MATLAB® C Math Library

The Language of Technical Computing

Computation

Visualization

Programming

The MATH WORKS Inc.

User's Guide

Version 2.1

How to Contact The MathWorks:

508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail
`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

MATLAB C Math Library User's Guide

Printing History:		
	October 1995	First printing
	January 1998	Revised for Version 1.2
	January 1999	Revised for Version 2.0 (Release 11)
	September 2000	Revised for Version 2.1 (Release 12) Online only

Contents

1 Getting Started

2 Writing and Building Programs

3 Working with MATLAB Arrays

4 Managing Array Memory

5 Indexing into Arrays

6 Calling Library Routines

7 Importing and Exporting Array Data

8 Handling Errors and Writing a Print Handler

9 Library Routines

A Directory Organization

B Errors and Warnings

1

Getting Started

Introduction

The MATLAB ® C Math Library makes the mathematical core of MATLAB available to application programmers. The library is a collection of more than 400 mathematical routines written in C. Programs written in any language capable of calling C functions can call these routines to perform mathematical computations.

The MATLAB C Math Library is based on the MATLAB language. The mathematical routines in the MATLAB C Math Library are C callable versions of features of the MATLAB language. However, you do not need to know MATLAB or own a copy of MATLAB to use the MATLAB C Math Library. If you have purchased the MATLAB C Math Library, then the only additional software you need is an ANSI C compiler. In addition, you may freely distribute applications you develop with the MATLAB C Math Library.

Who Should Read This Book

This book assumes that you are familiar with general programming concepts such as function calls, variable declarations, and flow of control statements. You also need to be familiar with the general concepts of C and linear algebra. The audience for this book is C programmers who need a matrix math library or MATLAB programmers who want the performance of C. This book will not teach you how to program in either MATLAB or C.

MATLAB C Math Library Features

Version 2.1 of the library provides the following new features:

- Support for the `eval` function for expressions that do not contain variables
- Support for the `input` function with the same restrictions as `eval`
- Performance enhancements in the core numerical routines

Note Existing Version 2.0 hand-written source code is compatible with the Version 2.1 libraries, but you must recompile your code. Additionally, any M-files which were compiled with Version 2.0 of the MATLAB Compiler must be recompiled with Version 2.1 of the MATLAB Compiler before using them with the Version 2.1 libraries. If you do not recompile your program, it will produce a runtime error.

Unsupported MATLAB Features

The library does not include any Handle Graphics® or Simulink® functions. For information about compiling an application that uses graphics functions, see the *MATLAB C/C++ Graphics Library User's Guide*.

In addition, the library does *not* support MATLAB objects

Library Routine Naming Convention

All routines in the MATLAB C Math Library begin with the prefix `mlf`.

The name of every routine in the MATLAB C Math Library is derived from the corresponding MATLAB function. For example, the MATLAB function `sin` is represented by the MATLAB C Math Library function `mlfSin`. The first letter following the `mlf` prefix is always capitalized.

MATLAB C Math Library Documentation

The documentation for the library includes:

- *MATLAB C Math Library User's Guide*—This manual provides tutorial information about the library. This manual is also available in PDF format, accessible through MATLAB Help.
- *MATLAB C Math Library Reference*—The reference pages for all the MATLAB C Math library routines are available in HTML and PDF versions, accessible through MATLAB Help.

How This Book Is Organized

This chapter provides an introduction to the MATLAB C Math Library and tells how to install it. The remainder of the book is organized as follows:

- **Chapter 2, “Writing and Building Programs”** introduces all the primary library features by showing how they are used in a simple example program. This chapter contains pointers to all the other chapters where you can find more detailed information about each new feature. In addition, it includes information about building applications.
- **Chapter 3, “Working with MATLAB Arrays”** describes how to create MATLAB arrays in your C program.
- **Chapter 4, “Managing Array Memory”** describes how to use the MATLAB C Math library automatic memory management facility.
- **Chapter 5, “Indexing into Arrays”** describes how to access individual elements, or groups of elements, in an array. Using indexing you can access, modify, or delete elements in an array.
- **Chapter 6, “Calling Library Routines”** describes the MATLAB C Math Library interface to the MATLAB functions. This chapter describes how to call MATLAB functions that have variable numbers of input arguments and return values.
- **Chapter 7, “Importing and Exporting Array Data”** describes how to use the `mlfLoad()` and `mlfSave()` routines to import data to your application or export data from your application.
- **Chapter 8, “Handling Errors and Writing a Print Handler”** describes how to customize error handling and print handling using MATLAB C Math Library routines.
- **Chapter 9, “Library Routines”** lists the functions available in the MATLAB C Math Library. The chapter groups the more than 400 library functions into functional categories and provides a short description of each function.
- **Appendix A, Directory Organization** describes the MATLAB directory structure.
- **Appendix B, Errors and Warnings** lists the error messages issued by the library.

Accessing Online Reference Documentation

To access the MATLAB online documentation, select the **Help** option from the MATLAB menu bar. MATLAB C Math Library documentation is available in in HTML and PDF formats.

To look up the syntax and behavior for a C Math Library function, refer to the *MATLAB C Math Library Reference*. This reference gives you access to a reference page for each function. Each page presents the function's C syntax and links you to the *MATLAB Function Reference* page for the corresponding MATLAB function.

If you are a stand-alone Math Library user:

1. Open the HTML file `<matlab>/help/mathlib.html` with your Web browser, where `<matlab>` is the top-level directory where you installed the C Math Library.

2. Select *C Math Library Reference*.

Additional Sources of Information

Also available from Help:

- Release notes for the MATLAB C Math Library (`<matlab>\extern\examples\cmath\release.txt`)
- *MATLAB Application Program Interface Reference*
- *MATLAB Application Program Interface Guide*
- *MATLAB Function Reference*
- *Installation Guide for UNIX*
- *Installation Guide for PC*

Installing the MATLAB C Math Library

The MATLAB C Math Library is available on UNIX workstations and PCs running Microsoft Windows (Windows 95, Windows 98, and Windows NT). The installation process is different for each platform.

Note that the MATLAB C Math Library runs on only those platforms (processor and operating system combinations) on which MATLAB runs. In particular, the Math Libraries do not run on DSP or other embedded systems boards, even if those boards are controlled by a processor that is part of a system on which MATLAB runs.

Installation with MATLAB

If you are a licensed user of MATLAB, there are no special requirements for installing the MATLAB C Math Library. Follow the instructions in the MATLAB *Installation Guide* for your specific platform:

- *Installation Guide for UNIX*
- *Installation Guide for PC*

Choose the MATLAB C/C++ Math Library selection from list of components that you can install, displayed by the installation program.

Before you begin installing the MATLAB C Math Library, you must obtain from The MathWorks a Personal License Password (PLP) and, if you are installing the library in a concurrent access environment, a valid License File. These are usually supplied by fax or e-mail. If you have not already received a License File or PLP, contact The MathWorks via:

- The Web at `www.mathworks.com`. On the MathWorks site, click on the MATLAB Access option, log in to the Access home page, and follow the instructions. MATLAB Access membership is free of charge and available to all customers.
- E-mail at `service@mathworks.com`
- Telephone at 508-647-7000; ask for Customer Service
- Fax at 508-647-7001

Installation Without MATLAB

The process for installing the MATLAB C Math Library on its own is identical to the process for installing MATLAB and its toolboxes. Although you are not actually installing MATLAB, you can still follow the instructions in the MATLAB *Installation Guide* for your specific platform.

Verifying a UNIX Installation

To verify that the MATLAB C Math Library has been installed correctly, build one of the example programs distributed with the library. You can find the example programs in the `<matlab>/extern/examples/cmath` directory, where `<matlab>` is your root MATLAB installation directory. See "Building Stand-Alone C Applications" on page 2-7 to learn how to build the example programs using the `mbuild` command.

To spot check the installation, `cd` to the directory `<matlab>/extern/include`, where `<matlab>` symbolizes the MATLAB root directory and verify that the file `matlab.h` exists.

Verifying a PC Installation

When installing a C compiler to use in conjunction with the Math Library, install both the DOS and Windows targets and the command line tools.

The C Math Library installation adds

```
<matlab>\bin
```

to your `$PATH` environment variable, where `<matlab>` symbolizes the MATLAB root directory. The `bin` directory contains the DLLs required by stand-alone applications. After installation, reboot your machine.

To verify that the MATLAB C Math Library has been installed correctly, build one of the example programs distributed with the library. You can find the example programs in the `<matlab>\extern\examples\cmath` directory, where `<matlab>` is your root MATLAB installation directory. See "Building Stand-Alone C Applications" on page 2-7 to learn how to build the example programs using the `mbuild` command.

You can spot check the installation by checking for the file `matlab.h` in `<matlab>\extern\include` and `libmmfile.dll`, `libmatlb.dll`, and `libmcc.dll` in `<matlab>\bin`.

2

Writing and Building Programs

Introduction

This section presents a simple example program that introduces C Math Library programming concepts and describes how to compile and link a C application with the libraries.

Example – Writing a Simple Program

This example application determines if two numbers are relatively prime; that is, the numbers share no common factors. While its function is trivial, the application serves well as an introduction to programming in C with the MATLAB C Math Library. The notes following the example highlight points of particular interest in the example and provide pointers to other sections in the documentation where specific topics are covered in more detail.

The source code for this example, named `intro.c`, is in the `<matlab>/extern/examples/cmath` directory on UNIX systems, and in the `<matlab>\extern\examples\cmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See “Building Stand-Alone C Applications” on page 2-7 for information on building the examples.

```
    /* intro.c*/

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
1   #include "matlab.h"

    int main()
    {
        double num1, num2;
2       mxArray *volatile factors1 = NULL;
        mxArray *volatile factors2 = NULL;
        mxArray *volatile common_factors = NULL;

3       mlfEnterNewContext(0,0);

4       printf("Enter a number: ");
        scanf("%lf", &num1);
        printf("Enter a second number: ");
        scanf("%lf", &num2);

        mlfTry
        {
6          mlfAssign(&factors1, mlfFactor(mlfScalar(num1)));
           mlfAssign(&factors2, mlfFactor(mlfScalar(num2)));

7          mlfAssign(&common_factors,
              mlfIntersect(NULL, NULL, factors1, factors2, NULL));

           if (mlfTobool(mlfIsempty(common_factors)))
8             printf("%0.0lf and %0.0lf are relatively prime\n",
                       num1, num2);
           else
           {
              printf("%0.0lf and %0.0lf share common factor(s):",
                       num1, num2);
              mlfPrintMatrix(common_factors);
           }
9       } /* end mlfTry */
        mlfCatch
```

```
    {
       mlfPrintf("In catch block: \n");
       mlfPrintMatrix(mlfLasterr(NULL));
    }
    mlfEndCatch

10  mxDestroyArray(factors1);
    mxDestroyArray(factors2);
    mxDestroyArray(common_factors);

11  mlfRestorePreviousContext(0,0);
    return(EXIT_SUCCESS);
}
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Applications must include the file"`matlab.h`" which contains the declaration of the `mxArray` data structure and the prototypes for all the functions in the library. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Applications must declare pointers to any MATLAB arrays (`mxArray *`) explicitly used as variables. All arguments to MATLAB routines must be MATLAB arrays. In addition, the routines return newly allocated MATLAB arrays as output. These pointers are declared as volatile pointers because they are assigned values within a try block and so may change without warning due to an error. For more information about working with MATLAB arrays in C programs, see Chapter 3.

3 Applications must enable MATLAB C Math Library automated memory management by calling `mlfEnterNewContext()`. With the library memory management facility enabled, the library can delete the arrays it creates automatically. This allows you to compose functions; that is, nest one function call within another. For more information about automated memory management, see Chapter 4.

4 Applications can define a try block using the MATLAB C Math library macro, `mlfTry`. When a library function included in a try block encounters a run-time error, it outputs an error message and then passes control to a catch block in the program. In a catch block, an application can free arrays that have been assigned to variables or perform other processing before exiting. For more information about defining try and catch blocks, see “Error Handling Overview” on page 8-3.

5 This call to `mlfScalar()`, which converts the number input by the user from an integer to a MATLAB array, illustrates routine nesting. The application can use nesting because it only uses the array returned by `mlfScalar()` as input to `mlfFactor()`. With automated memory management enabled, the library frees the array after `mlfFactor()` is finished using it.

   In contrast, because the example uses the array returned by `mlfFactor()` several times, it assigns this array to a variable, `factors1`, using the `mlfAssign()` routine. Any array that you assign to a variable, you must also free, using `mxDestroyArray()`. (Arrays returned as output arguments by library routines are implicitly assigned to the variables by the library and must also be destroyed.) For more information about assigning arrays to variables, see Chapter 4.

6 This call to `mlfIntersect()` illustrates how to call library routines that optional input arguments. The C Math library version of these functions include in their signatures *all* optional input and output arguments. If you do not use these optional arguments, you must pass `NULL` in their place. For more information about calling MATLAB C Math library routines, see Chapter 6.

7 The routine `mlfIsempty()` returns an array containing the value 1 (`TRUE`) if the array is empty and zero (`FALSE`) if the array contains data. However, because `mlfIsempty()` returns these values as MATLAB arrays, you cannot use the return value directly in the `if` statement. Instead, pass this return value to the `mlfTobool()` routine, which converts the return value to a standard C Boolean value.

   You can also access individual elements in an array using standard MATLAB indexing syntax; however, the values returned by indexing are MATLAB arrays, not scalar values. For more information about indexing into arrays, see Chapter 5.

8 This call to the `mlfCatch` macro defines the start of the application’s catch block. The call to the `mlfEndCatch` macro defines the end of the catch block. Catch blocks contain error handling code. This sample catch block calls the `mlfLasterr()` routine to retrieve the text of the error message associated

with the last error and then outputs the message to the user. For more information about handling errors with try and catch blocks, see Chapter 8.

9 The application frees the MATLAB arrays that were assigned to variables using `mlfAssign()`. The library automatically frees arrays that were not assigned to variables. For example, the arrays returned by the nested calls to `mlfScalar()` are deleted automatically. The arrays assigned to `factors1` and `factors2` are not deleted automatically. For more information about assigning an array to a variable using the `mlfAssign()` routine, see Chapter 4.

10 The sample application ends by disabling automated memory management using the `mlfRestorePreviousContext()`. For more information about enabling automated memory management, see Chapter 4.

Output

This sample program, when run in a DOS Command Prompt window, produces the following output:

```
Enter a number: 333

Enter a second number: 444
333 and 444 share common factor(s): 3     37
```

A second run illustrates the alternate output:

```
Enter a number: 11

Enter a second number: 4
11 and 4 are relatively prime
```

Building Stand-Alone C Applications

After you write your C code, you must compile and link it to create your C application. The section:

- Provides an overview of the compiling and linking process, introducing the MATLAB `mbuild` utility.
- Explains how to build stand-alone C applications on UNIX systems
- Explains how to build stand-alone C applications on PCs running Microsoft Windows.
- Describes how to build shared libraries
- Details the libraries with which you must link your C application if you do not use the `mbuild` utility.

Note You may freely distribute applications you develop with the MATLAB C Math Library.

Packaging Stand-Alone Applications

To distribute a stand-alone application, you must include the application's executable as well as the shared libraries with which the application was linked. The necessary shared libraries vary by platform and are listed within the individual UNIX and Windows sections that follow.

Overview

To build a stand-alone application using the MATLAB C Math Library, you must supply your ANSI C compiler with the correct set of compiler and linker options (or switches). To help you, The MathWorks provides a command line utility called `mbuild`. The `mbuild` script makes it easy to:

- Set your compiler and linker settings
- Change compilers or compiler settings
- Switch between C and C++ development
- Build your application

On UNIX and Microsoft Windows systems, follow these steps to build C applications with `mbuild`:

1 Verify that `mbuild` can create stand-alone applications.

2 Build your application.

You only need to reconfigure if you change compilers or upgrade your current compiler.

Compiler Options Files

`mbuild` stores compiler and linker settings in an options file. Options files contain the required compiler and linker settings for your particular C compiler. The MathWorks provides options files for every supported C compiler.

Much of the information on options files in this chapter is provided for those users who may need to modify an options file to suit their specific needs. Many users never have to be concerned with how the options files work.

Building UNIX Applications

This section:

- Explains how to configure your build environment on UNIX systems
- Describes how to compile and link C source code into a stand-alone UNIX application

This section also includes information about packaging your application for distribution.

Configuring the Build Environment

`mbuild` determines whether to compile in C or C++ by examining the type of files you are compiling. Table 2-1 shows the supported file extensions. If you include both C and C++ files, `mbuild` uses the C++ compiler and the MATLAB C++ Math Library. If `mbuild` cannot deduce from the file extensions whether to compile in C or C++, `mbuild` invokes the C compiler.

Table 2-1: UNIX File Extensions for mbuild

Language	Extension(s)
C	`.c`

C++	`.cpp`
	`.C`
	`.cxx`
	`.cc`

Note You can override the language choice that is determined from the extension by using the `-lang` option of `mbuild`. For more information about this option, as well as all of the other `mbuild` options, see Table 2-2.

Locating Options Files

`mbuild` locates your options file by searching the following:

- The current directory

- `$HOME/.matlab/R12`
- `<matlab>/bin`

`mbuild` uses the first occurrence of the options file it finds. If no options file is found, `mbuild` displays an error message.

Using the System Compiler

If your supported C compiler is installed on your system, you are ready to create C stand-alone applications. To create a stand-alone C application, you can simply enter

```
mbuild filename.c
```

This simple method works for the majority of users. Assuming `filename.c` contains a `main` function, this example uses the system's compiler as your default compiler for creating your stand-alone application.

- If you are a user who does not need to change C or C++ compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to "Building an Application."
- If you need to know how to select a different compiler or change the options file, continue with this section.

Changing the Default Compiler

You need to use the `setup` option if you want to change your default compiler. At the UNIX prompt type:

```
mbuild -setup
```

The `setup` option creates a user-specific options file for your ANSI C or C++ compiler. Using the `setup` option sets your default compiler so that the new compiler is used every time you use the `mbuild` script.

Note The options file is stored in the MATLAB subdirectory of your home directory, for example, `$HOME/.matlab/R12/mbuildopts.sh`. This allows each user to have a separate `mbuild` configuration.

Executing `mbuild -setup` presents a list of options files currently included in the `bin` subdirectory of MATLAB.

```
mbuild -setup

Using the 'mbuild -setup' command selects an options file that is
placed in ~/.matlab/R12 and used by default for 'mbuild'. An
options file in the current working directory or specified on the
command line overrides the default options file in ~/.matlab/R12.

Options files control which compiler to use, the compiler and link
command options, and the runtime libraries to link against.

To override the default options file, use the 'mbuild -f' command
(see 'mbuild -help' for more information).

The options files available for mbuild are:

  1: /matlab/bin/mbuildopts.sh :
        Build and link with MATLAB C/C++ Math Library
```

If there is more than one options file, you can select the one you want by entering its number and pressing **Return**. If there is only one options file available, it is automatically copied to your MATLAB directory if you do not already have an mbuild options file. If you already have an mbuild options file, you are prompted to overwrite the existing one.

Modifying the Options File

Another use of the setup option is if you want to change your options file settings. For example, if you want to make a change to the current linker settings, or you want to disable a particular set of warnings, you should use the setup option.

If you need to change the options that mbuild passes to your compiler or linker, you must first run

```
mbuild -setup
```

which copies a master options file to your local MATLAB directory, typically $HOME/.matlab/R12/mbuildopts.sh.

If you need to see which options mbuild passes to your compiler and linker, use the verbose option, -v, as in

```
mbuild -v filename1 [filename2 …]
```

to generate a list of all the current compiler settings.

To change the options, use an editor to make changes to your options file, which is in your local MATLAB directory. Your local MATLAB directory is a user-specific, MATLAB directory in your individual home directory that is used specifically for your individual options files.

You can also embed the settings obtained from the verbose option of `mbuild` into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

Note Any changes made to the local options file will be overwritten if you execute `mbuild -setup` again. To make the changes persist through repeated uses of `mbuild -setup`, you must edit the master file itself, `<matlab>/bin/mbuildopts.sh`.

Temporarily Changing the Compiler

To temporarily change your C or C++ compiler, use the `-f` option, as in

```
mbuild -f <options_file> filename.c [filename]
```

The `-f` option tells the `mbuild` script to use the options file, `<file>`. If `<file>` is not in the current directory, then `<file>` must be the full pathname to the desired options file. Using the `-f` option tells the `mbuild` script to use the specified options file for the current execution of `mbuild` only; it does not reset the default compiler.

Building an Application

The C source code for the example `ex1.c` is included in the `<matlab>/extern/examples/cmath` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system. To verify that `mbuild` is properly configured on your system to create stand-alone applications, copy `ex1.c` to your local directory and `cd` to that directory. Then, at the UNIX prompt enter:

```
mbuild ex1.c
```

This should create the file called `ex1`. Stand-alone applications created on UNIX systems do not have any extensions. For answers to some common build problems, see “Troubleshooting mbuild” on page 2-31.

Locating Shared Libraries

Before you can run your stand-alone application, you must tell the system where the API and C shared libraries reside. This table provides the necessary UNIX commands depending on your system’s architecture.

Architecture	Command
HP700	`setenv SHLIB_PATH <matlab>/extern/lib/hp700:$SHLIB_PATH`
IBM RS/6000	`setenv LIBPATH <matlab>/extern/lib/ibm_rs:$LIBPATH`
All others	`setenv LD_LIBRARY_PATH <matlab>/extern/lib/<arch>:$LD_LIBRARY_PATH`
	where: `<matlab>` is the MATLAB root directory `<arch>` is your architecture (i.e., `alpha`, `glnx86`, `sgi`, `sol2`)

It is convenient to place this command in a startup script such as `~/.cshrc`. Then, the system will be able to locate these shared libraries automatically, and you will not have to re-issue the command at the start of each login session. The best choice is to place the libraries in `~/.login`, which only gets executed once, if that option is available on your system.

Note On all UNIX platforms, the C libraries are shipped as shared object (`.so`) files or shared libraries (`.sl`). Any stand-alone application must be able to locate the C libraries along the library path environment variable (`SHLIB_PATH`, `LIBPATH`, or `LD_LIBRARY_PATH`) in order to be loaded. Consequently, to share a stand-alone application with another user, you must provide all of the required shared libraries. For more information about the required shared libraries for UNIX, see “Building Microsoft Windows Applications” on page 2-17.

Running Your Application

To launch your application, enter its name on the command line. For example,

```
ex1

     1     3     5
     2     4     6

   1.0000 + 7.0000i   4.0000 +10.0000i
   2.0000 + 8.0000i   5.0000 +11.0000i
   3.0000 + 9.0000i   6.0000 +12.0000i
```

mbuild Options

The `mbuild` script supports various options that allow you to customize the building and linking of your code. Many users do not need to know additional details about the `mbuild` script; they use it in its simplest form. The following information is provided for those users who require more flexibility with the tool.

The `mbuild` syntax and options are

```
mbuild [-options] filename1 [filename2 …]
```

Table 2-2: mbuild Options on UNIX

Option	Description
-c	Compile only; do not link.
-D<name>[=<def>]	Define C preprocessor macro <name> [as having value <def>].
-f <optionsfile>	Use <file> to override the default options file; <file> is a full pathname if it is not in the current directory.
-g	Build an executable with debugging symbols included.
-h[elp]	Help; prints a description of mbuild and the list of options.
-I<pathname>	Include <pathname> in the list of directories to search for header files.
-inline	Inlines matrix accessor functions (mx*). The generated MEX function may not be compatible with future versions of MATLAB.
-l<file>	Link against library lib<file>.
-L<pathname>	Include <pathname> in the list of directories to search for libraries.
-lang <language>	Override language choice implied by file extension. <language> = c for C cpp for C++ This option is necessary when you use an unsupported file extension, or when you pass all .o files and libraries.

Table 2-2: mbuild Options on UNIX (Continued)

Option	Description
`<name>=<def>`	Override options file setting for variable `<name>`. If `<def>` contains spaces, enclose it in single quotes, for example, `CFLAGS='opt1 opt2'`. The definition, `<def>`, can reference other variables defined in the options file. To reference a variable in the options file, prepend the variable name with a `$`, for example, `CFLAGS='$CFLAGS opt2'`.
`-n`	No execute flag. Using this option displays the commands that compile and link the target but does not execute them.
`-outdir <dirname>`	Place any generated object, resource, or executable files in the directory `<dirname>`. Do not combine this option with `-output` if the `-output` option gives a full pathname.
`-output <name>`	Create an executable named `<name>`. (An appropriate executable extension is automatically appended.)
`-O`	Build an optimized executable.
`-setup`	Set up the default compiler and libraries. This option should be the only argument passed.
`-U<name>`	Undefine C preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.

Building Microsoft Windows Applications

This section:

- Explains how to configure your build environment on PCs
- Describes how to compile and link C source code into a stand-alone PC application

This section also includes information about packaging your application for distribution.

Configuring the Build Environment

`mbuild` determines whether to compile in C or C++ by examining the type of files you are compiling. Table 2-1 shows the file extensions that `mbuild` interprets as indicating C or C++ files. If you include both C and C++ files, `mbuild` uses the C++ compiler and the MATLAB C++ Math Library. If `mbuild` cannot deduce from the file extensions whether to compile in C or C++, `mbuild` invokes the C compiler.

Table 2-3: Windows File Extensions for mbuild

Language	Extension(s)
C	`.c`

C++	`.cpp`
	`.cxx`
	`.cc`

Note You can override the language choice that is determined from the extension by using the `-lang` option of `mbuild`. For more information about this option, as well as all of the other `mbuild` options, see Table 2-5.

Locating Options Files

To locate your options file, the `mbuild` script searches the following:

- The current directory

- The user `Profiles` directory
- `<matlab>\bin`

`mbuild` uses the first occurrence of the options file it finds. If no options file is found, `mbuild` searches your machine for a supported C compiler and uses the factory default options file for that compiler. If multiple compilers are found, you are prompted to select one.

The User Profile Directory Under Windows. The Windows user `Profiles` directory is a directory that contains user-specific information such as Desktop appearance, recently used files, and Start Menu items. The `mbuild` utility stores its options file `compopts.bat` that is created during the `-setup` process in a subdirectory of your user `Profiles` directory, named `Application Data\MathWorks\MATLAB\R12`.

Under Windows NT and Windows 95/98 with user profiles enabled, your user profile directory is `%windir%\Profiles\username`. Under Windows 95/98 with user profiles disabled, your user profile directory is `%windir%`. Under Windows 95/98, you can determine whether or not user profiles are enabled by using the Passwords control panel.

Systems with Exactly One C/C++ Compiler

If your supported C compiler is installed on your system, you are ready to create C stand-alone applications. On systems where there is exactly one C compiler available to you, the `mbuild` utility automatically configures itself for the appropriate compiler. So, for many users, to create a C stand-alone application, you can simply enter

```
mbuild filename.c
```

This simple method works for the majority of users. It uses your installed C compiler as your default compiler for creating your stand-alone applications.

- If you are a user who does not need to change compilers, or you do not need to modify your compiler options files, you can skip ahead in this section to “Building an Application.”
- If you need to know how to change the options file or select a different compiler, continue with this section.

Systems with More than One Compiler

On systems where there is more than one C compiler, the mbuild utility lets you select which of the compilers you want to use. Once you choose your C compiler, that compiler becomes your default compiler and you no longer have to select one when you compile your stand-alone applications.

For example, if your system has both the Borland and Microsoft compilers, when you enter for the first time

```
mbuild filename.c
```

you are asked to select which compiler to use.

```
mbuild has detected the following compilers on your machine:

[1] : Borland compiler in T:\Borland\BC.500
[2] : MSVC compiler in T:\DevStudio\c.106

[0] : None

Please select a compiler. This compiler will become the default:
```

Select the desired compiler by entering its number and pressing **Return**. You are then asked to verify your information.

Changing the Default Compiler

To change your default C compiler, you select a different options file. You can do this at any time by using the setup command.

This example shows the process of changing your default compiler to the Microsoft Visual C/C++ Version 6.0 compiler.

```
mbuild -setup

Please choose your compiler for building standalone MATLAB
applications.

Would you like mbuild to locate installed compilers [y]/n? n

Choose your C/C++ compiler:
[1] Borland C/C++         (version 5.0, 5.2, or 5.3)
[2] Microsoft Visual C/C++ (version 4.2, 5.2, or 6.0)
```

```
[0] None

Compiler: 2

Choose the version of your C/C++ compiler:
[1] Microsoft Visual C/C++ 4.2
[2] Microsoft Visual C/C++ 5.0
[3] Microsoft Visual C/C++ 6.0

version: 3

Your machine has a Microsoft Visual C/C++ compiler located at
D:\Program Files\DevStudio6.
Do you want to use this compiler [y]/n? y

Please verify your choices:

Compiler: Microsoft Visual C/C++ 6.0
Location: D:\Program Files\DevStudio6

Are these correct?([y]/n): y

The default options file:
"C:\WINNT\Profiles\username
\Application Data\MathWorks\MATLAB\R12\compopts.bat" is being
updated...
```

If the specified compiler cannot be located, you are given the message:

```
The default location for compiler-name is directory-name,
but that directory does not exist on this machine.
Use directory-name anyway [y]/n?
```

Using the `setup` option sets your default compiler so that the new compiler is used every time you use the `mbuild` script.

Modifying the Options File

Another use of the `setup` option is if you want to change your options file settings. For example, if you want to make a change the current linker settings, or you want to disable a particular set of warnings, use the `setup` option.

The `setup` option copies the appropriate options file to your user profile directory and names it `compopts.bat`. Make your user-specific changes to `compopts.bat` in the user profile directory and save the modified file. This sets your default compiler's options file to your specific version.

Table 2-4, Compiler Options Files on the PC lists the names of the PC master options files included in this release of the MATLAB C Math Library.

If you need to see which options `mbuild` passes to your compiler and linker, use the verbose option, `-v`, as in

```
mbuild -v filename1 [filename2 …]
```

to generate a list of all the current compiler settings used by `mbuild`.

You can also embed the settings obtained from the verbose option into an integrated development environment (IDE) or makefile that you need to maintain outside of MATLAB. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

Note Any changes that you make to the local options file `compopts.bat` will be overwritten the next time you run `mbuild -setup`. If you want to make your edits persist through repeated uses of `mbuild -setup`, you must edit the master file itself. The master options files are located in `<matlab>\bin\win32\mbuild`.

Table 2-4: Compiler Options Files on the PC

Compiler	Master Options File
Borland C/C++, Version 5.0	`bcccompp.bat`
Borland C/C++, Version 5.2	`bcc52compp.bat`
Borland C++Builder 3.0	`bcc53compp.bat`
Borland C++Builder 4.0	`bcc54compp.bat`
Borland C++Builder 5.0	`bcc55compp.bat`
Lcc 2.4 (bundled with MATLAB)	`lcccompp.bat`
Microsoft Visual C/C++, Version 5.0	`msvc50compp.bat`
Microsoft Visual C/C++, Version 6.0	`msvc60compp.bat`

Combining Customized C and C++ Options Files

The same options files can be used to create both C and C++ stand-alone applications. If you have modified your own separate options files to create C and C++ applications, you can combine them into one options file.

To combine your existing options files into one universal C and C++ options file:

1. Copy from the C++ options file to the C options file all lines that set the variables `COMPFLAGS`, `OPTIMFLAGS`, `DEBUGFLAGS`, and `LINKFLAGS`.
2. In the C options file, within just those copied lines from step 1, replace all occurrences of `COMPFLAGS` with `CPPCOMPFLAGS`, `OPTIMFLAGS` with `CPPOPTIMFLAGS`, `DEBUGFLAGS` with `CPPDEBUGFLAGS`, and `LINKFLAGS` with `CPPLINKFLAGS`.

This process modifies your C options file to be a universal C/C++ options file.

Temporarily Changing the Compiler

To temporarily change your C compiler, use the `-f` option, as in

```
mbuild -f <file> …
```

The `-f` option tells the `mbuild` script to use the options file, `<file>`. If `<file>` is not in the current directory, then `<file>` must be the full pathname to the desired options file. Using the `-f` option tells the `mbuild` script to use the specified options file for the current execution of `mbuild` only; it does not reset the default compiler.

Building an Application

C source code for the example `ex1.c` is included in the `<matlab>\extern\examples\cmath` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system. To verify that `mbuild` is properly configured on your system to create stand-alone applications, enter at the DOS prompt:

```
mbuild ex1.c
```

This creates the file called `ex1.exe`. Stand-alone applications created on Windows 95 or NT always have the extension `.exe`. The created application is a 32-bit Microsoft Windows console application. For answers to some common build problems, see “Troubleshooting mbuild” on page 2-31.

Shared Libraries (DLLs)

All the WIN32 Dynamic Link Libraries (DLLs) for the MATLAB C Math Library are in the directory

```
<matlab>\bin
```

The `.def` files for the Microsoft and Borland compilers are in the `<matlab>\extern\include` directory. Import libraries for supported compilers can be found in `<matlab>\extern\lib\win32\<compiler>`.

Before running a stand-alone application, you must ensure that the directory containing the DLLs is on your path. The directory must be on your operating system `$PATH` environment variable. On Windows 95, set the value in your `autoexec.bat` file; on Windows NT, use the Control Panel to set it.

Running Your Application

You can now run your stand-alone application by launching it from the command line. For example,

```
ex1

     1     3     5
     2     4     6

   1.0000 + 7.0000i   4.0000 +10.0000i
   2.0000 + 8.0000i   5.0000 +11.0000i
   3.0000 + 9.0000i   6.0000 +12.0000i
```

mbuild Options

The `mbuild` script supports various options that allow you to customize the building and linking of your code. Many users do not need to know any additional details of the `mbuild` script; they use it in its simplest form. The following information is provided for those users who require more flexibility with the tool.

The `mbuild` syntax and options are

`mbuild [-`*`options`*`] filename1 [`*`filename2`*` …]`

Table 2-5: mbuild Options on Microsoft Windows

Option	Description
`@filename`	Replace `@filename` on the `mbuild` command line with the contents of `filename`. `filename` is a response file, i.e., a text file that contains additional command line options to be processed.
`-c`	Compile only; do not link.
`-D<name> [#<def>]`	Define C preprocessor macro `<name>` as having value `[#<def>]`
`-f <file>`	Use `<file>` as the options file; `<file>` is a full pathname if it is not in the current directory.
`-g`	Build an executable with debugging symbols included.
`-h[elp]`	Help; prints a description of `mbuild` and the list of options.
`-I<pathname>`	Include `<pathname>` in the list of directories to search for header files.
`<name>#<def>`	Override options file setting for variable `<name>`. This option is equivalent to `<ENV_VAR>#<val>`, which temporarily sets the environment variable `<ENV_VAR>` to `<val>` for the duration of the call to `mex`. `<val>` can refer to another environment variable by prepending the name of the variable with a `$`, e.g., `COMPFLAGS#"$COMPFLAGS -myswitch"`.

Table 2-5: mbuild Options on Microsoft Windows (Continued)

Option	Description
-inline	Inlines matrix accessor functions (mx*). The generated MEX-function may not be compatible with future versions of MATLAB.
-lang <language>	Override language choice implied by file extension. <language> = c for C cpp for C++ This option is necessary when you use an unsupported file extension, or when you pass all .o files and libraries.
-n	Print out any commands that mbuild would have executed, but do not actually execute any of them.
-O	Build an optimized executable.
-outdir <dirname>	Place any generated object, resource, or executable files in the directory <dirname>. Do not combine this option with -output if the -output option gives a full pathname.
-output <name>	Create an executable named <name>. (An appropriate executable extension is automatically appended.)
-setup	Set up the default compiler and libraries. This option should be the only argument passed.
-U<name>	Undefine C preprocessor macro <name>.
-v	Verbose; print all compiler and linker settings.

Distributing Stand-Alone Applications

You may freely distribute applications you develop with the MATLAB C Math Library, subject to The MathWorks software license agreement. When you package your application for distribution, remember to include, along with your application executable, these additional files:

- The contents, if any, of a directory named `bin`, created by `mbuild` in the same directory as your application executable
- Any custom MEX files your application uses
- All the MATLAB math run-time libraries

To make packaging an application easier, the C Math Library has prepackaged all the necessary MATLAB run-time libraries into a single, self-extracting archive file. For more information about how you can use this archive, see "Packaging the MATLAB Math Run-Time Libraries". For information about how customers who receive your application can use this archive, see "Installing Your Application" on page 2-28.

Packaging the MATLAB Math Run-Time Libraries

The MATLAB C Math library has prepackaged all the MATLAB run-time libraries required by stand-alone applications into a single, self-extracting archive file, called the MATLAB Math and Graphics Run-Time Library Installer. Instead of including all the run-time libraries individually in your stand-alone application distribution package, you can simply include this archive file.

The following table lists the name of the archive file for both PCs and UNIX systems. In the table `$MATLAB` represents your MATLAB installation directory and `$ARCH` represents your UNIX platform.

Platform	MATLAB Math and Graphics Run-Time Library Installer
UNIX systems	`$MATLAB/extern/lib/$ARCH/mglinstaller`
PCs	`$MATLAB\extern\lib\win32\mglinstaller.exe`

Installing Your Application

To install your application, your customers must:

- Run the MATLAB Math and Graphics Run-Time Library Installer. This program extracts the libraries from the archive and installs them in subdirectories of a directory specified by the user.
- Add the `bin/$ARCH` subdirectory to their path. This is the only MATLAB Math and Graphics Run-time library subdirectory that needs to be added to the path.

Note If a customer already has the MATLAB math and graphics run-time libraries installed on their system, they do not need to reinstall them. They only need to ensure that the library search path is configured correctly.

On UNIX Systems

On UNIX systems, your customer runs the MATLAB Math and Graphics Run-Time Library Installer by executing the `mglinstaller` command at the system prompt. Your customer can specify the name of the directory into which they want to install the libraries. By default, the installer puts the files in the current directory.

After the installer unpacks and uncompresses the libraries, your customers must add the name of the `bin/$ARCH` subdirectory to the `LD_LIBRARY_PATH` environment variable. (The equivalent variable on HP-UX systems is the `SHLIB_PATH` and `LIBPATH` on IBM AIX systems.)

For example, if a customer working on a Linux system specifies the installation directory `mgl_runtime_dir`, then they must add `mgl_runtime_dir/bin/glnx86` to the `LD_LIBRARY_PATH` environment variable.

On PCs

On PCs, your customer can run the MATLAB Math and Graphics Run-Time Library Installer by double-clicking on the `mglinstaller.exe` file. Your customer can specify the name of the directory into which they want to install the libraries. By default, the installer puts the files in the current directory.

After the installer unpacks and uncompresses the libraries, your customers must add the `bin\win32` subdirectory to the system path variable (`PATH`).

For example, if your customer specifies the installation directory mgl_runtime_dir, then they must add mgl_runtime_dir\bin\win32 to PATH.

Problem Starting Stand-Alone Application

Your application may compile successfully but fail when you or one of your customers tries to start it. If you run the application from a DOS command window, you or one of your customers may see an error message such as:

```
The ordinal #### could not be located in the dynamic-link library
dforrt.dll.
```

To fix this problem, locate the files named dforrt.dll or dformd.dll in your Windows system directory and replace them with the versions of these files in the <MATLAB>\bin\win32 directory, where <MATLAB> represents the name of your MATLAB installation directory.

This same solution works for customers of your application who encounter the same problem; however, they can replace the versions of these files in the Windows system directory with the versions they find in <MGLRUNTIMELIBRARY>\bin\win32 directory, where <MGLRUNTIMELIBRARY> is the name of the directory in which they installed the MATLAB Math and Graphics Run-Time Libraries. See “Distributing Stand-Alone Applications” on page 2-27 for more information.

Building Shared Libraries

You can use `mbuild` to build C shared libraries on both UNIX and the PC. All of the `mbuild` options that pertain to creating stand-alone applications also pertain to creating C shared libraries. To create a C shared library, you use the option

```
-link shared
```

and specify one or more files with the `.exports` extension. The `.exports` files are text files that contain the names of the functions to export from the shared library, one per line. You can include comments in your code by beginning a line (first column) with # or a *. `mbuild` treats these lines as comments and ignores them. `mbuild` merges multiple `.exports` files into one master exports list. For example, given `file2.exports` as:

```
times2
times3
```

and `file1.c` as:

```
int times2(int x)
{
   return 2 * x;
}

int times3(int x)
{
   return 3 * x;
}
```

The command

```
mbuild -link shared file1.c file2.exports
```

creates a shared library named `file1.`*ext*, where *ext* is the platform-dependent shared library extension. For example, on the PC, it would be called `file1.dll`.

Troubleshooting mbuild

This section identifies some of the more common problems that may occur when configuring `mbuild` to create applications.

Options File Not Writable

When you run `mbuild -setup`, `mbuild` makes a copy of the appropriate options file and writes some information to it. If the options file is not writable, you are asked if you want to overwrite the existing options file. If you choose to do so, the existing options file is copied to a new location and a new options file is created.

Directory or File Not Writable

If a destination directory or file is not writable, ensure that the permissions are properly set. In certain cases, make sure that the file is not in use.

mbuild Generates Errors

On UNIX, if you run `mbuild` *`filename`* and get errors, it may be because you are not using the proper options file. Run `mbuild -setup` to ensure proper compiler and linker settings.

Compiler and/or Linker Not Found

On PCs running Windows, if you get errors such as `Bad command or filename` or `File not found`, make sure the command line tools are installed and the path and other environment variables are set correctly.

mbuild Not a Recognized Command

If `mbuild` is not recognized, verify that `<matlab>\bin` is on your path. On UNIX, it may be necessary to rehash.

Cannot Locate Your Compiler (PC)

If `mbuild` has difficulty locating your installed compilers, it is useful to know how it goes about finding compilers. `mbuild` automatically detects your installed compilers by first searching for locations specified in the following environment variables:

- `BORLAND` for the Borland C/C++ Compiler, Version 5.0 or 5.2, and Borland C++Builder, Version 3.0, 4.0, and 5.0.

- MSVCDIR for Microsoft Visual C/C++, Version 5.0 or 6.0
- MSDEVDIR for Microsoft Visual C/C++, Version 4.2

Next, mbuild searches the Windows Registry for compiler entries.

Internal Error When Using mbuild -setup (PC)

Some antivirus software packages such as Cheyenne AntiVirus and Dr. Solomon may conflict with the mbuild -setup process. If you get an error message during mbuild -setup of the following form

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and rerun mbuild -setup. After you have successfully run the setup option, you can re-enable your antivirus software.

Verification of mbuild Fails

If none of the previous solutions addresses your difficulty with mbuild, contact Technical Support at The MathWorks at support@mathworks.com or 508-647-7000.

Linking Applications Without mbuild

To build any of the examples or your own applications without `mbuild`, compile the file with an ANSI C compiler. You must set the include file search path to contain the directory that contains the file `matlab.h`; compilers typically use the `-I` switch to add directories to the include file search path. See Appendix A to determine where `matlab.h` is installed. Link the resulting object files against the libraries in this order:

1. MATLAB M-File Math Library (`libmmfile`)
2. MATLAB Built-In Library (`libmatlb`)
3. MATLAB MAT-File Library (`libmat`)
4. MATLAB Application Program Interface Library (`libmx`)
5. ANSI C Math Library (`libm`)

Specifying the libraries in the wrong order on the command line typically causes linker errors.

On some platforms, additional libraries are necessary. See the platform-specific section of the `mbuild` script for the names and order of these libraries.

On PCs, import libraries can be found in `<matlab>\extern\lib\win32\<compiler>`.

3

Working with MATLAB Arrays

Introduction

To use the routines in the MATLAB C Math Library, you must pass your data to the routines in the form of a MATLAB array. This section:

- Describes the MATLAB arrays supported by the library and the C data type defined to represent them.
- Describes how to work with the various MATLAB array types, including:
  - Numeric Arrays
  - Sparse Matrices
  - Character Arrays
  - Cell Arrays
  - MATLAB Structures
- Perform programming tasks common to all array types.

Because the library routines work the same as the corresponding MATLAB functions, this chapter does not describe their function in detail. For more information about MATLAB arrays and their use, see *Using MATLAB*. Instead, this section provides an overview of working with MATLAB arrays and highlights where the syntax of the library routine is significantly different than its MATLAB counterpart.

Supported MATLAB Array Types

The MATLAB C Math Library supports the following MATLAB array types (or classes):

- Numeric arrays – The library supports multidimensional numeric arrays, where values are represented in double precision format. All MATLAB arithmetic functions operate on numeric arrays. For more information about working with numeric arrays, see “Numeric Arrays” on page 3-4.
- Sparse arrays – To conserve space, two-dimensional numeric arrays can be stored in sparse format, where only nonzero elements of the array are stored. Numeric arrays with more than two dimensions cannot be converted to sparse format. For more information about working with sparse arrays, see “Sparse Matrices” on page 3-17.
- Character arrays – The library supports multidimensional arrays of characters, represented in 16-bit ASCII Unicode format. For more

information about working with character arrays, see “Character Arrays” in Chapter 3.

- Cell arrays – The library supports multidimensional arrays of MATLAB’s primary container type called *cells.* Each cell can contain any type of MATLAB array, including other cell arrays. For more information about working with cell arrays, see “Cell Arrays” on page 3-28.
- Structures – The library supports multidimensional arrays of MATLAB’s other container type called *structures*. A structure can be thought of as a one-dimensional cell array where each cell is assigned a name. These named cells, called *fields*, define the organization of the structure. Do not confuse MATLAB structures with standard C structures. For more information about working with MATLAB structures, see “MATLAB Structures” on page 3-34.

Choose the MATLAB array type that best fits your data. For more detailed information about these array types, see *Using MATLAB*.

MATLAB Array C Data Type

The MATLAB C Math Library uses one data type, `mxArray`, to represent all types of MATLAB arrays. The `mxArray` data type is defined by the MATLAB Application Program Interface (API). Each instance of this data type contains information that identifies the type of array and the size and shape of the array.

The `mxArray` data type is an opaque data type. The MATLAB API includes routines to create arrays and access them. These routines are identified by the prefix `mx`. For a complete list of these routines, see Chapter 9.

As a convenience, the MATLAB C Math Library includes routines to create certain types of commonly used arrays. The sections in this chapter that describe the various types of arrays detail these routines. These routines, like all the library routines, are identified by the prefix `mlf`. You can use a mix of `mlf` and `mx` routines to create and manipulate arrays.

Numeric Arrays

The MATLAB C Math Library includes routines to create and manipulate numeric arrays. Numeric arrays are the fundamental MATLAB array type. The elements of the array are stored as a one-dimensional vector of double-precision numbers. Imaginary data, if present, is stored in a separate vector.

This table lists the MATLAB C Math Library routines that create numeric arrays and perform some basic operations on them. The sections that follow provide more detail about using these routines. For more detailed information about using numeric arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the online *MATLAB C Math Library Reference*.

Table 3-1: Numeric Array Routines

Task	Routine
Create a 1-by-1 array (scalar)	`mlfScalar()`
Create a 1-by-*n* array (vector)	`mlfColon()`
Create an *m*-by-*n* array (matrix)	`mlfDoubleMatrix()` or `mxCreateDoubleMatrix()`
Create a multidimensional, (*m*-by-*n*-by-*p*-by...) array	`mxCreateNumericArray()`
Create a numeric array by concatenating existing arrays	`mlfHorzcat()` `mlfVertcat()`
Create commonly useful, multidimensional arrays, such as arrays of ones, zeros, random numbers, identity matrices, and magic squares.	`mlfOnes()` `mlfZeros()` `mlfRand()`, `mlfRandn()` `mlfEye()` `mlfMagic()`

Creating Numeric Arrays

To create a numeric array, use any of the following mechanisms:

- Using an array creation routine

- Calling an arithmetic routine
- Concatenating existing arrays
- Assigning a value to an element in an array

Using Numeric Array Creation Routines

The MATLAB C Math Library contains many routines that create various types of numeric arrays, including scalar arrays, vectors, matrices, multidimensional arrays and some commonly useful arrays.

Creating Scalar Arrays. The simplest way to create an array is to use the `mlfScalar()` routine. When you pass this routine a numeric value, it creates an 1-by-1 numeric array containing the value, stored in double precision format. Whenever you have to pass a numeric value to a library routine, you can use `mlfScalar()`.

Creating Two-Dimensional Arrays (Matrices). Because two-dimensional arrays of double precision values are used so often in MATLAB, the library includes the routine `mlfDoubleMatrix()` that allows you to create a matrix of double precision values and initialize it with data. Note that you can use integers to specify the array dimensions; you do not need to convert these to arrays.

```
static double data[] = { 1, 4, 2, 5, 3, 6 };

mxArray *A = NULL;

mlfAssign(&A, mlfDoubleMatrix(2,     /*  Rows  */
                              3,     /*  Columns */
                              data,
                              NULL));/* No imaginary part */

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This code produces the following output.

```
1    2    3
4    5    6
```

In this code fragment, note that the values in the C array used to initialize the MATLAB array are not specified in numeric order because MATLAB stores arrays in column major order and C arrays are stored in row-major order. For more information about this, see “Initializing a Numeric Array with Data” on page 3-11.

You can also use the MATLAB API routine, `mxCreateDoubleMatrix()`, to create a matrix of doubles.

Creating Multidimensional Numeric Arrays. To create a multidimensional numeric array, use the MATLAB API routine `mxCreateNumericArray()`. The arguments to this routine are:

- The number of dimensions of the array
- The size of each dimension
- The type of data the array will contain
- Whether the data is real or complex

For example, the following code fragment creates a three-dimensional array, with dimensions 3-by-3-by-2. The `mxDOUBLE_CLASS` argument specifies that the array should contain double precision values. For a complete list of these class specifiers, use the MATLAB Help Desk to view the `mxClass_ID` online reference page.

Note in the example that the arguments specifying the number of dimensions, `ndim`, and the size of these dimensions, `dims`, do not need to be converted into a MATLAB array. The MATLAB API routines accept integer arguments.

```
int ndim    = 3;          /* Number of dimensions */
int dims[3] = { 3,3,2 }; /* Size of dimensions   */

mxArray *A = NULL;       /* declare pointer to mxArray */

mlfAssign(&A, mxCreateNumericArray( ndim,
                                    dims,
                                    mxDOUBLE_CLASS,
                                    mxREAL));

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This code creates the following 3-by-3-by-2 array.

```
(:,:,1) =
     0    0    0
     0    0    0
     0    0    0

(:,:,2) =
     0    0    0
     0    0    0
     0    0    0
```

For information about initializing this array with data, see “Initializing a Numeric Array with Data” on page 3-11.

Creating Commonly Used Numeric Arrays. The MATLAB C Math Library includes several routines that create commonly used multidimensional arrays:

- Array of ones, `mlfOnes()`
- Array of zeros, `mlfZeros()`
- Identity matrices, `mlfEye()`
- Random numbers, `mlfRand()`
- Normally distributed random numbers, `mlfRandn()`
- Magic squares (limited to two-dimensions), `mlfMagic()`

With all these routines, the number of dimensions in the resulting array equals the number of non-`NULL` arguments passed to the routine. To illustrate, the following example passes three arguments to `mlfOnes()` to create a three dimensional array. Because these routines allow you to create arrays of any number of dimensions, you must signify the end of the argument list by specifying a `NULL`.

```
mxArray *A = NULL;

mlfAssign(&A, mlfOnes( mlfScalar(2),
                       mlfScalar(3),
                       mlfScalar(2),
                       NULL));

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This code fragment creates the following array:

```
(:,:,1) =
     1    1    1
     1    1    1

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

Creating Vectors of Number Sequences. To create a one dimensional array (vector) that contains a number sequence, use the `mlfColon()` routine. This routine performs the same function as the MATLAB colon (:) operator.

For example, the following code fragment creates a vector of all the numbers between 1 and 10.

```
mxArray *A = NULL;

mlfAssign(&A, mlfColon(mlfScalar(1),mlfScalar(10),NULL));

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This code creates the following output.

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

You can optionally specify an increment between the values in the vector. For more information, see the `mlfColon()` reference page in the MATLAB C Math Library Reference online documentation.

Creating Numeric Arrays by Calling Arithmetic Routines

The MATLAB C Math Library arithmetic routines create numeric arrays as their output. For example, the sample application in Chapter 2 creates arrays by calling the library arithmetic routines, `mlfFactor()` and `mlfIntersect()`.

Creating Numeric Arrays by Concatenation

You can create arrays by grouping together existing MATLAB arrays using concatenation. In MATLAB, you use the `[]` (brackets) operator to concatenate arrays vertically or horizontally. For example, you can use the following syntax in MATLAB to concatenate arrays. In this MATLAB example, the numeric values being concatenated are scalar arrays. The semicolon indicates that you want to create rows for vertical as well as horizontal concatenation.

```
» A = [ 1 2 3; 4 5 6 ]
A =
     1     2     3
     4     5     6
```

The MATLAB C Math Library uses the `mlfHorzcat()` and `mlfVertcat()` routines to perform horizontal and vertical concatenation. You nest calls to `mlfHorzcat()` inside `mlfVertcat()` to create the same two-dimensional array in a C program.

Note This code fragment duplicates the preceding MATLAB syntax; however, it is more efficient to create a two dimensional array of constants using `mlfDoubleMatrix()`, rather than with `mlfHorzcat()` and `mlfVertcat()`.

```
mxArray *A = NULL;
mlfAssign(&A, mlfVertcat(mlfHorzcat(mlfScalar(1),
                                    mlfScalar(2),
                                    mlfScalar(3),
                                    NULL),
                         mlfHorzcat(mlfScalar(4),
                                    mlfScalar(5),
                                    mlfScalar(6),
                                    NULL),
                         NULL ));

mlfPrintMatrix(A);

mxDestroyArray(A);
```

Creating Multidimensional Numeric Arrays by Concatenation. Using `mlfVertcat()` and `mlfHorzcat()`, you can only create two-dimensional arrays. To create a multidimensional numeric array through concatenation, you must use the `mlfCat()` routine. As arguments to `mlfCat()`, you specify the dimensions along which to concatenate the arrays.

For example, the following code fragment creates two matrices, and then concatenates them to create a three-dimensional array.

```
mxArray *A = NULL;
mxArray *B = NULL;
mxArray *C = NULL;

static double data1[] = { 1, 4, 2, 5, 3, 6 };
static double data2[] = { 7, 10, 8, 11, 9, 12 };

mlfAssign(&A, mlfDoubleMatrix(2, 3, data1, NULL));
mlfAssign(&B, mlfDoubleMatrix(2, 3, data2, NULL));

mlfAssign(&C, mlfCat(mlfScalar(3), A, B, NULL));

mlfPrintMatrix(C);

mxDestroyArray(A);
mxDestroyArray(B);
mxDestroyArray(C);
```

This program displays the following output.

```
(:,:,1) =
   1   2   3
   4   5   6

(:,:,2) =
   7   8   9
  10  11  12
```

Creating Numeric Arrays by Assignment

You can also create a numeric array by assigning a value to a location in the array, using the `mlfIndexAssign()` routine. The MATLAB C Math Library creates a numeric array large enough to accommodate the specified location or expands an existing array.

The following example is equivalent to the MATLAB statement, `A(2,2) = 17`. The C character string `"(?,?)"` specifies the format of the index subscript. For more information about array indexing, see Chapter 5.

```
mxArray *A = NULL;
```

```
mlfIndexAssign(&A,
               "(?,?)",            /* Index subscript format */
                mlfScalar(2),mlfScalar(2), /* subscripts     */
                mlfScalar(17));    /* value to be assigned   */

mxDestroyArray(A);
```

This call creates the array `A` and fills it with zeros before performing the assignment. The following output shows the array created by this code fragment.

```
    0    0
    0    17
```

Initializing a Numeric Array with Data

You can specify the value of elements in an array using `mlfIndexAssign()`. However, if you want to assign values to many elements in an array, this method can become tedious.

The fastest way to initialize a MATLAB array is to obtain a pointer to the data area in the `mxArray` data type and copy data to this location. You use the MATLAB API routine `mxGetPr()` to retrieve this pointer and copy your data to this location using the standard C `memcpy()` routine. The API also includes a routine, `mxGetPi()`, that lets you initialize the imaginary part of an array in the same way.

Note Make sure the data you are copying into the array will fit into the storage associated with the pointer returned by `mxGetPr()`. The MATLAB C Math Library will not grow or expand an array when you copy data directly into the `mxArray` data pointer.

The following example illustrates this procedure, which is a common MATLAB C Math Library programming idiom. Note that MATLAB API routines accept integer arguments.

```
int ndim = 3;
int dims[3]   = {3,3,2};
int bytes_to_copy = (3 * 3 * 2) * sizeof(double);
```

```
double data[] = { 1,4,7,2,5,8,3,6,9,10,13,16,11,14,17,12,15,18};

double *pr = NULL;
mxArray *A = NULL;

/* create the array */
mlfAssign(&A ,mxCreateNumericArray( ndim,
                                    dims,
                                    mxDOUBLE_CLASS,
                                    mxREAL));

/* get pointer to data in array  */
pr = mxGetPr(A);

/* copy data to pointer */
memcpy(pr, data, bytes_to_copy);

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This program displays the following output.

```
(:,:,1) =
   1   2   3
   4   5   6
   7   8   9

(:,:,2) =
  10  11  12
  13  14  15
  16  17  18
```

Column-Major Storage versus Row-Major Storage

It is important to note in the previous example that the MATLAB C Math Library stores its arrays in column-major order, unlike C, which stores arrays in row-major order. Static arrays of data that are declared in C and that initialize MATLAB C Math Library arrays must store their data in column-major order. For this reason, we recommend not using two-dimensional C language arrays to initialize a MATLAB array because the mapping of array elements from C to MATLAB can become confusing.

As an example of the difference between C's row-major array storage and MATLAB's column-major array storage, consider a 3-by-3 matrix filled with the numbers from one to nine.

```
1    4    7
2    5    8
3    6    9
```

Notice how the numbers follow one another down the columns. If you join the end of each column to the beginning of the next, the numbers are arranged in counting order.

To recreate this structure in C, you need a two-dimensional array.

```
static double square[][3] = {{1, 4, 7}, {2, 5, 8}, {3, 6, 9}};
```

Notice how the numbers are specified in row-major order; the numbers in each row are contiguous. In memory, C lays each number down next to the last, so this array might have equivalently (in terms of memory layout) been declared.

```
static double square[] = {1, 4, 7, 2, 5, 8, 3, 6, 9};
```

To a C program, these arrays represent the matrix first presented: a 3-by-3 matrix in which the numbers from one to nine follow one another in counting order down the columns.

However, if you initialize a 3-by-3 MATLAB `mxArray` structure with either of these C arrays, the results will be quite different. MATLAB stores its arrays in column-major order. MATLAB treats the first three numbers in the array as the first column, the next three as the second column, and the last three as the third column. Each group of numbers that C considers to be a row, MATLAB treats as a column.

To MATLAB, the C array above represents this matrix.

```
1    2    3
4    5    6
7    8    9
```

Note how the rows and columns are transposed.

To construct a matrix where the counting order proceeds down the columns rather than across the rows, the numbers need to be stored in the C array in column-major order.

```
static double square[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

This C array, when used to initialize a MATLAB array, produces the desired result.

Example Program: Creating Numeric Arrays (ex1.c)

This program creates two arrays and then prints them. The code demonstrates only one of the ways to create an array. Each of the numbered sections of code is explained in more detail below. You can find the code for this example in the `<matlab>/extern/examples/cmath` directory on UNIX systems and in the `<matlab>\extern\examples\cmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation.

```
    /* ex1.c */

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
1   #include "matlab.h"

2   static double real_data[] = { 1, 2, 3, 4, 5, 6 };
    static double cplx_data[] = { 7, 8, 9, 10, 11, 12 };

    int main()
    {
        /* Declare two matrices and initialize to NULL */
        mxArray *mat0 = NULL;
        mxArray *mat1 = NULL;

        /* Enable automated memory management */
        mlfEnterNewContext(0, 0);

        /* Create the matrices and assign data to them */
3       mlfAssign(&mat0, mlfDoubleMatrix(2, 3, real_data, NULL));
       mlfAssign(&mat1, mlfDoubleMatrix(3, 2, real_data, cplx_data));

        /* Print the matrices */
4       mlfPrintMatrix(mat0);
        mlfPrintMatrix(mat1);

        /* Free the matrices */
5       mxDestroyArray(mat0);
        mxDestroyArray(mat1);

        /* Disable automated memory management */
        mlfRestorePreviousContext(0, 0);

        return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include “matlab.h”. This file contains the declaration of the mxArray data structure and the prototypes for all the functions in the library. stdlib.h contains the definition of EXIT_SUCCESS.

2 Declare two static arrays of real numbers that are subsequently used to initialize matrices. The data in the arrays is interpreted by the MATLAB C Math Library in column-major order. The first array, real_data, stores the data for the real part of both matrices, and the second, cplx_data, stores the imaginary part of mat1.

3 Create two full matrices with mlfDoubleMatrix(). mlfDoubleMatrix() takes four arguments: the number of rows, the number of columns, a pointer to a standard C array of data to initialize the real part of the array and a pointer to a C array of data to initialize the imaginary part, if present. mlfDoubleMatrix() allocates an mxArray structure and storage space for the elements of the matrix, initializing each entry in the matrix to the values specified in the initialization arrays. The first matrix, mat0, does not have an imaginary part. The second matrix, mat1, has an imaginary part. mat0 has two rows and three columns, and mat1 has three rows and two columns.

4 Print the matrices. mlfPrintMatrix() calls the installed print handler, which in this example is the default print handler. See the section Chapter 8 for details on writing and installing a print handler.

5 Free the matrices.

The program produces this output:

```
    1     3     5
    2     4     6

1.0000 + 7.0000i   4.0000 +10.0000i
2.0000 + 8.0000i   5.0000 +11.0000i
3.0000 + 9.0000i   6.0000 +12.0000i
```

Sparse Matrices

The MATLAB C Math Library includes routines to create and manipulate sparse arrays. Sparse matrices provides a more efficient storage format for two-dimensional numeric arrays with few non-zero elements. Only two-dimensional numeric arrays can be converted to sparse storage format.

This table lists the MATLAB C Math Library routines used to create sparse matrices and perform some basic operations on them. The sections that follow provide more detail about using these routines. For more detailed information about using sparse arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the online *MATLAB C Math Library Reference*.

Table 3-2: Sparse Matrix Routines

To ...	Use ...
Create a sparse matrix	`mlfSparse()`
Convert a sparse matrix into a full matrix	`mlfFull()`
Replace nonzero sparse matrix elements with ones	`mlfSpones()`

Replace nonzero sparse matrix elements with random numbers	`mlfSprand()`
	`mlfSprandn()`
	`mlfSprandnsym()`

Import from external sparse matrix format	`mlfSpconvert()`
Create a sparse identity matrix	`mlfSpeye()`
Extract a band or diagonal group of elements from a matrix and create a sparse matrix	`mlfSpdiags()`
Determine the number of nonzero elements in a numeric matrix.	`mlfNnz()`

Table 3-2: Sparse Matrix Routines (Continued)

To ...	Use ...
Determine if a matrix has any nonzero elements or if all elements are nonzero	`mlfAny()` or `mlfAll()`
Determine the amount of storage allocated for the nonzero elements of a sparse matrix	`mlfNzmax()`
Apply a function to all the nonzero elements of a sparse matrix	`mlfSpfun()`

Creating a Sparse Matrix

To create a sparse matrix, call the MATLAB C Math Library `mlfSparse()` routine. Using this routine, you can create sparse arrays in two ways:

- By converting an existing array to sparse format
- By specifying the data and the location of the data in the sparse array

Converting an Existing Matrix into Sparse Format

To create a sparse matrix from a standard numeric array, use the `mlfSparse()` routine. `mlfSparse()` converts the numeric array into sparse storage format.

To illustrate, the following code fragment creates a 12-by-12 identity matrix. Of the 144 elements in this matrix, only 12 elements have nonzero values. In full format, all 144 are allocated storage. When this identity matrix is converted to sparse matrix format, only the 12 nonzero elements have storage allocated for them.

In the example, the `NULL`s included in the call to `mlfSparse()` represent optional arguments. The following section describes these optional arguments.

```
mxArray *A = NULL;
mxArray *B = NULL;

/* Create the identity matrix  */
mlfAssign(&A, mlfEye(mlfScalar(12),NULL));
mlfPrintMatrix(A);
```

```
/* Convert the identity matrix to sparse format  */
mlfAssign(&B,mlfSparse(A,NULL,NULL,NULL,NULL,NULL));
mlfPrintMatrix(B);

mxDestroyArray(A); /* Free bound arrays */
mxDestroyArray(B);
```

This code displays the identity matrix in full and sparse formats.

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

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

Creating a Sparse Matrix from Data

You can create a sparse matrix, specifying the value and location of all the nonzero elements when you create it. Using `mlfSparse()`, you specify as arguments:

- Two vectors, `i` and `j`, that specify the row and column subscripts of the nonzero elements.
- One vector, `s`, containing the real or complex data you want to store in the sparse matrix. Vectors `i`, `j` and `s` should all have the same length.
- Two scalar arrays, `m` and `n`, that specify the dimensions of the sparse matrix to be created.
- An optional scalar array that specifies the maximum amount of storage that can be allocated for this sparse array.

The following code example illustrates how to create a sparse 8-by-7 sparse matrix from data. This call specifies a single value, `9`, for all the nonzero elements of the sparse matrix which is replicated in all nonzero elements by scalar expansion. To see the pattern formed by this sparse matrix, see the output of this code which follows.

```
static double row_subscripts[] = { 3, 4, 5, 4, 5, 6 };
static double col_subscripts[] = { 4, 3, 3, 5, 5, 4 };

mxArray *i = NULL;
mxArray *j = NULL;
mxArray *S = NULL;

mlfAssign(&i, mlfDoubleMatrix(1, 6, row_subscripts, NULL));
mlfAssign(&j, mlfDoubleMatrix(1, 6, col_subscripts, NULL));

mlfAssign(&S, mlfSparse(i,    /* Row subscripts     */
                        j,    /* Column subscripts  */
                        mlfScalar(9),   /* Data  */
                        mlfScalar(8),
                        mlfScalar(7),
                        NULL));

mlfPrintMatrix(S);
mlfPrintMatrix(mlfFull(S));

mxDestroyArray(i);
mxDestroyArray(j);
mxDestroyArray(S);
```

This code produces the following output.

```
(4,3)   9
(5,3)   9
(3,4)   9
(6,4)   9
(4,5)   9
(5,5)   9

0    0    0    0    0    0    0
0    0    0    0    0    0    0
0    0    0    9    0    0    0
0    0    9    0    9    0    0
0    0    9    0    9    0    0
0    0    0    9    0    0    0
0    0    0    0    0    0    0
0    0    0    0    0    0    0
```

Converting a Sparse Matrix to Full Matrix Format

You can convert a sparse matrix to a full format matrix by using the `mlfFull()` routine. The previous example nested a call to this routine in `mlfPrintMatrix()` to show the pattern created by the values in the sparse matrix.

```
mlfPrintMatrix(mlfFull(F));
```

Evaluating Arrays for Sparse Storage

To see if a MATLAB array is a good candidate for sparse format storage, determine the number of nonzero elements in an array, using the `mlfNnz()` routine. The following code fragment creates the same 12-by-12 identity matrix, shown in the example in “Converting an Existing Matrix into Sparse Format” on page 3-18, and then prints out the number of nonzero elements in the matrix.

```
mxArray *I = NULL;

/* Create the array */
mlfAssign(&I, mlfEye(mlfScalar(12),NULL));

/* Determine the number of nonzero elements */
mlfPrintMatrix(mlfNnz(I));
```

```
mxDestroyArray(I);
```

This code outputs the number of nonzero elements in the 12-by-12 identity matrix: 12.

Character Arrays

The MATLAB C Math Library also includes routines to create and manipulate character arrays. One-dimensional character arrays are also called *strings*. Multidimensional character arrays are also called *arrays of strings*. In an array of strings, each string must be the same length. The routines that create arrays of strings use blanks to pad the strings to the same length. In a cell array of strings, individual strings can be different lengths. For information about cell arrays, see "Cell Arrays" on page 3-28.

This table lists the MATLAB C Math Library routines used to create character arrays and perform some basic operations on them. The sections that follow provide more detail about using these routines. For more detailed information about using character arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the online *MATLAB C Math Library Reference*.

Table 3-3: Character Array Routines

To ...	Use ...
Create a character array	`mxCreateString()`
Create a character array from a numeric array	`mlfChar()`
Convert a character array to its underlying numeric representation.	`mlfDouble()`

Concatenate character strings into a multidimensional, blank-padded character array	`mlfStr2mat()`
	`mlfStrcat()`
	`mlfStrvcat()`

Convert an array of blank-padded character strings into a cell array of strings	`mlfCellstr()`
Concatenate each character string in a cell array of strings into a multidimensional array of strings.	`mlfChar()`

Table 3-3: Character Array Routines (Continued)

To ...	Use ...
Remove extra blank characters from individual rows in a character array.	`mlfDeblank()`
Display a character string.	`mlfDisp(), mlfPrintMatrix()`
Convert a number to its string representation, specifying format.	`mlfNum2Str()`
Convert an array of integers into a string array.	`mlfInt2str()`
Convert character string to a numeric array.	`mlfStr2num()`

Creating MATLAB Character Arrays

Wherever you pass a character string to a MATLAB C Math Library routine, the string must be a MATLAB character string array, not a standard C null-terminated character string. MATLAB represents characters in 16-bit, Unicode format.

Using Explicit Character Array Creation Routines

The easiest way to create a MATLAB character string is with the MATLAB API routine `mxCreateString()`. You pass this routine a standard C character string as an argument, delimited by double quotation marks. (In the MATLAB interpreted environment, strings are delimited by single quotation marks.)

```
mxArray *A = NULL;

mlfAssign(&A, mxCreateString("my string"));

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This code produces the following output.

```
my string
```

Converting Numeric Arrays to Character Arrays

To convert a numeric array into a character array, use the `mlfChar()` routine. The following code creates an array containing the ASCII codes for each character in “my string” and then call `mlfChar()` to convert this numeric array into a MATLAB character array.

```
mxArray *i;
static double ASCII_codes[] = {109,121,32,115, \
                               116,114,105,110,103 };

mlfAssign(&i, mlfDoubleMatrix(1, 9, ASCII_codes, NULL));

mlfPrintMatrix(mlfChar(i,NULL));

mxDestroyArray(A);
```

This code produces the following output.

```
my string
```

To convert this character array back into its underlying numeric representation in double precision format, use the `mlfDouble()` routine.

Creating Multidimensional Arrays of Strings

You can create a multidimensional array of MATLAB character strings; however, each string must have the same length. The MATLAB C Math Library routines that create arrays of character strings pad the strings with blanks to make them all a uniform length.

Note To create a multidimensional character array without padding, use cell arrays. For more information, see “Cell Arrays” on page 3-28.

To illustrate, the following code fragment creates a two-dimensional array character from two strings of different lengths.

```
mxArray *A = NULL;
mxArray *D1 = NULL;
mxArray *D2 = NULL;
```

```
/* create array of strings */
mlfAssign(&A, mlfChar(mxCreateString("my string"),
                      mxCreateString("my dog"),
                      NULL));

mlfPrintMatrix(A);

/* Get the size of each dimension of the array of strings */
mlfSize(mlfVarargout(&D1,&D2,NULL),A,NULL);

/* Print out the size of each dimension */
mlfFprintf(mlfScalar(1),
           mxCreateString("Resulting array is %d-by-%d.\n"),
           D1,
           D2,
           NULL);

mxDestroyArray(A);
mxDestroyArray(D1);
mxDestroyArray(D2);
```

As the following output illustrates, `mlfChar()` creates an 2-by-9 character array. This indicates that it added three blanks characters to the string "`my dog`" to make it the same length as "`my string`" .

```
my string
my dog

Resulting array is 2-by-9.
```

You can also use the `mlfStrcat()`, `mlfStrvcat()` and `mlfStr2mat()` routines to group strings into a multidimensional character array. For more information about these routines, see the online *MATLAB C Math Library Reference*.

Accessing Individual Strings in an Array of Strings

You can manipulate multidimensional character arrays just as you would a standard MATLAB numeric array. For example, to extract an individual string from a character array, use standard MATLAB indexing syntax. Note, however, that a string extracted from a character array in this fashion may contain blank padding characters. To remove these blank characters from the character array, use the `mlfDeblank()` routine.

The following code fragment extracts the string "`my dog`" from the character array, `A`, created in the previous section. This is equivalent to the MATLAB statement `B = A(2,:)`. The example displays the size of the extracted array, `B`, before and after removing blanks. Note that the index format string is passed as a standard C string; it does not need to be a MATLAB character array.

```
mxArray *B = NULL;

mlfAssign(&B,mlfIndexRef(A,
                        "(?,?)",  /* index format string  */
                        mlfScalar(2), /* row two   */
                        mlfCreateColonIndex()));

mlfPrintMatrix(mlfSize(NULL,B,NULL));
mlfPrintMatrix(mlfSize(NULL,mlfDeblank(B),NULL));

mlfPrintMatrix(B);

mxDestroyArray(B);
```

This code produces the following output.

```
1     9
1     6
```

Cell Arrays

MATLAB cell arrays provide a way to group together a collection of dissimilar MATLAB arrays.

This table lists the MATLAB C Math Library routines used to create cell arrays and perform basic operations on them. The sections that follow provide more detail about using these routines. For more detailed information about using cell arrays, see *Using MATLAB*. For more detailed information about any of the library routines, see the online *MATLAB C Math Library Reference*.

Table 3-4: Cell Array Routines

To ...	Use ...
Create a multidimensional array of empty cells	`mlfCell()`
Convert an array of blank-padded character strings into a cell array of strings	`mlfCellstr()`
Create a cell array by concatenating existing arrays	`mlfCellhcat()`
Convert a structure into a cell array	`mlfStruct2Cell()`
Convert a numeric array into a cell array	`mlfNum2cell()`
View the contents of each cell in a cell array	`mlfCelldisp()`

Creating Cell Arrays

The MATLAB C Math Library allows you to create cell arrays by:

- Using a cell array creation function
- Using a cell array conversion function
- Concatenating existing arrays
- Assigning a value to an element in a cell array

Using the Cell Array Creation Routine

You can create an array of empty cells using the `mlfCell()` routine. The following code fragment creates a 2-by-3-by-2 array of empty cells.

```
mxArray *A = NULL;

mlfAssign(&A, mlfCell(mlfScalar(2),
                      mlfScalar(3),
                      mlfScalar(2),
                      NULL));

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This code produces the following output.

```
(:,:,1) =
    []     []     []
    []     []     []

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

MATLAB uses brackets to indicate cell array elements and `[]` represents an empty cell. You can then assign values to cells in the array using assignment. For an example of assigning a value to a cell in a cell array, see “Using Assignment to Create Cell Arrays” on page 3-31.

Using Cell Array Conversion Routines

You can also create cell arrays by converting other MATLAB arrays into cell arrays. The MATLAB C Math Library includes routines that convert a numeric array into a cell array, `mlfNum2cell()`, or a structure into a cell array, `mlfStruct2cell()`.

The following code fragment creates a numeric array, using `mlfOnes()`, and converts it into a cell array using the `mlfNum2cell()` routine.

```
mxArray *N = NULL;
mxArray *C = NULL;
```

```
/*  Create a numeric array  */
mlfAssign(&N, mlfOnes(mlfScalar(2),mlfScalar(3),NULL));
mlfPrintMatrix(N);

/* Convert it into a cell array */
mlfAssign(&C, mlfNum2cell(N,NULL));

mlfPrintMatrix(C);

mxDestroyArray(N);
mxDestroyArray(C);
```

In this output, the brackets indicate that each element in the numeric array has been placed into a cell in the cell array.

```
1    1    1
1    1    1

[1]  [1]  [1]
[1]  [1]  [1]
```

The brackets indicate cell array elements.

Using Concatenation to Create Cell Arrays

You can group existing MATLAB arrays into a cell array by concatenation. In MATLAB, you use the {} (braces) operator to create cell arrays through concatenation. For example, you can use the following syntax in MATLAB to concatenate arrays into a cell array:

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

» B= 'my string'
B =
my string

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

```
» D = { A B C }
D =
    [1x10 double]    'my string'    [2x3 double]
```

To create the same cell array through concatenation in a C program, use the MATLAB C Math Library `mlfCellhcat()` routine. This routine performs the same function as `{}`, the MATLAB cell concatenation operator.

```
mxArray *A = NULL;
mxArray *B = NULL;
mxArray *C = NULL;
mxArray *D = NULL;
static double data[] = { 1, 4, 2, 5, 3, 6 };

mlfAssign(&A, mlfColon(mlfScalar(1),mlfScalar(10),NULL));
mlfAssign(&B, mxCreateString("my string"));
mlfAssign(&C, mlfDoubleMatrix(2, 3, data, NULL));

mlfAssign(&D, mlfCellhcat(A,B,C,NULL));
mlfPrintMatrix(D);

mxDestroyArray(A);
mxDestroyArray(B);
mxDestroyArray(C);
mxDestroyArray(D);
```

To see the output from this code fragment, see “Displaying the Contents of a Cell Array” on page 3-32.

Using Assignment to Create Cell Arrays

You can also create a cell array by assigning a value to a location in a cell array, using the `mlfIndexAssign()` routine. The MATLAB C Math Library creates a cell array large enough to accommodate the specified location or expands an existing array. For more information about using indexing with cell arrays, see Chapter 5.

The following example is equivalent to the MATLAB statement, `A(2,2) = {17}`. Note the use of curly braces in the index subscript format string: `"{?,?}"`. This syntax indicates you want to create a cell array. Also note that the index subscript format string may be passed as a standard C character string; it does not need to be a MATLAB character array.

```
mxArray *A = NULL;

mlfIndexAssign(&A,
               "{?,?}",   /* index subscript format string */
               mlfScalar(2),   /* index value  */
               mlfScalar(2),   /* index value  */
               mlfScalar(17)); /* value to be assigned  */

mlfPrintMatrix(A);

mxDestroyArray(A);
```

The following output shows the cell array created by this code fragment.

```
[]    []
[]    [17]
```

Displaying the Contents of a Cell Array

When you use `mlfPrintMatrix()` or `mlfDisp()` to display a cell array, the MATLAB C Math Library displays the type of array stored in each cell but it does not display the contents of the cell (except for string arrays and scalar values). To view the contents of each cell, you must use the `mlfCelldisp()` routine.

The `mlfCelldisp()` routine supports a second, optional argument which specifies the text string used to identify each cell in the output. In the example, the character "`D`" is passed to `mlfCelldisp()` as its second argument. This appears in the output in the line that prefixes each cell, such as `"D{1} ="`. If you do not specify this second argument, `mlfCelldisp()` uses the text string `"ans"`, as in `"ans{1} ="`.

The following code fragment creates a cell array and prints out the cell array using both `mlfPrintMatrix()` and `mlfCelldisp()`.

```
mxArray *A = NULL;
mxArray *B = NULL;
mxArray *C = NULL;
mxArray *D = NULL;
static double data[] = { 1, 4, 2, 5, 3, 6 };
```

```
mlfAssign(&A, mlfColon(mlfScalar(1),mlfScalar(10),NULL));
mlfAssign(&B, mxCreateString("my string\n"));
mlfAssign(&C, mlfDoubleMatrix(2, 3, data, NULL));

mlfAssign(&D, mlfCellhcat(A,B,C,NULL));

mlfPrintMatrix(mxCreateString("The mlfPrintMatrix() output:"));
mlfPrintMatrix(D);

mlfPrintMatrix(mxCreateString("The mlfCelldisp() output:"));
mlfCelldisp(D,mxCreateString("D"));

mxDestroyArray(A);
mxDestroyArray(B);
mxDestroyArray(C);
mxDestroyArray(D);
```

This code produces the following output:

```
The mlfPrintMatrix() output:
    [2x3 double]    [1x5 double]    'my string'

The mlfCellDisp() output:
D{1} =
      1     2     3     4     5     6     7     8     9    10

D{2} =
my string

D{3} =
     1     2     3
     4     5     6
```

MATLAB Structures

A MATLAB structure can be thought of as a one-dimensional cell array in which each cell is assigned a name. These named cells are called fields. You can create multidimensional arrays of structures but all the structures in the array must have the same fields.

This table lists the MATLAB C Math Library routines used to create structures and perform basic operations on them. The sections that follow provide more detail about using these routines. For more detailed information about using structures, see *Using MATLAB*. For more detailed information about any of the library routines, see the online *MATLAB C Math Library Reference*.

Table 3-5: MATLAB Structure Routines

To ...	Use ...
Create a structure an initialize it with values.	`mlfStruct()`
Convert a cell array into a structure.	`mlfCell2struct()`
Determine the names of the fields in a structure.	`mlfFieldnames()`
Determine if a string is the name of a field in a structure.	`mlfIsfield()`
Access the contents of a field in a structure.	`mlfGetfield()`
Specify the value of a field in a structure.	`mlfSetfield()`
Remove a field from each structure in an array of structures.	`mlfRmfield()`

Creating Structures

The MATLAB C Math Library allows you to create structures by:

- Using a structure creation routine
- Using a structure conversion routine
- Assigning a value to an element in a structure

Using a Structure Creation Routine

You can create a structure using the `mlfStruct()` routine. This routine lets you define the fields in the structure and assign a value to each field. For example, the following code fragment creates a structure that contains two fields, a text string and a scalar value.

```
mxArray *A = NULL;

mlfAssign(&A, mlfStruct(mxCreateString("name"),   /* Field */
                        mxCreateString("John"),   /* Value */
                        mxCreateString("number"), /* Field */
                        mlfScalar(311),           /* Value */
                        NULL));

mlfPrintMatrix(A);

mlfDestroyArray(A)
```

This code produces the following output:

```
  name: 'John'
number: 311
```

Because the `mlfStruct()` routine can accept a varying number of input arguments, you must terminate the argument list with a `NULL`.

Creating Multidimensional Arrays of Structures

The `mlfstruct()` routine defines the fields and values in a single instance of a structure, in effect a 1-by-1 structure array. To create a multidimensional array of structures, use MATLAB indexing to assign a value to a field in a structure with an index other than `(1,1)`. MATLAB will extend the array of structures to accommodate the location specified. For more information about using assignment with structures, see “Using Assignment to Create Structures” on page 3-37.

Using a Structure Conversion Routine

You can also create structures by converting an existing MATLAB cell array into a structure, using the `mlfCell2struct()` routine. This example creates a cell array to be converted, and a second cell array that specifies the names of the fields in the structure. You pass these two cell arrays, along with the dimensions of the structure array, as arguments to `mlfCell2struct()`.

```
mxArray *C = NULL; /* cell array to convert */
mxArray *F = NULL; /* cell array of field names */
mxArray *S = NULL; /* structure   */

/* create cell array to be converted */
mlfAssign(&C, mlfCellhcat(mxCreateString("tree"),
                          mlfScalar(37.4),
                          mxCreateString("birch"),
                          NULL));

/* create cell array of field names */
mlfAssign(&F, mlfCellhcat(mxCreateString("category"),
                          mxCreateString("height"),
                          mxCreateString("name"),
                          NULL));

/* convert cell array to structure */
mlfAssign(&S,mlfCell2struct(C,F,mlfScalar(2)));

mlfPrintMatrix(C);
mlfPrintMatrix(S);

mlfDestroyArray(C);
mlfDestroyArray(F);
mlfDestroyArray(S);
```

Note that, because `mlfCellhcat()` accepts a variable number of input arguments, you must terminate the input argument list with a `NULL`.

This code generates the following output.

```
'tree'    [37.4000]    'birch'

category: 'tree'
  height: 37.4000
    name: 'birch'
```

Using Assignment to Create Structures

You can also create a structure by assigning a value to a location in a structure, using the `mlfIndexAssign()` routine. The MATLAB C Math Library creates a structure (or array of structures) large enough to accommodate the location specified by the index string. For more information about structure indexing, see Chapter 5.

The following example is equivalent to the MATLAB statement, `A(2) = struct('name','jim','number',312)`.

```
mxArray *A = NULL;

mlfIndexAssign(&A,
               "(?)",       /* Index subscript format string  */
               mlfScalar(2), /* Index subscript value */
               mlfStruct(mxCreateString("name"),  /* Field */
                         mxCreateString("Jim"),   /* Value */
                         mxCreateString("number"),/* Field */
                         mlfScalar(312),          /* Value */
                         NULL));

mlfPrintMatrix(A);

mlfDestroyArray(A);
```

The following output shows the structure created by this code fragment.

```
1x2 struct with fields
    name
    number
```

For more detailed information about using `mlfIndexAssign()` to assign values to fields in a structure, see Chapter 5.

Performing Common Array Programming Tasks

The following sections describes common array programming tasks that you must perform for all types of MATLAB array.

- “Allocating and Freeing MATLAB Arrays”
- “Displaying MATLAB Arrays”
- “Determining Array Type”
- “Determining Array Size”
- “Determining Array Shape”

Allocating and Freeing MATLAB Arrays

When you create a MATLAB array, using any of the array creation mechanisms, the MATLAB C Math Library allocates the storage for the array. The responsibility for freeing the allocated storage is shared between you and the library automated memory management facility.

When automated memory management is enabled, all the arrays returned by library routines are temporary. That is, when these arrays are passed to another library routine, that routine destroys the array before returning. This capability allows you to nest, or *compose*, calls to MATLAB C Math Library routines without causing memory leaks. The calls to the `mlfScalar()` routine which are nested in the examples in this chapter illustrate routine nesting.

If your application needs to use an array several times, you must assign the array to an `mxArray` pointer variable to make it persist. This is called *binding* the array to a variable. You use the `mlfAssign()` routine to bind an array to an array pointer variable. Arrays returned as output arguments are bound to variables automatically by the library. *Any array you bind to a variable you must explicitly free.* Use the `mxDestroyArray()` routine to free bound arrays.

All the code examples in this chapter assume that the MATLAB C Math Library automated memory management is enabled. For information about enabling memory management, see Chapter 4.

Displaying MATLAB Arrays

To output an array to the display, use the MATLAB C Math Library `mlfPrintMatrix()` routine. This routine can display all types of MATLAB arrays of any dimension. The following code fragment creates a 2-by-2 matrix filled with ones and then uses `mlfPrintMatrix()` to output the array to the screen.

```
mxArray *A = NULL;

mlfAssign(&A,mlfOnes(mlfScalar(2),mlfScalar(2),NULL));

mlfPrintMatrix(A);

mxDestroyArray(A);
```

This code produces the following output.

```
1    1
1    1
```

When used with a cell array, the `mlfPrintMatrix()` output includes the type and size of the array stored in each cell but not the data in the array (except for scalar arrays and character arrays). To view the data in each cell in a cell array, you must use the `mlfCelldisp()` routine. See "Displaying the Contents of a Cell Array" on page 3-32 for more information.

Formatting Output

You can also create formatted array output using the `mlfFprintf()` routine. This routine allows you to create your own output formats and applies these formats to all the elements in a MATLAB array. For example, if you specify the format string `"%d"`, `mlfFprintf()` prints out each element in an array as an integer.

Note Do not confuse `mlfFprintf()` with `mlfPrintf()`. `mlfFprintf()` can format MATLAB arrays; `mlfPrintf()` does not. `mlfPrintf()` is the same as the standard C `printf()` routine except that it directs output to the MATLAB print handler. For more information about print handlers, see Chapter 8, "Handling Errors and Writing a Print Handler.".

The following code prints the 2-by-2 array, `A`, created in the previous section to the display.

```
mxArray *A = NULL;
```

```
mlfAssign(&A,mlfOnes(mlfScalar(2),mlfScalar(2),NULL));

mlfFprintf(mlfScalar(1),                     /* stdout  */
           mxCreateString("Array A = %d\n"), /* format string */
           A,                                /* array   */
           NULL);
```

This code produces the following output, illustrating how `mlfFprintf()` applies the format to each element in an array.

```
array A = 1
array A = 1
array A = 1
array A = 1
```

Determining Array Type

The MATLAB C Math Library includes several routines that allow you to determine the type of an array. Each routine tests for a particular type of array and returns 1 if the array being tested matches the indicated type and 0 (zero) otherwise.

Table 3-6: Array Type Routines

Array Type	Routine
Numeric array	`mlfIsnumeric()`
Character array	`mlfIschar()`
Sparse array	`mlfIssparse()`
Cell array	`mlfIscell()`
Cell array of strings	`mlfIscellstr()`
Structure	`mlfIsstruct()`

As an example, the following code uses the `mlfIsnumeric()` routine to test if a 2-by-2 array of ones is a numeric array.

```
mxArray *A = NULL;
mxArray *B = NULL;
```

```
/* Create two-dimensional array */
mlfAssign(&A, mlfOnes(mlfScalar(2),mlfScalar(2),NULL));

/* Determine if array is numeric */
mlfAssign(&B, mlfIsnumeric(A));

mlfFprintf(mlfScalar(1),                     /* stdout  */
           mxCreateString("Isnumeric returns %d.\n"),
           B,                                /* array   */
           NULL);


mxDestroyArray(A);
mxDestroyArray(B);
```

Because the array created is numeric, this code produces the following output:

```
Isnumeric returns 1.
```

Determining Array Size

To determine the size of an array, use the `mlfSize()` routine. The `mlfSize()` routine returns a row vector containing the dimensions of the array. This code example creates a 2-by-3-by-2 array and displays the size vector returned by `mlfSize()`.

```
mxArray *A = NULL;
mxArray *dims = NULL;

/* Create three-dimensional array */
mlfAssign(&A, mlfOnes(mlfScalar(2),
                      mlfScalar(3),
                      mlfScalar(2),
                      NULL));

/* Determine size of array */
mlfAssign(&dims, mlfSize(NULL,A,NULL));

/*  Display size vector    */
mlfPrintMatrix(mxCreateString("The size of the array is \n"));
mlfPrintMatrix(dims);
```

```
mxDestroyArray(A);
mxDestroyArray(dims);
```

This code produces the following output:

```
The size of the array is
     2  3  2
```

Length of a Single Dimension

You can also get the size of one particular dimension of an array by specifying the dimension as an input argument. To get the length of the longest dimension of a multidimensional array, use the `mlfLength()` routine. You can also use this routine to determine the size of a vector.

Dimension Return Values

The `mlfSize()` routine can optionally return each dimension in a separate array. You specify these arrays as output arguments passed to the `mlfVarargout()` routine. For more information about calling library routines that take variable number of input and output arguments, see Chapter 6.

The following code returns the dimensions in three output arguments: `dim1`, `dim2`, and `dim3`. When used with output arguments, you do not need to bind the return value from `mlfSize()` to a variable. The routine binds the return values to the output arguments specified. As with all bound arrays, you must explicitly free them.

```
mlfSize(mlfVarargout(&dim1,&dim2,&dim3,NULL), C, NULL);

mxDestroyArray(dim1);
mxDestroyArray(dim2);
mxDestroyArray(dim3);
```

If the array has more dimensions than the number of output arguments specified, the last output argument contains the product of the remaining dimensions.

Determining Array Shape

To determine the number of dimensions of an array, use the mlfNdims() routine. This code uses mlfNdims() to get the number of dimensions of a 2-by-3-by-2 array.

```
mxArray *A = NULL;
mxArray *ndims = NULL;

/* Create three-dimensional array */
mlfAssign(&A, mlfOnes(mlfScalar(2),
                      mlfScalar(3),
                      mlfScalar(2),
                      NULL));

/* Determine dimensions  */
mlfAssign(&ndims, mlfNdims( A ));

mlfFprintf(mlfScalar(1),
           mxCreateString("The array has %d dimensions"),
           ndims,
           NULL);

mxDestroyArray(A);
mxDestroyArray(ndims);
```

This code outputs the value 3, indicating that the array C is a three-dimensional array.

4

Managing Array Memory

Introduction

This section shows you how to manage array memory in the functions that you write with the MATLAB C Math Library. The section:

- Explains automated memory management and compares it to explicit memory management.
- Provides step-by-step guidelines for writing a routine that uses automated memory management, including a template you can use as a starting point. Be sure to look over the example program later in this chapter.
- Outlines the restrictions for calling between functions that use automated memory management and functions that do not.
- Describes how to replace MATLAB's allocation and deallocation routines with your own routines.

Automated Versus Explicit Memory Management

Routines in the MATLAB C Math Library return a pointer to a newly allocated `mxArray`. The MATLAB C Math Library provides two ways for you to manage this array memory:

- Explicit memory management
- Automated memory management

Explicit Memory Management

In explicit memory management, you must assign the returned `mxArray` to an array variable before passing it to another function. You use the assignment operator (=) to assign the return value from a library function to an `mxArray*` variable or by passing an `mxArray*` variable (uninitialized or initialized to `NULL`) as an output argument to a library function.

In this memory management scheme:

- You *cannot* nest calls to library functions. You must declare array variables for all your arrays.
- You must explicitly delete the array you create to avoid memory leaks.

See “Example Without Automated Memory Management” on page 4-26 for an example of explicit memory management.

Automated Memory Management

With automated memory management, you let the library manage array memory between the initialization of arrays and the deletion of array. To use automated memory management, you must:

1. Declare and initialize local array variables at the beginning of your function.
2. Use the local array variables to perform the work of your function, assigning values to them, and passing them as input or output arguments to other functions.
3. Destroy local array variables at the end of your function.

Temporary and Bound Arrays

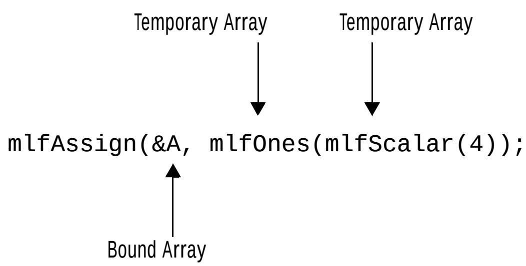
Automated memory management distinguishes between two types of arrays:

- *Temporary* arrays
- *Bound* arrays

Understanding the definition of the temporary and bound states for an array will help you understand:

- Why you can nest calls to library functions
- Why you need to call `mlfAssign()` rather than use the assignment operator (=) for assignments
- Why you need to follow the rules for writing functions presented in "Using Automated Memory Management" on page 4-8

This diagram illustrates how library functions return temporary arrays and how arrays become bound if they are assigned to an array variable (`mxArray *`).



Definition of a Temporary Array. MATLAB C Math Library functions return pointers to newly allocated `mxArrays` as their return values. The library marks these arrays as *temporary* arrays.

Key Behavior for a Temporary Array. When you pass a temporary array as an input argument to another library function, that function deletes the temporary array before it returns. You do not have to delete it yourself. This behavior allows you to embed calls to library functions as arguments to other library functions without leaking memory.

Definition of a Bound Array. To make an array persist, you must assign it to a variable by using the function `mlfAssign()` or pass an array variable as an output array argument to a library function. The MATLAB C Math Library marks the array as a *bound* array.

Key Behavior for a Bound array. When you pass a bound array as an input argument to another library function, the array still exists when the function completes. Bound arrays are not automatically deleted; you must explicitly delete the array by calling `mxDestroyArray()`.

Comparison of Memory Management Schemes

To see the benefits of automated memory management, compare a sample MATLAB code and the equivalent C code under both memory management schemes.

MATLAB code:

```
z = sin(x) + cos(y)
```

C code with explicit memory management:

```
mxArray *temp_x, *temp_y;

temp_x = mlfSin(x);
temp_y = mlfCos(y);
z = mlfPlus(tempx, temp_y);
mxDestroyArray(temp_x);
mxDestroyArray(temp_y);
```

C code using automated memory management.

```
mlfAssign(&z, mlfPlus(mlfSin(x), mlfCos(y)));
```

Benefits of Automated Memory Management

Using automated memory management makes your code more like MATLAB. The code is easier to write, easier to read, and far less likely to leak memory. Because you can embed calls to library functions as function arguments (also called nested functions calls), code that requires many lines in explicit memory management can be written as a single line of code under automated memory management. Using automated memory management, you don't need to declare `mxArray*` variables to store *temporary* values, or explicitly delete those temporary arrays.

Compatibility Between Memory Management Schemes

In versions of the MATLAB C Math Library prior to Version 2.0, explicit memory management was the only memory management technique. It is still available, and existing code is compatible with the routines that use automated memory management. See “Restrictions” on page 4-28 to learn about the compatibility between the two styles of managing memory.

Note You must choose either automated memory management or explicit memory management to manage array memory in your application.

Avoiding Memory Leaks in Your Functions

Follow these recommendations to avoid the memory leaks.

1. Never call a library function without assigning the array it returns to an array variable (by calling `mlfAssign()`) or without embedding the call as an argument to a library function.

   Memory leak:

   ```
   mlfSin(X);
   ```

   The array returned by `mlfSin()` is not bound to a variable and never freed.

2. Never assign a value to an array variable without subsequently deleting the array.

   Memory leak:

   ```
   void func(mxArray *y)
   {
   mxArray *x;

   mlfEnterNewContext(0,1,y);
   mlfAssign(&x, mlfSin(y));
   mlfRestorePreviousContext(0,1,y);
   }
   ```

   You must pair each `mxArray*` declaration with a call to `mxDestroyArray()`.

3 Never use the assignment operator to assign array values.

Unexpected termination of your program:

x is a temporary array. If x is subsequently passed as an input argument to a function, that function will delete x. Any subsequent reference to x will cause your program to crash.

```
x = mlfSin(y);                       /* x is temporary. */
a = mlfPlus(x, mlfScalar(1));        /* x is deleted. */
b = mlfPlus(a, x);                   /* Program crashes. */
```

Using Automated Memory Management

You *must* follow a set procedure when you write a function that conforms to the rules of automated memory management. The pattern is the same for each function.

1. Declare the interface for your function:
   - Does it return an array?
   - Does it take any array output arguments?
   - Does it take any array input arguments?
2. Initialize local array variables to `NULL` or to valid arrays:

   Library functions, including `mlfAssign()`, require that output arguments are initialized to `NULL` or to a valid array.
3. Call `mlfEnterNewContext()` to turn on automated memory management for your function and to change the status of temporary input arrays to bound arrays:

   Pair this call with a call to `mlfRestorePreviousContext()` at the end of your function.
4. Perform the work of your function, using `mlfAssign()`, rather than the assignment operator (=), to assign values to arrays.
5. Free any bound array variables by calling `mxDestroyArray()`:

   However, do not destroy the return value from your function.
6. Call `mlfRestorePreviousContext()` to reset the state for each input array the value it had on entering the function and then to delete any temporary arrays:

   Pass the same arguments that you passed to `mlfEnterNewContext()`.
7. Call `mlfReturnValue()` to make the array you are returning temporary and then return the temporary array:

Function Template

You can use the template code below as the basis for writing functions that use the library's automated memory management. You can find the general function template and a main routine template in <matlab>/extern/examples/cmath/mem_mgt_func_template.c and mem_mgt_main_template.c respectively where <matlab> represents the top-level directory of your installation.

This function template takes one array output argument, two array input arguments, and returns an array. Your functions will, of course, vary the number of input and output arguments.

Notice how mlfEnterNewContext() and mlfRestorePreviousContext() operate on the array arguments passed to FunctionName. mlfReturnValue() manipulates the array returned from FunctionName.

```
mxArray *FunctionName(mxArray **output_arg1, mxArray *input_arg1,
                    mxArray *input_arg2)
{
    mxArray *local_return_value = NULL;
    mxArray *local_var1 = NULL;
    mxArray *local_var2 = NULL;
    mlfEnterNewContext(1,2, output_arg1, input_arg1,
                        input_arg2);

    /* Perform the work of the function. */
    /* .... */

    /* Note: Don't destroy local_return_value */
    mxDestroyArray(local_var1);
    mxDestroyArray(local_var2);
    mlfRestorePreviousContext(1,2,output_arg1, input_arg1,
             input_arg2);

return mlfReturnValue(local_return_value);
}
```

Main Routine Template

The template for the main() routine is different from the general template because main() does not take any array input or output arguments or return an array. Passing `0` as an argument to `mlfEnterNewContext()` and `mlfRestorePreviousContext()` indicates that `main()` has no output and input array arguments. A call to `mlfReturnValue()` is not required.

```
int main()
{
    /* Initialize variables. */
     mlfEnterNewContext(0,0);

/* Perform the work of main(). */
     mlfRestorePreviousContext(0,0);

     return(EXIT_SUCCESS);
}
```

Enabling Memory Management

A function that uses automated memory management must begin with a call to `mlfEnterNewContext()`. Typically, you place the call after the declaration and initialization of local variables. You *must* call `mlfEnterNewContext()` before the first call to `mlfAssign()`.

The call to `mlfEnterNewContext()` signals that MATLAB C Math Library automated memory management is in effect for the function. `mlfEnterNewContext()` operates on the output and input array arguments passed to your function. It ensures that the memory allocated for those arrays, whether temporary or bound, persists for the duration of the function and will not be destroyed by your function or any function that it calls.

Pass these arguments to `mlfEnterNewContext()` in the order listed. You do *not* need to terminate the list of arguments with a `NULL` argument.

1. The number (`int nout`) of array *output* arguments declared by your function. Specify 0 if there are no array output arguments declared (in the same way the `main()` template function does on “Main Routine Template” on page 4-9).

   The template function declares one output argument.

2 The number (`int nin`) of array *input* arguments declared by your function. Specify 0 if there are no array input arguments declared (in the same way the `main()` template function does).

The template function on “Function Template” on page 4-9 declares two input arguments.

3 The array *output* arguments (`mxArray **`) themselves, in the order declared for the function.

In the template, `output_arg1` is passed.

4 The array *input* arguments (`mxArray *`) themselves, in the order declared for the function.

In the template, `input_arg1` and `input_arg2` are passed.

Note You only list `mxArray**` and `mxArray*` arguments. For example, if a function takes an argument of type `char*` or `int`, do not include it in the count of output and input arguments or in the list of the arguments themselves.

Creating Bound Arrays

`mlfEnterNewContext()` changes the state of temporary input arrays from temporary to bound, enabling them to persist for the duration of the function. If they are passed as input arguments to other functions, they are passed as bound arrays and not deleted.

Note `mlfEnterNewContext()` recognizes when the current function is called from a function that does not use automated memory management. In that context, it ensures that the input arguments, which are all temporary arrays, are handled correctly and not subsequently deleted by `mlfRestorePreviousContext()`. Output arguments that do not point to `NULL` or to a valid array are also handled correctly.

Assigning Arrays to mxArray* Variables

Under automated memory management, you can assign a value to an mxArray* variable in two different ways:

- By calling mlfAssign() to assign the return value from a library function to an mxArray* variable.
- By passing a pointer to an initialized (to NULL or a valid array) array variable as an output argument to a function.

The mlfAssign() routine copies the array value from its second argument src (representing the righthand side of the assignment) to its first argument *dest (representing the lefthand side of the assignment). If src is a temporary array, mlfAssign() only copies the pointer without copying the array data. For example,

```
mlfAssign(&Y, mlfCos(X));
```

assigns the array returned by mlfCos() to Y, a pointer to an mxArray.

mlfAssign() marks the assigned array as a *bound* array. You are responsible for deleting the bound arrays that result from a call to mlfAssign().

Note Always call mlfAssign() when you want an array to persist. Do not use the assignment operator (=). Becoming accustomed to programming with mlfAssign() rather than the assignment operator (=) is the biggest adjustment you'll need to make when programming with automated memory management.

Assigning a Value to an Array Destroys Its Previous Value

If you assign a value to an array variable that already has a value, mlfAssign() destroys the variable's previous value before assigning the new value. You do not need to call mxDestroyArray() before calling mlfAssign(). For example, in these two statements,

```
mlfAssign(&c, mlfScalar(5));
mlfAssign(&c, mlfScalar(6));
```

mlfAssign() destroys the contents of c (the scalar array 5) before assigning the scalar array containing 6 to c.

Exception. Just as the MATLAB language preserves the value of an array passed as an input argument across a function call, mlfAssign() leaves an array value unchanged (does not make a copy) if the array is a *bound* (not temporary) input array argument on entry to the function. For example, given this function

```
mxArray *func(mxArray **a, mxArray *b)
```

and this call within the function

```
mlfAssign(&b, mlfScalar(5));
```

mlfAssign() modifies the value of b locally within the function. However, because b is an input argument, the call to mlfAssign() does not destroy the old value.

Assignment by Value

mlfAssign() implements assignment by value. When the array on the righthand side of the assignment (the second argument to mlfAssign()) is already bound to a variable, the array on the lefthand side receives a copy of that array. For example,

```
mxArray *A = NULL;
mxArray *B = NULL;

mlfAssign(&A, mlfRand(mlfScalar(4)));
mlfAssign(&B, A);
```

A and B point to two different arrays.

Note that the copy is actually a shared-data copy until the application requires two separate copies of the data. The MATLAB C Math Library supports full copy-on-write semantics.

Nesting Calls to Functions that Return Arrays

You can nest calls to library functions as arguments to other library functions. When you nest calls, the library deletes the array returned from the call for you. For example, when you call the library's indexing functions, you can embed the calls to mlfScalar() that define the index values.

```
mlfAssign(&B,
          mlfIndexRef(A, "(?,?)", mlfScalar(2), mlfScalar(2)));
```

The two calls to `mlfScalar()` each return a temporary array that the function `mlfIndexRef()` deletes just before it returns.

See “Using Automated Memory Management” on page 4-8, which explains the rules for writing functions so that they can be nested.

Deleting Bound Arrays

You must explicitly delete:

- Any array that you’ve bound to a variable by calling `mlfAssign()`
- Any array that you’ve passed as an output argument to a function

`mxDestroyArray()` destroys the array (`mxArray*`) passed to it. For example,

```
mxDestroyArray(A);
```

destroys array `A`.

`mxDestroyArray()` does handle a `NULL` argument. However, `mxDestroyArray()` does *not* reinitialize the `mxArray*` pointer passed to it to `NULL`. If you assign an array to an `mxArray*` variable and subsequently delete that array by calling `mxDestroyArray()`, then you *must* reinitialize the `mxArray*` variable to `NULL` before reassigning another array to that variable. If you follow the “Automated Memory Management” on page 4-3, then you avoid this awkward coding.

```
mlfAssign(&c, mlfScalar(5));
mxDestroyArray(c);
c = NULL;
mlfAssign(&c, mlfScalar(6));
```

Restoring the Previous Context

Each function that you write must close with a call to `mlfRestorePreviousContext()`. Place the call just before the `return` statement for your function.

The call operates on the output and input array arguments passed to your function. It ensures that the memory allocated for those arrays is restored to its state at the time of the function call.

Note You can't return from your function before calling `mlfRestorePreviousContext()`, and you can call `mlfRestorePreviousContext()` only once in your function.

Prototype:

```
void mlfRestorePreviousContext(int nout, int nin, ...);
```

Sample Call from Template:

```
mlfRestorePreviousContext(1,2,output_arg1, input_arg1, input_arg2);
```

Arguments to mlfRestorePreviousContext( )

`mlfRestorePreviousContext()` takes the number of output arguments, the number of input arguments, and a variable-length list of the actual output and input arguments to the function. You do *not* need to terminate the list of arguments with a `NULL` argument.

Pass the same arguments to `mlfRestorePreviousContext()` as you passed to `mlfEnterNewContext()`:

1. The number (`int nout`) of array output arguments declared by your function. Specify 0 if there are no array output arguments declared (in the same way the `main()` template function does).

   The template function declares one output argument.

2. The number (`int nin`) of array input arguments declared by your function. Specify 0 if there are no array output arguments declared (in the same way the `main()` template function does).

   The template function declares two input arguments.

3. The array output arguments (`mxArray **`) themselves, in the order declared for the function.

   In the template, `output_arg1` is passed.

4 The array input arguments (`mxArray *`) themselves, in the order declared for the function.

   In the template, `input_arg1` and `input_arg2` are passed.

Note You only list `mxArray**` and `mxArray*` arguments. For example, if a function takes an argument of type `char*` or `int`, do not include it in the count of output and input arguments or in the list of the arguments themselves.

What Happens to the Array Arguments?

`mlfRestorePreviousContext()` restores the state of the input arrays to their state at the time of the function call:

- Any array input argument that was temporary at the time of the function call becomes temporary again.
- Any array input argument that was bound at the time of the function call remains bound.

`mlfRestorePreviousContext()` then performs an important action: it deletes any input array arguments that are temporary.

Note `mlfRestorePreviousContext()` recognizes when the current function is called from a function that does not use automated memory management. In that call context, where all arrays are temporary, it does not delete any arrays.

Purpose of mlfRestorePreviousContext( )

This is the key step that allows a call to another function to be embedded as an input argument to your function. Your function deletes the temporary arrays passed to it, ensuring proper deletion of the memory for temporary arrays.

Returning an Array

Before you pass an array to the `return` statement in your function, you must pass that array to `mlfReturnValue()`. `mlfReturnValue()` makes the array a temporary array. Your function can therefore return a temporary array just like each function in the MATLAB C Math Library does.

Prototype:

```
mxArray *mlfReturnValue(mxArray *a);
```

Sample Call from Template:

```
return mlfReturnValue(local_return_value);
```

Note You cannot have multiple `return` statements in your function. You must code in a style that ends your function with a call to `mlfRestorePreviousContext()` followed by a single `return` statement, `return(mlfReturnValue(result))`.

Argument and Return for mlfReturnValue( )

Pass the array that your function returns (an `mxArray*`) to `mlfReturnValue()`. The array is a *bound* array when it is passed to `mlfReturnValue()` and is typically the result of an assignment made within the function or the value of an output argument set by a function call.

`mlfReturnValue()` makes the array a temporary array and returns the same array. You can nest the call to `mlfReturnValue()` in the `return` statement for your function.

You do not need to call `mlfReturnValue()` if you are writing a function that does not return a pointer to an array (in the same way that the `main()` template doesn't call `mlfReturnValue()`).

Note Do not pass an array input argument to `mlfReturnValue()`. Instead, use `mlfAssign()` to assign the array to a local variable first and then pass that local variable to `mlfReturnValue()`.

Changing Bound Arrays to Temporary Arrays

`mlfReturnValue()` changes the *bound* state of the array passed to it to *temporary*. You then pass that array to the `return` statement.

Handling Return Values

By marking the returned array temporary, you ensure that a call to your function can be nested as an argument to another function without leaking memory.

Example – Managing Array Memory (ex2.c)

This example program demonstrates how to write functions that use automated memory management. Apply this technique to every function you write.

This section presents an annotated example; “Example Without Automated Memory Management” on page 4-26 shows comparable code that does uses explicit memory management.

Each of the numbered sections of code is explained in more detail below. You can find the code for the example in `<matlab>/extern/examples/cmath/ex2.c` where `<matlab>` represents the top-level directory of your installation. See “Building Stand-Alone C Applications” in Chapter 2 for information on building the examples.

The example is split into two parts. (In a working program, both parts would be placed in the same file.) The first part includes header files, declares two file static variables, and defines a routine that demonstrates how the library manages array memory. The second section contains the main program.

```
    /* ex2.c */

    #include <stdio.h>
    #include <stdlib.h>     /* used for EXIT_SUCCESS */
    #include <string.h>
1   #include "matlab.h"

    static double real_data1[] = { 1, 2, 3, 4, 5, 6 };
    static double real_data2[] = { 6, 5, 4, 3, 2, 1 };

2   mxArray *Automated_Mem_Example(mxArray **z_out, mxArray *x_in,
                        mxArray *y_in)
    {
3       mxArray *result_local = NULL;
        mxArray *q_local = NULL;

4       mlfEnterNewContext(1, 2, z_out, x_in, y_in);

        /* In MATLAB: result = sqrt(sin(x) + cos(x)) */
5       mlfAssign(&result_local,
                  mlfSqrt(mlfPlus(mlfSin(x_in), mlfCos(x_in))));

        /* In MATLAB: q = sqrt(cos(y) - sin(y)) */
6       mlfAssign(&q_local,
                  mlfSqrt(mlfMinus(mlfCos(y_in), mlfSin(y_in))));

        /* In MATLAB: z = q * result - q^3 */
        mlfAssign(z_out,
                  mlfMinus(mlfTimes(q_local, result_local),
                      mlfPower(q_local, mlfScalar(3))));

7       mxDestroyArray(q_local);

8       mlfRestorePreviousContext(1, 2, z_out, x_in, y_in);

9       return mlfReturnValue(result_local);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Include header files. `matlab.h` declares the `mxArray` data structure and the prototypes for all the functions in the MATLAB C Math Library. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Define the interface for your function. This function, `Automated_Mem_Example()`, takes two inputs and returns two outputs. It has one return value, one output array argument (`mxArray **z_out`), and two input array arguments (`mxArray *x_in` and `mxArray *y_in`). The definition of the function follows the MATLAB C Math Library calling conventions where output arguments precede input arguments. You can write functions that follow these calling conventions or implement your own.

The body of `Automated_Mem_Example()` performs three calculations that illustrate how the library manages the memory allocated for the arrays. The first two calculations operate on the two input arguments; the third on two local arrays that store the results from the previous calculations. The function returns one result in the output argument and the other result as the return value from the function.

3 Initialize the local variables to `NULL` or to valid arrays. `result_local` will be used to store the function's return value. `q_local` will be used in a local calculation.

Note All MATLAB C Math Library functions, including `mlfAssign()`, require that output arguments are initialized to `NULL` or point to a valid array.

---

4 Call `mlfEnterNewContext()` to create a new memory context for your function. `mlfEnterNewContext()` is always paired with a call to `mlfRestorePreviousContext()`, which appears at the end of the function.

The first integer argument, `1`, specifies the number of output array arguments (not including the return value) passed to `Automated_Mem_Example()`; the second integer argument, `2`, specifies the number of input array arguments. The arrays themselves (`mxArray**` for output arguments, `mxArray*` for input arguments), `z_out`, `x_in`, and `y_in`, are passed next, in the same order as they were passed to the function. You do not need to terminate the list with `NULL`. Note that `mlfEnterNewContext()` can take any number of arguments.

`mlfEnterNewContext()` changes the state of any temporary input arrays from temporary to bound enabling them to persist for the duration of the function. If they are passed as input arguments to other functions, they are passed as bound arrays.

5 The first calculation passes the input argument `x_in` to the MATLAB C Math Library functions `mlfSin()` and `mlfCos()`. These two calls return *temporary* arrays, which are passed to another library function, `mlfPlus()`. The temporary array returned from the call to `mlfPlus()` is then passed to the library function `mlfSqrt()`.

   In this series of nested calls *only* the return from `mlfSqrt()` is assigned to a variable via the `mlfAssign()` function. That array, `result_local`, becomes a *bound* array. The temporary arrays returned from `mlfSin()` and `mlfCos()` and passed to `mlfPlus()` are automatically deleted by `mlfPlus()`; `mlfSqrt()` deletes the temporary array returned from `mlfPlus()`.

6 Calls to `mlfAssign()` now appear as frequently as the assignment operator (=). The array on the left-hand side of the assignment (the first argument to `mlfAssign()`) becomes a bound array and persists. You must explicitly delete it; the library does not.

   These calculations again illustrate the automated memory management provided by the library. If you do not want to keep the array returned from a library function, just nest the call as an argument to a function. The return from the call (a temporary array) will be deleted by the other function.

   In these calculations, `q_local` and `z_out`, are each the targets of an assignment statement. Both are marked as *bound* arrays. Since `q_local` is a local variable, it must be explicitly deleted within this function. `z_out` is an output array argument that will be explicitly deleted in the calling function.

   See “Assigning Arrays to mxArray* Variables” on page 4-12 to learn how `mlfAssign()` determines whether to delete the contents of the target argument and whether to make a shared data copy of the source argument.

7 Free any local, bound array variables. If you've assigned (by calling `mlfAssign()`) a value to any local variable that is an array, you must destroy it by calling `mxDestroyArray()`.

Note The exception is a local, bound array that is the return value from the function. Do not call `mxDestroyArray()` on your return value.

---

8 Call `mlfRestorePreviousContext()` to restore the memory context that existed prior to the function call. Supply the same arguments that you passed to `mlfEnterNewContext()`.

Before ending your function and returning a value, you must call `mlfRestorePreviousContext()` to reestablish the state (temporary or bound) of input arguments prior to the function call. Any input argument that becomes temporary is then deleted by `mlfRestorePreviousContext()`. If you fail to call `mlfRestorePreviousContext()`, your program will leak memory.

If an input array is bound when it is passed to a function, it will never be destroyed automatically by that function or any function that it calls.

9 Return a temporary array. `mlfReturnValue()` marks its argument as temporary. You must pass any `mxArray*` that your function returns to `mlfReturnValue()` before passing it to the `return` statement. If you forget this step, the automated memory management of the library will be disrupted for the variable that you return, and memory will leak.

The second section of code contains the main program.

```
    int main()
    {
1       mxArray *mat0 = NULL;
        mxArray *output_array = NULL;
        mxArray *result_array = NULL;

2       mlfEnterNewContext(0, 0);

3       mlfAssign(&mat0, mlfDoubleMatrix(2, 3, real_data1, NULL));

4       mlfAssign(&result_array,
                  Automated_Mem_Example(&output_array,
                          mat0,
                          mlfDoubleMatrix(2, 3, real_data2, NULL)));

5       mlfPrintf("mat0:\n");
        mlfPrintMatrix(mat0);
        mlfPrintf("output_array:\n");
        mlfPrintMatrix(output_array);
        mlfPrintf("result_array:\n");
        mlfPrintMatrix(result_array);

6       mxDestroyArray(mat0);
        mxDestroyArray(result_array);
        mxDestroyArray(output_array);

7       mlfRestorePreviousContext(0, 0);

        return(EXIT_SUCCESS);
    }
```

The numbers in the list below correspond to the numbered sections of code above:

1 Initialize any local array variables to `NULL`. MATLAB C Math Library functions that use automated memory management require that you initialize any output arguments to `NULL` or a valid array. For example, before you call `mlfAssign()`, you must initialize its first argument, an output argument, to `NULL` or a valid array. Note that if you pass a pointer to a valid array, the contents of that array will be deleted before the assignment to the output argument takes place.

2 Pass 0 as the first and second argument to mlfEnterNewContext(), indicating that the main() routine does not have any array output or input arguments.

3 Call mlfAssign() to assign the return from the MATLAB C Math Library function mlfDoubleMatrix() to the array variable mat0 (a pointer to an mxArray). mlfDoubleMatrix() returns a 2-by-3 temporary array initialized with the data contained in the static C array, real_data1. mat0 becomes a bound array and will persist until explicitly deleted.

4 Call mlfAssign() to assign the return from a user-defined function to an array variable. Automated_Mem_Example() uses the automated memory management provided by the MATLAB C Math Library functions.

   mat0 is a bound array when it is passed as an input argument to Automated_Mem_Example(). The return value from mlfDoubleMatrix() is a temporary array. After the call to Automated_Mem_Example(), mat0 still exists; the temporary array has been deleted by Automated_Mem_Example().

5 Print the arrays. mlfPrintMatrix() returns void rather than an array (mxArray *). Both mlfPrintMatrix() and mlfPrintf() use the installed print handling routine to display their output. Because this example does not register a print handling routine, all output goes through the default print handler. The default print handler uses printf. See the section "Defining a Print Handler" in Chapter 8 for details on registering print handlers.

6 Free each local, bound array by calling mxDestroyArray(). Note that mxDestroyArray() can handle a NULL argument if you inadvertently pass a pointer to an array that has already been destroyed and set to NULL.

7 End the function by calling mlfRestorePreviousContext(). Pass 0 as the first and second argument to indicate that main() has no input and output arguments.

Output

When run, the program produces this output.

```
mat0:
     1     3     5
     2     4     6

output_array:
  -0.0714           -0.0331 + 0.2959i  -0.9461 + 1.5260i
  -0.6023           -1.2631 + 1.2030i        0 + 0.6181i

result_array:
   1.1755                 0 + 0.9213i        0 + 0.8217i
   0.7022                 0 + 1.1876i   0.8251
```

Example Without Automated Memory Management

The function `Explicit_Mem_Example()` performs the same calculations as `Automated_Mem_Example()` in the previous example. Compare the use of temporary variables, nonnested calls to the MATLAB C Math Library functions, and calls to `mxDestroyArray()`. It contains twenty-six lines of code compared to `Automated_Mem_Example()`'s nine lines.

Note You can still code to this interface. The MATLAB C Math Library continues to support it. However, you cannot call `Explicit_Mem_Example()` from a function that uses automated memory management. The routine that calls `Explicit_Mem_Example()`, whether it is `main()` or another function, cannot include calls to `mlfEnterNewContext()` and `mlfRestorePreviousContext()`.

If automated memory management were in effect, the calls to the library functions in `Explicit_Mem_Example()` could unexpectedly *delete* the temporary arrays passed to them as input arguments.

```
mxArray *Explicit_Mem_Example(mxArray **z_out, mxArray *x_in,
                    mxArray *y_in)
{
    mxArray *result_local, *q_local;
    mxArray *temp1, *temp2, *temp3;

    /* In MATLAB: r = sqrt(sin(x) + cos(x)) */
```

```
    temp1 = mlfSin(x_in);
    temp2 = mlfCos(y_in);
    temp3 = mlfPlus(temp1, temp2);
    result_local = mlfSqrt(temp3);

    mxDestroyArray(temp1);
    mxDestroyArray(temp2);
    mxDestroyArray(temp3);

    /* In MATLAB: q = sqrt(cos(y) - sin(y)) */
    temp1 = mlfCos(y_in);
    temp2 = mlfSin(y_in);
    temp3 = mlfMinus(temp1, temp2);
    q_local = mlfSqrt(temp3);

    mxDestroyArray(temp1);
    mxDestroyArray(temp2);
    mxDestroyArray(temp3);

    /* In MATLAB: z = q * r - q^3 */
    temp1 = mlfScalar(3);
    temp2 = mlfPower(q_local, temp1);
    temp3 = mlfTimes(q_local, result_local);
    *z_out = mlfMinus(temp3, temp2);

    mxDestroyArray(temp1);
    mxDestroyArray(temp2);
    mxDestroyArray(temp3);

    mxDestroyArray(q_local);

    return result_local;
}
```

Restrictions

Routines that use the MATLAB C Math Library's automated memory management can only call other routines written *with* automated memory management. This includes any of the MATLAB C Math Library Version 2.0 functions.

Routines that use automated memory management *cannot* call functions that use explicit memory management, for example, routines written with the MATLAB C Math Library prior to Version 2.0.

Note Functions written *with or without* automated memory management can call your function.

If you write a new function that does *not* use the library's automated memory management, you can:

- Call a function that *you* wrote with the MATLAB C Math Library prior to Version 2.0
- Call a new function that you've written *with* or *without* automated memory management
- Call MATLAB C Math Library Version 2.0 functions

Note Functions written *without* automated memory management can call your function; functions written with automated memory management cannot.

Recommendation

Though the explicit approach seems to offer flexibility, you lose the benefits of the library's automated memory management.

Mixing memory management styles is not recommended. Choose one style or the other. Use the `mlfEnterNewContext()` and `mlfRestorePreviousContext()` pair and `mlfAssign()` for all your functions or none of your functions.

Note Some functions have changed between Version 1.2 and Version 2.0 of the library. You must update any calls in existing code to the group of functions that have a different prototype for Version 2.0 of the MATLAB C Math Library. See the release notes, `release.txt`, in the `<matlab>/extern/examples/cmath` directory of your installation for a list of these functions.

Replacing Allocation and Deallocation Routines

The MATLAB C Math Library calls mxMalloc to allocate memory and mxFree to free memory. These routines in turn call the standard C runtime library routines malloc and free.

If your application requires a different memory management implementation, you can register your allocation and deallocation routines with the MATLAB C Math Library by calling the function mlfSetLibraryAllocFcns().

```
void mlfSetLibraryAllocFcns(calloc_proc calloc_fcn,
                            free_proc free_fcn,
                            realloc_proc realloc_fcn,
                            malloc_proc malloc_fcn);
```

You must write four functions whose addresses you then pass to mlfSetLibraryAllocFcns():

- calloc_fcn is the name of the function that mxCalloc uses to perform memory allocation operations. The function that you write must have the prototype:

  ```
  void * callocfcn(size_t nmemb, size_t size);
  ```

  Your function should initialize the memory it allocates to 0 and should return NULL for requests of size 0.

- free_fcn is the name of the function that mxFree uses to perform memory deallocation (freeing) operations. The function that you write must have the prototype:

  ```
  void freefcn(void *ptr);
  ```

  Make sure your function handles NULL pointers. free_fcn(0) should do nothing.

- realloc_fcn is the name of the function that mxRealloc uses to perform memory reallocation operations. The function that you write must have the prototype:

```
void * reallocfcn(void *ptr, size_t size);
```

This function must grow or shrink memory. It returns a pointer to the requested amount of memory, which contains as much as possible of the previous contents.

- `malloc_fcn` is the name of the function to be called in place of `malloc` to perform memory allocation operations. The prototype for your function must match:

```
void * mallocfcn(size_t n);
```

Your function should return `NULL` for requests of size 0.

Refer to the *MATLAB Application Program Interface Reference* online help for more detailed information about writing these functions.

Indexing into Arrays

Introduction

MATLAB provides access to elements in arrays, cell arrays, and structures through indexing. Using MATLAB indexing syntax, you can access individual elements, modify elements or delete elements in an array, cell array, or structure. For example, `A(3,1)` accesses the first element in row 3 of matrix `A`.

The MATLAB C Math Library provides the same indexing functionality as the MATLAB interpreter but through a different mechanism. Instead of an indexing operator, the MATLAB C Math Library provides the following indexing functions.

Function	Description
`mlfIndexRef()`	Read one or more values from an array, cell array, or structure.
`mlfIndexAssign()`	Change one or more values in an array, cell array, or structure.
`mlfIndexDelete()`	Remove one or more elements from an array, cell array, or structure.

This section:

- Provides an overview of indexing terminology
- Describes how to call indexing functions for one-dimensional, n-dimensional, and logical subscripts
- Describes how to assign values to array elements using indexing
- Describes how to delete array elements using indexing
- Describes how to index into cell arrays
- Describes how to index into structure arrays

The section ends with a comparison of MATLAB and Math library indexing syntax.

Terminology

These two diagrams illustrate the terminology used in this chapter.

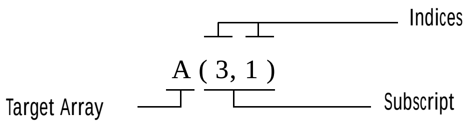


Figure 5-1: From the MATLAB Perspective

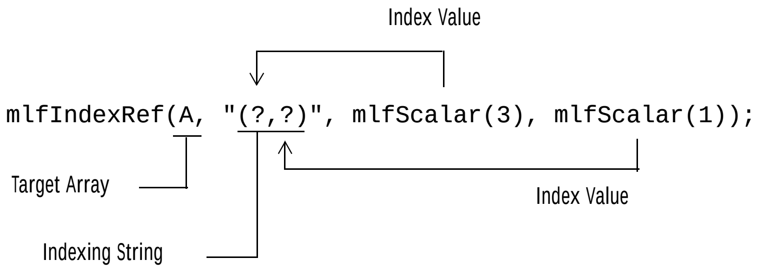


Figure 5-2: From the MATLAB C Math Library Perspective

Indexing Functions

An array subscript consists of one or more indices passed as `mxArray *` arguments to one of the indexing functions. For example, the two-dimensional indexing expression

```
mlfIndexRef(A, "(?,?)", mlfScalar(3), mlfScalar(1))
```

applies the subscript `(3,1)` to `A` and returns the element at row three, column one. `mlfIndexRef(A, "(?)", mlfScalar(9))`, a one-dimensional indexing expression, returns the ninth element of array `A`.

Note The indexing functions follow the MATLAB convention for array indices: indices begin at one rather than zero.

Both standard indexing and cell array indexing take numeric arguments, one argument for each dimension of the array being indexed into. Structure indexing uses only the name of the structure field.

Note Standard indexing can be used with all three types of data Cell array indexing can only be used on cell arrays and structure indexing only on structures. You can combine, for example, standard indexing and structure indexing on a structure.

The indexing functions in the MATLAB C Math Library support N-dimensional standard, cell array, and structure indexing.

An index `mxArray` argument can contain a scalar, vector, matrix, or the result from a call to the special function `mlfCreateColonIndex()`.

- A scalar subscript selects a scalar value.
- A subscript with vector or matrix indices selects a vector or matrix of values.
- The `mlfCreateColonIndex()` index, which loosely interpreted means "all," selects, for example, all the columns in a row or all the rows in a column.

You can also use the `mlfColon()` function, which is patterned after the MATLAB colon operator, to specify a vector subscript. For example, `mlfColon(mlfSclar(1), mlfScalar(10), NULL)` specifies the vector `[ 1 2 3 4 5 6 7 8 9 10 ]`.

Note You cannot index into an array with more dimensions than the array has, although you can use fewer dimensions.

Note `for`-loops provide an easy model for thinking about indexing. A one-dimensional index is equivalent to a single `for`-loop; a two-dimensional index is equivalent to two nested `for`-loops. The size of the subscript determines the number of iterations of the `for`-loop. The value of the subscript determines the values of the loop iteration variables.

Array Storage

MATLAB stores each array as a column of values regardless of the actual dimensions. This column consists of the array columns, appended top to bottom. For example, MATLAB stores

```
A = [2 6 9; 4 2 8; 3 0 1]
```

as

```
    2
    4
    3
    6
    2
    0
    9
    8
    1
```

Accessing `A` with a single subscript indexes directly into the storage column. `A(3)` accesses the third value in the column, the number 3. `A(7)` accesses the seventh value, 9, and so on.

If you supply more subscripts, MATLAB calculates an index into the storage column based on the array's dimensions. For example, assume a two-dimensional array like `A` has size `[d1 d2]`, where `d1` is the number of rows in the array and `d2` is the number of columns. If you supply two subscripts `(i,j)` representing row-column indices, the equivalent one-dimensional index is

```
(j-1)*d1+i
```

Given the expression `A(3,2)`, MATLAB calculates the offset into `A`'s storage column as `(2-1)*3+3`, or `6`. Counting down six elements in the column accesses the value `0`.

This storage and indexing scheme also extends to multidimensional arrays. You can think of an N-dimensional array as a series of "pages," each of which is a two-dimensional array. The first two dimensions in the N-dimensional array determine the shape of the pages, and the remaining dimensions determine the number of pages.

In a three- (or higher) dimensional array, for example, MATLAB iterates over the pages to create the storage column, again appending elements column-wise. You can think of three-dimensional arrays as “books,” with a two-dimensional array on each page. The term *page* is used frequently in this document to refer to a two-dimensional array that is part of a larger N-dimensional array.

Labeling the dimensions past three is more difficult. You can imagine shelves of books for dimension 4, rooms of shelves for dimension 5, libraries of rooms for dimension 6, etc. This document rarely uses an array of dimension greater than three or four, although MATLAB and the MATLAB C Math Library handle any number of dimensions that doesn’t exceed the amount of memory available on your computer.

For example, consider a 5-by-4-by-3-by-2 array `C`.

MATLAB displays C as

```
page(1,1) =

     1     4     3     5
     2     1     7     9
     5     6     3     2
     0     1     5     9
     3     2     7     5

page(2,1) =

     6     2     4     2
     7     1     4     9
     0     0     1     5
     9     4     4     2
     1     8     2     5

page(3,1) =

     2     2     8     3
     2     5     1     8
     5     1     5     2
     0     9     0     9
     9     4     5     3

page(1,2) =

     9     8     2     3
     0     0     3     3
     6     4     9     6
     1     9     2     3
     0     2     8     7

page(2,2) =

     7     0     1     3
     2     4     8     1
     7     5     8     6
     6     8     8     4
     9     4     1     2

page(3,2) =

     1     6     6     5
     2     9     1     3
     7     1     1     1
     8     0     1     5
     3     2     7     6
```

MATLAB stores C as

```
1
2
5
0
3
4
1
6
1
2
3
7
3
5
7
5
9
2
9
5
6
7
0
9
1
2
1
0
4
8
4
4
1
4
2
2
9
5
2
5
2
2
5
0
9
2
5
1
9
4
⋮
```

Again, a single subscript indexes directly into this column. For example, `C(4)` produces the result

```
ans =

     0
```

If you specify two subscripts `(i,j)` indicating row-column indices, MATLAB calculates the offset as described above. Two subscripts always access the first page of a multidimensional array, provided they are within the range of the original array dimensions.

If more than one subscript is present, all subscripts must conform to the original array dimensions. For example, `C(6,2)` is invalid, because all pages of `C` have only five rows.

If you specify more than two subscripts, MATLAB extends its indexing scheme accordingly. For example, consider four subscripts `(i,j,k,l)` into a four-dimensional array with size `[d1 d2 d3 d4]`. MATLAB calculates the offset into the storage column by

```
(l–1)(d3)(d2)(d1)+(k–1)(d2)(d1)+(j–1)(d1)+i
```

For example, if you index the array `C` using subscripts (3,4,2,1), MATLAB returns the value 5 (index 38 in the storage column).

In general, the offset formula for an array with dimensions $[d_1\ d_2\ d_3\ \ldots\ d_n]$ using any subscripts $(s_1\ s_2\ s_3\ \ldots\ s_n)$ is:

$$(s_n-1)(d_{n-1})(d_{n-2})\ldots(d_1)+(s_{n-1}-1)(d_{n-2})\ldots(d_1)+\ldots+(s_2-1)(d_1)+s_1$$

Because of this scheme, you can index an array using any number of subscripts. You can append any number of 1s to the subscript list because these terms become zero. For example, `C(3,2,1,1,1,1,1,1)` is equivalent to `C(3,2)`.

Calling the Indexing Functions

Each indexing function requires at least three arguments; `mlfIndexAssign()` requires at least four. The first argument is the array to which the indexing operation is being applied. Since both `mlfIndexAssign()` and `mlfIndexDelete()` modify the array, the first argument to these functions must be an `mxArray**`; as `mlfIndexRef()` does not modify the array, its first argument is an `mxArray*`.

The second argument is a string describing the indexing operation. This string uses a simplification of the MATLAB indexing syntax; `()`, `{}` and `.field` (depending on what type of indexing you're doing) are required, but the actual values that would appear in a MATLAB index operation are replaced by `?`'s in the MATLAB C Math Library. For example, the MATLAB expression `x{3}(2,4,2).color` (a combination of cell array, standard, and structure indexing) results in the following string: `"{?}(?,?,?).color"`.

The third and subsequent arguments are the values to use in place of the `?`'s in the string. These values must be `mxArray*`'s and are very often the result of a call to `mlfScalar()`, which creates an `mxArray*` from a double-precision floating-point number or a integer.

When calling `mlfIndexAssign()`, the last argument in the list is the source array that contains the values to write into the target array. Note that the source array must be exactly the same size as the subset of the target array specified by the indexing expression in the second argument and subsequent arguments.

Refer to the online *MATLAB C Math Library Reference* for more detail on the interface for the three functions.

Specifying the Target Array

Each indexing function takes a target array as its first argument. The subscript is applied to this array:

- For `mlfIndexRef()`, the first argument is the array that you want to extract elements from.
- For `mlfIndexAssign()`, the first argument is the array that you want to change elements of (be assigned to).

- For `mlfIndexDelete()`, the first argument is the array that you want to delete elements from.

Note `mlfIndexRef()` takes an `mxArray*`; `mlfIndexAssign()` and `mlfIndexDelete()` take `mxArray**` as their first argument.

Specifying the Index String

You pass an indexing string as the second argument to an indexing function. An indexing string is always surrounded by `""`. For example, the MATLAB indexing expression `A(2,1)` is written like this in the MATLAB C Math Library.

```
mlfIndexRef(A, "(?,?)", mlfScalar(2), mlfScalar(1))
```

`"(?,?)"` is the indexing string that specifies a two-dimensional index. The question mark, `?`, is a placeholder for each index value.

- If you're indexing into a regular array, use parentheses, `()`, to enclose the subscript.
- If you're indexing into a cell array, use braces, {}, to enclose the subscript.

Some more sample indexing strings:

`"(?,?,?,?)"`: standard indexing
`"{?}"`: cell array indexing
`"(?,?).y{?}"`: combined standard, structure, and cell array indexing

Table 5-1: Elements of Index String Syntax

Syntax Element	Definition	Example
`( )`	Encloses an array subscript.	`"(?,?)"`
`{ }`	Encloses a cell array subscript.	`"{?,?}"`
`,`	Separates dimensions of the subscript	`"(?,?,?)"`

Table 5-1: Elements of Index String Syntax (Continued)

Syntax Element	Definition	Example
`?`	Placeholder for a single array index value.	`"(?)"`
`.`*field*	Indicates a field in a structure	`("?.score")`

What an Indexing String Specifies

When you specify an indexing string, you provide the following information to the indexing functions:

- Number of dimensions in the subscript

  For example, the single `?` in `"(?)"` indicates a one-dimensional subscript. The three `?`'s in `"(?,?,?)"` indicates a three-dimensional subscript.
- Type of indexing

  For example, the parentheses in `"(?,?)"` indicate array indexing. The braces in `"{?,?}"` indicate that you are accessing the contents of a cell in a cell array.
- Which field in a structure you're accessing

  `.field` indicates you're accessing a field within a structure.

An indexing string does *not* specify:

- The values of the indices themselves

  The `?` is a placeholder for actual values. The values are specified as subsequent `mxArray*` arguments passed to the indexing functions.
- Nested subscripts

  Each `?` is a placeholder for a single array index.

Complex Indexing Expressions

In MATLAB, you can write complicated indexing expressions. For example, this MATLAB expression

```
B{3}(7).bfield(2,1)
```

combines cell array, standard, and structure indexing. The expression first selects the third element of cell array B; this third element must be an array. From this array it selects the seventh element, which must be a structure with at least one field, named bfield. From that structure it selects the array stored in the bfield field, and then the element at position (2,1) within that array.

In the MATLAB C Math Library, you can specify this entire indexing operation as a single string: "{?}(?).bfield(?,?)". Passing this string to any of the MATLAB C Math Library indexing functions selects that location for reading, writing, or deletion.

In the MATLAB C Math Library, the expression becomes

```
mlfIndexRef(B, "{?}(?).bfield(?,?)",
            mlfScalar(3), mlfScalar(7),
            mlfScalar(2), mlfScalar(1));
```

Nesting Indexing Operations

In MATLAB, you can nest indexing operations; when you do, the results of the inner indexing operation supply the index values for the outer index operation. Because the MATLAB C Math Library represents MATLAB indexing operations with calls to mlfIndexRef(), you can recreate the MATLAB behavior in the library by nesting calls to mlfIndexRef() inside one another.

For example, the MATLAB expression

```
x(y(4)) = 3
```

becomes

```
mlfIndexAssign(&x, "(?)", mlfIndexRef(y, "(?)", mlfScalar(4)),
                 mlfScalar(3));
```

The MATLAB expression

```
D = A(foo(1,B(2,3)), bar(4,C(:)))
```

becomes

```
mlfAssign(&D,
      mlfIndexRef(A,"(?,?)",
            foo(mlfScalar(1),
              mlfIndexRef(B,"(?,?)",mlfScalar(2),mlfScalar(3))),
            bar(mlfScalar(4),
```

```
                mlfIndexRef(C,"(?)",mlfCreateColonIndex())));
```

Specifying the Values for Indices

Because the second argument, the index string, only describes the types of operations to be performed and does not contain the actual subscript values, you must pass these values separately to the indexing functions. Following the indexing string argument, you pass a list of pointers to `mxArrays`. Each array contains the value of an index in your subscript(s).

For example, the two calls to `mlfScalar()` in the following indexing expression pass the values for the indices in the two-dimensional subscript `(2,1)`. If `A` were an array with more than two dimensions, you could specify more than two dimensions in the index string, and pass more than two index values to `mlfIndexRef()`.

```
mlfIndexRef(A, "(?,?)", mlfScalar(2), mlfScalar(1))
```

The indexing functions apply the subscript to the target array. Each function constructs the subscript based on the content of the indexing string. The indexing functions count the number of expressions that are delimited by commas within each parenthesized, `()`, or bracketed, `{}`, subscript within the indexing string to determine the structure of the subscript(s) and the number of `mxArray*` index arguments to expect.

Note Do *not* supply `NULL` to terminate the list of arguments passed to an indexing function. Each function detects the end of the argument list by counting the number of arguments indicated by the indexing string itself.

Specifying a Source Array for Assignments

`mlfIndexAssign()` requires one more argument than the other two indexing functions: a pointer to an `mxArray` that contains the new values for the target array. Pass the source array after the `mxArray*` arguments that specify the values for the subscript. Note that this source array must be exactly the same size as the subset of the target array specified by the indexing expression.

Assumptions for the Code Examples

The C code included in the following sections demonstrates how to perform indexing with the MATLAB C Math Library. For the most part, each example only presents the call to an indexing function. As you read the examples, assume that the code relies on declarations, assignments, and deletions that follow these conventions:

- Automated memory management is in effect. The functions that contain this code would begin with a call to `mlfEnterNewContext()` and end with calls to `mlfRestorePreviousContext()` and `mlfReturnValue()`. Assignments are made by calling `mlfAssign()`.
- The source arrays are created using the `mlfDoubleMatrix()` function. For example, this code creates matrix `A`:

  ```
  static double A_array_data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  mxArray *A;
  mlfAssign(&A, mlfDoubleMatrix(3, 3, A_array_data, NULL));
  ```

  See "Example Program: Creating Numeric Arrays (ex1.c)" in Chapter 3 for a complete example of how to use this function.
- Matrix `A`, which is used throughout the examples, is equal to:

  ```
  1 4 7
  2 5 8
  3 6 9
  ```

  Note how the value of each element in `A` is equal to that element's position in the column-major enumeration order. For example, the third element of `A` is the number 3 and the ninth element of `A` is the number 9.
- Nested calls to `mlfScalar()` create the arrays that contain the indices. Because automatic memory management is in effect, these scalar arrays are automatically deleted by the library because they are temporary arrays.
- Each `mxArray` that is the target of an assignment must be deleted after the program finishes with it.

  ```
  mxDestroyArray(A);
  ```

- Many of the examples use the `mlfHorzcat()` and `mlfVertcat()` functions to create the vectors and matrices that are used as indices. `mlfHorzcat()`

concatenates its arguments horizontally; `mlfVertcat()` concatenates its arguments vertically.

Refer to the online *MATLAB C Math Library Reference* for more information on `mlfScalar()`, `mlfCreateColonIndex()`, `mxCreateDoubleMatrix()`, `mxGetPr()`, `mlfHorzcat()`, and `mlfVertcat()`.

One-Dimensional Indexing

This section:

- Provides an overview of one-dimensional indexing
- Describes how to extract a single element with a one-dimensional scalar index
- Describes how to extract a vector with a one-dimensional vector index
- Describes how to extract a subarray with a one-dimensional matrix index
- Describes how to extract all the elements in the matrix

Overview

A one-dimensional subscript contains a single index. When you use the MATLAB C Math Library to perform one-dimensional indexing, you pass `mlfIndexRef()` a pointer to one array that represents the index. The index array can contain a scalar, vector, matrix, or the return from a call to the `mlfCreateColonIndex()` function.

The size and shape of the one-dimensional index determine the size and shape of the result; the size of the result is exactly equal to the size of the one-dimensional subscript. For example, a one-dimensional row vector index produces a one-dimensional row vector result. Given the matrix `A`, the expression `A([1 5 8])` produces the row-vector `[ 1 5 8 ]`. To view the definition of matrix `A`, see “Assumptions for the Code Examples” on page 5-14.

To apply a one-dimensional subscript to an N-dimensional array, you need to know how to go from the one-dimensional index value to a location inside the array. See “Array Storage” on page 5-5 for complete details on how MATLAB counts one dimensionally through arrays of N dimensions.

Note The range for a one-dimensional index depends on the size of the array. For a given array `A`, it ranges from 1, the first element of the array, to `prod(size(A))`, the last element in an N-dimensional array. Contrast this range with the two ranges for a two-dimensional index where the row value varies from 1 to M, and the column value from 1 to N.

Selecting a Single Element

Use a scalar index to select a single element from the array. For example,

```
mlfAssign(&B, mlfIndexRef(A, "(?)", mlfScalar(5)));
```

performs the same operation as `A(5)` in MATLAB and selects the fifth element of `A`, the number `5`.

Selecting a Vector

Use a vector index to select multiple elements from an array. For example,

```
mlfIndexRef(A, "(?)",
    mlfHorzcat(mlfScalar(2), mlfScalar(5), mlfScalar(8), NULL))
```

performs the same operation as `A([2 5 8])` in MATLAB and selects the second, fifth and eighth elements of the matrix `A`:

```
2 5 8
```

Because the index is a 1-by-3 row vector, the result is also a 1-by-3 row vector.

The code

```
mlfAssign(&B, mlfIndexRef(A, "(?)",
    mlfVertcat(mlfScalar(2), mlfScalar(5), mlfScalar(8), NULL)));
```

selects the same elements of `A`, but returns the result as a column vector because the call to `mlfVertcat()` produced a column vector:

```
2
5
8
```

`A([2;5;8])` in MATLAB performs the same operation. Note the semicolons.

Specifying a Vector Index with mlfEnd()

Sometimes you don't know how large an array is in a particular dimension, but you want to perform an indexing operation that requires you to specify the last element in that dimension. In MATLAB, you can use the `end` function to refer to the last element in a given dimension.

For example, `A(6:end)` selects the elements from `A(6)` to the end of the array. The MATLAB C Math Library's `mlfEnd()` function corresponds to MATLAB's end() function. Given an array, a dimension (1 = row , 2 = column, 3 = page, and so on), and the number of indices in the subscript, mlfEnd() returns (as a 1-by-1 array) the index of the last element in the specified dimension. You can then use that scalar array to generate a vector index.

Given the row dimension for a vector or scalar array, mlfEnd() returns the number of columns. Given the column dimension for a vector or scalar array, it returns the number of rows. For a matrix, mlfEnd() treats the matrix like a vector and returns the number of elements in the matrix.

Note that the number of indices in the subscript corresponds to the number of index arguments that you pass to mlfIndexRef().

This C code selects all but the first five elements in matrix A, just as A(6:end) does in MATLAB.

```
mxArray *end_index=NULL, *B=NULL;
mlfAssign(&end_index,
          mlfColon(mlfScalar(6),
              mlfEnd(A, mlfScalar(1), mlfScalar(1)), NULL));
mlfAssign(&B, mlfIndexRef(A, "(?)", end_index));
```

The second argument, mlfScalar(1), to mlfEnd() identifies the dimension where mlfEnd() is used, here the row dimension. The third argument, mlfScalar(1), indicates the number of indices in the subscript; for one-dimensional indexing, it is always one. This code selects these elements from matrix A:

```
6 7 8 9
```

Selecting a Matrix

Use a matrix index to select a matrix. A matrix index works just like a vector index, except the result is a matrix rather than a vector. For example, let B be the index matrix:

```
1 2
3 2
```

Then,

```
mlfAssign(&X, mlfIndexRef(A, "(?)", B));
```

is

```
1 2
3 2
```

Note that the example matrix `A` was chosen so that `mlfIndexRef(A,"(?)", X)` equals `X` for all types of one-dimensional indexing. This is not generally the case. For example, if `A` were changed to `mlfAssign(&A, mlfMagic(mlfScalar(3)));`

```
8 1 6
3 5 7
4 9 2
```

and `B` remains the same, then `mlfIndexRef(A, "(?)", B)` would equal

```
8 3
4 3
```

Note In both cases, `size(A(B))` is equal to `size(B)`. This is a fundamental property of one-dimensional indexing.

Selecting the Entire Matrix As a Column Vector

Use the colon index to select all the elements in an array. The result is a column vector. For example,

```
mlfAssign(&B, mlfIndexRef(A, "(?)", mlfCreateColonIndex()));
```

is:

```
1
2
3
4
5
6
7
8
9
```

The colon index means "all." It is a context-sensitive function. It expands to a vector array containing all the indices of the dimension in which it is used (its context). In the context of an M-by-N array `A`, `A(:)` in MATLAB notation is equivalent to `A([1:M*N]')`. When you use colon, you don't have to specify M and N explicitly, which is convenient when you don't know M and N.

N-Dimensional Indexing

This section:

- Provides an overview of n-dimensional indexing
- Describes how to extract a scalar from a matrix
- Describes how to extract a vector from a matrix
- Describes how to extract a subarray from a matrix
- Describes how to extend two-dimensional indexing to N dimensions

Overview

An N-dimensional subscript contains N indices. The first index is the row index, the second is the column index, the third the page index, and so on. When you use the MATLAB C Math Library to perform N-dimensional indexing, you pass `mlfIndexRef()` N index arrays as arguments that together represent the subscript. The first index array argument stores the row index, the second the column index, the third the page index, and so on. Each index array can store a scalar, vector, matrix, or the result from a call to the function `mlfCreateColonIndex()`.

The size of the indices determines the size of the result. The size of the result is equal to the product of the sizes of the N indices. For example, assume matrix `A` is set to:

```
1 4 7
2 5 8
3 6 9
```

If you index matrix `A` with a 1-by-5 vector and a scalar, the result is a five-element vector: five elements in the first index times one element in the second index. If you index matrix `A` with a three-element row index and a two-element column index, the result has six elements arranged in three rows and two columns.

There is no functional difference between two-dimensional indexing and N-dimensional indexing (where N > 2). See "Extending Two-Dimensional Indexing to N Dimensions" on page 5-27 to learn how to work with arrays of dimension greater than two.

To use the code samples in your own code, see “Assumptions for the Code Examples” on page 5-14, which explains the conventions used in the examples.

Selecting a Single Element

Use two scalar indices to extract a single element from an array.

For example,

```
mlfAssign(&B,
          mlfIndexRef(A, "(?,?)", mlfScalar(2), mlfScalar(2)));
```

selects the element 5 from the center of matrix A (the element at row 2, column 2).

Selecting a Vector of Elements

Use one vector and one scalar index, or one matrix and one scalar index, to extract a vector of elements from an array. You can use the functions `mlfHorzcat()`, `mlfVertcat()`, or `mlfCreateColonIndex()` to make the vector or matrix index, or use an `mxArray` variable that contains a vector or matrix returned from other functions.

The indexing routines iterate over the vector index or down the columns of the matrix index, pairing each element of the vector or matrix with the scalar index. Think of this process as applying a (scalar, scalar) subscript multiple times; the result of each selection is collected into a vector.

For example,

```
mlfAssign(&B,
    mlfIndexRef(A, "(?,?)",
                mlfHorzcat(mlfScalar(1), mlfScalar(3), NULL),
                mlfScalar(2)));
```

selects the first and third element (or first and third rows) of column 2:

```
4
6
```

In MATLAB `A([1 3], 2)` performs the same operation.

If you reverse the positions of the indices (`A(2, [1 3])` in MATLAB):

```
mlfAssign(&B,
```

```
    mlfIndexRef(A, "(?,?)",
                mlfScalar(2),
                mlfHorzcat(mlfScalar(1), mlfScalar(3), NULL)));
```

you select the first and third elements (or first and third columns) of row 2:

```
2 8
```

If the vector index contains the same number multiple times, the same element is extracted multiple times. For example,

```
mlfAssign(&B,
    mlfIndexRef(A, "(?,?)",
                mlfScalar(2),
                mlfHorzcat(mlfScalar(3), mlfScalar(3), NULL)));
```

returns two copies of the element at `A(2,3)`:

```
8 8
```

Note You can pass any number of arguments to `mlfHorzcat()` or `mlfVertcat()`. You can nest calls to either function.

Specifying a Vector Index with mlfEnd( )

The `mlfEnd()` function, which corresponds to the MATLAB `end()` function, provides another way of specifying a vector index. Given an array, a dimension (`1` = row , `2` = column, `3` = page, and so on), and the number of indices in the subscript, `mlfEnd()` returns the index of the last element in the specified dimension. You then use that scalar array to generate a vector index. See "Specifying a Vector Index with mlfEnd()" on page 5-17 for a more complete description of how and why you use the `end` function in MATLAB.

Given the row dimension, `mlfEnd()` returns the number of columns. Given the column dimension, it returns the number of rows. The number of indices in the subscript corresponds to the number of index arguments you pass to `mlfIndexRef()`.

This code selects all but the first element in row 3, just as

```
A(3, 2:end)
```

does in MATLAB.

```
mxArray *two=NULL, *end_index=NULL, *B=NULL;
mlfAssign(&two, mlfScalar(2));
mlfAssign(&end_index, mlfColon(two, mlfEnd(A, two, two), NULL));
mlfAssign(&B, mlfIndexRef(A, "(?,?)",
                  mlfScalar(3), end_index));
```

The second argument to `mlfEnd()`, `two`, identifies the dimension where `mlfEnd()` is used, here the column dimension. The third argument, `two`, indicates the number of indices in the subscript; for two-dimensional indexing, it is always two. This code selects these elements from matrix `A`:

```
6  9
```

Selecting a Row or Column

Use a colon index and a scalar index to select an entire row or column. For example,

```
mlfIndexRef(A, "(?,?)", mlfScalar(1), mlfCreateColonIndex())
```

selects the first row:

```
1 4 7
```

`mlfIndexRef(A, "(?,?)", mlfCreateColonIndex(), mlfScalar(2))` selects the second column:

```
4
5
6
```

Selecting a Matrix

Use two vector indices, or a vector and a matrix index, to extract a matrix. You can use the function `mlfHorzCat()`, `mlfVertcat()`, or `mlfCreateColonIndex()` to make each vector or matrix index, or use `mxArray` variables that contain vectors or matrices returned from other functions.

The indexing code iterates over both vector indices in a pattern similar to a doubly nested `for`-loop:

```
for each element I in the row index
    for each element J in the column index
```

```
        select the matrix element A(I,J)
```

For each of the indicated rows, this operation (`A([1 2], [1 3 2])` in MATLAB) selects the column elements at the specified column positions. For example,

```
mlfAssign(&B,
    mlfIndexRef(A, "(?,?)",
                mlfHorzcat(mlfScalar(1), mlfScalar(2), NULL),
                mlfHorzcat(mlfScalar(1), mlfScalar(3),
                           mlfScalar(2), NULL)));
```

selects the first, third, and second (in that order) elements from rows 1 and 2, yielding:

```
1 7 4
2 8 5
```

Notice that the result has two rows and three columns. The size of the result matrix always matches the size of the index vectors: the row index had two elements; the column index had three elements. The result is 2-by-3.

The indexing routines treat a matrix index as one long vector, moving down the columns of the matrix. The loop for a subscript composed of a matrix in the row position and a vector in the column position works like this:

```
for each column I in the row index matrix B
    for each row J in the Ith column of B
        for each element K in the column index vector
            select the matrix element A(B(I,J), K)
```

For example, let the matrix `B` equal:

```
1 1
2 3
```

Then the expression

```
mlfIndexRef(A, "(?,?)", B,
            mlfHorzcat(mlfScalar(1), mlfScalar(2), NULL))
```

performs the same operation as `A(B,[1 2])` in MATLAB and selects the first, second, first, and third elements of columns 1 and 2:

```
1 4
2 5
```

```
1 4
3 6
```

Selecting Entire Rows or Columns

Use a colon index and a vector or matrix index to select multiple rows or columns from a matrix. For example,

```
mlfIndexRef(A, "(?,?)",
            mlfHorzcat(mlfScalar(2), mlfScalar(3), NULL),
            mlfCreateColonIndex())
```

performs the same operation as `A([2 3],:)` in MATLAB and selects all the elements in rows two and three:

```
2 5 8
3 6 9
```

You can use the colon index in the row position as well. For example, the expression

```
mlfAssign(&B,
          mlfIndexRef(A, "(?,?)",
             mlfCreateColonIndex(),
             mlfHorzcat(mlfScalar(3), mlfScalar(1), NULL)));
```

performs the same operation as `A(:,[3 1])` in MATLAB and selects all the elements in columns 3 and 1, in that order:

```
7 1
8 2
9 3
```

Subscripts of this form make duplicating the rows or columns of a matrix easy.

Selecting an Entire Matrix

Using the colon index as both the row and column index selects the entire matrix. Although this usage is valid, referring to the matrix itself without subscripting is much easier.

Extending Two-Dimensional Indexing to N Dimensions

Two-dimensional indexing extends very naturally to N dimensions; simply use more index arguments. Let A be a 3-by-3-by-2 three-dimensional array (two 3-by-3 pages):

Page 1:

```
1 4 7
2 5 8
3 6 9
```

Page 2:

```
10 13 16
11 14 17
12 15 18
```

Then the MATLAB expression `A(:,:,2)` selects all of page 2, `A(1,:,:)` selects all the columns in row 1 on all the pages, `A(2,2,2)` selects the element at the middle of page 2 (the number `14`), and so on.

It is very simple to convert these MATLAB indexing expressions into MATLAB C Math Library indexing expressions:

`A(:,:,2)` becomes

```
mlfIndexRef(A, "(?,?,?)", mlfCreateColonIndex(),
            mlfCreateColonIndex(), mlfScalar(2))
```

The result of this operation is the 3-by-3 array on page 2 of A:

```
10 13 16
11 14 17
12 15 18
```

`A(1,:,:)` becomes

```
mlfIndexRef(A, "(?,?,?)", mlfScalar(1), mlfCreateColonIndex(),
            mlfCreateColonIndex())
```

The result of this operation is a three-dimensional array 1-by-3-by-2 in which each “page” consists of the first row of the corresponding page of A.

Page 1:

```
1 4 7
```

Page 2:

```
10 13 16
```

Finally, `A(2,2,2)` becomes:

```
mlfIndexRef(A, "(?,?,?)", mlfScalar(2), mlfScalar(2),
            mlfScalar(2))
```

The result of this operation is the 1-by-1 array `14`.

If the array `A` had more than three dimensions, the index strings would have more than three `?`'s in them, and they would be followed by more than three index values. All of the other types of indexing discussed in this chapter (selecting entire rows and columns, etc.) work equally well on N-dimensional arrays.

Logical Indexing

This section:

- Provides an overview of logical indexing
- Describes how to specify a logical index as a one-dimensional subscript
- Describes how to specify two logical vectors as indices in a two-dimensional subscript
- Describes how to specify a colon index and a logical vector as a two-dimensional subscript
- Describes how to specify a logical index to select elements from a row or column

Overview

Logical indexing is a special case of *n*-dimensional indexing. A logical index is a vector or a matrix that consists entirely of ones and zeros. Applying a logical subscript to a matrix selects the elements of the matrix that correspond to the nonzero elements in the subscript.

Logical indices are generated by the relational operator functions (`mlfLt()`, `mlfGt()`, `mlfLe()`, `mlfGe()`, `mlfEq()`, `mlfNeq()`) and by the function `mlfLogical()`. Because these functions attach a logical flag to a logical matrix, you cannot create a logical index simply by assigning ones and zeros to a vector or matrix.

You can form an *n*-dimensional logical subscript by combining a logical index with scalar, vector, matrix, or colon indices.

The examples work with matrix `A` and the logical array `B`.

```
A
1 4 7
2 5 8
3 6 9

B
1 0 1
0 1 0
1 0 1
```

"Assumptions for the Code Examples" on page 5-14 explains the conventions used in the examples.

Using a Logical Matrix as a One-Dimensional Index

When you use a logical matrix as an index, the result is a column vector. For example, if the logical index matrix B equals

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

Then

```
mlfAssign(&X, mlfIndexRef(A, "(?)", B));
```

equals

```
1
3
5
7
9
```

Notice that B has ones at the corners and in the center, and that the result is a column vector of the corner and center elements of A.

Note that you can create B by calling mlfAssign(&B, mlfLogical(matrix)) where matrix stores a matrix of 1's and 0's.

If the logical index is not the same size as the subscripted array, the logical index is treated like a vector. For example, if B = logical([1 0; 0 1]), then

```
mlfAssign(&X, mlfIndexRef(A, "(?)", B));
```

equals

```
1
4
```

since B has a zero at positions 2 and 3 and a 1 at positions 1 and 4. Logical indices behave just like regular indices in this regard.

Using Two Logical Vectors as Indices

Two vectors can be logical indices into an M-by-N matrix A. The size of a logical vector index often matches the size of the dimension it indexes, although this is not a requirement.

For example, let B = logical([1 0 1]) and C = logical([0 1 0]), two vectors that do match the sizes of the dimensions where they are used. Then,

```
mlfAssign(&X, mlfIndexRef(A, "(?,?)", B, C));
```

equals

```
4
6
```

B, the row index vector, has nonzero entries in the first and third elements. This selects the first and third rows. C, the column index vector, has only one nonzero entry, in the second element. This selects the second column. The result is the intersection of the two sets selected by B and C, that is, all the elements in the second columns of rows 1 and 3.

Or, let B = logical([1 0]) and C = logical([0 1]), two vectors that do not match the sizes of the dimensions where they are used. Then

```
mlfAssign(&X, mlfIndexRef(A, "(?,?)", B, C));
```

equals

```
4
```

This is tricky. B, the row index, selects row 1 but does not select row 2. C, the column index, does not select column 1 but does select column 2. There is only one element in array A in both row 1 and column 2, the element 4.

Using One Colon Index and One Logical Vector as Indices

This type of indexing is very similar to the two-vector case. Here, however, the colon index selects all of the elements in a row or column, acting like a vector of ones the same size as the dimension to which it is applied. The logical index works just like a nonlogical index in terms of size.

For example, let the index vector B = logical([1 0 1]). Then

```
mlfIndexRef(A, "(?,?)", mlfCreateColonIndex(), B)
```

equals

```
1 7
2 8
3 9
```

The colon index selects all rows, and B selects the first and third columns in each row. The result is the intersection of these two sets, that is, the first and third columns of the matrix.

For comparison,

```
mlfAssign(&X, mlfIndexRef(A, "(?,?)", B, mlfCreateColonIndex()));
```

equals

```
1 4 7
3 6 9
```

B selects the first and third rows, and the colon index selects all the columns in each row. The result is the intersection of the sets selected by each index, that is, the first and third rows of the matrix.

Using a Scalar and a Logical Vector

Let matrix X be a 4-by-4 magic square.

```
X = magic(4);

16   2   3  13
 5  11  10   8
 9   7   6  12
 4  14  15   1
```

Let B be a logical matrix that indicates which elements in row two of matrix X are greater than 9. B is the result of the greater than (>) operation

```
mlfAssign(&target_row,
          mlfIndexRef(X, "(?,?)", mlfScalar(2),
                      mlfCreateColonIndex()));
mlfAssign(&B, mlfGt(target_row, mlfScalar(9)));
```

and contains the vector

```
0 1 1 0
```

In MATLAB, `B = (X(2,:) > 9)` performs the same operation.

Use `B` as a logical index that selects those elements from matrix `X`.

```
mlfAssign(&C, mlfIndexRef(X, "(?,?)", mlfScalar(2), B));
```

selects these elements:

```
11 10
```

Extending Logical Indexing to N Dimensions

Logical indexing works on *n*-dimensional arrays just as you'd expect. The logical filtering happens the same way, and the subscript size governs the result size in the same manner. For details on the syntax, see "Extending Two-Dimensional Indexing to N Dimensions" on page 5-27.

Assigning Values to Array Elements

This section:

- Provides an overview on index assignment
- Describes how to assign a single element to an array
- Describes how to assign multiple elements to an array
- Describes how to assign to a subarray
- Describes how to assign values to all the elements in an array

The section includes information about extending two-dimensional indexing to *n*-dimensions.

Overview

Use the function `mlfIndexAssign()` to make assignments that involve indexing. The arguments to `mlfIndexAssign()` consist of a destination array, an index string, the index arrays themselves, and the source array. The subscript specifies the elements that are to be modified in the destination array. The source array specifies the new values for those elements.

You can use five different kinds of indices:

- Scalar
- Vector
- Matrix
- Colon
- Logical

Note The size of the destination `mxArray` (after the subscript has been applied) and the size of the source `mxArray` must be the same.

The examples in this section do not present all possible combinations of these index types. “Assumptions for the Code Examples” on page 5-14 explains the conventions used in the examples.

The examples work with matrix `A`.

```
A =

   1 4 7
   2 5 8
   3 6 9
```

There is no functional difference between two-dimensional indexed assignment and *n*-dimensional indexed assignment (where N > 2). Because it is easier to understand two-dimensional arrays, most of the examples in this section deal with two-dimensional arrays. See “Extending Two-Dimensional Assignment to N Dimensions” on page 5-37 to learn how to work with n-dimensional arrays.

Assigning to a Single Element

Use one or two scalar indices to assign a value to a single element in a matrix. For example,

```
mlfIndexAssign(&A, "(?,?)", mlfScalar(2), mlfScalar(1),
               mlfScalar(17));
```

changes the element at row 2 and column 1 to the integer 17. Here, both the source and destination (after the subscript has been applied) are scalars, and thus the same size.

Assigning to Multiple Elements

Use a vector index to modify multiple elements in a matrix.

A colon index frequently appears in the subscript of the destination because it allows you to modify an entire row or column. For example, this code

```
mlfIndexAssign(&A, "(?,?)", mlfScalar(2), mlfCreateColonIndex(),
               mlfColon(mlfScalar(1),mlfScalar(3),NULL));
```

replaces the second row of an M-by-3 matrix with the vector `1 2 3`. If we use the example matrix `A`, `A` is modified to contain:

```
1 4 7
1 2 3
3 6 9
```

You can also use a logical index to select multiple elements. For example, the assignment statement

```
mlfIndexAssign(&A, "(?)", mlfGt(A,mlfScalar(5)),
```

```
                mlfHorzcat(mlfScalar(17), mlfScalar(17),
                           mlfScalar(17), mlfScalar(17),NULL));
```

changes all the elements in A that are greater than 5 to 17:

```
1     4    17
2     5    17
3    17    17
```

Assigning to a Subarray

Use two vector indices to generate a matrix destination. For example, let the vector index B = [1 2], and the vector index C = [2 3]. Then,

```
mlfAssign(&source, mlfVertcat(mlfHorzcat(mlfScalar(1),
                                         mlfScalar(4),
                                         NULL),
                              mlfHorzcat(mlfScalar(3),
                                         mlfScalar(2),
                                         NULL),
                              NULL));
mlfIndexAssign(&A, "(?,?)", B, C, source);
```

copies a 2-by-2 matrix into the second and third columns of rows 1 and 2: the upper right corner of A. The example matrix A becomes:

```
1 1 4
2 3 2
3 6 9
```

You can also use a logical matrix as an index. For example, let B be the logical matrix:

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

Then,

```
mlfIndexAssign(&A, "(?)", B, source);
```

changes A to:

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

```
3 6 9
```

Assigning to All Elements

You can use the colon index to replace all elements in a matrix with alternate values. The colon index, however, is infrequently used in this context because you can accomplish approximately the same result by using assignment without any indexing. For example, although you can write

```
mlfIndexAssign(&A, "(?)", mlfCreateColonIndex(),
               mlfRand(mlfScalar(3),NULL));
```

writing

```
mlfAssign(&A, mlfRand(mlfScalar(3), NULL));
```

is simpler.

The first statement reuses the storage already allocated for A. The first statement will be slightly slower, because the elements from the source must be copied into the destination.

Note `mlfRand(mlfScalar(3), NULL)` is equivalent to `mlfRand(mlfScalar(3), mlfScalar(3), NULL)`.

Extending Two-Dimensional Assignment to N Dimensions

Two-dimensional assignment extends naturally to N-dimensions; simply use more index arguments. Let A be a 3-by-3-by-2 three-dimensional array (two 3-by-3 pages):

Page 1:

```
1 4 7
2 5 8
3 6 9
```

Page 2:

```
10 13 16
11 14 17
```

```
12 15 18
```

Then the MATLAB expression `A(:,:,2) = eye(3)` changes page 2 to the 3-by-3 identity matrix; `A(1,:,:) = ones(1,3,2)` changes row 1 on both pages to be all ones; `A(2,2,2) = 42` changes the element at the middle of page 2 (the number 14) to the number 42, and so on.

It is very simple to convert these MATLAB indexed assignment expressions into MATLAB C Math Library indexed assignment expressions.

`A(:,:,2) = eye(3)` becomes

```
mlfIndexAssign(&A, "(?,?,?)", mlfCreateColonIndex(),
               mlfCreateColonIndex(), mlfScalar(2),
               mlfEye(mlfScalar(3), NULL));
```

As a result of this operation the 3-by-3 array on page 2 of `A` becomes:

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

`A(1,:,:) = ones(1, 3, 2)` becomes

```
mlfIndexAssign(&A, "(?,?,?)", mlfScalar(1),
               mlfCreateColonIndex(),
               mlfCreateColonIndex(),
               mlfOnes(mlfScalar(1), mlfScalar(3),
                       mlfScalar(2), NULL));
```

As a result of this operation row 1 on both pages of `A` becomes all ones.

Page 1:

```
1 1 1
2 5 8
3 6 9
```

Page 2:

```
1   1  1
11 14 17
12 15 18
```

Finally, `A(2,2,2)= 42` becomes:

```
mlfIndexAssign(A, "(?,?,?)", mlfScalar(2), mlfScalar(2),
               mlfScalar(2), mlfScalar(42));
```

As a result of this operation the element at `(2,2,2)` changes to the number `42`.

Page 2:

```
10 13 16
11 42 17
12 15 18
```

If the array `A` had more than three dimensions, the index strings would have more than three `?`'s in them, and they would be followed by more than three index values. All of the other types of indexing discussed in this chapter (assigning to entire rows and columns, etc.) work equally well on N-dimensional arrays.

Deleting Array Elements

Use the function `mlfIndexDelete()` to delete elements from an array. This function is equivalent to the MATLAB statement, `A(B) = []`. Instead of specifying a subscript for the elements you want to replace with other values, specify a subscript for the elements you want removed from the matrix. The MATLAB C Math Library removes those elements and shrinks the array.

For example, to delete elements from example matrix `A`, you simply pass the target array, the index string, and the value of the indices that identify the elements to be removed.

When you delete a single element from a matrix, the matrix is converted into a row vector that contains one fewer element than the original matrix. For example, when element (`8`) is deleted from matrix `A`

```
mlfIndexDelete(&A, "(?)", mlfScalar(8));
```

matrix `A` becomes this row vector with element `8` missing:

```
1 2 3 4 5 6 7 9
```

Deleting Multiple Elements

You can delete more than one element from a matrix, shrinking the matrix by that number of elements. To retain the rectangularity of the matrix, however, you must delete one or more entire rows or columns. For example,

```
mlfIndexDelete(&A, "(?,?)", mlfScalar(2),
               mlfCreateColonIndex());
```

produces this rectangular result:

```
1 4 7
3 6 9
```

Note An N-dimensional subscript used in a deletion operation on the left side of the assignment statement can contain only one scalar, vector, or matrix index. The other indices must be colon indices. For example, if an array is three-dimensional and you delete row 2, you must delete row 2 from all pages.

Similar to reference and assignment, two-dimensional deletion extends to N-dimensions. If A has more than two dimensions, simply specify more than two dimensions in the index string and pass more than two index values.

Cell Array Indexing

This section provides an overview of structure indexing. This section covers the following topics:

- Referencing a cell in a cell array
- Referencing a subset of a cell array
- Referencing the contents of a cell
- Referencing a subset of the contents of a cell
- Indexing nested cell arrays
- Assigning values to a cell array
- Deleting elements from a cell array

Overview

A cell array is a regularly shaped N-dimensional array of *cells*. Each cell is capable of containing any type of MATLAB data, including another cell arrays. When using cell arrays, you must be careful to distinguish between the data values stored in the cells and the cells themselves, which are data values in their own right.

MATLAB supports two types of indexing on cell arrays. The first, standard indexing, uses parentheses `()` and allows you to manipulate the cells in a cell array. The second, cell array indexing, uses braces `{}` to manipulate the data values stored in the cells.

The following sections use this sample cell array, `N`, as an example. The cell array `N` contains a 2-by-2 double array, a string array, a complex number, and a scalar array. (For information about creating cell arrays in a C program, see“Cell Arrays” on page 3-28.)

`{1,1}`
`1   2`
`3   4`
	`{1,2}`
	`'eric'`

`{2,1}` `2-4i`	`{2,2}` `7`

Tips for Working with Cell Arrays

- Cell arrays must be regularly shaped. All rows must have the same number of columns, and all columns the same number of rows. This requirement extends into dimensions higher than two, as well. For example, all pages must be the same size in a three-dimensional cell array.
- You can't do arithmetic on a cell. You cannot, for example, write `N(2,2)+1`, which attempts to add one to a cell. However, `N{2,2}+1` works perfectly well, since the cell array indexing returns the contents of cell `(2,2)` rather than the cell itself.
- Cell array indexing follows the same rules as standard indexing. You can use the colon index to refer to multiple rows or columns; you can use vector and matrix indices to extract sub-cell arrays from a cell array, etc.

This documentation focuses on two-dimensional cell arrays. If `N` were a cell array of higher dimension, the examples would still work on `N` if you added the appropriate number of dimensions to the indexing expressions.

Referencing a Cell in a Cell Array

To obtain a cell from a cell array, use parentheses in the indexing string to indicate that you are referencing the *cell itself*, not its contents.

```
mlfAssign(&c,
          mlfIndexRef(N, "(?,?)", mlfScalar(1), mlfScalar(2)));
```

`c` is a 1-by-1 cell array containing the string array `'Eric'`.

`c = N(1,2)` performs the same operation in MATLAB.

Referencing a Subset of a Cell Array

To obtain a subset of the cells in a cell array, use the colon index or a vector or matrix index to access a group of cells. For example, to extract the second row of the cell array N, write this code:

```
mlfAssign(&B, mlfIndexRef(N, "(?,?)",
                          mlfScalar(2),
                          mlfCreateColonIndex()));
```

The result, B, is a 1-by-2 cell array containing the complex number 2-4i and the integer 7.

B = N(2,:) performs the same operation in MATLAB.

Cell arrays support vector-based (one-dimensional) indexing as well. To extract the first and last elements of N, first make a vector v that contains the integers 1 and 4 (use mlfHorzcat() to construct v). Then call mlfIndexRef() like this:

```
mlfAssign(&B, mlfIndexRef(N, "(?)", v));
```

The result, B, is a 1-by-2 cell array that contains a 2-by-2 matrix (element (1,1) of N) and the scalar 7 (element (2,2) of N).

B = N([1 4]) performs the same operation in MATLAB.

Referencing the Contents of a Cell

To obtain the contents of a single cell, use braces in the indexing string to indicate that you are referencing the *cell contents*, not the cell itself.

```
mlfAssign(&c,
          mlfIndexRef(N, "{?,?}", mlfScalar(1), mlfScalar(2)));
```

c is the string array 'Eric'.

c = N{1,2} performs the same operation in MATLAB.

Referencing a Subset of the Contents of a Cell

To obtain a subset of a cell's contents, concatenate indexing expressions. For example, to obtain element (2,2) from the array in cell N{1,1}, use an indexing string "{?,?}(?,?)" that concatenates an index that references the entire *contents* of a cell with an index that references a portion of those contents.

```
mlfAssign(&d,
          mlfIndexRef(N, "{?,?}(?,?)",
                      mlfScalar(1), mlfScalar(1),
                      mlfScalar(2), mlfScalar(2)));
```

d is 5.

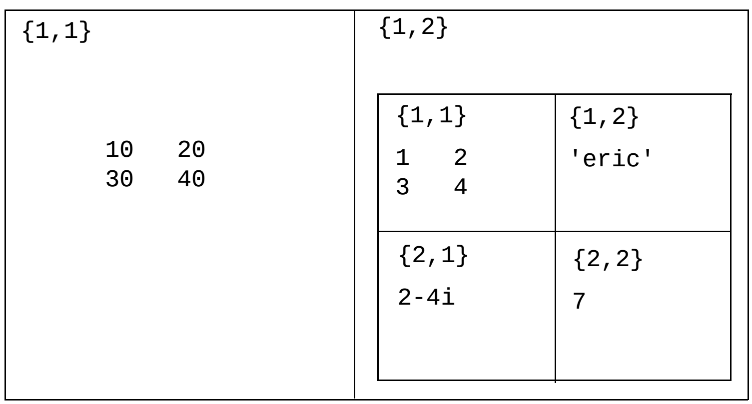
d = N{1,1}(2,2) performs the same operation in MATLAB.

Note that d is a scalar array, not a cell array.

Indexing Nested Cell Arrays

To index nested cells, concatenate subscripts in the indexing string. The first set of subscripts accesses the top layer of cells, and subsequent sets of braces access successively deeper layers of cells.

For example, array A illustrates nested cell arrays. A is a 1-by-2 cell array itself and it contains a 2-by-2 cell array nested in cell {1,2}.



Indexing the First Level

To access the 2-by-2 cell array in cell (1,2):

```
mlfIndexRef(A, "{?,?}", mlfScalar(1), mlfScalar(2))
```

In MATLAB A{1,2} performs the same operation.

Indexing the Second Level

To access the scalar value in position {2,2} of the cell array in cell {1,2}:

```
mlfIndexRef(A, "{?,?}{?,?}",
             mlfScalar(1), mlfScalar(2),
             mlfScalar(2), mlfScalar(2))
```

`A{1,2}{1,1}` in MATLAB performs the same operation.

Assigning Values to a Cell Array

You put a value into a cell array in much the same way that you read a value out of a cell array. In MATLAB, the only difference between the two operations is the position of the cell array relative to the assignment operator: left of the equal sign (=) means assignment, right of the operator means reference. No matter if you're reading or writing values, the indexing operations you use to specify which values to access remain the same.

This is true in the MATLAB C Math Library as well. The only difference between reading values from a cell array and writing values to a cell array is the function you call. `mlfIndexRef()` reads values; `mlfIndexAssign()` writes values.

For example, each of the `mlfIndexRef()` examples presented in the previous sections will also work with `mlfIndexAssign()` if you provide a source array of the correct size as the last argument to `mlfIndexAssign()`.

Like `mlfIndexRef()`, `mlfIndexAssign()` distinguishes between cell array indexing and standard indexing. For example, to assign a vector `[1 2 5 7 11]` to the contents of the cell `(1,2)` of `A`, you write `A{1,2} = [1 2 5 7 11]` in MATLAB and

```
mlfIndexAssign(&A, "{?,?}", mlfScalar(1), mlfScalar(2), vector);
```

in C with the MATLAB C Math Library. Assume the array variable `vector` is set to the vector of primes above.

You could have written the previous assignment in MATLAB as `A(1,2) = { [1 2 5 7 11] }`. The corresponding MATLAB C Math Library code is:

```
mlfIndexAssign(&A, "(?,?)", mlfScalar(1), mlfScalar(2),
               mlfCellhcat(vector,NULL));
```

Because this assignment uses parentheses instead of braces, it is an assignment between cells, which means the source array (on the right-hand side of the assignment operator) must be a cell array as well. The first line of C code creates a cell array from our initial vector of primes.

Deleting Elements from a Cell Array

Cell arrays follow the same rules as numeric arrays (and structure arrays, as you'll see in the next section) for element deletion. You can delete a single element from a cell array, or an entire dimension element, for example, a row or column of a two-dimensional cell array or a row, column, or page of a three-dimensional cell array. In MATLAB, you delete elements by assigning `[]` to them. In the MATLAB C Math Library, you use `mlfIndexDelete()`, which takes exactly the same type of arguments as `mlfIndexRef()`.

Deleting a Single Element

In order to delete a single element from an array of any type, you must use one-dimensional indexing. Deleting a single element from a two-dimensional cell array collapses it into a vector cell array. For example, deleting the `(2,1)` element of `N` (the complex number `2-4i`), produces a three-element cell array. `N(2)` refers to element `(2,1)` of `N` using one-dimensional indexing. Therefore, you'd write

```
N(2) = []
```

in MATLAB, and

```
mlfIndexDelete(&N, "(?)", mlfScalar(2));
```

in C to remove element 2,1 from `N`.

Deleting an Entire Dimension

You can delete an entire dimension by using vector subscripting to delete a row or column of cells. Use parentheses within the indexing string to indicate that you are deleting the *cells themselves*.

```
mlfIndexDelete(&N, "(?,?)",
               mlfScalar(2),mlfCreateColonIndex());
```

`N(2,:) = []` performs the same operation in MATLAB. `N{2, :} = []` is an error, because the number of items on the right- and left-hand side of the assignment operator is not the same. MATLAB does not do scalar expansion on cell arrays. In MATLAB if you want to set both cells in the second row of `N` to `[]`, write `N(2,:) = {[] []}`, thereby assigning a 1-by-2 cell array to another 1-by-2 cell array.

Note The last subscript in an array deletion must use `()`, not `{}`.

Structure Array Indexing

This section provides an overview of structure indexing, detailing how cell arrays and structure arrays interact. It includes these topics:

- Accessing a field in a structure array
- Accessing elements within a field of a structure
- Assigning a value to a field in a structure array
- Assigning a value to an element of a field
- Describes how to delete a field from a structure

Overview

A MATLAB structure is very much like a structure in C; it is a variable that contains other variables. Each of the contained variables is called a *field* of the structure, and each field has a unique name. For example, imagine you were building a database of images. You might want to create a structure with three fields: the image data, a description of the image, and the date the image was created. The following MATLAB code creates this structure:

```
images.image = image1;
images.description = 'Trees at Sunset';
images.date.year = 1998;
images.date.month = 12;
images.date.day = 17;
```

The structure `images` contains three fields: `image`, `description` and `date`. The `date` field is itself a structure, and contains three additional fields: `year`, `month` and `day`. Notice that structures can contain different types of data. `images` contains a matrix (the image), a string (the description), and another structure (the date).

Like standard arrays, structures are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value `5` is a 1-by-1 numeric array. You can build structure arrays with any valid size or shape, including multidimensional structure arrays. Assume you wanted to arrange the images from the previous example in a series of “pages,” where each page is three images wide (three columns) and four images tall (four rows). The images might be arranged this way in a photo album, or for publication in a journal.

The following code demonstrates how you use standard MATLAB indexing to create and access the elements of a 3-by-4-by-n structure array:

```
images(3,4,2).image = image24;
images(3,4,2).description = 'Greater Bird of Paradise';
images(3,4,2).date.year = 1993;
images(3,4,2).date.month = 7;
images(3,4,2).day = 15;
```

For simplicity, the examples in the book focus on two-dimensional structure arrays, but they'd work just as well with structure arrays of any dimension.

Tips for Working with Structure Arrays

- All the structures in a structure array have the same form: every structure has the same fields.
- Adding a field to one structure in a structure array adds it to all the structures in the structure array. Similarly, deleting a field from one structure in the array deletes it from all the structures in the array.
- You can access and modify data stored in the fields of a structure just as you would data stored in an ordinary variable.
- Structure fields are analogous to cell array indices, only they are names rather than numbers. Therefore, structure field access and creation use the same indexing routines as cell array (and numeric array) element access and creation do: `mlfIndexRef()`, `mlfIndexAssign()`, `mlfIndexDelete()`.
- Each field in a structure array is an array itself. For example, in the 3-by-4-by-2 example above, the array contains 24 structures. There are 24 images, 24 descriptions, etc., and you can treat each field of the structure as an array of 24 elements. If you typed `x.description`, for example, you'd get a 24-by-1 array of strings containing all the image descriptions in the structure array.

Accessing a Field

The simplest operation on a structure is retrieving data from one of the structure fields. To extract the `image` field from the second structure in a structure array, use

```
mlfAssign(&str, mlfIndexRef(images, "(?).image", mlfScalar(2)));
```

`image = imagelist(2).image` performs the same operation in MATLAB.

Accessing the Contents of a Structure Field

A structure field may contain another array. By performing additional indexing operations, you can access the data stored in that array. You must specify the field name and the type of indexing to perform on the array stored in that field:

- Use array subscripting if the field contains an array.
- Use cell array subscripting if the field contains a cell array.

For example, this code retrieves the first row of the image in the third structure:

```
mlfAssign(&n, mlfIndexRef(images,
                          "(?).image(?,?)",
                          mlfScalar(3),
                          mlfScalar(1),
                          mlfCreateColonIndex()));
```

`n = x(3).image(1,:)` performs the same operation in MATLAB.

Assigning Values to a Structure Field

To assign an initial value to a field (creating the field if it doesn't exist) or modify the value of an existing field, use `mlfIndexAssign()`. For example, to change the description field of the seventeenth image, you'd write this code:

```
mlfIndexAssign(&images,
               "(?).description",
               mlfScalar(17),
               mxCreateString("Cloned sheep embryo #1"));
```

Note that you must pass the array being modified to `mlfIndexAssign()` as an `mxArray **`, rather than the `mxArray *` that `mlfIndexRef()` requires.

`images(17).description = 'Cloned sheep embryo #1'` performs the same operation in MATLAB.

Assigning Values to Elements in a Field

By using `mlfIndexAssign()` you can also modify array data contained in a structure field. You must specify the field name and the type of indexing to perform on the contained array. For example, the following call to `mlfIndexAssign()` replaces a 3-by-3 subarray of the image data of the ninth image, with the data in the 3-by-3 array `x`. (You might do this as part of some image processing operation.)

```
mlfIndexAssign(&images,
               "(?).image(?,?)",
               mlfScalar(9),
               mlfColon(mlfScalar(1), mlfScalar(3), NULL),
               mlfColon(mlfScalar(2), mlfScalar(4), NULL),
               x);
```

`images(9).image(1:3,2:4) = x` performs the same operation in MATLAB.

Referencing a Single Structure in a Structure Array

To access a single structure within the structure array, use the standard array indexing function, `mlfIndexRef()`. For example, to reference the forty-second image structure in a structure array, use this code:

```
mlfAssign(&B, mlfIndexRef(images, "(?)", mlfScalar(42)));
```

`B = images(42)` performs the same operation in MATLAB.

Referencing into Nested Structures

Structures can contain other structures. The image structure used in these examples contains a date structure, for example. To retrieve data from nested structures, you only need a single call to `mlfIndexRef()`. Simply specify the nested structure reference operation in the second argument, as shown here:

```
mlfAssign(&y, mlfIndexRef(images,"(?).date.year",mlfScalar(2)));
```

`y = images(2).date.year` performs the same operation in MATLAB.

You can also assign to this location by using `mlfIndexAssign()` instead of `mlfIndexRef()`.

Note You can only reference or assign to single instances of nested structures. Though you might expect this MATLAB code `y = images.date.year` to set `y` to the array of years in the `date` field of the `images` structure array, this code generates an error because the result of `images.date` is a structure array rather than a single structure. It is also an error in the MATLAB C Math Library.

Accessing the Contents of Structures Within Cells

Cell arrays can contain structure arrays and vice-versa. Accessing a structure stored in a cell array is very similar to accessing a structure stored in a regular variable; you just need to extract it from the cell array first. You use `mlfIndexRef()` to combine all the operations into a single call. Assume the cell array c contains a three-element structure array of images.

You can combine cell array and standard indexing to access a single field of a single structure:

```
mlfAssign(&second_date, mlfIndexRef(c, "{?}(mlfScalar(2)).date",
                                    mlfScalar(1)));
```

`second_date = c{1}(2).date` performs the same operation in MATLAB. In this case, the result is a single date structure.

Deleting Elements from a Structure Array

There are three kinds of deletion operations you can perform on a structure array.

You can delete:

- An entire structure from the array
- A field from all the structures in the array
- Elements from an array contained by a field

Deleting a Structure from the Array

To delete an entire structure from a structure array, use `mlfIndexDelete()`. For example, if you have a three-element array of image structures, you can delete the second image structure like this:

```
mlfIndexDelete(&images, "(?)", mlfScalar(2));
```

`images(2) = []` performs the same operation in MATLAB. The result is a two-element array of image structures.

Deleting a Field from All the Structures in an Array

To delete a field from all the structures in the array, use `mlfRmfield()`. For example, you can remove the description field from an array of image structures with this code:

```
mlfAssign(&images, mlfRmfield(images,
                              mxCreateString("description")));
```

`images = rmfield(images, 'description')` performs the same operation in MATLAB.

Note that `rmfield()` does *not* allow you to remove a field of a nested structure from a structure array. For example, you cannot remove the `day` field of the nested `date` structure with this MATLAB code:

```
rmfield(images.date, 'day')
```

This is an error in MATLAB, and the corresponding call to `mlfRmfield()` is an error in the MATLAB C Math Library.

Deleting an Element from an Array Contained by a Field

To delete an element from an array contained by a field, use `mlfIndexDelete()`. For example, to remove the fifth column of the image in the third image structure, call `mlfIndexDelete()` like this:

```
mlfIndexDelete(&images, "(?).image(?,?)",
               mlfScalar(3),
               mlfCreateColonIndex(),
               mlfScalar(5));
```

`images(3).image(:,5) = []` performs the same operation in MATLAB.

Comparison of C and MATLAB Indexing Syntax

The table below summarizes the differences between the MATLAB and C indexing syntax. Although the MATLAB C Math Library provides the same functionality as the MATLAB interpreter, the syntax is very different. Refer to "Assumptions for the Code Examples" on page 5-14 to look up the conventions used for the code within the table.

Note For the examples in the table, matrix `X` is set to the 2-by-2 matrix `[ 4 5 ; 6 7 ]`, a different value from the 3-by-3 matrix `A` in the previous sections.

Example Matrix X

```
4 5
6 7
```

Table 5-2: MATLAB/C Indexing Expression Equivalence

Description	MATLAB Expression	C Expression	Result

Extract `1,1` element	`X(1,1)`	`mlfIndexRef(`
		`X,`
		`"(?,?)",`
		`mlfScalar(1),`
		`mlfScalar(1)`
		`)`
			4

Extract first element	`X(1)`	`mlfIndexRef(`
		`X,`
		`"(?)",`
		`mlfScalar(1)`
		`)`
			4

Table 5-2: MATLAB/C Indexing Expression Equivalence (Continued)

Description	MATLAB Expression	C Expression	Result

Extract third element	`X(3)`	`mlfIndexRef(`
		`X,`
		`"(?)",`
		`mlfScalar(3)`
		`)`
			5

Extract all elements into column vector	`X(:)`	`mlfIndexRef(`
		`X,`
		`"(?)",`
		`mlfCreateColonIndex()`
		`)`
			4
			6
			5
			7

Extract first row	`X(1,:)`	`mlfIndexRef(`
		`X,`
		`"(?,?)",`
		`mlfScalar(1),`
		`mlfCreateColonIndex()`
		`)`
			4 5

Extract second row	`X(2,:)`	`mlfIndexRef(`
		`X,`
		`"(?,?)",`
		`mlfScalar(2),`
		`mlfCreateColonIndex()`
		`)`
			6 7

Extract first column	`X(:,1)`	`mlfIndexRef(`
		`X,`
		`"(?,?)",`
		`mlfCreateColonIndex(),`
		`mlfScalar(1)`
		`)`
			4
			6

Table 5-2: MATLAB/C Indexing Expression Equivalence (Continued)

Description	MATLAB Expression	C Expression	Result

Extract second column	`X(:,2)`	`mlfIndexRef(`
		   `X,`
		   `"(?,?)",`
		   `mlfCreateColonIndex(),`
		   `mlfScalar(2)`
		   `)`
			5
			7

Replace first element with 9	`X(1) = 9`	`mlfIndexAssign(`
		   `&X,`
		   `"(?)",`
		   `mlfScalar(1),`
		   `mlfScalar(9)`
		   `);`
			9 5
			6 7

Replace first row with [11 12]	`X(1,:) = [ 11 12 ]`	`mlfIndexAssign(`
		   `&X,`
		   `"(?,?)",`
		   `mlfScalar(1),`
		   `mlfCreateColonIndex(),`
		   `mlfHorzcat(`
		      `mlfScalar(11),`
		      `mlfScalar(12),`
		      `NULL)`
		   `);`
			11 12
			6 7

Table 5-2: MATLAB/C Indexing Expression Equivalence (Continued)

<table>
<tr><th>Description</th><th>MATLAB Expression</th><th>C Expression</th><th>Result</th></tr>
<tr><td>Replace element 2,1 with 9</td><td><code>X(2,1) = 9</code></td><td><pre>mlfIndexAssign(
   &X,
   "(?,?)",
   mlfScalar(2),
   mlfScalar(1),
   mlfScalar(9)
   );</pre></td><td><pre>4 5
9 7</pre></td></tr>
<tr><td>Replace elements 1 and 4 with 8 (one-dimensional indexing)</td><td><code>X([1 4]) = [8 8]</code></td><td><pre>mlfIndexAssign(
   &X,
   "(?)",
   mlfHorzcat(
      mlfScalar(1),
      mlfScalar(4),
      NULL),
   mlfHorzcat(
      mlfScalar(8),
      mlfScalar(8),
      NULL)
   );</pre></td><td><pre>8 5
6 8</pre></td></tr>
</table>

6

Calling Library Routines

Introduction

The MATLAB C Math Library includes over 400 functions. Every routine in the library works the same way as its corresponding routine in MATLAB. This section describes the calling conventions that apply to the library functions, including how the C interface to the functions differs from the MATLAB interface. Once you understand the calling conventions, you can translate any call to a MATLAB function into a call to a C function.

Chapter 9, “Library Routines” lists of all the routines in the MATLAB C Math library by category. For more detailed information about the library functions, including the list of arguments and return value for each function, see the *MATLAB C Math Library Reference*. Each routine’s reference page includes a link to the documentation for the MATLAB version of the function.

This section also includes information about:

- Calling operators
- Passing a function to a MATLAB function or a function of your own creation
- Replacing an argument list with a cell array.

How to Call MATLAB Functions

Some MATLAB functions accept optional input arguments and return multiple output values. Some MATLAB functions can take a varying number of input and output values; these functions are called `varargin` and `varargout` functions.

C does not allow routine with the same name to accept different calling sequences nor does it allow a routine to return more than one value.

Thus, to translate MATLAB functions into callable C routines, the MATLAB C Math Library had to establish certain calling conventions. This chapter describes these conventions.

One Output Argument and Only Required Input Arguments

For MATLAB functions that do not have optional input arguments and that return only a single value, the translation to C syntax is direct. For example, in interpreted MATLAB, you invoke the cosine function, `cos`, like this

```
Y = cos(X);
```

where both `X` and `Y` are arrays.

Using the MATLAB C Math Library, you invoke cosine in much the same way

```
mlfAssign(&Y, mlfCos(X));
```

where both `X` and `Y` are pointers to `mxArray` structures. `Y` must be initialized to `NULL`. `mlfAssign()` assigns the return value from `mlfCos()` to `Y`.

Note The example above and all the remaining examples in this chapter use the automated memory management routine `mlfAssign()` to bind the array return value to a variable. For information about writing functions that use `mlfAssign()` and C Math Library automated memory management, see Chapter 4, “Managing Array Memory.”.

Optional Input Arguments

Some MATLAB functions take optional input arguments. `tril`, for example, which returns the lower triangular part of a matrix, takes either one input argument or two. The second input argument, `k`, if present, indicates which diagonal to use as the upper bound; `k=0` indicates the main diagonal, and is the default if no `k` is specified. In interpreted MATLAB you invoke `tril` either as

```
L = tril(X)
```

or

```
L = tril(X,k)
```

where `L`, `X`, and `k` are arrays. `k` is a 1–by–1 array.

Because C does not permit an application to have two functions with the same name, the MATLAB C Math Library version of the `tril` function always takes two arguments. The second argument is optional. The word "optional" means that the input argument is optional to the working of the function. However, some value must always appear in that argument's position in the parameter list. Therefore, if you do not want to pass the second argument, you must pass `NULL` in its place.

The two ways to call the MATLAB C Math library version of `tril` are

```
mlfAssign(&L, mlfTril(X,NULL));
```

and

```
mlfAssign(&L, mlfTril(X,k));
```

where `L`, `X`, and `k` are pointers to `mxArray` structures. `L` must be initialized to `NULL` before being passed to the `mlfAssign()` routine.

Optional Output Arguments

MATLAB functions may also have optional or multiple output arguments. For example, you invoke the `find` function, which locates nonzero entries in arrays, with one, two, or three output arguments.

```
k = find(X);
[i,j] = find(X);
[i,j,v] = find(X);
```

In interpreted MATLAB, find returns one, two, or three values. In C, a function cannot return more than one value. Therefore, the additional arrays must be passed to find in the argument list. They are passed as pointers to mxArray pointers (mxArray** variables).

Output arguments always appear before input arguments in the parameter list. In order to accommodate all the combinations of output arguments, the MATLAB C Math Library mlfFind() function takes three arguments, the first two of which are mxArray** parameters corresponding to output values.

Using the MATLAB C Math Library, you call mlfFind like this

```
mlfAssign(&k, mlfFind(NULL,NULL,X));
mlfAssign(&i, mlfFind(&j,NULL,X));
mlfAssign(&i, mlfFind(&j,&v,X));
```

where i, j, k, v, and X are mxArray* variables. i, j, k, and v are initialized to NULL.

The general rule for multiple output arguments is that the function return value, an mxArray*, corresponds to the first output argument. All additional output arguments are passed into the function as mxArray** parameters.

Optional Input and Output Arguments

MATLAB functions may have both optional input and optional output arguments. For example, the MATLAB singular value decomposition function svd, supports three syntaxes, each with optional input and output arguments.

```
s = svd(X)
[U, S, V] = svd(X)
[U, S, V] = svd(X, 0)
```

The three calls to svd differ both in the number of arguments passed to svd and in the number of values returned by svd. Notice that there is one constant among all three calls – the X input parameter is always present in the parameter list. X is therefore a *required* argument; the other four arguments (U, S, V, and 0) are *optional* arguments.

The MATLAB C Math Library function mlfSvd has an argument list that encompasses all the combinations of arguments the MATLAB svd function accepts. All the arguments to mlfSvd are pointers. The return value is a pointer as well. Input arguments and return values are always declared as mxArray*, output arguments as mxArray**.

```
mxArray *mlfSvd(mxArray **S, mxArray **V, mxArray *X,
                mxArray *Zero);
```

The return value and the parameters S and V represent the output arguments of the corresponding MATLAB function svd. The parameters X and Zero correspond to the input arguments of svd. Notice that all the output arguments are listed before any input argument appears; this is a general rule for MATLAB C Math Library functions.

mlfSvd has four arguments in its parameter list and one return value for a total of five arguments. Five is also the maximum number of arguments accepted by the MATLAB svd function. Clearly, mlfSvd can accept just as many arguments as svd. But because C does not permit arguments to be left out of a parameter list, there is still the question of how to specify the various combinations.

The svd reference page from the online *MATLAB Function Reference* indicates that there are three valid combinations of arguments for svd: one input and one output, one input and three outputs, and two inputs and three outputs. All MATLAB C Math Library functions have the same number of inputs and outputs as their MATLAB interpreted counterparts.

```
mxArray *X;
mxArray *U = NULL, *S = NULL, *V = NULL ;

mlfAssign(&S, mlfSvd(NULL, NULL, X, NULL));
mlfAssign(&U, mlfSvd(&S, &V, X, NULL));
mlfAssign(&U, mlfSvd(&S, &V, X, mlfScalar(0)));
```

In C, a function can return only one value. To overcome this limitation, the MATLAB C Math Library places all output parameters in excess of the first in the function argument list. The MATLAB svd function can have a maximum of three outputs, therefore the mlfSvd function returns one value and takes two output parameters, for a total of three outputs.

Notice that where the svd function may be called with differing numbers of arguments, the mlfSvd function is always called with the same number of arguments: four; mlfSvd always returns a single value. However, the calls to mlfSvd are not identical: each has a different number of NULLs in the argument list. Each NULL argument takes the place of an “optional” argument.

Variable Input Arguments

Some MATLAB functions accept any number of input arguments. In MATLAB these functions are called varargin functions. When the variable varargin appears as the last input argument in the definition of a MATLAB function, you can pass any number of input arguments to the function, starting at that position in the argument list.

MATLAB takes the arguments you pass and stores them in a cell array, which can hold any size or kind of data. The varargin function then treats the elements of that cell array exactly as if they were arguments passed to the function.

Whenever you see ... (an ellipsis) at the end of the input argument list in a MATLAB syntax description, the function is a varargin function. For example, the syntax for the MATLAB function cat includes the following specification in the online *MATLAB Function Reference*.

```
B = cat(dim,A1,A2,A3,A4,...)
```

cat accepts any number of arguments. The dim and A1 arguments to cat are required. You then concatenate any number of additional arrays along the dimension dim. For example, this call concatenates six arrays along the second dimension.

```
B = cat(2,A1,A2,A3,A4,A5,A6)
```

The C language supports functions that accept variable-length argument lists. MATLAB varargin functions translate easily into these functions. The variable number of arguments are always specified at the end of the argument list and are indicated in the function prototype as an ellipsis (...). For example, the prototype for the mlfCat() function in the MATLAB C Math Library is

```
mxArray *mlfCat(mxArray *dim, mxArray *A1, ...);
```

Though C uses its own mechanism, different from cell arrays, to process variable-length argument lists, the translation from a call to a MATLAB varargin function to a call to the MATLAB C Math Library function is straightforward. You invoke mlfCat() like this

```
mlfAssign(&B, mlfCat(mlfScalar(2),A1,A2,A3,A4,A5,A6,NULL));
```

where B is an mxArray * variable, initialized to NULL. The six A matrices are also mxArray * variables.

Note Always terminate the argument list to a varargin function with a NULL argument.

Pure Varargin Functions

Some MATLAB functions take a varargin argument as their only input argument, and are therefore called pure varargin functions. For example,

```
function [output_arg1] = Example_Pure_Varargin(varargin)
```

declares a pure varargin function in MATLAB.

Because the C language requires at least one explicit argument in the definition of a varargin function, this pure varargin function translates to

```
mxArray *Example_Pure_Varargin(mxArray *input_arg1, ...);
```

where input_arg1 is the first of the varargin parameters even though it is an explicit argument.

Variable Output Arguments

Some MATLAB functions return any number of outputs. In MATLAB these functions are called varargout functions. When the variable varargout appears as the last output argument in the definition of a MATLAB function, that function can return any number of outputs, starting at that position in the argument list.

When you call a varargout function in the interpreted MATLAB environment, MATLAB takes the arguments you pass and stores them in the cell array called varargout. A cell array can hold any size or kind of data. The MATLAB function accesses the varying number of arguments passed to it through the cell array.

Whenever you see ... (an ellipsis) within the output argument list of a MATLAB syntax description, the function is a varargout function. For example, this syntax in the online *MATLAB Function Reference* specifies a version of the MATLAB function size that returns a variable number of outputs depending on the number of dimensions in the array passed to it.

```
[M1,M2,M3,...,MN] = size(X)
```

If the input argument X is a two-dimensional array, size returns the length of the first dimension in the first output value and the length of the second dimension in a second output value. If the input argument is a four dimensional array, it returns four lengths.

For example, if the input array, X, has four dimensions, this code retrieves the length of each dimension.

```
[d1,d2,d3,d4] = size(X)
```

In the MATLAB C Math Library you invoke the same call to size like this

```
mlfSize(mlfVarargout(&d1,&d2,&d3,&d4,NULL),X,NULL));
```

where X, d1, d2, d3, and d4 are mxArray * variables. d1, d2, d3, and d4 are each initialized to NULL. The final input argument to mlfSize() is an optional input argument; in this version of the function, that argument is not used, and NULL is passed.

Note mlfSize() is what's called a pure varargout function. Pure varargout functions have no required or optional outputs, and no return value. The variable that would ordinarily be used to store the return value must instead be passed to the pure varargout function as the first argument of the mlfVarargout() routine.

Constructing an mlfVarargoutList

You recognize a varargout function prototype in the library by its argument of type mlfVarargoutList. The MATLAB C Math Library positions an mlfVarargoutList structure as the last output argument for functions that can return any number of output values, for example,

```
mxArray *mlfVarargout_function(mxArray **y,
                               mlfVarargoutList *varargout,
                               mxArray *a,
                               mxArray *b);
```

An mlfVarargoutList is always the last output argument passed to the function. Any required and optional arguments precede it.

The MATLAB C Math Library provides two functions that construct an mlfVarargoutList structure: mlfVarargout() and mlfIndexVarargout(). Whether you pass indexed varargout arguments to the varargout function determines which function you use:

- Use mlfVarargout() if you're *not* applying a subscript to any of your varargout output arguments.
- Use mlfIndexVarargout() if you *are* applying a subscript to at least one of your varargout output arguments.

Forming a List of Non-Indexed varargout Arguments. If you are not indexing into any of the arrays that you pass as varargout output arguments, you form an mlfVarargoutList by passing the address of each mxArray* to mlfVarargout(). Its prototype is

```
mlfVarargoutList *mlfVarargout(mxArray **pp_array, ...);
```

Follow these guidelines when you call mlfVarargout():

- Pass any number of mxArray** variables to mlfVarargout()
- Terminate your list of arguments with NULL

For example, if you want to pass three varargout output arguments to the example varargout function mlfVarargout_Function presented above, embed a call to mlfVarargout() as the second argument

```
mlfAssign(&x,
          mlfVarargout_Function(&y,
                                mlfVarargout(&z, &m, &n, NULL),
                                a,
                                b);
```

where all variables are mxArray* pointers. x is the return value; y is a required output argument. z, m, and n are varargout output mxArray* variables. a and b are input variables. Note that this function is not a pure varargout function.

In MATLAB, the function call looks like this.

```
[x, y, z, m, n] = mlfVarargout_Function (a, b);
```

Forming a List of Indexed varargout Arguments. If you are indexing into at least one of the arrays that you pass as a varargout output argument, you must form your mlfVarargoutList by passing indexed and nonindexed arguments to mlfIndexVarargout(), follow these guidelines:

- Pass an indexed array as a series of arguments, just as you do when indexing an array with mlfIndexRef():
  - The address of the pointer to the array (mxArray **)
  - The index string
  - The index subscripts
- Pass each non-indexed array as:
  - The address of the pointer to the array (mxArray **)
  - A NULL argument
- Terminate your entire list of arguments with NULL.

For example, if you want to pass three varargout output arguments, two of which are indexed, to the example varargout function mlfVarargout_Function presented above, embed a call to mlfindexVarargout() as the second argument.

In MATLAB, the function call looks like this.

```
[x, y, z(1), m, n{:}] = mlfVarargout_Function(a, b)
```

In C, the function call looks like this.

```
mxArray *x = NULL, *y = NULL, *z = NULL, *m = NULL, *n = NULL;

mlfAssign(&x, mlfVarargout_Function(&y,
                mlfIndexVarargout(&z, "(?)", mlfScalar(1),
                                        &m, NULL,
                                    &n, "{?}", mlfCreateColonIndex(),
                                        NULL),
                a, b)));
```

Pure Varargout Functions

Some MATLAB functions define a varargout argument as their *only* output argument, and are, therefore, called pure varargout functions. For example,

```
function [varargout] = Example_Pure_Varargout(a, b)
```

declares a pure `varargout` function in MATLAB.

The MATLAB C Math Library requires that you pass *all* `varargout` output arguments to `mlfVarargout()` or `mlfIndexVarargout()`. The variable that would ordinarily be used to store the return value must instead be passed to the pure varargout function as the first argument of the `mlfVarargout()` routine.

You construct an `mlfVarargoutList` by passing any number of array arguments to the function `mlfVarargout()` or any mix of indexed and non-indexed array arguments to `mlfIndexVarargout()`.

Summary of Library Calling Conventions

Though this section has focused on just a few functions, the principles presented apply to the majority of the functions in the MATLAB C Math Library. In general, a MATLAB C Math Library function call consists of a function name, a set of input arguments, and a set of output arguments. In addition to being classified as input or output, each argument is either required or optional.

The type of an argument determines where it appears in the function argument list. All output arguments appear before any input argument. Within that division, all required arguments appear before any optional arguments. The order, therefore, is: required outputs, optional outputs, `varargout` or `mlfVarargoutList` output (a `varargout` output list), required inputs, optional inputs, and variable-length inputs (`varargin` arguments).

To map a MATLAB function call to a MATLAB C Math Library function call, follow these steps:

1. Capitalize the first letter of the MATLAB function name that you want to call, and add the prefix `mlf`.

2. Examine the MATLAB syntax for the function.

   Find the MATLAB call with the largest number of arguments. Determine which input and output arguments are required and which are optional.

3. Make the first output argument the return value from the function.

4 Pass any other output arguments as the first arguments to the function. If the function accepts any number of output arguments, pass those arguments to `mlfVarargout()` or `mlfIndexVarargout()` in the last output argument position.

5 Pass a `NULL` argument wherever an optional output argument does not apply to the particular call you're making.

6 Pass the input arguments to the C function, following the output arguments. If the function accepts any number of input arguments, pass those arguments as the last input arguments.

7 Pass a `NULL` argument wherever an optional input argument does not apply to the particular call.

Passing the wrong number of arguments to a function causes compiler errors. Passing `NULL` in the place of a required argument causes runtime errors.

Note The online *MATLAB C Math Library Reference* does the mapping between MATLAB and C functions for you. Access the *Reference* from the Help Desk.

Exceptions to the Calling Conventions

The `mlfLoad()`, `mlfSave()` and `mlfFeval()` functions do not follow the standard calling conventions for the library. For information about `mlfLoad()` and `mlfSave()`, see Chapter 7, "Importing and Exporting Array Data.". For information about `mlfFeval()`, see "Passing Functions As Arguments to Library Routines" on page 6-20.

Example – Calling Library Routines (ex3.c)

This example program illustrates how to call library routines that take multiple, optional arguments. The example uses the singular value decomposition function `mlfSvd`.

You can find the code for this example in the `<matlab>/extern/examples/cmath` directory on UNIX systems or the `<matlab>\extern\examples\cmath` directory on PCs, where `<matlab>` represents the top-level directory of your installation. See "Building

Stand-Alone C Applications” in Chapter 2 for information on building the examples.

```
    /* ex3.c */

    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
1   #include "matlab.h"

2   static double data[] = { 1, 3, 5, 7, 2, 4, 6, 8 };

    main()
    {
3       mxArray *X = NULL;
        mxArray *U = NULL, *S = NULL, *V = NULL;

        mlfEnterNewContext(0, 0);
        mlfAssign(&X, mlfDoubleMatrix(4, 2, data, NULL));

4       mlfAssign(&U, mlfSvd(NULL, NULL, X, NULL));
        mlfPrintf("One input, one output:\nU = \n");
        mlfPrintMatrix(U);

5       mlfAssign(&U, mlfSvd(&S, &V, X, NULL));
        mlfPrintf("One input, three outputs:\n");
        mlfPrintf("U = \n"); mlfPrintMatrix(U);
        mlfPrintf("S = \n"); mlfPrintMatrix(S);
        mlfPrintf("V = \n"); mlfPrintMatrix(V);

6       mlfAssign(&U, mlfSvd(&S, &V, X, mlfScalar(0.0)));
        mlfPrintf("Two inputs, three outputs:\n");
        mlfPrintf("U = \n"); mlfPrintMatrix(U);
        mlfPrintf("S = \n"); mlfPrintMatrix(S);
        mlfPrintf("V = \n"); mlfPrintMatrix(V);

7       mxDestroyArray(X);
        mxDestroyArray(U);
        mxDestroyArray(S);
        mxDestroyArray(V);
        mlfRestorePreviousContext(0, 0);
        return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1. Include `"matlab.h"`. This file contains the declaration of the `mxArray` data structure and the prototypes for all the functions in the library. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2. Declare the eight-element static array that subsequently initializes the `mlfSvd` input matrix. The elements in this array appear in column-major order. The MATLAB C Math Library stores its array data in column-major order, unlike C, which stores array data in row-major order.

3. Declare and initialize the `mlfSvd` input array, `X`. Declare and initialize `mxArray*` variables, `U`, `S`, and `V`, to be used as output arguments in later calls to `mlfSvd`.

4. `mlfSvd` can be called in three different ways. Call it the first way, with one input matrix and one output matrix. Note that the optional inputs and outputs in the parameter list are set to `NULL`. Optional, in this case, does not mean that the arguments can be omitted from the parameter list; instead it means that the argument is optional to the workings of the function and that it can be set to `NULL`.

   Print the result of the call to `mlfSvd()`.

   If you want to know more about the function `mlfSvd()` or the calling conventions for the library, refer to the online *MATLAB C Math Library Reference*.

5. Call `mlfSvd` the second way, with three output arguments and one input argument. The additional output arguments, `S` and `V`, appear first in the argument list. Because the return value from `mlfSvd` corresponds to the first output argument, `U`, only two output arguments, `S` and `V`, appear in the argument list, bringing the total number of outputs to three. The next argument, `X`, is the required input argument. Only the final argument, the optional input, is passed as `NULL`.

   Print all of the output matrices.

6 Call `mlfSvd` the third way, with three output arguments and two input arguments. Print all of the output matrices.

Notice that in this call, as in the previous one, an ampersand (&) precedes the two additional output arguments. An ampersand always precedes each output argument because the address of the `mxArray*` is passed. The presence of an `&` is a reliable way to distinguish between input and output arguments. Input arguments never have an `&` in front of them.

7 Last of all, free all of the matrices that have been bound to variables.

Output

When the program is run, it produces this output.

```
One input, one output:
U =
   14.2691
    0.6268

One input, three outputs:
U =
    0.1525    0.8226   -0.3945   -0.3800
    0.3499    0.4214    0.2428    0.8007
    0.5474    0.0201    0.6979   -0.4614
    0.7448   -0.3812   -0.5462    0.0407

S =
   14.2691         0
         0    0.6268
         0         0
         0         0

V =
    0.6414   -0.7672
    0.7672    0.6414

Two inputs, three outputs:
U =
    0.1525    0.8226
    0.3499    0.4214
```

```
    0.5474    0.0201
    0.7448   -0.3812

S =
   14.2691         0
         0    0.6268

V =
    0.6414   -0.7672
    0.7672    0.6414
```

Calling Operators

Every operator in MATLAB is mapped directly to a function in the MATLAB C Math Library. Invoking MATLAB operators in C is simply a matter of determining the name of the function that corresponds to the operator and then calling the function as explained above. The section "Operators and Special Functions" on page 9-5 lists the MATLAB operators and the corresponding MATLAB C Math Library functions.

Passing Functions As Arguments to Library Routines

The MATLAB C Math Library includes *function-functions*: functions that execute a function that you provide. For example, the library function, `mlfOde23()`, is a function-function. Other function-functions include `mlfFzeros()`, `mlfFmin()`, `mlfFmins()`, `mlfFunm()`, and the other `mlfOde` functions.

In this section, you'll learn:

- How the function-functions use `mlfFeval()`
- How `mlfFeval()` works
- How to extend `mlfFeval()` by writing a "thunk function"

How Function-Functions Use mlfFeval( )

A function-function uses `mlfFeval()` to execute the function passed to it. For instance, `mlfOde23()` in "Example – Passing Functions As Arguments (ex4.c)" on page 6-22 calls `mlfFeval()` to execute the function `lorenz()`. The function-function passes the name of the function to be executed to `mlfFeval()` along with the arguments required by the function. In this example, the string array containing `"lorenz"` is passed to `mlfFeval()` along with the other arguments that were passed to `mlfOde23()`.

Because the functions passed to `mlfFeval()` take different numbers of input and output arguments, `mlfFeval()` uses a non-standard calling convention. Instead of listing each argument explicitly, `mlfFeval()` works with arrays of input and output arguments, allowing it to handle every possible combination of input and output arguments on its own.

The prototype for `mlfFeval()` is

```
mxArray *mlfFeval(mlfVarargoutList *varargout,
                  void (*mxfn)(int nlhs, mxArray **plhs,
                               int nrhs, mxArray **prhs),
                  ...);
```

Each function-function, therefore, constructs an array of input arguments (`prhs`) and an array of output arguments (`plhs`), and then passes those two arrays, along with the number of arguments in each array (`nrhs` and `nlhs`) and the name of the function (`name`), to `mlfFeval()`, which executes the function.

How mlfFeval() Works

`mlfFeval()` uses a built-in table to find out how to execute a particular function. The built-in table provides `mlfFeval()` with two pieces of information: a pointer that points to the function to be executed and a pointer to what's called a "thunk function."

As shipped, `mlfFeval()`'s built-in table contains each function in the MATLAB C Math Library. If you want `mlfFeval()` to know how to execute a function that you've written, you must extend the built-in table by creating a local function table that identifies your function for `mlfFeval()`.

It's the thunk function, however, that actually knows how to execute your function. In "Example – Passing Functions As Arguments (ex4.c)" on page 6-22, the thunk function, `_lorenz_thunk_fcn_`, executes `lorenz()`. A thunk function's actions are solely determined by the number of input and output arguments to the function it is calling. Therefore, any functions that have the same number of input and output arguments can share the same thunk function. For example, if you wrote three functions that each take two inputs and produce three outputs, you only need to write one thunk function to handle all three.

`mlfFeval()` calls the thunk function through the pointer it retrieves from the built-in table, passing it a pointer to the function to be executed, the number of input and output arguments, and the input and output argument arrays. Thunk functions also use the `mlfFeval()` calling convention.

The thunk function then translates from the calling convention used by `mlfFeval()` (arrays of arguments) to the standard C Math Library calling convention (an explicit list of arguments), executes the function, and returns the results to `mlfFeval()`.

Extending the mlfFeval() Table

To extend the built-in `mlfFeval()` table, you must:

1. Write the function that you want a function-function to execute.
2. Write a thunk function that knows how to call your function.
3. Declare a local function table and add the name of your function, a pointer to your function, and a pointer to your thunk function to that table.

4 Register the local table with `mlfFeval()`.

Note that your program can't contain more than 64 local function tables, but each table can contain an unlimited number of functions.

Writing a Thunk Function

A thunk function must:

1 Ensure that the number of arguments in the input and output arrays matches the number of arguments required by the function to be executed. Remember that functions in the MATLAB C Math Library can have optional arguments.

2 Extract the input arguments from the input argument array.

3 Call the function that was passed to it.

4 Place the results from the function call into the output array.

Note You don't need to write a thunk function if you want a function-function to execute a MATLAB C Math Library function. A thunk function and an entry in the built-in table already exist.

Example – Passing Functions As Arguments (ex4.c)

To illustrate function-functions, this example program uses the ordinary differential equation (ODE) solver `mlfOde23()` to compute the trajectory of the Lorenz equation. Given a function, `F`, and a set of initial conditions expressing an ODE, `mlfOde23()` integrates the system of differential equations, $y' = F(t,y)$, over a given time interval. `mlfOde23()` integrates a system of ordinary differential equations using second and third order Runge-Kutta formulas. In this example, the name of the function being integrated is `lorenz`.

For convenience, this example has been divided into three sections; in a working program, all of the sections would be placed in a single file. The first code section specifies header files, declares global variables including the local function table, and defines the `lorenz` function.

```
    /* ex4.c */

    #include <stdlib.h>
1   #include "matlab.h"

2   double SIGMA, RHO, BETA;

3   static mlfFuncTabEnt MFuncTab[] =
    {
        {"lorenz", (mlfFuncp)lorenz, _lorenz_thunk_fcn_ },
        { 0, 0, 0}
    };

4   mxArray *lorenz(mxArray *tm, mxArray *ym)
    {
        mxArray *ypm = NULL;
        double *y, *yp;

        mlfEnterNewContext(0, 2, tm, ym);

5       mlfAssign(&ypm, mlfDoubleMatrix(3, 1, NULL, NULL));
        y = mxGetPr(ym);
        yp = mxGetPr(ypm);

6       yp[0] = –BETA*y[0] + y[1]*y[2];
        yp[1] = –SIGMA*y[1] + SIGMA*y[2];
        yp[2] = –y[0]*y[1] + RHO*y[1] – y[2];

        mlfRestorePreviousContext(0, 2, tm, ym);
        return mlfReturnValue(ypm);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1. Include `"matlab.h"`. This file contains the declaration of the `mxArray` data structure and the prototypes for all the functions in the library. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

2 Declare SIGMA, RHO, and BETA, which are the parameters for the Lorenz equations. The main program sets their values, and the lorenz function uses them.

3 Declare a static global variable, MFuncTab[], of type mlfFuncTabEnt. This variable stores a function table entry that identifies the function that mlfOde23() calls. A table entry contains three parts: a string that names the function ("lorenz"), a pointer to the function itself ((mlfFuncp)lorenz), and a pointer to the thunk function that actually calls lorenz, (_lorenz_thunk_fcn_). The table is terminated with a {0, 0, 0} entry.

   Before you call mlfOde23() in the main program, pass MFuncTab to the function mlfFevalTableSetup(), which adds your entry to the built-in function table maintained by the MATLAB C Math Library. Note that a table can contain more than one entry.

4 Define the Lorenz equations. The input is a 1-by-1 array, tm, containing the value of t, and a 3-by-1 array, ym, containing the values of y. The result is a new 3-by-1 array, ypm, containing the values of the three derivatives of the equation at time = t.

5 Create a 3-by-1 array for the return value from the lorenz function.

6 Calculate the values of the Lorenz equations at the current time step. (lorenz doesn't use the input time step, tm, which is provided by mlfOde23.) Store the values directly in the real part of the array that lorenz returns. yp points to the real part of ypm, the return value.

The next section of this example defines the thunk function that actually calls lorenz. You must write a thunk function whenever you want to pass a function that you've defined to one of MATLAB's function-functions.

```
1  static int _lorenz_thunk_fcn_(mlfFuncp pFunc, int nlhs,
                                 mxArray **lhs, int nrhs,
                                 mxArray **rhs )
   {
2      typedef mxArray *(*PFCN_1_2)( mxArray * , mxArray *);
       mxArray *Out;

3      if (nlhs > 1 || nrhs > 2)
       {
           return(0);
       }

4      Out = (*((PFCN_1_2)pFunc))(
                                  (nrhs > 0 ? rhs[0] : NULL),
                                  (nrhs > 1 ? rhs[1] : NULL)
                                  );
5      if (nlhs > 0)
           lhs[0] = Out;

6      return(1);
   }
```

The numbered items in the list below correspond to the numbered sections of code example:

1. Define the thunk function that calls the `lorenz` function. A thunk function acts as a translator between your function's interface and the interface needed by the MATLAB C Math Library.

   The thunk function takes five arguments that describe any function with two inputs and one output (in this example the function is always `lorenz()`): an `mlfFuncp` pointer that points to `lorenz()`, an integer (`nlhs`) that indicates the number of output arguments required by `lorenz()`, an array of `mxArray`'s (`lhs`) that stores the results from `lorenz()`, an integer (`nrhs`) that indicates the number of input arguments required by `lorenz()`, and an array of `mxArray`'s (`rhs`) that stores the input values. The `lhs` (left-hand side) and `rhs` (right-hand side) notation refers to the output arguments that appear on the *left-hand* side of a MATLAB function call and the input arguments that appear on the *right-hand* side.

2 Define the type for the lorenz function pointer. The pointer to lorenz comes into the thunk function with the type mlfFuncp, a generalized type that applies to any function.

mlfFuncp is defined as follows:

```
typedef void (*mlfFuncp)(void)
```

The function pointer type that you define here must precisely specify the return type and argument types required by lorenz. The program casts pFunc to the type you specify here.

The name PFCN_1_2 makes it easy to identify that the function has 1 output argument (the return) and 2 input arguments. Use a similar naming scheme when you write other thunk functions that require different numbers of arguments. For example, use PFCN_2_3 to identify a function that has two output arguments and three input arguments.

3 Verify that the expected number of input and output arguments have been passed. lorenz expects two input arguments and one output argument. (The return value counts as one output argument.) Exit the thunk function if too many input or output arguments have been provided. Note that the thunk function relies on the called function to do more precise checking of arguments.

4 Call lorenz, casting pFunc, which points to the lorenz function, to the type PFCN_1_2. Verify that the two expected arguments are provided. If at least one argument is passed, pass the first element from the array of input values (rhs[0]) as the first input argument; otherwise pass NULL. If at least two arguments are provided, pass the second element from the array of input values (rhs[1]) as the second argument; otherwise pass NULL as the second argument. The return from lorenz is stored temporarily in the local variable Out.

This general calling sequence handles optional arguments. It is technically unnecessary in this example because lorenz has no optional arguments. However, it is an essential part of a general purpose thunk function.

Note that you must cast the pointer to lorenz to the function pointer type that you defined within the thunk function.

5 Assign the value returned by `lorenz` to the appropriate position in the array of output values. The return value is always stored at the first position, `lhs[0]`. If there were additional output arguments, values would be returned in `lhs[1]`, `lhs[2]`, and so on.

6 Return success.

The next section of this example contains the main program. Keep in mind that in a working program, all parts appear in the same file.

```
int main( )
{
①       mxArray *tm = NULL, *ym = NULL, *tsm = NULL, *ysm = NULL;
        double tspan[] = { 0.0, 10.0 };
        double y0[] = { 10.0, 10.0, 10.0 };
        double *t, *y1, *y2, *y3;
        int k, n;

        mlfEnterNewContext(0, 0);
②       mlfFevalTableSetup ( MFuncTab );

③       SIGMA = 10.0;
        RHO = 28.0;
        BETA = 8.0/3.0;

④       mlfAssign(&tsm, mlfDoubleMatrix(2, 1, tspan, NULL));
        mlfAssign(&ysm, mlfDoubleMatrix(1, 3, y0, NULL));

⑤       mlfAssign(&tm, mlfOde23(&ym, mlfVarargout(NULL),
                  mxCreateString("lorenz"), tsm, ysm, NULL, NULL));

⑥       n = mxGetM(tm);
        t = mxGetPr(tm);
        y1 = mxGetPr(ym);
        y2 = y1 + n;
        y3 = y2 + n;

⑦       mlfPrintf("      t        y1         y2       y3\n");
        for (k = 0; k < n; k++) {
            mlfPrintf("%9.3f %9.3f %9.3f %9.3f\n",
                      t[k], y1[k], y2[k], y3[k]);
        }

⑧       mxDestroyArray(tsm);
        mxDestroyArray(ysm);
        mxDestroyArray(tm);
        mxDestroyArray(ym);
        mlfRestorePreviousContext(0,0);
        return(EXIT_SUCCESS);
}
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Declare and initialize variables. `tspan` stores the start and end times. `y0` is the initial value for the `lorenz` iteration and contains the vector `10.0`, `10.0`, `10.0`.

2 Add your function table entry to the MATLAB C Math Library built-in `feval` function table by calling `mlfFevalTableSetup()`. The argument, `MFuncTab`, associates the string `"lorenz"` with a pointer to the `lorenz` function and a pointer to the `lorenz` thunk function. When `mlfOde23()` calls `mlfFeval()`, `mlfFeval()` accesses the library's built-in function table to locate the function pointers that are associated with a given function name, in this example, the string `"lorenz"`.

3 Assign values to the equation parameters: `SIGMA`, `RHO`, and `BETA`. These parameters are shared between the main program and the `lorenz` function. The `lorenz` function uses the parameters in its computation of the values of the Lorenz equations.

4 Create two arrays, `tsm` and `ysm`, which are passed as input arguments to the `mlfOde23` function. Initialize `tsm` to the values stored in `tspan`. Initialize `ysm` to the values stored in `y0`.

5 Call the library routine `mlfOde23()`. The return value and the first argument store results. `mlfOde23()` is a `varargout` function; `mlfVarargout(NULL)` indicates that you are not interested in supplying any `varargout` arguments. Pass the name of the function, two required input arguments, and `NULL` values for the two optional input arguments.

   `mlfOde23()` calls `mlfFeval()` to evaluate the `lorenz` function. `mlfFeval()` searches the function table for a given function name. When it finds a match, it composes a call to the thunk function that it finds in the table, passing the thunk function the pointer to the function to be executed, also found in the table. In addition, `mlfFeval()` passes the thunk function arrays of input and output arguments. The thunk function actually executes the target function.

6 Prepare results for printing. The output consists of four columns. The first column is the time step and the other columns are the value of the function at that time step. The values are returned in one long column vector. If there

are n time steps, the values in column 1 occupy positions 0 through n-1 in the result, the values in column 2, positions n through 2n-1, and so on.

7 Print one line for each time step. The number of time steps is determined by the number of rows in the array `tm` returned from `mlfOde23`. The function `mxGetM` returned the number of rows in its `mxArray` argument.

8 Free all bound arrays and exit.

Output

The output from this program is several pages long. Here are the last lines of the output.

```
     t         y1         y2         y3
   9.390     41.218     12.984      2.951
   9.405     39.828     11.318      0.498
   9.418     38.530      9.995     -0.946
   9.430     37.135      8.678     -2.043
   9.442     35.717      7.404     -2.836
   9.455     34.229      6.117     -3.409
   9.469     32.711      4.852     -3.778
   9.484     31.185      3.632     -3.972
   9.500     29.657      2.477     -4.029
   9.518     28.123      1.402     -3.989
   9.539     26.563      0.415     -3.899
   9.552     25.635     -0.116     -3.845
   9.565     24.764     -0.576     -3.807
   9.580     23.861     -1.014     -3.796
   9.598     22.818     -1.478     -3.833
   9.620     21.682     -1.948     -3.964
   9.645     20.488     -2.429     -4.245
   9.674     19.280     -2.960     -4.761
   9.709     18.143     -3.618     -5.642
   9.750     17.275     -4.545     -7.097
   9.798     17.162     -6.000     -9.461
   9.843     18.378     -7.762    -12.143
   9.873     20.156     -9.147    -13.971
   9.903     22.821    -10.611    -15.464
   9.931     26.021    -11.902    -16.150
   9.960     29.676    -12.943    -15.721
```

```
  9.988    32.932   –13.430   –14.014
 10.000    34.012   –13.439   –12.993
```

Replacing Argument Lists with a Cell Array

In MATLAB you can substitute a cell array for a comma-separated list of MATLAB variables when you pass input arguments to a function. MATLAB treats the contents of each cell as a separate input argument. To trigger this functionality, you specify multiple values by indexing into the cell array with, for example, the colon index or a vector index.

For example, the MATLAB expression

```
T{1:5}
```

when passed as an input argument is equivalent to a comma-separated list of the contents of the first five cells of `T`. Simply passing the cell array `T` produces an error.

The MATLAB C Math Library also supports the expansion of the contents of a cell array into separate input arguments for library functions. For functions that implement MATLAB `varargin` functions, you use the indexing function `mlfIndexRef()` and a cell array index to obtain an array reference that returns multiple values.

For example, given the `varargin` function

```
void mlfVarargin_Func(mxArray *A, mxArray *B, ...);
```

you can make the following call:

```
mlfVarargin_Func(A,
                 B,
                 mlfIndexRef(C, "{?}",
                    mlfColon(mlfScalar(1), mlfScalar(5), NULL)),
                 NULL);
```

`A` and `B`, pointers to existing `mxArrays`, are passed as explicit arguments. `C` is a pointer to a cell array that contains at least five cells. The embedded call to `mlfIndexRef()` uses the index `{1:5}` to return multiple values: the first five cells of `C`. The MATLAB C Math Library passes these as individual arguments to `mlfVarargin_Func()`.

Positioning the Indexed Cell Array

- Pass the return from the cell array indexing operation as one of the variable-length arguments in the input argument list. That reference identifies multiple arrays.
- Do *not* pass the return from a cell array indexing operation as an explicit argument.

  For example, you *cannot* make this call to the example `varargin` function.

  ```
  mlfVarargin_Func(mlfIndexRef(C, "{?}",
                      mlfColon(mlfScalar(1), mlfScalar(5), NULL)),
                   A, B, NULL);
  ```

  Given the definition of `mlfVarargin_Func()`, the first argument position is reserved for an explicit, single argument. The MATLAB C Math Library does not handle multiple values in an explicit position.
- You can pass other array arguments or other cell array indexing expressions before or after a cell array indexing expression, all in the `...` argument positions.

See “Cell Array Indexing” on page 5-42 to learn more about indexing into cell arrays.

Exception for Built-In Library Functions

For built-in MATLAB C Math Library functions that are `varargin` functions, for example, `mlfCat()`, `mlfRand()`, and `mlfOnes()`), consider the explicit argument that immediately precedes the `...` as part of the `varargin` arguments. This argument *can* accept an indexed cell array expression.

For example, in this code the embedded call to `mlfIndexRef()` is in the position of the explicit argument that precedes the `...` in the signature of the built-in function `mlfCat()`.

```
/* In MATLAB:
 * F{1} = pascal(3);
 * F{2} = magic(3);
 * F{3} = ones(3);
 * F{4} = magic(3);
 * G = cat(2,F{:});
 */
```

```
mxArray *F = NULL, *G = NULL;

mlfIndexAssign(&F, "{?}", mlfScalar(1),
                mlfPascal(mlfScalar(3),NULL));
mlfIndexAssign(&F, "{?}", mlfScalar(2),
                mlfMagic(mlfScalar(3)));
mlfIndexAssign(&F, "{?}", mlfScalar(3),
                mlfOnes(mlfScalar(3),NULL));
mlfIndexAssign(&F, "{?}", mlfScalar(4),
                mlfMagic(mlfScalar(3)));

mlfAssign(&G, mlfCat(mlfScalar(2),
           mlfIndexRef(F, "{?}", mlfCreateColonIndex()), NULL));
```

Importing and Exporting Array Data

Introduction

The section describes how to:

- Write data to a MAT-file
- Read data from a MAT-file

A MAT-file is a binary, platform-independent file format that MATLAB uses to save workspace variables. Each MAT-file contains a machine signature in its file header. MATLAB reads this header and performs any conversions required by varying system architectures.

The MATLAB C Math Library functions `mlfSave()` and `mlfLoad()` implement the MATLAB `load` and `save` functions. You can use these routines to share data with MATLAB applications or with other applications developed with the MATLAB C++ or C Math Library.

Note The `mlfSave()` and `mlfLoad()` routines do not implement all the variations of the MATLAB `load` and `save` syntax.

Writing Data to a MAT-File

Using `mlfSave()`, you can save the data within `mxArray` variables to disk. The prototype for `mlfSave()` is

```
void mlfSave(mxArray *file, const char* mode, ... );
```

where `file` points to an `mxArray` containing the name of the MAT-file and `mode` points to a string that indicates whether you want to overwrite or update the data in the file. The variable argument list consists of at least one pair of arguments – the name you want to assign to the variable you're saving and the address of the `mxArray` variable that you want to save. The last argument to `mlfSave()` is always a `NULL`, which terminates the argument list:

- You must name each `mxArray` variable that you save to disk. A name can contain up to 32 characters.
- You can save as many variables as you want in a single call to `mlfSave()`.
- There is no call that globally saves all the variables in your program or in a particular function.

- The name of a MAT-file must end with the extension `.mat`. The library appends the extension `.mat` to the filename if you do not specify it.
- You can either overwrite or append to existing data in a file. Pass `"w"` to overwrite, `"u"` to update (append), or `"w4"` to overwrite using V4 format.
- The file created is a binary MAT-file, not an ASCII file.

Reading Data from a MAT-File

Using `mlfLoad()`, you can read in `mxArray` data from a binary MAT-file. The prototype for `mlfLoad()`

```
void mlfLoad(mxArray *file, ... );
```

where `file` points to an `mxArray` containing the name of the MAT-file and the variable argument list consists of at least one pair of arguments – the name of the variable that you want to load and a pointer to the address of an `mxArray` variable that will receive the data. The last argument to `mlfLoad()` is always a `NULL`, which terminates the argument list:

- You must indicate the name of each `mxArray` variable that you want to load.
- You can load as many variables as you want in one call to `mlfLoad()`.
- There is no call that loads all variables from a MAT-file globally.
- You do not have to allocate space for the incoming `mxArray`. `mlfLoad()` allocates the space required based on the size of the variable being read.
- You must specify a full path for the file that contains the data. The library appends the extension `.mat` to the filename if you do not specify it.
- You must load data from a binary MAT-file, not an ASCII MAT-file.

Note Be sure to transmit MAT-files in binary file mode when you exchange data between machines.

For more information on MAT-files, consult the online version of the *MATLAB Application Program Interface Guide*.

Example – Saving and Loading Data (ex5.c)

This example demonstrates how to use the functions mlfSave() and mlfLoad() to write data to a disk file and read it back again.

You can find the code for this example in the <matlab>/extern/examples/cmath directory, on UNIX systems, or the <matlab>\extern\examples\cmath directory, on PCs, where <matlab> represents the top-level directory of your installation. See “Building Stand-Alone C Applications” in Chapter 2 for information on building the examples.

```
    /* ex5.c */

    #include <stdlib.h>
1   #include "matlab.h"

    main()
    {
2       mxArray *x = NULL, *y = NULL, *z = NULL;
        mxArray *a = NULL, *b = NULL, *c = NULL;

        mlfEnterNewContext(0, 0);

3       mlfAssign(&x, mlfRand(mlfScalar(4),mlfScalar(4),NULL));
        mlfAssign(&y, mlfMagic(mlfScalar(7)));
        mlfAssign(&z, mlfEig(NULL, x, NULL));

4       mlfSave(mxCreateString("ex5.mat"), "w",
                    "x", x, "y", y, "z", z, NULL);

5       mlfLoad(mxCreateString("ex5.mat"),
                    "x", &a, "y", &b, "z", &c, NULL);

6       if (mlfTobool(mlfIsequal(a, x, NULL)) &&
            mlfTobool(mlfIsequal(b, y, NULL)) &&
            mlfTobool(mlfIsequal(c, z, NULL)))
        {
            mlfPrintf("Success: all variables equal.\n");
        }
        else
        {
            mlfPrintf("Failure: loaded values not equal to saved
                       values.\n");
        }

7       mxDestroyArray(x);
        mxDestroyArray(y);
        mxDestroyArray(z);
        mxDestroyArray(a);
        mxDestroyArray(b);
        mxDestroyArray(c);
```

```
    mlfRestorePreviousContext(0, 0);

    return(EXIT_SUCCESS);
}
```

The numbered items in the list below correspond to the numbered sections of code example:

1. Include "matlab.h". This file contains the declaration of the mxArray data structure and the prototypes for all the functions in the library. stdlib.h contains the definition of EXIT_SUCCESS.

2. Declare and initialize variables. x, y, and z will be written to the MAT-file using mlfSave(). a, b, and c will store the data read from the MAT-file by mlfLoad().

3. Assign data to the variables that will be saved to a file. x stores a 4-by-4 array that contains randomly generated numbers. y stores a 7-by-7 magic square. z contains the eigenvalues of x. Note that mlfRand() is a varargin function; you must terminate the argument list with NULL.

   The MATLAB C Math Library utility function mlfScalar() creates 1-by-1 arrays that hold an integer or double value.

4. Save three variables to the file "ex5.mat". You can save any number of variables to the file identified by the first argument to mlfSave(). The second argument specifies the mode for writing to the file. Here "w" indicates that mlfSave() should overwrite the data. Other values include "u" to update (append) and "w4" to overwrite using V4 format. Subsequent arguments come in pairs: the first argument in the pair (a string) labels the variable in the file; the contents of the second argument is written to the file. A NULL terminates the argument list.

   Note that you must provide a name for each variable you save. When you retrieve data from a file, you must provide the name of the variable you want to load. You can choose any name for the variable; it does not have to correspond to the name of the variable within the program. Unlike arguments to most MATLAB C Math Library functions, the variable names and mode are not mxArray arguments; you can pass a C string directly to mlfSave() and mlfLoad().

5 Load the named variables from the MAT-file. As arguments, you pass the name of the MAT-file (the string `ex5.mat`), the name/variable pairs, and a `NULL` to terminate the argument list. Note that, because you're loading data into a variable, `mlfLoad()` needs the address of the variables: `&a`, `&b`, `&c`. Notice how the name of the output argument does not have to match the name of the variable in the MAT-file.

Note `mlfLoad()` is not a type-safe function. It is declared as `mlfLoad(mxArray *file, ... )`. The compiler will not complain if you forget to include an `&` in front of the output arguments. However, your application will fail at runtime.

---

6 Compare the data loaded from the file to the original data that was written to the file. `a`, `b`, and `c` contain the loaded data. `x`, `y`, and `z` contain the original data. Each call to `mlfIsEqual()` returns a temporary scalar `mxArray` containing `TRUE` if the compared arrays are the same type and size, with identical contents. `mlfTobool()` returns the Boolean value contained in the array. The calls to `mlfTobool()` are necessary because C requires that the condition for an `if` statement be a scalar Boolean, not a scalar `mxArray`.

7 Free each of the matrices used in the examples.

Output

When run, the program produces this output:

```
Success: all variables equal.
```

8

Handling Errors and Writing a Print Handler

Introduction

This section describes how you can:

- Customize error handling by using the `mlfTry` and `mlfCatch` macros and by defining your own error handler. This section also provides an overview of default error handling.
- Customize printing by defining your own print handler. This sectionincludes information about displaying output to a GUI.

Error Handling Overview

The MATLAB C Math library routines handle error conditions in two ways, depending on the severity of the error:

- For less-severe error conditions, the library routines output a warning message to the user and then return control to the application. The application can continue processing.
- For more-severe error conditions, the library routines output a message to the user and then exit, terminating the application. Control never returns to the application.

The following example program illustrates this default library error handling. The program deliberately causes a warning-level error condition, division by zero, and a more severe error condition, attempting to add two matrices of unequal size, that causes termination.

```
#include <stdio.h>
#include <stdlib.h>     /* used for EXIT_SUCCESS */
#include <string.h>
#include "matlab.h"

/* Matrix data. Column-major element order */
static double data[] = { 1, 2, 3, 4, 5, 6 };

main()
{
    /* Declare matrix variables */
    mxArray *mat0 = NULL;
    mxArray *mat1 = NULL;
    mxArray *mat2 = NULL;

    mlfEnterNewContext(0,0);

    /* Create two matrices of different sizes */
    mlfAssign(&mat0, mlfDoubleMatrix(2, 3, data, NULL));
    mlfAssign(&mat1, mlfDoubleMatrix(3, 2, data, NULL));

    /* Division by zero will produce a warning */
    mlfAssign(&mat2, mlfRdivide(mat1, mlfScalar(0)));
```

```
        mlfPrintf("Return to application after warning.\n");

        /* Adding mismatched matrices produces error */
        mlfPrintMatrix(mlfPlus(mat0, mat1));
        mlfPrintf("Should not be reached after error.\n");

        /* Free any matrices that were assigned to variables */
        mxDestroyArray(mat2);
        mxDestroyArray(mat0);
        mxDestroyArray(mat1);

        mlfRestorePreviousContext(0, 0);
        return(EXIT_SUCCESS);
    }
```

This program produces the following output. You can see how this program continues processing after a warning, but terminates after an error.

```
WARNING: Divide by zero.
Return to application after warning.
ERROR: Matrix dimensions must agree.

EXITING
```

Customizing Error Handling

Two aspects of the default error handling behavior of the MATLAB C Math Library routines may not suit every application:

- Exiting on error conditions
- Displaying error messages to the same display as other application messages.

You may want control to return to your application when an error occurs, allowing it to perform clean up processing before exiting. You may also prefer to direct error and warning messages to a different output stream than the normal messages output by your application. The following sections describe how to make these customizations.

Continuing Processing After Errors

To return control to your application after a library routine encounters an error condition, you must define *try* and *catch* blocks in your application.

When you define a try block, the library *warning*-level processing does not change. The routines output a warning message and return control back to the application. The default library *error* processing, however, does change. When you define a try block, the library routines do not output an error message to the user and then exit. Instead, the routines transfer control to your catch block, which performs the error handling processing.

For example, if you want to output an error message to the user, you must do so from your catch block. (You can use the `mlfLasterr()` routine to obtain the text of the message associated with the most recent error that occurred.)

Defining a Try Block. A try block is a group of one or more statements, enclosed in braces, introduced by the `mlfTry` macro:

```
mlfTry
{
    /* your code */
}
```

Note that the `mlfTry` macro does not require parentheses; it is not a procedure call.

Defining a Catch Block. A catch block is a group of one or more routines, enclosed in braces, introduced by the `mlfCatch` macro and terminated by the `mlfEndCatch` macro:

```
mlfCatch
{
    /* Your code */
}
mlfEndCatch
```

The catch block contains error processing code. For example, you could put clean-up code in your catch block to free allocated storage before exiting. The `mlfCatch` and `mlfEndCatch` macros do not require parentheses.

Note While this error handling mechanism is modeled on the C++ exception handling facility, there is no connection between the two. The MATLAB C Math Library does not "throw" exceptions, as the MATLAB C++ Math Library does. The syntax used with the macros is different than the C++ keywords. The library error handling mechanism is built on the `setjmp/longjmp` mechanism. For more information about `setjmp/longjmp`, see your system documentation.

Example – Defining Try/Catch Blocks (ex6.c)

The following example adds try and catch blocks to the example program introduced in the previous section (page 8-3).

You can find the code for this example in the `<matlab>/extern/examples/cmath` directory, on UNIX systems, or the `<matlab>\extern\examples\cmath` directory, on PCs, where `<matlab>` represents the top-level directory of your installation. See "Building Stand-Alone C Applications" in Chapter 2 for information on building the examples.

```
        /* ex6.c */

        #include <stdio.h>
        #include <stdlib.h>     /* Used for EXIT_SUCCESS */
        #include <string.h>
        #include "matlab.h"

        static double data[] = { 1, 4, 2, 5, 3, 6 };

        main()
        {
1           mxArray *mat0 = NULL;
            mxArray *mat1 = NULL;
            mxArray *volatile mat2 = NULL;

            mlfEnterNewContext(0,0);

            mlfAssign(&mat0, mlfDoubleMatrix(2, 3, data, NULL));
2           mlfAssign(&mat1, mlfDoubleMatrix(3, 2, data, NULL));

3           mlfTry
            {
                mlfAssign(&mat2, mlfRdivide(mat1, mlfScalar(0)));

                mlfPrintf("Return to try block after warning.\n");

4               mlfPrintMatrix(mlfPlus(mat0, mat1));
            }
5           mlfCatch
            {
                mlfPrintf("In catch block. Caught an error: ");

6               mlfPrintMatrix(mlfLasterr(NULL));
            }
            mlfEndCatch

7           mlfPrintf("Now in application after catch block.\n");

            mxDestroyArray(mat0);
            mxDestroyArray(mat1);
```

```
    mxDestroyArray(mat2);

    mlfRestorePreviousContext(0, 0);

    return(EXIT_SUCCESS);
}
```

The numbered items in the list below correspond to the numbered sections of code example:

1 Variables that will be set within a try block must be declared as `volatile`. When a variable is declared as `volatile`, it is not stored in a register. You can, therefore, assign a value to the variable inside the try block and still retrieve the value.

2 `mlfTry` macro defines the beginning of the try block.

3 The exampe deliberately triggers a warning, by attempting to divide by zero, and an error, by calling `mlfPlus` with two input matrices of unequal size.

4 The `mlfCatch` macro defines the beginning of the catch block.

5 The error handling code in this catch block is quite simple. It displays a message, `In my catch block. Caught an error:`, and then prints out the message associated with the last error by passing the return value of the `mlfLastErr()` routine to the `mlfPrintMatrix()` routine.

6 The `mlfEndcatch` macro marks the end of the catch block.

7 After the catch block completes executing, the application continues, freeing the matrices, `mat0`, `mat1`, and `mat2`, which were used as input arguments to `mlfPlus()` and `mlfRdivide()`.

The program produces this output.

```
WARNING: Divide by zero.
Return to try block after warning.
In catch block. Caught an error: Matrix dimensions must agree.
Now in application after catch block.
```

A more sophisticated error handling mechanism could do much more than simply print an additional error message. If this statement were in a loop, for example, the code could discover the cause of the error, correct it, and try the operation again.

Replacing the Default Library Error Handler

The default error handling behavior of the MATLAB C Math Library routines is implemented by the default library *error handler* routine. You can further customize error handling by replacing the default library error handler routine with one of your own design.

Note Because of the error handling capabilities provided by the library try/catch mechanism, applications typically do not need to replace the default error handler. However, in some instances, it can be useful. For example, you can write an error handler that directs error messages to a file.

To replace the default error handler you must:

- Write an error handler
- Register your error handler so that library routines call it when they encounter an error.

Writing an Error Handler

When you write an error handler, you must conform to the library prototype for error handling routines:

```
void MyErrorHandler( const char *msg, bool isError  )
{
    /* Your code */
}
```

In this prototype, note the following:

- An error handling routine must not return a value (return void).
- An error handling routine accepts two arguments, a `const` string and a Boolean value. The string is the text of the error message. When the value of this Boolean value is `TRUE`, it indicates an error message. If this value is `FALSE`, it indicates a warning message.

Registering Your Error Handler

After writing an error handler, you must register it with the MATLAB C Math Library so that the library routines can call it when they encounter an error condition at runtime. You register an error handler using the `mlfSetErrorHandler()` routine.

```
mlfSetErrorHandler(MyErrorHandler);
```

Example – Adding an Error Handler

The following example program adds an error handler to the sample program introduced on page 8-3. This error handler writes error messages to a log file, identifying each message as an error or warning. An error handler like this allows a program to run unattended, since any errors produced are recorded in a file for future examination. More complex error-handling schemes are also possible. For example, you can use two files, one for the messages sent to the print handler and one for errors, or you can pipe the error message to an e-mail program that sends a notification of the error to the user who started the program.

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include "matlab.h"

1   FILE *stream;

2   void MyErrorHandler(const char *msg, bool isError  )
    {
        if(isError)
        {
            fprintf( stream, "ERROR: %s \n", msg );
        }
        else
        {
            fprintf( stream, "WARNING: %s \n", msg );
        }
    }
    static double data[] = { 1, 2, 3, 4, 5, 6 };

    main()
    {
        mxArray *mat0 = NULL;
        mxArray *mat1 = NULL;
        mxArray *volatile mat2 = NULL;

        mlfSetErrorHandler(MyErrorHandler);

        stream = fopen("myerrlog.out","w");

3       mlfEnterNewContext(0,0);

4       mlfAssign(&mat0, mlfDoubleMatrix(2, 3, data, NULL));
        mlfAssign(&mat1, mlfDoubleMatrix(3, 2, data, NULL));
        mlfTry
        {
            mlfAssign(&mat2, mlfRdivide(mat1, mlfScalar(0)));
            printf("Return to try block after warning.\n");

            mlfPrintMatrix(mlfPlus(mat0, mat1));
```

```
            mxDestroyArray(mat2);
        }
        mlfCatch
        {
            mlfPrintf("In catch block. Caught an error: ");
            mlfPrintMatrix(mlfLasterr(NULL));

            if (mat2)
                mxDestroyArray(mat2);
        }
        mlfEndCatch

        mlfPrintf("Now in application after catch block.");

        mxDestroyArray(mat0);
        mxDestroyArray(mat1);
        mlfRestorePreviousContext(0, 0);

        fclose(stream);
        return(EXIT_SUCCESS);
    }
```

The numbered items in the list below correspond to the numbered sections of code example:

1. The example program declares a variable, `stream`, that is a pointer to the output log file.
2. The error handling routine, `MyErrorHandler`, determines the type of error message, and writes warnings and errors to a log file.
3. The main program opens the log file, named `myerrlog.out`. The program calls `fclose()` to close the log file before exiting.
4. Register the error handler with the MATLAB C Math Library with this call to `mlfSetErrorHandler()`.

Output

The program produces this output.

```
In MyErrorHandler. WARNING: Divide by zero.
Logging the warning to a file.
Returning to try block after warning.
In MyErrorHandler. Logging the error to a file.
In catch block. Caught an error: Matrix dimensions must agree.
Now in application after catch block.
```

Defining a Print Handler

In the past, when there were only character-based terminals, input and output were very simple; programs used scanf for input and printf for output. Graphical user interfaces (GUIs) make input and output routines more complex. The MATLAB C Math Library is designed to run on both character-based terminals and in graphical, windowed environments. Simply using printf or a similar routine is fine for character-terminal output, but insufficient for output in a graphical environment.

The MATLAB C Math Library performs some output; in particular it displays error messages and warnings, but performs no input. To support programming in a graphical environment, the library allows you to determine how the library displays output.

The MATLAB C Math Library's output requirements are very simple. The library formats its output into a character string internally, and then calls a function that prints the single string. If you want to change where or how the library's output appears, you must provide an alternate print handler.

Providing Your Own Print Handler

Instead of calling printf directly, the MATLAB C Math Library calls a print handler when it needs to display an error message or warning. The default print handler used by the library takes a single argument, a const char * (the message to be displayed), and returns void.

The default print handler is

```
static void DefaultPrintHandler(const char *s)
{
    printf("%s",s);
}
```

The routine sends its output to C's stdout, using the printf function.

If you want to perform a different style of output, you can write your own print handler and register it with the MATLAB C Math Library. Any print handler that you write must match the prototype of the default print handler: a single const char * argument and a void return.

To register your function and change which print handler the library uses, you must call the routine mlfSetPrintHandler.

`mlfSetPrintHandler` takes a single argument, a pointer to a function that displays the character string, and returns `void`.

```
void mlfSetPrintHandler ( void ( * PH)(const char *) );
```

Sending Output to a GUI

When you write a program that runs in a graphical windowed environment, you may want to display printed messages in an informational dialog box. The next two sections illustrate how to provide an alternate print handler under the X Window System and Microsoft Windows.

Each example demonstrates the interface between the MATLAB C Math Library and the windowing system. In particular, the examples do not demonstrate how to write a complete, working program.

Each example assumes that you know how to write a program for a particular windowing system. The code contained in each example is incomplete. For example, application start up and initialization code is missing. Consult your windowing system's documentation if you need more information than the examples provide.

Each example presents a simple alternative output mechanism. There are other output options as well, for example, sending output to a window or portion of a window inside an application. The code in these examples should serve as a solid foundation for writing more complex output routines.

Note If you use an alternate print handler, you must call `mlfSetPrintHandler` before calling other library routines. Otherwise the library uses the default print handler to display messages.

Example – Writing Output to X Windows/Motif

The Motif Library provides a `MessageDialog` widget, which this example uses to display text messages. The `MessageDialog` widget consists of a message text area placed above a row of three buttons labeled **OK**, **Cancel**, and **Help**.

The message box is a modal dialog box; once it displays, you must dismiss it before the application will accept other input. However, because the `MessageDialog` is a child of the application and not the root window, other applications continue to operate normally.

```
/* X-Windows/Motif Example */

/* List other X include files here */
#include <Xm/Xm.h>
#include <Xm/X11.h>
#include <Xm/MessageB.h>

static Widget message_dialog = 0;

/* The alternate print handler */
void PopupMessageBox(const char *message)
{
    Arg args[1];

    XtSetArg(args[0], XmNmessageString, message);
    XtSetValues(message_dialog, args, 1);
    XtPopup(message_dialog, XtGrabExclusive);
}

main()
{
    /* Start X application. Insert your own code here. */
    main_window = XtAppInitialize( /* your code */ );

    /* Create the message box widget as a child of */
    /* the main application window. */
    message_dialog = XmCreateMessageDialog(main_window,
                        "MATLAB Message", 0, 0);

    /* Set the print handler. */
    mlfSetPrintHandler(PopupMessageBox);

    /* The rest of your program */
}
```

This example declares two functions: `PopupMessageBox()` and `main()`. `PopupMessageBox` is the print handler and is called every time the library needs to display a text message. It places the message text into the `MessageDialog` widget and makes the dialog box visible.

The second routine, main, first creates and initializes the X Window System application. This code is not shown, since it is common to most applications, and can be found in your X Windows reference guide. main then creates the MessageDialog object that is used by the print handling routine. Finally, main calls mlfSetPrintHandler to inform the library that it should use PopupMessageBox instead of the default print handler. If this were a complete application, the main routine would also contain calls to other routines or code to perform computations.

Example – Writing Output to Microsoft Windows

This example uses the Microsoft Windows MessageBox dialog box. This dialog box contains an “information” icon, the message text, and a single **OK** button. The MessageBox is a Windows modal dialog box; while it is posted, your application will not accept other input. You must press the **OK** button to dismiss the MessageBox dialog box before you can do anything else.

This example declares two functions. The first, PopupMessageBox, is responsible for placing the message into the MessageBox and then posting the box to the screen. The second, main, which in addition to creating and starting the Microsoft Windows application (that code is not shown) calls mlfSetPrintHandler to set the print handling routine to PopupMessageBox.

```
/* Microsoft Windows example */

static HWND window;
static LPCSTR title = "Message from MATLAB";

/* The alternate print handler */
void PopupMessageBox(const char *message)
{
    MessageBox(window, (LPCTSTR)message, title,
               MB_ICONINFORMATION);
}

main()
{
    /* Register window class, provide window procedure */
    /* Fill in your own code here. */

    /* Create application main window */
```

```
    window = CreateWindowEx( /* your specification */ );

    /* Set print handler */
    mlfSetPrintHandler(PopupMessageBox);

    /* The rest of the program ... */
}
```

This example does no real processing. If it were a real program, the main routine would contain calls to other routines or perform computations of its own.

9

Library Routines

Introduction

This section lists all the functions in the MATLAB C Math Library by category.

The functions are divided into three sections:

- “The MATLAB Built-In Library” on page 9-4
- “MATLAB M-File Math Library” on page 9-24
- “Array Access and Creation Library” on page 9-46

The tables that group the functions into categories include a short description of each function. Refer to the online *MATLAB C Math Library Reference* for a complete definition of the function syntax and arguments.

Organization of the MATLAB Math Libraries

The MATLAB functions within the MATLAB C Math Library are delivered as two libraries: the MATLAB Built-In Library and the MATLAB M-File Math Library. The Built-In Library contains the functions that every program using the MATLAB C Math Library needs, including, for example, the elementary mathematical functions that perform matrix addition and multiplication. The M-File Math Library is larger than the Built-In Library and contains more specialized functions, such as polynomial root finding or the two-dimensional inverse discrete Fourier transformation. Both libraries follow the same uniform naming convention and obey the same calling conventions.

MATLAB C Math Library programs link dynamically against both math libraries, in addition to the Array Access and Creation Library. (See “Building Stand-Alone C Applications” in Chapter 2 for a complete list of the required libraries.)

The MATLAB Built-In Library

The routines in the MATLAB Built-In Library fall into three categories:

- C callable versions of MATLAB built-in functions

  Each Math Library built-in routine is named after its MATLAB equivalent. For example, the `mlfTan` function is the C callable version of the MATLAB built-in `tan` function.
- C routine versions of the MATLAB operators

  For example, the C callable version of the MATLAB matrix multiplication operator (`*`) is the function named `mlfMtimes()`.
- Routines that initialize and control how the library operates

  These routines do not have a MATLAB equivalent. For example, there is no MATLAB equivalent for the `mlfSetPrintHandler()` routine.

Note You can recognize routines in the Built-In and M-File libraries by the `mlf` prefix at the beginning of each function name.

General Purpose Commands

Managing Variables

Function	Purpose
mlfFormat	Set output format.
mlfLoad	Retrieve variables from disk.
mlfSave	Save variables to disk.

Operators and Special Functions

Arithmetic Operator Functions

Function	Purpose
mlfLdivide	Array left division (.\).
mlfMinus	Array subtraction (-).
mlfMldivide	Matrix left division (\).
mlfMpower	Matrix power (^).
mlfMrdivide	Matrix right division (/).
mlfMtimes	Matrix multiplication (*).
mlfPlus	Array addition (+).
mlfPower	Array power (.^).
mlfRdivide	Array right division (./).
mlfTimes	Array multiplication (.*).
mlfUnaryminus, mlfUminus	Unary minus.

Relational Operator Functions

Function	Purpose
mlfEq	Equality (==).
mlfGe	Greater than or equal to (>=).
mlfGt	Greater than (>).
mlfLe	Less than or equal to (<=).
mlfLt	Less than (<).
mlfNeq, mlfNe	Inequality (~=).

Logical Operator Functions

Function	Purpose
mlfAll	True if all elements of vector are nonzero.
mlfAnd	Logical AND (&).
mlfAny	True if any element of vector is nonzero.
mlfNot	Logical NOT (~).
mlfOr	Logical OR (|).

Set Operators

Function	Purpose
`mlfIsmember`	True for set member.
`mlfSetdiff`	Set difference.
`mlfSetxor`	Set exclusive OR.
`mlfUnion`	Set union.
`mlfUnique`	Set unique.

Special Operator Functions

Function	Purpose
`mlfColon`	Create a sequence of indices.
`mlfCreateColonIndex`	Create an array that acts like the colon operator (:).
`mlfCtranspose`	Complex Conjugate Transpose (').
`mlfEnd`	Index to the end of an array dimension.
`mlfHorzcat`	Horizontal concatenation.
`mlfTranspose`	Noncomplex conjugate transpose (.')
`mlfVertcat`	Vertical concatenation.

Logical Functions

Function	Purpose
mlfFind	Find indices of nonzero elements.
mlfFinite	Extract only finite elements from array.
mlfIsa	True if object is a given class.
mlfIschar	True for character arrays (strings).
mlfIsempty	True for empty array.
mlfIsequal	True for input arrays of the same type, size, and contents.
mlfIsfinite	True for finite elements of an array.
mlfIsinf	True for infinite elements.
mlfIsletter	True for elements of the string that are letters of the alphabet.
mlfIslogical	True for logical arrays.
mlfIsnan	True for Not-a-Number.
mlfIsreal	True for noncomplex matrices.
mlfIsspace	True for whitespace characters in string matrices.
mlfLogical	Convert numeric values to logical.
mlfTobool	Convert an array to a Boolean value by reducing the rank of the array to a scalar.

MATLAB as a Programming Language

Function	Purpose
mlfFeval	Function evaluation.
mlfLasterr	Last error message.
mlfMfilename	Return a NULL array. M-file execution does not apply to stand-alone applications.

Message Display

Function	Purpose
mlfError	Display message and abort function.
mlfLastError	Return string that contains the last error message.
mlfWarning	Display warning message.

Elementary Matrices and Matrix Manipulation

Elementary Matrices

Function	Purpose
mlfEye	Identity matrix.
mlfOnes	Matrix of ones (1s).
mlfRand	Uniformly distributed random numbers.
mlfRandn	Normally distributed random numbers.
mlfZeros	Matrix of zeros (0s).

Basic Array Information

Function	Purpose
mlfDisp	Display text or array.
mlfIsempty	True for empty array.
mlfIsequal	True for input arrays of the same type, size, and contents.
mlfIslogical	True for logical arrays.
mlfLength	Length of vector.
mlfLogical	Convert numeric values to logical.
mlfNdims	Number of dimensions.
mlfSize	Size of array.

Special Constants

Function	Purpose
mlfComputer	Computer type.
mlfEps	Floating-point relative accuracy.

Special Constants (Continued)

Function	Purpose
`mlfFlops`	Floating-point operation count. (Not reliable in stand-alone applications.)
`mlfI`	Return an array with the value `0+1.0i`.
`mlfInf`	Infinity.
`mlfJ`	Return an array with the value `0+1.0i`.
`mlfNan`	Not–a–Number.
`mlfPi`	3.1415926535897....
`mlfRealmax`	Largest floating-point number.
`mlfRealmin`	Smallest floating-point number.

Matrix Manipulation

Function	Purpose
`mlfDiag`	Create or extract diagonals.
`mlfPermute`	Permute array dimensions.
`mlfTril`	Extract lower triangular part.
`mlfTriu`	Extract upper triangular part.

Specialized Matrices

Function	Purpose
`mlfMagic`	Magic square.

Elementary Math Functions

Trigonemetric Functions

Function	Purpose
mlfAcos	Inverse cosine.
mlfAsin	Inverse sine.
mlfAtan	Inverse tangent.
mlfAtan2	Four-quadrant inverse tangent.
mlfCos	Cosine.
mlfSin	Sine.
mlfTan	Tangent.

Exponential Functions

Function	Purpose
mlfExp	Exponential.
mlfLog	Natural logarithm.
mlfLog2	Base 2 logarithm and dissect floating-point numbers.
mlfPow2	Base 2 power and scale floating-point numbers.
mlfSqrt	Square root.

Complex Functions

Function	Purpose
mlfAbs	Absolute value.
mlfConj	Complex conjugate.
mlfImag	Imaginary part of a complex array.

Complex Functions (Continued)

Function	Purpose
`mlfIsreal`	True for noncomplex matrices
`mlfReal`	Real part of a complex array.

Rounding and Remainder Functions

Function	Purpose
`mlfCeil`	Round toward plus infinity.
`mlfFix`	Round toward zero.
`mlfFloor`	Round toward minus infinity.
`mlfRem`	Remainder after division.
`mlfRound`	Round to nearest integer.
`mlfSign`	Signum function.

Numerical Linear Algebra

Matrix Analysis

Function	Purpose
`mlfDet`	Determinant.
`mlfNorm`	Matrix or vector norm.
`mlfRcond`	LINPACK reciprocal condition estimator.

Linear Equations

Function	Purpose
`mlfChol`	Cholesky factorization.
`mlfCholupdate`	Rank 1 update to Cholesky factorization.

Linear Equations (Continued)

Function	Purpose
mlfInv	Matrix inverse.
mlfLu	Factors from Gaussian elimination.
mlfQr	Orthogonal-triangular decomposition.

Eigenvalues and Singular Values

Function	Purpose
mlfEig	Eigenvalues and eigenvectors.
mlfHess	Hessenberg form.
mlfQz	Generalized eigenvalues.
mlfSchur	Schur decomposition.
mlfSvd	Singular value decomposition.

Matrix Functions

Function	Purpose
mlfExpm	Matrix exponential.

Factorization Utilities

Function	Purpose
mlfBalance	Diagonal scaling to improve eigenvalue accuracy.

Data Analysis and Fourier Transform Functions

Basic Operations

Function	Purpose
mlfCumprod	Cumulative product of elements.
mlfCumsum	Cumulative sum of elements.
mlfMax	Largest component.
mlfMin	Smallest component.
mlfProd	Product of elements.
mlfSort	Sort in ascending order.
mlfSum	Sum of elements.

Filtering and Convolution

Function	Purpose
mlfFilter	One-dimensional digital filter (see online help).

Fourier Transforms

Function	Purpose
mlfFft	Discrete Fourier transform.
mlfFftn	Multidimensional fast Fourier transform.

Character String Functions

General

Function	Purpose
`mlfChar`	Create character array (string).
`mlfDouble`	Convert string to numeric character codes.

String Tests

Function	Purpose
`mlfIschar`	True for character arrays.
`mlfIsletter`	True for elements of the string that are letters of the alphabet.
`mlfIsspace`	True for whitespace characters in strings.

String Operations

Function	Purpose
`mlfLower`	Convert string to lower case.
`mlfStrcmp`	Compare strings.
`mlfStrcmpi`	Compare strings ignoring case.
`mlfStrncmp`	Compare the first `n` characters of two strings.
`mlfStrncmpi`	Compare first `n` characters of strings ignoring case.
`mlfUpper`	Convert string to upper case.

String to Number Conversion

Function	Purpose
`mlfSprintf`	Convert number to string under format control.
`mlfSscanf`	Convert string to number under format control.

File I/O Functions

File Opening and Closing

Function	Purpose
`mlfFclose`	Close file.
`mlfFopen`	Open file.

File Positioning

Function	Purpose
`mlfFeof`	Test for end-of-file.
`mlfFerror`	Inquire file I/O error status.
`mlfFseek`	Set file position indicator.
`mlfFtell`	Get file position indicator.

Formatted I/O

Function	Purpose
`mlfFprintf`	Write formatted data to file.
`mlfFscanf`	Read formatted data from file.

Binary File I/O

Function	Purpose
`mlfFread`	Read binary data from file.
`mlfFwrite`	Write binary data to file.

String Conversion

Function	Purpose
`mlfSprintf`	Write formatted data to a string.
`mlfSscanf`	Read string under format control.

File Import/Export Functions

Function	Purpose
`mlfLoad`	Retrieve variables from disk.
`mlfSave`	Save variables to disk.

Data Types

Data Types

Function	Purpose
`mlfChar`	Create character array (string).
`mlfDouble`	Convert to double precision.

Object Functions

Function	Purpose
mlfClassName	Return a string representing an object's class.
mlfIsa	True if object is a given class.

Time and Dates

Current Date and Time

Function	Purpose
mlfClock	Wall clock.

Multidimensional Array Functions

Function	Purpose
mlfCat	Concatenate arrays.
mlfNdims	Number of array dimensions.
mlfPermute	Permute array dimensions.

Cell Array Functions

Function	Purpose
mlfCell	Create cell array.
mlfCell2struct	Convert cell array into structure array.
mlfCellhcat	Horizontally concatenate cell arrays.
mlfIscell	True for cell array.

Structure Functions

Function	Purpose
mlfFieldnames	Get structure field names.
mlfGetfield	Get structure field contents.
mlfSetfield	Set structure field contents.
mlfStruct	Create or convert to structure array.
mlfStruct2cell	Convert structure array into cell array.

Sparse Matrix Functions

Full to Sparse Conversion

Function	Purpose
mlfFind	Find indices of nonzero elements.
mlfFull	Convert sparse matrix to full matrix.
mlfSparse	Create sparse matrix.

Working with NonZero Entries of Sparse Matrices

Function	Purpose
mlfIssparse	True for sparse matrix.

Linear Algebra

Function	Purpose
mlfCholinc	Incomplete Cholesky factorization.
mlfLuinc	Incomplete LU factorization.

Utility Routines

The MATLAB C Math Library utility routines help you perform indexing, create scalar arrays, and initialize and control the library environment.

Error Handling

Function	Purpose
`void mlfSetErrorHandler(void (* EH)(const char *, bool));`	Specify pointer to external application's error handler function.

mlfFeval() Support

Function	Purpose
`void mlfFevalTableSetup(mlfFuncTab *mlfUfuncTable);`	Register a thunk function table with the MATLAB C Math Library.

Indexing

Function	Purpose
`mxArray * mlfIndexAssign(mxArray * volatile *pa, const char *index, ...);`	Handle assignments that include indexing.
`mxArray * mlfIndexDelete(mxArray * volatile *pa, const char *index, ...);`	Handle deletions that include indexing.
`mxArray * mlfIndexRef(mxArray *pa, const char* index_string, ...);`	Perform array references such as `X(5,:)`.
`mxArray * mlfColon(mxArray *start, mxArray *step, mxArray *end);`	Generate a sequence of indices. Use this where you'd use the colon operator (:) operator in MATLAB. `mlfColon(NULL, NULL,NULL)` is equivalent to `mlfCreateColonIndex()`.

Indexing (Continued)

Function	Purpose
`mxArray *` `mlfCreateColonIndex(void);`	Create an array that acts like the colon operator (:) when passed to `mlfArrayRef()`, `mlfArrayAssign()`, and `mlfArrayDelete()`.

`mxArray *`
`mlfEnd(mxArray *array, mxArray *dim,`
`               mxArray *numindices);`
	Generate the last index for an array dimension. Acts like `end` in the MATLAB expression `A(3,6:end)`. `dim` is the dimension to compute `end` for. Use 1 to indicate the row dimension; use 2 to indicate the column dimension. `numindices` is the number of indices in the subscript.

Memory Allocation

Function	Purpose

`void`
`mlfSetLibraryAllocFcns (calloc_proc calloc_fcn,`
`               free_proc free_fcn,`
`               realloc_proc realloc_fcn,`
`               malloc_proc malloc_fcn);`
	Set the MATLAB C Math Library's memory management functions. Gives you complete control over memory management.

Printing

Function	Purpose
`int` `mlfPrintf(const char *fmt, ...);`	Format output just like `printf`. Use the installed print handler to display the output.

Printing (Continued)

Function	Purpose
`void mlfPrintMatrix(mxArray *m);`	Print contents of matrix.
`void mlfSetPrintHandler(void (* PH)(const char *));`	Specify pointer to external application's output function.

Scalar Array Creation

Function	Purpose
`mxArray * mlfScalar (double v);`	Create a 1-by-1 `array` whose contents are initialized to the value of `v`.
`mxArray * mlfComplexScalar(double v, double i);`	Create a complex 1-by-1 array whose contents are initialized to the real part `v` and the imaginary part `i`.

MATLAB M-File Math Library

The MATLAB M-File Math Library contains callable versions of the M-files in MATLAB. For example, MATLAB implements the function `rank` in an M-file named `rank.m`. The C callable version of `rank` is called `mlfRank`.

Note You can recognize routines in the Built-In and M-File Libraries by the `mlf` prefix at the beginning of each function.

Operators and Special Functions

Arithmetic Operator Functions

Function	Purpose
`mlfKron`	Kronecker tensor product.

Logical Operator Functions

Function	Purpose
`mlfXor`	Logical exclusive-or operation.

Set Operators

Function	Purpose
`mlfIntersect`	Set intersection of two vectors.

Logical Functions

Function	Purpose
`mlfIsieee`	True for IEEE floating-point arithmetic.
`mlfIsspace`	True for whitespace characters in string matrices.

Logical Functions (Continued)

Function	Purpose
mlfIsstudent	True for student editions of MATLAB.
mlfIsunix	True on UNIX machines.
mlfIsvms	True on computers running DEC's VMS.

MATLAB As a Programming Language

Function	Purpose
mlfNargchk	Validate number of input arguments.
mlfXyzchk	Check arguments to 3-D data routines.

Elementary Matrices and Matrix Manipulation

Elementary Matrices

Function	Purpose
mlfAutomesh	True if the inputs require automatic meshgriding.
mlfLinspace	Linearly spaced vector.
mlfLogspace	Logarithmically spaced vector.
mlfMeshgrid	X and Y arrays for 3-D plots.

Basic Array Information

Function	Purpose
mlfIsnumeric	True for numeric arrays.

Matrix Manipulation

Function	Purpose
mlfCat	Concatenate arrays.
mlfFliplr	Flip matrix in the left/right direction.
mlfFlipud	Flip matrix in the up/down direction.
mlfIpermute	Inverse permute array dimensions.
mlfRepmat	Replicate and tile an array.
mlfReshape	Change size.
mlfRot90	Rotate matrix 90 degrees.
mlfShiftdim	Shift dimensions.

Specialized Matrices

Function	Purpose
mlfCompan	Companion matrix.
mlfHadamard	Hadamard matrix.
mlfHankel	Hankel matrix.
mlfHilb	Hilbert matrix.
mlfInvhilb	Inverse Hilbert matrix.
mlfPascal	Pascal matrix.
mlfRosser	Classic symmetric eigenvalue test problem.
mlfToeplitz	Toeplitz matrix.
mlfVander	Vandermonde matrix.
mlfWilkinson	Wilkinson's eigenvalue test matrix.

Elementary Math Functions

Trignometric Functions

Function	Purpose
mlfAcosh	Inverse hyperbolic cosine.
mlfAcot	Inverse cotangent.
mlfAcoth	Inverse hyperbolic cotangent.
mlfAcsc	Inverse cosecant.
mlfAcsch	Inverse hyperbolic cosecant.
mlfAsec	Inverse secant.
mlfAsech	Inverse hyperbolic secant.
mlfAsinh	Inverse hyperbolic sine.
mlfAtanh	Inverse hyperbolic tangent.
mlfCosh	Hyperbolic cosine.
mlfCot	Cotangent.
mlfCoth	Hyperbolic cotangent.
mlfCsc	Cosecant.
mlfCsch	Hyperbolic cosecant.
mlfSec	Secant.
mlfSech	Hyperbolic secant.
mlfSinh	Hyperbolic sine.
mlfTanh	Hyperbolic tangent.

Exponential Functions

Function	Purpose
mlfLog10	Common (base 10) logarithm.
mlfNextpow2	Next higher power of 2.

Complex Functions

Function	Purpose
mlfAngle	Phase angle.
mlfCplxpair	Sort numbers into complex conjugate pairs.
mlfUnwrap	Remove phase angle jumps across 360° boundaries.

Rounding and Remainder Functions

Function	Purpose
mlfMod	Modulus (signed remainder after division).

Specialized Math Functions

Specialized Math Functions

Function	Purpose
mlfBeta	Beta function.
mlfBetainc	Incomplete beta function.
mlfBetaln	Logarithm of beta function.
mlfCross	Vector cross product.
mlfEllipj	Jacobi elliptic functions.
mlfEllipke	Complete elliptic integral.
mlfErf	Error function.
mlfErfc	Complementary error function.
mlfErfcx	Scaled complementary error function.
mlfErfinv	Inverse error function.
mlfExpint	Exponential integral function.
mlfGamma	Gamma function.
mlfGammainc	Incomplete gamma function.
mlfGammaln	Logarithm of gamma function.
mlfLegendre	Legendre functions.

Number Theoretic Functions

Function	Purpose
mlfFactor	Prime factors.
mlfGcd	Greatest common divisor.
mlfIsprime	True for prime numbers.

Number Theoretic Functions (Continued)

Function	Purpose
mlfLcm	Least common multiple.
mlfNchoosek	All combinations of n elements taken k at a time.
mlfPerms	All possible permutations.
mlfPrimes	Generate list of prime numbers.
mlfRat	Rational approximation.
mlfRats	Rational output.

Coordinate System Transforms

Function	Purpose
mlfCart2pol	Transform Cartesian coordinates to polar.
mlfCart2sph	Transform Cartesian coordinates to spherical.
mlfPol2cart	Transform polar coordinates to Cartesian.
mlfSph2cart	Transform spherical coordinates to Cartesian.

Numerical Linear Algebra

Matrix Analysis

Function	Purpose
`mlfNormest`	Estimate the matrix 2-norm.
`mlfNull`	Orthonormal basis for the null space.
`mlfOrth`	Orthonormal basis for the range.
`mlfRank`	Number of linearly independent rows or columns.
`mlfRref`	Reduced row echelon form.
`mlfSubspace`	Angle between two subspaces.
`mlfTrace`	Sum of diagonal elements.

Linear Equations

Function	Purpose
`mlfCond`	Condition number with respect to inversion.
`mlfCondest`	1-norm condition number estimate.
`mlfLscov`	Least squares in the presence of known covariance.
`mlfNnls`	Nonnegative least-squares.
`mlfPinv`	Pseudoinverse.

Eigenvalues and Singular Values

Function	Purpose
`mlfCondeig`	Condition number with respect to eigenvalues.
`mlfPoly`	Characteristic polynomial.
`mlfPolyeig`	Polynomial eigenvalue problem.

Matrix Functions

Function	Purpose
`mlfFunm`	Evaluate general matrix function.
`mlfLogm`	Matrix logarithm.
`mlfSqrtm`	Matrix square root.

Factorization Utilities

Function	Purpose
`mlfCdf2rdf`	Complex diagonal form to real block diagonal form.
`mlfPlanerot`	Generate a Givens plane rotation.
`mlfQrdelete`	Delete a column from a QR factorization.
`mlfQrinsert`	Insert a column into a QR factorization.
`mlfRsf2csf`	Real block diagonal form to complex diagonal form.

Data Analysis and Fourier Transform Functions

Basic Operations

Function	Purpose
mlfCumtrapz	Cumulative trapezoidal numerical integration.
mlfMean	Average or mean value.
mlfMedian	Median value.
mlfSortrows	Sort rows in ascending order.
mlfStd	Standard deviation.
mlfTrapz	Numerical integration using trapezoidal method.

Finite Differences

Function	Purpose
mlfDel2	Five-point discrete Laplacian.
mlfDiff	Difference function and approximate derivative.
mlfGradient	Approximate gradient (see online help).

Correlation

Function	Purpose
mlfCorrcoef	Correlation coefficients.
mlfCov	Covariance matrix.
mlfSubspace	Angle between two subspaces.

Filtering and Convolution

Function	Purpose
mlfConv	Convolution and polynomial multiplication.
mlfConv2	Two-dimensional convolution (see online help).
mlfDeconv	Deconvolution and polynomial division.
mlfFilter2	Two–dimensional digital filter (see online help).

Fourier Transforms

Function	Purpose
mlfFft2	Two-dimensional discrete Fourier transform.
mlfFftshift	Shift DC component to center of spectrum.
mlfIfft	Inverse discrete Fourier transform.
mlfIfft2	Two-dimensional inverse discrete Fourier transform.
mlfIfftn	Inverse multidimensional fast Fourier transform.

Sound and Audio

Function	Purpose
mlfFreqspace	Frequency spacing for frequency response.
mlfLin2mu	Convert linear signal to mu-law encoding.
mlfMu2lin	Convert mu-law encoding to linear signal.

Polynomial and Interpolation Functions

Data Interpolation

Function	Purpose
`mlfGriddata`	Data gridding.
`mlfIcubic`	Cubic interpolation of 1-D function.
`mlfInterp1`	One-dimensional interpolation (1-D table lookup).
`mlfInterp1q`	Quick one-dimensional linear interpolation.
`mlfInterp2`	Two-dimensional interpolation (2-D table lookup).
`mlfInterpft`	One-dimensional interpolation using FFT method.

Spline Interpolation

Function	Purpose
`mlfPpval`	Evaluate piecewise polynomial.
`mlfSpline`	Piecewise polynomial cubic spline interpolant.

Geometric Analysis

Function	Purpose
`mlfInpolygon`	Detect points inside a polygonal region.
`mlfPolyarea`	Area of polygon.
`mlfRectint`	Rectangle intersection area.

Polynomials

Function	Purpose
mlfConv	Multiply polynomials.
mlfDeconv	Divide polynomials.
mlfMkpp	Make piecewise polynomial.
mlfPoly	Construct polynomial with specified roots.
mlfPolyder	Differentiate polynomial (see online help).
mlfPolyfit	Fit polynomial to data.
mlfPolyval	Evaluate polynomial.
mlfPolyvalm	Evaluate polynomial with matrix argument.
mlfResidue	Partial-fraction expansion (residues).
mlfResi2	Residue of a repeated pole.
mlfRoots	Find polynomial roots.
mlfUnmkpp	Supply information about piecewise polynomial.

Function-Functions and ODE Solvers

Optimization and Root Finding

Function	Purpose
`mlfFmin`	Minimize function of one variable.
`mlfFmins`	Minimize function of several variables.
`mlfFoptions`	Set minimization options.
`mlfFzero`	Find zero of function of one variable.

Numerical Integration (Quadrature)

Function	Purpose
`mlfDblquad`	Numerical double integration.
`mlfQuad`	Numerically evaluate integral, low order method.
`mlfQuad8`	Numerically evaluate integral, high order method.

Ordinary Differential Equation Solvers

Function	Purpose
`mlfOde23`	Solve differential equations, low order method.
`mlfOde45`	Solve differential equations, high order method.
`mlfOde113`	Solve nonstiff differential equations, variable order method.
`mlfOde15s`	Solve stiff differential equations, variable order method.
`mlfOde23s`	Solve stiff differential equations, low order method.

ODE Option Handling

Function	Purpose
`mlfOdeget`	Extract properties from `options` structure created with `odeset`.
`mlfOdeset`	Create or alter `options` structure for input to ODE solvers.

Character String Functions

General

Function	Purpose
`mlfBlanks`	String of blanks.
`mlfDeblank`	Remove trailing blanks from a string.
`mlfStr2mat`	Form text array from individual strings.

String Operations

Function	Purpose
`mlfFindstr`	Find a substring within a string.
`mlfStrcat`	String concatenation.
`mlfStrjust`	Justify a character array.
`mlfStrmatch`	Find possible matches for a string.
`mlfStrrep`	Replace substrings within a string.
`mlfStrtok`	Extract tokens from a string.
`mlfStrvcat`	Vertical concatenation of strings.

String to Number Conversion

Function	Purpose
`mlfInt2str`	Convert integer to string.
`mlfMat2str`	Convert matrix to string.
`mlfNum2str`	Convert number to string.
`mlfStr2double`	Convert string to double-precision value.
`mlfStr2num`	Convert string to number.

Base Number Conversion

Function	Purpose
`mlfBase2dec`	Base to decimal number conversion.
`mlfBin2dec`	Binary to decimal number conversion.
`mlfDec2base`	Decimal number to base conversion.
`mlfDec2bin`	Decimal to binary number conversion.
`mlfDec2hex`	Decimal to hexadecimal number conversion.
`mlfHex2dec`	IEEE hexadecimal to decimal number conversion.
`mlfHex2num`	Hexadecimal to double number conversion.

File I/O Functions

Formatted I/O

Function	Purpose
`mlfFgetl`	Read line from file, discard newline character.
`mlfFgets`	Read line from file, keep newline character.

File Positioning

Function	Purpose
`mlfFrewind`	Rewind file pointer to beginning of file.

Time and Dates

Current Date and Time

Function	Purpose
`mlfDate`	Current date string.
`mlfNow`	Current date and time.

Basic Functions

Function	Purpose
`mlfDatenum`	Serial date number.
`mlfDatestr`	Date string format.
`mlfDatevec`	Date components.

Date Functions

Function	Purpose
mlfCalendar	Calendar.
mlfEomday	End of month.
mlfWeekday	Day of the week.

Timing Functions

Function	Purpose
mlfEtime	Elapsed time function.
mlfTic, mlfToc	Stopwatch timer functions.

Multidimensional Array Functions

Function	Purpose
mlfInd2sub	Subscript from linear index.
mlfIpermute	Inverse permute array dimensions.
mlfShiftdim	Shift dimensions.
mlfSub2ind	Linear index from multiple subscripts.

Cell Array Functions

Function	Purpose
mlfCelldisp	Display cell array contents.
mlfCellfun	Apply a cell function to a cell array.
mlfCellstr	Create cell array of strings from character array.
mlfDeal	Deal inputs to outputs.
mlfIscellstr	True for a cell array of strings.
mlfNum2cell	Convert numeric array into cell array.

Structure Functions

Function	Purpose
mlfIsfield	True if field is in structure array.
mlfIsstruct	True for structures.
mlfRmfield	Remove structure field.

Sparse Matrix Functions

Elementary Sparse Matrices

Function	Purpose
mlfSpdiags	Sparse matrix formed from diagonals.
mlfSpeye	Sparse identity matrix.
mlfSprand	Sparse uniformly distributed random matrix.
mlfSprandn	Sparse normally distributed random matrix.
mlfSprandsym	Sparse random symmetric matrix.

Full to Sparse Conversion

Function	Purpose
mlfSpconvert	Import from sparse matrix external format.

Working with NonZero Entries of Sparse Matrices

Function	Purpose
mlfNnz	Number of nonzero matrix elements.
mlfNonzeros	Nonzero matrix elements.
mlfNzmax	Amount of storage allocated for nonzero matrix elements.
mlfSpalloc	Allocate space for sparse matrix.
mlfSpfun	Apply function to nonzero matrix elements.
mlfSpones	Replace nonzero sparse matrix elements with ones.

Reordering Algorithms

Function	Purpose
`mlfColmmd`	Column minimum degree permutation.
`mlfColperm`	Column permutation.
`mlfDmperm`	Dulmage-Mendelsohn permutation.
`mlfRandperm`	Random permutation.
`mlfSymmmd`	Symmetric minimum degree permutation.
`mlfSymrcm`	Symmetric reverse Cuthill-McKee permutation.

Linear Algebra

Function	Purpose
`mlfCondest`	1-norm condition number estimate.
`mlfEigs`	A few eigenvalues.
`mlfNormest`	Estimate the matrix 2-norm.
`mlfSvds`	A few singular values.

Linear Equations (iterative methods)

Function	Purpose
`mlfBicg`	BiConjugate Gradients Method.
`mlfBicgstab`	BiConjugate Gradients Stabilized Method.
`mlfCgs`	Conjugate Gradients Squared Method.
`mlfGmres`	Generalized Minimum Residual Method.
`mlfPcg`	Preconditioned Conjugate Gradients Method.
`mlfQmr`	Quasi-Minimal Residual Method.

Miscellaneous

Function	Purpose
mlfSpaugment	Form least squares augmented system.
mlfSpparms	Set parameters for sparse matrix routines.
mlfSymbfact	Symbolic factorization analysis.

Array Access and Creation Library

The Array Access and Creation Library contains the array access routines for the `mxArray` data type. For example, `mxCreateDoubleMatrix()` creates an `mxArray; mxDestroyArray()` destroys one.

Refer to the online *Application Program Interface Reference* and the MATLAB *Application Program Interface Guide* for a detailed definition of each function.

Note You can recognize an Array Access and Creation Library routine by its prefix `mx`. The functions listed are a subset of the Array Access and Creation Library.

Array Access Routines

Function	Purpose
`mxCalloc, mxFree`	Allocate and free dynamic memory using MATLAB's memory manager.
`mxClearLogical`	Clear the logical flag.
`mxCreateCellArray`	Create an unpopulated N-dimensional cell `mxArray`.
`mxCreateCellMatrix`	Create an unpopulated 2-D cell `mxArray`.
`mxCreateCharArray`	Create an unpopulated N-dimensional string `mxArray`.
`mxCreateCharMatrixFromStrings`	Create a populated 2-D string `mxArray`.
`mxCreateDoubleMatrix`	Create an unpopulated 2-D, double-precision, floating-point `mxArray`.
`mxCreateNumericArray`	Create an unpopulated N-dimensional numeric `mxArray`.
`mxCreateSparse`	Create a 2-D unpopulated sparse `mxArray`.
`mxCreateString`	Create a 1-by-n string `mxArray` initialized to the specified string.

Array Access Routines (Continued)

Function	Purpose
mxCreateStructArray	Create an unpopulated N-dimensional structure mxArray.
mxCreateStructMatrix	Create an unpopulated 2-D structure mxArray.
mxDestroyArray	Free dynamic memory allocated by an mxCreate routine.
mxDuplicateArray	Make a deep copy of an array.
mxGetCell	Get a cell's contents.
mxGetClassID	Get (as an enumerated constant) an mxArray's class.
mxGetClassName	Get (as a string) an mxArray's class.
mxGetData	Get pointer to data.
mxGetDimensions	Get a pointer to the dimensions array.
mxGetElementSize	Get the number of bytes required to store each data element.
mxGetEps	Get value of eps.
mxGetField	Get a field value, given a field name and an index in a structure array.
mxGetFieldByNumber	Get a field value, given a field number and an index in a structure array.
mxGetFieldNameByNumber	Get a field name, given a field number in a structure array.
mxGetFieldNumber	Get a field number, given a field name in a structure array.
mxGetImagData	Get pointer to imaginary data of an mxArray.
mxGetInf	Get the value of infinity.
mxGetIr	Get the ir array of a sparse matrix.
mxGetJc	Get the jc array of a sparse matrix.

Array Access Routines (Continued)

Function	Purpose
`mxGetM, mxGetN`	Get the number of rows (`M`) and columns (`N`) of an array.
`mxGetName, mxSetName`	Get and set the name of an `mxArray`.
`mxGetNaN`	Get the value of Not-a-Number.
`mxGetNumberOfDimensions`	Get the number of dimensions.
`mxGetNumberOfElements`	Get number of elements in an array.
`mxGetNumberOfFields`	Get the number of fields in a structure `mxArray`.
`mxGetNzmax`	Get the number of elements in the `ir`, `pr`, and (if it exists) `pi` arrays.
`mxGetPi, mxGetPr`	Get the real and imaginary parts of an `mxArray`.
`mxGetScalar`	Get the real component from the first data element of an `mxArray`.
`mxGetString`	Copy the data from a string `mxArray` into a C-style string.
`mxIsChar`	True for a character array.
`mxIsClass`	True if `mxArray` is a member of the specified class.
`mxIsComplex`	True if data is complex.
`mxIsDouble`	True if `mxArray` represents its data as double-precision, floating-point numbers.
`mxIsEmpty`	True if `mxArray` is empty.
`mxIsFinite`	True if value is finite.
`mxIsInf`	True if value is infinite.
`mxIsInt8`	True if `mxArray` represents its data as signed 8-bit integers.
`mxIsInt16`	True if `mxArray` represents its data as signed 16-bit integers.

Array Access Routines (Continued)

Function	Purpose
`mxIsInt32`	True if `mxArray` represents its data as signed 32-bit integers.
`mxIsLogical`	True if `mxArray` is Boolean.
`mxIsNaN`	True if value is Not-a-Number.
`mxIsNumeric`	True if `mxArray` is numeric or a string.
`mxIsSingle`	True if `mxArray` represents its data as single-precision, floating-point numbers.
`mxIsSparse`	Inquire if an `mxArray` is sparse. Always false for the MATLAB C Math Library.
`mxIsStruct`	True if a structure `mxArray`.
`mxMalloc`	Allocate dynamic memory using MATLAB's memory manager.
`mxRealloc`	Reallocate memory.
`mxSetCell`	Set the value of one cell.
`mxSetData`	Set pointer to data.
`mxSetDimensions`	Modify the number of dimensions and/or the size of each dimension.
`mxSetField`	Set a field value of a structure array, given a field name and an index.
`mxSetFieldByNumber`	Set a field value in a structure array, given a field number and an index.
`mxSetImagData`	Set imaginary data pointer for an `mxArray`.
`mxSetIr`	Set the `ir` array of a sparse `mxArray`.
`mxSetJc`	Set the `jc` array of a sparse `mxArray`.
`mxSetLogical`	Set the logical flag.

Array Access Routines (Continued)

Function	Purpose
`mxSetM, mxSetN`	Set the number of rows (`M`) and columns (`N`) of an array.
`mxSetNzmax`	Set the storage space for nonzero elements.
`mxSetPi, mxSetPr`	Set the real and imaginary parts of an `mxArray`.

Fortran Interface

Function	Purpose
`mxCopyCharacterToPtr`	Copy `CHARACTER` values from Fortran to C pointer array.
`mxCopyPtrToCharacter`	Copy `CHARACTER` values from C pointer array to Fortran.
`mxCopyComplex16toPtr`	Copy `COMPLEX*16` values from Fortran to C pointer array.
`mxCopyPtrToComplex16`	Copy `COMPLEX*16` values to Fortran from C pointer array.
`mxCopyInteger4ToPtr`	Copy `INTEGER*4` values from Fortran to C pointer array.
`mxCopyPtrToInteger4`	Copy `INTEGER*4` values to Fortran from C pointer array.
`mxCopyReal8toPtr`	Copy `REAL*8` values from Fortran to C pointer array.
`mxCopyPtrToReal8`	Copy `REAL*8` values to Fortran from C pointer array.

A

Directory Organization

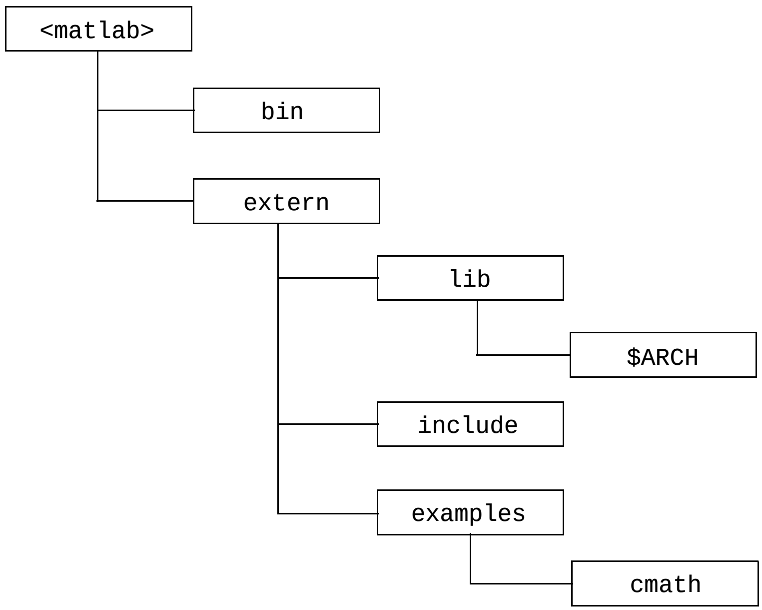
This section describes the directory organization of the MATLAB C Math Library on UNIX and on Microsoft Windows systems.

The MATLAB C Math Library is part of a family of tools offered by The MathWorks. All MathWorks products are stored under a single directory referred to as the MATLAB root directory.

Separate directories for the major product categories are located under the root. The MATLAB C Math Library is installed in the `extern` directory where products external to MATLAB are installed and in the `bin` directory. If you have other MathWorks products, there are additional directories directly below the root.

Directory Organization on UNIX

This figure illustrates the directory structure for the MATLAB C Math Library files on UNIX. `<matlab>` symbolizes the top-level directory where MATLAB is installed on your system. `$ARCH` specifies a particular UNIX platform.



<matlab>/bin

The `<matlab>/bin` directory contains the `mbuild` script and the scripts it uses to build your code.

`mbuild`	Shell script that controls the building and linking of your code.
`mbuildopts.sh`	Options file that controls the switches and options for your C compiler. It is architecture specific. When you execute `mbuild -setup`, this file is copied to your MATLAB root installation directory.

<matlab>/extern/lib/$ARCH

The `<matlab>/extern/lib/$ARCH` directory contains the binary library files; `$ARCH` specifies a particular UNIX platform. For example, on a Sun SPARCstation running Solaris, the `$ARCH` directory is `sol2`. The libraries that come with the MATLAB C Math Library are shown in this table:

`libmat.`*ext*	MAT-file access routines to support `mlfLoad` and `mlfSave`.
`libmatlb.`*ext*	MATLAB Built-In Math Library. Contains stand-alone versions of MATLAB built-in math functions and operators. Required for building stand-alone applications.
`libmi.`*ext*	Internal MAT-file access routines.
`libmmfile.`*ext*	MATLAB M-File Math Library. Contains stand-alone versions of the math M-files. Needed for building stand-alone applications that require MATLAB M-file math functions.
`libmx.`*ext*	MATLAB Array Access and Creation Library. Contains array creation and access routines.
`libut.`*ext*	MATLAB Utilities Library. Contains the utility routines used by various components.

The filename extension `.`*ext* is `.a` on IBM RS/6000; `.so` on Solaris, Alpha, Linux, and SGI; and `.sl` on HP 700. The libraries are shared libraries on all platforms.

<matlab>/extern/include

The `<matlab>/extern/include` directory contains the header files for developing MATLAB C Math Library applications. The header files associated with the MATLAB C Math Library are shown below.

`matlab.h`	Header file for the MATLAB C Math Library.
`libmatlb.h`	Header file containing the prototypes for the MATLAB Built-In Math Library functions.
`libmmfile.h`	Header file containing the prototypes for the MATLAB M-File Math Library functions.
`matrix.h`	Header file containing the definition of the `mxArray` type and function prototypes for array access routines.

<matlab>/extern/examples/cmath

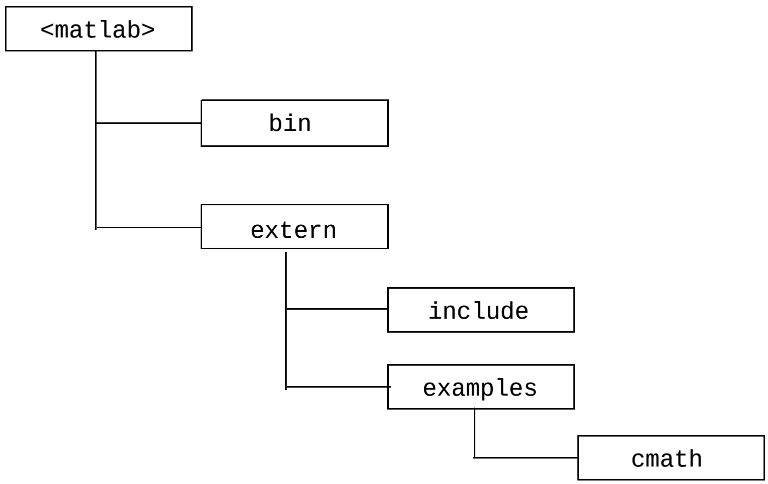
The `<matlab>/extern/examples/cmath` directory contains the sample C programs that are described throughout this book.

`intro.c`	The source code for "Example – Writing a Simple Program" on page 2-2.
`ex1.c`	The source code for "Example Program: Creating Numeric Arrays (ex1.c)" on page 3-14.
`ex2.c`	The source code for "Example – Managing Array Memory (ex2.c)" on page 4-19.
`ex3.c`	The source code for "Example – Calling Library Routines (ex3.c)" on page 6-13.
`ex4.c`	The source code for "Example – Passing Functions As Arguments (ex4.c)" on page 6-22.
`ex5.c`	The source code for "Example – Saving and Loading Data (ex5.c)" on page 7-4.

`ex6.c`	The source code for “Example – Defining Try/Catch Blocks (ex6.c)” on page 8-6.
`release.txt`	Release notes for the current release of the MATLAB C Math Library.

Directory Organization on Microsoft Windows

This figure illustrates the folders that contain the MATLAB C Math Library files. <matlab> symbolizes the top-level folder where MATLAB is installed on your system.



<matlab>\bin

The <matlab>\bin directory contains the Dynamic Link Libraries (DLLs) required by stand-alone C applications and the batch file mbuild, which controls the build and link process for you. <matlab>\bin must be on your path for your applications to run. All DLLs are in WIN32 format.

libmat.dll	MAT-file access routines to support mlfLoad() and mlfSave().
libmatlb.dll	MATLAB Built-In Math Library. Contains stand-alone versions of MATLAB built-in math functions and operators. Required for building stand-alone applications.
libmi.dll	Internal MAT-file access routines.

`libmmfile.dll`	MATLAB M-File Math Library. Contains stand-alone versions of the MATLAB math M-files. Needed for building stand-alone applications that require MATLAB M-file math functions.
`libmx.dll`	MATLAB Array Access and Creation Library. Contains array creation and access routines.
`libut.dll`	MATLAB Utilities Library. Contains the utility routines used by various components.
`mbuild.bat`	Batch file that helps you build and link stand-alone executables.
`compopts.bat`	Default options file for use with `mbuild.bat`. Created by `mbuild –setup`.
Options files for `mbuild.bat`	Switches and settings for C compiler to create stand-alone applications, e.g., `msvccomp.bat` for use with Microsoft Visual C.

<matlab>\extern\include

The `<matlab>\extern\include` directory contains the header files for developing MATLAB C Math Library applications and the `.def` files used by the Microsoft Visual C and Borland compilers. The `lib*.def` files are used by MSVC and the `_lib*.def` files are used by Borland.

`libmatlb.h`	Header file containing the prototypes for the MATLAB Built-In Math Library functions.
`libmmfile.h`	Header file containing the prototypes for the MATLAB M-File Math Library functions.
`matlab.h`	Header file for the MATLAB C Math Library.
`matrix.h`	Header file containing the definition of the `mxArray` type and function prototypes for array access routines.
`libmat.def` `_libmat.def`	Contains names of functions exported from the MAT-file DLL.

`libmatlb.def` `_libmatlb.def`	Contains names of functions exported from the MATLAB C Math Built-In Library DLL.
`libmmfile.def` `_libmmfile.def`	Contains names of functions exported from the MATLAB M-File Math Library DLL.
`libmx.def` `_libmx.def`	Contains names of functions exported from `libmx.dll`.

<matlab>\extern\examples\cmath

The `<matlab>\extern\examples\cmath` directory contains sample C programs developed with the MATLAB C Math Library. You'll find explanations for these examples throughout the book..

`intro.c`	The source code for "Example – Writing a Simple Program" on page 2-2.
`ex1.c`	The source code for "Example Program: Creating Numeric Arrays (ex1.c)" on page 3-14.
`ex2.c`	The source code for "Example – Managing Array Memory (ex2.c)" on page 4-19.
`ex3.c`	The source code for "Example – Calling Library Routines (ex3.c)" on page 6-13.
`ex4.c`	The source code for "Example – Passing Functions As Arguments (ex4.c)" on page 6-22.
`ex5.c`	The source code for "Example – Saving and Loading Data (ex5.c)" on page 7-4.
`ex6.c`	The source code for "Example – Defining Try/Catch Blocks (ex6.c)" on page 8-6.
`release.txt`	Release notes for the current release of the MATLAB C Math Library.

B

Errors and Warnings

Introduction

This section lists a subset of the error messages and warning messages issued by the MATLAB C Math Library. Each type of message is treated in its own section. The numbered items in the list below correspond to the numbered sections of code example:

The messages are listed in alphabetical order within each section. Following each message is a short interpretation of the message and, where applicable, suggested ways to work around the error.

Error Messages

This section lists a subset of the error messages issued by the library. By default, programs written using the library always exit after an error has occurred. For information about handling errors so that you can continue processing after an error occurred, see “Error Handling Overview” in Chapter 8.

`Argument must be a vector`

An input argument that must be either 1–by–N or M–by–1, i.e., either a row or column vector, was an M–by–N matrix where neither M nor N is equal to 1. To correct this, check the documentation for the function that produced the error and fix the incorrect argument.

`Division by zero is not allowed`

The MATLAB C Math Library detected an attempt to divide by zero. This error only occurs on non–IEEE machines (notably DEC VAX machines), which cannot represent infinity. Division by zero on IEEE machines results in a warning rather than an error.

`Empty matrix is not a valid argument`

Some functions, such as `mlfSize`, accept empty matrices as input arguments. Others, such as `mlfEig`, do not. You will see this error message if you call a function that does not accept `NULL` matrices with a `NULL` matrix.

`Floating point overflow`

A computation generated a floating-point number larger than the maximum number representable on the current machine. Check your inputs to see if any are near zero (if dividing) or infinity (if adding or multiplying).

`Initial condition vector is not the right length`

This error is issued only by the `mlfFilter` function. The length of the initial condition vector must be equal to the maximum of the products of the dimensions of the input filter arguments. Let the input filter arguments be given by matrices `B` and `A`, with dimensions bM–by–bN and aM–by–aN respectively. Then the length of the initial condition vector must be equal to the maximum of bM * bN and aM * aN.

`Inner matrix dimensions must agree`

Given two matrices, `A` and `B`, with dimensions aN–by–aM and bN–by–bM, the inner dimensions referred to by this error message are aM and bN. These dimensions must be equal. This error occurs, for example, in matrix multiplication; an N–by–2 matrix can only be multiplied by a scalar or a 2–by–M matrix. Any attempt to multiply it by a matrix with other than two rows will cause this error.

`Log of zero`

Taking the log of zero produces negative infinity. On non–IEEE floating point machines, this is an error, because such machines cannot represent infinity.

`Matrix dimensions must agree`

This error occurs when a function expects two or more matrices to be identical in size and they are not. For example, the inputs to `mlfPlus`, which computes the sums of the elements of two matrices, must be of equal size. To correct this error, make sure the required input matrices are the same size.

`Matrix is singular to working precision`

A matrix is singular if two or more of its columns are not linearly independent. Singular matrices cannot be inverted. This error message indicates that two or more columns of the matrix are linearly dependent to within the floating-point precision of the machine.

`Matrix must be positive definite`

A matrix is positive definite if and only if x'Ax > 0 for all nonzero vectors x. This error message indicates that the input matrix was not positive definite.

`Matrix must be square`

A function expected a square matrix. For example, `mlfQz`, which computes generalized eigenvalues, expects both of its arguments to be square matrices. An M–by–N matrix is square if and only if M and N are equal.

`Maximum variable size allowed by the program is exceeded`

This error occurs when an integer variable is larger than the maximum representable integer on the machine. This error occurs because all matrices contain double precision values, yet some routines require integer values; and the maximum representable double precision value is much larger than the maximum representable integer. Correct this error by checking the documentation for the function that produced it. Make sure that all input arguments that are treated as integers are less than or equal to the maximum legal value for an integer.

`NaN and Inf not allowed`

IEEE `NaN` (Not-A-Number) or `Inf` (Infinity) was passed to a function that cannot handle those values, or resulted unexpectedly from computations internal to a function.

`Not enough input arguments`

A function expected more input arguments than it was passed. For example, most functions will issue this error if they receive zero arguments. The MATLAB C Math Library should never issue this error. Please notify The MathWorks if you see this error message.

`Not enough output arguments`

A function expected more output arguments than were passed to it. Functions in the MATLAB C Math Library will issue this error if any required output arguments are `NULL`. If you see this error under any other conditions, please notify The MathWorks.

`Singularity in ATAN`

A singularity indicates an input for which the output is undefined. ATAN (arc tangent) has singularities on the complex plane, particularly at $z = \pm 1$.

`Singularity in TAN`

A singularity indicates an input for which the output is undefined. TAN (tangent function) has singularities at odd multiples of $\pi/2$.

`Solution will not converge`

This error occurs when the input to a function is poorly conditioned or otherwise beyond the capabilities of our iterative algorithms to solve.

`String argument is an unknown option`

A function received a string matrix (i.e., a matrix with the string property set to true) when it was not expecting one. For example, most of the matrix creation functions, for example, `mlfEye` and `mlfZeros`, issue this error if any of their arguments are string matrices.

`The only matrix norms available are 1, 2, inf and fro`

The function `mlfNorm` has an optional second argument. This argument must be either the scalars 1 or 2 or the strings `inf` or `fro`. `inf` indicates the infinity norm and `fro` the F–norm. This error occurs when the second argument to `mlfNorm` is any value other than one of these four values.

`Too many input arguments`

This error occurs when a function has more input arguments passed to it than it expects. The MATLAB C Math Library should never issue this error, as this condition should be detected by the C compiler. Please notify The MathWorks if you see this error.

`Too many output arguments`

This error occurs when a function has more output arguments passed to it than it expects. The MATLAB C Math Library should never issue this error, as this condition should be detected by the C compiler. Please notify The MathWorks if you see this error.

`Variable must contain a string`

An argument to a function should have been a string matrix (i.e., a matrix with the string property set to true), but was not.

`Zero can't be raised to a negative power`

On machines with non–IEEE floating point format, the library does not permit you to raise zero to any negative power, as this would result in a division by zero, since `x^(-y) == 1/(x^y)` and `0^n == 0`. Non–IEEE machines cannot represent infinity, so division by zero is an error on those machines (mostly DEC VAXes).

Warning Messages

All warnings begin with the string `Warning:`. By default, programs written using the library output a message after a warning-level event has occurred and then continue processing.

For most warning messages there is a corresponding error message. Generally, warning messages are issued in place of errors on IEEE–floating point compliant machines when an arithmetic expression results in plus or minus infinity or a nonrepresentable number. Where this is the case, the error message explanation has not been reproduced. See the section “Error Messages” for an explanation of these messages.

```
Warning: Divide by zero
Warning: Log of zero
Warning: Matrix is close to singular or badly scaled. Results may
be inaccurate
Warning: Matrix is singular to working precision
Warning: Singularity in ATAN
Warning: Singularity in TAN
```

Index

Symbols

A

D

O