COMSOL SCRIPT™

USER'S GUIDE

VERSION 1.2

COMSOL

How to contact COMSOL:

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

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

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

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

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

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

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

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

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

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

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

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

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

info@comsol.com
www.comsol.com

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

Company home page
www.comsol.com

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

COMSOL Script User's Guide

Version: October 2007 COMSOL 3.4

C O N T E N T S

Chapter 1: Introduction

Chapter 2: COMSOL Script Basics

Chapter 3: Vectors, Matrices, and Arrays

Chapter 4: Data Types for Non-Numeric Values: Strings, Cell Arrays, Structures

Chapter 5: The Programming Language

Chapter 6: Linear Algebra and Matrix Functions

Chapter 7: Scripts, Functions, and M-files

Chapter 8: Data Analysis, Statistics, and I/O

Chapter 10: Solving Differential Equations

Chapter 11: Creating User Interfaces

Chapter 12: User-Defined Classes

Chapter 13: Java Interface

Chapter 14: External API

1

Introduction

Welcome to COMSOL Script™! This *User's Guide* details features and techniques that help you use this powerful scripting language for all sorts of technical computing. Through examples and code samples you will get an understanding of the programming language; its data types; its mathematical, logical, and other functions; and its powerful graphics capabilities. This book also provides an introduction to the possibilities to build customized graphical user interfaces using COMSOL Script and to interface with the Java programming language.

This introductory chapter provides an overview of COMSOL Script.

The Documentation Set

The documentation set for COMSOL Script consists of the following volumes:

- *COMSOL Quick Installation Guide*—basic information for installing the COMSOL software and getting started. Included in the DVD/CD package.
- *COMSOL New Release Highlights*—information about new features and models in the 3.4 release. Included in the DVD/CD package.
- *COMSOL License Agreement*—the license agreement. Included in the DVD/CD package.
- *COMSOL Installation and Operations Guide*—besides covering various installation options for COMSOL Script, it describes the system requirements and options for running various COMSOL software products.
- *COMSOL Script User's Guide*—the book you are reading, it explains how to use the vast range of functions in the COMSOL Script language. This guide also describes the programming-language features in COMSOL Script as well as the powerful graphics capabilities and tools it provides for creating custom graphical user interfaces.
- *COMSOL Script Command Reference*—provided only as online documentation as a PDF and in HTML format, it reviews each function in the COMSOL Script environment with syntax descriptions and examples.

If you have received COMSOL Script together with COMSOL Multiphysics® 3.4, the full documentation set that ships with COMSOL Multiphysics additionally includes the following titles:

- *COMSOL Multiphysics Quick Start and Quick Reference*—provides a quick overview of COMSOL Multiphysics' capabilities and how to access them. A reference section contains comprehensive lists of predefined variable names, mathematical functions, COMSOL Multiphysics operators, equation forms, and application modes.
- *COMSOL Multiphysics User's Guide*—covers the functionality of COMSOL Multiphysics across its entire range of capabilities from geometry modeling to postprocessing. It serves as a tutorial and a reference guide to using COMSOL Multiphysics at every stage in the modeling process.

- *COMSOL Multiphysics Modeling Guide*—provides an in-depth examination of the software's application modes and how to use them to model different types of physics and to perform equation-based modeling using PDEs.
- *COMSOL Multiphysics Model Library*—consists of a collection of ready-to-run models that cover many classic problems and equations from science and engineering. These models have two goals: to show the versatility of COMSOL Multiphysics and the wide range of applications it covers; and to form an educational basis from which you can learn about COMSOL Multiphysics and also gain an understanding of the underlying physics.
- *COMSOL Multiphysics Scripting Guide*—shows how to access all of COMSOL Multiphysics's capabilities within the COMSOL Script environment or MATLAB.
- *COMSOL Multiphysics Reference Guide*—provided only as online documentation as a PDF file and in HTML format, it reviews each command that lets you access the functions in COMSOL Multiphysics from within the COMSOL Script environment or MATLAB. Additionally, it describes some advanced features and settings in COMSOL Multiphysics, and provides background material and references.

In addition, each of the optional discipline-specific modules

- *AC/DC Module*
- *Acoustics Module*
- *Chemical Engineering Module*
- *Earth Science Module*
- *Heat Transfer Module*
- *MEMS Module*
- *RF Module*
- *Structural Mechanics Module*

comes with its own *User's Guide* and *Model Library*.

The optional *CAD Import Module* and *Material Library* also come with their own *User's Guide*.

Note: The full documentation set is available in electronic versions—as PDF files and HTML format—after installation.

Typographical Conventions

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

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

About COMSOL Script

We are delighted you have chosen COMSOL Script for your technical computing needs. We believe that we have created a software tool that both complements and expands on the capabilities in the COMSOL Multiphysics multiphysics-modeling environment.

At its heart, COMSOL Script is a fully-featured technical analysis and visualization package. A command-line interface enables easy access to data, functions, and results. Its roughly 500 commands cover linear algebra, numeric and trigonometric calculations, support for objects and classes, as well as data visualization in both 2D and 3D. The COMSOL Script Desktop provides an interactive development environment for creating, editing, running, and debugging your scripts and functions. Because COMSOL Script is an interpreted language, you can quickly experiment with various commands. You can easily save a command sequence to create an optimized routine in a text editor.

Further, COMSOL Script's graphing and visualization capabilities set new standards for packages in this category. A set of Java-based tools enable you to quickly construct professional-looking graphical user interfaces. You can build a sophisticated model and then create a user interface that makes the model's functionality accessible to other users who do not need or have a mastery of the specific equations or physics upon which the model is based.

While COMSOL Script offers enormous power as a standalone technical computing environment, its competitive edge sits in a seamless integration with COMSOL Multiphysics, our flagship scientific-modeling software. Easy-to-use scripting combined with powerful modeling capabilities—in one and the same environment—fill a gap in the scientific-software market. Any model you create can be saved as a text-file representation that you can run in COMSOL Script to perform tasks such as design-optimization and parametric studies. You can, for instance, write a script that solves a model for a range of values to optimize that model around one or more variables. You can similarly automate many routine tasks such as creating a series of different geometries or solving using a series of different meshes to check the convergence of the solution.

This release also brings two add-on labs, the Optimization Lab and the Signals & Systems Lab, which bring additional functionality to COMSOL Scripts in the areas of optimization, signal processing, and systems analysis.

We hope that this documentation set introduces you to all of this power. We're also certain that your imaginations will come up with some very creative uses of COMSOL Script. We're anxious to hear about your innovative uses and invite you to get in touch with us with any feedback whatsoever. We plan on developing it even further, so let us know what kinds of functionality would serve you best.

2

COMSOL Script Basics

COMSOL Script provides a powerful scripting environment for many technical-computing applications. The following chapter provides an overview of the user interface and reviews the fundamentals for creating variables, statements, script files, and graphics.

Using COMSOL Script

Starting COMSOL Script

If you want to run COMSOL Script as a standalone program, start it by double-clicking the COMSOL Script icon (on the Windows desktop) or, on other platforms, by starting the program through a type of Start menu or from a command prompt.

To run COMSOL Script as a part of COMSOL Multiphysics, start it by choosing **File>COMSOL Script** in the latter environment.

The COMSOL Script Environment

THE DESKTOP ENVIRONMENT

COMSOL Script provides a desktop environment with several different views. The most important is the **Command Prompt** in which you interactively enter commands and also run scripts and functions, which hold sets of commands that you store as

M-files (text files with the extension `.m`). The desktop also contains an editor with which you can edit M-files.

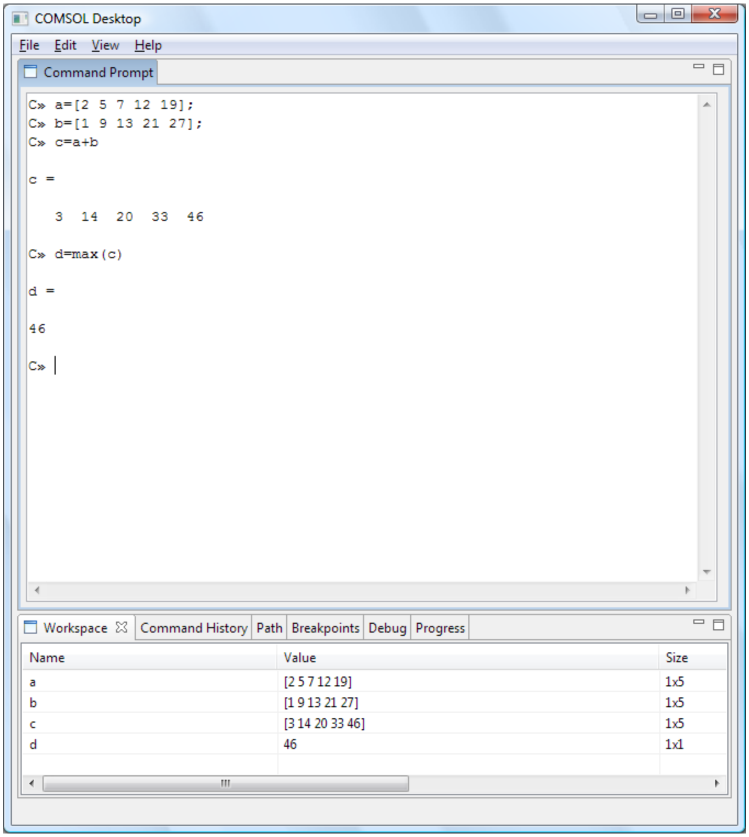


Figure 2-1: The COMSOL Script desktop environment.

THE DESKTOP VIEWS

COMSOL Script provides a number views into your code that are useful at various stages of code development and execution. These windows are not static; you can rearrange the views by dragging and dropping them inside the desktop window.

The **Workspace** view shows the variables in the active workspace.

The **Command History** view shows a history of the commands that you have entered at the command prompt. Double click on a command if you want to execute it again.

The **Path** view shows the path used to look for M-files. See “The M-file Path” on page 121 for information on how to add and remove directories to the path.

The **Breakpoints** view list the break points that have been set in different M-files. Double-click on a line to open the M-file in the editor with the line of the break point selected. See “Debug Commands” on page 99 for information on how to add and remove break points.

The **Debug** view shows the debug call stack when COMSOL Script has stopped at a break point. You can double click on different lines in the debug call stack to move to that point in the call hierarchy. During a debug session, the **Workspace** view can display the variables for the function being debugged.

In the **Progress** view you can see progress and convergence information such at times when you call command-line functions for things such as meshing and solving for a COMSOL Multiphysics model.

THE MAIN MENU

At the top of the desktop window you see a menu with four selections. The **File** menu and the **Edit** menu contain commands for the editor that is built into the desktop. The **View** menu contains commands for reopening any of the desktop views if you have closed them, and the **Help** menu leads you to documentation and help resources.

The **File** menu contains these commands:

- **New Editor:** This command opens a new session of the editor with an empty file.
- **Open File:** Select this command to browse for a file to open in the editor. You can also type `edit filename` at the command prompt to open a certain file.
- **COMSOL Multiphysics**: Select this command if you wish to run COMSOL Script together with COMSOL Multiphysics. This command brings up the **Model Navigator** window, which is the starting point for all COMSOL Multiphysics work. Note that this command exists only if you have COMSOL Multiphysics installed on your system.
- **Reaction Engineering Lab**: Select this command if you wish to run COMSOL Script together with the COMSOL Reaction Engineering Lab. Note that this command exists only if you have the COMSOL Reaction Engineering Lab installed on your system.

- **Exit**. Choose this command to exit the COMSOL Script environment, which also means that you lose the contents of the current workspace.
- The **File** menu also contains menu items for closing the active editor or all editors as well as for printing the contents of the active editor.

The **Edit** menu contains commands that are associated with the editor. They can be used to undo and redo things you have typed in. It also contains commands for cut, copy/paste, and for finding text in an editor window.

COMMAND-LINE EDITING AND NAVIGATION

Within the COMSOL Script editing window there is a command prompt. You can move the cursor within the current command as well as up and down the command history using the shortcut keys and navigation keys listed in Table 2-1.

TABLE 2-1: COMSOL SCRIPT SHORTCUT KEYS

KEY	COMMAND
Up arrow	Recall previous line
Down arrow	Recall next line
Left arrow or Ctrl+B	Move left one character
Right arrow or Ctrl+F	Move right one character
Ctrl+left arrow	Move left one word
Ctrl+right arrow	Move right one word
Home or Ctrl+A	Move to beginning of line
End or Ctrl+E	Move to end of line
Esc	Delete current line
Delete	Delete character at cursor
Backspace	Delete character left of cursor
Ctrl+K	Delete from cursor to end of line

With these shortcut keys it is easy to correct a typing mistake or to make small changes to a command. In addition, COMSOL Script saves previous commands in a buffer (the scrolling buffer, for which you can control the size in the **Preferences** dialog box).

STORING COMMAND WINDOW INPUT AND OUTPUT IN A FILE

COMSOL Script can create a text file that stores all commands that you type and all output from the software. Use the `diary` function to start the logging of command window inputs and outputs to a file. Typing

```
diary mylog.txt
```

stores all subsequent inputs and outputs that appear in the command window to the file mylog.txt. You can temporarily turn the logging off and flush the file by typing

```
diary off
```

To turn logging back on type

```
diary on
```

CLEARING THE COMMAND WINDOW

To clear the command window enter the command `clc`.

Using Commands and Creating Variables

CREATING VARIABLES AND ENTERING COMMANDS

COMSOL Script supports several data types and objects (see "The COMSOL Script Data Types" on page 22). As an example, suppose you want to create an array of double floating-point numbers using an explicit list of data. To do so, enclose the list using brackets (`[` and `]`) and type the data separated by blanks or commas. Use a semicolon to indicate the end of a row.

Enter the following to create a 3 x 3 identity matrix:

```
I = [1 0 0; 0 1 0; 0 0 1]
```

This results in the output

```
I =

 1 0 0
 0 1 0
 0 0 1
```

Use a semicolon at the end of the statement to prevent COMSOL Script from displaying the result:

```
I = [1 0 0; 0 1 0; 0 0 1];
```

In either case, the result is stored in the variable `I`, which remains in the main workspace until you clear the variable or exit from COMSOL Script.

If you want to continue with more commands or data entry on the next line, type `...` (an ellipsis), which acts as a line continuation symbol. COMSOL Script then interprets the entries on the next line as a continuation of the current line:

```
K = [1 2 3 4 5 6 7 8 9 10; ...
11 12 13 14 15 16 17 18 19 20];
```

These two lines create a 2 x 10 matrix with the numbers 1 to 10 in the first row and the numbers 11 to 20 in the second row (the semicolon indicates the start of a new row).

STATEMENTS, VARIABLES, AND THE WORKSPACE

COMSOL Script interprets and evaluates all statements that you enter from the keyboard (or statements contained in a script or function you call). Statements typically take the form

```
variable = expression
```

or

```
expression
```

The `expression` can contain mathematical functions and operators, data, and other variables. COMSOL Script evaluates the `expression` and assigns the result to the variable that appears to the left of the assignment operator `=`. If you enter the `expression` directly without assigning it to a variable, COMSOL Script automatically creates the variable `ans`. For instance:

```
2*pi
```

produces

```
ans =

6.283185
```

Getting Information about the Workspace and Variables

To list all the variable in the workspace, enter the command `who`:

```
who

I
ans
```

This output shows that there are two variables in the workspace, `I` and `ans`. To get more detailed information, type `whos`:

```
whos
  Name     Size     Memory     Type

  I        3x3      72         double array
  ans      1x1      8          double
```

Each element of a real double floating-point number requires eight bytes of memory.

Clearing the Workspace

To clear the workspace, use the clear command in one of its two forms:

```
clear all
```

clears all variables from the workspace, whereas

```
clear I
```

clears only the variable `I`. Enter `clear` *`variable1 variable2`* … to clear more than one variable.

Upper and Lower Case for Variables and Functions

COMSOL Script is case sensitive, so `a` and `A`, for example, represent different variables. For functions and operators, you must use only lower case: `asin(I)` gives a matrix with $\pi/2$ on the diagonal, whereas `ASIN(I)` produces an error because the software does not recognize the unknown function `ASIN`.

INPUT AND OUTPUT ARGUMENTS

Functions can have multiple input and output arguments:

```
[a,i] = sort(rand(5),2);
```

This example with the function `sort` uses two input arguments and two output arguments. Notice that the first input argument contains a function, `rand`.

Even if a function can have one or more output arguments, you can often call it with fewer or no output arguments. Compare the two function calls:

```
a = sort(rand(5),2);
sort(rand(5),2)
```

Using just one output argument (as in the first function call) you do not get the index array that `sort` provides as a second output argument. Using no output argument (as in the second function call), COMSOL Script returns the output in the variable `ans`.

Note: COMSOL Script never modifies any input arguments. Also, if the function does not modify the input arguments, the copy of the variables that you provide as inputs is by reference and not by value.

ENTERING FLOATING-POINT NUMBERS

To enter floating-point numbers use conventional decimal notation with an optional decimal point and a leading minus sign. Use either `E` or `e` to indicate an exponent in a power-of-ten scale factor: Both `-1.527E-10` and `-1.527e-10` equal $-1.527\cdot10^{-10}$.

COMSOL Script uses IEEE double-precision floating-point arithmetic, which gives a relative accuracy of 2^{-52} or approximately 16 significant digits. You can access this level of relative accuracy in the function `eps`, which provides the difference between a number and the next larger number. The range for a floating-point number is approximately 10^{-308} to 10^{308}.

USING OTHER NUMERICAL FORMATS

It is possible to create unsigned integer arrays using the `uint8` function. For example,

```
u = uint8(4);
```

creates the variable `u` that contains 4 as an unsigned integer, using 1 byte of storage instead of 8 bytes for the corresponding double-precision floating point number.

The functions `int8`, `int16`, `int32`, `int64`, `uint16`, `uint32`, and `uint64` also exist but return double-precision floating point data.

BASIC ARITHMETICS AND MATHEMATICAL FUNCTIONS

Table 2-2 lists the arithmetic operators built into the COMSOL Script language:

TABLE 2-2: ARITHMETIC OPERATORS

OPERATOR	ARITHMETIC OPERATION
+	Addition, unary plus
-	Subtraction, unary minus
*	Multiplication
/	Right division
\	Left division
^	Power

(For the difference between the right- and left-division operators see “Division” on page 44). The language also supplies element-by-element versions of the multiplication, right division, and power operators as well as a full range of trigonometric, logical, and other mathematical functions. See “Working with Matrices and Arrays” on page 35 for more information.

Operator Precedence and Associativity

COMSOL Script evaluates its operators using the following order of precedence:

1 Power (`^`)
2 Unary minus (`-`)
3 Multiplication and division (`*`, `/`, `\`)
4 Addition and subtraction (`+`, `-`)

You can force a different precedence using parentheses. The software evaluates the expressions from left to right using *left associativity*, which means that, for example, `12/4/3` evaluates to 1, just as does `(12/4)/3`, whereas `12/(4/3)` evaluates to 9.

Mathematical Functions and Constants

COMSOL Script includes, as built-in functions, most elementary mathematical functions such as `abs`, `sqrt`, `sin`, `log`, and many more. Full lists of all mathematical and logical functions and operators appear in the sections "Elementary and Special Math Functions" on page 42 and "Relational and Logical Operators and Functions" on page 50.

Further, several other built-in functions return mathematical constants:

- `pi` returns π.
- `i` and `j` each return i, the imaginary unit (the square root of −1).
- `Inf` stands for infinity. For example, the result of dividing a nonzero number by zero is `Inf`.
- `NaN` stands for "Not a Number" in the IEEE standard. NaN is the result of a mathematically undefined operations such as `0/0`.

Note: You can override the values of `pi`, `eps`, `i`, and `j` by defining variables using any of those names. This can lead to unexpected results. If you want to use `i` and `j` as named variables, then use `sqrt(-1)` for the imaginary unit.

Using Complex Numbers

You can use complex numbers in all COMSOL Script functions and operations. For example, to enter a matrix with complex-valued elements type either:

```
C = [1 0 1; 1 2 3; 0 1 0] + i*[1 1 1; 4 5 6; 1 0 1]
```

or the following line, which produces the same result:

```
C = [1+i i 1+i; 1+4i 2+5i 3+6i; i 1 i]
```

To get the real and imaginary parts of complex-valued data, use the `real` and `imag` functions, respectively. The `abs` function computes the absolute value:

```
x = 1+i;
imag(x)
ans =
  1
abs(x)
ans =
  1.414214
```

OUTPUT FORMATS

In its command window, COMSOL Script displays the results of any statements you enter unless the statements end with a semicolon. You control the way the output appears using the `format` command; internally, however, COMSOL Script performs all computation using full double-precision floating point numbers.

The default display format, `short`, shows roughly six significant decimal digits, except for integers:

```
A = [3 pi 1/3]
A =
  3  3.141593  0.333333
```

To show the data using full precision, use `format long` or `format hex`:

```
format long
A
A =
  3  3.1415926535898  0.3333333333333

format hex
A
A =
  4008000000000000 400921fb54442d18  3fd5555555555555
```

Type `format` with no argument to return to the default format.

Checking the Computer and Operating System

To see the characteristics of the computer on which you run COMSOL Script, use the `computer` function:

```
computer

ans =
```

```
PCWIN
```

The `computer` function can also returns the maximum number of bytes that a matrix can occupy.

In addition, use the function `ispc` and `isunix` to check if COMSOL Script is running on a PC or under UNIX.

Running General Operating System Commands

The functions `unix`, `dos`, and `system` all provide access to operating system commands. These functions run the command in the input string and provide an exit code as the output. If the function could run the command successfully, the exit code is 0. An optional second output argument contains the output of the command. For example, the following three statements all store the contents of the working directory as a string in the variable `out`:

```
[status, out] = dos('dir');
[status, out] = system(['cmd.exe /C ' 'dir']);
[statis, out] = unix('ls');
```

Use the two first statements, which are equivalent, when you run COMSOL Script in Windows; use the last statement in a UNIX or Linux environment.

Displaying and Changing Directory

COMSOL Scripts contains functions for displaying the name and contents of the current directory and to change directory.

To print the current (working directory) type:

```
pwd
```

To list the contents of the current directory use the `dir` function:

```
dir
```

You can also use an output argument:

```
f = dir;
```

This statement returns a structure array where each element contains four fields that contain the name, creation date, the file size (in bytes), and a Boolean flag, `isdir`, which is True for a directory and False for a file.

You can also list other directories using `dir` by passing the path as an input argument. The statement

```
dir('C:\')
```

lists the files and directories that reside directly under `C:\`.

The function `ls` is the same as `dir`.

To change the current directory use the function `cd`:

```
cd ..
```

moves one level up in the directory structure.

Running Commands at Startup

It is convenient to run some commands, such as adding paths to directories where you store M-files or predefined variables, directly at startup. Put those commands in a script file and save it as `startup.m` in the COMSOL Script startup directory. COMSOL Script then runs this script file each time you start the software.

For more information about scripts in general, see "Scripts" on page 124.

Getting Help

From the COMSOL Script Help Desk you have access to this book and the *COMSOL Script Command Reference* in HTML and PDF formats. To access the Help Desk, go to the **Help** menu at the top of the COMSOL Script command window and choose **Help Desk**, or simply press F1.

There is also online help available at the command prompt. Type `help` to get a list of help topics. To get help on a specific topic, function, or operator, type `help` *`topic`*. For example,

```
help plot
```

provides information about the use of the `plot` command for creating plots.

Saving the Workspace and Exiting the Program

To save variables in the workspace to a file, use the `save` command. For instance,

```
save temp A C
```

saves the variables A and C to the file `temp.flws`. Use the command `load` to restore variables into the COMSOL Script workspace:

```
load temp
```

To exit COMSOL Script, type `quit` or `exit`. Doing so clears all variables and data from the workspace, so be sure to save any workspace information if you plan to use it again.

3

Vectors, Matrices, and Arrays

The COMSOL Script environment offers a variety of data types that make for flexible programming. This and the following chapter review the available data types, their properties, and how to use them. This chapter focuses on matrices, sparse matrices, arrays, logical arrays, and the operations available to operate on them including indexing and subscripting, matrix inversion and transposition, and mathematical operators.

The next chapter, “Data Types for Non-Numeric Values: Strings, Cell Arrays, Structures” on page 61 focuses on data types that also work with non-numeric data such as character arrays, cell arrays, structures, as well as the special operators and functions available to work on them.

The COMSOL Script Data Types

Overview of Data Types

COMSOL Script provides a number of data types (classes):

- Double arrays: 1D arrays (vectors), 2D arrays (matrices), and multidimensional arrays of double floating-point complex numbers.
  - Class name and constructor function: `double`
- Sparse double arrays, which are similar to double arrays but store only the nonzero elements (and their locations).
  - Class name and constructor function: `sparse`
- Logical arrays of any dimension containing elements that are True or False.
  - Class name and constructor function: `logical`

The above data types are the subject of this chapter; the following data types are the subject of the following chapter:

- Character arrays (strings): an array of strings. For multiple strings of different length, it is convenient to use a cell array of strings.
  - Class name and constructor function: `char` (`cell` for cell array of strings)
- Cell arrays, which are arrays of any dimension where each element is a cell that can contain any other data type, for example, a string, a matrix, or another cell array.
  - Class name and constructor function: `cell`
- Structures: Structure arrays are flexible data types that can include multiple fields and values. The contents of each field can be any data type, for example, a string, a matrix, or another structures.
  - Class name and constructor function: `struct`
- Inline function object: An object that defines an inline function.
  - Class name and constructor function: `inline`
- Java object: It is possible to create Java objects and arrays of Java objects in the COMSOL Script workspace. See "Java Interface" on page 301 for more information about using Java objects.

Identifying Data Types

To list all the variables in the workspace and their data types, enter the `whos` command.

Alternately, to get the class (data type) of an object (variable), use the `class` function:

```
l=true;
class(l)

ans =

logical
```

To check if a variable is of a certain data type, use the `isa` function:

```
isa(l,'logical')

ans =

true

isa(l,'double')

ans =

false
```

Creating Vectors, Matrices, and Double Arrays

Vectors, matrices, and arrays are the workhorse data types in COMSOL Script, and this package makes it very easy to create them. Vectors and matrices are common instances of double arrays, those that use double floating-point numbers to store values in each element. The following section describes the various commands that allow you to create these data types.

Creating Double Arrays

COMSOL Script offers several ways to create a double array:

- Enter values directly with by using brackets (`[]`) to enclose all the data and semicolons (`;`) to indicate the end of rows (see the examples in this section as well as in Chapter 2, "COMSOL Script Basics,")
- Use the colon (`:`) operator to create a vector
- Use special functions to create vectors (`linspace`, `logspace`)
- Use special function to create arrays (`ones`, `zeros`, `rand`, `randn`, `eye`)
- Use special functions to create sparse matrices (`sparse`, `speye`)
- Read data from a binary data file (a file that contains workspace data and that you save using the `save` function) or text file using the `load` function.

USING THE COLON (:) OPERATOR TO CREATE VECTORS

The colon operator (`:`) can create a row vector with a unit increment from the starting value to the ending value. For example:

```
x = 1:10
x =
  1   2   3   4   5   6   7   8   9   10
```

To create vectors with increments other than one, use the syntax *start*:*increment*:*end* as in:

```
x = 0:.2:1
x =
  0   0.200000   0.400000   0.600000   0.800000   1
```

If the ending values is smaller than the starting value, you can use a negative increment to create a vector of decreasing numbers:

```
x = 10:-2:0
x =
  10  8  6  4  2  0
```

USING LINSPACE AND LOGSPACE TO CREATE VECTORS

Another way to create a vector of equally spaced numbers is to use the `linspace` function:

```
x = linspace(0,1,4)
x =
  0  0.333333  0.666667  1
```

`linspace(`*`start`*`,`*`end`*`,`*`N`*`)` creates a vector of N equally spaced numbers from *`start`* to *`end`*. If you do not provide a third argument, `linspace` creates a vector of 100 equally spaced numbers. When you ask for fewer than two values in the vector (that is, where $N < 2$), `linspace` returns the value *`end`* (the value in the second input argument).

Similarly, `logspace(`*`start`*`,`*`end`*`,`*`N`*`)` creates a vector of N numbers placed equally along the logarithmic scale from 10^{start} to 10^{end}.

```
x = logspace(0,1,4)
x =
  1  2.154435  4.641589  10
```

Note that if you set *`end`* equal to π, the command creates points between 10^{start} and π, not 10^{π}. If you do not provide a third argument, `logspace` creates a vector of 50 logarithmically equally spaced numbers. For $N < 2$, `logspace` returns 10^{end}.

USING SPECIAL FUNCTIONS TO CREATE ARRAYS

Creating Arrays of Zeros and Ones

Use the commands `zeros` and `ones` to create arrays filled completely with zeros or ones:

```
ones(2,1,2)
ans(:,:,1) =
  1
  1

ans(:,:,2) =
  1
  1
```

Each input argument provides the number of elements for that dimension of the array. The output for a three-dimensional array provides the first two dimensions as matrix "pages," one for each of the elements in the third dimension. Providing just one input argument creates a square matrix (a 2D array):

```
zeros(3)
ans =
  0   0   0
  0   0   0
  0   0   0
```

In other words, `zeros(3)` is equal to `zeros(3,3)`. Using `zeros` or `ones` with no input arguments creates a scalar of value 0 or 1, respectively.

Creating Identity Matrices

The command `eye(N)` creates an N-by-N *identity matrix*, that is, a matrix with ones on the diagonal and with all off-diagonal elements set to zero. It is also possible to use `eye` for an N-by-M rectangular matrix: `eye(N,M)`.

Creating Random Numbers

To create an array of random numbers distributed uniformly across the interval 0 to 1, use `rand`, where the first argument is the seed and the second is the number of values to generate:

```
rand(1,5)
ans =
  0.317308  0.913270  0.375324  0.225387  0.528039
```

To create an array of random numbers between from a normal distribution with a mean value of 0 and a variance (and standard deviation) of 1, use `randn`:

```
randn(1,5)
ans =
  0.749796  0.626937  -2.132375  -0.467405  0.308421
```

Both `rand` and `randn` produce pseudorandom numbers. See the *COMSOL Script Command Reference* entries on `rand` and `randn` for information about determining and changing the state of the random-number generator.

Array Sizes and Empty Matrices

Some 2D matrices are special cases and represent scalars, vectors, and even empty matrices.

VECTORS

Vectors are 1-by-n arrays (row vectors) or n-by-1 arrays (column vectors). The command `rv = rand(1,5)` creates a random row vector of length 5, whereas `v = ones(100,1)` creates a column vector of 100 ones. To check the length of a vector, use the `length` function:

```
length(v)

ans =

100
```

For arrays that are not vectors, `length` returns the maximum length of any dimension. To test if an array is a vector, use the `isvector` function.

SCALARS

Scalar numbers are 1-by-1 arrays. `p = pi` is an example of a statement that creates a scalar variable `p`. To test if an array is a scalar, use the `isscalar` function.

THE EMPTY MATRIX

An empty matrix is an array that in any dimension has the size 0. To create an empty 0-by-0 matrix, use `[]`:

```
e = [];
```

To check if an array is empty, use the `isempty` function.

Empty matrices of size n-by-0 or 0-by-m are useful in many applications of linear algebra. You can create such empty matrices directly using one of the special functions for creating arrays, for example, `zeros`:

```
ze = zeros(0,5);
```

creates a 0-by-5 empty array.

One way to use of the 0-by-0 empty matrix `[]` is to remove parts of an array:

```
A = rand(5);
A(1:3,:) = [];
```

removes the first three rows in `A`.

COMPLEX ARRAYS VS. DOUBLE ARRAYS

COMSOL Script stores the real and imaginary parts of complex-valued data and identifies these variables as complex arrays. A complex array requires twice the memory as a double array. The output from a computation becomes a double array or a complex array depending on if the result is complex or not. You can use complex arrays and double arrays without having to differentiate between them. For example, `a = sqrt(-1)` results in a complex array, whereas `b = imag(a)` results in a double array.

To check if a variable is a complex array or a double array, use the `isreal` function, which is True for double arrays but not for complex arrays. To explicitly create a complex or double array, use the `complex` and `double` commands:

```
a = complex(1,0)

a =

1

isreal(a)

ans =

false

b = double(a)

b=

1

isreal(a)

ans =

false
```

Sparse Matrices

Sparse matrices provide an efficient way to store data in a matrix that contains only a few nonzero elements. Instead of storing the entire array, you can store just those elements and their locations. Sparse matrices often appear in computational physics when solving large systems of linear equations, an example being system matrices such as the stiffness matrix from a finite-element analysis. When working with COMSOL Multiphysics, the `assemble` function provides such sparse matrices as output data.

Sparse matrices make it possible to store and process matrices that would be too large to handle if the software had to deal with them as full matrices. Special versions of the matrix-computational tools work with sparse matrices and produce sparse matrices as their output. The computations that take advantage of sparsity can also execute faster.

One issue with efficient algorithms for sparse matrices concerns *fill-in*, a reduction of the sparsity during the execution of an algorithm when zero elements change to nonzero elements. Sparse algorithms try to minimize fill-in by, for example, switching rows and columns. In many applications, the sparse matrix is a *band matrix*, and the *bandwidth* is an indication of the matrix sparsity.

Even if a set of data is sparse, COMSOL Script does not take advantage of that fact unless you store the data as a sparse matrix. To see the difference in performance and storage requirements, consider two 1000-by-1000 identity matrices:

```
s1 = speye(1000);
s2 = eye(1000);
whos
     Name     Size         Memory      Type

     s1       1000x1000    16000       sparse double array
     s2       1000x1000    8000000     double array

tic, s1*s1; toc

Elapsed time:  0.0 s

tic, s2*s2; toc

Elapsed time:  1.062 s
```

In this case, storing the sparse matrix requires only 0.2% of the memory needed to store the full matrix. Also, multiplying the sparse matrix with itself is much faster than doing the same thing with the full matrix.

The *density* of a sparse matrix is the ratio of the number of nonzero elements to the total number of elements, which in this case is 0.001. Low density for a matrix is a good indication that you should treat it as a sparse matrix.

SPARSE MATRIX FUNCTIONS

The following functions are available for working with sparse matrices:

TABLE 3-1: SPARSE-ALGORITHM FUNCTIONS

FUNCTION NAME	DESCRIPTION
`eigs`	Compute a few eigenvalues of a sparse matrix
`full`	Convert a matrix from sparse to full
`issparse`	True for a sparse matrix
`nnz`	Number of nonzero elements

TABLE 3-1: SPARSE-ALGORITHM FUNCTIONS

FUNCTION NAME	DESCRIPTION
`nzmax`	Number of nonzero elements for which space is allocated
`sparse`	Create a sparse matrix
`spdiags`	Sparse diagonal or band matrix manipulation
`speye`	Create a sparse identity matrix
`spones`	Sparse matrix of ones
`sprand`	Sparse random matrix with uniformly distributed numbers
`sprandn`	Sparse random matrix with normally distributed numbers
`sprandsym`	Symmetric sparse random matrix with normally distributed numbers

Note: The `sparse` data type is available only as sparse matrices, that is, 2D matrices or vectors of real or complex data.

CREATING SPARSE MATRICES

To create sparse matrices use the `sparse`, `spdiags`, `speye`, `spones`, `sprand`, `sprandn`, and `sprandsym` commands.

The `sparse` command works in several ways:

- It converts a standard double matrix into a sparse matrix:

```
s1 = eye(10);
s2 = sparse(s1);
```

- It creates an *mxn* all-zero sparse matrix:

```
s = sparse(m,n);
```

- You can provide the `sparse` function with the matrix size, row and column index vectors for the location of the nonzero data, plus the data itself.

As an example of creating a sparse matrix in this fashion, consider solving the following equation system:

$$\begin{bmatrix} 10 & 1 & 0 & 0 & 0 & -1 & 0 \\ 1 & 10 & 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 10 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 10 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 10 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 & 10 & 1 \\ 0 & -1 & 0 & 0 & 0 & 1 & 10 \end{bmatrix} x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{bmatrix} \tag{3-1}$$

This is a relatively sparse symmetric band matrix, which could represent the numeric solution to some kind of differential equation. To create the 7-by-7 matrix as a sparse matrix, first make sparse matrices from the diagonal and each subdiagonal, then add them together:

```
D = sparse(1:7,1:7,10*ones(1,7),7,7);
S1 = sparse(2:7,1:6,ones(1,6),7,7);
S2 = sparse(6:7,1:2,-ones(1,2),7,7);
S = S2+S1+D+S1'+S2';
```

You can use the `spdiags` command to extract diagonals from a sparse matrix or to create a sparse matrix from diagonals. When you specify the indices of the diagonals, 0 indicates the diagonal, –1 indicates the first subdiagonal, 1 indicates the first superdiagonal, and so on. For example,

```
S = spdiags(C,D,A);
```

returns a sparse copy of a matrix `A` with diagonals `D` replaced by the columns in `C`. If

```
A = [1 2 3; 4 5 6; 7 8 9]
D = [-1 0];
C = [10 0; -5 12; 0 0];
S = spdiags(C,D,A);

full(S)

ans =

    0   2   3
   10  12   6
    7  -5   0
```

`S` is a sparse matrix, but this example displays it as a full matrix to show how spdiags modifies the original matrix `A`. The call to spdiags replaces the first subdiagonal with `[10, -5]` and the main diagonal with `[0, 12, 0]`. The column in `C` is longer than the subdiagonal, and in this case spdiags uses the upper part of the column in `C`; for superdiagonals, elements in the resulting sparse matrix contains the lower part of the column.

See the command-line help and C*OMSOL Script Command Reference* for a full description of available syntaxes for the `spdiags` command.

DISPLAYING THE SPARSITY PATTERN

To plot the sparsity pattern, use the `spy` function:

```
spy(S)
```

produces the following plot for the matrix `S` from the previous example:

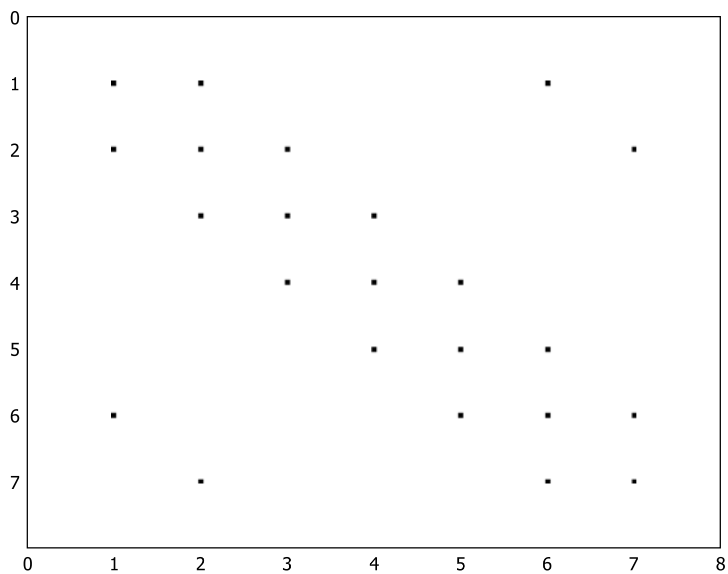


Figure 3-1: Sparsity patterns for a 7-by-7 sparse matrix.

WORKING WITH SPARSE MATRICES

For sparse matrices, COMSOL Script supports all of its matrix arithmetic, logical, and indexing functions as described in the following section. For example, to solve Equation 3-1, use the “backslash” or \ operator (see “Division” on page 44):

```
R = (1:7)';
x = S\R
x =

    0.127368
```

```
0.230168
0.243581
0.334021
0.416213
0.503852
0.672632
```

(The single string quote, `'`, is a transpose operator that converts `R` from a row vector to a column vector; see "Matrix Transposition" on page 40 for more information.)

Using the FIND Command

The `find` command, which finds nonzero entries, is very useful in connection with sparse matrices. As an example, revisit the matrix `S` from the previous example:

```
[i,j,v] = find(S);
[n,m] = size(S);
S = sparse(i,j,v,n,m);
```

The output from `find`, regardless whether it operates on a sparse or a full matrix, is in the same format that you use to create a sparse matrix.

Output From Operations With Sparse Matrices

As you saw from the previous example, solving a system of linear equations where the system matrix is a sparse matrix results in a full solution vector. On the other hand, multiplying a sparse matrix with itself results in another sparse matrix. The following rules apply for the output from operations with sparse matrices:

- If the function's output is a scalar or a vector, the output is always a full scalar or vector (for example, `size`).
- Indexing into a sparse matrix always produces a sparse matrix, except when you index into a single element to output a scalar. In that case, the output is a full scalar.
- Operations with mixed operands produce results that are full matrices, except for elementwise multiplication of a sparse array with a full array (`A.*B`, where `A` is sparse and `B` is full), which provides a sparse array.

To explicitly convert from a sparse matrix to a full matrix or vice versa, use the `sparse` and `full` functions.

Logical Arrays

The results from logical and relational operators are all logical arrays. To test if an array is logical, use the `islogical` function:

```
A = [0 0.6; 0.1 0.3];
B = (A>.5)
```

```
B =

  false  true
  false  false

islogical(A)

ans =

false

islogical(B)

ans =

true
```

To make an array logical, use the `logical` command. All nonzero values then become True, while zeros become False.

To convert a logical array to a double array, use the `double` command. If you use a logical array with arithmetic operators or functions, however, COMSOL Script interprets it as a double array with ones (where the elements are True) and zeros (where the elements are False):

```
logical(A)

ans =

  false  true
  true   true

double(B)

ans =

  0  1
  0  0

A+B

ans =

  0         1.600000
  0.100000  0.300000
```

To set individual elements in a logical array, use the `true` and `false` commands:

```
B(1,1) = true;
```

Working with Matrices and Arrays

COMSOL Script provides many vector and matrix operations but also supports a powerful, efficient form of coding that performs element-by-element operations on each element in a double array. When working with a matrix, it is not necessary to write a loop that operates on each element individually. For instance, you simply write `A+1` to add the scalar `1` to every element of the matrix `A`.

Consider another example that performs some trigonometric operations on every element in a matrix:

```
A = 1:10
B = round(sin(pi*A/2))

B =

  1  0  -1  0  1  0  -1  0  1  0
```

These functions make for much more compact and efficient code than writing the equivalent loop:

```
A = 1:10
B = zeros(1,10);
for i=1:10
  B(i) = round(sin(pi*A(i)/2));
end
```

This section starts by describing basic matrix operations such as indexing, inversions and transpositions; it then moves on to show how to perform elementary math and trig functions; it then looks at special functions such as relational operators and set functions.

Although COMSOL Script comes with a wide range of functions and operators, it is not unusual to have special requirements where a new type of function would prove helpful in streamlining your code. For such cases, it is easy to extend the library of mathematical or other functions by writing your own COMSOL Script functions. See “Functions” on page 127 for more information about creating your own functions.

Indexing and Subscripting

To access individual elements in a matrix (or an array of any size), enclose the subscript in parentheses:

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12];
A(2,3)

ans =

7
```

As you can see, `A(2,3)` refers to the element in the second row and the third column. You can also use linear indexing with a single subscript. In the case of a 2D matrix with n_i rows and n_j columns, the element with subscripts i and j gets the linear index

$$(j-1)n_i + i\ ,$$

so you can access the same element using `A(8)`.

To switch between a multidimensional index and a linear-index vector, use the functions `ind2sub` and `sub2ind`:

```
linind = sub2ind([3 4],2,3)

linind =

  8

[subi, subj] = ind2sub([3 4],linind);
```

The last call results in `subi` = 2 and `subj` = 3. The first input argument to both functions provides the size of the array into which you want to index.

You can also use subscripts that are vectors or matrices. For example, `A(1:2,3:4)` is a 2×2 submatrix consisting of the first two rows and the third and fourth columns in `A`. Using the colon alone refers to all the corresponding elements in that dimension (a row or column for a matrix). Thus

```
A(1:2,:)
```

contains the first two rows, and

```
A(:,2)
```

gives all the values in the second column.

Use `end` to refer to the last element in an array. For instance,

```
A(end,:)
```

is the last row in `A`.

You can also use a vector with negative increments to change the order of columns and rows:

```
A(:,end:-1:1)
```

reverses the column order in A:

Using the colon (:) alone in a subscript reshapes an array into a long column vector, as in the following example:

```
A(:)

ans =

     1
     5
     9
     2
     6
    10
     3
     7
    11
     4
     8
    12
```

LOGICAL INDEXING

Logical indexing means that you work with a logical array to index into another array. This acts as a sort of masking operation where the position of the elements that are True in the logical array determine which elements to access. This is a powerful feature, for example, to remove elements with a certain property.

As an example, set to zero all the elements in a matrix that are smaller than a certain threshold value:

```
A = rand(4);
threshold = 0.25;
A(A<threshold) = 0
A =

           0     0.432789            0     0.707649
           0     0.605391     0.804140     0.555403
    0.984947     0.611451     0.798391     0.803094
    0.461474     0.937855     0.264659     0.786913
```

INDEXING INTO A SUBSET OF AN ARRAY

It is possible to reference element in a subset of an array using multiple subscripting:

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12];
A(1:2,1:2)(2,:)(1)
```

```
ans =

5
```

The multiple subscripting `A(1:2,1:2)(2,:)(1)` first references a 2x2 submatrix from A (with the elements `[1 2; 5 6]`), then references the second row in that submatrix (`[5 6]`, and finally picks the first element in that row, `5`. This can sometimes be useful to avoid creating variables to store the intermediate results.

Building Arrays From Other Arrays

You can build larger arrays and matrices by concatenating arrays and matrices using brackets:

```
H = [A, A, ones(3,8)];
```

This concatenates the matrices horizontally, forming a 3x16 matrix. To concatenate the same matrices vertically, type:

```
V = [A; A; ones(6,4)];
```

This creates a 12x4 matrix. Notice that the array sizes must be consistent: in the horizontal concatenation, all matrices must have the same number of rows (3); in the vertical concatenation, the same holds true for the number of columns. You can extend an array in all directions as long as the array sizes match.

There are also functions that perform the equivalent array concatenations. Use `cat` to concatenate along any dimension (providing the dimension as the first input argument). `horzcat` and `vertcat` concatenates arrays horizontally and vertically, respectively.

```
H = cat(2,A,A,ones(3,8));
H = horzcat(A,A,ones(3,8));
```

both create the same matrix `H` as the previous example using brackets. Likewise:

```
V = cat(1,A,A,ones(6,4));
V = vertcat(A,A,ones(6,4));
```

both create the same matrix `V` as the previous example using brackets.

General Functions for Array Sizes

The function `size` returns the size of all dimensions of an array:

```
A = rand(5,9);
```

```
[nrow,ncol] = size(A)

nrow =

5

ncol =

9
```

The function `numel` returns the number of element in the array:

```
numel(A)

ans =

45
```

`numel(A)` is the same as `prod(size(A))`.

The function `ndims` returns the number of dimension in the array:

```
ndims(A)

ans =

2
```

`ndims(A)` is the same as `length(size(A))`.

SUMMARY OF FUNCTIONS FOR SIZES AND TYPES OF ARRAYS

The following list summarizes functions for getting the dimension of an array or for checking which type of array it is:

TABLE 3-2: ARRAY SIZE AND TYPE FUNCTIONS

FUNCTION NAME	DESCRIPTION
`isempty`	Check for empty arrays
`isscalar`	Check if a variable is a scalar
`isvector`	Check if a variable is a vector
`length`	Largest dimension of an array
`ndims`	Number of dimensions of an array
`numel`	Number of elements in an array
`size`	Size of an array

Matrix Transposition

To transpose a matrix, use the `'` (straight single quote) operator:

```
A = [1 2; 3 4];
A'
ans =
  1  3
  2  4
```

You can also transpose a vector:

```
A = [1 0 2]
A'
ans =
  1
  0
  2
```

There is no transpose operator for arrays of dimension larger than two.

TRANSPOSITION OF COMPLEX-VALUED DATA

The standard transpose operator `'` performs a *Hermitian transpose* (or *conjugate transpose*), where the transpose contains the complex conjugate:

```
C = [1+i 2-i;3-i 4+3i]
C =
  1+i  2-i
  3-i  4+3i

C'
ans =
  1-i  3+i
  2+i  4-3i
```

To get a transposition that is not conjugated, use the *nonconjugate transpose* operator, `.'` (period-straight quote):

```
C.'
ans =
  1+i  3-1
  2-i  4+3i
```

To achieve the same result, you can also use the function `conj`, which takes the complex conjugate:

```
conj(C')
```

Matrix Inversion

To compute the inverse of a matrix, use the `inv` command:

```
A = [1 0 0; 0 2 0; 0 0 3];
inv(A)
ans =
  1        0        0
  0  0.50000        0
  0        0  0.333333
```

If the matrix is *singular*, that is, it does not have full *rank* (all rows or columns are linearly independent), the `inv` command issues a warning for a singular matrix. To check for a matrix' rank (the number of linearly independent rows or columns), use the `rank` command:

```
rank(A)
ans =
  3
rank(zeros(3))
ans =
  0
```

The `inv` command works only for square matrices. For nonsquare matrices, the `pinv` command computes a *pseudoinverse* of a matrix A, which is a matrix A^+ such that $AA^+A = A$. A^+ has the following properties:

- A^+ has the same format as the transpose of A.
- $AA^+A = A$
- $A^+AA^+= A^+$
- AA^+ and A^+A are Hermitian matrices.

Consider the following example:

```
pinv(A)
ans =
  1        0        0
  0  0.50000        0
  0        0  0.333333
a = [10i 5 0]
b = pinv(a)
ans =
     0.800000  0-0.400000i    0
  0+0.400000i     0.200000    0
            0            0    0
```

Another command, `pinv(x,tol)`, uses the tolerance in *tol* to compute the pseudoinverse.

Elementary and Special Math Functions

You can use standard arithmetic operators (addition, subtraction, multiplication, division, and power) on large classes of 1D and 2D arrays (vectors and matrices). Only addition and subtraction are well-defined operators for arrays of any dimension. For all arrays, special element-by-element operators perform multiplication, division, and power operations on each element in an array (addition and subtraction do not differ between matrix operators and element-by-element operators).

Note: Additional information about logical and relational operators and math functions that work with double arrays appears in the *COMSOL Script Command Reference*.

As noted earlier, COMSOL Script provides a large set of math functions, all of which work on arrays in an elementwise fashion so you can always work in a "vectorized" way. For more information about specific functions, see the *COMSOL Script Command Reference* or type `help` followed by the function name at the command prompt.

ADDITION AND SUBTRACTION

Use the standard operators + and – to add or subtract arrays of the same dimension:

```
A = [5 6; 7 8]; B = [1 2; 3 4];
C = A+B
C =
   6   8
  10  12
C = A-B
C =
  4  4
  4  4
```

As a special case, you can add or subtract a scalar to a matrix or multidimensional array. COMSOL Script then adds this value to or subtracts this values from every element in the other operand:

```
A+1
ans =
  6  7
  8  9
B-1
ans =
  0  1
  2  3
```

MULTIPLICATION

Use the `*` operator for *matrix multiplication*. Matrix multiplication requires that the "inner dimension" of the matrices or vectors agree. An *inner product* (*dot product* or *scalar product*) of two vector produces a scalar number:

```
x = 1:5;
y = [1 0 2 0 3];
x*y'
ans =
  22
```

Note that `x*y` is not a valid matrix multiplication (you cannot multiply two row vectors) and results in an error message. There are two *outer products*, each being the transpose of the other:

```
x'*y
ans =
   1    0    2    0    3
   2    0    4    0    6
   3    0    6    0    9
   4    0    8    0   12
   5    0   10    0   15
y'*x
ans =
   1    2    3    4    5
   0    0    0    0    0
   2    4    6    8   10
   0    0    0    0    0
   3    6    9   12   15
```

COMSOL Script supplies a special function for inner products ($x \cdot y$) named `dot`, and a function for *vector cross products* ($x \times y$) named `cross`:

```
dot(x,y)
ans =
  22
a=[1 0 0];b=[0 1 0];
cross(a,b)
ans =
  0  0  1
```

You can also calculate *matrix-vector products* of the type $y = Ax$:

```
A=[1 2 3; 4 5 6; 7 8 9]; x = [1 0 1]';
y = A*x
y =
   4
  10
  16
```

For element-by-element multiplication or pointwise multiplication of two arrays with equal dimensions, use the `.*` operator:

```
A*A'
ans =
   14    32    50
   32    77   122
   50   122   194
A.*A'
ans =
    1    8   21
    8   25   48
   21   48   81
```

Multiplying an array by a scalar is equivalent to element-by-element multiplication with an array where each element contains that scalar:

```
A*2
ans =
    2    4    6
    8   10   12
   14   16   18
A.*[2 2 2; 2 2 2; 2 2 2]
ans =
    2    4    6
    8   10   12
   14   16   18
```

DIVISION

COMSOL Script provides two matrix-division operators or linear equation system solvers: `/` (right matrix divide, or "slash") and `\` (left matrix divide, or "backslash").

What is the difference? For a nonsingular matrix C and another matrix B, the right-matrix division `B/C` is equivalent to `B*inv(C)`, but it is computed without explicit matrix inversion. In the same way, the left-matrix division `C\B` is equivalent to `inv(C)*B`. In fact, the right-matrix division relies on the left-matrix division in that `B/C = (C'\B')'`. In general, it is also true that `X = C\B` is a solution to `C*X = B`, and that `X = B/C` is a solution to `X*C = B`.

If C is not square, the result from a matrix division is a solution in the least-squares sense to an overdetermined or underdetermined system of equations. For a left-matrix division (`C\B`), the number of rows in C and B must be the same. The result, X, is an mxn matrix, where m is the number of columns in C, and n is the number of columns in B.

For example, solve the equation system

$$\begin{cases} x_1 + 2x_2 = 0 \\ 3x_1 + 4x_2 = 1 \end{cases}$$

using the \ operator:

```
A = [1 2; 3 4];
B = [0; 1];
x = A\B

x =

   1.000000
  -0.500000
```

The element-by-element or pointwise division operators, `./` and `.\`, divide the elements in the left matrix or array with the elements in the right matrix or array, and vice versa:

```
x = [1 2 3];
y = [2 2 2];
x./y
ans =
  0.500000  1  1.500000
x.\y
ans =
  2  1  1.666667
```

Note that `x` and `y` must have the same dimension.

POWER

A matrix power X^y exists if X is a square matrix and y is a scalar. If y is a positive integer, it is possible to compute X^y using repeated matrix multiplication. To compute the matrix power for positive integer powers, use the ^ operator:

```
A = rand(3);
B = A^10;
```

This is the same as:

```
B = A*A*A*A*A*A*A*A*A*A;
```

The element-by-element or pointwise power operator `.^` works on elements in arrays and matrices:

- `z = x.^y` when `x` and `y` have the same dimension. The result for element z_{ij} is $x_{ij}^{y_{ij}}$.

- `z = x.^y` when the base `x` is a scalar. The result for element z_{ij} is then $x^{y_{ij}}$.
- `z = x.^y` when the exponent `y` is a scalar. The result for element z_{ij} is then $x_{ij}{}^{y}$.

SQUARE ROOTS

The function `sqrt` computes the elementwise square root for each element in an array. For the square root of a matrix, use the `sqrtm` function, which computes the principal square root X for the matrix A, that is, $X^2 = A$.

EQUIVALENT FUNCTIONAL FORM OF ARITHMETIC OPERATORS

All arithmetic operators have an equivalent function-based form. Thus, for example, you can write

```
C = plus(A,B);
```

which is equivalent to

```
C = A+B;
```

The following list shows the function-based form of all arithmetic and transpose operators:

TABLE 3-3: FUNCTIONAL FORM OF ARITHMETIC OPERATORS

OPERATOR	STANDARD FORM	FUNCTION-BASED FORM
Sum	`C = A+B`	`C = plus(A,B)`
Unary plus	`+A`	`C = uplus(A)`
Minus	`C = A–B`	`C = minus(A,B)`
Unary minus	`–A`	`C = uminus(A)`
Multiplication (pointwise)	`C = A.*B`	`C = times(A,B)`
Matrix multiplication	`C = A*B`	`C = mtimes(A,B)`
Left division (pointwise)	`C = A.\B`	`C = ldivide(A,B)`
Left division (pointwise)	`C = A.\B`	`C = ldivide(A,B)`
Right division (pointwise)	`C = A./B`	`C = rdivide(A,B)`
Right division (equation system solver)	`C = A/B`	`C = mrdivide(A,B)`
Power (pointwise)	`C = A.^B`	`C = power(A,B)`
Matrix power	`C = A^B`	`C = mpower(A,B)`

TABLE 3-3: FUNCTIONAL FORM OF ARITHMETIC OPERATORS

OPERATOR	STANDARD FORM	FUNCTION-BASED FORM
Matrix transpose	`C = A.'`	`C = transpose(A)`
Matrix complex conjugate transpose	`C = A'`	`C = ctranspose(A)`

ELEMENTARY MATH FUNCTIONS

Beyond the basic operators, COMSOL Script provides the following elementary math functions:

TABLE 3-4: ELEMENTARY MATH FUNCTIONS

FUNCTION NAME	FUNCTION DESCRIPTION
`abs`	Absolute value/complex magnitude
`angle`	Phase angle
`sqrt`	Square root
`realsqrt`	Square root of nonnegative real array
`real`	Real part
`imag`	Imaginary part
`complex`	Create complex array
`conj`	Complex conjugate
`isreal`	True for double, character, and logical arrays
`round`	Round to the nearest integer
`ceil`	Round to the nearest larger integer
`floor`	Round to the nearest smaller integer
`fix`	Round toward zero
`sign`	Signum function
`mod`	Modulus of arrays
`rem`	Remainder of modulus
`exp`	Exponential base e
`exp2`	Base-2 power
`realpow`	Power of a real matrix
`log`	Natural logarithm
`log10`	Logarithm base 10
`reallog`	Natural logarithm of nonnegative real number
`unwrap`	Remove phase jumps

TRIGONOMETRIC AND HYPERBOLIC FUNCTIONS

Available trigonometric functions and hyperbolic functions in COMSOL Script include:

TABLE 3-5: TRIGONOMETRIC FUNCTIONS

FUNCTION NAME	TRIGONOMETRIC FUNCTION
`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`cot`	Cotangent
`sec`	Secant
`csc`	Cosecant
`asin`	Inverse sine (arc sine)
`acos`	Inverse cosine (arc cosine)
`atan`	Inverse tangent (arc tangent)
`atan2`	Binary (four-quadrant) arc tangent
`acot`	Inverse cotangent (arc cotangent)
`asec`	Inverse secant
`acsc`	Inverse cosecant
`sinh`	Hyperbolic sine
`cosh`	Hyperbolic cosine
`tanh`	Hyperbolic tangent
`coth`	Hyperbolic cotangent
`sech`	Hyperbolic secant
`csch`	Hyperbolic cosecant
`asinh`	Inverse hyperbolic sine (hyperbolic arc sine)
`acosh`	Inverse hyperbolic cosine (hyperbolic arc cosine)
`atanh`	Inverse hyperbolic tangent (hyperbolic arc tangent)
`acoth`	Inverse hyperbolic cotangent (hyperbolic arc cotangent)
`asech`	Inverse hyperbolic secant
`acsch`	Inverse hyperbolic cosecant

The `atan2` function takes two input arguments; `atan2(Y,X)` computes the pointwise inverse tangent of the two matrices `X` and `Y`. For scalars, `atan2(y,x)` is the angle α such that $\tan(\alpha) = y/x$. All trigonometric function assume that the input angles are in radians.

SPECIAL MATH FUNCTIONS

COMSOL Script comes with the following special math functions that provide more advanced capabilities:

TABLE 3-6: SPECIAL MATH FUNCTIONS

FUNCTION NAME	FUNCTION DESCRIPTION
`airy`	Airy functions
`bessel`	Bessel function of the first kind
`besselh`	Bessel function of the third kind (Hankel function)
`besseli`	Modified Bessel function of the first kind
`besselj`	Bessel function of the first kind
`besselk`	Modified Bessel function of the second kind
`bessely`	Bessel function of the second kind
`beta`	Beta function
`betainc`	Incomplete beta function
`betaln`	Logarithm of beta function
`cart2pol`	Transform from Cartesian to polar coordinates
`cart2sph`	Transform from Cartesian to spherical coordinates
`cross`	Cross product
`dot`	Scalar product (dot product)
`erf`	Error function
`erfc`	Complementary error function
`erfcx`	Scaled complementary error function
`erfinv`	Inverse error function
`factor`	Prime factors
`factorial`	Factorial function
`gamma`	Gamma function
`gammainc`	Incomplete gamma function
`gammaln`	Logarithm of gamma function
`gcd`	Greatest common divisor
`isprime`	True for prime numbers
`lcm`	Least common multiple
`pol2cart`	Transform from polar to Cartesian coordinates
`primes`	Generate prime numbers

TABLE 3-6: SPECIAL MATH FUNCTIONS

FUNCTION NAME	FUNCTION DESCRIPTION
`psi`	Psi function (digamma function)
`rat`	Rational fraction approximation
`rats`	String representation of rational fraction approximation
`sph2cart`	Transform from spherical to Cartesian coordinates

Relational and Logical Operators and Functions

RELATIONAL OPERATORS AND THE FIND COMMAND

The following relational operators are available for comparing the contents of two arrays of equal size:

RELATIONAL OPERATOR	FUNCTION-BASED FORM	MEANING
`<`	`C = lt(A,B)`	Less than
`<=`	`C = le(A,B)`	Less than or equal
`>`	`C = gt(A,B)`	Greater than
`>=`	`C = ge(A,B)`	Greater than or equal
`==`	`C = eq(A,B)`	Equal
`~=`	`C = ne(A,B)`	Not equal

These operators compare each element in the first array with the corresponding element in the second array, producing a *logical array* of the same size, where the elements are True if the relation holds and False otherwise:

```
A=[1 2; 3 4];
B=[1 1; 3 3];
A>B

ans =

  false   true
  false   true
```

You can also compare an array to a scalar number:

```
A==1

ans =

   true   false
  false   false
```

Notice the difference between the == operator, which tests for equality, and the assignment operator =.

Using the FIND Command

It is often interesting to use the `find` command in combination with relational operators to determine the indices of certain elements in an array. To find the element in `A` that equals 3, type:

```
i = find(A==3)
```

This returns 2 in the variable `i`, because `A(2)` is 3 (the single subscript, or linear index, runs top-down from the first column to the last column). To convert the subscript 2 to row-and-column indices, use the `ind2sub` function:

```
[ii,jj] = ind2sub(size(A),i)
```

which returns `ii = 2` for row 2 and `jj = 1` for column 1. To go from row-and-column subscripts to single subscripts, use the `sub2ind` function.

If you instead want the row and column indexes directly from the call to `find`, use

```
[i,j] = find(A==3)
```

Finally, you can also get the nonzero values of a matrix (defined by the corresponding positions in the row and column indexes) in the third output variable:

```
[i,j,val] = ind(A)
```

If you want to replace the elements in `A` that are greater than 1 with zeros, type:

```
i = find(A>1);
A(i)=0;
```

This last statement works because of *scalar expansion*, that is, COMSOL Script assigns a scalar to all indices i in the matrix A.

Relational Operators and NaNs (Not-A-Numbers) and Nonfinite Numbers

All relational operators return `false` when comparing something to NaNs, for example:

```
A = ones(2);
A(1,1) = NaN;
A < NaN

ans =

  false  false
  false  false
```

So, to locate all elements in an array that are NaNs, you should instead use the special command `isnan`, which returns True in any such case:

```
isnan(A)

ans =

  true   false
  false  false
```

Next suppose you want to replace all occurrences of NaN with zeros. For this task, type:

```
A(isnan(A)) = 0;
```

(See also "Handling NaNs and Missing Data" on page 145.)

If any elements of the array are infinite (∞ or $-\infty$), you can make comparisons such as:

```
A(1,1) = inf;
A==inf

ans =

  true   false
  false  false
```

The command `isinf` returns `true` for all elements in an array that are `Inf` or `-Inf`. The command `isfinite` returns `true` for any element that is not `NaN`, `Inf`, or `-Inf`:

```
A(2,2) = NaN;
isfinite(A)

ans =

  false  true
  true   false
```

Comparing Entire Arrays

You can also compare two values or two arrays to check if they are equal using the `isequal` command. For two arrays, it returns True if and only if the sizes are the same and all elements are the same; `isequal(NaN,NaN)`, however, returns False. To compare two arrays and consider them equal if they have NaNs in the same elements, use the `isequalwithequalnans` command. Continuing with the same matrix `A` as earlier in this section:

```
B = A;
isequal(A,B)
```

```
ans =
  false

isequalwithequalnans(A,B)
ans =
  true
```

Note: The `isequal` and `isequalwithequalnans` functions also work for other data types such as cell arrays and structures.

LOGICAL OPERATORS

Logical operators in COMSOL Script are available both as element-wise operators for arrays and bitwise operators.

Element-wise Logical Operators

The following logical operator operate pointwise on arrays:

TABLE 3-7: LOGICAL OPERATORS

LOGICAL OPERATOR	FUNCTION-BASED FORM	MEANING
`|`	`C = or(A,B)`	Logical OR
`&`	`C = and(A,B)`	Logical AND
`Not available`	`C = xor(A,B)`	Logical XOR
`~`	`C = not(A)`	Not, logical complement

The logical complement, `not`, is a unary operator. `~A` returns True where A is False (or Zero), and it returns False where A is True (or nonzero).

In addition, consider the scalar AND and OR operators, `&&` and `||`. The sequence `a&&b` returns True if `a` and `b` are scalars that both have the value True (nonzero); otherwise it returns False. Further, `a||b` returns True if either `a` or `b` have the value True (nonzero); otherwise it returns False. In both cases, these are "short-circuited" operators, which means that COMSOL Script does not evaluate the second input argument (`b`) unless it is necessary, that is, `a&&b` returns False without evaluating `b` if `a` is already False, and `a||b` returns True without evaluating `b` if `a` is already True.

In addition, you can work with bitwise logical operators that operate on nonnegative integer arrays. The following table lists the bitwise logical operators:

TABLE 3-8: BITWISE LOGICAL OPERATORS

BITWISE LOGICAL OPERATOR	MEANING
`C = bitand(A,B)`	Bitwise AND
`C = bitor(A,B)`	Bitwise OR
`C = bitxor(A,B)`	Bitwise XOR
`C = bitcmp(A,B)`	Bitwise complement
`C = bitget(A,POS)`	Extracts values of the bits in position POS in A
`C = bitset(A,POS)`	Returns A with bits in position POS set
`C = bitset(A,POS,VAL)`	Returns A with bits in position POS set to VAL
`C = bitshift(A,SHIFT)`	Shifts the bits in A by SHIFT steps
`C = bitshift(A,SHIFT,NDIG)`	Shifts the bits in A by SHIFT steps and then zeroes out all bits with positions larger than NDIG
`C = bitmax`	Largest integer for use with bitwise functions (2^53-1)

The position entries for the bitwise functions must be integers in the range of 1 to 53. The least significant bit takes position 1 and the most significant bit takes position 53.

As an example, consider `b1 = 5` and `b2 = 3`. In binary form, `b1 = 101` and `b2 = 011`. Then

```
c1 = bitor(b1,b2)
c1 =

7
c2 = bitxor(b1,b2)
c2 =

6
c3 = bitset(b1,3,0)
c1 =

1
```

THE ANY AND ALL FUNCTIONS

Several additional relational and logical functions prove useful when working with logical operators.

The any function determines if any element along a specific dimension is nonzero. any(X) returns True if any element in X is nonzero. For a matrix X, the result of any(X) is a row vector where each element is True or False depending on whether or not any elements of the corresponding column of X are nonzero. To get a scalar value, write any(any(X)).

For the case when X is an array of a dimension higher than two, any(X) tests for nonzero elements along the first nonsingleton dimension of X. You can also specify a dimension along which any looks for nonzero elements; specifically, Y = any(X,dim) tests X for nonzero elements along the dimension dim. For example, if a program requires that all elements of a vector X be positive, the following code uses any to check for this and issue an error message:

```
if any(X<=0)
  error('All elements in X must be positive.')
end
```

The all function works in the same way but determines if *all* the elements along a specific dimension are nonzero.

SUMMARY OF LOGICAL AND RELATIONAL FUNCTIONS

The following table provides an overview of the logical and relational functions that this section covers (in addition to the operators). Additional logical functions deal with sizes (see "Summary of Functions for Sizes and Types of Arrays" on page 39) and check for data types (see the following sections on other data types).

TABLE 3-9: SUMMARY OF LOGICAL AND RELATIONAL FUNCTIONS

FUNCTION NAME	DESCRIPTION
any	True if any elements are nonzero
all	True if all elements are nonzero
find	Find indices of nonzero values
isequal	True if values are equal
isequalwithequalnans	True if values are equal (including NaNs)
islogical	True if array is logical
isfinite	True if elements are finite
isinf	True if elements are infinite
isnan	True if elements are NaNs

Special Functions for Modifying Arrays

COMSOL Script provides a number of functions for modifying and reshaping arrays and for creating special arrays from other arrays. The following table provides an overview of these functions:

TABLE 3-10: FUNCTIONS FOR MODIFYING AND CREATING SPECIAL ARRAYS

FUNCTION NAME	DESCRIPTION
`blkdiag`	Create a block-diagonal matrix (or cell array)
`circshift`	Shift the indices of a matrix circularly
`diag`	Extract diagonal from matrix or create diagonal matrix
`flipdim`	Flip dimension of matrix
`fliplr`	Flip matrix horizontally
`flipud`	Flip matrix vertically
`freqspace`	Create frequency range
`meshgrid`	Create 2D or 3D grid
`repmat`	Create matrix by repeating another matrix in a pattern
`reshape`	Reshape array
`rot90`	Rotate matrix counter-clockwise
`shiftdim`	Shift matrix dimensions
`tril`	Extract elements below the main diagonal of a matrix
`triu`	Extract elements above the main diagonal of a matrix

For example, create a matrix using `repmat`:

```
A = repmat(eye(2),2,3));
```

This creates a 4×6 matrix `A` where half the elements are ones and the other half are zeros.

Then reshape it into a 2×12 matrix:

```
B = reshape(A,2,12);
```

The resulting matrix has the same column-major order contents as the input (that is, `A(:)` and `B(:)` are identical).

In addition to using repmat, you can duplicate a vector using direct indexing:

```
v = [1; 2; 3; 4];
v(:,ones(1,4))

ans =
```

```
1   1   1   1
2   2   2   2
3   3   3   3
4   4   4   4
```

This works by providing a row vector of ones, referencing the one column in `v` multiple times.

USING MESHGRID TO GENERATE A GRID

The `meshgrid` function is useful for creating an equally spaced grid that is useful for plotting a function that depends on x and y (or x, y, and z). `meshgrid` creates both 2D and 3D grids from input vectors. Type

```
[x,y] = meshgrid(1:2:10,0:0.1:1);
```

to create a 2D grid defined by the x-range vector and the y-range vector in the input.The resulting matrices are both of size 11x5, that is, the length of the y-range vector times the length of the x-range vector.

For other gridding and triangulation functions, see “Data Gridding and Triangulation of Point Data” on page 166.

Creating and Using Multidimensional Arrays

Multidimensional arrays are arrays of dimension three or higher. Most elementwise functions and basic array functions work the same with multidimensional arrays. Other functions that work with matrices, such as matrix multiplication, are not defined for multidimensional arrays.

You create multidimensional array in the same way as you create matrices and vectors:

```
r = rand(2,2,3)

r(:,:,1) =

     0.677801     0.987402
     0.151510     0.739669

r(:,:,2) =

     0.585750     0.250217
     0.256429     0.566606

r(:,:,3) =
```

```
    0.097483      0.585205
    0.620439      0.379947
```

COMSOL Script displays multidimensional arrays by showing each two-dimensional "page."

Subscripting also extends into multidimensional arrays:

```
r(2,1,2)

ans =

0.256429
```

You can compute the sine of all elements in r independently of its dimension.

```
s = sin(r);
size(s)

ans =

  2  2  3
```

The output from all elementwise functions has the same size as the input, in this case a 2x2x3 array.

SPECIAL FUNCTIONS FOR WORKING WITH MULTIDIMENSIONAL ARRAYS

Some functions are especially useful when working with multidimensional array. The following table provides an overview of such functions:

TABLE 3-11: FUNCTIONS FOR WORKING WITH MULTIDIMENSIONAL ARRAYS

FUNCTION NAME	DESCRIPTION
`ipermute`	Inverse permute the order of dimensions of a multidimensional array
`ndgrid`	Create multidimensional grid
`permute`	Permute the order of dimensions of a multidimensional array
`squeeze`	Remove unit dimensions from a multidimensional array

The `permute` and `ipermute` functions are equivalents of the transpose operator for matrices:

```
A = ones(2,6,7);
B = permute(A,[3 1 2]);
size(B)

ans =
```

```
    7   2   6

C = ipermute(B,[3 1 2]);
```

Using `ipermute` makes `C` identical to `A`.

The function `squeeze` removes any "singleton" dimension in multidimensional array (demission with size 1):

```
A = rand(2,1,3,1,5);
B = squeeze(A);
size(B)

ans =

    2   3   5
```

Tensor Products and Contractions

The `tprod` function provides the possibility to compute various *tensor products*. The symbol $A \otimes B$ often appears to indicate the product of the tensors A and B. A tensor product represents the most general bilinear operation or "generalized multiplication" for the tensors. The general syntax for `tprod` is

```
C= tprod(A, B, IA, IB);
```

which computes the tensor product of the arrays (tensors) `A` and `B`, optionally followed by contractions and setting some indices equal. The vectors `IA` and `IB` describe the mapping from input indices to output indices as well as how to perform the contractions. To show how this works, the following examples uses matrices:

- `C = tprod(A, B, [1 2], [3 4])` is the tensor (outer) product of the matrices `A` and `B`.
- `C = tprod(A, B, [1 -1], [-1 2])` is the ordinary matrix product of the matrices *`A`* and `B`.
- `C = tprod(A, ONES(SIZE(A)), [-1 -2], [-1 -2])` is the sum of all entries in the matrix `A`.
- `C = tprod(A, EYE(SIZE(A)), [-1 -2], [-1 -2])` is the sum of all diagonal entries in the matrix `A` (the trace).

SPECIFYING THE INDEX VECTORS IA AND IB

The numbers in `IA` and `IB` must be distinct, and if a number occurs both in `IA` and `IB`, the corresponding dimensions in `A` and `B` must have the same size. The summation takes place over index variables with negative numbers. If a negative number occurs in

A, it must also occur in B, and the other way around. You must use the same set of negative numbers in both IA and IB. The tprod function also assumes that the union of the numbers in IA and IB, together with 0, form a contiguous sequence of integers.

4

Data Types for Non-Numeric Values: Strings, Cell Arrays, Structures

Although the core computations you likely perform with COMSOL Script are based on numerical data types, any sophisticated programming language also provides powerful capabilities to handle non-numeric data types such as characters, non-numeric arrays, and structures (for a complete list of data types, see "The COMSOL Script Data Types" on page 22). They can handle tasks for everything from simple purposes such as labeling a plot axis to being as complex as gridded panels in a user-interface window. This chapter introduces you to these various data types and how to work with them. The data structures in COMSOL Multiphysics make extensive use of data types like cell arrays and structures to store data structures and user inputs of various types.

Strings and Character Arrays

Although a string is simply a number of printable characters, symbols, and the control codes used with them, they are extremely useful in a programming language, especially for I/O operations and labeling a graphical user interface. COMSOL Script stores strings in *character arrays* (the data type `char`). The character array is typically a 1D array, where each element represents one character. It is also possible to use character matrices, where each row represents one string. All rows then have equal length, so you must add spaces to the end of shorter strings. A better alternative if you are working with several strings is to use a *cell array of strings*. In such a cell array, each cell contains a string, and each string can be of different sizes.

Creating and Modifying Strings

To create a string, define it by enclosing it in single straight quotes (`'`):

```
s = 'Hello, World!'

s =

  Hello, World!
```

On a string you can use the same indexing and matrix operations as with a numeric vector, as the following examples show:

- Access part of a string:

```
s(1:5)

ans =

  Hello
```

- Modify a string:

```
s(end) = '.'

s =

  Hello, World.
```

- Concatenate a string from several smaller strings:

```
s = ['Hello,', ' ', 'World!']

s =
```

```
    Hello, World!
```

To display a string, use the `disp` command:

```
disp(s)
Hello, World!
```

Because straight quotes or apostrophes enclose a string, you must use two single quotes inside a string if you want that quote to actually appear in the string itself:

```
s='A programmer''s code'

s =

    A programmer's code
```

To create a cell array of strings, use the curly braces to create the cell array and to access its contents in the same way as with other cell arrays:

```
sc = {'This', 'is', 'a', 'cell', 'array','of', 'strings'};
sc{end}

ans =

    strings
```

You can also use the function `cellstr` to convert a character array to a cell array of strings:

```
s = ['Hello ';'World!'];
c = cellstr(s)

c =

    'Hello '
    'World!'
```

Conversely, `char(c)` converts a cell array of strings back to a character array.

Summary of Functions for Converting and Modifying Strings

COMSOL Script provides a number of functions for converting strings to and from different formats and for modifying and checking strings:

TABLE 4-1: STRING FUNCTIONS

FUNCTION NAME	**DESCRIPTION**
`blanks`	Create a string of blanks
`cellstr`	Convert a character matrix to a cell array of strings

TABLE 4-1: STRING FUNCTIONS

FUNCTION NAME	DESCRIPTION
abs	Convert a character matrix to ASCII values
base2dec	Convert strings in a specific base to decimal integers
bin2dec	Convert binary strings to decimal integers
char	Convert a value to a character matrix
deblank	Remove trailing blanks
dec2base	Convert decimal integers to strings in a specific base
dec2bin	Convert decimal integers to binary strings
dec2hex	Convert decimal integers to hexadecimal strings
findstr	Find a shorter string within a longer string
hex2num	Convert IEEE-754 hexadecimal strings to decimal numbers
int2str	Integer-to-string conversion
iscellstr	Test if a variable is a cell array of strings
ischar	Test if variable is a character matrix
isletter	Test for letters in the alphabet
isspace	Test for white spaces (horizontal tab, new line, vertical tab, form feed, carriage return, and space)
isstr	Test if a variable is a character matrix
lower	Convert to lower-case letters
mat2str	Create a string from a value
num2hex	Convert decimal numbers to IEEE-754 hexadecimal strings
num2str	Convert a number to a string
sprintf	Convert data to a formatted string
sscanf	Read formatted date from a string
str2num	Convert a string to a number
strcat	Concatenate strings
strcmp	Compare strings
strcmpi	Compare strings, ignoring case
strfind	Find one string within another
strjust	Justify a character array
strmatch	Find string matches
strncmp	Compare a specific number of characters in two strings

TABLE 4-1: STRING FUNCTIONS

FUNCTION NAME	DESCRIPTION
`strncmpi`	Compare a specific number of characters in two strings, ignoring case
`strrep`	Search and replace strings
`strtok`	Retrieve the first token
`strtrim`	Remove leading and trailing white-space characters
`strvcat`	Concatenate strings vertically
`symvar`	Find identifiers in expression string
`upper`	Convert to upper-case letters

Using String Functions—Some Examples

CONVERTING BETWEEN ASCII CODES AND STRINGS

It is possible to convert a string to the corresponding ASCII codes using the commands `abs` or `double`:

```
s = 'ABC';
a = abs(s)

a =

  65   66   67
```

To go the other way, use `char` to convert ASCII data to a character array:

```
char(a)

ans =

  ABC
```

WORKING WITH STRING FUNCTIONS

Consider some additional useful examples of string functions:

- Combine `strtrim` and `upper` to remove blanks and convert to uppercase letters:

```
upper(strtrim('   comsol  '))

ans =

  COMSOL
```

- Use `strtok` to retrieve information from a comma-separated list:

```
fruits='apple, pear, banana, pineapple, mango, orange';
[fruit, remainder]=strtok(fruits,',')

fruit =

  apple
```

Calling `strtok` again with the remainder as the input retrieves the next item in the list:

```
[fruit, remainder]=strtok(remainder,',')

fruit =

  pear
```

CONVERTING DATA TO STRINGS

The functions `int2str`, `mat2str`, and `num2str` all convert numerical data to strings:

- `int2str` converts an integer to a string (rounding any noninteger value):

```
s = int2str(10.2/5)

s =

  2
```

- `mat2str` creates a string that, when you evaluate it, produces the same values as the input:

```
s = mat2str([linspace(-2,2,5);ones(1,5)])

s =

  [-2, -1, 0, 1, 2; 1, 1, 1, 1, 1]

eval(s)

ans =

  -2   -1   0   1   2
   1    1   1   1   1
```

- `num2str` converts a number to a string with the ability to format the string:

```
s = num2str(pi)

s =

  3.1416

s = num2str(pi,8)
```

```
s =

  3.1415927

s = num2str(pi,'%.9E')

s =

  3.141592654E+000
```

In addition, `base2dec`, `bin2dec`, `hex2dec`, `hex2num`, `dec2base`, `dec2hex`, and `num2hex` all convert between string and integers using different bases:

```
d = base2dec('101',2)

d =

  5

d = base2dec('101',10)

d =

  101
```

Using the SPRINTF Command

For maximum flexibility and control of formatting, use the `sprintf` command (the command `fprintf` works in the same way as `sprintf` but prints to a file). `sprintf` takes a C-style formatting string that can contain conversion specifications for the input data. Each specification begins with the % character followed by optional flags, width and precision fields, and a required conversion character. For example,

```
sprintf('%-+8.5f',pi)
```

prints π using a decimal floating point number with a (minimum) width of 8 characters width and 5 digits after the decimal point. In addition, the number includes the sign and is left justified.

As noted, the first field has the `%` character, which is optionally followed by one of the flags in this table:

TABLE 4-2: SPRINTF/FPRINTF FLAG CHARACTERS

FLAG CHARACTER	DESCRIPTION
-	Result is left justified
+	Always print the sign
0	Pad with Zeros instead of spaces

Note that it is possible to combine flags and have more than one flag at a time.

The width and precision fields typically are of the format `w.f`, where `w` is the width in characters for the data, and `f` is the precision as number of digits after the decimal point.

The `sprintf` and `fprintf` commands also require one of the following conversion characters:

TABLE 4-3: SPRINTF/FPRINTF CONVERSION CHARACTERS

CONVERSION CHARACTER	DESCRIPTION
`d`	Integer notation
`e`	Exponential notation using lowercase e
`E`	Exponential notation using uppercase E
`g`	Exponential or fixed-point notation.
`G`	Identical to `'g'`, but uses uppercase E for exponential notation
`i`	Integer notation (identical to `'d'`)
`s`	String

In this list, `%g` uses exponential notation when the exponent is larger than or equal to the precision, or if the exponent is less than –4. The default precision is 6. Precision means the number of digits to the right of the decimal point for `%f` and the total number of digits for `%g`—which always removes insignificant zeros.

Finally, you have access to the following character codes for special formatting in strings:

TABLE 4-4: SPRINTF/FPRINTF SPECIAL CHARACTER

SPECIAL CHARACTER	DESCRIPTION
`\n`	New line
`\t`	Tab
`''`	Apostrophe

Examine some examples using `sprintf`:

```
sprintf('A: %-+10.2f ',[10.045,1.02])

ans =

  A: +10.04      A: +1.02

sprintf('A: %.1f B: %.3e\n',[1.01 1.00001,1.1],[1e4 1e-4 1])
```

```
ans =

  A: 1.0 B: 1.000e+004
  A: 1.0 B: 1.000e-004
  A: 1.1 B: 1.000e+000
```

Evaluating Strings

COMSOL Script can evaluate commands and statements you store in strings.

EVALUATING STRINGS

The general command for evaluating strings is the `eval` command, which evaluates an expression or a sequence of statements:

```
eval('a=pi+2')

a =

5.141593
```

If you use one or more output arguments, `eval` evaluates the expressions in the input and returns the results:

```
a = eval('pi+3');
a

a =

6.141593
```

To trap errors, `eval` provides a second input argument. If you provide two input arguments, `eval` evaluates the expressions in the second input argument if the evaluation of the first input argument results in an error:

```
statement = 'sinu(pi/2);';
errormsg = 'disp(''Invalid function'')''
eval(statement,errormsg)
Invalid function
```

Note: To handle errors, it is usually better to use the `try` and `catch` functions instead of `eval`. See "The TRY and CATCH Statements" on page 91 for more information.

EXAMPLES OF USING THE EVAL FUNCTION

One use of `eval` is to create varying names for files or variables. The following two examples illustrate this technique:

To create twelve variables, `rand1`, `rand2`, …, `rand12`, and store a random matrix in each of them, type:

```
for k=1:12
  eval(['rand', int2str(k), '=rand(3);']);
end
```

To save the contents in variables `d1`, `d2`, …, `d10` as ASCII data in the text file `data1.txt`, `data2.txt`, …, `data10.txt`, type:

```
for k = 1:10
  filename = ['data', int2str(k), '.txt'];
  variablename = ['d', int2str(k)];
  eval(['save ', filename , ' ', variablename, ' -ascii'])
end
```

In many cases, however, you can replace the use of `eval` with a direct function call. In the previous example (saving text file), it is possible to replace the call to `eval` with the following code that uses the standard function calling syntax when calling the `save` function:

```
save(filename,variablename,'-ascii')
```

This improves readability and is easier to write.

RETRIEVING OUTPUT DURING EVALUATION

To retrieve any output that occurs during the evaluation of statement, use the `evalc` function instead of `eval`. `evalc` provides this output in a first output argument:

```
[out,...] = evalc(...);
```

Otherwise, `evalc` is similar to `eval`.

EVALUATING STATEMENTS IN DIFFERENT WORKSPACES

The function `evalin` works just like `eval` but provides the opportunity to do the evaluation of the statements in another workspace than the one where the `evalin` function occurs.

```
evalin('base',...)
```

evaluates the statements in the main workspace.

```
evalin('caller',...)
```

evaluates the statements in the parent workspace in the function call stack.

If you call the function that contains `evalin` from the COMSOL Script command window, the `caller` workspace is the main workspace.

EVALUATING FUNCTIONS

To evaluate functions, use the `feval` command. See “Evaluating Functions” on page 135 for more information about `feval`.

SUMMARY OF EVALUATION FUNCTIONS

COMSOL Script provides the following functions for evaluating expressions and functions:

TABLE 4-5: EVALUATION FUNCTIONS

FUNCTION NAME	DESCRIPTION
`eval`	Evaluate an expression or a sequence of statements
`evalc`	Evaluate an expression or a sequence of statements and retrieve all outputs
`evalin`	Evaluate an expression or a sequence of statements in a specific workspace
`feval`	Evaluate a function

Cell Arrays

A *cell array* is one that can contain data of different types in its elements or cells including numerical values and arrays, strings, structures, and even other cell arrays. This ability makes it a flexible structure for data that is nonuniform (and that cannot fit into a numeric array) or unstructured (so that a structure array is not suitable).

Creating Cell Arrays

You create cell arrays in several ways:

- Use the `{` and `}` symbols (curly braces) in the same way you use `[` and `]` to create numeric arrays:

  ```
  c1 = {'a string', rand(3), 5; {1, 'another string'}, 1, eye(3)}

  c1 =

    'a string'           [3x3 double]            [5]
    {[1] 'another string'}       [1]   [3x3 double]
  ```

- Assign a value to a cell using two different types of indexing:
  - With indexing into the cell array's contents, you specify the contents of a cell by indexing using curly braces (`{}`):

    ```
    c2{1,3} = 'the third cell'


    c2 =


      []   []   'the third cell'
    ```

  - With standard indexing, you specify an individual cell:

    ```
    c3(1,3) = {'the third cell'}
    ```

    You then have to provide a cell in the assignment and not the contents of the cell. This example produces the same results as the example for content indexing.

- Use the conversion functions `cellstr`, `mat2cell`, `num2cell`, and `struct2cell`. For example,

  ```
  c = mat2cell(rand(5, 15), [2 3], [4 5 6])
  ```

  creates a 2×3 cell array where `c(1,1)` is the 2×4 submatrix in the upper left corner of the random matrix.

- Create a cell array with empty cells using the `cell` command:

```
c3 = cell(2,3,3);
```

You can create cell arrays of any dimension. To create an empty cell array, use `{}`, which is similar to `[]` for numeric arrays (you can also use `cell(0,0)` to create an empty 0×0 cell array).

CREATING CELL ARRAYS WITHIN CELL ARRAYS (NESTED CELL ARRAYS)

It is permissible to create nested cell arrays in several levels by using multiple sets of curly braces and the `cell` command. The line

```
c = {{{1}, rand(2)},{pi}}
```

creates a 1x2 cell array where the first cell contains another 1x2 cell array, and the second cell contains a 1x1 cell array. To create a 1x2 cell array with two 3x2 cell arrays with empty cells, type:

```
c = {cell(3,2), cell(3,2)}

c =

  [2x2 cell]  [2x2 cell]
```

Working With Cell Arrays

If the contents of a cell array consist of another variable, note that the cell contains a copy of the variable and not a pointer to it. This means that changing the contents of the cell does not change the data in the other variable.

REFERENCING AND MODIFYING CELL ARRAYS

You reference and work with cell arrays in the same way as other arrays in COMSOL Script. As noted earlier, in addition to cell indexing, content indexing using curly braces makes it possible to modify cell contents.

For instance, start with the cell array:

```
c = {{{1}, rand(2)} 5; eye(3), {pi}}
```

To add a column with cells containing the number `2` and the string `'string'`, write this line of code:

```
c(:,end+1) = {2; 'string'};
```

To remove the middle column, try this line:

```
c(:,2) = [];
```

To create a double array `d` and store in it the random matrix currently in the cell array in the top left cell of `c`, use this line:

```
d = c{1,1}{1,2};
```

To replace the string `'string'` with `'character array'` using indexing into the cell array contents, type:

```
c{2,2} = 'character array';
```

or, using standard indexing, type:

```
c(2,2) = {'character array'};
```

You can reshape a cell array just as it is possible with other arrays using, for example, `reshape` or the colon (`:`) and transpose (`'`) operators. For instance,

```
c = reshape(c,1,4)
```

converts `c` from a 2×2 cell array to a 1×4 cell array. A faster way to do this job is

```
c(:)'
```

USING CELL ARRAYS AS LISTS OF VARIABLES

You can work with cell arrays as lists of variables, using indexing into the contents with `{}`, in a similar way as a comma-separated list of variables. The following example shows how this works with three input arguments to the function `plot`.

With separate variables for the input data you might write

```
x=linspace(0,2*pi,50);
y=sin(x);
format = 'ro--';
plot(x,y,format);
```

Using a cell array to store the input arguments, the equivalent call to `plot` changes to:

```
args = {x, y, format};
plot(args{:});
```

The syntax `args{:}` works in the same way as inputting the three comma-separated input arguments.

On the other hand, the function `deal` is useful for distributing the contents of a cell array into individual variables, such as in

```
[x,y,format]=deal(args{:});
```

The `varargin` and `varargout` arguments, which are available with COMSOL Script functions, are cell arrays that provide the ability to handle varying number of input and output arguments of different types. See “Variable Number of Input and Output Arguments” on page 132 for more information about varargin and varargout.

APPLYING FUNCTIONS TO THE CONTENTS OF CELL ARRAYS

If you are working with a function that return a scalar numerical value for any input, you can then apply that function directly to a cell array with the cellfun function:

```
c = {rand(3), eye(5), zeros(2), pi}
maxnorms = cellfun('norm',c,inf)

maxnorms =

  2.269629   1   0   3.141593
```

The first argument in cellfun is the name of the function to apply to the cell contents, in this case norm. The second input argument is the cell array, here c; any additional optional input arguments to cellfun in turn serve as input arguments to the function that works on the cell array.

In addition, two special functions work with cellfun—prodofsize and isclass:

```
sizes = cellfun('prodofsize',c)

sizes =

  9   25   4   1
```

The input argument 'prodofsize' generates an output that contains the number of elements in each cell, that is, prod(size(c{i}) for each cell c{i}. As for the second of those two functions,

```
dbls = cellfun('isclass',c,'double')

dbls =

  1   1   1   1
```

The input argument 'isclass' checks the type of class (data structure) in each cell. You provide the name of the class as the third input argument. The output is of the same size; it is 1 if the contents of the cell is of this class and 0 otherwise. Notice that this is a numerical array. To convert it to a logical array, use logical(dbls);

If the function you want to apply does not return a scalar numerical value, use a loop to apply the function to the contents of the cells. For example,

```
for i=1:length(c)
  cellmax{i}=max(c{i});
```

```
end
```

produces a cell array `cellmax`, where each cell is a row vector containing the maximum value of each column in the matrix in the corresponding cell in `c`.

SUMMARY OF FUNCTIONS FOR WORKING WITH CELL ARRAYS

The following table summarizes the functions related to cell arrays:

TABLE 4-6: CELL ARRAY FUNCTIONS

FUNCTION NAME	DESCRIPTION
`cell`	Create a cell array
`cellstr`	Convert a character array to a cell array of strings
`cell2mat`	Convert a cell array to a matrix
`cell2struct`	Convert a cell array to a structure
`cellfun`	Apply a function to the elements of a cell array
`deal`	Distribute function inputs to output variables
`iscell`	True if the variable is a cell array
`iscellstr`	True if the variable is a cell array of strings
`mat2cell`	Create a cell array from a matrix
`num2cell`	Create a cell array from a numerical array
`struct2cell`	Convert a structure array to a cell array

Set Functions

A number of functions are designed to work with sets. Using them, COMSOL Script interprets arrays or cell arrays of strings as sets. The following set functions are available:

TABLE 4-7: SET FUNCTIONS

FUNCTION NAME	DESCRIPTION
`intersect`	Set intersection
`ismember`	Determine set members
`setdiff`	Set difference
`setxor`	Set exclusive OR
`union`	Set union
`unique`	Retrieve unique elements

USING SET FUNCTIONS WITH VECTORS AND MATRICES

The standard syntax works for vectors:

```
A = 1:10

A =

  1   2   3   4   5   6   7   8   9   10

B = 6:15

B =

  6   7   8   9   10   11   12   13   14   15

C = ones(1,10);

intersect(A,B)

ans =

  6   7   8   9   10

ismember(10,A)

ans =

true

unique(C)

ans =

  1
```

If you have 2D sets (matrices), the `set` functions support an extra input argument, `'rows'`. For example, `setdiff(A,B,'rows')` returns the row set difference (the rows in `A` that are not in `B`, where `A` and `B` must have the same number of columns).

In addition, you can get index vectors into the set vectors by providing additional output arguments. For details, see the online help for the individual functions.

Structures

A structure, more formally known as a structure array, is an array with fields that act as separate containers for different data. It is somewhat similar to the *struct* data structure in C/C++. Structures are convenient when you have data of different types that are related; for instance, a structure array is suitable for creating a small database where each structure is a “record.” As with other arrays, you can have a single structure (a 1x1 structure array), 1D or 2D arrays of structures, or even multidimensional structure arrays.

This section works with one example, specifically, a structure array that stores data about elements in the periodic table. The structure array holds this information:

- The symbol, such as H for hydrogen (a string)
- The name (a string)
- The atomic number (a numerical scalar)
- The mass numbers for the most common isotopes (a numerical array)
- The atomic weight (a numerical scalar)

Creating Structures

You create structures in two ways:

- Use the `struct` command. For instance,

  ```
  s = struct
  ```

  creates an empty structure without any fields. To create a single structure, use pairs of field names and field values in the call to `struct`:

  ```
  elements = struct('symbol','H','name','Hydrogen',...
  'atomic_number',1,'mass_numbers',1:3,'atomic_weight',1.00797);
  elements =

            symbol:  'H'
              name:  'Hydrogen'
     atomic_number:  [1]
      mass_numbers:  [1 2 3]
     atomic_weight:  [1.007970]
  ```

  To create a structure array, pass the values for each structure into a cell array for each field (all cell arrays must be of the same size, but COMSOL Script expands scalar values):

```
elements =
struct('symbol',{'H','He'},'name',{'Hydrogen','Helium'},...
'atomic_number',{1 2},'mass_numbers',{1:3, 3:4},...
'atomic_weight',{1.00797 4.0026})

elements =

1x2 struct array:
  symbol
  name
  atomic_number
  mass_numbers
  atomic_weight
```

- Use direct assignment statements. The syntax for adding fields to a structure is to type a dot (`.`) followed by the field name:

```
elements.symbol = 'H';
elements.name = 'Hydrogen';
elements.atomic_number = 1;
elements.mass_numbers = 1:3;
elements.atomic_weight = 1.00797;
```

Use standard array subscripting to add additional structures:

```
elements(2).symbol = 'He';
elements(2).name = 'Helium';
elements(2).atomic_number = 2;
elements(2).mass_numbers = 3:4;
elements(2).atomic_weight = 4.0026;
```

You need not enter data in all fields. For fields into which you do not assign a value, COMSOL Script sets the default value of an empty double array (`[]`):

```
elements(3).symbol = 'Li';
elements(3).name = 'Lithium';
elements(3)

ans =

          symbol:  'Li'
            name:  'Lithium'
   atomic_number:  []
    mass_numbers:  []
   atomic_weight:  []'
```

CREATING NESTED STRUCTURE ARRAYS

The contents of a field in a structure can be any COMSOL data structure, including another structure, which makes it possible to create nested structure arrays. For example, if you want to include more information about the isotopes of an element, you could create another structure array, `isotopes`, that contains a numeric field for the mass number and a logical flag that indicates if the isotope is stable. For hydrogen, `isotopes` becomes a 1×3 structure array:

```
isotopes.mass_number = 1;
isotopes.stable = true;
isotopes(2).mass_number = 2;
isotopes(2).stable = true;
isotopes(3).mass_number = 3;
isotopes(3).stable = false;
```

To add this to the elements structure array, use the direct assignment syntax:

```
elements.isotopes = isotopes;
```

Working With Structures

Many of the standard ways of working with arrays in COMSOL Script also apply to structure arrays. Specific to structures is the "dot" syntax to access the value of an individual field.

REFERENCING AND MODIFYING STRUCTURE ARRAYS

You can access individual structures in a structure array with standard references using subscripts in parenthesis. To get or modify the value of an individual field, append `.` (dot) followed by the name of the field:

```
elements(13).name

ans =

  Aluminium
```

To change the name of that element from spelling in British English to American English, type:

```
elements(13).name = 'Aluminum';
```

You can also access the field values for all structures in the array using brackets ([]), which is most useful for scalar numerical data. The line

```
atomicweights = [elements.atomic_weights];
```

produces a double array with the atomic weights for each element.

To access field values in a nested structure array, use the dot syntax to reach underlying fields, indexing into each structure array:

```
elements(1).isotopes(2).stable
```

```
ans =

true
```

ADDING AND DELETING FIELDS

To add a field to a structure, simply use a direct assignment:

```
elements(2).group = 'Noble gas';
```

For structures in a structure array where you have not assigned a value to the new field, the value is the empty matrix.

If you want to remove a field from the structure, use the `rmfield` function. As an example, remove the `mass_numbers` fields, which became redundant after adding the `isotopes` structure:

```
elements = rmfield(elements,'mass_numbers');
```

APPLYING FUNCTIONS TO THE CONTENTS OF STRUCTURES

You can use functions and operate on data in the various structure fields just as with any other COMSOL Script arrays. For example, to get the number of common isotopes for hydrogen, type:

```
nhiso = length(elements(1).mass_numbers)

nhiso =

3
```

Using the bracket syntax it is possible to operate on a field in all of the structures in a structure array. This is useful for statistical analysis, particularly when the field value is a scalar value. For example, to compute the mean atomic weight for the first 13 elements:

```
mean([elements(1:13).atomic_weight])

ans =

14.404209
```

USING DYNAMIC FIELD NAMES IN PROGRAMS

In functions, where the name of a field name may be an input argument, it is convenient to use dynamic field name that COMSOL Script evaluates when the function runs. This is useful, for example, if you work with a structure that contains all elements as individual fields: `elements.hydrogen`, `elements.helium`, `elements.lithium`, and so on. Then you can create a function that takes the element name as an input argument, `element_name`, and computes the number of isotopes. To indicate a dynamics field name, surround it with parentheses:

```
no_iso = length([elements.(element_name).isotopes.mass_number]);
```

would access the mass numbers in `elements.hydrogen.isotopes.mass_number` if the variable `element_name` contains the string `hydrogen`. `(element_name)` is the dynamic field name, which COMSOL Script replaces with `hydrogen` when running the code.

You can also use the functions `getfield` and `setfield` to get the contents of a field and assign a value to a field, respectively. To achieve the same functionality using `getfield` instead of the dynamic field name syntax, type:

```
this_element = getfield(elements,element_name);
no_iso = length([this_element.isotopes.mass_number]);
```

Summary of Functions Related to Structure Arrays

The following table summarizes the functions related to structures:

TABLE 4-8: STRUCTURE FUNCTIONS

FUNCTION NAME	DESCRIPTION
`cell2struct`	Convert a cell array to a structure
`fieldnames`	Return a cell array with field names
`getfield`	Get the value of a structure field
`isfield`	True for fields
`isstruct`	True for structures
`rmfield`	Remove a field from a structure
`setfield`	Set the value of a structure field
`struct`	Create a structure array
`struct2cell`	Convert a structure array to a cell array

5

The Programming Language

COMSOL Script is a scripting environment that includes a full high-level programming language. In the following chapter you will learn how to use the programming language to control the flow of the code by means of conditional statements and loops as well as other language constructs. Further, a section describes the debugging tools that help you find and correct problems in scripts and functions.

Flow Control

For controlling the flow of a program or script, COMSOL Script provides the following statements:

- `if`—for conditional branching
- `while`—for looping as long as a certain condition is True
- `for`—for looping
- `break`, `continue`, and `return`—for breaking out of a loop, continuing in a loop, and returning from a function, respectively
- `switch`—for branching among several cases based on an expression

It is permissible to create nested control flows in COMSOL Script programs.

IF Statements

`if` is the simplest way to execute one or more statements when a condition is met:

```
if rank(A)<3
  warning('Matrix does not have full rank!')
end
```

The condition, here `rank(A)<3`, can be any expression that evaluates to a logical or double matrix. The block of statements between `if` and `end` execute if all the elements of the condition are True.

You can combine an `if` with an `else` to run either of two statement blocks:

```
if x<0
  heaviside = 0;
else
  heaviside = 1;
end
```

COMSOL Script runs the first statement block if the condition is True; otherwise it executes the second statement block. Under no circumstances does the program execute both blocks.

As just pointed out, use an `if-else` construct when there are two statement blocks to choose from; when there are more than two statement blocks, use `if-elseif` instead:

```
if (x<=0) || (x>=3)
  y = 0;
```

```
elseif x<1
  y = x;
elseif x<2
  y = 1;
else
  y = 3-x;
end
```

You can place any number of `elseif` blocks between `if` and `end`. It is also possible to combine `elseif` with `else`; in this case, `else` must come after all the `elseif` blocks.

WHILE Loops

Insert `while` loops to execute a block of statements for as long as a condition is fulfilled. The following example finds the smallest n such that the harmonic series $H(n)$ exceeds 10:

```
H = 0;
n = 1;
while H<10
  H = H+1/n;
  n = n+1;
end
n
n =
  12368
H
H =
10.000043
```

COMSOL Script executes the statements between `while` and `end` as long as the condition evaluates to an all-True logical matrix. If the condition is False from the beginning, the statements never execute.

FOR Loops

A `for` loop runs a block of statements once for each value in a list:

```
H = 0;
for n=1:12368
  H = H+1/n;
end
H
H =
10.000043
```

This example runs the loop body (the statement `H = H+1/N`) a total of 12,368 times, once for each *n* in the list.

In the previous example the list of values is numerical. This need not be the case—it is also possible to loop over the elements of a cell array:

```
for noble={'He', 'Ne', 'Ar', 'Kr', 'Xe', 'Rn'}
  disp(noble)
end
```

In this example, the loop body runs six times, once for each element in the cell array.

The `for` statement runs the loop body once for each column in the loop variable. For row vectors, as in the previous examples, this is the same as executing the loop body for each element, but not for matrices:

```
A = rand(5,10);
for col=A
  max(col)
end
```

This example creates a 5x10 random matrix and, for each column, determines the maximum element. The loop body executes ten times.

BREAK, CONTINUE, and RETURN Statements

You can break off the execution of a `for` loop or a `while` loop with the `break` command. If the loops are nested, `break` only stops the execution of the innermost loop. Outside of loops, you can stop the execution of a function using the `return` statement. COMSOL Script then returns to the keyboard or to the function that invoked the function that contains the `return` statement. Normally, COMSOL Script runs a function to the ending statement and then returns. When you use `return` for an early exit from a function that defines output arguments, make sure that you have assigned values to these output variables at the point where the `return` statement occurs.

To continue with the next iteration in a loop without running the remaining statements in the loop, use the `continue` statement. In nested loops, `continue` passes control to the next iteration of the loop that encloses it.

The SWITCH Statement

The `switch` statement is a generalization of `if` where there are several possible branches to take. The following example displays the name of an element when given its atomic number:

```
switch number
case 1
  disp('Hydrogen')
case 2
  disp('Helium')
case 3
  disp('Lithium')
end
```

The program evaluates the condition (`number`) just once and compares it to the `case` branches in order. It then executes the statements associated with the first case that matches the condition. If none of the cases match, then it does not execute any statements.

Note: Unlike languages such as Java and C, COMSOL Script executes at most one case—there is no implicit fall through that causes execution to proceed with a further case.

Many applications require that program flow follow the same branch for a number of values. This is possible in a `case` construct by using a cell array of values:

```
switch n
case {2 3 5 7}
  disp('Prime')
case {1 4 6 8 9 10}
  disp('Not prime')
otherwise
  disp('Don''t know')
end
```

The program logic first tests `n` against the values in the cell array `{2 3 5 7}`, and if it finds a match it executes the statement for the case. If not, it tests the second cell array. This example also illustrates the use of the `otherwise` statement block. If none of the cases match and there is an `otherwise` block at the end of the `switch` statement, then the commands following the `otherwise` statement execute.

If the expression following `switch` is numeric, then the program tests it against the cases using the equality operator (==). The expression can also be a string, in which case the test for equality uses the `strcmp` function:

```
switch element_name
case 'Hydrogen'
  mass = 1.008;
case 'Helium'
  mass = 4.003;
case 'Lithium'
  mass = 4.941'
end
```

To take the same branch for several different values, use cell arrays of strings in the cases as in this example:

```
switch element_name
case {'Helium', 'Neon', 'Argon', 'Krypton', 'Xenon', 'Radon'}
  disp('Noble gas')
case {'Lithium', 'Sodium', 'Potassium', ...
      'Rubidium', 'Cesium', 'Francium'}
  disp('Alkali metal')
end
```

Working with Variables

This section contains information about naming of variables in a COMSOL Script program, the possibility to assign a value to a variable in another workspace, and getting user input.

Naming Variables

The names of variables follow the same conventions as file names:

- Variable names are case sensitive
- A variable name can only contain letters, digits, and underscores
- A variable name must start with a letter

To check if a variable name is valid, use the `isvarname` function:

```
isvarname('2_fcn')

ans =

false
```

Also, you cannot use variable names that are reserved words in the COMSOL Script language such as `if`, `for`, and `while`. Use the function `iskeyword` to check is a string contains a reserved word:

```
iskeyword('case')

ans =

true
```

Assigning a Value to a Variables in Other Workspaces

Normally, when you assign a value to a variable, this takes place in the local workspace (which is the main workspace when working directly in the command windows or when running scripts). If you need to assign a value to a variable in another workspace, use the `assignin` function:

```
assingin('base',var,value)
```

assigns the value `val` to the variable in the string `var` in the main workspace.

```
assingin('caller',...)
```

assigns a value to a variable in the parent workspace in the function call stack.

One use of `assignin` is to store data from the local function in the main workspace.

If you call the function that contains `assignin` from the COMSOL Script command window, the `caller` workspace is the main workspace.

Getting User Input

You can let the user provide input to a variable when running a function or script. To do so, use the `input` function:

```
a = input('Question:');
```

displays `Question:` at the command prompt and waits for user input.

COMSOL Script evaluates the input in the current workspace and assigns the result to the output variable.

If you want to use the input as a string, call `input` using the string `s` as a second input argument:

```
a = input('Question:','s');
```

COMSOL Script then returns the string that the user enters without evaluating it.

Error Handling

When you write programs with COMSOL Script, a number of statements help you catch errors and then take appropriate action. You can also throw errors and issue warnings.

The TRY and CATCH Statements

The try-catch construct provides a means to handle errors gracefully:

```
try
  b = A(n);
catch
  disp('n out of range, using b=1.')
  b = 1;
end
```

If an error occurs during the execution of the statements between try and catch, COMSOL Script runs the statements in the catch block. In this example, A is a matrix indexed using n. If n is not a valid index, then the statements in the catch block inform the user of this fact and continue execution despite the error. Without the try-catch construct, an invalid index would halt execution and issue an error message.

It is possible to rewrite the previous example using try without a catch clause:

```
b = 1;
try
  b = A(n);
end
```

Here, if an error occurs when executing the statements between try and end, COMSOL Script continues and runs the statements following the try-end block. In other words, omitting the catch block has the same effect as putting no statements between catch and end.

Throwing Errors and Displaying Warnings

In COMSOL Script functions you can throw an error with the error function. For instance, the following code causes an error to occur and displays the text in errormsg:

```
error(erromsg)
```

COMSOL Script ignores a call to error if the string that you pass is empty.

You can also display a warning message. For example,

```
warning(warningmsg)
```

displays the text in `warningmsg`. This does not stop the program from running.

SETTING OR RETRIEVING THE LAST ERROR MESSAGE

You can retrieve or set the last (current) error message with the `lasterr` and `lasterror` functions. The difference between them is that `lasterr` works with a string as an error message, and `lasterror` works with an error-message structure that has a field for the error message and an identifier. Most often you are only interested in the error message itself. In that case, it is easier to use `lasterr`. The line of code

```
errormsg = lasterr;
```

returns the current error message, whereas

```
lasterr(newerrormsg)
```

sets the current error message to the string in `newerrormsg`.

Performance Considerations

In several situations you can improve the performance of a COMSOL Script program by using alternatives to flow-control statements. All of these cases are examples of "vectorization," that is, taking advantage of the ability of COMSOL Script functions to use arrays instead of scalar as input variables, thereby avoiding the need for a loop.

Using Built-in Functions Instead of FOR

The constructs described in the previous sections are general in scope, but they can sometimes lead to poor performance and verbose code. As an example, revisit an example from the `for` section:

```
H = 0;
for n=1:12368
  H = H+1/n;
end
```

You can replace this loop with a single line:

```
H = sum(1./(1:12368));
```

that is clearer and executes much faster. For many common operations, such as `sum` as shown here, built-in functions do the job better than loops.

Using Logical Operators Instead of IF

It is often possible to replace `if` statements with logical operators:

```
L = logical(size(A));
for i=1:numel(A)
  L(i) = A(i)>0;
end
```

This code computes a logical matrix L that is True in the positions where A is positive. A faster and more compact way to accomplish the same thing is with a matrix-wise comparison:

```
L = A>0;
```

Using Pointwise Operators Instead of Loops

The loop

```
C = zeros(size(A));
for i=1:numels(A)
  C(i) = A(i)*sin(A(i));
end
```

can also be optimized with a single command; write it instead using pointwise multiplication:

```
C = A.*sin(A);
```

Profiling to Find Bottlenecks

By generating profiling information for a function or script, you get an instant overview of which functions that run most frequently and where the COMSOL Script code spends most of the CPU time. This makes it possible to find potential bottlenecks in a large function or script. To start collecting profiling information, type

```
profile on
```

You can stop collecting information using profile off and clear the profiling information using profile clear. Type `profile report` followed by the function name to print a report showing the profiling information for that function or script file. The following code provides a small example:

```
profile clear
profile on
corrcoef(rand(1000),'row','complete','alpha',0.04);
profile off
profile report corrcoef
```

This prints a report to the command line about the total time spent in `corrcoef` and the number of times and relative time spent on each line in the M-file. An excerpt from the report is:

```
1      0.00%   alpha = 0.05;
1      0.00%   row = allstr;
1      0.00%   for rowid = flagind:2:nargin-1
2      0.00%     if isequal(varargin{rowid},'alpha')
1      0.00%       alpha = varargin{rowid+1};
1      0.00%       if ~isscalar(alpha) || alpha < 0 || alpha > 1
```

This reveals that COMSOL Script ran all the lines above once, except for the fourth line from the top of this section, which ran twice. None of this code contributed significantly to the total time spent running `corrcoef`. Further down the report, the following line explains where most of the time was spent:

```
1      82.07%     C = cov(x);
```

This is a call to another function, `cov`. To see the profiling information for that functions, type

```
profile report cov
```

Global and Persistent Variables

Global Variables

Sometimes one or more variables appear in a large number of functions, and it would be cumbersome to pass these variables around using function calls. The solution is to use global variables.

To declare a variable as global, use `global`:

```
global STEPSIZE
```

Variables declared as such have their values stored in a global workspace accessible from any function or workspace. As an example, consider the following lines running at the command prompt:

```
global STEPSIZE
STEPSIZE = 0.001;
...
area = quadrature('sin',0,1)
```

where the `quadrature` function performs numerical integration using the trapezoidal rule:

```
function out = quadrature(func,a,b)
global STEPSIZE
out = STEPSIZE*sum(feval(func,(a+STEPSIZE/2):STEPSIZE:b));
```

When `quadrature` runs, the `global STEPSIZE` declaration results in COMSOL Script taking the value of `STEPSIZE` from the main workspace. If `STEPSIZE` is modified within the function, then its value also changes in the main workspace.

When you declare a value as `global`, doing so removes any existing value it has. If you declare a variable as `global` when using it for the first time, it is initialized to the empty matrix:

```
A = 17;
global A
A
A =
[]
```

To check if a variable is global, use the `isglobal` function: `isglobal('A')`.

Persistent Variables

The fact that you can modify global variables everywhere is a strength as well as a weakness. Sometimes a function has some internal state that you want to save between consecutive calls. It is possible to implement this idea using a global variable, but there is always the risk that some other routine might also modify that global state variable. A better solution is to use a *persistent variable*, whose value as declared in a function remains across calls, but it is not possible to read or modify it elsewhere.

Consider the following function, which generates pseudorandom numbers between 0 and 1 with a linear congruential generator:

```
function out = lcg
persistent STATE
if isempty(STATE)
  STATE = 12345;
end
STATE = mod(1103515245*STATE+12345, 2^31);
out = STATE/2^31;
```

`lcg` uses the persistent variable `STATE` to remember the number-generator's state. Just as when working with global variables, the first time you make a `persistent` declaration for a variable, it is initialized with the empty matrix. The `if isempty(STATE)` test in `lcg` uses this behavior to detect if the code must initialize the `STATE` variable because it is the first time that the function runs.

A WORD OF CAUTION

Global variables are easy to use—and overuse. They introduce global dependencies, and as a result the code becomes more difficult to understand and maintain. Avoid the excessive use of global variables, especially in projects that consist of more than a few functions. Persistent variables do not suffer from this problem, so use them instead of global variables when possible.

When declaring global variables in functions that other programmers will use, try to select expressive variable names. This decreases the risk of collisions between global variables in different functions. Such collisions can be difficult to detect because they do not lead to immediate errors; global variables can silently receive bad values that lead to problems in completely unrelated code.

Debugging

Some errors can appear relatively frequently when you program with COMSOL Script. This section describes the most common errors and how to use the debugging tools to find the cause.

Common Errors

ARRAY INDEX OUT OF BOUNDS

An extremely common programming error is using invalid array indices:

```
A = 1:10;
...
A(10,1)
Error: Array index 1 is 10, dimension length is 1.
```

COMSOL Script catches all attempts to use invalid indices during program execution, but the sources of the errors can still be difficult to find. This example shows a common error: `A` is a 1×10 matrix (a row vector) that a routine later indexes as a 10×1 matrix, a column vector. You can remove the error here by replacing `A(10,1)` with `A(1,10)` or, even better, `A(10)`.

There is no foolproof way to avoid such errors, but they occur less often if you write code with consistency in mind. Avoid mixing row and column vectors among variables with similar usage.

MATRIX DIMENSIONS DISAGREE

All operators and most built-in functions require that matrix dimensions be compatible:

```
A = rand(3, 4);
...
B = rand(4, 3);
...
C = sin(A)+cos(B);
Error: Matrix dimensions must agree.
Error in built-in function plus.
```

The only exception to this rule is that COMSOL Script expands scalars to constant matrices in most situations. Code such as

```
A = rand(3, 4);
```

```
...
B = pi;
...
C = sin(A)+cos(B);
```

runs without error because the software implicitly expands B into a 3×4 matrix where all elements equal π.

MIXING POINTWISE AND MATRIX OPERATORS

Pointwise operators such as .* operate on individual matrix elements, but matrix operators such as * operate on entire matrices. A common error is to mix the two operator types in the same expression, which often leads to errors:

```
A = 1:5;
B = 6:10;
C = sin(A)+cos(B)+A*B;
Error: Incompatible dimensions in matrix multiplication.
Error in built-in function mtimes.
```

Replacing A*B with A.*B eliminates this problem.

CONFUSING / AND \

To solve the linear system of equations Ax = b, use x = A\b (see the section “Elementary and Special Math Functions” on page 42 for more information about / and \. The similar looking code x = A/b usually results in an error:

```
A = rand(4, 4);
b = rand(4, 1);
x = A/b;
Error: Incompatible matrix dimensions.
Error in built-in function mrdivide.
```

For some arguments, confusing A\b with A/b does not result in an error, but the answer is not as expected because the answer solves the wrong problem: x=A/b is the solution to the system b'x'=A'.

Debug Commands

When writing any nontrivial program, it is inevitable that errors occur. You can often correct common errors, such as the ones just described, by reading the error message, but as noted earlier many errors can be difficult to locate. This section covers debugging commands useful for locating errors, and they include setting breakpoints and stepping through code.

SETTING BREAKPOINTS AND STEPPING THROUGH CODE

The command `dbstop` sets a breakpoint on a function or a line in a function. Consider again the `lcg` function:

```
function out = lcg
persistent STATE
if isempty(STATE)
  STATE = 12345;
end
STATE = mod(1103515245*STATE+12345, 2^31);
out = STATE/2^31;
```

`dbstop lcg` sets a breakpoint on the `lcg` function. The next time that function is called, the program pauses its execution and enters a special state:

```
C» dbstop lcg
C» lcg
lcg   2   persistent STATE
D»
```

Notice how the prompt changes from `C»` to `D»` to highlight the new state. You can enter any command or expression at the debug prompt, but note that some debug commands can exist only in this state. For instance, to step through the code use the `dbstep` command:

```
D» dbstep
lcg   3   if isempty(STATE)
D» dbstep
lcg   4     STATE = 12345;
D» dbstep
lcg   6   STATE = mod(1103515245*STATE+12345, 2^31);
D» dbstep
lcg   7   out = STATE/2^31;
D» STATE
STATE =
1406932606
```

`dbstep` executes the line on which execution has stopped, then it stops on the next line in the same function. Repeated applications of `dbstop` is one way to follow program flow. To resume normal execution, use `dbcont`:

```
D» dbcont

ans =

0.655154
```

Remove one or more breakpoints with `dbclear`. Called without any arguments, it removes all breakpoints; called with arguments, it behaves as the opposite to `dbstop`. For instance, `dbclear lcg` removes the breakpoint set on the `lcg` function.

To set a breakpoint on a line of a function, use `dbstop` with a line-number argument:

```
C» dbclear
C» dbstop lcg 6
C» lcg
lcg   6   STATE = mod(1103515245*STATE+12345, 2^31);
D»
```

To remove a breakpoint set on a line, use `dbclear` with a line-number argument. To remove the breakpoint in this example, call `dbclear lcg 6`.

You can also set a manual breakpoint in a script or function using the `keyboard` command.

MOVING UP AND DOWN IN THE DEBUG CALL STACK

Use `dbup` and `dbdown` to move up and down in the debug call stack. These commands change the debug workspace to the parent (`dbup`) or child (`dbdown`) of the current debug workspace. Using these functions you can explore all workspaces in the call stack. To use `dbdown`, you must first run `dbup` at least once. Using an integer input argument to `dbup` and `dbdown`, for example, `dbup(3)`, is equivalent to making that many calls to `dbup` and `dbdown` without input arguments.

ERROR BREAKPOINTS

The breakpoints in the previous section were set on specific lines of a function. A common situation when debugging a functions is tracking down why an error occurs. To do so, run the function, note the line number where the error occurs, set a breakpoint on that line, and then run it again. COMSOL Script then stops running the function when it reaches that line, and you can examine the cause of the problem. This approach is cumbersome if the line where the error occurs executes many times. A better solution is to use `dbstop if error`, which causes execution to stop only when an error occurs.

Summary of DEBUG Functions

The following table summarizes the debug functions:

TABLE 5-1: DEBUG FUNCTIONS

FUNCTION NAME	DESCRIPTION
dbclear	Remove breakpoint
dbcont	Continue execution
dbdown	Move down in debug call stack
dbquit	Stop execution
dbstack	Display function-call stack
dbstatus	List breakpoint conditions
dbstep	Step to the next source code line
dbstop	Set breakpoint
dbtype	Display source code of function
dbup	Move up in debug call stack

6

Linear Algebra and Matrix Functions

The core of COMSOL Script consists of functions for linear algebra and matrix functions such as equation-system solvers, norms, eigenvalues, LU factorization, and singular-value decomposition (SVD). This chapter describes how to use these functions.

Matrix Functions and Matrix Analysis

For details about basic matrix operations such as inversion (including the pseudoinverse) and transposition, see "Working with Matrices and Arrays" on page 35.

Elementary Matrix Functions

Most COMSOL Script functions work on matrices in an element-by-element fashion so that, for example, `sqrt(M)` and `cos(M)` compute matrices of the same size as the matrix M containing elements $\sqrt{M_{ij}}$ and $\cos(M_{ij})$, respectively.

Matrix Analysis

Important matrix concepts that COMSOL Multiphysics addresses include trace, rank, determinant, norms, and condition numbers.

THE TRACE OF A MATRIX

The `trace` function computes the *trace* of a matrix (that is, the sum of the diagonal elements):

```
trace(diag([1 5 10]))

ans =

16
```

THE RANK OF A MATRIX

To compute the *rank* of a matrix, use the `rank` function, which returns the maximum number of independent rows or columns:

```
rank(eye(3))

ans =

3
rank(ones(3))

ans =

1
```

THE DETERMINANT OF A MATRIX

The *determinant* of a matrix A, $\det(A)$, is an interesting property. In particular, it is nonzero if and only if the matrix is *nonsingular*. To compute the determinant, use the `det` function:

```
det([1 2 3; 1 0 1; 1 1 1])

ans =

2
```

For a singular matrix, COMSOL Script issues a warning:

```
det(ones(2))
Warning: Singular matrix.

ans =

0
```

One property of determinants is that $\det(AB) = \det(A)\det(B)$:

```
A = [1 2; 1 4];
B = [5 0; 1 5];
det(A)

ans =

2

det(B)

ans =

25

det(A*B)

ans =

50.000000
```

VECTOR AND MATRIX NORMS

With the `norm` function you can compute a number of *vector norms* and *matrix norms*. A norm is a scalar quantity that is associated with a matrix or vector and that has certain properties (for example, it is nonnegative and zero exactly when the matrix or vector contains zeros only).

Vector Norms

Use `norm` to compute the following vector norms:

- Euclidean norm: `norm(V)`
- P-norms: `norm(V,p)`
- Maximum and minimum norms: `norm(V,inf)` and `norm(V,-inf)`

For a vector V, `norm(V)` is the same as `sqrt(V.^2)`; `norm(V,1)` is the same as `sum(V)`; while `norm(V,inf)` and `norm(V,-inf)` are the same as `max(V)` and `min(V)`, respectively:

```
V = [0 1 2 3 4 5]
norm(V)

ans =

7.416198

sqrt(V.^2)

ans =

7.416198
```

Matrix Norms

Use `norm` to compute the following matrix norms:

- Largest singular value: `norm(A)`, `norm(A,2)`
- 1-norm: `norm(A,1)`
- Frobenius norm: `norm(A,'fro')`
- Infinity norm: `norm(A,inf)`

The Frobenius norm is the same as `sqrt(sum(sum(abs(A).^2)))`, and the largest singular value norm is the same as `max(svd(A))`.

```
A = [10 2; 3 5];
norm(A)

ans =

11.052182

max(svd(A))

ans =
```

```
11.052182

norm(A,'fro')

ans =

11.747340

sqrt(sum(sum(abs(A).^2)))

ans =

11.747340
```

CONDITION NUMBERS

The `cond` function returns the *condition number* for an inversion, and it is a measure of *linear-system sensitivity*, that is, how much a small perturbation of the input matrix affects the resulting inverse. It is the product of a matrix's norm and that of its inverse. The larger the condition number $\kappa(A)$ of a matrix A, the more sensitive it is. A singular matrix has the convention that $\kappa(A) = \infty$.

`cond(A)` computes the 2-norm condition number. Use `cond(A,p)` for other P-norms. The condition number is equal to the ratio of the largest to smallest singular value:

```
A = [10 2; 3 5];
cond(A)

ans =

2.776153

sv = svd(A);
max(sv)/min(sv)

ans =

2.776153
```

Use `condeig` to compute the condition number for the eigenvalues.

Summary of Matrix Functions

The following list provides a summary of fundamental matrix functions and functions for matrix analysis:

TABLE 6-1: MATRIX FUNCTIONS

FUNCTION NAME	DESCRIPTION
`cond`	Condition number for inversion
`condeig`	Condition number for eigenvalue
`det`	Determinant
`inv`	Matrix inverse
`norm`	Matrix or vector norm
`pinv`	Pseudoinverse
`rank`	Rank of a matrix
`trace`	Trace of a matrix

Linear-Algebra Algorithms

This section covers the functions in COMSOL Script that implement fundamental linear-algebra algorithms for general linear systems, orthogonalization, and eigenvalue problems. For a theoretical treatment of matrix computations, see Ref. 1.

These algorithms are based on the linear-algebra package LAPACK (Ref. 2).

For complete information about available syntaxes for the functions, see the *COMSOL Script Command Reference* or the online help).

LU Decomposition and Solving Linear Equation Systems

When solving a system of linear equations $Ax = b$ using Gaussian elimination, it is useful to divide the system matrix A into a lower triangular matrix L and an upper triangular matrix U so that $A = LU$. To do so, use the `lu` function:

```
A = [1 2 3 ; 1 4 9; 1 8 27];
[L,U] = lu(A)
L =

    1      0          0
    1      0.333333   1
    1      1          0

U =

    1     2      3
    0     6     24
    0     0     -2
```

It is easy to verify that `A` and `L*U` are equal.

To solve a system of equations with COMSOL Script, use `/` and `\` (the slash and backslash operators, or right division and left division operators), which also work for over- and underdetermined equation systems. To solve the equation $Ax = b$ for x, with b equal to `[2; 10; 44]`, use

```
x = A\b
x =

   3
  -5
   3
```

For more information about the `/` and `\` operators see "Elementary and Special Math Functions" on page 42.

`lu`, `/`, and `\` all work with sparse matrices as well as standard full matrices. You can also call `lu` so that its output also includes permutation matrices.

HESSENBERG FORM

Another matrix form similar to an upper triangular matrix is the upper Hessenberg form of a square matrix, which is zero below the first subdiagonal. To compute the Hessenberg form of a matrix, use the `hess` function:

```
A = [1 1 0; 1 1 1; 1 0 1];
H = hess(A)

H=

    1          -0.707107    -0.707107
   -1.414214    1.500000    -0.500000
    0           0.500000     0.500000
```

The Hessenberg form of a matrix A has the same eigenvalues as A.

With an additional output argument, `hess` also returns unitary matrix `Q` such that `Q*H*Q'` is equal to `A`.

Matrix Factorization—Cholesky and QR

COMSOL Script also supplies functions for Cholesky factorization and orthogonal factorization using the QR algorithm.

CHOLESKY FACTORIZATION

For a symmetric (or Hermitian) positive definite matrix A, it is possible to compute a Cholesky factor C such that $A = CC^T$. To do so in COMSOL Script, use the `chol` function:

```
C = chol(A);
```

This operator uses the DPOTRF and ZPOTRF functions from LAPACK and supports matrices that are not positive definite; it does so through a second output argument `P` such that `C'*C = A(1:P-1,1:P-1)`.

QR FACTORIZATION

QR factorization, or orthogonal factorization, is not restricted to square matrices. The QR factorization of a matrix A gives a product of an orthogonal square matrix Q and an upper triangular matrix R such that $A = QR$. To compute the Q and R factors, type:

```
[Q,R] = qr(A)
```

You can also get the output from the LAPACK algorithms DGEQRF and ZGEQRF by typing

```
qr(A)
```

Other options include reduced-size factorizations.

Orthonormal Bases for Null Spaces and Ranges

Two functions in COMSOL Script compute orthonormal bases: `null`, which computes an orthonormal basis of the null space of a matrix, and `orth`, which computes an orthonormal basis of the range of a matrix.

The number of columns in the orthonormal basis of the range of a matrix are the same as the matrix rank.

The null space of a matrix A is the space of vectors x such that $Ax = 0$.

The following example illustrates these properties:

```
A = [1 2 3; 4 5 6; 7 8 9];
N= null(A)

N =

  -0.408248
   0.816497
  -0.408248

A*N

ans =

           0
  -4.441e-016
           0

O=orth(A)

O =
```

```
    -0.214837    0.887231
    -0.520587    0.249644
    -0.826338   -0.387943

rank(A)

ans =

 2
```

For both `null` and `orth` you can supply a tolerance as a second input argument.

Eigenvalues and Eigenvectors of a Matrix

Computing the eigenvalues λ and the eigenvectors x for a matrix A is important in many applications where you investigate a system's dynamics.

The eigenvectors and eigenvalues for A have the property that $Ax = \lambda x$. Eigenvalues can be complex even if the matrix A is real.

To compute the eigenvalues and eigenvectors, use the `eig` function:

```
[x,lambda] = eig(A)
```

where `lambda` is a diagonal matrix with the eigenvalues on the diagonal.

You can also compute the solution to a generalized eigenvalue problem $Ax = \lambda Bx$ by typing

```
[x,lambda] = eig(A,B)
```

As an example, compute the eigenvalues and eigenvectors of a 3-by-3 matrix A:

```
A = [0 1 1;-1 0 0; 0 0 1];
[x,lambda]=eig(A);
lambda
lambda =

   0 + 1i  0 + 0i  0 + 0i
   0 + 0i  0 - 1i  0 + 0i
   0 + 0i  0 + 0i  1 + 0i
```

In this case, two of the eigenvalues are complex (i and $-i$), and the third one is 1. From the definition of an eigenvector x, it is clear that you can multiply that vector by any number and it is still an eigenvector. The `eig` function scales the eigenvectors so that their Euclidean norm is 1.

For sparse eigenvalue problems, use the `eigs` function, which computes a few eigenvalues and eigenvectors for a sparse matrix.

Singular Value Decomposition and Schur Decomposition

The functions `svd` and `schur` implement singular value decomposition and Schur decomposition, respectively.

SINGULAR VALUE DECOMPOSITION

For a real matrix A, there exist orthogonal matrices U and V such that $U^T AV = \sigma$, which is a diagonal matrix that contains the *singular values* of A. To compute the singular values σ as well as U and V for a matrix, call the `svd` function:

```
[U,sigma,V] = svd(A);
```

One aspect of the singular values is their relation to matrix norms. For example, the largest singular value is a common 2-norm of a matrix.

SCHUR DECOMPOSITION

For a square matrix A, there exists a unitary matrix Q such that $Q^H AQ = T = D + N$ is a Schur decomposition of A, where D is a diagonal matrix of eigenvalues, and N is a strictly upper triangular matrix. There are two forms of the Schur decomposition: real and complex Schur forms.

To compute the real Schur form enter

```
T = schur(A)
```

and to get the complex Schur form of A type

```
T = schur(A,'complex')
```

Using the default real form, `schur` puts the eigenvalues on the diagonal if they are real and in a 2×2 block on the diagonal if they are complex. In the latter case, the complex eigenvalues are the eigenvalues of each block.

The complex Schur form gives the eigenvalues on the diagonal, independent of whether they are real or complex.

To get the unitary matrix Q, type

```
[U,T] = schur(A,...)
```

This example shows the structure of the Schur decomposition and its relationship to the eigenvalues of a matrix:

```
A=[1 2 1;2 2 1; 3 2 1];
schur(A)

ans =
```

```
    4.828427     2.041241     0.620171
           0    -0.828427     0.669867
           0            0    7.031e-018

eig(A)
eig(A)

ans =

     4.828427
    -0.828427
  -7.031e-018
```

To reorder the unitary matrix `U` and Schur matrix `T` (the output from a call to `schur`), so that a selected cluster of eigenvalues appears in the leading diagonal blocks, use the `ordschur` function.

```
[U1,T1] = ordschur(U,T,order)
```

where the third input argument (`order`) can be a logical vector, where a true (1) entry signifies a selected eigenvalue, or an integer vector, where each element corresponds to one eigenvalue.

The Matrix Exponential

The function `expm` computes the matrix exponential, e^A, for a square matrix A:

```
B = expm(A);
```

and it equals the series $I + A + A_2/2! + A^3/3! + \ldots$

The matrix exponential of a diagonal matrix is a diagonal matrix with the nonzero elements equal to $e^{A_{ii}}$.

Do not confuse this function with the elementwise exponential `exp`, which computes the exponential for each element in the input vector.

The Matrix Logarithm

The function `logm` computes the matrix logarithm, $\ln(A)$, for a square matrix A:

```
B = logm(A);
```

The function uses an iterative algorithm to compute the principal logarithm of a matrix A. This algorithm does not necessarily converge for all square matrices A, and it is only defined for matrices with positive real eigenvalues.

Do not confuse this function with the elementwise logarithm `log`, which computes the natural logarithm for each element in the input vector.

Evaluating Other Matrix Functions

In addition to the predefined functions, you can use the `funm` function to evaluate additional functions that are possible to evaluate on a square matrix. The function must have a Taylor series with an infinite radius of convergence, such as trigonometric functions. In the M-file that you create for a matrix function, you must also define the derivative of that function so that the function call can ask for a derivative of any order. For example, to implement a matrix version of `cosh`, `coshm`, create a short M-file, `coshm.m`:

```
function c = coshm(a,k)
if mod(k,2);
  c = sinh(a);
else
  c = cosh(a);
end
```

Depending on if k contains an even or odd number, the function returns the correct derivative, $\sinh(a)$ for odd derivatives, and $\cosh(a)$ itself for even derivatives. To then compute the hyperbolic cosine for a square matrix A:

```
coshypm = funm(A,'coshm');
```

In addition, you can pass additional properties such as tolerances using a third input argument to funm. See the command-line help for `funm` or the *COMSOL Script Command Reference* for details.

The Kronecker Tensor Product

The *Kronecker tensor product* of an m_1-by-n_1 matrix A and an m_2-by-n_2 matrix B is an (m_1m_2)-by-(n_1n_2) matrix with elements formed according to the following example for a 2-by-2 matrix A and a 4-by-2 matrix B:

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{bmatrix}$$

so that the result is an 8-by-2 matrix.

To compute the Kronecker tensor product, use the `kron` function:

```
C = kron(A,B)
```

The following example shows what the Kronecker tensor product looks like for the product of a 2-by-2 and a 2-by-3 matrix:

```
kron([1 2;0 2],[2,3,4;1,1,1])

ans =

  2   3   4   4   6   8
  1   1   1   2   2   2
  0   0   0   4   6   8
  0   0   0   2   2   2
```

Summary of Linear-Algebra Functions

The following list summarizes the linear-algebra functions in COMSOL Script:

TABLE 6-2: LINEAR ALGEBRA FUNCTIONS

FUNCTION NAME	DESCRIPTION
chol	Cholesky factorization
eig	Compute eigenvalues and eigenvectors
eigs	Compute a few eigenvalues and eigenvectors of a sparse matrix
expm	Matrix exponential
funm	Evaluate matrix function
hess	Hessenberg form
kron	Kronecker tensor product
logm	Matrix logarithm
lu	LU decomposition
mldivide, /	Solve linear system of equations
mrdivide, \	Solve linear system of equations
null	Orthonormal basis of the null space of a matrix
ordschur	Reorder Schur factorization
orth	Orthonormal basis of the range of a matrix
qr	QR factorization
schur	Schur decomposition
svd	Singular value decomposition

References

1. Gene H. Golub and Charles F. van Loan, *Matrix Computations*, 3rd ed., The Johns Hopkins University Press, 1996.

2. http://www.netlib.org/lapack/.

7

Scripts, Functions, and M-files

This chapter reviews the difference between script files and functions, and it also shows how to work with these M-files in the COMSOL Script environment. For more information about Model M-files, which are script files that contain COMSOL Multiphysics models, see the *COMSOL Multiphysics User's Guide* and the *COMSOL Multiphysics Scripting Guide*.

Overview of M-Files

COMSOL Script lets you write a program that contains a wide range of functions and language components. In that way you can automate computations and extend the COMSOL Script environment with new functions. To create a COMSOL Script program, simply collect the statements that make up the code into an *M-file*, a text file with the extension `.m`. To run the code in the M-file, save it in a directory that is part of the *M-file path* and call it from the command line.

There are two types of M-files:

- Script files, which are collections of COMSOL Script commands. You run these files exactly as if you were to type them at the command prompt. All commands operate in the main workspace and can modify, create, and delete variables in the that workspace. Scripts are useful for automating sequences of commands. A script allows no input or output arguments.
- Functions, which have a separate workspace and can handle multiple input and output arguments. Use functions to extend the COMSOL Script language with functionality that suits your applications.

Creating an M-file

Because all M-files are plain-text files, you can create or modify them with any text editor. An M-file must have the extension .m, and function names can include only the following characters:

- Letters (the name must begin with a letter)
- Numbers
- Underscores

For variable names, the same rules apply (see “Naming Variables” on page 89).

The length of an M-file name must not be longer than the number that the function `namelengthmax` returns (the same limit applies to variable names). The operating system can also have restrictions on the length of a file name.

INSERTING COMMENTS

You can insert comments anywhere in an M-file with the `%` character. All text on a line after the `%` character becomes a comment that COMSOL Script does not execute.

DISPLAYING CONTENTS OF AN M-FILE

Use the `type` function to display the contents of an M-file or any other text file. The input can be the absolute path to a text file or the name of an M-file. For example, `type std` displays the contents of the COMSOL Script function `std.m`.

ECHOING THE LINES OF A M-FILE

To enable echoing of the lines that COMSOL Script run in a user-defined function or script, use the `echo` function:

```
echo on
```

turns on echoing of all user-defined M-files.

```
echo off
```

turns off the echoing.

To toggle echoing on and off for the function `myfcn`, type

```
echo myfcn
```

The M-file Path

Any M-files you wish to call must reside in a directory somewhere on the M-file path. To check what the current path is, type

```
path
```

You can also use the `path` function to set the path and to prepend or append directories to the current path. For instance,

```
path('C:\MyCOMSOLFcns', path)
```

adds the directory `C:\MyCOMSOLFcns` to the top of the current path. You can also use the `addpath` function to add directories to the M-file path; the `rmpath` function removes paths.

To get the path string for an entire directory tree, including all subdirectories, use the function `genpath`.

To refresh the view of the path, use the function `rehash`. For each function on the path, `rehash` checks if that function has been modified since it was loaded into memory and reloads it if this is the case. Type

```
rehash path
```

to refresh the view of all directories on the path and load new and modified functions.

If you have created a new function that "shadows" an existing function on the path, you must run `rehash path`.

Precedence Order For M-files, Functions, and Variables

In an M-file path you can place M-file functions, built-in functions, and variables all with the same name. If you type that name at the command prompt, the following precedence order determines which of the variables and functions COMSOL Script calls first:

1. Variables in the current workspace and *inline functions* (see "Inline Functions" on page 135)
2. Built-in functions
3. Local functions (those inside a function; see "Local Functions (Subfunctions)" on page 129)
4. Functions in the current directory
5. Functions elsewhere on the path (if there is more than one, the function that appears first in the M-file path takes precedence)

This means that variables have the highest precedence, so you must clear any variable with the same name as a function. Also, you cannot write a function with the same name as a built-in function (most of the functions in the COMSOL Script libraries are built-in functions). To make COMSOL Script run a built-in function even if there is a variable with same name, use the `builtin` command. For example,

```
d = builtin('det',[1 6; 4 8])
```

computes the determinant of a matrix even if a variable `det` exists in the current workspace.

To check which function COMSOL Script runs, use the `which` function:

```
which fft

fft is a built-in function.
```

The `exist` function returns one of the following integers when you call it with the name of a variable or function:

- 0 if no variable or function with that name exists
- 1 if it is a variable
- 2 if it is a file

- 5 if it is a built-in function
- 7 if it is a directory
- 8 if it is a Java class

```
exist fft

ans =

5
```

Retrieving the Name of the Running M-File

Use the function `mfilename` to get the name of the running M-file (script or function). It is only useful to call this function in an M-file. From the COMSOL Script command window, a call to `mfilename` returns an empty string.

Encrypting M-files

You can protect the contents of custom M-files using the `encrypt` command, which creates encrypted versions of the files in the input arguments. The input files must exist and be valid M-files. For each input file, COMSOL Script creates an MC-file in the current directory (extension `.mc`). When you run an MC-file, it is equivalent to the original M-file, but `encrypt` has scrambled its contents to make it unreadable. Use the optional input argument `-inplace` to make `encrypt` store each MC-file in the same directories as the corresponding M-file.

Scripts

Creating and Running Scripts

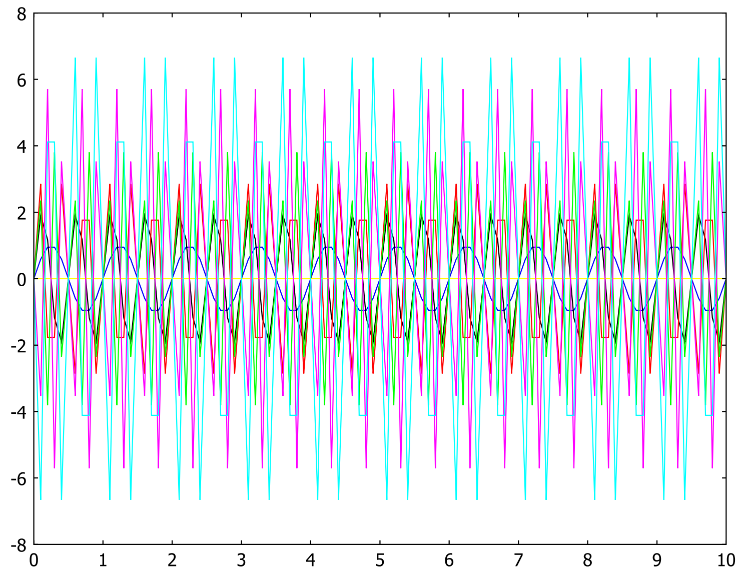
Scripts automate sequences of COMSOL Script commands. A script does not require any special syntax: simply add COMSOL Script statements and comments into an M-file. For instance, the following script plots a number of sine curves of different frequencies:

```
x=0:0.1:10;
colors = 'krgymcbkr'
plot(x,sin(2*pi*x));
hold on
for k=2:7
  plot(x,k*sin(2*pi*k*x),colors(k-1));
end
hold off
```

If you save this command sequence in an M-file called `sinescript.m` in the current directory or somewhere on the M-file path, you can then call it from the COMSOL Script command prompt at any time by just typing its name:

```
sinescript
```

This command then produces this plot:



In addition, it creates the variables `x`, `colors`, and `k`, which remain in the main workspace. This modification of the workspace contents is something that you can take advantage of by writing scripts that create workspace data, for example, by reading data from files.

You can also execute a script with the `run` function as in

```
run sinescript
```

This command also works when you put the name of the script in a string variable:

```
script = 'sinescript';
run(script)
```

Running Scripts in Batch Mode

It is possible to run COMSOL Script files in batch mode without using the command window. Instead, you invoke the scripts from the operating system command prompt with the `comsol` command:

```
comsol batch myscript
```

runs the script in `myscript.m` in batch mode. The script cannot contain any interactive commands. Plots work only if you have an active display, but the `comsol` command quits after running the script, so plots do not remain on the screen and you lose any data that the script does not save to file. Running scripts in batch mode can be useful, for example, for processing of large amounts of data to and from data files.

Note: In Windows, use `comsolbatch.exe` or `comsolbatch64.exe` instead of `comsol batch`; for example, `comsolbatch myscript` to run the script `myscript.m`.

Functions

Functions can extend the COMSOL Script language with custom commands that you call just like built-in functions (in fact, some of the functions in COMSOL Script are M-files just like those you add yourself).

Functions are M-files that typically accept one or more input arguments and provide results in one or more output arguments. The code in a function operates in a separate *function workspace*, so the variables in a function do not appear in the main workspace.

Syntax for Function M-Files

For an M-file to work as a function, it must contain certain components: a definition, help text, and a body.

THE FUNCTION DEFINITION

The function definition is the first line in the M-file and starts with the keyword `function`:

```
function f = fibonacci(n)
```

The function arguments include the following:

- The output arguments, which in this case is only one: `f`. If you have more than one output argument, enclose them in brackets and separate them by commas:

  ```
  function [a,b] = myfcn(c)
  ```

  If the function has no output arguments, the output can be blank:

  ```
  function drawthis(data)
  ```

- The function's name, which should be the same as the name of the M-file (if they differ, the name of the M-file invokes the function)

Note: Names of functions and variables are case sensitive. We recommend using all lowercase letters for function names.

- The input arguments, enclosed in parenthesis. Typically, a function has at least one input argument.

Note: You can create functions that work with a variable number of input and output arguments (see "Variable Number of Input and Output Arguments" on page 132).

ONLINE HELP TEXT

The second line in the M-file, immediately after the function definition, is a comment line (starting with `%`) that serves as a 1-line summary of the function. For example:

```
function f = fibonacci(n)
%FIBONACCI Compute the Fibonacci number.
```

This line appears when you search for functions using `lookfor`.

Immediately after this line you can add comment lines that acts as help text:

```
function f = fibonacci(n)
%FIBONACCI Compute the Fibonacci number.
%   y = fibonacci(n) computes the n:th Fibonacci number.
%   A Fibonacci number is the sum of the two previous
%   Fibonacci numbers. The first and second Fibonacci numbers
%   are 1.
```

This text appears when you type `help fibonacci` at the command prompt.

FUNCTION BODY AND ADDITIONAL COMMENTS

The Help text ends when the function body starts. It can start with a blank line. Any comment lines appearing after a line of code or an empty line is considered an internal comment that the program does not display. In the function body, you can mix lines of code, comment lines, and empty lines.

A Short Example—Fibonacci Numbers

As a short example of a function, write one that returns the n^{th} Fibonacci number. The function has one input argument, `n`, for the Fibonacci number to compute, and one output argument, `f`, the corresponding Fibonacci number.

The definition of a Fibonacci number F_n is

$$F_n = F_{n-1} + F_{n-2}$$

for $n = 3, 4, \ldots$ with $F_1 = F_2 = 1$.

Saving the following code in `fibonacci.m` implements a function `fibonacci`:

```
function f = fibonacci(n)
%FIBONACCI Compute the Fibonacci number.
%   y = fibonacci(n) computes the n:th Fibonacci number.
%   A Fibonacci number is the sum of the two previou
%   Fibonacci numbers. The first and second Fibonacci numbers
%   are 1.
y = [1 1];
for i=3:n
  y(i) = y(i-1)+y(i-2);
end
f = y(n);
```

This function computes all n Fibonacci numbers in `y` and returns the n^{th} one in `f`:

```
f = fibonacci(8)

f =

21
```

ALTERNATIVE SOLUTION USING RECURSIVE FUNCTION CALLS

It is possible for a function to call itself in *recursive* function calls. Because Fibonacci numbers are a sum of the two previous Fibonacci numbers, this case provides an example of where recursive function calls can be useful:

```
function f = fibonacci(n)
%FIBONACCI Compute the Fibonacci number.
% y = fibonacci(n) computes the n:th Fibonacci number.
% A Fibonacci number is the sum of the two previou
% Fibonacci numbers. The first and second Fibonacci numbers
% are 1.
if n<3
  f = 1;
else
  f = fibonacci(n-1)+fibonacci(n-2);
end
```

Note: Make sure that the chain of recursive function calls is limited. Otherwise you could reach a recursion limit and the function will not run to completion.

Local Functions (Subfunctions)

A function can contain other functions, which are *local functions* or *subfunctions* that only the code in the main function can invoke. They are not available outside of the main function. All subfunctions must reside in the same file as the main function, so a function with two subfunction has the following structure:

```
functin y = main_function(a,b)
…
t = sub_function1(a);
…
s = sub_function2(b);
…
y = t+s;
…
function y = sub_function1(a)
…
function y = sub_function2(a)
…
```

where … indicates additional code in the main function and the subfunctions.

Calling Functions

The calling syntax for functions is:

```
output = functionname(input1, input2, ...)
```

with one output argument, and

```
[output1, output2, …] = functionname(input1, input2, ...)
```

with multiple output arguments (notice that you must surround the variables for the output arguments in brackets). COMSOL Script passes the input variables by value.

It is possible to call a function with fewer input arguments than the ones that it defines (a function should support this capability or otherwise issue an error message). If you provide more than the maximum number of input arguments, an error occurs (see "Variable Number of Input and Output Arguments" on page 132 for information about supporting a variable number of input arguments).

ALTERNATIVE FUNCTION CALL SYNTAX

For functions that take strings as inputs, an alternative calling syntax is:

```
functionname input1 input2 …
```

Using this syntax, COMSOL Script interprets the input arguments as string literals. No output arguments are allowed. This syntax is useful with commands such as `help`, `save`, `load`, `disp`, and `clear`.

For example,

```
load mydata -ascii
```

is similar to

```
load('mydata','-ascii')
```

and is easier to type.

There are two situations where you must use the standard function-calling syntax:

- If the input is stored in a string:

```
i = 2;
datafile = ['mydata', int2str(i)];
load(datafile,'-ascii')
```

- If you want to store the output in a variable:

```
s = load('mydata','-ascii');
```

Working With Function Arguments

A number of functions can help you work with a variable number of input and output arguments.

CHECKING THE NUMBER OF INPUT AND OUTPUT ARGUMENTS

Use the `nargin` and `nargout` functions to check the number of input and output arguments, respectively, in a call to a function.

`nargin` makes it possible to branch the code based on the number of input arguments or to supply default values if the function call does not provide all the input variables:

```
function f = inputchk(x,y,z)

switch nargin
case 1
  error('inputchk must have at least 2 input arguments.');
case 2
  z = 0;
end
f = x+y+z;
```

You can also use `nargchk` to check the range for the number of input variables. It takes the lower bound, upper bound, and actual number of input arguments and returns an error message if the actual number is not within these bounds. The following code checks that there are either 2 or 3 input arguments and displays an error otherwise:

```
msg = nargchk(2,3,nargin);
if ~isempty(msg)
```

```
  error(msg)
end
```

Using nargout makes it possible to supply the output variables that the function call asks for. For example, a plot function can return a handle to the graphical object only if the user calls it with an output argument h. The following code snippet placed at the end of such a function uses nargout to accomplish this task:

```
if nargout==1
  h = h_object;
end
```

Here h_object is a local variable that contains the handle.

You can check the range for the number of output variables using nargoutchk in the same way as nargchk works for input arguments.

Note: For functions with output arguments, it is important that the function assigns a value for each of them in all branches where the function call returns. Otherwise, an error occurs because of an unassigned output variable.

VARIABLE NUMBER OF INPUT AND OUTPUT ARGUMENTS

Instead of providing a large number of input and output arguments to cover all possible cases, the varargin and varargout functions allow a variable number of input and output arguments. You can combine both functions with conventional input and output arguments, but they must appear last in the argument list.

Both varargin and varargout are cell arrays, where each cell contains one argument. To unpack them, loop over all cells:

```
function variableargs(varargin)
for k = 1:length(varargin)
  a1 = varargin{1};
  a2 = varargin{2}
...
end
```

The unpacking requires knowledge about what the contents of the different input arguments should be. A common use of varargin is to handle multiple input arguments in a call to another function from within the function that you write. For example, the following function draws a line plot of a mathematical function in the interval between 0 and 1, and it then accepts additional property names and property values that control the line's appearance:

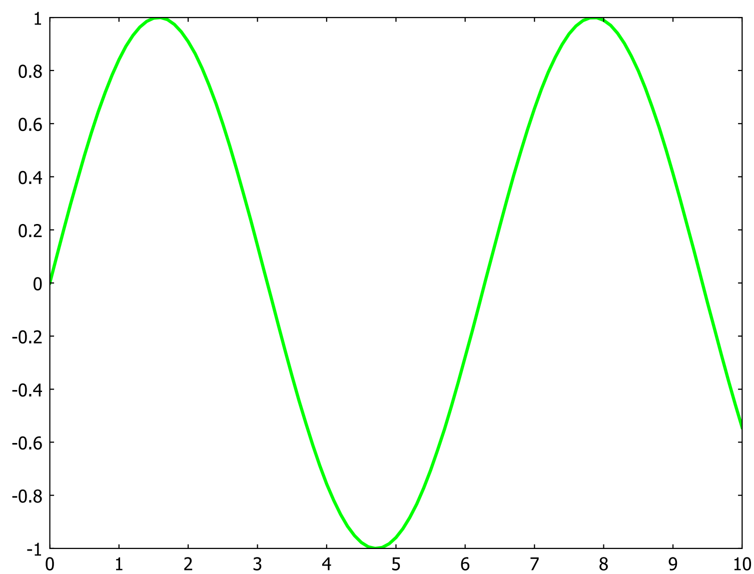
```
function h = fcnplot(fcn,varargin)
x=0:0.1:10;
y = feval(fcn,x);
cla
hl=line(x,y,varargin{:});
if nargout==1
  h = hl;
end
```

(See "Evaluating Functions" on page 135 for more information about `feval`.)

The call to `line` includes the *x* and *y* data plus additional comma-separated input arguments through `varargin{:}`. The call

```
fcnplot('sin','color','g','linewidth',3)
```

produces this plot:



The following code provides an example of how to use `varargout`:

```
function [varargout]=variableoutputs(input)
for k = 1:nargout
  varargout{k} = input(k);
end
```

This routine works if `input` contains a numeric vector of the same length as the number of output arguments. Each output argument then contains a scalar numeric value from the array.

CHECKING THE NAMES OF VARIABLES IN INPUT ARGUMENTS

To get the name of an input to a function, use the `inputname` function:

```
name = inputname(2);
```

returns the name of the variable that is used as the second input argument to the function that currently runs. If the input does not map to a variable in the calling workspace, `inputname` returns an empty string.

SUMMARY OF FUNCTION-ARGUMENT FUNCTIONS

The following table summarizes the functions for working with input and output arguments

TABLE 7-1: FUNCTION-ARGUMENT FUNCTIONS

FUNCTION NAME	DESCRIPTION
`inputname`	Get the name of an input to a function
`nargchk`	Check that the number of arguments supplied to a function is in a specified range
`nargin`	Number of input arguments
`nargout`	Number of output arguments
`nargoutchk`	Check that the number of outputs expected from a function is in a specified range
`varargin`	Retrieve arguments to a function with variable number of input arguments
`varargout`	Set outputs from a function with a variable number of outputs

Updating and Locking Functions

To remove all user-defined functions from the workspace to make sure that COMSOL Script runs an updated version, type

```
clear functions
```

To prevent this command from clearing a certain function, use the `mlock` function:

```
mlock fibonacci
```

It locks the function `fibonacci` in memory so that the `clear functions` command does not remove it. To remove the function lock, type

```
munlock fibonacci
```

Use `mislocked` to check if a function is locked in memory.

Locking a function can be useful to prevent the `clear` command from removing it. Locking a function prevents any persistent variables defined in the file from getting reinitialized.

Evaluating Functions

The `feval` command evaluates functions using a string with the function name as the first input argument. It is useful if the function name comes from a file or as an input argument. The following code snippet asks the user for a function name and then calls it with one input argument. The code assumes that this is a function of one variable that operates pointwise on a vector, as are most math functions in COMSOL Script.

```
fcn = input('Type a function name:','s');
y = feval(fcn,0:0.1:10);
```

Inline Functions

Inline functions provide a way to create functions based on an expression that you give as a string:

```
scfun = inline('sin(2*pi*x).*cos(y.^2)');
```

Then call `scfun` with two input variables, `x` and `y`:

```
scfun(0.25,0)

ans =

1
```

You can place explicit input arguments in the call to `inline`, but COMSOL Script finds the inputs as the identifiers that you can also find with the function `symvar`:

```
c=symvar('sin(2*pi*x).*cos(y.^2)')

c =

  'x'
  'y'
```

`symvar` ignores the following common identifiers: `eps`, `i`, `inf`, `Inf`, `nan`, `NaN`, and `pi`.

To check the argument names for an inline function, use `argnames`.

To get the expression for an inline function, call `formula`, which returns it in a string.

A useful function when working with expressions for inline functions is `vectorize`, which makes sure that all multiplication, division, and power operators are the pointwise operators `.*`, `./`, and `.^` instead of `*`, `/`, and `^`, which are matrix operators in COMSOL Script. Consider this example:

```
vectorize('sin(2*pi*x)*cos(y^2)')

ans =

  sin(2.*pi.*x).*cos(y.^2)
```

8

Data Analysis, Statistics, and I/O

This chapter describes the data-analysis capabilities, statistics functions, signal-processing tools, and I/O features in COMSOL Script. Using these functions, you can read data and perform statistical and other types of analysis. The chapter also contains information about date and time functions and functions.

Data-Analysis Overview

COMSOL Script provides many functions for data analysis and statistics:

- Statistical functions for the computation of minima and maxima, mean values, medians, and other statistical measures as well as sorting data and making histogram counts
- Signal-processing tools such as the FFT and a general function for 1D digital filtering
- Functions for interpolation, triangulation, and polynomials
- Numerical differentiation and integration routines

Statistical Analysis

COMSOL Script provides many basic functions for statistical analysis and sorting.

Computing Minimum and Maximum Values

Compute minimum and maximum values for any numerical array using the functions `min` and `max`:

```
a = rand(1,5)

a =

   0.132797   0.633481   0.136563   0.519387   0.242814

max(a)

ans =

0.633481

min(a)
ans =

0.132797
```

For complex data, `min` and `max` use the magnitude only and ignore the phase:

```
max(2-2i,2+i)

ans =

2 - 2i
```

This example also illustrates the syntax `min(A,B)` and `max(A,B)`, where the functions return the largest or smallest of `A` and `B`, both matrices that must be of the same size (or scalar, which COMSOL Script expands).

For a matrix, `min` and `max` return a row vector containing the min or max of each column in the matrix. For multidimensional arrays, they return the min/max along the first nonsingleton dimension of the array. You can also use the syntax `min(A,[],dim)` and `max(A,[],dim)` to take the min and max along the dimension in `dim`:

```
A = rand(3,1,2)

A(:,:,1) =
```

```
     0.873289
     0.750284
     0.841772

A(:,:,2) =

     0.297192
     0.763758
     0.751643

max(A)

ans(:,:,1) =

     0.873289

ans(:,:,2) =

     0.763758

max(A,[],2)

ans(:,:,1) =

     0.873289
     0.750284
     0.841772

ans(:,:,2) =

     0.297192
     0.763758
     0.751643

max(A,[],3)

ans =

     0.873289
     0.763758
     0.841772
```

In this case, 1 is the first nonsingleton dimension, so `max(A,[],1)` is the same as `max(A)`.

OBTAINING THE INDEX TO THE MINIMUM OR MAXIMUM VALUE

By adding an output argument in the calls to `min` and `max`, you get the indices for the smallest or largest elements:

```
x = [1 9 2 3 4 5];
[xmax, i] = max(x)

xmax =

9


i =

2
```

Computing Mean and Median Values

Similarly, call `mean` and `median` to compute the mean and median values for an array:

```
A = [1 1:10];
mean(A)

ans =

5.090909

median(A)
ans =

5
```

For a matrix, `mean` and `median` work in the same way as `max` and `mean`.

Computing Standard Deviations, Variances, and Correlations

STANDARD DEVIATION AND VARIANCE

To compute the standard deviation, use the `std` function. Specifically, `y = std(X)` and `y = std(X,0)` compute the standard deviation of `X`, normalizing `Y` by $N-1$, where N is the sample size. Similarly, `y = std(X,1)` computes the standard deviation of `X`, normalizing `y` by N.

You can also employ a weight vector `w` to compute the standard deviation. Here, `y = std(X,w)` computes the standard deviation of `X` using the weight vector `w`, which `std` normalizes to sum to one. Note that `w` must contain only nonnegative elements and must be of the same length as `X` along the dimension for which `std` computes the standard deviation.

For a matrix, `std` returns a row vector containing the standard deviation of each column in the matrix. For a multidimensional array, `std` returns the standard deviation along the array's first nonsingleton dimension. Use the syntax `std(A,w,dim)` to compute the standard deviation along the dimension in `dim`.

The `var` function computes the variance, and it has the same syntax and input arguments as `std`. For the same input, `std(A)` is the same as `sqrt(var(A))`.

COVARIANCE MATRICES

Use the `covar` function to compute the covariance matrix for a matrix `X`:

```
X = [0 1 2; 2 3 4;1 0 3];
c = cov(X);
v = diag(c)';
v1 = var(X);
```

As you can se if you run this script, the diagonal of `c` contains the variance of each column of `X`. The `cov` function uses normalization by $M–1$, where M is the number of observations, and it also removes the mean from each column before calculation.

CORRELATION COEFFICIENTS

If you want to compute correlation coefficients, use the `corrcoef` function. The input X is a matrix where each row is an observation and each column a variable. The resulting correlation coefficients matrix R is a matrix such that each element

$$R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}}$$

where C is the covariance matrix of the input X. Using an optional second output argument, `corrcoef` can also return the p-value, which represents the probability of getting a correlation as large as the observed value, given that the null hypothesis is true. Optional third and fourth output arguments contain lower and upper bounds, respectively, for a confidence interval that you can specify. The default setting provides 95% confidence intervals.

If the input data contains NaNs, which can represent missing data (see "Handling NaNs and Missing Data" on page 145 for more information), you can control how `corrcoef` treats these NaNs using the property `rows`. The string `all` (the default) means that all data is used, resulting in NaNs in the output. If you provide the string `complete`, the function ignores all rows that contain NaNs. The string `rows`, finally, makes `corrcoef` use rows with no NaNs in column i or j to compute R_{ij}.

Computing Sums and Products

COMSOL Script includes functions for both standard sums and products of the elements in an array and accumulated sums and products.

The function `sum` computes the sum of an array, while `prod` computes its product. For a matrix, the output of `sum` and `prod` is a row vector containing the sum or product of each column in the matrix. For a multidimensional array, the output is the sum or product along the first nonsingleton dimension. Using `sum(X,dim)` or `prod(X,dim)` returns the sum or product of `X` along the dimension in `dim`.

The functions `cumsum` and `cumprod` compute the accumulated sum and the accumulated product, respectively. The output is the same size as the input and contains the cumulative sum or cumulative product along the first nonsingleton dimension of the input array. Use `cumsum(X,dim)` and `cumprod(X,dim)` to compute the cumulative sum and cumulative product of the elements along dimension `dim` of the input array `X`.

Some examples:

- Compute the factorial of 10 (10!):

  ```
  prod(1:10);
  ```

  and all factorials from 1! to 10!:

  ```
  cumprod(1:10)
  ```

  (You can also use the `factorial` function to compute factorials.)

- Let the 3×12 array `A` contain sales data from three offices during each month of a year:

  ```
  A=round(1000*rand(3,12))
  A =

    471   415   522   154   171   203   981   998   591   533   602   768
    767   119   107   190   656   726   306   134   447   605   635   338
    968   909   369   600   152   129   455   792   663   238   496   950
  ```

  Then, to compute the total sales for each month:

  ```
  sum(A)

  ans =

    2206   1443    998    944    979   1058
    1742   1924   1701   1376   1733  2056
  ```

  To compute the accumulated total sales for each month and each office type:

```
cumsum(A)
```

The accumulated total sales for each month is:

```
cumsum(sum(A))
```

The sum total of all sales during the year is:

```
sum(sum(A))
```

The total sales during the year for each office is:

```
sum(A,2)
```

Sorting Data

Use the `sort` function to sort array elements in ascending order. If the data is in a matrix, the function sorts each column. For a multidimensional array, it sorts along the first nonsingleton dimension. Further, `y = sort(x,dim)` sorts `x` along the dimension in `dim`.

To get the original index of the elements, add an extra output argument:

```
[y,ind] = sort(x);
```

`ind` is then an array of the same size as `x` containing the original index of each element in `y` along the dimension in which `X` is sorted.

If the input data contains complex values, `sort` sorts them first by magnitude and then by angle.

To sort rows, the function `y = sortrows(x)` sorts the rows of `x` in ascending order (`x` must be a matrix or a column vector). Using a second input argument, `y = sortrows(x,col)`, sorts the rows of `x` according to the columns specified in `col`, which must be a vector of positive integers, where each entry specifies one column.

`sortrows` behaves like a dictionary sort:

```
A=strvcat('Azimuth','Aluminum','Aluminium','Alumina','Alligator')
A =

  Azimuth
  Aluminum
  Aluminium
  Alumina
  Alligator

char(sortrows(abs(A)))

ans =
```

```
  Alligator
  Alumina
  Aluminium
  Aluminum
  Azimuth

char(sortrows(abs(A),7))

As =

  Alumina
  Azimuth
  Aluminium
  Alligator
  Aluminum
```

The last call to `sortrows` in this example made it sort along Column 7.

Handling NaNs and Missing Data

When an NaN appears in an array, it can represent missing or non-numeric data. In COMSOL Script, an NaN propagates through computations because the output from most functions is NaN if the input contains an NaN. So, to do statistical measures on data with NaNs, you must first remove them and then make the computations on the remaining data.

Consider finding the mean value of the columns in the matrix `A`:

```
A = [1 2 NaN 3 4; 5 NaN 6 NaN NaN; NaN 7 8 9 10]'

A =

   1     5     NaN
   2     NaN   7
   NaN   6     8
   3     NaN   9
   4     NaN   10

mean(A)

ans =

   NaN   NaN   NaN
```

Instead of using `mean` directly, use a loop that removes the NaNs from each column:

```
for i=1:size(A,2)
  col = A(:,i);
```

```
    col = col(~isnan(col));
    mn(i) = mean(col);
  end

  mn

  mn =

     2.500000   5.500000   8.500000
```

The need for the loop arises because the columns of data vary in size after the removal of the NaNs. In the same way you could remove or replace data that, for example, represents outliers in a data set.

For information about how the function `corrcoef` treats NaNs when computing correlation coefficients, see "Correlation Coefficients" on page 142.

Summary of Functions for Statistical Analysis

The following table summarizes the functions for statistical analysis:

TABLE 8-1: FUNCTIONS FOR STATISTICAL ANALYSIS AND SORTING

FUNCTION NAME	DESCRIPTION
`corrcoef`	Compute correlation coefficients
`cov`	Compute the covariance matrix
`cumprod`	Compute the accumulated product of array elements
`cumsum`	Compute the accumulated sum of array elements
`histc`	Histogram count
`max`	Compute the maximum value
`mean`	Compute the mean value
`median`	Compute the median value
`min`	Compute the minimum value
`prod`	Compute the product of array elements
`sort`	Sort data in an array
`sortrows`	Sort rows
`std`	Compute the standard deviation
`subspace`	Compute the principal angle between subspaces
`sum`	Compute the sum
`var`	Compute the variance

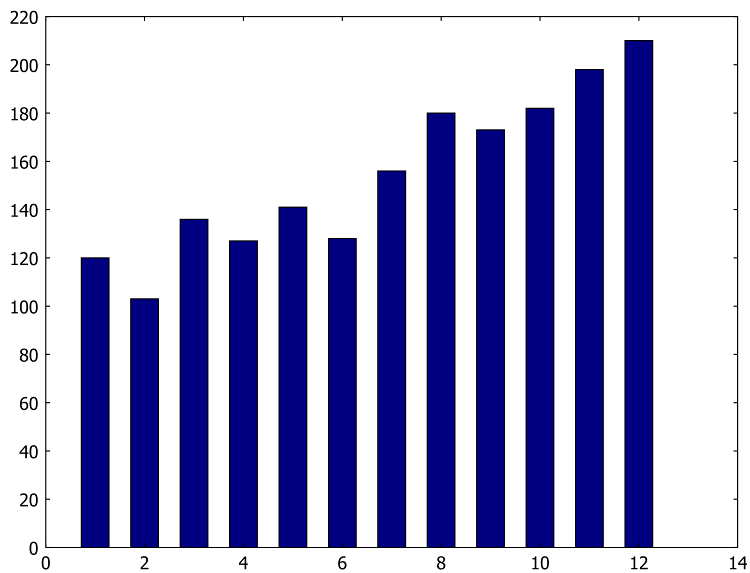
Data Analysis Plots

Bar Graphs

The function `bar` creates a bar graph based on the input data. For example,

```
sales = [120; 103; 136; 127; 141; 128; 156; 180; 173;
          182; 198; 210];
months = 1:12;
bar(months,sales)
```

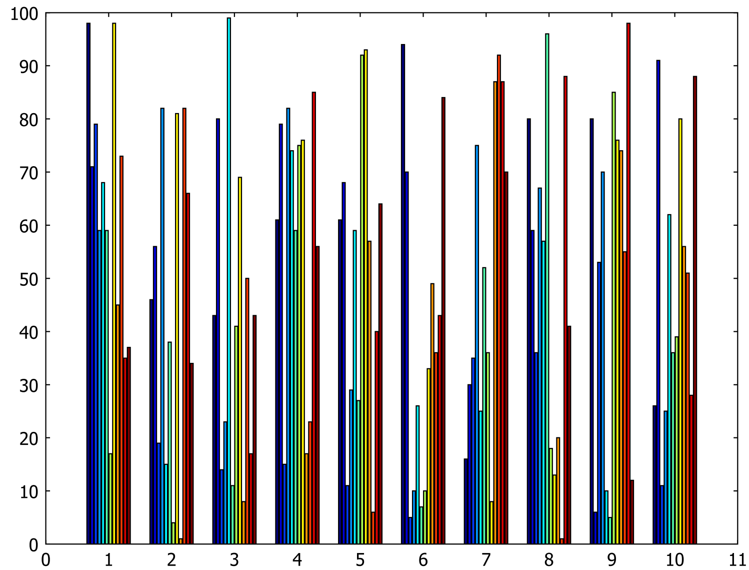
creates the following bar graph:



You can also create groups of bars using a matrix as the second input argument:

```
sales = round(100*rand(10,12));
years = 1:10;
bar(years,sales)
```

Each group of 12 bars in the following plot represents the monthly sales during one year

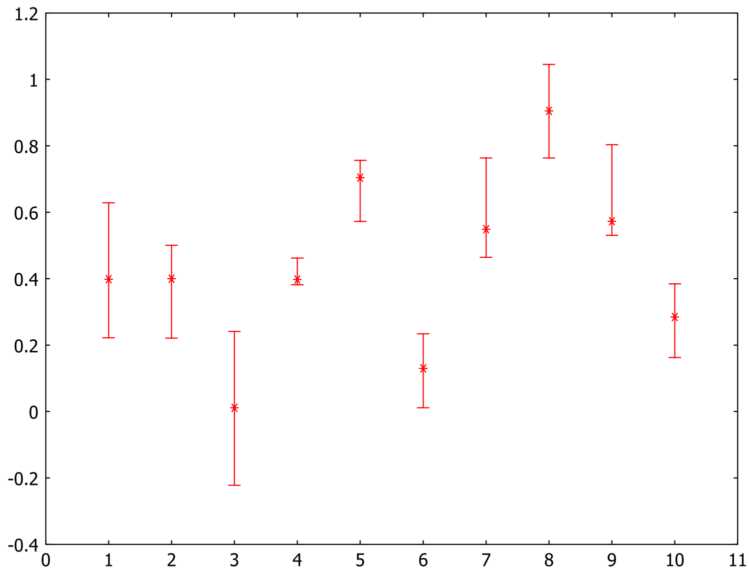


Using additional input arguments, you can stack the bars and change their color and relative width.

Error Bars

You can add error bars to plots of data using the `errorbar` function. Either provide both a lower and an upper error range or a single error range that applies to both the lower and upper error ranges. In addition, you can supply a string that controls the line style and color (see "The Plot Command" on page 205 for details) and also additional property name and property value pairs for the line object. As an example, show some random data with random lower and upper error ranges using red asterisks to indicate the data points:

```
x = 1:10;
y = rand(size(x));
low = 0.25*rand(size(x));
high = 0.25*rand(size(x));
errorbar(x,y,low,high,'r*');
```



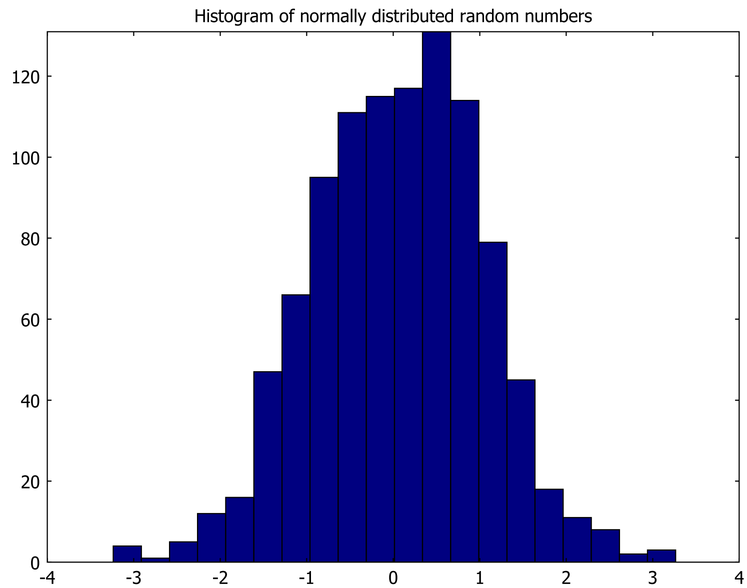
Histograms

To sort data into bins and plot them as a histogram, use the `hist` function.

```
r = randn(1,1000);
n = hist(r,20);
```

creates 1000 normally distributed random numbers and uses `hist` to divide them into 20 bins (the default is 10 bins) of equal size. The output `n` is a 1×20 row vector that contains the number of elements in `r` that falls into each bin. To plot this as a histogram, call `hist` without output arguments:

```
hist(r,20)
title('Histogram of normally distributed random numbers');
```



The function `histc` stores data in bins specified by a second input argument:

```
n = histc(x,edges)
```

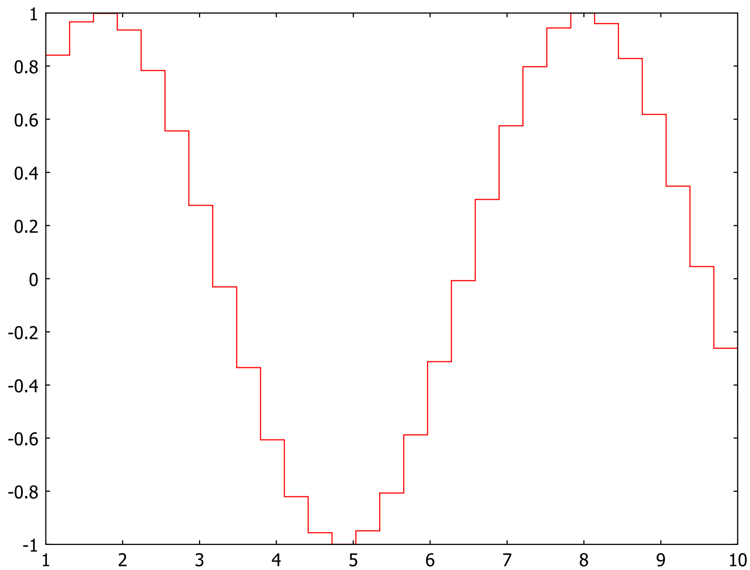
where `edges` is a vector containing monotonically nondecreasing values so that `x(i)` falls in bin `k` if `edges(k) <= x(i) < edges(k+1)`. At the end, the last bin includes the values in `x` that match it exactly, so to include all values (except NaNs) use `-inf` and `inf` at the ends of `edges`.

Stairstep Plots

The function `stairs` creates a stairstep graph based on the input data. For example,

```
x = linspace(1,10,30);
y = sin(x);
stairs(x,y,'r')
```

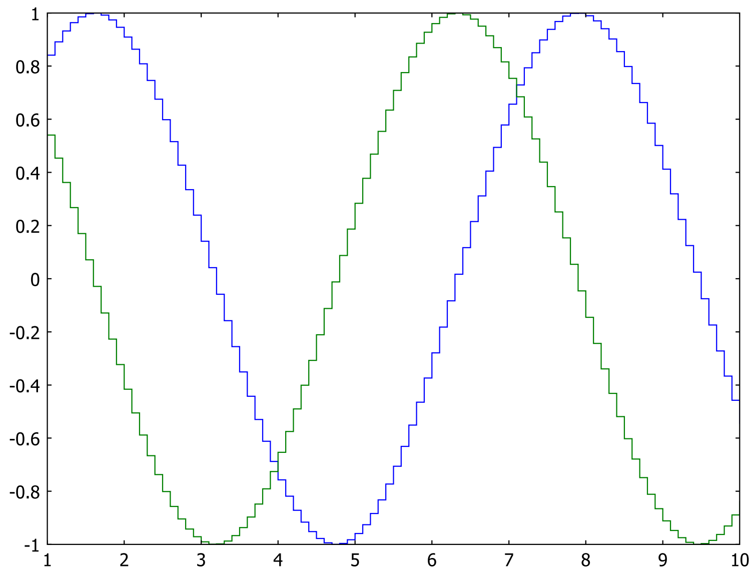
creates the following stairstep graph:



You can also create groups of stairs using matrices as input arguments:

```
x = 1:0.1:10;
y1 = sin(x); y2 = cos(x);
stairs([x(:),x(:)],[y1(:),y2(:)]);
```

This plots the sine and cosine functions as stairstep graphs next to each other:



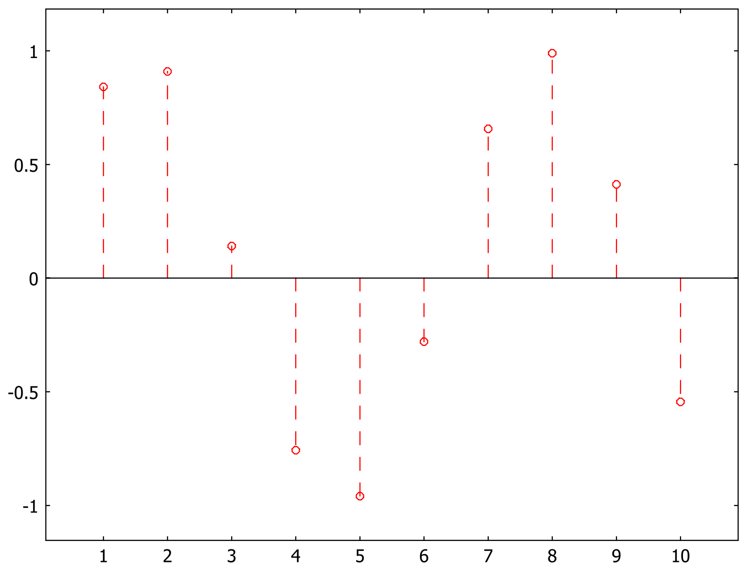
Using additional input arguments, you can change the properties of the graph, such as line width, color etc.

Stem Plots

The functions `stem` and `stem3` create a stem graphs based on the input data. For example,

```
x = 1:10;
y = sin(x);
stem(x,y,'r--');
```

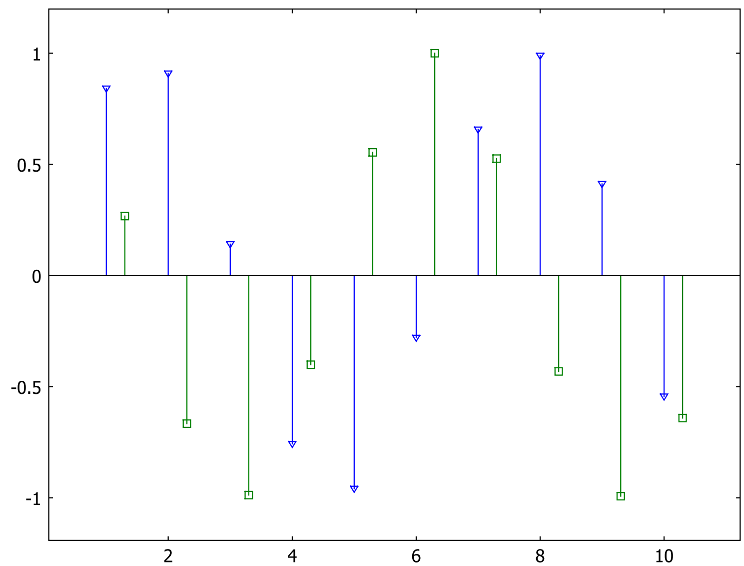
creates the following graph:



You can also create groups of stems using a matrices as input arguments:

```
x1 = 1:10; x2 = x1+0.3;
y1 = sin(x1); y2 = cos(x2);
h=stem([x1(:),x2(:)],[y1(:),y2(:)], 'marker','cycle');
```

This plots the sine and cosine functions as stem graphs next to each other, using different markers:



Using additional input arguments, you can change the properties of the graph, such as line width, color etc.

Summary of Functions for Data Analysis Plots

The following table summarizes the functions for statistical analysis:

TABLE 8-2: FUNCTIONS FOR STATISTICAL ANALYSIS AND SORTING

FUNCTION NAME	DESCRIPTION
`bar`	Bar graph plot
`errorbar`	Error bar plot
`hist`	Calculate histogram data or plot histogram
`stairs`	Stairstep plot
`stem`	Stem plot in 2D
`stem3`	Stem plot in 3D

Signal-Processing Tools

COMSOL Script contains two signal-processing tools:

- The fast Fourier transform (FFT) and inverse fast Fourier transform (IFFT)
- A general function for 1D digital (discrete) filtering.

Using the FFT Functions

The FFT algorithm computes the discrete fast Fourier transform of a sequence of input data. It is an extremely common numerical algorithm in many signal-processing applications such as filtering and power-spectrum estimations.

COMSOL Script uses the `fft` function to compute the FFT of a vector or a matrix. For a matrix, it computes the FFT of the columns. Additionally, 2D and n-dimensional FFT routines (`fft2` and `fftn`) are also available. To calculate the inverse FFT, select one of the functions `ifft`, `ifft2`, and `ifftn`.

EXAMPLE OF AN FFT FOR THE SINE FUNCTION

The sine function

$$y = \sin(2\pi k_0 x)$$

has the Fourier transform:

$$\frac{1}{2}i[\delta(k+k_0) - \delta(k-k_0)]$$

To show this using COMSOL Script, run the following code:

```
x = -10:0.1:10;
k0 = 0.2;
y = sin(2*pi*k0*x);
subplot(3,1,1); plot(x,y);
title('The sine wave')
yf = fft(y);
X = -5:1/20:5;
subplot(3,1,2); plot(X,fftshift(abs(yf)));
title('The Fourier transform of a sine wave')
subplot(3,1,3); plot(x,ifft(yf));
title('The sine wave computed using ifft')
```

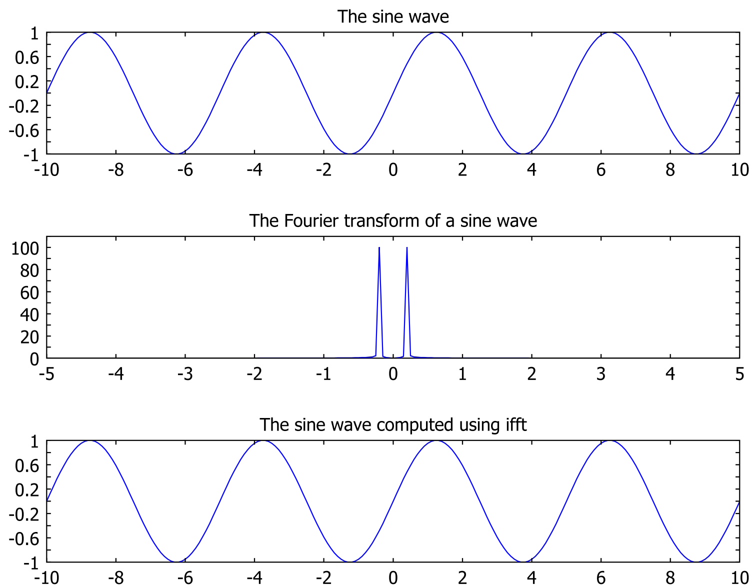


Figure 8-1: The original sine wave (top), the FFT of the sine wave (middle), and the reconstruction of the original sine wave from the inverse FFT of the FFT of the input data (bottom).

The function `ifft` shifts the frequency spectrum so that the 0 frequency point appears in the middle, showing the peaks in the Fourier transform at k_0 and $-k_0$.

Type `yf` to see that the FFT of the sine wave contains complex data (the plot shows only the real part).

Using the Digital Filter Function

`filter` is a general-purpose function for performing 1D digital filtering. `y = filter(b,a,x)` works with a filter that is a *direct form II transposed* implementation of the standard difference equation:

$$a_1 y_n = b_1 x_n + b_2 x_{n-1} + \ldots + b_{nb+1} x_{n-nb} - a_2 y_{n-1} - \ldots - a_{na+1} y_{n-na}$$

It implements a filter of order n–1. By specifying the a and b coefficients as vectors in the inputs `a` and `b`, you can realize different filter types, discrete transfer functions, or general difference equations by indentifying the number of a and b coefficients and their values. If a_1 is not equal to 1, `filter` normalizes all other coefficients using a_1.

EXAMPLE OF FILTER FOR A FIRST-ORDER STEP RESPONSE

Consider a standard first-order step response:

$$y = \frac{k}{1 + \tau s} u$$

Using a simple difference approximation with the step h, the equivalent discrete transfer function is

$$(h + \tau) y_n = khu_n + \tau y_{n-1}$$

where y_n is the output at time t, and y_{n-1} is the output at $t–h$. The following code implements this equation using the `filter` function and tests it on an input step:

```
h = 0.1;
t = 0:h:10;
u = zeros(size(t));
u(11:end) = 1;
plot(t,u)
k = 1;
tau = 2;
a = [h+tau, -tau];
b = h*k;
y = filter(b,a,u);
hold on;
plot(t,y,'k--')
```

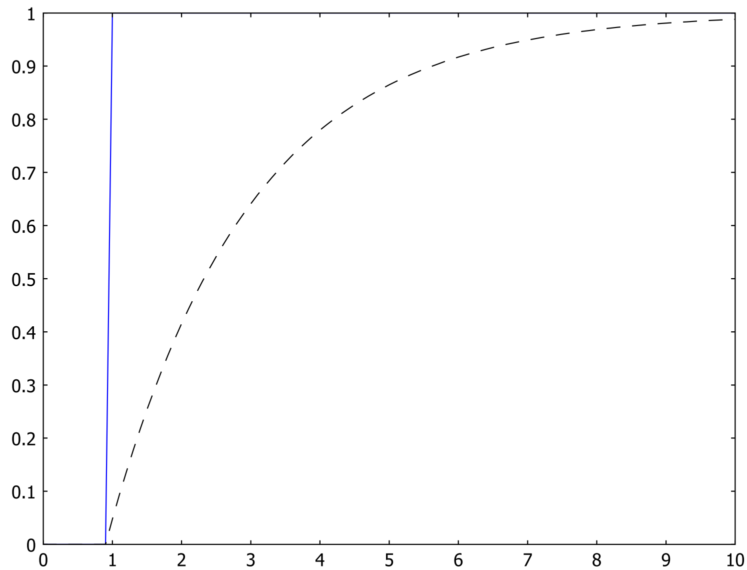


Figure 8-2: Input signal (blue, solid) and simulated first-order step response (black, dashed).

Simulating Discrete-Time State-Space Models

Use the `dlsim` function to simulate discrete-time state-space models on the form

$$x[k+1] = Ax[k] + Bu[k]$$
$$y[k] = Cx[k] + Du[k]$$

The dlsim function computes the output y and the states x with the input signal u and the state-space matrices A, B, C, and D as inputs. You can also provide an optional initial state x_0:

```
[y,x] = dlsim(A,B,C,D,u,x0);
```

Note: The `dlsim` function is a low-level function for state-space simulation. The Signals and Systems Lab contains a comprehensive suite of tools for discrete-time and continuous-time systems modeling and simulation.

Summary of Signal-Processing Functions

The following table summarizes the signal-processing functions in COMSOL Script:

TABLE 8-3: SIGNAL PROCESSING FUNCTIONS

FUNCTION NAME	DESCRIPTION
`dlsim`	Simulate discrete-time state-space model
`fft`	Compute the fast Fourier transform
`fft2`	Compute the 2D fast Fourier transform
`fftn`	Compute the n-dimensional fast Fourier transform
`fftshift`	Shift a frequency spectrum computed using FFT
`filter`	1D digital filtering
`ifft`	Compute the inverse fast Four transform
`ifft2`	Compute the inverse 2D fast Four transform
`ifftn`	Compute the inverse n-dimensional fast Four transform
`ifftshift`	Undo the frequency spectrum shift performed by `fftshift`

Interpolation and Polynomials

This section describes the interpolation, triangulation, and polynomial functions in COMSOL Script.

Interpolating Data

For interpolating data, COMSOL Script provides routines for 1D, 2D, and 3D data: `interp1`, `interp2`, and `interp3`, respectively.

`y = interp1(x,y,xi)` implements a linear interpolation to determine $y = f(xi)$ for $y = f(x)$. The default method is linear interpolation, but you can select another interpolation method by providing one of the strings in Table 8-4 as an extra input argument: `y = interp1(x,y,xi,method)`:

TABLE 8-4: 1D INTERPOLATION METHODS

STRING	DESCRIPTION
`cubic`	Piecewise cubic Hermite interpolation
`linear`	Linear interpolation
`nearest`	Nearest neighbor
`spline`	Cubic spline interpolation

For 2D and 3D interpolation, only `nearest` and `linear` are possible choices. The 2D and 3D interpolations can handle vectors of x, y, and z (3D only) data and also a grid matrix that you can generate with the `meshgrid` function.

EXAMPLE OF 1D INTERPOLATION

Consider interpolation within the following function:

$$y = 3\sin(\pi x)e^{\sin\left(\frac{\pi}{2}x\right)}$$

in the interval between 0 and 1:

```
x = 0:0.1:1;
y = 3*sin(pi*x).*exp(sin(pi/2*x));
```

To interpolate to the values of y at 0.25, $\pi/4$, and 0.75, type:

```
xi = [0.25 pi/4 0.75];
yi = interp1(x,y,xi,'cubic')
```

```
yi =

   3.110360   4.812198   5.343375
```

The exact values that result from the code

```
yi = 3*sin(pi*xi).*exp(sin(pi/2*xi))
```

when rounded to four significant digits, are 3.110, 4.812, and 5.344, and a comparison shows that the spline interpolation does a good job at providing the interpolated values:

EXTRAPOLATING OUTSIDE OF RANGE

The interpolation routines also support extrapolation using an extra input argument that can take the following values:

- The string `'const'` for extrapolating using a constant value
- The string `'extrap'` for extrapolation using the selected interpolation method
- A scalar value, which the interpolation function returns for out-of-range values

The default extrapolation method is `extrap` for `spline` and `cubic`. For other interpolation methods, the interpolation functions return `NaN`s for out-of-range values if you do not provide an extrapolation method. For example, using the same `x` and `y` as in the previous example:

```
xi = 1.5;
yi = interp1(x,y,xi,'cubic')
yi =

-15.839364

yi = interp1(x,y,xi)
yi =

NaN
```

For interpolation using polynomials and splines, see "Interpolation Using Polynomials and Splines" on page 164.

Working With Polynomials

To represent a polynomial in COMSOL Script, it is convenient to work with a row vector of polynomial coefficients. For example, to represent the polynomial

$$17x^4 + 10x^3 - 2x^2 + 5x - 3$$

type

```
p = [17 10 -2 5 -3]
```

The functions that work with polynomials interpret the elements in a row vector as the coefficients in a polynomial, ordered by descending powers (if there were no x^3 term, for example, the second element in the vector would be 0).

Use the function `roots` to find the roots of a polynomial:

```
r =roots(p)

r =

   -1.084828 +        0i
    0.038923 + 0.622059i
    0.038923 - 0.622059i
    0.418746 +        0i
```

so `r(4)` is a positive real root.

Use the function `polyval` to evaluate polynomials:

```
polyval(p,0)

ans =

-3

polyval(p,r(4))

ans =

-8.882e-016
```

The last value is not exactly zero due to limited numerical precision.

You can also integrate and differentiate polynomials using the `polyint` and `polyder` functions.

To multiply and divide polynomials, the convolution and deconvolution functions `conv` and `deconv` are useful:

```
p1 = [1 1];
p2 = [1 -1];
p3 = conv(p1,p2)

p3 =

  1  0  -1
```

```
[p4,r] = deconv(p1,p2)
p4 =

1

r =

   0   2
```

The conv and deconv calls compute $(x+1)(x-1)$ and $(x+1)/(x-1)$, respectively, where the polynomial division yields a remainder of 2 in the variable r.

For 2D and multidimensional convolution of matrices, use the conv2 and convn functions, respectively.

FITTING A POLYNOMIAL TO A SET OF DATA

The function polyfit is available for fitting a polynomial to a set of data. polyfit uses a least squares polynomial fit, so that for a set of x and y values, it computes a polynomial p that minimizes

$$\sum_{i=1}^{n} (p(x_i) - y_i)^2$$

so that $p(x)$ is an approximation of y.

As an example, use a 10th-degree polynomial to approximate the function $y = (1+x)/(1+(5-x)^2)$ in the interval between 0 and 10 and then plot the resulting polynomial (see Figure 8-3):

```
x=0:0.1:10;
y = (1+x)./(1+(5-x).^2);
p = polyfit(x,y,10);
yp=polyval(p,x);
plot(x,y,'b',x,yp,'r*');
```

polyfit can also return a second output variable S, which contains a structure with the fields R (the Cholesky factor of the Vandermonde matrix), df (the degrees of freedom), and normr (the norm of the residuals). You can use it with polyval to compute error estimates D as second output argument from polyval.

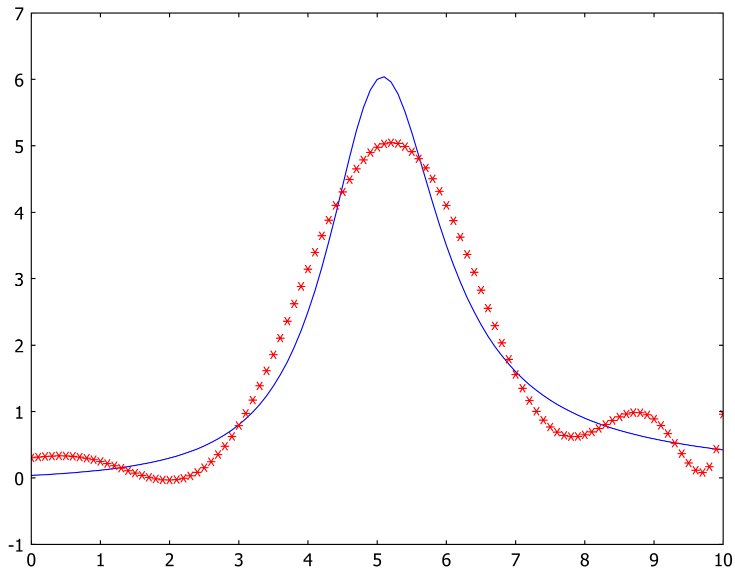


Figure 8-3: The original functions (blue, solid) and the 10th-degree polynomial approximation (red, asterisks)

INTERPOLATION USING POLYNOMIALS AND SPLINES

The `spline` function performs cubic spline interpolation. The basic syntax is:

```
yi = spline(x,y,xi);
```

which provides spline interpolation of `y` at the points in `x` and then returns an array `yi` with the values of `y` at the points in `xi`. You can also use the spline function to perform spline interpolation and return the cubic spline interpolant as a piecewise polynomial structure instead of returning interpolated data. To do so, use the following syntax:

```
pp = spline(x,y);
```

For a piecewise cubic Hermite interpolation, use the `pchip` function, which uses the same syntax as the `spline` function.

You can then reuse and evaluate this piecewise polynomial using the `ppval` function. The following example shows how to first interpolate points from a trigonometric function using both `spline` and `pchip` and then reuse the piecewise polynomial for another interpolation:

```
x = linspace(0,2*pi,10);
y = sin(x).*cos(x);
xi = linspace(0,2*pi,20);
yis = spline(x,y,xi);
yih = pchip(x,y,xi);
pps = spline(x,y);
pph = pchip(x,y);
xi1 = linspace(0,2*pi,100);
yips1 = ppval(pps,xi1);
yiph1 = ppval(pph,xi1);
plot(x,y,'b',xi,yis,'r--',xi,yih,...
     'm-.',xi1,yips1,'go',xi1,yiph1,'k*');
```

The following plot shows the results:

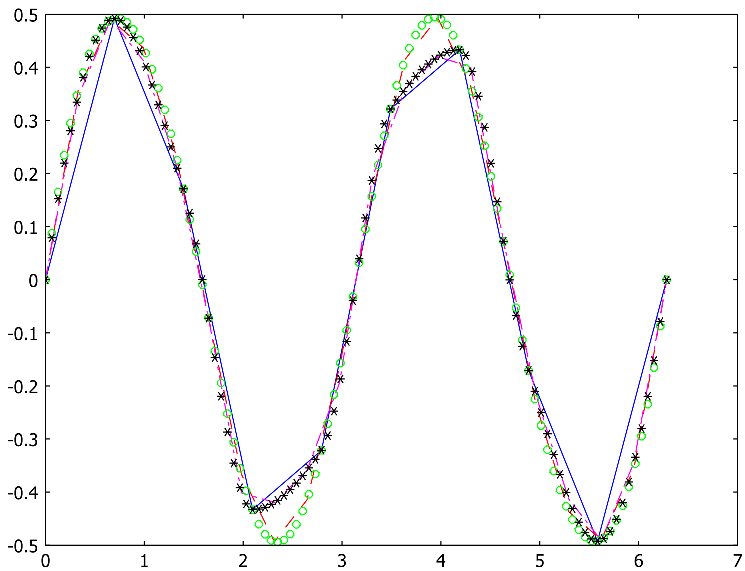


Figure 8-4: Interpolation using splines and cubic Hermite interpolation: original data (blue, solid), coarse spline interpolation (red, dashed), coarse Hermite interpolation (magenta, dash-dotted), fine spline interpolation (green, circles), and fine Hermite interpolation (black, asterisks).

To create piecewise polynomials directly, use the `mkpp` function, which returns a structure representing the piecewise polynomial described by its breaks and coefficients. The breaks is a vector with increasing elements, representing the start and end of each interval, and you provide the coefficients in a matrix where each row contains the coefficients (in order from highest to lowest exponent) of the polynomial for one interval. For example, to create a piecewise polynomial with two polynomial species, $2x^2+3x+5$, and x^2+x+4, on the intervals [1, 2] and [2, 5], respectively, type:

```
b = [1 2 5];
c = [2 3 5;1 1 4];
pp = mkpp(b,c);
```

To extract information from a piecewise polynomial structure, use the `unmkpp` function, which returns the breaks, coefficients, number of pieces, order, and dimension of the piecewise polynomial:

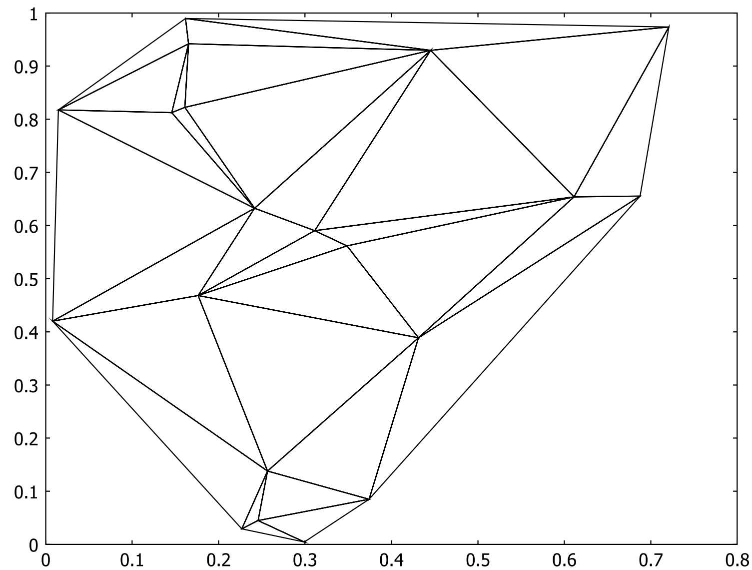
```
[breaks,coefs,pieces,order,dim] = unmkpp(pp);
```

Data Gridding and Triangulation of Point Data

DELAUNAY TRIANGULATION

The functions `delaunay` and `delaunay3` perform Delaunay triangulation of point data in 2D and 3D, respectively. A Delaunay triangulation creates a set of triangles (or tetrahedrons) such that no points are contained in any triangle's (or tetrahedron's) circumcircle (circumscribed circle). As an example, perform a Delaunay triangulation for some random 2D points and plot it as a mesh plot using `trimesh`:

```
x = rand(1,20);
y = rand(1,20);
t = delaunay(x,y);
trimesh(t,x,y);
```



You can also use boundary element information to control the triangulation for geometries that includes topological information about boundaries and subdomains. For more information, see the command-line help and the *COMSOL Script Command Reference* entry for `delaunay`.

The mesh plot function `trimesh` can take height and color data to create a 3D plot. You can also further control the properties of the line object or patch object that `trimesh` creates. For a corresponding surface plot, use the `trisurf` function.

DATA GRIDDING

For data gridding, use the functions `griddata`, `griddata3`, and `griddatan`, which work on 2D data, 3D data, and n-D data, respectively.

```
zi = griddata(x,y,z,xi,yi);
```

The `griddata` function performs a Delaunay triangulation on x and y, where $z = f(x, y)$, and interpolates x_i and y_i linearly to determine $z_i = f(x_i, y_i)$. The points do not need to be uniformly spaced. If the inputs `x` and `y` are not of the same size, COMSOL Script interprets them as vectors of different orientation and uses `x` and `y` that are the same as what you get from a call to `meshgrid` with `x` and `y` as inputs. `z` must either be the same size as x and y or, when they are vectors of different orientation, a matrix with the same number of rows as the length of x and the same number of columns as the length of y. The formats as for x and y apply to xi and yi. It is also possible for the data gridding functions to return a structure for interpolation purposes:.

```
s = griddata(x,y,xi,yi);
```

returns a structure s that contains the triangulation of x and y and information about which Delaunay element the points in xi and yi belong to, including local coordinates. You can use this together with the tinterp function to interpolate different data values using the same points and triangulation (see "Search and Interpolation Functions" on page 168). In addition, you can specify the interpolation method as an extra input argument to the data gridding functions. The interpolation method can be linear interpolation (linear, which is the default method) or nearest neighbor interpolation (nearest). In this case, the nearest neighbor signifies the closest vertex in the nearest Delaunay triangle.

The other data gridding functions, griddata3 and griddatan, work in a similar way. For more information about the data gridding functions, including some additional input arguments, see the *COMSOL Script Command Reference* or the command-line help. An example using griddata appears in "Data Gridding and Interpolation Example" on page 169.

SEARCH AND INTERPOLATION FUNCTIONS

To find Delaunay elements for a set of points, use the tsearch and tsearchn functions. For example,

```
ind = tsearch(x,y,tri,xi,yi);
```

provides the indices to the Delaunay elements for all points (x_i, y_i) defined by the vectors xi and yi. The vector ind contains indices into tri (or NaNs for points outside the mesh), which is the triangulation of x and y, typically from a call to the delaunay function. To get the barycentric coordinates for xi and yi, use the tsearchn function:

```
[ind,coord] = tsearchn([x(:),y(:)],tri,[xi(:),yi(:)]);
```

The tsearchn function finds Delaunay elements in any space dimension and uses point data matrices instead of vectors with x and y coordinates (containing $N \times 2$ elements in 2D and $N \times 3$ elements in 3D)

For interpolation on a Delaunay triangulation, use the tinterp function:

```
yi = tinterp(s,y);
```

This function uses the Delaunay triangulation in the structure `s`, which can be the output from a call to `griddata`, for example, and interpolates linearly to compute the interpolated values based on `y`, which must have a size that matches the original data points. For details on the structure `s`, see the command-line help for `tinterp` and the *COMSOL Script Command Reference*. The following example uses the `griddata` and `tinterp` functions.

Data Gridding and Interpolation Example

The following script first creates 100 random points for the function

$$z = \sin(x)\cos(y)e^{(-2x^2 - y^2)}$$

and then interpolates onto a 41x41 rectangular grid between –2 and 2 in the x direction and the y direction. The plot shows the original data and a wireframe surface of the interpolated data.

```
rand('state',0);
x = 4*rand(1,100)-2;y = 4*rand(1,100)-2;
ti = -2:.1:2;
[xi,yi] = meshgrid(ti,ti);
g = griddata(x,y,xi,yi,'linear',[],'closest');
z=sin(x).*cos(y).*exp(-2*x.^2-y.^2);
zi1 = tinterp(g,z);
plot3(x,y,z,'*');
hold on;
mesh(xi,yi,zi1);
```

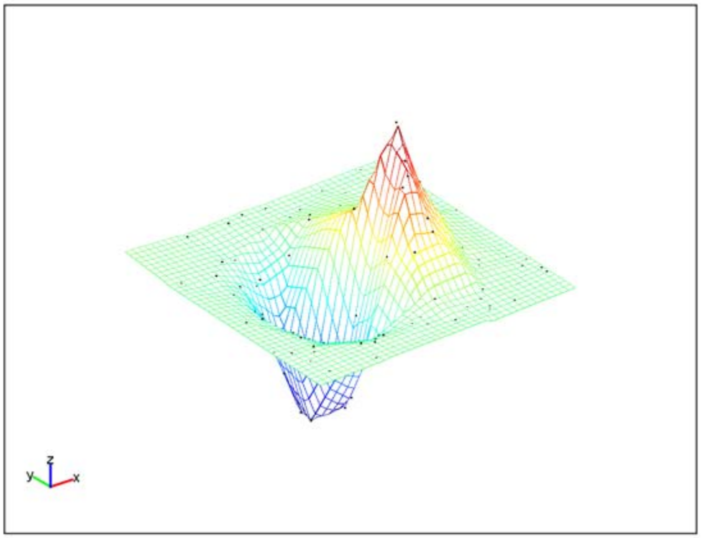


Figure 8-5: Wireframe plot of interpolated data based on the function value at 100 random points.

Summary of Interpolation and Polynomial Functions

The following table summarizes functions for interpolation, triangulation, and polynomials:

TABLE 8-5: INTERPOLATION, TRIANGULATION, AND POLYNOMIAL FUNCTIONS

FUNCTION NAME	DESCRIPTION
`conv`	Convolution of vectors (polynomial multiplication)
`conv2`	2D convolution of matrices
`convn`	Multidimensional convolution of matrices
`deconv`	Deconvolution of vectors (polynomial division)
`delaunay`	Delaunay triangulation
`delaunay3`	3D Delaunay triangulation
`griddata`	2D data gridding
`griddata3`	3D data gridding
`griddatan`	n-D data gridding

TABLE 8-5: INTERPOLATION, TRIANGULATION, AND POLYNOMIAL FUNCTIONS

FUNCTION NAME	DESCRIPTION
interp1	1D interpolation
interp2	2D interpolation
interp3	3D interpolation
mkpp	Make piecewise polynomial
pchip	Piecewise cubic Hermite interpolation
poly	Polynomial with specific roots
polyder	Differentiate a polynomial
polyfit	Polynomial fit
polyint	Integrate a polynomial
polyval	Evaluate a polynomial
ppval	Evaluate piecewise polynomial
roots	Find polynomial roots
spline	Cubic spline interpolation
tinterp	Interpolation on Delaunay triangulation
trimesh	Create a mesh plot with triangles
trisurf	Create a surface plot with triangles
tsearch	Find Delaunay element
tsearchn	Find Delaunay element in nD
unmkpp	Extract details from piecewise polynomial

Differentiation and Integration

COMSOL Script includes several functions for numeric differentiation and integration.

Difference, Gradients, and Laplacian Computations

COMPUTING THE DIFFERENCE AND APPROXIMATE DERIVATIVE

To approximate the derivative of a function, one approach is to compute the differences between adjacent elements in an array with the `diff` function:

```
x = [1 2 4 7 11 16 22 29]

x =

  1    2    4    7    11    16    22    29

diff(x)

ans =

  1    2    3    4    5    6    7
```

The following code now illustrates how `diff` provides a numerical approximation of the function's derivative:

```
h = 0.01;
x = 0:h:10;
f = x.^2;
der = diff(f);
plot(x(1:end-1),f(1:end-1))
hold on
plot(x(1:end-1),der/h,'r--')
```

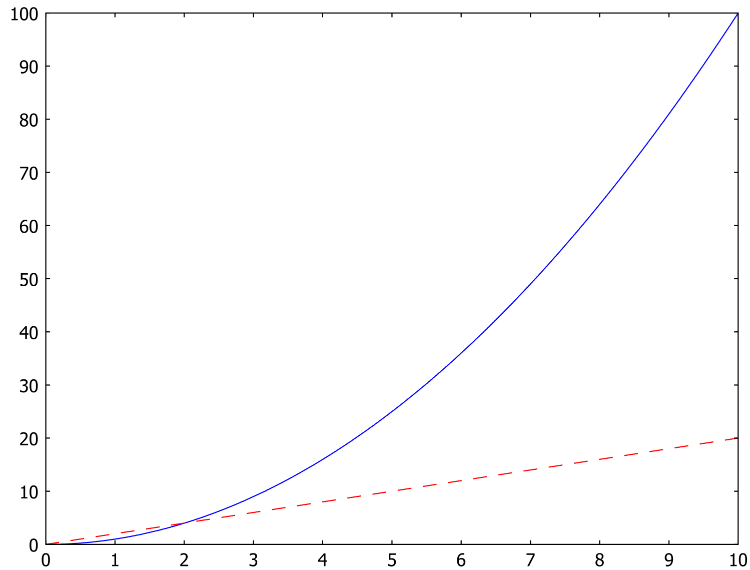


Figure 8-6: A plot of x^2 (solid) and its approximate derivative (dashed).

The straight line representing `der/h` is the approximation of the derivative to x^2, which is $2x$.

COMPUTING A VECTOR GRADIENT

The function `gradient` computes an approximate gradient of a vector or array. The function assumes a spacing of 1 unless you provide a second input argument for the spacing. For 2D data, `[fx,fy] = gradient(F)` computes the gradient of a matrix `f`, where `fx` corresponds to $\partial F/\partial x$, the differences in the column direction, and `fy` corresponds to $\partial F/\partial y$, the differences in the row direction. In 3D, an additional output variable contains the gradient in the z direction.

COMPUTING THE LAPLACIAN

The function `del2` computes the discrete Laplacian of a matrix or multidimensional array. The discrete 5-point Laplacian for a matrix is an approximation of the Laplace differential operator, Δ, which takes the average of the surrounding four elements and subtracts the original element. You can provide nonunit spacing using additional input arguments.

Numerical Integration

There are functions available for two methods for numerical integration: quadrature and trapezoidal numerical integration. Both methods determine numerical values of definite integrals.

QUADRATURE

The functions `quad` and `quadl` both provide numerical integration using quadrature formulas. `quad` uses Simpson's rule, which uses quadratic polynomials to provide exact results for integrals of polynomials up to degree 3. `quadl` uses Lobatto quadrature with a Kronrod extension (see Ref. 1). Both algorithms use adaptive quadrature, which means that they break the integration interval into subintervals and then apply the integration rule over each of the subintervals.

The inputs to `quad` and `quadl` are the name of the function (integrand) and the start value and end value of the integration interval. To integrate the function $y = x^3$ on the interval from 0 to 10, define a function, for example `myfcn.m`, with the following two lines:

```
function y=myfcn(x)
y = x.^3;
```

Notice the pointwise multiplication, which makes the function work for vector inputs.

```
int=quad('myfcn',0,10)

int =

2500
```

In this case, the integral is exact. To integrate the function $y = (1+x)/(1+(5-x)^2)$ on the same interval using `quadl`, change the second line in `myfcn.m` to

```
y = (1+x)./(1+(5-x).^2);
```

Then call `quadl` (or `quad`) with a second output argument, which returns the number of function evaluations:

```
[int,count]=quadl('myfcn',0,10);
```

This function call provides an integral of 16.480687 from 138 function evaluations using the default relative tolerance of 10^{-6}. To use a tolerance of 10^{-8} instead, type

```
[int,count]=quadl('myfcn',0,10,1e-8);
```

This case, the resulting integral is 16.480809 from 318 function calls.

You can also supply a relative tolerance and additional inputs for outputting a trace and providing parameters as extra arguments to the function that describes the integrand (see the online help or the *COMSOL Script Command Reference* for details).

TRAPEZOIDAL NUMERICAL INTEGRATION

The `trapz` function implements trapezoidal numerical integration. `trapz(x,y)` computes the integral of `y` as a function of `x` where `x` and `y` are vectors of the same size. It is also possible to use `trapz` to compute the integrals of each column in a matrix and for multidimensional arrays.

The following example computes the integral of the function $y = x^3$ on the interval from 0 to 10. The exact solution is 2500.

```
h = 0.1;
x=0:h:10;
y=x.^3;
int = trapz(x,y)

int =

2.500e+003
```

For cumulative numerical integration, use the function `cumtrapz`.

Summary of Differentiation and Integration Functions

The following table summarizes the differentiation and integration functions:

TABLE 8-6: DIFFERENTIATION AND INTEGRATION FUNCTIONS

FUNCTION NAME	DESCRIPTION
`cumtrapz`	Cumulative trapezoidal numerical integration
`del2`	Discrete Laplacian
`diff`	Compute the difference between adjacent array elements
`gradient`	Compute approximate gradient
`quad`	Numerical integration using adaptive Simpson quadrature
`quadl`	Numerical integration using adaptive Lobatto quadrature
`trapz`	Trapezoidal numerical integration

Reference

1. W. Gander and W. Gautschi, “Adaptive Quadrature—Revisited,” *BIT*, vol. 40, no. 1, pp. 84–101, 2000.

Data Input/Output

The basic functions for storing data to a file and loading data from a file are `save` and `load`. Additional basic file I/O functions are available for formatting and saving data to files. There are also functions for reading and writing sound data files and for playing sound.

Saving and Loading Data To and From a File

To save the entire workspace to a file with the name `data01`, type:

```
save datao1
```

This command saves all workspace variables as binary data in the file `data01.flws`. This is a binary file format that is the default file type when you save data using the `save` command.

To save to a filename that is a string variable, use the functional form of `save`:

```
save(filename)
```

To save only the variables `x`, `y`, and `u`, type:

```
save data01 x y u
```

To save data in ASCII text format, add the `-ascii` switch:

```
save data01 -ascii
```

Another switch, `-tabs`, saves ASCII data in a tab-separated format.

Note: You can save only numerical 2D matrices in ASCII format.

To load workspace data from a file, use the `load` function:

```
load data01
```

To read the loaded data into a structure variable, type

```
s = load(filename);
```

It is also possible to read numerical matrix data from a text file (for example, `matrix.txt`) using

```
load matrix.txt -ascii
```

There is a direct correspondence between the rows of data in the text file and the rows of data in the resulting COMSOL Script variable. The variable name becomes the same as the file name.

Note: Loading variables from a file overwrites any existing variables with the same name.

To read data from a text file where the data is separated with a delimiter (typically a whitespace character such as a space, a tab, or a line feed), use the `dlmread` function:

```
out = dlmread(filename);
```

reads the file `filename` and returns a matrix where each row contains a row of the file. The elements of the data must be real or complex numbers. The default delimiter is whitespace, but you can specify the delimiter as an additional input argument. Using additional range arguments makes `dlmread` read a subset of the data. This is convenient when you have a data file with header information, for example, and you want to read the data after the header. Contrary to array indexing in COMSOL Script, the numbering of rows and columns starts at 0, not 1. Keep this in mind when you specify the rows and columns to read a subset of the data. It is also possible to read a subset of the data file using Microsoft Excel's A1 notation for the rows and columns.

You can also write data separated by a delimiter to a text file using the `dlmwrite` function:

```
dlmwrite('mydata',data)
```

writes the `data` matrix as a comma-separated text file `mydata`. You can also specify the delimiter as a third input argument and use a number of empty starting rows and columns, which you specify as the fourth an fifth input argument, respectively. In addition, there are additional properties that you can use, for example, to control the precision. The property values is an integer number or significant digits or a format string of the form used by `fprintf` and `sprintf`. An example:

```
data = reshape(sin(1:9), 3, 3);
dlmwrite('TABLE', data, 'delimiter', ':', 'precision', 2)
```

creates a file called `TABLE` with the following contents:

```
0.84:-0.76:0.66
0.91:-0.96:0.99
0.14:-0.28:0.41
```

Reading and Writing Formatted Data To a File

Using other COMSOL Script I/O functions you can write formatted data to a file. The corresponding functions include `fopen` for opening files for reading and writing, `fprintf` for formatted printing to a file, `fscanf` for reading formatted data, and `fclose` for closing files. The table in the next section contains a complete list of I/O functions. Also see the Help item for each function accessible from the command line and found in the digital document the *COMSOL Script Command Reference*.

EXAMPLE OF WRITING FORMATTED DATA TO FILE

The following function takes two input vectors of the same size and writes the data to a text-file format that the interpolation function in COMSOL Multiphysics supports. The file has this syntax:

```
% Grid
x grid points separated by spaces
% Data
Data values separated by spaces
```

There are three basic steps to saving formatted data to a file:

1. Open a file in write mode with `fopen`. In the following example, the mode flag is `'wt'`, which opens a file for writing in text mode (on Windows only).
2. Use `fprintf` or `fwrite` to write data to the file.
3. Close the file with `fclose`.

The format in `fprintf` is `'%15.5e'`, a floating-point number with five significant digits.

```
function writefile(filename,X,Y)
% Creates an interpolation data file.
form = '%15.5e';
fid = fopen(filename,'wt');
fprintf(fid,'%% Grid\n');
fprintf(fid,strtrim(sprintf(form,X)));
fprintf(fid,'\n');
fprintf(fid,'%% Data\n');
for iZ = 1:size(Y,3)
    str = cellstr(num2str(Y(:,:,iZ)',form));
    fprintf(fid,'%s\n',str{:});
end
fclose(fid);
```

EXAMPLE OF READING TEXT FROM A FILE

A routine can read the Help text from a COMSOL Script file (here, `kron.m`). To do so, type the following:

```
kronlocation = which('kron.m');
fid = fopen(kronlocation,'r');
if fid>0
  kronstr = fread(fid);
  char(kronstr')
  fclose(fid)
end
```

The default format for `fread` is to read all characters in the file. The output is a double array with the corresponding ASCII values of the string. The check of the file ID from `fopen` make sure that COMSOL Script was able to open the file successfully; a file ID of –1 indicates that `fopen` failed to open the file.

Saving and Loading MAT-Files

To save or load data to MAT-files, use the `-mat` switch. For example,

```
load -mat mydata
```

loads the variables in `mydata.mat` into the COMSOL Script workspace. If you do not specify an extension, the default setting is to save to a `.ws`-file and to load from either a `.ws`- or a MAT-file (extension `.mat`). COMSOL Script can read MAT-files created using MATLAB version 5.0 or later. The following types of data in MAT-files are not fully supported:

- Integers and unsigned integers (data types `int*`, `uint16`, `uint32`, and `uint64`), which become double arrays in COMSOL Script.
- MATLAB objects, which the `load` function loads into structures, except for inline functions and COMSOL Multiphysics objects created by FEMLAB/COMSOL Multiphysics versions 3.0 and later. Loading these objects converts them into the corresponding COMSOL Script data types.

Interfacing With Microsoft Excel Spreadsheets

You can both read data from Microsoft Excel spreadsheet files (`.xls` files) and update Excel files with data from COMSOL Script.

READING DATA FROM EXCEL SPREADSHEETS

To read data from, for example, `data.xls`, type:

```
num = xlsread('data.xls');
```

With one output argument, `xlsread` only reads numerical data. You can add additional output arguments to include text and mixed data ("raw data"):

```
[num,txst,raw] = xlsread('data.xls');
```

You can also supply an integer or string that specifies the sheet to read from and a range using Excel's A1 notation, for example, `A1:K16`:

```
[num,string,raw] = xlsread('data.xls',1,A1:K16);
```

By default, `xlsread` trims leading and trailing rows and columns that contains NaNs (nonnumeric data) for the numeric and raw-data parts, and it also trims leading and trailing empty strings for the text part. It is possible to turn off this trimming by setting the `trim` property to `off`.

The `xlsread` function supports the Excel 97 format and later versions.

WRITING DATA TO EXCEL SPREADSHEETS

To write data from COMSOL Script to an Excel spreadsheet, use `xlswrite`:

```
xlswrite('mydata.xls',data)
```

The data can be a matrix of real-valued numerical data or a cell array. For a cell array, `xlswrite` only writes numerical data and strings. If you want to combine text data with numerical data, you must use a cell array. To convert a double matrix into a cell array, use the `num2cell` function.

Use one or two additional input arguments to specify a sheet and range in the spreadsheet where you want to store the data, for example:

```
xlswrite('mydata.xls',data,'Sheet 1',A2:G10)
```

Reading and Writing Sound Files and Playing Sounds

COMSOL Script includes functions for reading and writing sound data as wave sound files (with extension `.wav`) as well as functions for playing sounds.

READING AND WRITING WAVE SOUND FILES

You can read and write wave sound files using the functions `wavread` and `wavwrite`, respectively. The sound data is pulse-code modulated signal data, where the number of bits per sample must be 8 or 16. For a mono sound, the sound data is a column vector, and for stereo sound, the sound data is an $N \times 2$ matrix with two columns for the to stereo channels.

The basic syntax for wavread is

```
data = wavread(filename);
```

where `filename` is the name of the `.wav` file. You can also get the sample rate, the number of bits, and a description (if available) as the second, third, and fourth output argument, respectively.

To output the number of frames and channels but ignore the signal, type

```
[nframes, nchannels] = wavread(filename,'size');
```

In addition, you can use a scalar or vector of length 2 as the second input argument to `wavread` in order to only read the first few samples or a range of samples.

To write wave sound data to a `.wav` file, use `wavwrite`:

```
wavwrite(data,filename)
```

where `data` is the wave sound data, and `filename` is the name of the `.wav` file. Use optional second and third input arguments to specify the sample rate and the number of bits per sample (must be 8 or 16; the default is 16 bits per sample).

PLAYING SOUNDS

To play a sound, use the functions `sound` or `soundsc`:

```
sound(data)
```

interprets the contents of `data` in data as a pulse-code modulated signal and plays it with a sample rate of 8192 Hz using 16 bits per sample. The `sound` function clips signal values outside the range of [–1, 1]. You can use optional second and third input arguments to specify the sample rate and the number of bits per sample (8 or 16), respectively.

The `soundsc` function also plays sound and uses the same syntax but scales and translates the signal such that the minimum and maximum amplitudes sent to the output device are –1 and 1, respectively.

Note: The `sound` and `soundsc` functions are only available if the platform has support for sound.

Summary of Input/Output Functions

The following table summarizes the I/O functions:d

TABLE 8-7: I/O FUNCTIONS

FUNCTION NAME	DESCRIPTION
delete	Delete files or graphics objects
dlmread	Read a delimited file
dlmwrite	Write a delimited file
fclose	Close an open file
feof	Test if end-of-file has been reached
ferror	Return or clear the error message
fgetl	Read a line from a file (discarding line feed character)
fileparts	Split a filename into path, name, and extension
filesep	Get the system file separator
fopen	Open a file or get information about opening a file
fprintf	Write formatted output to a file
fread	Read binary data from a file
frewind	Rewind a file
fscanf	Read formatted data from a file
fseek	Move a file pointer
ftell	Get the position of the file pointer
fullfile	Create a file name
fwrite	Write data to a binary file
load	Load the workspace data from a file
pathsep	Get the system path separator
save	Save the workspace data to a file
sound	Play a sound
soundsc	Play a scaled sound
strread	Read formatted text
tempdir	Get the directory where temporary files can be created
tempname	Get a temporary file name
textread	Read a formatted text
wavread	Read a .wav sound file
wavwrite	Write a .wav sound file

TABLE 8-7: I/O FUNCTIONS

FUNCTION NAME	DESCRIPTION
xlsread	Read a .xls Excel spreadsheet file
xlswrite	Write to a .xls Excel spreadsheet file

Date and Time Functions

COMSOL Script includes functions for getting both the time and date and for measuring the time it takes to run a function, script, or series of statements.

Getting the Current Date and Time

To get the current date, use the `date` function:

```
today = date

today =

  22-Jul-2005
```

`date` returns the current date in a string with the format DD-MMM-YYYY. You can use the string function `strtok` to extract the day, month, and year into separate string variables:

```
[day,remainder] = strtok(today,'-');
[month,remainder] = strtok(remainder,'-');
year = strtok(remainder,'-');
```

To get the current time, use the `clock` function:

```
t = clock

t =

      2005       7      22       9      26      37.250000
```

`clock` returns the current time in a 1×6 vector (double array). The elements represent, from left to right:

- The current year
- The current month
- The current day
- The current hour
- The current minute
- The current second

All values except the value for seconds are integers.

Measuring Elapsed Time

Three functions are available to measure elapsed time while running COMSOL Script functions and programs:

- Use `etime` to get the elapsed time, in seconds, as the difference between two times. A typical application is to first store the current time in a variable

  ```
  t1 = clock;
  ```

  and then run the code that you want to clock and get the elapsed time as

  ```
  t2 = etime(clock,t1);
  ```

  With `etime` you can measure several elapsed times and store them in variables for further analysis.

- Use `tic` and `toc` for quick measurements of the elapsed time:

  ```
  tic, a = norm(rand(1000)); toc
  ```

  ```
  Elapsed time:  3.281 s
  ```

  `tic` starts a timer, and `toc` stops it. Use `t = toc` to store the elapsed time.

Summary of Functions for Date and Time

The following table summarizes the functions for date and time:

TABLE 8-8: DATE AND TIME FUNCTIONS

FUNCTION NAME	DESCRIPTION
`clock`	Current time
`date`	Current date
`etime`	Elapsed time
`tic`	Start the timer
`toc`	Stop the timer

9

Plotting and Visualizing Data

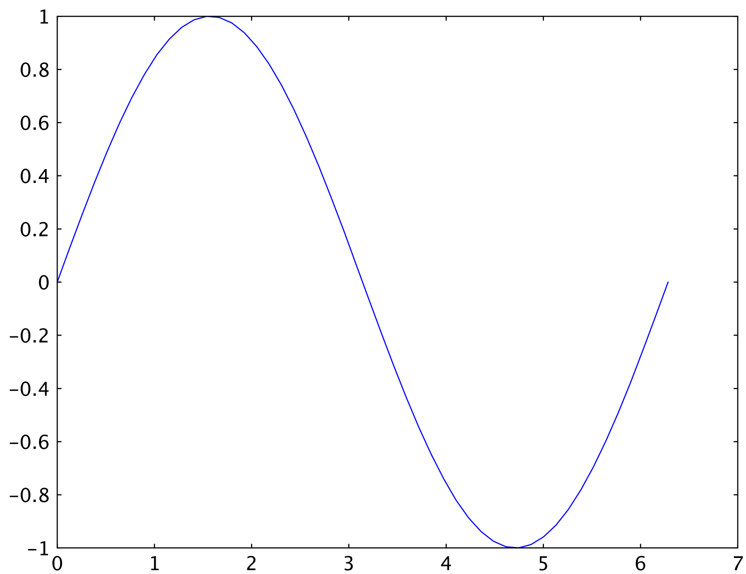
To plot data from an analysis or from a model, COMSOL Script provides a variety of plots and graphs. This chapter introduces you to the various graphics objects such as frames and axes objects as well as the different types of plots in 2D and 3D and the functions to create them.

Introduction to Graphics Objects

To most easily understand the terminology in this chapter, start with a short example.

Enter the following three lines of code into COMSOL Script and then examine the result:

```
x=linspace(0,2*pi,100);
y=sin(x);
plot(x,y);
```



The `plot` command generates a *figure window*, at the top of which is a *toolbar*. The figure window also contains one *axes object* in which the software has plotted the desired function, $\sin(x)$. As you will see, a figure window can contain multiple axes objects, in 2D or 3D, each plotting a different function.

Now let us look at each of these elements in detail and how to construct them.

The Figure Window

Figure Window Functions

A figure window is the highest-level plotting object. It contains one or more axes objects into which you plot data sets (see “Axes” on page 193 for more information).

Start by examining some basic functions associated with a figure window:

TABLE 9-1: FUNCTIONS RELATED TO FIGURE WINDOWS

FUNCTION NAME	DESCRIPTION
`clf`	Clear the contents in the current figure window
`drawnow`	Flushes graphics rendering and repaints the screen. Usually the screen is repainted only when a script has finished, but `drawnow` forces repaints while a script is running
`figure`	Creates a new figure window
`gcf`	Returns a handle to the current figure window

Plotting functions create a new figure window automatically as needed. You can also explicitly create a new figure window by typing

```
figure
```

To retrieve the handle to the current figure window, call `gcf`; to get the handle to the current axes object, call `gca`. To make the figure with handle `h` the current figure window, enter

```
figure(h)
```

Figure Window Toolbar

At the top of any figure window is a toolbar with buttons for quick access to commonly performed tasks.

The buttons on the toolbar are, from left to right (some are available only for 3D plots):

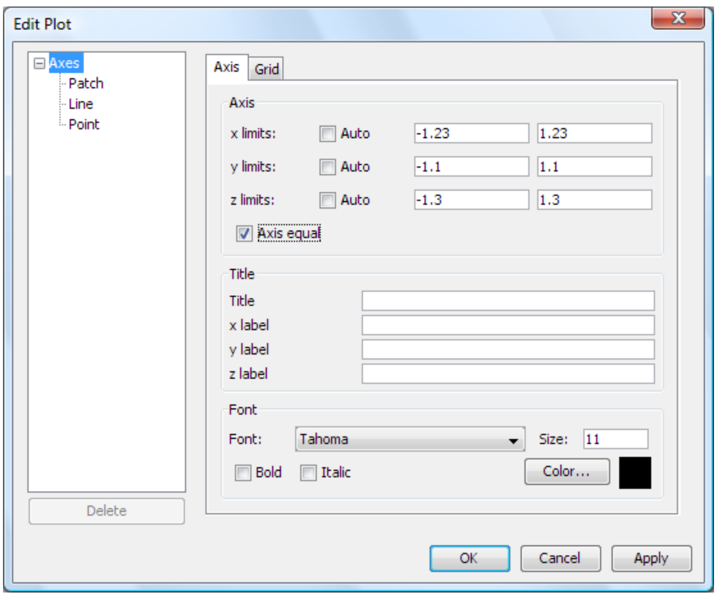
- **Export Image**—Export the plot in the figure window as an image.
- **Print**—Send the plot in the figure window to a printer.

- **Copy**—Copy the plot to the system clipboard.
- **Export Current Plot**—Export data values from the current plot to a text file.
- **Edit Plot**—Open a dialog box where you can change properties for axes, lines, patches, and so on that can normally be changed using `set` and `get` in conjunction with graphics handles.
- **Orbit/Pan/Zoom**—Selects the general orbit/pan/zoom mode for the mouse in 3D plots.
- **Pan**—Pan when clicking and moving the mouse.
- **Zoom**—Zoom when clicking and moving the mouse.
- **Dolly In/Out**—Dollies the camera in and out when clicking and moving the mouse.
- **Move as Box**—When this button is pressed, a box appears instead of updating the graphics in real time when changing camera settings with the mouse. It is useful for very large plots or for computers with poor graphics cards.
- **Scene Light**—Turns on all lights added to a plot.
- **Headlight**—Turn on a light mounted on the camera and looking in the direction of the camera.
- **Back in Camera History**—Return to the previous camera position after changing it interactively using the mouse.
- **Forward in Camera History**—Move forward in camera history.
- **Orthographic Projection**—Use orthographic projection.
- **Perspective Projection**—Use perspective projection.
- **Go to XY View**—View the plot in the *xy*-plane.
- **Go to YZ View**—View the plot in the *yz*-plane.
- **Go to ZX View**—View the plot in the *zx*-plane.
- **Go to Default 3D View**—View the plot in the default 3D view.
- **Increase Transparency**—Increase the transparency of patch and surface objects.
- **Decrease Transparency**—Decrease the transparency of patch and surface objects.
- **Zoom In**—Zoom into the plot by a factor of two.
- **Zoom Out**—Zoom out of the plot by a factor of two.
- **Zoom Window**—Zooms in on a rectangular area by clicking and dragging the mouse cursor.
- **Zoom Extents**—Zoom to the extents of the graphics objects currently in the figure window so they appear at the largest possible scale without truncation.

The Edit Plot Dialog Box

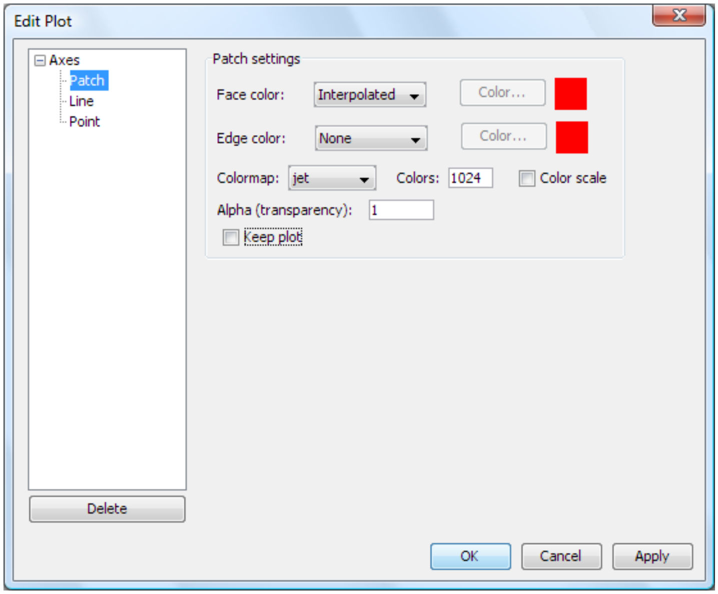
It is also possible to manually edit many of the properties for any graphic objects in a figure window, and you do so in the **Edit Plot** dialog box. To open it, click the **Edit Plot** button on the figure-window toolbar. To the left it shows a tree with the currently defined axes objects in the figure window along with the graphical objects that lie in each axes object. When you select a node in the tree, a panel appears to the right in which you can change properties for the selected object, specifically, most of the properties it is possible to change using `set` on the command line when given the corresponding graphics handle.

Here you can change axes limits, the title, axes labels, and so on by selecting the Axes node in the tree.



Axis ticks marks, the title, and axes labels appear using the selected font.

In the next example, change the `Facecolor`, `Edgecolor`, and `Colormap` properties, and so on, for a patch by selecting the **Patch** node in the tree of graphics objects.



Axes

Overview of Axes Functions

An axes object creates the area in which you plot data, and one figure window can support multiple axes objects. You control axes limits and ticks marks manually or let COMSOL Script calculate them automatically. Other functions add text, labels, and grid lines. The following functions are related to creating and modifying axes objects:

TABLE 9-2: FUNCTIONS RELATED TO THE AXES

FUNCTION NAME	DESCRIPTION
`axis`	Change or get axes limits
`box`	Display an axis box (3D only)
`cla`	Clear all graphics objects from an axes object
`gca`	Return the handle to the current axes object
`grid`	Display a grid in the axes object
`hold`	Specify that existing plots in the axes remain when you add new plots
`ishold`	Return the Hold state
`legend`	Display a legend with the plot
`newplot`	Return a handle to an axes object. If a current axes object already exists, all graphics objects are cleared before the handle is returned
`subplot`	Divide a figure window into a grid of several axes objects
`title`	Display a title above the axes object
`text`	Display a text at specified coordinates in the axes object
`xlabel`	Display a label on the x-axis
`xlim`	Set and get x-axis limits
`ylabel`	Display a label on the y-axis
`ylim`	Set and get y-axis limits
`zlabel`	Display a label on the z-axis
`zlim`	Set and get z-axis limits

Getting an Axes Object for Plotting

Normally you do not need to explicitly create an axes object for plotting; COMSOL Script automatically creates a new figure window with an axes object if one does not already exist. Plotting commands always pertain to the current axes object, which is the one in the figure window that was the latest to have focus. Use the `'parent'` property with an axes handle to explicitly specify which axes object plots the data.

The function

```
h = gca
```

returns a handle to the current axes object. A rough equivalent is the function

```
h = newplot
```

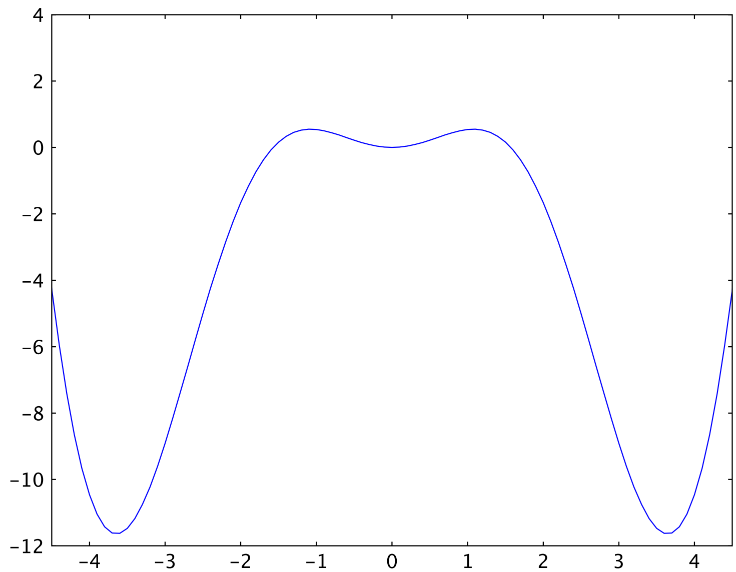
which returns a handle to the current axes object but also clears all graphics at the same time. It is normally needed only when working with low-level graphics functions such as `line` and `patch`; higher-level graphics functions such as `plot` and `surf` automatically clear the axes object they plot into before adding new graphics.

Controlling Axes Limits

COMSOL Script automatically updates axes limits to fit any plots added to an axes object. The `axis` function overrides this feature and allows you to set the limits manually. It also sets the properties `'Xlimmode'` and `'Ylimmode'` for the axes object to `'manual'`, thereby preventing anyone or anything from updating the limits when a program adds new graphics to the axes object.

To manually set the axis limits, pass an array with pairs of minimum and maximum values to the `axis` function as in this code snippet:

```
x=-5:0.1:5;
y=x.^2.*cos(x);
plot(x,y);
axis([-4.5 4.5 -12 4]);
```



The same effect is possible with the functions `xlim` and `ylim` or by setting the `Xlim` and `Ylim` properties on the handle to an axes object.

The following three ways of controlling the axes limits are equivalent:

- Using the `axis` function:

```
axis([-4.5 4.5 -12 4]);
```

- Using the `xlim` and `ylim` functions:

```
xlim([-4.5 4.5]);
ylim([-12 4]);
```

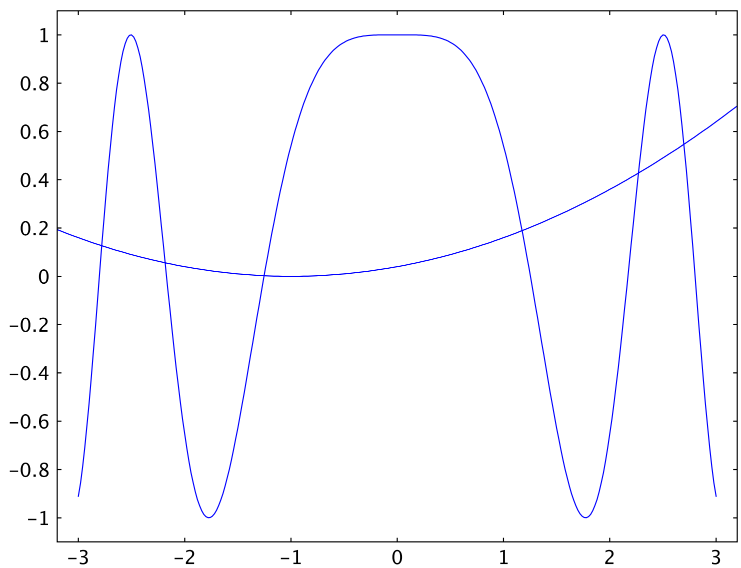
- Using the `set` function and the `Xlim` and `Ylim` properties of the axes object:

```
set(gca,'Xlim',[-4.5 4.5]);
set(gca,'Ylim',[-12 4]);
```

Adding Plots to an Existing Plot

High-level visualization commands such as `plot` and `surf` normally clear the contents of the axes object into which they plot before they add any new graphics objects. You can override this behavior with the `hold` function. It specifies that an axes object should retain plots even when it receives new plots. If the axes limits are in manual mode, they remain unchanged when plots are held. The following example illustrates how to work with the `hold` command:

```
x=-3:0.02:3;
y=cos(x.^2);
plot(x,y);
set(gca,'xlimmode','manual');
set(gca,'ylimmode','manual');
axis([-3.2 3.2 -1.1 1.1]);
hold on;
x1=-5:0.04:5;
y1=((x1+1)/5).^2;
plot(x1,y1);
```



Using Multiple Axes Objects

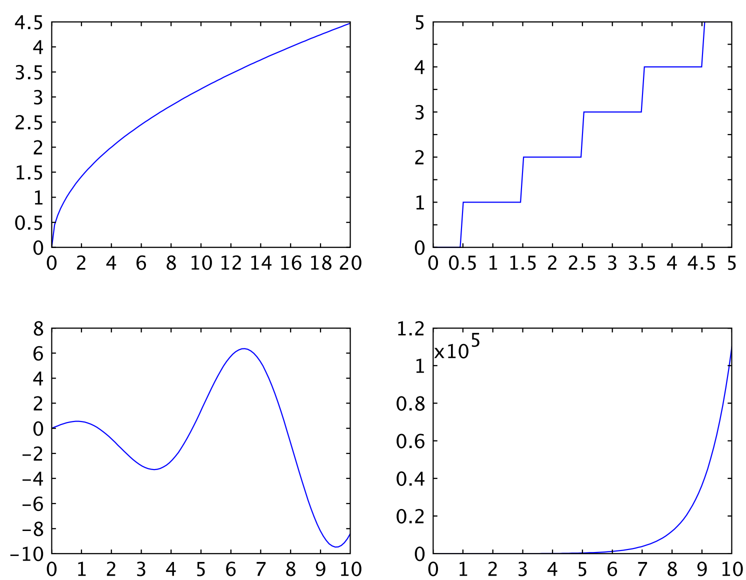
The `subplot` function divides a figure window into a grid that, in turn, contains several axes objects. Specifically, the command

```
subplot(rows,cols,current)
```

divides the figure window into a grid of `rows`×`cols` axes objects. The axes object with the number `current` becomes the current axes. Axes numbering increases along the columns of the first row, then along the second row, and so on.

Look at this code, which divides a figure window into four axes objects and plots a different function in each one:

```
subplot(2,2,1);
x1=linspace(0,20,100);
y1=sqrt(x1);
plot(x1,y1);
subplot(2,2,2);
x2=linspace(0,5,100);
y2=round(x2);
plot(x2,y2);
subplot(2,2,3);
x3=linspace(0,10,100);
y3=x3.*cos(x3);
plot(x3,y3);
subplot(2,2,4);
x4=linspace(0,10,100);
y4=x4.*sinh(x4);
plot(x4,y4);
```



Adding Annotations

The following table lists the commands available for adding text to a plot.

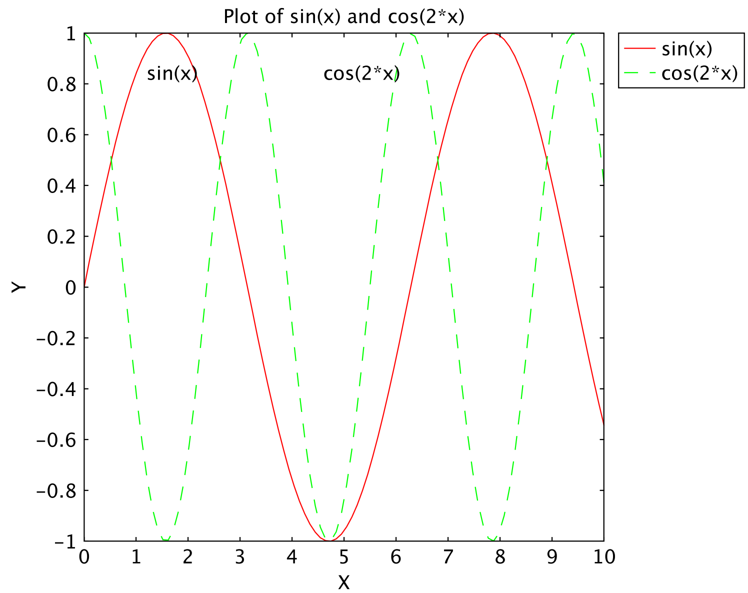
TABLE 9-3: FUNCTIONS FOR ADDING TEXT TO A PLOT

FUNCTION	DESCRIPTION
`legend`	Displays a legend to the right of the plot
`text`	Displays a text at arbitrary coordinates in the axes object
`title`	Displays a title above the axes object
`xlabel`	Displays a label on the x-axis
`ylabel`	Displays a label on the y-axis
`zlabel`	Displays a label on the z-axis

To see these commands at work, examine the following example. It creates a plot of `sin(x)` and `cos(2*x)` and displays a title, axis labels, legends, and some descriptive text:

```
x=linspace(0,10,100);
y1=sin(x);
y2=cos(2*x);
plot(x,y1,'r-',x,y2,'g--');
legend('sin(x)','cos(2*x)');
title('Plot of sin(x) and cos(2*x)');
xlabel('X');
ylabel('Y');
text(1.2,0.8,'sin(x)');
```

```
text(4.6,0.8,'cos(2*x)');
```



FORMATTING AND SYMBOLS

The `text` function can take formatted strings that include HTML tags, Greek letters, mathematical symbols, and Unicode characters. These formatting options include the strings in plot titles as well as *x*-axis, *y*-axis, and *z*-axis labels (the `title`, `xlabel`, `ylabel`, and `zlabel` functions, respectively).

The `text` function supports the following HTML tags in the text string:

TABLE 9-4: VALID HTML TAGS

HTML TAG	DESCRIPTION
`<B> </B>`	Enclosed text is rendered using a bold font
` `	Line break
`<CENTER> </CENTER>`	Centered text
`<I> </I>`	Enclosed text is rendered using an italic font
`<LI>`	List item. When the list used is `<OL>` (an ordered list) the `LI` element is rendered with a number. When the list used is `<UL>` (an unordered list) the `LI` element is rendered with a bullet
`<OL> </OL>`	Ordered list (see also: `<LI>`)

TABLE 9-4: VALID HTML TAGS

HTML TAG	DESCRIPTION
`<P> </P>`	Paragraph. This tag creates a line break and a space between lines
`<PRE> </PRE>`	Enclosed text preserves spaces and line breaks. The text is rendered using a monospaced font
`<STRIKE> </STRIKE>`	Enclosed text is rendered with a strike-through appearance
`<SUB> </SUB>`	Enclosed text is rendered in subscript with the enclosed text slightly lower than the surrounding text
`<SUP> </SUP>`	Enclosed text is rendered in superscript with the enclosed text slightly higher than the surrounding text
`<TT> </TT>`	Enclosed text is rendered using a monospaced font
`<U> </U>`	Enclosed text will be underlined
`<UL> </UL>`	Unordered list (see also: `<LI>`)

The `text` function supports the following Greek character tags in the text strings:

TABLE 9-5: VALID GREEK SYMBOL TAGS

TAG	SYMBOL	TAG	SYMBOL
\ALPHA	Α	\alpha	α
\BETA	Β	\beta	β
\GAMMA	Γ	\gamma	γ
\DELTA	Δ	\delta	δ
\EPSILON	Ε	\epsilon	ε
\ZETA	Ζ	\zeta	ζ
\ETA	Η	\eta	η
\THETA	Θ	\theta	θ
\IOTA	Ι	\iota	ι
\KAPPA	Κ	\kappa	κ
\LAMBDA	Λ	\lambda	λ
\MU	Μ	\mu	μ
\NU	Ν	\nu	ν
\XI	Ξ	\xi	ξ
\OMICRON	Ο	\omicron	ο
\PI	Π	\pi	π
\RHO	Ρ	\rho	ρ

TABLE 9-5: VALID GREEK SYMBOL TAGS

TAG	SYMBOL	TAG	SYMBOL
\SIGMA	Σ	\sigma	σ
\TAU	Τ	\tau	τ
\UPSILON	Υ	\upsilon	υ
\PHI	Φ	\phi	φ
\CHI	Χ	\chi	χ
\PSI	Ψ	\psi	ψ
\OMEGA	Ω	\omega	ω

The `text` function supports the following math symbol tags in the text string:

TABLE 9-6: VALID MATH SYMBOL TAGS

TAG	SYMBOL	TAG	SYMBOL
\approx	≈	\bullet	•
\lequal	≤	\partial	∂
\gequal	≥	\nabla	∇
\plusmin	±	\sqrt	√
\infinity	∞	\integral	∫

In addition to these Greek and math symbols, you can specify additional characters using Unicode numbers. Visit www.unicode.org for more information about Unicode characters.

Examples of Formatted Texts

To plot the following mathematical text in the position $(1, 2)$ in the current axes

$$\sin(2\pi x_i) \approx 0 ,$$

type:

```
text(1,2,'sin(2\pix<SUB>i</SUB>) \approx 0','FontName','Arial',...
'FontSize',16)
```

This example also specifies the font name and size using the optional FontName and FontSize properties.

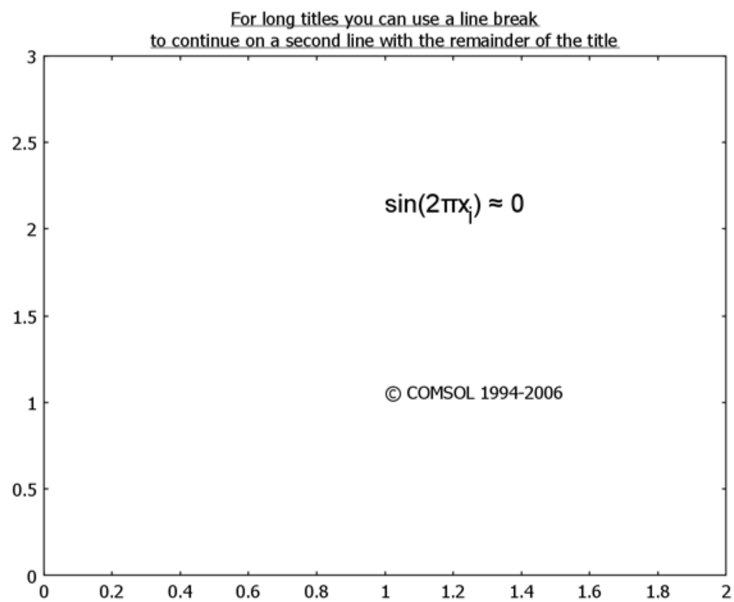
To add a text that includes the copyright symbol, you can use its Unicode:

```
text(1,1,'\u00A9 COMSOL 1994-2006')
```

To add an underlined title to a plot with the text divided into two lines, type:

```
title('<U>For long titles you can use a line break<BR>to continue
on a second line with the remainder of the title</U>')
```

`axis([0 2 0 3])` sets the axis to show all three text strings, as the following figure shows:



Axes Properties

If you have a handle to an axes object, you can read and modify properties for it using the `get` and `set` functions. The property names and their allowed values are:

TABLE 9-7: VALID PROPERTY/VALUE PAIRS

PROPERTY	**VALUE**	**DESCRIPTION**
`Box`	`On	Off`	Display an axes grid box with the plot (3D only)
`Clim`	2-element vector	The data values that map to the minimum and maximum colors in the colormap
`Climmode`	auto | manual	Determines if `Clim` is calculated automatically as the range of the plotted data, or if it is set manually

TABLE 9-7: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DESCRIPTION
`Equal`	On | Off	Determines if distances in different directions have equal length on the screen (so a circle looks like a circle)
`Fontangle`	normal | italic	Selects normal or italics font for the title, axis labels, and tick marks
`Fontcolor`	colorspec	Selects the font color for the title, axis labels, and tick marks
`Fontname`	string	Selects the font type for the title, axis labels, and tick marks
`Fontsize`	positive integer	Selects the font size for the title, axis labels, and tick marks
`Fontweight`	normal | bold	Selects normal or bold font for the title, axis labels, and tick marks
`Grid`	On | Off	Displays a grid
`Parent`	handle	The handle to the figure window that holds the axes object
`Tag`	string	The tag for retrieving the axes object
`Title`	string	The title to display above the axes object
`Xlabel` `Ylabel`	string	Label to display along the x- or y-axis

`Xlim`
`Ylim`
`Zlim`
	2-element vector	Minimum and maximum limits on the x-, y-, and z-axis

`Xlimmode`
`Ylimmode`
`Zlimmode`
	auto | manual	Determines if COMSOL Script automatically limits the x-, y-, or z-axis to fit the plotted data, or if you specify axis limits manually

`Xscale` `Yscale`	linear | log	Specifies a linear or log scale for the x- or y-axis

`Xtick`
`Ytick`
`Ztick`
	vector	Gives explicit positions for tick marks on the x-, y-, or z-axis

TABLE 9-7: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DESCRIPTION

`Xticklabel`
`Yticklabel`
`Zticklabel`
	cell array of strings	Gives explicit strings to display at the tick marks on the x-, y-, or z-axis

`Xtickmode`
`Ytickmode`
`Ztickmode`
	auto | manual	Determines if COMSOL Script calculates tick-mark positions automatically on the x-, y-, or z-axis, or if you specify them manually

2D Graphics

Overview of 2D Graphics Functions

Within an axes object you can plot a graph using vectors or matrices with *x*- and *y*-coordinates. One command can create several plots, each with a different color, line style, and marker.

Functions related to creating 2D plots are:

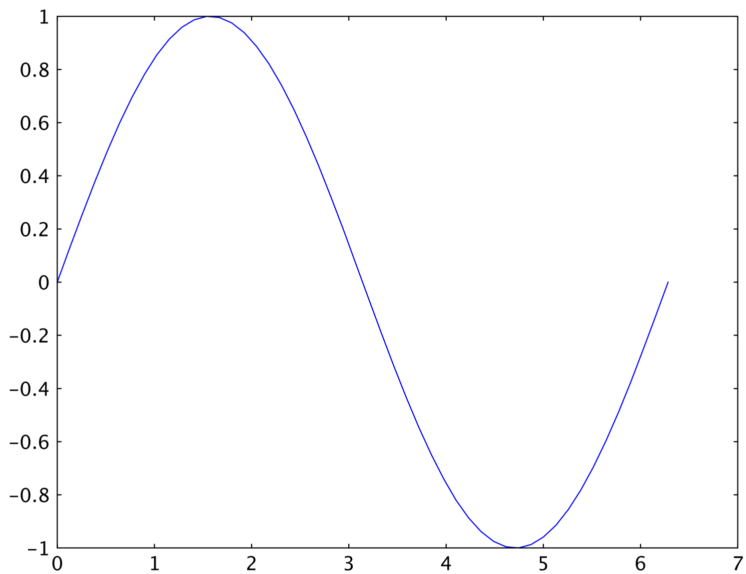
TABLE 9-8: FUNCTIONS FOR CREATING 2D PLOTS

FUNCTION NAME	DESCRIPTION
`line`	Low-level function for drawing line objects
`loglog`	Creates a plot with log scales on the x- and y-axes
`plot`	General function for plotting graphs with different colors, line styles, and markers
`semilogx`	Creates a plot with a log scale on the x-axis
`semilogy`	Creates a plot with a log scale on the y-axis

The Plot Command

The `plot` command creates 2D graphs. For example, to plot the function `sin(x)`, enter the following lines of code:

```
x=linspace(0,2*pi,100);
y=sin(x);
plot(x,y);
```



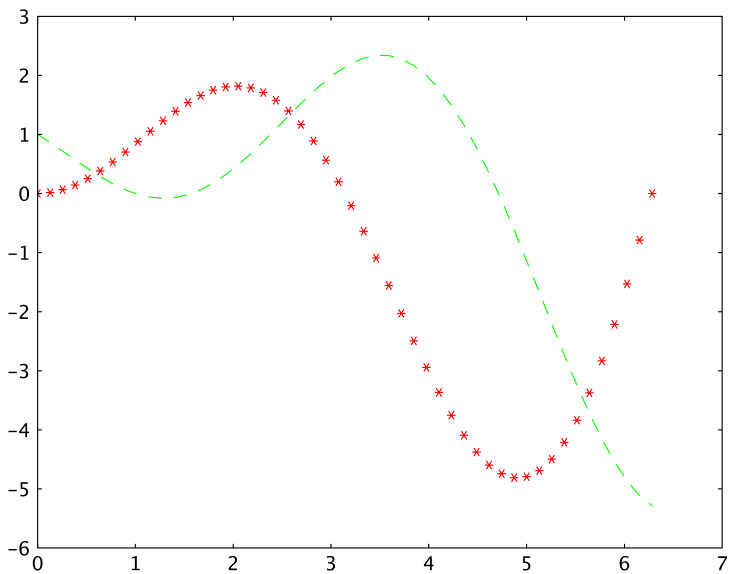
The `plot` command draws a line between the pairs of points in the vectors `x` and `y`. To plot several functions in one command, add arguments with pairs of vectors at the end:

```
x=linspace(0,2*pi,100);
y1=x.*sin(x);
y2=(1-x).*cos(x);
plot(x,y1,x,y2);
```

Each plot then gets a new color to make it easier to distinguish from other plots.

The default behavior of the `plot` command is to connect pairs of points with a solid line and to use a new color for each line. You can override this behavior by specifying an optional format string after each pair of vectors. That string controls the line color, style, and which marker to place at each point:

```
x=linspace(0,2*pi,50);
y1=x.*sin(x);
y2=(1-x).*cos(x);
plot(x,y1,'r*',x,y2,'g--');
```



The format string can contain one or more characters from the following table:

TABLE 9-9: STRINGS THAT CAN BE PART OF THE FORMAT STRING

	COLOR		MARKER		LINE STYLE
`r`	red	`+`	plus	`-`	solid
`g`	green	`o`	circle	`:`	dotted
`b`	blue	`*`	star	`-.`	dashdot
`c`	cyan	`v`	triangle	`--`	dashed
`m`	magenta	`s`	square		
`y`	yellow	`p`	pentagram		
`k`	black				

If you supply a marker string but no line-style string, the plot draws only the markers.

You can also use matrices as arguments to `plot`. In that case, each column creates a separate line. If one of the arguments is a vector that matches either the number of rows or the number of columns of the matrix in the other input argument, `plot` creates several lines:

```
x=1:0.1:5;
Y=[x ; x.^2 ; x.^3];
plot(x,Y);
```

As an alternative to the format strings, you can give property values at the end of the `plot` command.

TABLE 9-10: VALID PROPERTY-VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
color	colorspec	k	A string or an RGB triplet specifying the line color. If a string, it is one of the letters r, g, b, c, m, y, or k, meaning red, green, blue, cyan, magenta, yellow, and black, respectively
linestyle	One of the strings -, :, -., --	-	A string representing solid, dotted, dash-dot, and dashed line styles, respectively
linewidth	positive scalar	1	The line width
marker	v, +, o, *, s, p		The marker to show along the line
parent	Axes handle	gca	The axes object that gets the line

To see some of these properties in action, create a dash-dotted plot with a line width of 3:

```
x=linspace(0,10,100);
y=exp(x);
plot(x,y,'linestyle','-.','linewidth',3);
```

Plotting Complex Data

The `plot` command usually ignores the imaginary part of the input data and graphs only the real part. You can, however, pass it a single complex matrix and plot the imaginary part against the real part. In other words,

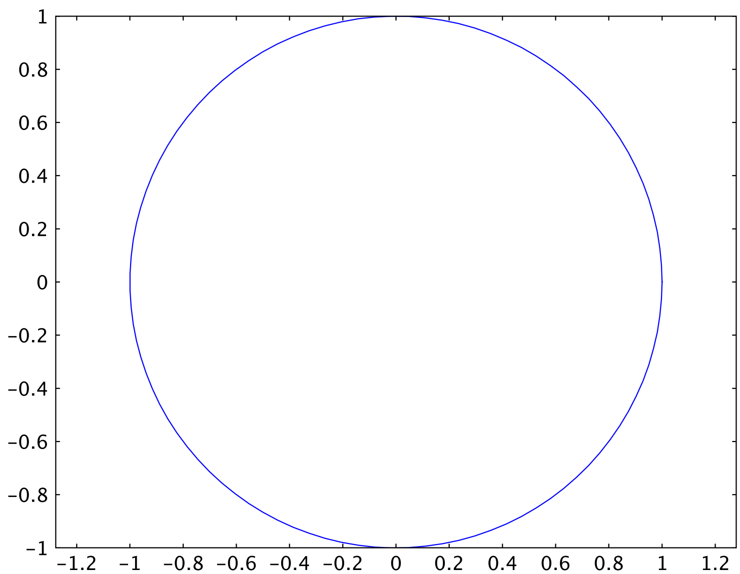
```
plot(data)
```

where the data is complex, is a shorthand for

```
plot(real(data),imag(data));
```

For instance, plot a circle using complex data:

```
x=linspace(0,2*pi,100);
Y=cos(x)+i*sin(x);
plot(Y);
axis equal;
```



Plotting Logarithmic Data

The functions `loglog`, `semilogx`, and `semilogy` have the same syntax and functionality as the `plot` function with the addition that a log scale appears on one or more of the axes. Specifically, `loglog` sets the log scale on both axes; `semilogx` sets a log scale on the x-axis; while `semilogy` sets a log scale on the y-axis.

When the data is logarithmic in y, it is preferable to use `semilogy` so as to better resolve what happens in the y direction:

```
x=linspace(0,10,100);
y=2.^x;
semilogy(x,y);
```

Low-Level Graphics

The plotting functions clear the axes object into which they plot and then add line objects corresponding to the plotted data. You can create line objects directly using the `line` function. It adds graphics objects to an axes object without deleting existing graphics objects. For example,

```
line(X,Y)
```

adds one line for each of the columns in the matrices **X** and **Y**. The property values available for the `plot` function are also available for the `line` function.

The following code snippet draws lines to form a starburst:

```
n = 12;
alpha = linspace(0,2*pi,n+1);
alpha = [alpha ; alpha+pi/n; alpha+2*pi/n];
r = [2*ones(1,n+1); ones(1,n+1); 2*ones(1,n+1)];
x = r.*cos(alpha);
y = r.*sin(alpha);
line(x,y);
```

Editing Plots

As part of the output from `plot` commands you can get handles to the corresponding graphics objects. You can then use them with the functions `get` and `set` functions to read and modify properties for the graphics objects.

The command

```
get(h)
```

returns a structure with the values of the properties for the graphics object to which `h` refers. The structure's field names correspond to the property names.

To get the value of a individual property, use

```
get(h,propname)
```

To set the value of a property, use

```
set(h,propname,value)
```

It is possible to set several property-value pairs at the same time with the `set` command by appending property-value pairs to the end of the argument list.

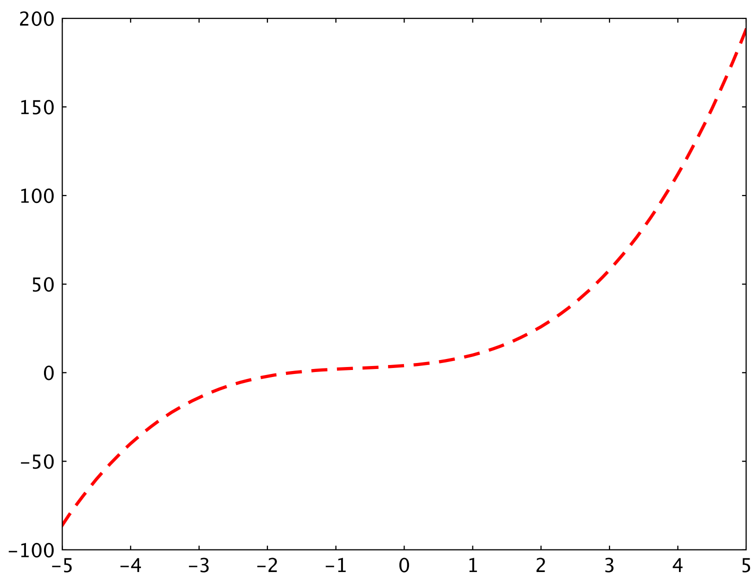
If you did not return a handle to a plot when initially creating it, you can later retrieve a handle to the graphics object with the `findobj` function. With it you can find graphic objects with a certain tag or of a certain type. You can, for example, type

```
h=findobj('type','line');
```

to return handles to all line objects.

To put some of these commands to work, create a plot and then change the color, line style, and line width:

```
x = -5:0.1:5;
y = x.^3+2*x.^2+3*x+4;
h = plot(x,y);
set(h,'color','r','linewidth',3,'linestyle',':');
```



For the line objects in 2D plots, you can get and set the properties in the following table:

TABLE 9-11: VALID PROPERTY/VALUE PAIRS FOR LINE OBJECTS

PROPERTY	VALUE	DESCRIPTION
`Color`	colorspec	A string or an RGB triplet specifying the line color. If a string, it is one of the letters `r`, `g`, `b`, `c`, `m`, `y`, or `k`, corresponding to red, green, blue, cyan, magenta, yellow, and black, respectively
`Edges`	n-by-2 matrix	A matrix with indices into vertices. Each row corresponds to a line segment
`Hold`	On | Off	Indicates if this line should be kept when new plots are added to the same axes object
`Legend`	On | Off	Indicates if a legend is displayed with a plot
`Legendstring`	string or cell array of strings.	The string to display as a plot legend. If the plot consists of several lines, this value is a cell array of strings with one legend string for each line in the plot
`Linestyle`	One of the strings `-,:,-.,--`	A string representing solid, dotted, dash-dot, and dashed line styles, respectively
`Linewidth`	positive scalar	The line width
`Marker`	`v,+,o,*,s,p`	The marker to show along the line
`Markerpos`	An integer or the string `'all'`	Tells where markers are placed on a line. It is either the desired number of markers, or the string `'all'` specifies that markers should be placed at all points in the plotted data
`Parent`	Axes handle	A handle to the axes object that holds the line
`Tag`	string	A tag that can be used to retrieve the line later on
`Type`	string	The value `'line'` indicates that it is a line object
`Vertices`	nv-by-2 matrix	The coordinates along the line. Edges contains indices into this matrix to form each edge segment (nv is the number of vertices)
`Visible`	On | Off	Indicates if the line is visible or not
`Xdata`	1-by-m matrix	The line's x-coordinates
`Ydata`	1-by-m matrix	The line's y-coordinates

Contour Plots

COMSOL Script provides a set of functions for creating contour plots (isolines). Table 9-12 summarizes the available functions:

TABLE 9-12: CONTOUR FUNCTIONS

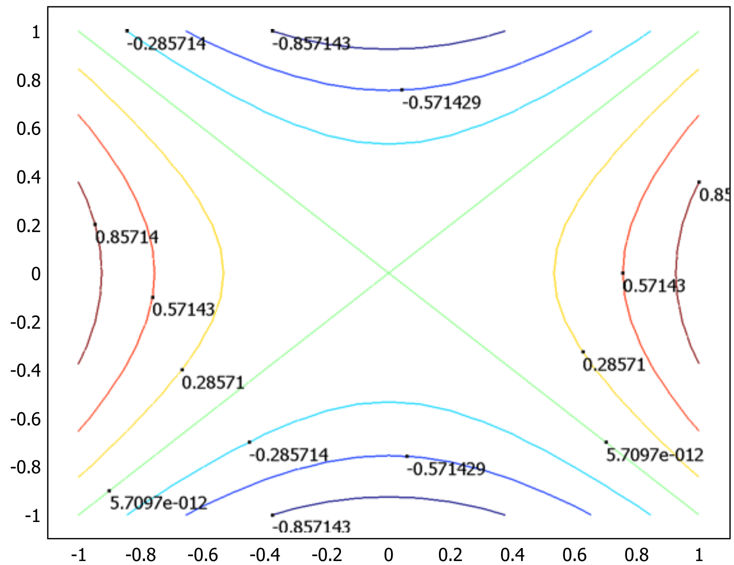
FUNCTION NAME	DESCRIPTION
`contour`	2D contour plot
`contour3`	3D contour plot
`contourc`	Contour data matrix
`contourf`	Filled 2D contour plot
`contours`	Contour data matrix (identical to contourc)
`clabel`	Contour labels

To make a contour plot with labels for the function

$$z(x, y) = x^2 - y^2$$

with both x and y in the range of -1 to 1, type:

```
[x,y] = meshgrid(-1:0.1:1,-1:0.1:1);
z = x.^2-y.^2;
c = contour(x,y,z);
clabel(c);
```



The function `contourc` (or `contours`) returns a contour data matrix that contains the contour levels and coordinates for each contour line (the contour plot function contour can also provide a contour data matrix as its first output argument, which the previous example shows). The contour label function `clabel` uses the contour data matrix as its input data. For 3D contour plots, use `contour3`.

3D Graphics

Overview of 3D Graphics Functions

Whereas 2D plots are limited to lines, 3D plots can create surface plots for functions of two variables. A 3D image can also show a patch, which is an image made up of individual triangles or quadrilaterals. An example of the use of patches is to visualize the computational mesh from a finite element analysis.

COMSOL Script also makes it easy to generate wireframe plots or add lights to a scene to improve its appearance. To eliminate the need to create multiple plots in the same figure window to accommodate 2D and 3D images, there exists a 3D plot function, `plot3`, similar to the 2D equivalent for drawing line plots.

The functions related to creating 3D plots are:

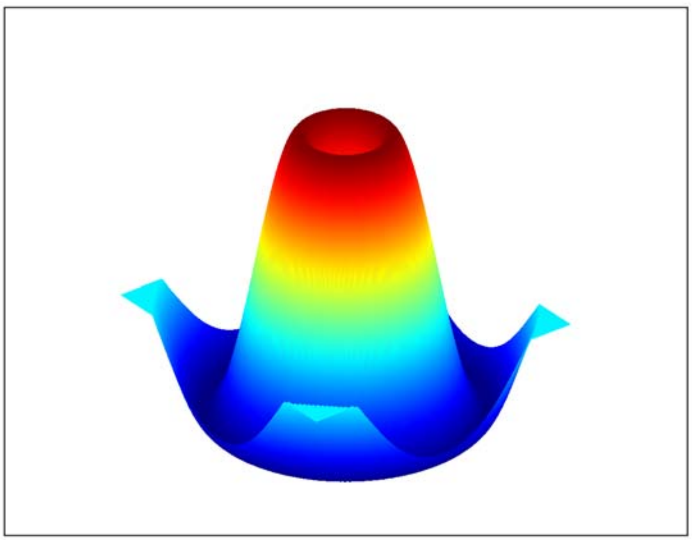
TABLE 9-13: FUNCTIONS FOR CREATING 3D PLOTS

FUNCTION NAME	DESCRIPTION
`colormap`	A colormap for coloring patches and surfaces
`hidden`	Turns hidden-line removal On and Off
`light`	Creates different types of lights
`lighting`	Turns the scene light On or Off
`material`	Controls the material to account for surface reflectance
`mesh`	Creates a colored wireframe surface of quadrilaterals
`meshz`	Creates a colored wireframe surface of quadrilaterals with a curtain
`patch`	Creates a patch consisting of triangles or quadrilaterals
`plot3`	Creates a line plot in 3D
`shading`	Controls how color is interpolated within the element of a patch or a surface
`surf`	Creates a colored surface of quadrilaterals
`surface`	Low-level function for creating a colored surface of quadrilaterals
`view`	Controls the viewpoint position

Surf and Mesh Commands

The `surf` command creates a plot from a function of two variables, giving it color and height. You supply the x, y, and z coordinates as matrices. The plot connects neighboring entries in the matrices to form each element in the colored surface. An optional fourth argument to `surf` can supply data values for coloring the surface. If you do not define this parameter, COMSOL Script varies the surface color based on its height as in the following example:

```
x1=linspace(-5,5,30);
y1=linspace(-5,5,30);
[x,y]=meshgrid(x1,y1);
r=sqrt(x.^2+y.^2);
z=sqrt(6-r).*sin(r);
surf(x,y,z);
```



The `mesh` function is the same as `surf` except that it colors only the edges between the elements and uses a white color for the elements' interior; `meshz` also adds a curtain around the plot drawn from the data points down to the lowest z value.

The `surface` function is the low-level function for creating surface objects. It has the same syntax as `surf` but does not clear the axes before adding a surface.

Colormaps and Color Bars

To color surfaces and patches you often work with a colormap. A colormap with n colors is an n-by-3 matrix of RGB values. For each data value, a script creates an index into the colormap by mapping linearly between the minimum value to the first color and the maximum value to last color. To create a colormap, call one of the functions in the following table and give the desired number of colors as an argument:

TABLE 9-14: COLORMAP FUNCTIONS

FUNCTION	DESCRIPTION
`bone`	Gray scale with a touch of blue
`cool`	Different shades of cyan and magenta
`gray`	A gray-scale colormap
`hot`	Colors from black to white, ranging through red, orange, and yellow
`jet`	All colors from dark blue to dark red, ranging through blue, cyan, green, yellow, and red
`pink`	Different shades of pink

The `Clim` property overrides the data values that map to the first and last color in the colormap. This feature can, for example, help avoid the effect of extreme values; values outside the range map to the first or the last color. A default value of `Clim` is associated with each axes object, but you can also supply a separate value as a property when creating a surface or patch.

Consider the command

```
colormap(cool(256))
```

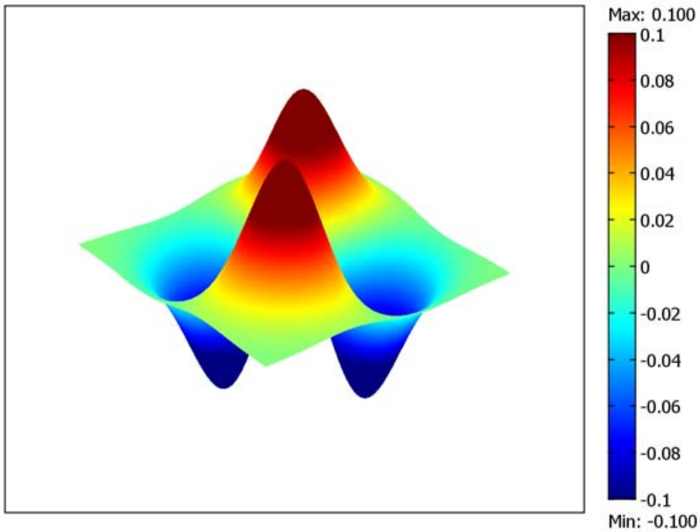
which sets the colormap for the current figure to the `cool` colormap function with 256 colors. This command affects all plots that are currently in the figure window and all plots that are added to it later on. However, each surface or patch can have a separate colormap if you pass the `Colormap` property when creating it.

The function `colorbar` displays a color scale (color bar) to the right of the plot that indicates which data value corresponds to each color in the colormap.

The following example displays the color bar, and it also uses `Clim` to specify a smaller range. Try it also without specifying the `Clim` property to see what effect it has.

```
x1=linspace(-2,2,100);
y1=linspace(-2,2,100);
[x,y]=meshgrid(x1,y1);
```

```
z=sin(x).*sin(y).*exp(-x.^2-y.^2);
surf(x,y,z,'colormap','jet(1024)','clim',[-0.1 0.1]);
colorbar
```



Patches

`patch` is the low-level function for creating a colored patch consisting of triangular or quadrilateral elements. Use

```
patch(x,y,c)
```

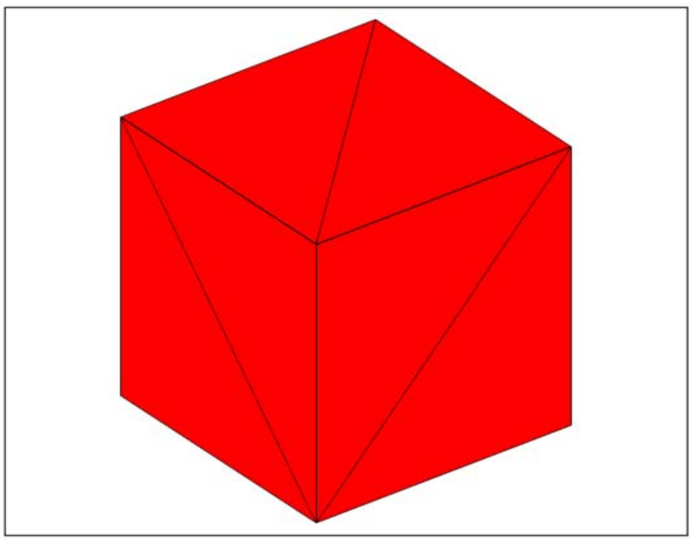
or

```
patch(x,y,z,c)
```

to create a 2D or 3D patch, respectively. The matrices `x`, `y`, and `z` have three or four rows. Each column creates an element in the corresponding patch. The matrix `c` holds data values mapped to a colormap. It either has one row or the same number of rows as `x`. If it has one row, each element gets one color with flat shading; otherwise, the corners of the elements each get a separate color with interpolated shading.

Consider the following code snippet, which create a cube using the `patch` command. The faces of the triangles are red, and the edges between them are colored black.

```
x=[0 1 1;0 1 0;0 1 1;0 1 0;0 0 0;0 0 0;
   1 1 1;1 1 1;0 0 1;0 1 1;0 0 1;0 1 1]';
y=[0 0 1;0 1 1;0 0 1;0 1 1;0 1 1;0 1 0;
   0 1 1;0 1 0;0 0 0;0 0 0;1 1 1;1 1 1]';
z=[0 0 0;0 0 0;1 1 1;1 1 1;0 0 1;0 1 1;
   0 0 1;0 1 1;0 1 1;0 1 0;0 1 1;0 1 0]';
patch(x,y,z,'r','edgecolor','k');
```



In addition to the fixed arguments, you can supply additional property-value pairs at the end of the command to further control how the patch is created. (Note that these same pairs are also applicable to the `mesh`, `meshz`, and `surf` functions.)

TABLE 9-15: PROPERTY-VALUE PAIRS FOR PATCHES AND SURFACES

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`Clim`	2-element vector		Tells which data values to map to the first and last colors in the colormap
`Colormap`	String of the type `'jet(256)'` or a matrix with 3 columns		The colormap for the patch. Either a string to be evaluated, or a matrix with one row for each color and one column for red, green, and blue values
`Edgecolor`	`none | flat | interp |` colorspec	`none`	Indicates how to color the edges between each element in the patch
`Facecolor`	`none | flat | interp |` colorspec	`interp`	Indicates how to color the interior of each element in the patch
`Parent`	Axes handle	`gca`	Determines which axes object gets the patch

In the previous table, the `Facecolor` and `Edgecolor` properties can take on one of several values as defined in the next table:

TABLE 9-16: DESCRIPTION OF FACECOLOR AND EDGECOLOR VALUES

VALUE	DESCRIPTION
`none`	Either the elements are not be filled or their edges are not drawn. `facecolor` and `none` can, for example, create a wireframe plot
`flat` or `interp`	How to interpolate the color using the vertex colors. `flat` means the entire element gets the same color; `interp` means color in the interior of the element is created by interpolation from the values at the vertices
colorspec	A string or an RGB triplet specifying the color of the entire patch. If a string, it is one of the letters `r`, `g`, `b`, `c`, `m`, `y`, or `k`, meaning red, green, blue, cyan, magenta, yellow, and black respectively

The `shading` function can serve as a shorthand for setting `Facecolor` and `Edgecolor` for existing plots in an axes object:

- `shading interp` sets `Facecolor` to `'interp'` and `Edgecolor` to `'none'`.

- `shading flat` sets `Facecolor` to `'flat'` and `Edgecolor` to `'none'`.
- `shading faceted` sets `Facecolor` to `'flat'` and `Edgecolor` to `'k'`.

Patch and Surface Properties

As with line objects, the `get` and `set` functions work with handles to patch and surface objects to read and modify the following properties:

TABLE 9-17: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DESCRIPTION
`Colorbar`	`On | Off`	Indicates if a colorbar is displayed with a plot
`Colormap`	string representing colormap	The colormap for the plot
`Edgecolor`	colorspec	How to color the edges between elements
`Facealpha`	scalar between 0 and 1	The tranparency of the patch or surface; 1 means no transparency, and 0 means full transparency
`Facecolor`	colorspec	How to color the interior of the elements
`Faces`	n-by-3 matrix or n-by-4 matrix	A matrix with indices into vertices. Each row corresponds to an element of the patch or surface
`Facevertexcdata`	nv-by-3 matrix	RGB values for the colors at the vertices of the elements (nv is the number of vertices)
`Hold`	`On | Off`	Indicates if the patch / surface should be kept when new plots are added to the same axes object
`Parent`	Axes handle	The handle to the axes object that holds the line
`Tag`	string	A tag that can help retrieve the line later on
`Type`	string	The value `'patch'` or `'surface'` indicating that it is a patch or surface object
`Vertices`	nv-by-2 matrix	Coordinates along the line. Faces contains indices into this matrix to form each element (nv is the number of vertices)
`Visible`	`On | Off`	Indicates if the patch or surface is visible
`Xdata`	3-by-n matrix or 4-by-n matrix	The x-coordinates of the patch or surface

TABLE 9-17: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DESCRIPTION
`Ydata`	3-by-n matrix or 4-by-n matrix	The y-coordinates of the patch or surface
`Zdata`	3-by-n matrix or 4-by-n matrix	The z-coordinates of the patch or surface

Lights and Materials

It is possible to add lights and materials to create plots with visual highlights and to make them more attractive. In the plots, you can turn on a *headlight*, which is positioned at the camera and directed toward the target., and *scene light***,** which radiates from a distance. COMSOL Script provides four kinds of scene lights:

- Ambient light—seems to come from all directions.
- Directional light—shines in a certain direction from infinity.
- Point light—shines equally in all directions from a certain position.
- Spotlight—an attenuated source that shines in a certain direction from a certain position. A spread angle and a concentration specify how quickly the light attenuates for directions close to the specified direction.

To create one of these sources, take the `light` function and pass the following property values:

TABLE 9-18: PROPERTY/VALUE PAIRS TO USE WHEN CREATING A LIGHT

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`color`	colorspec	`w`	A string or an RGB triplet specifying the light's color. If a string, it is one of the letters `r`, `g`, `b`, `c`, `m`, `y`, or `k`, meaning red, green, blue, cyan, magenta, yellow, and black, respectively
`concentration`	Real value between 0 and 128	`0`	The concentration for a spotlight
`direction`	A 3-element array	`[0 0 1]`	The direction for a directional light and a spot light
`parent`	Axes handle	`gca`	Indicates the axes object that gets the light

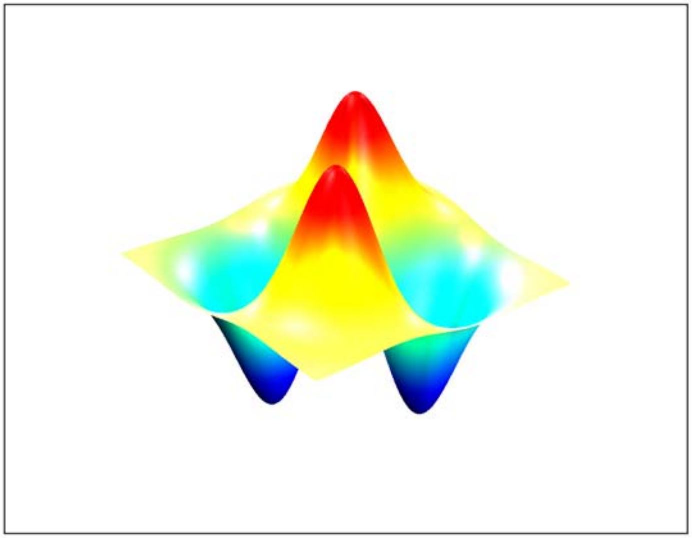
TABLE 9-18: PROPERTY/VALUE PAIRS TO USE WHEN CREATING A LIGHT

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`position`	A 3-element array	`[0 0 0]`	The position for a point light or a spotlight
`style`	`ambient` | `directional` | `point` | `spot`	`point`	The type of light to create
`spread`	Real number between 0 and pi	`pi`	The spread angle for a spotlight

To turn scene light on, either call `lighting('phong')` or click the **Scene Light** button on the toolbar in the figure window. You can also activate a headlight, which is "mounted" on the camera and looking in the same direction as the camera, by clicking the **Headlight** button on the toolbar.

This example creates a surface plot and adds some lights:

```
x1=linspace(-2,2,100);
y1=linspace(-2,2,100);
[x,y]=meshgrid(x1,y1);
z=sin(x).*sin(y).*exp(-x.^2-y.^2);
surf(x,y,z);
light('style','point','position',[-2 -2 1],'color','g');
light('style','directional','direction',[0 0 -1],'color','r');
lighting phong;
```



Another way to influence the appearance of a surface or patch is to select a material and thereby control how the graphic objects reflect light. To apply a material to all surfaces and patches in an axes object, use the `material` function, and then pass appropriate values to control the material's properties.

TABLE 9-19: PROPERTY VALUE PAIRS FOR CREATING A MATERIAL

PROPERTY	VALUE	DESCRIPTION
`Ambient`	colorspec	Specifies the ambient color (the color reflected from the surface due to ambient light). A string or an RGB triplet specifies the color of the light. If a string, it is one of the letters `r`, `g`, `b`, `c`, `m`, `y`, or `k`, meaning red, green, blue, cyan, magenta, yellow, and black, respectively
`Diffusive`	colorspec	Specifies the diffusive color
`Emissive`	colorspec	Specifies the emissive color, the color of the light that the surface or patch emits

TABLE 9-19: PROPERTY VALUE PAIRS FOR CREATING A MATERIAL

PROPERTY	VALUE	DESCRIPTION
`Specular`	colorspec	Specifies the specular color, which is the highlight color on the surface or patch
`Shininess`	Real number > 0 and <128.	Specifies the shininess

Compare the same plot with four different types of material applied to it.

```
x1=linspace(-2,2,100);
y1=linspace(-2,2,100);
[x,y]=meshgrid(x1,y1);
z=exp(cos(x).*sin(y));
subplot(2,2,1);
surf(x,y,z);
lighting phong;
material('shininess',10);
subplot(2,2,2);
surf(x,y,z);
lighting phong;
material('specular',[0 1 0],'shininess',100);
subplot(2,2,3);
surf(x,y,z);
lighting phong;
material('diffusive',[1 0 0],'shininess',100);
subplot(2,2,4);
surf(x,y,z);
lighting phong;
```

```
material('emissive','g','shininess',100);
```

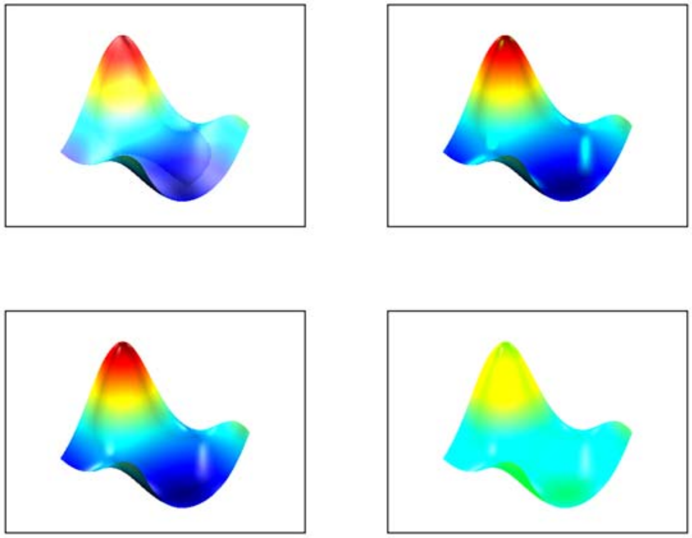


Figure 9-1: Surface plots using four different settings for how the material reflects light.

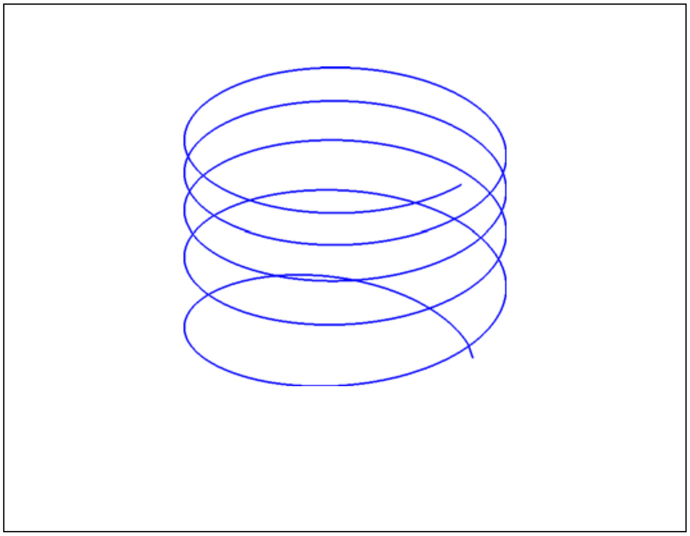
3D Plots and Lines

The `plot3` function is similar to the `plot` function in 2D except for an argument that accounts for the *z*-coordinate. In addition, you can use the `line` function with a third argument for the z-coordinate to create a line in 3D. You can set the color and line width for 3D lines, but line styles and markers are not available.

This code snippet plots a spiral in 3D:

```
t=0:0.05:30;
x=cos(t);
y=sin(t);
z=sqrt(t);
```

```
plot3(x,y,z,'b','linewidth',2);
```



Specifying the View

To specify the viewpoint for a plot, choose the `view` function (for 3D plots only). The command `view(3)` positions the camera in a default 3D view, while `view(2)` produces a default 2D view. Other options include `view('xy')`, `view('yz')`, and `view('zy')` to examine a plot in a certain plane.

Further, `view(azimuth,elev)` positions the camera at a certain azimuth and elevation for a 3D plot.

Another option for 3D plots is to move the cursor to position the camera. Hold down a mouse button and then move the mouse over the axes object. On a 3-button mouse button, the buttons have the following effects on the camera:

- left—orbit the camera around the plot
- middle—zoom in and out in the plot
- right—pan the camera in the current view plane

Camera View Angle, Target, Position, and Up Vector

To specify and retrieve the exact view angle, position, target, and up vector for the camera in a 3D plot, use the functions `camva`, `campos`, `camtarget`, and `camup`, respectively.

Use `camva` to set the camera view angle in degrees for the current axes. Increasing and decreasing the camera view angle correspond to zooming out and zooming in, respectively. For example, type

```
camva(camva-2)
```

to zoom in by decreasing the camera view angle by two degrees.

The camera target is a vector of coordinates for the location in the plot that the camera points to, regardless of the camera's position. Query and set the camera target using `camtarget`. Use `campos` to do the same with the camera position. For example,

```
campos(campos+[0 0.1 0]);
camtarget(camtarget+[0 0.1 0]);
```

moves both the camera position and the target position a distance of 0.1 in the y direction.

The camera up vector is a vector or coordinates that defines the direction in the plot that is oriented upward. Use the `camup` function to get or specify the camera up vector. For example,

```
up = camup;
```

stores the current camera up vector in the variable `up`.

Working With Images and Movies

Image and Movie Functions and Formats

This section describes how you can use COMSOL Script to read and display images, save images, and create movies.

Table 9-20 provides an overview of the functions for images and movies:

TABLE 9-20: IMAGE AND MOVIE FUNCTIONS

FUNCTION NAME	DESCRIPTION
`image`	Show an image
`imagesc`	Show an image using scaled mapping
`imread`	Read an image from file
`imshow`	Show an image
`imwrite`	Write an image to file
`movie`	Create a movie
`saveimage`	Save a plot as an image

IMAGE FORMATS

The `imread`, `imwrite`, and `saveimage` functions can read and write images on the following formats:

- BMP (Windows Bitmap) using extension `.bmp`
- JPEG (Joint Photographic Experts Group) using extension `.jpg` or `.jpeg`
- PNG (Portable Network Graphics) using extension `.png`
- TIFF (Tagged Image File Format) using extension `.tif` or `.tiff`
- EPS (Encapsulated PostScript) using extension `.eps` (only supported by the `saveimage` function)

The BMP and TIFF formats are only available on 32-bit Windows, Linux, Solaris, and Macintosh.

IMAGE DATA STRUCTURE

COMSOL Script stores image data as a height-by-width-by-3 matrix with RGB values for each pixel in the image. When you read an image from file using `imread`, a matrix of the `uint8` data type represents the RGB data as values between 0 and 255. The RGB values can also be a matrix of doubles (the default floating-point data type) between 0 and 1.

For information about movie formats and movie objects, see “Generating Movies” on page 234.

Reading and Displaying Images

Use `imread` to read images from file. The input to the function is the file name, including extension, and the output is a COMSOL Script variable that stores the images data. For example,

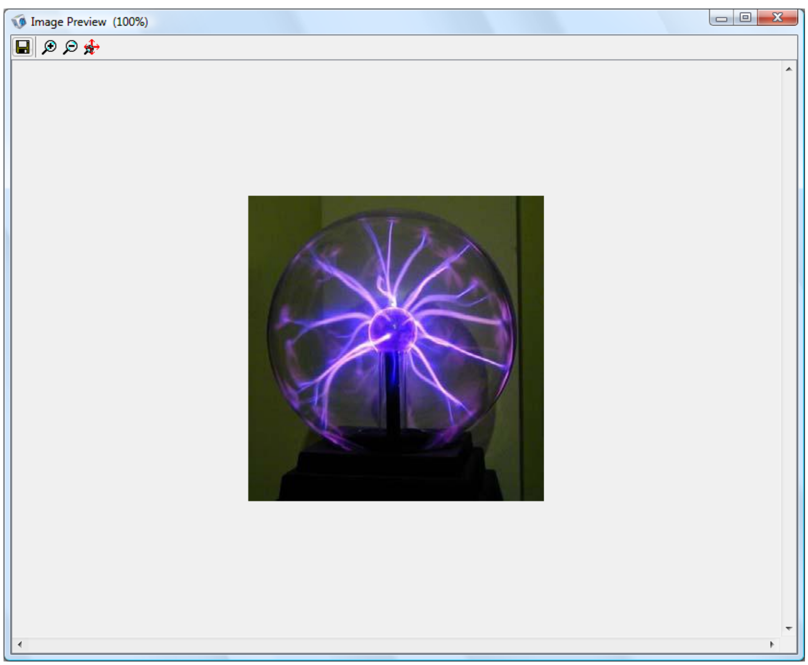
```
Img = imread('plasma_discharge.jpg');
```

reads the image data from the file `plasma_discharge.jpg`, which contains a JPEG image.

To display such an image, use the `imshow` function:

```
imshow(Img);
```

This brings up an **Image Preview** window:



For showing smaller images you can also use `image` and `imagesc`, where the latter uses a scaled mapping when mapping the data values to the colormap.

Saving Images

There are two functions that you can use to save an image to file: `imwrite`, which writes image data to a file, and `saveimage`, which saves the contents of a plot as an image.

To determine the image format, use the extension of the file name that you pass to imwrite. For example,

```
imwrite(Img,'picture1.tif')
```

writes the image stored in the variable `Img` to the file `picture1.tif` using the TIFF format.

SAVING PLOTS AS IMAGES

The `saveimage` function stores the contents of a figure window as an image. The following table contains some of the property-value pairs that you can use to control how the image is generated.

TABLE 9-21: PARTIAL LIST OF PROPERTY/VALUE PAIRS FOR THE SAVEIMAGE FUNCTION

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`figure`	handle	current figure	The handle of the figure window that contains the image of interest
`height`	Positive integer	600	The image height
`type`	`bmp | jpeg | png | tiff | eps`	`jpeg`	The type of image to create
`width`	Positive integer	800	The image width

For instance, to save a 1024x768 TIFF image `lineplot.tiff` in the current working directory of the line plot created with `plot` below, use this code:

```
x=linspace(0,10,100);
y=sin(x);
plot(x,y);
saveimage('lineplot','type','tiff','width',1024,'height',768);
```

To save a 600x800 JPEG image of the plot in the current figure window, use this code:

```
saveimage('currentplot');
```

EXPORTING IMAGES

You can also save an image by clicking the **Export Image** button on the toolbar at the top of the figure window. Doing so brings up a dialog box in which you specify a file name and type as well as the desired width and height for the image. Using the **Export Image** dialog box, you can control font sizes, line widths, and what to include in the image you export. To speed up the image-generation process, a preview feature and image rendering information are available.

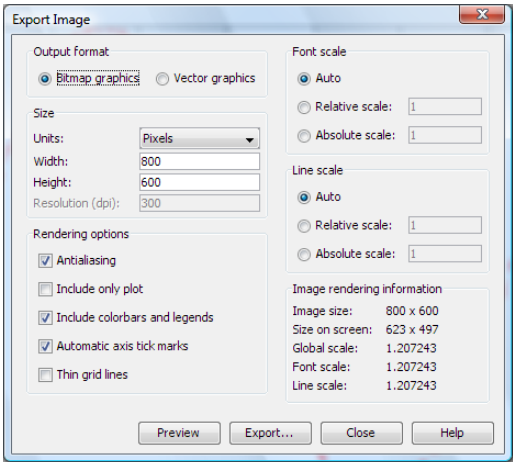


Figure 9-2: The Export Image dialog box.

SETTINGS IN THE EXPORT IMAGE DIALOG BOX

Output Format

In the **Output format** area, click the **Bitmap graphics** button to export an image using a bitmap-based formats such as TIFF and JPEG. You select the format after clicking the **Export** button. Click the **Vector graphics** button to save the image as an EPS file (Encapsulated PostScript).

Image Size

In the **Image size** area, you can specify the size of the image. Select the unit from the **Units** list: inches, centimeters, or pixels (pixels are available for bitmap graphics only).

When using inches or centimeters, you can set the size using the **Width**, **Height**, and **Resolution (dpi)** edit fields. When you use pixels as the unit, the **Resolution (dpi)** edit field is not available

Scaling

The settings in the **Font scale** and **Line scale** areas affect the scaling between the plot's size on the screen and the size in the image (size = number of pixels):

- Click **Auto** to use the global scale (you see its value in the **Image rendering information** area; also see "The Global Scale" on page 234) if you specify the size in pixels (the software scales text, lines, and other graphics equally). If you specify the size in centimeters or inches, the automatic scale is based on the resolution that you. The font size and line width that you specify when creating the plot should be preserved if you export an image using a certain resolution in dpi (dots per inch) and import it to a document as an image using the same dpi number (a text with a certain size in the plot will look like a text with the same size in the document).
- Click **Relative scale** to use a total font scale that is the automatic scale times the relative scale that you specify.
- Click **Absolute scale** to use a total font scale that is equal to the absolute scale that you specify.

RENDERING OPTIONS

The following settings are available in the **Rendering options** area:

Antialiasing

Select the **Antialiasing** check box to reduces stairstep-like lines (jaggies) and makes lines and edges look smooth

Include Only Plot

Select the **Include Only Plot** check box to include only the graphics objects in the drawing area, excluding colorbars, axes, tick marks, titles, and labels.

Include Colorbars and Legends

Select the **Include Colorbars and Legends** check box to include colorbars and legends, if present.

Automatic Axis Tick Marks

Select the **Automatic axis tick marks** check box to take advantage of a new feature that can hide axis tick marks if they overlap. Clear this check box if you want make sure that the image has the same axis tick marks as you see on the screen.

Rendering Thin Grid Lines

Select the **Thin grid lines** check box to render thin grid lines compared to other lines in the image. Use this option if you think that the grid is too dominating in the image.

Including the 3D Axis

Select the **Include 3D axis** check box to include the grid and coordinate system in the image.

THE GLOBAL SCALE

The global scale, which you find in the **Image rendering information** area, is the scale between the size of the plot on screen and the size of the image (size = number of pixels).

Generating Movies

COMSOL Script generates movies in either AVI or QuickTime format.

The command `m = movie(...)` creates a movie-generation object. You can then add frames to the movie from plots in figure windows. The properties `width` and `height` specify a desired width and height for the movie; if left unspecified, the movie size becomes the default size of 640x480 pixels.

You can interact with a movie-generation object with these methods:

TABLE 9-22: METHODS FOR MOVIE-GENERATION OBJECTS

METHOD	DESCRIPTION
`m.addFrame`	Adds the plot in the current figure window as a frame in the movie
`m.addFrame(h)`	Adds the plot in the figure window with handle `h` as a frame in the movie
`m.setFrameRate(rate)`	Sets the frame rate when generating the movie
`m.setQuality(qual)`	Sets the quality when generating the movie; `qual` is a real number between 0 and 1, where 1 is the best quality
`m.setFileType(type)`	Sets which type of movie to generate; `type` can be `'avi'` or `'quicktime'`
`m.listEncodings`	Displays a list of available encoding formats
`m.setEncoding(encoding)`	Sets which encoding format to use
`m.generate(filename)`	Generates a movie with the name `filename` from the frames that have been added to the movie

The following example generates an AVI movie at 800x600 pixels. It employs the `axis` command to set fixed limits and a view for all the frames:

```
x1=linspace(-5,5,100);
y1=linspace(-5,5,100);
[x,y]=meshgrid(x1,y1);
r=sqrt(x.^2+y.^2);
z=sqrt(6-r).*sin(r);
h=surf(x,y,z);
axis([-5 5 -5 5 -1 2]);
m=movie('width',800,'height',600);
for i=1:20,
  delete(h);
  h=surface(x,y,z*(1-i/10));
  m.addFrame;
end
m.generate('movie.avi');
```

10

Solving Differential Equations

Ordinary differential equations (ODEs) appear when using the method-of-lines to solve time-dependent partial differential equations in COMSOL Multiphysics. There are many other cases where ODEs and differential-algebraic equations (DAEs) provide good mathematical models of dynamic systems. This chapter describes the tools for solving ODEs and DAEs using COMSOL Script. For partial differential equations (PDEs), see the COMSOL Multiphysics documentation.

ODEs and DAEs

Introduction

ODEs are ordinary differential equations, that is, differential equations that only depend on one independent variable. Often ODEs describe dynamic processes, and the derivatives are with respect to time. When you also supply an initial condition, this forms an *initial-value problem*. A DAE is an extension of ODEs, where some equations are algebraic (no derivatives of the dependent variables appear). DAEs occur, for example, when solving the time-dependent Navier-Stokes equations using the method of lines.

Using the DASPK Solver

COMSOL Script includes the DAE solver DASPK created by Linda Petzold at University of California, Santa Barbara (see Ref. 1 and Ref. 2). The solver is an implicit time-stepping scheme, which means that it must solve a possibly nonlinear system of equations at each time step. It is well suited for solving stiff and nonstiff initial-value problems, including nonlinear problems. Use the `daspk` function to solve ODEs and DAEs for initial-values problems on the following form:

$$M(t, y)\dot{y} = f(t, y)$$

with initial conditions $y(t_0) = y_0$. The mass matrix M is the unit matrix by default. For DAEs, M is typically singular. y represents the dependent variables and is a vector for a system of ODEs. To solve higher-order ODEs, rewrite them into a system of first-order ODEs.

The syntax of `daspk` is:

```
[t, y] = daspk(f, tlist, y0)
```

where `f` is the name of a function that implements the right-hand side f. `tlist` contains a set of times for which you want the solution data (as a row vector). If you provide a vector with two entries, they are define the interval on which daspk solves the ODE or DAE. In that case, `t` contains all the internal time steps.

Using an additional input argument, you can provide various solver options in a structure variable. See “Setting and Retrieving ODE Solver Options” on page 239 for more information about the options structure.

The syntax for the function that implements `f` is the following:

```
function ydot = myode(t,y)
```

Notice that the function takes the input arguments `t` and `y`, even if the equations may not include `t` explicitly.

The input `t` is the independent variable and is a scalar.

The input `y` is a column vector of the same length as the number of dependent variables.

You can also have a function `f` that takes additional input arguments that can be, for example, parameters in the ODE or DAE. You then supply those as additional input arguments to `daspk`. See "Solving the van der Pol Equation" on page 243 for an example.

Setting and Retrieving ODE Solver Options

The options structure that you can provide as an input to `daspk` contains the following fields:

- `AbsTol`: The absolute tolerance—a scalar value or a vector with one value for each dependent variable in `y`. The default value is 1e-6.
- `Complex`: If you set this to True, `daspk` assumes that the solution is complex even if the initial value is real.
- `IntialStep`: Initial step size (a positive scalar value). By default, `daspk` computes this value automatically.
- `Jacobian`: A matrix or name of function that describes the Jacobian $\frac{\partial f}{\partial y}$.
- `Mass`: Matrix or name of function that computes the mass matrix $M(t, y)$. If you do not provide a mass matrix, `daspk` uses the unit matrix.
- `MaxOrder`: The maximum order of the backward differentiation formula (BDF), which must be an integer between 1 and 5. The default value is 5.
- `MaxStep`: Maximum step size (a positive scalar value). By default, the maximum step size is one tenth of the interval in `tlist`.
- `OutputFcn`: The name of a callback function that `daspk` invokes after each step.
- `RelTol`: The relative tolerance——a scalar positive value. It can also be a vector with one value for each dependent variable in `y` that `daspk` interprets as weights for a

single scalar relative tolerance value.The default value is 1e-3 (or 0.01% relative tolerance).

- `Stats`: Display statistics for the computational effort after computing the solution. Set it to `on` to display the statistics. The default value is `off`.

To set these values, provide them as pairs of option names (field names) and option values using the `odeset` function. The fields for options that you do not specify contain the empty matrix and `daspk` then uses the default value:

```
opts = odeset('abstol',1e-2,'maxstep',0.1,'stats','on')

opts =

        AbsTol:  [0.010000]
       Complex:  []
   InitialStep:  []
      Jacobian:  []
          Mass:  []
      MaxOrder:  []
       MaxStep:  [0.100000]
     OutputFcn:  []
     OutputSel:  []
        RelTol:  []
         Stats:  'on'
```

The input for the option names (field names) in not case sensitive. To update only certain options, pass the existing options structure as the first input argument:

```
newopts = odeset(opts,'abstol',5e-2);
```

To check the value of an ODE option in the options structure, use the `odeget` function:

```
atol = odeget(opts,'abstol');
```

Solving the Lotka-Volterra Equations

As a first example, consider the *Lotka-Volterra equations*, which describe population fluctuations in predators and preys that interact. This results in two coupled nonlinear ODEs for the prey populations (y_1) and the predator population (y_2). Four parameters control the dynamics: the growth rate of prey (R_1), the rate at which predators kill prey (R_2), the death rate of predators (R_3), and the rate of increase in predator population due to prey consumption (R_4):

$$\begin{cases} \dot{y}_1 = R_1 y_1 - R_2 y_1 y_2 \\ \dot{y}_2 = -R_3 y_2 + R_4 y_1 y_2 \end{cases}$$

To simplify the equations, set all four parameters to 1. Then the function that describes f, the right-hand side in the equations above:

```
function ydot = lotkavolterra(t,y)
ydot = zeros(2,1);
ydot(1) = y(1)-y(1).*y(2);
ydot(2) = -y(2)+y(1).*y(2);
```

Notice that you need to use pointwise multiplication (`.*`), because `y` is a vector.

If you save this as `lotkavolterra.m` in a directory that is on the M-file path, you can call daspk to compute the solution from 0 to 10, using a starting value of 2 for the prey population and 1 for the predator population:

```
[t,y] = daspk('lotkavolterra',[0 10], [2; 1]);
plot(t,y)
title('Solution to the Lotka-Volterra equations')
```

This gives the following plot:

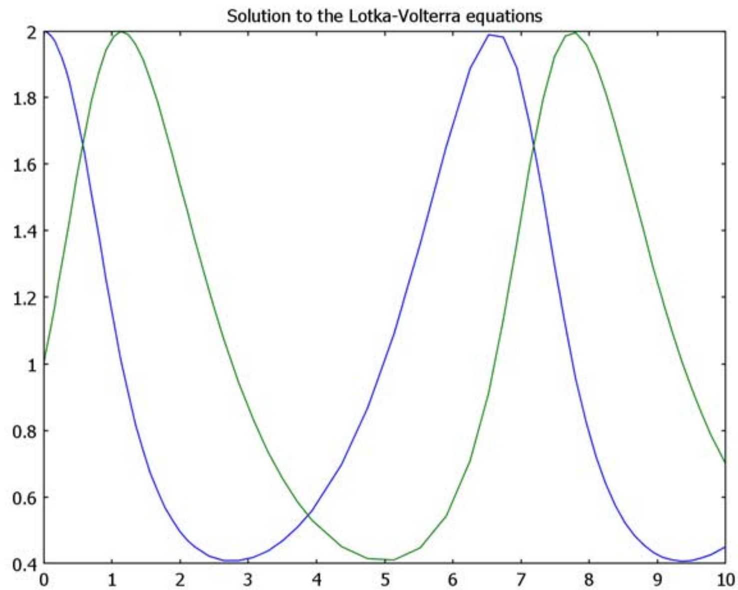


Figure 10-1: Prey population (blue) and predator population (green).

To plot a phase curve (y_2 as a function of y_1), type:

```
plot(y(:,2),y(:,1))
```

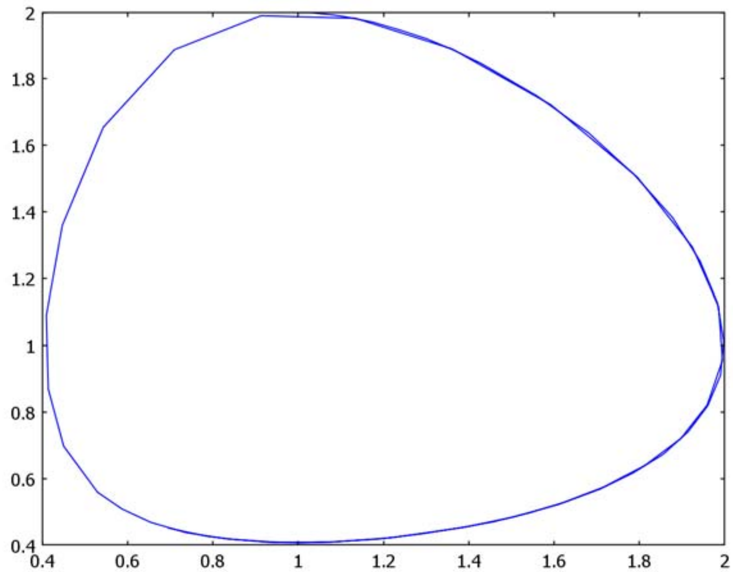


Figure 10-2: The phase curve for a solution to the Lotka-Volterra equations.

Solving the van der Pol Equation

The v*an der Pol equation* is the following second-order PDE:

$$\ddot{y} = \mu(1 - y^2)\dot{y} + y$$

It describes self-sustaining oscillations. The parameter μ has a great impact on the solution: A high value of μ makes the ODE "stiff," and it exhibits faster changes in the solution. In the special case of $\mu = 0$, it is the equation for a simple harmonic motion.

For this example, you must rewrite the second-order ODE into a system of two first-order PDEs:

$$\dot{y}_1 = \mu(1 - {y_2}^2)y_1 - y_2$$
$$\dot{y}_2 = y_1$$

where y_2 is y in the original equation, and y_1 is the time-derivative of y.

In this case, it is also interesting to vary the parameter μ. It is therefore an extra input argument to the function `vanderpol.m` that implements the right-hand side:

```
function ydot = vanderpol(t,y,mu)
ydot = zeros(2,1);
ydot(1) = mu*(1-y(2).^2).*y(1)-y(2);
ydot(2) = y(1);
```

It is also of interest to get some statistics for the computational cost. Therefore, turn on the statistics option:

```
opts = odeset('stats','on');
```

Call `daspk` with a time interval of 0 to 100, a small initial value for y, and $\mu = 0.1$:

```
[t,y] = daspk('vanderpol',[0 100], [0;0.0001],opts,0.1);
277 time steps taken
314 residual evaluations
6 Jacobian evaluations
302 linear system solutions
```

The solver reports the number of time steps, the number of residual evaluations, the number of Jacobian evaluations, and the number of linear system solutions. Figure 10-3 contains a plot of the solution.

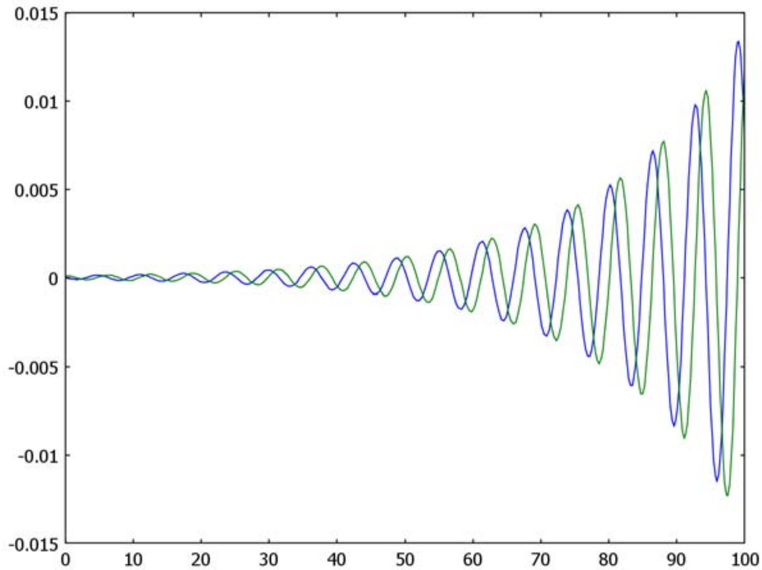


Figure 10-3: The solution to the van der Pol equation with $\mu = 0.1$.

In Figure 10-4 you can see the phase curve for this solution:

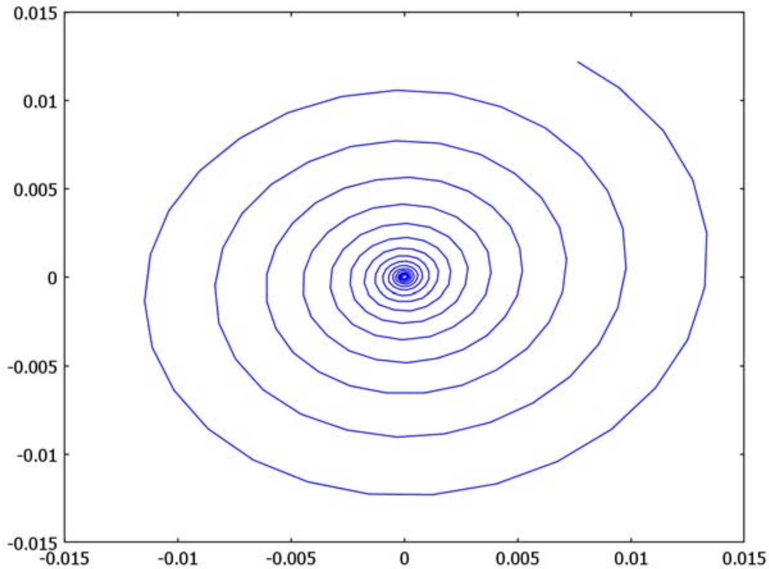


Figure 10-4: Phase curve for the solution to the van der Pol equation with $\mu = 0.1$.

If you increase μ to 1, the call is:

```
[t,y] = daspk('vanderpol',[0 100],[0;0.0001],opts,1);
1095 time steps taken
4069 residual evaluations
864 Jacobian evaluations
3113 linear system solutions
```

As you can see from the statistics information, the computational cost increases. Figure 10-5 shows the solution:

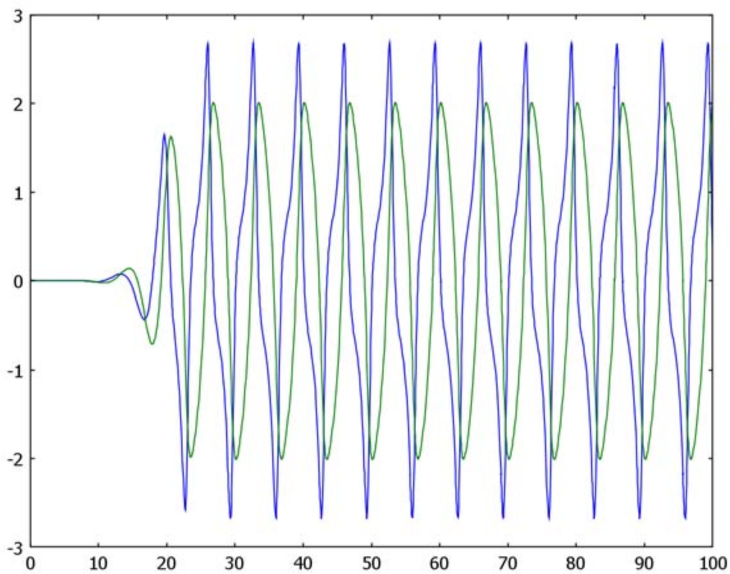


Figure 10-5: The solution to the van der Pol equation with $\mu = 1$.

Finally, increasing the value of μ to 100 makes the ODE much more stiff, and the dynamics change so that you get an abrupt change in the solution after some time. This To see this, increase the solution interval to 1000 and plot only y_2. You also have to use stricter tolerances than the default values:

```
opts = odeset(opts,'abstol',1e-9,'reltol',1e-9);
[t,y] = daspk('vanderpol',[0 1000],[0;0.0001],opts,100);
13041 time steps taken
31585 residual evaluations
5121 Jacobian evaluations
26166 linear system solutions
plot(t,y(:,2))
```

Figure 10-6 shows how the solution behaves in this case:

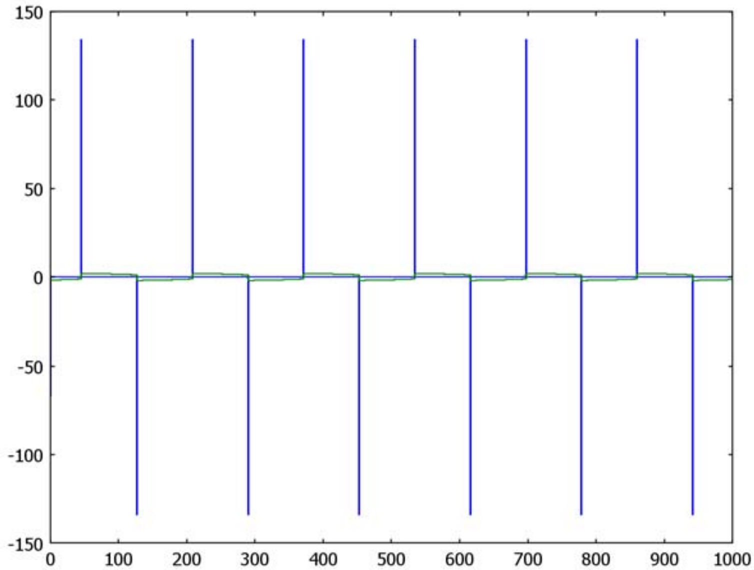


Figure 10-6: The solution to the van der Pol equation with $\mu = 100$.

References

1. .Brown, P. N., Hindmarsh, A. C., and Petzold, L. R., “Using Krylov Methods in the Solution of Large-Scale Differential-Algebraic Systems,” *SIAM J. Sci. Comput.*, vol. 15, pp. 1467–1488, 1994.

2. http://www.netlib.org/ode

11

Creating User Interfaces

The COMSOL Script environment includes Java-based tools for creating customized graphical user interfaces (GUIs) that can make calls to other COMSOL Script functions as well as to functions within COMSOL Multiphysics. In this way you can build sophisticated scripts or mathematical models and place a user interface around them that provides access only to those parameters that might need changing or that automatically present results in a very specific way. Introducing you to these tools, this chapter contains an overview of the various components and functions as well as an example of building a small user interface using COMSOL Script.

Frames and Dialog Boxes

A *frame* is the main window for an application. You supply the text to display on the title bar as the first argument to the `frame` function.

You can add menus to a frame, which also acts as a *panel* to which you can add user interface components. A dialog box has a frame as its parent but otherwise behaves the same as a frame. Use the `Size` property to specify the desired size of a frame or dialog box. If you give no size, COMSOL Script makes it the smallest size that fits all the components you have added to it. Frames and dialog boxes are invisible while you are adding components to them; call the `show` method to display the frame or dialog box when you have added all the desired components.

Menus

The `menu` function creates main menus, which appear at the top of a frame or dialog box. You pass the string to display on the menu as an argument. To create individual menu items that drop down from the main menu use the `menuitem` function. Here you pass the string to display as the first argument plus the name of the function to execute when the menu item is selected as the second argument.

As an exercise, create a small application that has a several menus and items:

```
f=frame('My Application','size',[300 150]);
m1=menu('File');
m1.add(menuitem('New','newaction'));
m1.add(menuitem('Exit','exitaction'));
m2=menu('Help');
m2.add(menuitem('Help','helpaction'));
m2.addSeparator;
m2.add(menuitem('About','aboutaction'));
f.addMenu(m1);
f.addMenu(m2);
f.setAlignment('center');
f.add(label('tag','label'));
f.show;
```

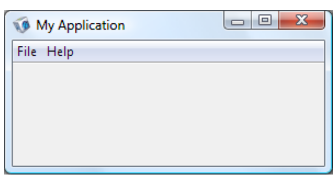
Create this script in a text editor and save it as an M-file somewhere on your M-file path.

Next you must add the code for the functions these menu items call. Define the functions `newaction`, `exitaction`, `helpaction`, and `aboutaction` as follows:

```
function newaction(event)
frame=event.parent;
frame.get('label').setText('New');

function exitaction(event)
frame=event.parent;
frame.close;

function helpaction(event)
frame=event.parent;
frame.get('label').setText('Help');

function aboutaction(event)
frame=event.parent;
dlg=dialog('About','parent',frame);
dlg.add(label('About My Application.'),1,1);
dlg.setAlignment('center');
dlg.add(button('OK'),2,1);
dlg.show;
```

Save each of these functions in separate M-files: `newaction.m`, `exitaction.m`, `helpaction.m`, and `aboutaction.m` on the M-file path.

The menu is now complete. Choosing the **New** and **Help** menu items updates the text on the label in the main area. The **Exit** menu item closes the frame, and the **About** menu item opens a new smaller dialog box (this code does not define an action for the **OK** button in the **About My Application** dialog box; see"Event Handling" on page 269 for information about how to associate an action with events such as mouse clicks).



Storing Application Data

When you write an application you usually have some kind of data structure that you want to access from different event handlers in the user interface. Use the `storedata` and `getdata` functions to accomplish this.

```
storedata(f,data)
```

stores the variable `data` in `f`, which can be a frame or a dialog box. Retrieve the variable later on using

```
getdata(f)
```

`data` can be any of the data types supported by COMSOL Script, but it is usually best to let it be a structure.

User Interface Components

Introduction

To make a dialog box or panel useful, you must populate it with components that accept operator instructions and display results. The GUI component library in COMSOL Script supplies functions for creating most of the user-interface components that are available in Java. These functions return a Java object that contains the corresponding Java Swing component. You can then interact with the objects by calling methods on them using the Java interface in COMSOL Script.

Swing is the standard GUI component library that is included in Java. Details about Java and Swing are available at http://java.sun.com.

The available components are:

TABLE 11-1: GUI COMPONENT FUNCTIONS

FUNCTION NAME	DESCRIPTION
`axes`	An axes object into which you plot arbitrary graphics (see "Axes" on page 193)
`button`	A button
`buttongroup`	A button group that synchronizes the selection state for option (radio) buttons and toggle buttons
`checkbox`	A check box
`combobox`	A combo box
`dialog`	A dialog box
`frame`	A user-interface window
`label`	Displays text and images
`listbox`	A list box
`menu`	A main menu or submenu added to a `dialog` or a `frame`
`menuitem`	A menu item that executes some action
`panel`	A panel into which you add components or other panels
`radiobutton`	An option (radio) button
`scrollpane`	Adds scroll bars around a component
`table`	A table
`tabbedpane`	Add panels to a `tabbedpane` to create different tabs.
`textarea`	A multiple-line area for entering text

TABLE 11-1: GUI COMPONENT FUNCTIONS

FUNCTION NAME	DESCRIPTION
`textfield`	A single-line area for entering text
`togglebutton`	A button that a user can select or deselect

To learn how to position components on a panel or frame, see "Panels and Layout Management" on page 262. For now, though, examine the various components that are at your disposal.

Labels and Image Icons

Labels display text or images (in the form of an image icon) at an arbitrary location on a frame or panel. To create a label, take the text you wish to display and supply it as an argument to the `label` function:

```
l=label('Enter name:');
```

An image icon is an image that is displayed on a label, button, check box, radio button, or toggle button. Create it by giving the name of a GIF, JPEG, or PNG file as an argument to the `imageicon` function:

```
im=imageicon('myimage.jpeg');
```

The `label` function also accepts the optional properties `Text` and `Image` with which you can directly specify either or both text and an image. For instance,

```
im=imageicon('myimage.jpeg');
l=label('text','Number of data points:','image',im);
```

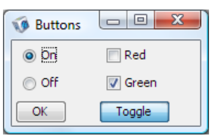
Buttons and Toggle Buttons

Buttons, check boxes, option (radio) buttons, and toggle buttons are all similar in that you create any of them by supplying a first argument that is the string to display with the component. As with labels, with these components you can also use the properties `Text` and `Image` to specify a text and/or an image to display on the component.

```
b=button('OK');
c=checkbox('Show grid');
r=radiobutton('On');
t=togglebutton('More>>');
```

Use `buttongroup` to synchronize the selection state for option buttons and toggle buttons so that selecting one option button in the group deselects all other option buttons. First create a button group, then add the components that should be synchronized to that group.

```
f=frame('Buttons');
p1=panel;
bg=buttongroup;
r1=radiobutton('On');
r2=radiobutton('Off');
bg.add(r1);
bg.add(r2);
p1.add(r1,1,1);
p1.add(r2,2,1);
c1=checkbox('Red');
c2=checkbox('Green');
p2=panel;
p2.add(c1,1,1);
p2.add(c2,2,1);
b1=button('OK');
t1=togglebutton('Toggle');
f.add(p1,1,1);
f.add(p2,1,2);
f.add(b1,2,1);
f.add(t1,2,2);

r1.setValue('on');
c2.setSelected(true);
t1.setSelected(true);

f.show;
```



Use the `getValue` method to determine if a button is selected by reading the resulting string `'on'` or `'off'`, and use the `setValue` method set the button's state to one of those strings. The methods `getSelected` and `setSelected` are similar but use `true` and `false` to indicate if the button is selected or not.

Text Fields and Text Areas

To enter a single line of text use a text field; in contrast, a text area holds multiple lines of text. You define the size of a text field as the number of characters it should hold, and define the size of a text area as the number of rows and columns of text it should hold.

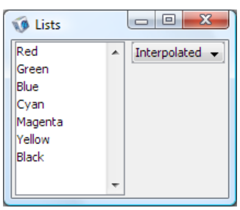
```
t1=textfield(12);
t2=textarea(5,20);
```

The optional property `Text` can set an initial string that should appear in the text field or text area. After you have created the component, read its text with the `getValue` method and modify the text with `setValue(text)`.

Combo Boxes and List Boxes

Combo boxes and list boxes display multiple descriptive strings and for each also store a corresponding value string. Specify the descriptive strings with the `Descr` property, and specify the corresponding values with the `Items` property. You can also use the `Items` property alone—in that case, the specified strings serve as both values and descriptions. Use both `Items` and `Descr` to let the strings in `Items` be the value of some property and the strings in `Descr` be the corresponding descriptions.

```
f=frame('Lists');
cb=combobox('items',{'interp','flat','none'}, ...
            'descr',{'Interpolated','Flat','None'});
lb=listbox('items',{'r','g','b','c','m'}, ...
           'descr',{'Red','Green','Blue','Cyan','Magenta'}, ...
           'size',[100 140]);
f.add(lb,1,1,2,1);
f.add(cb,1,2);
f.packColumn(2,2);
f.show;
```



Combo boxes and list boxes have almost the same methods for getting and setting the selection and the values to display. The list box has a few more methods, however, because it allows users to select multiple items.

TABLE 11-2: METHODS FOR MANIPULATING COMBO BOX AND LIST BOX OBJECTS.

METHOD	DESCRIPTION
`addListSelectionListener(name)`	Specifies that the function with the given `name` should run when the selection in the list box changes.
`getSelectedIndex`	Returns an index to the currently selected item in the combobox/listbox.

TABLE 11-2: METHODS FOR MANIPULATING COMBO BOX AND LIST BOX OBJECTS.

METHOD	DESCRIPTION
`getSelectedIndices`	Returns an array with indices to the selected items in the listbox.
`getValue`	Returns a string corresponding to the current selected item in the combobox/listbox.
`setItems(items)`	Sets the `items` to display in the combobox/listbox by passing a cell array of strings.
`setItems(items,descr)`	Sets the descriptions to display in the combobox/listbox and their corresponding values by passing two cell arrays of strings.
`setSelectedIndex(ind)`	Selects the item with the specified index in the combobox/listbox.
`setSelectedIndices(ind)`	Selects the items in the listbox corresponding to the indices in the vector `ind`.
`setValue(value)`	Selects the item with the specified value in the combobox/listbox.

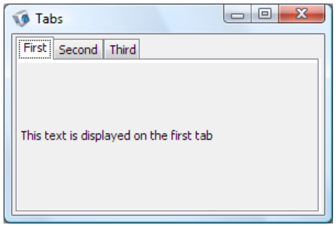
Tabbed Panes and Scroll Panes

COMSOL Script supports two types of the pane object. The first is the `tabbedpane`, which is the page that appears when you click one of multiple tabs across the top of a tabbed dialog box; the second is the `scrollpane`, which places scroll bars around a given component on a dialog box. The following example show how to use pane objects to create a tabbed dialog box.

To create a dialog box with multiple tabs, first call the `tabbedpane` function; then call the `addTab` method, and pass to it the text to display on the tab as well as the name of the page to display when the tab is clicked. Consider this example:

```
f=frame('Tabs','size',[300 200]);
f.setFill('both');
tp=tabbedpane;
p1=panel;
p2=panel;
p3=panel;
p1.add(label('This text is displayed on the first tab'));
p2.add(label('This text is displayed on the second tab'));
p3.add(label('This text is displayed on the third tab'));
tp.addTab('First',p1);
tp.addTab('Second',p2);
tp.addTab('Third',p3);
f.add(tp);
```

```
f.show;
```



While `tabbedpane` works with an entire page in a dialog box, `scrollpane` works with an individual component—it adds scroll bars around a component that is too large to display in an existing dialog box, panel or frame. When working with the `scrollpane` function, as the first argument pass the name of the component that gets the scroll bars, then use the `Size` property to specify the desired size of the scroll pane. Take care in this regard because if you do not specify a size, the scroll pane becomes very small. For an example of a scroll pane, see the result of the code snippet in the next section on tables.

Tables

Call the `table` function to create a table. Use the following property-value pairs to specify the number of rows and columns, column headers, and other appropriate parameters.

TABLE 11-3: VALID PROPERTY/VALUE PAIRS

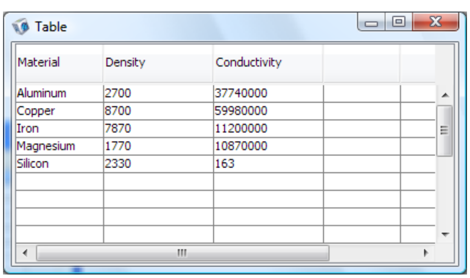
PROPERTY	VALUE	DEFAULT	DESCRIPTION
`autoadd`	On | Off	`off`	Determines whether rows should automatically be added to the end of the table as needed when the user enters values.
`cols`	integer	2	The number of columns in the table.
`editablecols`	integer array	all	Indices to the columns that should be editable.
`rows`	integer	`10`	The number of rows in the table.

TABLE 11-3: VALID PROPERTY/VALUE PAIRS

PROPERTY	VALUE	DEFAULT	DESCRIPTION
`titles`	cell array of strings		The headings for each column.
`width`	integer array		The desired width of the columns. It is either a scalar specifying the same width for all columns, or a vector of the same length as the number of columns. If you supply no value, each column is given a suitable width to fit its title.

As an exercise, create a table and display some material names in it. To the table add a scroll pane so the table's size can exceed that of the frame.

```
f=frame('Table');
t=table('rows',20,'cols',6,'width',[80 100 100 70 70 70], ...
        'titles',{'Material','Density','Conductivity','','',''});
scroll=scrollpane(t,'size',[400 200]);
f.add(scroll,1,1,'both');
f.show;

material{1,1}='Aluminum';
material{2,1}='Copper';
material{3,1}='Iron';
material{4,1}='Magnesium';
material{5,1}='Silicon';
settabledata(t,material);

data=[2700 3.774e7;8700 5.998e7;7870 1.12e7;
      1770 1.087e7;2330 163];
t.setValue(1:5,2:3,data);
```



Use the `getValue` and `setValue` methods on a table to read and modify numerical values. In addition, some corresponding methods take index vectors to rows and columns so that you can get and set values only in certain parts of a table. Table cells that are empty or that cannot be converted to numerical values return NaN in the corresponding positions in the output matrix.

The functions settabledata and gettabledata read and modify values from cell matrices of strings. This functionality is useful if you wish to format numerical values before setting them or to get and set non-numerical values such as strings.

Axes

The axes object is one of the most useful components in COMSOL Script user interfaces. It creates a set of 2D or 3D axes into which a COMSOL Script program then plots data in the form of lines or surfaces. See “2D Graphics” on page 205 and “3D Graphics” on page 215 for more information about 2D plots and 3D plots.

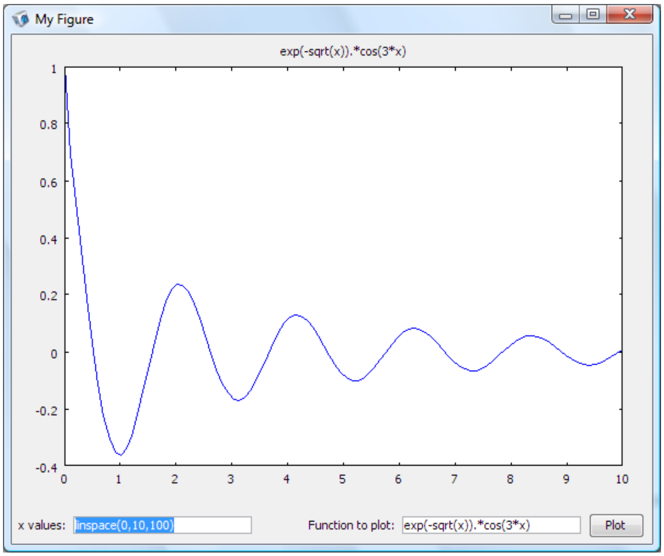
An axes object is the same component we use in the plotting windows built into COMSOL Script. You create an instance of this object using the axes function, then retrieve a handle to it with the getHandle method. This handle then serves as the Parent property for plotting functions such as plot, line, surf, and patch. The handle can also control properties such as axis limits, tick marks, labels, and titles.

As a default, an axes component has a very small size. Thus it is best to specify a fill style of 'both' and use a large weight for the grid cell when adding it to a panel (see the section in this chapter, “Panels and Layout Management” on page 262). These steps make sure that the axes component stretches to fill all extra space not needed by other components on a panel.

As an example, create a simple figure window by adding an axes object to a frame and then plot some data in it.

```
f1=frame('My Figure','size',[600 500]);
ax=axes;
p=panel;
p.add(label('x values:'),1,1);
p.add(textfield(20,'text','linspace(0,10,100)'),1,2);
p.addHSeparator(40,1,3);
p.add(label('Function to plot:'),1,4);
p.add(textfield(20,'text','exp(-sqrt(x)).*cos(3*x)'),1,5);
p.packRow(1,6);
p.add(button('Plot'),1,7);
f1.setWeight(1e6,1e6);
f1.add(ax,1,1,'both');
f1.resetWeight;
f1.add(p,2,1,'horizontal');
f1.show;
```

```
x=linspace(0,10,100);
y=exp(-sqrt(x)).*cos(3*x);
parent=ax.getHandle;
plot(x,y,'parent',parent);
set(parent,'title','exp(-sqrt(x)).*cos(3*x)');
```



The code that plots the function would normally be in an `actionListener` method for the **Plot** button. It could use `eval` calls to get *x* and *y* vectors from the text strings in the text fields and then plot that data in the axes object.

Panels and Layout Management

A panel is an area within a larger object such as a frame or dialog box, often distinguished from the rest of a dialog box with a ruled line, in which you add components or even other panels. To create a panel in COMSOL Script, you work with a layout manager that is based on the Java GridBagLayout class. This means that you divide a panel into a grid and then add components to different cells in the grid. Each component has a preferred size that defines the size of each cell. The cells, in turn, define the size of the entire panel. Thus there is no need to manually account for different font sizes on different computing platforms and so on when laying out components in a dialog box.

Adding Components

When adding a GUI component to a cell in the panel grid, you can specify how it should be aligned within the cell and if it should fill out the cell in the horizontal and vertical directions. With various `add` methods you specify the cell to which you want to add a component. You can also specify that a component should span several cells. There is no need to specify the grid size; the software determines it automatically from the maximum row and column numbers.

TABLE 11-4: METHODS FOR ADDING COMPONENTS TO A PANEL.

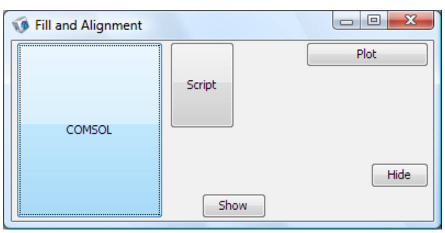
METHOD	**DESCRIPTION**
`add(comp,row,col)`	Adds a component to the cell at the given row and column.
`add(comp,row,col,nrows,ncols)`	Adds a component to the cell at the given row and column. The component spans the specified number of rows and columns.

TABLE 11-4: METHODS FOR ADDING COMPONENTS TO A PANEL.

METHOD	DESCRIPTION
`add(comp,row,col,fill)`	Adds a component and specifies how it should fill the cell to which it is assigned. `Fill` is a string that tells the component to stretch to fill the cell in certain directions. It can have one of the values `'both'`, `'horizontal'`, or `'vertical'`.
`add(comp,row,col,nrwos,ncols,fill)`	The same as `add(comp,row,col,fill)` but also allows you to specify the number of rows and columns the component should span.

As an exercise, create a frame with buttons that have different fill styles and alignments:

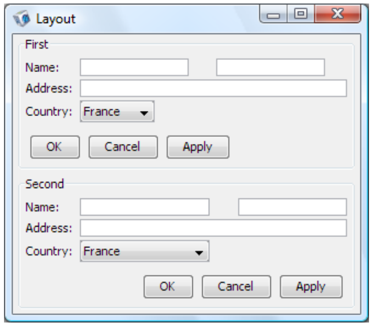
```
f=frame('Fill and Alignment','size',[400 200]);
f.add(button('COMSOL'),1,1,2,1,'both');
f.add(button('Script'),1,2,'vertical');
f.add(button('Plot'),1,3,'horizontal');
f.setAlignment('south');
f.add(button('Show'),2,2);
f.setAlignment('east');
f.add(button('Hide'),2,3);
f.show;
```



This code snippet divides the frame into a 2-by-3 grid. The **COMSOL** button in the first column is specified to span two rows and to fill the space in the cells in which it lies. The **Script** button fills its cell only vertically, and the width needed to display the **Script** string determines this component's width. The **Plot** button fills its cell horizontally, and its height is determined by that needed to display the text on it. The **Show** and **Hide** buttons have sizes determined by the strings they display, and they are aligned to the south and east in their cells, respectively.

The next example shows how to create a frame and on it place two panels with labels, text fields, buttons, and combo boxes.

```
f=frame('Layout');
p1=panel;
p1.add(label('Name:'),1,1);
p1.add(textfield(12),1,2);
p1.add(textfield(12),1,3);
p1.add(label('Address:'),2,1);
p1.add(textfield(30),2,2,1,2);
p1.add(label('Country:'),3,1);
countries={'France','Germany','Sweden','USA'};
p1.add(combobox('items',countries),3,2);
bp1=panel;
bp1.add(button('OK'),1,1);
bp1.add(button('Cancel'),1,2);
bp1.add(button('Apply'),1,3);
p1.addVSeparator(4,4,1);
p1.add(bp1,5,1,1,3);
p1.addBorder('First');

p2=panel;
p2.add(label('Name:'),1,1);
p2.add(textfield(12),1,2,'horizontal');
p2.setAlignment('east');
p2.add(textfield(12),1,3);
p2.setAlignment('west');
p2.add(label('Address:'),2,1);
p2.add(textfield(30),2,2,1,2);
p2.add(label('Country:'),3,1);
p2.add(combobox('items',countries),3,2,'horizontal');
bp2=panel;
bp2.add(button('OK'),1,1);
bp2.add(button('Cancel'),1,2);
bp2.add(button('Apply'),1,3);
p2.addVSeparator(4,4,1);
p2.setAlignment('east');
p2.add(bp2,5,1,1,3);
p2.addBorder('Second');

f.add(p1,1,1);
f.add(p2,2,1);

f.show;
```



The panels are similar and are each divided into a 5-by-3 grid. The first column holds the labels. The **Address** text field spans both Columns 2 and 3. Because it is more than twice as wide as the **Name** text fields, extra gray space is available in Columns 2 and 3 after the **Name** text fields. In the upper panel, the width of the **Country** combo box is determined by the length of the longest pull-down description in that combo box, whereas the lower panel specifies that the **Country** combo box and the **Name** text field should fill their cells horizontally—a requirement that results in a nice alignment between them. A vertical separator is added above the panels that hold the buttons, and in the lower panel its alignment is set to `'east'`.

Distributing Extra Space

If a panel is larger than required by its components, the extra space is distributed between the grid cells according to their relative weights. By default, each cell has a weight of 1 in both the *x* and *y* directions. You can set higher weight values for certain cells if you want them to get more of the extra space.

You can also use the `packRow` and `packColumn` methods to specify that all components in a row or a column should be moved away as much as possible from a certain cell.

TABLE 11-5: METHODS FOR CONTROLLING HOW EXTRA SPACE IS DISTRIBUTED.

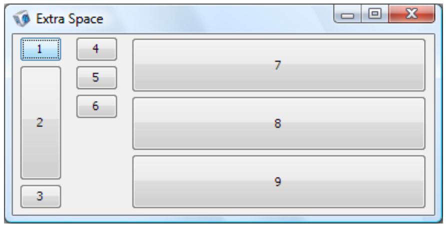
METHOD	DESCRIPTION
`pack`	Packs (shifts) components on the panel towards the upper left corner. Can, for example, be used before adding a panel to a `tabbedpane` to make sure the components on each tab stretch to fill the tab.
`packColumn(row,col)`	Packs (shifts) components in a column away from the specified row and column.

TABLE 11-5: METHODS FOR CONTROLLING HOW EXTRA SPACE IS DISTRIBUTED.

METHOD	DESCRIPTION
`packRow(row,col)`	Packs (shifts) components in a row away from the specified row and column.
`resetWeight`	Resets the weights to their default values, which is 1 in both the x and y directions.
`setWeight(x,y)`	Sets the weight in the x and y directions for any components added after you make this call. The components' relative values of the weights determine how extra space within the panel should be distributed if the panel is larger than needed by the preferred size of its components.
`setWeightX(x)`	Set the weight only in the x direction.
`setWeightY(y)`	Set the weight only in the y direction.

Consider this example:

```
f=frame('Extra Space','size',[400 200]);
f.setFill('both');
p1=panel;
p1.setFill('both');
p1.add(button('1'),1,1);
p1.setWeightY(1e6);
p1.add(button('2'),2,1);
p1.resetWeight;
p1.add(button('3'),3,1);
p2=panel;
p2.setFill('both');
p2.add(button('4'),1,1);
p2.add(button('5'),2,1);
p2.add(button('6'),3,1);
p2.packColumn(4,1);
p3=panel;
p3.setFill('both');
p3.add(button('7'),1,1);
p3.add(button('8'),2,1);
p3.add(button('9'),3,1);
f.add(p1,1,1);
f.add(p2,1,2);
f.setWeightX(1e6);
f.add(p3,1,3);
f.resetWeight;
f.show;
```



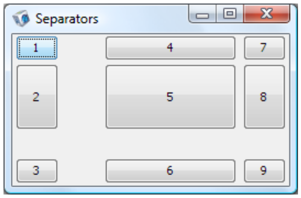
This example adds panels in three columns. In the first column, you set a large weight in the *y* direction before adding the **2** button. In this way that cell gets all the extra vertical space, and because you add the button with a fill style of `'both'`, it stretches to fill the cell. For the panel in the second column, a `packColumn` call moves the components away from Row 4 on that panel. Before the panel in the third column is added, you set the weight in the *x* direction to a large value for that cell. In this way the third column gets all the extra horizontal space.

Adding Empty Space

To give a cell a certain width or height without adding a component to it, use the `addHSeparator` and `addVSeparator` methods, which create extra space between components. You can also use these methods in cells where you have already added another component to force that cell to reach a certain minimum width or height.

```
f=frame('Separators');
f.setFill('both');
f.add(button('1'),1,1);
f.add(button('2'),2,1);
f.addVSeparator(20,3,1);
f.add(button('3'),4,1);
f.addHSeparator(30,1,2);
f.add(button('4'),1,3);
f.add(button('5'),2,3);
f.addVSeparator(60,2,3);
f.addHSeparator(120,2,3);
f.add(button('6'),4,3);
f.add(button('7'),1,4);
f.add(button('8'),2,4);
f.add(button('9'),4,4);
```

```
f.show;
```



Here the buttons are laid out on a 4-by-4 grid where there are no buttons on Row 3. This code snippet adds a horizontal separator with a width of 30 pixels in Column 2 and adds a vertical separator with a height of 20 pixels in Row 3. It also adds a horizontal separator and a vertical separator in the same cell as the **5** button to make that cell larger than required by the button itself.

Accessing Components

To specify a tag when creating a component, use the `Tag` property. With it you can later access that component by calling the `get` method on a `panel`, `dialog`, or `frame`. For instance, examine this code:

```
f=frame('Tags');
f.add(label('Width:'),1,1);
f.add(textfield(10,'tag','width'),1,2);
f.add(label('Height:'),2,1);
f.add(textfield(10,'tag','height'),2,2);
f.show;
t1=f.get('width');
t2=f.get('height');
t1.setValue('400');
t2.setValue('300');
```

The two text fields are created without assigning them to a specific variable. Instead they are given a tag when created. This tag is then used with the `get` method on frame to access the text fields and set the text in them.

Event Handling

With GUI components selected and placed on a panel, it is now time to assign actions to them. You can specify that GUI events such as the click of a button, mouse movements, or the change of the selection in a list box should run a certain M-file. Simply specify the name of the function to run as an argument to the appropriate method. For instance, to specify that the function `okaction` should run when the user clicks the **OK** button, write these two lines of code:

```
b=button('OK');
b.addActionListener('okaction');
```

The following table lists the methods that add event handlers to certain types of components.

TABLE 11-6: METHODS FOR ADDING EVENT HANDLERS

METHOD	COMPONENTS	DESCRIPTION
`addActionListener`	`button`, `checkbox`, `combobox`, `radiobutton`, `togglebutton`	Run a function when a button is clicked or the selection of the component changes.
`addActionListenerThread`	`button`	Run a function in a separate thread when a button is clicked. This can be used for operations that execute for a long time and need to update graphics while running.
`addFocusListener`	all	Run a function when a component gains or loses focus.
`addListSelectionListener`	`listbox`	Run a function when the selection in a listbox changes.
`addMouseListener`	all	Run a function when the cursor is moved or clicked over a component.

When calling an event-handling function, you supply one argument that is a structure containing information about the event. The event structure always has the following fields:

- `parent`—The dialog box or frame in which the source component for the event lies

- `source`—The component that is the source of the event
- `type`—A string specifying the type of event

Generally you use the `parent` field to identify the frame or dialog box in which the component causing the event is situated. Then, to access other components in the frame and their values, call the `get` method along with tags to the components.

The third field in the event structure, the `type` field, can have a number of values as indicated in the following table:

TABLE 11-7: VALUE OF THE TYPE FIELD FOR DIFFERENT TYPES OF EVENTS.

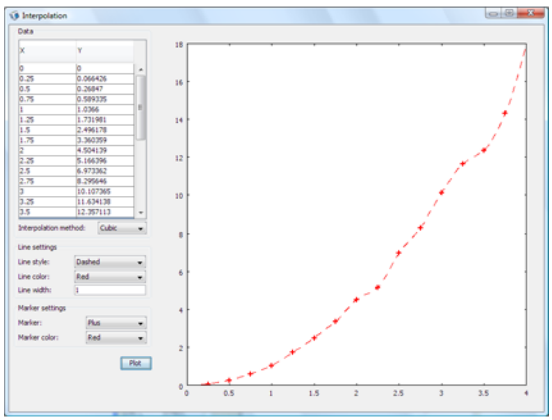
VALUES OF TYPE FIELD	TYPE OF EVENT
`'action'`	An action event has occurred.
`'focusgained'`	A component has gained focus.
`'focuslost'`	A component has lost focus.
`'list'`	The selection in a list box has changed.
`'mouseentered'`	The mouse was moved on top of a component.
`'mouseexited'`	The mouse was moved away from a component.
`'mouseclicked'`	A mouse button was clicked on top of a component.
`'mousepressed'`	A mouse button was pressed on top of a component.
`'mousereleased'`	A mouse button was released on top of a component.

In this list of values, note that for mouse events used on an axes component, the event structure also contains the fields `x` and `y` that give the coordinates where the mouse event occurred.

Example User Interface

This example shows how to create a simple GUI. It includes a table in which you can enter coordinates for data points. You then specify an interpolation method to interpolate between this data. It creates a plot with the data points as markers and the interpolated values as a line with a given color, line style, and line width.

Note: The files for this example are available in the `demos` directory in the installation.



Start by writing the code that draws the GUI and specifies which functions to use as event listeners:

```
f=frame('Interpolation');
p1=panel;

dp=panel;
dp.setFill('horizontal');
t1=table('rows',30,'cols',2,'titles',{'X','Y'},
'width',[90 90],'tag','data');
sc=scrollpane(t1,'size',[200 280],'horizontal','never');

dp.add(sc,1,1,1,2);
dp.add(label('Interpolation method:'),2,1);
dp.add(combobox('items',{'nearest','linear','cubic'}, ...
                'descr',{'Nearest','Linear','Cubic'}, ...
                'tag','method'),2,2);
```

```
dp.addBorder('Data');

lp=panel;
lp.setFill('horizontal');
lp.add(label('Line style:'),1,1);
lp.add(combobox('items',{'-','--',':','-.'}, ...
               'descr',{'Solid','Dashed','Dotted','Dash-dot'}, ...
                'tag','linestyle'),1,2);

lp.add(label('Line color:'),2,1);
lp.add(combobox('items',{'r','g','b'}, ...
                'descr',{'Red','Green','Blue'}, ...
                'tag','linecolor'),2,2);

lp.add(label('Line width:'),3,1);
lp.add(textfield(10,'text','1','tag','linewidth'),3,2);
lp.addBorder('Line settings');

mp=panel;
mp.setFill('horizontal');
mp.add(label('Marker:'),5,1);
mp.add(combobox('items',{'+','*','o','s'}, ...
                'descr',{'Plus','Star','Circle','Square'}, ...
                'tag','marker'),5,2);

mp.add(label('Marker color:'),6,1);
mp.add(combobox('items',{'r','g','b'}, ...
                'descr',{'Red','Green','Blue'}, ...
                'tag','markercolor'),6,2);

mp.addBorder('Marker settings');

p1.add(dp,1,1,'horizontal');
p1.add(lp,2,1,'horizontal');
p1.add(mp,3,1,'horizontal');
p1.setAlignment('east');
p1.addVSeparator(4,4,1);
p1.add(button('Plot','tag','plot'),5,1);

f.add(p1,1,1);
f.setWeight(1e6,1e6);
f.add(axes('tag','axes','size',[600 600]),1,2,'both');
f.resetWeight;

f.get('plot').addActionListener('interpaction');
f.get('axes').addMouseListener('interpmouse');
f.show;
```

Note the following:

- The combo boxes use descriptive values and corresponding property names that they pass to the `interp1` and `plot` functions.
- On the top level, the frame is divided into two panels: one to the left, and the axes object to the right.
- You add the panels to the left panel with the `'horizontal'` fill style to align the borders. You also specify the panels as having `'horizontal'` fill for the components you add to them. This means that the combo boxes and text fields line up even if the maximum lengths of the strings in the combo boxes differ.
- You add the table to a scroll pane with only a vertical scroll bar; specify the width of the columns explicitly to fit into the scrolling viewport.
- You set a high weight before adding the axes object, which has a fill style of `'both'`.
- You assign tags to the components so it is possible to access them in the event listeners.
- You specify `interpaction` as the function to run when the user clicks the **Plot** button.
- You specify `interpmouse` as the function to run when the user moves the mouse and clicks over the axes object.

The function `interpaction` uses the `parent` field in the event structure to get the frame, and it then calls the `get` method on the frame to get the components and their values. It calls the `getHandle` method on the axes object to obtain a handle to that object; that handle then serves as the `Parent` property to the `Plot` function.

```
function interpaction(event)

% Get parent frame of event source
frame=event.parent;

% Get value from components using their tags
method=frame.get('method').getValue;
linestyle=frame.get('linestyle').getValue;
linecolor=frame.get('linecolor').getValue;
linewidth=str2num(frame.get('linewidth').getValue);
marker=frame.get('marker').getValue;
markercolor=frame.get('markercolor').getValue;

% Get data from table and interpolate
data=frame.get('data').getValue;
data=data(~isnan(data(:,1)),:);  % Remove NaNs from empty rows.
```

```
xi=data(:,1)';
yi=data(:,2)';
[xi,ind]=sort(xi);
yi=yi(ind);
x=linspace(min(xi),max(xi),100);
y=interp1(xi,yi,x,method);

% Get handle to axes and clear before plotting into it
parent=frame.get('axes').getHandle;
cla(parent);
h=plot(x,y,'color',linecolor,'linewidth',linewidth, ...
       'linestyle',linestyle,'parent',parent);
set(h,'hold','on');
plot(xi,yi,'linestyle','none','marker',marker, ...
     'color',markercolor,'parent',parent);
```

The example also adds a mouse listener to the axes object; it lets the user click positions for the data points instead of entering them in the table. The function finds the first empty row in the data table and enters the clicked coordinate in that row. Note also that it checks that the event type is `'mouseclicked'`, because the function also runs when the user moves, presses, or releases the mouse.

```
function interpmouse(event)
if (strcmp(event.type,'mouseclicked'))
  x=event.x;
  y=event.y;
  frame=event.parent;
  table=frame.get('data');
  data=table.getValue;
  first=find(isnan(data(:,1)));
  if ~isempty(first)
    table.setValue(first(1),1:2,[x y]);
    parent=event.source.getHandle;
    line(x,y,'linestyle','none','marker','*','parent',parent);
  end
end
```

The `x` and `y` fields in the event structure get the coordinate in the axes object for the mouse click. The routine then draws a marker at that position in the axes object.

12

User-Defined Classes

Using COMSOL Script it is possible to define new data types, classes, and to create objects that are instances of these classes. A class is an aggregate of fields and methods.

The class model used is inspired by Java's class system.

Introductory Example: Rectangle

The file `Rectangle.csl` defines the class `Rectangle`:

```
class Rectangle
%A class for rectangles.

public x0 x1  % x0<x1
public y0 y1  % y0<y1

function Rectangle(varargin)
%RECTANGLE(X0, Y0, X1, Y1) creates a rectangle with vertices
%in (x0,y0) and (x1,y1).
switch nargin
case 4
  in = varargin;
  [x0 x1] = deal(min(in{1}, in{3}), max(in{1}, in{3}));
  [y0 y1] = deal(min(in{2}, in{4}), max(in{2}, in{4}));
case 1
  r = varargin{1};
  [x0 x1] = [r.x0 r.x1];
  [y0 y1] = [r.y0 r.y1];
otherwise
  error('Usage: Rectangle(x0, y0, x1, y1)');
end

public function out = area
%AREA returns the area of the rectangle.
out = (x1-x0)*(y1-y0);

public static function out = overlap(r1, r2)
%OVERLAP(R1, R2) returns the area of the overlap between the
%two rectangles R1 and R2.
out = max(min(r1.x1, r2.x1)-max(r1.x0, r2.x0), 0)*...
      max(min(r1.y1, r2.y1)-max(r1.y0, r2.y0), 0);
```

This defines a Rectangle *class* with the four *fields* `x0`, `y0`, `x1`, and `y1`, a *constructor*, and the two *methods* `area` and `overlap`. To create a `Rectangle` *object*, use the same syntax as for a function call:

```
r = Rectangle(1, 3, 2, 9)
r =
Rectangle object
  x0:  [1]
  x1:  [2]
  y0:  [3]
  y1:  [9]
```

```
r.area

ans =
6
```

The call `Rectangle(1, 3, 2, 9)` returns a rectangle with vertices in (1,3) and (2,9). The variable `r` is of the type `Rectangle`; it is an *instance* of the `Rectangle` class. There can only be one definition of a class, but there can be many instances of it. By default, the fields of an object are displayed in the same way as if it were a structure.

The call `r.area` invokes the `area` method of the object r. The syntax for invoking a method without arguments is the same as for accessing a field of a structure.

The class contains three methods: the constructor `Rectangle`, the accessor `area`, and the static function `overlap`. The method declarations look like declarations of local functions in a function M-file, but unlike local functions, they can be accessed outside the class.

`overlap` is an example of a static method: it is invoked when `overlap` is invoked with arguments where at least one is a `Rectangle`:

```
r = Rectangle(1, 5, 6, 8);
s = Rectangle(3, 2, 5, 7);
overlap(r, s)

ans =
4
```

The overlap method is not visible if none of the arguments is a `Rectangle`: for example, `overlap(1,2)` gives an error.

The Structure of the Class File

A class is defined by a `.csl`-file on the path that has the following contents:

- Class header
- Precedence declarations
- Field declarations
- Method declarations

Only the class header is mandatory. Below you find brief descriptions of the contents of each part of the file. The next section contains a more detailed description.

Class Header

The class header declares the name of the class, its copy semantics, and its superclass, if any. The most basic form is a value class with no superclass:

```
class Rectangle
```

The name of the class, here `Rectangle`, must coincide with the filename, `here Rectangle.csl`. That the class is a value class means that a copy is made whenever an instance of the class is used as argument to a function, or is used in an assignment. All other data types except for Java objects have value semantics. The opposite is a reference class, declared with the `reference` modifier:

```
reference class Rectangle
```

Reference semantics means that a true copy of the object is never made, only references to the same object. This is the semantics that Java objects have.

The superclass of a class is declared using the `extends` keyword:

```
class Rectangle extends Shape
```

The superclass must be defined by a `.csl`-file on the path.

Precedence Declarations

This optional section contains the classes that have higher or lower precedence than the class being declared. It contains zero or more lines of the form

```
superiorto <classname>
```

or

```
inferiorto <classname>
```

Field Declarations

This optional section contains zero or more field declarations of the forms

```
<access modifiers> <name> = <expression
```

or

```
<access modifiers> <name1> [<name2> ...]
```

The fields declared are like the fields of a structure, with the difference that a class always contains the same fields. The access modifiers control where the field is visible. You must specify one of `public`, `protected`, and `private`, and you can optionally use `static` and `transient`.

The first syntax allows for initial values to be supplied:

```
public version = 1;
```

This declares the field `version` with the default value 1. This means that the `version` field is assigned to 1 when an instance of the class is created. Fields without initial values are assigned to [ ].

Method Declarations

This optional section contains zero or more method declarations of the form

```
[<access modifiers>] <function declaration>
```

You can specify one of the access modifiers `public`, `protected`, and `private`, as well as `static`. The method is considered `public` unless you specify `protected` or `private`.

Except for the optional access modifiers, the syntax of a function declaration is the same as for a local function declaration in an M-file.

Access Modifiers

You can assign access modifiers to the fields and methods of a class. There are three types of modifiers:

- Visibility modifiers: `public`, `protected`, and `private`. They specify where the member is visible: `public` means that the member is visible everywhere, `protected` that it is visible in the class and classes inheriting from it, and `private` that it is visible only in the class where it was declared.
- Static modifier: `static`. A static member is one that is associated with the class, not with an instance of the class. The opposite is an instance member (the default).
- Transient modifier: `transient`. COMSOL Script does not save a field marked as transient when you run the command `save`. You can use this to avoid saving temporary data that is easy to recompute.

Each member field must have a visibility modifier specified. For methods this is not necessary; if omitted, the default visibility is `public`.

A class where all fields are `public` behaves a lot like a structure: It is possible to read and assign values to all fields using the `var.field` syntax. Restricting the visibility can be useful for hiding class internals that are inconsequential for the user of the class.

Member Fields

A class can be viewed as a structure with a predefined set of fields, the instance (nonstatic) fields. Static fields have semantics similar to `global` or `persistent` variables.

Instance Fields

Fields not declared as static are instance fields: They belong to an instance of the class, like fields in a structure. The fields of the `Rectangle` class are examples of instance fields:

```
class Rectangle

public x0 x1  % x0<x1
public y0 y1  % y0<y1
```

Outside of the class, you can use these fields like the fields of a structure:

```
r = Rectangle(2, 3, 6, 7);
r.x1

ans =
6
```

This is only possible for `public` fields; you cannot access `protected` and `private` fields this way. It is also possible to modify public fields like fields of a structure:

```
r = Rectangle(2, 3, 6, 7);
r.y1 = 10

r =
Rectangle object
  x0:  [2]
  x1:  [6]
  y0:  [3]
  y1:  [10]
```

When a nonstatic method of a class is run, you can access its instance fields like local variables:

```
public function out = area
%AREA returns the area of the rectangle.
out = (x1-x0)*(y1-y0);
```

The values of x0, x1, y0, and y1 are taken from the instance fields, and any assignments to them would result in the instance fields being changed.

Static Fields

Fields declared static belong to the class, not any instance of it. As an example, consider the class MathConst:

```
class MathConst

% The golden ratio
public static golden = (sqrt(5)+1)/2;
% Perfect numbers less than 10^11
public static perfect = [6 28 496 8128 33550336 8589869056];
```

The two fields golden and perfect are static. Their values are set using initializers, (see the next section for an explanation of this). To access them, access the class like a structure:

```
MathConst.golden

ans =
1.6180

MathConst.perfect(2:3)

ans =
28  496
```

It is also possible to change their values:

```
MathConst.golden = -17
```

Initialization of Fields

The MathConst class illustrates initialization:

```
public static golden = (sqrt(5)+1)/2;
```

The above means that when the class is loaded, the static field golden is assigned the expression in the right-hand side. For instance fields, initializers provide initial values when the object is created. As an example, consider the point class:

```
class Point

public x = 2;
public y = 5;
```

The default placement of the point is at coordinates (2, 5):

```
p = Point

p =
Point object
  x:  [2]
  y:  [5]
```

If a field is not given an initializer, it is assigned to [ ]. More specifically: When an object is created, all instance fields are initialized to [ ]. Then initializers, if any, are evaluated in the order the fields are declared. This means that

```
public x;
public y = {x 5};
public z = length(y);
```

is equivalent to

```
public x = [];
public y = {[] 5};
public z = 2;
```

Member Methods

Constructor

The constructor is a function with the same name as the class and is called when an instance of the class is created. The purpose of the constructor is to set up a valid object, possibly using input arguments. It cannot be static and must not return anything. As an example, consider the constructor of the `Rectangle` class:

```
function Rectangle(varargin)
%RECTANGLE(X0, Y0, X1, X1) creates a rectangle with vertices
%in (x0,y0) and (x1,y1).
switch nargin
case 4
  in = varargin;
  [x0 x1] = deal(min(in{1}, in{3}), max(in{1}, in{3}));
  [y0 y1] = deal(min(in{2}, in{4}), max(in{2}, in{4}));
...
```

The constructor is called whenever a Rectangle is created: When

```
r = Rectangle(1, 3, 2, 9)
```

is executed, an empty `Rectangle` object is created. At this point, the four instance fields `x0`, `x1`, `y0`, and `y1` have been initialized with [ ]. Then the constructor is invoked for the new object. The constructor is an instance method and can therefore modify the instance fields: The assignments to, for example, `x0` modify the field `x0` of the instance. The methods of a class differ from local functions in this respect: A local function `Rectangle` with the above contents would assign values to `x0` in its own workspace, but when leaving the function, these values would be lost.

A class does not need a constructor: If there is no constructor, the fields are assigned default values if provided, otherwise [ ].

Instance Methods

An instance method is a method that is not declared static. It operates on an instance of a class and has access to the instance fields. The `area` method in the `Rectangle` class is an example of an instance method:

```
public function out = area
%AREA returns the area of the rectangle.
out = (x1-x0)*(y1-y0);
```

An instance method can read or write the instance fields like local variables, just like the constructor. `area` uses the coordinates of the rectangle to compute its area.

Static Methods

A static method has access to the static fields of the class but not to any instance fields as it cannot be run for any instance of the class. The `overlap` method of the `Rectangle` class is static:

```
public static function out = overlap(r1, r2)
%OVERLAP(R1, R2) returns the area of the overlap between the
%two rectangles R1 and R2.
out = max(min(r1.x1, r2.x1)-max(r1.x0, r2.x0), 0)*...
      max(min(r1.y1, r2.y1)-max(r1.y0, r2.y0), 0);
```

There are two way to invoke a static method:

- Calling it like a function with one or more objects as arguments:

```
r = Rectangle(1, 5, 6, 8);
s = Rectangle(3, 2, 5, 7);
overlap(r, s)

ans =
4
```

- Specifying it explicitly:

```
Rectangle.overlap(r, s)

ans =
4
```

The first syntax works because at least one of the arguments to `overlap` is an object, here of the class `Rectangle`, and the class `Rectangle` has a visible static method called `overlap`.

Inheritance

Sometimes a class is a specialization or extension of another class. You can specify this relation in the class header using the `extends` keyword:

```
class Rectangle extends Shape
```

A rectangle is a specialization of the shape concept, and it therefore makes sense to let the class `Rectangle` inherit from the class `Shape`, which then becomes the *superclass* of `Rectangle`, the *derived class*. The members and methods of the superclass are visible in the derived class. Suppose that the Shape and Rectangle classes contain the following fields:

```
class Shape
public name
...

class Rectangle extends Shape
public x0 x1
public y0 y1
...
```

The `Rectangle` class contains five fields: The four fields declared in `Rectangle`, and the field declared in the superclass, `Shape`. For a user of the class, there is no distinction between the two types of fields:

```
r = Rectangle(1, 3, 2, 9);
r.name = 'COMSOL';
r.name

ans =
  COMSOL
```

Reference and Value Classes

Differences

When you make an assignment `a = b`, COMSOL Script normally assigns a copy of `b` to `a`. This is the case for all data types except for Java objects and instances of reference classes: For these data types, only a reference is copied. Consider the class `RefClass` defined as follows:

```
reference class RefClass
public x = 5;
```

The software only copies a reference when an assignment is made:

```
r = RefClass;
s = r;
r.x = 10;
s.x

ans =
10
```

Consider on the other hand the class `ValueClass`, defined as follows:

```
class ValueClass
public x = 5;
```

For a value class, the assignment makes a true copy:

```
v = ValueClass;
s = v;
v.x = 10;
s.x

ans =
5
```

The same rules apply when passing arguments to functions: For an instance of a value class, a copy of the value is passed; for an instance of a reference class, a reference to the value is passed.

Choosing Class Type

The main factor determining the class type is how you view the class:

- If you think of the class as a structure augmented with methods, declare it as a value class.
- If you think of the class as a lightweight Java class, declare it as a reference class.

A reference class cannot inherit from a value class, or vice versa. Thus the choice of class type for a base class affects the entire class hierarchy.

If you define several different classes, try to use the same class type for all of them. Mixing class types can easily lead to bugs as the difference in semantics is subtle.

Built-In Object Functions

Functions That You Can Only Use for Objects

THE THIS FUNCTION

Inside an instance method, `this` returns the instance for which the method is run.

THE SUPER FUNCTION

When a constructor is run in a derived class, you can use `super` to run the constructor of the superclass. Suppose `Rectangle` inherits from `Shape`:

```
class Rectangle extends Shape

...

function Rectangle(varargin)
super(varargin);
...
```

The call `super(varargin)` runs the constructor of the `Shape` class.

It is also possible to use `super` as a structure to access or modify visible fields or methods from the superclass: `super.a = 17;` sets the field `a` in the superclass to 17. This can be useful if a member in the superclass has been shadowed by a member with the same name in the derived class.

`super` can be used to resolve name conflicts between methods in a class and .M-file functions: Suppose that the class `Potential` contains a method called `gradient`. Then the .M-file function `gradient` is shadowed by the method `gradient`: Calling `gradient` from the `Potential` class invokes the method `gradient`, not the function `gradient`. By instead calling `super.gradient`, the function `gradient` is called unless there is a method `gradient` in a superclass of `Potential`. That is, the .M-file functions are implicit members of a common base class of all user-defined classes.

THE CLONE FUNCTION

The `clone` function creates a true copy of an object. For a value object, this has no effect, but for a reference object it is necessary in order to copy the contents, not only a reference to the object. The class `RefClass` was above defined as

```
reference class RefClass
public x = 5;
```

Use the `clone` function to create a true copy of a `RefClass` object:

```
r = RefClass;
s = clone(r);
r.x = 10;
s.x

ans =
5
```

Without `clone`, only a reference would be copied.

Functions with Special Semantics for Objects

CLEAR CLASSES

`clear classes` removes all variables, just like `clear` does, but it also tries to remove all class definitions. This is necessary if the definition of a class changes: Unless you perform `clear classes`, COMSOL Script uses the old class definition even if the class file on disk changes.

`clear classes` can only remove the definitions of classes of which there are no instances. This means that sometimes not all classes are removed: If `clear classes` is called from a function and there still are instances of some class in the root workspace, then that class cannot be cleared. The same thing can happen if you store objects in global or persistent variables.

HELP FOR A CLASS

`help` for a class displays the comment block following the class header as well as the comment blocks following each public nonstatic method:

```
help Rectangle

A class for rectangles.

RECTANGLE(X0, Y0, X1, Y1) creates a rectangle with vertices
in (x0,y0) and (x1,y1).

AREA returns the area of the rectangle.

OVERLAP(R1, R2) returns the area of the overlap between the
two rectangles R1 and R2.
```

You can also retrieve the help text for a method:

```
help Rectangle.overlap
```

```
OVERLAP(R1, R2) returns the area of the overlap between the
two rectangles R1 and R2.
```

If you have a variable which is an instance of a class you can get help for its class or a method of the class using a variation of the above syntax:

```
r = Rectangle(2, 3, 6, 7);
help r

A class for rectangles.

RECTANGLE(X0, Y0, X1, Y1) creates a rectangle with vertices
...

help r.overlap

OVERLAP(R1, R2) returns the area of the overlap between the
two rectangles R1 and R2.
```

This notation is more compact and has the advantage that you do not even have to know class `r` belongs to, something which is useful if you are working with a hierarchy of classes.

THE FIELDNAMES FUNCTION

For an object, `fieldnames` returns the names of the instance fields that are visible from the workspace where `fieldnames` is called:

```
r = Rectangle(2, 3, 6, 7);
fieldnames(r)

ans =
  'x0'
  'x1'
  'y0'
  'y1'
```

All the fields of the `Rectangle` class are `public`, and `fieldnames` therefore returns them. `private` and `protected` members are only returned by `fieldnames` when invoked from a method of the class.

THE METHODS FUNCTION

`methods` displays the methods declared by a class:

```
methods('Rectangle')

class Rectangle
  public function Rectangle
  public function area
```

By default, `methods` only displays public nonstatic methods. It is possible to select methods to display based on access modifiers:

```
methods('Rectangle', 'static')

class Rectangle
  public static function overlap
```

The second argument to `methods` is a string or cell array of strings that lists all access modifiers to include.

When requesting output, `methods` returns a cell array:

```
c = methods('Rectangle', 'static')

c =
{'overlap'}
```

THE STRUCT FUNCTION

When invoked for an object, `struct` returns a structure containing the fields of the object that are visible in the workspace where `struct` is called:

```
r = Rectangle(1, 3, 2, 9);
struct(r)

ans =
  x0:  [1]
  x1:  [2]
  y0:  [3]
  y1:  [9]
```

Overloading

Overloading Operators

You can overload all unary and binary arithmetic, relational, and logical operators, as well as the concatenation operators [,] and [;]. To overload an operator, create a `public static` function that defines the semantics of the overloaded function. As an example, consider a class `Rational` that represents rational numbers:

```
class Rational
%RATIONAL is a class for exact representation of a rational number
%as a ratio between integers.

private a  % Numerator
private b  % Denominator, always > 0

...

static function out = plus(r1, r2)
%PLUS Sum.
%   OUT = PLUS(R1, R2) returns the sum of R1 and R2.
r1 = Rational(r1);
r2 = Rational(r2);
out = Rational(r1.a*r2.b+r1.b*r2.a, r1.b*r2.b);

...
```

The static function `plus` provides an overload for the + operator:

```
Rational(1,3)+Rational(1/4)  % 1/3+1/4 = 7/12
  7 / 12
Rational(pi)+1               % uses 355/113 as approximation of pi
  468/113
```

A demonstration example in this release includes the complete definition of the `Rational` class. Run `help Rational` or `type Rational` to see its contents.

Each operator has a corresponding function that the software invokes when at least one of the operands is an object. This is the function that the class must provide for it to define an overloaded operator. The example above includes overloading of the +

operator by defining the `plus` member method. Table 12-1 contains the complete map between operators and functions:

TABLE 12-1: OPERATORS AND THEIR CORRESPONDING FUNCTIONS

OPERATOR	FUNCTION
+ (unary, +a)	uplus
+ (binary, a+b)	plus
- (unary, -a)	uminus
- (binary, a-b)	minus
*	mtimes
.*	times
/	mrdivide
./	rdivide
\	mldivide
.\	ldivide
^	mpower
.^	power
==	eq
~=	ne
>=	ge
>	gt
<=	lt
<	le
&	and
|	or
~	not
[, ,]	horzcat
[; ;]	vertcat
:	colon
'	ctranspose
.'	transpose

FIELD READ/WRITE

If a class has an instance method called `fieldread`, then COMSOL Script calls that function when it reads a field of an object using the `var.field` syntax. The `fieldread` method must take one input argument and return one output:

```
function out = fieldread(field)
```

The `field` argument is the string that follows the `.` (dot) in `var.field`. By overloading `fieldread` it is possible to let a class behave as if it has fields that it does not have. You can use this overloading for making the representation of the data independent of the interface to the class.

Similarly, if a class has an instance function called `fieldwrite`, then that function is called when a field of an object is written using the `var.field = val` syntax. It must take two input arguments and not return anything:

```
function fieldwrite(field, val)
```

The `field` parameter is the string that followed the `.` in `var.field` and the `val` parameter is the value to which the field was assigned. You can use `fieldwrite` when the fields of the class have dependencies: Changing the value of one field can force the recalculation of others.

ARRAY READ/WRITE

An object can only be indexed using parentheses if it has an instance method called `arrayread`: If `c` is an object, then `c(args)` is interpreted as `c.arrayread(args)`, where `arrayread` is invoked for the arguments (one or more) and is expected to return one value:

```
function out = arrayread(args)
```

Overloading `arrayread` can be useful for classes where the parentheses denote evaluation, for instance in a class representing a mathematical function of one or more variables.

Similarly, array assignment using the syntax `c(args) = val` is interpreted as `c.arraywrite(args, val)`.

CELL ARRAY READ/WRITE

It is only possible to index an object using curly brackets if it has an instance method called `cellread`: If `c` is an object, then `c{args}` is interpreted as `c.cellread(args)` where `cellread` is invoked for the arguments (one or more) and is expected to return one value:

```
function out = cellread(args)
```

Similarly, cell array assignment using the syntax `c{args} = val` is interpreted as `c.cellwrite(args, val)`.

OVERLOADING END

When indexing into an object that has an instance method called `end`, then the value of `end` is resolved by calling this method with two arguments: The index where `end` occurs and the total number of indices. E.g. `a(1:end, 5)` would be equivalent to `a(1:a.end(1, 2))` if `a` belongs to a class with an overloaded `end` method. If the class has no overloaded `end` method but instead has an overloaded `size` method, then the standard semantics of `end` are used with the size evaluated using the `size` method.

Overloading Save and Load

When an object is saved to file using `save`, the default behavior is to save all nontransient instance fields. It is possible to override this behavior by a class providing a `writeobject` method. This method must have the interface

```
public function out = writeobject
```

COMSOL Script writes the output from `writeobject` to file when the object is saved.

If the class has an overloaded `writeobject` method it usually also has to provide an overloaded `readobject` method: This method is called when a object is loaded from file using `load`. It must have the interface

```
public function readobject(data)
```

When the software loads an object, it first creates an empty instance of its class. Then the `readobject` method is invoked, and the data argument is the output from `writeobject` when the object was saved.

You can overload `readobject` but not `writeobject`. In this case, the input to `readobject` is the default representation of the object: As a cell array with one column for each level of the class hierarchy. Consider the `Rectangle` class inheriting from `Shape`:

```
class Shape
public name
...
```

```
class Rectangle extends Shape
public x0 x1
public y0 y1
```

A `Rectangle` object with vertices in (2, 3) and (5, 7) and the name `'MyRect'` would be saved as the cell array

```
{'Shape'                 'Rectangle'
 struct('name','MyRect') struct('x0',2,'x1',5,'y0',3, y1',7}
```

This would be the argument to an overloaded `readobject` in `Rectangle` if no overloaded `writeobject` was present. There is one column for each level of the class hierarchy. The first row contains the name class names, and the second row contains a structure with one field for each nontransient field on that level of the hierarchy.

Overloading Display

The default way to display an object is to display its public fields like the fields of a structure:

```
r = Rectangle(1, 3, 2, 9)

r =
Rectangle object
  x0:  [1]
  x1:  [2]
  y0:  [3]
  y1:  [9]
```

If the class has a public nonstatic function called `display`, COMSOL Script invokes it when it displays an object of the class. You can extend the `Rectangle` class with such a method:

```
public function display
sprintf('Rectangle with corners (%.2f,%.2f) and (%.2f,%.2f).', ...
        x0, y0, x1, y1)
```

This results in the following output:

```
r = Rectangle(1, 3, 2, 9)

ans =
Rectangle with corners (1.00, 3.00) and (2.00, 9.00).
```

Precedence

Precedence Between Functions and Methods

Methods in different classes can have the same name, and method names can also coincide with names of functions, both built-in functions and M-files. COMSOL Script resolves a function invocation of the form `func(arg1, ...)` by considering possible interpretations in the following order:

1. As an array index expression, if there is a variable called `func` in the workspace.
2. As the invocation of the static method `func` of the class of any object in the argument list.
3. As a call to the built-in function `func`.
4. As a call to the method `func` in the class where execution currently takes place.
5. As a call to the user-defined function `func`.
6. Interpretations 3–5 tried using case-insensitive name matching.

The `Rectangle` class contains a static method called `overlap`. Suppose that there also is an M-file, `overlap.m`. By rule 2 above,

```
overlap(Rectangle(1,2,3,4), Rectangle(5,6,7,8))
```

invokes the `overlap` method of the `Rectangle` class, but

```
overlap(5, 7)
```

instead calls `overlap.m`.

Precedence Between Methods from Different Classes

When a function call contains several object arguments, COMSOL Script normally considers classes in the order they appear in the argument list: Suppose that the classes `Rectangle` and `Circle` both have a static `overlap` method. Then

```
overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

is interpreted as

```
Rectangle.overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

The `overlap` method of the `Circle` class would only be run if there were no `overlap` method in `Rectangle`.

In the example above, the two classes `Rectangle` and `Circle` have equal precedence. You can specify the precedence between classes using `superiorto` and `inferiorto` in the class file. The classes listed in a `superiorto` declaration have lower priority than the current class, those listed in an `inferiorto` declaration have higher priority.

The `Circle` class can be modified as follows:

```
class Circle

superiorto Rectangle
```

With this change, the methods of the `Circle` class are always given priority over methods in the `Rectangle` class when the argument list contains `Circle` and `Rectangle` objects. Then

```
overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

is interpreted as

```
Circle.overlap(Rectangle(2, 3, 4, 5), Circle(1, 2, 3))
```

It is possible to combine several `superiorto` and `inferiorto` declarations in the same class:

```
class Circle

superiorto Rectangle Square
inferiorto Hexagon
```

Note: Excessive use of `superiorto` and `inferiorto` makes the program flow hard to follow.

Using Classes as Packages

You can use classes to bundle groups of functions together: A class without instance fields where all methods are `public` and `static` can serve as a container for a set of related functions. As an example, consider the following class:

```
class MyMaths

static function y = sinc(x)
%SINC(X) returns SIN(X)/X if X is nonzero, otherwise 1.
y = ones(size(x));
ix = find(x~=0);
y(ix) = sin(x(ix))./x(ix);

static function y = smhs(x, scale)
%SMHS(x) approximates the step function Y=(X>0) by smoothing the
%transition within the interval -SCALE < X < SCALE.
x=x./scale;
y=(x>-1 & x<1).*(0.5+x.*(0.75-0.25*x.*x))+(x>=1);
```

You can view `MyMaths` as a package that contains the two functions `sinc` and `smhs`:

```
MyMaths.sinc(0:3)

ans =
        1     0.8415     0.4546     0.0470
```

By creating a class containing a group of related functions, preferably small, it becomes easier to maintain the functions, as only one file is ever changed. Sharing code between functions also becomes easier.

13

Java Interface

Java is a powerful programming language with a large library of built-in classes. COMSOL Script provides support for creating and manipulating Java objects. This makes it possible to connect COMSOL Script with existing Java programs and to create scripts where the COMSOL Script and Java languages are mixed.

Declaration of Java Methods

For COMSOL Script to be able to invoke a Java method, it needs its declaration. You supply declarations using the `javaDeclare` command:

```
javaDeclare('my.decls')
```

The argument to `javaDeclare` is the name of a file containing Java declarations of all methods and constructors that you want to be able to call. A sample declaration file can have the following contents:

```
// Sample declaration file my.decls

// Constructor and methods taken from java.lang.String
java.lang.String(byte[]);
static java.lang.String java.lang.String.valueOf(double);
java.lang.String java.lang.String.substring(int, int);

/* Another class */
java.lang.Double(double);
```

Conventions and restrictions:

- `public` visibility is assumed if no visibility is specified. Only `public` methods can be called through the Java interface.
- Abbreviated class names are not accepted. For example, `String` is not expanded into `java.lang.String`.
- For overloaded methods, the number of arguments must differ. It is not possible to declare two methods in a class that have the same name and number of arguments.

Creating and Using Java Objects

Creating Java Objects and Invoking Methods

To create a Java object, use any Java class constructor:

```
s = java.lang.String('Multiphysics')
s =
Multiphysics

d = java.lang.Double(17)
d =
17.0
```

This code snippet creates both a `java.lang.String` object `s` with the value `'Multiphysics'` and a `java.lang.Double` object `d` with the value 17.

To invoke public fields or member methods, use the same notation as for structs:

```
s.toUpperCase
ans =
MULTIPHYSICS

s.substring(5)
ans =
physics

d.isInfinite
ans =
false
```

To create a Java object with a constructor without arguments, omit the argument list:

```
s = java.lang.String;
```

A similar notation is used when accessing static class members:

```
java.lang.Double.MAX_VAL
ans =
1.798e+308
```

Creating and Using Java Arrays

You cannot use the constructor syntax to create an array of Java objects. It is, however, possible to do so using the `javaArray` function:

```
s = javaArray('java.lang.String',3)
s =
```

```
null
null
null
```

This command creates a 3x1 array of `java.lang.String` objects. The elements of the array are not initialized during creation, hence all elements of the array are `null`. To access or modify the array, use the same syntax as for all other data types:

```
s(2) = java.lang.String('COMSOL')
s =
  null
  COMSOL
  null
```

COMSOL Script supports only 1D Java array objects.

Functions for Creating and Using Java Objects

The standard way to create a Java object is to use the constructor as described earlier. It requires that you know the class name, thus you cannot use a class constructor when the class name is stored in a variable. In this case, turn to the `javaObject` function:

```
cl = 'java.lang.String';
s = javaObject(cl, 'Multiphysics')
```

This is equivalent to the explicit creation method using the `java.lang.String` constructor syntax.

The same problem can arise when accessing a static method, and for this purpose use the `javaMethod` function:

```
javaMethod('parseDouble', 'java.lang.Double', '3.14')
ans =
3.140000
```

This is equivalent to `java.lang.Double.parseDouble('3.14')`.

Passing Java Objects as Arguments to Functions

Like all other data types, Java objects can be sent as arguments to functions. Parameter passing does not work in the same way for Java objects as it does for the other data types. In Java, when an object serves as an argument to a method, the software sends a reference. COMSOL Script uses the same semantics for Java objects, and this means that if a function modifies an input argument that is a Java object, then the corresponding object in the caller's workspace is modified. Similarly, when the code contains assignments between Java variables, only the reference is copied.

As an example, consider the following function, which returns the reversal of a `java.util.ArrayList` object:

```
function out = revlist(list)
out = list;
for i=0:(out.size-1)/2
  j = out.size-i-1;
  tmp = out.get(i);
  out.set(i, out.get(j));
  out.set(j, tmp);
end
```

The returned list is indeed the reversal of the input argument, but running this code modifies the input in the process:

```
list = java.util.ArrayList;
list.add(2);
list.add(3);
list.add(5);
list2 = revlist(list)
list2 =
[5.0, 3.0, 2.0]
list
list =
[5.0, 3.0, 2.0]
```

Note that the assignment `out = list;` in `revlist` does not solve the problem because this command only copies a reference to the Java object. A better solution is to replace this assignment with `out = list.clone;`, which creates a new object instead of a copy.

Type Conversions

Conversion of Arguments to Java Methods

For most types, the conversion of a value from COMSOL Script is straightforward. When converting from a COMSOL Script array to a Java array, it converts element-by-element, possibly using rounding or truncation if the range of the Java type is smaller than the range of the COMSOL Script type.

A special case is the conversion of the empty matrix, `[]`. When mapped to `java.lang.Object`, it is converted to the `null` reference.

Return Values From Java Methods

When converting return values from Java methods, COMSOL Script uses the following type map:

TABLE 13-1: TYPE MAP FROM JAVA TO COMSOL SCRIPTS

JAVA TYPES	SCRIPT TYPE
char, java.lang.Character	character
boolean, java.lang.Boolean	logical
byte, java.lang.Byte,short, java.lang.Short, int, java.lang.Integer, long, java.lang.Long, float, java.lang.Float, double, java.lang.Double	double
null	[]
java.Object	Java object of the same type

COMSOL Script preserves the number of dimensions during the return-value conversion; it converts a Java `double[][][]` array into a 3D double matrix in COMSOL Script.

The Char and Double Conversions

CHAR

The `char` function can convert Java objects to character matrices:

```
s = java.lang.String('COMSOL');
t = char(s);
ischar(t)
ans =
true
```

For a `java.lang.String` object, `char` returns a character row vector containing the characters of the string. For a `java.lang.String[]`, it returns a character matrix where each row corresponds to one element of the array.

`char` can convert other Java types only if they contain a public `toChar` method that returns a `java.lang.String`. If so, `char` invokes the `toChar` method and then proceeds to convert the returned `java.lang.String`.

DOUBLE

The `double` function can similarly convert Java objects to double matrices:

```
n = java.lang.Integer(8192);
d = double(n)
d =
8192
```

For `java.lang.Number` objects or arrays, `double` returns a double matrix that contains the result of converting each element to a double.

`double` can convert other Java types only if they contain a public `toDouble` method that returns a double.

14

External API

COMSOL Script is a powerful environment, but sometimes you might want to interface it to code written in a low-level language. This chapter describes an API used for connecting Script with C code.

Introductory Examples

This section contains two small examples that illustrate how to proceed when interfacing to external code. The files for the examples are available in the `script/demos` directory in the installation.

Compiling and Executing a Simple Function

Consider the expression `sum(sin(A(:)))`. It is very compact and usually fast enough, but the price you pay for the compactness is that it needs more memory and is slower than the corresponding C code. It is therefore a good candidate for implementation as an external function. An external function has an .M-file that describes its interface and contains its help text (if any). So, as a quick exercise, follow the steps to create `sinsum.m`:

```
function s = sinsum{'sinsum', 'work'}(A)
%SINSUM(A)
%   S = SINSUM(A) returns sum(sin(A(:))) where A is full real.
```

The syntax `{'sinsum', 'work'}` means that the function `work` in the shared library `sinsum` implements the function.

The source file `sinsum.c` contains the implementation of the algorithm:

```
#include <math.h>
#include "scriptext.h"

CL_EXPORT void work(clEnv *env, int nOut, clData *out[],
                    int nIn, clData *in[]) {
  int i;
  size_t nElems;
  double sum = 0;
  double *ptr;
  if ((nOut!=1) || (nIn!=1) || (clGetType(in[0])!=CL_REAL))
    clError(env, "Bad arguments.", 1);
  nElems = clGetNElems(in[0]);
  ptr = clGetRealPtr(in[0]);
  for (i=0; i<nElems; i++)
    sum += sin(ptr[i]);
  out[0] = clNewReal(env, sum);
}
```

This small file illustrates many parts of the API:

- The header file `scriptext.h` contains the declaration of the API.
- The interface of the library entry point, here the function `work`, always has the same format as in this example. The parameters have the following interpretations:
  - `env` is the script environment. Many API functions take it as an argument.
  - `nOut` is the number of outputs expected from the function.
  - `out` is a vector through which Script retrieves the outputs when the function completes successfully.
  - `nIn` is the number of input arguments.
  - `in` is a vector of input arguments.
- External code must perform complete argument tests because incorrect assumptions typically result in crashes. The function `clError` is used for abnormal termination of the external function.
- The contents of matrices are available through raw C pointers, here illustrated with the call to `clGetRealPtr`. All full matrices are stored in column-major order.

To compile the code, enter the following at the Script prompt:

```
compile -O sinsum.c
```

COMSOL Script might not be able to determine which compiler to use, especially if you are running Windows. See the section "Compilation" for more advice on this aspect. The `-O` option enables optimization; it can be omitted.

If the compilation was successful you can call `sinsum`:

```
sinsum(1:5)
ans =
   0.1762
```

You can also verify that it is faster than the corresponding .M-code:

```
A = rand(2000)-0.5;
tic, sum(sin(A(:))), toc
ans =
-495.4401
Elapsed time:  0.437 s
tic, sinsum(A), toc
ans =
-495.4401
Elapsed time:  0.281 s
```

The C version is almost twice as fast and uses less memory, but it is more specialized and requires much more code. This example is somewhat contrived, but there are cases where rewriting a tight loop in C code gives a large performance improvement.

Working with Sparse Matrices

You can retrieve the elements of full matrices by indexing in an array. For sparse matrices the representation is space-efficient but more complex. The topic of this example is a function that computes the Euclidean norms of the rows of a sparse matrix `A`. This can be computed as `sqrt(sum(A.^2, 2))`, but the more compact solution is slower and needs more memory.

The interface to the external function is defined as follows in `rownorm.m`:

```
function d = rownorm{'rownorm', 'work'}(A)
%ROWNORM(A)
%  D = ROWNORM(A) returns a vector D such that D(N) is the Euclidean
%  norm of the Nth row of the sparse matrix A. This is equivalent to
%  sqrt(sum(A.^2, 2)).
```

The source file `rownorm.c` contains the implementation of the algorithm:

```
#include <math.h>
#include "scriptext.h"

CL_EXPORT void work(clEnv *env, int nOut, clData *out[],
                    int nIn, clData *in[]) {
  int i, j;
  int nRows, nCols;
  int row, col;
  double *outPtr;
  size_t *colPtr;
  size_t *rowPtr;
  double *dataPtr;
  if ((nOut!=1) || (nIn!=1) || (clGetType(in[0])!=CL_REAL_SPARSE))
    clError(env, "Bad arguments.", 1);
  nRows = clGetSize(in[0], 0);
  nCols = clGetSize(in[0], 1);
  out[0] = clNewFull2D(env, CL_REAL, nRows, 1);
  outPtr = clGetRealPtr(out[0]);
  for (i=0; i<nRows; i++)
    outPtr[0] = 0.0;
  /* Retrieve sparse representation of in[0]. */
  colPtr = clGetSparseColPtr(in[0]);
  rowPtr = clGetSparseRowPtr(in[0]);
```

```
    dataPtr = clGetSparseRealPtr(in[0]);
    for (col=0; col<nCols; col++) {
      /* Iterate over nonzero elements in column col. */
      for (i=colPtr[col]; i<colPtr[col+1]; i++) {
        double val = dataPtr[i];
        outPtr[rowPtr[i]] += val*val;
      }
    }
    for (i=0; i<nRows; i++)
      outPtr[i] = sqrt(outPtr[i]);
  }
```

The entry point and error-handling code are similar to the previous example, but there are several new features:

- The return value, a real `nRows` × 1 vector, is allocated using `clNewFull2D`. A pointer to the data is obtained using `clGetRealPtr`.
- The sparse input argument, `in[0]`, is represented as a triplet of vectors:
  - A vector `colPtr` such that `colPtr[n]` is the number of nonzero entries in columns less than `n`
  - A vector `rowPtr` containing the nonzero elements' row numbers
  - A vector `dataPtr` containing the nonzero elements' values
- The loop iterating over the nonzero elements is typical for how this should be done. It iterates over the columns of `in[0]`, and for each column it deals with its nonzero elements. You should always iterate over the columns and not the rows of a sparse matrices because this leads to simpler and faster code.

Compiling and running the code is done as earlier:

```
compile -O rownorm.c
rownorm(sprand(3, 10, 0.1))

ans =
     0.3692
     0.3435
     0.0386
```

Using the API

M-File Interface

A function with an external implementation has a function header but no function body except for possibly a comment block. The function header has the following structure:

```
function outargs = func{lib name, entry point}(inargs)
```

`lib name` is the name of the shared library (a .dll or .so file depending on the operating system) containing the implementation, and `entry point` is the `name` of the entry point to use when invoking the function. The library name and entry point are both strings, such as in the earlier example:

```
function s = sinsum{'sinsum', 'work'}(A)
```

The library and entry point names can, but need not, coincide.

The function's input and output arguments, `inargs` and `outargs`, respectively, are specified in the same way as for normal functions.

Entry Point

The entry point was discussed in the examples in the previous section. It always has the same signature:

```
CL_EXPORT void work(clEnv *env, int nOut, clData *out[],
                    int nIn, clData *in[]);
```

The `nIn` elements of the `in` vector are the inputs to the function, the `nOut` elements of the `out` vector should contain the outputs from the function if it returns successfully. The `CL_EXPORT` tag is necessary for the function to get the proper linkage.

A library can provide implementations of several functions by having one entry point for each function.

Memory Management

Externally invoked code can allocate memory in two ways:

- Using the standard C allocation functions `malloc`, `calloc`, and `realloc`
- Using the API wrapper functions `clMalloc`, `clCalloc`, and `clRealloc`

The second alternative has the advantage that memory leaks are no longer possible; when execution returns to COMSOL Script from the external function, all allocated but not freed blocks are freed automatically.

The code invoking the external function assumes that it is a pure function in the sense that it does not maintain an internal state by storing items—for example, pointers to `clData`—in static variables.

Error Handling

The API functions communicate errors to the caller in their return values. For instance, a number of functions return `NULL` in case of a error. Always check the return values and do not make any unjustified assumptions about the data types of input arguments to the external function or else your code will crash.

API Reference

Data Types

CLENV

Represents the execution environment in which external functions run.

CLDATA

Represents a matrix of any type and size. Each matrix type corresponds to a symbolic constants defined in `scriptext.h`:

- `CL_INVALID`: Illegal matrix type
- `CL_REAL`: Full real
- `CL_COMPLEX`: Full complex
- `CL_LOGICAL`: Full logical
- `CL_CHAR`: Full character
- `CL_UINT8`: Full uint8
- `CL_REAL_SPARSE`: Sparse real
- `CL_COMPLEX_SPARSE`: Sparse complex

All values except for `CL_INVALID` can be returned by the function `clGetType`. `CL_INVALID` can be used as a sentinel value that is guaranteed not to collide with any existing matrix type.

CLLOGICAL

Represents a logical scalar, implemented as an `unsigned char`.

CLCHAR

Represents a character scalar, implemented as an `unsigned short`.

CLUINT8

Represents a uint8 scalar, implemented as an `unsigned char`.

CLCOMPLEX

Represents a complex element in a sparse matrix, implemented as a struct containing a real and an imaginary part.

CLFUNCTION

The function pointer type of an entry point to a library. Usually this type need not be referred to explicitly.

Functions

Most functions in this API are accessors for data stored in matrices, but some perform more complex tasks. A rough categorization of the functions is as follows:

TABLE 14-1: TYPE ACCESSOR

FUNCTION NAME	DESCRIPTION
`clGetType`	Matrix type

TABLE 14-2: SIZE ACCESSORS

FUNCTION NAME	DESCRIPTION
`clGetNDims`	Number of dimensions
`clGetSize`	Size of a dimension
`clGetNElems`	Number of elements

TABLE 14-3: FULL MATRIX ACCESSORS

FUNCTION NAME	DESCRIPTION
`clGetRealPtr`	Pointer to real data
`clGetImagPtr`	Pointer to imaginary data
`clGetCharPtr`	Pointer to character data
`clGetUint8Ptr`	Pointer to uint8 data
`clGetLogicalPtr`	Pointer to logical data

TABLE 14-4: SPARSE MATRIX ACCESSORS

FUNCTION NAME	DESCRIPTION
`clGetSparseColPtr`	Pointer to column pointers
`clGetSparseRowPtr`	Pointer to row indices
`clGetSparseRealPtr`	Pointer to nonzero data of real sparse matrix
`clGetSparseComplexPtr`	Pointer to nonzero data of complex sparse matrix

TABLE 14-4: SPARSE MATRIX ACCESSORS

FUNCTION NAME	DESCRIPTION
`clGetNzmax`	Allocation of nonzero elements
`clGetNnz`	Number of nonzero elements

TABLE 14-5: MATRIX CREATORS

FUNCTION NAME	DESCRIPTION
`clNewFull`	Create full matrix
`clNewFull2D`	Create full 2D matrix
`clNewReal`	Create real scalar
`clNewComplex`	Create complex scalar
`clNewChar`	Create character string
`clNewSparse`	Create sparse matrix
`clNewCopy`	Make copy of matrix

TABLE 14-6: MEMORY MANAGEMENTS

FUNCTION NAME	DESCRIPTION
`clMalloc`	Allocate memory with malloc
`clCalloc`	Allocate memory with calloc
`clRealloc`	Reallocate memory with realloc
`clFree`	Free allocated matrix or memory

TABLE 14-7: NUMERIC UTILITIES

FUNCTION NAME	DESCRIPTION
`clGetEps`	Get eps
`clGetInf`	Get infinity
`clGetNaN`	Get not-a-number

TABLE 14-8: ERROR HANDLING

FUNCTION NAME	DESCRIPTION
`clGetLastError`	Get last error message
`clClearError`	Clear error message

TABLE 14-8: ERROR HANDLING

FUNCTION NAME	DESCRIPTION
`clError`	Set error message
`clWarning`	Emit warning

TABLE 14-9: CALLBACKS

FUNCTION NAME	DESCRIPTION
`clEvalExpr`	Evaluate expression or statement
`clEvalFunc`	Call function

CLCALLOC

`void * clCalloc(clEnv *env, size_t nmemb, size_t size)`

Allocates and clears memory using the standard C function `calloc`. The memory block is deallocated when the external function returns unless it has already been deallocated. Returns `NULL` if memory allocation fails or if `size` is 0.

Inputs:

- `env` is the script environment.
- `nmemb` is the number of elements to allocate.
- `size` is the size of each element in bytes.

CLCLEARERROR

`void clClearError(clEnv *env)`

Clears the error message previously set either using `clError` or as a result of failures in the callback functions `clEvalExpr` and `clEvalFunc`.

Input: `env` is the script environment.

CLERROR

`void clError(clEnv *env, const char *msg, int returnNow)`

Sets the error message to `msg`. If an error message is set when the external function returns, then it is passed to the script that is running. Calling with an empty `msg` is equivalent to calling `clClearError`.

Inputs:

- `env` is the script environment.

- `msg` is the error message.
- `returnNow` is a flag that controls whether execution of the external function should stop immediately. Use this feature only when running C code; it is implemented using `longjmp`, and using it in C++ code almost never works because destructors are not run and exception handling could fail.

CLEVALEXPR

```
clData * clEvalExpr(clEnv *env, const char *expr, int getRes)
```

Evaluates an expression using Script. If evaluation fails, the error message is stored and can be retrieved using `clGetLastError`.

Inputs:

- `env` is the script environment.
- `expr` is the expression to evaluate.
- `getRes` is a flag to set if you want the results of the expression: If nonzero it is returned, otherwise it is ignored.

Example: `a = clEvalExpr(env, 'sin(1)', 1)` assigns to `a` the value of sin(1).

CLEVALFUNC

```
void clEvalFunc(clEnv *env, const char *func, int nOut,
                  clData *out[], int nIn, clData *in[])
```

Evaluates a function using Script. If evaluation fails, the error message is stored and can be retrieved using `clGetLastError`.

Inputs:

- `env` is the script environment.
- `func` is the function to call.
- `nOut` is the number of output arguments expected.
- `out` is a vector in which outputs are placed if the evaluation succeeds.
- `nIn` is the number of input arguments.
- `in` is a vector of input arguments.

Example:

`clData *grid[2]; evalFunc(env, 'meshgrid', 2, grid, xrange, yrange)` computes a 2D grid using `meshgrid` of `xrange` and `yrange` and places the grid's x-coordinates in `grid[0]` and its y-coordinates in `grid[1]`.

CLFREE

```
void clFree(clEnv *env, void *ptr)
```

Deallocates a matrix or memory block.

Inputs:

- `env` is the script environment.
- `ptr` is either a matrix created using the one of the `clNew` functions or a memory block allocated using either the standard C allocation functions (`malloc`, `calloc`, and `realloc`) or their counterparts in this API (`clMalloc`, `clCalloc`, and `clRealloc`).

CLGETCHARPTR

```
clChar * clGetCharPtr(clData *data)
```

Returns a pointer to the first element of a nonempty character matrix. For any other matrix type it returns `NULL`. The matrix elements are stored in column-major order.

Input: `data` is a nonempty character matrix.

CLGETEPS

```
double clGetEps(clEnv *env)
```

Returns the smallest real number ε such that 1+ε is greater than 1.

Input: `env` is the script environment.

CLGETIMAGPTR

```
double * clGetImagPtr(clData *data)
```

Returns a pointer to the first element of the imaginary part of a nonempty full complex matrix. For any other matrix type it returns `NULL`. The matrix elements are stored in column-major order.

Input: `data` is a nonempty full complex matrix.

CLGETINF

```
double clGetInf(clEnv *env)
```

Returns infinity.

Input: `env` is the script environment.

CLGETLASTERROR

```
const char * clGetLastError(const clEnv *env)
```

Returns the most recently set error message; returns `NULL` if there were none. Error messages are set when the callback functions `clEvalExpr` and `clEvalFunc` fail, or by the external function itself using `clError`. The pointer returned is only valid until the next time any function in the API is called.

Input: `env` is the script environment.

CLGETLOGICALPTR

```
clLogical * clGetLogicalPtr(clData *data)
```

Returns a pointer to the first element of a nonempty logical matrix. For any other matrix type it returns `NULL`. The matrix elements are stored in column-major order.

Input: `data` is a nonempty logical matrix.

CLGETNAN

```
double clGetNaN(clEnv *env)
```

Returns the not-a-number value, that is, the result of undefined operations such that 0/0.

Input: `env` is the script environment.

CLGETNDIMS

```
int clGetNDims(const clData *data)
```

Returns the number of dimensions of `data`. All matrices have at least two dimensions; trailing unit dimensions beyond dimension two are ignored.

Input: `data` is a matrix.

CLGETNELEMS

```
size_t clGetNElems(const clData *data)
```

Returns the total number of elements in a matrix, that is, the product of the size vector.

Input: `data` is a matrix.

CLGETNNZ

```
size_t clGetNnz(const clData *data)
```

Returns the number of nonzero elements in a matrix.

Input: `data` is a matrix.

CLGETNZMAX

```
size_t clGetNzmax(const clData *data)
```

Returns the allocation of nonzero elements. For a full matrix this is always the product of the size vector, but for a sparse matrix it can be anything from 0 to the product of the size vector. It is always greater than or equal to `clGetNnz`.

Input: `data` is a matrix.

CLGETREALPTR

```
double * clGetRealPtr(clData *data)
```

Returns a pointer to the first element of the real part of a nonempty full real matrix. For any other matrix type it returns `NULL`. The matrix elements are stored in column-major order.

Input: `data` is a nonempty full real or complex matrix.

CLGETSIZE

```
size_t clGetSize(const clData *data, int dim)
```

Returns the size of the `dim`th dimension. The dimensions start from 1, so you can use `clGetSize(data, 1)` and `clGetSize(data, 2)` to retrieve the number of rows and columns, respectively, of `data`. If `dim` is less than 1 or larger than the number of dimensions of `data`, it returns 1.

Inputs:

- `data` is a matrix.
- `dim` is the dimension whose size is returned.

CLGETSPARSECOLPTR

```
size_t * clGetSparseColPtr(clData *data)
```

Returns a pointer to the sparse matrix column pointers if `data` is sparse and `NULL` if `data` is full. The returned pointer `col` has the following properties:

- `col[n]` is the number of nonzero elements in the first `n` columns of data for any `n` between 0 and `nCols` inclusive, where `nCols` is the number of columns of `data`.
- The nonzero elements in column `n` have indices `col[n-1]...col[n]-1` in the blocks returned by `clGetSparseComplexPtr`, `clGetSparseRealPtr`, and `clGetSparseRowPtr`.

Note: Any changes made to the vector must respect these properties when the external function returns.

Input: `data` is a sparse matrix.

CLGETSPARSECOMPLEXPTR

`clComplex * clGetSparseComplexPtr(clData *data)`

Returns the vector containing the values of the nonzero elements of the complex sparse matrix `data`. The length of the vector is `clGetNzmax()`. It returns `NULL` if `data` is full or real.

See `clGetSparseColPtr` for how the flat vector of values is mapped to matrix elements.

Input: `data` is a sparse complex matrix.

CLGETSPARSEREALPTR

`double * clGetSparseRealPtr(clData *data)`

Returns the vector containing the values of the nonzero elements of the real sparse matrix data. The length of the vector is `clGetNzmax()`. it returns `NULL` if `data` is full or complex.

See `clGetSparseColPtr` for how the flat vector of values is mapped to matrix elements.

Input: `data` is a sparse real matrix.

CLGETSPARSEROWPTR

`size_t * clGetSparseRowPtr(clData *data)`

Returns the vector containing the row numbers of the nonzero elements of `data` if `data` is sparse, otherwise it returns `NULL`. The length of the vector is `clGetNzmax()` and the row numbers are zero-based.

See clGetSparseColPtr for how the flat vector of row numbers is mapped to matrix elements.

Input: data is a sparse matrix.

CLGETTYPE

```
int clGetType(const clData *data)
```

Returns the type of a matrix. Possible values are the following constants, all declared in scriptext.h:

- CL_REAL: Returned if data is full real
- CL_COMPLEX: Returned if data is full complex
- CL_LOGICAL: Returned if data is full logical
- CL_CHAR: Returned if data is full character
- CL_UINT8: Returned if data is full uint8
- CL_REAL_SPARSE: Returned if data is sparse real
- CL_COMPLEX_SPARSE: Returned if data is sparse complex

Input: data is a matrix.

CLGETUINT8PTR

```
clUint8 * clGetUint8Ptr(clData *data)
```

Returns a pointer to the first element of a nonempty uint8 matrix. For any other matrix type it returns NULL. The matrix elements are stored in column-major order.

Input: data is a nonempty uint8 matrix.

CLMALLOC

```
void * clMalloc(clEnv *env, size_t size)
```

Allocates memory using the standard C function malloc. The memory block is deallocated when the external function returns unless it has already been deallocated. Returns NULL if memory allocation fails or if size is 0.

Inputs:

- env is the script environment.
- size is the number of bytes to allocate.

CLNEWCHAR

```
clData * clNewChar(clEnv *env, const char *str)
```

Creates and returns a character row vector. The returned matrix is 0×0 if `str` is empty, otherwise $1 \times n$ where n is the length of `str`.

Inputs:

- `env` is the script environment.
- `str` is a null-terminated string.

CLNEWCOMPLEX

```
clData * clNewComplex(clEnv *env, double real, double imag)
```

Creates a complex 1×1 matrix.

Inputs:

- `env` is the script environment.
- `real` is the real part.
- `imag` is the imaginary part.

CLNEWCOPY

```
clData * clNewCopy(const clData *orig)
```

Makes a copy of a matrix. The contents of the two matrices are not shared, so changing the copy does not affect `orig`.

Input: `orig` is a matrix.

CLNEWFULL

```
clData * clNewFull(clEnv *env, int type, int nDims,
  const size_t *dims)
```

Creates and returns a new full matrix. It returns `NULL` if the creation failed. The returned matrix always has at least two dimensions; if 0 or 1 dimensions are supplied, trailing unit dimensions are added to make the matrix 2D.

Inputs:

- `env` is the script environment.
- `type` is the matrix type, must be one of `CL_REAL`, `CL_COMPLEX`, `CL_LOGICAL`, `CL_CHAR`, or `CL_UINT8`.

- `nDims` is the number of dimensions.
- `dims` contains the dimensions of the matrix.

CLNEWFULL2D

```
clData * clNewFull2D(clEnv *env, int type, size_t nRows,
  size_t nCols)
```

Creates and returns a new full 2D matrix. It returns `NULL` if the creation failed.

Inputs:

- `env` is the script environment.
- `nRows` is the number of rows.
- `nCols` is the number of columns.

CLNEWREAL

```
clData * clNewReal(clEnv *env, double val)
```

Creates and returns a real 1×1 matrix.

Inputs:

- `env` is the script environment.
- `val` is the value.

CLNEWSPARSE

```
clData * clNewSparse(clEnv *env, int type, size_t nRows,
  size_t nCols, size_t nzMax)
```

Creates and returns a sparse matrix. It returns `NULL` if the creation failed.

Inputs:

- `env` is the script environment.
- `type` is the matrix type, must be `CL_REAL_SPARSE` or `CL_COMPLEX_SPARSE`.
- `nRows` is the number of rows.
- `nCols` is the number of columns.
- `nzMax` is the allocation of nonzero elements. This is an upper bound on the number of nonzero elements with which the matrix can be populated.

CLREALLOC

```
void * clRealloc(clEnv *env, void *ptr, size_t size)
```

Reallocates memory using the standard C function `realloc`. The memory block is deallocated when the external function returns unless it has already been deallocated. It returns `NULL` if memory allocation fails or if `size` is 0.

Inputs:

- `env` is the script environment.
- `ptr` is the memory block to reallocate, originally allocated using either the standard C allocation functions (`malloc`, `calloc`, and `realloc`) or their counterparts in this API (`clMalloc`, `clCalloc`, and `clRealloc`).
- `size` is the new size of the block (in bytes) after reallocation.

CLWARNING

```
void clWarning(clEnv *env, const char *msg, const char *id)
```

Emits a warning when the external function returns. Calling this function has the same effect as running `warning(msg,id)` in a script.

Inputs:

- `env` is the script environment.
- `msg` is the warning message.
- `id` is the warning category.

Compilation

Compiling from Within COMSOL Script

The `compile` function is used when compiling from within COMSOL Script. It takes as its argument one or more C source files. The default behavior is to compile the files and link them into a shared library.

Configuration Files

A configuration file is used to define the characteristics of a platform/compiler combination. If this combination is not explicitly set with the `-f` argument to `compile` (see below), it uses a default configuration. The default files are located in `COMSOLDIR/bin` where `COMSOLDIR` is the directory where COMSOL Script was installed.

TABLE 14-10: CONFIGURATION FILE NAMES

PLATFORM	COMPILER
PC/Windows (32-bit)	`compileopts_win32`
PC/Windows (64-bit)	`compileopts_win64`
PC/Linux (32-bit)	`compileopts_glnx86`
PC/Linux (64-bit)	`compileopts_glnxa64`
Itanium/Linux	`compileopts_glnxi64`
Sun/Solaris (32-bit)	`compileopts_sol2`
Sun/Solaris (64-bit)	`compileopts_sol2`
Mac OS X	`compileopts_macosx`
Intel Mac	`compileopts_maci32`

The configuration file contains specifications of the compiler, paths, and options described using the format of Unix shell files: A line of the form `VAR=VALUE` assigns the value `VALUE` to the variable `VAR`. The following variables can be used:

TABLE 14-11: CONFIGURATION FILE VARIABLES

VARIABLE	INTERPRETATION
`BINPATH`	Search path for compiler, linker, and `SETUPCMD` (see below)
`CC`	C compiler
`CFLAGS`	Default compiler flags

TABLE 14-11: CONFIGURATION FILE VARIABLES

VARIABLE	INTERPRETATION
DEBUGFLAGS	Extra compiler flags that makes the C compiler generate debug information
INCLUDEPATHFLAG	Compiler flag used to set the include path
LD	Linker
LDFLAGS	Linker flags
LDLIBS	Linker flag used to set the library path
LINKFLAG	Linker flag that precedes a library linked against
OBJEXT	File extension of object files
OPTFLAGS	Extra compiler flags that makes the C compiler generated optimized object code
SETUPCMD	Command executed before compilation starts

The easiest way to create a configuration file is to copy the default configuration file from the platform at hand and make the necessary modifications.

Compilation Options

The following options are accepted by the `compile` function:

TABLE 14-12: COMPILATION OPTIONS

OPTION	FUNCTION
-c	The source code files are compiled but not linked
-DSYMBOL	Defines the preprocessor macro SYMBOL when compiling. Equivalent to inserting #define SYMBOL in the source code files
-DSYMBOL=VALUE	Assigns the value VALUE to the preprocessor macro SYMBOL. Equivalent to inserting #define SYMBOL VALUE in the source code files
-fFILE	Compilation options are read from FILE
-g	Debug information is generated by the compiler
-h, -help	Displays a help text
-IDIR	Adds the directory DIR to the include file path
-LDIR	Adds the directory DIR to the link directory path
-lLIB	Adds the library LIB to the list of libraries to link against
-oOUTLIB	Sets the name of the generated shared library to OUTLIB
-O	Enables optimization

Compiling from Within a Project or Makefile

The shared library containing the implementation of the external function need not be compiled from within COMSOL Script. For a library containing several source-code files a project or makefile is typically used. The following changes to the compilation environment are necessary for compiling external code:

- The directory `COMSOLDIR/script/external` must be added to the include path.
- The directory `COMSOLDIR/lib/PLATFORM` must be added to the link path, and the library `flscriptext` must be linked against. Here `COMSOLDIR` is the directory where COMSOL Script was installed, and `PLATFORM` is one of the platform abbreviations listed in Table 14-10, for example, `win32` or `glnx86`.

Other Languages

Fortran

Fortran code can be interfaced by writing a small C wrapper on top of the Fortran code. The wrapper declares an entry point, converts function arguments, and calls a function written in the Fortran library.

C++

C++ code can be interfaced without a C wrapper as long as the entry point is declared as `extern "C"`:

```
extern "C"
CL_EXPORT void work(clEnv *env, int nOut, clData *out[],
                    int nIn, clData *in[]);
```

C++ functions not tagged with `extern "C"` cannot be called because name mangling and calling conventions differ between C and C++.

A potential problem when interfacing C++ code is symbol collisions between libraries needed by the external code and libraries needed by COMSOL Script. This can happen if the external code and COMSOL's code are compiled with different compilers. The following compiler versions were used to compile COMSOL's code:

TABLE 14-13: C++ COMPILERS USED BY COMSOL

PLATFORM	COMPILER
PC/Windows (32-bit)	Visual Studio 2005
PC/Windows (64-bit)	Visual Studio 2005
PC/Linux (32-bit)	Intel 9.1 with GCC 3.4.4
PC/Linux (64-bit)	Intel 9.1 with GCC 3.4.4
Itanium/Linux	Intel 9.1 with GCC 3.4.4
Sun/Solaris (32-bit)	Sun Studio 8.0
Sun/Solaris (64-bit)	Sun Studio 11.0
Mac OS X	GCC 3.3
Intel Mac	GCC 4.0.1

INDEX