MATLAB®

C Math Library

Computation

Visualization

Programming

User's Guide

Version 1.2

How to Contact The MathWorks:

508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 24 Prime Park Way Natick, MA 01760-1500	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

Contents

1 Getting Ready

2 Writing Programs

3 Using the Library

4 Library Routines

5 Directory Organization

A Errors and Warnings

1

Getting Ready

Introduction

The MATLAB® C Math Library makes the mathematical core of MATLAB available to application programmers. The library is a collection of approximately 350 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 a feature of the MATLAB language. However, you do not need to know MATLAB or own a copy of MATLAB in order 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.

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.

While the library provides a great many functions, it does not contain all of MATLAB. The MATLAB C Math Library consists of mathematical functions only. It does not contain any Handle Graphics® or Simulink® functions. Nor does it contain those functions that require the MATLAB interpreter, most notably `eval()` and `input()`. In addition, multidimensional arrays, cell arrays, structures, and objects are not currently supported by the library. Finally, the MATLAB C Math Library cannot create or manipulate sparse matrices.

NOTE: Version 1.2 of the MATLAB C Math Library is a compatibility release that brings the MATLAB C Math Library into compliance with MATLAB 5. Although the MATLAB C Math Library is compatible with MATLAB 5, it does not support many of its new features.

Library Basics

When you're using the MATLAB C Math Library, remember these important features:

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

  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 functions operate on arrays. Arrays in the MATLAB C Math Library are represented by the `mxArray` data type.

  `mxArray` is an opaque data type. You must use functions to access its fields. The routines that you use to access and manipulate the fields of an `mxArray` begin with the `mx` prefix and belong to the Application Program Interface Library. See the online Help Desk for documentation on the Application Program Interface Library.

- The MATLAB C Math Library does not manage memory for you.

  Arrays returned by the MATLAB C Math Library are dynamically allocated. You are responsible for freeing all returned arrays once you are done using them. If you do not free these arrays using the routine `mxDestroyArray()`, your program will leak memory; if your program runs long enough, or uses large enough arrays, these memory leaks will eventually cause your program to run out of memory.

How This Book Is Organized

This book serves as both a tutorial and a reference. It is divided into five chapters and an appendix.

- **Chapter 1: Getting Ready**. The introduction, installation instructions, and build information.
- **Chapter 2: Writing Programs**. Examples that demonstrate how to accomplish several basic tasks with the MATLAB C Math Library.
- **Chapter 3: Using the Library**. The most technical chapter that explains in detail how to use the library.
- **Chapter 4: Library Routines**. The functions available in the MATLAB C Math Library. The chapter groups the more than 350 library functions into functional categories and provides a short description of each function.
- **Chapter 5: Directory Organization**. A description of the MATLAB directory structure that positions the library's files in relation to other products from The MathWorks.
- **Appendix A: Errors and Warnings**. A reference to the error messages issued by the library.

Documentation Set

The complete documentation set for the MATLAB C Math Library consists of printed and online publications. The online reference documents the C Math Library functions themselves.

Primary Sources of Information

- This book, the *MATLAB C Math Library User's Guide*
- The online *MATLAB C Math Library Reference*
- An online PDF version of the *MATLAB C Math Library User's Guide*
- An online PDF version of the *MATLAB C Math Library Reference*

Using the Online References

To look up the syntax and behavior for each of the C Math Library functions, refer to the online *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 online *MATLAB Function Reference* page for the corresponding MATLAB function.

If you are a MATLAB user:

1 Type `helpdesk` at the MATLAB prompt.

2 From the MATLAB Help Desk, select *C Math Library Reference* from the **Other Products** section.

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

- Online *MATLAB Application Program Interface Reference*
- Online *MATLAB Application Program Interface Guide*
- Online *MATLAB Function Reference*
- *Installation Guide for UNIX*
- *Installation Guide for PC and Macintosh*
- Release notes for the MATLAB C Math Library

Installing the C Math Library

The MATLAB C Math Library is available on UNIX workstations, IBM PC compatible computers running Microsoft Windows (Windows 95 and Windows NT), and Apple Macintosh computers. 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 C Math Library. Follow the instructions in the MATLAB *Installation Guide* for your specific platform:

- *Installation Guide for UNIX*
- *Installation Guide for PC and Macintosh*

The C Math Library will appear as one of the installation choices that you can select as you proceed through the installation screens.

Before you can install the C Math Library, you will require an appropriate `FEATURE` line in your License File (UNIX or networked PC users) or an appropriate Personal License Password (non-networked PC or Macintosh users). If you do not yet have the required `FEATURE` line or Personal License Password, contact The MathWorks immediately:

- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000, ask for Customer Service
- Via fax at 508-647-7001

MATLAB Access members can obtain the necessary license data via the Web (`www.mathworks.com`). Click on the MATLAB Access icon and log in to the Access home page. MATLAB Access membership is free of charge.

Installation Without MATLAB

The process for installing the 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:

- *Installation Guide for UNIX*
- *Installation Guide for PC and Macintosh*

Before you begin installing the C Math Library, you must obtain from The MathWorks a valid License File (UNIX or networked PC users) or Personal License Password (non-networked PC or Macintosh users). These are usually supplied by fax or e-mail. If you have not already received a License File or Personal License Password, contact The MathWorks by any of these methods:

- Via e-mail at `service@mathworks.com`
- Via telephone at 508-647-7000; ask for Customer Service
- Via fax at 508-647-7001

MATLAB Access members can obtain the necessary license data via the Web (`www.mathworks.com`). Click on the MATLAB Access icon and log in to the Access home page. MATLAB Access membership is free of charge.

Workstation Installation Details

To verify that the MATLAB C Math Library has been installed correctly, use the `mbuild` script, which is documented in “Building on UNIX” on page 1-11, to verify that you can build one of the example applications. Be sure to use `mbuild` before calling Technical Support.

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

PC Installation Details

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, use the mbuild script, which is documented in "Building on Microsoft Windows" on page 1-17, to verify that you can build one of the example applications. Be sure to use mbuild before calling Technical Support.

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

Macintosh Installation Details

To verify that the MATLAB C Math Library has been installed correctly, use the mbuild script, which is documented in "Building on Macintosh" on page 1-23, to verify that you can build one of the example applications. Be sure to use mbuild before calling Technical Support.

Power Macintosh. To spot check that the installation worked, look for the file matlab.h in <matlab>:extern:include and the files libmatlb, libmmfile and libmcc in <matlab>:extern:lib:PowerMac where <matlab> symbolizes the MATLAB root directory.

On a Power Macintosh, the installation script adds an alias of the <matlab>:extern:lib:PowerMac: folder to the System Folder:Extensions: folder.

68K Macintosh. The MATLAB C Math Library consists of three static libraries on Macintoshes with the 68K series microprocessor.

To spot check that the installation worked, check for the file matlab.h in <matlab>:extern:include and the libraries libmatlb.o, libmmfile.o, and libmcc.o in <matlab>:extern:lib:68k:MPW where <matlab> symbolizes the MATLAB root directory.

Building C Applications

This section explains how to build stand-alone C applications on UNIX, Microsoft Windows, and Macintosh systems.

The section begins with a summary of the steps involved in building C applications with the `mbuild` script and then describes platform-specific issues for each supported platform. `mbuild` helps automate the build process.

You can use the `mbuild` script to build the examples presented in Chapter 2 and to build your own stand-alone C applications. You'll find the source for the examples in the `<matlab>/extern/examples/cmath` subdirectory; `<matlab>` represents the top-level directory where MATLAB is installed on your system. See the "Directory Organization" chapter for the location of other C Math Library files.

Overview

On all three operating systems, you must follow three steps to build C applications with `mbuild`:

1. Configure `mbuild` to create stand-alone applications.
2. Verify that `mbuild` can create stand-alone applications.
3. Build your application.

Once you have properly configured `mbuild`, you simply repeat step 3 to build your applications. You only need to go back to steps 1 and 2 if you change compilers, for example, from Watcom to MSVC, or upgrade your current compiler.

Figure 1-1 shows the configuration and verification steps on all platforms. The sections following the flowchart provide more specific details for the individual platforms.

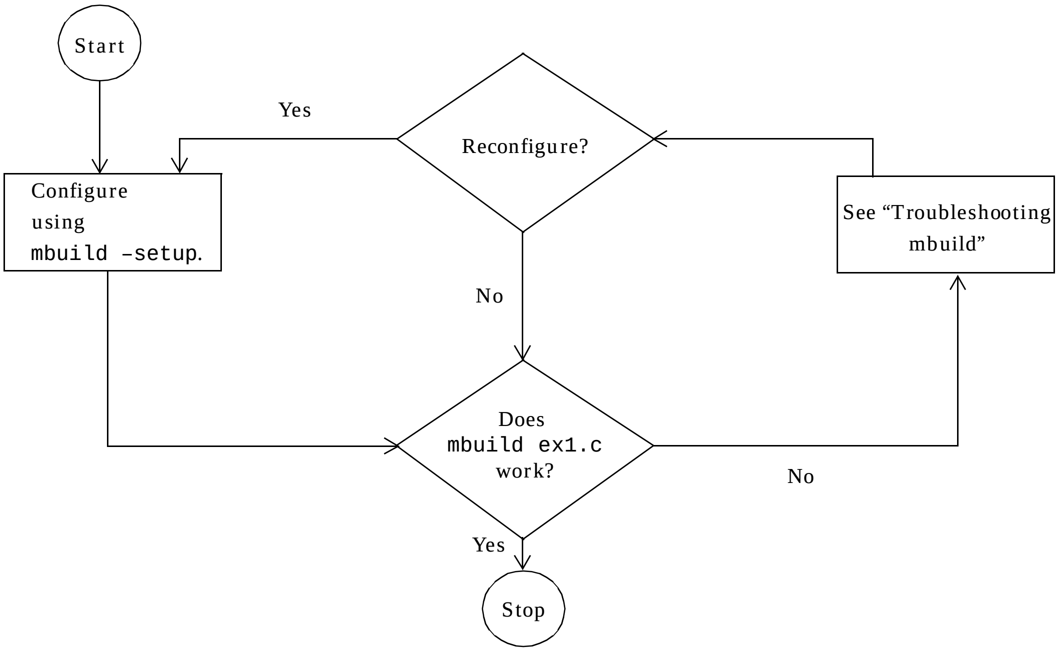


Figure 1-1: Sequence for Creating Stand-Alone C Applications

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 against. The necessary shared libraries vary by platform and are listed within the individual UNIX, Windows, and Macintosh sections that follow.

Getting Started

In order 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 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

`mbuild` stores your compiler and linker settings in an "options file." Before you can use `mbuild` to create an application, you must first configure it for your system. The configuration process is slightly different for each type of system.

Building on UNIX

This section explains how to compile and link C source code into a stand-alone UNIX application.

Configuring mbuild

To configure `mbuild`, at the UNIX prompt type:

```
mbuild –setup
```

The `setup` switch creates a user-specific options file for your ANSI C compiler.

NOTE: The default C compiler that comes with many Sun workstations is not an ANSI C compiler.

Executing `mbuild –setup` presents a list of options files.

```
mbuild –setup

Using the 'mbuild –setup' command selects an options file that is
placed in ~/matlab and used by default for 'mbuild' when no other
options file is specified on the command line.

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

Enter the number of the options file to use as your default
options file:
```

To select the proper options file for creating a stand-alone C application, enter `2` and press **Return**. If an options file doesn't exist in your MATLAB directory, the system displays a message stating that the options file is being copied to your MATLAB directory. If an options file already exists in your MATLAB directory, the system prompts you to overwrite it.

NOTE: The options file is stored in the MATLAB subdirectory of your home directory. This allows each user to have a separate `mbuild` configuration.

Changing Compilers. If you switch between C and C++, use the `mbuild –setup` command and make the desired changes. If you want to change to a different ANSI C compiler, you must edit `mbuildopts.sh`.

Verifying mbuild

The C source code for 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.

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`, `lnx86`, `sgi`, `sgi64`, `sol2`, or `sun4`)

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.

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
```

The mbuild Script

The `mbuild` script supports various switches that allow you to customize the building and linking of your code. All users must execute `mbuild -setup` at least once. During subsequent `mbuild`s, the other switches are optional. The `mbuild` syntax and options are:

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

Table 1-1: 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 <file>	Use <file> as the options file; <file> is a full path name if it is not in current directory. (Not necessary if you use the –setup option, but useful to override the default.)
-F <file>	Use <file> as the options file. (Not necessary if you use the –setup option.) <file> is searched for in the following manner: The file that occurs first in this list is used: • ./<filename> • $HOME/matlab/<filename> • $TMW_ROOT/bin/<filename>
–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.
–l<file>	Link against library lib<file>.
–L<pathname>	Include <pathname> in the list of directories to search for libraries.
<name>=<def>	Override options file setting for variable <name>.

Table 1-1: mbuild Options on UNIX (Continued)

Option	Description
`-n`	No execute flag. Using this option causes the commands that would be used to compile and link the target to be displayed without executing them.
`-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 switch should be the only argument passed.
`-U<name>`	Undefine C preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.

Customizing mbuild

If you need to change the switches that `mbuild` passes to your compiler or linker, use the verbose switch, `-v`, as in:

```
mbuild -v filename.c [filename1.c filename2.c …]
```

to generate a list of all the current compiler settings. If you need to change settings, use an editor to make changes to your options file, which is in your local MATLAB directory, typically `~/matlab`. You can also embed the settings obtained from the verbose switch into an integrated development environment (IDE) or makefile. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

`mbuild -setup` copies a master options file to your local MATLAB directory and then edits the local file. If you want to make your edits persist through repeated uses of `mbuild -setup`, you must edit the master file itself: `<matlab>/bin/mbuildopts.sh`.

NOTE: Any changes that you make to the local options file will be overwritten the next time you execute `mbuild -setup`.

Distributing Stand-Alone UNIX Applications

To distribute a stand-alone application, you must include the application's executable and the shared libraries against which the application was linked. This package of files includes:

- Application (executable)
- `libmmfile.`*ext*
- `libmatlb.`*ext*
- `libmcc.`*ext*
- `libmat.`*ext*
- `libmx.`*ext*
- `libut.`*ext*

where .*ext* is

`.a` on IBM RS/6000 and Sun4; `.so` on Solaris, Alpha, Linux, and SGI; and `.sl` on HP 700.

For example, to distribute the `ex1` example for Solaris, you need to include `ex1`, `libmmfile.so`, `libmatlb.so`, `libmcc.so`, `libmat.so`, `libmx.so`, and `libut.so`. The path variable must reference the location of the shared libraries.

Building on Microsoft Windows

This section explains how to compile and link C code into stand-alone Windows applications.

Configuring mbuild

To configure `mbuild`, at the DOS command prompt type:

```
mbuild –setup
```

The `setup` switch creates an options file for your ANSI C compiler.

You *must* run `mbuild –setup` before you create your first stand-alone application; otherwise, when you try to create an application with `mbuild`, you will get the message

```
Sorry! No options file was found for mbuild. The mbuild script
must be able to find an options file to define compiler flags and
```

```
other settings.  The default options file is
$script_directory\\$OPTFILE_NAME.

To fix this problem, run the following:

mbuild –setup

This will configure the location of your compiler.
```

Running mbuild with the setup option presents you with a list of questions. You will be asked to specify which library to link against and which compiler to use. Do not select the MATLAB C++ Math Library unless you have purchased it.

This example shows how to select the Microsoft Visual C/C++ compiler:

```
mbuild –setup
Welcome to the utility for setting up compilers
for building math library applications files.

Choose your default Math Library:
[1] MATLAB C Math Library
[2] MATLAB C++ Math Library

Math Library: 1

Choose your C/C++ compiler:
[1] Borland C/C++        (version 5.0 or version 5.2)
[2] Microsoft Visual C++ (version 4.2 or version 5.0)
[3] Watcom C/C++         (version 10.6 or version 11)

[0] None

compiler: 2
```

If we support more than one version of the compiler, you are asked for a specific version. For example,

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

version: 2
```

Next, you are asked to enter the root directory of your ANSI compiler installation:

```
Please enter the location of your C/C++ compiler: [c:\msdev]
```

Finally, you must verify that the information is correct:

```
Please verify your choices:

Compiler: Microsoft Visual C++ 5.0
Location: c:\msdev
Library: C math library

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

Default options file is being updated...
```

If you respond to the verification question with `n` (no), you get a message stating that no compiler was set during the process. Simply run `mbuild –setup` once again and enter the correct responses for your system.

Changing Compilers. If you want to change your ANSI (system) compiler, make other changes to its options file (e.g., change the compiler's root directory), or switch between C and C++, use the `mbuild –setup` command and make the desired changes.

Verifying mbuild

C source code for 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.

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
```

The mbuild Script

The `mbuild` script supports various switches that allow you to customize the building and linking of your code. All users must execute `mbuild –setup` at least once. During subsequent `mbuild`s, the other switches are optional. The `mbuild` syntax and options are:

mbuild [–*options*] [*filename1 filename2* …]

Table 1-2: mbuild Options on Microsoft Windows

Option	Description
`–c`	Compile only; do not link.
`–D<name>`	Define C preprocessor macro `<name>`.
`–f <file>`	Use `<file>` as the options file; `<file>` is a full pathname if it is not in the current directory. (Not necessary if you use the `–setup` option.)
`–g`	Build an executable with debugging symbols included.
`–h[elp]`	Help; prints a description of `mbuild` and the list of options.

Table 1-2: mbuild Options on Microsoft Windows (Continued)

Option	Description
`-I<pathname>`	Include `<pathname>` in the list of directories to search for header files.
`-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 switch should be the only argument passed.
`-U<name>`	Undefine C preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.

Customizing mbuild

If you need to change the switches that `mbuild` passes to your compiler or linker, use the verbose switch, `-v`, as in:

```
mbuild -v filename.c [filename1.c filename2.c …]
```

to generate a list of all the current compiler settings. If you need to change the settings, use an editor to make changes to the options file that corresponds to your compiler. The local options file is called `compopts.bat`. It is located in the `<matlab>\bin` directory. You can also embed the settings obtained from the verbose switch into an integrated development environment (IDE) or makefile. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

`mbuild -setup` copies a master options file to a current options file and then edits the current options file. If you want to make your edits persist through repeated uses of `mbuild -setup`, you must edit the master file itself. The current and master options files are in the same directory, typically `matlab\bin`.

Compiler	Master Options File
Borland C, Version 5.0 or 5.2	`bcccomp.bat`
Microsoft Visual C, Version 4.2	`msvccomp.bat`
Microsoft Visual C, Version 5.0	`msvc50comp.bat`
Watcom C, Version 10.6	`watccomp.bat`
Watcom C, Version 11	`wat11ccomp.bat`

NOTE: Any changes that you make to the current options file will be overwritten the next time you execute `mbuild –setup`.

Shared Libraries (DLLs)

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

```
<matlab>\bin
```

The relevant libraries for building stand-alone applications are WIN32 DLLs. Before running a stand-alone application, you must ensure that the directory containing the DLLs is on your path.

The `.def` files for the Microsoft and Borland compilers are in the `<matlab>\extern\include` directory; `mbuild` dynamically generates import libraries from the `.def` files.

Distributing Stand-Alone Microsoft Windows Applications

To distribute a stand-alone application, you must include the application's executable as well as the shared libraries against which the application was linked. This package of files includes:

- Application (executable)
- `libmmfile.dll`
- `libmatlb.dll`
- `libmcc.dll`
- `libmat.dll`
- `libmx.dll`
- `libut.dll`

For example, to distribute the Windows version of the `ex1` example, you need to include `ex1.exe`, `libmmfile.dll`, `libmatlb.dll`, `libmcc.dll`, `libmat.dll`, `libmx.dll`, and `libut.dll`.

The DLLs must be on the system path. You must either install them in a directory that is already on the path or modify the PATH variable to include the new directory.

Building on Macintosh

This section explains how to compile and link C code into a stand-alone Macintosh application.

NOTE: CodeWarrior users who do not have MATLAB installed on their systems *cannot* use `mbuild`. You should look at the sample projects included in the `matlab:extern:examples:cmath:codewarrior` folder, view the settings, make modifications if necessary, and apply them to your own projects.

Configuring mbuild

To configure `mbuild`, use

```
mbuild -setup
```

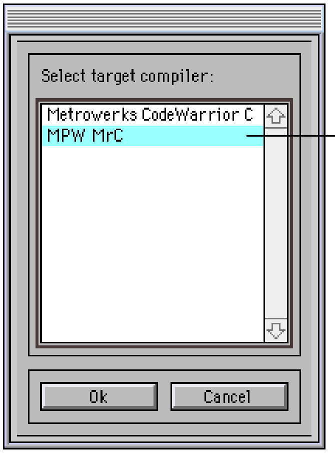
NOTE: You must run `mbuild -setup` before you create your first stand-alone application; otherwise, when you try to create a stand-alone application, you will get the error

```
An mbuildopts file was not found or specified.  Use
"mbuild -setup" to configure mbuild for your compiler.
```

Run the `setup` option from the MATLAB prompt if you are a MATLAB user or the MPW shell prompt.

```
mbuild -setup
```

Executing `mbuild` with the `setup` option displays a dialog with a list of compilers whose options files are currently included in the `<matlab>:extern:scripts:` folder. This figure shows MPW MrC selected as the desired compiler.



Selected Compiler

Click **Ok** to select the compiler. If you previously selected an options file, you are asked if you want to overwrite it. If you do not have an options file in your `<matlab>:extern:scripts:` folder, `mbuild –setup` creates the appropriate options file for you.

NOTE: If you select MPW, `mbuild –setup` asks you if you want to create `UserStartup•MATLAB_MEX` and `UserStartupTS•MATLAB_MEX`, which configures MPW and ToolServer for building stand-alone applications.

When this message displays, `mbuild` is configured properly:

```
MBUILD –setup complete.
```

Changing Compilers. If you want to change your current compiler, use the `mbuild –setup` command.

Verifying mbuild

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

```
mbuild ex1.c
```

This should create the file called `ex1`, a stand-alone application. You can now run your stand-alone application by double-clicking its icon. The results should be:

```
     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
```

The mbuild Script

The `mbuild` script supports various switches that allow you to customize the building and linking of your code. All users must execute `mbuild -setup` at least once. During subsequent `mbuild`s, the other switches are optional. The `mbuild` syntax and options are:

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

Table 1-3: mbuild Options on Macintosh

Option	Description
`-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. (Not necessary if you use the `-setup` option.) If `<file>` is specified, it is used as the options file. If `<file>` is not specified and there is a file called `mbuildopts` in the current directory, it is used as the options file. If `<file>` is not specified and `mbuildopts` is not in the current directory and there is a file called `mbuildopts` in the directory `<matlab>:extern:scripts:`, it is used as the options file. Otherwise, an error occurs.
`-g`	Build an executable with debugging symbols included.
`-h[elp]`	Help; prints a description of `mbuild` and the list of options.

Table 1-3: mbuild Options on Macintosh (Continued)

Option	Description
`-I<pathname>`	Include `<pathname>` in the list of directories to search for header files.
`<name>=<def>`	Override options file setting for variable `<name>`.
`-output <name>`	Create an executable named `<name>`.
`-O`	Build an optimized executable.
`-setup`	Set up the default compiler and libraries. This switch should be the only argument passed.
`-v`	Verbose; print all compiler and linker settings.

Customizing mbuild

If you need to change the switches that `mbuild` passes to your compiler or linker, use the verbose switch, `-v`, as in:

```
mbuild -v filename.c [filename1.c filename2.c …]
```

to generate a list of all the current compiler settings. If you need to change the switches, use an editor to make changes to your options file, `mbuildopts`. You can also embed the settings obtained from the verbose switch into an integrated development environment (IDE) or makefile. Often, however, it is easier to call `mbuild` from your makefile. See your system documentation for information on writing makefiles.

`mbuild –setup` copies a master options file to a current options file and then edits the current options file. If you want to make your edits persist through repeated uses of `mbuild –setup`, you must edit the master file itself.

Compiler	Master Options File
CodeWarrior v10 and v11	`mbuildopts.CW`
CodeWarrior PRO v1 (Power Macintosh only)	`mbuildopts.CWPRO`
MPW ETO 21, 22, and 23 (Power Macintosh only)	`mbuildopts.MPWC`

Distributing Stand-Alone Macintosh Applications

To distribute a stand-alone application, you must include the application's executable and the shared libraries against which the application was linked. These lists show which files should be included on the Power Macintosh and 68K Macintosh systems:

Power Macintosh.

- Application (executable)
- `libmmfile`
- `libmatlb`
- `libmcc`
- `libmat`
- `libmx`
- `libut`

68K Macintosh.

- Application (executable)

For example, to distribute the Power Macintosh version of the `ex1` example, you need to include `ex1`, `libmmfile`, `libmatlb`, `libmcc`, `libmx`, and `libut`. To distribute the 68K Macintosh version of the `ex1` example, you only need to include the application, `ex1`, since 68K libraries are static.

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, the process will terminate and you will not be able to use mbuild to create your applications.

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 Microsoft Windows, if you get errors such as unrecognized command 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.

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 508 647-7200.

Building on Your Own

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 Chapter 5 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 Compiler Library (`libmcc`)
3. MATLAB Built-In Library (`libmatlb`)
4. MATLAB MAT-file Library (`libmat`)
5. MATLAB Application Program Interface Library (`libmx`)
6. ANSI C Math Library (`libm`)

Specifying the libraries in the wrong order on the command line typically causes linker errors. Note that on the PC if you are using the Microsoft Visual C compiler, you must manually build the import libraries from the `.def` files. If you are using the Borland C Compiler, you can link directly against the `.def` files. If you are using Watcom, you must build them from the DLLs.

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 the platforms we support.

2

Writing Programs

The best way to learn how to use the library is to see it in use. This chapter contains six examples. The first five are straightforward, each illustrating a particular aspect of the MATLAB C Math Library. The example, “Passing Functions as Arguments,” is longer and more complex, more like a real application.

The subjects of the six examples are:

- Creating and Printing Arrays
- Writing Simple Functions
- Calling Library Routines
- Handling Errors
- Saving and Loading Data
- Passing Functions as Arguments

Each example presents a complete working program. The numbers to the left of code statements reference annotations presented in a “Notes” section that immediately follows each example. An “Output” section that shows the output produced by the example is presented next. You can find the code for each example in the `<matlab>/extern/examples/cmath` directory where `<matlab>` represents the top-level directory of your installation. See “Building C Applications” in Chapter 1 for information on building the examples.

Introduction

In this book, the examples are presented before the technical details of the library. Hopefully, you will find this organization convenient. However, before exploring the examples, you need to know a little more about how the MATLAB C Math Library works. The next two sections explain the array access functions and the physical memory layout of an array. Macintosh programmers should also read the section Macintosh Print Handlers.

Array Access Functions

Some of the functions used in the examples do not begin with the prefix `mlf`; they begin with `mx` instead. The `mx` functions are the array creation, deletion, and access functions that are part of the MATLAB Application Program Interface Library. For example, the examples demonstrate how to use the function `mxCreateDoubleMatrix()` to create a matrix that stores double values and the function `mxDestroyArray()` to free an array.

You use these functions when you work with arrays. Just like the mathematical routines in the MATLAB C Math Library, these functions most often require `mxArray *` arguments and return a pointer to an `mxArray`. Refer to the section “Array Access Functions” in Chapter 5 for a complete list of the functions and to the online *Application Program Interface Reference* for details on their behavior and arguments.

Array Storage: MATLAB vs. C

In reading the example code, it is important to note 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 because the mapping 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.

In order to construct our first 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 array, when used to initialize a MATLAB array, produces the desired result.

Macintosh Print Handlers

If you are using the MATLAB C Math Library on an Apple Macintosh computer and using `mbuild` or the example projects provided in the `<matlab>:extern:examples:cmath:codewarrior:` directory to build the examples, you may skip this section. However, if you are using a different method to build the examples, this section describes how to ensure that the output from the examples displays properly.

The MATLAB C Math Library uses `printf`, by default, to display its output. Macintosh computers, unlike UNIX workstations or machines running Microsoft Windows, do not have command-line shells. This means that all Macintosh programs must use some type of window or dialog box to display output. Each Macintosh compiler handles the output from the `printf` function in a different, nonstandard, way.

Because the MATLAB C Math Library supports more than one compiler on the Macintosh, there is no one appropriate choice for the default print handler. If you want to see output from the examples, you must install a print handler. You have two choices. You may either write and install a print handler (quite a simple task, actually), or you may use the slightly riskier method of using `printf` as your print handler.

If you want to install a print handler, read "Apple Macintosh Example" on page 3-41. If you'd like to use `printf`, add the following line of code to each example, just after the variable declarations within the `main()` routine.

```
mlfSetPrintHandler((void (*)(const char *))printf);
```

This approach is only safe if your compiler returns values in the registers rather than on the stack. It is known to work with both the Metrowerks and MPW compilers; try it at your own risk on other compilers.

Explaining in detail why installing a default print handler is necessary is beyond the scope of this document. Basically, you can't use the default print handler because the simple input/output library can't intercept the call to `CoWrite` in the MATLAB Built-in Library because that library is shipped as a shared library.

Example 1: Creating and Printing Arrays

This program creates two arrays and then prints them. The primary purpose of this example is to present a simple yet complete program. The code, therefore, demonstrates only one of the ways to create an array. Each of the numbered sections of code is explained in more detail below.

```
    /* 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 };

    main()
    {
        /* Create two matrices */
3       mxArray *mat0 = mxCreateDoubleMatrix(2, 3, mxREAL);
        mxArray *mat1 = mxCreateDoubleMatrix(3, 2, mxCOMPLEX);

4       memcpy(mxGetPr(mat0), real_data, 6 * sizeof(double));
        memcpy(mxGetPr(mat1), real_data, 6 * sizeof(double));
        memcpy(mxGetPi(mat1), cplx_data, 6 * sizeof(double));

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

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

        return(EXIT_SUCCESS);
    }
```

Notes

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 mxCreateDoubleMatrix(). mxCreateDoubleMatrix() takes three arguments: the number of rows, the number of columns, and a predefined constant indicating whether the matrix is complex (has an imaginary part) or real. It returns a full matrix: a matrix for which all elements in the matrix are allocated physical storage. This is in contrast to a sparse matrix in which only the nonzero elements have storage allocated. (Note that the library does not support sparse matrices at this time.)

   mxCreateDoubleMatrix() allocates an mxArray structure and storage space for the elements of the matrix, initializing each entry in the matrix to zero. The first matrix, mat0, does not have an imaginary part, therefore its complex flag is mxREAL. The second matrix, mat1, has an imaginary part, so its complex flag is mxCOMPLEX. mat0 has two rows and three columns, and mat1 has three rows and two columns.

4 Copy the data in the static array into the matrices. Using memcpy in this way is the standard programming idiom for initializing a matrix from user-defined data. You will see similar code throughout the examples. Both matrices have six elements in their real parts. mat1 has six elements in its imaginary part. Note that if an array has both a real and complex part, both parts must be the same size.

5 Print the matrices. mlfPrintMatrix() calls the installed print handler, which in this example is the default print handler. See the section “Print

Handlers” in Chapter 3 for details on modifying and installing a print handler.

6 Free the matrices. All matrices returned by MATLAB C Math Library routines must be manually freed. The library does not maintain a list of allocated matrices or perform any garbage collection. If you do not free your matrices after you are finished using them, your program will leak memory. If your matrices are large enough, or your program runs long enough, the program will eventually run out of memory.

Output

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
```

Example 2: Writing Simple Functions

This example demonstrates how to write a simple function that takes two `mxArray*` arguments and returns an `mxArray*` value. The function computes the average of the two input matrices. Each of the numbered sections of code is explained in more detail below.

```
    /* ex2.c*/

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

2   static double data0[] = { 2, 6, 4, 8 };
    static double data1[] = { 1, 5, 3, 7 };

    /* Calculates (m1 + m2) / 2 */
3   mxArray *average(mxArray *m1, mxArray *m2)
    {
        mxArray *sum, *ave, *two = mlfScalar(2);

4       sum = mlfPlus(m1, m2);
        ave = mlfRdivide(sum, two);
        mxDestroyArray(sum);
        mxDestroyArray(two);
        return ave;
    }

    main()
    {
        /* Create two matrices */
5       mxArray *mat0 = mxCreateDoubleMatrix(2, 2, mxREAL);
        mxArray *mat1 = mxCreateDoubleMatrix(2, 2, mxCOMPLEX);
        mxArray *mat2;

        memcpy(mxGetPr(mat0), data0, 4 * sizeof(double));
        memcpy(mxGetPr(mat1), data1, 4 * sizeof(double));
```

```
⑥      mat2 = average(mat0, mat1);

⑦      mlfPrintMatrix(mat0);
       mlfPrintf(" + \n");

       mlfPrintMatrix(mat1);
       mlfPrintf(" / 2 = \n");

       mlfPrintMatrix(mat2);

⑧      mxDestroyArray(mat0);
       mxDestroyArray(mat1);
       mxDestroyArray(mat2);

       return(EXIT_SUCCESS);
   }
```

Notes

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 four-element arrays that are used subsequently to initialize the 2-by-2 matrices. The numbers in these arrays are placed in column-major order, as that is how the MATLAB C Math Library will interpret them.

3. Declare the function `average`. This function takes two `mxArray*` arguments, adds the matrices together, and divides the result by two. It computes the element-wise average of the two matrices. Note that the divisor, `two`, must be a matrix as well; `average` creates it using `mlfScalar`, a MATLAB C Math Library utility function.

4. Add, with `mlfPlus`, the two input matrices together, and return the result in a newly allocated matrix. Then divide, with `mlfRdivide`, that sum by a scalar (1-by-1) matrix containing the number two. The `mlfRdivide`

operation (array right division) returns the average of the two matrices. `average` will return this value.

Before returning from `average`, however, free both the intermediate `sum` matrix (the result of the call to `mlfPlus`) and the scalar matrix `two`. If these matrices are not freed, the function will leak memory.

5 Create the initial 2-by-2 matrices. Note the use of `memcpy` to initialize the newly allocated matrices with data. This is the same programming idiom used in the first example.

6 Calculate the average of the two matrices.

7 Print the results. Both `mlfPrintMatrix()` and `mlfPrintf()` use the installed print handling routine to display their output. Because this example does not register a print handling routine, the default print handler displays all output. The default print handler uses `printf`. See the section “Print Handlers” in Chapter 3 for details on registering print handlers.

8 Finally, free the initial input matrices and the result of the average function.

Output

When the program runs, it produces this output:

```
     2     4
     6     8

+
     1     3
     5     7

/ 2 =
     1.5000    3.5000
     5.5000    7.5000
```

Example 3: Calling Library Routines

This example uses the singular value decomposition function `mlfSvd` to illustrate how to call library routines that take multiple optional arguments. The example demonstrates the subtleties of the MATLAB C Math Library calling convention that the calls to `mlfRdivide` and `mlfPlus` in the previous example did not demonstrate.

```
    /* 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()
    {
        /* Create the input matrix */
3       mxArray *X = mxCreateDoubleMatrix(4, 2, mxREAL);
        mxArray *U, *S, *V, *Zero = mlfScalar(0.0);

        memcpy(mxGetPr(X), data, 8 * sizeof(double));

        /* Compute the singular value decomposition and print it */
4       U = mlfSvd(NULL, NULL, X, NULL);
        mlfPrintf("One input, one output:\nU = \n");
        mlfPrintMatrix(U);
        mxDestroyArray(U);

        /* Multiple output arguments */
5       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);
        mxDestroyArray(U);
        mxDestroyArray(S);
        mxDestroyArray(V);
```

```
        /* Multiple input and output arguments */
(6)     U = mlfSvd(&S, &V, X, Zero);
        mlfPrintf("Two inputs, three outputs:\n");
        mlfPrintf("U = \n"); mlfPrintMatrix(U);
        mlfPrintf("S = \n"); mlfPrintMatrix(S);
        mlfPrintf("V = \n"); mlfPrintMatrix(V);
        mxDestroyArray(U);
        mxDestroyArray(S);
        mxDestroyArray(V);

(7)     mxDestroyArray(X);
        mxDestroyArray(Zero);

        return(EXIT_SUCCESS);
    }
```

Notes

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 Create and initialize the mlfSvd input arrays, X and Zero. Zero is a 1-by-1 array created with mlfScalar(). Declare 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` and then free the result matrix. Freeing return values is essential to avoid memory leaks.

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 and then free all of the output matrices.

6 Call `mlfSvd` the third way, with three output arguments and two input arguments. Print and then free 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 the two input matrices.

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
```

Example 4: Handling Errors

The MATLAB C Math Library's default response to an error is to call `exit()`, which terminates an application. In some cases, program termination may be unacceptable. For this reason, the library provides an Application Programming Interface (API) to control the error handling mechanism.

This example demonstrates a user-defined error handler and the use of two C system calls, `setjmp()` and `longjmp()`. Together they provide a more flexible response to an error than the default library response. This example only provides a brief description of how `setjmp()` and `longjmp()` work. For more details, consult your system's documentation.

Due to its length, this example is split into two parts. In a working program, both parts would be placed in the same file. The first part includes the proper header files, declares two file static variables, and contains the definition of the error handling routine.

```
    /* ex4.c */
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
1   #include <setjmp.h>
2   #include "matlab.h"

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

    /* User-defined error handling routine. */
5   void ErrorHandler(const char* msg, bool isError)
    {
        if (isError)
        {
            mlfPrintf("ERROR: %s\n", msg);
6           longjmp(env, -1);
        }
        else /* just a warning */
        {
            mlfPrintf("WARNING: %s\n", msg);
        }
    }
```

Notes

1 Include <setjmp.h>. This file contains the definition of the type `jmp_buf` and the prototypes for the functions `setjmp()` and `longjmp()`.

2 Include `"matlab.h"`. This file contains the declaration of the `mxArray` data structure and the prototypes for the functions in the library. `stdlib.h` contains the definition of `EXIT_SUCCESS`.

3 Declare an array that will be used in the main program to initialize two matrices. Arrange the elements of the C array in column-major order.

4 Declare the static `jmp_buf` variable, `env`. `setjmp()` will store various types of system-specific data in `env`. `longjmp()` will use the data to "return" to the point where `setjmp()` was invoked.

5 Define the error handler. If the argument `isError` is true, `ErrorHandler` calls the print handler to display the string contained in `msg` and then calls `longjmp()`. If `isError` is false, a warning is printed and the application continues.

Note that if you do not call `longjmp()` from your error handler, the library will call `exit()` and terminate your application.

6 Call `longjmp()` to transfer control back to the `if/else` statement where `setjmp()` was first called. The second argument to `longjmp()`, `-1`, is the value that `setjmp()` will return. Be sure to make this value nonzero, to distinguish a return induced by `longjmp()` from a normal return.

The second section of code contains the main program. The error in the program occurs in step 7 when two matrices of unequal size are added together.

```
    main()
    {
        /* Create two matrices of different sizes */
1       mxArray *mat0 = mxCreateDoubleMatrix(2, 3, mxREAL);
        mxArray *mat1 = mxCreateDoubleMatrix(3, 2, mxREAL);

        /* These pointers must be declared as volatile
        ** since their values may be changed inside the setjmp block;
        ** we need to access these values if a longjmp occurs.
        */
2       mxArray *volatile mat2 = NULL;
        mxArray *volatile mat3 = NULL;
        mxArray *volatile zero = NULL;

3       memcpy(mxGetPr(mat0), data, 6 * sizeof(double));
        memcpy(mxGetPr(mat1), data, 6 * sizeof(double));

4       if (setjmp(env) == 0)
        {
            /* Set the error handler */
5           mlfSetErrorHandler(ErrorHandler);

            /* Create a scalar matrix */
            zero = mlfScalar(0);

            /* Division by zero will produce a warning */
6           mat2 = mlfRdivide(mat1, zero);

            /* Illegal operation: matrix dimensions not equal */
7           mat3 = mlfPlus(mat0, mat1);
            mlfPrintMatrix(mat3);

            /* Free the matrices */
8           mxDestroyArray(mat2);
            mxDestroyArray(mat3);
            mxDestroyArray(zero);
        }
```

```
9       else
        {
            mlfPrintf("Caught an error! Recovering!\n");

          /* Clean up matrices allocated before the error occurred. */
            if (mat2)
                mxDestroyArray(mat2);
            if (mat3)
                mxDestroyArray(mat3);
            if (zero)
                mxDestroyArray(zero);
        }

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

        return(EXIT_SUCCESS);
    }
```

Notes

1 Create and initialize two matrices. The first matrix has two rows and three columns and the second has three rows and two columns. The same techniques used in previous examples are used here: a call to `mxCreateDoubleMatrix()` creates a matrix and a subsequent call to `memcpy()` initializes the matrix with data.

2 Declare the variables that will be set within the `setjmp` block as volatile. When a variable is declared as volatile, it is not stored in a register. You can, therefore, set a value to the variable inside the `if` block where `setjmp` is called and still retrieve the value if a `longjmp` occurs to the `else` portion of the `if-else` statement.

3 Copy the data into the two matrices using `mxGetPr()` to access the real portion of the matrix.

4 Call `setjmp()` and initialize `env`. `setjmp()` is a special function that has the potential to return more times than it is called. When explicitly called, as in this `if`-statement, it initializes the `jmpbuf` variable `env` and returns a normal status of 0. However, `setjmp()` can also return when it has not been called. Calling the `longjmp()` function (which never returns) causes control to return from a corresponding `setjmp()`. When `setjmp()` returns as a result of a `longjmp()`, `setjmp()` returns a nonzero status.

Error handling with `setjmp()` and `longjmp()` requires an `if`-statement like this one. The first branch (where `setjmp()` returns 0) contains your data-processing code. Control always enters this first branch. The `else` branch contains error handling code. Any call to `longjmp()` that results from an error in the first branch causes `setjmp()` to return again (with a nonzero status) and control to transfer to the error handling code in the `else` branch.

If an error occurs within the first branch of the `if`-statement, the error handler calls `longjmp()`. How does `longjmp()` know where to transfer control? The first call to `setjmp()` "marks" its position in your program. When `longjmp()` is ready to transfer control back to your code, it transfers control to that "mark" as a return from `setjmp()`. While the first call to `setjmp()` returns 0, subsequent calls return the second argument passed to `longjmp()`. An `if-else` statement that tests the return from `setjmp()` can therefore distinguish between a normal return from `setjmp()` and a return that indicates that an error has occurred.

5. Call `mlfSetErrorHandler()` to replace the default error handler provided by the MATLAB C Math Library with the user-defined error handler, `ErrorHandler()`, defined in the first part of this example. Note that any errors that occur prior to the first call to `mlfSetErrorHandler()` still cause the program to exit.

6. Deliberately causes a warning. The library calls the registered error handler, `ErrorHandler()`, with the parameter `isError` set to `FALSE`. After `ErrorHandler()` prints the warning, the program continues.

7. Deliberately causes an error by calling `mlfPlus` with two input matrices of unequal size. `mlfPlus` requires identically sized matrices. When `mlfPlus` detects that its two inputs are of different sizes, it invokes the registered error handler.

8. If all the code in the `if`-block executes without error, the matrices `mat2`, `mat3`, and `zero` are freed.

9 Handle any errors that occur. The error-handling code in this example is quite short and simply displays another error message. Be sure to note how the call to `longjmp()` in the error handler transfers control back to the main routine via a second return from `setjmp()`. The error handler itself does not return, and the program does not terminate.

After printing the error message, "`Caught an error! Recovering!`", free the matrices that may have been allocated in the `setjmp` block before the error occurred. If an error hadn't occurred, the `mxDestroyArray()` statements in the `if`-block would have cleaned up these matrices.

10 Free the matrices , `mat0` and `mat1`, which were used as input arguments to `mlfPlus()` and `mlfRdivide()`.

Output

When run, the program produces this output:

```
WARNING: Divide by zero.
ERROR: Matrix dimensions must agree.
Caught an error! Recovering!
```

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.

Example 5: Saving and Loading Data

This example demonstrates how to use the functions `mlfSave()` and `mlfLoad()`to write your data to a disk file and read it back again. `mlfLoad()` and `mlfSave()` operate on MAT-files, which use a special binary file format that ensures efficient storage and cross-platform portability. MATLAB can read and write MAT-files, too, so you can use `mlfLoad()` and `mlfSave()` to share data with MATLAB applications or with other applications developed with the MATLAB C++ or C Math Library.

The MATLAB C Math Library functions `mlfSave()` and `mlfLoad()` implement the MATLAB `load` and `save` functions. Note, however, that not all the variations of the MATLAB `load` and `save` syntax are implemented for the MATLAB C Math Library. See the section “Using mlfLoad( ) and mlfSave( )” in Chapter 3 for further information on the two functions.

```
   /* ex5.c */

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

   main()
   {
2      mxArray *x, *y, *z, *a, *b, *c;
       mxArray *r1, *r2, *r3;
       mxArray *four = mlfScalar(4);
       mxArray *seven = mlfScalar(7);

3      x = mlfRand(four,four);
       y = mlfMagic(seven);
       z = mlfEig(NULL, x, NULL);

       /* Save (and name) the variables */
4      mlfSave("ex5.mat", "w", "x", x, "y", y, "z", z, NULL);

       /* Load the named variables */
5      mlfLoad("ex5.mat", "x", &a, "y", &b, "z", &c, NULL);
```

```
        /* Check to be sure that the variables are equal */
6       r1 = mlfIsequal(a, x, NULL);
        r2 = mlfIsequal(b, y, NULL);
        r3 = mlfIsequal(c, z, NULL);

7       if (*mxGetPr(r1) == 1.0 &&
            *mxGetPr(r2) == 1.0 &&
            *mxGetPr(r3) == 1.0)
        {
            mlfPrintf("Success: all variables equal.\n");
        }
        else
        {
            mlfPrintf("Failure: loaded values not equal to saved
                       values.\n");
        }

8       mxDestroyArray(four);
        mxDestroyArray(seven);
        mxDestroyArray(x);
        mxDestroyArray(y);
        mxDestroyArray(z);
        mxDestroyArray(a);
        mxDestroyArray(b);
        mxDestroyArray(c);
        mxDestroyArray(r1);
        mxDestroyArray(r2);
        mxDestroyArray(r3);

        return(EXIT_SUCCESS);
    }
```

Notes

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(). r1, r2, and r3 will contain the results from comparing the saved data to the original data.

The C Math Library utility function mlfScalar() is used to initialize 1-by-1 arrays that hold an integer or double value. four and seven point to arrays that are used to initialize data.

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.

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 name (and filename) are not mxArray arguments; you can pass a string directly to mlfSave() and mlfLoad().

5 Load the named variables from the file "ex5.mat". Note that the function mlfLoad() does not follow the standard C Math Library calling convention where output arguments precede input arguments. The output arguments, a, b, and c, are interspersed with the input arguments.

Pass arguments in this order: the filename, then the name/variable pairs themselves, and finally a NULL to terminate the argument list. An important difference between the syntax of mlfLoad() and mlfSave() is the type of the variable portion of each pair. Because you're loading data into a variable, mlfLoad() needs the address of the variable: &a, &b, &c. a, b, and c are output arguments whereas x, y, and z in the mlfSave() call were input arguments.

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(const char *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 scalar `mxArray` containing `TRUE` if the compared arrays are the same type and size, with identical contents.

7 Use `mxGetPr()` to access the value stored in each scalar `mxArray`. If each of the three values is equal to 1 (or `TRUE`), then all variables were equal. The calls to `mxGetPr()` are necessary because C requires that the condition for an `if` statement be a scalar Boolean, not a scalar `mxArray`.

8 Free each of the matrices used in the examples.

Output

When run, the program produces this output:

```
Success: all variables equal.
```

Example 6: Passing Functions As Arguments

This example demonstrates how you work with the C Math Library "function-functions," functions that execute a function that you provide. The C Math Library function presented in this example, `mlfOde23()`, is a function-function. Other function-functions include `mlfFzeros()`, `mlfFmin()`, `mlfFmins()`, `mlfFunm()`, and the other `mlfOde` functions.

In this example, you'll learn:

- How the function-functions use `mlfFeval()`
- How `mlfFeval()` works
- How to extend `mlfFeval()` by writing a "thunk function"

The main program in this example computes the trajectory of the Lorenz equation using the ordinary differential equation solver `mlfOde23()`. 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 to be integrated is `lorenz`.

How function-functions Use mlfFeval( )

A function-function uses `mlfFeval()` to execute the function passed to it. For instance, `mlfOde23()` in this example 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()`.

`mlfFeval()` is in charge of executing any function passed to it. Because these functions take different arbitrary 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()`:

```
mlfFeval(int nlhs, mxArray **plhs, int nrhs, mxArray **prhs,
         char * name);
```

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 this example, 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 retrieved 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

In order 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; 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 correct 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.

This example is longer than the preceding four; because of its length, it 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.

```
   /* ex6.c */

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

②  double SIGMA, RHO, BETA;

③  static mlfFuncTabEnt MFuncTab[] =
   {
       {"lorenz", (mlfFuncp)lorenz, _lorenz_thunk_fcn_ },
       { NULL, NULL, NULL}
   };

④  mxArray *lorenz(mxArray *tm, mxArray *ym)
   {
       mxArray *ypm;
       double *y, *yp;

⑤      ypm = mxCreateDoubleMatrix(3,1,mxREAL);
       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];

       return(ypm);
   }
```

Notes

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 {NULL, NULL, NULL} entry.

   Before you call mlfOde23() in the main program, pass this variable 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. The third argument, the constant mxREAL, specifies that this array has no imaginary part.

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);
   }
```

Notes

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, *ym, *tsm, *ysm;
        mxArray *lorenz_function = mxCreateString("lorenz");
        double tspan[] = { 0.0, 10.0 };
        double y0[] = { 10.0, 10.0, 10.0 };
        double *t, *y1, *y2, *y3;
        int k, n;

2       mlfFevalTableSetup ( MFuncTab );

3       SIGMA = 10.0;
        RHO = 28.0;
        BETA = 8.0/3.0;

4       tsm = mxCreateDoubleMatrix(2, 1, mxREAL);
        ysm = mxCreateDoubleMatrix(1, 3, mxREAL);

5       memcpy(mxGetPr(tsm), tspan, sizeof(tspan));
        memcpy(mxGetPr(ysm), y0, sizeof(y0));

6       tm = mlfOde23(&ym, NULL, NULL, NULL, NULL, lorenz_function,
                      tsm, ysm, NULL, NULL);

7       n = mxGetM(tm);
        t = mxGetPr(tm);
        y1 = mxGetPr(ym);
        y2 = y1 + n;
        y3 = y2 + n;

8       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]);
        }
```

```
    /* Free the matrices. */
9   mxDestroyArray(tsm);
    mxDestroyArray(ysm);
    mxDestroyArray(tm);
    mxDestroyArray(ym);
    mxDestroyArray(lorenz_function);

    return(EXIT_SUCCESS);
}
```

Notes

1 Declare and initialize variables. `lorenz_function` stores the name of the function to be integrated. `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`. Note that the MATLAB C Math Library requires that you assign the function name to an `mxArray` before you pass it to `mlfOde23()`.

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.

5 Initialize `tsm` to the values stored in `tspan`. Initialize `ysm` to the values stored in `y0`.

6 Call the library routine `mlfOde23()`. The return value and the first argument store results. Pass the name of the function and the two required input arguments. You must pass NULL arguments to a MATLAB C Math Library function whenever you do not supply the value of an optional input or output argument.

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.

7 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.

8 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.

9 Free all arrays and exit. Failure to free these arrays causes a memory leak.

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
```

3

Using the Library

This chapter describes the technical details of the MATLAB C Math Library. It serves more as a reference guide than a tutorial. Be sure to read the section “Calling Conventions” on page 3-3; otherwise, read only those sections that interest you.

This chapter explains how to:

- Call the C Math Library functions
- Use the indexing functions to index into an array
- Use the `mlfLoad()` and `mlfSave()` functions
- Write your own print handler for output to a GUI
- Provide your own memory management routines
- Handle the errors generated by the library
- Reduce the size of the MATLAB M-File Math Library

Calling Conventions

The MATLAB C Math Library includes over 350 functions. Every routine in the MATLAB C Math 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 C call.

You'll find a complete reference for the library functions in the online *MATLAB C Math Library Reference* accessible from the Help Desk. That reference lists the arguments and return value for each function, shows you how to call each version of a function, and lets you access the documentation for the MATLAB version of the function.

How to Call Functions

The following sections use the `mlfCos()`, `mlfTril()`, `mlfFind()`, and `mlfSvd()` functions to demonstrate how to translate a MATLAB call to a function into a MATLAB C Math Library call. Each of the functions demonstrates a different aspect of the calling conventions, including what data type to use for C input and output arguments, how to handle optional arguments, and how to handle MATLAB's multiple output values in C.

One Output Argument, Required Input Arguments

For many functions in the MATLAB C Math Library, the translation from interpreted MATLAB to C is very simple. 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:

```
Y = mlfCos(X);
```

where both `X` and `Y` are pointers to `mxArray` structures.

Optional Input Arguments

Some MATLAB functions take optional input and output 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.

Since C does not permit the simultaneous coexistence of 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 or output 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:

```
L = mlfTril(X,NULL);
```

and

```
L = mlfTril(X,k);
```

where `L`, `X`, and `k` are pointers to `mxArray` structures.

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:

```
k = mlfFind(NULL,NULL,X);
i = mlfFind(&j,NULL,X);
i = mlfFind(&j,&v,X);
```

where `i`, `j`, `k`, `v`, and `X` are `mxArray*` variables.

The general rule for multiple output arguments is: 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. Consider the MATLAB function `svd`. The `svd` reference page begins like this:

Purpose

Singular value decomposition

Syntax

```
s = svd(X)
[U, S, V] = svd(X)
[U, S, V] = svd(X, 0)
```

The function prototypes given under the Syntax heading are not similar to those in a C language reference guide. Yet they contain enough information to tell you how to call the corresponding MATLAB C Math Library routine, `mlfSvd`, if you know how to interpret them.

The first thing to notice is that the syntax lists three ways to call `svd`. 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.

This translates to C in a straightforward fashion. 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. The mlfSvd() reference page that you find in the online *MATLAB C Math Library Reference* accessible from the Help Desk begins like this:

Purpose

Singular value decomposition

Syntax

```
mxArray *X;
mxArray *Zero = mlfScalar(0);
mxArray *U, *S, *V;

S = mlfSvd(NULL, NULL, X, NULL);
U = mlfSvd(&S, &V, X, NULL);
U = mlfSvd(&S, &V, X, Zero);
```

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 `NULL`s in the argument list. Each `NULL` argument takes the place of an "optional" argument.

Mapping Rules

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, required inputs, optional inputs.

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:

   Map from the 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.

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.

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.

How to Call 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 4-5 lists the MATLAB operators and the corresponding MATLAB C Math Library functions.

Exceptions

mlfLoad( ) and mlfSave( )

The `mlfLoad()` and `mlfSave()` functions do not follow the standard calling conventions for the library. They each take a variable, null-terminated list of arguments. The argument list for each function includes pairs of arguments where the argument representing the name of the variable to be loaded or saved is a `const char *`, rather than an `mxArray *` or an `mxArray **`. In addition, the standard order for output and input arguments in not followed: `mlfLoad()` intersperses input and output arguments.

"Example 5: Saving and Loading Data" in Chapter 2 demonstrates how to call the functions.

mlfFeval( )

`mlfFeval()` is able to execute any function passed to it. Because the functions it executes can take different arbitrary numbers of input and output arguments, `mlfFeval()` uses a nonstandard 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.

"Example 6: Passing Functions As Arguments" in Chapter 2 explains the calling convention in detail.

Functions with Variable, Null-Terminated Argument Lists

A group of functions in the MATLAB C Math Library functions takes a variable number of arguments. You must terminate the argument list with a `NULL` argument.

Refer to the online *MATLAB C Math Library Reference* for the complete syntax of these functions:

```
mlfCat();
mlfChar();
mlfFprintf ();
mlfHorzcat();
mlfIsequal ();
mlfReshape();
mlfSprintf ();
mlfStr2mat();
mlfStrcat();
mlfStrvcat();
mlfVertcat();
```

Indexing and Subscripts

The MATLAB interpreter provides a sophisticated and powerful indexing operator that accesses and modifies multiple array elements. The MATLAB C++ Math Library also supports an indexing operator. The MATLAB C Math Library provides the same indexing functionality as the MATLAB interpreter and the C++ Math Library but through a different mechanism. Instead of an indexing operator, the MATLAB C Math Library provides indexing functions.

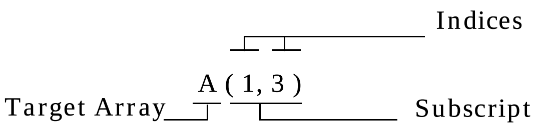
Conceptually, the indexing functions in C are very similar to the indexing operations in MATLAB. In MATLAB, you can access, modify, and delete elements of an array. For example, `A(3,1)` accesses the first element in row three of matrix `A`. In the MATLAB C Math Library, the functions:

- `mlfArrayRef()`
- `mlfArrayAssign()`
- `mlfArrayDelete()`

allow you to do exactly the same thing.

The functions support both one and two-dimensional indexing and follow the MATLAB convention for array indices: indices begin at one rather than zero. Three-dimensional and higher indexing is not supported.

This diagram illustrates the terminology used in this chapter.



The indexing functions apply a subscript to a target array just as the MATLAB syntax in the diagram does. An array subscript consists of one or two indices passed as `mxArray *` arguments to one of the indexing functions. For example, the two-dimensional indexing expression `mlfArrayRef(A, one, three, NULL)` applies the subscript `(1,3)` to `A` and returns the element at row one, column three. `mlfArrayRef(A, nine, NULL)`, a one-dimensional indexing expression, returns the ninth element of array `A`. The arguments `one`, `three`, and `nine` are `mxArray *` variables that each point to a scalar array containing `1`, `3`, and `9` respectively.

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(one, ten, NULL)` specifies the vector `[ 1 2 3 4 5 6 7 8 9 10 ]`. The `one` and `ten` arguments contain scalar arrays.

To modify the data in an array, use the `mlfArrayAssign()` function. For example,

```
mlfArrayAssign(A, fortyfive, three, one, NULL);
```

writes the value `45` into the element at row three, column one of array `A`. If you assign a value to a location that does not exist in the array, the array grows to include that element.

The function `mlfArrayDelete()` removes elements from an array. For example,

```
mlfArrayDelete(A, three, one, NULL);
```

removes the element at row three, column one. Note that removing an element from a matrix reshapes the matrix into a vector.

TIP: `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.

The next sections show you how to:

- Call the indexing functions
- Use two-dimensional, one-dimensional, and logical subscripts
- Make assignments and deletions using indexing

How to Call the Indexing Functions

Using the three indexing functions mlfArrayRef(), mlfArrayAssign(), and mlfArrayDelete() is straightforward once you understand how each forms and applies the subscript. The three functions work in a similar way.

The prototypes for the three functions:

```
mxArray *mlfArrayRef(mxArray *array, ... );

void mlfArrayAssign(mxArray *destination,
                    mxArray *source, mxArray * index1, ...);

void mlfArrayDelete(mxArray *destination, mxArray *index1, ...);
```

Specifying the Target Array

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

- For mlfArrayRef(), supply the array that you want to extract elements from as the first mxArray argument.
- For mlfArrayAssign(), supply the array that you want to change elements of (be assigned to) as the first mxArray argument.
- For mlfArrayDelete(), supply the array that you want to delete elements from as the first mxArray argument.

Specifying the Subscript

The indexing functions apply a subscript to the target array. Each function constructs a subscript from the mxArray arguments that you supply as indices. The functions are defined to accept a variable number of indices. Supply one index mxArray argument to perform one-dimensional indexing. Supply two index mxArray arguments to perform two-dimensional indexing.

- mlfArrayRef() extracts the elements specified by the subscript from the target array and returns the result in a new mxArray. mlfArrayRef() is the only indexing function to return a value.
- mlfArrayAssign() changes the elements in the target array indicated by the subscript. Note that the subscript is applied to the first mxArray argument, the target array, not the second mxArray argument, which is the source array that contains the new values.

- `mlfArrayDelete()` deletes from the target array the elements specified by the subscript.

Specifying a Source Array for Assignments

`mlfArrayAssign()` requires one more argument than the other two indexing functions: a pointer to an `mxArray` that contains the new values for the target array. The function interprets only one subscript; that subscript applies to the target array, not the source array.

Note that `mlfArrayDelete()` does not require a source array. The function assumes that you are applying a null array to the specified elements.

NOTE: To indicate the end of the argument list for each of these functions, supply `NULL` as the last argument. The functions do not follow the standard calling conventions.

The next sections provide information on how the indexing functions work. Refer to the online *C Math Library Function Reference* for more detail on the interface for the three functions.

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.

Scalar `mxArray` variables are named after the number they represent. For example,

```
mxArray *one = mlfScalar(1);
mxArray *two = mlfScalar(2);
```

declares two scalar arrays; `one` is equal to 1 and `two` to 2.

By convention, the pointer to the mxArray that represents the colon operator is called colon and stores the result of a call to mlfCreateColonIndex().

```
mxArray *colon;
colon = mlfCreateColonIndex();
```

The source matrices are created using the mxCreateDoubleMatrix() function. A static array of data is copied into the matrix with the mxGetPr() function. For example, this code creates matrix A:

```
static double A_array_data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
mxArray *A = mxCreateDoubleMatrix(3, 3, 0);
memcpy(mxGetPr(A), A_array_data, 9 * sizeof(double));
```

Matrix A, which is used throughout the examples, is equal to:

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

See “Example 1: Creating Arrays and Array I/O” in Chapter 3 for a complete example of how to use these functions.

Each mxArray must be deleted after the program finishes with it.

```
mxDestroyArray(A);
mxDestroyArray(one);
mxDestroyArray(two);
mxDestroyArray(colon);
```

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. Each argument to these two functions must be a pointer to an existing mxArray.

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

Using mlfArrayRef( ) for Two-Dimensional Indexing

A two-dimensional subscript contains two indices. The first index is the row index; the second is the column index. When you use the MATLAB C Math

Library to perform two-dimensional indexing, you pass `mlfArrayRef()` two index arrays as arguments that together represent the subscript: the first index array argument stores the row index and the second the column index. Each index array can store a scalar, vector, matrix, or the result from a call to the function `mlfCreateColonIndex()`.

The size of the indices rather than the size of the subscripted matrix determines the size of the result; the size of the result is equal to the product of the sizes of the two 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.

The next section describes how to use two-dimensional indices to extract scalars, vectors, and submatrices from a matrix. All examples work with example matrix `A`. "Assumptions for the Code Examples" on page 3-13 explains the conventions used in the examples.

Example Matrix A

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

Selecting a Single Element

Use two scalar indices to extract a single element from an array.

For example,

```
B = mlfArrayRef(A, two, two, NULL);
```

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,

```
mxArray *vector_index, *B;
vector_index = mlfHorzcat(one, three, NULL);
B = mlfArrayRef(A, vector_index, two, NULL);
```

selects the first and third element (or first and third rows) of column two:

```
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):

```
B = mlfArrayRef(A, two, vector_index, NULL);
```

you select the first and third elements (or first and third columns) of row two:

```
2 8
```

If the vector index repeats a number, the same element is extracted multiple times. For example,

```
mxArray *vector_index, *B;
vector_index = mlfHorzcat(three, three, NULL);
B = mlfArrayRef(A, two, vector_index, NULL);
```

returns two copies of the element at `A(2,3)`:

```
8 8
```

Large vectors work just as well as small vectors in these examples. For example, the expression

```
mxArray *vector_index, *B;
vector_index = mlfHorzcat(two, two, two, two, two, NULL);
B = mlfArrayRef(A, two, vector_index, NULL);
```

makes five copies of the element at A(2,2).

NOTE: You can pass mlfHorzcat() or mlfVertcat() any number of arguments. Remember that you cannot nest calls to either function.

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), 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.

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 mlfArrayRef().

This code selects all but the first element in row three, just as

```
A(3, 2:end)
```

does in MATLAB.

```
mxArray *end, *index, *two, *B;
two = mlfScalar(2);
end = mlfEnd(A, two, two);
index = mlfColon(two, end, NULL);
B = mlfArrayRef(A, three, index, NULL);
```

The first argument (two) to mlfEnd() identifies the dimension where mlfEnd() is used, here the column dimension. The second 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,

```
B = mlfArrayRef(A, one, colon, NULL);
```

selects the first row:

```
1 4 7
```

`mlfArrayRef(A, colon, two, NULL)` selects the second column:

```
4
5
6
```

Remember that the variable `colon` points to an `mxArray` created by `mlfCreateColonIndex()` and `one` and `two` point to scalar arrays.

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 two 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,

```
mxArray *row_vector_index, *column_vector_index, *B;

row_vector_index = mlfHorzcat(one, two, NULL);
column_vector_index = mlfHorzcat(one, three, two, NULL);
B = mlfArrayRef(A, row_vector_index, column_vector_index, NULL);
```

selects the first, third, and second (in that order) elements from rows one and two, 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

```
X = mlfArrayRef(A, B, one_two, NULL);
```

performs the same operation as `A(B,[1,2])` in MATLAB and selects the first, second, first, and third elements of columns one and two:

```
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,

```
mxArray *vector_index, *B;
vector_index = mlfHorzcat(two, three, NULL);
B = mlfArrayRef(A, vector_index, colon, NULL);
```

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

```
mxArray *vector_index, *B;
vector_index = mlfHorzcat(three, one, NULL);
B = mlfArrayRef(A, colon, vector_index, NULL);
```

performs the same operation as `A(:,[3,1])` in MATLAB and selects all the elements in columns three and one, 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.

Using mlfArrayRef( ) for One-Dimensional Indexing

A one-dimensional subscript contains a single index. When you use the MATLAB C Math Library to perform one-dimensional indexing, you pass `mlfArrayRef()` 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. For example, a one-dimensional column vector index produces a one-dimensional column vector result.

To apply a one-dimensional subscript to a two-dimensional matrix, you need to know how to go from the one-dimensional index value to a location inside the matrix. A one-dimensional index is like an offset. It tells you how far to count from the beginning of the matrix to reach the element you want.

To count one-dimensionally through a two-dimensional matrix, begin at the first element in the matrix (1,1) and count down the columns until you have counted up to the index value. When you come to the bottom of a column, continue at the top of the next column.

For example, for the 3-by-3 example matrix `A`,

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

the enumeration is:

```
Column 1:    A(1,1)    A(1)
             A(2,1)    A(2)
             A(3,1)    A(3)

Column 2:    A(1,2)    A(4)
             A(2,2)    A(5)
             A(3,2)    A(6)

Column 3:    A(1,3)    A(7)
             A(2,3)    A(8)
             A(3,3)
```

The one-dimensional indexing expression `mlfArrayRef(A, four, NULL)` accesses the first element in the second column, `A(1,2)`. Its value is 4. (The variable `four` is a pointer to an `mxArray` created by `mlfScalar(4)`.)

The elements themselves are visited in this order: `1 2 3 4 5 6 7 8 9`. Note that matrix `A` is specially chosen so that `A(1) = 1`, `A(2) = 2`, and so on.

The formal rule for a one-dimensional scalar index: Given an `M`-by-`N` array `R` and a scalar integer index `X`, the one-dimensional indexing expression `mlfArrayRef(R, X, NULL)` selects the element `R(row, column)`, where `row` equals `rem(X–1,M)+1` and `column` equals `ceil(X/M)`. `rem()` is the remainder function.

NOTE: The range for a one-dimensional index is from 1, the first element of the array, to `M*N`, the last element in an `M`-by-`N` 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.

The following sections demonstrate how to select a single element with a one-dimensional scalar index, a vector with a one-dimensional vector index, a submatrix with a one-dimensional matrix index, and all elements in the matrix with the `colon` index. All examples work with example matrix `A`. "Assumptions for the Code Examples" on page 3-13 explains the conventions used in the examples.

Example Matrix A

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

Notice that 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.

Selecting a Single Element

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

```
B = mlfArrayRef(A, five, NULL);
```

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,

```
mxArray *vector_index, B;
vector_index = mlfHorzcat(two, five, eight, NULL);
B = mlfArrayRef(A, vector_index, 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

```
mxArray *vector_index, B;
vector_index = mlfVertcat(two, five, eight, NULL);
B = mlfArrayRef(A, vector_index, 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.

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), 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.

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 `mlfArrayRef()`.

This code selects all but the first five elements in matrix `A`, just as

```
A(6:end)
```

does in MATLAB.

```
mxArray *end, *index, *one, *two, *B;
one = mlfScalar(1);
two = mlfScalar(2);
six = mlfScalar(6);
end = mlfEnd(A, one, one);
index = mlfColon(six, end, NULL);
B = mlfArrayRef(A, index, NULL);
```

The second argument (`one`) to `mlfEnd()` identifies the dimension where `mlfEnd()` is used, here the row dimension. The third argument (`one`) 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,

```
X = mlfArrayRef(A, B, NULL);
```

is

```
1 2
3 2
```

Note that the example matrix `A` was chosen so that `mlfArrayRef(A,X,NULL)` equals `X` for all types of one-dimensional indexing. This is not generally the case. For example, if `A` were changed to `A = mlfMagic(three);`

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

and `B` remains the same, then `mlfArrayRef(A, B, NULL)` 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,

```
B = mlfArrayRef(A, colon, NULL);
```

is:

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

The `colon` index means “all.” Think of it as 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.

Using mlfArrayRef( ) for Logical Indexing

Logical indexing is a special case of both one- and two-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 the MATLAB C Math Library attaches 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 a two-dimensional logical subscript by combining a logical index with a scalar, vector, matrix, or `colon` index.

Example Matrices A and B

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

`B` (a logical array)

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

Selecting from a Matrix

This section demonstrates several ways to use a logical index when selecting elements from a matrix.

- A one-dimensional matrix index
- A pair of logical vector indices for two-dimensional indexing
- A `colon` index and a logical vector index for two-dimensional indexing

"Assumptions for the Code Examples" on page 3-13 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

```
X = mlfArrayRef(A, B, NULL);
```

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`.

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

```
X = mlfArrayRef(A, B, NULL);
```

equals

```
1
4
```

since `B` has a zero at positions 2 and 3 and 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 though 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,

```
X = mlfArrayRef(A, B, C, NULL);
```

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`: all the elements in the second columns of rows one and three.

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

```
X = mlfArrayRef(A, B, C, NULL);
```

equals

```
4
```

This is tricky. `B`, the row index, selects row one but does not select row two. `C`, the column index, does not select column 1 but does not select column 2. There is only one element in array `A` in both row 1 and column two, 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])` and the `mxArray *` variable `colon` be created by `mlfCreateColonIndex()`. Then

```
X = mlfArrayRef(A, colon, B, NULL);
```

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: the first and third columns of the matrix.

For comparison,

```
X = mlfArrayRef(A, B, colon, NULL);
```

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: the first and third rows of the matrix.

Selecting from a Row or Column

This section demonstrates how to use a logical index to select elements from a row or column.

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:

```
target_row = mlfArrayRef(X, two, colon, NULL);
B = mlfGt(target_row, nine);
```

and contains the vector

```
0 1 1 0
```

In MATLAB, `B = (A(2,:) > 9)` performs the same operation.

Use `B` as a logical index that selects those elements from matrix `X`.

```
C = mlfArrayRef(X, two, B, NULL);
```

selects these elements:

```
11 10
```

Using mlfArrayAssign( ) for Assignments

Use the function `mlfArrayAssign()` to make assignments that involve indexing. The arguments to `mlfArrayAssign()` consist of a destination array, a source array, and one or two index arrays that represent the subscript. The subscript specifies the elements that are to be modified in the destination array; the source array specifies the new values for those elements. The subscript is only applied to the destination array.

You can use five different kinds of indices:

- Scalar
- Vector
- Matrix
- `colon`
- Logical

The examples below do not present all possible combinations of these index types. You are encouraged to experiment with other combinations.

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 work with matrix `A`. "Assumptions for the Code Examples" on page 3-13 explains the conventions used in the examples.

Example Matrix A

```
A =

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

Assigning to a Single Element

Use one or two scalar indices to assign a value to a single element in a matrix. For example,

```
mlfArrayAssign(A, seventeen, two, one, NULL);
```

changes the element at row two and column one to the integer 17. Here, both the source and destination (after the subscript has been applied) are scalars, and thus exactly 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

```
source = mlfColon(one,three,NULL);
mlfArrayAssign(A, source, two, colon, 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

```
logical_index = mlfGt(A,five);
source = mlfHorzcat(seventeen, seventeen, seventeen, seventeen,
    NULL);
mlfArrayAssign(A, source, logical_index, 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 Portion of a Matrix

Use two vector indices to generate a matrix destination. For example, let the vector index `B` equal `1 2`, and the vector index `C` equal `2 3`. Then,

```
source_matrix = mlfVertcat(one_four, three_two, NULL);
mlfArrayAssign(A, source_matrix, B, C, NULL);
```

copies a 2-by-2 matrix into the second and third columns of rows one and two: 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,

```
mlfArrayAssign(A, source_matrix, B, NULL);
```

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

```
source = mlfRand(three, NULL);
mlfArrayAssign(A, source, colon, NULL);
```

writing

```
A = mlfRand(three, 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(three, NULL)` is equivalent to `mlfRand(three, three)`.

Using mlfArrayDelete( ) for Deletion

Use the function `mlfArrayDelete()` 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 an element from example matrix `A`, you simply pass the target array and the indices that identify the elements to be removed. For example,

```
mlfArrayDelete(A, two, three, NULL);
```

deletes the third element in the second row from matrix `A`.

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 `(2,3)` is deleted from matrix `A`, matrix `A` becomes this row vector with element `8` missing:

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

You can also 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,

```
mlfArrayDelete(A, two, colon, NULL);
```

produces this rectangular result:

```
1 4 7
3 6 9
```

NOTE: A two-dimensional subscript can contain only one scalar, vector, or matrix index. The other index used in deletion operations must be a `colon` index.

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 3-13 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 3-1: MATLAB/C Indexing Expression Equivalence

Description	MATLAB Expression	C Expression	Result
Extract 1,1 element	`X(1,1)`	`mlfArrayRef( X, one, one, NULL )`	4
Extract 1st element	`X(1)`	`mlfArrayRef( X, one, NULL )`	4
Extract 3rd element	`X(3)`	`mlfArrayRef( X, three, NULL )`	5

Table 3-1: MATLAB/C Indexing Expression Equivalence (Continued)

Description	MATLAB Expression	C Expression	Result

Extract all elements into column vector	`X(:)`	`mlfArrayRef(`
		`X,`
		`colon,`
		`NULL`
		`)`
			4
			6
			5
			7

Extract 1st row	`X(1,:)`	`mlfArrayRef(`
		`X,`
		`one,`
		`colon,`
		`NULL`
		`)`
			4 5

Extract 2nd row	`X(2,:)`	`mlfArrayRef(`
		`X,`
		`two,`
		`colon,`
		`NULL`
		`)`
			6 7

Extract first column	`X(:,1)`	`mlfArrayRef(`
		`X,`
		`colon,`
		`one,`
		`NULL`
		`)`
			4
			6

Extract second column	`X(:,2)`	`mlfArrayRef(`
		`X,`
		`colon,`
		`two,`
		`NULL`
		`)`
			5
			7

Table 3-1: MATLAB/C Indexing Expression Equivalence (Continued)

Description	MATLAB Expression	C Expression	Result

Replace first element with 9	`X(1) = 9`	`mlfArrayAssign(`
		`X,`
		`nine,`
		`one,`
		`NULL`
		`);`
			9 5
			6 7

Replace first row with [11 12]	`X(1,:) = [ 11 12 ]`	`mlfArrayAssign(`
		`X,`
		`eleven_twelve,`
		`one,`
		`colon,`
		`NULL`
		`);`
			11 12
			6 7

Replace element 2,1 with 9	`X(2,1) = 9`	`mlfArrayAssign(`
		`X,`
		`nine,`
		`two,`
		`one,`
		`NULL`
		`);`
			4 5
			9 7

Replace elements 1 and 4 with 8 (one-dimensional indexing)	`X([1 4]) = [ 8 8 ]`	`mlfArrayAssign(`
		`X,`
		`eight_eight,`
		`one_four,`
		`NULL`
		`);`
			8 5
			6 8

Print Handlers

Back in the days 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 and windowed desktops 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. In order 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 *) );
```

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 three sections illustrate how to provide an alternate print handler under the X Window System, Microsoft Windows, and the Macintosh.

Each example demonstrates the interface between the MATLAB C Math Library and one of the windowing systems. 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.

X Windows / Motif Example

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 labelled **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.

Microsoft Windows Example

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.

Apple Macintosh Example

The Macintosh does not provide a widget or message box similar to the MessageDialog widget provided by Motif or the MessageBox dialog box available with Microsoft Windows, making this example more complex than the examples for the other two systems. This example is divided into two parts; in an actual program both parts would be stored in the same file.

The first part includes the proper header files, declares two static variables, and then declares a function, `InitMessageBox`, that uses Apple's QuickDraw to set up a message box one-quarter the size of the screen. This message box is used repeatedly by the print handling routine. The message box is actually an instance of a TextEdit object, one of the simple objects that is built into the Macintosh operating system.

```
/* Macintosh example */

#include <Windows.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <TextEdit.h>
#include <stdio.h>

static WindowPtr theWindow = NULL;
static TEHandle hTE = NULL;

void InitMessageBox()
{
    Rect boundsRect;

    boundsRect = qd.screenBits.bounds;
    InsetRect(&boundsRect,
             (boundsRect.right–boundsRect.left) / 4,
             (boundsRect.bottom–boundsRect.top) / 4);
    theWindow = NewWindow(NULL, &boundsRect, "\pSimple Output",
                          true, dBoxProc, (WindowPtr) –1,
                          false, 0);
    SetPort(theWindow);
    boundsRect.bottom –= boundsRect.top;
    boundsRect.right –= boundsRect.left;
    boundsRect.top = 0; boundsRect.left = 0;

    hTE = TENew(&boundsRect, &boundsRect);
}
```

The second part of the Macintosh example is very similar to both the X Window System and Microsoft Windows examples. It declares a print handler function called `PopupMessageBox`, which writes the message text into the text edit window created by `InitMessageBox`. When the program terminates, the function `CloseSimpleOutput` cleans up the resources allocated to the text edit window. Finally, the main routine starts up the application, calls

`InitMessageBox` to create the text edit window, and then sets up the print handler.

```
/* the alternate print handler */
void PopupMessageBox(const char *text)
{
    TEInsert((Ptr) text, strlen(text), hTE);
}

void CloseSimpleOutput(void)
{
    TEDispose(hTE);
    DisposeWindow(theWindow);
}

void main()
{
    /* Mac-specific startup code. Be sure to initialize */
    /* QuickDraw, FontManager, WindowManager and TextEdit */

    /* Set the print handler */
    InitMessageBox();
    mlfSetPrintHandler(PopupMessageBox);

    /* Do some actual work... */

    /* Clean up - call this in error handler too. */
    CloseSimpleOutput();
}
```

As is, this is not a complete program, but it should serve as enough of an example to get you started.

Using mlfLoad( ) and mlfSave( )

The MATLAB C Math Library provides two functions, `mlfLoad()` and `mlfSave()`, which let you import and export array data. `mlfSave()` writes variables to a MAT-file as named variables; `mlfLoad()` reads variables back in. Since MATLAB also reads and writes MAT-files, you can use `mlfLoad()` and `mlfSave()` to share data with MATLAB applications or with other applications developed with the MATLAB C++ or C Math Library.

`mlfLoad()` and `mlfSave()` operate on MAT-files. A MAT-file is a binary, machine-dependent file. However, it can be transported between machines because of a machine signature in its file header. The MATLAB C Math Library checks the signature when it loads variables from a MAT-file and, if a signature indicates that a file is foreign, performs the necessary conversion.

mlfSave( )

Using `mlfSave()`, you can save the data within `mxArray` variables to disk. The prototype for `mlfSave()` is:

```
void mlfSave(const char *file, const char *mode, ...);
```

`file` points to the name of the MAT-file; `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.

mlfLoad( )

Using `mlfLoad()`, you can read in `mxArray` data from a binary MAT-file. The prototype for `mlfLoad()`:

```
void mlfLoad(const char *file, ...);
```

`file` points to the name of the MAT-file; 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*.

Memory Management

Routines in the MATLAB C Math Library allocate new arrays for their return values and for each of their output arguments. For example, when you call the version of the function `mlfSvd()` that takes two output arguments and one input argument:

```
U = mlfSvd(&S,&V,X,NULL);
```

where `U`, `S`, `V`, and `X` are all declared as `mxArray*`, the library allocates new arrays for `U`, `S`, and `V`.

When you are finished using the arrays that the library creates for you, you must call `mxDestroyArray()` to free each array. In the `mlfSvd()` example, you must make three calls to `mxDestroyArray()`.

```
mxDestroyArray(U);
mxDestroyArray(S);
mxDestroyArray(V);
```

You must also free the arrays returned from any `mx` Application Program Interface Library routine that you call, for example, `mxCreateDoubleMatrix()` or `mxCreateString()`. If the input array `X` in the `mlfSvd()` example were created with a call to `mxCreateDoubleMatrix()`, you would need to free `X`, too, when you are finished using it:

```
mxDestroyArray(X);
```

NOTE: If you do not free the arrays that have been allocated by the MATLAB C Math Library, your application will leak memory. If your program runs long enough, or manipulates large arrays, it will eventually run out of memory. In addition, you should not nest calls to library functions.

Setting Up Your Own Memory Management

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():

1. 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.

2. 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.

3. 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.

4 `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.

Error Handling

Errors encountered by the MATLAB C Math Library result in error messages, which need to be made visible to the user. By default, the error handling mechanism calls the print handler to display the message and then calls `exit`. See the section "Print Handlers" on page 3-37 for details on print handling.

The default scheme makes two assumptions: first, that errors and other messages may properly appear in the same place; and second, that it is appropriate for the program to exit when an error occurs. In some cases, one or both of these assumptions may be wrong. Therefore, the MATLAB C Math Library provides the function `mlfSetErrorHandler()` that allows you to control how errors are displayed and handled.

There is an important difference between the print and error handlers. The print handler always returns control to the program that invoked it. When the MATLAB C Math Library issues an error, it does not expect the error handler to return. It expects your error handler to call `exit()` or the function `longjmp()`.

Therefore, an error handler that you write should perform the following tasks:

- Print the error message, possibly by calling the print handling routine.
- Perform any necessary clean-up, for example, of data structures.
- Terminate the program, or call `longjmp()`.

  Note that if your error handler returns without calling `exit()` or `longjmp()`, the MATLAB C Math Library will call `exit()`.
- Print the messages that are warnings rather than errors.

The error handling routine takes two arguments, a single `const` string that contains the error message and a Boolean value indicating whether the message is an error or a warning. The error handling routine returns `void`. Any error handler that you write must have the same prototype.

Here is the code for the default error handler. Notice that the default error handler does not call exit() or longjmp(); the C Math Library will, therefore, call exit().

```
void DefaultErrorHandler(const char* msg, bool isError)
{
    char buf[MAXERRLEN + 12];
    if (!isError) {
        sprintf(buf, "WARNING: %s\n", msg);
    } else {
        sprintf(buf, "ERROR: %s\n", msg);
    }
    print_handler(buf);
}
```

The section “Print Handlers” on page 3-37 includes detailed examples that demonstrate how to set the print handling function. The examples in that section demonstrate how to bring up messages in dialog boxes on the X Window System, Microsoft Windows, and Macintosh systems. You can easily adapt those examples to the error handling mechanism.

If you want to write an error handler that does not cause the termination of your program every time an error occurs, you need to use the system calls setjmp() and longjmp(). For an example of how to write an error handler that uses setjmp() and longjmp(), see “Example 4: Handling Errors” on page 2-16. A detailed explanation of how these two functions work is beyond the scope of this book. For further information on the use of setjmp() and longjmp(), refer to your system documentation.

Using mlfSetErrorHandler( )

This example redirects errors to a different location than other messages. Suppose you want to direct all error messages to a file. This strategy allows a program to run unattended, since any errors produced are recorded in a file for future examination. The simplest way to log error messages in a file is to modify the print handler to send its messages to a file. The error handler may be left untouched. However, to illustrate the use of mlfSetErrorHandler(), this example includes an error handler that prints Fatal Error! before it prints the error message itself.

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. Only the first example is presented here.

```
#include <stdio.h>
#include "matlab.h" /* Include MATLAB C Math Library prototypes */

static FILE *fp = 0; /* Pointer to message file */
static char[] message_file = "message.txt"; /* Msg. file name */

void PrintHandler(const char *message)
{
    /* Make sure file is open, then print to it */
    if (fp)
        fprintf(fp, message);
}

/* Use the PrintHandler to "display" error messages */
void ErrorHandler(const char *message, bool isError)
{
    if (isError)
    {
        PrintHandler("Fatal Error!\n");
        PrintHandler(message);
        exit(-1); /* exit() will close and flush files */
    }
    else /* just a warning */
    {
        PrintHandler("WARNING: %s\n", message);
    }
}
```

```
main()
{
    fp = fopen(message_file, "w");
    if (!fp) /* Can't use PrintHandler here... */
    {
        printf("Can't open file %s, check permissions.\n",
                message_file);
        exit(-1);
    }
    mlfSetPrintHandler(PrintHandler);
    mlfSetErrorHandler(ErrorHandler);
    /* Do some work */
}
```

This example is quite short. First, the new print handler and error handler are defined. The print handler uses `fprintf` to send its output to a file, checking first to make sure the file is open. When an error occurs, the error handler calls the print handler to display introductory text (`Fatal Error!`) and then calls the print handler again with the text of the error message. Finally, the error handler calls `exit`, which flushes and closes any open files and then terminates the program. If a warning occurs, the error handler calls the print handler to display the warning and does not terminate the program.

The `main` routine is a skeleton; it opens the output file and sets up the print and error handlers by calling `mlfSetPrintHandler()` and `mlfSetErrorHandler()`. If this were an actual application, `main` would also contain code to call other routines or perform calculations of its own.

Performance and Efficiency

The MATLAB C Math Library is delivered as a set of shared libraries (or DLLs). In general, library size is only a problem for shared libraries on machines with small amounts of physical memory. In contrast to static libraries, most shared libraries are loaded in their entirety when a user program references a routine in the library.

In some cases, the size of the Math M-File Library (`libmmfile`) may exceed your needs. `libmmfile` contains the compiled versions of every math M-file included in the MATLAB C Math Library. For example, `rank`, `gradient`, and `hadamard` are all implemented as M-files and are therefore part of `libmmfile`. Since the average application calls only a small subset of the routines in `libmmfile`, dynamically linking against the entire M-File Library typically uses excess memory.

An alternative to dynamically linking against the entire `libmmfile` is to use the MATLAB Compiler to compile only those function M-files that your application references.

NOTE: In order to use the MATLAB Compiler to compile M-File Library files, you must own both MATLAB and the MATLAB Compiler.

Reducing Memory

To compile only the function M-files that you require:

1. Add the MATLAB Compiler-compatible M-files directory to your path.

   The directories containing these M-files are:

   On UNIX, `<matlab>/extern/src/math/tbxsrc`

   On Windows, `<matlab>\extern\src\math\tbxsrc`

   On Macintosh, `<matlab>:extern:src:math:tbxsrc:`

2. Compile each M-file that you need with the MATLAB Compiler.

3 Edit your `mbuild` options file so that it does not link with `libmmfile` but does link with the files you just compiled.

   Note that you will receive link errors if you have not compiled and linked in all the functions that you are using from `libmmfile`.

4 Run `mbuild` to build and link your application.

Your program may now make calls to the M-file functions that you compiled and statically linked with your application without dynamically linking to the entire MATLAB Math M-File Library.

NOTE: You still need to link against the MATLAB Built-In Library (`libmatlb`) and the other supporting libraries: `libmcc`, `libmx`, and `libmat`.

For more information on the MATLAB Compiler, refer to the *MATLAB Compiler User's Guide*.

4

Library Routines

This chapter serves as a reference guide to the more than 350 functions contained in the MATLAB C Math Library.

The functions are divided into three sections:

- The Built-In Library
- The M-File Math Library
- The Application Program Interface Library

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.

Why Two 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 considerably larger than the Built-In Library and contains functions that not every program needs, 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.

We divided the MATLAB functions into two shared libraries, or DLLs, to help you write more space-efficient programs. In general, shared library size is a problem only on machines with small amounts of physical memory. In contrast to static libraries, most shared libraries are loaded in their entirety when a user program calls a routine in the library.

Most MATLAB C Math Library programs link dynamically against both math libraries, in addition to the Application Program Interface Library. (See “Building C Applications” in Chapter 1 for a complete list of the required libraries.) If you find that the size of the shared M-File Math Library is impairing your machine’s performance, you can use the MATLAB Compiler to compile only the M-file routines that you need and then statically link your application against this smaller set. The section entitled “Performance and Efficiency” in Chapter 3 provides more details on how to reduce the size of the M-File Math Library.

NOTE You always need to link dynamically against the MATLAB Built-In Library. There is no way to reduce its size.

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 MATLAB built-in function is named after its MATLAB equivalent. For example, the `mlfTan` function is the C callable version of the MATLAB built-in `tan` function.

- C function 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.

Set Operators (Continued)

Function	Purpose
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.

Logical Functions (Continued)

Function	Purpose
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.

MATLAB as a Programming Language

Function	Purpose
mlfFeval	Function evaluation.
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 (always 2).
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.

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.
mlfStrncmp	Compare the first n characters of two strings.
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`	Is file position indicator at the end of the 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.

Utility Routines

The C Math Library utility routines help you perform indexing, create scalar arrays, and initialize and control the library environment. Note that these functions are covered in more detail in Chapter 2 and in Chapter 3.

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);`	Registers a thunk function table with the MATLAB C Math Library.

Indexing

Function	Purpose
`void mlfArrayAssign(mxArray *destination, mxArray *source, ... );`	Handle assignments that include indexing.
`void mlfArrayDelete(mxArray *destination, mxArray *index1, ... );`	Handle deletions that include indexing.

Indexing (Continued)

Function	Purpose
`mxArray * mlfArrayRef(mxArray *array, ... );`	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()`.
`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.
`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.

Logical Functions

Function	Purpose
`mlfIsieee`	True for IEEE floating point arithmetic.
`mlfIsspace`	True for whitespace characters in string matrices.
`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.

Matrix Manipulation (Continued)

Function	Purpose
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	Non-negative 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.

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.
mlfStrcmp	Compare strings.
mlfStrjust	Justify a character array.
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.
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.

Application Program Interface Library

The Application Program Interface 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 Application Program Interface Library routine by its prefix mx. These functions are a subset of the Application Program Interface Library. In the MATLAB C Math Library, these functions support arrays with at most two dimensions.

Array Access Routines

Function	Purpose
mxCalloc, mxFree	Allocate and free dynamic memory using MATLAB's memory manager.
mxClearLogical	Clear the logical flag.
mxCreateCharArray	Create an unpopulated N-dimensional string mxArray.
mxCreateCharMatrixFromStrings	Create a populated 2-dimensional string mxArray.
mxCreateDoubleMatrix	Create an unpopulated 2-dimensional, double-precision, floating-point mxArray.
mxCreateNumericArray	Create an unpopulated N-dimensional numeric mxArray.
mxCreateString	Create a 1-by-n string mxArray initialized to the specified string.
mxDestroyArray	Free dynamic memory allocated by an mxCreate routine.
mxDuplicateArray	Make a deep copy of an array.
mxGetClassID	Get (as an enumerated constant) an mxArray's class.

Array Access Routines (Continued)

Function	Purpose
`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.
`mxGetImagData`	Get pointer to imaginary data of an `mxArray`.
`mxGetInf`	Get the value of infinity.
`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.
`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.

Array Access Routines (Continued)

Function	Purpose
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.
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.
mxIsUint8	True if mxArray represents its data as unsigned 8-bit integers.
mxIsUint16	True if mxArray represents its data as unsigned 16-bit integers.
mxIsUint32	True if mxArray represents its data as unsigned 32-bit integers.
mxMalloc	Allocate dynamic memory using MATLAB's memory manager.
mxRealloc	Reallocate memory.
mxSetData	Set pointer to data.

Array Access Routines (Continued)

Function	Purpose
`mxSetDimensions`	Modify the number of dimensions and/or the size of each dimension.
`mxSetImagData`	Set imaginary data pointer for an `mxArray`.
`mxSetLogical`	Set the logical flag.
`mxSetM`, `mxSetN`	Set the number of rows (M) and columns (N) of an array.
`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.

5

Directory Organization

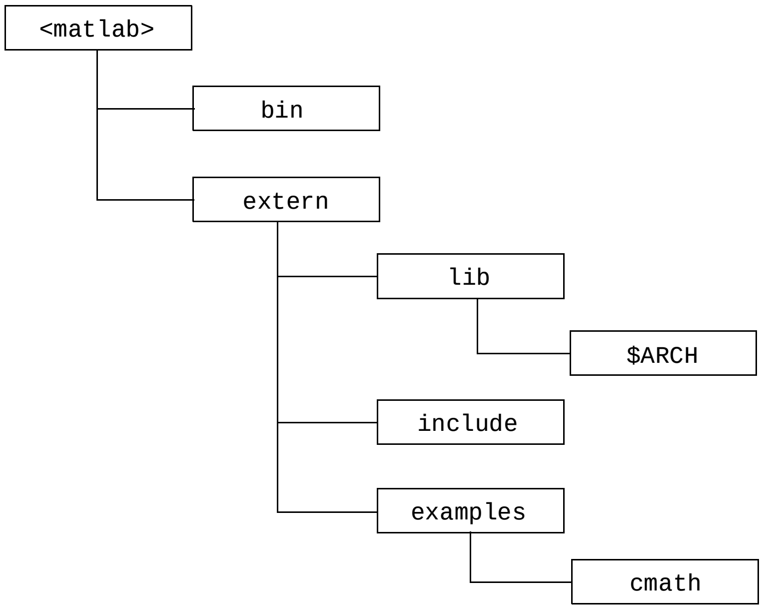
This chapter describes the directory organization of the MATLAB C Math Library on UNIX, Microsoft Windows, and Macintosh systems.

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: the MATLAB root directory.

Separate directories for the major product categories are located under the MATLAB root. The C Math Library is installed in the `extern` directory where products external to MATLAB are installed, and on UNIX and Microsoft Windows systems, in the `bin` directory. If you have other MathWorks products, there are additional directories directly below the MATLAB 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 home 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 SunOS4, the `$ARCH` directory is `sun4`. The libraries that come with the MATLAB C Math Library are:

`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.
`libmcc.`*ext*	MATLAB Compiler Library for stand-alone applications. Contains the `mcc` and `mcm` routines required for building stand-alone applications.
`libmi.`*ext*	Internal math 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 Application Program Interface Library. Contains array access routines.
`libut.`*ext*	MATLAB Utilities Library. Contains the utility routines used in the background by various components.

where .*ext* is

`.a` on IBM RS/6000 and Sun4; `.so` on Solaris, Alpha, Linux, and SGI; and `.sl` on HP 700. The libraries are shared libraries for all platforms except Sun4.

<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:

`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.

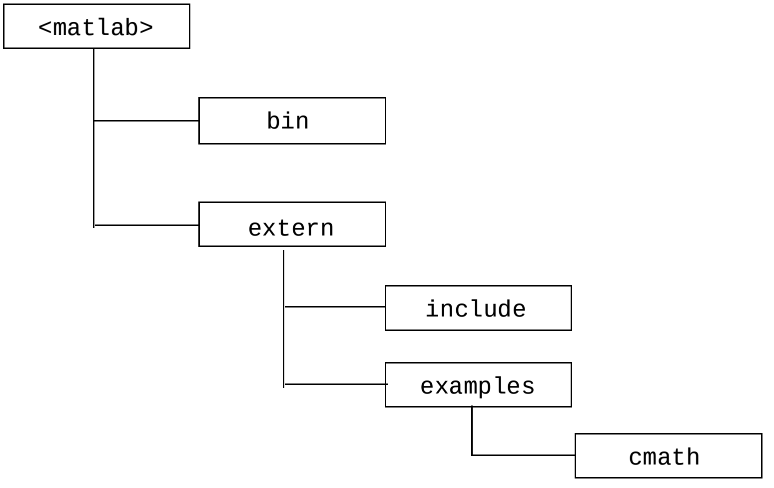
<matlab>/extern/examples/cmath

The `<matlab>/extern/examples/cmath` directory contains the sample C programs that are described in Chapter 2.

`ex1.c`	The source code for "Example 1: Creating and Printing Arrays" on page 2-6.
`ex2.c`	The source code for "Example 2: Writing Simple Functions" on page 2-9.
`ex3.c`	The source code for "Example 3: Calling Library Routines" on page 2-12.
`ex4.c`	The source code for "Example 4: Handling Errors" on page 2-16.
`ex5.c`	The source code for "Example 5: Saving and Loading Data" on page 2-22.
`ex6.c`	The source code for "Example 6: Passing Functions As Arguments" on page 2-26.
`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.
libmcc.dll	MATLAB Compiler Library for stand-alone applications. Contains the mcc and mcm routines required for building stand-alone applications.
libmi.dll	Internal math 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 Application Program Interface Library. Contains array 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.

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.
libmcc.def _libmcc.def	Contains names of functions exported from the MATLAB Compiler Library DLL for stand-alone applications.
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.
libut.def _libut.def	Contains names of functions exported from libut.dll.

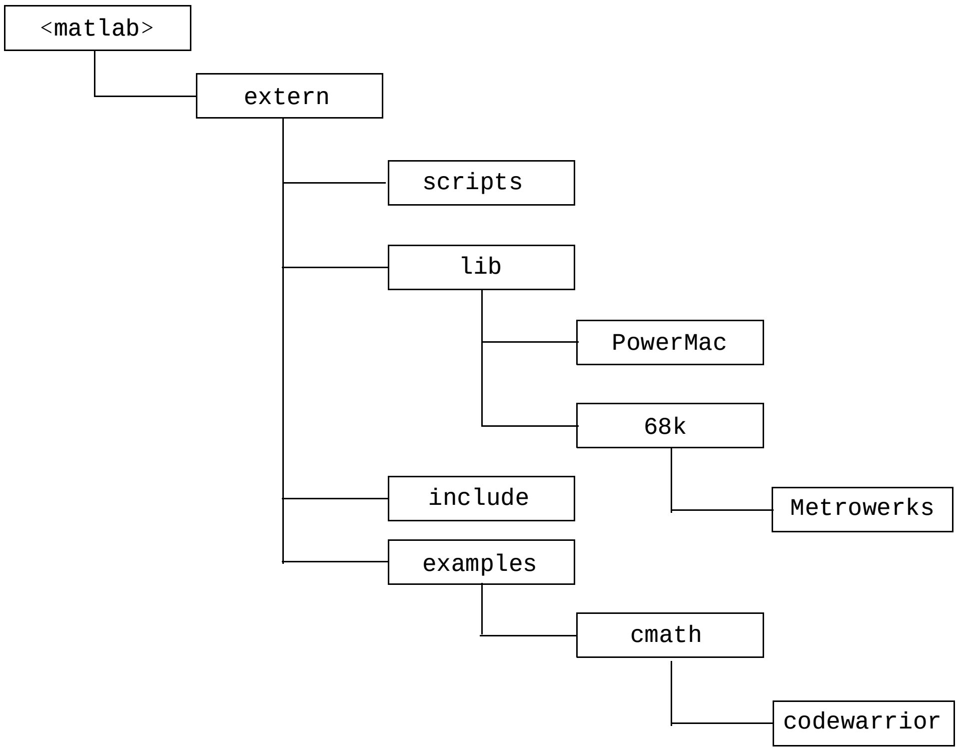
<matlab>\extern\examples\cmath

The `<matlab>\extern\examples\cmath` directory contains sample C programs developed with the C Math Library. You'll find explanations for these examples in Chapter 2.

`ex1.c`	The source code for "Example 1: Creating and Printing Arrays" on page 2-6.
`ex2.c`	The source code for "Example 2: Writing Simple Functions" on page 2-9.
`ex3.c`	The source code for "Example 3: Calling Library Routines" on page 2-12.
`ex4.c`	The source code for "Example 4: Handling Errors" on page 2-16.
`ex5.c`	The source code for "Example 5: Saving and Loading Data" on page 2-22.
`ex6.c`	The source code for "Example 6: Passing Functions As Arguments" on page 2-26.
`release.txt`	Release notes for the current release of the MATLAB C Math Library.

Directory Organization on Macintosh

This figure illustrates the folders that contain the files of the MATLAB C Math Library. `<matlab>` symbolizes the top-level folder where MATLAB is installed on your system.



<matlab>:extern:scripts:

The `<matlab>:extern:scripts:` folder contains:

`mbuild`	Script that helps you build and link stand-alone executables.
Various `mbuildopts.*` files	Options files that control the switches and options for your C compiler. They are architecture specific. When you execute `mbuild -setup`, a copy of one of these files is made in the scripts directory.

<matlab>:extern:lib:PowerMac:

The `<matlab>:extern:lib:PowerMac:` folder contains the required libraries for MPW and Metrowerks programmers.

`libmat`	MAT-file access routines to support `mlfLoad()` and `mlfSave()`. This is a shared library.
`libmatlb`	MATLAB Math Built-In Library. Contains stand-alone versions of MATLAB built-in math functions and operators. This is a shared library.
`libmcc`	MATLAB Compiler Library for stand-alone applications. Contains the `mcc` and `mcm` routines required for building stand-alone applications. This is a shared library.
`libmi`	Internal math routines.
`libmmfile`	MATLAB M-File Math Library. Contains stand-alone versions of the MATLAB math M-files . This is a shared library.
`libmx`	MATLAB Application Program Interface Library. Contains array access routines. This is a shared library.
`libut`	MATLAB Utilities Library. Contains the utility routines used in the background by various components. This is a shared library.

<matlab>:extern:lib:68k:Metrowerks:

The `<matlab>:extern:lib:68k:Metrowerks:` folder contains the required libraries for Metrowerks programmers working on Motorola 680x0 platforms. These libraries are static libraries.

`libmat.lib`	MAT-file access routines to support `mlfLoad` and `mlfSave`.
`libmatlb.lib`	MATLAB Math Built-In Library. Contains stand-alone versions of MATLAB built-in math functions and operators. This is a shared library.
`libmcc.lib`	MATLAB Compiler Library for stand-alone applications. Contains the `mcc` and `mcm` routines required for building stand-alone applications.
`libmi.lib`	Internal math routines.
`libmmfile.lib`	MATLAB M-File Math Library. Contains stand-alone versions of the MATLAB math M-files. This is a shared library.
`libmx.lib`	MATLAB Application Program Interface Library. Contains array access routines. This is a shared library.
`libut.lib`	MATLAB Utilities Library. Contains the utility routines used in the background by various components. This is a shared library.

<matlab>:extern:include:

The `<matlab>:extern:include:` folder contains the header files for developing C applications. The header files associated with the MATLAB C Math Library are:

`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.

<matlab>:extern:examples:cmath:

The `<matlab>:extern:examples:cmath:` folder contains the sample C programs described in Chapter 2.

`ex1.c`	The source code for “Example 1: Creating and Printing Arrays” on page 2-6.
`ex2.c`	The source code for “Example 2: Writing Simple Functions” on page 2-9.
`ex3.c`	The source code for “Example 3: Calling Library Routines” on page 2-12.
`ex4.c`	The source code for “Example 4: Handling Errors” on page 2-16.
`ex5.c`	The source code for “Example 5: Saving and Loading Data” on page 2-22.
`ex6.c`	The source code for “Example 6: Passing Functions As Arguments” on page 2-26.
`release.txt`	Release notes for the current release of the MATLAB C Math Library.

<matlab>:extern:examples:cmath:codewarrior:

The `<matlab>:extern:examples:cmath:codewarrior` folder contains project for the CodeWarrior compiler.

ex*_CW_PPC.proj	CodeWarrior 10 and 11 project files for each of the MATLAB C Math Library examples. For use on Power Macintosh.
ex*_CW_68K.proj	CodeWarrior 10 and 11 project files for each of the MATLAB C Math Library examples. For use on Motorola 680x0 platforms.
ex*_CWPRO_PPC.proj	CodeWarrior PRO (12) project files for each of the MATLAB C Math Library examples. For use on Power Macintosh.

Errors and Warnings

This section lists the a subset of the error and warning messages issued by the MATLAB C Math Library. Each type of message is treated in its own section. Within each section the messages are listed in alphabetical order. Following each message is a short interpretation of the message and, where applicable, suggested ways to work around the error.

Errors

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.

`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 x and x'Ax = 0 only when x = 0. 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 = +/– 1.

`Singularity in TAN`

A singularity indicates an input for which the output is undefined. TAN (tangent) has singularities at odd multiples of ¼/2.

`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).

Warnings

All warnings begin with the string `Warning:`. 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 section Errors 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

C

D

N