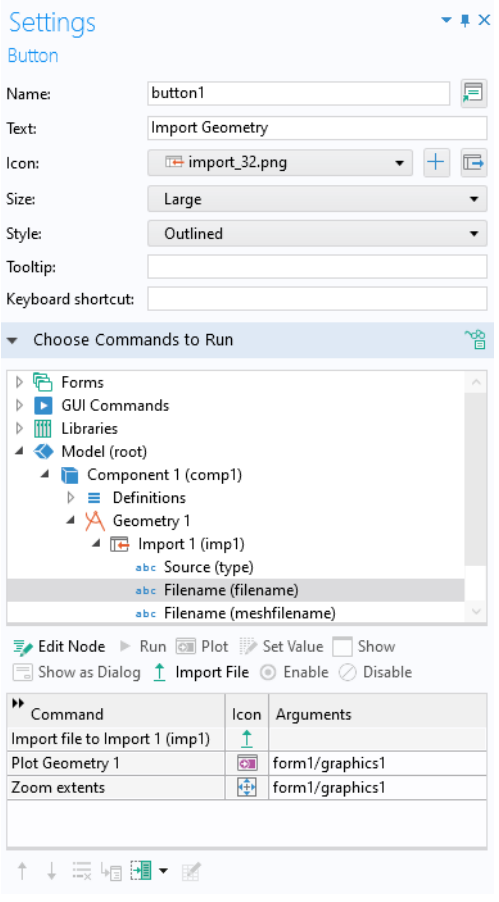
By clicking the **Create New File Type and Use It** you can use your own file format.The figure to the right shows a custom file type. Adding a custom file type in this way will also create a **File Type** declaration, as shown in the figure below.



Alternatives to Using a File Import Object

If an input field for the file path and name is not needed, then there are other methods for file import 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.



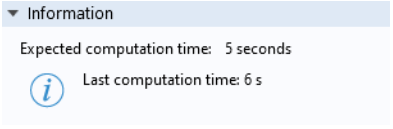
A **File Import** object can also reference a **File** declaration. For more information, see “File” on page 172. For more information on file handling in general, see “Appendix C — File Handling and File Scheme Syntax” on page 328.

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

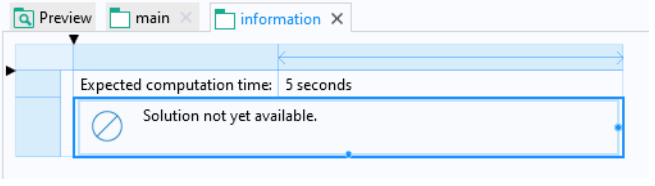
```
success=importFile("file1",new
String[]{"ALL_IMPORTABLE_IMAGES","PNG","JPEG","BMP","GIF"});
```

Information Card Stack

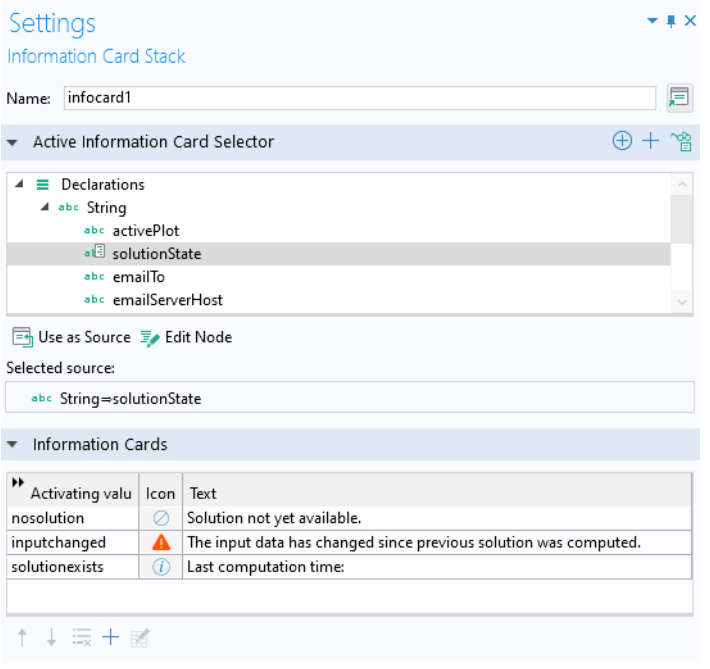
An **Information Card Stack** object is a specialized type of **Card Stack** object used to display information on the relationship between the inputs given by the user to an application and the solution. The figure below shows a portion of a running application in which an information card stack is used together with information on the expected computation time.



The corresponding form objects are shown below:

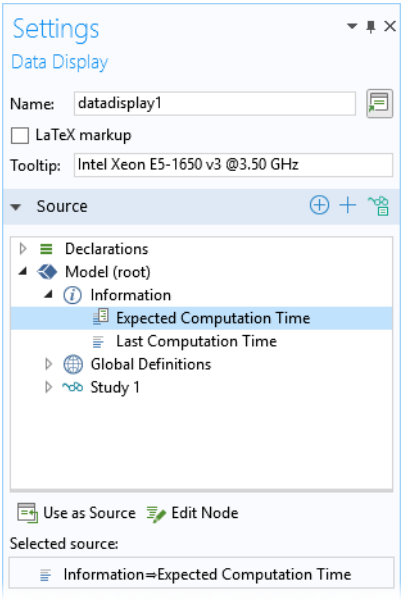


The figure below shows the **Settings** window where a string variable `solutionState` is used as the source.



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.

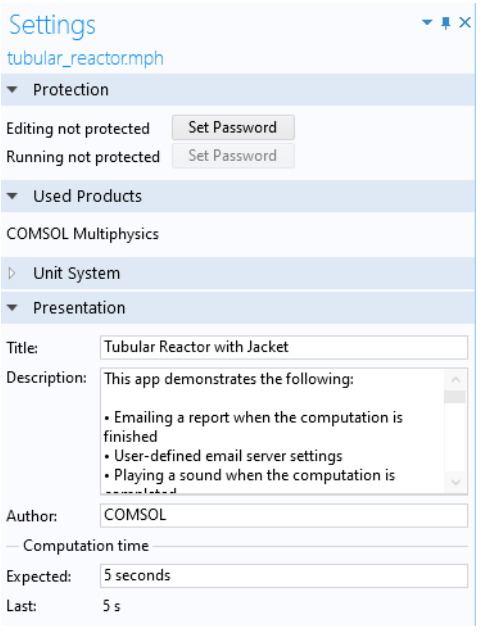


Note that information nodes in the model tree are only shown when working with the Application Builder. They are made available in the **Source** section in the **Settings** window for form objects, when applicable.

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.



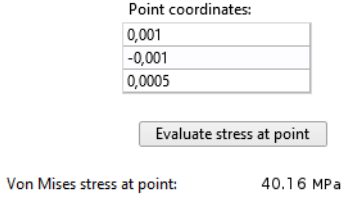
If the computation time is predominantly spent in a method, such as when the same study is called repeatedly, then you can manually measure the computation time by using the built-in methods `timeStamp` and `setLastComputationTime`. For more information, see “Date and Time Methods” on page 364.

Array Input

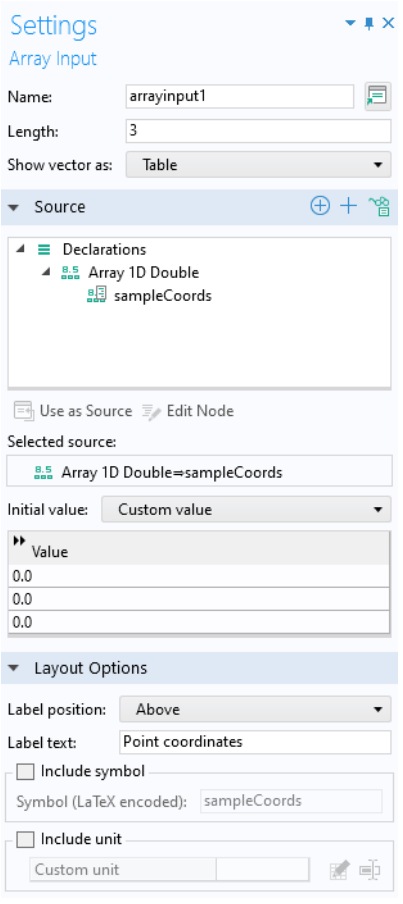
An **Array Input** object has an input table used to enter array or vector-valued input data. An array input object supports string arrays as data sources. You can add an optional label, symbol, and unit.

Using an Array Input Object for 3D Point Coordinate Input

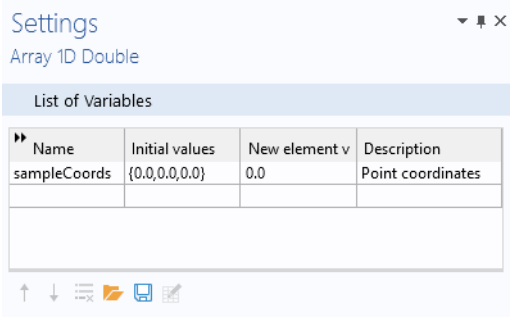
Consider an application where the user enters 3D coordinates for a point where the stress is evaluated. The figure below shows a screenshot from an application with an array input, button, text label, and data display object.



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 1D Double** with the following **Settings**.



In the **Settings** window of the array input object:

- In the **Length** field, enter the length of the array as a positive integer. The default is 3.
- From the **Show vector as** list, choose **Table** (the default) to show the array components as a table, or choose **Components** to show each array component as a separate input field with a label.
- In the **Value** table, enter the initial values for the components in the array.
- The **Layout Options** section provides settings for adding optional labels and units to the array input.

In this example, when the user clicks the button labeled **Evaluate stress at point**, the following method is run:

```
with(model.result().dataset("cpt1"));
  set("pointx", sampleCoords[0]);
  set("pointy", sampleCoords[1]);
  set("pointz", sampleCoords[2]);
endwith();
```

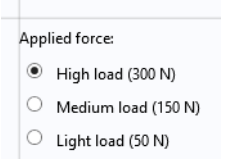
where the values `pointx`, `pointy`, and `pointz` will be used subsequently as coordinates in the evaluation of the stress.

Radio Button

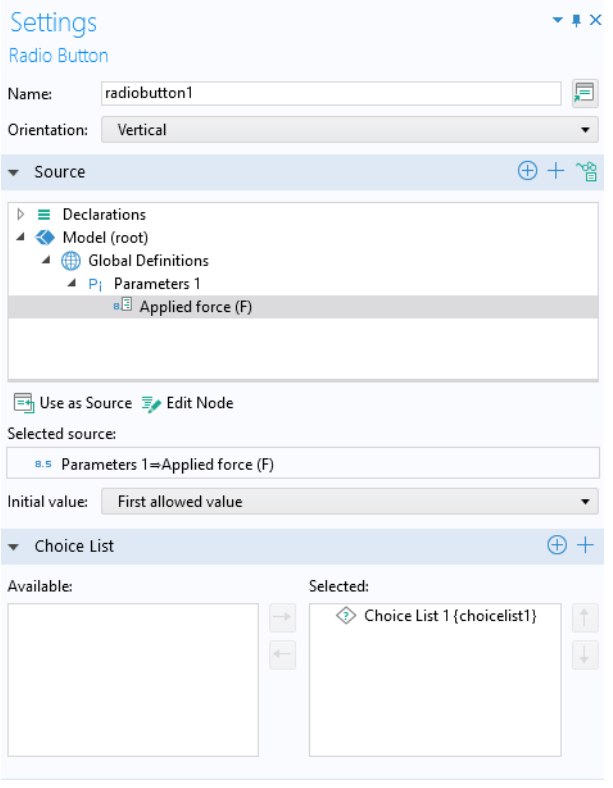
A **Radio Button** object has a fixed number of options from which you can choose one. It is most useful when you have just a handful of options.

Using Radio Buttons to Select a Load

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



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

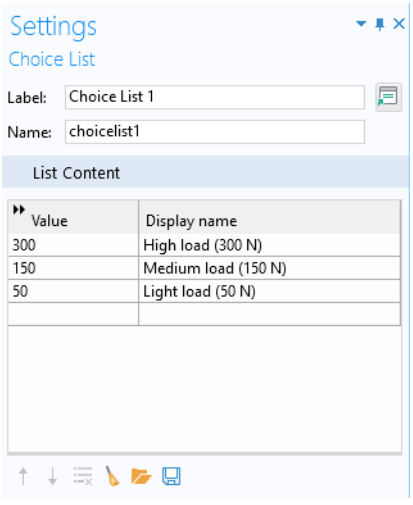


The **Orientation** can be set to **Vertical** (default) or **Horizontal**.

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

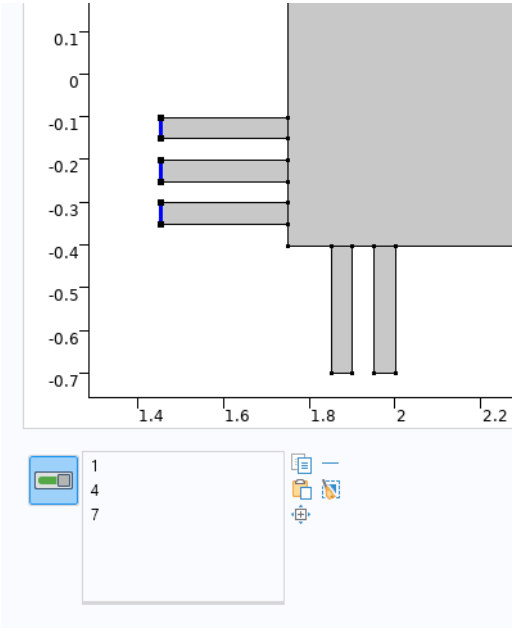
If the radio button will be used for the purpose of changing units, then a **Unit Set** can be used instead of a **Choice List** (You still select it in the **Choice List** section of the **Settings** window of the radio button object).

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 92.

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.

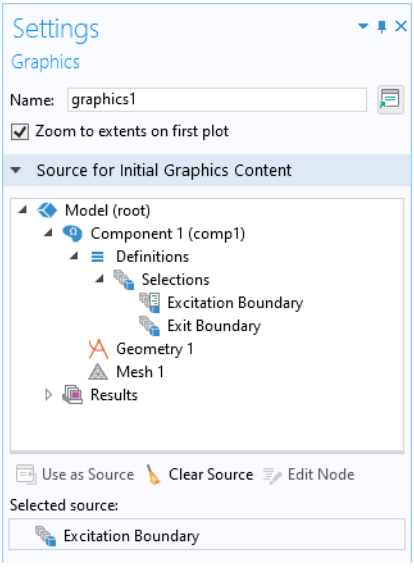
In the example below, the embedded model has a boundary condition defined with an **Explicit** selection. Both a **Selection Input** object and a **Graphics** object are used to let the user select boundaries to be excited by an incoming wave.



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.

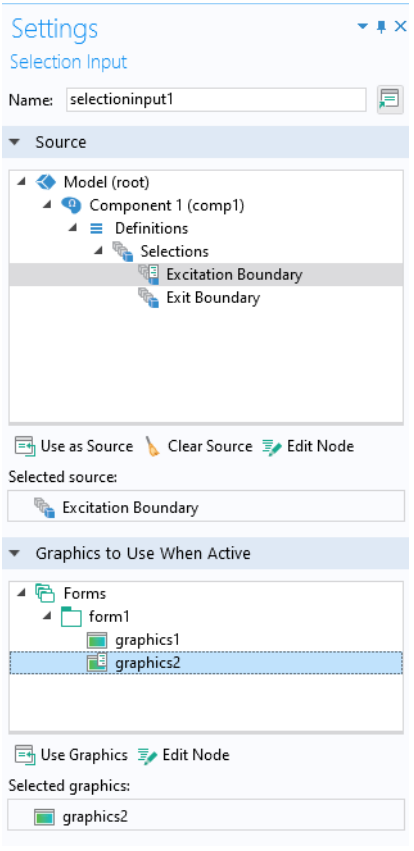
To make it possible to directly select a boundary by clicking on it, you can link a graphics object to an explicit **Selection** used to group boundaries, as shown in the figure below. Select the explicit selection and click **Use as Source**.

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**.



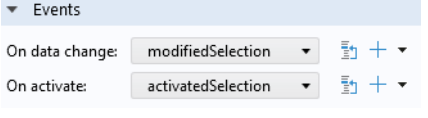
When a graphics object is linked directly to an explicit selection in this way, the graphics object displays the geometry and the user can interact with it by clicking on the boundaries. The boundaries will then be added (or removed) to the corresponding explicit selection.

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.



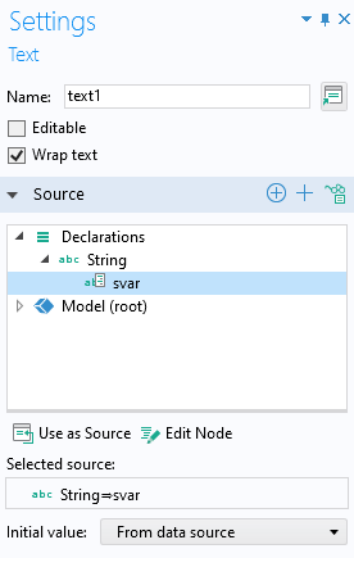
In a selection input object, you can copy, paste, remove, clear, and zoom into selections.

You can have events associated with selections. The **On data change** event will be triggered when the selection is changed. If you have a local method associated with this event, you will get a method with an integer array argument. The method is called with the new entities of the selection. The **On activate** event will be triggered when the **Activate Selection** button is clicked.



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.

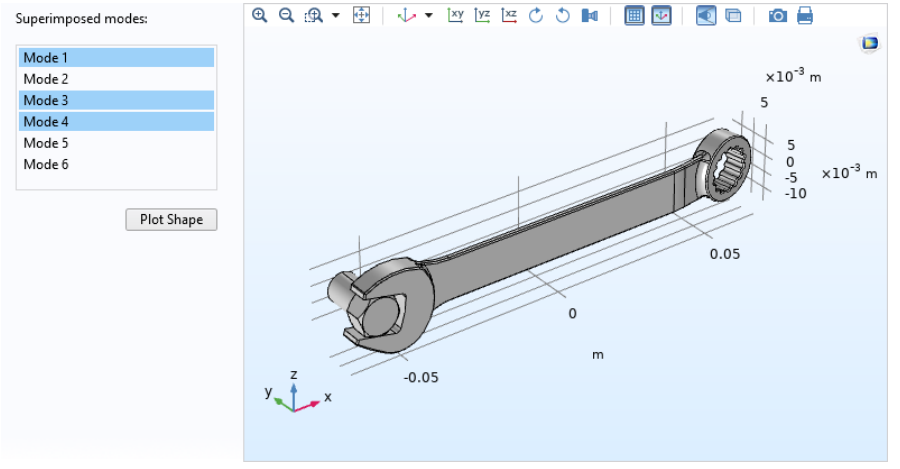
For more information on **Information** nodes, see “Data Display” on page 105.

List Box

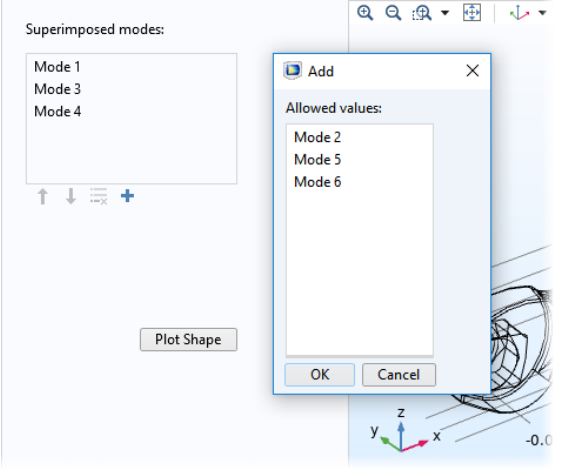
A **List Box** object is similar to a radio button object, except that it allows for the simultaneous selection of multiple options.

Using a List Box to Superimpose Vibrational Modes

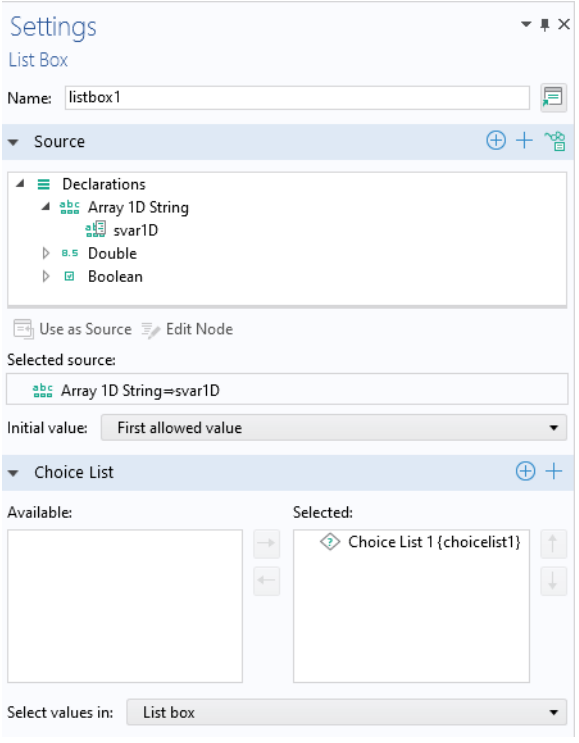
Consider an application where the first six vibrational modes of a mechanical part can be superimposed and visualized by selecting them from a list box, as shown in the figure below.



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



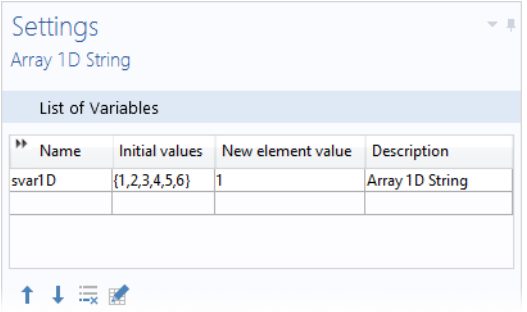
The **Settings** window for the list box of this example is shown in the figure below.



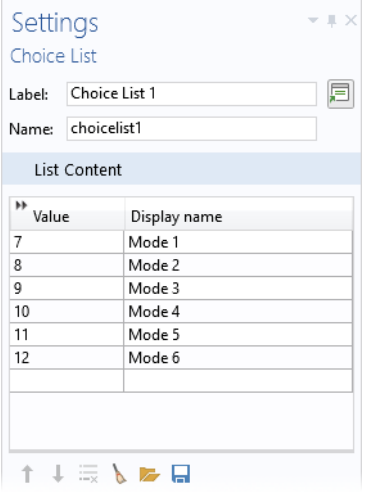
The **Select values in** list allows you to choose between two alternatives, **List box** or **Dialog**, for displaying the list.

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 `with(1,u)+with(3,u)+with(5,u)`. This expression is then used for the *x*-displacement (dependent variable `u`) in a displacement plot. In a similar way, the method automatically creates expressions for the variables `v` and `w` associated with the *y*- and *z*-displacement, respectively. Note that the command `with()`, used in 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 368.

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).

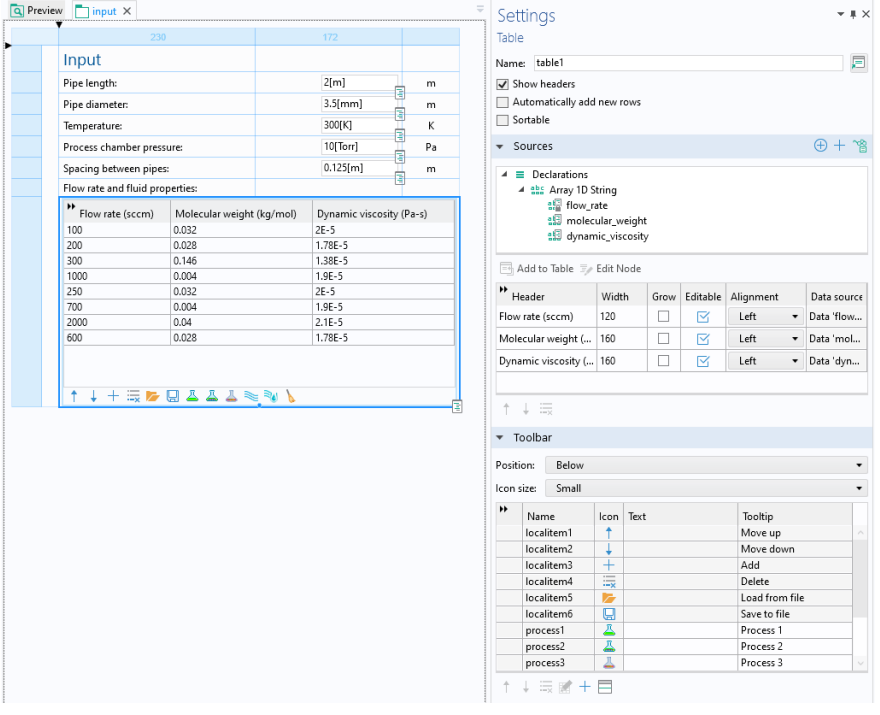
Table

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.



In this example, the data source references three 1D string arrays. You can select any type of array as the source and then click **Add to Table**.

Three check boxes control the overall appearance of the table:

- **Show headers**
- **Automatically add new rows**
- **Sortable**

The **Automatically add new rows** check box ensures that an additional empty row is always available when a user is filling out a table. If all of the 1D string arrays, which are used as a source to the table, have nonempty values for **New element value** in their declaration **Settings** window, then this functionality is deactivated. In this case, new rows can only be added by clicking the **Add** button in the associated table toolbar, if such a button has been made available.

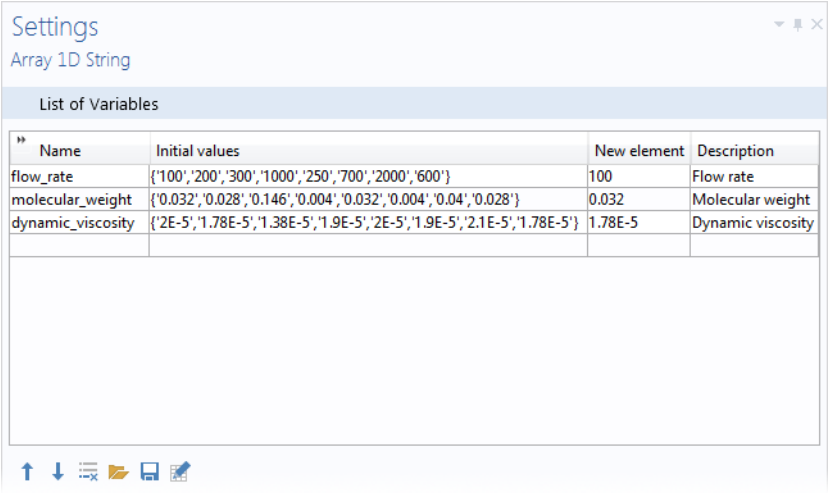
The **Sortable** check box makes it possible to sort the table with respect to a particular column by clicking the corresponding column header.

The **Sources** section contains a table with five columns:

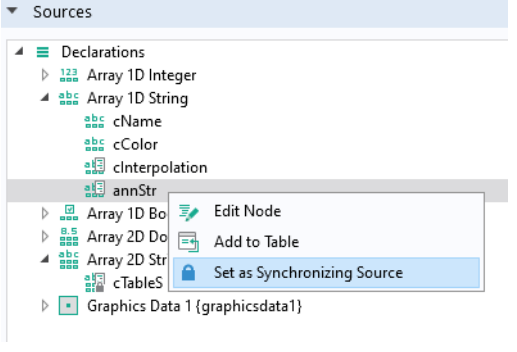
- **Header**
- **Width**
- **Grow**
- **Editable**
- **Alignment**
- **Data source**

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:



You can control which source that determines the number of rows in the table. To set this, right-click the corresponding array declaration in the **Sources** section and select **Set as Synchronizing Source**.



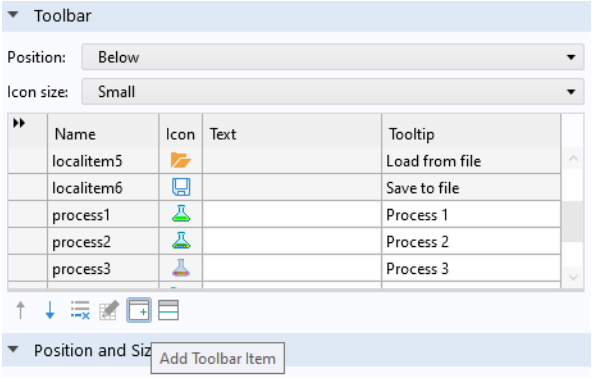
TOOLBAR

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:

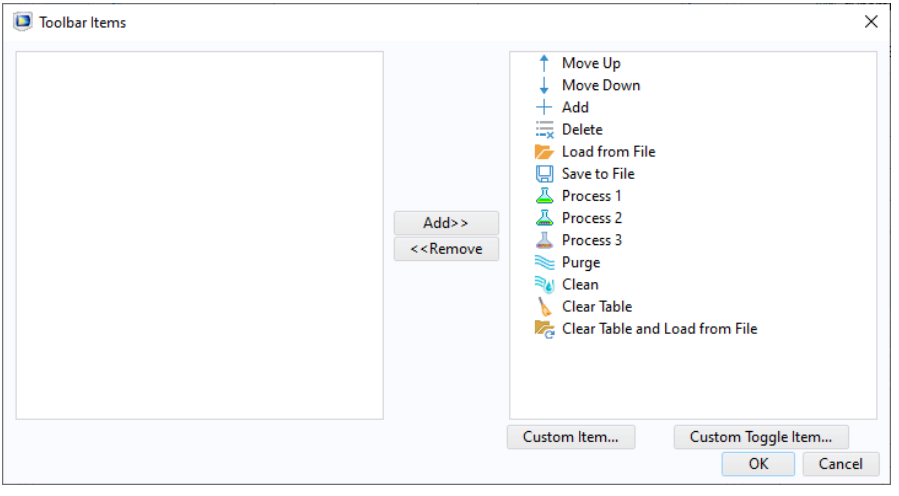
- **Below**

- **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.



The following dialog box is then shown.

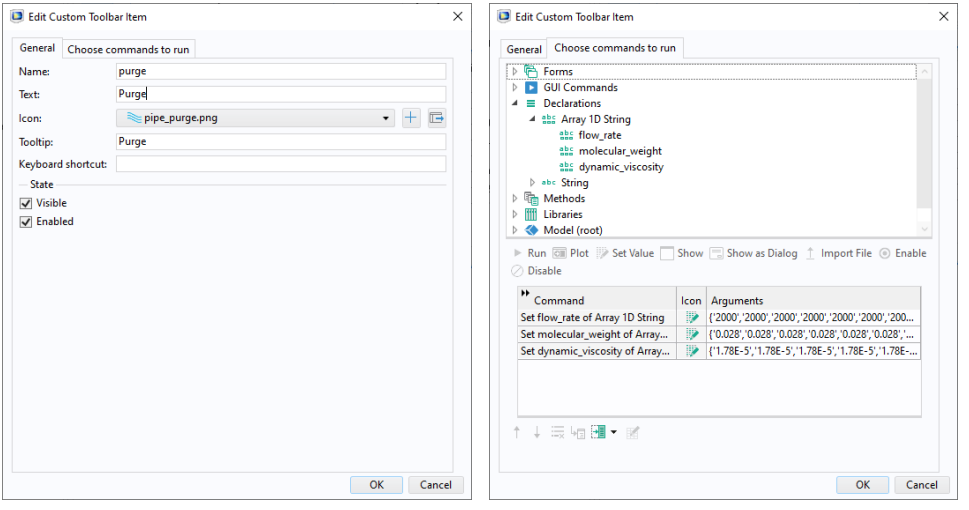


You can always add the following items:

- **Move Up**
- **Move Down**
- **Add**
- **Insert Above**

- **Delete**
- **Clear Table**
- **Load from File**
- **Clear Table and Load from File**
- **Save to 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.

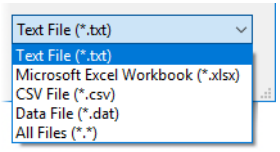


The **Choose commands to run** tab is similar to that of menu, ribbon, and toolbar items, as well as buttons.

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

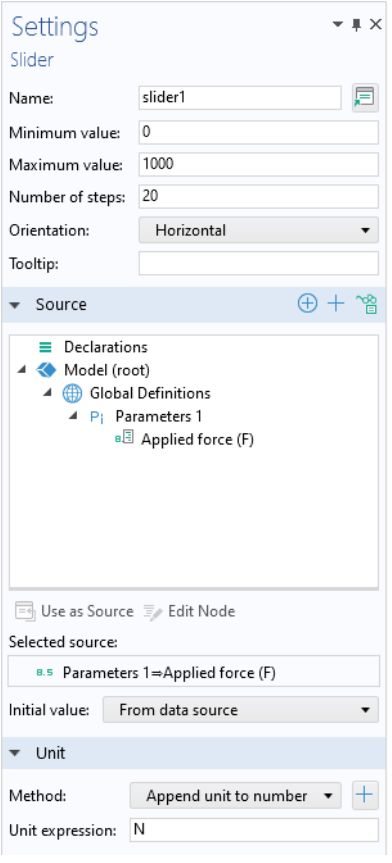
A **Slider** is a form object for choosing numerical input using a slider control.

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.

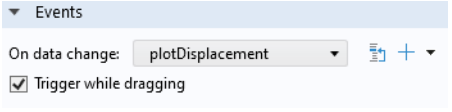


In this example, the slider uses a global parameter `F` as its source. You can select any parameter, variable, or declared scalar variable as a source. Select from the application tree and click **Use as Source**.

You determine the range of values for the data source by defining the **Minimum value**, **Maximum 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 174 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.



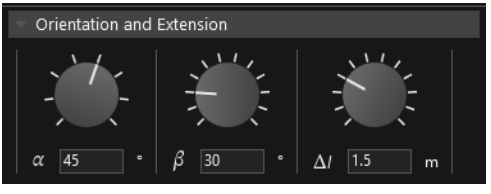
This setting can be useful if the method that is called by the **On data change** event is computationally heavy, so that there is a lag when dragging the slider.

Knob

A **Knob** is a form object for choosing numerical input using a control knob, similar to a slider.

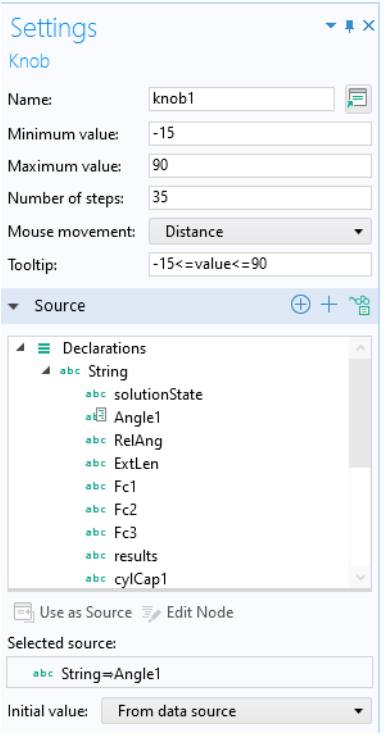
Using a Knob to Change the Angle of a Crane Arm

Consider an application where the angle of a truck mounted crane arm can be changed by control knobs, such as in the figure below (which uses a dark theme).



In this example, the knobs are accompanied by input fields that are used to display the selected value.

The **Settings** window of one of the knobs is shown in the figure below.



In this example, the knob uses a string variable `Angle1` 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**.

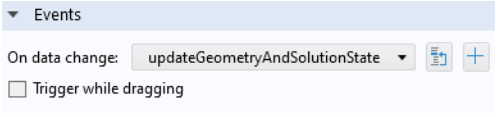
You determine the range of values for the data source by defining the **Minimum value**, **Maximum value**, and **Number of steps** for the knob.

The **Mouse movement** can be **Distance**, **Vertical**, or **Circular**. **Distance** changes the value with a linear mouse movement in any direction. **Vertical** changes the value when you move the mouse vertically. **Circular** changes the value when you make a circular mouse movement. A physical control knob is usually controlled with a circular movement. However, when using a mouse this is usually not the most convenient way. Instead, use a linear mouse movement by selecting **Distance** or **Vertical**.

You can also set a **Tooltip** that is shown when hovering over the knob. The settings for units are similar to that of a slider.

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

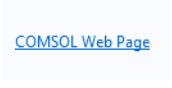
In the **Events** section, in addition to specifying which method to call for an **On data change** event, you can select the check box **Trigger while dragging**. This setting determines if the event method should be called continuously while the knob is being dragged or only upon its release.



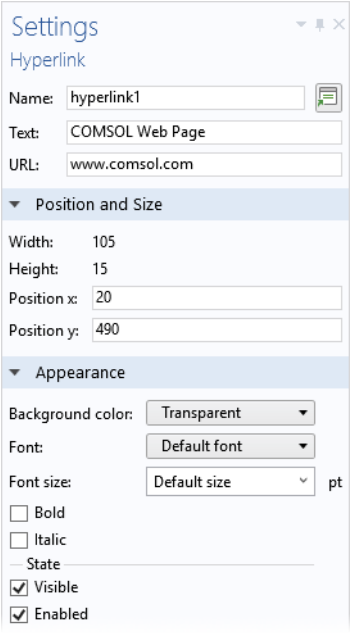
This setting can be useful if the method that is called by the **On data change** event is computationally heavy, so that there is a lag when dragging the knob.

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.



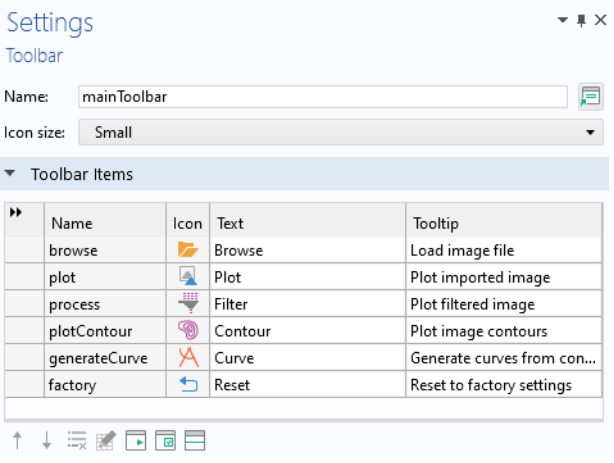
The **Hyperlink** object supports the types of URLs that you can use in a web browser, including:

- 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 356.

Toolbar

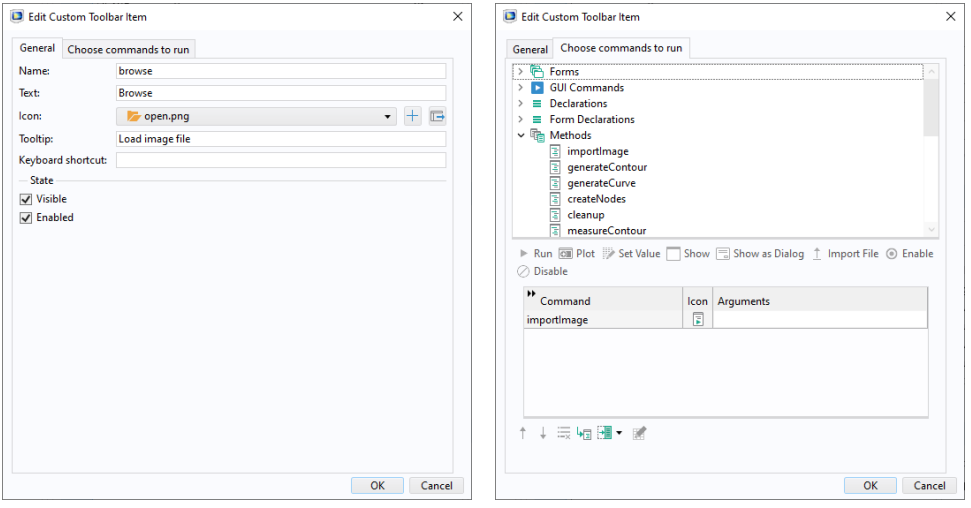
A **Toolbar** object contains the specifications of a toolbar with toolbar buttons. The figure below shows a toolbar with several buttons.

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



Each row in the **Toolbar Items** table contains either an **Item** or **Toggle Item** corresponding to a toolbar button or toggle button, respectively, 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 the **Browse** item.

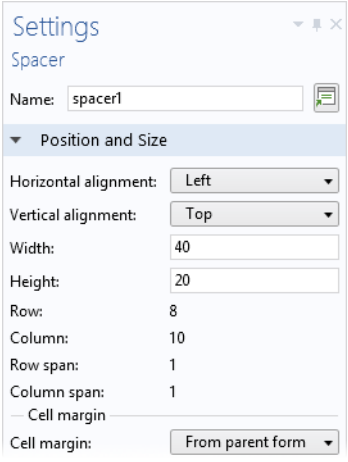


The text in the **Tooltip** field will be shown as a tooltip when hovering over the toolbar button. The text in the **Text** field will be shown next to the icon, if any; otherwise just the text is shown. Similarly you can choose to just have an icon and no text. The **Icon** list, the **Keyboard shortcut** field, and the **Choose commands to run** tree represent the same functionality as a button object. For more information, see “Button” on page 67.

Spacer

A **Spacer** object is invisible in the user interface and is only used when working in grid layout mode. It defines a space of fixed size that you can use to ensure that neighboring form objects have enough space to show their contents. Typically, you would use a spacer next to a table or graphics object to ensure that they are rendered properly. If the user resizes the window so that it becomes smaller than

the size of the spacer, the effective size of the window is maintained by displaying scroll bars. The figure below shows the **Settings** window of a spacer object.



Appendix B — Copying Between Applications

Many nodes in the application tree can be copied and pasted between applications, including: forms, form objects, menu items, methods, Java® utility methods, external libraries, file declarations, choice list declarations, menus, menu items, ribbon sections, ribbon tabs, and ribbon items.

When you copy and paste forms, form objects, and items between applications, the copied objects may contain references to other objects and items. Such references may or may not be meaningful in the application to which it is copied. The following set of rules apply when objects are pasted from the clipboard:

- A declaration referenced in a form object or menu item is included when copying the object, but is not necessarily pasted. It is only pasted if there is no compatible declaration present. If a compatible declaration exists, that is used instead. A compatible declaration is defined as one having the same name and type. For example, a string declaration is not compatible with an integer declaration. An existing declaration may have an invalid default, but no such check is done when pasting.
- A referenced global parameter may have a different unit, but will still be considered compatible.
- A form or form object directly referenced from another form object is not included automatically when copying objects. The direct reference will point to an existing object if it has the same name. If the original reference is among the copied objects, then that object will be used in the reference instead of any existing objects having the same name. The name of the copied reference will be changed to avoid name collisions.
- No objects in the model tree will be automatically copied, for example, a graphics object referring to a geometry or an input field referring to a low-level setting exposed by 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).

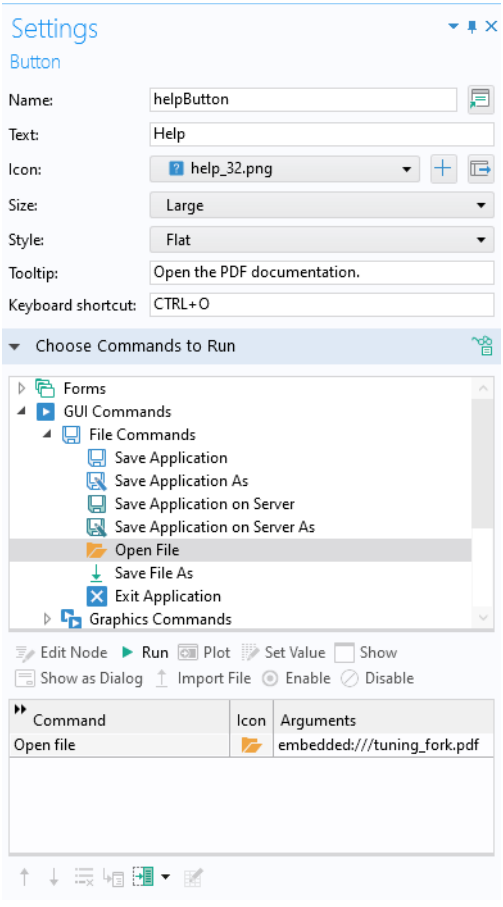
However, there are techniques for transferring files to and from the client file system when running an application both with a web browser and the COMSOL Client.

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 334.

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 341.

Saving and Opening Files using File Commands

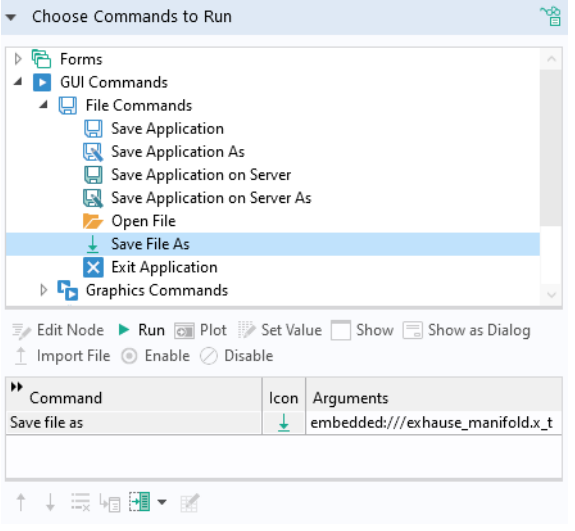
In the editor tree used in a command sequence, the **File Commands** folder contains commands to save and load applications and files, as well as exiting an application.



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 331.

To open files from a method, use the built-in method `fileOpen`; see also “Operating System Methods” on page 356.

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 362. For more information on saving and exporting files, see “File Export” on page 341.

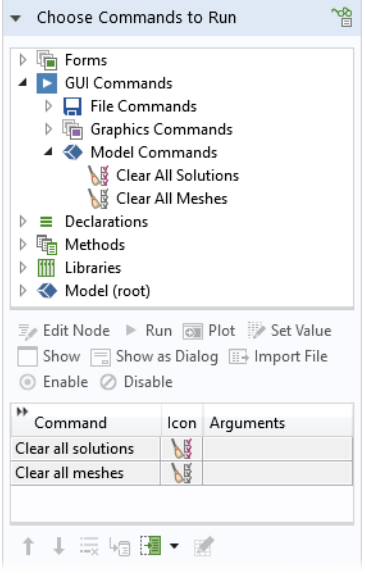
The **Save Application** and **Save Application As** commands are available for use in the command sequence for certain form objects. The **Save Application As** command will display a **Save As** dialog box where the user can specify a client path where the entire application will be saved.

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 362.

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 `dbfile` file scheme is for file versions that are stored in a Model Manager database.
- 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.

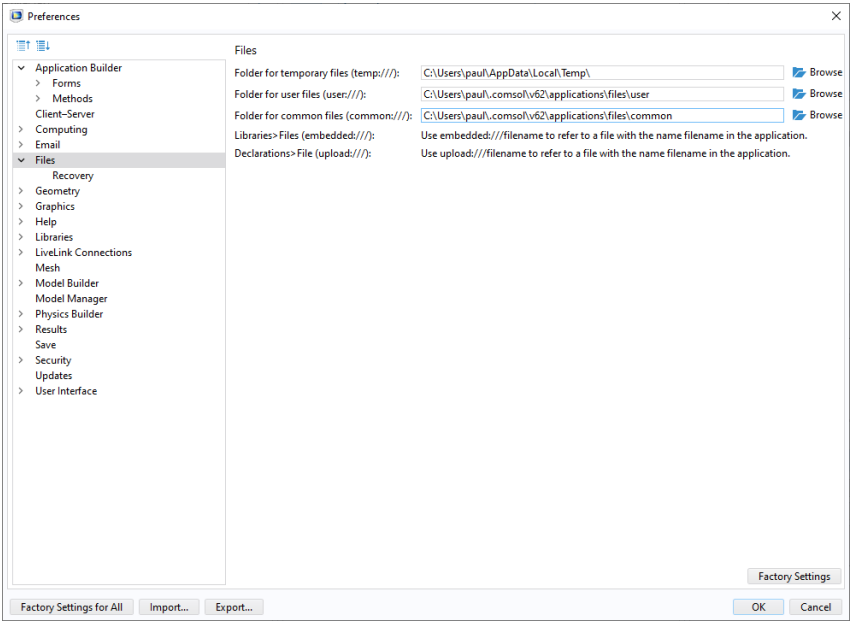
SCHEME	REFERS TO	DEFAULT PATH	TYPICAL USAGE
`embedded:///`	Files embedded in the application using Libraries>Files.	N/A	Experimental data, CAD files, mesh files, interpolation data
`upload:///`	Files to be uploaded by the user at run time.	Determined by the Target directory in the Settings window of 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
`dbfile:///`	File versions stored in a Model Manager database.	N/A	Experimental data, CAD files, mesh files, interpolation data

For more information on files in the Libraries node accessible by the `embedded:///` syntax, see “Libraries” on page 233.

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. The `dbfile` file scheme is available for all usages.

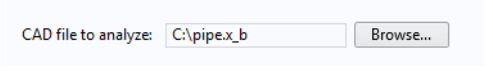
USAGE	EMBEDDED	UPLOAD	TEMP	USER	COMMON
File is used as input	√ (r)	√			√
File is output			√ (r)	√	
Method reading a file	√ (r)	√	√	√	√
Method writing a file			√ (r)	√	
File is client-side	√	√	√ (r)	√	√

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



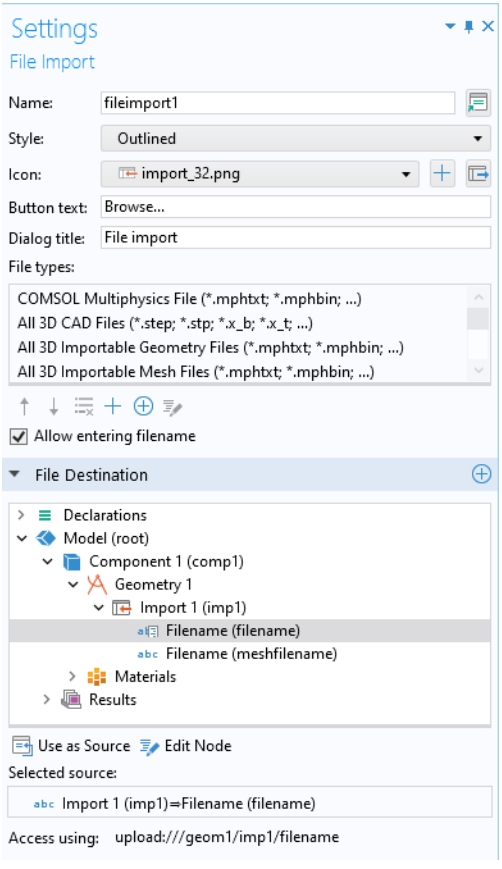
CAD Import using the Model Tree and a File Import Object

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.

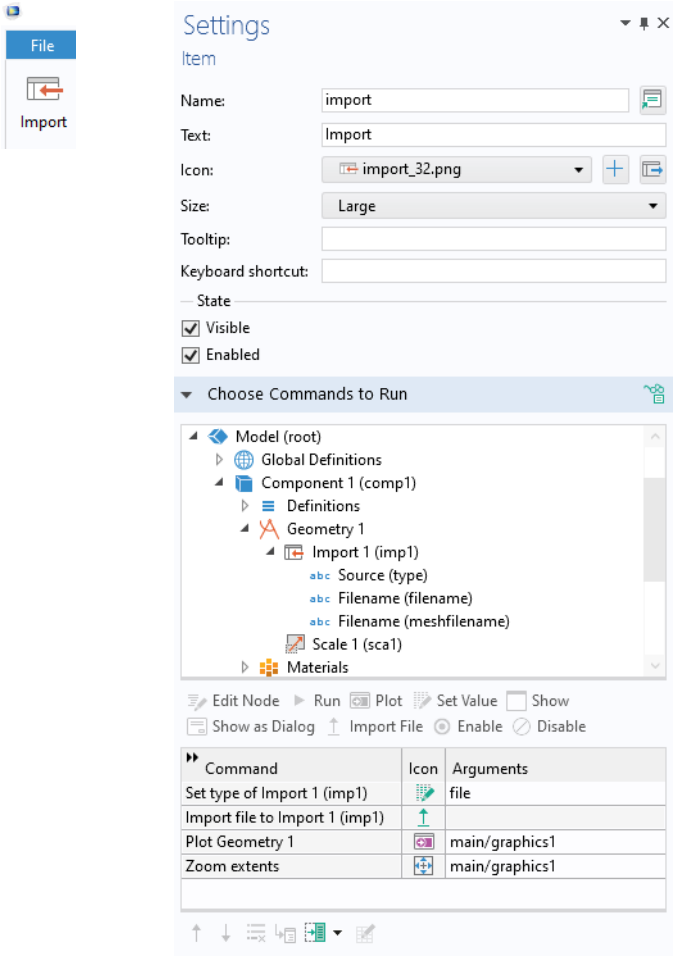


If you do not wish to use a **File Import** object, you can directly reference a **Filename** from a button or an item in a menu, ribbon, or toolbar, or alternatively create a method that calls the built-in method `importFile` as an event, for example

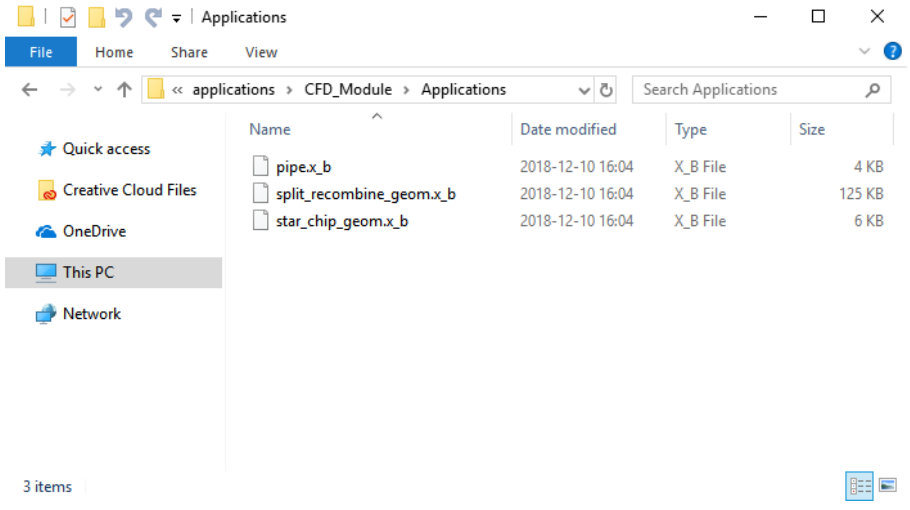
```
importFile("file1");
```

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 1** will open a file browser for the user to select a file, as shown in the figure below.



The preceding command **Set Type of Import** allows you to filter the file extensions displayed in the file browser. The available arguments are: `file`, `mesh`, `native`, `cad`, and `ecad`.

The subsequent commands build and plot the geometry and zoom out using zoom extents.

For more information on the **File Import** object, see "File Import" on page 291.

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:

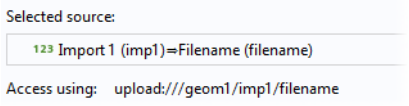
```
importFile(model.geom("geom1").feature("imp1"), "filename");
useGraphics(model.geom("geom1"), "main/graphics1");
zoomExtents("main/graphics1");
```

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 354 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**

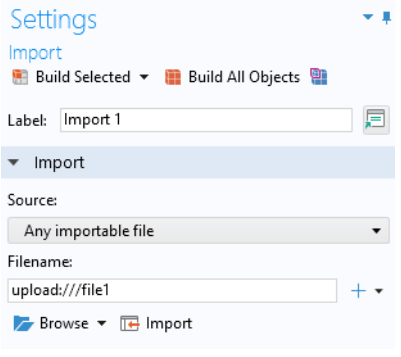
using:). The figure below shows an example where a **File Destination** and **Filename** are used.



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 172.

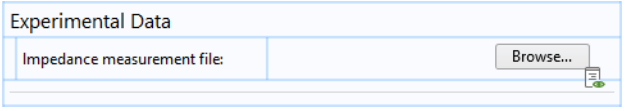
This syntax can also be used in any file browser text fields within the Model Builder nodes. The figure below shows a file reference used in the **Filename** field of the **Import** model tree node for a model using geometry import.



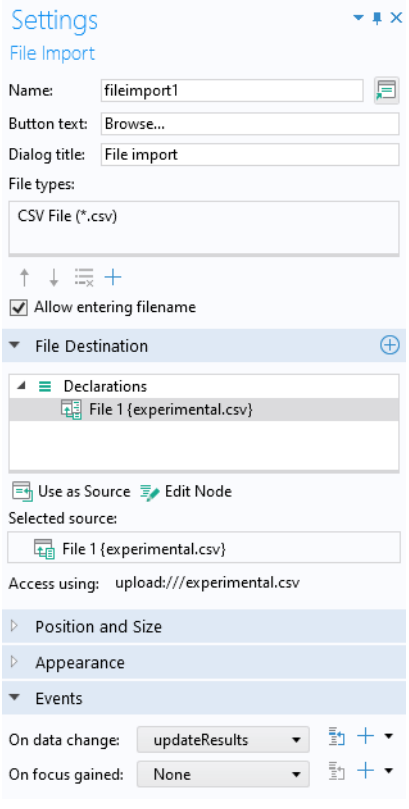
However, a quicker way is to link a file import object directly to the **Filename** field, as described previously in the section "CAD Import using the Model Tree and a File Import Object" on page 334.

Importing Experimental Data

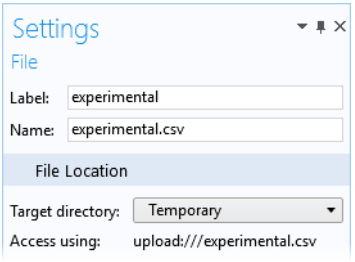
Consider an application where the user is providing a file with experimental data at run time. The figure below shows the file import object of such an application as it appears in grid layout mode.



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.

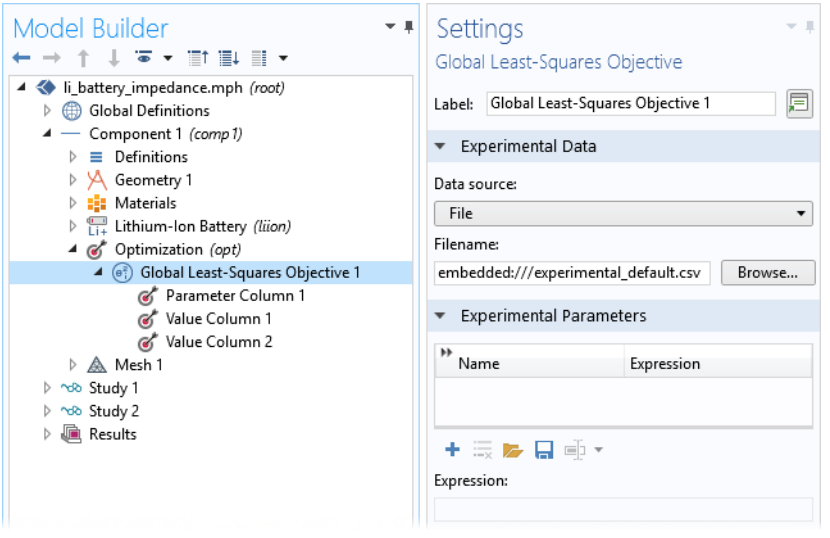


The file declaration serves as the "destination" of the imported data, which is written to the file `upload:///experimental.csv`.

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

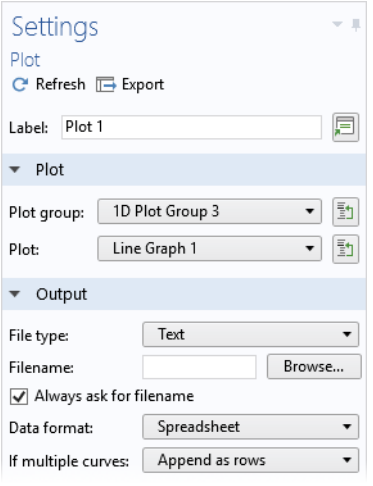
File Export using the Model Tree

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.

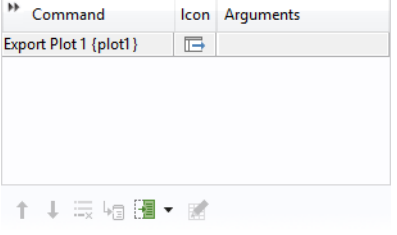
In the model tree, the **Export** node may contain several types of subnodes for file export, including:

- **Data**
- **Plot**
- **Mesh**
- **Table**
- **3D Image**
- **2D Image**
- **1D 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.

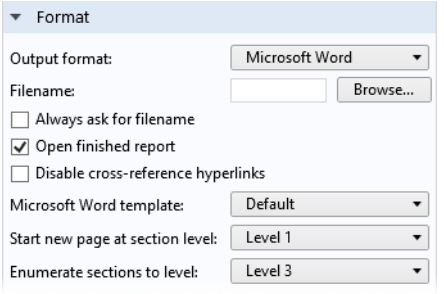


You can leave the **Filename** field blank, as shown in the figure above. In the command sequence of, for example, a button, you can run the corresponding **Export > Plot** node and, at run time, it will open a file browser window for the user to select a location and file name, as seen in the figure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



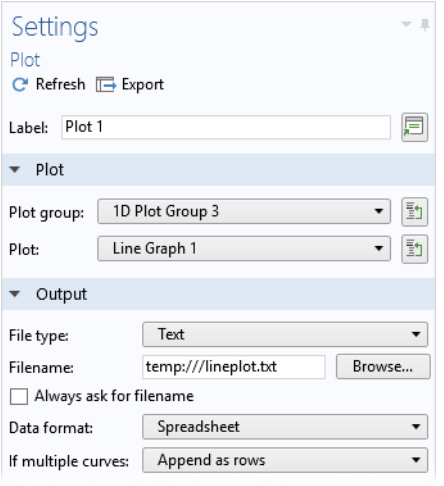
By running a **Report** subnode, a file browser window is opened for the user to select a location and file name for the report.

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.

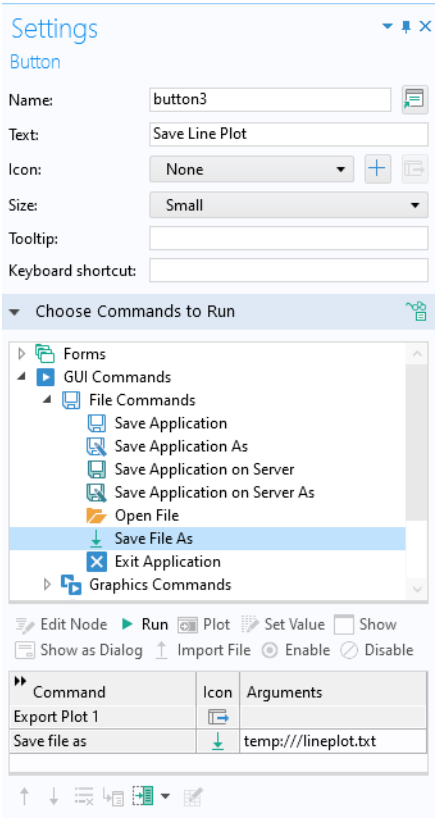
The example below shows the **Settings** window of an **Export > Plot** node that is used to export plot data as numerical values.



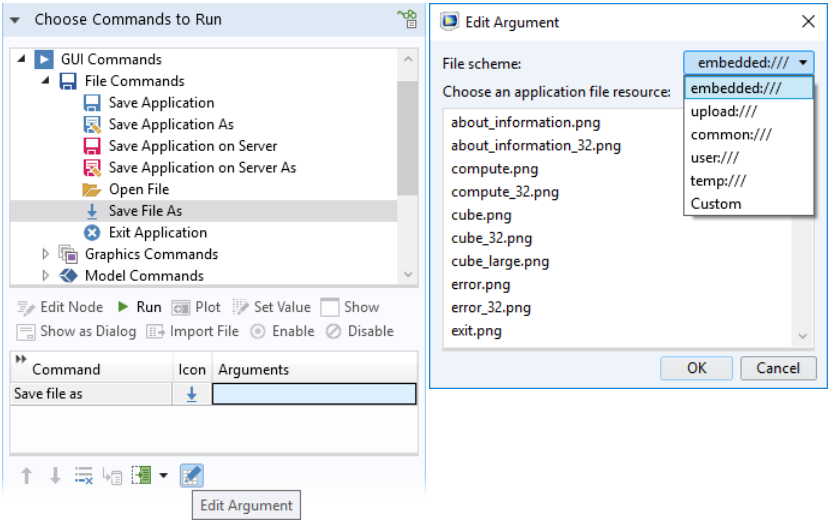
The **Filename** in its **Output** section is set to `temp:///lineplot.txt`.

To make it possible to save the plot 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

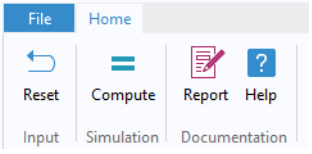
Note that as a first step, in the example above, the file is written to a temporary file, using the call to `model.result().export("plot1").run()`. This step is done automatically by the application. In the second step, the method `fileSaveAs` opens a file browser and lets the user of the application 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*.

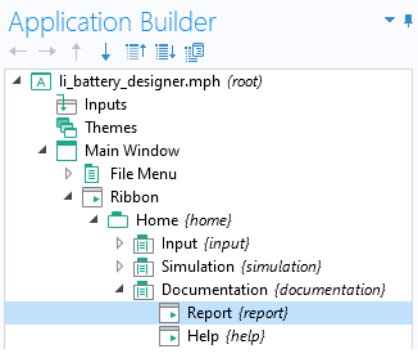
Creating Reports using Low-Level Functionality

This section describes creating reports using low-level functionality. For a more direct method, see "File Export" on page 341.

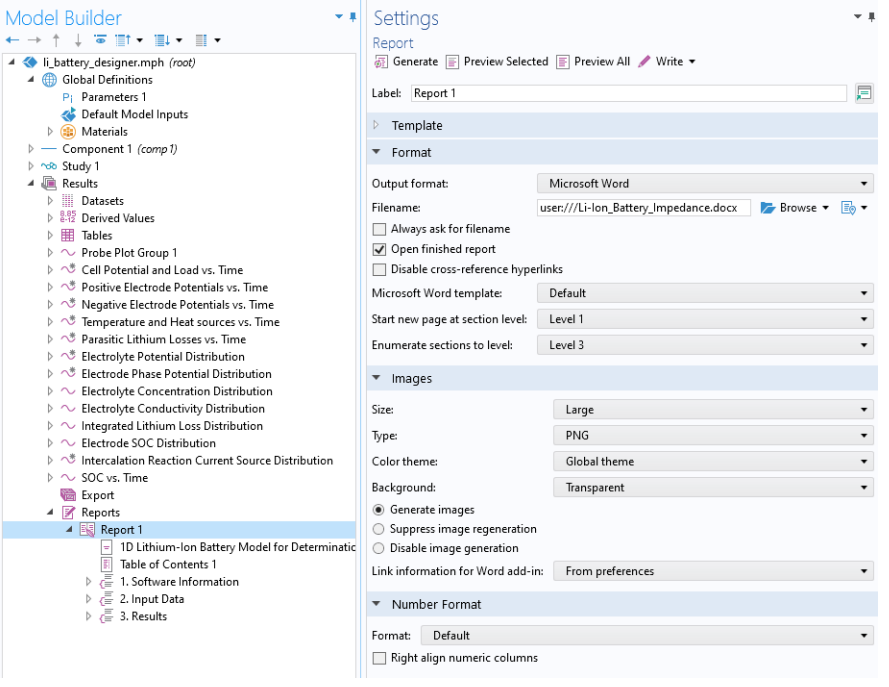
The example below shows an application where a report in the Microsoft® Word format (`.docx`) can be saved by the user. The figure below shows a tab in the ribbon of the application. In this tab, there is a **Report** button in the **Documentation** section.



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



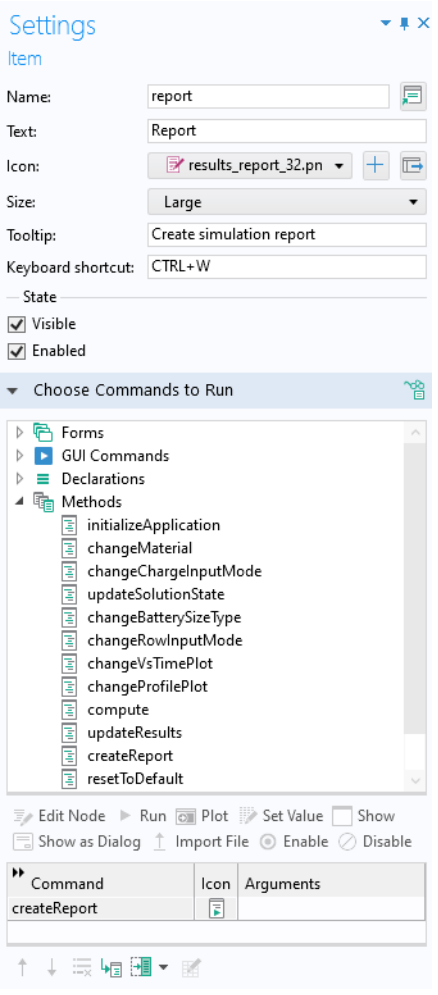
The following figure shows how the syntax `user:///` was used in the **Filename** field in the **Settings** window of the **Report** node of the Model Builder.



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 `createReport` includes the following call:

```
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. For a list of additional keyboard and mouse shortcuts, see the book *Introduction to COMSOL Multiphysics*.

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		√	
Pause	Break or suspend a running method as soon as possible			√
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)			√

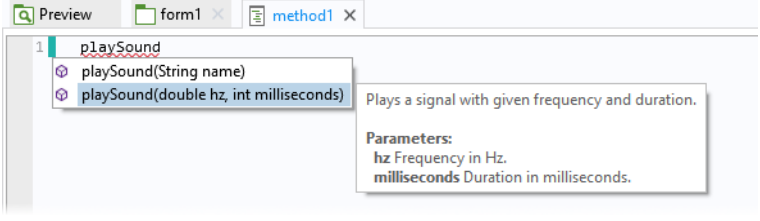
SHORTCUT	ACTION	APPLICATION BUILDER	FORM EDITOR	METHOD EDITOR
F7	Step into (in debugger)			√
Shift+F7	Step out of a method (in debugger)			√
F8	Run a method or method call. Create an executable or an add-in.	√		
F9	Check syntax			√
Ctrl+F	Find and replace text in methods			√
Ctrl+Space, Ctrl+/, or Ctrl+OEM2	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++	Ctrl key and plus key: Zoom in, in form editors, method editors, and the Main Window editing window.	√	√	√
Ctrl+-	Ctrl key and minus key: Zoom in, in form editors, method editors, and the Main Window editing window.	√	√	√
Ctrl+0	Reset zoom to 100% in form editors, method editors, and the Main Window editing window.	√	√	v
Ctrl+Alt+0	Zoom to fit the available space in form editors and the Main Window editing window.	√	√	
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		√	

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

File Methods

The file methods support, in addition to reading and writing to data files on the file system, reading and writing of data files stored in a Model Manager database by accepting a file location URI.

NAME	DESCRIPTION
`readFile`	Returns the contents in a given file as a string.
`openFileStreamReader`	Returns a `CsReader` object that can be used to read line-by-line or character-by-character from a given file `name`.
`openBinaryFileStreamReader`	Returns a `CsBinaryReader` object that can be used to read from a given file byte-by-byte.
`readMatrixFromFile`	Reads the contents of the given file into a double matrix. The file has the same spreadsheet-type format as available in the model tree Export node.
`readStringMatrixFromFile`	Reads the contents of the given file into a string matrix. The file has the same spreadsheet-type format as available in the model tree Export node.
`readCSVFile`	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.
`writeFile`	Writes array `data` to a given file. If the spreadsheet format is used, then the data can be read by `readMatrixFromFile` or `readStringMatrixFromFile`.
`openFileStreamWriter`	Returns a `CsWriter` object that can write to a given file.
`openBinaryFileStreamWriter`	Returns a `CsBinaryWriter` object that can be used to write to a given file byte-by-byte.
`writeCSVFile`	Writes a given double or string array to a CSV file. The RFC 4180 format is used for the CSV.
`exists`	Tests whether a file with a given name exists.
`deleteFile`	Deletes a file with a given name if it exists. The file is deleted on the server.
`copyFile`	Copies a file on the server. Both the source and target names can use file scheme paths.
`importFile`	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.

NAME	DESCRIPTION
`writeExcelFile`	Writes the given string array `data` starting from a specified cell in a specified sheet of an Excel file.
`readExcelFile`	Reads a specified sheet of an Excel file, starting from a specified cell, into a 2D string array.
`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
`executeOSCommand`	Executes the OS command with a given command (full path) and parameters. When applicable, the command is run server side.
`fileOpen`	Opens a file with the associated program on the client. See also the section "File Methods".
`getUser`	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.
`openURL`	Opens a URL in the default browser on the client.
`playSound`	Plays a sounds on the client.

Email Methods

NAME	DESCRIPTION
`emailFromAddress`	Returns the email from address from the COMSOL Server or preferences setting.
`sendEmail`	Sends an email to the specified recipient(s) with the specified subject, body text, and zero or more attachments created from Report, Export, and Table nodes in the embedded model.
`userEmailAddress`	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

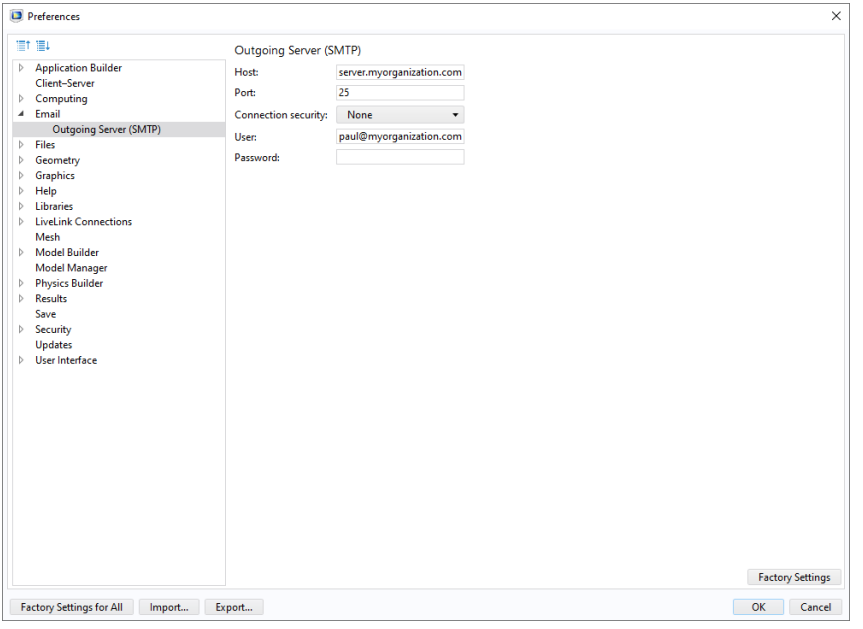
The class `EmailMessage` can be used to create custom email messages.

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

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



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, `method1()`, `method2()`.
`callMethod`	Alternate way to call a method from the Methods list; used internally and in cases of name collisions.
`useGraphics`	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.
`openForm`	Shows the form with the given `name` in the current main window. In a single window application, the form replaces the current one. In an application with subwindows, the form must exist in the main window layout. If not, it will not be opened. Showing a form that is already open will only activate the form.
`closeForm`	Closes the form with the given `name`. Closing forms is only possible in applications using subwindows. This method is not applicable for single window applications.
`closeDialog`	Closes the form, shown as a dialog box, with a given name.
`dialog`	Shows the form with a given name as a dialog box. Equivalent to the `dialog` method of a `Form` object; see below.
`alert`	Stops execution and displays an alert message with a given text.
`alert`	Stops execution and displays an alert message with a given text and title.
`confirm`	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".
`error`	Stops execution and opens an error dialog box with a given message.
`request`	Stops execution and displays a dialog box with a text field, requesting input from the user.
`message`	Sends a message to the message log.
`clearLog`	Clears the log window.
`clearMessageLog`	Clears the message log window.

NAME	DESCRIPTION
`evaluateToResultsTable`	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.
`evaluateToDoubleArray2D`	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.
`evaluateToIntegerArray2D`	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.
`evaluateToStringArray2D`	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.
`useResultsTable`	Shows the values from the `tableFeature` in the `resultsTable` form object.
`getChoiceList`	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 *Application Programming Guide*.
`setFormObjectEnabled`	Sets the enable state for the form object specified by the name or name path.
`setFormObjectVisible`	Sets the visible state for the form object specified by the name or name path.
`setFormObjectText`	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.
`setFormObjectEditable`	Sets the editable state for the form object specified by the name or name path. This functionality is only available for text field objects.

NAME	DESCRIPTION
`setMenuBarItemEnabled`	Sets the enable state for the menu bar item specified by the `name` or name path (from the menu bar) in the first argument.
`setMainToolbarItemEnabled`	Sets the enable state for the main toolbar item specified by the name or name path (from the main toolbar) in the first argument.
`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.
`goToView`	Goes to one of the "xy", "xz", "yx", "yz", "zx", or "zy" views in the main graphics window or in the graphics window of a form object.
`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
`clearAllMeshes`	Clears all meshes.
`clearAllSolutions`	Clears all solutions.
`clearSelection`	Clears the selection in the given graphics object.
`environmentReflections`	Adds environment reflections to graphics
`exit`	Exits the application.
`fileOpen`	Opens a file with the associated program on the client.
`fileSaveAs`	Downloads a file to the client. See also the section "File Methods".
`printGraphics`	Prints the given graphics object.
`rotateEnvironment`	Rotate the environment used for reflections
`saveApplication`	Saves the application.
`saveApplicationAs`	Saves the application under a different name. (Or as an MPH-file.)
`scenelight`	Toggles scene light in the given graphics object.
`selectAll`	Selects all objects in the given graphics object.
`skyBox`	Shows skybox in graphics
`transparency`	Toggles transparency in the given graphics object.
`zoomExtents`	Makes the entire model visible in the given graphics object.
`zoomToSelection`	Zooms to the current selection.

Debug Methods

NAME	DESCRIPTION
`clearDebugLog`	Clears the Debug Log window.
`debugLog`	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
`external`	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 ***Application Programming Guide***.

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
`setProgressInterval`	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 `setProgress()`.
`setProgress`	Sets a value for the user-controlled progress level.
`resetProgress`	Removes all progress levels and resets progress to `0` and the message to an empty string.
`showIndeterminateProgress`	Shows a progress dialog box with an indeterminate progress bar, given `message`, and an optional cancel button.
`showProgress`	Shows a progress dialog box with an optional cancel button, optional model progress, and one or two levels of progress information.
`closeProgress`	Closes the currently shown progress dialog box.
`startProgress`	Resets the value of a given progress bar form object name to 0.
`setProgressBar`	Sets the value of a given progress bar form object name in the range 0 –100 and the associated progress message.

Date and Time Methods

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

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

Conversion Methods

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

Array Methods

NAME	DESCRIPTION
`getColumn`	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.
`getSubMatrix`	Returns a rectangular submatrix of an input matrix. Available for string, double, integer, or Boolean 2D arrays.
`insert`	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.
`getDblStringArray`	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.

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.)

Naming Conventions

In the demo applications in the Application Libraries, all forms, events, declarations, and methods use camelCase. You can adopt this convention also in your own applications. Following this convention, a name should be composed of a number of words joined without spaces, with each word's initial letter in capitals except the first letter that should be lowercase. Use a descriptive name and long names are better than hard-to-understand short names.

Examples of names for forms:

- main
- inputParameters
- geometryTab

Examples of names for events:

- updatePlot
- moveToVelocityTab

Examples of names for declarations:

- Strings — state, waveguideType
- Boolean — isError, didChange, hasBeenInitialized
- Integer — year, nextYear
- Double — speed, heatTransferCoefficient

Examples of names for methods:

- compute();
- computeStudy1();
- computeStudyAndPlot();
- getDataForPostProcessing();
- setPlotType();

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.
- 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.
- Use the Form Wizard templates to get started with creating forms.
- Consider using subwindows instead of form collections.
- 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. Note that some applications may require additional products to run.

NAME	APPLICATION LIBRARY
Cluster Setup Validation	COMSOL Multiphysics
Curve Digitizer	COMSOL Multiphysics
Helical Static Mixer	COMSOL Multiphysics
Transmission Line Calculator	COMSOL Multiphysics
Tubular Reactor	COMSOL Multiphysics
Tuning Fork	COMSOL Multiphysics
B-H Curve Checker	AC/DC Module
Induction Heating of a Billet	AC/DC Module
Effective Nonlinear Magnetic Curves	AC/DC Module
Organ Pipe Design	Acoustics Module, Pipe Flow Module
Lithium Battery Designer	Battery Design Module
Lithium Battery Pack Designer	Battery Design Module
Lithium-Ion Battery Impedance	Battery Design Module
Water Treatment Basin	CFD Module
Reaction Equilibrium - Gas Phase Conversion of Ethylene to Ethanol	Chemical Reaction Engineering Module
Cathodic Protection Designer	Corrosion Module
Cyclic Voltammetry	Electrochemistry Module
Electrochemical Impedance Spectroscopy	Electrochemistry Module
Concentric Tube Heat Exchanger	Heat Transfer Module
Equivalent Properties of Periodic Microstructures	Heat Transfer Module
Finned Pipe	Heat Transfer Module

NAME	APPLICATION LIBRARY
Forced Air Cooling with Heat Sink	Heat Transfer Module
Inline Induction Heater	Heat Transfer Module
Thermoelectric Cooler	Heat Transfer Module
Mixer	Mixer Module
Charge Exchange Cell Simulator	Molecular Flow Module, Particle Tracing Module
Truck Mounted Crane Analyzer	Multibody Dynamics Module
General Parameter Estimation	Optimization Module
Heat Recovery for System for 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
Slot-Coupled Microstrip Patch Antenna Array Synthesizer	RF Module
Rotor Bearing System Simulator	Rotordynamics Module
Si Solar Cell with Ray Optics	Semiconductor Module
Beam Section Calculator (Using LiveLink™ for Excel®)	Structural Mechanics Module, LiveLink™ for Excel®
Beam Section Calculator	Structural Mechanics Module
Bike Frame Analyzer	Structural Mechanics Module, LiveLink™ for SOLIDWORKS®
Homogenized Material Properties of Periodic Microstructures	Structural Mechanics Module
Fiber Simulator	Wave Optics Module
Plasmonic Wire Grating Analyzer	Wave Optics Module
Polarizing Beam Splitter	Wave Optics Module

The following sections highlight some of the 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

This app demonstrates the following:

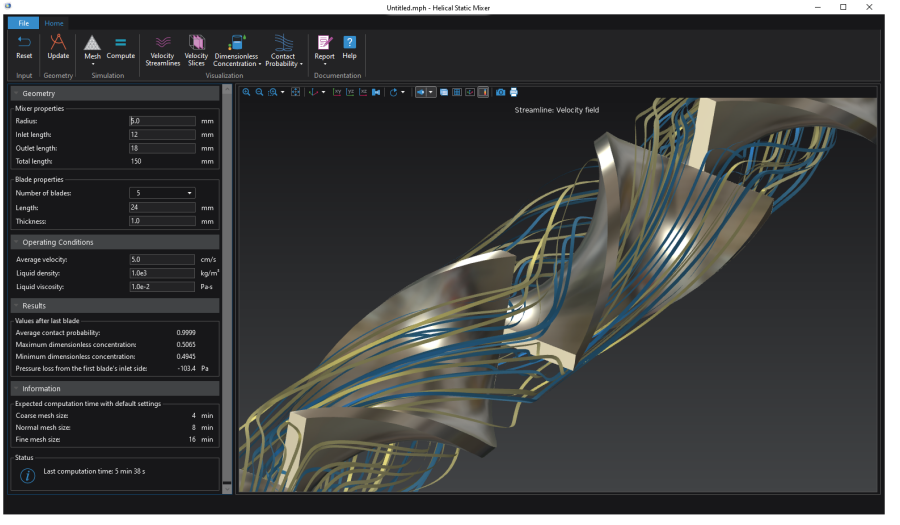
- Geometry parts and parameterized geometries

- Dark theme
- The use of subwindows
- Material appearance visualization with environment reflections
- Report generation for both Microsoft® Word and Microsoft® PowerPoint
- Options for setting different mesh sizes
- Improved graphics visualization when showing and hiding different geometry objects
- Enabling and disabling ribbon items based on the solution state.

Helical static mixers are often used to mix monomers and initiators which then react during a polymerization process. The concentration field is included in the analysis in order to compute the extent of mixing between two streams injected into the static mixer through semicircle-shaped inlets.

The app can be used to estimate the degree of mixing in a system including one to five helical blades whose dimensions can also be varied. The monomers' liquid properties and inlet velocity can also be varied.

This application does not require any add-on products.



Transmission Line Calculator

This app demonstrates the following:

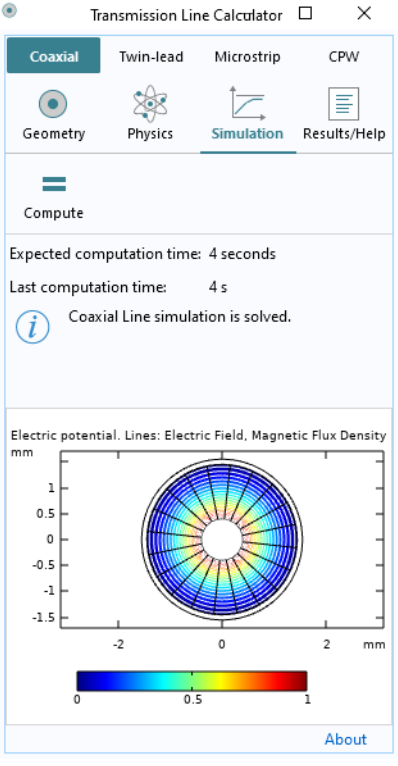
- Creating apps for small screens such as smartphones
- User-interface navigation with a top menu typically used on websites

- Dynamically hiding forms using card stacks to minimize the space required by the app
- Changing appearance by using different background colors.

Transmission line theory is a cornerstone in the teaching of RF and microwave engineering. Transmission lines are used to guide waves of electromagnetic fields at radio frequencies. They exist in a variety of forms, many of which are adapted for easy fabrication and employment in printed circuit board (PCB) designs. Often, they are used to carry information, with minimal loss and distortion, within a device and between devices.

Electromagnetic fields propagate along transmission lines as transverse electromagnetic (TEM) waves. The Transmission Line Parameter Calculator app computes resistance (R), inductance (L), conductance (G), and capacitance (C) as well as the characteristic impedance and propagation constant for some common transmission lines types: coaxial line, twin lead, microstrip line, and coplanar waveguide (CPW).

This application does not require any add-on products.



Tubular Reactor

This app demonstrates the following:

- Sending an email with a report when the computation is finished
- User-defined email server settings
- Playing a sound when the computation is completed
- Language localization
- Options to visualize plots tiled or tabbed.

The app exemplifies how students in chemical engineering can model nonideal tubular reactors (radial and axial variations) and investigate the impact of different operating conditions. It is also a great example of how teachers can build tailored interfaces for problems that challenge the students' imaginations.

The model describes a tubular reactor where propylene oxide (A) reacts with water (B) to form propylene glycol (C):

A + B -> C

Since water is the solvent and present in abundance, the reaction kinetics may be described as first order with respect to propylene oxide

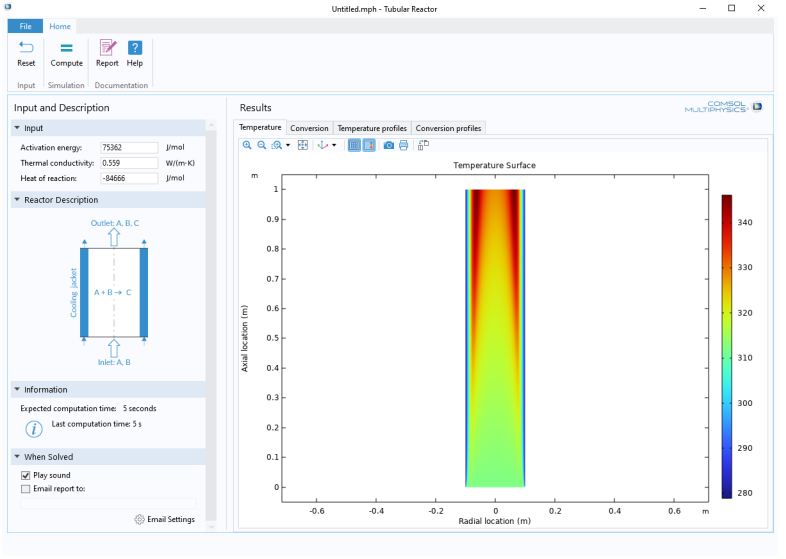
R = k*C_A

Alternatively, a second-order reaction can also be implemented according to

R = kf*C_A*C_B - kr*C_C

The reaction is exothermic and a cooling jacket is used to cool the reactor. The reactor is modeled in 2D axisymmetry and the simulation results yield composition and temperature variations in both the radial and axial directions.

This application does not require any add-on products.



Another version of this app demonstrates how the computational speed can be increased with the use of a surrogate model, as opposed to a fully-fledged finite element model. A surrogate model is a simpler, usually computationally cheaper model, which is used to approximate the behavior of a more complex, and often more computationally expensive, model. The faster model evaluation offered by the surrogate model provides the user of the app a more interactive user experience and makes it easier to spread the use of simulations in an organization. This app is called Tubular Reactor Surrogate Model Application.

Tuning Fork

This app demonstrates the following:

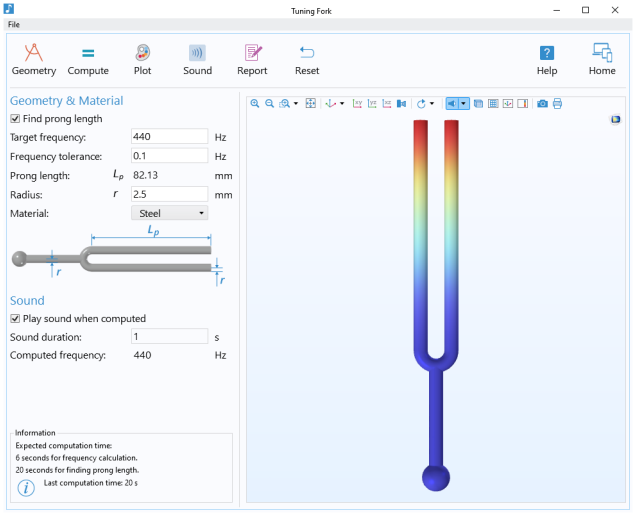
- Playing a sound at a specific computed frequency
- Selecting different materials from a combo box
- Visualizing material appearance, color, and texture
- Choice of three different user interface layouts for computer, tablet, or smartphone
- Custom implementation of the secant method
- Custom window icon.

When a tuning fork is struck, it vibrates in a complex motion pattern that can be described mathematically as the superposition of resonant modes, also known as

eigenmodes. Each mode is associated with a particular eigenfrequency. The tuning fork produces its characteristic sound from the specific timbre that is created by the combination of all of the eigenfrequencies.

The app computes the fundamental resonant frequency of a tuning fork where you can change the prong length. Alternatively, you can provide a user-defined target frequency and the application will find the corresponding prong length using an algorithm based on a secant method.

This application does not require any add-on products.



B-H Curve Checker

This app demonstrates the following:

- Importing measured data from a text file
- Handling measured data using methods
- Exporting the results to a text file.

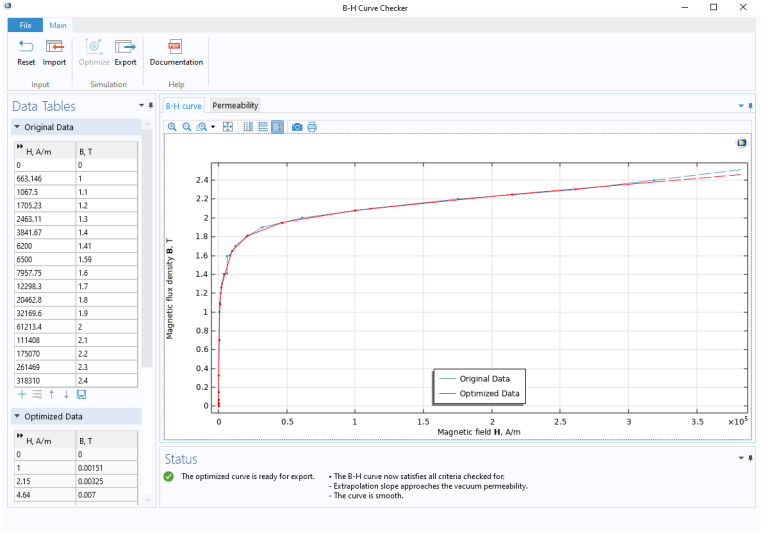
The app can be used to verify and optimize B-H curves using experimental data. It also generates curve data in the over-fluxed region, where measurement are difficult to perform. It removes the unphysical ripples of the slope of the B-H curve that might cause numerical instability.

The original B-H curve is evaluated from two aspects. Firstly, to verify that the extrapolation of the curve is reasonable from the physical point of view. Secondly, to check if the slope of the curve is smooth. The optimization algorithms are

mainly based on the simultaneous exponential extrapolation method and the linear interpolation method, respectively.

The app requires the original curve data defined in a text input file. Once the curve is imported, the application checks if optimization is required. By clicking the **Optimize Curve** button, the user can generate the optimized curve data, which can be exported to a text file.

This application does not require any add-on products.



Induction Heating of a Steel Billet

This app demonstrates the following:

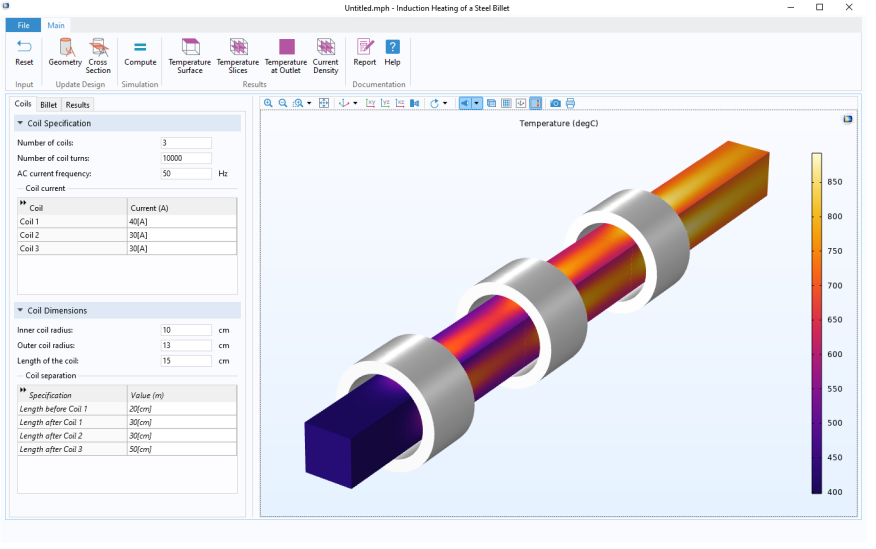
- Geometry parts and parameterized geometries
- Using tables for user input parameters
- Visualization on a 2D cross-section of a 3D geometry
- Improved visualization and user experience when a geometry object (the air object) is hidden.

Induction heating is a method used to heat metals for forging and other applications. Compared with more traditional heating methods, such as gas or electric furnaces, induction heating delivers heating power directly to the piece in a more controlled way and allows for a faster processing time.

The app is used to design a simple induction heating system for a steel billet, consisting of one or more electromagnetic coils through which the billet is moved at a constant velocity. The coils are energized with alternating currents and induce

eddy currents in the metallic billet, generating heat due to Joule heating. The billet cross section; the coil number, placement, and size; as well as the initial and ambient temperature and the individual coil currents can all be specified as inputs in the app.

This application requires the AC/DC Module.



Effective Nonlinear Magnetic Curves Calculator

This app demonstrates the following:

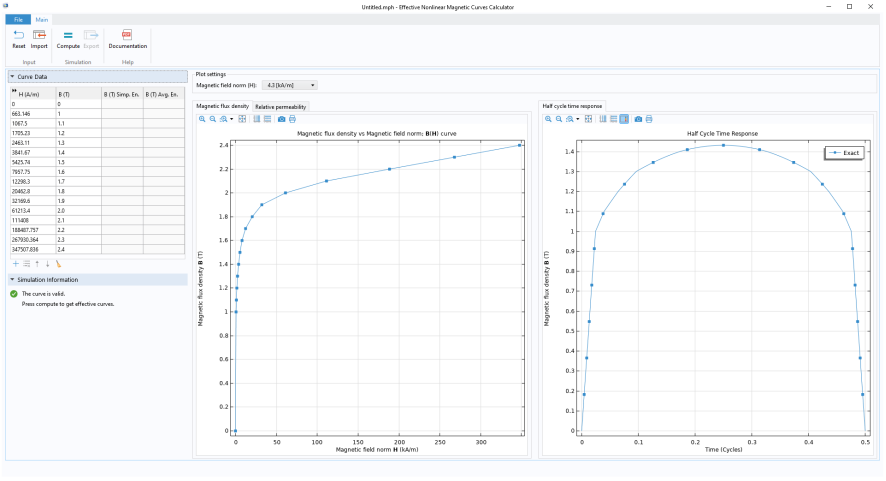
- Importing measured data from a text file
- Handling measured data using methods
- Exporting the results to a text file
- Exporting the results as COMSOL Material Library file.

The app is a companion to the Effective Nonlinear Constitutive Relations functionality. Magnetic-based interfaces in the AC/DC Module support the Effective HB/BH Curve material model that can be used to approximate the behavior of a nonlinear magnetic material in a frequency domain simulation without the additional computational cost of a full transient simulation.

The Effective HB/BH Curve material model requires the effective `Heff(B)` or `Beff(H)` relations defined as interpolation functions. This utility app can be used to compute the interpolation data starting from the material's *H(B)* or *B(H)* relations.

The interpolation data for the *H(B)* or *B(H)* relations can be imported from a text file or entered in a table. The app then computes the interpolation data for the `Heff(B)` or `Beff(H)` relations using two different energy methods. The resulting effective material properties can be exported as a COMSOL Material Library file and be further used in a model with the Magnetic Fields interface.

This application does not require any add-on products.



Organ Pipe Design

This app demonstrates the following:

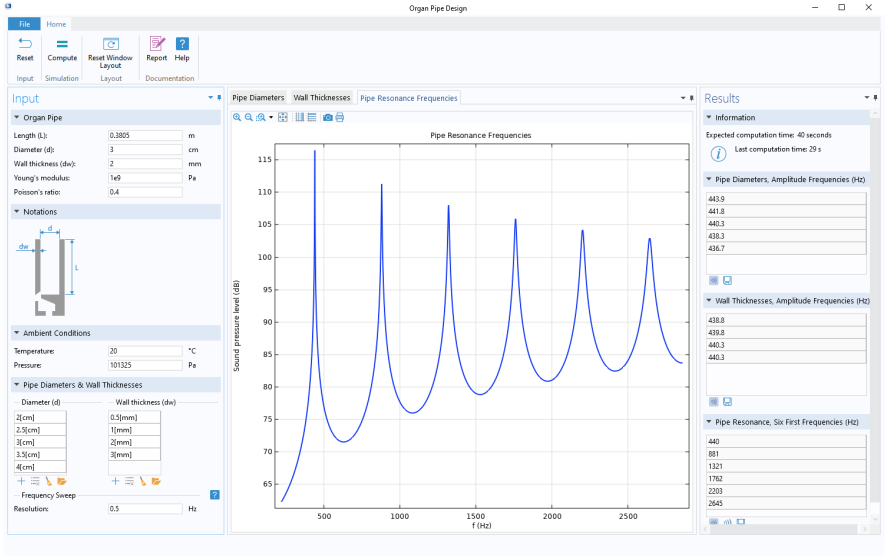
- Using a Java® utility class for combining several waveforms and for playing sound
- Using tables for presenting results.

The app allows you to study the design of an organ pipe and then play the sound and pitch of the changed design. The pipe sound includes the effects of different harmonics with different amplitudes.

The organ pipe is modeled using the Pipe Acoustics, Frequency Domain interface. The app allows you to analyze how the first fundamental resonance frequency varies with the pipe radius and wall thickness, as well as with the ambient pressure and temperature.

Using the app, you can find the full frequency response, including the fundamental frequency and the harmonics. With a method written in Java®code, the app detects the location and amplitude of all harmonics in the response, thus extending the analysis beyond the built-in functionality of the COMSOL Multiphysics user interface.

This application requires the Acoustics Module.

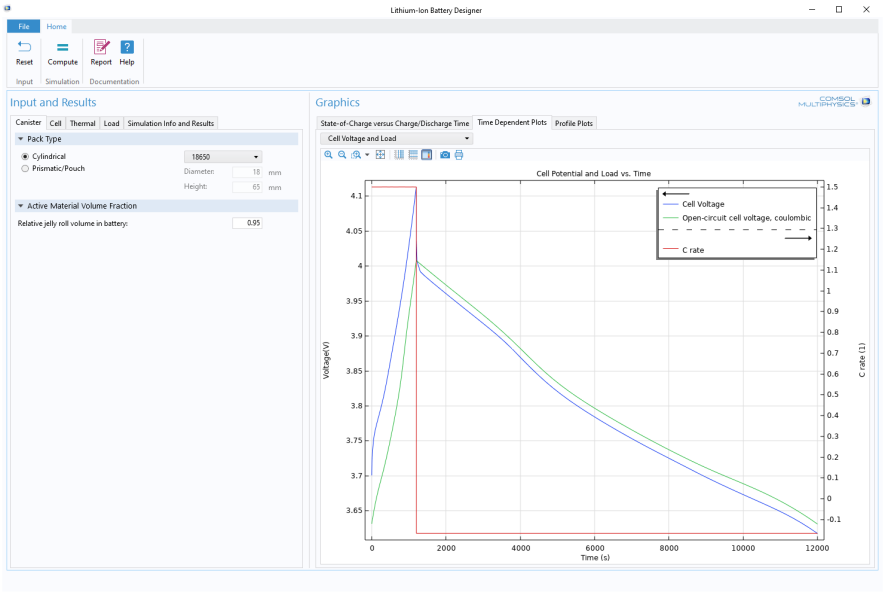


Lithium Battery Designer

This app can be used as a design tool to develop an optimized battery configuration for a specific application. The application computes the capacity, energy efficiency, heat generation, and capacity losses due to parasitic reactions of a battery for a specific load cycle.

Various battery-design parameters consist of: geometrical dimensions of the battery canister, the thicknesses of the different components (separator, current collectors and electrodes), the positive electrode material, and the volume fractions of the different phases of the porous materials can be changed. The load cycle is a charge-discharge cycle using a constant current load, which may be different for the charge and discharge stages.

The app also computes the battery temperature (assuming an uniform internal battery temperature), based on the generated heat and the thermal mass. Cooling is defined using an ambient temperature parameter and a heat transfer coefficient.



Li-Ion Battery Pack Designer

This app demonstrates the following:

- Dynamic help system using card stacks
- Multiple components (1D and 3D) in a single app
- Toggle buttons in the ribbon for showing different input, hiding/showing geometry selections, and for dynamic help
- Geometry parts and parameterized geometries
- Importing experimental data
- Options for creating different mesh sizes
- Resetting a portion of the input parameters or all
- Generating a results table during the app session
- Exporting results to a text file or to Microsoft® Excel if a license of LiveLink™ for Excel® is available
- Sliders and buttons to control the time step to plot

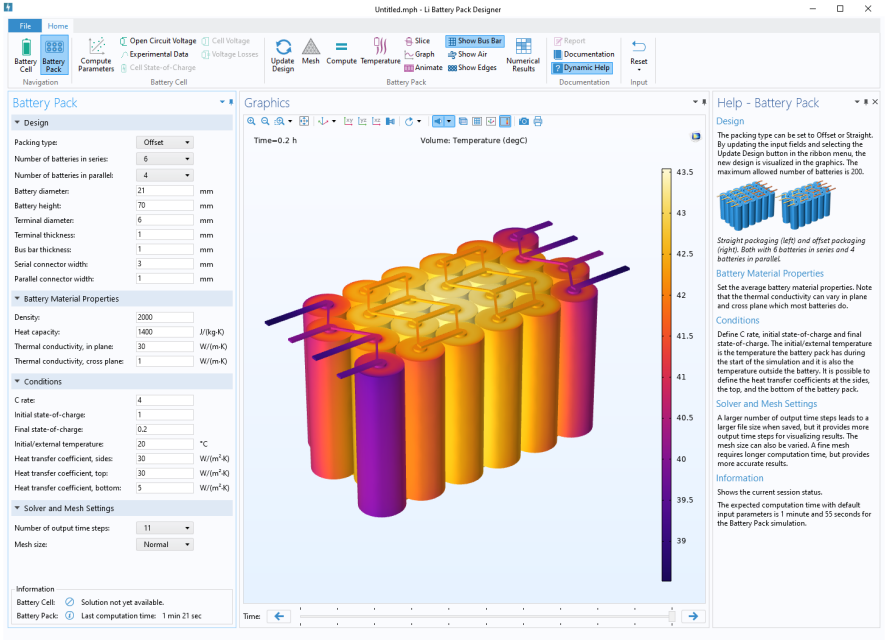
- Visualizing results with animations
- Custom window icons.

It is a tool for investigating the dynamic voltage and thermal behavior of a battery pack, using load cycle and SOC vs OCV dependence experimental data.

Parameter estimation of various parameters such as the ohmic overpotential, the diffusion time constant, and the dimensionless exchange current can be performed by the app. The app may then be used to compute a battery pack temperature profile based on the thermal mass and generated heat associated with the voltage losses of the battery.

Various battery pack design parameters (packing type, number of batteries, configuration, geometry), battery material properties, and operating conditions can be varied.

This application requires the Battery Design Module and the Optimization Module.



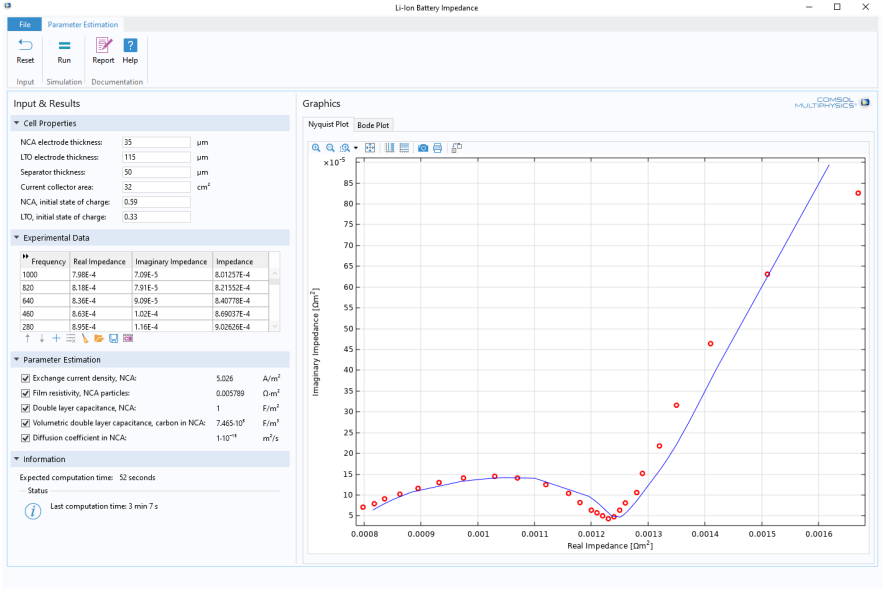
Li-Ion Battery Impedance

The goal with this app is to explain experimental electrochemical impedance spectroscopy (EIS) measurements and to show how you can use a simulation app, along with measurements, to estimate the properties of lithium-ion batteries.

The app takes measurements from an EIS experiment and uses them as inputs. It then simulates these measurements and runs a parameter estimation based on the experimental data.

The control parameters are: the exchange current density, the resistivity of the solid electrolyte interface on the particles, the double-layer capacitance of NCA, the double-layer capacitance of the carbon support in the positive electrode, and the diffusivity of the lithium ion in the positive electrode. Fitting is done to the measured impedance of the positive electrode at frequencies ranging from 10 mHz to 1 kHz.

The application requires the Optimization Module and the Battery Design Module.



Water Treatment Basin

This app demonstrates the following:

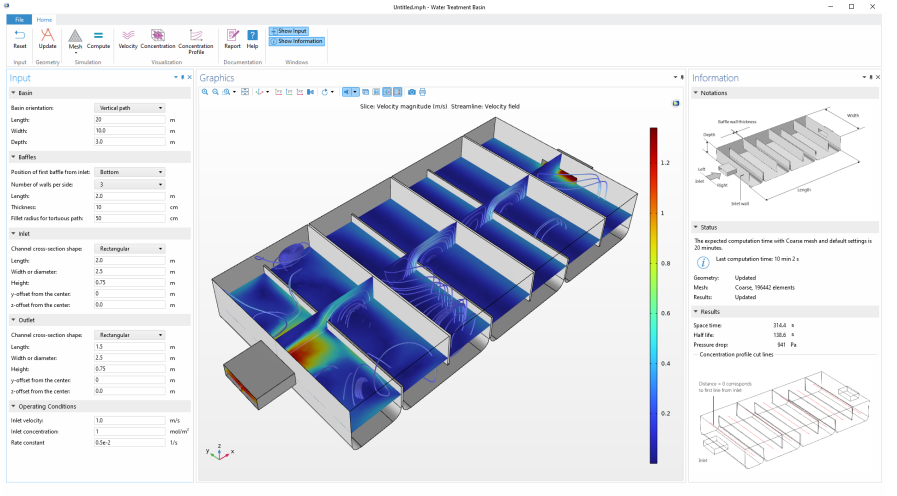
- Parameterized geometry containing a geometry sequence with if-statements to produce different types of designs
- Options to set the mesh size
- Light Theme
- A graphical user interface that includes different windows that can be shown or hidden.

Water treatment basins are used in industrial-scale processes in order to remove bacteria or other contaminants.

The app exemplifies modeling turbulent flow and material balances subject to chemical reactions. You can specify the dimensions and orientation of the basin, mixing baffles, and inlet and outlet channels. You can also set the inlet velocity, species concentration, and reaction rate constant in the first-order reaction.

The app solves for the turbulent flow through the basin and presents the resulting flow and concentration fields as well as the space-time, half-life, and pressure drop.

The application requires the CFD Module.



Reaction Equilibrium—Gas Phase Conversion of Ethylene to Ethanol

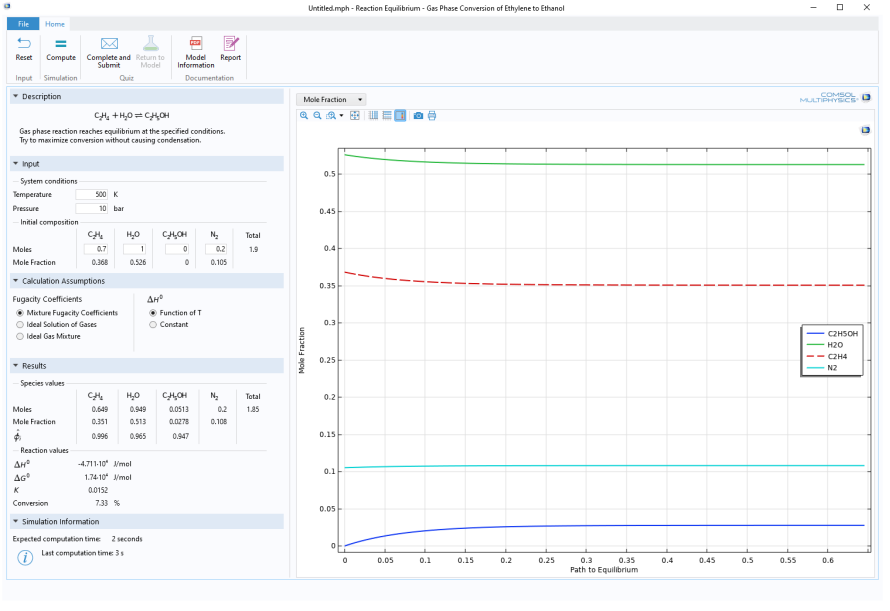
This app demonstrates the following:

- How an app can be used as a teaching tool
- An 8 question multiple choice quiz where the answers can be sent to the grader by email

This app calculates the equilibrium compositions in gas phase conversion of ethylene to ethanol. It allows you to study how the initial conditions and the operating conditions affect the ethanol production.

The app is designed to teach you how to compute quantitative results for the equilibrium composition and provide an understanding for the dynamics of a chemical equilibrium.

The application requires the Chemical Reaction Engineering Module.



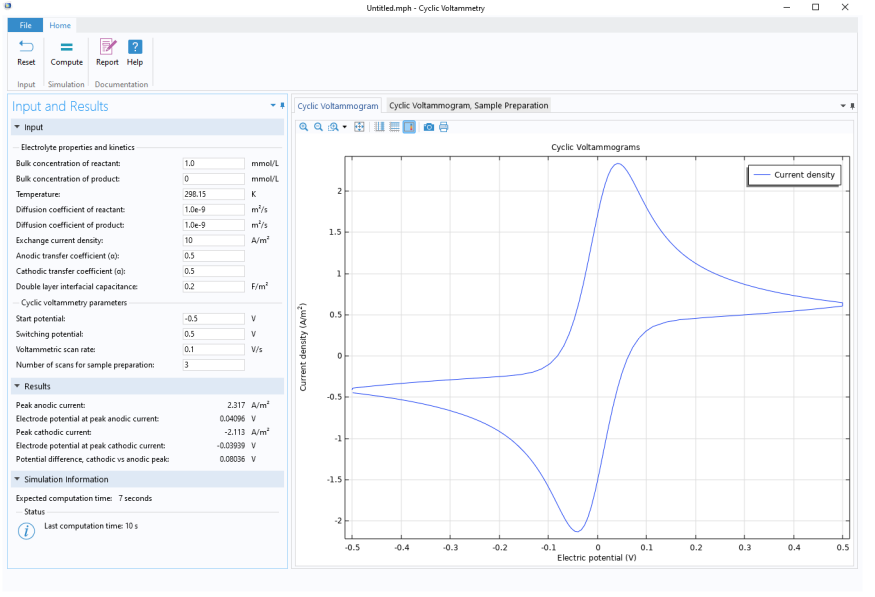
Cyclic Voltammetry

The purpose of the app is to demonstrate and simulate the use of cyclic voltammetry. You can vary the bulk concentration of both species, transport properties, kinetic parameters, as well as the cycling voltage window and scan rate.

Cyclic voltammetry is a common analytical technique for investigating electrochemical systems. In this method, the potential difference between a working electrode and a reference electrode is swept linearly in time from a start

potential to a vertex potential, and back again. The current-voltage waveform, called a voltammogram, provides information about the reactivity and mass transport properties of an electrolyte.

The application requires one of the Battery Design Module, Electrochemistry Module, Electrodeposition Module, Corrosion Module, or Fuel Cell & Electrolyzer Module.



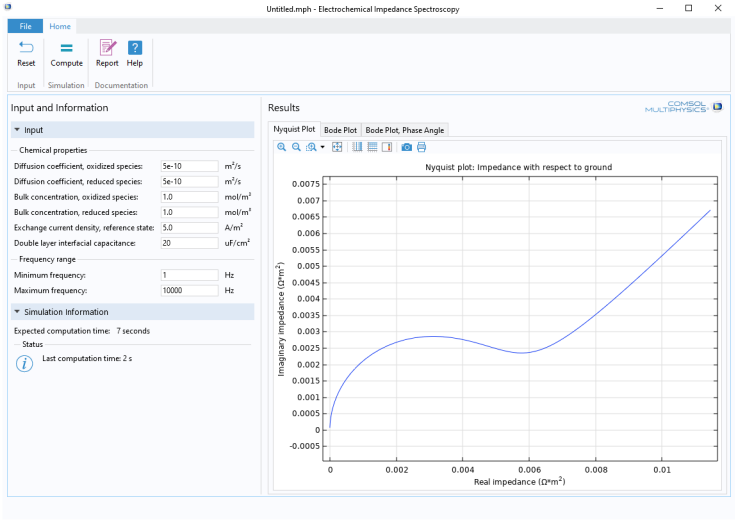
Electrochemical Impedance Spectroscopy

The purpose of this app is to understand EIS, Nyquist, and Bode plots. The app lets you vary the bulk concentration, diffusion coefficient, exchange current density, double layer capacitance, and the maximum and minimum frequency.

Electrochemical impedance spectroscopy (EIS) is a common technique in electroanalysis used to study the harmonic response of an electrochemical system. A small, sinusoidal variation is applied to the potential at the working electrode, and the resulting current is analyzed in the frequency domain.

The real and imaginary components of the impedance give information about the kinetic and mass transport properties of the cell, as well as the surface properties through the double layer capacitance.

The application requires one of the Battery Design Module, Electrochemistry Module, Electrodeposition Module, Corrosion Module, or Fuel Cell & Electrolyzer Module.



Concentric Tube Heat Exchanger

This app demonstrates the following:

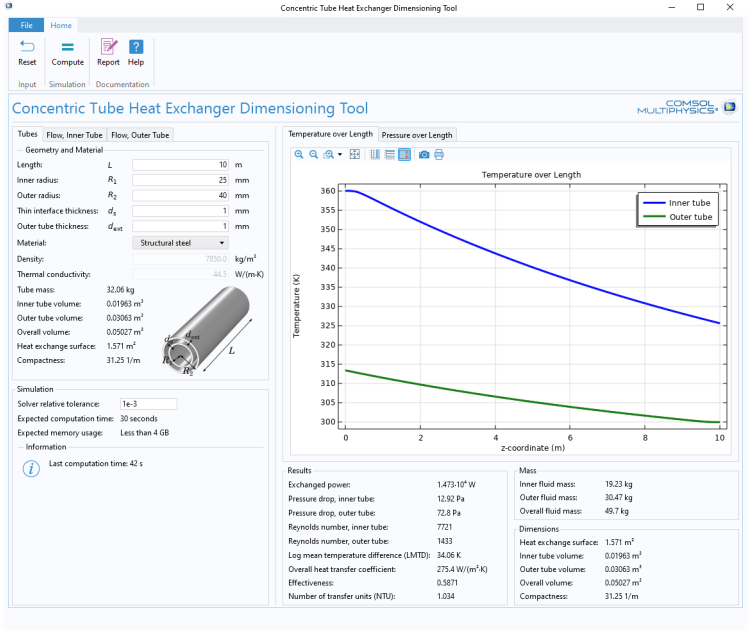
- Selecting predefined or user-defined materials
- User option to switch between laminar flow or turbulent flow
- Changing boundary conditions using methods
- Visualizing temperature dependent material properties as graph plots
- User option to set the solver tolerance.

Finding the right dimensions for a heat exchanger is imperative to ensure its effectiveness. Other properties must also be considered in order to design a heat exchanger that is both of the right size and provides heated or cooled fluid of the right temperature.

The app computes these quantities for a heat exchanger made of two concentric tubes. The fluids can flow either in parallel or in counter current flow.

The fluid properties, heat transfer characteristics, and dimensions of the heat exchanger can all be varied. The Nonisothermal Flow multiphysics interface is used to model the heat transfer.

This application requires the Heat Transfer Module.



Equivalent Properties of Periodic Microstructures

This app demonstrates the following:

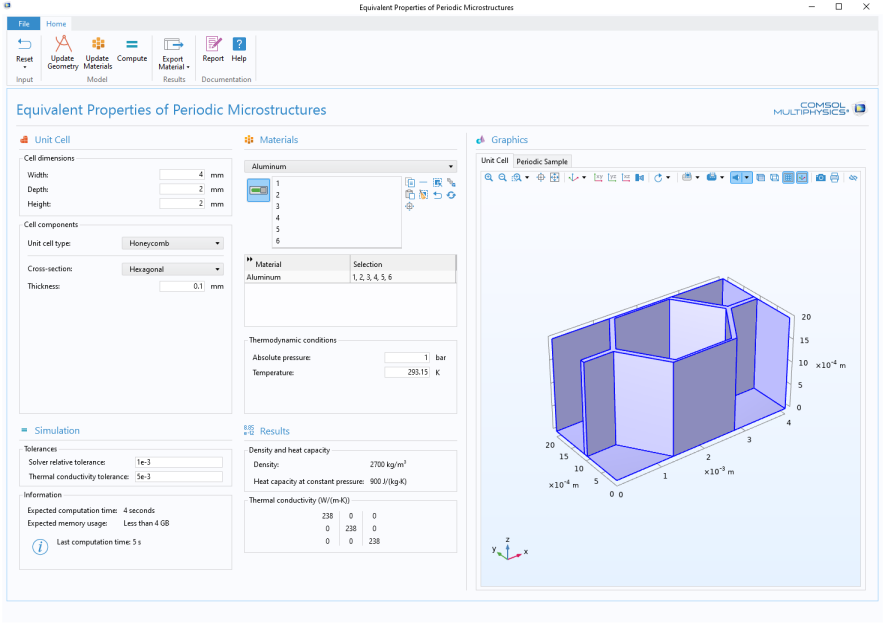
- Visualization of a periodic structure from a unit cell
- Resetting some or all input parameters
- Export the resulting material properties as an MPH-file or an XML file that can be imported to a COMSOL Multiphysics session.

Periodic microstructures are frequently found in composite materials, such as carbon fibers and honeycomb structures. They can be represented by a unit cell repeated along three directions of propagation.

To reduce computational costs, simulations may replace all of the microscopic details of a composite material with a homogeneous domain with equivalent properties. This app computes the equivalent properties for a geometrical configuration and the material properties of a unit cell to be used in a macroscopic model that uses these composite materials.

Nine different microstructures are given, with dimensional characteristics that are modifiable by the user, as well as thirteen predefined materials. The app calculates the equivalent density, heat capacity, and thermal conductivity or diffusivity of the composite materials.

This application does not require any add-on products.



Finned Pipe

This app demonstrates the following:

- Geometry parts and parameterized geometry
- A results table form object containing outputs.

Finned pipes are used for coolers, heaters, or heat exchangers to increase heat transfer. They come in different sizes and designs depending on the application and requirements.

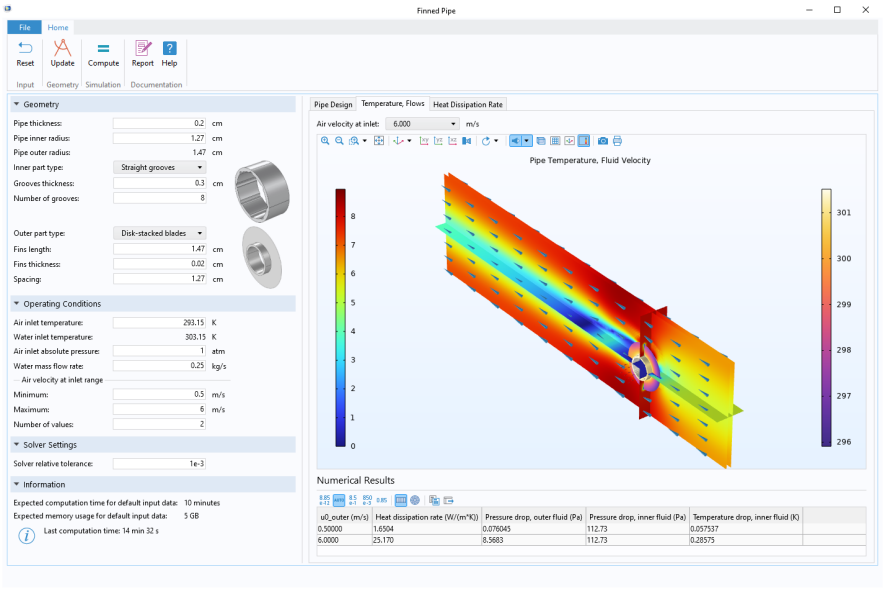
When the fins are placed outside the pipe, they increase the heat exchange surface of the pipe so that a cooling or heating external fluid can exchange heat more efficiently. When placed inside the pipe, it is the inner fluid that benefits from an increased heat exchange surface. Instead of fins, grooves can also increase the heat exchange surface, particularly inside the pipe where space is limited.

With this app, you can customize a long cylindrical pipe with predefined inner and outer fins or grooves to observe and evaluate their cooling effects. The app calculates the thermal performance of a pipe that is filled with water and then cooled or heated by surrounding air with forced convection.

Various geometric configurations are available for the outer structure (disk-stacked blades, circular grooves, helical blades, helical grooves, or none) and for the inner structure (straight grooves or none).

The app computes the dissipated power and the pressure drop as functions of the geometry and air velocity.

This application requires the Heat Transfer Module



Forced Air Cooling with Heat Sink

This app demonstrates the following:

- Geometry parts and parameterized geometries
- Sending an email with a report when the computation is finished
- User-defined email server settings which is useful when running compiled standalone applications
- Options for setting different mesh sizes
- Error control of input parameters using methods.

Heat sinks are usually benchmarked with respect to their ability to dissipate heat for a given fan curve. One possible way to carry out this type of experiment is to place the heat sink in a rectangular channel with insulated walls.

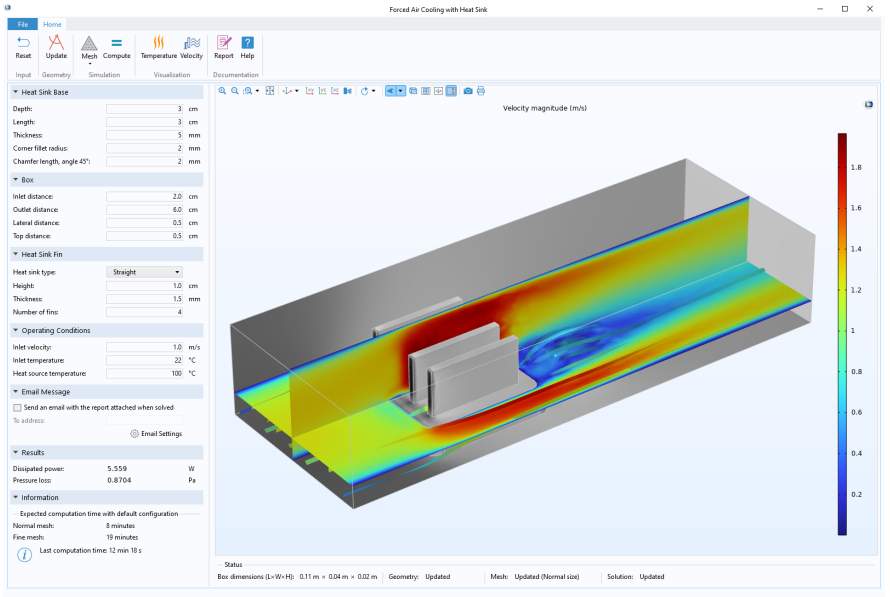
The temperature and pressure at the channel's inlet and outlet, as well as the power required to keep the heat sink base at a given temperature, is then measured. Under these conditions, it is possible to estimate the amount of heat dissipated by the heat sink and the pressure loss over the channel.

The purpose of the app is to carry out investigations of such benchmarking experiments. You can vary the type of heat sink as well as the number of fins or

pins and their dimensions to find the optimal design for a given pressure loss over the channel.

Air velocities and heat source rates can be varied and the app solves for nonisothermal flow, assuming turbulence as described by the algebraic yPlus model.

This application requires the Heat Transfer Module.



Inline Induction Heater

This app demonstrates the following:

- A model using symmetry while the results are visualized in full 3D
- Provides info if the results are above or below certain critical values
- Selecting predefined or user-defined materials
- Error control of geometry parameters using methods and presentation of possible errors using card stacks
- Sliders and buttons to control the position of the slice when visualizing the results with a slice plot.

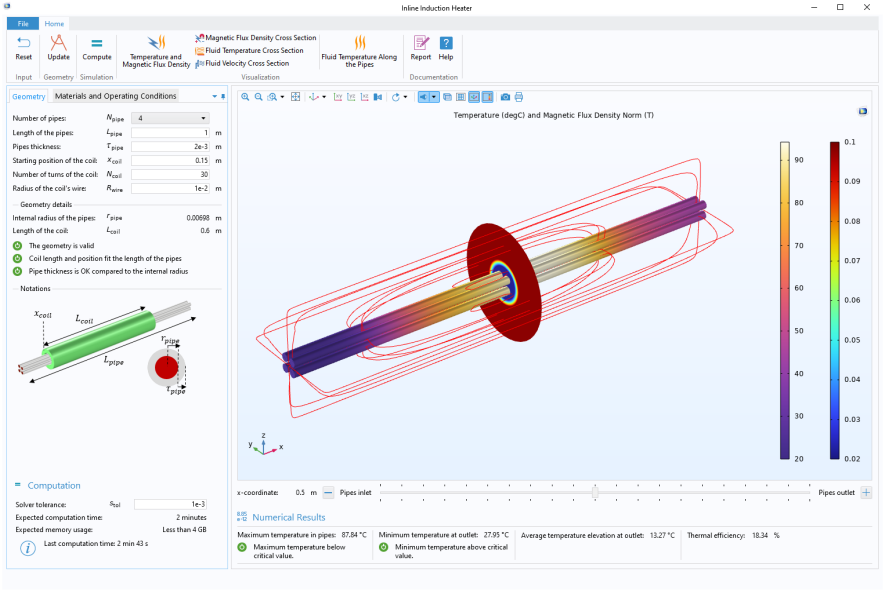
The app computes the efficiency of a magnetic induction apparatus for the heating of liquid food flowing in a set of ferritic stainless steel pipes.

Ferritic stainless steels become more and more used in food processing due to their relatively low and stable price, and their magnetic properties that allow using new heating techniques.

A circular electromagnetic coil is wound around a set of pipes in which a fluid flows. The alternating current passing through the coil generates an alternating magnetic field that penetrates the pipes, generates eddy currents inside them, and heats them up. Then heat is transferred to the fluid essentially by conduction.

Various configurations are available for the set of pipes (number, length, thickness, material) and for the coil (number of turns, wire radius, current density, and excitation frequency) to optimize the heat exchange with the fluid, while ensuring homogeneous temperatures within it for a given flow rate.

This application requires the Heat Transfer Module and the AC/DC Module.



Thermoelectric Cooler

This app demonstrates the following:

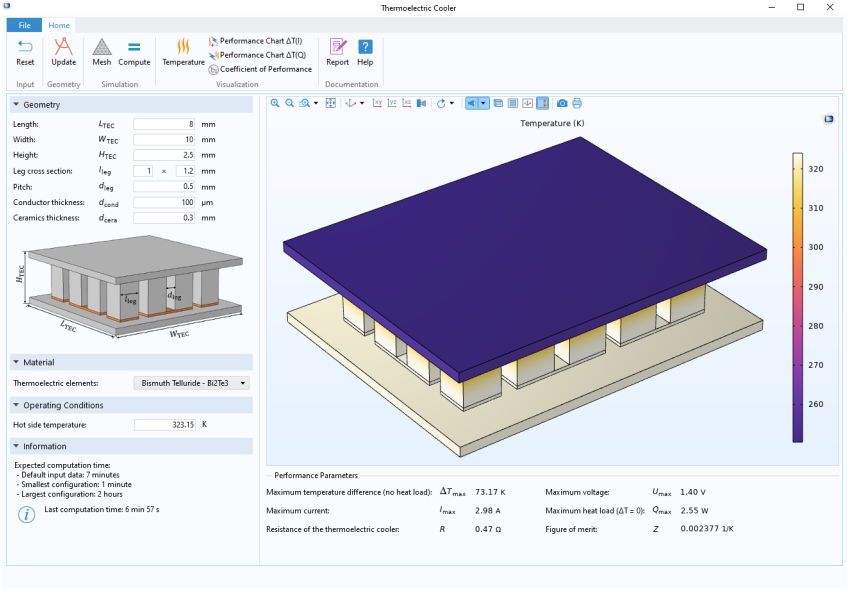
- Visualizing material appearance, color, and texture
- Showing info below the graphics about geometry parameters, results, and performance depending on the selected plot action

Thermoelectric coolers are widely used for electronics cooling in various application areas, ranging from consumer products to spacecraft design. A

thermoelectric module is a common type of component used in thermoelectricity applications. A typical module consists of several thermoelectric legs sandwiched between two thermally conductive plates, one cold and one hot. The device that needs to be cooled down must be attached to the cold face.

Due to the variety of applications, there can be many different thermoelectric cooler configurations. This app covers the basic design of a single-stage thermoelectric cooler of different sizes with different thermocouple sizes and distributions. It also serves as a starting point for more detailed calculations with additional input options and can be extended to multistage thermoelectric coolers.

This application requires the Heat Transfer Module, AC/DC Module, and Optimization Module. Instead of the AC/DC Module you could alternatively use the MEMS Module or the Plasma Module.



Mixer

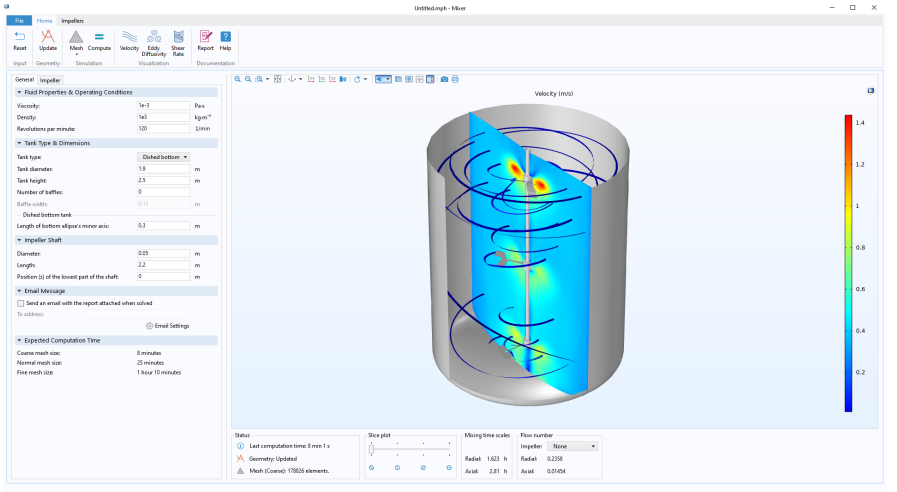
This app demonstrates the following:

- Multiple tabs in the ribbon
- Geometry parts and parameterized geometries
- Parts and cumulative selections can be used to automatically set domain and boundary settings in the embedded model
- Adding or removing geometry parts with different geometrical configuration
- Options for creating different mesh sizes
- Sending an email with a report when the computation is finished
- User-defined email server settings which is useful when running compiled standalone applications
- Sliders to control the visualization of a slice plot.

The app provides a user-friendly interface where scientists and process engineers can investigate the influence that vessels, impellers, baffles, and operating conditions have on the mixing efficiency and on the power that is required to drive the impellers. You can use this application to understand and optimize the design and operation of a mixer for a given fluid.

You can specify the dimensions of the vessel from a list of three types and the dimensions and configuration of the impellers from a list of eleven types. The vessels can also be equipped with baffles. You can further specify the impeller speed and the properties of the fluid that is being mixed.

The application requires either the CFD Module or the Polymer Flow Module.



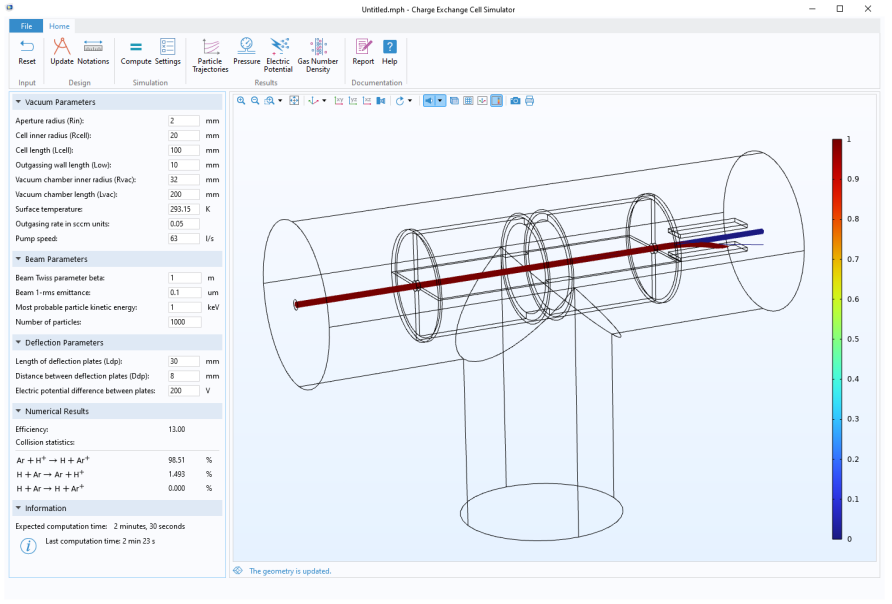
Charge Exchange Cell Simulator

A charge exchange cell consists of a region of gas at an elevated pressure within a vacuum chamber. When an ion beam interacts with the higher-density gas, the ions undergo charge exchange reactions with the gas which then create energetic neutral particles. It is likely that only a fraction of the beam ions will undergo charge exchange reactions. Therefore, in order to neutralize the beam, a pair of charged deflecting plates are positioned outside the cell. In this way, an energetic neutral source can be produced.

This app simulates the interaction of a proton beam with a charge exchange cell containing neutral argon. User input includes several geometric parameters for the gas cell and vacuum chamber, beam properties, and the properties of the charged plates that are used to deflect the remaining ions.

The simulation app computes the efficiency of the charge exchange cell, measured as the fraction of ions that are neutralized, and records statistics about the different types of collisions that occur.

This application requires the Particle Tracing Module and the Molecular Flow Module.



Truck Mounted Crane Analyzer

This app demonstrates the following:

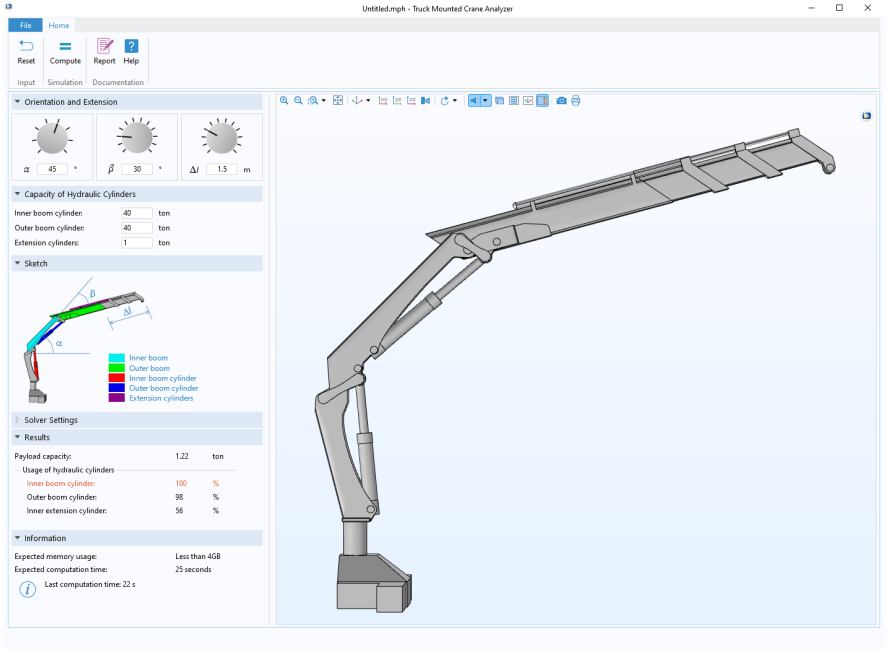
- Using the knob form object
- Updating the geometry by rotating a knob
- Provides info if the results are above or below certain critical values

Many trucks are equipped with cranes for handling loads and such cranes have a number of hydraulic cylinders that control the motion of the crane. These cylinders and other components that make up the crane are subjected to large forces when handling heavy loads. In order to determine the load-carrying capacity of the crane, these forces must be computed.

In the app, 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.

Inputs include the angle between the booms, the total extension length, the capacity of the inner and outer boom cylinders, and the capacity extension cylinders. Results from the app include the payload capacity and hydraulic cylinder usage.

The application requires the Multibody Dynamics Module.



General Parameter Estimation

This app demonstrates the following:

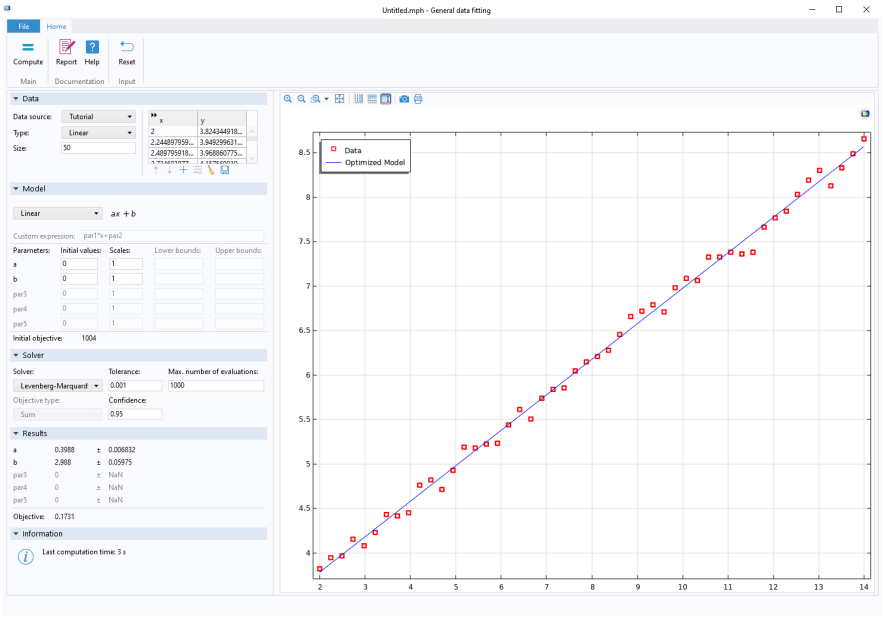
- Importing measured data from a text file or use built-in functionality for data generation
- Automatically change solver options based on the input
- Dynamically update the equation display.

The app can be used to estimate parameters in models without any physics. Data can be imported from a file or the built-in functionality for data generation can be utilized.

The models include linear, quadratic, sigmoid, sloped Gaussian, and a custom model with up to 5 parameters.

The Levenberg–Marquardt solver computes confidence intervals for the estimated parameters, while the other solvers (MMA, SNOPT, and BOBYQA) allow for specification of parameter bounds. MMA and BOBYQA allow for minimization of the maximum square instead of the sum.

The application requires the Optimization Module.



Geothermal Heat Pump

This app demonstrates the following:

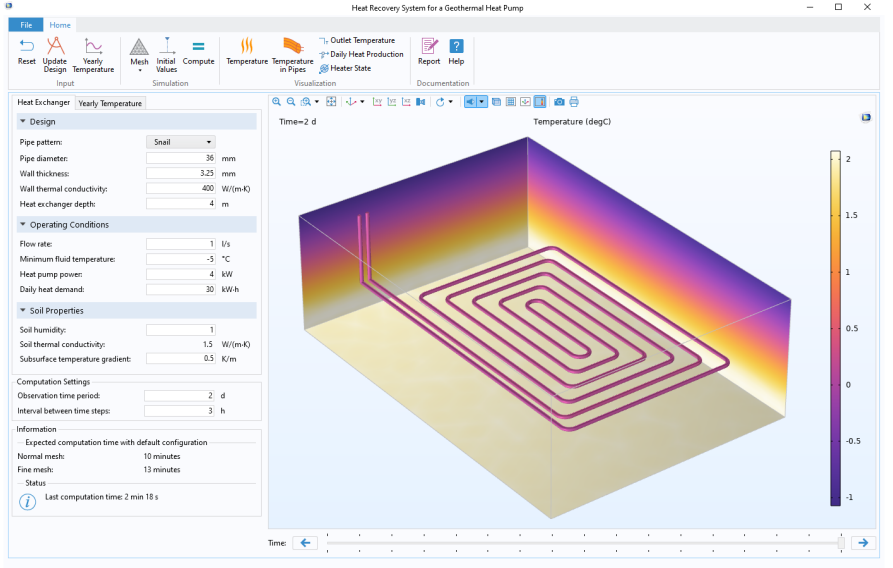
- Changing the design by using a combo box with predefined options
- Options for creating different mesh sizes
- Editing and plotting monthly data input
- Setting the end time and the time steps size of a time dependent simulation
- Visualizing the initial values for a time dependent simulation
- Includes a simple control system to manage the temperature.

Geothermal heating is an environmentally friendly and energy-efficient method to supply modern and well insulated houses with heat. Heat exchangers placed at a sufficient depth in the ground below the house utilize subsurface heat, where temperatures are almost constant throughout the year.

The app studies different pipe configurations of a ground heat exchanger. It provides information on the performance of ground-coupled heat exchangers for different specifications (depth, pattern, pipes configuration, and heating conditions), temperature conditions, soil thermal conductivity, and temperature gradient.

The heater can also be turned off if the daily heat demand is achieved, and then turned on again after 24 hours. The temperature at the pipe's outlet can be controlled and compared to the minimum temperature required in the heat exchanger specifications.

This application requires the Pipe Flow Module.



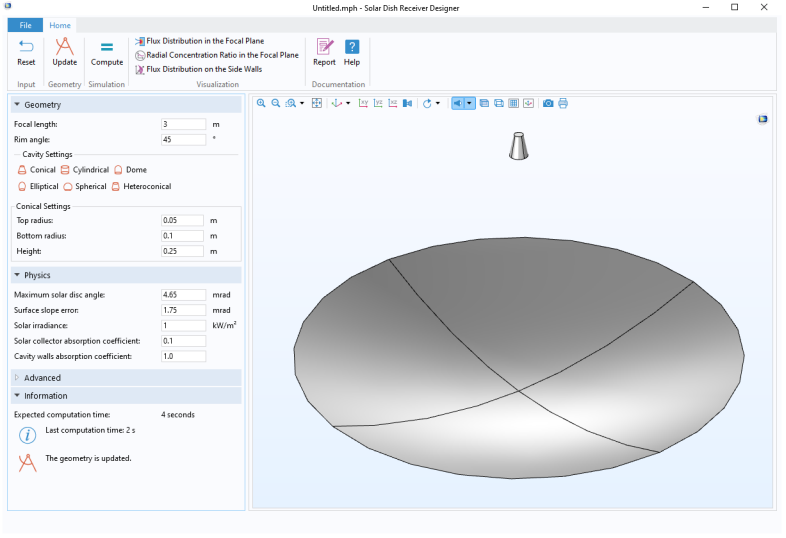
Solar Dish Receiver Designer

Solar concentrator/cavity receiver systems can be used to focus incident solar radiation into a small region, generating intense heat which can then be converted to electrical or chemical energy. A common figure of merit in solar thermal power systems is the concentration ratio, or the ratio of the solar flux on the surface of the receiver or in the focal plane to the ambient solar flux.

This app is an application based on the Solar Dish Receiver tutorial model. In this app, incident solar radiation is reflected by a parabolic dish, while the concentrated solar radiation is collected in a small cavity. A total of six different parameterized cavity geometries are available for investigation: Cylindrical, Dome, Heteroconical, Elliptical, Spherical, and Conical. It is also possible to take several different types of perturbation into account, including solar limb darkening and surface roughness. For each cavity geometry, built-in plots show the flux distribution and concentration ratio in the focal plane as well as the incident flux on the interior surfaces of the cavity.

You can learn more about this example in a related blog post: “Efficiently Optimizing Solar Dish Receiver Designs”: https://www.comsol.com/blogs/efficiently-optimizing-solar-dish-receiver-designs/.

This application requires the Ray Optics Module.



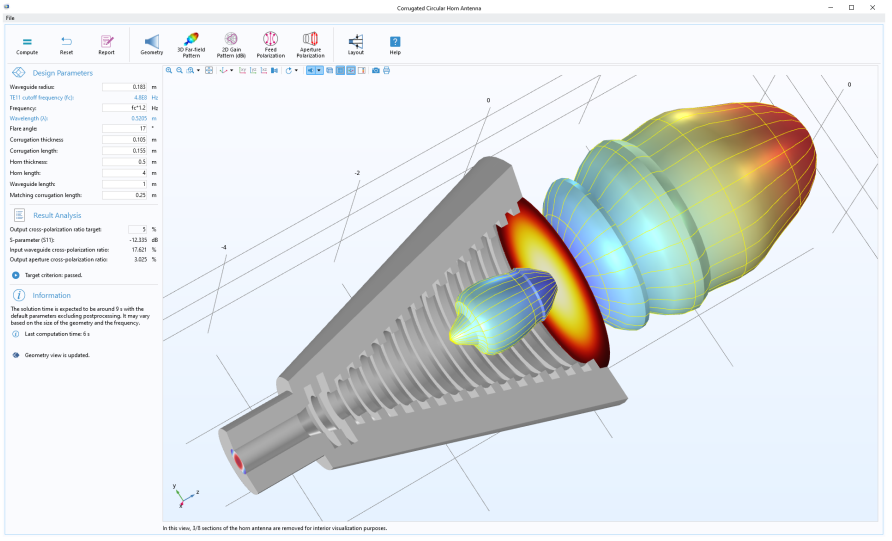
Corrugated Circular Horn Antenna

This app demonstrates the following:

- A toolbar with large buttons for the navigation instead of a ribbon
- Subforms used as sections and the sections' headings include an image
- Provides info if the results are within a certain range
- Visualizes a 2D axisymmetric model in full 3D

The excited TE mode from a circular waveguide passes along the corrugated inner surface of a circular horn antenna where a TM mode is also generated. When combined, these two modes give lower cross-polarization at the antenna aperture. By using this app, the antenna radiation characteristics, as well as aperture cross-polarization ratio can be improved by modifying the geometry of the antenna.

This application requires the RF Module.



Frequency Selective Surface Simulator

This app demonstrates the following:

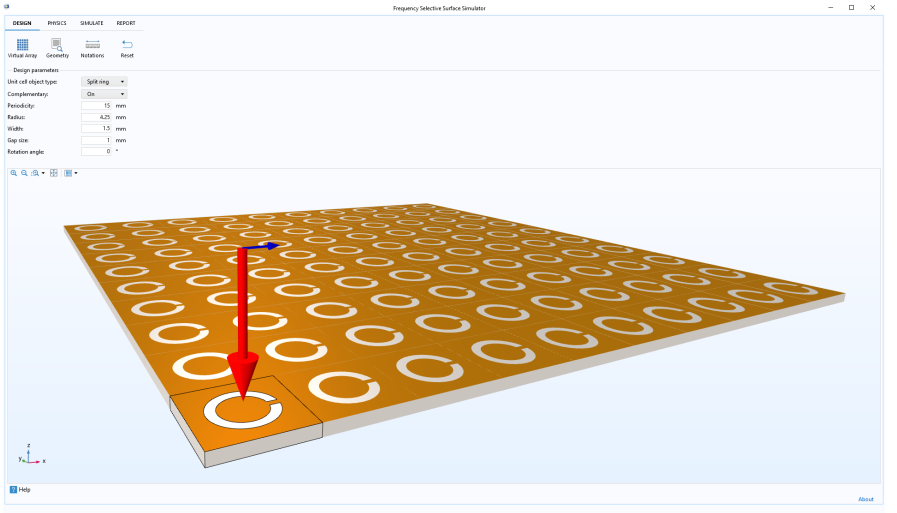
- Designing an app for small screens such as smartphones
- User-interface navigation with a top menu typically used on websites
- Geometry parts and parameterized geometries
- Visualizing periodicity of a geometry with material rendering
- Warning messages on icons when properties are not updated
- Sending an email with a report attached when the computation is finished

Frequency selective surfaces (FSS) are periodic structures that generate a bandpass or a bandstop frequency response. They are used to filter or block RF, microwave, or, in fact, any electromagnetic wave frequency. For example, you see these selective surfaces on the doors of microwave ovens, which allow you to view the food being heated without being heated yourself in the process.

The app simulates a user-specified periodic structure chosen from the built-in unit cell types. It provides five unit cell types popularly used in FSS simulations along with two predefined polarizations in one fixed direction of propagation that has normal incidence on the FSS. 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.

You can change the polarization, center frequency, bandwidth, number of frequencies, substrate thickness and its material properties, and unit cell type (circle, ring, split ring, etc.) as well as their geometry parameters, including periodicity (cell size).

This application requires the RF Module.



Microstrip Patch Antenna Array Synthesizer

This app demonstrates the following:

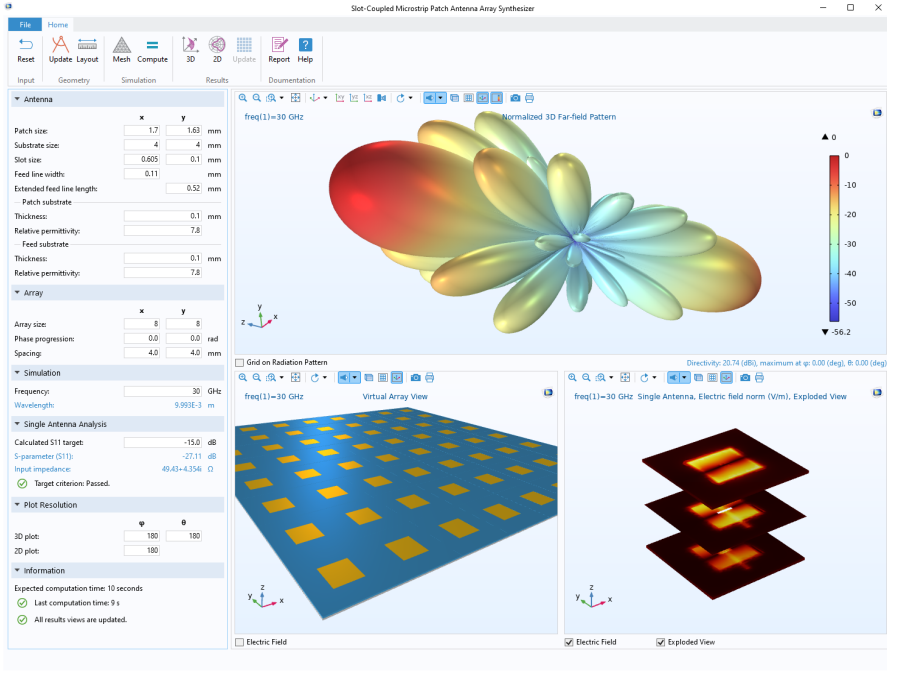
- Parameterized geometries
- Visualizing material appearance, color, and texture
- Multiple plots in the same window to visualize the results
- Options to visualize the results with different views using check boxes

Microstrip patch antenna arrays are used in a number of industries as transceivers of radar and RF signals. This is a prime candidate for the 5G mobile network system.

The app simulates a single slot-coupled microstrip patch antenna, fabricated on a multilayered low-temperature cofired ceramic (LTCC) substrate. When using this app, you will be able to simulate 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.

You can also evaluate phased antenna array prototypes for 5G mobile networks with a default input frequency of 30 GHz. You can do this by varying antenna properties such as the geometric dimension and substrate material.

This application requires the RF Module.



Rotor Bearing System Simulator

This app demonstrates the following:

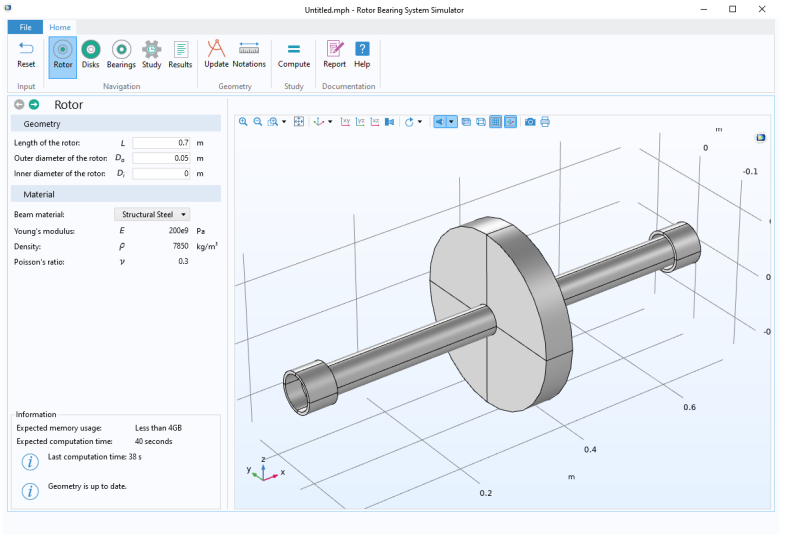
- Navigation system using toggle buttons in the ribbon and Back/Forward buttons in the settings window
- Selecting predefined or user-defined materials
- Using a table for input of geometry objects

The app simulates a rotor bearing system consisting of disks and bearings mounted on a rotating shaft. An eigenfrequency analysis is performed for a range of angular speeds, to identify critical speeds of the system.

An app of this kind is useful at an early design stage where design modifications can be made to move critical speeds away from the operating speed of the system.

Results include whirl modes, a Campbell plot, and a list of critical speeds.

This application requires the Structural Mechanics Module and the Rotordynamics Module.



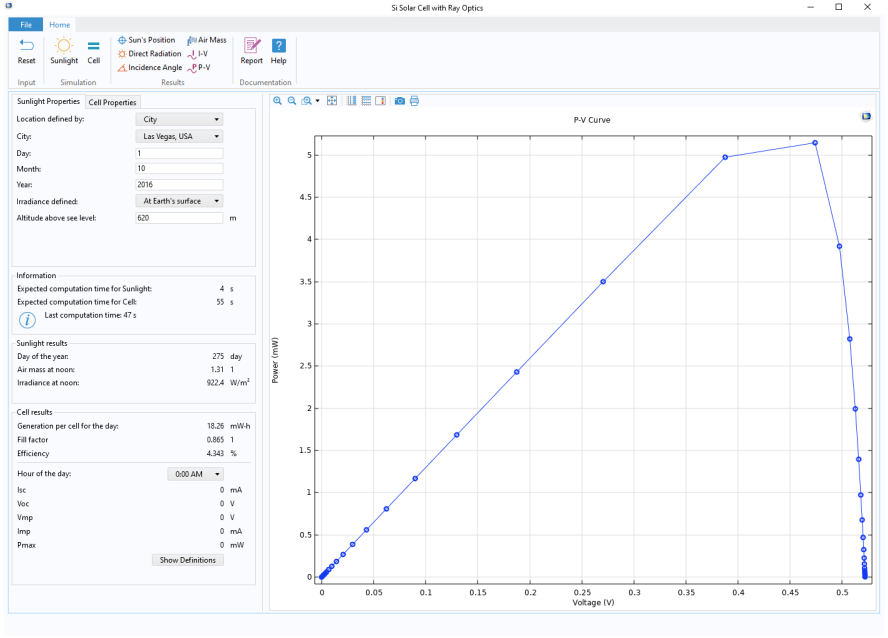
Si Solar Cell with Ray Optics

This app demonstrates the following:

- Multiple components (1D and 3D) in a single app
- Using the same choice list in the app as in the model using Data Access functionality
- Output numerical results for a specific time step using a combo box

The app combines the Ray Optics Module and the Semiconductor Module to illustrate the operation of a silicon solar cell at a location specified by the user. The Ray Optics Module computes the average illumination over a day of the year. The Semiconductor Module computes the normalized output characteristics of a solar cell with design parameters specified by the user. The normalized output characteristics is then multiplied by the computed average illumination to obtain

the output characteristics of the cell at the specified date and location, assuming simple linear relationship between the output and the illumination.



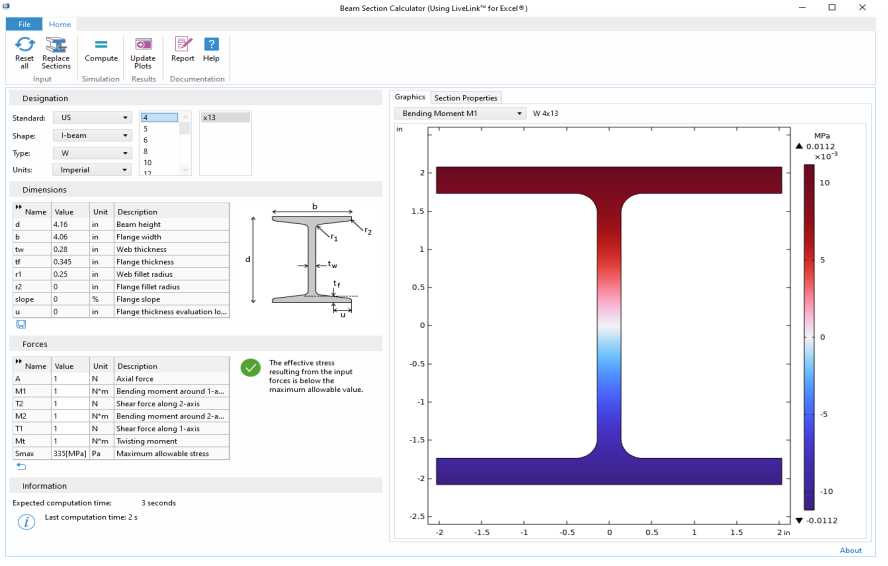
Beam Section Calculator

This app demonstrates the following:

- Reading and importing data from an Microsoft® Excel file
- Exporting data to an Microsoft® Excel file

The app computes the beam section properties and true stress distribution in a designated steel beam section. A broad range of American and European beam standards are available. It uses LiveLink™ for Excel® to read and store the beam data in Excel® worksheets.

This application requires the Structural Mechanics Module and LiveLink™ for Excel®. A version of this app is also provided without Microsoft® Excel functionality.



Bike Frame Analyzer

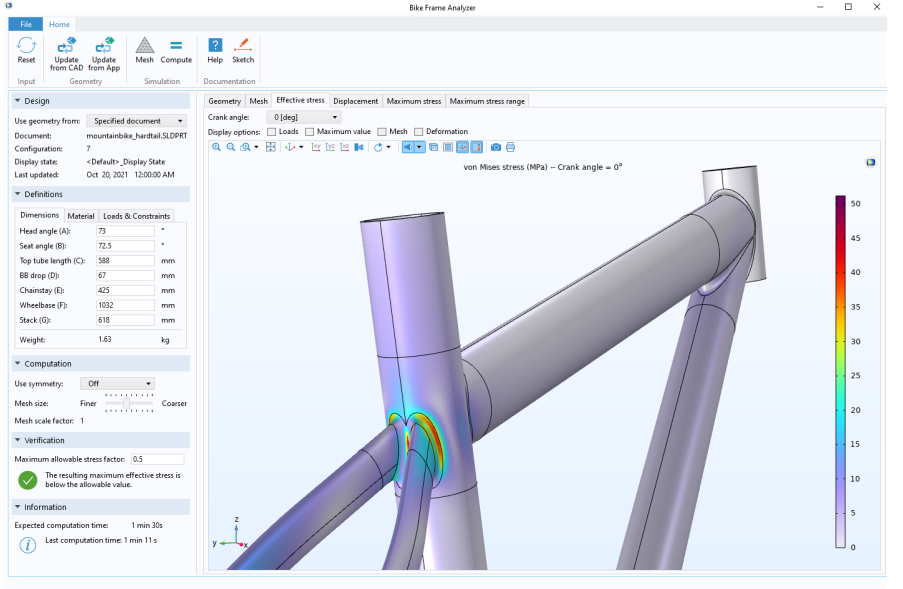
This app demonstrates the following:

- Connecting an app to a SOLIDWORKS® session
- Setting a maximum allowed value which the solution is compared to
- Selecting predefined or user-defined materials
- Changing boundary conditions with a combo box using methods

The app computes the stress distribution and the deformation of a bike frame based on user configurable loads and constraints. It leverages LiveLink™ for

SOLIDWORKS® to load the geometry, and to update the frame dimensions for studying their effect on the results.

This application requires the Structural Mechanics Module and LiveLink™ for SOLIDWORKS®.

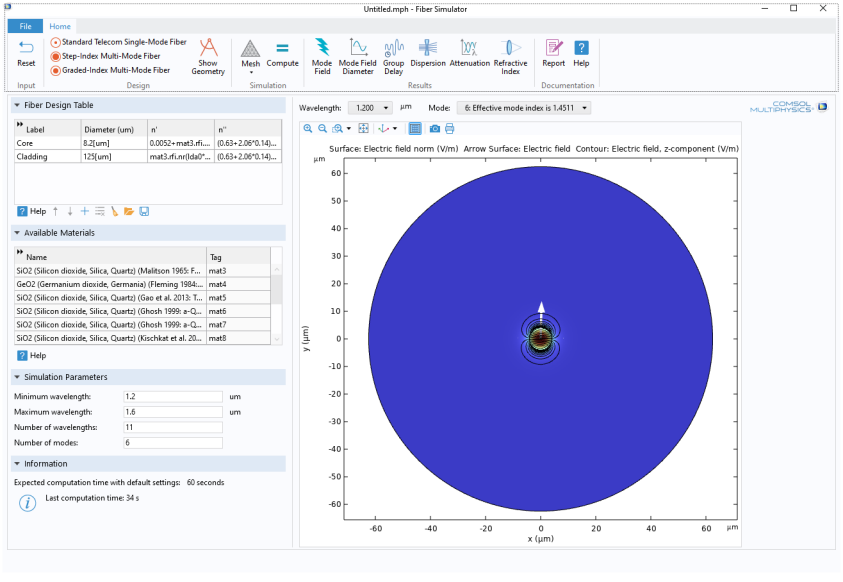


Fiber Simulator

For almost all commercial optical fiber types, the design consists of a concentric layer structure with the inner layer(s) forming the core and the outer layer(s) forming the cladding. Since the core has a higher refractive index than the cladding, guided modes can propagate along the fiber.

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.

This application requires the Wave Optics Module.



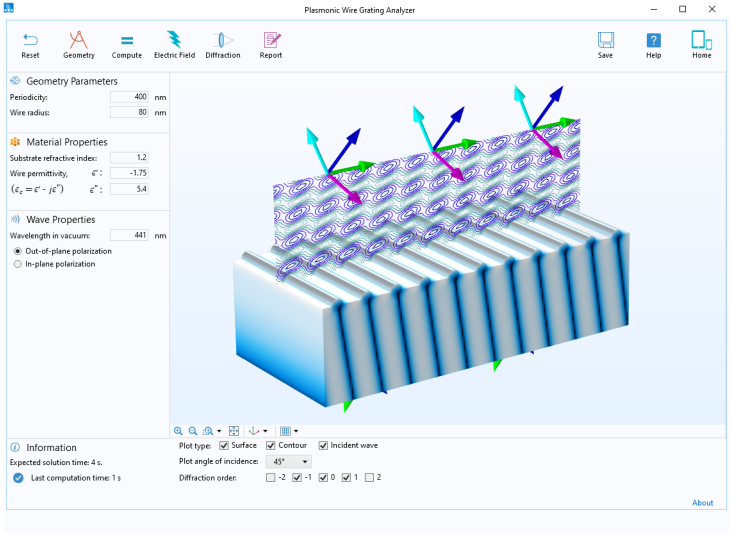
Plasmonic Wire Grating

This app demonstrates the following:

- Choice of different user interface layouts for computer/tablet or smartphone
- Custom background image and color
- Graphics appearance with custom top color and bottom color
- Custom position of the graphics toolbar

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.

This application requires the Wave Optics Module.



Polarizing Beam Splitter

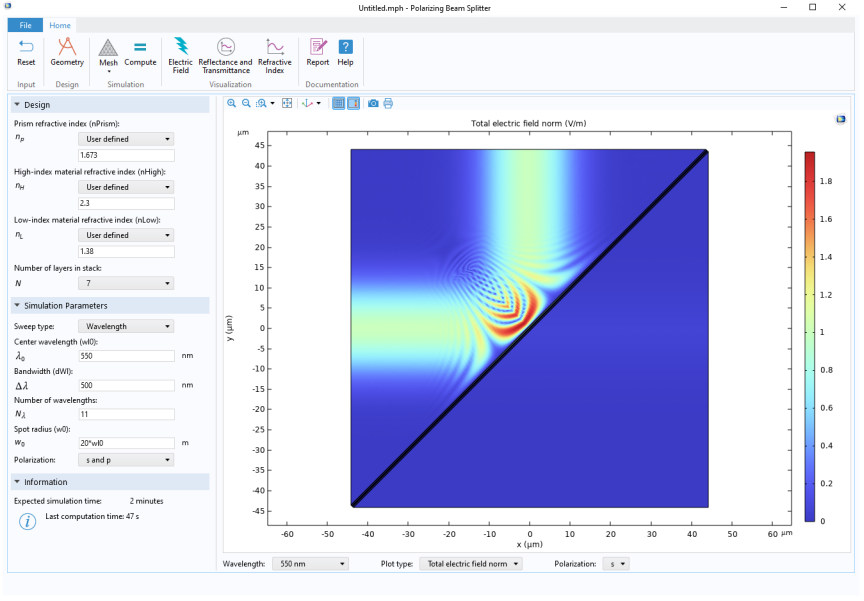
A Gaussian beam is incident on a 45-degree thin-film stack embedded in glass material prisms. The thin-film stack is designed from alternating high and low refractive index materials. The wave will be refracted at the Brewster angle at each internal interface. Thus, mainly p-polarized waves (polarization in the plane of incidence) will be transmitted, whereas mainly s-polarized waves (polarization

orthogonal to the plane of incidence) will be reflected. Changing the spot radius for the Gaussian beam modifies the polarization discrimination.

The reflectance and transmittance spectra are calculated for different Gaussian beam spot radii.

The app automatically calculates the phase expressions necessary for the Electromagnetic Waves, Beam Envelopes interface, when the user changes the design parameters.

This application requires the Wave Optics Module.



1D array *167*
2D array *168*
3D coordinates *299*

A
About dialog box *43*
About to shutdown event *151*
action *65, 69, 150*
activation condition *171, 253, 260*
active card selector *287*
Add New Choice List *171*
Add New Form Choice List *171*
add-in *14, 133, 225, 226*
  selections *95*
Add-in Libraries *230*
add-on products *10, 374*
alert *218, 359*
aligning form objects *115, 124*
animation *80, 87, 90*
appearance
  button object *79*
  forms *54, 56*
  graphics object *81*
  input field object *103*
  multiple form objects *63*
  table *313*
  text *62*
append unit from unit set *177*
append unit to number *100, 318*
application
  saving *330*
Application Argument *13, 134*
Application Builder *21*
  desktop environment *12, 20*
  window *12, 15, 18*
Application Builder Reference Manual *11, 233*
application example
  beam section calculator *413*
  B-H curve checker *380*
  concentric tube heat exchanger *392*
  equivalent properties of periodic microstructures *393*
  fiber simulator *415*
  frequency selective surface simulator *407*
  helical static mixer *375*
  induction heating of a steel billet *381*
  li-ion battery impedance *386*
  li-ion battery pack designer *385*
  lithium battery designer *384*
  microstrip patch antenna array synthesizer *409*
  mixer *399*
  organ pipe designer *383*
  plasmonic wire grating *416*
  Si solar cell with ray optics *411*
  transmission line calculator *376*
  truck mounted crane analyzer *401*
  tubular reactor *378*
  tuning fork *379*
  water treatment basin *388*
Application Gallery *29*
Application Libraries *10, 28, 29, 47, 374*
Application Library
  COMSOL Server *33*
application object *185*
Application Programming Guide *11, 185, 215*
application tree *12, 15, 17, 18*
applications
  publishing *45*
applications folder *10, 29, 374*

apply changes *24*
arguments
input and output *210*
arranging form objects *115*
array *167*
1D *167*
2D *168*
2D, interactively defining *169*
syntax *168*
array input object *156, 159, 298*
auto complete *205*
Automatic Notation *280*

B
background color *56*
background image *56*
Blank form template *57*
BMP file *234*
Boolean variable *164, 166, 167, 238*
conversion *366*
Boundary Point Probe *96, 184*
Break *213*
breakpoint *212*
browser
web *32, 136*
built-in method library *353*
button *67, 80, 113, 208*
command sequence *69*
icon *67*
keyboard shortcut *68*
on click event *67, 69*
size *67, 117*
style *67*
text *117*
tooltip *68*

C
C libraries
external *362*
CAD-file import *291, 332, 334*
Call Stack window *212*
cancel shutdown *151*
card *286*
card stack object *113, 164, 165, 286, 295*
cell margins *123, 130*
cells
merging *123*
splitting *123*
check box object *113, 156, 159, 166, 189, 208, 238*
check syntax *198*
choice list *66, 160, 170, 171, 246, 250, 255, 263, 264, 302, 303, 310, 311, 360*
clear selection
graphics *84*
click-through agreement *41, 42*
clipboard *268, 280, 326*
close application icon *151*
Close brackets automatically *204*
code completion *205*
tooltip *206*
code folding *204*
color
material *84*
selection *84*
column settings *121, 129*
combo box object *113, 156, 164, 170, 174, 244*
command line *134*
command sequence *18, 51, 52, 69, 72, 73, 79, 89, 143, 150, 154, 156, 185, 209, 210, 235, 316, 324*
comments
toggle on and off *203*
common, file scheme *332, 348*
compatible with physical quantity, unit dimension check *100*
compatible with unit expression, unit dimension check *100*

Compiler
button *38*
node *39*
compiler *10, 24, 27, 38*
complex numbers *280*
component syntax *204*
computation time *298*
expected *106, 295, 297*
last *106, 297, 365*
COMSOL Client *10, 26, 27, 32, 35*
file handing *328*
running applications in *35, 328*
COMSOL Compiler *10, 24, 27, 38*
COMSOL Desktop environment *12, 20*
COMSOL Multiphysics *10, 24, 25, 30, 31, 35, 131, 185, 215, 270, 276, 278*
COMSOL Runtime *39*
COMSOL Server *10, 24, 26, 27, 29, 31, 32, 35, 36, 332*
manual *38*
COMSOL Software License Agreement *45*
confirm *218, 359*
Continue *212*
Convert to Form Method *18, 74, 185, 209*
Convert to Local Method *19, 74, 185, 209*
Convert to Method *18, 73, 75, 185, 187, 209, 337*
Coordinate *182, 183*
Copy Table and Headers to Clipboard *280*
copying
forms and form objects *131, 326*
objects *60*
rows or columns *122*
Create Local Method *208*
Create Local Variable *207*
Create New Declaration and Use It as Source *99*
Create New Form Declaration and Use It as Source *99*
creating
forms *16, 47*
methods *18*
CSV file *164, 281, 316, 317*
curly brackets *204*
custom file type *172*
custom ribbon tab *226*
custom settings window *132*

D

DAT file *164, 281, 316, 317*
Data Access *110, 193, 197*
data change *65, 156, 210, 240, 241, 373*
data display object *103, 105, 113*
information node *297*
tooltip *108*
data file *164, 281, 316*
Data picking *183*
data picking *96, 183*
data validation *100, 177*
date *364*
dbfile, file scheme *332*
Debug Log *214*
debug log window *214, 362*
debugging *212, 362*
Decimal Notation *280*
Declaration and Use it as Source *162*
Declarations *14, 160, 162*
form *99, 160, 161*
global *99*
local *99*
declarations node *238*
delete button *60, 72, 293*
deleting an object *60*
Depth Along Line *184*
derived values *106, 279*

description text
Boolean variable *240*
derived value *104*
desktop icon *25, 40, 136*
desktop shortcut *25, 136*
Developer tab *18, 108, 133, 195, 221*
dialog box *359*
digital twin *13, 155*
Disable All *213*
disable form object *360*
display name, for choice list *170, 174, 247, 303, 310, 360*
displayed text *62*
Domain Point Probe *184*
domain point probe *96*
double variable *164, 166, 167*
conversion *366*
double, data validation *101*
download
option for compiling *39*
drag and drop, form objects *60*
duplicating
rows or columns *122*
duplicating an object *60*

E

edit local method *210*
edit node *70, 191, 192*
Editor Tools *191*
editor tools *65, 194, 249, 250*
window *18*
editor tree *66, 69, 83, 191, 329, 331*
element size *112*
change *261*
email *322*
class *356*
methods *356*
email attachment
export *356*
report *356*
table *356*
embed
option for compiling *40*
embedded, file scheme *233, 236, 332, 340*
Enable code folding *204*
enable form object *360*
enabled state, for form objects *79*
Engineering Notation *280*
equation object *265*
error message, data validation *100, 102*
Evaluation 2D *184*
Evaluation 3D *184*
evaluation tables *184*
event *65, 150, 156, 208, 210, 252*
About to shutdown *151*
button on click *69*
for multiple form objects *63, 158*
form *156*
form object *156*
global *13, 113, 150*
keyboard shortcut *68*
knob *321*
local *150*
node *151*
on close *158*
on data change *65, 156, 210, 240, 241, 373*
on load *65, 158*
on startup *151*
slider *319*
Events *13*
example application
bike frame analyzer *413*
charge exchange cell simulator *400*
corrugated circular horn antenna *405*
effective nonlinear magnetic curves calculator *382*
finned pipe *394*
forced air cooling with heat sink *395*

general parameter estimation *402*
geothermal heat pump *403*
inline induction heater *396*
polarizing beam splitter *416*
rotor bearing system simulator *410*
solar dish receiver designer *404*
thermoelectric cooler *397*
Excel® file *164, 281, 316*
executable *10, 24, 38*
explicit selection *92, 94*
exponent, number format *106*
export
email attachment *356*
export button, results table *280*
export node *328, 341*
Export Selected Image File *234*
exporting
results *328, 341*
external C libraries *362*
extracting subform *117*

F

File *160*
file
commands *329*
declaration *172*
destination *292, 335*
download *34, 330*
import *70, 156, 159, 172, 233*
menu *144*
methods *354*
opening *329*
saving *330*
types *293*
upload *34, 330*
file browser *346*
file custom *172*
file import object *156, 159, 172, 233, 291, 328, 334*
File menu editor *17*
file open
system method *356*
file scheme
common *332, 348*
dbfile *332*
embedded *233, 236, 332, 340*
syntax *233*
temp *332*
upload *172, 332, 338, 341*
user *332, 348*
File Type *160*
file type
declaration *172*
filename *292, 335, 354*
files library *236*
Find *199*
fit, row and column setting *117, 121*
fixed, row and column setting *117, 121*
for statement *217*
form *15, 55*
Declarations *160*
local *58*
form collection *113, 139, 283*
Form Editor *21*
desktop location *12*
overview *15*
preferences *20, 59*
using *55*
form event *156*
form method *18, 150, 156, 185*
form object *15, 59, 65, 237*
event *156*
with associated methods *189*
form reference *283*
Form tab, in ribbon *15*
form window *15*
Form wizard *65, 66, 104*
Forms *13*
Full Precision *280*
function *17*

G
gauge object *273*
parameter *274*
scalar variable *274*
variable *274*
geometry *31, 50, 69, 80, 86, 95, 305, 326, 336, 338, 359*
import *291, 332, 334*
operations *270, 276, 278*
Geometry Entity Level *184*
geometry node *69*
get *369*
GIF file *234*
global evaluation *106, 286*
global event *13, 150*
global method *18, 156, 185*
global parameter *217*
go to method *18, 73, 187*
graphics *74*
clearing contents *86*
commands *83*
hardware acceleration *45*
hardware limitations *85*
object *49, 50, 79, 80, 156, 359*
plot group *86*
Source for Initial Graphics Content *182*
source for initial graphics content *80*
tab, New Form wizard *50*
toolbar *86, 113*
using multiple objects *85*
view *83, 90, 361*
graphics data *96, 160, 182*
grid layout mode *34, 56, 105, 114*
grid lines, sketch layout mode *115*
grow, row and column setting *117, 121*
growth rules *117*

H
Home tab, in ribbon *47*
HTML
code *267*
report *267, 349*
HTTP and HTTPS protocols *322*
hyperlink object *321*

I
icon *234, 324*
button *67*
close application *151*
command *70*
desktop *25, 40, 136*
graphics *81*
help *102*
main window *138*
menu item *143*
method *189*
ribbon item *143*
toolbar *324*
if statement *217*
ignore license errors *26*
image
background *56*
formats *234*
object *268*
Preview *234*
scaling *268*
thumbnail *29*
Images library *234*
Immediately
Store changes *78*
import
file *70, 156, 159, 172, 233, 292, 335*
Indent and Format *203*
Indent and format automatically *204*
information card stack object *113, 295*
information node *297*
inherit columns *129*
initial size, of main window *140*
initial values, of array *167*
initialize
parameter *75*
variable *75*

Initializing Installer progress window *41*
input arguments *134, 210*
input field object *97, 113, 156, 159, 164, 177*
adding *97*
information node *297*
text object *104*
tooltip *99*
unit object *104*
Inputs *13, 134*
inputs tab, outputs tab, New Form wizard *49*
inserting
form objects *63, 65*
rows and columns *117, 120*
rows or columns *122*
integer
data validation *101*
variable *164, 166, 167*
variable conversion *366*
interactive editing
menus and ribbon items *146*
item
menu *143, 208*
ribbon *145*
toolbar *323*

J
Java utility class *233*
Java® programming language *185, 215*
JPG file *30, 234*

K
keyboard shortcut *20, 65, 150, 192, 205, 207, 213, 350*
event *68, 143, 324*
knob object *319*
parameter *320*
scalar variable *320*
variable *320*

L
Language *141*
language elements window *18, 190, 215*
language localization *141*
LaTeX *104, 107, 265*
layout mode *56, 114*
layout options, form collection *283*
layout template *16, 48, 59*
Libraries *14*
libraries node *233, 268*
license agreement *45*
license errors
ignoring *26*
Line Entry Method *184*
line object *266*
list box object *113, 156, 159, 164, 170, 174, 307*
LiveLink™ for Excel® *164, 281, 316*
LiveLink™ products *35*
local event *150*
local form *58*
local method *18, 65, 74, 150, 156, 159, 185, 189, 208, 210, 240, 247, 326*
local variable *207*
Localization *141*
log object *276*
logo image *81*
low-resolution displays *34*

M
Main Form *139*
Main Window *13, 16, 139, 270*
node *138, 147*
Main Window Editor *21, 146*
desktop location *12*
overview *16*
Main window type *139*
margins
cell *123, 130*
material *253*
material color and texture *84*
math functions *216*
menu *143, 145*
bar *139, 142*

item *80, 113, 143, 208*
toggle item *143, 242*
Menu editor *17*
menu toggle item *113*
merging cells *117, 123*
mesh *50, 80, 86, 112*
change element size *261*
size *112*
meshing *270, 276, 278*
message log object *277, 359*
method *14, 17, 63, 73, 80, 160, 185, 353*
event *151, 156*
form *18, 150, 156, 185*
form object *189*
global *18, 156, 185*
local *18, 65, 74, 150, 156, 159, 185, 189, 208, 240, 247, 326*
Model Builder *219*
window *18*
Method Call *219, 226*
Method Editor *21, 353*
desktop location *12*
overview *17*
Preferences *204*
using *185*
Method Errors and Warnings window *198*
Method tab, in ribbon *18*
method, called from the Model Builder *219*
Methods *14*
Microsoft® Word® format *347*
minimum size
form objects *123*
Model Builder
method *205, 219*
model commands *331*
model data access *113, 153, 326*
Model Expressions *200*
model expressions window *18*
model object *185, 215, 353*
model tree node, controlling if active *219*
model utility methods *353*
move down
command sequence *72*
move up
command sequence *72*
MP4 file *269*
MPH-file *14, 24, 25, 27, 31, 47, 53, 233, 331, 362*
multiline text *105*
multiple form objects
selecting *63, 158*

N

name
button *67*
check box *240*
choice list *170*
form *55*
form object *63*
graphics object *79*
menu *143*
method *203*
shortcut *179*
variable *163*
named selection *92*
new element value *167*
new form *16*
New Form wizard *64, 104*
graphics tab *50*
inputs and outputs tabs *49*
ribbon buttons tab *51*
new method *18*
notation
data display number format *106*
unit *106*
number format *103, 106*
number of rows and columns *117*

numerical validation *101, 177*

O
OGV file *269*
on click event, button *67*
on close event *158*
on data change event *65, 156, 210, 240, 241, 373*
on load event *65, 158*
On request
Store changes *78*
on startup event *151*
open file *329*
OpenGL graphics hardware acceleration *45*
operating system command line *134*
operators *215, 216*
optimization *387*
orthographic projection *83*
OS commands *356*
output arguments *210*
Output directory
for compiled applications *39*

P
panes *283*
parameter *17, 50, 74, 98, 100, 164, 217, 326, 369*
combo box object *244*
declarations *14, 160*
events *13, 150*
input field object *97*
method *197, 217*
slider object *318*
text label object *104*
parentheses *204*
password protected application *31*
pasting
form objects *60*
forms and form objects *131*
image *268*
rows or columns *122*
pixels *62, 114*
play sound *34, 235, 356*
plot *50, 51, 69, 70, 80, 86, 166, 238, 248, 311, 337, 345, 359*
plot geometry command *70*
plot group *74, 166*
PNG file *30, 234*
Point Being Modified *184*
Polar Complex Numbers *280*
position and size *62, 114, 116*
multiple form objects *63*
positioning form objects *59*
precedence, of operators *216*
precision *280*
precision, number format *106*
preferences *20, 59, 204, 332, 333*
for compiled applications *44*
security *31*
Preview
image *234*
preview form *24*
printing
graphics *84, 362*
Probe *184*
procedure *17*
Programming Reference Manual *11*
progress *270, 364*
progress bar *139*
progress bar object *270, 364*
progress bar, built in *270*
progress dialog box *272, 364*
publishing applications *45*

Q
Quick Access Toolbar *24*
Find *199*

R
radio button object *113, 156, 170, 174, 301*
Record Code *195*
recording code *195*

Rectangular Complex Numbers *280*
recursion *211*
regular expression *102*
Remove All *213*
removing
password protection *31*
rows and columns *117, 120*
rows or columns *122*
report *371, 374*
creating *328, 341, 347, 348*
creating automatically *220*
email attachment *356*
embedding *267*
HTML *267, 349*
image *30*
node *328, 341, 348*
request *218, 359*
reset current view *83, 90*
Reset Window Layout *52*
resizable graphics *34*
resizing form objects *60*
Results Evaluation *182, 183*
results table object *279, 360*
ribbon *139, 144*
item *113, 145*
section *145*
tab *145*
toggle item *113, 145, 242*
Ribbon button *68*
ribbon buttons tab, New Form wizard *51*
ribbon tab
custom *226*
Ribbon tab editor *17*
row settings *120*
run application *24, 26*
running applications
compiled *38*
in a web browser *32, 328*
in the COMSOL Client *35*
runtime *39*

S

save
application *53*
running application *27*
save application command *330*
save as *362*
save file *330*
Scalar *164*
scalar variable *164, 246, 286, 318*
scene light *83, 362*
Scientific Notation *280*
security settings *31*
select all
graphics *84*
selection *50, 80, 86*
explicit *94*
selection colors *84*
selection input object *94, 304*
selections *92*
add-in *95*
selectNode method *219*
separator
menu *143*
ribbon *145*
toolbar *143, 323*
separators
CSV, DAT, and TXT files *317*
set value command *112*
Settings Form *133, 219, 226*
Settings Forms *132*
settings window
customized *132*
Form editor *12, 15*
Method editor *18*
shortcut
desktop *25, 136*
shortcuts *160, 179*
use *201*

Show as Dialog command *75*
Show Dialog *133*
Show Errors *131*
show form command *76*
Show in Model Builder *132*
shutdown
  cancel *151*
Single form template *57*
Single window *139*
sketch grid *115*
sketch layout mode *56, 105, 114*
slider object *113, 156, 317*
smartphones
  running applications on *34*
software rendering *45*
solving *270, 276, 278*
sound
  play *235*
Sounds library *234*
Source for Initial Graphics Content *182, 183*
spacer object *324*
special characters *136*
splash screen *40, 41*
splitting cells *117, 123*
state
  enabled, for form objects *79*
  visible, for form objects *79*
Status bar *139*
status bar *270*
Step *212*
Step Into *213*
Step Out *213*
Stop *212*
Stop Recording *197*
stopping a method *212*
Store changes *78*
Stretch image *268*
string variable *13, 75, 150, 152, 164, 167, 245, 247, 255, 284, 296, 307*
  conversion *366*
  methods *368*
subroutine *17*
Subwindow editor *17*
subwindow layout *16*
Subwindows *139*
subwindows *147*
Switch to Model Builder and Activate Data Access *99*
syntax errors *198*
syntax highlighting *203, 205*
system methods *356*
  OS commands *356*

T

table
  email attachment *356*
table object *156, 167, 312, 360*
  add to table *313*
  source *313*
tables, model tree *279*
tablets
  running applications on *34*
Target for Data Picking *96, 183*
temp, file scheme *332*
template *16, 48, 59*
templates *147*
temporary file *346*
test application *24, 26*
test in web browser *24*
text *143*
text color *56*
text file *164, 280, 316*
text label object *97, 104, 106*
text object *113, 156, 159, 307*
  information node *297*
Theme *205*
Themes *13*
Themes node *54*
thumbnail image *29, 30*

time *364*
time parameter
combo box object *250*
Timer events *13, 155*
timestamps *278*
title
form *55*
main window *138*
menu *143*
toggle button *113, 242*
size *117*
text *117*
toggle item
menu *113, 143, 242*
ribbon *113, 145, 242*
toolbar *323*
toolbar *142, 280, 323*
button, table object *316*
graphics *86, 113*
item *113, 323*
separator *143, 323*
tooltip
button *68*
data display object *108*
input field object *99*
knob object *320*
method editor *206*
slider object *275, 318*
toolbar button *324*
unit mismatch *101*
transparency *83, 362*
TXT file *164, 280, 316, 317*

U

Unicode *104, 107*
unit
changing using unit set *174*
data display *106*
dimension check *100, 101, 177*
expression *100*
groups *174*
lists *174*
object *97, 104*
Unit Groups *174*
Unit Lists *174*
Unit Set *174*
unit set *102, 160, 264, 303, 311*
Untitled.mph *27*
upload
file scheme *172, 332, 338, 341*
URL *136, 267, 322*
use as source
array input object *300*
card stack object *287*
check box object *240*
combo box object *246*
data display object *106*
declaration *162*
explicit selection *94, 304*
gauge object *274*
graphics object *80*
information card stack object *296*
input field object *97*
knob object *320*
list box object *309*
radio button object *302*
results table object *280*
selection input object *94, 304*
slider object *318*
text object *307*
Use component syntax *204*
use shortcuts *201*
user
file scheme *332, 348*
user interface layout *15*
username *356*

V

Value *170, 174*
variable *14, 160, 197*
accessing from method *216*
activation condition *171*

Boolean *166, 167, 238*
declaration *14, 160*
derived values *106*
double *166, 167*
events *13, 150, 152*
input field object *97*
integer *166, 167*
name completion *205*
scalar *246, 286, 318*
slider object *318*
string *164, 167, 245, 247, 284, 296, 307*
text label object *104*
Variables window *212*
video
controls *270*
player *270*
video object *269*
view
go to default 3D *89*
graphics *83, 90, 361*
reset current *83, 90*
View all code *204*
visible state, for form objects *79*
volume maximum *106*

W WAV file *234*
web browser *10, 27, 32, 136*
file handling *328*
web page
hyperlink *322*
web page object *267*
WebGL *32*
WebM file *269*
while statement *217*
Window Layout *52, 139, 148*
with statement *204, 217, 368*
with statements *204*
wrap text
text label object *105*

Z zoom extents *80, 83, 311, 337, 362*