APPENDIX

B MathScript RT Module Basics

B.1 INTRODUCTION

LabVIEW is short for Laboratory Virtual Instrument Engineering Workbench. It is a flexible graphical development environment from National Instruments, Inc. Engineers and scientists in research, development, production, test, and service industries as diverse as automotive, semiconductor, aerospace, electronics, chemical, telecommunications, and pharmaceutical use LabVIEW, especially in the area of testing and measurements, industrial automation, and data analysis. Users of LabVIEW are familiar with the use of the graphical programming language to create programs relying on graphic symbols to describe programming actions. LabVIEW MathScript RT Module provides a text-based command line environment using m-files and command line prompts. It is assumed here that the reader has LabVIEW 2009 installed and knows how to access the LabVIEW Getting Started window. This appendix only provides an introduction to MathScript RT Module. Readers should refer to *Learning with LabVIEW 2009*[1] for a more complete introduction to LabVIEW and MathScript.

In this appendix, we discuss the MathScript Interactive Window. The essentials of creating user-defined functions and scripts, of saving and loading data files are presented. With the MathScript Interactive Window, students will be able to interact with LabVIEW through a command prompt.

B.2 WHAT IS MATHSCRIPT?

MathScript RT Module provides access to a text-based math-oriented language with a command prompt from within the LabVIEW development environment. MathScript RT Module does not require additional third-party software to compile and execute, but it is a separate download from LabVIEW. MathScript RT Module includes hundreds of built-in functions. There are linear algebra functions, curve-fitting functions, digital filters, functions for solving differential equations, and probability and statistics functions. And since MathScript RT Module employs a commonly used syntax, it follows that you can work with many of your previously developed mathematical computation scripts, or any of those openly available in engineering textbooks or on the internet. The fundamental math-oriented data types are matrices with built-in operators for generating data and accessing elements. In addition, you can extend the application with custom user-defined functions. An overview of the features of MathScript RT Module is presented in Table B.1.

[1]Bishop, R. H., *Learning with LabVIEW 2009*, Pearson Education, 2010.

Table B.1 Overview of the Features of MathScript RT Module

MathScript Feature	Description
Powerful textual math	MathScript RT Module includes more than 800 functions for math, signal processing, and analysis; functions cover areas such as linear algebra, curve fitting, digital filters, differential equations, probability/statistics, and much more.
Math-oriented data types	MathScript RT Module uses matrices as fundamental data types, with built-in operators for generating data, accessing elements, and other operations.
Data type highlighting	MathScript RT Module analyzes m-files during editing for enhanced script debugging and readability to determine the data types of inputs and constants.
Compatible	MathScript RT Module can process certain files utilizing other text-based syntaxes. However, it does not support all m-file script syntaxes; hence not all existing text-based scripts are compatible.
Extensible	You can extend MathScript RT Module by defining your own custom functions.
Part of LabVIEW	MathScript RT Module does not require additional third-party software to compile and execute.

You can find more information on MathScript at http://ni.com/mathscript including lists of built-in MathScript functions and links to online examples.

B.3 ACCESSING THE MATHSCRIPT INTERACTIVE WINDOW

The **MathScript Interactive Window** provides a user interface in which you can enter and execute commands and see the result after the commands complete. You can enter commands one-at-a-time through a command line or as a group through a simple text editor. You can access the interactive window from the **Getting Started** screen or any VI by selecting **Tools»MathScript Window**…, as illustrated in Figure B.1. The MathScript Interactive Window is a user interface comprised of a **Command Window** (the user command inputs), an **Output Window** echoing the inputs and showing the resulting outputs, a **Script Editor** window (for loading, saving, compiling, and running scripts), a **Variables** window (showing variables, dimensions, and type), and a **Command History** window providing a historical account of commands. A MathScript Interactive Window is shown in Figure B.2 with the various components highlighted.

As you work, the Output Window updates to show your inputs and the subsequent results. The Command History window tracks your commands. The history view is very useful because there you can access and reuse your previously executed commands by double clicking a previous command to execute it again. You can also navigate up and down through the previous commands (which will appear in the Command Window) by using the "↑" and "↓" keys. In the Script Editor window, you can enter and execute groups of commands and then save the commands in a file (called a script) for use in a later LabVIEW session.

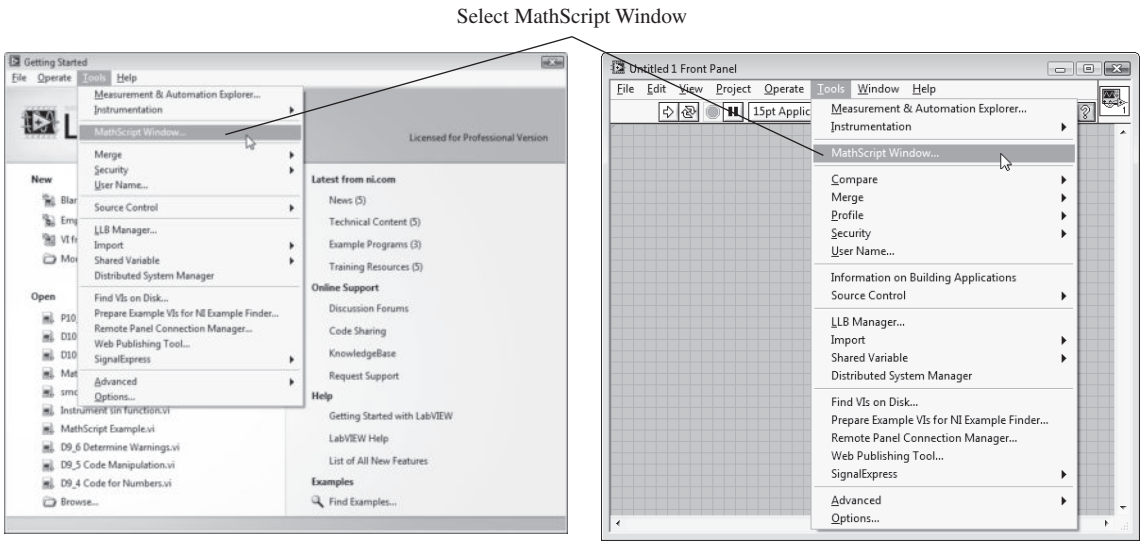


FIGURE B.1 Accessing the MathScript Interactive Window from (a) the Getting Started screen, or (b) the **Tools** pull-down menu on the front panel or block diagram.

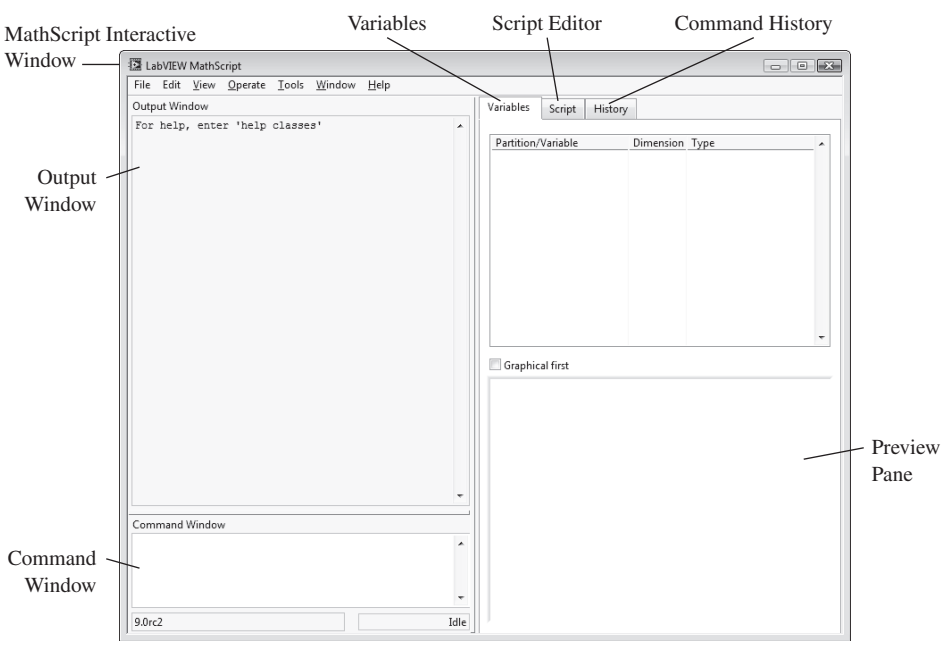


FIGURE B.2 The basic components of the MathScript Interactive Window.

The Command History Window

The commands entered in previous sessions using the MathScript Interactive Window will reappear in subsequent sessions. In the Command History window you will find a header that shows the day and time that you entered the commands. This feature allows you to easily discern when the commands were entered.

Clearing the Output Window

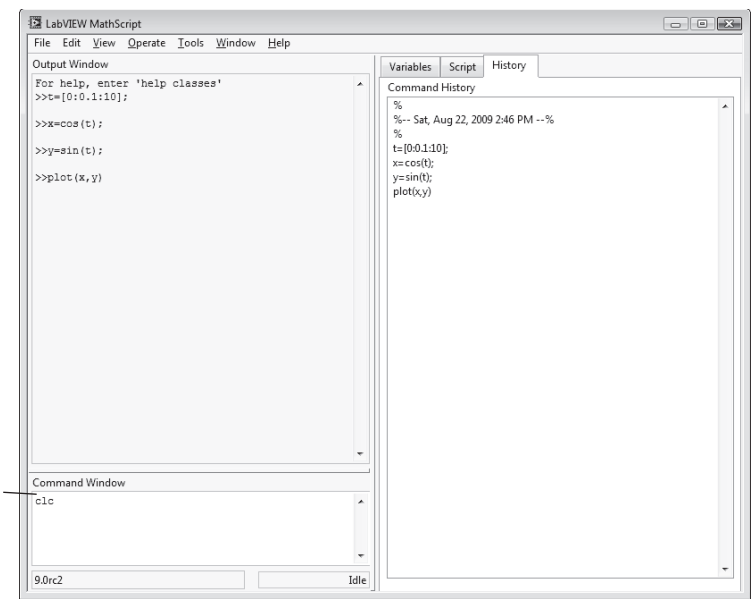
You can clear the Output Window by typing in clc in the Command Window, then pressing <Enter>, as illustrated in Figure B.3.

Copying Output Window Data

You can copy data from the Output Window and paste it in the Script Editor window or a text editor by highlighting the desired text in the Output Window and selecting **Edit**»**Copy** or pressing the <Ctrl-C> keys to copy the selected text to the clipboard.

Viewing Data in a Variety of Formats

In the MathScript Interactive Window you can view the variables in a variety of formats, as shown in Figure B.4. Table B.2 shows the available data types in MathScript. Depending on the variable type, the available formats include: numeric, string, graph, XY graph, sound, surface, and picture. You can edit a variable in the **Preview Pane** when the display type is **Numeric** or **String**. Selecting **Sound** plays the data as a sound, but works for one-dimensional variables only. The remaining display types show the data as graphs of one sort or another: **Graph** displays the data on a waveform graph, **XY Graph** displays the data on an XY graph, **Surface** displays the data



Use the clc function in the Command Window

FIGURE B.3 Clearing the Command History and the Output Window.

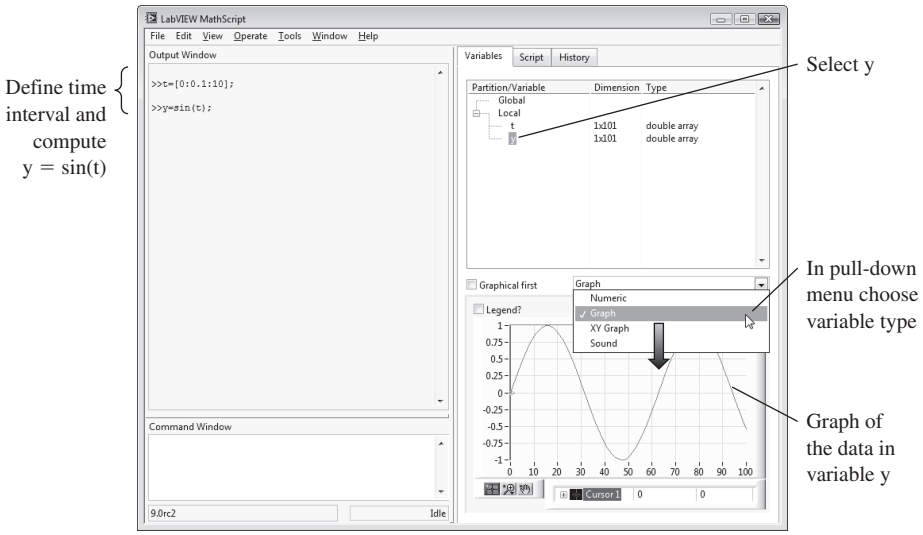


FIGURE B.4
Showing the data type in various formats.

on a 3D surface graph, and **Picture** displays the data on an intensity graph. The **Graphical first** button specifies whether to display first the numerical or graphical representation of variables in the Preview Pane. You can view only the numerical representation of scalar variables.

Open a new MathScript Interactive Window from the Getting Started screen by selecting **Tools»MathScript Window**…, as illustrated in Figure B.1. In the Command Window input the time from $t = 0$ seconds to $t = 10$ seconds in increments of 0.1 seconds, as follows:

$$t = [0:0.1:10];$$

Then, compute the $y = \cos(t)$ as follows:

$$y = \cos(t);$$

Notice that in the Variables window the two variables t and y appear, as illustrated in Figure B.5(a). Select the variable y and note that in the Preview Pane the variable appears in the numeric format. Now, in the pull-down menu above the Preview Pane, select **Graph**. The data will now be shown in graphical form, as illustrated in Figure B.5(b). The graph can be undocked from the Preview Pane for re-sizing and customization. To undock the graph, right-click on the graph and select **Undock Window**. The window can now be re-sized and the plot can be customized interactively and printed.

As an alternative to using the Preview Pane, you can also obtain a plot of the y versus t programmatically using the **plot** command:

```
plot(t,y)
```

The process is illustrated in Figure B.6. A new window appears that presents the graph of y versus t. Following the same procedure, see if you can obtain a plot of $y = \cos(\omega t)$ where $\omega = 4$ rad/sec.

Table B.2 Overview of MathScript Data Types and Relationship to Syntax

	MathScript Syntax			
Data Type	**Scalar**	**1D-Array**	**2D-Array**	**Matrix**
Unsigned integer numeric				
8-bit	Scalar≫U8	1D-Array≫U8 1D	2D-Array≫U8 2D	
16-bit	Scalar≫U16	1D-Array≫U16 1D	2D-Array≫U16 2D	
32-bit	Scalar≫U32	1D-Array≫U32 1D	2D-Array≫U32 2D	
64-bit	Scalar≫U64	1D-Array≫U64 1D	2D-Array≫U64 2D	
Signed integer numeric				
8-bit	Scalar≫I8	1D-Array≫I8 1D	2D-Array≫I8 2D	
16-bit	Scalar≫I16	1D-Array≫I16 1D	2D-Array≫I16 2D	
32-bit	Scalar≫I32	1D-Array≫I32 1D	2D-Array≫I32 2D	
64-bit	Scalar≫I64	1D-Array≫I64 1D	2D-Array≫I64 2D	
Single-precision, floating-point numeric	Scalar≫SGL	1D-Array≫SGL 1D	2D-Array≫SGL 2D	
Double-precision, floating-point numeric	Scalar≫DBL	1D-Array≫DBL 1D	2D-Array≫DBL 2D	
Extended-precision, floating-point numeric	Scalar≫EXT	1D-Array≫EXT 1D	2D-Array≫EXT 2D	
Complex single-precision, floating-point numeric	Scalar≫CSG	1D-Array≫CSG 1D	2D-Array≫CSG 2D	
Complex double-precision, floating-point numeric	Scalar≫CDB	1D-Array≫CDB 1D	2D-Array≫CDB 2D	
Complex extended-precision, floating-point numeric	Scalar≫CXT	1D-Array≫CXT 1D	2D-Array≫CXT 2D	
Boolean	Scalar≫Boolean	1D-Array≫Boolean 1D	2D-Array≫Boolean 2D	
String	Scalar≫String	2D-Array≫Boolean 2D		
Matrix				
Real				Matrix≫Real Matrix
Complex				Matrix≫Complex Matrix

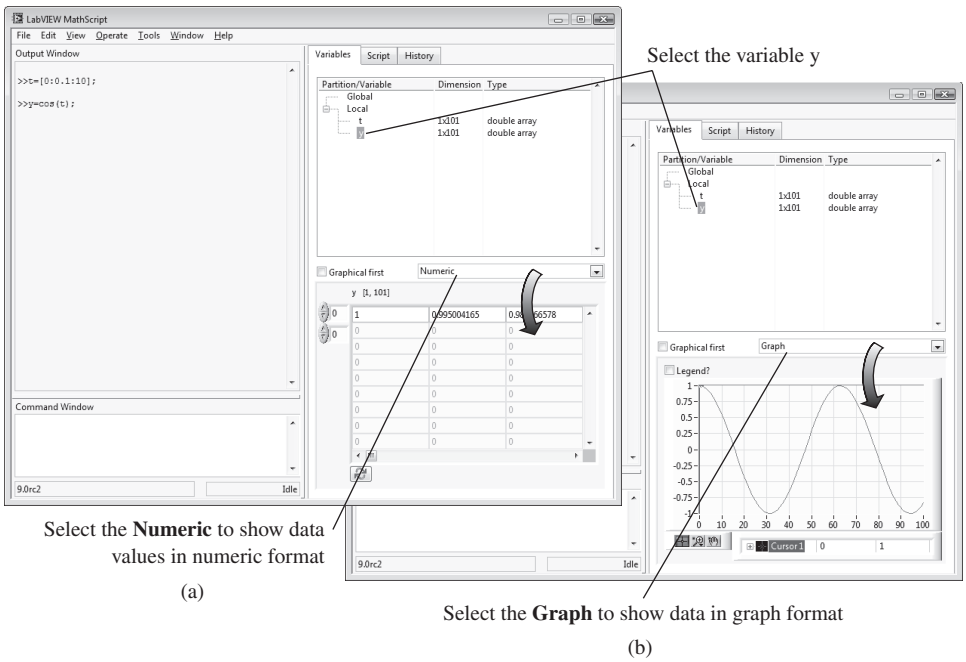


FIGURE B.5 (a) Entering the time, computing $y = \cos(t)$, and viewing the variable y in numerical form. (b) Viewing the variable y in graphical form.

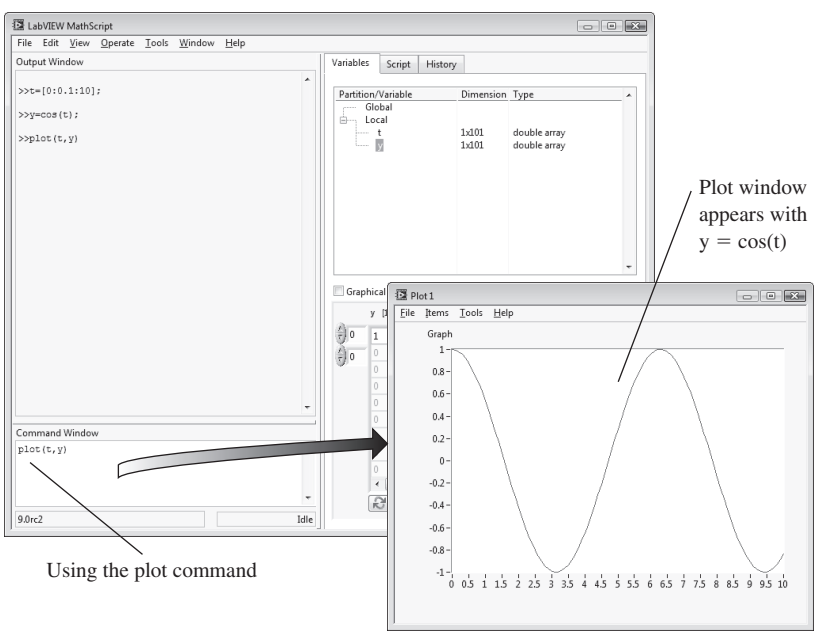


FIGURE B.6 Obtaining a plot of the cosine function using the **plot** command.

B.4 MATHSCRIPT HELP

You can display several types of help content for MathScript by calling different help commands from the Command Window. Table B.3 lists the help commands you can call and the type of help these commands display in the Output Window.

As illustrated in Figure B.7, entering help classes in the Command Window results in an output showing all classes of functions and commands that MathScript supports. Examples of the classes of functions are basic and matrixops. Entering help basic in the Command Window results in a list of the members of the basic

Table B.3 Help Commands for MathScript

Command	Description of Help Provided
help	Provides an overview of the MathScript Window.
help classes	Provides a list of all classes of MathScript functions and topics as well as a short description of each class.
help *class*	Provides a list of the names and short descriptions of all functions in a particular MathScript class. Example: help basic
help *function*	Provides reference help for a particular MathScript function or topic, including its name, syntax, description, inputs and outputs, examples to type in the Command Window, and related functions or topics. Example: help abs

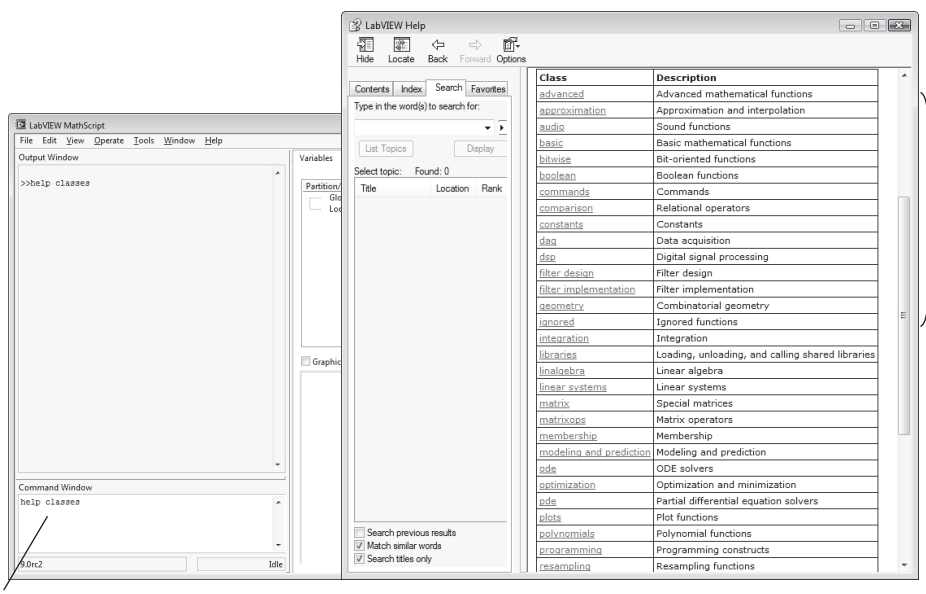


FIGURE B.7 Accessing the help for MathScript classes, members, and functions.

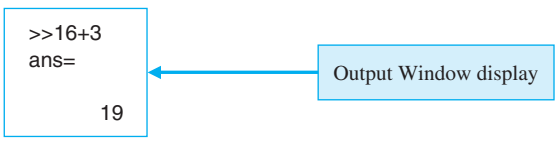
class, including abs (i.e., the absolute value), conj (i.e., the complex conjugate function), and exp (i.e., the exponential function). Then, entering help abs in the Command Window will result in an output that contains a description of the abs function, including examples of its usage and related topics.

B.5 SYNTAX

The syntax associated with MathScript is straightforward. Most students with some experience programming a text-based language will be comfortable with the programming constructs in MathScript. If you need help getting started with MathScript, you can access help by selecting Help»Search the LabVIEW Help from the MathScript Interactive Window and typing mathscript in the search window.

Eleven basic MathScript syntax guidelines are:

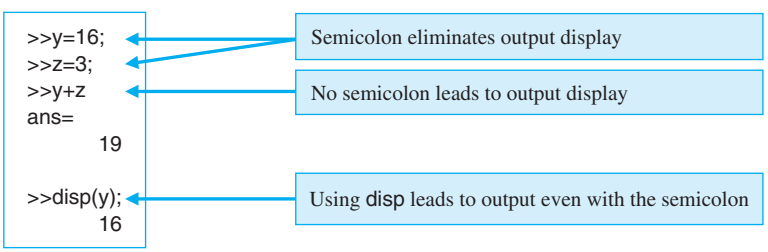
1. **Scalar operations:** MathScript is ideally suited for quick mathematical operations, such as addition, subtraction, multiplication, and division. For example, consider the addition of two scalar numbers, 16 and 3. This is a simple operation that you might perform on a calculator. This can be accomplished using the MathScript command:



In MathScript, if you perform any calculation or function without assigning the result to a variable, the default variable ans is used. If you want to assign the value of the addition of two scalars to the variable *x*, enter the following command:

```
>>x=16+3
x=
     19
```

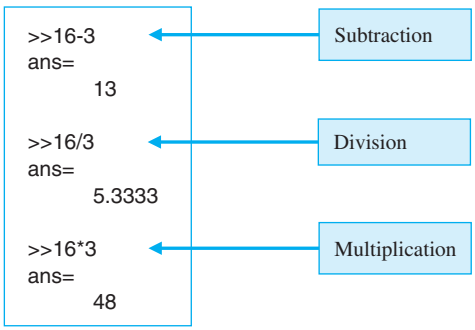
In the same manner, you can add two scalar variables *y* and *z* by entering the following commands:



Notice that in the previous example a semicolon was used for the first two lines, and no output was displayed. In MathScript, if you end a command line with a semicolon, the MathScript Interactive Window does not display the output for that command.

Some functions display output even if you end the command line with a semicolon. For example, the disp function displays an output even if followed by a semicolon.

You use the symbol '-' for subtraction, the symbol '/' for division, and the symbol '*' for multiplication, as illustrated below:



2. **Creating matrices and vectors:** To create row or column vectors and matrices, use white space or commas to separate elements, and use semicolons to separate rows. Consider for example, the matrix **A** (a column vector),

$$\mathbf{A} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

In MathScript syntax, you would form the matrix as

```
A = [1; 2; 3]
```

Consider for example, the matrix **B** (a row vector)

$$\mathbf{B} = [1 \quad -2 \quad 7].$$

In MathScript syntax, you would form the matrix as

```
B = [1, -2, 7] or B = [1 -2 7]
```

As a final example, consider the matrix **C** (a 3×3 matrix)

$$\mathbf{C} = \begin{bmatrix} -1 & 2 & 0 \\ 4 & 10 & -2 \\ 1 & 0 & 6 \end{bmatrix}.$$

In MathScript syntax, you would form the matrix as

```
C = [-1 2 0; 4 10 -2; 1 0 6] or C = [-1, 2, 0; 4, 10, -2; 1, 0, 6]
```

3. **Creating vectors using the colon operator:** There are several ways to create a one-dimensional array of equally spaced elements. For example, you will often need to create a vector of elements representing time. To create a one-dimensional array equally spaced and incremented by 1, use the MathScript syntax

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

To create a one-dimensional array equally spaced and incremented by 0.5, use the MathScript syntax

```
>> t = 1:0.5:10
t =
    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000    5.5000
    6.0000    6.5000    7.0000    7.5000    8.0000    8.5000    9.0000    9.5000   10.0000
```

4. **Accessing individual elements of a vector or matrix:** You may want to access specific elements or subsets of a vector or matrix. Consider the 3×3 matrix **C**:

$$\mathbf{C} = \begin{bmatrix} -1 & 2 & 0 \\ 4 & 10 & -2 \\ 1 & 0 & 6 \end{bmatrix}.$$

In MathScript syntax, you can access the element in the second row and third column of the matrix **C**, as follows:

```
>>C = [-1 2 0;4 10 -2;1 0 6]
C=
  -1    2    0
   4   10   -2
   1    0    6

>>C(2,3)
ans=

   -2
```

C(2,3) denotes the second row of the third column

You can assign this value to a new variable by entering the following command:

```
>>F=C(2,3)
F=
    -2
```

You can also access an entire row or an entire column of a matrix using the colon operator. In MathScript syntax, if you want to access the entire second row of matrix **C**, enter the following command:

```
>>C(2,:)
ans=

     4    10    -2
```

The second row of **C**

In the same way, if you wish to access the entire third column of matrix **C**, enter the following command:

```
>>C(:,3)
ans=

     0
    -2
     6
```

The third column of **C**

Suppose you want to extract the 2×2 submatrix from **C** consisting of rows 2 and 3 and columns 1 and 2. You use brackets to specify groups of rows and columns to access a subset of data as follows:

```
>>C([2 3], [1 2])
ans=

     4    10
     1     0
```

A submatrix of **C**

5. **Calling functions in MathScript:** You can call MathScript functions from the Command Window. Consider the creation of a vector of a certain number of elements that are equally distributed in a given interval. To accomplish this in MathScript syntax, you can use the built-in function linramp. Using the command help linramp you find that this function uses the syntax

linramp(a, b, n)

where a specifies the start of the interval, b specifies the end of the interval, and n identifies the number of elements. Thus, to create a vector of $n = 13$ numbers equally distributed between $a = 1$ and $b = 10$, use the following command:

```
>>G = linramp (1, 10, 13)
G=
    1     1.75     2.5     3.25     4     4.75     5.5     6.25     7
 7.75     8.5     9.25     10
```

If you do not specify a value for n, the linramp command will automatically return a vector of 100 elements. To select a subset of **G** that consists of all elements after a specified index location, you can use the syntax described in guideline 4 and the end function to specify the end of the vector. For example, the following command will return all elements of **G** from the fifth element to the final element:

```
>>H=G(5: end)
H =
  4     4.75     5.5     6.25     7     7.75     8.5     9.25     10
```

The function linramp is an example of a built-in MathScript function. Calling user-defined functions are discussed further in Section B.6.

6. **Assigning data types to variables:** MathScript variables adapt to data types. For example, if

```
a = sin(3*pi/2)
```

then *a* is a double-precision floating-point number. If

```
a = 'temperature'
```

then *a* is a string.

7. **Using complex numbers:** You can use either *i* or *j* to represent the imaginary unit equal to the square root of −1. If you assign values to either *i* or *j* in your scripts, then those variable names are no longer complex numbers. For example, if you let $y = 4 + j$, then y is a complex number with real part equal to 4 and imaginary part equal to $+j$. If however, you assign $j = 3$, and then compute $y = 4 + j$, the result is $y = 7$, a real number.

8. **Matrix operations:** Many of the same mathematical functions used on scalars can also be applied to matrices and vectors. Consider adding two matrices **K** and **L**, where

$$\mathbf{K} = \begin{bmatrix} -1 & 2 & 0 \\ 4 & 10 & -2 \\ 1 & 0 & 6 \end{bmatrix} \text{ and } \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

To add the two matrices **K** and **L**, element by element, enter the following MathScript command:

```
>>K+L
ans =
   0     2     0
   4    11    -2
   1     0     7
```

In a similar fashion, you can also multiply two matrices **K** and **L**, as follows:

```
>>K*L
ans =
  -1     2     0
   4    10    -2
   1     0     6
```

Consider the 3 × 1 matrix **M** (column vector) and the 1 × 3 matrix **N** (row vector)

$$\mathbf{M} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \text{ and } \mathbf{N} = [0 \quad 1 \quad 2].$$

Then, the product **M** ∗ **N** is the 3 × 3 matrix

```
>>M*N
ans =
   0     1     2
   0     2     4
   0     3     6
```

and the product **N** ∗ **M** is a scalar

```
>>N*M
ans =
      8
```

To multiply two matrices, they must be of compatible dimensions. For example, suppose a matrix **M** is of dimension $m \times n$, and a second matrix **N** is of dimension $n \times p$. Then you can multiply **M** × **N** resulting in an $m \times p$ matrix. In the example above, the 3 × 1 matrix **M** (column vector) was multiplied with the 1 × 3 matrix **N** (row vector) resulting in a 3 × 3 matrix. You cannot multiply **N** × **M** unless $m = p$. In the example above, the 1 × 3 matrix **N** (row vector) was multiplied with the 3 × 1 matrix **M** (column vector) resulting in a 1 × 1 matrix (a scalar), so in this case, $m = p = 1$.

When working with vectors and matrices in MathScript, it is often useful to perform mathematical operations element-wise. For example, consider the two vectors

$$\mathbf{M} = \begin{bmatrix} -1 \\ 4 \\ 0 \end{bmatrix} \text{ and } \mathbf{N} = \begin{bmatrix} 2 \\ -2 \\ 1 \end{bmatrix}.$$

In light of our previous discussion, it is not possible to compute **M** ∗ **N** since the dimensions are not compatible. However, you can multiply the vectors element-wise using the syntax '.*' for the multiplication operator, as follows:

$$\mathbf{M}.*\mathbf{N} = \begin{bmatrix} -1*2 \\ 4*(-2) \\ 0*1 \end{bmatrix} = \begin{bmatrix} -2 \\ -8 \\ 0 \end{bmatrix}.$$

By definition, matrix addition and subtraction occurs element-wise. However, it is also possible to perform division element-wise. For **M** and **N** as above, we find that element-wise division yields

$$\mathbf{M}./\mathbf{N} = \begin{bmatrix} -1/2 \\ 4/(-2) \\ 0/1 \end{bmatrix} = \begin{bmatrix} -0.5 \\ -2 \\ 0 \end{bmatrix}.$$

Element-wise operations are useful in plotting functions. For example, suppose that you wanted to plot $y = t\sin(t)$ for $t = [0:0.1:10]$. This would be achieved via the commands

```
t=[0:0.1:10];
y=t .* sin(t);
plot(t,y)
```

Element-wise multiplication using .*

9. **Logical expressions:** MathScript can evaluate logical expressions such as EQUAL, NOT EQUAL, AND, and OR. To perform an equality comparison, use the statement

```
a == b
```

If *a* is equal to *b*, MathScript will return a 1 (indicating True); if *a* and *b* are not equal, MathScript will return a 0 (indicating False). To perform an inequality comparison, use the statement

```
a ~= b
```

If *a* is not equal to *b*, MathScript will return a 1 (indicating True); if *a* and *b* are equal, MathScript will return a 0 (indicating False).

In other scenarios, you may want to use MathScript to evaluate compound logical expressions, such as when at least one expression of many is True (OR), or when all of your expressions are True (AND). The compound logical expression AND is executed using the '&' command. The compound logical expression OR is executed using the '|' command.

10. **Control flow constructs:** Table B.4 provides the MathScript syntax for commonly used programming constructs.
11. **Adding comments:** To add comments to your scripts, precede each line of documentation with a % character. For example, consider a script that has two inputs, *x* and *y*, and computes the addition of *x* and *y* as the output variable *z*.

```
% In this script, the inputs are x and y
% and the output is z.
% z is the addition of x and y
z = x + y;
```

} Comments

The script shown above has three comments, all preceded by the % character. In the next section, we will discuss more details on how to use comments to provide help documentation.

Some considerations that have a bearing on your usage of MathScript follow:

1. You cannot define variables that begin with an underscore, white space, or digit. For example, you can name a variable *time*, but you cannot name a variable *4time* or *_time*.
2. MathScript variables are case sensitive. The variables *X* and *x* are not the same variables.
3. LabVIEW MathScript does not support *n*-dimensional arrays where $n > 2$, cell arrays, sparse matrices, or structures.

Key MathScript Functions

MathScript offers more than 800 textual functions for math, signal processing, and analysis. These are in addition to the more than 600 graphical functions for signal processing, analysis, and math that are available as VIs within LabVIEW. Table B.5 lists many of the key areas with supporting MathScript functions. For a comprehensive function list, visit the National Instruments website at http://www.ni.com/mathscript or see the online help.

Table B.4 MathScript Syntax for Commonly Used Constructs

Construct	Grammar	Example

Case-Switch Statement	**switch** expression
	case expression
	statement-list
	[**case**
	expression
	statement-list]
	...
	[**otherwise**
	statement-list]
	end
		switch mode
		case 'start'
		a = 0;
		case 'end'
		a = −1;
		otherwise
		a = a + 1;
		end
		When a case in a case-switch statement executes, LabVIEW does not select the next case automatically.Therefore, you do not need to use break statements as in C.

For Loop	**for** expression statement-list **end**	for k = 1:10 a = sin(2*pi*k/10) end
If-Else Statement	**if** expression statement-list [**elseif** expression statement-list] ... [**else** statement-list] **end**	if b == 1 c = 3 else c = 4 end
While Loop	**while** expression statement-list **end**	while k < 10 a = cos(2*pi*k/10) k = k + 1; end

B.6 DEFINING FUNCTIONS AND CREATING SCRIPTS

You can define functions and create scripts to use in the MathScript Interactive Window. Functions and scripts can be created in the Script Editor window on the MathScript Interactive Window (see Figure B.2). You can also use your favorite text editor to create functions and scripts. Once your function or script is complete, you should save it for later use. The filename for a function must be the same as the name of the function and must have a lowercase .m extension. For example, the filename for a user-defined starlight[2] function must be starlight.m. Use unique names for all functions and scripts and save them in a directory that you specified in the **Path** section of the **File**»**MathScript Preferences** dialog box.

User-Defined Functions

MathScript offers more than 800 textual functions for math, signal processing, and analysis. But what if you have a special purpose function that you want to add to

[2]The name starlight does not represent a real function. It is used here for illustrative purposes only.

Table B.5 MathScript Function Classes

Function Classes	Brief Description
Control Design and Analysis	Classical and state-space control design and analysis functions. Dynamic characteristics, root locus, frequency response, Bode, Nyquist, Nichols, model construction, connection, reduction, and more.
Plots (2D and 3D)	Standard *x-y* plot; mesh plot; 3D plot; surface plot; subplots; stairstep plot; logarithmic plots; stem plot and more.
Digital Signal Processing (DSP)	Signal synthesis; Butterworth, Chebyshev, Parks-McClellan, windowed FIR, elliptic (Cauer), lattice and other filter designs; FFT (1D/2D); inverse FFT (1D/2D); Hilbert transform; Hamming, Hanning, Kaiser-Bessel, and other windows; pole/zero plotting and others.
Approximation (Curve Fitting & Interpolation)	Cubic spline, cubic Hermite and linear interpolation; exponential, linear and power fit; rational approximation and others.
Ordinary Differential Equation (ODE) Solvers	Adams-Moulton, Runge-Kutta, Rosenbrock and other continuous ordinary differential equation (ODE) solvers.
Polynomial Operations	Convolution; deconvolution; polynomial fit; piecewise polynomial; partial fraction expansion and others.
Linear Algebra	LU, QR, QZ, Cholesky, Schur decomposition; SVD; determinant; inverse; transpose; orthogonalization; solutions to special matrices; Taylor series; real and complex eigenvalues and eigenvectors; polynomial eigenvalue and more.
Matrix Operations	Hankel, Hilbert, Rosser, Vandermonde special matrices; inverse; multiplication; division; unary operations and others.
Vector Operations	Cross product; curl and angular velocity; gradient; Kronecker tensor product and more.
Probability and Statistics	Mean; median; Poisson, Rayleigh, chi-squared, Weibull, T, gamma distributions; covariance; variance; standard deviation; cross correlation; histogram; numerous types of white noise distributions and other functions.
Optimization	Quasi-Newton, quadratic, Simplex methods and more.
Advanced Functions	Bessel, spherical Bessel, Psi, Airy, Legendre, Jacobi functions; trapezoidal, elliptic exponential integral functions and more.
Basic	Absolute value; Cartesian to polar and spherical and other coordinate conversions; least common multiple; modulo; exponentials; logarithmic functions; complex conjugates and more.
Trigonometric	Standard cosine, sine and tangent; inverse hyperbolic cosine, cotangent, cosecant, secant, sine and tangent; hyperbolic cosine cotangent, cosecant, secant, sine and tangent; exponential; natural logarithm and more.
Boolean and Bit Operations	AND, OR, NOT and other logic operations; bitwise shift, bitwise OR and other bitwise operations.
Data Acquisition/Generation	Perform analog and digital I/O using National Instruments devices.
Other	Programming primitives such as if, for and while loops; unsigned and signed datatype conversions; file I/O; benchmarking and other timing functions; various set and string operations and more.

your personal library? This function may be particular to your area of study or research, and is one that you need to call as part of a larger program. With MathScript it is simple to create a function once you understand the basic syntax.

A MathScript function definition must use the following syntax:

```
function outputs = function_name(inputs)
% documentation
script
```

An example of a user-defined function definition utilizing the proper syntax is

```
function ave = compute_average(x, y)
% compute_average determines the average of the two inputs x and y.
ave = (x + y)/2;
```

Begin each function definition with the term **function**. The *outputs* lists the output variables of the function. If the function has more than one output variable, enclose the variables in square brackets and separate the variables with white space or commas. The *function_name* is the name of the function you want to define and is the name that you use when calling the function. The *inputs* lists the input variables to the function. Use commas to separate the input variables. The *documentation* is the set of comments that you want MathScript to return for the function when you execute the help command. Comments are preceded with a % character. You can place comments anywhere in the function; however, LabVIEW returns only the first comment block in the Output Window to provide the help to the user. All other comment blocks are for internal documentation. The *script* defines the executable body of the function.

Checking the help on the function compute_average.m and then executing the function with $x = 2$ and $y = 4$ as inputs yields

```
»help compute_average
compute_average determines the average of the two inputs x and y.
»x = 2; y = 4; compute_average(x, y)
ans =
     3
```

Note that there is a MathScript function named mean that can also be used to compute the average of two inputs, as follows:

```
»mean([2 4])
ans =
      3
```

Functions can be edited in the Script Editor window and saved for later use. In Figure B.8, the buttons **Load**, **Save As**, **New Script**, and **Run Script** are shown. Selecting **Load** will open a window to browse for the, desired function (or script) to

load into MathScript. Similarly, selecting **Save As** will open a browser to navigate to the desired folder to save the function.

In Figure B.8, the function compute_average is used to compute the average of two arrays. Notice that the function computes the average element-wise. If compute_average had inadvertently been named mean, then LabVIEW would execute the user-defined function instead of the built-in function. Generally it is not a good idea to redefine LabVIEW functions, and students should avoid doing so. If you define a function with the same name as a built-in MathScript function, LabVIEW executes the function you defined instead of the original MathScript function. When you execute the help command, LabVIEW returns help content for the function you defined and not the help content for the original MathScript function.

Other examples of valid function syntax for the starlight function include:

```
function starlight                    % No inputs and no outputs
function a = starlight                % No inputs and one output
function [a b] = starlight            % No inputs and two outputs
function starlight (g)                % One input and no outputs
function a = starlight (g)            % One input and one output
function [a b] = starlight (g)        % One input and two outputs
function starlight (g, h)             % No inputs and two outputs
function a = starlight (g, h)         % One input and two outputs
function [a b] = starlight (g, h)     % Two inputs and two outputs
```

There are several restrictions on the use of functions. First, if you define multiple functions in one MathScript file, all functions following the first are **subfunctions** and are accessible only to the main function. A function can call only those functions that you define below it. Second, you cannot call functions recursively. For example, the function starlight cannot call starlight. And third, LabVIEW also does not allow circular recursive function calls. For example, the function starlight cannot call the function bar if bar calls starlight.

Scripts

A script is a sequence of MathScript commands that you want to perform to accomplish a task. For convenience and reusability, once you have created a script, you can save it and load it into another session of LabVIEW at a later time. Also, often you can use a script designed for a different task as a starting point for the development of a new script. Since the scripts themselves are saved as common ascii text and editable with any text editor (including the one found in the MathScript Interactive Window), it is easy to do this. The MathScript functions as well as the user-defined functions can be employed in scripts.

Continuing the example above, suppose that we used a script to compute the average of two numbers. The compute_average function could be used within the script. Once saved, the script can subsequently be loaded into MathScript for use in another session. A script using the compute_average function is shown in Figure B.9.

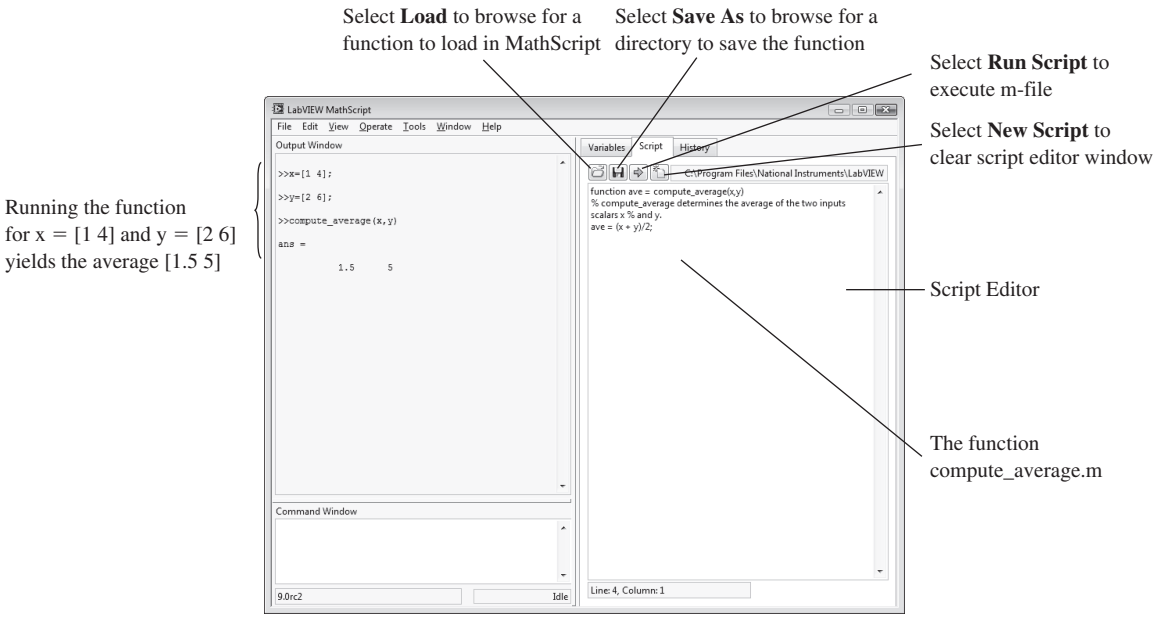


FIGURE B.8 Loading and saving functions.

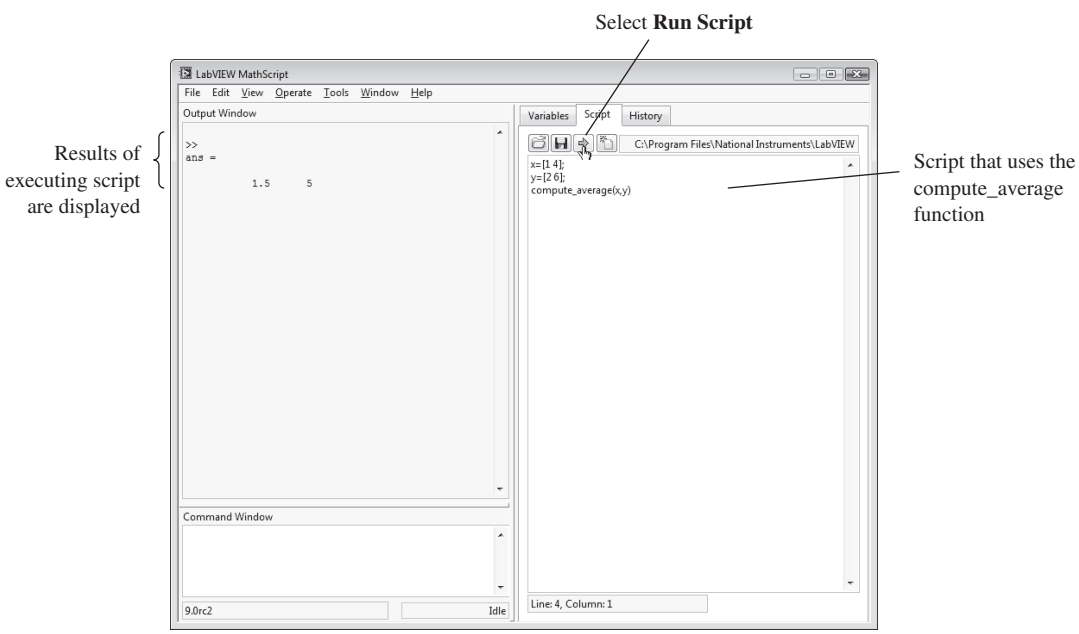


FIGURE B.9 Editing, saving, and running a script to compute the average of two arrays element-wise.

B.7 SAVING AND LOADING SCRIPTS

Being able to save scripts is an important feature giving you the capability to develop a library of scripts that you can readily access in future MathScript sessions. To save a script that you have created in the Script Editor window, select **File**»**Save**, as illustrated in Figure B.10(a). You can also save your script by clicking the **Save As** button on the Script Editor window of the MathScript Interactive Window, as illustrated in Figure B.10(b). In both cases, a file dialog box will appear for you to navigate to the directory in which you want to save the script. Enter a name for the script in the **File name** field. The name must have a lowercase .m extension if you want LabVIEW to run the script (in this example, we use the name average_example.m). Click the **OK** button to save the script.

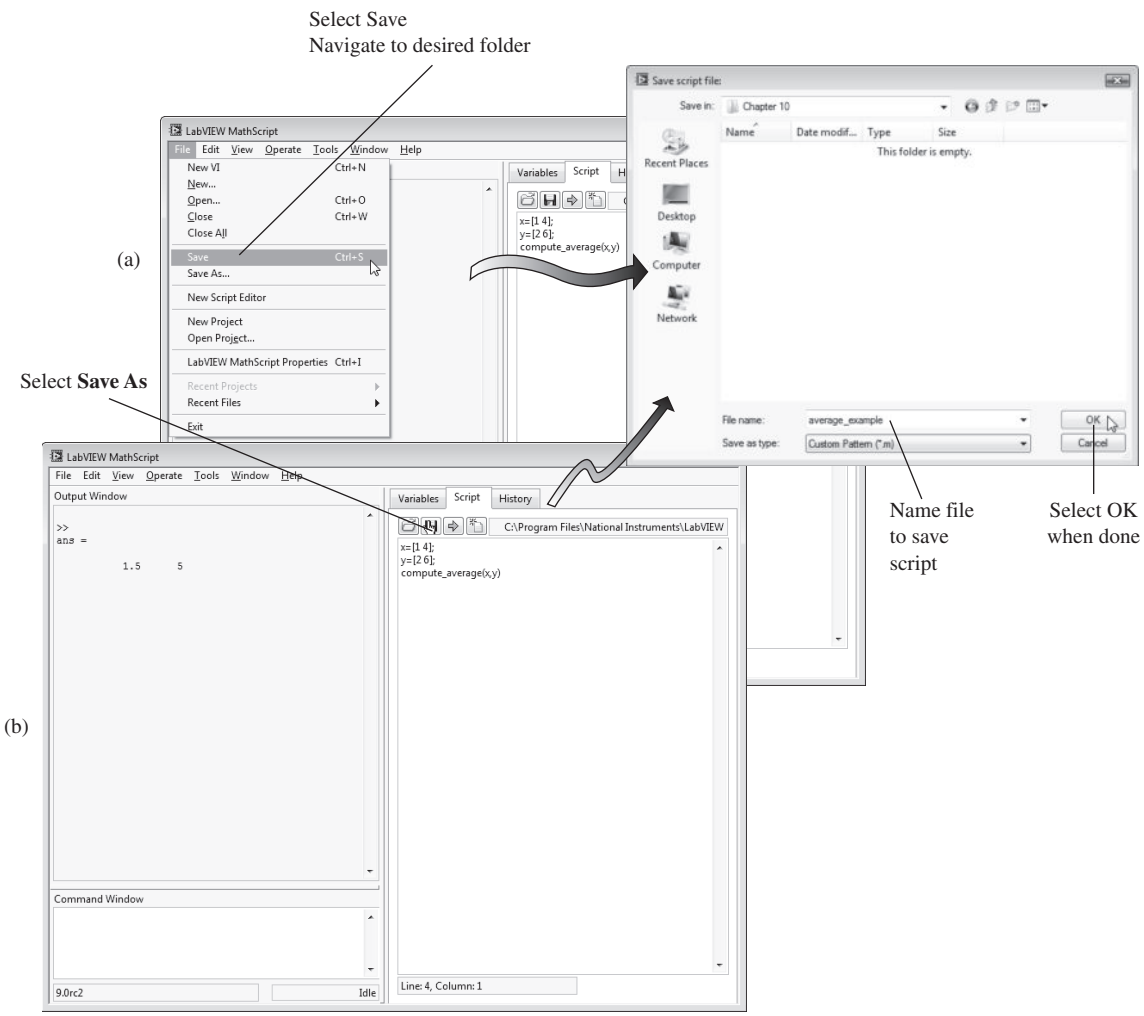


FIGURE B.10 (a) Saving a script using the **File**»**Save** pull-down menu. (b) Saving a script using the **Save As** button on the MathScript Interactive Window.

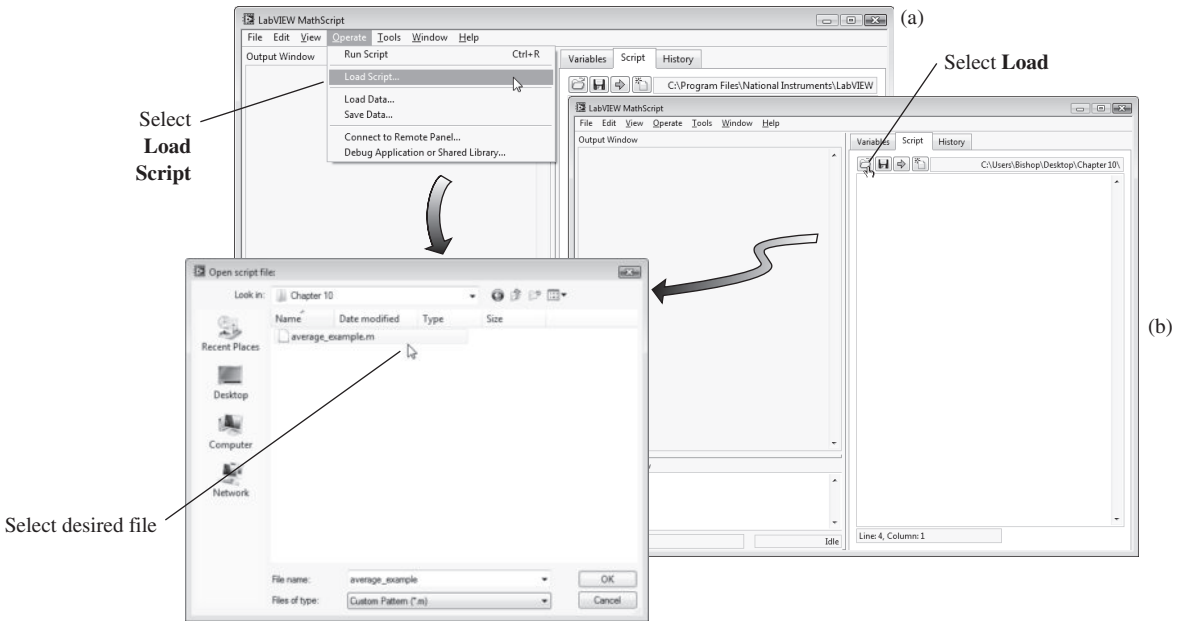


FIGURE B.11 (a) Loading a script using the **Operate**»**Load Script** pull-down menu. (b) Loading a script using the **Load** button on the MathScript Interactive Window.

You can load existing scripts into the MathScript Interactive Window. This will be useful upon returning to a MathScript session or if you want to use a script in the current session that was developed in a previous session. To load an existing script, select **Operate**»**Load Script** or click the **Load** button on the **Script Editor** on the MathScript Interactive Window.

Figure B.11 illustrates the process of loading scripts. In the example, the script compute_average.m is loaded into a MathScript session, and then using the **Run Script** button, the script is executed.

B.8 SAVING AND LOADING DATA FILES

In MathScript you can save and load data files in the MathScript Interactive Window. A data file contains numerical values for variables. Being able to save and load data gives you the flexibility to save important data output from a MathScript session for use in external programs. There are two ways to save data files. The first method saves the data for *all* the variables in the workspace, and the second method allows you to select the variables to save to a file.

To save all the variables in the workspace, select **Operate**»**Save Data** in the MathScript Interactive Window. In the file dialog box, navigate to the directory in which you want to save the data file. Enter a name for the data file in the **File name** field and click the **OK** button to save the data file.

The second method allows you to select the variables to save. In this case, in the Command Window, enter the command **save filename var1**, **var2**, …, **varn**, where

filename is the name of the file to store the data and **var1**, **var2**, … **varn** are the variables that you want to save. In this case, the data will be saved in filename in the LabVIEW Data directory in the path specified in **File»LabVIEW MathScript Properties**. In Figure B.12 the process of saving data is illustrated. In Figure B.12(a), all the variables are saved in the file save_all.mlv after navigating to the folder LabVIEW Data. In Figure B.12(b), the variable **x** is saved in the file save_x.mlv.

You also can load existing data files into your MathScript session. In the MathScript Interactive Window, select **Operate»Load Data**, as illustrated in Figure B.13. Note that you must save data files before you can load them into the MathScript Interactive Window.

In some instances, it may be useful to export your data to an external application. In MathScript this is readily accomplished. On the MathScript Interactive

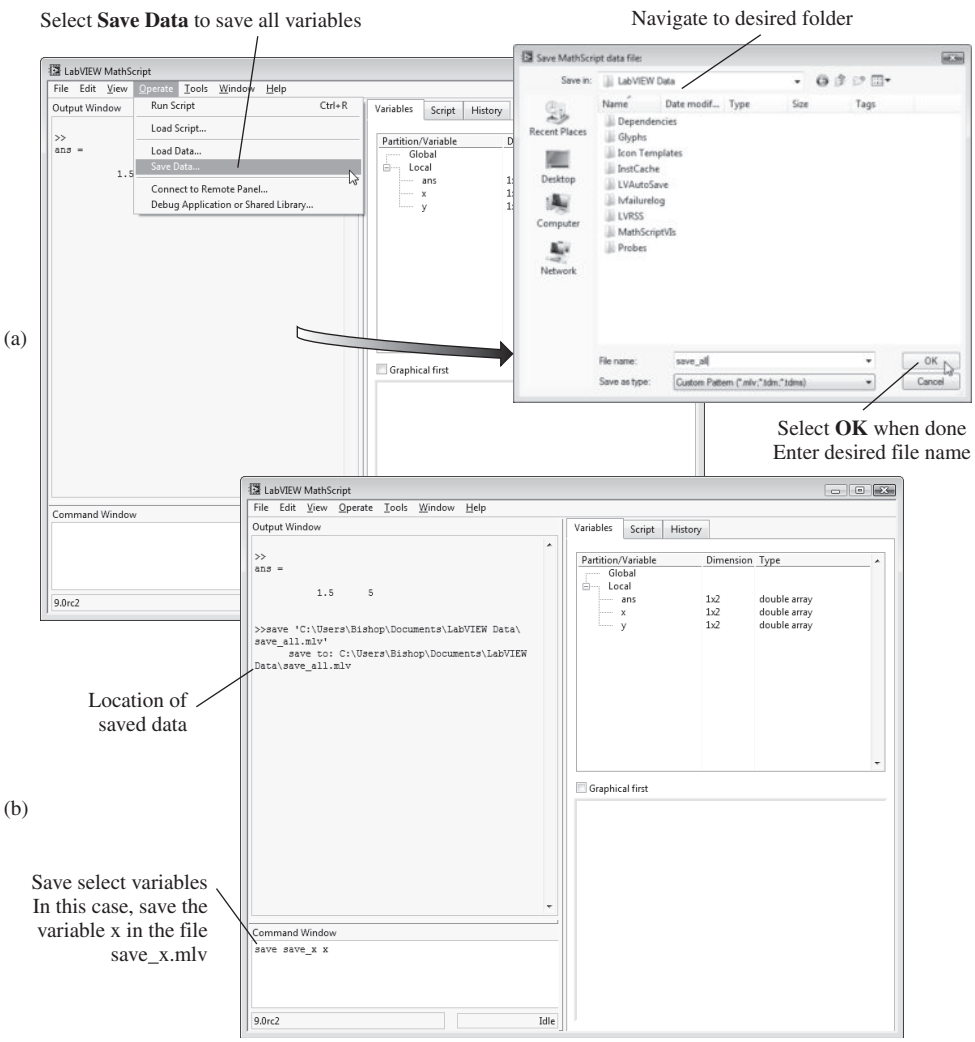


FIGURE B.12 Saving data files. (a) Saving all the variables in the workspace. (b) Saving select variables.

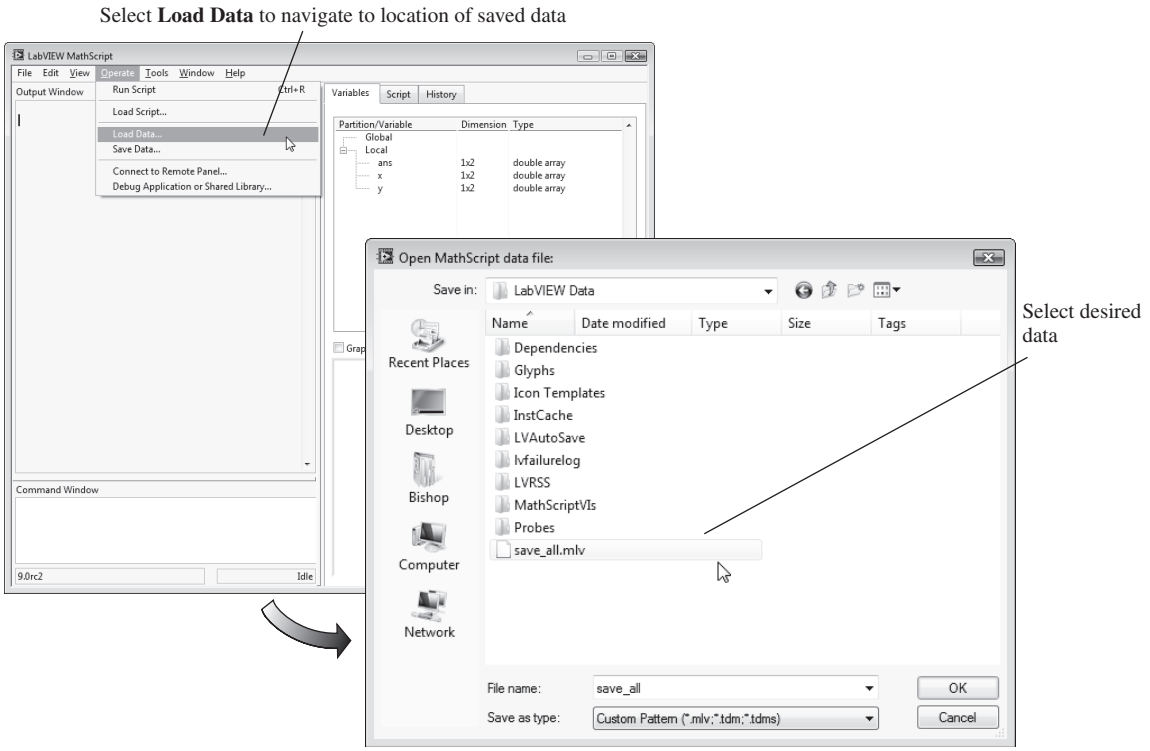


FIGURE B.13 Loading data from a previous MathScript session.

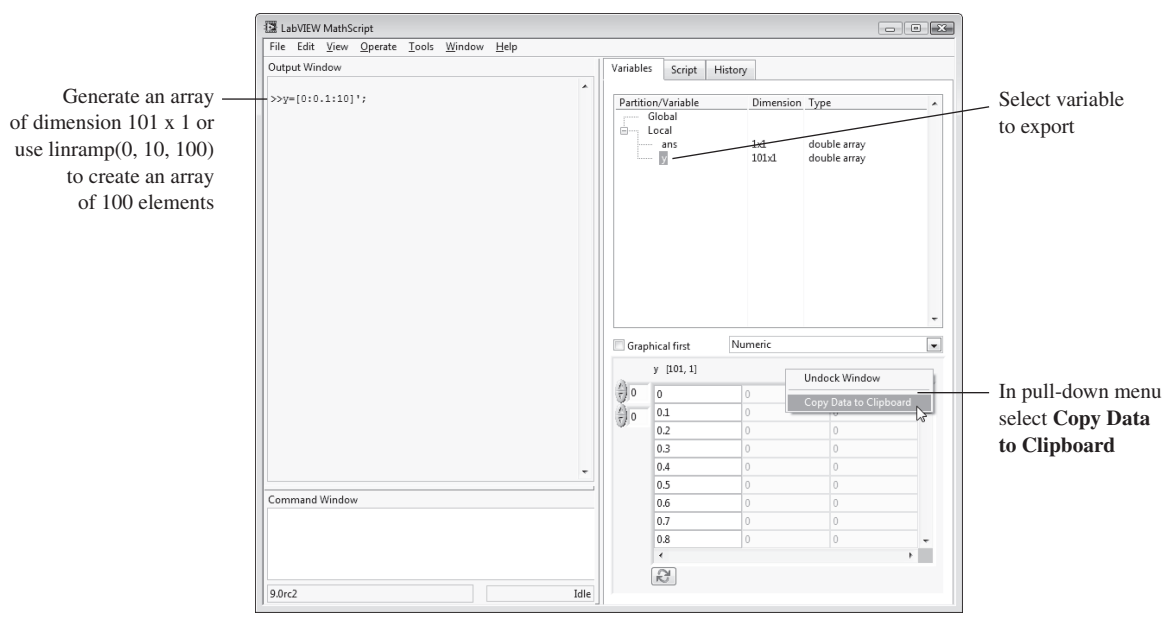


FIGURE B.14 Saving data for external applications.

Window, navigate to the Variables window. Select one of the variables that you desire to export. Exporting is accomplished from the Preview Pane and can be done if the display type is **Numeric**, **Graph**, **XY Graph**, or **Picture**. To do so, right-click the Preview Pane and select **Copy Data to Clipboard** from the shortcut menu to copy the data for the desired variable, as illustrated in Figure B.14. You then can paste the data in your external application. In Figure B.14, the time vector t is being exported. The next step would be to highlight the variable y and export it. Then, both t and y would exist in the spreadsheet for further analysis. Since data in spreadsheets is generally in column format, make sure that the variables that you export are in array format of size $n \times 1$, where n is the length of the array. If the array is a $1 \times n$ array, then when exported to a spreadsheet, it will paste in a row rather than a column.

MATHSCRIPT BASICS: PROBLEMS

B.1 Write a script to generate a 3×2 matrix **M** of random numbers using the rand function. Use the help command for syntax help on the rand function. Verify that each time you run the script the matrix **M** changes.

B.2 In the MathScript Interactive Window, create a script that generates a time vector over the interval 0 to 10 with a step size of 0.5 and creates a second vector, y, according to the equation

$$y = e^{-t}(0.5 \sin 0.1t - 0.25 \cos 0.2t).$$

Add the plot function to generate a graph of y versus t. Type the script in the Script Editor window and, when done, use the **Save** button in the Script Editor to save the script. Clear the script from the Script Editor window, and then **Load** the script back into the Script Editor and select **Run**.

B.3 Create a plot of the cosine function, $y = \cos(t)$, where t varies from 0 to π, with an increment of $\pi/20$.

B.4 Open the MathScript Interactive Window. In the Command Window, create the matrices **A** and **B**:

$$\mathbf{A} = \begin{bmatrix} 1 & -2 \\ 0 & 3 \\ -1 & 5 \end{bmatrix} \text{ and } \mathbf{B} = \begin{bmatrix} 1 & -1 & 7 \\ 2 & 0 & -2 \end{bmatrix}.$$

Is it possible to perform the following math operations on the matrices? If so, what is the result?

(a) **A** * **B**
(b) **B** * **A**
(c) **A** + **B**
(d) **A** + **B**' (where **B**' is the transpose of **B**)
(e) **A**./**B**'

B.5 Using MathScript, generate a plot of a sine wave of frequency $\omega = 10$ rad/sec. Use the linramp function to generate the time vector starting at $t = 0$ and ending at $t = 10$. Label the x-axis as **Time (sec)**. Label the y-axis as **sin(w * t).** Add the following title to the plot: **Sine wave with frequency w = 10 rad/sec**.

B.6 The rand function generates uniformly distributed random numbers between 0 and 1. This means that the average of all of the random values generated by the rand function should approach 0.5 as the number of random numbers increases. Using the rand function, generate random vectors of length 5, 100, 500, and 1000. Confirm that as the number of elements increases, the average of the random numbers approaches 0.5. Generate a plot of the average of the random numbers as a function of the number of random numbers. Use both rand and mean functions in your script.